M1 Coursework Report

Keying Song (ks2146)

18th December 2024

Word Count: 2,524 words

Abstract

In this coursework, a neural network pipeline for inferring the sum of two handwritten MNIST digits is built, evaluated and analysed. In addition, three other classifiers, random forest classifier, support vector classifier and linear classifier with two training methods are compared and shown. In order to have a clearer understanding of this fully-connected neural network model, I finally use the t-SNE technique to visualise the distribution of different classes in embedding and input layers, as well as the optimisation process for the perplexity.

Contents

1) Introduction

This essay reports on the implementation and evaluation of an inference pipeline which classifies the sum of two handwritten digits into labels 0 to 18. In the following section, I will provide the details of implementation process of the whole pipeline, which will begin with the balanced dataset construction with appropriate training, validation and test sets, followed by the design and training of a neural network for the classification task, then the comparison and analysis with other three methods will be made, as well as the demonstration of t-SNE distribution. Finally, a conclusion of the preceding paragraphs and the prospects for future research will be presented in Section 3.

2) Implementation process

2.1 Balanced Dataset Construction

2.1.1 Problem Statement

> Write code that allows you to construct a dataset which combines the images and provides the appropriate labels. The input images should be of shape 56×28. Generate appropriate training, validation, and test datasets. Justify your choices in the generation of these datasets, i.e. ensure that the appropriate statistical properties are guaranteed.

For the first step, I constructed a dataset named `combined_mnist.npz` by combining pairs of MNIST digits to create 56×28 pixel images as the features. The dataset generation process mainly involved two parts: building and splitting, while the appropriate statistical properties were guaranteed. In addition, the information of original label pairs in MNIST was also retained in the new dataset for subsequent t-SNE analysis.

2.1.2 Building and Splitting

In the building process, I firstly concatenated the original training and test sets in MNIST and randomly selected pairs of MNIST digits to form the features in the new dataset by stacking them horizontally. Then, each feature was attached with a label which was the sum of the two original numbers. The size of the new dataset was set at 100,000 samples for easy observation and future use.

Using the `train_test_split` function in `sklearn`, the new dataset was divided into training, validation and test datasets, accounting for 70%, 10% and 20% of the whole dataset respectively, see table(1) for the basic statistics of the new dataset. Additionally, the pixel values of the images were normalised to the range [0,1] to ensure uniformity and effective model training.

Table 1: Dataset Statistics

Subset	Size
Training set	55999
Validation set	24001
Test set	20000

Random seeds were set in both the construction and splitting parts to ensure the reproducibility of the dataset, see figure(1) for the first five samples in `combined_mnist.npz`.

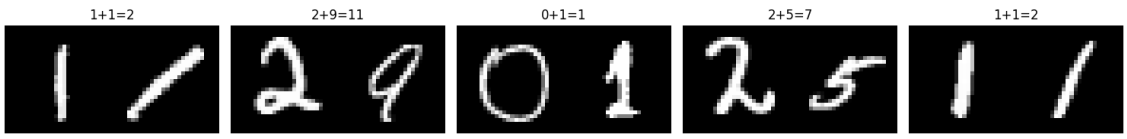


Figure 1: Visualisation of first five samples

2.1.3 Statistical Properties

In order to balance the sample size between the 19 labels (0 to 18), I created a max-sample counter for each label, so that I can decide whether to continue adding features to this label by judging whether the number of samples in it already has reached the upper limit (max-samples) or not.

```
# create a counter for each possible sum_lables (0-18)
counter = np.zeros(19)
max_samples = N // 19 + 1

while len(images) < N:
    # randomly select two indices
    i1, i2 = np.random.randint(0, len(x_data), size=2)
    img1, img2 = x_data[i1], x_data[i2]
    y_sum = y_data[i1] + y_data[i2]

    # check if more samples are needed for this sum
    if counter[y_sum] < max_samples:
        combined = np.hstack((img1, img2))
        images.append(combined)
        sum_labels.append(y_sum)
        original_pairs.append((int(y_data[i1]), int(y_data[i2])))
        counter[y_sum] += 1
```

With the core above, I achieved an even distribution of the labels. Furthermore, a balanced class distribution was preserved in all subsets by employing stratified sampling technique during the splitting. I printed more statistical information about the training set as shown in table(2).

Table 2: The label distribution in training set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Size	2939	2948	2948	2948	2947	2948	2948	2948	2947	2947	2948	2948	2948	2948	2948	2948	2948	2948	2947
(%)	5.25	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26	5.26

2.1.4 Retention of original label pairs

Considering the subsequent implementation of the t-SNE visualisation, I retained the original label pairs for each combined image during both the building and splitting process of the dataset. Specifically, information about different groups under the same new label, such as '2+8' and '1+7', was preserved. I would like to visualise the subcategories under the same label while reducing the dimensionality of all the data, which can provide a clearer understanding of the interpretability in machine learning.

2.2 Neural Network for Classification

2.2.1 Problem Statement

> Using these datasets, develop a neural network pipeline using fully connected neural networks which allows you to address this inference/classification process. The output of your neural network can correspond to a single number, e.g. $7 + 3 = 10$. You should perform some hyper-parameter tuning to establish a good architecture (a choice of five hyper-parameters is what we look for and you should restrict the number of experiments that you can run them on a personal laptop using no more than a few hours). Your report should include a short overview of the results you have obtained in this hyper-parameter search. You should provide the weights of your best performing neural network and the neural networks utilised in plots (i.e. the plots should be reproducible).

2.2.2 Architecture Design

The optimal neural network architecture was determined by the tuning of five hyper-parameters as shown in table(3), and the result of the tuning is shown in table(4). The maximum number of trials was set to 20, and the parameters and strategy were the same as the final model training (see the following section). Meanwhile, the history of the optimising process was recorded and preserved, as plotted in figure(2). The best validation accuracy occurring during the trials was around 96.14%.

Table 3: Hyper-parameter tuning range

Parameter	Range
Hidden Layers	[2, 3]
Neurons per Layer	[128, 256]
Dropout Rate	[0.2, 0.3]
Learning Rate	[0.001, 0.01]
Activation	relu, sigmoid

Table 4: Optimal Hyper-parameters

Parameter	tuning result
Hidden Layers	3
Neurons per Layer	256
Dropout Rate	0.21550
Learning Rate	0.00157
Activation	relu

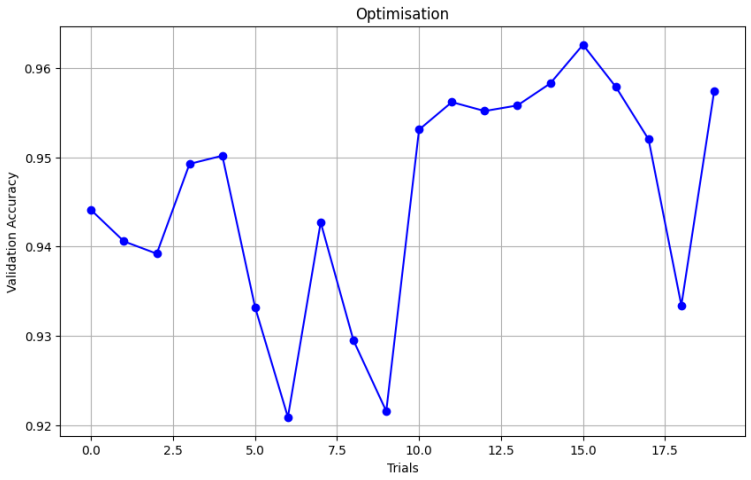


Figure 2: The hyper-parameter tuning process

After the searching process, I created the model with hyper-parameters suggested by `Optuna` trial. The neural network begins with a `Flatten` layer that reshapes the input data from its original dimensions of 28×56 into a one-dimensional array, which ensures the compatibility with the subsequent fully connected layers.

Subsequently, three hidden layers were added, each consisting of a fully connected layer, a `BatchNormalization` layer, a activation layer and a `Dropout` layer. The `Dense` layer contains 256 neurons to learn complex representations of input data through weighted linear combinations of features. It is followed by a `BatchNormalization` layer for normalising activation to improve training stability and accelerate convergence, which is in front of the activation function `relu`. To prevent overfitting, a `Dropout` layer was applied after activation function, with the optimal dropout rate presented in table(4).

Finally, the model concludes with a `Dense` output layer of 19 neurons with a `softmax` activation function. This design is tailored for multi-class classification, as the softmax activation converts the raw output scores of the neural network into a probability distribution[1], which ensures the output represents probabilities of 19 different classes.

2.2.3 Training Process

The training process was configured to optimise model performance while speeding up training efficiency. The model was trained using the Adam optimiser with a learning rate determined during the hyper-parameter tuning. The loss function, `sparse_categorical_crossentropy`, was chosen to handle multi-class classification tasks, and accuracy was used as the assessment indicator.

For the purpose of improving training efficiency, an `EarlyStopping` callback was implemented to monitor validation accuracy. I set `patience=5`, which means the training will be interrupted if validation accuracy dose not improve for 5 epochs, and the best weights at that time will be restored automatically by setting `restore_best_weights = True`.

The maximum of epoch was set to 30 with a batch size of 32. The validation set was used to evaluate the model performance after each epoch, providing a consistent measure of its generalisation performance. After training, the model performance was evaluated on the test set, with the test loss and test accuracy recorded. The best performing model weights were restored automatically from the checkpoint.

2.2.4 Results

After the training, the final model achieved an accuracy of about 96.14% on both the validation set and the test set. However, a strange phenomenon appears in the accuracy on the training set. As we can see in figure(3), the model consistently demonstrates higher accuracy on validation set compared to the training set, while the validation loss is always lower than the training loss. Refer to the section "Why is the training loss much higher than the testing loss?" in [2], this can be explained by the inherent differences in the operation modes between training and testing in `Keras` models. During training process, the `Dropout` layer as a regularisation mechanisms are active, contributing to an increase in training loss. However, these mechanisms are disabled during validation and testing, leading to lower losses on the validation and test sets.

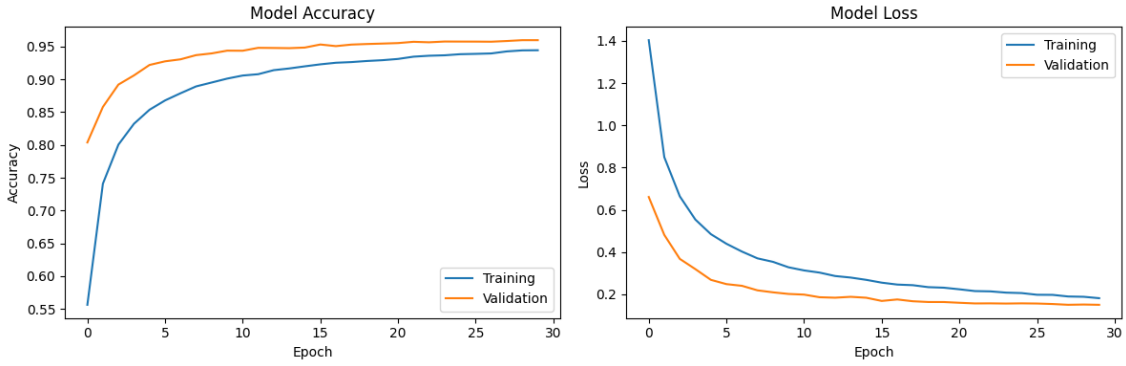


Figure 3: Training history of final model

2.3 Classifiers from Random Forest and SVM

2.3.1 Problem Statement

> Now, showcase the performance using other inference algorithms implemented in `scikit-learn` covered in the lectures such as random forest classifiers and support vector machines and compare the performance. Note that you are not required to describe the algorithms in detail.

2.3.2 Results

In the same context, a random forest classifier and a support vector classifier were applied for the same task, in order to compare them with the fully connected neural network model.

Results showed in table (5) and figure(4):

Table 5: Model Performance Comparison

Model	Training Acc	Validation Acc	Test Acc	Training time
Neural Network	94.45%	96.26%	95.98%	169m 58s
Random Forest	99.89%	76.80%	76.60%	16.5s
SVC	85.64%	51.75%	51.70%n	451.1s

The neural network achieved the highest test accuracy at 95.98%, but also consumed the most considerable computational cost, with a training time of approximately 170 minutes. Conversely, the random forest took the shortest training time (16.5s) and showed a near perfect accuracy on the training set (99.89%). However, its significant decrease in test and validation accuracy indicated that it had insufficient generalisation ability and resulted in overfitting. The SVM, on the other hand, displayed the weakest performance, with about 51.70% test and validation accuracy despite a training time of 451.1 seconds. This suggests that the SVM is not a good choice to handle this kind of classification task.

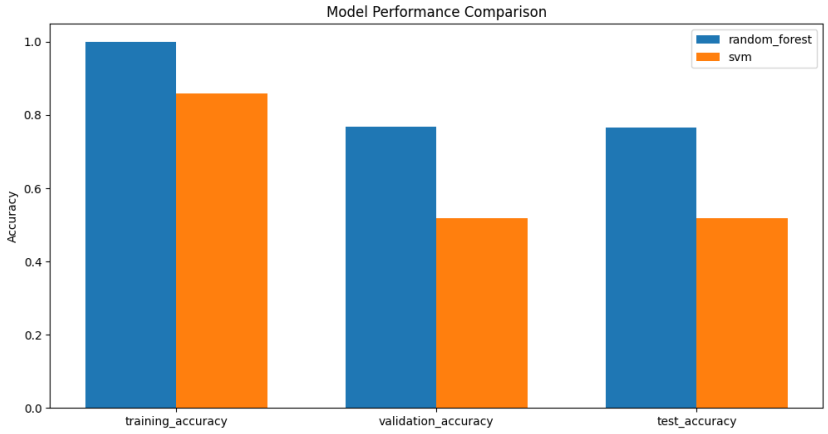


Figure 4: Comparison with random forest and SVM

2.4 Linear Classifiers

2.4.1 Problem Statement

> Using a weak linear classifier, compare the classifier probabilities for a linear classifier trained on the 56×28 dataset with the probabilities of a single linear classifier which is applied on the two images sequentially. How do the performances on the test set compare when trained with few samples (i.e. train with varying number of samples including 50, 100, 500, and 1000 samples respectively).

2.4.2 Methodology

Exploiting the `LogisticRegression` method in `scikit-learn`, two types of linear classifier were developed under different training sample sizes: A linear classifier directly trained on combined images (56×28) by loading the `combined_mnist`, while a single linear classifier applied on two images sequentially (two 28×28) by loading the `MNIST` for training set and separating the images on `combined_mnist` for test set. Subsequently, different sample sizes (50, 100, 500, and 1000) were tested in the two classifiers respectively.

For both classifiers, the 'multinomial' type with 'lbfgs' solver in `LogisticRegression` were chosen for multi-class task and the maximum iterations was set to 1000. In addition, the inverse of the L2 regularisation strength C was set to the default value of 1.

2.4.3 Results

After a quick training (total time around 12 seconds), the accuracy on test set of these two linear classifiers are shown in the table(6) and figure(5).

Table 6: Test accuracy of linear classifiers with different sample sizes

Classifier \ Size	50	100	500	1000
Combined	14.62%	14.61%	21.66%	23.72%
Separate	37.72%	58.73%	72.51%	76.46%

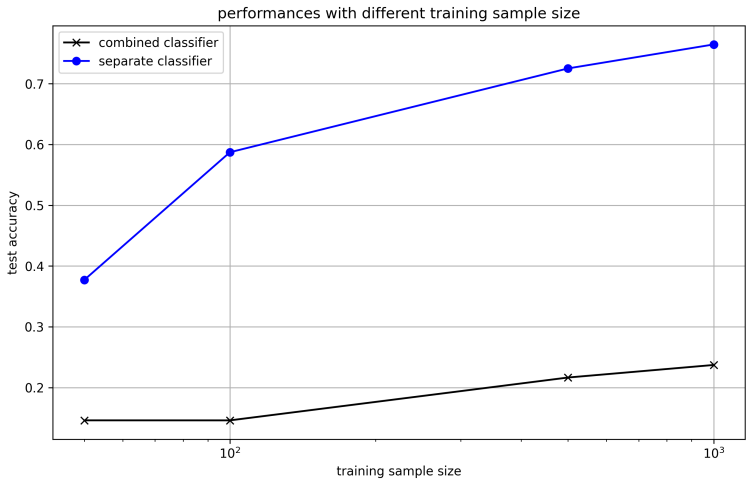


Figure 5: Test accuracy of linear classifiers with different sample sizes

We can see that the separate classifier consistently performed better than the combined classifier across all sample sizes, with accuracy 2-4 times higher than the combined classifier. On the other hand, both classifiers

showed improved performance with increased training samples, and the separate classifier demonstrated higher improvement from 50 to 1000 samples, which showed potential for further improvement with more samples.

These results indicated that, for linear method, treating this task as two separate digit recognition problems is better than learning the sum directly from combined images, which may be attributed to the clearer structure of each features and the less types of label in separated classifier.

2.5 t-SNE Analysis

2.5.1 Problem Statement

> For your best performing neural network, show the t-SNE distribution of the various classes in the embedding layer (i.e. the layer before the output). Compare this representation with the representation obtained by directly applying t-SNE on the input dataset. You should optimise the perplexity.

2.5.2 Methodology

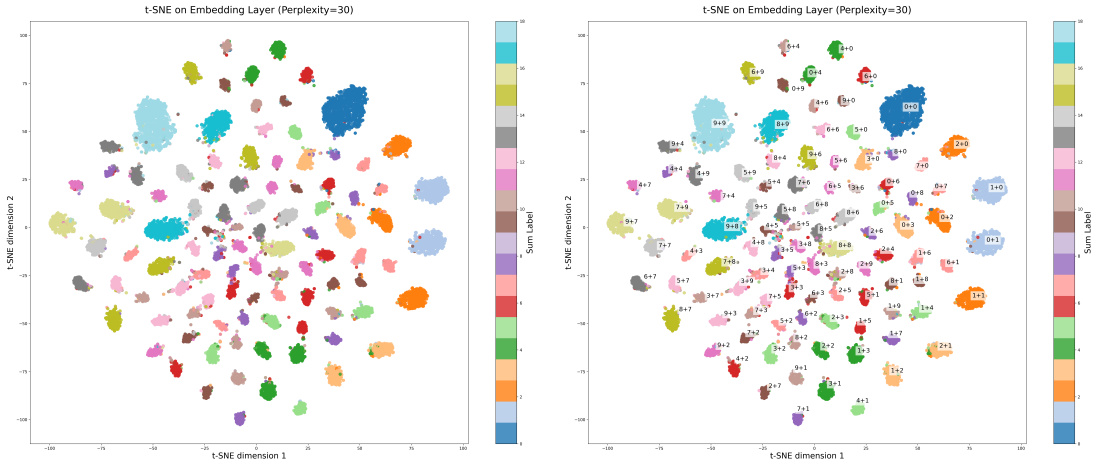
Taking use of `TSNE` class in `sklearn.manifold` module, a python module named `tsne` was built for t-SNE analysis and visualisation. I firstly set the perplexity to 30, and performed t-SNE for both input and embedding layers of our final NN model.

Consequently, the 2-dimensional plots can be drawn to show the high-dimensional model structure. Additionally, the original label pairs I retained in the dataset allowed me to add the detailed annotates in the figure(6) to further observe the specific distribution of the 18 label categories.

Then, in order to optimise the perplexity parameter in t-SNE method, I iterated over four perplexities [5, 10, 30, 50, 100] and made a visualisation of their results as shown in Figure(7).

2.5.3 Results

In figure (6), the well-separated clusters shows that the t-SNE algorithm had successfully captured some underlying patterns in how the neural network had learned to infer digit combinations.



(a) t-SNE in the embedding layer

(b) t-SNE in the embedding layer with original label pairs

Figure 6: t-SNE distribution in the embedding layer with perplexity = 30

Firstly, the result exhibits clear clustering behaviour, as marked by different colours. Yet, it is worth noting that most original digit pairs with same sum label formed multiple sub-clusters rather than single one. For instance, sum label of 0 only formed one cluster while larger sums displayed more clusters. Further more, the number of

sub-clusters in sum label N (0 to 18) can be described by the kind of combinations:

$$kinds = \begin{cases} N+1 & \text{if } 0 \leq N \leq 9 \\ 19-N & \text{if } 10 \leq N \leq 18. \end{cases} \tag{2.1}$$

, which is mathematically correct.

Secondly, it can be found that two pairs containing one or two same digits tended to be close together (e.g., 1+0, 0+1, 1+6,...), suggesting the network had captured underlying visual patterns in identical digits. Moreover, two pairs including visually similar digits also close to each other (e.g., 9+3, 7+2).

Finally, some of visually distinct digit pairs whose sum were same or close might be also close in position (such as 2+2, 1+3, same sum 4), and it can be seen that the smaller sums were more likely to locate in the right part of the figure. These suggest that the network had learned some numerical relationships beyond pure visual features. However, this pattern is not uniform across the whole area, with some regions showed more complex organisational principles.

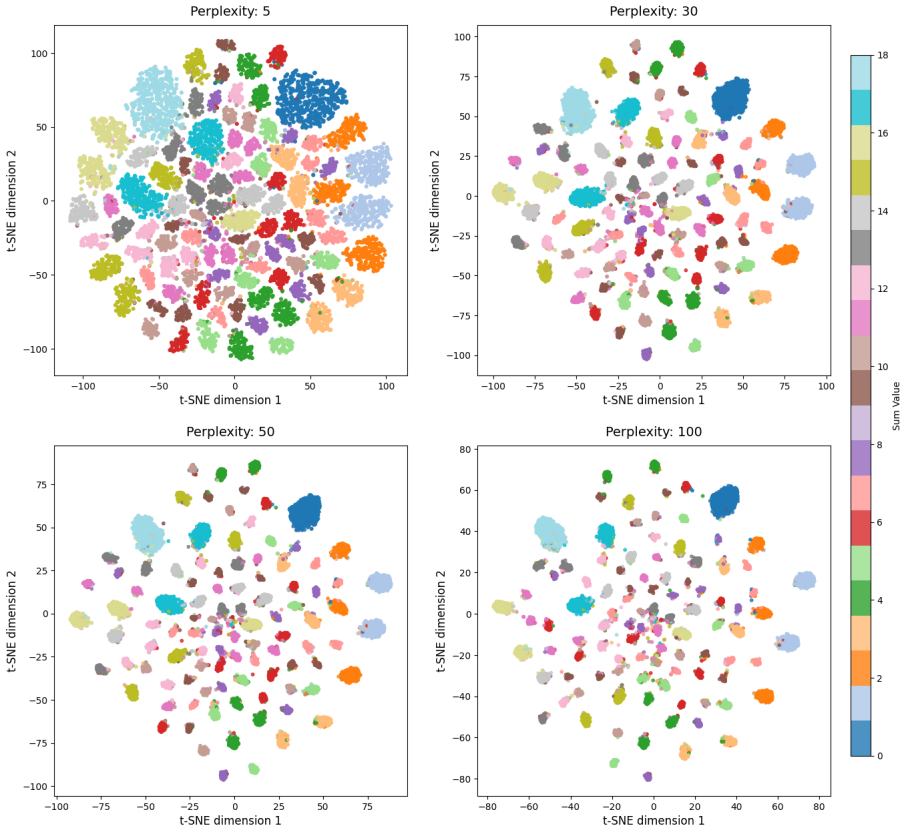


Figure 7: Perplexities tuning in t-SNE

It can be seen that the best performance in this range might be the perplexity=30, as the intra-class density was low under the lower perplexity while the higher perplexities resulted in a longer distance between some separated sub-clusters in the same label, such as the '4+0' and '0+4' clusters (in green).

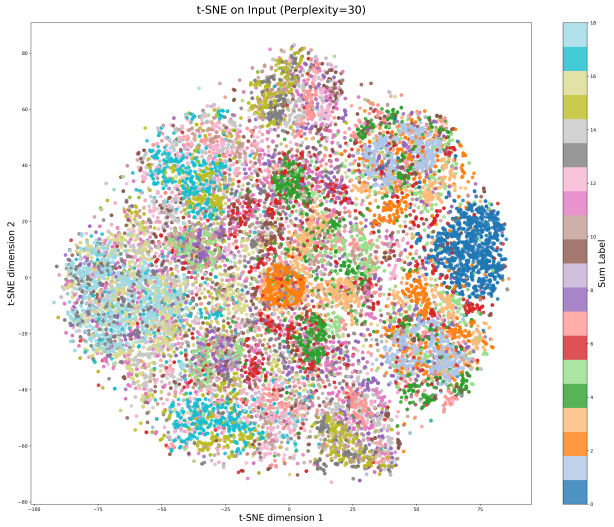


Figure 8: t-SNE distribution on the input dataset

Comparing with the results of embedding layer, the model projection on the raw input dataset showed a worse cluster separation effect in figure(8), which is quite justified because at this point the model had not yet started to classify the data, so the homogeneous original dataset still showed a great deal of clutter after the dimensionality reduction. This comparison successfully presented the classifying process by the NN model.

3) Conclusion

In summary, an inference pipeline was successfully developed for a classification task on a new self-built dataset `combined_mnist`, using multiple approaches including neural networks, random forests, support vector machines, and linear classifiers. Additionally, the dataset was visualised through both the input and embedding layers of the NN model exploiting the t-SNE technique, presenting the internal working modes of the NNs.

References

[1] Wikipedia contributors, "Softmax function — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Softmax_function&oldid=1261563147, 2024. [Online; accessed 10-December-2024].

[2] F. Chollet *et. al.*, "Keras faq." https://keras.io/getting_started/faq/, 2015. Accessed: 2024-12-10.

Appendix

Declaration: No auto-generation tools were used in this report except for generation of BibTeX references.