Neural Networks and Genetic Algorithms Coursework

UP939163
up939163@myport.ac.uk

UP940148
up940148@myport.ac.uk

May 26, 2022

Abstract

This report details the design, implementation, and results of a neural network for classification of the Iris dataset, as well as a genetic algorithm for the optimisation of the 2-dimensional *Shubert2* function, which have been implemented using MATLAB's *Deep Learning*, and *Global Optimization* toolboxes, respectively.

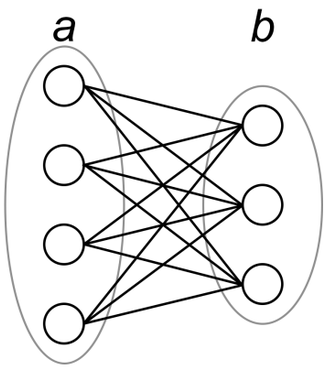
A feed-forward neural network was used in an attempt to accurately classify the flowers of the Iris dataset. This involved normalising the data so that every input fell within the same range, separating the data into different subsets for the training, validation, and testing of the network, and configuring the network's architecture, activation functions, training function, and stopping criteria to produce a network that can reliably classify sample data. The resulting network was able to accurately and reliably classify the data provided, with the only misclassifications occurring in the initial training phase. The network's stopping criteria were the only hindrance to its performance and so future work could be done to further test and improve these.

The implemented GA configures a number of options and parameters to maximise the efficiency of its minimisation. Factors such as its creation function, selection strategy, and reproductive operators were all altered to improve performance. Overall, the GA produced accurate results converging at the *Shubert2* function's global minimum on 3 out of 4 runs, with the run that converged on a local minimum still being very close to the global minimum. Future work to improve the GA would most prominently involve optimising the stopping criteria, as the GA converges on the global minimum relatively rapidly.

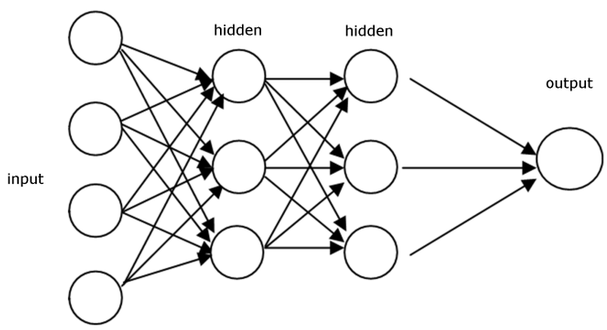
Chapter 1
Introduction

A neural network (NN) is a type of algorithm loosely based on the workings of the human brain. NNs are designed to recognise patterns and classify findings (Jordanov & Georgieva, 2009), sometimes also recognising patterns in order to react to its surroundings (SethBling, 2015). They take in some form of sensory data as an input, and run it through a series of layers consisting of perceptrons (neurons), before outputting some kind of result based on the input data. A classic example of a classification problem is using the MNIST handwritten digit dataset (LeCun et al., n.d.) to train a network to accurately classify handwritten digits 0 through 9. In the case of the MNIST digit classification, the network will have 784 inputs, and will classify the data into one of 10 potential outputs.

In a traditional feed-forward NN, every neuron in a layer is connected to every neuron in the next layer via a selection of weights, in the same way as a complete bipartite graph where each subsequent layer of the network acts as an independent set (Fig. 1.1). These weights tell the network how 'relevant' the information from the neuron in the previous layer is to the neuron in the current layer.



(a) Complete bipartite graph.



(b) Basic feed-forward neural network structure.
Source: "Feed-forward and feedback networks" (n.d.)

Figure 1.1: Comparison of a complete bipartite graph and the layers of a basic feed-forward neural network.

Feed-forward NNs are often trained through backpropagation. Backpropagation deploys a cost/error function which tells the network how far its solution was from the correct answer (the cost/error). Subsequently, every weight in the network is updated based on this calculated error. In traditional backpropagation, the amount that each weight affects the output is determined by implementing the chain rule to get the derivative of the cost, with respect to the current weight (Eq. 1.1).

$$\frac{\delta C}{\delta W} = \frac{\delta C}{\delta a} \bullet \frac{\delta a}{\delta W} \tag{1.1}$$

A genetic algorithm (GA) is a computational method most commonly used to solve optimisation and search problems through the application of the natural selection process. GAs create a population of chromosomes, each representing an individual solution to the proposed problem. Generations can then be iteratively created through modification of the population. This modification imitates the natural selection process by selecting pairs of chromosomes as parents for reproduction to then produce offspring comprised of a combination of the parent's solutions. More accurate or 'fit' chromosomes are more likely to be selected for reproduction, leading to the population growing exponentially fitter over successive generations and, ideally, the convergence of an optimal solution. There are a number of parameters that can be configured to optimise the GA's performance, such as the selection strategy, population size, and mutation method, which all affect its convergence to a global extremum.

An example application of a GA can be seen implemented by Fonseca et al. (2005), who utilised a GA to reduce the noise and vibration of vehicle transmission gears and, as a result of the strong correlation between acoustic gear noise and gear transmission error, improve transmission quality. Their GA took input data pertaining to the design parameters of two mating gears and calculated the transmission error (the deviation in rotation from its ideal position as the gear pair is rotated under constant torque), to hasten calculations and assist in producing optimal gear designs.

In this paper, a neural network will be used to classify plants into three classes using the Iris dataset and a GA will be used to minimise the *Shubert2* function, finding the values for x_1 and x_2 that produce the function's global minimum. All source code can be found in Appendix I.

Chapter 2
Neural Network

2.1 Dataset

For this report, the Iris dataset was chosen (Fisher, 1936). It is made up of 150 instances, evenly divided between three classes: Iris Setosa, Iris Versicolour, and Iris Virginica. Every instance contains five attributes, the first four of which are numeric, predictive attributes, and the fifth is the class that the instance belongs to. The four numeric attributes of each instance represent the plant's sepal length, sepal width, petal length, and petal width respectively in cm.

2.1.1 Pre-Processing

Before the input data could be passed to the NN, some pre-processing was required. Normalisation of the data was performed using min-max normalisation (Eq. 2.1), using the minimum and maximum values for each feature from the Iris dataset[1]. Normalisation will ensure the data is within an effective input range for the hidden layer's activation function, eliciting faster convergence. The activation function used will be the Hyperbolic Tangent function, which will be discussed in later sections. The function is asymptotic and inputs (x values) below -1 and beyond 1 produce outputs (y values) with marginal differences tending towards -1 and 1 respectively. Therefore the inputs have been normalised in the interval [-1, 1].

$$x_i' = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{2.1}$$

Standardisation of the inputs was also considered as an alternative to normalisation. It would be performed using equation 2.2 and would result in the data being re-scaled such that the mean equals 0 and the standard deviation equals 1. Hence, the values would generally sit in the interval [-1, 1]. However outliers would not sit within the interval and thus would have a lesser effect due to the aforementioned asymptotic nature of the activation function. It is also important to consider, the effect of the outliers during the calculation of the mean and standard deviation, which could affect the re-scaled values. Nonetheless it was decided that this method would not be used as it is most useful when applied to input ranges that have a Gaussian distribution. In this case, the input values of the Iris dataset do not follow a Gaussian distribution, with the exception of sepal length, as can be seen in appendix A.

$$z = \frac{x_i - \mu}{\sigma} \tag{2.2}$$

The order of the inputs was also randomised before they were passed to the network to attempt to prevent any training biases that could arise from the data potentially being grouped. A random order helps to ensure the NN is presented with input data that is largely distinct, this is especially important early in its training, to prevent premature convergence as the data is not randomised between epochs.

2.1.2 Subset Split

After the data had been normalised and randomised, it was split into different subsets for training, validation, and testing. These splits were configured using the `net.divideParam` options `trainRatio`, `valRatio`, and `testRatio` respectively. Splitting the data up into these subsets allows us to evaluate the performance of our network. The training data is the data that will be provided to the network to help it build up its classification ability, the validation data will be used during training to test the network against its first unseen data which allows us to see how well the model can make predictions as well as preventing the network from overfitting to the training data, and then finally the testing data serves to make sure that the network model can reliably make accurate predictions on new data.

Different subset splits were tested, both with and without validation data, the performance plots and confusion matrices of these models can be found in appendix D. In the two models without validation data (Figs. D.1 & D.2), we can see the network overfitting to the training data, this is evidenced by the performance plots showing the error on the training data decreasing whilst the error for the testing data increases. Once validation data was added to the mix (Figs. D.3 & D.4), we can see the network is no longer overfitting as drastically, and in fact the training is being halted when the network starts to overfit due to the maximum validation checks stopping criteria.

The most optimal split found was to separate the data set into training/validation/testing in the ratio 70/15/15. This seemed to give the most consistently promising results, as can be seen in appendix F.

2.2 Neural Network Architecture

The chosen network architecture consists of five neurons in the input layer, where four of those are for the predictive attributes of the dataset and the last is a bias, a single hidden layer consisting of five neurons, including a bias, and then a three-neuron output layer to represent the three possible classes (Fig. B.1).

[1] Choosing a minimum and maximum outside of the range of the dataset could be used to potentially improve generalisation with the ability of classifying unseen values outside of the dataset's range. However this may not be robust, as the network will not be trained on these unseen, out of range values, thus the NN will use only the minimum and maximum of the dataset it is trained upon.

The activation function used for the neurons in the hidden layer was the Hyperbolic Tangent function (Eq. 2.3). Originally, the Sigmoid function (Eq. 2.4) was chosen as it is a commonplace and well-established activation function used in hidden layers, however the Hyperbolic tangent function was chosen based on its ability to converge faster (mvs, 2019).

$$tanh(x) = \frac{sinh(x)}{cosh(x)} \tag{2.3}$$

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

For the output layer, the Softmax function was used (Eq. 2.5). The Softmax function was an ideal candidate because it uses Luce's choice axiom to normalise the output to a probability distribution of the output classes (Luce, 2008, Eq. 4)[2].

$$\sigma(\overrightarrow{Z})_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{Z_j}} \tag{2.5}$$

2.3 Learning and Training

2.3.1 Training function

After investigating the use of different training functions, scaled conjugate gradient backpropagation (SCG) was chosen as it seemed to provide the best results for our implementation. The conjugate gradient backpropagation function, with Polak-Ribiére's updates, (CGP) was also considered as it gave similar results, however SCG performed slightly faster which led to it being preferred for this classification problem. It should be noted that the SCG is regarded as a better training function for solving linearly separable classifications, and this is likely why SCG seems to yield flawless classification of the first target class, the *Setosa* class, as this is linearly separable from the other two classes (Fig. E.1). Despite SCG seemingly being better suited towards linearly separable classes and the Iris dataset containing two classifications that are not linearly seperable, after testing multiple other functions, it was still found that SCG yields the best results for the network.

2.3.2 Stopping Criteria

The stopping criteria has not been altered from the default criteria set by MATLAB. Whilst designing and testing the neural network, different stopping criteria were altered and tested, but no configurations proved to be more accurate or efficient than MATLAB's default configuration. The set stopping criteria can be seen in table 2.1. The stopping criteria most often met by the network is the `max_fail` parameter, which is the maximum number of iterations where the validation performance fails to decrease. Increasing this and preventing it from being met resulted in the network overfitting to the training data and no longer being capable of generalisation. Figure C.1 demonstrates this, displaying the performance plot of the NN when the maximum number of times the validation error fails to decrease is set to 100, showing the error function continuously decreasing for the training data, but increasing after approximately the first 12 epochs for the validation and testing data.

Criteria	Set Value
`epochs`	200
`min_grad`	0.000001
`max_fail`	6
`sigma`	0.00005
`lambda`	0.0000005

Table 2.1: Stopping Criteria Used by the Neural Network

2.4 Testing

The confusion matrix, shown in figure F.1d, shows that the network performs well overall, and the 'Test Confusion Matrix' shows that the network is capable of generalising to classify unseen data. The only misclassifications occurred with the training data, which is acceptable as this data is used to teach the network and thus mistakes are less significant. Additionally, this shows that the network isn't overfitting to the said training data.

Figure F.1a shows the plot of the NN's error function (cross entropy) output per epoch for each subset of the training data. It can be observed that after 6 epochs, the validation and testing subsets produce much lower error values than the training subset. This could indicate that the network has a greater ability to generalise as it has classified unseen data more accurately.

It is also worth noting that Figure F.1a and F.1b display a small spike in the error value and backpropagation gradient respectively at approximately 5 epochs. This could indicate the NN encountering a local extrema, which is then subsequently overcome, with a harsh drop in the error value and backpropagation gradient.

[2]Equation 4 from Luce (2008) appears to be exactly the same as the Softmax function, however it uses function notation ($v(x)$) rather than exponential function notation (e^{Z_i}).

2.5 Conclusion

In conclusion, the NN configured in this paper performs well and can generalise to categorise the Iris plants as required. As can be seen in figure F.1d, only two misclassifications were made by the network during training. It is worth noting that whilst all three forms of the Iris plant are poisonous, the Iris setosa can be made safe for human consumption via cooking and its *rhizome* can be used for medicinal purposes (Kelso & Society, 2011, p. 101). For this reason the sensitivity of the Iris setosa's classification is important, whilst its specificity is less so. The misclassifications show that neither the Iris versicolor (Class 2) or Iris virginica (Class 3) plants have been falsely classified as an Iris setosa (Class 1), only the Iris versicolor has once being classified as the Iris virginica, and vice versa.

Nonetheless, the NN could be improved, most notably by re-configuring the stopping criteria. Whilst creating the network, no alternatives could be found that offered greater performance than the default stopping criteria set by MATLAB. An improvement in performance could be made here by better tailoring the stopping criteria for the network's purpose.

Chapter 3
Genetic Algorithm

3.1 Chosen Function

The 2-dimensional *Shubert2* function has been chosen for optimisation using a GA. This function has a single global minimum where $f_2(x_1, x_2) = -186.730909$. The *Shubert2* function can be seen defined in equation 3.1.

$$\begin{aligned} f_2(x_1, x_2) =& \prod_{i=1}^{2} \left[\sum_{j=1}^{5} j \cos((j+1)x_i + j) \right] + \frac{1}{2} \left((x_1 + 1.42513)^2 + (x_2 + 0.80032)^2 \right) \\ Where: & -10 \leq x_i \leq 10, i = 1, 2 \\ Search\ Domain: & \left\{ (x_1, x_2) \in \mathbb{R}^2 : -10 \leq x_i \leq 10,\ i = 1, 2 \right\} \end{aligned} \tag{3.1}$$

Originally, the *Shubert1* function was chosen, however it has 760 local minima, of which 18 are global minima. This made it difficult to evaluate the GA's performance as it would consistently find global minima regardless of how it was configured, most likely due to how frequently they occurred.

3.2 Configuration

3.2.1 Creation Function

The creation function in MATLAB is the function that generates the GA's initial population. MATLAB has four built-in creation functions that can be chosen from: `gacreationlinearfeasible`, `gacreationuniform`, `gacreationsobol`, and `gacreationuniformint`. In testing, `gacreationlinearfeasible` and `gacreationsobol` both converged on local minima (Figs. G.1 & G.3), whilst `gacreationuniform` and `gacreationuniformint` both converged towards the global minimum, and both converged in 67 generations.

The reason for `gacreationuniform` and `gacreationuniformint` converging in the same way is likely due to the fact that `gacreationuniformint` operates in the same way as `gacreationuniform`, however it then applies any integer constraints present onto the initial population once it has been generated. As our problem doesn't have any integer constraints, both creation functions were generating populations in the exact same way as one another, thus creating similar results.

As these functions produce similar results, the `gacreationuniform` creation function was chosen to avoid making the algorithm unnecessarily attempt to apply integer constraints which aren't applicable for our problem.

3.2.2 Selection Strategy

After some trial and error, the tournament selection algorithm was selected as it repeatedly produced the best results when testing the convergence compared to a multitude of strategies (Fig. G.5). Tournament selection works by randomly choosing groups of n individuals from the population and selecting the fittest individual from each group to be a parent for the next generation. With tournament selection, the size of n can make a large impact on how the parents are selected. If $n = 1$, then the result is just a random selection of individuals (1-way tournament), and the larger n is, the more likely it is that the resulting parent pool will be made up of stronger individuals. This is because if n is increased, groups of larger sizes are selected, eliciting a higher probability that a fitter individual will be selected from each tournament, lowering the number of weaker candidates that make it through. Thus, n must be configured to balance exploitation and exploration in selection.

3.2.3 Fitness Function

The *Shubert2* function was used without any modifications for the fitness function, as the GA is simply searching for the values that produce the minimum output from it. No modifications are needed as we are not using fitness proportional selection as our selection strategy, which is effected by negative fitness values. Fitness proportional

selection works by calculating each chromosome's percentage of the population's total fitness and subsequently selecting those to be reproduced based upon their calculated percentages. If there are negative fitness values, it ruins the calculations of the population's total fitness and each chromosome's percentages, thus negatively affecting selection. To fix this, the fitness function would need to be offset by adding some specific value such that the outputs are always positive. However, as the GA is using the tournament selection strategy, negative values have no negative effect as chromosomes are simply compared in groups of size n, selecting the chromosome with, in this case, the lowest fitness value to be reproduced, which still functions with negative values.

Additionally, a fitness scaling function is used to convert the raw fitness function scores for each chromosome to be more suitable for use in the selection function. `fitscalingrank` was used as it provided the most consistently accurate results from the GA. `fitscalingrank` scales the raw fitness function scores based on on the chromosome's position in the sorted scores (their rank) using equation 3.2.

$$\frac{1}{\sqrt{r}} \tag{3.2}$$

where r is the chromosome's rank

3.2.4 Reproductive Operators

Crossover Function

The two-point crossover function (`crossovertwopoint`), was selected to be used as the crossover function. Out of all the available crossover functions, `crossoversinglepoint`, `crossovertwopoint`, and `crossoverlaplace` were the only functions which converged toward the global minimum, with `crossovertwopoint` converging in the least generations. Two-point crossover works similarly to single point crossover, randomly selecting two crossover points from the parent chromosomes and trading the bits between the selected points between the parents to produce their offspring.

Mutation Function

Because of the *Shubert2*'s search domain (Eq. 3.1), the `mutationgaussian` and `mutationuniform` were unsuitable for the algorithm, as these mutation functions could create children that search outside of this domain (The MathWorks, Inc., n.d.). This left `mutationadaptfeasible`, `mutationpositivebasis`, and `mutationpower` to choose from. Out of these three, `mutationadaptfeasible` was the only one which converged towards the global minimum during testing (Figs. G.6, G.7, & G.8). Because of these results, we decided to use `mutationadaptfeasible` going forward.

3.2.5 Stopping Criteria

Once again, the stopping criteria has not been altered from the defaults set by MATLAB. Numerous variations were tested, with none providing as accurate or consistent results as the defaults. The set stopping criteria can be seen in table 3.1. The criteria most often met was `FunctionTolerance`, which is used in conjunction with `MaxStallGenerations`, stopping the GA if the average change of the best fitness function value is less than or equal to `FunctionTolerance` for over `MaxStallGenerations` generations. Altering `MaxStallGeneration` can ensure a minimum number of generations are complete before the GA stops. Nonetheless, setting `FunctionTolerance` or `MaxStallGenerations` too small prevents the GA from exploring the search space and most often leads to premature convergence in a local minimum. On the other hand, setting them too large can lead to unnecessarily long optimisation times. The default `FunctionTolerance` and `MaxStallGenerations` criteria set by MATLAB perform the best.

Criteria	Set Value
`MaxGenerations`	200
`MaxStallGenerations`	50
`FunctionTolerance`	0.000001

Table 3.1: Stopping Criteria Used by the GA

3.2.6 Parameters

Mutation Rate

The `mutationadaptfeasible` function does not have any configurable parameters, unlike other mutation functions, such as `mutationuniform` which has `MUTATIONRATE` and `mutationgaussian` which has both `SHRINK`, and `SCALE`. For this reason, the mutation rate of the GA cannot be set, and so no parameters for it were changed.

Population Size

The default population size in MATLAB is 50 when the number of variables ($nvar$) ≤ 5 and 200 when $nvar > 5$. As the *Shubert2* function accepts two variables, the default population size for our genetic algorithm is 50.

Changing the population size for a GA can seriously affect its performance. Having a population size which is too small will often lead to finding a worse solution due to the limited amount of exploration that can occur, but it will be computationally cheaper. Conversely, choosing a large population size will often find a better result

with less generations due to the increased diversity of each generation, however the larger population size will also require more computational power to process. The initial population range can also affect the convergence in a similar way, as having a larger range can lead to a more diverse population, which can help convergence.

An array of population sizes were tested, ranging from 25 up to 10,000. The population sizes ranging from 25 to 50 had a similar convergence time, both in time taken and generations taken, whilst the accuracy of the solution improved as the size increased. The population sizes ranging from 50 to 200 showed no differences in how many generations were required to converge, with minimal increases in the solution accuracy, however there was a minor increase in the time taken to converge as the population size increased. For the sizes ranging from 200 to 10,000, the time taken increased rapidly as the size increased and there was no noticeable change in solution accuracy. Additionally, the generations taken to converge increased somewhat as the population size grew larger.

After testing these population sizes, it was determined that the default size of 50 was the most optimal, as convergence occurs in an acceptable amount of time and the search domain is not particularly large (Eq. 3.1). This population size is able to be suitably distributed throughout the domain such that the convergence of the solution on the global minimum is reliable.

Generation Gap

The generation gap, referred to as the crossover fraction by MATLAB, was set to 0.8 (80%). The generation gap is used to specify the percentage of the population's chromosomes to be replaced when producing the next generation, allowing some portion of chromosomes to survive into the next generation. With a higher generation gap, more chromosomes are replaced, favouring exploration, whilst a lower generation gap favours exploitation, allowing more chromosomes to survive into the next generation. Those that survive into the next generation are picked using the same selection strategy for reproduction, most likely resulting in fitter individuals surviving. Using a generation gap of 0.8, most of the population is replaced, with a small portion of the, most likely, fittest individuals surviving into the next generation, providing an ideal ratio between exploration and exploitation. Other generation gaps were tested, but none performed as well as 0.8.

3.3 Results

3.3.1 Final configuration

The final configuration options can be seen below in table 3.2.

Option	Chosen Value
Population Size	50
Creation Function	`gacreationuniform`
Crossover Function	`crossovertwopoint`
Crossover Fraction	0.8
Selection Function	`selectiontournament`
Mutation Function	`mutationadaptfeasible`
Fitness Scaling Function	`fitscalingrank`

Table 3.2: Final GA configuration settings.

3.3.2 Results Analysis

To test the GA produces consistent results, due to the initial population being generated randomly, four runs were completed. The fully configured GA reached the global minimum of -186.730909 for three runs out of the four conducted. These three runs completed in 65, 72, and 75 generations, with x_1 and x_2 equalling -1.4251 and -0.8003 respectively (Figs. H.1, H.2, & H.4). The remaining run reached a local minimum of -186.340605, close to the global minimum, in 78 generations, with x_1 and x_2 equalling -0.8005 and -1.4250 respectively (Fig. H.3). As discussed in the previous sections, multiple different configurations were required to ensure the GA did not prematurely converge, which seems to have been successful, converging at the global minimum the majority of the time. Convergence was relatively rapid, generally reaching the initial global minimum within approximately 20–30 generations. This could prompt an alteration to the GA's stopping criteria, more specifically the maximum stall generations (`MaxStallGenerations`), which could be lowered to allow the GA to exit sooner after reaching the global minimum, however it is uncertain how this would affect the accuracy.

3.4 Conclusion

In conclusion, the configured GA produces accurate results and converges at the *Shubert2* function's global minimum most of the time, failing to do so only once out of the four runs performed. Nevertheless, the GA could still be improved, with one potential improvement being the re-configuration of its stopping criteria. As previously stated, the GA converges on the global minimum rapidly and so optimising the stopping criteria will allow the GA to take advantage of this and exit rapidly too.

If this study were to be performed again, a significant recommendation would be to create custom functions for some of the GA's operators, e.g. mutation. This would allow for greater control over the GA's performance and provide a method to implement non-standard methods to improve performance.

Bibliography

Feed-forward and feedback networks. (n.d.). O'Reilly. https://www.oreilly.com/library/view/neural-networks-with/9781788397872/eb49931a-7122-4b97-a843-bb6e1817cc12.xhtml

Fisher, R. A. (1936). *Iris data set*. Retrieved April 13, 2022, from http://archive.ics.uci.edu/ml/datasets/Iris

Fonseca, D. J., Shishoo, S., Lim, T. C., & Chen, D. S. (2005). A genetic algorithm approach to minimize transmission error of automotive spur gear sets. *Applied Artificial Intelligence*, *19*(2), 153–179. https://doi.org/10.1080/08839510590901903

Jordanov, I., & Georgieva, A. (2009). Neural network classification: A cork industry case. *2009 IEEE International Symposium on Industrial Electronics*, 232–237. https://doi.org/10.1109/ISIE.2009.5213287

Kelso, F., & Society, T. O. B. (2011). *Plant Lore of an Alaskan Island: Foraging in the Kodiak Archepelago* (2nd ed.).

LeCun, Y., Cortes, C., & Burges, C. J. (n.d.). *THE MNIST DATABASE of handwritten digits*. http://yann.lecun.com/exdb/mnist/

Luce, R. D. (2008). *Luce's Choice Axiom*. Scholarpedia. http://www.scholarpedia.org/article/Luce%5C%27s_choice_axiom

mvs, C. (2019, December 23). *Activation Functions: Why "tanh" outperforms "logistic sigmoid"?* Medium. https://medium.com/analytics-vidhya/activation-functions-why-tanh-outperforms-logistic-sigmoid-3f26469ac0d1

SethBling. (2015, June 13). *MarI/O - Machine Learning for Video Games* [Video]. YouTube. https://www.youtube.com/watch?v=qv6UVOQ0F44

The MathWorks, Inc. (n.d.). *Genetic algorithm options*. Retrieved May 23, 2022, from https://uk.mathworks.com/help/gads/genetic-algorithm-options.html

Appendix A

Iris Dataset Distribution

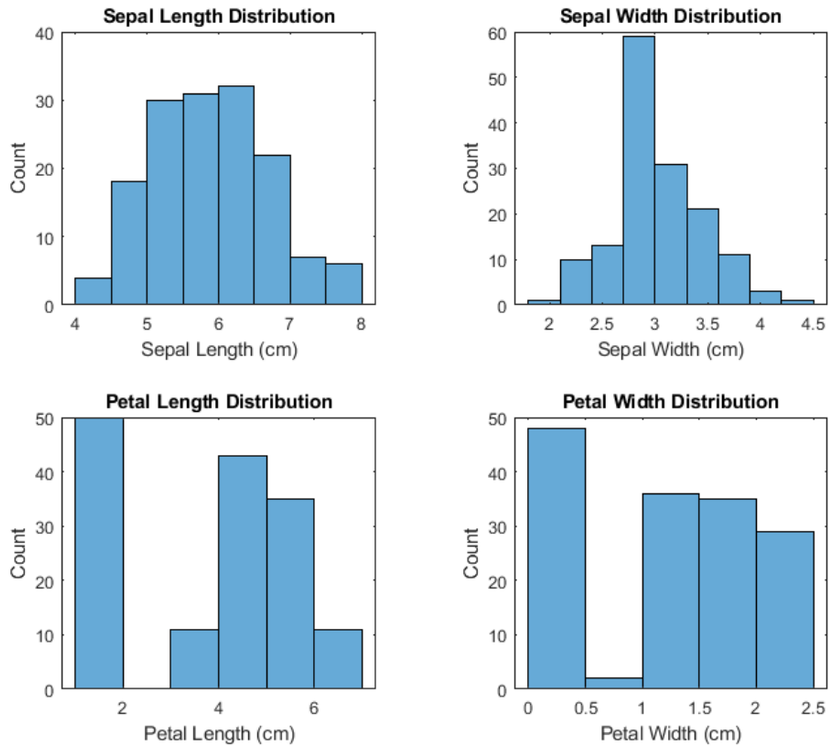


Figure A.1: Iris Dataset Distribution

Appendix B

Network structure

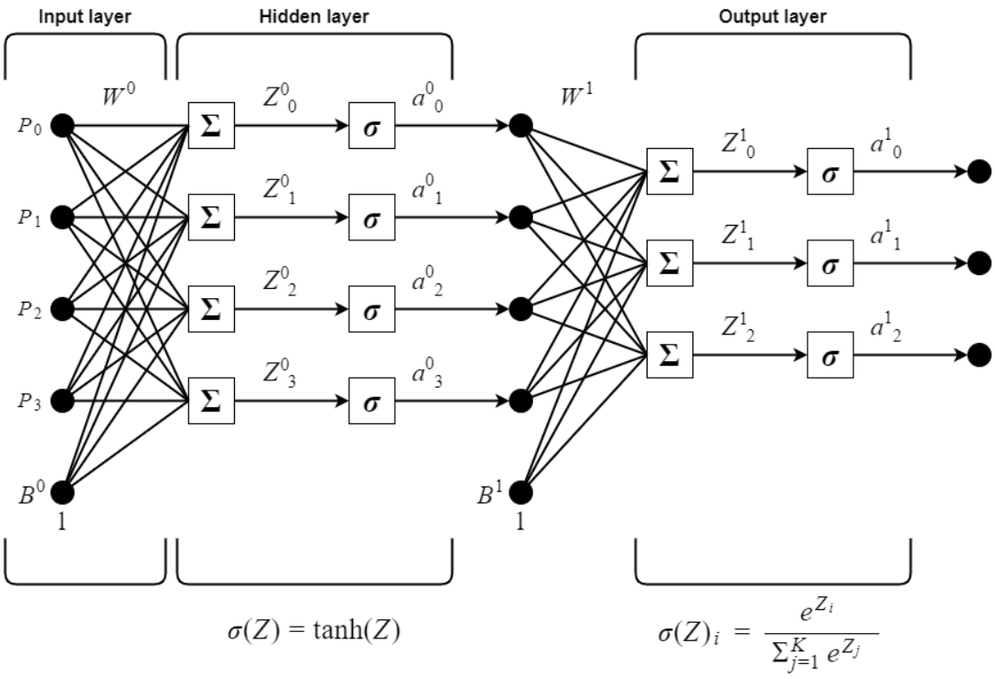


Figure B.1: Network structure.

Appendix C

Stopping Criterion Over-Fitting

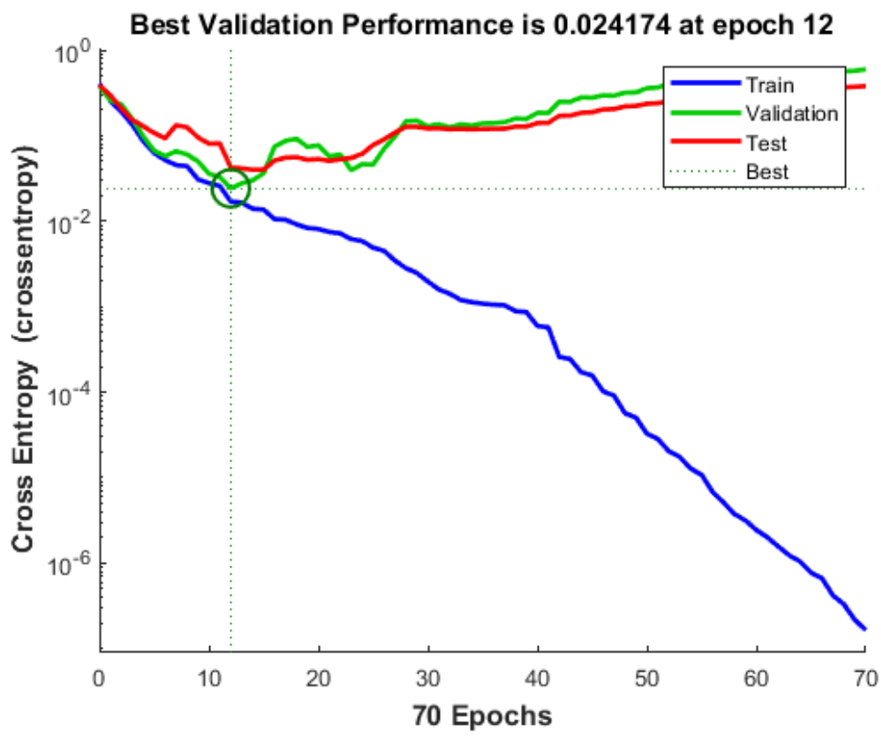
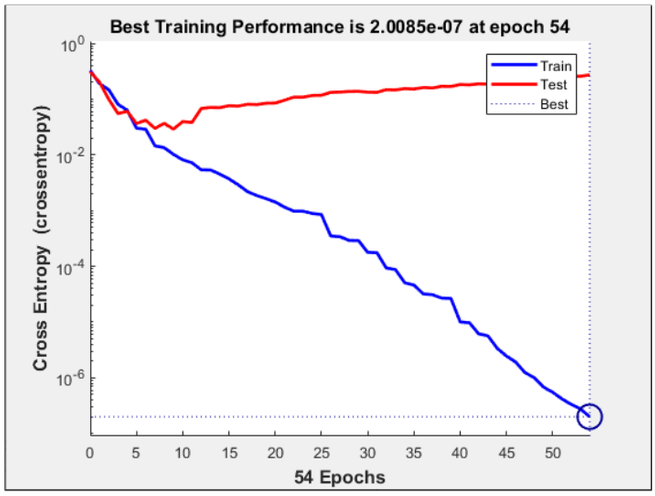


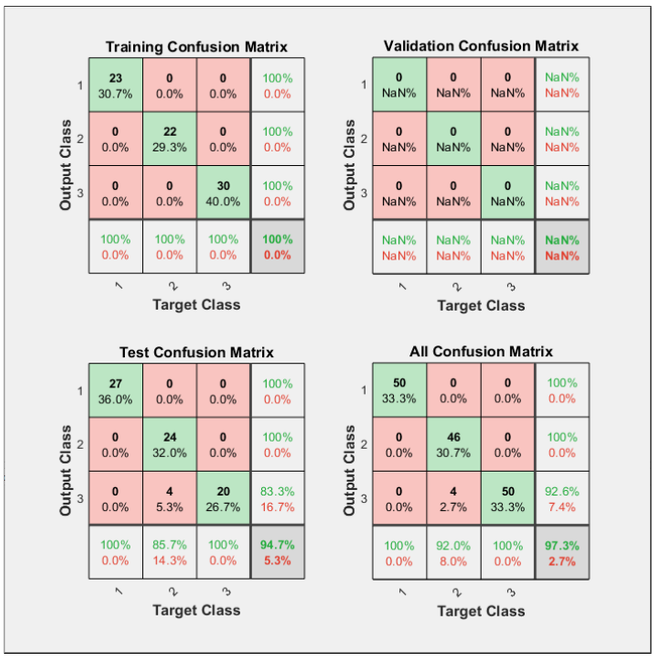
Figure C.1: Performance plot of the neural network when the maximum number of validation increases is set to 100.

Appendix D

Subset Split Plots

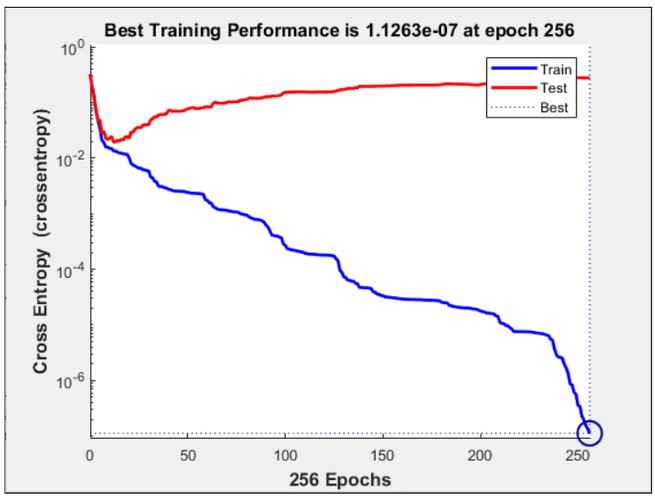


(a) Performance plot.

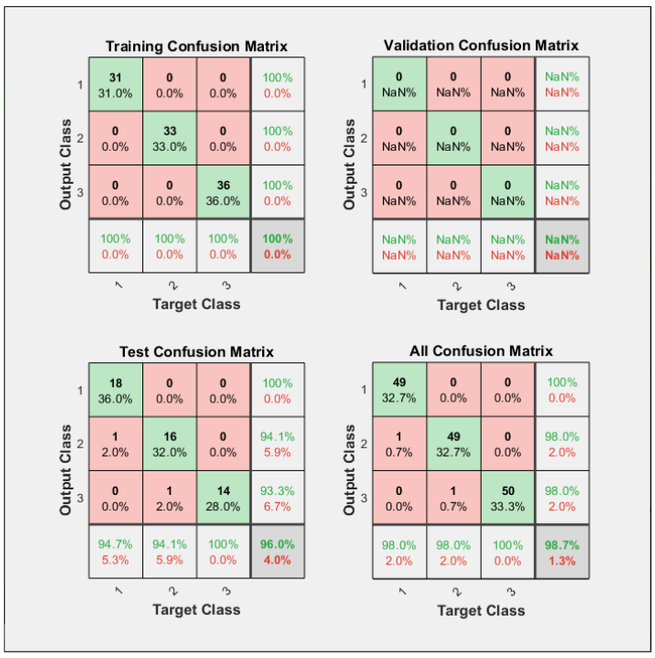


(b) Confusion matrix.

Figure D.1: Performance plot and confusion matrix for training/testing split 75/75.

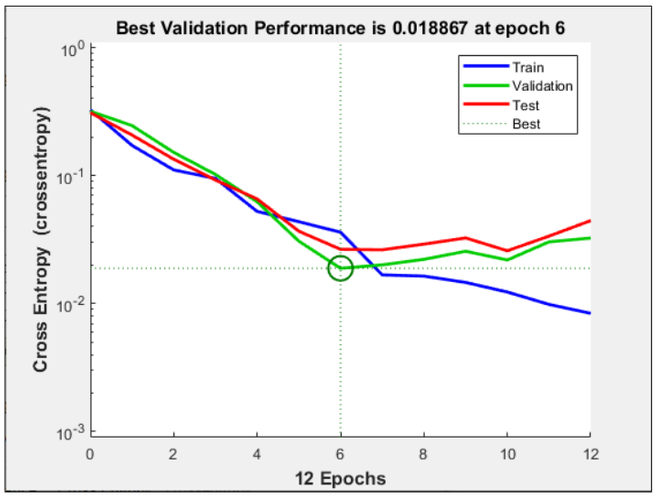


(a) Performance plot.

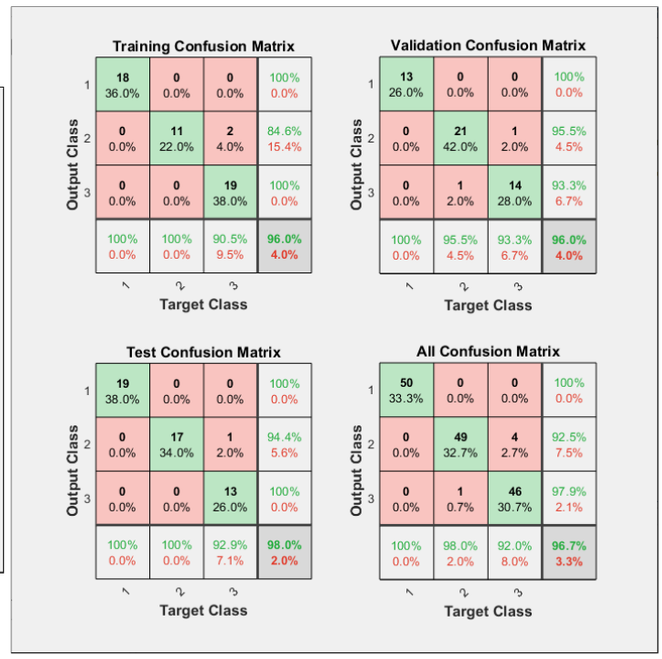


(b) Confusion matrix.

Figure D.2: Performance plot and confusion matrix for training/testing split 100/50.

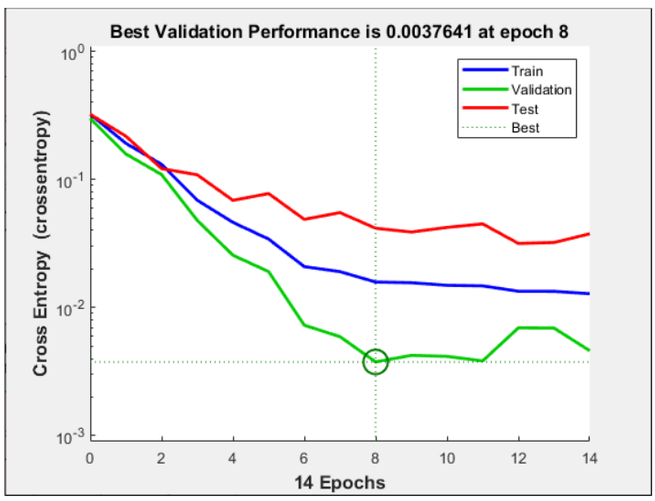


(a) Performance plot.

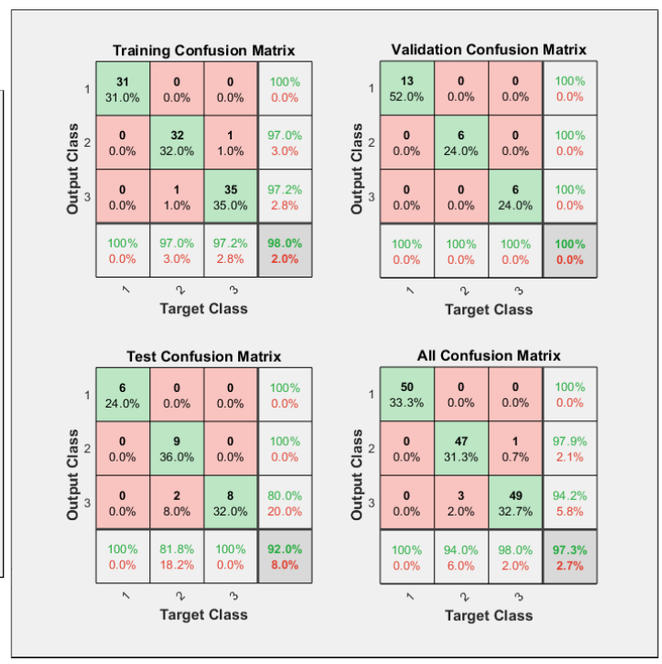


(b) Confusion matrix.

Figure D.3: Performance plot and confusion matrix for training/validation/testing split 50/50/50.



(a) Performance plot.



(b) Confusion matrix.

Figure D.4: Performance plot and confusion matrix for training/validation/testing split 100/25/25.

Appendix E

Iris class plots

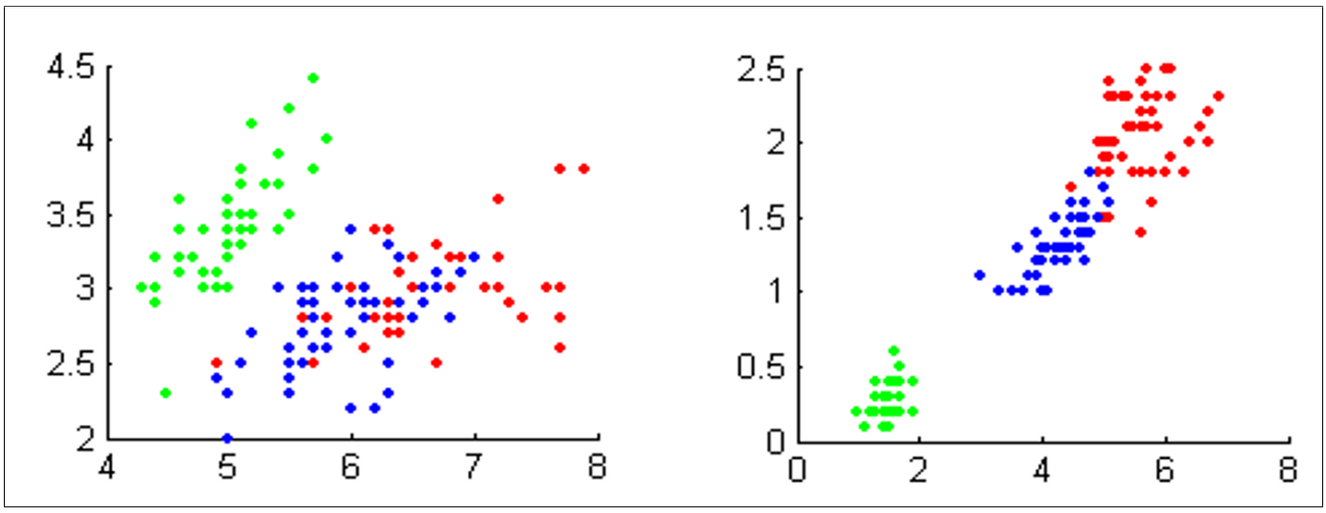
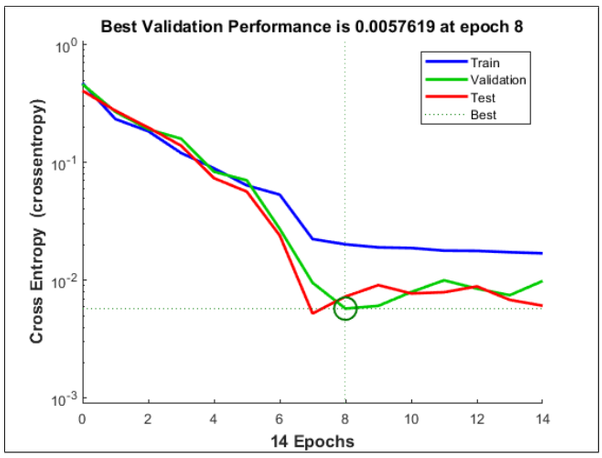


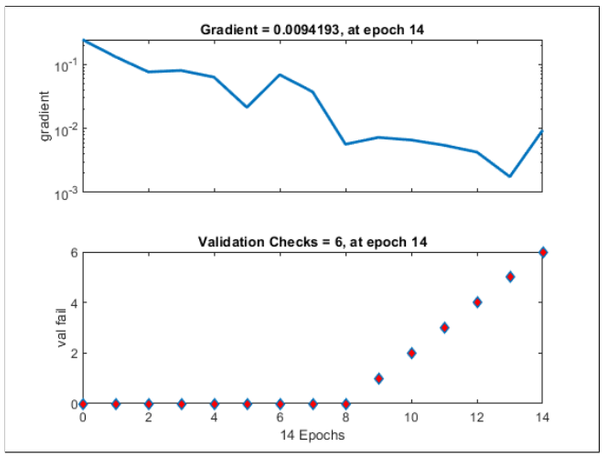
Figure E.1: Pair-wise plots of the class distribution of the three Iris classes (left: sepal width vs. sepal length; right: petal width vs. petal length). It can be seen that two of the classes are not linearly separable.
Source: CwkAppendixA.docx

Appendix F

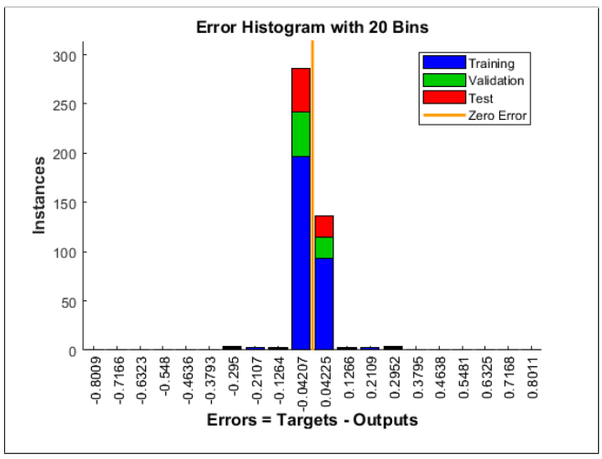
Neural Network Results



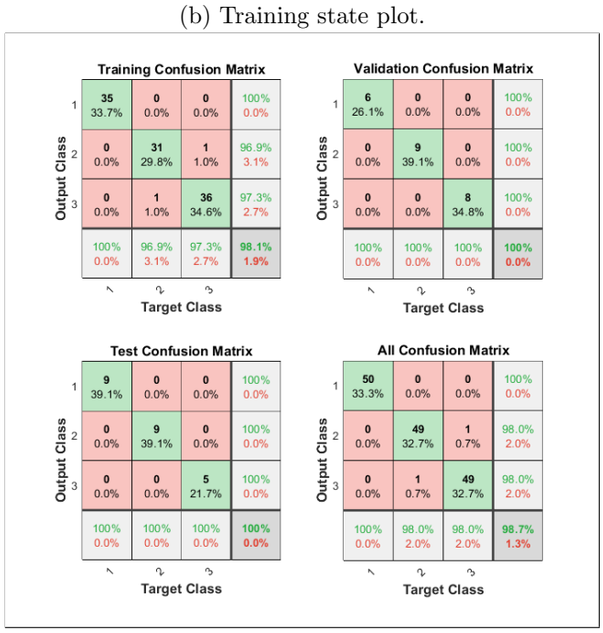
(a) Performance plot.



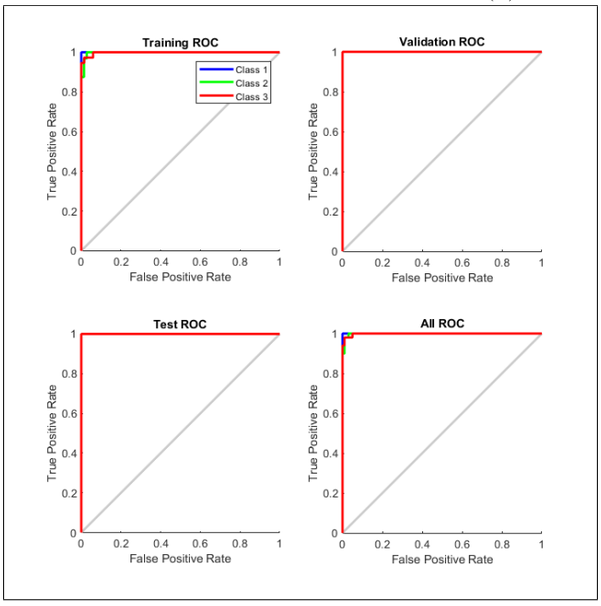
(b) Training state plot.



(c) Error histogram.



(d) Confusion matrix.



(e) Receiver Operating Characteristic (ROC) plot.

Figure F.1: Result plots from the testing of the neural network.

Appendix G

Genetic Algorithm Testing

G.1 Creation Function

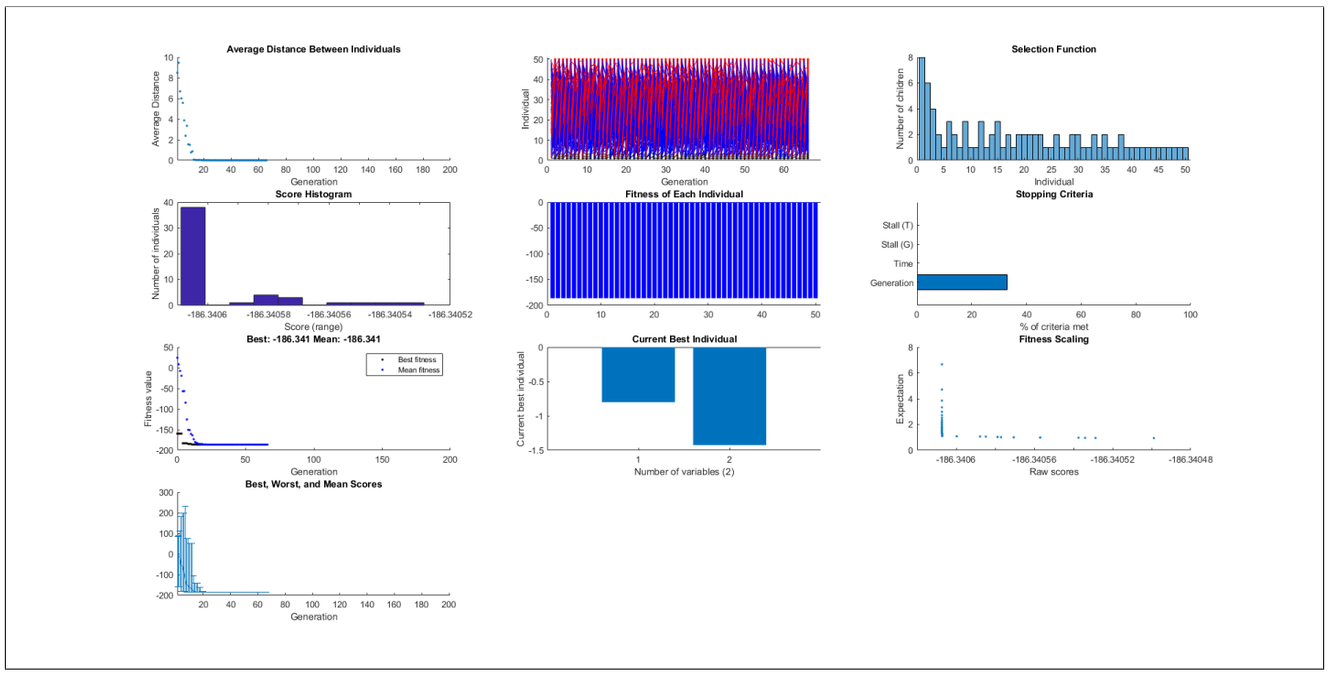


Figure G.1: Results of genetic algorithm using the Linearfeasible creation function. In 66 generations, converged towards a local minimum of -186.3406 at [-0.8005,-1.4250].

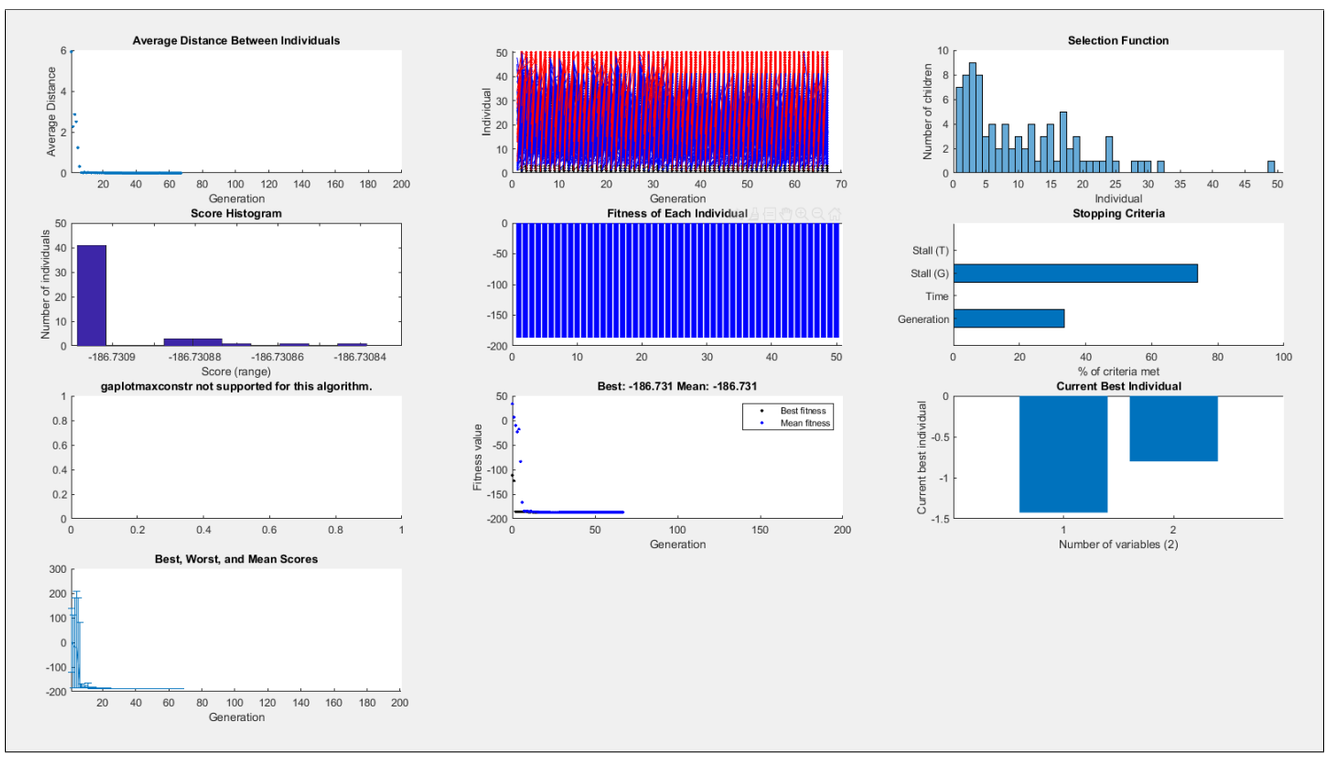


Figure G.2: Result of genetic algorithm using the Uniform creation function. In 67 generations, converged towards the global minimum of -186.7309 at [-1.4251,-0.8003].

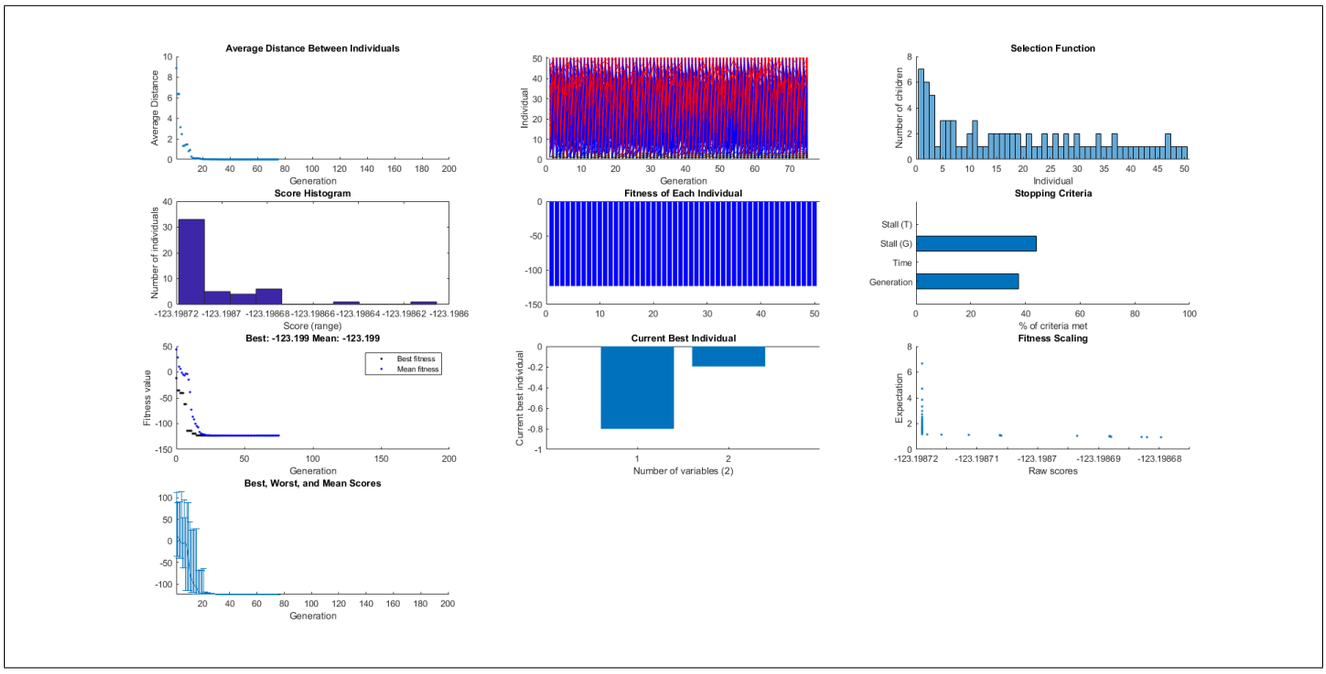


Figure G.3: Result of genetic algorithm using the Sobol creation function. In 75 generations, converged towards a local minimum of -123.1987 at [-0.8005,-0.1955].

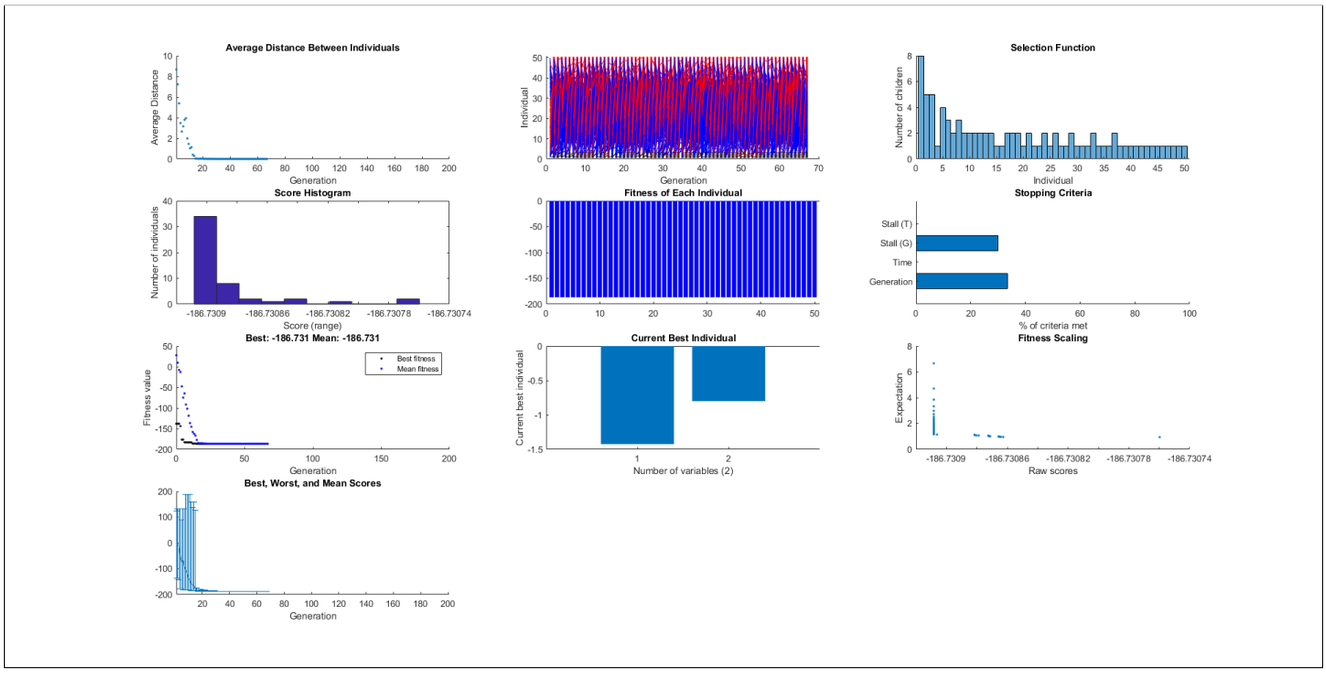


Figure G.4: Result of genetic algorithm using the Uniformint creation function. In 67 generations, converged towards the global minimum of -186.7309 at [-1.4251,-0.8003].

G.2 Selection Function

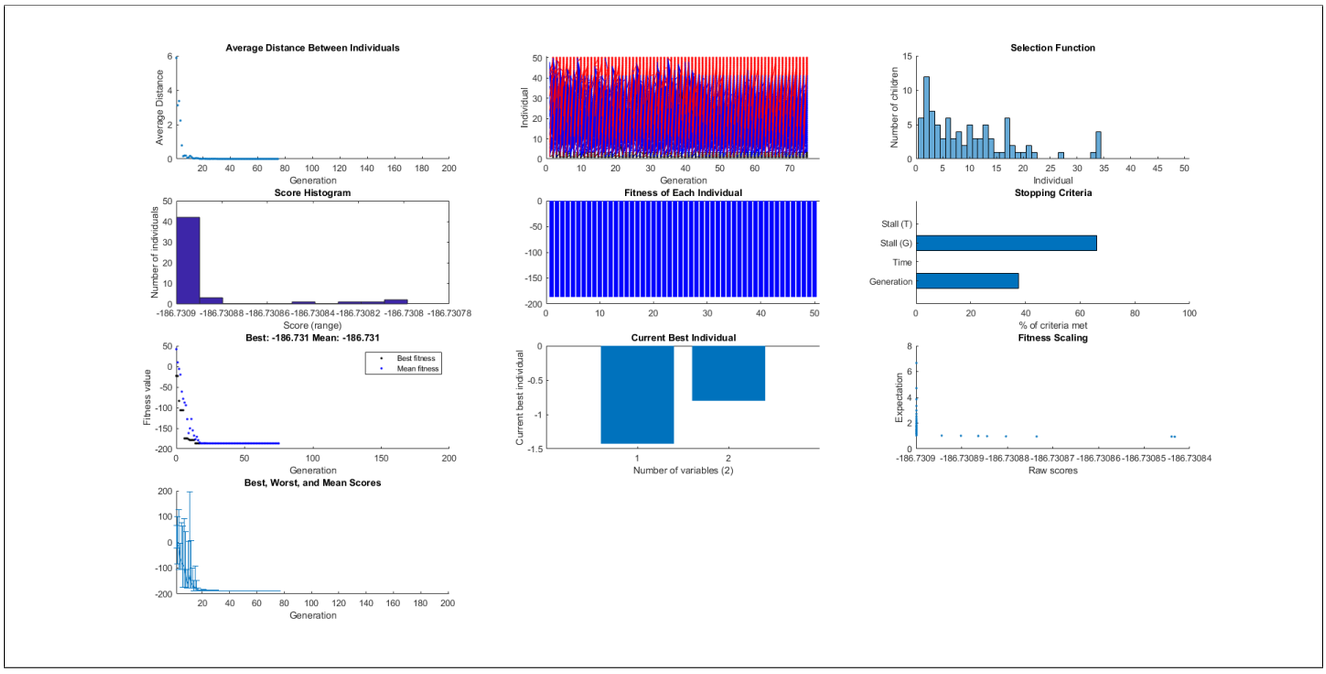


Figure G.5: Result of genetic algorithm using the Tournament selection strategy. In 75 generations, converged towards the global minimum of -186.7309 at [-1.4251,-0.8003].

G.3 Mutation Function

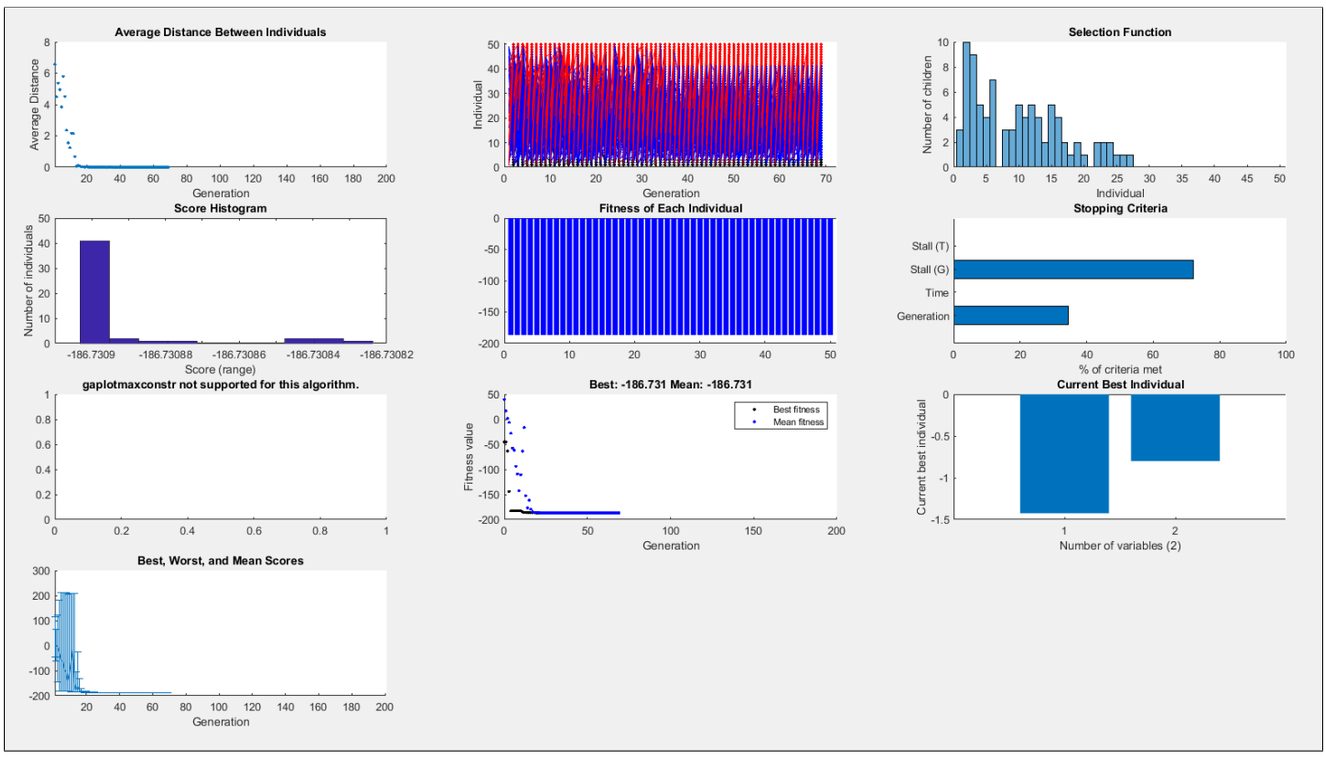


Figure G.6: Result of genetic algorithm using the Adaptfeasible mutation function. In 69 generations, converged towards the global minimum of -186.7309 at [-1.4252, -0.8003].

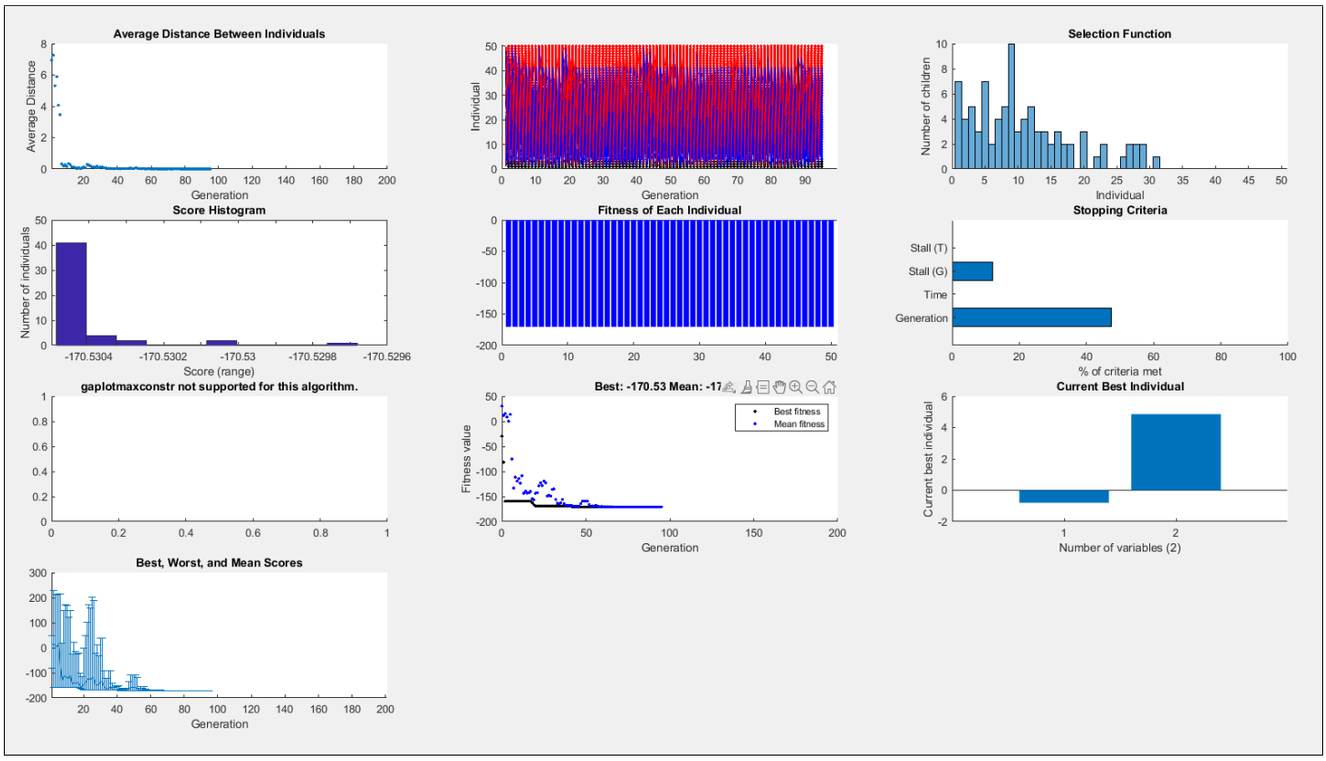


Figure G.7: Result of genetic algorithm using the Positivebasis mutation function. In 95 generations, converged towards a local minimum of -170.5305 at [-0.8003, 4.8570].

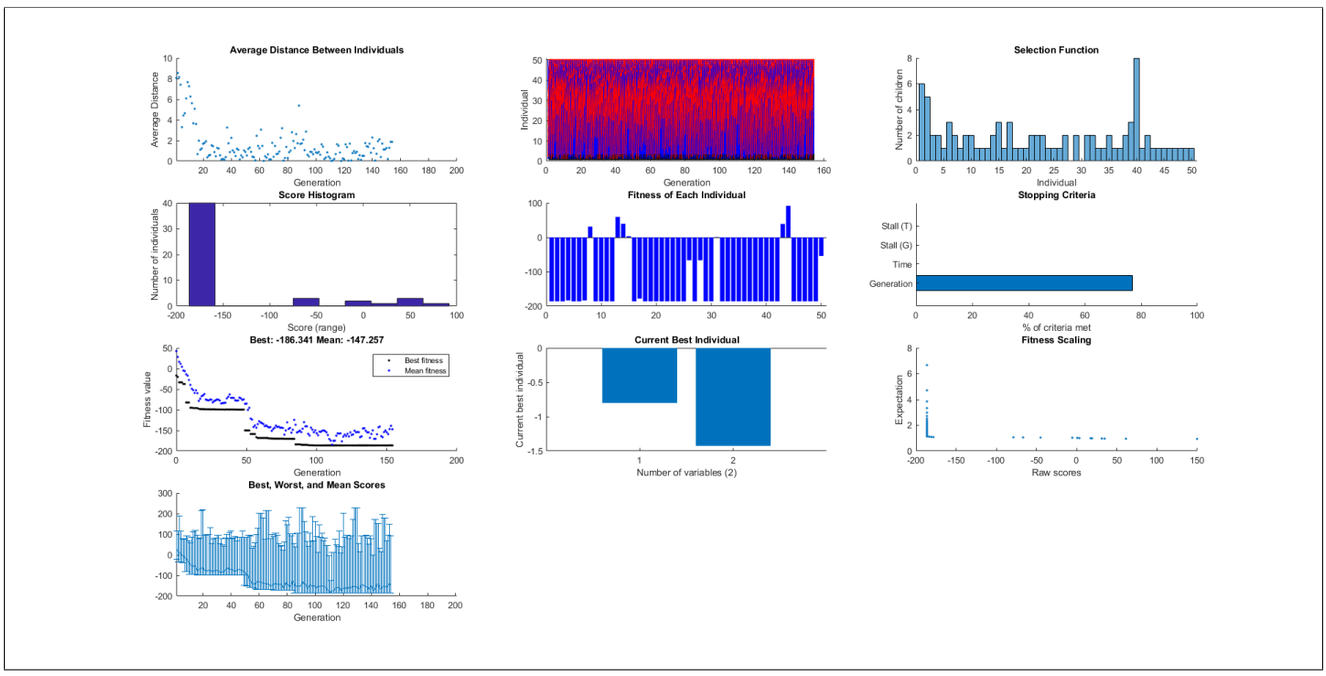


Figure G.8: Result of genetic algorithm using the Power mutation function. In 154 generations, converged towards a local minimum of -186.3406 at [-0.8005,-1.4250].

Appendix H

Genetic Algorithm Results

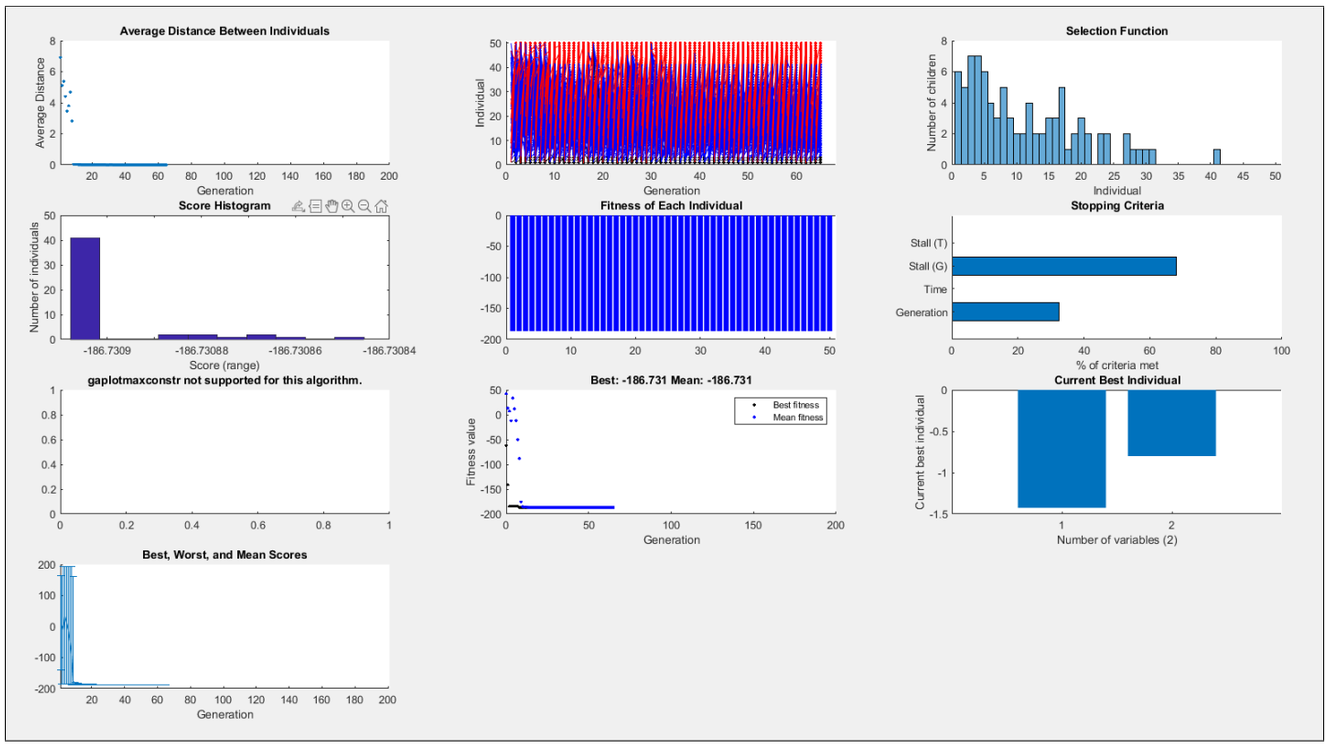


Figure H.1: Results from Run 1 of the fully configured genetic algorithm. In 65 generations, converged towards the global minimum of -186.730909 at [-1.4251, -0.8003].

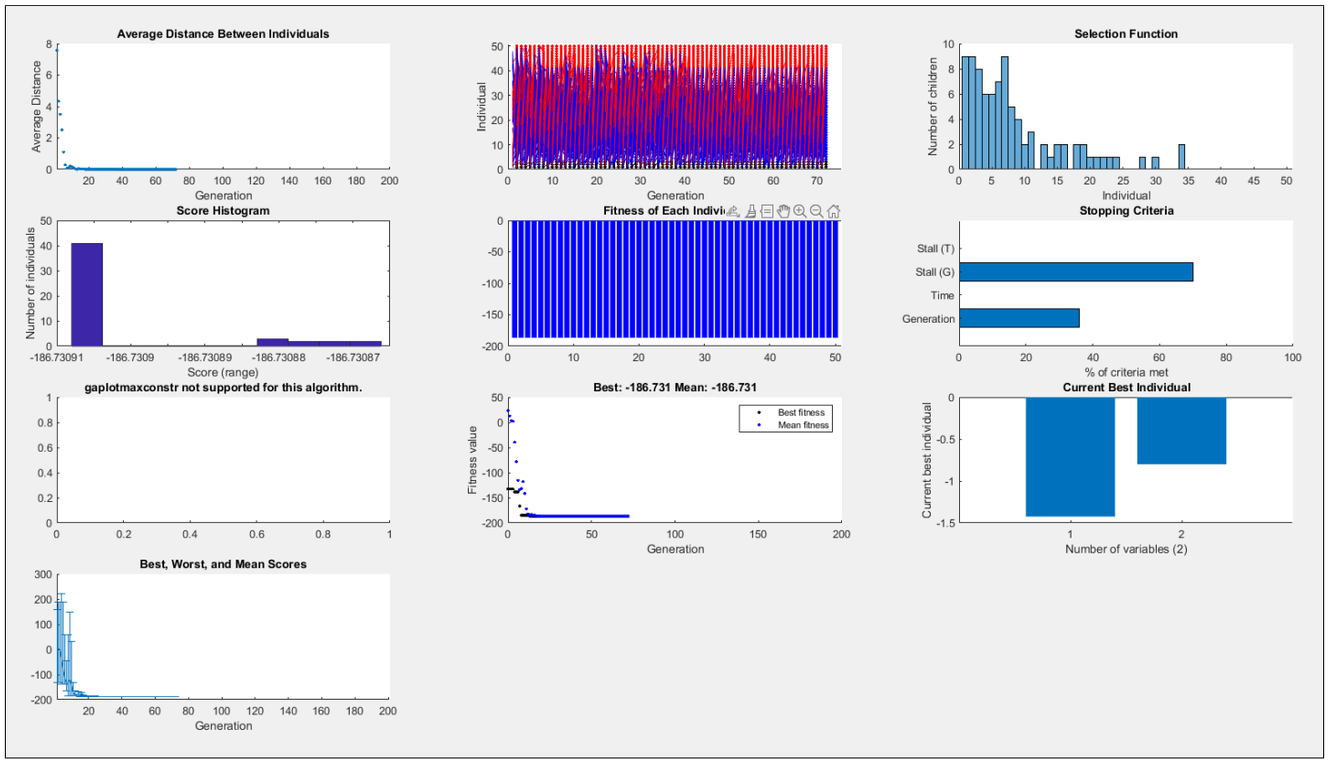


Figure H.2: Results from Run 2 of the fully configured genetic algorithm. In 72 generations, converged towards the global minimum of -186.730909 at [-1.4251, -0.8003].

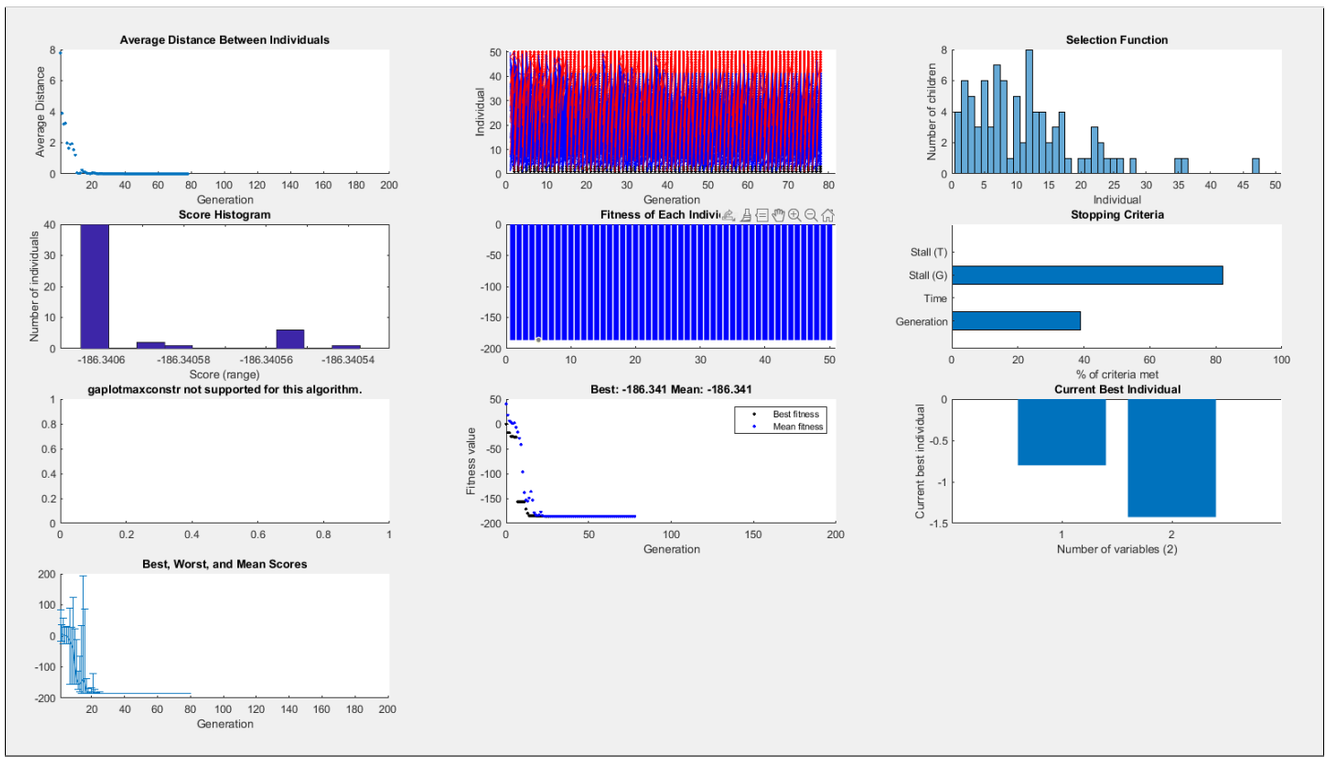


Figure H.3: Results from Run 3 of the fully configured genetic algorithm. In 78 generations, converged towards a local minimum of -186.340605 at [-0.8005,-1.4250].

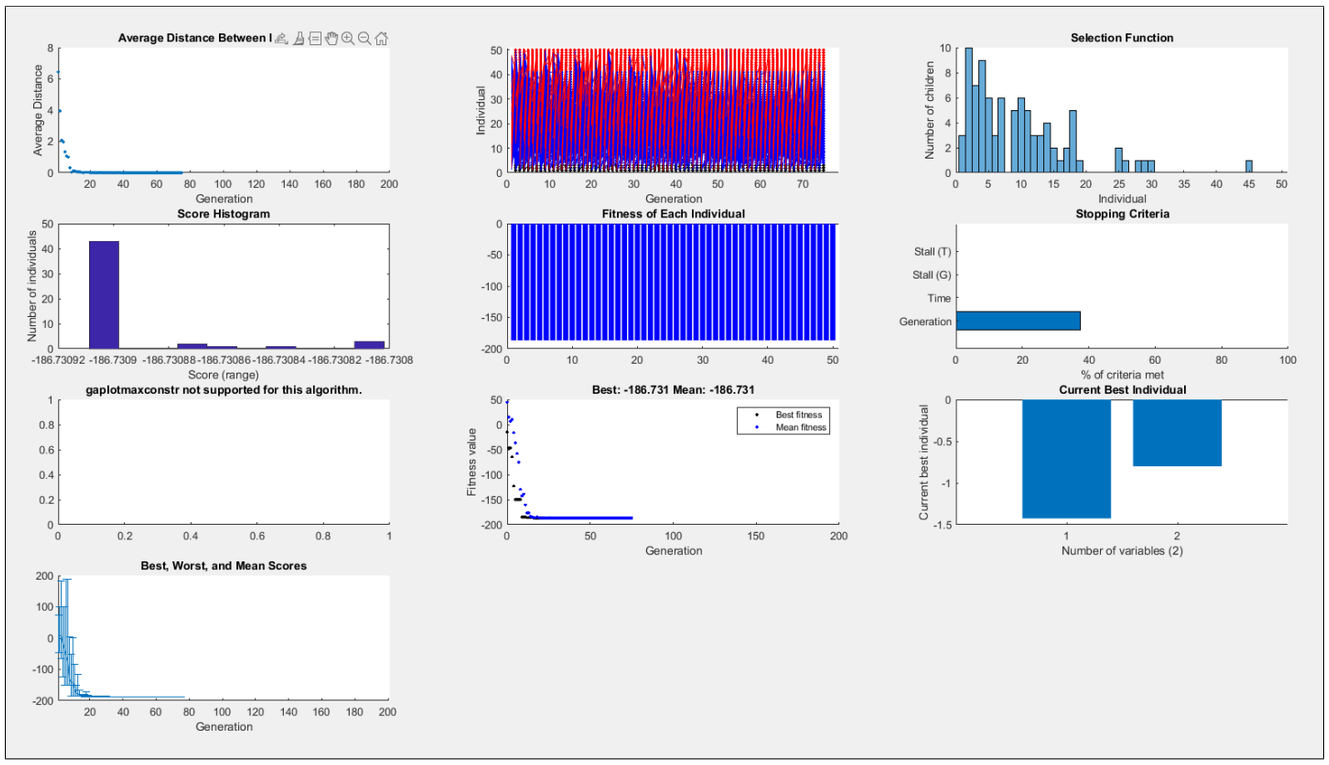


Figure H.4: Results from Run 4 of the fully configured genetic algorithm. In 75 generations, converged towards the global minimum of -186.730909 at [-1.4251, -0.8003].

Appendix I

Source Code

The repository containing the source code and other resources in this document can be found at https://github.com/UP939163/nenga-cw.

I.1 Neural Network

```
% File must be named nn.m
function nn()
    % Load and Pre-Process Data
    data = load('data.txt');
    data = preProcess(data);
    [input, target] = getInputOutput(data);
    x = input';
    t = target';

    % Choose Training Function
    trainFcn = 'trainscg';

    % Performance / Loss function
    performFcn = 'crossentropy';

    % Create a Pattern Recognition Network
    hiddenLayerSize = 4;
    net = patternnet(hiddenLayerSize, trainFcn, performFcn);
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'softmax';

    % Setup Division of Data for Training, Validation, Testing
    net.divideParam.trainRatio = 70/100;
    net.divideParam.valRatio = 15/100;
    net.divideParam.testRatio = 15/100;

    % Randomise Weights
    net = configure(net,x,t);
    net = setwb(net, rand(35,1));

    % Train the Network
    [net,tr] = train(net,x,t);

    % Test the Network
    y = net(x);
    e = gsubtract(t,y);
    performance = perform(net,t,y)
    tind = vec2ind(t);
    yind = vec2ind(y);
    percentErrors = sum(tind ~= yind)/numel(tind);

    % View the Network
    view(net)
end

% Randomise and normalise data
function [data] = preProcess(data)
    % Randomise dataset order
    data = data(randperm(size(data, 1)), :);
    % Normalise inputs
    data(:,1) = normalise(data(:,1), 5.84, 0.83);
    data(:,2) = normalise(data(:,2), 3.05, 0.83);
    data(:,3) = normalise(data(:,3), 3.76, 0.83);
    data(:,4) = normalise(data(:,4), 1.2, 0.83);
end
```

```
function [normalised] = normalise(data, mean, std)
    zScore = (data - mean) ./ std;
    minZ = min(zScore);
    maxZ = max(zScore);
    normalised = ((zScore - minZ) / (maxZ - minZ) * 2) - 1;
end

function [input, output] = getInputOutput(data)
    % First 4 values are input
    input = data(:,1:4);
    % Fifth value is classification
    % Initialise empty matrix
    output = zeros(size(data, 1), 3);
    for r = 1:size(output, 1)
        output(r, data(r, 5)) = 1;
    end
    disp(data(1:5, :));
    disp(output(1:5, :));
end
```

I.2 Genetic Algorithm

```
% File must be named GA.m
function GA()
    % Define inputs
    nvar = 2;
    fun = @Shubert2;

    % Set nondefault solver options
    options = optimoptions("ga",...
        "PopulationSize",50,...
        "CreationFcn","gacreationuniform",...
        "CrossoverFcn","crossovertwopoint",...
        "SelectionFcn","selectiontournament",...
        "MutationFcn","mutationadaptfeasible",...
        "MaxGenerations",200,...
        "Display","iter",...
        "PlotFcn",[...
            "gaplotdistance",...
            "gaplotgenealogy",...
            "gaplotselection",...
            "gaplotscorediversity",...
            "gaplotscores",...
            "gaplotstopping",...
            "gaplotbestf",...
            "gaplotbestindiv",...
            "gaplotexpectation",...
            "gaplotrange"...
        ]...
    );

    % Solve
    [solution,objectiveValue] = ga(fun,nvar,[],[],[],[],repmat(-10,nvar,1),...
    repmat(10,nvar,1),[],[],options);

    % Clear variables
    clearvars options
end

function y = Shubert2(x)
    n = length(x);
    p = 1.0;
    for i = 1: n
        s = 0.0;
        for j = 1:5
```

```
            s = s+j.*cos((j+1).*x(i)+j);
        end
        p = p.*s;
    end
    y = p + (0.5 * ((x(1) + 1.42513).^2 + (x(2) + 0.80032).^2));
end
```