

Neural Network Mathematics: A Complete Walkthrough for Beginners

From Logistic Regression to Backpropagation

December 12, 2025

Contents

1	Introduction and Setup	2
1.1	Required Libraries	2
2	Logistic Regression Foundation	2
2.1	The Binary Cross-Entropy Loss Function	2
2.2	Deriving the Gradient	2
2.2.1	Step 1: Derivative of Sigmoid Function	3
2.2.2	Step 2: Derivative of Loss with Respect to Predictions	3
2.2.3	Step 3: Complete Gradient	4
2.3	Implementation	4
2.4	Why Can't We Solve This Directly?	5
3	Gradient Descent: The Solution	5
3.1	The Algorithm	5
3.2	Understanding the Learning Rate α	6
3.3	Implementation	6
4	From Logistic Regression to Neural Networks	8
4.1	Network Architecture	8
4.2	Forward Propagation	8
4.2.1	Layer 1: Input to Hidden	8
4.2.2	Layer 2: Hidden to Output	9
4.3	Implementation	9
5	Backpropagation: Computing Gradients	10
5.1	Overview	10
5.2	Step 1: Gradient at Output Layer	11
5.3	Step 2: Gradient at Hidden Layer Activations	12
5.4	Step 3: Gradient Through ReLU Activation	12
5.5	Step 4: Gradient at First Layer Bias	13
5.6	Step 5: Gradient at First Layer Weights	14
5.7	Complete Backpropagation Implementation	14
6	Summary of Key Formulas	17

7	Practice Exercises	17
7.1	Exercise 1: Verify Gradient Computation	17
7.2	Exercise 2: Learning Rate Sensitivity	18
7.3	Exercise 3: Network Extension	18
8	Conclusion	18
A	Appendix A: Matrix Calculus Review	19
A.1	Useful Identities	19
A.2	Chain Rule in Multiple Dimensions	19
B	Appendix B: Common Activation Functions	19
C	Appendix C: Further Reading	19

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

```
1 import pandas as pd
2 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

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

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:

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

2.2.1 Step 1: Derivative of Sigmoid Function

The sigmoid function is:

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

Deriving its derivative:

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

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

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

$$= \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} \quad (7)$$

$$= \frac{1}{1 + e^{-z}} \cdot \frac{1 + e^{-z} - 1}{1 + e^{-z}} \quad (8)$$

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

Important Result

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

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:

$$\frac{\partial L_i}{\partial z_i} = \frac{\partial L_i}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \quad (11)$$

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

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

$$= -y_i + y_i\hat{y}_i + \hat{y}_i - y_i\hat{y}_i \quad (14)$$

$$= \hat{y}_i - y_i \quad (15)$$

Gradient with Respect to Predictions

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

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:

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

The gradient becomes:

Final Gradient Formula

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

where:

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

2.3 Implementation

```
1 def sigmoid(z):
2     """
3     Compute sigmoid function: sigma(z) = 1 / (1 + exp(-z))
4
5     Parameters:
6     -----
7     z : array-like
8         Input values
9
10    Returns:
11    -----
12    array-like : Sigmoid of input values
13    """
14    return 1 / (1 + np.exp(-z))
15
16 def loss(y, y_hat):
17     """
18     Binary cross-entropy loss function
19
20     Parameters:
21     -----
22     y : array-like, shape (N,)
23         True labels (0 or 1)
24     y_hat : array-like, shape (N,)
25         Predicted probabilities
26
27     Returns:
28     -----
29     float : Average loss across all examples
```

```

30 """
31 N = len(y)
32 # Add small epsilon to avoid log(0)
33 epsilon = 1e-15
34 y_hat_clipped = np.clip(y_hat, epsilon, 1 - epsilon)
35
36 # Compute cross-entropy
37 loss_value = -np.mean(
38     y * np.log(y_hat_clipped) +
39     (1 - y) * np.log(1 - y_hat_clipped)
40 )
41 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:

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

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

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

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

```
1 def gradient_loss(A, y, y_hat):
2     """
3     Compute gradients of loss w.r.t. weights (w, b) with respect to
4     the design matrix A and predictions.
5
6     For logistic regression where:
7         dw = (1/N) * A^T * (y_hat - y) # Weights gradient
8         db = (1/N) * sum(y_hat - y)    # Bias gradient
9
10    Parameters:
11    -----
12    A : ndarray, shape (N, d+1)
13        Design matrix where:
14        - N = number of training examples (rows)
15        - d = number of features
16        - First column is all ones (for bias)
17
18    y : ndarray, shape (N,)
19        True labels (0 or 1)
20
21    y_hat : ndarray, shape (N,)
22        Predicted probabilities
23
24    Returns:
25    -----
26    tuple : (dw, db)
27        dw : ndarray, shape (d,)
28            Gradient w.r.t. feature weights
29        db : float
30            Gradient w.r.t. bias
31    """
32    N = A.shape[0]
33
34    # Compute residuals (errors)
```

```

35     residuals = y_hat - y
36
37     # Gradient for weights (excluding bias column)
38     # A[:, 1:] excludes the first column of ones
39     dw = (1/N) * np.dot(A[:, 1:].T, residuals)
40
41     # Gradient for bias (first column is all ones)
42     db = (1/N) * np.sum(residuals)
43
44     return dw, db

```

Listing 3: Gradient Computation for Logistic Regression

```

1 def train_model(A, y, learning_rate=0.01, num_iterations=1000):
2     """
3     Train logistic regression using gradient descent.
4
5     Parameters:
6     -----
7     A : ndarray, shape (N, d+1)
8         Design matrix with bias column
9     y : ndarray, shape (N,)
10        True labels
11     learning_rate : float
12        Step size alpha for gradient descent
13     num_iterations : int
14        Number of training iterations
15
16     Returns:
17     -----
18     tuple : (w, b, loss_history)
19         w : ndarray
20             Trained weights
21         b : float
22             Trained bias
23         loss_history : list
24             Loss value at each iteration
25     """
26     # Initialize parameters randomly
27     np.random.seed(42)
28     N, d_plus_1 = A.shape
29     d = d_plus_1 - 1 # Number of features (excluding bias)
30
31     w = np.random.randn(d) * 0.01 # Small random weights
32     b = 0.0
33
34     loss_history = []
35
36     # Training loop
37     for iteration in range(num_iterations):
38         # Forward propagation
39         z = np.dot(A[:, 1:], w) + b # Linear combination
40         y_hat = sigmoid(z) # Predictions
41
42         # Compute loss
43         current_loss = loss(y, y_hat)
44         loss_history.append(current_loss)
45
46         # Backward propagation (compute gradients)

```



```

47     dw, db = gradient_loss(A, y, y_hat)
48
49     # Update parameters (gradient descent step)
50     w = w - learning_rate * dw
51     b = b - learning_rate * db
52
53     # Print progress every 100 iterations
54     if iteration % 100 == 0:
55         print(f"Iteration {iteration}: Loss = {current_loss:.4f}")
56
57     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:

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

where:

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

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

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (24)$$

Activation Function (ReLU):

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

4.2.2 Layer 2: Hidden to Output

Output Layer Computation

Linear Transformation:

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

where:

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

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

Sigmoid Prediction:

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

Total Parameters

Our network has:

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

4.3 Implementation

```
1 def relu(z):
2     """
3     ReLU activation function: max(0, z)
4
5     Parameters:
6     -----
7     z : array-like
8         Input values
9
10    Returns:
11    -----
12    array-like : ReLU(z) = max(0, z)
13    """
14    return np.maximum(0, z)
15
16 def relu_derivative(z):
17     """
18     Derivative of ReLU:
19         dReLU/dz = 1 if z > 0
20                   = 0 if z <= 0
21
22     Parameters:
23     -----
24     z : array-like
```

```

25         Input values
26
27     Returns:
28     -----
29     array-like : Gradient of ReLU
30     """
31     return (z > 0).astype(float)

```

Listing 5: ReLU Activation and Its Derivative

```

1 def model(x, w1, b1, w, b):
2     """
3     Forward pass through a 2-layer neural network.
4
5     Architecture:
6         Input (2) -> Hidden (2, ReLU) -> Output (1, Sigmoid)
7
8     Parameters:
9     -----
10    x : ndarray, shape (2,)
11        Input features
12    w1 : ndarray, shape (2, 2)
13        Hidden layer weights
14    b1 : ndarray, shape (2,)
15        Hidden layer biases
16    w : ndarray, shape (2,)
17        Output layer weights
18    b : float
19        Output layer bias
20
21    Returns:
22    -----
23    float : Predicted probability y_hat
24    """
25    # Hidden layer
26    a1 = np.dot(w1, x) + b1           # Linear transformation
27    h1 = relu(a1)                     # ReLU activation
28
29    # Output layer
30    z = np.dot(w, h1) + b             # Linear transformation
31    y_hat = sigmoid(z)                # Sigmoid activation
32
33    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:

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

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:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y \quad (31)$$

Gradient w.r.t. output bias b :

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial b} \quad (32)$$

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

$$= \hat{y} - y \quad (34)$$

Gradient w.r.t. output weights \mathbf{w} :

Since $z = \mathbf{w}^T \mathbf{h}^{(1)} + b = w_1 h_1^{(1)} + w_2 h_2^{(1)} + b$:

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

Therefore, in vector form:

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

Using chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{w}} \quad (37)$$

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

Boxed Results:

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \hat{y} - y} \quad (39)$$

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

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

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

In vector form:

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

By chain rule:

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

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

Boxed Result:

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

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:

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

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

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

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

In vector form:

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

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:

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

Therefore:

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

This gives us:

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

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:

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

Taking the derivative:

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

Step-by-step for each weight:

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

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

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

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

In matrix form:

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

This can be written as an outer product:

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

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

```

1 def gradients(x, y, w1, b1, w, b):
2     """
3     Compute all gradients for the neural network using backpropagation.
4
5     Network: Input(2) -> Hidden(2, ReLU) -> Output(1, Sigmoid)
6
7     Parameters:
8     -----
9     x : ndarray, shape (2,)
10         Input features
11     y : float

```

```

12     True label (0 or 1)
13     w1 : ndarray, shape (2, 2)
14         Hidden layer weights
15     b1 : ndarray, shape (2,)
16         Hidden layer biases
17     w : ndarray, shape (2,)
18         Output layer weights
19     b : float
20         Output layer bias
21
22     Returns:
23     -----
24     tuple : (dw1, db1, dw, db)
25         dw1 : ndarray, shape (2, 2)
26             Gradient w.r.t. hidden layer weights
27         db1 : ndarray, shape (2,)
28             Gradient w.r.t. hidden layer biases
29         dw : ndarray, shape (2,)
30             Gradient w.r.t. output layer weights
31         db : float
32             Gradient w.r.t. output layer bias
33     """
34     # Forward pass (saving intermediate values)
35     # Layer 1: Input -> Hidden
36     a1 = np.dot(w1, x) + b1          # Pre-activation (linear)
37     h1 = relu(a1)                   # Post-activation (ReLU)
38
39     # Layer 2: Hidden -> Output
40     z = np.dot(w, h1) + b           # Pre-activation (linear)
41     y_hat = sigmoid(z)              # Post-activation (Sigmoid)
42
43     # Backward pass (computing gradients)
44     # Output layer gradients
45     dL_dz = y_hat - y               # Gradient at output
46
47     db = dL_dz                      # Gradient w.r.t. output bias
48     dw = dL_dz * h1                 # Gradient w.r.t. output weights
49
50     # Hidden layer gradients
51     dL_dh1 = dL_dz * w              # Backprop to hidden activations
52
53     # Gradient through ReLU
54     dL_da1 = dL_dh1 * relu_derivative(a1) # Element-wise
multiplication
55
56     # First layer parameter gradients
57     db1 = dL_da1                    # Gradient w.r.t. hidden bias
58     dw1 = np.outer(dL_da1, x)       # Gradient w.r.t. hidden weights
59
60     return dw1, db1, dw, db

```

Listing 7: Gradient Computation for Neural Network

```

1 def train_nn(x, y, learning_rate=0.01, num_iterations=1000):
2     """
3     Train the neural network using gradient descent.
4
5     Parameters:
6     -----

```



```

7  x : ndarray, shape (N, 2)
8      Training features
9  y : ndarray, shape (N,)
10     Training labels
11  learning_rate : float
12     Step size for gradient descent
13  num_iterations : int
14     Number of training epochs
15
16  Returns:
17  -----
18  tuple : (w1, b1, w, b, loss_history)
19     Trained parameters and loss history
20  """
21  # Initialize parameters
22  np.random.seed(42)
23  w1 = np.random.randn(2, 2) * 0.01
24  b1 = np.zeros(2)
25  w = np.random.randn(2) * 0.01
26  b = 0.0
27
28  loss_history = []
29
30  # Training loop
31  for iteration in range(num_iterations):
32      total_loss = 0
33
34      # Iterate over all training examples
35      for i in range(len(x)):
36          x_i = x[i]
37          y_i = y[i]
38
39          # Forward pass
40          a1 = np.dot(w1, x_i) + b1
41          h1 = relu(a1)
42          z = np.dot(w, h1) + b
43          y_hat = sigmoid(z)
44
45          # Compute loss for this example
46          epsilon = 1e-15
47          y_hat_clipped = np.clip(y_hat, epsilon, 1 - epsilon)
48          loss_i = -(y_i * np.log(y_hat_clipped) +
49                     (1 - y_i) * np.log(1 - y_hat_clipped))
50          total_loss += loss_i
51
52          # Backward pass (compute gradients)
53          dw1, db1, dw, db = gradients(x_i, y_i, w1, b1, w, b)
54
55          # Update parameters (gradient descent)
56          w1 = w1 - learning_rate * dw1
57          b1 = b1 - learning_rate * db1
58          w = w - learning_rate * dw
59          b = b - learning_rate * db
60
61      # Average loss over all examples
62      avg_loss = total_loss / len(x)
63      loss_history.append(avg_loss)
64

```

```

65     # Print progress
66     if iteration % 100 == 0:
67         print(f"Iteration {iteration}: Loss = {avg_loss:.4f}")
68
69     return w1, b1, w, b, loss_history

```

Listing 8: Complete Training Function

6 Summary of Key Formulas

Backpropagation Formulas Summary

Forward Pass:

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

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

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

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

Backward Pass (Gradients):

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y \quad (65)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \hat{y} - y \quad (66)$$

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

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

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

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

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

Parameter Updates:

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

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:

$$\frac{\partial f}{\partial \mathbf{w}} = \mathbf{A}^T \quad (73)$$

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

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

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

A.2 Chain Rule in Multiple Dimensions

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

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{x}} \quad (75)$$

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)