بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيم

USE OF MACHINE LEARNING TECHNIQUES IN THE LEARNING AND PREDICTION OF ALPHA-DECAY HALF-LIVES

COMPUTER SCIENCE NEA 2021/22

MUHAMMAD AMJAD
SPALDING GRAMMAR SCHOOL

Table of Contents

Analysis

Identification of the Problem

Initial Ideas

Over the course of the A-Level, I have taken a deeper interest in the creation of AI using Machine Learning techniques, creating a neural network which feeds forward information into a number of hidden layers, training itself by minimising a cost function using backpropagation. I wished to pair this with my love for Physics and apply ML techniques to a real-world Physics problem. I had a few initial ideas, namely:

1) Analysis of Star Spectra
    - By analysing which wavelengths of light different stars absorb, we are able to see the colour of the star and also which elements are being fused in the core of the star. This allows us to see where the star is in its life cycle and therefore accurately determine its age. The user could manually input the wavelengths absorbed and the AI would be able to determine the elements being fused and provide an estimation of the age of the star.

2) Detection of Gravitationally Lensed Images
    - This was inspired by a bout of research into Special and General Relativity. Captured images of distant galaxies and stars are often warped by a phenomenon called gravitational lensing when very gravitationally dense objects, such as galaxy clusters, in the way, bend the path of the light, resulting in images which would otherwise be unable to be seen to be seen, albeit slightly warped. The proposed project will be able to use ML to detect whether or not an image has been gravitationally lensed and possibly reverse the effects of gravitational lensing to form the original image.

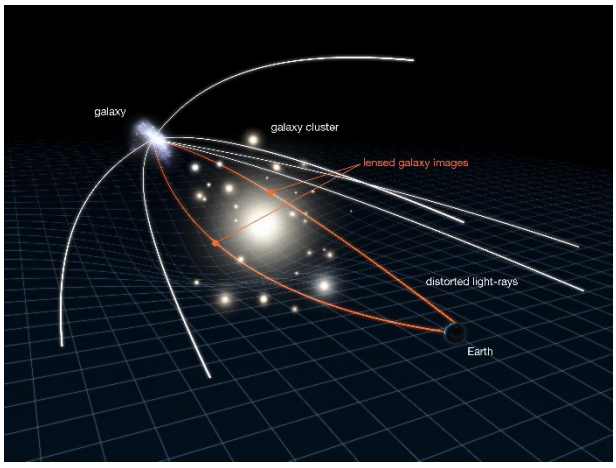


Figure 1: Diagram representing gravitational lensing from (https://esahubble.org/images/heic1106c/)

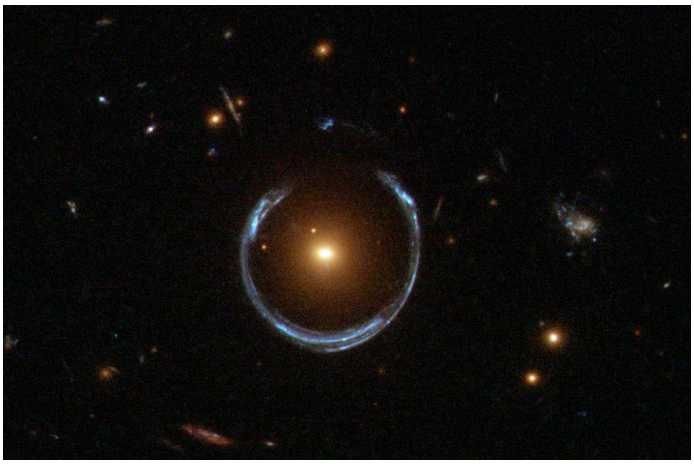


Figure 2: Einstein ring formed due to gravitational lensing (https://en.wikipedia.org/wiki/Gravitational_lens)

3) Prediction of Half-Lives of Radioactive Isotopes
    - This was inspired by recent Physics lessons on Radioactivity in which incredibly long and incredibly short half-lives were discussed. The proposed solution will be able to predict the half-life of a given (radioactive) isotope based on factors such as proton number and neutron number, among other factors.

Preliminary Analysis of Initial Ideas and Selection of Final Idea

1) Analysis of Star Spectra
    - The good thing about this project is that it will involve the AI learning of the link between elements fused and the star spectra of the star. The problem with this project, however, is that this is not an actual problem as all the possible colours and their meanings are already known, meaning there is not much use of an AI which does this. The project can be hard-coded and will perform the same.

2) Detection of Gravitationally Lensed images using ML
    - While there had been research into using ML for this[1], the level of Mathematics and data required for a project like this was much beyond my reach, especially if I wanted to add functionality to reverse the image. This is because it entailed knowing things like when the image was taken, by which telescope, and then the state of all the clusters in the way at that moment, before having to plug it into the field equations for general relativity to be able to reverse the image. The project would have been too data intensive and so I decided against it.

3) Prediction of Half-Lives of Radioactive Isotopes
    - Preliminary research into this showed that using ML for this had been proposed and shown to be effective by a paper in 2014[2] and had since been used for research purposes for both alpha decay[3] and beta decay[4]. The databases were also freely published and available for usage and there is not much to learn in terms of the concept itself. The difficulty in experimentally measuring incredibly long and incredibly short half-lives also justifies the need for a project like this for use in a research setting. For this reason, **I am choosing to continue with this for my final project**.

Stakeholders

For an academic project like this, the main stakeholders include educators and/or researchers. Educators will be able to use this product to show their students the half-lives of specific radioactive nuclei and I could add diagrams to show how half-life changes depending on different factors if I want to make it more specialised to education. For a more academic product, it would be used for predicting half-lives of novel radioactive isotopes which may be discovered or predicted, helping guide their research and expectations.

Why Is It Amenable to a Computational Solution?

Due to the difficulties in measuring very long and very short half-lives, as well as the random nature of radioactive decay, a computational solution may be much more suitable than an analytical one. While there do exist analytical models for alpha decay half-lives, namely the Effective Liquid Drop Models, the Generalised Liquid Drop Models (ELDM and GLDM respectively) and the Viola-Seaborg Model, a paper in 2019[5] showed that an approach based around ML is able to produce more accurate results. Due to the error contained in experimental results as well, an AI which is able to take this into account could potentially be more accurate than experiment. Part of my evaluation process will be to compare my model against the existing analytical models and to see if my model is more accurate.

Another aspect to consider is the fact that the development of neural networks (NN) is very processor heavy, especially in the optimisation and storage of the floating point weights and biases (more on this later) to create an accurate model. For this reason, any approach using a NN requires computationally advanced components, especially a GPU, which is specialised for floating point operations. This is another reason as to why a computational solution is ideal for this project.

Research into Problem

While doing preliminary research, I found that all the papers that did use ML for learning half-lives only focused on predicting half-lives based on 1 mode of decay rather than all of them. I believe the reason this was done was due to the differing complexities of each type of decay; beta decay consisted of many different subtypes while alpha decay has only 1 type. Due to the importance and variety in the types of beta decay, I do not feel that a project which solely focuses on half-lives, like mine, would be a sufficiently good tool to use in education or research. As the complexities

---

[1][Pearson, J., Pennock, C. and Robinson, T. (2018). *Auto-detection of strong gravitational lenses using convolutional neural networks.* Emergent Scientist, 2, p.1.] and [arXiv:2104.01014 [astro-ph.IM]]

[2] [Bayram, T., Akkoyun, S. and Kara, S.O. (2014). *α-decay half-life calculations of superheavy nuclei using artificial neural networks.* Journal of Physics: Conference Series, 490, p.012105.]

[3] [Freitas, Paulo & Clark, J.. (2019). *Experiments in machine learning of alpha-decay half-lives*.]

[4] [Niu, Z.M., Liang, H.Z., Sun, B.H., Long, W.H. and Niu, Y.F. (2019). *Predictions of nuclear β -decay half-lives with machine learning and their impact on r -process nucleosynthesis.* Physical Review C, 99(6).]

[5] [Freitas, Paulo & Clark, J.. (2019). *Experiments in machine learning of alpha-decay half-lives*.]

of beta decay are also out of the scope of A-Level syllabus, and I already have a quite advanced project, I do not feel confident in learning all the pre-requisites for a project which focuses on beta decay.

Another reason to focus solely on alpha decay is due to the increasing amounts of research into super-heavy elements (SHEs). More specifically, this includes their production and usage in nuclear fusion. One of the main properties researchers are interested are their half-lives, and as most SHEs decay via alpha decay, my project could go towards research in that field. For this reason, I have decided to focus on alpha decays for my project.

Potential Successes with Selected Problem

Potential successes include being highly accurate in my predictions of alpha decay half-lives and being more accurate than existing models in this regard. A standard way of measuring the efficacy of a model is by measuring the smallness of the standard deviation of the model from experiment[6]:

$\sigma = \frac{1}{n_T}\left(\sum_{n=1}^{n_T}\left(t_n^{exp.} - t_n^{mod.}\right)^2\right)^{\frac{1}{2}}$, where t is the base-10 logarithm of the half-life.

I will include this in my testing to allow myself to compare my model to existing analytical models.

Potential Issues with Selected Problem

The main potential issues occur with not having enough data and therefore not having a very accurate model. To try and combat this I will try and gather data from as many different sources as possible. This may cause me to have duplicate data, however, and so I will have to take care to combat this.

Another issue to consider is the one of predicting a half-life for things which do not have a half-life. This will occur because my training data will not be varied enough to include every single possible isotope and then define which ones will and will not have a half-life with alpha decay, and so, the network will return a half-life for every single input. Because this is to be used in research and education, however, the researchers will know whether or not an isotope will decay via alpha emission, and so it is not too much of an issue. For the sake of completeness, however, I will implement features to allow the dataset to be updated so that in future, these issues can be accounted for.

Analysis of Existing Similar Solutions

While there are similar existing solutions within academia which have used similar techniques, there is no publicly available software employing a NN for this problem. This is understandable as there is not really a need for the public to predict alpha-decay half-lives, however, I still think a tool should be publicly available to researchers and academics who wish to use the tool.

The first paper[7] I looked at used a network with 6 input neurons: atomic number, neutron number, parity, decay energy, distance from (nearest) proton magic number and distance from (nearest) neutron magic number. They then had a hidden layer (of 4 units), a learning rate of 0.001, and the activation function $\tanh(x)$. They also used Nesterov momentum during their gradient descent with the value of the momentum being 0.99, and regularisation with a strength of 0.1.

They used 3000 epochs (passages through the training data) and their cost function was the standard deviation σ. In practice, their results had a standard deviation of 0.4910 from the true values when measured on the test set, which was noticeably better than the standard deviation of the ELDM model, which was 0.5845.

The second source[8] I looked at used only 3 input neurons: atomic number, neutron number and mass number. While it did not go into detail in regard to the structure, it was found that when the network was trained to find the half-life, it was not very successful, however, when the network was trained to find the base-10 logarithm of the half-life, it was found to be very accurate.

---

[6] [Freitas, Paulo & Clark, J.. (2019). *Experiments in machine learning of alpha-decay half-lives*.]
[7] Ibid.
[8] [Bayram, T., Akkoyun, S. and Kara, S.O. (2014). *α-decay half-life calculations of superheavy nuclei using artificial neural networks.* Journal of Physics: Conference Series, 490, p.012105.]

Features Adopted and Rejected from the Existing Solutions

Based on the existing solutions, I have decided to follow the following structure for my neural network:

- 6 input neurons: proton number, neutron number, atomic number, decay energy, distance from proton magic number and distance from neutron magic number.
- Learning rate of 0.01
- Minimum of 4 hidden layers of 16 neurons each
- Cost function of σ (standard deviation).
- Output of the base-10 logarithm of the half-life.
- Activation function being the sigmoid function as opposed to the tanh function. More detail on this later.

Interviews/Client Research

For my stakeholders, I chose university researchers and professors in higher education, as they will be the ones that benefit the most from a specialised program such as mine. Because this is not a general half-life predictor, it will not necessarily be of as much value for science-communication or for the general public. Another reason researchers are the ones who will extract the most benefit from a project like this is due to the increasing amount of research into SHEs as was mentioned earlier. Due to this, I got in contact with a researcher from a nearby university who wished to remain anonymous, and after explaining the premise of what I was setting out to do, I asked him the following questions.

Interview Questions

1) How useful are the statistical models for alpha-decay (ELDM, GLDM) in your work?
2) How would you quantify the accuracy of a half-life model?
3) How would a more accurate model benefit your work?
4) How would you like to interact with the software?
5) Would you like the ability to update the dataset which the network is trained from?
6) Would you like the ability to change the structure of the model?
7) How important is the transparency of how the model works?

Interview Results

1) The ELDM and GLDM models are very useful in guiding our assumptions of the behaviour of these isotopes. They act as a useful tool in shaping our expectations and allowing us to build a pretty good model of how things will behave and interact. Obviously, they're not exact but they are close enough to allow us to build a good picture
2) The accuracy is generally quantified by the standard deviation of the logarithm of the model compared to the experimental half-life because it helps get around the different orders of magnitude the half-lives can span.
3) A more accurate model which can predict the half-lives of novel isotopes would be quite useful. We are always trying to develop more and more accurate models based on our increasing understanding of the processes governing radioactive decay, so a modelling method which is more accurate would be very useful. One thing which I may suggest is that, although the model may be very accurate, it should also be clear how it works, as one of the advantages of these models is that they build off of what we understand conceptually to help us build a picture of what is around us.
4) A simple interface would be sufficient in which we can enter the relevant details if we have them and get a prediction. Some software can be very complicated to use, as you have to worry about whether or not you have entered every piece of data etc. so a simple interface would be quite useful.
5) Having the ability to update the dataset would be very useful as it means we will be able to update your model as and when new experimental data comes in. This would help us in keeping as up-to-date as we can with our models.
6) This is not much of a concern as I don't expect us to be fiddling around with the settings of the model, as long as it functions well enough and we can update it based on new experimental data, it should be sufficient.
7) As physicists, we are always trying to increase our understanding of how the world works, however, I understand that with a neural network, it can be difficult to understand the implications of why the model

sets itself up as it does. While it would be nice to understand how the model works, I don't feel that it is necessary in helping us deriving experimental value from it,.

Conclusions from Interview(s)

It is clear from the interviews that the ELDM and GLDM models are very important in research (1) and it was also clear that having a more accurate model would be of great benefit to the researchers (3). For this reason, building a more accurate model than the statistical models will be my aim. I will quantify this using the standard deviation of the logarithm of the half-life compared to the model (2).

The GUI will be very simplistic to allow the researchers to interact quickly with it and not have to worry about whether they have entered the right bits of data in the right places (4). I will also try and allow for the ability to update the dataset as per the stakeholder's wants (5). I may add the ability to update the structure of the network and allow them to retrain it, although, it is clear that this is not something very important to the stakeholder (6).

Requirements

Software Requirements

I am planning to keep the software quite lightweight so that any device is able to run it. Therefore, the software requirements are limited to:

- Python 3.10 (with the appropriate libraries)
- A Python Interpreter

Hardware Requirements

- A PC which is capable of running Python

No specific hardware is required for my program. Any PC capable of running Python should be able to run my program.

Stakeholder Requirements

- A better accuracy than the statistical models for alpha decay
- Simple GUI with easy navigation

These requirements were based on the answers the stakeholders gave to my questions above.

Success Criteria

Essential

Criterion	Type	How to Evidence
Predictions of the neural network must have better accuracy than the statistical model	Functional	Compare standard deviations of both the statistical model and the neural network
The network is able to account for updates to the database of isotopes	Functional	Screenshots of lack of hard-coding
User is able to easily navigate the GUI	Usability	Stakeholder feedback and screenshots of simplistic GUI

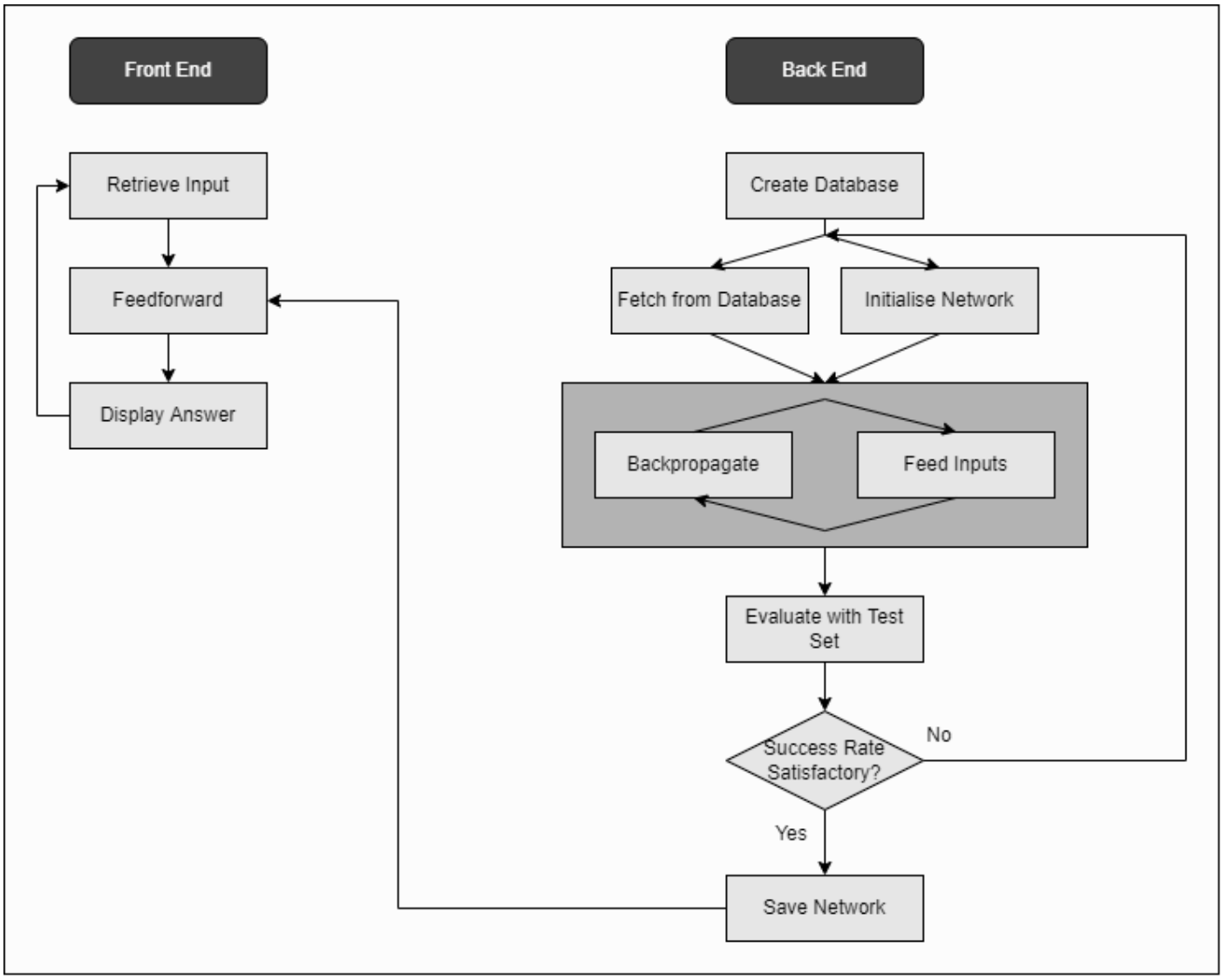
Desirable

Criterion	Type	How to Evidence
User is able to retrain the network with their own structure and settings	Functional and Usability	Screenshots of retraining screen allowing the user to do this
It is easy for the user to make any adjustments to the network	Usability	Stakeholder feedback
User is easily able to update the database	Usability	Screenshots of code allowing the user to do this and stakeholder feedback in regards to the ease

Other desirable criteria include anything which may arise in development, as I have not yet started development and do not know other success criteria may arise.

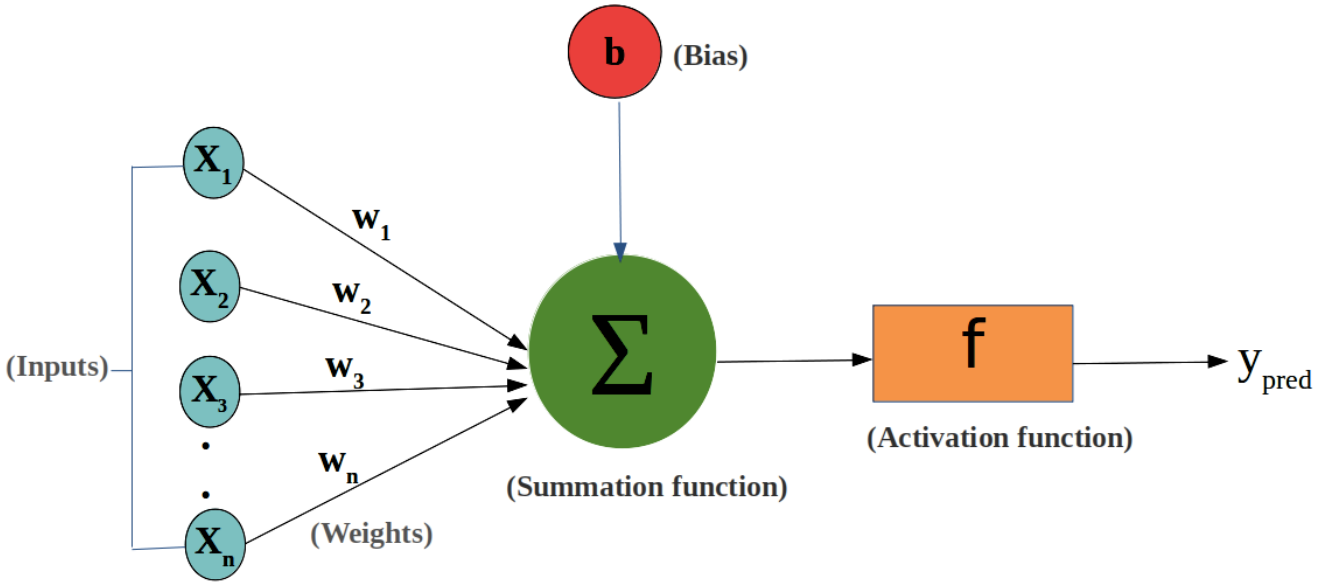
Design

Due to this being a large project, I will have to split it into multiple smaller components to allow it to be manageable. I first split the program into front-end and back-end, to signify which parts of the program the user will and won't interact with. I will update my front-end and back-end development plans once I have done more research into neural networks and machine learning:



Neural Networks and How They Work

Structure and Feeding Forward



2 https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f

Neural networks consist of multiple units of "neurons", structured in layers, with each layer feeding into the next. A neuron can just be thought of as something which stores a number, called its activation. Each neuron has a specific weight which its activation is multiplied by, and a bias which is added to it, which contribute to the activations of each neuron in the next layer. The activation of that next neuron depends on the weighted sum of all the activations in the previous layer, added to their individual biases (pictured above). This sum then put through an activation function, such as the sigmoid activation function or a ReLU (Rectified Linear Unit), to give it its final activation. Every neuron in a layer is linked to every neuron in the next layer, with each of them having their own individual weights and biases. For larger networks, it becomes incredibly complicated to keep track of all the weights and biases, as well as all the indexing of which layer each neuron is in, and so we often use matrix multiplication to represent this more simply:

The activation of neuron 0 is the weighted sum of all the activations of the previous layer (with their own specific weights) + a bias, all plugged through a specific activation function:

$$a = neuron \quad a^{(layer)}_{number} \quad w_{layer,neuron}$$

$$a^{(1)}_0 = \sigma(w_{0,0}\ a^{(0)}_0 + w_{0,1}\ a^{(0)}_1 + \ldots + w_{0,n}\ a^{(0)}_n + b_0)$$

$$\sigma = activation\ function$$

This can be represented more simply using matrices:

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a^{(0)}_0 \\ a^{(0)}_1 \\ \vdots \\ a^{(0)}_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}\right)$$

$$a^{(1)} = \sigma(\mathbf{W}a^{(0)} + \mathbf{b})$$

By using matrices, we can greatly decrease the amount of processing power and data storage we need, as the activations of a whole layer are dependent on only 3 stored quantities. To do this, I will use a library called numpy in Python to define and multiply matrices, as numpy is highly optimised for this, meaning it will greatly decrease the load on the user's PC.

The activation function I am choosing to use is the sigmoid function, which is defined as $y = \frac{1}{1+e^{-x}}$. This is one which is often used in industry because it normalises the value of the sum, giving it a value between 1 and 0, preventing certain factors from influencing the activation a lot more than others. I decided to use this as it is easier to differentiate than something like $y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, and, unlike ReLU, it is non-linear, meaning that its derivative is not a constant. This makes backpropagation work more effectively as the neural network will not be a set of linear transformations.

To ultimately receive an output from the network, a set of inputs has to be multiplied through each layer, before eventually reaching the output layer from which the activations are interpreted to represent the final output of the network.

Backpropagation and Minimising Cost

Gradient Descent

The initial weights and biases of a neural network are initialised randomly, however, due to this, the output from the network will be incredibly inaccurate. I will then need to adjust the weights and biases accordingly to allow a useful output to be reached. This has to be done iteratively, step by step. The reason for this is because it is a practically intractable problem to iterate through every single possible combination of weights and biases in a network before you decide on the one which gives you the outputs you require. For this reason, I will use a technique called gradient descent.

Gradient descent consists of taking the rate of change of the cost (a function of the difference between the desired value and the value given by the network) with respect to each weight and bias. Because we are working with vectors and matrices, the rate of change will give us a gradient vector pointing in the direction of increase. We then negate this to get the vector giving us the direction of the decrease of the cost, and adjust all the weights and biases proportionally. This is similar to finding a local minimum of a graph in a cartesian space. The constant of proportionality is called the learning rate and it can be thought of as the size of the step which the weights and biases take, or the size of the step down the graph. The smaller this is, the less likely the network is to overshoot and miss the local minima, however, it will take longer to train. I have decided to use a learning rate of 0.01 as this is sufficiently small enough to allow for this, but not so small that the network won't converge in a reasonable amount of time.

Backpropagation

Backpropagation is a technique which allows you to compute this gradient vector. By taking the error of the output layer, we know how much we need to adjust the output value by, however, because each neuron in the previous layer contributes a different amount to the overall value, we need to propagate the error backwards using the weights and biases to see how much we need to adjust each activation in the previous layer. As these next activations are dependent on the activations of the layer before that, we propagate the error back etc. and continue doing this until we have the adjustments that need to be made to each weight and bias. To propagate the error backwards, we have to use the transpose of the weight matrix to undo the transformation the weight matrix makes on the activations.

To compute the change of the cost with respect to each weight and bias, we first need to compute an intermediary quantity called the error. If our activation is defined as $a^1 = \sigma\left(\boldsymbol{W}a^{(0)} + \boldsymbol{b}\right)$, (activation in layer 1 is the weighted sum of the activations in layer 0 + the biases, all in the sigmoid function) we can define $z = \boldsymbol{W}a + \boldsymbol{b}$. If we nudge this "presigmoid" value by a small change $\boldsymbol{\Delta}z$, the overall change to the cost is $\frac{\partial C}{\partial z}\boldsymbol{\Delta}z$. If $\frac{\partial C}{\partial z}$ is small, then that means the small change in z will not have much of an effect, meaning that the neuron is already quite optimal. If it is large, then that means a small adjustment has a large effect on the cost. Because of this, $\frac{\partial C}{\partial z}$ can be thought of as the error δ.

By the chain rule:

$$\delta = \frac{\partial C}{\partial z} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial C}{\partial a}\sigma'(z)$$

In matrix form, this can be represented as:

$$\delta = \nabla_a C \odot \sigma'(z)$$

where $\odot$ is the Hadamard product (element-wise product), and $\nabla_a C$ is the gradient vector of the cost with respect to the activations. My cost function for a single training example is $(a - t^{exp.})^2$. The derivative of this (without the constants) is simply $(a - t^{exp.})$. I can disregard the constants as the result will be multiplied by the learning rate anyway. The derivative of the sigmoid function $\sigma(x)$ can be shown to be $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. In this way, we now have a formula for finding the error of a particular layer of neurons.

Backpropagation works because δ can be written in terms of the layer in front using the idea of propagating the error backwards, giving us a recursive relationship between the error of each layer. If L represents the current layer:

$$\delta^L = ((w^{L+1})^T \delta^{L+1}) \odot \sigma'(z^L)$$

It can then be shown that:

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L$$

That is, the derivative of the cost with respect to a bias is just the error of that particular neuron. For the weights, the derivative of each weight is the activation of the neuron from previous layer multiplied by the error of the neuron (each connection has a weight):

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L$$

When applied iteratively backwards through a network, these relationships then give us the gradient vector of the cost function with respect to each of the weights and biases, which is exactly what we want.

I will apply backpropagation to adjust the weights and biases feeding a training example through the network and measuring the error. When this is applied and averaged over all the training examples multiple times, the weights should hopefully be in a good enough state to make accurate predictions over all training examples. To avoid the need for the user to constantly have to train the network, I will then save the states of the weights and just feed forward the user inputs.

Turning Classification into Regression

One of the things I have to consider is that I am not trying to get an output between 0 and 1. For this reason, I will not apply the sigmoid function to the output layer neuron, meaning the output will just be a linear combination of the activations of the previous neurons. This will mean my σ' or the derivative of my activation function will be 1. This is something I will have to account for in my code.

Development Plan

My development plan will involve me first extracting and storing the data in a convenient way so that it can be read and written to easily. I will then develop the network in a class. I have chosen to develop it in a class as this will allow me to encapsulate all the methods and attributes I need under one structure. It will also take away the need to rewrite code as I can just call all the relevant functions I need. I will then finally develop a user interface to allow the network to be interacted with and predictions to be made.

Extracting Data

Based on a paper[9] which did something similar to my project, I decided to take my data from 2 main sources:

- Cui, J.P., Zhang, Y.L., Zhang, S. and Wang, Y.Z. (2018). *α -decay half-lives of superheavy nuclei.* Physical Review C, 97(1).
- Cui, J.P., Xiao, Y., Gao, Y.H. and Wang, Y.Z. (2019). *α-decay half-lives of neutron-deficient nuclei.* Nuclear Physics A, [online] 987, pp.99–111.

This gives me a total of 213 isotopes which I can use to train and evaluate my data. I will write all the information about these, including the half-life predicted by the ELDM model, into a text file which I will convert into a csv file. I will do it like this as it is simpler to write to text files than it is to csv files in Python. As the papers storing the data are pdfs, I will first copy and paste the data into a text file to read from, before rewriting it in a useful format.

Database
Data Array
Get Proton Number Array
Get Neutron Number Array
Get Nucleon Number Array
Get Proton MN Distance Array
Get Neutron MN Distance Array
Get Energy Release Array

I will then develop a database class in which I can encapsulate all the methods I need to fetch whichever data I need from the database. This will greatly improve readability and also mean I do not have to worry about where each relevant piece of data is stored.

To allow for any future additions to the database, I will not hard-code the database or the feature retrieval, improving the longevity of the code and conforming to the stakeholder's wants.

Usability Features

This part of the project will not be interacted with by the user so I do not need to consider any usability features in this section.

Developing the Network

Network	
Attributes	**Datatype**
Weights	Array of Numpy Matrices
Biases	Array of Numpy Matrices
Activations	Array of Numpy Matrices
Pre-Activations	Array of Numpy Matrices
Learning Rate	Float
Methods	**Explanation**
Feed Forward	Feed an input through and make a prediction
Backpropagate	Backpropagate for a given input
Train	Iterate through a training set, feeding forward and backpropagating for a given number of epochs
Evaluate	Evaluate the performance on a testing set

As was mentioned earlier, the network will be developed in a class to allow me to easily encapsulate all the methods and variables I need. I drew a class diagram on the left to help me visualise what needs to be done. In terms of developing the algorithms themselves, it will be a case of transferring the above mathematical formulas into code. As I don't want to hard-code the structure of the program, I will be coding the backpropagation and feedforward algorithms iteratively, letting the user define the structure of the network, however, I will first hard-code each step to make sure I understand the whole process. This will make it easier to first spot any errors but also prevent me getting confused with indexing in the loops as I anticipate this will be something that trips me up. To help with this, I decided to write some pseudocode to make sure I understood all the maths above:

[9] [Freitas, Paulo & Clark, J.. (2019). *Experiments in machine learning of alpha-decay half-lives*.]

Pseudocode for Network

```
class Network ():

    def __init__ (self, structure):
        self.weights = []
        self.biases = []

        for x in range (len(structure) - 1):
            self.weights.append (random numpy matrix of size specified by structure) # initialise weights randomly
            self.biases.append (random numpy matrix of size specified by structure) # initialise biases randomly

        self.activations = []
        self.preactivations = []

        for x in range (len(structure))
            self.activations.append(numpy matrix of 0) # initialise activations as being 0
            self.preactivations.append (numpy matrix of 0) # initialise preactivations as being 0

        self.learningrate = 0.01 # default value for learning rate

    def feedforward (Z,N,Q,A,ZDist,NDist):
        self.activations[0] = [Z,N,Q,A,ZDist,NDist] # first layer activations are fed in

        for x in range (len(structure)-1): # from the first layer to the penultimate layer
            activation = self.activations[x]
            weighted = np.matmul (self.weights[x], a) # multiply weights with activations
            sum = weighted + biases [x] # calculate weighted sum + biases to get a final sum
            self.preactivations.append (sum) # append non-activated sum to preactivations
            self.activations.append (sigmoid (sum) # append sigmoid-ed sum to activation

        aFinal = np.array([]) # calculate last layer activations manually to avoid sigmoid
        weighted = np.matmul(self.weights[len(structure)-1])
        sum = weighted + biases [len(structure)-1]
        self.preactivations.append(sum)
        self.activations.append(sum) # append non-activated sum to activations

    def backpropagate (activation, target):
        deltaWeights = [] # declare change in weights array
        deltaBiases = [] # declare change in biases array

        # error in last layer
        errorL = target - activation # last layer error
        deltaBiases.insert (0, errorL) # change in biases = error
        ''' change in weights = error multiplied with previous layer activations (no hadamard as last layer
        activation is linear) '''
        deltaWeights.insert (0, np.matmul (errorL, self.activations[len(structure)-2].transpose()))

        # error in second-last layer
        # propagate previous layer error backwards using transpose of weight matrix
        propagated = np.matmul (self.weights(size-2).transpose(), errorL)
        error2L = np.multiply (propagated, sigmoid_prime (preactivations[size-3]))
        deltaBiases.insert (0, error2L) # inserting at start so I can iterate through normally
        deltaWeights.insert (0, np.matmul(error2L, self.activations[len(structure)-3].transpose()))

        .
        .
        .

        # do for each layer (in loop in final version)
        for x in range (0, len(structure - 1)):
            self.weights [x] = weights [x] + deltaWeights [x]
            self.biases [x] = biases [x] + deltaBiases [x]

    def train (trainSet, targets, epochs):
        for x in range (epochs):
            for y in range (len(trainSet)):
                activation = self.feedforward(trainSet[y])
                self.backpropagate (activation, targets[y])

    def evaluate (testSet, targets):
        errors = []
        for x in range (len(testSet)):
            error = self.feedforward (testSet[x]) - targets[x]
            errors.append (error**2)
        stddev = sum (errors) ** (1/2) / (len(testSet))
        return stddev
```

Once I have trained the network to a suitable standard, I will save the states of the weights and biases using a joblib dump which will convert the matrices into a bitstream which can be read from later. I am using joblib as it is specialised for storing numpy matrices, which is what all the weights and biases in my program will be stored as. I can then load these wherever I need to use the neural network.

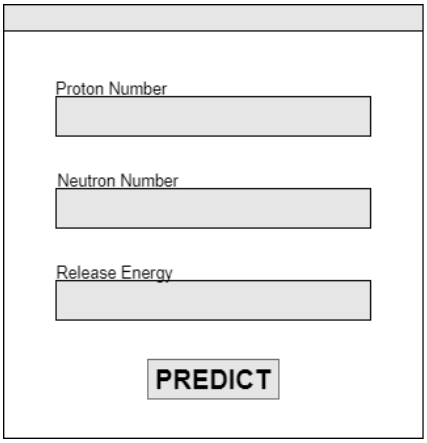
Usability Features

This part of the project will not be interacted with by the user so I do not need to consider any usability features in this section.

User Interface

This will be the last part of the project which I code and should be the simplest. There will be a basic UI which will allow the user to enter the data they need. For this, I will only need 3 text boxes to allow the user to enter value for the number of protons, neutrons, and the release energy. The distance from magic numbers and the nucleon numbers can be calculated from these. I will then feed them through my (trained) network which I will load from the joblib dump. The prediction will then pop up in a new window to make it clear which number on screen is the half-life.

The UI will be coded in tkinter as it is simple to design and add interactive elements onto. I mocked up a simple diagram on the right to illustrate how the interface may look.



Usability Features

This is the main part, if not the only part of the program which the user interacts with. For this reason, I need to make sure the UI is intuitive to understand and use. I also need to cater for all the incorrect inputs which a user may give, such as typing a letter when the program wants numbers. This will be done by creating try, catch statements in Python to catch any of the errors which may occur when I try to parse the user input. I will then need to clearly prompt the user for correct input. Overall, the UI is quite simple, so it should not be too difficult to both implement or use.

Testing Plan

Back-End Testing

Num.	Test	Success Criteria	Purpose
(1)	Feed forward a 3 neuron input through a small network manually	The network successfully outputs a 1x1 array	This will determine whether or not the matrix multiplication is correct and whether I can move forward with creating a more complicated network.
(2)	Feed forward a 6 neuron input with varying dimensions iteratively	The network successfully outputs a 1x1 array	This will determine whether or not my iterative feed-forward algorithm works.
(3)	Backpropagate a 3 layer network	The network successfully updates both sets of weights	This will determine whether or not my backpropagation is dimensionally correct
(4)	Check the error of my procedural backpropagation algorithm	Tends towards 0	This will determine whether or not the backpropagation is training the network to predict the half-life more and more accurately or not
(5)	Check the error of my iterative backpropagation with varying dimensions	Tends towards 0	This will determine whether or not the backpropagation is training the network to predict the half-life more and more accurately or not
(6)	Upload and load a small joblib array	Uploaded and downloaded are the same	If successful, I can use joblib to upload weights and biases to download and use them in my front end

Front-End Testing

Num.	Test	Success Criteria	Purpose
(i)	Enter incorrect data	The software catches incorrect data and prints out that incorrect data has been entered	This allows me to catch incorrect inputs and prompt the user for correct inputs
(ii)	Enter correct data	UI stores correct data and calculates extra data it needs	Allows me to determine whether my input system works as it should
(iii)	Feedforward an input	UI successfully computes half-life	Determines whether my feedforward and joblib loading is functional
(iv)	Try and create window	Window is created	Allows me to prompt the user for re-entry of data and to show them the results of the prediction
(v)	Enter correct data	Prediction is outputted correctly in new window	Determines whether the UI is functional in terms of feeding forward and showing data

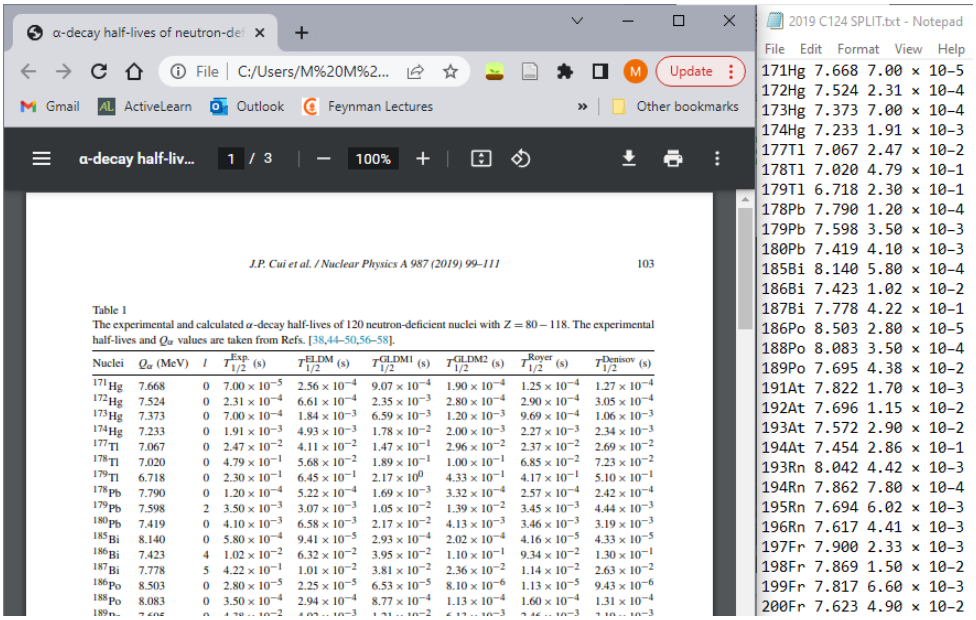
Development and Implementation

Extracting Data

I first started on my data extraction. The data which I will use comes from 2 main academic papers, mentioned above, however, the main problem I initially had was that they were pdfs, and reading from tables in pdfs is very difficult in Python. For this reason, I decided to copy and paste the data into text files and read from them. The 2 sources were formatted differently, however, and so I had to read from them separately. I decided to use the ELDM model to test against as this was common to both sources.

Retrieving the Data

I first copied the data into a text file because of the reasons mentioned above:



I then read from the text file into an array using this simple program:

```python
for line in file:
  N  = file.read(3)
  name = file.read(2)
  spaces = file.read(1)
  Q = file.read(5)
  spaces = file.read(1)
  t12 = file.read(11)
  spaces = file.read()
  eldm = file.read(11)
  data.append(name,N,Q,t12,eldm)
data.pop()
#python sees end of document as a \n so there was an extra
empty cell in the array
for x in range(0,len(data)):
  for y in range (0,len(elements)):
    if elements[y][0] == data[x][0]:
      data[x].insert(1, elements[y][1])
```

```
215U
216U
217U
218U
219Np
```

This works because the values under each of the columns were each of the same length. This made it easy to specify a number. One of the outliers, however, was Uranium, as its chemical symbol is U. For this reason, I made sure to add an extra space after the U in my text file (above right) This was feasible as there were only 4 Uranium isotopes.

The second code block is one where I was adding all the proton numbers. This source didn't use isotope numbers and instead listed all the elements according to their name. For this reason, I looked for an easy way to convert all these elements into their proton numbers. I found a csv file[10] online which contained this information, and after stripping it of what I didn't need, I compared the name of the element given in the database to the elements in the csv file and inserted the appropriate proton numbers in.

I then wrote the data into a text file. The code was not very readable, however, I knew that I would only have to do this once and never come back to it, so I didn't take the time to overly neaten it. This was when I encountered another issue: although the bases of the exponents were of the same length, the exponents weren't. This meant that I had to read different lengths for different exponents, and so the one size fits all approach which I was implementing didn't work. You can see I had artefacts of following data in my ELDM data (last column in screenshot on left) due to the different exponent lengths (below).

```
Fm, 100, 243, 8.690, 2.54 × 10-1, 7.52 × 10-1
Md, 101, 246, 8.890, 9.20 × 10-1, 3.95 × 10-1
Md, 101, 247, 8.764, 1.20 × 100, 1.05 × 100 1.
No, 102, 251, 8.752, 9.64 × 10-1, 2.06 × 100 3
Lr, 103, 253, 8.918, 7.02 × 10-1, 1.40 × 100 2
Lr, 103, 254, 8.816, 2.38 × 101, 2.79 × 100 3.
Rf, 104, 255, 9.055, 3.46 × 100, 1.35 × 100 1.
Rf, 104, 256, 8.923, 2.08 × 100, 2.87 × 100 3.
Md, 101, 247, 8.764, 1.20 × 100, 1.05 × 100 1.
No, 102, 251, 8.752, 9.64 × 10-1, 2.06 × 100 3
Lr, 103, 253, 8.918, 7.02 × 10-1, 1.40 × 100 2
Lr, 103, 254, 8.816, 2.38 × 101, 2.79 × 100 3.
Rf, 104, 255, 9.055, 3.46 × 100, 1.35 × 100 1.
Rf, 104, 256, 8.923, 2.08 × 100, 2.87 × 100 3.
Db, 105, 256, 9.340, 2.84 × 100, 3.82 × 10-1 5
Db, 105, 257, 9.206, 2.45 × 100, 9.08 × 10-1 1
Db, 105, 258, 9.500, 5.58 × 100, 1.24 × 10-1 1
Sg, 106, 259, 9.804, 3.11 × 10-1, 3.90 × 10-2
Sg, 106, 260, 9.901, 1.24 × 10-2, 2.07 × 10-2
Sg, 106, 261, 9.714, 1.87 × 10-1, 6.38 × 10-2
```

```
243Fm 8.690 2.54 × 10-1   7.52 × 10-1
246Md 8.890 9.20 × 10-1   3.95 × 10-1
247Md 8.764 1.20 × 100   1.05 × 100
251No 8.752 9.64 × 10-1   2.06 × 100
253Lr 8.918 7.02 × 10-1   1.40 × 100
254Lr 8.816 2.38 × 101   2.79 × 100
255Rf 9.055 3.46 × 100   1.35 × 100
256Rf 8.923 2.08 × 100   2.87 × 100
256Db 9.340 2.84 × 100   3.82 × 10-1
257Db 9.206 2.45 × 100   9.08 × 10-1
```

I thought about coding an elegant solution which takes into account the '–' signs but in the end, I decided to instead, introduce a gap of 3 whitespaces between all the data which may cause an issue. Then, I can read a fixed number of digits and stripped them of the whitespaces with the Python strip function. It did take some time to go through adding all the spaces but it was easier and quicker than reading and accounting for '–' signs. My function became:

[10] [https://gist.github.com/GoodmanSciences/c2dd862cd38f21b0ad36b8f96b4bf1ee]

```python
for line in file:
    N = file.read(3)
    name = (file.read(2)).strip()
    Q = (file.read(7)).strip()
    t12 = (file.read(12)).strip()
    eldm = (file.read(14)).strip()
    data.append([name,N, Q, t12, eldm])
data.pop()
```

```
237Cf 8.220 1.14 × 100    4.96 × 100    8.57 × 100
240Cf 7.719 9.76 × 101    2.41 × 102    4.35 × 102
242Es 8.160 3.12 × 101    1.52 × 101    2.90 × 101
243Es 8.072 3.29 × 102    3.11 × 101    5.51 × 101
243Fm 8.690 2.54 × 10-1    7.52 × 10-1    1.25 × 10
246Md 8.890 9.20 × 10-1    3.95 × 10-1    5.98 × 10
247Md 8.764 1.20 × 100    1.05 × 100    1.42 × 100
251No 8.752 9.64 × 10-1    2.06 × 100    3.14 × 100
253Lr 8.918 7.02 × 10-1    1.40 × 100    2.09 × 100
```

This worked very well, and I no longer had any of the artefacts as you can see below. The stripping also got rid of the space after the U for Uranium which I forgot to account for, also pictured below.

```
Cm, 96, 234, 7.365, 1.28 × 102, 9.07 × 102
Cm, 96, 236, 7.067, 2.27 × 103, 1.27 × 104
Cf, 98, 237, 8.220, 1.14 × 100, 4.96 × 100
Cf, 98, 240, 7.719, 9.76 × 101, 2.41 × 102
Np, 93, 223, 9.650, 2.15 × 10-6, 1.13 × 10-5
Np, 93, 225, 8.790, 3.60 × 10-3, 1.77 × 10-3
Np, 93, 226, 8.200, 3.50 × 10-2, 9.42 × 10-2
```

```
U , 92, 215, 8.588,
U , 92, 216, 8.542,
U , 92, 217, 8.169,
U , 92, 218, 8.775,
```

The next thing to get rid of was the exponent, as the" x 10" cannot be read or used by Python. Instead of converting out of standard form, I decided to convert the exponent to "e" which means the same thing as " x 10" and is understood by Python. This was an easy fix:

```python
for x in range(0,len(data)):
    data[x][5] = data[x][5].replace(" × 10", "e")
    data[x][4] = data[x][4].replace(" × 10", "e")
    # note: x=/=×
```

This did not work the first time when I typed in " x 10" by myself. To fix this, I decided to copy and paste what the text file had and realised that the multiply sign was not the letter x but the actual Unicode multiplication sign, which is why it did not work. I fixed this but didn't screenshot the difference because the code looked virtually the same. This gave me the format I wanted:

```
Cm,96,138,234,7.365,1.28e2,9.07e2
Cm,96,140,236,7.067,2.27e3,1.27e4
Cf,98,139,237,8.220,1.14e0,4.96e0
Cf,98,142,240,7.719,9.76e1,2.41e2
```

Source 2

I then repeated the same processes above for the second source. When I wrote the database, I formatted it as if it was a csv file. This was because I was planning to convert it to a csv file anyway (by changing the file extension) as they are very easy to read data from. I wrote to the text file using the following code:

```python
Database = open('Database.txt','w')

Database.write("Element, Z (Protons), N (Neutrons), Nucleons, Energy, Half-Life, ELDM\n")

for x in range(0,len(data)):
  for y in range(0,7):
    Database.write(str(data[x][y]))
    if y==6:
      Database.write("\n")
    else:
      Database.write(",")

#Database.write("Source 2 -----------------\n") # Debugging

for x in range(0,len(data2)):
  for y in range(0,7):
    Database.write(str(data2[x][y]))
    if y==6:
      Database.write("\n")
    else:
      Database.write(",")

print("Databasing Done!")
Database.close()
pTable.close()
file.close()
source2.close()
```

In the end, I had a database of 213 elements which I could use to train and test my neural network. I decided to use the traditional 80-20 split, where 80% of my data will be used to train the network and the remaining 20% will be used to test it. The whole databasing was done using an online IDE and is available here: https://replit.com/@MAmjad/Read-Data-Test#main.py.

Reading from Database and Encapsulation

The next step was reading from the database. I used the class diagram I made earlier and read from the database into a master data array. Whenever a function is called to retrieve a particular feature/data item, I read from this master array to a smaller array and returned that smaller array:

```python
import csv as csv
import numpy as np

class Data ():

    def __init__ (self):
        self.data = []
        with open("Database.csv") as database:
            csvreader = csv.reader (database)
            for row in csvreader:
                self.data.append([row[0],int(row[1]),int(row[2]),int(row[3]),float(row[4]),float(row[5]),float(row[6])])

        # Format: Element, Z, N, A, Q, T12, ELDM
        for isotope in self.data:

            Zdist = min([abs(isotope[1]-2), abs(isotope[1]-8), abs(isotope[1]-20),
                abs(isotope[1]-28), abs(isotope[1]-50), abs(isotope[1]-82), abs(isotope[1]-126)])
            isotope.insert (2, Zdist)

            Ndist = min([abs(isotope[3]-2), abs(isotope[3]-8), abs(isotope[3]-20),
                abs(isotope[3]-28), abs(isotope[3]-50), abs(isotope[3]-82), abs(isotope[3]-84), abs(isotope[3]-126)])
            isotope.insert (4, Ndist)

        # Format: Element, Z, ZDist, N, NDist, A, Q, T12, ELDM
```

I also added the Zdist and Ndist data items. These are the distances from the nearest proton magic numbers and neutron magic numbers respectively (2, 8, 20, 28, 50, 82, 126 for protons and neutrons as well as 84 for neutrons). This was one of the inputs of a solution I looked at which had better results than the statistical model.

```python
def getZ (self):
    Z = []
    for x in self.data:
        Z.append(x[1])
    return Z

def getZDist (self):
    ZDist = []
    for x in self.data:
        ZDist.append(x[2])
    return ZDist
```

```python
def getN (self):
    N = []
    for x in self.data:
        N.append (x[3])
    return N

def getNDist (self):
    NDist = []
    for x in self.data:
        NDist.append(x[4])
    return NDist
```

```python
def getA (self):
    A = []
    for x in self.data:
        A.append (x[5])
    return A

def getQ (self):
    Q = []
    for x in self.data:
        Q.append (x[6])
    return Q
```

```python
def getHL (self):
    HL = []
    for x in self.data:
        HL.append (x[7])
    return HL

def getModel (self):
    Model = []
    for x in self.data:
        Model.append (x[8])
    return Model
```

I then defined all the above subroutines to read from the data array and allow me to retrieve any of the data whenever I wanted. This was relatively straightforward, and so there weren't any bugs to fix in this part of the project. This did make my life easier later on when retrieving data to train the model. I then went onto the model next.

Developing the Network

Feed-Forward Method

Step-By-Step

When doing the feed-forward method, I decided to first code a smaller version of my final network so I could get my head round the matrix multiplication. For this, I chose a 3 layer network with 3 input neurons, 1 hidden layer of 20 neurons and 1 output neuron which I defined as an array called `structure`, where `structure = [3, 20, 1]`.

I then defined and randomly initialised the weight matrix `w1` as being the weights going into layer 1. This was a 20x3 matrix which would be multiplied by the 3x1 activations matrix (for the first layer neurons), giving me a 20x1 matrix, which signify the 20 neurons in layer 2. I then defined the bias randomly as well. I was originally going to define and initialise these manually with Python's random library to randomly make up values, but then I discovered numpy's `np.random.rand(shape)` which automatically defines a matrix with the given shape with random float values between 0 and 1:

```
import numpy as np
'''
This will be a 3 layer structure, 3 input neurons, 20 neurons in 1 hidden layer
and then 1 output neuron
'''
structure = [3,20,1]

# sigmoid ##########################################################################
def sigmoid(x):
    return 1.0/(1.0+np.exp(-x))

# define matrix for first sets of weights to second layer using a 2d array ####################

'''w1noob = []
for x in range (structure[1]):
    w1noob.append([0,0,0])
w1 = np.array(w1noob)'''

# or:

w1 = np.asarray(np.random.rand(structure[1],structure[0]))
'''
print("Initial Weights -------------------------")
print(w1)
print(w1.shape)
'''

# define biases ####################################################################
b1 = np.asarray(np.random.rand(structure[1],1))
'''
print("Initial Biases -------------------------")
print(b1)
print(b1.shape)
'''
```

```
# store given activations for first layer ##############################

Z = 92
N = 126
Q = 8.775

'''
Z = int(input("Protons: "))
N = int(input("Neutrons: "))
Q = float(input("Energy Release: "))
'''

print("Initial Layer 2 Activations ----------------------")
a0 = np.array([[Z],[N],[Q]])
'''
print(a0)
print(a0.shape)
'''
```

I then went to compute the second layer's activations by multiplying the weight and activation matrices and adding the biases on. When I printed the shape of the matrix to check it was correct however, it gave me an unexpected result:

```
# compute activations of second layer
a1 = np.array([])
mult = np.array([np.matmul(w1,a0)])
print (mult.shape)
a1 = sigmoid (mult+b1)
print (a1)
```

```
(1, 20, 1)
```

It was saying that the `mult` array was a 3D matrix of size 1x20x1. This could cause errors further down the line if I didn't sort it so to be on the safe side, I used numpy's `reshape` function which lets you reshape arrays without losing any of the data (if it is possible to do so).

```
# compute activations of second layer
a1 = np.array([])

mult = np.array([np.matmul(w1,a0)])
mult = np.reshape(mult, (20,1))

a1 = sigmoid((mult+b1))
print(a1)
#print(a1.shape)
```

I then printed `a1` thinking that the sigmoid function would give me a range of values between 0 and 1, as was its job. When I printed the result, however, I got the following (left):

```
Initial Layer 2 Activations -------
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
[Finished in 852ms]
```

This was clearly not what I wanted. The sigmoid function only outputs values close to 1 for very high inputs. The fact that they were all 1 indicated to me that my weights may not be doing what they should be in weakening or strengthening the activations as appropriate.

To check this, I printed out my weighted sum to see what the matrix going into the sigmoid function was:

```
presigmoid = mult + b1
print(presigmoid)
```

This outputted the following:

```
Initial Layer 2 Activations ----------------------
[[ 74.27588157]
 [ 95.75869071]
 [220.0405366 ]
 [102.66039312]
 [ 90.52561058]
 [140.31741174]
 [200.58701259]
 [ 46.79606826]
 [138.84777331]
 [109.0972436 ]
 [108.98609931]
 [ 79.92519062]
 [135.35501945]
 [115.42719549]
 [ 32.33742461]
 [ 78.98760372]
 [131.86221612]
 [204.4874777 ]
 [ 56.39202016]
 [164.09997967]]
[Finished in 873ms]
```

From this, it is clear what went wrong. Although the weights and biases had been initialised randomly and were small (between 0 and 1), this did not account for the massive discrepancies in the sizes of my input data. My release energy was below 10 while my proton and neutron numbers were in or near the hundreds. This made me realise that I needed to **normalise** the data.

Normalisation

Normalisation is the taking of a value and transforming it in some way to be between 0 and 1. An article by Tensorflow[11] stated that it is good practice to normalise features that use different scales and ranges. You are unlikely to converge to a solution without normalisation.

After researching normalisation, I determined that min-max normalisation would be the easiest form to use in this context. The formula for min-max normalisation is as follows:

$$x = \frac{x - x_{min}}{x_{max} - x_{min}}$$

This guarantees a value between 0 and 1, and because the minimum and maximum values were easily found due to my database class having all the arrays I needed along with the python `min()` and `max()` functions, I decided to use this. I imported my Data class and initialised an object called gd (getData) and added the following:

```
Z = 92
N = 126
Q = 8.775

# min max normalisation

Qmin = min(gd.getQArray())
Qmax = max(gd.getQArray())

Z = (Z-80)/(118-80)
N = (N-92)/(177-92)
Q = (Q-Qmin)/(Qmax-Qmin)
```

I knew the minimum and maximum values for the protons and neutrons so I decided to just use them rather than retrieving the array to save time. This had the desired effect:

```
# compute activations of second layer
a1 = np.array([])
mult = np.array([np.matmul(w1,a0)])
mult = np.reshape(mult, (20,1))
presigmoid = mult + b1
print(presigmoid)
```

```
[[[1.4976883 ]
  [1.47894352]
  [1.37529387]
  [0.57983906]
  [1.34072499]
  [1.42724619]
  [1.0644913 ]
  [0.47814135]
  [1.10725799]
  [0.79080841]
  [0.87429872]
  [1.21127459]
  [1.33845756]
  [0.65039746]
  [1.58987488]
  [1.28279449]
  [0.88129606]
  [1.36796447]
```

[11] [https://www.tensorflow.org/tutorials/keras/regression#the_normalization_layer]

I then put this through the sigmoid function:

```
# compute activations of second layer
a1 = np.array([])
mult = np.array([np.matmul(w1,a0)])
mult = np.reshape(mult, (20,1))
presigmoid = mult + b1
#print(presigmoid)
a1 = sigmoid(presigmoid)
print(a1)
```

```
[[0.68779067]
 [0.78360365]
 [0.75375648]
 [0.71434007]
 [0.63102291]
 [0.66414528]
 [0.6519257 ]
 [0.77343133]
 [0.73042665]
 [0.80363014]
 [0.78391699]
 [0.71545361]
 [0.72994329]
 [0.65117453]
 [0.79762319]
 [0.64716551]
 [0.77420752]
 [0.71269508]
 [0.7934463 ]
 [0.78701092]]
```

These activations looked much better. I then moved on, calculating the final layer activation and this time, not plugging it into the sigmoid function because I wanted a regression network rather than a classification network:

```
# define matrix for second set of weights to 3rd layer using 2d array (which will be 1d anyway)
w2 = np.asarray(np.random.rand(structure[2],structure[1]))
b2 = np.asarray(np.random.rand(structure[2],1))
mult = np.matmul(w2,a1)
result = mult + b2
print("Final Half-Life: ", result)
```

```
Final Half-Life:  [[7.48140465]]
```

[Evidence for Test 1]

This worked. The final output is a 1x1 numpy matrix, showing all my matrix multiplication was consistent. This meant I was ready to code a more sophisticated network which would be able to perform the feed forward iteratively for a given structure.

```
# compute activations of second layer ################################
a1 = np.array([])

mult = np.array([np.matmul(w1,a0)])
mult = np.reshape(mult, (20,1))

presigmoid = mult + b1

print(presigmoid)
print ("-----------------")

a1 = sigmoid(presigmoid)
print(a1)
print(a1.shape)

# define matrix for second set of weights to 3rd layer using 2d array (which will be 1d anyway)
print("--------------------------------------------------")
w2 = np.asarray(np.random.rand(structure[2],structure[1]))
print (w2)
result = np.matmul(w2,a1)
print(result.shape)
print("Final Half-Life: ", result)
```

Iterative Feed Forward

I then decided to code my iterative feed-forward which had all six inputs:

- Proton Number
- Neutron Number
- Nucleon Number
- Release Energy
- Distance from Proton Magic Number
- Distance from Neutron Magic Number

I first used my Data library to import all the arrays I needed and defined my structure with 4 hidden layers of 16 neurons each:

```
Zdist = gd.getZdist()
Ndist = gd.getNdist()
ZArr = gd.getZ()
NArr = gd.getN()
AArr = gd.getA()
QArr = gd.getQ()
```

```
structure = [6,16,16,16,16,1]
```

I then used my procedural code from above to create an iterative feed method, manually adding and normalising the first layer activations and manually applying the last layer activations:

Defining and initialising `weights` and `biases`: ------->

Normalising input values using min-max normalisation: ------>

Setting normalised values to be initial activations: ---------->

Defined an array ah for hidden activation and tried to compute activations: --------->

Manually apply last layer activations ---------->

```
weights = []
biases = []
for x in range (0,(len(structure)-1)):
    weights.append(np.asarray(np.random.rand(structure[x+1],structure[x])))
    biases.append(np.asarray(np.random.rand(structure[x+1],1)))
print (len(weights))
print (len(biases))

def iterfeed(Z, N, A, Q, Zd, Nd, structure):
    # min-max scaling
    Z = (Z-min(ZArr))/(max(ZArr)-min(ZArr))
    N = (N-min(NArr))/(max(NArr)-min(NArr))
    A = (A-min(AArr))/(max(AArr)-min(AArr))
    Q = (Q-min(QArr))/(max(QArr)-min(QArr))
    Zd = (Zd-min(Zdist))/(max(Zdist)-min(Zdist))
    Nd = (Nd-min(Ndist))/(max(Ndist)-min(Ndist))

    a0 = np.array([[Z], [N], [A], [Q], [Zd], [Nd]]) # use scaled inputs as initial activations

    ah = np.array([])
    for x in range (0, len(structure) - 2):
        mult = np.array([np.matmul(weights[x], ah)])
        mult = np.reshape(mult, (structure[x],1))
        ah = sigmoid(mult + biases[x])

    af = np.array([])

    mult = np.array([np.matmul(weights[len(structure)-1],ah)])
    mult = np.reshape (mult, (structure[len(structure)-1],1))
    af = mult + biases[len(structure)]

    return af
```

When I ran this, however, I got an error:

```
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\NetworkPrototype2.py", line 53, in iterfeed
    mult = np.array([np.matmul(weights[x], ah)])
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 0 is different from 6)
```

It was saying that it could not multiply a matrix of size 0 with the weights matrix. I realised that the reason this threw the error was because I had defined ah as being empty, and then tried to use the values in ah to compute the next values in ah. For the second layer activations, I had to multiply by the first layer activations but I was multiplying by ah. I did not want to use if statements as that would have been inefficient and difficult to scale, so I decided to change the strategy with which I stored the activations and brought in an `activations` array. I was planning to do this later on when I introduced backpropagation, but this made me realise that I needed it now. I then added this in:

```
weights = []
biases = []
activations = []

for x in range (0,(len(structure) - 1)):
    weights.append(np.asarray(np.random.rand(structure[x+1],structure[x])))
    biases.append(np.asarray(np.random.rand(structure[x+1],1)))

for x in range (0, len(structure)):
    activations.append(np.zeros((structure[x],1)))

size = len(structure)

print (len(weights))
print (len(biases))
print (len(activations))
```

The reason `activations` is longer than the `weights` and `biases` arrays is because the weights act between layers, and so there are only `l-1` weights (1->2, 2->3, …, n-1->n), and the biases don't change the activations of the first layer, meaning there will be `l-1` biases. Every layer, however, has an activation, so there needs to be as many activations as layers. I initialised the activations with 0s rather than random numbers as then it would be clear when an activation has been computed or not. (It was around this point I experimented with different IDE themes so I apologise for the inconsistencies in the colours).

I updated my feed method accordingly:

Apply `weights[x]` to `activations [x]` and add the biases to get `activations [x+1]` (the next activation):

```
def iterfeed(Z, N, A, Q, Zd, Nd, structure):
    # min-max scaling
    Z = (Z-min(ZArr))/(max(ZArr)-min(ZArr))
    N = (N-min(NArr))/(max(NArr)-min(NArr))
    A = (A-min(AArr))/(max(AArr)-min(AArr))
    Q = (Q-min(QArr))/(max(QArr)-min(QArr))
    Zd = (Zd-min(Zdist))/(max(Zdist)-min(Zdist))
    Nd = (Nd-min(Ndist))/(max(Ndist)-min(Ndist))

    activations [0] = np.array([[Z], [N], [A], [Q], [Zd], [Nd]]) # use scaled inputs

    for x in range (0, size - 2):
        a = activations [x]
        mult = np.array([np.matmul(weights[x], a)])
        mult = np.reshape(mult, (structure[x+1],1))
        activations [x+1] = sigmoid(mult + biases[x])

    af = np.array([])
    mult = np.array([np.matmul(weights[size - 2],activations [size - 2])])
    mult = np.reshape (mult, (structure[size - 1],1))
    af = mult + biases[size - 2]
    activations.append ([af])
    return af
```

This worked:

```
i=0

t12 = iterfeed (ZArr[i], NArr[i], AArr[i], QArr[i], Zdist[i], Ndist[i], structure)
print ("Half-Life I: "+ str(t12[0][0]) + "s")
```

```
Half-Life I: 8.108464016701873s
[Finished in 880ms]
```

In order to test whether it was doing what I thought, however, I decided to recode my iterative feed-forward quickly and see what outputs it gave when I initialised the weights and biases the same:

```
weights[0], weights[1], weights[2], weights[3], weights[4] = w01, w12, w23, w34, w45
biases[0], biases[1], biases[2], biases[3], biases[4]= b01,b12,b23,b34,b45
```

```
    a0 = np.array([[Z], [N], [A], [Q], [Zd], [Nd]]) # use scaled inputs as initial activations

    a1 = np.array([]) # declaring array
    mult = np.array([np.matmul(w01,a0)]) # multiply weights with activations
    mult = np.reshape(mult, (structure[1],1)) # reshape to make 20x1 matrix
    a1 = sigmoid(mult+b01) # add biases and plug into sigmoid # fix subtract 1 to allow for neg activations

    a2 = np.array([])
    mult = np.array([np.matmul(w12,a1)])
    mult = np.reshape (mult, (structure[2],1))
    a2 = sigmoid (mult + b12)

    a3 = np.array([])
    mult = np.array([np.matmul(w23,a2)])
    mult = np.reshape (mult, (structure[3],1))
    a3 = sigmoid (mult + b23)

    a4 = np.array([])
    mult = np.array([np.matmul(w34,a3)])
    mult = np.reshape (mult, (structure[4],1))
    a4 = sigmoid (mult + b34)

    a5 = np.array([])
    mult = np.array([np.matmul(w45,a4)])
    mult = np.reshape (mult, (structure[5],1))
    a5 = mult + b45

    return a5
```

When I did this, I got the following outputs:

```
Half-Life V: 8.053886135524223s
Half-Life I: 8.053886135524223s
```

(I = Iterative, V = Original) This showed that they had the exact same outputs and therefore were doing the same things. However, to truly test whether this worked, I needed to change the structure and see if it still gave me an output.

When I changed the structure to the following: I received the following output:

```
structure = [6,25,30,30,25,1]
```

```
Half-Life: 13.679712681562432s
```

I then changed the length of structure to the following, and received the following output:

```
structure = [6,10,25,30,25,10,1]
t12 = iterfeed (ZArr[i], NArr[i], AArr[i], QArr[i], Zdist[i], Ndist[i], structure)
print ("Half-Life I: "+ str(t12[0][0]) + "s")
```

```
Half-Life I: 5.834881078276307s
[Finished in 1.4s]
```

[Evidence for Test 2]

This showed that my iterative feedforward method did in fact work for varying lengths and sizes of hidden layers. I was ready to move on to the next step.

One thing I saw which was quite clunky was the way in which I entered the data, as I had to enter each data item individually. For this reason, I decided to amend this and instead have it so the network takes in an array of data items. One issue which may arise is that the data may be entered in the wrong order, however, as the user does not interact with this directly, I do not have to worry about this. I then created a `getIsotope()` method in my data class which returns an array `[Z, N, A, Q, ZDist, NDist]`:

```
def getIsotope (self):
    Isotope = []
    for x in self.data:
        Isotope.append ([x[1], x[3], x[5], x[6], x[2], x[4]])
    return Isotope
```

This then cleaned up my feedforward call:

```
def feedforward (self, data):

    # min-max normalise all values
    Z = (data[0]-min(d.getZ()))/(max(d.getZ())-min(d.getZ()))
    N = (data[1]-min(d.getN()))/(max(d.getN())-min(d.getN()))
    A = (data[2]-min(d.getA()))/(max(d.getA())-min(d.getA()))
    Q = (data[3]-min(d.getQ()))/(max(d.getQ())-min(d.getQ()))
    Zd = (data[4]-min(d.getZDist()))/(max(d.getZDist())-min(d.getZDist()))
    Nd = (data[5]-min(d.getNDist()))/(max(d.getNDist())-min(d.getNDist()))
```

This also means there is less chance for error when entering the data as I won't swap 2 data items around in the call. I was now quite happy with my feedforward method and so I decided to begin the Network class and placing everything I needed into the class.

Object-Oriented Encapsulation

I put my constructor and feedforward method into a Network class which initialises all the weights, biases, activations etc. based on the structure of NN specified by the user:

```
class Network ():

    def __init__ (self, structure, learningrate = 0.01):
        d = Data ()
        self.size = len(structure) # set size equal to number of layers
        self.structure = structure # set structure (n of neurons) to match input

        self.weights = [] # declare weight array
        self.biases = [] # declare bias array
        for x in range (0, (self.size - 1)):
            # initialise weights and biases with random values
            self.weights.append(np.asarray(np.random.rand(structure[x+1],structure[x])))
            self.biases.append(np.asarray(np.random.rand(structure[x+1],1)))

        self.preactive = []

        self.activations = [] # declare activations
        for x in range (0, self.size):
            # initialise activations with zeros
            self.activations.append(np.zeros((structure[x],1)))

        self.learningrate = learningrate
```

```
# iterate through hidden layers using weights and biases (except for last)
for x in range (0, self.size - 1):
    a = self.activations [x]
    mult = np.array([np.matmul(self.weights[x], a)])
    mult = np.reshape(mult, (self.structure[x+1],1))
    presig = mult + self.biases [x]
    self.preactive.append(presig)
    self.activations [x+1] = sigmoid(mult + self.biases[x])

#manually apply last layer weights and biases to avoid sigmoid

af = np.array([])
mult = np.array([np.matmul(self.weights[self.size - 2], self.activations [self.size - 2])])
mult = np.reshape (mult, (self.structure[self.size - 1],1))
af = mult + self.biases[self.size - 2]

self.activations.append ([af])
self.preactive.append ([af])

return af
```

After confirming it was still working at this point[12], that I started on my backpropagation method.

Backpropagation Method

I coded my backpropagation method in a similar way to how I coded my feed-forward method, first doing each thing step-by-step on a 3-layer network before then coding it iteratively.

Step-By-Step

The first step of the backpropagation algorithm is to compute the error in the last layer, which is the (derivative of the cost function with respect to the activation) multiplied by (the derivative of the activation function with respect to z). The last layer activations, however, were not fed through an activation function, and instead were a linear combination of the previous layer activations. Because of this, the derivative of the activation function was 1. The derivative of the cost with respect to the activation is the `activation-target`, which is easy to compute. I added this in:

```
def backprop (target):
    outputError = target - result
    grad = learningrate * outputError
    hidden_T = np.transpose(weights[1])
    weights[1] += np.matmul(hidden_T, grad)  ✕ File "C:\Users\M M Amjad

    hiddenError = np.matmul (outputError, np.transpose(w2))
    grad = learningrate * hiddenError * sigmoid (a1) * (1-sigmoid(a1))
    w1 += np.matmul (grad, np.transpose (a0))
```

I then defined the hidden error according to the maths above and shifted the weights in the last line accordingly. As you can see, I got an error:

```
ValueError: matmul: Input operand 1 does not have enough dimensions (has 0, gufunc core with signature (n?,k),(k,m?)->(n?,m?) requires 1)
```

It was saying that my `grad` vector had a dimensionality of 0. I was curious as to why so I tried to print `grad.shape`:

```
    print (grad.shape)
AttributeError: 'float' object has no attribute 'shape'
```

It was saying, in essence, that grad was not a vector, it was a scalar. I fixed this by adding a line after grad saying `grad = np.array (grad)` which converted grad to an array, and when I ran it, there were no errors, and the weights were shifted from the original. This means my backpropagation was dimensionally correct and so I decided to try this on the 6 layer network I was planning to use.

[12] Note: While looking over the project once I had finished, I noticed a logic error with my activations. I initialised all the activations I needed with 0s, however, my feedforward appended a new activation layer onto the end meaning I had an extra layer. For this reason, I changed the following line:
self.activations.append([af]) → self.activations[-1] = [af] (see end for code block). This does not affect the weights and biases, however, as the activations before this are 0s and it they do not affect the last, additional layer and so the rest of the project does not need to be changed.

6 Layer Step-By-Step Backpropagation

For my backpropagation algorithm I am taking advantage of a Python feature in which you can use negative indexing to iterate through an array backwards (-1 refers to the final item, -2 to the second last etc). To make my life easier, I also defined a sigmoid prime function (derivative of the sigmoid) using lambda functions as follows:

```
sigmoid = lambda x : 1.0/(1.0+np.exp(-x))

sigmoid_prime = lambda z : sigmoid(z)*(1-sigmoid(z))
```

```
# remember, last weight = weights[size-2], last bias = biases [size-2]
# do last weight linearly, do the rest with inverse sigmoid

def backpropagation (activation, target, learningrate):

    global activations
    global biases
    global weights

    error_1 = activations [-1] - target
    error_1 = np.reshape (error_1, (1, structure[-1]))
    deltab = []
    deltab.insert (0, error_1)
    deltaw = []
    deltaw.insert (0, np.matmul(activations[-2], error_1))

    # second last layer

    #print (weights[-1].shape)
    #print (error_1.shape)
    foo = np.matmul (weights[-1].transpose(), error_1)
    error_21 = np.multiply (foo, sigmoid_prime (preactive[-2]))
    error_21 = np.reshape (error_21, (1,structure[-2]))

    deltab.insert (0, error_21)
    deltaw.insert (0, np.matmul(activations[-3], error_21))


    # third last layer
    #print (weights[-2].transpose().shape)
    #print (error_21.shape)
    foo = np.matmul (weights[-2].transpose(), error_21.transpose())
    error_31 = np.multiply (foo, sigmoid_prime (preactive[-3]))
    error_31 = np.reshape (error_31, (1, structure[-3]))
    deltab.insert (0, error_31)
    deltaw.insert (0, np.matmul(activations[-4], error_31))

    # fourth last layer
    foo = np.matmul (weights[-3].transpose(), error_31.transpose())
    error_41 = np.multiply (foo, sigmoid_prime (preactive[-4]))
    error_41 = np.reshape (error_41, (1,structure[-4]))
    deltab.insert (0, error_41)
    deltaw.insert (0, np.matmul(activations[-5], error_41))
```

I then coded my backpropagation algorithm following the above maths and pseudocode. The target was the base 10 logarithm of the half-life.

I frequently kept printing the sizes and shapes of each of the vectors I was multiplying in order to avoid errors in my matrix multiplication being inconsistent. It was also done to check for numpy "quirks" such as the following:

```
# second last layer
foo = np.matmul (weights[-1].transpose(), error_1)
error_21 = np.multiply (foo, sigmoid_prime (preactive[-2]))
print (error_21.shape)
error_21 = np.reshape (error_21, (1,16))
print (error_21.shape)
print (activations [-3].shape)
```

```
(1, 16, 1)
(1, 16)
(16, 1)
```

I continued the algorithm on the left until I had a delta for each of the weights and biases, and then adjusted them using the following loop. I checked the shape and resized due it throwing a similar error to the above:

```
for x in range (0,5):
    print(f"Original Shape: {weights[x].shape}")
    weights[x] = weights[x] + learningrate * deltaw [x]
    weights[x] = np.reshape (weights[x], (structure[x+1], structure[x]))
    print(f"New Shape: {weights[x].shape}")

for b,db in zip (biases, deltab):
    b = b+learningrate*db
```

I then ran my backpropagation algorithm, printing out the original error when feeding forward an isotope and then the error after running the backpropagation a few times. I received the following results:

```
error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error: {error}")

for z in range (0, 3):
    activation = feedforward (isotope[i], structure)
    backpropagation (feedforward (isotope[i], structure), np.log10 (d.getHL()[i]))
    #print (f"{z+1} iterations done")
    error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (error)
```

```
Error: [[13.58972631]]
1 iterations done
2 iterations done
3 iterations done
Error: [[552.78375256]]
```

This seemed strange as it indicated that my error was actually increasing. I had a look over my algorithm and realised the reason for this was that I was increasing my weights and biases by the delta rather than decreasing them (see above). For this reason, I simply changed the + sign to a – sign and reprinted the errors which gave me the following:

```
for x in range (0,5):
    #print(f"Original Shape: {weights[x].shape}")
    weights[x] = weights[x] - learningrate * deltaw [x]
    weights[x] = np.reshape (weights[x], (structure[x+1], structure[x]))
    #print(f"New Shape: {weights[x].shape}")

for b,db in zip (biases, deltab):
    b = b-learningrate*db
```

```
Error: [[11.69336131]]
1 iterations done
2 iterations done
3 iterations done
Error: [[2.07139489]]
[Finished in 1.3s]
```

This was a good sign, as my error was decreasing. I then decided to run the backpropagation for just a few more iterations and see how low I could get my error for a particular training example. I did this with a different isotope:

```
143    for z in range (0, 2000):
Error: [[11.87025293]]
Error: [[3.63638802]]
[Finished in 4.7s]
```

```
143    for z in range (0, 3000):
Error: [[9.98561257]]
Error: [[3.63638802]]
[Finished in 9.4s]
```

This showed that my error was converging to a particular value, which it should not have done. Backpropagation should theoretically reduce the error to 0, and this was not. I initially thought to look at my weights, as I thought the reason this could be happening may be because the weights don't budge past a particular value. When I did this, I realised I did not initialise my weights with negative values as `numpy.random.rand(shape)` initialises with float values between 0 and 1. I thought about just subtracting 1 from the weights and biases but then I came across the `nump.random.uniform` function in which it initialises a matrix with values in a uniform distribution between 2 values. I changed my weights and biases lines to the following:

```
for x in range (0, (size - 1)):
    weights.append(np.asarray(np.random.uniform(-1,1, (structure[x+1],structure[x]))))
    biases.append(np.asarray(np.random.uniform(-1,1, (structure[x+1],1))))
```

This gives me initial values distributed randomly with a uniform distribution between -1 and 1. I then rechecked the error:

```
Error: [[1.7460133]]
Error: [[3.63638356]]
[Finished in 4.4s]
```

All this did was decrease the initial error. I then thought it might be an issue with the algorithm itself, so I tried to find in which iteration/how quickly the error converges by appending the error to an array at each iteration and finding the index of the error:

```
errors = []

for z in range (0, 2000):
    #activation = feedforward (isotope[i], structure)
    backpropagation (feedforward (isotope[i], structure), np.log10 (d.getHL()[i]), 0.01)
    #print (f"{z+1} iterations done")
    error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
    errors.append (error)

print (f"Index of Converged Value: {errors.index(3.63632997)}")  x File "C:\Users\M M Amjad\Documents\[=

Error: [[0.69515337]]
Traceback (most recent call last):
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\BackpropPrototype.py", line 150, in <module>
    print (f"Index of Converged Value: {errors.index(3.63632997)}")
ValueError: 3.63632997 is not in list
[Finished in 6.7s]
```

This said that the exact error which it converged to was not in the list. I then decided to just print the error array to see if anything else was happening:

```
Error: [[5.52755923]]
[array([[4.58913909]]), array([[4.19554979]]), array([[3.98373609]]), array([[3.8585255]]), array([[3.78084367]]), array([[3.73129043]]), array([[3.69913864]]), array([[3.67805211]]),
array([[3.6641264]]), array([[3.65488799]]), array([[3.64874083]]), array([[3.64464246]]), array([[3.64190645]]), array([[3.64007832]]), array([[3.6388561]]), array([[3.63803865]]),
array([[3.63749178]]), array([[3.63712586]]), array([[3.63688099]]), array([[3.63671711]]), array([[3.63660743]]), array([[3.63653402]]), array([[3.63648489]]), array([[3.636452]]),
array([[3.63642999]]), array([[3.63641525]]), array([[3.63640539]]), array([[3.63639879]]), array([[3.63639437]]), array([[3.63639142]]), array([[3.63638944]]), array([[3.63638811]]),
array([[3.63638723]]), array([[3.63638664]]), array([[3.63638624]]), array([[3.63638597]]), array([[3.6363858]]), array([[3.63638568]]), array([[3.6363856]]), array([[3.63638555]]),
array([[3.63638551]]), array([[3.63638549]]), array([[3.63638547]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]),
array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]),
array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638545]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]),
array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]), array([[3.63638546]]),
array([[3.63638546]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]),
array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638547]]), array([[3.63638548]]), array([[3.63638548]]),
array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]),
array([[3.63638548]]), array([[3.63638548]]), array([[3.63638548]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]),
array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]), array([[3.63638549]]),
```

This showed me that it converged incredibly quickly to a value very close to 3.6368549 before slowly eventually reaching that value. At this point I had run out of things to check so I decided to check my backpropagation algorithm against the maths.

When checking against the maths, I realised I had been doing my matrix multiplication the wrong way round the whole time. One of the reasons I didn't spot this earlier was because all 4 of my hidden layers had the same dimensions, so no matter which way you multiplied them, they gave a correctly shaped output. It was my inept multiplication that also caused me to add transposes where I did not need transposes to make everything consistent. I amended the backpropagation to match the maths exactly and ended up with the following:

```
    # error in last layer
    error_l = activation - target # getting error
    error_l = error_l = np.reshape (error_l, (1, structure[-1])) # reshaping error to be array
    deltab.insert (0, error_l)
    deltaw.insert (0, np.matmul(error_l, activations[-2].transpose()))

    # error in second last layer
    foo = np.matmul (weights[-1].transpose(), error_l)
    error_2l = np.multiply (foo, sigmoid_prime (preactive[-2]))
    error_2l = np.reshape (error_2l, (structure[-2], 1))
    deltab.insert (0, error_2l)
    deltaw.insert (0, np.matmul(error_2l, activations[-3].transpose()))

    # error in third last layer
    foo = np.matmul (weights[-2].transpose(), error_2l)
    error_3l = np.reshape (error_2l, (structure[-3], 1))
    deltab.insert (0, error_3l)
    deltaw.insert (0, np.matmul(error_3l, activations[-4].transpose()))

    # error in fourth last layer
    foo = np.matmul (weights[-3].transpose(), error_3l)
    error_4l = np.reshape (error_3l, (structure[-4], 1))
    deltab.insert (0, error_4l)
    deltaw.insert (0, np.matmul(error_4l, activations[-5].transpose()))

    # error in 5th last layer
    foo = np.matmul (weights[-4].transpose(), error_4l)
    error_5l = np.reshape (error_4l, (structure[-5], 1))
    deltab.insert (0, error_5l)
    deltaw.insert (0, np.matmul(error_5l, activations[-6].transpose()))

[Finished in 1.1s]
```

I updated the weights in the same way I did before and rechecked the error:

```
i = 100
isotope = d.getIsotope()

error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error Before: {error}")

for z in range (0, 100):
    activation = feedforward (isotope[i], structure)
    backpropagation (activation, np.log10 (d.getHL()[i]), 0.01)

error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error After: {error}")

Error Before: [[1.49498189]]
Error After: [[0.00197646]]
[Finished in 1.6s]
```

This was promising as my error was much lower than it was previously. I then ran it again for more iterations to see whether or not it still tended towards a particular value:

```
i = 100
isotope = d.getIsotope()

error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error Before: {error}")

for z in range (0, 1000):
    activation = feedforward (isotope[i], structure)
    backpropagation (activation, np.log10 (d.getHL()[i]), 0.01)

error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error After: {error}")

Error Before: [[4.06367428]]
Error After: [[8.8817842e-16]]
[Finished in 2.7s]
```

[Evidence for Test 4]

The error was tending towards 0. This was exactly what I wanted, as the NN is being trained to predict the value of the training example more and more accurately, tending the error closer and closer to 0. My final non-iterative backpropagation algorithm is as follows:

```
def backpropagation (activation, target, learningrate):
    global weights
    global biases

    deltaw = []
    deltab = []

    # error in last layer
    error_1 = activation - target # getting error
    error_1 = error_1 = np.reshape (error_1, (structure[-1], 1)) # reshaping error to be array
    deltab.insert (0, error_1) # error = delta biases
    deltaw.insert (0, np.matmul(error_1, activations[4].transpose())) # weight delta is error propagated backwards

    # error in second last layer
    foo = np.matmul (weights[-1].transpose(), error_1)
    error_2l = np.multiply (foo, sigmoid_prime (preactive[-3]))
    error_2l = np.reshape (error_2l, (structure[-2], 1))
    deltab.insert (0, error_2l)
    deltaw.insert (0, np.matmul(error_2l, activations[3].transpose()))

    # error in third last layer
    foo = np.matmul (weights[-2].transpose(), error_2l)
    error_3l = np.multiply (foo, sigmoid_prime (preactive[-4]))
    error_3l = np.reshape (error_3l, (structure[-3], 1))
    deltab.insert (0, error_3l)
    deltaw.insert (0, np.matmul(error_3l, activations[2].transpose()))

    # error in fourth last layer
    foo = np.matmul (weights[-3].transpose(), error_3l)
    error_4l = np.multiply (foo, sigmoid_prime (preactive[-5]))
    error_4l = np.reshape (error_4l, (structure[-4], 1))
    deltab.insert (0, error_4l)
    deltaw.insert (0, np.matmul(error_4l, activations[1].transpose()))

    # error in 5th last layer
    foo = np.matmul (weights[-4].transpose(), error_4l)
    error_5l = np.multiply (foo, sigmoid_prime (preactive[-6]))
    error_5l = np.reshape (error_5l, (structure[-5], 1))
    deltab.insert (0, error_5l)
    deltaw.insert (0, np.matmul(error_5l, activations[0].transpose()))


    for x in range (0,5):
        weights [x] = weights [x] - (learningrate * deltaw[x])
    for x in range (0,5):
        biases[x] = biases [x] - (learningrate * deltab[x])
```

I then decided that I was ready to start coding it iteratively.

Iterative Backpropagation

I then followed the procedural algorithm above and copied it into my iterative algorithm:

```python
def iterbackprop (activation, target, learningrate):
    global weights
    global biases

    deltaw = []
    deltab = []

    # error in last layer
    error_l = activation - target # getting error
    error_l = error_l = np.reshape (error_l, (structure[-1], 1)) # reshaping error to be array
    deltab.insert (0, error_l) # error = delta biases
    deltaw.insert (0, np.matmul(error_l, activations[4].transpose())) # weight delta is error propagated backwards

    for x in range (0, size - 2): # iterating from 0 to 4
        prop = np.matmul (weights[-(x+1)].transpose(), error_l)
        error_l = np.multiply (prop, sigmoid_prime (preactive[-(x+3)]))
        error_l = np.reshape (error_l, (structure[-(x+2)] , 1))
        deltab.insert (0, error_l)
        print (f"x={x}")
        deltaw.insert (0, np.matmul(error_l, activations[(x-3)].transpose()))  x File "C:\Users\M M Amjad\Docume

    for x in range (0,5):
        weights [x] = weights [x] - (learningrate * deltaw[x])
    for x in range (0,5):
        biases[x] = biases [x] - (learningrate * deltab[x])
```

```
Error Before: [[-0.84655192]]
x=0
x=1
Traceback (most recent call last):
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\BackpropPrototype2.py", line 150, in <module>
    iterbackprop (activation, np.log10 (d.getHL()[i]), 0.01)
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\BackpropPrototype2.py", line 127, in iterbackprop
    deltaw.insert (0, np.matmul(error_l, activations[(x-3)].transpose()))
AttributeError: 'list' object has no attribute 'transpose'
[Finished in 1.5s]
```

As you can see, this gave me an error. I printed out the index values to see which iteration caused the issue. It was when x=3, and it only affected the `deltaw` line. The only thing which depended on x in that line was the activation which it used to multiply the error by. After following the logic through, I realised that I was iterating through the activations the wrong way, going forwards rather than backwards, meaning I reached the end before I should have. I then amended this line by multiplying x-3 by -1 to get 3-x which fixed the program and gave me no errors:

```python
    for x in range (0, size - 2): # iterating from 0 to 4
        prop = np.matmul (weights[-(x+1)].transpose(), error_l)
        error_l = np.multiply (prop, sigmoid_prime (preactive[-(x+3)]))
        error_l = np.reshape (error_l, (structure[-(x+2)] , 1))
        deltab.insert (0, error_l)
        deltaw.insert (0, np.matmul(error_l, activations[(3-x)].transpose()))

    for x in range (0,5):
        weights [x] = weights [x] - (learningrate * deltaw[x])
    for x in range (0,5):
        biases[x] = biases [x] - (learningrate * deltab[x])
```

```
Error Before: [[-0.37588114]]
Error After: [[-4.02455846e-16]]
[Finished in 2.8s]
```

I decided to run it again but for more iterations to see if the error value suddenly changed, but instead I got the following:

```python
for z in range (0, 5000):
    activation = feedforward (isotope[i], structure)
    iterbackprop (activation, np.log10 (d.getHL()[i]), 0.01)

error = feedforward (isotope[i], structure) - np.log10 (d.getHL()[i])
print (f"Error After: {error}")
```

```
Error Before: [[2.18834266]]
Error After: [[0.]]
[Finished in 7.9s]
```

This meant my algorithm was predicting the half-life with perfect accuracy. I then changed the structure to the following and checked again:

```
            deltaw.insert (0, np.matmul(error_l, activations[(3-x)].transpose()))

        for x in range (0,5):
            weights [x] = weights [x] - (learningrate * deltaw[x])
        for x in range (0,5):
            biases[x] = biases [x] - (learningrate * deltab[x])

    structure = [6,16,16,16,16,16,1]
```

```
Error Before: [[0.04354551]]
Traceback (most recent call last):
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\BackpropPrototype2.py", line 149, in <module>
    iterbackprop (activation, np.log10 (d.getHL()[i]), 0.01)
  File "C:\Users\M M Amjad\Documents\[=] A Level\Sixth Form\CS Project\Code\BackpropPrototype2.py", line 126, in iterbackprop
    deltaw.insert (0, np.matmul(error_l, activations[(3-x)].transpose()))
AttributeError: 'list' object has no attribute 'transpose'
[Finished in 1.3s]
```

This threw an error for the same line. I rechecked why I started at layer 3 and realised that this was hardcoded to match my 6 layer structure. I changed 3 to size-3 and this fixed the error:

```
        for x in range (0, size - 2):
            prop = np.matmul (weights[-(x+1)].transpose(), error_l)
            error_l = np.multiply (prop, sigmoid_prime (preactive[-(x+3)]))
            error_l = np.reshape (error_l, (structure[-(x+2)] , 1))
            deltab.insert (0, error_l)
            deltaw.insert (0, np.matmul(error_l, activations[((size-3)-x)].transpose()))

        for x in range (0,5):
            weights [x] = weights [x] - (learningrate * deltaw[x])
        for x in range (0,5):
            biases[x] = biases [x] - (learningrate * deltab[x])

    structure = [6,16,16,16,16,16,1]
```

```
Error Before: [[1.77903619]]
Error After: [[5.66213743e-15]]
[Finished in 8.2s]
```

I also realised that me starting at activations[4] for my first layer error was hardcoded and so amended that line as well later on to the following:

```
deltaw.insert (0, np.matmul(error_l, self.activations[self.size-2].transpose()))
```

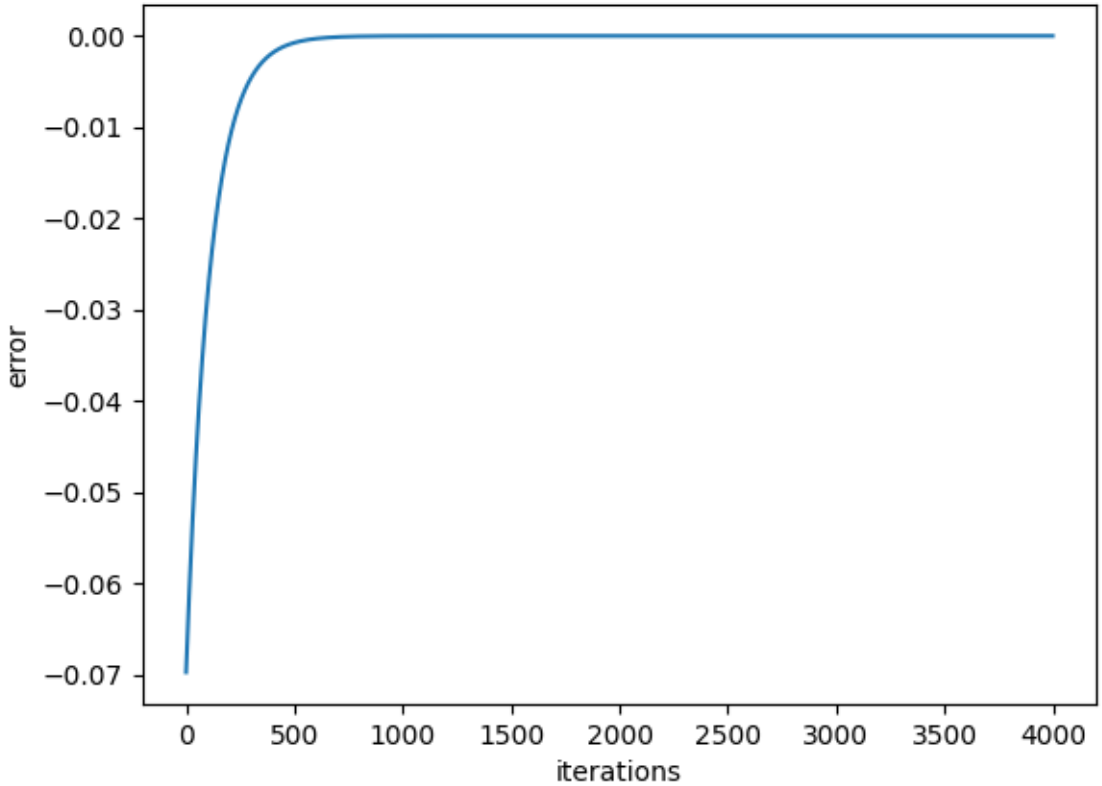
When I then ran my backpropagation iteratively, however, I ran into a similar issue I had earlier. The error was converging:

```
    print (f"Error Before Backpropagation: {net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])}")
    for x in range (0, 4000):
        activation = net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])
        net.backpropagation (activation, np.log10 (d.getHL()[i]), 0.01)

    error = net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])
    print (f"Error After: {error}")
```

```
Error Before Backpropagation: [[2.72776958]]
Error After: [[-0.7212464]]
[Finished in 11.2s]
```

I decided to use the same strategy as before of appending to an errors array and seeing where it starts converging, but this time I decided to plot it to make it easier to see. I added the following:

This gave me the following output:



This showed me that my error was in fact tending towards 0. This was contrary to what the error value I calculated suggested. I decided to then print the final error value and received the following output:

```
print (f"Error Before Backpropagation: {net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])}")
for x in range (0, 4000):
    activation = net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])
    net.backpropagation (activation, np.log10 (d.getHL()[i]), 0.01)
    error = activation - np.log10 (d.getHL()[i])
    errors.append(float(error))
    if x == 3999:
        print (f"Final Error in BackProp: {error}")

error = net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])
print (f"Error After: {error}")
```

```
Error Before Backpropagation: [[-0.61139538]]
Final Error in BackProp: [[1.48769885e-14]]
Error After: [[-0.7212464]]
[Finished in 11.3s]
```

This showed the error was actually going towards 0, but the error after was something completely different.

I decided to check the actual values of my predictions and target by printing them and received the following:

```python
errors = []

print (f"Error Before Backpropagation: {net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])}")
for x in range (0, 4000):
    activation = net.feedforward (isotope[i]) - np.log10 (d.getHL()[i])
    net.backpropagation (activation, np.log10 (d.getHL()[i]), 0.01)
    error = activation - np.log10 (d.getHL()[i])
    errors.append(float(error))
    if x == 3999:
        print (f"Final Error in BackProp: {error}")
        print (f"Log 10 of Predicted Half Life = {activation}")
        print (f"Log 10 of Actual Half Life = {np.log10 (d.getHL()[i])}")

error = (net.feedforward (isotope[i]) - np.log10 (d.getHL()[i]))
print (f"Error After: {error}")
```

```
Error Before Backpropagation: [[0.94118985]]
Final Error in BackProp: [[7.10542736e-15]]
Log 10 of Predicted Half Life = [[-0.7212464]]
Log 10 of Actual Half Life = -0.721246399047171
Error After: [[-0.7212464]]
[Finished in 17.4s]
```

This showed me that instead of printing the error, Python was printing the actual half-life predictions (it was negative as I am training to find the base 10 log of the half-life). This was why the error seemed to converge. This didn't matter, however, as my backpropagation itself was working. This meant that I had all the tools I needed to train and test my network. It was ready to implement.

Appending to Class

```python
def backpropagation (self, activation, target):

    deltaBiases = [] # define arrays for change in biases
    deltaWeights = [] # define arrays for change in weights

    learningrate = self.learningrate

    # error in last layer
    error_l = activation - target # getting error
    error_l = np.reshape (error_l, (self.structure[-1], 1)) # reshaping error to be array
    deltaBiases.insert (0, error_l) # error = delta biases
    deltaWeights.insert (0, np.matmul(error_l, self.activations[self.size-2].transpose())) # weight delta is error propagated backwards

    for x in range (0, self.size - 2):
        prop = np.matmul (self.weights[-(x+1)].transpose(), error_l) # propagate error backwards
        error_l = np.multiply (prop, sigmoid_prime (self.preactive[-(x+3)])) # hadamard product with preactivationss
        error_l = np.reshape (error_l, (self.structure[-(x+2)] , 1)) # reshape because numpy is peculiar
        deltaBiases.insert (0, error_l) # error = delta bias so add to start
        deltaWeights.insert (0, np.matmul(error_l, self.activations[((self.size-3)-x)].transpose())) # add delta weights

    for x in range (0,self.size-1):
        self.weights [x] = self.weights [x] - (learningrate * deltaWeights[x]) # adjust each weight by delta weight
        self.biases[x] = self.biases [x] - (learningrate * deltaBiases[x]) # adjust each bias by delta bias
```

Training and Testing the Network

I defined a simple subroutine to generate all the test and training sets I would need and added this to my network.py document but not my network class:

```python
def getSets ():

    isotopes = d.getIsotope()
    halflives = d.getHL()

    numTrain = len(isotopes)*8//10 # number of training isotopes = len(dataset) * 0.8
    numTest = len(isotopes)-numTrain # number of testing is what remains

    totalNums = np.arange(len(isotopes)) # all the possible indices I can use for isotopes

    randomNums = np.random.choice (len(isotopes), numTrain, replace = False) # replace: whether or not a sample is returned to the sample pool
    testingNums = np.delete(totalNums, randomNums) # define testing indices to be whatever is left after getting rid of training

    trainSet = []
    trainLabels = []

    for x in range (numTrain):
        trainSet.append (isotopes[randomNums[x]])
        trainLabels.append (np.log10(halflives[randomNums[x]]))

    testSet = []
    testLabels = []

    for x in range (numTest):
        testSet.append (isotopes[testingNums[x]])
        testLabels.append (np.log10(halflives[testingNums[x]]))

    model = d.getModel()
    errors = []

    for x in range (numTest):
        actual = np.log10(halflives[testingNums[x]])
        models = np.log10(model[testingNums[x]])
        error = models - actual
        errors.append(error**2)

    sigma =  (np.sum(errors)**(1/2)) / (numTest)
    print (f"Statistical Model Error: {sigma}")

    return trainSet, trainLabels, testSet, testLabels
```

I also made it print the error of the statistical model so that I could make an easy comparison between my model and the test model. I defined training and testing methods in my class as follows:

```python
    def train (self, dataset, targets, epochs):
        for x in range (epochs): # for the specified amount of epochs
            #print (f"Epoch {x+1}")
            for y in range (0, len(dataset)): # for each value in the dataset
                activation = self.feedforward (dataset[y]) # feed it forward
                self.backpropagation (activation, targets[y]) # backpropagate against the target

    def evaluate (self, dataset, targets):
        predictions = [] # define array for predictions
        errors = [] # define array for errors
        for isotope in dataset:
            prediction = self.feedforward (isotope)
            predictions.append(prediction) # add predictions to array of predictions
        for x in range (len(predictions)):
            error = predictions[x] - targets[x] # calculate errors
            errors.append (error**2) # append error^2 to array
        stddev =  (np.sum(errors)**(1/2)) / len(dataset) # sigma = (1/n)*(sum of (errors)^2)^(1/2)
        return float(stddev)
```

I then made sure it worked by running the following code:

```python
trainSet, trainLabels, testSet, testLabels = getSets ()

print ("Training...")
net.train(trainSet, trainLabels, epochs = 100)
print ("Training Complete")
stddev = net.evaluate(testSet, testLabels)
print (f"Standard Deviation: {stddev}")
```

I received the following output:

```
Statistical Model Error: 0.08317506780411073
Training...
Training Complete
Standard Deviation: 0.074377018577843
[Finished in 23.1s]
```

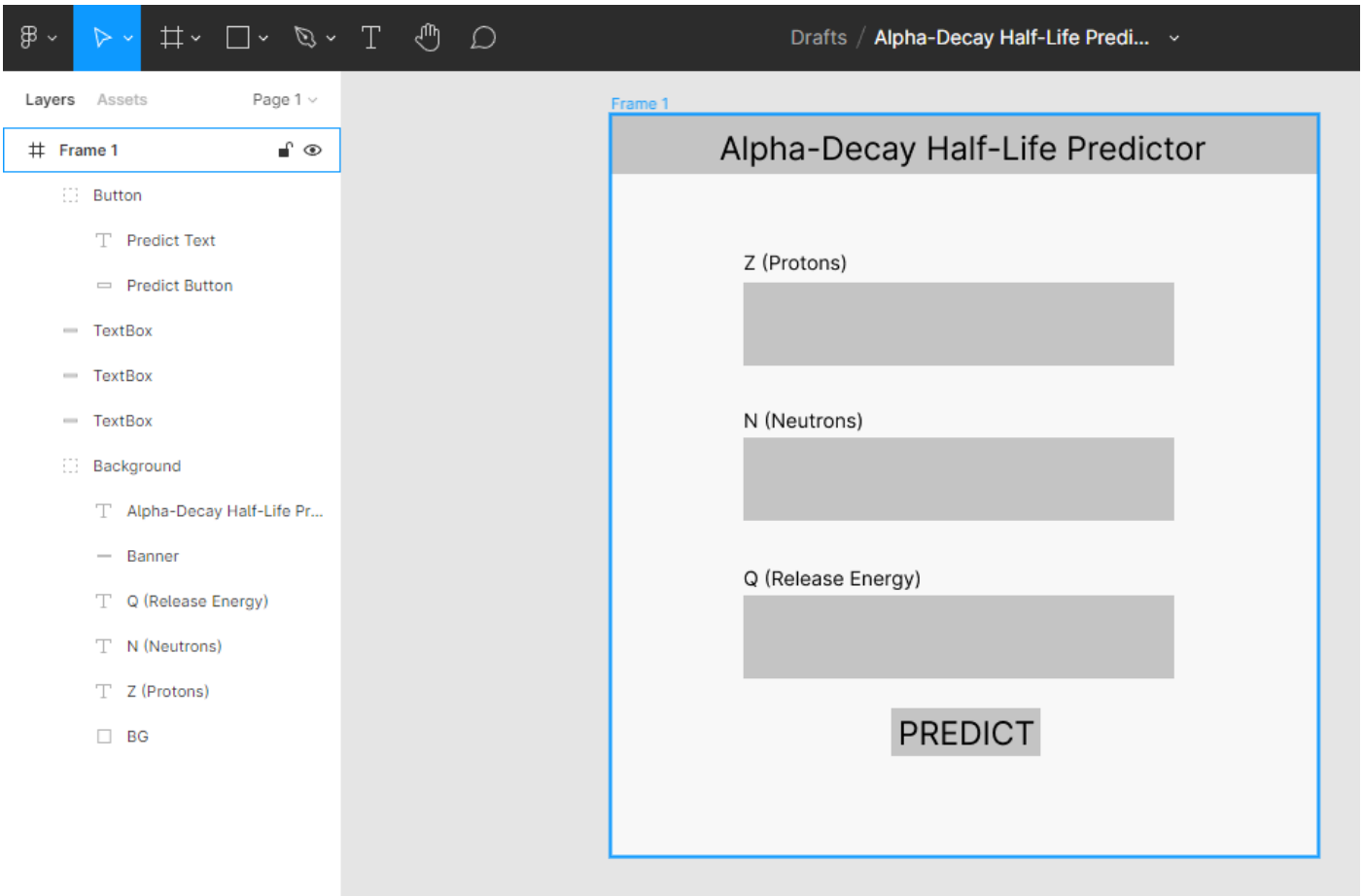
This indicated that my model worked and it achieved a better score than the statistical model! This was also only with 100 epochs. This indicated that my network was completely up and running so at this point, I decided to work on my UI, as the main back-end of my project was done.

User Interface

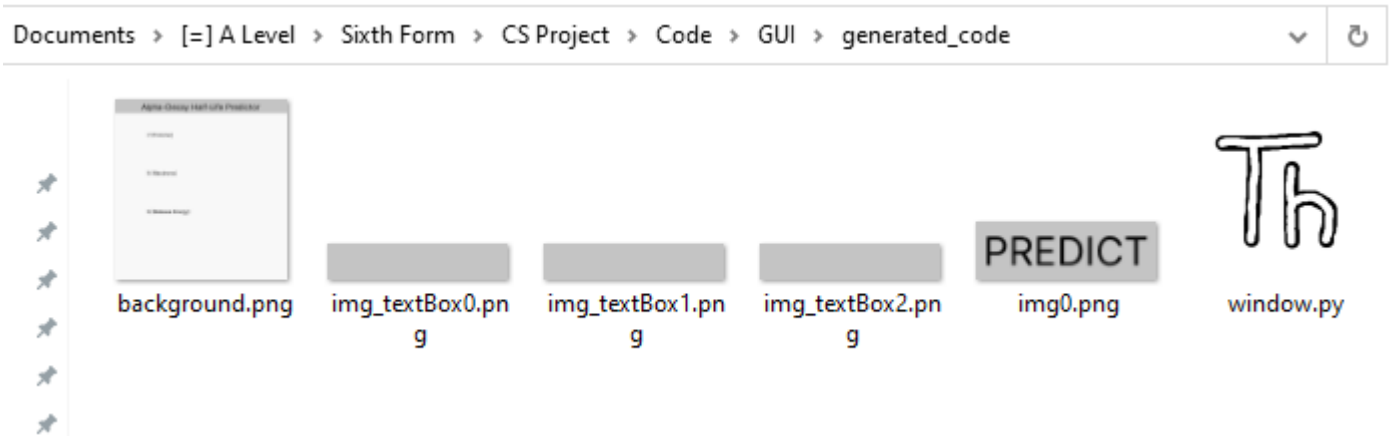
I created my user interface using a tool called figma[13], which allows you to create templates for UIs and feed them into other programs. I did this because I also found on GitHub[14], a tool which takes figma templates and renders them in tkinter. I decided this would be a valid way of creating my UI as my UI is very simple, and this also saves time.

Input Space

I created my template on figma as follows:



I defined the buttons and text boxes on the left. I then fed this into the GUI-Designer software I found on GitHub, which generated for me the following folder containing all the back-end I needed for the UI:



I then went into the window.py file and added all the functionality I needed, as this was not added by the GUI designer.

[13] [https://www.figma.com/]
[14] [https://github.com/TestTest4253/GUI-Designer]

Adding Functionality

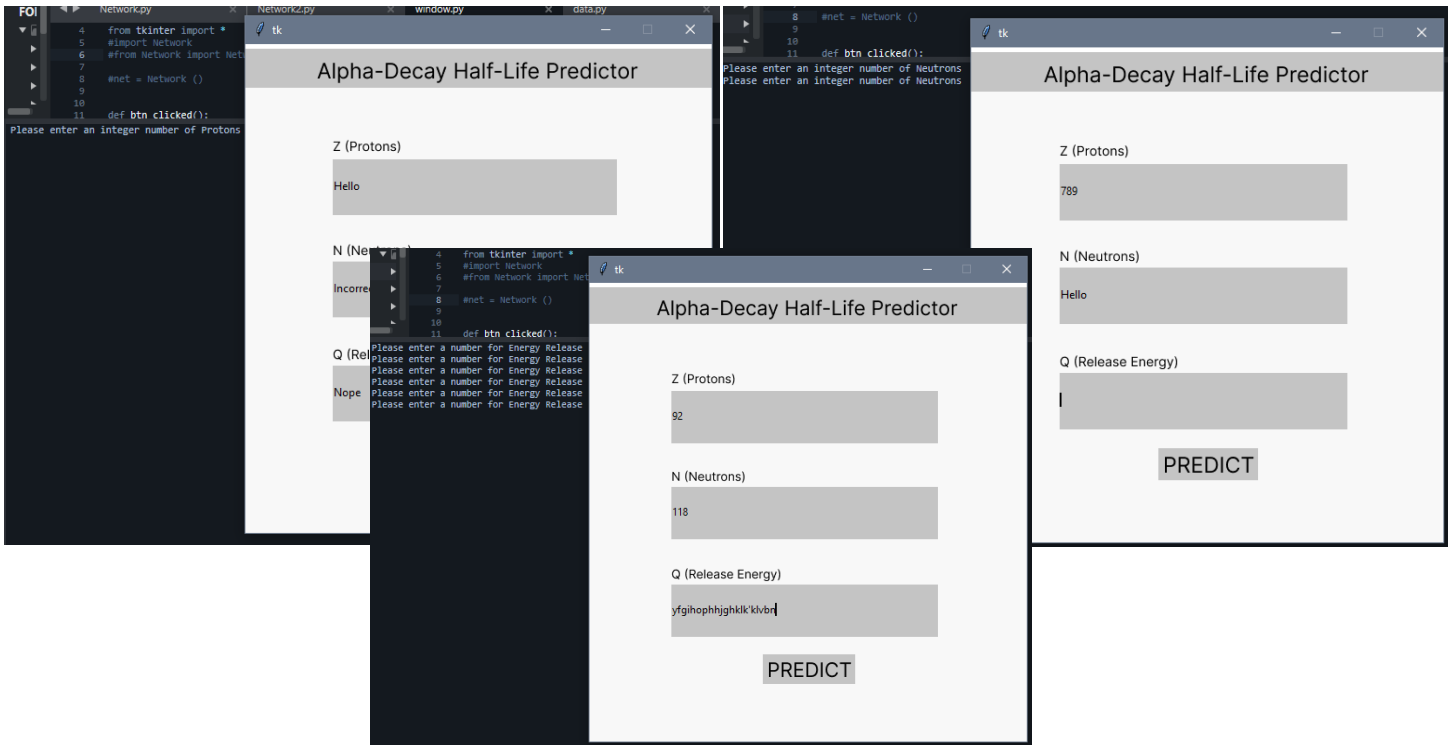
The main functionality of my UI rests in what happens when the user presses the PREDICT button. When this is pressed, the software needs to read the content of each of the boxes and feed them into my neural network. This also needs to account for the user inputting incorrect data. I decided that try, except statements would be the best way to implement this. The button was defined as follows:

```
b0 = Button(
    image = img0,
    borderwidth = 0,
    highlightthickness = 0,
    command = btn_clicked,
    relief = "flat")
```

This called the btn_clicked function on click. I then defined the btn_clicked function as follows:

```
def btn_clicked():
    data = []

    try:
        Z = int (entry0.get())
    except ValueError:
        print ("Please enter an integer number of Protons")
        return

    try:
        N = int (entry1.get())
    except ValueError:
        print ("Please enter an integer number of Neutrons")
        return

    try:
        Q = float (entry2.get())
    except ValueError:
        print ("Please enter a number for Energy Release")
        return

    A = Z+N
    Zdist = min ([abs(Z-28), abs(Z-50), abs(Z-82), abs(Z-126)])
    Ndist = min ([abs(N-28), abs(N-50), abs(N-82), abs(N-84), abs(N-126)])
    data.append(Z)
    data.append(N)
    data.append(A)
    data.append(Q)
    data.append(Zdist)
    data.append(Ndist)

    print (data)
```

This does the following:



[Evidence for Test i]

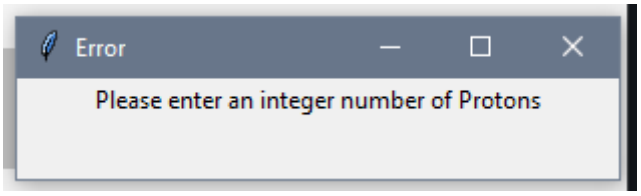
The software printed to the console for every incorrect input, meaning that the software can correctly recognise them. I now need a way of displaying the prompts to the user. For this, I decided to open a new window which had the prompt on. For this, I defined a new window class[15] which took in string as a parameter that defined the error text:

```
class NewWindow(Toplevel):

    def __init__(self, master = None, texts = "Error"):

        super().__init__(master = master)
        self.title("Error")
        self.geometry("300x50")
        label = Label(self, text = texts)
        label.pack()
```

I then called this class in each of my except statements with the text defined for each of them:

```
def btn_clicked():
    data = []

    try:
        Z = int (entry0.get())
    except ValueError:
        print ("Please enter an integer number of Protons")
        entry0.delete (0,END)
        NewWindow (window, "Please enter an integer number of Protons")
        return

    try:
        N = int (entry1.get())
    except ValueError:
        print ("Please enter an integer number of Neutrons")
        entry1.delete(0,END)
        NewWindow (window, "Please enter an integer number of Neutrons")
        return

    try:
        Q = float (entry2.get())
    except ValueError:
        print ("Please enter a number for Energy Release")
        entry2.delete(0,END)
        NewWindow (window, "Please enter a number for Energy Release")
        return
```

I also used a feature called entry.delete(0,END) which deletes the contents of the entry box whenever the user enters something incorrect so that they do not have to do this manually:

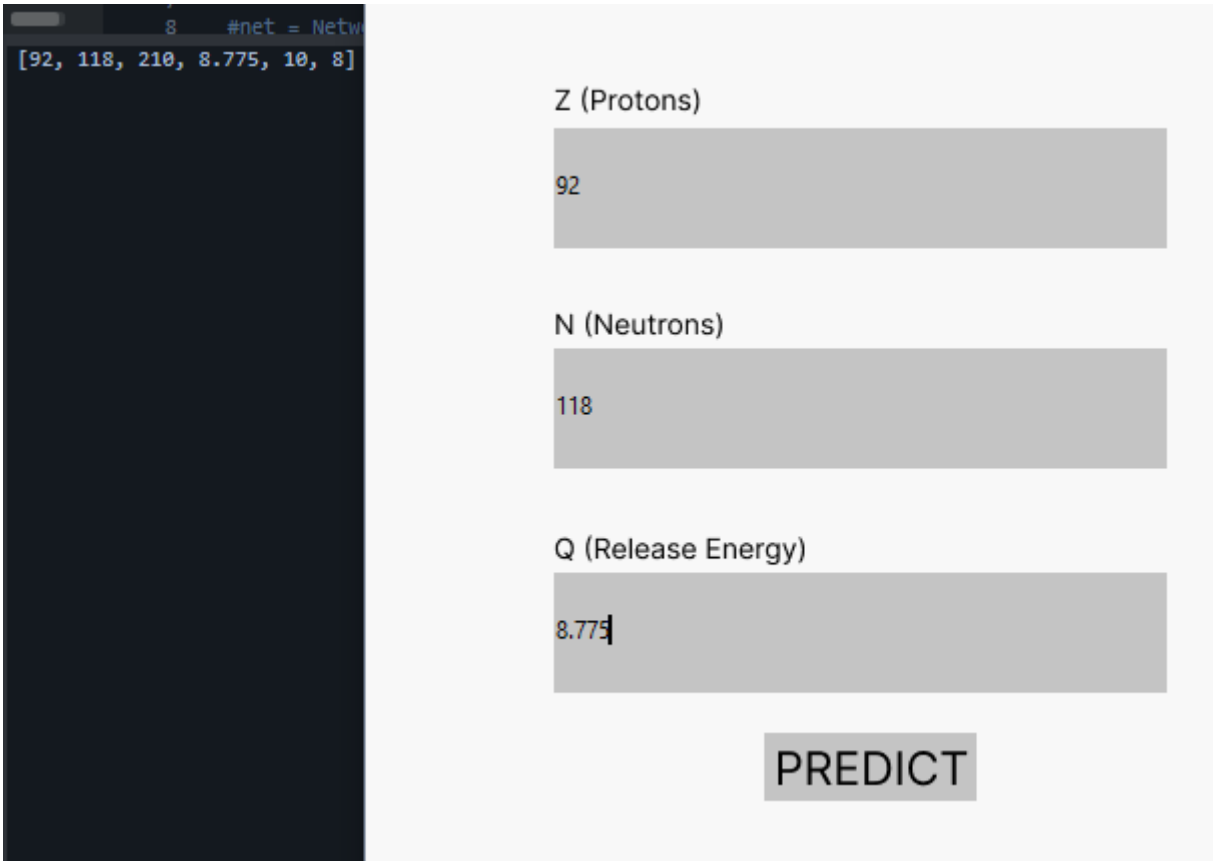


A video of this working is at the end of the project. Once the user entered completely correct data, I defined the rest of the variables based on their input (nucleon number, ZDist and NDist) and stored them in an array:

```
A = Z+N
Zdist = min ([abs(Z-2), abs(Z-8), abs(Z-20), abs(Z-28), abs(Z-50), abs(Z-82), abs(Z-126)])
Ndist = min ([abs(N-2), abs(N-8), abs(N-20), abs(N-28), abs(N-50), abs(N-82), abs(N-84), abs(N-126)])
data.append(Z)
data.append(N)
data.append(A)
data.append(Q)
data.append(Zdist)
data.append(Ndist)
```

This gave the following output when I printed `data`:

[15] [https://www.geeksforgeeks.org/open-a-new-window-with-a-button-in-python-tkinter/]



[Evidence for Test ii]

I now had to calculate the half-life and then finally make a window to display the half-life, which I will add when I amalgamate the front-end and back-end.

Amalgamating Front-End and Back-End

The last part was to combine the front-end and back-end. As all the network has to do to make a prediction is feedforward an input through the specified weights and biases, I determined the best way to implement this would be to store the states of my trained weights and biases and feed them forward locally. I would first have to train the network to a suitable standard, and then store the weights and biases using a joblib dump. I can then load this dump locally and define my weights and biases from it.

To implement this, I first tested to see how the joblib dump would work. I did this by generating a small ordered array using `numpy.arange()` and dumping that using joblib to a file called test1:

```
import numpy as np
import joblib

numbers = np.arange (15)
joblib.dump(numbers, "test1")
```

test1	15/04/2022 11:15	File	1 KB

I then loaded this and printed the 2 arrays:

```
import numpy as np
import joblib

numbers = np.arange (15)
joblib.dump(numbers, "test1")
numbers2 = joblib.load ("test1")

print (numbers)
print(numbers2)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[Finished in 1.2s]
```

[Evidence for Test 6]

The 2 arrays were the same. This means I could use joblib for my weights and biases and it would return the same arrays. I then defined a Network.save() method as follows:

```
def save (self):
    joblib.dump(self.weights, "weights")
    joblib.dump(self.biases, "biases")
```

This will allow me to save the weights and biases with filenames “weights” and “biases” respectively if I found them suitable. I then had to train the network to a suitable standard. I defined a suitable standard as being better than the statistical model on 3 separate randomly generated test sets. To determine this, I set up the following routine:

```
trainSet, trainLabels, testSet, testLabels, stddevModel = getSets ()

net = Network ([6,16,20,20,16,1])

print ("Training...")
net.train(trainSet, trainLabels, epochs = 500) # train network
print ("Training Complete")

print ("Round 1")
stddevNetwork = net.evaluate(testSet, testLabels) # evaluate network on given test set
print (f"Network Deviation: {stddevNetwork}") # print net score
print (f"Stat Model Deviation: {stddevModel}") # print model score

if stddevNetwork < stddevModel: # if net is better than model
    print("Round 2") # round 2
    trainSet, trainLabels, testSet, testLabels, stddevModel = getSets () # fresh sets
    stddevNetwork = net.evaluate(testSet, testLabels) # evaluate using already trained weights and biases
    print (f"Network Deviation: {stddevNetwork}") # print net score
    print (f"Stat Model Deviation: {stddevModel}") # print model score

    if stddevNetwork < stddevModel:
        print("Round 3")
        trainSet, trainLabels, testSet, testLabels, stddevModel = getSets () # repeat above
        stddevNetwork = net.evaluate(testSet, testLabels)
        print (f"Network Deviation: {stddevNetwork}")
        print (f"Stat Model Deviation: {stddevModel}")

        if stddevNetwork < stddevModel: # if net better than model 3 times
            print ("Saving")
            net.save() # save weights and biases
```

I also modified the structure to have more neurons in the central 2 hidden layers as this would hopefully also increase the accuracy. I upped the epoch number to 500 to better train the network. I then set this running, however, it was not very successful. It rarely got to Round 3 and when it did it rarely saved. Due to this, after around 10 attempts, I upped the epoch number to 1000. On my 2nd attempt, I got the following:

```
Training...
Training Complete
Round 1
Network Deviation: 0.07139960162705565
Stat Model Deviation: 0.08820546738830637
Round 2
Network Deviation: 0.0768375723413901
Stat Model Deviation: 0.09222547068959369
Round 3
Network Deviation: 0.06477328713484788
Stat Model Deviation: 0.08070372707862815
Saving
[Finished in 60.3s]
```

This was, looking at the results, a very good network. It had a difference of nearly 0.02 on each round compared to the statistical mode, meaning the network was outperforming it by a fair way on each test. Because of this, I decided to stop here and take these weights and biases. The reason I did not want to go for many more epochs was because I did not want to run the risk of overfitting the network to the current dataset. This is when the network has been

trained so much that it starts to spot patterns in the random noise of the training set. It then overly fits its weights and biases to the training set, such that it performs less effectively when being used on the testing set. For fun, I ran the training algorithm for 10,000 epochs and got the following results:

```
Training...
Training Complete
Round 1
Network Deviation: 0.0660569986370034
Stat Model Deviation: 0.08288146937820234
Round 2
Network Deviation: 0.05556136776135095
Stat Model Deviation: 0.08279958484413724
Round 3
Network Deviation: 0.05161126429918567
Stat Model Deviation: 0.09256688260351674
Saving
[Finished in 587.5s]
```

Although this has an even lower standard deviation from the test set than my current model, I have decided not to use these weights and biases to avoid running the risk of overfitting the network.

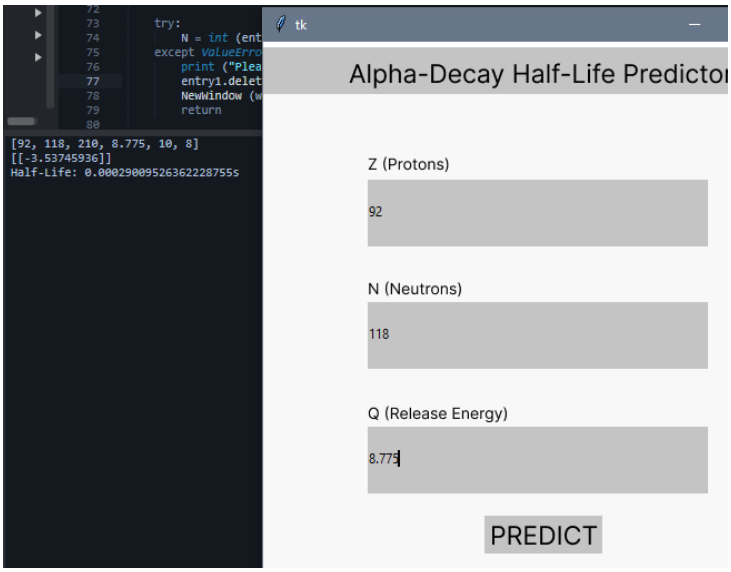
The net.save did its job and the weights and biases were saved in the local directory of my GUI. I then added the following code to my GUI:

```
weights = joblib.load("weights")
biases = joblib.load ("biases")
activations = []

structure = [6,16,20,20,16,1]
size = len(structure)
for x in range (size):
    activations.append(np.zeros((structure[x],1)))

def feedforward (data):
    # min-max normalise all values
    Z = (data[0]-min(d.getZ()))/(max(d.getZ())-min(d.getZ()))
    N = (data[1]-min(d.getN()))/(max(d.getN())-min(d.getN()))
    A = (data[2]-min(d.getA()))/(max(d.getA())-min(d.getA()))
    Q = (data[3]-min(d.getQ()))/(max(d.getQ())-min(d.getQ()))
    Zd = (data[4]-min(d.getZDist()))/(max(d.getZDist())-min(d.getZDist()))
    Nd = (data[5]-min(d.getNDist()))/(max(d.getNDist())-min(d.getNDist()))

    activations [0] = np.array([[Z], [N], [A], [Q], [Zd], [Nd]])


    # iterate through hidden layers using weights and biases (except for last)
    for x in range (0, size - 1):
        a = activations [x]
        mult = np.array([np.matmul(weights[x], a)])
        mult = np.reshape(mult, (structure[x+1],1))
        presig = mult + biases [x]
        activations [x+1] = sigmoid(mult + biases[x])

    #manually apply last layer weights and biases to avoid sigmoid
    af = np.array([])
    mult = np.array([np.matmul(weights[size - 2], activations [size - 2])])
    mult = np.reshape (mult, (structure[size - 1],1))
    af = mult + biases[size - 2]
    activations.append ([af])
    return af
```

In my btn_clicked routine, I added the following at the end:

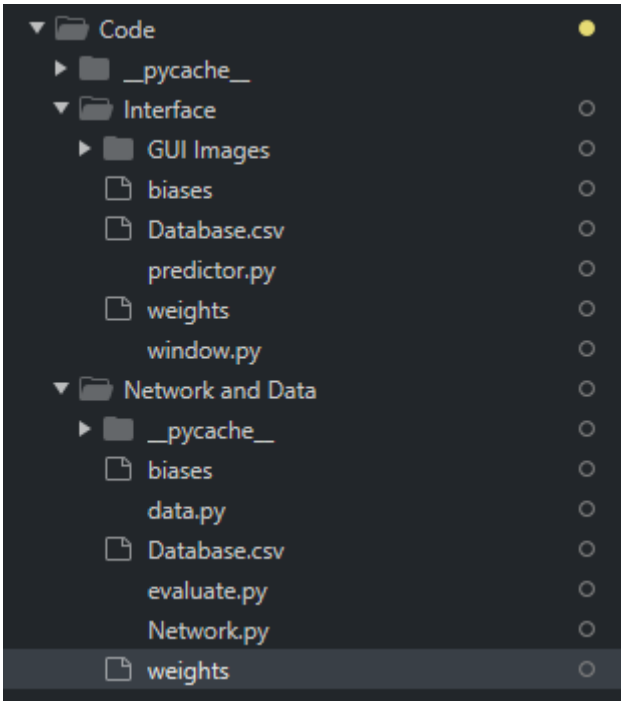
```
A = Z+N
Zdist = min ([abs(Z-2), abs(Z-8), abs(Z-20), abs(Z-28), abs(Z-50), abs(Z-82), abs(Z-126)])
Ndist = min ([abs(N-2), abs(N-8), abs(N-20), abs(N-28), abs(N-50), abs(N-82), abs(N-84), abs(N-126)])
data.append(Z)
data.append(N)
data.append(A)
data.append(Q)
data.append(Zdist)
data.append(Ndist)

print (data)
loghalflife = feedforward (data)
print (loghalflife)
halflife = np.float_power (10, loghalflife)
print (f"Half-Life: {float(halflife)}s")
```

When I then ran the code, I received the following output for this input:



[Evidence for Test iii]

This showed me that my predictor was working. The last thing I had to do was just display this on the user's screen, however, before this, I decided to clean up my code directories. Until this point, things like my network class and database were in a folder with old prototypes and many different things were amalgamated into single files. As my project was nearing completion, I thought that now would be a good time to organise my source code files. I organised them into the following tree:



The plan is to have my window.py call the predictor.py which will hold the feedforward method. This feedforward method will import the data it needs from data.py and return a value back to window.py which will display it. To do this, I have to play with the directories. I have to be careful, however, not to hardcode this to only work with my file paths. I added the following to my predictor.py to allow it to import data.py:

```
import os
import sys

data_dir = os.path.abspath (os.path.join("..","Network and Data"))
# find directory to current file
# go to parent directory ("..")
# from parent, add "Network and Data" to the path
sys.path.insert(1, data_dir) # add this path to system path

from data import Data
d = Data()
```

I then placed the feedforward into my predictor.py:

```python
sigmoid = lambda x : 1.0/(1.0+np.exp(-x))

weights = joblib.load("weights")
biases = joblib.load ("biases")
activations = []

structure = [6,16,20,20,16,1]
size = len(structure)
for x in range (size):
    activations.append(np.zeros((structure[x],1)))

def feedforward (data):
    # min-max normalise all values
    Z = (data[0]-min(d.getZ()))/(max(d.getZ())-min(d.getZ()))
    N = (data[1]-min(d.getN()))/(max(d.getN())-min(d.getN()))
    A = (data[2]-min(d.getA()))/(max(d.getA())-min(d.getA()))
    Q = (data[3]-min(d.getQ()))/(max(d.getQ())-min(d.getQ()))
    Zd = (data[4]-min(d.getZDist()))/(max(d.getZDist())-min(d.getZDist()))
    Nd = (data[5]-min(d.getNDist()))/(max(d.getNDist())-min(d.getNDist()))

    activations [0] = np.array([[Z], [N], [A], [Q], [Zd], [Nd]])


    # iterate through hidden layers using weights and biases (except for last)
    for x in range (0, size - 1):
        a = activations [x]
        mult = np.array([np.matmul(weights[x], a)])
        mult = np.reshape(mult, (structure[x+1],1))
        presig = mult + biases [x]
        activations [x+1] = sigmoid(mult + biases[x])

    #manually apply last layer weights and biases to avoid sigmoid
    af = np.array([])
    mult = np.array([np.matmul(weights[size - 2], activations [size - 2])])
    mult = np.reshape (mult, (structure[size - 1],1))
    af = mult + biases[size - 2]
    activations.append ([af])
    return af
```

I then imported this as ff for feedforward in my window.py:

```python
from pathlib import Path
designPath = Path ("GUI Images/")
import predictor as ff
from tkinter import *
import numpy as np
```

I then called this in my btn_click method:

```python
loghalflife = ff.feedforward (data)
halflife = np.float_power (10, loghalflife)
```

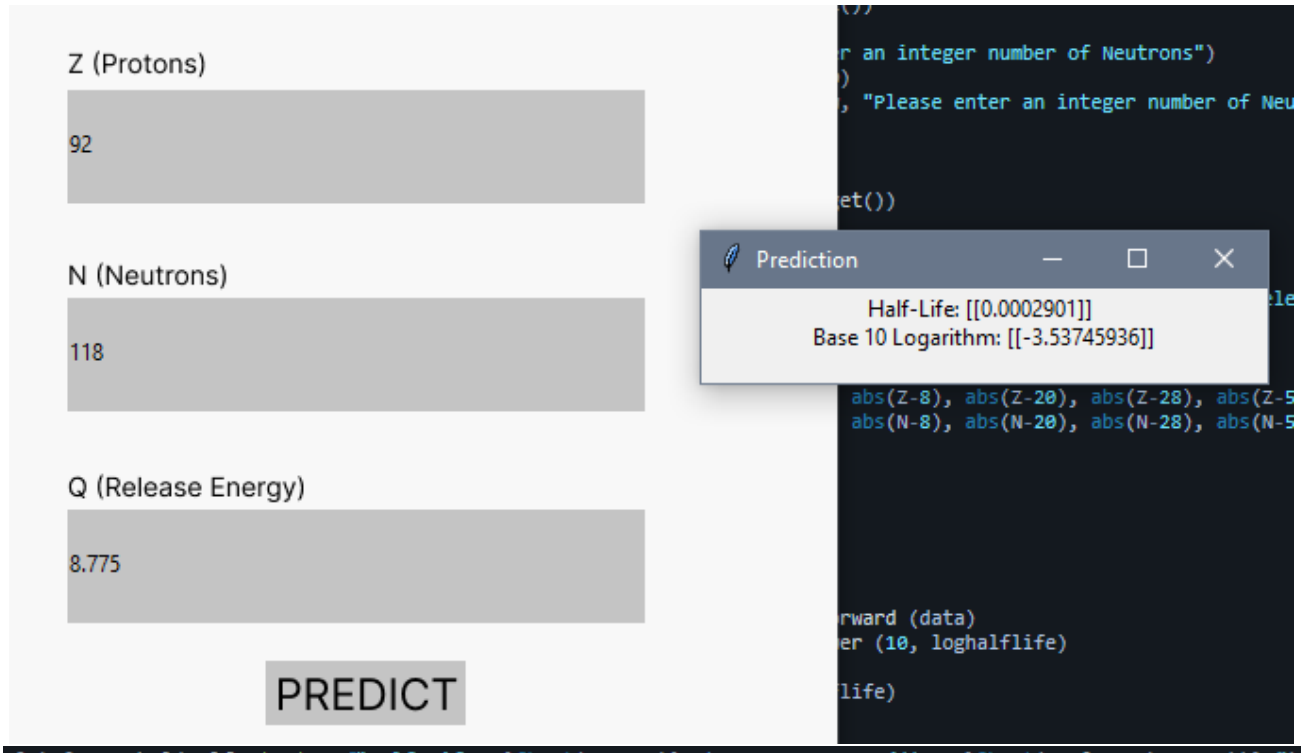
I then also defined a new class for my answer window, which will display the answer of the half-life to the user:

```python
class answindow (Toplevel):

    def __init__ (self, master = None, answer = "Error"):
        super().__init__(master = master)
        self.title("Prediction")
        self.geometry("300x50")
        label = Label(self, text = f"Half-Life: {answer} \nBase 10 Logarithm: {np.log10(answer)}")
        label.pack()
```

I set the default answer to Error as I wanted to know if I hadn't passed an answer to the box. I then created an answer window with my predicted half-life:

```python
answindow (window, halflife)
```

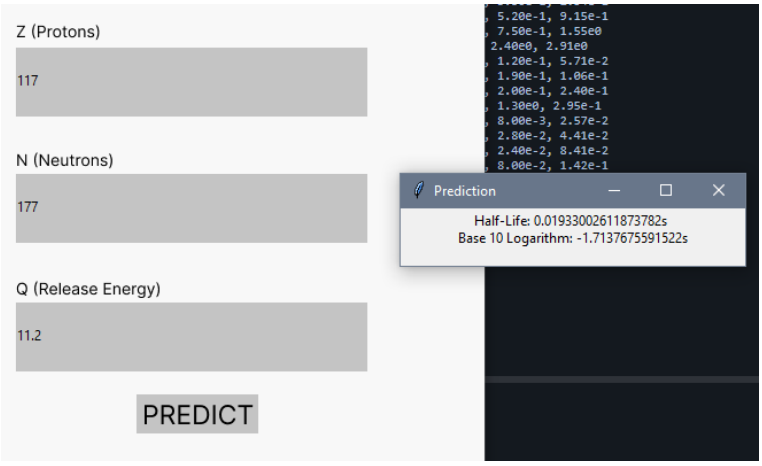
When I inputted everything correctly, it gave me the following output:




```
label = Label(self, text = f"Half-Life: {float(answer)}s \nBase 10 Logarithm: {float(np.log10(answer))}s")
label.pack()
```

This looked ugly so I casted them as floats and added “s” on the end to signify seconds:

This was the output:



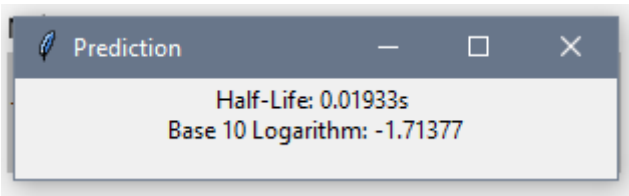
[Evidence for Test v]

I realised that taking the exponent of the half-life and re-logging it will introduce more inaccuracies in my output so I decided to amend this by printing the base 10 log as it was given:

```
loghalflife = ff.feedforward (data)
#halflife = np.float_power (10, loghalflife)

answindow (window, loghalflife)
```

I then changed the output text to the following and rounded to 5 decimal places to make it look better:

```
label = Label(self, text =
    f"Half-Life: {round(float(np.float_power(10, answer)), 5)}s \nBase 10 Logarithm: {round(float(answer), 5)}")
label.pack()
```



The UI was able to output a predicted half-life successfully. The prediction is based on my neural network which was trained via backpropagation to find the base-10 logarithm of alpha-decay half-lives of multiple radioactive isotopes. By saving the states of the weights and biases, I was able to save the state of my trained network and therefore feedforward new input values for new predictions.

This demonstrated that my UI was functional and I was now ready to evaluate the success of my project. Before this, however, I asked my stakeholders for feedback in regards to the GUI.

User Feedback and Updates

I sent my GUI to the stakeholder, along with asking for ratings out of 5 and comments for the following criteria:

- Simplicity: 5/5
    - The GUI is simple and easy to use, it is clear what is wanted from the user and where it is to be entered, and the lack of too many options and boxes make it easy to use
- Understandability: 5/5
    - The software is intuitive to use as the labels for each of the text boxes and the button are very clear in what they are prompting you to do. The prompts for incorrect data are also clear in what they want the user to do as well.
- Design: 4/5
    - The design is clean and simple although could have looked very slightly more polished.
- Other Comments?
    - None, see above.

Evaluation

Success Criteria

Essential Success Criteria

Criterion	Criterion Met?	Justification of Previous Column	Plans for Future Development
Predictions of the neural network must have better accuracy than the statistical model	Yes - Fully	Evidenced by my network giving a lower standard deviation than the statistical model on 3 separate occasions with random test sets (pg. 42)	There are no plans to update the neural network or how it functions in future development
The network is able to account for updates to the database of isotopes	Yes – Fully	The databasing aspect is completely modular, and none of it is hard-coded, meaning that it is able to account for any additions or removals from the database (pg. 19)	There are no plans to update how the database is managed in future development
User is able to easily navigate the GUI	Yes – Fully	Evidenced by my stakeholder feedback (pg. 47)	Update the GUI based on future developments to allow the GUI to be usable with new features

Desirable Success Criteria

Criterion	Criterion Met?	Justification of Previous Column	Plans for Future Development
User is able to retrain the network on their own with their own specified number of epochs	Partially	The user is able to retrain the network, but only by going into the back-end files and modifying them. If they do not understand the code, this feature will not be available to them.	Add a screen/section in the GUI which allows the user to easily be able to do this.
It is easy for the user to make any adjustments to the network's shape and size	Partially	It is possible, but not easy. The user will have to go into the network files and modify them there.	Add a screen/section in the GUI which allows the user to easily be able to do this.
User is easily able to update the database	No	The user can update the database, but only by going into the back-end files. They also do not know the format of the database, making it even more difficult for them to do so.	Add a screen/section in the GUI which allows the user to easily be able to do this.

Post-Development Testing

Back-End

Test	Success Criteria	Met?	Evidence	Explanation
Accuracy of NN	Better than statistical model consistently	Yes	Page 42	This demonstrates that my NN is a more accurate tool for determining half-lives than existing statistical models

Front-End

Test	Success Criteria	Met?	Evidence	Explanation
Incorrect User Input (Robustness)	Able to catch incorrect input and prompt the user for correct input	Yes	Page 39	This demonstrates that my NN is a more accurate tool for determining half-lives than existing statistical models

Limitations

One of the main limiting features of my program is that it does not let the user interact with the network in any way shape or form. They are unable to modify or train it easily, as there is no option in the interface that allows them to do this. This is something I hope to amend in future development by adding menus and sections in the GUI that allow the user to interact with these things.

One of the limitations of my project as a whole is to do with the problem itself, as I chose a project which focused solely on alpha decay. This limits the capabilities of the program as it can only be used by a very specific group of people who only need to focus on one type of decay. In future, I hope to be able to develop a program which will predict the half-life for any conceivable isotope with any decay mode, with it also telling the user what type of decay it will undergo. For the current project, however, focusing on alpha decay was sufficient in terms of computational complexity, and I believe my solution is effective.

Evaluation Summary

Overall, I believe that my solution was a success as it reached all the essential success criteria, as well as satisfying the stakeholder requirements. I am very happy with the development and the performance of the neural network and the fact that my solution performs better than existing models is also very reassuring to me. Despite this, there are still ways in which my project can be improved, however, I believe that my solution is more than sufficient for the problem specified.

A video of the working code can be found here: https://youtu.be/lB3MoOg_INM
The full code is also available on GitHub with the following link: https://github.com/MAmJ4/CS-Project

Final Code

Back-End

data.py

```
import csv as csv
import numpy as np

import sys
import os
data_dir = os.path.abspath(os.path.join("..","Network and Data"))
sys.path.insert(1, data_dir) # add this path to system path

class Data ():

        def __init__ (self):
                self.data = []
                with open("Database.csv") as database:
                        csvreader = csv.reader (database)
                        for row in csvreader:

        self.data.append([row[0],int(row[1]),int(row[2]),int(row[3]),
                                        float(row[4]),float(row[5]),float(row[6])])

                # Format: Element, Z, N, A, Q, T12, ELDM
                for isotope in self.data:

                        Zdist = min([abs(isotope[1]-2), abs(isotope[1]-8),
                                abs(isotope[1]-20), abs(isotope[1]-28),
                                abs(isotope[1]-50), abs(isotope[1]-82),
                                abs(isotope[1]-126)])
                        isotope.insert (2, Zdist)

                        Ndist = min([abs(isotope[3]-2), abs(isotope[3]-8),
                                abs(isotope[3]-20), abs(isotope[3]-28),
                                abs(isotope[3]-50), abs(isotope[3]-82),
                                abs(isotope[3]-84), abs(isotope[3]-126)])
                        isotope.insert (4, Ndist)

                # Format: Element, Z, ZDist, N, NDist, A, Q, T12, ELDM


        def getZ (self):
                Z = []
                for x in self.data:
                        Z.append(x[1])
                return Z

        def getZDist (self):
                ZDist = []
                for x in self.data:
                        ZDist.append(x[2])
                return ZDist

        def getN (self):
                N = []
                for x in self.data:
                        N.append (x[3])
                return N

        def getNDist (self):
                NDist = []
                for x in self.data:
                        NDist.append(x[4])
                return NDist
```

```
        def getA (self):
                A = []
                for x in self.data:
                        A.append (x[5])
                return A

        def getQ (self):
                Q = []
                for x in self.data:
                        Q.append (x[6])
                return Q

        def getHL (self):
                HL = []
                for x in self.data:
                        HL.append (x[7])
                return HL

        def getModel (self):
                Model = []
                for x in self.data:
                        Model.append (x[8])
                return Model

        def getIsotope (self):
                Isotope = []
                for x in self.data:
                        Isotope.append ([x[1], x[3], x[5], x[6], x[2], x[4]])
                return Isotope
```

network.py

```
import numpy as np
import matplotlib.pyplot as plt
import data
from data import Data
import joblib

# miscellaneous functions
sigmoid = lambda x : 1.0/(1.0+np.exp(-x))
sigmoid_prime = lambda z : sigmoid(z)*(1-sigmoid(z))

# database
d = Data()

# network class
class Network ():

        def __init__ (self, structure, learningrate = 0.01):
                d = Data ()
                self.size = len(structure) # set size equal to number of layers
                self.structure = structure # set structure (n of neurons) to match
input

                self.weights = [] # declare weight array
                self.biases = [] # declare bias array
                for x in range (0, (self.size - 1)):
                        # initialise weights and biases with random values
                        self.weights.append(np.asarray(np.random.uniform(-1,1,
                                (structure[x+1],structure[x]))))
                        self.biases.append(np.asarray(np.random.uniform(-1,1,
                                (structure[x+1],1))))

                self.preactive = []

                self.activations = [] # declare activations
                for x in range (0, self.size):
                        # initialise activations with zeros
                        self.activations.append(np.zeros((structure[x],1)))
```

```
        self.learningrate = learningrate

    def feedforward (self, data):

        # min-max normalise all values
        Z = (data[0]-min(d.getZ()))/(max(d.getZ())-min(d.getZ()))
        N = (data[1]-min(d.getN()))/(max(d.getN())-min(d.getN()))
        A = (data[2]-min(d.getA()))/(max(d.getA())-min(d.getA()))
        Q = (data[3]-min(d.getQ()))/(max(d.getQ())-min(d.getQ()))
        Zd = (data[4]-min(d.getZDist()))/(max(d.getZDist())-min(d.getZDist()))
        Nd = (data[5]-min(d.getNDist()))/(max(d.getNDist())-min(d.getNDist()))

        # use normalised inputs as initial activations
        self.activations [0] = np.array([[Z], [N], [A], [Q], [Zd], [Nd]])

        # iterate through hidden layers using weights and biases (except for
        # last)
        for x in range (0, self.size - 1):
                a = self.activations [x]
                mult = np.array([np.matmul(self.weights[x], a)])
                mult = np.reshape(mult, (self.structure[x+1],1))
                presig = mult + self.biases [x]
                self.preactive.append(presig)
                self.activations [x+1] = sigmoid(mult + self.biases[x])

        #manually apply last layer weights and biases to avoid sigmoid

        af = np.array([])
        mult = np.array([np.matmul(self.weights[self.size - 2],
                self.activations [self.size - 2])])
        mult = np.reshape (mult, (self.structure[self.size - 1],1))
        af = mult + self.biases[self.size - 2]

        self.activations[-1] = [af] # was self.activations.append([af])
        self.preactive.append ([af])

        return af

    def backpropagation (self, activation, target):

        deltaBiases = [] # define arrays for change in biases
        deltaWeights = [] # define arrays for change in weights

        learningrate = self.learningrate

        # error in last layer
        error_l = activation - target # getting error
        error_l = np.reshape (error_l, (self.structure[-1], 1))
        # reshaping error to be array
        deltaBiases.insert (0, error_l) # error = delta biases
        deltaWeights.insert (0,
                np.matmul(error_l, self.activations[self.size-2].transpose()))
                # weight delta is error propagated backwards

        for x in range (0, self.size - 2):
                # propagate error backwards
                prop = np.matmul (self.weights[-(x+1)].transpose(), error_l)

                # hadamard product with preactivations
                error_l = np.multiply (prop,
                        sigmoid_prime (self.preactive[-(x+3)]))

                # reshape because numpy is peculiar
                error_l = np.reshape (error_l, (self.structure[-(x+2)] , 1))
                # error = delta bias so add to start
                deltaBiases.insert (0, error_l)
```

```
                        # add delta weights
                        deltaWeights.insert (0, np.matmul(error_l,
                                self.activations[((self.size-3)-x)].transpose()))


                for x in range (0,self.size-1):
                        # adjust each weight by delta weight
                        self.weights [x] = self.weights [x] -
                                (learningrate * deltaWeights[x])

                        # adjust each bias by delta bias
                        self.biases[x] = self.biases [x] -
                                (learningrate * deltaBiases[x])

        def train (self, dataset, targets, epochs):
                for x in range (epochs): # for the specified amount of epochs
                        #print (f"Epoch {x+1}")
                        # for each value in the dataset
                        for y in range (0, len(dataset)):
                                # feed it forward:
                                activation = self.feedforward (dataset[y])
                                # backpropagate against the target:
                                self.backpropagation (activation, targets[y])

        def evaluate (self, dataset, targets):
                predictions = [] # define array for predictions
                errors = [] # define array for errors
                for isotope in dataset:
                        prediction = self.feedforward (isotope)
                        # add predictions to array of predictions:
                        predictions.append(prediction)

                for x in range (len(predictions)):
                        error = predictions[x] - targets[x] # calculate errors
                        errors.append (error**2) # append error^2 to array

                # sigma = (1/n)*(sum of (errors)^2)^(1/2)
                stddev =  (np.sum(errors)**(1/2)) / len(dataset)
                return float(stddev)

        def save (self):
                joblib.dump(self.weights, "weights")
                joblib.dump(self.biases, "biases")
```

evaluate.py

```
import numpy as np
from network import Network
import data
from data import Data

d=Data()

def getSets ():
        global stddevModel

        isotopes = d.getIsotope()
        halflives = d.getHL()

        # number of training isotopes = len(dataset) * 0.8 rounded down
        numTrain = len(isotopes)*8//10
        numTest = len(isotopes)-numTrain # number of testing is what remains

        # all the possible indices I can use for isotopes
        totalNums = np.arange(len(isotopes))

        randomNums = np.random.choice (len(isotopes), numTrain, replace = False)
        # replace: whether or not a sample is returned to the sample pool
```

```
        # define testing indices to be whatever is left after getting rid of training
        testingNums = np.delete(totalNums, randomNums)

        trainSet = []
        trainLabels = []

        for x in range (numTrain):
                trainSet.append (isotopes[randomNums[x]])
                trainLabels.append (np.log10(halflives[randomNums[x]]))

        testSet = []
        testLabels = []

        for x in range (numTest):
                testSet.append (isotopes[testingNums[x]])
                testLabels.append (np.log10(halflives[testingNums[x]]))

        model = d.getModel()
        errors = []

        for x in range (numTest):
                actual = np.log10(halflives[testingNums[x]])
                models = np.log10(model[testingNums[x]])
                error = models - actual
                errors.append(error**2)

        stddevModel =  (np.sum(errors)**(1/2)) / (numTest)
        #print (f"Statistical Model Error: {stddevModel}")

        return trainSet, trainLabels, testSet, testLabels, stddevModel

trainSet, trainLabels, testSet, testLabels, stddevModel = getSets ()

net = Network ([6,16,20,20,16,1])

print ("Training...")
net.train(trainSet, trainLabels, epochs = 500) # train network
print ("Training Complete")

print ("Round 1")
stddevNetwork = net.evaluate(testSet, testLabels) # evaluate network on given test set
print (f"Network Deviation: {stddevNetwork}") # print net score
print (f"Stat Model Deviation: {stddevModel}") # print model score

if stddevNetwork < stddevModel: # if net is better than model
        print("Round 2") # round 2
        # fresh sets
        trainSet, trainLabels, testSet, testLabels, stddevModel = getSets ()

        # evaluate using already trained weights and biases
        stddevNetwork = net.evaluate(testSet, testLabels)

        print (f"Network Deviation: {stddevNetwork}") # print net score
        print (f"Stat Model Deviation: {stddevModel}") # print model score

        if stddevNetwork < stddevModel:
                print("Round 3")
                # repeat above
                trainSet, trainLabels, testSet, testLabels, stddevModel = getSets ()

                stddevNetwork = net.evaluate(testSet, testLabels)
                print (f"Network Deviation: {stddevNetwork}")
                print (f"Stat Model Deviation: {stddevModel}")

                if stddevNetwork < stddevModel: # if net IS better than model 3 times
                        print ("Saving")
                        net.save() # save weights and biases
```

```
''' CODE FOR COMPARING ERRORS
errors = []

epochs = 2000

print (f"Error Before Backpropagation: {net.feedforward (isotopes[i]) - np.log10
(d.getHL()[i])}")

for x in range (0, epochs):
        activation = net.feedforward (isotopes[i]) - np.log10 (d.getHL()[i])
        net.backpropagation (activation, np.log10 (d.getHL()[i]))
        error = activation - np.log10 (d.getHL()[i])
        errors.append(float(error))
        if x == (epochs-1):
                print (f"Final Error in BackProp: {error}")
                print (f"Log 10 of Predicted Half Life = {activation}")
                print (f"Log 10 of Actual Half Life = {np.log10 (d.getHL()[i])}")

print (f"Error After Backpropagation: {float(net.feedforward (isotopes[i])) -
float(np.log10 (d.getHL()[i]))}")
'''
```

Front-End

gui.py

```
from pathlib import Path
designPath = Path ("GUI Images/")
import predictor as ff
from tkinter import *
import numpy as np

class errorWindow(Toplevel):

    def __init__(self, master = None, message = "Error"):

        super().__init__(master = master)
        self.title("Error")
        self.geometry("300x50")
        label = Label(self, text = message)
        label.pack()

class ansWindow (Toplevel):

    def __init__ (self, master = None, answer = "Error"):
        super().__init__(master = master)
        self.title("Prediction")
        self.geometry("300x50")
        label = Label(self, text =
            f"Half-Life: {round(float(np.float_power(10, answer)), 5)}s \n
               Base 10 Logarithm: {round(float(answer), 5)}")
        label.pack()

def btn_clicked():
    data = []

    try:
        Z = int (entry0.get())
    except ValueError:
        # print ("Please enter an integer number of Protons")
        entry0.delete (0,END)
        errorWindow (window, "Please enter an integer number of Protons")
        return

    try:
        N = int (entry1.get())
    except ValueError:
        # print ("Please enter an integer number of Neutrons")
```

```
        entry1.delete(0,END)
        errorWindow (window, "Please enter an integer number of Neutrons")
        return

    try:
        Q = float (entry2.get())
    except ValueError:
        # print ("Please enter a number for Energy Release")
        entry2.delete(0,END)
        errorWindow (window, "Please enter a number for Energy Release")
        return

    A = Z+N
    Zdist = min ([abs(Z-2), abs(Z-8), abs(Z-20), abs(Z-28),
        abs(Z-50), abs(Z-82), abs(Z-126)])
    Ndist = min ([abs(N-2), abs(N-8), abs(N-20), abs(N-28),
        abs(N-50), abs(N-82), abs(N-84), abs(N-126)])
    data.append(Z)
    data.append(N)
    data.append(A)
    data.append(Q)
    data.append(Zdist)
    data.append(Ndist)

    loghalflife = ff.feedforward (data)
    # halflife = np.float_power (10, loghalflife)

    ansWindow (window, loghalflife)

window = Tk()
window.title("α-Predictor")
window.geometry("502x526")
window.configure(bg = "#ffffff")
canvas = Canvas(
    window,
    bg = "#ffffff",
    height = 526,
    width = 502,
    bd = 0,
    highlightthickness = 0,
    relief = "ridge")
canvas.place(x = 0, y = 0)

background_img = PhotoImage(file = designPath / "background.png")
background = canvas.create_image(
    251.0, 268.0,
    image=background_img)

entry0_img = PhotoImage(file = designPath / "img_textBox0.png")
entry0_bg = canvas.create_image(
    247.0, 153.5,
    image = entry0_img)

entry0 = Entry(
    bd = 0,
    bg = "#c4c4c4",
    highlightthickness = 0)

entry0.place(
    x = 94, y = 124,
    width = 306,
    height = 57)

entry1_img = PhotoImage(file = designPath / "img_textBox1.png")
entry1_bg = canvas.create_image(
    247.0, 263.5,
    image = entry1_img)
```

```
entry1 = Entry(
    bd = 0,
    bg = "#c4c4c4",
    highlightthickness = 0)

entry1.place(
    x = 94, y = 234,
    width = 306,
    height = 57)

entry2_img = PhotoImage(file = designPath / "img_textBox2.png")
entry2_bg = canvas.create_image(
    247.0, 375.5,
    image = entry2_img)

entry2 = Entry(
    bd = 0,
    bg = "#c4c4c4",
    highlightthickness = 0)

entry2.place(
    x = 94, y = 346,
    width = 306,
    height = 57)

img0 = PhotoImage(file = designPath / "img0.png")

b0 = Button(
    image = img0,
    borderwidth = 0,
    highlightthickness = 0,
    command = btn_clicked,
    relief = "flat")

b0.place(
    x = 199, y = 426,
    width = 106,
    height = 34)

window.resizable(False, False)
window.mainloop()
```

predictor.py

```
import numpy as np
import joblib

import os
import sys

data_dir = os.path.abspath (os.path.join("..","Network and Data"))
# find directory to current file
# go to parent directory ("..")
# from parent, add "Network and Data" to the path
sys.path.insert(1, data_dir) # add this path to system path

from data import Data
d = Data()

sigmoid = lambda x : 1.0/(1.0+np.exp(-x))

weights = joblib.load("weights")
biases = joblib.load ("biases")
activations = []

structure = [6,16,20,20,16,1]
size = len(structure)
for x in range (size):
```

```
    activations.append(np.zeros((structure[x],1)))

def feedforward (data):
    # min-max normalise all values
    Z = (data[0]-min(d.getZ()))/(max(d.getZ())-min(d.getZ()))
    N = (data[1]-min(d.getN()))/(max(d.getN())-min(d.getN()))
    A = (data[2]-min(d.getA()))/(max(d.getA())-min(d.getA()))
    Q = (data[3]-min(d.getQ()))/(max(d.getQ())-min(d.getQ()))
    Zd = (data[4]-min(d.getZDist()))/(max(d.getZDist())-min(d.getZDist()))
    Nd = (data[5]-min(d.getNDist()))/(max(d.getNDist())-min(d.getNDist()))

    activations [0] = np.array([[Z], [N], [A], [Q], [Zd], [Nd]])


    # iterate through hidden layers using weights and biases (except for last)
    for x in range (0, size - 1):
        a = activations [x]
        mult = np.array([np.matmul(weights[x], a)])
        mult = np.reshape(mult, (structure[x+1],1))
        presig = mult + biases [x]
        activations [x+1] = sigmoid(mult + biases[x])

    #manually apply last layer weights and biases to avoid sigmoid
    af = np.array([])
    mult = np.array([np.matmul(weights[size - 2], activations [size - 2])])
    mult = np.reshape (mult, (structure[size - 1],1))
    af = mult + biases[size - 2]
    activations.append ([af])
    return af
```

Joblib Dumps

Weights

```
8004 95b0 0000 0000 0000 005d 9428 8c13
6a6f 626c 6962 2e6e 756d 7079 5f70 6963
6b6c 6594 8c11 4e75 6d70 7941 7272 6179
5772 6170 7065 7294 9394 2981 947d 9428
8c08 7375 6263 6c61 7373 948c 056e 756d
7079 948c 076e 6461 7272 6179 9493 948c
0573 6861 7065 944b 104b 0686 948c 056f
7264 6572 948c 0143 948c 0564 7479 7065
9468 0768 0e93 948c 0266 3894 8988 8794
5294 284b 038c 013c 944e 4e4e 4aff ffff
ff4a ffff ffff 4b00 7494 628c 0a61 6c6c
6f77 5f6d 6d61 7094 8875 62e3 977a e6ed
86d1 bf37 2713 b885 30d0 bf45 4967 d524
81f2 bf6d d6a1 2b06 70ee 3fa3 d6fb d33f
9cbe 3fd4 362a d7ec 6ef2 bfb6 ee9b 39bc
9bee bf7f 66ac 51c7 9be4 3f88 3632 5faa
d2de bffc 47bb dc0e 44e3 bfb2 1828 5cf2
00dc bf22 b531 4f17 dee5 bf19 fb71 cb14
0cfa 3f48 50d4 1daa 52d2 3f1e 85ba e970
```

61e0 3fef d49d 1d6b 1c02 c03a d157 1cfd

02ab bf85 12fc 0288 c2f5 3fe0 ea59 89c5

24f0 bfbe e0b8 f9ee 8bd3 bf04 31b1 4e34

7ae4 bf21 9ed3 884b af01 40e8 8e88 a8bc

bbd1 bf82 952d 0328 01b1 bf14 a58f a1d4

77e7 bf51 eb45 1e49 40e6 bfee 1240 ba24

0ccd 3f94 3e0e 5c94 e4ec 3f58 bbf7 91e8

cfe7 bf56 d680 1ae3 0fd5 3f06 5ac7 aa25

f8e3 3f3b 6204 e25f 27e7 3f9b b270 80a5

95e5 3fd2 155e f9ec b4e5 3f71 7432 fca6

27f4 3fb3 7394 8b05 d3f4 bfa2 648c 1321

83fb bfd0 82d8 6db9 b1fb bfed 8548 5a71

44ec bf01 5842 5aaf ec15 405e 0992 92cd

afe6 bf23 59a9 f42c f2e8 3f7c a584 1321

2dc7 bf03 1642 089b ced5 3f3e 1876 3eb7

10c4 3f04 c485 277b d9fb bf45 eba9 813e

c8c9 3f0c 2b45 16ff efe5 3fba 4f66 724f

55e6 3fb9 9fc2 7611 4de7 3f64 b77d 5b1e

55ca 3fb2 f60f c9aa edde bfa3 aa8b 58d8

56d6 bf35 4d61 61b9 64db bffb 680e f7eb

edfd bf06 2a29 4542 1bc7 bf79 d63e b944

3cf4 bf2e 620c 46bc 1309 402b 3fd5 e2fc

18c8 bfda 1f09 65aa d8f5 bfc2 ad0d f03b

fbe0 bf3c 2e99 54e7 5ed8 bff9 847a 740f

aef0 bf58 db18 1cf2 c4e5 bf30 d4f9 2f1b

70f4 bfae 2540 65ad 0ed8 3f61 f69e 271e

6de9 3feb 98e4 af19 77e9 bfb0 7caf 2582

ccef 3ff7 e371 400d 4a00 c082 5bad 6d11

b0e9 3f7f 13a6 4e7a ade9 bfea ad26 abce

c1e7 bf2e b583 3a6c 01c1 bfbd b974 a80d

7bde 3fdf 014b f517 acfb bf66 c574 2912

bbce bf0c f80e 11a5 14ec bf8c 4f6c 83f7

33f6 bfb4 1a53 1e0a b0c9 bf19 ef04 98cd

8af7 bfbc 8507 a551 c4fc 3f62 0dc2 0797

119f 3fbb b300 fff9 e3f1 bff6 9705 be69

63bf bfe4 0573 16e7 d9e5 3f37 c3c3 e0b5

3de2 bfc3 baa9 e405 710d c0f2 3365 aeaf

7ae4 bf80 3c0c 54f3 61de 3f70 1b19 c726

f7e4 bfa9 131e 2f92 47e7 bf04 5bff 4815

d1e6 bf8b 7c95 935b 8dfc 3fad 576a 19c2

37c9 3faa 2e78 319a 31de 3f95 2100 0000

0000 0000 6803 2981 947d 9428 6806 6809

680a 4b14 4b10 8694 680c 680d 680e 6812

6815 8875 62a9 b721 bf9b b8a9 3fe8 1743

d4b0 2cbc 3f5b fea2 cdc1 81f0 bf77 9141

7d9c dad7 bf03 f2a1 1c13 00e5 3f30 dc36

12ef 29e0 bff9 d522 29b6 d1e0 3f16 861e

5dbc 5aea 3fc6 64ff efb6 87e1 3fea 8941

b1b6 61c5 3faa 8d60 2cd3 6ef2 bf5f 0040

26aa 75ed bf17 1002 ff92 03df 3fe4 cb99

1cc1 d285 3f3f 5fea f0e9 8bdc bfc0 460e

c2f9 afb5 bfa1 3dfe 4c98 51c9 3f4b 6297

8e1f 5ae5 3f2f 5e19 0da5 e0bf bf09 9abb

f400 b9dc 3f33 f531 38ab a2df 3f05 b1e3

739e 1ae0 bf67 6976 9f83 37df 3f84 b9e6

9293 51be bf9c 6417 cf5d af98 3f32 d9bd

d264 dbf1 3f9c ed41 8b8d a2ea 3fdd 3319

3ad9 45f0 bfdb 4b16 5bcd ce74 bf7d 8782

a768 72c5 bf7d 34dd 7fba bde5 3f31 1192

7bf7 aee9 bff5 b90b 3708 c7d4 bfc9 6d0f

684b 53dc bfdb 542b 1511 b3a1 3fb2 4d69

4a66 e2e9 3f19 b3cc a4e5 82e1 3f33 4369

0603 f7c3 3fe8 13f8 eee0 bbee bfd7 5334

8e16 a0d8 bf90 5546 6629 a0ad bfd8 5692

1ab5 cdf0 bf08 b411 828c 22ef bfe4 3f89

ca68 1acd 3f66 ca1e 30ca f7dc 3fe5 e9e6

8295 c7ef bf8b 398d 15d1 09e4 3f0c e6b3

4379 5bde 3f8f 1e43 6c41 88e3 3fa7 7b41

724a 0be9 3f15 22e9 0649 8fd8 bf4c 0943

41f3 c9bb bfbd 2c50 a793 9ea2 3feb 914c

fa66 0bd8 3fbf 7825 ece5 77b9 bfce 6e26

0513 87c7 bfd4 3bfb 0531 d2e8 3fab 93d0

a7d2 d4ea 3f4a d97d ae2e 0eef bfc3 4eee

3169 47e1 3ff7 b4af 90df ccb8 bfaf 9c66

c9a4 07b7 3fcb 4ffe b6b4 83ea 3fe7 7a02

1243 2fd4 3fee 4343 6695 dde3 bfb5 cbee

d521 9bbb bf83 7c01 dd87 41e0 3f13 3230

e6cb bf77 3fba c922 8f99 c1dc 3fc8 b32a

6466 49ea bf40 d4b2 817d e2f3 bf87 bc69

e343 4de7 3f66 5110 326c 7bd8 bf28 0204

419f 9fa6 bf0d 7435 0a1e 4bd6 3f8d 9529

f678 0ea2 bfaf 4f35 e6a2 ad6a bfad 0aa1

9115 cdc3 3f50 66ab fa59 82e9 3fef 57e1

67c2 f6ef bf14 6c3c 5de5 b6e6 3fa2 f099

5fd1 5ce4 bfac 95dd a3a2 f3bb bfcb dbb8

4118 b1f3 3fce eef3 38d3 27e2 3f03 4a63

3a00 81e9 bf86 b1a9 cc53 50fa 3ffa 1574

2a2d 64d2 3fd9 4d62 1503 98b3 3fb1 0d4b

8af3 53e0 3fdb 8942 abf5 3bea 3fe5 09b5

4d61 aee0 bf21 a5d2 1a29 21e4 3f89 23c5

6fe1 a1ee 3f8b 7d6e c317 0cf0 bfe8 6004

097b 15d7 3fb6 8166 ef8b 52e7 3f55 42e0

afc8 8fe5 3f1b 1873 5559 3ac2 bff4 01d9

d605 d8e6 3ff0 4d78 794e bee7 bf8f 8ea8

d2b0 2ae3 bf0c 6a6d 42ed 46db bffc e066

d52e 6bbe bffb ff62 95d0 fde0 bf89 f6ca

c798 e2d6 bfdd 164d e3c8 8de5 3f40 ed3d

e6f3 fae7 bf99 03d7 214e 14c0 bfa9 c1b6

eb88 dec8 3fad d7dc 6f76 f1e9 bfcc 8c84

0a2e f8e7 bfd9 7afc a61f aab8 3fab 5c75

c106 deeb 3f67 e63c 62ae b4e6 3ffe e877

dfbc bff0 bfad 7a45 9611 b9ef bfcc cc00

9a9c 8a82 3f36 6242 8360 ffd4 3f9b 7c11

a730 bdda 3f41 ce31 3cf2 ade9 bf04 a518

71da d2c4 bfbd 7366 ebdf 5ae9 3f6a 9418

855c 81c2 3f86 ef8d 072f 87f0 bf58 f579

7419 27dc 3fc1 4479 2660 4cde bf91 40e8

1163 0dce 3f4e 7a81 c866 dac2 bf4d 56e6

5149 79cf 3f18 6e7c 42ab 75f1 bff8 a216

71d0 0dec 3f43 4015 f056 bde1 3f57 5f9c

cba7 3897 bfac 6264 9a7c b2d7 3f39 0029

d99a 7af7 bf7e d702 daf8 75f5 bf58 f4c4

9bd5 bdf1 3fe6 bcbc b76b 44c5 3fe1 cdfd

9cd0 3cec bf23 8766 6572 b9e3 3fab 69f8

a989 aaf3 3fec c90f b6b1 9cf8 bff6 a852

115f 6dd5 bfc2 9009 383c 98c9 3f31 2f51

6770 17df 3fd4 e1ac e643 1fe5 3f81 1c78

1198 b9d2 bfd5 5bf4 30af a9e5 bfd1 658e

426a 08d0 3f49 e7e3 9eff 8afe bfc7 2e99

1b20 1ee3 3fb1 4bbd c97d 9be0 bfd6 f9ae

dd25 29d6 bf1e 20bb 1820 23f0 3fa7 4eea

be60 b2e9 3f9c c851 3adb 41f7 3f2a 2262

087b 9ae3 bf2d 856b 217b 57c4 3f5a d30f

2819 8bd1 bf88 903e 0e91 8cec 3fef cc71

97b2 deda bfe3 f27e 6e61 affc bf48 54ee

ddad dfdf 3f41 aa8c 2c5f 5fe8 3f22 38fe

3ebf efd8 3f6d c53f eb46 77fd 3f8d 14b2

ab8a 04e2 bf61 b05a 32e5 95f1 bf50 ac6e

cea2 95f6 3f54 9b04 4359 b5db bfb3 87c2

5a5c 1ced bf1e 78f3 b774 9ebd 3fb8 f643

ed2f adca 3f12 28ce 00cb 22fe bf4b 9e06

51d1 d5e4 3f04 84d0 274f 59e4 3f3a 00ca

253a 34b4 bfcc ff40 3a52 2de2 3fc2 ed97

112f c7e6 bf53 bf35 a432 b5cf bfcb 1dfd

96ae faee bfd4 b6bc a11a 52e6 bf2d 079e

0cd4 88c2 bf1c 94e2 a301 24eb bfc5 9ce1

53d6 a2b7 bfe4 f9e9 8a61 f2d7 bf2a 9512

dae3 92dd bfbd 8b18 432b 5ada 3fcb 5f20

9de9 27eb bfa1 8c53 b3f4 4b99 3fdc c6f7

89f9 3fe1 3f2a 0b6d d752 80e1 bfa1 0c83

d925 03e6 3f3e 4c2d 8fe6 c8dd 3f9d f474

5528 5ef1 bfd0 b509 dafa 3cdd 3f10 e5d0

3be3 e0f2 bf9e fc22 ba31 63e3 bfef 8bd2

96f5 02da 3f95 f299 d77f 19e0 3fea b195

a972 fef1 bfdf 3f88 a97e fabf 3f98 4c7c

b0ac 42d7 3f07 fa22 4ce8 4dea bff6 72f1

5b14 bae3 3f22 72e5 caf8 75e7 3f20 1896

7d8c cfe9 3f6e 2eac 4ead 34c9 bf90 dd9a

d1de d3e7 bfbf 1b95 e6d9 1ff1 bf6d 1c06

07a3 81ec 3fbd 2ef7 eee9 c8ba 3fbd 548b

422d 67de 3fb9 6361 7aca f9f6 3f0c bf1e

32c4 d0b7 3f14 7339 91ef 0fe2 bf47 484c

855d 7bf1 3fab 9010 bfff 47e2 3f2e e41c

aead 8cee bff0 d0da e50a 06f3 bf06 cec5

716a 41eb 3fa2 d3e7 2c37 02f2 bfcc b386

6753 8ff0 3f5d 1977 80cf dcb9 bfde aed0

b573 8ee9 bf63 3993 0d52 57d5 3f3e 544c

fb9b 24d7 bfbb 352a 22bb 01c3 3fde 85f6

8cfc 787f 3fde 1241 7f18 84fe bf37 f853

4e82 4ef1 3f38 9374 bf54 7aea bf97 d67b

5372 91f4 bf32 d7c3 f149 38e4 bf97 b72f

d214 77d6 3f94 9e15 902f 27f3 3f90 d4f0

cc7d 31e5 bfd5 e5e4 cbdb 8ef9 3f82 bb6f

472b 55f4 bf31 b27b 2002 52de 3f71 9dbb

992f 5fe1 bf70 2b2f 5131 fab2 bf85 6791

debd 6ae5 bff1 e1ae fe14 8ea4 bf26 50a4

c0eb e8d6 3fcd 1a23 6310 b8f4 bf34 3374

899f 42e6 3f15 5b1f 65b8 a5de bf25 b560

ab9a 3cd6 3f88 9494 561c dbe2 3f6a 5406

7a6c 3ae8 3f09 2f19 be46 f6ef 3f9f 3769

c650 d9ca bff1 0b62 2812 51e4 3f81 d48d

076b fbf4 bf00 c9ec 9efe dfd8 3fb5 0085

7c6a fcec bfab ee29 dc53 6ad8 3f4a f9e4

f8e6 d797 bf90 07a7 b63a bce5 bf03 d767

bdf9 9aaa 3f90 09e5 9972 d7f2 3f20 481f

99c3 9bd8 3f09 0c9b e7e1 32d8 bffc 94b0

a414 c4e8 3f3b f7c4 5581 73eb bf1c 9fcb

7ed0 8aef bf6a 9de8 3cd3 dddb bf4e 34d7

0734 65e9 3fd1 5a14 eb4a 84e7 bfbc eb2b

fd75 0be7 bfcc a47f 973e f4e2 bfc8 921a

3a9f eabb bfca 9c26 8e97 5ad0 bf26 4de9

2970 ecbc 3f1d ce32 d3b2 60c9 3fb2 5655

e251 43f5 bf4e 50f1 50ed 4cfa bfa1 4ce6

7b2e acef 3ff6 4157 c39d c2e8 3fd4 394f

6b31 a1f2 bf6d 5157 6c02 a7f3 3f20 c999

6ed8 5dde bf65 ab02 6f2b 17f5 3f8a 67a0

9117 65e1 bff4 3071 bcea 23fc 3f04 9918

a2d0 58e3 bf00 d836 0e9d 5fec bf11 42f5

b47a 3cda 3fc8 11f8 2f4c 3ef8 3f28 0a4d

9fcb 82df bf1c 2d7d 97c8 47b8 bffd bfbd

6517 37db 3fe6 1026 45d3 4af5 bf5b 81a5

b268 9bda 3f67 e057 d96a b3af bf17 e63b

da4f 5ef4 bf2a 3b47 2471 2ccd bf79 8842

8812 b3da 3ff9 3d93 d8c9 ead4 bf13 73da

6282 c6f5 bfb8 7805 eece 43ca bfad d028

6207 62c3 bf31 8ea2 0b03 6fc5 3f42 2707

d7ef b1dc 3f45 17fe 8f5c aedc bf9f 64ed

7c98 b9d8 3f83 dcbd 9d88 b3e7 3f9d 0d37

4f52 2ce1 3fa8 2b6b e0de 11db bf88 7298

e781 dadd bf26 72c2 415a 92df bfce ac39

9be6 56f1 3f0f 18b4 b683 0fec 3fd7 1d7b

1c2c 76e5 3faa edb1 0ea1 39db 3fa6 6892

7398 13e6 3f41 f516 2e11 bbbf 3f47 d1e6

4801 22e2 bf95 2100 0000 0000 0000 6803

2981 947d 9428 6806 6809 680a 4b14 4b14

8694 680c 680d 680e 6812 6815 8875 6261

ddfe 6ab7 32ea bff8 ca39 0c0b e3d8 bfd3

5c08 51e8 d9e5 bfc9 90aa f216 abe8 3fed

2a42 cda4 42c5 bfe1 743e 4d94 aeea bf47

ccfb bff6 9ecc bfed 8f9e 5952 f3e2 bfe2

3ee9 d767 2fe6 bf3b 8345 f8e1 bee0 bff0

75f3 204b 1cf2 3f8c 8ca3 6cd2 eaea 3f94

c529 692c 26e8 bfad afcd 4261 31d6 3f42

80f3 66d6 99ee bfd0 8c9f 0953 73ea 3f69

4aee 9c2d 20e8 bf16 2493 643d f6a3 bfe6

28a8 1fb4 aae9 bffa fbba cca3 36c3 3feb

b92b d118 e0ec bfc7 b8fa 50b1 fbd3 bfa1

c28e bc44 05d6 bf9c 4212 7d16 bee7 bfbf

5515 9118 4ce5 3f00 4606 0d52 ffca bf35

ca6b 6acf 06d5 3f97 eecf c703 31d0 3f48

7293 22c1 d5e3 bf8f e7ba 08c4 73c8 bf6a

5625 d308 31e8 bf9d d7fa ba47 f4d5 3f9e

f29d f6ba 4ded bf9b ffbb 1712 cfc0 3fb8

5704 0225 e3d6 bf06 a20d 502c a5d2 bf43

08a3 4ee9 38ae bf80 c9e5 21d2 aaca 3feb

11c6 f407 1acb 3f6b c0ad 786f cfe5 3f21

fc20 c6ab a8b6 bf5f fb8d e4f0 14c9 bf52

59e2 b2f2 c3e4 3fb5 db05 c59d a4d3 3f8c

a33b 9284 e6c5 3f6c c8db dba7 faed bf32

045c 15a6 e9d6 bf56 539a 27f5 0bed 3f47

9550 d27c 52e5 bf56 6e49 c48c 7dea 3f13

5402 d323 3df0 bf27 e835 c830 9dc7 3f1d

3ea2 fdd8 ad8e 3fe6 8399 67a7 30ea bf93

ab11 fd56 95e4 bf74 6b50 9873 6bb8 3fde

9f10 0e32 5beb bf23 2690 55d1 60e1 bf47

b2f0 f98b ebef 3f2f 4d41 0389 5ce3 bf64

19e0 b0e5 3ae4 bf7c 76d2 460e 84c3 bf18

f7d3 c496 e6b1 bf8c 901a 7082 0adf 3ff2

abad cb96 52c7 bfb3 f71a 5c7d 19e8 bf3f

97e9 0bf7 8fb1 3fb4 3748 858f 07e6 bf16

1524 d595 93e2 3fb8 4b2f 3807 34a3 bf87

ea24 0c99 1dc8 bf8a 2f4b ba12 f6a3 3f9f

6482 7cc6 0fd4 bf16 7af1 540b 71f3 bf26

ff9a e926 48e5 3f22 6ac3 60dc f7e2 3f80

6ed0 f3b6 17e3 3fc0 1a4a e345 44db bfa0

f050 d0c6 41d7 3f75 451b 8deb 18d1 bfd2

1f64 6326 fca4 3fbf aa89 fc4e 6ae5 3f92

51dd 79f1 62f3 3fa0 8acd 7222 95de bf3a

cdee c894 86d2 3ff5 d2bc 14a5 aae3 bf1a

1ae7 8ae1 d9e7 bf73 6c98 b0f8 61eb bf92

bb9c 6c59 11eb bf70 dd65 31ba d4da 3fdf

ab8a fcac 8ac2 bf42 208e f409 90c3 3fcb

e630 9597 dff5 3fad 63c2 e6c2 cac4 bff3

6120 b81b 09f1 3f42 281b 8d5a d6e7 3f0e

d79d 8216 24a7 bf08 48ba db50 4dd7 bfa9

9e70 c6f4 efe5 3f10 356e 5ee5 c2b8 3fb6

32ed 5b00 54ae bfed ce03 ff03 b2ef bf76

f1ba 9bb1 53ce 3fee e9c7 176a 55f1 bf7e

3ac1 2569 27f8 bf1b 9ad3 b15e 85d7 3faf

b89a 7719 89e0 bf34 3034 4db7 ace4 bf61

96e6 55aa fbf6 3f74 e008 273c 6cf3 bf98

f9e6 8fa1 0600 40f3 454b 2466 31db bfc9

780e 417e e0df bf7d 9b14 f2d3 15f3 3f6c

6923 66dc f9fa bfd1 7740 1dc4 a3b2 3f43

8039 6eff ddd9 bf24 0176 2952 5ff5 bf22

4dc3 8b24 1ed2 bf16 ad59 313a 34e1 3f8a

3530 9af0 aada bfd8 04c9 a3d1 4fc9 3f81

6a64 989f a5f0 bf51 d436 6c6d 46df bfc5

0b51 1c04 8aec bf9e 3bdb 19fd 53c2 3f48

6a1d 95c9 d3db bf1b 042a fa60 17dd 3ff7

69a0 b0b1 61f4 3f06 6de5 c6ad 19e2 3fb2

5ff2 11f8 26a8 3f68 d24b a218 64c5 3f5b

cdf1 39ac 1de1 bfa3 46f4 8b05 d3e2 3f49

06f6 6eaf a097 3f9a 7c0c 4f71 e9ef bf0d

9a82 3a6d 75e6 bf1c f2d4 4a83 f3f2 bf5a

93c9 48d6 b1f1 bfdb 8d58 76dc b7dc bf23

6cdf e14e 06db 3f08 7d36 b7c3 39b0 3f5b

0fcb 86ab 01e3 3f27 3208 028c 73d6 bf1b

98c3 5468 21d0 3f34 333f dc63 47d6 bf54

31ac 736e 83d9 bf4b 19a7 ca80 abe7 3f85

5702 1086 75df 3fe9 5de6 6fab 89ed bf06

3fc6 adbe d9f2 3fea c5a8 c5e9 43e4 3feb

7454 51d9 c5de bf58 3010 d3c9 a5e2 bfa5

ea99 de49 a7da bf67 8f8d f54a dbf2 bf97

f1e5 2ff4 34e8 3f96 359e 7127 77f7 bf1d

d4e5 8185 35f5 bf25 cb6e b7b2 f5d2 3f80

999d 5c5e 8ed1 3f8e 90f4 1639 58dc 3f05

caa1 e28e 96e2 bf62 4303 7fbe 2ff2 bf6a

feba 1a39 e8e8 3f3d 29c9 b4df dde7 3ffe

993a 3b5f 72cb 3f7d 805a 07ae cce3 bf61

30c5 03c3 a7b9 3f32 fcf3 6c51 4ee7 bf0c

8dfc 99f8 f2e7 3f9b 72fa 6d4a e3ea 3faa

d6ed 6596 039e bfa8 68b1 cb69 92e7 3fc2

e536 12da 09b8 bfd0 225d 7d12 94e8 3f36

e6f2 b0c6 a3e2 bfc6 d892 0fdb 92ac bf93

7453 2d75 78e8 3ff6 a5be e4d5 c2e0 bfbd

89a7 1643 29ec bfda c176 99f6 a8ec bfbf

dce7 5ba5 f1ef 3f4b 65c3 8c8d 69ca bf7a

cddb 7b59 8fe9 bf1d 3f4d f448 b8d2 bf50

5b1c 3a55 0de1 bf11 7e5a bcd7 36de bf61

bc98 d7f0 33dd 3f37 0ed7 cdb0 ace7 3f4d

efce e9d3 9eab 3fef 5447 4491 69e2 3f88

1ef3 1376 1deb bf41 c14e 70ba 1dd4 bf07

acba e18d 1aee 3f46 480d eb28 93c3 3f46

f0ca c425 39c6 bf3f c129 372a 3f9a bf77

ba01 eb1a 9cc2 3f49 50b1 055e 2fcc bf31

59f6 e1f7 76ed 3fb5 176a 9da5 f0d6 3fb6

c428 a618 88d0 bf17 8318 4cea ade0 3f73

be4f df9c 48f1 bf5d 20fa 8aa0 51dc bf27

2110 6d64 abc3 bfe4 347f 2b77 d2ec bf7d

28bc e5a4 fded 3fa8 d20f 7395 5198 bfaa

8523 50df 64f9 3f5a 5b54 d708 50b5 3f2a

1b8e efa2 0ae0 bf3e 70d3 98fd 79ba bfaf

2f05 9b23 71da bf3a bf0e 14e4 62e0 3f21

6fb3 c6c5 b5d5 bfe4 f7a8 d2f3 05f8 bf6d

8ccc 8491 99f3 bf03 6f35 7c59 e3dc bfff

da4a a082 98dd bfea c5bd f46d 10ee bf2d

4822 0a92 39e3 3f64 7fa0 1ded e5d3 bfca

58d4 63a9 03e8 bf9b 5e26 d8c3 29e1 bfad

6b09 1360 87e8 bfb6 a7ef 7b5d 1ee4 bfcb

2668 df8f e9dd 3f9e 14ba 47c8 76c8 bf42

9bd3 e12b 53b2 bf1d a445 fef7 08e5 bf88

ce6a 342f b3ee 3fac def2 0274 82db 3f5e

92dd a502 97d6 3fa7 59f4 6ffa cddf bf53

f992 9fb9 bacf 3f8d a1d0 7cb8 c8c6 3fbb

c425 95ba bfe8 bfc0 e428 68a4 d4e1 3fe6

b1ee 87ff 9ae4 3ff0 a84b 721b cadf bf18

7b0d 9f40 17ee 3f7d 919d 0cd5 34d9 3f4b

41e0 362c 8ccc 3f15 181a e0a5 cff5 bfa5

a09e ebee 59d7 bf60 ee1a ce3b 0de7 3f0f

949b 49e1 41ec bf05 3919 65b5 eae7 3f4f

3722 9a4b 29be 3f26 9cb5 bc3d 2fe6 3f7d

719d 0f79 1994 bfbd 27a4 2955 5aeb bf7a

dec4 d8a7 fae8 3f59 6bc8 fcbc 42be 3fd3

4f5f 10a2 77f1 bfb7 0ee5 65d1 84d8 3f36

94b5 e23f a1d1 3f6c fda1 430f 97e1 3fa8

2629 f81f 18e5 3fb1 76e4 632f 4af2 bf37

8d64 b328 b4e7 3f66 a66a a2f8 cbe6 3f02

c776 8a5d 3dcd 3f08 7e16 6e84 24ed bf6f

c551 6412 0eb6 3fdf 233f 5813 22e1 3f02

5aea 3368 82a0 3f73 dd62 97fe 5de4 bfda

c963 cc32 b4e1 bf76 42b0 79dc c1e1 bf4b

7051 66da 27ea bf98 b724 d453 f291 3f93

e8cb f262 638f bfeb eaec 3f20 21ec 3f67

e860 4ac1 08e7 bf3a 9a2f a820 e2de bfd1

5bf5 b83e c9ec 3f17 3f14 5c8f 99d6 3f52

b7cf bb3c 00ec 3f76 4676 d14b bce1 bfa5

7b2e 8f5b 38c6 bfce c469 1e52 f4d8 3fd8

3c57 2102 4fd3 bfbd 72e2 70eb 29ee bf41

0332 5569 6ee3 3f07 42f0 55dd bcaa 3fa9

0aae b044 12ef bf64 80d4 14a3 87e0 3f40

e53e 86dc 1ff7 bfe0 927b 51b0 3ed9 bfcd

a5b5 25b7 93c2 3f75 9c98 a659 1ff3 bf3f

6425 b8ef 95f3 3f95 91a1 309b 8de1 3f0b

2fe0 acef 41d2 bff1 d1b3 50a7 92ca bfdd

f361 2e3a 13ec 3f82 35b1 f388 9ce4 bfc5

094a 77e2 f2f1 bf1a ac6f 5f1a 3ee3 3f52

a468 d5fb 22cd bfa5 9f3f 9496 20b1 3f57

fc74 4a18 00ef 3f40 0cb2 a7c4 97f4 bf58

0b39 f469 bfec bf71 0904 91a4 3fd7 bf63

c967 f858 28c6 3f24 ca23 4305 51f0 3fd1

9281 d863 20f7 bfda 0cf3 f360 9ae2 3f2a

4a58 5c13 35e0 bf8f 1475 7757 aaf9 bf5e

e6a4 9d21 baf4 3fed ff82 d0a6 2ce4 3f4f

a26d 4f21 0df7 bf66 e52b 5100 06e5 3fda

a535 09a4 3fc6 bf24 c8ac 025a 8fed bf95

6133 e883 5eda bf0a 76c7 694d 1de1 3f44

5c1d 9334 32d6 bf75 d97d 2129 a7ad 3f08

7c7a 6ce5 77a6 bfe8 e16b 6867 3bc7 3f18

76a1 9f0a 48d3 3faf 8309 7d1c aec4 bfd6

e3f4 9a20 72f0 3feb 8149 06e4 50f1 bf91

529c dd98 bad3 bf41 4362 7592 aee0 bf6e

e78c 0e64 63e8 3f71 76e8 a1ab 08e6 bf92

15f0 fb7f 4ad0 bf37 20b5 ba4a 53ee bfcc

6f4d 0bba c4e4 3f42 0790 b48c 609c bf4d

51d9 5aaa 41f2 bf90 18ef fba9 4be7 3f07

df62 93b5 c2ef bf06 77a5 71d7 0ceb bf9d

842e 6f76 cde0 bfbc ecb5 ffb8 39e5 bfa7

effa 5835 f0d0 3f41 9d0e a2ca 0ef4 bfb0

d9cb 2898 eace 3fad 81ac 3e0b c9c6 3fa0

4f99 789f 78c3 3fee 2c45 0ebf d4f3 3f4f

fe66 6990 4cd7 bfe9 0bb3 a730 eeda 3f3f

fc89 0bbd 78d7 bf0f 027d 21d7 d4e4 bf7a

49d1 baf5 e0f4 3f51 f2ac 6fb4 b6b4 3f2e

7ac3 a127 07ec bf04 885e 068b 07f5 3fc0

f46f 3efd 86e3 bf96 be95 a073 c5b6 bfca

527e 83e7 66eb 3fc2 47b1 e3d4 afeb 3fc6

ea8a ce47 49d3 bf9b 07ed 9072 b7c9 bf5f

eb34 99f3 ebdb 3fb1 2fbe 41c6 2ad6 bfca

fd6d f5d5 a4e7 bfc4 fb10 7bb4 bfda 3f4a

b9de 35c7 df8d bff6 5ddc 8816 c5c9 bfb8

7b7b 7b6f bff5 3f4d c83e 85ab 40b9 3f16

2e96 7798 b9ec bf58 2ff8 c20f eef3 3f1e

79ed 4fc3 37e3 bfc3 9c83 0720 75e5 bf90

c387 d9ff 6fd1 3f0c c347 4d0f 60f7 bf34

26af e63f b8e7 bff2 da63 fc7e 01f0 3f2c

0789 683d 09a9 bf79 0e94 3f19 e2e1 bf93

bcd8 6b0d c8e4 3fcd 93b1 6d6b 4ee2 3f4f

bb1b be18 d4cc bfaa 2011 93dd 03e6 3faa

e438 99d8 e7d8 bf58 6560 c0b3 33ca bf46

b372 e159 e5e3 bf96 c0f1 8d78 4cd6 bfba

0d4e 7bd6 c8d7 3f1f ef24 b392 51d5 bfa1

80b9 72c0 5ef1 bffa 633b 1e71 05e9 3f0c

4307 d90f 0cdc 3f76 ea04 38c7 2ab6 bf87

ca02 9c11 cfe7 3fa6 e6ab d689 95aa 3f05

c598 75d4 e8df bf01 32e4 17b5 7bf1 bf95

2100 0000 0000 0000 6803 2981 947d 9428

6806 6809 680a 4b10 4b14 8694 680c 680d

680e 6812 6815 8875 62ed e47a 913e 60db

bf2d f433 ac77 01e3 bfdb c425 5a85 52ee

bf9d 90c4 615a 1ca8 bf72 fc41 1cd7 8be8

bfa5 3aeb cccd 14e1 3fb5 828f 3f07 94e0

3f13 faab bfd8 b4d1 3fb3 96d8 3ca6 d0ec

bfa5 6504 1bc7 a1db 3f06 1cd2 77b5 8cef

3f51 73ec bfb8 5ec3 bfde b994 d2e4 e8c6

3fda 1d1a 53d8 27e4 bfed f6cb 9945 acc7

3f57 7ae3 bc3f baf2 bfba 4c64 ede9 37ab

bf6f 715c f7c8 bcd3 bf58 84f3 87a5 c8ec

3f8c ab3b 70f1 5b9c 3ff3 5110 84c0 eedd

bf51 59d0 2bc9 4ec1 3f80 e12f fdfd 33e8

3fbe 5133 5464 c8da bf6e 45be b1eb 9bc5

3fcb 819e 91a8 49f4 bfe5 0295 1708 09c6

3f15 173b 55b5 fbd5 bfe6 f6a8 c7eb 25e3

bfd9 3e3f d96c a0d6 3f7a 1a8e a1f1 6de4

bf8a 938e 3098 4ec5 3f75 0086 fbb5 c1f0

3fc0 2bce c6a5 f6c0 3fb2 f5d6 407a 0df6

3f34 4872 c0c2 58d1 3fab 8bd7 a080 13ea

bfb4 4958 aaf9 5dd4 bf27 2d37 569e e2f5

bfbd 438d c899 cee4 3f38 9129 0a0a a3e5

3f2b 9998 ecf4 3cd6 3f7f fc40 1439 1bf0

bf29 80e0 2a26 82ed bf48 4f09 f8ef f0f7

bfbd bece 9bc7 3ef6 3f8a fd94 52d5 4cee

3fd3 2420 2d5e 8feb 3f2b 011f 1267 cdb3

3fc8 3585 5708 bed9 bf26 d17f e460 e7ef

3fed 679f 6e5c 8599 3f4b d603 0406 e9ef

bf43 44db 0814 cbe9 bf68 8cb1 eac8 6df4

bf7f 0c99 b5eb 84e6 bf36 78b1 1bcd a1c0

bf3a 8b3d 051a 9ff2 bf94 37a7 8396 7ef2

bfbf 19f9 3044 eac4 bfe4 3290 0c46 9ddd

3fc5 f0f6 420b 62e0 bf60 9479 1876 2be4

bf4b 83d8 6b6f 2bc2 3f61 d2af 9bc6 26e5

bfbf b792 ac16 94e2 bff2 d6aa a2be 35b1

bfad 25e0 6b8c 37e4 3fc1 ef4f 4746 68ef

3ff7 fa6f 8707 5ac9 3f39 d235 e7b8 eddd

bf4a b648 283e 72e5 bf23 1793 13e8 dce7

3fba 635c f89b e2e9 3f41 66e2 6091 fce4

3f6e 57a1 38a4 46c1 3f33 4f8e ac52 e8e0

3f64 a38a b72e 9ce8 bf7d 44d5 261b 14ea

bf8e 5faa 3132 e5ea 3fbf 8b10 e757 e4d5

bfd2 4202 6606 49df 3fad 4597 abb6 d1e2

3f52 aaba 833a 24e2 bfe9 ed98 c588 d4e8

bfc5 4685 d710 00f2 bf11 add2 e7e7 cbe4

bf7a 3037 0c93 b4f0 bfd9 c98d 5e95 3cd9

3f2d 797c 6c88 6fea 3f46 d4eb 9194 8bdd

bfc8 bda3 2e5d 2fe3 bf6e 34e5 e701 7fe9

3f94 c5a3 97c1 54f3 bf34 daab cf58 f0e6

3f84 958d d3b5 72ec 3f1c eb7b 76da 78a2

bfc2 d060 b54f c7e8 3fd2 8e32 c5bc e1ee

bf55 e3f1 9ee3 b5eb bf1a 721a 537d 45ec

bfcb 1196 7661 cdcb 3f5c 4d2f 31ff e5dd

3f01 e1e5 eea0 25f0 bfae 446d b018 a0e1

bf4f 47c8 ae07 62f6 bfcc 2d37 100b 89f1

bf33 3a5b c66a cfe4 bf26 e71c 0b49 bfdc

3fd1 3983 beb1 90a2 bf34 b075 b04d b7bd

bf4c ad5a a7d6 4cc9 bfe9 16a9 a608 cadd

bf0c cc3d f29e 9bd2 3fcb 5c39 a184 2be5

3f9e 0376 1e07 fef8 3f89 6cc6 b9f8 46ee

bf7d da94 d0d0 e3f2 3f3a 176e 0d7d dff3

bf0a 40c6 dc97 2eda bfd5 587b 3dda b8ce

3f4f 26a1 f7f3 3bea 3f26 351f feba d4ea

bfc9 40c5 9215 90c5 bfbb c6e0 1e0d 29eb

bf62 2d5e 9401 46a0 bfb3 0a81 a59d bcc8

3fb8 63b4 e5f4 dcba bf83 9dec 466a 78c3

bf96 c6c7 27cf ace6 bf5c f8c7 1af7 0ac5

3f83 7377 1f02 c7e2 bf77 fac2 5b0e b1e0

3f2c f37d d4ef 66eb bfea 9e3f ba68 29e4

bf44 d732 0712 b8e2 bf85 936e 077e 13e7

3f91 437b 97b4 89c4 3f1d d94f 452d f5c9

bf1e 6050 93c4 4ce0 bfea 6cd5 5c4d 84e6

bfcb 86cc 3669 9cdb 3f61 39ba 7ca0 5ce2

3ff4 cc5b ae3f 7fa0 3f86 9158 2fb5 44f2

bf71 f8b1 80b3 aeb3 3f0c b2d2 2ee4 c1ef

3f4c 0bcf 5415 a5d9 3f9f 3604 0683 8bec

bfd4 9fec 8082 99c6 3fc0 f073 b82b 25f0

3f38 e6e6 ef52 11e9 bf38 d27b 1e68 85f1

bfe8 01ea bb1b cde0 3fdc f704 b030 00c5

3fbc 64d7 ac0d fddb bf27 4178 4b42 e7e9

bfac c3e1 624f 31d1 bf63 333b 5651 2dd1

3fc6 64af 417e c6e4 3f73 ae00 c03c e0e4

bf70 f94c 66ae f3de bf5d 4bc6 d405 14b7

bf7c f602 0029 7298 3f43 62f1 fd5c 4ec3

3fa0 b934 8aa4 25e3 3f40 87a2 3560 6bb9

3f65 354d 4380 62e7 3fb9 87b4 d010 37eb

bfb6 3ac9 34d9 c0eb 3f1b ac53 a501 11e8

3ffa f119 6ed0 e8a3 3fa8 9417 acd3 46e5

bf8c 8f80 34c1 80e5 bf6c 0268 7e9b 1dcf

3f4c 39c3 c304 89ab 3ffe ee7d 2ddd c2e2

bf55 4d9b 4e3f d8f2 bf0f 3069 8c74 35e0

3fc9 16a5 182d 76e8 3ff2 c55e 14de addc

3f2a 88d8 d6a0 fed1 bf88 83cc 1dc6 88b4

3fd6 2263 eeca a8d1 3fd7 14bc 828a bde0

bf2b c939 b6f9 c9ef 3f09 ad66 0b96 a7e7

bf22 9fab b8be e2c1 3fd3 247d 583b 71f1

bfc2 def0 0e55 78d4 bfc0 e950 48fb 9de4

bf67 9796 f982 21ed 3ff6 e29f 6452 61f2

bfec f117 8a9f 82e8 3ff3 9f67 6319 33eb

3fa3 b72c 6a14 f2ab bf96 d2e6 4d9a b9dc

3f29 0d1c e7e6 2ebd 3f4a 9c39 8aeb 53e9

bf58 1dee 751d ebd2 bfaf 5f1f d46e b6da

3fb5 117b 0c2d efc0 bfe8 418f e583 5ebc

3fb7 072f fa3c e787 bf19 57cc fc0f 90f2

bff3 f7d9 0ac3 a6dc bf4f 5bfc 6b5e 8ddd

bf11 b76a 5be1 dad7 3f66 db75 5ada 48e6

3f71 c703 1c25 c5d2 bf42 e05b 3ad1 29e8

3f35 560f 4d75 26da 3fe7 bc62 c0a3 d9f2

bf96 ad84 7904 65f0 bfcc ddda 6b84 29c5

3fbb 9ef1 7201 6be0 3f1b eb9e 0645 03f0

bf32 bfd4 2eca 5fce bf0d afda da53 70e5

3f9b 976e d253 4aed bf5c b344 508c 6dd3

3f6a 51b4 c512 49db 3fe0 0be0 f59f 6ec7

3ffb 8894 b5d6 93e8 3f02 4ae0 edf7 18d2

bf04 5a23 4b41 7df6 bf1b 2474 0e0f 15d7

bf55 8b3e 48fc 89bc 3f3a d0f7 8e1d 82e1

bf28 5fd9 5943 a1a8 bf84 6504 3147 62ec

bf0e 74f3 2d22 5ade bf7d b46a 8a19 a7cf

3fc9 f565 df63 32d5 bfc5 a591 33a1 34d7

3ff1 275c f7ba 44ea 3f80 6ad6 7cef d3ee

bfc9 0d3f ae75 03de 3ff9 f66d 6f68 00f8

bf71 c497 2772 f6e7 bf81 32a0 d384 7fe5

3fbe a397 4ab8 d7d3 bf15 52c7 689c 98e3

bf21 f997 1b78 eedd 3f04 b493 7a96 12ee

bfec 9a05 f4a2 73f3 3f02 da3f a2e0 9be1

bf9a 2795 68d8 b0e0 bfea 9136 2db5 6ee5

3fe1 fa88 a369 80c2 3f0b 00bc 50fe 15ef

3f4c d332 9f9f 2cb2 3fc7 c307 eda6 e5cc

bfc5 3348 d957 24d9 3ff3 893b 4ce5 7779

bf77 3d1a e6df c8f3 bf55 b246 91d9 80a1

3f4e 74da b1ac 8cd6 3f50 2d9d a4d0 5bd2

bfc2 844a d40e e9e4 bf0f d684 172e 88c4

bf52 add6 e342 dec3 bf13 b7f5 0eed ebd5

3faa f7b5 b0fd 9beb bfbc 4623 279e 22d4

bfb8 0bc2 38b1 10f4 3f99 7f04 a183 83ed

3f70 c414 03c6 aee3 3fba fbec c96d f0e8

3f1a 139f 2c5d 92e6 3fbd 0f6f 3450 46e7

3f35 039e dc23 decb bf6b 8ffd f939 96e8

bfa7 2f07 5b8d eda2 bf84 f241 9408 5af4

bf31 b953 fb83 40ef bfcb aaf0 364f 92ea

bf49 b1ec 229e bbd8 bf2a 3b6b d80b 1e97

bf4f aa88 a1a1 5ce3 3f5b df07 4ee5 7fe9

3fad 4771 000f dbe9 bfa0 c2d7 df5f 2ed2

3ff0 59e9 140b a8ed bf82 1355 3ba7 cde2

bff5 792f 66af 87e6 3f94 7758 00be bfec

bf93 4800 ddf7 99bb bf63 1c72 9e5a e8ed

bf5c 4fdb dc91 54ef bf7e f8c4 99d5 5dde

3fa6 5ab7 889d b9d5 bfd3 9ebd 09f8 93e2

```
3f1b f108 e98a eeef bfab 5922 8d28 deb8
3f1e fe63 737a e9e5 3f7d 354c 7e16 54dd
3fcc a0b5 3ee1 eae7 bfbe c632 afd4 33d7
bf9a 15be 2ea0 f1ee bf4c 462d 7603 46e2
3fe9 39f9 eba6 93ee bf3d 2547 a413 4dc3
3f69 476e 9eb9 35eb 3f1c be69 4226 42ce
bffb 5df1 a619 3bdc bffb fc5b e17d baee
bfaa 6cf7 c76e f4e7 bf8a 1a59 193c 64e8
bf63 784c 95f4 49ee 3f62 a028 5a5a 52c4
3f26 5f08 f77e 18d9 3f70 3712 57db 90ca
bf69 8209 97c9 3cb3 3f2b 8649 32f5 7dd5
bfb4 c05d c8a6 e4f2 3f25 f292 96ce 93ea
bf97 997b f7f4 ced8 3f56 f6df b15a 9add
bf4d 3b2f ef3d a3ee bf95 2100 0000 0000
0000 6803 2981 947d 9428 6806 6809 680a
4b01 4b10 8694 680c 680d 680e 6812 6815
8875 62bd 0624 802e 0bf3 bf36 a137 7fef
e4fa 3fd4 063e c1b4 0106 c00b fde8 b24c
6eca bf6c 1270 29c2 c9fb 3f07 585e a40d
a201 40d4 833e 4a3c 2ee0 bfcb c05a c7da
49e8 bf3e cb0d 2e31 12eb bf5f 3423 f162
5dc2 bfe1 f00c 5261 76e4 bf30 e18c 109f
bafb 3f1f f358 6c1b d7ee bf63 28b5 38bd
f300 c067 833b 747b c1dc bfdd c78a 4fc7
e3f3 3f65 2e
```


Biases


```
8004 95b0 0000 0000 0000 005d 9428 8c13
6a6f 626c 6962 2e6e 756d 7079 5f70 6963
6b6c 6594 8c11 4e75 6d70 7941 7272 6179
5772 6170 7065 7294 9394 2981 947d 9428
8c08 7375 6263 6c61 7373 948c 056e 756d
7079 948c 076e 6461 7272 6179 9493 948c
0573 6861 7065 944b 104b 0186 948c 056f
7264 6572 948c 0143 948c 0564 7479 7065
9468 0768 0e93 948c 0266 3894 8988 8794
5294 284b 038c 013c 944e 4e4e 4aff ffff
ff4a ffff ffff 4b00 7494 628c 0a61 6c6c
```

6f77 5f6d 6d61 7094 8875 62cc b77e ab38

6ec2 bfda 6824 0e11 94ed bf0d 0ec7 b2e0

ecb2 bf56 fed9 4707 41de bff4 f296 98ce

e3dc bf19 5629 57a2 1eba 3f00 44ad 8300

2fd7 bf94 8552 8374 78c9 3f90 f673 d2f9

d3b2 3f43 d138 4330 18d7 3f81 1864 f892

ddb8 bfcc f676 0885 95d3 3f4b c1f3 30be

b9d8 bf04 f679 2200 97e4 3f48 3e97 a1a8

32cd 3f61 f103 119a 82a1 bf95 2100 0000

0000 0000 6803 2981 947d 9428 6806 6809

680a 4b14 4b01 8694 680c 680d 680e 6812

6815 8875 6226 98a6 d8ad a3c8 bf14 ddf1

35c4 59df 3fb3 4080 e35c 25e2 bfda fba9

4832 add9 3ff7 ada4 f81f 4dd9 3f20 508b

2d5e f6ed bf58 48e4 07f4 f2c8 bfb5 cd45

37be 4fe6 bf4b 1073 58ca cad2 3feb 3113

66ab 0ee7 bfbd 0627 6cf0 99f2 bf02 c783

38df 9edb bf4e c4b9 0fd7 8bd8 3f2c 8011

a947 b7b8 bfb6 2004 98f8 8d92 bf8a 3d2e

406c 19f1 bf9a fe46 a743 43e9 3fbc d9b4

4523 68d0 3f13 66c7 2e78 b2de 3fc4 d0c5

bf9a c7ae bf95 2100 0000 0000 0000 6803

2981 947d 9428 6806 6809 680a 4b14 4b01

8694 680c 680d 680e 6812 6815 8875 624b

cec0 a3d4 ace6 3f01 37a5 d7c9 e0d2 bf00

0905 2c67 3be0 bfc9 5dba 3499 08e1 3f06

0902 fb25 ade2 3fc8 408f 5c4b b9e1 bf08

3b94 05d0 07ce 3f6f a510 1326 a8e2 bf6b

f754 9b2b 0ae4 bf57 f301 2111 52e9 bf08

e632 a81d c4e7 bfe5 d98b 8bc1 8beb bfc3

e60b 3fe5 9ae1 bfe1 1f0a 6e23 c8be bfca

3fbd 8bd5 86e0 bf7c 0153 d937 d6ac bf76

3c62 b2ab a6e2 3f90 1e37 5e90 a4e5 3f59

08bf 13a3 a9e2 3f6c 228e 2cb6 5be1 bf95

2100 0000 0000 0000 6803 2981 947d 9428

6806 6809 680a 4b10 4b01 8694 680c 680d

680e 6812 6815 8875 62f9 2c19 5745 90d1

```
bf91 6f3c 1a76 5ad6 3f66 d593 4ebe 29d0
3f38 7b5f a4cf 71ef bf7e 65a1 c06f 2dcc
3f06 0a9f 5f4c bfd3 3fdc 79bb 913a 79e9
bfe7 04f5 0f42 08d8 3fcb ac93 f714 81ce
bf43 e615 6267 a5da bfee aeb0 4c39 a7dc
3f90 3cb4 1fd7 20d7 3f43 8843 d763 29e7
bf00 5cfa eb3b a1d6 bff3 aad0 d70c 44d7
3fd0 8071 e08b aace bf95 2100 0000 0000
0000 6803 2981 947d 9428 6806 6809 680a
4b01 4b01 8694 680c 680d 680e 6812 6815
    8875 62bf 4fdf a9f4 b6e2 bf65 2e
```