Chem277B: Machine Learning Algorithms

Homework assignment #3: Meta-heuristic algorithms

In [1]:
```
import numpy as np
from numpy import linalg as LA
import time
from pylab import *
import matplotlib.pyplot as plt
import math
import scipy
import pandas as pd
import numba
```

1. Genetic Algorithms.

(a) The solutions for encodings A and B, as well as their schema are listed below:

Encoding A Solutions	Fitness	Vector	Schema
x = 3	30	1000	$*0**$
x = 4	31	0010	$*0**$
x = 5	30	0001	$*0**$

Encoding B Solutions	Fitness	Vector	Schema
x = 3	30	1101	$1**1$
x = 4	31	1011	$1**1$
x = 5	30	1111	$1**1$

The length and order of the two schema are shown below:

Encoding	Schema	Length	Order
Encoding A	$*0**$	0	1
Encoding B	$1**1$	3	2

Given that the principle of schema should be lower length and lower order, I will choose encoding A.

(b) The listed solutions and fitness and their grouping are below:

Candidate Solutions	fitness	Encoding	Pairing Group
x = 10	-5	0101	Group 2
x = 1	22	0011	Group 2
x = 15	-90	1111	Group 1 (least fit)
x = 6	27	0000	Group 1 (fittest)
x = 0	15	1011	Group 3
x = 9	6	1100	Group 3

(c) After cross-over operation, the new encodings are as follow:

Group	Candidate Solutions	Original fitness	Encoding	After cross-over	New fitness	New solutions
1	x = 15	-90	1|111	*1000*	30	x = 3
1	x = 6	27	0|000	*0111*	-33	x = 12
2	x = 10	-5	0|101	*0011*	22	x = 1
2	x = 1	22	0|011	*0101*	-5	x = 10
3	x = 0	15	1|011	*1100*	6	x = 9
3	x = 9	6	1|100	*1011*	15	x = 0
---	Sum fitness	-25	---	---	35	---

For group 1 with the fittest and least fit pairs, the original fitness are -90 and 27. Now the fitness after cross-over operation are 30 and -33. I consider the fitness increased.

For group 2 the fitness didn't change. They remain as 22 and -5.

For group 3 the fitness didn't change. They remain as 6 and 15.

Group 1 still has the highest fitness score. Total fitness of the new group after cross-over operation increased from -25 to 35. The population increased as a whole. The best solution is encoding 1000 with a fitness of 30. Now the solution x = 3.

(d) After mutation, the new encodings, the new solutions and their corresponding fitness are shown below:

Group	Solutions after cross-over	fitness	Encoding	Encoding after mutation	New solutions	New fitness
1	x = 3	30	10[0]0	10*1*0	x = 7	22

Group	Solutions after cross-over	fitness	Encoding	Encoding after mutation	New solutions	New fitness
1	x = 12	-33	01[1]1	0101	x = 10	-5
2	x = 1	22	00[1]1	0001	x = 5	30
2	x = 10	-5	01[0]1	0111	x = 12	-33
3	x = 9	6	11[0]0	1110	x = 14	-69
3	x = 0	15	10[1]1	1001	x = 2	27
---	Sum fitness	35	---	---	---	-28

It seems mutation didn't increase the total fitness of the population. We found a solution of x = 5 with fitness of 30, same fitness as before mutation. It's not a better solution in my opinion.

(e) The least fit member after step d is in group 3, entry 5, x = 14 that leads to a fitness of -69.

The best fit member is in group 2, entry 3, x = 5 that leads to a fitness of 30.

We will perform cloning then followed by 2-point cross-over as shown below, with grouping remaining the same since there are now two fittest entries and no instruction to regroup:

Group	Solutions after cloning	fitness	Encoding	Encoding after 2-point cross-over	New solutions	New fitness
1	x = 7	22	1[01]0	1100	x = 9	6
1	x = 10	-5	0[10]1	0011	x = 1	22
2	x = 5	30	0[00]1	0111	x = 12	-33
2	x = 12	-33	0[11]1	0001	x = 5	30
3	x = 14 -> x = 5	-69 -> 30	1110 -> 0[00]1	0001	x = 5	30
3	x = 2	27	1[00]1	1001	x = 2	27
---	Sum fitness	71	---	---	---	82

Now we can see the new solution is clearly better. We now have a total fitness of 82. There are two fittest solutions still, both of which are x = 5 and fitness of 30. We don't have a better solution than before in this case though.

(f) We will perform cloning then followed by cross-over as shown below:

Group	Solutions after cloning	fitness	Encoding	Encoding after cross-over	New solutions	New fitness
1	x = 9	6	[110]|0	0010	x = 4	31
1	x = 1	22	[001]|1	1101	x = 13	-50

Group	Solutions after cloning	fitness	Encoding	Encoding after cross-over	New solutions	New fitness
2	x = 12 -> x = 5	-33 -> 30	0111 -> [000]|1	0001	x = 5	30
2	x = 5	30	[000]|1	0001	x = 5	30
3	x = 5	30	[000]|1	1001	x = 2	27
3	x = 2	27	[100]|1	0001	x = 5	30
---	Sum fitness	82 -> 145	---	---	---	98

The result is good. We not only increased the total fitness, but also found the best solution to the fitting equation.

(g) I think the encoding of the solution space is adequate, in the sense that it was able to find the best solution of the fitting space with a genetic algorithm style evolution, in a finite number of steps. The algorithm converged in short time.

2. Artificial Neural Networks.

We first define an artificial neural network class as follow.

In [103…

```
class NN():
    def __init__(self, architecture, learning_rate, activation):
        # initialize the model
        self.architecture = architecture     # Structure of NN, in HW case, [6, 2, 2] list
        self.activation = activation         # Activation function, defined separately outside the class
        self.learning_rate = learning_rate   # Learning rate lambda is a constant
        self.layer = len(self.architecture)  # Number of layers of the NN

    def init_weight(self):
        self.weights = []     # list of matrices for each layer
        self.biases = []      # list of derivatives
        for i in range(self.layer - 1):
            prev_layer_num = self.architecture[i]
            current_layer_num = self.architecture[i+1]
            # The biases and weights for the network are initialized randomly with normal distribution
            self.weights.append(np.random.random((current_layer_num, prev_layer_num)))
            self.biases.append(np.random.random(current_layer_num))

    def feed_forward(self, a):
        self.z_s = []
        self.a_s = [a]
        for i in range(self.layer - 1):
            z_i = self.weights[i].dot(self.a_s[i]) + self.biases[i]
            a_i = self.activation(z_i)

            # list of numpy arrays
            self.z_s.append(z_i)
```

```python
            self.a_s.append(a_i)
        return self.a_s[-1]

    def calc_error(self, y, activation_grad):
        self.error = self.a_s[-1] - y
        self.weights_grad = [0] * (self.layer - 1)
        self.biases_grad = [0] * (self.layer - 1)
        return self.error * activation_grad(y)

    def calc_grad(self, y, activation_grad):
        self.biases_grad = [np.zeros(b.shape) for b in self.biases]
        self.weights_grad = [np.zeros(w.shape) for w in self.weights]
        sp = activation_grad(y)
        delta = self.error * sp[-1]
        self.weights_grad[-1] = np.outer(delta, self.a_s[-2])
        self.biases_grad[-1] = delta
        for i in range(2, self.layer):
            delta = np.dot(self.weights[-i+1].T, delta) * sp[-i]
            self.weights_grad[-i] = np.outer(delta, self.a_s[-i-1])
            self.biases_grad[-i] = delta

    def back_prop(self, activation_grad):
        # calculate the gradient of the output layer
        delta = self.calc_error(y, activation_grad) * self.activation(self.z_s[-1])
        self.weights_grad[-1] = np.outer(delta, self.a_s[-2])
        self.biases_grad[-1] = delta
        # calculate the gradients of the hidden layers, from the back to the front
        for l in range(2, self.layer):
            z = self.z_s[-l]
            sp = self.activation(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            self.weights_grad[-l] = np.outer(delta, self.a_s[-l-1])
            self.biases_grad[-l] = delta
        # update the weights and biases
        for i in range(self.layer - 1):
            self.weights[i] -= self.learning_rate * self.weights_grad[i]
            self.biases[i] -= self.learning_rate * self.biases_grad[i]

    def fit(self, x, y, activation_grad):                   # Iterate the NN
        self.feed_forward(x)
        self.errors = self.calc_error(y, activation_grad)
        self.calc_grad(y, activation_grad)
        self.back_prop(activation_grad)

    def predict(self, x):
        return self.feed_forward(x)
```

(a) After initializing, the weights and biases of the neural network are as follow:

In [115…
```
np.random.seed(0)
network = NN([6,2,2], 0.1, np.tanh)
network.init_weight()
network.weights
```

Out[115]:
```
[array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ,
         0.64589411],
        [0.43758721, 0.891773  , 0.96366276, 0.38344152, 0.79172504,
         0.52889492]]),
 array([[0.07103606, 0.0871293 ],
        [0.0202184 , 0.83261985]])]
```

(b) With an input of [-1, 1, -1, -1, 1, -1], the fitting is [0.64299999 0.79969983], and the predicted secondary structure would be hydrophobic helix

In [116…
```
x = [-1, 1, -1, -1, 1, -1]
print("Initialized prediction:", network.predict(x))
```

```
Initialized prediction: [0.64299999 0.79969983]
```

(c) The calculated errors for the hidden layer nodes are calculated as follow: [0.6305039 0.66240387]. The prediction after fitting once is: [0.47417151 0.53893351].

In [124…
```
y = [-1, -1]
def tanh_grad(x):
    return 1 - np.tanh(x)**2
network.fit(x, y, tanh_grad)
errors = network.calc_error(y, tanh_grad)
print("Error in nodes", errors)
print("Prediction after fitting once: ", network.predict(x))
```

```
Error in nodes [0.6305039  0.66240387]
Prediction after fitting once:  [0.47417151 0.53893351]
```

(d) The general formula for weight updates is as follow:

w' = w - learning_rate * activation_grad