Unsupervised Deep Learning

Tommaso Faorlin (2021857)

Università degli Studi di Padova
tommaso.faorlin@studenti.unipd.it

I. Introduction

In this report we solve the second homework of the course in Neural Networks and Deep Learning. The topic is unsupervised deep learning: we will display how to learn important features from an unlabeled dataset. We are going to design and optimize Convolutional and Variational Autoencoders, have a look at their latent spaces and implement techniques of transfer learning, to speed up the training phase of supervised models. We work within the `PyTorch Lightning` framework for a high level interface, and `Optuna` for the hyper-parameters optimization.

II. Convolutional Autoencoder (CAE)

An Autoencoder is a type of deep neural network used to learn efficient codings of unlabeled data. By inserting convolutional layers inside its architecture we can talk about Convolutional Autoencoders (CAE): these are particularly suitable in applications with images, as in our case. They are usually composed of two parts, as shown in Fig. 2: the whole network tries to encode the information contained in the images that receives in input inside a lower dimensional space called latent space, without compromising the capabilities of the decoder in reconstructing the original inputs from a vector of that small space.

I. Dataset

In our case, we are going to use 28×28 black and white images of apparels, taken from the FashionMNIST database. Our training set is composed of 60000 samples, further divided into an effective training and validation sets with a 80%-20% split. Additional 10000 samples are contained in a test set for the final benchmark of the best architecture.

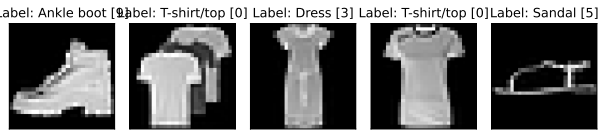

Figure 1: *Some samples coming from the FashionMNIST dataset.*

II. Implementation

As first thing, we implement with `Pytorch Lightning`, the CAE used in the laboratory and we test its operation. We report, in Fig. 2, the architecture in detail. All the layers are activated with a `ReLU`, chosen in order to avoid the infamous vanishing gradient issue. To compute the reconstruction error (difference between input of the encoder and output of decoder), we use the Mean Squared Error (MSE) as loss function. For this first trial we use a bidimensional latent space.

We train the model for 20 epochs, with `Adam` as optimizer, learning rate $\texttt{lr} = 5 \cdot 10^{-4}$ and a small L2-regularization term $\texttt{reg} = 10^{-5}$. The result we obtain in terms of reconstruction error per epoch is reported in Fig. 3. After 20 epochs we reach an overall loss of 0.0328 in validation and this is only the starting point: in the following paragraph we are going to optimize the hyper-parameters of the model. We

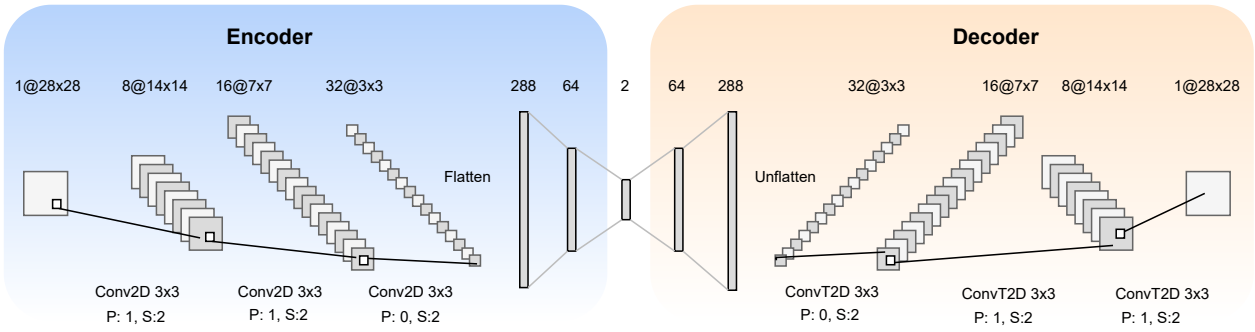

Figure 2: *Architecture of the CAE that has been used: three convolutional and two successive fully-connected layers and the mirrored equivalent in the second part. The size of the encoder output, that is equal to that of the decoder input, is the dimension of the latent space* `encoded_space_dim` $= 2$ *as an example. 'P' stands for padding, 'S' for stride. Custom drawing done with* `https://app.diagrams.net/`

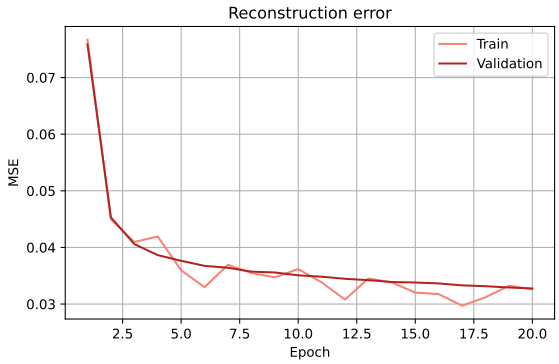

Figure 3: *Train and validation losses per epoch (reconstruction error).*

show some reconstructed samples with this Autoencoder in Fig. 4. As we can see, some information is lost in the encoding process and the images differ greatly between input and output.

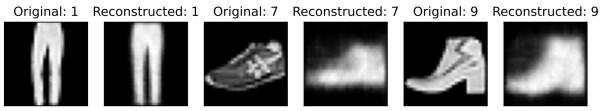

Figure 4: *Plot of three oriinal and reconstructed images (chosen at random) with the first non-optimized CAE.*

III. Optimization

III.1 Method

The model just defined and tested must now be optimized, and to do so we choose to use Optuna. We define an objective function, and inside of it the space of parameters over which to perform the hyper-parameters search:

```
1 def objective(trial: optuna.trial.Trial) -> float:
2     #OPTUNA
3     encoded_space_dim = trial.suggest_int('encoded_space_dim', 2, 50)
```

```
    opt             = trial.suggest_categorical('opt', ['SGD', 'Adam', 'Adadelta', 'Adagrad'])
    lr              = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    reg             = trial.suggest_loguniform('reg', 1e-5, 1e-4)
    #[...]
```

We notice how, by forcing the first parameter to be in that range, the size of the images is drastically reduced from $28 \times 28 = 784$ pixels to a vector of dimension between 2 and 50.

III.2 Results

The study is created with the name `Autoencoder`, saved in a database for future uses and pruned. The optimization runs over 150 models and we show some plots from the optimization in Fig. 5.

```
study  = optuna.create_study(study_name     = "AutoEncoder",
                             storage        = 'sqlite:///AutoEncoder.db',
                             direction      = "minimize",
                             pruner         = pruner,
                             load_if_exists = True)
study.optimize(objective, n_trials = 150, timeout = None)
```

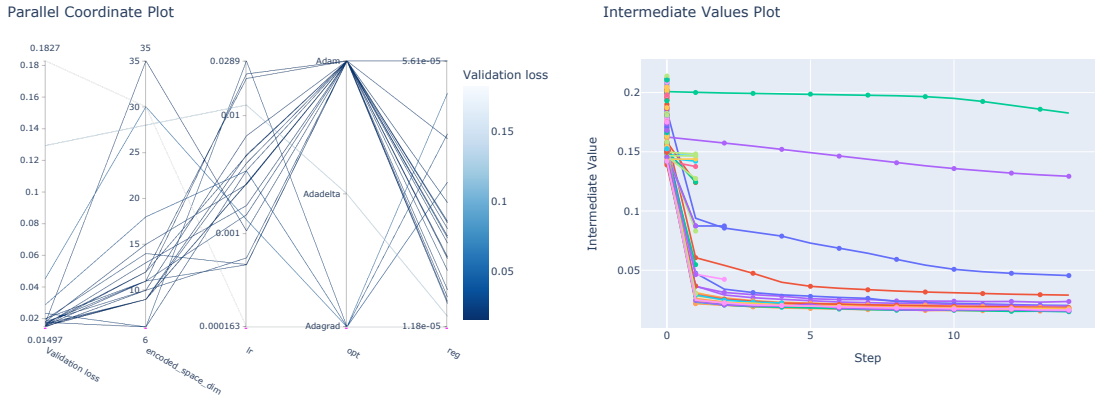

Figure 5: *(**Left**). We can observe how the best optimizer is Adam. It is interesting to note that all attempts with stochastic gradient descent as an optimizer have been pruned, since it is not reported among the optimizers. On the (**Right**) graph we show the reconstruction loss for different models and we may notice how many attempts were pruned at the initial step.*

After the optimization the best set of parameters found is reported in Table 1. We notice that the encoded space dimension is equal to 10, not the highest allowed (50), meaning that few dimensions suffice to capture and distinguish the different features from the different classes. The best Autoencoder, built on these

encoded_space_dim	lr	opt	reg
10	0.007	Adam	0.00002

Table 1: *Parameters of the best model after the optimization with Optuna.*

parameters, is achieved after a training on a maximum of 100 epochs (early stopped at 30 in the process) and amounts to a MSE loss of 0.0150 in validation. In Fig. 6 we show some reconstructed examples. We can notice an improvement in the quality with respect to what we obtained in Fig. 4

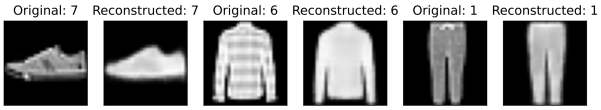

Figure 6: *Plot of three original and reconstructed images (chosen at random) with the optimized CAE.*

IV. Fine-tuning

We now fine-tune our CAE, in order to make it able to perform a supervised learning task, and compare the final results and performances with what was obtained in the first assignment. This technique is part of a more general framework called transfer learning: we store knowledge gained while solving one problem to solve a different one.

The architecture of the model is unchanged until after the encoder. Instead of the decoder in fact two fully connected layers are docked. The former has an input of dimension equal to the latent space and 64 output units, while the second has 64 input units and 10 outputs (as the number of classes). The first layer is activated with a `ReLU`, while the second with a `LogSoftmax`. We choose the `NegativeLogLikelihood` as loss function. The parameters of the `encoder` are freezed to be the best found up to this point, while the one of the `fine_tuner` are trained. The accuracy of this model is 83.3% on the validation dataset and 83.3% on the test set. By training an entire Convolutional Neural Network we obtained accuracy of 90.5% and 90.0% on validation and test set respectively. The new results are slightly worse but still acceptable, even considering the fact that the training of two layers fully connected took much less computational time and resources than to train an entire CNN.

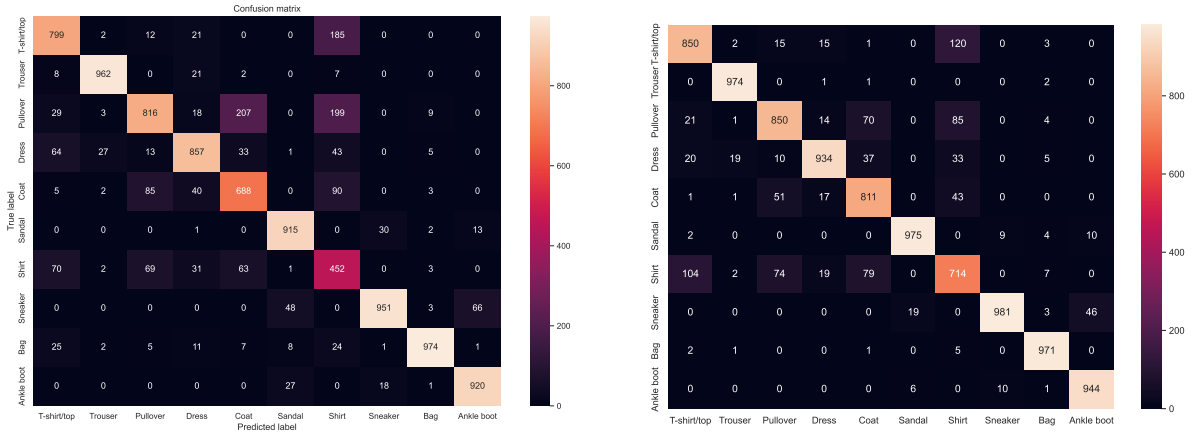

Figure 7: ***(Left)****. We report the confusion matrix on the predictions of the test dataset, obtained with the transfer learning technique implemented in this report (test accuracy:* 83.3%*). (**Right**). Confusion matrix obtained in the first assignment with a heavier CNN, always on the test dataset (test accuracy:* 90.0%*). The new result is slightly worse and we observe the model's difficulty in assigning the right label to the shirts and distinguishing coats from pullovers.*

V. CAE Latent space analysis

We now want to explore the latent space of the CAE, and see how information are compressed to a 10 dimensional space. Studying spaces with a dimension higher than 3 is a highly non-trivial task, so we make use of two powerful statistical methods that have been developed to make high-dimensional spaces understandable: PCA and tSNE. The former is a linear dimensionality reduction technique that reduces the dimensionality of data to a set of vectors known as Principal component, which captures the dimensions with higher variability. The latter instead, t-distributed Stochastic Neighbor Embedding is a non-linear technique that, in simple words, associate to each data point a correspondent point in a two

or three-dimensional vector space, trying to display closely points with similar features and separate the different ones. So, after having encoded all the test samples in the 10-dimensional latent space of the CAE, we show the PCA and the t-SNE for a bidimensional encoded space:

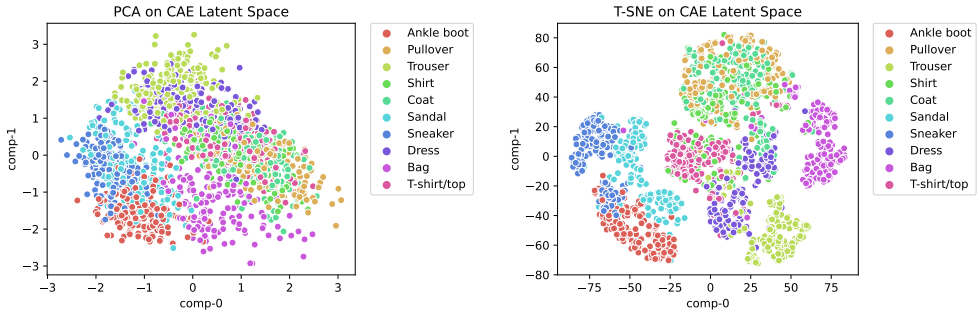

Figure 8: *In both the graphs, we display the first 2000 representation of samples in the test dataset. (**Left**). PCA projection of the latent representations of FashionMNIST apparels. The different garments are not clearly divided into clusters and it is not easy to interpret this data (**Right**). The t-SNE algorithm clusters together the representations of similar images without any label. It is interesting to notice how a 'macro-cluster' can be identified on the left and contains all the foot wears (ankle boots, sandals, sneakers), while on the top pullovers and coats are mixed together since they are very similar.*

III. Variational Autoencoder (VAE)

I. Implementation

I.1 Method

The substantial difference between a CAE and a Variational Autoencoder lies in the structure of the latent space. In fact, the output of the first part of a normal Autoencoder is a point in the latent space, while for the Variational version are the parameters of a multivariate Gaussian distribution $\mathcal{N}(\mu, \Sigma)$, that in our specific implementation comes with a diagonal covariance matrix. Practically speaking, we keep the first convolutional part unchanged and we address its output to two fully connected layers: one contains the entries of the mean vector $\boldsymbol{\mu}$, while the other accounts for the logarithm of the variances $\log \boldsymbol{\sigma}^2$. The latent vector $\boldsymbol{s}$ will be then sampled from this distribution as:

$$\boldsymbol{s} = \boldsymbol{\mu} + \boldsymbol{\varepsilon}\boldsymbol{\sigma} \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{1}). \tag{1}$$

This equation is also called *reparametrization trick*: since the sampling of the latent vector is a random process, hence not differentiable, we make $\boldsymbol{s}$ differentiable with respect to network's parameter, allowing a backpropagation scheme to work properly. The loss function is composed by the reconstruction error, the Mean Squared Error (MSE) as above, and by a regularization term, i.e. a Kullback-Leibler divergence that measure the statistical distance between the actual distribution and $\mathcal{N}(\mathbf{0}, \mathbf{1})$:

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 + D_{KL}(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \| \mathcal{N}(\mathbf{0}, \mathbf{1})) \tag{2}$$

In this way, when we minimize this quantity we are actually minimizing the reconstruction error, with a constraint fixed by the KL divergence to have the latent space distribution as close as possible to $\mathcal{N}(\mathbf{0}, \mathbf{1})$.

I.2 Results

The model is defined with a latent space dimension of 10, is trained on 100 epochs with `Adam` optimizer, `lr` $= 0.001$ and no regularization. We obtain a convergence in 40 epochs with early stopping. We show some reconstructed samples in Fig. 9. The quality is slightly lower compared to the one of the CAE (Fig. 6) but a small change in the input is now probabilistically mapped to a small change in the latent space, i.e. the model is more robust to perturbations.

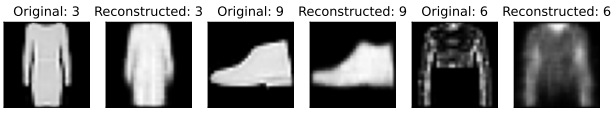

Figure 9: *Plot of three original and reconstructed images (chosen at random) with the VAE.*

II. VAE Latent space analysis

The same considerations made in Sec. V are valid, and we plot PCA and t-SNE results on a bidimensional embedded space for the VAE's latent space. In the end, the method that is able to cluster better the different classes is the t-SNE, both with a Convolutional and a Variational Autoencoder. However, we must point out that it is not simple to interpret a projection of the data in a lower dimensional space. The results for the VAE follow:

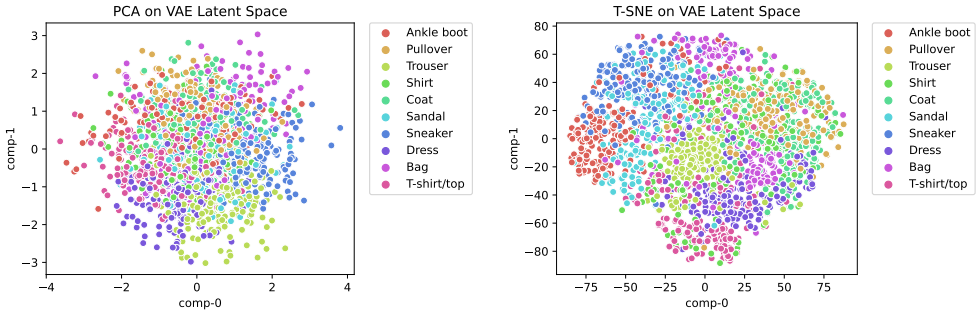

Figure 10: *In both the graphs, we display the first 2000 representation of samples in the test dataset. (**Left**). PCA projection of the latent representations of FashionMNIST apparels. (**Right**). Also in this case, we can distinguish clusters with a t-SNE algorithm even if they are not as well separated as in the first case. This could be related to the greater robustness of VAE*

III. Samples generation

We perform samples generation in two ways. At first, we select a random pair of images belonging to two different classes, and we see how the model is able to smoothly pass from the first to the second. Two sweeps are presented in Fig. 12. As a last thing, we sample a 10-dimensional vector of numbers in $[-3, 3]$,

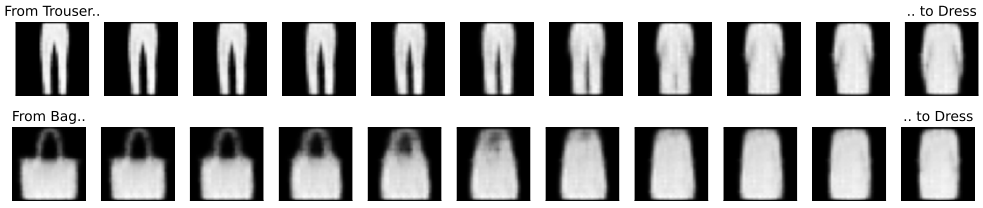

Figure 11: *Plot of two sweeps with the VAE.*

and we decode the corresponding image. This is the result we obtain with random sampling:

Figure 12: *Images randomly sampled from the latent space.*