Image–to–Image Translation with Conditional Adversarial Networks

Cecilia Serrano Moreno, Kirill Sirotkin

***Abstract*—CycleGANs have become a very popular solution for the image-to-image translation task because of their exceptional results and the capacity of avoiding the need of paired datasets. This quality eases the utilization of this model for any problem where the acquisition of a dataset is particularly difficult, for example, in any medical imaging application. In our previous work, we used two CycleGANs working together to create a generalization of the model where the translation is performed between three domains instead of two. The addition of an extra domain proved to contribute to better results when the translation is performed between the other two domains. Nevertheless, the higher complexity of the system that was presented makes the training process unstable and the convergence tough to accomplish. For all these factors, it is hard to analyse the results or to answer why and how the improvement occurs. Motivated by this, in this work we propose to perform an analysis of the CycleGAN architecture, focusing on the visualization of the inner layers of the generator. For this, we shift the application of the model to a very basic colorization task using a simple toy dataset for colorizing geometric shapes. To perform the experiments, we developed an interface that allows to create a personalized toy dataset and visualize the training and testing processes. Finally, we conducted a list of experiments were we try to shed light on the behaviour of the CycleGAN filters and how the latent space of the U-Net encodes the information for the generation of the translated images.**

I. Introduction

Generative Adversarial Networks (GANs) and specifically CycleGANs [1] have shown impressive results in the recent past. However, along with the advantages that they have, there are certain problems and difficulties related to the training process. Among those, one can list:

- Mode collapse - the collapse of the generator, which results in the generation of limited series of samples
- Non-convergence or slow convergence - the inability of the loss function to converge to a global minimum due to its high complexity
- High sensitivity to hyperparameter selection
- Vanishing gradients - the discriminator becomes too successful at distinguishing fake samples from the real ones. As a result, the gradient of the generator vanishes and its learning stops
- Difficulty to produce high-resolution images

Many recent works were dedicated to these problems. Some of them are based on large-scale empirical studies, where different GAN architectures are exhaustively fine-tuned to produce high performance [2]. Others evaluate the benefits of different cost functions, as in the Boundary Equilibrium Generative Adversarial Networks (BEGAN) [3] and Wasserstein GANs (WGAN) [4].

The key to the understanding of the causes of these problems is a deep and thorough knowledge of parameter optimization process. It is generally gained through consistent theoretical analysis, besides that, sometimes it is very useful to have visual data to notice certain patterns and make conclusions from them. In this work, we focus on the latter, i.e., in the visualization techniques to view the responses of inner layers of a U-Net [5] based generator, using toy datasets and simple CycleGAN architectures. We build a responsive GUI that allows to view the evolution of weights in the generator networks during the training process, create simple datasets and observe the responses of the inner layers during test.

The rest of the paper is organized as follows: Section II reviews the existing research, state-of-the-art and our previous work. Section III introduces the proposed visualization solution from the conceptual point of view. Section IV presents the technical details of the implementation of the proposed solution, that lead to a set of conclusions in section VII. Finally, the results obtained from the visual information are discussed in the section VI, and the summary of the potential future work, as well as the conclusions drawn from this work, are given in the section VII.

II. Related work

A. Generative Adversarial Networks

The introduction of the concept of adversarial networks in [10] allowed to develop a method of training Generative models, which can produce sharp probability distributions. In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution.

Further research suggested new training techniques and refinements that allowed to implement Deep Convolutional GANs (DCGANs) [11]. The suggested architectural guidelines allow the possibility of unsupervised learning with DCGANs, hence one can make use of large quantities of easily accessible unlabeled data, i.e., videos and images. Apart from that, the authors managed to identify the purposes of some separate neurons, therefore making the whole neural network less of a black box. This allowed to change the function of a trained neural network by simply disabling or enabling certain neurons.

The following works proved that, instead of having the probabilities of a sample being fake or not fake at the output of the discriminator, one can utilize a different approach, like the values of the energy function [12] or the features extracted from intermediate layers [13]. Also, various methods like minibatch discrimination, historical average, one-sided label smoothing and virtual batch normalization were proposed to be used in the GANs training process.

B. Image-to-image translation using GANs

The work that was done in GANs built the foundation for the general framework for the image-to-image translation task known as pix-2-pix [9], capable of converting or 'translating' one possible representation of data into another. This approach uses a GAN model and, instead of using random noise —as it was done in the original architecture—, takes an image from one domain as the input. The generator is supposed to convert that image into the second domain, while the discriminator compares the result with the corresponding image in that domain.

A serious issue with pix-2-pix framework, however, was its dependency on the paired data, which was resolved with the advent of CycleGANs [1] that can operate with unpaired datasets. To achieve this, the authors used two GANs instead of one and introduced a cycle consistency loss to train them. In this novel architecture, one of the GANs learns the translation from one domain (A) to another (B) and the second GAN learns to perform the opposite conversion: B to A. This way, an image can be translated from A to B and back to A, so the difference between the original image and the result after the whole cycle can be used to evaluate the quality of the translations.

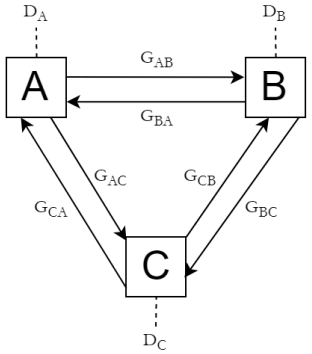
C. Previous work

Seeing the improving results and the amount of research related to CycleGANs [1] we dedicated the first part of our TRDP to research of these models. Specifically, we proposed a slight change in the original architecture, by incorporating a third domain, as can be seen on the Fig. 1, section (a), and modifying the objective function. Considering a significant increase in complexity which the proposal introduces, a simplification to this architecture can be made. It is shown on the Fig. 1, section (b). The final objectives for both: full and simplified proposals are given in the equations 1 and 2, respectively.

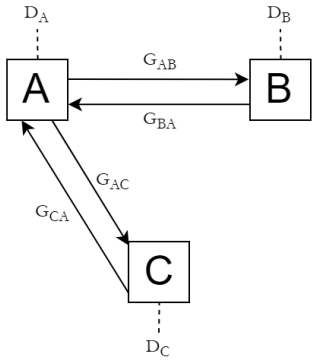
$$\begin{aligned}\mathcal{L}(G_{AB}, G_{BA}, G_{AC}, G_{CA}, G_{BC}, G_{CB}, D_A, D_B, D_C) = \\ \mathcal{L}_{GAN}(G_{AB}, D_B, A, B) + \mathcal{L}_{GAN}(G_{BA}, D_A, B, A) + \\ \mathcal{L}_{GAN}(G_{AC}, D_C, A, C) + \mathcal{L}_{GAN}(G_{CA}, D_A, C, A) + \\ \mathcal{L}_{GAN}(G_{BC}, D_C, B, C) + \mathcal{L}_{GAN}(G_{CB}, D_B, C, B) + \\ \lambda_{AB}\mathcal{L}_{cyc}(G_{AB}, G_{BA}) + \lambda_{AC}\mathcal{L}_{cyc}(G_{AC}, G_{CA}) + \\ \lambda_{BC}\mathcal{L}_{cyc}(G_{BC}, G_{CB}) \quad (1)\end{aligned}$$

$$\begin{aligned}\mathcal{L}(G_{AB}, G_{BA}, G_{AC}, G_{CA}, D_A, D_B, D_C) = \\ \mathcal{L}_{GAN}(G_{AB}, D_B, A, B) + \mathcal{L}_{GAN}(G_{BA}, D_A, B, A) + \\ \mathcal{L}_{GAN}(G_{AC}, D_C, A, C) + \mathcal{L}_{GAN}(G_{CA}, D_A, C, A) + \\ \lambda_{AB}\mathcal{L}_{cyc}(G_{AB}, G_{BA}) + \lambda_{AC}\mathcal{L}_{cyc}(G_{AC}, G_{CA}) \quad (2)\end{aligned}$$

These changes made it possible to outperform the original CycleGAN model in a task of semantic labelling of objects on the road footage. Apart from that, they also showed increasing difficulty in the training process and brought up some of the problems discussed in the Introduction I of this paper. In order to understand and solve them, further analysis is needed. That is why we dedicate visualization approaches of CycleGANs.



(a) Proposed architecture



(b) Simplified proposed architecture

Fig. 1: In the proposed architecture (a) all the possible connections between the three domains are performed, hence, six generators are needed. The simplified version (b) differs from it in the lack of connections between B and C.

D. Filter visualization in Convolutional networks

Filter visualization of Convolutional Neural Networks (CNNs) is an important task that allows them to be interpretable and not be used just as "black boxes". Many approaches for filter visualization have been proposed during the last decade to obtain images that offer meaningful information. Probably the most straightforward option for solving this task is simply displaying the filters without making any changes on them. The problem with this technique is that, when trying to inspect deeper layers, the filters become more intractable.

A more elaborated practice for visualizing the task that each filter has learned is to look for the input that produces the highest activation in the filter under study. A popular method for accomplishing this task is the optimization of the input (starting, for example, from random noise), to obtain the input that maximizes the response of a given neuron, channel or layer, as it is done in [7] and [6]. Other methods are the characterization the filters as a linear combination of the filters in the previous layer with the largest weights, as it is used in [6], and the inversion of the convolution block that would map features to pixels, as it is proposed in [8]. All these techniques

produce satisfying results, but they are quite complex, as some of them require an optimization step.

The solution that we propose for this problem is to visualize the responses of a chosen image to the filters. In this way, we expect to obtain more meaningful images that can be interpretable even for deeper layers of the network.

E. U-Net

U-Net [5] is a convolutional network that was originally created for solving the task of image segmentation in the context of medical imaging. One of its most remarkable strengths is that the dataset required for training this network can be very small and the model still yields proper results, what makes it perfect for medical imaging, where the amount of data is very limited. Apart form the segmentation application, U-Net has a great potential to be used in generative models or image-to-image translations; for instance, U-Net is one of the architectures that are used in the CycleGAN implementation.

The architecture of the U-Net is similar to an autoencoder, but it also has skip connections that preserve the spatial information that might be lost in the bottleneck. In Fig. 2 we can see the structure of the model. The contractive part consists of a classical convolutional network with four blocks composed of two convolutional layers, each of them followed by a ReLU, and a final max pooling layer. After the contractive path, there is a bottleneck where we can find two convolutional layers, also each of them followed by a ReLU. In the bottleneck, the dimensionality is reduced to an array of values that can be seen as the latent representation in which the image is encoded. The bottleneck is followed by an expansive path that consists of four ”up-convolutional” blocks, similar to the block from the other path, but in this case they yield an upsampling after each block. The input information for this path comes from both the latent representation from the bottleneck and also the skip connections. More specifically, the input of each ”up-convolutional” block is the output of the previous layer concatenated with the output of the corresponding block from the contractive path.

In the CycleGAN model, the one that is used in this work, one of the architectures that is utilized is a version of U-Net. The smaller size of this network in comparison with other networks that are supported in the CycleGAN implementation, made us choose it for the experiments.

III. Proposed method

The objective in this work is to visualize the filters learned by the generator. For this purpose, the idea is to use a very simple problem of image-to-image translation that allows to focus on the properties of the network over the performance. Our proposal is to use a toy example only composed by geometrical shapes on a uniform background, that is expected to be easy to train and visualize. The modality of translation in this case is changed to a colorization problem, i.e., one of the domains is composed by gray level images and the other

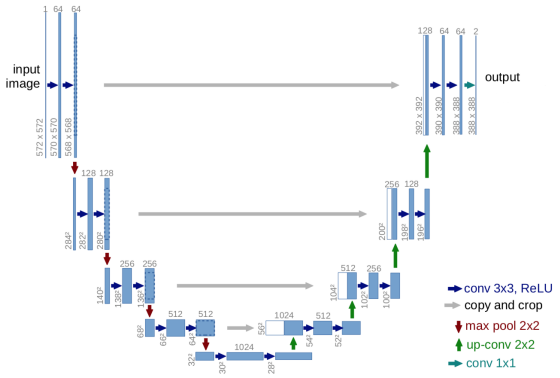


Fig. 2: Architecture of U-Net, from [5]

one by RGB images. On the dataset, we impose the constraint that all the shapes on the gray level images should have the same gray level value, this way, the generation of color is not learned from the value that is encountered in the gray level image, but only from its context, this is, from the shape of the geometric figures.

Regarding the visualization of the filters, the proposal is to utilize the response of each filter to a chosen input image. In this way, we expect the results to be more meaningful than the simple display of the filters, since it allows to see, not only the isolated filters, but also the effect of the concatenation of the different operations performed in the layers.

Apart from the visualization, we propose another method for understanding the involvement of each module of the network in the generation of color: the cancellation of the response of some filters to see its repercussion on the output of the network. This method would potentially isolate the different functionalities encoded in the network and locate them in each of the units.

IV. Implementation

Developed software was designed to meet the following requirements:

- Have an easy to use GUI to train a CycleGAN and track the training progress
- Have an easy to use GUI to test an already trained generator network (U-Net), view the responses of all the layers to input data, change the responses to see how they affect the image translation
- Have a GUI to generate toy Datasets

To implement these functionalities the following modules have been implemented:

- The underlying pix2pix and CycleGAN framework
- Toy dataset generator
- Network parser
- Filter responses modifier
- GUI

All parts of the software architecture are discussed in detail below.

A. pix2pix and CycleGAN framework

Because the research work is dedicated to CycleGANs based image-to-image translation, the original implementation was chosen. It is written in python 3 with PyTorch. It is a fairly big project, therefore, not all parts are addressed in this work. However, some of them are worth mentioning. Among those, are the packages **models** and **data**.

1) Package models: All neural network models are defined in this package and are inherited from the abstract class BaseModel implemented in the module base_model.py. The helper functions are present in the module networks.py. In our work, only CycleGANModel from the module cycle_gan_model.py was used. This class implements a classical CycleGAN architecture. The object named *model* used in the visualization package refers to the instance of some network model, and in this work it is always an object of the class CycleGANModel. Our contribution from the previous work is the addition of the new module cycle_gan_3_model.py implementing a CycleGAN network with 3 domains. This is done by the addition of 1 discriminator, 2 generator objects, as well as the methods running backpropagation on these networks. Finally, the helper functions' module networks.py was slightly extended by specifying custom U-Net architectures in the function define_G(...).

2) Package data: All dataset classes used in the framework are inherited from the abstract class BaseDataset implemented in the module base_dataset.py. In this research work, only objects of the class UnalignedDataset from the module unaligned_dataset.py were used. This class allows to create unaligned datasets of images, as well as performing basic preprocessing, such as resizing, cropping, flipping and converting to grayscale. The preprocessing is done via torchvision.transforms. A defined dataset object, apart from the other information, contains 2 lists: A_paths and B_paths, that store paths to images from two domains. Each dataset model inherited from the BaseDataset abstract class implements the __getitem__ method that loads the images from A_paths and B_paths and applies preprocessing. Our contribution to this package is the following modification of the get_transform(...) function that allows to load grayscale images in the UnalignedDataset model.

Besides that, we wrote the implementation of the Unaligned3Dataset class that allows to create unaligned datasets of 3 domains, which was used in the architecture presented on the Fig. 1. The aforementioned implementation is given in the module unaligned_3_dataset.py

B. Toy dataset generator

The implementation of the toy dataset generator is given in the module data_generator.py of the visualization package. The main logic is encoded in the methods gen_triang(...), gen_circle(...) and add_gaussian_noise(...). By default, all generated shapes are centered in the middle of the canvas and have no noise.

C. Network parser

In order to visualize the network, it is necessary to be able to access its different layers independently. However, sometimes it is not easy, because the architecture might be not a simple sequential model. Indeed, the implementation of the U-Nets in the original CycleGAN repository is somewhat recursive and the **UnetSkipConnectionBlock**s are wrapped in each other (as can be seen in the class UnetGenerator in the module networks.py of the package models). In order to unwrap such architecture, we wrote a network parser, which can be found in the module pytorch_parser.py of the package visualization. The code extracts the independent layers from the wrapped model and returns two lists: one containing the layers themselves, and one containing the indices of layers between which the skip-connections were found. This allows to later restore the U-Net architecture and perform forward propagation through slices of the whole U-Net, instead of the whole network.

D. Filter responses modifier

In order to manually manipulate the values of filter responses, we use masks for each layer of the network. A mask is a tensor of the same shape as the layer it is applied to, containing numbers of type double. Whenever forward propagation through the network is requested, the result that the layer produces is multiplied by the mask element-wise, as a result one can amplify or suppress responses of certain filters and track their influence on the final image. The masks are stored in dictionary with the keys that correspond to each of the CycleGAN generator networks, and values that correspond to the lists of tensors representing masks for each layer. The implementation is done in the methods make_mask(...), reset_masks(...), suppress_layer(...), update_response_at(...) and forward(...) of the module visualizer.py in the package visualization.

E. GUI

The GUI of the developed software is implemented with the use of PyQt version 5.9.7, as a flexible cross-platform toolkit. The GUI consists of three main windows:

- Training visualization
- Test visualization
- Dataset generator

A more detailed explanation of each of them is given below:

1) Training visualization: The main window of the GUI (presented on the Fig. 3) consists of five graphics widgets that are, in fact, objects of the class ImageViewer, declared in the module image_viewer.py of the package visualization. ImageViewer is inherited from standard PyQt5.QtWidgets.QGraphicsView widget and extends its functionality by adding features such as zooming and dragging, which allow the user to research the images more carefully. From the coding point of view, the implementation of this window and the logic behind it is given in the module visualizer_train.py of the package visualization. This module consists of 3 classes:

- VisualizerTrain - loads the UI layout and handles interactions between the user and the application, i.e., handling input events, displaying the visual results of the training process and others.

- Training - inherited from PyQt5.QtCore.QRunnable, implements the actual neural network training proccess and runs as an independent thread in the PyQt5.QtCore.QThreadPool defined in the VisualizerTrain.
- Communicate - implements communication between the classes VisualizerTrain and Training by the means of Qt signals.

Once the dataset is selected and the network model is loaded, the responses of the selected layers are displayed in the graphics view widgets. This is done via the methods view_response(...) and forward(...), while display(...) simply plots the data in the graphics view widget and view_response(...) constructs a concatenation of small images representing responses of multiple feature maps, the forward(...) method implements the main logic, that is, a forward propagation through a slice of the network in a recursive manner. Apart from the visualization of the responses of the feature maps, the GUI contains a graphics view depicting the evolution of network losses. The plots are implemented with the use of matplotlib.pyplot and seaborn.

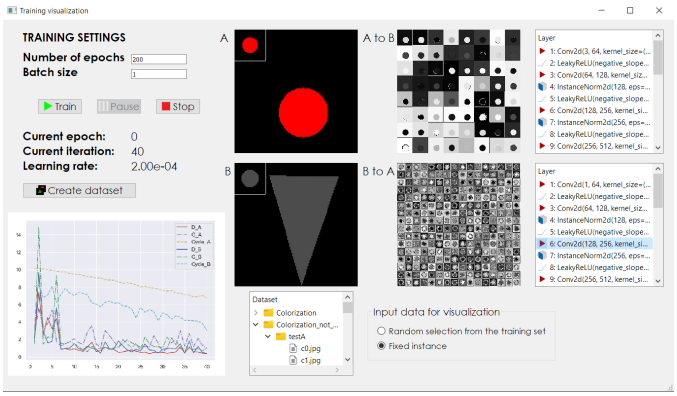


Fig. 3: GUI of the training visualization.

2) Test visualization: The implementation of the part of the GUI responsible for visualization of the feature maps' responses during test time (presented on the Fig. 4) is similar to the implementation of the "Train visualization" part of the GUI. It is stored in the module visualizer.py of the package visualization and consists of one class Visualizer. The class implements user-machine interaction and graphics rendering in the same manner as the "Train visualization" part of the GUI.

3) Dataset generator: Finally, the last part of the GUI is the interactive toy-dataset generator, presented on the Fig. 5. The implementation is given in the class DataGenerator, located in the module data_generator.py of the package visualization. The main logic of the data generation is encoded in the methods gen_traing(...) and gen_circle(...). All of the methods are fairly simple and do not require a deep explanation.

F. Requirements

The software requirements to run the code are the following:

- Python 3.6.9

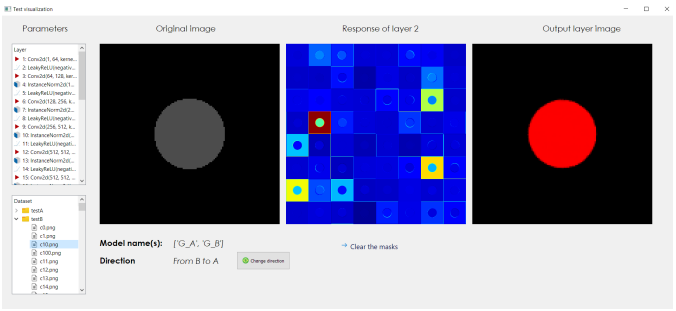


Fig. 4: GUI of the test visualization.

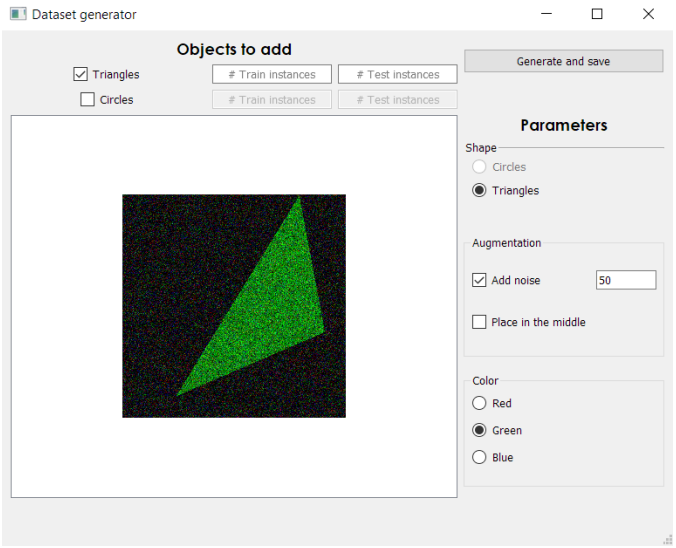


Fig. 5: GUI of the toy dataset generator.

- pytorch 1.2.0
- torchvision 0.4.1
- cudnn 7.6.4
- opencv-python 4.0.0.21
- dominate 2.3.5
- visdom 0.1.8.8
- numpy 1.16.5
- PyQt 5.9.7
- seaborn 0.9.0
- matplotlib 3.0.2

G. Gantt chart

The Gantt chart of the project is presented on the Fig. 6

V. Experiments

The experiments section focuses on the problem of colorization of gray scale images on a toy dataset. Overall, the conducted experiments address two main issues: the first one is to find the best performing architecture so that the following experiments are more meaningful and the second one is the research of a series of situations where we change some features of the network and analyse the outcome. Apart from that, although some of the experiments are evaluated quantitatively, primarily we provide a qualitative evaluation for them.

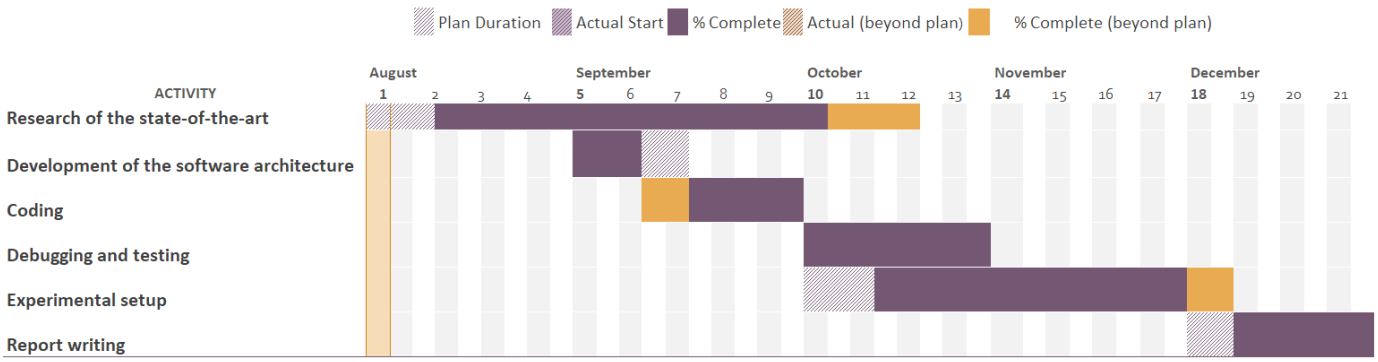


Fig. 6: Gantt chart of the project

A. *Toy dataset*

The dataset used in the experiments described below represent the colorization problem and consists of geometric shapes, namely, circles and triangles, in RGB and gray scale domains. To ensure that the colorization is learned from the context of the image, we created the dataset with only red circles and green triangles, so that the network learns to colorize with one color or another depending only on the shape of the object. In all the experiments, the figures are placed in the center of the image, with the exception of the experiment devoted to the displacement of the figures, where a random shift is introduced in all the images. An example of the dataset without displacement is given on the Fig. 7. All images have a size of 256x256 pixels. The network models evaluated in this work resize the input images to 128x128 pixels.

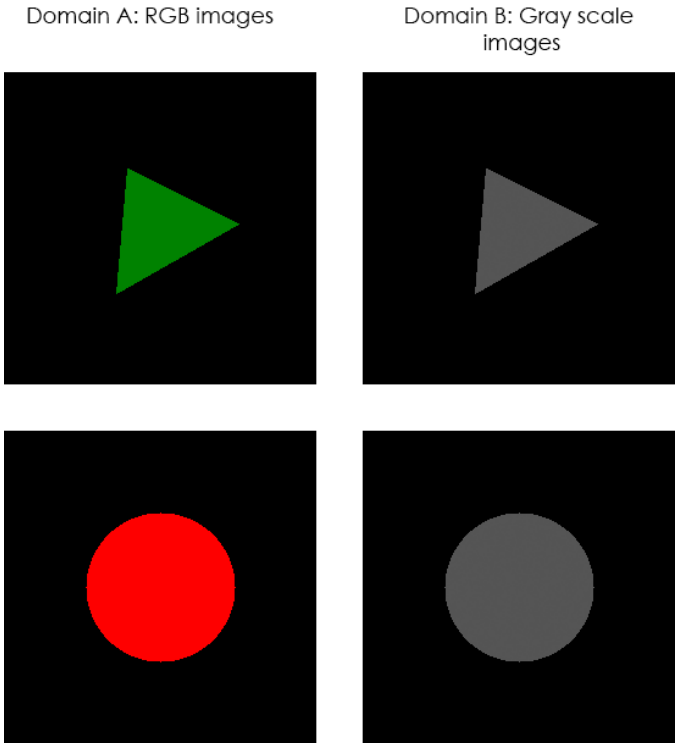


Fig. 7: The toy-dataset used for the experiments. Domain A consists of green triangles and red circles, located in the center of the image; domain B consists of their respective gray scale versions

B. *Setup*

All the experiments have been conducted on two machines with the following specifications:

- Machine #1
  - CPU: Intel Core i7-7700 HQ @ 2.80 GHz, 4 physical cores, 8 logical cores
  - GPU: NVIDIA GeForce GTX 1050, 4GB, GDDR5
  - RAM: SODIMM Samsung [M471A1K43CB1-CRC], 16 GB, DDR4, 2400 MHz
  - OS: Windows 10 Home edition
- Machine #2
  - CPU: Intel Core i7-5500 U @ 2.40 GHz, 2 physical cores, 4 logical cores
  - GPU: NVIDIA GeForce GTX 850, 4GB
  - RAM: SODIMM, 16 GB, DDR3, 1600 MHz
  - OS: Windows 10 Home edition

The used programming language is Python 3.6.9.

C. *Selecting the best performing model*

In order to perform meaningful experiments in the following sections, we first need to train a network with a proper performance on the proposed dataset. The aim of this section is to find the optimal architecture and settings that will be used in the visualization experiment. For this purpose, the following architectures have been evaluated:

- U-Net 128: A U-net architecture with 64 filter in the last convolutional layer and 7 downsampling blocks, so that an image of size 128x128 pixels becomes as small as 1x1 pixels in the bottleneck. This architecture is the one that is used originally in the CycleGAN implementation.
- U-Net shallow: A U-net architecture with 64 filter in the last convolutional layer and 3 downsampling blocks, so that an image of size 128x128 pixels becomes as small as 16x16 pixels in the bottleneck.
- U-Net mini: A U-net architecture with 4 filter in the last convolutional layer and 7 downsampling blocks, so that an image of size 128x128 pixels becomes as small as 1x1 pixels in the bottleneck.

For all the experiments, unless a different setting is specified, we used a batch size of 1, following the same practice as in [1].

1) Effectiveness of larger datasets: In the previous experiments, we used a dataset consisting of 400 images in total. It order to check whether bigger datasets improve the results of the models trained on them, we trained the best performing model on a dataset that consists of 1200 images in total (300 circles and 300 triangles in both domains). Surprisingly, the use of more instances achieves a worse result than the one with a smaller dataset. Instead of helping the generalization, it triggers overfitting. The evolution of the losses of this experiment is shown in the Fig. 15, and it can be observed that the validation losses increase through the last iterations. The same experiment was conducted using a batch size of 25, and the results are similar. In Fig. 16, the losses evolution during training is shown. In the light of the obtained results, we can conclude that a larger dataset does not provide better results; on the contrary, it produces overfitting. For this reason, the following experiments will be carried out using the small dataset with 400 images.

2) Analysis of data overfitting: In order to check how fast the aforementioned models overfit to the training data, we plotted the losses of discriminator networks and generator networks, as well as cycle-consistency losses on both training and validation datasets. The results are given on the Fig. 12, Fig. 13, Fig. 14, Fig. 15 and 16. The conclusions obtained from these figures are that some settings like a batch of size 25 or a large dataset produce overfitting, as can be observed in Fig. 15 and 16.

To quantitatively compare the performance of different models, we evaluated them on a previously unused test dataset of 160 images in both domains: A and B. The summary of the losses of all evaluated models is presented in the Table I. As can be seen from the table, the very first architecture, i.e., U-Net 128, performs better than the others. That is why, we choose this model for the further experiments.

D. Nullification of different parts of the network

In this experiment, the objective is to understand what information is encoded in each part of the network by disabling or isolating parts of the network. For this, we used the test interface, where the functionality of disabling filters is implemented. For understanding the role of the bottleneck in the colorization task, the first experiment that was conducted was the nullification of all the responses in the bottleneck. What we observed in the output of this modified network is that there is not a big difference with the original, so it can be inferred that the bottleneck does not affect substantially the colorization process. Two examples of this experiment can be seen in Fig. 8.

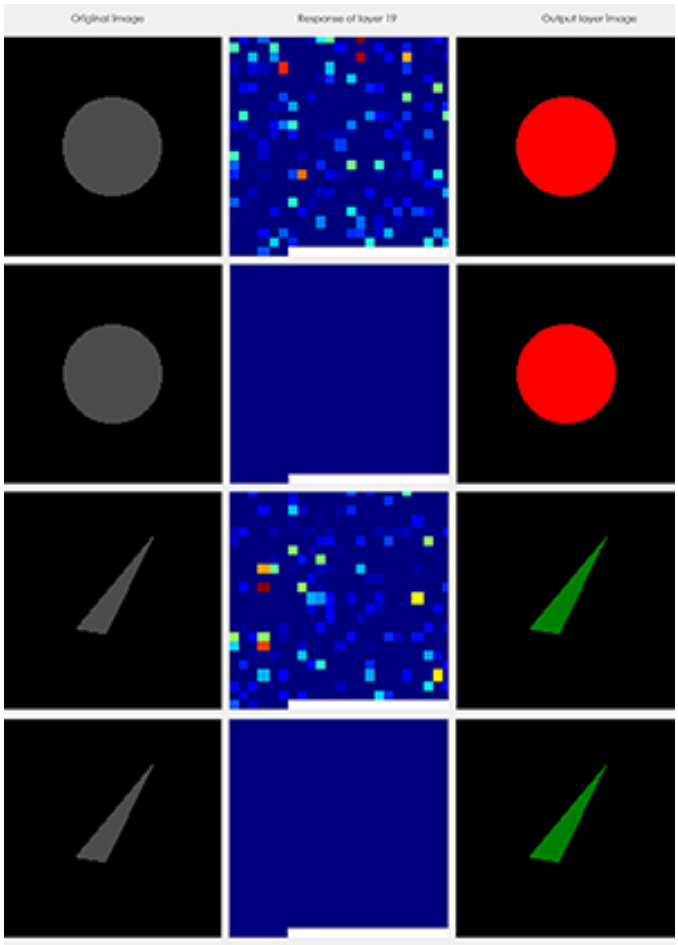


Fig. 8: Results of the bottleneck nullification experiment. On the left, the input images. In the middle, the responses of the bottleneck before (up) and after (down). On the right, the output of each model. It can be seen in the figure that the consequence of removing all the information in the bottleneck is almost insignificant.

The second change that we tried in this experiment is to isolate the first skip connection of the U-Net, i.e., nullify all the filters of the contractive path and the bottleneck, so that, in the end, the network works as if it only consisted of the two upper blocks from each path. The qualitative results, as it could be expected, suggest that the first stage of the network works as an edge detector, since only the edges of the figures are colorized. In Fig. 9 one example of this experiment can be observed.

E. Response of the network to unseen figures

In this experiment, the objective is to understand what general shapes were learned by the network and how it uses them for colorizing. For this, we introduced figures that the network had never seen before in the training set, such as rectangles with different shapes. What we could see is that the network normally colorizes the rectangles that are close to be a square in red —as circles— and the narrow rectangles, with one side much bigger than the other, in green —as triangles—. This results suggest that the network learned

	D_A	D_B	G_A	G_B	Cycle_A	Cycle_B	Total loss
U-Net 128	1.553	1.838	1.892	1.729	9.950	10.261	27.22
U-Net shallow	2.650	1.920	2.166	2.368	9.876	9.649	28.63
U-Net mini	1.511	3.103	1.513	3.813	9.460	9.721	29.12
U-Net 128 trained with batch size of 25	1.518	2.313	1.630	3.122	9.918	10.201	28.70
U-Net 128 trained on a big dataset	2.196	2.186	2.996	1.446	9.873	9.470	28.17
U-Net 128 trained on a big dataset with batch size of 25	2.171	2.855	2.209	4.261	10.094	9.143	30.73

TABLE I: Discriminator losses (D_A, D_B), generator losses (G_A, G_B), Cycle losses (Cycle_A, Cycle_B) and total loss for 6 versions of a U-Net generator, evaluated on a test dataset.

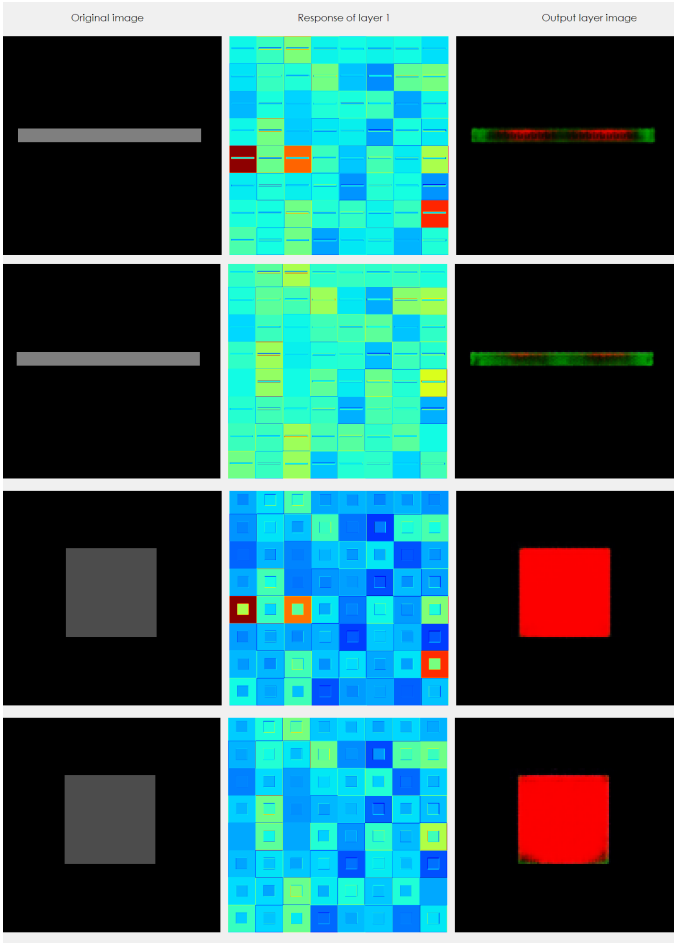


Fig. 9: Results of the experiment with unseen figures. On the left, the input images. In the middle, the responses of the first layer. On the right, the output of each model. In the rows 2 and 4, we also removed the responses from layer 1 that were more reactive to the background color.

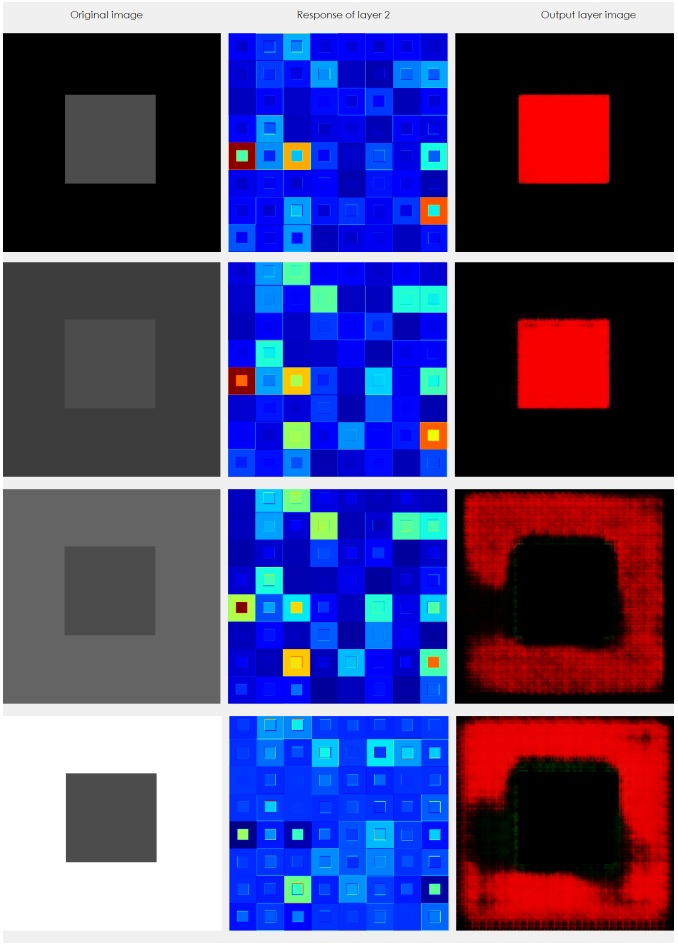


Fig. 10: On the left, the input images with different levels of gray for the background. In the middle, the responses of the layer 2. On the right, the output of each model.

the proportional relation in the shape of the given object, considering red the objects with a more concentrated area —like circles or squares— and green the objects whose shape is not concentrated in its center —like triangles or narrow rectangles. Apart from the aforementioned quality, we also realized that, while the squares are normally colorized in red, their corners are usually colorized in green. From this, it can be deduced that the network learned that the figures with corners should be colorized in green, just like the triangles.

F. Response to the change of background

In this experiment we wanted to check the effect of using a black background through all the dataset. For this, we changed the value of the background from 0 to higher values in the gray level scale. What we found is that, while the colorization still works when the gray level of the background is lower than the one in the figure (even if the shape of the figure that was colorized presents a worse quality), it stops working

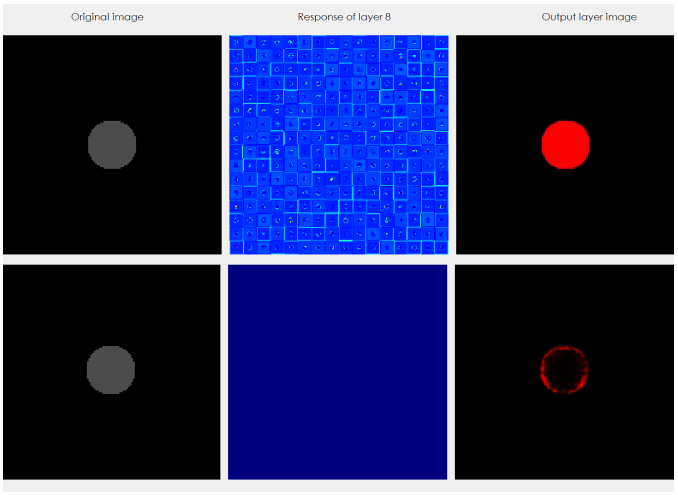


Fig. 11: Results of the first skip connection isolation experiment. On the left, the input images. In the middle, the responses of the filters on one of the nullified layers before (up) and after (down). On the right, the output of each model.

properly when the value of the background is higher. The conclusion extracted from this is that the network just learned that the lighter shade of gray corresponds with the object that is to be colorized and the darker shade corresponds with the background, i.e., with black. A good option for the future would be to include different backgrounds in the toy dataset to avoid this kind of biases.

VI. Results

VII. Conclusions and future work

The results in this paper suggest the following main conclusions:

- In this work, we provide a tool that can be useful for the analysis of CycleGANs and the interpretation of the functioning of the inside of a generator, specifically of a U-Net based one.
- We performed a series of experiments that shed light on the performance of the toy model that we built. These experiments can be seen as examples of the potential applications of the interface that was created.
- The analysis of more complex models is not straight forward, but this work can be an starting step for a greater work.

The prospects for the further work include the following:

- The interface that we created in this work should be extended to support the architecture that we proposed in our previous work, in order to perform the same kind of experiments on this more complex architecture and obtain some conclusions regarding its behaviour. Additionally, it would be interesting to adapt the interface to other architectures apart from U-Net, such as ResNet.
- Other shapes, colors and modalities of data augmentation should be added to the tool for the generation of toy datasets in order to allow other experiments. Especially, an option for different backgrounds would be very useful to avoid the high dependency of the training with the background color.
- In the future, the created interface can be extended in a way that allows to not juts nullify the filter responses, but change their value to any desirable number.
- The method that was used for the visualization of the filters could be substituted for a more sophisticated one, like the ones that were presented in the state-of-the-art.

References

[1] Zhu, Jun-Yan, et al. ”Unpaired image-to-image translation using cycle-consistent adversarial networks.” Proceedings of the IEEE international conference on computer vision. 2017.
[2] Lucic, Mario, et al. ”Are gans created equal? a large-scale study.” Advances in neural information processing systems. 2018.
[3] Berthelot, David, Thomas Schumm, and Luke Metz. ”Began: Boundary equilibrium generative adversarial networks.” arXiv preprint arXiv:1703.10717 (2017).
[4] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. ”Wasserstein gan.” arXiv preprint arXiv:1701.07875 (2017).
[5] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. ”U-net: Convolutional networks for biomedical image segmentation.” International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.
[6] Olah, Chris, Alexander Mordvintsev, and Ludwig Schubert. ”Feature visualization.” Distill 2.11 (2017): e7.
[7] Erhan, Dumitru, et al. ”Visualizing higher-layer features of a deep network.” University of Montreal 1341.3 (2009): 1.
[8] Zeiler, Matthew D., and Rob Fergus. ”Visualizing and understanding convolutional networks.” European conference on computer vision. Springer, Cham, 2014.
[9] Isola, Phillip, et al. ”Image-to-image translation with conditional adversarial networks.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
[10] Goodfellow, Ian, et al. ”Generative adversarial nets.” Advances in neural information processing systems. 2014.
[11] Radford, Alec, Luke Metz, and Soumith Chintala. ”Unsupervised representation learning with deep convolutional generative adversarial networks.” arXiv preprint arXiv:1511.06434 (2015).
[12] Zhao, Junbo, Michael Mathieu, and Yann LeCun. ”Energy-based Generative Adversarial Networks.” (2016).
[13] Salimans, Tim, et al. ”Improved techniques for training GANs.” Proceedings of the 30th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2016.
[14] Wang, Zhou, et al. ”Image quality assessment: from error visibility to structural similarity.” IEEE transactions on image processing 13.4 (2004): 600-612.
[15] Cordts, Marius, et al. ”The cityscapes dataset for semantic urban scene understanding.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
[16] Borji, Ali. ”Pros and cons of gan evaluation measures.” Computer Vision and Image Understanding 179 (2019): 41-65.

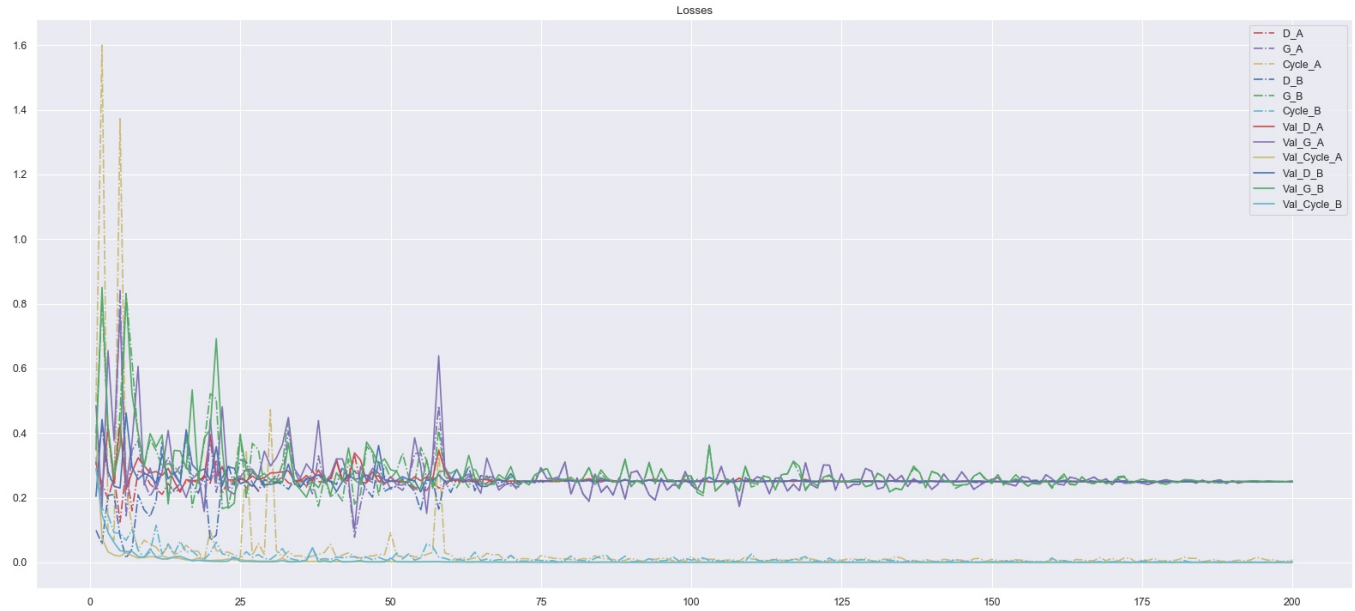


Fig. 12: Example of low - to zero overfitting during the training of a CycleGAN with a U-Net network (7 downsamplings from 128x128 px to 1x1 px, 64 filters in the last convolutional layer) as a generator, using batch size of 1. The size of the training dataset is 400 images (100 circles and 100 triangles, both in RGB and Gray scale domains)

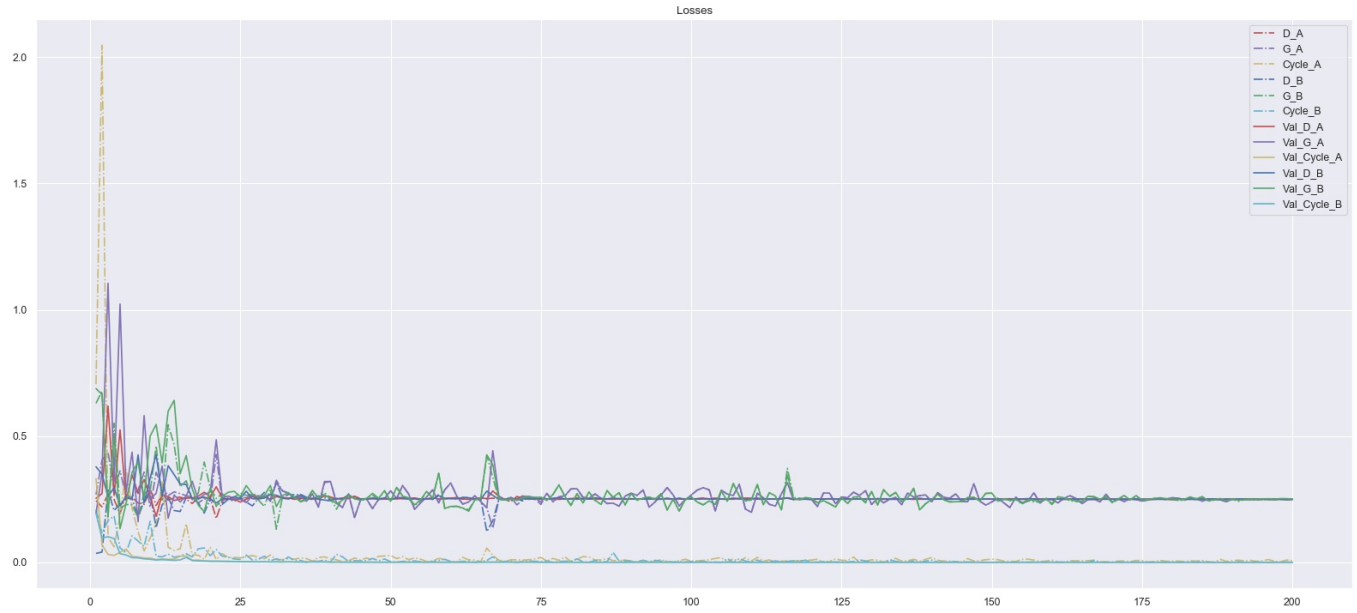


Fig. 13: Example of low - to zero overfitting during the training of a CycleGAN with a U-Net network (3 downsamplings from 128x128 px to 1x1 px, 64 filters in the last convolutional layer) as a generator, using batch size of 1. The size of the training dataset is 400 images (100 circles and 100 triangles, both in RGB and Gray scale domains)

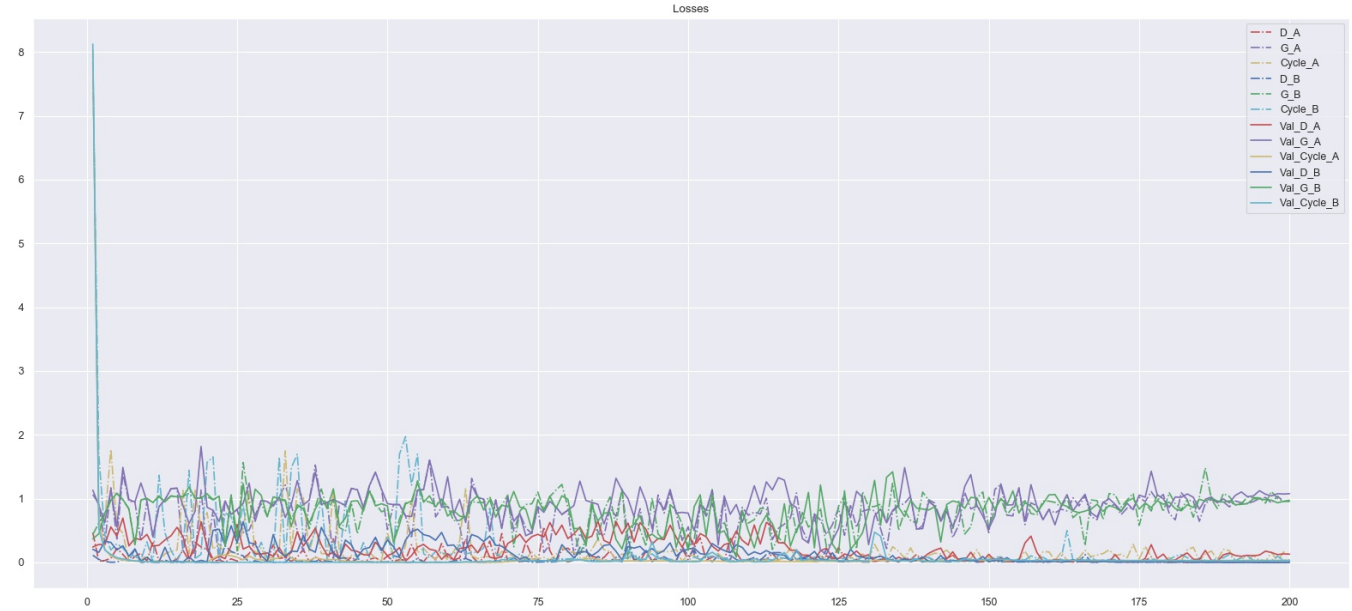


Fig. 14: Example of low - to zero overfitting during the training of a CycleGAN with a U-Net network (7 downsamplings from 128x128 px to 1x1 px, 4 filters in the last convolutional layer) as a generator, using batch size of 1. The size of the training dataset is 400 images (100 circles and 100 triangles, both in RGB and Gray scale domains)

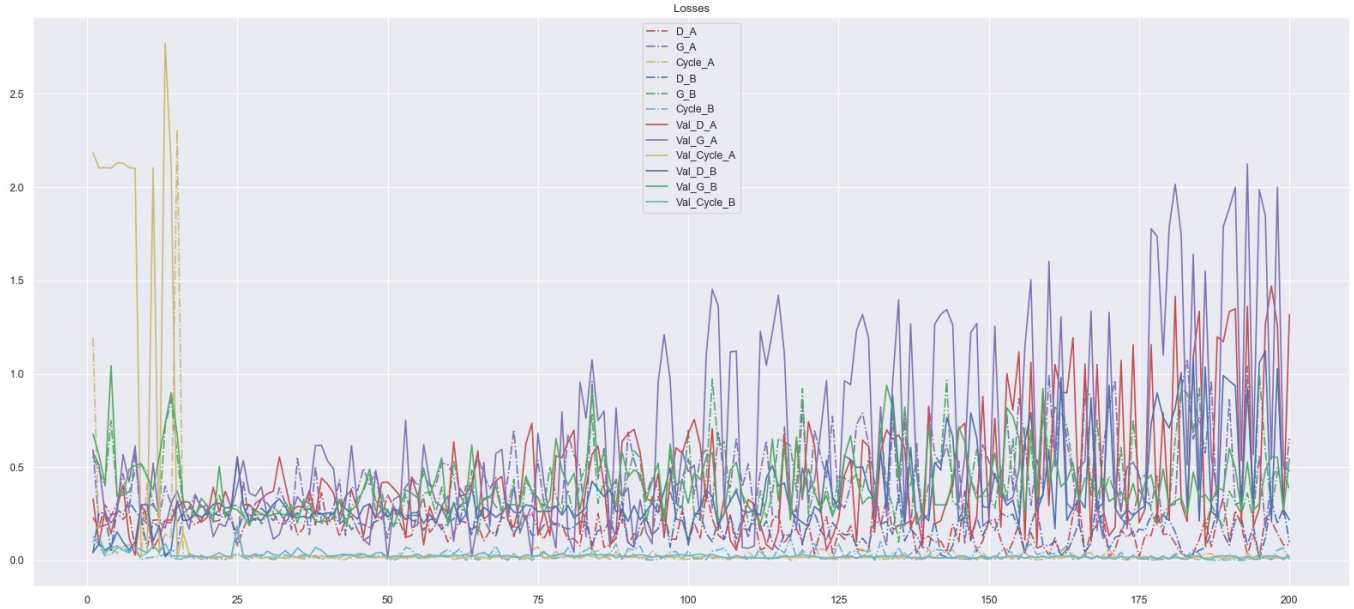


Fig. 15: Example of high overfitting during the training of a CycleGAN with a U-Net network (7 downsamplings from 128x128 px to 1x1 px, 4 filters in the last convolutional layer) as a generator, using batch size of 1. The size of the training dataset is 1200 images (300 circles and 300 triangles, both in RGB and Gray scale domains)

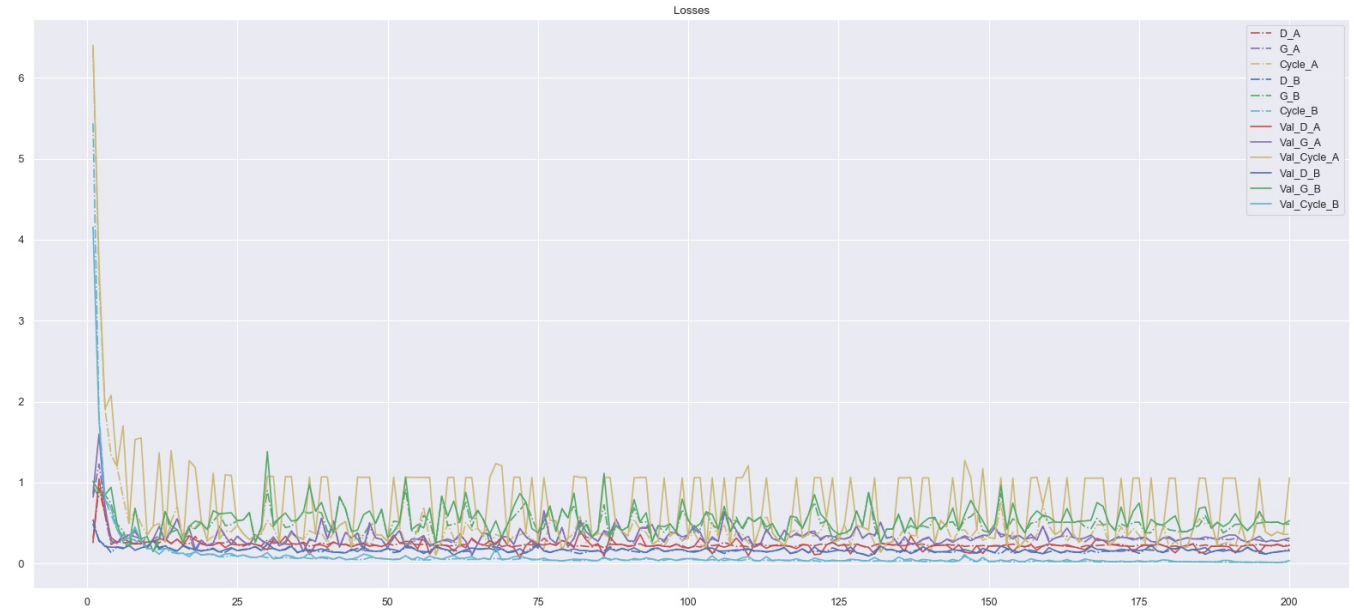


Fig. 16: Example of high overfitting during the training of a CycleGAN with a U-Net network (7 downsamplings from 128x128 px to 1x1 px, 4 filters in the last convolutional layer) as a generator, using batch size of 25. The size of the training dataset is 1200 images (100 circles and 100 triangles, both in RGB and Gray scale domains)