中国传媒大學
COMMUNICATION UNIVERSITY OF CHINA

# 题 目 ： MNIST handwriting recognition based on deep learning:Comparison of CNN and MLP

学年学期：2023-2024-1

课程名称：人工智能进阶：强化学习与深度学习

课程编号：2131030217

课程序号：01

任课教师：张地

姓　　名：闫荻嘉 夏汉强 王蓓蓓 颜秉庆 高海东

学　　号：2021213083001 2020216123012 202213093044
202213103030 2020213053026

**评分区域（由阅卷老师填写）：**

结课成绩：

总评成绩：

提交时间：2023 年 12 月 31 日

# MNIST handwriting recognition based on deep learning: Comparison of CNN and MLP

**Abstract**

Handwriting recognition has been a long-standing challenge in the field of artificial intelligence, and deep learning techniques have shown remarkable success in addressing this task. This report presents a comprehensive exploration of handwritten digit recognition using the MNIST dataset, employing two distinct deep learning architectures: Convolutional Neural Networks (CNN) and Multi-Layer Perceptrons (MLP). We compare the performance of these models, analyzing their strengths and weaknesses in terms of accuracy, training time, and computational complexity.The CNN model, characterized by its ability to automatically learn hierarchical features from input images, demonstrates superior performance in capturing spatial dependencies, making it well-suited for image recognition tasks. On the other hand, the MLP model, with its fully connected layers, provides insights into how a more traditional neural network architecture performs in this context.

All in all, CNN does better than MLP in the handwriting recognition.

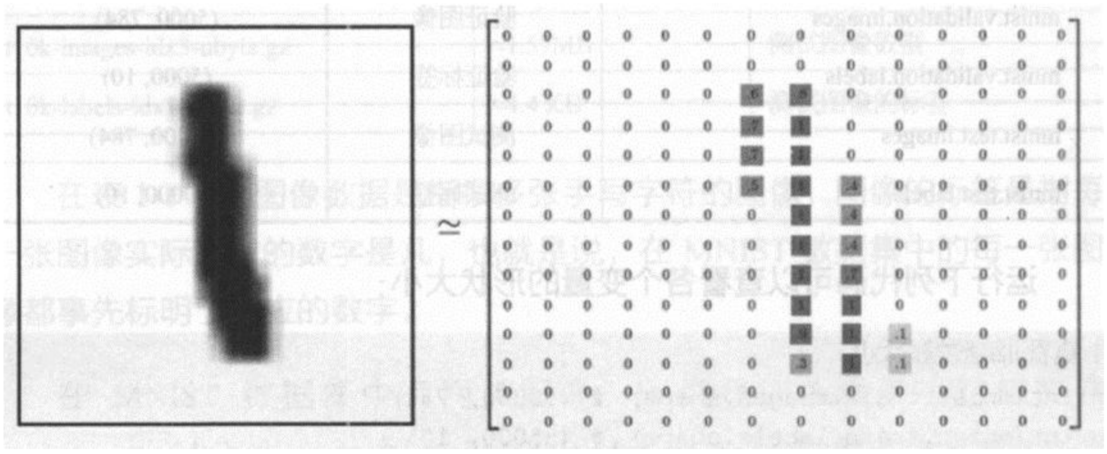
**Keyword**: handwriting recognition, CNN, MLP

# TABLE OF CONTENTS

# I. Introduction

## Introduction of MNIST

The MNIST dataset( Mixed National Institute of Standards and Technology database) is a large collection of hand-written digits dataset compiled by the National Institute of Standards and Technology in the United States. It comprises a training set of 60,000 samples and a test set of 10,000 samples. In MNIST, each sample transforms the original 28x28 grayscale image into a one-dimensional vector of length 784, where each element corresponds to a grayscale value in the image. MNIST utilizes a 10-dimensional vector as the label for each sample, where the index of the vector corresponds to the predicted probability of the sample being the digit represented by that index.



The original MNIST dataset consists of the following four types files.

Table 1：The files included in the MINST dataset

| File_name | File_size | File_usage |
|---|---|---|
| train-images-idx3-ubyte.gz | 9.45MB | Training images' data |
| train - labels-idx1-ubyte.gz | 0.03MB | Training images' labels |
| t10k-images-idx3-ubyte.gz | 1.57MB | Test images' data |

|  |  |  |
|---|---|---|
| t10k-labels-idx1-ubyte.gz | 4.4KB | Test images' labels |

In the MNIST dataset, there are two types of images: one is the training images (corresponding to the files train-images-idx3-ubyte.gz and train-labels-idx1-ubyte.gz), and the other is the test images (corresponding to the files t10k-images-idx3-ubyte.gz and t10k-labels-idx1-ubyte.gz). There are a total of 60,000 training images available for researchers to train suitable models. Additionally, there are 10,000 test images provided for researchers to evaluate the performance of the trained models.
Handwritten digit recognition has a wide range of practical applications in real-life scenarios, such as ordinary information forms in postal and courier services, data statistics in the financial sector, and handwriting input methods on smartphones. This has made research in handwritten digit recognition a focal point in the field of deep learning.

In the past, various methods, including those based on backpropagation neural networks, class centroid Euclidean distance, Bayesian classification algorithms, etc., have been proposed for handwritten digit recognition. However, their effectiveness has not always met expectations.

**The Purpose of our research**

Our team utilizes convolutional neural networks (CNNs) and fully connected neural networks based on deep learning to address the handwritten digit recognition problem. In comparison to traditional machine learning algorithms, they offer the following advantages:

1. Efficient Feature Extraction: There is no need to pre-design the content and quantity of features. Convolutional neural networks can train and self-correct to achieve effective feature extraction.

2. Simplicity in Data Format: Minimal preprocessing is required for the data format.

3. Fewer Parameters: Initialization involves assigning random weights and biases to each neuron, and the model subsequently self-corrects these parameters to optimize performance. CNNs not only overcome the drawbacks of preprocessing in traditional methods but also enhance prediction accuracy.
The final aim is to find which is better, CNN or MLP, and can be more widely used and analyzed.

## II.Code Explanation

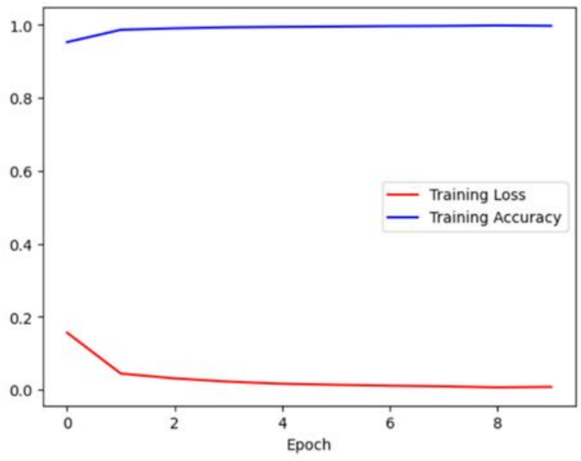
The experiment involves MNIST handwritten digit recognition, a classic problem. The MNIST dataset consists of a series of 28x28 pixel handwritten digit images. Our goal is to use a deep learning model to accurately recognize these digits.

First, we preprocess the MNIST data. Using `transforms.Compose`, we transform the images into tensors and normalize them to ensure they are ready for training before feeding them into the model.

**Convolutional Neural Network (CNN) Model:**

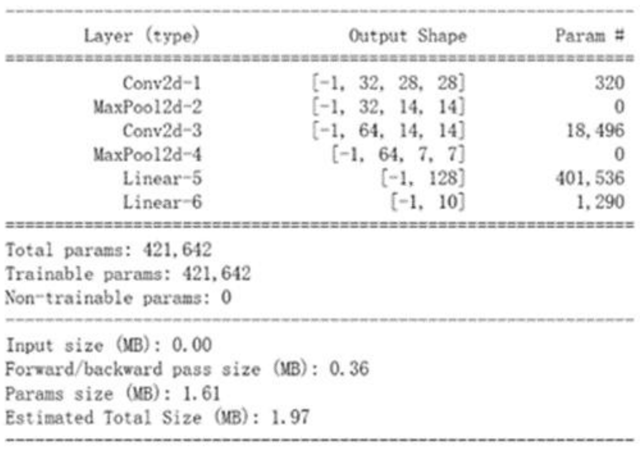
We designed a simple CNN model comprising convolutional layers, pooling layers, and fully connected layers. This structure aims to extract features from the images, adapting to the characteristics of handwritten digits.

CNN Training and Learning Curve:
We trained the CNN model using the Adam optimizer and cross-entropy loss function. By iterating through the training data, we recorded the loss and accuracy for each epoch. The learning curve was plotted using the matplotlib library.



Summary of CNN Model Structure:

We used the torchsummary library to print the structure and parameter count of the CNN model, aiding in understanding the model's scale and complexity.



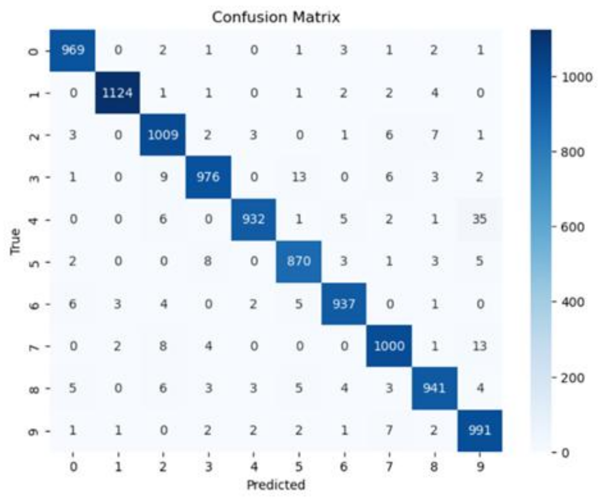
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
         MaxPool2d-2           [-1, 32, 14, 14]               0
            Conv2d-3           [-1, 64, 14, 14]          18,496
         MaxPool2d-4             [-1, 64, 7, 7]               0
            Linear-5                  [-1, 128]         401,536
            Linear-6                   [-1, 10]           1,290
================================================================
Total params: 421,642
Trainable params: 421,642
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.36
Params size (MB): 1.61
Estimated Total Size (MB): 1.97
----------------------------------------------------------------
```

The model starts with a convolutional layer (Conv2d) with 32 filters, each sized 3x3, responsible for extracting low-level features like edges and textures. This layer has 320 learnable parameters to adjust filter weights. Following this is a max-pooling layer (MaxPool2d) to reduce the output size by half for better processing of large-scale images without sacrificing efficiency.

We then employ another convolutional layer with 64 filters, followed by another max-pooling layer. This sequence of operations allows the network to gradually capture more abstract features. Next, two fully connected layers (Linear) with 128 and 10 neurons respectively map the extracted features to the 10 output categories of interest. In total, there are 401,536 parameters for adjusting the weights in these fully connected layers.

The entire model has 421,642 trainable parameters. The Adam optimizer is used to adjust these parameters during training, enabling the network to learn more accurate features for precise classification of handwritten digits.

CNN Confusion Matrix and Heatmap Analysis:



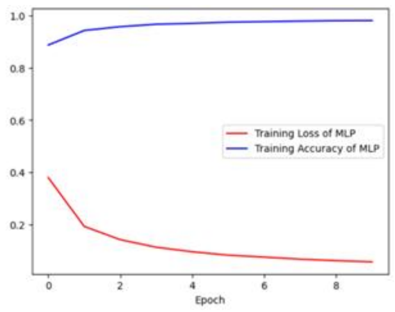
Predictions were made on the test set, and a confusion matrix was generated. A heatmap was plotted using seaborn for in-depth analysis of the model's performance on each category.

**Multilayer Perceptron (MLP) Model:**

To provide a comparison, we designed a simple MLP model for further validation of our experimental results.

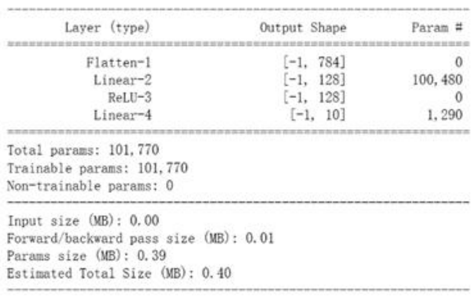
MLP Training and Learning Curve:

Similar training processes were applied to the MLP model, and corresponding learning curves were plotted.



Summary of MLP Model Structure:

Using the torchsummary library, we printed the structure and parameter count of the MLP model.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1                  [-1, 784]               0
            Linear-2                  [-1, 128]         100,480
              ReLU-3                  [-1, 128]               0
            Linear-4                   [-1, 10]           1,290
================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.39
Estimated Total Size (MB): 0.40
----------------------------------------------------------------
```

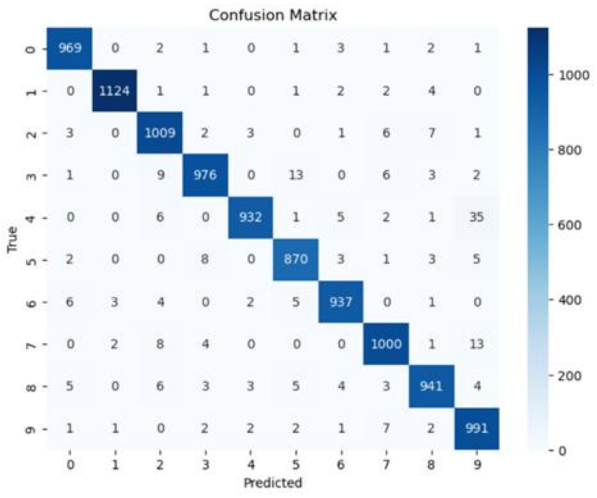
The hierarchical structure of the model begins with a Flatten layer, flattening the 28x28 pixel image into a 784-dimensional vector. A fully connected layer then maps this vector to 128 features, involving around 100,480 learnable parameters to better fit the training data. Following this, a ReLU activation function introduces non-linearity, connecting to the final fully connected layer mapping 128 features to the 10 output categories of interest.

The total trainable parameters for the MLP model are 101,770. During training, the Adam optimizer is utilized to adjust these parameters, allowing the model to better adapt to the features of handwritten digit images.

In summary, this structure is intuitive, starting with a simple image representation and gradually extracting higher-level features, enabling the model to correctly classify handwritten digits.

MLP Confusion Matrix and Heatmap Analysis:

Similar confusion matrix analysis and heatmap plotting were conducted on the MLP model to further compare the performance of the two models.

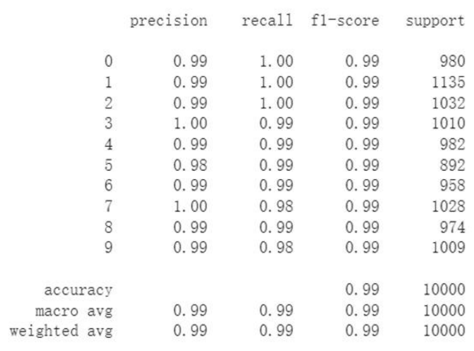


## III.Result and Analysis

In deep learning, evaluating the performance of a model is a crucial step. It helps us understand how the model performs in practical applications and allows us to adjust the model's hyperparameters and structure promptly.Handwritten digit recognition is a multi-class classification problem. For this specific scenario, we introduce three key evaluation index—Accuracy, F1-score, and Confusion Matrix—to compare the quality of the model. They provide a comprehensive understanding of the model's performance in multi-class classification tasks. Below, I will explain their importance and present the data of our model in these index.

To begin with, we generated a classification report which indicated that all index of the CNN were at 1, which represents an ideal state .It potentially hints at **overfitting** issues.

Based on the advice provided by the professor last week, we added **noise** through the technique of Dropout.This enhances the model's robustness and generalization ability

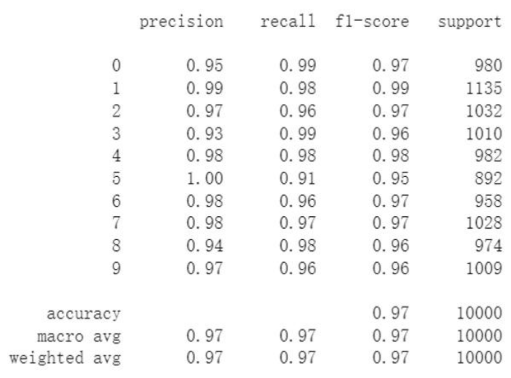


```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       980
           1       0.99      1.00      0.99      1135
           2       0.99      1.00      0.99      1032
           3       1.00      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.99      0.99       892
           6       0.99      0.99      0.99       958
           7       1.00      0.98      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

classification report of CNN after adding noisy



```
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       980
           1       0.99      0.98      0.99      1135
           2       0.97      0.96      0.97      1032
           3       0.93      0.99      0.96      1010
           4       0.98      0.98      0.98       982
           5       1.00      0.91      0.95       892
           6       0.98      0.96      0.97       958
           7       0.98      0.97      0.97      1028
           8       0.94      0.98      0.96       974
           9       0.97      0.96      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

classification report of MLP

**Accuracy:**

In the task of handwritten digit recognition, accuracy intuitively represents the proportion of correctly classified samples to the total samples. This is crucial for gaining an initial understanding of the overall performance of the model. However, in certain cases, accuracy might not fully reflect the model's performance, especially when there's an imbalance in class distribution, such as certain digits appearing far more frequently than others. Hence, accuracy needs to be considered in conjunction with other evaluation metrics to comprehensively understand the model's performance.

| Model Accuracy Comparison | | |
|---|---|---|
| | CNN | MLP |
| **Accuracy** | 0.9990 | 0.9717 |

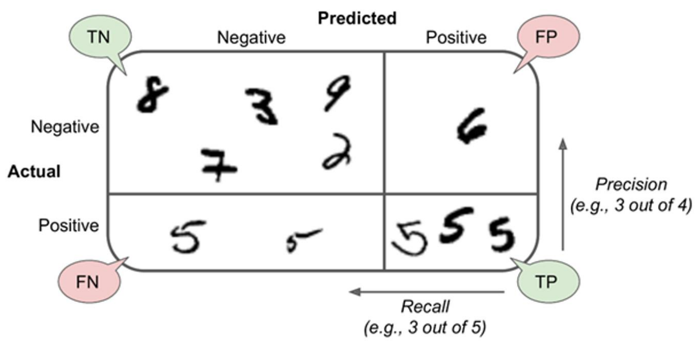
Model Accuracy Comparison

From the comparison in this table, it's very clear that the accuracy of CNN is 0.99, higher than MLP， which is 0.97.

BUT, Accuracy alone might not fully showcase the model's performance and should be considered alongside other evaluation metrics to grasp the overall picture.

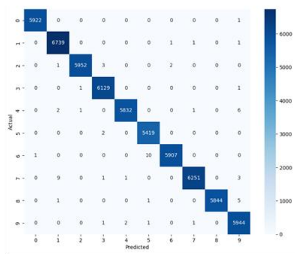
**Confusion Matrix:**

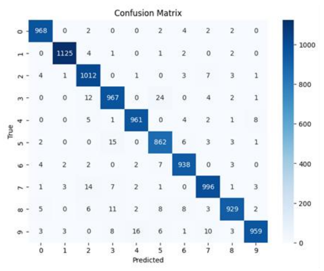
The confusion matrix, also known as an error matrix, represents a standard format for assessing accuracy. It is depicted in a matrix form of n rows and n columns. Various evaluation metrics such as overall accuracy, producer's accuracy, and user's accuracy are utilized, reflecting the precision of image classification from different perspectives.



In a binary classification problem,It categorizes samples based on their true labels and model predictions into four scenarios: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN).

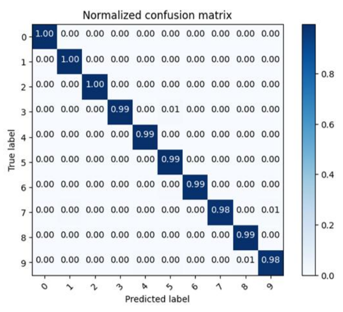
These index aid in evaluating the model's performance in each class, providing a comprehensive understanding of the model's performance and error types. We generated and visualized the confusion matrices for both models.



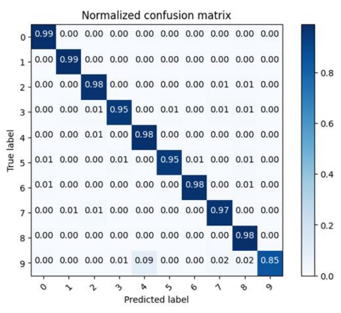


Confusion Matrix of CNN Confusion Matrix of MLP

BUT The initial visualization wasn't good at comparing their quality. So,I transformed it into a probability-based visualization.



CNN



MLP

Comparing the two matrices, it's observable that CNN's recognition quality **exceeds** MLP.

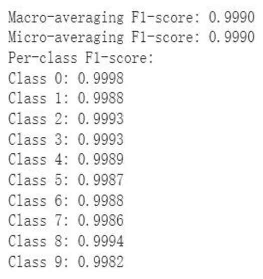
**F1-score:**

The F1-score is a metric that combines a model's precision and recall.In multi-class classification, precision, recall, and F1-score can be calculated for each class. These values can then undergo macro-average or micro-average processing to derive a single evaluation metric. The F1-score comprehensively considers the model's performance across each class, making it a more holistic and accurate measure to assess the model's performance.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

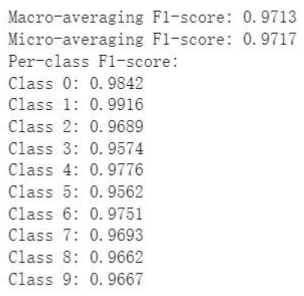
The formula for the F1-score

I visualized the F1-scores of CNN and MLP for each class and created a bar graph (with the y-axis range set from 0.95 to 1.0 for clearer comparison).



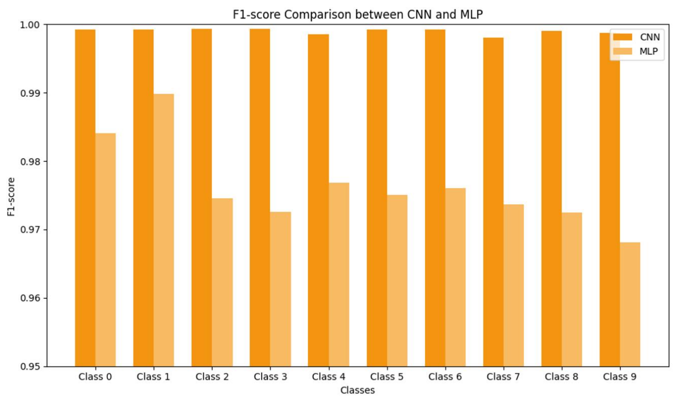
```
Macro-averaging F1-score: 0.9990
Micro-averaging F1-score: 0.9990
Per-class F1-score:
Class 0: 0.9998
Class 1: 0.9988
Class 2: 0.9993
Class 3: 0.9993
Class 4: 0.9989
Class 5: 0.9987
Class 6: 0.9988
Class 7: 0.9986
Class 8: 0.9994
Class 9: 0.9982
```

F1-score of CNN



```
Macro-averaging F1-score: 0.9713
Micro-averaging F1-score: 0.9717
Per-class F1-score:
Class 0: 0.9842
Class 1: 0.9916
Class 2: 0.9689
Class 3: 0.9574
Class 4: 0.9776
Class 5: 0.9562
Class 6: 0.9751
Class 7: 0.9693
Class 8: 0.9662
Class 9: 0.9667
```

F1-score of MLP



From this graph, it's evident that although both models have commendable scores, CNN's scores are notably higher than MLP's.

**Conclusion:**

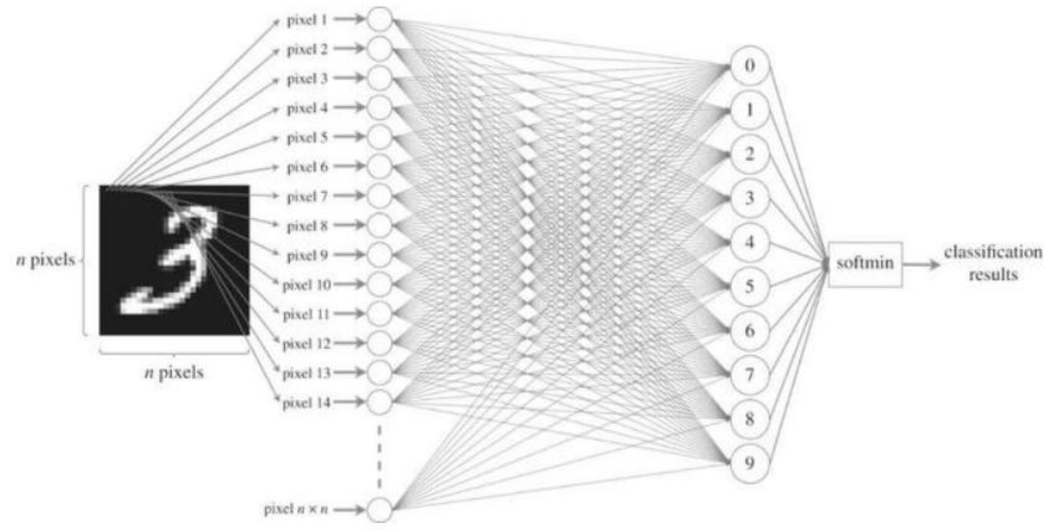
Thus far, we have utilized various metrics such as accuracy, confusion matrix, and F1-score to evaluate CNN and MLP models. Finally, the conclusion was drawn that the CNN model exhibits superior capability in recognizing handwritten digits, achieving a relatively ideal level of accuracy and performance on the test dataset.

The initial stage in the vast field of Artificial Intelligence and Computer Vision is handwritten digit recognition. CNN outperforms alternative classifiers, as evidenced by the results of the trial. With more convolution layers and buried neurons, the findings may be made more precise. Handwritten Digit Recognition (HDR) has three phases. The first step is pre-processing, involving converting the dataset to binary format and applying image processing. The second phase is segmentation, where the picture is divided into several parts. The third step is feature extraction, identifying picture characteristics. The use of CNN greatly improves HDR results.

# IV.Conclusion

### The problems we solved

The MNIST dataset plays a crucial role in the field of deep learning, where numerous models have achieved outstanding accuracy on this dataset, showcasing the effectiveness of various architectures such as CNNs (Convolutional Neural Networks), MLPs (Multilayer Perceptrons), and others in handwritten digit recognition. These experiments provide us with an opportunity to deeply understand the behavior of neural networks, offering insights into how models learn and recognize patterns. Furthermore, MNIST is widely utilized in educational resources, serving as an introductory guide for beginners in concepts like image classification, neural networks, and machine learning

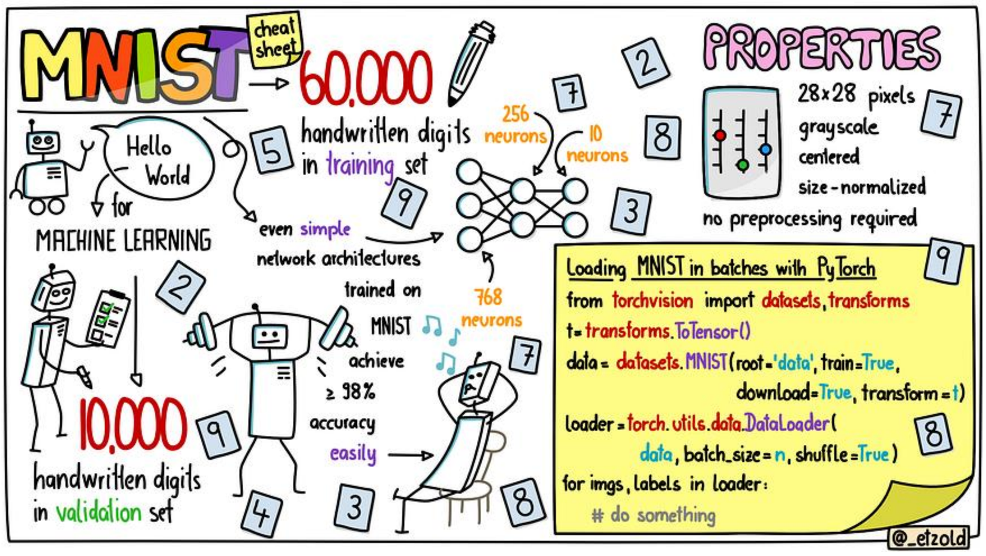


**Possible research in future ：**

Although models perform well on the standard MNIST dataset, there's a need to enhance their adaptability to variations in handwritten digits, encompassing writing styles, distortions, noise, and digits presented against diverse backgrounds. This drives us to explore continual learning techniques, enabling models to continually learn from new data without forgetting previously acquired knowledge. Additionally, research into few-shot or zero-shot learning is being pursued to improve generalization for rare classes or scenarios with limited samples.Moreover, shifting focus towards more realistic scenarios involving damaged, distorted handwritten digits and enhancing model interpretability becomes crucial for applications. Furthermore, there's a possibility of expanding towards multi-digit recognition, such as recognizing sequences of numbers like phone numbers from images.

These research directions aim to propel handwritten digit recognition beyond the MNIST dataset towards higher generalization, robustness in real-world scenarios, covering advanced concepts like continual learning, interpretability, and model fairness.

[Rosbutness: Evaluating the model's performance under various conditions such as different writing styles, noise levels, or low-resolution inputs.]


MNIST
cheat sheet
60.000
handwritten digits in training set
Hello World
for
MACHINE LEARNING
even simple network architectures trained on MNIST achieve ≥ 98% accuracy easily
256 neurons
10 neurons
768 neurons
10.000
handwritten digits in validation set
PROPERTIES
28x28 pixels
grayscale
centered
size-normalized
no preprocessing required
@_etzold

Loading MNIST in batches with PyTorch

```
from torchvision import datasets, transforms
t = transforms.ToTensor()
data = datasets.MNIST(root='data', train=True,
                      download=True, transform=t)
loader = torch.utils.data.DataLoader(
         data, batch_size=n, shuffle=True)
for imgs, labels in loader:
    # do something
```

## Reference

[1]Y. Yang and T. Wang, "Research on MNIST Handwritten Number Recognition Based on Convolutional Neural Networks," 2023 2nd International Conference on Artificial Intelligence and Blockchain Technology (AIBT), Zibo, China, 2023, pp. 28-32, doi: 10.1109/AIBT57480.2023.00012.

[2] C. Xu, "Applying MLP and CNN on Handwriting Images for Image Classification Task," 2022 5th International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE), Wuhan, China, 2022, pp. 830-835, doi: 10.1109/AEMCSE55572.2022.00167.

## Appendix

Division of labour:

Background: Hanqiang Xia 2020213053026, Dijia Yan 2021213083001

Code Explanation: Bingqing Yan 2020216123012

Result and analysis: Haidong Gao 202213093044

Conclusion: Beibei Wang 202213103030

Code: Bingqing Yan, Haidong Gao

Project file:

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from tqdm import tqdm
from torchsummary import summary
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score,
classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Download and load MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=64,
shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64,
shuffle=False)

# Simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Simple MLP model
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Initialize CNN model, loss function, and optimizer
cnn_model = SimpleCNN()
cnn_criterion = nn.CrossEntropyLoss()
cnn_optimizer = optim.Adam(cnn_model.parameters(), lr=0.001)

# Training loop for CNN
num_epochs = 10
cnn_train_losses = []
cnn_train_accuracies = []

for epoch in range(num_epochs):
    cnn_model.train()
    running_loss = 0.0
    correct_predictions = 0
    total_samples = 0

    for images, labels in train_loader:
        cnn_optimizer.zero_grad()
        outputs = cnn_model(images)
        loss = cnn_criterion(outputs, labels)
        loss.backward()
        cnn_optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs.data, 1)
        total_samples += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()

    average_loss = running_loss / len(train_loader)
    cnn_train_losses.append(average_loss)

    accuracy = correct_predictions / total_samples
    cnn_train_accuracies.append(accuracy)

    print(f'CNN Epoch [{epoch+1}/{num_epochs}], Loss: {average_loss:.4f}, 
Accuracy: {accuracy * 100:.2f}%')

# Plot CNN learning curves
plt.plot(cnn_train_losses, label='CNN Training Loss', color='r')
plt.plot(cnn_train_accuracies, label='CNN Training Accuracy', color='b')
plt.legend()
plt.xlabel('Epoch')
plt.show()

# CNN Model summary
summary(cnn_model, input_size=(1, 28, 28))

# Evaluate CNN on test set
cnn_model.eval()
cnn_all_preds = []
cnn_all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        outputs = cnn_model(images)
        _, predicted = torch.max(outputs.data, 1)
        cnn_all_preds.extend(predicted.numpy())
        cnn_all_labels.extend(labels.numpy())

# CNN F1-score and accuracy calculation
cnn_macro_f1 = f1_score(cnn_all_labels, cnn_all_preds, average='macro')
cnn_micro_f1 = f1_score(cnn_all_labels, cnn_all_preds, average='micro')
```

```python
cnn_per_class_f1 = f1_score(cnn_all_labels, cnn_all_preds, average=None)

cnn_accuracy = accuracy_score(cnn_all_labels, cnn_all_preds)
print(f'CNN Accuracy: {cnn_accuracy:.4f}')
print(f'CNN Macro-averaging F1-score: {cnn_macro_f1:.4f}')
print(f'CNN Micro-averaging F1-score: {cnn_micro_f1:.4f}')

# Confusion matrix for CNN
cnn_cm = confusion_matrix(cnn_all_labels, cnn_all_preds)
sns.heatmap(cnn_cm, annot=True, fmt='g', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('CNN Confusion Matrix')
plt.show()

# Simple MLP model initialization
mlp_model = SimpleMLP()
mlp_criterion = nn.CrossEntropyLoss()
mlp_optimizer = optim.Adam(mlp_model.parameters(), lr=0.001)

# Training loop for MLP
mlp_num_epochs = 10
mlp_train_losses = []
mlp_train_accuracies = []

for epoch in range(mlp_num_epochs):
    mlp_model.train()
    running_loss = 0.0
    correct, total = 0, 0

    for images, labels in train_loader:
        mlp_optimizer.zero_grad()
        outputs = mlp_model(images)
        loss = mlp_criterion(outputs, labels)
        loss.backward()
        mlp_optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    mlp_train_accuracy = correct / total
    mlp_train_loss = running_loss / len(train_loader)

    mlp_train_accuracies.append(mlp_train_accuracy)
    mlp_train_losses.append(mlp_train_loss)

    print(f'MLP Epoch [{epoch + 1}/{mlp_num_epochs}], Loss: 
{mlp_train_loss:.4f}, Accuracy: {mlp_train_accuracy * 100:.2f}%')

# Plot MLP learning curves
plt.plot(mlp_train_losses, label='MLP Training Loss', color='r')
plt.plot(mlp_train_accuracies, label='MLP Training Accuracy', color='b')
plt.legend()
plt.xlabel('Epoch')
plt.show()

# MLP Model summary
summary(mlp_model, input_size=(1, 28, 28))

# Evaluate MLP on test set
mlp_model.eval()
mlp_all_preds = []
mlp_all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        outputs = mlp_model(images)
        _, predicted = torch.max(outputs, 1)
        mlp_all_preds.extend(predicted.numpy())
        mlp_all_labels.extend(labels.numpy())

# MLP F1-score and accuracy calculation
mlp_macro_f1 = f1_score(mlp_all_labels, mlp_all_preds, average='macro')
mlp_micro_f1 = f1_score(mlp_all_labels, mlp_all_preds, average='micro')
mlp_per_class_f1 = f1_score(mlp_all_labels, mlp_all_preds, average=None)

mlp_accuracy = accuracy_score(mlp_all_labels, mlp_all_preds)
print(f'MLP Accuracy: {mlp_accuracy:.4f}')
print(f'MLP Macro-averaging F1-score: {mlp_macro_f1:.4f}')
print(f'MLP Micro-averaging F1-score: {mlp_micro_f1:.4f}')

# Confusion matrix for MLP
mlp_cm = confusion_matrix(mlp_all_labels, mlp_all_preds)
sns.heatmap(mlp_cm, annot=True, fmt='g', cmap='Blues')
plt.xlabel('Predicted')
```

```python
plt.ylabel('True')
plt.title('MLP Confusion Matrix')
plt.show()

# Compare results between CNN and MLP
print("CNN Results:")
print(f'Accuracy: {cnn_accuracy:.4f}')
print(f'Macro-averaging F1-score: {cnn_macro_f1:.4f}')
print(f'Micro-averaging F1-score: {cnn_micro_f1:.4f}')
print("MLP Results:")
print(f'Accuracy: {mlp_accuracy:.4f}')
print(f'Macro-averaging F1-score: {mlp_macro_f1:.4f}')
print(f'Micro-averaging F1-score: {mlp_micro_f1:.4f}')
```