M1 Machine Learning Coursework Report

Hania Rezk (hhmmer2)

MPhil Data Intensive Science, Department of Physics

5 December 2024

Word count: [3041]

Contents

1 Introduction

MNIST is a famous database of handwritten digits that has become a benchmark for evaluating different machine learning algorithms because of its consistency and high quality.

While the original MNIST classification problem aims at classifying a handwritten image into one of 10 classes representing integers from 0 to 9, our objective will be to create an inference pipeline that can successfully classify the result of the addition of two handwritten MNIST digits.

This report outlines the steps taken to address the problem, along with the results obtained.

Note: Please refer to the README file for notes about running the notebooks and more details about the components of the project folder.

2 Generating the New Dataset

2.1 Exploring and Scaling the Data

First, we load the MNIST dataset using the `TensorFlow` library and visualise its components. Each image, in `x_train` and `x_test`, is an array of shape 28×28, corresponding to its pixels, while `y_train` and `y_test` contain their corresponding labels.

The next step is to scale our datasets by dividing each pixel value by 255.0 (the maximum value of the pixels) to normalise them to the range of [0,1]. This preprocessing step is important as it ensures that our models will be able to learn without being influenced by large pixel values.

In `data.ipynb`, I included the code I used to understand the data. I explored its shape, visualised its first 20 images, and plotted the distribution of labels in `y_train` and `y_test`. The plots (Figures 1 and 2) below reveal that the labels for both datasets have the same distribution: they are uniformly distributed.

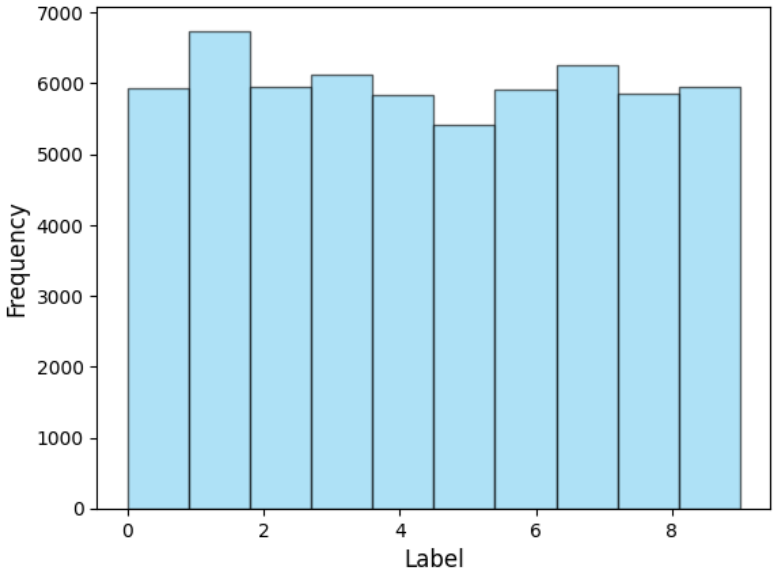

Figure 1: Label Distribution in the MNIST Training Dataset

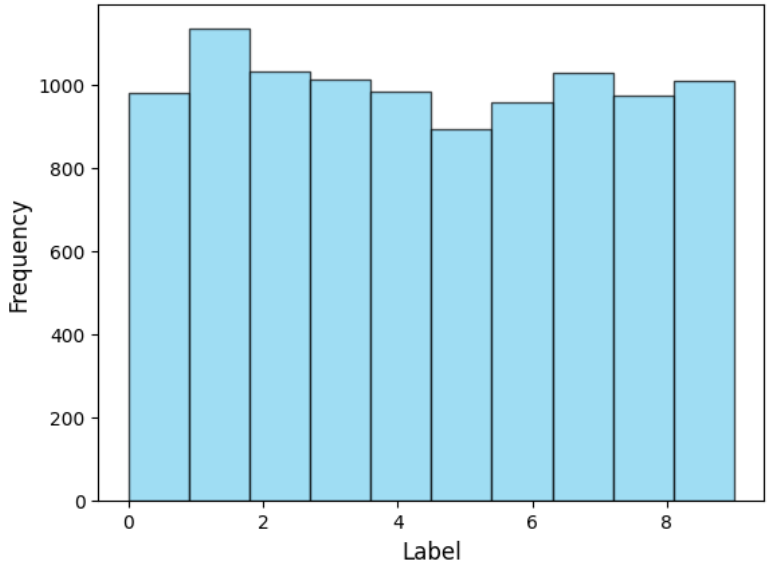

Figure 2: Label Distribution in the MNIST Testing Dataset

2.2 The Generation Process

To generate the combined data, my code selects with replacement, two random images from `x_train` and `x_test` (`image1` and `image2` in the code) and

adds their combination to `combined_x_train` and `combined_x_test`. The combined images are created using the `np.concatenate()` method with `axis=0`, which stacks the images vertically, resulting in a combined image of size 56x28. Simultaneously, the code also adds the sum of the labels of these images to `combined_y_train` and `combined_y_test`.

This process was repeated, `len(x_train)` times, until the size of the new datasets (`combined_x_train` and `combined_y_train`) matched those of the original datasets.

The validation set was created by splitting 20% of the new training set, using the `train_test_split` method. A 20% split is widely used in machine learning, shown to provide the best results in empirical studies (references).

The histograms (Figure 3, 4 and 5) of the labels of the three new combined datasets reveal a distribution resembling a normal distribution.

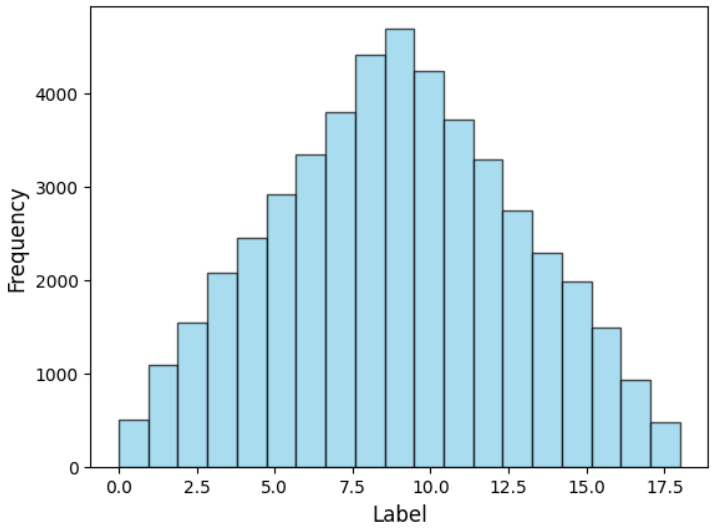

Figure 3: Label Distribution in the New Training Dataset

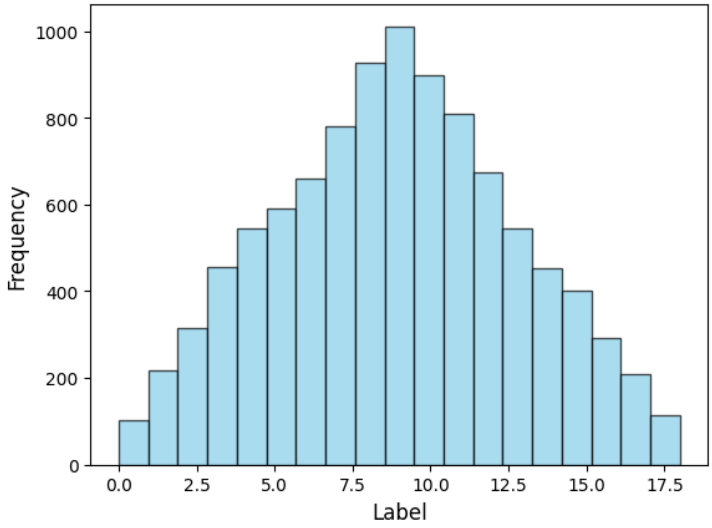

Figure 4: Label Distribution in the New Testing Dataset

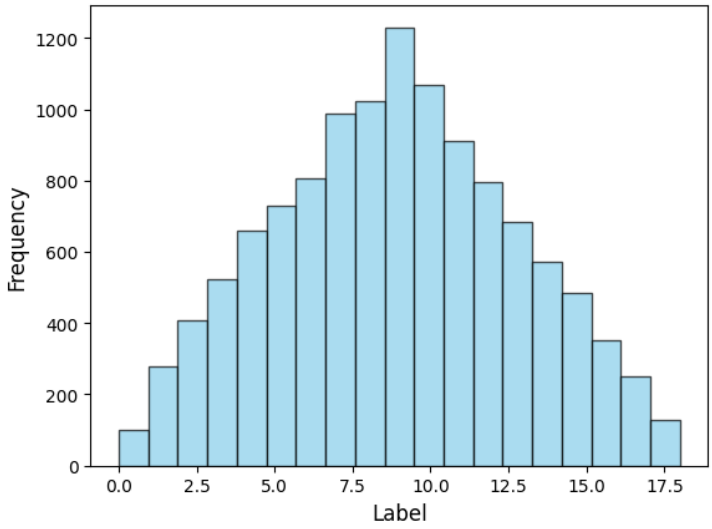

Figure 5: Label Distribution in the New Validation Dataset

The imbalance in the number of images across the classes arises because some numbers (e.g., 9) can be obtained from multiple digit pairs (3+6, 4+5, 2+7,..), while others (e.g., 0 and 18) can only be formed by a single digit pair (0+0 and 9+9).

Therefore, the imbalance in the dataset is not an issue of data collection or processing, but rather a natural outcome of how the numbers are constructed

from their constituent digits. Despite this imbalance, the new combined datasets mirrors the behavior of numbers in real life and we can observe from above that the three new datasets (train , test and validation) have similar class distributions. This ensures that the datasets maintain all the necessary statistical properties.

Note: Sampling with Replacement: I chose to sample with replacement to generate the new combined datasets. Sampling without replacement would have significantly reduced the size of our datasets to half of the original size, as we would have quickly exhausted unique combinations of images.

We can visualize the first 20 images of our new combined datasets (Figure 6), we always get the same images because our results are reproducible.

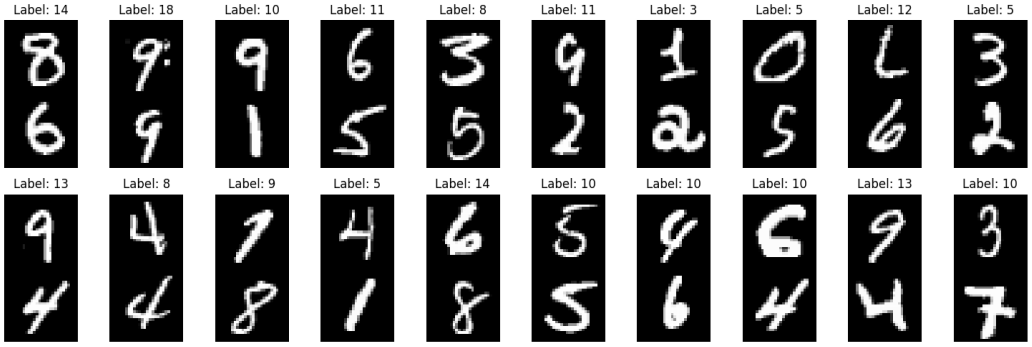

Figure 6: First 20 Combined Images Of The New Training Dataset

Note: The lists with the word sequential in their names (e.g, `sequential_y1_train`....) of the `models.ipynb` notebook will be explained in section6.

3 Neural Networks

Neural networks are the most powerful machine learning models for image classification. In this section, we will test their performance on our problem.

For my model architectures, I selected the Adam optimizer due to its adaptive learning rate and fast convergence. I also chose categorical cross-entropy for the loss function, as it's the most appropriate for multi-class classification tasks, especially after the conversion of my labels to one-hot encoded vectors.

The batch size for the models is fixed at 128, a value that was chosen to generate stable gradients and therefore ensure faster convergence. Finally, I started with a number of epochs set to 20 for my trials, which I will tune after optimizing other hyperparameters.

3.1 Base Model

I started my inference pipeline with the implementation of a neural network with a basic architecture consisting of: 1 layer, 50 hidden units, `relu` activation, and an Adam learning rate of 0.001 (chosen arbitrarily). This model, its weights and training are saved in the files `base_model.h5`, base_training_history.json and base_model.weights.h5 respectively, in the `NN_files` folder of the trials folder.

The test accuracy of that model is 0.7009 As expected, the performance improves gradually across epochs (Figure 7).

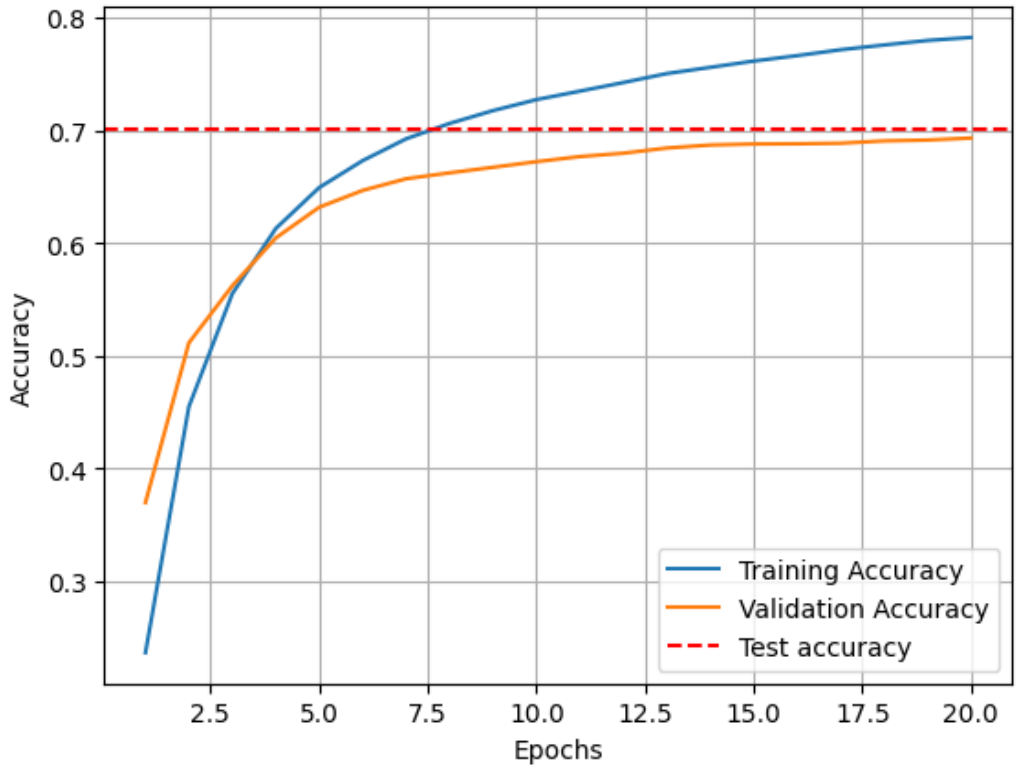

Figure 7: Training, Validation and Testing Accuracies Of The Base Neural Network Across Epochs

However, we can still dramatically improve this performance by hyperparameter tuning. I decided to focus on five key hyperparameters to fine-tune for the first part of my studies. These parameters are: the number of layers, the dropout rate, the number of units, the learning rate for the Adam optimizer, and the activation function. To optimize these parameters, I used the `optuna` module, an automatic hyperparameter optimization framework. You can load the following `trials` and visualize their results from `trials.ipynb`.

Note: Refer to the README before running the `trials.ipynb` notebook.

3.2 Studies

This section is divided into four sequential studies, as presented in figure 8, each focusing on a critical aspect of model performance.

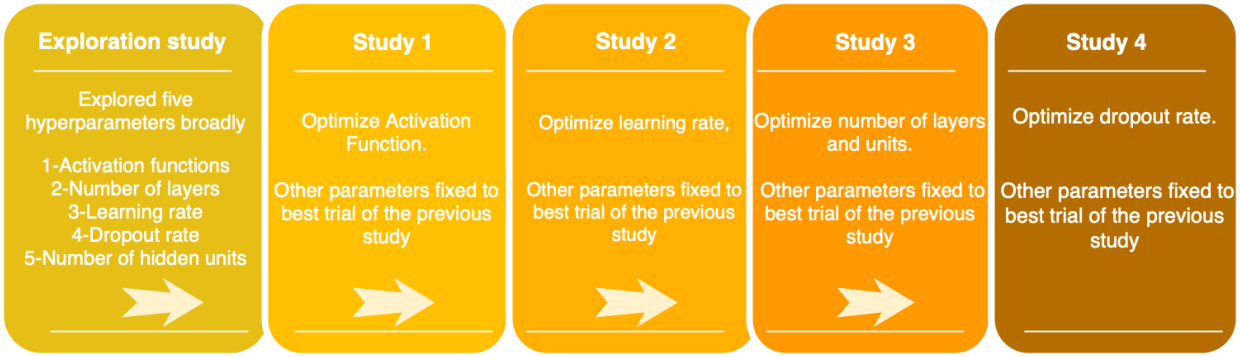

Figure 8: Neural Network Hyperparameter Optimization Pipeline

My first study, stored in `study1.db`, is an exploration study where I set a large range for the hyper parameters to monitor their general performance. The best model from this study achieved a test accuracy of 0.9081, representing a significant improvement over the base model's test accuracy.

In the `Activation_tuning` function, I chose to focus on fine-tuning the activation functions. I fixed the other hyperparameters based on the best model of the exploration study and varied the activation functions. I didn't use `optuna` for this study; instead, I just defined a function that loops over the best model found in the exploration study by only modifying the activation. From the results, we can clearly observe that the `relu` activation, with a test

accuracy of 0.9079, performs better than the `sigmoid` and `tanh` activations, with test accuracies of 0.8413 and 0.8688 respectively.

The goal of `study2`, `study_2.db`, is to find the best learning rate for the network. Here, I fixed the hyperparameters of the best model from the exploration study, the best activation found in `study1` (`relu`), and focused on optimizing the learning rate. The best model had a test accuracy of 0.9132 for a learning rate of 0.00099.

In `study3`, `study_3.db`, I aimed to fine-tune the optimal number of layers and units in the network. I set the value of the best learning rate to the one found in `study2` and I varied the number of layers and units per layer to determine how the architecture impacts the model's performance. The best model had a test accuracy of 0.9210 for 5 layers with the following number of units: [253,219,112,198,215].

Finally, in the fourth study, I fixed all hyperparameters based on the best values found in the previous studies, except for the dropout rate. This study aimed to identify this optimal rate to prevent overfitting and improve generalization. The best model achieved an accuracy of 0.9219 with a dropout rate of 0.0856.

At the end of `study4`, we have already identified a strong architecture that provides us with good accuracies.

3.3 Further Tuning

We can still improve the performance of this best-performing model by using early stopping and batch normalization, which are techniques we have learned in our M1 course. We use batch normalization to improve the stability of our training and reduce our model's sensitivity to weight initialization, while early stopping helps to enhance the model's generalization performance.

Our model stops training after 7 consecutive epochs with no improvement in validation accuracy. I have decided to set an upper limit on the number of epochs to 50. In the case of our best model, early stopping occurred after epoch 46, and we achieved an accuracy of 0.9300, better than the one obtained before.

This best model, its weights and training history are saved in the `best_model.h5`, best_model.weights.h5 and `best_training_history.json` respectively, which can be found in the `NN_files` folder of the `trials` folder.

We have determined a good neural network architecture (Figure 9).

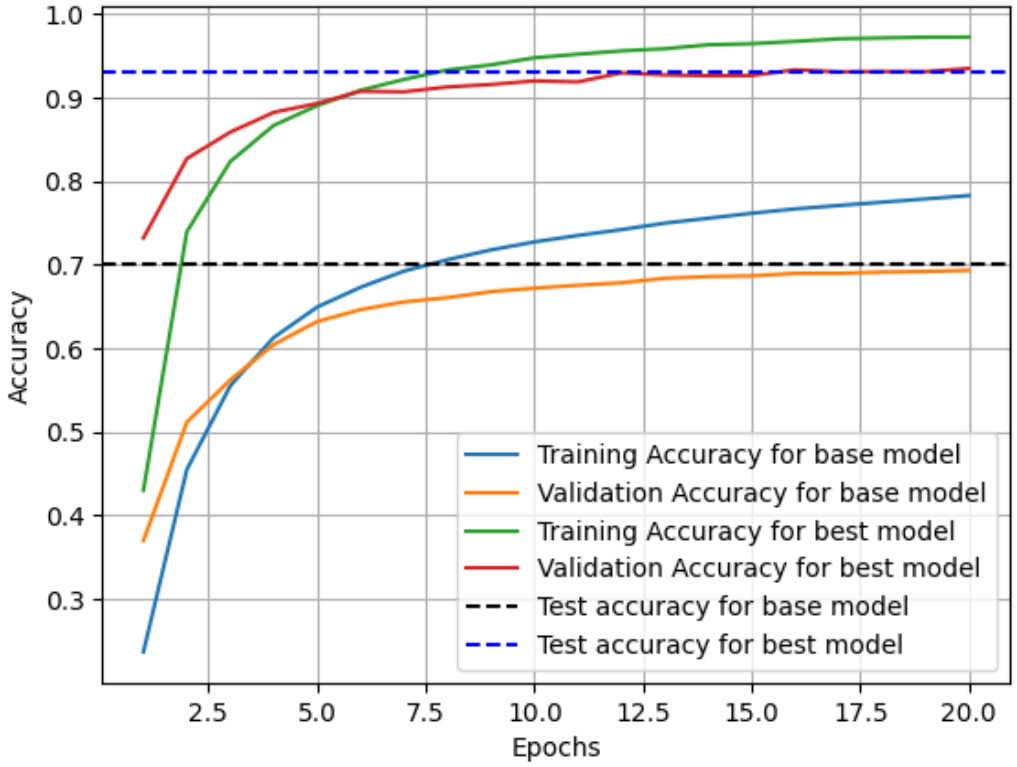

Figure 9: Training, Validation and Testing Accuracy Comparison Between Base and Best Model Across Epochs

4 Question 5: t-SNE

To evaluate how effectively the best model has learned to represent the data, we visualize its representation in the embedding layer using t-SNE, which projects the high-dimensional data into a lower-dimensional space.

The t-SNE plot (Fig. 10) applied to the embedding layer reveals the formation of distinct clusters. This can be explained by the fact that, in the penultimate layer, the best performing neural network has already learned to represent the data effectively.

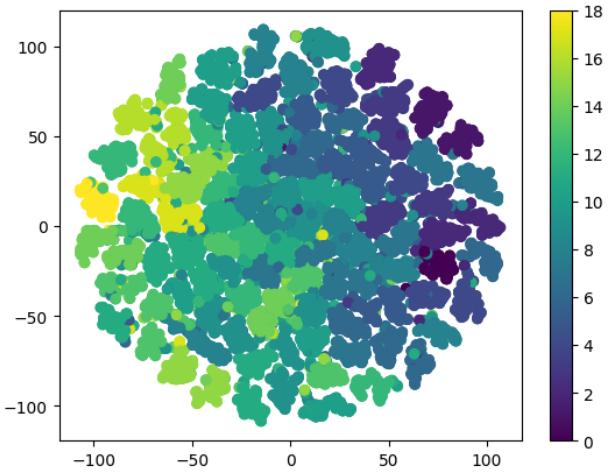

Figure 10: t-SNE Of The Embedding Layer

This observation becomes even more evident when compared to the t-SNE plot applied to the input layer. In the input data plot (Fig. 11), the data points are scattered randomly without being assigned to a cluster, which reflects the lack of learned structures in the input data.

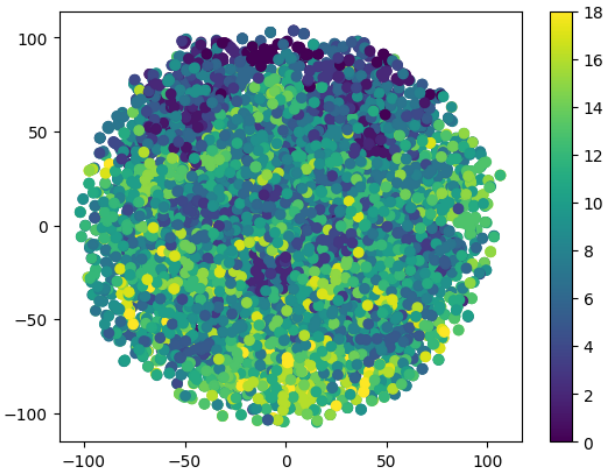

Figure 11: t-SNE Of The Input Layer

When we optimise the perplexities for t-SNE (Fig. 12), we observe that the clusters get more compact and separated. Perplexities around the value 20 seem to provide the clearest and most optimal cluster representations. As the perplexity increases beyond 20, the model starts to detect more subtle relationships within the combined data, and exaggerates the differences

between that data in the same class. One example of this intra-class variability can be the different combinations for each label, e.g., 3+5 is assigned to a cluster that is separate from the cluster of 4+4, even though they both produce the same label 8.

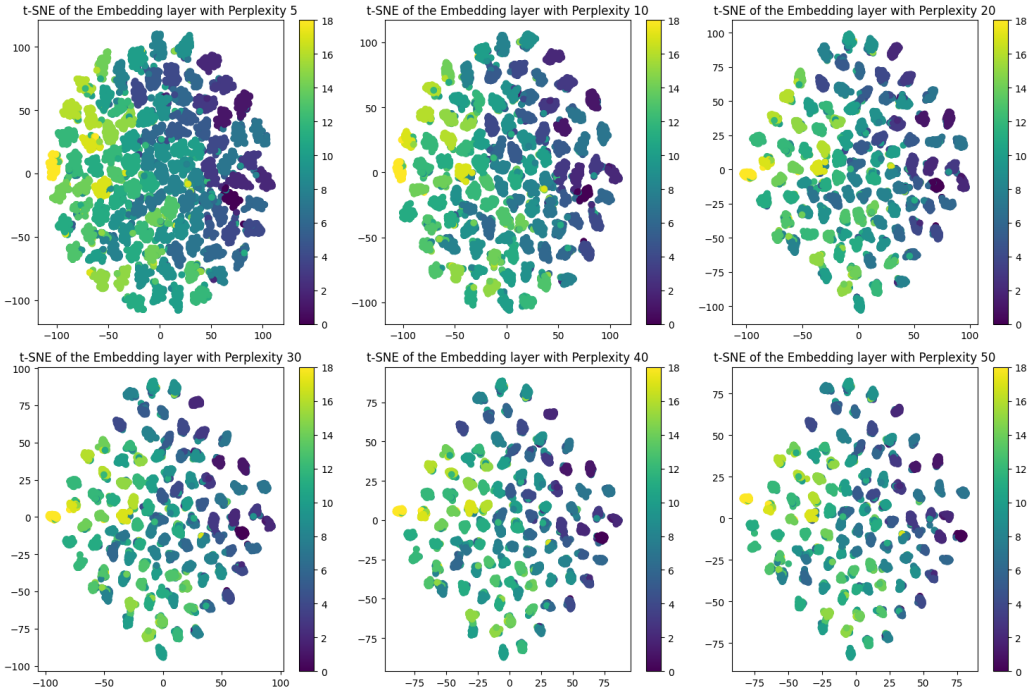

Figure 12: t-SNE of the Embedding Layer for Different Perplexities

This section shows how, for a reasonable value of the perplexity, our best model was able to detect and differentiate between the different classes.

5 Algorithms Implemented in Scikit-learn

To put the performance of our neural networks (NNs) in perspective, let's see how more traditional machine learning algorithms perform for this problem.

5.1 Logistic Regression

We start by applying a simple logistic regression model with default parameters. The model does not converge and outputs the following accuracies: test accuracy of 0.2204, and a training accuracy of 0.3010.

To determine whether this performance is due to the chosen parameters, we perform hyperparameter tuning using two different solvers, `sag` and `lbfgs`, along with different regularization values. The following results were obtained (see Figure 13).

C Value	Solver	Test Accuracy	Validation Accuracy
0.1	lbfgs	0.1636	0.163833
0.1	sag	0.2247	0.228
1	lbfgs	0.1636	0.163583
1	sag	0.2201	0.219
10	lbfgs	0.1636	0.163583
10	sag	0.2174	0.217083
100	lbfgs	0.1636	0.163583
100	sag	0.2169	0.216917
1000	lbfgs	0.1636	0.163583
1000	sag	0.2165	0.21675
1e+15	lbfgs	0.1636	0.163583
1e+15	sag	0.2164	0.21725

Figure 13: Hyperparameter Tuning Results for Logistic Regression

The best model is the one with the highest validation accuracy: the model using the `sag` solver with an l_2 penalty value of 0.1, which has a test accuracy of 0.228 (slightly better than the base model). This makes sense, as the `sag` solver is more suitable for large datasets due to its stochastic nature and fast training.

We then also compare the performance of that best logistic regression model for different `max_iter` values, with the results below (see Figure 14).

Iterations	Test Accuracy	Validation Accuracy
100	0.2253	0.228417
500	0.2237	0.22725
1000	0.2249	0.22775
1500	0.2242	0.22825

Figure 14: Hyperparameter Tuning Results for Logistic Regression with `sag` solver and l_2 Penalty

We can see that test and validation accuracies are almost identical for all the iterations.

5.2 Support Vector Machine (SVM)

Let's apply SVM to this problem and evaluate its suitability. We start by training the model with default parameters. This model achieves a test accuracy of 0.3507 and a training accuracy of 0.3807.

Let's try to improve these accuracies through hyperparameter tuning. This model took the longest to train, around 40 minutes for the full dataset (on google colab), which is why hyperparameter tuning was only done on 25% of the dataset (see results in Figure 15). We notice from these results that we cannot determine a best model based on the validations accuracy: either the model overfits with more than a 20% difference between the training and testing accuracy or the model has very low testing accuracies that do not compare to the ones found for our NNs.

Kernel	Gamma	C	Test Accuracy	Validation Accuracy	Training Accuracy
poly	scale	0.1	0.2963	0.29475	0.38825
poly	scale	1	0.6364	0.632417	0.95925
poly	scale	10	0.6574	0.657917	0.999917
poly	scale	100	0.658	0.658333	1
poly	auto	0.1	0.1011	0.1025	0.0970833
poly	auto	1	0.1011	0.1025	0.0970833
poly	auto	10	0.1011	0.1025	0.0970833
poly	auto	100	0.2719	0.266167	0.341583
rbf	scale	0.1	0.1482	0.147	0.158417
rbf	scale	1	0.5911	0.587	0.864333
rbf	scale	10	0.6693	0.666333	1
rbf	scale	100	0.6694	0.666083	1
rbf	auto	0.1	0.1011	0.1025	0.0970833
rbf	auto	1	0.2174	0.219333	0.256417
rbf	auto	10	0.3864	0.384167	0.573
rbf	auto	100	0.5194	0.513333	0.964583

Figure 15: Hyperparameter Tuning Results for Support Vector Machine

5.3 Random Forest

The next model we will use is Random Forest. The base model with default parameters has a test accuracy of 0.7312 and a training accuracy of 1.0, which indicates overfitting.

Hyperparameter tuning is essential in this case. I started by visualizing the test, validation, and training accuracies for the Random Forest model while varying the `max_depth` from 1 to 25. Additionally, I created a list, `overfitting_depths`, to track the depths at which the difference between training and testing accuracies exceeds 10%, signaling overfitting. We obtain the following results (see Figure 16).

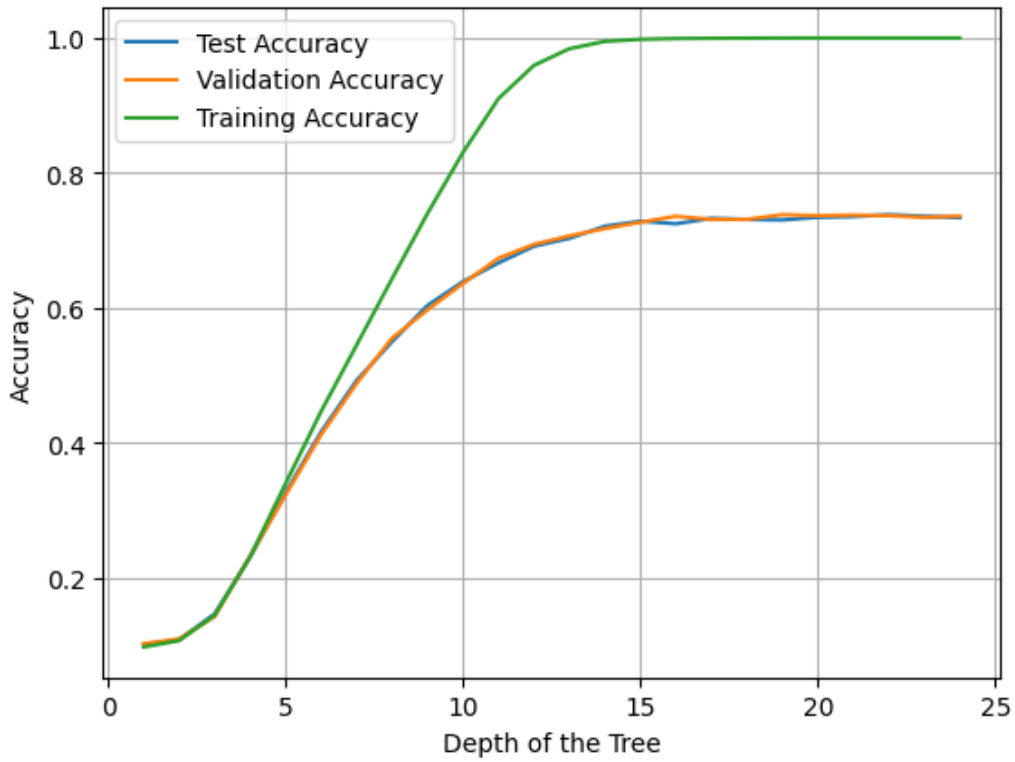

Figure 16: Effect of Tree Depth on Random Forest Accuracy: Training, Testing, and Validation

From the plots, we can say that overfitting starts to happen around depth 9: the training accuracy increases dramatically, while the validation and testing accuracy start to stagnate around depth 12. For depths between 3 and 15, I also experimented with adjusting other parameters such as `min_samples_split`, to prevent the model from making too many splits, and `min_samples_leaf`, to avoid creating leaves based on a small subset of the data. These adjustments slightly reduced the training error but are globally insignificant (Figure 17).

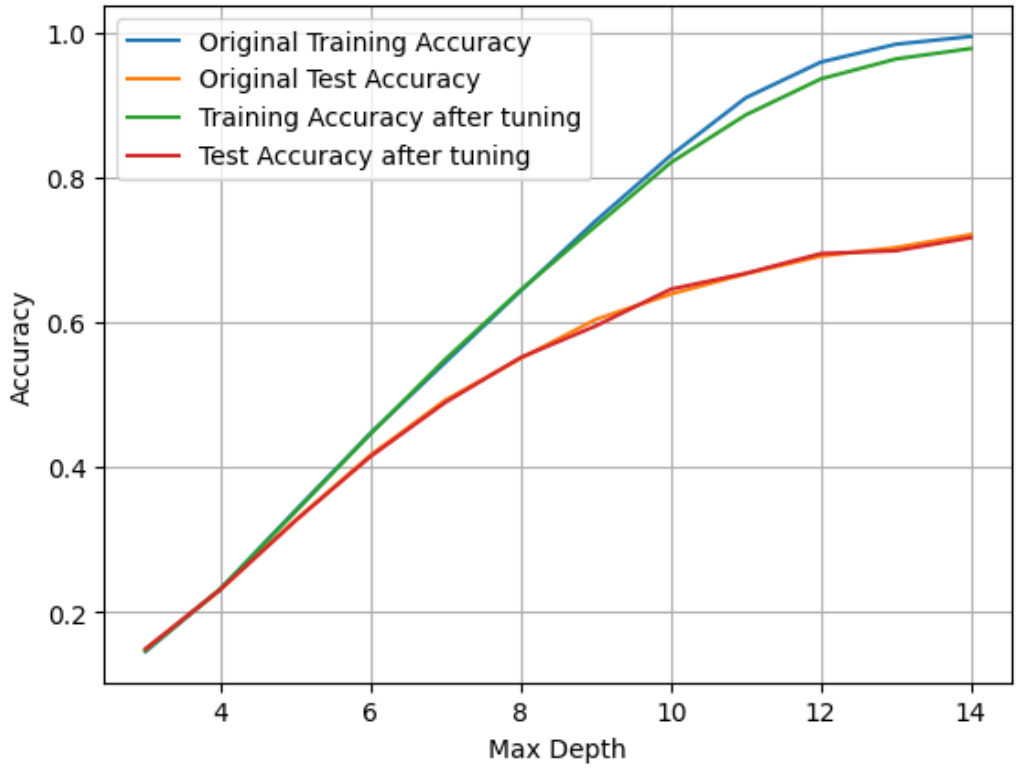

Figure 17: Effect of Tree Depth on Random Forest Accuracy: Training and Testing After Hyperparameter Tuning

Finally, I decided to limit the `max_depth` of the tree to value 8 (first element in the `overfitting_depth` list -1) to prevent overfitting. For this depth, the test accuracy is 0.5572 and the training accuracy is 0.6426.

Based on these results, we can conclude that these classifiers are not suitable for this problem; logistic regression gives very poor test accuracies while SVM and Random Forest struggle to generalize and tend to overfit. Their poor performance, with test accuracy lower than that of a random guesser (less than 50%), did not improve with hyperparameter tuning.

6 Question 4

In this section, we will compare the classifier probabilities for a linear classifier trained on the 56×28 dataset with the probabilities of a single linear classifier applied to the two images sequentially.

6.1 Deconstructing the Images

The first step is to deconstruct the combined images and labels used for training and testing. This is the purpose of the `sequential` lists in the data generation: `sequential_x1_train` corresponds to the first image of the combined training images, while `sequential_x2_train` contains the second image.

These lists were also shuffled and split in the same way as the original combined datasets to ensure that the images and labels match in the sequential lists (Figure 18), and that they are indeed the deconstruction of the corresponding combined image (Figure 20),.

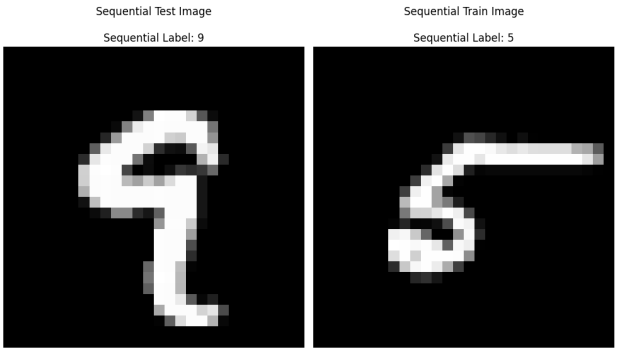

Figure 18: 60th Components of the Sequential Lists

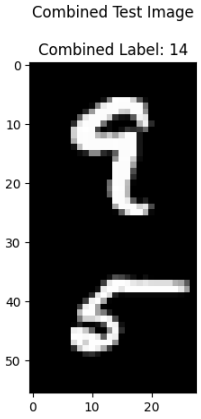

Figure 19: 60th Components of the Combined List

6.2 Models

The single linear classifier I chose for this section is logistic regression (`sag` solver with l_2 penalty of value: 0.1) because SVM requires a significant amount of time to run and because Random Forest tends to overfit.

I initialized two logistic regression (LR) models:

- `log_combined`: trained on the combined dataset.
- `log_reg_image1`: trained on the two individual images of the combined data sequentially.

For the sequential model, the test accuracy was determined by comparing the predicted sum of the classes for each pair of images in the sequential lists (obtained by selecting the class with the highest probability for each image) to the true combined label.

For the different sizes of the training set, we observe the following results (Figure 20).

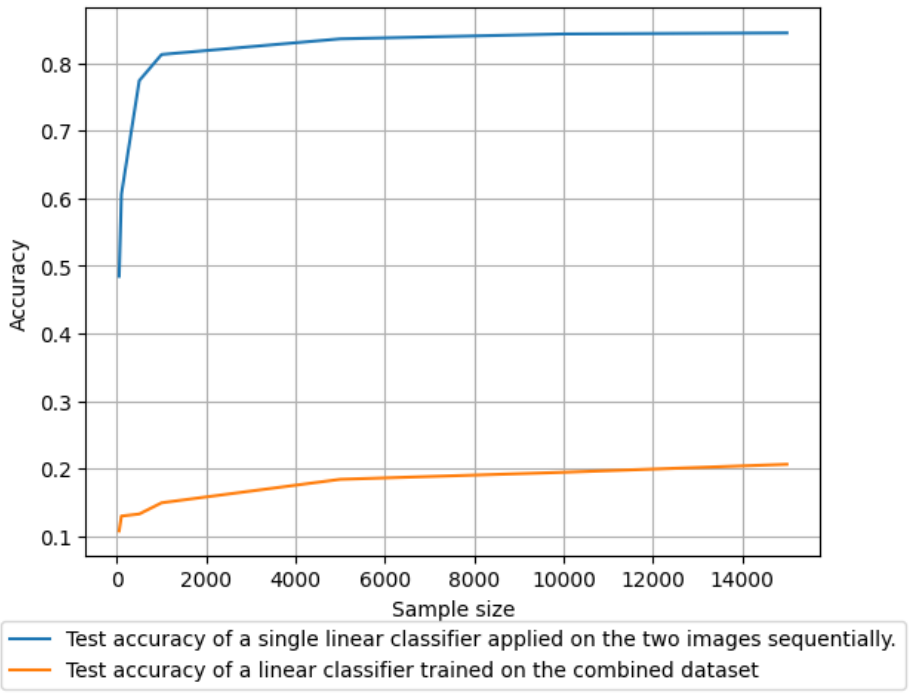

Figure 20: Compairison of Test Accuracy Between Single Linear Classifier And Combined Linear Classifier

We can clearly see that the single LR model trained on the sequential data performs significantly better than the LR model trained on the combined dataset. The test accuracy for the sequential model is more than four times higher than that of the combined model. This can be explained by the fact that LR is highly effective for the MNIST dataset (as demonstrated in one of the lectures), achieving a test accuracy higher than 90% on the original dataset. Therefore, when the predictions are summed, it remains highly accurate. In contrast, the LR model trained on the combined images struggles to achieve a test accuracy above 30% making it inefficient for the 56x28 images.

7 Data Reproducibility

To guarantee that all of our observations are reproducible, we reset the random seed using a fixed value of 42, for every random operation. Before using the random function of the numpy library, we set `np.random.seed(seed_value)` and `random.seed(seed_value)` to generate the same random numbers and ensure that the same random images are selected in every run of our code. We also set a seed in the `train_test_split` function to ensure that the data is split in the same way between validation and training.

Finally, we reset our seeds before running our studies (part 3) so that the same parameters are chosen for each trial.

8 Declaration of Use of AI Tools

ChatGPT provided me with the code to present the results for the hyperparameter search for logistic regression. It allowed me to learn how to use the `tabulate` library.

AI was also used to check for spelling mistakes in this report.

9 References

1. M1 gitlab lecture material
2. `https://medium.com/all-things-ai/in-depth-parameter-tuning-for-random-fores`
3. `https://udlbook.github.io/udlbook/`
4. `https://optuna.readthedocs.io/en/stable/faq.html`

5. https://www.geeksforgeeks.org/how-to-tell-keras-to-stop-training-based-on-l