Understanding Neural Networks and Backpropagation: Mathematical Foundations

September 7, 2025

1 Introduction

Neural networks are mathematical models that learn to map inputs to outputs by automatically adjusting internal parameters. Understanding how this parameter adjustment works—the mathematical principles behind neural network training—is fundamental to understanding one of the most important computational techniques in modern artificial intelligence.

This document explains how neural networks learn, focusing on a technique called backpropagation that enables efficient computation of parameter updates. The mathematics involves elegant applications of calculus, particularly the chain rule, combined with computational techniques that make training large networks feasible.

The core question we address is: How do neural networks automatically find parameter values that minimize prediction errors? We'll work through the mathematics step-by-step, using concrete numerical examples to illustrate abstract concepts. The chapter concludes with a short bridge from mean-squared error regression to the cross-entropy training used in modern large language models (LLMs).

2 The Learning Problem: Finding Patterns in Data

2.1 What Are We Trying to Accomplish?

To make the mathematics concrete, we'll use a simple example: predicting a risk score based on complexity. We have:

- **Input data** (x): A complexity score (ranging from 1–10)

- **Output data** (y): A risk score (any real number, where higher values indicate higher risk)
- **Goal**: Find a mathematical function that accurately maps complexity to risk scores

Note: For pedagogical clarity, we simplify to a single input feature. Real systems use multiple features simultaneously, but the same mathematical principles apply. We explore the multi-feature case in the appendix.

Our neural network will learn this mapping by adjusting internal parameters—specifically, weights (W_1 and W_2) and biases (b_1 and b_2) that determine how the input influences the final prediction.

2.2 The Architecture: A Three-Layer Neural Network

We'll examine a simple but representative neural network with three computational layers. Crucially, we structure our predictive function as a **composition of simpler functions**—this layered architecture is fundamental to how backpropagation works:

$$f(x) = W_2 \cdot \sigma(W_1 \cdot x + b_1) + b_2$$

where

$$\text{Layer 1:} \quad z_1 = W_1 \cdot x + b_1 \quad \text{(linear transformation)} \tag{1}$$

$$\text{Layer 2:} \quad a_1 = \sigma(z_1) \quad \text{(sigmoid activation)} \tag{2}$$

$$\text{Layer 3:} \quad y = W_2 \cdot a_1 + b_2 \quad \text{(final linear output)} \tag{3}$$

Key insights:

- The function is built as a **sequence of layers**, each applying a simple transformation.
- The sigmoid layer (σ) has no learnable parameters—it's a fixed mathematical function.
- Only the weights W_1, W_2 and biases b_1, b_2 are adjusted during learning.
- Each component function has a **known derivative**, which is essential for backpropagation.
- The output y can be any real number (positive or negative), representing our risk score.

3 Mathematical Foundations

3.1 The Sigmoid Function: Introducing Non-Linearity

The sigmoid function is crucial for neural networks because it introduces non-linearity while maintaining a smooth, differentiable form:

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

The sigmoid has several important properties:

- Always outputs values between 0 and 1
- Smooth, S-shaped curve that's differentiable everywhere
- Derivative can be computed efficiently: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Provides non-linear transformation essential for learning complex patterns

Why modern networks often favor ReLU/GELU (context). Sigmoid saturates for large $|x|$ (outputs near 0 or 1), making $\sigma'(x)$ tiny and slowing learning in deep stacks. Piecewise-linear (ReLU) and smooth Gaussian-shaped (GELU) activations maintain larger gradients over wider input ranges, enabling faster training and deeper models. We keep sigmoid here to make the calculus transparent; the same backpropagation logic applies to other activations.

3.2 The Chain Rule: Tracing Cause and Effect

If we have a composed function $f(g(x))$, then:

$$\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)).$$

Intuitively, the rate of change of the whole function equals the rate of change of the outer function times the rate of change of the inner function. This principle extends naturally to longer chains of composed functions.

4 The Learning Challenge: Why Direct Optimization Fails

4.1 Requirements for Backpropagation

Backpropagation presumes:

1. **Composed function structure**: the model is a sequence of differentiable layers.
2. **Known derivatives**: each layer's derivative is computable (e.g., sigmoid, tanh, ReLU).
3. **Differentiable path**: an unbroken chain from inputs to outputs.

These enable **automatic differentiation**[1].

4.2 The Naive Approach and Its Limits

One might write partial derivatives of the loss $L = \frac{1}{2}(y - \text{target})^2$ with respect to every parameter and expand them symbolically, but for realistic networks this becomes intractable (combinatorial growth of terms, numerical instability, deeply nested compositions).

5 Backpropagation: The Elegant Solution

5.1 Key Insight and Mechanics

Backpropagation applies the chain rule layer-by-layer, reusing forward-pass intermediates and working from the output layer backward.

5.2 The Forward Pass: Compute & Store

$$z_1 = W_1 \cdot x + b_1 \quad (\text{store } z_1) \tag{4}$$

$$a_1 = \sigma(z_1) \quad (\text{store } a_1) \tag{5}$$

$$y = W_2 \cdot a_1 + b_2 \quad (\text{store } y) \tag{6}$$

$$L = \tfrac{1}{2}(y - \text{target})^2 \tag{7}$$

[1] Automatic differentiation (autodiff) computes exact derivatives for programs by systematically applying the chain rule.

Bridge to LLM training (from regression to next-token prediction). In language modeling, the output is a vector of logits $z \in \mathbb{R}^{|V|}$ over a vocabulary V. A softmax converts logits to probabilities $p = \text{softmax}(z)$, and training minimizes the negative log-likelihood of the observed next token t:

$$L = -\log p_t, \qquad p_t = \frac{e^{z_t}}{\sum_{j \in V} e^{z_j}}.$$

Backpropagation proceeds identically in spirit; only the output layer and loss differ.

5.3 The Backward Pass: Efficient Gradients

Start with the loss gradient

$$\frac{\partial L}{\partial y} = y - \text{target}.$$

Final layer

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial y}\, a_1, \quad \frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y}.$$

Hidden layer

$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial y}\, W_2, \quad \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}\, a_1(1 - a_1), \quad \frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1}\, x, \quad \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}.$$

5.4 The Efficiency Breakthrough

Computation	Reuses from Forward Pass
$\frac{\partial L}{\partial W_2}$	stored a_1
$\frac{\partial L}{\partial a_1}$	stored W_2
$\frac{\partial L}{\partial z_1}$	stored a_1
$\frac{\partial L}{\partial W_1}$	stored x

Table 1: Computation Reuse in Backpropagation

This reuse keeps the cost within a small constant factor of the forward pass.

6 Parameter Updates and Learning Rates

6.1 Using the Gradients

With learning rate α,

$$W_1^{\text{new}} = W_1^{\text{old}} - \alpha \frac{\partial L}{\partial W_1}, \quad W_2^{\text{new}} = W_2^{\text{old}} - \alpha \frac{\partial L}{\partial W_2}, \tag{8}$$

$$b_1^{\text{new}} = b_1^{\text{old}} - \alpha \frac{\partial L}{\partial b_1}, \quad b_2^{\text{new}} = b_2^{\text{old}} - \alpha \frac{\partial L}{\partial b_2}. \tag{9}$$

We move *against* the gradient to decrease loss.

6.2 Learning Rate Tradeoffs

Small α gives slow, steady progress; large α risks divergence. Moderate values typically converge efficiently.

6.3 Worked Example: Forward and Backward Pass (Well-Behaved Initialization)

Setup: two training examples, processed sequentially (batch size = 1).

- Example 1: $x_1 = 5$, $\text{target}_1 = 0.7$
- Example 2: $x_2 = 3$, $\text{target}_2 = 0.3$

Initial parameters (chosen to avoid saturation): $W_1 = 0.2,\ W_2 = 0.5,\ b_1 = -1.0,\ b_2 = 0.0$.
Learning rate: $\alpha = 0.1$.

Example 1 (forward).

$$z_1 = 0.2 \cdot 5 - 1 = 0, \quad a_1 = \sigma(0) = 0.5, \quad y_1 = 0.5 \cdot 0.5 + 0 = 0.25,$$

$$L_1 = \tfrac{1}{2}(0.25 - 0.7)^2 = \tfrac{1}{2}(-0.45)^2 = 0.10125.$$

Example 1 (backward).

$$\frac{\partial L_1}{\partial y_1} = -0.45,\ \frac{\partial L_1}{\partial W_2} = -0.45 \cdot 0.5 = -0.225,\ \frac{\partial L_1}{\partial b_2} = -0.45,$$

$$\frac{\partial L_1}{\partial a_1} = -0.45 \cdot 0.5 = -0.225, \ \frac{\partial L_1}{\partial z_1} = -0.225 \cdot 0.5 \cdot (1 - 0.5) = -0.05625,$$

$$\frac{\partial L_1}{\partial W_1} = -0.05625 \cdot 5 = -0.28125, \quad \frac{\partial L_1}{\partial b_1} = -0.05625.$$

Update after Example 1:

$$W_1 = 0.228125, \ W_2 = 0.5225, \ b_1 = -0.994375, \ b_2 = 0.045.$$

Example 2 (forward with updated parameters).

$$z_1 = 0.228125 \cdot 3 - 0.994375 = -0.3100, \quad a_1 \approx \sigma(-0.3100) = 0.4230,$$

$$y_2 = 0.5225 \cdot 0.4230 + 0.045 \approx 0.2660, \quad L_2 = \tfrac{1}{2}(0.2660 - 0.3)^2 \approx 0.0006.$$

Example 2 (backward).

$$\frac{\partial L_2}{\partial y_2} = -0.0340, \ \frac{\partial L_2}{\partial W_2} \approx -0.0144, \ \frac{\partial L_2}{\partial b_2} = -0.0340,$$

$$\frac{\partial L_2}{\partial a_1} = -0.0340 \cdot 0.5225 \approx -0.0178, \quad a_1(1 - a_1) \approx 0.2440, \quad \frac{\partial L_2}{\partial z_1} \approx -0.0043,$$

$$\frac{\partial L_2}{\partial W_1} \approx -0.0130, \quad \frac{\partial L_2}{\partial b_1} \approx -0.0043.$$

Update after Example 2:

$$W_1 \approx 0.2294, \ W_2 \approx 0.5239, \ b_1 \approx -0.9939, \ b_2 \approx 0.0484.$$

Aside: what goes wrong with bad initializations (saturation). If instead $W_1 = 3, \ b_1 = 5, \ x = 5$, then $z_1 = 20$, so $a_1 = \sigma(20) \approx 1$ and $\sigma'(z_1) \approx 0$. Gradients flowing through $\sigma'(z_1)$ are nearly zero, and W_1, b_1 barely change. This is why we choose small weights and biases that keep pre-activations near 0 in shallow sigmoid networks.

6.4 Iterative Training

Repeat *forward* $\rightarrow$ *loss* $\rightarrow$ *backward* $\rightarrow$ *update* until convergence.

7 Exit Ticket (Answer in $\leq$ 1–2 sentences each)

1. Which intermediate quantities from the forward pass are explicitly reused during backpropagation, and why does reuse matter computationally?
2. If we swap the MSE head for softmax + cross-entropy, which parts of the gradient flow change and which remain structurally identical?
3. In one sentence, explain why sigmoid saturation slows learning in deeper networks.

8 Hands-On Micro-Exercises

1. Compute $\partial L/\partial W_2$ for Example 1 *by hand* before checking the expression above.
2. Re-run Example 1 with $\alpha = 0.01$ (mentally or roughly) and explain how the update magnitudes change.
3. In the vector extension, label the *shapes* (rows$\times$cols) of each Jacobian used to obtain $\partial L/\partial \mathbf{W_1}$.

A Extension to Multiple Features: The Vector Case

In practice, systems analyze multiple input features simultaneously. This section extends our scalar example to the more realistic vector case.

A.1 Multi-Feature Input

Instead of a single complexity score, consider a feature vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

A.2 Linear Layers (Dense Layers)

The first layer (dense/fully connected) is

$$\mathbf{z_1} = \mathbf{W_1}\mathbf{x} + \mathbf{b_1}, \quad \mathbf{W_1} \in \mathbb{R}^{2\times 3}.$$

A.3 Vector Derivatives and Gradients

Gradients become matrices:

$$\frac{\partial L}{\partial \mathbf{W_1}} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \cdots & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \cdots & \frac{\partial L}{\partial w_{23}} \end{bmatrix}.$$

A.4 Chain Rule for Vectors

$$\frac{\partial L}{\partial \mathbf{W_1}} = \frac{\partial L}{\partial \mathbf{z_1}}\, \mathbf{x}^\top.$$

B Batching and Mini-Batch Training

Rather than updating after each example, we can average gradients over a *batch* of examples. For batch size B,

$$L_{\text{batch}} = \frac{1}{B}\sum_{i=1}^{B} L_i, \qquad \frac{\partial L_{\text{batch}}}{\partial \theta} = \frac{1}{B}\sum_{i=1}^{B} \frac{\partial L_i}{\partial \theta}.$$

Averaging reduces gradient noise and improves stability; GPUs also exploit parallelism efficiently.

Small numeric illustration (using the well-behaved initialization). With W_1=0.2, W_2=0.5, b_1=−1 and the two examples above, Example 1 yields $\partial L/\partial(W_1, W_2, b_1, b_2) \approx (-0.2813, -0.2250, -0.0563, -0.4500$ at the initial point; Example 2 (evaluated at the *same* initial point) yields approximately $(-0.0358, -0.0399, -0.0119, -0.0993)$. The averaged batch gradient is therefore about $(-0.1585, -0.1329, -$ giving one update step:

$$W_1 \approx 0.2159,\ W_2 \approx 0.5133,\ b_1 \approx -0.9966,\ b_2 \approx 0.0275 \quad (\alpha = 0.1).$$

C Implementation in Python: Symbolic Mathematics with SymPy

C.1 Setting Up the Symbolic Framework

```
import sympy as sp
import numpy as np

# Define symbolic variables
x = sp.Symbol('x')            # Input feature
W1, W2 = sp.symbols('W1 W2')  # Weights
b1, b2 = sp.symbols('b1 b2')  # Biases
target = sp.Symbol('target')  # True label

# Define the sigmoid function symbolically
def sigmoid(z):
    return 1 / (1 + sp.exp(-z))

print("Symbolic variables created")
```

C.2 Constructing the Neural Network Symbolically

```
# Layer 1: Linear transformation
z1 = W1 * x + b1

# Layer 2: Sigmoid activation
a1 = sigmoid(z1)
```

```
# Layer 3: Final linear output (no sigmoid)
y = W2 * a1 + b2

# Loss function: Mean squared error
loss = sp.Rational(1, 2) * (y - target)**2
```

C.3 Automatic Differentiation with SymPy

```
# Compute gradients symbolically
grad_W1 = sp.diff(loss, W1)
grad_W2 = sp.diff(loss, W2)
grad_b1 = sp.diff(loss, b1)
grad_b2 = sp.diff(loss, b2)

print("L/W1 =", grad_W1)
print("L/W2 =", grad_W2)
print("L/b1 =", grad_b1)
print("L/b2 =", grad_b2)
```

C.4 Demonstrating the Chain Rule

```
# Manual chain rule computation for L/W1
dL_dy = sp.diff(loss, y)
dy_da1 = sp.diff(y, a1)
da1_dz1 = sp.diff(a1, z1)
dz1_dW1 = sp.diff(z1, W1)

dL_dW1_manual = dL_dy * dy_da1 * da1_dz1 * dz1_dW1
dL_dW1_direct = sp.diff(loss, W1)

print("Chain rule equal?",
      sp.simplify(dL_dW1_manual - dL_dW1_direct) == 0)
```

C.5 Numerical Evaluation (Well-Behaved Initialization)

```
# Non-saturating initial values
values = {
```

```
    x: 5,
    W1: 0.2,
    W2: 0.5,
    b1: -1.0,
    b2: 0.0,
    target: 0.7
}

prediction = y.subs(values)
loss_value = loss.subs(values)

print(f"Prediction: {float(prediction):.4f}")   # 0.2500
print(f"Loss: {float(loss_value):.4f}")         # 0.1013

print("Gradients at this point:")
print(f"dL/dW1 = {float(grad_W1.subs(values)):.4f}")  # -0.2813
print(f"dL/dW2 = {float(grad_W2.subs(values)):.4f}")  # -0.2250
print(f"dL/db1 = {float(grad_b1.subs(values)):.4f}")  # -0.0563
print(f"dL/db2 = {float(grad_b2.subs(values)):.4f}")  # -0.4500
```

For contrast only: Saturating initialization (do not use in the main example).

```
saturating = {
    x: 5,
    W1: 3,
    W2: -2,
    b1: 5,
    b2: -6,
    target: 0.7
}
a1_sat = a1.subs(saturating)
sigprime_sat = (a1 * (1 - a1)).subs(saturating)
print(f"a1(sat)  {float(a1_sat):.4f}")        #  1.0000
print(f"'(z1) (sat)  {float(sigprime_sat):.6f}")  #  0.0000
```

C.6 Exploring Sigmoid Properties

```
z = sp.Symbol('z')
sig = sigmoid(z)
sig_prime = sp.diff(sig, z)
print("Equal derivatives?",
      sp.simplify(sig_prime - sig*(1 - sig)) == 0)
```

D Real-World Implementation with PyTorch

D.1 Problem Setup: Non-Linear Pattern Recognition

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

torch.manual_seed(42)
np.random.seed(42)

X, y = make_circles(n_samples=1000, noise=0.1, factor=0.3, random_state=42)

X_tensor = torch.FloatTensor(X)
y_tensor = torch.FloatTensor(y).reshape(-1, 1) * 4 - 2  # Scale to [-2, 2]
```

D.2 Neural Network Definition

```
class RiskNet(nn.Module):
    def __init__(self, input_size=2, hidden_size=10, output_size=1):
        super(RiskNet, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.sigmoid1 = nn.Sigmoid()  # For pedagogy; in practice ReLU/GELU often prefer
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
```

```
        z1 = self.linear1(x)
        a1 = self.sigmoid1(z1)
        y = self.linear2(a1)
        return y

net = RiskNet()
```

D.3 Training Setup and Loss Function

```
criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.1)

num_epochs = 1000
batch_size = 32
num_batches = len(X_tensor) // batch_size

losses = []
tolerance_rates = []
```

D.4 Training Loop: Backpropagation in Action (with Tolerance-Rate Metric)

```
def tolerance_rate_metric(predictions, targets, tol=0.5):
    """Fraction of predictions within ±tol of target (regression proxy)."""
    return (torch.abs(predictions - targets) <= tol).float().mean().item()

for epoch in range(num_epochs):
    epoch_loss = 0.0
    epoch_tol = 0.0

    for b in range(num_batches):
        s = b * batch_size
        e = s + batch_size
        batch_X = X_tensor[s:e]
        batch_y = y_tensor[s:e]

        preds = net(batch_X)
```

```
        loss = criterion(preds, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_tol += tolerance_rate_metric(preds, batch_y)

    losses.append(epoch_loss / num_batches)
    tolerance_rates.append(epoch_tol / num_batches)

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1:4d} | Loss {losses[-1]:.4f} | ToleranceRate {tolerance_ra
```

D.5 Visualizing Training Progress and Results

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(losses)
ax1.set_title('Training Loss Over Time')
ax1.set_xlabel('Epoch'); ax1.set_ylabel('Loss'); ax1.grid(True)

ax2.plot(tolerance_rates)
ax2.set_title('Training Tolerance Rate (±0.5)')
ax2.set_xlabel('Epoch'); ax2.set_ylabel('Tolerance Rate'); ax2.grid(True)

plt.tight_layout(); plt.show()
```

D.6 Demonstrating Automatic Differentiation

```
single_x = X_tensor[0:1]
single_y = y_tensor[0:1]
pred = net(single_x)
loss = criterion(pred, single_y)

net.zero_grad()
```

```
loss.backward()

print("Gradients available for all parameters:")
for name, p in net.named_parameters():
    print(name, "grad shape:", None if p.grad is None else tuple(p.grad.shape))
```

Activation choice (practical note). For deeper networks, ReLU/GELU plus He/Glorot initialization help maintain gradient magnitudes and avoid saturation.