```python
# Feed Forward Back Propagation, version 2: 2 inputs -> 2 hidden -> 1 output ,no biases)
# by Andrew Taylor

import numpy as np

def sigmoid(a):
    return 1.0 / (1.0 + np.exp(-a))
def d_sigmoid_from_y(y):
    return y * (1.0 - y) # derivative of sigmoid

class Network:
    def __init__(self, W_in_h, W_h_out, eta=0.1):
        self.W_in_h = np.array(W_in_h, dtype=float)     # (2,2)
        self.W_h_out = np.array(W_h_out, dtype=float)   # (2,)
        self.eta = float(eta)

        self.x = None
        self.Ah = None
        self.h = None
        self.Ay = None
        self.y = None

    def forward(self, x):
        self.x = np.array(x, dtype=float)
        self.Ah = self.x @ self.W_in_h           # (2,) = (2,)@(2,2)
        self.h  = sigmoid(self.Ah)               # (2,)
        self.Ay = float(self.h @ self.W_h_out)   # scalar
        self.y  = sigmoid(self.Ay)               # a scalar
        return self.Ah, self.h, self.Ay, self.y

    def train_pass(self, x, d):
        self.forward(x)

        e = d - self.y
        dE_de   = e
        de_dy   = -1.0
        dy_dAy  = d_sigmoid_from_y(self.y)
        common_out = dE_de * de_dy * dy_dAy      # shared chain-rule at output

        # grads wrt hidden->out weights
        dE_dW_h_out = common_out * self.h        # delA_y/delw_j = h_j

        # Hidden node gradients
        # For each hidden unit j:
        # delE/delw_ij = common_out * (delA_y/delh_j) * (delh_j/delA_hj) * (delA_hj/delw_ij)
        #          = common_out * W_h_out[j] * h_j*(1-h_j) * x_i
        back_to_hidden = common_out * self.W_h_out * d_sigmoid_from_y(self.h)   # shape (2,)
        dE_dW_in_h = np.outer(self.x, back_to_hidden)                           # (2,2)


        print("\nForward")
        print(f"x = {self.x}")
        print(f"Hidden activities Ah = {self.Ah}")
        print(f"Hidden activations  h = {self.h}")
        print(f"Output activity  Ay = {self.Ay:.6f}")
        print(f"output activation y = {self.y:.6f}")
        print("\nOutput node grads")
        print(f"[delE/dele={dE_de:.6f}, dele/dely={de_dy:.1f}, dely/delAy={dy_dAy:.6f}]")
        print(f"dE/dW_h_out (h1->y, h2->y) = {dE_dW_h_out}")
        print("\nHidden node grads")
        print(f"back_to_hidden (each hidden unit) = {back_to_hidden}")
        print("dE/dW_in_h matrix =")
        print(dE_dW_in_h)

        # Gradient descent updates
        self.W_h_out -= self.eta * dE_dW_h_out
        self.W_in_h  -= self.eta * dE_dW_in_h

        print("\nUpdated weights")
        print(f"W_h_out = {self.W_h_out}")
        print("W_in_h =")
        print(self.W_in_h)

    def one_bp_update(self, x, d):
        Ah, h, Ay, y = self.forward(x)

        # output grads
        common_out = (d - y) * (-1.0) * (y * (1 - y))
        dE_dW_h_out = common_out * h

        # hidden grads
        back_to_hidden = common_out * self.W_h_out * (h * (1 - h))
        dE_dW_in_h = np.outer(np.array(x, float), back_to_hidden)

        # updates
        W_in_h_old  = self.W_in_h.copy()
        self.W_h_out -= self.eta * dE_dW_h_out
        self.W_in_h  -= self.eta * dE_dW_in_h
        return (Ah, h, Ay, y, W_in_h_old)

# Run one pass with the values  x1=1, x2=3 , desired output d=0.95 ; eta=0.1 , no biases
if __name__ == "__main__":
    x = np.array([1.0, 3.0], dtype=float)
    d = 0.95
    eta = 0.1

    # weights
    W_in_h = [[0.8, 0.5],
             [0.1, 0.2]]

    # hidden to output weights
    W_h_out = [0.2, 0.7]

    net = Network(W_in_h, W_h_out, eta)
    net.train_pass(x, d)

    # 1вЂ"3) Initial activations
    net = Network(W_in_h, W_h_out, eta)
    Ah, h, Ay, y = net.forward(x)
    print("Answer to #1:", f"{h[0]:.4g}")  # Node 1 activation
    print("Answer to #2:", f"{h[1]:.4g}")  # Node 2 activation
    print("Answer to #3:", f"{y:.4g}")     # output activation

    # 4вЂ"5) After one backprop updte toward d=0.95
    net = Network(W_in_h, W_h_out, eta)
    Ah, h, Ay, y, W_in_h_old = net.one_bp_update(x, d)
    print("Answer to #4 (w h1->y):", f"{net.W_h_out[0]:.4g}")
    print("Answer to #5 (w h2->y):", f"{net.W_h_out[1]:.4g}")

    # delta for x1->h2
    delta_w_x1_h2 = net.W_in_h[0,1] - W_in_h_old[0,1]
    print("Answer to #6 (delta w x1->h2):", f"{delta_w_x1_h2:.4g}")
```