Neuroevolutionary and Logic Circuitry Optimization Approach to Solving Binary Classification Problems

By Isaam Rameez

UWE Bristol | University of the West of England

1. INTRODUCTION

This paper is about solving three binary classification datasets using any form of evolutionary intelligence. Dataset 1 and 2 contains binary inputs and binary outputs. Dataset 3 contains floating value inputs and binary outputs.

For my attempt, I will be using a genetic algorithm. In this method, multiple candidates in a solution will be assigned a genome and the candidates will make their decisions based on the genome (Mirjalili, 2019). Keeping the best candidates in a population, allowing their genes to pass on to the next generation and discarding the rest will result in the average accuracy of the population to rise. While this concept should not work in principle it does work in practice.

2. RESEARCH AND EXPERIMENTATION

For this implementation, I have depended mostly on experimentation alone. This applies especially for dataset 1 and 2. For dataset 3 I used a classic neuroevolutionary approach. The reason I chose this method was because it is a fairly simple system to implement. The basic concept behind the neuroevolutionary method is to first create a neural network and implement a method to convert the weights and biases into a genome that the genetic algorithm can use to breed new candidates and form mutations (Floreano and Mattiussi, 2008). From that point, using roulette selection for the genetic algorithm, the neural network was able to relatively easily create a neural network that scores above average.

For dataset 1 and 2 it was a different story. Since the inputs in dataset 1 and 2 were binary rather than floating values, I could not use the neuroevolutionary method, because the neural network expects floating values. So I shifted my focus towards using a rule based genetic algorithm. However that fitness function had an upper limit on its score at 45 - 50% accuracy. After some more testing I had the idea to use logic gates. For my first test I used a single logic gate and it reliably scored a high accuracy for its computing power.

So I proceeded to create a fitness function that nests multiple logic gates and the gene sequence will define what logic gates and how the inputs will be connected to the logic circuitry.

2.1 DATASET 1 & 2

Dataset one and two are very similar. The main difference being their input format. Dataset one has 5 inputs while dataset two has six inputs. Another difference is the number of queries available for training the genetic algorithm. Dataset one has 32 entries while dataset two has 64 entries. However these changes will not be an issue when training the genetic algorithm.

The fitness function uses a circuit to figure out what the candidate will output for the given input. The circuit will look something like figure 1.1

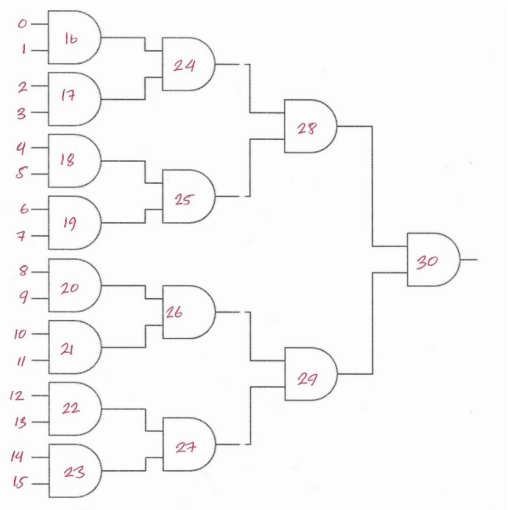

Figure 1.1 : *Reference sheet for logic gate and input placement*

Now let us look at the format of the gene sequence. In code the gene sequence will be a list of integers in string representation. They

will look something like the string in the header of Figure 1.2

The numbers separated by the dashes represent how the logic circuitry is connected to the inputs and the numbers at the end represent the type of logic gates that will be used in the circuit. The numbers in Figure 1.1 show the index of the bit from the gene sequence it represents. Table 1.1 will show what integers represent what logic gate.

Integer	Logic Gate
0	AND Gate
1	OR Gate
2	XOR Gate
3	OR with input 1 inverted
4	OR with input 2 inverted
5	NAND Gate
6	NOR Gate

Figure 1.2: *Integer representation of logic gates*

The genome can be bred normally as any gene sequence. However when mutating, extra attention should be paid to the position of the bit that is being mutated. If the bit defines the logic gate being used, the new value should be within 0 and 6. And if the bit defines how the logic circuit connects to the input, it should be between zero and the length of the number of inputs. If these checks are not performed during mutation it will cause out of bound exceptions.

Using logic gates is pretty effective at training a set with a few numbers of inputs such as dataset 1 and 2. In this scenario a logic circuit can effectively replace a neural network. Figure 1.2 represents a circuit that was evolved after 1109 generations for dataset 2. The circuit yields an accuracy of 90.6%.

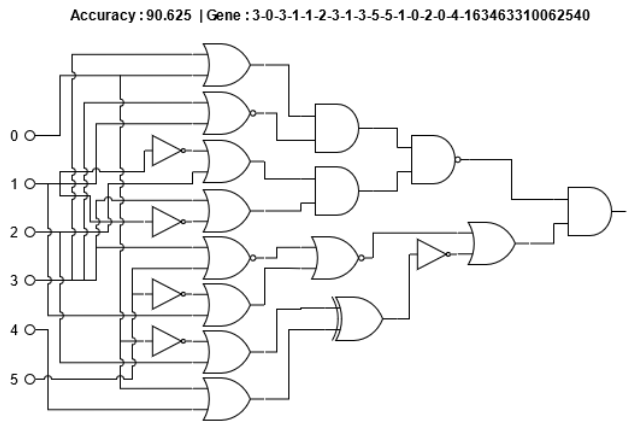

Figure 1.2 : *Logical circuit for dataset 2*

In addition to the high accuracy, logic circuit genetic algorithm simulations also have a very high improvement rate. In most simulations the best candidate is able to score over 80% accuracy before it reaches 1000 generations and in some extreme cases before 100 generations.

Now let us divert our focus from the fitness function and gene sequence to the actual simulation itself. The behaviour of the simulation does not differ much from normal genetic algorithms apart from the special bounds defined by the mutation function itself. Another thing that is different from the usual genetic algorithm process is the act of passing the best performing candidates to the next generation directly. This prevents the score from going down. However it comes at a cost. If we pass the best performing candidate, it might end with the simulation getting stuck in local optimas (Harth, 1997)(which is a small incline in the accuracy curve but not the biggest incline). We can observe the difference between not passing the best performing candidate in Figure A.1 and Figure A.2

Another test I performed was using 2 point crossover. In this test I noticed no notable changes (See Figure A.3) from the baseline chart, apart from the speed at which it reached the global optima. The baseline reached the global optima at around generation 2130 while

the 2 point crossover reached the global optima in 640 generations.

Mutation is an important part of genetic algorithms. If we set a mutation rate that is too high, the simulation will not have a chance to explore some optimas. Mutation rates have a very huge impact on the performance of the final candidate. (See figure A.4)

2.2 DATASET 3

Dataset 3 has six floating value inputs and binary output and has 2000 queries. So this it's the perfect dataset for neuroevolution. In neuroevolution there are 2 important components. The neural network or as I like to call them, the brain of the candidate. Then there is the genetic algorithm component. As we know at this point a genetic algorithm needs a gene sequence for it to work. The neural network will have an output node and will have its value scaled down between 0 and 1. If the value is greater than 0.5 the output will be one, otherwise it will be zero

In the neuroevolutionary model the gene sequence will be the weights from the neural network. We can take them and put them in an array effectively creating a gene sequence (See figure 1.3)

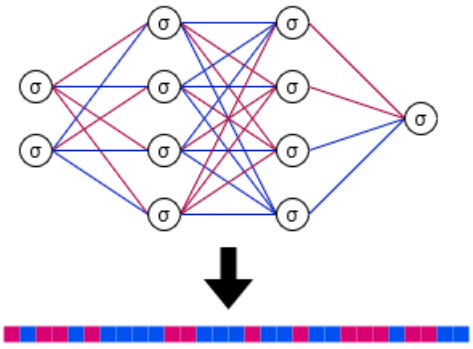

Figure 1.3 : *Converting neural network weights to a gene sequence that can be used in a genetic algorithm*

This model works rather well. It goes beyond 70% accuracy in the first few hundred generations. (See Figure A.5). However from this point it will be harder to get a higher accuracy. So to combat that I have rigged the mutation to adjust the weights and biases more aggressively when improvements keep being found and if improvements are not found for N generations, the mutation magnitude will be reduced, resulting in more fine tuned changes to weights and hopefully, finding more improvements in the population score.

The brain configuration (number of hidden layers and number of nodes per hidden layer) is also very important (Stanley and Miikkulainen, 2002). They have a big impact on the processing speed, number of improvements and the final accuracy. (See figures A.6, A.7 and A.8). The best performing and speed efficient configuration is 7 hidden layers with 2 neurons per hidden layer. But the overall best configuration is 8 hidden layers and 5 neurons per hidden layer.

Another important thing is the activation function (Hu *et al.*, 2015). In my testing the best activation function is rectified linear activation (ReLU) [5] which works way better than sigmoid squishification function.

2. CONCLUSION

For dataset one and two, having a genetic algorithm design a logic gate circuit is an effective way thanks to the nature of the input. Even though the final result is not perfect, it is able to reach a high level of accuracy in minimal time.

The performance of dataset 3 can be improved. Given enough time for training, the genetic algorithm will be able to come up with a neural network with a high degree of accuracy. It also could have been improved by using an alternate genetic algorithm variation. A particularly promising variation is the NEAT (NeuroEvolution of Augmenting Topologies) (Stanley and Miikkulainen, 2002). This is promising because of its ability to morph its neural network structure along with its weights

and biases, resulting in a more accurate recreation of the development of neural pathways in biological life. There was an attempt made to utilize NEAT in the processor for dataset 3. However due to time constraints and the complex nature of implementing NEAT it was not used in the end. Needless to say it was a promising technology that could have really helped in solving dataset 3.

2.1 REFERENCES

•

1. Floreano, D., Dürr, P. and Mattiussi, C. (2008) Neuroevolution: From Architectures to Learning. *Evolutionary Intelligence* [online]. 1, pp. 47-62. [Accessed 02 August 2021].

2. Harth, E. (1997) From Brains to Neural Nets to Brains. *Neural Networks* [online]. 10 (7), pp. 1241-1255. [Accessed 03 August 2021].

3. Hu, X., Niu, P., Wang, J. and Zhang, X. (2015) A Dynamic Rectified Linear Activation Units. *IEEE Access* [online]. 7, pp. 180409-180416. [Accessed 03 August 2021].

4. Mirjalili, S. (2019) Genetic Algorithm. *Evolutionary Algorithms and Neural Networks* [online]. 780, pp. 43-55. [Accessed 06 August 2021].

5. Stanley, K.O. and Miikkulainen, R. (2002) Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* [online]. 10 (2), pp. 99-127. [Accessed 05 August 2021].

Figure A.1 : *The baseline population statistic for dataset 2. Sampled at 5 over a generation of 2500*

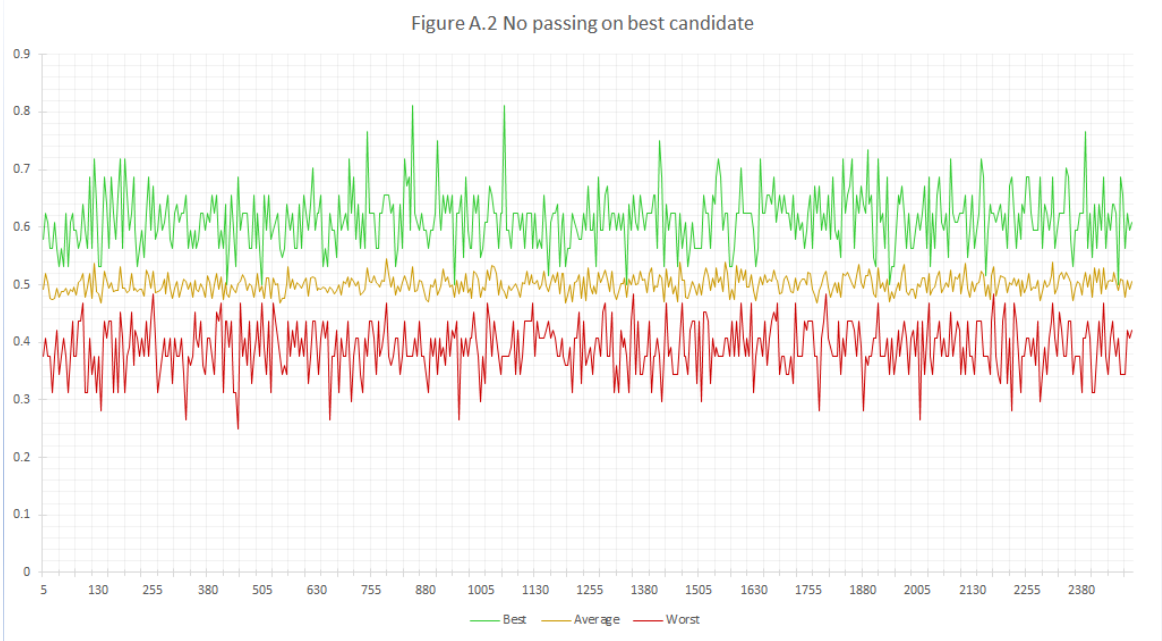

Figure A.2 : *The population statistic for dataset 2 when the best performing candidate is not passed down to the next generation*

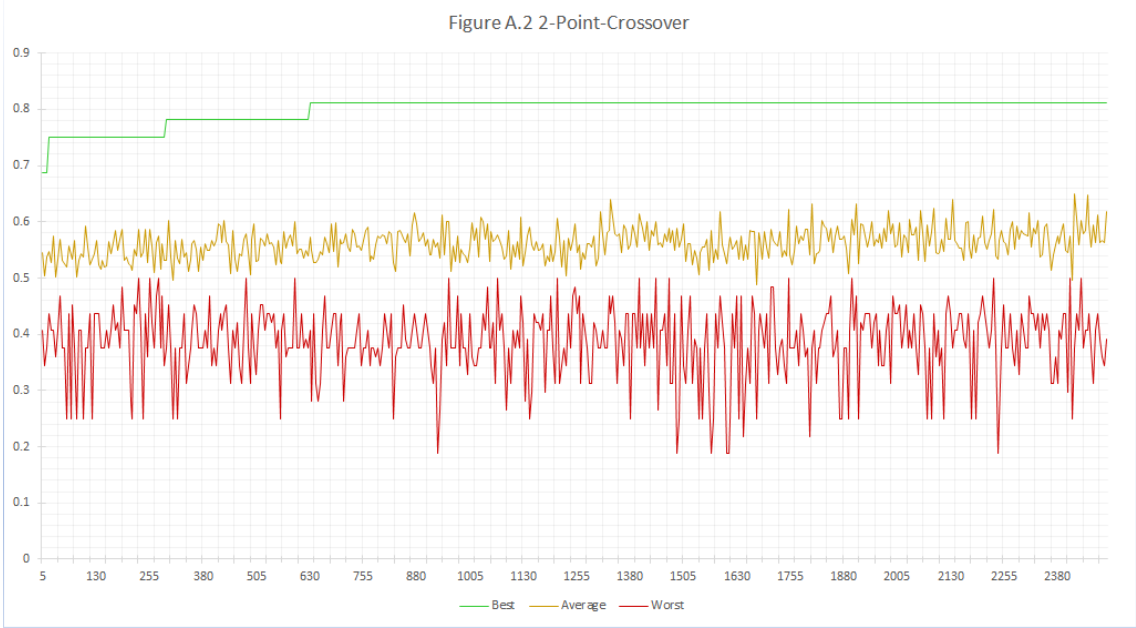

Figure A.3 : *The population statistic for dataset 2, using 2 point crossover rather than single point crossover.*

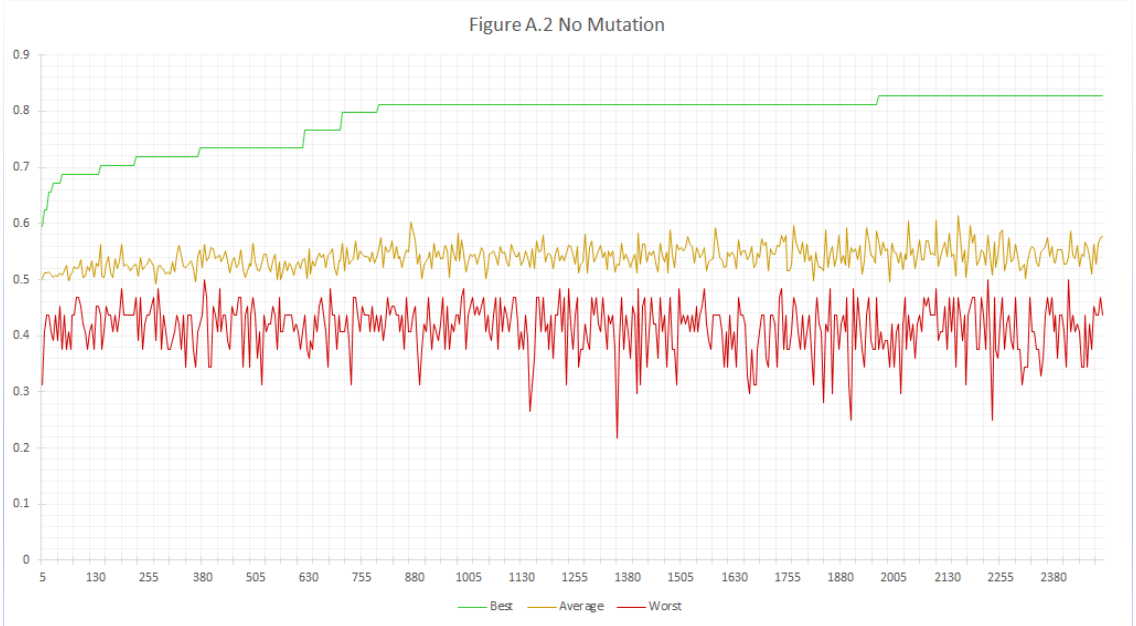

Figure A.4 : *The population statistic for dataset 2 using no mutation at all.*

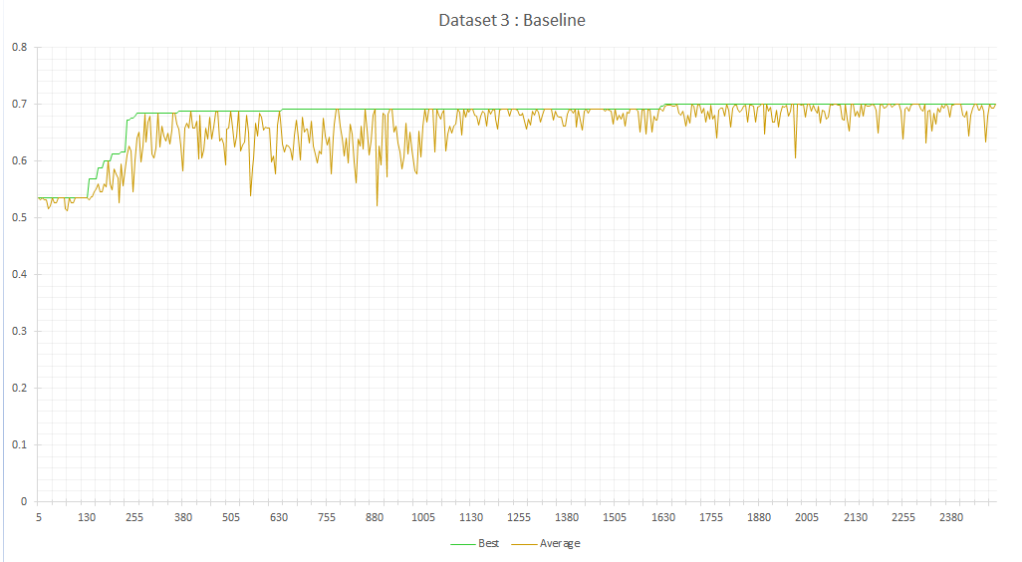

Figure A.5 : *Baseline simulation chart for dataset 3 via neuroevolution*

Number of Layers	Number of Neurons								
	1	2	3	4	5	6	7	8	9
1	5.480495	6.104856	6.874997	7.745145	8.748395	9.31286	10.50395	11.39819	12.42149
2	6.352388	8.144629	9.967143	11.92687	13.76559	16.83572	17.87681	19.23475	20.89753
3	7.371158	10.67627	13.36673	16.60242	21.92741	22.0584	25.07663	28.64179	31.40161
4	8.444771	12.97538	17.61165	21.27492	31.74486	29.87737	33.95108	39.27131	45.13502
5	10.49146	18.16188	25.64295	32.21178	46.76028	46.85278	54.82317	62.43362	73.07433
6	11.50674	20.70472	31.09237	40.61814	54.30663	58.48524	73.22748	83.11819	92.25674
7	12.3235	23.77332	36.66108	50.43007	60.0711	70.18914	82.33669	98.16525	106.8749
8	13.42829	27.94362	44.04063	56.16269	71.3185	84.24044	97.79529	124.6492	128.6896
9	15.62247	35.21226	54.48743	74.05233	94.10726	112.6083	130.1322	154.3507	170.3566

Figure A.6 : *Impact of neurons and layers on processing speed of simulation*

Number of Layers	Number of Neurons								
	1	2	3	4	5	6	7	8	9
1	11	10	13	10	7	2	1	1	1
2	12	9	11	11	13	9	13	8	12
3	15	15	13	14	11	13	13	12	11
4	12	14	11	10	10	13	12	13	13
5	15	16	14	15	13	15	13	13	14
6	15	15	15	14	14	15	14	11	12
7	12	14	15	11	14	15	13	14	13
8	16	10	14	12	17	16	15	15	13
9	11	14	15	12	14	17	12	16	16

Figure A.7 : *Impact of neurons and layers on number of improvements in the the population*

Number of Layers	Number of Neurons								
	1	2	3	4	5	6	7	8	9
1	0.7026667	0.6986666	0.696	0.6853333	0.6813333	0.5706667	0.536	0.536	0.536
2	0.7093332	0.7146667	0.7066667	0.7	0.7066666	0.696	0.6946667	0.688	0.6906667
3	0.7266667	0.7173333	0.72	0.7173333	0.7066666	0.692	0.6826667	0.716	0.692
4	0.7320001	0.728	0.7093334	0.7106667	0.7173333	0.72	0.7026667	0.7266667	0.7
5	0.7373334	0.74	0.7373333	0.728	0.7426667	0.7213333	0.7120001	0.7066667	0.7226667
6	0.7373333	0.7306666	0.7306666	0.704	0.7506666	0.7333333	0.7186667	0.6973333	0.7293333
7	0.7320001	0.7453334	0.7386667	0.7186667	0.7173333	0.736	0.7186666	0.7226667	0.712
8	0.724	0.724	0.7426667	0.7026667	0.752	0.7306666	0.7173333	0.7173334	0.7373333
9	0.7213333	0.7386667	0.7266667	0.728	0.7386667	0.7146667	0.7173333	0.724	0.708

Figure A.8 : *Impact of neurons and layers on the score of the best candidate at the end of the simulation*