Arabic Handwritten Digits Classification

Neural Networks and Deep Learning
University of Zurich, Fall semester 2021
Robert Earle

Group

Axon Hillock

Authors

Raul Hochuli / 13-713-110

Liehao Xu / 19-736-412

Martin Zeller / 14-930-762

Table of Contents

Introduction

In our neural network project, we wanted to do something in the field of handwriting classification for digitisation, since we consider this very useful for various occasions. A few examples are the digitisation of handwritten archives, the digitisation of handwritten notes for example in the medical sector or in the education sector. We considered the digit classification similar to MNIST as a good start to get familiar with neural networks. Since we did not want to use the MNIST dataset we went looking for another suitable dataset to train our neural network.

Data description

We have found a suitable dataset for our project on (http://datacenter.aucegypt.edu/shazeem/) from the American University in Cairo. The dataset contains images of handwritten Eastern Arabic Numerals instead of Western Arabic Numerals used in MNIST. It contains 70'000 digits written by 700 different people. Each one wrote the Arabic digit 0 to 9 ten times. So every digit appears with the same frequency in both the training and the testing data. The data is split into 60'000 digits for training and 10'000 digits for testing. The images are saved in grayscale with a standardized size of 28 by 28 pixels. On (https://www.kaggle.com/mloey1/ahdd1) we found the same images already saved as a csv-file, where each number between 0 and 255 represents the brightness of a pixel.

Figure 1: Western Arabic Numerals and Their Corresponding Eastern Arabic Numeral

Western Arabic	0	1	2	3	4	5	6	7	8	9
Eastern Arabic[a]	٠	١	٢	٣	٤	٥	٦	٧	٨	٩

Source: https://en.wikipedia.org/wiki/Eastern_Arabic_numerals

Figure 2: Example Images from the Dataset

Source: http://datacenter.aucegypt.edu/shazeem/

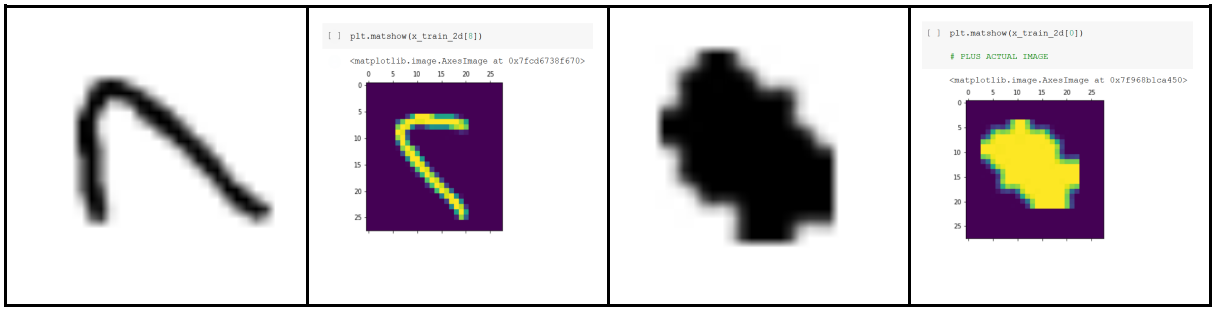
As the test data is already separated from the training data, we leave this partition untouched and will later separate the training dataset into training data and validation data. Throughout all the network setups, we always use a validation split of 0.3 on the training data and set the 'shuffle' operator to 'True'. This represents a decent partition of training to validation data, which resembles the partitions shown during the lecture. After loading the 4 data sets into the jupyter notebook, we quickly controll the dimensions. Looking at the output we see that the training dataset has the dimensions 60000x784 (pixel values), 60000x10 (digit labels), and similarly for the test data with 10000x784 (pixel values), 10000x10 (digit labels).

The 60000 corresponds to the number of observations of the data set. 784 results from the flattened pixel value matrix (dimensions 28x28). The data is therefore already pre-processed (flattened) and ready to be used for computation in neural networks.

Before starting we want to make sure that during this process, no structural errors were made. We therefore reshape the flattened array of pixel values into multidimensional matrices, which then can be plotted, using matplotlib. Comparing the actual image of the a digit and the plot reconstructed with matplotlib, we see that both images look identical in their pixel brightness. Thus we can continue without concern of faulty data.

In the following figure we see two examples of an original picture compared to a reconstructed image. We see that we messed up the reconstruction and the reconstructed images are mirrored on the diagonal.

Figure 3: Comparison of Original Pictures and the Reconstruction from the CSV-File



The value of a pixel is bound between 0 and 255. We use these “raw” values in the beginning for our first neural net. following the lecture slides, we decide later to scale the values (dividing all values by 255) to range between 0 and 1. This should decrease noise and help the network’s accuracy, as the values are more closely distributed around 0.

We do not consider further exploratory data analysis or descriptive statistics, as the data is already well pre-processed and digits are equally distributed (as mentioned before).

Neural Networks

Before starting to evaluate individual networks, we would like to mention the thought process going into this section. We approach the classification of the eastern Arabic numerals, by first computing very simple and basic networks. These are augmented and refined, using multiple layers, dropout layers, pooling layers and eventual convolutional layers. With each new network we try another layer specification, paying extra attention to one of the aforementioned layer classes. During the evaluation, the network's hyperparameter configuration like “optimizer”, “loss” function and “metrics” remain unchanged. We intend to improve our network's accuracy first. After finding a well-performing network structure, we then continue to improve the network's performance by experimenting with the mentioned hyper parameters.

Testing each network in the end against the test data, we can see if the network suffers from over- or underfitting. All networks (with a few exceptions) have a very similar validation accuracy and test accuracy. This is an indicator for a good fit. The third convolutional network with the parameter specification (Different optimizer, learning rate 3) has an accuracy of 10%, while the validation accuracy bounced around much

higher values. This would be a clear indicator of overfitting, where the network performs seemingly well validation, but much less so in the actual test.

First Neural Network

For a start we run a very simple one layered neural network. We chose 10 nodes in our network, such that each node is “responsible” for 1 of the 10 digits which we are looking for. We use a basic sigmoid function for the activation function as well as the following parameters: optimizer='adam',loss='categorical_crossentropy' and metrics=['accuracy']. Returning an accuracy on the test data of 94%, we see that the classification does not work badly, but there is still room for improvement.

Second Neural Network with Standardized Pixel Values

In the second neural network we kept the first network and standardized the pixel values by dividing them by 255, so that the values range from 0 to 1. We did this because the algorithms often work better with standardized values. This change increases the accuracy from around 94% to around 95%. As this gives better accuracy, even if only a little bit, we henceforth continue to use the standardized pixel values.

Third Neural Network Using Multiple Layers

In our third attempt we used a network presented in the lecture, where it was used for MNIST. This network uses ‘relu’ as the activation function instead of the ‘sigmoid’ function from our first two networks. Further it features multiple dense layers, each with 512 nodes and the ‘relu’ activation function, with some dropout layers in between the dense layers. For the last layer, we still use a dense layer with 10 nodes (for 10 digits) but changed the activation function to ‘softmax’ as this is supposed to work better for number classifications. This last layer remains unchanged for all the following networks. This network gives an accuracy of 98.08% for 5 epochs which already is a considerable improvement, using only a couple more layers with very little computation time. But we still wish to perform better.

Fourth Neural Network

Having seen during the course lectures that flatten sometimes improves performance, we take our third network and add a flatten layer after each dropout layer. After 10 epochs the network now has an accuracy of 98.23% which is slightly higher.

Fifth Neural Network Using More Layers with Decreasing Sizes

In an attempt to find an even better network for the 1-dimensional pixel value arrays, we construct a multi-layered network, featuring a repetitive structure of dense, dropout and flatten layers. For each repetition the number of nodes is decreasing from 512, 342, 225, 135, 81 (each with the 'relu' activation) and eventually to the last known layer with 10 nodes with 'softmax' activation function. Using 10 epochs, the network has an accuracy of 98.18% with a runtime of roughly 5 seconds per epoch.

Even though the runtime is not very big, we feel that we reached a certain limit of accuracy with the use of 1-dimensional pixel value arrays and multi-layered dense neural networks. We proceed to implement convolutional neural networks, as the lecture also features those in the use of image recognition. We first reshape the training and test data sets to an array of dimensions (60000,28,28,1) for training data and (10000,28,28,1).

First Convolutional Neural Network

For our first convolutional neural network, we use a Conv2D layer with 12 filters and a filter size of 5, followed by a max pooling layer with a pool size of 2. This structure (Conv2D, max pooling) is also used in the lecture. After a flatten layer the final layer of the network remains the 10 node dense layer with the softmax activation function. Using 10 epochs we achieve an accuracy of 98.70% with a runtime per epoch of ca. 14s. This is again a slight improvement to the fifth network, using only dense layers.

Second Convolutional Neural Network

In our next network setup, we again use a Conv2D layer followed by a max pooling layer. This time however, we use a kernel the size of 6x6, filter output space of 32 and again the 'relu' activation function. The following max pooling layer features a pooling size of 5x5.

Using 10 epochs we achieve an accuracy of 98.74% with a runtime per epoch of approximately 15s.

Third Convolutional Neural Network

In this network setup, we keep the setting of the first Conv2D and max pooling layer and add more dense layers afterwards, first using 255 nodes and 128 nodes, both times again with the 'relu' activation function. The last layer is again the dense 10 node 'softmax' layer. We hope to refine the accuracy with those two dense layers while simultaneously not increasing the computing time too much. Using 10 epochs we achieve an accuracy of 99.13% with a runtime per epoch of approximately 14s. Compared to the fifth neural network (using only dense layers), this is a considerable increase in accuracy with the same computation time.
While this is a good progress, we still try to find a better setup of convolutional layers to improve accuracy even further.

Fourth Convolutional Neural Network

We now try to use sequential convolutional layers. first adding a padding zeros of 1 pixel around the standardized pixel value array, we then use 3 Conv2D layers. With each layer the output space increases from 32, 64 to 128 and the kernel size decreases from 14x14, 6x6 to 3x3. After the max pooling layer (2x2) we try again to improve the accuracy with several dense layers (also decreasing in the number of nodes like in the previous network) and also several dropout layers in between. As the validation accuracy is still increasing up to 10 epochs, we increase the number of epochs such that the accuracy and validation accuracy converge around a threshold.

Using 15 epochs we achieve an accuracy of 98.90% with a runtime per epoch of approximately 114s. Compared to the third convolutional network, the additional layers did noticeably increase the runtime by almost a factor 10 while not improving the accuracy.

Fifth Convolutional Neural Network

Still being convinced that sequential convolutional networks might improve accuracy, we construct another neural network. The previous network featured a very large kernel at the beginning and an increasing output space with each layer. This time we start with a much smaller kernel dimension of 6x6 which decreases all the way down to 2x2, each time decreasing by 1 pixel in each dimension. The output space oscillates between 32 and 18, thus clearly not increasing. With the switches of output space between the layers, we hope to decrease runtime while still increasing accuracy.

Using 15 epochs we achieve an accuracy of 98.830% with a runtime per epoch of approximately 133s. With the added convolutional layers, the runtime has increased again drastically (despite having a smaller output space for each layer). This did however do nothing to improve our accuracy. On the contrary, the accuracy is slightly lower compared to the third convolutional network. We arrive at the conclusion that increasing the number of convolutional layers does not really help to improve accuracy.

Sixth Convolutional Neural Network

With the last network, we have seen that using more sequentially added convolutional layers does not improve performance (both in runtime and accuracy). In our last network we therefore try to further follow the approach of the third convolutional neural network, which is: having first a convolutional layer with medium kernel size, which is followed by multiple dense layers which are decreasing in the number of nodes. We therefore add 4 dense layers (each followed by a dropout layer) with 255, 128, 64 and 32 nodes.

Using 15 epochs we achieve an accuracy of 99.03% with a runtime per epoch of approximately 15s. Similar to the third convolutional network, we have a higher accuracy than with the previous two networks and a much lower runtime. However, the additionally added dense layers did not further improve accuracy.

Parameters specification

Having demonstrated multiple network structures in the previous section, we draw the conclusion that the third convolutional network has the best performance, whilst featuring a small and simple layer setup. We now continue trying to improve the network further by experimenting with the parameters 'loss' (function), 'optimizer' and 'batch_size'.

Different loss function

Using *loss = keras.losses.categorical_crossentropy*, we have an accuracy of 99.14%, with a runtime of 15s per epoch for 15 epochs.

Different loss function, different optimizer

Using *loss = keras.losses.categorical_crossentropy* and *optimizer=keras.optimizers.Adadelta()*, we have an accuracy of 77.44%, with a runtime of 15s per epoch for 15 epochs. This clearly performs much worse than the initial setup.

Different loss function, different batch size

Using *loss = keras.losses.categorical_crossentropy* and *batch_size = 512*, we have an accuracy of 99.10%, with a runtime of 13s per epoch for 15 epochs. Changing the batch size seems to slightly improve runtime but has no apparent effect on the accuracy of the network.

Only different batch size

Using again the *loss = 'categorical_crossentropy'* (as in the initial setup of the third convolutional neural network) and *batch_size = 512*, we have an accuracy of 98.76%, with a runtime of 13s per epoch for 15 epochs. We see that maxing out the batch size does not further improve accuracy.

Only different batch size 2

Using again the *loss = 'categorical_crossentropy'* (as in the initial setup of the third convolutional neural network) and *batch_size = 2000*, we have an accuracy of 99.08%, with a runtime of 13s per epoch for 15 epochs.

Different optimizer, learning rate

Using *loss = keras.losses.categorical_crossentropy* and the RMSprop optimizer with a vanishingly small learning rate of 1e-6, we have an accuracy of 66.55%, with a runtime of 15s per epoch for 15 epochs. Clearly the learning rate has been chosen too small for the network to improve in performance. We continue to experiment with a higher learning rate.

Different optimizer, learning rate 2

Using *loss = keras.losses.categorical_crossentropy* and the RMSprop optimizer with a small learning rate of 0.01, we have an accuracy of 98.08%, with a runtime of 15s per epoch for 15 epochs. This learning rate seems to be suitable, as the accuracy has increased again to the "previously known level" without taking a toll on the runtime. Maybe further increasing the learning rate might yield a higher accuracy.

Different optimizer, learning rate 3

Using *loss = keras.losses.categorical_crossentropy* and the RMSprop optimizer with a rather big learning rate of 0.1, we have a terrible accuracy of 10.00%, with a runtime of 15s per epoch for 15 epochs. Clearly the learning rate has been chosen too high. The network bounces around, not finding suitable weights for the optimization, resulting in an prediction accuracy not better than simply guessing the values.

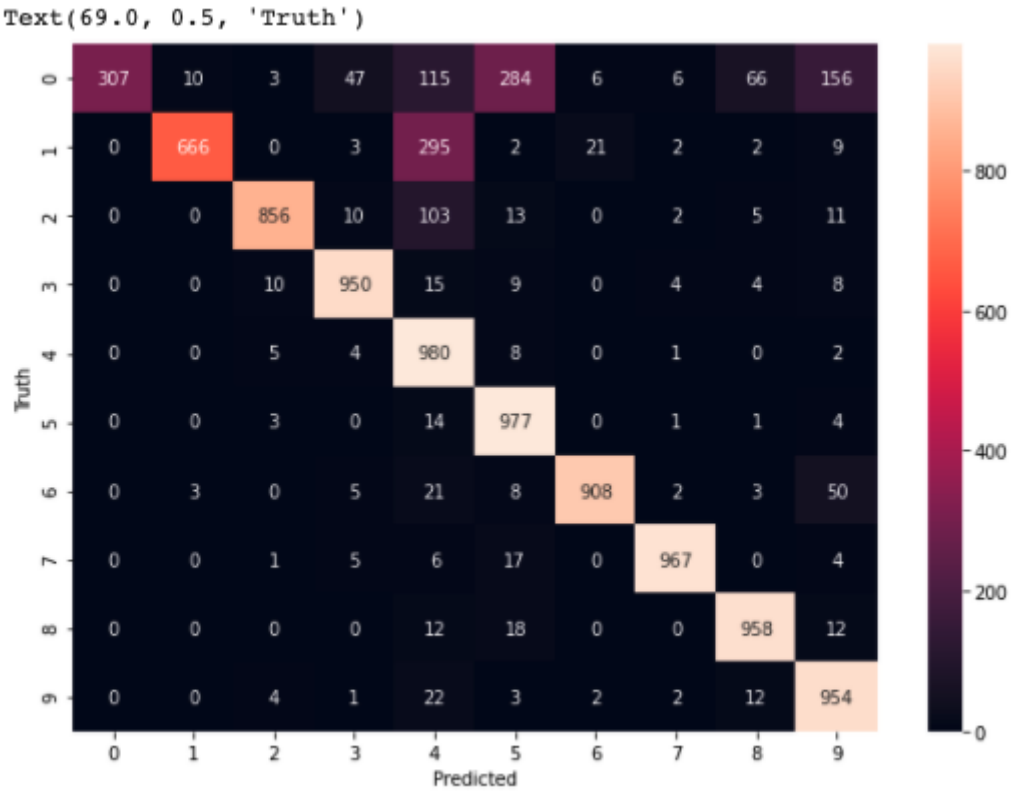
Considering the use of the different parameter specifications shown above, we see that none of them yielded an additional improvement of the network accuracy.

Prediction accuracy

To further improve and understand our networks, we decide to plot a confusion matrix for two of the networks shown in the Neural Networks section. In the figure below we show a confusion matrix for our very first (and very basic) neural network. We remember that the accuracy was 94%. The figure shows both heat map colouring indicating the cluster of miss predictions, as well as the actual number of miss predictions for the digits.

The first neural network clearly performed poorly in predicting the true label for the number 0. Interestingly the number 0 has been predicted to be 5, almost as often as it has been predicted correctly. Vice-versa the number 5 has been predicted correctly with a very high accuracy.

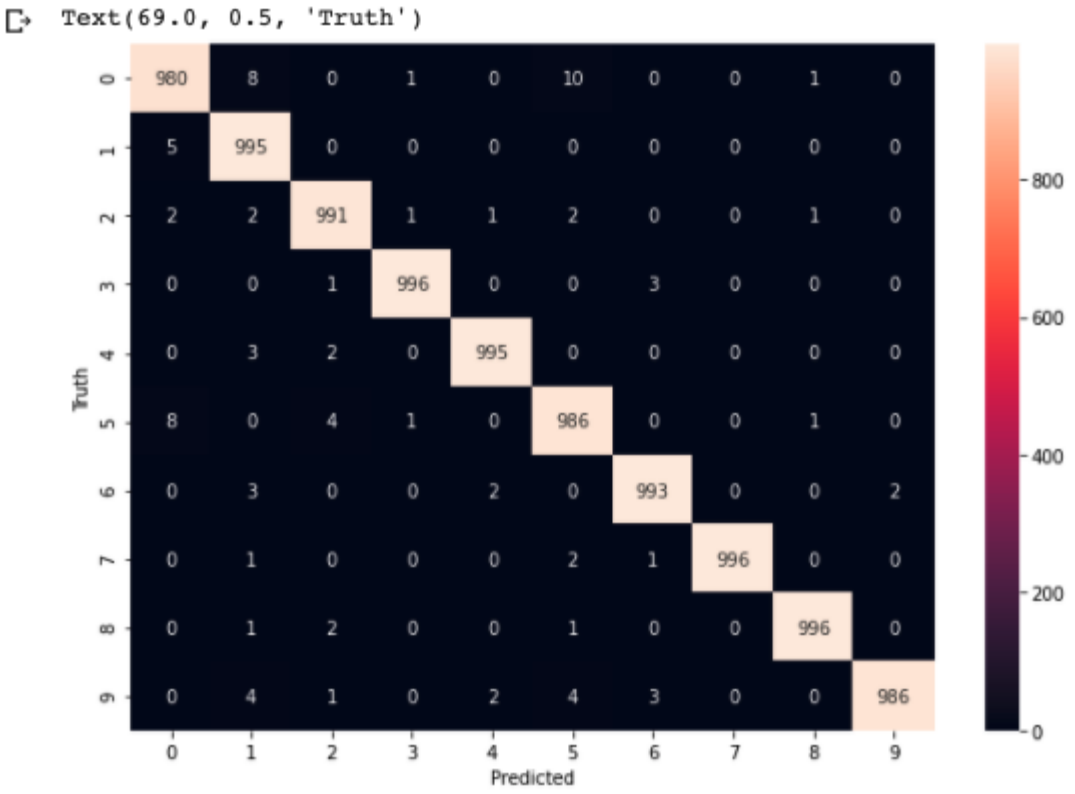
Figure 4: Confusion Matrix of our First Neural Network



Further looking at Figure 5, we see that those inaccuracies have been adjusted for in the third convolutional neural network. The cell (True = 0, Predicted = 5) in the confusion matrix has still the highest value, being the biggest cluster for miss prediction. However,

the number 0 is being much better predicted, which improves the overall performance of the network a lot, up to 99.14%.

Figure 5: Confusion Matrix of our Third Convolutional Neural Network



Network Comparison

Out of interest we compared our best network (third convolutional neural network) with the convolutional network for the recognition of MNIST handwritten digits from the lecture (mnist_cnn.ipynb). Using 10 epochs and a batch size of 128, the network in mnist_cnn.ipynb gives an accuracy of 99.12% with a runtime of 79s per epoch. Our network performs at a very similar accuracy with 99.14%. However, it runs much more efficiently, using only approximately 14s per epoch.

Conclusion

In our project we reached our goal to find a suitable neural network for the classification of handwritten digits. Having already discussed MNIST during the lecture at length, our project expends on those existing findings and extends them to handwritten eastern Arabic numerals. We experimented with numerous network structures, eventually finding a well performing network. During the process of achieving increased accuracy, the importance of computation time became clear. Finding a good balance between runtime and accuracy is essential, for more complex networks which would also recognise more than numerals.

Another idea to further improve our network is to standardize the input values (pixel values) to values between -1 and 1 instead of 0 to 1. This might help the 'relu'-function to perform better and eventually lead to an even higher accuracy.