M1 Coursework Report

Zhimei Liu (zl474)

December 18, 2024

Word count: 2234

Declaration: I, Zhimei Liu, hereby declare that the work presented in this report and the GitLab repository is entirely my own. I have only used ChatGPT for some debugging to speed up the process.

Contents

0 Introduction

In this coursework, I aimed to develop and optimize a neural network model capable of classifying digit pairs from the MNIST dataset based on the sum of their labels. The task involved converting the original MNIST dataset into a new format containing paired digit images, where each pair corresponds to a specific sum ranging from 0 to 18. This posed unique challenges, particularly in the design of the dataset, model architecture, training process, and fine-tuning of hyperparameters.

To address the challenges, I employed a *fully connected neural network* (FCNN) to classify the digit pairs. The network was trained and fine-tuned using advanced techniques, including hyperparameter optimisation with Optuna. Hyperparameters such as the number of hidden layers, learning rates, batch sizes, dropout rates and activation functions were systematically tuned to minimize validation loss and achieve the best model performance. Additionally, careful data preprocessing and sampling strategies were applied to ensure the appropriate statistical properties are guaranteed.

This report documents the entire process, from data preprocessing and model design to hyperparameter optimization and performance evaluation. The results demonstrate the effectiveness of the implemented techniques in achieving accurate and robust classification of digit pairs.

All functions in this coursework are defined in the notebook named `M1coursework.ipynb`.

Sections of this report are divided corresponding to the questions listed.

1 Dataset Preparation

1.1 Dataset Overview

The MNIST dataset was chosen as the basis for this project due to its simplicity, popularity in machine learning research, and suitability for the task of handwritten digit recognition. The MNIST dataset contains 70,000 grayscale images of handwritten digits (0-9), each of size 28×28 pixels. Every image is paired with a label corresponding to the digit it represents. The dataset is divided into 60,000 training samples and 10,000 test samples.

1.2 Data Transformation

To adapt the MNIST dataset for this project, pairs of digit images were generated, with each pair assigned a label equal to the sum of the two digits. The transformation involved the following steps:

- **Pairing images**: Two images were randomly sampled from the dataset. Their pixel data were concatenated vertically to create a new image of size 56×28.
- **Assigning labels**: The labels of the individual images were summed to form the label for the new paired image.

This process resulted in a dataset suitable for training a neural network to classify the sum of paired digits, with labels ranging from 0 to 18.

Figure 1 shows an example of a transformed image.

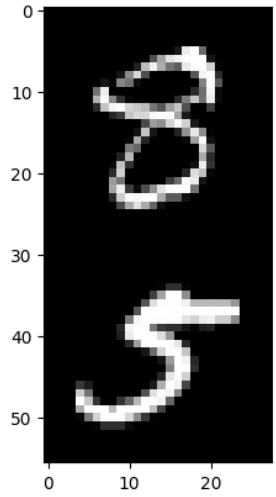

Figure 1: An example of a transformed image. Two MNIST images with labels 8 and 5 respectively were concatenated to form an image of size 56×28. The label for this transformed image is $8+5 = 13$.

1.3 Dataset Preparation

To prepare for the required dataset, I selected data from the 60,000 MNIST training set images. The sizes of the transformed training set, validation set and test set are 70,000, 15,000 and 15,000 respectively.

The distribution of labels was obtained by randomly choosing two images from the MNIST dataset and then concatenating them together with the function `data_converter`. The transformed dataset has the bell-shaped distribution across 19 labels as shown in Figure 2 for all training set, validation set and test set. The shape of the distribution makes sense because for example to get a sum of 9, there are more possibilities to combine two digits, resulting in more data. During training, we also need more data for sums equal to digits in the middle of the range $[0, 18]$ for classification, as there are more combinations for the model to identify.

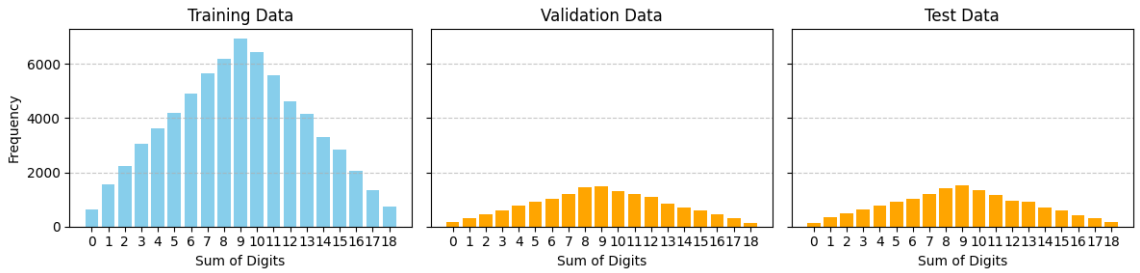

Figure 2: The distribution of transformed training dataset, validation dataset and test dataset without any constraints on the size of each label.

2 Fully Connected Neural Networks

In this section, I developed a neural network pipeline using *fully connected neural networks* (FCNN) to classify the sum of two digits from the MNIST dataset. The goal was to design an architecture that can infer the sum of two digits, where the neural network outputs a single number corresponding to the sum of the two digits (e.g., $7 + 3 = 10$). I performed hyperparameter tuning with Optuna to optimize the model architecture, testing a range of hyperparameters to find the best performing setup for the task. Due to computational limitations, I restricted the experiments to a small number of trials, ensuring that the process would be completed within a few hours on a personal laptop.

2.1 Model Architecture

The FCNN model is a simple feedforward neural network designed to process input data and generate predictions for classification or regression tasks. It is implemented using PyTorch's `nn.Module` and consists of multiple fully connected layers with an option to apply different activation functions and regularization techniques like dropout.

To build up a FCNN, my model consisted of the following parameters:

```
input_size = 56 * 28  # Flattened image size
hidden_layers_trial = [512, 256]  # Number of neurons in hidden layers
learning_rate_trial = 0.001
```

```
    batch_size_trial = 64
    dropout_rate_trial = 0.2
    activation_func_trial = nn.ReLU() # Activation function
    output_size = 19
    epochs = 20
    criterion = nn.CrossEntropyLoss() # Loss function
```

The code for the construction of the FCNN is as follows:

```
class FCNN(nn.Module):
    def __init__(self,
    input_size, hidden_layers, output_size, dropout_rate, activation_func):
        super(FCNN, self).__init__()
        layers = []
        prev_size = input_size

        for hidden_size in hidden_layers:
            layers.append(nn.Linear(prev_size, hidden_size))
            layers.append(activation_func)
            layers.append(nn.Dropout(dropout_rate))
            prev_size = hidden_size

        layers.append(nn.Linear(prev_size, output_size))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

    def extract_embeddings(self, x):
        """
        Extract embeddings after the last hidden layer (before the final output
            layer) for Question 5.
        The method returns the activations from the last hidden layer.

        Parameters:
        - x: The input tensor to the network.

        Returns:
        - The embeddings (feature map) after the last hidden layer.
        """
        # Pass through all layers except for the final output layer
        for layer in self.network[:-1]:  # Exclude the last layer (the output layer)
            x = layer(x)
        return x
```

and the summary of the structure of the trial model is

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [64, 1, 512]         803,328
              ReLU-2               [64, 1, 512]               0
           Dropout-3               [64, 1, 512]               0
            Linear-4               [64, 1, 256]         131,328
              ReLU-5               [64, 1, 256]               0
           Dropout-6               [64, 1, 256]               0
```

```
            Linear-7                [64, 1, 19]           4,883
================================================================
Total params: 939,539
Trainable params: 939,539
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.38
Forward/backward pass size (MB): 1.13
Params size (MB): 3.58
Estimated Total Size (MB): 5.10
----------------------------------------------------------------
```

Then I run the function `train_model` to train with the above parameters and the optimizer I used is `Adam`.

After training for 25 epochs, the plot for training and validation losses as a function of number of epochs and the plot for training and validation accuracies as a function of number of epochs are shown in Figure 3.

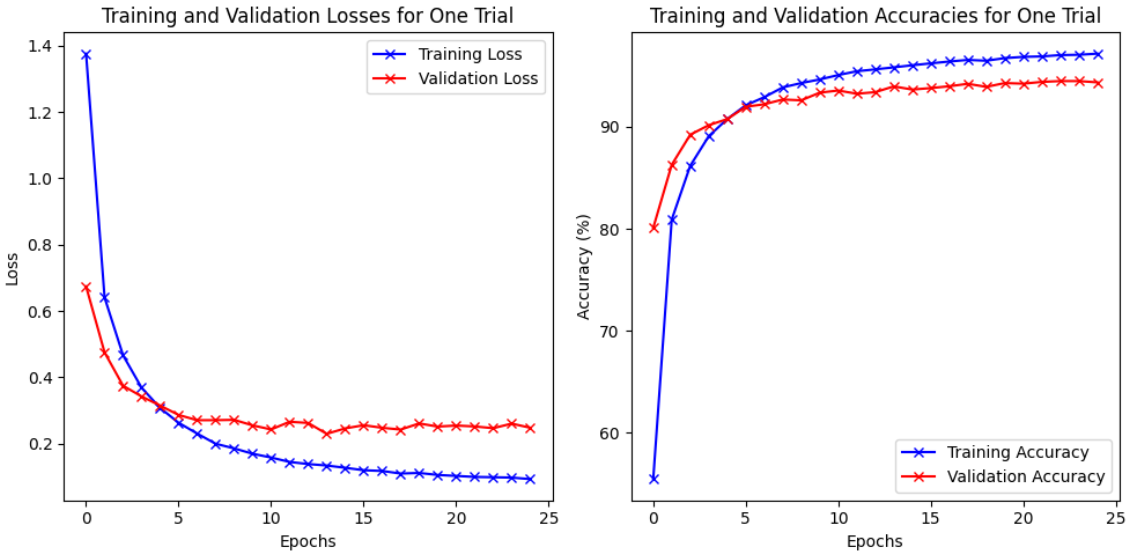

Figure 3: The left figure shows the training and validation losses for this trial model as a function of the number of epochs; The right figure shows the training and validation accuracies for this trial model as a function of the number of epochs.

We can see that after this trial model is trained for 25 epochs, the training and validation losses and accuracies tend to converge and become relatively stable. After 25 epochs, the final training loss is 0.0933 and the final validation loss is 0.2481. The final training accuracy is 97.17% and the final validation accuracy is 94.36%. I also calculated the test accuracy of this trial model with the test set, using the function called `test_model` in the notebook `M1coursework.ipynb`, and it gives an accuracy of 94.19%.

The results demonstrate that this trial model is good enough to perform classification, however, since the final training loss is small compared to the final validation loss, it suggests that the model is probably overfitting. To deal with this problem, I perform hyperparameter tuning in the following subsection.

2.2 Hyperparameter Tuning

The five hyperparameters I was tuning with `Optuna` are hidden layers, learning rate, batch size, dropout rate and activation functions. The ranges of values I tried for the hyperparameters are the following:

```
hidden_layers = trial.suggest_categorical('hidden_layers', [[128, 64], [256, 128],
    [512, 256, 128], [512, 256, 128, 64]])
learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-1, log=True)
batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])
dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.4)
activation_name = trial.suggest_categorical('activation', ['ReLU', 'Tanh',
    'Sigmoid'])
```

The `objective` function in the notebook `M1coursework.ipynb` was defined by minimising the validation loss. After running 50 trials with `Optuna`, the best trial model has the following validation loss and parameters:

```
Best trial:
  Value (best validation loss): 0.21500527630064717
  Params: {'hidden_layers': [512, 256, 128, 64], 'learning_rate': 0.000633489537397628,
  'batch_size': 64, 'dropout_rate': 0.17439213235063694, 'activation': 'ReLU'}
```

We can see that the best validation loss is indeed smaller than the validation loss from the previous trial model. When I test the accuracy of the best model with optimised batch size on the test set, I get the accuracy of 95.05%, which is higher than the previous accuracy 94.19%.

The weights of the best model are saved in the file called `trained_best_model_weights.pth` and the best hyperparameters are saved in the file named `optuna_study_5params.pkl`.

Note however that I can fine tune more hyperparameters such as decaying rate, or splitting the parameter `hidden_layers` into two hyperparameters: number of layers and number of units for the hidden layer. But due to the limitations of personal laptop and requirement of the question, I choose the above five hyperparameters to perform fine tuning.

2.3 Analysis of the Best Model

The best model after hyperparameter tuning has the following structure:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [64, 1, 512]         803,328
              ReLU-2               [64, 1, 512]               0
           Dropout-3               [64, 1, 512]               0
            Linear-4               [64, 1, 256]         131,328
              ReLU-5               [64, 1, 256]               0
```

```
           Dropout-6                [64, 1, 256]                0
            Linear-7                [64, 1, 128]           32,896
              ReLU-8                [64, 1, 128]                0
           Dropout-9                [64, 1, 128]                0
           Linear-10                 [64, 1, 64]            8,256
             ReLU-11                 [64, 1, 64]                0
          Dropout-12                 [64, 1, 64]                0
           Linear-13                 [64, 1, 19]            1,235
================================================================
Total params: 977,043
Trainable params: 977,043
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.38
Forward/backward pass size (MB): 1.42
Params size (MB): 3.73
Estimated Total Size (MB): 5.53
----------------------------------------------------------------
```

The plots for the training and validation losses and the training and validation accuracies as a function of the number of epochs are shown in Figure 4.

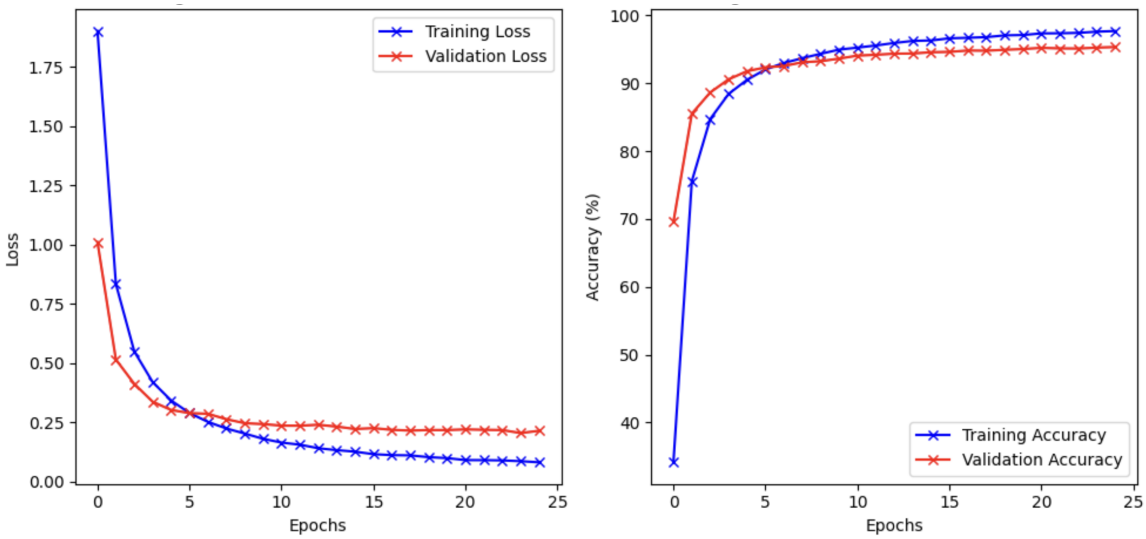

Figure 4: The left figure shows the training and validation losses for the best model as a function of the number of epochs; The right figure shows the training and validation accuracies for the best model as a function of the number of epochs.

Let's also visualise the performance of the best model from the perspective of the *confusion matrix*. It compares the true labels with the predicted labels generated by the model. Each row represents the instances in a actual class, while each column represents the instances in an predicted

class.

Figure 5 is the confusion matrix of the best model. In the confusion matrix, the diagonal cells show the numbers of the correct predictions, while the off-diagonal cells display the number of misclassifications. We can see that the best model has good accuracy across each label.

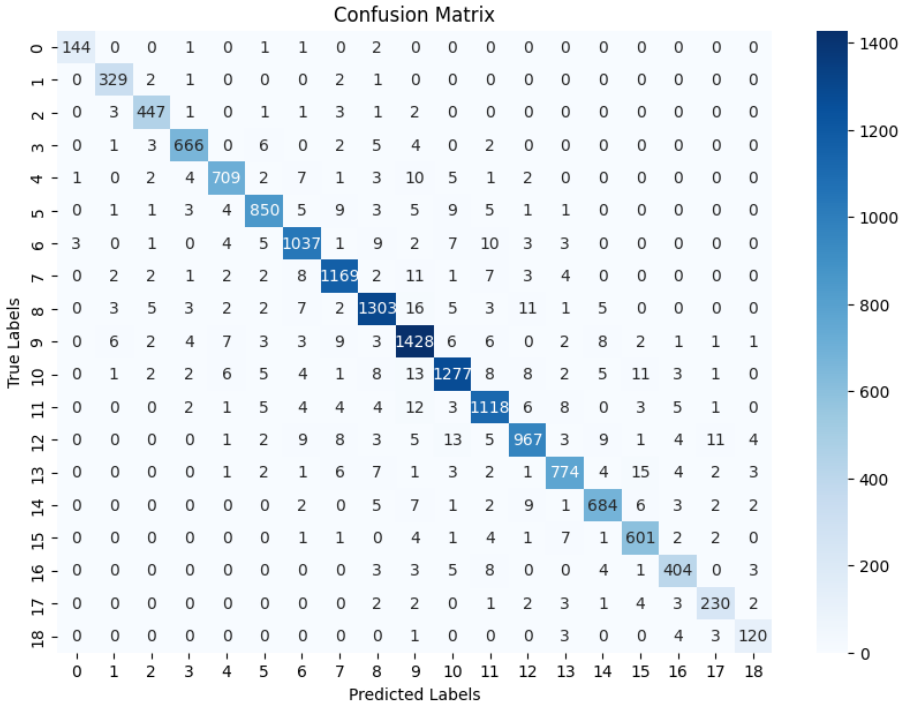

Figure 5: Confusion matrix of the best model.

3 Other Inference Algorithms

In this section, we compare the performance using other inference algorithms implemented in `scikit-learn`, including random forest classifiers and support vector machines.

I used a training set of size 30,000 and a test set of 6,000.

3.1 Random Forest Classifiers

The performance of the random forest classifier on the test set after training is as follows:

```
Random Forest Classifier Performance:
Accuracy: 0.6732
              precision    recall  f1-score   support

           0       0.93      0.98      0.95        63
```

1	0.84	0.93	0.89	169
2	0.83	0.86	0.84	165
3	0.82	0.74	0.78	286
4	0.76	0.74	0.75	270
5	0.76	0.71	0.73	372
6	0.69	0.73	0.71	415
7	0.74	0.73	0.74	505
8	0.67	0.71	0.69	560
9	0.60	0.73	0.66	610
10	0.59	0.64	0.61	561
11	0.59	0.63	0.61	450
12	0.55	0.63	0.59	422
13	0.60	0.60	0.60	313
14	0.73	0.47	0.57	286
15	0.72	0.55	0.62	232
16	0.66	0.47	0.55	176
17	0.78	0.33	0.46	98
18	0.81	0.28	0.41	47
accuracy			0.67	6000
macro avg	0.72	0.66	0.67	6000
weighted avg	0.68	0.67	0.67	6000

Let's first define the concepts appeared in the above table. Firstly, **precision** measures the proportion of correctly predicted positive instances out of all instances that were predicted as positive. The formula to calculate precision is

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}. \tag{1}$$

Therefore, high precision means that most predictions for a given class are correct.

The definition of **recall** is that it measures the proportion of correctly predicted positive instances out of all actual positive instances in the dataset. The formula for recall is

$$\text{Recall} = \frac{\text{True Positive (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}. \tag{2}$$

High recall means that the model is capturing most of the true positive cases.

The **F1-score** is the harmonic mean of precision and recall. It balances the tradeoff between the two metrics with the following formula

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{3}$$

High F1-score means that the model has a good balance between precision and recall.

Lastly, **support** is the number of actual instances in the dataset for each class, i.e., the size of the ground truth for each class.

From the above table, we can see that on a training set of size 30,000, the overall accuracy is 67.32%. There are some key observations we can make:

- High precision and recall for the majority of classes with balanced data (e.g., Classes 0, 1, 2).
- Struggles with minority classes (e.g., Classes 17 and 18), leading to low F1-scores for these categories.
- Performance degrades for larger and more imbalanced classes (e.g., Class 9).

3.2 Support Vector Machines (SVM)

The performance of the SVM on the test set after training is as follows:

```
Cross-validation scores: [0.72866667 0.736      0.72566667 0.74633333 0.72966667]
Mean CV Accuracy: 0.7333
Support Vector Machine Performance:
Accuracy: 0.6987
              precision    recall  f1-score   support

           0       0.97      0.98      0.98        63
           1       0.85      0.93      0.89       169
           2       0.87      0.91      0.89       165
           3       0.83      0.77      0.80       286
           4       0.80      0.79      0.79       270
           5       0.75      0.74      0.74       372
           6       0.72      0.73      0.73       415
           7       0.72      0.71      0.71       505
           8       0.70      0.68      0.69       560
           9       0.66      0.71      0.68       610
          10       0.65      0.66      0.66       561
          11       0.61      0.69      0.65       450
          12       0.59      0.61      0.60       422
          13       0.59      0.64      0.61       313
          14       0.67      0.58      0.62       286
          15       0.78      0.63      0.70       232
          16       0.68      0.60      0.64       176
          17       0.73      0.60      0.66        98
          18       0.85      0.60      0.70        47

    accuracy                           0.70      6000
   macro avg       0.74      0.71      0.72      6000
weighted avg       0.70      0.70      0.70      6000
```

From the table, it is clear that the overall accuracy is 69.87%, with a cross-validation accuracy (mean CV score) of 73.33%. The following observations can be made:

- SVM demonstrates better generalization due to the use of cross-validation.
- High precision and recall for smaller classes (e.g., Classes 0, 1, and 2).
- Still underperforms in minority or imbalanced classes (e.g., Classes 17 and 18).

3.3 Neural Network Performance

The performance of the neural network from the previous best model on the test set after training is as follows:

```
Neural Network Performance:
Test Accuracy: 0.9524
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       149
           1       0.97      0.97      0.97       335
           2       0.96      0.96      0.96       459
           3       0.98      0.96      0.97       689
           4       0.97      0.95      0.96       747
           5       0.95      0.95      0.95       897
           6       0.95      0.96      0.96      1085
           7       0.95      0.96      0.96      1214
           8       0.96      0.95      0.96      1368
           9       0.95      0.94      0.95      1492
          10       0.95      0.95      0.95      1357
          11       0.94      0.95      0.95      1176
          12       0.95      0.95      0.95      1045
          13       0.95      0.95      0.95       826
          14       0.95      0.96      0.95       724
          15       0.94      0.96      0.95       625
          16       0.94      0.96      0.95       431
          17       0.92      0.90      0.91       250
          18       0.93      0.94      0.94       131

    accuracy                           0.95     15000
   macro avg       0.95      0.95      0.95     15000
weighted avg       0.95      0.95      0.95     15000
```

The overall accuracy of the above neural network is 95.24%. We can observe the following:

- Consistently high precision, recall, and F1-scores across all classes.
- Robust performance even in minority classes (e.g., Classes 17 and 18).
- Handles class imbalances more effectively than Random Forest and SVM due to its ability to learn complex patterns in the data.

3.4 Comparison Between Different Algorithms

Having looked at the individual performance of the above three inference algorithms, let's make some comparisons between them.

1. **Random Forest:**
    - Performs adequately but struggles with imbalanced data.

- Simple and interpretable but lacks flexibility for complex patterns.

2. **SVM:**
   - Outperforms Random Forest, particularly for moderately sized classes.
   - Computationally intensive for large datasets.
3. **Neural Network:**
   - The best-performing model in terms of accuracy and handling imbalanced classes.
   - Requires more training time and resources compared to other methods.

4 Weak Linear Classifier

In this section, firstly I trained the linear classifier on the 56×28 paired-image dataset with different sizes of training samples including 50, 100, 500, 1000, 2000, 5000 and 10000. The function `train_linear_classifier_on_paired_data` performs this task. I got the following accuracies as the size of the training set varies:

Training set size	50	100	500	1000	2000	5000	10000
Accuracy	0.1071	0.1158	0.1222	0.1376	0.1571	0.1676	0.1839

Next, I trained the linear classifier applied on the two separated MNIST images sequentially. The function `train_linear_classifier_on_two_images_sequentially` performs the task. The accuracies as the size of the training set varies are as follows:

Training set size	50	100	500	1000	2000	5000	10000
Accuracy	0.3778	0.4548	0.6851	0.7227	0.7482	0.7925	0.8178

It is clear that when the classifier is applied to the two images sequentially, the accuracies are relatively higher than applying the classifier on the concatenated images. Figure 5 shows the accuracy curves from the two methods as the training set size varies.

4.1 Key Observations and Analysis

When the linear classifier is applied on two images sequentially, the model processes each image independently, extracting linear features from each image. Since the images are evaluated separately, the classifier has a better chance to identify discriminative features and assign probabilities effectively. The process of treating the two images separately preserves the input structure, enabling the classifier to generalize better even with fewer training samples.

When training directly on paired images (56×28), the model combines the two images into a single representation. This concatenation may mix the features of the two images, causing a loss of discriminative information. Essentially, the combined representation introduces additional complexity, which a linear model struggles to handle, particularly when trained on a small number of samples.

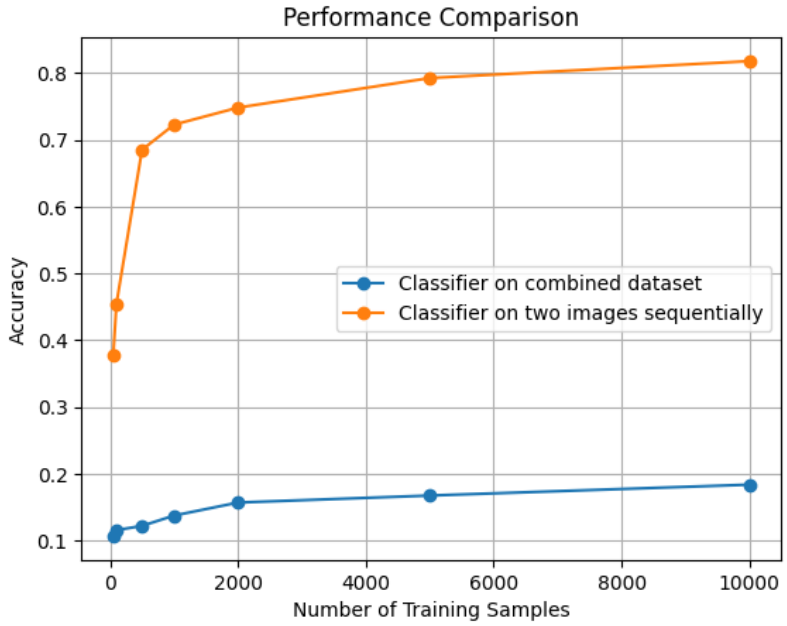

Figure 6: Comparison of accuracies between the two ways of applying the linear classifier as the size of the training set varies.

5 t-SNE of the Best Performing Neural Network

5.1 t-SNE Distribution in the Embedding Layer

From Figure 7, we can see that different classes cluster well (there are distinct regions for distinct classes). As the perplexity increases, different classes cluster more closely and are further apart. Therefore, we can see that for perplexity equals to around 5, the t-SNE plot behaves better by capturing distinct clusters while maintaining some sense of their relative distances.

This indicates that the neural network has successfully learned to transform the input data into a more structured and meaningful representation in the embedding space. The optimized embedding allows the network to better distinguish between classes, which aligns with the improved performance of the neural network.

5.2 t-SNE Distribution on the Input Dataset

Figure 8 shows the t-SNE distributions directly on the input dataset. As we can see, none of the plots produce distributions with distinct clustering of different classes. The points appear scattered with very little structure or separation between classes. Even as the perplexity increases, the separation between classes is not clearly visible. This suggests that in the original input space, the data lacks a clear structure or discriminative features that separate the various classes.

5.3 Conclusion

The comparison clearly demonstrates that the neural network's embedding layer produces a much more structured representation of the data, as evidenced by the well-separated clusters in the t-SNE visualization. This confirms that the model has successfully learned meaningful features that separate the different classes, while the input space lacks this structure.

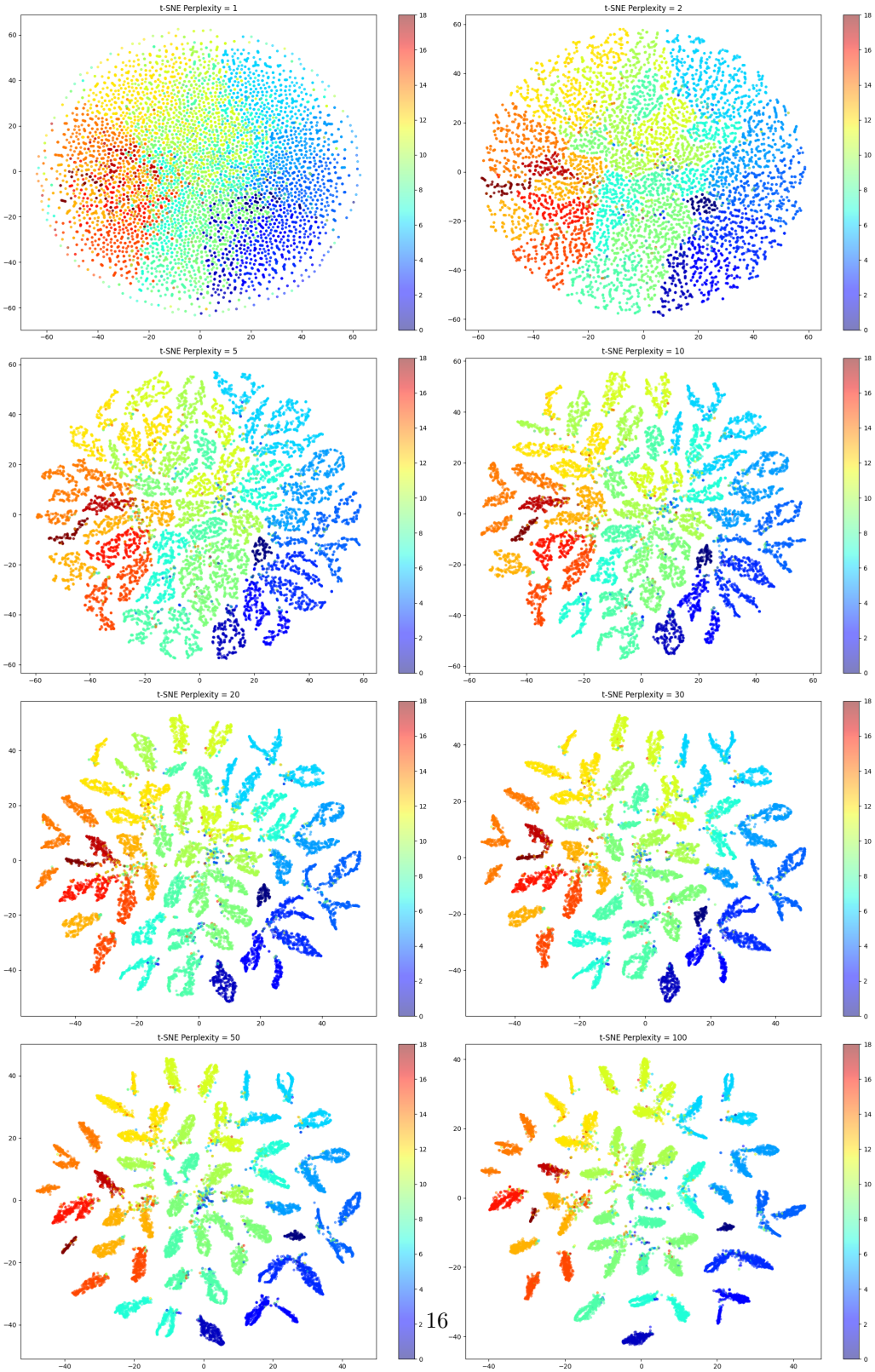

Figure 7: t-SNE distribution plots of the various classes in the **embedding layer** with different values of perplexity.

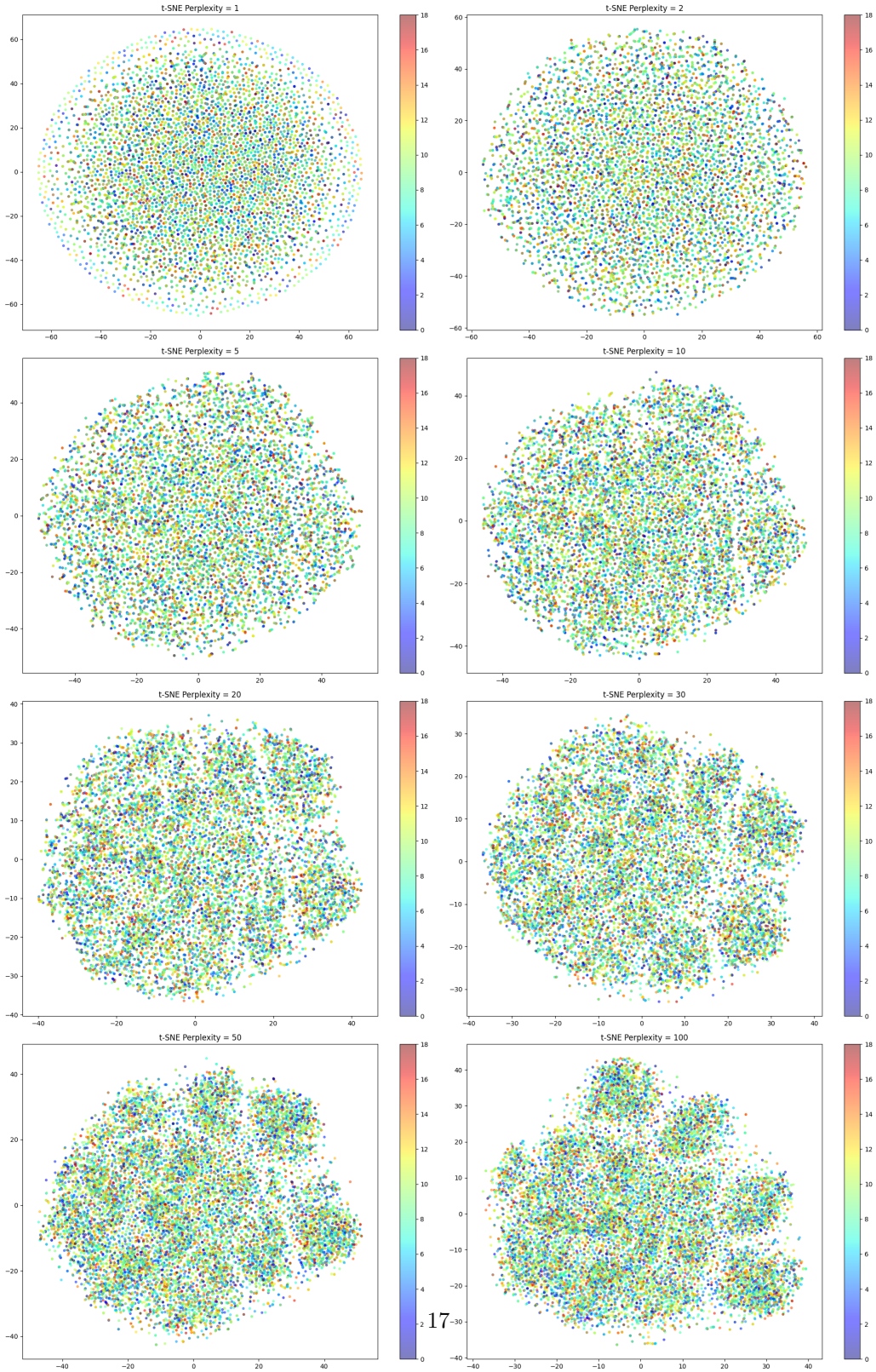

Figure 8: t-SNE distribution plots of the various classes on the **input dataset** with different values of perplexity.