Deep Learning Assignments

MTech (CS), IIIT Bhubaneswar
March - 2025

Student ID: A124013
Student Name: Snehashisa Ratna

Solutions of Deep Learning Assignments

1. **Deriving the Optimal Weights for Linear Regression:**
For a D-dimensional input vector, show that the optimal weights can be represented by the expression:

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

Also, interpret what this expression means in terms of estimating $\mathbf{w}$.

Solution:

Let's consider we are given a dataset with N samples. Let:

- $\mathbf{X}$ be the $N \times D$ matrix of inputs (called the design matrix),
- $\mathbf{t}$ be the $N \times 1$ target vector, and
- $\mathbf{w}$ be the $D \times 1$ vector of weights we want to find.

The objective in linear regression is to minimize the squared error between the predicted outputs $\mathbf{Xw}$ and the actual target values $\mathbf{t}$. So we define the error function:

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

Expanding the norm:

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

Now, to minimize this, we take the derivative with respect to $\mathbf{w}$ and set it to zero:

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

Rearranging terms:

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

Interpretation:

The final formula we derived, $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$, is called the *normal equation*. It gives the optimal linear weights that minimize squared error on the training data.

This is also known as the **Best Linear Unbiased Estimator (BLUE)** — it works well when the typical assumptions of linear regression (like no multicollinearity, constant variance of errors, etc.) are satisfied.

So, this equation provides a neat and powerful way to compute weights directly, without iterative optimization methods.

2. **OR Gate Implementation Using a Single-Layer Neural Network**

Solution:

To implement the OR gate, I used a single-layer perceptron model with two inputs. The perceptron calculates output based on the following equation:

$$y = f(w_1 x_1 + w_2 x_2 + b)$$

Where:

- Initial weights: $w_1 = 1.5,\ w_2 = 2$
- Bias: $b = -2$
- Learning rate: $\eta = 0.5$
- Activation function:

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

OR Gate Truth Table:

x_1	x_2	t (Target)
0	0	0
0	1	1
1	0	1
1	1	1

Epoch 1 – Initial Check:

- Input (0,0):

$$z = 1.5 \cdot 0 + 2 \cdot 0 + (-2) = -2 \Rightarrow y = 0 ✓$$

- Input (0,1):

$$z = 1.5 \cdot 0 + 2 \cdot 1 + (-2) = 0 \Rightarrow y = 1 ✓$$

- Input (1,0):

$$z = 1.5 \cdot 1 + 2 \cdot 0 + (-2) = -0.5 \Rightarrow y = 0 \times \quad \text{(Expected 1)}$$

Since this was incorrect, I updated the weights and bias:

$$\Delta w_1 = \eta(t - y)x_1 = 0.5(1 - 0)(1) = 0.5$$
$$\Delta w_2 = \eta(t - y)x_2 = 0.5(1 - 0)(0) = 0$$
$$\Delta b = \eta(t - y) = 0.5(1 - 0) = 0.5$$

Updated parameters:

$$w_1 = 1.5 + 0.5 = 2.0$$
$$w_2 = 2.0 + 0 = 2.0$$
$$b = -2 + 0.5 = -1.5$$

- Input (1,1):

$$z = 2 \cdot 1 + 2 \cdot 1 + (-1.5) = 2.5 \Rightarrow y = 1 \checkmark$$

Epoch 2 – Rechecking After Update:

Now I tested again using the updated values:

New values: $w_1 = 2.0$, $w_2 = 2.0$, $b = -1.5$

- Input (0,0):

$$z = 2 \cdot 0 + 2 \cdot 0 + (-1.5) = -1.5 \Rightarrow y = 0 \checkmark$$

- Input (0,1):

$$z = 2 \cdot 0 + 2 \cdot 1 + (-1.5) = 0.5 \Rightarrow y = 1 \checkmark$$

- Input (1,0):

$$z = 2 \cdot 1 + 2 \cdot 0 + (-1.5) = 0.5 \Rightarrow y = 1 \checkmark$$

- Input (1,1):

$$z = 2 \cdot 1 + 2 \cdot 1 + (-1.5) = 2.5 \Rightarrow y = 1 \checkmark$$

Final Outcome:

After just one update, the perceptron correctly predicts all OR gate outputs.

Final Parameters:

$$w_1 = 2.0, \quad w_2 = 2.0, \quad b = -1.5$$

Decision Boundary:

$$2x_1 + 2x_2 + (-1.5) = 0$$

3. **Design a Perceptron algorithm to classify Iris flowers using either sepal or petal features and create a decision boundary.**

Solution:

We design a simple perceptron algorithm to classify two classes of Iris flowers using either their sepal or petal measurements. The goal is to train a model that learns to distinguish between species using a linear decision boundary.

1. Data Preparation

- **Feature Selection:**

$$\text{Features} = \begin{cases} (\text{sepal length}, \text{sepal width}) & \text{if using sepal features} \\ (\text{petal length}, \text{petal width}) & \text{if using petal features} \end{cases}$$

- **Binary Encoding (e.g., Setosa vs Virginica):**

$$y = \begin{cases} 1 & \text{Virginica} \\ 0 & \text{Setosa} \end{cases}$$

- **Min-Max Normalization:**

$$x_{\text{norm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

2. Model Initialization

- $w_1, w_2 \sim \mathcal{U}(-0.01, 0.01)$ (small random weights)
- $b = 0$ (initial bias)
- $\eta = 0.1$ (learning rate)

3. Training Phase (repeat until convergence)

1. **Compute Activation:**

$$z = w_1 x_1 + w_2 x_2 + b$$

2. **Apply Step Function:**

$$\hat{y} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

3. **Update Parameters (if misclassified):**

$$\Delta w_i = \eta(y - \hat{y})x_i \quad \text{for } i = 1, 2$$

$$\Delta b = \eta(y - \hat{y})$$

4. Decision Boundary

Once the perceptron has converged, the decision boundary separating the two classes is described by:

$$w_1 x_1 + w_2 x_2 + b = 0$$

We can rearrange this into slope-intercept form to visualize it:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

This line divides the input space into two halves, with each side corresponding to one of the flower classes.

Python Code with Comments:

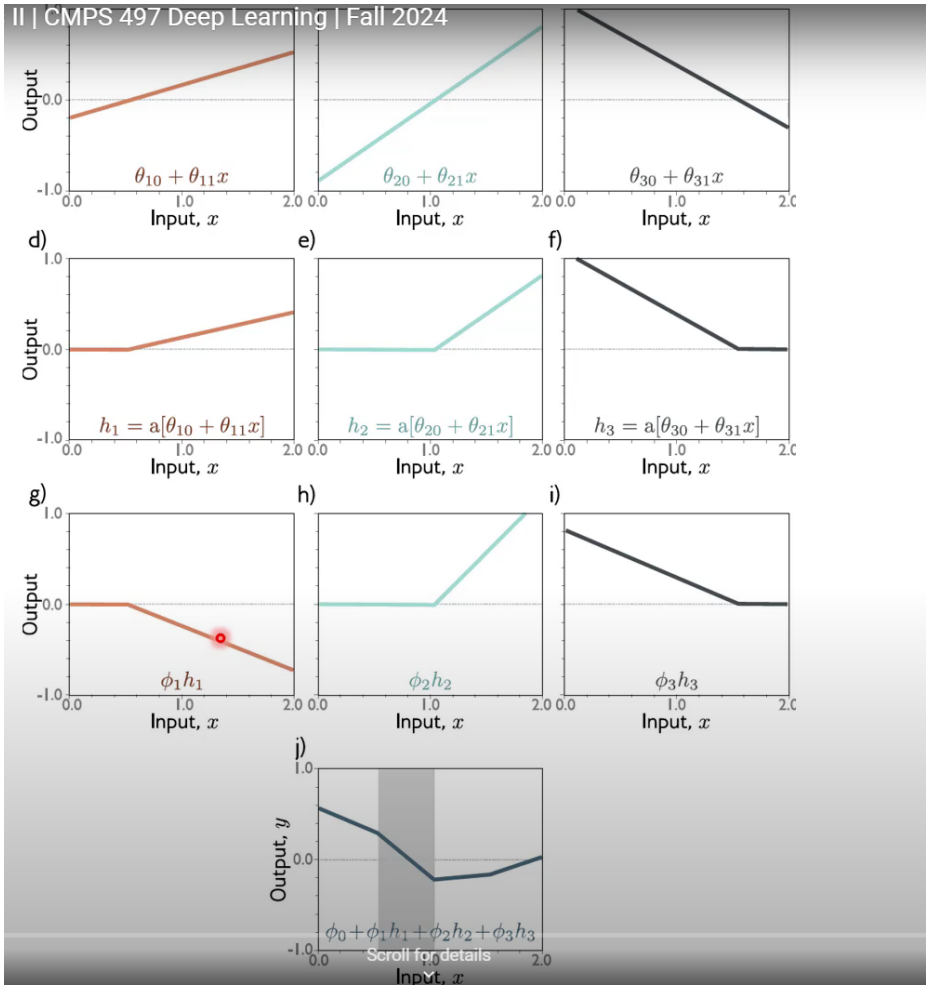
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Ask user to choose features and class pair
def get_user_config():
    print("Choose input feature type:")
```

```
    print("1. Sepal (Length & Width)")
    print("2. Petal (Length & Width)")
    feature_mode = int(input("Enter choice (1 or 2): "))

    print("\nSelect class pair:")
    print("1. Setosa (0) vs Versicolor (1)")
    print("2. Setosa (0) vs Virginica (2)")
    print("3. Versicolor (1) vs Virginica (2)")
    class_option = int(input("Enter class pair (1-3): "))
    return feature_mode, class_option

# Load and preprocess the dataset based on chosen options
def load_and_prepare_data(feature_mode, class_option):
    iris = load_iris()
    X = iris.data[:, :2] if feature_mode == 1 else iris.data[:,
        2:]
    feature_labels = iris.feature_names[:2] if feature_mode == 1
        else iris.feature_names[2:]

    class_map = {1: (0, 1), 2: (0, 2), 3: (1, 2)}
    cls_a, cls_b = class_map[class_option]

    selected = np.isin(iris.target, [cls_a, cls_b])
    X = X[selected]
    y = np.where(iris.target[selected] == cls_b, 1, 0)
    return X, y, feature_labels, iris.target_names[cls_a], iris.
        target_names[cls_b]

# Train a simple perceptron using the rule: update weights if
    prediction is wrong
def train_perceptron(X, y, lr=0.1, epochs=500):
    weights = np.zeros(X.shape[1])
    bias = 0
    for _ in range(epochs):
        for xi, target in zip(X, y):
            z = np.dot(xi, weights) + bias
            pred = int(z > 0)
            update = lr * (target - pred)
            weights += update * xi
            bias += update
    return weights, bias

# Evaluate model on test data
def evaluate(X, y, weights, bias):
    preds = (np.dot(X, weights) + bias > 0).astype(int)
    return np.mean(preds == y)

# Plot the decision boundary learned by the perceptron
def plot_decision_boundary(X_train, X_test, y_train, y_test,
    weights, bias, features, classes):
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() +
        1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() +
        1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
```

```
                            np.linspace(y_min, y_max, 200))
    Z = (weights[0]*xx + weights[1]*yy + bias > 0).astype(int)

    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='
        coolwarm', edgecolors='k', label="Train")
    plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='
        coolwarm', marker='x', label="Test")
    plt.xlabel(f"{features[0]} (standardized)")
    plt.ylabel(f"{features[1]} (standardized)")
    plt.title(f"Perceptron Classifier: {classes[0]} vs {classes
        [1]}")
    plt.legend()
    plt.grid(True)
    plt.show()

# Main function to run the whole process
def main():
    feature_mode, class_option = get_user_config()
    X, y, feature_names, cls_a, cls_b = load_and_prepare_data(
        feature_mode, class_option)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
        test_size=0.2, random_state=1)
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(XTest)
    weights, bias = train_perceptron(X_train, y_train)
    acc = evaluate(X_test, y_test, weights, bias)
    print(f"\nTest Accuracy: {acc:.2%}")
    plot_decision_boundary(X_train, X_test, y_train, y_test,
        weights, bias, feature_names, (cls_a, cls_b))

if __name__ == "__main__":
    main()
```

Listing 1: Perceptron Classifier for Iris Dataset

4. **Figure 1: Generalization of Intersection in Shallow Neural Networks**



(a) **Generalized Point of Intersection for Shallow Neural Networks for input space parameterized by spherical coordinates θ and ϕ**

Solution:

Step 1: Structure of a Shallow Neural Network

We consider a shallow neural network with the following elements:

- Input dimension: d
- 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 network output is:

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

Step 2: Weight Vectors in Spherical Coordinates

The weight vector in spherical form is:

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

Step 3: Decision Boundary Condition
For each neuron, the decision boundary is where:

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

In spherical coordinates:

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

Step 4: Intersection of Two Neuron Hyperplanes
To find an intersection point of two neurons:

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

This becomes:

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

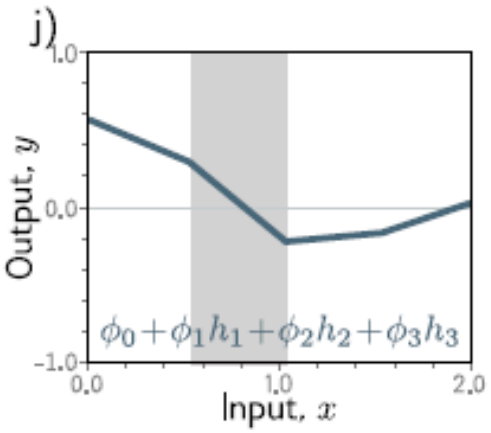
Step 5: Solving for the Intersection Point
We form a system:

$$x = A^{-1} b$$

Where:

- A is formed by direction vectors $v(\theta_i, \phi_i)$
- b is formed by the bias terms $-b_i / \|\mathbf{w}_i\|$

(b) **Equation of Line Segments in Output Graph**



Solution:

We consider a shallow network with 3 hidden ReLU units. The output is given by:

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

Where:

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

The output $y(x)$ has four piecewise-linear segments:

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

Where:

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

Line Segment Equations:

- Segment 1: $y = \phi_0$
- Segment 2: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11} x)$
- Segment 3: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11} x) + \phi_2(\theta_{20} + \theta_{21} x)$
- Segment 4: $y = \phi_0 + \phi_1(\theta_{10} + \theta_{11} x) + \phi_2(\theta_{20} + \theta_{21} x) + \phi_3(\theta_{30} + \theta_{31} x)$

This piecewise form illustrates how shallow neural networks construct nonlinear outputs using linear regions stitched at the activation thresholds.

5. **What is the general form of the second output in a Two-Output Feedforward Neural Network (2D case), if the first output is already given?**

Solution:

We analyze a feedforward neural network architecture that has:

- Two input features: x_1 and x_2
- A hidden layer with D neurons
- Two output units: y_1 and y_2
- Nonlinear activation function $a(\cdot)$ used in the hidden layer

1. Hidden Layer Computation:

Each hidden neuron processes the input vector (x_1, x_2) by applying a weighted sum followed by a non-linear activation. For the d^{th} hidden neuron:

$$h_d = a\left(\theta_{d0} + \theta_{d1} x_1 + \theta_{d2} x_2\right) \quad \text{for } d = 1, 2, \ldots, D$$

Where:

- θ_{d0} is the bias term for the d^{th} hidden neuron
- θ_{d1}, θ_{d2} are weights connecting inputs to neuron d
- $a(\cdot)$ is typically ReLU, sigmoid, or tanh

2. Output Layer Computation:

Each output neuron combines the hidden activations linearly, with its own set of weights and a bias:

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

So if we already have:

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

Then the second output is given by:

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

Observation:

- Both y_1 and y_2 use the same set of hidden layer outputs $h_1, h_2, \ldots, h_D$
- The distinction lies in how each output neuron weighs these hidden activations
- This architecture allows multiple outputs to react differently to the same input

3. Combined General Form:

By substituting the hidden layer equations into the output equations, we get a complete form of both outputs in terms of the inputs:

$$y_1 = \phi_{10} + \sum_{d=1}^{D} \phi_{1d} \cdot a\left(\theta_{d0} + \theta_{d1} x_1 + \theta_{d2} x_2\right)$$

$$y_2 = \phi_{20} + \sum_{d=1}^{D} \phi_{2d} \cdot a\left(\theta_{d0} + \theta_{d1} x_1 + \theta_{d2} x_2\right)$$

4. Intuition:

This shows that both y_1 and y_2 are just different weighted views of the same non-linear transformations of the input vector. One can think of the hidden layer as extracting features, while the output layer interprets those features differently for different tasks or labels.

Optional: A visual representation of the architecture would involve:

- 2 input nodes
- D hidden neurons with activation $a(\cdot)$
- 2 output nodes with independent weights from the hidden layer

6. **Let $x_1, x_2, \ldots, x_n$ be independent and identically distributed (i.i.d.) vectors from a multivariate normal distribution:**

$$x_i \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

where $\boldsymbol{\mu}$ is the unknown mean vector and $\boldsymbol{\Sigma}$ is a known covariance matrix. Derive the Maximum Likelihood Estimate (MLE) of $\boldsymbol{\mu}$.

Solution:

We are given that $x_1, x_2, \ldots, x_n$ are i.i.d. samples from a p-dimensional multivariate normal distribution:

$$x_i \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

Here, $\boldsymbol{\mu}$ is the unknown parameter to be estimated, while $\boldsymbol{\Sigma}$ is known.

Step 1: Probability Density Function (PDF)

The PDF of a single sample x_i under the multivariate normal distribution is:

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

Step 2: Likelihood Function

Assuming independence of the n observations, the joint likelihood function is the product of the individual PDFs:

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

Taking the natural logarithm gives the log-likelihood:

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

Step 3: Maximizing the Log-Likelihood

We now differentiate the log-likelihood with respect to $\boldsymbol{\mu}$ and set the derivative to zero:

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

Multiplying through by $\boldsymbol{\Sigma}$:

$$\sum_{i=1}^{n}(x_i - \boldsymbol{\mu}) = 0 \quad \Rightarrow \quad \sum_{i=1}^{n} x_i = n\boldsymbol{\mu}$$

Solving for $\boldsymbol{\mu}$:

$$\hat{\boldsymbol{\mu}} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Conclusion:

The maximum likelihood estimate (MLE) of the mean vector $\boldsymbol{\mu}$ is simply the **sample mean**:

$$\boxed{\hat{\boldsymbol{\mu}} = \frac{1}{n}\sum_{i=1}^{n} x_i}$$

This result makes intuitive sense: the best estimate for the average of a Gaussian distribution, when covariance is known, is just the average of the observed data.

7. **Backpropagation for the cross-entropy loss function of a network with 3 outputs (f_1, f_2, f_3). Assume the outputs are the only parameters of the loss.**

Solution:

We consider a neural network with 3 output logits: f_1, f_2, f_3. These are passed through the softmax function to produce class probabilities:

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

Assume the ground truth label is one-hot encoded:

$$\mathbf{y} = (y_1, y_2, y_3) \quad \text{where } y_k = 1 \text{ for the correct class and } y_i = 0 \text{ for } i \neq k$$

Cross-Entropy Loss Function:

The loss is computed as:

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

Since only $y_k = 1$, the loss simplifies to:

$$L = -\log(p_k)$$

Goal: Compute the gradient $\frac{\partial L}{\partial f_i}$ for $i = 1, 2, 3$

Step 1: Compute $\frac{\partial L}{\partial p_j}$

$$\frac{\partial L}{\partial p_j} = -\frac{y_j}{p_j}$$

Step 2: Compute $\frac{\partial p_j}{\partial f_i}$ (derivative of softmax):

$$\frac{\partial p_j}{\partial f_i} = p_j(\delta_{ij} - p_i)$$

Where δ_{ij} is the Kronecker delta, equal to 1 if $i = j$, and 0 otherwise.

Step 3: Apply the Chain Rule:

$$\frac{\partial L}{\partial f_i} = \sum_{j=1}^{3} \frac{\partial L}{\partial p_j} \cdot \frac{\partial p_j}{\partial f_i} = \sum_{j=1}^{3} \left(-\frac{y_j}{p_j} \cdot p_j(\delta_{ij} - p_i) \right) = -\sum_{j=1}^{3} y_j(\delta_{ij} - p_i)$$

Since $y_j = 1$ only for $j = k$, the summation simplifies:

$$\frac{\partial L}{\partial f_i} = -(\delta_{ik} - p_i) = p_i - y_i$$

Final Result:

$$\frac{\partial L}{\partial f_i} = p_i - y_i \quad \text{for } i = 1, 2, 3$$

Conclusion:

This elegant result shows that the gradient of the cross-entropy loss with softmax is simply the difference between the predicted probability and the true label. It forms the backbone of efficient training in multi-class classification networks using gradient descent.

8. **Backpropagation for 3-class classification using a neural network with 2 inputs, 2 hidden sigmoid units, and 3 softmax output neurons. Derive the forward and backward pass expressions assuming cross-entropy loss.**

Solution:

We consider a neural network with the following structure:

- **Input layer:** 2 features (x_1, x_2)
- **Hidden layer:** 2 neurons with sigmoid activation
- **Output layer:** 3 neurons with softmax activation
- **Loss function:** Cross-entropy loss

Notation:

- $W^{[1]} \in \mathbb{R}^{2\times 2}$: weights from input to hidden layer
- $b^{[1]} \in \mathbb{R}^2$: biases for hidden layer
- $z^{[1]} = W^{[1]}x + b^{[1]}$: pre-activation for hidden layer
- $a^{[1]} = \sigma(z^{[1]})$: activations from sigmoid
- $W^{[2]} \in \mathbb{R}^{3\times 2}$: weights from hidden to output layer
- $b^{[2]} \in \mathbb{R}^3$: biases for output layer

- $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$: pre-softmax activations
- $\hat{y} = \text{softmax}(z^{[2]})$: predicted probabilities
- $y \in \mathbb{R}^3$: one-hot encoded true label

Forward Pass:

1. **Hidden Layer:**
$$z^{[1]} = W^{[1]}x + b^{[1]}, \quad a^{[1]} = \sigma(z^{[1]})$$
2. **Output Layer:**
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \quad \hat{y} = \text{softmax}(z^{[2]})$$
3. **Loss Function (Cross-Entropy):**
$$L = -\sum_{i=1}^{3} y_i \log(\hat{y}_i)$$

Backward Pass:

1. **Gradient w.r.t. Output Pre-activation ($z^{[2]}$):**
$$\frac{\partial L}{\partial z^{[2]}} = \hat{y} - y$$
2. **Gradients for Output Layer Weights and Biases:**
$$\frac{\partial L}{\partial W^{[2]}} = (\hat{y} - y)(a^{[1]})^T \quad , \quad \frac{\partial L}{\partial b^{[2]}} = \hat{y} - y$$
3. **Backprop to Hidden Layer:**
$$\delta^{[1]} = \left(W^{[2]^T}(\hat{y} - y)\right) \circ \sigma'(z^{[1]})$$
Where:
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$
4. **Gradients for Hidden Layer Parameters:**
$$\frac{\partial L}{\partial W^{[1]}} = \delta^{[1]}x^T \quad , \quad \frac{\partial L}{\partial b^{[1]}} = \delta^{[1]}$$

Conclusion:

These steps detail the full forward and backward passes for a 3-class neural network with one hidden layer of sigmoid units. The gradients derived can be used to update parameters using optimization algorithms like SGD or Adam.

† This document is created using LaTeX typesetting system.