```python
#!/usr/bin/env python
import csv, sys
from sympy import *
import numpy as np
from numpy import linalg as ln
import numexpr as ne
import pdb


class CSVInput:
    def __init__(self, filename, first_row_titles=False, num_convert=True, set_true_false_01=True):
        self.titles = []
        self.data = []
        self.boolean_false = ['F', 'f', 'False', 'FALSE', 'false']
        self.boolean_true  = ['T', 't', 'True', 'TRUE', 'true']
        with open(filename, 'rb') as file:
            reader = csv.reader(file, delimiter='\t')
            for i, row in enumerate(reader):
                if i==0 and first_row_titles:
                    self.titles += row
                else:
                    if num_convert:
                        row_list = []
                        for elem in row:
                            try:
                                value = float(elem)
                            except ValueError:
                                try:
                                    value = int(elem)
                                except ValueError:
                                    value = elem
                                    if any(false in value for false in self.boolean_false):
                                      value = 0
                                    elif any(true in value for true in self.boolean_true):
                                      value = 1
                            row_list.append(value)
                    self.data.append(row_list[1:])
        self.rows = len(self.data)
        self.cols = len(self.data[0])


class Sigmoid:
    def __init__(self):
        t = symbols('t')
        self.sigmoid = 1 / (1 + exp(-t))
        self.sigmoid_dif = self.sigmoid * (1 - self.sigmoid)
        self.sigmoid_reverse = t * (1 - t)

    def Sigmoid(self, t):
        return ne.evaluate(str(self.sigmoid))

    def SigmoidPrime(self, t):
        return ne.evaluate(str(self.sigmoid_dif))

    def SigmoidReverse(self, t):
        return ne.evaluate(str(self.sigmoid_reverse))

class Gamma(object):
    def __init__(self, initial, size):
        self.gamma =  np.ones((size, 1)).T * initial

    def Update(self, E):
        pass

class GammaSpeed(Gamma):
    def __init__(self, initial, u, d, size):
        super(GammaSpeed, self).__init__(initial, size)
```

```python
        self.u = u
        self.d = d
        self.E_p = 0
    def Update(self, E):
        pass

class GammaRPROP(Gamma):
    def __init__(self, initial, u, d, gamma_min, gamma_max, size):
        super(GammaRPROP, self).__init__(initial, size)
        self.u = u
        self.d = d
        self.gamma_min = gamma_min
        self.gamma_max = gamma_max
        self.E_p = np.zeros((size, 1)).T

    def Update(self, E):
        self._Evaluate(self.E_p, E, self.gamma, self.gamma_min, self.gamma_max, self.u, self.d)

    def _Evaluate(self, E_p, E, gamma, gamma_min, gamma_max, u, d):
        self.method = \
            'where( E_p * E == 0,\
                gamma,\
                where(E_p * E > 0,\
                    where(gamma * u < gamma_max,\
                        gamma * u,\
                        gamma_max\
                    ),\
                    where(gamma * d < gamma_min,\
                        gamma_min,\
                        gamma * d\
                    )\
                )\
            )'

        # self.method = 'gamma * E'

        self.gamma = ne.evaluate(self.method)
        self.E_p = E

class NeuralNetwork:
    def __init__(self, feature_size, compute_components, output_size):
        #self.o_zero     = np.matrix(np.ones((1, feature_size)))
        self.o_zero_bar = np.matrix(np.ones((1, feature_size + 1)))

        self.W_one_bar  = np.matrix(np.random.rand(feature_size + 1, compute_components))
        #self.W_one_bar   = np.ones([feature_size + 1, compute_components]) * 1.0
        #self.W_one      = self.W_one_bar[0:-1,:]

        #self.o_one      = np.matrix(np.ones((1, compute_components)))
        self.o_one_bar  = np.matrix(np.ones((1, compute_components + 1)))

        self.W_two_bar  = np.matrix(np.random.rand(compute_components + 1, output_size))
        #self.W_two_bar  = np.ones([compute_components + 1, output_size]) * 1.0
        #self.W_two      = self.W_two_bar[0:-1,:]

        self.o_two      = np.matrix(np.ones((1, output_size)))
        self.S = Sigmoid()

        self.delta_W_one_trans = np.zeros(self.W_one_bar.T.shape)
        self.delta_W_two_trans = np.zeros(self.W_two_bar.T.shape)
        self.gamma = 1

    def FeedForward(self, feature, truth):
        # Compute the Feed forward pass of an iteration on the Neural Network
        # Need to append one onto the end of each of the weight and feature vector
        self.o_zero_bar[0, :-1] = np.matrix(feature)
        o_one = self.S.Sigmoid(self.o_zero_bar * self.W_one_bar)
```

```
        self.o_one_bar[0, :-1] = o_one
        self.o_two = self.S.Sigmoid(self.o_one_bar * self.W_two_bar)

        self.error = (truth - self.o_two)
        #print self.error
        return 1.0 / 2.0 * ln.norm(truth - self.o_two)**2.0

    def BackProp(self):
        self.D_two = np.matrix(np.diag(self.S.SigmoidReverse(self.o_two)[0]))
        self.D_one = np.matrix(np.diag(self.S.SigmoidReverse(self.o_one_bar[0,:-1])[0]))

        self.S_two = self.D_two * self.error.T
        self.S_one = self.D_one * self.W_two_bar[0:-1,:] * self.S_two

        self.delta_W_two_trans += (- self.gamma * self.S_two * self.o_one_bar)
        self.delta_W_one_trans += (- self.gamma * self.S_one * self.o_zero_bar)


        # Now A decision based on the parameters of the neural network need to be made. In this case the
Either we perform the offline approach(batch mode) or the online update)

    def UpdateWeights(self):
        self.W_two_bar = self.W_two_bar - self.delta_W_two_trans.T
        self.W_one_bar = self.W_one_bar - self.delta_W_one_trans.T
        self.delta_W_two_trans = np.zeros(self.delta_W_two_trans.shape)
        self.delta_W_one_trans = np.zeros(self.delta_W_one_trans.shape)

    def OnlineNeuralNetwork(self, iterations, features, truth):
        truth_matrix = 0  # This will prdoduce the necessary vectors for the error calculation at the end
of each feed forward step

    def BatchNeuralNetwork(self, iterations, features, truth):
        truth_matrix = 0



def main():
    # Read Data in and convert numbers to numbers
    reader = CSVInput(sys.argv[1], first_row_titles=True)

    # gamma = GammaRPROP(1, .001, 3, .00001, 10, )
    # print reader.data
    num_compute_nodes = int(sys.argv[2])
    gamma = float(sys.argv[3])
    num_output_nodes = 2

    #pdb.set_trace()
    neural = NeuralNetwork(reader.cols-1, num_compute_nodes, num_output_nodes)

    network_error_threshold = 1e-4
    np_data = np.matrix(reader.data)
    truth = np.matrix(np.ones((reader.rows, 2)))
    truth[:,0:1] = np_data[:,-1]
    truth[:,1:2] = 1-np_data[:,-1]

    count = 0
    network_error = network_error_threshold + 1

    with open('result_%d_%f.csv' % (num_compute_nodes, gamma), 'w') as file:
        while network_error > network_error_threshold:
            network_error = 0.0
            for sample_index in range(reader.rows):
                x = np.matrix(reader.data[sample_index][0:-1])
                t = truth[sample_index]
                network_error += neural.FeedForward(x, t)
                neural.BackProp()
```

```
            neural.UpdateWeights()
            count = count + 1
            network_error /= reader.rows
            file.write('%s;\n' % (repr(network_error)))
            file.flush()
            # print "Iterations %d, error:%s"%(count,repr(network_error))
        print "Final error",network_error

if __name__ == '__main__':
    main()
```