UNIVERSITÀ DEGLI STUDI DI PADOVA

Unsupervised Deep Learning

Ziliotto Filippo

November 30, 2021

In this homework it is requested to solve several unsupervised learning tasks on the $FashionMNIST$ clothes dataset. For this purpose, a Convolutional Autoencoder was built, tuned with the optimal set of hyperparameters thanks to *optuna* library. Finally it was tested on the rest 20% of the dataset. Also the latent space was explored after the data encoding. Moreover, some transfer learning was performed in order to solve a supervised classification problem, exploiting the encoder network previously trained. Finally, the second notebook shows how to implement both a Variational Convolutional AutoEncoder (with a varibational parameter $\beta = 2$ to have more disentagled representations) and a GAN.

1 INTRODUCTION

[1] The $FashionMNIST$ dataset consists of 28Ö28x1 pixels, grey-scale, images originally labeled to perform supervised learning. However, one might want to neglect labels and implement different networks able to extract features and building a compact internal representation of data, hence performing unsupervised learning. In this way, one is also able to generate data by inspecting the latent space where features are efficiently encoded by the network, in either a "punctual" way or through a some probability densities. The aim of this homework is therefore to implement an AutoEncoder and exploring its latent space, eventually also testing it. Later on, exploiting the learned features the network was able to (using transfer learning) train some fully connected layer to perform supervised tasks. Finally, adding "continuity" to the latent space, a Variational AutoEncoder was implemented. In addition, also a GAN is trained, in order to generate samples which are resembling as much as possible the the original dataset.
One should recall that unsupervised learning generally allows to reduce the dimensionality of problems, thus permitting to efficiently store the most important features in less "space". However, these tasks are heavily computational demanding and do not allow to infer casual relationships between variables, but only correlation due to the intrinsic nature of this learning process.

2 CONVOLUTIONAL AUTOENCODER

An AutoEncoder is a neural network whose aim is to extract features from a specific domain, which in the Convolutional case are images, and, through the encoder, encode them in a latent space whose optimal dimensionality is to be found. Starting from the latter, the encoder part has to be trained to reconstruct

[1] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

as better as possible the original data. Usually, encoder and decoder architectures are symmetric w.r.t. each other, but this is not a hard constraint. Namely, for the Encoder:

1. **First Convolutional Layer** : having one channel in input, n channels as output, kernel 4×4, stride 2 and padding 1. Activation function is ReLU.
2. **Second Convolutional Layer**: having n channels as input, $2 \cdot n$ channels as output, kernel 3 ×3, stride 2 and padding 1. Activation function is ReLU.
3. **Third Convolutional Layer**: having $2 \text{ů} n$ channels as input, $4n$ channels as output, kernel 3 ×3, stride 2 and padding 1. Activation function is ReLU.
4. **Flattening First Linear Layer**: with $3 \cdot 3 \cdot 4 \cdot n$ units and ReLU activation function. Note as input image is reduced to be 3 ×3.
5. **Second Linear Layer**: with 64 units, finally "encoding" the input to a n_{dim} latent space.

The $n_{channles}$ hyperparameter was first set to to 8 as in the lab practice and then tuned to find the optimal one. Conversely for the Decoder, we omit to write the parameters for every named layer being the same ones as before, taking care of obviously exchanging the respective roles of output and input:

1. **Second Linear Layer**: accepting as input the variables encoded in the n_{dim} latent space.
2. **Flattening First Linear Layer**
3. **Third Convolutional Layer**
4. **Second Convolutional Layer**
5. **First Convolutional Layer**: but this time having as activation function a Sigmoid, since pixels are considered "valid" only when in $[0,1]$ range.

[2]For this task a $MSELoss$ function was used, and after training the following trend of reconstruction loss (see Figure 2.1) was obtained. Moreover the output resembles better the data if we train for at least 20 or 30 epochs (see Figure 2.2). Note as the input images were previously converted to tensor but it was not normalized with the related function because it seems the autoencoder learns better to "clusterize" the input data if the single training pixels have very different values and not only between 0 and 1. Also data augmentation was not performed since the risk of overfitting is low for this kind of unsupervised learning, besides we want some kind of overfitting since we want to recompute the initial data.

[2]All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

2.1 Optimization

Due to the large dataset and in order to evaluate model accuracy, we split the training data in 80% – 20% to perform CV. Using optuna we perform some hyperparameter optimization for a maximum of 5 epochs, taking care of pruning not promising trials. As in the previous homework, it has been chosen this library to exploit its Bayesian sampling procedure, which allows better performance and efficiency. It should be noted that in general optuna results may very since the training computation is huge. Having some serious GPU's would allow to better exploit this library.
The metrics used for the task is the validation loss, and the aim is its minimization through the objective value. The overall number of trials is 100, which has taken a really long time (runned on Colab). Some interesting plots can be found below (see Figure 5.4, Figure 5.3, Figure 5.1). The optimal set of hyperparameters found is:

Hyperparameters	**Optimal value**
$\mathrm{N}_{channels}$	11
latent_{dim}	9
Optimizer	$Adam$
Learning rate	0.0091
weight decay	$\sim 1e-05$

Table 2.1: Optimal hyperparameter values obtained with the *Optuna* library. They all minimize the objective function of the library and was trained on the GPU of Colab.

The performances achieved in reconstructing images exploiting such architecture can be seen in Figure 2.2, and the network seems to be working properly.

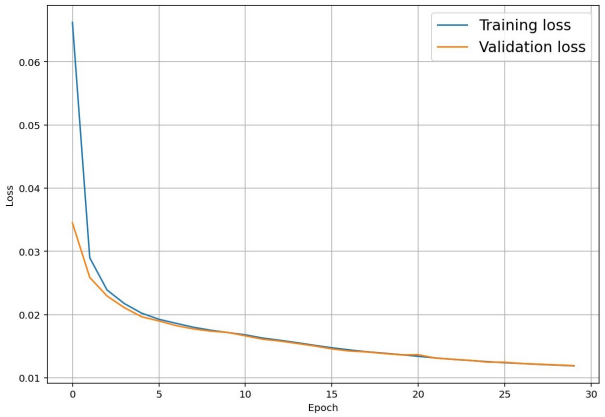

Figure 2.1: Loss functions for the standard autoencoder. We see that the training takes about 10 epochs to somehow stabilize.

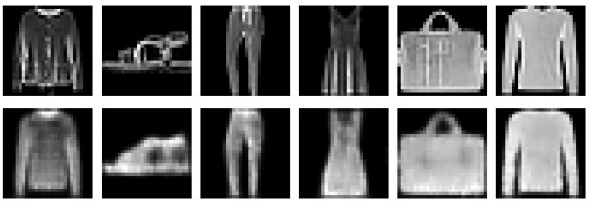

Figure 2.2: Output of the autoencoder after 30 epochs of training. We see how the network learns to correctly encode the data, hence the output is clearly very similar to the input.

2.2 Latent space exploration

Let inspect the latent space and see how variables have been encoded by the Convolutional AutoRncoder. In order to make it visualisable by a human, we exploit some powerful algorithms to reduce the dimensionality of such space:

- **PCA**: Prinipal Componet Analysis is useful to understand how many are the dimensions that retain the most variance of the data. In this case just half of the latent space dimensions are enough to retain at least 90% of the variance. In the end PCA was not used because of the better efficency in clustering the data with t-SNE.
- **T-SNE**: t-Distributed stochastic Neighbor Embedding is a non-linear transformation that allows to reduce the data dimensionality, via the assignation of larger probabilities to pairs of objects that might share features (e.g. are “close”) in the original space. Finally it reduces the dimensionality to the one requested, trying then to minimize the Kullblach-Leibler’s divergence between the pdf over pairs constructed in the new low-dimensional space and the original one. In this way, points that were sharing features in the original space are now clustered, thus (visually) close in the new space. Indeed its effects can be clearly told in Figure 2.3, where the network form clusters that are related to the type of clothing, (e.g. shoes,boots are near in space while trousers are very far from bags).

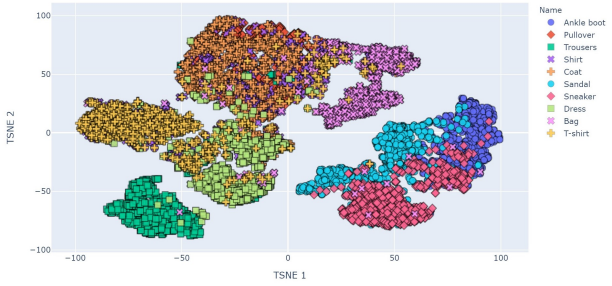

Figure 2.3: T-SNE dimensionality reduction. Here the encoded latent dimensions were 9. We can clearly see that the network is able to clusterize the data.

Finally, random samples generated from the latent space can be shown in Figure 5.2. Some clothes can be clearly recognized, while others are simply sketches. This is due to the fact that the sampling is random for every dimension so maybe the sampled points belong to a part of the space where zero "cluster" lay.
[3]

2.3 Fine-Tuning

We want now, through transfer learning technique, to perform the fine tuning of a network thus making it able to perform classification task, i.e. supervised learning. Practically, one may want to exploit the variables encoded in the latent space by the best encoder and attach some fully connected layer to it. In this way, the classifier is able to take advantage of the encoded representations which are performed by

[3] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

the encoder, eventually sparing computational effort since there is no need to train a full Convolutional Neural Network. In this assignment, it was chosen to stack a single fully connected layer with no hyperparameter optimization (due to good performace already achieved by the standard implementation), in addition to the input and output layers. Note as the input one must have a number of units that is equal to the dimensionality of the latent space, while the output layer is constrained by the nature of the problem (i.e. it should output only $28 \cdot 28 \cdot 1$ images). Moreover, ReLU activation function was used. The metrics used is the validation loss, computed through the standard Categorical $CrossEntropy$ criterion.

With the network performances being the one in Figure 2.4. Actual average overall accuracy is 86.3% not as good as in the previous homework, which turned out to be 88.3%, though being still acceptable. However, the convergence was reached much faster, even though performing hyperparameters optimization. The computational is indeed about 3 to 5 times faster. [4]
The network architecture implemented is the following:

1. **First Convolutional Layer** : with 1 channel in input, being the image in grey scale, 8 as output, kernel 4×4, padding 1 and stride 2, in order to reduce the dimensionality of the image to 14×14. The activation function is a ReLU.
2. **Second Convolutional Layer**: with 16 channels in input, 32 channels as output, kernel 5×5, and stride 2, in order to reduce the dimensionality of the image to 5×5. The activation function chosen is the ReLU.
3. **Flatten layer** : Flatten layer in order to construct a normal feedforward neural network structure.
4. **First hidden layer**: : consisting of 128 units. The activation function is the ReLU.
5. **Output Layer**: consisting of 10 units
6. **Output Activation**: Softmax in order to have a "probability" like output predction class.

No Pooling layer was used, instead it has been chosen to exploit the stride parameters proper of Convolutional layer in order to reduce the dimensionality of the problem, thus diminishing the computational effort needed for the training. As before, the number of output units is constrained due to the nature of the task. The loss used is the CrossEntropyLoss, moreover since the training took much more time rather than the previous task, it has been implemented a EarlyStopping class which eventually stops the network in the case some metrics (e.g. validation loss) does not improve for a number of iterations As before in the definition of the learning rate a scheduler has been added to better lower the validation loss.

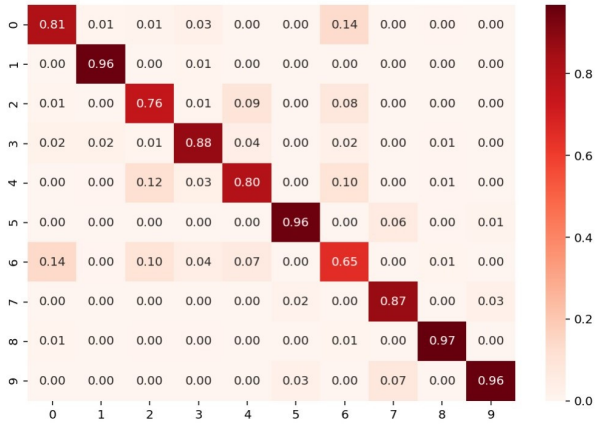

Figure 2.4: Fine tuning network performances. Overall accuracy is ~ 86%

3 Variational Autoencoder

A further and different implementation for AutoEncoders is the Variational AutoEncoder. Differently from before, the mapping between visible and latent spaces is now probabilistic, thus allowing to generate samples coming from the latent space in a "smoother" way. Practically, the encoder output is not a point in the latent space any more, rather a probability distribution which for simplicity is often chosen to be a multivariate Gaussian, namely a vector $(-\vec{\mu}, -\vec{\sigma})$, with the Covariance matrix having only diagonal terms. The loss function now includes two terms, one that measures the MSE accuracy reconstruction, whereas the second one being

[4] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

the $Kullblach-Leibler$ divergence between sample coming from the probabilistic latent space and the above mentioned Normal distribution:

$$Loss = \|\vec{x} - \hat{x}\|^2 + KL\Big(\mathcal{N}(\mu, \sigma^2), \mathcal{N}(0,1)\Big)^2 \quad (3.1)$$

where $\vec{x}$ is the input and is the reconstructed input. However, since sampling happens to be a random process, thus not differentiable and not allowing backpropagation for learning, we exploit the so called reparametrization trick, where latent vectors $\vec{z}$ are reparametrized in such way to be sampled according to:

$$\vec{z} = \vec{\sigma}_x \cdot \vec{\xi} + \vec{\mu}_x \quad (3.2)$$

Where $\vec{\xi}$ is sample from a normal distribution (0,1). The sampled outputs visible in Figure 3.1, where we smoothly vary the variables belonging to latent space, are surprisingly good.

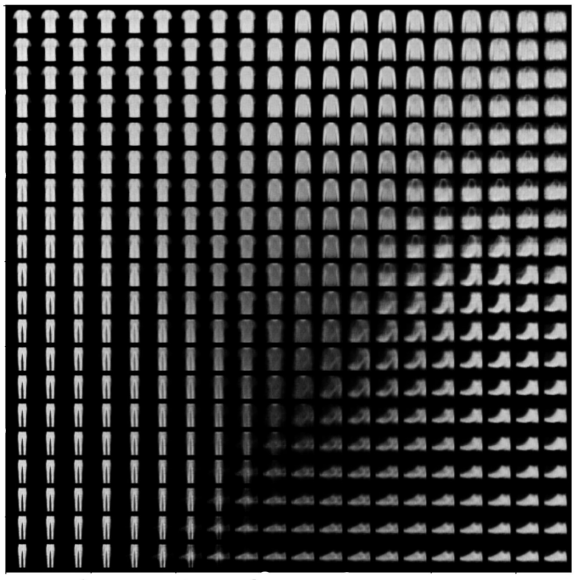

Figure 3.1: Variational Autoencoder. The plot is produced after adding a $\beta = 2$ term for more disentagled representations. We see how the distribution learned changes in space.

Figure 4.1: GAN network produced images. We see how the images are not perfect but still the output looks very reasonable.

4 Generative Adversarial Networks

Generative Adversarial Networks have been introduced to somehow specifically weight some portions of input, which might have high saliency in its semantic, though having a small MSE loss when "missed". The key idea is to build a good generative model of the data whose task is to fool a supervised classifier which must tell whether the image has been artificially generated or is coming from the original dataset ($BinaryCrossEntropy$ loss function). The input for the Generator is some noise vector Normally distributed (whose size is 64), whereas the input for the Discriminator is a $28 \cdot 28$ image, which is flattened to a 784-unit Layer.
The evolution of samples generated by the network can be inspected in the avi files attached to the repo for this homework: as expected, initially the Generator is badly performing, increasing the quality of generated samples as time passes by thus being able to efficiently fool the Discriminator. Also in the appendix figure we see how in this implementation the GAN generator is learning quite well but the discriminator is being fooled less times than expected. This

is probably due to the difference in the parameters number for the two models.[5]

5 APPENDIX

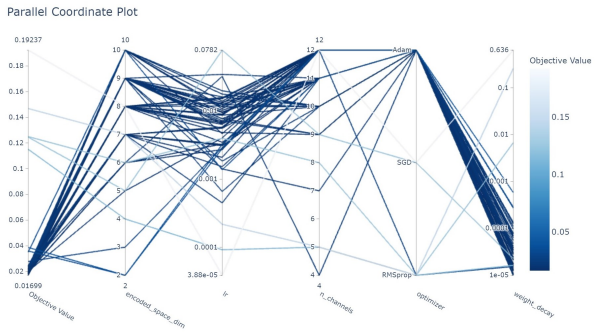

Figure 5.1: Parallel coordinate Hypeparameters, related to the optimization done in the autoencoder network.

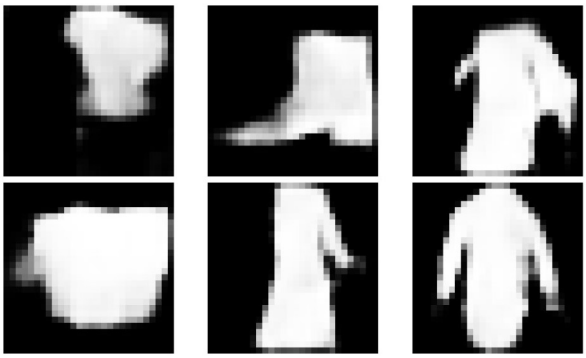

Figure 5.2: Generated samples from the autoencoder latent space. We see some clear shapes and some strange figures due to the sampling in zero "cluster" regions.

[5]All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

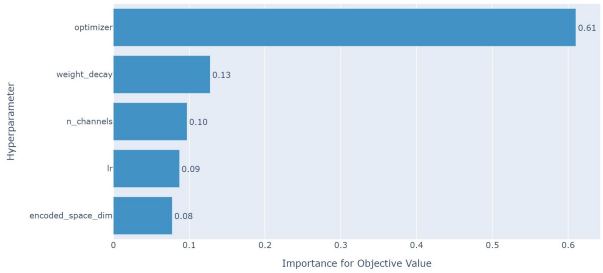

Figure 5.3: Importance of the single Hyperparameter during the training. Results may vary with different optimization due to the large computational time required.

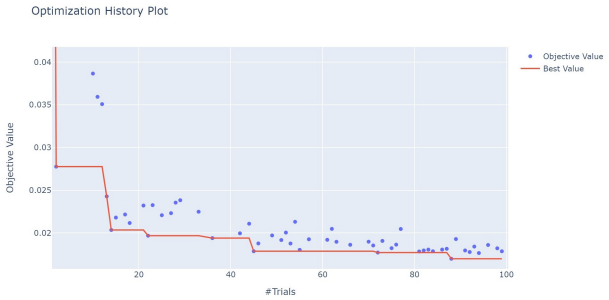

Figure 5.4: *Optuna* objective function minimization. We see how in general a good value is achieved in the first trials.

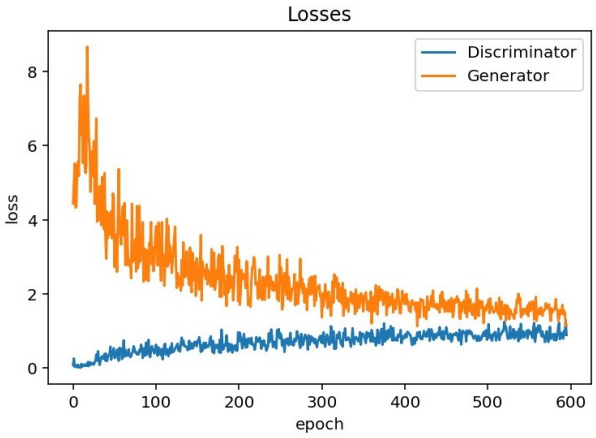

Figure 5.5: Loss function for the Generator and the Discriminator for the GAN network. We can see how one decreases while the other increases.