```python
# Feed Forward Back Propagation algorithm with two inputs and a hidden node by Andrew Taylor

import numpy as np


#helpers
def sigmoid(activity):
    """Sigmoid f(A) = 1/(1+e^{-A}). Derivative is y(1-y)."""
    return 1.0 / (1.0 + np.exp(-activity))

def sigmoid_derivative_from_output(y):
    """df/dA = y(1-y)."""
    return y * (1.0 - y)



# Perceptron: 2 inputs -> 1 node
class Perceptron:
    """Single neuron with weights, bias, activity (A), activation (y).

    Chain rule (FFBP handout) for OUTPUT node:
      E = 1/2 (d - y)^2,  e = d - y
      delE/delw_i = (delE/dele)(dele/dely)(dely/delA)(delA/delw_i)
              = (d - y) * (-1) * y(1 - y) * x_i
      delE/deltheta   = (d - y) * (-1) * y(1 - y) * 1
    """

    def __init__(self, weights, bias):
        self.weights = np.array(weights, dtype=float)  # shape: (2,)
        self.bias = float(bias)
        self.activity = 0.0
        self.activation = 0.0

    def forward(self, inputs):
        """Compute A = wB·x + theta and y = sigma(A)."""
        self.activity = float(np.dot(self.weights, inputs) + self.bias)
        self.activation = sigmoid(self.activity)
        return self.activation

    def output_gradients(self, desired_output, inputs):
        """Return (grad_w, grad_b) using the 4-factor/3-factor chain rule."""
        e = desired_output - self.activation            # error signal
        dE_de = e                                       # delE/dele
        de_dy = -1.0                                    # dele/dely
        dy_dA = sigmoid_derivative_from_output(self.activation)  # dely/delA
        common = dE_de * de_dy * dy_dA                  # first three factors (they are shared)
        grad_w = common * inputs                        # delA/delw_i = x_i (the input vector is the coeffiecient when we take the partial)
        grad_b = common * 1.0                           # delA/deltheta = 1 (the derivative of theta w.r.t. itself)
        return grad_w, grad_b, (dE_de, de_dy, dy_dA)    #  printing


# NetworkLayer
class NetworkLayer:
    def __init__(self, perceptron):
        self.perceptron = perceptron

    def forward(self, inputs):
        return self.perceptron.forward(inputs)



# for 2->1 network
class TwoInputOneNodeNet:
    """Two inputs feeding a single output node (one perceptron)."""
    def __init__(self, input_weights, bias, learning_rate):
        self.layer = NetworkLayer(Perceptron(input_weights, bias))
        self.learning_rate = float(learning_rate)

    def forward(self, inputs):
        return self.layer.forward(inputs)

    def train_pass(self, inputs, desired_output, print_every= True):
        """One Perceptrondelta update (steepest descent)."""
        y = self.forward(inputs)
        p = self.layer.perceptron

        grad_w, grad_b, (dE_de, de_dy, dy_dA) = p.output_gradients(desired_output, inputs)

        if print_every:
            # Existing prints
            print(f"Activity (sum) = {p.activity:.4g}, Activation (sigmoid) = {p.activation:.4g}")
            print(f"dE/dw_1 = {grad_w[0]:.4g}, dE/dw_2 = {grad_w[1]:.4g}, dE/dtheta = {grad_b:.4g}")
            # Transparency: show the three shared chain-rule factors
            print(f"[delE/dele={dE_de:.4g}, dele/dely={de_dy:.4g}, dely/dela={dy_dA:.4g}]")


        # Gradient descent update
        eta = self.learning_rate
        p.weights -= eta * grad_w
        p.bias    -= eta * grad_b

    def fit(self, inputs, desired_output, epochs):
        for ep in range(1, epochs + 1):
            print(f"\n Pass {ep}/{epochs}")
            self.train_pass(inputs, desired_output, print_every=True)

    def read_activity_activation(self, inputs):
        """Ensure forward has been run, then return (A, y)."""
        self.forward(inputs)
        p = self.layer.perceptron
        return p.activity, p.activation



if __name__ == "__main__":
    # Inputs and initial conditions
    inputs = np.array([0.8, 0.9], dtype=float)
    input_weights = np.array([0.24, 0.88], dtype=float)   # your 2 -> 1 weights
    bias = 0.0
    eta = 5.0

    # 1) Initial activation (no updates), desired=0.95
    net = TwoInputOneNodeNet(input_weights.copy(), bias, eta)
    A0, y0 = net.read_activity_activation(inputs)
    print("Answer to #1:")
    print(f"Initial (no update): activity = {A0:.4g}, activation = {y0:.4g}")

    # 2) 75 updates toward desired=0.95
    net = TwoInputOneNodeNet(input_weights.copy(), bias, eta)
    net.fit(inputs, desired_output=0.95, epochs=75)
    A75, y75 = net.read_activity_activation(inputs)
    print("Answer to #2:")
    print(f"\nAfter 75 passes: activity = {A75:.4g}, activation = {y75:.4g}")
```

```python
# 3) 30 updates toward desired=0.15
net = TwoInputOneNodeNet(input_weights.copy(), bias, eta)
net.fit(inputs, desired_output=0.15, epochs=30)
A30, y30 = net.read_activity_activation(inputs)
print("Answer to #3:")
print(f"\nAfter 30 passes: activity = {A30:.4g}, activation = {y30:.4g}")

# 4) Derive the partial of the Error w.r.t the bias
print("Answer to #4 is delE/deltheta (grad_b):")

y = 0.3
d = 0.4
e = d - y

# factors
dE_de = e
de_dy = -1.0
dy_dA = sigmoid_derivative_from_output(y)
dA_dtheta = 1.0

# Final
grad_b = dE_de * de_dy * dy_dA * dA_dtheta

print(
    f"Derivation for delE/deltheta:\n\n"
    f"Step 1: Error signal e = d - y = {d:.3f} - {y:.3f} = {e:.3f}\n"
    f"Step 2: delE/dele = e = {dE_de:.3f}\n"
    f"Step 3: dele/dely = -1\n"
    f"Step 4: dely/dela = y(1-y) = {y:.3f}(1-{y:.3f}) = {dy_dA:.3f}\n"
    f"Step 5: dela/deltheta = 1 (bias contributes additively)\n\n"
    f"So: delE/deltheta = (d - y)(-1)(y(1-y))(1)\n"
    f"                    = ({d:.3f} - {y:.3f})(-1)({dy_dA:.3f})(1)\n"
    f"                    = {grad_b:.3f}\n")
```