Unsupervised Deep Learning

Assignment 2

Andrea Nicolai - 1233407

03/09/2021

In this homework we are requested to solve several unsupervised learning tasks on the MNIST handwritten digits dataset. For this purpose, we first build, find the optimal set of hyperparameters thanks to *optuna* and finally test a Convolutional AutoEncoder. Thus, we explore the latent space where our network has encoded data. Moreover, we perform some transfer learning in order to solve a supervised problem exploiting the network we have previously trained. Then, using noisy input data as training set, we implement a Denoising AutoEncoder. Finally, we create both a Variational Convolutional AutoEncoder and a GAN.

Introduction

The MNIST handwritten digits dataset consists of 28×28 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 thus 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 the best, in the sense with optimized hyperparameters, AutoEncoder finally exploring its latent space and eventually testing its denoising capabilities. Later on, we exploit the features the network was able to assimilate, thus using transfer learning and train some fully connected layer to perform supervised tasks. Finally, adding "continuity" to the latent space, we implement a Variational AutoEncoder. 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.

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 as better as possible the original data. Usually, encoder and decoder architectures are symmetric wrt each other, but this is not a hard constraint. Namely, we have that for the **Encoder**:

- **First Convolutional Layer**: having one channel in input, n channels as output, *kernel* 4×4, *stride* 2 and *padding* 1. Activation function is *ReLU*.

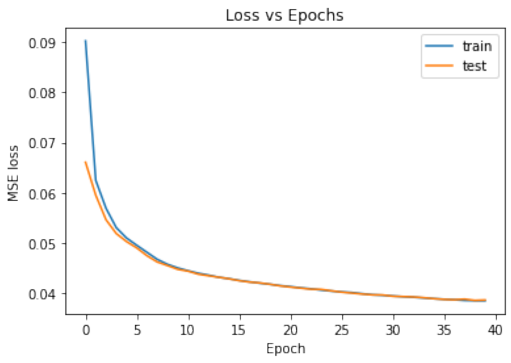

Figure (1) – MSEloss curve in function of the number of epochs for the Convolutional AutoEncoder.

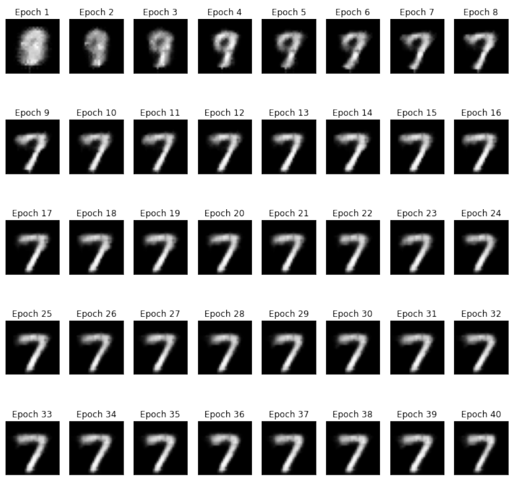

Figure (2) – Images reconstructed with respect the number of iterations. Initially blurred and confused, after some time one can clearly distinguish a 7.

- **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*.
- **Third Convolutional Layer**: having $2 \cdot n$ channels as input, $4 \cdot n$ channels as output, *kernel* 3×3, *stride* 2 and *padding* 1. Activation function is *ReLU*.
- **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.
- **Second Linear Layer**: with 64 units, finally "encoding" the input to a n_{dim} latent space.

And 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:

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

For this task we use a $MSELoss$ function, and after training we obtain the following trend of reconstruction loss (see Fig. 1). Moreover, it is shown how reconstructed images change according to the number of iterations (see Fig. 2), we can clearly tell after some steps that the input label is a 7. Note as the input images were previously converted to tensor and normalised via the PyTorch function *Normalize((0.1307,), (0.3081,))*, but differently from previous homework without any *augmentation*.

Hyperparameters 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 40 epochs, taking care of pruning not promising trials. It has been chosen this library to exploit its *Bayesian sampling* procedure, which allows better performance and efficiency. The metrics used for the task is the *validation loss*, and the aim is its minimization. The overall number of trials is 100, which has taken a really long time. Some interesting plots about this process can be seen in Figg. 8, 9. The optimal set of hyperparameters found is:

- $n_{channels} \rightarrow 12$

- $n_{dimension} \rightarrow$ 7
- **Optimizer**→ 'Adam'
- **Learning rate**→ 0.002218
- **Weight Decay**→ $1.01e^{-5}$

The performances achieved in reconstructing images exploiting such architecture can be seen in Fig. 10, and the network seems to be working properly.

Latent space exploration

Let us now try to 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**: the Principal Components Analysis is an unsupervised algorithm which tries to compute what are the variables having the largest variance, thus projecting the ones with less variance onto the subspaces spanned by the most informative variables. Hence, data variability is preserved as much as possible through this *linear* transformation. Its effect can be seen in Fig. 11. Points however are mixed up, despite some clusters can be identified thanks to the color which denotes their true label.
- **t-SNE**: the 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 Fig. 3, where digits form clusters that are separate from each other but some exceptions.

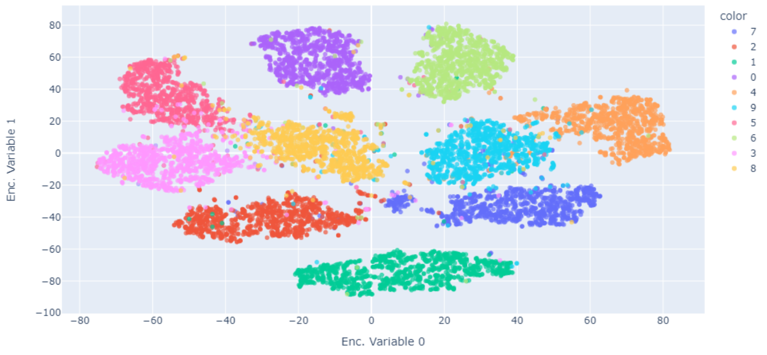

Figure (3) – The original 7-dim latent space was projected into only two dimensions using TSNE algorithm, to clusterize points and better visualize it.

Finally, random samples generated from the latent space can be shown in 12. Some digits can be clearly recognized, while others are simply sketches.

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, one is able to take advantage of the encoded representations which are performed by the

encoder, eventually sparing computational effort since there is no need to train a full Convolutional Neural Network.
In this assignment, we have chosen to stack a *single* fully connected layer with a number of units n_{hidden} to be optimized, 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. 10). Moreover, we use *ReLU* activation function, along with a dropout p layer between the hidden and the output.
Some hyperparameter optimization (see Figg. 13 and 14) is done via optuna, for a total of 50 trials. The metrics used is the validation loss, with the latter being computed using the *CrossEntropy* function. The search of the optimal set of hyperparameters returned:

- $n_{hidden} \rightarrow 928$
- **p**$\rightarrow$ 0.104258
- **Optimizer**$\rightarrow$ 'Adam'
- **Learning rate**$\rightarrow$ 0.001839
- **Weight Decay**$\rightarrow$ $5.81e^{-5}$

With the network performances being the one in Fig. 4. Actual average overall accuracy is 96.3% not as good as in the previous homework, which turned out to be 98.6%, though being still acceptable. However, the convergence was reached much faster, even though performing hyperparameters optimization.

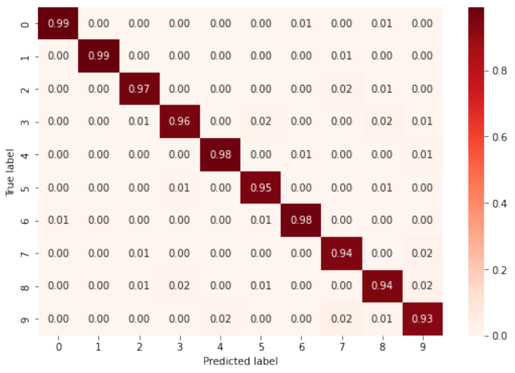

Figure (4) – Fine tuning network performances. Overall accuracy is 96.3%.

Denoising AutoEncoder

Another variant of AutoEncoders one may encounter are Denoising AutoEncoders. These particular architectures are devoted to reconstruct images which are corrupted by noise.
For this task we feed the network, which has the *same* architecture of the best Convolutional AutoEncoder we previously defined, with some corrupted images according to the following noise:

- **Gaussian Noise**: the usual Gaussian noise sampled from a Normal distribution $\mathcal{N}(\mu = 0., \sigma = 1.)$
- **Salt & Pepper**: pixels are chosen with a certain probability $p = 0.2$ and switched either to "1" w.p. $p' = 0.5$ or to "0" w.p. $1 - p' = 0.5$.

The input data is indeed corrupted either using *both noises*, or *either*, or none (see Fig. 15). One could have, maybe more properly and sticking with the theoretical definition, have trained the network *only* with noisy data. However, we have decided to inspect how the network performs also when is fed with “clean” inputs though it might have been considered like “cheating”. Performances of the network in the denoising task are the ones in Fig. 5. One can see that results are acceptable and even in presence of really high noise which does not allow to tell the digits, at least for a human eye, it is able to reconstruct the image despite being blurred. Some uncertainty, however, can be seen in the 0 at the right-bottom line which resembles a 6. Finally it is worth to notice that in the right-center line the network is able to fix even digits (0, in this case) which are poorly handwritten, but without noise.

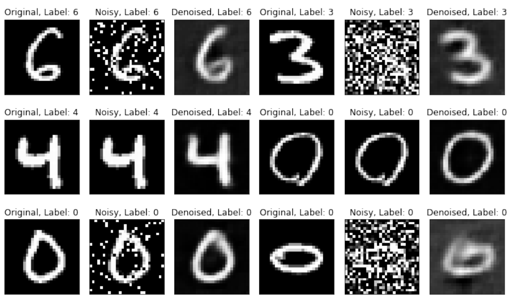

Figure (5) – Denoising AutoEncoder output for different inputs, that can eventually exhibit some noise. It is interesting to see that, even in absence of noise, the network "fixes" the effects of bad handwriting, such as in the "0" case.

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

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

where $\vec{x}$ is the input and $\vec{\hat{x}}$ 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:

$$\vec{z} = \vec{\sigma_x}\vec{\xi} + \vec{\mu_x} \qquad \text{with } \vec{\xi} \text{ sampled from } \mathcal{N}(\vec{0}, \vec{1})$$

The sampled outputs visible in Fig. 6, where we smoothly vary the variables belonging to latent space, are surprisingly good.

GAN

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 28×28 image, which is flattened to a 784-unit Layer. The architecture for the **Generator** is:

- **Input Layer**: 64 units, *LeakyReLU* activation function with 0.2 negative slope
- **First Linear Layer**: 256 units, *LeakyReLU* activation function (0.2 neg.sl.)
- **Second Linear Layer**: 512 units, *LeakyReLU* activation function (0.2 neg.sl.)

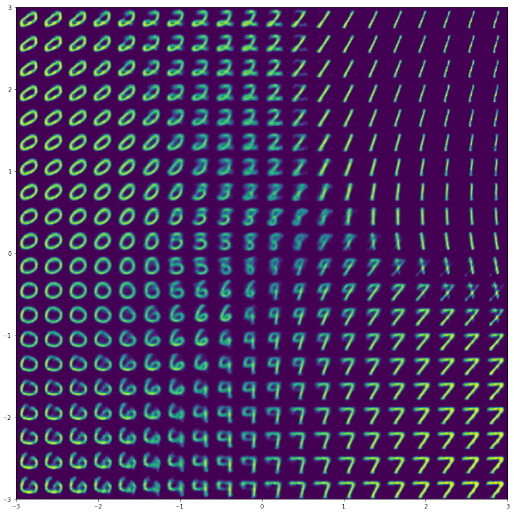

Figure (6) – The latent space spanned by the Variational AutoEncoder. Samples smoothly interchange when either one of the latent variables is perturbed.

- **Third Linear Layer**: 1024 units, *LeakyReLU* activation function (0.2 neg.sl.)
- **Output Layer**: 784 units, *Tanh* activation function

Whereas for the **Discriminator**:

- **Input Layer**: 784 units, *LeakyReLU* activation function with 0.2 negative slope and 0.3 dropout
- **First Linear Layer**: 1024 units, *LeakyReLU* activation function (0.2 neg.sl.) and 0.3 dropout
- **Second Linear Layer**: 512 units, *LeakyReLU* activation function (0.2 neg.sl.) and 0.3 dropout
- **Third Linear Layer**: 256 units, *LeakyReLU* activation function (0.2 neg.sl.) and 0.3 dropout
- **Output Layer**: 1 unit, with *Sigmoid* activation function

Note as the Discriminator's Output Layer has a Sigmoid activation function and is composed by a single unit. The latter one is a constraint by the nature of the problem.
Specifically for this task, samples are normalized via *Normalize((0.500,),(0.500,))* otherwise the two losses do not converge, as we experencied in many trials. The expected, and obtained, behavior of the loss is indeed the one depicted in Fig. 7, where initially the Generator is not able to build good samples and the Discriminator has easy life. As training evolves, the Generator is able to generate more and more high-quality images which fool the Discriminator that in the meanwhile has in turn been trained, finally reaching a an equilibrium value for such process.

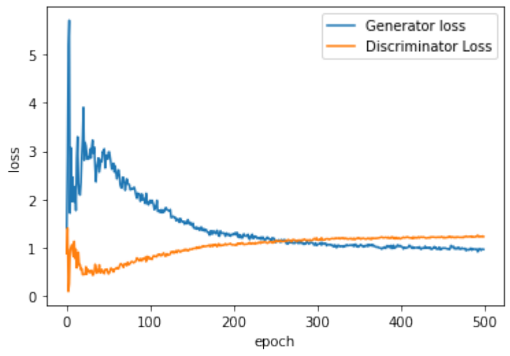

Figure (7) – GAN losses trend for Generator and Discriminator

The evolution of samples generated by the network can be inspected in the *gif* files attached to this report: 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.

Appendix

This section contains some additional figures, not fundamental for the drafting of the report, nevertheless worth to be showed.

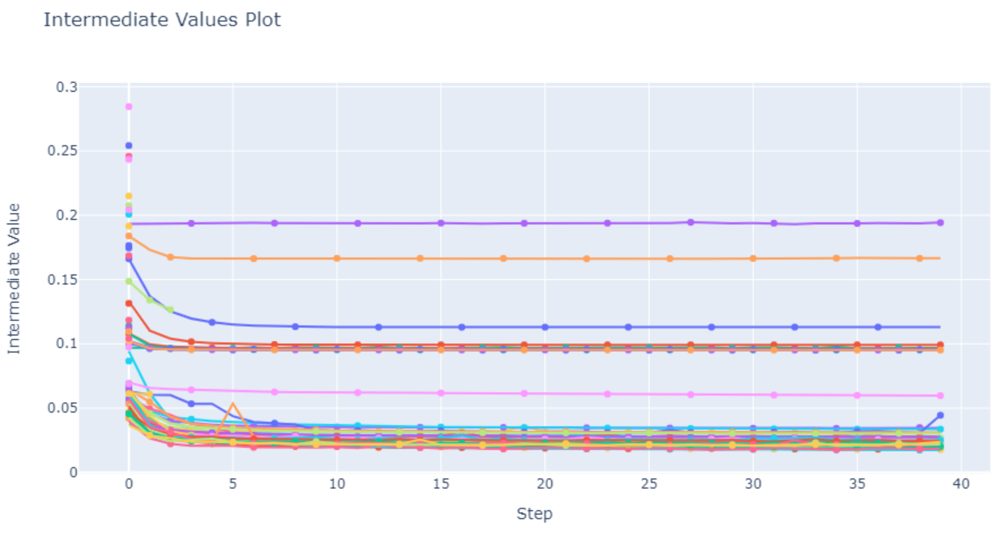

Figure (8) – Loss trend for the 100 different Optuna trials when trying to find the set of hyperparameters for the best autoencoder. One can see that, when loss is too high due to poor initialization, the trial is pruned.

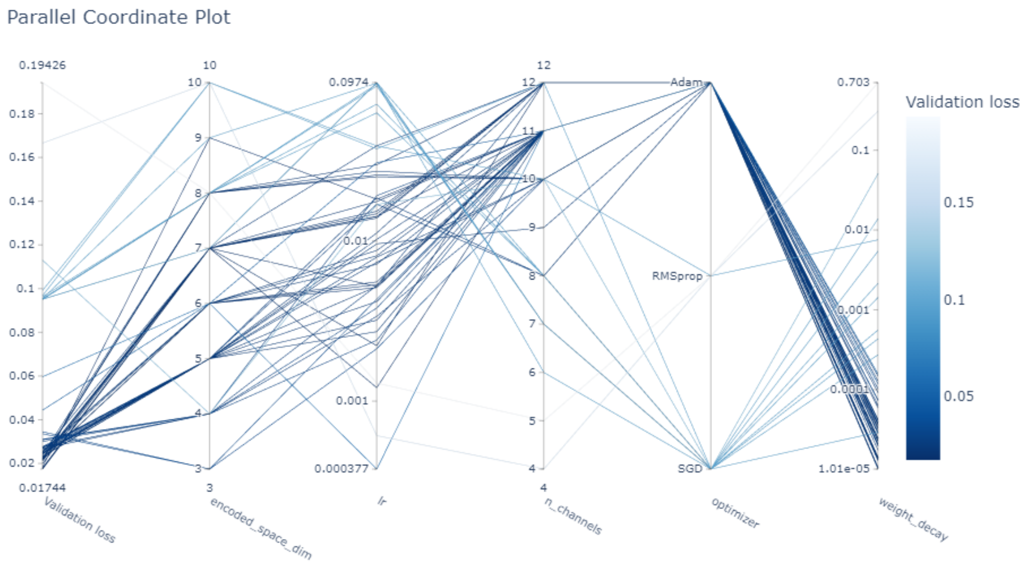

Figure (9) – Parallel plot of hyperparameters for the 100 different Optuna trials when trying to find the set of hyperparameters for the best autoencoder. Thanks to this one can infer what and in which measure a hyperparameter affect the metrics (validation loss).

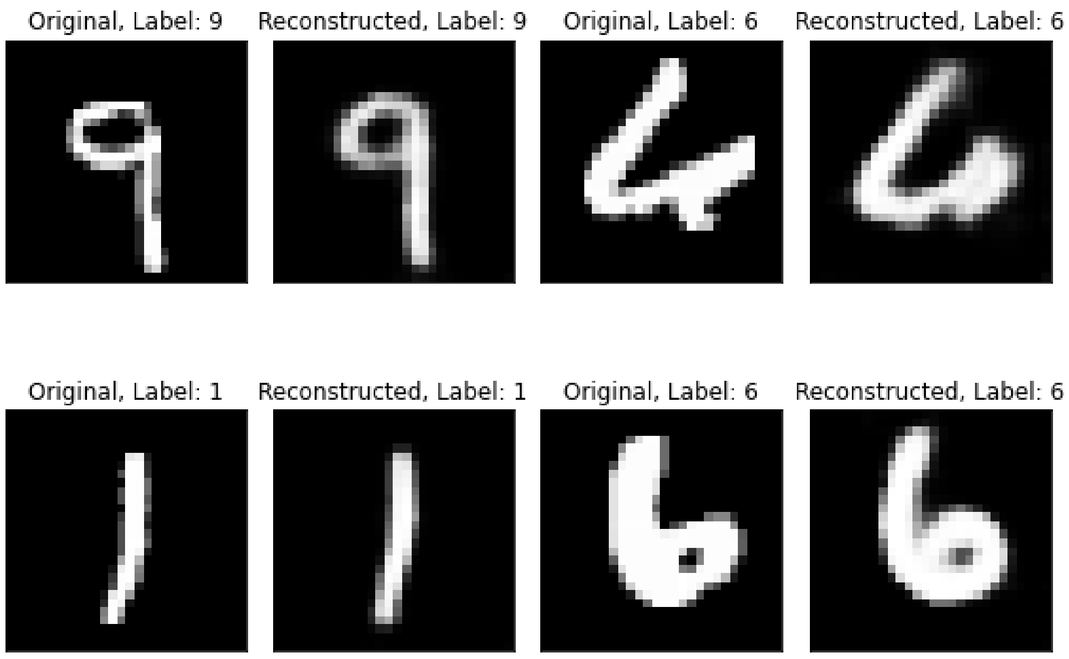

Figure (10) – How autoencoder with best hyperparameters actually performs.

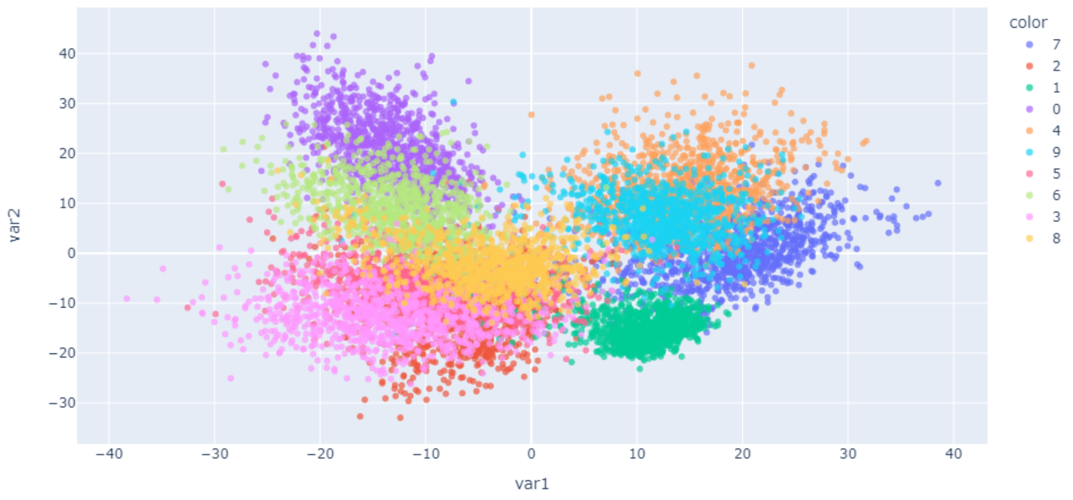

Figure (11) – Latent space when applied PCA to reduce the dimensionality to only 2 dimensions. Labels are still mixed, and no clear pattern can be distinguished.

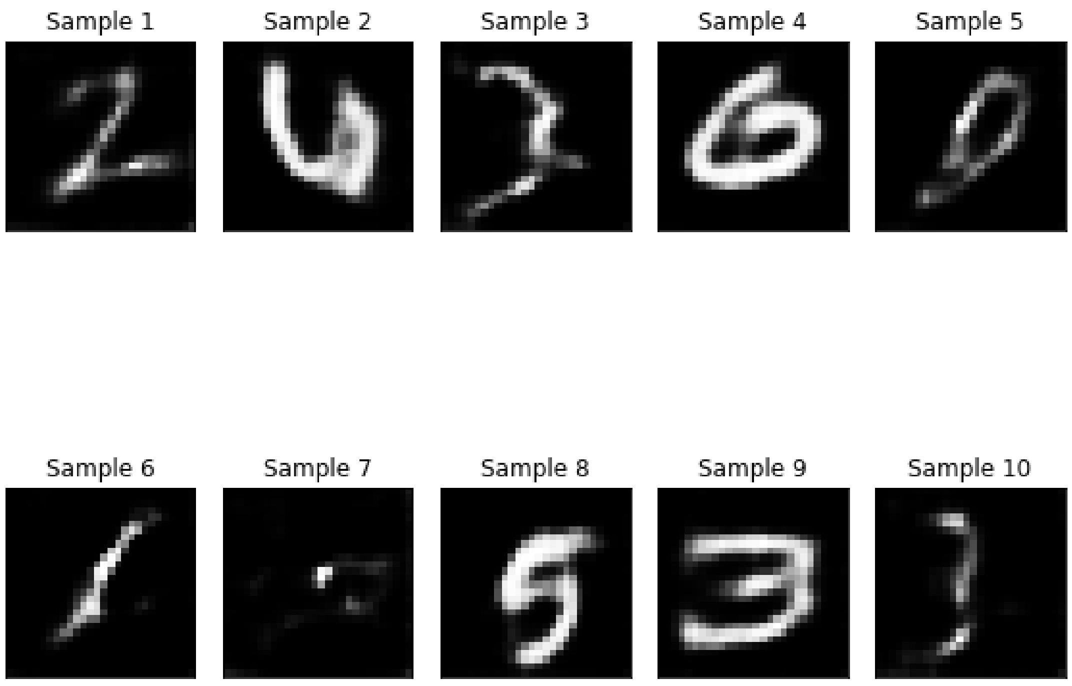

Figure (12) – Samples generated from the latent space of the best AutoEncoder architecture. Some shapes of digits can be clearly seen, but others are simply aesthetic, though meaningless, sketches.

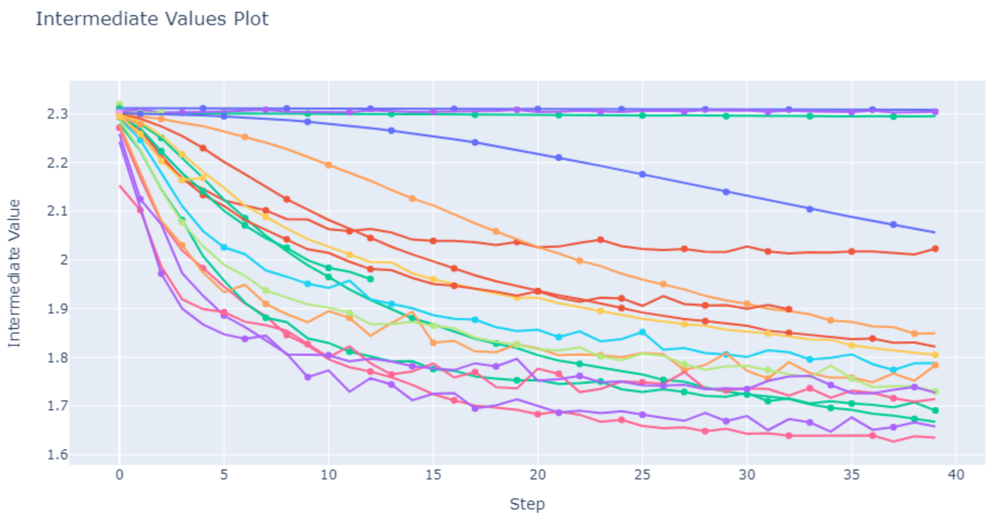

Figure (13) – Loss trend for the 50 different Optuna trials when trying to find the fine-tuning network for supervised task. One can see that, when loss is too high due to poor initialization, the trial is pruned.

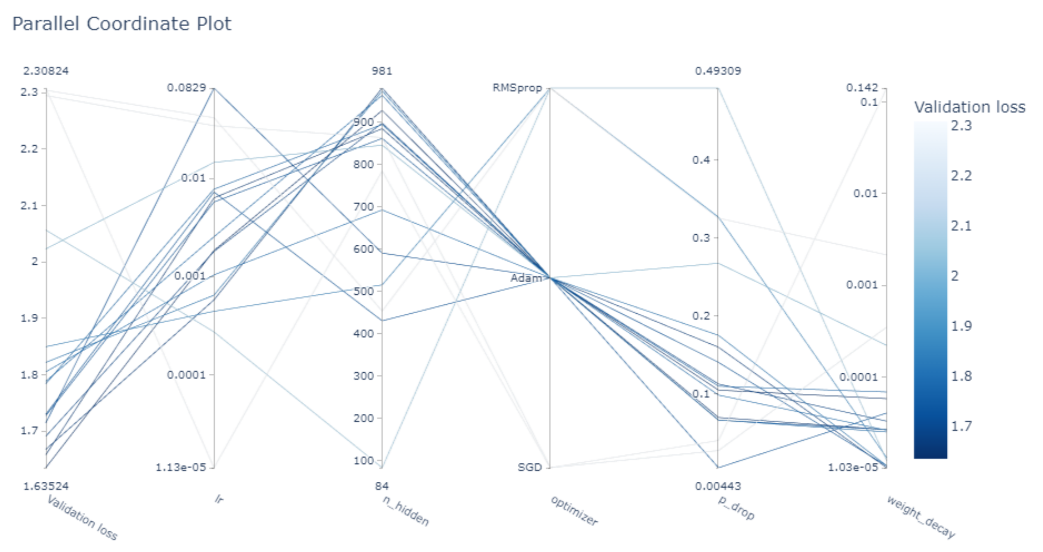

Figure (14) – Parallel plot of hyperparameters for the 50 different Optuna trials when trying to find the set of hyperparameters for the best fine-tuning network for supervised task. Thanks to this one can infer what and in which measure a hyperparameter affect the metrics (validation loss).

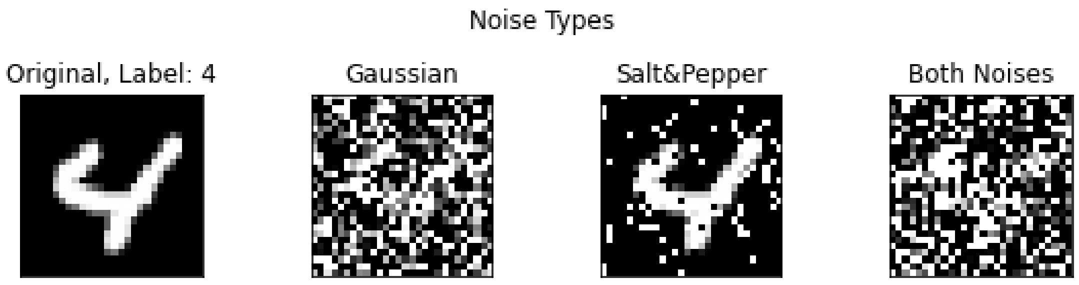

Figure (15) – Different type of noises applied to input data.

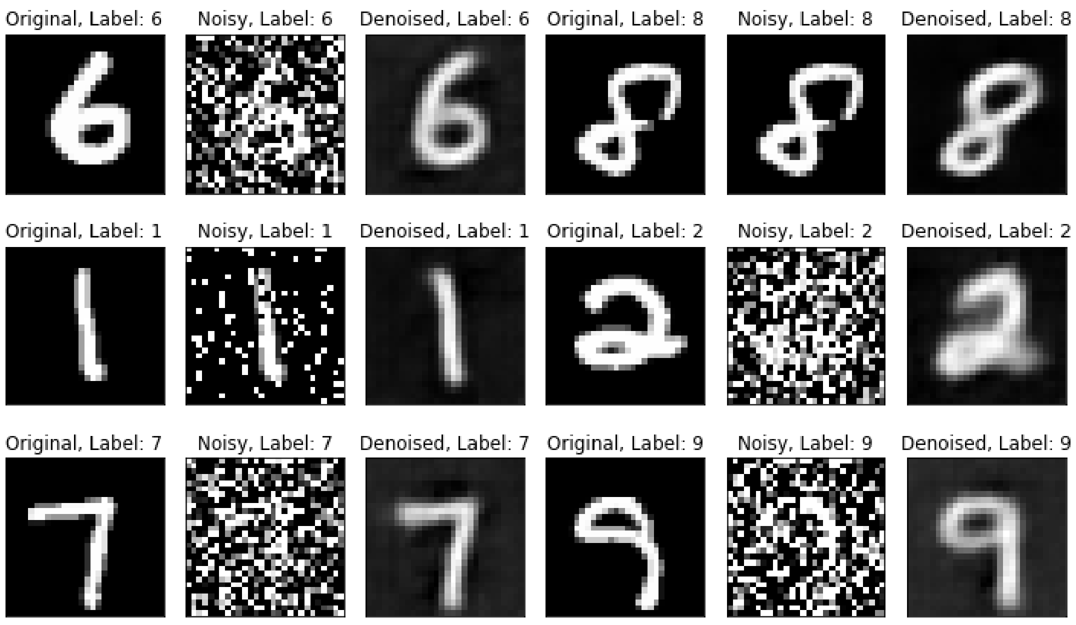

Figure (16) – Other denoised samples. Note as the 8 is "fixed" back to its ideal shape.