Digital Forensics - Generative Adversarial Networks

Carlo Facchin (1234374)

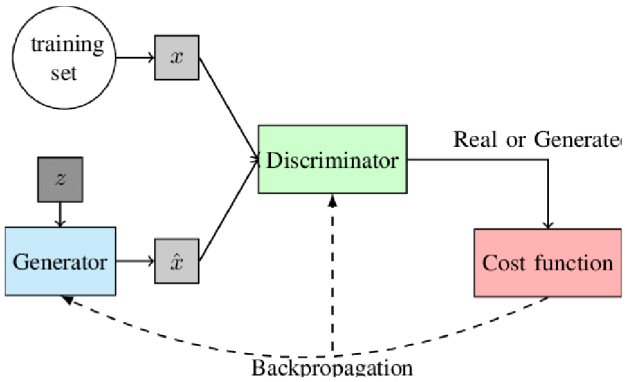
Project Report

1 Introduction

Generative Adversarial Network (GAN), introduced for the first time by Ian Goodfellow in 2014, is an architecture used to train deep-learning based generative models.
The framework is formed by two neural networks, one generator and one discriminator, that are trained together in a competitive zero-sum adversarial manner in order to learn and be able to generate new plausible examples that ideally are indistinguishable from real samples given in the training dataset [1].

In this report we implemented first a Deep Convolutional Generative Adversarial Network (DCGAN) following the guidelines provided by [2] using Python and Keras on MNIST-DIGIT and FASHION-MNIST datasets. After that, we implemented a second Generative Adversarial Network using instead the PyTorch framework on the CelebA dataset.

2 DC-GAN Structure

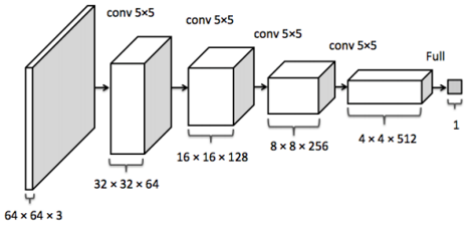


The discriminator takes as input samples from the training dataset and is trained directly on real and generated images to learn the distribution and be able to distinguish real or fake samples.
The generator instead is trained via the discriminator through a cost function, for binary classification case the binary-cross entropy function is commonly used

$$V(D,G) = \underbrace{\mathbb{E}_{x\sim p(x)}[\log D(x)]}_{\text{true samples}} + \underbrace{\mathbb{E}_{z\sim q(z)}[\log(1 - D(G(z)))]}_{\text{fake (generated) samples}}$$

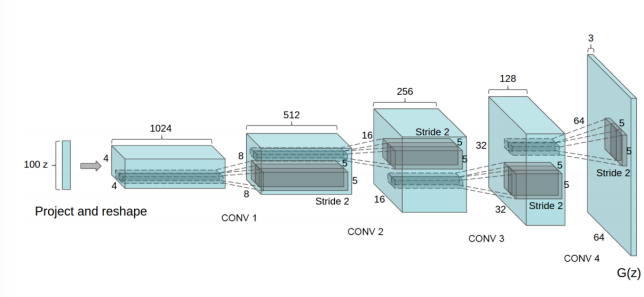
The discriminator try to maximize the reward V(D,G), meanwhile the generator try to minimize[log(1 – D(G(z)))] in order to reach the Nash Equilibrium where the discriminator reach the probability of 50% to distinguish between real-generated images.
In practice this cost function saturates for the generator, means that it cannot learn quickly as the discriminator, ending with the 'defeat' of the generator, and the model cannot be trained effectively. To overcome this saturation the cost function has been changed letting the generator maximize[log(D(G(z)))][1].

2.1 The Discriminator



The discriminator is a convolutional neural network formed by a sequence of Convolutional layers with Leaky ReLU activation function, Batch normalization layer and Dropout layer, followed in the end by a Flattening and a fully connected output layer with a single-neuron that has value 0 or 1 to classify is the inputted image is real or fake.
At each Convolutional Layer, the size of the input image is halved to learn 'features' at different scale of the image.

2.2 The Generator



The generator takes as input a randomly generated vector weights from the latent space and generates a grayscale image. There is no specification for the latent space structure, so we used vectors formed by 100 Gaussian-distributed values as input for the generator.
The generator is usually built with a fully connected Layer at first, and a sequence of up sampling followed by batch normalization layers. In the end finish with a convolutional output layer that reshape the network output to our image size.
In the literature is advised to use a Conv2dTranspose as up sampling layer [2], but [3] states that a normal up sampling followed by a convolution removes some checkboard artifacts that Conv2dTranspose layer generates.

3 Training

The training phase is the most critical part of the whole framework because we must train both generator and discriminator at same time to achieve the Nash Equilibrium.
The training phase is subdivided in epochs, in each epoch the discriminator and generator are trained on batches, that define the number of samples to work through before updating the internal model parameters. It is also important to highlight that during the training phase of each one, the other one's network weights are frozen.
The discriminator is trained on batches formed half by real sampled training images and half by images generated by the generator with appropriate real/fake label to distinguish the two cases. The generator instead is trained over the Discriminator's error by feeding the whole GAN framework with several latent-space vectors equal to the batch size.

DC-GAN is known to be hard to train due to the sensitivity to small changes in parameters, and one way to identify if the learning process is going well is to check the discriminator and generator losses: generally in a stable GAN the discriminator loss should be around 0.5 and 0.8, and the generator loss, typically higher.
A common problem during the training phase is the convergence failure in which the model loss cannot stabilizes during the process. In particular we can have a scenario where the Discriminator dominates: the generator score reaches zero or near zero and

the discriminator score reaches one or near one. A second scenario instead is when the Generator that dominates, having a score that reaches one.
In both cases the we diverge from the Nash Equilibrium very rapidly [6] and is possible to verify the problems also checking the generated images quality. With very noisy and random generated images in fact, its easy for the discriminator to classify them correctly leading to instability.
Another possible problem is the so called Mode Collapse in which the generator learns to generate only a small subset of images, leading to low variance of possible outcome and often generate training-set elements with different input noise.
To solve these problems, literature can give guidelines for the network modeling and parameter selection. In particular some suggestions from [2] and [5] report the importance of:

- Normalize the images values between [-1, 1] and use Tanh activation as the last layer of generator output.
- Use the modified loss function for the generator maxlog(D(G(z))) by flipping the generated images label as 'real' instead of 'fake'.
- Sample the latent space noise vector from gaussian distribution.
- Uses mini-batches that contains only real or generated images and normalize them.
- Use Leaky ReLU on both generator and discriminator.
- Use Adam Optimizer with learning-rate=0.0002 and momentum(beta1)=0.5.
- Use Dropout in the discriminator.

4 Implementation

We implemented the network using Python and Keras framework for the first network, PyTorch for the second, as reference we used [7], [8], [4]. For the DCGAN we directly used the Sequential model from Keras. The discriminator receive as input a four-dimensional vector of size (None, 28, 28, 1). The first None element is needed by the Keras lib and is added after we load the dataset that is formed by 60'000 28x28 gray-scale images. Following we have a sequence of 2 Convolutional, LeakyReLU, Dropout Layer and a final flattening and fully connected layer for the output. For the Convolutional Layers we tried to use higher number of features, 128 and 256, but in the end there is no noticeable improvement. We used a kernel size of 5, we tried also 3 but the results seems not to change. The Batch-Normalization use the default momentum value, same as the LeakyReLU. After some test the Dropout-rate value chosen is 0.3.
The loss-function is the binary-cross entropy and the optimizer is Adam with learning-rate=0.0002 and momentum=0.5.

In total the discriminator has 212,865 parameters.

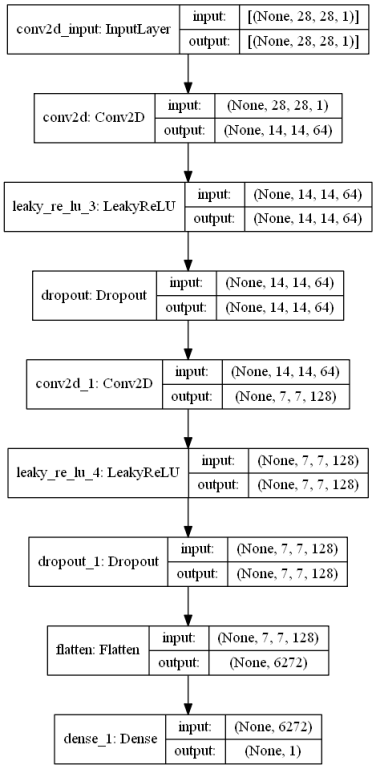


Figure 1: Discriminator structure

The generator receive as input a noise-vector of size 100 drawn from the latent-space, after that they are fully connected to the second layer formed by 7 * 7 * 256 = 12544 neurons.
This is the base where start Upsampling. We used 2 Convolutional layers that uses 128 and 64 features, kernel of 5 in inverse size respect the discriminator. As before we used BatchNormalization and LeakyreLU, same loss-function and optimizer parameters.
In total the generator has 2,330,944 parameters.

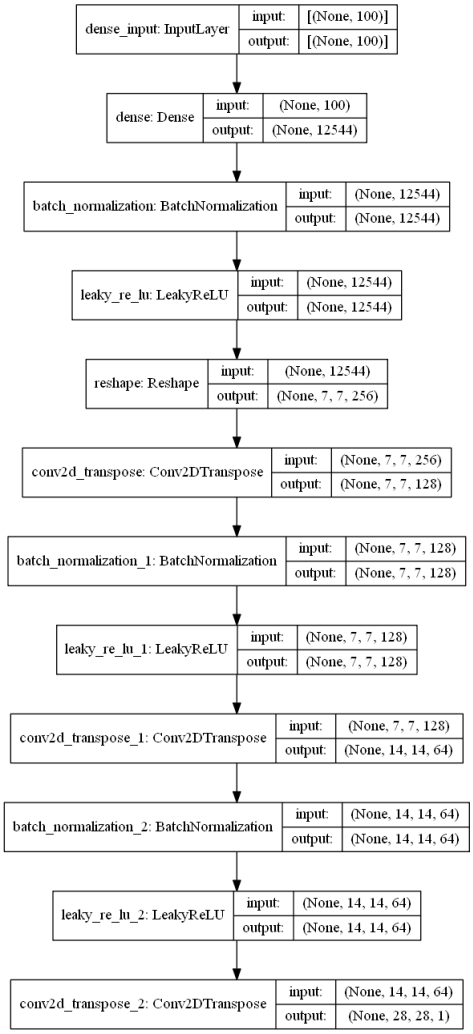


Figure 2: Generator structure

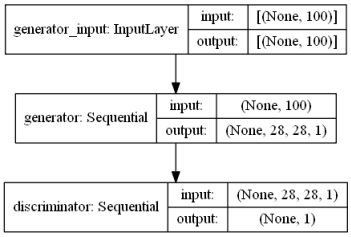


Figure 3: Combined GAN structure

For the dataset import we preprocessed it normalizing the size and slicing it already in batches of size 128 and shuffling it. After initializing the network and the parameters the train iteration of the Network follows those steps. The discriminator training: for

each epoch is sampled a batch of random points in the latent space and it is turned into fake images using the generator. Then real and generated images are combined and the discriminator is trained to classify generated vs. real images.
Then the generator is trained, using the label 1 in order to let the generator try to create images that cannot be detected as false ("fooling" the discriminator).
The saved metrics are the losses and accuracies from generator and discriminator. In each epoch we also save both the generator and discriminator model ad the generated image based on a 16 latent-space vector generated at beginning of the the training phase in order to be able to see how much is learned in every epoch.

The second network use instead the CelebA dataset that uses images of the celebrity faces. The input data consist of 3x64x64 images and it is normalized with mean=0.5 and std=0.5. The batch size dimension is 128. Since the both the dimension and the channel are a lot higher than the MNISTs datasets, using a single local machine with limited resources made difficult to test the networks in terms of time and GPU memory consumed.
The discriminator structure consist of five convolutional layers with kernel dimension equals to 4, batch normalization, LeakyReLU as activations except for the Sigmoid output. The generator similary have 5 convolutional layers with kernel of 4, batchNorm, but Relu and Tanh as output activation.
The loss function used is BCELoss while as optimizer for both the generator and the discriminator has been used Adam with learning rate of 0.0002.

5 Results

The model has been trained for 50 epochs using as GPU a Nvidia GTX 1060Ti. The following sample are generated in different intermediate epochs by the generator on MNIST-Digits and MNIST-Fashion dataset. The two larger images plots represent the generation at the final epoch of the training.
Even if the digits have well defined edges, some are still difficult to recognize. This effect is increased on the generated clothes.

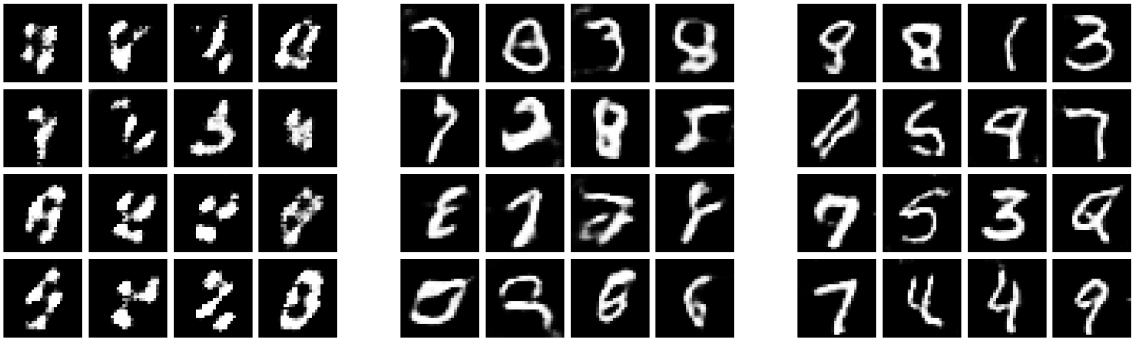


Figure 4: Generated MNIST images at epochs 1 - 10 -20

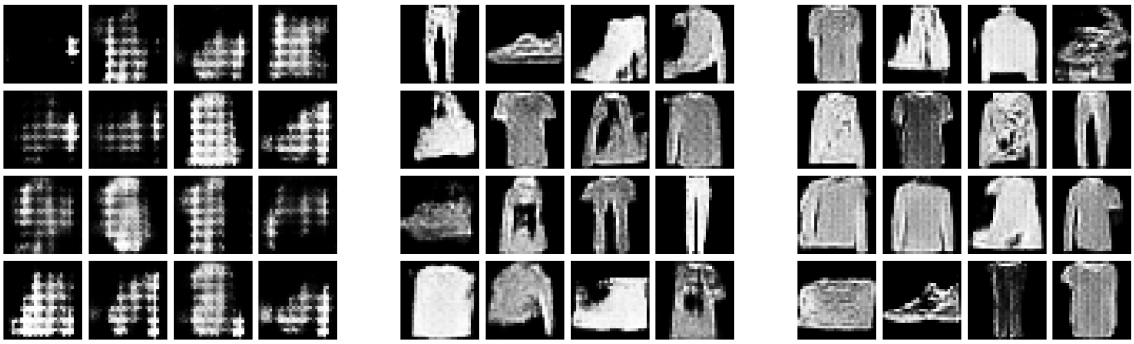


Figure 5: Generated Fashion MNIST images at epochs 1 - 10 -20

Figure 6: Generated images for MNIST and Fashion MNIST

The following graphs are the loss of discriminator and generator of the DCGAN trained on MNIST and Fashion MNIST datasets. Is possible to notice that both reach a sort of equilibrium in very few epochs, maintaining a balance between 0.6 and 0.75 for all the

training, as expected with the discriminator loss a bit lower and more stable.
The discriminator accuracy is intended as the probability to correctly identify the real samples, and generator accuracy is the probability that the generated fake sample fool the discriminator. The generator accuracy remain always very low at about 5% while the discriminator one at about 60%.

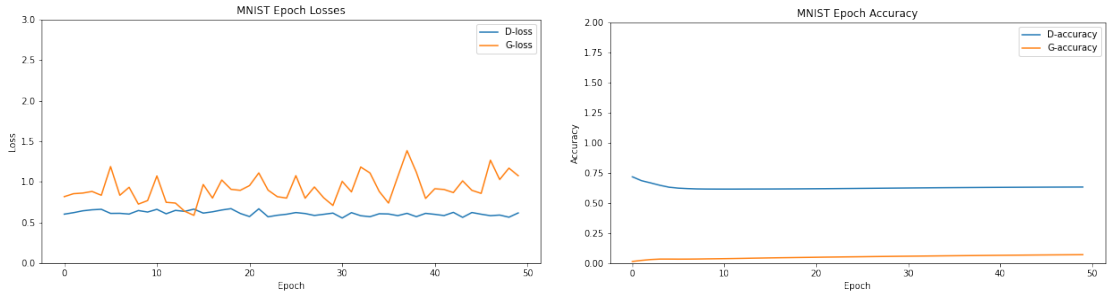


Figure 7: MNIST losses and accuracy plot

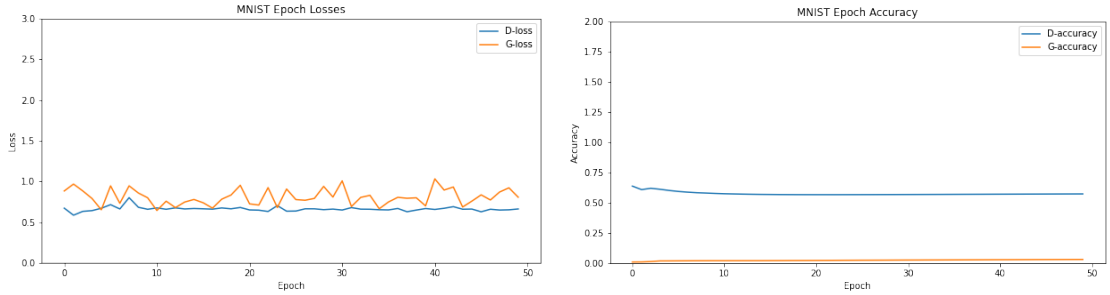


Figure 8: Fashion MNIST losses and accuracy plot

For the CelebA network unfortunatly the time for training the model was more than I expected so I used only few epochs. I tried to train the model for 20 epochs in 13525 seconds. While the losses of the discriminator keep decreasing, improving the classification of real and fake images, the generator loss started loosing the minmax game only after few iterations diverging from the discriminator loss. For this reason the resulting generated images are the generator output after only few epochs.

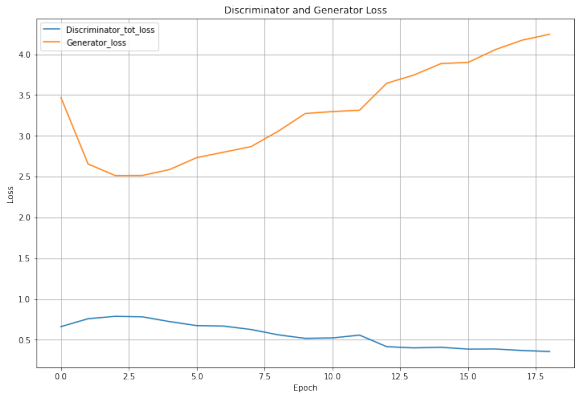


Figure 9: Discriminator and generator losses

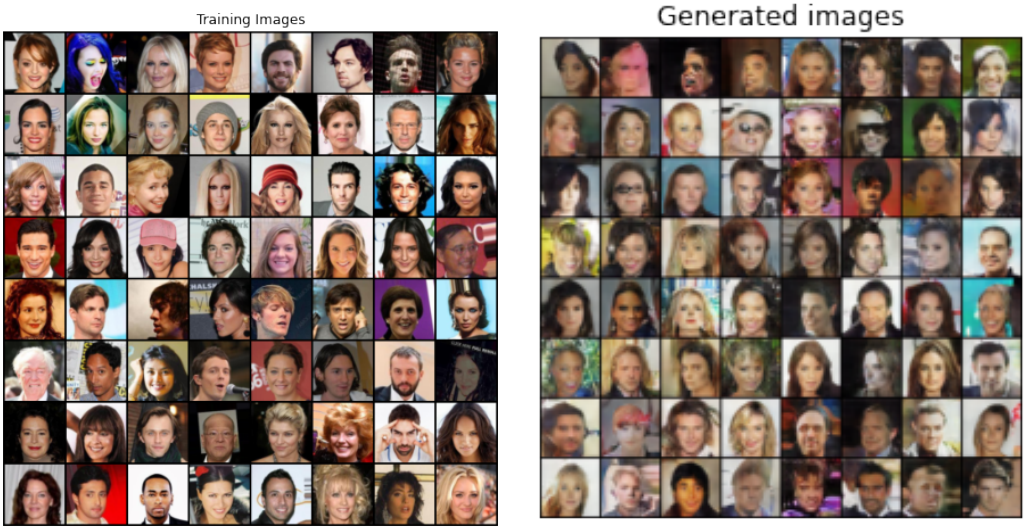


Figure 10: Real and generated images

References

[1] Ian 1 Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

[2] Alec 2 Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* (2015).

[3] Mehdi 3 Mirza and Simon Osindero. "Conditional generative adversarial nets". In: *arXiv preprint arXiv:1411.1784* (2014).

[4] 7 Jason Brownlee. 2019. https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/.

[5] 4 Soumith Chintala. *How to train GAN? Tips and tricks to make GAN work'*. https://github.com/soumith/ganhacks. 2016.

[6] Ned Gulley. *Monitor GAN Training Progress and Identify Common Failure Modes*. 2021. https://it.mathworks.com/help/deeplearning/ug/monitor-gan-training-progress-and-identify-common-failure-modes.html.

[7] 5 Nathan Inkawhich. *Faces DCGAN introduction*. 2020. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

[8] 6 Margaret Maynard-Reid. *Get Started: DCGAN for Fashion-MNIST*. 2021. https://www.pyimagesearch.com/2021/11/11/get-started-dcgan-for-fashion-mnist/.