```python
import numpy as np
import math

np.random.seed()

def sig(x):
    return 1 / (1 + math.exp(-x))

def sigmoid(x):
    return np.array(map(sig, x))

def d_sig(x):
    return sig(x)* (1 - sig(x))

def d_sigmoid(x):
    return np.array(map(d_sig, x))

class MLP:
    def __init__(self, shape, eta=0.15, momentum=0, init_lower_bound=-1, init_upper_bound=1):
        '''
            A Multi-Layer Perceptron class to create a Neural network

            shape               a tuple of the shape for the network, including input,
                                hidden, and output layers
            eta                 the learning rate parameter
            momentum            the momentum parameter (between 0 and 1)
            init_lower_bound    the lower bound of the random initialization of weights
            init_upper_bound    the upper bound of the random initialization of weights
        '''

        self.shape = shape
        self.eta = eta
        self.a = momentum
        self.weights = []
        self.outputs = []
        self.prev_w_deltas = []
        self.act_func = sigmoid
        self.backprop_func = d_sigmoid

        # add input and hiden layers, plus 1 bias term for each
        for i in range(0, len(shape) - 1):
            self.outputs.append(np.ones(shape[i] + 1))

        # Add output layer
        self.outputs.append(np.ones(shape[-1]))

        for i in range(0, len(self.outputs) - 1):
            layer = len(self.outputs[i])
            next_layer = len(self.outputs[i+1])
            if (i < len(self.outputs) - 2):
                next_layer -= 1
            weights = np.random.random(layer * next_layer)
            weight_range = init_upper_bound - init_lower_bound
            weights = weights * weight_range + init_lower_bound
            self.weights.append(weights.reshape((layer, next_layer)))

        self.weight_change = [0,]*len(self.weights)

        for i in range(0, len(self.weights)):
            self.prev_w_deltas.append(np.zeros_like(self.weights[i]))

```

```
    def _forward_pass(self, input_vector):

        x = input_vector[0]
        y = input_vector[1]

        # Put inputs as the initial activation outputs.
        # 1 is in front from initialization for the bias term
        self.outputs[0][1:] = x

        for i in range(0,len(self.shape) - 1):

            output = self.act_func(np.dot(self.weights[i].T, self.outputs[i]))

            # add a 1 for the bias node as an output for hidden layers
            if (i < len(self.shape) - 2):
                output = np.hstack((1, output))

            self.outputs[i+1] = output

        # return the output to the network
        return self.outputs[-1]


    def _backpropogate(self, target):

        deltas = []

        # Derive delta_k for output layer
        error = target - self.outputs[-1]
        delta_k = error * self.backprop_func(self.outputs[-1])
        deltas.append(delta_k)

        # Derive delta_j's for hidden layers
        for i in range(1, len(self.shape) - 1):
            output = self.outputs[-(i + 1)]
            d_out = np.array(self.backprop_func(output))
            delta_j = d_out * np.dot(deltas[-i],self.weights[-i].T)
            deltas.insert(0,delta_j[1:])

        # Update the weights
        for j in range(0, len(self.weights)):
            for k in range(0, len(self.weights[j].T)):
                weight_change = self.eta * deltas[j][k] * self.outputs[j]
                self.weights[j].T[k] += weight_change + self.a * self.prev_w_deltas[j].T[k]
                self.prev_w_deltas[j].T[k] = weight_change

    '''
        Train the network with data samples in an array with structure

        training_data   [
                            [x1, x2, x3, ...], [y1, y2, ...],
                            [x1, x2, x3, ...], [y1, y2, ...],
                            ...
                        ]

    '''
    def train(self, training_data, max_epoch=10000):

        epoch = 0
        errors = [True for i in range(len(training_data))]
```

```
        has_errors = True

        while (has_errors and epoch < max_epoch):
            epoch += 1
            if (epoch % 50 == 0):
                has_errors = sum(errors) != 0
            if (epoch % 10000 == 0):
                print("epoch {0}; {1} errors above 0.05".format(epoch, sum(errors)))
            if (epoch % 1000 == 0):
                    print("Epoch: {0}".format(epoch))

            # shuffle the training data around
            np.random.shuffle(training_data)

            for i,training_sample in enumerate(training_data):
                expected_output = training_sample[1][0]
                actual_output = self._forward_pass(training_sample)
                self._backpropogate(training_sample[1])
                errors[i] = abs(expected_output - actual_output[0]) > 0.05
        if (epoch == max_epoch):
            print("Did not converge")
        else:
            print("Converged in {0} epochs".format(epoch))
        return epoch


    '''
        Test the network with data samples in an array with structure
        [
            [x1, x2, x3, ...], [y1, y2, ...],
            [x1, x2, x3, ...], [y1, y2, ...],
            ...
        ]
    '''
    def test(self, training_data):
        print("Expected  |  Actual ")
        error = 0
        for training_sample in training_data:
            actual = self._forward_pass(training_sample)
            print("  {0}  |  {1}  ".format(training_sample[1], actual))
            error += abs(actual[0] - training_sample[1][0])
        return error
```