CNN Implementation for MNIST Digit Recognition

Group Members:

1) Bhagyaban Ghadai - 101159655

2) Sai Kireeti Chalamalasetty- 101164549

3) Charith Kandula- 101164075

4) Praveen Kumar Gera- 101165210

5) Chethan Thubharahalli Ramesh - 101159801

6) Vasu Kumar Battula – 101163969

7) Aravind Mangalagiri - 101160817

Introduction

Recognizing handwritten digits accurately is a challenging task for machines due to the wide variation in handwriting styles and orientations. This project aims to develop a highly accurate handwritten digit recognition system using deep learning, specifically Convolutional Neural Networks (CNNs). CNNs have proven effective for image classification by automatically learning hierarchical feature representations from raw pixel data.

The MNIST dataset, consisting of 70,000 grayscale images of handwritten digits, is used to train and evaluate the CNN model. Despite its simplicity, MNIST presents sufficient complexity to assess machine learning models' performance on this task. This study implements a tailored CNN architecture, employing techniques like convolutional layers, pooling layers, and dropout regularization.

Through rigorous training and k-fold cross-validation, the goal is to achieve state-of-the-art accuracy. Detailed analysis, including learning curves and a confusion matrix, provides insights into the model's performance and potential areas for improvement. Successful handwritten digit recognition has far-reaching applications in automating data entry, processing forms, and developing intelligent document processing systems. This project demonstrates deep learning's power for complex pattern recognition tasks.

Dataset Description

The dataset employed in this project is the MNIST dataset from the UCI Machine Learning Repository, containing 70,000 grayscale images of handwritten digits. These images vary widely in style due to the diversity of the contributors.

Normalization and Reshaping

For effective processing, the original pixel values of the images, ranging from 0 to 255, are normalized to a scale from 0 to 1. This step is essential for enhancing the numerical stability and efficiency of the training process. Additionally, the images are reshaped from a flat array of 784 elements to a 3-dimensional array of 28x28x1, preparing them for input into the CNN.

Dataset Analysis

The MNIST dataset presents several challenges:

Variability in Styles: The diversity in handwriting styles introduces complexity, requiring the CNN to learn highly generalizable features.

Image Quality: The images' low resolution (28x28 pixels) challenges the model's ability to distinguish finer details essential for accurate digit classification.

Balanced Classes: Although relatively balanced, any minor class imbalances could skew the training results, necessitating careful model evaluation and validation.

These factors dictate the need for a robust CNN architecture capable of extracting key features from low-resolution inputs and achieving high accuracy across varied handwriting styles.

Methodology

CNN Model Architecture

The architecture of the CNN employed in this study is specifically designed to address the challenges posed by the MNIST dataset's characteristics. It incorporates multiple layers, each with a distinct role in processing and classifying the images of handwritten digits.

Convolutional Layers

The model utilizes two convolutional layers, each followed by a max-pooling layer. The first convolutional layer applies 32 filters of size 3x3, which helps in capturing basic features like edges and simple textures from the input images. The second convolutional layer expands on this by applying 64 filters of the same size to extract more complex and abstract features. This hierarchical feature extraction is crucial for effectively understanding the varying styles and shapes present in the dataset.

Activation Functions

Each convolutional layer is equipped with the ReLU (Rectified Linear Unit) activation function. ReLU is chosen for its ability to introduce non-linearity into the network, allowing it to learn more complex patterns without significantly increasing the computational burden. Importantly, ReLU helps in mitigating the vanishing gradient problem, which is common in networks with deep architectures**.**

Pooling and Dropout

Following each convolutional layer, a max-pooling layer of size 2x2 is used to reduce the spatial dimensions of the feature maps. This reduction not only decreases the computational load and the model's susceptibility to overfitting but also helps in making the detection of features invariant to scale and orientation.

Dropout layers are strategically placed after pooling layers to further combat overfitting. By randomly dropping a proportion (25% after the first pooling and 50% before the final dense layer) of the neuron connections during training, the model is forced to learn more robust features that are not reliant on any specific set of input features.

Flatten and Dense Layers

The output from the final pooling layer is flattened into a single vector and fed into a dense layer of 128 units. This layer serves to integrate the features extracted by the convolutional and pooling layers into a format suitable for classification. The final layer is a dense layer with 10 units (corresponding to the 10-digit classes) and uses a softmax activation function to output a probability distribution over the classes.

Model Architecture

Libraries and Frameworks Used

TensorFlow/Keras: The primary framework used for building and training the CNN model. Keras provides a high-level interface for neural network layers, optimizers, and utilities.

NumPy: Utilized for numerical operations, particularly for handling arrays and matrix transformations which are crucial for data preprocessing.

Matplotlib and Seaborn: These libraries are used for visualizing data and results, such as plotting training metrics and confusion matrices.

Scikit-learn: Employed for data manipulation tasks and evaluation metrics, including generating confusion matrices and classification reports, as well as for implementing k-fold cross-validation.

CNN Model Architecture

Input Layer: The input layer is designed to receive images reshaped to 28x28x1, representing the height, width, and channel of the grayscale images.

First Convolutional Layer: Consists of 32 filters of size 3x3, using ReLU activation. This layer captures basic features from the images.

First Max Pooling Layer: A 2x2 pooling layer follows, reducing the spatial dimensions of the feature maps to enhance computational efficiency and help in feature abstraction.

Second Convolutional Layer: Increases the complexity of the model with 64 filters of size 3x3, also with ReLU activation. This layer focuses on capturing more complex and abstract features.

Second Max Pooling Layer: Another 2x2 pooling layer follows, further reducing the size of the feature maps and abstracting features.

Dropout Layers: Dropout is implemented at two stages (after each pooling layer) to prevent overfitting. The first dropout rate is 25%, and the second before the dense layer is 50%.

Flatten Layer: This layer converts the 2D feature maps into a 1D vector necessary for the subsequent fully connected layer.

Dense Layer: A dense layer with 128 units and ReLU activation integrates all features.

Output Layer: The final layer is a dense layer with 10 units (one for each digit) using softmax activation to output a probability distribution over the classes.

Training Process

Compilation: The model is compiled using the Adam optimizer, which is known for its efficiency and effectiveness in handling sparse gradients on noisy problems. The loss function used is categorical_crossentropy, suitable for multi-class classification tasks.

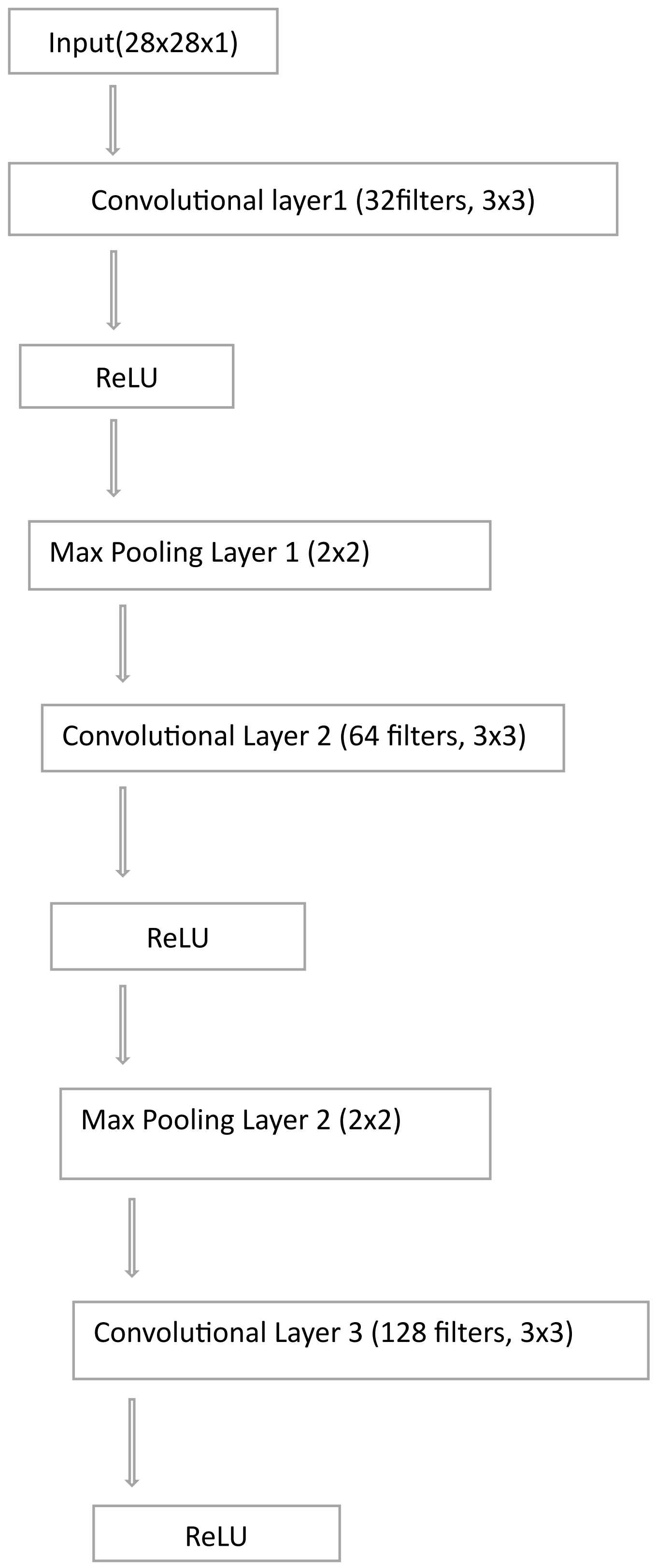
K-Fold Cross-Validation: To validate the model's effectiveness and generalize ability, k-fold cross-validation is employed. The dataset is split into k subsets; the model trains on k-1 subsets and validates on the remaining subset, cycling through all k folds. This approach helps ensure that the model performs well across different parts of the dataset.

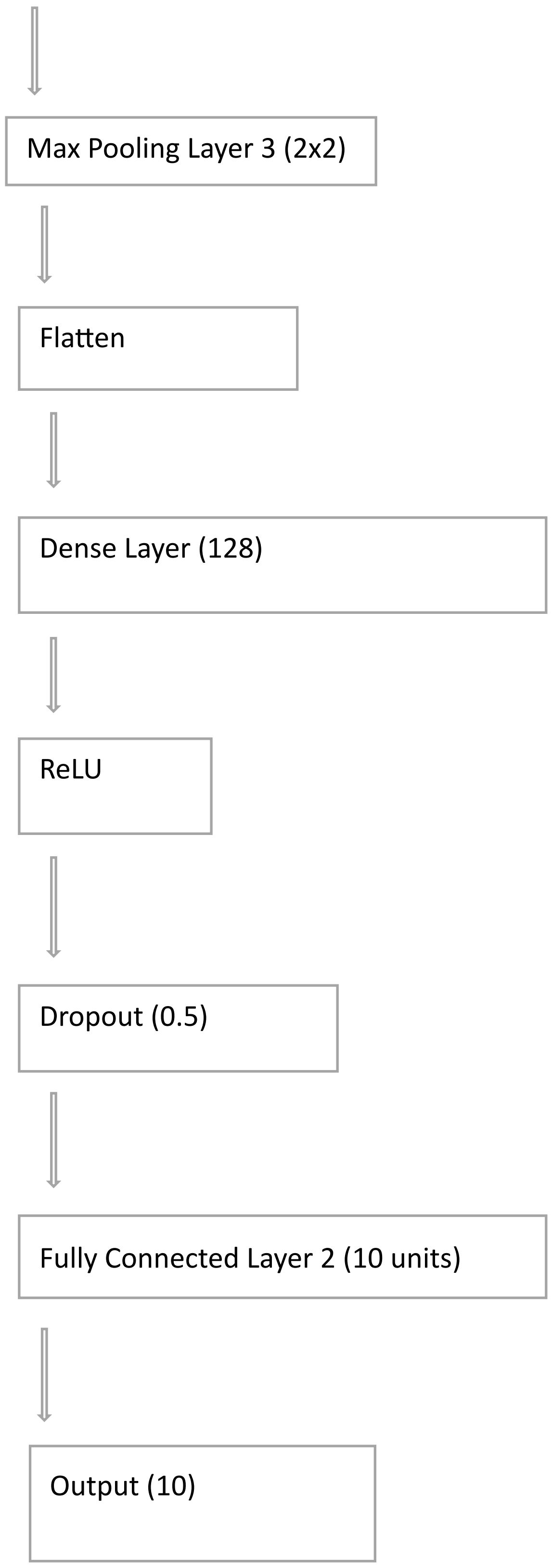
Training: The model is trained iteratively over multiple epochs, where each epoch represents a full iteration over the entire training dataset. During training, metrics like accuracy and loss are monitored.

Evaluation: Post training, the model's performance is evaluated using the validation set in each fold of the cross-validation. Additionally, a confusion matrix and classification report are generated to assess the model's precision, recall, and f1-score across all digit classes.

This comprehensive approach ensures that the CNN model is robust, capable of handling the intricacies of handwritten digit recognition, and performs consistently across different data subsets, highlighting the effective use of various libraries and frameworks in its construction and validation.

Convolutional Neural Network Architecture:


Input(28x28x1)
Convolutional layer1 (32filters, 3x3)
ReLU
Max Pooling Layer 1 (2x2)
Convolutional Layer 2 (64 filters, 3x3)
ReLU
Max Pooling Layer 2 (2x2)
Convolutional Layer 3 (128 filters, 3x3)
ReLU


Max Pooling Layer 3 (2x2)
Flatten
Dense Layer (128)
ReLU
Dropout (0.5)
Fully Connected Layer 2 (10 units)
Output (10)

Results

Training Results Analysis

The CNN model's training over multiple epochs demonstrated consistent improvements in accuracy and reductions in loss, indicating effective learning and adaptation to the handwriting recognition task. Initially, larger decreases in loss were observed, typical of the early phases of training as the model began to fit to the training data.

Validation and Test Results

Validation results from the k-fold cross-validation process reflected a robust generalization of the model across different data subsets. The model achieved high accuracy, typically around 98.7% on average, suggesting that the network effectively captures and classifies the variations in handwritten digits.

Results Analysis

The training and validation metrics showed that the model performs well with a high degree of reliability. However, small fluctuations in validation accuracy across different folds indicated minor variability in model performance, which could be due to the inherent differences in data distribution within each fold. This underscores the importance of using k-fold cross-validation to ensure comprehensive performance evaluation**.**

Performance Evaluation

Confusion Matrix and Classification Report

The confusion matrix and classification report provided deeper insights into the model's performance across individual digit classes. The model displayed high precision and recall for most digits, with particular strength in distinguishing between more distinct digits such as '0' and '1'. However, some confusion was noted between digits with similar shapes, such as '4' and '9', which occasionally were misclassified.

Performance Evaluation Analysis

The detailed analysis of the confusion matrix revealed that while the model performs exceptionally well for most digits, there are areas of potential improvement. The misclassifications between similar-shaped digits suggest that the model might benefit from further fine-tuning of the convolutional layers to better capture the nuances that differentiate similar digits. Enhancements in data augmentation, such as introducing rotations or slight distortions, might also help improve the model's ability to generalize across more challenging scenarios.

Conclusion

The CNN model demonstrated excellent performance in recognizing and classifying handwritten digits from the MNIST dataset, achieving high accuracy and robust generalization as evidenced by the k-fold cross-validation results. The architecture proved effective, particularly with the integration of dropout layers that mitigated overfitting—a common challenge in deep learning models.

References

"Optical Recognition of Handwritten Digits Data Set," UCI Machine Learning Repository.

Code

CNN Implementation for MNIST Digit Recognition

May 1, 2024

[1]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, classification_report
from tensorflow.keras import layers, models, utils
from sklearn.datasets import fetch_openml

# Load the MNIST dataset
mnist = fetch_openml(name='mnist_784', version=1, parser='auto')    # Suppresses␣
↪FutureWarning

# Extract features and labels
X = np.array(mnist.data)
y = np.array(mnist.target)

# Normalize and reshape data for CNN input
X = X.reshape((X.shape[0], 28, 28, 1)).astype('float32') / 255.0  # Normalize␣
↪pixel values to range [0, 1]
y = utils.to_categorical(y, 10)  # Assuming there are 10 classes (digits 0-9)

# Build the CNN Model
def create_model():
    model = models.Sequential([
        # Input layer
        layers.Input(shape=(28, 28, 1)),
        # Convolutional layers
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu',␣
↪padding='same'),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Dropout(0.25),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Dropout(0.25),
        layers.Flatten(),
        # Fully connected layers
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')  # Output layer with softmax␣
↪activation for classification
    ])
    model.compile(optimizer='adam',  loss='categorical_crossentropy',␣
↪metrics=['accuracy'])
    return  model

# Perform K-Fold Cross Validation
def kfold_cross_validation(X, y, n_splits=5, batch_size=128, epochs=10):
    kf  = KFold(n_splits=n_splits)
    fold_no  = 1
    losses  = []
    accuracies  = []

    for train, test in kf.split(X):
        print(f'Training fold {fold_no}...')
        model = create_model()      # Create a new model for each fold
        history  =  model.fit(X[train],  y[train],
                              batch_size=batch_size, epochs=epochs,
                              validation_data=(X[test], y[test]),
                              verbose=1) # Enable verbosity during training

        scores  =  model.evaluate(X[test],  y[test],  verbose=0)
        print(f'Score  for  fold  {fold_no}:  Loss  =  {scores[0]};  Accuracy  =␣
↪{scores[1]*100}%')
        losses.append(scores[0])
        accuracies.append(scores[1])
        fold_no  +=  1

    # Average scores after cross-validation
    print(f'Average  loss:  {np.mean(losses)},  Average  Accuracy:  {np.
↪mean(accuracies)*100}%')

    return  losses,  accuracies,  history

#  Plot  learning  curves
def plot_learning_curves(history):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.show()

# Predictions for Confusion Matrix and Classification Report
def evaluate_model(model, X, y):
    y_pred = model.predict(X)
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_true = np.argmax(y, axis=1)

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred_classes)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

    # Classification Report
    print(classification_report(y_true, y_pred_classes))

# Perform K-Fold Cross Validation
losses, accuracies, history = kfold_cross_validation(X, y)

# Plot learning curves
plot_learning_curves(history)

# Evaluate the model
model = create_model()
model.fit(X, y, epochs=10, batch_size=128, verbose=1)  # Training on entire
 ↪dataset
evaluate_model(model, X, y)

# Display model summary
model.summary()
```

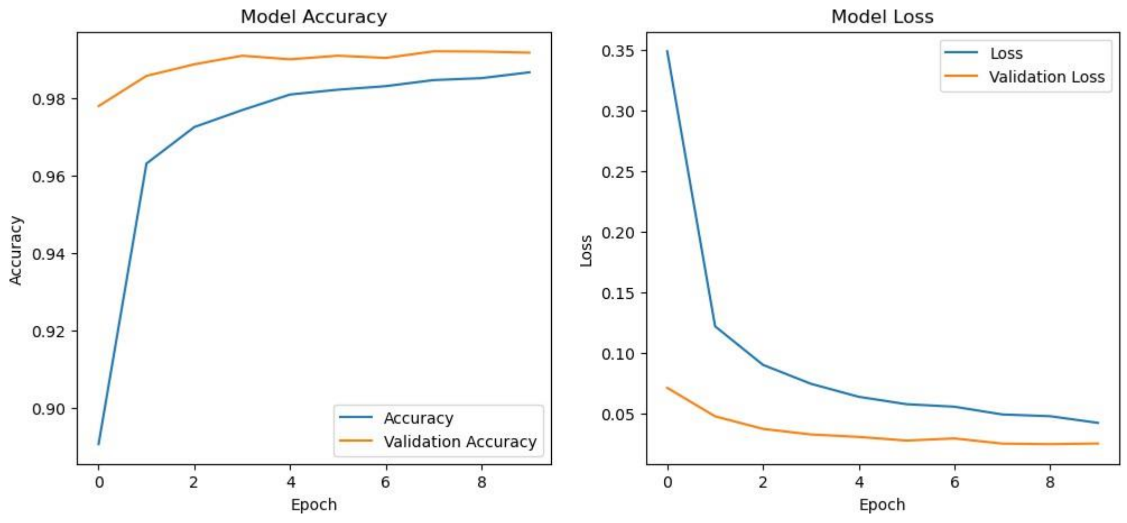
```
Training fold 1...
Epoch 1/10
438/438                    33s 62ms/step -
accuracy: 0.7749 - loss: 0.6848 - val_accuracy: 0.9761 - val_loss: 0.0805
Epoch 2/10
```

```
438/438                 28s 64ms/step -
accuracy: 0.9634 - loss: 0.1217 - val_accuracy:  0.9829 - val_loss: 0.0583
Epoch 3/10
438/438                 24s 55ms/step -
accuracy: 0.9727 - loss: 0.0897 - val_accuracy:  0.9861 - val_loss: 0.0443
Epoch 4/10
438/438                 30s 30ms/step -
accuracy: 0.9780 - loss: 0.0723 - val_accuracy:  0.9885 - val_loss: 0.0402
Epoch 5/10
438/438                 21s 48ms/step -
accuracy: 0.9800 - loss: 0.0643 - val_accuracy:  0.9885 - val_loss: 0.0375
Epoch 6/10
438/438                 19s 44ms/step -
accuracy: 0.9830 - loss: 0.0575 - val_accuracy:  0.9893 - val_loss: 0.0365
Epoch 7/10
438/438                 21s 48ms/step -
accuracy: 0.9850 - loss: 0.0501 - val_accuracy:  0.9907 - val_loss: 0.0321
Epoch 8/10
438/438                 20s 46ms/step -
accuracy: 0.9855 - loss: 0.0460 - val_accuracy:  0.9895 - val_loss: 0.0350
Epoch 9/10
438/438                 20s 45ms/step -
accuracy: 0.9852 - loss: 0.0481 - val_accuracy:  0.9909 - val_loss: 0.0321
Epoch 10/10
438/438                 19s 43ms/step -
accuracy: 0.9884 - loss: 0.0365 - val_accuracy: 0.9911 - val_loss: 0.0304
Score for fold 1: Loss = 0.030399281531572342; Accuracy = 99.10714030265808%
Training fold 2...
Epoch 1/10
438/438                 26s 46ms/step -
accuracy: 0.7837 - loss: 0.6669 - val_accuracy:  0.9773 - val_loss: 0.0794
Epoch 2/10
438/438                 20s 45ms/step -
accuracy: 0.9634 - loss: 0.1264 - val_accuracy:  0.9825 - val_loss: 0.0556
Epoch 3/10
438/438                 19s 42ms/step -
accuracy: 0.9727 - loss: 0.0887 - val_accuracy:  0.9866 - val_loss: 0.0421
Epoch 4/10
438/438                 15s 35ms/step -
accuracy: 0.9773 - loss: 0.0751 - val_accuracy:  0.9891 - val_loss: 0.0364
Epoch 5/10
438/438                 18s 40ms/step -
accuracy: 0.9805 - loss: 0.0650 - val_accuracy:  0.9886 - val_loss: 0.0363
Epoch 6/10
438/438                 20s 46ms/step -
accuracy: 0.9816 - loss: 0.0575 - val_accuracy:  0.9899 - val_loss: 0.0323
Epoch 7/10
438/438                 20s 46ms/step -
```

```
accuracy: 0.9838 - loss: 0.0529 - val_accuracy: 0.9904 - val_loss: 0.0312
Epoch 8/10
438/438                 21s 47ms/step -
accuracy: 0.9854 - loss: 0.0458 - val_accuracy: 0.9906 - val_loss: 0.0315
Epoch 9/10
438/438                 76s 174ms/step -
accuracy: 0.9867 - loss: 0.0431 - val_accuracy: 0.9902 - val_loss: 0.0304
Epoch 10/10
438/438                 18s 42ms/step -
accuracy: 0.9860 - loss: 0.0434 - val_accuracy: 0.9916 - val_loss: 0.0267
Score for fold 2: Loss = 0.02671658620238304; Accuracy = 99.15714263916016%
Training fold 3...
Epoch 1/10
438/438                30s 46ms/step -
accuracy: 0.7819 - loss: 0.6743 - val_accuracy: 0.9755 - val_loss: 0.0793
Epoch 2/10
438/438                19s 44ms/step -
accuracy: 0.9636 - loss: 0.1219 - val_accuracy: 0.9825 - val_loss: 0.0576
Epoch 3/10
438/438                19s 44ms/step -
accuracy: 0.9705 - loss: 0.0946 - val_accuracy: 0.9861 - val_loss: 0.0460
Epoch 4/10
438/438                19s 44ms/step -
accuracy: 0.9761 - loss: 0.0764 - val_accuracy: 0.9875 - val_loss: 0.0388
Epoch 5/10
438/438                13s 31ms/step -
accuracy: 0.9804 - loss: 0.0664 - val_accuracy: 0.9879 - val_loss: 0.0388
Epoch 6/10
438/438                12s 28ms/step -
accuracy: 0.9834 - loss: 0.0554 - val_accuracy: 0.9874 - val_loss: 0.0387
Epoch 7/10
438/438                12s 28ms/step -
accuracy: 0.9829 - loss: 0.0569 - val_accuracy: 0.9891 - val_loss: 0.0349
Epoch 8/10
438/438                11s 26ms/step -
accuracy: 0.9848 - loss: 0.0486 - val_accuracy: 0.9892 - val_loss: 0.0323
Epoch 9/10
438/438                11s 26ms/step -
accuracy: 0.9857 - loss: 0.0459 - val_accuracy: 0.9894 - val_loss: 0.0338
Epoch 10/10
438/438                 11s 25ms/step -
accuracy: 0.9878 - loss: 0.0404 - val_accuracy: 0.9912 - val_loss: 0.0296
Score for fold 3: Loss = 0.029643999412655583; Accuracy = 99.12142753601074%
Training fold 4...
Epoch 1/10
438/438                 13s 25ms/step -
accuracy: 0.7762 - loss: 0.6892 - val_accuracy: 0.9687 - val_loss: 0.1004
Epoch 2/10
```

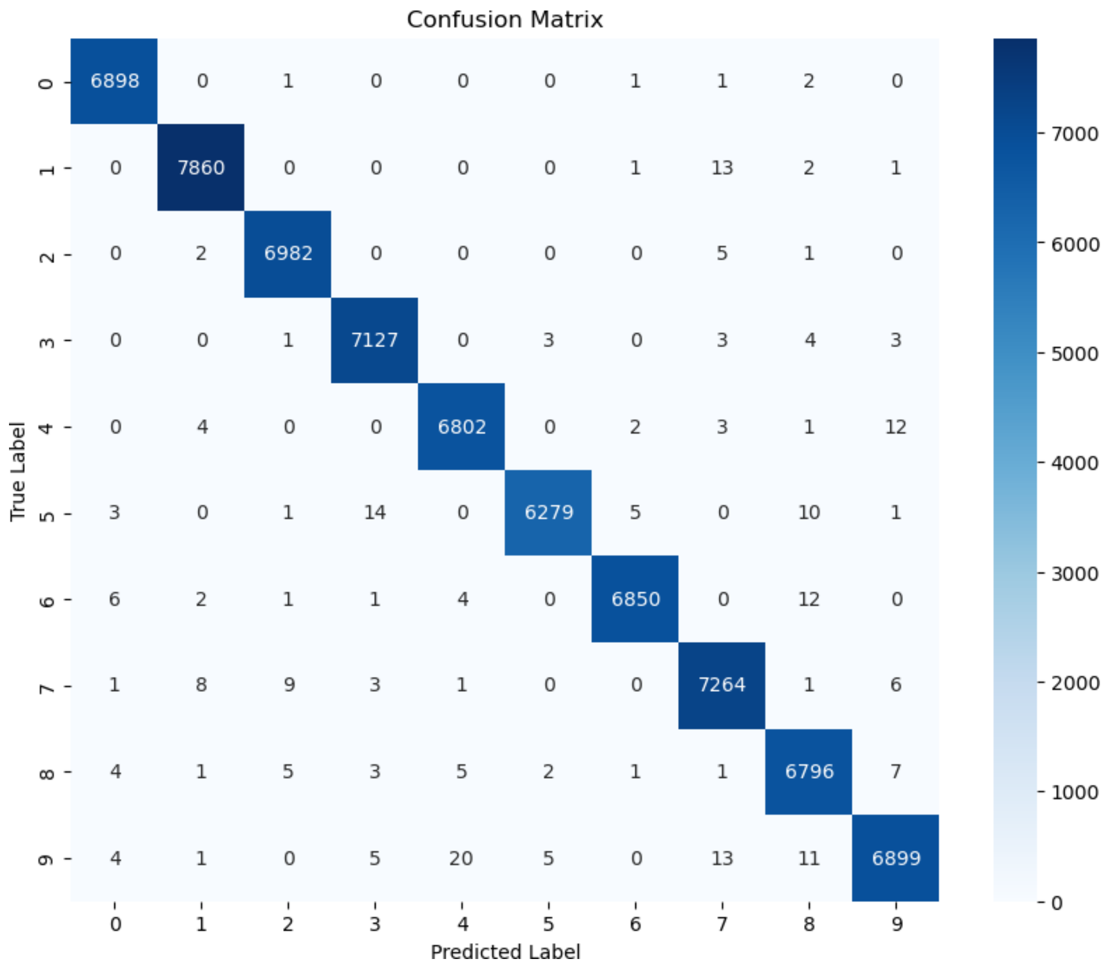
```
438/438                 11s 25ms/step -
accuracy: 0.9626 - loss: 0.1253 - val_accuracy: 0.9807 - val_loss: 0.0623
Epoch 3/10
438/438                 11s 25ms/step -
accuracy: 0.9708 - loss: 0.0927 - val_accuracy: 0.9839 - val_loss: 0.0546
Epoch 4/10
438/438                 11s 25ms/step -
accuracy: 0.9775 - loss: 0.0753 - val_accuracy: 0.9871 - val_loss: 0.0440
Epoch 5/10
438/438                 11s 25ms/step -
accuracy: 0.9804 - loss: 0.0618 - val_accuracy: 0.9873 - val_loss: 0.0411
Epoch 6/10
438/438                 11s 25ms/step -
accuracy: 0.9808 - loss: 0.0602 - val_accuracy: 0.9859 - val_loss: 0.0483
Epoch 7/10
438/438                 11s 25ms/step -
accuracy: 0.9828 - loss: 0.0544 - val_accuracy: 0.9889 - val_loss: 0.0383
Epoch 8/10
438/438                 11s 25ms/step -
accuracy: 0.9862 - loss: 0.0439 - val_accuracy: 0.9893 - val_loss: 0.0356
Epoch 9/10
438/438                 11s 25ms/step -
accuracy: 0.9861 - loss: 0.0451 - val_accuracy: 0.9888 - val_loss: 0.0385
Epoch 10/10
438/438                 11s 26ms/step -
accuracy: 0.9863 - loss: 0.0432 - val_accuracy: 0.9900 - val_loss: 0.0369
Score for fold 4: Loss = 0.03690212965011597; Accuracy = 99.00000095367432%
Training fold 5...
Epoch 1/10
438/438                 13s 26ms/step -
accuracy: 0.7795 - loss: 0.6791 - val_accuracy: 0.9779 - val_loss: 0.0710
Epoch 2/10
438/438                 11s 26ms/step -
accuracy: 0.9610 - loss: 0.1308 - val_accuracy: 0.9856 - val_loss: 0.0475
Epoch 3/10
438/438                 11s 26ms/step -
accuracy: 0.9712 - loss: 0.0936 - val_accuracy: 0.9886 - val_loss: 0.0372
Epoch 4/10
438/438                 11s 26ms/step -
accuracy: 0.9777 - loss: 0.0724 - val_accuracy: 0.9909 - val_loss: 0.0326
Epoch 5/10
438/438                 11s 25ms/step -
accuracy: 0.9803 - loss: 0.0647 - val_accuracy: 0.9899 - val_loss: 0.0306
Epoch 6/10
438/438                 19s 21ms/step -
accuracy: 0.9816 - loss: 0.0595 - val_accuracy: 0.9909 - val_loss: 0.0275
Epoch 7/10
438/438                 11s 25ms/step -
```

```
accuracy: 0.9822 - loss: 0.0567 - val_accuracy: 0.9903 - val_loss: 0.0293
Epoch 8/10
438/438                    18s 20ms/step -
accuracy: 0.9852 - loss: 0.0476 - val_accuracy: 0.9920 - val_loss: 0.0250
Epoch 9/10
438/438                    11s 24ms/step -
accuracy: 0.9851 - loss: 0.0471 - val_accuracy: 0.9919 - val_loss: 0.0246
Epoch 10/10
438/438                    11s 25ms/step -
accuracy: 0.9861 - loss: 0.0426 - val_accuracy: 0.9916 - val_loss: 0.0251
Score for fold 5: Loss = 0.025106439366936684; Accuracy = 99.16428327560425%
Average loss: 0.02975368723273277, Average Accuracy: 99.10999894142151%
```



```
Epoch 1/10
547/547                  13s 22ms/step -
accuracy: 0.7947 - loss: 0.6255
Epoch 2/10
547/547                  12s 22ms/step -
accuracy: 0.9661 - loss: 0.1146
Epoch 3/10
547/547                  12s 23ms/step -
accuracy: 0.9753 - loss: 0.0831
Epoch 4/10
547/547                  12s 23ms/step -
accuracy: 0.9791 - loss: 0.0709
Epoch 5/10
547/547                  12s 23ms/step -
accuracy: 0.9830 - loss: 0.0560
Epoch 6/10
547/547                  12s 22ms/step -
```

```
accuracy: 0.9836 - loss: 0.0555
Epoch 7/10
547/547                12s 22ms/step -
accuracy: 0.9850 - loss: 0.0487
Epoch 8/10
547/547                12s 23ms/step -
accuracy: 0.9867 - loss: 0.0436
Epoch 9/10
547/547                13s 23ms/step -
accuracy: 0.9875 - loss: 0.0407
Epoch 10/10
547/547                12s 23ms/step -
accuracy: 0.9883 - loss: 0.0376
2188/2188                 6s 3ms/step
```



```
              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00      6903
           1       1.00      1.00      1.00      7877
           2       1.00      1.00      1.00      6990
           3       1.00      1.00      1.00      7141
           4       1.00      1.00      1.00      6824
           5       1.00      0.99      1.00      6313
           6       1.00      1.00      1.00      6876
           7       0.99      1.00      1.00      7293
           8       0.99      1.00      0.99      6825
           9       1.00      0.99      0.99      6958

    accuracy                           1.00     70000
   macro avg       1.00      1.00      1.00     70000
weighted avg       1.00      1.00      1.00     70000
```

```
Model: "sequential_5"

 Layer  (type)                  Output Shape              Param  #

 conv2d_10 (Conv2D)             (None, 28, 28, 32)             320

 max_pooling2d_10 (MaxPooling2D) (None, 14, 14, 32)              0

 dropout_15 (Dropout)           (None, 14, 14, 32)               0

 conv2d_11 (Conv2D)             (None, 14, 14, 64)          18,496

 max_pooling2d_11 (MaxPooling2D) (None, 7, 7, 64)                0

 dropout_16 (Dropout)           (None, 7, 7, 64)                 0

 flatten_5 (Flatten)            (None, 3136)                     0

 dense_10 (Dense)               (None, 128)                401,536

 dropout_17 (Dropout)           (None, 128)                      0

 dense_11 (Dense)               (None, 10)                   1,290


 Total  params: 1,264,928  (4.83  MB)

 Trainable  params: 421,642  (1.61  MB)
```

```
Non-trainable params: 0 (0.00 B)

Optimizer params: 843,286 (3.22 MB)
```

```
[ ]:
```