Jericho McLeod

Homework 3

Problem 4.7

Terms:
$\langle \vec{x}, \vec{t} \rangle$ = training example, where $\vec{x}$ is the vector of network input and $\vec{t}$ is the vector of target network output values.
η = learning rate (e.g. 0.5, or some other small value from 0 to 1)
x_{ji} = the input from $unit_i$ to $unit_j$ (For $unit_j$, the input x comes from $unit_i$, hence x_{ji})
w_{ji} = the weight from $unit_i$ to $unit_j$
n_{in} = the number of network inputs
n_{out} = the number of network outputs
n_{hidden} = the number of units in the hidden layer

Algorithm:
Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units
Initialize all network weights to small random numbers (e.g. between -0.5 and 0.5)
Until the termination condition is met:

For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, do:

Propogate the input forward through the network:

1) Input instance $\vec{x}$ to the network and compute the output of o_u of every unit u in the network

Propogate the errors backward through the network:

2) For each network output unit k, calculate its error term δ_k:

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3) For each hidden unit h, calculate its error terms δ_h:

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

4a) Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \triangle\, w_{ji}$$

where

$$\triangle\, w_{ji} = \eta \delta_j x_{ji}$$

4b) Adding momentum changes the delta equation such that:

$$\triangle\, w_{ji} = \eta \delta_j x_{ji} + \alpha \triangle\, w_{ji}(n-1)$$

This serves to make future updates depend partially on the prior updates, i.e., adds momentum.

Citation: Thomas M. Mitchell. 1997. Machine Learning (1 ed.), Page 98, McGraw-Hill, Inc., New York, NY, USA.

Assigning Variables:
Weights reference:
0: w_{ca}
1: w_{cb}
2: w_{c0}
3: w_{dc}
4: w_{d0}

To solve this, we need to load data into a data structure and import any necessary libraries.

In [234]:
```
import math
training_examples = {'a':[1,0],'b':[0,1],'d':[1,0]}
for i in range(2):
    print('Observation',i+1)
    for k,v in training_examples.items():
        print(k,v[i])
```

```
Observation 1
a 1
b 0
d 1
Observation 2
a 0
b 1
d 0
```

Next we need to creat a Neural Network class that contains the functions we will need to solve this problem.

Note that this implementation is a first attempt, and is not able to create a neural network to any specific conditions, but only a solution to the specific problem at hand.

The init function instatiates the class with several variables, including the input arrays, the weights and deltas of the weights in the network, the output and hidden layer results, and the learning and momentum rates.

The print weights function is merely an output function for ease of showing the results required; the weights after the first two iterations of training.

The feed-forward function passes inputs through the network using a sigmoid function for the results. Note that it uses the 'itera' variable: if this function were to be trained by looping through the dataset multiple times, this would need to be altered so that itera%length(dataset) = value to be used from the dataset, and the hidden layer and output values would need to be a single value rather than an array, as the size of the array could be massive otherwise.

The back-propogation formula calculates the delta_k and delta_h values, then uses these to update the weights, along with momentum that is 0.9 * the prior iteration's delta weight.

In [235]:

```
class ANN:
    def __init__(self, a, b, d):
        self.input_a     = a
        self.input_b     = b
        self.weights_ca = 0.1
        self.weights_cb = 0.1
        self.weights_c0 = 0.1
        self.weights_dc = 0.1
        self.weights_d0 = 0.1
        self.weights_ca_delta = 0
        self.weights_cb_delta = 0
        self.weights_c0_delta = 0
        self.weights_dc_delta = 0
        self.weights_d0_delta = 0
        self.d           = d
        self.output      = []
        self.hidden_layer = []
        self.learning_rate = 0.3 # eta in the formula
        self.alpha = 0.9

    def print_weights(self):
        print('Weight CA',self.weights_ca)
        print('Weight CB',self.weights_cb)
        print('Weight C0',self.weights_c0)
        print('Weight DC',self.weights_dc)
        print('Weight D0',self.weights_d0)


    def feed_forward(self,itera):
        output_calc = lambda x: 1/(1+math.e**-x) #sigmoid
        temp_hidden_val = self.weights_ca * self.input_a[itera]\
                        + self.weights_cb * self.input_b[itera]\
                        + self.weights_c0
        hidden_val = output_calc(temp_hidden_val)
        self.hidden_layer.append(hidden_val)
        temp_out_val = self.weights_dc * self.hidden_layer[itera] + self
.weights_d0
        out_val = output_calc(temp_out_val)
        self.output.append(out_val)

    def backprop(self,itera):
        ca_2 = 0
        cb_2 = 0
        c0_2 = 0
        dc_2 = 0
        d0_2 = 0
        error = []
        hidden_error = []
        delta_k = self.output[-1]*(1-self.output[-1])*(self.d[itera]-sel
f.output[-1])
        delta_h = self.hidden_layer[-1]*(1-self.hidden_layer[-1]) * (sel
f.weights_dc * delta_k)
        error.append(delta_k)
        hidden_error.append(delta_h)
        delta_ca_2 = self.learning_rate * delta_h * self.input_a[itera]
+ self.alpha * self.weights_ca_delta
```

```
        delta_cb_2 = self.learning_rate * delta_h * self.input_b[itera]
+ self.alpha * self.weights_cb_delta
        delta_c0_2 = self.learning_rate * delta_h * 1 + self.alpha * sel
f.weights_c0_delta
        delta_dc_2 = self.learning_rate * delta_k * self.hidden_layer[it
era] + self.alpha * self.weights_dc_delta
        delta_d0_2 = self.learning_rate * delta_k * 1 + self.alpha * sel
f.weights_d0_delta
        self.weights_ca_delta = delta_ca_2
        self.weights_cb_delta = delta_cb_2
        self.weights_c0_delta = delta_c0_2
        self.weights_dc_delta = delta_dc_2
        self.weights_d0_delta = delta_d0_2
        self.weights_ca = self.weights_ca + self.weights_ca_delta
        self.weights_cb = self.weights_cb + self.weights_cb_delta
        self.weights_c0 = self.weights_c0 + self.weights_c0_delta
        self.weights_dc = self.weights_dc + self.weights_dc_delta
        self.weights_d0 = self.weights_d0 + self.weights_d0_delta
```

Now we can instantiate the neural network with the input data, run through two iterations of training, and print the outputs.

In [236]:
```python
nn = ANN(training_examples['a'],training_examples['b'],training_examples
['d'])
print("Starting Weights")
nn.print_weights()

print("\nWeights after 1 iteration")
nn.feed_forward(0)
nn.backprop(0)
nn.print_weights()

print("\nWeights after 2 iterations")
nn.feed_forward(1)
nn.backprop(1)

nn.print_weights()
```

```
Starting Weights
Weight CA 0.1
Weight CB 0.1
Weight C0 0.1
Weight DC 0.1
Weight D0 0.1

Weights after 1 iteration
Weight CA 0.10085128181864877
Weight CB 0.1
Weight C0 0.10085128181864877
Weight DC 0.1189103977991797
Weight D0 0.1343929220303063

Weights after 2 iterations
Weight CA 0.10161743545543266
Weight CB 0.09879852785432702
Weight C0 0.10041596330975969
Weight DC 0.11347416438338892
Weight D0 0.12452152136346561
```

Problem 4.8

Revise the backpropagation algorithm in Table 4.2 so that it operates on units using the squashing function $tanh$ in place of the sigmoid function. That is, assume the output of a single unit is $o = tanh(\vec{w} \times \vec{x})$. Give the weight update rule for output layer weights and hidden layer weights.

The sigmoid output error function is

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

and the hidden layer error function is

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

because the sigmoid function is

$$\sigma(y) = \frac{1}{1 + e^{-} y}$$

and its derivative is

$$\frac{d\sigma(y)}{dy} = \sigma(y) \times (1 - \sigma(y))$$

Extrapolating that the error terms are then

$$\delta_k \leftarrow G'(k) \times (t_k - o_k)$$

and

$$\delta_h \leftarrow G'(k) \sum_{k \in outputs} w_{kh}\delta_k$$

Then, for a $tanh$ function

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

for which the derivative is

$$\tanh'(x) = 1 - \tanh^2(x)$$

the output error function is

$$\delta_k \leftarrow 1 - \tanh^2(o_k) \times (t_k - o_k)$$

and the hidden layer output error function is

$$\delta_h \leftarrow 1 - \tanh^2(o_k) \times \sum_{k \in outputs} w_{kh}\delta_k$$

Therefore, the complete algorithm for backpropagation using the $tanh$ squashing function is:

Algorithm:
Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units
Initialize all network weights to small random numbers (e.g. between -0.5 and 0.5)
Until the termination condition is met:

For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, do:

Propogate the input forward through the network:

1) Input instance $\vec{x}$ to the network and compute the output of o_u of every unit u in the network

Propogate the errors backward through the network:

2) For each network output unit k, calculate its error term δ_k:

$$\delta_k \leftarrow 1 - \tanh^2(o_k) \times (t_k - o_k)$$

3) For each hidden unit h, calculate its error terms δ_h:

$$\delta_h \leftarrow 1 - \tanh^2(o_k) \times \sum_{k \in outputs} w_{kh}\delta_k$$

4a) Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \triangle w_{ji}$$

where

$$\triangle w_{ji} = \eta \delta_j x_{ji}$$