NLP Assignment 2

IMT2022025 Varnit Mittal,
IMT2022075 Aditya Priyadarshi,
IMT2022086 Ananthakrishna K,
IMT2022519 Vedant Mangrulkar

March 17, 2025

Part 1

Introduction

The XOR classification problem is a classical example where a simple perceptron fails due to non-linearity. The goal is to train an MLP to classify data points based on two binary input features (x_1, x_2) and an output label y. This study investigates different neural network architectures, optimization techniques, and preprocessing methods to improve classification accuracy.

Why do we need MLP?

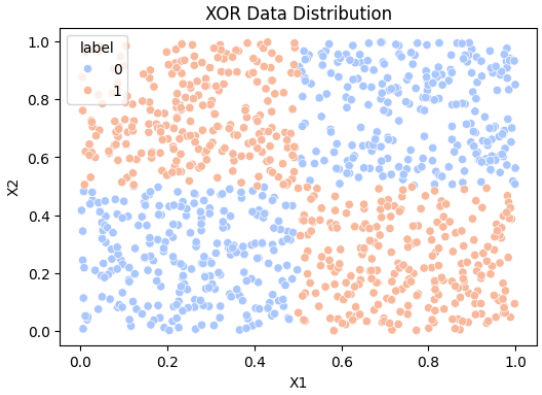

Figure 1: XOR Data Distribution. The points belonging to different classes are not linearly separable.

As observed in Figure 1, the XOR dataset is not linearly separable. A single linear decision boundary cannot correctly classify the points into their respective

classes. We will now prove this formally.

Proof: A perceptron can only solve linearly separable problems, meaning there must exist a weight vector $\mathbf{w} = (w_0, w_1, w_2)$ such that for all inputs (x_1, x_2) in the XOR dataset:

$$y = \text{sign}(w_0 + w_1 x_1 + w_2 x_2)$$

The XOR basically dataset consists of:

$$(0,0) \to 0, \quad (0,1) \to 1, \quad (1,0) \to 1, \quad (1,1) \to 0$$

For these points to be linearly separable, a single linear equation of the form $w_0 + w_1 x_1 + w_2 x_2 = 0$ must exist that correctly classifies all points. However, solving:

$$w_0 + w_1(0) + w_2(0) < 0, \quad w_0 + w_1(0) + w_2(1) > 0$$

$$w_0 + w_1(1) + w_2(0) > 0, \quad w_0 + w_1(1) + w_2(1) < 0$$

leads to a contradiction, proving that no single hyperplane can separate XOR classes. Hence, a simple perceptron fails for the XOR problem.

Methodology

Dataset Description

The dataset consists of three features:

- $x_1, x_2 \in \mathbb{R}, \quad 0 \leq x_1, x_2 \leq 1$
- y: Output label (0 or 1) following the XOR function

Each input follows the logical XOR pattern:

$$y = (x_1 > 0.5) \oplus (x_2 > 0.5)$$

where $\oplus$ denotes the XOR operation.

Two primary architectures were tested with different optimization techniques:

Architecture 1: MLP with Adam Optimization

- **Input Layer**: 2 neurons (x_1, x_2)
- **Hidden Layer**: 2 neurons with **tanh** activation
- **Output Layer**: 1 neuron with **sigmoid** activation
- **Optimizer**: Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
- **Weight Initialization**: He Initialization

- **Regularization**: L2 ($\lambda = 0.001$)
- **Loss Function**: Mean Squared Error (MSE)
- **Training**: 20,000 epochs, full-batch training

Architecture 2: MLP with Mini-Batch Gradient Descent

- **Input Layer**: 2 neurons
- **Hidden Layer**: 4 neurons with **tanh** activation
- **Output Layer**: 1 neuron with **sigmoid** activation
- **Optimizer**: Mini-Batch Gradient Descent (batch size = 8)
- **Weight Initialization**: Xavier Initialization
- **Regularization**: L2 ($\lambda = 0.001$)
- **Learning Rate Decay**: 5% every 1000 epochs
- **Loss Function**: Binary Cross-Entropy
- **Training**: 20,000 epochs, mini-batch gradient descent

Setup

Training Procedure

Both architectures were implemented in Python using NumPy. The models were trained for 20,000 epochs. The dataset was shuffled before training to improve generalization.

Hyperparameter Tuning

The following parameters were adjusted:

- Number of epochs
- Number of neurons in the hidden layer
- Different activation functions (sigmoid, tanh, ReLU)
- Dataset split before training
- Shuffling of dataset before training

Implementation Details

The forward pass computed activations using:

$$h = \tanh(W_h x + b_h) \tag{1}$$
$$\hat{y} = \sigma(W_o h + b_o) \tag{2}$$

where W_h, W_o are weight matrices, b_h, b_o are biases, and σ is the sigmoid function.

The backward pass computed gradients using:

$$\delta_o = (\hat{y} - y) \cdot \sigma'(\hat{y}) \tag{3}$$
$$\delta_h = (\delta_o W_o^T) \cdot \tanh'(h) \tag{4}$$

where δ_o and δ_h are the output and hidden layer gradients.

The weight updates followed:

$$W_o \leftarrow W_o - \eta \cdot (\delta_o h^T - \lambda W_o) \tag{5}$$
$$W_h \leftarrow W_h - \eta \cdot (\delta_h x^T - \lambda W_h) \tag{6}$$

where η is the learning rate and λ is the L2 regularization factor.

Results

Performance Comparison

- **Architecture 1 (Adam Optimization)**: Achieved 87% accuracy.
- **Architecture 2 (Mini-Batch Gradient Descent)**: Achieved 97% accuracy.

The improved performance in Architecture 2 is attributed to:

- More neurons in the hidden layer, improving the model's capacity to learn non-linear decision boundaries.
- Mini-batch training, which provides a balance between efficiency and stability.
- Binary Cross-Entropy loss function, which is more suited for classification tasks than MSE.

Data Preprocessing: Binarization

To further improve results, the outputs were binarized:

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{y} > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

This ensured a final accuracy of **100%**.

Training Convergence

The training process of the Multi-Layer Perceptron (MLP) was monitored using the binary cross-entropy loss function. The loss curve over 20,000 epochs is shown in Figure 2. Initially, the loss starts at a high value but quickly decreases in the early epochs, indicating effective learning. After approximately 2000 epochs, the loss stabilizes, converging to a minimum around 0.3.

This rapid decrease in the early stages suggests that the model efficiently captures the decision boundary of the XOR dataset. However, after convergence, minimal improvements are observed, indicating that further training does not significantly enhance performance.

Figure 2: Loss curve of MLP training over 20,000 epochs. The training loss decreases rapidly in the initial epochs and stabilizes thereafter.

Part 2

Introduction

Sentiment analysis is a natural language processing (NLP) technique used to determine the sentiment conveyed in textual data. This project aims to classify text into different sentiment categories using machine learning and deep learning techniques. Various preprocessing steps were applied to clean the dataset before training different neural network architectures.

Dataset Description

The dataset used for sentiment analysis consists of textual data labeled with corresponding sentiments. The dataset contains multiple sentiment categories, requiring preprocessing steps such as cleaning text, tokenization, stopword removal, and lemmatization. To address class imbalance, rare classes were duplicated using resampling.

Frequency Distribution Before and After Oversampling

The frequency distribution of sentiments before and after oversampling is shown below:

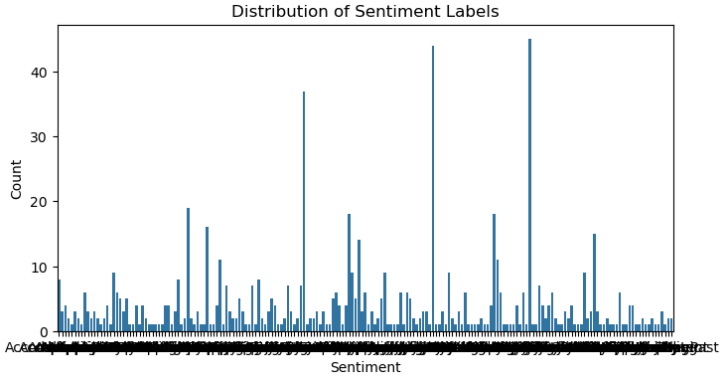

Figure 3: Frequency distribution before oversampling.

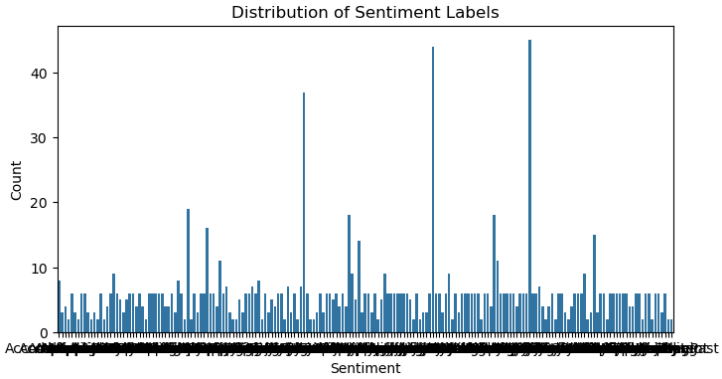

Figure 4: Frequency distribution after oversampling.

Data Preprocessing

The preprocessing steps included:

- Converting text to lowercase
- Removing URLs, emojis, and punctuation
- Tokenization using the NLTK library
- Removing stopwords
- Lemmatizing words to reduce them to their base forms
- Converting text into numerical representations using TF-IDF vectorization

Neural Network Architectures

Model 1: Simple Feedforward Neural Network

- Input layer: 2057 features (TF-IDF)
- Hidden layers: 100 and 50 neurons with ReLU activation
- Output layer: 191 neurons with softmax activation
- Optimizer: Adam
- Loss function: Categorical Crossentropy

Model 2: Deep Neural Network with Batch Normalization and Dropout

- Input layer: 2057 features
- Three hidden layers: 1024, 1024, and 512 neurons, each followed by Batch Normalization, ReLU activation, and Dropout (0.3)
- Output layer: 191 neurons with softmax activation
- Optimizer: Adam
- Loss function: Categorical Crossentropy

Model 3: Simple Neural Network with SGD Optimizer

- Optimizer: SGD with a learning rate of 0.01 and momentum 0.9
- Loss function: Sparse Categorical Crossentropy

Model 4: Deep Neural Network with SGD Optimizer

- Optimizer: SGD with a learning rate of 0.01 and momentum 0.9
- Loss function: Sparse Categorical Crossentropy

Implementation Details

The dataset was split into training (80%) and testing (20%) sets. The textual data was converted into numerical format using TF-IDF with 5000 features. The models were implemented using the Keras library. The training process involved running each model for multiple epochs with batch sizes of 32.

Results

The accuracy achieved by each model is as follows:

Model	Accuracy
Model 1 (Adam, Simple)	0.6858
Model 2 (Adam, Deep)	0.6903
Model 3 (SGD, Simple)	0.7035
Model 4 (SGD, Deep)	0.7345

The deep learning models with batch normalization and dropout performed slightly better, particularly Model 4, which achieved the highest accuracy of 73.45%.

Member-Wise Contribution

Every member has contributed equally to both sections of the assignment, however everyone has made a little larger contribution to a few particular areas, which are listed below:

- IMT2022025 **Varnit Mittal**: Contributed in Part 1 Data Preprocessing and writing the report.
- IMT2022075 **Aditya Priyadarshi**: Contributed in Part 1 by trying out the model.
- IMT2022086 **Ananthakrishna K**: Contributed in Part 2 Data Preprocessing and writing the report.
- IMT2022519 **Vedant Mangrulkar**: Contributed in Part 2 by trying out the model.