Deep Learning Assignment-01

MTech (CS), IIIT Bhubaneswar
March - 2025

Student ID: A124002
Student Name:Hitesh Singh Pathania

Solutions of Deep Learning Assignment 01

1. : For a D -dimensional input vector, show that the optimal weights can be represented by the expression: 1

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

What is the possible estimation of $\mathbf{w}$?

Solution:

To derive the optimal weights $\mathbf{w}$ for a linear regression problem, we start with the least squares objective. Given a dataset with N samples, where $\mathbf{X}$ is the $N \times D$ design matrix (each row corresponds to a D-dimensional input vector), $\mathbf{t}$ is the $N \times 1$ target vector, and $\mathbf{w}$ is the $D \times 1$ weight vector, the goal is to minimize the sum of squared errors:

$$E(\mathbf{w}) = \|\mathbf{t} - \mathbf{X}\mathbf{w}\|^2$$

Expanding the squared error term:

$$E(\mathbf{w}) = (\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w})$$

Taking the derivative of $E(\mathbf{w})$ with respect to $\mathbf{w}$ and setting it to zero for minimization:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = -2\mathbf{X}^T(\mathbf{t} - \mathbf{X}\mathbf{w}) = 0$$

Rearranging the equation:

$$\mathbf{X}^T\mathbf{t} - \mathbf{X}^T\mathbf{X}\mathbf{w} = 0$$

Solving for $\mathbf{w}$:

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{t}$$

Assuming $\mathbf{X}^T\mathbf{X}$ is invertible, the optimal weight vector $\mathbf{w}$ is:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

This is the least squares solution for the weight vector $\mathbf{w}$.

Estimation of w

The expression $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$ provides the optimal weights that minimize the sum of squared errors between the predicted values $\mathbf{Xw}$ and the target values $\mathbf{t}$. This is the best linear unbiased estimator (BLUE) under the assumptions of linear regression (e.g., no multicollinearity, homoscedasticity, and normally distributed errors).

Thus, the estimation of $\mathbf{w}$ is given by:

$$\boxed{\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}}$$

2. : OR Gate in single neural network

Solution:

OR Gate Using a Single-Layer Neural Network

Perceptron Model

A perceptron computes a weighted sum of its inputs and applies an activation function:

$$y = f(w_1x_1 + w_2x_2 + b)$$

Where:

- x_1, x_2 are the inputs
- w_1, w_2 are the weights
- b is the bias
- $f(z)$ is the activation function (step function):

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Initial Example: How a Perceptron Works

To understand how a perceptron works, let's consider initial values:

$$w_1 = 0.5, \quad w_2 = 0.5, \quad b = -0.7$$

We will compute the output for each input using these values:

1. For $x_1 = 0$, $x_2 = 0$:
$$z = (0.5)(0) + (0.5)(0) + (-0.7) = -0.7 \Rightarrow y = 0$$
2. For $x_1 = 0$, $x_2 = 1$:
$$z = (0.5)(0) + (0.5)(1) + (-0.7) = -0.2 \Rightarrow y = 0$$
3. For $x_1 = 1$, $x_2 = 0$:
$$z = (0.5)(1) + (0.5)(0) + (-0.7) = -0.2 \Rightarrow y = 0$$
4. For $x_1 = 1$, $x_2 = 1$:
$$z = (0.5)(1) + (0.5)(1) + (-0.7) = 0.3 \Rightarrow y = 1$$

We can see that with the initial weights and bias, the perceptron does not produce the correct output for an OR gate in all cases. Hence, we need to adjust the weights and bias.

Truth Table for OR Gate

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	1

Determining Weights and Bias

We now choose values that correctly implement the OR gate:

$$w_1 = 1, \quad w_2 = 1, \quad b = -0.5$$

Verification

Now, we verify the perceptron's output for each input combination:

1. For $x_1 = 0$, $x_2 = 0$:
$$z = (1)(0) + (1)(0) + (-0.5) = -0.5 \Rightarrow y = 0$$
2. For $x_1 = 0$, $x_2 = 1$:
$$z = (1)(0) + (1)(1) + (-0.5) = 0.5 \Rightarrow y = 1$$
3. For $x_1 = 1$, $x_2 = 0$:
$$z = (1)(1) + (1)(0) + (-0.5) = 0.5 \Rightarrow y = 1$$
4. For $x_1 = 1$, $x_2 = 1$:
$$z = (1)(1) + (1)(1) + (-0.5) = 1.5 \Rightarrow y = 1$$

Conclusion

The final values:

$$\boxed{w_1 = 1, \quad w_2 = 1, \quad b = -0.5}$$

So this correctly implement the OR gate using a perceptron.

3. Design a perceptron algo to classify over the dataset to create a decision boundary

Solution:

Problem Statement

Design a Perceptron algorithm to classify Iris flowers using only **sepal length** (x_1) and **sepal width** (x_2). We will perform binary classification:

Setosa (label 1) vs. Non-Setosa (label 0)

Algorithm Steps

1. **Load and preprocess the data** using sepal length and sepal width.
2. **Initialize** weights w_1, w_2 and bias b to zero.
3. **For each training sample:**
   - Compute:
     $$z = w_1x_1 + w_2x_2 + b$$
   - Apply the step activation function:
     $$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$
   - Update weights and bias:
     $$w_i \leftarrow w_i + \eta(t - y)x_i, \quad b \leftarrow b + \eta(t - y)$$

     where η is the learning rate and t is the true label.

Python Implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
iris = load_iris()
X = iris.data[:, :2]  # Sepal length and width
y = (iris.target == 0).astype(int)  # Setosa vs others

# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perceptron training function
def perceptron_train(X, y, lr=0.1, epochs=1000):
    weights = np.zeros(X.shape[1])
    bias = 0
    for epoch in range(epochs):
        for i in range(len(X)):
            z = np.dot(X[i], weights) + bias
            y_pred = 1 if z > 0 else 0
            error = y[i] - y_pred
            weights += lr * error * X[i]
            bias += lr * error
    return weights, bias

# Train model
weights, bias = perceptron_train(X_scaled, y)
```

Listing 1: Perceptron on Sepal Length and Width

```python
def plot_decision_boundary(X, y, weights, bias):
    x_min, x_max = X[:, 0].min()-1, X[:, 0].max()+1
    y_min, y_max = X[:, 1].min()-1, X[:, 1].max()+1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))
    Z = (weights[0]*xx + weights[1]*yy + bias > 0).astype(int)

    plt.contourf(xx, yy, Z, alpha=0.2, cmap='RdBu')
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='RdBu', edgecolors='k')
    plt.xlabel("Sepal Length (scaled)")
    plt.ylabel("Sepal Width (scaled)")
    plt.title("Perceptron Decision Boundary")
    plt.show()

plot_decision_boundary(X_scaled, y, weights, bias)
```

Listing 2: Plotting Decision Boundary

4. :for given graph give the following solutions
   (a) :Generalized Point of Intersection for Shallow Neural Networks for input space parameterized by spherical coordinates θ and ϕ

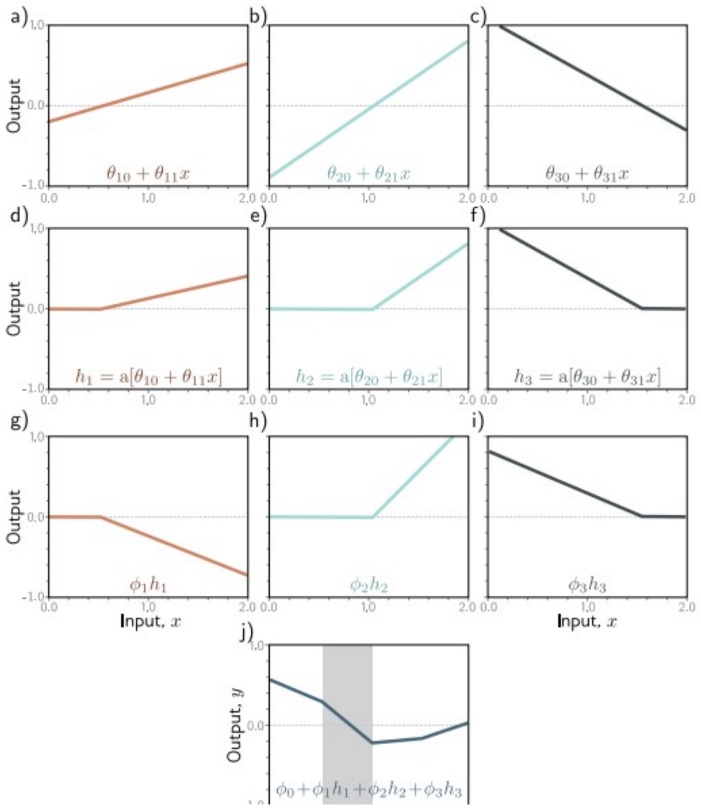

Figure 1: generalization of intersection

Solution:

Generalizing the Point of Intersection in Terms of θ and ϕ for Shallow Neural Networks

Step 1: Structure of a Shallow Neural Network

Consider a shallow neural network with:

- Input dimension: d
- Number of hidden neurons: m
- Activation function: σ
- Weight vectors: $\mathbf{w}_i \in \mathbb{R}^d$
- Bias terms: $b_i \in \mathbb{R}$
- Output weights: $a_i \in \mathbb{R}$

The output of the network is given by:

$$f(\mathbf{x}) = \sum_{i=1}^{m} a_i \, \sigma(\mathbf{w}_i^T \mathbf{x} + b_i)$$

Step 2: Weight Vectors in Angular Coordinates

In spherical coordinates:

$$\mathbf{w} = \|\mathbf{w}\| \begin{bmatrix} \sin(\theta)\cos(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\theta) \end{bmatrix}$$

Step 3: Decision Boundary Condition

For each neuron, the decision boundary satisfies:

$$\mathbf{w}_i^T\mathbf{x} + b_i = 0,$$

which in spherical coordinates becomes:

$$\|\mathbf{w}_i\| \left[x_1 \sin(\theta_i)\cos(\phi_i) + x_2 \sin(\theta_i)\sin(\phi_i) + x_3 \cos(\theta_i)\right] + b_i = 0$$

Step 4: Intersection of Decision Boundaries

If two neurons intersect, we solve the system:

$$\mathbf{w}_i^T\mathbf{x} + b_i = 0, \quad \mathbf{w}_j^T\mathbf{x} + b_j = 0,$$

which translates to:

$$\|\mathbf{w}_i\|\mathbf{x}\cdot\mathbf{v}(\theta_i,\phi_i) + b_i = 0, \quad \|\mathbf{w}_j\|\mathbf{x}\cdot\mathbf{v}(\theta_j,\phi_j) + b_j = 0$$

Step 5: General Solution

The point of intersection $\mathbf{x}$ can be computed by solving the linear system:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b},$$

where $\mathbf{A}$ is the matrix formed by the weight directions in spherical coordinates, and $\mathbf{b}$ is the bias vector.

(b) Give the equation of 4 line segment in graph in θ_1, θ_2, θ_3 etc for figure

Solution:

Consider a shallow neural network with three hidden units and ReLU activations. Let the output y of the network be defined by the following equation:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

where each hidden unit h_i is given by the ReLU activation function:

$$h_i = a(\theta_{i0} + \theta_{i1}x) = \max(0, \theta_{i0} + \theta_{i1}x)$$

The output $y(x)$ is composed of four linear segments, which can be written as:

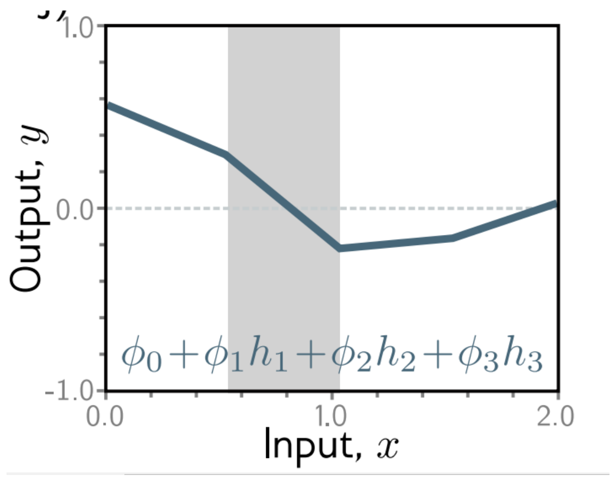

Figure 2: 4 line equations

$$
y(x) = \begin{cases} \phi_0, & x < x_1 \\ \phi_0 + \phi_1(\theta_{10} + \theta_{11}x), & x_1 \le x < x_2 \\ \phi_0 + \phi_1(\theta_{10} + \theta_{11}x) + \phi_2(\theta_{20} + \theta_{21}x), & x_2 \le x < x_3 \\ \phi_0 + \phi_1(\theta_{10} + \theta_{11}x) + \phi_2(\theta_{20} + \theta_{21}x) + \phi_3(\theta_{30} + \theta_{31}x), & x \ge x_3 \end{cases}
$$

Explicitly, the four line segments are:

- First segment: $y = \phi_0$
- Second segment: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11}x)$
- Third segment: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11}x) + \phi_2(\theta_{20} + \theta_{21}x)$
- Fourth segment: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11}x) + \phi_2(\theta_{20} + \theta_{21}x) + \phi_3(\theta_{30} + \theta_{31}x)$

The activation thresholds x_1, x_2, and x_3 where each hidden unit is activated are given by:

$$
x_i = -\frac{\theta_{i0}}{\theta_{i1}}, \quad \text{for each neuron.}
$$

The output function combines the contributions of all active hidden units according to their weights and is expressed in the above piecewise form.

5. How do for the 2D case, what if there were two of the other one look like?

Achieving tips:

$$
h_d = a\left(\frac{P_{th}}{P_{eq}} + \sum_{d=1}^{n} \phi_{jd}^{*} h_d\right)
$$

Solution:

Neural Network Architecture Description

1. Hidden Layer Computation

Each hidden unit h_d takes the input vector and applies a linear transformation followed by a nonlinear activation:

$$h_d = a\left(\theta_{d0} + \sum_{i=1}^{2} \theta_{di} x_i\right) \quad \text{for } d = 1, 2, \ldots, D$$

Where:

h_d: output of the d-th hidden unit θ_{d0}: bias term for hidden unit d θ_{di}: weight from input x_i to hidden unit d $a(\cdot)$: activation function (e.g., sigmoid, tanh, ReLU)

2. Output Layer Computation

Each output neuron performs a linear combination of all hidden unit outputs with a bias term:

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d \quad \text{for } j = 1, 2$$

Where:

y_j: output of the j-th output unit ϕ_{j0}: bias for output unit j ϕ_{jd}: weight from hidden unit d to output unit j

3. General Equation of Both Outputs

If one of the outputs is:

$$y_1 = \phi_{10} + \sum_{d=1}^{D} \phi_{1d} h_d$$

Then the other output is similarly given by:

$$y_2 = \phi_{20} + \sum_{d=1}^{D} \phi_{2d} h_d$$

Explanation:

Both outputs depend on the same hidden layer outputs $h_1, h_2, \ldots, h_D$ The only difference lies in the weights ϕ_{jd} and biases ϕ_{j0} used for each output neuron This allows the network to produce different outputs from the same input vector through different weightings

4. Final Combined Form

$$y_1 = \phi_{10} + \sum_{d=1}^{D} \phi_{1d} \cdot a\left(\theta_{d0} + \sum_{i=1}^{2} \theta_{di} x_i\right)$$
$$y_2 = \phi_{20} + \sum_{d=1}^{D} \phi_{2d} \cdot a\left(\theta_{d0} + \sum_{i=1}^{2} \theta_{di} x_i\right)$$

Thus, both outputs are linear combinations of nonlinearly transformed weighted inputs.

6. :Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be independent and identically distributed (i.i.d.) vectors from a multivariate normal distribution:

$$\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$

where $\boldsymbol{\mu}$ is the unknown mean vector and Σ is the known covariance matrix.

Solution:

Maximum Likelihood Estimate of Unknown Mean Vector

Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be independent and identically distributed (i.i.d.) vectors from a multivariate normal distribution:

$$\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$

where $\boldsymbol{\mu}$ is the unknown mean vector and Σ is the known covariance matrix.

The probability density function (PDF) of $\mathbf{x}_i$ is given by:

$$f(\mathbf{x}_i|\boldsymbol{\mu}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}_i - \boldsymbol{\mu})\right)$$

Likelihood Function

Given the independence of the samples, the likelihood function is the product of the individual densities:

$$L(\boldsymbol{\mu}) = \prod_{i=1}^{n} f(\mathbf{x}_i|\boldsymbol{\mu})$$

Taking the natural logarithm of the likelihood function (log-likelihood):

$$\log L(\boldsymbol{\mu}) = -\frac{np}{2}\log(2\pi) - \frac{n}{2}\log|\Sigma| - \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}_i - \boldsymbol{\mu})$$

Maximizing the Log-Likelihood

To find the MLE of $\boldsymbol{\mu}$, we differentiate $\log L(\boldsymbol{\mu})$ with respect to $\boldsymbol{\mu}$ and set the result to zero:

$$\frac{\partial \log L}{\partial \boldsymbol{\mu}} = \sum_{i=1}^{n} \Sigma^{-1}(\mathbf{x}_i - \boldsymbol{\mu}) = 0$$

Simplifying:

$$\sum_{i=1}^{n}(\mathbf{x}_i - \boldsymbol{\mu}) = 0 \Rightarrow n\hat{\boldsymbol{\mu}} = \sum_{i=1}^{n}\mathbf{x}_i$$

Result: MLE of Mean Vector

$$\hat{\boldsymbol{\mu}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$$

Thus, the maximum likelihood estimate of the unknown mean vector is the sample mean.

7. Backpropagation for Cross-Entropy Loss with Softmax (3 Outputs)

Solution:

We are given a neural network with 3 output logits $f = [f_1, f_2, f_3]$. These logits are passed through a softmax activation to produce probabilities, and the loss is computed using cross-entropy. We aim to compute the derivative of the loss with respect to each logit f_k, i.e., $\frac{\partial L}{\partial f_k}$ for $k = 1, 2, 3$.

Softmax Function

Let the softmax function be defined as:

$$p_i = \frac{e^{f_i}}{\sum_{j=1}^{3} e^{f_j}} \quad \text{for } i = 1, 2, 3$$

where p_i is the predicted probability for class i.

Cross-Entropy Loss

The cross-entropy loss function is defined as:

$$L = -\sum_{i=1}^{3} y_i \log p_i$$

where $y = [y_1, y_2, y_3]$ is a one-hot encoded true label vector.

Step 1: Derivative of Softmax

We first compute the derivative of the softmax function with respect to the logits f_k.

$$\frac{\partial p_i}{\partial f_k} = p_i(\delta_{ik} - p_k)$$

where δ_{ik} is the Kronecker delta:

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

Step 2: Derivative of Loss

Now, compute the derivative of the loss with respect to f_k using the chain rule:

$$\frac{\partial L}{\partial f_k} = -\sum_{i=1}^{3} y_i \cdot \frac{\partial}{\partial f_k} \log p_i = -\sum_{i=1}^{3} y_i \cdot \frac{1}{p_i} \cdot \frac{\partial p_i}{\partial f_k}$$

$$= -\sum_{i=1}^{3} y_i \cdot \frac{1}{p_i} \cdot p_i(\delta_{ik} - p_k) = -\sum_{i=1}^{3} y_i(\delta_{ik} - p_k)$$

$$= -y_k(1 - p_k) - \sum_{i \neq k} y_i(-p_k) = -y_k + y_k p_k + p_k \sum_{i \neq k} y_i$$

Since $\sum_{i=1}^{3} y_i = 1$, we get $\sum_{i \neq k} y_i = 1 - y_k$. Therefore:

$$\frac{\partial L}{\partial f_k} = -y_k + y_k p_k + p_k(1 - y_k) = p_k - y_k$$

Final Result

$$\boxed{\frac{\partial L}{\partial f_k} = p_k - y_k \quad \text{for } k = 1, 2, 3}$$

This is the gradient used in backpropagation when using a softmax output layer and cross-entropy loss function.

8. Backpropagation for 3-class classification (2 inputs, 2 hidden units, sigmoid activation)

Solution:

Problem Setup (Black):
We are given a shallow neural network with the following architecture:

- 2 input neurons: x_1, x_2
- 2 hidden neurons with sigmoid activation
- 3 output neurons with softmax activation (for 3-class classification)

The network is trained using the cross-entropy loss function.

Step 1: Forward Pass

Let the input vector be $x \in \mathbb{R}^2$, weights and biases as:

- $W^{(1)} \in \mathbb{R}^{2\times 2}$, $b^{(1)} \in \mathbb{R}^2$: hidden layer weights and biases
- $W^{(2)} \in \mathbb{R}^{3\times 2}$, $b^{(2)} \in \mathbb{R}^3$: output layer weights and biases

$$z^{(1)} = W^{(1)}x + b^{(1)}, \quad h = \sigma(z^{(1)})$$

$$z^{(2)} = W^{(2)}h + b^{(2)}, \quad f = \text{softmax}(z^{(2)})$$

$$L = -\sum_{k=1}^{3} y_k \log(f_k)$$

Step 2: Gradient at Output Layer

$$\frac{\partial L}{\partial z_k^{(2)}} = f_k - y_k$$

$$\boxed{\frac{\partial L}{\partial z^{(2)}} = f - y}$$

Step 3: Gradients for Output Weights and Bias

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \cdot h^T$$

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial z^{(2)}}$$

$$\boxed{\frac{\partial L}{\partial W^{(2)}} = (f-y)h^T, \quad \frac{\partial L}{\partial b^{(2)}} = f - y}$$

Step 4: Backprop to Hidden Layer

$$\frac{\partial L}{\partial h} = (W^{(2)})^T \cdot \frac{\partial L}{\partial z^{(2)}}$$

$$\frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial h} \circ \sigma'(z^{(1)}), \quad \text{where } \sigma'(z) = \sigma(z)(1-\sigma(z))$$

$$\boxed{\frac{\partial L}{\partial z^{(1)}} = \left((W^{(2)})^T(f-y)\right) \circ \sigma(z^{(1)})(1-\sigma(z^{(1)}))}$$

Step 5: Gradients for First Layer Weights and Bias

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial z^{(1)}} \cdot x^T$$

$$\frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial z^{(1)}}$$

$$\boxed{\frac{\partial L}{\partial W^{(1)}} = \left(\frac{\partial L}{\partial z^{(1)}}\right) x^T, \quad \frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial z^{(1)}}}$$

Conclusion

successfully computed the gradients for all parameters in a shallow 2-layer neural network using sigmoid activation and softmax+cross-entropy loss for 3-class classification.