Machine Learning Coursework Assignment

Adding two MNIST digits through an inference pipeline

Laura Just Fung (lj441)

December 18, 2024

Word count: 2994

1 Introduction

MNIST is a dataset consisting of handwritten digits and their labels commonly used for training machine learning models and image processing systems [1]. The classical problem represented by the MNIST dataset is for machine learning algorithms to be able to learn to recognise handwritten digits, no matter how they are shaped.

For this project, the problem at hand is not just for the inference pipeline to learn to recognise MNIST digits accurately, it must also learn addition.

First, a stratified dataset of paired MNIST digits will be generated, along with the label denoting their sum. Then, a neural network will be trained and perform supervised learning on this dataset. Hyperparameter tuning will be performed using `Optuna`.

To explore this problem further, a SVM, a random forest, and an AdaBoost classifier were also trained on the MNIST dataset. Weak linear classifiers were also used.

The t-SNE distribution of the dataset and the embedding layer of the best-performing neural network were also extracted and analysed, optimising for perplexity.

2 Dataset generation

The dataset upon which the models were trained on was first loaded in from MNIST using `Tensorflow`. For datasets greater than and divisible by 100, stratified data sampling was used, such that each permutation of digit pairs was represented equally. For datasets less than 100 samples in size, each pair was randomly picked from a list of all possible permutations without replacement.

For larger datasets, equal representation of the pairs was judged to be more important than sampling without replacement from the MNIST dataset. The risk of data leakage and the models learning from the handwriting shapes rather than the numbers was judged to be minimal, as the training and validation/testing data were drawn from separate parts of the MNIST dataset. This was achieved by splitting the MNIST dataset when it was first loaded and then constructing the training and validation/testing datasets from each subset separately. The results of these samplings can be found in Fig. 1 and Fig. 2 respectively.

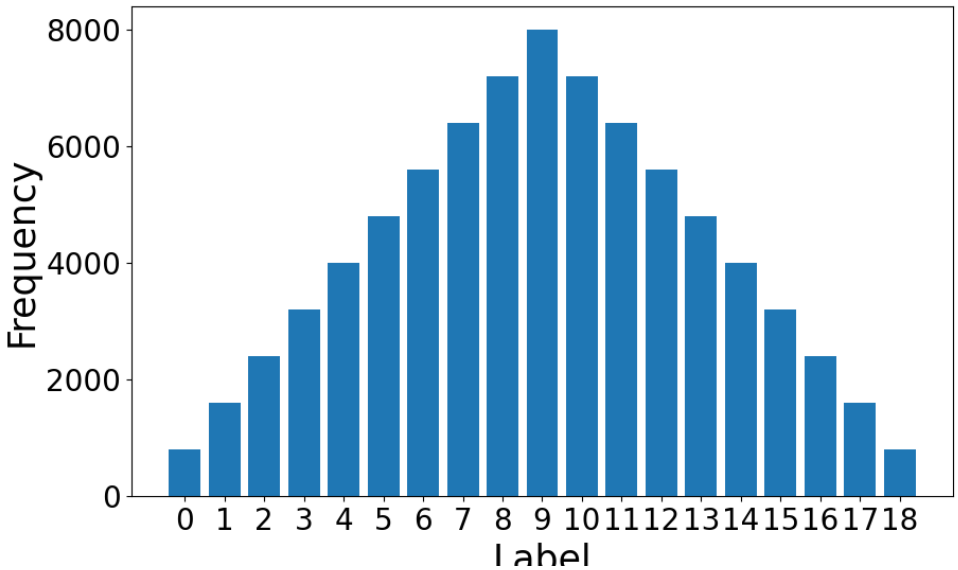

Figure 1: Distribution of labels in a dataset of 80,000 samples.

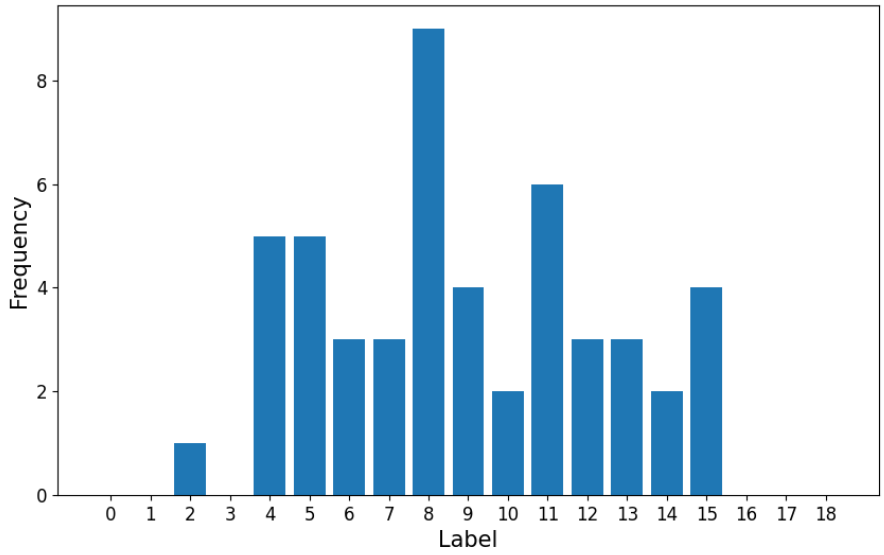

Figure 2: Distribution of labels in a dataset of 50 samples.

3 Neural network pipeline

`Tensorflow` was used to create the neural network architecture and `Optuna` was used to tune the hyperparameters for this architecture.

3.1 Neural networks

The neural network used for this project is a `Sequential` model, where each layer has exactly one input tensor and one output tensor. This is a feedforward neural network, where information passes through the layers in only one direction, starting at the input layer, through the hidden `Dense` layers, and exiting at the output layer. In the hidden layers, each neuron computes a weighted sum of its inputs, adds a bias, and then applies a non-linear activation function to generate the layer's output. The outputs of each `Dense` layer become the inputs of the following `Dense` layer. The final output layer, which in this project has as many neurons as there are classes, produces the probabilities of the digit pair adding up to be any of the numbers in the range 0–18.

For this project, the rectified linear unit (ReLU) activation function was used. This is because ReLU tends be computationally efficient and shows better convergence than sigmoid functions [2].

In order to train feed forward neural networks, a loss function is required. For this project, the loss function $\mathscr{L}(\theta, \hat{\theta})$ was the categorical cross-entropy loss function, which is used for multi-class classification problems. It is defined as

$$\mathscr{L}(\theta, \hat{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}\theta_{ij}\log\hat{\theta_{ij}}, \tag{1}$$

where θ_{ij} is the true value of the ith element belonging to the jth class and $\hat{\theta_{ij}}$ is the predicted value of that element. This function is used to calculate the loss during training with the backpropagation algorithm, which computes the gradients of the loss with respect to each weight by applying the chain rule. This allows the neural network to learn how to adjust its weights by optimising this function. The weights are adjusted using the Adam optimisation function in the following process:

1. The model parameters θ; learning rate α; and Adam hyperparameters β_1, β_2, and ε are initialised.

2. The gradient g of the loss function $\mathscr{L}$ with respect to the model parameters θ are computed:
$$g^{(t)} = \nabla\mathscr{L}(\theta^{(t-1)}) \tag{2}$$

3. The first moment estimate m is updated:
$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)g^{(t)} \tag{3}$$

4. The second moment estimate v is updated:

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)[g^{(t)}]^2 \tag{4}$$

5. The biases for m and v are corrected:

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t}, \qquad \hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t} \tag{5}$$

6. The adaptive learning rate $\alpha^{(t)}$ is updated:

$$\alpha^{(t)} = \frac{\alpha^{(t-1)}\sqrt{1-\beta_2^t}}{1-\beta_1^t} \tag{6}$$

7. Finally, the model parameters $\theta^{(t)}$ are updated:

$$\theta^{(t)} = \theta^{(t-1)} \frac{\alpha^{(t)}\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\varepsilon} \tag{7}$$

3.2 Hyperparameter Tuning

For this project, 5 hyperparameters were tuned: the initial learning rate for the Adam optimiser, the dropout rate of the dropout layers, the L2 regularisation penalty, the number of layers, and the batch size carried over from each layer. In addition to this, batch normalisation was also used.

The Adam optimisation algorithm is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments [3]. It was used as it is computationally efficient and well-suited to large-scale problems, as advised by Kingma and Ba [3]. The inital learning rate $\alpha^{(0)}$ was tuned for the optimal rate of adaptative estimation. Too-large of a learning rate can lead to missing the minimum entirely, while too-small of a learning rate leads to computationally-inefficient code.

The dropout rate of the dropout layers was tuned so that the resulting neural network would not over-fit to the training data but also not forget what it was meant to be fitting during training [4]. In `Tensorflow`, the `Dropout` layer randomly resets neurons to zero with a certain frequency. The rest of the neurons are scaled up such that the total sum over all inputs remain unchanged.

Regularisation is when a penalty is added to the loss function $\mathscr{L}$ in order to prevent overfitting [4]. L2 regularisation, or ridge regression, adds the squared sum of the coefficients as the penalty term. Adding L2 regularisation to Eq. 1 results in the following equation:

$$\mathscr{L}(\theta, \hat{\theta}, \alpha_{L2}, \mathrm{w}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}\theta_{ij}\log\hat{\theta}_{ij} + \alpha_{L2}||\mathrm{w}||_2^2, \tag{8}$$

where θ_{ij} and $\hat{\theta}_{ij}$ mean the same things, α_{L2} refers to the penalty parameter, and $||\mathrm{w}||_2^2$ is the L2 norm squared of the weight vector of the parameters.

L2 regularisation was preferred over L1 regularisation as its derivative is smooth and continuous and thus less computationally expensive.

The number of layers were tuned so that the neural network would neither overfit nor underfit. The range of the number of layers to tune was restricted so that neither too many layers were attempted, which would have required a lot of computational resources, nor too few, which would have failed to properly capture all relevant features of the dataset.

The batch size was also tuned to determine the number of training samples used in one epoch of training. Larger batch sizes represent more of the total dataset, which helps in computing more accurate estimations of the gradient, but are more computationally expensive and also might lead to overfitting. Smaller batch sizes tend to introduce noise, but are faster to compute.

The number of neurons in each layer was not a hyperparameter that was tuned using `Optuna`. Instead, the number of neurons were dependant upon the number of layers and constructed such that each successive layer has fewer neurons than the last. This can be represented as:

$$N_k = 2^{N-1-k} \times 64, \quad \text{for } k = 0, 1, 2, \ldots, N-1, \tag{9}$$

where N_k is the number of neurons in layer k.

Batch normalisation is a regularisation scheme that causes the inputs to be centered about zero with respect to the bias [4]. This prevents neuron saturation and gradients vanishing, improving the learning speed. This was implemented by adding `BatchNormalization` layers which standardised the inputs by the mean and variance of the mini-batch.

The results of the `Optuna` hyperparameter tuning is shown in Table 1, which contains the optimal hyperparameters. As the final architecture contains 6 hidden `Dense` layers, the number of neurons in each layer range from 2048 to 64, descending, as seen in Table 2. The architecture of the best-performing neural network is shown in Table 3.

Hyperparameter	Value
Adam initial learning rate α	0.000236 (3 s.f.)
Dropout rate	0.104 (3 s.f.)
L2 penalty parameter α_{L2}	0.000189 (3 s.f.)
Number of layers	6
Batch size	69

Table 1: Values of the hyperparameters of the best-performing neural network.

	`Dense` Layer					
	1	2	3	4	5	6
Neurons	2048	1024	512	256	128	64

Table 2: The number of neurons in each layer of the best-performing neural network.

Layer	Type	Additional information
1	Input layer	28×56 input shape
2	`Dense` layer	2048 neurons
3	`Dropout` layer	0.104 dropout rate
4	`BatchNormalization` layer	
5	`Dense` layer	1024 neurons
6	`Dropout` layer	0.104 dropout rate
7	`BatchNormalization` layer	
8	`Dense` layer	512 neurons
9	`Dropout` layer	0.104 dropout rate
10	`BatchNormalization` layer	
11	`Dense` layer	256 neurons
12	`Dropout` layer	0.104 dropout rate
13	`BatchNormalization` layer	
14	`Dense` layer	128 neurons
15	`Dropout` layer	0.104 dropout rate
16	`BatchNormalization` layer	
17	`Dense` layer	64 neurons
18	`Dropout` layer	0.104 dropout rate
19	`BatchNormalization` layer	
20	Output layer	19 neurons

Table 3: The architecture of the best-performing neural network.

3.3 Results

The best-performing neural network was evaluated using `Tensorflow`'s `tf.keras.Model.evaluate` method. The results of this can be seen in Table 4.

Metrics	Value (3 s.f.)
Accuracy	0.955

Table 4: The metrics of the best-performing neural network as calculated by `tf.keras.Model.evaluate`.

The resulting confusion matrix is shown in Fig. 3. The rows represent the true class label of each digit pair whilst the columns represent the predicted class labels from the neural network. Off-diagonal elements represent misclassifications.

An example of the neural network in action can be seen in Fig. 4. The neural network's predictions are quite accurate, except for the first image, where the number '8' is quite deformed. It is likely that the neural network mistook it as a '7' instead, and then proceeded to correctly perform addition: $7 + 1 = 8$.

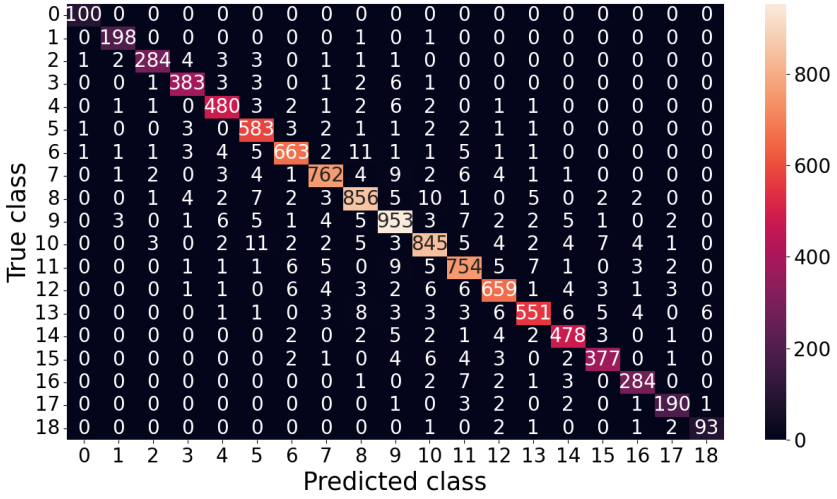

Figure 3: Confusion matrix of the best-performing neural network.

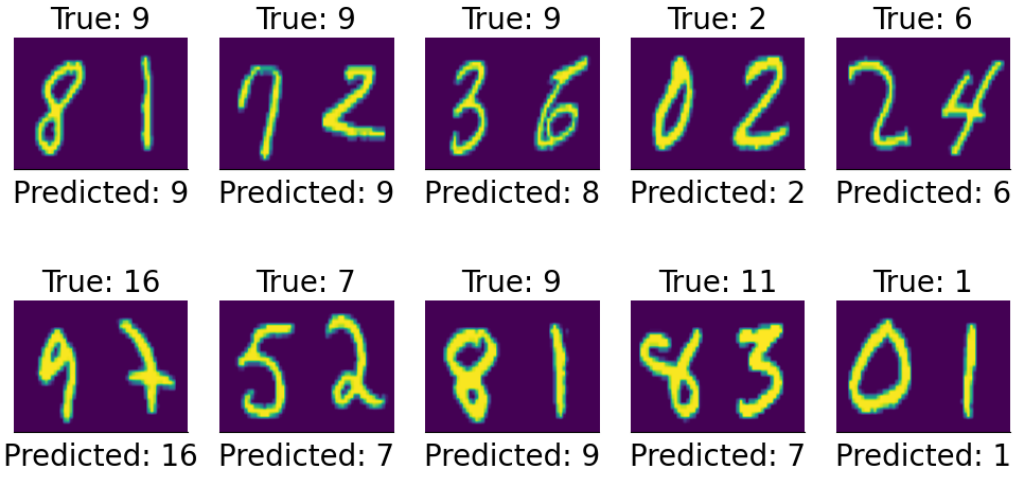

Figure 4: Ten images of random digit pairs, their true labels, and the corresponding predictions given by the best-performing neural network.

4 Other inference algorithms

To explore this problem further, other inference algorithms were used, including SVMs, Random Forest classifiers, and AdaBoost classifiers.

4.1 SVM

SVM stands for Support Vector Machine, which is a supervised learning method used for classification in machine learning. SVMs find the optimal hyperplane for a linearly separable pattern.

The SVM classifier was implemented using `scikit-learn`'s `sklearn.svm.SVC` class. `Optuna` was used to tune the γ, c, and polynomial degree hyperparameters. The polynomial kernel trick produced the best-performing SVM classifier during initial tuning, thus that kernel trick was the one used for further tuning.

During training, it was found that the best-performing SVM was one with a large number of training samples. However, SVMs have an order complexity around $\mathcal{O}(N^2)$ [5] and it is for this reason `Optuna` hyperparameter tuning was limited to 10,000 samples and training was limited to 64,000 samples.

The hyperparameters of the best-performing SVM can be found in Table 5. This was evaluated using `scikit-learn`'s `sklearn.metrics.accuracy_score` and `classification_report` methods, resulting in Table 6. Precision refers to the positive predictive value and is the fraction of predicted positive results that are true. Recall refers to the fraction of true positive results that are predicted to be true. The F-score is calculated from the precision and recall, as seen in Eq. 10 and ranges from 0 to 1. The higher the F-score, the better. The resulting confusion matrix can be seen in Fig. 5.

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{10}$$

Hyperparameter	Value
Kernel	Polynomial
Degree	3
Gamma (γ)	21.8 (3 s.f.)
C	1.61 (3 s.f.)

Table 5: Values of the hyperparameters of the best-performing SVM.

Class	Precision (2 s.f.)	Recall (2 s.f.)	F1-Score (2 s.f.)	Support
0	0.86	0.96	0.91	160
1	0.89	0.99	0.93	320
2	0.87	0.89	0.88	480
3	0.85	0.87	0.86	640
4	0.86	0.86	0.86	800
5	0.82	0.85	0.83	960
6	0.82	0.82	0.82	1120
7	0.80	0.80	0.80	1280
8	0.77	0.79	0.78	1440
9	0.80	0.81	0.80	1600
10	0.77	0.78	0.78	1440
11	0.76	0.72	0.74	1280
12	0.79	0.78	0.79	1120
13	0.79	0.78	0.78	960
14	0.81	0.77	0.79	800
15	0.82	0.76	0.79	640
16	0.81	0.79	0.80	480
17	0.81	0.76	0.78	320
18	0.81	0.88	0.84	160
Total accuracy (2 s.f.)		0.80		16000

Table 6: Classification report of the best-performing SVM along with the total accuracy of the SVM's performance on the test dataset.

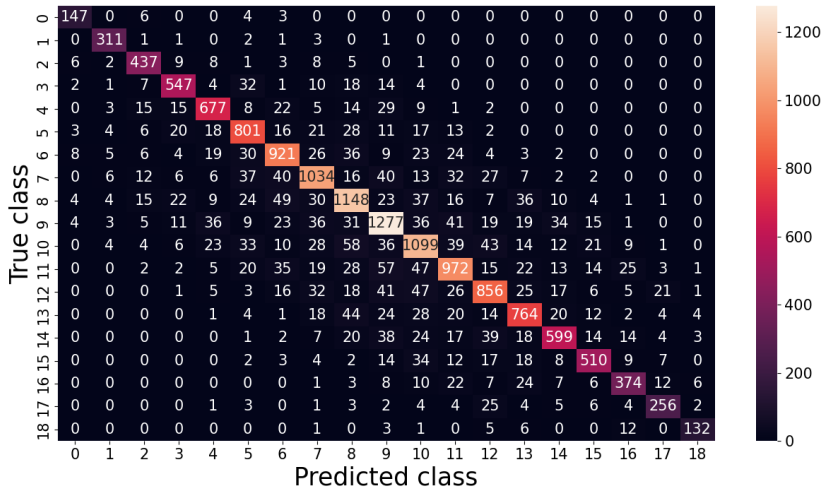

Figure 5: Confusion matrix of the best-performing SVM.

4.2 Random forest

Random forest classifiers are a type of ensemble classifier, where multiple decision trees are used in aggregate in order to calculate predicted labels. Hyperparameter tuning and training for the best-performing random forest was much quicker than for SVMs and offered similar accuracies. This is because the time complexity for training random forest is on the order $\mathcal{O}(NMD\log(N))$, where N is the number of samples, M the number of classes, and D the number of decision trees [6].

Hyperparameter tuning used for their maximum depth, the number of allowed decision trees, the criterion to measure the quality of a split, and the method to calculate the maximum number of features to consider when looking for the best split. The optimal hyperparameters can be found in Table 7.

Hyperparameter	Value
Depth	591
Trees	249
Criterion	gini
Maximum features	$\sqrt{19}$

Table 7: Values of the hyperparameters of the best-performing random forest.

'Gini' refers to the Gini impurity [7], where the impurity of a node is calculated through Eq. 11

$$\text{gini} = 1 - \sum_i p_i^2, \tag{11}$$

where p_i refers to the probability of class i. When the impurity is zero, all elements within a node are of the same class.

The results of the best-performing random forest classifier's evaluation can be found in Table 8. The random forest's confusion matrix is shown in Fig. 6.

Class	Precision (2 s.f.)	Recall (2 s.f.)	F1-Score (2 s.f.)	Support
0	0.90	0.97	0.94	160
1	0.88	0.97	0.92	320
2	0.88	0.86	0.87	480
3	0.87	0.84	0.86	640
4	0.86	0.81	0.83	800
5	0.85	0.80	0.83	960
6	0.82	0.81	0.81	1120
7	0.84	0.79	0.81	1280
8	0.73	0.80	0.77	1440
9	0.73	0.84	0.78	1600
10	0.75	0.78	0.76	1440
11	0.71	0.77	0.74	1280
12	0.70	0.79	0.74	1120
13	0.78	0.76	0.77	960
14	0.83	0.65	0.73	800
15	0.88	0.73	0.80	640
16	0.79	0.66	0.72	480
17	0.82	0.60	0.69	320
18	0.87	0.63	0.73	160
Total accuracy (2 s.f.)		0.79		16000

Table 8: Classification report of the best-performing random forest along with the total accuracy of the random forest's performance on the test dataset.

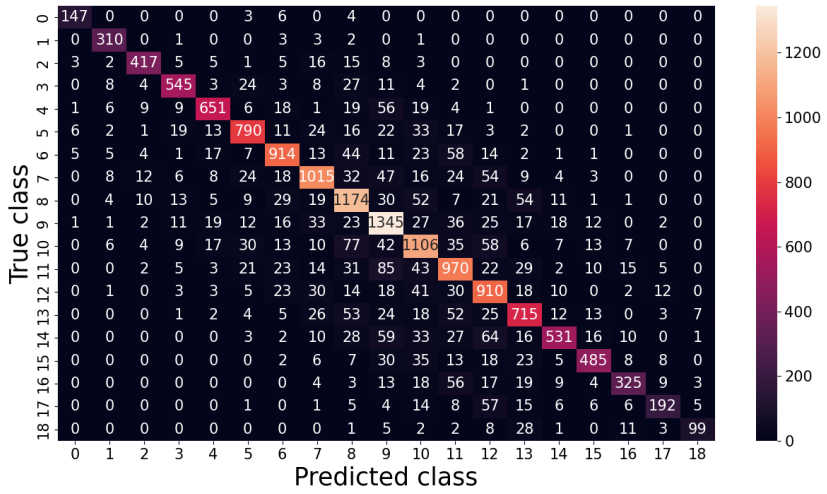

Figure 6: Confusion matrix of the best-performing random forest.

4.3 AdaBoost

AdaBoost classifiers are another type of ensemble classifier, which trains multiple weak classifiers subsets of the training data. In each iteration, the AdaBoost algorithm assigns higher weights and thus focuses more on misclassified samples from previous iterations. This allows the weak classifiers to improve their performance by spending the majority of their time training on the previously misclassified samples.

The AdaBoost classifier used in this project used the decision stump classifier as its weak classifier. Hyperparameter tuning was performed on the number of decision trees used. The number of hyperparameters tuned was deliberately limited as tuning and training of the AdaBoost classifier took very long amounts of time and did not produce very accurate classifiers. This is to be expected as decision stumps do not perform very well on multi-class datasets. However, this classifier still produced an accuracy of 14%, which is still better than random guessing, which has an accuracy of 5.2% for a 19-feature dataset.

The optimal hyperparameters can be found in Table 9. The result of the AdaBoost classifier evaluation can be found in Table 10 and the resulting confusion matrix is shown in Fig. 7.

Hyperparameter	Value
Stumps	53

Table 9: Values of the hyperparameters of the best-performing AdaBoost.

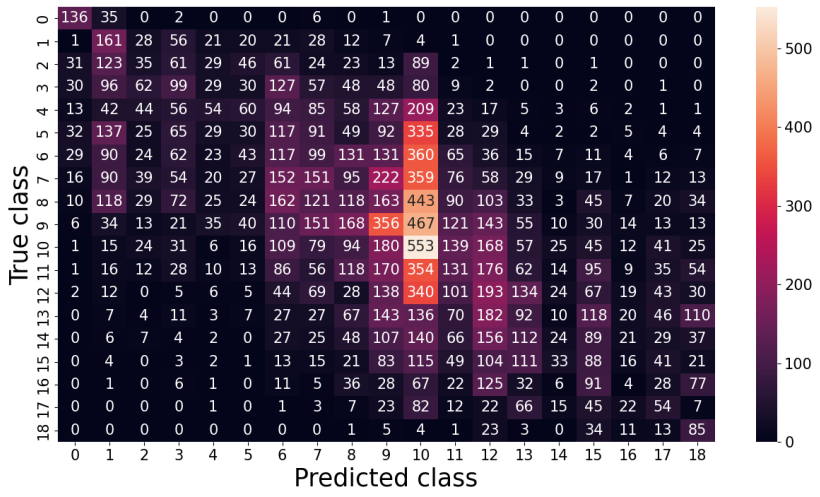

Figure 7: Confusion matrix of the best-performing AdaBoost classifier.

Class	Precision (2 s.f.)	Recall (2 s.f.)	F1-Score (2 s.f.)	Support
0	0.90	0.97	0.94	160
1	0.88	0.97	0.92	320
2	0.88	0.86	0.87	480
3	0.87	0.84	0.86	640
4	0.86	0.81	0.83	800
5	0.85	0.80	0.83	960
6	0.82	0.81	0.81	1120
7	0.84	0.79	0.81	1280
8	0.73	0.80	0.77	1440
9	0.73	0.84	0.78	1600
10	0.75	0.78	0.76	1440
11	0.71	0.77	0.74	1280
12	0.70	0.79	0.74	1120
13	0.78	0.76	0.77	960
14	0.83	0.65	0.73	800
15	0.88	0.73	0.80	640
16	0.79	0.66	0.72	480
17	0.82	0.60	0.69	320
18	0.87	0.63	0.73	160
Total accuracy (2 s.f.)		0.79		16000

Table 10: Classification report of the best-performing AdaBoost along with the total accuracy of the AdaBoost's performance on the test dataset.

5 Weak linear classifiers

To explore this dataset further, the results of a single linear regression classifier trained on the whole image of the digit pair was compared to that of a linear regression classifier that was fitted to each digit of the pair sequentially. That is to say, the first classifier was trained on digit pairs, while the second was on trained on single digits.

An example of the resulting classifier probabilities is shown in Figures 8 and 9. The result is that the single classifier naturally has probabilities for label classes ranging from 0-18, whilst the sequential classifier's probability must be manually calculated, as it natively has probabilities for label classes ranging from 0-9. This calculation is trivial, as the digits of each pair are independent of each other and thus their probabilities can be directly multiplied.

The accuracy is also compared in Fig. 10. As expected, the sequential classifier is more accurate by the single classifier as it is simply trained on single digit recognition and the labels are combined after the fact to produce the final labels. In comparison, the single linear regression classifier has the much harder task of both recognising the digit pair and performing addition.

For both classifiers, the most significant jump in accuracy is when the number of samples go from 50 to 100, as there are 100 possible permutations of digit pairs. After that, the accuracy still increases, but there is not as great of an improvement as before. The accuracies for both the single and sequential linear regression classifier can be found in Table 11.

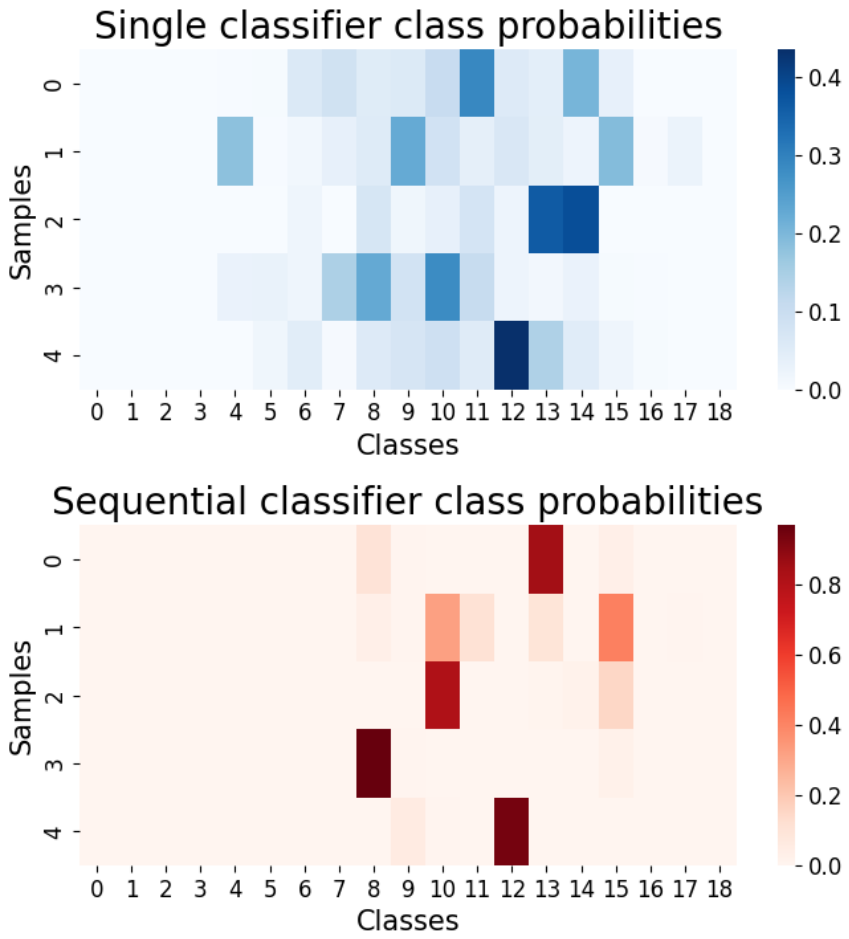

Figure 8: Heatmaps of classifier probabilites for a single classifier (top) and a sequentially applied classifier (bottom). Shown are the first five class probabilities for linear regression classifiers trained on 10,000 samples.

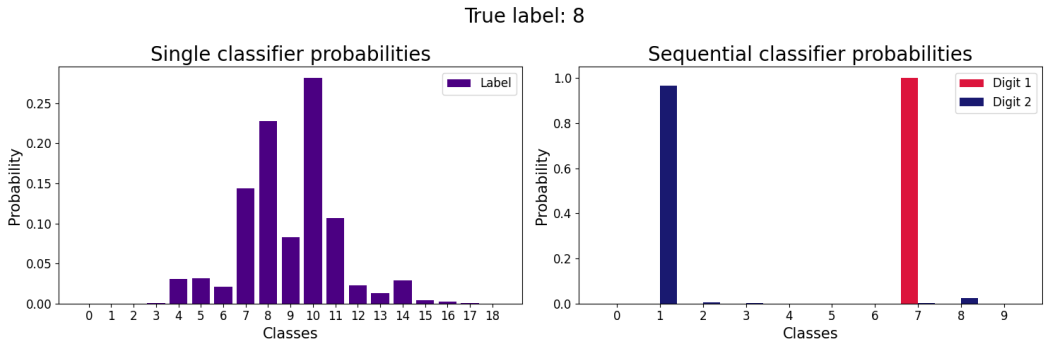

Figure 9: Barplot of classifier probabilites for a single classifier (left) and a sequentially applied classifier (right). Shown specifically are the probability distributions of sample 3 from Fig. 8.

	Sample sizes				
Accuracy	50	100	500	1000	10000
Single classifier (2 s.f.)	0.08	0.10	0.11	0.14	0.18
Sequential classifier (2 s.f.)	0.36	0.71	0.73	0.75	0.82

Table 11: Accuracies of the single and sequential linear regression classifiers as sample sizes increase.

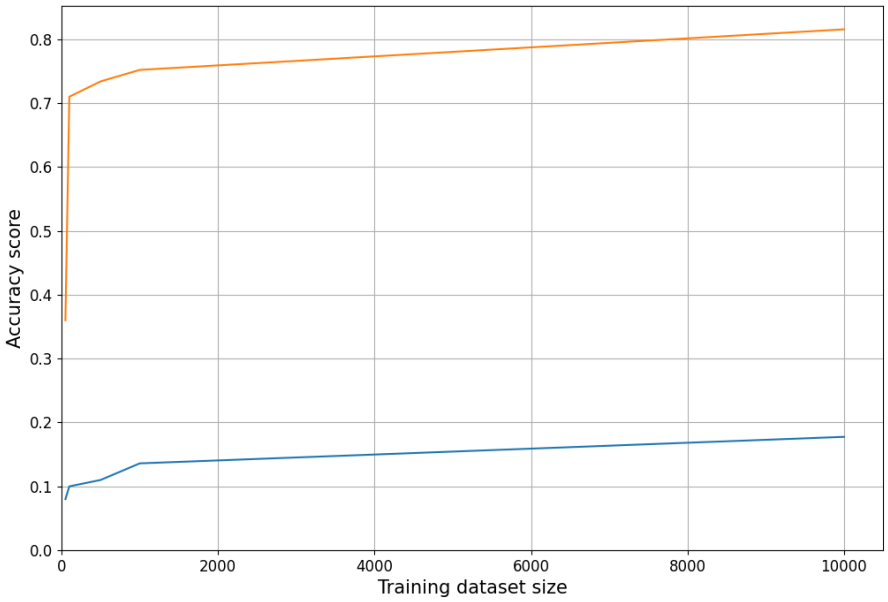

Figure 10: Accuracy comparison between single and sequential linear regression classifiers as dependant on sample size.

6 t-SNE distributions in neural networks

The t-SNE distributions of the raw input data and the embedding layer of the best-performing neural network were obtained and optimised for perplexity. t-SNE refers to stochastic nieghbour embedding and is appropriate for visualisation of high-dimensional datasets [8]. What occurs is unsupervised learning where each datapoint is given a location in 2D space, similarities between data points are converted to joint probabilities and the Kullback-Lieber divergence between joint probabilities of the low-dimensional embedding and the high-dimensional data are minimised.

Perplexity is a hyperparameter for t-SNE and it relates to the number of nearest neighbours allowed. The higher the perplexity, the greater the variance. Perplexity was first optimised by eye in Fig. 11 and then by silhouette score, which is a metric used to measure the quality of clustering results. The best-performing t-SNE according to silhouette score is shown in Fig. 12.

In comparison to the t-SNE representation of the input data to the embedding layer, the data from the embedding layer is much easier to separate into the requisite classes than the data from the input layer. Even with high perplexity, the t-SNE distribution representation of the input layer appears mostly random with little structure. This is expected as there is high dimensionality in the input data and related data points may not be grouped together. This is highly likely, as the dataset was generated randomly. In contrast, the embedding layer of the neural network has similar inputs already mapped closer together and is more structured. Essentially, the neural network has already done the work of grouping similar data points together.

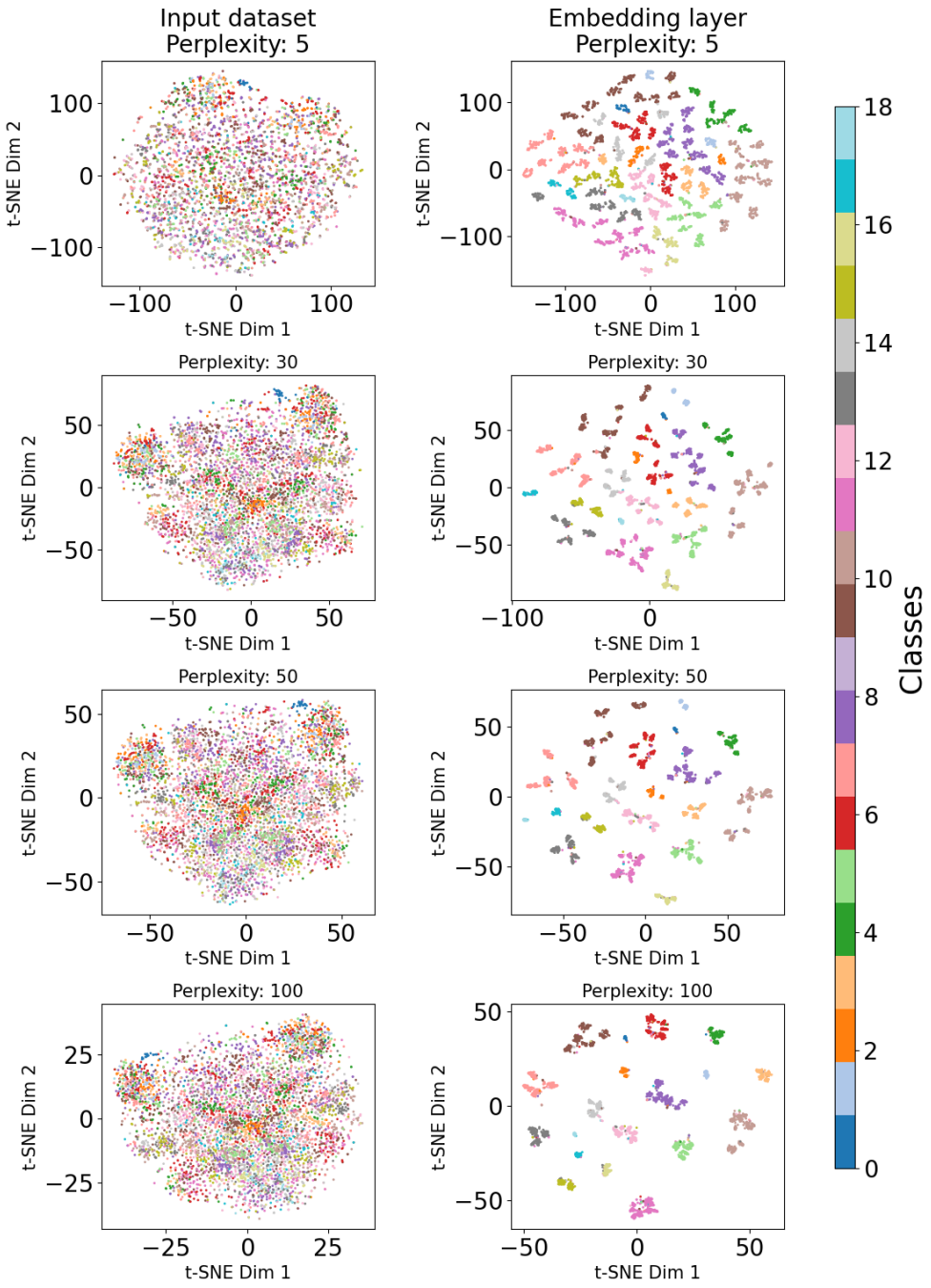

Figure 11: t-SNE distribution study as a function of complexity over the raw data and embedding layer of the best-performing neural network.

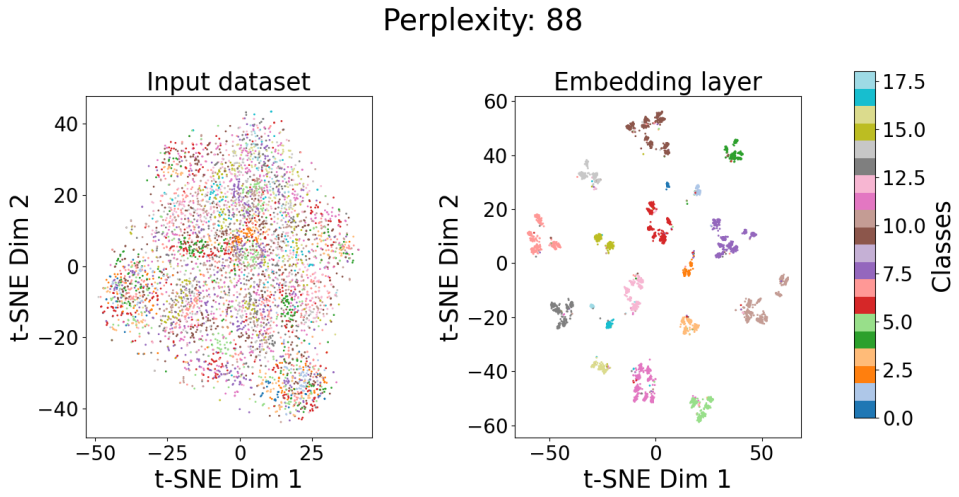

Figure 12: Best t-SNE distribution of the input data and embedding layer of the best-performing neural network, optimised over perplexity.

7 Summary

An inference pipeline for a neural network to recognise and add together MNIST digit pairs has been built with `Tensorflow`, hyperparameter tuned with `Optuna`, and trained to an accuracy of 95%. Additionally, other inference algorithms have been explored, including SVMs, random forest classifiers, and AdaBoost classifiers, with respective accuracies of 80%, 79%, and 14%.

Linear regression classifiers have also been trained on the dataset, varying the number of samples. It was found that the greatest increase in accuracy comes from when the number of samples is enough to encompass all digit pair permutations.

Additionally, the t-SNE distribution of the input data and embedding layer of the best-performing neural network was produced with an optimal perplexity of 88. It was shown that the embedding layer is inherently more structured than the input data.

References

[1] Deng L. The MNIST Database of Handwritten Digit Images for Machine Learning Research. IEEE Signal Processing Magazine. 2012;29(6):141-2.

[2] Zeiler MD, Ranzato M, Monga R, Mao MZ, Yang K, Le QV, et al.. On Rectified Linear Units for Speech Processing; 2013. Available from: `https://doi.org/10.1109/ICASSP.2013.6638312`.

[3] Kingma DP, Ba J. Adam: A Method for Stochastic Optimization; 2017. Available from: `https://arxiv.org/abs/1412.6980`.

[4] Mehta P, Bukov M, Wang CH, Day AGR, Richardson C, Fisher CK, et al. A high-bias, low-variance introduction to Machine Learning for physicists. Physics Reports. 2019 May;810:1–124. Available from: `http://dx.doi.org/10.1016/j.physrep.2019.03.001`.

[5] Chang CC, Lin CJ. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology. 2007 07;2.

[6] Zheng X, Jia J, Guo S, Jin-song C, Sun L, Xiong Y, et al. Full Parameter Time Complexity (FPTC): A Method to Evaluate the Running Time of Machine Learning Classifiers for Land Use/Land Cover Classification. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing. 2021 01;PP:1-1.

[7] Breiman L, Friedman JH, Olshen RA, Stone CJ. Classification And Regression Trees. Routledge; 2017.

[8] van der Maaten L, Hinton G. Visualizing Data using t-SNE. Journal of Machine Learning Research. 2008;9(86):2579-605. Available from: `http://jmlr.org/papers/v9/vandermaaten08a.html`.

A Use of auto-generation tools

Auto-generation tools were used to help setup `Tensorflow` on a WSL2 environment, parsing error messages throughout the project, and to help format this LaTeX report.

Auto-generation tools were also used for code prototyping in plotting the results for task 4. Auto-generation tools were not used for code generation elsewhere.