Machine Learning for Computer Vision Project

Valentina Boriano and Simone Vizzuso

September 30, 2022

Abstract

The aim of this project is to develop a neural network able to color images from grayscale. To achieve it, we started the process applying GAN architecture and using the U-net architecture as a generator. We made two different models trained with different datasets [Section: 2]. The results are not very precise, especially in one dataset, due to its complexity. The chosen colors by the model are mostly not similar to the real RGB image. But we also had satisfying results in landscape images, in which the colors are well-represented and in few succeeding in coloring the image entirely. In the model of the other dataset, we had much better results in colors and segmentation [Section: 4].

The code can be found on our GitHub repository of the project [1].

1 Introduction

We decided to undertake the project with a GAN to test its potential and limitations. The choice settled on a GAN to observe how a possible discriminator would perform associated with the U-net. To do this, we created two models with two datasets, one that has a main domain and one that has multiple domain. Naturally, with the first dataset. We had much better results, the colored shape were much more outlined and some of the chosen colors were correct. For the other dataset that contains much more images and with no focus on a specific category, we did not had so satisfying results. Not only on the colors but also on the segmentation. To achieve the best generated images, we have tried different ways. Naturally, the first changes were on the network of the generator and the discriminator. Then we fine-tuned our model, changing the learning rate, loss functions and batch size.

2 Datasets

We have decided to utilize two different datasets. One is composed only by flowers, in fact, it is easier for the GAN to colorize some images. The other dataset is composed by images that include many domains. In the test set, it is visible how more difficult is for the GAN to find the areas to colorize

2.1 Flower Dataset

The flower Dataset, taken from Kaggle [2] is composed by 4242 RGB images of flowers in different foreground. It contains 6 types of flowers: Daisy, Dandelion, Rose, Sunflower and Tulip

All the images before being fed to the neural network are converted in LAB channels and then split in L channel and AB channels and resized into 128x128.

2.2 Multiple domains Dataset

The dataset we have chosen is composed of images of different types; in fact, it does not have a main category of images. They can range from people to cartoons, from landscapes to animals. We chose this type of dataset because we want to better test the power of GAN.

The dataset from Kaggle [3] consists of two compressed zip files:

1. **ab.zip** : This contains 25.npy files consisting of a and b dimensions of LAB color space images.
2. **l.zip** : This consists of a gray_scale.npy file.

There are 25000 images in the dataset but we used only 12660 due to memory problems given by Colab. This is also the reason why we could not test larger datasets. In the training set 12500 images were used, while 160 images are present in the test set.

However, we realized that numerous images in the dataset were already in black and white, some had "off" colors, and some represented unrecognizable subjects (some had only lines or stripes).

3 Model

3.1 What is GAN?

Generative adversarial networks (GANs) are a recent innovation in machine learning. GANs are unsupervised deep learning techniques. It is implemented using two neural networks: **Generator** and **Discriminator**. These two models compete with each other in a form of a game setting.

The GAN model is trained in both, real data and generated data by the generator model. The discriminator's aim is the to determine if the images are real or are generated. The generator is a learning model, so initially, it is likely to produce low or even completely noisy data that does not reflect the real distribution or the properties of the real data.

The **generator model**'s primary goal is generating artificial data that can fool the discriminator. The model starts taking some noise, usually Gaussian noise and produces an image formatted as a vector of pixels. The generator must learn how to trick the discriminator and win a positive classification (produced image classified as real). The discriminator has to learn how to identify those fake images progressively.

Initially, before training has begun, the generator's fake output is very easy for the discriminator to recognize. Since the output of the generator is fed directly into the discriminator as input, this means that when the discriminator classifies an output of the generator, we can apply the backpropagation algorithm through the whole system and update the generator's weights. Over time, the generator's output becomes more realistic and it gets better at fooling the discriminator. Eventually the generator's outputs are so realistic that the discriminator is unable to distinguish them from the real examples.

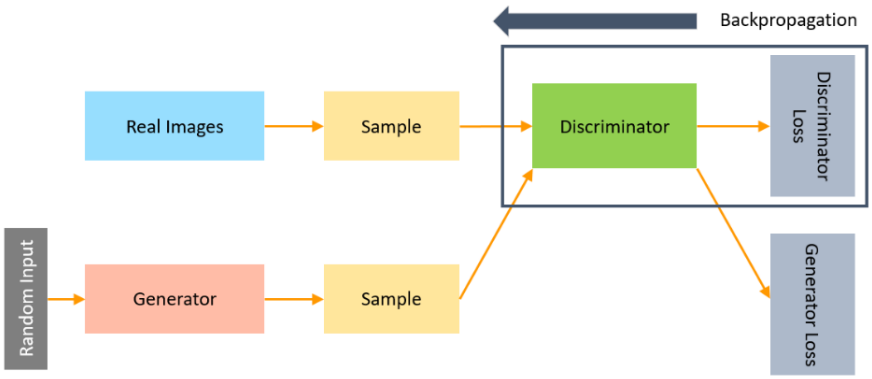


Figure 1: GAN

3.2 Our Approach

Our model is slightly different from the original GAN model. In order to simplify GAN's work, we decided to not utilize the RGB color space but **LAB color space**.

LAB color space uses the L channels for Lightness and two channels a and b (also called Lab or L*a*b). The first channel (a) encodes green-red colors, while the second channel (b) represents yellow-blue colors. With the L channel, it is converted and the final image will be identical to the RGB image. This strategy allows learning with only 2 channels instead of 3, so the probability of "guessing" each pixel drops from 256^3 to 256^2, assuming 256 possible choices per pixel (the number of possibilities for 8-bit images).

Then we did some **preprocessing** for both datasets. The flower dataset was composed only by RGB images, so we had to change the color space into LAB and then divided into L channel and AB channel. After, for both datasets we decided to resize all the images into 128x128 (due to compiling reasons) and we normalize each image. Instead of having values from 0 to 255 we converted it from -1 to 1.

Another important change is the input of our generator model that is not the random noise as in GAN architecture, but is the L channel image. Once fed, the generator has the aim to output an image with 128x128x2 dimensions to fool the discriminator. Then the generated image is concatenated with the corresponding in L channel, and we fed it in the discriminator. The other main change of GAN architecture is about the structure of the Generator. In the section below, we will explain it.

To have the best results, we have tried different batch sizes, learning rates, epochs and loss functions. First of all, we decided to utilize a high batch size (64) but we noticed that reducing it the results were better. The opposite was for the learning rate, in fact, we started with the value 0.0005, till the one that we utilize in our model. The choice of the optimizer is relapsed on the classical GAN optimizer: Adam. For *beta1*, *beta2* and *epsilon* we utilize the standard values of Tensorflow (0.9,0.999,$1e^{-7}$)
The parameters that we decided to utilize are:

- optimizer: Adam
- learning rate: $2e^{-4}$
- loss function: Binary Cross-Entropy with Logits Loss
- batch size: 16
- epochs: 100

3.2.1 Generator

The generator is a U-net that has been divided into code blocks for convenience. The network takes as input the black-and-white image, with only one channel (L) and proceeds first with the encoder blocks, arriving at the plateau and then going up again with the decoder. The final output will consist of an image with 2 channels, in our case a and b channels, which will then serve to be reassembled with the L channel to form the final RGB image.

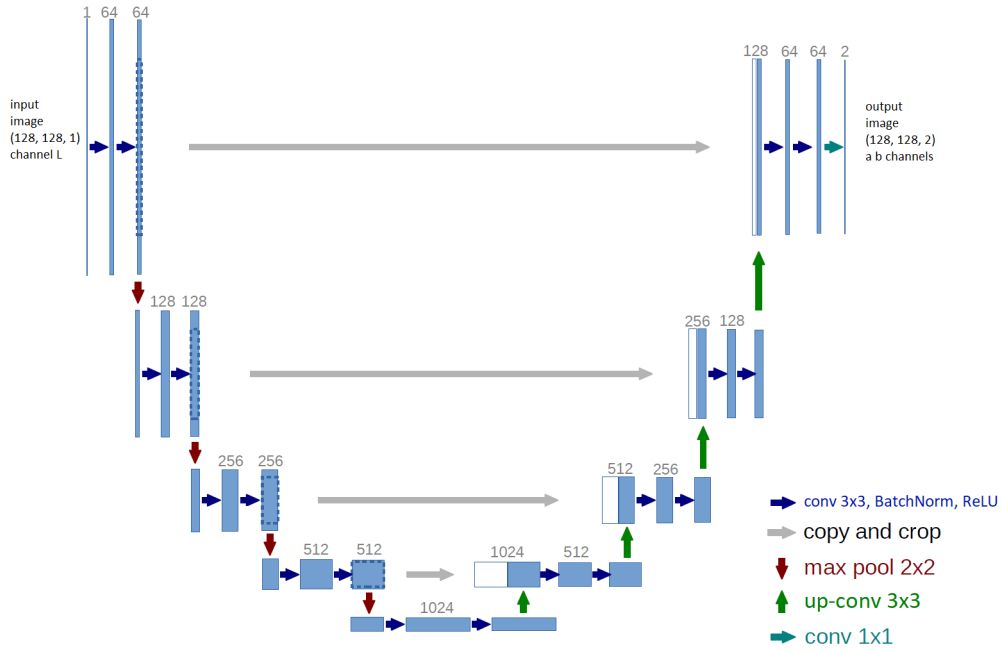


Figure 2: The U-net architecture used as generator

The architecture we used is a standard U-net, a convolutional network architecture for segmentation of images. The idea is to have a network with an initial descending phase that reduces the resolution, increases the number of channels, and, each layer is kept in memory to then be used by the corresponding layer in the ascending phase. In the middle is a plateau that goes back up into an ascending phase where upsampling is done on the image so as to return to the original resolution. Layers in the second part use those in the first part in order to return the image to its original size

There are a considerable number of feature channels in the upsampling part, which grant the network to propagate context information to higher resolution layers. As a consequence, the expansive path is more or less symmetric to the contracting part, and yields a u-shaped architecture. The network only uses the valid part of each convolution without any fully connected layers. To predict the pixels in the border region of the image, the missing context is extrapolated by mirroring the input image. This tiling strategy is important to apply the network to large images, since otherwise the resolution would be limited by the GPU memory.

In our case, as mentioned above, the output will consist of an image with 2 channels. And starting from the initial image, the idea will be to recognize regions of space that can have the same color. The sky, the ground or a wall might be easier to segment, but areas such as a field of flowers or multiple overlapping objects, in our case where we are coloring and not classifying, might be a problem.

3.2.2 Discriminator

The discriminator that we utilized take in input the image with 3 channels (LAB), so the concatenation between the input image in L channel and the generated image in AB channels. At the beginning, the architecture of our discriminator was a model built by stacking 4 blocks of Conv-BatchNorm-LeackyReLU to decide whether the input image is fake or real. During the first training, we have seen that the discriminator was too clever, in fact in some epochs he was able to define with high certainty if the image was real and if it wasn't. To reduce that problem, we decided to add a dropout layer to each block of our discriminator. In the last block, we do not utilize the sigmoid activation because it is already embedded in the loss function that we utilize.

3.2.3 Loss

Both the generator and the discriminator use the **binary cross-entropy with Logits** loss to train the models. Binary cross-entropy with logits loss combines a Sigmoid layer and the BCELoss in one single class. It is more numerically stable than using a plain Sigmoid followed by a BCELoss.

The **discriminator** can have two possible inputs, real or fake.

For a real input, y = 1. Substituting this in the binary cross entropy loss function gives the below loss value:

$$L(1, \hat{y}) = -log(D(x))$$

where D(x) is the output of the discriminator for the real image

Similarly, for a fake input, y = 0, which gives the below loss value:

$$L(0, \hat{y}) = -log(1 - D(G(x)))$$

where G(x) is the generated image by the generator, so D(G(x)) is the output of the discriminator with the generated image

Combining both the above losses for the discriminator, one for a real input and the other for a fake input gives the below discriminator loss

$$L(D) = max[log(D(x)) + log(1 - D(G(x)))]$$

For the generator, loss is calculated from the discriminator loss. The generator is trying to fool the discriminator into classifying the fake data as real data. This implies that the generator tries to minimize the second term in the discriminator loss equation. The generator loss function is:

$$L(G) = min[log(1 - D(G(x)))]$$

4 Results

The results are not as good as expected, but are still satisfying. We trained two different models, one for the flower dataset and one for the multi domain dataset.

4.1 Flower Dataset

With this dataset, being easier to learn, we had very nice results with little change from the real image:

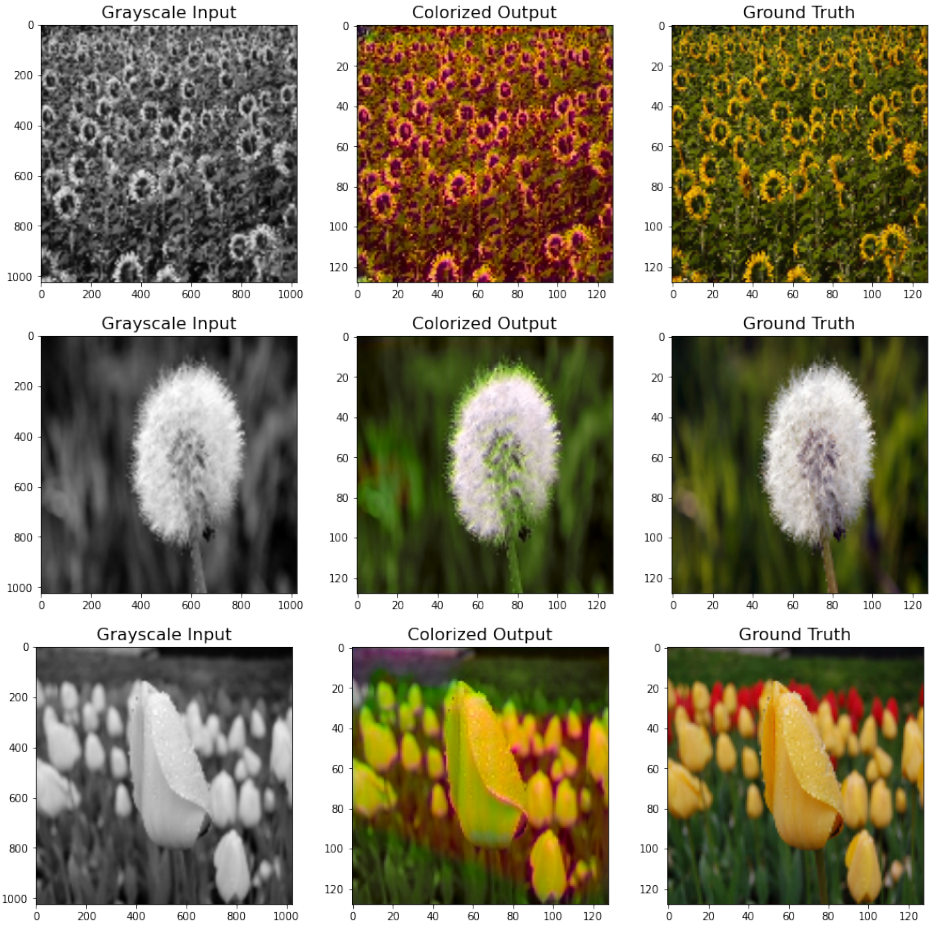
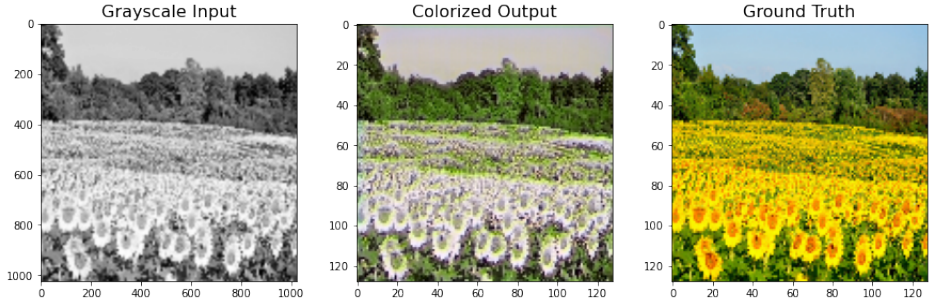


Figure 3: Best Results on flower dataset

We had also not that precise results, with the correct segmentation on the flower and the background but with the wrong colors:



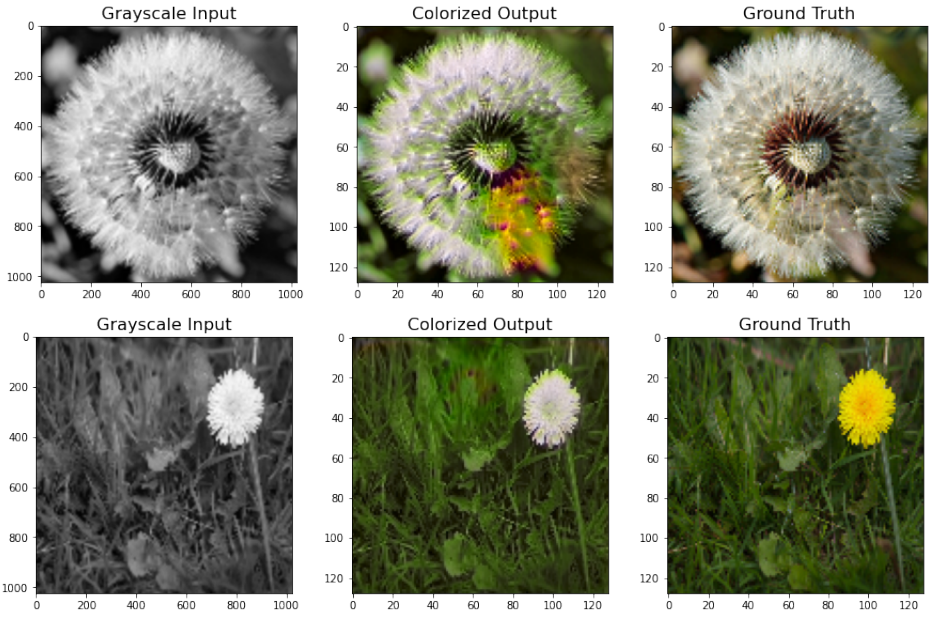
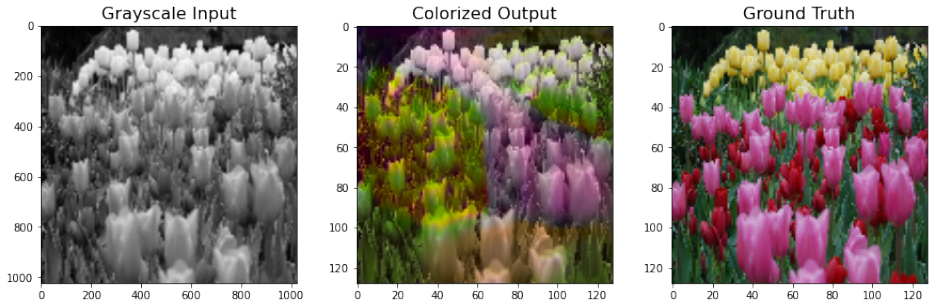


Figure 4: Wrong colors on flower dataset

As you can see, all these images, have not the correct color for the flower or some stain of color inside. The color error, is probably due to not having the labels that define the flower. So for the GAN is more difficult to choose the correct color.

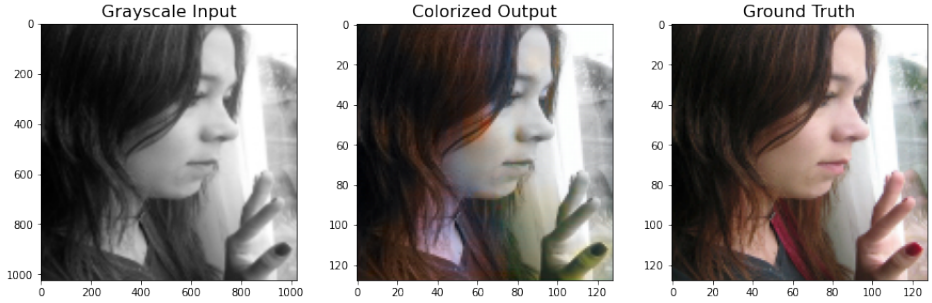
Then the last type of error is on defining the correct segment to color. This is an example:



As you can see, the tulips were divided in four main sections and all of them colored with different colors.

4.2 Multiple Domain Dataset

With this dataset, the difficulties increased much. Not having a main domain in which being trained, the GAN tries to improve itself to have the best results. The worst generated images were the one that included people or very complex images like:



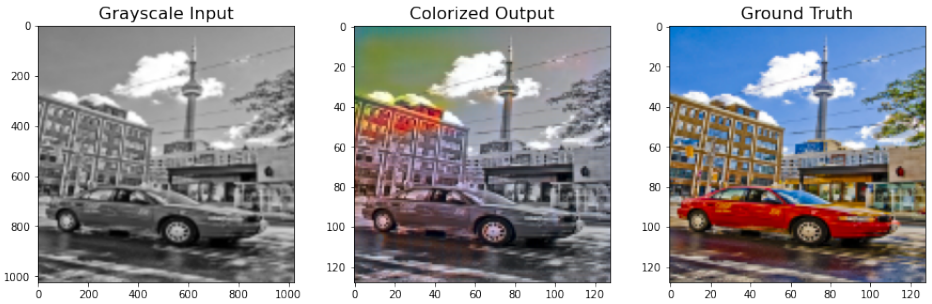


Figure 5: Worst results on multiple domain dataset

As you can notice, not only the colors are wrong but also the segmentation is not correct.

The most satisfying results were on landscape images, in which the colors are not perfect but still believable:

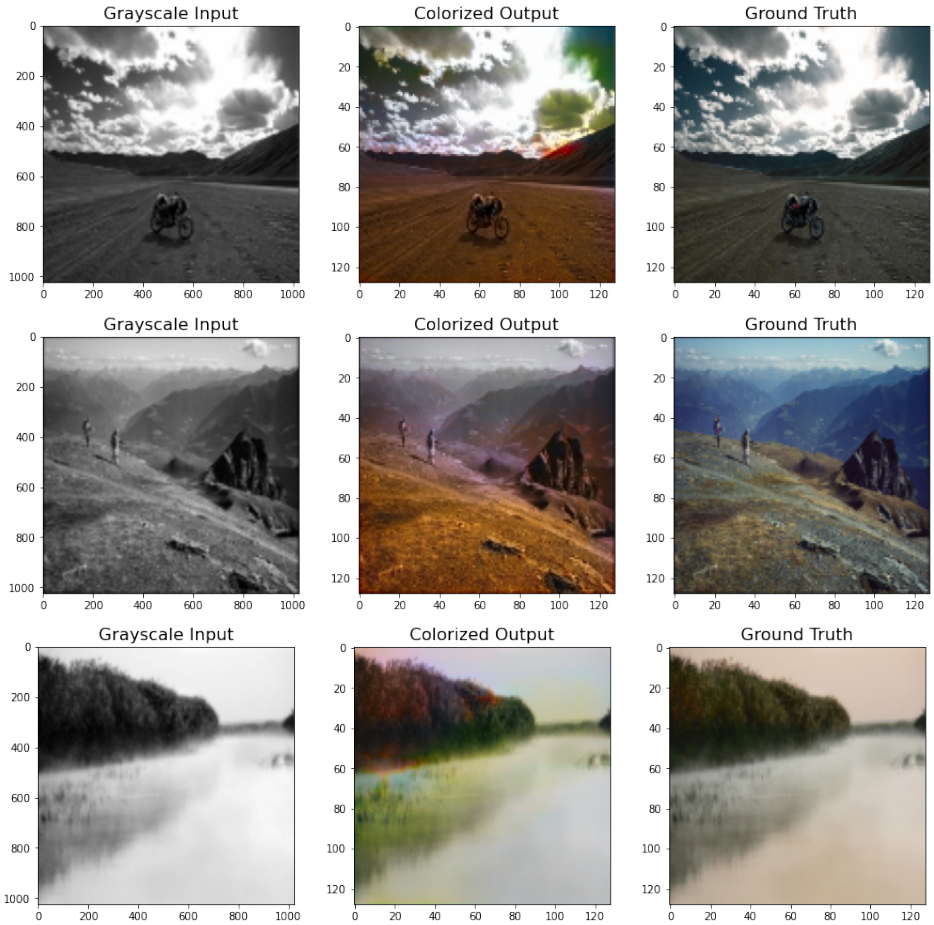


Figure 6: Best results on multiple domain dataset

We also have nice results on "easy" images concerning animals or "stuff":

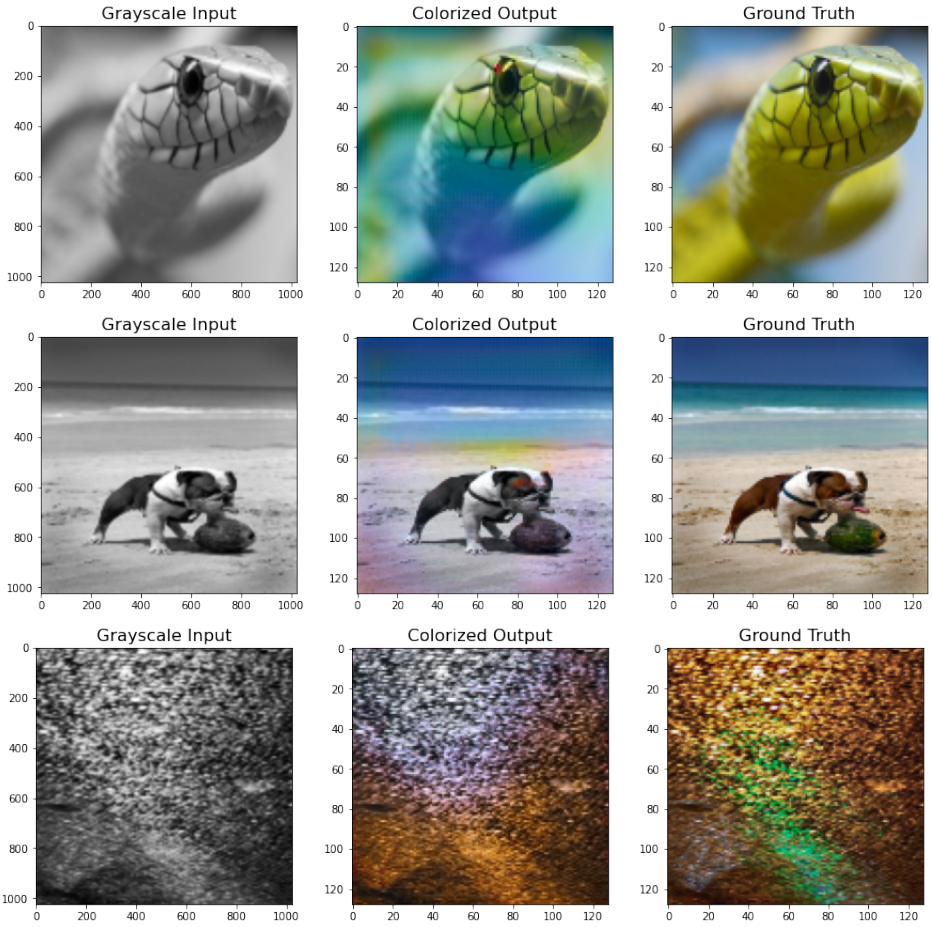


Figure 7: Wrong colors on multiple domain dataset

As said before, the complexity of the dataset, does not allow our model to be precise neither with segmentation nor in colors. We decided to try also this dataset just to see how much was powerful and able to generate colored images.

5 Conclusion

The results are not very satisfying, in fact, as you can see many generated images have not the correct color (as in Figure: 4) and in some also the colored area is wrong in respect of the background or foreground (as in Figure:5). Instead, some results are very realistic, although it has distinct colours in respect to the original one (as in Figure: 6). One improvement that we can implement is to utilize a DCGAN rather than a GAN, because with that architecture is possible to include also labels of images, and this can enhance the right choice of the color and the segmentation. Another possible improvement could be to utilize a pre-trained generator.

References

[1] Project URL https://github.com/valentinaboriano/Image-Colourization

[2] Flower Dataset https://www.kaggle.com/datasets/alxmamaev/flowers-recognition

[3] Multi Domain Dataset https://www.kaggle.com/datasets/shravankumar9892/image-colorization