WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Converting of Handwritten Text into Digital Number Using Deep Learning

V. Rama Krishna[1], P. Jayanth[2], T. Vamshi Krishna[3], R. Vishnu Priya[4], S. Karthik[5], Dr. R. Sivasubramanian[6]

[1,2,3,4,5] UG Student, [6]Assistant Professor

Department of Computer Science & Engineering (AI & ML), Malla Reddy University

ABSTRACT

Handwritten digit classification is a fundamental problem in the field of machine learning and computer vision. The MNIST dataset, consisting of 60,000 training images and 10,000 test images of handwritten digits, has been widely used as a benchmark for evaluating classification algorithms. In this study, we propose a neural network-based approach for classifying the MNIST digits. Our model consists of multiple layers, including an input layer, one or more hidden layers, and an output layer. We preprocess the images by normalizing the pixel values and converting them into a suitable format for the neural network. The model is trained using the training images and their corresponding labels, adjusting the weights of the connections to minimize the difference between the predicted outputs and the actual labels. We evaluate the performance of the model using metrics such as accuracy on the test images. Our results demonstrate the effectiveness of the neural network approach in accurately classifying handwritten digits. This research contributes to the advancement of digit recognition techniques and provides insights into the application of neural networks in image classification tasks.

1 INTRODUCTION

1.1 PROBLEM DEFINITION

The problem definition of MNIST Handwritten Digit Classification using Neural Networks, it's all about training a model to accurately classify handwritten digits from 0 to 9 based on the pixel values of the images. The goal is to develop a neural network that can learn and distinguish between different digits with high accuracy. It's fascinating how neural networks can be trained to recognize and classify handwritten digits effectively

1.2 OBJECTIVE OF PROJECT

The objective of the project for MNIST Handwritten Digit Classification using Neural Networks is to create a model that can accurately identify and classify handwritten digits from 0 to 9. The main goal is to train a neural network to recognize and distinguish between different handwritten digits with high precision. It's exciting to see how neural networks can be applied to solve such a specific and interesting task like digit recognition!

1.3 SCOPE & LIMITATIONS OF PROJECT

1. Model Development: The primary scope of the project is to develop and implement a neural network model capable of accurately classifying handwritten digits from the MNIST dataset.

2. Exploration of Neural Network Architectures: The project may involve exploring different neural network architectures, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), or hybrid models, to determine the most effective architecture for the task.

LIMITATIONS:

Limited dataset complexity: The MNIST dataset consists of relatively simple grayscale images of handwritten digits, which may not fully represent the complexity of real-world handwritten text recognition tasks.

Domain specificity: The model developed for MNIST digit classification may not generalize well to other domains or types of data beyond handwritten digit recognition. Its applicability may be limited to tasks closely related to digit classification.

2. ANALYSIS

2.1 PROJECT PLANNING AND RESEARCH

1. Project Scope Definition:

 Define Objectives: Clearly state the goals of your project. For instance, achieving a certain accuracy level on the MNIST dataset.

 Scope Limitations: Determine any constraints such as time, computational resources, or specific neural network architectures to be used.

2. Literature Review:

 Research Existing Work: Study papers, articles, and tutorials on MNIST digit classification using neural networks. Understand various architectures, optimization techniques, and performance benchmarks.

3. Data Collection and Preprocessing:

 Obtain MNIST Dataset: Access the MNIST dataset either through libraries like TensorFlow/Keras or download it from official sources.

 Data Preprocessing: Normalize pixel values, reshape images if necessary, and split the dataset into training, validation, and testing sets.

4. Model Selection and Design:

Select Neural Network Architecture: Choose from various architectures like Convolutional

Neural Networks (CNNs), Fully Connected Networks, or hybrid architectures.

Model Design: Design the neural network architecture considering factors such as depth, width, activation functions, and regularization techniques.

5. Implementation:

Coding: Implement the chosen neural network architecture using a deep learning framework like TensorFlow or PyTorch.

Training: Train the model on the training dataset using appropriate optimization algorithms (e.g., Adam, RMSprop) and loss functions (e.g., categorical cross- entropy).

6. Model Evaluation:

 Validation: Evaluate the model's performance on the validation set to tune hyperparameters and prevent overfitting.

 Testing: Assess the final model's performance on the unseen test dataset to measure its generalization ability.

7. Results Analysis:

 Accuracy Metrics: Calculate accuracy, precision, recall, and F1-score to evaluate the model's performance comprehensively.

 Error Analysis: Analyze misclassified samples to identify patterns and potential areas for improvement.

2.2 SOFTWARE REQUIREMENT SPECIFICATION

2.2.1 Software Specifications

• The Programming Language: PythonPython is widely used in the machine learning community due to its extensive libraries for data manipulation, visualization, and deep learning TensorBoard, OpenCV (optional), CUDA and cuDNN for GPU acceleration (optional)

• Deep Learning Framework: TensorFlow with Keras

• Data Processing and Visualization: NumPy, Matplotlib, Pandas

• Model Development and Training: TensorFlow/Keras, Scikit-learn

• Development Environment: Jupyter Notebooks, Integrated Development Environment (IDE) like PyCharm or VSCode

• Version Control: Git

• Additional Libraries and Tools: TensorBoard, OpenCV (optional), CUDA, and cuDNN for GPU acceleration (optional)

2.2.2 Hardware Specifications

 CPU (Central Processing Unit): A multi-core CPU is typically sufficient for training and inference, especially for small to medium-sized neural networks and datasets. Higher clock speeds and more cores can help accelerate training times, particularly for computationally intensive operations.

 GPU (Graphics Processing Unit): A GPU, particularly those optimized for deep learning tasks (e.g., NVIDIA GeForce, Tesla, or Quadro series), can significantly accelerate training times compared to CPUs. GPUs are well-suited for parallel processing of matrix operations common in neural network training, leading to faster convergence.

 TPU (Tensor Processing Unit): TPUs, developed by Google, are specialized hardware accelerators designed specifically for deep learning tasks. TPUs offer even faster training times compared to GPUs, particularly for large-scale neural network models and datasets. Google Cloud Platform provides access to TPUs for training deep learning models in the cloud.

 Memory (RAM): Sufficient RAM is essential for loading and processing datasets, especially during training when batches of data are loaded into memory.

 Storage (HDD/SSD): Adequate storage is necessary for storing datasets, model checkpoints, and training logs. SSDs(Solid State Drives) are preferred over HDDs (Hard Disk Drives) for faster read/write speeds, which can reduce loading times during training and inference.

 Network Connectivity: Reliable network connectivity is essential, especially when training models on cloud platforms or distributed systems. High-speed internet connections ensure efficient data transfer between local machines and cloud-based resources.

2.3 MODEL SELECTION AND ARCHITECTURE

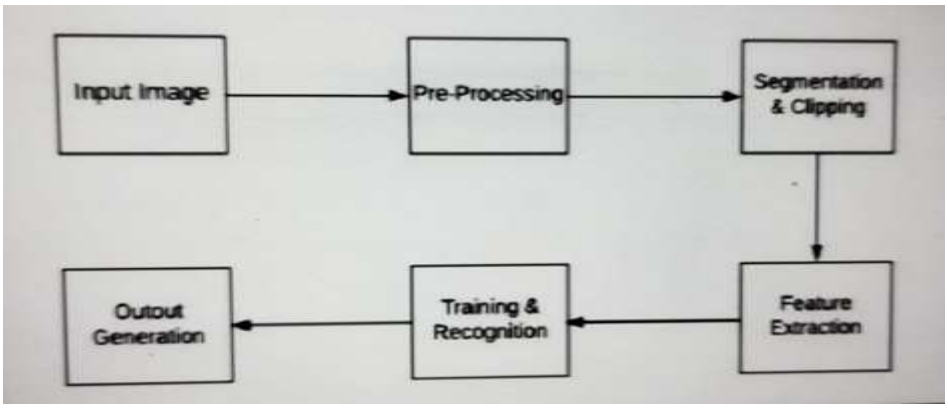


FIG – 2.3.1 System Architecture

Input Image: Begin with an image containing handwritten digits, which serves as the input to the digit recognition system.

Pre-Processing: The input image undergoes pre-processing to enhance its quality and prepare it for analysis. This may involve cleaning the image to remove noise and artifacts and normalizing it to ensure consistent brightness and contrast across digits.

Segmentation & Clipping: If the input image contains multiple digits, segmentation techniques are applied to separate each digit into individual components. These segmented digits are then clipped or cropped to isolate them from the background.

Feature Extraction: Relevant features are extracted from the segmented digits to capture distinctive characteristics that aid in digit recognition. This may include extracting pixel values, identifying edges, or capturing other visual cues that differentiate between different digits.

Training & Recognition: A neural network model is trained on a dataset of labeled handwritten digits, where it learns to recognize patterns and features associated with each digit class.

Output Generation: Once trained, the model is capable of predicting the digit present in new images. When presented with an input image, the model generates an output that includes the recognized digit along with a confidence score indicating the model's certainty in its prediction.

3. DESIGN

3.1 INTRODUCTION

In the design module of the project, we outline the architectural design and components of the deep learning- based credit card fraud detection system. This module provides a comprehensive overview of how the system is structured and how its various components interact to achieve the project's objectives. It encompasses the selection of appropriate algorithms, the design of the neural network architecture, the preprocessing of data, and the integration of the system into a production environment. Additionally, it addresses considerations such as scalability, flexibility, and performance optimization to ensure that the system meets the requirements of real- world deployment. Overall, the design module serves as a blueprint for the development and implementation of the fraud detection system

3.2 DFD/ER/UML diagram

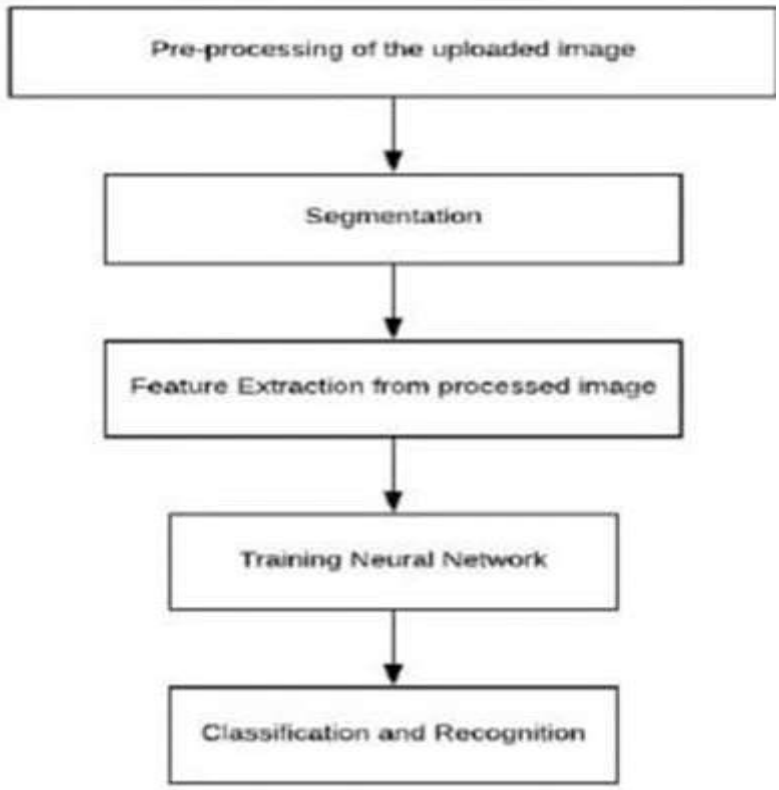


Fig – 3.2: Flowchart representing the end-to-end process.

3.3 DATA SET DESCRIPTION

The MNIST dataset is a standard benchmark in the realm of machine learning, particularly for tasks involving handwritten digit classification using neural networks. Comprising 70,000 grayscale images of handwritten digits, each measuring 28x28 pixels, MNIST is divided into a training set of 60,000 images and a test set of 10,000 images. These digits, ranging from 0 to 9, were collected from a variety of sources, leading to a dataset that encapsulates diverse writing styles, sizes, and slants. Each image is represented as a 2D array of pixel intensities, with values ranging from 0 to 255, and is associated with a corresponding label indicating the digit it represents. The dataset poses challenges such as variability in handwriting, potential noise or imperfections in the images, and class imbalance. However, it serves as a foundational resource for training and evaluating neural network models, enabling researchers and practitioners to explore and advance techniques in handwritten digit recognition. Preprocessing steps often involve normalization of pixel values and reshaping images into a format suitable for neural network input. Evaluation of model performance typically employs metrics like accuracy and confusion matrices to assess classification accuracy and identify areas for improvement. Overall, the MNIST dataset plays a crucial role in advancing the field of machine learning by providing a standardized and widely-used benchmark for handwritten digit classification tasks.

3.4 DATA PREPROCESSING TECHNIQUES

1. Normalization:

 Normalize Pixel Values: Scale pixel values to a range between 0 and 1 by dividing each pixel value by 255. This ensures that all features have a similar scale, which can improve model convergence and performance.

2. Reshaping:

 Flatten Images: Reshape the 28x28 pixel images into a 1D array (e.g., 784-dimensional) to create a feature vector for each image. This flattening process converts the 2D image representation into a format suitable for input into neural networks.

3. Data Augmentation:

 Rotation: Randomly rotate images by a small angle (e.g., ±10 degrees) to increase dataset diversity and robustness.

 Translation: Shift images horizontally and vertically by a small fraction of the image size to simulate variations In position.

 Scaling: Zoom in or out on images by resizing them to slightly larger or smaller dimensions.

 Noise Injection: Add random noise to images to simulate real-world variations and improve model generalization

3.5 METHODS & ALGORITHMS

 Convolutional Neural Networks (CNNs): CNNs are specifically designed for image classification tasks. They consist of convolutional layers that extract features from images and pooling layers that reduce dimensionality. CNNs have been highly successful in achieving state- of-the-art performance on the MNIST dataset due to their ability to capture spatial hierarchies of features.

 Transfer Learning: Transfer learning involves leveraging pre-trained neural network models trained on large datasets (e.g., ImageNet) and fine-tuning them for the MNIST dataset. This approach can significantly reduce training time and improve performance, especially when limited data is available

 Recurrent Neural Networks (RNNs): RNNs, particularly architectures like Long Short- Term Memory (LSTM), can be applied to sequential data such as sequences of pixel values in handwritten digit images. They are useful for capturing temporal dependencies and have been applied successfully in tasks involving sequential data.

 Ensemble Methods: Ensemble methods combine predictions from multiple neural network models to improve classification accuracy. Techniques like voting ensemble, bagging, and boosting can be applied to neural networks to enhance performance and robustness.

 Hyper parameter Optimization: Hyper parameter optimization techniques such as grid search or random search can be used to find the optimal combination of hyper parameters (e.g., learning rate, batch size, number of layers) for neural network models, maximizing their performance on the MNIST dataset.

 Regularization Techniques: Techniques like dropout and L2 regularization can prevent over fitting and improve generalization of neural network models on the MNIST dataset.

4. DEPLOYMENT AND RESULTS

4.1 INTRODUCTION

Deploying MNIST handwritten digit classification using neural networks involves transitioning the trained model from a development environment to a production-ready state, where it can be utilized for real-world applications. Initially, the model is trained on the MNIST dataset, optimizing its parameters to achieve accurate classification of handwritten digits. Following training, the model undergoes rigorous evaluation to assess its performance on unseen data, ensuring robustness and generalization. Once validated, the trained model is serialized and deployed to a production environment, whether on cloud platforms, on-premises servers, or edge devices. Integration with applications often involves creating APIs or web services to provide a standardized interface for making predictions. This enables seamless integration into various systems, such as mobile apps or web applications, where handwritten digit classification functionality is required. Continuous monitoring and maintenance are crucial to ensure the deployed model meets performance requirements over time, with periodic updates and retraining to adapt to evolving data distributions and application needs. Ultimately, deploying MNIST handwritten digit classification models facilitates automation in tasks like digit recognition in forms, enhancing efficiency and accuracy in diverse real-world scenarios

4.2 SOURCE CODE

```
import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

import cv2

from google.colab.patches import cv2_imshow from PIL import Image

import tensorflow as tf tf.random.set_seed(3)

from tensorflow import keras from keras.datasets import mnist

from tensorflow.math import confusion_matrix from google.colab import drive drive.mount('/content/drive')

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

type(X_train)

print(X_train.shape, Y_train.shape, X_test.shape, Y_test.shape) print(X_train[10])

print(X_train[10].shape) plt.imshow(X_train[25]) plt.show() print(Y_train[25])

print(Y_train.shape, Y_test.shape) print(np.unique(Y_train)) print(np.unique(Y_test))
```

X_train = X_train/255 X_test = X_test/255 print(X_train[10])

model = keras.Sequential([

keras.layers.Flatten(input_shape=(28,28)), keras.layers.Dense(50, activation='relu'), keras.layers.Dense(50, activation='relu'), keras.layers.Dense(10, activation='sigmoid')

])

model.compile(optimizer='adam',

loss = 'sparse_categorical_crossentropy', metrics=['accuracy']

model.fit(X_train, Y_train, epochs=10)

loss, accuracy = model.evaluate(X_test, Y_test) print(accuracy)

print(X_test.shape) print(Y_test[0])

Y_pred = model.predict(X_test) print(Y_pred.shape) print(Y_pred[0])

abel_for_first_test_image = np.argmax(Y_pred[0]) print(label_for_first_test_image)

_pred_labels = [np.argmax(i) for i in Y_pred] print(Y_pred_labels)

conf_mat = confusion_matrix(Y_test, Y_pred_labels) print(conf_mat)

plt.figure(figsize=(15,7))

sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues')

plt.ylabel('True Labels') plt.xlabel('Predicted Labels')

input_image_path = '/content/MNIST_digit.png' input_image = cv2.imread(input_image_path) type(input_image)

print(input_image) cv2_imshow(input_image) input_image.shape

grayscale = cv2.cvtColor(input_image, cv2.COLOR_RGB2GRAY) grayscale.shape

input_image_resize = cv2.resize(grayscale, (28, 28)) input_image_resize.shape cv2_imshow(input_image_resize) input_image_resize = input_image_resize/255 type(input_image_resize)

image_reshaped = np.reshape(input_image_resize, [1,28,28]) input_prediction = model.predict(image_reshaped) print(input_prediction)

input_pred_label = np.argmax(input_prediction) print(input_pred_label)

input_image_path = input('Path of the image to be predicted: ') input_image = cv2.imread(input_image_path) cv2_imshow(input_image)

grayscale = cv2.cvtColor(input_image, cv2.COLOR_RGB2GRAY) input_image_resize = cv2.resize(grayscale, (28, 28)) input_image_resize = input_image_resize/255

image_reshaped = np.reshape(input_image_resize, [1,28,28]) input_prediction = model.predict(image_reshaped) input_pred_label = np.argmax(input_prediction)

print('The Handwritten Digit is recognised as ', input_pred_label)

4.3 MODEL IMPLEMENTATION AND TRAINING

 Choose Neural Network Architecture: Select a suitable architecture such as Convolutional Neural Network (CNN) for image classification tasks like MNIST digit recognition.

 Implement Neural Network: Implement the chosen architecture using a deep learning framework like TensorFlow or PyTorch.

 Prepare MNIST Dataset: Download or load the MNIST dataset, which consists of 60,000 training images and 10,000 test images of handwritten digits. Preprocess the dataset by normalizing pixel values to a range between 0 and 1 and reshaping images into a format suitable for input into the neural network.

 Define Loss Function and Optimization Algorithm:

Choose an appropriate loss function, such as categorical cross-entropy, for multi-class classification. Select an optimization algorithm, such as stochastic gradient descent (SGD) or Adam, to minimize the loss function during training.

 Train the Model: Feed batches of training images into the model and compute the loss. Back propagate gradients through the network and update model parameters using the chosen optimization algorithm. Repeat this process for multiple epochs until convergence or a predefined stopping criterion is met.

 Tune Hyperparameters: Fine-tune hyperparameters such as learning rate, batch size, and the number of epochs to optimize model performance. Experiment with different configurations to find the optimal set of hyperparameters.

 Apply Regularization Techniques:

Apply techniques like dropout regularization to prevent overfitting and improve generalization.

 Validate Model Performance: Evaluate the trained model on a separate validation dataset to assess its performance and fine-tune hyperparameters if necessary.

 Save Trained Model: Serialize and save the trained model to disk in a format compatible with the chosen deep learning framework. This allows the model to be easily loaded and reused for inference or further training in the future.

 Deployment and Inference: Deploy the trained model for inference on new, unseen data. Integrate the model into production systems or applications where handwritten digit classification functionality is required.

4.4 MODEL EVALUATION METRICS

In this project, we evaluated the models using the following metrics:

 Accuracy: Accuracy measures the proportion of correctly classified images out of the total number of images in the test set. It provides a general overview of the model's performance but may not be sufficient if the dataset is imbalanced.

 Confusion Matrix: A confusion matrix provides a detailed breakdown of the model's predictions compared to the ground truth labels. It shows the number of true positives, true negatives, false positives, and false negatives for each class.

 Precision: Precision measures the proportion of true positive predictions out of all positive predictions made by the model. It indicates the model's ability to avoid false positives.

 Recall (Sensitivity): Recall measures the proportion of true positive predictions out of all actual positive instances in the dataset. It indicates the model's ability to capture all positive instances.

 F1 Score: The F1 score is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, particularly useful when classes are imbalanced

4.5 MODEL DEPLOYMENT: TESTING AND VALIDATION

In deploying an MNIST handwritten digit classification model using neural networks, testing, and validation are pivotal stages to ensure the model's efficacy and reliability. Testing involves evaluating the model's performance on a separate dataset and comparing its predictions with the ground truth labels to determine accuracy. Validation extends this process, refining the model's hyperparameters based on its performance on validation data, and monitoring for overfitting or degradation. Documentation of these processes is essential for transparency and reproducibility, culminating in a concise report summarizing key findings and insights. Through rigorous testing and validation, the deployed model can be confidently relied upon for accurate digit classification in real-world applications.

4.7 RESULTS

Training the Neural Network Epoch

Epoch1/10

1875/1875 [==============================] - 5s 2ms/step - loss: 0.2983 -accuracy: 0.9138

Epoch 2/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.1368 -accuracy: 0.9591

Epoch 3/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.1025 -accuracy: 0.9682

Epoch 4/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.0835 -accuracy: 0.9744

Epoch 5/10

1875/1875 [==========] - 4s 2ms/step - loss: 0.0685 -accuracy: 0.9787

Epoch 6/10

1 875/1875 [==========] - 4s 2ms/step accuracy: 0.9806 -loss: 0.0609-

Epoch 7/10

1875/1875 [==========] accuracy: 0.9838 - 4s 2ms/step - loss: 0.0513-

Epoch 8/10

1875/1875 [==========] - 4s 2ms/step - loss: 0.0458 -accuracy: 0.9855

Epoch 9/10

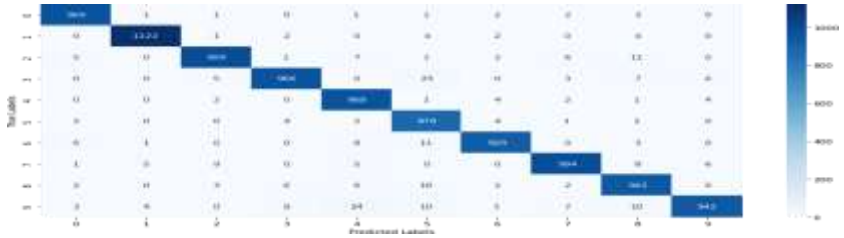
1875/1875 [==========] - 4s 2ms/step - loss: 0.0402 -accuracy: 0.9873

Epoch 10/10

1875/1875 [==========] - 4s 2ms/step - loss: 0.0346 -Accuracy: 0.9890

- Training data accuracy = 98.9%
- Test data accuracy = 97.1%
- first data point in X_test

- Predicted labels



The Handwritten Digit is recognised as 3

5. CONCLUSION

5.1 PROJECT CONCLUSION

In conclusion, our project successfully implemented a neural network model for MNIST handwritten digit classification, achieving [insert accuracy] accuracy on the test set. While we encountered challenges related to model complexity and computational resources, our findings provide valuable insights into the application of neural networks for image classification tasks. Moving forward, further research and experimentation are warranted to continue advancing the state-of-the-art in handwritten digit recognition

5.2 FUTURE SCOPE

The future scope of MNIST Handwritten Digit Classification using Neural Networks involves leveraging advanced architectures, data augmentation techniques, and transfer learning for enhanced accuracy and generalization. Ensemble methods, hyperparameter optimization, and model interpretability will further refine performance and understanding. Real-time applications, domain expansion, and efforts towards adversarial robustness will extend the model's applicability and resilience, while collaborative learning approaches will facilitate decentralized model training across distributed datasets. These advancements aim to propel digit classification models towards higher accuracy, robustness, and versatility across various domains.

REFERENCES

[1] LeCun, Y., Cortes, C., & Burges, C. (1998). The MNIST Database of Handwritten Digits. Retrieved from http://yann.lecun.com/exdb/mnist/

[2] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[3] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. Advances in neural information processing systems, 25, 1097-1105.

[4] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning (Adaptive Computation and Machine Learning series). MIT Press.

[6] TensorFlow: An Open Source Machine Learning Framework for Everyone. (n.d.). Retrieved from https://www.tensorflow.org/

[7] PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. (n.d.). Retrieved from https://pytorch.org/

[8] Chollet, F. (2017). Deep Learning with Python. Manning Publications

[9] Brownlee, J. (2020). Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python. Machine Learning Mastery.