Machine Learning Engineer Nanodegree

Capstone Project

Roser Sara Garcia

December 31st, 2019

I. Definition

Project Overview

The applications of GANs (Generative Adversarial Networks) are endless. From science (they can improve astronomical images and simulate gravitational lensing for dark matter research), video games (to up-scale low-resolution images) to fashion, art and advertising (where new photographs can be generated without the need of models, photographers, etc), amongst many others. This class of machine learning systems were invented by Ian Goodfellow and his colleagues in 2014, as described in this paper, and has been extensively evolving since then.

GANs consist on having two neural networks, the generator and the discriminator, which compete against each other in a game; the generative network generates candidates while the discriminative network evaluates them. The objective of the generator is to increase the error rate of the discriminative network (meaning that the discriminator is accepting the synthetic images as real images). Backpropagation is applied to both networks so that the generator produces better images, while the discriminator becomes more skilled at flagging synthetic images. [1]

In this project I created a master painter system that produces synthetic abstract paintings. The system used will be the unsupervised machine learning DCGAN (Deep Convolutional Generative Adversarial Network), described in the paper Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks.
Regarding the dataset, the system will be trained with a dataset of abstract paintings from the Wikiart.org.

Problem Statement

The goal of this project is to create an unsupervised machine learning system that can generate synthetic paintings based on art work.

To accomplish this, the following tasks will be performed:

1. Retrieve the dataset from the Wikiart.org and store it in a directory.
2. In a Jupyter notebook:
    a. Create the dataset as well as the data loader.
    b. Create the Generator and the Discriminator classes using PyTorch.
    c. Create the loss functions and optimizers.
    d. Create the training script.
    e. Validate the results.

Metrics

As our goal is to generate synthetic images that resemble real images, one way to validate if our model is producing good enough images is to have its output being evaluated by a classification model.

For this, we will train our model as well as a classification model with the MNIST database. Next, we will check the accuracy obtained, which will tell us about the performance of the classifier.

$$accuracy = \frac{true\ positives + true\ negatives}{size\ of\ the\ dataset}$$

Then, we will attempt to classify the images generated by the DCGAN with the classification model.

Apart from this, we will also look at:

- The losses of both the discriminator and the generator network during the training process.
- A visual comparison between the synthetic paintings and the real paintings.

II. Analysis

Data Exploration

As the ultimate goal is to generate synthetic paintings that resemble real paintings, we need to train the Discriminator model with real paintings. For this, the dataset used will be a subset of paintings from the Wikiart.org repository.

The abstract paintings from the Wikiart.org can be downloaded form this Google Drive file. The images here were taken from a much bigger dataset of paintings. [2]
The dataset contains 6,000 images and each of them have a size between 50KB and 5KB approximately. To train the model all the images will be resized to ultimately have all RGB images with the same size 3x64x64.
At the same time, we also need to train the Discriminator with fake paintings. These images will be the ones produced by our Generator, which in turn, will have an input of a generated fixed batch of latent vectors drawn from a Gaussian distribution.

Apart from this, we will also be working with two more datasets: the Large-scale CelebFaces Attributes (CelebA) Dataset and the MNIST database of handwritten digits.

The CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200,000 celebrity images, each with forty attribute annotations. The images in this dataset cover large pose variations and background clutter. CelebA has large diversities, large quantities, and rich annotations. The size of the images is approximately 10K each.
This dataset will be used to be able to objectively compare our system with the selected benchmark. (More information about this can be found in the Benchmark section below).

Regarding the MNIST database of handwritten digits, it has a training set of 60,000 examples, and a test set of 10,000 examples. The images contain grey levels and are centered in a 28x28 field.
The MNIST database will be used to evaluate the outcome of our generator, as explained in the metrics section.

Exploratory Visualization

In the case of developing a system that generates fake images, it is important to know how the images in the dataset look like in order to set expectations of what it will produce during the training process.

Abstract art work dataset

This dataset is very broad on the type of images that you can find. The idea of using these images is to teach the system what is expected from art work and see what new contributions it can make.

Fig. 1 Images of abstract art work from the Wikiart.org.

Celebrities faces dataset

As we can see in Fig. 2, this dataset is full of celebrities faces from woman and man with different attributes each.

The idea behind using this dataset is to be able to create fake images of people that does not exist.

Fig. 2 Images of celebrities' faces from the CelebA dataset.

MNIST database

Our last dataset is made up of handwritten digits (see Fig. 3). What we will expect here is that our generator creates new samples of handwritten digits that can be understood by humans.

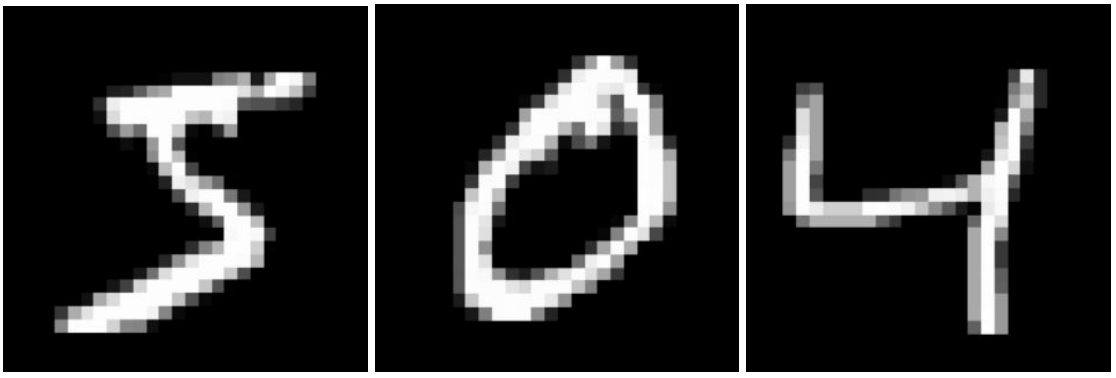


Fig. 3 Images of handwritten digits from the MNIST dataset.

Algorithms and Techniques

The proposed solution for this problem is to use DCGAN. As explained in the first section, DCGAN is a kind of unsupervised machine learning system that is composed of two different networks.

- Discriminator: Binary classifier made up of convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 image and the output is a scalar probability indicating if the input is a real image.
- Generator: Convolutional neural network comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, that is

drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided convolutional-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image.

The loss function used will be the Binary Cross Entropy loss. Regarding the optimizers, as specified in the DCGAN paper, both will be Adam with learning rate 0.0002 and Beta1 = 0.5.

To train these two networks, we will construct mini-batches for real and fake images.

First, we will train the discriminator, which will have a mini-batch of real images and then a second mini-batch of fake images generated by the generator. Then, with all the gradients accumulated from all real and fake images' batches we will call the discriminator's optimizer. The goal here will be to maximize the probability of correctly classifying an image as real or fake.

Secondly, we will train the generator. In this case, we will aim to maximize the probability of the discriminator to classify fake images as real. To accomplish this, we will classify the generator output from the discriminator training part, computing the generator's loss using real labels as ground truth, computing the generator's gradients in a backward pass, and finally updating the generator's parameters with an optimizer step.
The reason why we are using the real labels in the generator training part is because we want to maximize the probability of the discriminator to classify fake images as real, instead of minimizing the probability of detecting fake images. The latter was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process.

To check the results we will look at the losses of both networks as well as a visual comparison between the real images and the synthetic ones. [3]

As explained in the metrics section, a supervised machine learning classifier will also be developed in order to check if the images generated by the DCGAN system seem real or not. To do this, we will create a neural network of three hidden layers, each of them with a ReLU activation function and dropout. Next, we will create a training script where training loss, test loss and accuracy will be measured periodically.

Benchmark

As a benchmark model I have selected the one in this tutorial.

To be able to objectively compare the solution against the results in the selected tutorial, the DCGAN implemented here will be trained (as well as in the tutorial) with the Celeb-A Faces dataset. Next, we will be able to compare the performance.

III. Methodology

Data Preprocessing

Before training the models in the DCGAN system, the following preprocessing is done:

1. All images are resized, cropped and normalized in order to fit with the requirements of the models in a way that the dataset is independent from the used dataset.
2. A data loader is created with shuffled images.

Regarding the classifier, the following steps are performed:

1. The images are normalized.
2. Create a data loader for the training and testing set with shuffled images.

For the prediction of the fake images in the classifier it is necessary to:

1. Save the images corresponding to the last batch of our DCGAN (see Fig. 4 as an example of output images).
2. Next, the image is cut in 64 smaller images of similar size, each containing one unique number.
3. Then, the images are loaded, resized and then converted into a Tensor in order to be an appropriate input for the classifier.

Fig. 4 Output of the generator corresponding to the last batch.

Implementation

The implementation of this project can be broken down into three parts: benchmark model comparison, model evaluation and generation of art work. Let's have a look into each of them.

1. Benchmark model comparison

This section is comprised in the Jupyter Notebook called 1-benchmark-model-comparison. Here, we attempt to reproduce the results achieved in the selected benchmark model. This way, we are able to objectively tell that we got the level of performance that we were aiming for. The dataset used in this first part corresponds to the CelebA.

The implementation for DCGAN can be broken down into the following sections:

- Weight initialization strategy

As stated in the DCGAN paper, the model weights should be initialized from a Normal distribution with mean=0, stdev=0.02. This is accomplished by using a function weights_init, which is called immediately after model initialization.

- Generator

The goal of the Generator is to produce an image similar to the ones from the original dataset from a latent space vector.

This is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of [-1,1] .

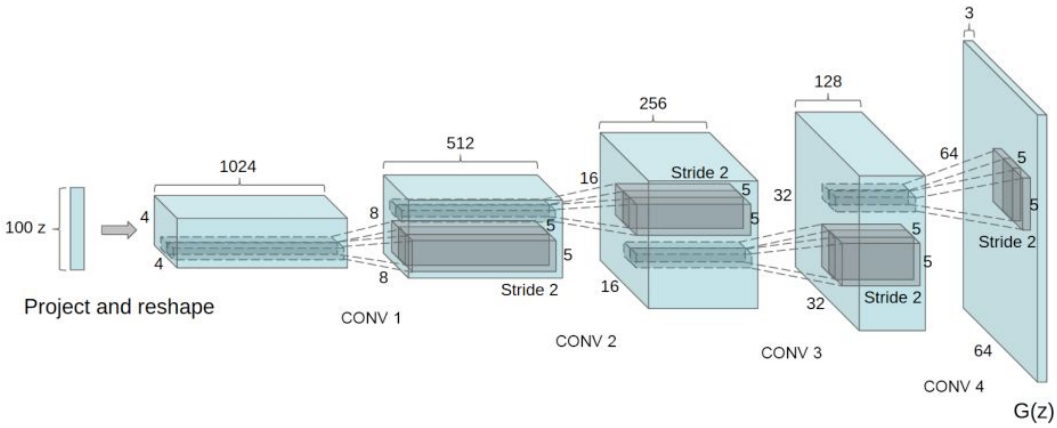


Fig. 5 Generator architecture. Source: DCGAN paper, Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks.

- Discriminator

To goal of the Discriminator is that given an image, it can tell if that image is real or fake. For this, it is implemented as a binary classification network that takes an image as input and returns a probability indicating whether the image is real or not. The network is built up by a series of BatchNorm2d, LeakyReLu, Conv2d and ultimately has a Sigmoid activation function.

- Loss functions and optimizers

Binary Cross Entropy loss (BCELoss) is used as a loss function. The reason for this is that it allows to compute the losses for each component of the system using the y parameter, due to the BCELoss formula nature (see Fig. 6).

```
def CrossEntropy(yHat, y):
    if y == 1:
      return -log(yHat)
    else:
      return -log(1 - yHat)
```

Fig. 6 Code implementing the Binary Cross Entropy formula. Source: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

To compute the losses it is necessary to have some labels that indicate if the images are real or fake. For this, as mentioned in the DCGAN paper, real images are labeled with 1, whilst fake images with 0.

Regarding the optimizers, according to the DCGAN paper, the most suitable are Adam optimizers with learning rate 0.0002 and Beta1 = 0.5. There will be one optimizer for the generator and another one for the discriminator.

- Training loop

The training script will consist of mini-batches of real and fake images. During the first part of the script the discriminator is trained, followed by the generator.

Updating the Discriminator

In order to optimize the Discriminator two iterations are performed:

- A batch of real samples from the training set is forward pass through the Discriminator, then the gradients are calculated in the backward pass.
- Then, a new forward pass is done with fake images generated by the Generator and again the gradients are calculated in the backward pass.
- Finally, the accumulated gradients are used to optimize the Discriminator.

Updating the Generator

To optimize the Generator we will attempt to maximize the result of the Discriminator when evaluating the Generator's images. This is accomplished by classifying the Generator output from Part 1 with the Discriminator, computing the generator loss using the real labels, computing the gradients in the backward pass and finally updating the parameters in the optimizer step.

2. Model evaluation

The evaluation of the model is explained throughout the Jupyter Notebook 2-model-evaluation. An evaluation of the generated images from the system is pursued using the MNIST database and a classifier.

First we start by implementing the DCGAN (same process as in Part 1) but using the MNIST dataset. After the images are generated, the last batch is saved and cut in equal sizes to extract the fake digits.

At this point, the classifier is implemented. Again, the dataset used is the MNIST and the first step is to download the training and testing data and to create the data loaders.

The classifier is built up by a series of linear, ReLU activation functions and dropout. It contains three hidden layers and at the end the LogSoftmax activation function is used to gather the probabilities for all the classes (digits from 0 to 9).

The training loop does 15 epochs and it computes training loss, testing loss and accuracy periodically.

Once the classifier is fit some random tests are performed and the corresponding predictions are correct.

The last step is to actually evaluate the fake images generated by the DCGAN system. For this, a function called predict is implemented. This function takes the path of an image as an input, preprocesses it, evaluates the model with that image and finally displays the image and the output probabilities.

3. Generation artwork

The generation of paintings that resemble real ones is done in the Jupyter Notebook 3-generate-art-work. This process corresponds to the same done in part 1, with the difference that this time the dataset is of abstract paintings from the Wikiart.org.

Refinement

There were two parts of the project were continuous improvement needed to be done in order to achieve the desired performance.

Neural Network Classifier

The first neural network that was implemented was a simple network with one hidden layer, no dropout and with five epochs. The resulting accuracy wasn't reaching the expected percentage (which was to be higher than 90%).

To improve this, two more linear layers were added as well as a dropout. Also, the number of epochs was increased to fifteen. With these changes, the resulting accuracy was already above 90%.

Generation of artwork

During the generation of artwork several problems were encountered. For instance, in the chosen dataset, the images were too big to be processed by the DCGAN. For this reason, only images smaller than 50KB were selected for the training processes. This decreased significantly the original size of the dataset, resulting the final one in 4,000 images.

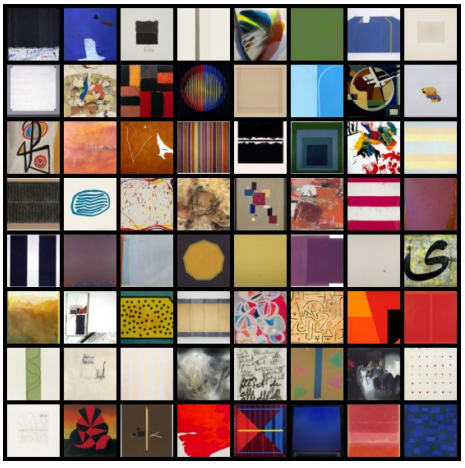
CelebA	MNIST	Artwork
200,000	60,000	4,000

Fig. 7 Comparison of the sizes of the different datasets that were used to train the DCGAN.

As we can see in Fig. 7, because the training set was smaller than the rest of previous used ones (CelebA and MNIST) the resulting images and the performance of the system was poor.

In the first iteration, performed with 5 epochs, the resulting images were noise and the losses were very high. (See Fig. 8)

Real images



Fake images

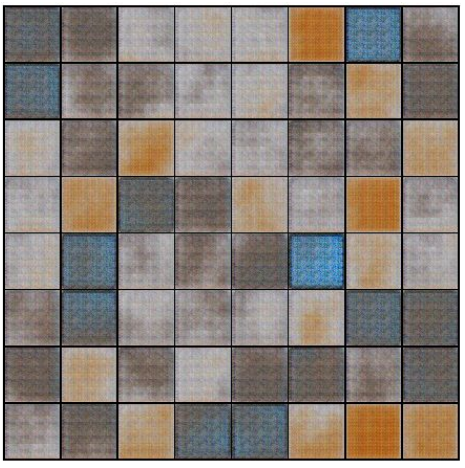


Fig. 8 From left to right, a few real images from the training set vs the generated images. The training set was 4,000 images and the number of epochs was 5.

In the second iteration, the number of epochs was increased to 100. With this change, there was already a significant improvement in the generated images plus the losses of both the discriminator and the generator were better as well.

Real images

Fake images

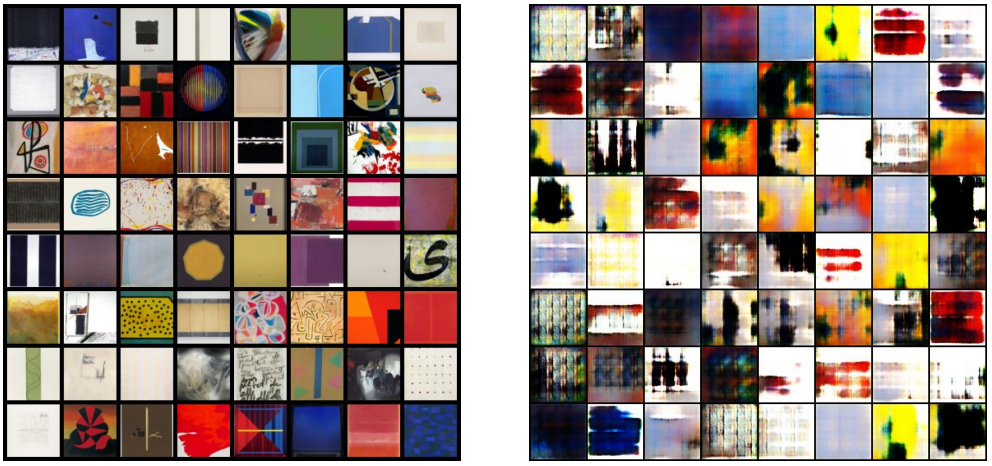


Fig. 9 From left to right, a few real images from the training set vs the generated images. The training set was 4,000 images and the number of epochs was 100.

In the third iteration, more images of paintings were included (approximately 2,000) in the dataset and the number of epochs was increased to 200.

Real images

Fake images

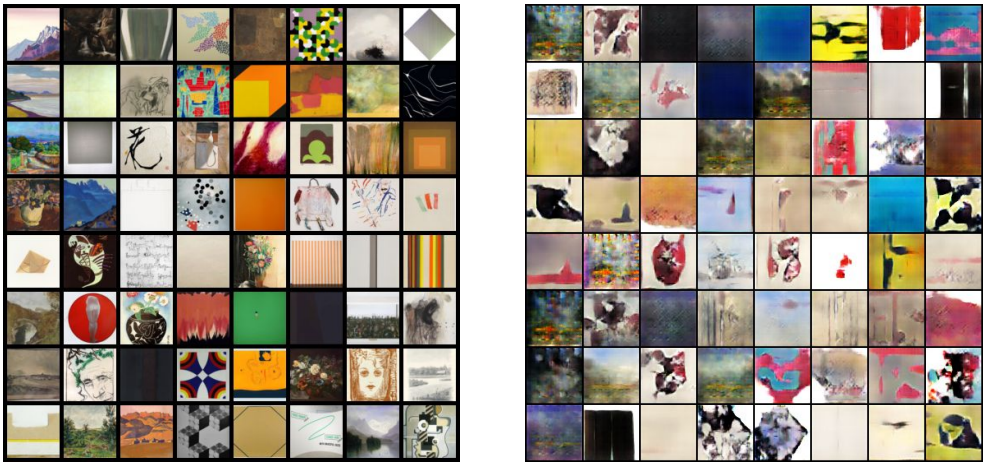


Fig. 10 From left to right, a few real images from the training set vs the generated images. The training set was 6,000 images and the number of epochs was 100.

IV. Results

Model Evaluation and Validation

The model selected for this task is a class of convolutional neural network called deep convolutional generative adversarial network (DCGAN). DCGAN was chosen as they have proven to be a strong candidate for unsupervised learning. This can be seen by:

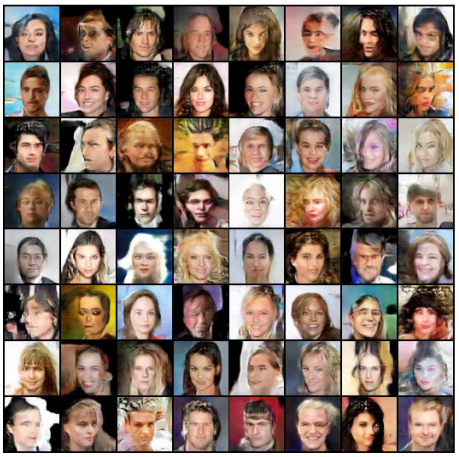
- The capacity of the system to reproduce different kinds of images, from human faces to handwritten digits or even paintings. We have seen this in the study performed in the three Jupyter Notebooks, each of them using a different dataset.
- The fact that a classifier trained on the MNIST dataset was able to correctly predict the synthetic handwritten digits generated by our DCGAN (also trained with the MNIST database).
- From the benchmark model comparison, we can also conclude that the results that we obtained have reached the expected level.

Justification

Let's compare some of the results obtained with the benchmark model as well as with the original paper of DCGAN.

In Fig.11 we can see the output of the DCGAN trained in the benchmark model (left) as well as ours (right). It's clear to see that the images obtained represent human faces. Furthermore, the quality of the images obtained from both systems is similar. In both cases, some images look more real and others look blurry in some parts.

Fake images from the benchmark model



Fake images from our DCGAN

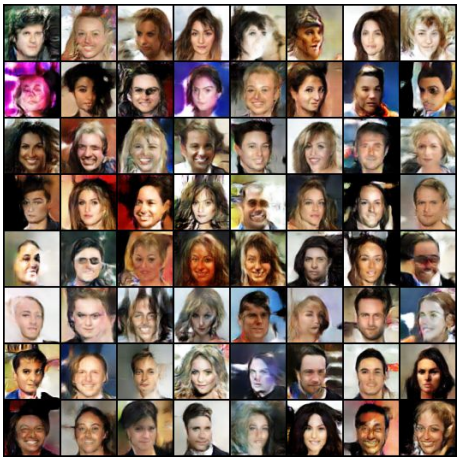
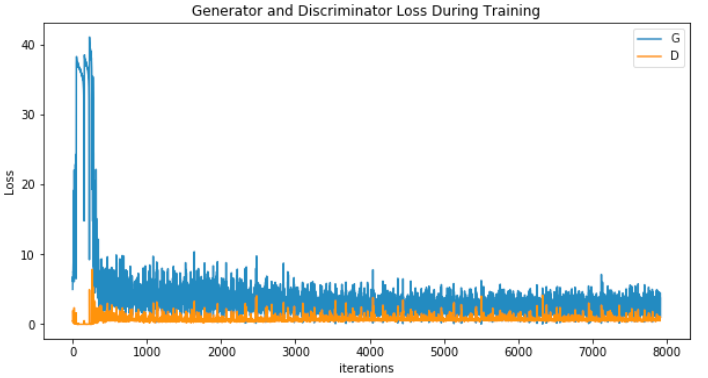


Fig. 11 From left to right, the fake images generated by the benchmark model vs the fake images generated in this project using the CelebA dataset. Source of the left image: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Next, let's evaluate the generator and discriminator losses during training (see Fig. 12). The first image (top) corresponds to the losses of the benchmark model, the second one (bottom) corresponds to ours.

We can observe that although at the beginning the losses behave very differently, at the end they are practically the same. The fact that they are so different at the beginning could be explained by the fact that the weights of the models are randomly initialized following a normal distribution.



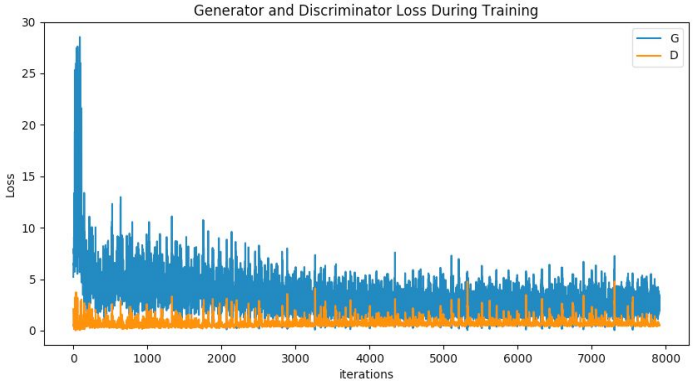


Fig. 12 From top to bottom, the Generator and Discriminator Loss During training by the benchmark model vs the Generator and Discriminator Loss During training in this project. Source of the first image: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

With these two comparisons (Fig. 11 and Fig. 12) with the benchmark model we can conclude that we reached the expected level that we were aiming for.

Let's take a look now at a comparison of the handwritten generation. In Fig. 13 we can see some fake handwritten digits generated by the authors of the DCGAN (left) and some fake handwritten digits generated by us (right) during this project. Although the digits from the DCGAN authors look somehow more clear, the handwritten digits that we obtained can be interpreted. This also proves that our model design is also capable of producing the expected results.

Fake images from the authors of DCGAN

Fake images generated with our DCGAN

Fig. 13 From left to right, the fake images generated by the authors of the DCGAN paper vs the fake images generated in this project using the MNIST database. Source of the left image: https://arxiv.org/pdf/1511.06434.pdf

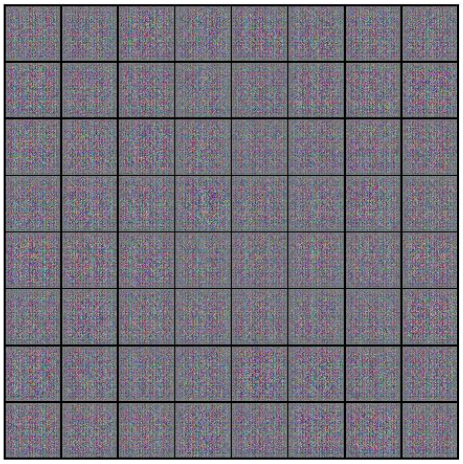
V. Conclusion

Free-Form Visualization

To appreciate what is actually happening during the process of generating the synthetic images and how our model is actually learning, let's see a few videos with the different training sets. See Fig. 14 and Fig. 15.

As we can see, the first batch of images is always noise, which corresponds to the latent space vector that we input into the generator. From that point, the images get gradually better while the generator learns the features.

First generated batch



Last generated batch

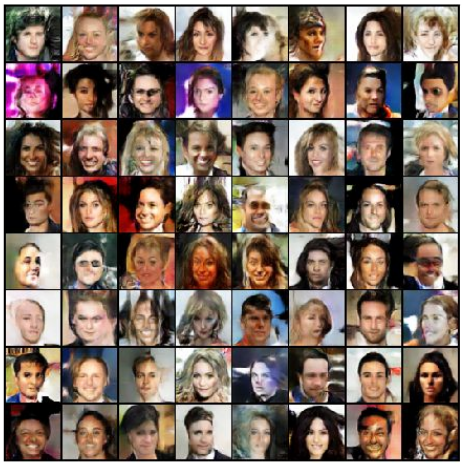
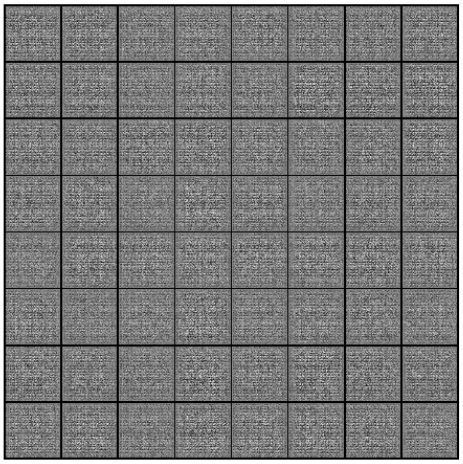


Fig. 14 From left to right the first and last batch image generated during the training of the Generator with the CelebA dataset. Link to the animation with the full progression.

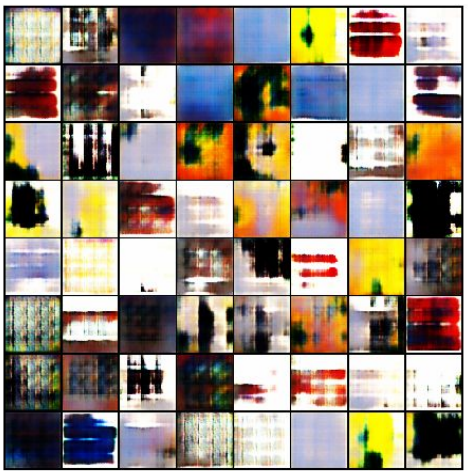
First generated batch



Last generated batch

Fig. 15 From left to right the first and last batch image generated during the training of the Generator with the MNIST database. Link to the animation with the full progression.

First generated batch



Last generated batch

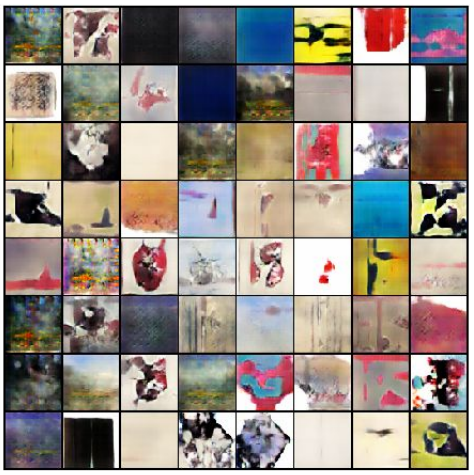


Fig. 16 From left to right the first and last batch image generated during the training of the Generator with the paintings from the Wikiart.org dataset. Notice that this time the initial weights of the model were the ones computed during the previous training process. Link to the animation with the full progression.

Reflection

The process followed in order to accomplish this project was:

1. Research the state of the art regarding GANs.
2. Research the available datasets that could potentially fit the needs of this project.
3. Implement the DCGAN system introduced here using the CelebA dataset.
4. Implement again the DCGAN system this time trained with the MNIST dataset and save the generated images.
5. Implement a classifier trained with the MNIST dataset to evaluate the fake images produced by our DCGAN.
6. Implement the DCGAN system to produce images of art work. (This step needed several attempts in order to reach the expected level).

The most interesting part of this project was to see how the model could actually learn the features in the images from the different datasets and then see the process of generating then. One thing to notice is that DCGAN could learn any type of dataset.

The most difficult thing was to actually having the quantity of images as well as the resolution needed to perform the training process. One measure that was taken to overcome the lack of images was to increase the number of epochs.

Overall, the solution obtained fits the expectation I had for this project.

Improvement

Better results could be obtained by having a much richer dataset (abundant images with the proper resolution needed to train the model), as in the case of abstract art work only a small dataset was found.

Another interesting thing that could potentially be explored is the capabilities of interpolation of the system.

Last but not least, I would also like to explore the possibilities of video generation using this kind of technology.

VI. References

[1] https://en.wikipedia.org/wiki/Generative_adversarial_network

[2] https://towardsdatascience.com/gangogh-creating-art-with-gans-8d087d8f74a1

[3] https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html