Project 2 - README

ΜΥΣΤΑΚΙΔΗΣ ΙΩΑΝΝΗΣ - 1115201600113
ΧΟΥΣΙΑΔΑ ΕΥΑΓΓΕΛΙΑ - 1115201600200

Github: https://github.com/myioannis/Project-2 (Public Repository).

How to run:

We've included the executables in two forms. A .py and a .ipynb version of the files:

- .py: You can run the files exactly as specified in the assignment. That is:

  $python autoencoder.py –d dataset

  $python classification.py –d trainingSet –dl trainingLabels
  -t testSet -tl testLabels -model autoencoder.h5

  If you ommit any of the aforementioned arguments, the program will explicitly ask for them from the user

- .ipynb: You can run the files as a jupyter notebook either **Locally** or on **Google Colab** for a better performance:
  - Locally: You can run these jupyter notebooks on your local computer, although it is probably going to run very slowly. Every argument, that was previously given through the command line, is now going to be explicitly asked as input from the user. Make sure to provide the correct (relative) paths when asked.
  - Google Colab: In order to run these jupyter notebooks on google colab you have to upload them to your google drive account and then open them from colab. About the datasets:
    * You can either upload the datasets to your google drive account as well and use them from there. However, in this case, you'll have to mount your google drive folder with your current colab session in order to have access to these files.
    * Or you can simply run the first cell in our code, which clones our github repository, that contains the datasets as well, into your

current colab session and changes the current path to this directory. So you can access the datasets with no further effort. Our best autoencoder model is also uploaded in our github repository. After you run the first cell, the current working directory on your session will be the cloned repository's folder, so you can refer to the datasets and the autoencoder by simply using their names as the path. We made the repository public in order to avoid the authentication that would be asked in case the repository was private. The last cell in our notebook deletes the folder of the cloned github repository from current colab session but this is not really needed since all files are deleted when the session ends.

Make sure the datasets are **not unzipped**, because the unzipping is done by our code.

In case you want to load our own autoencoder model in the classification, the keras version at the time of training was 2.4.3 and the tensorflow version was 2.3.0.

Modules:

- **autoencoder** module: Contains the code for the neural network autoencoder (N1). It creates a model according to the user's specified hyperparameters and can save this model for futher use by the classification module.
- **classification** module: Contains the code for classification part of the assignment (N2). It makes use of the encoding part of an autoencoder model and classifies the test data.
- **utilities** module: This module contains some functions used by the other two modules, like parsing the command line arguments, reading hyperparameters, preprocessing data etc.

Experiments:

We carried out way too many experiments to be able to show the results of all of them and, since we tried out large epoch sizes, it would require way too much time to reproduce them. Therefore, we thought that it would be better to present the plots and some statistics for our best models and comment on the rest of our experiments verbally on chunks of experiments (not visually on individual experiments) to justify our hyperparameters choices.

Autoencoder:

After various experiments with the hyperparameters and the architecture of the autoencoder's model, we concluded to the following insights:

- There wasn't any substantial change in the accuracy of the model when the batchNormalization after the convolutional layer and when doing it after the maxPooling+dropout (i.e. before the next convolutional layer).
- A simple model works best, because a more complex one usually overfits on the data.
- A large batch size fits better on the data than a smaller one (because it takes into consideration more data every time it updates the weights) but leads to poor generalization for our models (for the same reason), whereas a small one does not.
- For us, dropout layers were essential in order to avoid overfitting. Not so much in the autoencoder itself, but those dropout layers in the autoencoder helped avoid (in some cases) the overfitting that appeared in the classification later. We tried various positions for the dropout layers (e.g. before the convolutional layers, after the batchNormalization layer, before the maxPooling etc.), some of them didn't work well and some didn't show any difference from the current position of the dropout layers in our code. A probablility of 0.2 worked best for us. A smaller probability would make the autoencoder converge a little faster with obvious benefits for the autoencoder's loss and no overfitting observed, but this small probability would have a negative impact on the classification where the overfitting would be very high.
- There wasn't any significant difference observed when changing the size of the kernel.
- Two convolutional layers worked best at avoiding overfitting (even with 3 layers the difference was very high) and therefore the number of convolutional filters has to be high in order for the autoencoder to fit enough on the data (a small number of filters for each layer prevented the autoencoder from converging to a small loss, even for the training data, not just for the validation).
- After about 30-50 epochs there wasn't substantial increase in the performance of the autoencoder. The loss was dropping very slowly.

An example of the aforementioned insights is the following model:

Layer (type)	Output Shape	Param #
input_12 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_67 (Conv2D)	(None, 28, 28, 128)	1280
batch_normalization_56 (Batc	(None, 28, 28, 128)	512
max_pooling2d_22 (MaxPooling	(None, 14, 14, 128)	0
dropout_44 (Dropout)	(None, 14, 14, 128)	0
conv2d_68 (Conv2D)	(None, 14, 14, 256)	295168
batch_normalization_57 (Batc	(None, 14, 14, 256)	1024
max_pooling2d_23 (MaxPooling	(None, 7, 7, 256)	0
dropout_45 (Dropout)	(None, 7, 7, 256)	0
conv2d_69 (Conv2D)	(None, 7, 7, 256)	590080
batch_normalization_58 (Batc	(None, 7, 7, 256)	1024
up_sampling2d_22 (UpSampling	(None, 14, 14, 256)	0
dropout_46 (Dropout)	(None, 14, 14, 256)	0
conv2d_70 (Conv2D)	(None, 14, 14, 128)	295040
batch_normalization_59 (Batc	(None, 14, 14, 128)	512
up_sampling2d_23 (UpSampling	(None, 28, 28, 128)	0
dropout_47 (Dropout)	(None, 28, 28, 128)	0
conv2d_71 (Conv2D)	(None, 28, 28, 1)	1153

Total params: 1,185,793
Trainable params: 1,184,257
Non-trainable params: 1,536

Figure 1: Model: Number of Convolutional Layers=2, Kernel Size=3, Number of Filters for Layer1=128, Number of Filters for Layer2=256, Epochs=200, Batch Size = 64

As shown above, the decoder is a mirror of the encoder. A batch size of 64 worked pretty well compared to larger batch sizes and the number of epochs is not mandatory to be that high. We have only 2 convolutional layers (simple

models worked best) and a large number of filters per layer (the accuracy that was lost due to the simplicity of the model is now gained back by the large number of filters).

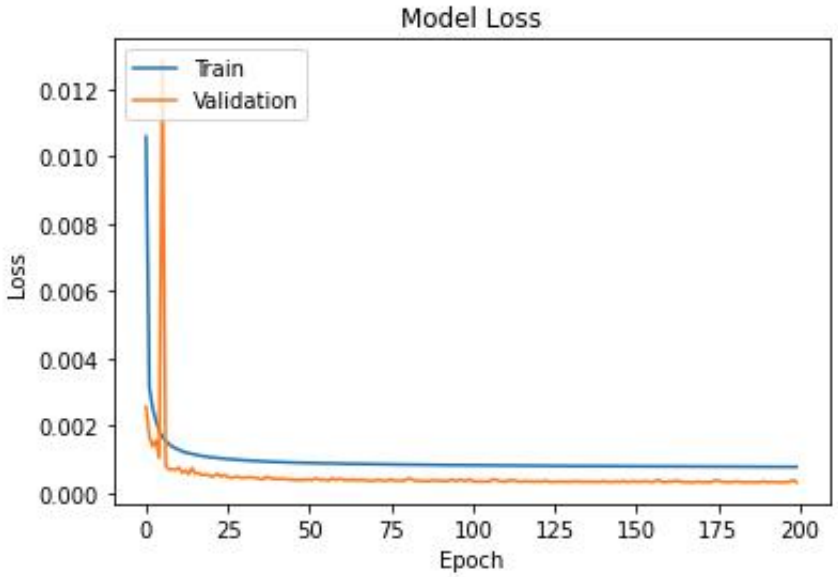


Figure 2: Model: Number of Convolutional Layers = 2, Kernel Size = 3, Number of Filters for Layer1 = 128, Number of Filters for Layer2 = 256, Epochs = 200, Batch Size = 64

The above plot shows that there wasn't any overfitting observed with our aforementioned autoencoder model. In fact, the validation loss is about half the training loss. This happens due to the dropout layers. Removing the dropout layers would close this gap and may even present overfitting in some cases, but even it doesn't present any overfitting on the autoencoder, this removal would have a negative impact later on the classification.

This above model was submitted alongside our code as autoencoder_model.h5.

Classification:

It appears that there is a trade-off between the different hyperparameters. We will sum up how the values of each parameter affected our experiments and then we will show how we tried to keep a balance between the effects of the choices of each hyperparameter value.

- A large number of nodes for the fully connected layer would increase the fitting on the data, whereas a smaller number would simplify the model

and make the classifier fit slower. The former presents the danger of overfitting on the data, whereas the latter presents the danger of not fitting enough.

- As usually, a very large number of epochs will make our model overfit. The number of epochs that worked for us depended on the values of the other hyperparameters. For example, if we ended up with a model that would converge very slowly to a high accuracy, then we'd want a high number of epochs, whereas a model that presented a danger of early overfitting would be treated with a smaller number of epochs.

- As mentioned earlier, a large batch size fits better on the data than a smaller one (because it takes into consideration more data every time it updates the weights) but leads to poor generalization for our models (for the same reason), whereas a small one goes up and down very often but at the end generalizes better (if it ever reaches a good accuracy).

- A large dropout probability (about 0.7) prevents the model from overfitting if handled correctly but makes the model fit slower on the data. A small dropout probability (around 0.3) almost always made the model overfit.

We conducted out countless experiments in order to balance the values of the all the variables above that affected our model, in order to **avoid overfitting** and achieve **high accuracy/low loss**.
We found out that the most difficult part was to avoid overfitting, because relatively good prediction results on the test set were possible even when overfitting was present. For example, a complex model with the encoding part of the autoencoder mentioned earlier and 256 or 512 nodes on the fully connected layer could achieve around only 70 incorrect labels, but the overfitting was extreme, since the accuracy on the training set almost reached the unit (1). Since we believe that such a model (even though it acted well upon the test set) is not a good model for the 'real world', we tried to reduce the overfitting and keep the accuracy at the same levels.

A dropout probability of 0.5 showcased a high sensitivity to overfitting for 128,256 and 512 nodes for the fully connected model (so even lower probability presented even higher overfitting). The only way to avoid overfitting was to use only 64 nodes for the fully connected layer. However, in that case, the model wouldn't fit enough on the data for smaller batch sizes (64 or 128), even for a large number of epochs, and it would overfit for larger batch sizes. It appeared that there wasn't a good model in-between these approaches that would balance the effects of each variable. So we thought that, since we couldn't increase the accuracy, we should increase the dropout probability in order to avoid the overfitting in aforementioned cases that it appeared.

We managed to get rid of overfitting with a dropout probability of 0.7, and

a fairly good model with 128 nodes on the fully connected layer and a batch size of 256. It achieved around only 120 incorrect labels after 30 epochs and no overfitting was present. However, we managed to do even better:

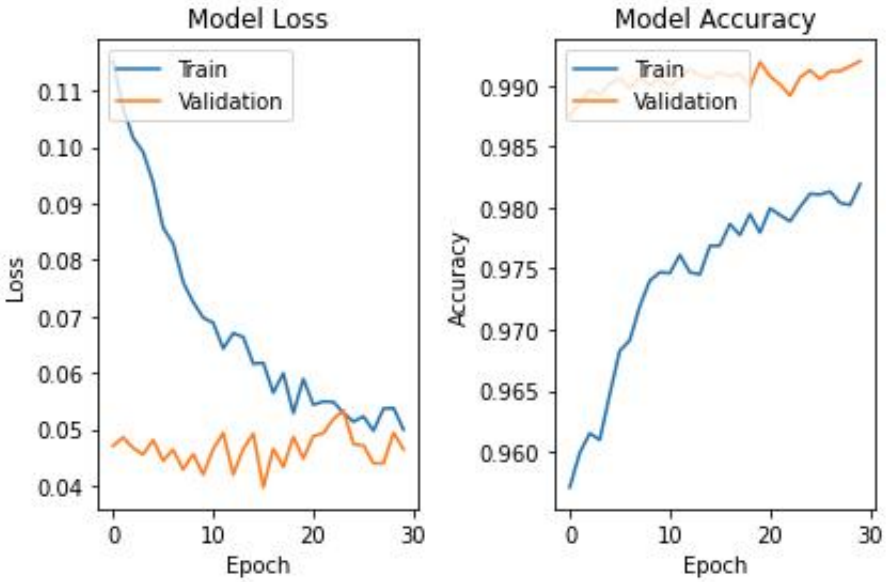
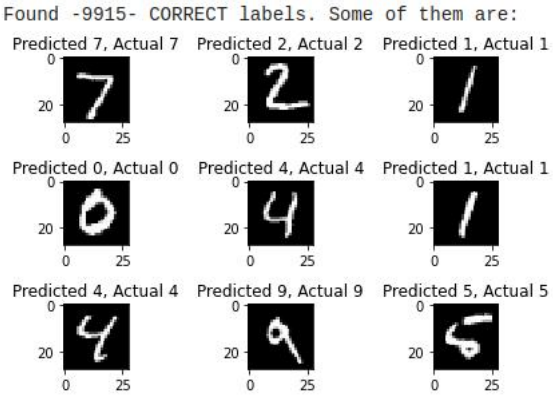
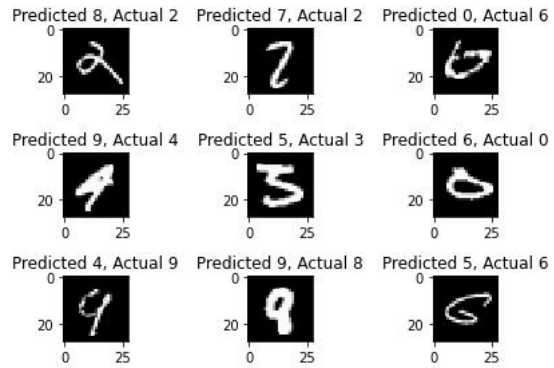


Figure 3: Model: Number of Fully Connected Nodes = 64 Epochs = 30, Batch Size = 512 and Dropout Probability = 0.7



Found 85 INCORRECT labels. Some of them are:



	precision	recall	f1-score	support
Class 0	0.99	1.00	0.99	980
Class 1	0.99	1.00	1.00	1135
Class 2	0.99	1.00	0.99	1032
Class 3	0.99	0.99	0.99	1010
Class 4	0.99	0.99	0.99	982
Class 5	0.99	0.99	0.99	892
Class 6	1.00	0.99	0.99	958
Class 7	0.99	0.98	0.99	1028
Class 8	0.99	0.99	0.99	974
Class 9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

As the above figures show, we managed to avoid overfitting and still achieve a very high accuracy and low loss on the test set (only 85 incorrect labels). The large batch size makes up for the low number of nodes on the fully connected layer (different hyperparameter combinations either lead to overfitting or resulted in very low accuracy).

In conclusion, we believe that it isn't possible to do **a lot** better than that, in terms of **accuracy, without overfitting** using conventional methods. Avoiding overfitting is quite difficult and if you manage to do it, you instantly fall on accuracy. Keeping a balance between the different variables affecting the model and avoiding overfitting doesn't mean that we can keep increasing the accuracy. We believe that the only way to do a lot better is to make our model more complex (e.g. higher number of nodes on the fully connected layer and a smaller dropout probability) in order to improve the accuracy and make use of a little bit more advanced methods to avoid overfitting, like **data augmentation**.