Machine Learning Engineer Nanodegree Capstone Project – Image Colorization

Maxime GENDRE

Udacity

Abstract- In computer vision, there are many typologies of problems to resolve, like object detection, image segmentation, classification, image generation. A very exciting thematic is about how to get a good quality picture, with a noisy / bad quality image, or colorize a black and white / old picture.

I. Project Overview

For a long time, scientists, developers and researchers have been working on : How to generate colors in a black and white image. Nowadays, we have a lot of papers which is the result of several years of their work, to guide and allow people to do this task more easily, and efficiently. Moreover, a lot of open data are available today, especially in computer vision (MS Coco, Open Image Dataset V6, …)

Machine Learning tasks in image processing generally requires a large amount of data to achieve a good result in a specific task, due to the large number of factors that define a "good" and "usable" image.

Color photography is photography that uses media capable of capturing and reproducing colors. By contrast, black and white (monochrome) photography records only a single channel of luminance (brightness) and uses media capable only of showing shades of gray. (source: Wikipedia)

The first method to colorize a photography in 3 channels has been released in a 1855 paper, by Scottish physicist James Clerk Maxwell.

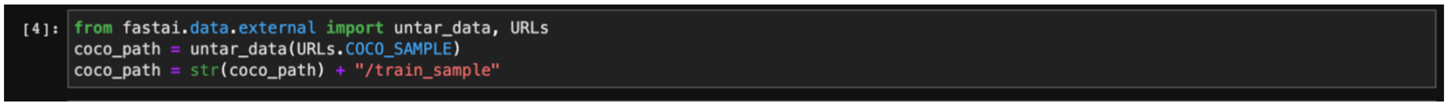
As you can see, since a long time, humans are working on this problem.

After some research, I found these papers:
- Image-to-Image Translation with Conditional Adversarial Networks (Authors: Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros)
- Colorful Image Colorization (Authors: Richard Zhang, Phillip Isola, Alexei A. Efros)

In this project, I will show how to colorize images with GANs.

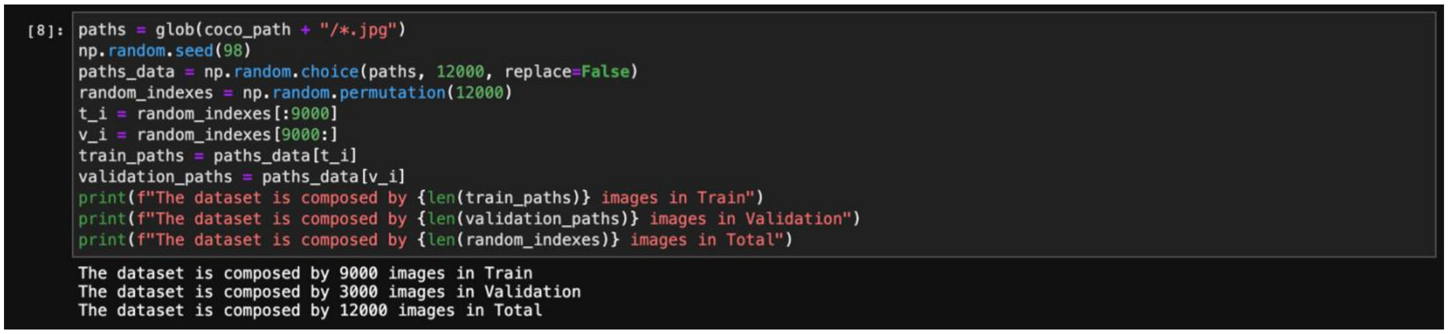
The dataset which I will use will be the MS COCO 2017 dataset to obtain a great variety of pictures. "COCO is a large-scale object detection, segmentation, and captioning dataset.". To download data, we can use **fastai** python package. To feed my Train set and Validation set, I will only take 12000 images (9000 Train samples, 3000 Validation samples). We can find several classes in COCO Dataset, but I won't target to do object detection or classification, so I'll not take consideration about classes and I will just use random images. I need a great diversity of scenes, landscapes, brightness images.

Here you can see the code used to get the **COCO Dataset** in the .***cache* folder** of my python environment.



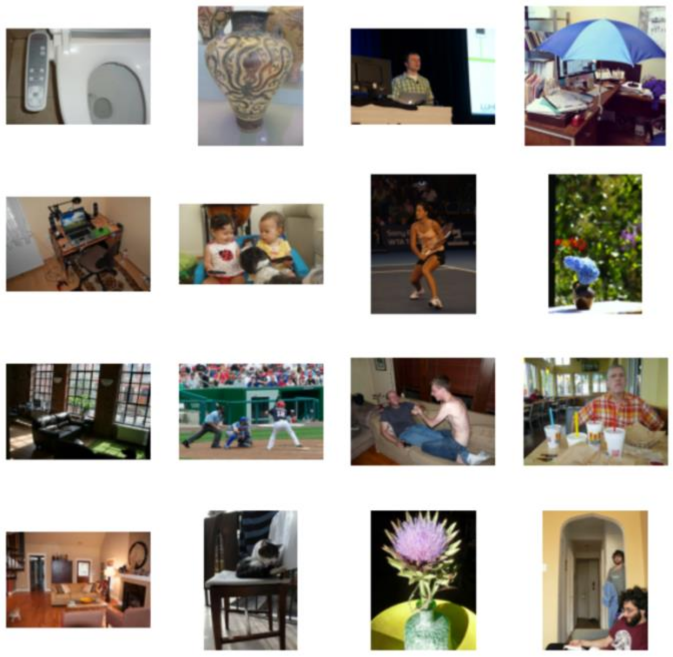
```
[4]: from fastai.data.external import untar_data, URLs
     coco_path = untar_data(URLs.COCO_SAMPLE)
     coco_path = str(coco_path) + "/train_sample"
```

As I said, I choose random images to feed my Train / Validation set.



```
[8]: paths = glob(coco_path + "/*.jpg")
     np.random.seed(98)
     paths_data = np.random.choice(paths, 12000, replace=False)
     random_indexes = np.random.permutation(12000)
     t_i = random_indexes[:9000]
     v_i = random_indexes[9000:]
     train_paths = paths_data[t_i]
     validation_paths = paths_data[v_i]
     print(f"The dataset is composed by {len(train_paths)} images in Train")
     print(f"The dataset is composed by {len(validation_paths)} images in Validation")
     print(f"The dataset is composed by {len(random_indexes)} images in Total")

     The dataset is composed by 9000 images in Train
     The dataset is composed by 3000 images in Validation
     The dataset is composed by 12000 images in Total
```



A short extract of my Train set – Images from COCO Dataset

II. Problem Statement

A first method to realize our colorization is to use $L*a*b*$.
$L*a*b*$ is a color space, defined by the CIE (*Commission Internationale de l'Eclairage*)
Respectively, $L*$ indicates lightness, $a*$ is the red/green coordinate, and b* the yellow/blue.
Secondly, we have RGB, which is commonly used today to attribute colors to images. R is Red, G is Green, and B is Blue. On an image, we will have for each pixel, a value for those three variables.
There are more color space than those two, but I won't describe them in this project.

To train a model for colorization, we have X, which is our train data, and y, our target. In our case, we have a grayscale image as input, which don't have any color space, and as output, our colors.
The problem is to determine with a grayscale image / value, a corresponding color.

We can use many methods to do it, for example, let's take a look to the $L*a*b*$. We will assume that X is our lightness factor known as $L*$ and y our target is $a*b*$ which describe colors. Once our model predicts colors, we just have to concatenate predictions with lightness, and here we are. Our black and white image has been colorized.
A second hypothesis is to use RGB color space. Here again, we will try to predict our 'R' 'G' 'B' values, from our input, which is a grayscale image.

I choose to use the $L * a * b *$ color space to colorize pictures. With GANs, I'm going to generate "Fake Images" with a Generator, and a Discriminator will be trained to recognize "Fake" and "Real" images.

There are many ways to complete our job with machine learning. Some papers do classification models, others use regression approach. In my case, I'm going to use **Conditional Adversarial Networks**, with **pix2pix** (Image-to-Image Translation with Conditional Adversarial Networks) paper. Two losses are used in this paper: **L1 loss**, to do regression, and an **adversarial loss** (GAN). The proposed solution is to use conditional **GAN** for our task to colorize images. A GAN has a generator, and a discriminator model, to learn how to solve a problem together. **Generator** will take black and white images ($L *$) and produces 2 channels images ($a * b *$). **Discriminators** aim to determine if it's a fake, or a real image, with all previous channels concatenated.

In this paper , the loss of conditional GAN is a function which works like this: $\boldsymbol{x}$ is our grayscale, $\boldsymbol{y}$ our 2-channels output that the generator produces, $\boldsymbol{z}$ as input noise for the generator, $\boldsymbol{G}$ the generator model and $\boldsymbol{D}$ the discriminator. This loss function will help to produce "real" colorful images.

$$\begin{aligned}\mathcal{L}_{cGAN}(G, D) =& \mathbb{E}_{x,y}[\log D(x, y)]+\\ & \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z))],\end{aligned}$$

Image from paper – Objective of a conditional GAN can be expressed like this

Another loss that we will use is the **L1 loss**, known as **mean absolute error**. In the paper, they are combining the previous loss we saw with the **L1 loss** to assure that the model's color choices are the best.

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1].$$

Image from paper – L1 loss

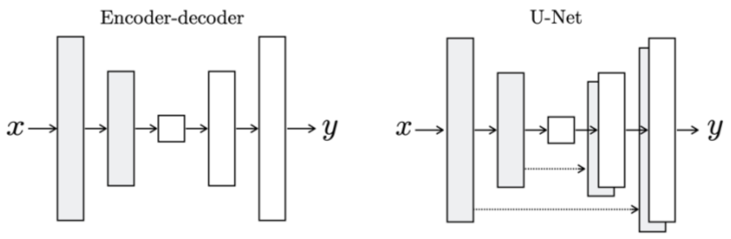
The final objective with the combination of those two functions is:

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G).$$

λ is a coefficient to balance the contribution of the two losses to the final loss.

You can find the GAN Loss Calculation in my notebook, the class name is **GANLoss**.

Below, this is a representation of our U-Net, which will be our Generator layers architecture (*U-Net*).



"Two choices for the architecture of generator. "U-Net" is an encoder-decoder with skip connections between mirrored layers in the encoder and decoder stacks." - paper

III. Metrics

Regardless of the metric used, it is difficult to associate a GAN result with a metric since the quality of the result is primarily based on human perception, the result is more of a subjective assessment. In my implementation, I will try to take a sample of varied SSIM, to see and demonstrate that a low SSIM can still produce a realistic result in order to have a better understanding of the model result.

The metric I will use is the **Structural Similarity Index** (**SSIM**).
SSIM is a metric used to measure the similarity between two given images. It's a perception-based model that measures changes in the structural information of images as a good approximation of perceived image distortion. The abstract of the paper of **SSIM** show globally the background of SSIM.

"Objective methods for assessing perceptual image quality traditionally attempted to quantify the visibility of errors (differences) between a distorted image and a reference image using a variety of known properties of the human visual system. Under the assumption that human visual perception is highly adapted for extracting structural information from a scene, we introduce an alternative complementary framework for quality assessment based on the degradation of structural information."

Image Quality Assessment: From Error Visibility to Structural Similarity paper - Zhou Wang

There are 3 key features that the SSIM use in an image: **Luminance**, **Contrast**, and **Structure**.

$$\text{SSIM}(\mathbf{x},\mathbf{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{\left(\mu_x^2 + \mu_y^2 + C_1\right)\left(\sigma_x^2 + \sigma_y^2 + C_2\right)}.$$

SSIM Index Formula - Image Quality Assessment: From Error Visibility to Structural Similarity paper

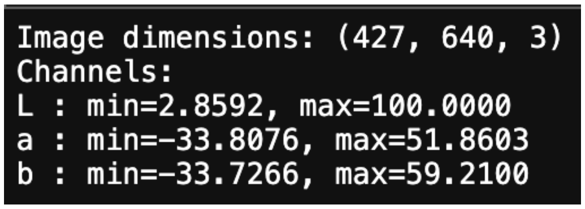
SSIM is based on the comparison between luminance l(x,y), contrast c(x,y), and structure s(x,y), where x and y are the original image and the compared image α, β, and γ are weights for each of those characteristics.
The lowest value for **SSIM** is **-1**, and the highest is **1**. (1 means a perfect similarity)
This measure will be used to compare two images and calculate a score for similarity between them with window sliding (local features sensibility).

To calculate SSIM, I'm using **scikit-image** library, which have a function called **structural_similarity** in the **metrics** module.

IV. Data Exploration

Remember that we have: 12000 images in total. 9000 in Train, 3000 in Validation.

To explore my data, I first looked at the $L * a * b *$ values for an image.

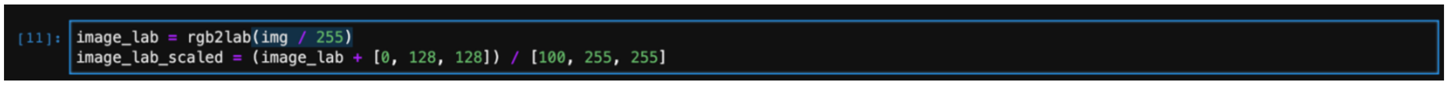


```
Image dimensions: (427, 640, 3)
Channels:
L : min=2.8592, max=100.0000
a : min=-33.8076, max=51.8603
b : min=-33.7266, max=59.2100
```

*Print summary method to show min / max values properties of a l*a*b image color space*

As we can observe, a & b values can be negative, and positive, whereas the L channel is between 0 and 100, and like **RGB**, we have a shape (**Height**, **Width**, **Channels)** → **427**px by **640**px with **3** channels in this example.

So the first way to get this format is to convert the RGB color space to Lab. **Scikit-image** library provide a **rgb2lab** method to accomplish this.

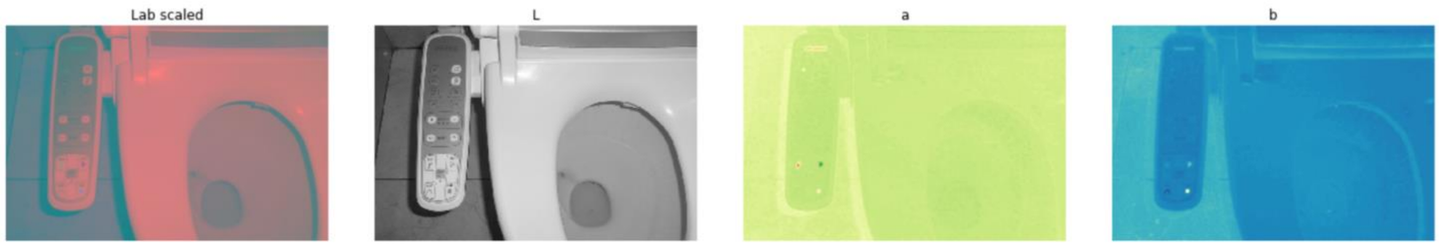


```
[11]: image_lab = rgb2lab(img / 255)
      image_lab_scaled = (image_lab + [0, 128, 128]) / [100, 255, 255]
```

For the **rgb2lab** input, I divide my Image (which is a Numpy array) by 255 (Max range of each RGB Channel).

In the second line, you can see a "scaled" version of a ***Lab*** image. The purpose of this operation is to make the image "visible" and "clean" for us. I wanted to see some ***Lab*** images with separated channels. To do it, after the scaled operation to get channels in true values to display it as **RGB** pictures, I isolate the different channels, and I display each of them.

V. Exploratory Visualization

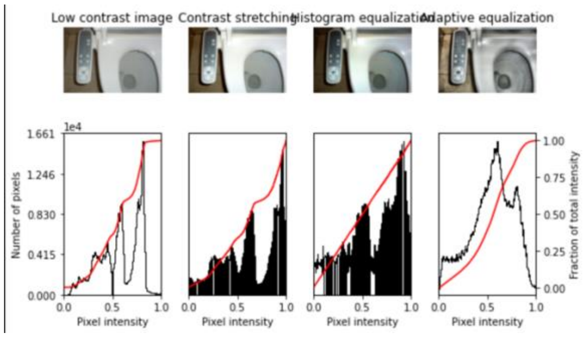


<u>image_lab_scaled[:,:,0]</u> → Channel 1 → (**L**)
<u>image_lab_scaled[:,:,1]</u> → Channel 2 → (**a**)
<u>image_lab_scaled[:,:,2]</u> → Channel 3 → (**b**)
imshow of the full lab scaled image, and each channel separately

We can clearly see the different inputs for our models. ***L*** is the input, which we want to transform to full ***Lab***.
a & b are our target, and we want our model to predict with L as input, the ***a*** and ***b*** channels to concatenate all of them and get a beautiful ***Lab*** picture.

In order to explore a pre-processing way, I have analyzed the pixel intensity (it mainly defines the ***L*** channel values), with 4 technics.

Set image to low contrast, do a contrast stretching, an histogram equalization, and an adaptative equalization.



As you can see, low contrast image, Histogram equalization and adaptative equalization can misrepresent the original image. The pixel intensity variates a lot depending on the technique used.

I choose to not use those techniques because I prefer to train my model on native images because it will be more generalist.

In my opinion, these techniques can still be a good improvement for data augmentation. We have to keep it in mind, even if I didn't do that.

VI. ALGORITHMS AND TECHNIQUES

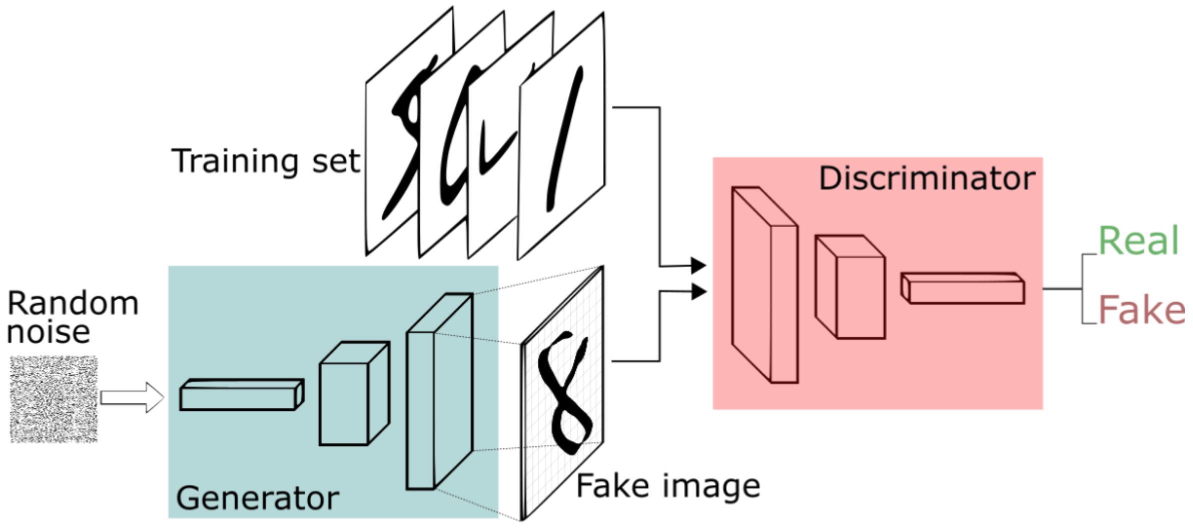
As I said earlier, I'm going to use GANs to solve this colorization problem (See **II. Problem Statement Section**).

In order to solve correctly the colorization task, I choose to train my Generator before the global training.

To train my model, I use a pretrained model for classification (***ResNet18***). This model has been trained on **ImageNet**. This pretrained model will be the backbone of my generator. Next, I pretrain my generator (**U-Net**) with this backbone on the task of colorization with **L1 Loss**.

To loads the pretrained weights of ***ResNet18***, I use **create_body** method from **fastai** package, and set the parameter "**cut**" to **-2** in order to remove the two last layers (Classification tasks layers), and then, with ***DynamicUnet***, we use this backbone to build a ***U-Net*** for our task, with our output dimension (2 for 2 channels, **a** and **b**), and as input size, we have our images resized in 256 by 256 pixels.

Next, when our generator is trained, I put my generator with my discriminator, and train the whole GAN.



Workflow of GANs

VII. BENCHMARK MODEL

It is difficult to find a true benchmark about the GANs because the metrics evaluating the quality of the result are mainly based on human's perception. However, GANs are in constant evolution, and there are some metrics which allow us to do a well-performing approach. Some papers / implementations are using the "Fréchet Inception Distance", "Inception Score", or even the "SSIM". We can also use the MSE, but it's the less used metric. In the following benchmark, you will see 2 examples of metrics benchmark.

SAGAN (paper):
- Fréchet Inception Distance: **18.65**
- Inception Score: **52.52**

(Do not take consideration of SAGAN because of the metrics used. I used SSIM)
cDCGANs (medium post):
- SSIM: **0.93**

My results:

- SSIM Average on Validation Set (3K Images): **0.87**

You can find in my notebook the result of the SSIM, and how I have calculated my SSIM average on validation set. (method ***get_perf_on_val***)

Get a SSIM average of **0.87** Is not bad, but we could get better results I think with:

- More epochs
- More data

In general, implementation of GANs are done on very large datasets (More than 100K train samples in general) and a lot of epochs.
In my case, 9K train samples with 70 epochs (+30 epochs to pretrain the generator).

VIII. Data Preprocessing

In order to load my data, and use it with **Pytorch**, I create a Dataset class named *ColorizationDataset* which inherit of Dataset class of **Pytorch**.

The Dataset class is where we have to define some preprocessing / data augmentation to fit from our native data, to our input format to feed our neural networks.

In this class, we can find a first parameter which define if we are using train samples, or validation samples. The difference is important because we don't want for example, to do data augmentation on the Validation Set.

1. __init__() method take 2 arguments. : a list of path to get the path for each images, and as I said, the type of split (train or validation).
   If the split parameter is "**train**", we are using transforms module from **Pytorch** to preprocess our images. We create a Compose element to describe what are we doing on our preprocessing. First, we resize input to a 256 by 256 resolution. Next, we do a Random Horizontal Flip. This little data augmentation will be useful to make the generator more generalist and avoid overfitting.
   If the split parameter is "**validation**", we just resize input to 256x256 (Minimal require to be a valid image as input).

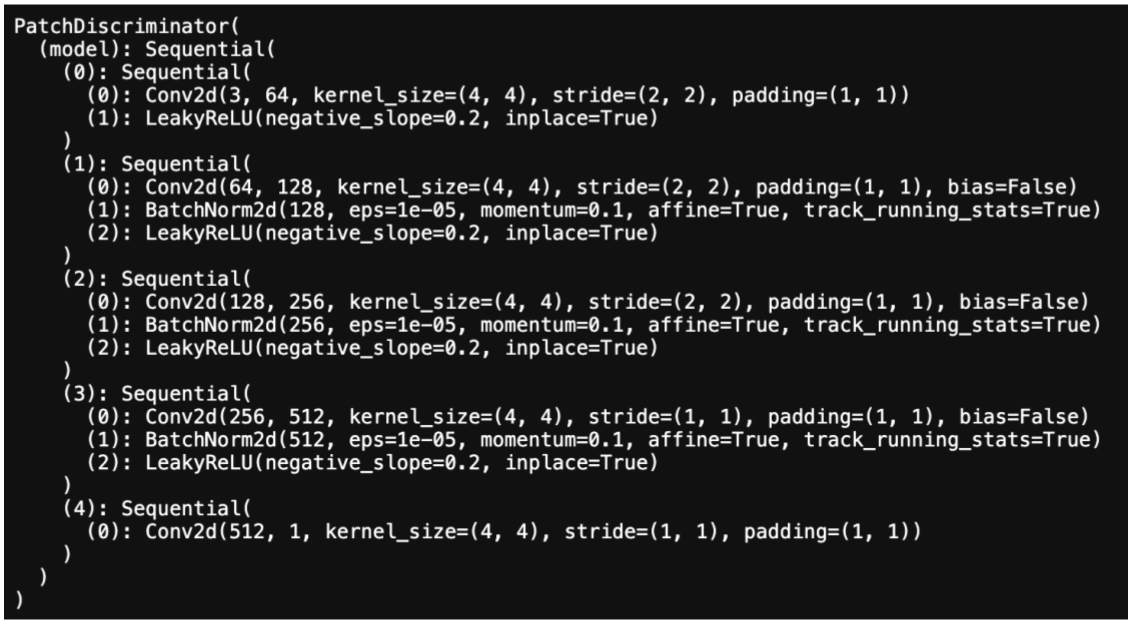
2. __getitem__() method take 1 argument : **idx**, the index of the image we want to get. This method is called when we try to get a data from the Dataset object. So, each time our **Dataloader** will get a data from the Dataset object, it will be triggered.
   First, we pick an image in the paths (list of path of images) with the idx param, and we open it, and to ensure that the image is **RGB**, we do ***.convert('RGB')*** on image opening.
   We apply transforms that are described in the Compose element in the __init__ of our Dataset Class and transform our image to a Numpy array.
   Finally, we convert ***RGB*** image to ***Lab*** and separate ***L*** and ***ab*** in order to return them separately, and before return, we convert our Numpy array to Tensor.

Next, we create our **Dataloaders** for validation and train, using our dataset objects, and we define a **batch_size** to **16**.

IX. IMPLEMENTATION

Implementations steps to train are the following.

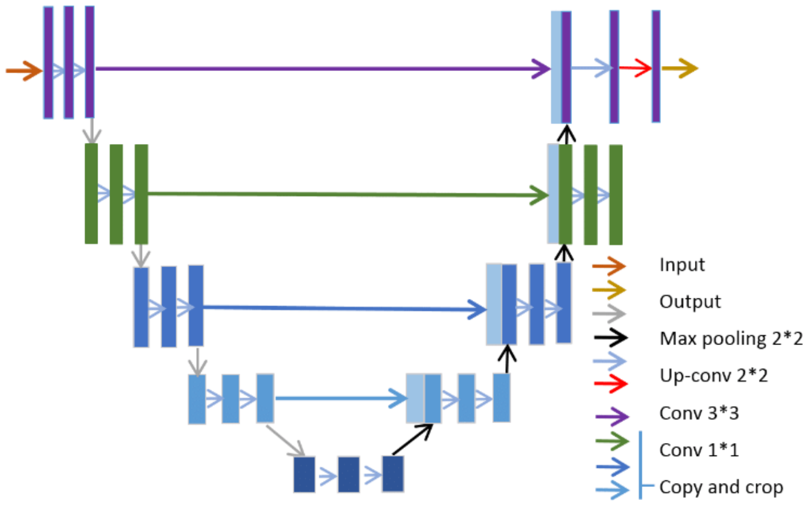
PatchDiscriminator class creation. This class is the discriminator for our GAN.
You can see here, what layers are in the Discriminator.



```
PatchDiscriminator(
  (model): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
    )
  )
)
```

Discriminator print – PatchDiscriminator Class

Next, I created a MainModel Class, which contains the discriminator and our generator. This is the model we have in final in order to predict, and this is the model we train during 70 epochs. It regroups everything we talked before, with backward methods for Generator and Discriminator.



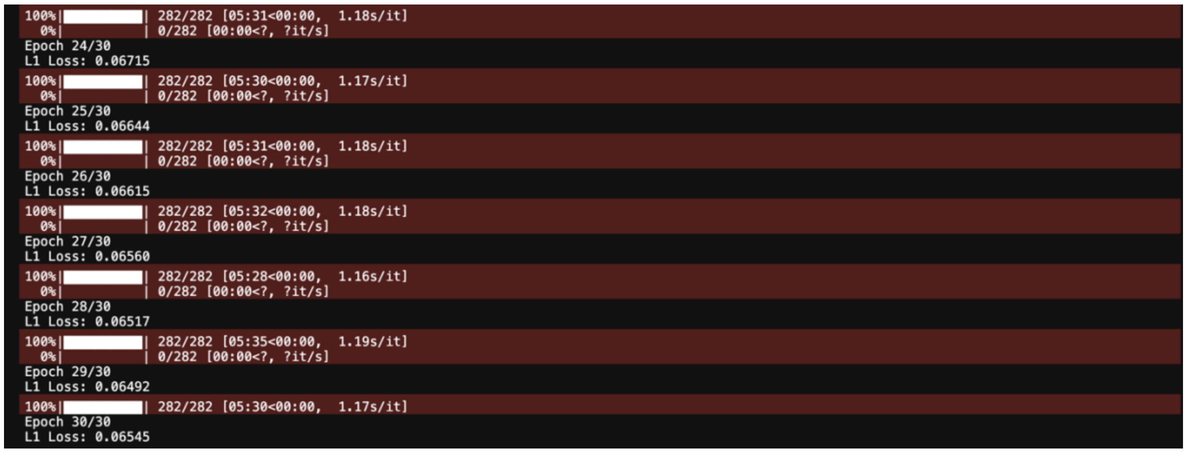
Architecture of U-Net – Generator

After pretrain our Generator, we can now train the GAN, using de MainModel. In the training method (called **train_model**) you can find a parameter called display_every.
I put it to 250 to display prediction on validation set every end of epochs, in order to see how the GAN is learning, and the results, step by step, epoch after epoch. (I will show you the results of this visualization, it's pretty cool to see how he increase his performance epochs after epochs).

As you will see on the notebook, and as I said earlier, I trained the MainModel on 70 epochs. My first try was on 20 epochs with a batch size of 32, and the results was pretty bad. The model was confused between red and blue, we always had grayscale zones on Fake Images, so I choose to increase the number of epochs. And I saw that when I trained during 70 epochs, the model has converged after 35 epochs, and next, slowly decreased the Loss.

I created also handy code is the train method to save a dict as a pickle, containing for each epochs, the loss meter dicts, which regroup : Loss Discriminator Fake, Loss Discriminator Real, Loss Discriminator, Loss Generator GAN, Loss Generator L1, and Loss GAN.
In this dict, I saved also SSIM progressions or each epochs, based on 5 images on the validation set.

The pretrain for the generator model took 2 hours, and the train or the MainModel took around 9 hours for 70 epochs with a batch size at 16.
I was using a GPU V100 Tesla 32GB.

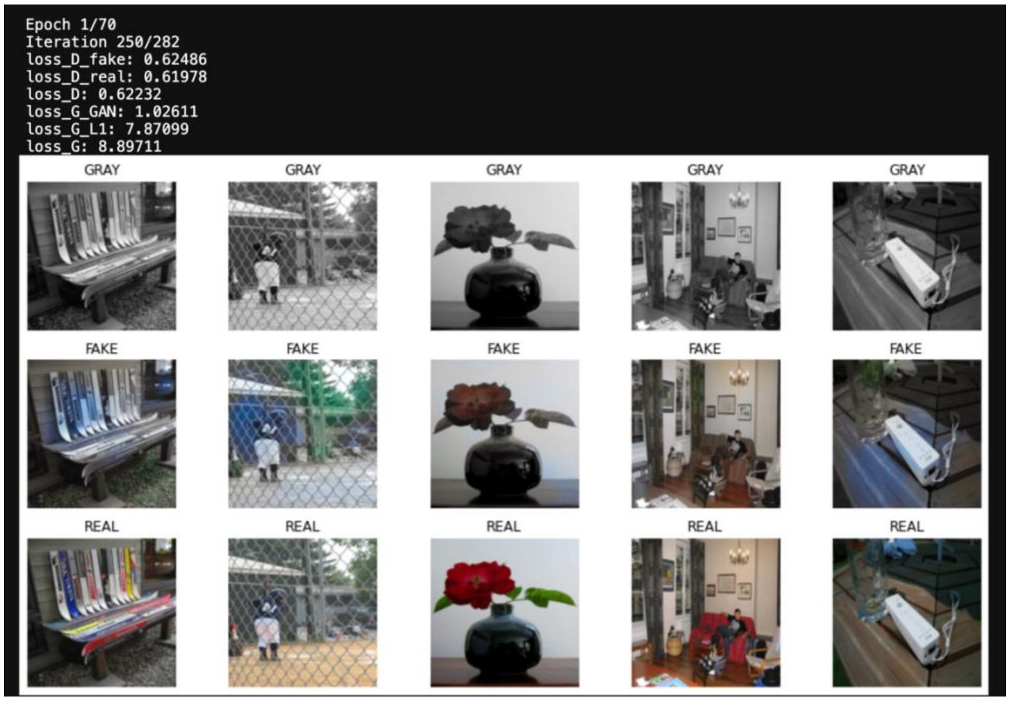


```
100%|██████████| 282/282 [05:31<00:00,  1.18s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 24/30
L1 Loss: 0.06715
100%|██████████| 282/282 [05:30<00:00,  1.17s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 25/30
L1 Loss: 0.06644
100%|██████████| 282/282 [05:31<00:00,  1.18s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 26/30
L1 Loss: 0.06615
100%|██████████| 282/282 [05:32<00:00,  1.18s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 27/30
L1 Loss: 0.06560
100%|██████████| 282/282 [05:28<00:00,  1.16s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 28/30
L1 Loss: 0.06517
100%|██████████| 282/282 [05:35<00:00,  1.19s/it]
  0%|          | 0/282 [00:00<?, ?it/s]
Epoch 29/30
L1 Loss: 0.06492
100%|██████████| 282/282 [05:30<00:00,  1.17s/it]
Epoch 30/30
L1 Loss: 0.06545
```

Screenshot of output during pretrain of Generator. Final L1 Loss: 0.065

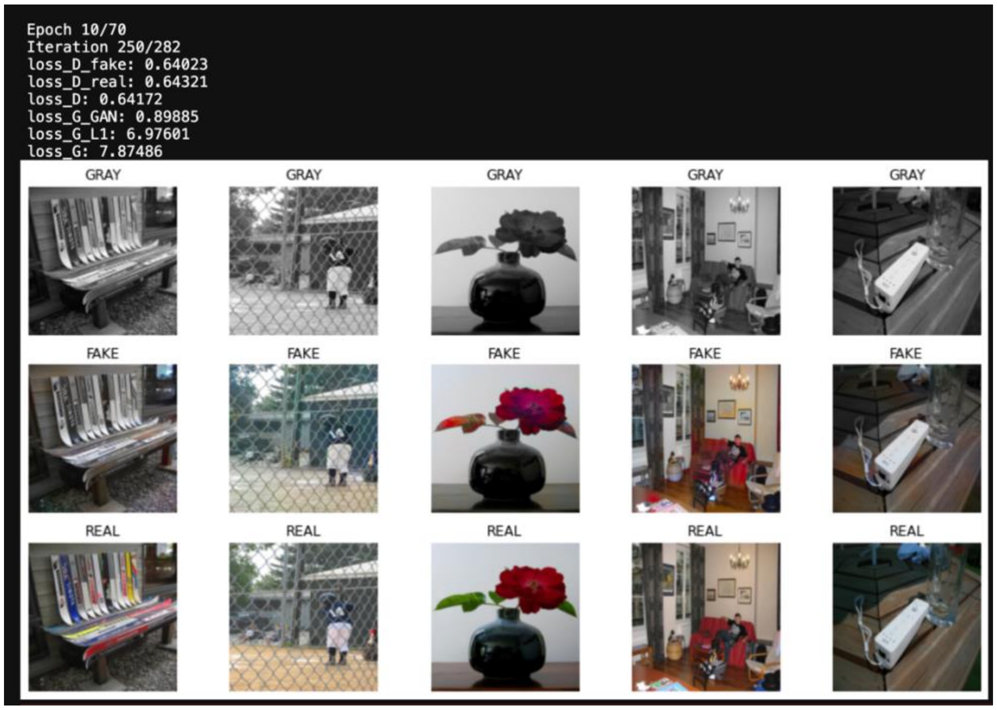
As I explained before, you can now see outputs during the GAN Training with my visualize method every 250 steps for each epochs.

***Gray** are our **L**, **FAKE** is the output of Generator, **REAL** is our target (original image)*



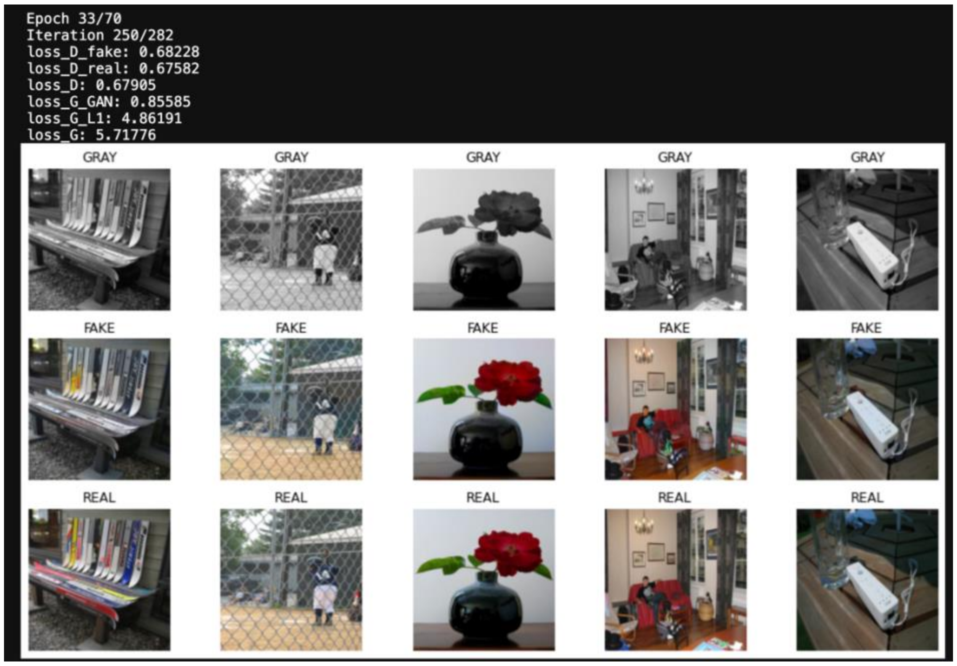
```
Epoch 1/70
Iteration 250/282
loss_D_fake: 0.62486
loss_D_real: 0.61978
loss_D: 0.62232
loss_G_GAN: 1.02611
loss_G_L1: 7.87099
loss_G: 8.89711
```

Epoch 1 – Flower is black and white, bad.



```
Epoch 10/70
Iteration 250/282
loss_D_fake: 0.64023
loss_D_real: 0.64321
loss_D: 0.64172
loss_G_GAN: 0.89885
loss_G_L1: 6.97601
loss_G: 7.87486
```

Epoch 10 – Flower becomes fully Red, with a some green pixels, better but not good.



```
Epoch 33/70
Iteration 250/282
loss_D_fake: 0.68228
loss_D_real: 0.67582
loss_D: 0.67905
loss_G_GAN: 0.85585
loss_G_L1: 4.86191
loss_G: 5.71776
```

Epoch 33 – Flower is identical to the REAL Image. Model has converged. (8.9 Loss epoch 1 vs 5.7 Epoch 33)

X. MODEL EVALUATION AND VALIDATION

To evaluate my model, the first step was to try to predict on all validation samples and see the mean SSIM.

I code this method to do it:

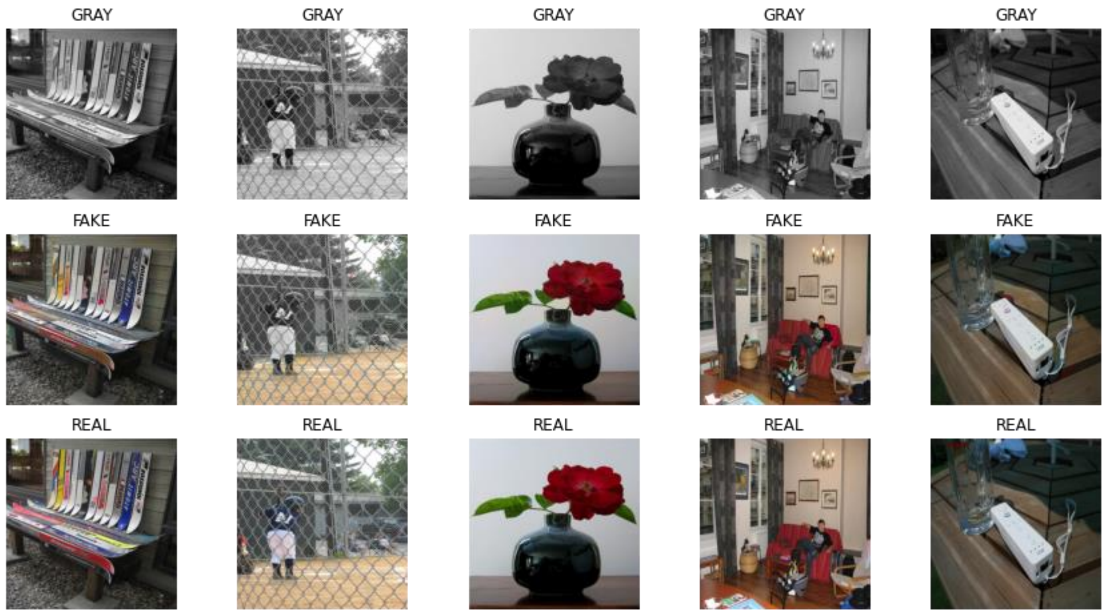
```python
def get_perf_on_val(model, dataloader):
    model.net_G.eval()
    ssim_list = []
    for data in dataloader:
        with torch.no_grad():
            model.setup_input(data)
            model.forward()
        model.net_G.train()
        fake_color = model.fake_color.detach()
        real_color = model.ab
        l = model.l
        fake_imgs = lab_to_rgb(l, fake_color)
        real_imgs = lab_to_rgb(l, real_color)
        for fake_img, real_img in zip(fake_imgs, real_imgs):
            ssim_metric = ssim(fake_img, real_img, data_range=fake_img.max() - fake_img.min(), multichannel=True)
            ssim_list.append(ssim_metric)
    return mean(ssim_list)
```

model parameter is my Generator, and ***dataloader*** is my validation dataloader.

This method works like my visualization method, but it calculate the mean ssim not only for one batch_size on an epoch. It takes the whole data on validation dataloader, and for each of sample in each batch, separate fake_colors and real_colors, create the image list for a given batch, and calculate ssim between real and fake image, in order to append the ssim metric in a list.

Finally, it returns the mean of this list.

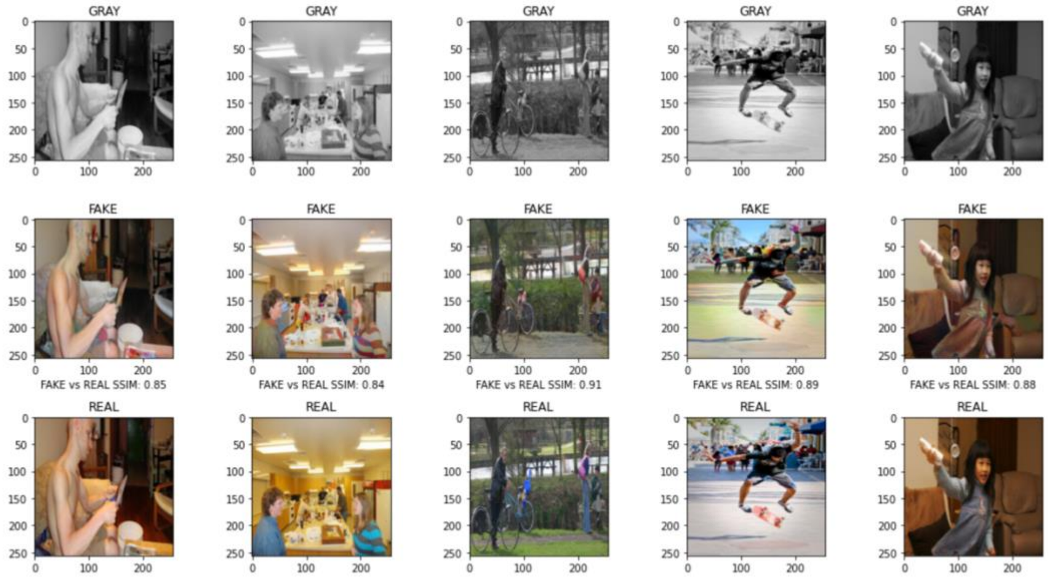
My model performed my validation dataset with a mean SSIM of **0.87**.



Predict on Validation Set Example – Based on Epoch 70, final weights

In my capstone proposal, I said that It's difficult to find a true benchmark about the GANs because the metrics evaluating the quality of the result are mainly based on human's perception. I also said that I will try to demonstrate as I said, an example of bad SSIM metric, but realistic result even if the score is bad.

After benchmark my SSIM metric, I printed some predictions on validation samples, with corresponding SSIM result.



Print images predicted by Generator from Validation Dataset with SSIM score FAKE vs REAL

As you can see, fake images are not so bad, but we can see some mistakes from the model, which is confused between blue / red, or we can see some noisy colors on the head of the man, on the first column. The SSIM score is High (> 80) but do you think that 85% similarity is a good result for the first column?

In my opinion, we have to look to our data results when we are using GANs, we can't just look at measures, loss, results, and tell if our model is good or bad.

This model performs well the task of image colorization, but it can be improved as I said earlier. More Data, More Epochs, but more time and infrastructure consumption too.

You can find more details about the implementation in my notebook.

I hope that you liked this project. I had never done GAN, and I found this project very interesting.