Homework 2: Unsupervised Deep Learning

Alessandro Lambertini - 1242885
(Dated: August 9, 2022)

In this homework we are required to solve different tasks related to unsupervised deep learning. All the tasks in the homework are carried out within the FashionMNIST dataset. The basic task is to correctly implement a convolutional autoencoder and report its performances. Moreover, we are asked to exploit hyperparameters tuning strategies, advanced optimizers and regularization methods to improve the autoencoder performance. After that we fine-tune our convolutional autoencoder using the FashionMNIST dataset in a supervised way and we compare the results with the one obtained in the classification task of the first homework. As a more advanced task, we implement and test a variational autoencoder. Finally, we explore the latent space structure of the autoencoders we implemented and generate new samples from them. Plots that shows the learning process and the optimization procedure are produced for both autoencoders.

I. INTRODUCTION

A. Convolutional autoencoder

The autoencoder is an unsupervised deep learning algorithm that learns encoded representations of the input data and then reconstructs the same input as output. It consists of two networks, Encoder and Decoder. The Encoder compresses the high-dimensional input into a low-dimensional latent space, by extracting the most relevant features from it, while the Decoder decompresses the encoded data and tries to recreate the original input.
When the inputs to our autoencoder are images convolutional neural networks comes into play as the architecture for the Encoder and the Decoder, from here the name convolutional autoencoder.
The goal of this architecture is to maximize the relevant informations encoded in the latent space and minimize the reconstruction error. The reconstruction error is usually the mean-squared error between the reconstructed input and the original input when the input is real-valued.

B. Variational autoencoder

A problem that could arise with convolutional autoencoders is that the latent space can be irregular or incomplete. This is a real problem if we would like to exploit the latent space for generative purposes. In order to solve this problem variational autoencoders are introduced, that is an autoencoder whose training is regularized to avoid overfitting and to ensure good properties of the encoded space. The regularization procedure consists in encoding an input as a distribution over the latent space instead of a single point. The distributions returned by the encoder are forced to be close to a normal distribution, in this way it can be shown that both local and global regularity of the latent space is obtained. Therefore the loss function that has to be minimized in this case has two components. One that tries to reduce the reconstruction error and one that tends to regularize the latent space. The regularization consists in making the distributions returned by the encoder close to a standard normal distribution through the Kulback-Leibler divergence.

II. METHODS

A. Convolutional autoencoder

The implementation of the convoolutional autoencoder exploit the Pytorch framework, while the part concerning the optimization of the hyperparameter is carried out with OPTUNA. To tackle the task 4 classes and one script are implemented:

- *Encoder*: This class is contained in "Conv_autoenc.py". It defines the architecture of the encoder and it is structured in the following way:

  The convolutional part:

  - First convolutional layer: it takes 1 channel in input and returns 8 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and a symmetric padding of 1 pixel.

  - Second convolutional layer: it takes 8 channels in input and returns 16 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and a symmetric padding of 1 pixel.
  - Third convolutional layer: it takes 16 channels in input and returns 32 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and no padding.

  The linear part:

  - A flattening layer that lowers the dimensionality of our tensors to 1.
  - First linear layer: it takes 288 features in input and returns 64 features in output.
  - Second linear layer: it takes 64 features in input and returns a in output a number of features equal to the dimension of the latent space, that is treated as an hyperparameter to be optimized.

  The forward pass consists in applying to the image in input the convolutional part, the flatten layer and the linear part sequentially.

- *Decoder*: This class is contained in "Conv_autoenc.py". It defines the architecture of the decoder and it is structured in the following way:

  The linear part:

  - First linear layer: it takes in input a number of features equal to the dimensionality of the latent space and returns 64 features in output.
  - Second linear layer: it takes 64 features in input and returns 288 features in output.
  - An unflattening layer which raises the dimensionality of our tensor to 3, with size $32 \times 3 \times 3$

  The (de)convolutional part:

  - First (transpose) convolutional layer: it takes 32 channel in input and returns 16 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and no padding.
  - Second (transpose) convolutional layer: it takes 16 channels in input and returns 8 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and a symmetric padding of 1 pixel and an output padding of 1 that is needed to correctly compute the output shape since stride is >1.
  - Third (transpose) convolutional layer: it takes 8 channels in input and returns 1 channels in output. It has a kernel of 3×3 pixels with a stride of 2 pixels and a symmetric padding of 1 pixel and an output padding of 1.

  The forward pass consists in applying to the point of the latent space in input the linear part, the unflattening layer and the (de)convolutional part sequentially. Finally, a sigmoid function is applied to force the output to assume valid pixel values.

- *Classifier*: This class is contained in "Conv_autoenc.py". It defines the architecture of the classifier used for fine tuning. It contains just one hidden layer with 120 features in input and 84 features in output, and two dropout layers with default values for the probabilities of 0.2 and 0.1.

- In "Conv_autoenc.py" are also defined two functions that implement the training and the test processes for each batch.

- *Hype_Select_Convaut*: This class is contained in "Hype_Select_Convaut.py", and it takes care of the optimization procedure. In this case, the parameters that can be optimized through the OPTUNA study are: number of epochs, batch size, learning rate, weight decay, optimizer, momentum if needed, and the dimension of the latent space. The optimization is carried out by the Hype_sel() method that try to minimize the validation loss of the model. The outputs of the optimization are a *.txt* file with the best combination of parameters found, and some optuna plots about the study. Finally, the best values found during the study are assigned to the attributes of the Hype_Select_Convaut object.

- *Hype_study_Convaut.py* : This is the script where the optimization and training procedures are performed. The optimization is carried out over 15000 items of the training dataset, 10000 for training and 5000 for validation. Once the OPTUNA study is completed, a k-fold cross validation is implemented over the training dataset. During the training we store the models and their performances so we can exploit them later during the evaluation of the results. Moreover, during the training of the last model we store the output of the network to monitor the progress and produce a plot later. Once the training is completed the autoencoder with the

minimum value of validation loss is chosen to perform the fine tuning procedure over the test dataset, generate new samples and for the exploration of its latent space by the encoded representation of the test dataset. The script will output:

- A plot that shows the average training and validation loss across the k folds.
- a scatter plot of the encoded representation of the test sample in the latent space. If the optimal dimension of the latent space found during the optimization is greater than 2, then 2 scatter plots are produced The first one exploit PCA and the second one exploit t-SNE to perform dimensionality reduction.
- A plot that shows 25 images generated from the latent space. The points are sampled from a normal distribution with the mean and the standard deviation of the testset encoded data.
- A confusion matrix that shows the performance of the model in classification over the test dataset.
- A plot that shows the training and validation losses of the fine tuning procedure.
- A plot that shows the training and validation accuracies of the fine tuning procedure.

I decided to run a study that performs 150 different trials over the parameters in Listing 1. The activation function is 'ReLU' and the loss functions are the mean squared error loss for the training and validation of the convolutional autoencoder and the cross entropy loss for the fine tuning. Once the optimization has ended, the k-fold CV procedure is carried out with $k = 10$. All the results are stored in a folder named "study_convaut_results" that is created during the optimization procedure. To run the script it is sufficient to load the files in the folder "code" in the Colab notebook named *Convolutional_autoencoder.ipynb* and run the cells.

```
#ATTRIBUTES OF THE CLASS THAT CORRESPONDS TO HYPERPARAMETERS TO BE OPTIMIZED
model.encoded_space_dim = [2,10]
model.n_epochs_range = [10,150]
model.batch_size_range = [64,256]
model.learning_rate_range = [0.00001, 0.001]
model.weight_decay_range = [1e-6,1e-4]
model.optimizer = ['Adam','SGD','RMSprop']
```

Listing 1. Parameters to be optimized for both the autoencoders

B. Variational autoencoder

As for the convolutional autoencoder we exploit Pytorch and OPTUNA. To build, train and test the variational autoencoder 3 classes and one script are implemented. For sake of clarity, from now on, let's call x our input image and z it's encoded representation in the latent space.

- Encoder: This class is contained in "Var_autoenc.py". It implements the architecture of the variational encoder, whose job is to learn the conditional probability distribution $z \approx p(z|x)$ and return z. In practice we parametrize the distribution as a Gaussian and learn it's parameters μ and σ, then we write it as a factorized normal distribution: $P(z|x) \sim \mu + \mathcal{N}(0,1) \otimes \sigma$. This is necessary to allow the gradients to backpropagate from the decoder to the encoder without encountering a random node that stops them. This is known as reparametrization trick. We have the same three convolutional layers defined for the convolutional autoencoder, and then three linear layers, one reduces the number of features in output and the last two are used to compute μ and σ. Finally we output z after that KL divergence has been computed.

- Decoder: This class is contained in "Var_autoenc.py". It implements the architecture of the variational decoder, whose job is to learn the conditional probability distribution $x \approx p(x|z)$ and return x. It takes in input z from the encoder and return x through an architecture that is the same as the one used for the convolutional autoencoder.

- In "Var_autoenc.py" are defined also the functions that defines the functions that define the training and validation/test processes for each batch. The loss in this case is defined as the reconstruction term given by the reconstruction term seen for the VAE plus the regularization term given by the KL divergence.

- *Hype_Select_Varaut*: This class is contained in "Hype_Select_Varaut.py". It is structured in the very same way of the class Hype_Select_Convaut and perform the optimization of the hyperparameters.

- textitHype_study_Varaut.py: This is the script where the optimization and training procedures are launched. It has the same structure of Hype_study_Convaut.py, and also the outputs are the same.

I decided to run a study that performs 150 trials over the same parameters searched for the convolutional autoencoder in 1. The activation function also is "ReLU" and the loss function is given by the sum between the reconstruction term and the KL divergence. After the study the k-fold CV procedure is carried out with $k = 10$. All the results are stored in a folder named "study_varaut_results". In order to run the script it is sufficient to load the file in the folder "code" in the Colab notebook named *Variational_autoencoder.ipynb* and run the cells.

III. RESULTS

A. Convolutional autoencoder

Through the optimization strategy described above we find that the best combination of hyperparameters for our convolutional autoencoder is the one described in II

Study numbers
Number of finished trials: 150
Number of pruned trials: 132
Number of complete trials: 18
Best value: 0.0168

TABLE I.

Hyperparameters found:
latent space dimension: 7
batch size: 66
number of epochs: 60
learning rate: 9.6×10^{-4}
l2 regularization: 1.1×10^{-5}
optimizer: RMSprop

TABLE II.

In FIG.1 it can be seen that all the 10 models trained behave similarly, as expected.

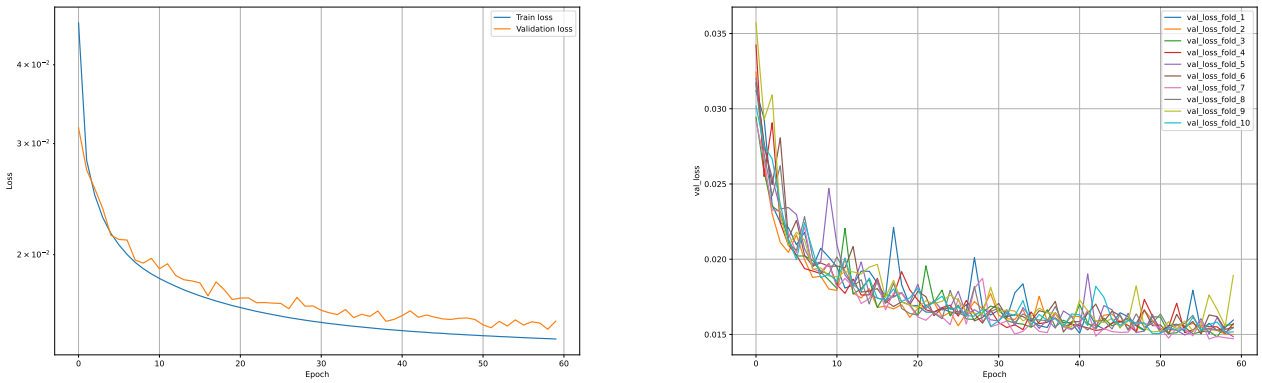

FIG. 1. Left: average train and validation reconstruction losses; Right: validation reconstruction losses of the 10 different network

In FIG.2 on the left is shown the progress made during the training. We can see that already after the first epoch the network has perfectly learned the edges, while the shoe is still a lot blurry. After the 71 epochs the reconstruction improved a lot, however the reconstructed image seems to be more uniform with respect to the original one. On the right instead we see some sampled generated by the best performing network. Since the optimal dimension found during the study is 7 and therefore a systematic graphical exploration of the space is difficult, then we have sampled points from a normal distribution around the average value computed over the test dataset. Most of the generated images resemble clothes or shoes, however the performance of the variational autoencoder seems to be way better as we can see in FIG.11.
The fine tuning procedure is carried out with Adam optimizer, 50 epochs and a batch size of 128, a learning rate and an l2 regularization of 10^{-4}. As we can see in FIG.3 the performance is not comparable with the one obtained in the previous homework. As suggested by the losses plot, maybe better performances could've been reached with a higher number of epochs. However, for this part of the homework hyperparameters optimization is not carried out. In Appendix B can be found the results of the fine tuning also for the variational autoencoder which, as can be seen in FIG.12, performs even worse.

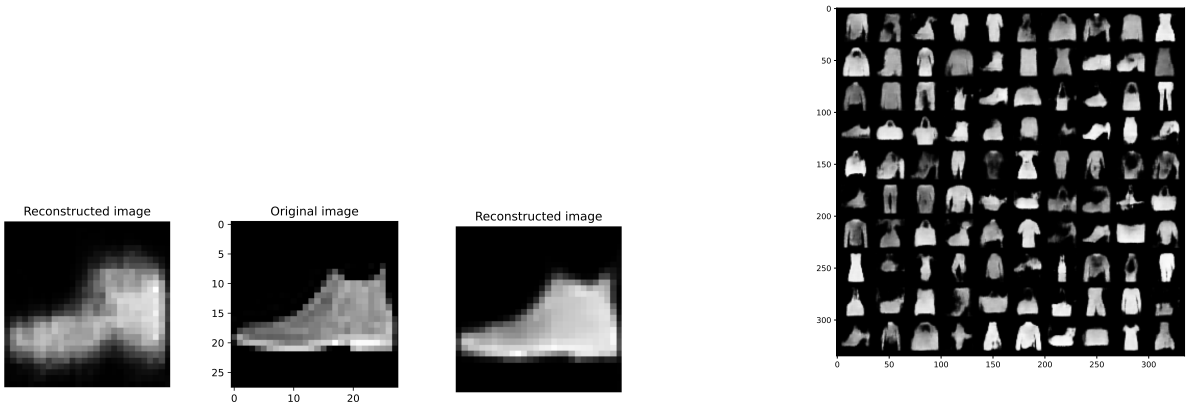

FIG. 2. Left: reconstructed image at epoch 1/60, original image, reconstructed image at epoch 60/60; ; Right: grid of 100

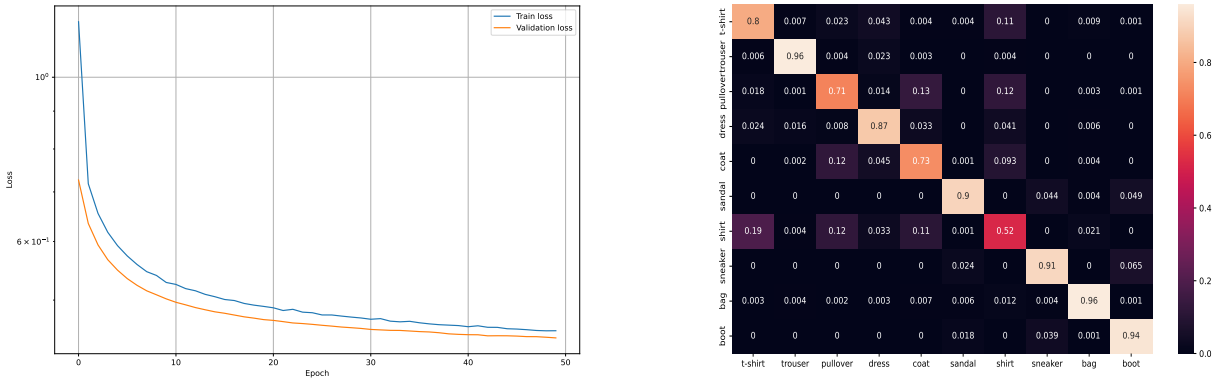

FIG. 3. Left: Losses of the classifier during training with variational autoencoder; Right: confusion matrix of the accuracies of the classifier over the test dataset.

The results of the PCA and t-SNE for the convolutional autoencoder are shown in FIG.4. From the plots we can see that although in most cases it is possible to identify a region that describes a specific class, the representation of the space is less regular and "complete" than the one obtained with the variational autoencoder shown in FIG.6.

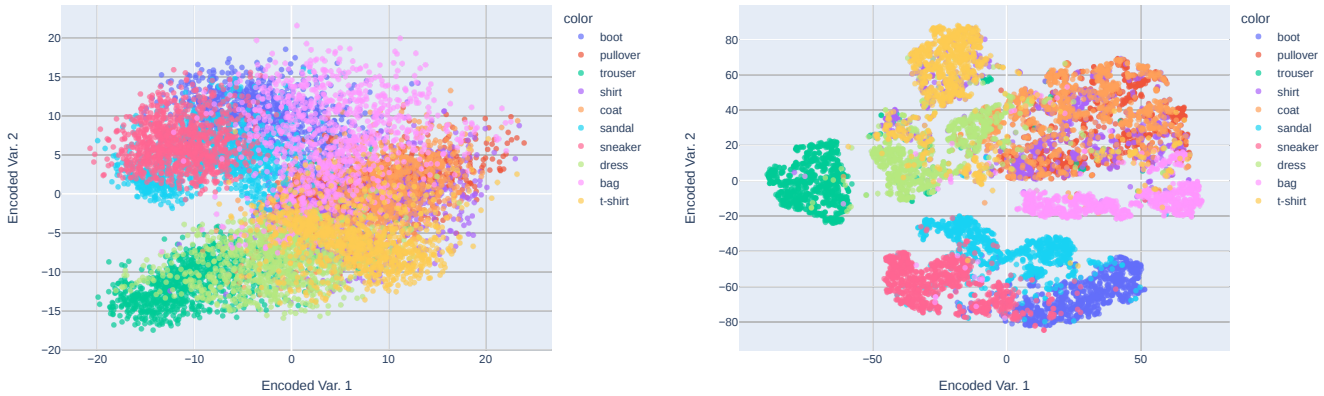

FIG. 4. Left: Latent space representation obtained through PCA; Right: latent space representation obtained through t-SNE

B. Variational autoencoder

Through the optimization strategy described above we find that the best combination of hyperparameters for our variational autoencoder is the one described in IV.

Study numbers
Number of finished trials: 150
Number of pruned trials: 109
Number of complete trials: 41
Best value: 27.269

TABLE III.

Hyperparameters found:
latent space dimension: 7
batch size: 100
number of epochs: 71
learning rate: 9.9×10^{-4}
l2 regularization: 2.5×10^{-5}
optimizer: Adam

TABLE IV.

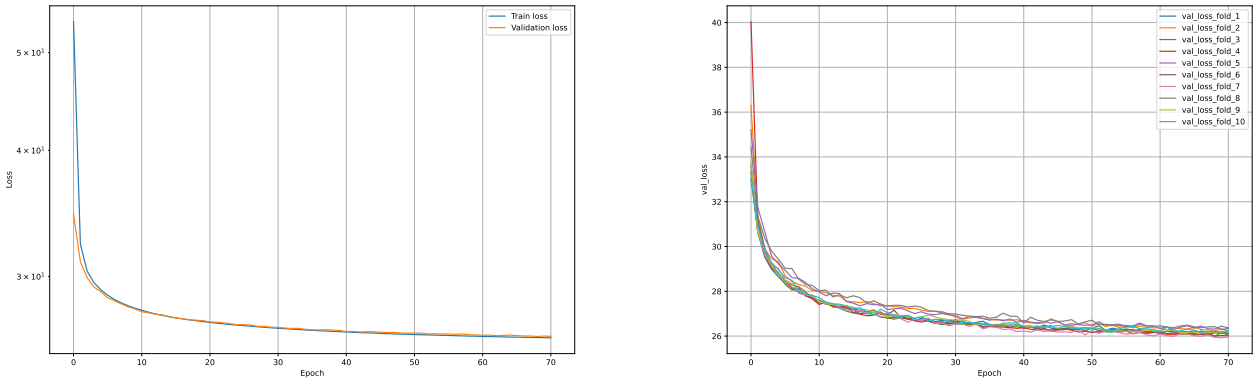

FIG. 5. Left: average train and validation reconstruction losses; Right: validation losses of the 10 different network

FIG. 6. Left: Latent space representation obtained through PCA; Right: latent space representation obtained through t-SNE

IV. CONCLUSIONS

To conclude, one can say that both the tasks proposed has been explored and that the networks trained performs sufficiently good in both cases. Moreover, all the features of the models have been reasonably justified and no too strange behaviour are observed in the outputs.

Appendix A: Convolutional autoencoder

1. optimization

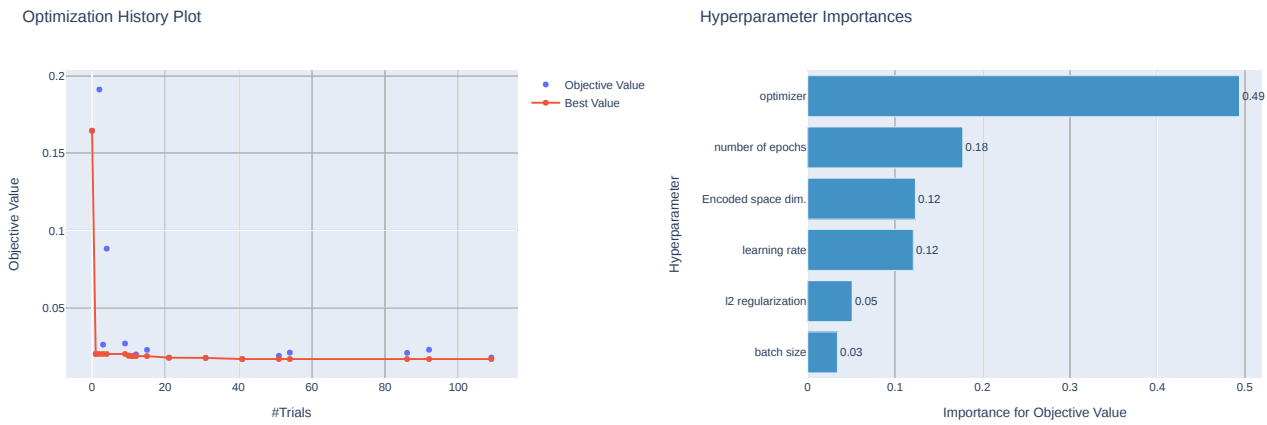

FIG. 7. Left: Optimization history; Right: Hyperparameters importances

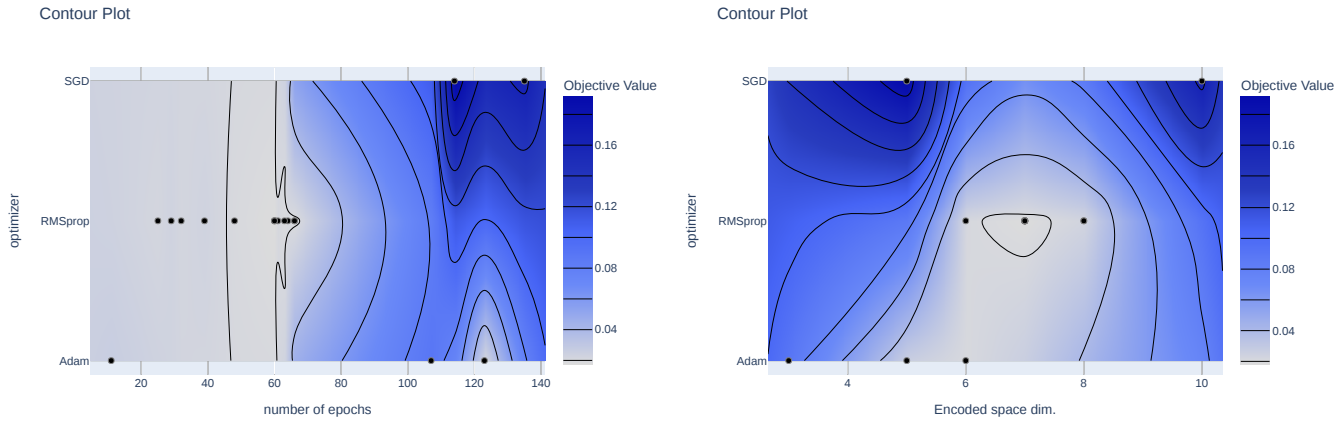

FIG. 8. OPTUNA countur plot of the most important parameter, optimizer, with respect to the others two most important parameters, number of epochs and encoded space dimensionality

Appendix B: Variational autoencoder

1. optimization

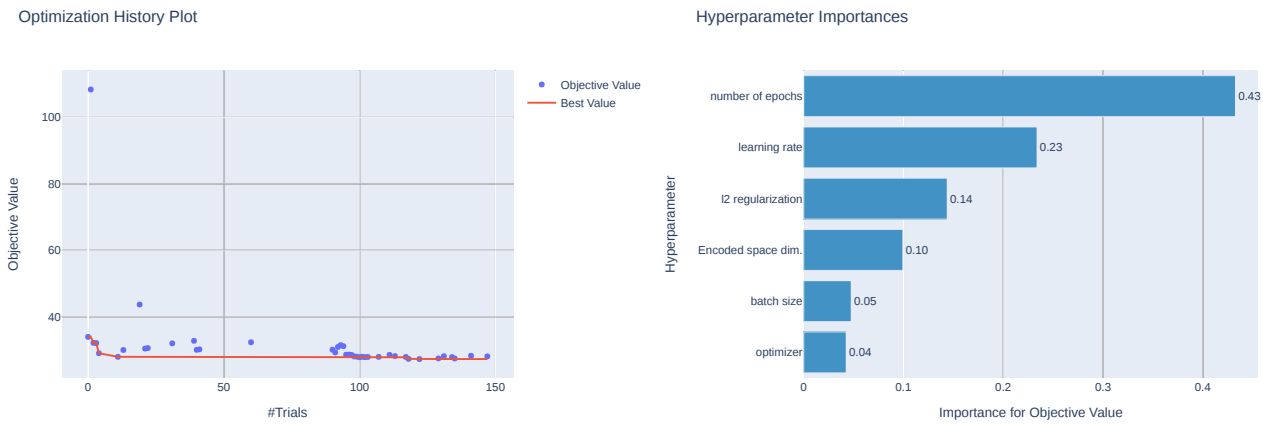

FIG. 9. Left: Optimization history; Right: Hyperparameters importances

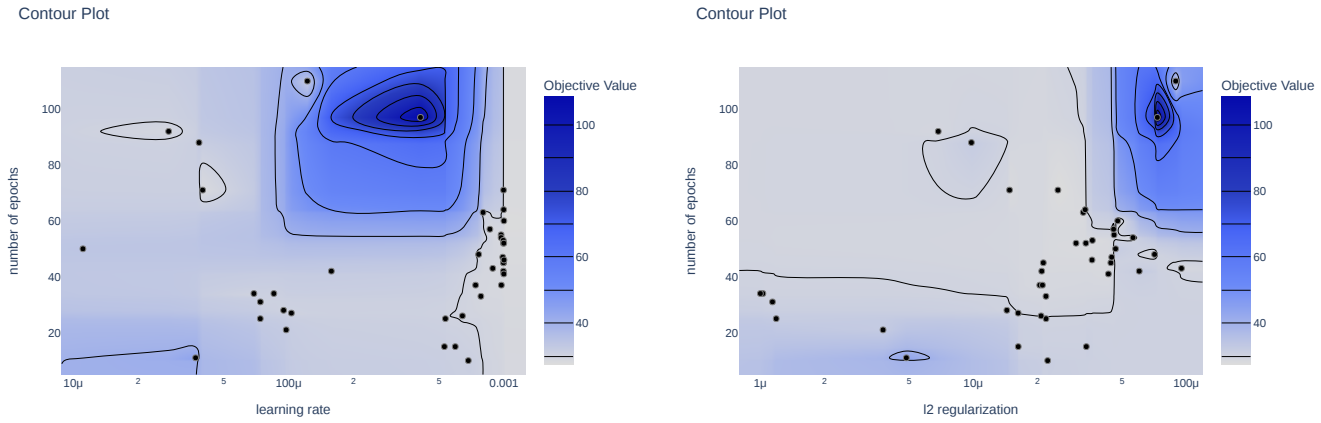

FIG. 10. OPTUNA countur plot of the most important parameter, number of epochs, with respect to the others two most important parameters, learning rate and l2 regularisation

2. Training

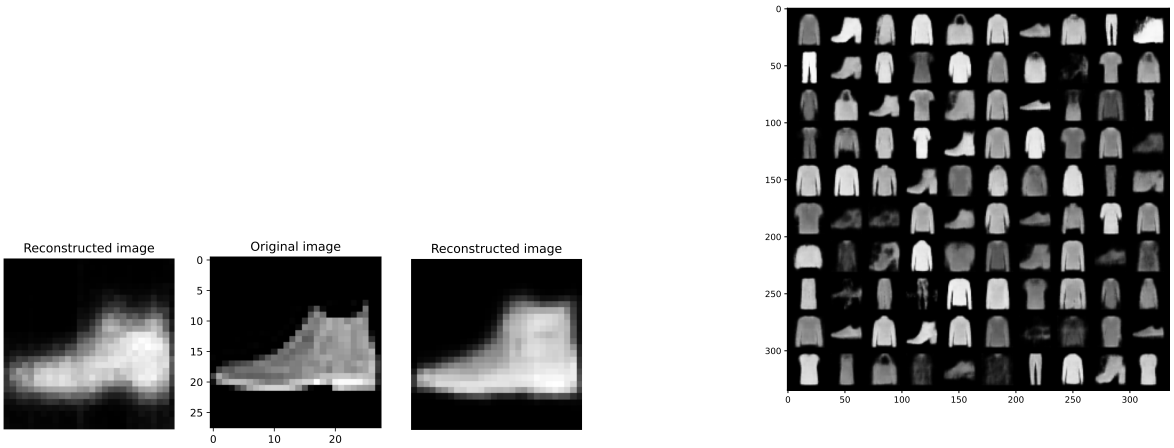

FIG. 11. Left: reconstructed image at epoch 1/71, original image, reconstructed image at epoch 71/71; ; Right: grid of 100 generated images sampled from a normal distribution with mean and standard deviation computed over the test dataset.

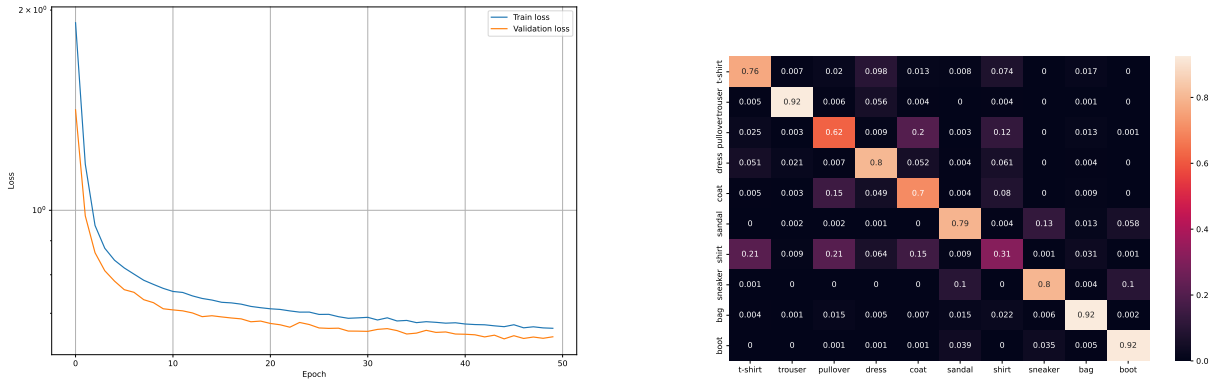

FIG. 12. Left: Losses of the classifier during training with variational autoencoder; Right: confusion matrix of the accuracies of the classifier over the test dataset.