CHEM277B Homework 3

Trevor Oldham

February 2023

Problem 1

(A)

List all good solutions with f(x) > 27 for each encoding:

Encoding A:

Solution 3: 1000

Solution 4: 0010

Solution 5: 0001

Schema: (*0* | *)

Length: 0

Order: 1

Encoding B:

Solution 3: 1101

Solution 4: 1011

Solution 5: 1111

Schema: (1 * * 1)

Length: 3

Order: 2

We want to choose a schema with short length and low order because those choices are less likely to be disrupted by genetic operations like splicing so we should choose Encoding A as the most appropriate encoding.

(B)

Drawing from the pool of candidates we can make pairs by pairing the most fit with the least fit, the second most fit with the second least fit, and so on.

Solution 10: 0101 - fitness = -5

Solution 1: 0011 - fitness = 22

Solution 15: 1111 - fitness = -90

Solution 6: 0000 - fitness = 27

Solution 0: 1011 - fitness = 15

Solution 9: 1100 - fitness = 6

Pairs

(6, 15) - (0000, 1111)

(1, 10) - (0011, 0101)

(0, 9) - (1011, 1100)

(C)

Create a cross-over point after the first element of the encoding and swap the elements after that point.

Pairs

(6, 15) - (0000, 1111) - After cross-over (0111, 1000)

(1, 10) - (0011, 0101) - After cross-over (0101, 0011)

(0, 9) - (1011, 1100) - After cross-over (1100, 1011)

The first pair results in two new solutions after the cross-over. The second pair and the third pair do not create new solutions. The two new solutions are 0111 (x=12) and 1000 (x=3). They have fitness f(12)=-33 and f(3)=30. The sum total of the fitness for the new generation is larger than the fitness before, and x=3 becomes the most fit member. The current population is now (12, 3, 10, 1, 9, 0) with a fitness of 35 total.

(D)

Mutate the third element of the encodings for each string in the new population.

Solution 12: 0111 - After mutation 0101 - f(1) = 15

Solution 3: 1000 - After mutation 1010 - f(7) = 22

Solution 10: 0011 - After mutation 0001 - f(5) = 30

Solution 1: 0101 - After mutation 0111 - f(12) = -33

Solution 9: 1100 - After mutation 1110 - f(14) = -69

Solution 0: 1011 - After mutation 1001 - f(2) = 27

There are new solutions (7, 5, 14, 2) and we no longer have (3, 10, 9, 6, 0). The total fitness of the new population is now -8 meaning the new population is less fit than the previous population.

(E)

Eliminate the least fit member and replace with clone of the most fit member. Then create pairs and do two-point crossover on the two inner characters in the strings.

Replace Solution 14 with Solution 5 (0001)

Pairs

(5, 1) - (0001, 0101) - After cross-over (0101, 0001) - (1, 5)

(5, 12) - (0001, 0111) - After cross-over (0111, 0001) - (12, 5)

(2, 7) - (1001, 1010) - After cross-over (1011, 1000) - (0, 3)

Fitness

f(0) = 15

f(1) = 22

f(3) = 30

f(5) = 30

f(5) = 30

f(12) = -33

Total = 94

The new population contains (0, 1, 3, 5, 5, 12) and has a new fitness of 94 which is better than the previous generation and the generation before that. The current best solution is x=3 or x=5 with a fitness of 30.

(F)

Eliminate the least fit member and replace with a clone of the most fit member. Then create pairs and do a cross-over of the first three elements in the string encoding.

Replace Solution 12 with Solution 3 (1000)

Pairs

(5, 0) - (0001, 1011) - After crossover (1011, 1001) - (0, 2)

(3, 1) - (1000, 0011) - After crossover (0010, 1001) - (4, 2)

(3, 5) - (1000, 0001) - After crossover (0000, 1001) - (6, 2)

Fitness

f(0) = 15

f(2) = 27

f(2) = 27

f(2) = 27

f(4) = 31

f(6) = 27

Total = 154

The new population is (0, 2, 2, 2, 4, 6) and the new population fitness is 154 which is the best generation yet. The most fit member is x=4 with a fitness of 31 which is the correct answer for maximizing the function given. The values seem to be converging to a range around 4 which contains the solutions closest to the true solution. This generation could be different depending on the pairings that were chosen.

(G)

I think the encoding of the solution space was a good choice because the numbers seem to converge around the true solution at x=4. I imagine that the value of 4 would begin to appear more and more as subsquent iterations of cross-over and mutation are performed, because the Encoding A has a schema which defines the most fit members having a short length and a low order, which has been proven to guarantee an exponential increase in the most fit solutions. We could also replace the least fit member with the most fit member which would increase the occurrence of solution 4 as time goes on.

Problem 2

In [6]:
```
import numpy as np
```

In [412…
```
class NN():
    def __init__(self, architecture, learning_rate, activation_function):
        #initialize the model
        self.arch = architecture
        self.activation = activation_function
        self.learning_rate = learning_rate
        self.depth = len(self.arch)

    def init_weights(self):
        self.weights = []
        self.biases = []
        for l in range(self.depth - 1):
            prev_layer_number = self.arch[l]
            current_layer_number = self.arch[l + 1]
            #tip: generate random matrix for weights rather than zeros for h
            self.weights.append(np.random.rand(current_layer_number, prev_la
            self.biases.append(np.random.rand(current_layer_number))
        #print("weights after init")
        #print(self.weights)

    def feed_forward(self, x):
        self.z_s = []
        self.a_s = [x]
        for l in range(self.depth - 1):
            z_l = np.dot(self.weights[l], (self.a_s[-1])) + self.biases[l]
            a_l = self.activation(z_l)
            self.z_s.append(z_l)
            self.a_s.append(a_l)
        #print("activations")
        #print(self.a_s)
        return self.a_s[-1]
```

```
    def calc_error(self, y, activation_grad):
        #todo
        self.errors = [0] * (self.depth - 1)
        self.errors[1] = (self.a_s[-1] - y) *  activation_grad(self.z_s[1])
        self.errors[0] = np.multiply(np.transpose(self.weights[1]).dot(self.
        #print('Errors after calc_error')
        #print(self.errors)

    def calc_grad(self):
        self.biases_grad = self.errors
        self.weights_grad = []
        self.weights_grad.append(np.dot(self.a_s[1], self.errors[0]))
        self.weights_grad.append(np.dot(self.a_s[2], self.errors[1]))

    def back_prop(self):
        for l in range(self.depth - 1):
            self.weights[l] = self.weights[l] - self.learning_rate * self.we
            self.biases[l] = self.biases[l] - self.learning_rate * self.bias

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

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

In [425…

```
def tan_h(x):
    return np.tanh(x)

def tan_h_grad(x):
    return 1 - tan_h(x)**2

np.random.seed(0)

nn = NN([6, 2, 2], learning_rate = 0.05, activation_function = tan_h)
nn.init_weights()
x = [-1, 1, -1, -1, 1, -1]
print("Initialized prediction:", nn.predict(x))
y = [-1, -1]
nn.fit(x, y, tan_h_grad)
print("Error in nodes", nn.errors)
print("Prediction after fitting once", nn.predict(x))
```

```
Initialized prediction: [0.64299999 0.79969983]
Error in nodes [array([0.05582764, 0.57265422]), array([0.96370331, 0.648756
12])]
Prediction after fitting once [0.62330622 0.78936865]
```

(A)

The weights are initialized in the function init_weights to be random values from 0 to 1.

(B)

A new NN object is created and given the activation function, the inital x vector, and the learning rate. The weights are then initialized and then the funcion nn.predict(x) is called to compute the first prediction of the neural net before fitting. I chose the learning rate parameter to be 0.05 which resulted in an output which is closer to the reference output provided. From this initial prediction the neural net suggests that the predicted output is [0.64299999 0.79969983] which does not fall into the classifications of helix [1, -1], beta sheet [-1, 1], or coil [-1, -1] but can be rounded to [1, 1] which suggests that the secondary structure is hydophobic (1) and has propensity to form a helix (1). This is only half correct because we know that the observed structure for x is [-1, -1] which is a coil.

In [426…
```
def tan_h(x):
    return np.tanh(x)

def tan_h_grad(x):
    return 1 - tan_h(x)**2

np.random.seed(0)

nn = NN([6, 2, 2], learning_rate = 0.05, activation_function = tan_h)
nn.init_weights()
x = [-1, 1, -1, -1, 1, -1]
print("Initialized prediction:", nn.predict(x))
```

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

(C)

The error is calculated in the function nn.calc_error which creates an array of all the nodes which are not the input layer. A value for y is given as [-1, -1] and is passed to the nn.fit() function along with the initial vector x and the derivative of the activation function. Then the array of errors is reported to the user along with a new prediction from the neural net after one call to the nn.fit() function.

In [427…
```
y = [-1, -1]
nn.fit(x, y, tan_h_grad)
print("Error in nodes", nn.errors)
print("Prediction after fitting once", nn.predict(x))
```

```
Error in nodes [array([0.05582764, 0.57265422]), array([0.96370331, 0.648756
12])]
Prediction after fitting once [0.62330622 0.78936865]
```

(D)

The weights is a list of arrays which contains all the weights from each layer except the final layer. The equation to calculate the new weights is given in the function nn.back_prop(), which uses the current weight values and the gradient of the cost function with respect to each weight which is an array nn.weights_grad. The equation to compute the weights_grad values is given in the reference reading as well as the equation to calculate the biases. The gradient of the cost function with respect to the biases is equal to the array nn.errors. The learning rate parameter is chosen to be 0.05 which results in a new prediction which is close to the suggested output.

In [ ]: