Introduction

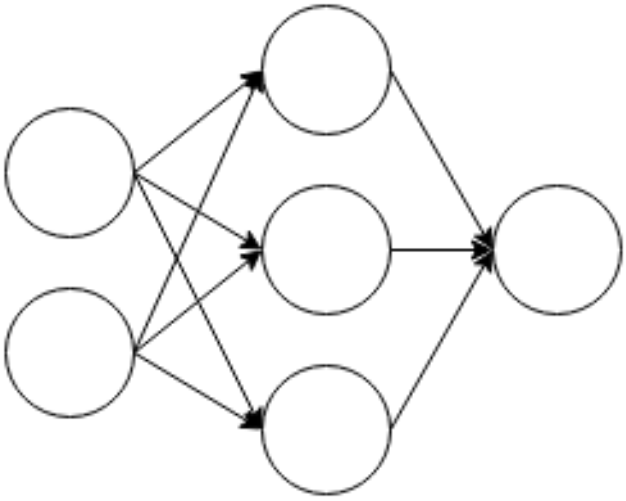
The aim of this project was to create a neural network capable of calculating an output from the received inputs using a genetic algorithm. This is done by creating nodes with weights between them and then multiplying then multiplying the inputs by the values of the weights, with the weights being changed by mutation each generation. A cost function is created to measure how close the calculated output is to the expected output. This is used to measure the fitness in the genetic algorithm. The mutation amount is calculated using the cost, causing changes to get smaller the closer the results get to the expected values. Whether the mutation is added or subtracted from the weights is decided by the difference between the result and the expected values. If the result is higher, the mutation value is subtracted from the weight, otherwise the mutation value is added to the weight. Using the cost fitness function, the best cost, result and weights are stored, with the smallest cost being the assigned as the best.

The neural network is used to replicate different COCO optimisation functions taking only the input values and the expected output.

Setup of Experiments

The genetic algorithm is set up to run for 100’000 generations, with the weights being used as chromosomes. The genetic algorithm does not use crossover, therefore the only changes in the weights occur from mutation. Because of this, the mutation rate has been set at 0.5, giving each weight a 50% chance to mutate to allow faster progression. 100’000 generations are used to ensure that there are enough generations for the results to work closer to the expected result, making it more likely that the best result found is accurate. When selecting the best result, if the new cost is lower than the best cost, then the current weights are assigned as the best weights. If the new cost is not lower than the best cost, then the best weights are assigned as the current weights. This ensures that the weights mutate in a direction that changes the results to be closer to the expected results.

The neural network is designed to store an input that is a tuple and an output that is a single value. It uses a total of 6 nodes, 2 nodes for the input, 1 node for the output and 3 nodes in between, which is referred to as the hidden layer. Only one hidden layer is used, as Universal Approximation Theorem states that a neural network with only one hidden layer is capable of representing the continuous functions used with the right parameters(Cybenko, 1989). These are represented by the circles in the diagram below, with the arrows representing the weights between the nodes. On initialisation the weights are set as random values, and for each generation, weights are a mutated version of the previous weights.



The input values of the first two nodes are multiplied with the weights using matrix multiplication, giving new values for the middle layer nodes. The activation function is then applied to the values of the middle layer nodes. The result is then multiplied by the second set of weights, and then the activation function is applied to the result, giving the final output. The network then calculates the cost using $\sum$ ½ (expected result – actual result)^2. This cost is used to measure fitness in the genetic algorithm, where the lowest value is considered the best. It is also used in the mutation of weights, where a value is calculated using cost multiplied by 0.1. This is to lower the size of the change the closer the results are to the expected results, so as a change will not overshoot the target drastically. The mutation value is either added or subtracted from the weight. This is determined by the results in comparison to the expected results. The total of the results is calculated and is compared to the total of the expected results. If it is greater, the mutation is subtracted and if it is less, the mutation is added.

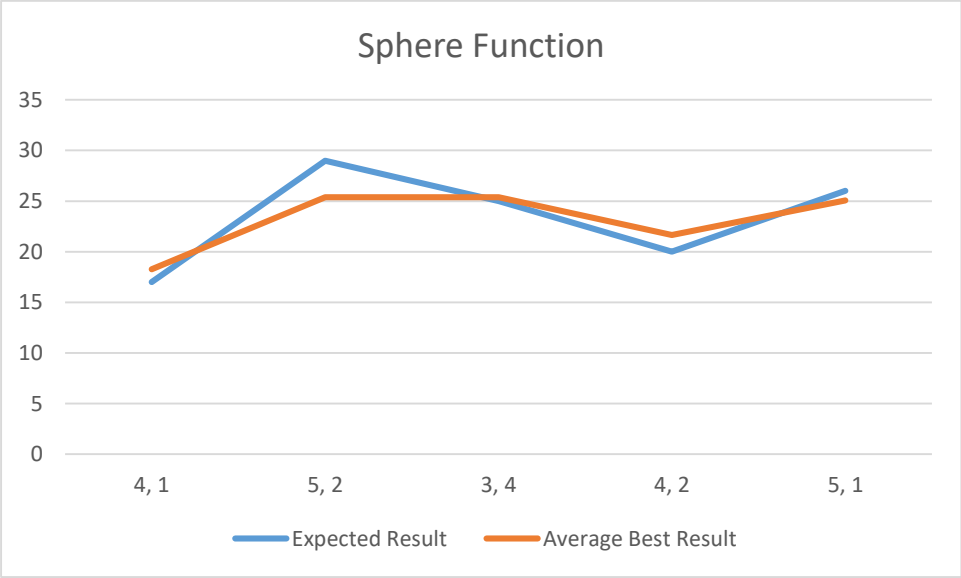
Results

When running the neural network to calculate the sphere function, the expected output is the sum of both input values squared, f(x, y) = $\sum x^2 + y^2$. Using the following inputs, the best found value was calculated 10 times, and the average best result calculated.

Input	Expected Output	1	2	3	4	5	6	7	8	9	10	Average Best Result
4, 1	17	15.85	17.08	22.14	20.21	15.43	17.51	20.14	20.1	17.31	16.84	18.26
5, 2	29	23.98	24.76	26.1	26.81	23.17	25.17	26.73	27.62	24.94	24.63	25.39

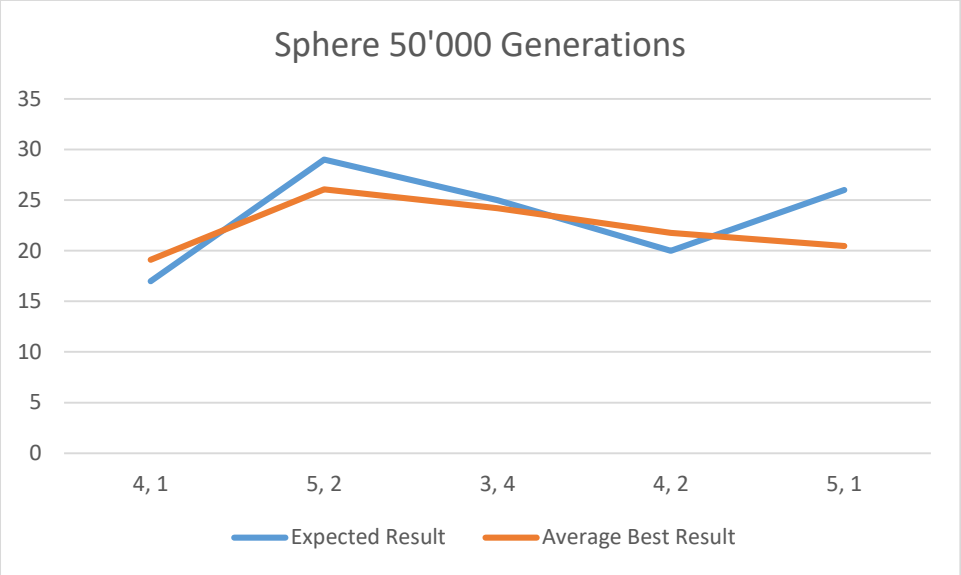
3, 4	25	29.92	27.55	21.19	21.88	28.14	26.99	21.85	20.94	27.27	28.19	25.39
4, 2	20	21.4	21.62	21.67	22.27	20.57	21.83	22.22	22.2	21.17	21.6	21.65
5, 1	26	23.6	25.8	20.89	24.02	28.02	24.56	26.51	28.28	25.2	23.62	25.05

The average best result for each input was then plotted on a chart, along with the expected output.



The results show that the average best result is more generalised compared to the actual result, with the results being closer together than what the expected results should be. Despite this, the results are for the most part close to the expected result, with only the output for 5, 2 being more than 2 away.

When the number of generations is set to 50’000, the average best result is shown to still be close to the expected results, however overall the average best results are less accurate.



Also, even though the average best results are still somewhat similar to the expected results, looking at the individual best results shows a greater deal of variation.

Input	Expected Output	1	2	3	4	5	6	7	8	9	10	Average Best Result
4, 1	17	23.14	21.77	18.8	19.49	19.04	19.87	18.89	13.56	20.23	16.21	19.1
5, 2	29	28.89	28.04	25.91	26.35	25.98	26.61	25.93	24.23	25.33	23.39	26.07
3, 4	25	17.21	19.92	24.5	23.23	23.74	22.62	24.23	31.2	25.05	30.37	24.21
4, 2	20	23.09	22.88	22	21.14	21.95	22.24	21.99	21.77	20.29	20.37	21.77
5, 1	26	2.13	8.75	19.9	12.4	33.87	16.24	19.43	23.59	29.38	38.93	20.46

Discussion

The results displayed are shown to be closer together than the expected values. The reason for this is due to the if statement used to determine whether the mutation value calculated should be added or subtracted. The sum of the expected values and the sum of actual results are calculated. Then the sum of the actual results is compared to sum of expected results. If the expected results are higher then the mutation value is added and if it is less, the mutation value is subtracted. As the sum of the values is used, it does not take into account the difference of specific outputs,

only the total of the outputs. This means that if the values are far apart, it is possible to increase the weights for inputs where it should be decreased to accommodate the other values, and vice versa, causing the results to group more to the middle, with accuracy decreasing the further apart the expected outputs are.

If all the input values and the expected output values are the same, the neural network will be able to get close to the expected result, but will never be entirely accurate. The reason for this is that cost is used to calculate the mutation amount. Due to this, as all values will be close to the expected result without the difference between expected results causing inaccuracy, the cost will become exceptionally small. This means that mutation will also become exceptionally small, causing any changes in the weights to be too small to make any difference.

Conclusion

In conclusion, I feel that the neural network and the genetic algorithm are well designed, but will admittedly suffer a drop in accuracy if expected results are very spread apart. I was unable to utilise COCO in the project due to not being able to get the blackbox code to work successfully with Python. This also limited the functions that I was able to test the neural network with, due to a lack of mathematical understanding of how to utilise them. Therefore only the sphere function was used, with the inputs and expected output coded into the program before the code is run. If I were to redo the code, I would choose to write it using Matlab, which is much simpler to use alongside the COCO optimisation functions.

References

Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function

Github Repository

https://github.com/kieraninglis/bio-inspired-computation/tree/master