Homework 2: Unsupervised Deep Learning

Michele Guadagnini - ID 1230663

August 8, 2022

Abstract

In this second homework we are asked to study and implement Unsupervised Learning algorithms. In particular, we use a convolutional autoencoder network to learn a latent representation of the FashionMNIST dataset. To build the model we use the *PyTorch-Lightning* framework and we use the *Optuna* package to optimize the hyper-parameters. Also, we try to use the features learned by the autoencoder to classify the image samples and compare the obtained accuracy with the one of the previous homework, where a simpler Convolutional Neural Network (CNN) has been used.
Finally, we test two other autoencoder flavors: a denoising autoencoder and a variational one. In these cases hyper-parameters have been set to the ones obtained from the previous *optuna* optimization.

1 Introduction

FashionMNIST is a dataset of Zalando's articles images composed of 70000 (60000 for training, 10000 for testing) gray-scale pictures of size 28×28 and labelled into 10 classes. In this work however we will not make use of the labels in training, as we are studying unsupervised learning methods. Indeed, an *AutoEncoder* aims at learning a latent representation of the dataset distribution by mean of a Encoder/Decoder pair of networks and the loss is calculated by comparing the reconstructed image (decoded) with the original sample. Its main applications are dimensionality reduction, data compression, image denoising and generation.

In this work all the implemented autoencoders are built to be *symmetric*, meaning that the shapes of the batch through the Encoder is specular with respect to the shapes through the Decoder. All the models have been implemented using *PyTorch* and *PyTorch-Lightning*. The *Optuna* package has been used to optimize the hyper-parameters of the first model (*Convolutional Autoencoder*), and the same set of hyper-parameters has been used in the other models (*Denoising* and *Variational autoencoders*).

2 Method

2.1 Convolutional Autoencoder

As done in the first homework we defined the datamodule class (within PyTorch-Lightning framework) that will be used to split dataset into train and validation sets and to create dataloaders.
The encoder network is composed of a convolutional part and a fully connected part. The first is a sequence of three convolutional layers, eventually with a normalization layer inbetween. The second part is made of a single linear layer eventually followed by a dropout layer. The activation function is set as a hyper-parameter of the model, although in this work we have fixed it to be *Leaky_ReLU* after some tests. Attached to the encoder there is the latent space layer, which is a linear layer with a certain number of units to be optimized. The model is completed by the decoder network, which is built to be symmetric with respect to the encoder and its hyper-parameters are derived from the encoder ones. In particular, the list of channels per layer is inverted together with the other convolutional parameters (*kernel size*, *stride* and *padding*). Also, the proper *output_padding* is computed in order to obtain symmetric intermediate output shapes of the images through the layers. The whole model is encapsulated in a *PyTorch-Lightning* class named *SymmetricAutoencoder*. The loss function used to

evaluate the quality of the reconstruction is the *Mean Squared Error* (MSE) and the optimizer is selected from a list during optimization.

The model class receives as input a dictionary containing the set of hyper-parameters to adopt. To optimize them we have created a class for the hyper-parameters space (*SymmetricAutoencoderHPS*) that defines a sampling method for both model and optimizer parameters. This allows us to call the sampling method within the *objective* function of the *Optuna* study. As in Homework 1 we made use of the *Tree-structured Parzen Estimator* (TPE) to guide the sampling and we used *pruning* and *early_stopping* to save time. The study has been run for 60 trials and the metric used to identify the best set of hyper-parameters is the minimum validation loss reached during each trial, computed on 8800 samples taken from the 60000 of the training set.
The hyper-parameters search space is reported in Table 1. Parameters of the convolutional layers has been selected from a set of three candidate configurations that defines the kernel size, the stride and the padding for each of the three layers. The same applies to the number of output channels. Note that as normalization layer we have used *Instance normalization* instead that *Batch normalization* to ensure independence of latent space representation for different images. Some of the parameters (*dropout probability* and *activation*) has been selected by some short tests and fixed before the study to reduce the searching space. For the learning rate instead we have used a *PyTorch-Lightning* heuristic procedure called *Learning rate finder* that can produce an estimate of the optimal value (details in Appendix).

Once obtained the optimized set of hyper-parameters, we train the autoencoder on the whole dataset, using the same train/validation split as before. Some callbacks have been used during training in order to save the best performing model (*checkpoint* callback) and to save time (*early stopping*). We also created a callback to plot the reconstruction of a random image at the end of each epoch. In order to test the trained model and evaluate its performance we apply some dimensionality reduction algorithms (*PCA* and *t-SNE*) to project the latent encodings of the test dataset into a 2D space that can be visualized. Finally, we generate some images by decoding some random samples from the latent space and the centroids of each class calculated using the labels.

2.2 Transfer Learning

To exploit the convolutional kernels learned by the encoder for classification we implement a fully-connected network with just 2 layers that receives as input the flattened output of the convolutional part of the encoder and outputs a class label. The first layer is cloned from the first linear block of the encoder, on which training and gradient tracking is activated. The second layer is a simple linear layer that outputs the class label. This approach is equivalent to create a CNN starting from the encoder network with freezed parameters in the convolutional layers, but it allows us to call the forward pass of the convolutional layers just once for each sample instead that at every epoch, saving a lot of repeated computations and time. This of course is possible because the dataset is small and the preprocessed inputs fit into memory.

For this model we skip the hyper-parameters optimization and we set activation and optimizer to the same values of the autoencoder model, while we remove L2 penalty. To set the learning rate instead, we use again the learning rate finder procedure from *PyTorch-Lightning*.

Once trained the network, as done in Homework 1, we calculate over the test dataset the *CrossEntropy* loss, the accuracy and the confusion matrix.

2.3 Denoising Autoencoder

To implement the *Denoising* version of the autoencoder we create a new class (*DenoisingAutoencoder*) that inherits from *SymmetricAutoencoder* class. To adapt the model to the new task we need to define a corruption transformation and then modify the training procedure such that the input image is corrupted and passed to the network, but the reconstruction loss is evaluated using the original (not corrupted) image. The corruption function can be passed as an argument at class initialization and we use a composition of 4 transformations: horizontal and vertical flipping, gaussian noise with zero mean and with standard deviation of 0.4 and random erasing of a portion of the image. Each of the 4 transformations is applied with a probability of 0.5; in this way the probability of having at least one transformation applied is 0.9375.

Also in this case we skip the hyper-parameters optimization due to its time requirements and we use the same hyper-parameters set resulting from the optimization of the standard autoencoder described in Subsection

Parameter name	**Sampling space**
Convolutional config.	{[3, 2, 1],[3, 2, 1],[3, 1, 0]}
	{[5, 2, 1],[5, 2, 1],[3, 1, 1]}
	{[7, 2, 1],[5, 2, 1],[3, 1, 1]}
Channels config.	[16, 32, 32]
	[16, 32, 64]
	[32, 32, 64]
Linear units	[128, 256, 8]
Latent space units	[10, 40, 2]
Instance norm.	{TRUE, FALSE}
Optimizer	{adam, sgd, adamax}
L2 penalty	[1e-7, 1e-4]
*Learning rate**	//

Table 1: Hyper-parameters search space. Regarding *Linear units* and *Latent space units* sampling space, the first two numbers defines the range where to sample, while the third number represent the gap to keep between values. *Learning rate* has been estimated at each trial using learning rate finder functionality (see Appendix).

Parameter name	**Selected value**
Convolutional config.	[3, 2, 1],
	[3, 2, 1],
	[3, 1, 0]
Channels config.	[32, 32, 64]
Linear units	256
Latent space units	30
Instance norm.	TRUE
Optimizer	adamax
L2 penalty	5.27E-07
Learning rate	0.0083176
Dropout prob.	0.16
Activation	leaky_relu

Table 2: Set of hyper-parameters adopted to train the model. Each of the three arrays inside the convolutional configuration reports *kernel size*, *stride* and *padding* of one layer. Parameters fixed before optimization are reported in *italics*.

2.1. Training and testing of the model are done in the same way as above. The only differences are the usage of the learning rate finder procedure from *PyTorch-Lightning* and the addition of a visualization of the capability of the network to reconstruct the original sample from the corrupted one.

2.4 Variational Autoencoder

Variational Autoencoder architecture aims at a more disentagled representation of the data samples into the latent space. The implementation of this architecture requires several modifications if starting from the standard autoencoder:

- the linear layer that represent the latent space in standard autoencoder is replaced with mean and variance layers as we want to make mapping between the data samples and the latent space probabilistic.
- in order to restore the possibility to use the back-propagation for training, we need to reparametrize the latent represetation to make it differentiable.
- we need also to modify the loss function to include a regularization term that evaluates the similarity between the latent distribution and a standard multivariate gaussian (*Kullback-Leibler divergence*). We also add the β parameter which is a multiplicative constant in front of the KL term that allows us to control the trade-off between reconstruction accuracy and disentaglement of the latent representation.

As done for the denoising autoencoder, we adopt the set of hyper-parameters optimized for the standard autoencoder model. The parameter β instead is set to 2, selected by running some tests with values between 1 and 10.

Finally, training and testing is again done in the same way as in the standard autoencoder of Subsection 2.1.

3 Results

3.1 Convolutional Autoencoder

The hyper-parameters space used is reported in Table 1 and the best configuration resulting from the *optuna* study is reported in Table 2.

To verify the quality of the study in Figure 1 we report its history with all the trials and their corresponding minimum validation loss and in Figure 2 the scatter plot *Latent space dim* vs *Min. validation loss*.

Looking at the Tables 1 and 2 it can be noticed that the optimization of the hyper-parameters have maximized the degrees of freedom regarding the number of channels and of linear units while almost minimizing the

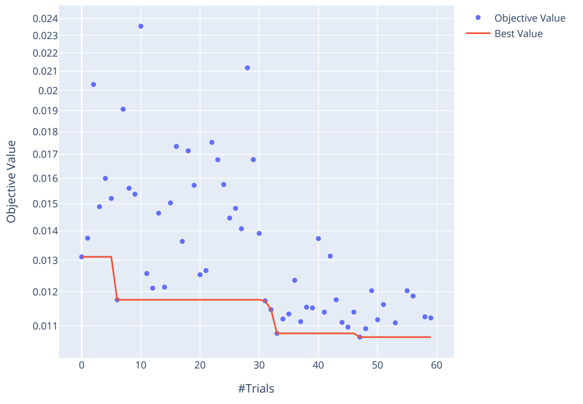


Figure 1: *Optuna* optimization history. Random sampling of hyper-parameters has been used for the first half (30 trials) of the study.

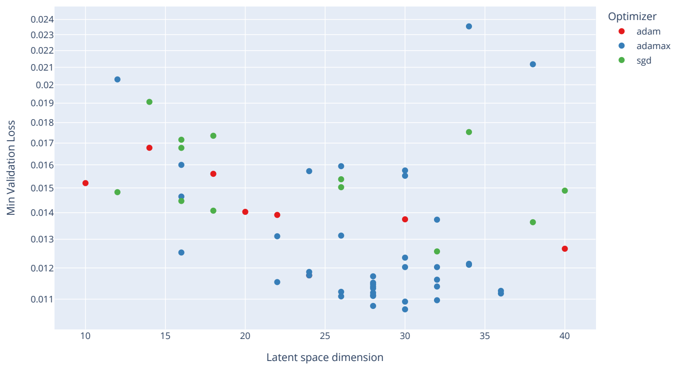


Figure 2: Scatter plot of *Latent space dimension* vs *Min. validation loss* for each trial. Different colours have been used for different optimizers.

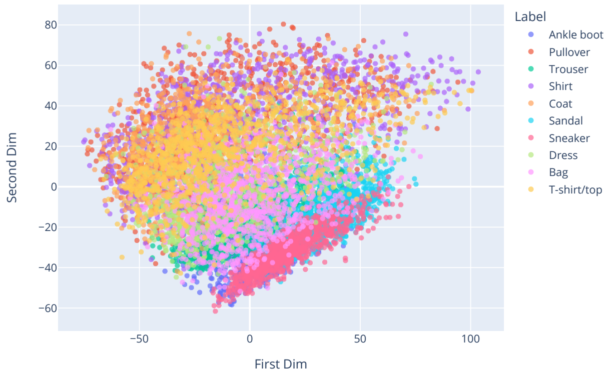


Figure 3: Visualization of the latent space encodings of the test dataset reduced with PCA for the standard autoencoder.

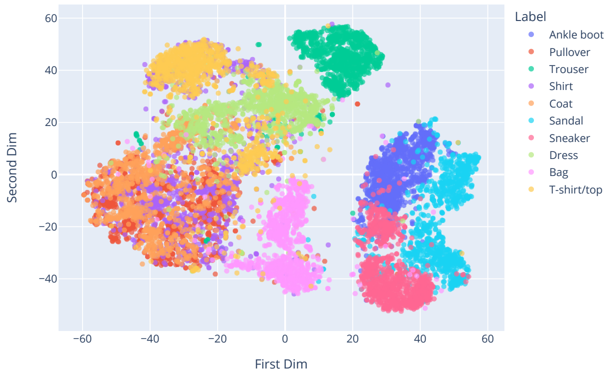


Figure 4: Visualization of the latent space encodings of the test dataset reduced with t-SNE with perplexity parameter set to 80.

L2 penalty constraint. This is expected as hyper-parameters have been chosen by looking only at the similarity between sample and reconstruction. Probably a better optimization would make use of also other criteria to choose the best set of hyper-parameters, maybe based on model complexity or time requirements. However, it must be noticed that the latent space dimension chosen by optimization is 30, which is in the middle of the provided range. Figure 2 suggests that a total of 60 trials is not enough to exhaustively explore the search space. In particular, the first part of the optimization where random sampling of hyper-parameters is applied should have been longer. Indeed, only 7 trials have used *Adam* optimizer, which can not be enough if we consider that a lot of other hyper-parameters are varying. However, a longer study would have been difficult to perform due to the time that would be required.

In Figures 3 and 4 we report visualizations of the latent space encodings produced by the model from the test dataset. Looking at the *PCA* reduction (Figure 3) we notice that the samples are not clearly separated in classes, but this can be expected considering that we are doing a projection from 30-dimensional space to a 2-dimensional one. Still, we can see that similar classes occupy the same regions. Indeed, looking at the *t-SNE* projection (Figure 4), we can see a good separation of the samples into different clusters, with objects of similar classes closer to each other.

Finally, in Figure 5 we report some examples of images generated by decoding random points of the latent space. The majority of the images are not well representing any class, meaning that the learned representation of the data distribution is highly irregular, even if we have seen in Figure 6 that the model at the end of the training is able to acceptably reconstruct an encoded image.

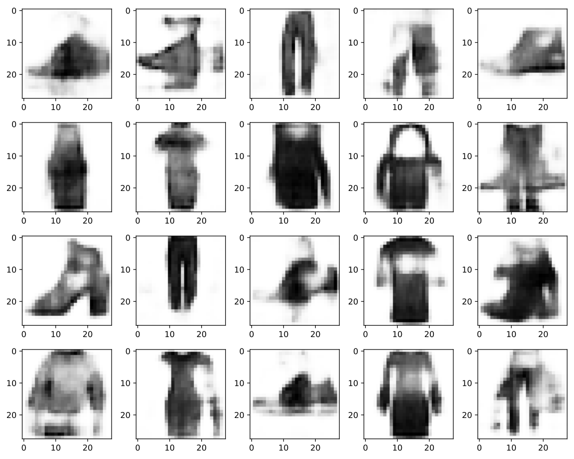


Figure 5: Images generated from the autoencoder latent space.

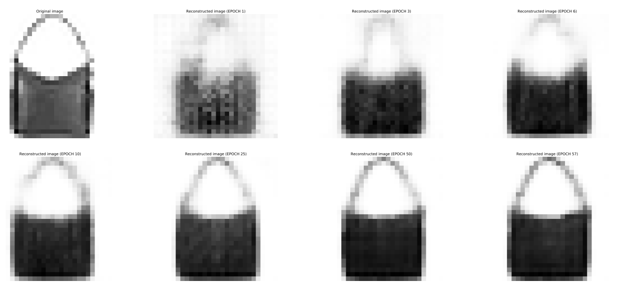


Figure 6: Grid of picture representing autoencoder ability to reconstruct a randomly selected image (top-left image) at different epochs (in top-left order at epochs: 1, 3, 6, 10, 25, 50, 57).

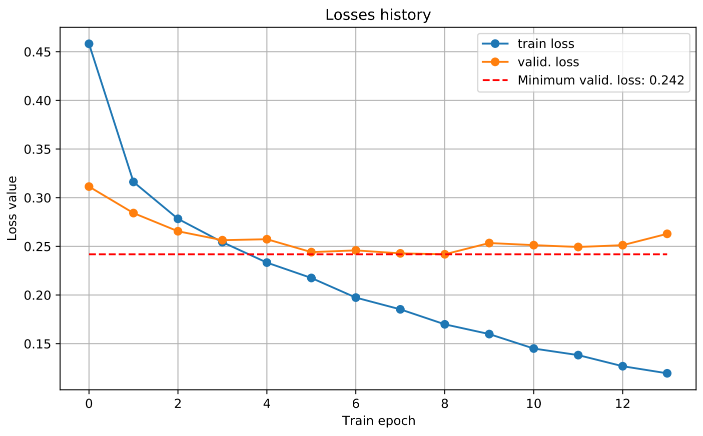


Figure 7: History of train and validation losses for the transfer-learning classifier.

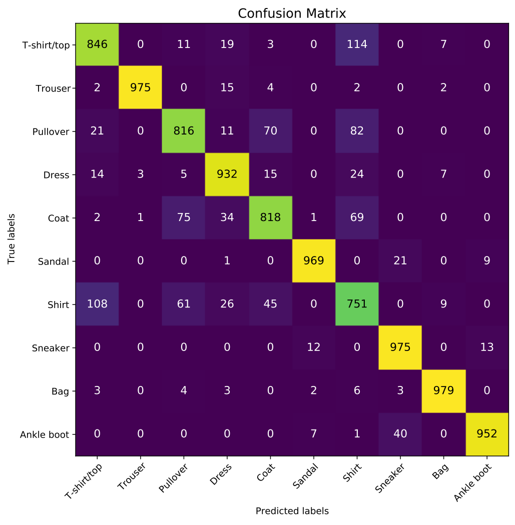


Figure 8: Confusion matrix of the transfer-learning classifier.

3.2 Transfer Learning

The learning rate suggested by the learning rate finder procedure (see Appendix) for the transfer-learning classifier is 0.0033113. In Figure 7 we report the losses history for each epoch and in Figure 8 the confusion matrix of the classification over the test dataset. Final test loss and accuracy are 0.278 and 0.901, respectively. Comparing these results with the ones obtained in the first homework (see Appendix), we can notice that the performance of the transfer-learning classifier is similar to the one of the CNN of homework 1. However, it must be noticed that the complexity of the transfer-learning model is much smaller: indeed, homework 1 CNN has 2 hidden linear layers with 1024 and 320 neurons, while the transfer-learning model has just one hidden layer with 256 units. Also, training time of the linear classifier (without the forward call of the pre-trained convolutional part) is about 2 minutes against 27 minutes of homework 1 CNN.

3.3 Denoising Autoencoder

Denoising Autoencoder losses and reconstruction accuracy per epoch are similar to the ones of the standard autoencoder. The few differences are that training convergence is slower and less stable due to the randomness of the applied transformations and that the quality of the reconstructed examples is worse, as visible in Figure 9. Also images generated from random latent codes are confused similarly as in the standard autoencoder case reported in Figure 5. Instead, a result that is worth reporting is the denoising capability test of the model (Figure 10), where we take some random samples from the test dataset, apply a corruption transformation and require

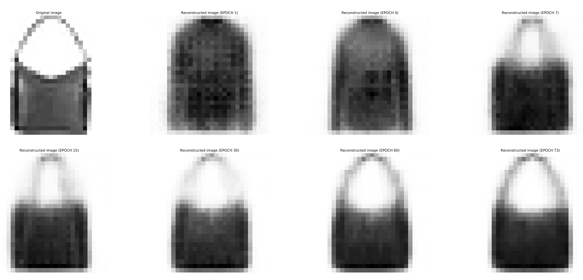


Figure 9: Grid of picture representing denoising autoencoder ability to reconstruct a randomly selected image (top-left image) at different epochs (in top-left order at epochs: 1, 4, 7, 15, 30, 60, 73). No random transformations were applied to input images.

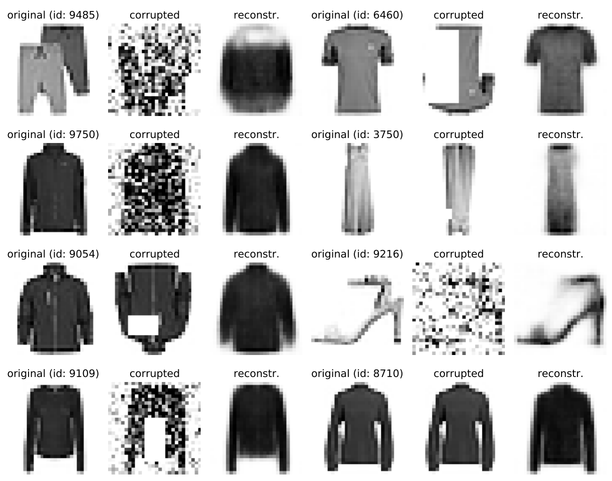


Figure 10: Denoising capability test for the denoising autoencoder model.

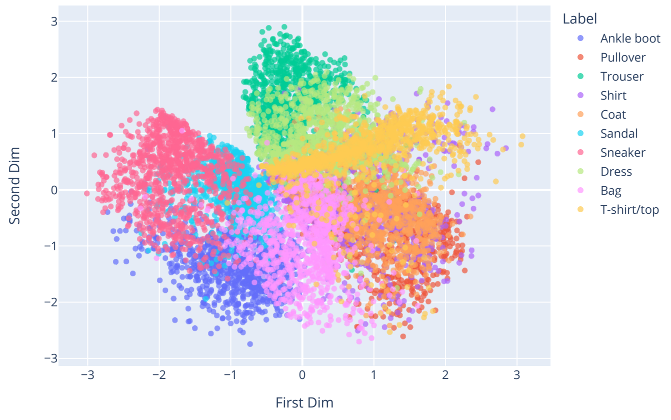


Figure 11: Visualization of the latent space encodings of the test dataset reduced with PCA for the variational autoencoder.

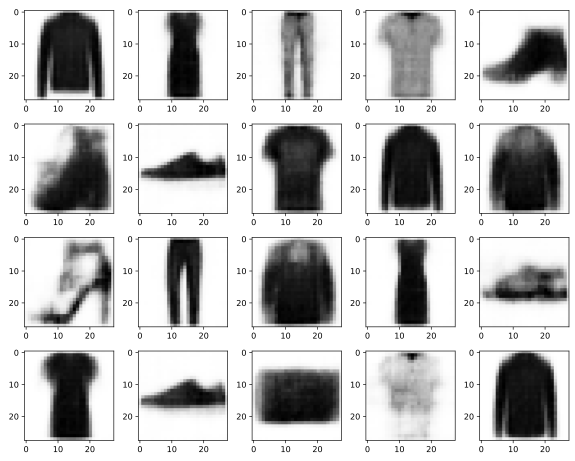


Figure 12: Images generated from the variational autoencoder latent space.

the model to reconstruct the original image. We can see that almost all the images are correctly retrieved, loosing just the finer details.

3.4 Variational Autoencoder

To compare the latent representation learned by the variational autoencoder and the standard one we report the latent space of the test dataset reduced with PCA in Figure 11. It can be seen that the latent space reports distinct clusters for different class, just partially overlapped in some cases, while the latent space of the standard autoencoder (Figure 3) is more entangled. Comparing the generated images in Figure 12 with the ones of the standard autoencoder (Figure 5) we can notice that images are more blurred and with less details, but less confused and clearly representing a class object. As expected, we have sacrificed some reconstruction accuracy to have a more regular latent space.

4 Conclusions

In this work we have explored Unsupervised Learning through the implementation of three autoencoder models: convolutional, denoising and variational. We have also tested the usage of transfer learning to implement a classifier starting from our pretrained autoencoder. The hyper-parameters optimization, done using *Optuna* library, have produced a good set of values, although it has been not exhaustive and limited by time requirements. We could have for sure obtained a better hyper-parameters configuration with more time or computational power.

Appendix

Learning rate finder (PyTorch-Lightning)

Learning rate finder functionality does a small run where the learning rate is increased after each processed batch and the corresponding loss is saved. This produces a plot *Learning rate* vs *Loss* that can be used as a guide to choose a good learning rate; the value suggested by the tool is calculated as the point with the steepest descent. Figure 13 reports the curve used to choose the learning rate of the transfer-learning model.

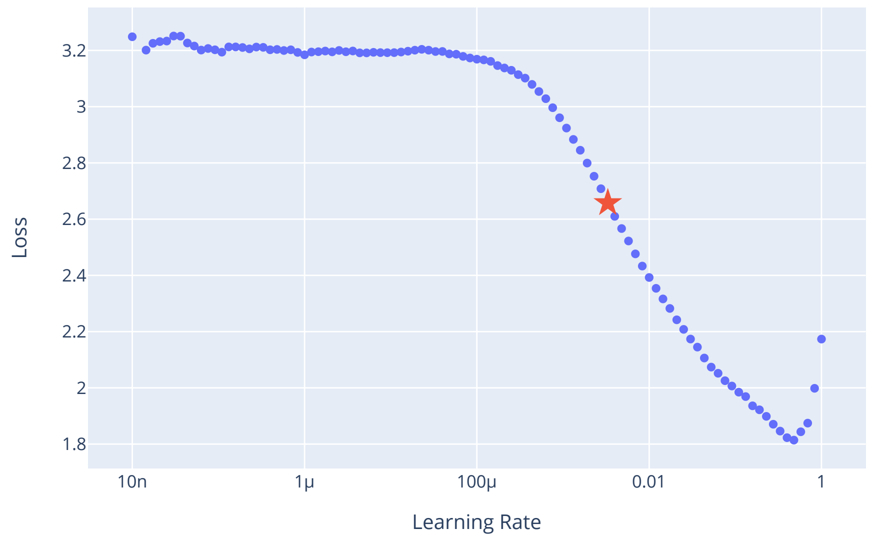


Figure 13: Image produced by the learning rate finder procedure applied to the transfer-learning classifier. The star in red indicates the chosen value.

Classification results of Homework 1

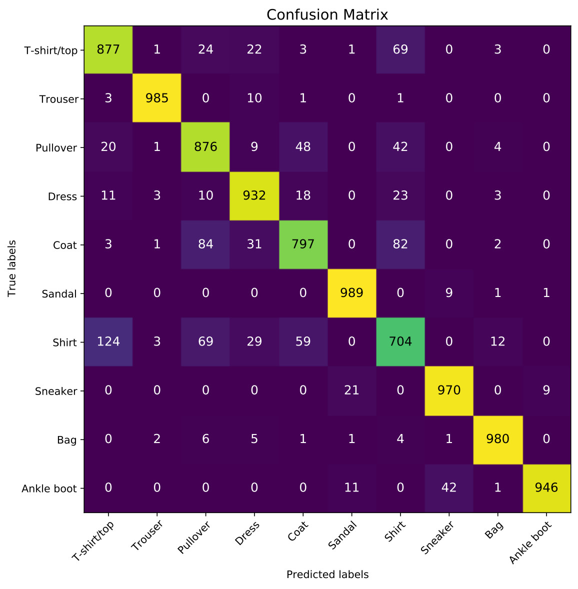
Here we report the classification results of homework 1 to be compared with the ones obtained with the transfer-learning classifier.

Name	*Value*	Name	*Value*
# of conv. layers	3	Nl1	1024
kernel sizes	[7, 3, 3]	Nl2	320
strides	[2, 2, 1]	Batch norm	TRUE
paddings	[2, 2, 1]	optimizer	Adam
channels config.	[96,64,32]	Batch size	256
learning rate	0.000566		
L2 penalty	0.000169		

Table 3: Set of hyper-parameters used with the CNN of Homework 1. Left column: parameters optimized with *optuna*; right column: fixed parameters.

Random transf.	*Test Loss*	*Test Accuracy*
FALSE	0.287	0.906
TRUE	0.259	0.911

Table 4: Test loss and accuracy for both the trained models without (*FALSE*) and with (*TRUE*) random transformations applied to input images for the CNN of Homework 1.



True \ Predicted	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	877	1	24	22	3	1	69	0	3	0
Trouser	3	985	0	10	1	0	1	0	0	0
Pullover	20	1	876	9	48	0	42	0	4	0
Dress	11	3	10	932	18	0	23	0	3	0
Coat	3	1	84	31	797	0	82	0	2	0
Sandal	0	0	0	0	0	989	0	9	1	1
Shirt	124	3	69	29	59	0	704	0	12	0
Sneaker	0	0	0	0	0	21	0	970	0	9
Bag	0	2	6	5	1	1	4	1	980	0
Ankle boot	0	0	0	0	0	11	0	42	1	946

Figure 14: Confusion matrix of the CNN model of Homework 1 (without random transformation applied to input images).

Additional *Optuna* optimization results

Here we report some additional plots regarding the *Optuna* optimization.

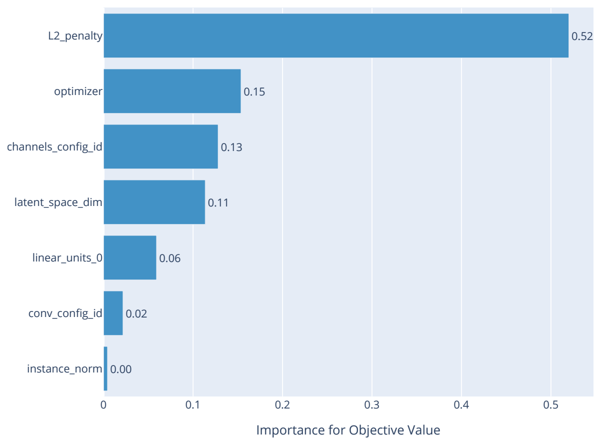


Figure 15: Importance of each hyper-parameter estimated by *Optuna library*.

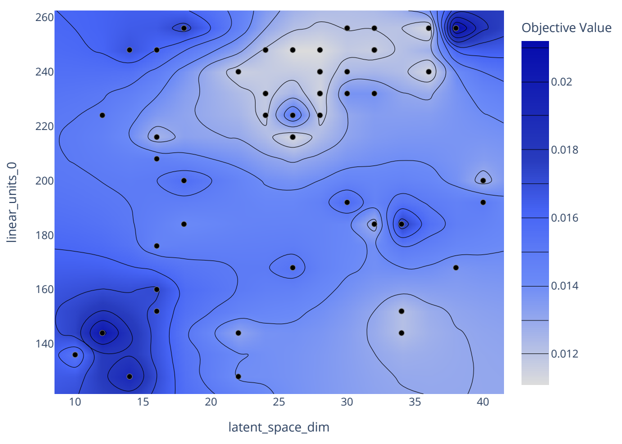


Figure 16: Contour plot of the minimum validation loss as function of *Latent dimension* and *Linear units*.

Comparisons between autoencoder models

Here we group some equivalent images produced for the different types of autoencoder models to make some direct comparisons.

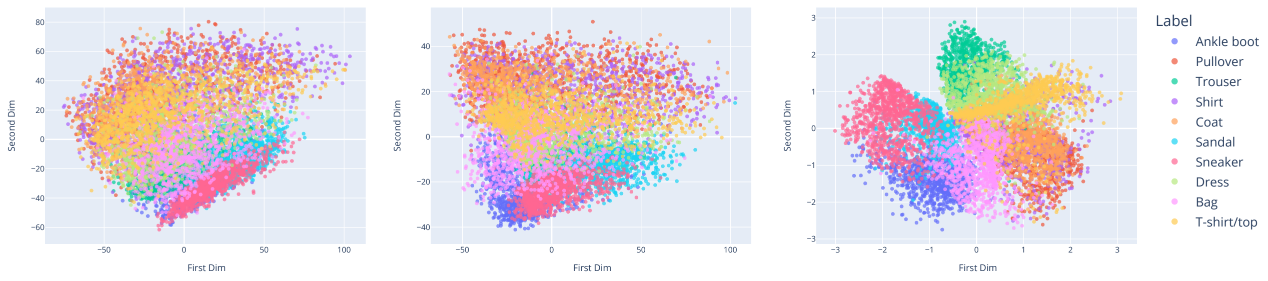


Figure 17: Latent representation of the test dataset reduced with PCA. From left to right: standard autoencoder, denoising autoencoder, variational autoencoder.

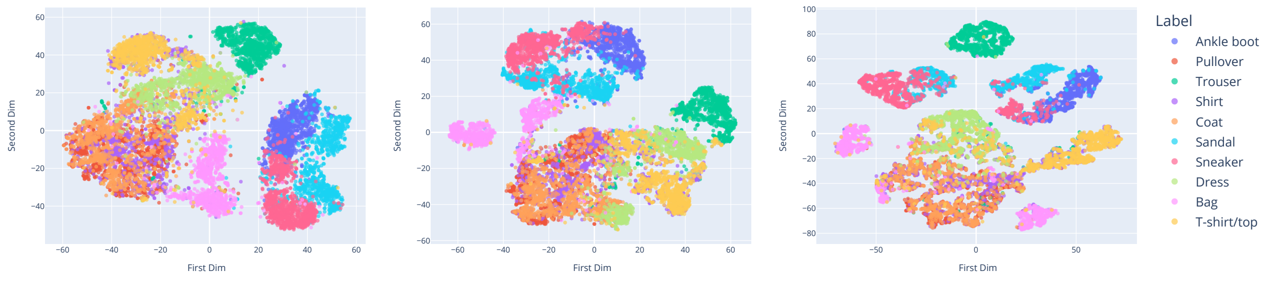


Figure 18: Latent representation of the test dataset reduced with t-SNE. From left to right: standard autoencoder with perplexity 80, denoising autoencoder with perplexity 80, variational autoencoder with perplexity 50.

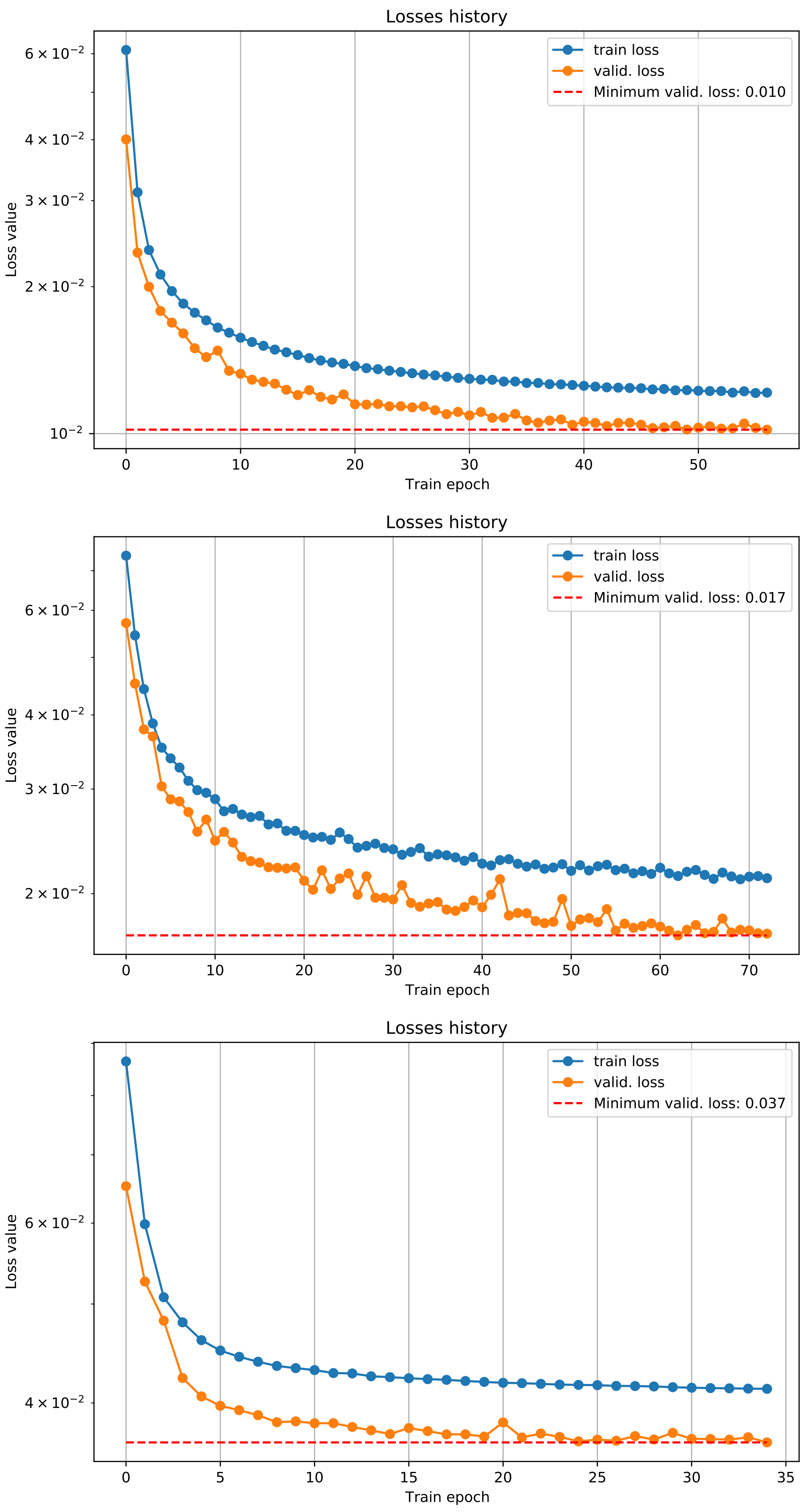
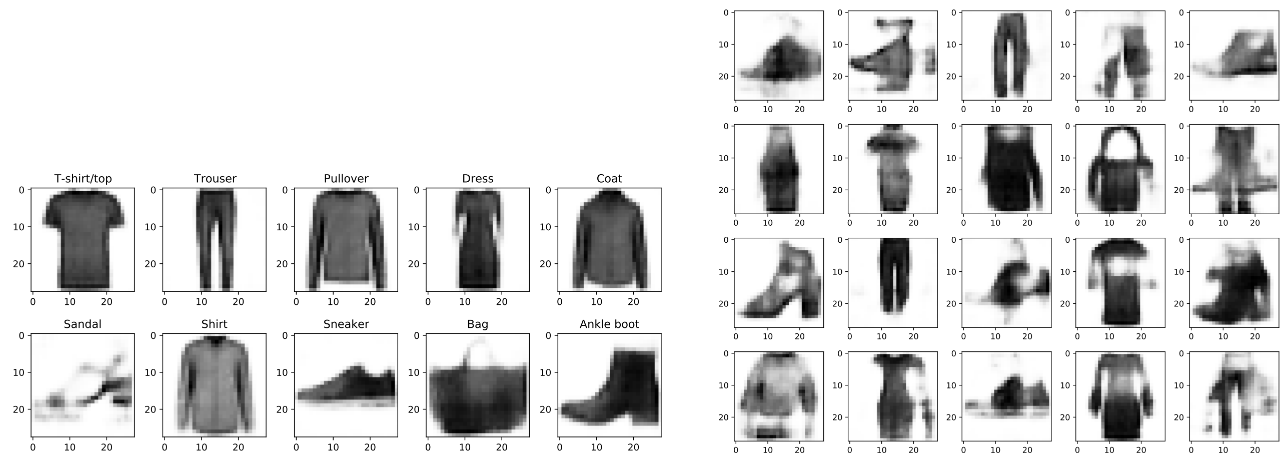
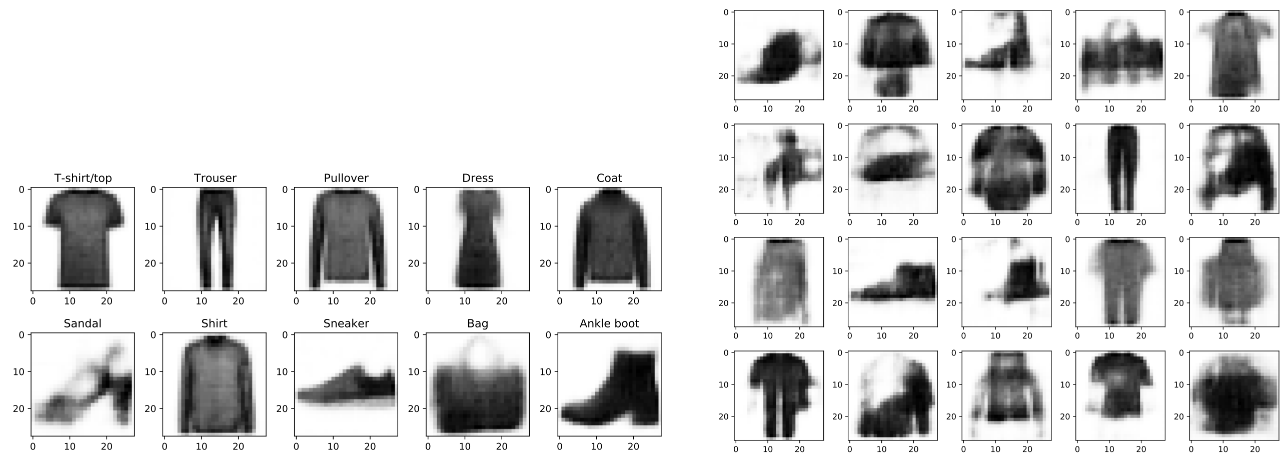


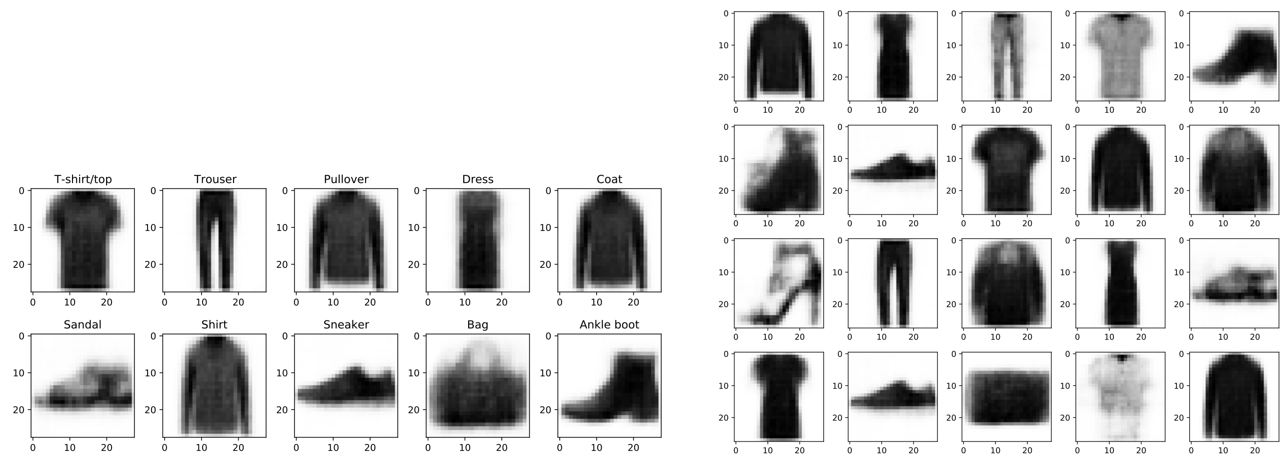
Figure 19: Losses history during training. From top to bottom: standard autoencoder, denoising autoencoder, variational autoencoder.



(a) Standard autoencoder. **Left:** centroids; **Right:** random latent codes.



(b) Denoising autoencoder. **Left:** centroids; **Right:** random latent codes.



(c) Variational autoencoder. **Left:** centroids; **Right:** random latent codes.

Figure 20: Images generated by the autoencoder models. Images on the left are produced by decoding the centroids of each class cluster in the latent representation of the test dataset. Centroids have been calculated using the labels included with the dataset. Image on the right are produced by decoding some randomly selected latent space points.