Reimplementation of a Cycle-Consistent Adversarial Network for unpaired Image-to-Image Translation

Fabian Imkenberg, Jacqueline Naether

Abstract

In this project, we reimplement the paper "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" and test its performance with three different datasets, two given by the paper and the third one downloaded from Kaggle. After the reimplementation of the Cycle Generative Adversarial Network (GAN) including minor justified changes, two modifications are applied. This results in three variations of CycleGANs: Classical CycleGAN, Regularized CycleGAN and Cycle Wasserstein GAN (WGAN). After the implementation, the performance of the CycleGAN variations on the different datasets is analyzed. Here it becomes clear that the CycleGAN already shows a good progress after just a few training steps, where changing colors is learned much faster and easier than more advanced concepts like adapting the contour of eyes.

Keywords

CycleGAN — Unpaired Image-to-Image Translation — CycleWGAN – Regularized CycleGAN

Contents

Introduction

As the basis of this project, the paper *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* from the Berkeley AI Research (BAIR) laboratory is studied. The aim of the paper is to change different information in images. The developed network creates a generated image based on an existing input image. Three different datasets for image generation are presented in this paper. A translation from a photo to a painting by Monet, from zebras to horses, and from summer to winter and vice versa in each case. The paper complements what has been learned in the module Implementing Artificial Neural Networks with Tensorflow (IANNWTF) by two essential building blocks that are basically interdependent. [1]

Until now, training data was mostly available in input-output pairs. If one takes the translation from summer to winter as an example, this means that for every photo of a location in summer, there also exists a photo of the same location in winter. Thus, the network would learn the translation based on one-to-one examples. In contrast, in case of the paper, the network is trained based on two domains. So, the network learns the different properties of summer and winter based on random images belonging to one of the two domains. As a side effect, this significantly facilitates the acquisition of suitable datasets, since no direct correlation between the images of the two domains is necessary. [1]

Furthermore, the GAN as introduced in IANNWTF, is extended to a so called CycleGAN. The name is motivated by the use of cycle consistency loss in the training phase.

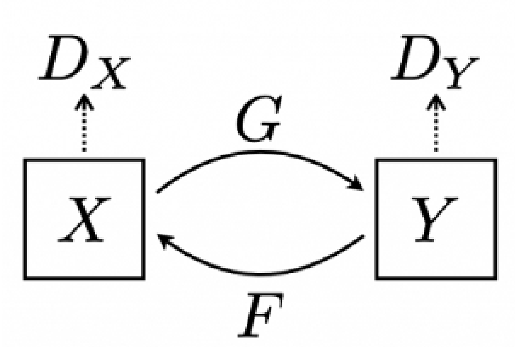


Figure 1. Mapping functions of the CycleGAN [1]

In this paper, the CycleGAN architecture, based on [1], will be reimplemented and tested to obtain a more in-depth understanding of its benefits and drawbacks. Additionally, different variations of the classical CycleGAN are analysed and their performances on different datasets compared.

1. Methodology

The architecture described in this paper should be based on the classical CycleGAN approach, but must also enable some modifications for the implementation of variations. So, it has to work for different datasets and needs to have a well-organized structure that allows the architecture to be modified while maintaining a clear and understandable network design. Therefore, amomg other things, the methodological basics of the architecture from the paper have to be worked out. In the following, these basics will be explained.

1.1 Related approaches

First, an overview of related work for this paper is provided.

Generative Adversarial Networks (GAN) basically consist of two neural networks which more or less compete with each other, the *Generator* and the *Discriminator*. The generator takes the input, often provided in form of an encoding, and generates a fake image from it, while the discriminator tries to discriminate between generated and real images. Both parts of the network are trained individually from each other. The loss used in training is the adversarial loss. [2, 3]

CycleGANs extend this concept, in terms of unpaired image-to-image translation. As mentioned in the introduction, the chosen paper uses domains of input images D_x and output images D_y. The Generator is trained by two main functions: Generating images from domain D_x to images from domain D_y ($G : X \to Y$) and vice versa ($F : Y \to X$). This principle is shown in Figure 1. [1]

From this, it can be inferred that a simple for- and backward translation may result in a back-translation that does not correspond to the original input image, but to a different image within the input image domain. To address this issue, cycle consistency loss is introduced. It is used in addition to the adversarial loss so that the network learns to return to the original image after a for- and backward translation. The cycle consistency losses for the two mapping functions are shown in Figure 2. [1]

Cycle Wasserstein GANs are an improvement of the classical CycleGAN. This type is introduced due to the frequent training instabilites of CycleGANs. CycleWGANs aim to overcome this issue by introducing the Wasserstein distance which has better continuity and differentiability properties in contrast to local saturated loss functions. Thus, the approach is intended to reduce the problem of vanishing gradients. [4].

1.2 Formulation

For a better understanding of the function and the subsequent implementation of the losses, the mathematical background is explained in more detail.

Adversarial Loss is already used in the training of GANs. It arises from the fact that the generator G and discriminator D compete with each other. If the generator creates particularly good fake images which the discriminator can no longer identify as such, the discriminator's loss automatically increases. On the other hand, the loss of the generator decreases. This behaviour is the same the other way round. It is mathematically described in [5] as follows:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] \\ + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Cycle Consistency Loss is used in addition to the adversarial loss in the CycleGAN. As briefly explained before, a generator G translates an image *Input_A* from domain X to an image *Generated_B* from domain Y. In the backward translation of the generator F, an image *Cyclic_A* from domain X will be created using image *Generated_B* from domain Y. The difference between image *Input_A* and *Cyclic_A* forms the cycle consistency loss [2]. In Figure 2 the cycle consistency loss is shown as well as it is described in [2] mathematically as follows:

$$Loss_{cyc}(G,F,X,Y) = \frac{1}{m} \sum_{i=1}^{m} [F(G(x_i)) - x_i] + [G(F(y_i)) - y_i]$$

In summary, the CycleGAN is composed of two generators, *Generator A2B* and *Generator B2A*, as well as two discriminators, *Discriminator A* and *Discriminator B*. This architecture is shown in simplified form in Figure 3.

Identity Loss is also introduced in paper [1]. The loss can be seen as a regularization technique for encouraging the generator to preserve the color composition between the input- and the output image. This is done by providing the generator real images from the target domain, with the goal to not translate them. Mathematically, the identity loss is defined as follows [1]:

$$Loss_{identity}(G,F) = \mathbb{E}_{y \sim p_{data}(y)}[||G(y) - y||_1] \\ + \mathbb{E}_{x \sim p_{data}(x)}[||F(x) - x||_1]$$

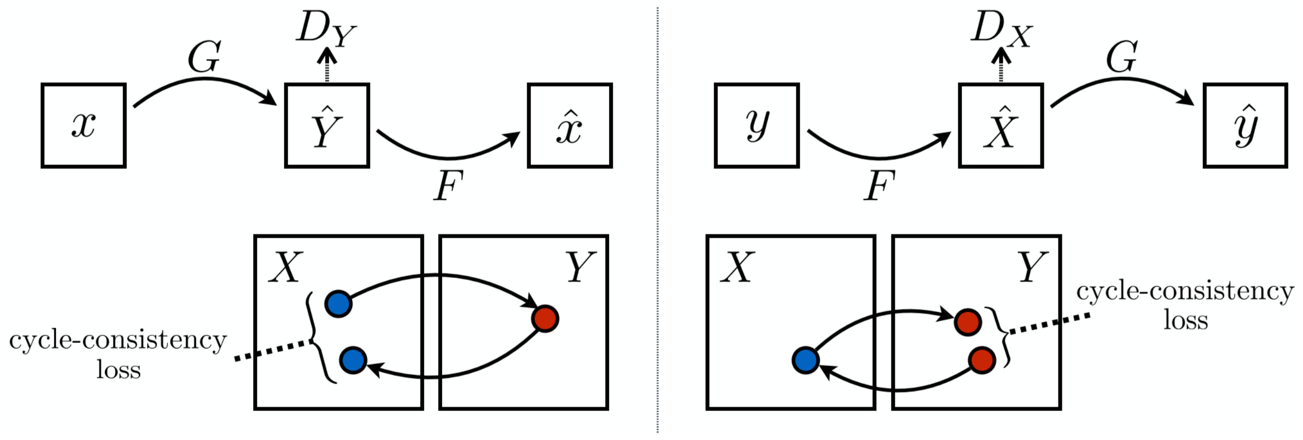


Figure 2. Cycle Consistency Losses in the for- and backward translation [1]

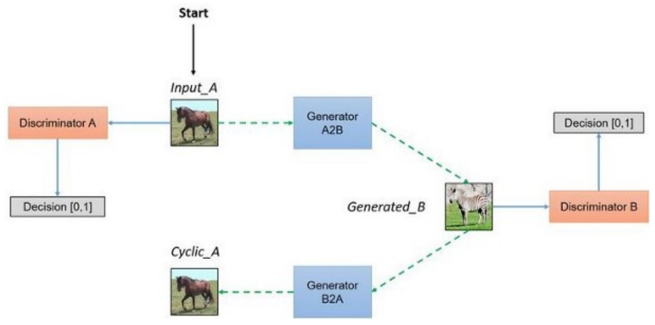


Figure 3. A simplified architecture of a CycleGAN [2]

1.3 Datasets

In addition to the architecture, suitable datasets must be selected. These should be versatile and contain images that are suitable for the classification into two domains.

Tensorflow provides different datasets specifically suited for CycleGANs including for example apple and orange domains as well as photo and Monet painting domains. These are called *cycle_gan/apple2orange* and *cycle_gan/monet2photo*. The datasets are divided into training and test datasets. Table 1 shows the four parts of the standard configuration with the number of examples included in each dataset. Every example consist of image-label pairs. [6]

Table 1. Structure of the standard configuration split of the Tensorflow datasets *cycle_gan/apple2orange* and *cycle_gan/monet2photo* [6]

index	examples `apple2orange`	examples `monet2photo`
'testA'	266	121
'testB'	248	751
'trainA'	995	1072
'trainB'	1019	6287

Kaggle provides another dataset called *arnaud58/selfie2-anime*. This one is selected to analyze the performance of the CycleGAN on a dataset which is not part of the classical CycleGAN paper [1]. It contains images of animes and women selfies. The number of examples included in this dataset are shown in Table 2. [7]

Table 2. Structure of the standard configuration split of the Kaggle dataset *arnaud58/selfie2anime* [7]

index	examples `selfie2anime`
'testA'	100
'testB'	100
'trainA'	3400
'trainB'	3400

2. Implementation

The implementation is done with the learned three steps from the module IANNWTF: pre-processing, training and test [8, 9]. In addition, the methods mentioned in the section before are used to develop the CycleGAN architecture. Furthermore, functions of the Tensorflow and Keras libraries are utilized in nearly every part of the implementation. The realization of the implementation and use of the various libraries will now be explained in more detail.

2.1 Network Architecture

The architecture is built up as explained in the section before, with implementing two classes `Cycle_GAN_Generator` and `Cycle_GAN_Discriminator`. In the following, two instances of each class are created, `generator_A_to_B` and `generator_B_to_A` as well as `discriminator_A` and `discriminator_B`.

Generator Class inherits from the Keras class `Layer` and is build up of three parts, the *Encoding*, the *Transformation* and the *Decoding*. Each part again consists of different layers. The whole structure of the generator is shown in Figure 4. [2]

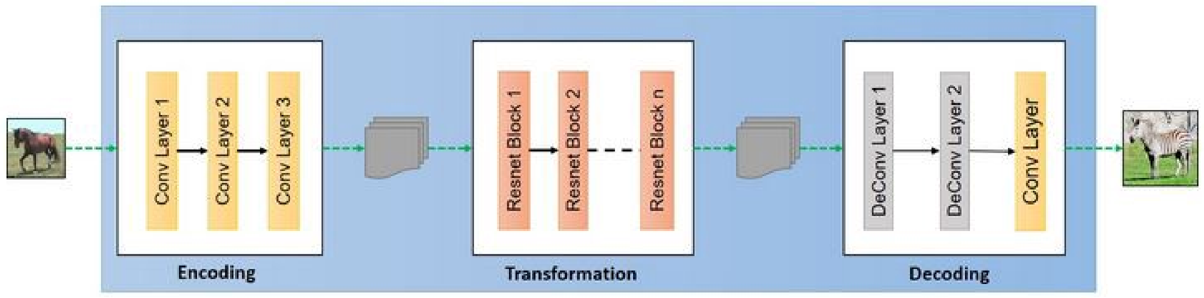


Figure 4. The high-level structure of the CycleGAN's generator [2]

The **Encoding** extracts the features of the original input image. In `Cycle_GAN_Generator`, the encoding consists of exactly three convolutional layers, as depicted in Figure 4. The Keras class `Conv2D` is used for this. As activation function Rectified Linear Units (ReLu) is selected, using the Keras class `ReLu`. Furthermore, the Keras class `InstanceNormalization` is utilized to enable this special form of group normalization, see [10] for more details. Instance Normalization is well suited for datasets with batch size 1, such as recommended in the paper. The filter sizes of each `Conv2D` layer are quartered to reduce the complexity and enable a faster training of the CycleGAN. All other relevant arguments like `kernel_size`, `strides`, `activation`, `padding` and the `kernel_initializer` are given by the paper. The specifications and arguments for the three `Conv2D` layers are shown in Table 3. [1]

Table 3. Values of the convolutional layer arguments in the encoding part of the class `Cycle_GAN_Generator` [1].

variable	Conv2D Layer 1	Conv2D Layer 2	Conv2D Layer 3
filters	16	32	64
kernel_size	(7,7)	(3,3)	(3,3)
strides	(1,1)	(2,2)	(2,2)
activation	None	None	None
padding	"same"	"same"	"same"
kernel_initializer	Random Normal	Random Normal	Random Normal

The **Transformation** is needed to relate closely spaced features extracted from the input image by the previous encoding [2]. This is exactly what the blocks of Residual Networks (ResNet), like shown in Figure 4, are needed for. Every ResNet is build up by two `Conv2D` layers, two layers of `InstanceNormalization`, a ReLu activation layer and a layer `Concatenate` which is also a Keras class. A new class `ResNet_Block`, which also inherits from the Keras class `Layer`, is implemented. All arguments for the layers are given by the paper. The convolutional layer arguments are shown in Table 4.

Instances of this class are created as a layer in the class `Cycle_GAN_Generator`. The number of instances depends

Table 4. Values of the convolutional layer arguments in the ResNet block (part of the transformation part of the class `Cycle_GAN_Generator` [1].

variable	Conv2D Layer 1	Conv2D Layer 2
filters	256	256
kernel_size	(3,3)	(3,3)
strides	(1,1)	(1,1)
activation	None	None
padding	"same"	"same"
kernel_initializer	Random Normal	Random Normal

on the instance variable `n_resnet`, which is set to six by default. Six ResNet blocks are used if the image size is *128 x 128*, nine blocks are use for an image size of *256 x 256* or higher [1].

The **Decoding** now leads back to an output image. Therefore, the low-level features are worked out with the help of the transpose convolution, see [2]. To achieve this, the Keras class `Conv2DTranspose` is used in this part. Finally, as shown in Figure 4, another `Conv2D` is used, in conjunction with an activation function of the Keras class `tanh`. The values of the transposed convolution layer arguments can be found in Table 5 and of the convolutional layer arguments in Table 6.

Table 5. Values of the transposed convolutional layer arguments in the decoding part of the class `Cycle_GAN_Generator` [1].

variable	Conv2D Transpose Layer 1	Conv2D Transpose Layer 2
filters	32	16
kernel_size	(3,3)	(3,3)
strides	(2,2)	(2,2)
activation	None	None
padding	"same"	"same"
kernel_initializer	Random Normal	Random Normal

Table 6. Values of the convolutional layer arguments in the decoding part of the class `Cycle_GAN_Generator` [1].

variable	Conv2D Layer 4
filters	3
kernel_size	(7,7)
strides	(1,1)
activation	None
padding	”same”
kernel_ initializer	Random Normal

Discriminator Class also inherits from the Keras class `Layer`. In this reimplementation, the class is a modified discriminator class, a so called *PatchGAN*. This one is made up to not only output a single value, but a one-channel feature map of predictions. In this case the output refers to a *70 x 70* receptive field of the original input. Since it has an image as input and a decision vector as output, `Conv2D` layers are associated with `InstanceNormalization` and the activation function `LeakyReLu`. The arguments of the layers are given by the paper and shown in Table 7. The original `filters` size is here again divided by four to reduce the training time. After the first convolutional layer, they deliberately use no `InstanceNormalization`. The last layer is a convolution to produce an output of dimension 1. The structure of the discriminator is also shown in Figure 5. [2]

Table 7. Values of the convolutional layer arguments in the discriminator class `Cycle_GAN_Discriminator`[1].

variable	Conv2D Layer 1	Conv2D Layer 2	Conv2D Layer 3
filters	16	32	64
kernel_size	(4,4)	(4,4)	(4,4)
strides	(2,2)	(2,2)	(2,2)
activation	None	None	None
padding	”same”	”same”	”same”
kernel_ initializer	Random Normal	Random Normal	Random Normal

variable	Conv2D Layer 4	Conv2D OutputLayer
filters	128	1
kernel_size	(4,4)	(4,4)
strides	(2,2)	(1,1)
activation	None	sigmoid
padding	”same”	”same”
kernel_ initializer	Random Normal	Random Normal

2.2 Pre-Processing

In addition to creating the network architecture, the dataset must also be pre-processed. In order to implement easier switching between datasets and comparison versions different mechanisms are included. For switching between datasets, additional variables (`is_selfie2anime_dataset`, `is_apple2orange_dataset`, `is_monet2photo_dataset`) are implemented to select the dataset to be pre-processed. At the beginning, these variables are used to select which dataset is loaded, by setting it `true`. Only one dataset can be loaded each time.

Additionally, the dataset size is set to 900 training samples and 100 test samples after pre-processing to have a comparable number of images for the various datasets.

Moreover, since two different sources for the three datasets are used, there are a some different steps for pre-processing between the Tensorflow and Kaggle datasets.

Tensorflow datasets can be loaded more easily with the `tfds.load` function from the tensorflow library. In general there is more or less no difference between pre-processing *cylce_gan/apple2orange* or *cylce_gan/monet2photo* images. When the dataset is loaded, it is also split into test and training data, as well as into two domains for each, and requested as a 2-tuple structure (input, label). To ensure that all images are in the same format and that the training time is not too high, the images are subsequently resized to *128 x 128*. After normalization to a value between -1 and 1 $(2*(img/255)-1)$ and reshaping the tensor to *128 x 128 x 3*, the dataset is shuffled and prefetched with a `buffer_size` of 128. Batching is omitted, or rather set to one, to facilitate the later definition of real samples in training. All functions used are offered by the Tensorflow library. The values for the arguments of the different pre-processing steps are taken from the paper. [1]

Kaggle datasets need a few more pre-processing steps than Tensorflow ones. The dataset *arnaud58/selfie2anime* is downloaded with Keras function `image_dataset_from_directory` and directly batched to one. Because the dataset comes up as one big file, in which the dataset is separated in the different classes mentioned in Table 2, see [7], it then needs to be split manually. This is done by using the functions `take` and `skip`. Thereupon, the same steps are carried out as for the Tensorflow datasets mentioned before.

2.3 Training

Three training epochs are chosen with each epoch having 900 training steps. With these numbers, the training can still be done in a reasonable amount of time. To make the training clearer, various functions have been programmed for different steps in the execution of the program.

In the first step the generators are trained. The training starts with taking an image-label pair from domain A and from domain B with the help of the function `generate_real_dataset`. These pairs are again randomly taken from the domains for each training step. Afterwards the training of the `generator_A_to_B` and the `generator_B_to_A` takes place. For this purpose, the function `generator_train_step` is programmed, in which the adversarial loss

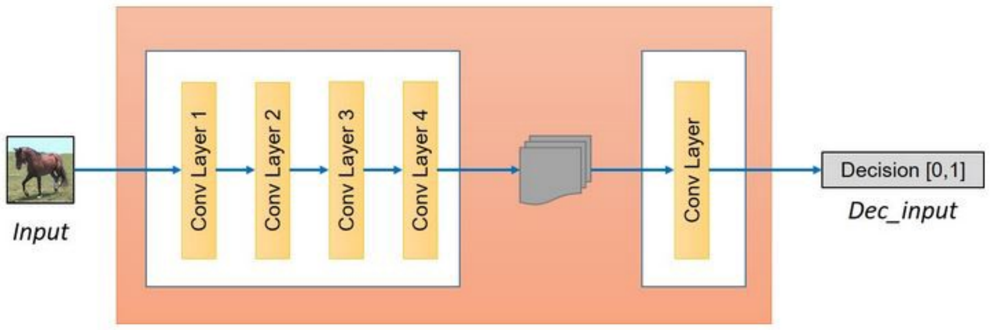


Figure 5. The high-level structure of the CycleGAN's discriminator [2]

(`adv_loss`), the identity loss (`id_loss`), the cycle consistency loss for the forward (`for_loss`) and backward cycle (`back_loss`) and the entire training loss (`train_loss`) is computed. The training loss is composed of all other losses as follows:

$$\begin{aligned} train\_loss = &(1 * adv\_loss) \\ &+ (10 * (for\_loss + back\_loss)) \\ &+ (5 * id\_loss) \end{aligned}$$

In the papers implementation, the identity loss is only used for the dataset *cylce_gan/monet2photo*. For better comparability between the different datasets, it is used for all datasets in the reimplementation. The training loss is then utilized to update the generator parameters. For updating the gradients of the generators, the loss function *MeanAbsoluteError* is used. For updating the generator parameters, the optimizer *Adam* is used. [1, 11, 12]

In the second step of each training step, the discriminators are trained. Since the discriminator has to identify the output image of the generator, the image from the selected image-label pair of the domains A and B is fed to the respective generator before training and the result image is returned (function: `generate_fake_dataset`). This image is now stored in a history buffer. The function `update_image_pool` stores the image in the buffer and returns it at the same time, if the pool is not filled with 50 images yet. As soon as the pool is filled with 50 images, a random image from the buffer is replaced by the current training image when the function is called. The replaced image is then given to the discriminator as input. It is possible that the discriminator gets an image, which makes it easier to distinguish a generated image from a real image, because the image is generated by a worse generator version. This prevents the oscillation of GANs [13]. With the passed image a forward step is performed and the loss is calculated as *MeanSquaredError*. The loss is divided by two when updating the gradients to reduce the learning rate of the discriminator. The parameter update is also performed with the Optimizer Adam. [1, 11, 12]

To be able to document the training progress, the losses are averaged over the last 150 training steps and saved in a list. At the end of the program, these lists are saved in csv files in Google Drive. The evaluation is then carried out with the values from these lists.

2.4 Test

Like during the training, image-label pairs are taken from the test datasets of domain A and domain B. The image-label pairs are then used to generate the test data. Subsequently, the `generator_test_step` and `discriminator_test_step` functions calculate all the different losses like in the training. However, this time the parameters of the generators and discriminators are not updated. The losses are only returned for tracking. For the representation and storage of the results, further functions are programmed in addition to the reimplementation. Like mentioned before, the test losses are saved in csv files in Google Drive, too. After 300 training steps a test is conducted. Each test phase conducts 100 steps. The losses are averaged over these 100 test steps and saved. Moreover, the test and training losses are displayed in the notebook every 450 training steps, so two times per epoch. Additionally, five images are selected from the test dataset for the training at the beginning, which are fed to CycleGAN in each test, i.e., three times per epoch. The original images and the images generated from them are stored in a Google Drive folder structure. This ensures a better comparability between the different ones after different training periods. The folder structure is built up as shown in Figure 6.

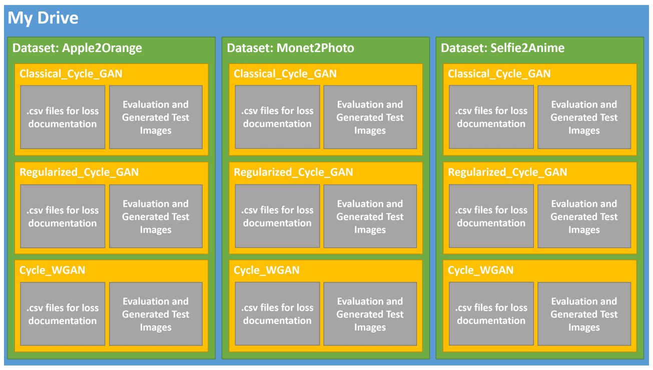

My Drive
Dataset: Apple2Orange
Classical_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Regularized_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Cycle_WGAN
.csv files for loss documentation
Evaluation and Generated Test Images
Dataset: Monet2Photo
Classical_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Regularized_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Cycle_WGAN
.csv files for loss documentation
Evaluation and Generated Test Images
Dataset: Selfie2Anime
Classical_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Regularized_Cycle_GAN
.csv files for loss documentation
Evaluation and Generated Test Images
Cycle_WGAN
.csv files for loss documentation
Evaluation and Generated Test Images

Figure 6. The folder structure to store the data in Google Drive.

3. Variations of the CycleGAN

After the pure reimplementation of the CycleGAN, two further improvements are implemented. All three variations are then compared with each other. The implementation of the two improved versions is explained in more detail in the following two subsections.

3.1 Regularized CycleGAN

In this reimplementation, a further class for the ResNet block, the `Regularized_ResNet_Block`, is programmed. This contains an additional dropout layer after the first convolutional layer. The dropout rate is assumed to be 0.5.

Since the code should be written lean and collected in one colab notebook, the generator class is passed another argument at initialization, the `Regularized_Cycle_GAN`. If this argument is set to *false*, the default ResNet is used during initialization. If this argument is set to *true*, the CycleGAN is built with regularized ResNet blocks.

Beside the adjustment of the ResNet blocks in the generator, also the discriminator is adjusted. Already when initializing an instance of the discriminator class, the argument `kernel_regularizer` is introduced to the convolutional layers with a L2 regularization. In the classical version of the CycleGAN this is also initialized, but not used. Therefore, the argument `Regularized_Cycle_GAN` is passed to the training step of the discriminator to control the computation of the loss dependent on it. The L2 loss for the discriminator is only added and used for backpropagation if this argument is set to *true*.

3.2 Cycle Wasserstein GAN

Furthermore, another method, the WGAN, is introduced according to [4]. In a WGAN, the optimizer *Root Mean Squared Propagation (RMSProp)* is used instead of *Adam*. In addition, the weights in the discriminator are clipped, which leads to the fact that the discriminator has to be trained five times more often than the generator. Furthermore, the *sigmoid* activation is removed from the output layer of the discriminator. To implement all of this, the new class `Cycle_WGAN_Discriminator` is created.

For including it in the same lean colab notebook, the argument `Cycle_WGAN` is introduced. If the argument is set to *true*, a Cycle-WGAN is created, in the other case a classical CycleGAN. Addtionally, the optimizer is initialized as *RMSProp instead of Adam*.

4. Results and Analysis

In this section, the results are presented and an analysis is carried out. Since not all results can be shown here, only selected analyses are provided. Any other results are available in the associated Github repository [14].

4.1 Training and Test performance

Figure 7 shows a graph including the average training and test losses of the discriminators and generators of a classical CycleGAN. In general, the discriminator losses are about 30 to 40% smaller than the generator losses. Both types show a decreasing trend, whereby this is more evident for the generator than for the discriminator. Furthermore, it can be seen that the test losses for both discriminators fluctuate significantly more than the training losses, whereas the opposite is the case for the generators. At the end of the 2700 training steps (3 epochs), the generator training losses and the discriminator B test loss have a decreasing trend, while all the others have a slightly increasing trend. But since the trend is downward over the entire 3 epochs, these slightly increasing trends will probably only be local maxima in further training.

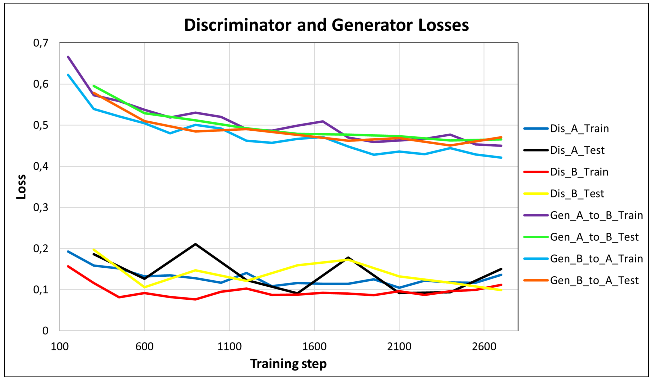


Figure 7. Average discriminator and generator losses during training and test with a classical CycleGAN on the dataset *apple2orange*.

Thus, the training and test losses run in tandem with each other and show a tendency to improve the network architecture during training.

For the dataset *monet2photo*, the discriminator losses are also lower than those of the generators. Furthermore, falling trends can be recognized here as well. In contrast to the dataset *apple2orange*, the discriminator test and training losses diverge more. Overall, the generator losses are about 10% smaller than those of the dataset *apple2orange*.

4.2 Components of the Generator loss

Another interesting point for analyzing is the examination of the various losses of the generator presented earlier. For the dataset *monet2photo*, these are taken from `generator_A_to_B` in Figure 8. Almost all losses are on a downward trend. The adversarial loss is quite noticeable. It has a strong upward tendency after about 1200 training steps which might occur due to the improved quality of the generated images. This is indicated by the overall loss of the generator which maintains the downward trend. Towards the end, the adversarial loss shows a downward trend again. Nevertheless, there is a global maximum at training step 2550.

The adversarial loss in Figure 8 never exceeds the value of the overall loss, as it is the case for the dataset *apple2orange*. Despite this, the other losses show a clear downward trend. Accordingly, it can be said that the network improves during training.

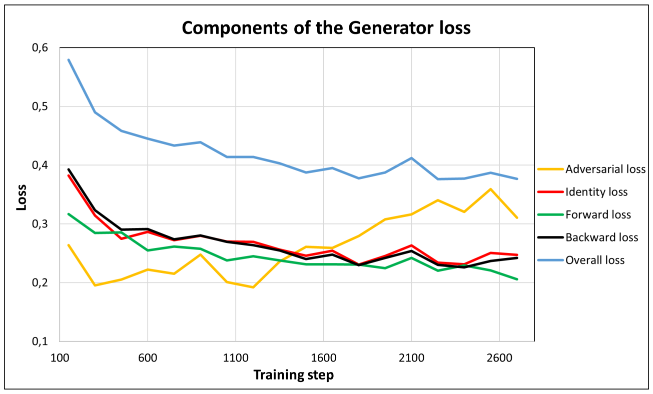


Figure 8. Different Generator losses (`generator_A_to_B`) during training with a classical CycleGAN on the dataset *monet2photo*.

4.3 Comparison of the CycleGAN variations

The different CycleGAN variations show different progressions for their losses, see Figure 9 and Figure 10. While the discriminator losses show only a slight downward trend within the three epochs, this trend can be seen more clearly in the generator losses. While the curve for the CycleWGAN generator loss is smoother, it is more erratic for the discriminator than compared to the other variations. The lowest generator loss is achieved by the classic CycleGAN, whereas this is not the case for the discriminator. This may also be related to the fact that better generated images are more difficult to detect by the discriminator. On the basis of these results, it is not possible to say whether one of the networks works better or worse than another. To do so, a longer training process would be necessary.

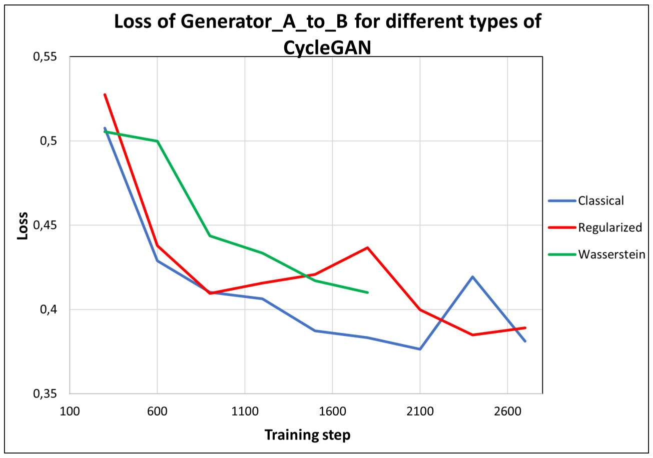


Figure 9. Overall losses during training of `generator_A_to_B` for the different CycleGAN variations on the dataset *monet2photo*.

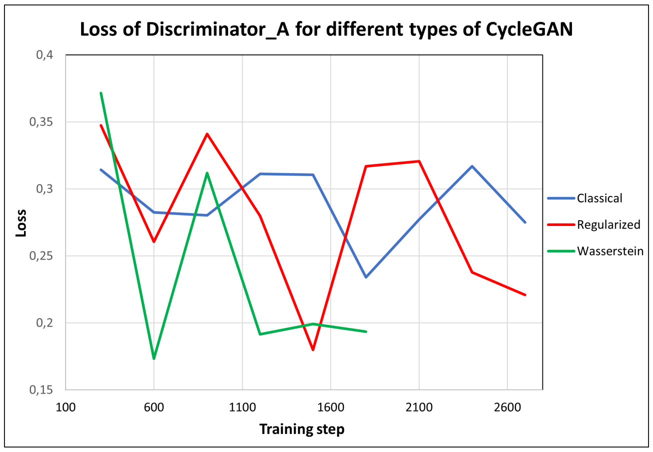
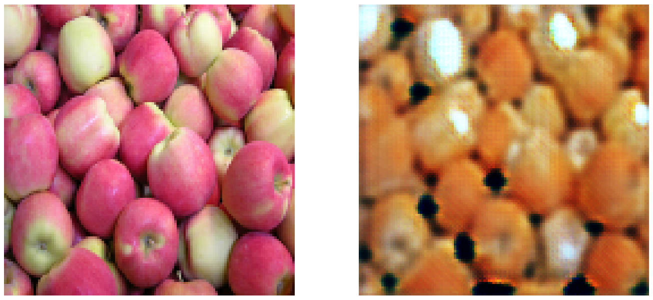


Figure 10. Overall losses during training of `discriminator_A` for the different CycleGAN variations on the dataset *monet2photo*.

Another possibility for evaluation is the qualitative comparison of the generated images. Figure 11 already shows great success in the translation of an apple image to an orange image after three epochs. In comparison, the generated images of the other variations, see Figure 12 and Figure 13, do not look so much like oranges. However, no qualitative statement can be made about whether the generated images of the Regularised CycleGAN or the CylceWGAN are more realistic. The transitions between the individual fruits are quite well visible. Thus, all CycleGAN variations are basically able to recognise and keep object boundaries.



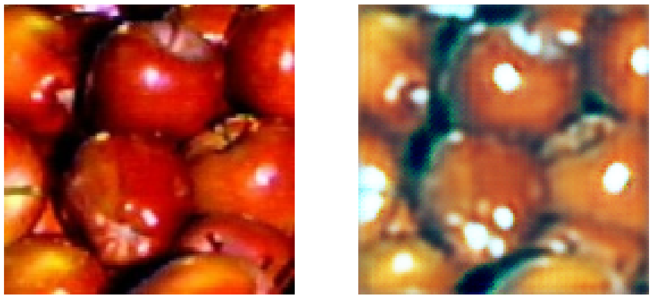
(a) Original example image from domain A **(b)** Generated image after 2700 training steps

Figure 11. Example of an image processed by the classical CycleGAN from domain A to B of dataset *apple2orange*

4.4 Limitations of the CycleGAN architecture

As the training of the *selfie2anime* dataset takes too long for the scope of this project, only a few training steps (300) could be carried out and only one test could be conducted afterwards. The result is shown in Figure 14. The colours are transformed well and quickly, whereas the contours are not. This is quite obvious in the area of the eyes as Anime characters have significantly larger and circular eyes than humans.

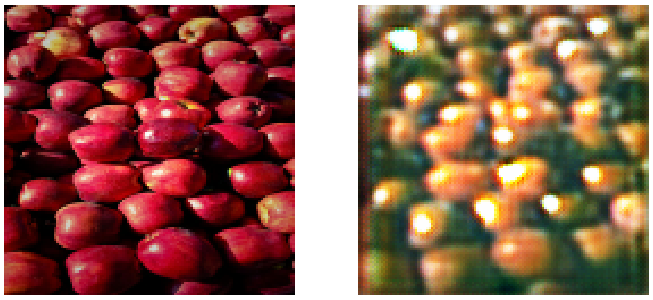
This result is also reflected in the comparison to the *apple2orange* dataset, where the colour of the apples changes quite quickly to orange, but the contours remain almost untouched.



(a) Original example image from domain A

(b) Generated image after 1500 training steps

Figure 12. Example of an image processed by the CycleWGAN from domain A to B of dataset *apple2orange*



(a) Original example image from domain A

(b) Generated image after 2700 training steps

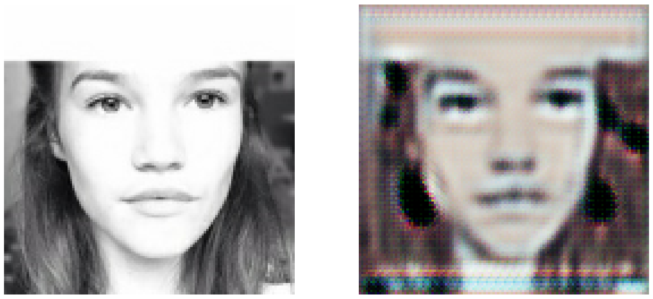
Figure 13. Example of an image processed by the Regularized CycleGAN from domain A to B of dataset *apple2orange*

5. Conclusion and Future work

Unlike previously planned, the analysis of the dataset *selfie2 anime* could not be carried as detailed as for the Tensorflow datasets due to the training process which duration is too long for the scope of this project. Despite a subsequent reduction of the epochs to one, the network needed more than 12 hours. In addition, with Goolge Colab's limitation of GPU time, it is hardly possible to train for such a long period. However, a few results of the first training steps are available. Since these are difficult to compare with the ones from the other datasets, a more detailed analysis is not performed here. Nevertheless, a few interesting features already developed within the first training steps. So it would be interesting to extend this analysis in further projects.

The images of the datasets had to be reduced in size in order to be able to train for more epochs. So the behaviour of the different CycleGAN variations at a higher image resolution is another interesting point to investigate.

Furthermore, it would be possible to extend the tests and results. In order to have an even better comparability of the CycleGAN variations, the same five test images could be used for all variations. A difficulty for the implementation is the design of the notebook as this is always executed new. The five test images would therefore have to be administered



(a) Original example image from domain A

(b) Generated image after 300 training steps

Figure 14. Example of an image processed by the Regularized CycleGAN from domain A to B of dataset *selfie2anime*

externally beforehand in order to always be able to guarantee access to the same images.

Another improvement can be found in the saving of the data. Here, an even more detailed evaluation could take place if the data was saved after each training step. But due to the huge flood of data and the increased computing time, this was not conducted.

In summary, the results of the project are more meaningful than previously expected. Nevertheless, there is much potential in the design and training of the CycleGANs that could not be realised in this project due to time constraints.

References

[1] Jun-Yan Zhu; Taesung Park; Phillip Isola; Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. UC Berkeley, 2020. Berkeley AI Research (BAIR) laboratory.

[2] Aamir Jarda. A gentle introduction to cycle consistent adversarial networks. https://towardsdatascience.com/a-gentle-introduction-to-cycle-consistent-adversarial-networks-6731c8424a87, 2020.

[3] Leon Schmid; Nion Schuermeyer; Charlotte Lange; Annemarie Witschas. 08 generative adversarial networks. http://www.studip.uni-osnabrueck.de, 2020.

[4] Weining Hu; Meng Li; Xiaomeng Ju. Improved cyclegan for image-to-image translation. University of British Columbia, 2018.

[5] Ian J. Goodfellow; Jean Pouget-Abadie; Mehdi Mirza; Bing Xu; David Warde-Farley; Sherjil Ozair; Aaron Courville; Yoshua Bengio. Generative adversarial nets. Montreal, 2014. University of Montreal.

[6] Jun-Yan Zhu; Taesung Park; Phillip Isola; Alexei A. Efros. cycle_gan. https://www.tensorflow.org/datasets/catalog/cycle_gan, 01 2021. Creative Commons Attribution 4.0 License.

[7] Arnaud Rougetet. selfie2anime. https://www.kaggle.com/arnaud58/selfie2anime, 08 2019. Creative Commons Attribution 4.0 License.

[8] Leon Schmid; Nion Schuermeyer; Charlotte Lange; Annemarie Witschas. 02 training nn. http://www.studip.uni-osnabrueck.de, 2020.

[9] Leon Schmid; Nion Schuermeyer; Charlotte Lange; Annemarie Witschas. tfa.layers.instancenormalization. http://www.studip.uni-osnabrueck.de, 2020.

[10] Google Developers. tfa.layers.instancenormalization. https://www.tensorflow.org/addons/api_docs/python/tfa/ layers/InstanceNormalization, 01 2021. Creative Commons Attribution 4.0 License.

[11] Google Developers. tf.keras.optimizers.adam. https://www.tensorflow.org/api_docs/python/tf/keras/ optimizers/Adam, 02 2021. Creative Commons Attribution 4.0 License.

[12] Google Developers. tf.gradienttape. https://www.tensorflow.org/api_docs/python/tf/ GradientTape, 02 2021. Creative Commons Attribution 4.0 License.

[13] Ashish Shrivastava; Tomas Pfister; Oncel Tuzel; Josh Susskind; Wenda Wang; Russ Webb. Learning from simulated and unsupervised images through adversarial training. Apple Inc, 2017.

[14] Fabian Imkenberg; Jacqueline Naether. Final project iannwtf. https://github.com/fabian9697/IANN_group30/tree/main /Final_Project, 2021.