Exercise 02 - Unsupervised Deep Learning

University of Padua

Name:	Daniele Mellino
Student Number:	2013373
Course of study :	Physics of Data
Date:	February 13, 2022

1 Introduction

This project concerns the implementation and testing of neural network models in the framework of unsupervised deep learning. In particular a Convolutional-Auto-Encoder(CAE) has been trained and tested on the FashionMNIST database. The CAE latent space has been explored and used to generate new samples. The same model is also being tested as a reconstructor of noisy images. Afterwards the CAE encoder has been fine tuned in order to compare its performance in the classification task. In the end a Generative Adversarial Network(GAN) has been implemented with the goal of generate new samples.

This report is divided by model. Methods and a results is discussed for each task.

2 Convolutional Auto Encoder

2.1 Model and Training

A convolutional autoencoder(i.e. an autoencoder that uses convolutional layers) is a neural networks composed of two main structures.

An **Encoder**, which learns how to reduce the input dimensions and compress the input data into an encoded representation. The Encoder is composed as follow :

- First Convolutional Layer: one input channel(grayscale images 28x28), n (n_feature in the class) channel output , 3x3 kernel, stride 2 and padding 1 ; (14,14)
- Second Convolutional Layer , n input channel, 2·n output, 3x3 kernel, stride 2 and padding 1 ;(7,7)
- Third Convolutional Layer , 2·n input channel, 4·n output, 3x3 kernel, stride 2 and padding 0;(3x3)
- Flatten into a Linear Layer with $3 \cdot 3 \cdot 4$·n input units and 64 output;
- Second Linear Layer with 64 input and "neur_encoding" output .

For each layer ReLU activation function has been used.

A **Decoder**, in which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible. In this project it is the symmetric version of the Encoder, therefore:

- Two Linear layers: input "neur_encoding" nodes $\rightarrow 64 \rightarrow$ 36·n;
- Unflatten the output of the previous layer in $4 \cdot \mathrm{n} \cdot 3 \cdot 3$ channels;
- Three Transposed Convolutional Layers with stride 2 to reconstruct the original size. In particular the channels are respectively $(4 \cdot \mathrm{n}) \rightarrow (2 \cdot \mathrm{n}) \rightarrow (\mathrm{n}) \rightarrow 1$.

The activation function is the ReLU, except for the last deconvolutional layer which have Sigmoid activation function to force output to be between 0 and 1 (valid pixel value).

As in the previous experience the data has been normalized and randomly divided into training and validation sets[48000-12000]. In order to train the model, we use the standard Mean Squared Error loss between the input images and the output of the decoder. In order to tune the network a Random Search using optuna is implemented. In this search the following parameters and optimiser has been considered :

- n: the number of convolutional channels, it is chosen in a range [4,10];
- neur_encoding : the number of final nodes of the Encoder, it is sampled in a range [10,60];

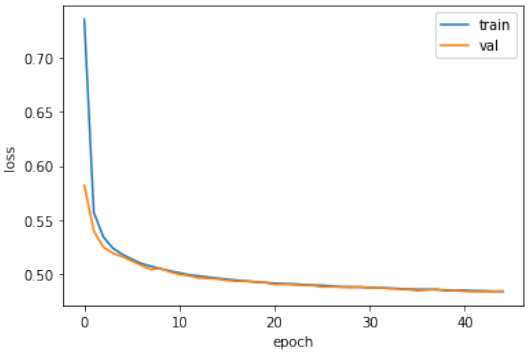
optimiser	lr	rg	neur_encoding	n
Adam	0.004295	3.96 e-06	27	9

Table 1: Optimised parameters for CAE, via Random Search (50 trials)

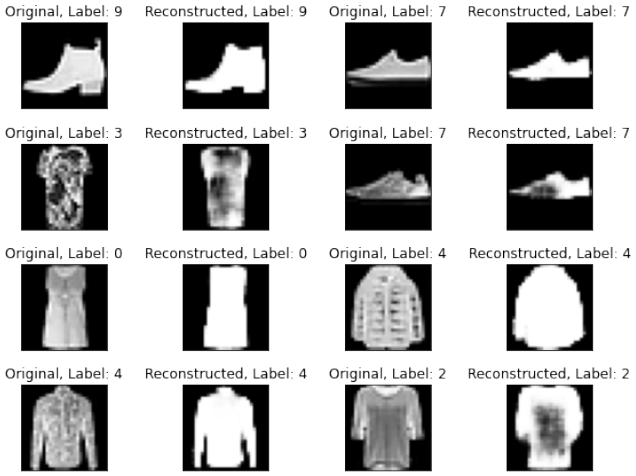
- Adam or SGD with momentum;
- rg: the value of the L2 regularization parameter, sampled from a log uniform distribution between 1e-6 and 1e-4;
- lr : The optimiser's learning rate, sampled from a log uniform distribution between 5e-5 and 5e-2.

As in the previous exercise we use pruning to speed up the computation(MedianPruner with 3 startup trial and 2 warmup steps).

The best model obtained from this procedure (Table 1)is then trained again since the validation loss does not decrease for 4 consecutive epochs(early stopping). The trend of validation and training loss is shown in Figure 1(a). In Figure 1(b) some examples of test image reconstruction is shown. The network seem to be working properly even though the reconstruction is not perfect.



(a) MSEloss curve as function of the number of epochs



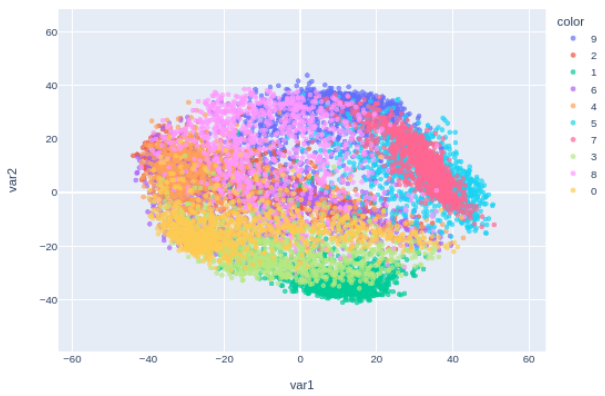
(b) Examples of image reconstruction.

2.2 Latent space and new Sample generation

The relevant information of the input data are reduced in the latent space. The latter has 27 dimension, in order to visualise it two technique of dimensionality reduction is implemented:

- **PCA**(Principal Component Analysis) aims to find the directions of maximum variance in the hight-dimensional data and projects it onto a new subspace with fewer dimensions than the original one. In it, data variability is preserved as much as possible. The results of it can be seen in Figure 1(c). Points are mixed up, however we can distinguish some cluster.
- **t-SNE**(t-distributed stochastic neighbor embedding) differently from PCA is a *non linear* technique which tries to preserve the local distance between points i.e. keeps points that were distant in the high-dimensional space distant in the reduced space. In Figure 1(d) the results of this procedure can be shown. Also here points are mixed up, even though the clusters are more distinguishable.

After the analysis of the latent space, the Decoder has been used to generate new samples. To do that a random point in the 2-dimensional space is selected, then that point is transformed into a 27-dimensional vector through the PCA inverse transformation and finally sent through the decoder. The results is shown in Figure 2.



(c) PCA of the latent space

(d) t-SNE of the latent space

Figure 1: In both graph the latent space has been obtained from the test set. As one can see the encoder does not learn a clear distinction between similar classes like "Pullover", "Coat" and "Shirt"[red-orange-purple] or "Sandal", "Sneaker" and "Ankle Boot"[light blue-fuchsia-blue]. Even though, this is reasonable because the shape of those objects are closely resembling.

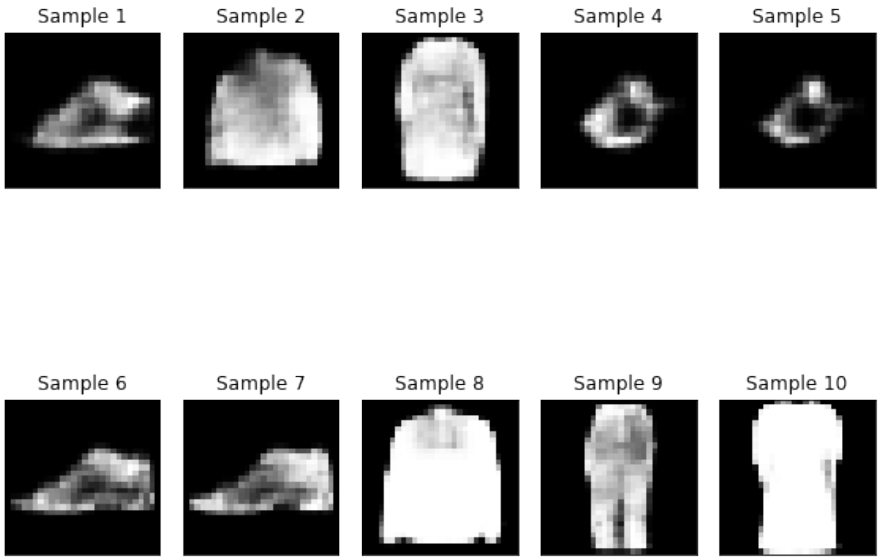


Figure 2: New sample generated from the CAE Decoder

2.3 Fine tuning and Classification

Since the encoder has encrypted the relevant features of the images in a low dimensional space it is convenient to use them to classify the objects.

To do that two fully connected layer used for classification are connected to the previously trained Encoder. Those two layer are composed as follow :

- The first has neur_encoding inputs(the output od the Encoder) and N_h output, here dropout is inserted with probability dp;
- The second has N_h inputs and 10 outputs corrisponding to the different classes.

The ReLU activation function is implemented and the loss function is the usual Cross Entropy Loss. The weights of the previously trained Encoder has been frozen, afterwards a brief fine tuning of the parameters is executed. This time in order to explore a different sampler, TPE algorithm is adopted. The tuned variables were N_h, dp, the learning rate (lr) and the L2 weight regularization (wg). The best hyper-parameters found is shown in Table 2. Here Adam optimiser is used.

lr	wg	N_h	dp
0.0031332	1.014e-05	276	0.06

Table 2: Optimised parameters for classification, via TPE Search (25 trials)

Consequentially the model is trained with the best parameters and then tested on the test dataset obtaining $86,41\%$ accuracy. As one could have expected from the latent space analysis the model performs well on the categories that are visually different like "Bag" and "Trousers" [class 1 and 8], while is not able to fully learn how to distinguish similar classes. As one can see from the Covariance Matrix in Figure 3 "Shirt"(6) is often confused with "T-shirt/Top", "Pullover" or "Coat"(1,2,4). Nevertheless the results that it achieves is good since performances are similar to the CNN architecture of the previous experience(see Homework-01) even though it has less parameters to train (once you have the Encoder), therefore it is faster.

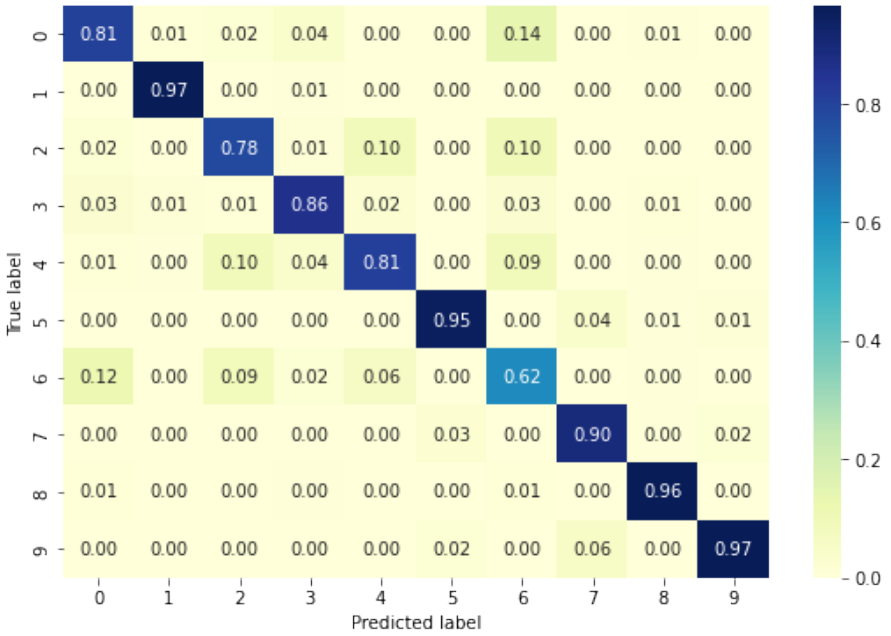


Figure 3: Covariance matrix of the Fine Tuned model

2.4 CAE as a denoiser

In this section CAE is tested as a Denoiser, as mentioned during the laboratory. First the previous model is tested with noisy images(Salt and Pepper, Gaussian Noise and both). The results is in Figure 5. As one can see it has reasonable performance only if the noise does not change the edges of the object.

Afterwards the training data is transformed with the union of Salt and Pepper and Gaussian noise transformations. This new data is used to re-train the CAE model architecture. The new model results in Figure 6, are quite good if the noise is the one used during the training(column 4 and 5). Otherwise the reconstruction is not satisfying.

3 GAN

3.1 Methods

GANs are a framework for teaching a DL model to capture the training data's distribution so we can generate new data from that same distribution. They are made of two distinct models, a generator and a discriminator. The job of the generator is to spawn 'fake' images that look like the training images. The job of the discriminator is to look at an image and output whether or not it is a real training image or a fake image from the generator. During training, the generator is constantly trying to outsmart the discriminator by generating better and better fakes, while the discriminator is working to become a better detective and correctly classify the real and fake images.

Training GAN is in general very time consuming, therefore in this section hyperparameter optimization is done "by hand", starting with the suggestion contained in the original paper from Goodfellow et al. (2014).

In the paper the main suggestion are :

- Use strided convolutions for downsampling(discriminator) and fractional-strided convolution for upsampling(generator);
- Use batch normalization in both generator and architecture, it standardizes the input layer to have 0 mean and unit variance;
- Use ReLU activation in generator for all layers, except for the output which uses Tanh;
- Use LeackyReLU activation in the discriminator for all layers with 0.2 as negative slope.
- Use normal initialization for all layers.
- Use Adam optimiser with beta parameters equal to (0.5,0.999)

Before starting the training dataset is normalised to be in the range [-1,1]. This is needed because as suggested in the paper, we use Tanh activation in the last layer of the Generator. The images is also rescaled to size = 32 for model design.

The model Generator and Discriminator architecture are resumed in Figure 7 and Figure 8. Notice that in the Generator we start with a latent vector, we upsample it through the transposed convolution to a set of features each of size(4x4). After that setting stride equal to 2 we double the dimension of the features in each layer arriving to 32.

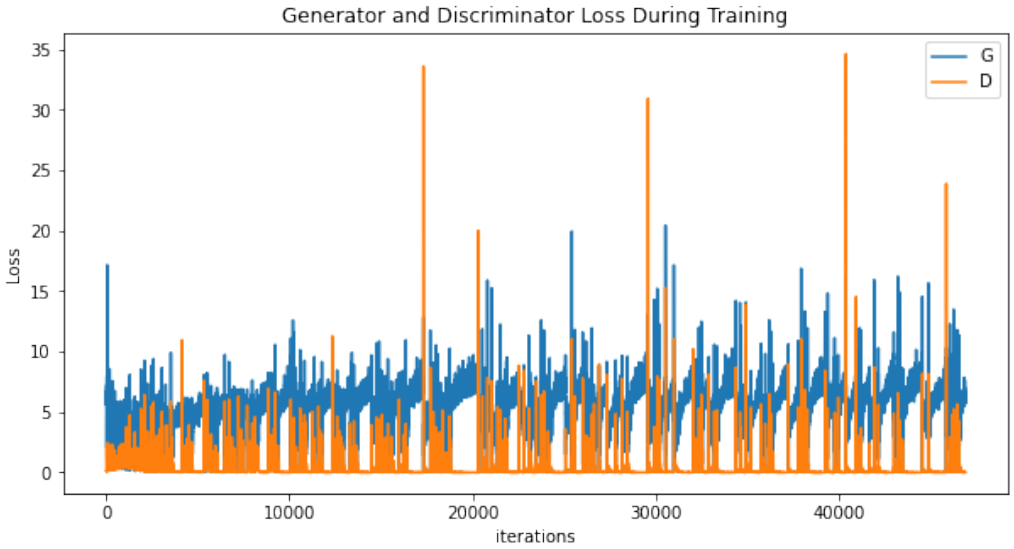
The competition between Discriminator and Generator is expressed in the loss function. If one call, x the image, $D(x)$ the output of the discriminator which is a number between 0 and 1 which resemble the probability that x is real, in addition $G(x)$ the output of the generator. The GAN loss function want to minimize :

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_{\text{generated}}(z)}[1 - \log D(G(z))]$$

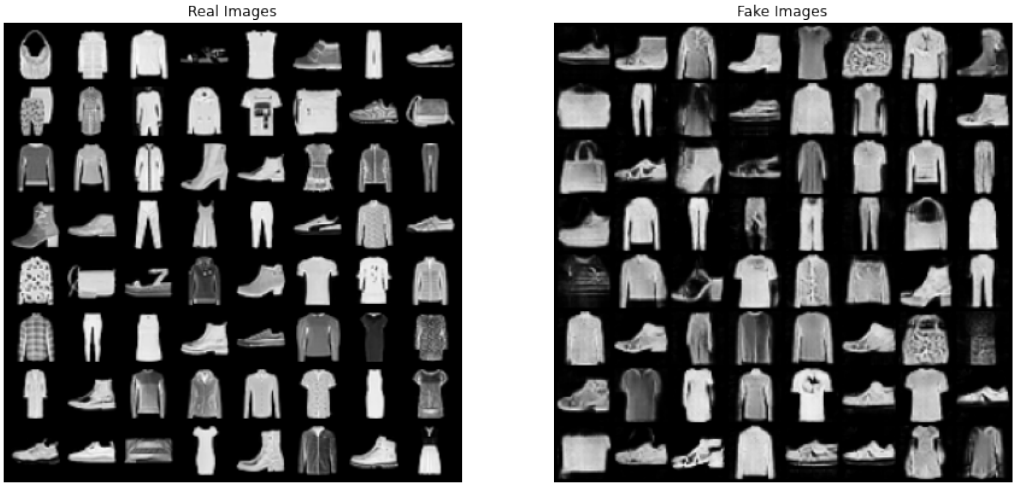
In this architecture is implemented a modified minimax loss, which uses the binary cross-entropy(BCE) loss function. Since there are two groups of images being fed into the discriminator (real images and fake images), the loss on each of them(which is computed by the Binary Cross Entropy loss) is combined to obtain the discriminator loss. For the generator loss, rather than training G to minimize $1 - \log D(G(z))$ (probability that D classify fake images as fake), we train G to maximize $\log D(G(z))$(probability that D incorrectly classifies the fake images as real). The latter "trick" was introduced by Goodfellow to provide sufficient gradients, especially early in the learning process.

3.2 Results

The Generator and Discriminator loss is displayed in Figure 4(a), as one can see the model oscillates and does not seem to converge yet, however looking at the generated samples in Figure 4(b) the results are good and the model implementation is considered accomplished.



(a) Generator and Discriminator loss per iteration(batch processed)



(b) Real images on the left, GAN generated images on the right.

Figure 4:

References

Goodfellow et al. 2014, in Advances in Neural Information Processing Systems, Vol. 27 (Curran Associates, Inc.). https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

4 Appendix

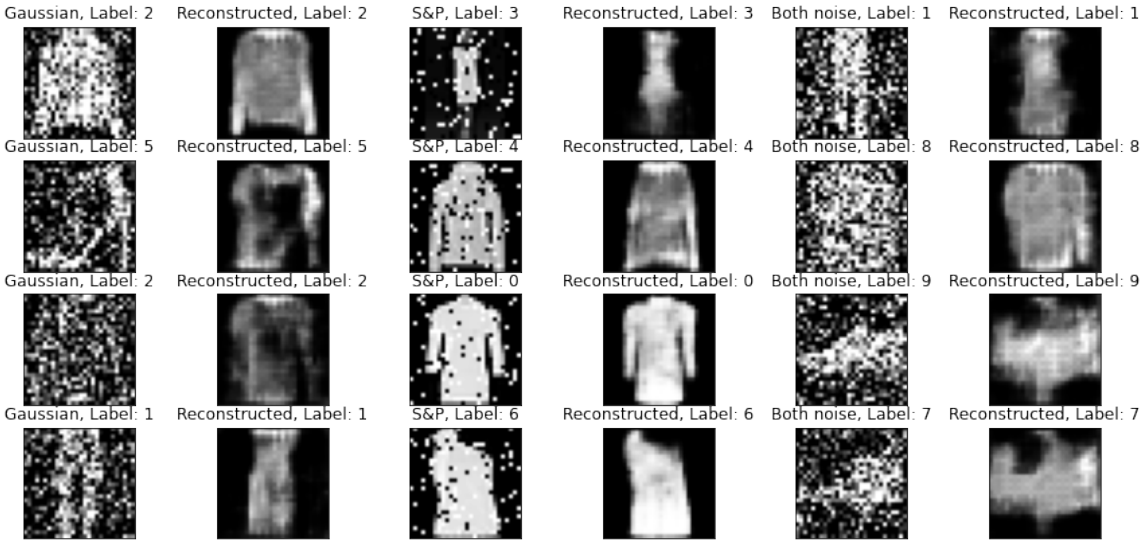


Figure 5: CAE performance with noising test images

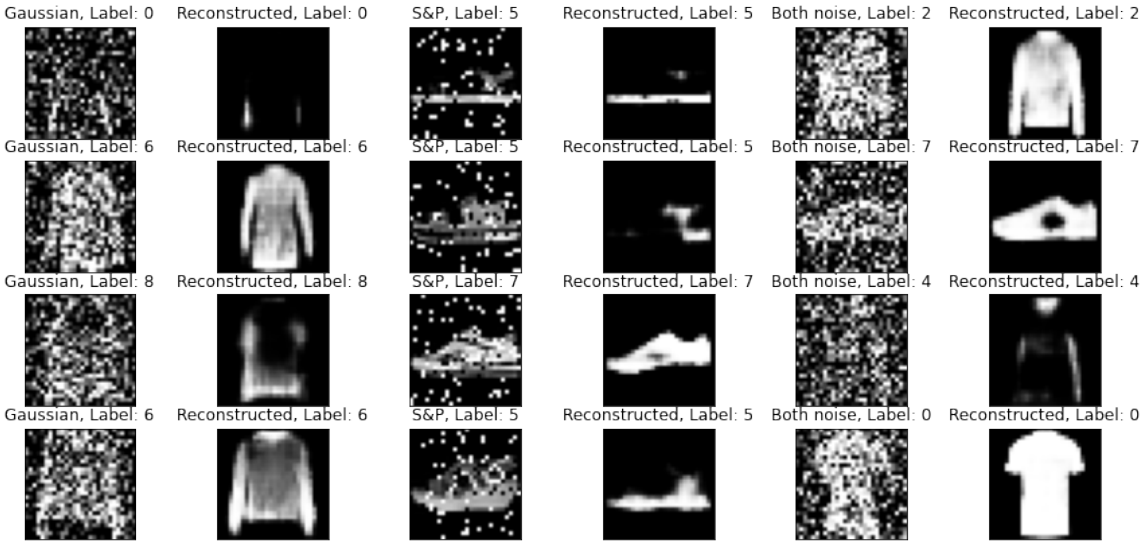


Figure 6: CAE trained with both noise performances.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(90, 384, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(384, 192, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(192, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): Tanh()
  )
)
```

Figure 7: Generator architecture summary

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 192, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(192, 384, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(384, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
```

Figure 8: Discriminator architecture summary