ECE566 MACHINE LEARNING AND DEEP LEARNING

INTRODUCTION

The main objective of this project is to design, train and test a Convolutional Neural Network (CNN) for a multi-class classification problem. Convnets work very well for image recognition, one of the reasons is that in a convolutional layer, neurons are not connected to every input pixel, but only to pixels in a certain field. That way, the network can concentrate in low level features on the first layers and then assemble them into higher-level features.

In this problem, the MNIST handwritten digit database with additive correlated noise will be used. This time, instead of classifying combinations of two classes, the problem will be to classify among ten classes (digits from 0 to 9).

This database consists of 60000 training images and 10000 testing images of handwritten digits from 0 to 9. The size of this images is 28x28, therefore, totally in one image there are 784 pixels.

In the next section, a first approach to this problem is described.

Convolutional Neural Network

1. Data Analysis

First, after loading the MNIST dataset to our repertory and separating the training and testing data, we will have a look to how the noise affects to the images.

In Figure 1, an example of a handwritten one with noise is shown:

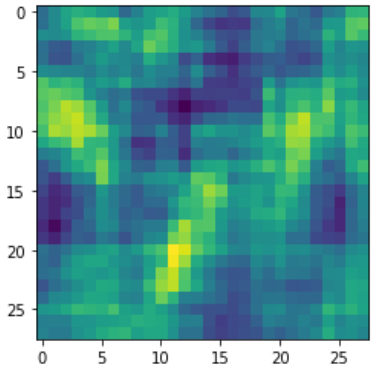

Figure 1.Example of a handwritten noisy digit.

Comparing to the real MNIST database, now the digits are more difficult to recognize even by human vision.

Before starting the model creation, training data will be normalized, and the label attributes will be one-hot encoded. That way, algorithms will not assume that two nearby values (7 and 8 e.g.) are more similar than two classes that are distant.

2. First Approach. Classifying Training Data. Model 1

First, we will create a simple model to measure how this dataset can be classified. From Keras, a Sequential model has been imported. This first model is composed by a 2D convolutional layer, followed by a flatten layer to transform matrix input into vector output and finally a dense layer that allows having ten outputs, one for each class

The next table shows the structure of the CNN:

Layer	Output shape	Parameter #
Conv2D	26x26	10
Flatten	676	0
Dense	10	6770

Table 1. Structure of Model 1

For this first approximation, 100 epochs will be defined and for validation, we will use a batch size of 100 and a 30% of validation samples. On Figure 2, model accuracy is plotted for each epoch:

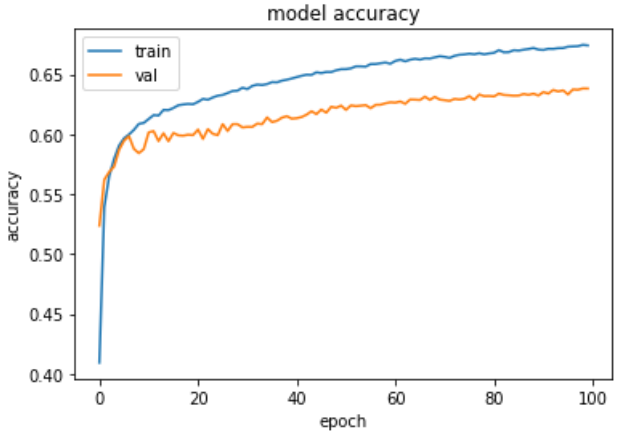

Figure 2 Accuracy vs epoch in model 1

After 100 epochs while training accuracy is around 0.67, validation is around 0.63 Accuracy in test set is 0.634 which is similar to the validation accuracy.

Regarding to the generalization error, one can see that validation accuracy is lower than training accuracy. One solution for this could be to reduce the validation split percentage. That way, we could use more training data and reduce overfitting.

3. Model Structure Optimization. LeNet-5

After implementing a relatively simple model, now we are going to create a more complex CNN adding some hidden layers. For that purpose, we are taking the famous CNN LeNet-5 which has been used for handwritten digit and character recognition.

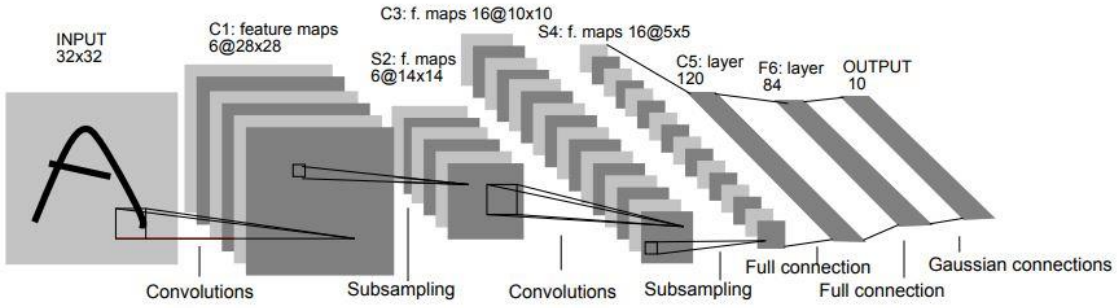

Figure 3 LeNet-5 Architecture. Credit: LeCun et al., 1998

The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier.

The input is a 32×32 image which passes through the first convolutional layer with 6 feature maps or filters of size 5×5 and a stride of one. The image dimensions changes from 32x32x1 to 28x28x6 because zero-padding is applied to the input image to take all the image with the defined stride. Then the LeNet-5 applies average pooling layer with a filter size of 2×2 and a stride of two. The resulting image dimensions will be 14x14x6.

Next, there is a second convolutional layer with 16 feature maps having size 5×5 and a stride of 1. The fourth layer is again an average pooling layer with filter size 2×2 and a stride of 2 and the output will be reduced to 5x5x16.

The fifth layer is a fully connected convolutional layer with 120 feature maps each of size 1×1. The sixth layer is a fully connected layer with 84 units.

Finally, there is a fully connected softmax output layer ŷ with 10 possible values corresponding to the digits from 0 to 9.

These are the results obtained with a batch size of 500, a validation percentage of 10% and 100 epochs:

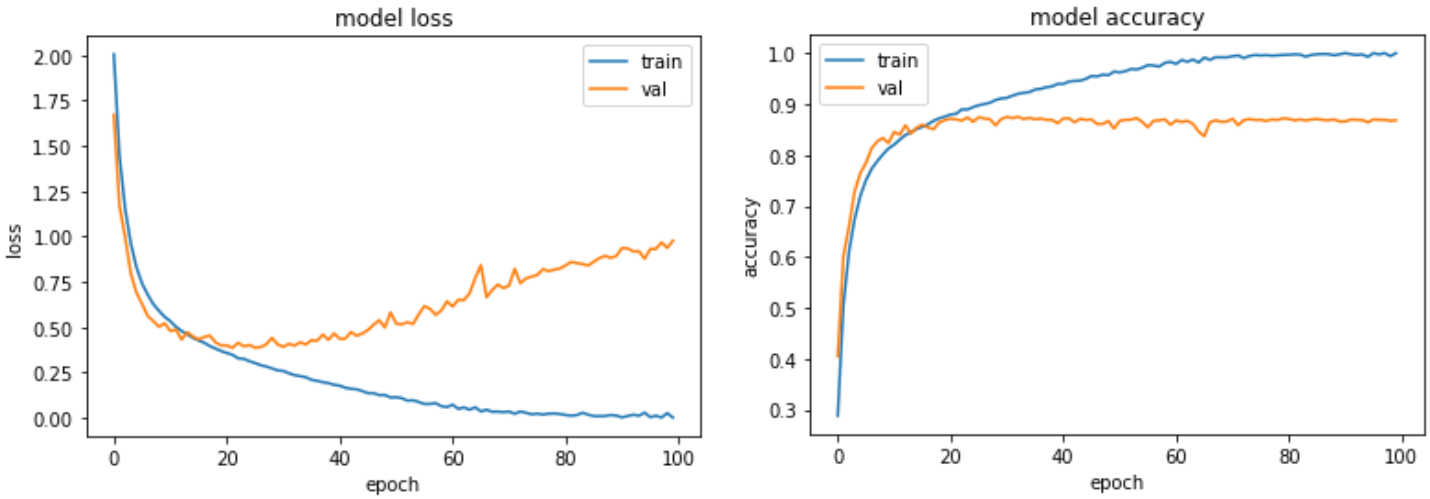

Figure 4. Second model loss and accuracy

One can see that after the 20th epoch model validation loss increases after having a minimum, while training loss continues reducing. This is due to model overfitting, it predicts well the training set but gets a worse result with the validation set. After 100 epochs the model can predict 100% of the training data correctly but the validation accuracy is of 0.87 which is a lower result.

Therefore, we should change the parameters to avoid the generalization error

4. Model Hyper-Parameter Optimization

In this section, hyper-parameters of our model will be optimized. Batch size and number of training epochs are the parameters that we are taking into account in this section.

In order to optimize those parameters, we will use the grid search cross validation method from Sklearn's Model_selection class. With that, an exhaustive search over specified parameter values for an estimator is done.

```
from sklearn.model_selection import GridSearchCV
```

We define a dictionary that contains the grid search parameters, for the first search, batch size values are [200, 500, 1000]. On the other side, we have seen in the previous section that from approximately 15 epochs validation accuracy is not being increased. Therefore, we will test among values [10, 20] for number of epochs.

For this search, a 3-fold cross validation splitting strategy will be used. In total 6 possible combinations can be analyzed, and their scores are defined in the next table:

Batch size	Epochs	Mean score	Score std
200	10	0.8206	0.007713
200	20	0.8227	0.00707
500	10	0.7792	0.007363
500	20	0.813	0.0095
1000	10	0.6851	0.01471
1000	20	0.7812	0.01301

Table 2. Scores of the first grid search

Some conclusions can be drawn from those results. The best Accuracy is 0.8227 using {'batch_size': 200, 'epochs': 20}, however, there is not a significant difference between scores of 10 and 20 epochs with that batch size. Smaller batch sizes between these values have better scores and with a higher batch size, higher is the difference between results of 10 and 20 epochs.

Next, we are going to analyze the score for lower batch sizes (50, 100 and 200), so these are the results for the second grid search:

Batch size	Epochs	Mean score	Score std
10	10	0.8546	0.001006
50	10	0.8411	0.001167
100	10	0.8394	0.009166
200	10	0.8202	0.003092

Table 3 Scores of the second search

Best accuracy is 0.8411 using {'batch_size': 50, 'epochs': 10}, so, we can conclude that lower batch sizes can give better results, and this is what in [2] is stated:
“It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize”

However, small batch sizes need more time to train. Since in one epoch, all the training data is passed forward and backward, when the batch size is lower, more iterations must be done. So, for the same number of epochs, reducing so much the batch size can be very time-consuming.

If we measure the tradeoff between small batch size and few epochs VS big batch size and many epochs, we must look at Table 2. A batch size of 200 and 10 epochs have an accuracy of 0.8206, which is quite better than 0.813, the accuracy of the batch size of 500 and 20 epochs.

5. Training Optimization Algorithm Tuning

Keras offers many different optimization algorithms. In this section, we will tune the optimization algorithm used to train the network, each with default parameters. We will

test the following algorithms: 'SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam'.

First, a fast search will be performed with a batch size of 500 and 10 epochs in order to have an idea of which algorithms work better. Then, we will perform another search between the best algorithms and with a lower batch size which will take longer.

These are the results of the first search:

Algorithm	Batch size	Epochs	Mean score	Score std
SGD	500	10	0.2978	0.1135
RMSprop	500	10	0.6985	0.009637
Adagrad	500	10	0.6997	0.02661
Adadelta	500	10	0.7668	0.01401
Adam	500	10	0.6963	0.001657
Adamax	500	10	0.6471	0.03239
Nadam	500	10	0.8036	0.006422

Table 4 Scores of different optimization algorithms on the first search

The best Accuracy is 0.8036 using {'optimizer': 'Nadam'}. Second, we have 'Adadelta' algorithm and then we have three algorithms with similar scores: RMSprop, Adagrad and Adam. For the next search, we will narrow the searching set and we will search only for the algorithms mentioned in this paragraph. This is what we get with a batch size of 10 and 5 epochs:

Algorithm	Batch size	Epochs	Mean score	Score std
RMSprop	10	5	0.834	0.007
Adagrad	10	5	0.7343	0.02383
Adadelta	10	5	0.8497	0.005248
Adam	10	5	0.8375	0.006142
Nadam	10	5	0.8514	0.003616

Table 5 Scores of different optimization algorithms on the second search

Table 5 shows that Nadam algorithm (Nesterov-accelerated Adaptive Moment Estimation) is again the best choice. This algorithm is a combination of NAG and Adam and is usually employed for noisy gradients. Therefore, this will be the chosen optimizer for our model.

6. Network Weight Initialization Tuning

In this section, we will look at tuning the selection of network weight initialization by evaluating all the available techniques that Keras provides.

We will use the same weight initialization method on each layer. However, using different weight initialization schemes according to the activation function used on each layer would be better. We will let this for future work.

Weight initialization	Batch size	Epochs	Mean score	Score std
Uniform	500	10	0.7558	0.01031
Lecun_uniform	500	10	0.7934	0.01081
Normal	500	10	0.7747	0.008086

Zero	500	10	0.1124	0.002416
He_uniform	500	10	0.809	0.01484

Table 6 Scores of different weight initialization methods

Table 6 shows that He_uniform (see reference [3]) is the best initializer. On the other hand, it can be seen that Zero initializer has a worse score than the random initializers. From now on, the He_uniform initializer will be used.

7. Second architecture optimization

Now, we will update the CNN architecture, adding more parameters to see if it can predict better the validation set.

The Architecture summary is shown in the next table:

Layer	**Output shape**	**Parameters**
Conv2D 1	30x30x64	640
Average pooling 1	15x15x64	0
Conv2D 2	13x13x128	73856
Average pooling 2	6x6x128	0
Conv2D 3	4x4x256	295168
Flatten	4096	0
Dense 1	175	716975
Dense 2	50	8800
Dense 3	10	510

Table 7. Architecture of the third model.

Input image is still 32x32x1, this model has more parameters than the previous one. Increasing complexity, may help collecting more features. In this model, 2D convolutional layers have a kernel size of 3x3 and a “relu” activation function. To train the model Nadam optimizer will be used with a batch size of 70, val_split of 0.1 and 100 epoch. However, as we are using the early stopping callback, training should stop earlier.
Results show that training stopped at epoch 6 with a validation accuracy of 0.928. Which is better than any of the previous results.

To improve even further the accuracy, we may consider decreasing de validation split, that way we can train the model with more data. The last experiment consists on reducing val_split to 0.001. This time, validation accuracy is 0.9667 at epoch 7 (see figure 5), however, since validation set is really small comparing to training data set, validation accuracy should not show the real accuracy.

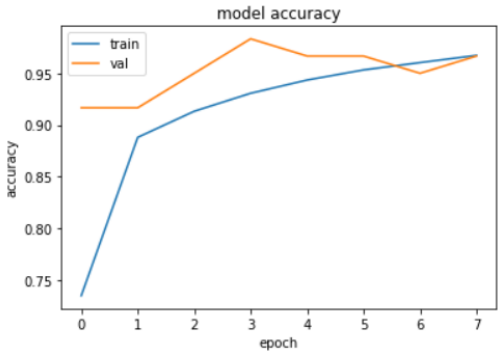

Figure 5 Third model accuracy

Last, we are going to predict the values on the test data and see if this last model can predict it well:

To test the model we have to reshape and standardize the x_test and predict classes using `model.predict_classes(x_test)`.

Accuracy in test set is: 0.9102.

As expected, this is less than the validation accuracy because we reduce the validation split, but it is an acceptable result. Note that we improved the first and second model accuracies.

8. Error analysis

A better way to evaluate the performance of the chosen model is to calculate a confusion matrix. This matrix shows what kind of errors the classifier makes, counting the number of times the algorithm misclassifies one class. For example, the value in the 5th row and 3rd column shows how many times the classifier confused images of 5's with 3's.

Next figure shows the mentioned confusion matrix where white color shows the most frequent confused digits:

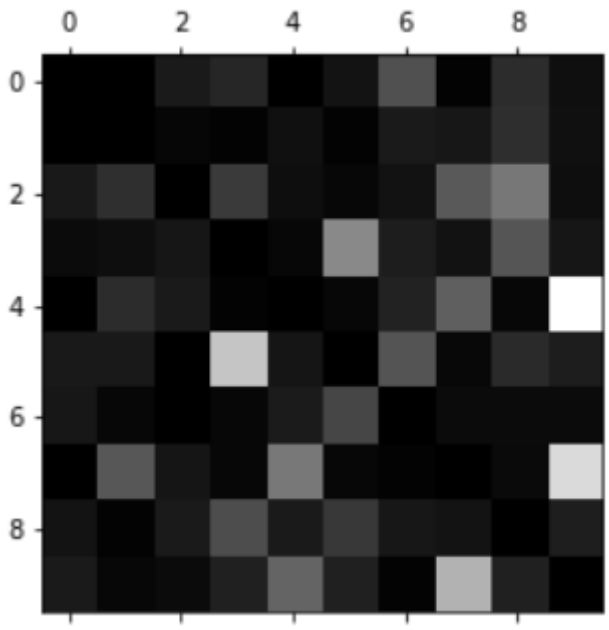

Figure 6 Confusion matrix

The diagonal has been set to zero to keep only the errors. One can clearly see that 0's and 1's are almost never confused, while 3's and 5's or 7's and 9's are many times misclassified.

9. Future work

Possibly, this problems accuracy could be improved. I will mention some ideas that can be applied on future projects:

1. Input images are quite noisy, so preprocessing those images before training the model so that images that enter to the CNN are clearer could be beneficial.
2. Trying different architectures, in this project we have used three different architectures but there are a lot of more options or combinations.
3. Getting more training data would be also beneficial, because generalization error could be decreased.
4. Use confusion matrix to see which are the common errors and figure out how to improve the model to avoid them.

In conclusion, we developed a convolutional neural network to predict noisy MNIST images. First, we started with a very simple architecture, then, we imitated the LeNet architecture and optimized model hyperparameters. Third, we upgraded the architecture adding more filters and layers to get our final network. Last, we defined the confusion matrix and gave some ideas for future work.

10. Reference

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," November 1998.

[2] Shirish Keskar, Nitish; Mudigere, Dheevatsa; Nocedal, Jorge; Smelyanskiy, Mikhail; Tang, Ping Tak Peter "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima" September 2016

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification" 2015