Neural Networks and Deep Learning
Homework 2, Unsupervised models

Daniele Foscarin

2 February 2021

1 Introduction

In this homework we dealt with models that solve unsupervised problems, so we implemented and trained a convolutional autoencoder, a denoising convolutional autoencoder, we applied those model in a classification task and we developed a variational autoencoder as an example of a generative model. We performed the trainings on Google Colab using extensively the Optuna Framework in order to search for the best parameters and evaluate the performances of the different models.
We have implemented four tasks that are

- Implementation of an autoencoder
- Implementation of a denoising autoencoder
- Applying the autoencoders in a classification task
- Implementing a generative model, variational autoencoder.

We organize this report in order to have for each task a section with the description of the method, and the discussion of the results.

1.1 Dataset "notMNIST"

Instead of using the MNIST dataset we based this homework on the notMNIST dataset[1]. It is a dataset composed by the letter of the alphabet from A to J, so we have ten classes of graylevel images with resolution 28×28 pixel. This dataset has been created to be an alternative to MNIST, keeping the same number of classes and samples size, but it provides a much bigger variance between the samples. For this reason the encoding/decoding or the classification of this dataset is more difficult than for MNIST. We can see an examples of some randomly chosen samples in Figure 1 and it is evident how there is a big variance between samples from the same class.
There are various version of this dataset available that differs on the number of samples, we chose the small version that contains 18724 samples. The repository containing the dataset is cloned into Colab and the samples are read and normalized, then we define a dataset class and we split the dataset so the training set contains 11234 samples and the validation and the training set contain 3745 samples.

[1]Dataset from Kaggle https://www.kaggle.com/quanbk/notmnist

2 Autoencoder

2.1 Method

At first we want to implement a convolutional autoencoder by defining separately the encoder and the decoder class. The encoder is defined with three convolutional layers and two fully connected layers. In the convolutional layers we used kernels of size 3×3 and stride equal to 2 in order to halve the size of the tensor, in the last convolutional layer we use padding equal to zero to further reduce the size. So the dimensionality of the input samples is 28×28, then it became 14×14 and finally 3×3. After this the tensor is taken by the linear layers and the output has the size of the latent space. Every layer but the last one has activation function ReLU.
The decoder has a symmetrical architecture with respect to the encoder, it takes in input a batch of samples with the size of the latent space and produce a batch of samples of size 28×28. The last layer of the decoder is activated by a sigmoid function, since we want to have the output samples with values in the [0,1] range.
The number of channels in every convolutional layers as well as the number of neurons in the linear layers and the dimension of the encoded space is given in input at the classes constructor.

We performed an optimized grid search for the best parameters using Optuna, the following parameters were suggested by Optuna during the different trials:

- encoded space dimension, integer in [2,10],
- batch size, integer [200,1000],
- optimizer, chosen between Adam and RMSprop
- learning rate, chosen with logarithmic uniform distribution in $[1e-4, 1e-1]$,
- conv1 channels, chosen from [32,64,128],
- conv2 channels, chosen from [32,64,128],
- conv3 channels, chosen from [32,64,128],
- fully connected neurons, chosen from [32,64,128].

The loss function used is MSELoss. We let Optuna perform 100 trials, for every trial we used 50 epochs. We also implemented the pruning of bad performing trials. To do so, we report the value of the validation loss for every epoch inside the *objective* function, so Optuna can raise the TrialPruned exception. We used the MedianPruner with 5 startup trials and 5 warmup steps. The use of pruning inside the Optuna study allow us to interrupt the trials that are performing worse than the median of the precedent trials and we can greatly reduce the time spent by this process. The search was performed on the reduced training set and the validation set defined before.

At the end of the study, the best set of parameters is passed as a dictionary, so we can train the autoencoder on the full training set with the following parameters:

encoded space dimension	10
batch size	266
optimizer	Adam
learning rate	0.0033638691442384257
conv1 channels	64
conv2 channels	128
conv3 channels	128
fully connected neurons	128

2.2 Results

The final autoencoder model produce a loss on test set equal to 0.0272. We can see the plot of the loss during the training in Figure 2 and an example of reconstructed samples from the test set in Figure 3. We can also try to take a sample from the latent space and feed it into the decoder in order to see how the model can decode an image starting from latent space point that do not correspond to encoded samples. In Figure 8 we can see the reconstruction of random samples chosen from the latent space. They are blurry and have low quality. This is an expected result from a base autoencoder, and it get worse with the increasing of the latent space dimension. We will see that with a variational autoencoder the quality of this result is completely different.

We can analyze the latent space by encoding all the samples and plotting them as points. Since the latent space dimension that has been chosen by Optuna is 10, it is not possible to easily plot them, we need to perform a dimensionality reduction operation. So we use the Scikit Learn implementation of the PCA to reduce the encoded samples in 2 dimensions. Since we started from a much higher number of dimensions a lot of information is lost in this process and in the plot the clusters of the different letters are not easily detectable (Figure 5).

3 Denoising autoencoder

3.1 Method

After implementing a base convolutional autoencoder we want to implement a denoising autoencoder, that is able to encode and decode the samples removing the noise from the samples during the process. In order to efficiently implement this task we just need to modify the function that perform the training step of the model by injecting random noise into the training set batch that the model is using. Doing so we can train the model using samples that contains different (randomly generated) noise at every epoch, so the model learn to ignore the noise during the process.
In practice for every batch in the dataloader we define a tensor containing gaussian noise with mean 0 and std equal to 0.1, then we add it to the batch, and proceed with the training steps as before. The model architecture is the same used before for the base autoencoder, and the Optuna search is performed varying the same set of parameters used before. We expect to have the best model trained with the best parameters that provides a bigger loss on the test set with respect to the loss of the base autoencoder, since the presence of noise in the sample makes the encoding and ecoding process more difficult.

The parameters chosen by optuna after 100 trials are the following:

encoded space dimension	10
batch size	505
optimizer	Adam
learning rate	0.003301244567030449
conv1 channels	64
conv2 channels	64
conv3 channels	128
fully connected neurons	128

3.2 Results

The denoising autoencoder is trained on the full training set with the test set as validation and it gives a test loss of 0.0283, that is as expected as little higher than the loss on the base autoencoder (Figure 6).
Finally we can evaluate the denoising capabilities of this model trying the encode and decode random chosen samples from the test set that has been injected with noise. We can see in Figure 7 that the images are well reconstructed and perfectly denoised. As before we can generate images decoding any point from the latent space, but similarly to the previous model, they have low quality (Figure 8).

4 Using the autoencoder in a classification task

4.1 Method

At this point we can apply the autoencoder we have trained before in order to use the encoded samples for a classification task. We can define a new model where we add few fully connected layers at the end of the encoder, and then train it after freezing the layers that belonged to the autoencoder in order to change only the weight of the newly added classification oriented layers. But this fine tuning approach can be made way more efficient if we completely separate the encoding of the samples and their classification. So if we produce an entire dataset of encoded samples we can train it on a simple classifier. This is a more efficient solution because the encoding of the samples is performed only once at the beginning and not during the training.
We pass the entire dataset into the encoder and we generate a new dataset of encoded samples, we split it as before into training, validation and test set.

We define a classifier model using four linear layers that are activated by a ReLU function and Softmax on the last layer. The number of neurons in each layers are given in input to the constructor of the class.

Once again we use Optuna to find the best parameters. We decide to test the variation of these parameters:

- batch size, integer in [100, 1000],
- learning rate, chosen with logarithmic uniform distribution in $[1e-4, 1e-1]$,
- linear layer 1 number of neurons, integer in the range [16, 64],

- linear layer 2 number of neurons, integer in the range [16, 64],
- linear layer 3 number of neurons, integer in the range [16, 64].

We used CrossEntropyLoss, Adam optimizer and 30 epochs in every trial. This time we used Optuna with the classification accuracy as the metric that controls the trials, so we need to initialize the study with the "maximize" direction, since we want find the set of parameters that maximizes the accuracy . We used pruning with the MedianPruned set as before.
The best set of parameters is the following

batch size	411
learning rate	0.0018263185631044734
linear layer 1 neurons	45
linear layer 2 neurons	52
linear layer 3 neurons	17

4.2 Results

We train the model on these parameters on the full training set to obtain a loss on the test set of 1.5658 and an accuracy of 0.8958. The accuracy is high and satisfying for this dataset, we can affirm that the use of encoded samples is useful in a classification task. In Figure 9 we can see those values during the training process.

5 Variational Autoencoder

5.1 Method

At last we want to implement a generative model, so instead of using an autoencoder we can implement a variational autoencoder. This model is also composed by an encoder and a decoder but the difference is that the latent space gets regularized before the action of the decoder. Doing so we force the mapping of the input images in the latent space to follow a distribution that is similar to a Gaussian distribution. This gives us a kind of control over the decoder behaviour, meaning that we can exploit the regularized structures in the latent space for the generation of new content through the decoder[2].

We define the encoder with two convolutional layers with kernel size 4, stride 2, padding 1 and ReLU activation. After every convolutional layer we apply a dropout with probability 0.3. Two identical linear layers receive the flattened output of the convolutional layers and produce two arrays with the size of the latent space, called *mu* and *logvar*, since they will represent the mean and the variance af the encoded samples. The encoder is defined with a linear layer and two convolutional layers, with symmetry respect to the encoder.
The class called VariationalAutoencoder uses an instance of both the encoder and the decoder, encode the input batch end pass to the decoder the regularized encoded batch that has been built with the processing of *mu* and *logvar*.

We train the variational autoencoder with Adam optimizers and a fixed set of parameters:

[2]Our code is made with reference to this tutorial: https://colab.research.google.com/github/smartgeometry-ucl/dl4g/blob/master/variational_autoencoder.ipynb

encoded space dimension	10
batch size	256
learning rate	1e-3
conv1 channels	200
conv2 channels	400
optimizer	Adam
epochs number	100

The loss we are using with this model is not part of the PyTorch loss functions. This function compute the reconstruction loss using binary cross entropy and adding it to the Kullback Liebler divergence.

5.2 Results

We can see in Figure 10 the plot of the losses during the training process, the test loss decrease without overfitting until circa 70 epochs and then it settles around the final value. The final loss on the test set is 183.59. We can now compare same samples from the test set with its reconstruction (Figure 11) and we notice that the model is performing a very close reconstruction on the majority of the samples.
We can try to generate new samples from the latent space. Since we have 10 dimensions on the latent space, we cannot easily visualize in a grid the decoded images generated from a regular sampling (Figure 12) of the latent space, but we can perform a random sampling or we can perform a partial exploration of the latent space varying only two dimensions on a regular grid and fixing the other dimensions (Figure 13). In the first case we can see that the generated images have a much higher quality than the images generated using the previous models, this is in fact the main goal of a variational autoencoder. It is also interesting to look at the grid exploration of the latent space performed by varying 2 dimensions and fixing the others. In the grid of images that we obtain we can see the transformation of the results from images that resemble the letter F to the E, G, and J in the corners, and B in the center.

6 Appendix: figures

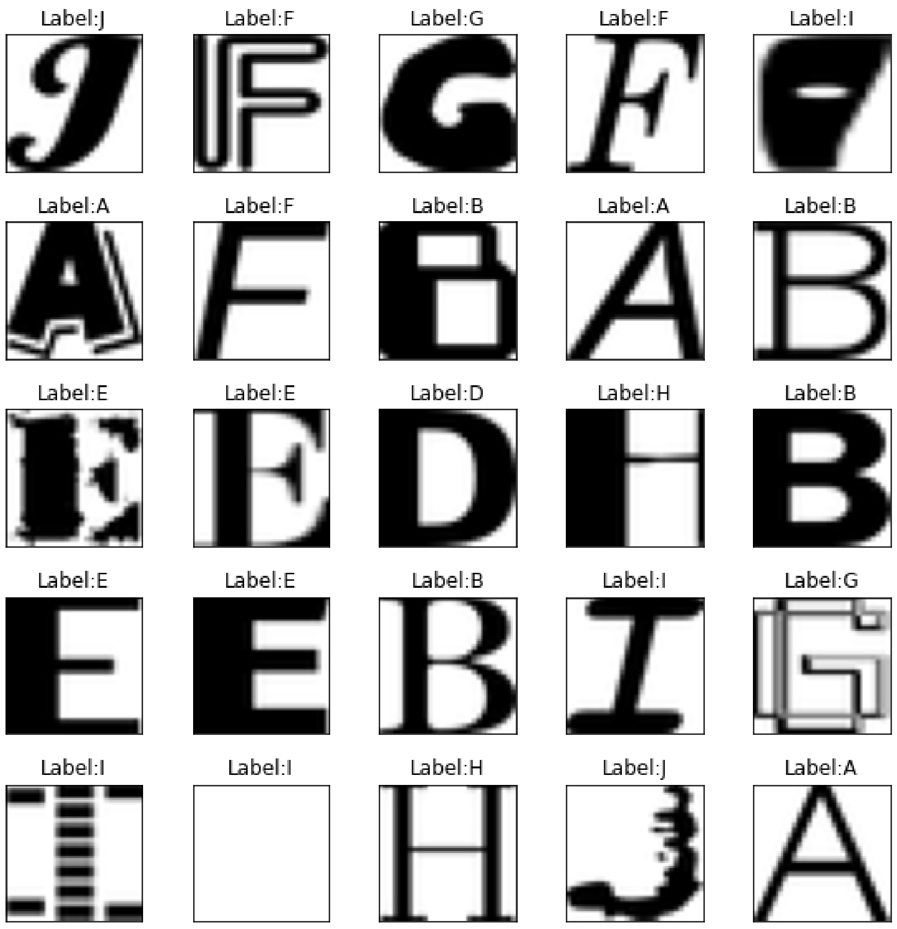

Figure 1: Example of samples in the notMNIST dataset

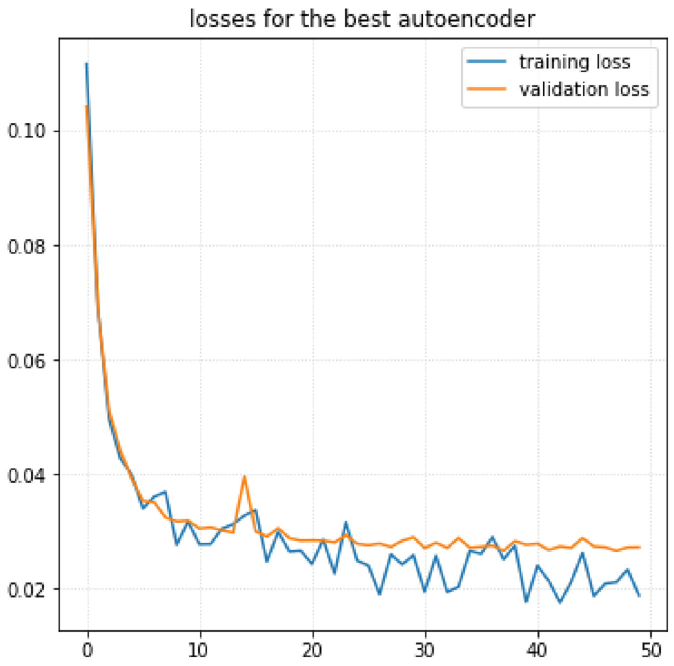

Figure 2: Losses on the training of the best autoencoder on the full training set and the test set as validation.

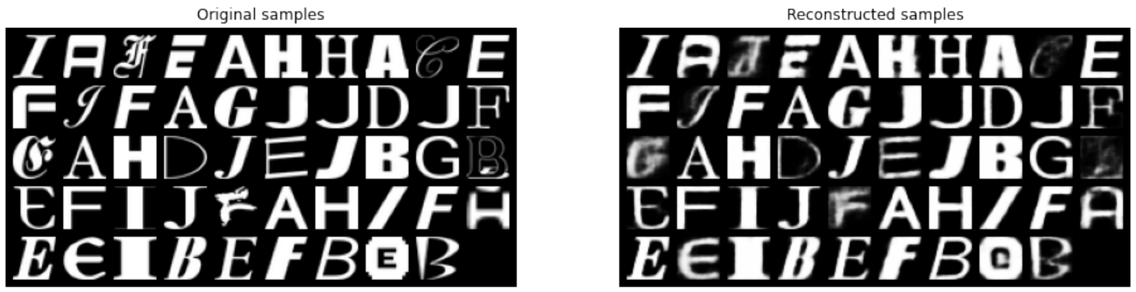

Figure 3: Examples of samples from the test set reconstructed by the autoencoder.

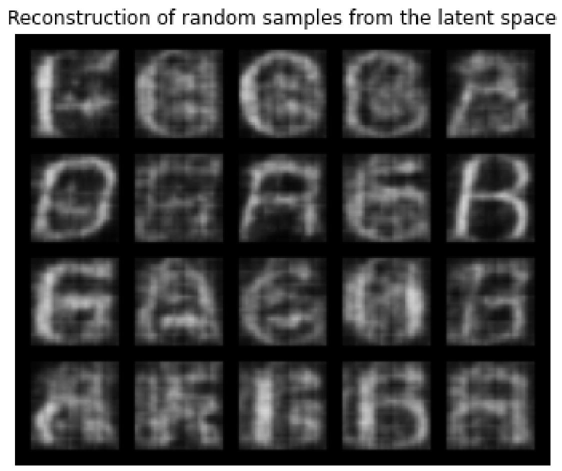

Figure 4: Images decoded from random samples in the latent space.

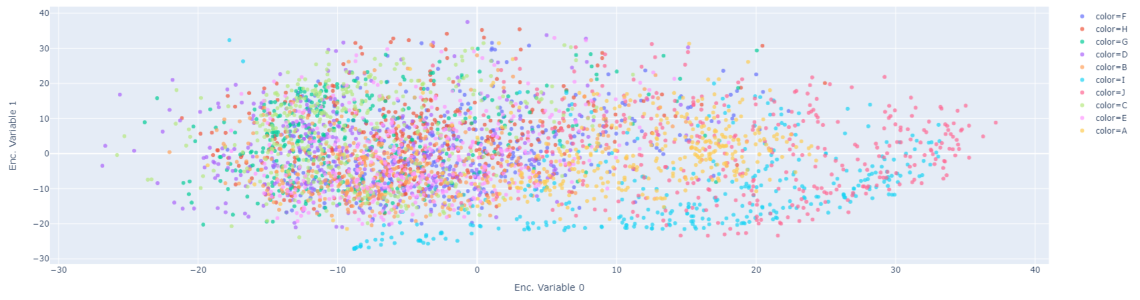

Figure 5: Plot of the latent space samples reduced from 10 dimensions to 2 dimensions with PCA.

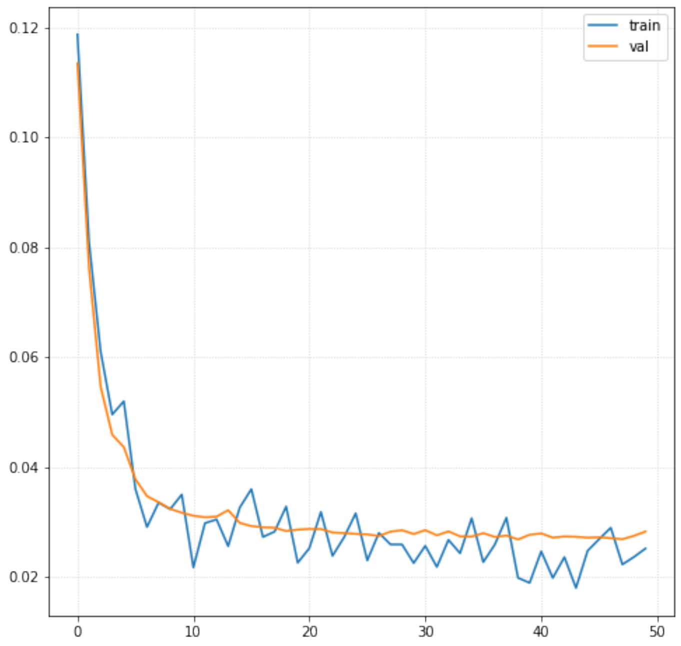

Figure 6: Losses on the training of the best denoising autoencoder on the full training set and the test set as validation.

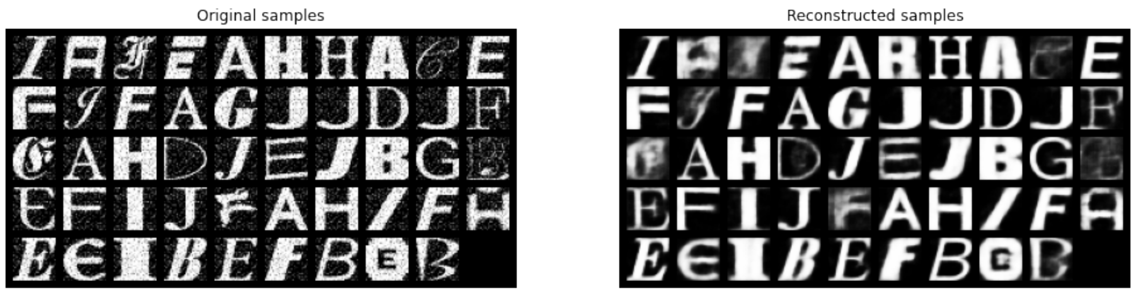

Figure 7: Examples of the denoising of noise injected samples from the test set performed by the denoising autoencoder.

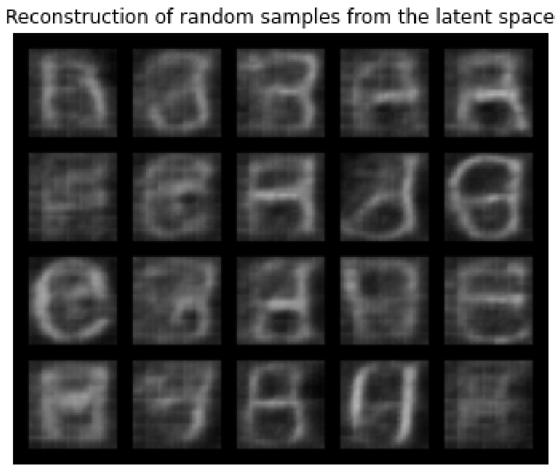

Figure 8: Images decoded from random samples in the latent space of the denoising autoencoder.

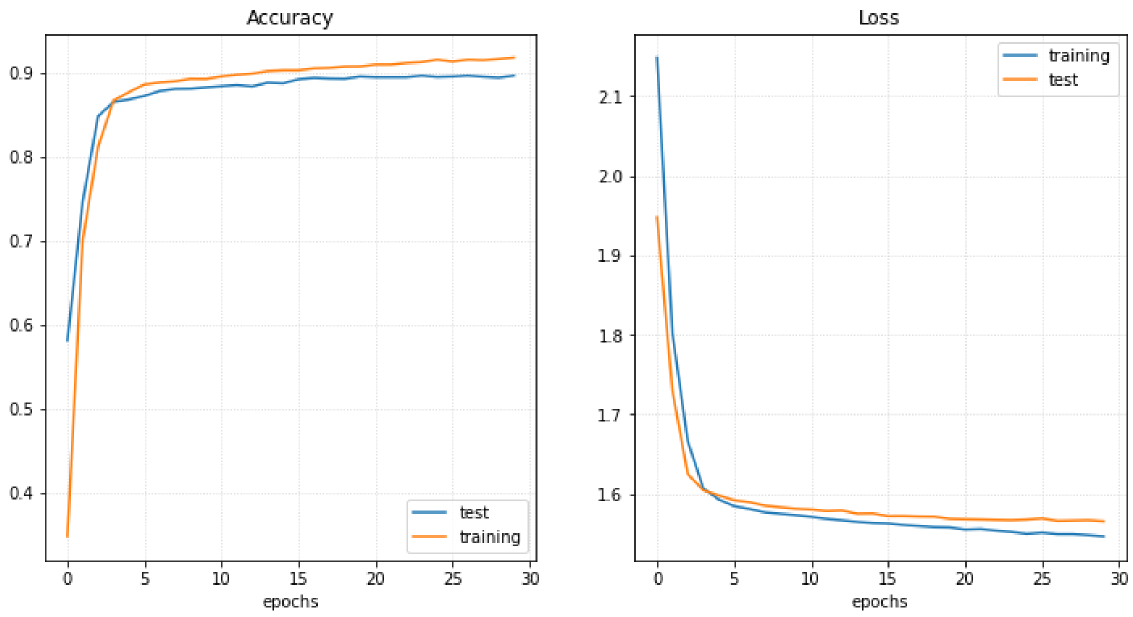

Figure 9: Loss and accuracy during the training of the classifier on the full training set with the best parameters.

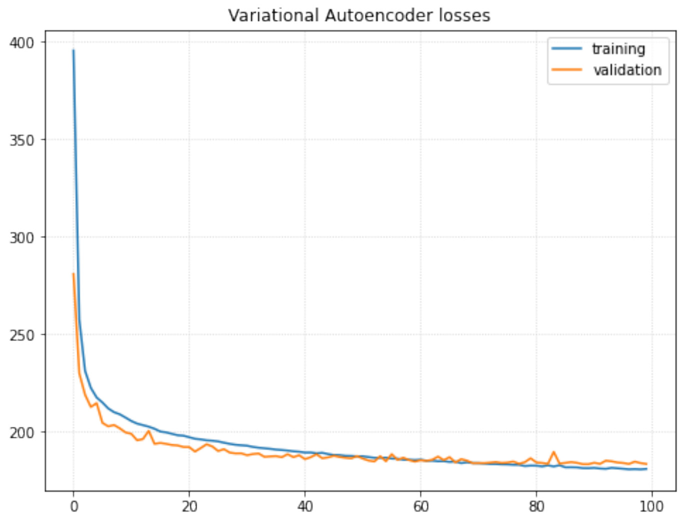

Figure 10: Losses during the training of the variational autoencoder with the full training set and the test set.

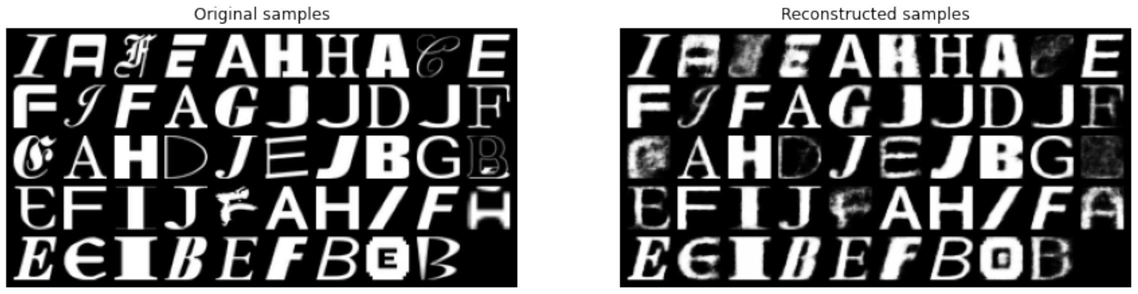

Figure 11: Example of samples from the test set and samples reconstructed by the variational autoencoder.

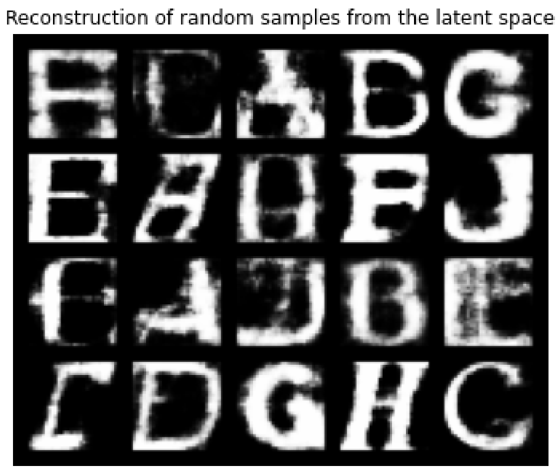

Figure 12: Images decoded from random samples in the latent space of the variational autoencoder.

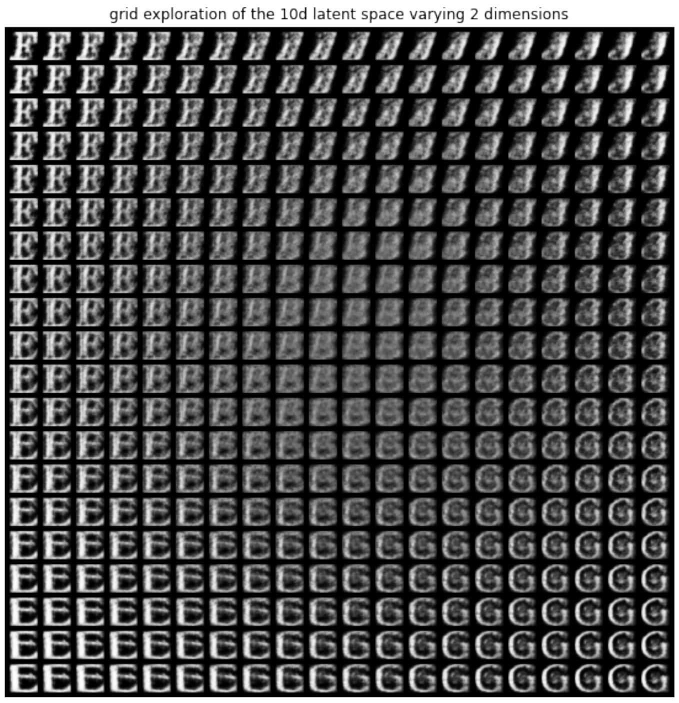
grid exploration of the 10d latent space varying 2 dimensions

Figure 13: Images decoded from random a grid of samples in the latent space of the variational autoencoder, by varying two dimensions and fixing the others.