```
import numpy as np
import math


class Node:
    def __init__(self, layer, layer_index, input_weights, bias):
        self.layer = layer
        self.layer_index = layer_index
        self.input_weights = input_weights
        self.bias = bias

    def sum_weights(self, input_vals):
        if len(input_vals) != len(self.input_weights):
            raise RuntimeError("The number of inputs are not equal to the
number of weights")
        # return the dot product of the input values and input weights
        weight_sum = sum(np.multiply(input_vals, self.input_weights))
        return weight_sum + self.bias

    def get_activity(self, input):
        return 1 / (1 + math.exp(-input))

    def update_weight(self, index, value):
        self.input_weights[index] = value

    def update_bias(self, value):
        self.bias = value


class Network:
    def __init__(self, layers, eta, layer0, layer1, layer2=None):
        self.layers = layers
        self.network = []
        self.network.append(layer0)
        self.network.append(layer1)
        if layer2:
            self.network.append(layer2)
        self.eta = eta
        self.layer_input_vals = []

    def feed_forward(self, input_vals):
        current_layer = 1

        # track the results of feed forward for each layer
        self.layer_input_vals = []
        self.layer_input_vals.append(input_vals)

        # iterate over each layer to feed forward
        while current_layer < net.layers:
            layer_vals = []

            # get the activation values for each node
            for node in self.network[current_layer]:
                sum_weights = node.sum_weights(input_vals)
                print(f"sum of weights for node {node.layer_index} in layer
{current_layer} is {sum_weights}")
                sigmoid = node.get_activity(sum_weights)
```

```
                print(f"sigmoid node {node.layer_index} in layer
{current_layer} is {sigmoid}")
                layer_vals.append(sigmoid)

            # the input for the next layer is the result of the current layer
            print(f"the input for layer {current_layer+1} is {layer_vals}")
            input_vals = layer_vals
            self.layer_input_vals.append(input_vals)
            current_layer += 1

        print(f"the layer input vals are: {self.layer_input_vals}")
        # the result of feed forward is the result of the output layer
        print(f"the result of feed forward is: {layer_vals[0]}\n\n")
        return layer_vals[0]

    def back_prop(self, output, desired_output):
        error_vals = [desired_output - output]
        layer_error = []
        current_layer = self.layers - 1
        while current_layer > 0:
            for node_index, node in enumerate(self.network[current_layer]):
                for weight_index, input_weight in
enumerate(node.input_weights):
                    node_output =
self.layer_input_vals[current_layer][node_index]
                    print(f"output for layer {current_layer}, node index
{node_index} was {node_output}")
                    delta = error_vals[node_index] * (1-node_output) *
node_output
                    print(f"delta for weight {weight_index} of val
{input_weight} is {delta}")
                    weight_input_val = self.layer_input_vals[current_layer-
1][node_index]
                    new_weight = input_weight + (self.eta * delta *
weight_input_val)
                    print(f"new weight for layer {current_layer} node
{node_index} weight index {weight_index} is: {new_weight}\n")
                    layer_error.append(delta * input_weight)
                    node.update_weight(weight_index, new_weight)
                new_bias = node.bias + self.eta * delta * 1
                print(f"new bias for layer {current_layer} node {node_index}
is {new_bias}")
                node.update_bias(new_bias)
            print(f"\nlayer error for layer {current_layer} was:
{layer_error}\n")
            error_vals = layer_error
            layer_error = []
            current_layer -= 1

    def print_weights(self):
        print("network weights")
        for layer_index, layer in enumerate(self.network):
            print(f"----layer {layer_index}")
            for node_index, node in enumerate(layer):
                print(f"node {node_index} input weights:
{node.input_weights}; bias: {node.bias}")
```

```
# lecture example
input1 = Node(0, 0, [], 0)
input2 = Node(0, 1, [], 0)
hidden1 = Node(1, 0, [0.3, 0.3], 0)
hidden2 = Node(1, 1, [0.3, 0.3], 0)
output = Node(2, 0, [0.8, 0.8], 0)
net = Network(3, 1, [input1, input2], [hidden1, hidden2], [output])

net.print_weights()
ff = net.feed_forward([1, 2])
net.back_prop(ff, 0.7)
net.print_weights()

# print("lecture sample:")
# input1 = Node(0, 0, [], 0)
# input2 = Node(0, 1, [], 0)
# output = Node(1, 0, [0.8, 0.8], 0)
# net = Network(2, 1, [input1, input2], [output])
# net.print_weights()
# ff = net.feed_forward([.7109495, .7109495])
# net.back_prop(ff, 0.7)
# net.print_weights()

print("----------------------------")
print("Q1")

# question1
input1 = Node(0, 0, [], 0)
input2 = Node(0, 1, [], 0)
output = Node(1, 0, [0.24, 0.88], 0)
net = Network(2, 5, [input1, input2], [output])

ff = net.feed_forward([0.8, 0.9])
print(f"Q1 sol: {ff}")

print("----------------------------")
print("Q2")

# question2
# input1 = Node(0, 0, [])
# input2 = Node(0, 1, [])
# output = Node(1, 0, [0.24, 0.88])
# net = Network(2, 5, [input1, input2], [output])
input1 = Node(0, 0, [], 0)
input2 = Node(0, 1, [], 0)
output = Node(1, 0, [0.24, 0.88], 0)
net = Network(2, 5, [input1, input2], [output])

for iter in range(75):
    ff = net.feed_forward([0.8, 0.9])
    net.back_prop(ff, 0.95)
ff = net.feed_forward([0.8, 0.9])
print(f"Q2 sol: {ff}")

print("----------------------------")
print("Q3")
```

```
# question3
input1 = Node(0, 0, [], 0)
input2 = Node(0, 1, [], 0)
output = Node(1, 0, [0.24, 0.88], 0)
net = Network(2, 5, [input1, input2], [output])

for iter in range(30):
    ff = net.feed_forward([0.8, 0.9])
    net.back_prop(ff, 0.15)
ff = net.feed_forward([0.8, 0.9])
print(f"Q3 sol: {ff}")
```