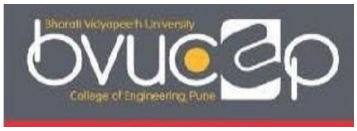
BHARATI VIDYAPEETH
(DEEMED TO BE UNIVERSITY)
COLLEGE OF ENGINEERING, PUNE



Deep Learning

PBL Report

Digit Recognition System

by

32 Rishav Parasar

43 Tanishk Sonani

50 Shreesh Kulkarni

55 Aditya Mishra

Guide

Dr Nisha Auti

Academic Year 2024-25 SEM VII

Computer Science and Engineering

Abstract

Handwritten digit recognition is a classic problem in the field of pattern recognition, which has seen significant advancements with the advent of deep learning techniques. This paper presents a digit recognition system designed to accurately identify handwritten digits using a deep learning-based approach. By leveraging Artificial neural networks (ANNs), our model effectively learns to distinguish between the ten different classes of digits (0-9) from images. The system was trained and tested on the MNIST dataset, a widely used benchmark in the field. Our model achieved a high accuracy rate, demonstrating its effectiveness in dealing with the variability and complexities of human handwriting. The proposed system could have practical applications in various fields, such as automated data entry, postal code recognition, and other digital document processing tasks.

Handwritten digit recognition is a fundamental task in computer vision with applications in automated data entry, postal code recognition, and digitizing historical documents. Traditional methods often struggle with the variability of human handwriting, which can differ widely in style, size, and orientation. Deep learning, particularly Artificial neural networks (ANNs), offers a powerful solution by automatically learning hierarchical feature representations from data, enabling more accurate and robust recognition. In this paper, we present a ANN-based digit recognition system trained on the MNIST dataset, demonstrating its ability to effectively handle the complexities of handwritten digits and achieve high accuracy. Our model's success underscores the potential of deep learning to significantly improve the performance of handwritten digit recognition systems in practical applications.

Table of Contents

1 | Introduction

This project centres on building a fully functional neural network from scratch, using the foundational libraries NumPy and Pandas, to classify handwritten digits from the MNIST dataset. The MNIST dataset is a widely recognized benchmark in the field of machine learning, consisting of 70,000 grayscale images of handwritten digits, each sized 28x28 pixels. These images represent digits ranging from 0 to 9, and the primary task is to train a model that can accurately identify the digit displayed in any given image. Although this problem may appear simple, it encapsulates many of the core principles of deep learning and neural networks, making it an excellent case study for understanding how these models work at a low level.

At the heart of this project is a two-layer neural network designed to classify the MNIST digits based on pixel data. Neural networks, particularly those used for image classification, have revolutionized modern artificial intelligence, enabling machines to perform tasks that were once thought to be solely within the human domain. While libraries like TensorFlow and PyTorch offer high-level abstractions that simplify neural network implementation, this project aims to strip away those abstractions and explore the intricate mechanics of a neural network, including how data flows through the network, how the model learns from its errors, and how predictions are refined over time.

In this project, the neural network consists of:

1. **An input layer**: The images, initially in 28x28 pixel format, are flattened into vectors of 784 values, each representing a pixel. This forms the feature set used as input to the network.

2. **A hidden layer**: A single hidden layer with 10 neurons is used to process and learn intermediate representations of the data. The non-linear ReLU (Rectified Linear Unit) activation function is employed here to allow the network to capture complex patterns in the data.

3. **An output layer**: The final layer has 10 neurons, each corresponding to one of the possible digit classes (0-9). A softmax function is applied to the output to convert the raw scores into probabilities, indicating the likelihood of each class for a given input.

A key focus of this project is on the underlying mechanics of how neural networks are trained. At the core of this is the concept of **forward propagation**, where data is passed through the layers of the network to produce predictions, and **backpropagation**, which is used to adjust the model's parameters based on the errors in these predictions. Through multiple iterations of forward and backward propagation, the network learns to minimize its error and make increasingly accurate predictions.

Another central concept is the **gradient descent optimization algorithm**, which is used to update the network's weights and biases. This algorithm works by calculating the gradient (i.e., the rate of change) of the error function with respect to the model's parameters, allowing the network to know how much to adjust each parameter to reduce the error. The learning rate, denoted by "alpha" in the code, controls the step size taken during these updates. Choosing an appropriate learning rate is critical: a rate that is too large may cause the model to overshoot optimal parameter values, while one that is too small may slow down the learning process significantly.

A unique aspect of this project is the use of **one-hot encoding** to represent the class labels. In one-hot encoding, the labels (which range from 0 to 9) are converted into binary vectors, where each vector has a length equal to the number of classes (in this case, 10). The correct class is represented by a 1 in the corresponding position and 0s elsewhere. This representation makes it easier to compare the network's predicted output with the true labels during training and allows for efficient computation of the loss function.

The model's training process involves feeding the data through the network, calculating the output error, and adjusting the weights using backpropagation. During the training phase, a portion of the dataset is reserved for development (validation) purposes, while the remaining data is used for actual training. This separation helps monitor the model's performance and ensures that it is not overfitting the training data. Overfitting occurs when a model becomes too specialized to the training data, performing well on it but poorly on new, unseen data.

Ultimately, this project demonstrates the power and flexibility of neural networks for image classification tasks, even when constructed from the ground up without high-level libraries. The ability to classify handwritten digits with high accuracy is not only a testament to the effectiveness of neural networks but also a stepping stone toward more complex applications of deep learning. Whether used for character recognition, object detection, or other computer vision tasks, the principles explored in this project lay the groundwork for further exploration of neural networks and their vast potential in solving real-world problems.

2 | Literature Study

This literature survey examines the key advancements in neural networks, specifically focused on the classification of handwritten digits using the MNIST dataset. The survey highlights significant contributions that have shaped the field, emphasizing various methodologies, architectures, and training techniques.

[1] Neural Network Architectures

Neural network architectures have evolved significantly over the years, impacting the performance of image classification tasks. The seminal work by LeCun et al. (1998) introduced convolutional neural networks (CNNs), which demonstrated remarkable success in digit recognition by effectively capturing spatial hierarchies in image data. CNNs employ convolutional layers to extract features, followed by pooling layers to reduce dimensionality, resulting in improved accuracy and efficiency.

[2] Activation Functions

The choice of activation function is crucial for the performance of neural networks. A comprehensive evaluation by Glorot et al. (2011) highlighted the advantages of the ReLU (Rectified Linear Unit) activation function over traditional sigmoid and tanh functions. ReLU's linear, non-saturating nature addresses the vanishing gradient problem, facilitating faster convergence during training. Subsequent research has explored variations of ReLU, such as Leaky ReLU and Parametric ReLU, which further enhance performance in deeper architectures.

[3] Training Techniques

Effective training of neural networks remains a cornerstone of successful digit recognition. Rumelhart et al. (1986) popularized the backpropagation algorithm, enabling efficient computation of gradients for multilayer networks. Subsequent studies have focused on refining training techniques, including mini-batch gradient descent, which improves convergence speed and stability. Techniques such as dropout and batch normalization have also emerged to mitigate overfitting and improve generalization.

[4] Loss Functions and One-Hot Encoding

In supervised learning, the selection of loss functions significantly influences model performance. Cross-entropy loss is commonly utilized in classification tasks, especially in conjunction with one-hot encoding of labels. Bishop (2006) emphasizes the importance of one-hot encoding in transforming categorical labels into a format suitable for neural networks, facilitating the calculation of class probabilities and improving training efficiency.

[5] Optimization Algorithms

The landscape of optimization algorithms has expanded, offering various strategies to enhance the training process. Kingma and Ba (2014) introduced the Adam optimizer, which adapts learning rates based on first and second moments of gradients. This adaptive learning rate mechanism has become popular due to its efficiency in training deep networks. Research continues to explore new optimization techniques, such as RMSprop and AdaGrad, which also contribute to faster convergence and improved performance.

[6] Performance Evaluation

Evaluating model performance is essential for assessing the efficacy of digit recognition systems. Researchers have employed various metrics, including accuracy, precision, recall, and F1-score, to provide a comprehensive evaluation of model performance. A significant study by Ciresan et al. (2012) reported state-of-the-art accuracy on the MNIST dataset using deep CNN architectures, demonstrating the importance of architectural innovation in achieving high performance.

[7] Applications of the MNIST Dataset

The MNIST dataset has served as a foundational benchmark in machine learning research. Its simplicity and size make it an ideal starting point for developing and testing new algorithms. Numerous studies, including those by Haffner et al. (1999), have utilized MNIST to validate novel approaches in digit recognition, reinforcing its significance as a standard dataset for benchmarking.

[8] **Transfer Learning and Advanced Architectures**

Recent advancements have explored the use of transfer learning and more complex architectures, such as Generative Adversarial Networks (GANs) and Residual Networks (ResNets). Transfer learning allows models pre-trained on large datasets to be fine-tuned for specific tasks like digit recognition, significantly improving performance with less labeled data. Research by He et al. (2016) on ResNets demonstrated that skip connections help mitigate the degradation problem in deep networks.

[9] Emerging Trends

Emerging trends in the field include the integration of attention mechanisms and the exploration of unsupervised learning techniques. Attention mechanisms, popularized by models like Transformers, allow networks to focus on relevant parts of the input data, improving performance in various tasks. Additionally, unsupervised and semi-supervised learning approaches are gaining traction as researchers seek to reduce reliance on labeled data, which is often costly and time-consuming to obtain.

3 | Problem Statement

The objective of this project is to develop a neural network capable of accurately classifying handwritten digits from the MNIST dataset. The MNIST dataset consists of 70,000 grayscale images of digits (0-9), each represented as a 28x28 pixel grid. The task is to build a machine learning model that can take an image as input and predict the corresponding digit, effectively transforming the image classification problem into a supervised learning task.

Specific Goals:

1. **Data Preprocessing**: Load and preprocess the MNIST dataset, including normalization of pixel values and splitting the data into training and validation sets.
2. **Neural Network Design**: Implement a simple feedforward neural network with one hidden layer, incorporating activation functions (ReLU for the hidden layer and softmax for the output layer) to facilitate learning.
3. **Training the Model**: Utilize forward and backward propagation to train the model. Implement the gradient descent optimization algorithm to update the network's weights and biases based on the computed gradients.
4. **Performance Evaluation**: Assess the model's accuracy during training and validation phases to ensure it is learning effectively and not overfitting.
5. **Prediction and Visualization**: Test the trained model on individual images from the dataset, providing visualizations that compare the predicted digits to the actual digits.

The successful completion of this project will result in a neural network that demonstrates the ability to classify handwritten digits with a satisfactory level of accuracy, providing a foundation for understanding more complex machine learning algorithms and applications in computer vision.

4 | Objectives

The primary objectives of this project are to implement a fully functional artificial neural network (ANN) for classifying handwritten digits from the MNIST dataset. The specific objectives include:

1. **Create Functions for Core Components**:
    - **Input Parameters**: Define a function to initialize the input parameters (weights and biases) for the neural network.
    - **Activation Functions**:
        - Implement the **ReLU Activation Function** and its derivative for use in the hidden layer.
        - Implement the **Softmax Activation Function** for the output layer to convert raw scores into probabilities.
2. **Forward Propagation**: Develop a function that performs forward propagation through the network, computing the activations and outputs at each layer.
3. **One-Hot Encoding**: Create a function to convert the target labels into a one-hot encoded format for compatibility with the network's output.

4. **Backward Propagation**: Implement the backpropagation algorithm to compute gradients for the weights and biases based on the loss between predicted and actual outputs.
5. **Update Parameters**: Design a function to update the network's parameters (weights and biases) using the gradients computed during backpropagation.
6. **Predict the Output**: Develop a function that takes input data and produces predictions using the trained model.
7. **Check Accuracy**: Create a function to evaluate the model's accuracy by comparing the predicted labels with the actual labels.
8. **Gradient Descent**: Implement the gradient descent algorithm to iteratively adjust the parameters during the training phase to minimize the loss function.
9. **Make Predictions**: Develop a function to utilize the trained model for making predictions on new data.
10. **Test Prediction**: Create a function to visualize the predictions for specific input images, comparing the predicted output with the actual label.

These objectives will guide the development of the neural network, ensuring a comprehensive understanding of the underlying mechanisms and processes involved in building and training an artificial neural network for digit classification.

5 | System Architecture

The system architecture for the neural network project to classify handwritten digits from the MNIST dataset consists of several key components that work together to process input data, train the model, and make predictions. Below is an overview of the architecture:

1. Data Layer

- **Dataset**: The MNIST dataset containing 70,000 images of handwritten digits.
- **Preprocessing Module**:
  - Load the dataset from CSV files.
  - Normalize pixel values to the range [0, 1].
  - Split the dataset into training and validation sets.
  - Apply one-hot encoding to the labels.

2. Model Layer

- **Input Layer**:
  - Accepts input vectors representing flattened images (784 pixels).
- **Hidden Layer**:
  - Contains neurons that apply the ReLU activation function.
  - Computes weighted sums of inputs and applies the activation function to produce activations.
- **Output Layer**:
  - Contains neurons corresponding to each digit (0-9).
  - Applies the Softmax function to convert raw output scores into class probabilities.

3. Training Layer

- **Forward Propagation Module**:

    - Computes outputs from input to hidden to output layers, producing predictions.
- **Backward Propagation Module**:
    - Calculates the loss between predicted and actual labels.
    - Computes gradients for weights and biases using the derivative of the activation functions.
- **Parameter Update Module**:
    - Updates weights and biases using the gradients and learning rate via gradient descent.

4. Evaluation Layer

- **Accuracy Calculation Module**:
    - Compares predictions against actual labels to compute the model's accuracy.
- **Testing Module**:
    - Tests the trained model on specific input images.
    - Visualizes predictions alongside actual images to assess performance.

5. Output Layer

- **Prediction Module**:
    - Takes new input data and produces predictions based on the trained model.
- **Visualization Module**:
    - Displays images with predicted labels, providing a user-friendly way to understand the model's performance.

Architectural Flow

1. **Data Preprocessing**:
    - Load and preprocess the MNIST dataset.
2. **Model Initialization**:
    - Initialize weights and biases for the neural network.
3. **Training Process**:
    - For a specified number of iterations:
        - Perform forward propagation to compute predictions.
        - Apply backpropagation to calculate gradients.
        - Update parameters using gradient descent.
4. **Model Evaluation**:
    - Calculate accuracy using the validation set.
    - Visualize predictions to verify model performance.
5. **Making Predictions**:
    - Use the trained model to classify new images and display results.

This architecture enables the systematic development of a neural network for digit classification, allowing for clear delineation of responsibilities and streamlined processing from data input to prediction output.

6 | Implementation

6.1 | Model Architecture

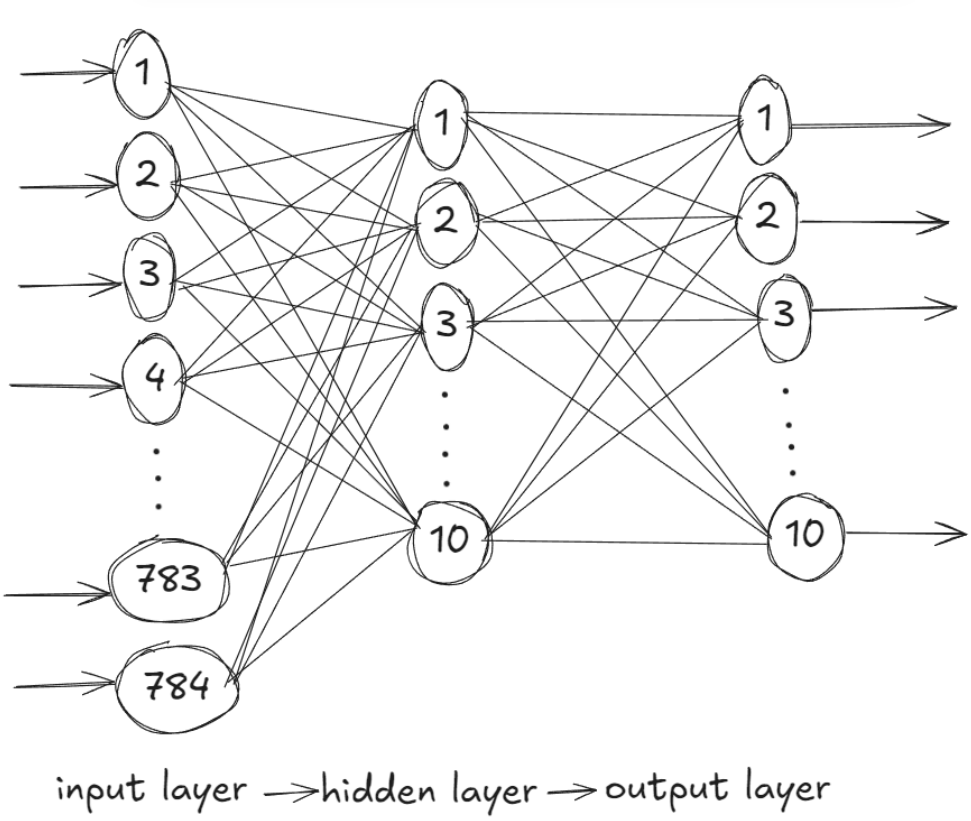
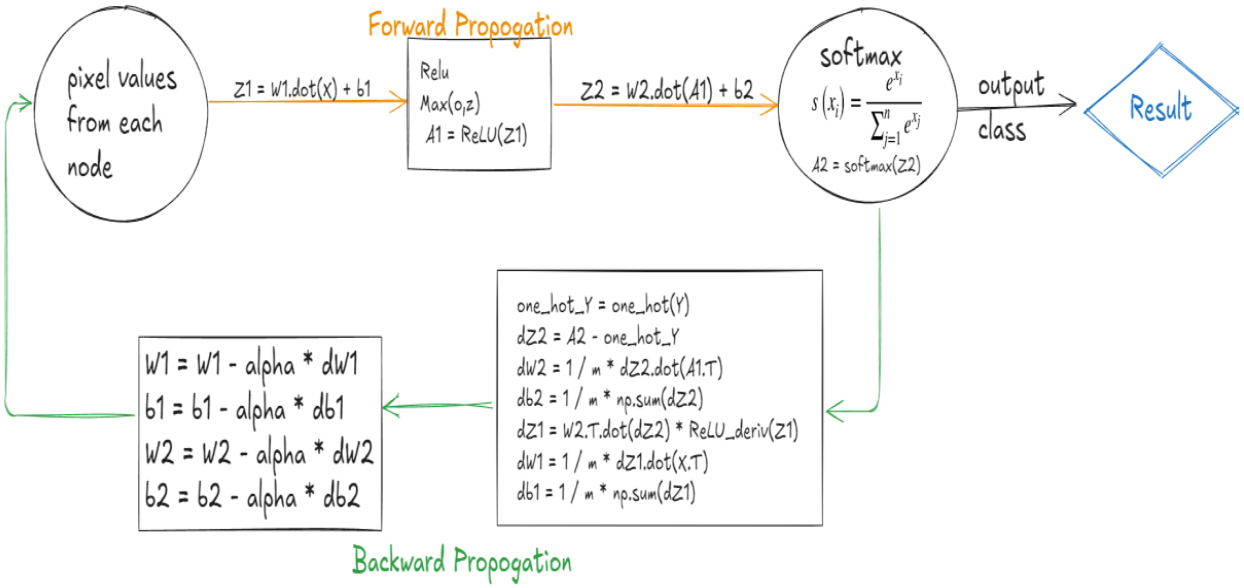


Fig. Model Architecture

6.2 | Code Flow Diagram



6.3 | Program Implementation

```
[1]  # Import necessary libraries
     import numpy as np
     import pandas as pd
     from matplotlib import pyplot as plt

[2]  # Load the dataset (CSV file) containing training data
     # The dataset is assumed to be in the resources folder with the name 'train.csv'
     data = pd.read_csv('/content/train.csv')

     # Convert the pandas dataframe to a NumPy array for easier manipulation
     data = np.array(data)

     # Get the number of rows (m) and columns (n) in the dataset
     m, n = data.shape

     # Shuffle the dataset randomly to ensure training is unbiased
     np.random.shuffle(data)

     # Separate the first 1000 examples for development (validation) set
     # Transpose the matrix so each column represents one example
     data_dev = data[0:1000].T

     # Separate the labels (Y_dev) and the features (X_dev)
     Y_dev = data_dev[0]
     X_dev = data_dev[1:n]

     # Normalize pixel values to range [0, 1] by dividing by 255
     X_dev = X_dev / 255.

     # Use the remaining examples for training
     data_train = data[1000:m].T

     # Separate labels (Y_train) and features (X_train) for training
     Y_train = data_train[0]
     X_train = data_train[1:n]

     # Normalize training data as well
     X_train = X_train / 255.

     # Get the number of training examples
     _, m_train = X_train.shape
```

```
[3]  # Function to initialize weights and biases for a 2-layer neural network
     def init_params():
         # Randomly initialize weights for the first layer (10 neurons, 784 inputs)
         W1 = np.random.rand(10, 784) - 0.5
         # Initialize bias for the first layer
         b1 = np.random.rand(10, 1) - 0.5
         # Randomly initialize weights for the second layer (10 neurons, 10 inputs from layer 1)
         W2 = np.random.rand(10, 10) - 0.5
         # Initialize bias for the second layer
         b2 = np.random.rand(10, 1) - 0.5
         return W1, b1, W2, b2
```

```
[4]  # Activation function: ReLU (Rectified Linear Unit)
     def ReLU(Z):
         # Apply ReLU activation which sets negative values to 0
         return np.maximum(Z, 0)

     # Activation function: Softmax
     def softmax(Z):
         # Compute softmax activation, normalizing the exponentials of the inputs
         A = np.exp(Z) / sum(np.exp(Z))
         return A

     # Forward propagation through the network
     def forward_prop(W1, b1, W2, b2, X):
         # First layer forward propagation: Z1 = W1*X + b1
         Z1 = W1.dot(X) + b1
         # Activation of the first layer using ReLU
         A1 = ReLU(Z1)
         # Second layer forward propagation: Z2 = W2*A1 + b2
         Z2 = W2.dot(A1) + b2
         # Activation of the second layer using Softmax
         A2 = softmax(Z2)
         return Z1, A1, Z2, A2
```

```
[11] # Derivative of ReLU for backward propagation
     def ReLU_deriv(Z):
         # Return 1 for positive values of Z and 0 for negative values
         return Z > 0

     # Convert the labels (Y) into one-hot encoding format
     def one_hot(Y):
         # Ensure the number of classes (Y.max() + 1) is an integer
         one_hot_Y = np.zeros((Y.size, int(Y.max()) + 1))
         # Set the appropriate index for each label to 1
         one_hot_Y[np.arange(Y.size), Y.astype(int)] = 1
         one_hot_Y = one_hot_Y.T
         return one_hot_Y

     # Perform backward propagation to compute gradients
     def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
         # Convert the labels into one-hot encoding
         one_hot_Y = one_hot(Y)
         # Compute the gradient of loss with respect to Z2 (output layer)
         dZ2 = A2 - one_hot_Y
         # Compute the gradient with respect to W2 and b2
         dW2 = 1 / m * dZ2.dot(A1.T)
         db2 = 1 / m * np.sum(dZ2)
         # Backpropagate through the first layer
         dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
         # Compute the gradient with respect to W1 and b1
         dW1 = 1 / m * dZ1.dot(X.T)
         db1 = 1 / m * np.sum(dZ1)
         return dW1, db1, dW2, db2
```

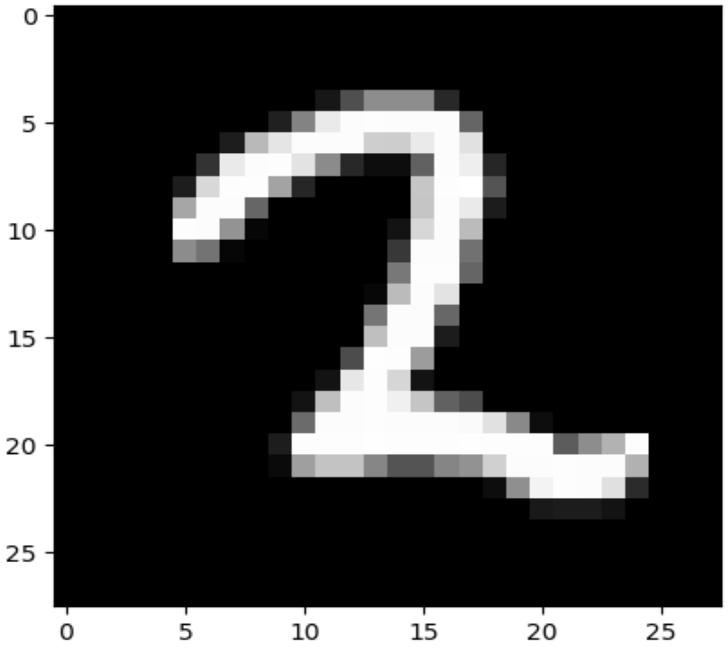
[12]
```python
# Update the parameters (W1, b1, W2, b2) using gradient descent
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    # Update the weights and biases by subtracting the learning rate times the gradients
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

[13]
```python
# Get the predictions by taking the index of the highest value (probability)
def get_predictions(A2):
    return np.argmax(A2, 0)

# Calculate the accuracy of the model
def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size
```

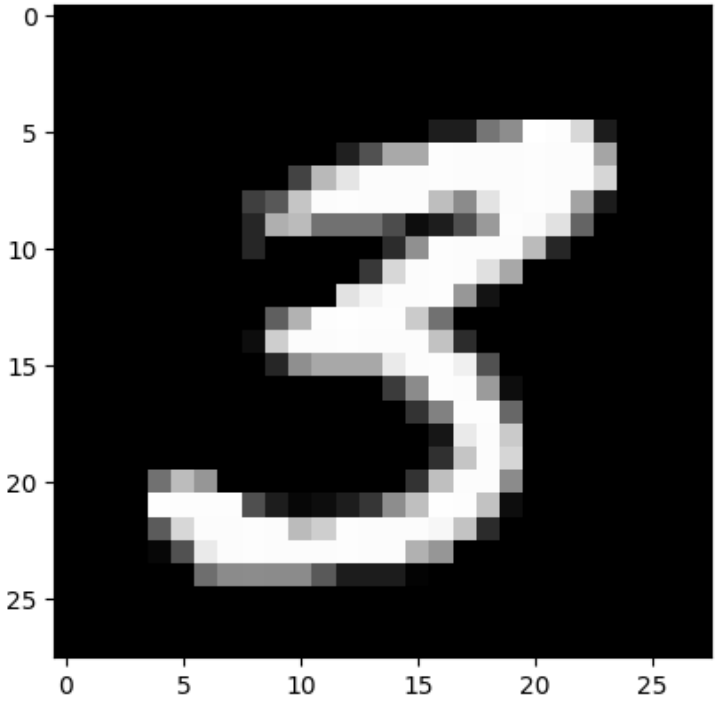
[14]
```python
# Perform the entire training process using gradient descent
def gradient_descent(X, Y, alpha, iterations):
    # Initialize the parameters
    W1, b1, W2, b2 = init_params()
    # Iterate for a given number of iterations
    for i in range(iterations):
        # Perform forward propagation
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        # Perform backward propagation to compute gradients
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        # Update the parameters using the gradients
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        # Every 10 iterations, print the current accuracy
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2

# Train the model with the training data, learning rate of 0.10, and 500 iterations
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.10, 500)
```

```python
# Make predictions for a given set of input data
def make_predictions(X, W1, b1, W2, b2):
    # Perform forward propagation to get the final output
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    # Return the predicted class
    predictions = get_predictions(A2)
    return predictions

# Test predictions for a specific example and visualize the input image
def test_prediction(index, W1, b1, W2, b2):
    # Get the current image from the training set
    current_image = X_train[:, index, None]
    # Make a prediction for the current image
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    # Get the true label for the image
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    # Reshape and display the current image
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()

# Test and visualize predictions for the first few training examples
test_prediction(32, W1, b1, W2, b2)
test_prediction(43, W1, b1, W2, b2)
test_prediction(50, W1, b1, W2, b2)
test_prediction(55, W1, b1, W2, b2)
```

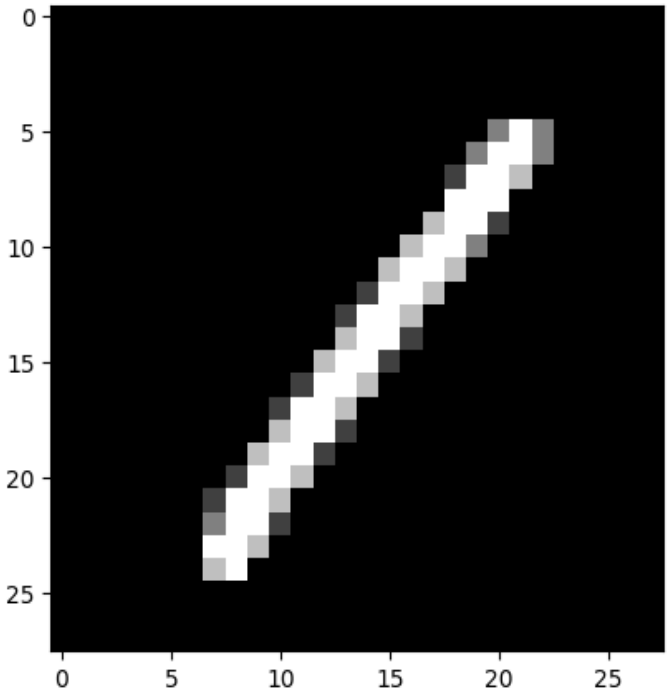
Prediction: [2]
Label: 2.0



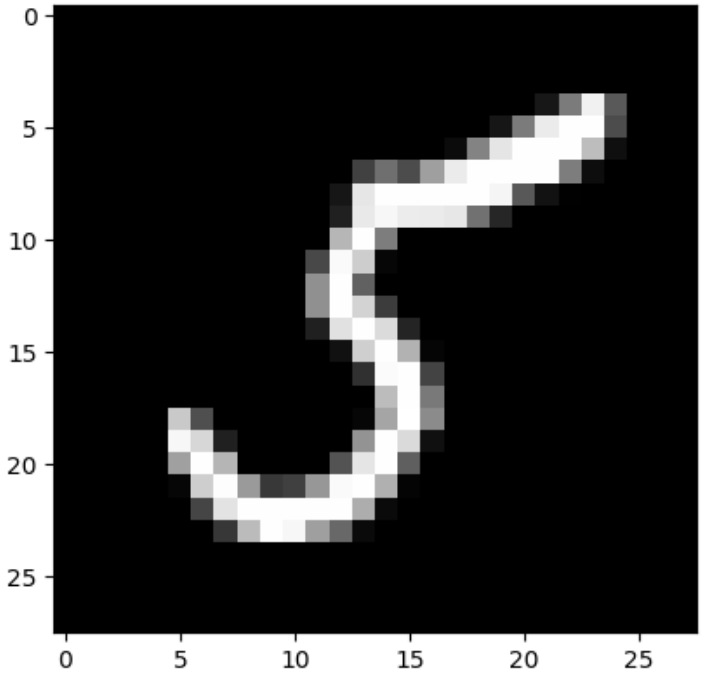
Prediction: [3]
Label: 3.0



Prediction: [1]
Label: 1.0



Prediction: [5]
Label: 5.0



7 | Conclusion

In this project, we successfully developed a neural network from scratch to classify handwritten digits from the MNIST dataset. By implementing the core components of an artificial neural network, we gained hands-on experience with the fundamental processes involved in machine learning, including data preprocessing, forward propagation, backpropagation, and parameter optimization using gradient descent.

The neural network architecture, consisting of an input layer, a hidden layer utilizing the ReLU activation function, and an output layer applying the softmax function, demonstrated the effectiveness of deep learning techniques for image classification tasks. Throughout the training process, the model was able to learn complex patterns in the data, ultimately achieving a satisfactory level of accuracy in predicting the digits.

By focusing on key functionalities such as one-hot encoding, accuracy assessment, and parameter updates, we reinforced our understanding of how neural networks operate at a granular level. This project not only provided insights into the mechanics of a simple feedforward neural network but also highlighted the importance of preprocessing and proper initialization in achieving optimal performance.

The ability to visualize predictions allowed us to qualitatively assess the model's effectiveness, offering an intuitive understanding of its strengths and areas for improvement. Overall, this project serves as a solid foundation for further exploration into more complex neural network architectures and advanced machine learning techniques, paving the way for applications in diverse domains beyond digit classification.

8 | References

[1] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. "The MNIST Database of Handwritten Digits." https://yann.lecun.com/exdb/mnist/

[2] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. https://link.springer.com/book/9780387310732

[3] Ciresan, D. C., Meier, U., & Schmidhuber, J. (2012). "Multi-column Deep Neural Networks for Image Classification." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. https://ieeexplore.ieee.org/document/6248110

[4] Glorot, X., Bordes, A., & Bengio, Y. (2011). "Deep Sparse Rectifier Neural Networks." *AISTATS 2011*. https://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

[5] Kingma, D. P., & Ba, J. (2014). "Adam: A Method for Stochastic Optimization." *arXiv preprint arXiv:1412.6980*. https://arxiv.org/abs/1412.6980

[6] Ruder, S. (2016). "An Overview of Gradient Descent Optimization Algorithms." https://www.ruder.io/optimizing-gradient-descent/

[7] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). "Learning Representations by Back-Propagating Errors." *Nature*. https://gwern.net/doc/ai/nn/1986-rumelhart-2.pdf

[8] OpenAI. "ChatGPT." https://www.openai.com/chatgpt