```
import math
import itertools

class Node:
    def __init__(self, children, weights, bias=None):
        if len(children) != len(weights):
            raise RuntimeError("The number of children are not equal to the
number of weights")
        self.children = children
        self.weights = weights
        self.bias = bias
        self.parents = []
        self.activation_val = None
        self.delta = None

        # build the node parent list
        for child in self.children:
            child.parents.append(self)

    def sum_weights(self):
        # sum the weights times the input values
        output_val = 0
        for (child, weight) in zip(self.children, self.weights):
            output_val += child.activation_val * weight

        # if the neuron has a bias add on the bias
        if self.bias is not None:
            output_val += self.bias
        return output_val

    def get_activation(self, outut_val):
        return 1 / (1 + math.exp(-outut_val))

    def fire(self):
        print(f"input values: [", end="")
        for node in self.children:
            print(f"{node.activation_val}, ", end="")
        print(f"]\nweights: {self.weights}")
        print(f"bias: {self.bias}")
        neuron_output = self.sum_weights()
        print(f"weight sum {neuron_output}")
        neuron_activation = self.get_activation(neuron_output)
        self.activation_val = neuron_activation
        print(f"activation value: {self.activation_val}")
        return neuron_activation

    def set_activation(self, value):
        self.activation_val = value

    def set_delta(self, error):
        print(f"delta value: {error * self.activation_val * (1 -
self.activation_val)}")
        self.delta = error * self.activation_val * (1 - self.activation_val)

    def update_weights(self, eta):
        for index, (child, weight) in enumerate(zip(self.children,
self.weights)):
```

```
            # print(f"loop index: {index}; child: {child}; child_activation:
{child.activation_val}; weight: {weight}")
            weight_increment = eta * self.delta * child.activation_val
            print(f"weight increment: {weight_increment}")
            self.increment_weight(index, weight_increment)
        print(f"new weights: {self.weights}")

        if self.bias is not None:
            self.increment_bias(eta * self.delta)
            print(f"new bias: {self.bias}")

    def increment_weight(self, index, increment):
        self.weights[index] += increment

    def increment_bias(self, increment):
        self.bias += increment


class Network:
    def __init__(self, structure, eta=1):
        self.structure = structure
        self.eta = eta

    def feed_forward(self, input_vals):

        network_result = []

        if len(input_vals) != len(self.structure[0]):
            raise RuntimeError(f"There are a differnet number of input values
{len(input_vals)} from input nodes {len(self.structure[0])}")

        # set the values for each of the input nodes
        for (node, value) in zip (self.structure[0], input_vals):
            node.set_activation(value)

        # feed the values forward in the network
        for layer_num, layer in enumerate(self.structure[1:]):
            print(f"feed forward layer {layer_num+1}")
            for node in layer:
                # layer is not the last layer
                if layer_num+1 != len(self.structure)-1:
                    node.fire()
                else:
                    network_result.append(node.fire())

        print(f"feed forward result: {network_result}\n")
        return network_result

    def back_prop(self, output, desired_output):

        # set the delta values for the last layer
        for node in self.structure[-1]:
            print(f"set delta values for output layer")
            error = desired_output - output
            node.set_delta(error)

        current_layer = len(self.structure) - 1
```

```
        while current_layer > 0:
            # set the delta values for the layer one below the current layer
            print(f"set delta values for layer {current_layer-1}")
            for node in self.structure[current_layer-1]:
                error = 0
                for parent in node.parents:
                    parent_weight = None
                    parent_delta = None
                    for index, child in enumerate(parent.children):
                        if child == node:
                            parent_weight = parent.weights[index]
                            parent_delta = parent.delta
                    error += parent_weight * parent_delta
                # print(f"node error value: {error}")
                node.set_delta(error)

            # update the weights for the current layer
            print(f"set weights for layer {current_layer}")
            for node in self.structure[current_layer]:
                node.update_weights(self.eta)

            current_layer -= 1
        self.clean_network()

    def clean_network(self):
        for layer in self.structure:
            for node in layer:
                node.activation_val = None
                node.delta = None

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


# lecture example
input1 = Node([], [])
input2 = Node([], [])
hidden1 = Node([input1, input2], [0.3, 0.3])
hidden2 = Node([input1, input2], [0.3, 0.3])
output = Node([hidden1, hidden2], [0.8, 0.8])
net = Network([[input1, input2], [hidden1, hidden2], [output]])

net.print_weights()
ff = net.feed_forward([1, 2])
net.back_prop(ff[0], 0.7)
net.print_weights()

# print("----------------------------")
# print("Q1")
#
# # question1
```

```
# input1 = Node([], [])
# input2 = Node([], [])
# output = Node([input1, input2], [0.24, 0.88], 0)
# net = Network([[input1, input2], [output]])
#
# net.print_weights()
# ff = net.feed_forward([0.8, 0.9])
# print(f"Q1 sol: {ff}")
#
# print("----------------------------")
# print("Q2")
#
# # question2
# input1 = Node([], [])
# input2 = Node([], [])
# output = Node([input1, input2], [0.24, 0.88], 0)
# net = Network([[input1, input2], [output]], 5)
#
# net.print_weights()
# for iter in range(75):
#     ff = net.feed_forward([0.8, 0.9])
#     net.back_prop(ff[0], 0.95)
# ff = net.feed_forward([0.8, 0.9])
# print(f"Q2 sol: {ff[0]}")
# net.print_weights()
#
# print("----------------------------")
# print("Q3")
#
# # question3
# input1 = Node([], [])
# input2 = Node([], [])
# output = Node([input1, input2], [0.24, 0.88], 0)
# net = Network([[input1, input2], [output]], 5)
#
# for iter in range(30):
#     ff = net.feed_forward([0.8, 0.9])
#     net.back_prop(ff[0], 0.15)
# ff = net.feed_forward([0.8, 0.9])
# print(f"Q3 sol: {ff[0]}")

print("----------------------------")
print("M6")

# Module 6 problem
input1 = Node([], [])
input2 = Node([], [])
hidden1 = Node([input1, input2], [0.8, 0.1])
hidden2 = Node([input1, input2], [0.5, 0.2])
output = Node([hidden1, hidden2], [0.2, 0.7])
net = Network([[input1, input2], [hidden1, hidden2], [output]], .1)

ff = net.feed_forward([1, 3])
net.back_prop(ff[0], 0.95)
```