Optimising Neural Network Inference Pipelines for Digit Summation

Lewis McConkey
Cambridge University
Word count: 2119

December 19, 2024

Abstract

In this report we focus on building a wide range of neural network pipelines to predict the sum of two MNIST digits. We optimize the hyper-parameters using a grid search and end up with our best neural network producing a validation accuracy of 97.19%. We compare this performance to various other inference algorithms like support vector machines and random forests to determine that the neural networks give far better accuracy and perform better. Further, a weak linear classifier is explored, both on the combined dataset and sequentially applied to individual images, to analyze performance over small samples for which the resulting test accuracy is poor. We finish with the embedding layer of the best-performing neural network being visualized using t-SNE and compared to a direct t-SNE application on the input data.

1 Introduction

Throughout this report our aim is to build an inference pipeline which can successfully add two handwritten MNIST digits. Through hyperparameter tuning and model comparison we build our best performing model to complete this task. We test a range of different neural networks, inference algorithms and weak linear classifiers to perform this task. We look at importance of adding batch normalisation and dropout to our neural network and why other inference algorithms perform the way they do. We also show the difference in performance of a single and a sequential linear classifier trained on the dataset. Finally we look at directly applying t-SNE on the input dataset and compare this representation with the representation of the t-SNE distribution of the various classes in the embedding layer.

2 Generating Datasets

To construct the appropriate training, validation and test sets we start by combining pairs of MNIST digits by concatenating them horizontally to form images of size 56 x 28 and providing the appropriate labels with them. Before splitting our data, we shuffle it to avoid any biases and help prevent overfitting. We now split our dataset into 80% training data, 10% test data and 10% validation data. We use 80% in the training set to ensure that the model has enough examples to learn effectively and we use 10% in the validation set to guide model tuning but also not to generalise the model's performance. We use 10% in the test set to provide a statistically reliable measure of performance but also not weakening the model's learning by taking more percent out of the training set. This particular split balances the bias-variance tradeoff by allocating sufficient data for training to minimise bias while reserving validation and test sets to prevent overfitting, ensuring robust generalisation. When finding the best performing model on our data with hyperparameter tuning we can also experiment on a subset of the data and for a small amount of epochs (e.g. 5) by switching the percentages of our split.

3 Neural Networks

A simple way to perform hyper-parameter tuning is to use a grid search to perform many neural networks on the data rather than creating an individual model every time we want to experiment with a new model or manually changing the model every time. While we can do this we should also limit the time taken for the model to run and hence we use 5 epochs and only use 20,000 images in our data when tuning our hyper-parameters. Before hyperparameter tuning and when initially setting up the model we still have choices of what optimiser to use, what loss function to use and what activation function to use.

We choose ReLU for the activation function in the hidden layers due to it's effectiveness in image-related tasks like with MNIST as also seen in [1] where it is discussed that "ReLU function is the most efficient function by a narrow margin" and also states "One of the reasons ReLU was the most efficient is that it avoided the problem of vanishing gradients.". In terms of the optimisers we perform a small test where the winner is the Adam optimiser (RMSprop was not too far behind however stochastic gradient descent was awful in this case). Finally for the loss function we use sparse categorical cross-entropy because this is commonly used for classification tasks when dealing with multi-class problems.

Now for the hyper-parameter tuning we compare between 300,400 and 500 neurons in each layer and compare between 2,3,4,5 and 6 layers. We also compare through learning rates of 0.01,0.001 and 0.0001 and compare between batch sizes of 60,80,100,120 and 140. This produces a wide range of possible models from our grid search. A lot of models were very close in accuracy so we take the five highest performing neural networks which are shown in this table:

N	L	LR	BS	Accuracy
400	4	0.001	60	83.75 %
400	5	0.001	80	83.75 %
500	5	0.001	60	83.40 %
500	5	0.001	80	82.90 %
500	4	0.001	100	82.85 %

Where N is the number of neuron's in our hidden layers, L is the number of hidden layers, LR is the learning rate, BS is the size of the batches we use and accuracy is the validation accuracy of the model. The first thing we notice is that they all have 0.001 learning rate and so this is the clear winner from the range of learning rates. It looks like 400 or 500 neuron's might be best. There is no clear winner in the number of layers (so we will test 4 and 5 again) and we will take 60, 80 and 100 as the batch sizes. We now perform a second grid search to be completely sure. This time we compare between 4 and 5 number of layers, 400 and 500 neuron's, 60, 80 and 100 batch size and keep the learning rate fixed at 0.001. After performing the second grid search we see that 500 neuron's, 4 layers, 0.001 learning rate and 80 batch size is our optimal model with 84.5% accuracy.

Now that we have our best performing neural network, the next question is can we do better? We see that when we now use 20 epochs and 100,000 pairs

of MNIST images we get an accuracy of 94.78%. We look first at using dropout to improve on this and this produces an accuracy of 96.84% which is an big improvement. Next we add batch normalisation to our model and we get an accuracy of 97.19%. We could also use other methods such as early stopping or adding L1/L2 regularisation however early stopping did not affect the model too much in this case and adding L1/L2 regularisation severely damaged the performance so we don't use it. Our overall model consists of 500 neurons, 4 layers, 0.001 learning rate and 80 batch size and uses batch normalisation and dropout to get an accuracy of 97.19%. We give the weights of our best performing model in the code and also in the file best_model_weights.h5. We can visualise the performance of our last three neural networks (with and without the batch normalisation and dropout) by plotting their training and validation accuracies together:

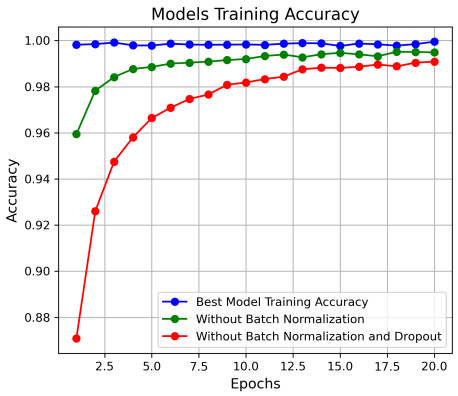

Figure 1: Best three models training accuracy

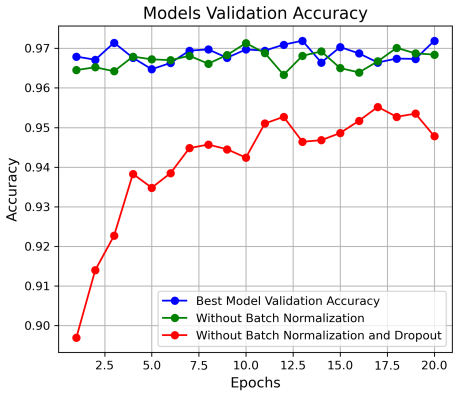

Figure 2: Best three models validation accuracy

From this we see the model without batch normalisation and dropout convergences a lot slower compared to the one with dropout and our best model. Batch normalisation stabilises training by normalising activations, allowing for faster convergence and higher learning rates and hence why adding this in makes our models converge to a better accuracy faster. Adding dropout improves generalisation and prevents overfitting which helps the model train better. We do also see consistently the model with batch normalisation and dropout performs better for most of the epochs although the model without batch normalisation actually has better validation accuracy in some epochs.

4 Other Inference Algorithms

We now implement other algorithms to perform the same task as we did with neural networks and we compare the results with our best performing neural network. We test out the performance of support vector machines (SVMs), random forests, K-nearest neighbors (KNNs), gradient boosting and hierarchical clustering again using a grid search to cover all these algorithms efficiently. Here are the results that we obtain when we train on 100,000 pairs of MNIST digits:

Algorithm	Accuracy
Best Neural Network	95.93 %
Support Vector Machines	79.16 %
Random Forest	77.73 %
K-nearest Neighbors	67.20 %
Gradient Boosting	48.36 %
Hierarchical Clustering	N/A

From these results we deduce that the neural networks way outperform the other models which is because they excel at learning complex, non-linear patterns in high-dimensional data like the MNIST dataset we have. We also used far more hyper parameters and methods to increase our neural network performance compared to the other algorithms. SVMs performed best out of the other algorithms showing its strong ability to separate data in a high-dimensional space but also took the most time to run from the algorithms that ran. Random forest performed in the middle range compared to the other algorithms suggesting they may not be able to handle the raw image data that we provided to them as well as SVMs and neural networks. KNNs performed worse than random forest which highlights its limitations with dealing with high-dimensional image data (gradient boosting was also the worst due to this reason) especially the "curse of dimensionality". Finally hierarchical clustering failed to run in a sufficient time (It did not run over night) which is due to its computational complexity, which scales poorly with the number of samples. We can also show this performance comparison (of the models that ran) in a plot:

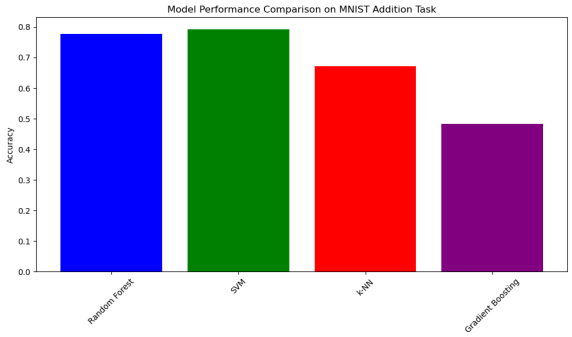

Figure 3: Plot showing model performance comparision

5 Linear Classifiers

We train a single linear classifier on 100,000 digit pairs from the 56 x 28 MNIST dataset and also train a single linear classifier applied to the two images sequentially. We train both models with varying number of samples (50,100,500,1000) and use this to view the performance of the test set. Here are the results:

Sample	Single	Sequential
50	11.12 %	9.74 %
100	12.52 %	10.14 %
500	13.26 %	10.62 %
1000	14.09 %	10.91 %

The table shows the comparison between our single linear classifier and our sequential linear classifier through their test accuracy at each sample size. We only use small samples but the performance is really poor for both especially compared to our neural network performances. The single classifier appears to perform better than the sequential classifiers from initial results but we can also plot both across a range of samples and view their accuracy across the range.

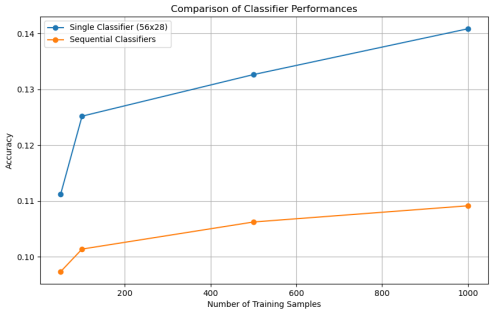

Figure 4: Plot showing number of training samples vs accuracy for both single and sequential linear classifier

The plot shows a range of training samples for both the classifiers we are comparing and the resulting test accuracy. We can see that for all sample sizes that the single linear classifier performs better, this could be because it operates on the full input data whereas the sequential classifier is split into two halves. Sequential models are also computationally faster but at the cost of the accuracy so when accuracy is the priority then the single linear classifier is preferred.

6 t-SNE

To better understand feature representations for the MNIST task we are wanting to complete we applied t-SNE on the input dataset and compared it to the t-SNE distribution of the various classes in the embedding layer to compare the two. This comparison allowed us to assess how well the neural network's learned representations capture the underlying structure of the data compared to the raw input. We can visualise both of these plots here:

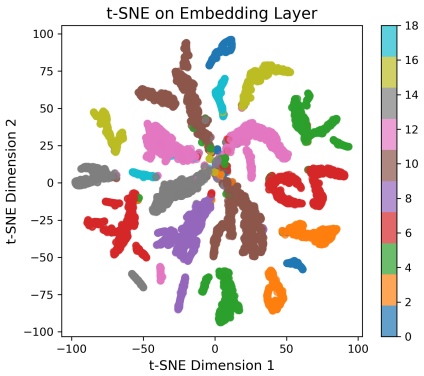

Figure 5: t-SNE on the embedding layer with perplexity 30

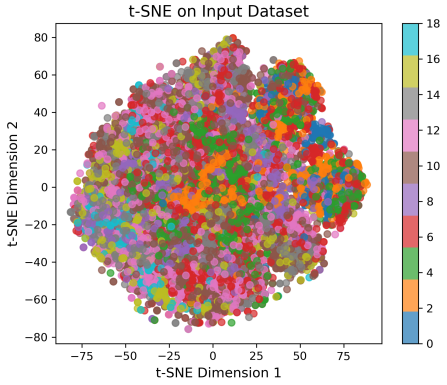

Figure 6: t-SNE on the input dataset with perplexity 30

Here we see a clear improvement in class separation when using the embedding layer. The clusters of the input dataset are poorly separated and overlapping significantly likely due to the fact that raw pixel values do not provide meaningful abstractions of the data, making it difficult for t-SNE to identify the underlying structure. The t-SNE visualisation of the embedding layer shows more distinct and well separated clusters showing the ability of the neural network to learn high-level features that align closely with the task objective.

We also experimented with a range of perplexities to improve the local and global structure and relationships in the data. With small ranges of perplexities (for the embedding layer) we saw the local neighbourhoods dominating and clusters merged together to form a shape similar to figure 6. With larger ranges of perplexities (for the embedding layer) global structure dominated and clusters separated significantly. For the input dataset we did not observe much difference in the shape when changing the perplexity. We chose 30 as the perplexity to balance these two aspects. We can see overall the superiority of learned embeddings over raw input features in representing the data from our t-SNE study.

References

[1] Shin-Hao Wang, Erik Sakk (2023), The effect of activation function choice on the performance of convolutional neural networks, Journal of emerging investigatiors