Autoencoder in PyTorch on MNIST Dataset

Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

January 10, 2021

1 Introduction

In the present work an autoencoder (AE) is implemented in the machine learning framework PyTorch in order to reconstruct the digits present in the images of the MNIST dataset. After having select the best found hypothesis class, with also a dedicated analysis of the compression-performance trade-off, beyond the standard test set we also include one with the add of gaussian noise and another one where the images are partially occluded. Moreover, also a denoising version is implemented, which is trained with noised images in order to teach it how to return as output the cleaned images. Lastly, some samplings from the latent space are performed, in order to explore the generative capabilities of the AE.

2 Program Description

The assignment is solved through one python script AEs.py and a jupyter notebook EX4_Matteo_Guida.ipynb. In the script AEs.py the same implementation presented in the laboratory program is conserved, with just few important modifications: we pass a dictionary to the class containing the hyperparameters, we add the dropout and we use the function trainin, which call internally the two already implemented functions train epoch and test epoch, to train the model. This last allows to manage different optimizers and the whole procedure of the training in a more flexible way.

2.1 Random Search

To find the best model class, i.e. to select the values of the hyperparameters of the model, we use random search with the same scheme illustrated in the assignment 2. The architecture taken from the laboratory script is not changed, so there is a symmetric structure between the encoder part and the decoder one with 3 convolutional layers and 2 linear layers for the first one and 2 linear layers and 3 convolutional layers for the second one. Because a dedicated study on compression-performance trade-off is required in the assignment, in this first part we simply fix the dimension of the encoded space to 4. The table 1 shows the hyperparameters with the respective extrema used in random search.

Optimizer	[Adam, Adamax,AdamW],
Learning Rate	[10^{-2}, 10^{-4}]
L2 Penalty	[10^{-4}, 10^{-3}]
Dropout	[0.1, 0.4]
# Epochs	[60, 100]

Table 1: Values of the hyperparameters used for random search. For the learning rate and the L2 penalty a logarithmic prior between the two extrema is used, for the number of epochs and the dropout a uniform one.

The original dataset is split in 60000 samples for the training set and 10000 for the test set. Moreover the training set is further divided according to the percentage 80% - 20% in order to create a validation set used to perform model selection. In fact, noticing that the difference of the validation loss on different folders is insignificant and we have enough data to properly do the training, the k-fold cross-validation procedure is not included, reducing a lot the computational time. In the table 2 the hyperparameters of the hypothesis class which leads to the lowest values of the MSE loss function on the validation set are reported.

Optimizer	AdamW,
Learning Rate	$4.9 \cdot 10^{-3}$
Dropout	0.10
L2 Penalty	$4.1 \cdot 10^{-4}$
# Epochs	72

Table 2: Hyperparameters for the best found model with random search, fixing the dimension of the encoded space to 4.

2.2 Size of the Hidden Layer

The size of the hidden layer is probably the most important hyperparameter in an AE, because it fixes the size of the **code** used to represent the input data, extracting the relevant features from them. We want this size to be smaller than that of the inputs, in this case the AE is called undercomplete, otherwise the network will just learn to copy them. As one can imagine enlarging this hyperparameter leads to improve the performance of the model, because more information is used to encode the input, so a trade-off is necessary to be found. A large list of possible values are used, concretely a sequence with step 2 in the set [2,20]. The performances of the different models are shown in figure 1.

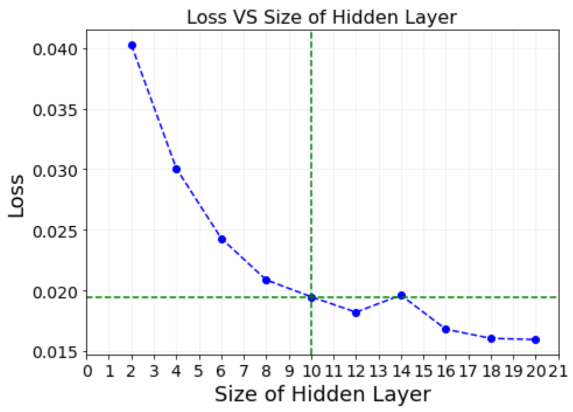

Figure 1: Validation loss score for different values of the hidden layer size.

As can be seen, we have a variation w.r.t. the trend for the point corresponding to the size 14, probably due to a statistic fluctuation. Bearing in mind the compression-performance trade-off, the size equal to 10 is selected as the one of the best selected hypothesis class, this decision is not made thanks to a rigorous method, but only looking at the plot 1.

2.3 Corruption of Data Set: Gaussian Noise and Occlusion.

In this section we test the robustness of the learned code feeding the AE with two modified versions of the test set. In the first case we add to each pixel a gaussian noise with $\mu = 0$ and different standard deviations ranging from [0,1], it must be remembered that data are normalized between [0,1] when the dataloader is created and subsequently the transformation is performed. Some examples of the effect of the transformation can be seen in figure 2.

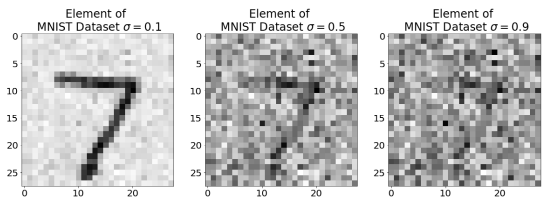

Figure 2: Effect of the gaussian noise transformation on a sample of the MNIST dataset for different values of the standard deviation.

The described operation is done thanks to a lambda function, given to the the transform argument of the torchvision.datasets.MNIST class. The performance of the network for different values of the standard deviation is shown in figure 3.

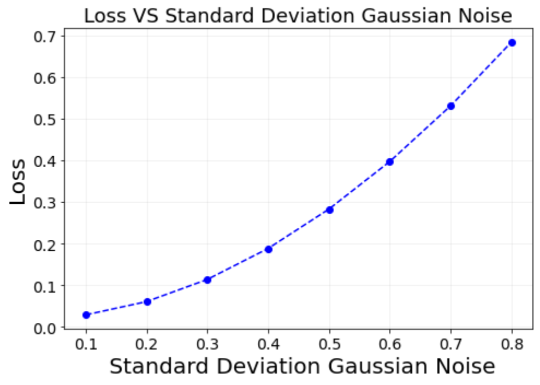

Figure 3: Test loss scores for different values of the standard deviation in the gaussian noise.

The results show a reasonable exponential behaviour, i.e. the performance of the network gets worse exponentially increasing the noise present in the images.

In the second case we occlude a square portion of the given images in line with a certain percentage, setting the values of the pixels belonging to that square to zero. This is done by the function MNIST_occlusion. In the figure 4 we can see two example of the transformations undergone by the images.

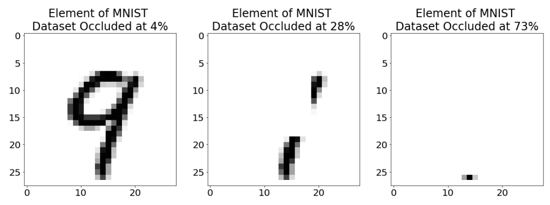

Figure 4: Example of occlusion of an image present in MNIST dataset for different percentages of occlusion.

As one can expect the larger the corruption the lower the performances of the network, in the figure 5 the obtained results are shown.

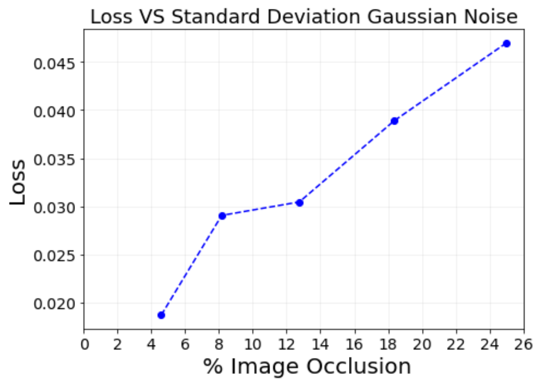

Figure 5: Test loss scores for different values of the percentage of occlusion.

2.4 Denoising Autoencoders

The denoising autoencoder (DAE) is an AE which is feed with corrupted data and is trained in order to output the uncorrupted data. We keep using the best found hypothesis class described in sections 2.1 and 2.2. The main difference with what comes before is that we feed the DAE with corrupted images and we set as a target the uncorrupted images, so that the network learns how to go from one to the other. In the figure 6 the result for an input image with $\sigma = 0.6$ is shown, we can see that the DAE does excellently its job.

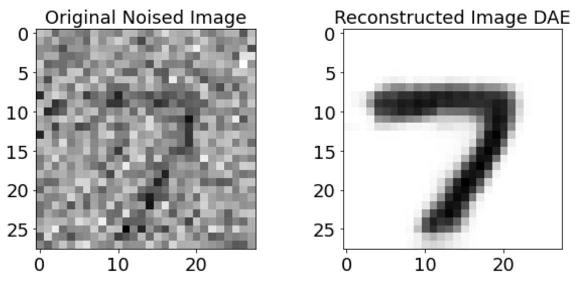

Figure 6: Denoising operation on an input image corrupted by gaussian noise with $\sigma = 0.6$.

In the figure 7 the result for an input image corrupted with $\sigma = 0.68$ is shown, we can see that we crossed the value for which the DAE starts to be in trouble and we obtain a confused and wrong reconstruction.

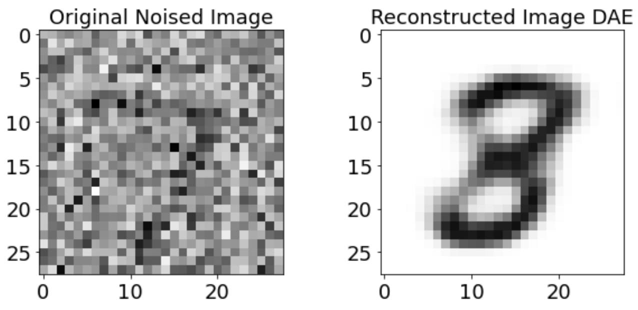

Figure 7: Problematic denoising operation on an input image corrupted by gaussian noise with $\sigma = 0.68$.

2.5 Generative Capabilities

The last point treated is the generative capabilities of the AE, sampling from the code in the latent space. We know that the code associated with the hidden layer with smallest size h is a compressed description of the inputs. So drawing randomly a point in the space $\mathbb{R}^{dim(h)}$ and processing it through the decoding part of the AE allows to obtain a sample of the same category of the input dataset. To verify that the representational space of our autoencoder is regular enough to allow for a smooth sampling we fix two digits of departure and arrival. We have several examples in the dataset of those two digits, after having compute their representation in the latent space, for each entry of the vector related to the code, we compute the centroid. After this operation the number of steps is fixed at will and for each value of the neurons related to the code a vector with dimension equal to the number of steps is created with evenly spaced numbers between the two extrema, i.e. the centroids for that neuron related to the two digits. Each image of the sequence in figure 8 is created, for a fixed step, decoding the vector composed by the values of each hidden neuron of the layer h.

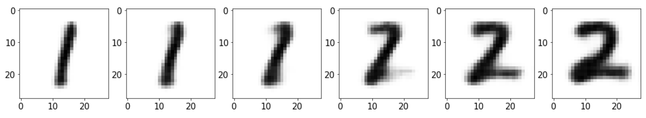

Figure 8: Sequence of images formed sampling from hidden layer h for values of neurons evenly spaced between the centroids associated with the two selected digits, in the example 1 and 2.

The transition between the two digits seems quite smooth, for this reason the regularity of the representational space seems verified and the implementation of the variational autoencoder is neglected.