ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ + ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Software Development for Algorithmic Problems. 7th Semester 2020, Assignment 2: Convolutional Autoencoders using MNIST database.

`National and Kapodistrian University of Athens, DiT.                1/12/2020.`

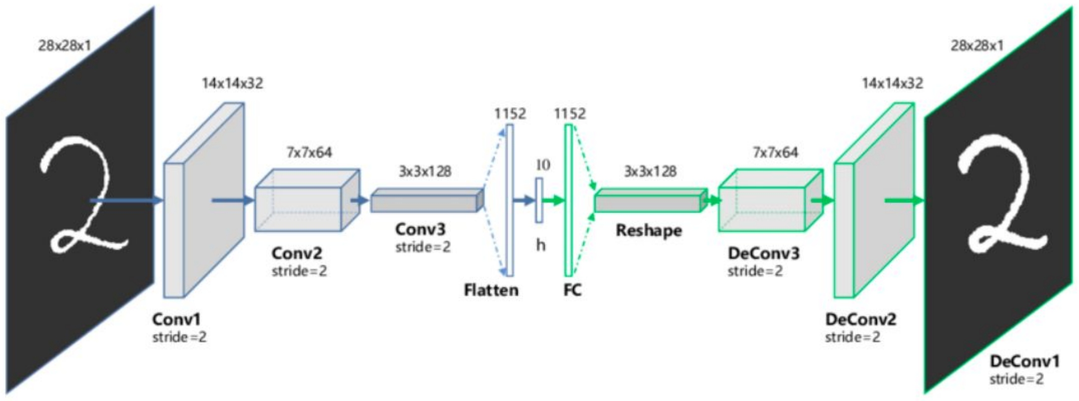
Evgenia Kalyva | sdi1600256
Dimitrios Foteinos | sdi1700181

Table of Contents:

Abstract.

Part A) Constructing a Convolutional Autoencoder Neural Network.

In this specific section of this assignment, we had to Construct a fully dense Convolutional Autoencoder Neural Network. Of course, in order to make this possible, we had to use Keras API and its utilities.



As shown in the image, the main idea here is to decompose a given image (28 x 28 x 1), and pass it through a multiple number of convolutional layers. Also, we will do some downsampling in order to reduce the dimensionality of our data.

In the process of training the model, the user can choose among a wide range of hyperparameters, such as:

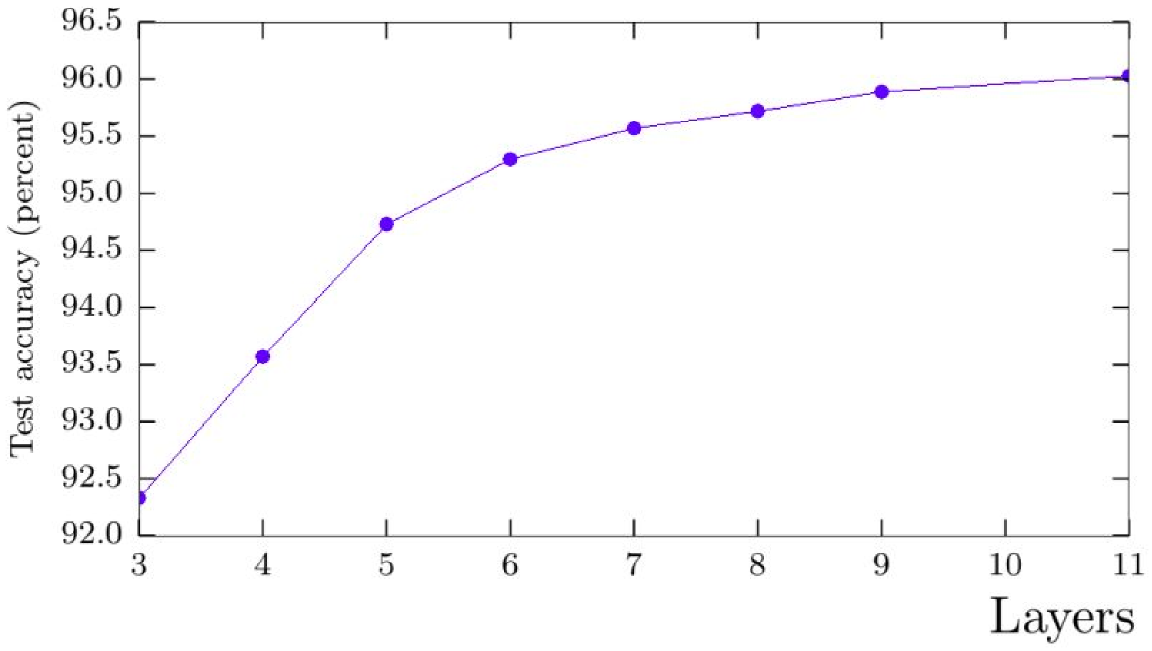
1. Epochs
2. Batch Size
3. Number of Convolutional Layers
4. Kernels per layer
5. Kernel Size

In this part, we must proceed in a research in order to find the best values for these hyperparameters so that the loss can be minimized as much as possible. Last but not least, we must avoid falling into the trap of overfitting.
To run our experiments, we used Google Colab and we utilized the power of the given GPU (the .ipynb file will be included within this README).

In order to start exploring the best combinations of values for our hyperparameters, we must first make clear how the model behaves after said values change.

The first Hyperparameter we want to examine, is the number of the convolutional layers.

Many tests have shown that deeper networks generalize better and they give us better results. But, after a finite number of layers, the accuracy finally converges, and thus, more layers will be a **waste** of computational power. We can see that at the image below:



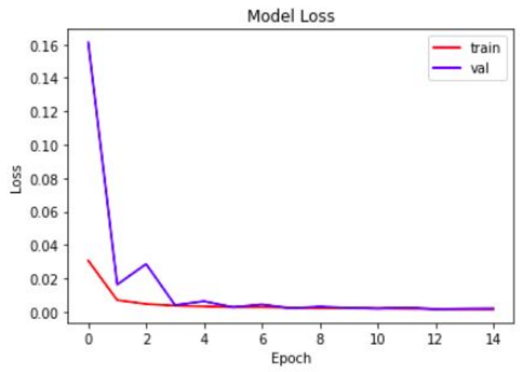
With all that being said, we'll use the default values:

1. Batch Size: 256
2. Epochs: 15
3. Kernel Size: 3 x 3
4. Kernel per layer: 32

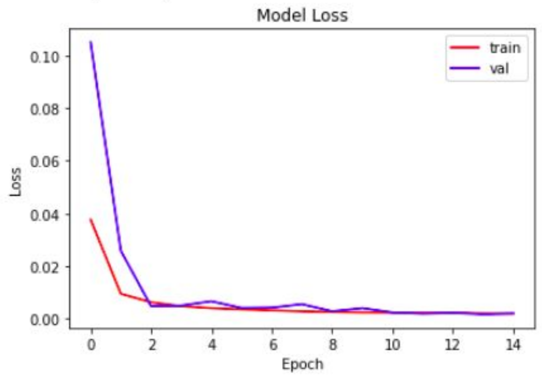
And see how the number of convolutional layers can affect our loss. (Consider we use a mirroring architecture, so we'll have the **same** number of layers at encoder and decoder respectively.

<u>Convolutional Layers:</u>

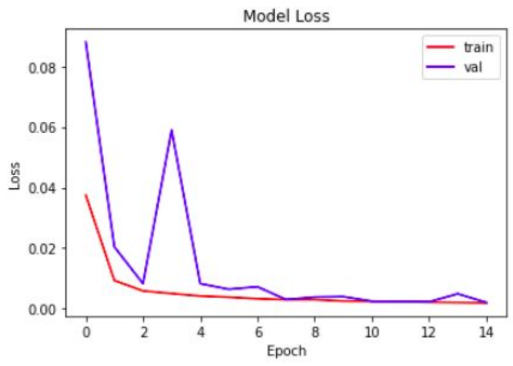
```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



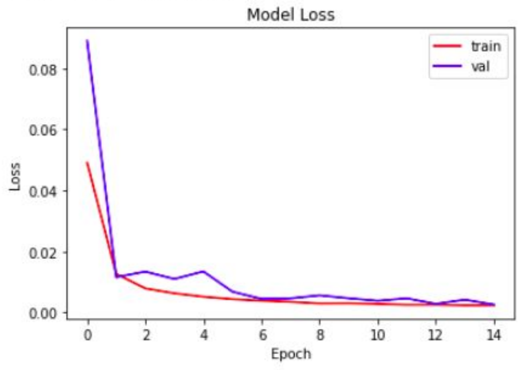
```
Epochs: 15
Batch size: 256
Convolutional Layers: 12
Kernel size: 3 x 3
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 14
Kernel size: 3 x 3
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 18
Kernel size: 3 x 3
Kernels per layer: 32
```



-

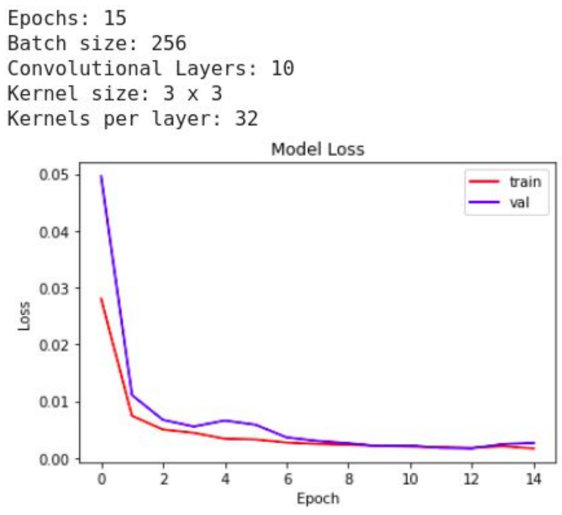
And for the last epoch for each number of layers, we have:

1. 10L : - 6s 34ms/step - loss: 0.0016 - val_loss: 0.0020
2. 12L : - 8s 43ms/step - loss: 0.0021 - val_loss: 0.0021
3. 14L : - 10s 51ms/step - loss: 0.0019 - val_loss: 0.0020
4. 18L : - 19s 102ms/step - loss: 0.0024 - val_loss: 0.0026

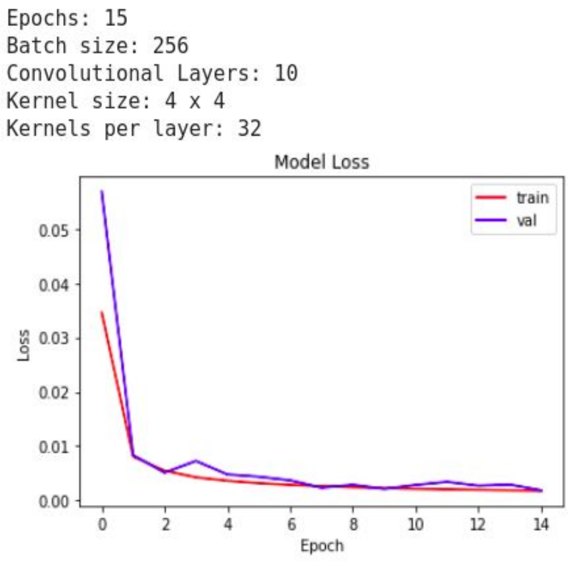
Surprisingly, the best number of Layers was only 10! We had almost similar results and for the other numbers of layers, i.e for 14L , but the time was dramatically more slow. So, we'll choose 10 Layers only to continue our examination!

The next parameter we want to check is the Kernel size. The Kernel size will play an important role in the behavior and results of our program. We'll start from Kernel_Size = 3, because the lower ones are producing **terrible** results. (i.e, for KS = 1 : `- 4s 22ms/step - loss: 0.0422 - val_loss: 0.6703`)

<u>Kernel Size:</u>



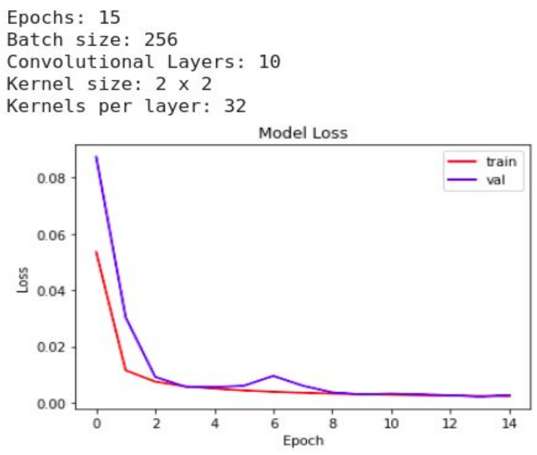
```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 4 x 4
Kernels per layer: 32
```

And for the last epoch, for each number of KS:

1. *2KS = - 6s 33ms/step - loss: 0.0023 - val_loss: 0.0027*
2. *3KS = - 6s 34ms/step - loss: 0.0017 - val_loss: 0.0026*
3. *4KS = - 10s 56ms/step - loss: 0.0018 - val_loss: 0.0019*



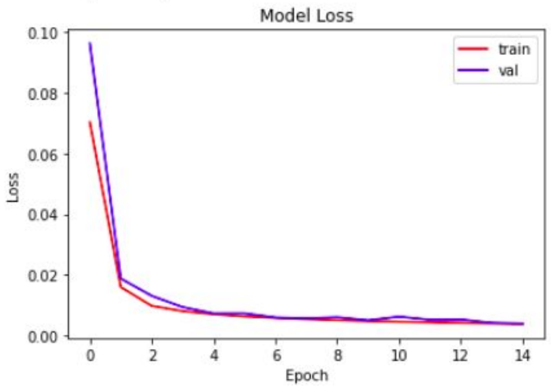
```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 2 x 2
Kernels per layer: 32
```

We can clearly see that the **optimal** value for this experiment is Kernel Size = 3. The trade-off between time and loss is the best one.

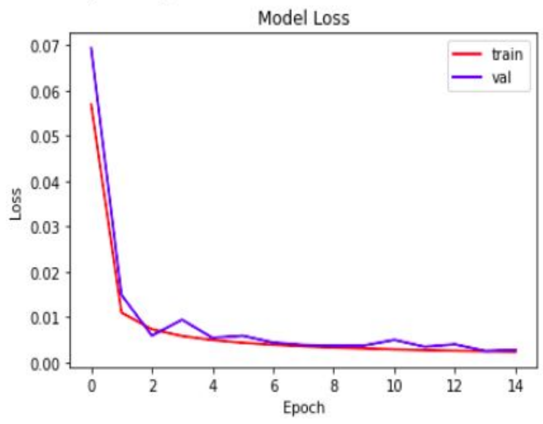
The next value that we want to configure, is none other than the Kernels per Layer. Let's see what happens when we experiment with that parameter:

Kernel Number:

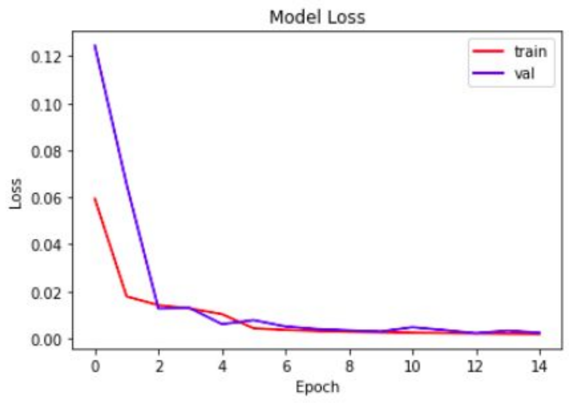
```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 2 x 2
Kernels per layer: 16
```



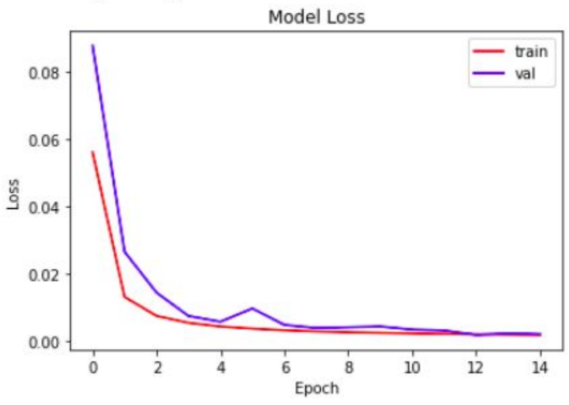
```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 2 x 2
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 2 x 2
Kernels per layer: 50
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 2 x 2
Kernels per layer: 64
```



And for the Last Epoch, we have respectively:

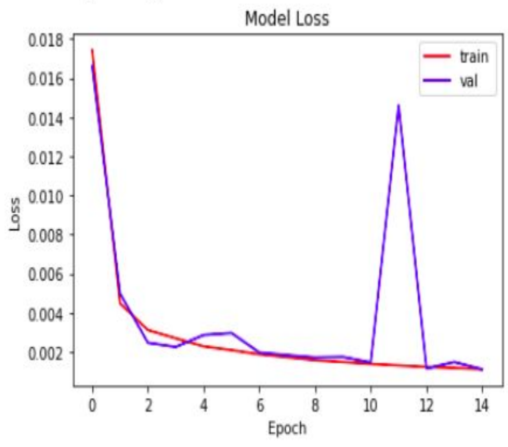
1. KN=16: - 3s 18ms/step - loss: 0.0039 - val_loss: 0.0040
2. KN=32: - 6s 33ms/step - loss: 0.0023 - val_loss: 0.0027
3. KN=50: - 11s 60ms/step - loss: 0.0019 - val_loss: 0.0023
4. KN=64: - 14s 74ms/step - loss: 0.0017 - val_loss: 0.0019

For this case, we can see that as we increase the number for Kerner per Layer, we manage to have better loss, but the time increases. For example, the step from KN=32 to KN=50, take almost x2 time(!) for only 0.004 better loss in the last epoch. Here, any value between 32-40 may be good for our research. We'll continue with KN=32 , because we are just fine with the loss that it gives in this value.

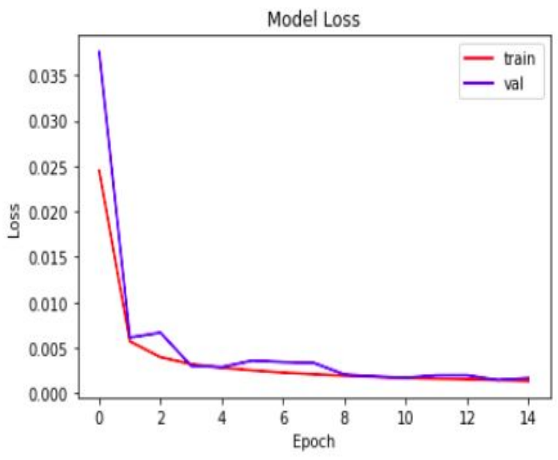
Before we come to an end, we have 2 more values to check. The first of these two, is the Batch Size. The Batch Size is the number of input samples that we check before proceeding to **change** the weights in our Neural Network. It's obvious that the bigger a batch size is, the faster the program will be. But it's possible we may lose some "accuracy" and thus, the loss will increase. Let's see a few examples:

Batch Size:

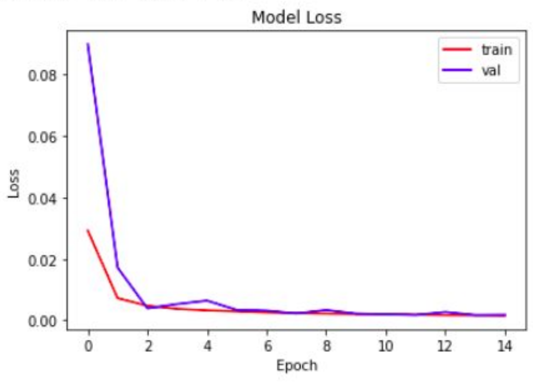
```
Epochs: 15
Batch size: 128
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



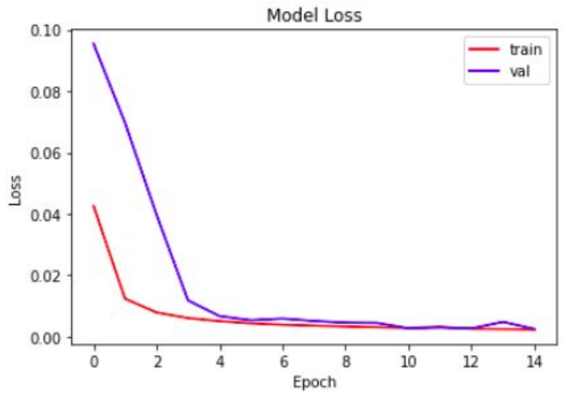
```
Epochs: 15
Batch size: 190
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 256
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



```
Epochs: 15
Batch size: 512
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```



And for the last epoch, we have:

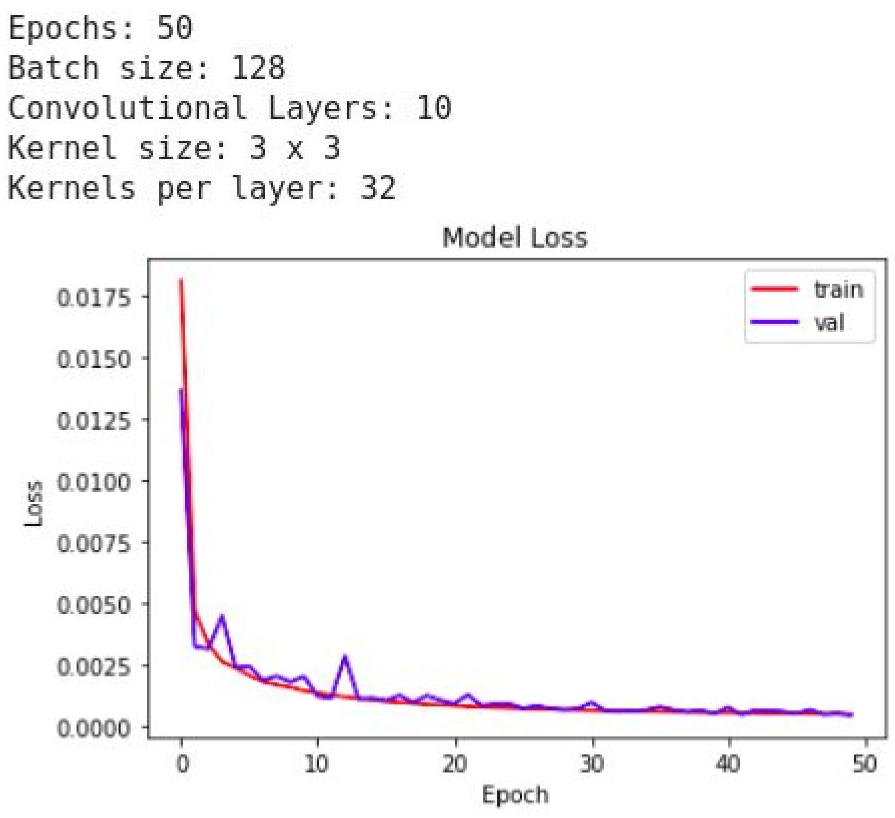
1. BS=128: - 7s 18ms/step - loss: 0.0011 - val_loss: 0.0011
2. BS=190: - 7s 26ms/step - loss: 0.0014 - val_loss: 0.0017
3. BS=256: - 6s 34ms/step - loss: 0.0016 - val_loss: 0.0018

4. BS=512: - 6s 64ms/step - loss: 0.0024 - val_loss: 0.0025

The results from Batch Size=128 are extraordinary. We'll choose this batch size because even if it's slightly more slow than the others (by 1s), the loss is exceptionally low.

Last but not least, we are going to test how the value of the epochs affects our results. Trying out the different values we will gradually observe a better convergence and a lower loss, but after a specific but yet unknown epoch value, our model will not be able to achieve any more enhancements.

<u>Epochs:</u>



```
Epochs: 50
Batch size: 128
Convolutional Layers: 10
Kernel size: 3 x 3
Kernels per layer: 32
```

We can see that after ~ 20 epochs, the loss doesn't become any lower.

Epochs:

1. E15: `7s 18ms/step - loss: 0.0011 - val_loss: 0.0012`
2. E16: `7s 18ms/step - loss: 0.0010 - val_loss: 0.0011`
3. E17: `7s 18ms/step - loss: 9.8810e-04 - val_loss: 0.0013`

4. E18: `7s 18ms/step - loss: 9.5444e-04 - val_loss: 9.8346e-04`
5. E19: `7s 18ms/step - loss: 9.0674e-04 - val_loss: 0.0013`
6. E20: `7s 18ms/step - loss: 8.9020e-04 - val_loss: 0.0011`
7. …
8. E50: `7s 18ms/step - loss: 5.1822e-04 - val_loss: 4.7909e-04`

Eventually, we notice that after the 20th epoch , the loss converges very slowly, as it is clearly shown in the plot above. Thus, a good size for epoch would be around 15 to 20. We'll choose 20.

Finally, the hyperparameters that we found are ideal for us are as follows:

- **Batch Size: 128**
- **Epochs: 20**
- **Convolutional Layers: 10**
- **Kernel Size: 3 x 3**
- **Kernels per Layer: 32**

Part B) Convolutional Autoencoder for Classification on MNIST images

The second part of this project implements a Convolutional Autoencoder for Classification using labels on MNIST datasets(train and test) using the framework

Keras and the open source library Tensorflow alongside modules such as numpy, matplotlib etc.

The implemented process can be summarized in the following simple steps:

I) Initialization

1. Import all needed libraries, modules and frameworks.

2. Get given arguments from the command line and ask the user for hyperparameters.

3. Get the hyperparameters(same as part A with the addition of fc, number of fully connected layers we want to have).

4. Read the autoencoder.h5 model that was created in part A and was given as an argument.

5. Load the training and test datasets and labels.

II) Process datasets and labels

6. Reshape the train and test data to the matrix size of 28 x 28 x 1, so they can be “fed” to the convolutional neural network.

7. Make sure the training and testing matrices are in float32 format and rescale them according to the maximum pixel value of the training and testing data respectively(in this case 255).

8. Convert labels into one-hot encoding vectors , a 1 x 10 dimensional row vector for each image

9. Split data into training and validation sets.

III) Perform training & evaluate on the classification model and predict labels

10. Define the model using the encoder we implemented in part A , which will stack up the Fully Connected layers of our choosing(note that the model we create in this step should have similar weights with the classification weight we loaded from autoencoder.h5).

11.Train , compile the model and save the classification model.

12. Retrain the entire model and save it again.

13. Maybe something about drop out

14.Evaluate model and predict labels.

IV) Results and options

After all those steps are completed the user is given the chance to see either plots about the loss of the train and validation sets or examples of the correct and incorrect predictions and rerun the program or the classification report, same as part A of the project.

Here are few examples of the results we get based on respective hyperparameters:

Classification Reports:

1) For **FC = 64:**

```
              precision    recall  f1-score   support

     Class 0       0.99      0.99      0.99       980
     Class 1       0.99      0.99      0.99      1135
     Class 2       0.99      0.99      0.99      1032
     Class 3       0.99      1.00      0.99      1010
     Class 4       0.99      0.99      0.99       982
     Class 5       0.99      0.99      0.99       892
     Class 6       0.99      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      0.98      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000


Test loss: 0.06451869755983353
Test accuracy: 0.9901000261306763
Correct labels found: 9899
Incorrect labels found: 101
```

2) For **FC = 128:**

```
              precision    recall  f1-score   support

     Class 0       0.99      1.00      0.99       980
     Class 1       1.00      0.99      1.00      1135
     Class 2       0.99      0.99      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       0.99      0.99      0.99       958
     Class 7       0.99      0.99      0.99      1028
     Class 8       0.99      0.99      0.99       974
     Class 9       0.99      0.98      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000

Test loss: 0.07818238437175751
Test accuracy: 0.9898999929428101
Correct labels found:  9898
Incorrect labels found:  102
```

3) For **FC = 256:**

```
              precision    recall  f1-score   support

     Class 0       0.99      1.00      0.99       980
     Class 1       0.99      0.99      0.99      1135
     Class 2       0.99      0.99      0.99      1032
     Class 3       0.99      0.99      0.99      1010
     Class 4       0.98      0.99      0.99       982
     Class 5       0.99      0.98      0.99       892
     Class 6       0.99      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      0.97      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000

Test loss: 0.08802665024995804
Test accuracy: 0.9890000224113464
Correct labels found:  9888
Incorrect labels found:  112
```