Using Evolutionary Algorithms to Design Neural Network Architecture and Hyperparameter Selection

Final Report for CS39440 Major Project

Author: Marc-Osric Ruben Nikias Cooper (mnc3@aber.ac.uk)

Supervisor: Dr. Chuan Lu (cul@aber.ac.uk)

3rd May 2018

Version 1.0 (Release)

This report is submitted as partial fulfilment of a BEng Degree in Software Engineering (inc Integrated Industrial and Professional Training) (G600)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of Originality

In signing below, confirm that:

- The submission is my own work, expect where clearly indicated
- I understand that there are severe penalties for Unacceptable Academic Practise, which can lead to loss of marks or even the withholding of a degree
- I have read the regulations of Unacceptable Academic Practise from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science
- In submitting this work I understand and agree to abide by the University's regulations governing these issues

Name: Marc Cooper

Date: 03/05/18

Consent to share this work

By including my name below, I hereby agree to this dissertation made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Marc Cooper

Date: 03/05/18

Acknowledgements

I would like to thank my sixth form physics teacher Darrel Hamilton for his unrivalled ability to inspire and challenge me like no other teacher has managed before or since. Without his guidance it is safe to say I would never have found myself still in education writing these words of thanks, reminiscing over the time spent sparring over the subject he taught.

Abstract

Neural Networks have become an increasingly popular machine learning tool for creating complex, flexible and robust data models. One of the main drawbacks of this technology is the need for expert knowledge to be able to build effective network architectures for particular data problems and systems, thus resulting in slow and expensive development. This paper aims to present an automatic solution of using an evolutionary algorithm to optimise both the parameters and architectures of neural networks in an effort to improve dataset classification.

Contents

1. Introduction

1.1 Introduction

Since the early days of computing, scientists have been using biology as inspiration to help solve complex computational problems such as Cellular Automata [1] and Emergence [2] to name just a few. One such ideas which, due to the computational limitations of the time, was abandoned in the 1940's but found its renaissance in the early 1980's and is now in wide spread use today is the Neural Network [3]. This bio-inspired supervised learning model works by connecting many simple computational nodes (neurons) together in such a way that each node can influence others with varying degrees dependant on the strength (weight) of the connection between them. This means the computation power comes from the network structure and not the individual nodes, allowing for flexible yet powerful model creation.

Even though neural networks, even when using noisy data [4], have been found to perform very well in a wide variety of tasks, they can only perform well if their parameters are set correctly to suit the problem. Ill-fitting parameters cause either underfitting or overfitting [5] of the model to the data resulting in bad performance, meaning that construction of a neural network is inherently also a hyperparameter optimisation problem.

Taking the above into consideration alongside the fact that there is an ever-increasing adoption of neural networks in more and more tasks, the need for well-constructed neural networks is ever increasing. Current methods often employ experts who, through use of experience, judge and consequently build what they see as suitable architecture. This way of constructing neural networks is expensive and slow as the expert needs time to fully understand the data and system they are working with before they can start building the network.

For clarification; unless expressly given a specific name of a different type, this paper will be using the term "Neural Network" to refer to a "Multi-layered Perceptron" from now on.

1.2 Aim

The aim of the project is to develop a distributed system that automatically explores, improved and selects both the architecture and hyperparameters of neural networks through use of an evolutionary algorithm to improve the classification accuracy for a given problem.

The system should be used through a command line interface which allows the user to set experimental parameters as well as system settings such as the Server IP address when setting up the clients.

2. Research & Analysis

This chapter will explore the relevant components of the proposed system and present evaluations of the different technologies available.

2.1 Background

The main inspiration comes from a paper submitted to the 2017 GECCO conference [6] titled *"A Genetic Programming Approach to Designing Convolutional Neural Network Architectures"* [7]. In this paper, the authors describe how they use a genetic algorithm to design the architecture of a convolutional neural network by having it select the type of layers that should be used and how they are to be connected within the architecture. The results show this method to be incredibly competitive when compared to other state of the art classifiers which mostly require high levels of manual tuning and apart from a two-week computational time, shows that this method to be an interesting and viable solution.

Taking the general concept of the paper and adapting its scope to fit the computational power available to the average user, it was decided to use a genetic algorithm to design a simpler neural network; a multi-layered perceptron.

2.2 Neural Networks

Neural Networks are bio-inspired artificial structures which, unlike von Neumann machines [8] that processes instructions sequentially, use massively parallel networks built up of simple computational nodes called neurons to explore many hypotheses in the solution space simultaneously. This parallel computation is achieved with changeable weighted connections between the individual neurons which act as signal modifiers. It is these connection weights which determine how the input data is transformed into an output and therefore are the elements of the network which are modified to ensure better performance of the model. Before this process of modification of the weights is explained it is important to understand the other fundamental parts that make up a neural network.

2.2.1 Neurons

$$\text{Neuron Input: } Y = \sum(weight * input) + bias$$

Neurons can be considered the computational component of a neural network as they perform the calculations which produce the network output. The calculations they perform work by taking in the aggregate of the inputs into that neuron and performing a fixed mathematical operation called an activation function. The specific activation function operation depends on what the network needs to do but some of the most common operations are outlined below.

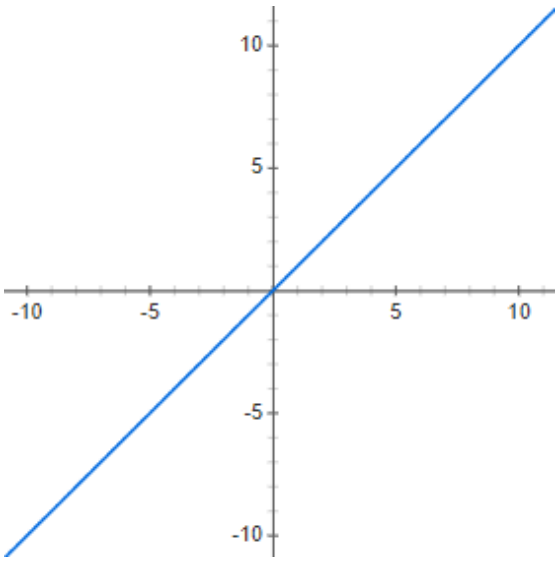
2.2.2 Activation Function

The purpose of activation functions is to introduce non-linearity into the neural network by mapping the sum of the inputs to an output, therefore enabling the formation of complex functional mappings from data.

Linear

A linear function is the simplest form of activation function as it outputs in a directly proportional fashion to its input. If this were to be used as an activation function, it would turn the neural network into a *Linear Regression Model* [9] and would mean its learning capabilities of backpropagation (2.2.4) would be lost as no variable gradient is available to ascertain the direction and velocity of a connections needed weight change.

Another issue that arises when using a linear activation function is adding multiple layers would result in the neural network behaving as if it were a single layer perceptron; as summing these layers together would produce just another linear function, thus negating the computational power of the neural network.



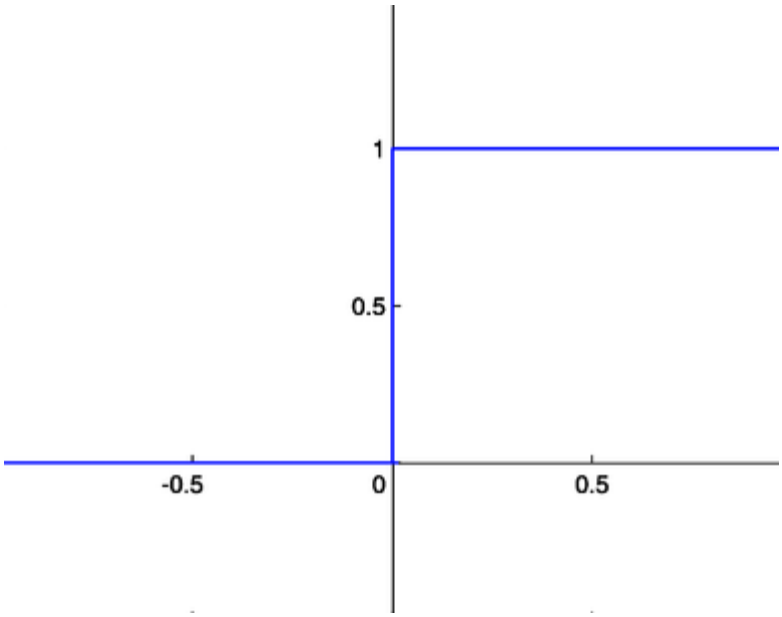
$$f = cx$$

Figure 1. Graphical representation of a linear activation function

Step Function

The step function can be thought of as a threshold function, where the neurons is activated if the combined inputs reach a certain limit. Even though this looks like it could potentially satisfy the needs of an activation function for a binary classifier [10], it struggles to give a useable output with larger and deeper networks. This is due to sum of the multiple inputs neuron being likely to exceed the threshold value required and therefore resulting in the output giving no meaningful information in regards to the input.

Even though this is a specialised activation function not suitable for true neural networks, they have specialised applications in use with binarized neural networks [11] which are networks designed for fast and light training.



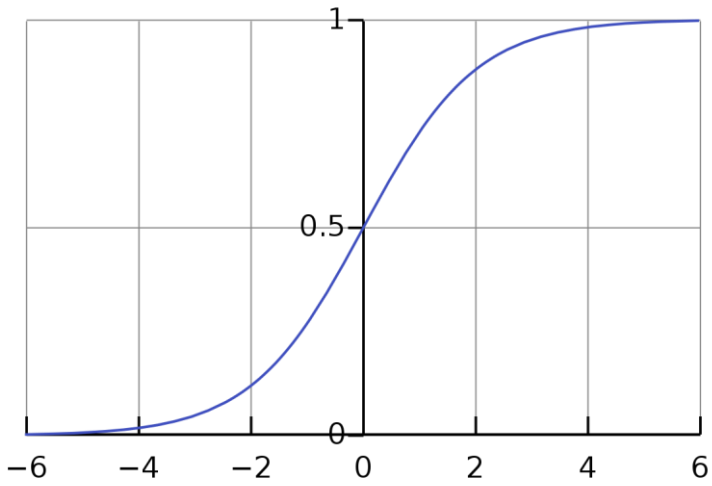
$$f = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geq 0 \end{cases}$$

Figure 2. Graphical representation of a step activation function

Sigmoid

The sigmoid function is one of the most commonly implemented activation functions as it allows a meaningful mapping of an unlimited input range into a discrete output, therefore allowing for a theoretically unlimited number of inputs to be fed into a single neuron and still deliver a usable output.

Another advantage of the sigmoid function is that due to the steep *y*-values around 0.5; the output of the neuron will tend towards either 1 or 0 during training of the network as any small change in the x-axis will have a large impact on the y-axis.



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

Figure 3. Graphical representation of a sigmoid activation function

2.2.3 Bias

Bias is an essential but often overlooked part of the training process as it shifts the activation function line or curve to along the y-axis, producing a more varied mapping of input to outputs.

This can be shown when comparing the two graphs of sigmoid activation functions that employ bias and ones that don't. Figure 4 shows a very simple single connection model between two neurons and Figure 5 the associated sigmoid graph with example weights.

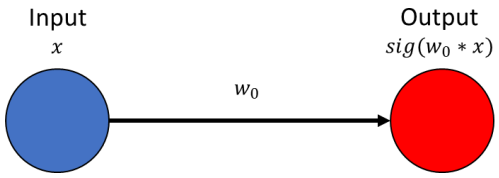


Figure 4. Single connection model without bias

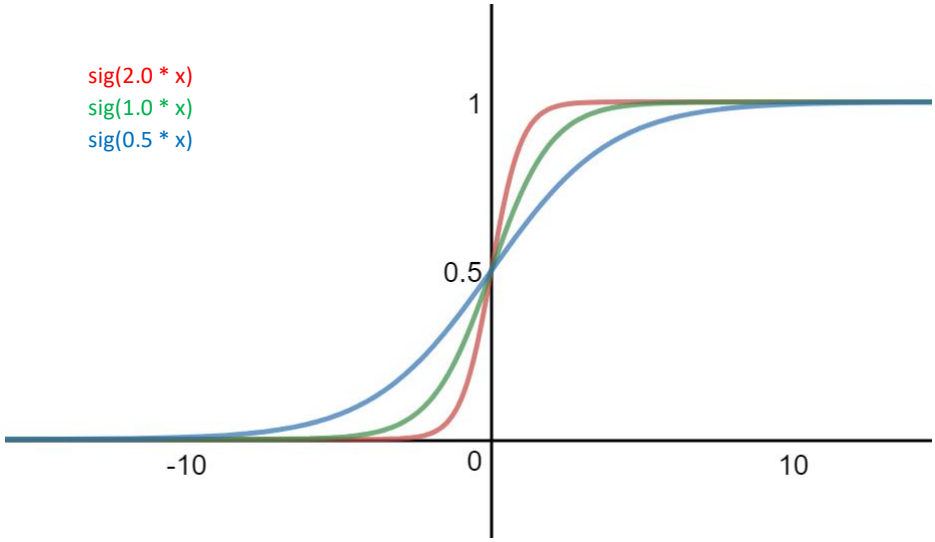


Figure 5. Graph for output where no bias is added with different connection weight examples

As can be seen in Figure 5, without bias the weight of the incoming connection merely changes the steepness of the curve but as the optimal model may not pass through *(0, 0.5)* it is important to be able to shift the activation function to find the optimal y intercept.

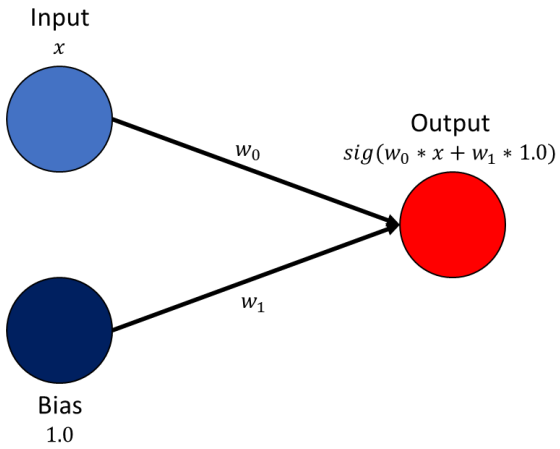


Figure 6. Single connection model with bias

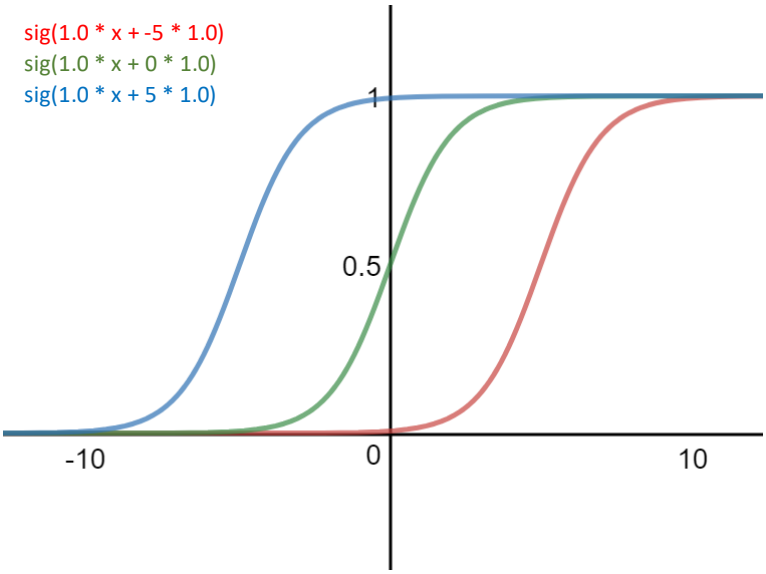


Figure 7. Graph for sigmoid activation functions with bias

As can be seen in Figure 7, adding the bias and its weight into the equation enables the translation of the activation function along the x-axis, which in turn changes the Y intercept. Thus, the model is able to explore and find an optimal Y intercept for the layer which results in a more effective mapping of the complex functions found in the relationships in the input data.

2.2.4 Layers

Neural networks consist of three different types of neurons: input, output and hidden neurons; these three types of neurons also reflect the kinds of layers present in a neural network. Layers comprise of multiple neurons that are all connected to the same neurons in the previous and next layer.

The input and output layers are unique as they are defined by the dataset itself, with the input layer mirroring the size of the input vector and the output layer being the same size as the output vector. These two layers are instrumental to the operation as they are able to introduce the data to the neural network and transform the abstracted format of the neural network into readable output. It is this abstraction of the dataset where the computational power of a neural network is found where a final type of layer is needed; the hidden layer.

The hidden layers, as mentioned, are the abstraction layers of the network and they achieve this by introducing higher levels of non-linearity when forming a data model. Meaning each layer can be seen as a filter that is able to recognise and form itself to complex generalised patterns within the data and are then able to output that they "*recognised"* that pattern. This crucial abstraction of input data to pattern recognition is the function the hidden layers provide and is what gives the neural network its powerful classification ability.

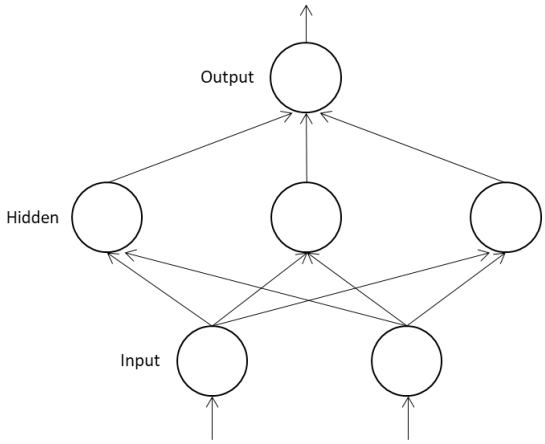


Figure 8. Example of a simple neural network structure

2.2.4 Backpropagation:

Backpropagation is the most common and effective training method for feedforward neural networks and was first introduced the in the 1970's but was brought into prominence in the 1980's by D. Rummelhart et al [12]. It works to reduce the total cost function, Figure X, by adjusting the weights of the connections to the outputs in proportion to the error rate of said outputs prediction.

$$Cost\ Function = \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Figure 9. Equation for single prediction cost function where x represents the actual and $\bar{x}$ the predicted result

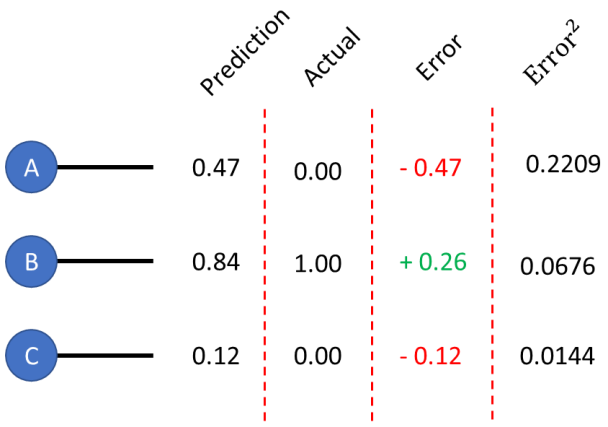


Figure 10. How the square error of each prediction is calculated of each output in a neural network

This is essence means that for a particular training input, if the output is that of Figure 10 then all connections that influence "*output A"* should decrease more than an equivalent weight would for connections influencing "*output C"* due to the larger error, while all connections to "*output B"* would be increased in weight to strengthen the link between a training input and its desired output.

Therefore, as more training examples are passed through the network architecture; the weights will be adjusted so that the network will be better at predicting the correct output and therefore reduce the cost function as the network becomes better trained. This reduction in overall cost function is achieved through a processed called *"gradient descent training"* [13] which reduces the cost function of individual weights by calculating the direction and velocity of the change needed to achieve the local minima cost of that weight.

This can be demonstrated by using a single connection optimisation as an example, as seen in Figure 11, where the *X-axis* represents the weight of the connection and the *Y-axis* the cost function for the example. By calculating the gradient (green line) from the difference between the actual and predicted output, the direction and magnitude of the weight change can be ascertained.

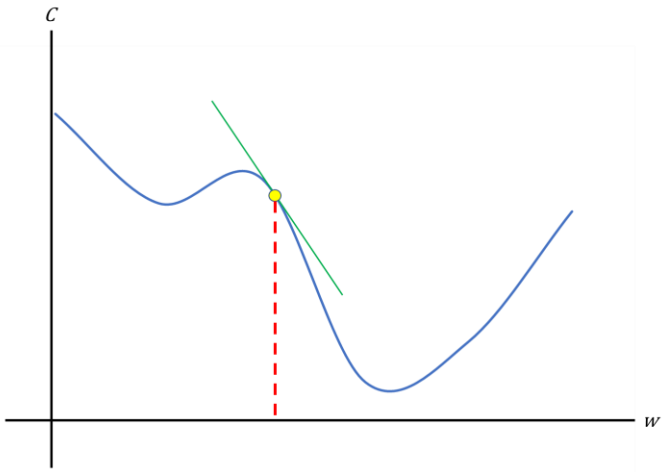


Figure 11. Single connection local minima search using gradient descent as indicator to change weight of connection

2.2.5 Learning Rate

If the gradient descent could influence the connection weight directly and without modification, the resulting change to each of weight would be so large that the local minima would be overshot by most of the iterations. This is where a learning rate is introduced; it reduces the gradient descents influence over the change made to the individual connections weights to ensure that the local minima for that particular weight is realised as closely as possible.

2.2.6 Batch Size

Batch size is a technique that combines multiple training examples into a single batch, which is then fed through the network without running backpropagation on each of the training examples. It is only once all the training examples have been fed through the network does the sum of the changes of the entire batch get used for the gradient descent. This technique reduces the amount of computational power needed as only a fraction of the gradient descent calculations are performed, therefore significantly speeding up training but consequently reducing the accuracy of the connection weight adjustments.

2.3 Genetic Algorithms

Genetic algorithms are another bio-inspired algorithm designed to imitate the processes of natural selection in order to explore the solution spaces of a problems. Even though Genetic Algorithms come in a variety of different flavours, they all still all adhere to five basic components outlined in [14]:

1) Generic representation of a solution to the problem
2) A way to create an initial population of solutions
3) An evaluation function rating system; in terms of their fitness
4) Genetic operators that alter the genetic composition of children during the reproduction
5) Values for the parameters of genetic algorithms

To follow the steps outlined above, the first step would be to transform the solution space into a generic representation called a chromosome, which allows for genetic operations to be performed on it. If looking to transform the number and size of the hidden layers in a neural network into a chromosome, we might choose to do so by storing each layer as a separate number in an array, thus simultaneously storing the number of layers and how many neurons in each as seen in Figure 12.

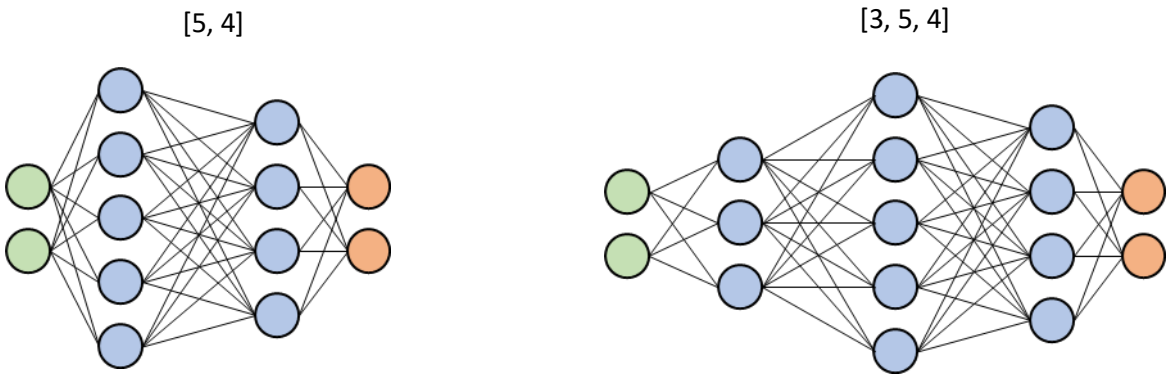


Figure 12. Example of how simple chromosome can represent a solution

After a generic representation of the solution space is achieved, an initial population of random candidate solutions (referred to as organisms) are created and evaluated. This evaluation is an assessment of how well an organism performs as a solution for the problem and is called its *fitness function*. Continuing the example of using a genetic algorithm to design a neural network, a suitable fitness function could be the classification accuracy achieved of the neural network.

Once each of the potential organisms have been evaluated, the next step is to perform genetic operations on their chromosomes in order to create the next generation of candidate solutions. This process aims to improve upon the previous generations success by crossing over the chromosomes of two parent solutions and mutating the resultant offspring.

There are several methods which can be used to select parent organisms, the simplest of which is by selecting the most successful organisms and using them as the parents. This will ensure that the resultant offspring inherits properties that made its parents successful, but can lead to the population getting stuck in local maxima caused by premature stagnation [15] due to low genetic diversity [16].

This is why it is desirable to implement a selection method that encourages genetic diversity as this increases the number of different strategies being explored, therefore increasing the chances of a better solution being found. One of the methods that allows for this is the *roulette wheel selection method,* which works by allocating each organism a "chance" to be selected proportionally to its fitness.

$$Chance\ to\ be\ select = \frac{Fitness\ of\ Organism}{Total\ Fitness\ of\ population}$$

This method is called "roulette wheel" selection as it can be imagined as a roulette wheel where each organism occupies a section of the whole dependant on its fitness, so when the wheel is spun the chances of the sector landing on an organism is related to the area it occupies.

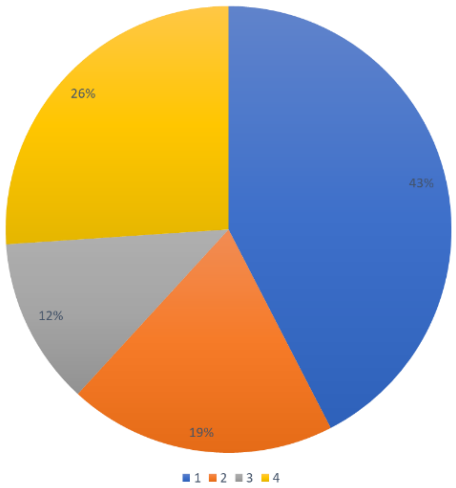


Figure 13. Visual example of how roulette wheel selection

Once the parents have been selected their chromosomes are crossed over, which works by combining the chromosomes of two parent solutions and producing an offspring that contains the genes of both parents. The types of cross-over implemented is dependent on the chromosome, but when considering a one-dimensional chromosome, such as the one expressed in the example above, either single-point or multi-point crossover could be implemented.

Single-crossover works by selecting a random point along the chromosome and selecting everything up to the point from parent A and everything after from parent B as shown in Figure 14.

[10, 45, | 50, 60, 70]
[40, 25, | 30, 55, 60]

[10, 45, 30, 55, 60]

Figure 14. Single-point crossover with the parents, crossover point line and child

Conversely, multi-point crossover employs multiple points along the chromosomes and alternatively inserts genes from the two parents along each crossover point as demonstrated in Figure 15

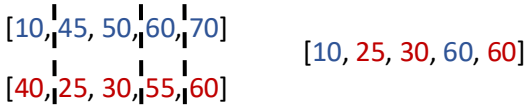


Figure 15. Multi-point crossover with the parents, crossover point lines and child

After crossover has been performed the next step is to apply the mutation operators on the resulting offspring solution. The possible mutation operators must be relevant to the search space and so taking the neural network example above further, the possible mutation operators are: changing the number of neurons in a layer, deleting a layer and removing a layer.

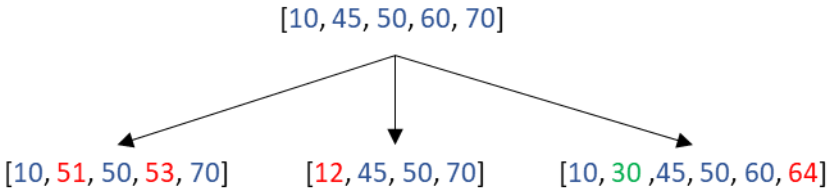


Figure 16. Example of possible mutations for a chromosome that

One of the things to note is that when exploring search spaces whose potential solutions have a high computational cost, such as neural network architectures, it is often beneficial to force mutation as this means that previously explored solutions aren't evaluated again, saving computational resources.

The final stage is for the offspring's fitness's to be evaluated and compared against the general population, if the offspring performs or matches a criteria better than another individual in general population then it replaces the less suited organism.

2.4 Technologies

2.4.1 Programming Languages

One of the challenges of this project is choosing the right programming language as the system has to incorporate many different technologies and functions while still being easy to either run on a platform or be easily compiled for those platforms.

R

R [17] is a programming language designed for statistical analysis and graphical data displaying. It is a popular language to use for things such as machine learning due to its focus on statistical data analysis and processing. One of the many advantages R has is its inbuild data pre-processing ability aids in developing machine learning tools for tasks such as data extraction and sanitation.

One of the main drawbacks of R is the lack of support for "general purpose programming", such as; networking protocols and operating system functions which can cause issues when trying to write something that has to perform both statistical analysis and general tasks.

Python

Python [18] is considered a very good all round and powerful programming language mainly due to its extensive standard library support which can handle complicated tasks such as string manipulation, networking and machine learning. This means that it is ideal for writing complex programs with relative ease and speed as these libraries can be called upon instead of having to write the protocols and functions from the ground up.

Java

One of the main attractions of Java [19] as a language, especially for this project, is the fact that it is platform independent thanks to the Java Virtual Machine (JVM), meaning it can easily be ported to any device without fear of cross-platform incompatibility. Though by using a JVM, the language is inherently slower and more memory-consuming than natively compiled languages such as C and C++.

2.4.2 Neural Network Libraries

The core of the system will be the processing and evaluation of the potentially very different neural network architectures. Therefore, it is important that the library implemented allows for flexible network generation while offering good performance during run-time to reduce the inevitable high computational cost of the system.

Tensorflow

Tensorflow [20] is one of the most popular deep learning libraries in use today as it offers a wide variety of options and flexibility when creating neural network architectures. Alongside this, Tensorflow has been found to be one of the most competitive libraries when considering computational speed. These two things in combination make Tensorflow a very attractive machine learning library to use.

One of the main disadvantages is the apparent steep learning curve that is required when first using the library as it requires a more solid understanding of machine learning than many other libraries to be used effectively.

Supported Languages: C++ and Python

Keras

Keras [21] is a high level neural network API layer that is designed to run on top of Tensorflow, CNTK or Theano and has been designed to make building neural network architectures simple and painless. This simplification does mean that much of the customisation and flexibility available is sacrificed and often means that the framework has to be used in non-intended ways to create custom architectures that fall outside of normal use cases.

Supported Languages: Python

H20

H20 [22] is considered a user-friendly library as it designed to offer a broad foundation for machine learning techniques, while still offering a powerful distribution option making it an effective choice for enterprise level solutions. It offers a powerful graphical user interface that can be used to control the training.

Supported Languages: Java, Python, R

Weka

Weka [23] is primarily designed to be used as a standalone program, but the code it uses is available as a Java API library. Even though the library is fairly easy to use, the customisation options are limited when compared to other libraries. The main drawback of Weka is the fact that it often encounters memory issues for even smaller datasets such as the MNIST dataset when using deeper neural networks on low-mid powered systems.

Supported Languages: Java

2.4.3 Evolutionary Algorithm Libraries

DEAP

DEAP [24] is a general purpose genetic algorithm library designed to provide basic genetic algorithm structure and implementation, while relying on the user to implement genome fitness evaluation. This means that it is suited well for small and simple problem spaces, but struggles to handle more complex tasks without heavy modification.

Supported Languages: Python

Jenetics

Jenetics [25] is also a simple genetic algorithm library similar to DEAP, but allowing for finer control over various aspects such as, Genotype and Phenotypes during the evolutionary process as well as offer greater control over aspects such as the fitness function output; by being able to manipulate the fitness value through in built library functions.

Supported Languages: Java

GA

GA [26] is a R library that offers similar functionality as many of the other languages GA packages, but with the added benefits of having graph generation built into the library through the use of the powerful graphing capabilities of R.

Supported Languages: R

3. System Design

3.1 Introduction

This section will discuss the design and architecture of the overall system choices and the choices made for other relevant topics such as development language and environment. Due to the nature of the chosen methodology no detailed plan was created for the development of the system, but instead a more informal and general outline of what the different modules would have to do and how they interact with one another was created.

3.2 Process Methodology

When considering the scope of this project and the limited resources available for development, it is important for the development methodology being followed to compliment the project. This meant that the developers limited knowledge had to be evaluated and considered while attempting to predict the project development from start to finish. This meant that plan-orientated approaches, such as "*Waterfall*", were immediately ruled out as limited "*real world*" development experience meant that predicting an accurate timeline and recognising possible pitfalls throughout development was next to impossible. Therefore, it was decided to opt for an agile based approach that would allow for development to stay much for flexible and reactive as the project grew and new information about the various technologies were discovered.

Once this had been established, it was decided that using Feature Drive Development (FDD) was the most appropriate way to develop the system as the project naturally subsections itself into modules (discussed in 3.3), and therefore seemed like the most appropriate choice. FDD works by dividing the problem into small individual "*mini-projects*" which are developed in a short time frame, allowing for progress to be tracked easily and effectively. With the project naturally falling into two main components; server and client, and with those components also naturally subdividing into three subcomponents it was relatively easy to build a feature list (3.2.1).

Apart from the above, the FDD methodology allows for a general component list to be laid out without the need for in-depth and detailed planning whilst recognising that changes to the requirements are to be expected at the project develops and matures. This fact would be a huge advantage during development due to the relative inexperience of the developer, as it allows for changes to be made as new technologies and information is learnt.

One of the other methodologies considered was Extreme Programming (XP), with slight adaptations to compensate for the lack of multiple developer to take advantage of pair-programming, as its short development time cycle combined with small manageable *"mini-project"* mindset would enable effective tracking of the project. But after consideration it was decided that the strict pace and work load would be unsustainable over a longer development as external priorities would interfere with the development and end up hindering progress.

3.3 Feature List

As mentioned previously, the project naturally finds itself in two main components; the server core and the client. And therefore, the feature list is built to reflect the logical division of the system components.

Core/Server	Module	Feature
Server	**HTTP Server**	Simple HTTP Server able to handle requests
		Allow Clients to Register with client UUID
		Assign Architectures to clients
		Return which dataset client should
		Return neural network model based on client UUID
		Return "Server Ready" status signal to clients upon request
		Process sent result using client UUID
	Evo Handler	Generate new population
		Mutate given population
		Cross-breed given population
		Return next generation
		Load population from file
	Server Core	Setup HTTP Server and Neural Network Pop
		Manage Server and Evo Handler interactions
Client	**HTTP Client**	Get new client ID and register with Server
		Get dataset and download to local file store
		Get Neural Network architecture
		Check Server is ready
		Post results to server
	Neural Net Handler	Create + train single layered neural net
		Create + train fixed size multi-layered net
		Create + train variably sized layered net
		Read dataset from file and use in neural net
	Client Core	Manage client setup with Server
		Request data from server once ready
		Manage interactions between Net Handler and HTTP Client results back to server

Table 1. Outline of the features that the project must be capable of

3.4 System Architecture

As outlined in the Feature List (3.3), the system will be made up of two main components which will, in turn, have three subcomponents. The two main components will be the server and the client modules and will be treated as two separate programs that interact only a network.

A basic overview of how the system will connect is shown in Figure 17, alongside the proposed method for distributing the population of architectures amongst the connected clients. As can be seen, the clients are only able to interact with the Server and are blind to the rest of the system and therefore can be seen as purely processing nodes while the Server acts as the control unit of the system.

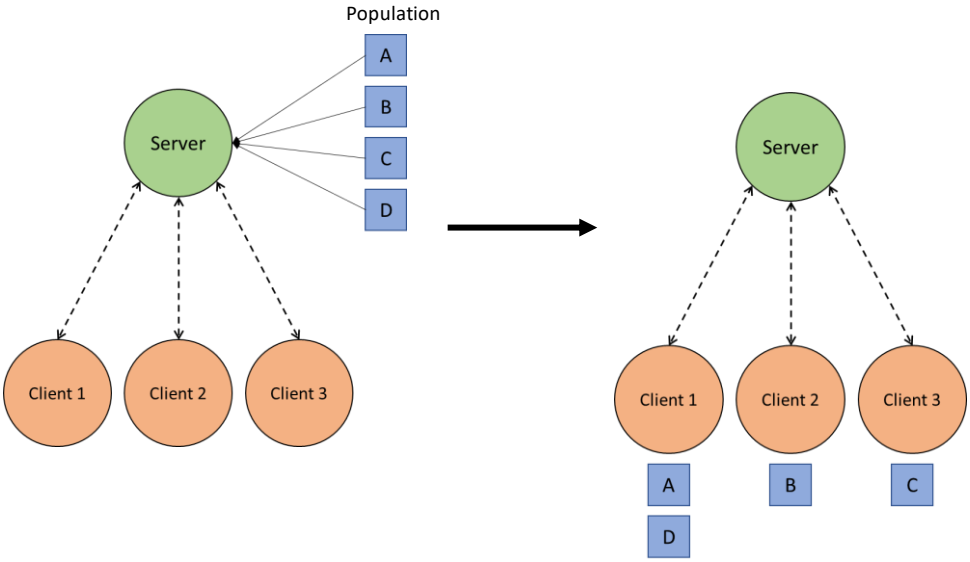


Figure 17. Diagram showing how the overall structure of the system will be structured to allow to distribution of the neural network architectures

The structures of the two main components will be very similar to one another with both consisting of three components; the communication module (HTTP), the speciality module (Evolutionary/Neural Network Handler) and the core module which will manage the interactions and general tasks of the component. How each component links to the rest of the system is illustrated in Figure 18.

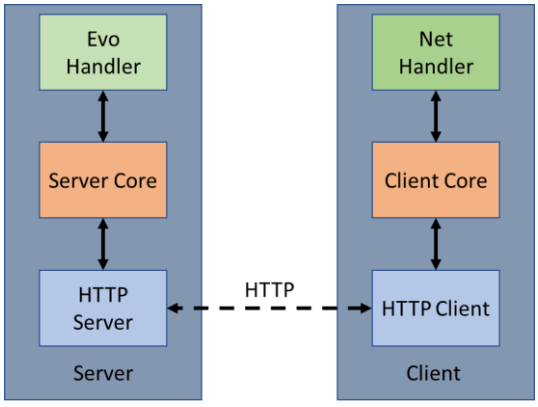


Figure 18. Overall architecture design of the system

3.5 Client - Server Communication

As the client and the server will be exclusively using HTTP as the communication technology, the requests sent must be able to transmit information such as chromosome structure or results effectively. One of the simplest way to do this would be to send the data using the request headers, but this would make sanitising the data on the other end difficult and unwieldy.

Therefore, it was decided that JSON would be used to transmit data between the two modules as it allows for variables to be labelled allowing for easier data encoding and decoding on either end of the communications.

3.6 Chromosome

It was decided to allow for the system to explore the following parameters to allow for diverse architectures and parameters to be explored while simultaneously limiting the computational cost needed with the modest capabilities available:

- Learning Rate
- Epoch Number
- Batch Size
- Network Architecture

To represent the explorable parameters in a chromosome it was decided to split the chromosome into two logical parts that describe the parameters and the hidden layer configuration respectively.

[0.025, 20, 150, 600, 500, 350]

Figure 19. The chromosome structure that will be used in the system

As can be seen in Figure 19 the chromosome is split into the two logical parts with the red representing, in order; the *learning rate, training epochs* and *batch size* while the blue represents the number of neurons in the respective hidden layer. As the parameters part of the chromosome will always stay the same length they are placed first as the number of hidden layers can change and so from the 4th element onwards can be considered to be describing the hidden layer structure of the neural network.

3.7 Genetic Algorithm Choices

As the genetic algorithm is the instrumental for the success of the system, the individual genetic operators chosen must encourage effective search space exploration. Therefore, the following components have been chosen.

3.7.1 Selection Method

To ensure that genetic diversity is encouraged, a *"roulette wheel"* selection method will be used as it allows for low scoring solutions to continue to propagate into the next generations and to spread potentially beneficial genes throughout the population.

3.7.2 Crossover Method

To maximise the diversification effect the crossover method has on the relatively small gene, the multi-point crossover method will be implemented as it will ensure that a more equal mix of the parent genes are used compared to single point cross-over.

3.8 Neural Network Choices

3.8.1 Activation Function

To ensure that the inputs of the neural networks are properly dealt and propagated through the network, the sigmoid activation function was chosen at it allows for discrete input to output mapping for a theoretically unlimited range. This means that it will be well suited for use in variably sized networks where the inputs to a single neuron are unknown prior to creation.

3.9 System Module Plan

3.9.1 Client - Neural Network Processor

This part of the system is effectively the processing centre of the system as it has to process and evaluate the different neural networks architectures that have been created. The module was designed to be able to take in the genome from the server and transform that into a viable neural network architecture, process them and return the result back to the client core.

3.9.2 Client – Core

The client core acts as the controller for the client application; managing the connection with the server and mediating the interactions with the server as well as downloading datasets.

3.9.3 Server – HTTP Server

This module would be tasked to handle incoming requests from the multiple clients such as client registration, processing requests for the next neural network model and processing returned results. This meant that it had to not only handle requests but also keep track and manage clients and the models they were being tasked to process as well as the current model generation population.

3.9.4 Client – HTTP Client

This module is responsible for carrying out the interactions with the server such as client registration, downloading datasets from the server, getting the next neural network model as well as posting the results from the neural network processor.

3.9.5 Server – Evolutionary Algorithm Handler

The purpose of this module was the create an initial population of neural network architectures and then perform genetic operations such as mutation and crossover on subsequent generations of architectures.

3.9.6 Server – Core

Similar to the Client core, this modules task was to act at the mediator between the server and the evolutionary module of the system. The core would also start off the evolutionary module and the server as two separate threads and setup the connections between the two by using things such as pipes and flags.

3.10 Technology Choices

3.10.1 Programming Language

Using the information gained from the Research the language most suitable for a project of this size and complexity should make the development of the system as easy as possible by having a wide variety of support of libraries available which can be used to speed up development instead of having to write the individual functions from the ground up. Another major factor would be cross compatibility for multiple operating systems to enable as many clients to be able to run the application as possible to help with neural network processing.

This is why the choice was made to use Python as a development language. It has a wide variety of machine learning libraries which can be utilised and implemented with little effort compared to many other languages, while also providing good general-purpose computing capabilities which will make things such as networking easier compared to other languages.

3.10.2 Libraries

The machine learning library most suited for the system would have to be Tensorflow, even when considering the steep learning curve required to use it. This is due to its great flexibility when designing and building neural network architectures which would allow for finer control of the parameters and behaviour the neural networks by the genetic algorithm compare to many of the simpler libraries.

As the programming language chosen was Python, the library used for the genetic algorithm was DEAP. As the library was designed with flexibility and personalisation in mind, it will allow for the genetic operations to be customised and optimised for the more complex chromosomes being used by the system.

4 Process and Implementation

The section will discuss the process used to develop the system in respects to the methodology as well as the final implementation of the system.

4.1 Neural Network Handler

The neural network handler main objective was to build a system that allowed for the simple chromosome, outlined in 3.6, to be converted into a working and fully parameterised neural network. This meant that development was focused on making the system flexible and able to handle networks of theoretically any size and structure. Alongside the conversion from chromosome to realised network, the Network Handler was also tasked to process the dataset through the created network.

The main challenge presented by this section of the project revolved around, as the research indicated, the complexity of implementing the Tensorflow library without prior experience of doing so. This meant that, as the feature list suggests, the implementation of the network builder was broken up into very small iterations to compensate for the steep learning curve.

Alongside this the other function of the Network Handler was to load the chosen dataset from file and format in such a way to be usable by the Tensorflow library. During the formatting of the dataset it was noticed that for equivalent network structures and settings the dataset loaded from file took around five times as long to process than the dataset loaded from the inbuild Tensorflow API. This was initially puzzling as the number of entries in both datasets were equal and therefore meant that there was a data formatting issue with the manually loaded dataset. A resolution was eventually found when it became apparent that due to Tensorflows flexible design, it allowed for different data formats (in this case Python list and Numpy) to be accepted without additional pre-processing required by the user, this combined with lack of familiarity with the framework meant that this detail was initially missed during development.

4.2 HTTP Server

The plan for the server was to be a simple way to manage the distributed clients by using predefined HTTP protocols that would enable inter as well as intra networking capabilities. But due to the FDD methodology encouraging only essential planning and an unforeseen need to run both the HTTP Server and Evolutionary Handler in separate threads (discussed in detail in 4.7 – Ready Signal), the complexity of the server grew as the project progressed.

This threading meant that the easiest way for the two parts of the system to communicate was through *pipe* and *flags* provided by the Python multithreading library instead of the planned method of using the core as an intermediate between the two. This meant that operations such as population assignment and result processing, tasks planned for the C*ore*, were moved to the *Server* instead of it acting as a simple passthrough module.

4.3 HTTP Client

Compared to *HTTP Server*, the client end of the HTTP interactions could stay as a simple passthrough to the core of the *Client*. This means that the only thing this part of the system is tasked to do is send, receive and format the responses in such a way for core to process further. As the actual interactions between the server and client never changed as a result of the changes made to the *Server*, the client behaves and acts as planned in the feature list.

4.4 Evolutionary Handler

The *evolutionary handler* was a designed to use the *DEAP* Python library to handle all the genetic operations needed for the system, but due to library limitations ended up needed heavy modification to be able to perform the actions required.

The reason why the standard *DEAP* library was unsuitable for use in this project came down to the unique properties of the chosen chromosome structure. As the system required different mutation rules to apply to different parts of the chromosome (genes were not allowed to be added or deleted in the parameter section for instance), it meant that custom genetic operations had to be written for all aspects of the library. This resulted in the *DEAP* library essentially being phased out of the system as each genetic operation was added, therefore meaning that *evolutionary handler* barely uses the intended library.

One of the main functions of the *evolutionary handler* was to manage and store the most recent generation population, but this function was given to the *Server Core* instead as the built-in evaluation function of the library required the conversion of the population to its proprietary format which would have added unnecessary complexity as none of the genetic operations of the library were being used anyway.

4.5 Client Core

The client cores intended function was to handle the overall functionality of the clients by instigating the registration and continually requesting of new network models to pass onto the *Neural Network Handler*. These functions were achieved and the end functionality of the module is near identical to the intended ones.

4.6 Server Core

As mentioned previously in the *HTTP Server* and *Evolutionary Handler*, the required functionality of modules in the server side of the system changed from the original intent and so this meant that the functionality of the *Server Core* needed to change to accommodate these changes.

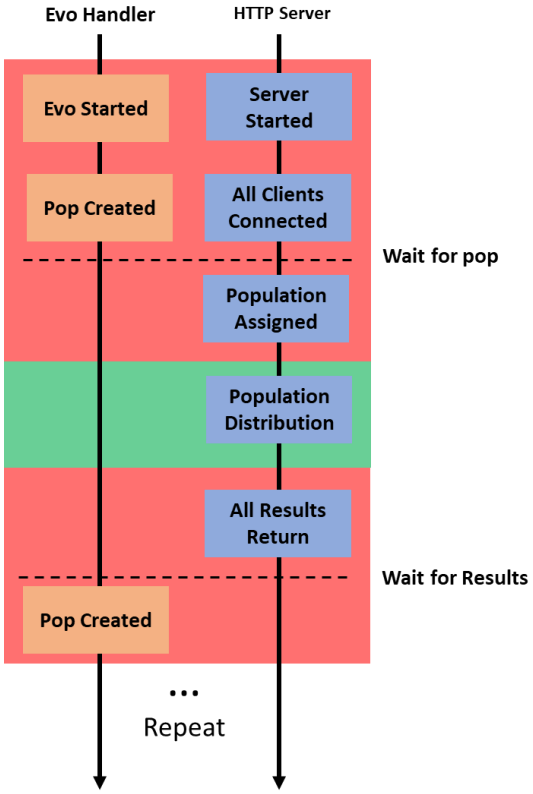
Due to the need to implement parallel threads for the *HTTP Server* and *Evolutionary Handler*, a lot of the proposed interaction handling the *Server Core* was transferred to *HTTP Server* as the data was sent directly between the two modules instead the proposed method of having the core handle the interactions. Therefore, the functionality of the *core* was greatly reduced and mainly serves a similar purpose to the *HTTP Client* as it initialises the other modules attached to it as well as handle the overall interactions between the modules. One of the functionalities that was added to the *Server Core* was the population evaluations as this was a simple task that didn't require the population to be converted to the *DEAP* libraries proprietary format.

4.7 Issues

4.7.1 Server Ready Signal

One of the biggest challenges to overcome was the implementation of the ready response that was used to inform the clients whether it was to request the architecture to process or to wait until the next population has been generated. The complication arose when the need to ensure that the server was always able to respond to requests from clients, and so *evoHandler* and *httpServer* were put onto separate threads. This meant that a way for the two threads to transfer data between one another was set up as well as a flagging system to enable the syncing up of the threads.

The problem arose out of this flagging system as the syncing between the two threads ended up being more complicated than first anticipated; causing the server to send a premature ready signal to the clients who would then try to request their next architecture to process which didn't exist. The first fix for this allowed for a single client to connect and process network, but would still act unpredictably with multiple clients connected after they had processed the initial population.



The final solution ended up needed a rewrite of the functions that controlled the ready signal and at which point the architectures were assigned to the clients. Even though this solution worked, it was not an elegant one and made what was meant to be a simple server status function that was being called several times a second by multiple clients too complicated and a potential hinderance to the projects scalability.

Figure 20. Illustration of proposed ready signal mechanics. Green area indicated when server is ready for model distribution. Dotted lines show when one thread has to wait for other to finish an operation.

5 Testing

During development of the system, a host of small tests were performed in parallel and were used to ensure that issues were addressed immediately, meaning that development adopted some aspects of Test Driven Development (TDD). This method of parallel testing was dropped once the project was nearing completion and more rigorous full system tests needed to be performed.

5.1 Mid-Development Testing

As mentioned above, some informal aspects of TDD were adopted during the development of the system but these were limited to instantiating controlled scenarios to test individual functions or subsections of the system. The main test scenarios put in place revolved around the creation and manipulation of the population between the sections of the system. In these controlled tests a population of known architectures would be created to see whether the system behaved as intended and to isolate problem areas.

As this method was both informal and only used to highlight errors within the code through console output, it is impractical to highlight these test as they are too numerous and none had enough influence on development direction to be worthy of mention.

5.2 Alpha Testing

Prior to delivering the mid-project demonstration, the project was at a stage where the fundamental features had been implemented into a semi-working state. Therefore, a suite of tests were performed in order to gauge the overall development of the project and to ascertain problem areas which needed to be addressed.

Module	ID	Function	Pass	Notes
HTTP Server	1.1	Server HTTP Requests	Yes	
	1.2	Allow Clients to Register with UUID	Yes	
	1.3	Assign Architectures to Clients	No	Works with single client not multiple
	1.4	Return which dataset	Yes	
	1.5	Return neural network model	No	Works with single client not multiple
	1.6	Ready Signal	No	Works with single client not multiple
	1.7	Process Results	No	Works with single client not multiple
Evo Handler	2.1	Generate New population	Yes	
	2.2	Mutate given population	Yes	
	2.3	Cross-breed given population	No	Not implemented
	2.4	Return next generation	No	Returns pop but 2.3 not implemented. Partially working
	2.5	Load population from file	No	Not implemented
Server Core	3.1	Setup HTTP Server and Neural Network Pop	Yes	
	3.2	Manage Server and Evo Handler interactions	No	Works with single client not multiple
HTTP Client	4.1	Get new client ID and register with Server	Yes	
	4.2	Get dataset and download to local file store	Yes	
	4.3	Get Neural Network architecture	Yes	

	4.4	Check Server is ready	Yes	
	4.5	Post Results to server	Yes	
Neural Net Handler	5.1	Create + train single layered neural net	Yes	
	5.2	Create + train fixed size multi-layered net	Yes	
	5.3	Create + train variably sized layered net	Yes	
	5.4	Read dataset from file and use in neural net	No	Not implemented
Client Core	6.1	Manage client setup with Server	Yes	
	6.2	Request data from server once ready	Yes	
	6.3	Send results back to server	Yes	

5.2.1 Changes to be made

As can be seen from the test results, the system was capable of processing new generation of architectures, distributing them, evaluating and then mutating them as long as only one client had been connected to the server. Alongside this, a few of the intended functionality is still to be implemented.

The main change that needs to be made, apart from implementing the rest of the features, was to fix the bug that stopped multiple clients from connecting to the server as this is a key part of the functionality of the system. The underlying problem causing bug looked to be related to incorrect signalling between the Server and Evo Handler threads, causing them to become out of sync, allow clients to request data that wasn't ready yet and therefore are causing errors on both the server and client side.

5.3 Beta Testing

This set of tests were performed right after all of the features had been implemented but had not undergone rigorous testing yet and therefore these set of tests were used to find overall system errors to fix prior to final release.

Module	ID	Function	Pass	Notes
HTTP Server	1.1	Server HTTP Requests	Yes	
	1.2	Allow Clients to Register with UUID	Yes	
	1.3	Assign Architectures to Clients	No	Works with single client not multiple
	1.4	Return which dataset	Yes	
	1.5	Return neural network model	No	Works with single client not multiple
	1.6	Ready Signal	No	Works with single client not multiple
	1.7	Process Results	No	Works with single client not multiple
Evo Handler	2.1	Generate New population	Yes	
	2.2	Mutate given population	Yes	
	2.3	Cross-breed given population	Yes	
	2.4	Return next generation	Yes	
	2.5	Load population from file	Yes	
Server Core	3.1	Setup HTTP Server and Neural Network Pop	Yes	
	3.2	Manage Server and Evo Handler interactions	No	Works with single client not multiple
HTTP Client	4.1	Get new client ID and register with Server	Yes	
	4.2	Get dataset and download to local file store	Yes	

	4.3	Get Neural Network architecture	Yes	
	4.4	Check Server is ready	Yes	
	4.5	Post Results to server	Yes	
Neural Net Handler	5.1	Create + train single layered neural net	Yes	
	5.2	Create + train fixed size multi-layered net	Yes	
	5.3	Create + train variably sized layered net	Yes	
	5.4	Read dataset from file and use in neural net	Yes	
Client Core	6.1	Manage client setup with Server	Yes	
	6.2	Request data from server once ready	Yes	
	6.3	Send results back to server	Yes	

5.3.1 Changes to be made

As can be seen by the tests the only issue left with the project is the same as in the alpha testing; the system is still not able to accept multiple clients at the same time. As mentioned before the problem looks originate from the flagging system used to synchronise the interactions between them.

5.4 Final Testing

The final set of tests were run after all known bugs had been fixed and the project was in a near complete state with no features left to add and only refactoring left.

Module	ID	Function	Pass	Notes
HTTP Server	1.1	Server HTTP Requests	Yes	
	1.2	Allow Clients to Register with UUID	Yes	
	1.3	Assign Architectures to Clients	Yes	
	1.4	Return which dataset	Yes	
	1.5	Return neural network model	Yes	
	1.6	Ready Signal	Yes	
	1.7	Process Results	Yes	
Evo Handler	2.1	Generate New population	Yes	
	2.2	Mutate given population	Yes	
	2.3	Cross-breed given population	Yes	
	2.4	Return next generation	Yes	
	2.5	Load population from file	Yes	
Server Core	3.1	Setup HTTP Server and Neural Network Pop	Yes	
	3.2	Manage Server and Evo Handler interactions	Yes	
HTTP Client	4.1	Get new client ID and register with Server	Yes	
	4.2	Get dataset and download to local file store	Yes	
	4.3	Get Neural Network architecture	Yes	
	4.4	Check Server is ready	Yes	
	4.5	Post Results to server	Yes	
Neural Net Handler	5.1	Create + train single layered neural net	Yes	
	5.2	Create + train fixed size multi-layered net	Yes	
	5.3	Create + train variably sized layered net	Yes	
	5.4	Read dataset from file and use in neural net	Yes	
Client Core	6.1	Manage client setup with Server	Yes	
	6.2	Request data from server once ready	Yes	
	6.3	Send results back to server	Yes	

6 Use Case Experiment and Results

As outlined in the projects Aim, the system should be capable of automatically designing, evaluating and choosing neural networks better designed to classify the dataset they are being tasked to process. Therefore, the following use case experiment was developed to test whether the system was able to deliver on the aims set out.

6.1 Experimental Parameter

The experiments will be using the MNIST Dataset due to its relatively small size to help compensate for the modest processing capabilities of the available system, as well as being class balanced which will allow for classification accuracy to be a fair measurement of performance instead of having to investigate other indicators such as ROC Area or F-Measure. MNIST Dataset is also very well documented which will allow for comparisons to other classification methods to be made. The experiments will be exploring how system performs is affected by the size of the available population.

The experiment settings for each run, par the population limit, were identical and are shown in Table 2.

Parameter	Setting
Generations	20
General Mutation Rate	0.1
Mutation Rate (Adding/Deleting Layers)	0.05
Maximum Mutation Variance Allowed	0.15
Dropout	Enabled
Forced Mutation	Enabled

Table 2. Genetic algorithm settings for experiments

The last two parameters shown in Table 2 were chosen to help encourage the development of varied architectures as neuron *dropout* has been shown to improve the classification accuracy of deeper neural networks [27], combined with *forced mutation* should allow for the development of deeper neural networks as well as shallow ones.

Alongside this the initial population of neural networks were being created at random using the ranges outlined in Table 3, with all apart from the hidden layers (due to computational limitations) being allowed to escape the initial ranges through genetic operations.

Parameter	Min	Max
Learning Rate	0.001	0.1
Training Epochs	1	20
Batch Size	10	1000
Hidden Layers	1	6

Table 3. Ranges allowed for population initialisation.

6.2 System Hardware

Here the hardware used for the system and the experiments will be outlined.

6.2.1 Server

Processor	RAM	Operating System
AMD Anthlon 2.6GHz	4GB	Ubuntu 16.04.2

Table 4. Server hardware

6.2.2. Clients

Processor	RAM	Operating System
i5-5200 2.20GHz	8GB	Ubuntu 16.04.2
Intel Atom 1.66Ghz	1GB	Ubuntu 16.04.2
Intel Atom 1.66Ghz	1GB	Ubuntu 16.04.2
Intel Atom 1.66Ghz	1GB	Ubuntu 16.04.2
Intel Atom 1.66Ghz	1GB	Ubuntu 16.04.2
Intel Atom 1.66Ghz	1GB	Ubuntu 16.04.2

Table 5. Client hardware

6.3 Classification Accuracy

With classification accuracy being used as the fitness measure of an individual organisms in the population, it is the key metric which the performance of the system is evaluated with. Table X shows the final results of the experiments.

Population	Average	Max	Min
40	0.9094	0.9136	0.9078
20	0.9057	0.9136	0.9016
10	0.9012	0.9047	0.8986
5	0.8869	0.8888	0.8849
3	0.8900	0.8913	0.8885

Table 6. Final results from experiments after 20 generations

The general trend shown in shows a direct correlation between the size of the population and the final average performance except for the 3 and 5 sized population experiments, though the reason why this occurs may be explained when looking at Figure X.

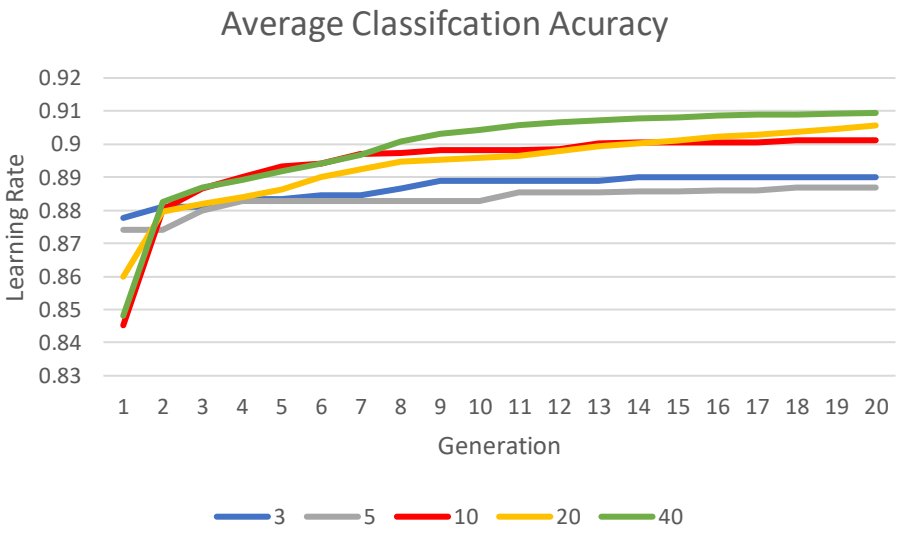


Figure 21. Average classification performance over generations

As can be seen in Figure 21 the initial population created for the 3 population experiment had a slightly higher average classification accuracy than the 5 population (0.0036%) yet ended up with a classification rate that was similarly better by a very similar amount (0.0031%). From this a conclusion could be drawn that the reason why the 3 population experiment outperformed the 5 population one is simply due to the stochastic nature of genetic algorithm initialisation in combination with very similarly small population resulting in genetic stagnation around the same classification accuracy.

When looking at the rest of the data presented in Figure 21, it becomes apparent that the higher population counts enable the 10, 20 and 40 experiments to benefit from larger populations and therefore are able to explore more solutions per generation while sustaining better genetic diversity for longer than their smaller popultion counter-parts. These advantages the larger population experiments had meant that they all reached a higher average classification accuracy, with 20 and 40 seemingly trending towards the same maximum classification rate the experimental settings allowed. With the possible reasons why there was a maximum classification rate with the system being explored below in 6.4.

6.4 Genetic Diversity

As outlined in [15], genetic diversity is an important factor when using genetic algorithms as it gives a wider availability of diverse solutions within the population that could potentially provide new and more successful solutions than current leading ones. Therefore, the roulette selection method was chosen as it theoretically allows for initially low scoring species and traits to still proliferate into the next population.

During the run through of all the experiments though, a noticeable trend towards smaller gene pools was seen across all the populations. This was likely due to the roulette wheel selection method tending to favour the reproduction of the most successful organisms causing similar successful organisms to propagate [28] their genes through the population and for an eventual *converged allele* [29] effect to occur.

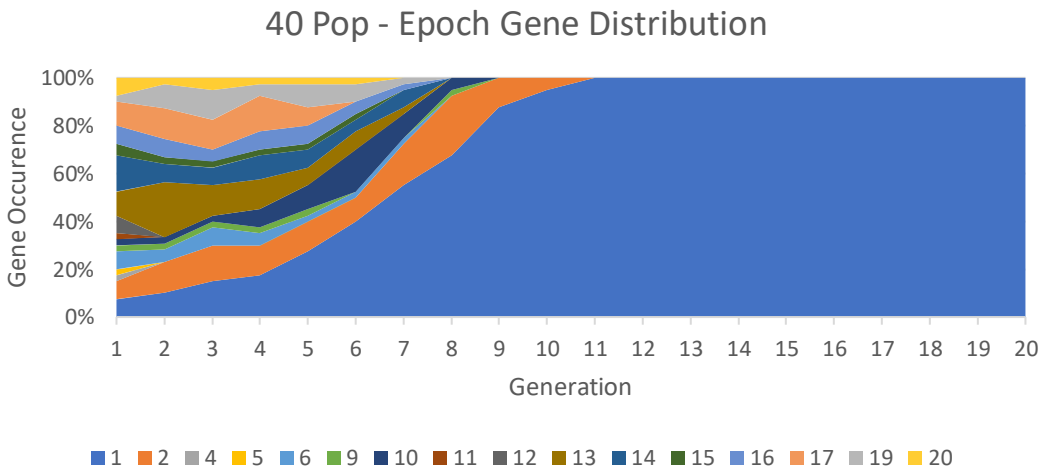


Figure 22. Epoch Gene occurrence throughout the 40 population experiment

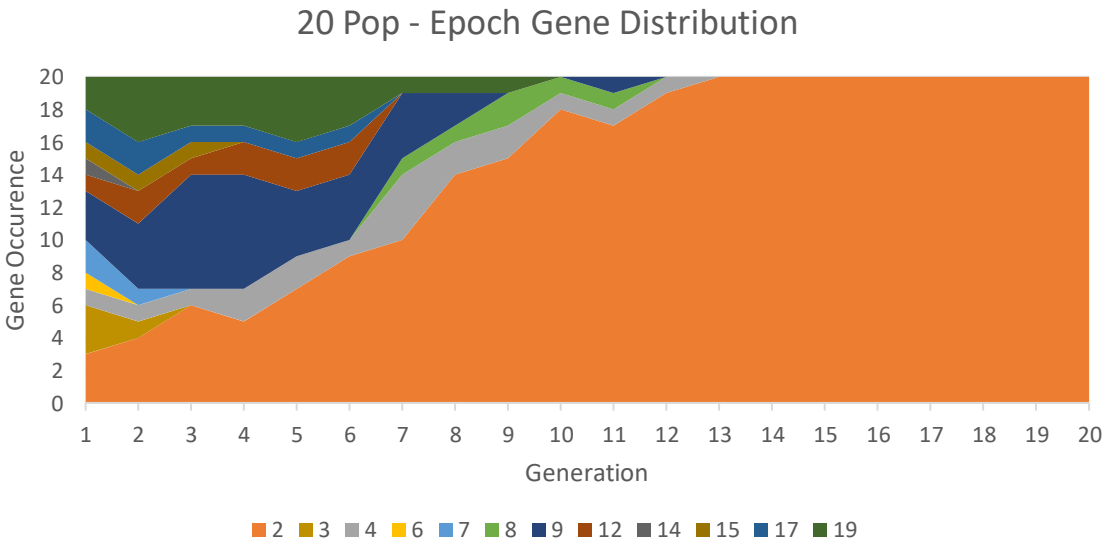


Figure 23. Epoch Gene occurrence throughout the 20 population experiment

As can be seen by both Figure 22 and Figure 23, low epoch genes quickly came to dominate the population which in turn favour would result in complimentary genes to also become common such as higher learning rates needed to train the networks over few epochs, thus making it less likely for an alternate solution to spontaneously mutate competitive alternative genes. This correlation is further supported in Figure 24.

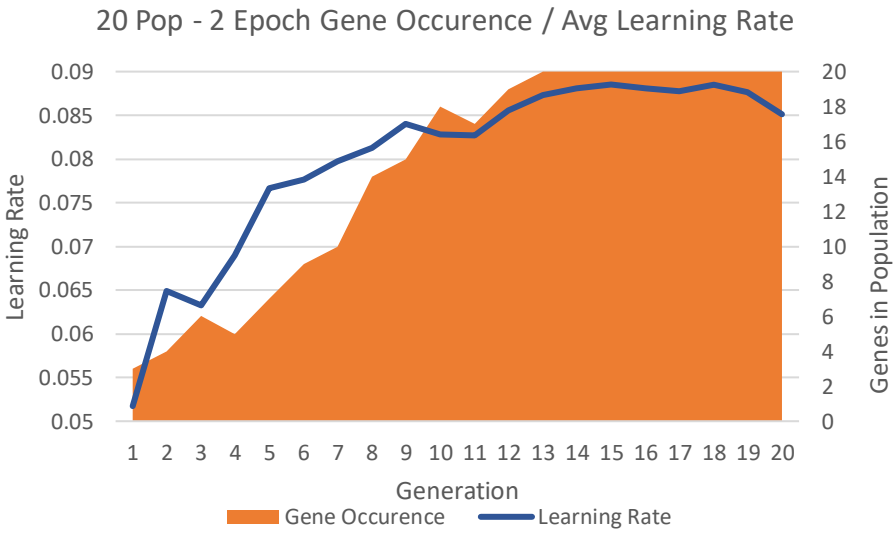


Figure 24. Relationship between the gene dictating 2 epochs and the average learning rate for the 20 population experiment

The above suggesting that the two genes are intrinsically linked is further supported by the results of the 10 population experiment, which shows a dominance of genes dictating higher epoch numbers than those found in both the 20 and 40 forcing lower learning rates as seen in Figure 25.

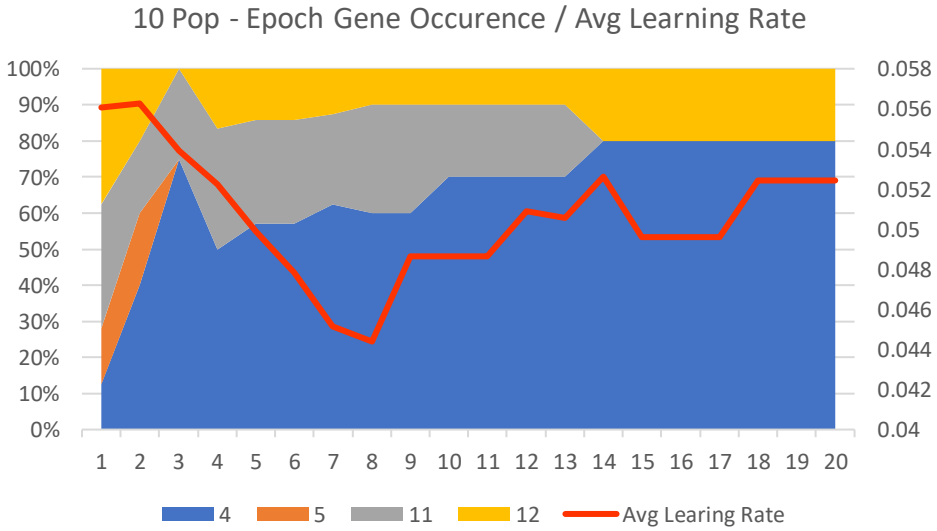


Figure 25. Relationship between Epoch Gene Occurrence and Average Learning Rate for 20 pop experiment

6.5 Final Structures & Parameters

Population	Hidden Layer Structure	Epochs	Batch Size	Learning Rate	Accuracy
40	[675]	1	785	0.107721823	0.9136
20	[642]	2	1133	0.089601745	0.9136
10	[687]	4	874	0.058695653	0.9047
5	[284, 332, 263]	10	471	0.101488832	0.8888
3	[354, 591, 706]	6	665	0.07338447	0.8913

Table 7. Final neural network structures and parameter settings for most successful individuals

Table 7 gives further evidence that the conclusion drawn from 6.2 is correct as system looks to optimise the parameters and architectures of neural networks as the larger population (10, 20 and 40) experiments all have similar layer architectures yet have learning rates inversely proportional to the epochs which, through knowledge of how neural networks train, would be the correct way to set the parameters for such an architecture.

6.6 Discussion

As mentioned above, the results of the experiment showed that the system worked, yet struggled to perform as intended due to extreme genetic stagnation. This would indicate that the genetic algorithm was not working as intended, but after reviewing the data it becomes apparent that the fault lies with the experimental settings rather than the system itself.

The reasoning behind this argument is rooted in the low average epoch amount given to the starting population. This was intended to be a compromise between offering architectures enough room to evolve above that limit if needed (as epoch mutation was not limited) and the modest computational capabilities available to the system. These settings were meant to allow for deeper networks to be able to evolve higher epoch numbers, while simpler ones would develop needing fewer epochs to help minimise overall computational cost. Due to this decision, deeper neural networks struggled to gain significant foothold are were easily outcompeted as can be seen in Figure 26 and Figure 27.

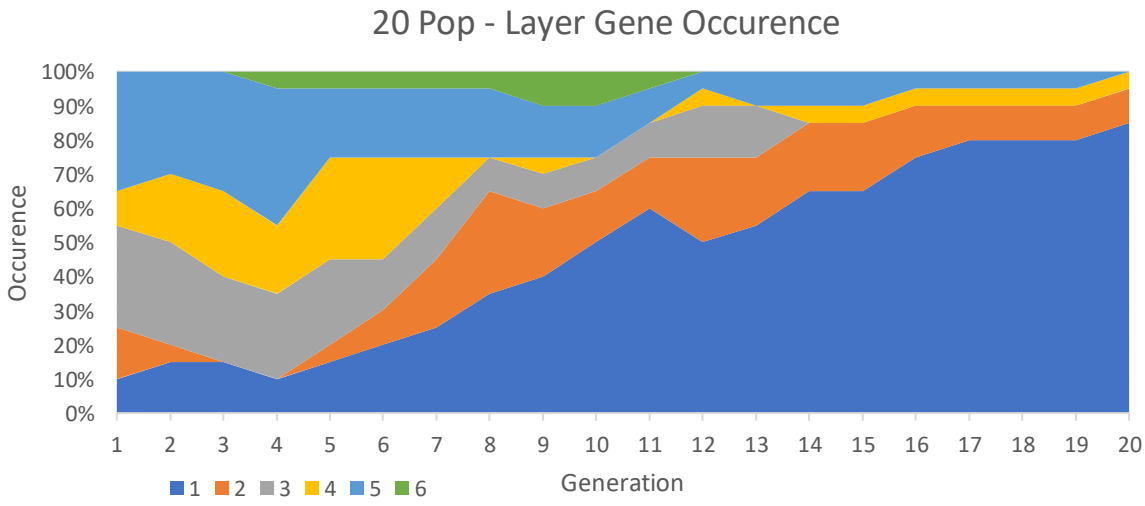


Figure 26. Architecture depth over time for the 20 experiment

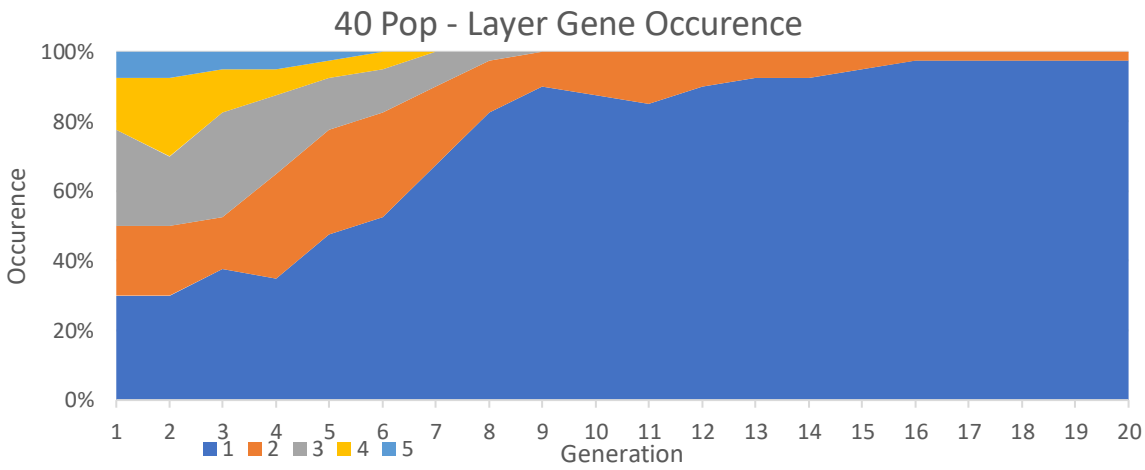


Figure 27. Architecture depth over time for the 40 experiment

This inability for deeper networks to gain an early foothold in combination with the success of an organism being heavily dictated on the harmonious pairing of the learning rate and epoch number, made the spontaneous mutation of a competitive alternative organism very unlikely. Alongside this the selection methods inability to promote genetic diversity compounded the issue until genetic stagnation occurred and the chances of an alternate solution being explored neared zero.

To explore whether this speculation has grounds for reason, a final experiment was conducted with the exact same parameters are seen in Table X, but with a raised initialisation limit of 100 epochs and a population of 20. The classification accuracy results shown in Table X and show that similar accuracies are achieved to the other experiments.

Population	Hidden Layer Structure	Epochs	Batch Size	Learning Rate	Accuracy
20	[770, 712, 661]	94	610	0.031163	0.9081

Table 8. Most successful organism for 20 Pop 100 Epoch experiment

Even though the classification results don't indicate that these settings affect the classification in any significant way, Figures 28 and 29 show greater genetic diversity as well as speciation occurring due to better experiment parameter settings. This means that the chances for more suited parameters being discovered in future generations is increased significantly compared to the original experiments as a larger area of the solution space is being explored.

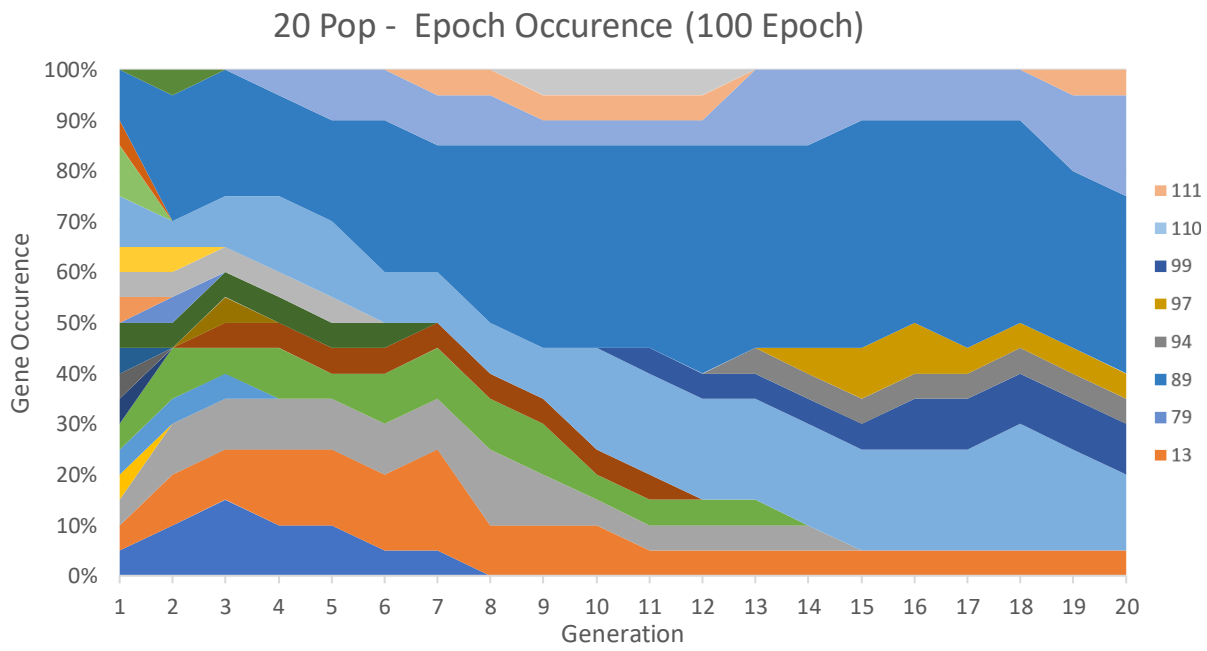


Figure 28. Epoch gene occurrence for 20 Pop experiment with 100 epoch as initial max limit (Only finalists are show in legend)

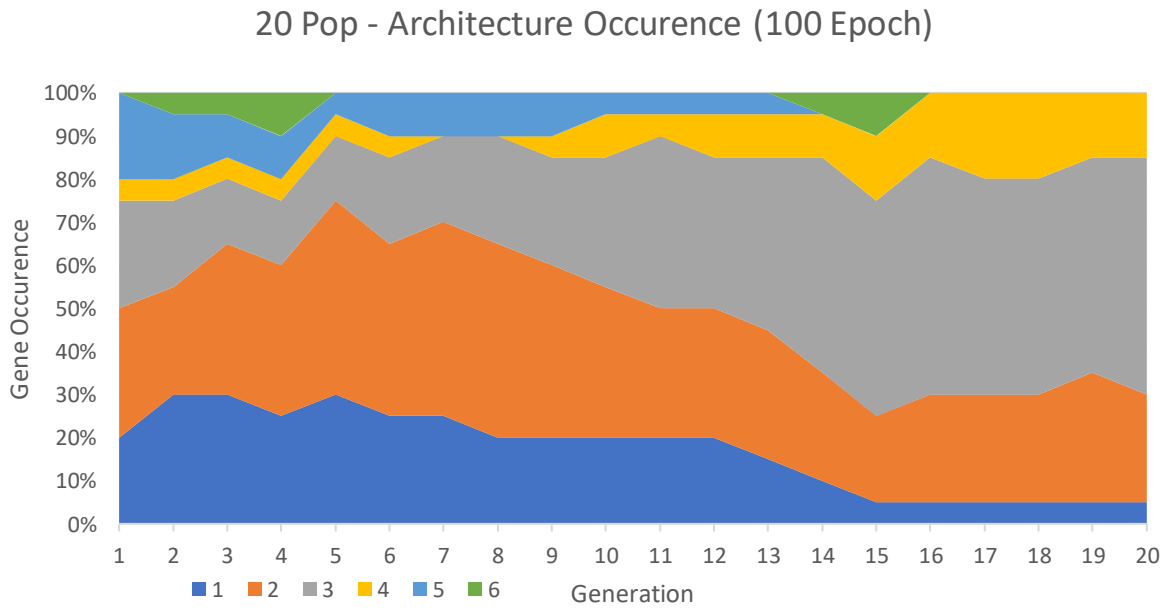


Figure 29. Layer depth occurrence for 20 Pop experiment with 100 epoch as initial max limit

7 Critical Evaluation

This section will explore the how well the project met the aims and objectives set out at the beginning of this report, as well as give insights into any shortcomings the project has along with suggestions of how to improve the system in future iteration. Alongside this the development process of the project will also be evaluated and whether the implemented methodology was an effective way to manage this project.

7.1 Aim

The aim of the project was to create a system that is capable of generating a population of random neural network architectures and through iterative genetic operations generate improved neural networks that are better at the classification task they are being asked to perform. To this, the project has succeeded in producing a system capable of that exact criteria, though this success cannot be given so boldly without explaining some of the shortcomings the system has.

The system does indeed produce more competitive neural network architectures, but the improvements gained over the average of a purely random are minimal when considering the computational cost required by the system for larger populations.

7.2 Research & Development Choices

The research performed was one of the more extensive parts of the project which allowed for a thorough exploration of different options for both tools and technologies. This meant the project could be focused on actual development rather than explorative searches to find the best fits for the project. Alongside this, the deep knowledge gained about the algorithms implemented meant that useful insights about the systems behaviour could be gained from the results and therefore help in the suggestions for future work on the system.

One of the best choices based on the research, was the decision to use Python as the development language as it offered great flexibility and forgiveness during some of the more difficult parts of the project. This praise does come with a small caveat as the flexibility Python allowed lead to situations where the interpreter would force bad code to work where other and stricter languages would have thrown errors, thus resulting in extended development time.

7.3 Experimentation & Results

One of the major limitations of the project are made apparent in the results section that show that the system itself can achieve its purpose but through incorrect experimental settings and design choices made early on in its development it was hampered from exploring this concept to its full potential.

As explained in the results section, the lack of genetic diversity seen within the population had huge negative consequences on the results produced by the system and therefore the genetic algorithm is the weakest part of the system. The reason why this is, revolves around three main areas; the selection method, experimental parameter settings and a limited chromosome.

7.3.1 Selection Method

Due to the roulette wheel selection method favouring the proliferation of better performing organisms without concern for genetic diversity, it became incredibly easy for a snowballing effect to occur where genes found in early well performing organisms would spread incredibly fast. This meant that other genes inside those organisms would have to be well suited for that fast spreading gene, therefore the chances of a competitor species being created or surviving alongside the dominant species unlikely.

7.3.2 Experimental Parameter Settings

When considering the experimental parameter settings outlined in the choice to create an initial population with a maximum of 20 epochs was one of the largest contributing factors to the genetic stagnation of the system. This initial low maximum epoch number was put in place to compensate for the modest computational capabilities available to the system. With the intension that species which benefitted from higher epochs, presumably ones with deeper networks, evolving these characteristics to explore these higher and more computationally expensive parameters. This theoretically would have worked if the selection method chosen promoted genetic diversity even species were underperforming. Therefore, these areas in the solution space were never explored and resulted in extreme genetic stagnation as individual gene settings were too intrinsically linked to others for successful competitor organisms to spontaneously mutate once stagnation occurred.

7.3.3 Limited Chromosome

Another reason that cannot be proven without further experimentation, but can be theorised with the information presented, it the fact that the relative small size of the chromosome instead of benefitting the search by limiting the search space, instead acted against the optimisation.

As mentioned before, certain gene settings complement one another as demonstrated in Table X by 10,20 and 40 all having low epochs, high learning rate and a shallow neural network at intervals that suggest a pattern. The fact that these patterns formed so well and after relatively few generations, may indicate that the limited chromosome only gave

7.4 Comparison to other Methods

As mentioned in 6.1, the reason why the MNIST Dataset was used as it is a widely used dataset and therefore comparison between various classification methods are easily attainable. Below is a table of some of the classification accuracies taken from [30] found on the official MNIST Dataset website [31]:

Classifier	Pre-processing	Error Rate (%)
Linear classifiers (1-layer NN)	None	12.0
K-nearest Neighbors, Euclidean (L2)	None	5.0
2 Layer NN, 300 hidden units, MSE	None	4.7
Convolutional net LeNet-4	None	1.1
Using GA to Design NN Architecture	None	8.6

Table 9. Comparison of classification results between suggested and other methods

As can be seen in Table 9, the performance of the suggested method is not as good as others that have attempted the same problem.

7.4 Improvements and Future Work

As the results show, the system is capable of exploring and selecting neural network parameters and architectures that improve classification accuracy, and so it successful in what it was meant to achieve. This being said, there are several improvements that can be offered as suggestions for future work in hopes to improve the systems capabilities.

7.4.1 Selection Method

One of the main issues this project suffered from is extreme genetic stagnation and even though the results give indications that this is due to improper experiment parameter settings, it doesn't negate the fact that the selection method implemented was ill equipped to handle such parameters and contributed heavily towards the genetic stagnation.

Therefore, one of the main changes that should be implemented in future version is a selection method that helps to promote genetic diversity such as a variation of "*Island Model Genetic Algorithms"* [32]. This parallel method encourages the formation of small independent populations of organisms which are exclusive to one another and so are able to explore different areas of the search space without competition from initially successful solutions. Implementing such a selection method would enable experiments with restrictive initial parameters, as the ones given for this paper, a fair chance to explore the total search area effectively.

7.4.2 Solution Space Exploration

Another improvement that could be made to this project is to allow for a greater solution space to be explore by increasing the parameters being explored by the genetic algorithm. This would allow for more interesting and diverse solutions to be explored as the results of this experiment show very little deviation from optimisation of epoch and learning rate optimisation.

Other parameters that could be explored could be whether the network should use learning rate decay [3], weight decay [33] or even allow for the chromosome to directly dictate how the individual neurons connect with one another as seen in Neuroevolution [34].

7.4.3 Development & Methodology

Using FDD to develop this project meant that it allowed for greater flexibility and so was forgiving during times when new technologies needed to be learnt and explored. Even though this proved to be a positive as a few of the modules needed to be reworked to accommodate for unforeseen challenges and requirements, this also means the project suffers from certain aspects not being as well structured as originally intended.

Therefore, if this project was to be reworked with knowledge gained so far from the outset, FDD would still be the methodology used but with a more defined plan of how the system is to be constructed as well as defining how the individual models would communicate with one another.

7.5 Conclusion

Considering the relative inexperience of the developer and even though multiple potential changes and improvements have been highlighted, the project has to be labelled successful. It achieved what it set out to and has provided a solid foundation to start from to further explore this concept.

8 Appendices

8.1 Appendix A – Third Party Technologies, Code and Libraries used

8.1.1 General:

Python 3.6.5: This was the development language used and with expectation of the third-party libraries list below all the code present was developed by Marc Cooper.

8.1.2 Server:

OS: This was used to open and present files for the server to serve to clients as well as save to and load from files

UUID: This library was used to generate a random ID number for both the clients and the population of architectures

HTTP.Serve: This was the framework implemented, without modification to the source code, to server the HTTP Server as well as a foundation to handle incoming HTTP Requests

JSON: This was used to decode and encode JSON data from and into HTTP requests for data transfer between client and server

Threading: This was used to separate and manage the *Evo Handler* and *HTTP Server* threads

Multiprocessing: Used to transmit data and flags between the threads

Time: Used to display a waiting timer for the user

CSV: Used to load and export population data from and to CSV files

Random: Used as a way to generate random number used in population control as well as all genetic operations

DEAP: Library functions were used but edited to suit the custom genetic operations required from the project

Itertools: Dependency of the DEAP Library. Used for iterating over the population

Collections: Dependency of the DEAP Library. Used for managing object collections such as lists

8.1.3 Client:

Tensorflow: Neural network framework used to build and process neural networks

Urllib: Used as a framework for the http requests as well as parsing return data from the server

Tarfile: Used to unpack tarfiles downloaded from the server

Numpy: Used as the data format to convert the dataset into and be used by Tensorflow as vector inputs

Tqdm: Used to visually display the progress of the epochs and batches being processed client side

Appendix B – Ethics Form

AU Status
Undergraduate or PG Taught

Your aber.ac.uk email address
mnc3@aber.ac.uk

Full Name
Marc-Osric Cooper

Please enter the name of the person responsible for reviewing your assessment.
Chuan Lu

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
cul@aber.ac.uk

Supervisor or Institute Director of Research Department
cs

Module code (Only enter if you have been asked to do so)

Proposed Study Title
Using Evolutionary Algorithms to Design Neural Network Architecture and Hyperparameter Selection

Proposed Start Date
1st February 2018

Proposed Completion Date
4th May 2018

Are you conducting a quantitative or qualitative research project?
Qualitative

Does your research require external ethical approval under the Health Research Authority?
No

Does your research involve animals?
No

Are you completing this form for your own research?
Yes

Does your research involve human participants?
No

Institute
IMPACS

Please provide a brief summary of your project (150 word max)

I propose to develop a system that will automatically explore neural network architectures and hyperparameters, through use of a genetic algorithm in an effort to improve neural network classification accuracy.

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?
Not applicable

Will appropriate measures be put in place for the secure and confidential storage of data?
Yes

Does the research pose more than minimal and predictable risk to the researcher?
Not applicable

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?
No

Please include any further relevant information for this section here:

If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.
Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.
Yes

Please include any further relevant information for this section here:

Bibliography

[1] S. Wolfram, "Statistical Mechanics of Cellular Automata," in *Cellular Automata And Complexity*, CRC Press, 1994, pp. 1-67.

[2] T. Knight, "Computing with emergence," in *Environment and Planning B: Planning and Design*, Department of Architecture, School of Architecture and Planning, Massachusetts Institute of Technology, Cambridge, MA, 2002, pp. 125-155.

[3] R. P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine,* pp. 4-22, April 1987.

[4] K. Matsuoka, Noise injection into inputs in back-propagation learning, 1992.

[5] I. Tetko, D. Livingstone and A. Luik, "Neural Network Studies. Comparison of Overfitting and Overtraining," University of Portsmouth, 1995.

[6] Sigevo, "GECCO 2017," [Online]. Available: http://gecco-2017.sigevo.org/index.html/HomePage. [Accessed 7 April 2018].

[7] M. Suganuma, S. Shirakawa and T. Nagao, "A Genetic Programming Approach to Designing Convolutional Neural Network Architecture," GECCO 2017, 2017.

[8] H. Cragon, "Von Neumann Model," in *Computer Architecture and Implementation*, Cambridge Press, 2000, pp. 2-6.

[9] D. Freedman, Statistical Models: Theory and Practice, Cambridge University Press, 2009.

[10] N. Chistianini and J. Shawe-Taylor, "Linear Learning Machines," in *An Introduction to Support Vector Machines and other kernel-based learning methods*, Cambridge University Press, 2000, pp. 9-11.

[11] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1," 2016.

[12] D. Rummelhart, G. Hinton and R. Williams, "Learning Representation by Back-Propagating Errors," in *Nature Vol 323*, Nature Publishing Group, 1986, pp. 533 - 536.

[13] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton and G. Hullender, "Learning to Rank using Gradient Descent," in *Proceedings of the 22nd International Conference on Machine Learning*, Bonn, 2005, pp. 89-96.

[14] M. Gen and R. Cheng, "Foundations of Genetic Algorithms," in *Genetic Algorithms and Engineering Optimization*, John Wilet & Sons, INC., 2000, pp. 1-40.

[15] Y. Leung, Y. Gao and Z.-B. Xu, "Degree of population diversity - a perspective on premature convergence in genetic algorithms and its Markov chain analysis," 1997.

[16] H.-P. Schwefel, Evolution and Optimum Seeking pp 151 - 160, 1995.

[17] R Core Team, “R-Project,” [Online]. Available: https://www.r-project.org. [Accessed 22 April 2018].

[18] Python Software Foundation, “Python,” [Online]. Available: https://www.python.org/. [Accessed 10 April 2018].

[19] Oracle, “Java,” [Online]. Available: https://java.com. [Accessed 11 April 2018].

[20] Google, “Tensorflow,” [Online]. Available: https://www.tensorflow.org/. [Accessed 15 April 2018].

[21] Keras, “Keras,” [Online]. Available: https://keras.io/. [Accessed 30 April 2018].

[22] H20.ai, “H20,” [Online]. Available: https://www.h2o.ai. [Accessed 25 April 2018].

[23] University of Waikato, “Weka,” [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/. [Accessed 19 April 2018].

[24] Computer Vision and Systems Laboratory (CVSL) at Université Laval, “DEAP,” [Online]. Available: http://deap.readthedocs.io. [Accessed 1 May 2018].

[25] F. Wilhelmstötter, “Jenetics,” [Online]. Available: http://jenetics.io/. [Accessed 16 April 2018].

[26] L. Scrucca, “R-Project / GA,” [Online]. Available: https://cran.r-project.org/web/packages/GA/vignettes/GA.html. [Accessed 24 April 2018].

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” Journal of Machine Learning Research 15, 2014.

[28] J. E. Baker, “Adaptive Selection Methods for Genetic Algorithms,” in *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, 1985, pp. 101-106.

[29] K. A. De Jong, “Ananlysis of the behaviour of a class of Genetic Adaptive Systems,” University of Michigan, 1975.

[30] Y. LeChun, L. Bottou, Y. Bengio and P. Haffer, “Gradient-Based Learning Applied to Document Recognition,” Proceedings of the IEEE, 1998.

[31] Y. LeCun, C. Cortes and C. J. Burges, “The MNIST Datasbase,” [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed 7 April 2018].

[32] R. Tanese, “Parallel Genetic Algorithm for a Hypercube,” in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, 1987, pp. 177-183.

[33] A. Krogh and J. Hertz, “A Simple Weight Decay can Improve Generalization,” 1992.

[34] D. Floreano, P. Durr and C. Mattiussi, “Neuroevolution: from architectures to learning,” Springer, 2008.

[35] A. Krizhevsky, in *Learning Multiple Layers of Features from Tiny Images*, 2009, pp. 32-35.

[36] G. Griffin, “Caltech 256,” 15 November 2006. [Online]. Available: http://www.vision.caltech.edu/Image_Datasets/Caltech256. [Accessed 7 April 2018].