Neural Network Mathematics: A Complete Walkthrough for Beginners

From Logistic Regression to Backpropagation

Contents

1 Introduction and Setup

This document walks through the mathematical foundations of neural networks, starting from logistic regression and building up to backpropagation. We implement everything from scratch using NumPy to understand each component.

1.1 Required Libraries

```
import pandas as pd
import numpy as np
```

Listing 1: Import necessary libraries

Why these libraries?

- `numpy`: Essential for efficient mathematical operations and matrix computations
- `pandas`: Useful for data manipulation and analysis

2 Logistic Regression Foundation

Before diving into neural networks, we must understand logistic regression, which forms the building block of neural network outputs.

2.1 The Binary Cross-Entropy Loss Function

For binary classification, we use the binary cross-entropy (log loss) function:

Loss Function Definition

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)] \tag{1}$$

Understanding the components:

- N: Number of training examples
- y_i: Actual label (0 or 1) for example i
- $\hat{y}_i$: Predicted probability that $y_i = 1$
- The loss measures how "surprised" we are by the predictions

2.2 Deriving the Gradient

To optimize our model, we need to find $\nabla_{\mathbf{w}} \mathcal{L}$. Using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{w}} \tag{2}$$

2.2.1 Step 1: Derivative of Sigmoid Function

The sigmoid function is:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{3}$$

Deriving its derivative:

$$\frac{d\sigma}{dz} = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \tag{4}$$

$$= \frac{(1+e^{-z})\cdot 0 - 1\cdot(-e^{-z})}{(1+e^{-z})^2} \tag{5}$$

$$= \frac{e^{-z}}{(1+e^{-z})^2} \tag{6}$$

$$= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \tag{7}$$

$$= \frac{1}{1+e^{-z}} \cdot \frac{1+e^{-z}-1}{1+e^{-z}} \tag{8}$$

$$= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) \tag{9}$$

Important Result

$$\frac{d\sigma}{dz} = \sigma(z)(1-\sigma(z)) \tag{10}$$

Or equivalently: $\sigma'(z) = \sigma(z)\cdot(1-\sigma(z))$

2.2.2 Step 2: Derivative of Loss with Respect to Predictions

For a single example:

$$\frac{\partial L_i}{\partial z_i} = \frac{\partial L_i}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \tag{11}$$

$$= \left(-\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i}\right) \cdot \hat{y}_i(1-\hat{y}_i) \tag{12}$$

$$= -y_i(1-\hat{y}_i) + (1-y_i)\hat{y}_i \tag{13}$$

$$= -y_i + y_i\hat{y}_i + \hat{y}_i - y_i\hat{y}_i \tag{14}$$

$$= \hat{y}_i - y_i \tag{15}$$

Gradient with Respect to Predictions

$$\frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i \tag{16}$$

This beautifully simple result shows the error between prediction and truth!

2.2.3 Step 3: Complete Gradient

Given the design matrix $\mathbf{A}$ where each row is a training example:

$$\mathbf{A} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ 1 & x_{31} & x_{32} & \cdots & x_{3d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{Nd} \end{bmatrix} \in \mathbb{R}^{N \times (d+1)} \tag{17}$$

The gradient becomes:

Final Gradient Formula

$$\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{A}^T(\hat{\mathbf{y}} - \mathbf{y}) \tag{18}$$

where:

- $\mathbf{A}^T$: Transpose of design matrix (features)
- $\hat{\mathbf{y}}$: Vector of predictions
- $\mathbf{y}$: Vector of true labels

2.3 Implementation

```python
def sigmoid(z):
    """
    Compute sigmoid function: sigma(z) = 1 / (1 + exp(-z))

    Parameters:
    -----------
    z : array-like
        Input values

    Returns:
    --------
    array-like : Sigmoid of input values
    """
    return 1 / (1 + np.exp(-z))

def loss(y, y_hat):
    """
    Binary cross-entropy loss function

    Parameters:
    -----------
    y : array-like, shape (N,)
        True labels (0 or 1)
    y_hat : array-like, shape (N,)
        Predicted probabilities

    Returns:
    --------
    float : Average loss across all examples
```

```
    """
    N = len(y)
    # Add small epsilon to avoid log(0)
    epsilon = 1e-15
    y_hat_clipped = np.clip(y_hat, epsilon, 1 - epsilon)

    # Compute cross-entropy
    loss_value = -np.mean(
        y * np.log(y_hat_clipped) +
        (1 - y) * np.log(1 - y_hat_clipped)
    )
    return loss_value
```

Listing 2: Sigmoid and Loss Functions

2.4 Why Can't We Solve This Directly?

The Transcendental Equation Problem

The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ is a **transcendental function**, meaning:

1. **No Polynomial Form**: Unlike linear regression where loss is quadratic in $\mathbf{w}$, logistic regression involves exponentials and logarithms
2. **No Closed-Form Solution**: Setting $\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{0}$ gives us:

$$\mathbf{A}^T(\sigma(\mathbf{A}\mathbf{w}) - \mathbf{y}) = \mathbf{0} \tag{19}$$

   This equation cannot be solved algebraically for $\mathbf{w}$!
3. **Implicit Dependence**: The predictions $\hat{\mathbf{y}} = \sigma(\mathbf{A}\mathbf{w})$ depend on $\mathbf{w}$ through the sigmoid, making the equation **implicit** and impossible to solve directly

3 Gradient Descent: The Solution

Since we can't solve directly, we use **iterative optimization**.

3.1 The Algorithm

Gradient Descent Update Rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}) \tag{20}$$

where:

- $\mathbf{w}^{(t)}$: Parameters at iteration t
- α: Learning rate (step size)
- $\nabla_{\mathbf{w}}\mathcal{L}$: Gradient of loss $= \mathbf{A}^T(\hat{\mathbf{y}} - \mathbf{y})$

3.2 Understanding the Learning Rate α

Role of Learning Rate

The learning rate determines **how big our steps** are in parameter space:

- **Large** α: Fast convergence, but risk of overshooting the minimum
- **Small** α: Stable convergence, but slow progress
- **Just right** α: Efficient convergence to global optimum

Why we need it:

1. The gradient $\nabla_{\mathbf{w}}\mathcal{L}$ tells us the *direction* to move
2. The magnitude of $\nabla_{\mathbf{w}}\mathcal{L}$ depends on problem scale
3. α converts the gradient direction into an appropriate step size

3.3 Implementation

```python
def gradient_loss(A, y, y_hat):
    """
    Compute gradients of loss w.r.t. weights (w, b) with respect to
    the design matrix A and predictions.

    For logistic regression where:
        dw = (1/N) * A^T * (y_hat - y)  # Weights gradient
        db = (1/N) * sum(y_hat - y)     # Bias gradient

    Parameters:
    -----------
    A : ndarray, shape (N, d+1)
        Design matrix where:
        - N = number of training examples (rows)
        - d = number of features
        - First column is all ones (for bias)

    y : ndarray, shape (N,)
        True labels (0 or 1)

    y_hat : ndarray, shape (N,)
        Predicted probabilities

    Returns:
    --------
    tuple : (dw, db)
        dw : ndarray, shape (d,)
            Gradient w.r.t. feature weights
        db : float
            Gradient w.r.t. bias
    """
    N = A.shape[0]

    # Compute residuals (errors)
```

```
    residuals = y_hat - y

    # Gradient for weights (excluding bias column)
    # A[:, 1:] excludes the first column of ones
    dw = (1/N) * np.dot(A[:, 1:].T, residuals)

    # Gradient for bias (first column is all ones)
    db = (1/N) * np.sum(residuals)

    return dw, db
```

Listing 3: Gradient Computation for Logistic Regression

```
def train_model(A, y, learning_rate=0.01, num_iterations=1000):
    """
    Train logistic regression using gradient descent.

    Parameters:
    -----------
    A : ndarray, shape (N, d+1)
        Design matrix with bias column
    y : ndarray, shape (N,)
        True labels
    learning_rate : float
        Step size alpha for gradient descent
    num_iterations : int
        Number of training iterations

    Returns:
    --------
    tuple : (w, b, loss_history)
        w : ndarray
            Trained weights
        b : float
            Trained bias
        loss_history : list
            Loss value at each iteration
    """
    # Initialize parameters randomly
    np.random.seed(42)
    N, d_plus_1 = A.shape
    d = d_plus_1 - 1  # Number of features (excluding bias)

    w = np.random.randn(d) * 0.01  # Small random weights
    b = 0.0

    loss_history = []

    # Training loop
    for iteration in range(num_iterations):
        # Forward propagation
        z = np.dot(A[:, 1:], w) + b  # Linear combination
        y_hat = sigmoid(z)              # Predictions

        # Compute loss
        current_loss = loss(y, y_hat)
        loss_history.append(current_loss)

        # Backward propagation (compute gradients)
```

```
        dw, db = gradient_loss(A, y, y_hat)

        # Update parameters (gradient descent step)
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Print progress every 100 iterations
        if iteration % 100 == 0:
            print(f"Iteration {iteration}: Loss = {current_loss:.4f}")

    return w, b, loss_history
```

Listing 4: Training Loop with Gradient Descent

4 From Logistic Regression to Neural Networks

Now we extend logistic regression to a simple neural network with one hidden layer.

4.1 Network Architecture

Our neural network consists of:

1. **Input Layer**: $\mathbf{x} \in \mathbb{R}^2$ (2 features)
2. **Hidden Layer**: $\mathbf{h}^{(1)} \in \mathbb{R}^2$ (2 neurons with ReLU activation)
3. **Output Layer**: $z \in \mathbb{R}$ (single output with sigmoid activation)

4.2 Forward Propagation

4.2.1 Layer 1: Input to Hidden

Hidden Layer Computation

Linear Transformation:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{21}$$

where:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} \in \mathbb{R}^{2\times 2} \quad \text{(4 parameters)} \tag{22}$$

$$\mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \in \mathbb{R}^{2\times 1} \quad \text{(2 parameters)} \tag{23}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{24}$$

Activation Function (ReLU):

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{a}^{(1)}) = \begin{bmatrix} \max(0, a_1^{(1)}) \\ \max(0, a_2^{(1)}) \end{bmatrix} \tag{25}$$

4.2.2 Layer 2: Hidden to Output

Output Layer Computation

Linear Transformation:

$$z = \mathbf{w}^T \mathbf{h}^{(1)} + b = [w_1, w_2] \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \end{bmatrix} + b \tag{26}$$

where:

$$\mathbf{w} \in \mathbb{R}^{2\times 1} \quad (2 \text{ parameters}) \tag{27}$$

$$b \in \mathbb{R} \quad (1 \text{ parameter}) \tag{28}$$

Sigmoid Prediction:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1) \tag{29}$$

Total Parameters

Our network has:

- Hidden layer: $4 + 2 = 6$ parameters
- Output layer: $2 + 1 = 3$ parameters
- **Total: 9 trainable parameters**

4.3 Implementation

```
def relu(z):
    """
    ReLU activation function: max(0, z)

    Parameters:
    -----------
    z : array-like
        Input values

    Returns:
    --------
    array-like : ReLU(z) = max(0, z)
    """
    return np.maximum(0, z)

def relu_derivative(z):
    """
    Derivative of ReLU:
        dReLU/dz = 1 if z > 0
                 = 0 if z <= 0

    Parameters:
    -----------
    z : array-like
```

```
        Input values

    Returns:
    --------
    array-like : Gradient of ReLU
    """
    return (z > 0).astype(float)
```

Listing 5: ReLU Activation and Its Derivative

```
def model(x, w1, b1, w, b):
    """
    Forward pass through a 2-layer neural network.

    Architecture:
        Input (2) -> Hidden (2, ReLU) -> Output (1, Sigmoid)

    Parameters:
    -----------
    x : ndarray, shape (2,)
        Input features
    w1 : ndarray, shape (2, 2)
        Hidden layer weights
    b1 : ndarray, shape (2,)
        Hidden layer biases
    w : ndarray, shape (2,)
        Output layer weights
    b : float
        Output layer bias

    Returns:
    --------
    float : Predicted probability y_hat
    """
    # Hidden layer
    a1 = np.dot(w1, x) + b1           # Linear transformation
    h1 = relu(a1)                      # ReLU activation

    # Output layer
    z = np.dot(w, h1) + b              # Linear transformation
    y_hat = sigmoid(z)                 # Sigmoid activation

    return y_hat
```

Listing 6: Neural Network Model Definition

5 Backpropagation: Computing Gradients

Backpropagation is the algorithm for efficiently computing gradients in neural networks using the chain rule.

5.1 Overview

We need to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}}, \quad \frac{\partial \mathcal{L}}{\partial b}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} \tag{30}$$

Strategy: Work backwards from the loss, applying chain rule at each layer.

5.2 Step 1: Gradient at Output Layer

Output Layer Gradients

From earlier, we know:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y \tag{31}$$

Gradient w.r.t. output bias b:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial b} \tag{32}$$

$$= (\hat{y} - y) \cdot 1 \tag{33}$$

$$= \hat{y} - y \tag{34}$$

Gradient w.r.t. output weights w:
Since $z = \mathbf{w}^T \mathbf{h}^{(1)} + b = w_1 h_1^{(1)} + w_2 h_2^{(1)} + b$:

$$\frac{\partial z}{\partial w_i} = h_i^{(1)} \tag{35}$$

Therefore, in vector form:

$$\frac{\partial z}{\partial \mathbf{w}} = \mathbf{h}^{(1)} \tag{36}$$

Using chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{w}} \tag{37}$$

$$= (\hat{y} - y) \cdot \mathbf{h}^{(1)} \tag{38}$$

Boxed Results:

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \hat{y} - y} \tag{39}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = (\hat{y} - y) \cdot \mathbf{h}^{(1)}} \tag{40}$$

5.3 Step 2: Gradient at Hidden Layer Activations

Hidden Activation Gradients

We need $\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}}$ to backpropagate further.
Since $z = \mathbf{w}^T \mathbf{h}^{(1)} + b$:

$$\frac{\partial z}{\partial h_i^{(1)}} = w_i \tag{41}$$

In vector form:

$$\frac{\partial z}{\partial \mathbf{h}^{(1)}} = \mathbf{w} \tag{42}$$

By chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{h}^{(1)}} \tag{43}$$

$$= (\hat{y} - y) \cdot \mathbf{w} \tag{44}$$

Boxed Result:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z} \cdot \mathbf{w} = (\hat{y} - y) \cdot \mathbf{w}} \tag{45}$$

5.4 Step 3: Gradient Through ReLU Activation

ReLU Gradient

Recall: $\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{a}^{(1)}) = \max(0, \mathbf{a}^{(1)})$
The derivative is:

$$\frac{\partial h_i^{(1)}}{\partial a_i^{(1)}} = g'(a_i^{(1)}) = \begin{cases} 1 & \text{if } a_i^{(1)} > 0 \\ 0 & \text{if } a_i^{(1)} \leq 0 \end{cases} \tag{46}$$

Using element-wise multiplication (Hadamard product $\odot$):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{a}^{(1)}} \tag{47}$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot g'(\mathbf{a}^{(1)}) \tag{48}$$

In vector form:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot g'(\mathbf{a}^{(1)})} \tag{49}$$

where $g'(\mathbf{a}^{(1)})^T = [g'(a_1^{(1)}), g'(a_2^{(1)})]$.

5.5 Step 4: Gradient at First Layer Bias

First Layer Bias Gradient

Since $\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$:
For each component:

$$a_i^{(1)} = \sum_k W_{ik}^{(1)} x_k + b_i^{(1)} \tag{50}$$

Therefore:

$$\frac{\partial a_i^{(1)}}{\partial b_j^{(1)}} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{51}$$

This gives us:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}}} \tag{52}$$

Interpretation: The gradient flows directly through the bias without modification!

5.6 Step 5: Gradient at First Layer Weights

First Layer Weight Gradient

Since $\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$:
For each weight:

$$a_i^{(1)} = \sum_{k=1}^{2} W_{ik}^{(1)} x_k + b_i^{(1)} \tag{53}$$

Taking the derivative:

$$\frac{\partial a_i^{(1)}}{\partial W_{jk}^{(1)}} = \begin{cases} x_k & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{54}$$

Step-by-step for each weight:

$$\frac{\partial \mathcal{L}}{\partial W_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial W_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot x_1 \tag{55}$$

$$\frac{\partial \mathcal{L}}{\partial W_{12}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial W_{12}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot x_2 \tag{56}$$

$$\frac{\partial \mathcal{L}}{\partial W_{21}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial W_{21}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot x_1 \tag{57}$$

$$\frac{\partial \mathcal{L}}{\partial W_{22}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial W_{22}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot x_2 \tag{58}$$

In matrix form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot x_1 & \frac{\partial \mathcal{L}}{\partial a_1^{(1)}} \cdot x_2 \\ \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot x_1 & \frac{\partial \mathcal{L}}{\partial a_2^{(1)}} \cdot x_2 \end{bmatrix} \tag{59}$$

This can be written as an outer product:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \right) \mathbf{x}^T} \tag{60}$$

where $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}}$ is a column vector and $\mathbf{x}^T$ is a row vector.

5.7 Complete Backpropagation Implementation

```python
def gradients(x, y, w1, b1, w, b):
    """
    Compute all gradients for the neural network using backpropagation.

    Network: Input(2) -> Hidden(2, ReLU) -> Output(1, Sigmoid)

    Parameters:
    -----------
    x : ndarray, shape (2,)
        Input features
    y : float
```

```
        True label (0 or 1)
    w1 : ndarray, shape (2, 2)
        Hidden layer weights
    b1 : ndarray, shape (2,)
        Hidden layer biases
    w : ndarray, shape (2,)
        Output layer weights
    b : float
        Output layer bias

    Returns:
    --------
    tuple : (dw1, db1, dw, db)
        dw1 : ndarray, shape (2, 2)
            Gradient w.r.t. hidden layer weights
        db1 : ndarray, shape (2,)
            Gradient w.r.t. hidden layer biases
        dw : ndarray, shape (2,)
            Gradient w.r.t. output layer weights
        db : float
            Gradient w.r.t. output layer bias
    """
    # Forward pass (saving intermediate values)
    # Layer 1: Input -> Hidden
    a1 = np.dot(w1, x) + b1           # Pre-activation (linear)
    h1 = relu(a1)                      # Post-activation (ReLU)

    # Layer 2: Hidden -> Output
    z = np.dot(w, h1) + b              # Pre-activation (linear)
    y_hat = sigmoid(z)                 # Post-activation (Sigmoid)

    # Backward pass (computing gradients)
    # Output layer gradients
    dL_dz = y_hat - y                  # Gradient at output

    db = dL_dz                         # Gradient w.r.t. output bias
    dw = dL_dz * h1                    # Gradient w.r.t. output weights

    # Hidden layer gradients
    dL_dh1 = dL_dz * w                 # Backprop to hidden activations

    # Gradient through ReLU
    dL_da1 = dL_dh1 * relu_derivative(a1)  # Element-wise
   multiplication

    # First layer parameter gradients
    db1 = dL_da1                       # Gradient w.r.t. hidden bias
    dw1 = np.outer(dL_da1, x)          # Gradient w.r.t. hidden weights

    return dw1, db1, dw, db
```

Listing 7: Gradient Computation for Neural Network

```
def train_nn(x, y, learning_rate=0.01, num_iterations=1000):
    """
    Train the neural network using gradient descent.

    Parameters:
    -----------
```

```
    x : ndarray, shape (N, 2)
        Training features
    y : ndarray, shape (N,)
        Training labels
    learning_rate : float
        Step size for gradient descent
    num_iterations : int
        Number of training epochs

    Returns:
    --------
    tuple : (w1, b1, w, b, loss_history)
        Trained parameters and loss history
    """
    # Initialize parameters
    np.random.seed(42)
    w1 = np.random.randn(2, 2) * 0.01
    b1 = np.zeros(2)
    w = np.random.randn(2) * 0.01
    b = 0.0

    loss_history = []

    # Training loop
    for iteration in range(num_iterations):
        total_loss = 0

        # Iterate over all training examples
        for i in range(len(x)):
            x_i = x[i]
            y_i = y[i]

            # Forward pass
            a1 = np.dot(w1, x_i) + b1
            h1 = relu(a1)
            z = np.dot(w, h1) + b
            y_hat = sigmoid(z)

            # Compute loss for this example
            epsilon = 1e-15
            y_hat_clipped = np.clip(y_hat, epsilon, 1 - epsilon)
            loss_i = -(y_i * np.log(y_hat_clipped) +
                       (1 - y_i) * np.log(1 - y_hat_clipped))
            total_loss += loss_i

            # Backward pass (compute gradients)
            dw1, db1, dw, db = gradients(x_i, y_i, w1, b1, w, b)

            # Update parameters (gradient descent)
            w1 = w1 - learning_rate * dw1
            b1 = b1 - learning_rate * db1
            w = w - learning_rate * dw
            b = b - learning_rate * db

        # Average loss over all examples
        avg_loss = total_loss / len(x)
        loss_history.append(avg_loss)

```

```
        # Print progress
        if iteration % 100 == 0:
            print(f"Iteration {iteration}: Loss = {avg_loss:.4f}")

    return w1, b1, w, b, loss_history
```

Listing 8: Complete Training Function

6 Summary of Key Formulas

Backpropagation Formulas Summary

Forward Pass:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{61}$$

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{a}^{(1)}) \tag{62}$$

$$z = \mathbf{w}^T\mathbf{h}^{(1)} + b \tag{63}$$

$$\hat{y} = \sigma(z) \tag{64}$$

Backward Pass (Gradients):

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y \tag{65}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \hat{y} - y \tag{66}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = (\hat{y} - y) \cdot \mathbf{h}^{(1)} \tag{67}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} = (\hat{y} - y) \cdot \mathbf{w} \tag{68}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot g'(\mathbf{a}^{(1)}) \tag{69}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \tag{70}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}}\mathbf{x}^T \tag{71}$$

Parameter Updates:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta \mathcal{L} \tag{72}$$

7 Practice Exercises

7.1 Exercise 1: Verify Gradient Computation

Given:

- $\mathbf{x} = [1, 2]^T$
- $y = 1$

- $\mathbf{W}^{(1)} = \begin{bmatrix} 0.5 & 0.3 \\ 0.2 & 0.4 \end{bmatrix}$
- $\mathbf{b}^{(1)} = [0.1, 0.2]^T$
- $\mathbf{w} = [0.6, 0.7]^T$
- $b = 0.3$

Compute all gradients manually and verify with code.

7.2 Exercise 2: Learning Rate Sensitivity

Train the network with different learning rates: $\alpha \in \{0.001, 0.01, 0.1, 1.0\}$. Plot the loss curves and discuss which works best.

7.3 Exercise 3: Network Extension

Extend the network to have 3 hidden neurons instead of 2. Derive the new gradient formulas and implement them.

8 Conclusion

You've now learned:

1. The mathematical foundations of logistic regression
2. Why we need gradient descent (transcendental equations)
3. How to extend logistic regression to neural networks
4. The complete backpropagation algorithm with detailed derivations
5. How to implement everything from scratch in Python

Key Takeaways:

- Backpropagation is just the chain rule applied systematically
- We compute gradients layer-by-layer from output to input
- Understanding dimensions is crucial for correct implementation
- The gradient tells us the direction; learning rate controls step size

A Appendix A: Matrix Calculus Review

A.1 Useful Identities

For $f(\mathbf{w}) = \mathbf{A}\mathbf{w}$ where $\mathbf{A}$ is constant:

$$\frac{\partial f}{\partial \mathbf{w}} = \mathbf{A}^T \tag{73}$$

For $f(\mathbf{w}) = \mathbf{w}^T\mathbf{A}\mathbf{w}$:

$$\frac{\partial f}{\partial \mathbf{w}} = (\mathbf{A} + \mathbf{A}^T)\mathbf{w} \tag{74}$$

If $\mathbf{A}$ is symmetric: $\frac{\partial f}{\partial \mathbf{w}} = 2\mathbf{A}\mathbf{w}$

A.2 Chain Rule in Multiple Dimensions

For $f = g(h(\mathbf{x}))$:

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{x}} \tag{75}$$

B Appendix B: Common Activation Functions

Function	**Formula**	**Derivative**
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	$\sigma(z)(1-\sigma(z))$
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$1 - \tanh^2(z)$
ReLU	$\text{ReLU}(z) = \max(0, z)$	$\begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
Leaky ReLU	$\max(0.01z, z)$	$\begin{cases} 1 & z > 0 \\ 0.01 & z \leq 0 \end{cases}$

Table 1: Common activation functions and their derivatives

C Appendix C: Further Reading

- *Deep Learning* by Goodfellow, Bengio, and Courville
- *Neural Networks and Deep Learning* by Michael Nielsen (online)
- *Pattern Recognition and Machine Learning* by Christopher Bishop
- CS231n: Convolutional Neural Networks for Visual Recognition (Stanford)