# Training a binary classification neural network with a genetic algorithm, and a review of the heuristic optimisation Grey Wolf Optimiser

**Luca Pike 18018121**

## 1 INTRODUCTION

An evolutionary algorithm (GA) is a type of optimisation algorithm, inspired by the principles of evolution in biology. The algorithm iteratively improves a population of solutions to a problem, using the mechanisms of natural selection and evolution.

The task provided was to develop a GA, which was then used to design neural networks, which was to be modelled to 3 datasets. Then, to demonstrate the impacts of parameter changes, with an understanding of what is happening.

This is married together with research, describing another nature-inspired optimisation algorithm, with comparisons to GA where appropriate.

This paper's approaches the background research, explaining, evaluating, and comparing Grey Wolf Optimiser with GA. Later, systematic, and exploratory parameter changes are explored and explained.

## 2 BACKGROUND RESEARCH

### 2.1 Summary and comparison of Grey Wolf Optimiser and Genetic Algorithm

Grey Wolf Optimizer (GWO), like GA, is a metaheuristic optimisation algorithm inspired by nature. As suggested by the name, GWO's inspiration comes from the grey wolf (Canis lupus), more specifically their pack hierarchy and hunting strategy.

As described by Kumar et al. (2016), the group or 'pack' are categorised into 4 types: alpha (α), beta (β), delta (δ) and omega (ω). Figure 1 below presents the hierarchy each category holds, with the alpha holding the most authority:

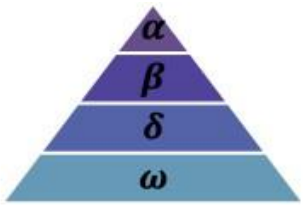


Figure 1: 'GWO hierarchy (Mirjalili, Mirjalili and Lewis, 2014)

Algorithmically, Mirjalili et al. (2014) represent the alpha as the fittest solution, the second fittest solution as beta, third fittest solution and delta and all other solutions are deemed as omega.

GWO's fitness is defined by the position of the alpha wolf in the search space, which contrasts the GA's more traditional approach of evaluating the quality of the potential solution based on a set of predefined criteria.

This difference in approach also comes into play during selection, as in GWO the alpha has the highest chance of being selected for reproduction. In contrast, GA uses a selection process based on the fitness of the individual as the most likely to be selected for reproduction

Muro et al. (2011) describes the hunting method of grey wolves as tracking and encircling the prey, and then attacking the prey.

To encircle the prey, the alpha, beta, and delta wolves lead the pack during the hunting process, where the pack moves together towards the global optimum solution. The alpha wolf uses its knowledge and experience to guide the pack towards the prey, as it constantly adjusts it's position.

Provided by Mirjalili et al. (2014), GWO's encircling behaviour can be mathematically represented, where Xp and X are the position vectors of prey and grey wolf (Kumar et al., 2016), as:

$$\vec{D} = |\vec{C} \cdot \vec{X}_p(t) - \vec{X}(t)|$$

Figure 2 (Mirjalili, 2014)

$$\vec{X}(t+1) = \vec{X}_p(t) - \vec{A} \cdot \vec{D}$$

Figure 3 (Mirjalili, 2014)

T indicates the current iteration, and $\vec{A}$ and $\vec{C}$ are coefficient vectors (Mirjalili et al., 2014). These vectors are calculated as:

$$\vec{A} = 2\vec{a} \cdot \vec{r}_1 - \vec{a}$$

$$\vec{C} = 2 \cdot \vec{r}_2$$

Figure 4 (Mirjalili, 2014)

Hunting in grey wolves is typically done by the alpha, however the beta and delta may sometimes be involved. In the wild, the wolves would know the location of the prey, where the opposite is true in the abstract search space for the optimum (prey). Mirjalili et al. (2014) therefore simulate this behaviour by saving the best three solutions so far (alpha, beta, and delta), and command the omega wolves to update their positions in accordance with the best search candidate (Hatta, N.M. et al., 2019).

Here is the formula that formulas proposed by Mirjalili et al. (2014) to achieve this:

$$\vec{D}_\alpha = |\vec{C}_1 \cdot \vec{X}_\alpha - \vec{X}|, \vec{D}_\beta = |\vec{C}_2 \cdot \vec{X}_\beta - \vec{X}|, \vec{D}_\delta = |\vec{C}_3 \cdot \vec{X}_\delta - \vec{X}|$$

$$\vec{X}_1 = \vec{X}_\alpha - \vec{A}_1 \cdot (\vec{D}_\alpha), \vec{X}_2 = \vec{X}_\beta - \vec{A}_2 \cdot (\vec{D}_\beta), \vec{X}_3 = \vec{X}_\delta - \vec{A}_3 \cdot (\vec{D}_\delta)$$

$$\vec{X}(t+1) = \frac{\vec{X}_1 + \vec{X}_2 + \vec{X}_3}{3}$$

Figure 5 (Mirjalili et al., 2014)

Figure 6 provides a visual representation of the positional updating:

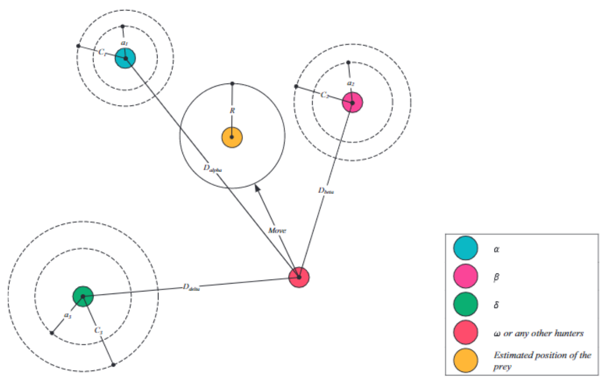


Figure 6 (Mirjalili et al., 2014)

Lastly, the hunt is concluded with attacking the prey, which occurs once the prey stops moving. The wolf approaching the prey is mathematically modelled by decreasing the value of $\vec{a}$, which also results in a decrease of $\vec{A}$ (Mirjalili et al., 2014). Essentially, $\vec{A}$ is a random value in the range of [-2a, 2a], where a decreases from 2 to 0, edging closer the prey over iterations. As presented, attacking the prey is how GWO conducts exploitation.

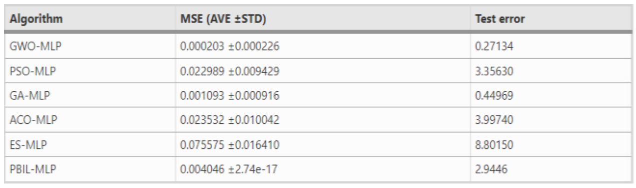
Opposingly, GA exploits by refining and improving best solutions found so far. This is done by selecting two solutions from the current population, generating their offspring, and adding the best offspring to the next generation. The offspring also goes crossover and mutation, which provides a generation that shares many characteristics of their parents (Kumar, M., 2010).

GWO employs exploration by searching for prey. As Hatta, N.M. et al. mentions, based on the position of their superiors, omega diverge from other members of the pack to search, and converge to attack prey (2019). A is used as either greater than 1, or less than -1 to signal search and converge from prey. However, it is key to point out that Mirjalili et al. admit that the GWO is prone to local solutions with the proposed operators, and so more operators to emphasize on exploration is required (2014).

## 2.2 Related work and benchmarks

Since GWO's introduction in 2014, the algorithm has gained popularity, providing a large number of real-world applications and benchmarks.

In 2015, Mirjalili (2015) applied GWO to training a Multi-Layer Perceptron (MLP), which has significant relevance towards how we employed our GA. Below are the results from a sigmoid dataset, with 61 training samples, and 121 testing samples:

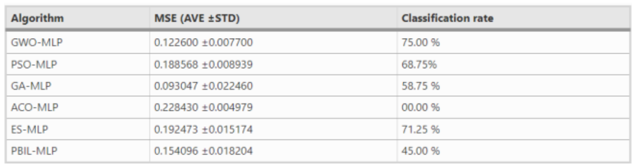


| Algorithm | MSE (AVE ±STD) | Test error |
|---|---|---|
| GWO-MLP | 0.000203 ±0.000226 | 0.27134 |
| PSO-MLP | 0.022989 ±0.009429 | 3.35630 |
| GA-MLP | 0.001093 ±0.000916 | 0.44969 |
| ACO-MLP | 0.023532 ±0.010042 | 3.99740 |
| ES-MLP | 0.075575 ±0.016410 | 8.80150 |
| PBIL-MLP | 0.004046 ±2.74e-17 | 2.9446 |

Figure 7 (Mirjalili, 2015)

Very clearly, both GWO and GA yield the best results, with GWO outperforming GA.

However, GWO has better performance with more complex datasets. Here is the comparison of the two with a binary classification dataset containing 22 features, where the MLP structure is 22-45-1 (Mirjalili, 2015):



| Algorithm | MSE (AVE ±STD) | Classification rate |
|---|---|---|
| GWO-MLP | 0.122600 ±0.007700 | 75.00 % |
| PSO-MLP | 0.188568 ±0.008939 | 68.75% |
| GA-MLP | 0.093047 ±0.022460 | 58.75 % |
| ACO-MLP | 0.228430 ±0.004979 | 00.00 % |
| ES-MLP | 0.192473 ±0.015174 | 71.25 % |
| PBIL-MLP | 0.154096 ±0.018204 | 45.00 % |

Figure 8 (Mirjalili, 2015)

The comparisons exhibit GWO's ability to provide more accurate results without additional computation, proving it to be very adept optimiser in reference to MLPs.

## 3 EXPERIMENTATION

### 3.1 Brief overview of GA

The GA contains a population of neural networks, containing a single hidden layer. It utilises the sigmoid activation function, and uses common GA functions such as mutation, recombination, evaluation and parent selection. Crossover has been omitted, due to the nature of the neural networks.

The model is a binary classification MLP that is searching for the global minimum. Thus, forward passes are made between layers during the evaluation. Based on the output from the output node, error is applied to the network when providing the incorrect answer.

The initial parameters are as follows:

- Mutation rate: 0.2
- Mutation step: 0.3
- Population size: 50
- Generations: 50
- Hidden nodes: 4
- Mutation range: -5.12, 5.12

### 3.2 Overview of testing

Initially for the experimentation of my algorithm's parameters, I wanted a way to gauge the impact different parameters would have on the accuracy of the model. Therefore, I initially began with a systematic approach of varying parameters, either individually or in pairs, and evaluate the outcomes.

The systematic testing was only conducted on one dataset (dataset 2), containing 1,000 rows. I made this decision as I felt the other data sources were either too large, or too small, and testing on these data sources would be redundant.

In order to bypass random fluctuation, and to better see the value of parameters, all tests and results provided in this section are a result of an average of 3 runs. The tests were run with a ratio of 2:1 training data to testing

The systematic tests ran were on the following:

- Mutation rate
- Mutation step
- Mutation rate and Mutation step
- Population size
- Hidden node number

### 3.3 Systematic testing

#### 3.3.1 Mutation rate and Mutation step

Firstly, I incremented mutation rate and mutation step from 0.1 to 1.0 independently

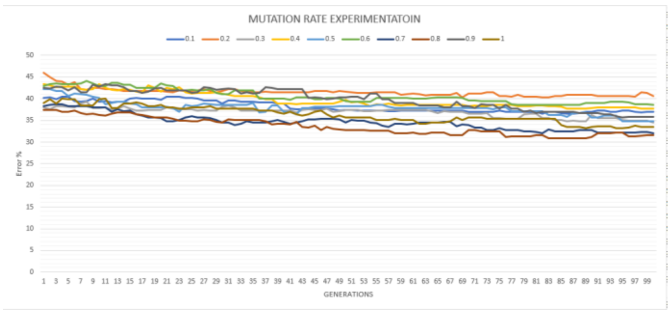


Figure 9: Mutation rate increment

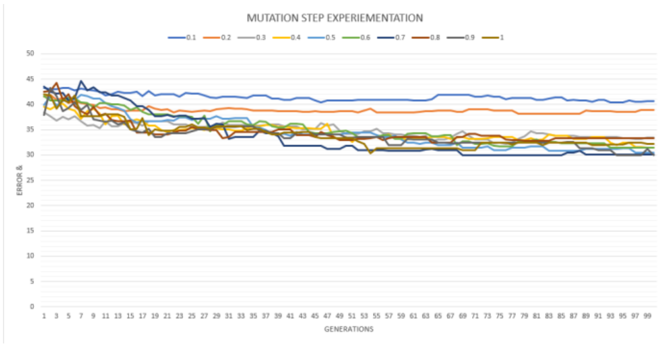


Figure 10: Mutation step increment

In mutation rate, there is a general pattern that the higher the value, the lower the error rate is, with 0.8 and 0.7 proving best. Despite this, mutation rate 0.6 proved to be the second worst, displaying a lot of volitility.

A higher mutation rate means that there is a great chance of introducing new, random solutions into the popualtion, which can help the algorithm escape from local optima. However, a high mutation rate can also cause the algorithm to lose good solutions that have been found, which can hinder its ability to converge on an optimal solution

In mutation step, there is a much clearer correlation between the value and the error rate. Overall, 0.7 proved the best, with 0.1 and 0.2 the worst.

Mutation step determins the magnitude of the changes made to a solution during mutation. There its implications are the same as mutation rate.

It is important to note that incrementing these values independetly of each other lacks thorough analysis of mutation rate's and mutation step's relationship.

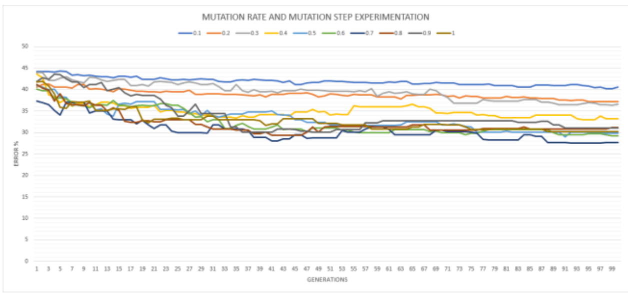


Figure 11: Mutation rate and mutation step increment

Figure 11 showcases that when incrementing the values in unison, we receive similar results. With values 0.1, 0.2 and .03 proving worst, it is clear that mutation has no room for larger mutations, with a much smaller probability of mutation actually occuring.

### 3.3.2 Population

For population, I started at 10, and incremented in jumps of 20 until reaching 130.

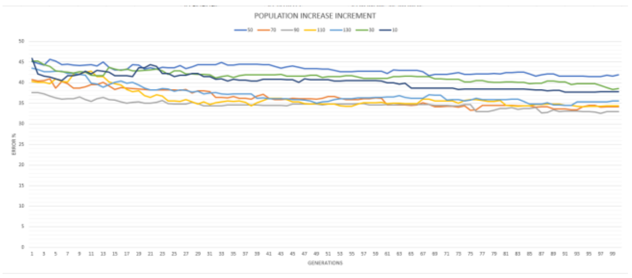


Figure 12: Population increment

A larger populous allows larger potential of superior solutions. Ultimately, a population of 90 proved to be best. This is seemingly validated by values 110 and 70 scoring very similar results and proving the be the second third best results.

### 3.3.3 Hidden Nodes

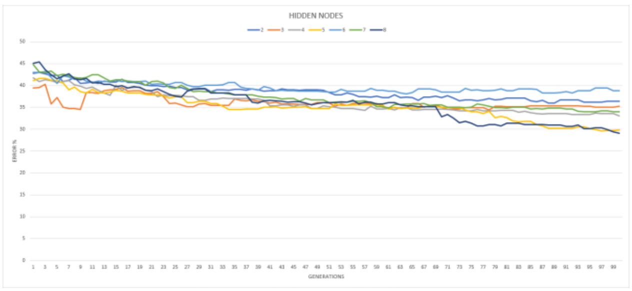


Figure 13: Hidden node increment

Whilst at first glance it may look like 8 hidden nodes performed best, 5 hidden nodes seems to have a better average. More hidden nodes can also increase the computational cost. Adding more hidden nodes also increases the number of parameters in the model, which can make it prone to overfitting. This leads to the model performing well on the training set, but poorly on unseen data.

## 3.4 Experimental testing

For my initial test, I decided to set all tested parameters to their optimum from the systematic testing as a base, this was the outcome:

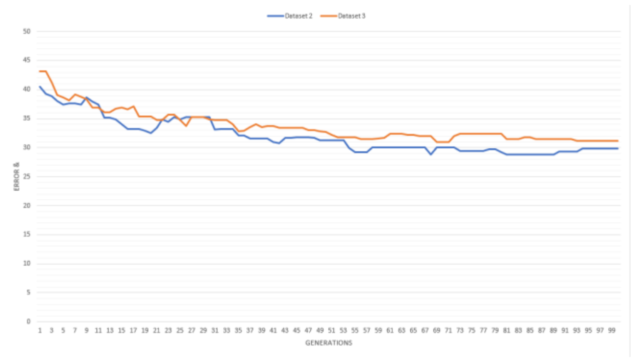


Figure 14: First test

The model achieved an average of 31.94% (averages based on dataset 2), which left a lot of room for improvement. Especially for larger datasets, I felt like 5 hidden nodes was too low given the complexity, so the hidden node count was increased to 6:

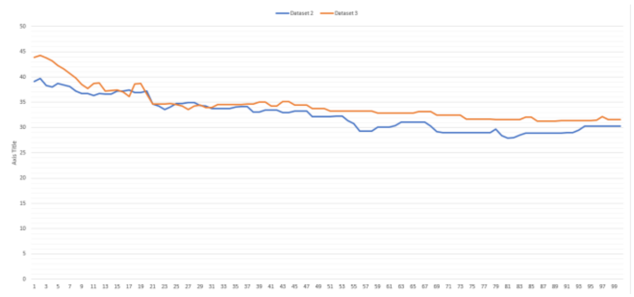


Figure 15: Hidden nodes increased to 6

Sadly, with an average error rate of 32.48%, the result was far too familiar with the previous test.

Whilst the model over many iterations was gradually able to decrease, a larger number of generations would allow for better performance overall. Additionally, The results are too dependent on the initial population's fitness, despite the high mutation rate and mutation step. To compensate, a larger population of 150 was introduced, to increase the potential of a better initial solution, along with 200 generations.

Mutation rate was also reduced from 0.7 to 0.5 to reduce some volatility experienced, meaning members of the population were less likely to be modified.

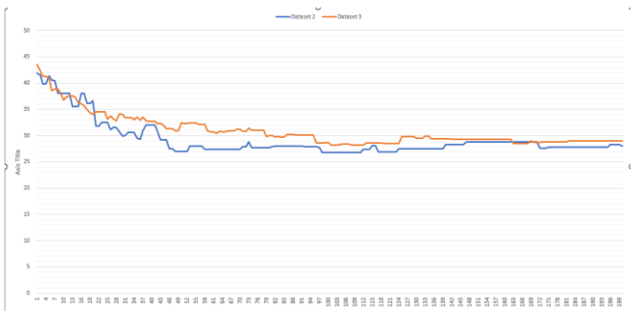


Figure 16: Mutation rate: 0.5

The results returned successful, as the error rate was brought down to 29.26%, proving some effectiveness.

In general, increasing sizes of population and generations can improve the chances of finding a good solution. A larger population allows for more diverse solutions to be explored and can increase the probability of stumbling upon a high-quality solution. Additionally, more generations can help to avoid getting stuck in local optima, as it allows the algorithm to explore a wider range of solutions.

On the other hand, the consequence is the increased computational cost and time. This means there is a trade-off between accuracy and performance of the algorithm. This change increased the average run time from 153 seconds to 602 seconds, an increase of 393%.

Lastly, tanh activation function was employed. Activation functions are an essential component of neural networks. They provide the nonlinearity that allows neural networks to learn complex, nonlinear relationships between the input and output data (Rasamoelina et al., 2020)

The choice of activation function can have a significant impact on the performance of a neural network, such as speed and accuracy. Different activation functions are better suited to different types of data and tasks, so choosing the correct activation function is important.

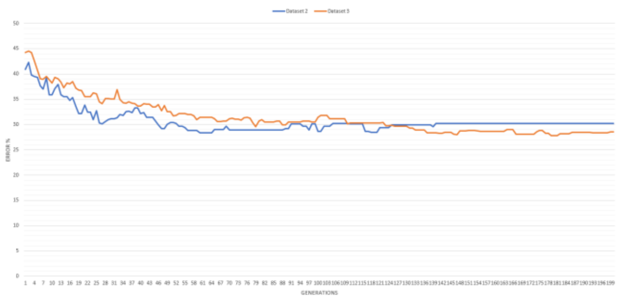


Figure 17: tanh activation function

Whilst tanh performs well, its slightly drops in performance in comparison with sigmoid, with an average error rate of 30.71% (increase of 1.45%).

The similarity in performances comes from the similarity of sigmoid and tanh. Typically, tanh has an advantage of being a steeper curve than sigmoid, but the results are dependant on a multitude of factors, such as the datasets and parameters.

Below are the curves for comparison:

Sigmoid:

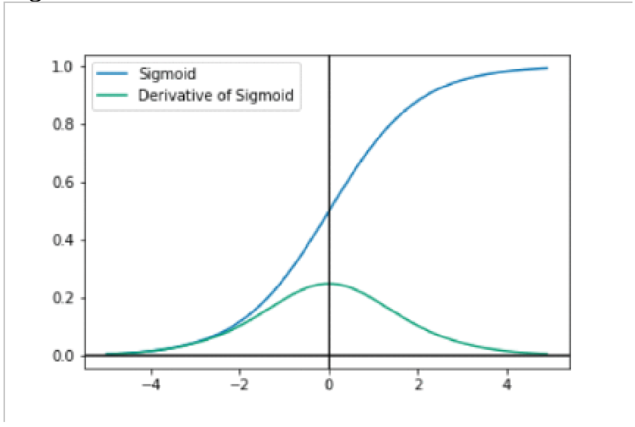


Figure 18: Sigmoid activation function (Rasamoelina et al., 2020)

Tanh:

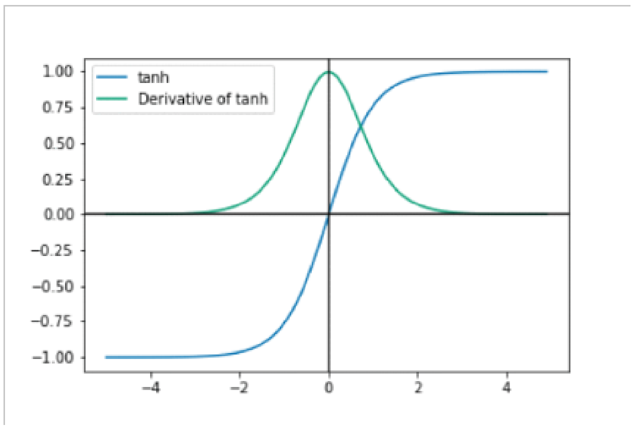


Figure 19: tanh activation function (Rasamoelina et al., 2020)

## 4 CONCLUSIONS

In general, GAs and other metaheuristic algorithms can be used to optimise the parameters of an MLP in order to improve its performance on a given task. The parameters of the GA are used to control the generation and combination of the different MLPs, whilst the parameters of the individual are the potential solutions.

In comparison of GWO and GA, it became evidently clear that GWO generally outperforms GA, in terms of efficiency and accuracy. This being said, there are specific problems which GA would be more suitable for.

Ultimately, in experimentation the error rate has not had much progress, as I would liked to have gotten closer to 20%. Disappointingly, error rate only progressed from 31.93% to 29.26%. However, the values taken from the systematic parameter testing performed better than expected.

The largest change recommended would be to conduct systematic parameter changes with multiple (3-4) parameters at the same time, providing more insightful and valuable results. With this kind of testing, the likelihood of greater solutions is much higher.

## REFERENCES

1. Hatta, N.M., Zain, A.M., Sallehuddin, R., Shayfull, Z., and Yusoff, Y. (2019) Recent studies on optimisation method of Grey Wolf Optimiser (GWO): a review. *Artificial Intelligence Review* [online]. 52 (4), pp.2651-2683. [Accessed 05 December 2022].
2. Kumar, A., Pant, S., and Mangey, R. (2016) System Reliability Optimization Using Grey Wolf Optimizer Algorithm. *Quality and Reliability Engineering International* [online]. 33 (7), pp.1327-1335. [Accessed 01 December 2022].
3. Kumar, M., Husain, D., Upreti, N., and Gupta, D. (2010) Genetic Algorithm: Review and Application (December 1, 2010) [online]. Available at SSRN: https://ssrn.com/abstract=3529843 or http://dx.doi.org/10.2139/ssrn.3529843 [Accessed 02 December 2022].
4. Mirjalili, S. (2015) How effective is the Grey Wolf optimizer in training multi-layer perceptrons. *Applied Intelligence* [online]. 43 (1), pp. 150-161. [Accessed 04 December 2022].
5. Mirjalili, S., Mirjalili, S, M., and Lewis, A. (2014) Grey Wolf Optimizer. *Advances in Engineering Software* [online]. 69, pp.46-61. [Accessed 01 December 2022].
6. Muro, C., Escobedo, R., Spector, L., Coppinger, R.P. (2011) Wolf-pack (Canis lupus) hunting strategies emerge from simple rules in computational simulation. *Behavioural Processes* [online]. 88 (3), pp.192-197. [Accessed 02 December 2022].
7. Rasamoelina, A.D., Adjailia, F., and Sinčák, P. (2020) A review of activation function for artificial neural network. *In 2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)* [online]. pp. 281-286. [Accessed 05 December 2022].

## Source code as an appendix

**Note:** please double click the code the open the full file

```python
import random, copy, matplotlib.pyplot as plt
import numpy as np
import math
import csv
import time

# class to create network
class network:
    def __init__(self):
        self.hweight = [[0 for i in range(inpNodesNum+1)] for j in hidNODES]
        self.oweight = [[0 for i in range(hidNodesNum+1)] for j in outNODES]
        self.error = 0

############################################################
# global variables
############################################################

# population
P = 150
# generations
G = 200
# min gene
MIN = -5.12 #-5.12
# max gene
MAX = 5.12 # 5.12

population = []
offspring = []
bestInd = []

# these are commented out as mutrate and mutstep are passed as arguments
MUTRATE = 0.5
MUTSTEP = 0.7

# lists to plot
trainingPopAverage = []
trainingPopHighest = []
trainingPopLowest = []
bestIndPlot = []

# node quantity, inpNodeNum to be overwritten by importData()
inpNodesNum = 0
hidNodesNum = 6
outNodesnum = 1

# node lists
inpNODES = [0 for _ in range(inpNodesNum)]
inpNodeOut = [0 for _ in range(inpNodesNum)]
```