Spiking Neuron
IT3708

Nordmoen, Jørgen H.

March 5, 2013

1 Description

For this project I continued using the Evolutionary Algorithm(EA) that I developed for the previous project. Because of the modularity in the previous code not that much had to be changed apart from some minor changes to achieve better performance. As with the last project when needing to adapt the EA to a new problem one has to extend some core classes for which the default behaviour is not appropriate. For this project all the extended code can be found in the file *neuron.py* which contains all the code to simulate the task at hand.

For this project I decided that I wanted to change the crossover function in the genotype which meant that I had to extend the default genotype and create a new class. This is quite easy and only the code for the crossover needed to be changed. I will explain a bit more about why I wanted to change the default crossover below, but for now in *neuron.py* line 179 to 208 one can see the code for the neuron genotype.

1.1 Genotype

Since the phenotype of the spiking neuron relied on some values being in a certain range the code uses a variable amount of bits to represent these variables. The number of bits is user configurable to allow for testing of different sizes. The genotype then is just a bit array with 5xBITS where 5 represent the five variables. Since the genotype is just an array of bits I decided that for this project I would try out Gray Code to interpret those bits. This was done to ensure that mutation would not have the large impact it would have if regular conversion from a bit array to an integer was used. Gray code is another way of interpreting a bit array which tries to minimize the difference between two sequential numbers, e.g. 4 and 5 differ only by one bit. The genotype code does not need to do anything extra in order for this to apply and only in the conversion between genotype and phenotype is this extra bit of information used.

As mentioned above the crossover method was changed for this project. The change enables the crossover to take into consideration each of the five variables. This was done to make inheritance stronger. The change in code means that if we imagine the first variable representing a as the first 10 bits of the array, a child made from two such parents would inherit 50% of a from one parent and another 50% from the other. There is no crossover between different variables which should mean that the inheritance is stronger between parents and child and random bits from a variable will not be put into another variable. To illustrate a bit better I have tried to create an approximate figure in 1.

There is however a rather large problem with this. This way of doing crossover can lead to quite a bit of stagnation, which is clearly present as will be clear in the results. I decided that I wanted to do it this way however because it is quite easy coding wise and I wanted to see how the stagnation would affect the results and how the, stronger, inheritance would affect the results.

1.2 Fitness

To evaluate the fitness of each spike train I implemented the three different distance metrics suggested in the assignment. Since those metrics evaluated

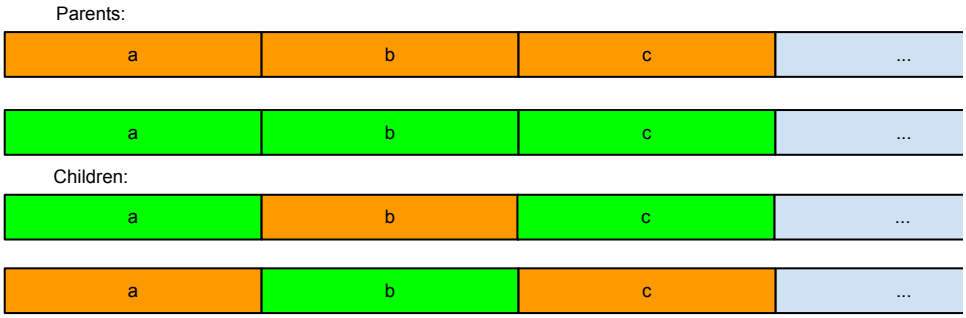


Figure 1: A representation of crossover in the neuron genotype

distance, meaning a smaller distance is better, and the EA code was set up for larger is better the natural choice was to just do 1.0/f(x), where f(x) is one of the three distance metrics. Since both *Spike Time Distance Metric(STDM)* and *Spike Interval Distance Metric(SIDM)* needed the peaks I created one more parent class which could calculate that. In listing 1 we can see the minimal code needed for SIDM[1].

Listing 1: The SIDM implementation

```
class SIDM(NeuronFitness):
    '''Spike Interval Distance Metric fitness'''
    def sub_eval(self, pheno, population):
        spike_pheno = self.calc_spikes(pheno.get_spike_train())
        spike_data = self.spike_data
        s = 0
        n = min(len(spike_pheno), len(spike_data))
        for i in range(1, n):
            s += abs((spike_pheno[i][0]- spike_pheno[i - 1][0])
                -
                    (spike_data[i][0] - spike_data[i - 1][0]))
                        **self.p
        s = s ** (1.0 / self.p)
        s += self.spike_penalty(len(spike_pheno), len(
            spike_data), len(self.data))
        n -= 1
        if n:
            s /= float(n)
        return 1.0 / s
```

2 Results

Below are the results for the different test cases. The layout should be pretty similar so they should be quite easy to follow.

To get some good results I ran all of the test cases several times testing the different variables, these tests are not included here. One problem with this

[1]Note that this code does not include the code necessary for peak calculation

is that I can't know for certain that this is good, because it can easily end up in a local maximum regarding the different variables, e.g. I can test different mutation rates, but find the best one when considered with all of the others.

2.1 Test Case 1

In this test we are testing STDM with the test case *izzy-train1.dat*.

2.1.1 EA Parameters

Listing 2: Command-line to replicate the results

```
python src/main.py 350 5 50 --mutation=0.1 --cross_rate=0.8
    full_generational
tournament --k=5 --elite=3 stdm convert_neuron neuron_plot
    training\
data/izzy-train1.dat --bits_per_num=35 --e=0.02
```

Generations: 350, Population size: 50, Mutation rate: 0.1, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Tournament selection, k=5 and e=0.02, Elitism: 3, Bits per number: 35

2.1.2 End Results

As we can see in figure 2b the end result is quite good for the STDM. Although for this test I had to up the bits per number to quite high and had to run with a higher population than I had initially wanted[2]. The end result is good and there is nothing really special about this. From the last project I found that *Tournament selection* was the better of the four mechanisms and that is reflected here, all the other did worse under the same conditions.

Looking at figure 2a we can see that the analysis about the new crossover was quite right. If we look at the standard deviation we can see that it is very low to begin with and then after some time, goes up. This is signs of stagnation, but because of the high mutation rate we get a good end result.

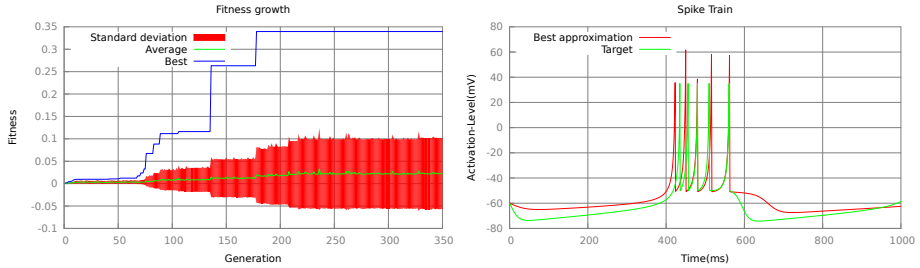
a	b	c	d	k	Fitness
0.0323	0.0441	-50.924	1.1294	0.0412	0.34

Table 1: The neuron variables which gave the best solution for test case 1

2.2 Test Case 2

In this test we are testing SIDM with the test case *izzy-train1.dat*.

[2]Mostly do to performance concerns



(a) Fitness progression for test case 1 (b) Spike train comparison for test case 1

Figure 2: Test case 1 result

a	b	c	d	k	Fitness
0.0010	0.0207	-55.985	5.2820	0.0497	1.33

Table 2: The neuron variables which gave the best solution for test case 2

2.2.1 EA Parameters

Listing 3: Command-line to replicate the results

```
python src/main.py 350 5 50 --mutation=0.05 --cross_rate=0.8
    full_generational
tournament --k=5 --elite=3 sidm convert_neuron neuron_plot
    training\
data/izzy-train1.dat --bits_per_num=30 --e=0.02
```

Generations: 350, Population size: 50, Mutation rate: 0.05, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Tournament selection, k=5, e=0.02, Elitism: 3, Bits per number: 30

2.2.2 End Results

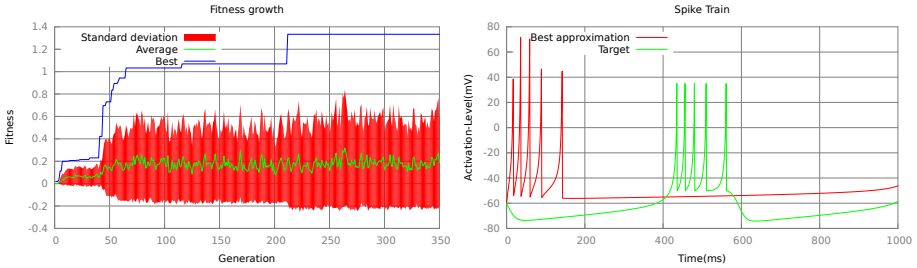
In figure 3b we can see the spike train for this test case, and the story is quite different from the last case, but it is quite expected considering what SIDM actually does. Since it is mostly conserned with the distance between spikes this technique is very good at getting the distance right, but the placement in the spike train is often wrong.

The fitness(see figure 3a) for this case is much closer to the previous case than the spike train. We can see that in the start the standard deviation is low, but then after some, lucky, mutations it grows and we get a more diverse population.

2.3 Test Case 3

In this test we are testing WDM with the test case *izzy-train1.dat*.

2.3.1 EA Parameters



(a) Fitness progression for test case 2 (b) Spike train comparison for test case 2

Figure 3: Test case 2 result

a	b	c	d	k	Fitness
0.0246	0.1645	-49.704	1.9947	0.0401	13.5

Table 3: The neuron variables which gave the best solution for test case 3

Listing 4: Command-line to replicate the results

```
python src/main.py 500 5 50 --mutation=0.05 --cross_rate=0.7
    full_generational
sigma --elite=3 wdm convert_neuron neuron_plot training\ data/
    izzy-train1.dat
--bits_per_num=20
```

Generations: 500, Population size: 50, Mutation rate: 0.05, Crossover rate: 0.7, Protocol: Full generational replacement, Mechanism: Sigma, Elitism: 3, Bits per number: 20

2.3.2 End Results

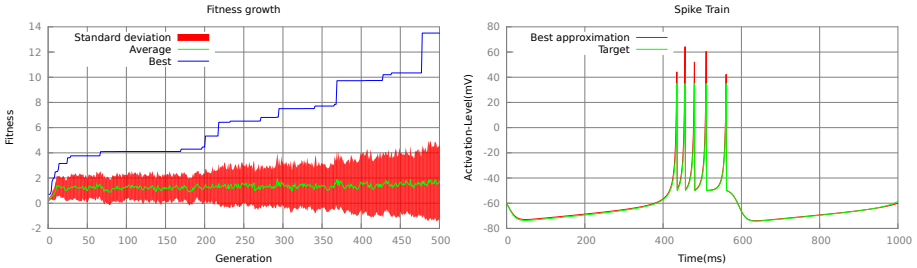
The result for this one is very good, actually a lot better than I had initially though, but the result is very much a result of tweaking to get every thing to work at optimum. The strange thing this time was that Sigma scaling was better than Tournament selection. The number of generations also went up in this test, but that can be lowered or increased depending on what sort of result one wants. I tried increasing and got a slightly better result, which indicates that this probably could become perfect eventually, but I had to draw the line somewhere. If we look at figure 4b we can see that this is almost perfect.

The fitness plot this time around is quite familiar and nothing that special. Since this was run for a longer time than the two previous tests the standard deviation is a bit ”flatter”, but we can still see traces of early stagnation before some mutations kick in.

2.4 Test Case 4

In this test we are testing STDM with the test case *izzy-train2.dat*.

2.4.1 EA Parameters



(a) Fitness progression for test case 3 (b) Spike train comparison for test case 3

Figure 4: Test case 3 result

a	b	c	d	k	Fitness
0.0229	0.2902	-47.672	5.7082	0.0470	0.537

Table 4: The neuron variables which gave the best solution for test case 4

Listing 5: Command-line to replicate the results

```
python src/main.py 500 5 50 --mutation=0.1 --cross_rate=0.9
    full_generational
sigma --elite=3 stdm convert_neuron neuron_plot training\
data/izzy-train2.dat --bits_per_num=40
```

Generations: 500, Population size: 50, Mutation rate: 0.1, Crossover rate: 0.9, Protocol: Full generational replacement, Mechanism: Sigma scaling, Elitism: 3, Bits per number: 3

2.4.2 End Results

In this we can again see that STDM is quite good a placing the spike about where they should be, but it can't seem to place them perfect nor get the spacing completely correct.

The fitness is again quite representative for my earlier concerns, but high mutation combined with some elitism and crossover actually produce decent results. We can see that the standard deviation increases along with the best becoming better. This is again one which could probably run forever, but the later increases was not large enough for me to warrant the longer run times.

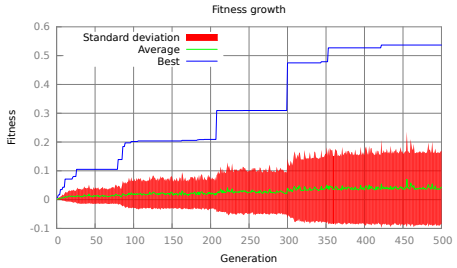
2.5 Test Case 5

In this test we are testing SIDM with the test case *izzy-train2.dat*.

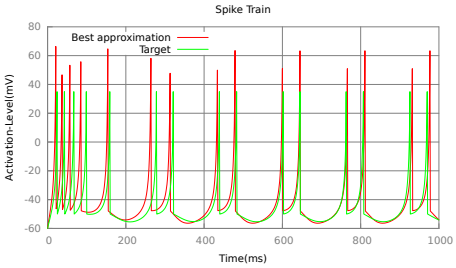
2.5.1 EA Parameters

Listing 6: Command-line to replicate the results

```
python src/main.py 250 5 20 --mutation=0.1 --cross_rate=0.8
    full_generational
```



(a) Fitness progression for test case 4



(b) Spike train comparison for test case 4

Figure 5: Test case 4 result

a	b	c	d	k	Fitness
0.0313	0.1220	-43.496	7.9364	0.0502	0.899

Table 5: The neuron variables which gave the best solution for test case 5

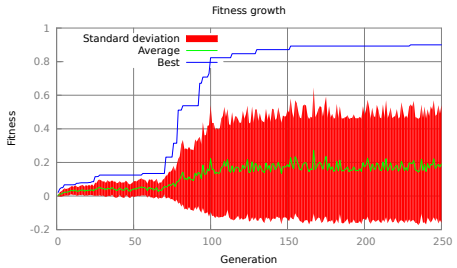
```
tournament --k=5 --e=0.02 --elite=3 sidm convert_neuron
    neuron_plot training\
data/izzy-train2.dat --bits_per_num=40
```

Generations: 250, Population size: 20, Mutation rate: 0.1, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Tournament, k=5, e=0.02, Elitism: 3, Bits per number: 40

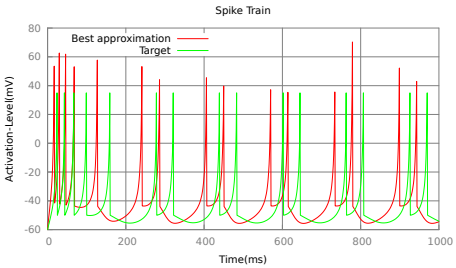
2.5.2 End Results

This was one of those cases which probably ended up in a local maximum, since changing any variable slightly would plummet the fitness far down. I chose to include it because it was so good and in many cases no other case, so far, have done as good with SIDM.

This case was again quite similar to the others in regard to fitness, it starts out low and then increases. Again this is a test case which could have ran longer, but trying to double the number of generations only gave an increase of about 0.02 which I felt was not good enough.



(a) Fitness progression for test case 5



(b) Spike train comparison for test case 5

Figure 6: Test case 5 result

a	b	c	d	k	Fitness
0.0118	0.0538	-55.717	8.2982	0.0569	2.10

Table 6: The neuron variables which gave the best solution for test case 6

2.6 Test Case 6

In this test we are testing WDM with the test case *izzy-train2.dat*.

2.6.1 EA Parameters

Listing 7: Command-line to replicate the results

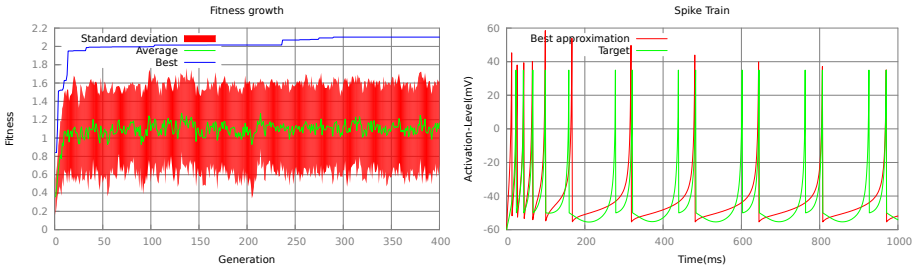
```
python src/main.py 400 5 50 --mutation=0.05 --cross_rate=0.9
    full_generational
sigma --elite=3 wdm convert_neuron neuron_plot training\
data/izzy-train2.dat --bits_per_num=40
```

Generations: 400, Population size: 50, Mutation rate: 0.05, Crossover rate: 0.9, Protocol: Full generational replacement, Mechanism: Sigma, Elitism: 3, Bits per number: 40

2.6.2 End Results

This test case seemed quite hard for WDM. No good results no matter the settings tried which is very strange. The choice of going with Sigma scaling for this was quite random as I tried several times with both this and Tournament selection which came very close to each other and in the end I went with Sigma.

The fitness plot for this is also very strange, which again does not help explain the results.



(a) Fitness progression for test case 6 (b) Spike train comparison for test case 6

Figure 7: Test case 6 result

2.7 Test Case 7

In this test we are testing STDM with the test case *izzy-train3.dat*.

a	b	c	d	k	Fitness
0.0726	0.0872	-42.132	3.8164	0.0409	0.73

Table 7: The neuron variables which gave the best solution for test case 7

2.7.1 EA Parameters

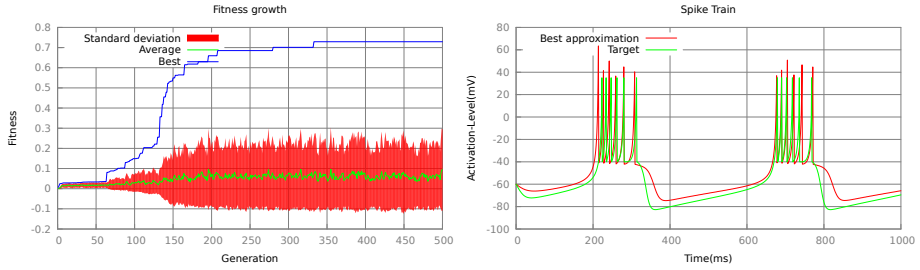
Listing 8: Command-line to replicate the results

```
python src/main.py 500 5 100 --mutation=0.05 --cross_rate=0.9
    full_generational
tournament --k=5 --e=0.02 --elite=3 stdm convert_neuron
    neuron_plot training\
data/izzy-train3.dat --bits_per_num=25
```

Generations: 500, Population size: 100, Mutation rate: 0.05, Crossover rate: 0.9, Protocol: Full generational replacement, Mechanism: Tournament selection, k=5, e=0.02, Elitism: 3, Bits per number: 25

2.7.2 End Results

This was another one where STDM did quite good, but again could not match up the space between spikes completely. The fitness is quite good, and could possibly be better when run for much longer. Again the selection mechanism did not influence all that much and Tournament was selected a little at random after some testing.



(a) Fitness progression for test case 7 (b) Spike train comparison for test case 7

Figure 8: Test case 7 result

2.8 Test Case 8

In this test we are testing SIDM with the test case *izzy-train3.dat*.

2.8.1 EA Parameters

Listing 9: Command-line to replicate the results

a	b	c	d	k	Fitness
0.0789	0.0985	-30.949	9.2191	0.0412	0.976

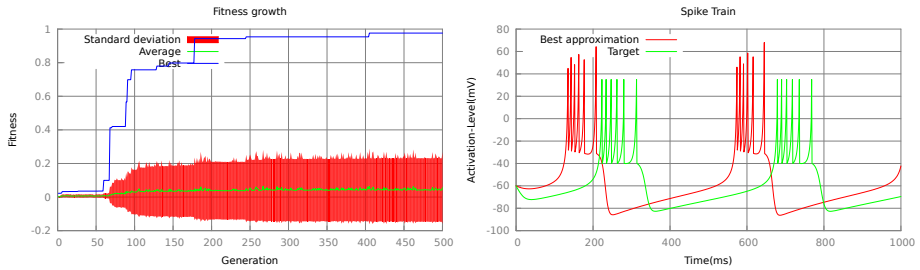
Table 8: The neuron variables which gave the best solution for test case 8

```
python src/main.py 500 5 75 --mutation=0.1 --cross_rate=0.8
    full_generational
sigma --elite=3 sidm convert_neuron neuron_plot training\
data/izzy-train3.dat --bits_per_num=30
```

Generations: 500, Population size: 75, Mutation rate: 0.1, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Sigma, Elitism: 3, Bits per number: 30

2.8.2 End Results

In this case we can see very much the same story as before. The population is somewhat homogeneous in the beginning, before it shoots up and becomes quite a bit better. The end result is quite good, but again we see that SIDM struggles with placing the spikes at precisely the spot.



(a) Fitness progression for test case 8 (b) Spike train comparison for test case 8

Figure 9: Test case 8 result

2.9 Test Case 9

In this test we are testing WDM with the test case *izzy-train3.dat*.

2.9.1 EA Parameters

Listing 10: Command-line to replicate the results

```
python src/main.py 250 5 100 --mutation=0.1 --cross_rate=0.8
    full_generational
tournament --k=5 --e=0.02 --elite=3 wdm convert_neuron
    neuron_plot training\
data/izzy-train3.dat --bits_per_num=20
```

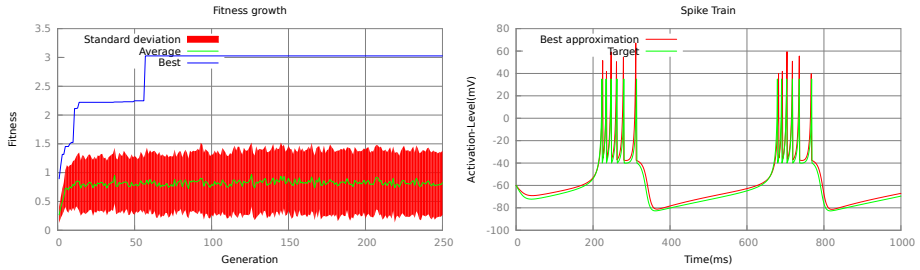
a	b	c	d	k	Fitness
0.0838	0.1136	-37.405	6.2404	0.0404	3.026

Table 9: The neuron variables which gave the best solution for test case 9

Generations: 250, Population size: 100, Mutation rate: 0.1, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Tournament selection, k=5, e=0.02, Elitism: 3, Bits per number: 20

2.9.2 End Results

This time WDM was again back up to from, it performed very well and the fitness grew very good in the short number of generations ran. For this the strange thing is that a low number of bits was very good combined with a large population. The large population is not that strange, but the low number of bits being good is a bit strange.



(a) Fitness progression for test case 9 (b) Spike train comparison for test case 9

Figure 10: Test case 9 result

2.10 Test Case 10

In this test we are testing STDM with the test case *izzy-train4.dat*.

2.10.1 EA Parameters

Listing 11: Command-line to replicate the results

```
python src/main.py 250 5 100 --mutation=0.1 --cross_rate=0.8
    full_generational
tournament --k=5 --e=0.02 --elite=3 stdm convert_neuron
    neuron_plot training\
data/izzy-train4.dat --bits_per_num=20
```

Generations: 250, Population size: 100, Mutation rate: 0.1, Crossover rate: 0.8, Protocol: Full generational replacement, Mechanism: Tournament selection k=5, e=0.02, Elitism: 3, Bits per number: 20

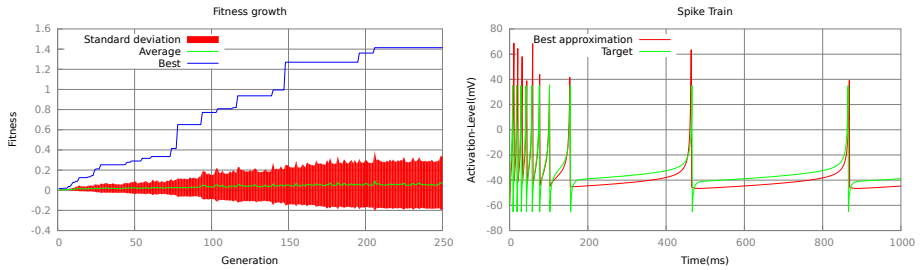
a	b	c	d	k	Fitness
0.0027	0.2811	-45.365	7.7698	0.0667	1.414

Table 10: The neuron variables which gave the best solution for test case 10

2.10.2 End Results

With this one I was quite luck with one of the first tries. The large population size managed to create a good result right away. This is probably one of the major mistakes I have had in previous runs and not take a large population. From this run, which could just be lucky, it seems a bit like number of bits is not as important as the population size.

The fitness graph also illustrate very well the inherent stagnation which could have been a problem if not for the large mutation used. We can easily see where there have been a favorable mutation which has improved the best individual.



(a) Fitness progression for test case 10 (b) Spike train comparison for test case 10

Figure 11: Test case 10 result

2.11 Test Case 11

In this test we are testing SIDM with the test case *izzy-train4.dat*.

2.11.1 EA Parameters

Listing 12: Command-line to replicate the results

```
python src/main.py 250 5 100 --mutation=0.1 --cross_rate=0.9
    full_generational
tournament --k=5 --e=0.02 --elite=3 sidm convert_neuron
    neuron_plot training\
data/izzy-train4.dat --bits_per_num=25
```

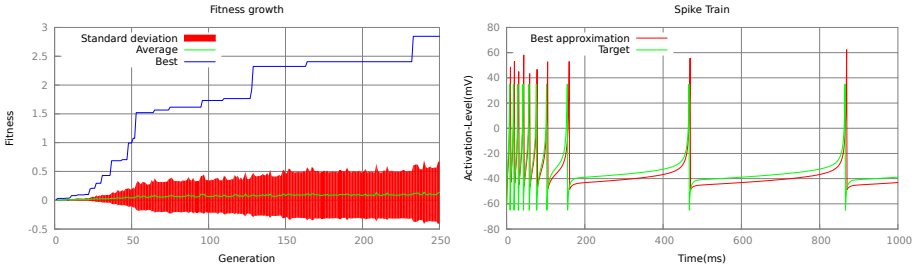
Generations: 250, Population size: 100, Mutation rate: 0.1, Crossover rate: 0.9, Protocol: Full generational replacement, Mechanism: Tournament selection, Elitism: 3, Bits per number: 25

a	b	c	d	k	Fitness
0.0029	0.2074	-49.351	8.4007	0.0699	2.846

Table 11: The neuron variables which gave the best solution for test case 11

2.11.2 End Results

This is another quite good result for SIDM. The reason for this, again, might be random, but the technique behind SIDM also is a good fit for this spike train because there are some small gap spikes to start with and then there are some larger gaps which SIDM can "use" to perfect the distance and also placement, since there are not a whole lot of room to be displaced in.



(a) Fitness progression for test case 11 (b) Spike train comparison for test case 11

Figure 12: Test case 11 result

2.12 Test Case 12

In this test we are testing WDM with the test case *izzy-train4.dat*.

2.12.1 EA Parameters

Listing 13: Command-line to replicate the results

```
python src/main.py 500 5 100 --mutation=0.05 --cross_rate=0.8
    full_generational
tournament --k=5 --e=0.02 --elite=3 wdm convert_neuron
    neuron_plot training\
data/izzy-train4.dat --bits_per_num=35
```

Generations: 500, Population size: 100, Mutation rate: 0.05, Crossover rate: 0.08, Protocol: Full generational replacement, Mechanism: Tournament selection, Elitism: 3, Bits per number: 35

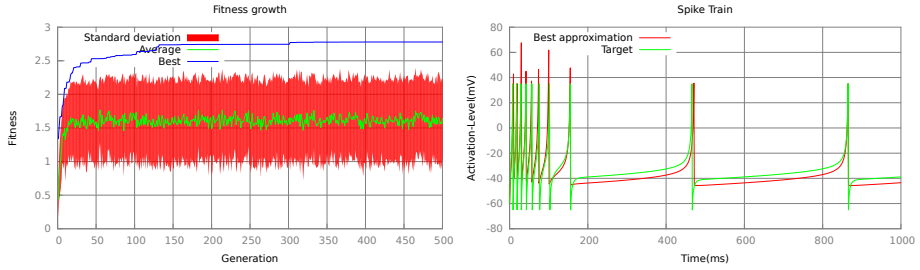
2.12.2 End Results

This was again another good fit for WDM. WDM can get quite close to the target, but compared to the other two metrics it needs more generations and higher number of bits. The number of bits does make sense since WDM is

a	b	c	d	k	Fitness
0.0034	0.0115	-45.429	8.2475	0.0689	2.779

Table 12: The neuron variables which gave the best solution for test case 12

more dependant on perfect fits than the two others. More generations is a bit stranger and if we look at the graph it is not that many improvements beyond the 300 mark.



(a) Fitness progression for test case 12 (b) Spike train comparison for test case 12

Figure 13: Test case 12 result

3 Discussion

The transformation between genotype and phenotype is quite straight forward in this project. The transformation is direct where some number of sequential bits in the bit array determines the value in the phenotype. The genotype is fixed length which makes the conversion quite easy. The only problem which could arise is the problem of mutation in regular binary to integer conversion. For this project that challenge was solved using Gray coding, as described above, which to some extent solves the problem. This step does however introduce a small step between binary and integer, but it would still be classified as direct. Since the conversion is between binary and integers the underlying model would be data oriented as compared to program oriented.

3.1 Practical Implications

The practical implications of this project is very interesting as it actually can be used to do some real simulation and not just trivial problems. The current implementation might not be robust enough to be published, but with some more work it could actually be used.

For use in the real world, this tool could be used to match up spike trains from the brain to test out either new models or trying to find the proper configuration in already tried model, very much like the 12 cases tested previously. One example which I can think of off the top of my head, is the use in brain computer interfaces(BCI). In BCI one want to match up spike trains to certain thoughts in the brain, this problem is of course a regression/classification problem, but testing the models of what a person could be thinking could be tested

with this tool and even developing the model to optimize the classification algorithm could use this tool.

A computational neuroscientist could of course also use this tool in the same way we have used it, trying to match a model of spiking neurons to actual spiking neurons. From my understanding of what they would do they could use this tool to model neurons trying to approximate real spike trains and use those models further to try and understand the underlying processes in the brain, but also have a way of testing how one neuron or a group of neurons impact their neighbours.

3.2 Other Uses

There are several other uses for a more general version of this tool. Any place where there are some results and a model trying to explain those results a similar tool to this project could be used. For instance in complex mathematics or physics we could have a result from the real world and trying to come up with a model explaining it. If that model contains several variables which can be mapped in the same fashion as in this project we can often use EA to solve the problem. Often times such a complex model might also be to big to efficiently approach with other methods, but with an EA we can get close enough very fast. Another domain is in financial analysis where the complexities of the system is to big to model in real time, but we would still like to know what is going on. For such a problem we could, after the fact, try and do model fitting and try to explain anomalies or maybe explain in a bit more detail why the curve is the way it is to try and help.

To extend on the example of financial analysis we can think about events like Flash crash which for a long time remained a mystery. The reason was ultimately a combination of several factors, but key among them was automatic traders. These events are extremely hard to predict beforehand and are also very hard to analyse after the fact. This tool could potentially help to analyse such event by helping to verify models. To an extent this tool could also be used to try and understand electronic traders which are often company secrets and very hard to predict. With knowledge about automatic traders and how they act the large stock exchanges could help prevent events such as the one linked to above. In figure 14 there is a rudimentary illustration of the above example.

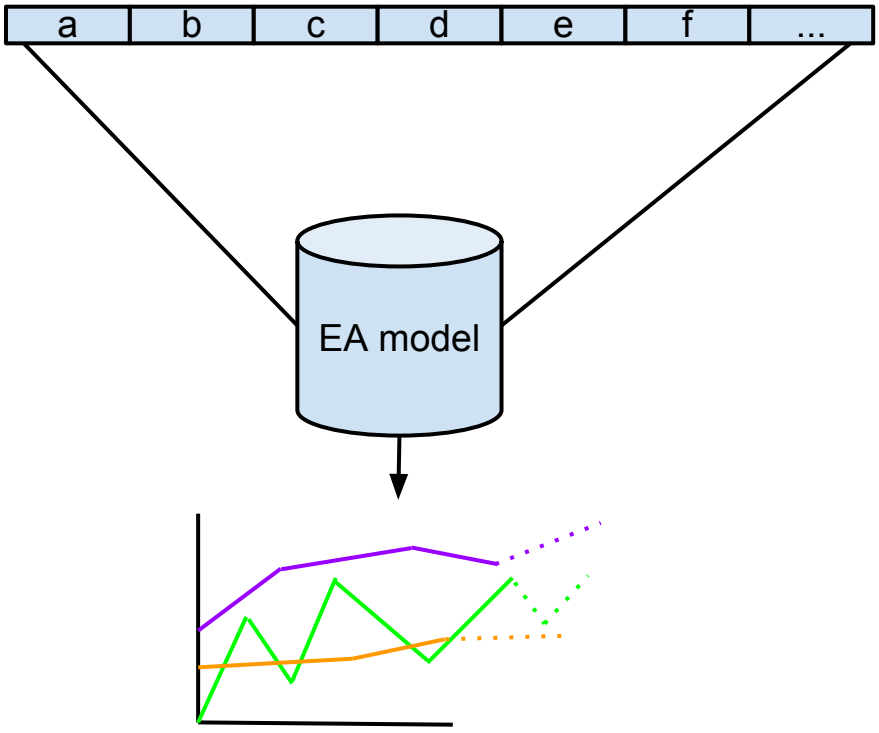


Figure 14: An illustration of financial analysis using Evolutionary algorithms