NNDL: Homework 2
Unsupervised Deep Learning

Alessandro Lovo

January 27, 2021

1 Introduction

This homework will deal with unsupervised deep learning applied to MNIST dataset of handwritten digits. The focus will be on different architectures of convolutional autoencoders for both classification and image generation problems. In the end also Generative Adversarial Networks (GAN) will be tested.

1.1 General framework

Network architectures and training procedures rely on a framework of python classes that unfolds as follows:

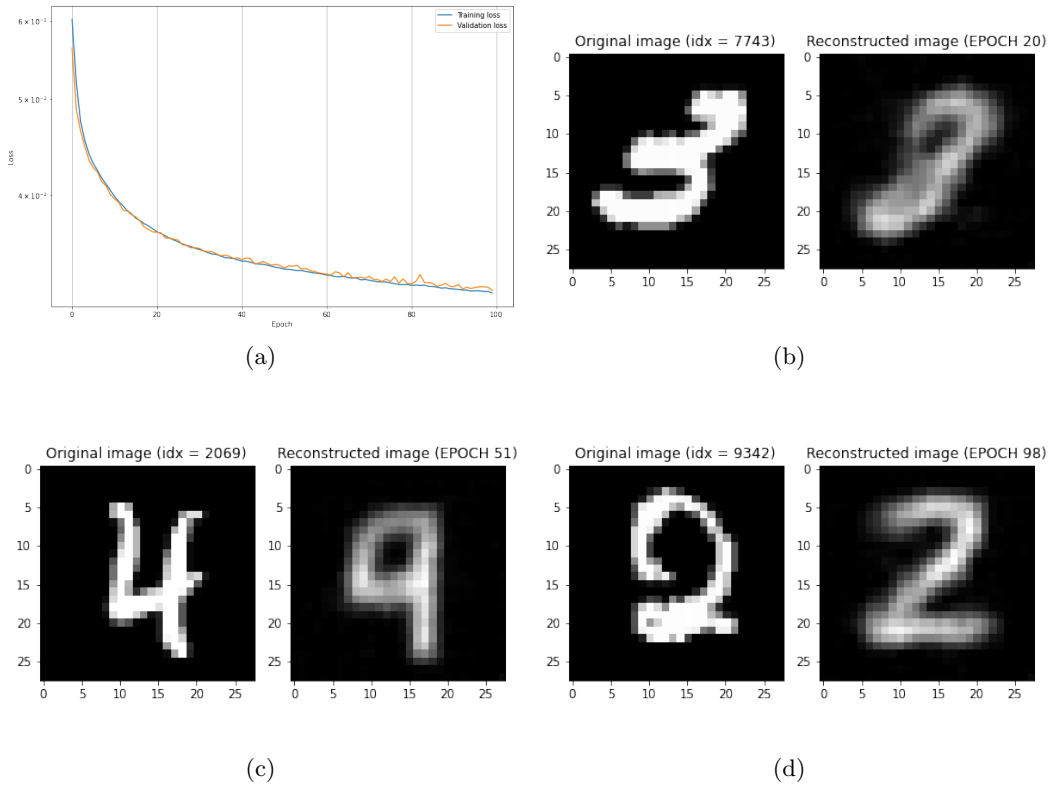
- **Encoder**: class inheriting from *torch.nn.Module* that contains the specific architecture of the first part of the autoencoder: it consists in a block of convolutional layers followed by one of fully connected layers. It receives as input a 28 by 28 pixel gray scale image and outputs N_{latent} real numbers.
- **Decoder**: counterpart of the encoder: it takes as input N_{latent} real numbers, passes them through a block of linear layers and one of transposed convolutional layers outputting a 28 by 28 gray scale image.
- **Evolver**: class for handling the training and validation of a general list of concatenated networks, in particular here the list is a *Encoder* followed by a *Decoder*. In this class there is a check at the end of every training epoch to interrupt the learning process. To implement early stopping one just needs to inherit from the *Evolver* class and specify that check condition. In particular the learning process stops if the validation loss isn't decreasing after *patience* number of epochs.
- **KFoldCrossValidator**: class for performing k fold cross validation on a particular set of hyperparameters.

2 Convolutional autoencoders

For simplicity I restricted the hyperparameter space to symmetric autoencoders, i.e. the decoder is a 'mirrored' version of the encoder. For this reason I will point out only the encoder hyperparameters.

2.1 Basic solution

The structure of this autoencoder is very similar to the one seen in the Lab practices: 3 convolutional layers with number of channels [8, 16, 32], kernel sizes of 3, strides of 2, paddings [1, 1, 0]; then 2 linear layers with 128 and 16 neurons and finally an output layer with $N_{latent} = 4$ neurons. The activation functions between all layers are ReLU and this architecture is trained for 100 epochs with the Adam optimizer with learning rate 10^{-3} and weight decay 10^{-5} with a batch size of 256 and a 80%-20% train validation split of the 60000 training samples. The loss function is the mean square error (MSE) between the original and reconstructed image.



(a) (b)

(c) (d)

Figure 1: Behavior of training and validation loss during learning and some examples of original versus reconstructed images (at epochs 20, 51 and 98).

As we can see from fig 1 there is no definite convergence, as the loss slowly keeps decreasing, moreover training and validation loss are virtually identical, showing there is no overfitting.

2.2 More advanced methods

In the basic example we witnessed a pretty slow convergence to pretty mediocre results: to improve both speed and reconstruction error one needs better architectures and training methods. First of all instead of the MSE loss it is better to use the Binary Cross Entropy (BCE) loss; then I also implemented early stopping, checkpointing the nets back to when the validation loss was at its minimum if after *patience* epochs the validation loss didn't reach a new minimum. At this point I tried more advanced strategies:

- **Pruning**: every *prune_every* epochs the *amount* fraction of the weights of the nets that have the lowest L1 norm are set to zero. The *amounts* are potentially different between linear and convolutional layers and between decoder and encoder. Usually the amounts for the convolutional layers are lower.
- **Dropout**: adding a dropout layer before every hidden linear layer of the nets. Experimenting a bit I found that the nets perform better when dropout is applied only to the encoder.
- **Iterative autoencoding**: up to now we considered only the error between the original image and the first reconstruction. Here I tried to add to the loss also the error between the original image and second and third reconstructions, namely feeding the output of the decoder back to the encoder. However this slowed considerably the training process with yielding considerably better results.

At this point I proceeded to a hyperparameter search over the following degrees of freedom:

- Number of convolutional layers: 2, 3 or 4

- Number of channels: from 4 to $4 \cdot 2^j$ for the j^{th} convolutional layer
- Kernel size: from 2 to 6. Stride: from 1 to 4. Padding: from 0 to half of the kernel size
- Number of fully connected hidden layers: 1, 2 or 3
- Number of neurons: from 8 $^{512}/_{2^j}$ for the j^{th} layer
- Dropout probability: from 0 to 0.7
- N_{latent}: from 1 to 16
- Optimizer learning rate (from 10^{-5} to 10^{-1}) and weight decay (from 10^{-7} to 10^{-1})
- train batch size (from 64 to 512) and *patience* for early stopping (from 4 to 16)
- *prune_every*: never or from 1 to 10 and the four pruning *amounts* from 0 to 0.5

The search was performed with *optuna* at its default setting using a 5 fold cross validation for every hyperparameter combination and adding to average validation loss a penalty of 10^{-5} times the average training time in minutes plus $10^{-2} \cdot N_{latent}$ as regularizing terms.

After 60 trials, half of which were pruned due to an invalid net architecture, the best hyperparameters are 3 convoultional layers with channels [6,12,17], kernel sizes [2,2,3], strides [1,1,3], paddings [1,0,1]; a single hidden linear layer with 57 neurons and after a dropout layer with probability 0.1905 and $N_{latent} = 6$; learinig rate of $2.89 \cdot 10^{-3}$, weight decay of $6.51 \cdot 10-7$, patience of 13 and train batch size of 383. The net is pruned every 5 eopchs with amounts [0.398, 0.1, 0.214, 0.068], respectively for encoder linear layers, encoder convolutional layers, decoder linear layers and decoder convolutional layers.

The best net is then obtained by training with this hyperparameters in a 5 fold setup and choosing the one with the smallest validation loss.

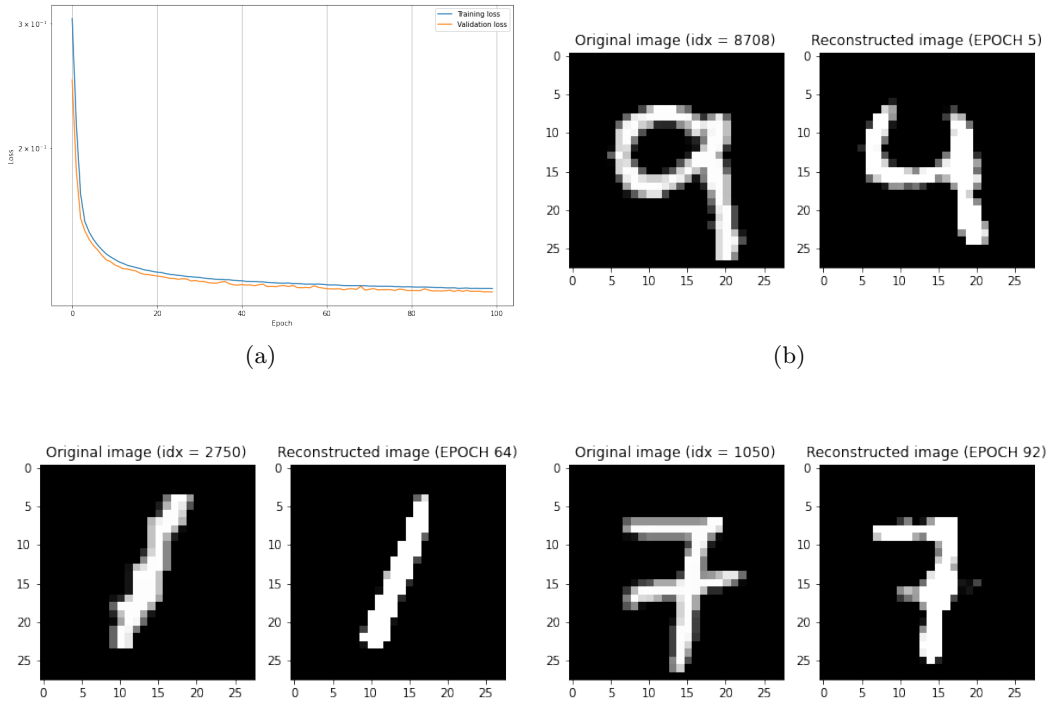
2.3 Denoising autoencoder

An interesting subset of autoencoders is the one of denoising autoencoders, where the encoder is provided a noisy image and te goal is reconstructing the original image. To do this one can simply act on the *transform* attribute of the dataset; in particular I tried the following types of noise:

- **UniformRandomNoise**(*amount*): every pixel value (that ranges from 0 (black) to 1 (white)) is summed to a random number extracted from a uniform distribution between -*amount*/2 and *amount*/2. The pixel values are then clipped to be between 0 and 1.
- **SaltNPepper**(*amount*): *amount* of the pixels of the image are randomly set to 0 or 1.
- **RandomHole**(*size*): a rectangle of size *size* is randomly placed above the image setting to 0 the pixels lying beneath.
- **RandomRotation**(*degrees*): the image is rotated by a random angle extracted by the uniform distribution between -*degrees* and *degrees*.

Now: in the MNIST dataset digits already appear with different orientations (expecially the digit 1), so it is virtually impossible for the net to discriminate between an 'intrinsic rotation' and a RandomRotation. Also, the RandomHole doesn't produce very interesting results, so I focused on the first two types of noise.

To deal with noise I started from the pretrained best net previously found and trained and tested it against different types of noise. The training is performed using the best combination of hyperparameters found with optuna and the results are shown in tab 1



(a) (b)

(c) (d)

Figure 2: Behavior of training and validation loss during learning of the best net and some examples of original versus reconstructed images (at epochs 5, 64 and 92). Thanks to the BCE loss the reconstructed images are far less blurred than the ones of the basic autoencoder.

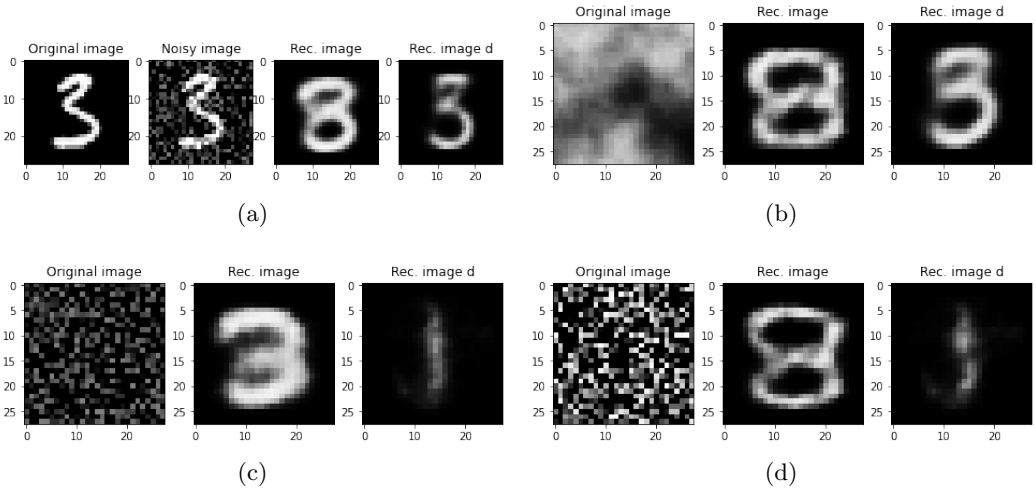


Figure 3: In all images 'Rec image' is the one reconstructed by the best autoencoder with no denoising additional training, while 'Rec image d' is the one reconstructed by the autoencoder trained against URN(2) + SNP(0.1). (a): The denoising autoencoder manages to correctly reconstruct the noisy image, while the simple one fails. Also the simple one is more prone to 'hallucinations', i.e. seeing digits in pure noise (c, d.), however when fed Perlin noise (which wasn't trained aginst), it hallucinates as well.

Trained against	Test loss against		
	No noise	SNP(0.1)	URN(0.7)
No additional training	0.1246	0.2459	0.3150
SNP(0.1) (52 epochs)	0.1239	0.1249	0.2920
URN(0.7) (55 epochs)	0.1233	0.2539	0.1239
URN(0.7) + SNP(0.1) (60 epochs)	0.1236	0.1248	0.1242
URN(2) + SNP(0.1) (100 epochs)	0.1407	0.1404	0.1403

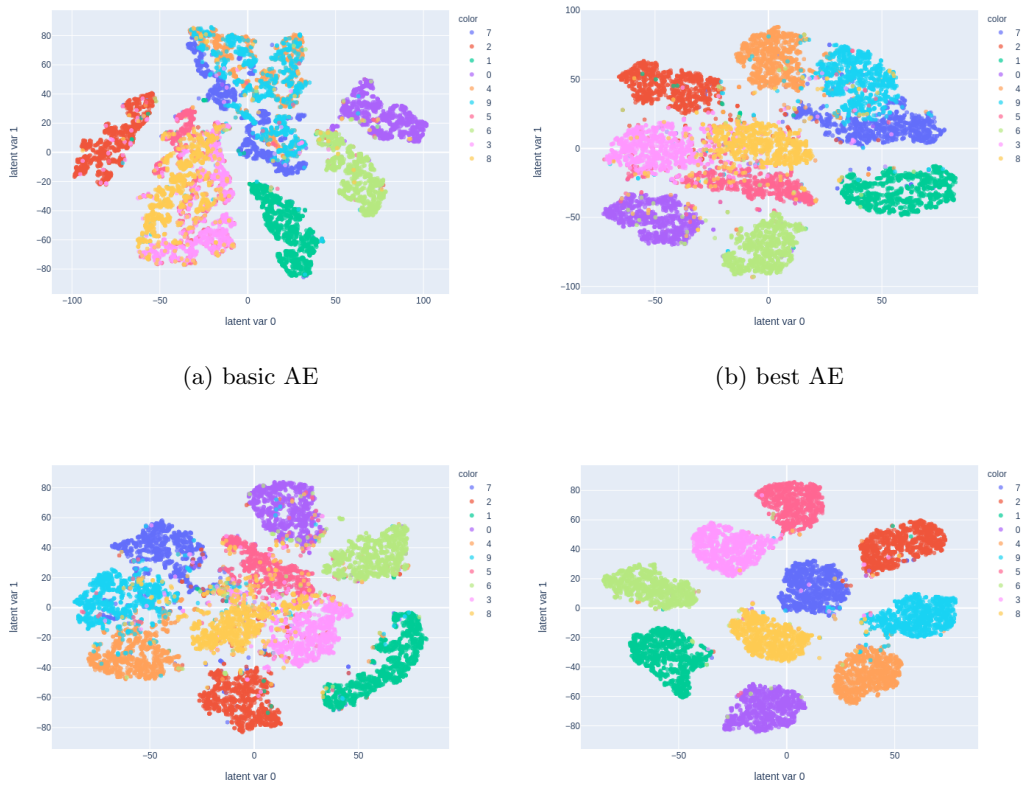
Table 1: Test losses against different types of noise (URN stands for UniformRandomNoise and SNP for SaltNPepper) of the best architecture after different types of training.

2.4 Supervised fine-tuning

If we take just the pretrained encoder (without denoising) and stack a linear layer with 10 neurons on top of it, we can fine tune some of the weight with supervised learning, obtaining a classifier. To do so I used again the best hperparameters (without pruning), with the exception of the loss function, which now is the CrossEntropyLoss. I tried letting the gradient propagate up different depths in the pretrained encoder, testing every choice in a 5 fold setup. The results are in tab 2, where there is an evident jump in performance once we allow some hidden layers of the encoder to be updated.

Deepest non frozen layer	Average train loss	Average validation loss	Average test accuracy
Classifying layer (10 neurons)	0.3417	0.3237	0.9126
Encoder out layer (6 neurons)	0.2108	0.1912	0.9452
Encoder hidden linear layer	0.0309	0.0466	0.9862
Encoder last conv. layer	0.0206	0.0427	0.9874

Table 2: The test accuracy is computed as the fraction of corectly classified samples.



(a) basic AE

(b) best AE

(c) denoising AE

(d) fine-tuned AE

Figure 4: 2 components TSNE representation of the latent space of different nets.

2.5 Visualization techniques

There are several ways to visualize the latent space of the encoder: I tried taking a 2D slice of said space, applying PCA and TSNE. Of all these techniques the latter proved to be the most interesting. In fig 4 one can appreciate how much the supervised fine tuning helped correctly clusterizing the digits together. Instead to visualize the encoder I made a simple interface to interactively generate images by setting the values of the latent space variables: I recordered the screen of me using it, the videos are `code/interactive_generation_basic.mp4` and `code/interactive_generation_best.mp4`. For the latter, being the latent space six dimensional it is more tricky to obtain meaningful images.

3 GAN