Homework #3 Answers

Chongye Feng

In [1]:
```
# importing libraries
import numba
import numpy as np
import pandas as pd
from pylab import *
from mpl_toolkits.mplot3d import axes3d
from scipy.optimize import minimize
```

In [2]:
```
# setting the random seed
np.random.seed(0)
```

Q1:

a:

For $f(x) > 27$:

Encoding A

	Solution	Fitness	Vector
	3	30	1000
	4	31	0010
	5	30	0001
	Schema		* 0 **
	Order		1
	Length	=2-2	0

Encoding B

Solution	Fitness	Vector
3	30	1101
4	31	1011
5	30	1111
Schema		1 ** 1
Order		2
Length	=4-1	3

The well-encoded schema is the one with short length and low order. So I will choose Encoding A.

b:

Encoding A

Solution	Fitness	Vector
6	27	0000
15	-90	1111
1	22	0011
10	-5	0101
9	6	1100
0	15	1011

c:

Encoding A

Parents	Children	New?	Child-Solution	Child-Fitness
0000	0111	Y	12	-33
1111	1000	Y	3	30
0011	0101	N		
0101	0011	N		
1100	1011	N		
1011	1100	N		

The population increased by two more individuals. Since the fitness of a population is the sum of the fitnesses of the individual population members. The net fitness of a population changes by (-33) + (30) = -3, so the fitness of a population is decreasing by 3. The best solution would be 3, with a fitness of 30.

d:

Encoding A

Parents	Children	New?	Child-Solution	Child-Fitness
0000	0010	Y	4	31
1111	1101	Y	13	-50
0111	0101	N		
0011	0001	Y	5	30
0101	0111	N		
1000	1010	Y	7	22
1100	1110	Y	14	-69
1011	1001	Y	2	27

The population increased by six more individuals. The net fitness of a population changes by (31) + (-50) + (30) + (22) + (-69) + (27) = -9, so the fitness of a population is decreasing by 9 from the population of 1c. The best solution would be 4, with a fitness of 31.

e:

Encoding A

Solution	Fitness	Replacing	New-Vector
0	15	N	1011
1	22	N	0011
2	27	N	1001
3	30	N	1000
4	31	N	0010
5	30	N	0001
6	27	N	0000
7	22	N	1010
9	6	N	1100
10	-5	N	0101
12	-33	N	0111
13	-50	N	1101
14	-69	N	1110
15	-90	Y	0010 (Cloned)

Encoding A

Parents (paired)	Children	New?	Solution	Fitness
0010	0110	Y	11	-18
1110	1010	N		
0010	0100	Y	8	15
1101	1011	N		
1000	1110	N		
0111	0001	N		
0001	0101	N		
0101	0001	N		
1001	1101	N		
1100	1000	N		
0000	0010	N		
1011	1001	N		
0011	0011	N		
1010	1010	N		

The population increased by two more individuals. The net fitness of a population changes by (-18) + (15) = -3, so the fitness of a population is decreasing by 3 from the population of 1d. The best solution would still be 4, with a fitness of 31.

f:

There are 16 individuals of the population right now, and the total amount of possible solutions is 16 = 2^4. So, there would be no more new solution would be generated in this step.

Below is the example of the cross-over operation between the fittest (0010) and the least-fit (1110, ps. where 1111 is replaced by cloning the fittest):

Encoding A	1f Cross Over Example	
Parents (paired)	Children	New?
001-0	111-0	N
111-0	001-0	N

The population increased by no more individuals. The net fitness of a population would not be changed. The best solution would still be 4, with a fitness of 31.

g:

I think the encoding "type-A" is adequate of the solution space. 1) The encoding accuately represent all possible solutions in the solution space. 2) New solutions are easily generated by the mutation and cross-over operations from limited parents population. 3) After 1c, 1d and 1e operations, all of the possible solutions are generated from just 6 original parental population. 4) The global maximum, 4, was found by just two generations.

Q2:

In [3]:
```
# activation function and its derivative
def tanh(x):
    return np.tanh(x);

def tanh_prime(x):
    return 1-np.tanh(x)**2;
```

In [4]:
```
class NN():
    def __init__(self, architecture, learning_rate, activation=tanh):
        # initializing the model
        self.arch = architecture
        self.activation = activation
        self.learning_rate = learning_rate
        self.depth = len(architecture)
        self.init_weight()

    def init_weight(self):
        self.weights = []
        self.biases = []
        for l in range(self.depth - 1):
```

```
            prev_layer_num = self.arch[l]
            current_layer_num = self.arch[l+1]
            # initialize weights randomly with values between 0 and 1.
            self.weights.append(np.random.rand(current_layer_num, prev
            self.biases.append(np.zeros(current_layer_num))

    def calc_error(self, y, activation_grad=tanh_prime):
        self.errors = []
        # calculate the error of the output layer
        delta = (self.a_s[-1] - y) * activation_grad(self.z_s[-1])
        self.errors.append(delta)
        # propagate the error backwards to previous layers
        for l in range(self.depth - 2, 0, -1):
            delta = (self.weights[l].T @ delta) * activation_grad(self
            self.errors.append(delta)
        # reverse the errors to match the layer order
        self.errors = self.errors[::-1]

    def calc_grad(self):
        self.weights_grad = []
        self.biases_grad = []
        # calculate the gradients for each layer
        for l in range(self.depth - 1):
            weight_grad = np.inner(nn.errors[l],nn.a_s[l])
            bias_grad = self.errors[l]
            self.weights_grad.append(weight_grad)
            self.biases_grad.append(bias_grad)

    def back_prop(self):
        # update the weights and biases using the gradients and learni
        for l in range(self.depth - 1):
            self.weights[l] = self.weights[l] - self.learning_rate * s
            self.biases[l] = self.biases[l] - self.learning_rate * sel

    def feed_forward(self, x):
        self.z_s = []
        self.a_s = [x]
        for l in range(self.depth - 1):
            z_l = self.weights[l] @ self.a_s[l] + self.biases[l]
            a_l = self.activation(z_l)
            self.z_s.append(z_l)
            self.a_s.append(a_l)

    def fit(self, x, y):
        self.feed_forward(x)
        self.calc_error(y)
        self.calc_grad()
        self.back_prop()

    def predict(self, x):
```

```
        self.feed_forward(x)
        return self.a_s[-1]
```

In [5]:
```
nn = NN(architecture=[6, 2, 2], learning_rate=0.1)
```

a:

In [6]:
```
nn.init_weight()
```

In [7]:
```
nn.weights
```

Out[7]:
```
[array([[0.0202184 , 0.83261985, 0.77815675, 0.87001215, 0.97861834,
         0.79915856],
        [0.46147936, 0.78052918, 0.11827443, 0.63992102, 0.14335329,
         0.94466892]]),
 array([[0.52184832, 0.41466194],
        [0.26455561, 0.77423369]])]
```

The weights of each layer were initialized.

b:

In [8]:
```
x = np.array([[-1, 1], [-1, -1], [0, 0], [0, 0], [0, 0],[1, -1]])
```

In [9]:
```
nn.predict(x)
```

Out[9]:
```
array([[-0.14670542, -0.68325685],
       [-0.23345948, -0.71824085]])
```

It looks like the first prediction is closed to (-0.2, -0.7) which is a coil. But it is just an initial guess, and our model needs to be fitted.

c:

In [10]:
```
observe = np.array([-1,-1])
```

In [11]:
```
nn.calc_error(observe)
```

In [12]:
```
nn.errors
```

Out[12]:
```
[array([[0.6256418 , 0.01830307],
        [0.83163061, 0.04811582]]),
 array([[0.83492957, 0.1688748 ],
        [0.72476151, 0.13640808]])]
```

It shows the error here.

d:

In [13]:
```
nn.fit(x,observe)
```

In [14]:
```
nn.weights
```

Out[14]:
```
[array([[0.08095227, 0.89701433, 0.77815675, 0.87001215, 0.97861834,
         0.73842469],
        [0.53983084, 0.86850382, 0.11827443, 0.63992102, 0.14335329,
         0.86631744]]),
 array([[0.54191963, 0.45317065],
        [0.28103817, 0.80679346]])]
```

In [15]:
```
nn.predict(x)
```

Out[15]:
```
array([[-0.45721908, -0.71159509],
       [-0.5387804 , -0.73672515]])
```

After one fitting, the prediction is close to (-0.5,-0.7), which is closer to our observation. It seems that our model has been trained properly.