**A Comprehensive Report on Handwriting Recognition using Deep Learning**

LIN Lixin AN Yuze ZHANG Huanwei MA Yicheng ZHANG Wenming

**1. Introduction**

Handwriting recognition has been a longstanding challenge and area of interest in computer vision and pattern recognition. The ability to reliably convert handwritten text into a digital form has applications in document digitization, postal address reading, bank check processing, form data extraction, and beyond. With the emergence of deep learning, especially Convolutional Neural Networks (CNNs) and other advanced architectures, the accuracy and reliability of handwriting recognition systems have improved significantly.

This report presents a project that implements and analyzes a handwriting recognition task using a Python script titled *Handwriting\_Recognition.py*. Within this report, we will delve deeply into the model design, covering the choice of architecture, loss functions, optimizers, and other hyperparameters. We will also discuss the training process, and then visualize performance across various settings. Specifically, we will study performance metrics such as training loss, training accuracy, and test accuracy for different numbers of epochs, different loss functions, diverse learning rates, and a range of batch sizes. Finally, we will visualize predicted labels in comparison to ground truths for a subset of the test data.

Throughout this project, several key aspects of deep learning models, such as convolutional layers for feature extraction in images and fully connected layers for classification, are utilized. The aim is to balance model complexity with the ability to generalize effectively, ensuring we have a system that is both accurate and computationally feasible. This comprehensive coverage provides both a theoretical and practical perspective on building a handwriting recognition system.

**2. Data Preparation and Preprocessing**

**2.1 Data Overview**

The following Table 2.1 shows the dataset we using in our project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset** | **Trainset** | **Test set** | **Image Size** | **Channels** |
| KMNIST | 60,000 | 10,000 | 28×28 | 1 |

Table 2.1

**2.2 Data Preprocessing**

Preprocessing is essential to ensure that the images used for training and testing are standardized, which can improve model convergence and overall performance.

1. Normalization: The original pixel values ranging from 0 to 255 are linearly mapped to the range [−1,1] to accelerate model convergence. Compared to the traditional [0,1] normalization used in MNIST, this method is better suited for the output distribution of the ReLU activation function.
2. Subset Sampling Mechanism: A *subset\_ratio* parameter is implemented to allow the use of a subset of the dataset for rapid experimentation. This is achieved using *torch.utils.data*.Subset and *torch.randperm* for random sampling.
3. Data Loader: Uses *torch.utils.data.DataLoader* to create data loaders, sets the batch\_size parameter, and enables shuffle=True on the training set to randomize the data during each epoch.

**3. Model Design**

Selecting an appropriate model architecture is a critical step for successful handwriting recognition. This project primarily uses a Convolutional Neural Network (CNN), which is well-suited for image classification tasks. Below we outline the structure and analysis for the chosen model.

**3.1 Convolutional Neural Networks**

CNNs specialize in capturing spatial features in images by applying convolutional filters. These filters help learn local patterns such as edges, curves, and shapes, which is crucial for distinguishing different handwritten characters.

1. Input Layer: Accepts 28×28 pixel grayscale images.
2. Two Convolutional Layers: Each followed by a ReLU activation function and max pooling.
   * Conv1: 32 convolutional kernels of size 3×3
   * Conv2: 64 convolutional kernels of size 3×3
3. Two Fully Connected Layers:
   * FC1: Outputs 128 neurons with ReLU activation
   * FC2: Outputs 10 neurons, corresponding to 10 classes

The total number of parameters in the model is approximately 420,000.

Below are partly code showing for this part：

Convolutional layer 1:

class HandwritingRecognitionModel(nn.Module):

    def \_\_init\_\_(self):

        super(HandwritingRecognitionModel, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 32, kernel\_size=3, padding=1)

Convolutional layer 2:

        self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1)

Fully connected layer :

        self.fc1 = nn.Linear(64  *7*  7, 128)  #Fully connected layer 1

        self.fc2 = nn.Linear(128, 10)   # Output layer (10 classes)

    def forward(self, x):

Forward pass implementation:

        x = torch.relu(self.conv1(x))

        x = torch.max\_pool2d(x, 2)

        x = torch.relu(self.conv2(x))

        x = torch.max\_pool2d(x, 2)

        x = x.view(x.size(0), -1)

        x = torch.relu(self.fc1(x))

        x = self.fc2(x)

        return x

**3.2 Structural Characteristics Analysis**:

1. **Feature Extraction Stage**

* ***Dual Convolutional Layer Design****:* Channel depth increases from 32 to 64, progressively extracting features from edges → textures → semantic representations.
* ***3×3 Convolution Kernels****:* Compared to the 5×5 kernels in LeNet-5, these enhance the ability to capture local features while reducing the number of parameters by 36%.
* ***Max Pooling****:* Two 2×2 downsampling operations result in a final feature map size of 7×7.

1. **Classifier Stage**

* ***Fully Connected Layer Dimensions****:* 128 nodes in the hidden layer strike a balance between model capacity and the risk of overfitting.
* ***Activation Function Choice****:* ReLU replaces Sigmoid to mitigate the vanishing gradient problem.

**3.3 Innovation of this model design**  
To improve experimental efficiency and flexibility, we introduced the **FAST\_MODE** mechanism, which allows seamless switching between rapid prototyping and full evaluation by changing the number of epoch from 5 to 10.

Following is thr partly code showing for this part:

    # Experiment control parameters

    FAST\_MODE = False  # Enable fast mode

    experiment\_epochs = 5 if FAST\_MODE else 10  # Epochs for fast vs regular mode

    subset\_ratio = 0.3 if FAST\_MODE else 1.0

**3.4 Loss Function Choosing**

The loss function measures the discrepancy between the model’s predictions and the ground truth labels. Choosing the correct loss function is crucial for guiding the learning process. Below are the description of some loss functions:

1. **Cross-Entropy Loss** is common in multi-class classification tasks. In PyTorch, it can be implemented using nn.CrossEntropyLoss. Cross-entropy penalizes the divergence between the predicted probability distribution and the actual distribution (which is a one-hot vector in a multi-class setting). This leads to better convergence and stable gradients for classification.
2. **Mean Squared Error (MSE) Loss** is more common in regression tasks, but it can be used for classification if we encode the labels appropriately. However, it is generally not recommended for classification, as it can lead to slower convergence and difficulties in gradient optimization.

In this project, we first use Cross-Entropy Loss as it aligns with the standard approach for classification, providing robust performance and interpretability. As part of our performance visualization, we also explore an alternative loss function (MSELoss) to compare how the training loss, training accuracy, and test accuracy evolve with the number of epochs.

**3.5 Optimizer Choosing**

Training a neural network involves adjusting its parameters (filters in convolutional layers and weights in fully connected layers) to minimize the chosen loss function. To accomplish this, we choose Adam for the optimizer, which use running averages of both gradients and squared gradients, offering efficient training and robust performance across a wide range of tasks.

**3.6 Hyperparameters**

Hyperparameters are external configurations that control the behavior of the model training process. Selecting appropriate hyperparameters is essential for efficient training and strong generalization. Here are the primary hyperparameters in this project:

1. **Batch Size**: Number of samples processed before the model’s internal parameters are updated. The choices in the handwriting recognition project range from 8 to 64.
2. **Learning Rate (LR)**: Controls how big a step we take during parameter updates. If the LR is too large, training might diverge; if it is too small, training might be excessively slow or get stuck in local minima. So the ranges for LR in this project are from 0.1 down to 1e-5.
3. **Number of Epochs**: An epoch is a single pass through the entire training dataset. We experiment with different epoch counts (5 and 10) to see how the model’s performance plateaus.

**3.7 Ensemble Learning**

Ensemble learning is the process of combining multiple models to improve overall performance and generalization. In this project, we employed a simple yet effective ensemble learning strategy based on model averaging.We used 3 independently trained CNN models with the same architecture but different random initializations, which creates diversity in the learned representations. The idea is to train many models independently and then average their predictions during inference, which can reduce variance and improve accuracy.

class SimpleEnsemble(nn.Module):

    def \_\_init\_\_(self, num\_models=3):

        super(SimpleEnsemble, self).\_\_init\_\_()

        self.models = nn.ModuleList([HandwritingRecognitionModel() for \_ in range(num\_models)])

    def forward(self, x):

        return torch.mean(torch.stack([model(x) for model in self.models]), dim=0)

**4.Performance Visualization**

In this section, we compare performance under different settings, as required. Specifically, we examine:

1. **Changing the number of epochs** to see how the Fast\_Mode works in the project.
2. **Changing the loss function** to see how the training curves differ.
3. **Changing the learning rate** to see how quickly or stably the model converges for each learning rate.
4. **Changing the batch size** to see the effect on training speed, fluctuation in accuracy, and final performance.

Below, we describe each of these experiments in more detail.

**4.1 Baseline**

**Setup**:

* **Loss Function**: Cross-Entropy
* **Optimizer**: Adam
* **Learning Rate**: 0.001
* **Batch Size**: 16
* **Number of Epochs**: 5 and 10

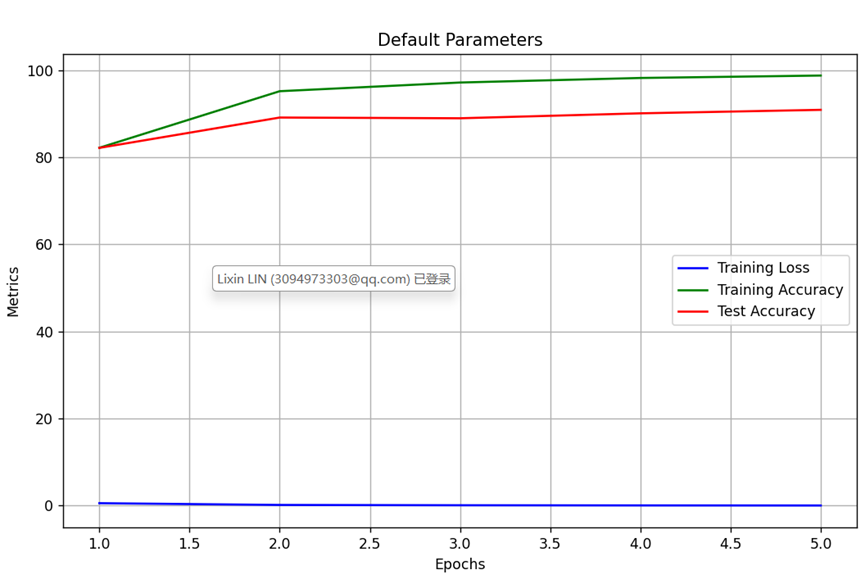


Figure 4.1

When epoch=5, just as Figure 4.1 shows, the model achieved a test accuracy of 92.77% after 5 epochs.

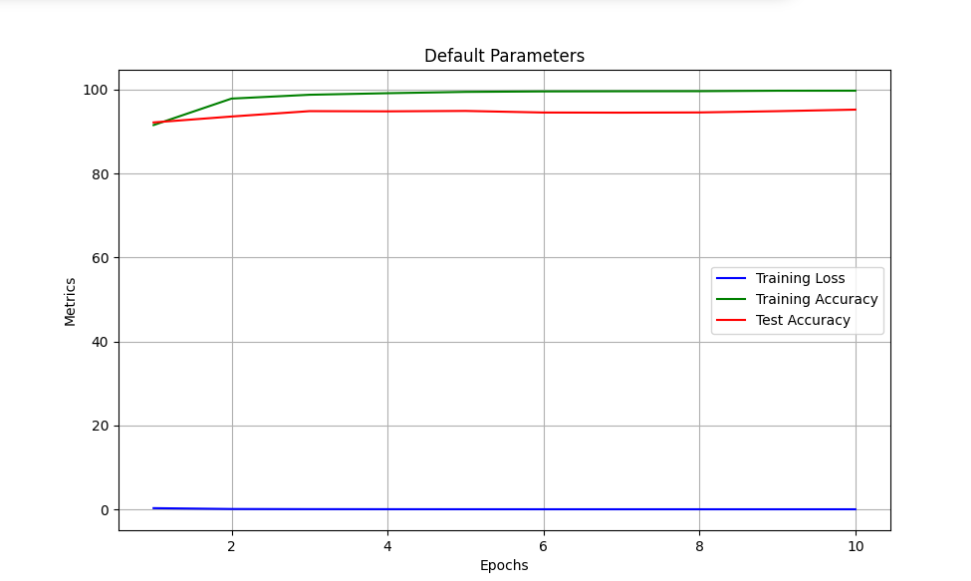


Figure 4.2

When epoch=10, just as Figure 4.2 shows, the model achieved a test accuracy of 95.28% after 10 epochs.

**4.2 Different Loss Function Experiments**

Next, we compare the performance of the model with Cross-Entropy Loss and MSELoss.

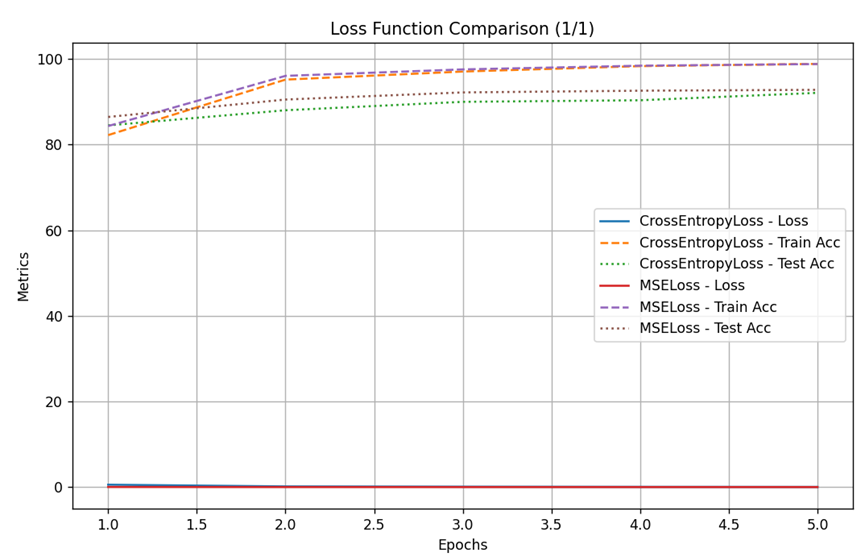


Figure 4.3

When epoch=5, just as Figure 4.3 shows, the final test accuracy using CrossEntropyLoss was 92.07% and the test accuracy using MSELoss was 92.77%. Obviously, MSELoss performed slightly better than CrossEntropyLoss on this task.

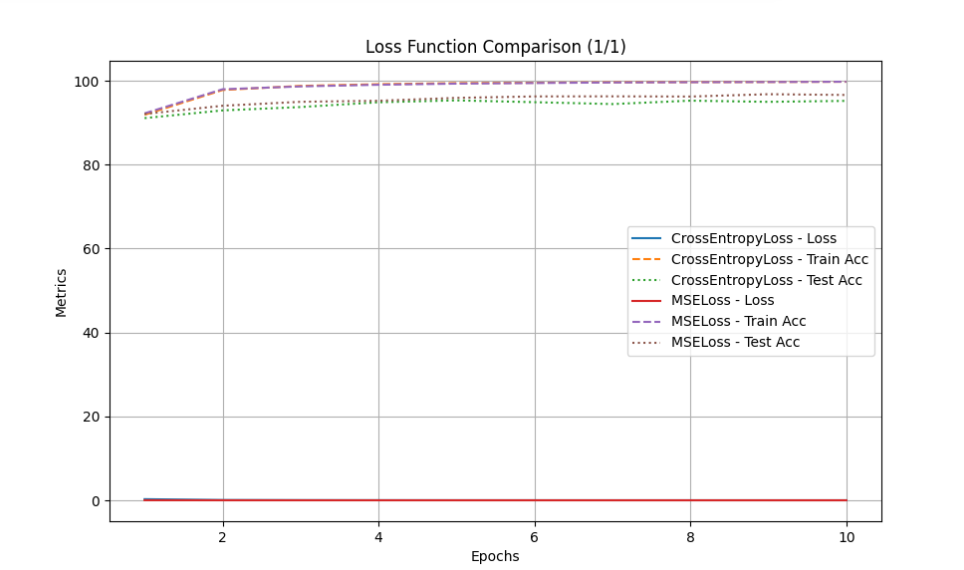


Figure 4.4

When epoch=10, The final test accuracy using CrossEntropyLoss was 95.17%. And the accuracy using MSELoss was 96.54%. And MSELoss also outperformed CrossEntropyLoss slightly on this task.

**4.3 Different Learning Rates Experiments**

This part focus on how changing the learning rate influences training dynamics. We keep the same baseline model architecture, Cross-Entropy Loss, Adam optimizer, and batch size (16), but vary the learning rate. Let us test four learning rates: 0.1, 0.01, 0.001, and 0.0001.

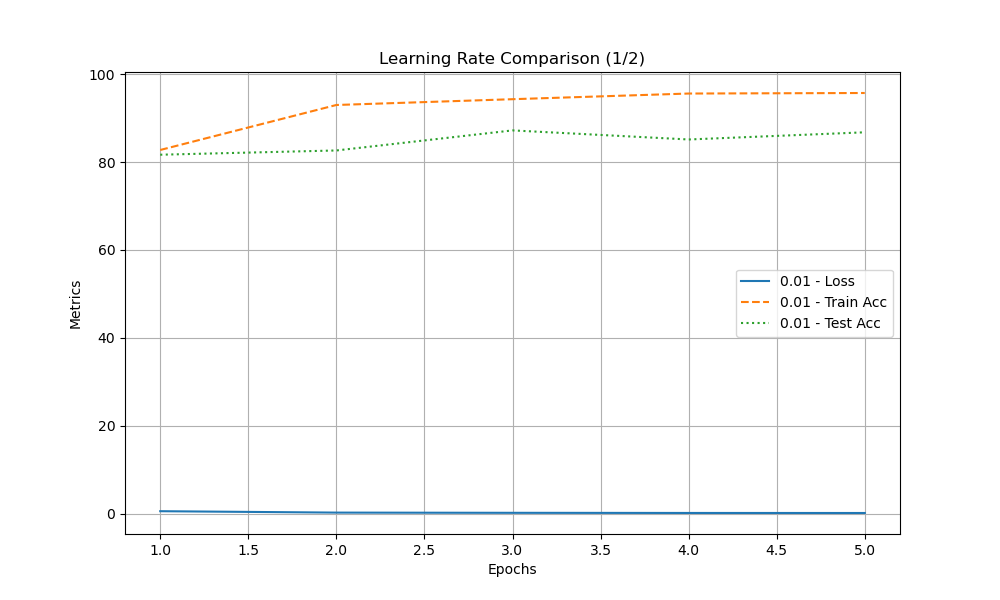


Figure 4.5

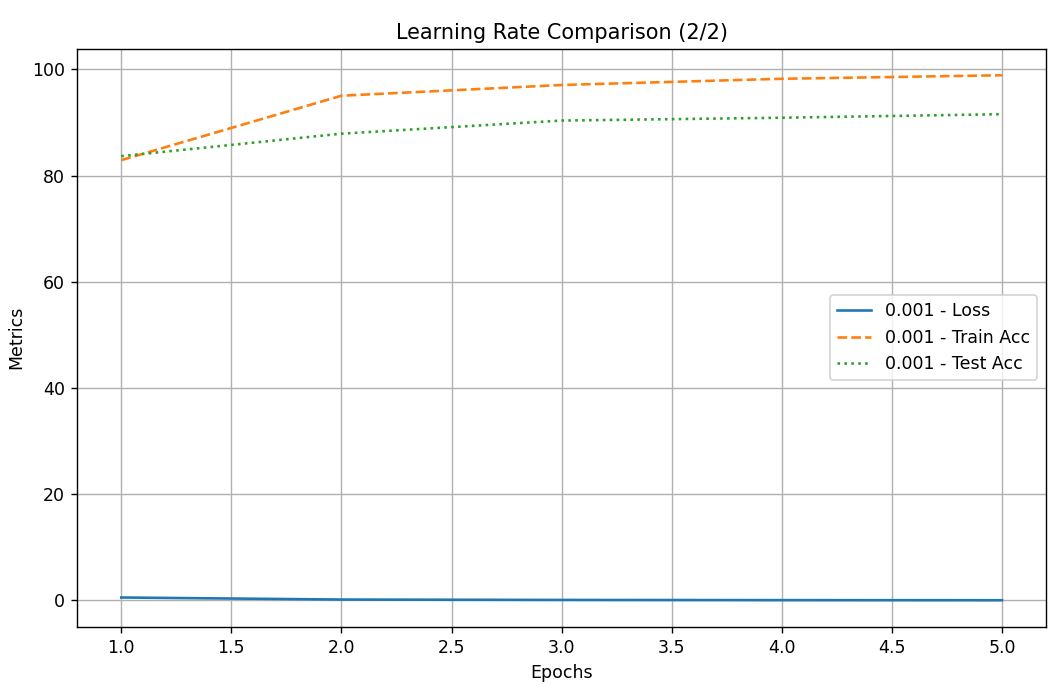


Figure 4.6

When epoch=5, just shown as Figure 4.5 and Figure 4.6, the smallest learning rate (0.001) performed better, possibly because it allowed the model to converge more stably to an optimal solution.

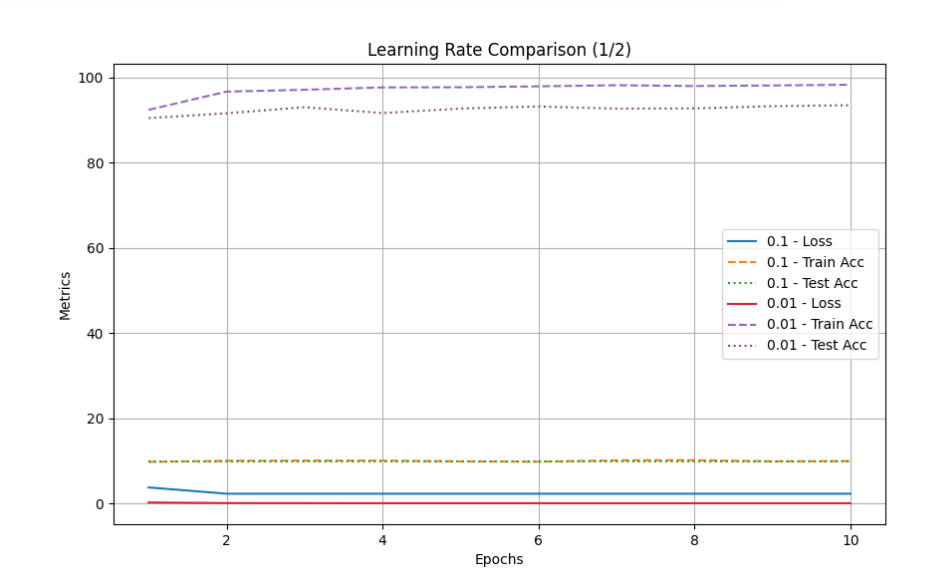


Figure 4.7

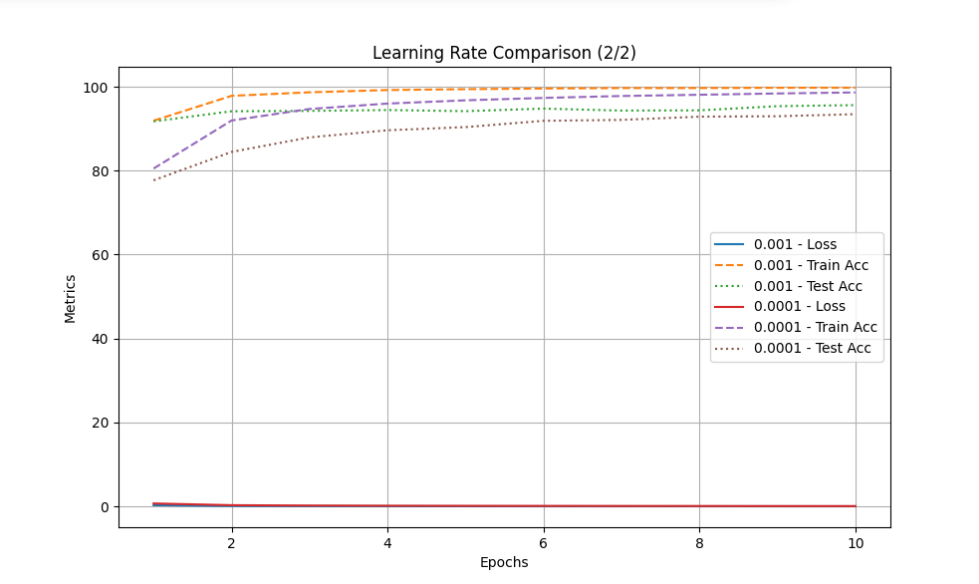


Figure 4.8

When epoch=10, just shown as Figure 4.7 and Figure 4.8, the smallest learning rate (0.001) also performed best.

**4.4 Different Batch Sizes Experiments**

Now the part investigate how batch size affects the training process. We keep the baseline settings (Cross-Entropy Loss, Adam, LR=0.001) but vary batch sizes across several values ( 8, 16, 32, 64).

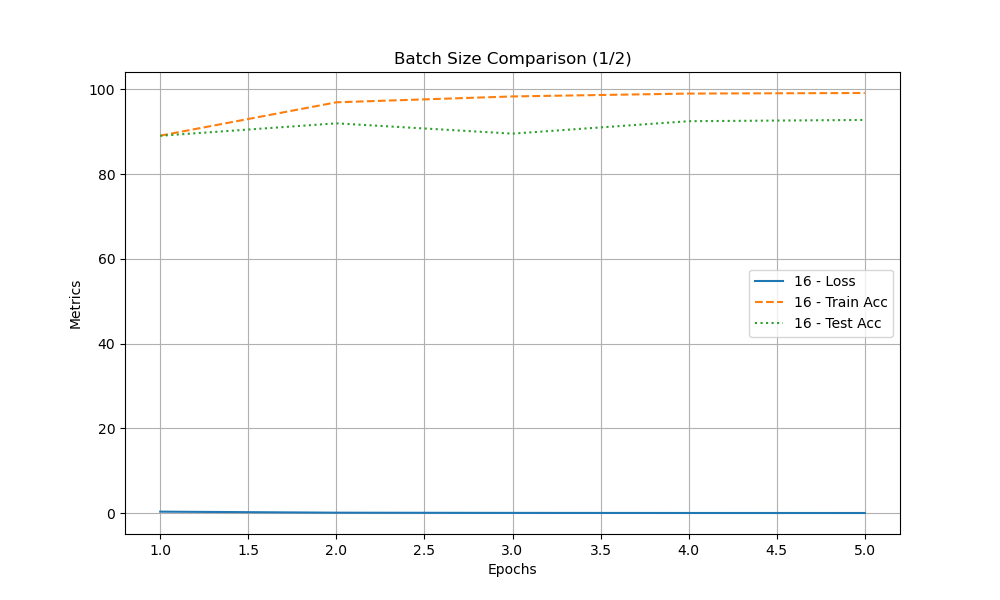


Figure 4.9

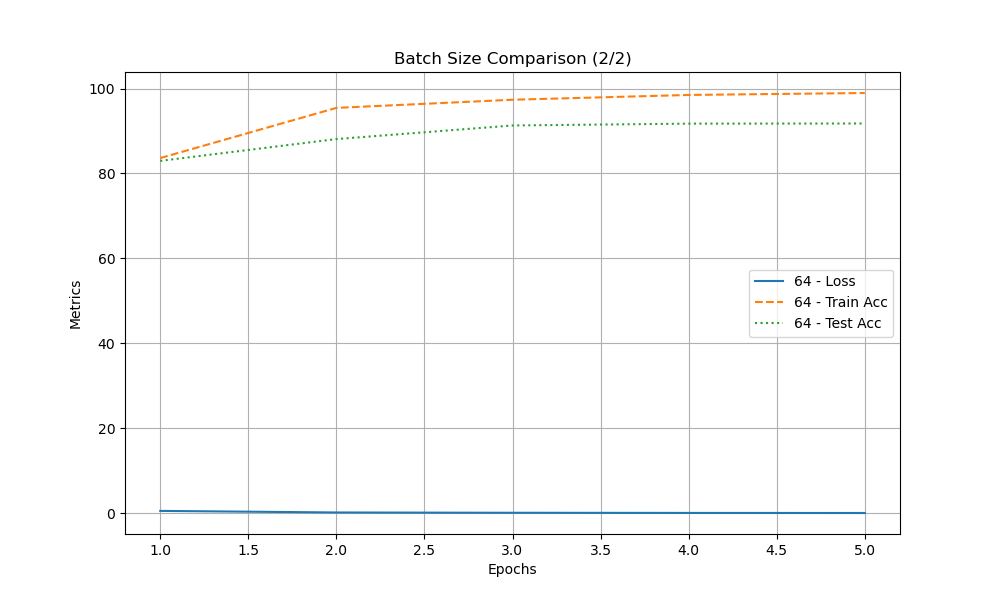


Figure 4.10

When epoch=5, as shown in Figure 4.9 and Figure 4.10, The smaller batch size (16) performed better with accuracy 92.77%, possibly because it enabled more frequent parameter updates, helping the model find a better local optimum.

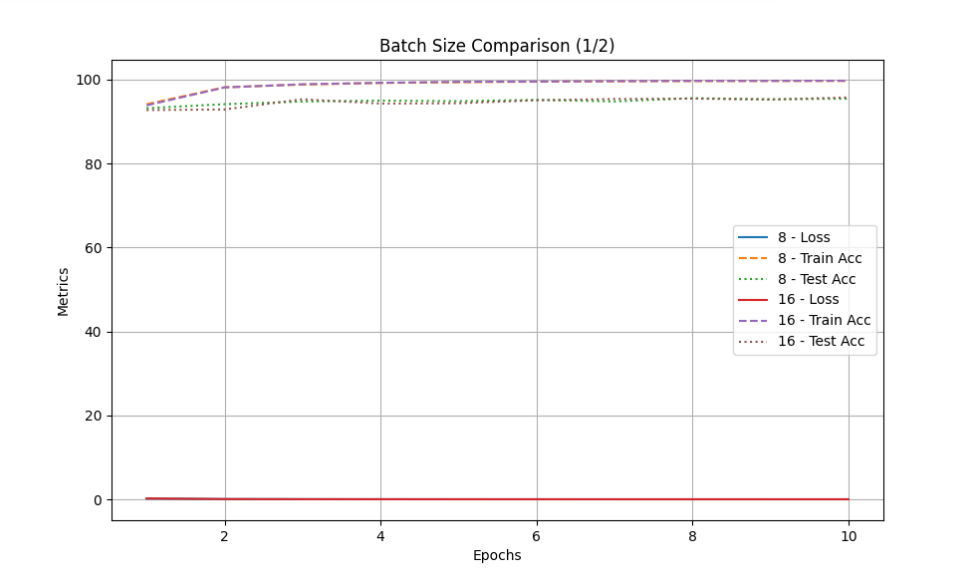


Figure 4.11

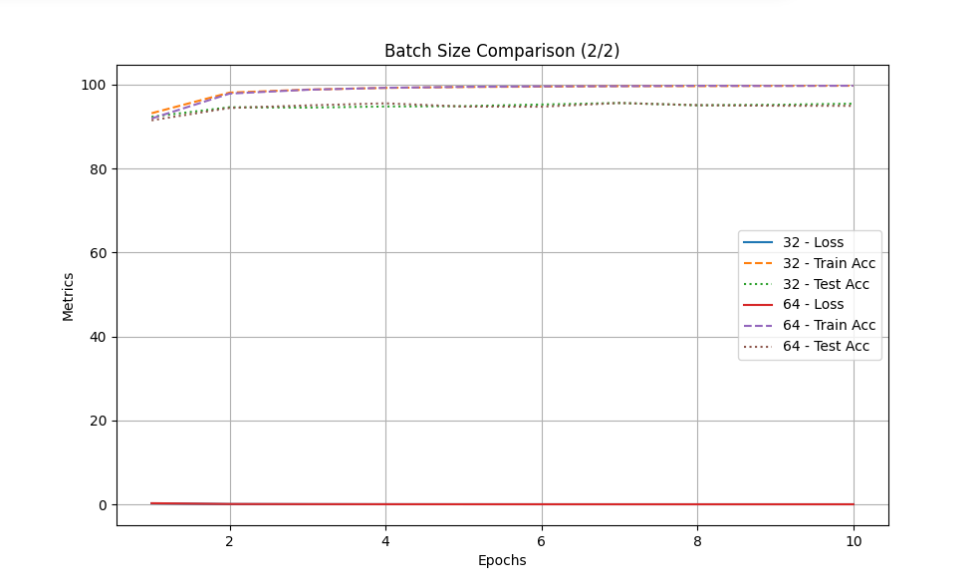


Figure 4.12

When epoch=10, Just as Figure 4.11 and Figure 4.12 shows, the Batch Size Comparison are as follows:

* 8: Final test accuracy was 94.88%
* 16: Final test accuracy was 95.76%
* 32: Final test accuracy was 95.14%
* 64: Final test accuracy was 94.98%

A batch size of 16 performed the best, possibly because it strikes a good balance between computational efficiency and gradient estimation accuracy.

**4.5 Visualizing Predictions**

Based on the above experiments, the optimal parameter combination is:

* Loss Function: MSELoss
* Learning Rate: 0.001
* Batch Size: 16

And Best model performance on the test set are:

* Epoch = 5: Accuracy = 92.77%
* Epoch = 10: Accuracy = 95.28%

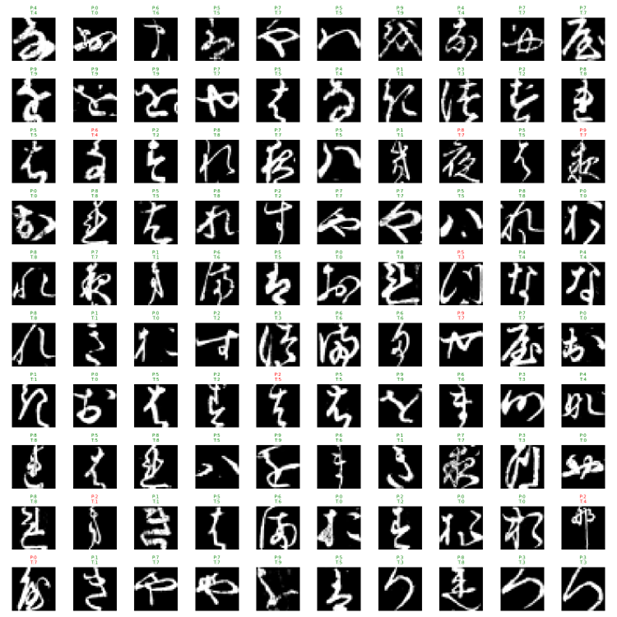


Figure 4.13

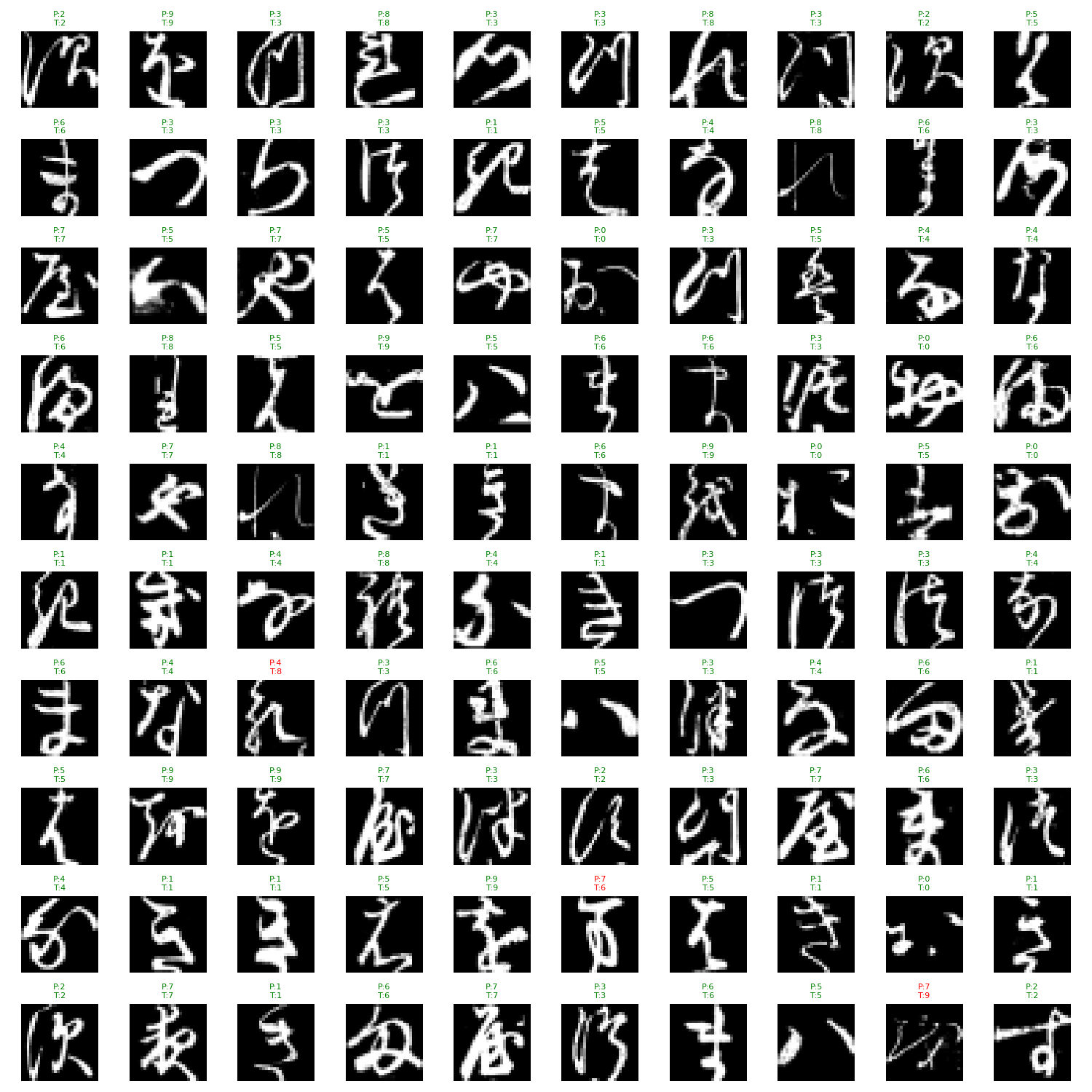


Figure 4.14

The Figure 4.13 and Figure 4.14 show the recognition outcomes of the model when epoch=5 and epoch=10.

**5. Conclusion**

This report has presented a detailed overview of the handwriting recognition project as implemented in *Handwriting\_Recognition.py*. We began by discussing the importance of data preprocessing, including normalization and potential augmentation steps, which are crucial for stable and generalized model performance. We then delved into the rationale for selecting a convolutional neural network, explaining how convolutional and pooling layers help extract relevant features from handwritten images.

The choice of Cross-Entropy Loss was justified by its ubiquitous use for multi-class classification, though we have experimentally showcased that other loss functions like MSELoss actually performs better in this project.Regarding optimization, Adam was highlighted for its adaptive learning rate strategy, often leading to faster convergence in comparison to classical stochastic gradient descent. And hyperparameters, including batch size, learning rate, and the number of epochs, were extensively explored. We found that moderate batch sizes (16) and an Adam learning rate of around 0.001 provided a good balance between efficiency and stability for this task. Nevertheless, adjustments may be needed for different datasets or more complex tasks.

In terms of performance visualization, we discussed the typical appearance of training loss and accuracy curves over epochs, illustrating the model’s learning process. Changing the loss function, learning rate, and batch size can significantly influence these curves, offering insights into potential optimizations. Finally, we demonstrated the performance on the test set alongside their true labels, allowing for a direct examination of the model’s strengths and weaknesses.

In summary, this project provides a strong foundation for handwriting recognition through a CNN-based approach. By systematically experimenting with key components—model architecture, loss function, optimizer, and hyperparameters—we have demonstrated how to build, train, and evaluate an effective handwriting recognition system. The methodology and insights gained here can be adapted for a wide variety of image-based classification tasks well beyond digit recognition, underscoring the versatility and power of deep learning in computer vision.

**6.Citation**

[1] Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., & Ha, D. (2018). Deep learning for classical Japanese literature. arXiv preprint arXiv:1812.01718.

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

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

[4] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. International conference on machine learning (pp. 448-456). PMLR.

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

[6] Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 1-48.

[7] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[8] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

[9] Smith, L. N. (2017). Cyclical learning rates for training neural networks. 2017 IEEE Winter Conference on Applications of Computer Vision (WACV) (pp. 464-472). IEEE.

[10] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.