In [175]:

```
#I have run some open source code for neural network.

import math
import numpy as np

def sigmoid(t, deriv=False):
    if deriv:
        return t * (1 - t)
    return 1 / (1 + np.exp(-t))

def neural_network_with_shapes(X, y, neurons_per_layer, iterations=100000, verbose=False, plot=Fals
e):
    """ Trains an `n`-layer neural network on feature matrix `X` and outcomes `y`
    :param X: input feature matrix where rows are observation and columns are features
    :param y: outcome associated to each observation in `X`
    :param neurons_per_layer: array of integers representing the number of neurons for each layer
    :return: matrix of weights
    Usage:
        To train a neural network with one input layer, one hidden layer and one output layer,
        # For reproducibility, set the random seed
        >>> np.random.seed(1)
        First, generate training data
        >>> X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])     # observations
        >>> y = np.array([0,0,1,1]).reshape(4,1)                # output
        Then, set the network shape:
            3: input layer with 3 neurons    (features)
            5: hidden layer with 4 neurons
            1: output layer with 1 neurons   (predicted value)
        >>> neurons = [ 3, 5, 1, ]
        From that neuron configuration, the network will look like this:
            (ascii is failing me here... every neuron in the input layer is connected
            to every neuron in the hdiden layer and the same for the hidden and output
            layers).
          input   hidden   output
            O-------O-----\
                \---O------\
            O-------O-------O
                /---O------/
            O-------O-----/
        Now the fun part. Build and train a neural network with the structures you just defined.
        `neural_network_with_shapes()` with return the matrix of weights (all the synapses)
        >>> weights = neural_network_with_shapes(X, y, neurons_per_layer=neurons, iterations=100,
verbose=False, plot=True)
        To use the network, call `forward_propagation` with the weights you just trained.
        >>> print predict(X, weights)
        [[ 0.09494851]
         [ 0.06620708]
         [ 0.922808  ]
         [ 0.90488084]]
    """

    num_synapses = len(neurons_per_layer) - 1  # number of synapses in the nn (consequently, there
are `num_synapses+1` layers.
    synapses = []  # list of matrix of weights connecting layer i and i+1
    errors = []

    # initialize weights randomly with mean 0
    for i in range(num_synapses):
        shape = (neurons_per_layer[i], neurons_per_layer[i + 1])
        synapses.append(2 * np.random.random(shape) - 1)

    # train the network
    for j in range(iterations):

        # X,y = regression.shuffle_in_unison_inplace(X,y)

        # FORWARD PROPAGATION
        layers = forward_propagation(X, synapses)

        # BACK PROPAGATION

        # first process last layer using `y`
```

```
        ln = layers[-1]
        ln_error = y - ln

        print("Error: ",ln_error)

        if verbose and j % 10 == 0:
            print("Error: {}".format(np.mean(np.abs(ln_error))))

        if plot:
            errors.append((ln_error ** 2).sum())

        s = sigmoid(ln, deriv=True)
        delta = ln_error * sigmoid(ln, deriv=True)

        # then update every preceding layers
        for i in reversed(range(num_synapses)):
            # update the synapse based on succeeding layers
            d = layers[i].T.dot(delta)
            print (d)
            synapses[i] += layers[i].T.dot(delta)

            # if we've reached the input layer
            if i == 0:
                break

            # how much did `layers[i]` contribute to the `layers[i]+1` error
            error = delta.dot(synapses[i].T)

            # error weighted derivative
            delta = error * sigmoid(layers[i], deriv=True)

    # if plot:
    #     import matplotlib.pyplot as plot
    #     plot.plot(range(iterations), errors)
    #     plot.show()

    return synapses


def error(a, y, f='quadratic'):
    """Computes the error of the output `a` given true value `y`"""

    if f is 'quadratic':
        return y - a
        # return ((y-a)**2)/2
    if f is 'cross-entropy':
        n = a.shape[0]
        return -1 / n * (y * np.log(a) + (1 - y) * np.log(1 - a)).sum()


def forward_propagation(X, synapses):
    """Performs feed forward propagation through layers 0, 1, ..., len(synapses)-1"""
    layers = [X]
    for i in range(len(synapses)):
        layers.append(sigmoid(np.dot(layers[i], synapses[i])))
    return layers


def predict(X, weights):
    return forward_propagation(X, weights)[-1]
```

In [164]:

```
np.random.seed(1)
```

In [168]:

```
#X = np.array([[.9,.1,.1,.5,.9,.1,.9,.1,.1], [.9,.1,.1,.9,.1,.1,.1,.3,.7],
[.1,.9,.1,.5,.1,.1,.1,.6,.1]])
#y = np.array([.9,.1,.9]).reshape(3,1)

X = np.array([[.9,.1,.1,.5,.9,.1,.9,.1,.1],
            [.9,.1,.1,.9,.1,.1,.1,.3,.4],
            [.1,.9,.1,.5,.1,.1,.1,.6,.1],
            [.9,.1,.9,.9,.1,.9,.1,.3,.3],
```

```
            [.9,.1,.9,.9,.9,.1,.9,.1,.9],
            [.1,.9,.1,.5,.5,.9,.9,.9,.1],
            [.1,.9,.1,.1,.1,.9,.1,.6,.1],
            [.1,.1,.1,.5,.5,.9,.9,.3,.1],
            [.1,.9,.9,.9,.1,.9,.1,.6,.9],
            [.9,.9,.9,.9,.9,.1,.9,.9,.3],
            [.1,.1,.1,.1,.1,.1,.1,.3,.1],
            [.9,.9,.9,.9,.1,.1,.1,.6,.7]])

y = np.array([.9,.1,.9,.9,.1,.9,.1,.9,.1,.1,.1,.9]).reshape(12,1)
```

In [169]:

```
neurons = [ 9, 3, 1 ]
```

In [ ]:

```
weights = neural_network_with_shapes(X, y, neurons_per_layer=neurons, iterations=1500, verbose=False, plot=True)
#You need to run this command to see all the iterations
```

In [177]:

```
print (predict(X, weights))
```

```
[[0.90000903]
 [0.100773  ]
 [0.89945941]
 [0.89962391]
 [0.09979444]
 [0.90128944]
 [0.10046674]
 [0.89902345]
 [0.10135316]
 [0.10056924]
 [0.09739317]
 [0.89883331]]
```

Error: [[-8.89459056e-06] [-7.74315989e-04] [5.41574270e-04] [3.76883220e-04] [2.05945622e-04] [-1.29076100e-03] [-4.67879707e-04] [9.77490971e-04] [-1.35495904e-03] [-5.70236833e-04] [2.61065168e-03] [1.16833005e-03]]

neural network with three hidden nodes is used and the weight learned are: node1:[[5.11995577e-05] node2:[2.78366644e-05] node3: [1.04827823e-05]]

The dimension from input layer to hidden layer: [[-1.06603359e-05 8.42058154e-06 -1.57902699e-05] [-2.03985728e-05 -5.73937833e-05 1.26868012e-05] [8.99823125e-07 4.65129495e-05 -2.25829817e-05] [1.84962138e-05 9.47029617e-06 4.24276246e-05] [-5.38428117e-06 -1.57504376e-05 1.03051419e-05] [-1.69429909e-05 -5.21152580e-06 3.05112724e-05] [-1.16304962e-05 -2.19313527e-05 -2.41380125e-06] [4.27627419e-05 -1.80428856e-05 -2.56999611e-05] [2.07220530e-05 3.89642432e-05 1.58582282e-05]]

predicted value is below: [[0.90030195] [0.11940869] [0.89019049] [0.89740176] [0.09671028] [0.90097833] [0.10309001] [0.89300469] [0.11162736] [0.10737455] [0.07713583] [0.88841444]], which are very close to the output value.

In [ ]:

In [ ]: