Homework 2 - Report

Fabio Vaccaro

9 Dicembre 2020

1 Autoencoder

1.1 Introduction

The goal of this section is to implement, test different optimizer and tune hyper-parameters of an autoencoder. This section and the others refer to the MNIST dataset, chosen to facilitate with the speed and complexity. In fact this is an image dataset much simpler than others.

1.2 Method

The Encoder and the Decoder are structured as displayed in Figure 1 in the Appendix.

1.2.1 Optimizers

I have tried different optimizers to be sure to use the best one. I have tested them in conjunction with the tuning of the hyper-parameters listened in the section 1.2.2. Optimizers tested were: Adagrad, RMSProp, SGD, AdaDelta and Adam. After that, further hyper-parameters tuning was done with the best optimizer found in this phase: Adam.

1.2.2 Hyper-parameters

The hyper-parameters chosen for optimization are:

- *Learning rate.* Learning rate is a very important value to pick to tune properly the model. It varies in regard to the optimizer utilized, so the final tuning was done using Adam. The interval used for optimization was: [0.001, 0.005]. Before using this narrower interval, larger ones were used.
- *Weight decay.* I decided to try also a regularization method using a L2 penalty. This is picked from this interval: [1e-06, 2e-06]. Also this interval was reduced after a first broader search.
- *Batch size.* I tried different batch sizes: 128, 256, 512, 1024. The bigger one seems to perform at least as the other ones so I picked the bigger value to speed up the process.
- *Latent space size.* Maybe the most important hyper-parameter to tune in un autoencoder is the latent space size. I have tried 2, 3, 4, 5 and as expected the higher one performed better as it can "contain more information".

1.3 Results

For hyper-parameters optimization I decided to implement the Ray Tune library[1], integrating it into the train function to let it change the hyper-parameters and get the report of the validation and training loss. I implemented the AxSearch Search Algorithm, a Bayesian Optimization algorithm[2] to tune the hyper-parameters. Initially I implemented a cross validation approach (the notebook contains both the solutions), but the time for learning became k times (in this case k=5), infeasible for a proper optimization, so I decided to continue without cross validation. Results for the hyper-parameters tuning made with 40 epochs are those in the following table. 200 trials were used to find the best ones.

Learning rate	0.0033
Weight decay	1.4770e-06
Batch size	1024
Latent space size	5

2 Denoising autoencoder

2.1 Introduction

In this section I have implemented a denoising autoencoder. This NN, trained with a specific dataset of images, is able to remove noise from the same type of images.

2.2 Method

I have modified the original training function adding a noise_factor parameter. If set to zero the training is done as normal and a standard autoencoder is trained. If this parameter is set to some value different from 0 the denoising autoencoder is trained. The noise_factor controls also the amount of noise added to the image.
To train the denoiser I have given as input the noisy image and encoded it. The loss function is applied to the decoded image and the original image and not to the noisy image. In this way the autoencoder is able to learn to identify and remove noise from the clean image.

[1] Ray Tune website: https://docs.ray.io/en/master/tune/index.html
[2] Ax website: https://ax.dev/

2.3 Results

I have tested the denoising autoencoder with noise_factor=0.3 and 0.6. Result are incredible, the denoiser was able to remove noise and to keep the digit visible. Loss are reported in Table 1 and reconstruction examples are in the appendix.

noise_factor	train loss	test loss
0.3	0.0245	0.02461
0.6	0.03635	0.0332

Table 1: Train and test loss of a denoising autoencoder

As expected we obtain higher loss when we add more noise.

3 Autoencoder Classifier

3.1 Introduction

In this section I will implement a classifier, but differently from the classic approach I will use a autoencoder to create a latent space from which then I will classify images. It is like a pre-training phase.

3.2 Method

I have used the autoencoder trained before to create a latent code. Each image of the training and test set was encoded using only the encoder (the decoder now is not useful anymore). So each image has a tuple of numbers which identifies it. From this tuple I have created a classifier with the following structure:

```
LatentClassifier(
  (fc): Sequential(
    (0): Linear(in_features=32, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

First I have tried with the original autoencoder trained before. I then realized than maybe the latent space size was not sufficient for this task, so I increased it from 5 to 32. This time results was pretty good. I tested both SGD and Adam as optimizers and as always the second one performs better, or at least it is faster to converge.

3.3 Results

These are the results for the classification task:

Configuration	Train loss	Validation accuracy
40 epochs (SGD)	0.2900	0.9248
40 epochs (Adam)	0.0543	0.9823
80 epochs (SGD)	0.2174	0.9432
80 epochs (Adam)	0.0367	0.9836
120 epochs (SGD)	0.1957	0.9478
160 epochs (SGD)	0.1728	0.9535
320 epochs (SGD)	0.1410	0.9613
1000 epochs (SGD)	0.0945	0.9735

As we can see Adam reaches a good accuracy even with only 80 epochs and SGD can come close to it as far as accuracy only with 1000 epochs. Final model will be trained with Adam and 80 epochs.

4 Latent space structure

4.1 Introduction

In this section I have examined the latent space resulting from the training of the autoencoder with a latent space size of 5. I will also investigate a dimensionality reduction method to try to visualize the latent space.

4.2 Method

Using only the encoder (previously trained with the decoder as a "full autoencoder") I have encoded all the images of the test dataset into the latent space. Then I have used the t-distributed stochastic neighbor embedding (t-SNE) which is a machine learning algorithm for visualization to reduce dimensionality to two dimensions to be able to plot the latent space.

To further investigate on the latent space I have plotted the images reconstructed from the latent space. Using only the decoder (previously trained) I have decoded images from random latent codes. It has to be said that numbers are taken randomly but with a min and max value taken from the real dataset to avoid very strange images.

4.3 Results

The latent space of the test dataset is shown in Figure 4.

Images generated by the decoder from random latent code are shown in Figure 5. I expect to get better result with the Variational autoencoder.

5 Variational Autoencoder (VAE)

5.1 Introduction

In this section I will creare a variational autoencoder and test it trying to generate images from the latent space as done before with the standard autoencoder, expecting better results.

5.2 Method

I have used a similar structure to the standard autoencoder, I have just modified the last part of the encoder and the first part of the decoder. The structure is as displayed in Figure 6 in the Appendix.
I have decided to use Adam optimizer as, for prior investigation, it is clear that it is the best performing. An optimization was done with the same hyper-parameters and values intervals.

5.3 Results

Hyper-parameters tuning gave me the following values as the best performing:

Learning rate	0.0052
Weight decay	2.3686e-06
Batch size	1024
Latent space size	5

I have tried the final model (trained with the full training dataset) with some sample images (the same used with the standard autoencoder) and I have analyzed reconstructed images. It is clear from a quick inspection that the VAE confuses the "4" with the "9" and some "2" with "8". Some examples can be seen in Figure 7.

I have then used the VAE encoder to encode all test images into latent codes. From this latent space I have extracted the min and max value for the different dimensions to try to generate meaningful images. Originally I have calculated the min and max globally, not considering different dimensions. I have then discovered that different dimensions could have very different intervals. For this reason I have calculated min and max for every dimension. To have a kind of scientific comparison between the generated images from the standard autoencoder and the VAE (with both general min and max calculated between all dimensions and dimension specific min and max), I have introduced a human test. I have shown 250 images generated from each model for a total of 1000 images. The tester had no idea of the origin of the single image and was asked him to count "good and recognizable digits". The results are the following:

Model	Number of good images
Standard autoendoer (same min/max)	6
Standard autoendoer (different min/max)	41
VAE (same min/max)	94
VAE (different min/max)	100

Samples of generated images from random latent codes are present in the appendix in Figure 8. To see the improvements of the VAE compared to the standard autoencoder see Figure 5 and Figure 8 (those are the autoencoders with different min/max). If you want to appreciate the difference with the versions with same min/max see Figure 9 and Figure 10.

A Appendix

```
Encoder(
  (encoder_cnn): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2))
    (5): ReLU()
  )
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (encoder_lin): Sequential(
    (0): Linear(in_features=288, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=5, bias=True)
  )
)

Decoder(
  (decoder_lin): Sequential(
    (0): Linear(in_features=5, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=288, bias=True)
    (3): ReLU()
  )
  (unflatten): Unflatten(dim=1, unflattened_size=(32, 3, 3))
  (decoder_conv): Sequential(
    (0): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2))
    (1): ReLU()
    (2): ConvTranspose2d(16, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
    (3): ReLU()
    (4): ConvTranspose2d(8, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  )
)
```

Figure 1: Autoencoder structure

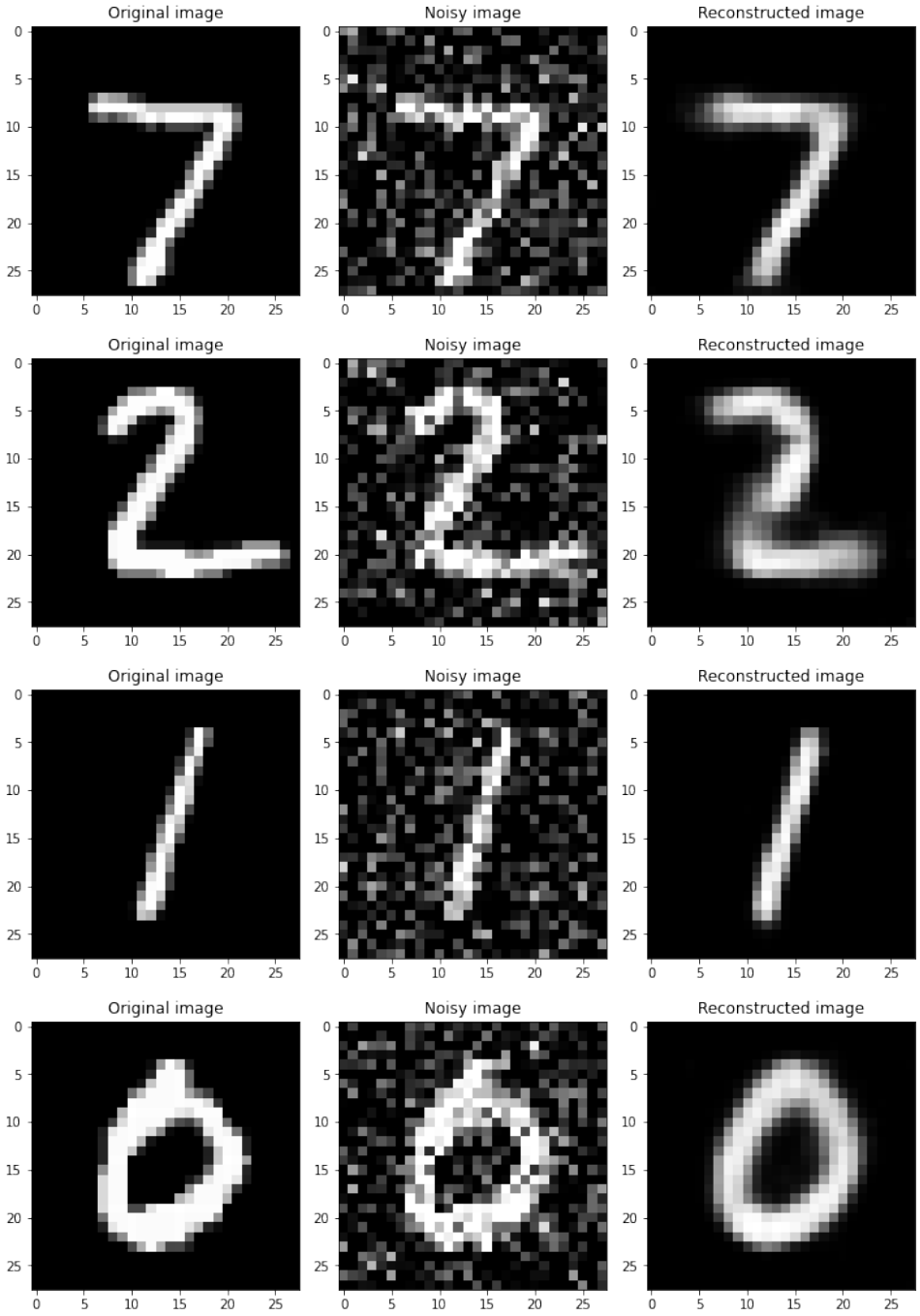

Figure 2: Denoising autoencoder (noise_factor=0.3)

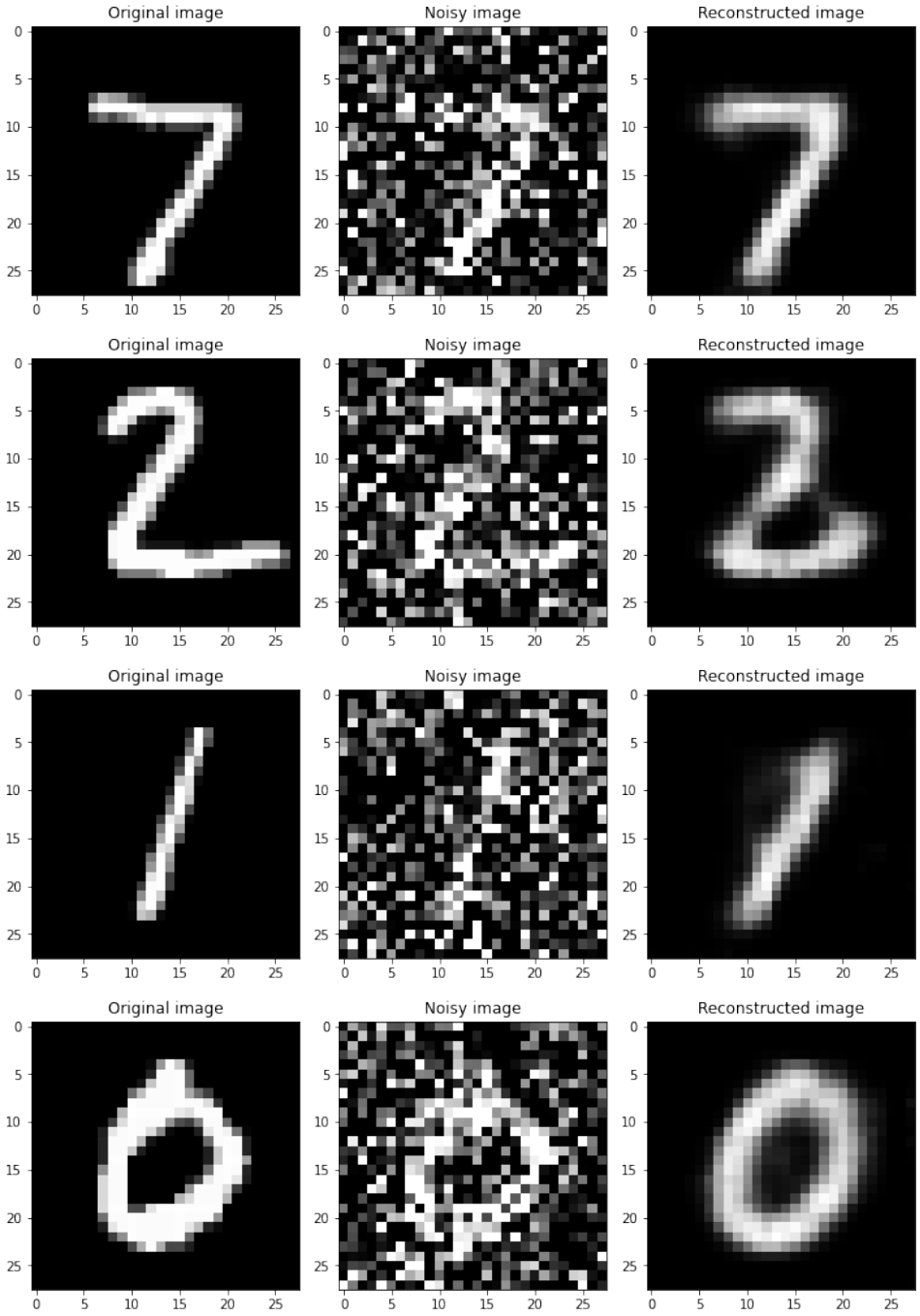

Figure 3: Denoising autoencoder (noise_factor=0.6)

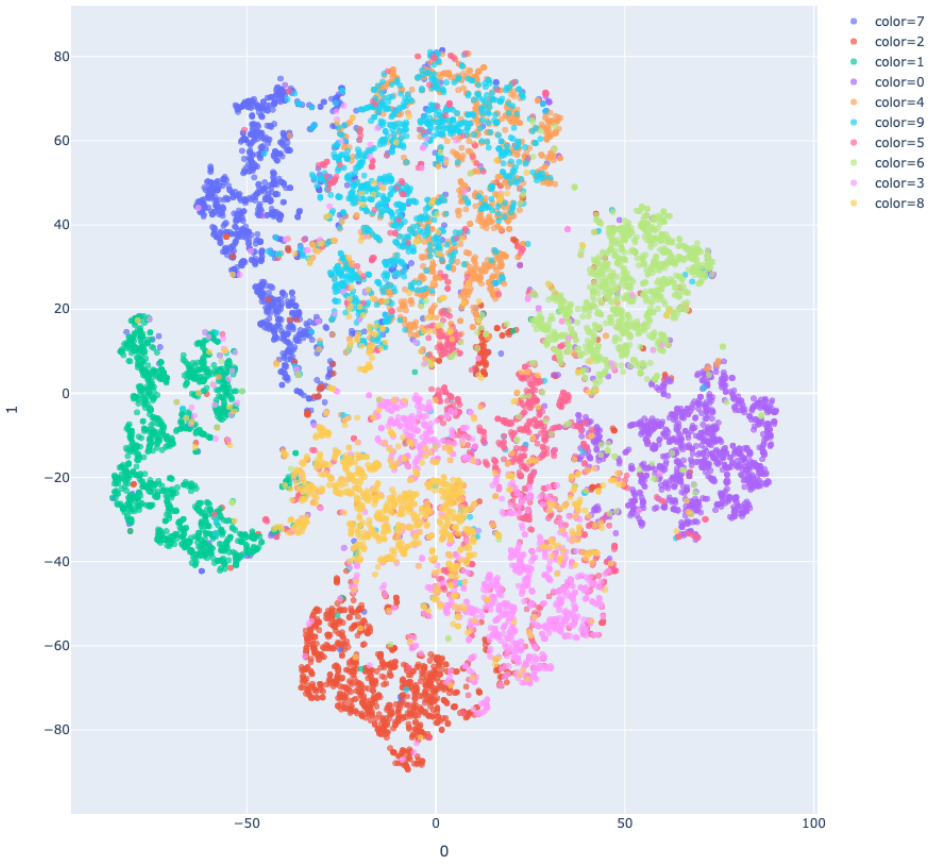

Figure 4: Latent space (from 5 to 2 dimentions)

Figure 5: Samples from latent code (autoencoder) - Different min/max

```
EncoderVAE(
  (encoder): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2))
    (5): ReLU()
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=288, out_features=128, bias=True)
    (8): ReLU()
    (9): Linear(in_features=128, out_features=64, bias=True)
  )
  (mu): Linear(in_features=64, out_features=5, bias=True)
  (logvar): Linear(in_features=64, out_features=5, bias=True)
)

DecoderVAE(
  (decoder): Sequential(
    (0): Linear(in_features=5, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=288, bias=True)
    (5): ReLU()
    (6): Unflatten(dim=1, unflattened_size=(32, 3, 3))
    (7): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2))
    (8): ReLU()
    (9): ConvTranspose2d(16, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
    (10): ReLU()
    (11): ConvTranspose2d(8, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  )
)
```

Figure 6: Variational autoencoder structure

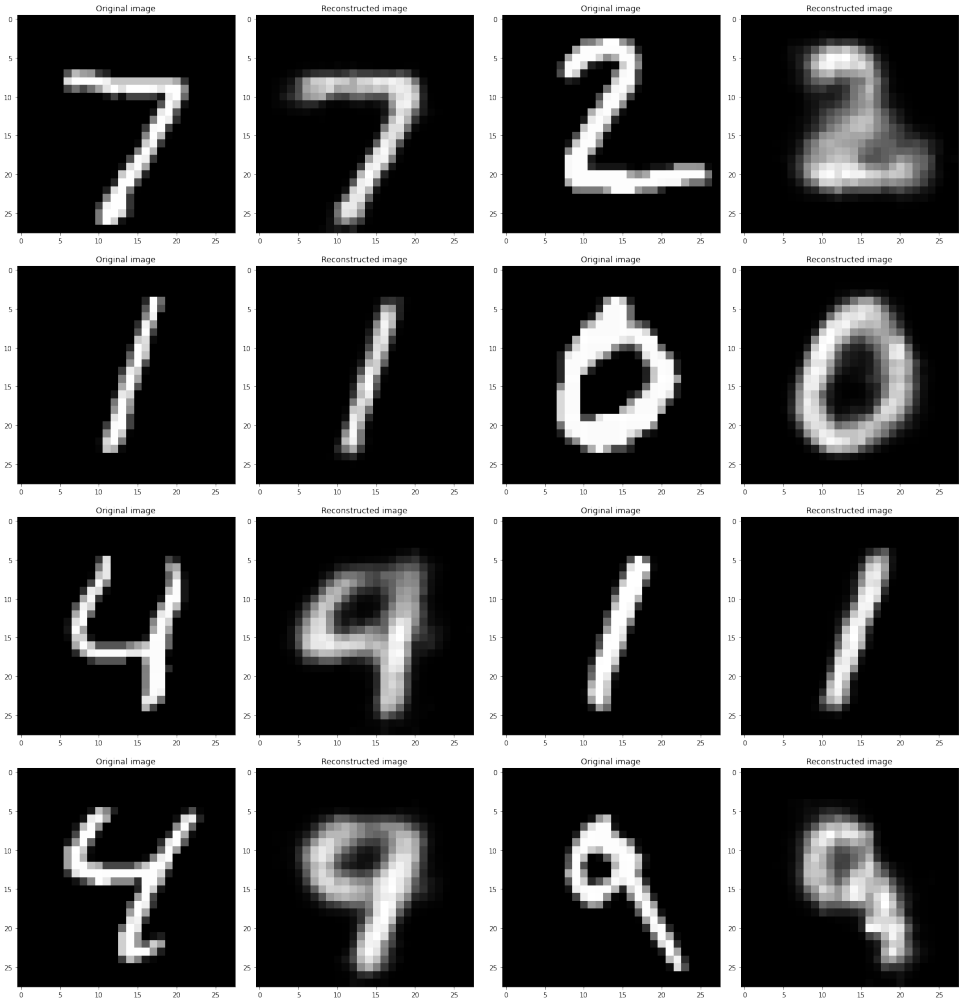

Figure 7: VAE reconstructed images

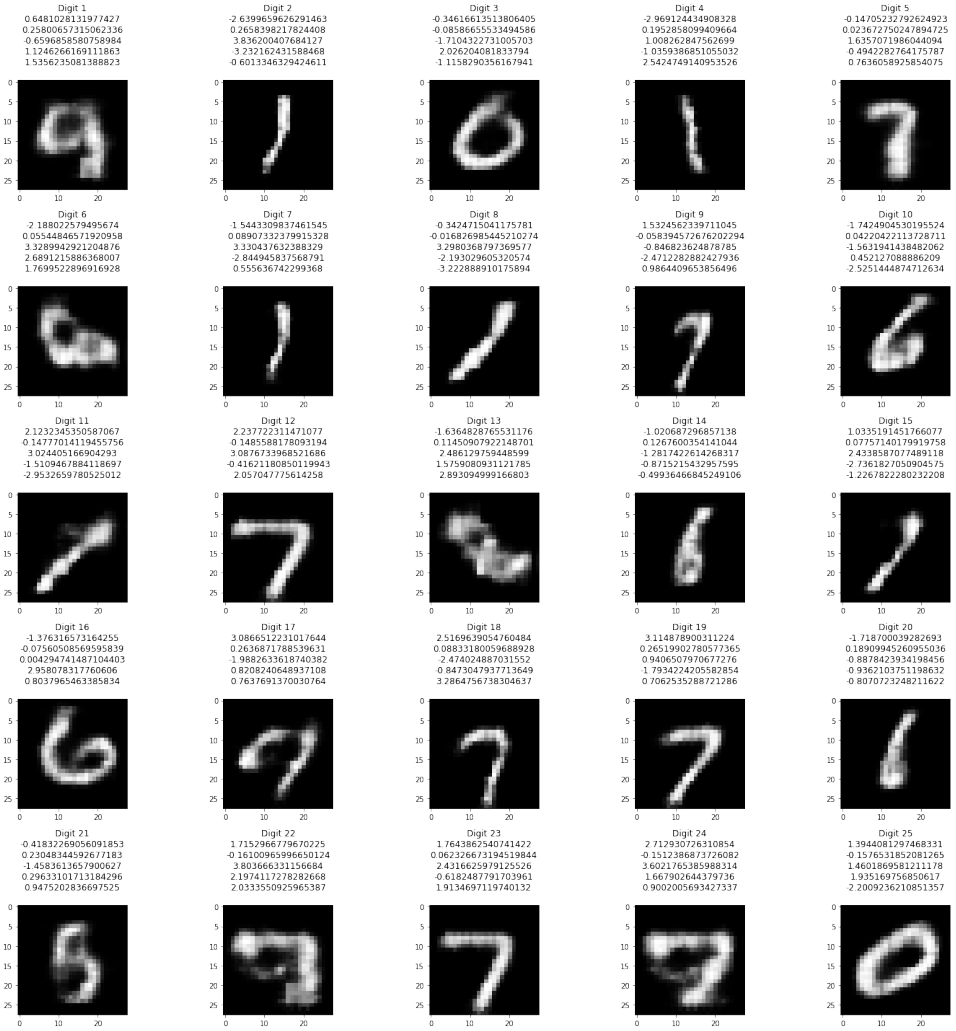

Figure 8: Samples from latent code (VAE) - Different min/max

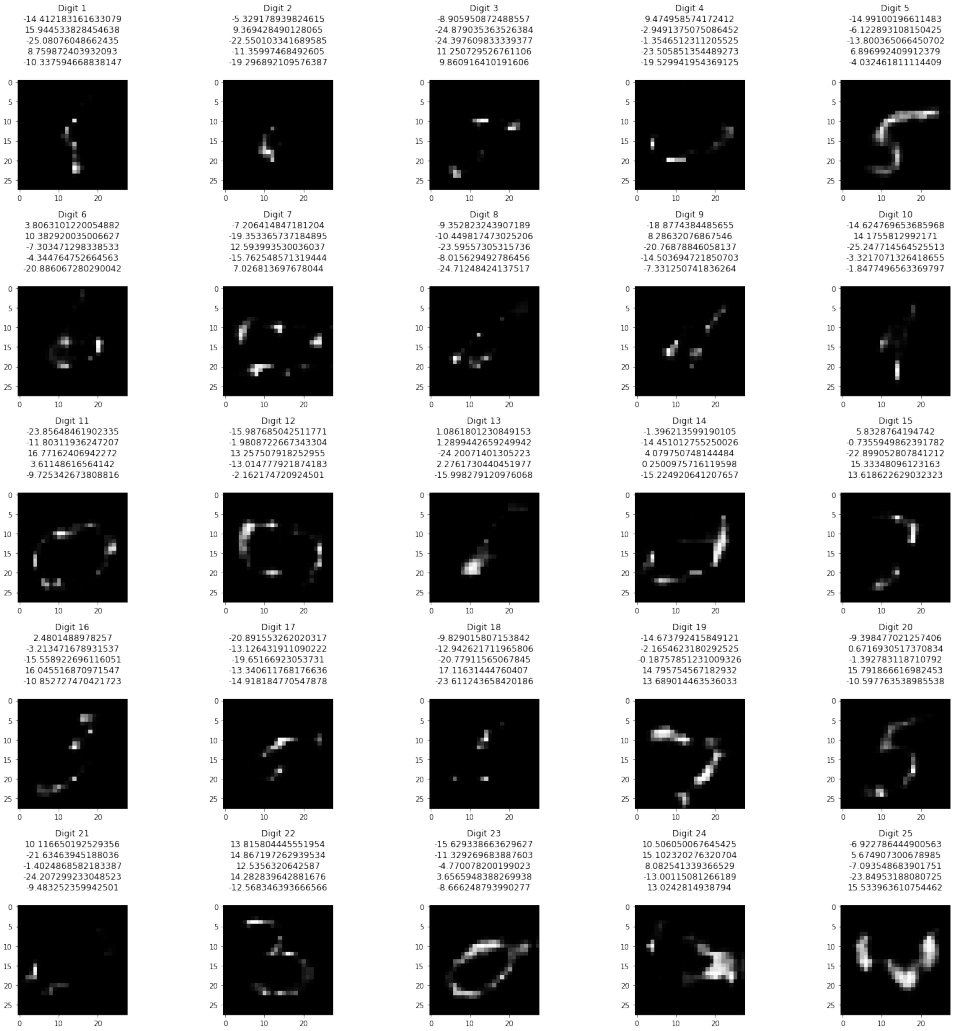

Figure 9: Samples from latent code (VAE) - Same min/max

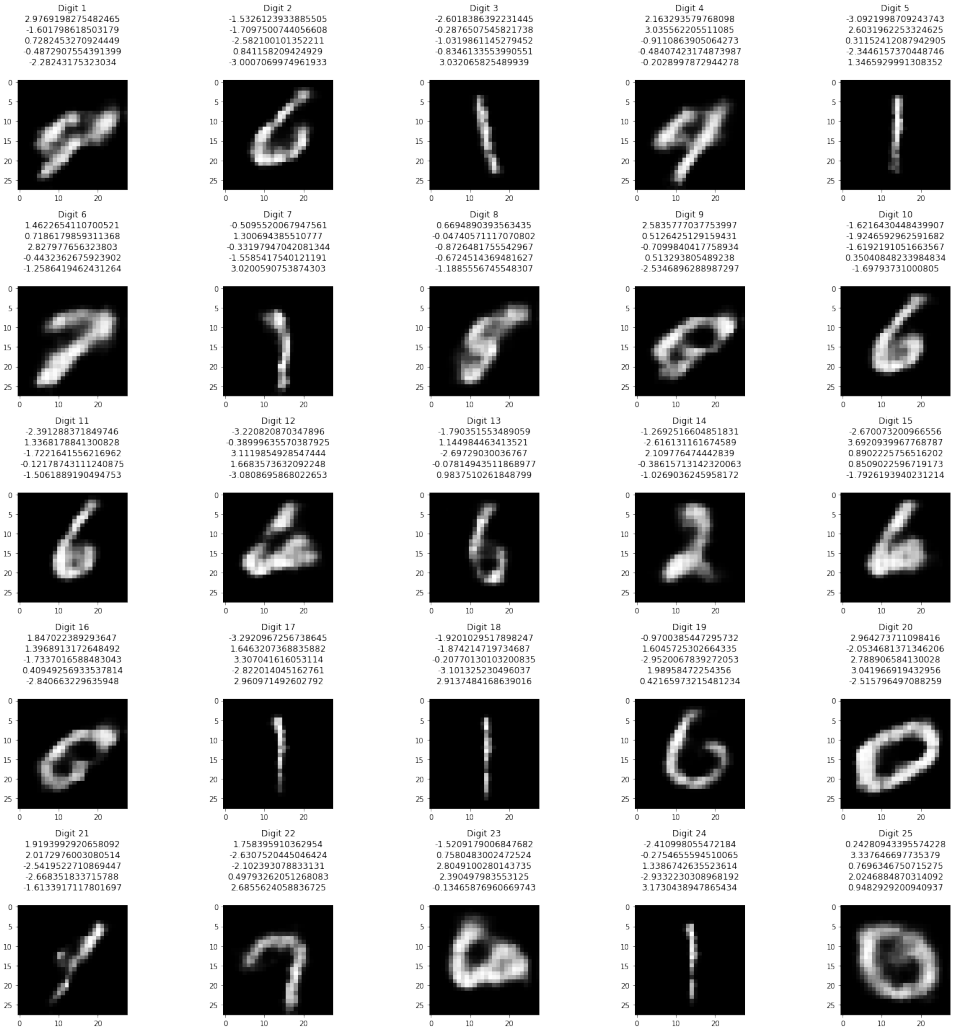

Figure 10: Samples from latent code (VAE) - Same min/max