The Backpropagation Algorithm

Training Neural Networks Through Gradient-Based Learning

Deep Learning for Perception
BSCS — FAST-NUCES

Week 04

Advance Organizer — What You'll Learn

What You Will Learn in This Lecture:
By the end of this session, you will understand:

1. What backpropagation is and why it is essential for training neural networks
2. The two-phase structure: Forward Pass and Backward Pass
3. How to compute outputs in the forward pass using weighted sums and activation functions
4. How to calculate errors and propagate them backward through the network
5. The mathematical derivation of weight update rules using gradient descent
6. Multiple complete worked examples showing all calculations step-by-step
7. How the chain rule of calculus enables efficient gradient computation

Prerequisites: You should be familiar with gradient descent, loss functions (MSE, cross-entropy), activation functions (sigmoid, ReLU), and basic feedforward neural network architecture.

1 Introduction: The Learning Problem in Neural Networks

1.1 What is Backpropagation?

Definition

Backpropagation (short for "backward propagation of errors") is an algorithm used to train multilayer neural networks by efficiently computing the gradient of the loss function with respect to each weight in the network. It uses the chain rule from calculus to propagate error signals backward from the output layer to earlier layers.

Why It Matters

Before backpropagation was developed in the 1980s, training neural networks with hidden layers was extremely difficult. The algorithm solved the **credit assignment problem**: How do we know which weights in earlier layers should be adjusted, and by how much, when we only observe the error at the output?
Backpropagation made deep learning possible by providing an efficient, systematic way to compute gradients for every parameter in networks with many layers.

1.2 The Challenge: Training Networks with Hidden Layers

In a single-layer perceptron, adjusting weights is straightforward because we can directly see how each input affects the output. But in a multilayer network:

- Hidden layer neurons don't have target values — we only know targets for output neurons
- Errors must somehow be "assigned" to hidden units based on their contribution to the output error
- We need to compute gradients for potentially millions of weights efficiently

Backpropagation solves this by computing gradients layer-by-layer, working backward from the output, using the chain rule to decompose the total error gradient into manageable pieces.

2 The Two-Phase Structure of Backpropagation

Backpropagation operates in two distinct phases for each training example:

Definition

Phase 1: Forward Pass
Computes the **functional signal** by propagating input values forward through the network, layer by layer, to produce output predictions.
Phase 2: Backward Pass
Computes the **error signal** by calculating the difference between predicted and target outputs, then propagating this error backward through the network to determine how each weight contributed to the error.

Analogy — Think of It Like This

Think of backpropagation like this:
Forward Pass = Delivering a package through a delivery network. The package (input data) moves forward through sorting centers (hidden layers) until it reaches the final destination (output layer).

Backward Pass = After delivery, if the package arrived at the wrong address or damaged, the delivery company traces the route backward to find where things went wrong at each sorting center, so they can fix the process.

2.1 Visual Representation: Forward Activity, Backward Error

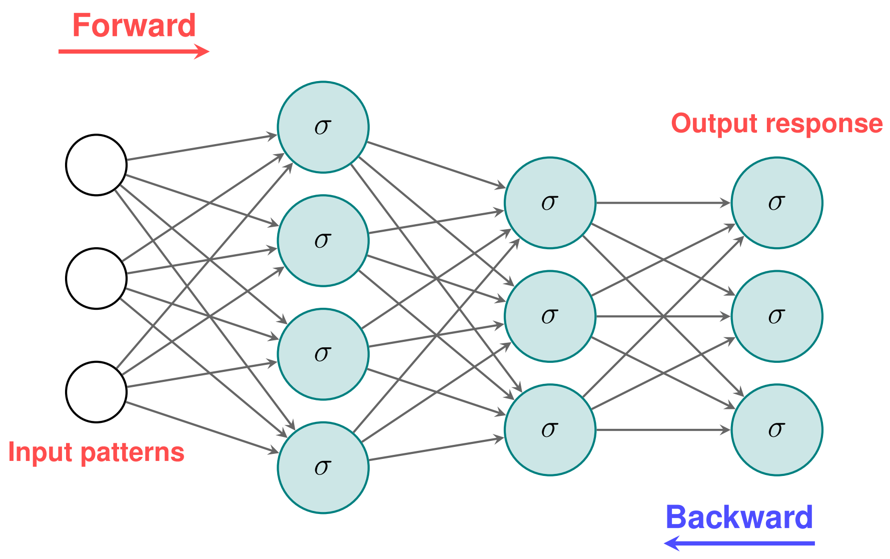

Figure 1: Conceptual view of backpropagation: Information flows forward during the forward pass (red arrow) to compute predictions. Errors flow backward during the backward pass (blue arrow) to update weights.

Memory Hook — Remember This!

Remember the directional flow:

- **Forward** = Data flows forward, predictions computed
- **Backward** = Errors flow backward, gradients computed

The key insight: We need **both** directions. Forward to make predictions, backward to learn from mistakes.

3 Mathematical Foundation

3.1 The Learning Objective

Backpropagation learns the weights for a multilayer network with a **fixed architecture** (predetermined number of layers and neurons). The goal is to minimize the error between network predictions and target values.

Definition

The backpropagation algorithm employs **gradient descent** to minimize the squared error between the network output values and the target values across all output units.

3.2 Error Function for Multiple Output Units

Since we're working with networks that may have multiple output units, we define the total error $E(\vec{w})$ as:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 \quad (1)$$

where:

- $\vec{w}$ represents all weights in the network
- D is the set of training examples
- outputs is the set of all output units in the network
- t_{kd} is the target value for output unit k on training example d
- o_{kd} is the actual output of unit k on training example d
- The factor $\frac{1}{2}$ simplifies derivatives

Memory Hook — Remember This!

Why sum over both training examples AND output units?
Because the total error depends on:

1. How wrong we are on each training example (sum over d)
2. How wrong each output neuron is (sum over k)

We want to minimize the error across **all examples** and **all outputs** simultaneously.

Numerical Example

Problem: Calculate the total error for a network with 2 output units given the following predictions and targets for 2 training examples:
Given Data:

- Example 1: $o_{11} = 0.7$, $o_{21} = 0.4$; Targets: $t_{11} = 1$, $t_{21} = 0$
- Example 2: $o_{12} = 0.3$, $o_{22} = 0.8$; Targets: $t_{12} = 0$, $t_{22} = 1$

Solution:
Using the error formula: $E = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$
Step 1: Calculate error for Example 1:

$$
\begin{aligned}
E_1 &= \frac{1}{2}[(t_{11} - o_{11})^2 + (t_{21} - o_{21})^2] \\
&= \frac{1}{2}[(1 - 0.7)^2 + (0 - 0.4)^2] \\
&= \frac{1}{2}[(0.3)^2 + (-0.4)^2] \\
&= \frac{1}{2}[0.09 + 0.16] \\
&= \frac{1}{2}[0.25] = 0.125
\end{aligned}
$$

Step 2: Calculate error for Example 2:

$$
\begin{aligned}
E_2 &= \frac{1}{2}[(t_{12} - o_{12})^2 + (t_{22} - o_{22})^2] \\
&= \frac{1}{2}[(0 - 0.3)^2 + (1 - 0.8)^2] \\
&= \frac{1}{2}[(-0.3)^2 + (0.2)^2] \\
&= \frac{1}{2}[0.09 + 0.04] \\
&= \frac{1}{2}[0.13] = 0.065
\end{aligned}
$$

Step 3: Total error:

$$E_{total} = E_1 + E_2 = 0.125 + 0.065 = \boxed{\mathbf{0.190}}$$

Interpretation: The total error of 0.190 represents how far our network's predictions are from the target values across both examples. Our goal in training is to minimize this error.

3.3 Notation and Terminology

Before diving into the algorithm, let's establish clear notation:

Symbol	Meaning
x_i	Input from unit i (or activation of unit i)
w_{ji}	Weight from unit i to unit j
net$_j$	Weighted sum of inputs to unit j: net$_j = \sum_i w_{ji} x_i$
o_j	Output (activation) of unit j: $o_j = \sigma(\text{net}_j)$
σ	Activation function (typically sigmoid)
t_j	Target value for output unit j
η	Learning rate (controls step size in gradient descent)
δ_j	Error term for unit j

Table 1: Notation used throughout backpropagation derivations

Numerical Example

Problem: Calculate the weighted sum (net input) for a neuron with the following inputs and weights:

Given Data:

- Inputs: $x_1 = 0.5$, $x_2 = 0.8$, $x_3 = 0.2$
- Weights: $w_{j1} = 0.4$, $w_{j2} = -0.3$, $w_{j3} = 0.6$
- Bias: $b_j = -0.2$

Solution:

The weighted sum formula is: $\text{net}_j = \sum_i w_{ji} x_i + b_j$

Step 1: Multiply each input by its corresponding weight:

$$\begin{aligned} x_1 \cdot w_{j1} &= 0.5 \times 0.4 = 0.20 \\ x_2 \cdot w_{j2} &= 0.8 \times (-0.3) = -0.24 \\ x_3 \cdot w_{j3} &= 0.2 \times 0.6 = 0.12 \end{aligned}$$

Step 2: Sum all weighted inputs and add bias:

$$\begin{aligned} \text{net}_j &= 0.20 + (-0.24) + 0.12 + (-0.2) \\ &= 0.20 - 0.24 + 0.12 - 0.2 \\ &= \boxed{\mathbf{-0.12}} \end{aligned}$$

Step 3: Apply sigmoid activation (if needed):

$$o_j = \sigma(\text{net}_j) = \frac{1}{1 + e^{-(-0.12)}} = \frac{1}{1 + e^{0.12}} = \frac{1}{1.1275} = \boxed{\mathbf{0.4700}}$$

Interpretation: The neuron receives a slightly negative weighted sum (-0.12), which after sigmoid activation produces an output of 0.47, slightly below the midpoint of 0.5.

Self-Test — Check Your Understanding

Quick Check:

1. What does the subscript notation w_{ji} mean? (Answer: weight FROM unit i TO unit j)
2. If a neuron has 3 inputs with values $[0.5, 0.2, 0.8]$ and weights $[0.3, -0.4, 0.6]$, what is net$_j$? (Answer: $0.5(0.3) + 0.2(-0.4) + 0.8(0.6) = 0.15 - 0.08 + 0.48 = 0.55$)

4 The Sigmoid Activation Function

4.1 Definition and Properties

Definition

The **sigmoid function** is defined as:

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

It maps any real number to the range $(0, 1)$, making it useful for modeling probabilities and binary classifications.

4.2 Important Property: The Sigmoid Derivative

One of the most important properties of the sigmoid function is its derivative:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

This elegant form means we can compute the derivative using only the sigmoid's output value!

Numerical Example

Problem: Calculate the sigmoid activation and its derivative for different net input values.

Given Data: Calculate for: (a) net $= 0$, (b) net $= 2$, (c) net $= -1.5$

Solution:

Case (a): net = 0

$$\sigma(0) = \frac{1}{1+e^{0}} = \frac{1}{1+1} = \boxed{\mathbf{0.5}}$$
$$\sigma'(0) = \sigma(0)(1 - \sigma(0)) = 0.5 \times 0.5 = \boxed{\mathbf{0.25}}$$

Case (b): net = 2

$$\sigma(2) = \frac{1}{1+e^{-2}} = \frac{1}{1+0.1353} = \frac{1}{1.1353} = \boxed{\mathbf{0.8808}}$$
$$\sigma'(2) = 0.8808 \times (1 - 0.8808) = 0.8808 \times 0.1192 = \boxed{\mathbf{0.1050}}$$

Case (c): net = -1.5

$$\sigma(-1.5) = \frac{1}{1+e^{1.5}} = \frac{1}{1+4.4817} = \frac{1}{5.4817} = \boxed{\mathbf{0.1824}}$$
$$\sigma'(-1.5) = 0.1824 \times (1 - 0.1824) = 0.1824 \times 0.8176 = \boxed{\mathbf{0.1491}}$$

Key Observations:

- At net $= 0$, sigmoid gives 0.5 (midpoint) with maximum derivative 0.25
- Large positive values saturate toward 1 with small derivatives
- Large negative values saturate toward 0 with small derivatives
- The derivative is always positive and maximum at the center

5 The Backpropagation Algorithm: Step-by-Step

5.1 Algorithm Structure

Algorithm

BACKPROPAGATION$(training\_examples, \eta, n_{in}, n_{out}, n_{hidden})$
Input:

- Each training example is a pair $(\vec{x}, \vec{t})$, where $\vec{x}$ is the vector of network input values and $\vec{t}$ is the vector of target network output values
- η is the learning rate (e.g., 0.05)
- n_{in} is the number of network inputs
- n_{hidden} is the number of units in the hidden layer
- n_{out} is the number of output units
- The input from unit i into unit j is denoted x_{ji}
- The weight from unit i to unit j is denoted w_{ji}

Steps:

1. Create a feedforward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units
2. Initialize all network weights to small random numbers (e.g., between -0.05 and 0.05)
3. Until the termination condition is met, Do:
   a. For each $(\vec{x}, \vec{t})$ in $training\_examples$, Do:
      i. **Forward Pass:** Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network
      ii. **Backward Pass:**
         - For each network output unit k, calculate its error term δ_k:
$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$
         - For each hidden unit h, calculate its error term δ_h:
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k$$
      iii. **Update Weights:** Update each network weight w_{ji}:
$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Why It Matters

Why process one example at a time?
This is called **stochastic gradient descent** (SGD). Instead of computing the gradient over the entire training set (which can be slow), we:

- Update weights after each example
- Get noisy but faster updates
- Can escape local minima better than batch gradient descent

Modern variants use **mini-batches** — updating after small groups of examples to balance speed and stability.

6 The Training Process: Four Repeated Steps

Neural network training is an iterative process. We repeat the following four steps until the network converges:

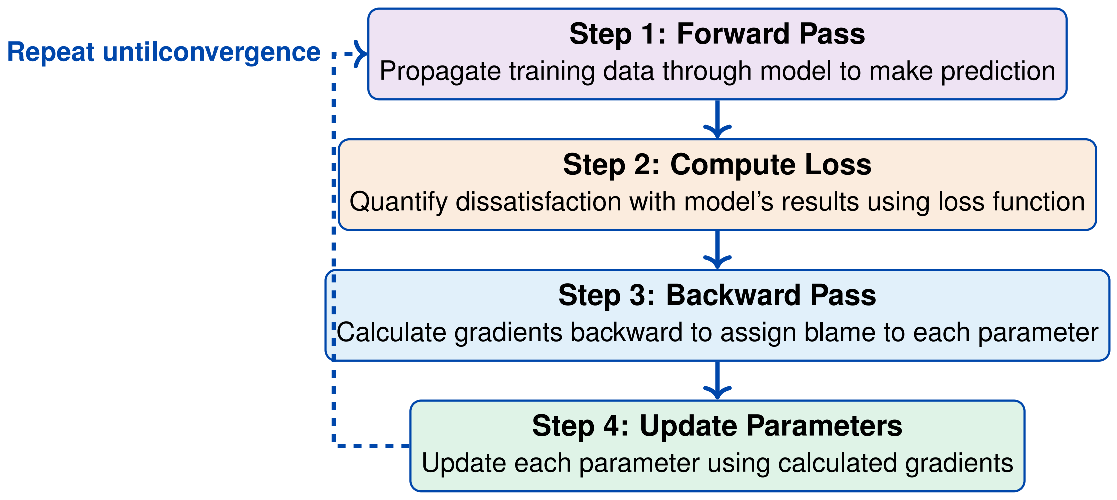
Repeat untilconvergence
Step 1: Forward Pass
Propagate training data through model to make prediction
Step 2: Compute Loss
Quantify dissatisfaction with model's results using loss function
Step 3: Backward Pass
Calculate gradients backward to assign blame to each parameter
Step 4: Update Parameters
Update each parameter using calculated gradients

Figure 2: The four-step training cycle repeated for each training example

6.1 Detailed Breakdown of Each Step

6.1.1 Step 1: Forward Pass

Goal: Compute the output of every neuron in the network given the current input.

Process:

1. Start with input layer: Set neuron activations to input values
2. For each subsequent layer, left to right:
   a. Compute weighted sum: $\text{net}_j = \sum_i w_{ji} x_i + b_j$ (where b_j is bias)
   b. Apply activation function: $o_j = \sigma(\text{net}_j)$
3. Store all intermediate activations (needed for backward pass)

6.1.2 Step 2: Compute Loss

Goal: Measure how far the network's prediction is from the target.

For regression tasks, we typically use **Mean Squared Error**:

$$E = \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$$

The derivative of this loss with respect to the output is:

$$\frac{\partial E}{\partial o_k} = -(t_k - o_k)$$

6.1.3 Step 3: Backward Pass

Goal: Compute the error signal δ_j for every neuron, which tells us how much that neuron contributed to the total error.

For output neurons:

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

This combines:

- $(t_k - o_k)$ — how wrong the output is
- $o_k(1 - o_k)$ — derivative of sigmoid activation

For hidden neurons:

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{downstream}} w_{kh} \delta_k$$

This weighted sum propagates errors from neurons in the next layer back to this neuron.

Numerical Example

Problem: Calculate the error terms (δ) for output and hidden neurons.
Given Data:

- Output neuron: $o_k = 0.8$, target $t_k = 1$
- Hidden neuron: $o_h = 0.6$

- Connections from hidden to output: weight $w_{kh} = 0.5$
- We already computed δ_k for the output neuron

Solution:
Step 1: Calculate output neuron error term δ_k:

$$\begin{aligned}\delta_k &= o_k(1 - o_k)(t_k - o_k)\\ &= 0.8 \times (1 - 0.8) \times (1 - 0.8)\\ &= 0.8 \times 0.2 \times 0.2\\ &= \boxed{\mathbf{0.032}}\end{aligned}$$

Step 2: Calculate hidden neuron error term δ_h:

$$\begin{aligned}\delta_h &= o_h(1 - o_h)\sum_k w_{kh}\delta_k\\ &= 0.6 \times (1 - 0.6) \times (0.5 \times 0.032)\\ &= 0.6 \times 0.4 \times 0.016\\ &= 0.24 \times 0.016\\ &= \boxed{\mathbf{0.00384}}\end{aligned}$$

Interpretation:

- The output neuron has a larger error term (0.032) because it's directly compared to the target
- The hidden neuron's error (0.00384) is much smaller because:
  1. It's proportional to the downstream error
  2. It's weighted by the connection strength (0.5)
  3. It's scaled by the derivative term $o_h(1 - o_h) = 0.24$

6.1.4 Step 4: Update Parameters

Goal: Adjust each weight to reduce the error.

$$w_{ji}^{\text{new}} = w_{ji}^{\text{old}} + \Delta w_{ji}$$

where the weight change is:

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

This means:

- Move in the direction that reduces error (gradient descent)
- Learning rate η controls step size
- Larger δ_j (bigger error) means bigger weight change

- Larger x_{ji} (more active input) means that weight is more responsible

Numerical Example

Problem: Update a weight given the error term, input activation, and learning rate.
Given Data:

- Current weight: $w_{ji} = 0.3$
- Error term for neuron j: $\delta_j = 0.15$
- Input activation from neuron i: $x_{ji} = 0.8$
- Learning rate: $\eta = 0.1$

Solution:
Step 1: Calculate weight change:

$$\begin{aligned}\Delta w_{ji} &= \eta \times \delta_j \times x_{ji}\\ &= 0.1 \times 0.15 \times 0.8\\ &= \boxed{\mathbf{0.012}}\end{aligned}$$

Step 2: Update weight:

$$\begin{aligned}w_{ji}^{\text{new}} &= w_{ji}^{\text{old}} + \Delta w_{ji}\\ &= 0.3 + 0.012\\ &= \boxed{\mathbf{0.312}}\end{aligned}$$

Interpretation:

- The weight increased from 0.3 to 0.312
- This positive change means strengthening this connection will reduce the error
- The magnitude of change (0.012) is small because:
  - Learning rate is conservative (0.1)
  - Error term is moderate (0.15)
  - We're taking small steps to avoid overshooting

Memory Hook — Remember This!

The Key Insight of Backpropagation:
We don't need to compute gradients from scratch for every weight. Instead:

1. Compute error at output layer (easy — we know the targets)
2. Use chain rule to propagate errors backward, reusing computations
3. Each layer's error depends on the next layer's error times the weights

This makes gradient computation **efficient** — linear in the number of weights instead of exponential!

7 Complete Worked Example: Step-by-Step Calculations

Let's work through a complete example to see every calculation. We'll use a small network with:

- 3 input units
- 2 hidden units
- 1 output unit
- Sigmoid activation function: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Learning rate: $\eta = 0.9$

7.1 Initial Network Configuration

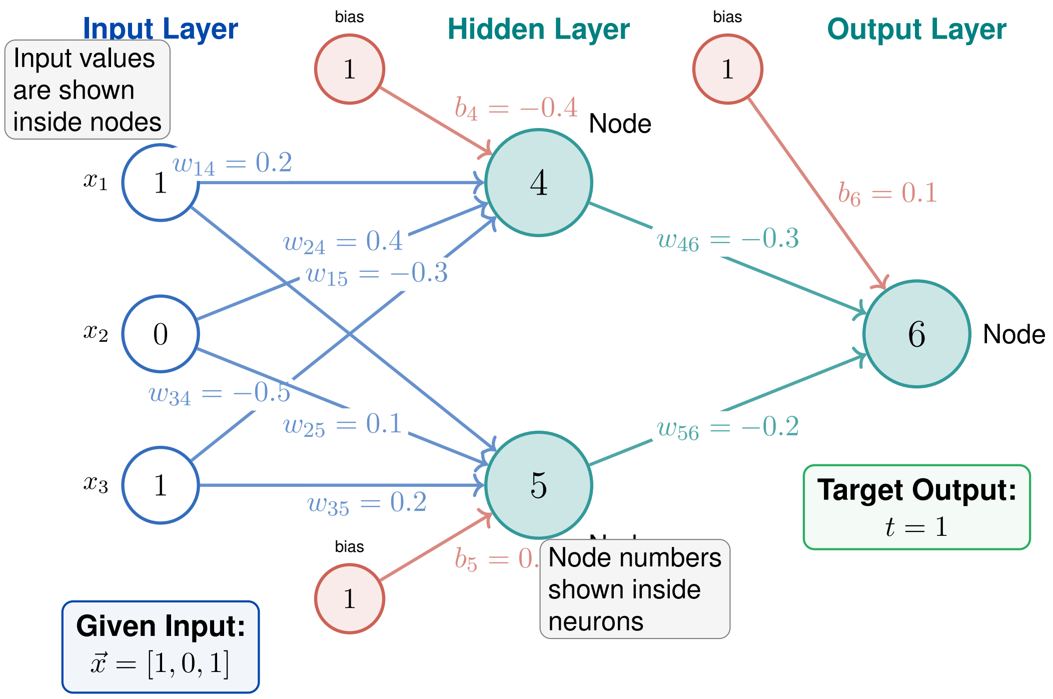

Figure 3: Initial network configuration with all weights and biases. The network has 3 inputs ($x_1 = 1, x_2 = 0, x_3 = 1$), 2 hidden neurons (nodes 4 and 5), and 1 output neuron (node 6). Target output is $t = 1$. All weights and biases are labeled on their respective connections.

Complete Backpropagation Example

Given Data:

- Input values: $x_1 = 1$, $x_2 = 0$, $x_3 = 1$
- Target output: $t = 1$
- Learning rate: $\eta = 0.9$
- Initial weights (as shown in diagram above)

Solution:

Part 1: Forward Pass - Computing Activations

Hidden Layer - Node 4:
Step : Compute weighted sum:

$$\begin{aligned}
\text{net}_4 &= x_1 w_{1,4} + x_2 w_{2,4} + x_3 w_{3,4} + b_4 \\
&= (1)(0.2) + (0)(0.4) + (1)(-0.5) + (-0.4) \\
&= 0.2 + 0 - 0.5 - 0.4 \\
&= -0.7
\end{aligned}$$

Step : Apply sigmoid activation:

$$\begin{aligned}
o_4 = \sigma(\text{net}_4) &= \frac{1}{1+e^{-(-0.7)}} = \frac{1}{1+e^{0.7}} \\
&= \frac{1}{1+2.0138} = \frac{1}{3.0138} \\
&= 0.332
\end{aligned}$$

Hidden Layer - Node 5:
Step : Compute weighted sum:

$$\begin{aligned}
\text{net}_5 &= x_1 w_{1,5} + x_2 w_{2,5} + x_3 w_{3,5} + b_5 \\
&= (1)(-0.3) + (0)(0.1) + (1)(0.2) + 0.2 \\
&= -0.3 + 0 + 0.2 + 0.2 \\
&= 0.1
\end{aligned}$$

Step : Apply sigmoid activation:

$$\begin{aligned}
o_5 = \sigma(\text{net}_5) &= \frac{1}{1+e^{-0.1}} \\
&= \frac{1}{1+0.9048} = \frac{1}{1.9048} \\
&= 0.525
\end{aligned}$$

Output Layer - Node 6:
Step : Compute weighted sum:

$$\begin{aligned}
\text{net}_6 &= o_4 w_{4,6} + o_5 w_{5,6} + b_6 \\
&= (0.332)(-0.3) + (0.525)(-0.2) + 0.1 \\
&= -0.0996 - 0.105 + 0.1 \\
&= -0.1046
\end{aligned}$$

Step : Apply sigmoid activation:

$$\begin{aligned} o_6 = \sigma(\text{net}_6) &= \frac{1}{1+e^{-(-0.1046)}} = \frac{1}{1+e^{0.1046}} \\ &= \frac{1}{1+1.1103} = \frac{1}{2.1103} \\ &= 0.474 \end{aligned}$$

Forward Pass Summary:

- Hidden layer: $o_4 = 0.332$, $o_5 = 0.525$
- Output: $o_6 = 0.474$
- Error: $(t - o_6) = 1 - 0.474 = 0.526$

Part 2: Backward Pass - Computing Error Terms

Output Layer Error (Node 6):
Using formula: $\delta_k = o_k(1 - o_k)(t_k - o_k)$

$$\begin{aligned} \delta_6 &= o_6(1 - o_6)(t_6 - o_6) \\ &= 0.474(1 - 0.474)(1 - 0.474) \\ &= 0.474 \times 0.526 \times 0.526 \\ &= 0.1311 \end{aligned}$$

Hidden Layer Errors:
Using formula: $\delta_h = o_h(1 - o_h)\sum_k w_{kh}\delta_k$
Node 5:

$$\begin{aligned} \delta_5 &= o_5(1 - o_5) \cdot w_{65}\delta_6 \\ &= 0.525(1 - 0.525)(-0.2)(0.1311) \\ &= 0.525 \times 0.475 \times (-0.2) \times 0.1311 \\ &= -0.0065 \end{aligned}$$

Node 4:

$$\begin{aligned} \delta_4 &= o_4(1 - o_4) \cdot w_{64}\delta_6 \\ &= 0.332(1 - 0.332)(-0.3)(0.1311) \\ &= 0.332 \times 0.668 \times (-0.3) \times 0.1311 \\ &= -0.0087 \end{aligned}$$

Part 3: Weight Updates

Using formula: $w_{ji}^{\text{new}} = w_{ji}^{\text{old}} + \eta\delta_j x_{ji}$ with $\eta = 0.9$
Output Layer Weights:

$$\begin{aligned} w_{46}^{\text{new}} &= -0.3 + (0.9)(0.1311)(0.332) = -0.3 + 0.0391 = -0.261 \\ w_{56}^{\text{new}} &= -0.2 + (0.9)(0.1311)(0.525) = -0.2 + 0.0619 = -0.138 \\ b_6^{\text{new}} &= 0.1 + (0.9)(0.1311)(1) = 0.1 + 0.118 = 0.218 \end{aligned}$$

Hidden Layer Weights (into Node 4):

$$\begin{aligned}
w_{14}^{\text{new}} &= 0.2 + (0.9)(-0.0087)(1) = 0.2 - 0.0078 = 0.192\\
w_{24}^{\text{new}} &= 0.4 + (0.9)(-0.0087)(0) = 0.4 + 0 = 0.4\\
w_{34}^{\text{new}} &= -0.5 + (0.9)(-0.0087)(1) = -0.5 - 0.0078 = -0.508\\
b_{4}^{\text{new}} &= -0.4 + (0.9)(-0.0087) = -0.4 - 0.0078 = -0.408
\end{aligned}$$

Hidden Layer Weights (into Node 5):

$$\begin{aligned}
w_{15}^{\text{new}} &= -0.3 + (0.9)(-0.0065)(1) = -0.3 - 0.0059 = -0.306\\
w_{25}^{\text{new}} &= 0.1 + (0.9)(-0.0065)(0) = 0.1 + 0 = 0.1\\
w_{35}^{\text{new}} &= 0.2 + (0.9)(-0.0065)(1) = 0.2 - 0.0059 = 0.194\\
b_{5}^{\text{new}} &= 0.2 + (0.9)(-0.0065) = 0.2 - 0.0059 = 0.194
\end{aligned}$$

Verification: Does the error reduce?

Using the updated weights, let's compute the new output:
Hidden layer with new weights:

$$\begin{aligned}
\text{net}_{4}^{\text{new}} &= 1(0.192) + 0(0.4) + 1(-0.508) + (-0.408) = -0.724\\
o_{4}^{\text{new}} &= \sigma(-0.724) = 0.327\\
\text{net}_{5}^{\text{new}} &= 1(-0.306) + 0(0.1) + 1(0.194) + 0.194 = 0.082\\
o_{5}^{\text{new}} &= \sigma(0.082) = 0.521
\end{aligned}$$

Output with new weights:

$$\begin{aligned}
\text{net}_{6}^{\text{new}} &= 0.327(-0.261) + 0.521(-0.138) + 0.218 = 0.061\\
o_{6}^{\text{new}} &= \sigma(0.061) = 0.515
\end{aligned}$$

Error Comparison:

- Old error: $|1 - 0.474| = 0.526$
- New error: $|1 - 0.515| = 0.485$
- **Error reduced by: 0.041** (Success!)

The network has learned! After one training iteration, the output moved from 0.474 closer to the target of 1.0.

Weight	**Old Value**	**New Value**	**Change**
w_{14}	0.200	0.192	-0.008
w_{24}	0.400	0.400	0.000
w_{34}	-0.500	-0.508	-0.008
w_{15}	-0.300	-0.306	-0.006
w_{25}	0.100	0.100	0.000
w_{35}	0.200	0.194	-0.006
w_{46}	-0.300	-0.261	+0.039
w_{56}	-0.200	-0.138	+0.062
b_4	-0.400	-0.408	-0.008
b_5	0.200	0.194	-0.006
b_6	0.100	0.218	+0.118
Output	**0.474**	**0.515**	**+0.041**
Error	**0.526**	**0.485**	**-0.041**

Table 2: Summary of weight updates after one training iteration. Note how weights connected to the output layer changed the most, and weights from inactive inputs ($x_2 = 0$) didn't change.

Key Observations:

- Weights connected to the output layer changed the most (larger error signal δ_6)
- Weights from inactive inputs ($x_2 = 0$) didn't change
- All changes push the network toward producing output closer to target $t = 1$
- We would repeat this process for all training examples, cycling through the dataset multiple times (epochs) until convergence

Self-Test — Check Your Understanding

Test Your Understanding:

1. Why did w_{24} and w_{25} not change? (Answer: Because $x_2 = 0$, so $\Delta w = \eta\delta x_{ji} = \eta\delta \times 0 = 0$)
2. Which weight changed the most and why? (Answer: b_6 (bias of output unit) changed the most because it has the largest error signal δ_6 and is always "active" (multiplied by 1))
3. If we ran another forward pass with the updated weights, would the output be closer to 1? (Answer: Yes! We verified it: output improved from 0.474 to 0.515, reducing error from 0.526 to 0.485)

8 Mathematical Derivation: Where Do the Formulas Come From?

In this section, we'll derive the backpropagation update rules from first principles using calculus. This explains **why** the formulas work.

8.1 The Gradient Descent Framework

Recall that in stochastic gradient descent, for each training example d, we update each weight by:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

where E_d is the error for training example d:

$$E_d = \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$$

Our task is to compute $\frac{\partial E_d}{\partial w_{ji}}$ for every weight in the network.

8.2 Using the Chain Rule

The key insight is to use the **chain rule** to decompose this derivative:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ji}}$$

Let's compute each piece:

Part 1: $\frac{\partial \text{net}_j}{\partial w_{ji}}$

Since $\text{net}_j = \sum_i w_{ji} x_i$, we have:

$$\frac{\partial \text{net}_j}{\partial w_{ji}} = x_{ji}$$

Part 2: $\frac{\partial E_d}{\partial \text{net}_j}$

This is more complex and depends on whether unit j is an output unit or a hidden unit. Let's define:

$$\delta_j \equiv -\frac{\partial E_d}{\partial \text{net}_j}$$

Then our weight update becomes:

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Now we just need to derive δ_j for output and hidden units.

8.3 Derivation for Output Units

For an output unit k, we need to compute:

$$\delta_k = -\frac{\partial E_d}{\partial \text{net}_k}$$

Using the chain rule:

$$\frac{\partial E_d}{\partial \text{net}_k} = \frac{\partial E_d}{\partial o_k} \cdot \frac{\partial o_k}{\partial \text{net}_k}$$

Step 1: Compute $\frac{\partial E_d}{\partial o_k}$

$$\begin{aligned} E_d &= \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2 \\ \frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j} \left[\frac{1}{2}(t_j - o_j)^2 \right] \\ &= \frac{1}{2} \cdot 2(t_j - o_j) \cdot \frac{\partial}{\partial o_j}(t_j - o_j) \\ &= (t_j - o_j)(-1) \\ &= -(t_j - o_j) \end{aligned}$$

Step 2: Compute $\frac{\partial o_k}{\partial \text{net}_k}$

Since $o_k = \sigma(\text{net}_k)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$, we need the derivative of the sigmoid function. The sigmoid derivative has a convenient form:

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

Therefore:

$$\frac{\partial o_k}{\partial \text{net}_k} = o_k(1 - o_k)$$

Step 3: Combine using chain rule

$$\begin{aligned}
\delta_k &= -\frac{\partial E_d}{\partial \text{net}_k} \\
&= -\frac{\partial E_d}{\partial o_k} \cdot \frac{\partial o_k}{\partial \text{net}_k} \\
&= -(-(t_k - o_k)) \cdot o_k(1 - o_k) \\
&= (t_k - o_k) \cdot o_k(1 - o_k) \\
&= o_k(1 - o_k)(t_k - o_k)
\end{aligned}$$

Result: For output units,

$$\boxed{\delta_k = o_k(1 - o_k)(t_k - o_k)}$$

8.4 Derivation for Hidden Units

For a hidden unit h, the derivation is more involved because the error must be propagated back from all units in the next layer.

$$\delta_h = -\frac{\partial E_d}{\partial \text{net}_h}$$

The key observation: net$_h$ affects E_d indirectly through all the units k in the next layer that h connects to.

Using the chain rule:

$$\frac{\partial E_d}{\partial \text{net}_h} = \sum_{k \in \text{Downstream}(h)} \frac{\partial E_d}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial \text{net}_h}$$

Step 1: Compute $\frac{\partial \text{net}_k}{\partial \text{net}_h}$

Since net$_k = \sum_i w_{ki} o_i$ and $o_h = \sigma(\text{net}_h)$:

$$\frac{\partial \text{net}_k}{\partial \text{net}_h} = \frac{\partial \text{net}_k}{\partial o_h} \cdot \frac{\partial o_h}{\partial \text{net}_h} = w_{kh} \cdot o_h(1 - o_h)$$

Step 2: Substitute back

$$\begin{aligned}
\frac{\partial E_d}{\partial \text{net}_h} &= \sum_k \frac{\partial E_d}{\partial \text{net}_k} \cdot w_{kh} \cdot o_h(1 - o_h) \\
&= o_h(1 - o_h) \sum_k w_{kh} \frac{\partial E_d}{\partial \text{net}_k} \\
&= o_h(1 - o_h) \sum_k w_{kh}(-\delta_k) \\
&= -o_h(1 - o_h) \sum_k w_{kh} \delta_k
\end{aligned}$$

Therefore:

$$\begin{aligned}\delta_h &= -\frac{\partial E_d}{\partial \text{net}_h} \\ &= o_h(1-o_h)\sum_k w_{kh}\delta_k\end{aligned}$$

Result: For hidden units,

$$\boxed{\delta_h = o_h(1-o_h)\sum_{k\in\text{Downstream}(h)} w_{kh}\delta_k}$$

Memory Hook — Remember This!

The Beautiful Symmetry of Backpropagation:
Notice that both formulas have the same structure:

$$\delta_j = \underbrace{o_j(1-o_j)}_{\text{local derivative}} \times \underbrace{\text{error signal}}_{\text{from next layer}}$$

For output units, the error signal is simply $(t_k - o_k)$ (how wrong we are).
For hidden units, the error signal is $\sum_k w_{kh}\delta_k$ (weighted sum of errors from downstream units).
This recursive structure is what makes backpropagation efficient!

Numerical Example

Problem: Verify the chain rule calculation for a simple 2-node network.
Given Data:

- Simple network: Input $x \to$ Hidden $h \to$ Output o
- $w_{xh} = 0.5$ (weight from input to hidden)
- $w_{ho} = 0.8$ (weight from hidden to output)
- Input: $x = 1$, Target: $t = 1$
- All biases = 0 for simplicity

Solution:
Step 1: Forward pass

$$\begin{aligned}\text{net}_h &= x \cdot w_{xh} = 1 \times 0.5 = 0.5 \\ o_h &= \sigma(0.5) = \frac{1}{1+e^{-0.5}} = 0.6225 \\ \text{net}_o &= o_h \cdot w_{ho} = 0.6225 \times 0.8 = 0.498 \\ o_o &= \sigma(0.498) = 0.622\end{aligned}$$

Step 2: Compute output error term

$$\begin{aligned}\delta_o &= o_o(1-o_o)(t-o_o)\\ &= 0.622(1-0.622)(1-0.622)\\ &= 0.622 \times 0.378 \times 0.378\\ &= 0.0889\end{aligned}$$

Step 3: Propagate error to hidden layer using chain rule

$$\begin{aligned}\delta_h &= o_h(1-o_h) \times w_{ho} \times \delta_o\\ &= 0.6225(1-0.6225) \times 0.8 \times 0.0889\\ &= 0.6225 \times 0.3775 \times 0.8 \times 0.0889\\ &= 0.0167\end{aligned}$$

Step 4: Update weights

$$\begin{aligned}\Delta w_{ho} &= \eta\delta_o o_h = 0.9 \times 0.0889 \times 0.6225 = 0.0498\\ w_{ho}^{\text{new}} &= 0.8 + 0.0498 = 0.8498\\ \Delta w_{xh} &= \eta\delta_h x = 0.9 \times 0.0167 \times 1 = 0.0150\\ w_{xh}^{\text{new}} &= 0.5 + 0.0150 = 0.5150\end{aligned}$$

Interpretation: The chain rule allowed us to compute how much to adjust w_{xh} even though it's two layers away from the output, by multiplying local derivatives at each step.

9 Practical Considerations and Common Issues

9.1 Stopping Criteria

When should we stop training? Common approaches:

1. **Fixed number of epochs**: Train for a predetermined number of passes through the data
2. **Validation error**: Stop when error on a held-out validation set stops decreasing
3. **Error threshold**: Stop when training error falls below a target value
4. **Patience-based early stopping**: Stop if validation error doesn't improve for N consecutive epochs

9.2 Learning Rate Selection

The learning rate η is critical:

- **Too large**: Training becomes unstable, weights oscillate or diverge
- **Too small**: Training is very slow, may get stuck in poor local minima
- **Typical values**: 0.01 to 0.5 for backpropagation with sigmoid
- **Modern approach**: Use adaptive learning rates (Adam, RMSprop) that adjust automatically

Numerical Example

Problem: Compare weight updates with different learning rates.
<u>**Given Data:**</u>

- Initial weight: $w = 0.5$
- Error term: $\delta = 0.2$
- Input activation: $x = 0.8$
- Test three learning rates: $\eta = 0.01$, $\eta = 0.1$, $\eta = 1.0$

<u>**Solution:**</u>
The weight change formula: $\Delta w = \eta\delta x$
Case 1: Small learning rate ($\eta = 0.01$)

$$\begin{aligned}\Delta w &= 0.01 \times 0.2 \times 0.8 = 0.0016 \\ w^{\text{new}} &= 0.5 + 0.0016 = 0.5016\end{aligned}$$

Case 2: Medium learning rate ($\eta = 0.1$)

$$\begin{aligned}\Delta w &= 0.1 \times 0.2 \times 0.8 = 0.016 \\ w^{\text{new}} &= 0.5 + 0.016 = 0.516\end{aligned}$$

Case 3: Large learning rate ($\eta = 1.0$)

$$\begin{aligned}\Delta w &= 1.0 \times 0.2 \times 0.8 = 0.16 \\ w^{\text{new}} &= 0.5 + 0.16 = 0.66\end{aligned}$$

Comparison Table:

η	Δw	w^{new}	**Effect**
0.01	0.0016	0.5016	Very slow, stable
0.1	0.016	0.516	Balanced
1.0	0.16	0.66	Fast, risk of overshooting

Interpretation:

- Small η: Takes many iterations but very stable
- Medium η: Good balance of speed and stability
- Large η: Fast changes but may overshoot optimal values

9.3 Weight Initialization

Why random small weights?

- **Symmetry breaking**: If all weights are identical, all hidden units learn the same thing
- **Activation range**: Small weights keep activations in the linear region of sigmoid initially
- **Gradient flow**: Prevents vanishing gradients at the start of training
- **Typical range**: $[-0.05, 0.05]$ or use specialized initialization (Xavier, He)

9.4 Common Problems and Solutions

Problem	Solutions
Vanishing Gradients	Use ReLU activation instead of sigmoid; use batch normalization; use residual connections
Exploding Gradients	Use gradient clipping; reduce learning rate; use batch normalization
Overfitting	Use dropout regularization; reduce network size; increase training data; use early stopping
Slow Convergence	Increase learning rate; use momentum; use adaptive optimizers (Adam); use batch normalization
Local Minima	Use momentum; random restarts; use better initialization; increase learning rate initially

Table 3: Common problems in training neural networks and their solutions

10 Example with Multiple Output Units

Backpropagation with 2 Output Units

Problem: Calculate one full iteration of backpropagation for a network with 2 inputs, 2 hidden units, and 2 output units.

Given Data:

Network Architecture:

- Input: $x_1 = 0.5$, $x_2 = 0.8$
- Targets: $t_1 = 1$, $t_2 = 0$
- Learning rate: $\eta = 0.5$

Initial Weights:

- Input to Hidden: $w_{11} = 0.15$, $w_{12} = 0.20$, $w_{21} = 0.25$, $w_{22} = 0.30$
- Hidden to Output 1: $w_{13} = 0.40$, $w_{23} = 0.45$
- Hidden to Output 2: $w_{14} = 0.50$, $w_{24} = 0.55$
- All biases = 0 for simplicity

<u>**Solution:**</u>

Forward Pass

Hidden Layer:

$$\begin{aligned}
\text{net}_1 &= 0.5(0.15) + 0.8(0.25) = 0.075 + 0.2 = 0.275 \\
h_1 &= \sigma(0.275) = 0.568 \\
\text{net}_2 &= 0.5(0.20) + 0.8(0.30) = 0.1 + 0.24 = 0.34 \\
h_2 &= \sigma(0.34) = 0.584
\end{aligned}$$

Output Layer:

$$\begin{aligned}
\text{net}_3 &= 0.568(0.40) + 0.584(0.45) = 0.227 + 0.263 = 0.49 \\
o_1 &= \sigma(0.49) = 0.620 \\
\text{net}_4 &= 0.568(0.50) + 0.584(0.55) = 0.284 + 0.321 = 0.605 \\
o_2 &= \sigma(0.605) = 0.647
\end{aligned}$$

Compute Error

Total error:

$$\begin{aligned}
E &= \frac{1}{2}[(t_1 - o_1)^2 + (t_2 - o_2)^2] \\
&= \frac{1}{2}[(1 - 0.620)^2 + (0 - 0.647)^2] \\
&= \frac{1}{2}[0.144 + 0.419] \\
&= 0.282
\end{aligned}$$

Backward Pass

Output Layer Deltas:

$$\begin{aligned}
\delta_3 &= o_1(1 - o_1)(t_1 - o_1) = 0.620(0.380)(0.380) = 0.0895 \\
\delta_4 &= o_2(1 - o_2)(t_2 - o_2) = 0.647(0.353)(-0.647) = -0.148
\end{aligned}$$

Hidden Layer Deltas:

$$
\begin{aligned}
\delta_1 &= h_1(1-h_1)(w_{13}\delta_3 + w_{14}\delta_4) \\
&= 0.568(0.432)(0.40 \times 0.0895 + 0.50 \times (-0.148)) \\
&= 0.245(0.0358 - 0.074) \\
&= 0.245(-0.0382) = -0.00936 \\
\delta_2 &= h_2(1-h_2)(w_{23}\delta_3 + w_{24}\delta_4) \\
&= 0.584(0.416)(0.45 \times 0.0895 + 0.55 \times (-0.148)) \\
&= 0.243(0.0403 - 0.0814) \\
&= 0.243(-0.0411) = -0.00999
\end{aligned}
$$

Weight Updates

Hidden to Output weights:

$$
\begin{aligned}
w_{13}^{\text{new}} &= 0.40 + 0.5 \times 0.0895 \times 0.568 = 0.4254 \\
w_{23}^{\text{new}} &= 0.45 + 0.5 \times 0.0895 \times 0.584 = 0.4761 \\
w_{14}^{\text{new}} &= 0.50 + 0.5 \times (-0.148) \times 0.568 = 0.4580 \\
w_{24}^{\text{new}} &= 0.55 + 0.5 \times (-0.148) \times 0.584 = 0.5068
\end{aligned}
$$

Input to Hidden weights:

$$
\begin{aligned}
w_{11}^{\text{new}} &= 0.15 + 0.5 \times (-0.00936) \times 0.5 = 0.1477 \\
w_{21}^{\text{new}} &= 0.25 + 0.5 \times (-0.00936) \times 0.8 = 0.2463 \\
w_{12}^{\text{new}} &= 0.20 + 0.5 \times (-0.00999) \times 0.5 = 0.1975 \\
w_{22}^{\text{new}} &= 0.30 + 0.5 \times (-0.00999) \times 0.8 = 0.2960
\end{aligned}
$$

Result Summary:

- Initial error: 0.282
- After weight updates, the network will produce outputs closer to $(1, 0)$
- Weights toward output 1 (target=1) increased
- Weights toward output 2 (target=0) decreased

11 Summary and Key Takeaways

Memory Hook — Remember This!

The Five Essential Ideas of Backpropagation:

1. **Two-Phase Process**: Forward pass computes predictions; backward pass computes gradients
2. **Chain Rule Decomposition**: Break complex gradient into simple local derivatives

3. **Error Propagation**: Output errors are propagated backward, weighted by connection strengths
4. **Efficient Computation**: Reuse forward pass activations during backward pass
5. **Gradient Descent**: Use computed gradients to update weights in the direction that reduces error

11.1 The Core Formulas

Backpropagation Formulas

Forward Pass:

$$\text{net}_j = \sum_i w_{ji} x_i$$

$$o_j = \sigma(\text{net}_j) = \frac{1}{1 + e^{-\text{net}_j}}$$

Output Layer Error:

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

Hidden Layer Error:

$$\delta_h = o_h(1 - o_h) \sum_{k \in \text{Downstream}(h)} w_{kh} \delta_k$$

Weight Update:

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j x_{ji}$$

11.2 Why Backpropagation Matters

Why It Matters

Backpropagation is the foundation of modern deep learning because it:

- Enables training of networks with many layers (deep learning)
- Scales to millions of parameters efficiently
- Works with various activation functions and architectures
- Can be extended to convolutional networks, recurrent networks, transformers, etc.
- Forms the basis of automatic differentiation in modern frameworks

Without backpropagation, we wouldn't have:

- Image recognition systems
- Natural language processing models
- Speech recognition

- Autonomous vehicles
- AlphaGo and modern game-playing AI

Every modern deep learning success builds on this fundamental algorithm!

12 Self-Test: Check Your Understanding

Self-Test — Check Your Understanding

Conceptual Questions:

1. What are the two main phases of backpropagation, and what does each compute?
2. Why can't we use the same error formula for hidden units as we do for output units?
3. Explain in your own words why backpropagation is called "backward propagation of errors."
4. What role does the chain rule play in backpropagation?
5. Why do we initialize weights to small random values instead of zeros?

Computational Questions:

6. Given a sigmoid neuron with net $= 0.5$, compute:
   (a) The output $o = \sigma(0.5)$
   (b) The derivative $o(1 - o)$
7. If an output neuron has $o = 0.8$, $t = 1$, compute the error term δ.
8. A hidden neuron has $o_h = 0.6$ and receives error from two downstream neurons:
   - $\delta_1 = 0.2$, $w_{1h} = 0.5$
   - $\delta_2 = -0.1$, $w_{2h} = -0.3$

   Compute δ_h.
9. Given $\delta_j = 0.15$, $x_{ji} = 0.8$, $\eta = 0.1$, and $w_{ji} = 0.3$, compute the new weight.

Answers:
1. Forward pass: computes predictions; Backward pass: computes gradients
2. Hidden units don't have target values; their error must be inferred from downstream errors
3. Errors at output layer propagate backward through the network, weighted by connections
4. Chain rule lets us decompose complex gradients into products of simple local derivatives
5. To break symmetry and avoid all neurons learning the same features
6a. $o = 1/(1 + e^{-0.5}) = 0.622$; 6b. $0.622(1 - 0.622) = 0.235$
7. $\delta = 0.8(1 - 0.8)(1 - 0.8) = 0.8 \times 0.2 \times 0.2 = 0.032$
8. $\delta_h = 0.6(1 - 0.6)[0.2(0.5) + (-0.1)(-0.3)] = 0.24[0.1 + 0.03] = 0.0312$

9. $w_{ji}^{\text{new}} = 0.3 + (0.1)(0.15)(0.8) = 0.3 + 0.012 = 0.312$

Further Reading

1. **Tom M. Mitchell**, *Machine Learning*, Chapter 4: Artificial Neural Networks
2. **Ian Goodfellow, Yoshua Bengio, and Aaron Courville**, *Deep Learning* (2016), Chapter 6: Deep Feedforward Networks
3. **Original Paper**: Rumelhart, Hinton, and Williams (1986), "Learning representations by back-propagating errors," *Nature*