UNIVERSITY OF CAMBRIDGE

M1 Machine Learning Coursework

xl628

Department of Physics, University of Cambridge

December 18, 2024

Document Statistics:
Words in text: 1736
Words in headers: 82
Words outside text (captions, etc.): 560
Number of headers: 34
Number of floats/tables/figures: 13
Number of math inlines: 5
Number of math displayed: 1

1 Introduction

The MNIST Addition task involves creating a synthetic dataset where pairs of digits from the MNIST dataset are combined to form an image, and their sum serves as the label. This report documents the methodology for generating this dataset, the implementation of a neural network pipeline optimized using Optuna, and the comparison of performance with Random Forest, Support Vector Machine (SVM), and Linear Classifiers. We also analyze the underlying data and model representations using t-SNE visualizations to demonstrate the effectiveness of our optimal model.

2 Dataset Generation

The primary implementation for dataset generation is located in `notebooks/data_generation_analysis.ipynb`.

2.1 Implementation: `MNISTAdditionDataset` class

The dataset is created using the `MNISTAdditionDataset` class, defined in `src/data_handler.py`. Key methods and features include:

Normalization and Concatenation The MNIST dataset is normalized from [0, 255] to [0, 1] upon loading. Two random digits are sampled with replacement and concatenated horizontally via the `_combine_images` method:

```
indices = np.random.choice(len(images), size=(num_samples, 2), replace=True)
combined_images = np.zeros((num_samples, 28, 56))
combined_labels = labels[indices].sum(axis=1)
digit_labels = labels[indices]

for i in range(num_samples):
combined_images[i, :, :28] = images[indices[i, 0]]
combined_images[i, :, 28:] = images[indices[i, 1]]
```

Listing 1: Code snippet for `_combine_images`

Random Seed To ensure reproducibility, a default seed (42) is set during initialization.

Dataset Split The generated dataset is divided into training, validation, and test sets. We first use `_combine_images` twice to generate the training and test datasets. Next, we further split the training dataset into training and validation datasets. During this process, we do not apply shuffling again, as randomness was already ensured during sampling.

TensorFlow Dataset Creation For efficient batching and shuffling during training, the `get_tf_dataset` method converts the dataset into a `tf.data.Dataset` object. Shuffling is typically enabled for training and validation datasets but disabled for test datasets.

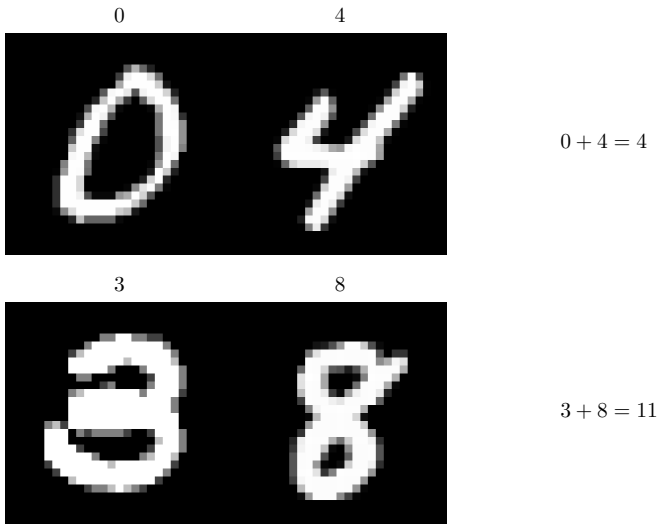
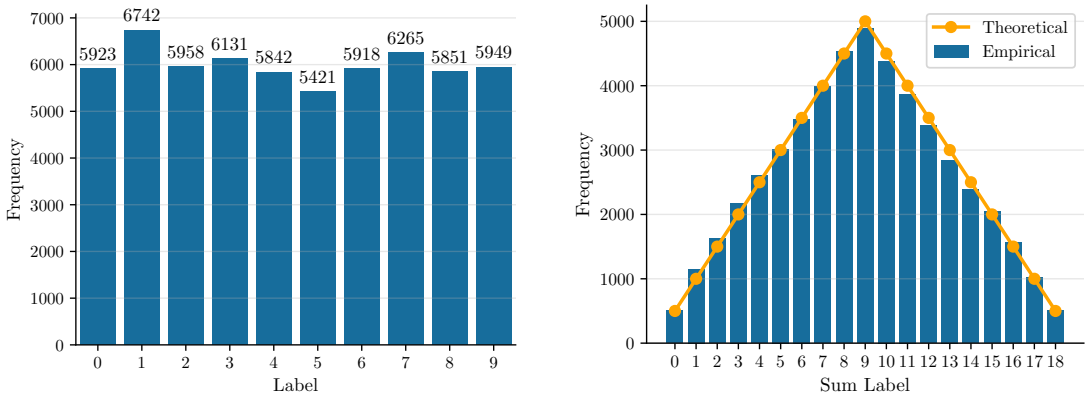


Figure 1: The shape of the generated dataset is `(N, 28, 56)`, representing images of size 56×28 in computer dimensions. Here, N denotes the number of samples in the dataset. Two samples are randomly selected for visualization.

2.2 Statistical Properties



(a) (b)

Figure 2: (a) The label distribution of the MNIST dataset. (b) The label distribution in the training set of the dataset generated by the default instance of the `MNISTAdditionDataset` class (i.e., using `dataset = MNISTAdditionDataset()` followed by `dataset.create_datasets()`), containing a total of 50,000 data points. The orange line represents the theoretical distribution as defined in the equation 1, while the blue bars represent the empirical distribution.

The generated dataset's quality is assessed through its statistical properties. Assuming that the original MNIST dataset is uniformly distributed, the generated dataset should theoretically follow a discrete triangular distribution.

Theoretical Distribution If X and Y are independent random variables uniformly distributed over the integers from 0 to 9, then their sum $S = X + Y$ follows:

$$P(S = s) = \begin{cases} \frac{s+1}{100} & \text{for } s = 0, 1, \ldots, 9, \\ \frac{19-s}{100} & \text{for } s = 10, 11, \ldots, 18. \end{cases} \tag{1}$$

Empirical Analysis Figures 2(a) and 2(b) compare the label distributions of the original and generated datasets. Results confirm alignment with theoretical expectations.

3 Neural Network Pipeline

The main notebook for this part is `notebooks/neural_network_optuna.ipynb`. And the main modules we will use in this section are `src/data_handler.py` and `src/models.py`.

3.1 Dataset Configurations

We use the dataset generated by the default instance of the `MNISTAdditionDataset` class (e.g., `dataset = MNISTAdditionDataset(seed=42)` followed by `dataset.create_datasets()`).

- **Training Dataset**: Input size `(50000, 28, 56)`; Output size `(50000,)`.
- **Validation Dataset**: Input size `(10000, 28, 56)`; Output size `(10000,)`.
- **Test Dataset**: Input size `(10000, 28, 56)`; Output size `(10000,)`.

3.2 Baseline Model

The baseline fully connected neural network, detailed in `notebooks/neural_network.ipynb`, achieves over 90% test accuracy. This section outlines its configuration and chooses five hyper-parameters for further tuning.

TensorFlow Dataset Key configurations include shuffled training and validation datasets (with fixed random seeds for reproducibility), a default batch size of 32, and `prefetch(tf.data.AUTOTUNE)`, which allows the pipeline to fetch and prepare the next batch of data asynchronously while the current batch is being processed, to optimize data pipeline efficiency.

Architecture The neural network will be a normal fully connected network, with various hyper-parameters.

- **Input and Output** - The input layer processes data of shape `(28, 56)`. After flattening, it outputs `(1568,)` (normally in row-major order). The final layer outputs `(19,)`.
- **Batch Normalization** - Applied after each activation function for stability and efficiency. Batch normalization can be strategically implemented between the linear transformation and the activation function; however, in this specific task, it is applied following each activation function.
- **Dropout** - Dropout layers follow batch normalization. The dropout rate will be a tunable hyper-parameter.
- **Dense Layers** - Fully connected layers will have three tunable parameters, including the number of layers, activation functions, and neurons.

Training

- **Optimizer** - The `Adam` optimizer is used, with a tunable learning rate.
- **Epochs** - Fixed at 50 to balance efficiency and overfitting.
- **Batch Size** - Set to 32.
- **Loss Function** - `sparse_categorical_crossentropy` is employed for simplicity. L2 regularization is added to dense layers to mitigate overfitting.

3.3 Hyper-parameter Optimization with Optuna

We optimize the number of dense layers, activation functions, neurons per layer, dropout rate, and learning rate.

Model Class The `DigitAdditionModel` class in `src/models.py` simplifies hyper-parameter testing and model management.

Optuna Optuna[3] provides automated hyper-parameter optimization using efficient search algorithms. The key components include:

We implement a class `OptunaOptimizer` for this specific MNIST addition hyper-parameter tuning task for better organization.

Study Configured using `optuna.create_study()`, which defines sampling and pruning algorithms:

Samplers basically continually narrow down the search space using the records of suggested parameter values and evaluated objective values, leading to an optimal search space which giving off parameters leading to better objective values. Pruners automatically stop unpromising trials at the early stages of the training (a.k.a., automated early-stopping).[1]

We utilize the standard Tree-structured Parzen Estimator algorithm for sampling. Furthermore, a pruning algorithm is omitted because early-stopping, which is set by default, or other callback methods are integrated into each training session as defined in the `DigitAdditionModel` class.

Objective[2] Evaluates each trial using `DigitAdditionModel.build()` and returns validation accuracy.

Optimization We log the search process, store optimal parameters and model weights.

Running See `notebooks/neural_network_optuna.ipynb` for code, where we set the hyper-parameter search range and display results.

Eventually, we present an overall report from a study of 20 trials:

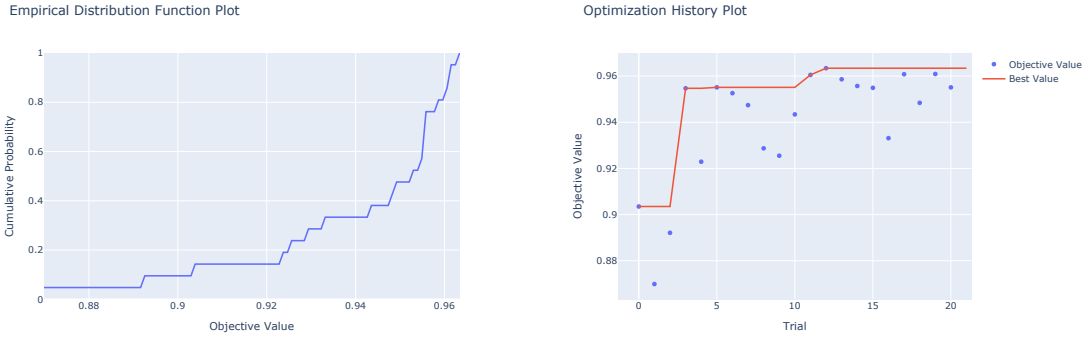
```
Final Results Summary:
Hyperparameter Optimization Results:
--------------------------------------------------

Best Trial:
  Value (Validation Accuracy): 0.9634
  Hyperparameters:
    n_layers: 4
    dropout_rate: 0.17753778267309397
    learning_rate: 0.00040593004933114283
    l2_reg: 1e-06
    activation: elu
    units_l0: 567
    units_l1: 215
    units_l2: 208
    units_l3: 823

Parameter Importance:
  units_l0: 0.4870
  n_layers: 0.3066
  learning_rate: 0.0712
  dropout_rate: 0.0632
  units_l1: 0.0558
  activation: 0.0162
  l2_reg: 0.0000
```

Listing 2: Hyperparameter Optimization Results

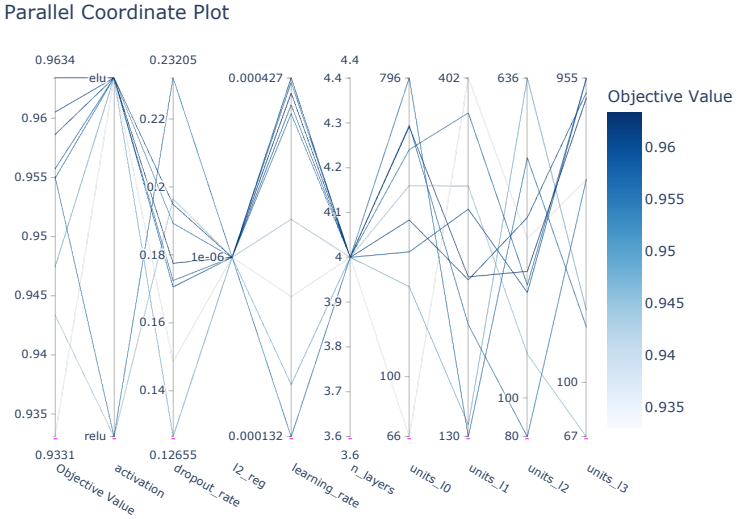
Parameter importance is evaluated using the fANOVA importance evaluator[4], which fits a random forest regression model to predict objective values from parameter configurations of COMPLETE trials. The range of `l2_reg` is non-tunable in our setting and fixed at (1e-6, 1e-6), resulting in an importance of 0.

The search log is in `notebooks/logs/optimization.log`, and the best parameters and model weights are saved in `notebooks/results/optimization_results.json` and `notebooks/models/best_model_weights.weights.h5`, respectively.

The inherent visualizations provided by Optuna reveal several interesting insights during the search process. Refer to Figure 3 for details.



(a) (b)



(c)

Figure 3: The visualization module provided by Optuna was used to generate the plots for this search. (a) **The empirical cumulative distribution function (CDF) of the objective values** (highest validation accuracy corresponding to hyperparameters). Within the given search space, the accuracy of most models exceeds 90%, and the probability density function seems to rise sharply after reaching 95%, indicating that further improving accuracy beyond this point becomes increasingly challenging. (b) **Optimization history**. As the number of searches increases, the accuracy of the parameters found tends to improve, showcasing the advantage of the Optuna search algorithm. (c) **Path lines of parameter combinations**. Darker path lines represent parameter combinations with higher accuracy. Notable observations include: most searches utilized the `elu` activation function; for the `elu` activation function, a dropout rate of 0.17–0.2 tends to be more suitable. Learning rates generally performed better at higher or lower values within the illustrated range $[0.000132, 0.000427]$, with middling values performing less effectively. The value of `n_layers` is generally better at 4, which is also the maximum set value, possibly indicating that more complex models might perform better. Additionally, configurations where the 1st and 4th hidden layers have more units, while the middle two hidden layers have relatively fewer units, tend to perform better.

3.4 Optimal Model Evaluation

The architecute of the best model, as detailed in `notebooks/best_neural_network_tsne.ipynb`, is summarized in Table 1. A visualization can be referred in Figure 4. For all other parameters,

please refer to Section 3.2.

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 28, 56)	0
flatten_1 (Flatten)	(None, 1568)	0
dense_5 (Dense)	(None, 567)	889,623
batch_normalization_4 (BatchNormalization)	(None, 567)	2,268
dropout_4 (Dropout)	(None, 567)	0
dense_6 (Dense)	(None, 215)	122,120
batch_normalization_5 (BatchNormalization)	(None, 215)	860
dropout_5 (Dropout)	(None, 215)	0
dense_7 (Dense)	(None, 208)	44,928
batch_normalization_6 (BatchNormalization)	(None, 208)	832
dropout_6 (Dropout)	(None, 208)	0
dense_8 (Dense)	(None, 823)	172,007
batch_normalization_7 (BatchNormalization)	(None, 823)	3,292
dropout_7 (Dropout)	(None, 823)	0
dense_9 (Dense)	(None, 19)	15,656

Table 1: Neural Network Model Summary

The optimal model achieved a test accuracy of 94.84%. Training and validation curves (Figure 5) indicate potential overfitting after 20 epochs.

We visualize misclassified data in Figure 6. Some MNIST digits are indistinguishable even to humans, affecting accuracy and generalization.

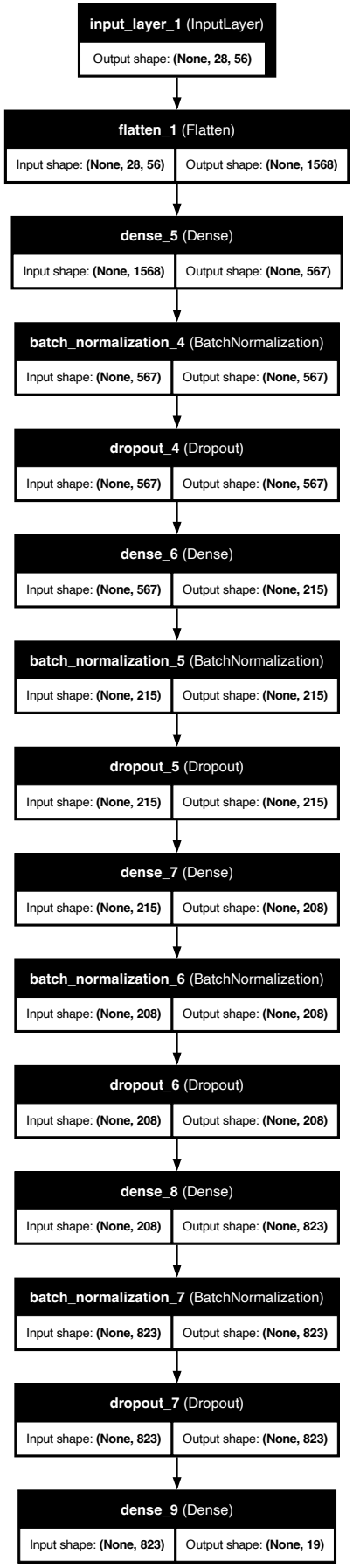


Figure 4: Architecture of the optimal fully connected neural network.

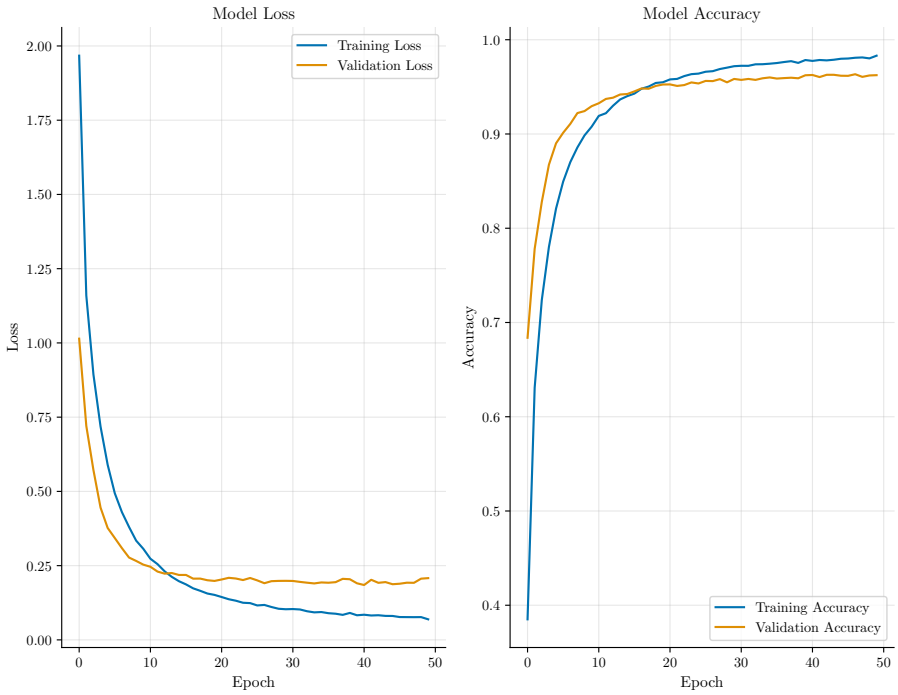


Figure 5: The loss and accuracy curves during the training process of the optimal model. It can be observed that after approximately the 20th epoch, the performance on the validation set stagnates, while the training set performance continues to improve, indicating a sign of overfitting.

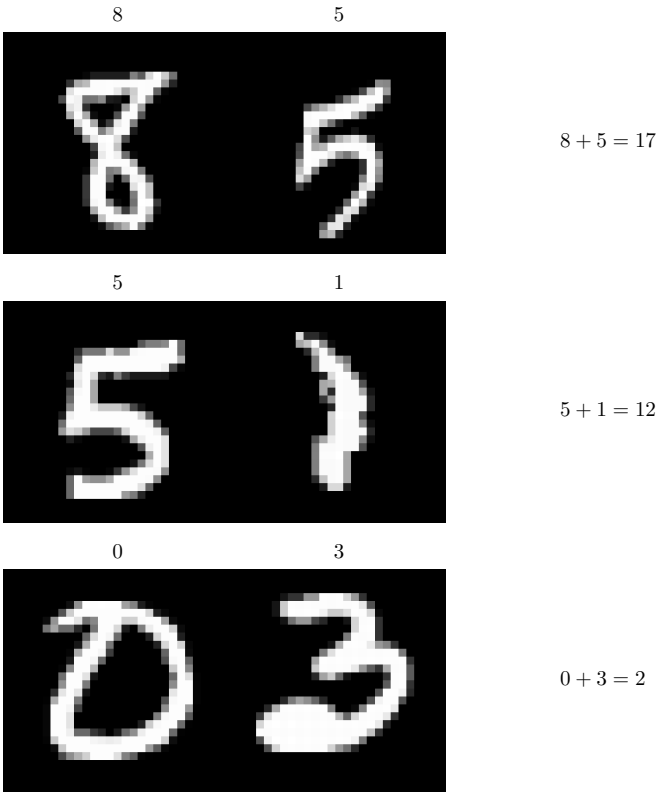


Figure 6: Examples of some misclassified data on the test set.

4 Random Forest and Support Vector Machine

This section provides an evaluation of the performance of Random Forest and Support Vector Machine (SVM) classifiers for the MNIST addition task.

The implementation is detailed in `notebooks/classifier_comparison.ipynb`, where specific configurations are presented.

We use the flattened image as input. However, feature engineering, which is not discussed in this report, is also a significantly important and large topic.

Moreover, given the non-uniform distribution of sample numbers across classes, certain techniques will be employed to mitigate this imbalance.

Random Forest Random Forest classifiers are robust ensemble learning models that aggregate decisions from multiple decision trees.

An approach to address the issue of imbalance in Random Forest classifiers involves changing the split criterion during the tree construction process. Two fundamental criteria are `'gini'` and `'entropy'`.

- **Gini Index** - The default criterion in `RandomForestClassifier` from `scikit-learn`. Gini Index measures the impurity of a dataset based on the probability of misclassifying a randomly chosen element. While computationally efficient, it may exhibit bias toward features with fewer categories.
- **Entropy** - Derived from information theory, entropy measures dataset impurity or disorder and uses information gain to optimize tree splits.

Moreover, using `class_weight='balanced'` setting counteracts imbalance by automatically adjusting class weights inversely to their frequency, giving minority classes more attention in training [5].

Six Random Forest configurations were evaluated, with results summarized in Figure 7. Despite varied configurations, none surpassed the test accuracy of neural networks, with all models achieving less than 80% accuracy.

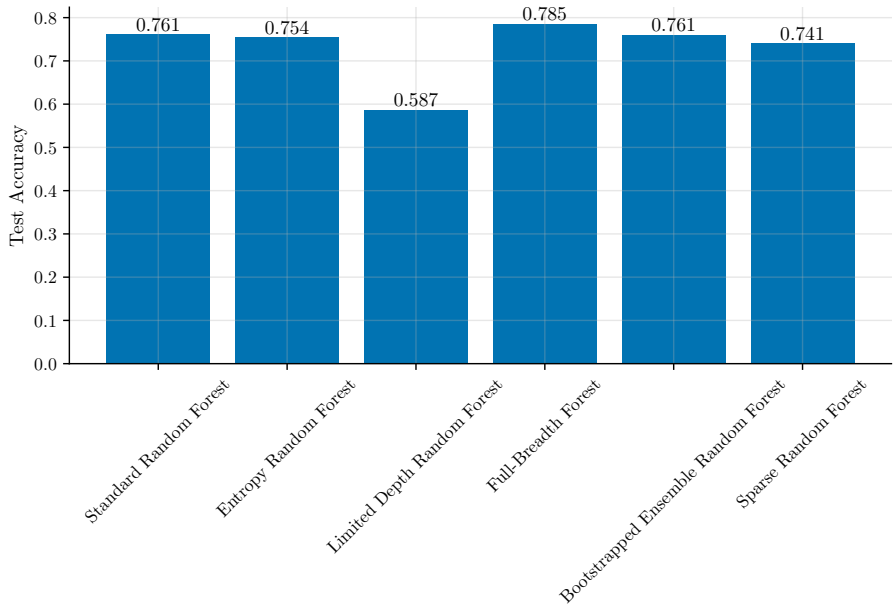


Figure 7: Test accuracy for six Random Forest models. Limited Depth and Full-Breadth Forest configurations differ in tree depth, number of trees, and leaf node requirements. It has been shown that more complex Random Forest models perform better for this problem. The two models on the far right configure the `max_samples` and `max_features` used by each base learner, respectively.

Support Vector Machine (SVM) SVMs are computationally intensive, particularly on high-dimensional datasets like MNIST. Consequently, we trained SVM models on a reduced dataset comprising 40% of the original training data.

Seven SVM configurations were tested, and their results are presented in Figure 8. Overall, SVMs performed worse than both Random Forest and neural networks. Linear Kernel SVM and low-penalty (C) RBF SVM especially exhibited lower performance.

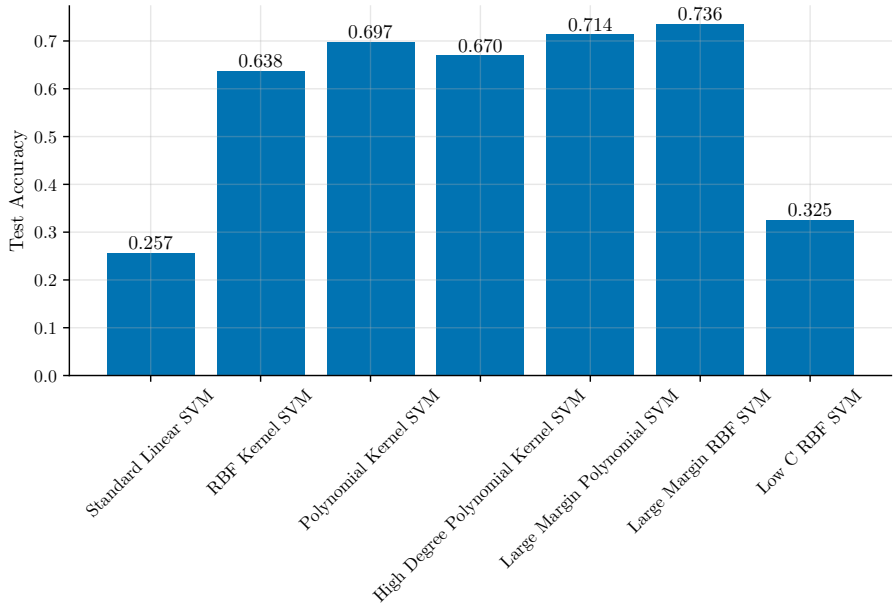


Figure 8: Test accuracy for seven SVM configurations.

5 Linear Classifiers

In `notebooks/linear_classifiers.ipynb`, we implement two types of classifiers on the 56×28 dataset: the **Joint Classifier**, which predicts the sum directly, and the **Sequential Classifier**, which predicts each digit sequentially and then calculates their sum.

We evaluate these classifiers across various training set sizes (`[50, 100, 500, 1000, 10000, 50000]`) using a fixed test set size of `10000`. Figure 9 illustrates the average confidence for different training sizes and the confidence distribution for classifiers trained on the largest dataset (50000 samples). Confidence is defined as the highest softmax probability predicted for each test data point.

The test set performance is shown in Figure 10. The results indicate that the Sequential Classifier significantly outperforms the Joint Classifier in both accuracy and confidence, making it a more effective design.

All in all, Sequential Classifier design has much higher confidence and accuarcy.

6 t-Distributed Stochastic Neighbor Embedding (t-SNE) Distribution

The t-SNE is a powerful dimensionality reduction technique. It projects high-dimensional data into lower-dimensional spaces, preserving meaningful structure.

Perplexity and Its Role Perplexity is a critical hyperparameter in t-SNE that balances global and local data structures. Higher perplexity emphasizes broader relationships, while lower values focus on local clustering.

In `notebooks/best_neural_network_tsne.ipynb`, we visualize the first two t-SNE dimensions for the embedding layer of the optimal neural network (see Section 3.4) and the raw input dataset. The chosen perplexity values result from multiple trials to optimize visualization.

The results, shown in Figures 11 and 12, demonstrate that the neural network effectively transforms unstructured input data into meaningful feature representations within its embedding layer.

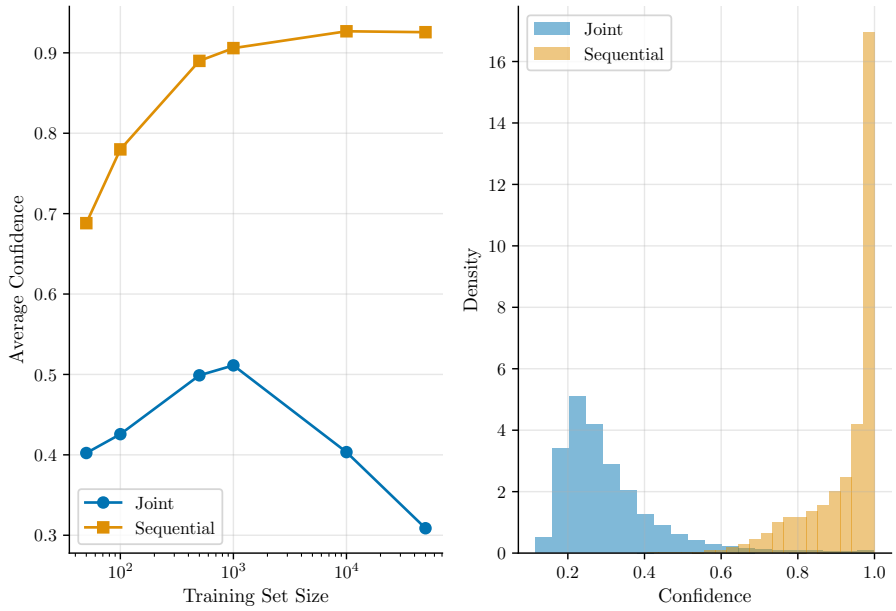


Figure 9: Left: Average confidence of the test set across different training set sizes. As the size increases, the confidence of the Joint Classifier decreases after a certain limit, whereas the Sequential Classifier improves, reaching 90%. Right: Confidence distribution for the test set at a training size of 50,000.

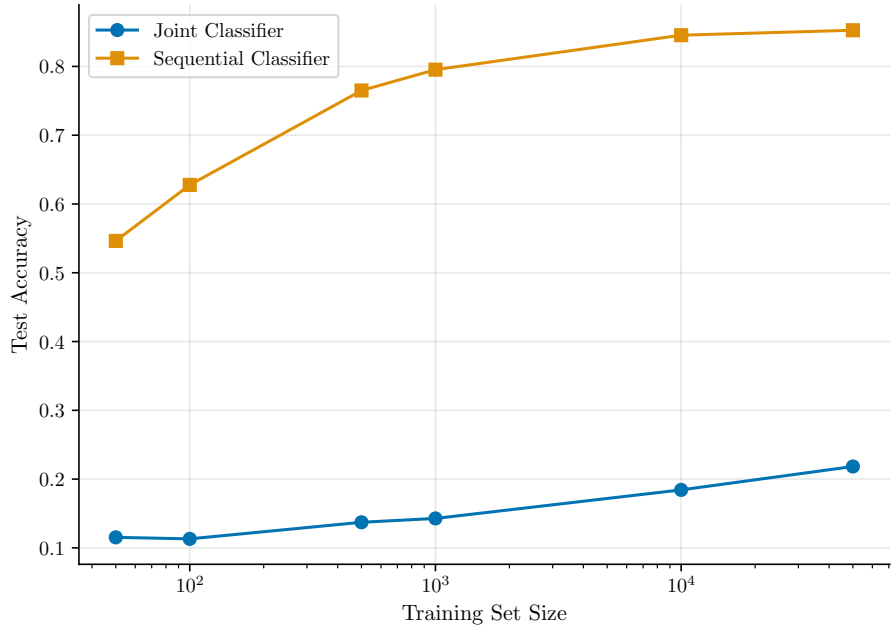


Figure 10: Sequential Classifier accuracy steadily increases, surpassing 80% as the training size grows. The Joint Classifier also improves but remains weak.

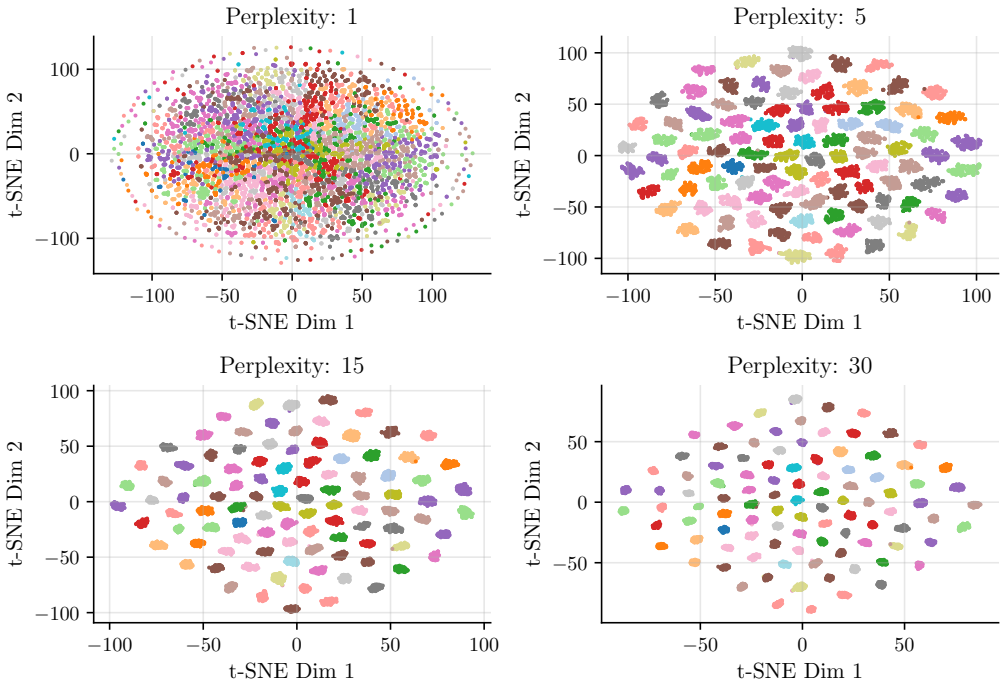


Figure 11: t-SNE visualization of the embedding layer with different perplexity values. Clusters are evident at Perplexity=5. While 19 colors represent the 19 classes, the observed number of clusters exceeds 19, likely reflecting different additive combinations for the same sum.

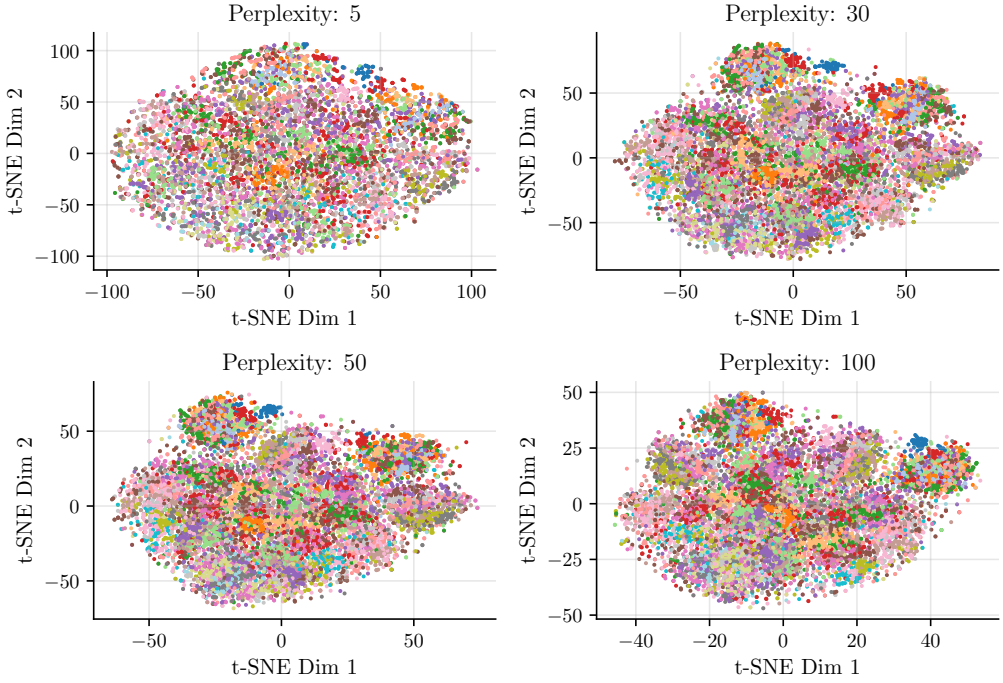


Figure 12: t-SNE visualization of the raw input dataset under various perplexity settings. No discernible structure emerges, regardless of perplexity.

7 Conclusion

This study demonstrates that neural networks, particularly when optimized with advanced hyper-parameter tuning techniques like Optuna, outperform traditional machine learning models such as Random Forest and SVM for the MNIST Addition task. Sequential Classifier designs within linear classifiers also show promise, achieving competitive accuracy and confidence. Additionally, t-SNE visualizations highlight the capability of neural networks to transform unstructured input data into meaningful feature representations. These findings reinforce the importance of tailored model design and hyper-parameter optimization in achieving high performance.

A Use of AI Tools

The following describes how AI tools were utilized in the preparation of this report:

- ChatGPT 4o
  - **Drafting** - Used for drafting certain sections of the report, including the Introduction, t-SNE section, Conclusion, and Appendix.
  - **Proofreading** - Reviewed grammatical correctness, improved sentence structure, and suggested alternative wordings for clarity across the report.
  - **LaTeX Assistance** - Helped resolve issues encountered during the LaTeX formatting process, such as debugging compilation errors, optimizing figure placement, and improving overall document layout.

B Template Information

This report was formatted using the LaTeX template from the given Example Coursework (by Michal Dorko). Some modifications were made.

References

[1] *Efficient Optimization Algorithms — Optuna 4.1.0 documentation.* URL: https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/003_efficient_optimization_algorithms.html. (accessed: 12.12.2024).

[2] *Lightweight, versatile, and platform agnostic architecture — Optuna 4.1.0 documentation.* URL: https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/001_first.html. (accessed: 12.12.2024).

[3] *Optuna - A hyperparameter optimization framework.* URL: https://optuna.org/. (accessed: 11.12.2024).

[4] *optuna.importance.get_param_importances — Optuna 4.1.0 documentation.* URL: https://optuna.readthedocs.io/en/stable/reference/generated/optuna.importance.get_param_importances.html. (accessed: 12.12.2024).

[5] *RandomForestClassifier — scikit-learn 1.6.0 documentation.* URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. (accessed: 12.12.2024).