Neural Networks and Deep Learning - academic year 2020-21

Homework 2: Unsupervised Deep Learning

Luca Dal Zotto - 1236343
luca.dalzotto.1@studenti.unipd.it

Assignment

The aim of this homework is to learn how to implement and test neural network models for solving unsupervised problems. For simplicity and to allow continuity with the kind of data we have seen before, the homework will be based on images of handwritten digits (MNIST). The basic tasks for the homework will require to test and analyze the convolutional autoencoder implemented during the Lab practice. As for the previous homework, we should explore the use of advanced optimizers and regularization methods. Learning hyper-parameters should be tuned using appropriate search procedures, and final accuracy should be evaluated using a cross-validation setup. More advanced tasks will require the exploration of denoising and variational architectures.

1. Introduction

All the assigned tasks of this homework will use the **MNIST** dataset (Figure 1), which has already been presented and discussed in the first homework. However, while in the first homework we focused on some supervised classification techniques, here we will create some **unsupervised** deep learning methods. For simplicity, we can divide the analysis into four main building blocks:

1. implementation and analysis of a standard (convolutional) **autoencoder**;
2. application to a **classification** task;
3. implementation of a **denoising autoencoder**;
4. implementation of a **variational autoencoder**.

Figure 1: Examples from the MNIST dataset.

The aim of the first part is to train an autoencoder able to **reconstruct input images**. For this purpose, I have tried different architectures, whose hyper-parameters have been optimized through a random-search procedure, using a validation set. Different **optimizers** have been considered and various **regularization methods** have been tried. Moreover, an analysis of the trend of the **reconstruction loss** have been performed, along with the plot of some examples of image reconstruction. Then, I explored the **latent space structure**, considering two different techniques (namely, **PCA** and **t-SNE**), and finally, I generated new samples from latent codes.

In the second part, I fine-tuned some of the best models using a **classification task**. In particular, I explored the effect of using encoders with different latent code sizes and also the effect of freezing a different number of layers. The performances have been briefly compared with those reached in the first homework, both in terms of accuracy and learning speed.

For what concerns the implementation of the **denoising autoencoder**, I considered two different types of corruption: the addition of Gaussian noise and the random erasing of small rectangular regions in the input images. The goal is to be able to remove these forms of corruptions restoring the original image.

In the last part, I faced the most challenging task of this homework: the implementation a **variational autoencoder**.

The structure of this report is the following: in the next section, I will described the methods used for the implementation, such as the learning architectures, the hyper-parameter choices and the motivations behind them. Then, in the third section, I will show the main results of the analysis, along with some relevant figures and comments. Both this two sections have been divided into 4 sub-sections, reflecting the task division described before.

Finally, I also included an Appendix with additional plots. However, for the complete list of results and for more implementation details, the reader is invited to consult the notebook “LDZ_nndl_2020__homework_2.ipynb”.

2. Methods

2.1. Task 1 - Autoencoder implementation

Before starting implementing AE architectures, I extracted from the training set a subset of 10,000 samples which have been used as validation set. I decided this standard division instead of a cross-validation approach to keep the computational cost low, since I preferred to exploit the available resources to try different techniques and play with more hyper-parameter configurations. Moreover, a complete implementation of a (nested) cross-validation procedure has been shown in the first homework.

Therefore, I obtained a training, a validation and a test set of size 50,000, 10,000 and 10,000 respectively, and to each of them I applied the transformation ToTensor. Then, I created a dataloader object setting the parameter batch size equal to 250 and enabling data shuffling only for the training set.

Then, I defined the **encoder** architecture. Since the input is an image, I chose some convolutional layers as automatic feature extractors, followed by some fully connected layers. Since the dataset at handle is quite simple, I have not used very deep architectures, limiting the total number of layers. With the convolutions, I tried different numbers of output channels, kernel sizes, strides and padding sizes. Note that each convolutional layer may have different hyper-parameters. I tried different code sizes (from 1 to 32), and I have put a particular emphasis in the comparison of the different reconstruction ability achieved with them. The activation functions considered have been the Sigmoid, the LeakyReLU and the ELU. Note that for the **decoder** part, I considered a **symmetric** architecture with respect to the encoder.

As **optimizers**, I tried SGD, RMSprop and Adam. For the first one, I considered a basic implementation without momentum. The advantage of the other two consists in **adapting the learning rate** individually for each weight depending on the training circumstances. In particular, RMSprop considers a running average of past values of a certain weight to modify the learning rate of that weight, while Adam optimizer takes into consideration also an estimate of the second moments of the gradients.

For what concerns the **regularization techniques**, I considered different values of the optimizers' parameter **weight decay** and I also tried some **batch normalization** layers. The former can be shown to be equivalent (under some hypothesis) to the L2 regularization, which consists in imposing a norm penalty to the parameters. The latter essentially normalizes the hidden activations on each mini-batch. To avoid possible interference between these approaches, I always tried at most one of them at the time.

Since we wanted to measure the reconstruction error, I opted for the **Mean Squared Error loss function**.

Regarding the training procedure, I manually implemented an **early stopping** routine which terminates the training loop if the validation loss does not increase for more than a certain number of epochs (**patience** parameter). However, I also set a maximum number of iterations. Another observation is that I have used the validation loss as criterion for selecting the best hyper-parameters, but I also plotted the **learning curves** and an example of reconstruction error to have additional feedback about the learning progress.

The different hyper-parameter combinations have been tested with a random-search approach, where each value was sampled from a list of candidates. To give more details, I made three main searches: in the first one, I started from a wide range of possible values for each hyper-parameter, while in the other ones, I focused only on the regions that apparently led to the best results in the previous search. Finally, I considered a list of possible dimensions for the latent code (1, 2, 4, 8, 16, 32) and for each of them I trained 10 different autoencoders, in order to make a reasonable comparison of the reconstruction ability based on the code size.

Then, I selected the best AE with code size 8 and the best with code size 32. For brevity, in this report I will call them **AE8** and **AE32**. For each of them I made a deeper analysis performing the following steps:

1. evaluation using the test set;
2. plot of some examples of reconstructions;
3. exploration of the latent space;
4. generation of samples from the latent space.

2.2. Task 2 - Classification

In this part, we were asked to fine-tune our models using a **supervised classification task**. The main idea is to consider the trained encoder as a feature extractor, and use the latent code as input for a classifier. In this homework, this role has been played by some additional dense layers, but in principle, I could have used any other classification algorithm, such as Random Forest or SVM.

For this task, I decided to load the encoder part of **AE32**. After having freezed all its layers to prevent weights update, I just stacked a fully connected layer with 32 input neurons and 10 output neurons. Clearly, weights update was enabled for this layer. Then I run a training loop very similar to the one used in the first homework. To be precise, I trained the new models with different learning rates and evaluated them in the validation set by simply considering the confusion matrix and computing the classification accuracy.

This entire procedure has been repeated changing the number of additional fully connected layers, and also enabling the update of some of the last encoder's layers. Then, I merged training and validation set, trained the best model in this set, and finally evaluated it in the test set.

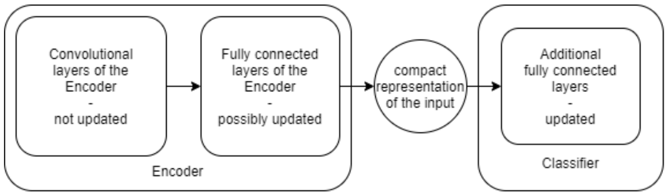

Convolutional layers of the Encoder - not updated
Fully connected layers of the Encoder - possibly updated
compact representation of the input
Additional fully connected layers - updated
Encoder
Classifier

Figure 2: Schematic representation of the classification task.

2.3. Task 3 - Denoisy autoencoder

A **denoisy AE** is like a standard AE with the only difference that the input image fed to the encoder has been previously **corrupted**. The final goal is to **restore the original picture** (which is used as target output), removing the noise. This can be seen as a form of **regularization**. Besides, the denoisy task may also be learnt per se: indeed, if we are interesting in restoring some noisy images, we can artificially apply a similar kind of noise in another set of uncorrupted images and train a denoisy AE on them. Once trained, we can apply it to the images we wanted to clean.

In this homework, I considered two different types of noise. The first one was the **Gaussian noise**, so each pixel has been randomly perturbed. The second one was a bit more challenging: in each image, a random rectangular portion has been selected and its pixels erased (changing their color into black, the color of the background). In the results sections, it is possible to see some clarifying examples.

2.4. Task 4 - Variational autoencoder

One of the limit of the AEs is that the latent space can be extremely **irregular**. To limit this issue, with **Variational AEs**, the encoder returns no longer a single point in the latent space, but a distribution.

A possible choice, which is also the one I used in this homework, is to train the encoder to estimate the mean and the covariance matrix' diagonal (assuming the latent variables independent) of a multivariate Normal distribution. In order to force the latent distribution to be as close as possible to the Gaussian one, we need to add a term to the loss function, represented by the **Kullback–Leibler divergence**. Note that, since we are considering normal distributions, it can be expressed in an equivalent form which can be implemented quite easily.

Another crucial aspect to keep in mind for the implementation is the **re-parameterization** of the latent code, which is required in order to allow the use of the back-propagation algorithm. This is achieved by the element-wise multiplication of the vector representing the diagonal of the covariance matrix with a random vector of the same size sampled from a standard normal distribution, and summing the result with the mean vector.

Note that in this context, when I will speak about a d-dimensional code, it means that the mean vector is d-dimensional and the covariance matrix has size dxd.

3. Results

3.1. Task 1 - Autoencoder implementation

In Table 1, I report the list of values tried for the most relevant hyper-parameters.

hyper-parameter	**values**
convolutional layers	2, 3, 4
output channels conv layers	8, 16, 32, 64, 128
kernel size	3, 5
stride	1, 2
padding	0, 1
fully connected layers	2, 3
hidden neurons	32, 64, 128, 256, 512, 1024
code size	1, 2, 4, 8, 16, 32
activation	Sigmoid, LeakyReLU, ELU
optimizers	SGD, RMSprop, Adam
learning rate	1e-5, 3e-4,1e-4, 3e-3, 1e-3, 1e-2
weight deacy	1e-6, 1e-5, 1e-4, 1e-3
max number of epochs	16, 32, 50
patience	4, 5, 7

Table 1: Hyper-parameters tested.

As anticipated in the previous section, the search procedure was articulated in different stages, where I gradually explored more restricted regions of the hyper-parameters space. The whole procedure and the results obtained with all configurations can be consulted in the notebook. In this report, I will focus only on the most interesting results, starting from a comparison of the models depending on the latent code size. In Figure 3, I reported the reconstruction loss obtained with the best model for each code size considered. In the Appendix, I also reported a picture where I computed the average reconstruction loss over the five best configurations, for each code size, to have a more reliable indication.

As expected, the higher the code dimension, the lower the reconstruction loss, since we can store more information about the input. To have an idea of the quality of the results, values below 0.04 are, in this specific case, already satisfactory. Therefore, we can conclude that

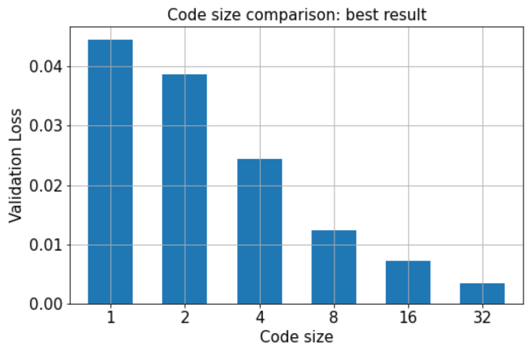


Figure 3: Code size comparison.

the configuration with code size 1 was over-ambitious: essentially, we are storing one (small) picture in a single number! On the other hand, the reconstruction obtained with code size larger or equal to 8 were very accurate. For this reason, I decided to analyzed more in depth the best AE with code size 8 (**AE8**) and that with code size 32 (**AE32**).

First of all, I evaluated these two autoencoder on the test set. **AE8** provided a test loss of 0.01055, while **AE32** a test loss of 0.0044. In Figure 4, I reported some original images (first and third column) along with the corresponding reconstructions (second and fourth column) for **AE8**.

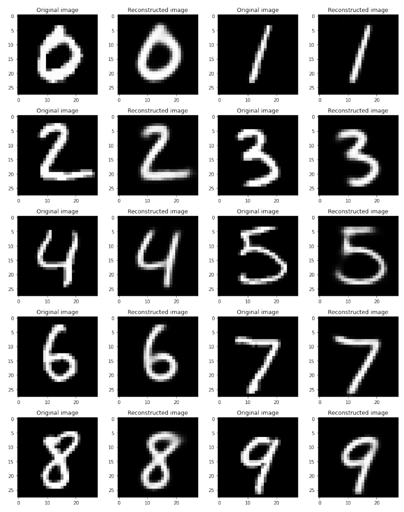


Figure 4: Reconstruction examples (code size = 8).

Then, I analyzed the **latent space**. In order to visualize it, I considered two dimensionality reduction methods: **Principal Component Analysis** (**PCA**) and **t-distributed Stochastic Neighbor Embedding** (**t-SNE**), setting the number of components equal to 2 or 3. With PCA, the different digits were already clustered in different sets, but they were a bit overlapping. On the other hand, with t-SNE all clusters were well separated, with some exceptions, as we can see in Figure 5, where each color corresponds to a different digit.

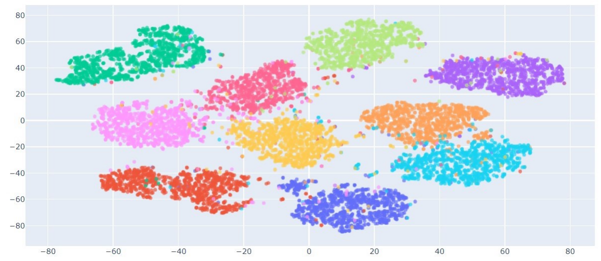


Figure 5: 2-dim embedding of the latent space (AE8).

Other results and plots can be found in the appendix and in the notebook.

To conclude this first part, I **generated some samples from latent codes**. To do so, I considered a 2-dimensional grid of values, used the inverse_transform method of PCA to obtain 8-dimensional (or 32-dim) vectors, and gave them as input to the decoders. Then, I plotted in a 2-dim grid the generated images. In Figure 6, we can see the results obtained with code size = 8.

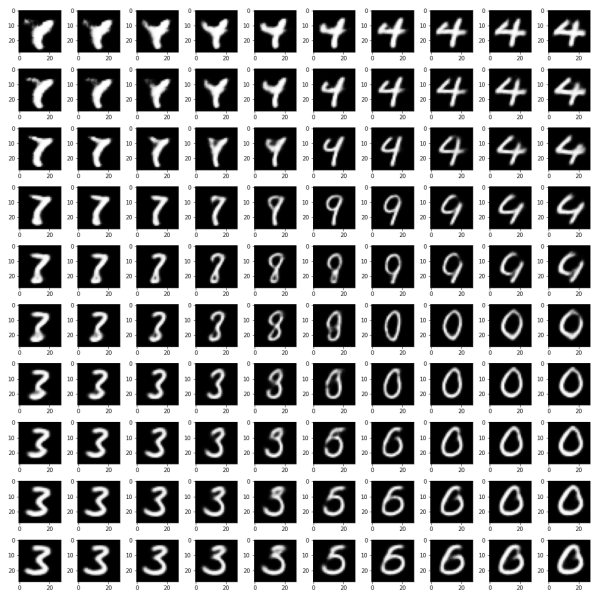


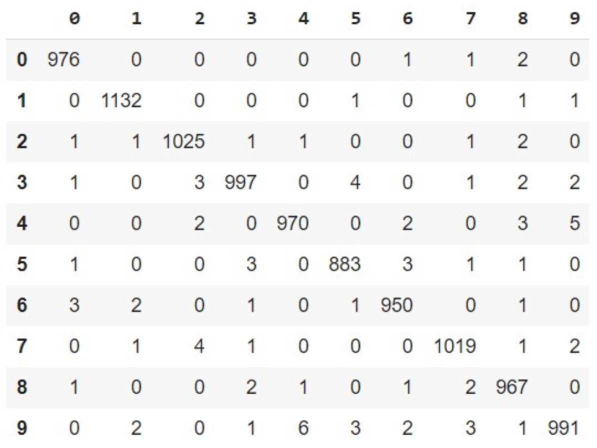
Figure 6: Generation of images from the latent space (AE8).

From this plot, we can make two observations: firstly, some images are meaningless (e.g., upper left corner) and in some regions (e.g., second row from below), close points produce different results. These are two of the motivations behind the implementation of variational AEs, an architecture which tries to solve the problem of latent space irregularity of standard AEs.

3.2. Task 2 - Classification

In the second part, I fine-tuned the AEs using a classification task. In my first attempt, I have prevented the update of the encoder weights and added a single layer connecting the 32-dim code with the 10-dim output. Since the validation accuracy was quite low (92.2%), I increased the number of additional fully connected layers to 2. In this way, the validation accuracy reached 95%.

Better performances were obtained enabling the update of the last, or the last two encoder's layers. In these cases, I reached 97/98% of accuracy. However, the best model was the one with 3 additional fully connected layers and with only the weights of the convolutional part of the encoder freezed. After having re-trained it in the the merged train-validation set, I conduced a final evaluation in the test set. The accuracy reached was finally in line with those seen in the first homework (99.10%). For completeness, in Figure 7, I report the confusion matrix.



	0	1	2	3	4	5	6	7	8	9
0	976	0	0	0	0	0	1	1	2	0
1	0	1132	0	0	0	1	0	0	1	1
2	1	1	1025	1	1	0	0	1	2	0
3	1	0	3	997	0	4	0	1	2	2
4	0	0	2	0	970	0	2	0	3	5
5	1	0	0	3	0	883	3	1	1	0
6	3	2	0	1	0	1	950	0	1	0
7	0	1	4	1	0	0	0	1019	1	2
8	1	0	0	2	1	0	1	2	967	0
9	0	2	0	1	6	3	2	3	1	991

Figure 7: Confusion matrix.

Regarding the training time, it is difficult to make a comparison with the first homework giving some precise numbers, since I used Google Colab to train my models, and the hardware available was not always the same. However, I can say that in general, the time required for each training epoch was smaller with the models tested in this homework, especially when I prevented the update of a large part of the network.

3.3. Task 3 - Denoisy autoencoder

For this part, I chose to neglect the hyper-parameter tuning phase, in order to focus only on the denoisy task. Indeed, I considered the **AE32** architecture and kept the same values seen in the first part of the homework for the other hyper-parameters. For this reason, I directly trained the model in the full training-validation set and evaluated it in the test set.

As first denoisy task, I simply added some noise following a standard normal distribution to each pixel of the input images. Actually, in my implementation, the level of corruption could be adjusted modifying a scale factor, which I set equal to 0.1. In Figure 8, I reported some examples of images before being corrupted, after the corruption and after the reconstruction made by the denoisy AE.

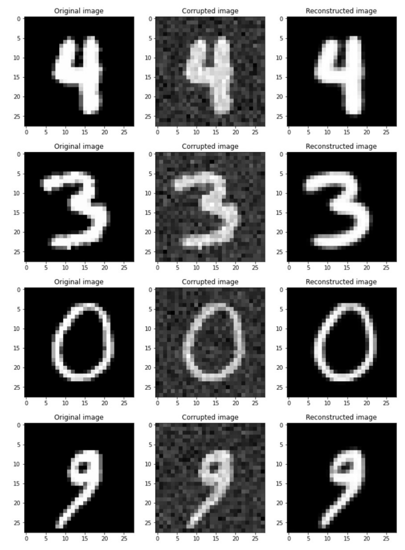


Figure 8: Denoisy AE examples - Gaussian noise.

This task has been solved extremely well, as certificated by the very low value of the test loss: 0.0049. Recall that the test loss of the standard AE with code size 32 was very similar (0.0044).

The second type of corruption was a **random erasing** of a rectangular area of the image. To implement it, I used a transformation described in the Pytorch documentation called RandomErasing. I modified its parameter in the following way:

- the probability parameter p was set equal to 1, meaning that all images would have received this transformation;
- the maximum proportion of erased area against the image was set equal to 15%;
- the range of aspect ratio of the erased area was set equal to (0.7, 1.7).

In addition to them, I also leaved the erasing value equal to zero, meaning that to erase a pixel, black color has to be used, and the inplace option equal to False, since I need to keep the original image to use it as target. In this case, the test loss was a bit higher but still very good (0.0087). There are sometimes some problems occurring when the erased area coincides with a discriminating region of certain digits: for example, in Figure 9, a 4 has been corrupted in a way that even a person would have some doubts in determining whether it was a 4 or 9, and in fact, it has been reconstructed like something in the middle between these two digits.

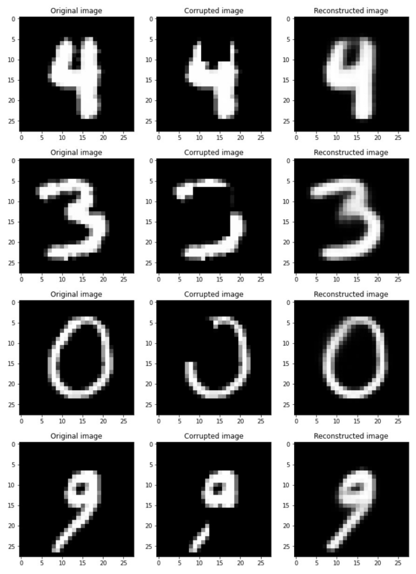
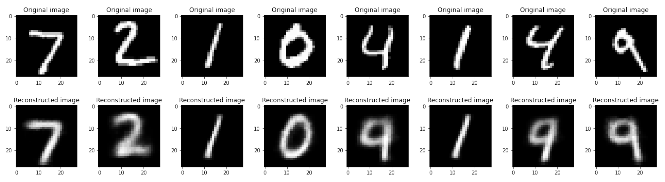


Figure 9: Denoisy AE examples - Random erase.

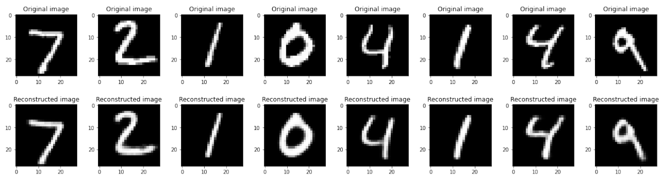
3.4. Task 4 - Variational autoencoder

For this last task, I trained 2 architectures with a different code size (2 and 16). The most important steps have already been described in the previous section, and the complete implementation can be found in the notebook.

In Figure 10, I plotted some examples of reconstruction obtained with the two different VAEs.



(a) *Code size = 2.*



(b) *Code size = 16.*

Figure 10: Examples of original images (first row) and their reconstructions (second row) for the two VAEs.

What we can see is that in the first case, the reconstruction is a bit noisy, and some digits are wrongly reconstructed. On the other hand, when using a code of size 16, the reconstruction ability increases, obtaining very satisfactory results.

In addition to this, I explored the latent code, plotting the re-parameterized vectors relative to the test set. Moreover, I also generated some images starting from the latent space. Here I report only the plots of the 2-dimensional code VAE.

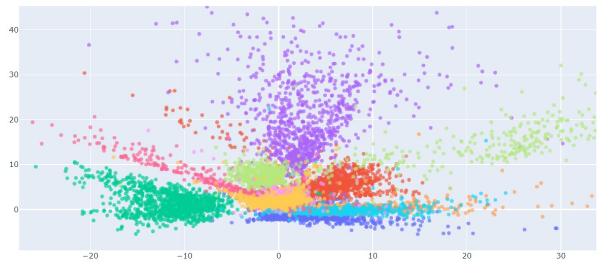


Figure 11: Latent space representation.

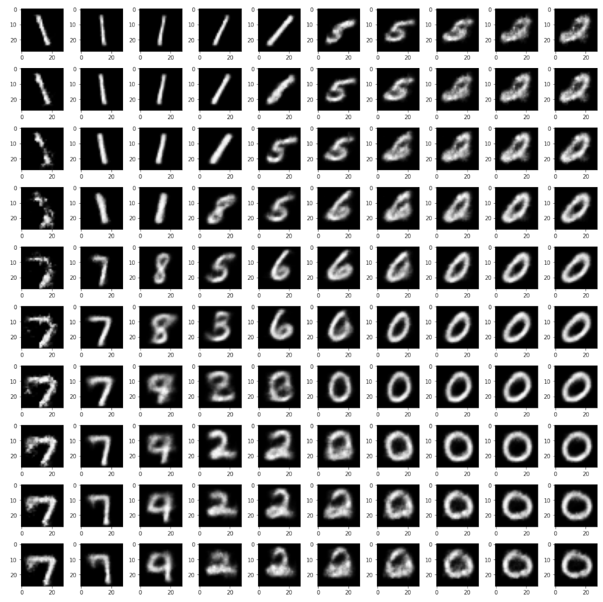


Figure 12: Samples generated from the latent space (code size = 2).

What we can see is that the clusters of the different digits have significant overlaps, and also occupy better the latent space. For what concerns the generation of images instead, we can notice that the digits vary smoothly and also it seems that the variables which changes in the x-axis encodes an high level feature regarding the orientation of the digit in the picture.

Appendix

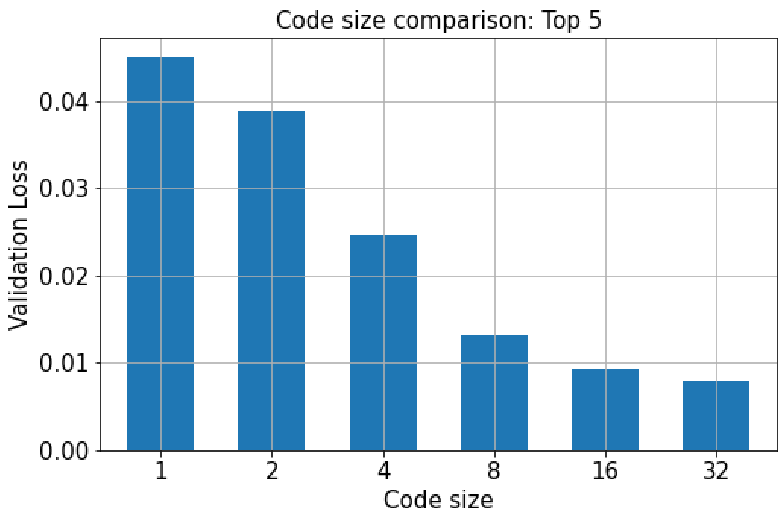


Figure 13: Code size comparison (average validation loss of the best 5 models for each code size).

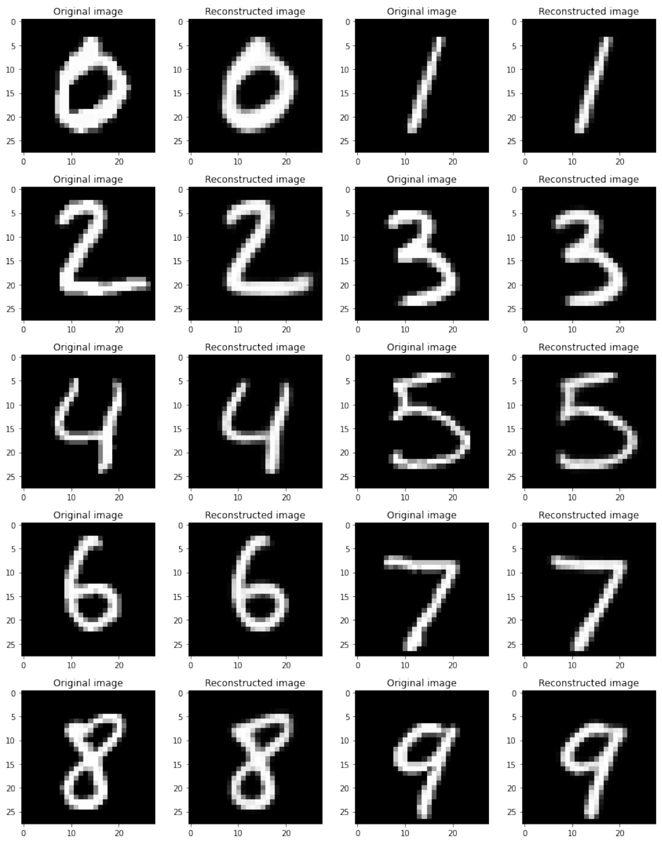


Figure 14: Reconstruction examples (AE32): original images in columns 1 and 3, reconstructions in columns 2 and 4.

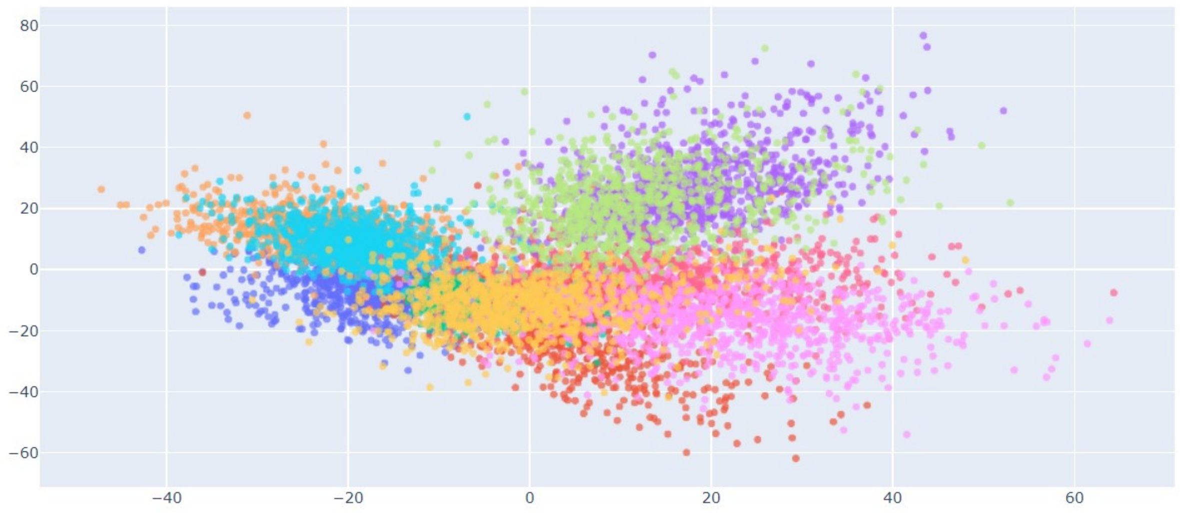


Figure 15: 2-dim embedding of the latent space (AE8) with PCA.

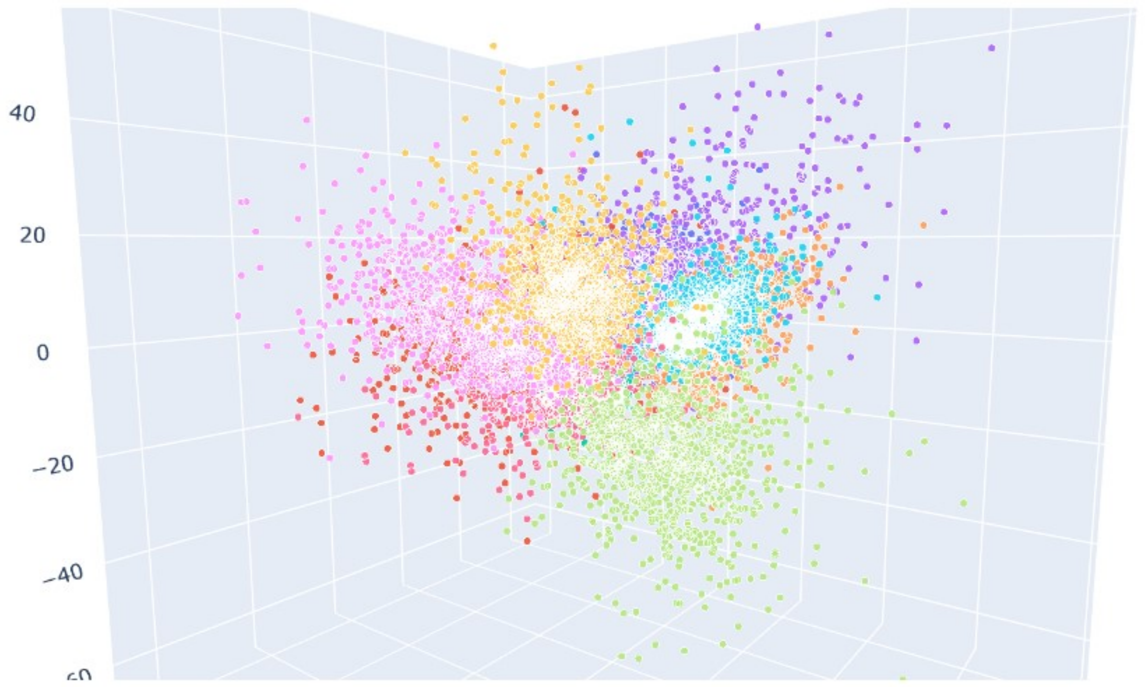


Figure 16: 3-dim embedding of the latent space (AE8) with PCA.

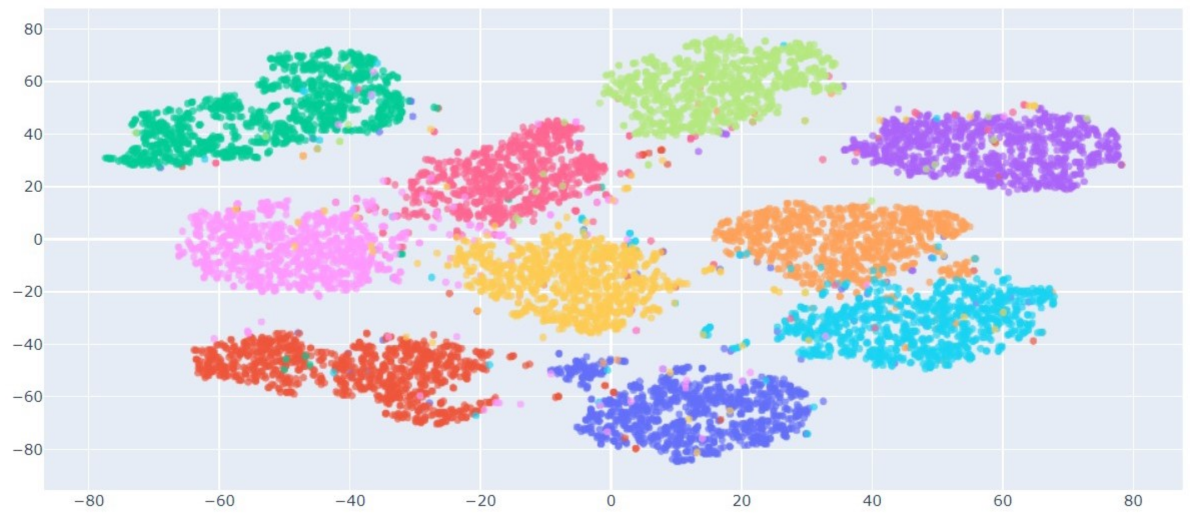


Figure 17: 2-dim embedding of the latent space (AE8) with t-SNE.

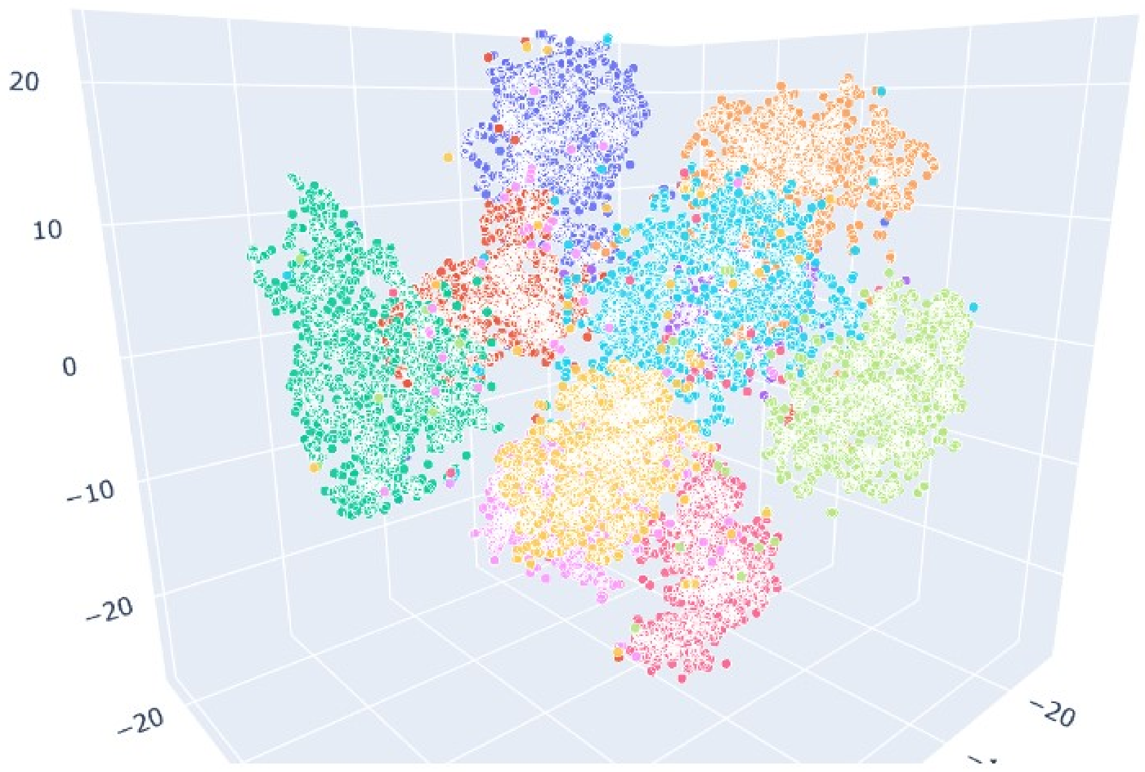


Figure 18: 3-dim embedding of the latent space (AE8) with t-SNE.

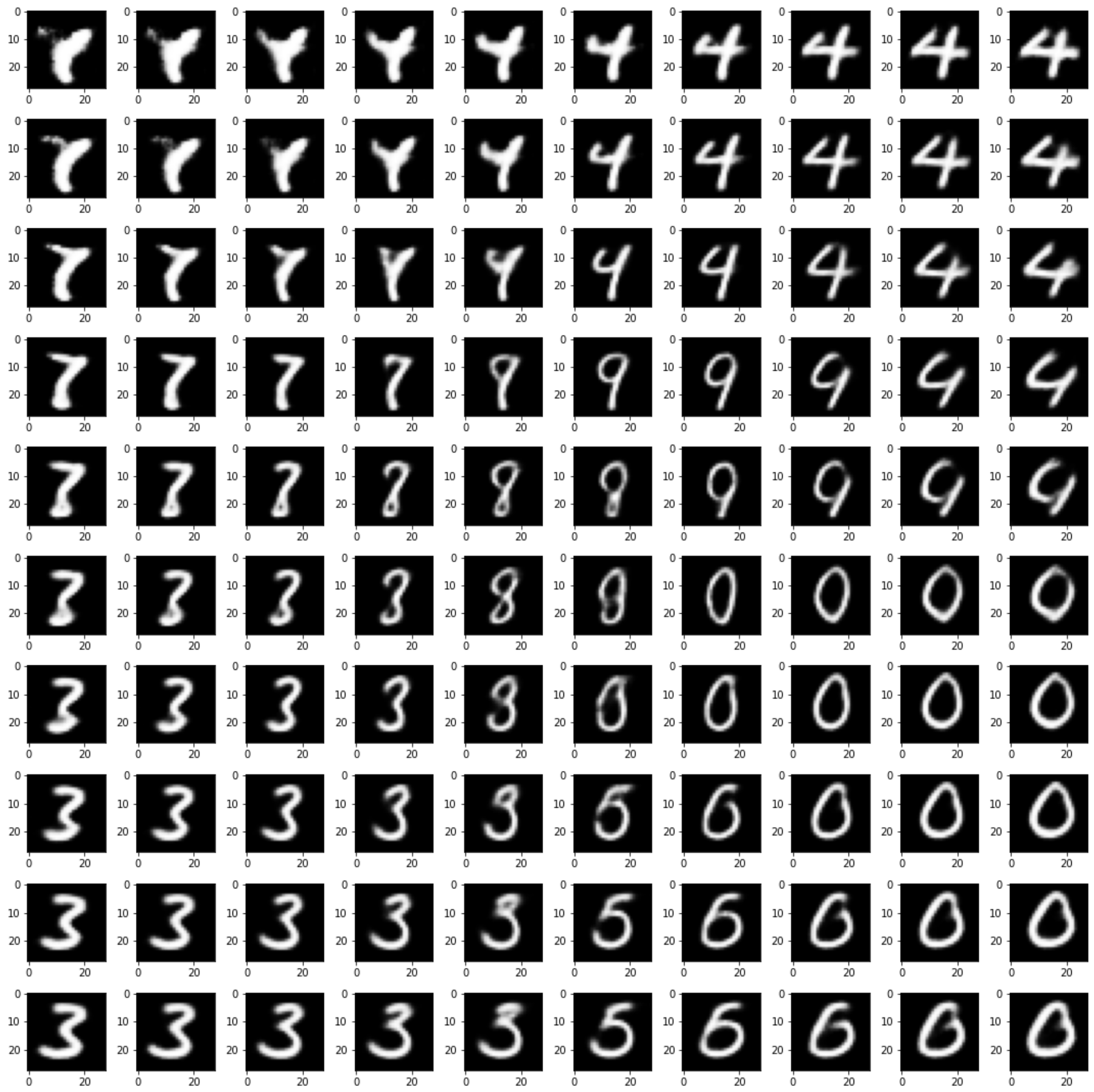


Figure 19: Generation of images from the latent space (AE8).

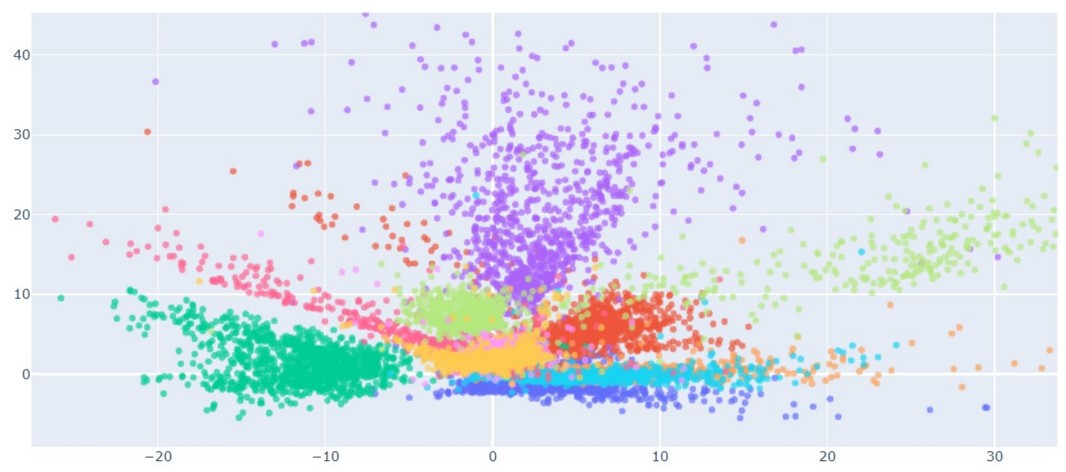


Figure 20: Variational AE with code size 2: latent space representation.

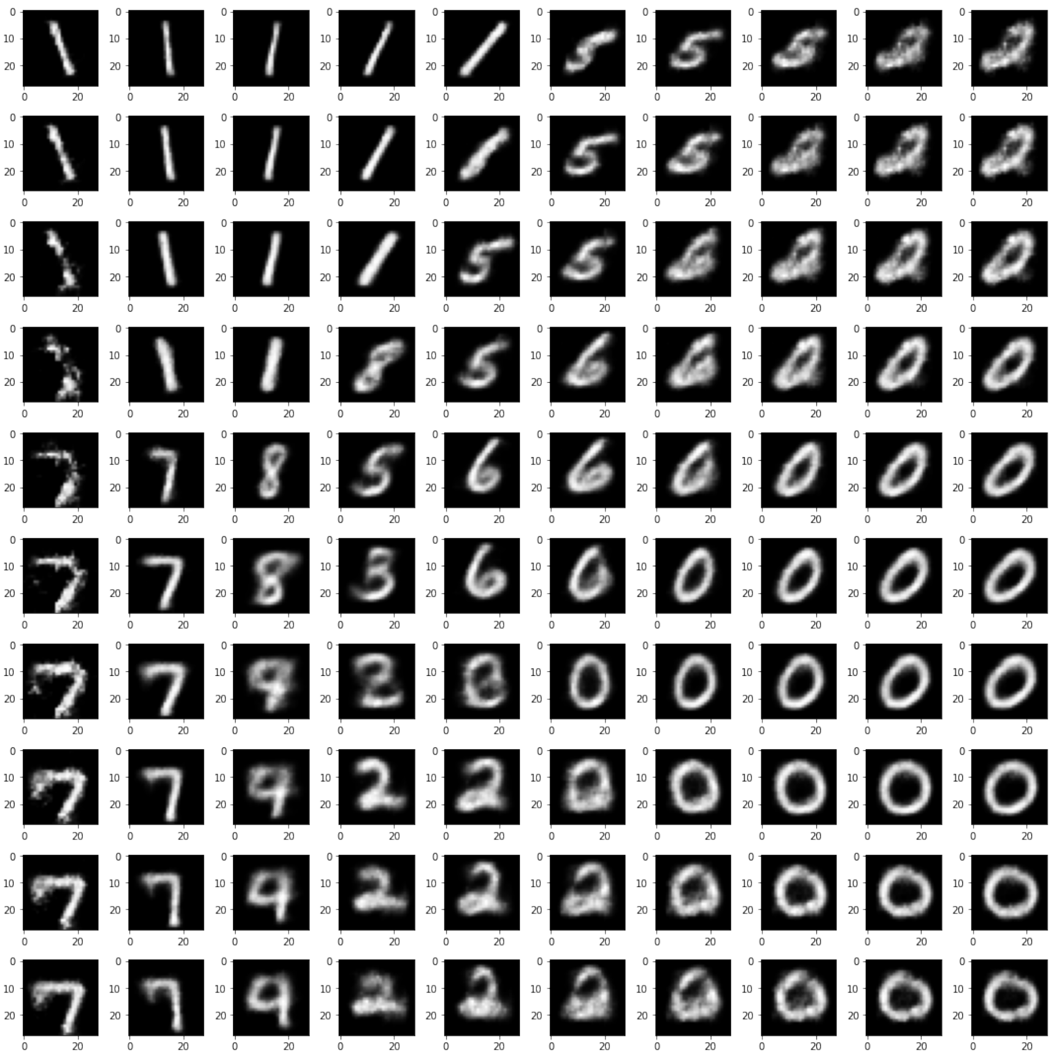


Figure 21: Variational AE with code size 2: samples generated from latent codes.