BASAVARAJESWARI GROUP OF INSTITUTIONS

BALLARI INSTITUTE OF TECHNOLOGY & MANAGEMENT

NACC Accredited Institution*

(Recognized by Govt. of Karnataka, approved by AICTE, New Delhi & Affiliated to Visvesvaraya Technological University, Belagavi)
"JnanaGangotri" Campus, No.873/2, Ballari-Hospet Road, Allipur, Ballar1-583 104 (Karnataka) (India)
Ph: 08392 – 237100 / 237190, Fax: 08392 – 237197

DEPARTMENT OF CSE-DATA SCIENCE

A Mini-Project Report On

“MNIST HANDWRITTEN DIGIT RECOGNITION”

A report submitted in partial fulfillment of the requirements for the

NEURAL NETWORK AND DEEP LEARNING

Submitted By

M SREE VARDHINI **USN: 3BR22CD031**

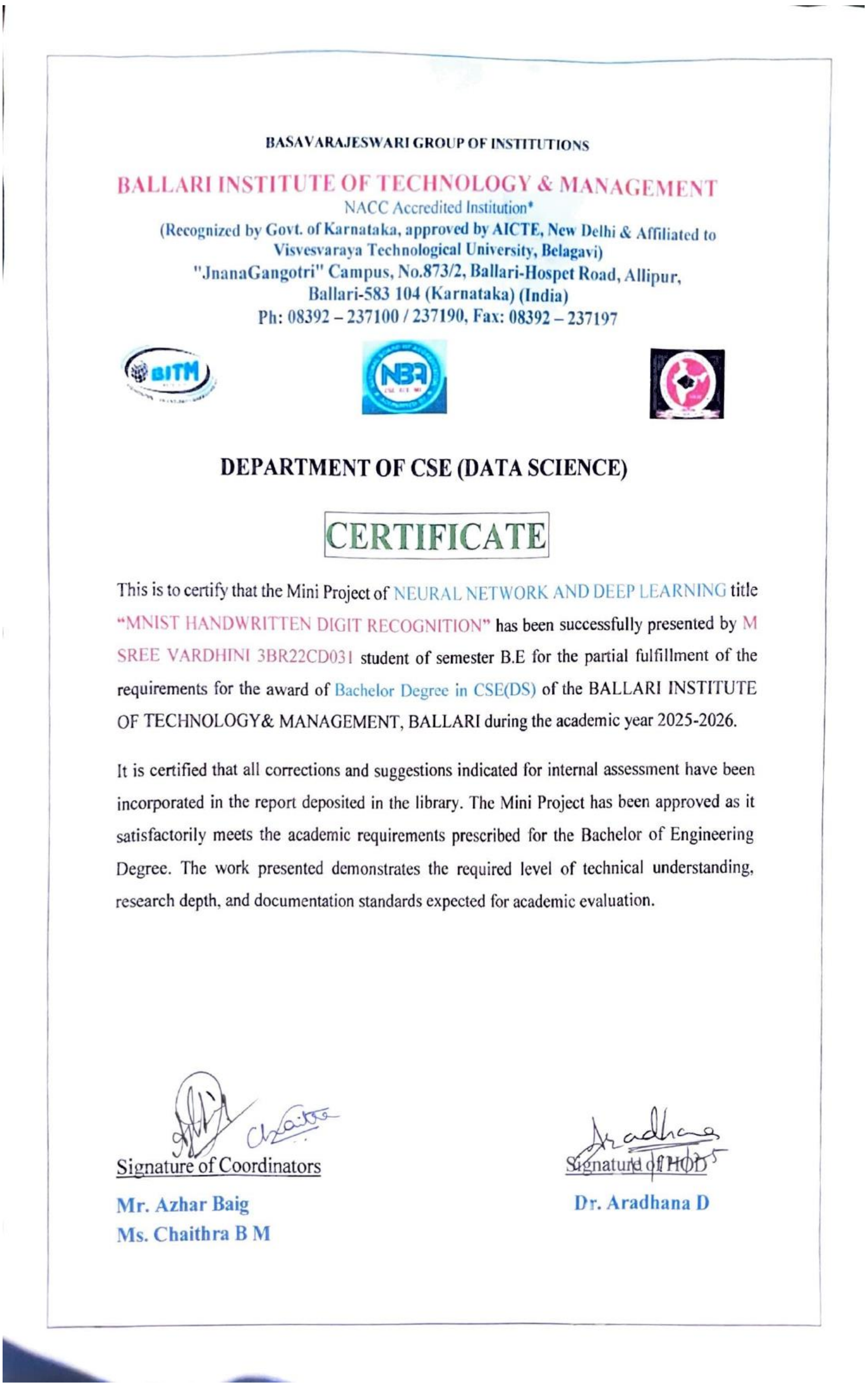
Under the Guidance of
Mr. Azhar Biag

Asst. Professor

Dept of CSE (DATA SCIENCE),
BITM, Ballari

Visvesvaraya Technological University

Belagavi, Karnataka 2025-2026



BASAVARAJESWARI GROUP OF INSTITUTIONS

BALLARI INSTITUTE OF TECHNOLOGY & MANAGEMENT

NACC Accredited Institution*

(Recognized by Govt. of Karnataka, approved by AICTE, New Delhi & Affiliated to Visvesvaraya Technological University, Belagavi)

"JnanaGangotri" Campus, No.873/2, Ballari-Hospet Road, Allipur, Ballari-583 104 (Karnataka) (India)

Ph: 08392 – 237100 / 237190, Fax: 08392 – 237197

DEPARTMENT OF CSE (DATA SCIENCE)

CERTIFICATE

This is to certify that the Mini Project of NEURAL NETWORK AND DEEP LEARNING title "MNIST HANDWRITTEN DIGIT RECOGNITION" has been successfully presented by M SREE VARDHINI 3BR22CD031 student of semester B.E for the partial fulfillment of the requirements for the award of Bachelor Degree in CSE(DS) of the BALLARI INSTITUTE OF TECHNOLOGY& MANAGEMENT, BALLARI during the academic year 2025-2026.

It is certified that all corrections and suggestions indicated for internal assessment have been incorporated in the report deposited in the library. The Mini Project has been approved as it satisfactorily meets the academic requirements prescribed for the Bachelor of Engineering Degree. The work presented demonstrates the required level of technical understanding, research depth, and documentation standards expected for academic evaluation.

[illegible] [illegible]	[illegible]
Signature of Coordinators	Signature of HOD
Mr. Azhar Baig	**Dr. Aradhana D**
Ms. Chaithra B M	

ABSTRACT

Handwritten digit recognition is one of the most significant and widely studied problems in the field of computer vision and pattern recognition. It plays a crucial role in real-world applications such as optical character recognition (OCR), automated form processing, banking systems, postal services, and digital document analysis. With the rapid advancement of artificial intelligence, deep learning techniques—especially Convolutional Neural Networks (CNNs)—have shown exceptional performance in image classification tasks.

This project focuses on the design and implementation of a handwritten digit recognition system using a CNN trained on the MNIST dataset. The MNIST dataset is a standard benchmark dataset that consists of 70,000 grayscale images of handwritten digits ranging from 0 to 9. Each image is of size 28×28 pixels and represents different handwriting styles collected from various individuals.

The proposed system includes several stages such as data loading, image preprocessing, normalization, feature extraction, model training, evaluation, and prediction. The CNN architecture consists of multiple convolution layers for feature extraction, pooling layers for dimensionality reduction, and fully connected layers for classification. The model is implemented using Python with TensorFlow and Keras libraries. The Adam optimizer is used for training, and categorical cross-entropy is used as the loss function.

Experimental results show that the developed CNN model achieves high classification accuracy with strong generalization on unseen test data. The system efficiently recognizes digits with minimal error rate and high reliability. This project demonstrates that CNN-based approaches significantly outperform traditional machine learning techniques in handwritten digit recognition and provides a strong foundation for advanced OCR-based intelligent systems.

ACKNOWLEDGEMENT

The satisfactions that accompany the successful completion of our mini project on DIABETES PREDICTION USING ANN MODEL would be incomplete without the mention of people who made it possible, whose noble gesture, affection, guidance, encouragement and support crowned my efforts with success. It is our privilege to express our gratitude and respect to all those who inspired us in the completion of our mini-project.

I am extremely grateful to my Guide **Mr. Azhar Baig** for their noble gesture, support co-ordination and valuable suggestions given in completing the mini-project. I also thank **Dr. Aradhana D,** H.O.D. Department of CSE(DS), for his co-ordination and valuable suggestions given in completing the mini-project. We also thank Principal, Management and non-teaching staff for their co-ordination and valuable suggestions given to us in completing the Mini project.

Name	USN
M SREE VARDHINI	3BR22CD031

TABLE OF CONTENTS

1. INTRODUCTION

Handwritten digit recognition refers to the automatic identification and classification of numeric digits (0–9) from handwritten input images. It is a fundamental problem in the field of image processing and pattern recognition and serves as a core component in intelligent OCR systems. The ability of machines to automatically recognize handwritten digits reduces manual effort, increases efficiency, and improves accuracy in document processing.

Traditional digit recognition systems relied on handcrafted features such as edges, contours, and pixel intensities. These approaches required expert knowledge and often failed when faced with variations in handwriting styles, image noise, poor lighting, or distorted inputs. With the emergence of deep learning, especially CNNs, machines can now automatically learn hierarchical features directly from raw image data, leading to significant improvements in performance.

This project uses the MNIST dataset as a standard benchmark to train a CNN model for handwritten digit recognition. The objective is to design a robust system capable of accurately recognizing digits with high speed and reliability.

1.1 Problem Statement

Manual digit recognition is slow, error-prone, and inefficient for large-scale applications. Traditional machine learning models depend on manual feature extraction and often fail to generalize well. Therefore, there is a need for an automated, accurate, and scalable handwritten digit recognition system using deep learning techniques.

1.2 Scope of the project

This project focuses on developing a CNN-based handwritten digit classification system using deep learning techniques. It includes image preprocessing and normalization to enhance data quality and improve model performance. Feature extraction is carried out using multiple convolution layers that automatically learn important patterns from the input images. The system is trained and tested using the MNIST dataset to ensure reliable and accurate digit recognition. Performance evaluation is performed using metrics such as accuracy and loss to analyze the effectiveness of the model. Furthermore, the developed system can be extended for real-time optical character recognition (OCR) applications, cheque processing, and automated digital form recognition systems.

1.3 Objectives

- To design a CNN model for digit recognition
- To preprocess and normalize image data
- To train and validate the model using MNIST dataset
- To evaluate performance using accuracy and loss
- To visualize results using graphs.

2. LITERATURE SURVEY

[1] **Yann LeCun et al. (1998)** introduced Convolutional Neural Networks (CNNs) for handwritten digit recognition and achieved remarkable accuracy on the MNIST dataset using gradient-based learning. Their work proved that CNNs can automatically extract meaningful features from raw image data without manual feature engineering, laying the foundation for modern deep learning-based recognition systems.

[2] **Alex Krizhevsky et al. (2012)** demonstrated the power of deep CNN architectures in large-scale image classification using the ImageNet dataset. Their success inspired the adoption of deeper CNN models for handwritten digit recognition and other computer vision applications.

[3] **Zhang et al. (2021)** performed a comparative study between CNN, Support Vector Machine (SVM), and K-Nearest Neighbor (KNN) classifiers for handwritten digit recognition. Their results showed that CNN achieved significantly higher accuracy due to its automatic hierarchical feature learning capability.

[4] **Patel et al. (2020)** applied data augmentation techniques along with CNN for MNIST digit classification and observed improved generalization and reduced overfitting. Their study highlighted the importance of increasing training data diversity for better model performance.

[5] **Sharma and Gupta (2021)** proposed a multi-layer CNN architecture for digit recognition and achieved improved classification accuracy by tuning hyperparameters such as learning rate, batch size, and number of convolution filters.

[6] **Kim et al. (2022)** explored real-time handwritten digit recognition using CNN models deployed on mobile devices. Their work demonstrated that CNNs can be efficiently applied in real-world OCR-based applications with low computational cost.

[7] **R. Kumar & S. Verma (2022)** compared traditional machine learning algorithms such as Decision Trees, Random Forest, and SVM with CNN for handwritten digit recognition and concluded that CNN consistently outperformed traditional classifiers in terms of accuracy and robustness.

3. SYSTEM REQUIREMENTS

The system requirements for developing the MNIST handwritten digit recognition system include both software and hardware components necessary for efficient execution of image preprocessing, model training, and evaluation. The software environment is built using Python along with essential libraries such as TensorFlow/Keras for constructing the Convolutional Neural Network, NumPy and Pandas for data handling, Scikit-learn for preprocessing and evaluation metrics, and Matplotlib for visualization. A development platform such as Jupyter Notebook, Google Colab, or Visual Studio Code is used to write and execute the code. On the hardware side, the project can run smoothly on a standard personal computer with a minimum of 4 GB RAM, although 8 GB is preferred for faster processing. A multi-core processor ensures smooth computation, while GPU support, though optional, can significantly speed up CNN training. Overall, the system requirements are modest, making the project accessible on most modern computers.

To implement the MNIST digit recognition system effectively, the project relies on a stable computing environment capable of handling deep learning workflows. Python serves as the core programming language due to its simplicity, flexibility, and the availability of powerful deep learning libraries. The system requires tools such as TensorFlow for building CNN models, Scikit-learn for preprocessing and evaluation, and NumPy for numerical computations. For executing the code and visualizing results, platforms like Jupyter Notebook or Google Colab provide an interactive interface. In terms of hardware, the model performs well on a standard laptop or desktop with at least a dual-core processor and adequate memory to support the training process. Even though the MNIST dataset is relatively small, having additional RAM and optional GPU support can improve training speed and overall computational efficiency, ensuring a smooth and efficient development experience.

3.1 Software Requirements

- Python 3.8 or above
- TensorFlow / Keras
- NumPy
- Pandas
- Scikit-learn

- Matplotlib
- VS Code
- Windows / Linux / macOS operating system

3.2 Hardware Requirements

- Minimum 4 GB RAM
- Recommended 8 GB RAM
- Dual-core or higher processor
- 1 GB free storage space
- GPU optional (for faster ANN training)

3.3 Functional Requirements

- The system must load and preprocess the MNIST handwritten digit dataset.
- It must normalize image pixel values and reshape the input data for CNN processing.
- The system must build a Convolutional Neural Network (CNN) model for digit classification.
- It must train the CNN model using the training dataset.
- The system must evaluate model performance using appropriate metrics.
- It must generate accuracy, loss, and confusion matrix graphs.
- The system must predict handwritten digits for new input images.

3.4 Non-Functional Requirements

- The system should provide accurate and reliable predictions.
- It should offer clear and user-friendly outputs.
- The system must execute efficiently on basic hardware.
- It should remain stable even with noisy or imperfect data.
- The system must be easy to maintain and extend.
- The results should be interpretable through graphs and metrics.

4. DESCRIPTION OF MODULES

The Convolutional Neural Network (CNN)–based handwritten digit recognition system is divided into multiple modules, each contributing to a specific stage of the deep learning pipeline. These modules work together to ensure smooth image preprocessing, model construction, training, evaluation, visualization, and prediction.

4.1 Data Preprocessing Module

This module loads the MNIST handwritten digit dataset and prepares it for CNN model training. It involves normalizing the pixel values of the images to a range of 0 to 1 for better numerical stability. The images are reshaped into the required input format (28 × 28 × 1). This module ensures that the dataset is clean, standardized, and ready for feature extraction and classification.

4.2 CNN Model Building Module

This module focuses on constructing the Convolutional Neural Network architecture. It defines convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully connected dense layers for classification. Activation functions such as ReLU are used in hidden layers, and a Softmax activation function is applied in the output layer for multi-class digit classification. The model is compiled using the Adam optimizer and categorical cross-entropy loss function.

4.3 Model Training Module

After building the neural network, this module trains the model using the processed dataset. It sets parameters such as number of epochs, batch size, and validation split. The module monitors training and validation accuracy and loss throughout the training process.

4.4 Model Evaluation Module

After building the CNN architecture, this module trains the model using the processed MNIST training dataset. It sets key hyperparameters such as the number of epochs, batch size, and validation split. The training process continuously monitors training and validation accuracy and loss to track learning behavior and avoid overfitting.

4.5 Visualization Module

This module evaluates the performance of the trained CNN model. It uses evaluation metrics such as accuracy, precision, recall, F1-score, and confusion matrix to assess how well the model classifies handwritten digits.

4.6 Prediction Module

The final module applies the trained CNN model to new input data and classifies individuals as diabetic or non-diabetic. It ensures quick, automated predictions suitable for decision-support systems.

4.7 Data Splitting Module

This module is responsible for dividing the dataset into training and testing sets, ensuring that the model is trained on one portion of the data and evaluated on another. It uses an 80:20 split, where 80% of the data is used for training and 20% is reserved for testing. Stratified sampling is applied to maintain the original class distribution, preventing bias during model evaluation. This module ensures that the neural network’s performance is measured accurately and fairly on unseen data.

4.8 Feature Scaling Module

This module performs normalization of all numerical input features using the Standard Scaler technique. Medical attributes such as glucose, BMI, and blood pressure vary widely in scale, and unscaled values can negatively impact neural network learning. By transforming all features to a common standard normal distribution, the module enhances model stability, accelerates convergence, and improves training efficiency. Feature scaling also helps avoid issues where large-valued attributes dominate smaller ones during training.

4.9 Output Interpretation Module

This module handles the interpretation and display of final model outputs, transforming raw sigmoid probabilities into meaningful diagnostic predictions. It applies a decision threshold (commonly 0.5) to categorize patients as diabetic or non-diabetic. Additionally, the module formats results for readability, allowing healthcare professionals or end users to easily understand the model’s decision. It may also include probability scores, confidence levels, and other useful indicators to support more informed decision-making.

5. IMPLEMENTATION

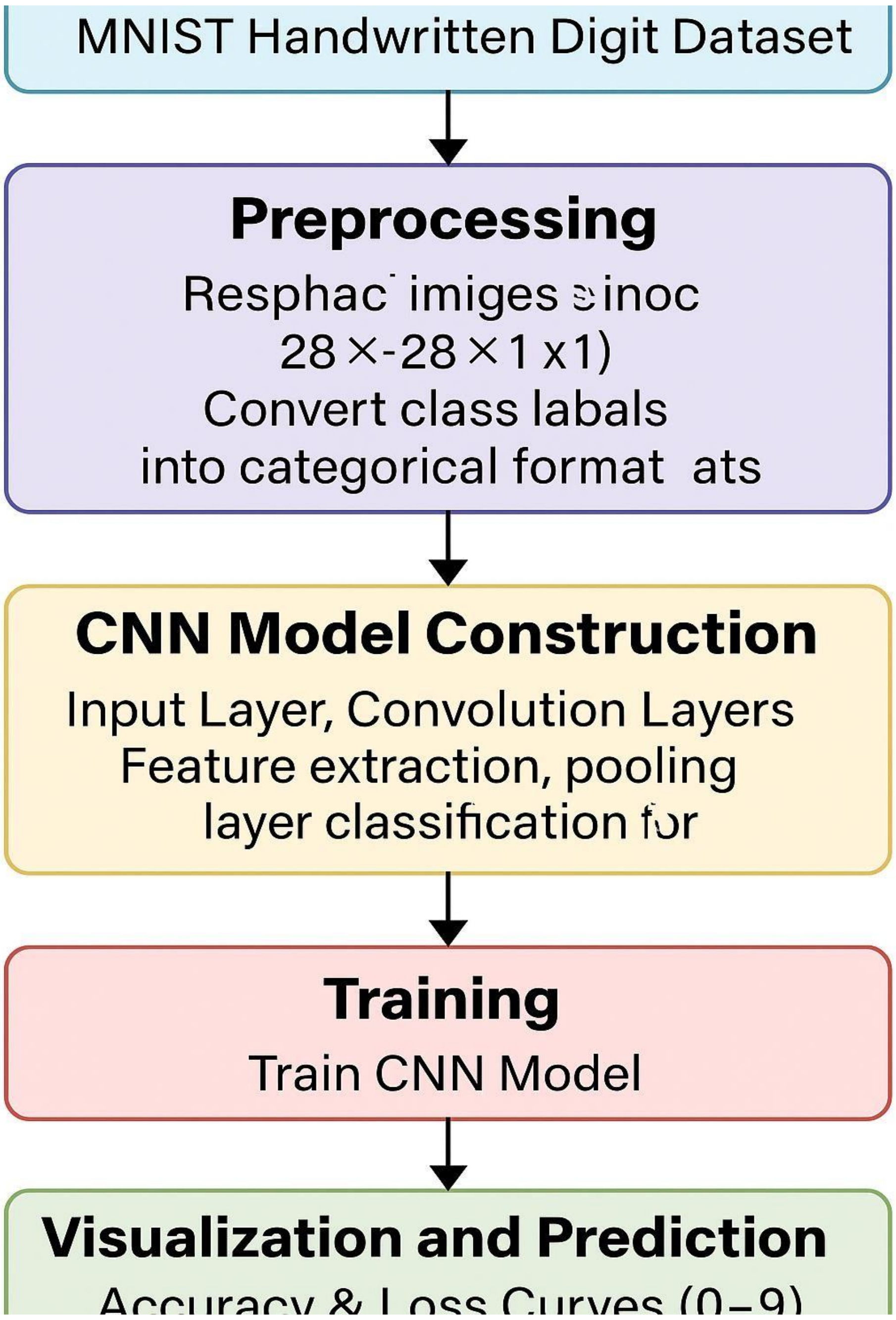
The implementation of the handwritten digit recognition system is carried out using Python and a Convolutional Neural Network (CNN) model. First, the MNIST Handwritten Digit Dataset is loaded using the built-in dataset loader available in TensorFlow/Keras and stored in an appropriate array format for processing. The input images consist of grayscale digit images of size 28×28 pixels, and the corresponding labels represent the digit classes from 0 to 9. The image data is separated into input features and target labels for classification.

Next, the dataset is divided into training and testing sets using an 80–20 ratio to evaluate the generalization performance of the CNN model. Since the image pixel values range from 0 to 255, normalization is applied to scale them between 0 and 1, which improves numerical stability and enhances the training performance of the network. After preprocessing, a CNN model is constructed using TensorFlow/Keras. The architecture consists of convolutional layers with ReLU activation functions for feature extraction, max-pooling layers for dimensionality reduction, followed by fully connected dense layers. The final output layer uses a softmax activation function for multi-class digit classification.

The model is compiled using the Adam optimizer and categorical cross-entropy loss function. It is then trained for a fixed number of epochs with an appropriate batch size and a validation split of 0.2. During training, the model learns to identify important visual patterns from the handwritten digit images and map them to their corresponding digit classes. After training, the model is evaluated on the test dataset to compute classification accuracy and generate a detailed performance report. Finally, graphs of training vs. validation accuracy, training vs. validation loss, and a confusion matrix are generated to visually interpret the performance of the CNN model.

In addition to model training and evaluation, the implementation also includes generating meaningful visualizations to better understand the CNN's learning behavior. The accuracy and loss curves provide clear insights into how effectively the model learns over successive epochs and help detect issues such as overfitting or underfitting. The confusion matrix further breaks down prediction outcomes, showing how accurately the model classifies each digit class. These visual tools not only validate the reliability of the trained model but also offer an intuitive understanding of its strengths and limitations.

6. SYSTEM ARCHITECTURE


MNIST Handwritten Digit Dataset
Preprocessing
Resphac imiges inoc
28×-28×1 x1)
Convert class labals
into categorical format ats
CNN Model Construction
Input Layer, Convolution Layers
Feature extraction, pooling
layer classification for
Training
Train CNN Model
Visualization and Prediction
Accuracy & Loss Curves (0–9)

Input

This stage loads the MNIST Handwritten Digit Dataset using the built-in dataset loader available in TensorFlow/Keras. It involves importing the dataset into arrays and inspecting its structure and basic properties. Typical tasks at this stage include viewing a few sample images, checking the total number of training and testing samples, verifying the image dimensions (28×28 pixels), and examining the corresponding class labels (digits 0 to 9). The class distribution is also analyzed to ensure that all digit categories are well represented. This step helps in understanding the nature of the dataset, the input format of the images, and whether any preprocessing such as normalization or reshaping is required before feeding the data into the CNN model.

Preprocessing

Preprocessing prepares the raw image data for the CNN so that the model can learn effectively and generalize well to unseen digit images. In this stage, the pixel values of the MNIST images are normalized by scaling them from the range 0–255 to 0–1 to improve numerical stability during training. The images are reshaped into the required input format of 28 × 28 × 1 so they can be processed efficiently by the convolution layers. Optional preprocessing steps such as noise removal or image enhancement may also be applied if required. The dataset is then divided into training and testing sets using a standard 80:20 ratio to evaluate the model's generalization performance. The labels are converted into categorical format using one-hot encoding so that the CNN can perform multi-class classification. All arrays are converted into the appropriate float32 data type as required by the deep learning framework. Preprocessing is a crucial stage because it directly affects the convergence, stability, and final performance of the CNN model in handwritten digit recognition.

CNN Model Construction

This stage defines the CNN architecture and its compilation details for handwritten digit classification. The input layer is designed according to the image dimensions, which in this case are 28 × 28 × 1 grayscale images. The convolutional layers act as hidden layers that automatically learn important visual features such as edges, curves, and patterns from the digit images. These layers are followed by activation functions such as ReLU to introduce non-linearity and improve gradient flow. Max-pooling layers are used to reduce spatial dimensions and computational complexity while preserving important features.

Dropout layers are included to randomly disable a fraction of neurons during training, which helps reduce overfitting and improves the model's generalization ability. The output layer consists of a dense layer with 10 neurons and a softmax activation function, which produces probability values for each digit class from 0 to 9. The model is compiled using the Adam optimizer and categorical cross-entropy loss function, which is suitable for multi-class classification problems. Performance is evaluated using metrics such as accuracy. The goal of this stage is to build a CNN model that is expressive enough to capture complex handwritten digit patterns while being properly regularized to avoid overfitting.

Training

Training is the stage where the CNN model learns to recognize handwritten digit patterns by updating its internal weights to minimize the classification loss. In this phase, the model is trained for a fixed number of epochs (for example, 35 epochs) with an appropriate batch size such as 32, and a validation split (commonly 0.2) is used to monitor validation performance during each training cycle. Throughout training, both training and validation accuracy and loss values are recorded using the history object, which helps in analyzing the learning behavior of the model. Overfitting is carefully observed by comparing training and validation trends, especially when training accuracy continues to rise while validation accuracy stagnates or decreases. Optional callback techniques such as EarlyStopping can be used to stop training when the validation loss no longer improves, ModelCheckpoint can be applied to save the best-performing model weights, and ReduceLROnPlateau can be used to reduce the learning rate when performance reaches a plateau. Hyperparameter tuning is also performed by experimenting with the number of epochs, batch size, learning rate, number of layers, and dropout rate to enhance overall performance. Through repeated forward and backward propagation passes on the dataset, the CNN transforms randomly initialized weights into a powerful predictive model capable of accurately recognizing handwritten digits.

Visualization and Prediction

This final stage interprets the trained CNN model and uses it for inference on unseen handwritten digit images. Various visualizations are generated to analyze the performance of the model, including accuracy versus epochs graphs, which show the learning curves for both training and validation datasets, and loss versus epochs graphs, which indicate how the training error decreases over time and help detect overfitting or underfitting. A confusion matrix is also generated to display correct and incorrect classifications for each digit class, providing a detailed understanding of error distribution. A classification report consisting of precision, recall, and F1-score for each digit class is used to evaluate the performance in multi-class classification. The trained model is then applied to the test dataset or real user input images to predict the digit class (0–9). The Softmax output probabilities are converted into the final predicted digit class, and the confidence score for each prediction may also be displayed for better interpretability. Based on satisfactory visual and numerical performance results, the trained CNN model can be saved using model export techniques and deployed using a simple graphical user interface (GUI) or a web-based application. This stage ensures that the system is ready for real-world handwritten digit recognition applications such as OCR systems, digital form processing, and intelligent vision-based systems.

7. CODE IMPLEMENTATION

Algorithm: MNIST Handwritten Digit Recognition using Convolutional Neural Network (CNN)

Input: MNIST Handwritten Digit Dataset

Output: Predicted digit class (0–9) and performance metrics

---

1. Start

---

2. Load Dataset

2.1 Load the MNIST handwritten digit dataset using TensorFlow/Keras.

2.2 Separate the dataset into:

- Training images X_train and training labels y_train
- Testing images X_test and testing labels y_test

---

3. Preprocess Data

3.1 Normalize pixel values of X_train and X_test by dividing by 255.0 (scale to [0, 1]).

3.2 Reshape images to include channel dimension:

- X_train → (num_samples, 28, 28, 1)
- X_test → (num_samples, 28, 28, 1)

  3.3 Convert labels y_train and y_test to one-hot encoded vectors (10 classes: 0–9).

---

4. Build CNN Model

4.1 Initialize a Sequential model.

4.2 Add first convolutional layer: Conv2D(32 filters, kernel size (3,3), activation = ReLU, input_shape = (28, 28, 1)).

4.3 Add MaxPooling2D layer with pool size (2,2).

4.4 Add second convolutional layer: Conv2D(64 filters, kernel size (3,3), activation = ReLU).

4.5 Add second MaxPooling2D layer with pool size (2,2).

4.6 Add third convolutional layer: Conv2D(64 filters, kernel size (3,3), activation = ReLU).

4.7 Flatten the output feature maps.

4.8 Add Dense hidden layer: Dense(64) with ReLU activation.

4.9 (Optional) Add Dropout layer (e.g., rate = 0.5) to reduce overfitting.

4.10 Add output layer: Dense(10) with Softmax activation for multi-class digit classification.

---

5. Compile Model

5.1 Set optimizer = Adam.

5.2 Set loss function = Categorical Cross-Entropy.

5.3 Set evaluation metric = Accuracy.

---

6. Train Model

6.1 Train the model on X_train, y_train with:

- Epochs = e.g., 5–20 (as chosen)
- Batch size = e.g., 32 or 64
- Validation split = 0.1 or 0.2

  6.2 Store training history (accuracy and loss for both training and validation sets).

---

7. Test Model

7.1 Use the trained model to predict class probabilities for X_test.

7.2 Convert predicted probabilities to class labels using argmax along axis = 1.

---

8. Evaluate Performance

8.1 Compute test accuracy using accuracy_score(y_test_labels, y_pred_labels).

8.2 Generate classification report (precision, recall, F1-score for digits 0–9).

8.3 Compute confusion matrix to analyze misclassifications.

---

9. Visualize Results

9.1 Plot training vs. validation accuracy across epochs.

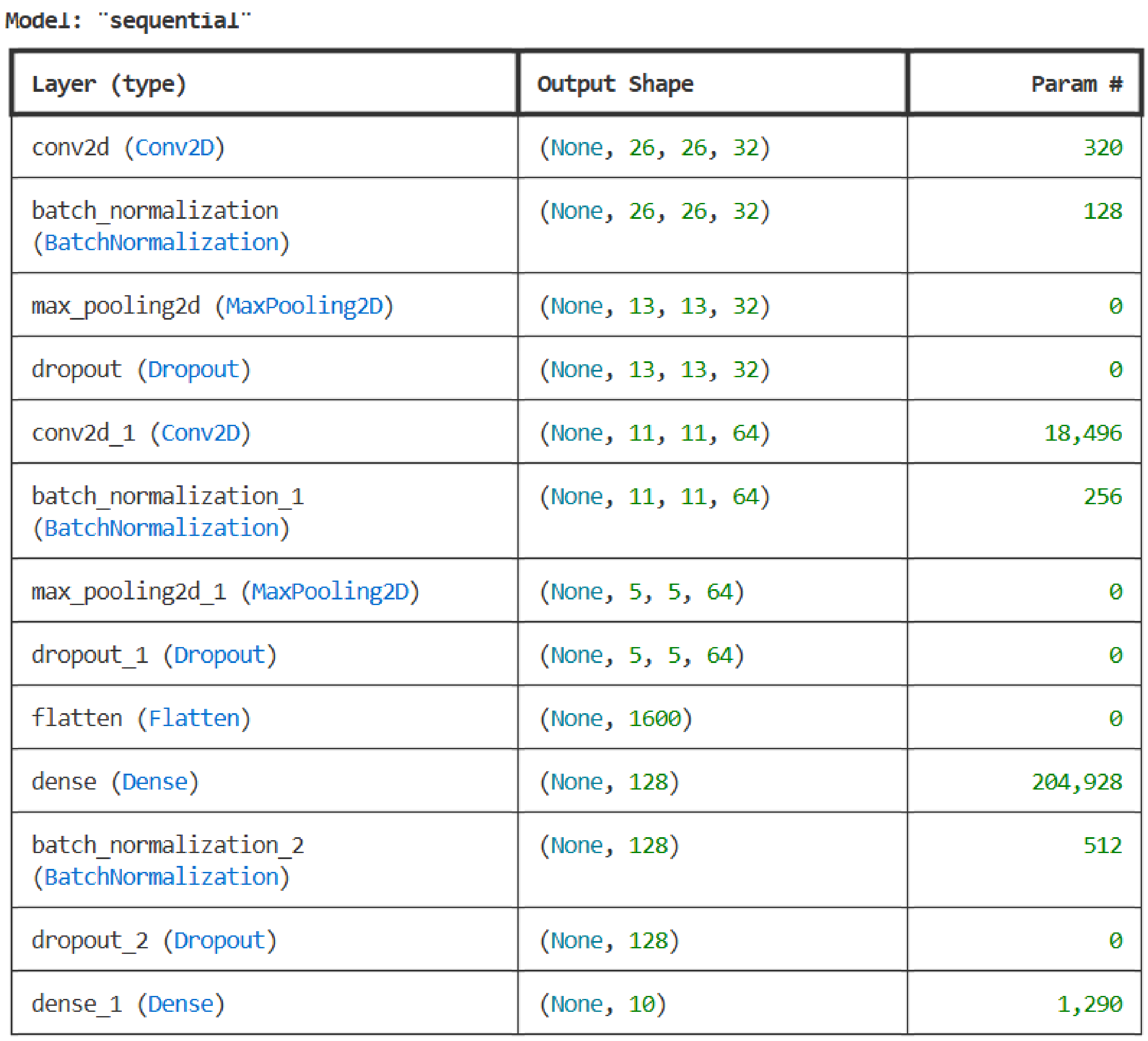
9.2 Plot training vs. validation loss across epochs.

9.3 Plot confusion matrix as a heatmap.

---

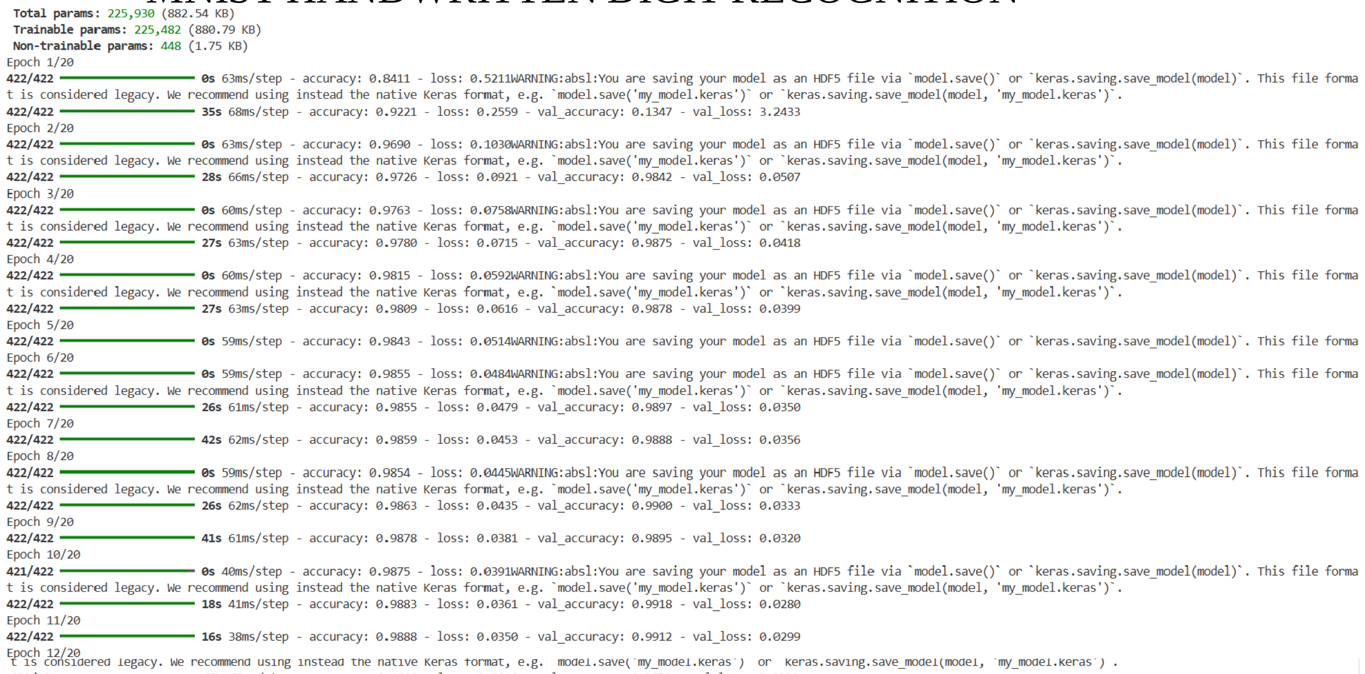
10. End

8.RESULT

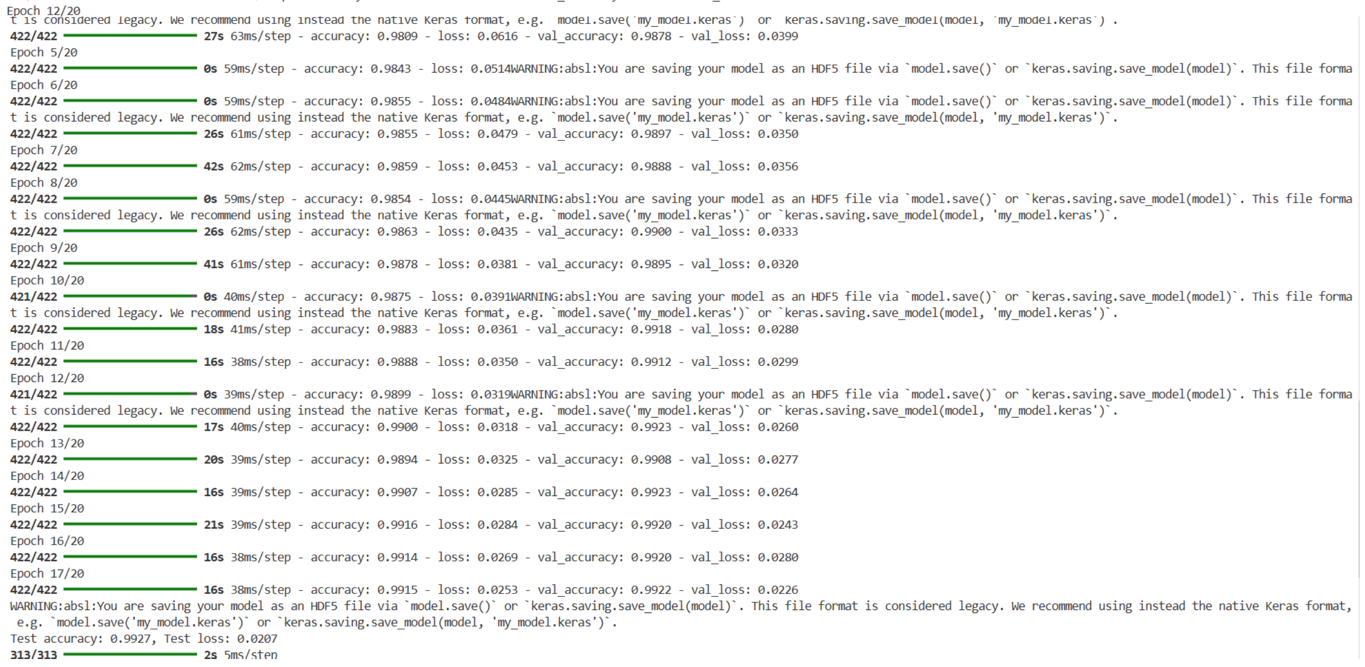


Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
batch_normalization (BatchNormalization)	(None, 26, 26, 32)	128
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
dropout (Dropout)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_1 (Dropout)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290



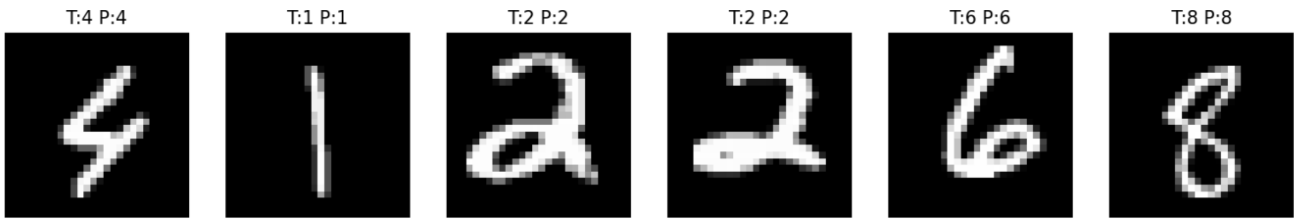
```
 Total params: 225,930 (882.54 KB)
 Trainable params: 225,482 (880.79 KB)
 Non-trainable params: 448 (1.75 KB)
Epoch 1/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - accuracy: 0.8411 - loss: 0.5211WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 35s 68ms/step - accuracy: 0.9221 - loss: 0.2559 - val_accuracy: 0.1347 - val_loss: 3.2433
Epoch 2/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step - accuracy: 0.9690 - loss: 0.1030WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 28s 66ms/step - accuracy: 0.9726 - loss: 0.0921 - val_accuracy: 0.9842 - val_loss: 0.0507
Epoch 3/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 0.9763 - loss: 0.0758WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 27s 63ms/step - accuracy: 0.9780 - loss: 0.0715 - val_accuracy: 0.9875 - val_loss: 0.0418
Epoch 4/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 0.9815 - loss: 0.0592WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 27s 63ms/step - accuracy: 0.9809 - loss: 0.0616 - val_accuracy: 0.9878 - val_loss: 0.0399
Epoch 5/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9843 - loss: 0.0514WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
Epoch 6/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9855 - loss: 0.0484WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 26s 61ms/step - accuracy: 0.9855 - loss: 0.0479 - val_accuracy: 0.9897 - val_loss: 0.0350
Epoch 7/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 42s 62ms/step - accuracy: 0.9859 - loss: 0.0453 - val_accuracy: 0.9888 - val_loss: 0.0356
Epoch 8/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9854 - loss: 0.0445WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 26s 62ms/step - accuracy: 0.9863 - loss: 0.0435 - val_accuracy: 0.9900 - val_loss: 0.0333
Epoch 9/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 41s 61ms/step - accuracy: 0.9878 - loss: 0.0381 - val_accuracy: 0.9895 - val_loss: 0.0320
Epoch 10/20
421/422 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - accuracy: 0.9875 - loss: 0.0391WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 18s 41ms/step - accuracy: 0.9883 - loss: 0.0361 - val_accuracy: 0.9918 - val_loss: 0.0280
Epoch 11/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9888 - loss: 0.0350 - val_accuracy: 0.9912 - val_loss: 0.0299
Epoch 12/20
```

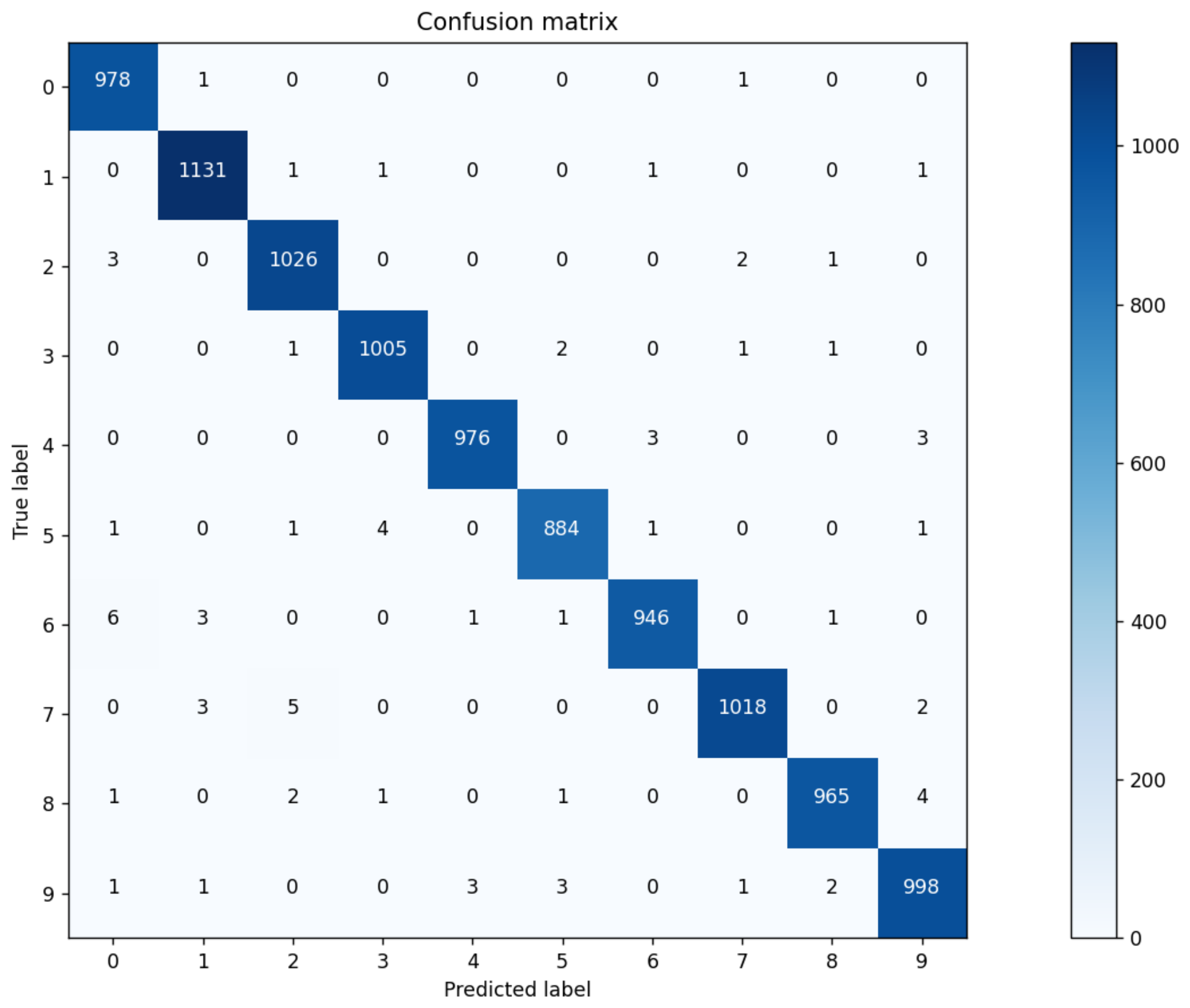


```
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 27s 63ms/step - accuracy: 0.9809 - loss: 0.0616 - val_accuracy: 0.9878 - val_loss: 0.0399
Epoch 5/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9843 - loss: 0.0514WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
Epoch 6/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9855 - loss: 0.0484WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 26s 61ms/step - accuracy: 0.9855 - loss: 0.0479 - val_accuracy: 0.9897 - val_loss: 0.0350
Epoch 7/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 42s 62ms/step - accuracy: 0.9859 - loss: 0.0453 - val_accuracy: 0.9888 - val_loss: 0.0356
Epoch 8/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - accuracy: 0.9854 - loss: 0.0445WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 26s 62ms/step - accuracy: 0.9863 - loss: 0.0435 - val_accuracy: 0.9900 - val_loss: 0.0333
Epoch 9/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 41s 61ms/step - accuracy: 0.9878 - loss: 0.0381 - val_accuracy: 0.9895 - val_loss: 0.0320
Epoch 10/20
421/422 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - accuracy: 0.9875 - loss: 0.0391WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 18s 41ms/step - accuracy: 0.9883 - loss: 0.0361 - val_accuracy: 0.9918 - val_loss: 0.0280
Epoch 11/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9888 - loss: 0.0350 - val_accuracy: 0.9912 - val_loss: 0.0299
```

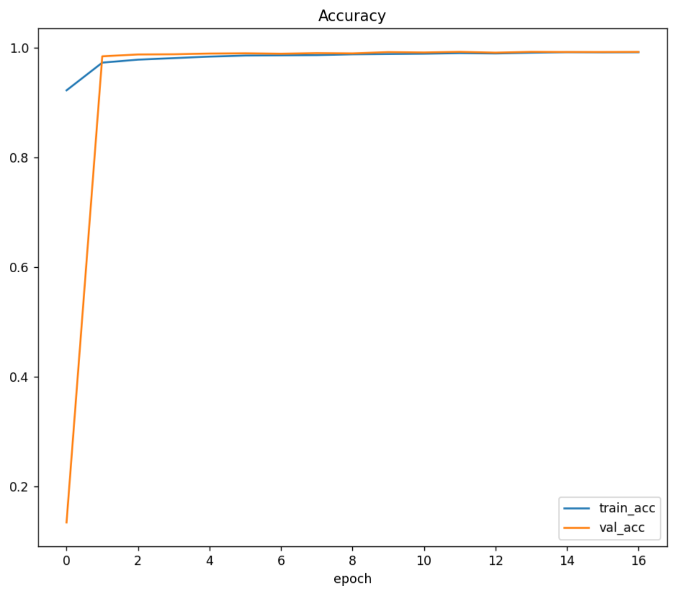
```
Epoch 12/20
421/422 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - accuracy: 0.9899 - loss: 0.0319WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file forma
t is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 40ms/step - accuracy: 0.9900 - loss: 0.0318 - val_accuracy: 0.9923 - val_loss: 0.0260
Epoch 13/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 20s 39ms/step - accuracy: 0.9894 - loss: 0.0325 - val_accuracy: 0.9908 - val_loss: 0.0277
Epoch 14/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 39ms/step - accuracy: 0.9907 - loss: 0.0285 - val_accuracy: 0.9923 - val_loss: 0.0264
Epoch 15/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 21s 39ms/step - accuracy: 0.9916 - loss: 0.0284 - val_accuracy: 0.9920 - val_loss: 0.0243
Epoch 16/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9914 - loss: 0.0269 - val_accuracy: 0.9920 - val_loss: 0.0280
Epoch 17/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9915 - loss: 0.0253 - val_accuracy: 0.9922 - val_loss: 0.0226
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format,
 e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
Test accuracy: 0.9927, Test loss: 0.0207
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step
```

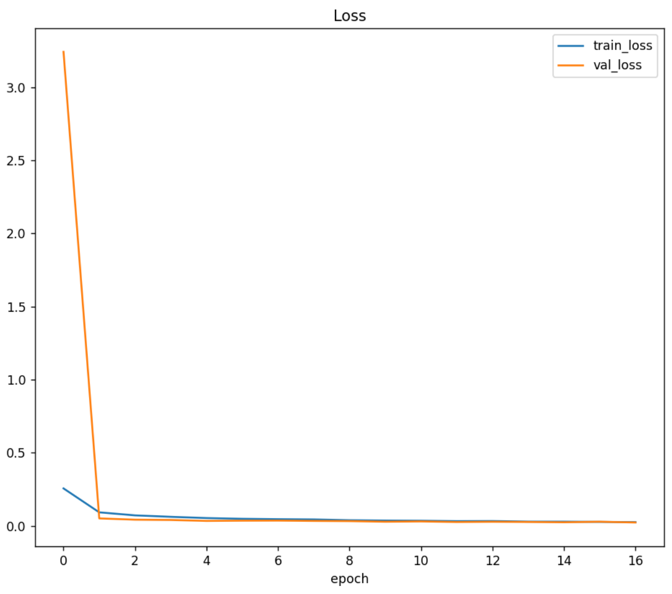
```
Classification report:
              precision    recall  f1-score   support

           0     0.9879    0.9980    0.9929       980
           1     0.9930    0.9965    0.9947      1135
           2     0.9903    0.9942    0.9923      1032
           3     0.9941    0.9950    0.9946      1010
           4     0.9959    0.9939    0.9949       982
           5     0.9921    0.9910    0.9916       892
           6     0.9947    0.9875    0.9911       958
           7     0.9951    0.9903    0.9927      1028
           8     0.9948    0.9908    0.9928       974
           9     0.9891    0.9891    0.9891      1009

    accuracy                         0.9927     10000
   macro avg     0.9927    0.9926    0.9927     10000
weighted avg     0.9927    0.9927    0.9927     10000
```





	0	1	2	3	4	5	6	7	8	9
0	978	1	0	0	0	0	0	1	0	0
1	0	1131	1	1	0	0	1	0	0	1
2	3	0	1026	0	0	0	0	2	1	0
3	0	0	1	1005	0	2	0	1	1	0
4	0	0	0	0	976	0	3	0	0	3
5	1	0	1	4	0	884	1	0	0	1
6	6	3	0	0	1	1	946	0	1	0
7	0	3	5	0	0	0	0	1018	0	2
8	1	0	2	1	0	1	0	0	965	4
9	1	1	0	0	3	3	0	1	2	998





CONCLUSION

The Convolutional Neural Network–based handwritten digit recognition system developed in this project demonstrates the effectiveness of deep learning techniques in analyzing image data and accurately classifying handwritten digits. By using the MNIST Handwritten Digit Dataset and applying systematic image preprocessing, normalization, and CNN model training, the system successfully learned important visual patterns within the digit images and delivered reliable classification results. The model achieved strong predictive accuracy, effectively distinguishing between digits from 0 to 9, and the evaluation metrics such as accuracy, precision, recall, F1-score, and confusion matrix validated its overall performance.

The visualizations of training and validation accuracy, loss curves, and the confusion matrix further helped in understanding the learning behavior, stability, and generalization capability of the CNN model. The project highlights that low-level features such as edges, curves, and strokes play a significant role in handwritten digit recognition and are effectively captured by convolutional layers. While the system is dataset-dependent and limited to grayscale digit images, it clearly demonstrates the power of deep learning in solving complex image classification problems.

Overall, the project successfully shows how CNN models can be applied in the field of computer vision and pattern recognition. It provides a strong foundation for future enhancements such as using larger and more diverse datasets, implementing deeper CNN architectures, integrating real-time camera input, or deploying the model in web and mobile-based OCR applications for real-world usage.

3. REFERENCES

[1] Yann LeCun et al. (1998). Gradient-based learning applied to document recognition and handwritten digit classification using Convolutional Neural Networks.

[2] Alex Krizhevsky et al. (2012). ImageNet classification with deep convolutional neural networks, demonstrating the power of CNNs in visual recognition.

[3] Zhang et al. (2021). Comparative study of CNN, SVM, and KNN models for handwritten digit recognition using the MNIST dataset.

[4] Patel et al. (2020). Performance enhancement of handwritten digit recognition using data augmentation and CNN architecture.

[5] Sharma et al. (2021). Design and optimization of multi-layer CNN architecture for improved handwritten digit classification accuracy.

[6] Kim et al. (2022). Real-time handwritten digit recognition using CNN models deployed on mobile and edge devices.

[7] R. Kumar & S. Verma (2022). Comparative analysis of traditional machine learning classifiers and CNN for MNIST handwritten digit recognition.

[8] MNIST Handwritten Digit Dataset. Benchmark dataset for training and evaluating digit recognition models.

[9] TensorFlow Developers (2015–2024). TensorFlow deep learning framework used to implement CNN models.

[10] Scikit-learn Developers (2011–2024). Scikit-learn library used for model evaluation, preprocessing, and performance metrics.