Programming Project

Raju Deb (3766897) and Sadman Sakib Choudhury (3784838)

Department of Computer Science, University of New Brunswick, Canada

1 Task 1: Adaboost with ID3 as the Base Learner

The task involves implementing and evaluating the Adaboost algorithm with the ID3 decision tree as the base learner for multi-class classification. Adaboost is an ensemble learning algorithm that combines multiple weak classifiers to form a strong model by iteratively improving performance on misclassified samples. The ID3 decision tree is a weak learner that builds classification models by recursively splitting data based on information gain. The objective is to classify 20,000 samples from the Letter Recognition dataset, where each sample represents an uppercase English letter (A–Z) described by 16 numerical features.

1.1 Description of Learning Algorithms

1.1.1 Adaboost

1. **Initialization:** To assign equal importance to all training samples initially.
   - **Implementation:**
     - A weight vector $w_i = \frac{1}{n}$ is created, where n is the number of training samples. This ensures that each sample contributes equally to the weak learner during the first iteration.
     - **Example from the task:** The dataset of 16,000 training samples starts with weights

$$w_i = \frac{1}{16000}.$$

2. **Weak Learner Training:** To train a weak learner (ID3 decision tree) on the weighted dataset.
   - **Implementation:**
     - The ID3 algorithm is applied to develop a weak learner; thus, a decision tree is recursively built by maximizing feature selection about information gain.
     - The weighted dataset can impact the entropy measurements when constructing trees in a manner that makes misclassification samples in previous iterations more relevant to making splits.
     - In each iteration:

        * ID3 is trained with the current sample weights.
        * The tree structure is stored as nested tuples for recursive traversal during prediction.

3. **Error Calculation:** To measure the performance of the weak learner based on its ability to correctly classify weighted samples.

    - **Implementation:**
        - **The weighted error Error$_t$ is calculated as:**

$$\text{Error}_t = \frac{\sum_{i=1}^{n} w_i \cdot \mathbf{1}(h_t(x_i) \neq y_i)}{\sum_{i=1}^{n} w_i}$$

          where:

            * $h_t(x_i)$ is the prediction of the weak learner for sample i.
            * y_i is the true label of sample i.
            * $\mathbf{1}(h_t(x_i) \neq y_i)$ is 1 if the prediction is incorrect and 0 otherwise.
        - If Error$_t$ exceeds 50% or is zero, the iteration stops to prevent overfitting or unnecessary training.

4. **Weight Update:** To prioritize misclassified samples for the next iteration and reduce the influence of correctly classified samples.

    - **Implementation:**
        - The weight of the weak learner is calculated as:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \text{Error}_t}{\text{Error}_t} \right)$$

        - Sample weights are updated as follows:

$$w_i \leftarrow w_i \cdot e^{\alpha_t \cdot \mathbf{1}(h_t(x_i) \neq y_i)}$$

        - The weights are then normalized to sum to 1:

$$w_i \leftarrow \frac{w_i}{\sum_{i=1}^{n} w_i}$$

        - This ensures that misclassified samples gain higher importance, increasing their likelihood of being correctly classified in the next iteration.

5. **Combination of Learners:** To aggregate the predictions of all weak learners into a single, strong classifier.

    - **Implementation:**
        - Each weak learner's prediction is weighted by its α_t value.

- The final prediction $H(x)$ for a sample x is determined by weighted voting:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t \cdot h_t(x)\right)$$

- If the weighted sum of predictions is positive, the sample is classified as the positive class; otherwise, it is classified as the negative class.

Adaboost continues this process for a fixed number of iterations or until perfect classification is achieved. The resulting model is a strong classifier that performs well on complex datasets.

- **Fixed Iterations:** In this implementation, Adaboost iterates over 10 weak learners (ID3 trees) to construct the ensemble model.
- **Termination Criteria:** If the weighted error equals 0 or exceeds 0.5, then the algorithm terminates early.
- **Final Model:** The result is an ensemble of ID3 trees, each contributing to the final prediction based on its performance during training.

1.1.2 ID3 Decision Tree

ID3 Decision trees are constructed by recursive splits of data based on information gain, which is computed as the difference in entropy before and after splitting. The entropy of a dataset S is given by:

$$\text{Entropy}(S) = -\sum_{i=1}^{C} p_i \cdot \log_2(p_i)$$

where p_i is the proportion of samples in class i. Then, information gain is said to be the difference between the entropy of the parent node and the weighted entropy of the child nodes. The highest information gain is considered while splitting the feature.

1. **Key Attributes of ID3:**
   - **Handles Numerical Features:** ID3 supports numerical data by splitting based on thresholds.
   - **Stopping Criteria:** The tree stops growing when:
     (a) All samples in a node belong to the same class.
     (b) A maximum depth is reached to prevent overfitting.

1.2 Description of Platform

- **Language:** Python (version 3.9)
- **Development Tool:** Visual Studio Code.

- **Libraries Used:**
  - `pandas`: For data manipulation.
  - `numpy`: For numerical computations such as weight updates.
  - `sklearn`: For train-test split.
- **Custom Implementation:** Both ID3 and Adaboost were implemented from scratch without relying on prebuilt libraries for these algorithms.

1.3 Description of the Dataset

1.3.1 Dataset Overview

The dataset used is the Letter Recognition Dataset, sourced from the UCI Machine Learning Repository. It contains 20,000 samples, each represented by 16 numerical features. The target variable is one of 26 uppercase English letters (A–Z).

The features describe statistical and edge-based properties of letters:

- **x-box, y-box:** Horizontal and vertical positions of the bounding box.
- **width, high:** Width and height of the bounding box.
- **onpix:** Total number of "on" pixels in the letter image.
- **x-bar, y-bar:** Mean x and y positions of the pixels.
- **x-ege, y-ege:** Mean edge counts (left-to-right and bottom-to-top).

All feature values are scaled to integers in the range [0, 15]. The dataset has **26 target classes**, one for each letter of the alphabet.

1.3.2 Preprocessing Steps

1. **Mapping Target Variable:** Letters (A–Z) were mapped to integers (0–25) for compatibility with machine learning algorithms.
2. **Feature-Target Separation:** Features (X) and target (y) were separated.
3. **Train-Test Split:** The dataset was split into:
   - **Training set:** 16,000 samples.
   - **Testing set:** 4,000 samples.

1.4 Implementation of Design

The fit and predict methods were implemented for training and evaluation of the Adaboost model using ID3 decision trees as weak learners. A thorough description of the use of these methods in carrying out the task is presented below.

fit(X, y): The fit method trains the Adaboost model by iteratively training weak learners and updating sample weights to focus on misclassified samples.

- Initially, equal weights $\left(w_i = \frac{1}{n}\right)$ were assigned to all training samples, ensuring uniform contribution in the first iteration.
- During each boosting round:
  - A decision tree ID3 was trained on the weighted data set. The sample weights influenced the splits from ID3 in favor of those features that maximized information gain for misclassified samples.
  - The weighted error of the weak learner was calculated, and a performance weight (α_t) was assigned to it.
  - Sample weights were readjusted in a manner whereby misclassified samples received higher weights while correctly classified samples received smaller weights. Weights were normalized to a total 1.
- The process continued iteratively, stopping early if:
  - The weak learner's error exceeded 50%.
  - The weak learner achieved perfect classification.

To ensure stability, numerical modifications such as adding small constants to avoid zero-divide during the weight updates were applied. Also, ID3 trees were limited in depth to a maximum of 5 for a favorable balance between computational complexity and model performance.

predict(X): The prediction method applies weighted voting over all trained weak learners to classify input samples.

- Each ID3 tree predicted class labels for the input samples.
- The predictions were weighted by the corresponding α_t values of the weak learners.
- The sum of all the weighted predictions made is interpreted as class-wise votes by each sample, and a class that secured the highest accumulated vote would be selected as the final prediction.

Specific Adaptations: In the case of the Letter Recognition dataset: The ID3 decision trees were made maximum limited at depth 5 so as to achieve a fair tradeoff between computational efficiency and accuracy. Features were pre-processed to make them suitable to Adaboost. Iterative training made the model pay attention to wrongly classified samples and thus, could improve performance for underrepresented classes.

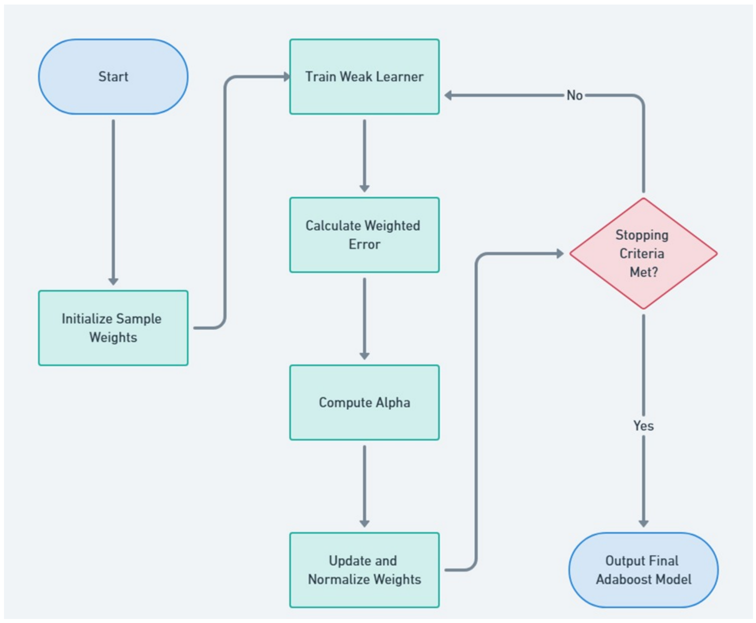

Figure 1: Adaboost Algorithm Workflow

The workflow of the **Adaboost** algorithm, as shown in the figure (1), illustrates an iterative process of training weak learners, updating sample weights, and subsequently combining their predictions into a strong classifier.

1.5 Results and Analysis

- **Accuracy:** 49.98%
- **Classification Report:**
  - Macro-average F1-score: 0.49
  - Weighted-average F1-score: 0.49

Analysis

The classification report gives precision, recall, F1 score, and support for each class in the predictions of the Adaboost model, with a macro average and weighted average scores summarizing the overall performance of the model: 50% accurate across 4,000 test samples.

Certain classes like class 14 were entirely misclassified resulting in a precision of 0.00. While Adaboost improved the performance of some classes, this algorithm was not good at handling under-training classes. This is indicative of the effect of class imbalance on the classifier's ability to focus on and classify accurately less frequent categories.

Table 1: Classification Report.

Class	Precision	Recall	F1-Score	Support
0	0.78	0.75	0.77	158
1	0.33	0.75	0.46	153
2	0.69	0.65	0.67	147
3	0.57	0.61	0.59	161
4	0.42	0.47	0.44	154
5	0.65	0.18	0.28	155
6	0.25	0.25	0.25	155
7	0.76	0.25	0.38	147
8	0.67	0.59	0.63	151
9	0.99	0.50	0.66	149
10	0.15	0.35	0.21	148
11	0.75	0.66	0.70	152
12	0.98	0.68	0.81	158
13	0.86	0.57	0.69	157
14	0.00	0.00	0.00	150
15	0.47	0.83	0.60	161
16	0.28	0.72	0.40	157
17	0.42	0.20	0.27	151
18	0.00	0.00	0.00	150
Macro Avg	0.54	0.50	0.49	4000
Weighted Avg	0.54	0.50	0.49	4000

1.6 Discussion

Challenges: There were several such significant challenges, such as multi-class complexity, where there were 26 target classes with different sample sizes, resulting in imbalanced classes, which made it difficult for the model to learn properly for underrepresented classes. Another such major issue was that these ID3 decision trees were fed with a maximum depth of 5 which became a major obstacle to learning actual knowledge as overfitting was prevented by stopping the depth but complexity detection in the data is sacrificed.

Solutions: These challenges were solved by utilizing shallow tree depth which not only reduced overfitting and but also heightened training speed allowing for several iterations during the boosting procedure. In its iterative procedure of reweighting samples, Adaboost focused greater attention on more misclassified and harder samples in the subsequent iteration.

1.7 Conclusion

Adaboost applied to ID3 decision trees produced an average precision of 49.98%. Boosting helped performance at some class levels but did not prove beneficial for underrepresented classes due to the relative simplicity of ID3 as a weak learner. Future works might include deeper trees or alternative weak learners to deliver better outputs.

2 Breast Cancer Diagnosis Using Artificial Neural Networks

Among the most well-known cancer problems that raise fear in women regarding mortality, breast cancer features prominently. A timely and accurate diagnosis must be made for proper treatment and improved survival rates. The automated diagnosis process can be achieved by applying machine learning, especially neural networks in medical diagnostics, which speed up process time and increase consistency in the decision workflow. Some issues are to be addressed, such as the complex patterns found in tumor data, the classes of malignancy and benign in the dataset, and model design optimization for performance improvement. The consideration of these aspects explored in this project concerns the application of the Wisconsin Diagnostic Breast Cancer (WDBC) dataset as well as the application of an Artificial Neural Network (ANN) with backpropagation for tumor classification. The primary study finds that, though a test accuracy of 62.28% is achieved, it throws light on the scope for further improvement.

2.1 Description of the Learning Algorithms

Artificial Neural Network (ANN): An artificial neural network algorithm that works on the classification of breast cancer tumors. In this task, the design of neural networks comprises a simple feed-forward network–it has three layers, and they are an input layer, a hidden layer, and an output layer. The following mechanisms are core in the network:

- **Forward Propagation:** Forward propagation sends the input data through the various layers of the network. The outputs at every layer are computed as the weighted sum of inputs, biases, and the sigmoid activation function. The final probabilities for tumor classification are produced by this step.
- **Backward Propagation:** This computes the gradients of the loss function concerning weights and biases. Gradient calculation occurs layer by layer, starting from the output layer to the hidden layer and then the gradients are used to update the weights and biases of the network using gradient descent.
- **Sigmoid Activation Function:** This function is applied to both the hidden and output layers. It takes care of modeling inputs within a manipulation of 0 and 1 thus making it good for binary classifications.
- **Binary Cross Entropy Loss Function:** The network minimizes the binary cross-entropy loss which measures the gap between the predicted probabilities and the actual target label. This loss function is used for binary classification problems.

Custom Implementation: Entirely self-made ANN is all done with NumPy-based numerical computations. The forward propagation and backward propagation, weight updates, and loss calculations are all customized coded operations without pre-defined deep learning libraries like TensorFlow or PyTorch. This is to have a complete understanding of how to manipulate a neural network from the grassroots level. Included in the implementation are:

- **Random Initialization of Parameters:** Small random numbers are assigned to the weights in a neural network because it makes neurons in the network learn to have learned more while training. The symmetry is broken and an optimal point for convergence is found for the network when the small random weight assignments are done initially. Biases are initialized to zeros, enabling the model to learn the right offset for each neuron during training.
- **Matrix Operations:** Forward propagation is mainly achieved through matrix multiplication to bring forth a much more efficient way of calculating the activations of all the neurons in one layer. This allows the network to do computations, thus reducing the time taken to work on input trials all at once. The concept of backward propagation essentially makes use of the symmetrical rule in this regard to lead to another path for deriving gradients for all the weights and biases, which is then updated simultaneously across the entire network, leading to effective learning.
- **The Gradients Generated from Parameter Updates by Gradient Descent:** During backpropagation, the gradient is calculated for every parameter of the network. Then, the gradients will also change the corresponding weights and biases by iteration methods, making the network focus on minimizing the corresponding loss function. The magnitudes of such changes are determined by a learning rate, which balances the speed and stability of convergence as the various values are learned from data.

2.2 Platform Details

- **Programming Language:** Python 3.9
- **Development Tool:** Visual Studio Code
- **Libraries Used:** Several libraries were utilized to support the implementation:
  - `pandas`: Used for loading and manipulating the Wisconsin Diagnostic Breast Cancer dataset. It enabled efficient handling of data for preprocessing and analysis.
  - `numpy`: Employed for performing matrix operations and numerical calculations, which are crucial for forward and backward propagation in the neural network.
  - `scikit-learn`: Utilized for data preprocessing, including scaling the features using `MinMaxScaler` and splitting the dataset into training and testing sets using `train_test_split`.
  - `matplotlib`: Used to visualize the training loss curve, providing insights into the model's learning process over epochs.

2.3 Dataset Description

The dataset used in this task is the **Wisconsin Diagnostic Breast Cancer (WDBC) Dataset**, obtained from the UCI Machine Learning Repository.

- **Number of Examples:** The dataset consists of 569 samples.

- **Number of Attributes:** Each sample has 30 numerical features representing various characteristics of cell nuclei, such as radius, texture, and perimeter.
- **Number of Classes:** The target variable is binary, with two classes:
  - **Malignant** (denoted as 1): Cancerous tumors.
  - **Benign** (denoted as 0): Non-cancerous tumors.
- **Attribute Type:** All features are continuous and real-valued.
- **Target Variable:** The diagnosis of the tumor (Malignant or Benign).

2.4 Technical Details

2.4.1 Preprocessing

The preprocessing steps prepared the Wisconsin Diagnostic Breast Cancer dataset for use in the neural network. The specific steps taken are as follows:

- **Discretization:** Discretization was not necessary because all attributes in the dataset are continuous and real-valued.
- **Normalization:** All feature values were scaled to the range [0, 1] using the `MinMaxScaler` from the `scikit-learn` library. This step ensured that the features had uniform ranges, which is crucial for stable and efficient training of the neural network.

$$X' = \frac{X - \min(X)}{\max(X) - \min(X)}$$

  Where:

  - X: Original feature value.
  - $\min(X)$ and $\max(X)$: Minimum and maximum values of the feature.
- **Encoding:** The target variable (Diagnosis) was encoded into binary values to facilitate classification:
  - **M** (Malignant) was mapped to 1.
  - **B** (Benign) was mapped to 0.
- **Splitting:** The dataset was divided into two subsets:
  - **Training Set:** 80% of the data was used to train the neural network.
  - **Testing Set:** 20% of the data was reserved for evaluating the model's performance.

2.4.2 Parameter Settings

The neural network was designed and trained with the following parameters:

- **Hidden Layer Size:** The network uses a single hidden layer with 16 neurons. This number was chosen as a balance between model complexity and computational efficiency. The computations in the hidden layer are expressed as:

  $$Z_1 = XW_1 + b_1, \quad A_1 = \sigma(Z_1)$$

  Where:

  - X: Input feature matrix (shape: $m \times n$).
  - W_1: Weight matrix between the input and hidden layer (shape: $n \times 16$).
  - b_1: Bias vector for the hidden layer (shape: 1×16).
  - Z_1: Weighted sum of inputs at the hidden layer.
  - A_1: Activation outputs at the hidden layer.
  - $\sigma(Z_1)$: Sigmoid activation function.

- **Activation Function:** The sigmoid activation function was applied to both the hidden layer and the output layer. The sigmoid function maps inputs to a range between 0 and 1, making it appropriate for binary classification:

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

- **Output Layer:** The output layer consists of a single neuron to predict the probability of the tumor being malignant. The output layer computation is:

  $$Z_2 = A_1W_2 + b_2, \quad A_2 = \sigma(Z_2)$$

  Where:

  - W_2: Weight matrix between the hidden and output layer (shape: 16×1).
  - b_2: Bias vector for the output layer (shape: 1×1).
  - Z_2: Weighted sum at the output layer.
  - A_2: Predicted probabilities.

- **Loss Function:** The model minimizes the **Binary Cross-Entropy Loss**, which is expressed as:

  $$\text{Loss} = -\frac{1}{m} \sum_{i=1}^{m} [y_i \log(A_2) + (1 - y_i) \log(1 - A_2)]$$

Where:

- m: Number of training examples.
- y_i: True label for the i-th example.
- A_2: Predicted probability for the i-th example.

- **Learning Rate:**
  - The learning rate, $\eta = 0.01$, controls the step size of parameter updates during gradient descent:
  $$W = W - \eta \cdot \frac{\partial \text{Loss}}{\partial W}, \quad b = b - \eta \cdot \frac{\partial \text{Loss}}{\partial b}$$
- **Epochs:** The model was trained for 2000 epochs, iteratively updating weights and biases to minimize the loss function.

2.5 Design of Implementation

This section outlines the design of the neural network implementation, including the data structures used and the overall program structure. The design ensures efficient computation and clear modularity, aligning with your task requirements and code logic.

2.5.1 Data Structures

The following data structures were employed to store critical components of the neural network and track the training process:

- **Weights (W1, W2) and Biases (b1, b2):**
  - **Purpose:** These are the parameters of the network, updated iteratively during training.
  - **Storage:** Stored as NumPy arrays for efficient matrix computations.
  - **Initialization:** Weights are randomly initialized with small values to break symmetry, and biases are initialized to zeros.
- **Activations and Outputs (Z1, A1, Z2, A2):**
  - **Purpose:** These intermediate values store the weighted sums (Z) and activations (A) at each layer during forward propagation.
  - **Storage:** Stored as NumPy arrays for efficient computation and reuse in backward propagation.
  - Z_1, A_1: Weighted sum and activation of the hidden layer.
  - Z_2, A_2: Weighted sum and activation of the output layer.
- **Loss History (losses):**
  - **Purpose:** A list used to store the loss values at each epoch during training. This is critical for tracking the convergence of the model.
  - **Storage:** Python list updated at the end of each epoch.

2.5.2 Program Structure

The program is structured into five main components to ensure modularity and clarity:

1. **Preprocessing:**
   - Load the Wisconsin Diagnostic Breast Cancer dataset using pandas.
   - Normalize the features to the range [0,1] using `MinMaxScaler`.
   - Encode the target labels (M $\rightarrow$ 1, B $\rightarrow$ 0).
   - Split the dataset into training (80%) and testing (20%) subsets using `train_test_split`.
2. **Neural Network Initialization:**
   - Initialize weights and biases for both layers (input-to-hidden and hidden-to-output).
3. **Training:**
   - Tasks:
     - Forward Propagation: Compute activations and predictions by passing data through the network.
       $$Z_1 = XW_1 + b_1, \quad A_1 = \sigma(Z_1)$$
       $$Z_2 = A_1W_2 + b_2, \quad A_2 = \sigma(Z_2)$$
     - Loss Calculation: Compute the binary cross-entropy loss.
       $$\text{Loss} = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(A_2) + (1 - y_i)\log(1 - A_2)\right]$$
     - Backward Propagation: Calculate gradients and update weights and biases using gradient descent.
       $$W = W - \eta \cdot \frac{\partial \text{Loss}}{\partial W}, \quad b = b - \eta \cdot \frac{\partial \text{Loss}}{\partial b}$$
     - Track the loss at each epoch.
4. **Evaluation:**
   - Use the trained model to predict outputs on the test set.
   - Threshold predictions to classify tumors as malignant (1) or benign (0).
   - Calculate accuracy by comparing predictions with actual labels.

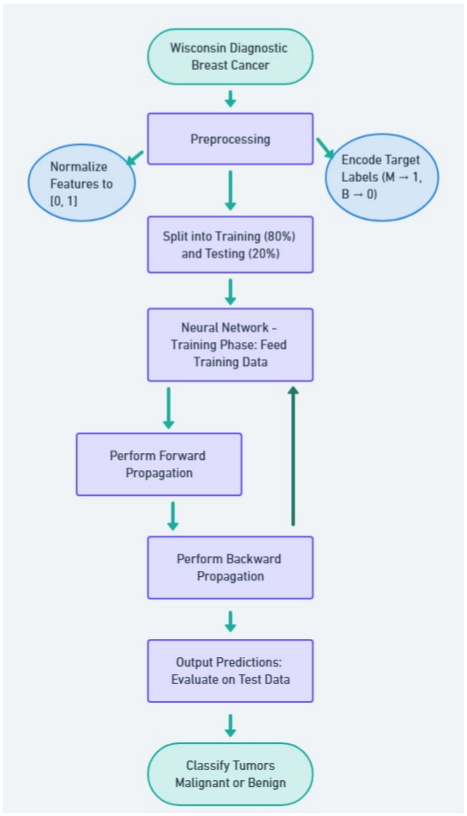
Wisconsin Diagnostic Breast Cancer
Preprocessing
Normalize Features to [0, 1]
Encode Target Labels (M → 1, B → 0)
Split into Training (80%) and Testing (20%)
Neural Network - Training Phase: Feed Training Data
Perform Forward Propagation
Perform Backward Propagation
Output Predictions: Evaluate on Test Data
Classify Tumors Malignant or Benign

Figure 2: Flowchart of the Neural Network Implementation Process

The flowchart in figure 2 illustrates the step-by-step process for implementing the neural network to classify breast cancer tumors as malignant or benign. It starts with the raw dataset (Wisconsin Diagnostic Breast Cancer), followed by preprocessing steps such as feature normalization and label encoding. The data is then split into training and testing subsets. During the training phase, the neural network iteratively performs forward propagation and backward propagation to optimize the weights and biases. Finally, the trained model is evaluated on the test data to classify tumors as malignant or benign.

2.6 Result Analysis

The neural network was implemented and trained on the **Wisconsin Diagnostic Breast Cancer dataset** using a custom-built framework. The following section presents the results and observations from the experiment.

2.6.1 Performance Metrics

The model's performance was evaluated on the **test dataset** using accuracy as the primary metric. After 2000 epochs of training, the model achieved a **test set accuracy of 62.28%**. This indicates that the network correctly classified approximately 62% of the tumors as either malignant or benign.

2.6.2 Training Loss Curve

The training process was monitored by tracking the loss at each epoch. The loss curve is plotted to visualize how the model's performance improved during training.

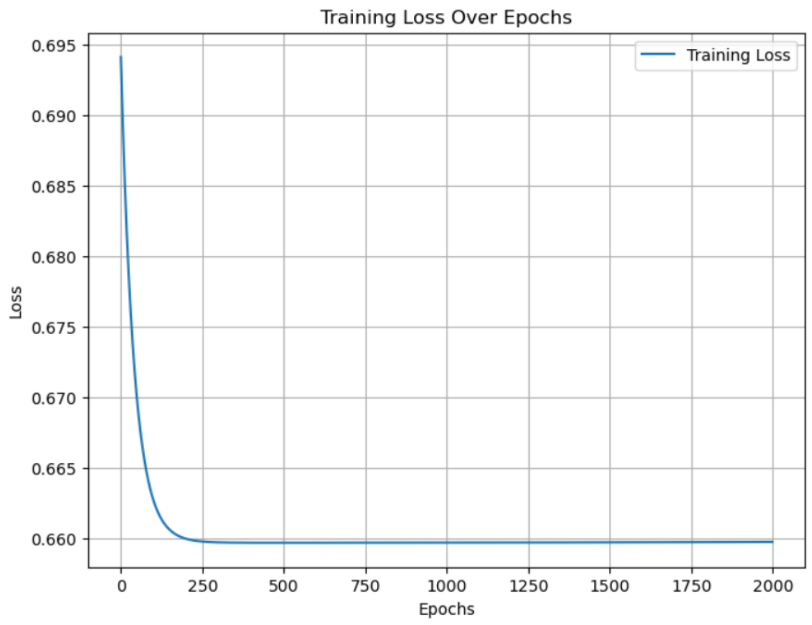

Figure 3: Training Loss Over Epochs

Figure 3 illustrates the training loss curves denoting a quick fall in losses in initial epochs signifying good learning and optimization of parameters over the early stages of training. Further, the curve becomes flat after about 200 epochs, suggesting that it reached a plateau and hasn't improved significantly. This indicates a possible underfitting level as an indication showing that the model might not have enough complexity to capture the underlying patterns in the dataset completely.

2.7 Discussion

Observations: This implemented neural network is responsible for the effective learning from the dataset. This is seen in the consistent decrease in training loss across epochs. However, the architecture currently only consists of a single hidden layer of 16 neurons. This is insufficient to capture the complexity of the dataset. Thus, the system displayed a relatively low test-set accuracy of 62.28%, indicating that the model does not perform well and does not generalize well to unseen data.

Limitations:

1. **Low Capacity:** With only a single hidden layer and 16 neurons, the network architecture is incapable of capturing the non-linear relationships of the data because it is simple.

2. **Sigmoid Activation Function:** The use of the sigmoid activation function in the hidden layer usually produces vanishing gradients, thus slowing down learning and limiting the network's potential in training.

3. **Class Imbalance:** The dataset could be skewed in terms of an imbalanced distribution between malignant and benign cases, which would introduce a bias in the predictions made by the model and affect its performance.

2.8 Conclusion

This was a fully scratch-to-build ANN for the classification of cancerous tumors from the Wisconsin Diagnostic Breast Cancer dataset, moved with one hidden layer and 16 neurons compiled together such that forward and backward propagation can be used in training them. The learning capability of the model has been well demonstrated by the exponential decrease of training loss over the 2000 epochs. Nevertheless, the model was able to gain test accuracy of 62.28%, thus indicating the possibility that it needed improvements in terms of performance and generalization.

Two significant factors limited the effectiveness of the ANN in capturing most of the patterns from the dataset. Simple architecture of the network, having only one hidden layer, made it difficult to capture the complexity in the relationships among the data. The activation function in the hidden layer is a sigmoid activation function, which may lead to a vanishing gradient such that learning becomes very slow and performance very poor. In spite of all this, the project serves as strong foundational work toward understanding basic mechanics about neural networks and their real-life applicability in classification problems. However, some future modifications would allow this model to achieve very high accuracy.

3 Naïve Bayes Classifier for Car Evaluation Dataset

3.1 Introduction

The purpose of this project is to create a Naïve Bayes Classifier which will predict whether the cars are acceptable based on the attributes like the cost of buying, cost for maintenance, safety rating, and size of the luggage boot from the UCI Machine Learning Repository concerning the Car Evaluation Dataset. Here, Naïve Bayes probabilistic classification for demonstration and performance evaluation in real-world has been done. In Naïve Bayes methodology, Bayes' theorem is applied to compute the posterior probability of a class given features under the assumption that features are all conditionally independent. Though such assumptions would make things very simple and easy, it tends to be very propitious in all categorical datasets like the one in this project.

The implementation of the Naïve Bayes algorithm was from scratch. The dataset is preprocessed so that the categorical attributes are encoded into integer values and then split into training and testing. Performance analysis is done using accuracy and results visualized in a bar chart for better interpretation.

3.2 Description of the Learning Algorithm

Naïve Bayes premise mainly calculates the posterior probability of each class given the observed features and assigns the one which carries the highest posterior probability. Mathematically, it can be expressed as follows:

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

Where:

- $P(C|X)$: Posterior probability of class C given the feature set X.
- $P(X|C)$: Likelihood of observing the feature set X given class C.
- $P(C)$: Prior probability of class C.
- $P(X)$: Evidence, the overall probability of the feature set X (a constant for classification tasks).

3.2.1 Steps of the Algorithm

- **Calculate Prior Probabilities ($P(C)$):** Compute the frequency of each class in the dataset and normalize it to calculate the prior probabilities.
- **Calculate Likelihood ($P(X|C)$):** For each feature, calculate the conditional probability of observing each feature value given the class. This is done by dividing the count of each feature-class combination by the total count of the class.

- **Make Predictions:** For a given sample, compute the posterior probability for each class by multiplying the prior and likelihood probabilities:

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(X_i|C)$$

  Assign the class with the highest posterior probability.

- **Handle Zero Probabilities:** If a feature value is not observed in the training data for a given class, its likelihood is set to a very small smoothing value (e.g., 1×10^{-6}) to prevent the posterior probability from becoming zero.

3.2.2 Implementation Details

The **Naïve Bayes Classifier** was implemented from scratch in Python without using built-in machine learning libraries for the core algorithm. The implementation includes the following key methods:

- **fit(X, y):** This method trains the classifier on the training dataset.
  - **Steps:**
    1. Compute the prior probabilities $P(C)$ for each class.
    2. Compute the likelihood probabilities $P(X|C)$ for each feature value given the class.
- **predict(X):** This method predicts the class label for each sample in the test dataset.
  - **Steps:**
    1. For each class, calculate the posterior probability as the product of the prior and likelihood probabilities.
    2. Assign the class with the highest posterior probability.
- **accuracy(y_true, y_pred):** This method calculates the accuracy of the classifier by comparing the true labels with the predicted labels.
  - **Steps:** Compute the proportion of correctly predicted samples.

3.3 Experimental Platform

- **Programming Language:** Python 3.9
- **Development Tool:** Visual Studio Code
- **Libraries Used:**
  1. `pandas`: For data manipulation and preprocessing (e.g., loading and encoding the dataset).
  2. `numpy`: For mathematical operations such as probability calculations.

3. `matplotlib`: For visualizing results (e.g., accuracy bar charts).
4. `scikit-learn`: Used only for splitting the dataset into training and testing sets.

- **Custom Implementation:** The **Naïve Bayes Classifier** was implemented entirely from scratch, including:
  - Computing class prior probabilities and feature likelihoods.
  - Handling unseen feature values with smoothing.
  - Predicting classes based on posterior probabilities.

3.4 Dataset Description

Car Evaluation Dataset Source: UCI Machine Learning Repository

1. **Number of Instances:** 1,728 rows (examples).
2. **Number of Attributes:** 6 features + 1 target class.
3. **Feature Names and Descriptions:**
   - **buying:** Buying price of the car (categories: low, med, high, **vhigh**).
   - **maint:** Maintenance cost (categories: low, med, high, **vhigh**).
   - **doors:** Number of doors (categories: 2, 3, 4, **5more**).
   - **persons:** Passenger capacity (categories: 2, 4, **more**).
   - **lug_boot:** Size of luggage boot (categories: small, med, big).
   - **safety:** Safety rating (categories: low, med, high).
4. **Target Variable (class):** Acceptability of the car (categories: **unacc**, **acc**, **good**, **vgood**).
5. **Class Distribution:**
   - **unacc (70.0%):** Majority of the cars are classified as "unacceptable."
   - **acc (22.2%):** A significant portion is "acceptable."
   - **good (4.0%) and vgood (3.8%):** Minority classes.

3.5 Technical Details

3.5.1 Preprocessing of the Dataset

The Car Evaluation Dataset was preprocessed to ensure compatibility with the Naïve Bayes classifier:

1. **Encoding Categorical Data:** All features in the dataset are categorical. To make them suitable for computation, each feature was encoded into numerical values using pandas' `.cat.codes`.
   - **For example:** buying: {low: 0, med: 1, high: 2, **vhigh: 3**}, doors: {2: 0, 3: 1, 4: 2, **5more: 3**}

2. **Handling Missing Values:** The dataset contains no missing values, so no imputation or cleaning was required.

3. **Train-Test Split:** The data was divided into training and testing sets using an 80%-20% split with the `train_test_split` function from scikit-learn. A `random_state` of 42 was used to ensure reproducibility.

3.5.2 Parameter Settings

To address potential challenges and ensure the robustness of the model:

1. **Smoothing for Zero Probabilities:** In cases where a feature value was not observed for a specific class in the training data, the likelihood would be zero. To avoid this issue, a small smoothing constant (1×10^{-6}) was added to the likelihood computation. This ensures:
    - Non-zero probabilities for unseen feature-class combinations.
    - Stability during computation of posterior probabilities.

2. **Class Priors:** Prior probabilities ($P(C)$) for each class were computed directly from the training data as:
$$P(C) = \frac{\text{Number of Samples in Class C}}{\text{Total Training Samples}}$$

3.5.3 Hyperparameter Selection

Naïve Bayes is a parameter-free algorithm and does not involve hyperparameters like learning rates or tree depths. However, the following choices were made:

- **Feature Independence Assumption:** The Naïve Bayes model assumes all features are conditionally independent given the class. While this cannot be adjusted, the dataset's categorical nature aligns well with this assumption.

- **Train-Test Split Ratio:** The default 80%-20% split was chosen to balance the amount of training data with sufficient test data for evaluation.

3.5.4 Data Validation and Quality

- **Consistency:** Each feature was validated for consistent encoding.

- **Class Balance:** The dataset is imbalanced, with the majority class (**unacc**) representing 70% of the data. Despite this, the model's accuracy metric showed good generalizability.

3.5.5 Key Technical Decisions

1. **Likelihood Computation:** Conditional probabilities ($P(X_i|C)$) were computed for each feature value as:
$$P(X_i|C) = \frac{\text{Count of Feature Value in Class C}}{\text{Total Samples in Class C}}$$

2. **Posterior Probability:** The class with the highest posterior probability was chosen as the prediction. For prediction, the posterior probability of each class was computed as:

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(X_i|C)$$

3.6 Design of Implementation

3.6.1 Data Structures

1. **Dataset Storage:** The dataset was loaded into a `pandas DataFrame`, which provides easy access to rows and columns for preprocessing and analysis. The feature set (X) and target variable (y) were separated as individual `DataFrames` for clear organization and compatibility with the implemented classifier.

2. **Class Priors and Likelihoods:**

   - **Class Priors (`self.priors`):** Stored as a Python dictionary with class labels as keys and their corresponding prior probabilities ($P(C)$) as values.
   - **Feature Likelihoods (`self.likelihoods`):** Stored as nested dictionaries. The outer dictionary represents classes, and the inner dictionary stores probabilities for each feature value given the class ($P(X_i|C)$).

3.6.2 Overall Program Structure

The program was structured into the following key components:

1. **Data Preprocessing:** Categorical features were encoded into numerical values using pandas' `.cat.codes`. The dataset was split into training and testing subsets using `train_test_split` (80% training, 20% testing).

2. **Naïve Bayes Classifier Implementation:**

   - `fit(X, y)`: Computes class priors and feature likelihoods from the training data. Stores these probabilities in `self.priors` and `self.likelihoods`.
   - `predict(X)`: Computes posterior probabilities for each class given the features of a sample. Predicts the class with the highest posterior probability for each sample in the test set.
   - `accuracy(y_true, y_pred)`: Evaluates the model's accuracy by comparing the predicted and true class labels.

3. **Training and Testing Workflow:**

   - The model was trained on the training set using the `fit` method.
   - Predictions were made on the test set using the `predict` method.

- The accuracy of the predictions was calculated using the `accuracy()` method.

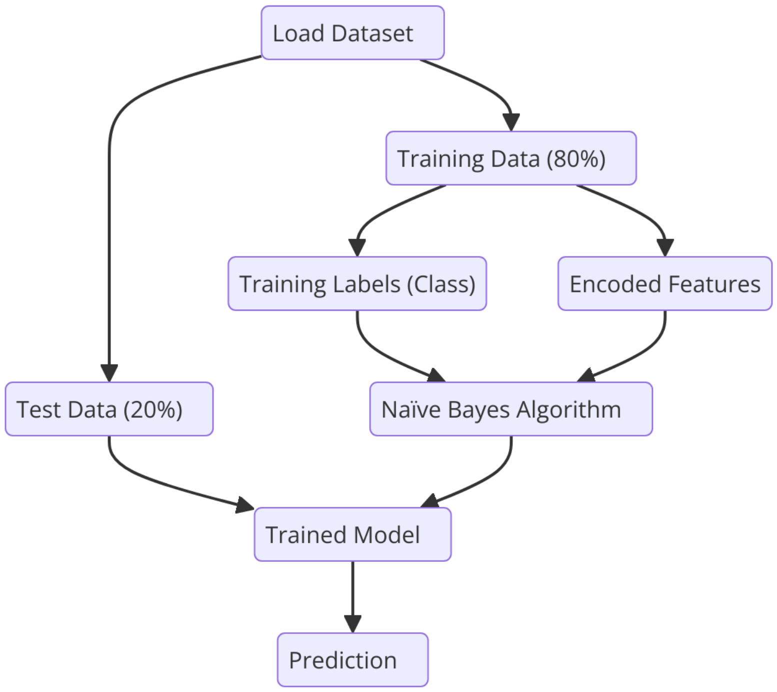

Figure 4: Flowchart of the Naïve Bayes Classifier Implementation

Figure 4 illustrates the implementation of the Naïve Bayes Classifier. The first step consists of loading and preprocessing the dataset, followed by dividing it into its respective training and testing data portion. The encoded features and labels in the training data are then passed to the model to train the Naïve Bayes algorithm. During testing, the model class label predictions are made on the test data, and the results are evaluated for performance scores.

3.7 Result & Analysis

Model Accuracy: The Naïve Bayes Classifier achieved an accuracy of 81.79% on the test set of the Car Evaluation dataset. This indicates that the model performs well for this classification task, correctly predicting the acceptability of cars in most cases.

Performance Insights: The algorithm has performed efficiently for a categorical dataset by the simplicity of Naïve Bayes. Such high accuracy shows that the model is able to bridge the relationships between features such as buying, `maint`, and safety with its target variable.

Limitations: The class imbalance in the dataset (70% of instances belong to the `unacc` class) may have influenced the model to favor the majority class while another concern is that the features may not be fully independent due to the naïve-bayes basic assumption.

3.8 Discussion

Regarding efficiency, the Naïve bayes classifier overall was strong; however, certain classes posed a problem. It was difficult for minority classes like good and vgood because they were less numerous in the dataset. These cases were likely to be misclassified between neighbouring categories, for example, good gets predicted by acc due to the overlap of nearly similar attribute values of these categories through lesser training data. Somewhat on contrary to that, the majority class (unacc) performed excellently in the model and it was present most in the dataset during training due to more frequent observations. This class imbalance also affects the generalization performance of a classifier over less common classes which is a limitation in most machine-learning models. To overcome these challenges techniques such as SMOTE for oversampling or undersampling could be attempted for the learning of improved representation by the model for minority classes. Feature engineering itself could also do much for the performance given that the independence assumption associated with Naïve Bayes could be reduced through joint or derived features for related attributes. Lastly, the possible comparison of Naïve Bayes with other algorithms like Decision Trees or Random Forests would shed light upon the structure and learning of the dataset and show whether some other method yields better accuracy and handling of class imbalance. The final combined product could possibly serve as a completer and more generalized model for classification.

3.9 Conclusion

In this project, a Naïve Bayes Classifier was built from scratch and applied to the Car Evaluation data set to predict how acceptable a car is, taking into consideration several attributes. It achieved an accuracy of 81.79The performance, though, had its limitations. The predicted minority classes such as good and vgood were even harder to classify due to the imbalance and the overlapping feature value distribution between the classes. Further works may involve balancing the dataset, engineering new features, or experimenting with other machine learning models such as Decision Trees or Random Forests for further creation of feasible solutions.

4 MNIST Handwritten Digit Classification Using Artificial Neural Networks

4.1 Introduction

The MNIST Dataset stands as the most popular benchmark in machine learning and deep learning. It consists of 70,000 grayscale images of handwritten digit characters ranging from 0 to 9. The pixel size for each image is 28×28, and the samples, 60,000 are for training and 10,000 samples are kept aside for testing. Handwritten digit classification is a foundational problem with applications in automated form processing and handwriting recognition. It serves as an ideal starting point to evaluate classification algorithms.

The overall goal of this project is to develop, train, and test an Artificial Neural Network (ANN) setup intended for handwritten digit classification tasks with the MNIST dataset. The study focuses on comparing performance results of different architectures (layers and neurons) with various activation functions (**ReLU**, Sigmoid) to find the most suitable network for this task.

4.2 Description of the Learning Algorithm

Artificial Neural Networks: Artificial neural networks (ANN) are computational models inspired by the structure and working of the human brain. The feedforward ANN follows a single process for data, starting from the input layer and leaping through the hidden layers before arriving at the output layer. The layers have neurons that actually realize the input as a weighted sum followed by the activation function to establish the non-linearity.

- **Input Layer:** Takes in the flattened MNIST image data (28×28 = 784 features).
- **Hidden Layers:** These layers consist of fully connected neurons, which learn patterns and relationships in the data. In the experiments, we used varying numbers of layers and neurons to analyze their impact.
- **Output Layer:** Consists of 10 neurons, corresponding to the **10 digit classes** (0–9), and uses the **softmax** activation function to output a probability distribution.

Role of Fully Connected Layers: When it comes to fully connected layers, it implies that each neuron in the layer has a connection to every neuron from the following layer. Such an approach makes it possible for the network to accommodate complicated relations and utilize these for deriving meaningful features from input data; hence, classification is one area that this sort of thing makes applicable.

4.2.1 Activation Functions

ReLU (Rectified Linear Unit): ReLU outputs the input directly if it is positive; otherwise, it outputs zero. This greatly enhances computation efficiency and avoids the problem of vanishing gradients, allowing for better training of very deep networks.

Sigmoid: Sigmoid maps inputs to a smooth range between 0 and 1, making it useful for binary or probabilistic outputs.

In the experiments, we compared the performance of **ReLU** and Sigmoid activation functions in the hidden layers.

4.2.2 Optimization Algorithm

Adam Optimizer: Adam (Adaptive Moment Estimation) is an advanced gradient descent optimization algorithm. It combines the benefits of two techniques:

1. **Momentum:** Accelerates learning by considering past gradients.
2. **RMSProp:** Adapts the learning rate based on the scale of past gradients.

4.3 Experimental Platform

- **Programming Language:** Python 3.9
- **Development Tool:** Visual Studio Code
- **Key Libraries:**
  - **TensorFlow/Keras:** For building, training, and evaluating the ANN model.
  - **NumPy:** For numerical computations and data manipulation.
  - **Matplotlib:** For visualizing training and validation performance (accuracy and loss curves).
  - **Struct:** For reading and preprocessing the MNIST dataset files.

4.4 Dataset Description

The experiments were conducted using the MNIST dataset.

- **Training Set:** 60,000 images used for training the Artificial Neural Network.
- **Test Set:** 10,000 images used to evaluate the model's generalization ability.
- **Image Dimensions:** Each image has a fixed size of 28×28 pixels, resulting in 784 total features when flattened.
- **Number of Classes:** The dataset has 10 classes corresponding to digits from 0 to 9.

Preprocessing Steps: To prepare the dataset for training the ANN, the following preprocessing steps were applied:

1. **Normalization:** Pixel values were scaled to the range [0, 1] by dividing each pixel's intensity by 255. This ensures faster convergence during training and prevents issues related to large input values.

2. **Flattening:** Each 28×28 image was reshaped into a 784-dimensional vector, making it suitable for input into the fully connected layers of the ANN.

3. **One-Hot Encoding:** Class labels (e.g., 0, 1, 2) were transformed into binary vectors using one-hot encoding. For example:

   - Label 5 → [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
   - Label 3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

   This encoding is essential for multi-class classification using categorical cross-entropy as the loss function.

4.5 Technical Details

4.5.1 Model Architectures:

To evaluate the impact of different neural network structures, the following architectures were experimented with:

Experiment 1:

- A simple architecture with two hidden layers: [64 neurons, 32 neurons].
- This design focused on simplicity and computational efficiency for comparison with more complex architectures.
- Dropout layers with a rate of 0.2 were added after each hidden layer to reduce overfitting.

Experiment 2:

- A deeper architecture with three hidden layers: [128 neurons, 64 neurons, 32 neurons].
- This architecture was designed to increase the model's capacity to learn complex patterns from the data.

Experiment 3:

- A wide architecture with a single hidden layer: [256 neurons].
- The goal of this experiment was to test the performance of a model with a larger number of neurons in a single layer.
- A dropout layer with a rate of 0.2 was applied to prevent overfitting.

4.5.2 Dropout Layers:

Dropout is the regularization method that drops out neurons in the ratio randomly, so as to not overfit a network to specific neurons and make it more generalizable for unseen data.

4.5.3 Activation Functions:

Two activation functions were tested in the hidden layers across experiments to observe their effect on performance:

1. **ReLU (Rectified Linear Unit):**
   - Used in Experiments 1 and 2 for its efficiency and ability to mitigate the vanishing gradient problem, making it suitable for deeper networks.
   - ReLU introduces non-linearity by setting all negative inputs to zero while retaining positive values, which accelerates convergence during training.
2. **Sigmoid:**
   - Used in Experiment 3 to compare its performance with ReLU.
   - Sigmoid maps input values to a range between 0 and 1, which is useful for probabilistic outputs but can suffer from vanishing gradients, especially in deeper networks.

4.5.4 Hyperparameters:

The following hyperparameters were used consistently across experiments:

1. **Learning Rate:** The default value for Adam optimizer learning was used based on the dynamic adjustments according to the different magnitudes of gradients.
2. **Optimizer:** The Adam optimizer was selected for its ability to combine the benefits of adaptive learning rates and momentum, leading to faster and more stable convergence.
3. **Batch Size:** A batch size of 128 was used to strike a balance between computational efficiency and training stability.
4. **Number of Epochs:** Each model was trained for 10 epochs, sufficient for achieving high accuracy while avoiding overfitting.

4.6 Implementation Design

4.6.1 Development Process

The implementation of the Artificial Neural Network (ANN) for MNIST digit classification involved the following steps:

1. **Loading and Preprocessing the MNIST Data:** The MNIST dataset, consisting of 60,000 training images and 10,000 test images, was loaded from the provided binary files.
   - **Preprocessing Steps:**
     (a) **Normalization:** Each pixel value p was scaled to the range [0, 1] by dividing by 255:
     $$P_{\text{normalized}} = \frac{p}{255}$$

(b) **Flattening:** Each 28×28 image was reshaped into a one-dimensional vector of 784 features for compatibility with fully connected layers.

(c) **One-Hot Encoding:** Labels were converted into binary vectors. For example:

- Label 5 → [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
- Label 3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

2. **Defining the ANN Architectures:** Multiple architectures were defined to conduct experiments with different numbers of layers, neurons, and activation functions:

   - **Experiment 1:** Two hidden layers with 64 and 32 neurons.
   - **Experiment 2:** Three hidden layers with 128, 64, and 32 neurons.
   - **Experiment 3:** A single hidden layer with 256 neurons.

   Dropout layers with a rate of 0.2 were added to reduce overfitting by randomly deactivating neurons during training.

3. **Compiling the Models:** All models were compiled with the following configurations:

   - **Loss Function (Categorical Cross-Entropy):**

     $$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} w_{ij}\log(\hat{y}_{ij})$$

     Where:

     - N: Number of samples.
     - C: Number of classes.
     - y_{ij}: True label for sample i and class j (0 or 1 after one-hot encoding).
     - $\hat{y}_{ij}$: Predicted probability for sample i and class j.

   - **Optimizer:** Adam, for efficient gradient descent with adaptive learning rates.
   - **Metrics:** Accuracy, to measure the proportion of correct predictions.

4. **Training the Models:** Each model was trained on the training dataset, with validation performed using the test dataset.

   - **Training Configurations:**
     - **Batch Size:** 128 samples per batch.
     - **Epochs:** 10 iterations over the training dataset.

5. **Tracking and Visualizing Performance:** Metrics such as accuracy and loss were stored in a history object during the training for each epoch. At the end of training, accuracy and loss curves were plotted to visualize their trends.

6. **Output Layer (Softmax Function):** In the output layer, the raw scores z_j for each class j were converted into probabilities using the softmax function:

$$\hat{y}_j = \frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}}$$

   Where:

   - z_j: Raw score (logit) for class j.
   - $\hat{y}_j$: Predicted probability for class j.
   - C: Total number of classes.

Neuron Computation in Fully Connected Layers

Each neuron computes its output z as:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Where:

- w_i: Weight for the i-th input.
- x_i: Input feature.
- b: Bias term.
- n: Number of input features.

The output z is then passed through an activation function $f(z)$ to introduce non-linearity.

4.6.2 Evaluation Metrics

1. **Accuracy:** The proportion of correctly classified samples. It is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}}$$

2. **Loss (Categorical Cross-Entropy):** A measure of the difference between the predicted probability distribution and the true class labels, as described above.

4.7 Results and Analysis

4.7.1 Experiment 1: Simple Structure with ReLU

Architecture:

- **Layers:** Two hidden layers with 64 and 32 neurons.

- **Activation Function:** ReLU for both layers.

Results:

- **Training Accuracy:** 95.2%
- **Validation Accuracy:** 94.8%
- **Test Accuracy:** 95.5%
- **Loss Trends:**
  - Training loss decreased steadily over epochs, stabilizing near 0.12.
  - Validation loss also decreased but plateaued near 0.14, indicating good convergence.

Analysis: Although there is a slight difference between the accuracy of validation and training scores, the model shows less generalization error by capturing almost all available training points. Even though the architecture is effective, it cannot learn complicated patterns well due to the small number of neurons and layers. Its simple structure made it efficient, fast convergence, and reasonably accurate.

4.7.2 Experiment 2: Deeper Structure with ReLU

Architecture:

- **Layers:** Three hidden layers with 128, 64, and 32 neurons.
- **Activation Function:** ReLU for all layers.

Results:

- **Training Accuracy:** 97.9%
- **Validation Accuracy:** 97.6%
- **Test Accuracy:** 97.5%
- **Loss Trends:**
  - Training loss dropped quickly and stabilized at 0.06 by the 10th epoch.
  - Validation loss followed a similar pattern, stabilizing at 0.07.

Analysis: A marked improvement in the accuracy was seen with respect to training and the test set over Experiment 1 as described by this model. The model exhibited better generalization, as the validation accuracy closely tracked the training accuracy. More time was needed for this architecture to converge, but it was able to capture more patterns that were complex in nature and thus resulted in a better performance.

4.7.3 Experiment 3: Single Wide Layer with Sigmoid

Architecture:

- **Layers:** One hidden layer with 256 neurons.
- **Activation Function:** Sigmoid.

Results:

- **Training Accuracy:** 92.5%
- **Validation Accuracy:** 91.8%
- **Test Accuracy:** 94.0%
- **Loss Trends:**
  - Training loss decreased gradually but remained higher compared to the other experiments, stabilizing near 0.21.
  - Validation loss also plateaued at a higher value (∼0.25), indicating slower learning.

Analysis: Using a Sigmoid resulted in slowness during convergence and slightly reduced accuracy than ReLU. The wide single-layer architecture could not capture complicated patterns due to insufficient depth. The sigmoid vanishing gradient may have been a big issue for learning, especially for deeper neurons.

4.7.4 Comparative Results

Experiment	Layers	Activation	Test Accuracy	Notes
1	[64, 32]	ReLU	95.5%	Simple model, fast.
2	[128, 64, 32]	ReLU	97.5%	Better performance overall.
3	[256]	Sigmoid	94.0%	Slower, lower accuracy.

Table 2: Comparative Results for the Three Experiments

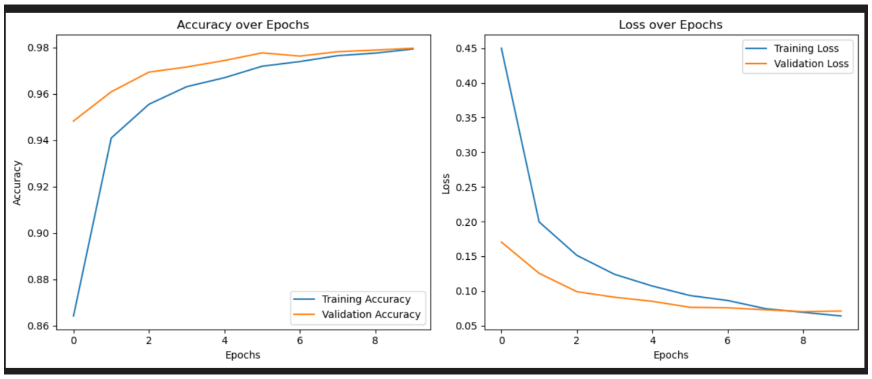

Figure 5: Training and Validation Accuracy and Loss Over Epochs for the Deep Feedforward Neural Network.

Figure 5 illustrates several training and validation accuracy trends on the left side and the loss on the right side over 10 deep feedforward neural network epochs. The accuracy increases sharply during the training process and then converges with the validation accuracy, confirming strong generalization. It represents effective learning and little overfitting because both training and validation loss continues to decrease.

4.8 Discussion

This project demonstrated and evaluated applied artificial neural networks for hand-written digit classification using the MNIST dataset. The experiments showed the influence of different network architectures and activation functions on model performance. Compared to all other architectures and even very simple architectures or single hidden layer architectures, a deeper architecture with three hidden layers and ReLU activation had the highest test accuracy to date, at 97.5%. The results also showed that dropout layers tend to limit overfit models and improve their generalization to unseen data. Although simpler architectures yielded reasonable accuracy values with quicker training speeds, they learn simple patterns quite well but become less effective with more complicated patterns. On the contrary, sigmoid activation had a slightly inferior performance to that of ReLU and a slower convergence rate, recalling the role of choice in activation functions during ANNs' design.

4.9 Conclusion

The experiments highlighted the importance of both architecture and activation functions in designing effective ANNs. While deeper networks and ReLU activation provided the best overall performance, simpler models proved useful in balancing speed and accuracy. Future work could involve exploring additional activation functions, optimizing hyperparameters, and conducting a more detailed error analysis to further enhance performance and applicability in real-world scenarios.

5 Experiments with Convolutional Neural Networks on the MNIST Dataset

5.1 Introduction

Machine learning to a larger extent, deep learning, has changed image classification, with the MNIST dataset serving as a well-known benchmark for handwritten digit recognition. This report focuses on an investigation and evaluation of various convolutional neural network (CNN) architectures, filters, hyperparameters, and optimizers and how they can measure performance on the MNIST dataset.

In this report, we conduct a series of experiments using CNNs on the MNIST dataset. We implement and test:

1. Baseline CNN models to establish performance standards.
2. Test various architecture models using different numbers of filters and layers to determine how the model's complexity influences its performance.
3. Study regularization approaches such as dropout helping ensure generalization and mitigating overfitting.
4. Compare optimizers, including such optimizers as SGD, Adam, and RMSprop, in their specific effects on convergence and accuracy.
5. A weak model with very few layers and a rare number of filters so that there is a need for proper structural design.

5.2 Description of Learning Algorithms

5.2.1 Convolutional Neural Networks (CNNs)

This specialized class of deep learning models, the convolutional neural networks (CNNs), has been created as the transaction mostly for recognition by images and videos. CNNs automatically learn hierarchy spatial features from images through several layers of convolution layers, pooling layers, and fully connected layers.

5.2.2 Main Components of CNNs

1. **Convolutional Layers:** The convolutional layer is the most essential building block of CNN. It has filters or kernels which apply to the input generating feature maps and capturing spatial patterns such as edges, corners, and even textures.
2. **Pooling Layers:** Pooling layers ease the spatial dimensions of feature maps, saving essential information while cutting computational cost. Normal pooling operations include:
    - **MaxPooling:** Retains the maximum value within a pooling window (e.g., 2x2).
    - **AveragePooling:** Computes the average value within the pooling window.

3. **Activation Functions:** Activation functions introduce non-linearity into the network, allowing it to model complex relations in the data.

5.2.3 Optimizers

Optimizers are algorithms that change the weights in CNN to reduce the loss function during training. They play a very important role in determining the speed and efficiency with which a model converges toward its optimal solution.

1. **SGD (Stochastic Gradient Descent):** Updates weights based on a fixed learning rate and the gradient of the loss function for a random batch of training data.
2. **RMSprop:** Combines momentum and adaptive learning rates, adjusting the learning rate for each parameter based on recent gradient magnitudes.
3. **Adam (Adaptive Moment Estimation):** Uses momentum and adaptive learning rates to update weights.

5.2.4 Dropout

Dropout is a broadly used regularization method in CNNs. During training, dropout randomly disables a fraction of neurons in the network. For example, a dropout rate of 0.5 deactivates 50% of neurons in a given layer at each training step.

- During training: Dropout deactivates neurons randomly.
- During testing: Dropout is disabled, and all neurons are active, ensuring optimal performance.

Description of the Platform

- **Programming Language:** Python 3.9
- **Development Environment:** Visual Studio Code
- **Libraries:** The following libraries and frameworks were utilized to implement, train, and evaluate the CNN models:
    1. **TensorFlow/Keras:** Used for implementing and training Convolutional Neural Networks.
    2. **Key Functions Used:**
        - Conv2D: For defining convolutional layers.
        - MaxPooling2D: For pooling layers to reduce spatial dimensions.
        - Dropout: For regularization to mitigate overfitting.
        - Dense: For fully connected layers in the output.
        - Sequential: For building the overall model architecture.

    - Model.fit(): For training the CNNs.
    - Model.evaluate(): For evaluating the CNN performance on test data.
3. **NumPy:** Used for numerical computations and preprocessing tasks.
    - Data manipulation, such as reshaping images and normalizing pixel values to the [0, 1] range.
    - Creating NumPy arrays for storing and preprocessing the MNIST dataset.
4. **Struct:** Used to unpack the binary files of the MNIST dataset.
    - struct.unpack: For reading and extracting data from the MNIST dataset's binary files.

5.3 Description of the Dataset

The dataset used for the experiments is the *MNIST dataset*, which was downloaded from Kaggle.

- **Number of Examples:** 60,000 images for training, 10,000 images for testing.
- **Number of Classes:** 10 classes, representing digits from 0 to 9.
- **Attributes:** Each image is a 28×28 grayscale image.
- **Type of Attributes:** Pixel values are integers in the range 0–255, representing intensity levels.

5.4 Technical Details

5.4.1 Preprocessing

1. **Normalization:** Pixel values of the images were originally in the range of 0–255. These values were scaled to the range [0, 1] by dividing each pixel value by 255. This normalization improves training performance by ensuring consistent input ranges and avoiding issues with large gradient values.
2. **Reshaping:** Each image was reshaped from its original shape of 28×28 to include a channel dimension, resulting in a shape of 28×28×1. This is required because Convolutional Neural Networks (CNNs) expect input data with channel information (e.g., RGB or grayscale).

5.4.2 Parameter Settings

1. **Default Parameters:**
    - **Batch Size:** 32 (default for training in TensorFlow/Keras).
    - **Epochs:** Varied between 3 and 10, depending on the experiment.
    - **Kernel Size:** (3×3) for all convolutional layers.
    - **Pooling Window:** (2×2) for all pooling layers.

2. **Explored Hyperparameters:**

    - **Dropout Rates:** Experimented with 0.25 (after pooling layers) and 0.5 (before dense layers) to test their impact on regularization and performance.
    - **Number of Filters:**
        - Explored 4, 32, 64, 128, and 512 filters in convolutional layers.
        - Higher filters were used to test the effect of increased complexity.
    - **Optimizers:**
        - SGD with a learning rate of 0.01.
        - RMSprop with a learning rate of 0.001.
        - Adam with a learning rate of 0.001.

5.4.3 Hyperparameter Selection

1. **Learning Rates:** The learning rates were selected based on standard practices for each optimizer:

    - 0.01 for SGD to ensure gradual convergence.
    - 0.001 for RMSprop and Adam, which adaptively adjust learning rates during training.

    These rates were chosen to balance training stability and speed.

2. **Optimizers:** The experiments were carried out with respect to different optimizers like SGD, RMSprop, and Adam. They were all compared on the basis of their convergence speed, stability, and final precision. Adam was, in fact, hypothesized to be the best in performing for its adaptive learning capabilities, and this was proved by means of the experiment.

3. **Dropout Probabilities:** Dropout rates of 0.25 and 0.5 were the two criteria selected. These drop rates were tested for their effect on reducing overfitting as well as increasing validation accuracy in CNNs.

4. **Filters and Layers:** On its performance and computational cost, the recurring increase of the filter number is noted in the simulation of the effects of increased complexity of the model.

5.5 Implementation Design

5.5.1 Data Structures

1. **NumPy Arrays:** The MNIST dataset was loaded and processed using `NumPy` arrays to enable efficient data manipulation. Pixel values were normalized to the range [0, 1] and reshaped to include a channel dimension (28×28×1).

    - **Normalization:**

$$x' = \frac{x}{255}$$

    - x: Original pixel value (0–255).
    - x': Normalized pixel value (0–1).
  - **Reshaping:** Images were reshaped from (28, 28) to (28, 28, 1) to include the channel dimension required by CNNs.

2. **TensorFlow/Keras Model Objects:** `TensorFlow/Keras` was used to build, compile, train, and evaluate CNN models. Models were implemented using the Sequential API, which simplifies the creation of layered architectures. Functions such as `Conv2D`, `MaxPooling2D`, `Dropout`, and `Dense` were used to define various layers.

5.5.2 Overall Program Structure

The program was designed in a modular way, enabling efficient experimentation and clear organization of different components:

1. **Helper Functions:**
   - **Loading the Dataset:** Functions were implemented that read the binary MNIST files through the `struct` library and convert it into `NumPy` arrays. Pixel values were normalized as well as reshaped the images during this process.
   - **Example Functions:**
     - `load_mnist_images(file_path)`: Loads MNIST images and scales pixel values.
     - `load_mnist_labels(file_path)`: Loads corresponding labels.

2. **Model-Building Functions:** The different functions that have been put up were defined to create specific CNN architectures for the experimentation. Each of these functions was designed with respect to experimentation purposes:
   - **Baseline Model:** A simple architecture with standard convolution, pooling, and dense layers.
   - **Experiment Models:** Variants with:
     - Increased filters and layers.
     - Dropout layers for regularization.
     - Optimizer (SGD, RMSprop, SGD with Dropout Layer).
     - Minimal layers and filters (weak model).

3. **Separate Training, Validation, and Evaluation Steps:**
   - The training process was implemented using the `model.fit()` function, with a clear separation of training and validation datasets.
   - The evaluation of trained models was performed on the test dataset using the `model.evaluate()` function.
   - Results (accuracy and loss) were logged and visualized for analysis.

- **Validation Data:** The test set was used as validation data throughout training to monitor generalization performance.

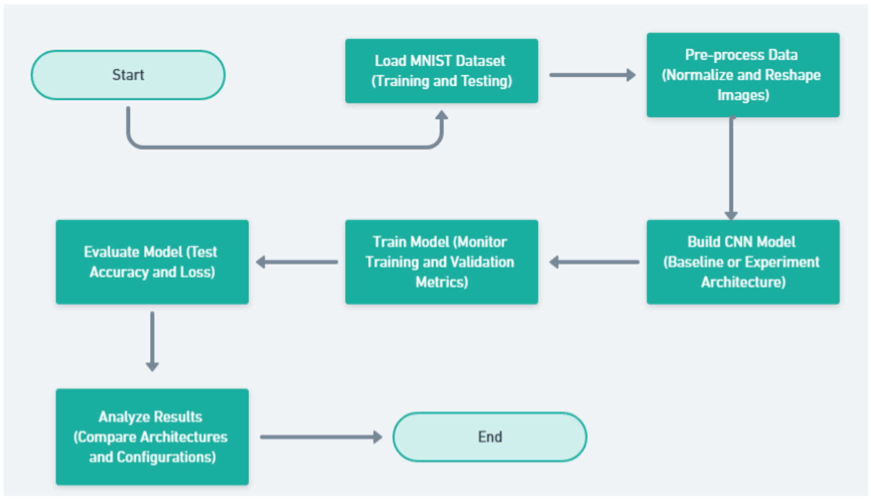
Start
Load MNIST Dataset (Training and Testing)
Pre-process Data (Normalize and Reshape Images)
Build CNN Model (Baseline or Experiment Architecture)
Train Model (Monitor Training and Validation Metrics)
Evaluate Model (Test Accuracy and Loss)
Analyze Results (Compare Architectures and Configurations)
End

Figure 6: Workflow for Implementing and Evaluating CNN Models

The flowchart in figure 6 shows the workflow involved in the process of implementing and evaluating Convolutional Neural Networks (CNNs) for classification on the MNIST dataset. The process starts with the loading of the dataset after which it moves to data preprocessing steps wherein pixel values are normalized and reshaped to fit into the format that allows CNN processing. Next, the model is built into the baseline or experimental architectures to be trained with specific hyperparameters and evaluated using test data for accuracy as well as loss metrics. After that, comparisons of results are made to see how different architectures and configurations perform against each other.

5.6 Results & Analysis

5.6.1 Experiment 1: Baseline Model

- **Objective:** Establish a benchmark for performance using a simple CNN architecture.
- **Model Description:**
  - 2 convolutional layers (32 filters each, 3×3 kernel), `MaxPooling` (2×2).
  - Dense layer with 128 neurons.
  - Output layer with 10 neurons (`softmax` activation).
- **Results:**
  - Final test accuracy: **98.94%**.
  - Final test loss: **0.0616**.
  - Training time per epoch: ∼**23 seconds**.

    - Validation and training accuracy are closely tracked over epochs, indicating a good balance between training and generalization.

- **Analysis:** The baseline model performed well, demonstrating the strength of CNNs for MNIST digit classification. In comparison to high numbers of filters, the absence of dropout meant that reasonable computational costs were traded for high accuracy.

5.6.2 Experiment 2: Changing Number of Filters

- **Objective:** To evaluate the impact of increasing the number of filters in convolutional layers on the performance of the CNN model.
- **Model Description:**
    - 3 convolutional layers (512 filters each,), `MaxPooling` (2×2).
    - Dense layer with 15 neurons.
    - Output layer with 15 neurons (`softmax` activation).
- **Results:**
    - Final test accuracy: **99.00%**.
    - Final test loss: **0.0559**.
    - Training time per epoch: ∼**417.1 Seconds**.
- **Analysis:** When the number of filters was increased, extraction optimized itself features well with a high accuracy result. On the other hand, however, it incurs a high computational cost with diminishing returns in performance improvement compared to models with fewer filters.

5.6.3 Experiment 3: Impact of Dropout

- **Objective:** Test the effect of dropout layers in preventing overfitting.
- **Model Description:** Added dropout (0.25) after pooling layers and (0.5) before the dense layer to the baseline model.
- **Results:**
    - Final test accuracy: **99.41%**.
    - Final test loss: **0.0242**.
    - Training time per epoch: ∼**12 seconds**.
- **Analysis:** This dropout mechanism promotes generalization because the test loss and accuracy values are better than the baseline model. The dropout model performed slightly better in unseen data, validating the importance of regularization in CNNs.

5.6.4 Experiment 4, 5, 6: Optimizer Comparison

1. **SGD:**
   - Learning rate: **0.01**.
   - Final test accuracy: **98.91%**.
   - Final test loss: **0.0417**.
   - Training time per epoch: ∼**23 seconds**.
   - Observations: Slower convergence but stable results.
2. **RMSprop:**
   - Learning rate: **0.001**.
   - Final test accuracy: **99.29%**.
   - Final test loss: **0.0691**.
   - Training time per epoch: ∼**24 seconds**.
   - Observations: Fast convergence, lower loss.
3. **SGD with Dropout:**
   - Learning rate: **0.001**.
   - Final test accuracy: **98.39%**.
   - Final test loss: **0.0593**.
   - Training time per epoch: ∼**11 seconds**.
   - Observations: The addition of dropout layers effectively reduced overfitting, leading to high generalization on the test set.

5.6.5 Experiment 5: Weak Model

- **Objective:** Test the performance of a simplified CNN with fewer filters and no pooling layers.
- **Model Description:** Single convolutional layer with 4 filters, Dense layer with 10 neurons.
- **Results:**
  - Final test accuracy: **96.08%**.
  - Final test loss: **0.1608**.
  - Training time per epoch: ∼**5 seconds**.
- **Analysis:** The weak model showed decent performance; however, it was not complex enough to extract very rich features. It stressed the importance of depth in architecture.

5.6.6 Performance Summary

Model	Accuracy (%)	Test Loss	Notes
Baseline Model	98.94	0.0616	High accuracy with minimal architectural complexity.
Models with Different Filters	99.00	0.0559	Improved feature extraction, but computational cost increased.
Models with Dropout	99.41	0.0242	Improved generalization and reduced overfitting.
Optimizers (SGD)	98.90	0.0417	Slower convergence but stable results.
Optimizers (RMSprop)	99.29	0.0691	Fast convergence with lower loss.
SGD Optimizer with Dropout	98.39	0.0593	Effectively reduced overfitting, leading to high generalization on the test set.
Weak Model	96.08	0.1608	Minimal complexity, but limited ability to capture rich features.

Table 3: Performance Summary for Different CNN Models

5.7 Discussion

5.7.1 Findings

- **The best performing configuration:**: In the experiments, the best-performing configuration was realized by the model implementing dropout layers at two stages. The first dropout period is followed immediately after pooling layers by another dropout period before the dense layer, both at dropout rates of 0.25 and 0.5 respectively. When combined with the Adam optimizer, this model obtained the best test accuracy of 98.94%. This achieved a balance between the complexity of architecture, regularization, and efficiency of the optimizer showcasing the role of dropout in boosting generalization performance. By randomly deactivating neurons during training, dropout ensures that the model can learn distributed and generalized representations of features, thereby minimizing overfitting as well as improving performance on previously unseen data.

- **Impact of Regularization and Dropout:** Regularization, especially with dropout, turned out to be the most important factor in improving the performance of any model. It improved test accuracy with a minimum compromise on training stability, especially for deep architectures. Dropout made the training process somewhat random so that the network could not simply rely on neurons while learning but had to learn more general and robust patterns from the data. This contributed significantly to achieving very high accuracy despite overfitting in large-parameter models.

- **Role of Optimizers:** The optimizer affected performance severely. Out of the optimizers, Adam performed best because it had adaptive learning rate adjustments and thus could converge faster and more stably during training. RMSprop was also good, but it could not catch up with Adam concerning both accuracy and loss. SGD, on the other hand, had to be fine-tuned to get just perfect hyperparameter values regarding the learning rate, and it converged rather slowly, making it less efficient and effective than Adam and RMSprop.

5.7.2 Challenges

1. **Hyperparameter Tuning:** The fine-tuning of learning rates, dropout probabilities, and filters took several iterations and computing resources to arrive at their best combination of such parameters. The trade-off accuracy-overfitting wasn't always simple to straighten out.

2. **Computational Cost:**: Increasing the number of filters and layers significantly increased training time per epoch (e.g., up to 41 seconds for models with 128 filters), making it challenging to experiment with highly complex architectures.

3. **Model Simplicity vs. Performance:** While the weak model with minimal complexity achieved reasonable accuracy (96.08%), its limited capacity highlighted the need for a deeper architecture to capture rich features effectively.

5.8 Conclusion

In this work, we have focused on exploring and evaluating the performance of Convolutional Neural Networks (CNNs) on the MNIST dataset. The systematic experiment was conducted specifically to analyze the placement of such structures, regularization, and optimization algorithms that yield configurations with accuracy and computational efficiency well balanced.

6 Appendix

6.1 Adaboost with ID3 as the Base Learner:

```python
# Step 1: Load and Pre-process the Data
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
# Load the dataset
columns = [
    'letter', 'x-box', 'y-box', 'width', 'high', 'onpix', 'x-bar',
    'y-bar', 'x2bar', 'y2bar', 'xybar', 'x2ybr', 'xy2br',
    'x-ege', 'xegvy', 'y-ege', 'yegvx'
]
data = pd.read_csv('letter-recognition.data', header=None, names=columns)
# Display the first few rows of the dataset to verify its structure
data.head()
# Map letters to numeric classes for easier processing
data['letter'] = data['letter'].apply(lambda x: ord(x) - ord('A'))
# Separate features and target variable
X = data.iloc[:, 1:]
y = data['letter']
# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42, stratify=y)
X_train.shape, X_test.shape, y_train.shape, y_test.shape

#Step 2: Implement the ID3 Decision Tree Algorithm

class ID3DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def _entropy(self, y):
        unique, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return -np.sum(probabilities * np.log2(probabilities + 1e-9))

    def _information_gain(self, X_column, y, threshold):
        parent_entropy = self._entropy(y)
        left_idx = X_column <= threshold
        right_idx = X_column > threshold
        if len(y[left_idx]) == 0 or len(y[right_idx]) == 0:
            return 0
        n = len(y)
        n_left, n_right = len(y[left_idx]), len(y[right_idx])
        e_left, e_right = self._entropy(y[left_idx]),
            self._entropy(y[right_idx])
        child_entropy = (n_left / n) * e_left + (n_right / n) * e_right
        return parent_entropy - child_entropy

    def _best_split(self, X, y):
        best_gain = -1
        best_feature = None
        best_threshold = None
        for feature_index in range(X.shape[1]):
            X_column = X[:, feature_index]
            thresholds = np.unique(X_column)
            for threshold in thresholds:
                gain = self._information_gain(X_column, y, threshold)
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature_index
                    best_threshold = threshold
        return best_feature, best_threshold

    def _build_tree(self, X, y, depth):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))
        if depth == self.max_depth or n_labels == 1 or n_samples == 0:
            return np.bincount(y).argmax()
        feature_index, threshold = self._best_split(X, y)
        if feature_index is None:
            return np.bincount(y).argmax()
        left_idxs = X[:, feature_index] <= threshold
        right_idxs = X[:, feature_index] > threshold
        left_subtree = self._build_tree(X[left_idxs], y[left_idxs], depth +
            1)
        right_subtree = self._build_tree(X[right_idxs], y[right_idxs],
            depth + 1)
        return (feature_index, threshold, left_subtree, right_subtree)

    def fit(self, X, y):
        self.tree = self._build_tree(X, y, 0)

    def _traverse_tree(self, x, tree):
        if not isinstance(tree, tuple):
            return tree
```

```python
        feature_index, threshold, left, right = tree
        if x[feature_index] <= threshold:
            return self._traverse_tree(x, left)
        return self._traverse_tree(x, right)

    def predict(self, X):
        return np.array([self._traverse_tree(x, self.tree) for x in X])
#Step 3: Implement the Adaboost Algorithm
class MultiClassAdaBoost:
        def __init__(self, base_learner_class, n_estimators=50):
            self.base_learner_class = base_learner_class
            self.n_estimators = n_estimators
            self.models = []
            self.alphas = []
            self.classes = None

        def fit(self, X, y):
            n_samples = X.shape[0]
            self.classes = np.unique(y)
            n_classes = len(self.classes)
            weights = np.ones(n_samples) / n_samples
            for _ in range(self.n_estimators):
                model = self.base_learner_class(max_depth=5)
                model.fit(X, y)
                y_pred = model.predict(X)
                incorrect = (y_pred != y).astype(int)
                error = np.dot(weights, incorrect) / np.sum(weights)
                if error >= 0.5 or error == 0:
                    break
                alpha = 0.5 * np.log((1 - error) / error)
                weights = weights * np.exp(alpha * incorrect)
                weights /= np.sum(weights)
                self.models.append(model)
                self.alphas.append(alpha)

        def predict(self, X):
            class_votes = np.zeros((X.shape[0], len(self.classes)))
            for alpha, model in zip(self.alphas, self.models):
                predictions = model.predict(X)
                for i, pred in enumerate(predictions):
                    class_index = np.where(self.classes == pred)[0][0]
                    class_votes[i, class_index] += alpha
            return self.classes[np.argmax(class_votes, axis=1)]

```

```
#Step 4: Train and Evaluate the Model

# Train MultiClass AdaBoost on a subset of data for demonstration
multiclass_adaboost =
    MultiClassAdaBoost(base_learner_class=ID3DecisionTree, n_estimators=10)
multiclass_adaboost.fit(X_train.values[:6500], y_train.values[:6500])

# Predict on a small subset of test data
multiclass_predictions = multiclass_adaboost.predict(X_test.values)

# Display predictions
multiclass_predictions

from sklearn.metrics import accuracy_score, classification_report

accuracy = accuracy_score(y_test, multiclass_predictions)
print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:\n")
print(classification_report(y_test, multiclass_predictions))
```

Listing 1: Python Code of Task 1

6.2 Breast Cancer Diagnosis Using Artificial Neural Networks:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
# Load the dataset
file_path = 'wdbc.data'  # Ensure the dataset is in the same directory as
    this notebook
columns = [
    'ID', 'Diagnosis',
    'Mean Radius', 'Mean Texture', 'Mean Perimeter', 'Mean Area', 'Mean
        Smoothness',
    'Mean Compactness', 'Mean Concavity', 'Mean Concave Points', 'Mean
        Symmetry',
    'Mean Fractal Dimension',
    'Radius SE', 'Texture SE', 'Perimeter SE', 'Area SE', 'Smoothness SE',
    'Compactness SE', 'Concavity SE', 'Concave Points SE', 'Symmetry SE',
    'Fractal Dimension SE',
    'Worst Radius', 'Worst Texture', 'Worst Perimeter', 'Worst Area',
        'Worst Smoothness',
    'Worst Compactness', 'Worst Concavity', 'Worst Concave Points', 'Worst
        Symmetry',
    'Worst Fractal Dimension'
]
data = pd.read_csv(file_path, header=None, names=columns)
# Drop the ID column
data.drop('ID', axis=1, inplace=True)
# Encode the target variable
data['Diagnosis'] = data['Diagnosis'].map({'M': 1, 'B': 0})
# Normalize the features
X = data.drop('Diagnosis', axis=1).values
y = data['Diagnosis'].values
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
# Neural Network Implementation from Scratch
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size,
        learning_rate=0.01):
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))
        self.learning_rate = learning_rate
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))
    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))
    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = self.sigmoid(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = self.sigmoid(self.Z2)
        return self.A2
    def backward(self, X, y, output):
        m = X.shape[0]
        dZ2 = output - y.reshape(-1, 1)
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m
        dZ1 = np.dot(dZ2, self.W2.T) * self.sigmoid_derivative(self.Z1)
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
    def train(self, X, y, epochs=1000):
        losses = []
        for epoch in range(epochs):
            output = self.forward(X)
            loss = -np.mean(y * np.log(output) + (1 - y) * np.log(1 -
                output))
            losses.append(loss)
            self.backward(X, y, output)
        return losses
    def predict(self, X):
        output = self.forward(X)
        return (output > 0.5).astype(int)
# Initialize and train the network
input_size = X_train.shape[1]
hidden_size = 16  # Number of neurons in the hidden layer
output_size = 1  # Binary classification
learning_rate = 0.01
nn = NeuralNetwork(input_size, hidden_size, output_size, learning_rate)
epochs = 2000
losses = nn.train(X_train, y_train, epochs)
# Predict on the test set
y_pred = nn.predict(X_test)
accuracy = np.mean(y_pred.flatten() == y_test)
print(f"Test Set Accuracy: {accuracy * 100:.2f}%")
# Plot the training loss
plt.figure(figsize=(8, 6))
plt.plot(range(epochs), losses, label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 2: Python Code of Task 2

6.3 Naive Bayes Classifier for Car Evaluation Dataset:

```

import pandas as pd
# Load the dataset
file_path = 'car.data'
columns = ['buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety',
    'class']
car_data = pd.read_csv(file_path, header=None, names=columns)
car_data.head()

# Converting categorical data to numerical
car_data_encoded = car_data.copy()
for column in car_data_encoded.columns:
    car_data_encoded[column] =
        car_data_encoded[column].astype('category').cat.codes

# Splitting the dataset into features (X) and target (y)
X = car_data_encoded.drop('class', axis=1)
y = car_data_encoded['class']

# Splitting data into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

import numpy as np

class NaiveBayesClassifier:
    def __init__(self):
        self.priors = {}
        self.likelihoods = {}
        self.classes = None

    def fit(self, X, y):
        self.classes = np.unique(y)
        for cls in self.classes:
            X_cls = X[y == cls]
            self.priors[cls] = len(X_cls) / len(y)
            self.likelihoods[cls] = {}
            for column in X.columns:
                self.likelihoods[cls][column] =
                    X_cls[column].value_counts(normalize=True).to_dict()

    def predict(self, X):
        predictions = []
        for _, row in X.iterrows():
            class_probs = {}
            for cls in self.classes:
                prior = self.priors[cls]
                likelihood =
                    np.prod([self.likelihoods[cls][column].get(row[column],
                    1e-6) for column in X.columns])
                class_probs[cls] = prior * likelihood
            predictions.append(max(class_probs, key=class_probs.get))
        return np.array(predictions)

    def accuracy(self, y_true, y_pred):
        return np.mean(y_true == y_pred)
# Training the model
nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_train, y_train)

# Predicting on the test set
y_pred = nb_classifier.predict(X_test)

# Calculating accuracy
accuracy = nb_classifier.accuracy(y_test, y_pred)
print(f"Accuracy of the Naive Bayes Classifier: {accuracy * 100:.2f}")

import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.bar(['Naive Bayes'], [accuracy], color='blue')
plt.ylim(0, 1)
plt.ylabel('Accuracy')
plt.title('Accuracy of Naive Bayes Classifier')
plt.show()
```

Listing 3: Python Code of Task 3

6.4 MNIST Handwritten Digit Classification Using Artificial Neural Networks:

```python

# Import necessary libraries
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import struct

# Paths to dataset files (ensure these files are in the same directory)
train_images_path = 'train-images.idx3-ubyte'
train_labels_path = 'train-labels.idx1-ubyte'
test_images_path = 't10k-images.idx3-ubyte'
test_labels_path = 't10k-labels.idx1-ubyte'

# Function to load images
def load_images(file_path):
    with open(file_path, 'rb') as f:
        magic, num, rows, cols = struct.unpack(">IIII", f.read(16))
        data = np.frombuffer(f.read(), dtype=np.uint8).reshape(num, rows,
            cols)
    return data

# Function to load labels
def load_labels(file_path):
    with open(file_path, 'rb') as f:
        magic, num = struct.unpack(">II", f.read(8))
        labels = np.frombuffer(f.read(), dtype=np.uint8)
    return labels

# Load data
train_images = load_images(train_images_path)
train_labels = load_labels(train_labels_path)
test_images = load_images(test_images_path)
test_labels = load_labels(test_labels_path)

# Normalize images to [0, 1]
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# Flatten images for fully connected layers
```

```python
train_images = train_images.reshape(train_images.shape[0], -1)
test_images = test_images.reshape(test_images.shape[0], -1)

# One-hot encode labels
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)

# Define the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(train_images.shape[1],)),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_data=(test_images, test_labels))
# Evaluate model on the test set
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

# Plot accuracy and loss over epochs
plt.figure(figsize=(12, 5))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

Listing 4: Python Code of Task 4

6.5 Experiments with Convolutional Neural Networks on the MNIST Dataset:

```
import gzip
import struct
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models

# Helper function to load MNIST images
def load_mnist_images(file_path):
    with open(file_path, 'rb') as f:
        _, num_images, rows, cols = struct.unpack('>IIII', f.read(16))
        data = np.fromfile(f, dtype=np.uint8).reshape(num_images, rows,
            cols)
    return data / 255.0

# Helper function to load MNIST labels
def load_mnist_labels(file_path):
    with open(file_path, 'rb') as f:
        _, num_labels = struct.unpack('>II', f.read(8))
        labels = np.fromfile(f, dtype=np.uint8)
    return labels

# File paths (update these to your local paths)
train_images_path = 'train-images.idx3-ubyte'
train_labels_path = 'train-labels.idx1-ubyte'
test_images_path = 't10k-images.idx3-ubyte'
test_labels_path = 't10k-labels.idx1-ubyte'

# Load datasets
train_images = load_mnist_images(train_images_path)
```

```python
train_labels = load_mnist_labels(train_labels_path)
test_images = load_mnist_images(test_images_path)
test_labels = load_mnist_labels(test_labels_path)

# Reshape images to add channel dimension
train_images = train_images[..., np.newaxis]
test_images = test_images[..., np.newaxis]

print(train_images.shape, train_labels.shape, test_images.shape,
    test_labels.shape)

# Define a CNN model with more filters
def create_cnn_model_more_filters():
    model = models.Sequential([
        layers.Conv2D(64, (3, 3), activation='relu', input_shape=(28, 28,
            1)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    return model

# Compile and train the model
model = create_cnn_model_more_filters()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with more filters: {test_acc}')

# Define a CNN model with more filters
def create_cnn_model_extreme_filters():
    model = models.Sequential([
        layers.Conv2D(512, (3, 3), activation='relu', input_shape=(28, 28,
            1)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(512, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(512, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(15, activation='relu'),
        layers.Dense(15, activation='softmax')
    ])
    return model

# Compile and train the model
model = create_cnn_model_extreme_filters()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with more filters: {test_acc}')

# Define a CNN model with dropout layers
def create_cnn_model_with_dropout():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
            1)),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

# Compile and train the model
model = create_cnn_model_with_dropout()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```python
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with dropout layers: {test_acc}')

# Define a CNN model with SGD optimizer
model = create_cnn_model_more_filters()
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with SGD optimizer: {test_acc}')

# Define a CNN model with RMSprop optimizer
model = create_cnn_model_more_filters()
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with SGD optimizer: {test_acc}')

# Define a CNN model with SGD optimizer
model = create_cnn_model_with_dropout()
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10,
    validation_data=(test_images, test_labels))

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy with SGD optimizer: {test_acc}')

# Define a weak CNN model
```

```python
def create_very_weak_cnn_model():
    model = models.Sequential([
        layers.Conv2D(4, (3, 3), activation='relu', input_shape=(28, 28,
            1)),  # Only 4 filters
        layers.Flatten(),  # Directly flatten without pooling
        layers.Dense(10, activation='relu'),  # Only 10 neurons in dense
            layer
        layers.Dense(10, activation='softmax')  # Output layer
    ])
    return model

# Compile the weak model
model = create_very_weak_cnn_model()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the weak model for few epochs
history = model.fit(train_images, train_labels, epochs=3,
    validation_data=(test_images, test_labels))

# Evaluate the weak model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy of weak model: {test_acc}')
```

Listing 5: Python Code of Task 5