4. Problem : Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

Below is a small contrived dataset that we can use to test out training our neural network. X1 X2 Y 2.7810836 2.550537003 0 1.465489372 2.362125076 0 3.396561688 4.400293529 0 1.38807019 1.850220317 0 3.06407232 3.005305973 0 7.627531214 2.759262235 1 5.332441248 2.088626775 1 6.922596716 1.77106367 1 8.675418651 -0.242068655 1 7.673756466 3.508563011 1 Below is the complete example. We will use 2 neurons in the hidden layer. It is a binary classification problem (2 classes) so there will be two neurons in the output layer. The network will be trained for 20 epochs with a learning rate of 0.5, which is high because we are training for so few iterations.

In [1]:

```
import random
from math import exp
from random import seed
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range
(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range
(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network

# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

```python
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['o
utput'])
```

```
# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inpu
ts[j]
            neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in rang
e(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum
_error))

#Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
```

```
    [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)

#for layer in network:
 #   print(layer)

i= 1
for layer in network:
    j=1
    for sub in layer:
        print("\n Layer[%d] Node[%d]:\n" %(i,j),sub)
        j=j+1
    i=i+1
```

```
>epoch=0, lrate=0.500, error=4.763
>epoch=1, lrate=0.500, error=4.558
>epoch=2, lrate=0.500, error=4.316
>epoch=3, lrate=0.500, error=4.035
>epoch=4, lrate=0.500, error=3.733
>epoch=5, lrate=0.500, error=3.428
>epoch=6, lrate=0.500, error=3.132
>epoch=7, lrate=0.500, error=2.850
>epoch=8, lrate=0.500, error=2.588
>epoch=9, lrate=0.500, error=2.348
>epoch=10, lrate=0.500, error=2.128
>epoch=11, lrate=0.500, error=1.931
>epoch=12, lrate=0.500, error=1.753
>epoch=13, lrate=0.500, error=1.595
>epoch=14, lrate=0.500, error=1.454
>epoch=15, lrate=0.500, error=1.329
>epoch=16, lrate=0.500, error=1.218
>epoch=17, lrate=0.500, error=1.120
>epoch=18, lrate=0.500, error=1.033
>epoch=19, lrate=0.500, error=0.956

 Layer[1] Node[1]:
 {'weights': [-1.435239043819221, 1.8587338175173547, 0.791764422414809
```

```
4], 'output': 0.029795197360175857, 'delta': -0.0060187301177683358}

 Layer[1] Node[2]:
 {'weights': [-0.7704959899742789, 0.8257894037467045, 0.21154103288579
731], 'output': 0.06771641538441577, 'delta': -0.005025585510232048}

 Layer[2] Node[1]:
 {'weights': [2.223584933362892, 1.2428928053374768, -1.351954892552745
4], 'output': 0.23499833662766154, 'delta': -0.042246618795029306}

 Layer[2] Node[2]:
 {'weights': [-2.509732251870173, -0.5925943219491905, 1.25996572748409
3], 'output': 0.7543931062537561, 'delta': 0.04550706392557862}
```

Predict

Making predictions with a trained neural network is easy enough. We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class. It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the arg max function. Below is a function named predict() that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

In [2]:
```python
from math import exp

# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation

# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

```
# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

# Test making predictions with the network
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]
network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.0783
81922048799]}, {'weights': [0.23244990332399884, 0.3621998343835864, 0.
40289821191094327]}],
    [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757
805829]}, {'weights': [-2.429350576245497, 0.8357651039198697, 1.069921
7181280656]}]]
for row in dataset:
    prediction = predict(network, row)
    print('Expected=%d, Got=%d' % (row[-1], prediction))
```

```
Expected=0, Got=0
Expected=0, Got=0
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```