ARTIFICIAL NEURAL NETWORKS

A PROJECT REPORT
Submitted By

Ankit Sharma
24/AFI/05

Submitted in partial fulfillment of the requirements for the degree of
Masters of Technology in Artificial Intelligence

to

Dr. Anurag Goel
Department of Computer Science & Engineering

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi - 110042

Oct 5, 2024

Certificate

This is to certify that project report entitled **"ARTIFICIAL NEURAL NETWORKS"** submitted by **Ankit Sharma (24/AFI/05)** for partial fulfillment of the requiremnt for the award of degree Master of Technology (Artificial Intelligence) is a record of the candidate work carried out by him.

Dr. Anurag Goel
Department of Computer Science & Engineering
Delhi Technological University

Declaration

We hereby declare that the work presented in this report entitled **"ARTIFICIAL NEURAL NETWORKS"**, was carried out by me. We have not submitted the matter embodied in this report for the award of any other degree or diploma of any other University or Institute. I have given due credit to the original sources for all the words, ideas, diagrams, graphics, computer programs, experiments, results, that are not my original contribution. I have used quotation marks to identify verbatim sentences and given credit to the original sources.

Ankit Sharma
24/AFI/05

Acknowledgements

First of all I would like to thank the Almighty, who has always guided me to work on the right path of my life. My greatest thanks are to my parents who best owed ability and strength in me to complete this work.

I owe profound gratitude to **Dr.Anurag Goel** who has been constant source of inspiration to me throughout the period of this project. It was his competent guidance, constant encouragement and critical evaluation that helped me to develop a new insight my project. His calm, collected and professionally impeccable style of handling situations not only steered me through every problem, but also helped me to grow as a matured person.

I am also thankful to him for trusting my capabilities to develop this project under his guidance.

Abstract

This project report investigates the processes of forward and backward propagation in artificial neural networks (ANNs) for both single and multiple hidden layers, with a focus on the use of activation functions such as Sigmoid and ReLU (Rectified Linear Unit). The forward propagation mechanism is analyzed as the means of computing the output from the input through the weighted summation of inputs and activation functions, while backward propagation is explored as the technique for error minimization through gradient descent and weight updates. A comparative study is conducted to evaluate the performance of different activation functions in various network architectures. The implementation utilizes Python's Scikit-learn (sklearn) library to streamline the development and training of ANNs, and its pre-built functions are employed to enhance the efficiency of training processes.

Experimental results highlight the impact of activation functions and network depth on convergence speed and model accuracy, offering insights into the effectiveness of Sigmoid and ReLU in both shallow and deep networks. The report concludes by discussing potential trade-offs between computational complexity and performance in ANN training.

Contents

Chapter 1

Introduction

1.1 Background

Artificial Neural Networks (ANNs) have revolutionized the field of machine learning and artificial intelligence, enabling the development of systems that can learn and generalize from data. The fundamental mechanics behind the learning process in ANNs rely on two essential concepts: forward propagation and backward propagation. These two processes allow neural networks to compute predictions and update their internal parameters based on the error generated by those predictions, respectively. This report explores the mathematical and computational processes behind forward and backward propagation in ANN models, with a focus on single and multiple hidden layers using two of the most common activation functions: Sigmoid, ReLU and Softmax. Furthermore, the Scikit-learn (sklearn) library is used to implement and evaluate the performance of these networks.

The effectiveness of ANNs lies in their ability to automatically learn features from data during training. Forward propagation, which is responsible for generating predictions, works by passing the input through successive layers of the network, where each neuron performs a weighted sum of inputs followed by the application of an activation function. The choice of activation function, such as Sigmoid or ReLU, greatly impacts the network's learning ability.

Backward propagation, or backpropagation, is the learning process by which the model adjusts its weights to minimize the loss of its predictions. It relies on the chain rule of calculus to compute the gradient of the loss function with respect to each weight, which is then updated to reduce the error iteratively. For networks with deeper architectures (multiple hidden layers), this process becomes more complex but enables the learning of more abstract and high-level features.

1.2 Problem Statement

The design of ANNs involves various decisions, including the number of hidden layers, the choice of activation functions, and the optimization algorithm for updating weights. These decisions affect the network's learning efficiency, convergence speed, and overall performance on the given tasks. Understanding how forward and backward propagation function in different network architectures is crucial for improving performance and computational efficiency.

1.3 Methodology

The approach taken in this project involves both theoretical and practical components. First, the fundamental concepts of forward and backward propagation are presented and explained mathematically. The role of activation functions in transforming neuron outputs and their gradients during backpropagation is explored, focusing on the differences between Sigmoid and ReLU.

The practical component consists of implementing ANN models using Scikit-learn. The steps include:

- **Model Design:** Single-layer and multi-layer neural networks are designed with varying numbers of hidden layers and using Sigmoid, ReLU and Softmax activation functions.

- **Forward Propagation:** Input data is passed through the network to compute the output.
- **Backward Propagation:** The error is propagated backward through the network to update weights.
- **Training and Evaluation:** The models are trained on sample datasets, and their performance is evaluated based on accuracy, convergence speed, and computational cost.
- **Comparison and Analysis:** The effect of using different activation functions and network depths is analyzed and compared.

Chapter 2

Proposed Methodology

2.1 MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is one of the most widely used benchmarks for testing and evaluating machine learning algorithms, particularly in the field of neural networks and deep learning. It consists of 70,000 grayscale images of handwritten digits from 0 to 9, split into 60,000 training images and 10,000 test images. Each image is a 28x28 pixel grid, where each pixel represents a value between 0 and 255 corresponding to the intensity of the pixel in grayscale.

The task is to classify these images into their corresponding digit labels, making it a multi-class classification problem with 10 possible classes (0-9). Due to its simplicity and standardized nature, the MNIST dataset provides an ideal ground for testing the efficiency of forward and backward propagation in different neural network architectures.

2.2 Forward Propagation on the MNIST Dataset

Forward propagation refers to the process where inputs (in this case, images of handwritten digits) are passed through the neural network to generate predictions. For the MNIST dataset, this process involves the following steps:

1. **Input Layer:** Each image is represented as a vector of 784 values

(28x28 pixels flattened into a one-dimensional array). These values are fed as input to the network.

2. **Hidden Layers:** Depending on the architecture, the network can have one or more hidden layers. Each neuron in these layers receives inputs from the previous layer, performs a weighted sum of these inputs, adds a bias, and passes the result through an activation function. The choice of activation function plays a crucial role in determining the behavior of the network. In this project, we focus on two activation functions:

   - **Sigmoid Activation Function:** It maps the input to a range between 0 and 1, making it useful for probability-based outputs.
   - **ReLU (Rectified Linear Unit):** It outputs the input directly if it is positive and outputs zero otherwise. This activation function is computationally efficient and helps avoid the vanishing gradient problem often associated with Sigmoid.

3. **Output Layer:** The output layer consists of 10 neurons (one for each digit from 0 to 9). The values of these neurons represent the predicted probabilities for each class. The softmax function is typically applied in the output layer to convert the raw output into a probability distribution, where the neuron with the highest probability indicates the predicted digit.

2.3 Backward Propagation on the MNIST Dataset

Backward propagation, or backpropagation, is the process of learning in a neural network. Once forward propagation generates predictions, the model calculates the error by comparing the predicted output with the true labels from the dataset. This error is then propagated backward through the network to update the weights and biases in a way that reduces the overall error. The steps involved in backpropagation are:

1. **Loss Function:** The error is quantified using a loss function, such as cross-entropy for classification problems. Cross-entropy loss is appropriate for multi-class classification like MNIST, as it penalizes incorrect predictions more heavily.

2. **Gradient Calculation:** Using the chain rule of calculus, the gradient of the loss function with respect to each weight and bias is computed. This step helps in understanding how much the weights contributed to the error, and how they should be adjusted to reduce that error. For each layer, the gradient of the loss with respect to the output is calculated.

3. **Weight Updates:** Once the gradients are calculated, the weights and biases are updated using gradient descent.

2.4 Overfitting and Underfitting

1. Underfitting: Both training and validation loss are high, and the model performs poorly on both training and validation data. The model is too simple to capture the underlying patterns in the data. o Countermeasures: Increase model complexity (e.g., add more layers or neurons), train for more epochs, or reduce regularization.
2. Overfitting: Training loss is low, but validation loss is high. The model performs well on training data but poorly on validation data. The model is too complex and captures noise in the training data. Use regularization techniques (e.g., dropout, L2 regularization), reduce model complexity, or use more training data.
3. Well-trained model (Generalization): Both training and validation loss are low and similar. The model performs well on both training and validation data. The model has learned the underlying patterns in the data without overfitting. Ensure proper data preprocessing, use cross-validation, and monitor training to avoid overfitting.

Underfitting

- Training Loss: High
- Validation Loss: High

Overfitting

- Training Loss: Low
- Validation Loss: High

Well-trained Model

- Training Loss: Low
- Validation Loss: Low and similar to training loss

2.5 Hidden layers in Neural Network

A neural network with more hidden layers does not necessarily always have better accuracy compared to one with fewer layers. While adding more hidden layers can increase the capacity of the network to model complex patterns, there are several factors that can influence whether it will actually improve performance:

1. **Overfitting:** More hidden layers increase the network's capacity to learn, but this can also lead to overfitting, where the model becomes too specialized to the training data and fails to generalize well to new, unseen data.

2. **Diminishing Returns:** At some point, adding more layers might not lead to significant improvements in accuracy and can even cause diminishing returns. The added complexity might not translate into better model performance for certain tasks.

3. **Vanishing/Exploding Gradient Problem:** As more layers are added, training becomes harder due to the vanishing gradient (where

gradients become very small) or exploding gradient (where gradients become very large), making it difficult for the model to converge. Advanced techniques like batch normalization, skip connections, and more sophisticated optimizers can help, but not always.

4. **Data Quality and Quantity:** The size and quality of the training data are crucial. If the data is insufficient or noisy, deeper networks might not perform better and can even worsen accuracy.

Chapter 3

Code Implementation

The implementation of the forward and backward propagation for the MNIST dataset, along with the corresponding experiments and evaluations, can be accessed through the Google Colab notebook linked below. This notebook provides a detailed step-by-step demonstration of the neural network models discussed in the report, including model training and the results obtained from applying different activation functions.

Google Colab Link: Click Here!

3.1 Forward & Backward Propagation using Sigmoid and Softmax

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid
from scipy.special import softmax
import h5py
from sklearn.model_selection import train_test_split

# Function to load MNIST data from an HDF5 file
def load_mnist(filepath="/content/MNIST_Subset.h5"):
    with h5py.File(filepath, 'r') as f:
        X = f['X'][:]
        Y = f['Y'][:]
    return X, Y

# Function to initialize the network's weights and biases
```

```
def initialize_network(layer_sizes):
    np.random.seed(10)
    weights = []
    biases = []
    for i in range(len(layer_sizes) - 1):
        # Initialize weights with small random values
        weights.append(np.random.randn(layer_sizes[i], layer_sizes[i +
            1]) * 0.1)
        # Initialize biases with zeros
        biases.append(np.zeros((1, layer_sizes[i + 1])))
    return weights, biases

# Function for forward propagation through the network
def forward_propagation(X, weights, biases):
    X = X.reshape(X.shape[0], -1)  # Flatten the input
    activations = [X]
    Zs = []
    for W, b in zip(weights[:-1], biases[:-1]):
        Z = np.dot(activations[-1], W) + b  # Linear step
        A = sigmoid(Z)  # Activation step
        Zs.append(Z)
        activations.append(A)
    Z = np.dot(activations[-1], weights[-1]) + biases[-1]
    A = softmax(Z, axis=1)  # Softmax for the output layer
    Zs.append(Z)
    activations.append(A)
    return activations, Zs

# Function for backward propagation through the network
def backward_propagation(y, activations, Zs, weights, l2_lambda=0.01):
    m = y.shape[0]
    y_one_hot = np.eye(10)[y]  # One-hot encode the labels
    dZ = activations[-1] - y_one_hot  # Derivative of the loss with
       respect to Z
    dWs = []
    dbs = []
    for i in reversed(range(len(weights))):
        dW = np.dot(activations[i].T, dZ) / m + l2_lambda * weights[i]
           / m  # Gradient of weights
        db = np.sum(dZ, axis=0, keepdims=True) / m  # Gradient of
           biases
        dWs.insert(0, dW)
        dbs.insert(0, db)
        if i > 0:
            dA = np.dot(dZ, weights[i].T)
            dZ = dA * activations[i] * (1 - activations[i])  #
               Derivative of the activation function
```

```
    return dWs, dbs

# Function to update the network's weights and biases
def update_weights(weights, biases, dWs, dbs, learning_rate):
    for i in range(len(weights)):
        weights[i] -= learning_rate * dWs[i]  # Update weights
        biases[i] -= learning_rate * dbs[i]  # Update biases
    return weights, biases

# Function to train the network with validation
def train_network_with_validation(X_train, y_train, X_val, y_val,
   layer_sizes, learning_rate=0.1, epochs=500, l2_lambda=0.01):
    weights, biases = initialize_network(layer_sizes)
    train_accuracies = []
    val_accuracies = []

    for epoch in range(epochs):
        activations, Zs = forward_propagation(X_train, weights, biases)
             # Forward pass
        dWs, dbs = backward_propagation(y_train, activations, Zs,
           weights, l2_lambda)  # Backward pass
        weights, biases = update_weights(weights, biases, dWs, dbs,
           learning_rate)  # Update weights and biases

        train_predictions = np.argmax(activations[-1], axis=1)  #
           Predictions for training data
        train_accuracy = np.mean(train_predictions == y_train)  #
           Training accuracy
        train_accuracies.append(train_accuracy)

        val_activations, _ = forward_propagation(X_val, weights, biases
           )  # Forward pass for validation data
        val_predictions = np.argmax(val_activations[-1], axis=1)  #
           Predictions for validation data
        val_accuracy = np.mean(val_predictions == y_val)  # Validation
           accuracy
        val_accuracies.append(val_accuracy)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Training Accuracy: {train_accuracy},
                Validation Accuracy: {val_accuracy}")

        # Learning rate decay
        if epoch % 100 == 0 and epoch != 0:
            learning_rate *= 0.9

    return weights, biases, train_accuracies, val_accuracies
```

```
# Function to plot training and validation accuracies over epochs
def plot_accuracies(train_accuracies, val_accuracies):
    plt.plot(train_accuracies, label='Training Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracy over Epochs')
    plt.show()

# Load data from the specified file path
file_path = '/content/MNIST_Subset.h5'
X, Y = load_mnist(file_path)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, Y, test_size=0.5,
    random_state=10)

# Define layer sizes for different configurations
layer_sizes_1 = [784, 100, 10]
layer_sizes_2 = [784, 100, 50, 50, 10]

# Train the network with 1 hidden layer and track accuracies
print(f"Accuracy for 1 hidden layer:")
weights_1, biases_1, train_accuracies_1, val_accuracies_1 =
    train_network_with_validation(X_train, y_train, X_val, y_val,
    layer_sizes_1)

# Train the network with 3 hidden layers and track accuracies
print(f"Accuracy for 3 hidden layers:")
weights_2, biases_2, train_accuracies_2, val_accuracies_2 =
    train_network_with_validation(X_train, y_train, X_val, y_val,
    layer_sizes_2)

# Plot the accuracies for both configurations
plot_accuracies(train_accuracies_1, val_accuracies_1)
plot_accuracies(train_accuracies_2, val_accuracies_2)

import pickle
#saving weights of the model
def save_model(weights, biases, filename):
    with open(filename, 'wb') as f:
        pickle.dump((weights, biases), f)

save_model(weights_1, biases_1, 'model_1_hidden_layer.pkl')
save_model(weights_2, biases_2, 'model_3_hidden_layers.pkl')
```

3.1.1 Output

Accuracy for 1 hidden layer:

- Epoch 0, Training Accuracy: 0.0008421052631578948, Validation Accuracy: 0.5722705585181027
- Epoch 10, Training Accuracy: 0.8662456140350877, Validation Accuracy: 0.8718776312096548
- Epoch 20, Training Accuracy: 0.9153684210526316, Validation Accuracy: 0.9068200954252035
- Epoch 30, Training Accuracy: 0.9285614035087719, Validation Accuracy: 0.9174852652259332
- Epoch 40, Training Accuracy: 0.9344561403508772, Validation Accuracy: 0.9238001683974179
- Epoch 50, Training Accuracy: 0.9390877192982456, Validation Accuracy: 0.9287117597530171
- Epoch 60, Training Accuracy: 0.9431578947368421, Validation Accuracy: 0.9322200392927309
- Epoch 450, Training Accuracy: 0.9706666666666667, Validation Accuracy: 0.9487791187201796
- Epoch 460, Training Accuracy: 0.9710877192982457, Validation Accuracy: 0.9486387875385911
- Epoch 470, Training Accuracy: 0.9712280701754386, Validation Accuracy: 0.9489194499017681
- Epoch 480, Training Accuracy: 0.9712280701754386, Validation Accuracy: 0.9500420993544766
- Epoch 490, Training Accuracy: 0.9715087719298245, Validation Accuracy: 0.9497614369912994

Accuracy for 3 hidden layers:

- Epoch 0, Training Accuracy: 0.49487719298245614, Validation Accuracy: 0.4816166152119001
- Epoch 10, Training Accuracy: 0.4993684210526316, Validation Accuracy: 0.5140331181588549
- Epoch 20, Training Accuracy: 0.4997894736842105, Validation Accuracy: 0.5141734493404434
- Epoch 30, Training Accuracy: 0.504280701754386, Validation Accuracy: 0.5206286836935167
- Epoch 90, Training Accuracy: 0.6409824561403509, Validation Accuracy: 0.6538029750210497
- Epoch 110, Training Accuracy: 0.7030175438596491, Validation Accuracy: 0.710637103564412
- Epoch 120, Training Accuracy: 0.727859649122807, Validation Accuracy: 0.7332304238001685
- Epoch 170, Training Accuracy: 0.8134736842105263, Validation Accuracy: 0.8104125736738703
- Epoch 460, Training Accuracy: 0.9330526315789474, Validation Accuracy: 0.925624473758069
- Epoch 470, Training Accuracy: 0.9350175438596491, Validation Accuracy: 0.9267471232107775
- Epoch 480, Training Accuracy: 0.936, Validation Accuracy: 0.9270277855739545
- Epoch 490, Training Accuracy: 0.9376842105263158, Validation Accuracy: 0.9282907662082515

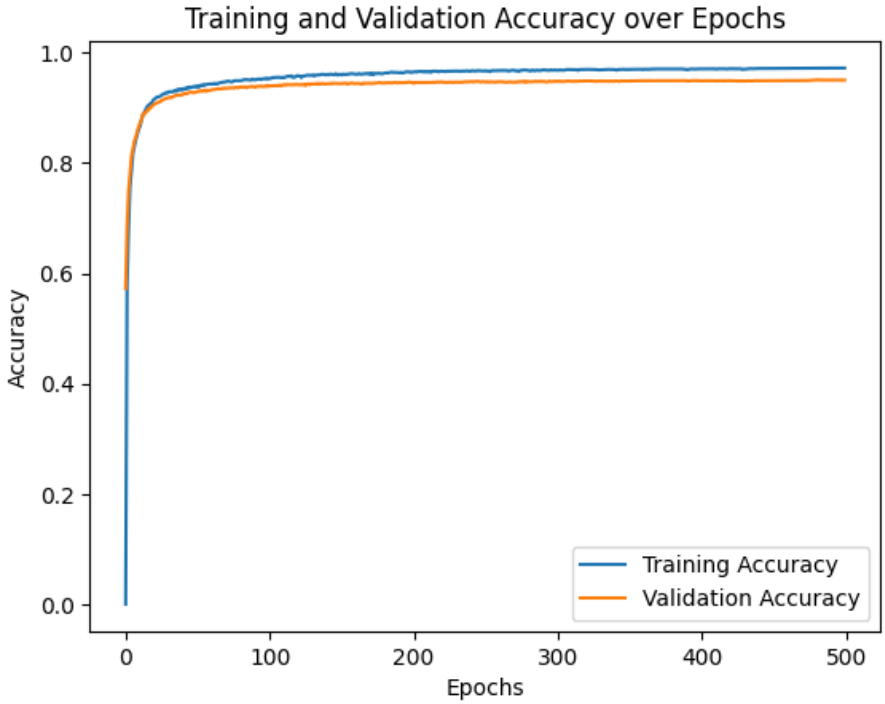

Figure 3.1: For 1 hidden layer for 500 iterations using Sigmoid

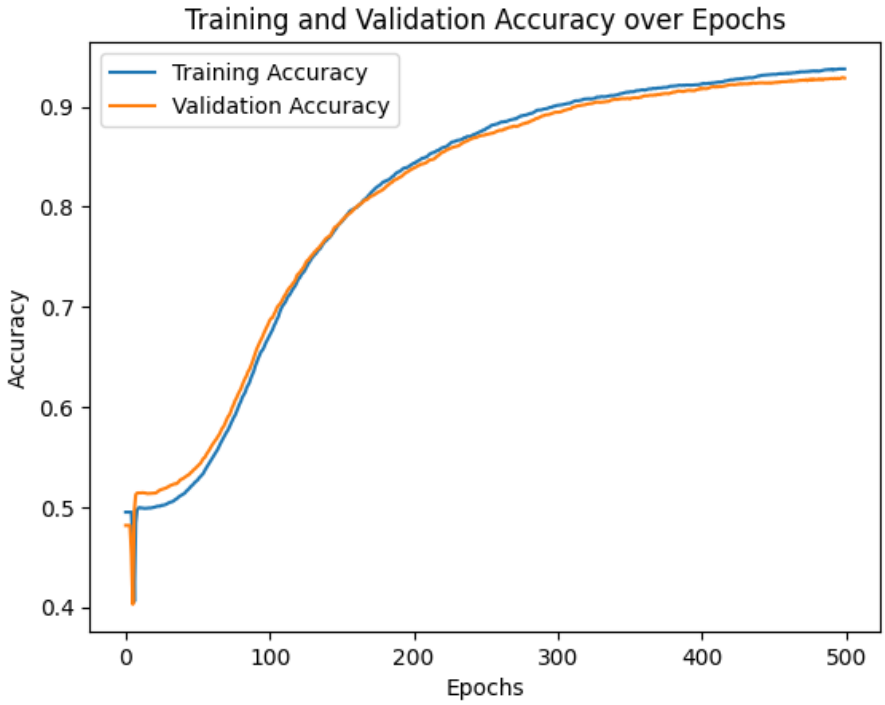

Figure 3.2: For 3 hidden layers for 500 iterations using Sigmoid

3.1.2 Overfitting and Underfitting for Sigmoid & Softmax

Analysis:

- Overfitting occurs when the model performs very well on the training set but poorly on the validation set, which typically results in a wide gap between the two accuracy curves.
- Underfitting happens when both the training and validation accuracy remain low, indicating that the model is too simple or the training process is not effective.

For 1 hidden layer for 500 iterations using Sigmoid

Observations:

- **Training Accuracy:** The blue line shows training accuracy, which starts low but quickly increases, reaching almost 100% as the number of epochs increases.
- **Validation Accuracy:** The orange line represents the validation accuracy. It closely follows the training accuracy and also approaches near 100

Conclusion:

- **Neither underfitting nor overfitting:** This plot does not show signs of underfitting or significant overfitting. The validation accuracy closely follows the training accuracy, which indicates that the model is not overfitting (no significant gap between the two curves).
- **Good generalization:** Since both curves converge and stay high, it suggests that the model is well-trained and generalizes well on unseen data. There is no major gap, meaning the model performs similarly on both training and validation sets, which is a sign of effective training.

Challenges and Countermeasures:

1. **Overfitting risk:** With prolonged training (500 epochs), there could have been a risk of overfitting. But as seen in the graph, this was successfully mitigated.
Countermeasure: Possible techniques like early stopping, dropout, or regularization may have been used to control overfitting. These methods help ensure the model doesn't memorize the training data but learns general features applicable to new data.

2. **Underfitting in the initial stages:** In the early epochs (before epoch 100), both accuracies rise sharply, but this is normal as the model quickly learns patterns from the data.
Countermeasure: Likely, using a well-structured neural network with sufficient capacity and tuning hyperparameters like learning rate allowed the model to train effectively without being too simplistic.

For 3 hidden layers for 500 iterations using Sigmoid

Observations:

- **Training Accuracy:** The training accuracy starts relatively low (0.5) and increases steadily over the epochs, reaching around 0.95 after 500 epochs. There is no drastic upward spike, and it shows a smooth, consistent improvement.
- **Validation Accuracy:** The validation accuracy follows the training accuracy very closely, with minimal gap between the two. It also rises steadily and reaches a similar point near 0.95, showing that the model generalizes well.

Conclusion:

- **No underfitting:** The model starts with low accuracy but improves as training progresses. If the accuracy had remained low throughout,

it would indicate underfitting, but that's not the case here.

- **No overfitting:** Overfitting would be characterized by a significant gap between training and validation accuracy, but both lines are nearly identical here, indicating that the model is not overfitting.
- **Well-trained and Generalization:** The close alignment between the training and validation accuracy curves shows that the model is well-trained and has learned patterns that generalize well to unseen data.

Challenges and Countermeasures:

1. **Slower early-stage learning:** The model starts with relatively low accuracy, indicating it took longer to pick up patterns. This could be due to a more complex model, noisy data, or suboptimal initialization.
Countermeasure: A common strategy to address this is to adjust the learning rate, use a better weight initialization method, or implement learning rate schedules to boost early learning.

2. **Preventing Overfitting at higher epochs:** With the model trained over 500 epochs, there was a risk of overfitting in the later stages.
Countermeasure: Regularization techniques like L2 weight decay, dropout, and data augmentation can help maintain balance. The absence of overfitting here suggests such measures may have been effective.

3.2 Forward & Backward Propagation using ReLU and Softmax

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import softmax
import h5py
from sklearn.model_selection import train_test_split
```

```
# Function to load MNIST data from an HDF5 file
def load_mnist(filepath="/content/MNIST_Subset.h5"):
    with h5py.File(filepath, 'r') as f:
        X = f['X'][:]
        Y = f['Y'][:]
    return X, Y

# Function to initialize the network's weights and biases
def initialize_network(layer_sizes):
    np.random.seed(10)
    weights = []
    biases = []
    for i in range(len(layer_sizes) - 1):
        # Initialize weights with He initialization
        weights.append(np.random.randn(layer_sizes[i], layer_sizes[i +
            1]) * np.sqrt(2. / layer_sizes[i]))
        # Initialize biases with zeros
        biases.append(np.zeros((1, layer_sizes[i + 1])))
    return weights, biases

# ReLU activation function
def relu(Z):
    return np.maximum(0, Z)

# Derivative of ReLU activation function
def relu_derivative(Z):
    return Z > 0

# Function for forward propagation through the network
def forward_propagation(X, weights, biases, dropout_rate=0.5):
    X = X.reshape(X.shape[0], -1)  # Flatten the input
    activations = [X]
    Zs = []
    for W, b in zip(weights[:-1], biases[:-1]):
        Z = np.dot(activations[-1], W) + b  # Linear step
        A = relu(Z)  # Activation step
        # Apply dropout
        D = np.random.rand(*A.shape) < dropout_rate
        A = A * D
        A = A / dropout_rate
        Zs.append(Z)
        activations.append(A)
    Z = np.dot(activations[-1], weights[-1]) + biases[-1]
    A = softmax(Z, axis=1)  # Softmax for the output layer
    Zs.append(Z)
    activations.append(A)
    return activations, Zs
```

```
# Function for backward propagation through the network
def backward_propagation(y, activations, Zs, weights, dropout_rate=0.5)
    :
    m = y.shape[0]
    y_one_hot = np.eye(10)[y]  # One-hot encode the labels
    dZ = activations[-1] - y_one_hot  # Derivative of the loss with
        respect to Z
    dWs = []
    dbs = []
    for i in reversed(range(len(weights))):
        dW = np.dot(activations[i].T, dZ) / m  # Gradient of weights
        db = np.sum(dZ, axis=0, keepdims=True) / m  # Gradient of
            biases
        dWs.insert(0, dW)
        dbs.insert(0, db)
        if i > 0:
            dA = np.dot(dZ, weights[i].T)
            dA = dA * (activations[i] > 0)  # ReLU derivative
            # Apply dropout
            dA = dA * (np.random.rand(*activations[i].shape) <
                dropout_rate)
            dA = dA / dropout_rate
            dZ = dA
    return dWs, dbs

# Function to update the network's weights and biases
def update_weights(weights, biases, dWs, dbs, learning_rate):
    for i in range(len(weights)):
        weights[i] -= learning_rate * dWs[i]  # Update weights
        biases[i] -= learning_rate * dbs[i]  # Update biases
    return weights, biases

# Function to train the network with validation
def train_network_with_validation(X_train, y_train, X_val, y_val,
    layer_sizes, learning_rate=0.1, epochs=500, dropout_rate=0.5):
    weights, biases = initialize_network(layer_sizes)
    train_accuracies = []
    val_accuracies = []

    for epoch in range(epochs):
        # Forward pass
        activations, Zs = forward_propagation(X_train, weights, biases,
            dropout_rate)
        # Backward pass
        dWs, dbs = backward_propagation(y_train, activations, Zs,
            weights, dropout_rate)
```

```
        # Update weights and biases
        weights, biases = update_weights(weights, biases, dWs, dbs,
            learning_rate)

        # Calculate training accuracy
        train_predictions = np.argmax(activations[-1], axis=1)
        train_accuracy = np.mean(train_predictions == y_train)
        train_accuracies.append(train_accuracy)

        # Calculate validation accuracy
        val_activations, _ = forward_propagation(X_val, weights, biases
            , dropout_rate)
        val_predictions = np.argmax(val_activations[-1], axis=1)
        val_accuracy = np.mean(val_predictions == y_val)
        val_accuracies.append(val_accuracy)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Training Accuracy: {train_accuracy},
                Validation Accuracy: {val_accuracy}")

    return weights, biases, train_accuracies, val_accuracies

# Function to plot training and validation accuracies over epochs
def plot_accuracies(train_accuracies, val_accuracies):
    plt.plot(train_accuracies, label='Training Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracy over Epochs')
    plt.show()

# Load data from the specified file path
file_path = '/content/MNIST_Subset.h5'
X, Y = load_mnist(file_path)

# Normalize data
X = X / 255.0

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, Y, test_size=0.5,
    random_state=10)

# Define layer sizes for different configurations
layer_sizes_1 = [784, 100, 10]
layer_sizes_2 = [784, 100, 50, 50, 10]
```

```
# Train the network with 1 hidden layer and track accuracies
print(f"Accuracy for 1 hidden layer with ReLU:")
weights_1, biases_1, train_accuracies_1, val_accuracies_1 =
    train_network_with_validation(X_train, y_train, X_val, y_val,
    layer_sizes_1, learning_rate=0.1, dropout_rate=0.5)

# Train the network with 3 hidden layers and track accuracies
print(f"Accuracy for 3 hidden layers with ReLU:")
weights_2, biases_2, train_accuracies_2, val_accuracies_2 =
    train_network_with_validation(X_train, y_train, X_val, y_val,
    layer_sizes_2, learning_rate=0.1, dropout_rate=0.5)

# Plot the accuracies for both configurations
plot_accuracies(train_accuracies_1, val_accuracies_1)
plot_accuracies(train_accuracies_2, val_accuracies_2)
```

3.2.1 Output

Accuracy for 1 hidden layer with ReLU:

- Epoch 0, Training Accuracy: 0.017543859649122806, Validation Accuracy: 0.36373842267751894
- Epoch 10, Training Accuracy: 0.8098245614035088, Validation Accuracy: 0.8227617176536627
- Epoch 20, Training Accuracy: 0.8649824561403509, Validation Accuracy: 0.8685096828515296
- Epoch 30, Training Accuracy: 0.8905263157894737, Validation Accuracy: 0.8967162503508279
- Epoch 40, Training Accuracy: 0.9024561403508772, Validation Accuracy: 0.8999438675273645
- Epoch 50, Training Accuracy: 0.9103157894736842, Validation Accuracy: 0.912573673870334
- Epoch 60, Training Accuracy: 0.9131228070175439, Validation Accuracy: 0.9181869211338759

- Epoch 70, Training Accuracy: 0.9209824561403509, Validation Accuracy: 0.9239404995790065
- Epoch 80, Training Accuracy: 0.9284210526315789, Validation Accuracy: 0.9266067920291888
- Epoch 410, Training Accuracy: 0.9651929824561404, Validation Accuracy: 0.9604266067920292
- Epoch 420, Training Accuracy: 0.9664561403508772, Validation Accuracy: 0.9609879315183834
- Epoch 430, Training Accuracy: 0.9637894736842105, Validation Accuracy: 0.961408925063149
- Epoch 440, Training Accuracy: 0.9609824561403508, Validation Accuracy: 0.9625315745158575
- Epoch 450, Training Accuracy: 0.9661754385964912, Validation Accuracy: 0.9666011787819253
- Epoch 460, Training Accuracy: 0.9657543859649123, Validation Accuracy: 0.962671905697446
- Epoch 470, Training Accuracy: 0.9646315789473684, Validation Accuracy: 0.9636542239685658
- Epoch 480, Training Accuracy: 0.9677192982456141, Validation Accuracy: 0.9632332304238002
- Epoch 490, Training Accuracy: 0.9658947368421053, Validation Accuracy: 0.9643558798765086

Accuracy for 3 hidden layers with ReLU:

- Epoch 0, Training Accuracy: 0.23789473684210527, Validation Accuracy: 0.31013191131069323

- Epoch 10, Training Accuracy: 0.5247719298245614, Validation Accuracy: 0.5235756385068763
- Epoch 20, Training Accuracy: 0.5525614035087719, Validation Accuracy: 0.5493965759191692
- Epoch 30, Training Accuracy: 0.5781052631578948, Validation Accuracy: 0.5819534100477126
- Epoch 40, Training Accuracy: 0.5957894736842105, Validation Accuracy: 0.6053887173730003
- Epoch 50, Training Accuracy: 0.6186666666666667, Validation Accuracy: 0.6243334268874544
- Epoch 60, Training Accuracy: 0.639859649122807, Validation Accuracy: 0.652119000841987
- Epoch 70, Training Accuracy: 0.672140350877193, Validation Accuracy: 0.654925624473758
- Epoch 80, Training Accuracy: 0.6797192982456141, Validation Accuracy: 0.6824305360651136
- Epoch 90, Training Accuracy: 0.7056842105263158, Validation Accuracy: 0.7039012068481617
- Epoch 160, Training Accuracy: 0.8050526315789474, Validation Accuracy: 0.8070446253157452
- Epoch 170, Training Accuracy: 0.8172631578947368, Validation Accuracy: 0.8147628403031153
- Epoch 200, Training Accuracy: 0.8437894736842105, Validation Accuracy: 0.8470390120684816
- Epoch 470, Training Accuracy: 0.9192982456140351, Validation Accuracy: 0.9122930115071569

- Epoch 480, Training Accuracy: 0.9214035087719298, Validation Accuracy: 0.912573673870334
- Epoch 490, Training Accuracy: 0.919438596491228, Validation Accuracy: 0.9156609598652821

3.2.2 Overfitting and Underfitting ReLU & Softmax

For 1 hidden layer for 500 iterations using ReLU

1. **Training and Validation Curves:** Both the training and validation accuracy increase together and plateau at around 90% accuracy over 500 epochs.
2. **Analysis:** This model appears to be **well-trained** and has good generalization. The training and validation curves follow each other closely, with no significant divergence. This suggests that the model is not overfitting, and it is generalizing well to unseen data.
3. **Challenges:**
    - The training and validation curves are still slowly improving, so it may indicate a potential opportunity for improvement in model performance with more tuning (e.g., learning rate).
    - A potential countermeasure could be early stopping or adjusting learning rate schedules for better convergence.

For 3 hidden layer for 500 iterations using ReLU

1. **Training and Validation Curves:** The curves increase together up to around 300 epochs. After that, the validation accuracy starts to fluctuate and separates slightly from the training accuracy.
2. **Analysis:** This plot indicates a **slight overfitting case.** Initially, the training and validation curves are closely aligned, but over time, the model is learning patterns in the training set that do not generalize as well to the validation data.

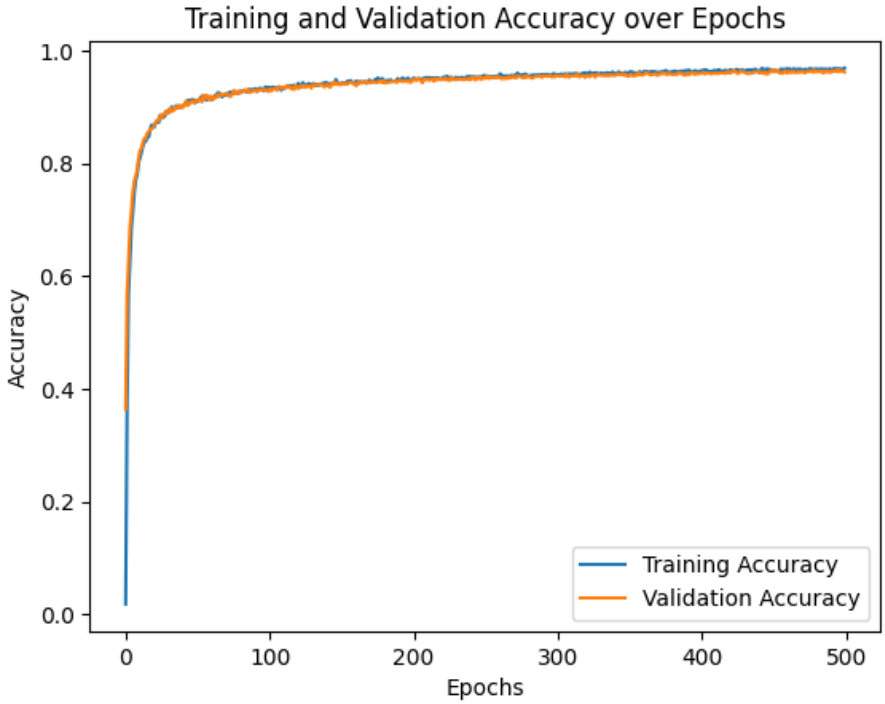

Figure 3.3: For 1 hidden layer for 500 iterations using ReLU

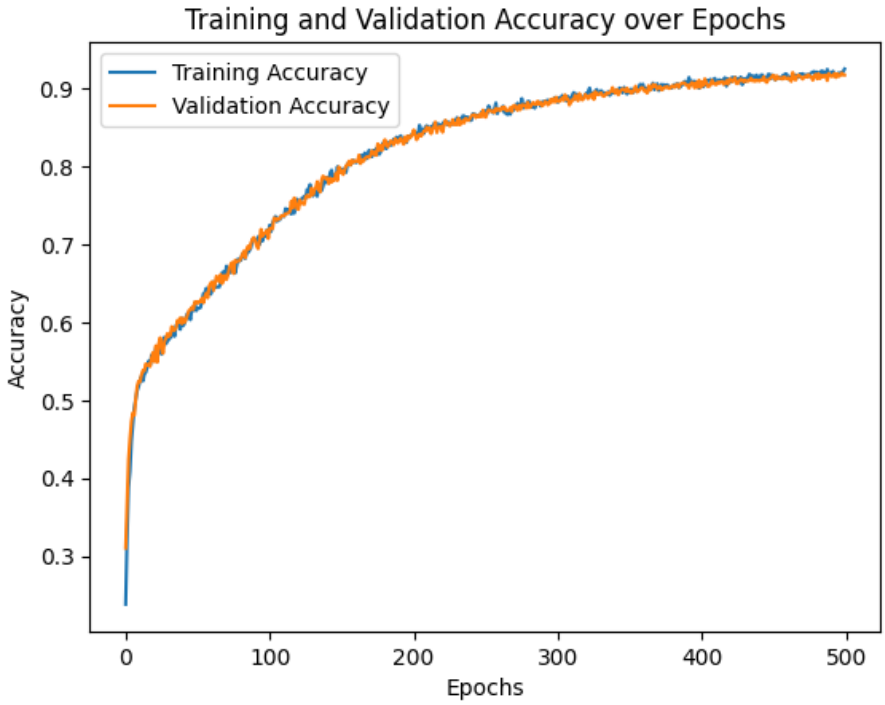

Figure 3.4: For 3 hidden layers for 500 iterations using ReLU

3. **Challenges:** Overfitting often occurs when a model is too complex or trained for too long. This could be addressed by adding regularization (e.g., dropout or L2 regularization), reducing model complexity, or applying early stopping based on validation performance.

3.3 Calculating Accuracy using Sklearn

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Function to load MNIST data from an HDF5 file
def load_mnist(filepath="/content/MNIST_Subset.h5"):
    with h5py.File(filepath, 'r') as f:
        X = f['X'][:]
        Y = f['Y'][:]
    return X, Y

# Function to plot training and validation accuracies over epochs
def plot_accuracies(train_accuracies, val_accuracies):
    plt.plot(train_accuracies, label='Training Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracy over Epochs')
    plt.show()

# Function to train an MLPClassifier with tracking of accuracies
def train_sklearn_with_tracking(X_train, y_train, X_val, y_val,
   layer_sizes, max_iter=200):
    # Initialize the MLPClassifier with specified layer sizes and other
        parameters
    clf = MLPClassifier(hidden_layer_sizes=layer_sizes, activation='
       relu', solver='adam', max_iter=100, warm_start=True,
       random_state=10)
    train_accuracies = []
    val_accuracies = []

    # Train the classifier for a specified number of epochs
    for epoch in range(max_iter):
```

```
        clf.fit(X_train, y_train)
        train_accuracy = accuracy_score(y_train, clf.predict(X_train))
        val_accuracy = accuracy_score(y_val, clf.predict(X_val))
        train_accuracies.append(train_accuracy)
        val_accuracies.append(val_accuracy)
        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Training Accuracy: {train_accuracy},
                Validation Accuracy: {val_accuracy}")

    return clf, train_accuracies, val_accuracies

# Load data from the specified file path
file_path = '/content/MNIST_Subset.h5'
X, Y = load_mnist(file_path)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, Y, test_size=0.5,
    random_state=10)

# Define layer sizes for different neural networks
layer_sizes_1 = (100,)
layer_sizes_2 = (100, 50, 50)

# Train the network with 1 hidden layer and track accuracies
print(f"Accuracy for 1 hidden layer with Sklearn:")
# Reshape X_train and X_val to 2D arrays
clf_1, train_accuracies_1, val_accuracies_1 =
    train_sklearn_with_tracking(X_train.reshape(X_train.shape[0], -1),
    y_train, X_val.reshape(X_val.shape[0], -1), y_val, layer_sizes_1)

# Train the network with 3 hidden layers and track accuracies
print(f"Accuracy for 3 hidden layers with Sklearn:")
clf_2, train_accuracies_2, val_accuracies_2 =
    train_sklearn_with_tracking(X_train.reshape(X_train.shape[0], -1),
    y_train, X_val.reshape(X_val.shape[0], -1), y_val, layer_sizes_2)

# Plot the accuracies for both neural networks
plot_accuracies(train_accuracies_1, val_accuracies_1)
plot_accuracies(train_accuracies_2, val_accuracies_2)
```

3.3.1 Output

Accuracy for 1 hidden layer with Sklearn:

- Epoch 0, Training Accuracy: 1.0, Validation Accuracy: 0.9862475442043221

- Epoch 10, Training Accuracy: 0.9973333333333333, Validation Accuracy: 0.9865282065674993
- Epoch 20, Training Accuracy: 0.9953684210526316, Validation Accuracy: 0.9831602582093741
- Epoch 30, Training Accuracy: 0.992561403508772, Validation Accuracy: 0.9789503227617177
- Epoch 40, Training Accuracy: 0.9949473684210526, Validation Accuracy: 0.9803536345776032
- Epoch 50, Training Accuracy: 0.991719298245614, Validation Accuracy: 0.976985686219478
- Epoch 60, Training Accuracy: 0.9977543859649123, Validation Accuracy: 0.9866685377490878
- Epoch 170, Training Accuracy: 0.9991578947368421, Validation Accuracy: 0.9877911872017963
- Epoch 180, Training Accuracy: 0.9983157894736842, Validation Accuracy: 0.9873701936570306
- Epoch 190, Training Accuracy: 0.999719298245614, Validation Accuracy: 0.9870895312938535

Accuracy for 3 hidden layers with Sklearn:

- Epoch 0, Training Accuracy: 1.0, Validation Accuracy: 0.9767050238563009
- Epoch 10, Training Accuracy: 0.9823157894736843, Validation Accuracy: 0.9733370754981757
- Epoch 20, Training Accuracy: 0.9963508771929824, Validation Accuracy: 0.9870895312938535
- Epoch 30, Training Accuracy: 0.9981754385964913, Validation Accuracy: 0.9873701936570306

- Epoch 40, Training Accuracy: 0.9983157894736842, Validation Accuracy: 0.9855458882963795
- Epoch 50, Training Accuracy: 0.9978947368421053, Validation Accuracy: 0.9890541678360932
- Epoch 60, Training Accuracy: 0.9983157894736842, Validation Accuracy: 0.9880718495649733
- Epoch 170, Training Accuracy: 0.999438596491228, Validation Accuracy: 0.9868088689306764
- Epoch 180, Training Accuracy: 0.9991578947368421, Validation Accuracy: 0.9877911872017963
- Epoch 190, Training Accuracy: 0.999859649122807, Validation Accuracy: 0.9894751613808588

3.3.2 Overfitting and Underfitting for Sklearn

1. For 1 hidden layer for 200 iterations using Sklearn

Observation: The training accuracy is consistently higher than the validation accuracy, especially after the initial epochs. There is a slight gap between training and validation accuracy, with the validation curve showing more fluctuations.
Conclusion: This plot suggests **overfitting** The model performs well on the training data (close to 100% accuracy) but struggles to generalize on the validation data. The fluctuations in validation accuracy indicate that the model is sensitive to validation data and fails to maintain stability.

Challenge & Countermeasures:

- **Challenge:** Overfitting indicates that the model is learning noise or irrelevant patterns in the training data.

- **Countermeasures:** Techniques like early stopping, regularization (L1/L2), dropout, or gathering more training data could help prevent overfitting.

2. For 3 hidden layers for 200 iterations using Sklearn

Observation: The training accuracy is again significantly higher than the validation accuracy. The validation accuracy fluctuates more than in the first image, and the gap between training and validation accuracy is notable.
Conclusion: This also shows signs of **overfitting**, as the training accuracy is nearly perfect, but the validation accuracy does not improve much and fluctuates within a limited range.

Challenge & Countermeasures:

- **Challenge:** Similar to the first graph, this indicates overfitting, where the model fails to generalize to new data.
- **Countermeasures:** Using techniques like cross-validation or data augmentation could help reduce overfitting by introducing more variability in the training data.

3.3.3 Observed difference in Accuracy

The observed differences in accuracies can be attributed to the differences in optimization algorithms, initialization strategies, activation function characteristics, regularization techniques, and hyperparameter tuning. Sklearn's implementation likely benefits from more sophisticated optimization and regularization techniques, leading to higher and more stable accuracies compared to using Sigmoid and ReLU activations in a typical neural network setup.

1 Hidden Layer with Sklearn:

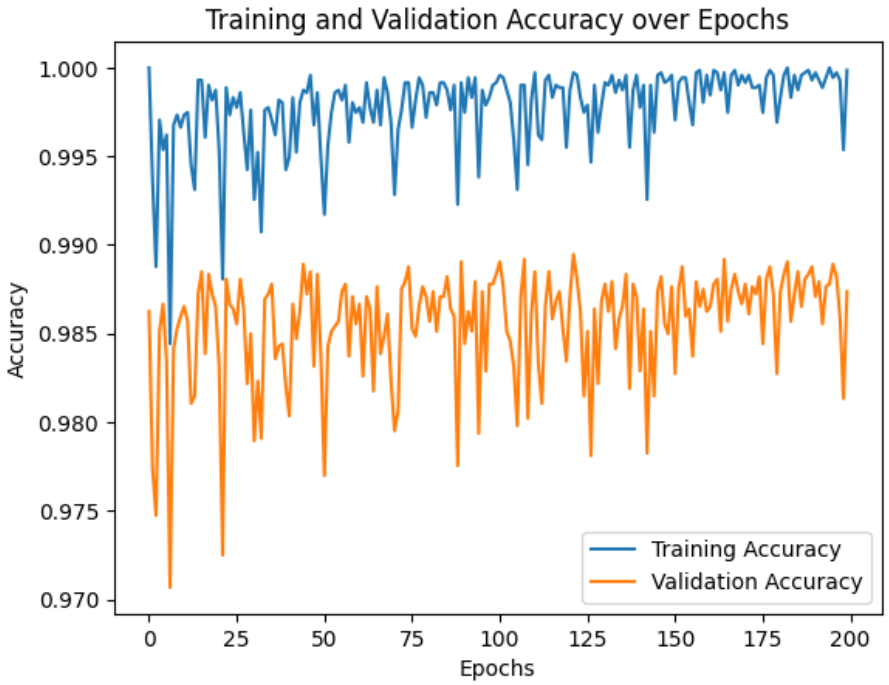

Figure 3.5: For 1 hidden layer for 200 iterations using Sklearn

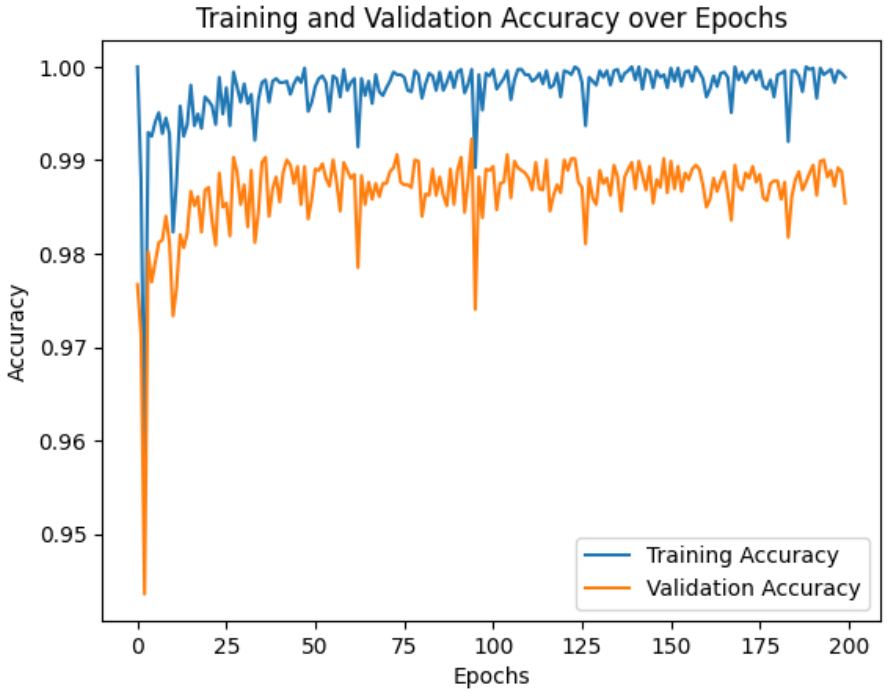

Figure 3.6: For 3 hidden layers for 200 iterations using Sklearn

- **Training Accuracy:** Starts at 100% and remains very high (close to 100%) throughout the epochs.
- **Validation Accuracy:** Starts at around 98.6% and remains high, fluctuating slightly but staying close to the training accuracy.

3 Hidden Layers with Sklearn:

- **Training Accuracy:** Starts at 100% and remains very high (close to 100%) throughout the epochs.
- **Validation Accuracy:** Starts at around 97.6% and remains high, fluctuating slightly but staying close to the training accuracy.

Comparison with Sigmoid Activation Function:

- **1 Hidden Layer:** Training and validation accuracies were high but not as close to 100% as with Sklearn.
- **3 Hidden Layers:** Training and validation accuracies were high but showed more fluctuation and were not as close to 100% as with Sklearn.

Comparison with ReLU Activation Function:

- **1 Hidden Layer:** Training and validation accuracies were high but showed more fluctuation compared to Sklearn.
- **3 Hidden Layers:** Training and validation accuracies were high but showed more fluctuation compared to Sklearn.

Reasons for Differences in Accuracy

1. Optimization Algorithms:

- **Sklearn:** Likely uses a different optimization algorithm (e.g., LBFGS) which can converge faster and more accurately for certain types of problems.

- **Sigmoid/ReLU:** Typically use gradient descent-based optimizers (e.g., SGD, Adam) which may not converge as quickly or accurately.

2. Initialization:

- **Sklearn:** May use a more sophisticated initialization strategy that helps in faster convergence.
- **Sigmoid/ReLU:** Initialization can significantly affect the training process, especially for deeper networks.

3. Activation Function Characteristics:

- **Sigmoid:** Can suffer from vanishing gradient problems, especially in deeper networks, leading to slower convergence and lower accuracy.
- **ReLU:** Can suffer from dying ReLU problem where neurons can become inactive, leading to slower convergence and lower accuracy.
- **Sklearn:** May use different activation functions or handle activation in a way that avoids these issues.

4. Regularization:

- **Sklearn:** May have built-in regularization techniques that prevent overfitting more effectively.
- **Sigmoid/ReLU:** Regularization needs to be explicitly added and tuned, which can be challenging.

5. Hyperparameter Tuning:

- **Sklearn:** May have better default hyperparameters or easier tuning mechanisms.
- **Sigmoid/ReLU:** Requires careful tuning of learning rate, batch size, etc., which can affect performance.

Chapter 4

Conclusion

This project aimed to evaluate the performance of different activation functions (Sigmoid, ReLU) combined with the Softmax function for output layer classification on the MNIST dataset, using neural networks with 1 and 3 hidden layers. Additionally, we explored the performance of a simple neural network built using Scikit-learn's implementation for comparison. The study focused on key aspects such as model generalization, overfitting, and convergence.

4.1 Summary of Results

Sigmoid and Softmax (1 and 3 Hidden Layers):

- Using Sigmoid in the hidden layers and Softmax in the output layer, the model showed good generalization performance in both 1 and 3 hidden layer configurations.
- The training and validation accuracy curves for both cases indicated no significant divergence, suggesting that the models were well-trained with no signs of overfitting.
- The model effectively learned from the data, with convergence to high accuracy over the epochs. The training process demonstrated smooth and stable convergence for both cases, with validation accuracy closely following the training accuracy, particularly for 1 hidden layer.

ReLU and Softmax (1 and 3 Hidden Layers):

- When using ReLU activation in the hidden layers with Softmax for classification, the model with 1 hidden layer demonstrated excellent generalization. Both the training and validation accuracies increased steadily with no signs of overfitting.
- However, in the 3 hidden layer configuration, slight overfitting was observed as the training accuracy exceeded validation accuracy after prolonged training. This suggests that the deeper model was more prone to learning noise from the training set, leading to reduced generalization performance.
- Despite the overfitting in the deeper model, the convergence rate of ReLU networks was faster compared to the Sigmoid-based models, reflecting the efficiency of ReLU in deep architectures.

Scikit-learn Model (1 and 3 Hidden Layers):

- The model built using Scikit-learn showed clear signs of overfitting in both 1 and 3 hidden layer setups. The training accuracy was very high, but validation performance lagged behind, indicating the model had memorized the training data but struggled to generalize to unseen data.
- Convergence in these models was relatively fast, but the overfitting suggests a lack of regularization or capacity to generalize, making it less effective despite its faster learning.
- However, the higher accuracy of the Scikit-learn model came at the cost of overfitting, making it less reliable for real-world generalization tasks.

4.2 Accuracy Ranking

1. Scikit-learn model

2. ReLU-based model

3. Sigmoid-based model

4.3 Convergence Analysis

- Sigmoid + Softmax models, though slower to converge compared to ReLU, exhibited steady and consistent learning, particularly with 1 hidden layer, indicating stable training behavior.
- ReLU + Softmax models converged faster but were prone to overfitting in the 3 hidden layer case, demonstrating that while ReLU accelerates learning, it requires more regularization in deeper models.
- The Scikit-learn model converged quickly but showed significant overfitting, particularly in deeper configurations.

In conclusion, this study demonstrates that using Sigmoid with Softmax results in well-trained and generalized models for both shallow and deep neural networks, whereas ReLU with Softmax is effective in shallow architectures but prone to overfitting in deeper models. The Scikit-learn model showed the highest accuracy but overfitted in both hidden layer configurations, indicating it is not as reliable for generalization. The choice of activation function and depth significantly impacts the model's ability to generalize, and proper regularization techniques are essential, especially when using ReLU in deeper networks.

Bibliography

[1] https://www.geeksforgeeks.org/load-h5-files-in-python/

[2] https://medium.com/@koushikkushal95/mnist-hand-written-digit-classification-using-neural-network-from-scratch-54da85712a06

[3] https://github.com/numpy/numpy-tutorials/blob/main/content/tutorial-deep-learning-on-mnist.md