ML Assignment One

ARTIFICIAL AND CONVOLUTIONAL NEURAL NETWORKS AND THEIR EFFECTIVENESS ON THE **MNIST HANDWRITTEN DIGITS** DATASET

COMPLETED BY

DAVID PULLINGER

The University of Cape Town

CSC3022F

3 May 2022

TABLE OF CONTENTS

Introduction

The purpose of this report is to analyse the effectiveness of Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) in classifying digits from the **MNIST Handwritten Digits** dataset. In doing so, I will explain the reasoning behind the choices of various hyperparameters of the learning models, namely network topologies, loss functions, optimizers, learning rates, batch sizes and number of training epochs.

Input and Pre-processing

Input

The **MNIST Handwritten Digits** dataset that was used to train and test the networks consisted (conveniently) of a training set and a testing set. The training set and testing set consisted of 60000 and 10000 data items, respectively. Each data item was a 28-by-28 pixels grayscale image, with each pixel value ranging between 0 and 255.

Pre-processing

Each data item was already centred and the same size (28-by-28 pixels), so little pre-processing was needed on the dataset. When the dataset was imported, I used a transform that was built into PyTorch called ToTensor. This transform converted the binary image into a Tensor with size [1,28,28], and scaled the pixel values (which ranged between 0 and 255) to a range between 0 and 1.

After this, I applied another pre-processing step called Normalization. Normalizing scales each pixel value of the image so that they all belong to the same distribution, namely the Gaussian distribution with mean 0.1307 (see footnote[1]) and standard deviation 0.3081 (see footnote[2]). Normalization tends to make the loss function more symmetrical, which is easier to optimize because the gradients tend to point towards the global minimum and we can take larger steps while training the model.

Finally, I attempted to apply one last pre-processing step, which was to filter the pixel values which were below a certain threshold. The effect of this can be seen in Figures 1 2 below:

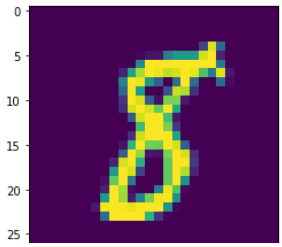

Figure 1: An image before pre-processing

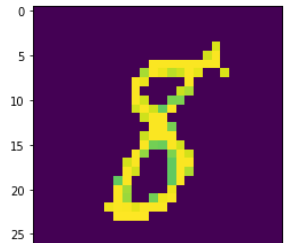

Figure 2: An image after pre-processing

[1] 0.1307 is the mean of pixel values across the MNIST dataset
[2] 0.3081 is the standard deviation of pixel values across the MNIST dataset

The aim of this step was to make the images clearer and remove noise, which would reduce the training time of the network. Since there were so many diverse data items, I hoped that removing this noise would not make the model less robust to noise, as it would already be trained with data that had numerous rotations, positions, etc. Furthermore, this pre-processing step could also be done to user-supplied input to the model.
I soon realized, however, that applying this "threshold filtering" to the data was quite a labour-intensive task. Therefore, I decided to only apply this pre-processing step to user-supplied test data, if the user indicated that they wanted it, for accuracy comparison.

Artificial Neural Networks

Introduction

I began my analysis by creating a few ANNs, and later, to achieve a 99+% accuracy, a few CNNs. Since the 28-by-28 pixel images were flattened before being fed to the networks, the ANNs consisted of an input layer with 784 (28*28) neurons. They also consisted of various hidden layers, and an output layer with 10 neurons, to classify the ten classes (digits from 0 – 9). My analysis, therefore, mainly consisted of tweaking the hyperparameters in the hidden layers, and seeing what their effects were.

Baseline Model

As a baseline to test my ANNs against, I created a linear regression model, which directly mapped the 784 input neurons to 10 output neurons. Since this was the simplest learning model that you can create, I believed it would be appropriate to expect my ANNs to perform the same, if not much better, than this model.

Topology

When discussing topology of the model throughout this report, I will be referring to the topology of the hidden layers in the ANN. That is to say, I will be referring to the following three properties of the hidden layers:

1. Number of hidden layers
2. Number of nodes in the hidden layers
3. The activation functions used between the layers

Since the baseline model was a simple linear regression model, there were no hidden layers, and so no topology had to be considered.

Loss Function

Since the task at hand was multi-class classification, the obvious loss function to use was Cross Entropy. The only alternative was the Negative Log Loss (NLL) function, which is applied to data that has been activated using the LogSoftMax[3] function. In PyTorch, the CrossEntropy function applies this activation function for us.

[3] SoftMax is a function that converts our raw output (size 10 tensor) into an output (again, size 10 tensor) that sums to 1. In doing this, it essentially converts our raw output tensor into a tensor of probabilities. LogSoftMax is the same activation function as SoftMax, but with the log function applied as a final step.

The reason we do not use Mean Squared Error (MSE) as a loss function is because MSE is used for simple **regression** models, while Cross Entropy is used for **classification** models. This is because, when deriving MSE loss function, one assumes that the loss comes from a Normal Distribution. Conversely, one assumes it comes from a Binomial Distribution when deriving the Cross Entropy loss function. The latter makes more sense for classification since the output of the model is how confident it is whether the input belongs to a specific class or not.

Optimizer

Optimizers are functions that adjust weights in the network, based on the loss the network receives based on its output. While Stochastic Gradient Descent (SGD) is a popular optimization algorithm, Adam is becoming more popular as the "go to" optimizer.
A major disadvantage of SGD alone is that it uses a fixed learning rate, which may cause the model to not find a maximally optimal solution (the SGD will find a local minimum in the loss function, instead of a global minimum). More advanced optimizers, on the other hand, are able to dynamically change the learning rate. For instance, they may start with a high learning rate in the beginning of training, and decay (decrease) the learning rate as training progresses. This allows the model to hopefully find a global minimum during the beginning of training (because of a high learning rate), and never "jump" out of the global minimum during the end of training (because of a low learning rate).

Adam is once such optimizer, and it has become the industry standard when training deep neural networks. It not only is able to dynamically decay the learning rate, but it is also able to keep track of a learning rate for every single connection (weight) in the network. Adam was therefore my choice of optimizer for the Baseline Model, as well as all future models.

Batch Size

When training a model with a lot of input data available, it is best practice to feed the data to the network in batches, usually of size greater than 32, to improve performance during training. However, using a batch size that is too large will cause the model to overfit and not generalize well. For the baseline model, I arbitrarily used a batch size of 32, but experimented with this factor in future models.

Number of Epochs

I trained the baseline model and all subsequent models using early stopping (discussed more under Generalization) to avoid over-fitting. Thus, I would train the model until it converged[4], or until 30 epochs had been reached.

Training and Validation

The baseline model was trained using a training set of 48 000 data items, and validated using a validation set of 12 000 data items. After each epoch, the test accuracy and validation accuracy was recorded, and used in early stopping (discussed more under Generalization).

[4] Convergence in a neural network can be defined as the point when additional training will not improve the model.

Performance

The performance of all the ANNs was measured via their accuracy on the validation set. Each ANN was trained 3 times, and the average of the validation accuracies was taken as a final measure of performance.

The performance recordings of the baseline model are given below:

Iteration	Convergence Epoch	Validation Accuracy	Test Accuracy
1	9	91.87%	92.62%
2	8	92.18%	92.44%
3	8	91.53%	92.59%
Average	8.33	91.86%	92.55%

Table 1: Performance of baseline model

The baseline model thus had a validation accuracy of 91.86%, and this was the validation accuracy the ANNs had to beat.

Similarities

It is important to note that, for the reasons outlined in my discussion of the baseline model, the ANNs that were designed thereafter shared the following hyperparameters:

1. Loss function (CrossEntropyLoss)
2. Optimizer (Adam)
3. Learning rate (start with lr = 0.001, and Adam optimizes this hyperparameter)
4. Number of epochs (30 epochs, or less if the model converged)

For this reason, they need not be discussed hereafter.

Classifier_ANN3_N16/64/256

Topology

My initial ANN was one with 3 hidden layers, each consisting of 16 nodes, and varying batch sizes. The activation function used between each layer was the ReLU function. A non-linear activation function needs to be implemented to achieve a non-linear transformation of the data, thus allowing us to classify complex classes which need to be separated by hyperplanes. ReLU is one such function, which has the added benefit of being:

1. Simple
2. Fast to compute
3. Easily differentiable
4. Less prone to vanishing gradient[5]

[5] The vanishing gradient problem refers to the phenomenon where, as loss is backpropagated through a network, and each layer uses the gradient of the loss function, the gradient becomes tiny, so weights in the first few layers of a network are hardly updated.

The justification for the hidden layers was not so straightforward. I thought that 3 layers was a good starting point, and I would then decrease or increase the number of layers, and track the performance of the changes.

Similarly, the choice of 16 neurons was arbitrary. I thought that 16 was a decent starting point, and I would increase this to 64 and then, finally, to 256 neurons per layer and track the performance of the changes.

Batch Sizes and Performance

In total, I tested 9 ANNs with 3 layers: 3 with 16 neurons/layer, 3 with 64 neurons/layer and 3 with 256 neurons/layer. For each class of ANN (the classes being 16, 64, and 256 neurons/layer), I tested the ANN with various batch sizes, namely 64, 128 and 256. The results of these tests (average best accuracy on the validation set after 3 iterations) are given below:

		Neurons per Layer		
		16	64	256
Batch Sizes	64	95.04%	97.12%	97.82%
	128	95.23%	96.62%	97.77%
	256	95.12%	96.83%	97.78%

Table 2: Effects of neurons per layer and batch sizes on the average best accuracy on the validation set

As can be seen, the model performed much better as more neurons were added. Also, the effect of the batch size was variable depending on the topology of the network. However, while testing I noticed that the batch size of 128 converged very quickly for the models with 64 and 256 neurons per layer (6 and 9 epochs, respectively). This is likely because the model was able to overfit the data more quickly as the batch size increased (since it was reducing loss on a larger portion of the training data). This was confirmed when the training set accuracy was 99.4%!

The likely reason that the model performed better when more neurons were added was that the network became more complex, allowing it to better classify the data.

Classifier_ANN6_N64/256

Topology

These ANNs had 6 hidden layers, with either 64 neurons/layer or 256 neurons/layer. I opted to not test the case with 16 neurons/layer as it was clear from the 3-layer ANNs that the performance was well below its counter parts. My hope was that these ANNs would perform

marginally better than the ANNs with 3 hidden layers as they would have more neurons and thus be more complex. On the other hand, these ANNs would most likely suffer from the vanishing gradient problem more than the 3-layer ANNs.

Again, the activation function used between each layer was the ReLU activation function. This was for the reasons given above.

Batch Sizes and Performance

The results of these tests (average best accuracy on the validation set after 3 iterations) are given below:

		Neurons per Layer	
		64	256
Batch Sizes	64	96.66%	97.46%
	128	96.76%	97.80%
	256	96.67%	97.57%

Table 3: Effects of neurons per layer and batch sizes on the average best accuracy on the validation set

Clearly, the model did not improve on the performance of the 3-layer model with 256 neurons per layer. As abovementioned, this was partly due to the vanishing gradient problem. Furthermore, since the network was more complex, it was able to memorize the training data better, and it began to overfit. This caused a slight decrease in validation accuracy between the models with 3 layers and 6 layers.

This lead me to believe that increasing the amount of layers was not going to improve the model, so I decided to instead decrease the amount of layers slightly, and investigate what would happen.

Classifier_ANN2_N64/256

Topology

These ANNs had only 2 hidden layers, with either 64 neurons/layer or 256 neurons/layer. I again opted to not test the case with 16 neurons/layer as it was clear from the 3-layer ANNs that the performance was well below its counter parts. I did not expect these ANNs to perform as well as the ANNs with 3 hidden layers as they did not have as many neurons and were thus less complex, and were unable to make complex classifications.

Again, the activation function used between each layer was the ReLU activation function. This was for the reasons given above.

Batch Sizes and Performance

The results of these tests (average best accuracy on the validation set after 3 iterations) are given below:

		Neurons per Layer	
		64	256
Batch Sizes	64	97.26%	97.57%
	128	97.00%	97.58%
	256	96.80%	97.63%

Table 4: Effects of neurons per layer and batch sizes on the average best accuracy on the validation set

As can be seen, the performance of the 2-layer ANN was not far off that of the 3-layer ANN (the best cases differed by only 0.19%). In fact, the worst case of the 2-layer ANN (96.80%) was better than the worst case of the 3-layer ANN in the 64 neuron per layer class (96.62%). This is likely because the 2-layer ANN's non-complexity doesn't allow it to overfit the training data as much as the 3-layer ANN, allowing it to perform better on the validation set.

Similarly to before, the model performed better when there were 256 neurons per layer as it was able to make more complex classifications. However, the effect of the batch size was not clear, as before.

I concluded that an ANN with 3 hidden layers was the ideal depth. Hereafter, I decided to tweak the amount of neurons in each layer in the 3 hidden layers.

Classifier_ANN3_DESC

Topology

This ANNs had 3 hidden layers, with 256 neurons in the first layer, 128 neurons in the second layer, and 64 neurons in the third layer. I was not sure what to expect for this model's performance. I assumed that it would be similar to that of the 3-layer ANN with 256 neurons in all layers. It was fair to assume this because the total number of neurons was high, so the complexity of the two models was similar. It is very difficult to understand exactly what an ANN is doing when it adjusts its weights, but I could only assume that this topology would allow the network to extract features from the image, from larger features such as circles, to smaller features such as arcs and dashes.

Again, the activation function used between each layer was the ReLU activation function. This was for the reasons given above.

Performance

Since the batch size seemed to have a minimal effect on the performance of the model in the testing above, I decided to arbitrarily use a batch size of 128. The results of this model are given below:

Iteration	Validation Accuracy
1	97.60%
2	97.78%
3	97.68%
Average	97.69%

As can be seen, this "descending" topology was not as effective as 3 layers of 256 neurons each. This was most likely because, contrary to what I initially thought, the difference in the number of neurons, however small, may make a large difference in the performance of the models.

Classifier_ANN3_ASC

Topology

This ANNs had 3 hidden layers, with 64 neurons in the first layer, 128 neurons in the second layer, and 256 neurons in the third layer. Again, it is very difficult to understand exactly what an ANN is doing when it adjusts its weights, but I could only assume that this topology would allow the network to extract features from the image, this time from smaller features such as arcs, to larger features such as circles.

Again, the activation function used between each layer was the ReLU activation function. This was for the reasons given above.

Performance

Since the batch size seemed to have a minimal effect on the performance of the model in the testing above, I decided to arbitrarily use a batch size of 128. The results of this model are given below:

Iteration	Validation Accuracy
1	97.30%
2	96.93%
3	97.18%
Average	97.14%

This "ascending" topology was not much better than the "descending" topology, or the 3-layer ANN with 256 neurons per layer. This was again because the number of neurons in the network were less, leading to a less complex model, which was not able to make complex classifications.

Conclusion

It is clear from these results that, until the point where overfitting happens, having more neurons in your network is better. The network simply becomes a lot more complex as its number of neurons increases, and this allows it to make more complex classifications. I can therefore propose that, from my results, the best ANN to classify the **MNIST Handwirtten Digits** dataset has the following hyperparameters:

1. Topology: 3 layers with 256 neurons each, with the ReLU activation function used between each layer.
2. Loss function: CrossEntropyLoss
3. Optimizer: Adam
4. Batch Size: 128
5. Number of epochs: Train until convergence happens, which is usually around 20 epochs

I should note that I should have experimented with networks with 512 or even 1024 neurons per layer, as these gave me a validation accuracy of greater than 98% when I first started using PyTorch. However, I had to experiment with CNNs to achieve an accuracy greater than 99%.

Convolutional Neural Networks

Classifier_CNN3

Topology

I began my investigation with CNNs by creating one with 3 convolutional layers, and no fully connected layers. The first convolutional layer had 32 features, the second had 48, and the last had 64. After each convolution layer, there was no batch normalization[6] layer and no max pooling layer[7]. I wanted to see what the performance of a bare bones CNN would be, and I wanted it to serve as a baseline models for all CNNs that would come after.

The activation function used between each convolutional layer was the ReLU activation function. This was for the reasons given above.

Performance

I knew that the CNNs would have a better performance than the ANNs, because they are built to analyse data that can be in higher than one dimension. This is because the convolutional layers build these feature maps, and applying activation functions to them, before they are ever flattened.

The results of this CNN are given below:

Convergence Epoch		13	
Training Loss	0.008	Training Accuracy	99.75%
Validation Loss	0.075	Validation Accuracy:	98.65%

[6] Batch normalization is the same normalization method that was applied during preprocessing, except it is applied in between the layers of a neural network, instead of on raw data.

[7] Max pooling is a pooling operation which slides a kernel over the output of a convolution (called a feature map) and returns the biggest element in the kernel. This ensures that only the most prominent features in the feature map are returned, and it reduces computational load as the feature map becomes smaller.

As one can see, the performance of the model exceeded that of the ANNs with the same number of layers.

Classifier CNN3_NORM

Topology

The topology of this network was the same as the previous one, except it had a Batch Normalization layer after each convolutional layer. I hoped that this would improve training times as the weights would not be able to explode.

Performance

The results of this CNN are given below:

Convergence Epoch		6	
Training Loss	0.007	Training Accuracy	99.81%
Validation Loss	0.070	Validation Accuracy:	98.72%

Clearly, batch normalization did reduce training times, although it had a minimal effect on the accuracy of our model.

Classifier CNN3_NORM_POOL

Topology

The topology of this network was the same as the previous one, except it had a Max Pooling layer after each convolutional layer and before each Batch Normalization layer. I hoped that this would improve training times as the weights would not be able to explode, and the feature maps would be smaller and more effective (as only prominent features are kept after max pooling).

Performance

The results of this CNN are given below:

Convergence Epoch		11	
Training Loss	0.009	Training Accuracy	100.0%
Validation Loss	0.022	Validation Accuracy:	99.54%

Clearly, this CNN performed exceptionally, allowing us to achieve an accuracy of greater than 99%! Again, this is probably because the most important features of the image are extracted. However, the fact that the training accuracy was 100% implied that the CNN had overfit the data. I therefore added a dropout layer (discussed under Regularisation), in the hope that this would reduce overfitting.

Regularisation

Regularisation is a set of techniques that can be used to reduce overfitting of a neural network, allowing it to generalize better, and thus get a higher accuracy when classifying unseen data items. Two techniques that I used in the neural networks were early stopping and dropout.

Early Stopping

This technique has already been mentioned but to reiterate, early stopping is a technique whereby the model stops training after further training does not yield an improvement on a validation set (a subset of the training set used for validation such as this). I implemented this by keeping track of the best validation accuracy encountered thus far, and checking whether we have trained for a number of epochs, say x, without beating this accuracy. While this variable x can be tweaked, I opted for a value of 5 for computational reasons.

This technique was effective as every time I trained the network, it converged before the original number of epochs I wanted it to train for (30 epochs).

Dropout

Dropout is a regularisation technique whereby a fraction of the weights in the network are disabled, prohibiting co-dependence of neurons, which in turn would have led to overfitting. This technique also gives us the abstraction of training several different network topologies in parallel, as each time weights are disabled, a new network topology is formed.

I only applied this technique to CNN3_NORM_POOL, to see if I could reduce overfitting and perhaps increase validation accuracy, and while I saw an increase in training time, the accuracy on the validation set did not improve.