
Deep Convolutional Generative Adversarial Networks report

Paolo Messina
Student Number: 18037634
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
Paolo2.Messina@live.uwe.ac.uk

Introduction

This second creative task assigned to us consist of working on **Generative Adversarial Networks (GAN)** for create fake images. **GAN** is a framework for estimating generative models through an antagonistic process, which involves the simultaneous training of a **Generator** and a **Discriminator**.

The first appearance was in 2014, **Ian J. Goodfellow** et al. presented an academic article that introduced a new framework for the estimation of generative models through an adversary, or antagonistic process, using two networks: one generative, the other discriminatory (Ian J. Goodfellow, 2014).

To work on **GAN**, was decided for this second creative task, to work on the dataset already used by the first task between planets and galaxies. Still, this time the goal is to generate a batch of fake images of astronomical objects (Stars, planets, satellites, galaxies).

For this specific project has used the architecture typically known as **Deep Convolutional Generative Adversarial Networks (DCGAN)**, which is one of the most popular architectures for the generative network.

Background & Research

The first step for work on **GANs** is to understand what they are and how they work, to generate fake images from this framework.

Yoshua Bengio, a Canadian computer scientist who has studied deep learning, promises to uncover rich hierarchical patterns that represent probability distributions on data such as images, audio conversations, and text (Yoshua Bengio, 2014).

The goal of a **Deep Learning** project is to develop a model that represents a certain reality: an image, a text, etc.

Therefore, it is possible to establish mathematical functions and successive iterations and aid an algorithm, obtaining the parameters that define the function from the data. Consequently, it is possible to define the model.

GANs are a creative system, and they allow to generate data, mainly images (therefore videos) from scratch and can be used to support **Reinforcement Learning systems**.

The **Generative Adversarial Nets** framework includes two **Deep Networks**: a generator, **Generator G**, and a discriminator, **Discriminator D**.

GANs use the **Generative** model, indicated as **Generator** or **G**, to capture the original distribution. In contrast, the **Discriminatory** model, called **Discriminator** or **D**, estimates the probability that data comes from the training dataset instead of **G**.

The training procedure for **G**, the goal function, expects to maximize the probability that **D** is wrong.

As an input to the model, it needs to generate random numbers. However, it is not a simple thing as it is impossible to generate truly random numbers.

However, defining the algorithms that generate number sequences is possible to notice that whose properties are like those generated randomly.

By increasing the complexity, it is possible to say that using a pseudorandom number generator, it is possible to create sequences of numbers that follow approximately a uniform distribution between 0 and 1.

By having a generic input **Z**, it is possible to use the **Generator G** model to create an image **X**. Conceptually, **Z** represents the latent characteristics of the generated image.

As previously mentioned, was chosen **DCGAN** architecture for its polarity and its successful network design for **GAN**. It replaces all max pooling with convolutional stride and uses transposed convolution for up sampling.

Moreover, it eliminates connected layers and uses batch normalization except the output layer for the generator and the input layer of the discriminator.

The **DCGAN** Through multiple transposed convolutions, up samples from \mathbf{z} to generate \mathbf{x} . It is possible to think structure as a **deep learning classifier** with the opposite functioning. Then **Discriminator** guides the **Generator** on the image to be created (Alec Radford, 2016).

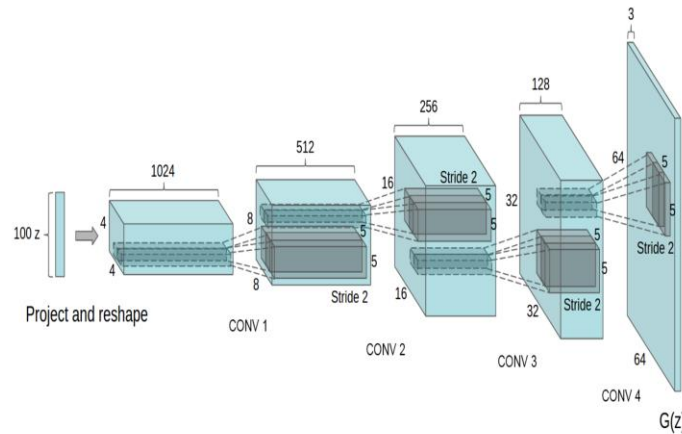


Figure 1 DCGAN generator.

Trained with authentic images and generated one, **Generative Adversarial Nets** can improve the **Discriminator** to recognize actual characteristics. The same **Discriminator** then provides feedback to the **Generator** to create images.

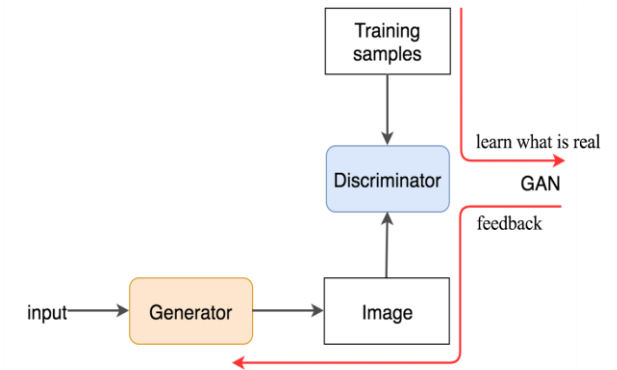


Figure 2 Model GAN.

The **Discriminator** examines the real photos (from the training dataset) and generates images separately.

It produces a probability $\mathbf{D}(\mathbf{x})$ that the input image is authentic or fictitious.

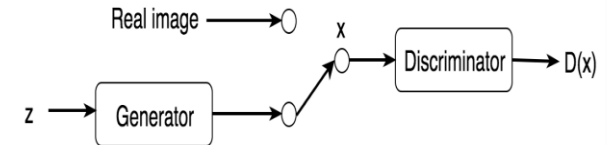


Figure 3 Model output GAN generated images.

The **Discriminator** is like a deep network binary classifier: 1, real image; 0, generated image.

At the same time, the **Generator** must create images that reproduce the real ones and are therefore interpreted by the **Discriminator** as $D(x) = 1$.

At the end of the training process, the **GAN** models converge and produce images that look real (Jun-yan Zhu, 2020).

Implementations

After understanding what **GANs** are and how they work, it is possible to move on to **DCGAN** implementation on the project.

As for the first classification task, was used **Google Colab** to work on the **DCGAN** framework. The choice falls again on **Google Colab** because it is a simple tool to work on projects with **PyTorch**.

It is effortless to connect the notebook on **Google Drive** or **GitHub**, and in addition, the notebook, can be downloaded to run locally on other software.

Regarding the choice of the dataset, it was decided to take the dataset used in the first creative task and at the same time add new images to generate the image on the framework.

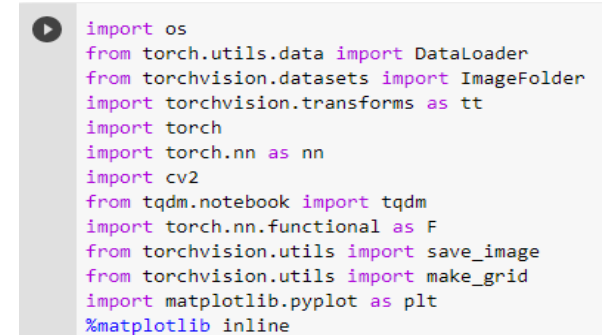
The Second task was to create fake astronomical objects, such as Stars, planets, satellites, and galaxies and for the has been repurposed to use the same samples from the first task.

Another tool used again was **Image Downloader**, which allows to download images in any format easily. However, on the first task, there were some issues related to this extension for chrome. For the collection of the dataset for the second task this tool, in the end was proved to be quite a helpful tool for the collection of new datasets.

The Blog followed for creating the **GAN** creative task was published by **Shubham Gupta**, which explains how to get started with **GANs** Using **PyTorch** (Shubham Gupta, 2020).

The dataset collected includes 200 images in total between random space objects. He uses GAN to generate images of cats' faces, which consists of more than 15,700 cats images. Having a large number of images makes it possible to obtain a better quality of the generated images. However, 200 is enough to obtain a discreet quality image.

The first step consists of importing all the required libraries.



```
import os
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import torchvision.transforms as tt
import torch
import torch.nn as nn
import cv2
from tqdm.notebook import tqdm
import torch.nn.functional as F
from torchvision.utils import save_image
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
%matplotlib inline
```

Figure 4 Screenshot of the required libraries.

Since for this task it is used Google Colab, the dataset uploaded to google drive was imported, unzipped, and used.

```
[ ] from google.colab import drive
    drive.mount('/content/gdrive')
```

```
[ ] !unzip '/content/gdrive/MyDrive/Star&Galaxies2.zip' > /dev/null
```

Figure 5 Screenshot of the imported input.

Create a data loader for loading data into batches from google drive. The next step was to resize and crop the images to 64x64 px and normalize the pixel values with a mean and standard deviation of 0.5. This will ensure pixel values are within the range (-1, 1), which is more convenient for discriminator training.

```
[ ] DATA_DIR = 'Star&Galaxies2'
```

```
[ ] image_size = 64
    batch_size = 10
    stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
```

```
▶ train_ds = ImageFolder(DATA_DIR, transform=tt.Compose([
    tt.Resize(image_size),
    tt.CenterCrop(image_size),
    tt.ToTensor(),
    tt.Normalize(*stats)]))
```

```
[ ] train_dl = DataLoader(train_ds, batch_size, shuffle=True, num_workers=3, pin_memory=True)
```

Figure 6 Screenshot of data loader settings.

Discriminator

The Discriminator takes an image as its input and attempts to classify it as “generated”. This will take input as (3 x 64 x 64) tensor and convert it into (1, 1, 1) tensor.

```
▶ discriminator = nn.Sequential(
    # in: 3 x 64 x 64

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 64 x 32 x 32

    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 16 x 16

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 8 x 8

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 4 x 4

    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid())
```

Figure 7 Screenshot of Discriminator.

Generator

The input to the Generator is a vector or matrix of random numbers used as a seed for image generation. The Generator can convert a shape tensor (128, 1, 1) to a (3 x 28 x 28) shape tensor image.

```
[ ] generator = nn.Sequential(
    # in: latent_size x 1 x 1

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1, padding=0, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    # out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    # out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh()
    # out: 3 x 64 x 64
)
```

Figure 8 Screenshot of Generator.

The next step is to build a training function for Discriminator and Generator. This function will use to train both neural networks.

This is the training function for the Discriminator.

```
def train_discriminator(real_images, opt_d):
    # Clear discriminator gradients
    opt_d.zero_grad()

    # Pass real images through discriminator
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)
    real_score = torch.mean(real_preds).item()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)
    fake_score = torch.mean(fake_preds).item()

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item(), real_score, fake_score
```

Figure 9 Screenshot train Discriminator.

This is the training function for the Generator.

```
def train_generator(opt_g):
    # Clear generator gradients
    opt_g.zero_grad()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Try to fool the discriminator
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)
    loss = F.binary_cross_entropy(preds, targets)

    # Update generator weights
    loss.backward()
    opt_g.step()

    return loss.item()
```

Figure 10 Screenshot train Generator.

However, it still needs to build a training function for both Discriminator and Generator.

```
def fit(epochs, lr, start_idx=1):
    torch.cuda.empty_cache()

    # Losses & scores
    losses_g = []
    losses_d = []
    real_scores = []
    fake_scores = []

    # Create optimizers
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dl):
            # Train discriminator
            loss_d, real_score, fake_score = train_discriminator(real_images, opt_d)
            # Train generator
            loss_g = train_generator(opt_g)

            # Record losses & scores
            losses_g.append(loss_g)
            losses_d.append(loss_d)
            real_scores.append(real_score)
            fake_scores.append(fake_score)

        # Log losses & scores (last batch)
        print("Epoch [{} / {}], loss_g: {:.4f}, loss_d: {:.4f}, real_score: {:.4f}, fake_score: {:.4f}".format(
            epoch+1, epochs, loss_g, loss_d, real_score, fake_score))

        # Save generated images
        save_samples(epoch+start_idx, fixed_latent, show=False)

    return losses_g, losses_d, real_scores, fake_scores
```

Figure 11 Screenshot training function **D**, and **G**.

To conclude it was assigned the learning rate(**lr**) = 0.002, and the number of epochs = 200.

Conclusion

After training the model for 200 epochs, these are the generated Images.

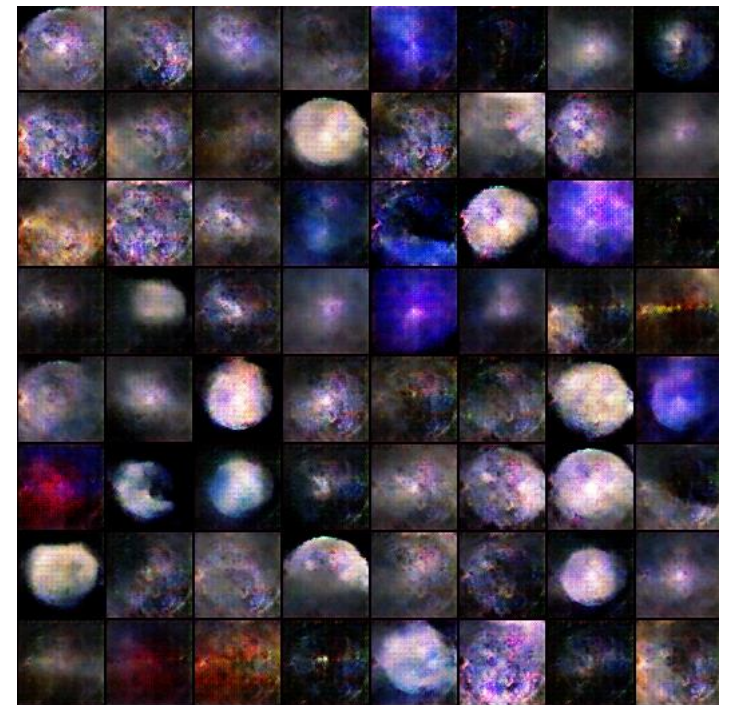


Figure 12 Screenshot of results.

It is possible to notice that the fake images are not exactly like the original one. However, given the complexity of the images, it is possible to agree that the framework has generated images of good quality.

Moreover, there are more results need to be analyzed which the diagrams displayed below.

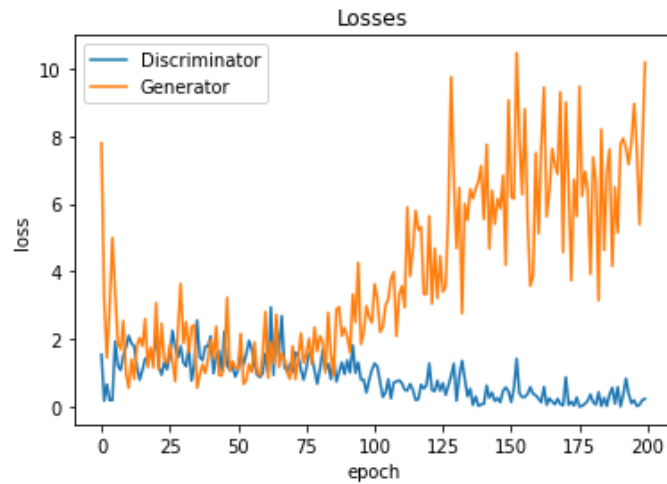


Figure 13 Screenshot of loss diagram.

It is possible to see that the Discriminator is doing much better than the Generator over the 200 epochs. Furthermore, the diagram display that the best moment where the Discriminator and the Generator were similar, was around 70/90 epochs.

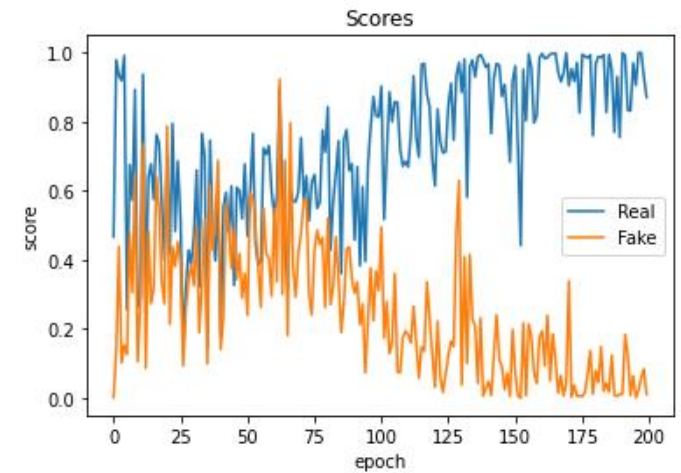


Figure 14 Screenshot of scores diagram.

The scores diagram helps to visualize better the results of the Loss diagram. The scores for Discriminator and the Generator were similar around 70/90 epochs as well.

During the testing phase, was tried to lower the **lr** = 0.0002. Theoretically, lowering the **lr** rate should give the Discriminator and Generator more time to training. However, it seems that whit this **lr**, the Generator do much worse than when the **lr** is set at 0.002.

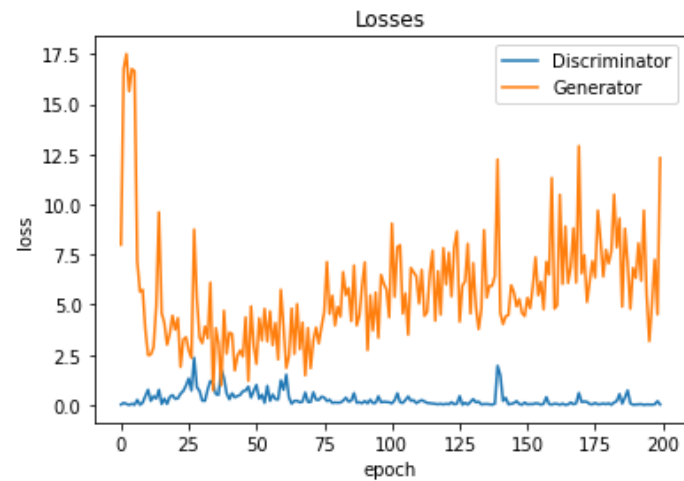


Figure 15 Screenshot los diagram at 0.0002 **lr**.

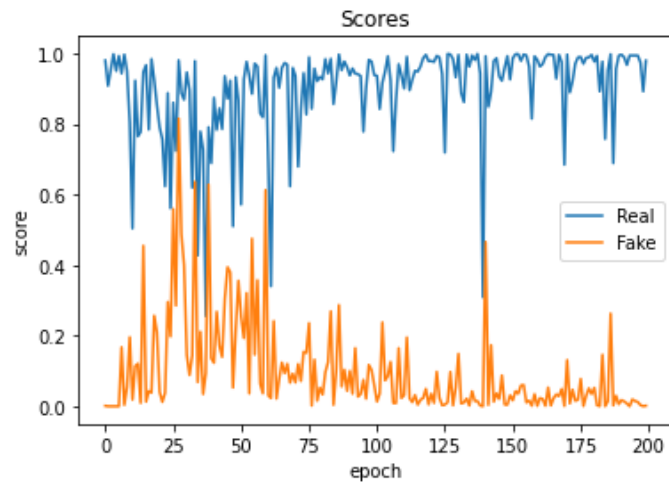


Figure 16 Screenshot scores diagram at 0.0002 **lr**.

However, it seems that it does not impact the final images created by the Generator.

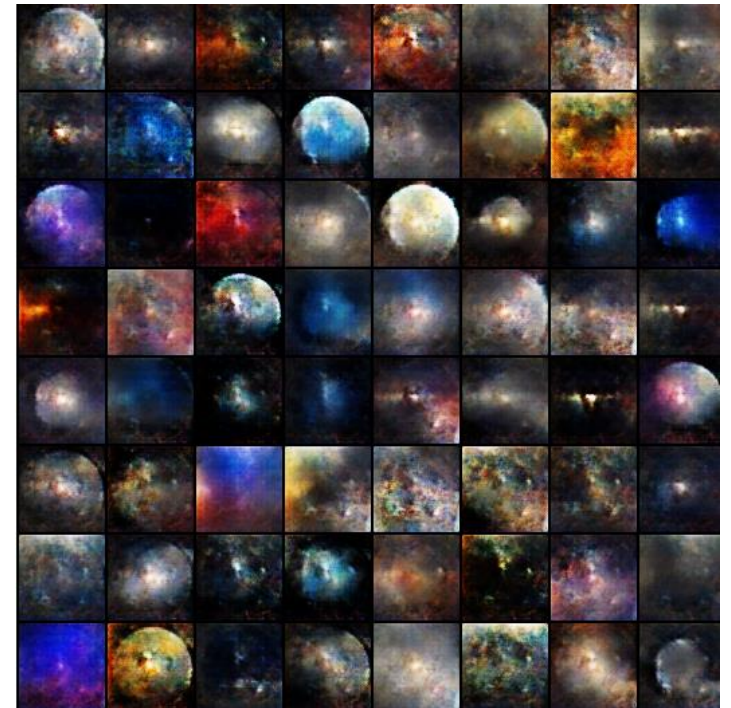


Figure 17 Screenshot of results at 0.0002 **lr**.

A hypothesis is that the Generator does better results when it has a high **lr** is because the Discriminator has less time to train the system. Consequently, the Generator does have better results at the end.

References.

- Ian J. Goodfellow, Jean Pouget-abadie*, Mehdi Mirza, Bing Xu, David Warde-farley, Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡, (2014) Generative Adversarial Nets. *Universite De Montreal* [online]. [Accessed 29 April 2021]. <https://arxiv.org/pdf/1406.2661.pdf>
- Alec Radford, Luke Metz & Soumith Chintala, (2016) Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *Conference Paper at Iclr 2016* [online]. [Accessed 29 April 2021]. <https://arxiv.org/pdf/1511.06434.pdf>
- Jun-yan Zhu* Taesung Park* Phillip Isola Alexei A. Efros, (2020) Unpaired Image-to-image Translation Using Cycle-consistent Adversarial Networks. *Berkeley Ai Research (Bair) Laboratory, Uc Berkeley* [online]. [Accessed 29 April 2021]. <https://arxiv.org/pdf/1703.10593.pdf>
- Yoshua Bengio, (2014) Learning Deep Architectures For Ai. *Universite De Montreal* [online]. [Accessed 29 April 2021]. <https://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf>
- Shubham Gupta, (2020) Getting Started with Gans Using Pytorch. *Towards Data Science* [online]. [Accessed 29 April 2021]. <https://towardsdatascience.com/getting-started-with-gans-using-pytorch-78e7c22a14a5>