

Deep Learning

Laboratory 4

Q1. In this first task, you will implement a variant of Autoencoder : Denoising Auto Encoder to remove the salt and pepper noise given the noisy version of Mnist Dataset in the ipython notebook : 4-1_0_DCAE_Question.ipynb. (25 Points).

In this first task we are asked to implement a Autoencoder Neural Network which learns to reconstruct MNIST noisy dataset into the dataset without noise. This means to get a noisy MNIST input (see Figure 1) and learn to take the noise off of it for the output (see Figure 2).

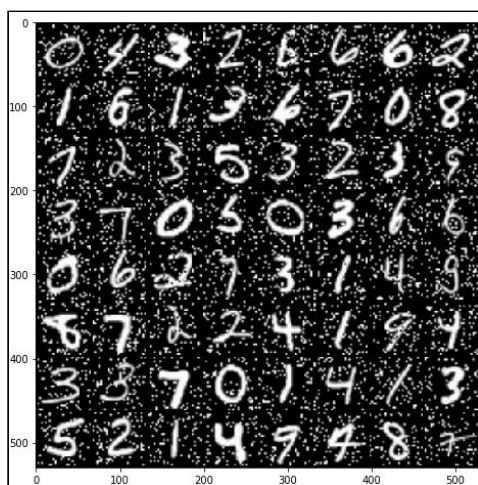


Figure 1

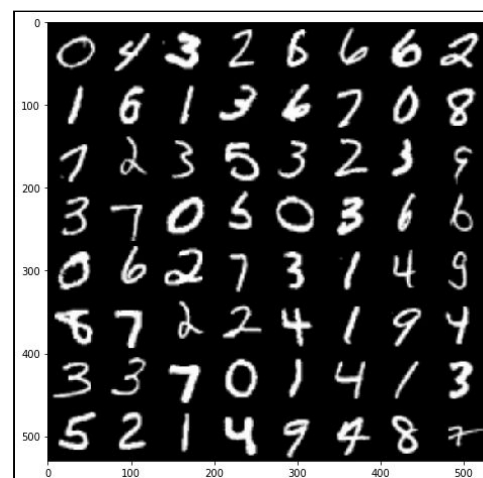


Figure 2

The Autoencoder that we have implemented is the one that can be seen in the following Figure.

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(64 * 64, 256),
            nn.ReLU(True),
            nn.Linear(256, 64),
            nn.ReLU(True), nn.Linear(64, 12), nn.ReLU(True), nn.Linear(12, 3))
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 256),
            nn.ReLU(True), nn.Linear(256, 64 * 64), nn.Tanh())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Figure 3

The implemented Autoencoder is similar to the one of the examples given by the teacher in the ipython notebook: *4-1_0_AEMnist.ipynb*. The only difference is that in this case we were given images of the size 64 by 64. So it changes both the input size of the first linear in the encoder and the one in the last linear of the decoder.

Also we have tweaked with the first size layer output and input of the second in the encoder and consequently in the corresponding layers of the decoder. We changed it from 128 nodes to 256 so in the first layer we do not lose as much information when going from $64*64$ to 256 as going from $64*64$ to 128. We saw a slight improve in the loss and that is why we changed it, see the table underneath.

Second layer size of 128	Second layer size of 256
epoch [1/100], loss:0.0693	epoch [1/100], loss:0.0649

Regarding the loss function, we have used a Mean Squared Error Loss, with an Adam optimizer and weight decay of $1e-5$. Furthermore we have used a learning rate of .001, batch size of 4068 and number of epochs 100. So, with those parameters and Network, after training it we have obtained the following loss plot.

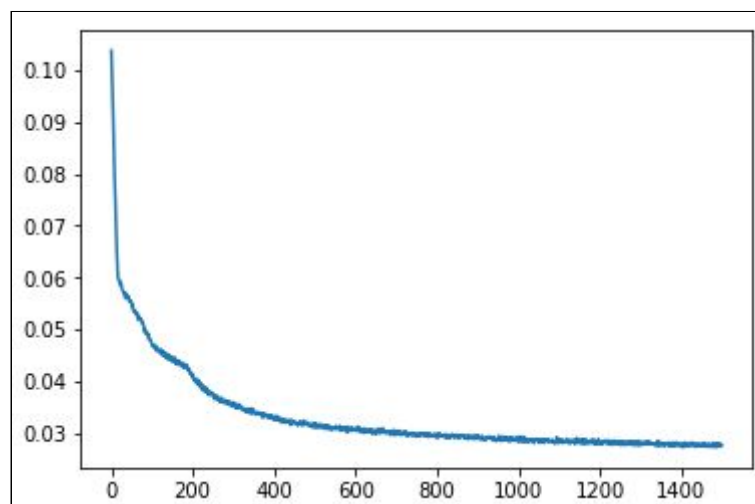
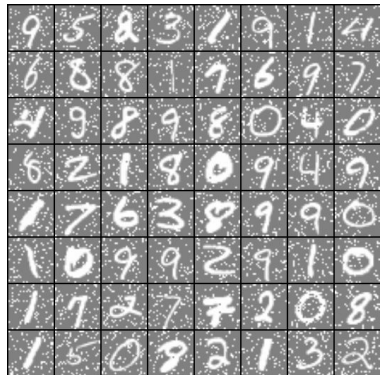

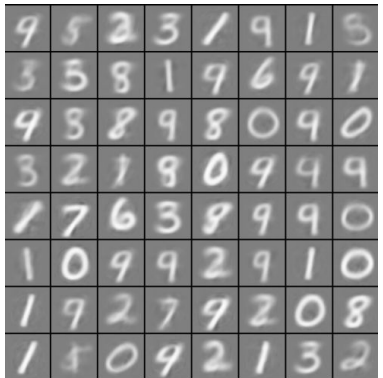


Figure 4

Every 10 epochs we have saved the image predicted output to compare them to the target and the noisy input. Just to check that it has learned almost perfectly let's see all three images for the last iteration of the epoch number 90.

Noisy Image	Target image	Predicted image
		

Q2. In the second task, you will implement the Variational Autoencoder on the CK Dataset in the ipython notebook : 4-1_1_CVAE_CK_Question.ipynb. (25 Points).

In this second task we are asked to implement a Variational Autoencoder Neural Network on the CK Dataset to be able to do the reconstruction of a series of face images of certain people (see Figure 5).

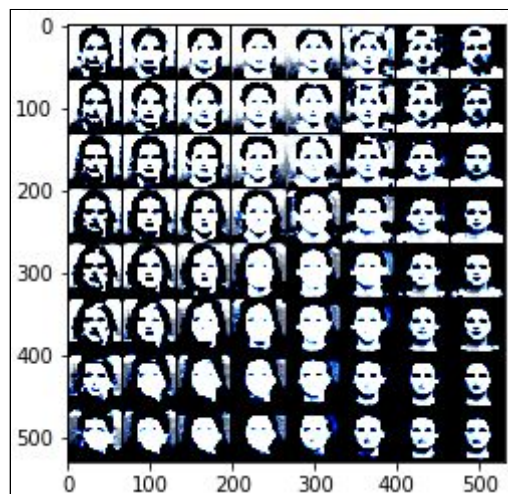


Figure 5

The Variational Autoencoder that we have implemented is the one that can be seen in the following Figures.

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(4, 4), padding=(15, 15), stride=2)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(4, 4), padding=(15, 15), stride=2)
        self.fc11 = nn.Linear(in_features=128 * 37 * 37, out_features=1024)
        self.fc12 = nn.Linear(in_features=1024, out_features=ZDIMS)

        self.fc21 = nn.Linear(in_features=128 * 37 * 37, out_features=1024)
        self.fc22 = nn.Linear(in_features=1024, out_features=ZDIMS)
        self.relu = nn.ReLU()

        # For mu
        self.fc1 = nn.Linear(in_features=ZDIMS, out_features=1024)
        self.fc2 = nn.Linear(in_features=1024, out_features=37 * 37 * 128)
        self.conv_t1 = nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=(4,4), padding=(15,15), stride=2)
        self.conv_t2 = nn.ConvTranspose2d(in_channels=64, out_channels=3, kernel_size=(4,4), padding=(15,15), stride=2)
```

Figure 6

In the above Figure 6 we can see the members of the class of the Convolutional Variational Autoencoder which differs from the traditional autoencoder in the fact that it has convolutions instead of Linears (fully connected layers). Now with the encode, decode functions where this layers are will be clarified.

```
def encode(self, x):

    x = x.view(-1, 3, 64, 64)
    x = F.elu(self.conv1(x))
    x = F.elu(self.conv2(x))
    x = x.view(-1, 128 * 37 * 37)

    mu_z = F.elu(self.fc11(x))
    mu_z = self.fc12(mu_z)

    logvar_z = F.elu(self.fc21(x))
    logvar_z = self.fc22(logvar_z)

    return mu_z, logvar_z
```

Figure 7

```
def decode(self, z) :

    x = F.elu(self.fc1(z))
    x = F.elu(self.fc2(x))
    x = x.view(-1, 128, 37, 37)
    x = F.relu(self.conv_t1(x))
    x = F.sigmoid(self.conv_t2(x))

    return x.view(-1, 4096)
```

Figure 8

On the one hand, there is the encoder in Figure 7 which has as first layers both conv1 and conv2 (convolutions) and then the fully connected. On the other hand, in Figure 8 we have the decode function in which we have the inverse operation of a convolution, which is the Transposed Convolution. So we have a Convolutional autoencoder which has two layer convolutions on the encoder, connected to two fully connected layers that then connect to another two fully connected of the decoder, the output of which goes to two transposed convolutions.

This Convolutional Autoencoder is very similar to the example provided by the teacher in the ipython notebook *4-1_1_CVAEMnist.ipynb*. In this problem case we had to change some parameters to be able to adapt it to the CK dataset. The input size was different since the images have other sizes and the input has 3 channels (as there are coloured faces). So this fact has consequences in the output size of the convolutions which is the input of the first fully connected as well. The same happens with the transpose convolutions. Finally as all those sizes and parameters changed, some *x.view()* calls also had to change.

Then comes the training of the CVAE. For this task we have used the Binary Cross Entropy Loss and as we have Gaussian distributions as well we use Kullback-Leibler to measure the deviation of one distribution and another. As optimizer we chose again the Adam optimizer with 0.000001 learning rate. Furthermore we have used 7 as number of epochs and a batch size of 128.

About the learning rate, we had to lower it from the suggested one, because other wise we were getting negative losses very fast. Regarding the batch size, we had to lower it as well, given that with the suggested one of 256, Google Colab was running out of GPU. We were getting the following error.

```

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:1386: UserWarning: nn.functional.sigmoid is deprecated. Use torch.nn.functional.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.nn.functional.sigmoid instead.")
Train Epoch: 1 [0/6849 (0%)] Loss: 0.002895
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-6-9ee39243a7e7> in <module>()
    17     for epoch in range(1, EPOCHS + 1):
    18         if trainModel:
--> 19             train(epoch)
    20             test(epoch)
    21

----- 2 frames -----
/usr/local/lib/python3.6/dist-packages/torch/autograd/_init_.py in backward(tensors, grad_tensors, retain_graph, create_graph,
    91     Variable._execution_engine.run_backward(
    92         tensors, grad_tensors, retain_graph, create_graph,
--> 93         allow_unreachable=True) # allow_unreachable flag
    94
    95
RuntimeError: CUDA out of memory. Tried to allocate 686.00 MiB (GPU 0; 14.73 GiB total capacity; 12.81 GiB already allocated;

```

Figure 9

Following the definition of those params, we trained the CVAE. The obtained plot of the loss during the training is the following.

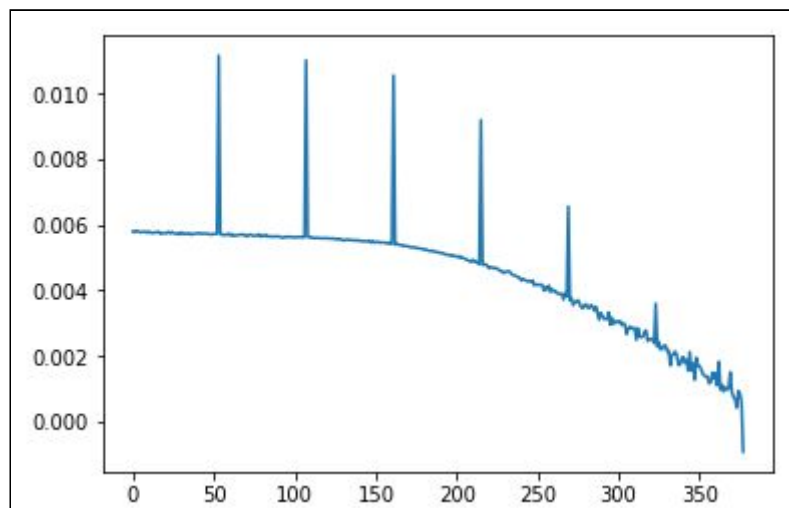



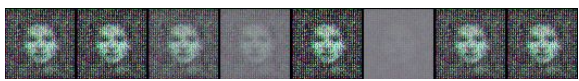


Figure 10

The result obtained in Figure 10 follows a normal training process in which the loss goes down as the epoch number goes up. As can be seen it ends up converging to a number very close to 0, which means that the error between the reconstructed faces and the target ones is not that much.

Just to check how this reconstructed faces are, and that they do not differ that much from the original, we see both a target image sample and an output in the table below.

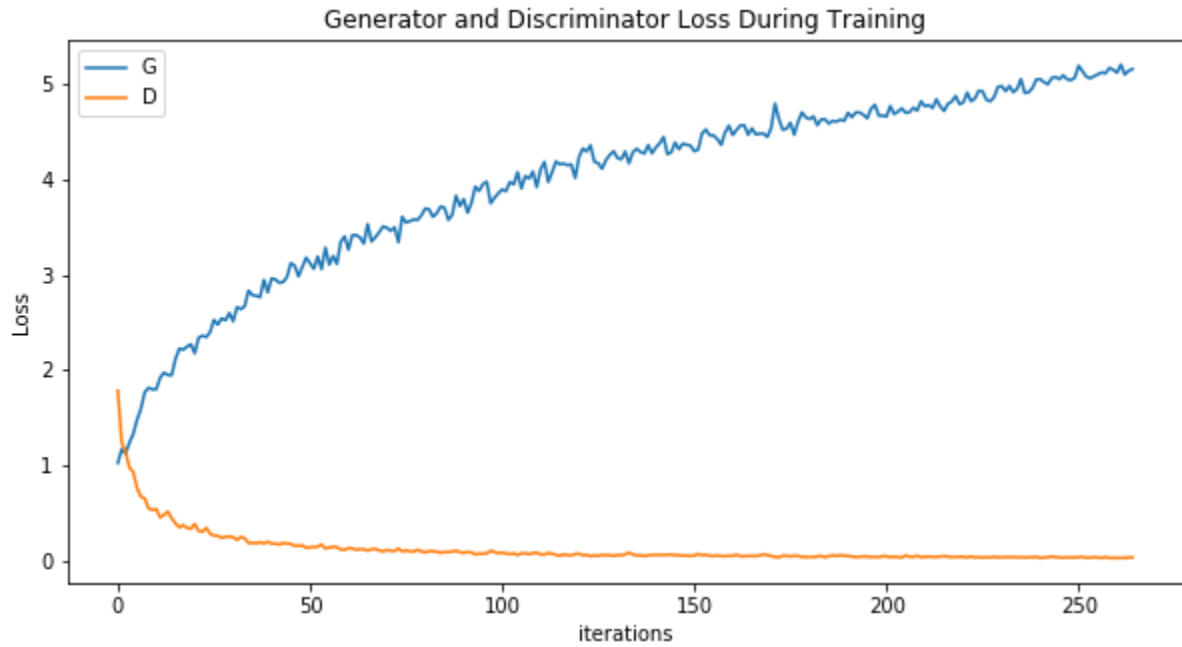
Original Image	Reconstructed Image
	
	

Q3. In the third task, you will implement the Generative Adversarial Neural Network on the CK Dataset in the ipython notebook : 4-1_2_CGAN_CK_Question.ipynb (25 Points).

In this exercise we have implemented a DCGAN model using transposed convolutional layers for the generator and convolutional layers for the discriminator.

For our experiments we have used a batch size of 128, an image size of 64x64, a learning rate of 0.0002 and an Adam optimizer.

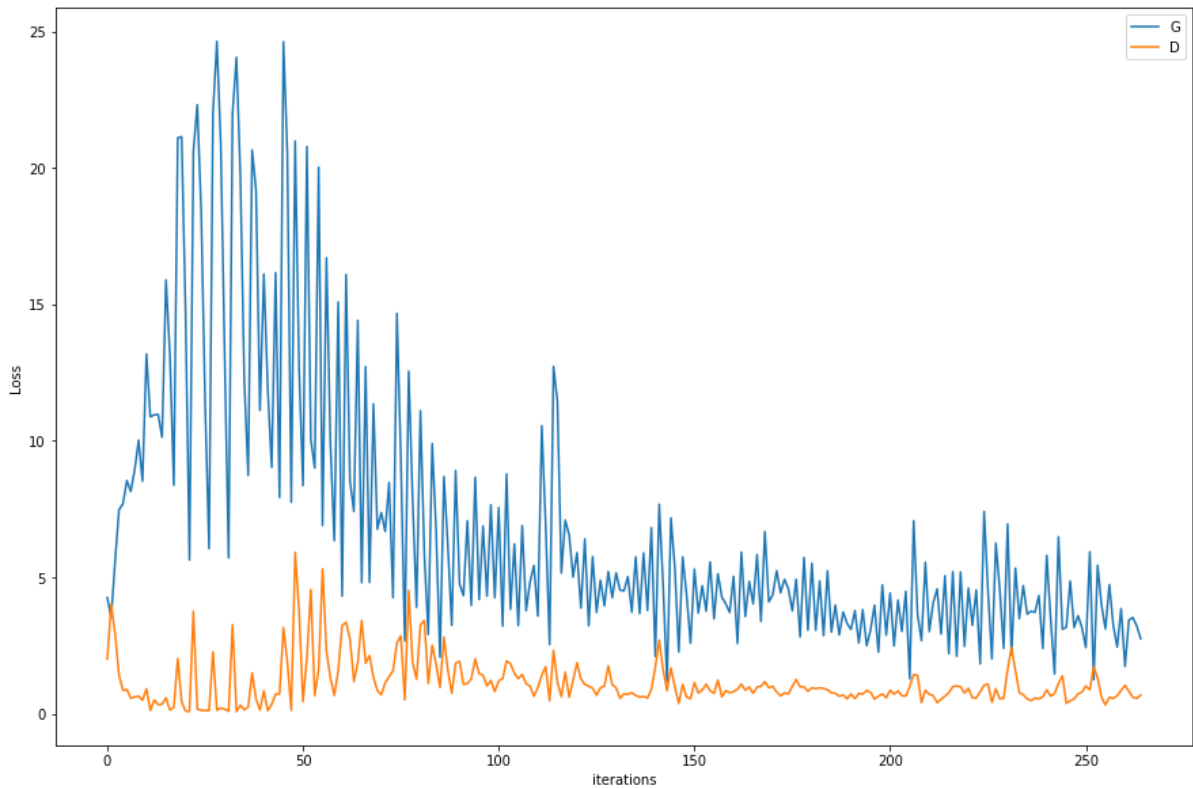
When we first trained the Network we noticed that the discriminator was learning way faster than the Generator and hence we had the vanishing gradient problem where the Generator was not able to learn because the Discriminator had learned too much. We can see an example in the following image:



Here we see how the error of the discriminator diverges to infinite while the error of the Discriminator converges to 0.

We wanted the discriminator to learn to discern between real and fake samples at the beginning to send good gradients to the Generator and allow him to learn how to generate real images and fool the Discriminator eventually. We tried to multiply the Generator's learning rate by 10 but we obtained the inverse results, the discriminator was not able to learn anything and was classifying fake samples as real ones. We saw though that playing with this parameter we would be able to obtain the desired results. On our final results we have used a learning rate of 0.0004 for the Generator and the results can be seen on the next picture:

We first show our learning curves for both, the Generator and the Discriminator by plotting the losses over iterations:



As we can see, although the losses are really unstable we eventually reach the point that we wanted where the loss of the generator decreases and the loss of the discriminator increases. This happens at the iteration 50 more or less and then it starts to converge. Also, notice that we just trained the model for 5 epochs and it is already able to output results like the following:

Víctor Pérez 183950

Diego Vincent 174458

Óscar Font 183826



We can see that although they are not perfect, faces can be clearly seen.

Some interesting things about this question have been the difficulty of parameter tuning to make this models converge and all the time we had to wait to see if each parameter was working or not.

Q4. In the last task, you will implement conditional GAN on on the CK Dataset and optionally with VAE structure: 4-1_3_CondGan_CK_Question.ipynb (25 Points).

In this last exercise we have used the generator and discriminator architecture used in StarGAN as baseline. We have embedded a vector representing the class of each sample in a regular Generator and now we do not just detect real or fake samples in the Discriminator but we also make that it learns to classify each image to a certain label.

The Generator architecture consists on a bottleneck architecture where each input image was first downsampled several times using convolutional layers, then several other convolutional layers with residual connections were added and finally they used a series of Transposed Convolutions which resulted in the final RGB image. The idea behind the architecture of this network is to let it learn which parts of the input image it needs to change. The latent component (which could be interpreted as the values in the middle of this network) is created from the input image so it can keep all the information that it wants and decide to keep or drop the desired values from it.

We attacked the issue as if it was a multiple conditioning problem. We have concatenated the class label vector to each input image of the Generator creating a $H \times W \times (C+L)$ matrix where H , W and C represent the height, width and number of channels respectively of the original image and L the number of possible labels that in this case is 8. This concatenation requires a previous upsample of each component of the class vector. This upsample is done by just creating a $H \times W$ matrix for each element with their same value in all its components. So if we had the vector $[0, 1, 0]$ we would obtain 3 $H \times W$ matrices, two filled with zeros and one filled with ones.

For the Discriminator we have used the popular PatchGAN architecture. This network consist on a set of convolutional layers with the goal of predict the probability of 70×70 overlapping patches from the input image of being real.

We added a component to this architecture to make the regression of each class vector in a Conditional GAN fashion at the same time that we compute the probability of each image of coming from the real dataset or not.

We tried to use as much code as we could from the previous exercise, so we have used almost the same parameters. One of the differences is that the batch size was reduced to 32 because this model was way bigger and it took to long on print every update and now we use the same learning rate for both the Generator and the Discriminator.

What we observed when we started training this model was that the Generator was outperforming the Discriminator. We hypothesized that this was caused because now the

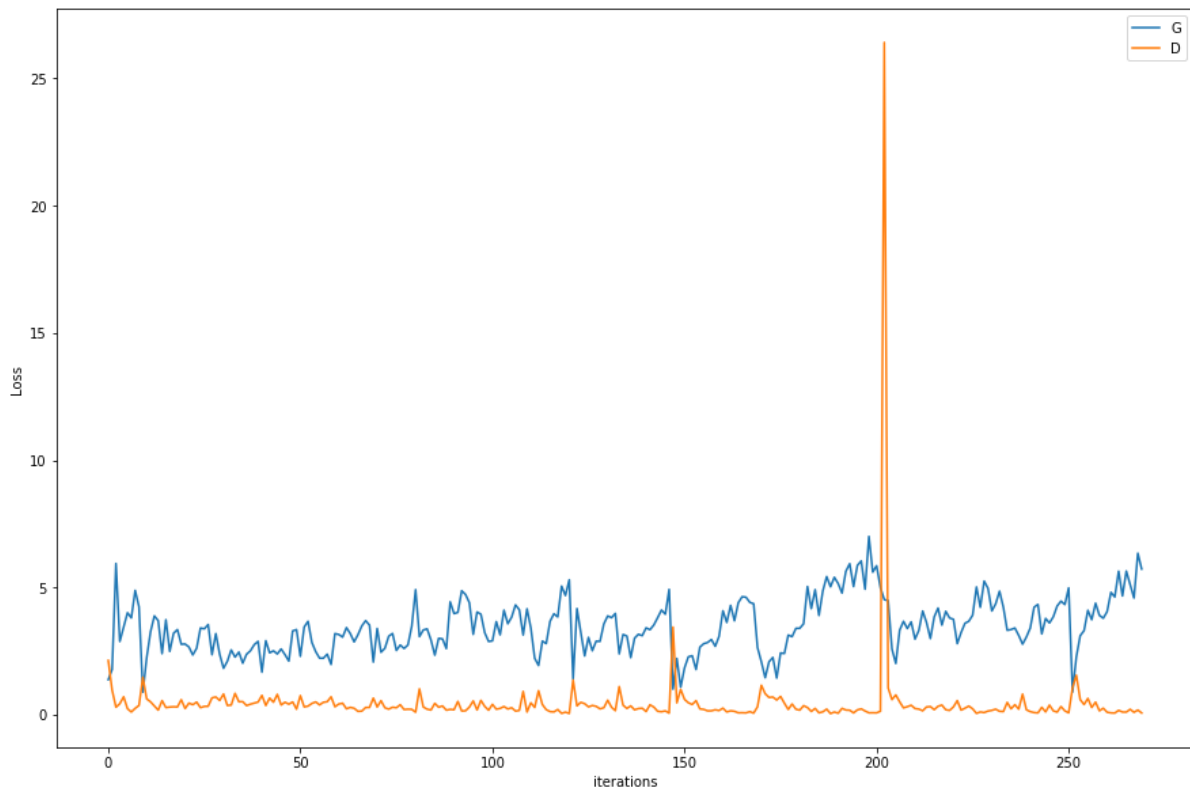
Víctor Pérez 183950

Diego Vincent 174458

Óscar Font 183826

Generator contains image information it is able to generate realistic samples from an early step. To solve this issue we trained the Generator one time for every 4 steps in the Discriminator. This resulted in better results where both Networks were able to learn at their correct pace.

With this change we were able to maintain the loss stable for both components as we can see in the following image, where at some point we had high values but in average is stable.



Our final faces can be seen in the following images:



We had no enough time to do an extensive study about if we were really able to achieve this class translation, bot at least the output images look like faces.

Víctor Pérez 183950
Diego Vincent 174458
Óscar Font 183826

We spent too much time working on the code of this last exercise that we did not have time to experiment more with it, it also took a long time to train which made even more difficult to play with it and try to change more features of it.