Backpropagation in a Simple Neural Network

Gregory Glickert

February 17, 2026

Introduction

This document demonstrates backpropagation in a simple feedforward neural network with 2 inputs, 2 hidden nodes, and 1 output node. All nodes use the sigmoid activation function. We will compute the forward pass step-by-step, then perform one backward pass using cross-entropy loss and gradient descent with learning rate 1.

What is Backpropagation?

Backpropagation is the fundamental algorithm for training neural networks. It consists of two phases:

1. **Forward Pass**: Input data flows through the network, activating each node in sequence, until we reach the output.
2. **Backward Pass**: We compute the error at the output and then propagate it backward through the network using the chain rule. This tells us how much each weight should change to reduce the error.

The key insight of backpropagation is that we can efficiently compute gradients for all weights using the chain rule, enabling us to update all weights in a single pass. This makes training deep networks feasible.

Verification

All calculations in this document have been verified using Python code (see `solve_network.py`). The numerical values presented have been computed to machine precision and rounded for display.

Network Setup

We work with a simple feedforward neural network with 2 inputs, 2 hidden nodes, and 1 output node.

$$x_1 = 0.35, \quad x_2 = 0.7$$

Weights:

$$w_{x1\to h1} = 0.2, \quad w_{x2\to h1} = 0.2, \quad w_{x1\to h2} = 0.3, \quad w_{x2\to h2} = 0.3$$

$$w_{h1\to o} = 0.3, \quad w_{h2\to o} = 0.9$$

Figure 1 shows the complete network architecture with all weights labeled. Before examining each layer's computation below, it is helpful to visualize the full network structure.

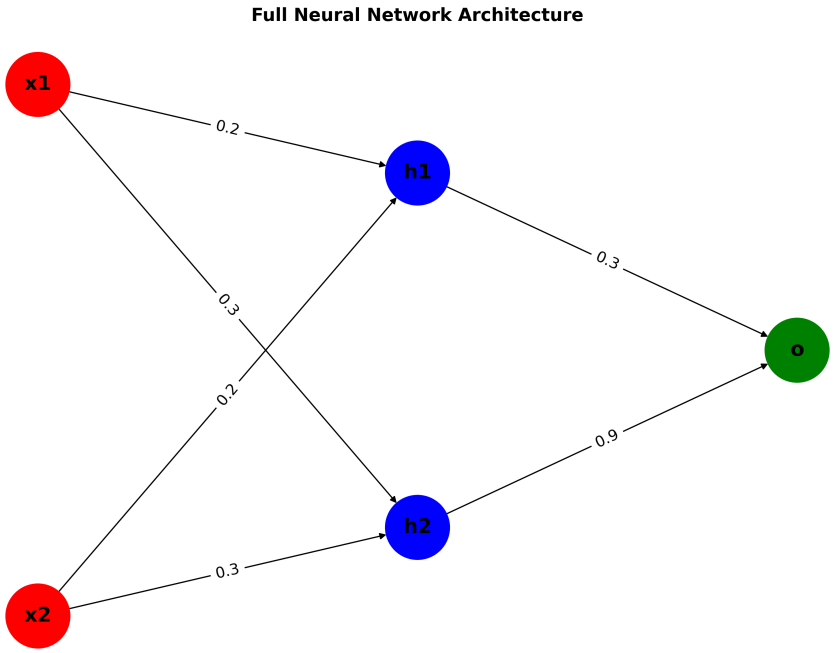

Figure 1: Complete Neural Network: 2 inputs $\to$ 2 hidden nodes $\to$ 1 output node

Target output: $y = 0.5$
All neurons use the sigmoid activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid derivative (needed for backpropagation):

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

No biases are used in this network.

Forward Pass

The forward pass computes the activation of each node by propagating inputs through the network, layer by layer. For each node, we first compute a weighted sum of its inputs (the pre-activation z), then apply the sigmoid activation function to produce the node's output.

Compute Hidden Nodes

Hidden Node 1 (h_1):

We compute the pre-activation sum for the first hidden node as a weighted sum of the inputs:

$$z_{h1} = x_1 w_{x1 \to h1} + x_2 w_{x2 \to h1}$$

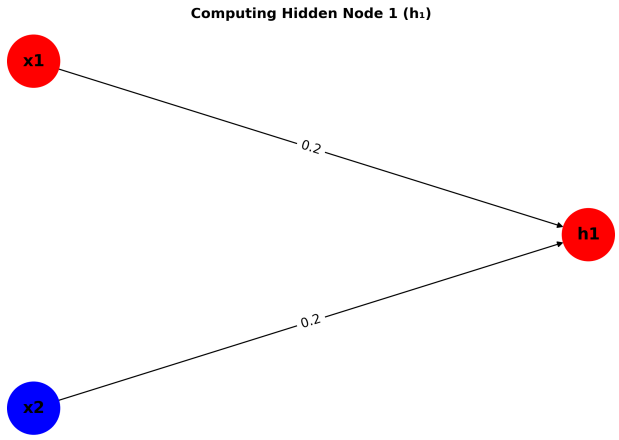

Figure 2: Sub-network for computing h_1

Substituting the input values and weights:

$$z_{h1} = (0.35)(0.2) + (0.7)(0.2) = 0.07 + 0.14 = 0.21$$

Apply the sigmoid activation function:

$$h_1 = \sigma(0.21) = \frac{1}{1 + e^{-0.21}} = 0.552308$$

Hidden Node 2 (h_2):

We apply the same computation for the second hidden node.
Pre-activation sum:

$$z_{h2} = x_1 w_{x1 \to h2} + x_2 w_{x2 \to h2} = (0.35)(0.3) + (0.7)(0.3) = 0.315$$

Apply sigmoid:

$$h_2 = \sigma(0.315) = 0.578105$$

Compute Output Node

We compute the final output layer using the same approach: weighted sum followed by sigmoid activation.

Pre-activation:

$$z_o = h_1 w_{h1 \to o} + h_2 w_{h2 \to o}$$

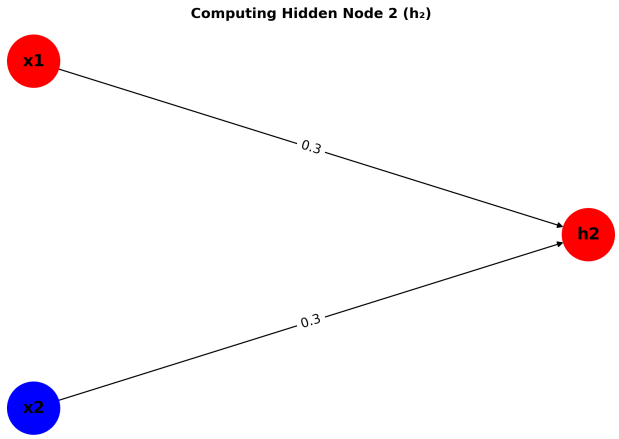

Figure 3: Sub-network for computing h_2

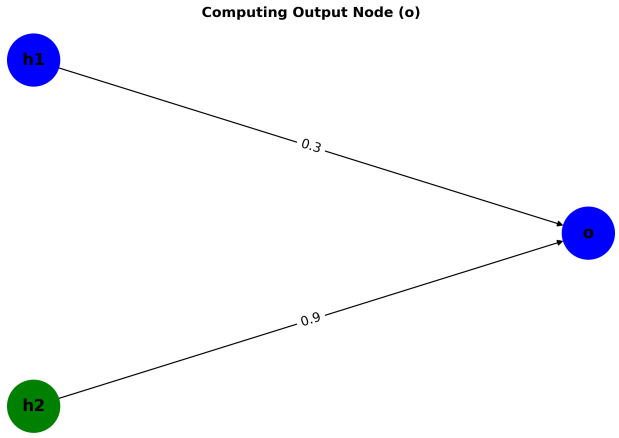

Figure 4: Sub-network for computing output o

Substituting values:

$$z_o = (0.552308)(0.3) + (0.578105)(0.9) = 0.165692 + 0.520295 = 0.685987$$

Apply sigmoid:

$$o = \sigma(0.685987) = 0.665074$$

Forward Pass Summary: The network predicts $o = 0.665074$, but our target is $y = 0.5$. The error is $0.665074 - 0.5 = 0.165074$ (positive, meaning output is too high). We will now perform the backward pass to compute how to update weights to reduce this error.

Backward Pass

In the backward pass, we compute how much each weight contributed to the error and adjust the weights to reduce that error. We use the chain rule to propagate error signals backward through the network.

Output Layer Error

First, we compute the error at the output node. For **Mean Squared Error (MSE) loss**, we need to include the sigmoid derivative in the delta:

$$\delta_o = (o - y) \cdot \sigma'(z_o)$$

The sigmoid derivative is:

$$\sigma'(z_o) = o(1 - o) = (0.665074)(1 - 0.665074) = (0.665074)(0.334926) = 0.222774$$

Thus:

$$\delta_o = (0.665074 - 0.5) \cdot 0.222774 = (0.165074) \cdot (0.222774) = 0.036770$$

This value is smaller than if we had used the raw error, which is appropriate because the sigmoid function saturates (its derivative is smaller in magnitude) when the pre-activation is away from zero.

Sigmoid Derivatives

To backpropagate errors to the hidden layer, we need the derivatives of the sigmoid function at each hidden node's pre-activation. Recall that the sigmoid derivative is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

For h_1:

$$\sigma'(z_{h1}) = h_1(1 - h_1) = (0.552308)(1 - 0.552308) = (0.552308)(0.447692) = 0.247264$$

For h_2:

$$\sigma'(z_{h2}) = h_2(1 - h_2) = (0.578105)(0.421895) = 0.243900$$

These derivatives measure how sensitive each hidden node's output is to changes in its pre-activation input.

Hidden Layer Deltas

Now we propagate the output error back to the hidden layer using the chain rule. The delta at each hidden node is the output error multiplied by the weight connecting it to the output, multiplied by the sigmoid derivative:

Hidden Node 1:

$$\delta_{h1} = \delta_o \cdot w_{h1 \to o} \cdot \sigma'(z_{h1})$$

$$\delta_{h1} = (0.165074)(0.3)(0.247264) = 0.012245$$

Hidden Node 2:

$$\delta_{h2} = \delta_o \cdot w_{h2 \to o} \cdot \sigma'(z_{h2}) = (0.165074)(0.9)(0.243900) = 0.036235$$

Notice that h_2 has a larger error signal than h_1 because it has a larger weight (0.9 vs 0.3) connecting to the output.

Weight Updates

Now that we have error signals at each layer, we can update the weights using gradient descent. The weight update rule is:

$$\Delta w = \eta \cdot \delta \cdot \text{input}$$

$$w_{\text{new}} = w_{\text{old}} - \Delta w$$

where η is the learning rate, δ is the error signal of the receiving neuron, and the input is the activation of the sending neuron. We **subtract** the update because we move opposite to the direction of the gradient to minimize the loss.

Input-to-Hidden Weights:

$$\Delta w_{x1\to h1} = \eta \cdot \delta_{h1} \cdot x_1 = (1.0)(0.002728)(0.35) = 0.000955$$

New weight: $w_{x1\to h1} = 0.2 - 0.000955 = 0.199045$

$$\Delta w_{x2\to h1} = (1.0)(0.002728)(0.7) = 0.001909$$

New weight: $w_{x2\to h1} = 0.2 - 0.001909 = 0.198091$

$$\Delta w_{x1\to h2} = (1.0)(0.008071)(0.35) = 0.002825$$

New weight: $w_{x1\to h2} = 0.3 - 0.002825 = 0.297175$

$$\Delta w_{x2\to h2} = (1.0)(0.008071)(0.7) = 0.005650$$

New weight: $w_{x2\to h2} = 0.3 - 0.005650 = 0.294350$

Hidden-to-Output Weights:

$$\Delta w_{h1\to o} = \eta \cdot \delta_o \cdot h_1 = (1.0)(0.036770)(0.552308) = 0.020309$$

New weight: $w_{h1\to o} = 0.3 - 0.020309 = 0.279691$

$$\Delta w_{h2\to o} = (1.0)(0.036770)(0.578105) = 0.021257$$

New weight: $w_{h2\to o} = 0.9 - 0.021257 = 0.878743$

All weights have been decreased (since all deltas are positive and we subtract), which makes sense because the output was too high and we need to make it smaller.

Second Forward Pass: Demonstrating Error Reduction

Now let's see if our weight updates actually reduced the error! We perform another forward pass using the updated weights.

Compute Hidden Nodes (Updated Weights)

$$z_{h1} = x_1 w_{x1\to h1} + x_2 w_{x2\to h1} = (0.35)(0.199045) + (0.7)(0.198091) = 0.208329$$
$$h_1 = \sigma(0.208329) = 0.551895$$

$$z_{h2} = (0.35)(0.297175) + (0.7)(0.294350) = 0.310056$$
$$h_2 = \sigma(0.310056) = 0.576899$$

Compute Output Node (Updated Weights)

$$z_o = h_1 w_{h1\to o} + h_2 w_{h2\to o} = (0.551895)(0.279691) + (0.576899)(0.878743) = 0.661306$$
$$o = \sigma(0.661306) = 0.659554$$

Error Comparison

Let's compare the performance before and after the weight update:

- **Before weight update:** Output $o = 0.665074$, Error $= |0.665074 - 0.5| = 0.165074$
- **After weight update:** Output $o = 0.659554$, Error $= |0.659554 - 0.5| = 0.159554$
- **Error reduction:** $0.165074 - 0.159554 = 0.005520$
- **Percent improvement:** $\frac{0.005520}{0.165074} \times 100\% = 3.34\%$

The error has decreased! Although the improvement is modest for a single iteration with learning rate $\eta = 1.0$, this demonstrates the fundamental principle of backpropagation: **we can compute gradients that reliably point toward reducing the error**. By repeating this process many times (multiple epochs) with smaller learning rates, we can progressively reduce the error and train the network to make accurate predictions.

This is the power of backpropagation—it automatically discovers how to adjust each weight to reduce the error, using the chain rule to efficiently compute gradients for all parameters.