Biocomputation: Data Mining Worksheet

Alfie Lamerton 19013233

1 INTRODUCTION

This assignment requires the development of a genetic algorithm (GA) for evolving the weights of an artificial neural network (ANN), such that said ANN performs optimally on 3 provided sets of data. This report is an opportunity to demonstrate an understanding of how learning can be seen as a search process and the effect parameter changes have on the ability of search techniques to solve optimisation problems. This approach is characterised by the systematic modification of a variety of parameters and informed by a number of journal articles and reports.

2 BACKGROUND RESEARCH

Cuckoo Search

Developed by Xin-She Yang and Suash Deb in 2009, Cuckoo Search is an optimisation algorithm based on the proclivity of some species of cuckoo, such as Ani and Guira cuckoos [1], to lay their eggs in the nests of other birds. The developers extracted the following three rules, inspired by the nesting habits of these cuckoos:

1. Each cuckoo lays 1 egg at a time and dumps it in a randomly chosen nest.
2. The best nests with the high-quality eggs will be carried over to the next generations.
3. There is a fixed number of available host nests, and the equation 1 shows the probability of the egg laid by the cuckoo being discovered by the host bird.

Equation 1

$$p_a \in (0, 1)$$

The algorithm is also enhanced by a type of random walk (a random method of vector movement) called Lévy Flights, inspired by the behaviour of things such as fish, birds and insects which dictate the movement of the cuckoos when searching for nests.

Comparison to the Genetic Algorithm

In a journal article comparing a genetic algorithm and cuckoo search, it was found that the CS took twice as long as the GA because of the structural nature of the CS. It also found the CS to be slower than the GA per generation. [2] However, a comparison of CS with GA on an NP-complete job-scheduling minimisation problem found the CS to outperform the GA [3] Furthermore, another report obtained better results on 5 benchmark mathematical functions with CS than with a GA, and even concluded that with better parameter tuning, CS may have performed better. [4]

Akin to GAs, CS has been used to optimise ANNs. [5] CS and GAs have been combined to produce effective results on other benchmark optimisation problems. [6] Despite its success with optimisation, CS has been criticised for 'not bringing any novel ideas to optimisation'. [7] The report claims CS clearly resembles another evolutionary algorithm called (μ+λ)–ES, and concludes not only that CS is not useful, novel or soundly motivated, but also that it doesn't seem closely related to the behaviour of cuckoos at all.

Comparison to Other Evolutionary Algorithms

A comparative report of CS and Ant Colony Optimisation (ACO) found no significant advantage of either algorithm, aside from the platform independence and portability of CS compared with ACO. [8] A report comparing CS with Particle Swarm Optimisation (PSO) with respect to optimising power flow to distributed power systems found CS to outperform PSO. CS was found to converge (settle on a range of solutions close to the best) and solve faster than PSO.

Firefly Algorithm

Another nature-inspired optimisation algorithm developed by Xin-She Yang is the Firefly Algorithm (FA). The inspiration for the FA is the tendency of fireflies to flash intermittently. This behaviour pattern is transmuted into the algorithm whereby fireflies represent candidate solutions, and firefly brightness represents their performance on the objective function with the following rules: [9]

1. Fireflies are unisex, so attraction disregards a firefly's sex.
2. The attractiveness of a firefly depends on its brightness, and brightness decreases as the distance apart of two fireflies decreases.
3. The brightness of a firefly is determined by the landscape and the objective function

Comparison to the Genetic Algorithm

A comparative study of GAs and the FA for an optimisation problem concerning flexible job scheduling found the FA to converge faster than the GA, suggesting that the FA was 'more effective and efficient' than the GA. [10] Another study found the GA easier to implement than the FA. However, the FA was found more efficient and robust. [11] In a study comparing the GA and FA for variable selection in spectroscopy, the FA was found to optimise more effectively than the GA. [12] An article was written in the Journal of Environmental Management comparing the FA and GA when combined with an adaptive neuro-fuzzy inference system (ANFIS) for the prediction of wildfire probability. It found that when combined with the ANFIS, the GA was more accurate and converged faster on a minimisation function. [13]

Comparison to Other Evolutionary Algorithms

A report comparing the FA with PSO for finding the source of emissions using air sensors found the FA to optimise better and find results faster than the PSO. It also found that the FA was more effective than PSO at optimising when the data being used contains noise, or if the problem contains many local optima. [14] Another study comparing FA and PSO also found that the FA performed better on data with high noise levels, but also that PSO converged faster than PSO.

3 EXPERIMENTATION

This report demonstrates the optimisation of the weights of an ANN using a GA. ANNs are mathematical models of the vastly interconnected networks of biological neurons present in human brains. Like GAs, they are inspired by nature and introduce biological concepts mathematically, namely neurons and neural networks. [15]. A multilayer perceptron (MLP) is a kind of ANN, characterised by two things, being feedforward and fully connected, meaning they only process information in one direction, from the input to the output. MLPs exist under the supervised learning paradigm of machine learning, the purpose of which is to allow for the creation of a system that learns to predict the desired output expected from a given input. [16]

The algorithm in this report is example of Neuroevolution, a form of Artificial Intelligence that uses

evolutionary algorithms in combination with ANNs, in this case to optimise the weights of an ANN. Neuroevolution employs evolutionary computing as an alternative to training ANNs using backpropagation. This report is an exhibition of the ability of evolutionary computing to optimise an ANNs weights instead.

This algorithm is written in the Python programming language. In the GA, the individuals are represented by an Individual class. They are encoded with chromosomes, which are represented by lists of a fixed length of floating-point numbers in a fixed range and held in the Individual class's chromosome attribute. These floating-point numbers are known as genes. The algorithm has three main operators, characteristic of a GA: mutation, selection, and crossover.

Population Size and Number of Generations

The objective of the first set of experiments was to find optimal settings for two variables: population size and number of generations. The experimentation started with the following encodings as a starting point:

Table 1

Provided Dataset	2
Population Size	50
Number of Generations	50
Number of Iterations	10
Mutation Rate	0.2
Mutation Step	0.2
Crossover	False
Gene Minimum	-1
Gene Maximum	1
Data Splitting Ratio	80/20
Number of Hidden Nodes	3

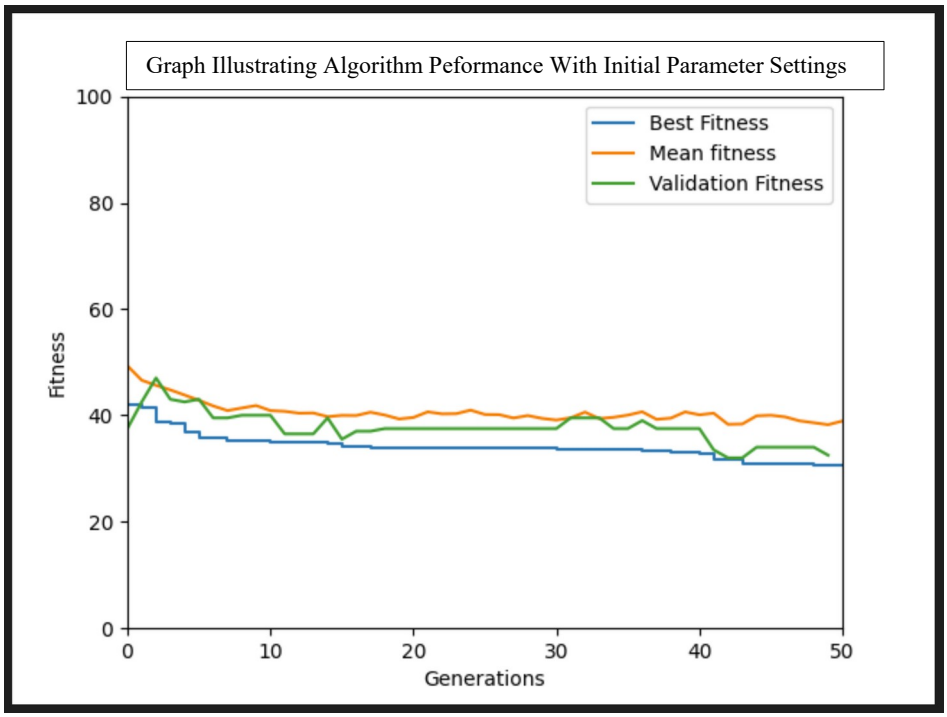


Figure 1

Running the algorithm with these encodings produced a mean lowest fitness of 31.9%.

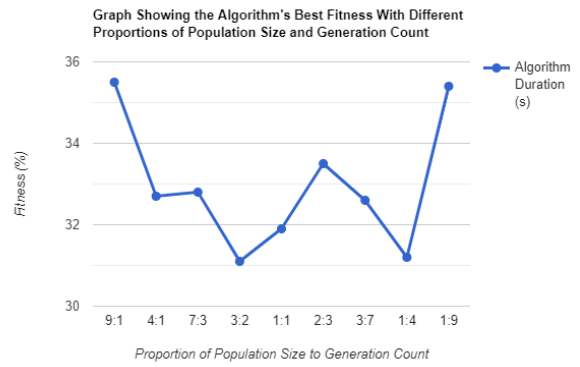


Figure 2

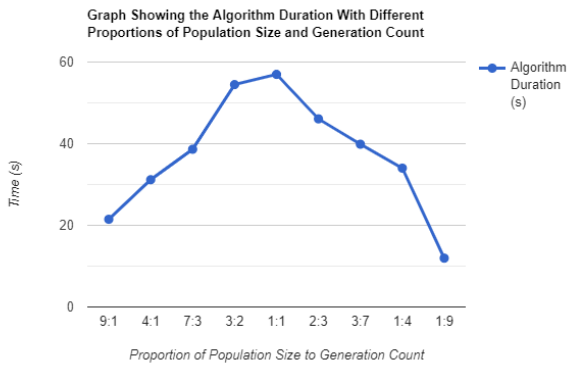


Figure 3

A report comparing the effects of increasing population size in proportion to the number of generations suggested that doing so was more beneficial for minimisation. [17] However, experiments with a variety of population/generation ratios produced different results. It can be observed in figures 2 and 3 that ratios 3:2 and 1:4 produce the lowest fitness, but that the ratio 1:4 has a lower runtime with little sacrifice to fitness. For this reason, the ratio of population size to number of generations was set to 1:4 for the rest of the experiments in this assignment. Interestingly, this is another example of the pareto principle.

Table 2

Provided Dataset	2
Population Size	*20*
Number of Generations	*80*
Number of Iterations	10
Mutation Rate	0.2
Mutation Step	0.2
Crossover	False
Gene Minimum	-1
Gene Maximum	1
Data Splitting Ratio	80/20
Number of Hidden Nodes	3

Running the algorithm with these encodings produced a mean lowest fitness of *32.2%.*

The next thing to set was the actual population size and number of generations. In an ideal world, the actual sizes of these variables could be tinkered with to no end, and a concrete conclusion about their size may be drawn. Alas, a compromise was made here with due to the runtime of the algorithm. As the actual sizes of these variables increase, the overall performance of the algorithm increases. The parameters listed below are based on this reasoning.

Table 3

Provided Dataset	2
Population Size	*50*
Number of Generations	*200*
Number of Iterations	*3*
Mutation Rate	0.2
Mutation Step	0.2
Crossover	False
Gene Minimum	-1
Gene Maximum	1
Data Splitting Ratio	80/20
Number of Hidden Nodes	3

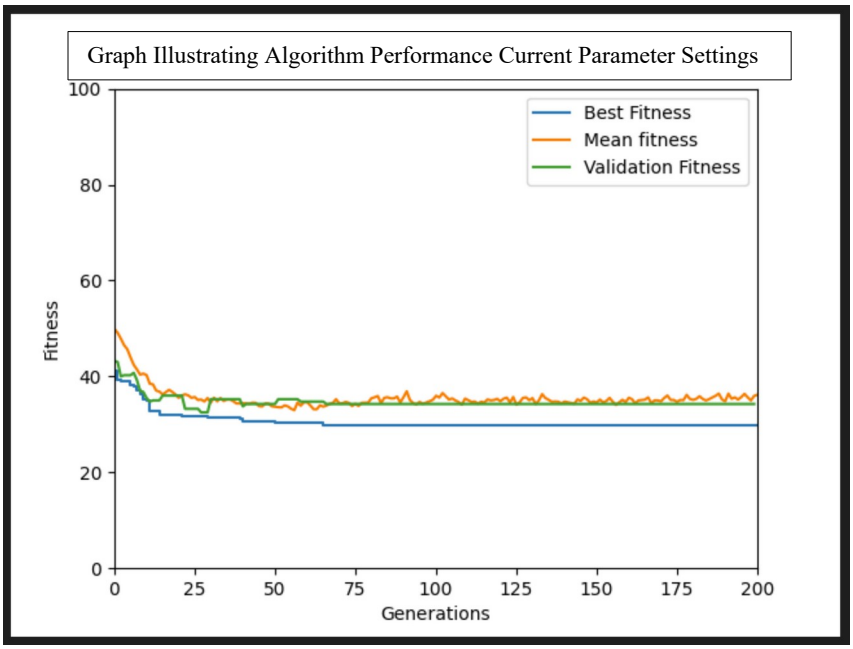


Figure 4

Running the algorithm with these encodings produced a mean lowest fitness of *29.33%.*

The Mutation Operator

Rate and Step Size of Mutation

The next thing to explore was the rate and step size of the mutation operator. In order to traverse the vast combination of settings for the mutation rate and step size, a parameter sweep in the form of a grid-search was used. Parameter sweeping involves iteratively running an algorithm with varying parameter settings and comparing the results to determine the parameters' best settings and is useful when optimal parameters do not obviously present themselves. Such is the case for the mutation rate and step size. The initial values were 0.2 as a starting point, and similar values were explored.

Table 4

	MS 0.1	MS 0.15	MS 0.2	MS 0.25	MS 0.3
MR 0.1	33.0	28.0	30.0	28.0	27.67
MR 0.15	32.0	27.33	27.33	28.33	29.67
MR 0.2	31.0	27.33	26.0	26.67	28.0
MR 0.25	30.0	27.0	26.67	28.0	27.0
MR 0.3	29.0	29.0	29.0%	28.0	29.0

Table 5

	MS 0.18	MS 0.19	MS 0.2	MS 0.21	MS 0.22
MR 0.18	28.067	29.0	29.0	28.0	29.67
MR 0.19	29.0	27.67	29.0	28.67	28.33
MR 0.2	28.33	27.0	26.0	29.67	30.0
MR 0.21	29.875	29.0	28.0	27.67	28.33

MR 0.22	27.33	28.0	30.33	27.0	27.0

The parameter sweep results subverted an expectation that a smaller mutation rate, closer to 1 divided by the number of chromosomes [18], would be more suitable. This search shows that increasing the rate of mutation even by a small amount causes the candidate solutions to move across the problem space too frequently, and decreasing the rate causes them not to move around enough. It also shows that increasing the mutation step size causes the candidate solutions to move too far across the problem space at a time and decreasing the mutation step causes them to not move far enough. Therefore, the following parameters were optimal at this point in the experimentation process:

Table 6

Provided Dataset	2
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.2
Crossover	False
Gene Minimum	-1
Gene Maximum	1
Data Splitting Ratio	80/20
Number of Hidden Nodes	3

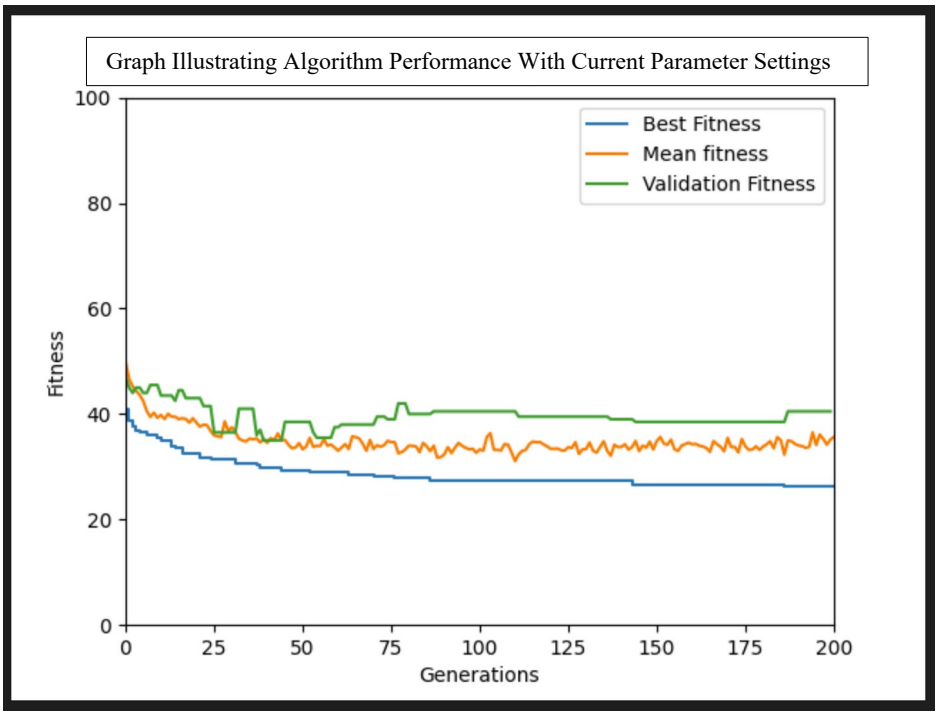


Figure 5

Running the algorithm with these encodings produced a mean lowest fitness of *26%.*

Minimum and Maximum Weight Values

The initial minimum and maximum weight values were -1 and 1. From this point onwards, the mutation step was multiplied by the weight maximum value to preserve the size of the step when the weight range was changed. The values were incremented by -1 and 1 respectively from (-1, 1) to (-10, 10), and it was found that the mean lowest fitness decreased until (-4, 4), after which point it gently increased. Figure 6 shows that beyond these values more of the data begins to be wrongly classified.

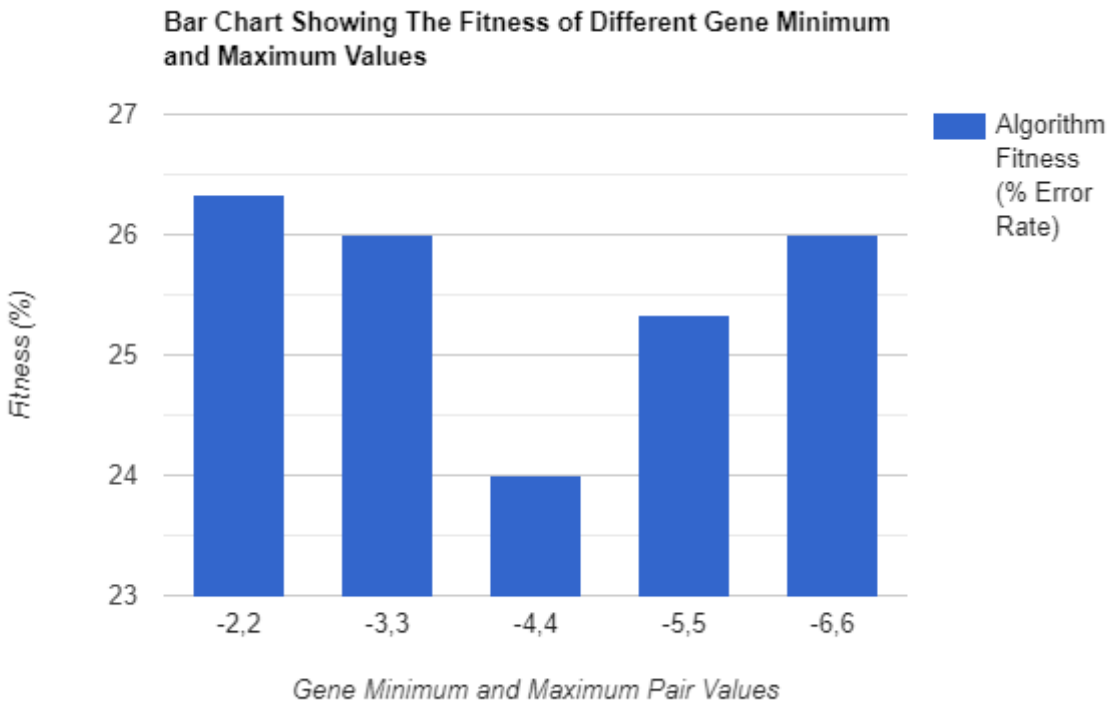


Figure 6

Table 7

Provided Dataset	2
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.8
Crossover	False
Gene Minimum	-4
Gene Maximum	4
Train/Test Split	80/20
Number of Hidden Nodes	3

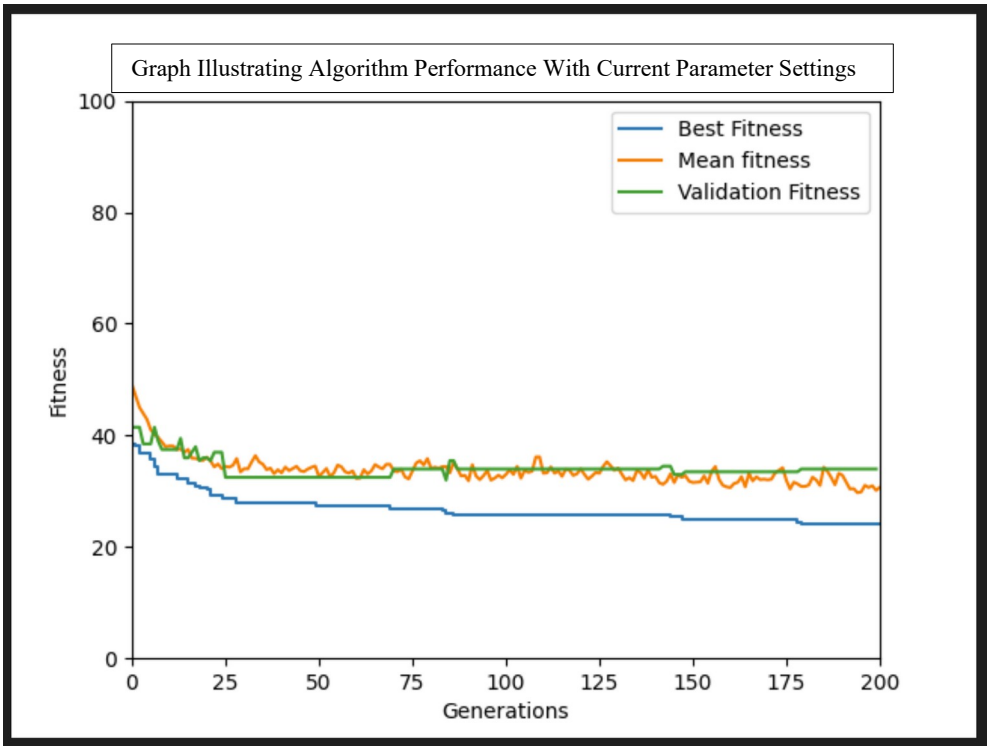


Figure 7

Running the algorithm with these encodings produced a mean lowest fitness of *24%*, but the lowest validation fitness was *32%*.

Data Splitting

The ratios used for data splitting vary, but an accepted starting ratio is 80:20. This ratio is inspired by the Pareto Principle or 80/20 rule, which explains a range of phenomena. [20] Up until this point, the 80/20 ratio was used. However, there were signs of overfitting appearing in the results ("the situation where a model fits very well to the current data but fails when predicting new samples"). Data splitting is done to avoid this. It is clear that throughout the current experiments, the algorithm fails to converge as far on the validation data as it does on the training data. The model is not useful if it cannot correctly classify new data. The aim of this experiment was to reduce overfitting, so the best results are those where there is little difference between the lowest fitness on the training and validation set. The next split tested was 67/33.

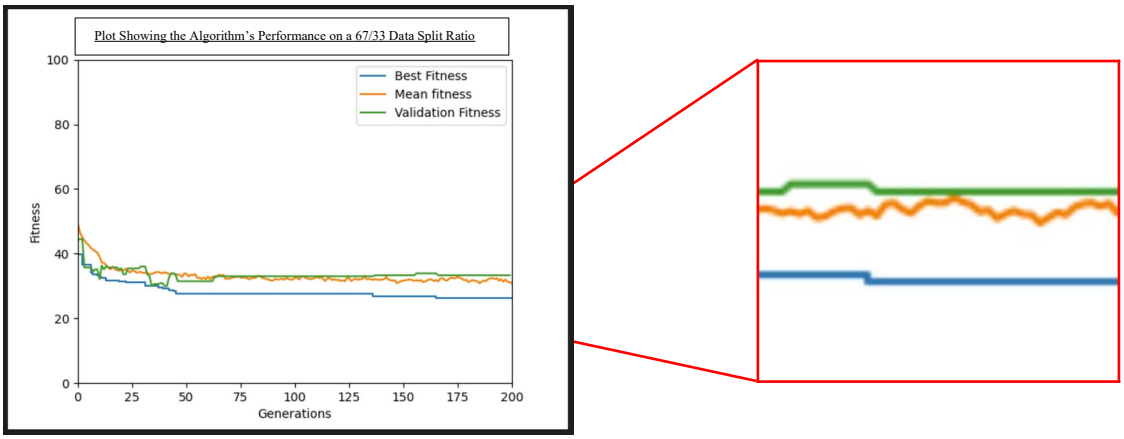


Figure 8

Running the algorithm with these encodings produced a mean lowest fitness of *25.33%*, but the lowest validation fitness was *30*%.

At this setting, the similarity in convergence is closer. With a 67/33 split, the difference between the lowest fitnesses found on the training and testing sets was 4.33, compared to 8.25 with an 80/20 split. To investigate whether a further increase in size of the test set, the next experiment tested a 50/50 split.

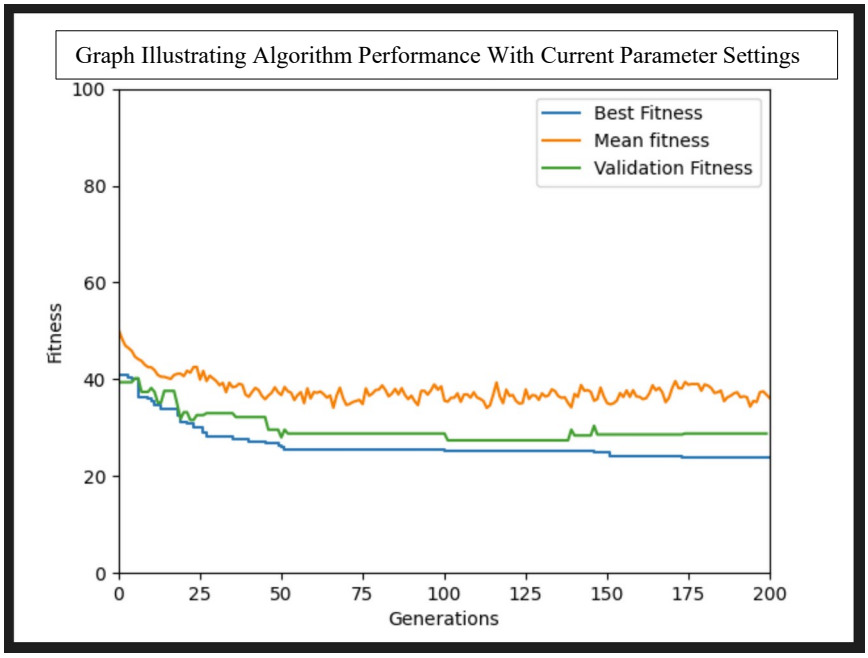


Figure 9

Running the algorithm with these encodings produced a mean lowest fitness of *23.0%*, but the lowest validation fitness was *32*%.

At this setting, the similarity in convergence is more distant, and the effects of overfitting are clear on the plot. The difference at this split ratio was 9, suggesting that using only 50% of the data on training is not enough to accurately train the model. The next experiment ran the algorithm with a 70/30 split to see if any closer convergence in the results for training and validation could be reached.

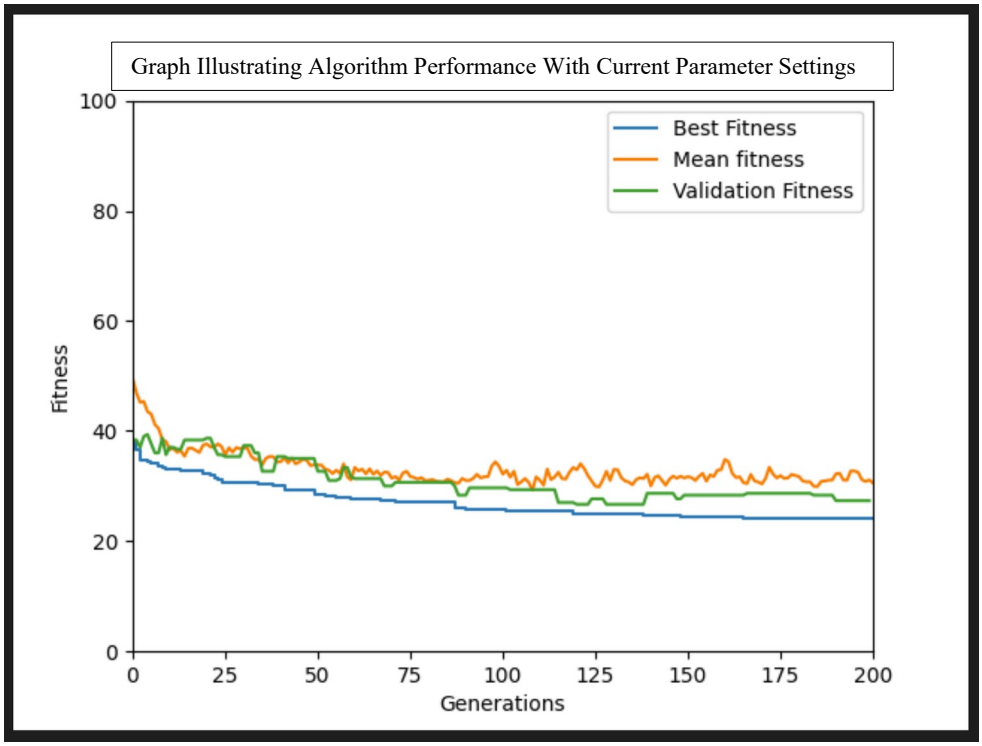


Figure 10

Running the algorithm with these encodings produced a mean lowest fitness of *24%,* and the lowest validation fitness was *29*%.

The difference at the ratio 70/30 was 5. The mean lowest fitness for the ratio 67/33 was higher, but the lowest fitnesses of the training and validation data sets were closer together. For the ongoing experimentation, a data splitting ratio of 67/33 was used because the purpose of this experiment was to find a data splitting ratio that brought the training and validation results closer together. For future experiments minimising fitness, it will be known that the ANN created is classifying accurately.

The data splitting in this algorithm was performed solely with the scikit-learn train_test_split function. This model randomly splits an array into train and test subsets. I use this function to split an array of 'TestDatum' objects (objects created from the provided datasets, containing a list of input values and a classification value).

The parameter settings at this point in experimentation are:

Table 8

Provided Dataset	2
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.8
Crossover	False
Gene Minimum	-4

Gene Maximum	4
Train/Test Split	67/33
Number of Hidden Nodes	3

The Number of Hidden Nodes

James Heaton, the author of the book An Introduction to Neural Networks for Java, Second Edition, puts forward three rules-of-thumb for the number of hidden nodes in an ANN: "between the size of the input layer and the size of the output layer", "2/3 the size of the input layer, plus the size of the output layer" and "less than twice the size of the input layer". [20] Up until this point in the process, the number of hidden nodes was 3.

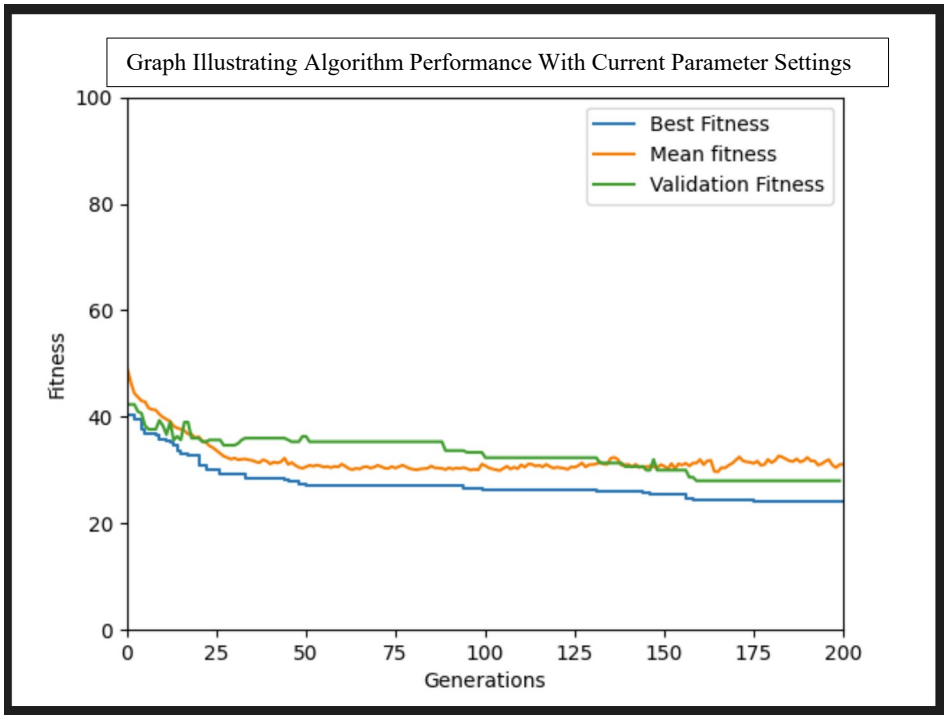
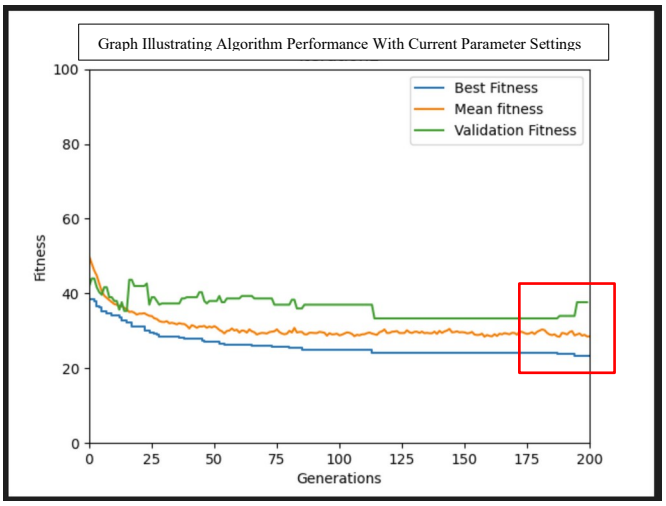


Figure 11

Running the algorithm with these encodings produced a mean lowest fitness of *24.33%,* and the lowest validation fitness was *28*%.

Next, 12 hidden nodes were tested (less than twice the size of the input layer – according to Heaton's third suggestion).



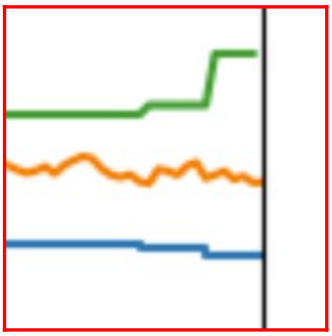


Figure 12

Running the algorithm with these encodings produced a mean lowest fitness of *22.67%,* and the lowest validation fitness was *30*%.

Overfitting can be observed in figure 12, suggesting the ANN is classifying too closely to the training data, less able to classify as on the testing data set. Next, 2 hidden nodes were tested (between the size of the input layer and the size of the output layer – according to Heaton's first suggestion).

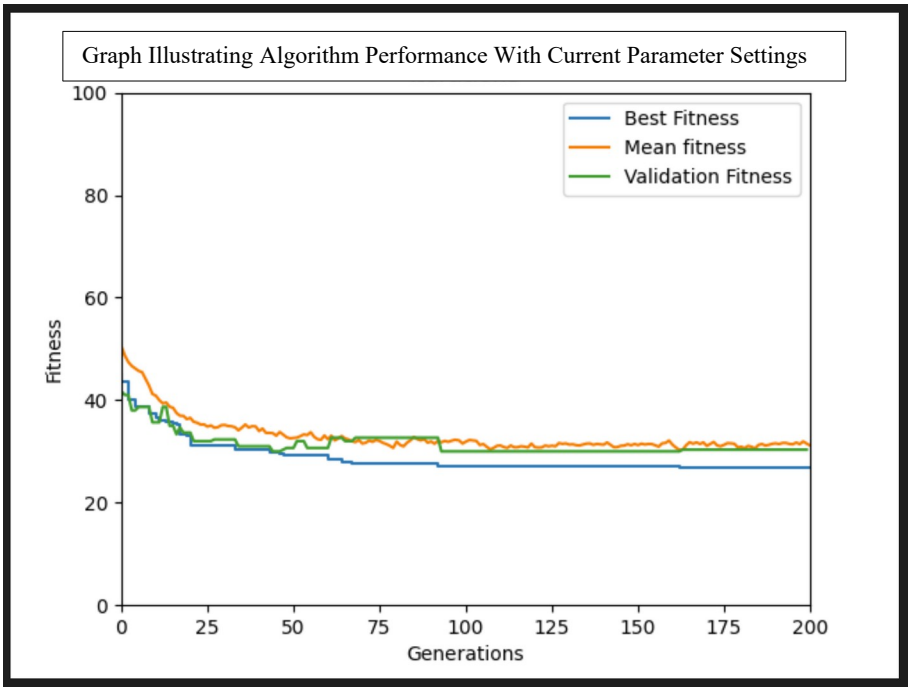


Figure 13

Running the algorithm with these encodings produced a mean lowest fitness of *27.33%,* and the lowest validation fitness was *30*%.

This saw a big reduction in the difference between the training and validation sets. The convergence for both sets were also much closer together. Next, 6 hidden nodes were tested (again, between the size of the input layer and the size of the output layer – according to Heaton's first suggestion, but higher this time).

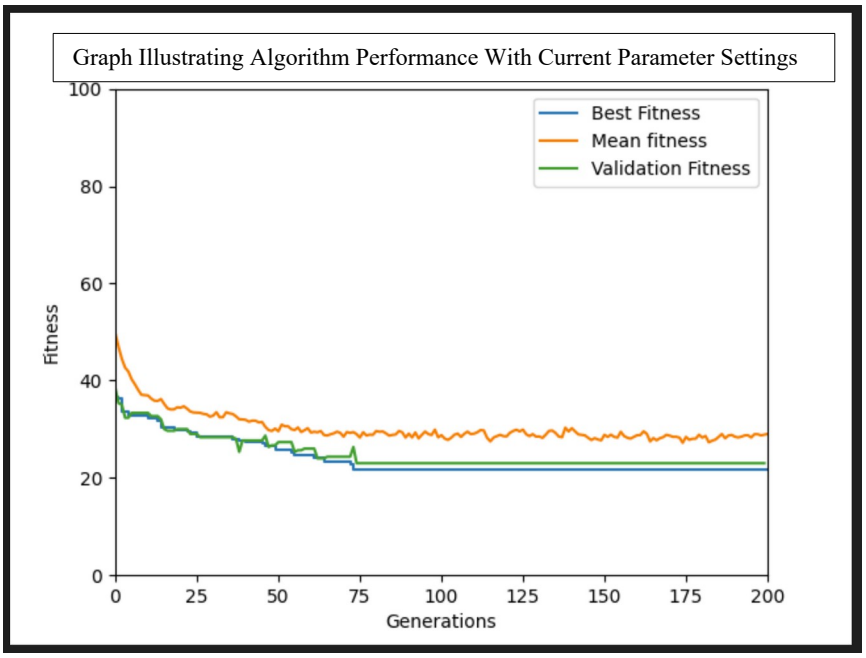


Figure 14

Running the algorithm with these encodings produced a mean lowest fitness of *24.67%,* and the lowest validation fitness was *28*%.

6 hidden nodes produced the best result. The mean lowest fitness was only 0.34% higher than that with 8 hidden nodes, but the difference between the lowest fitness of the training and testing data set was 3.33. The convergence of the algorithm on the training and validation sets are nearly the same. Running the algorithm with these parameter settings creates an effective binary classification model:

Table 9

Provided Dataset	2
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.8
Crossover	False
Gene Minimum	-4
Gene Maximum	4
Train/Test Split	67/33
Number of Hidden Nodes	6

Other Methods of Mutation

There are other methods of mutation to use in neuroevolution. One method sees the weight value multiplied instead of added to. The following tests use a mutation multiplication range of (1,2) and (negative mutation rate, mutation rate).

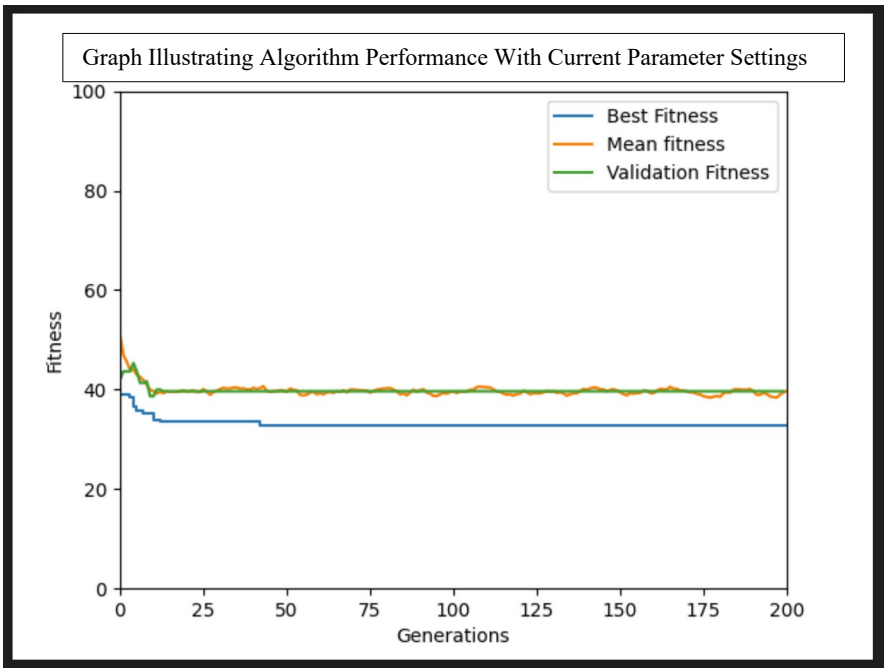


Figure 15

Running the algorithm with these encodings produced a mean lowest fitness of *31%,* and the lowest validation fitness was *39*%.

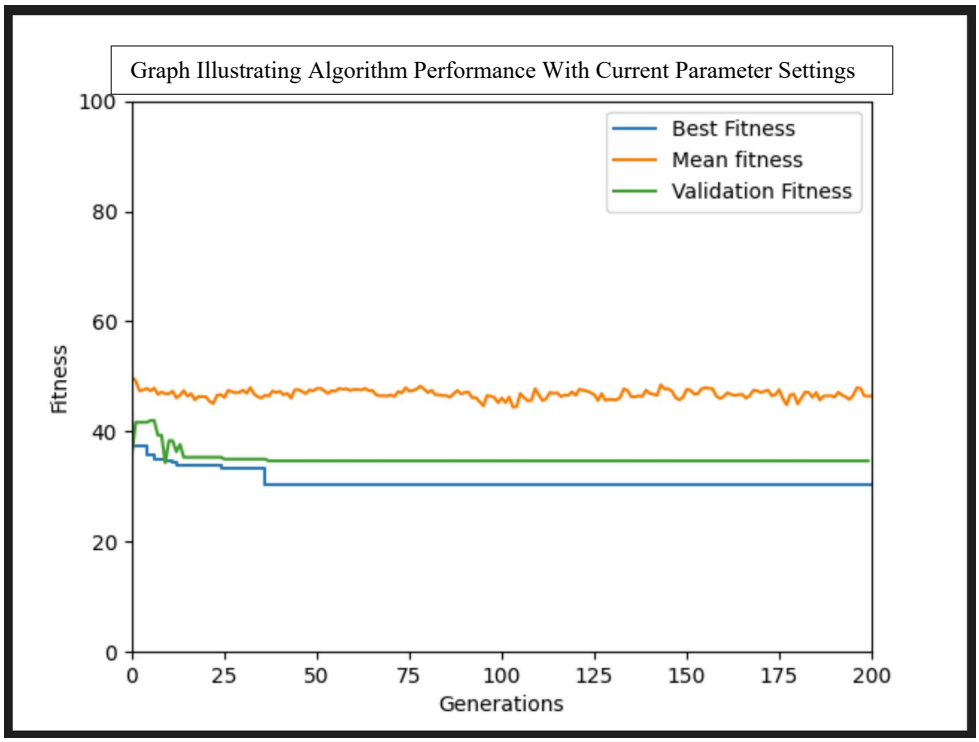


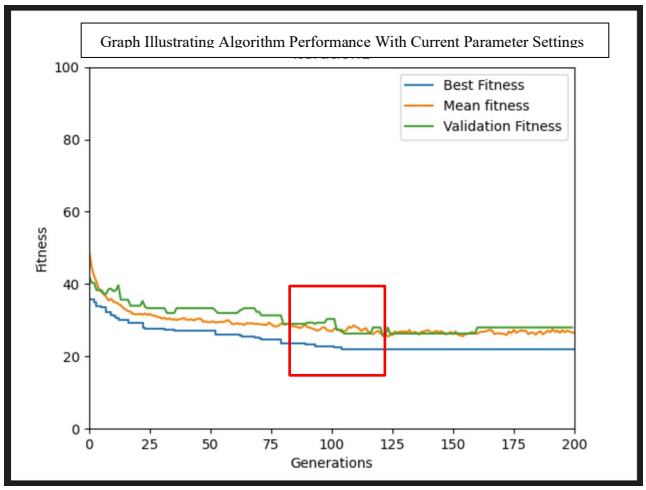
Figure 16

Running the algorithm with these encodings produced a mean lowest fitness of *30.33%,* and the lowest validation fitness was *33%.*

These tests show that this method of mutation is far less appropriate for seeking the best fitness. Other methods, not tested in this experiment, include replacing some weights or biases with new values, negating some weights or biases, and swapping some weights with other weights.

The Crossover Operator

The usual approach to the crossover operator fails when evolving the weights of ANNs due to the competing conventions problem, it causes the resulting weight sets to miss out information. [21] Thus, there are two experimental avenues that present themselves with respect to crossover: modifying it or omitting it completely. All the experiments up to this section inherently omit crossover. It can be seen from the results that evolution is still effective with only the mutation and selection operators. The figure shows a test using crossover:



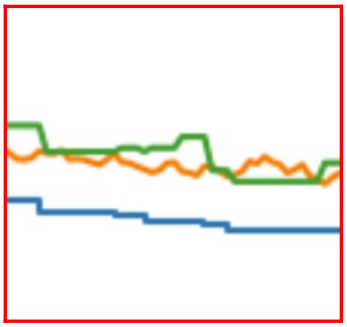


Figure 17

Running the algorithm with these encodings produced a mean lowest fitness of *21.67%,* but the lowest validation fitness was *30%.*

The way in which ANN nodes are arranged is important for the output value. Single-point crossover causes some of the correct settings in an ANN's weights to be replaced with completely different values, potentially removing important weights, and leading to solutions that are much more changeable throughout the generations, as can be seen when comparing the graph with crossover to that without. Also worth noting is that there are alternative approaches to applying crossover to neuroevolution algorithms such as CoSyNE, SANE, ESP and NEAT.

The Selection Operator

All of the experiments up until this section have been conducted using tournament selection, which randomly selects two individuals and discards that with the lowest fitness, replacing it with a copy of the other. It repeats this until all members of the population have been evaluated, eliminating unhelpful individuals from the population. Other methods of selection include roulette wheel selection (calculating the probability of an individual being selected by its fitness), and stochastic universal sampling, which uses a random value to choose each required solution at evenly spaced intervals based on their fitness.

Using The Final Dataset

The following parameter settings have been proven to create an optimal ANN for dataset 2. Continued experimentation took place using the largest dataset.

Table 10

Provided Dataset	3
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.8
Crossover	False

Gene Minimum	-4
Gene Maximum	4
Train/Test Split	67/33
Number of Hidden Nodes	6

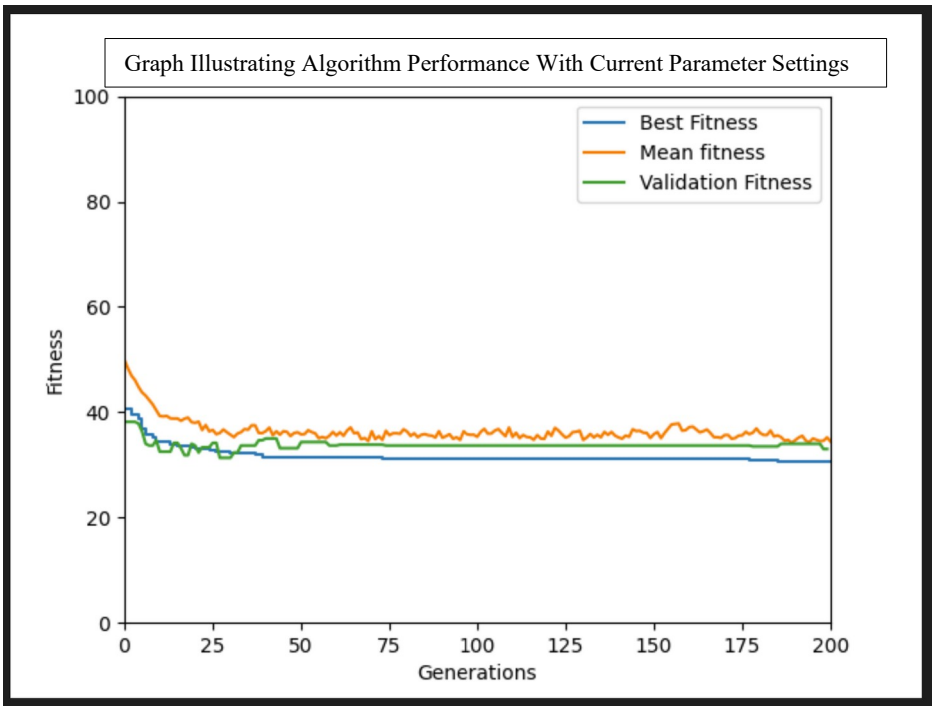


Figure 18

Running the algorithm with these encodings produced a mean lowest fitness of *29.33%,* but the lowest validation fitness was *30%.*

The number of hidden nodes was then adjusted to the best proportion as for dataset 2, because of the increase in number of input nodes (8 hidden nodes for 10 input nodes).

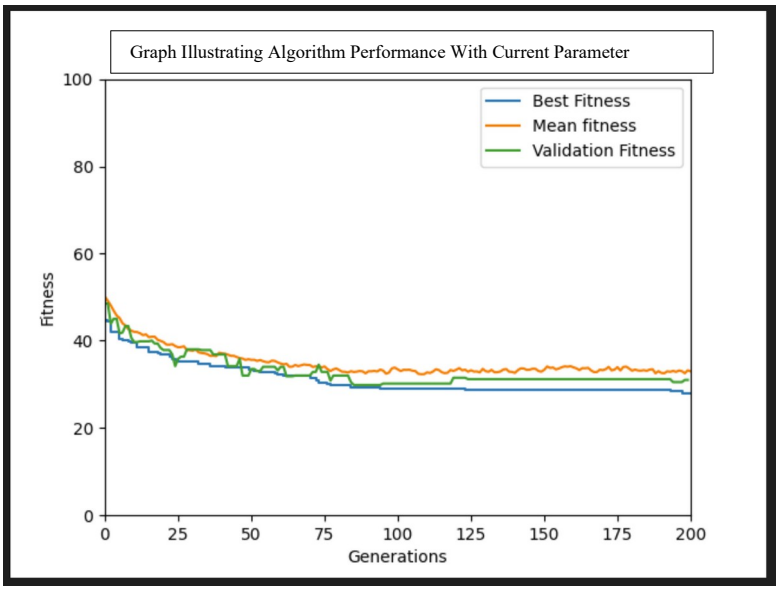


Figure 19

Running the algorithm with these encodings produced a mean lowest fitness of *25.93%,* and the lowest validation fitness was *27.0%.*

Finally, the better-performing population/generation ratio from the initial experimentation was implemented. However, subverting expectations, this produced a mean lowest fitness of 23.33%, worse than the same settings with a 67/33 population/generation ratio. Therefore, the former ratio was reinstated, producing the final parameter settings, outlined below.

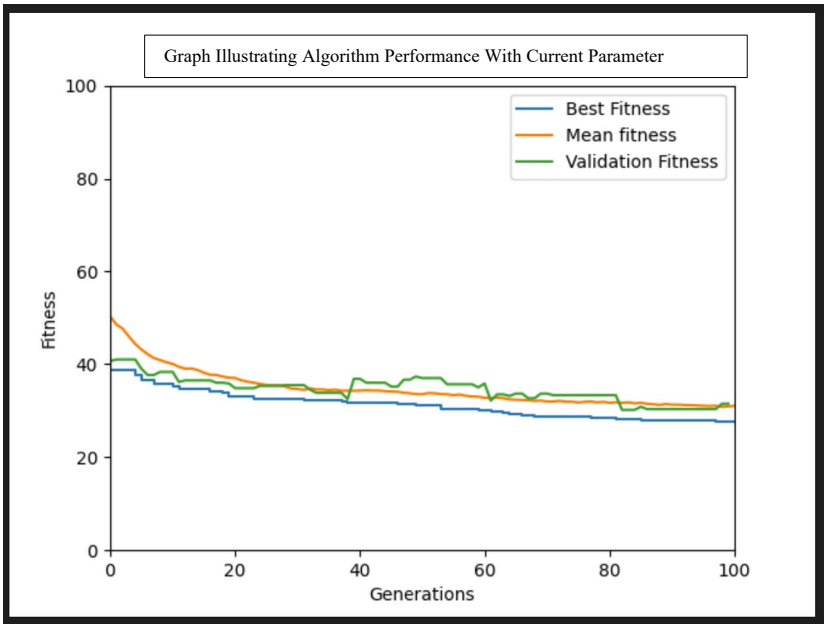


Figure 20

Final parameter settings:

Table 11

Provided Dataset	3
Population Size	50
Number of Generations	200
Number of Iterations	3
Mutation Rate	0.2
Mutation Step	0.8
Crossover	False
Gene Minimum	-4
Gene Maximum	4
Train/Test Split	67/33
Number of Hidden Nodes	8

Comparison With Weka

Table 12

Random Forest Classifier	23.9861 %.
Random Tree Classifier	19.0953 %
Decision stump	93.602 %
Gaussian Processes	86.6664 %
Linear Regression	81.1499 %
Multilayer Perceptron	75.5547 %
Simple linear regression	94.8276 %
SMOreg	78.235 %
Lazy IBK	41.8898 %
Lazy KStar	44.6066 %
Lazy LWL	44.6066 %

4 CONCLUSIONS

Because the GA finds the best solution by searching, evolving the weights for the ANN, the learning process has been conducted using search. This report reviews two optimisation algorithms inspired by nature, Cuckoo Search is an effective algorithm, but makes little improvement on performance when compared with other evolutionary algorithms, it is also criticised for unoriginality. The Firefly algorithm was found to perform better than other evolutionary algorithms, especially with noisy data. The algorithm tested in this report is outlined, and the effects of modifying parameters is illustrated in a number of ways. It can be observed that the adjustment of these parameters has implications on the performance of the entire algorithm, and insight as to the causes of this has been outlined. It is clear that the adjustment of these parameters is analogous to a search process, where each parameter change gets the algorithm closer to or more distant from the optimal solution.

Criticism of Methods

One criticism to be made in retrospect is the order in which these experiments were conducted. Although they

followed a logical flow, the order in which they were conducted could have been improved. Another issue with this report is that the starting values for the evolutionary algorithm were arbitrary. There was no logical reasoning behind the parameter settings at the beginning of the experimentation process. 10 iterations would have been better than 3, but for the number of experiments, it would have taken too long. Lastly, more parameters could have been swept for mutation, potentially making the mutation operator more effective.

Potential Future Experiments

- Larger reflections of the population/generation ratio
- More parameters for mutation
- More iterations of each experiment to make the results stochastic/more representative.
- Measure the convergence of each algorithm setting.
- Calculate the mutation step using normal distribution.
- Finding the Minimal Network for An Accurate Model
- Use the EA for hyperparameter setting of deep nets

REFERENCES

1. Yang, X.S. and Deb, S. (2010) Engineering Optimisation by Cuckoo Search. *Int. J. Mathematical Modelling and Numerical Optimisation*. 1 (4), pp. 330-343.

2. Ghosh, T., Das, S., Barman, S. and Goswami, R. (2016) A Comparison Between Genetic Algorithm and Cuckoo Search Algorithm to Minimize the Makespan For Grid Job Scheduling. *Advances in Computational Intelligence*. 509, pp. 141-147

3. Klempka, R. and Filipowicz, B. (2017) Comparison of Using the Genetic Algorithm and Cuckoo Search For Multicriteria Optimisation with Limitation. Turkish Journal of Electrical Engineering and Computer Sciences. 25 (2), pp. 1300-1310.

4. Guerrero, M., Castillo, O. and Garcia, M. (2014) Cuckoo Search Via Lévy Flights and a Comparison with Genetic Algorithms. *Studies in Computational Intelligence*. 574, pp. 91-103.

5. Valian, E., Mohanna, S. and Tavakoli, S. (2011) Improved Cuckoo Search Algorithm For Feed Forward Neural Network Training. *International Journal of Artificial Intelligence & Applications*. 2 (3), pp. 36-43.

6. Abdel-baset, M. and Al-mishnanah, I.M.H. (2016) Cuckoo Search and Genetic Algorithm Hybrid Schemes For Optimization Problems. *Applied Mathematics & Information Sciences*. 10 (3), pp. 1185-1192.

7. Camacho Villalon, C.L., Dorigo, M., Stutzle, T. and , (2021) An Analysis of Why Cuckoo Search Does Not Bring Any Novel Ideas to Optimization. *Iridia – Technical Report Series*. 6, pp. 1-7.

8. Wang, G. (2018) A Comparative Study of Cuckoo Algorithm and Ant Colony Algorithm in Optimal Path Problems. *Matec Web of Conferences*. 232, pp. 1-6.

9. Xin-she, Y. (2014) *Nature-inspired Optimization Algorithms*. London: O'Reilly Online Learning.

10. Lunardi, W.T. and Voos, H. (2018) Comparative Study of Genetic and Discrete Firefly Algorithm For Combinatorial Optimization. *Symposium on Applied Computing*.

11. Zhou, G.D., Ting-hua, Y.I., Zhuang, H. and Li, H.N. (2015) A Comparative Study of Genetic and Firefly Algorithms For Sensor Placement in Structural Health Monitoring. *Distributed Sensor Networks For Health Monitoring of Civil Infrastructures*., pp. 1-10.

12. Attia, K.A.M., Nassar, M.M.I., El-zeiny, M.B. and Serag, A. (2017) Firefly Algorithm Versus Genetic Algorithm as Powerful Variable Selection Tools and Their Effect on Different Multivariate Calibration Models in Spectroscopy: A Comparative Study. *Spectrochim Acta a Mol Biomol Spectrosc*., pp. 117-123.

13. Jaafaria, A., Termeh, S.V.R. and Bui, D.T. (2019) Genetic and Firefly Metaheuristic Algorithms For an Optimized Neuro-fuzzy Prediction Modeling of Wildfire Probability. *Journal of Environmental Management*. 243, pp. 358-369.

14. Lohrer, M.F. (2013) A Comparison Between the Firefly Algorithm and Particle Swarm Optimization. *Computer Engineering, Applied Mathematics*., p. 43.

15. D'addona, D.M. (2019) Neural Network. *Cirp Encyclopedia of Production Engineering*., pp. 911-918.

16. Liu, Q. and Wu, Y. (2012) Supervised Learning. *Encyclopedia of the Sciences of Learning*., p. 3243–3245.

17. Vrajitoru, D. (2000) Large Population Or Many Generations For Genetic Algorithms. *Implications in Information Retrieval*. 50, pp. 199-222.

18. Ochoa, G. (2002) Setting the Mutation Rate: Scope and Limitations of the 1/l Heuristic. *Proceedings of the Genetic and Evolutionary Computation Conference*.

19. Roshan, J.V. (2022) Optimal Ratio For Data Splitting. *Statistical Analysis and Data Mining*. 15 (4), pp. 531-538.

20. Heaton, J. (2008) *Introduction to Neural Networks For Java*. 2nd ed. St. Louis: Heaton Research, Inc.

21. Uriot, T. and Izzo, D. (2020) Safe Crossover of Neural Networks Through Neuron Alignment. *Neural and Evolutionary Computing*. 3, pp. 435-443.

Source code as an appendix

```
import copy
import math
import random
from operator import attrgetter
import matplotlib.pyplot as plt
from datetime import datetime
import os
from sklearn.model_selection import train_test_split
import time

experimentTitle = "..."

dataSource = "data/12102638.txt"
HidNODES = 8 # don't forget to modify when changing datasets
P = 50  # stick with these (20/80)
NUMBEROFGENERATIONS = 200
NUMBEROFITERATIONS = 3
GENEMIN = -4
GENEMAX = 4
MUTATIONRATE = 0.2
MUTATIONSTEP = 0.8
crossover = False

bestFitnesses = []
meanFitnesses = []
validationFitnesses = []
generations = []
bestFitnessesOverAllGens = []
totalMeanCalculationList = []

def setNumberOfInputNodes():
```

```
    dataFile = open(dataSource, "r")
    fileLine = dataFile.readline()
    lineList = fileLine.split()
    nodeCount = 0
    for i in range(0, len(lineList) - 1):
        nodeCount = nodeCount + 1
        if (fileLine[i] == 1) or (fileLine[i] == 0):
            break
    return nodeCount

InpNODES = setNumberOfInputNodes()
OutNODES = 1
N = ((InpNODES + 1) * HidNODES) + ((HidNODES + 1) * OutNODES)
HNodeOUT = [0] * ((HidNODES + 1) * OutNODES)  # stores the hidden node outputs
INodeOUT = [0] * ((InpNODES + 1) * HidNODES) # stores the output node outputs #
This is our result

# Object with gene and fitness parameters

class Individual:
    def __init__(self):
        self.chromosome = [0]*N # @ USE THIS NOTATION FOR CREATING THE 2D WEIGHT
ARRAYS
        self.fitness = 0

# Create an object containing a list of input values and one classification float
value

class TestDatum:
    def __init__(self):
        self.input = [0] * InpNODES # cant be inpNodes. data size
        self.expectedResult = 0

# Create an object containing a list of hidden weights, a list of output weights and
a value to track error

class Network:
    def __init__(self):
        # hweights is a 2D array with HidNODES rows and InpNODES + 1 columns
        self.hweights = [[0 for i in range(InpNODES + 1)] for j in range (HidNODES)]
        # oweights is a 2D array with OutNODES rows and HidNODES+1 columns
        self.oweights = [[0 for i in range(HidNODES + 1)] for j in range (InpNODES)]
        # this is the error resulting from running the network with the given
weights
```

```
        self.error = 0  # float

# Perform the sigmoid function

def sigmoid(node):
    if node >= 0:
        z = math.exp(-node)
        sig = 1 / (1 + z)
        return sig
    else:
        z = math.exp(node)
        sig = z / (1 + z)
        return sig

# Function that reads data from an input file and stores it in an object, returning
a list of those objects

def extractData():
    file = open(dataSource, "r")
    lines = file.readlines()
    listOfTestData = []

    for i in range(0, len(lines)): # For each line in the data file
        lines[i] = lines[i].replace("\n", "").split(' ')
        object = TestDatum()
        for j in range(0, InpNODES):
            object.input[j] = float(lines[i][j])
        object.expectedResult = float(lines[i][InpNODES])
        listOfTestData.append(object)

    return listOfTestData

# Assign individual gene values to a neural network's weights

def populateNeuralNetwork(individual):
    network = Network()
    count = 0
    for i in range(HidNODES):
        for j in range(InpNODES + 1):
            network.hweights[i][j] = individual.chromosome[count]
            count += 1
    for i in range(OutNODES):
        for j in range(HidNODES + 1):
            network.oweights[i][j] = individual.chromosome[count]
```

```
            count += 1
    return network

# Use evaluate the fitness of an individual using a neural network's classification
error value as the fitness value

def calculateFitnessWithNeuralNetwork(individual, dataSet):
    network = populateNeuralNetwork(individual)

    for t in range(0, len(dataSet)):  # Iterate over each object extracted from the
input data
        for currentHiddenNode in range(0, HidNODES):
            HNodeOUT[currentHiddenNode] = 0
            for currentInputNode in range(0, InpNODES):
                HNodeOUT[currentHiddenNode] += (network.hweights[currentHiddenNode]
[currentInputNode] * dataSet[t].input[currentInputNode])
            HNodeOUT[currentHiddenNode] += network.hweights[currentHiddenNode]
[InpNODES]  # Adds Bias to sum
            HNodeOUT[currentHiddenNode] = sigmoid(HNodeOUT[currentHiddenNode])

        for currentOutputNode in range(0, OutNODES):
            INodeOUT[currentOutputNode] = 0
            for currentHiddenNode in range(0, HidNODES):
                INodeOUT[currentOutputNode] += (network.oweights[currentOutputNode]
[currentHiddenNode] * HNodeOUT[currentHiddenNode])
            INodeOUT[currentOutputNode] += network.oweights[currentOutputNode]
[HidNODES] # Bias
            INodeOUT[currentOutputNode] = sigmoid(INodeOUT[currentOutputNode])

        if dataSet[t].expectedResult == 1.0 and INodeOUT[0] < 0.5:
            network.error += 1.0  # Class 1 if output > 0.5
        if dataSet[t].expectedResult == 0.0 and INodeOUT[0] >= 0.5:
            network.error += 1.0 # Does this mean one error each go?

    return 100 * (network.error/len(dataSet)) # Added to return percentage

# Evaluate fitness of each individual in a population using the fitness function.

def evaluatePopulation(population, dataSet):
    for individual in population:
        individual.fitness = calculateFitnessWithNeuralNetwork(individual, dataSet)

# Evaluate the fitness of each individual in a population, and then return the sum
of those fitnesses
```

```
def calculateTotalFitness(population, dataSet):
    evaluatePopulation(population, dataSet)
    populationFitness = 0
    for individual in population:
        populationFitness += individual.fitness
    return populationFitness

# Return the individual in a population with the lowest fitness using the min
function on the attrgetter operator.

def returnBestIndividual(population):
    return min(population, key=attrgetter('fitness'))

# Find the weakest individual in a population and return its index

def returnIndexOfWorstIndividual(population):
    return population.index(max(population, key=attrgetter('fitness')))

# Return the fitness of the individual in a population with the lowest fitness using
the min function on the attrgetter operator.

def returnMinimumFitness(population):
    return returnBestIndividual(population).fitness

# Return the mean fitness of individuals in a population.

def returnMeanFitness(population, dataSet):
    totalFitness = calculateTotalFitness(population, dataSet)
    meanFitnesses = totalFitness/len(population)
    return meanFitnesses

# Instantiate a population by adding a gene populated with binary values to an
individual object, and adding it to a list

def initialisePopulation():
    population = []
    for x in range(0, P):
        tempChromosome = []
        for y in range(0, N):
            tempChromosome.append(random.uniform(GENEMIN, GENEMAX))
        newInd = Individual()
        newInd.chromosome = tempChromosome.copy() # gene should be named genome
        population.append(newInd)
```

```
    return population

# Tournament Selection. Create offspring population by choosing a pair of random
parent individuals, and adding the fittest of the two offspring created to a new
list. (Selection Code)

def performTournamentSelection(population):
    offspring = []
    for i in range(0, P):
        # Choose two parents from the population, and create an offspring from each.
        parent1 = random.randint(0, P-1)
        off1 = copy.deepcopy(population[parent1])
        parent2 = random.randint(0, P-1)
        off2 = copy.deepcopy(population[parent2])
        # Add the least fit offspring to a new list.
        if off1.fitness < off2.fitness:
            offspring.append(off1)
        else:
            offspring.append(off2)
    # print("Most fit - ", calculateTotalFitness(offspring))
    return offspring

# Roulette wheel selection

# For each pair of objects in offspring, select a crossover point and perform
crossover, replacing the original individuals. Now depending on crossover
probability.

def performCrossover(population, dataSet):
    toff1 = Individual()
    toff2 = Individual()
    temp = Individual()
    for i in range(0, P, 2):
        toff1 = copy.deepcopy(population[i])
        toff2 = copy.deepcopy(population[i+1])
        temp = copy.deepcopy(population[i])
        crosspoint = random.randint(1, N)
        # At the crossover point, perform crossover on the pair and replace the
originals.
        for j in range(crosspoint, N):
            toff1.chromosome[j] = toff2.chromosome[j]
            toff2.chromosome[j] = temp.chromosome[j]
        if (calculateFitnessWithNeuralNetwork(toff1, dataSet) <
population[i].fitness):
```

```
            population[i] = copy.deepcopy(toff1)
        if (calculateFitnessWithNeuralNetwork(toff2, dataSet) <
population[i+1].fitness):
            population[i] = copy.deepcopy(toff2)
    # print("Crossover - ", calculateTotalFitness(population))
    return population

# Flip values in individual's gene strings depending on the mutation probability,
and replace it.

def performMutation(population):
    for i in range(0, P):
        newInd = Individual()
        newInd.chromosome = []
        for j in range(0, N):
            gene = population[i].chromosome[j]
            mutationProbability = random.random()
            if mutationProbability < MUTATIONRATE:
                alteration = random.uniform(-MUTATIONSTEP, MUTATIONSTEP)
                gene = gene + alteration
                if gene > GENEMAX:
                    gene = GENEMAX
                elif gene < GENEMIN:
                    gene = GENEMIN
            newInd.chromosome.append(gene)
        population[i] = copy.deepcopy(newInd)
    # print("Mutation - ", calculateTotalFitness(population))
    return population

# Take the weakest individual in a population and place it in the place of the
fittest individual in its offspring population

def performElitism(parentPopulation, offspringPopulation):
    weakestIndividualInParentPop = min(
        parentPopulation, key=attrgetter('fitness'))
    indexOfFittestIndividualInOffspringPop = returnIndexOfWorstIndividual(
        offspringPopulation)
    offspringPopulation[indexOfFittestIndividualInOffspringPop] =
weakestIndividualInParentPop

# Save mean and best values to lists for plotting purposes

def saveGenerationInfo(population, dataSet):
    meanFitnesses.append(returnMeanFitness(population, dataSet))
```

```
    bestFitnesses.append(returnMinimumFitness(population))

# Function containing all the code needed to create the matplotlib plot

def savePlot(iteration):
    bestFitnessLabel = "Best Fitness "
    meanFitnessLabel = "Mean fitness"
    validationFitnessLabel = "Validation Fitness"
    plt.step(bestFitnesses, '', label=bestFitnessLabel)
    plt.plot(meanFitnesses, '', label=meanFitnessLabel)
    plt.plot(validationFitnesses, '', label=validationFitnessLabel)
    plt.xlim(0, NUMBEROFGENERATIONS)
    plt.xlabel("Generations")
    plt.ylim(0, 100)
    plt.ylabel("Fitness")
    plt.legend()
    plt.title("Iteration" + str(iteration + 1))
    plt.savefig(path + "Iteration_" + str(iteration + 1) + "_" +
str(int(min(bestFitnesses))) + ".png")
    plt.clf()

# What the tin says

def createAlgorithmInfo():
    return ("Experiment Title: " + experimentTitle + "\n"
        + "Input Nodes: " + str(InpNODES) + "\n"
        + "Input Node Outputs: " + str(len(INodeOUT)) + "\n"
        + "Hidden Nodes: " + str(HidNODES) + "\n"
        + "Hidden Node Outputs: " + str(len(HNodeOUT)) + "\n"
        + "Output Node: " + str(OutNODES) + "\n"
        + "Candidate Solutions (P): " + str(P) + "\n"
        + "Number of Generations: " + str(NUMBEROFGENERATIONS) + "\n"
        + "Number of Iterations: " + str(NUMBEROFITERATIONS) + "\n"
        + "Genome Length (N): " + str(N) + "\n"
        + "Gene Min: " + str(GENEMIN) + "\n"
        + "Gene Max: " + str(GENEMAX) + "\n"
        + "Mutation Rate: " + str(MUTATIONRATE) + "\n"
        + "Mutation Step: " + str(MUTATIONSTEP) + "\n"
        + "Crossover: " + str(crossover) + "\n"
        + "Lowest Fitness: " + str(min(bestFitnessesOverAllGens)) + "%\n"
        + "Mean Lowest Fitness: " + str(round(sum(totalMeanCalculationList) /
len(totalMeanCalculationList), 2)) + "%\n"
        + "Mean of Mean Fitnesses: " + str(round(sum(meanFitnesses) /
len(meanFitnesses), 2)) + "%\n"
```

```
        + "Lowest Validation Fitness: " + str(round(min(validationFitnesses))) + "%\
n"
        + "Elapsed time in seconds: " + str(round(time.time() - startTime, 2))) #
Turn into hh:mm:ss

# Save data from a run to a file

def saveAlgorithmInfo():
    f = open(path + "Incarnation" + datetime.now().strftime("%m-%d_%H_%M_%S") +
".txt","w+")
    f.write(createAlgorithmInfo())
    f.close()

# Print data from a run to a file

def printAlgorithmInfo():
    print(createAlgorithmInfo())

# Start recording duration

startTime = time.time()

# Choose where and how to save algorithm information

path = os.path.join("Plots/", "plot_" + datetime.now().strftime("%m-%d_%H_%M_%S")+
experimentTitle + "/")
os.mkdir(path)

# Extract and split data # I could be doing this inside the iteration loop
dataList = extractData()
trainingData, testingData = train_test_split(dataList, train_size=0.7)

for iteration in range(NUMBEROFITERATIONS): # Stochastic check

    bestFitnesses.clear()
    meanFitnesses.clear()
    validationFitnesses.clear()

    # TRAINING

    # Initialise a population of random individuals containing network weight
genomes
    population = initialisePopulation()
```

```
    # Try the neural network with these weights, and assign each individual a
fitness.

    evaluatePopulation(population, trainingData)
    saveGenerationInfo(population, trainingData)

    for gen in range(NUMBEROFGENERATIONS):
        # Select the best individuals in the population.
        offspring = performTournamentSelection(population)
        if crossover == True: performCrossover(offspring, trainingData)
        # Mutate the individuals
        performMutation(offspring)
        # run the network with the new weights
        evaluatePopulation(offspring, trainingData)
        # Replace the worst individual in this generation with the best from the
last one
        performElitism(population, offspring)

        # VALIDATION # Any reason why I can't keep this in this loop? It's using a
different data set # I should refactor this to be inside saveGenerationInfo
        bestIndividualInGeneration = returnBestIndividual(population)
        fitnessOfBestOnValidationData =
calculateFitnessWithNeuralNetwork(bestIndividualInGeneration, testingData)
        validationFitnesses.append(fitnessOfBestOnValidationData)

        population = copy.deepcopy(offspring)

        # Save information from this generation
        saveGenerationInfo(population, trainingData)
        print(min(bestFitnesses))
        bestFitnessesOverAllGens.append(min(bestFitnesses))

    totalMeanCalculationList.append(int(min(bestFitnesses)))
    savePlot(iteration)

saveAlgorithmInfo()
printAlgorithmInfo()
```