Spiking Neuron
IT3708

Nordmoen, Jørgen H.

March 2, 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 *neuronpy* 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 listing 1 one can see the code.

Listing 1: The default genotype extended to support a different kind of crossover more appropriate for the problem.

```
class NeuroGenome(genome.Genome):
    '''Use X bits per variable in the neuron, this means that
        in order to
    ensure that crossover passes on proper genes we need to re
        implement
    crossover. This genome will also just do 1 point crossover
        per variable.'''
    def __init__(self, val, cross_rate, mute_rate, convert_func
        , bits_per_num=20):
        super(NeuroGenome, self).__init__(val, 0.0, cross_rate,
             mute_rate,
                convert_func)
        assert len(self) == 5*bits_per_num, 'The length is not
            correct for a neuron genome'
        self.bits = bits_per_num

    def crossover(self, other):
        assert other, 'Other can\'t be nothing'
        my_val = self.get_value()
        other_val = other.get_value()
        my_cpy = self.get_value()
        other_cpy = other.get_value()
        half_bits = self.bits / 2
        if random() < self.cross_rate:
            for i in range(0, self.len, self.bits):
                my_val[i:i+half_bits] = other_cpy[i:i+half_bits
                    ]
        if random() < self.cross_rate:
            for i in range(0, self.len, self.bits):
                other_val[i+half_bits:i+self.bits] = my_cpy[i+
                    half_bits:i+self.bits]
        return (NeuroGenome(my_val, self.cover_rate,
            self.mute_rate, self.convert_func, self.bits),
                NeuroGenome(other_val,
                  self.cross_rate, self.mute_rate, self.
                      convert_func, self.bits))
    def __repr__(self):
        return "NeuroGenome({!r}, {!r}, {!r}, {})".format(self.
            val,
                self.cross_rate, self.mute_rate, self.
                    convert_func)
```

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. 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(listed in 1) 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.

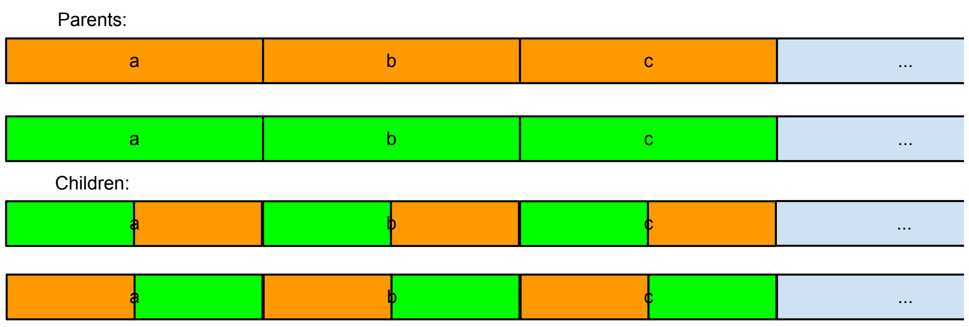

Figure 1: A representation of crossover in the neuron

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 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 2 we can see the minimal code needed for SIDM[1].

Listing 2: The SIDM implementation

```python
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
```

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