UNIVERSITÄT OSNABRÜCK

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

Fabian Imkenberg, Jacqueline Naether

Abstract
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Keywords
CycleGAN — Unpaired Image-to-Image Translation — Keyword3

Contents

Introduction

As the basis of our project, we studied the paper *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* from the **BAIR!** (**BAIR!**) laboratory. The aim of the paper is to change different information in images. The developed network creates a new generated image based on an existing input image. Three different types of generated images are presented in this paper. A translation of a photograph to a painting by Monet, zebras to horses, and summer to winter and vice versa. The paper complements what has been learned in the module **IANNWTF!** (**IANNWTF!**) by two essential building blocks, which basically depend on each other. [**?**]

Until now, training data was mostly available in input-output pairs. This means that in the example translation summer to winter, for each photo of a location in summer there also exists a photo of the same location in winter. Thus, the network previously learned translation using complete individual examples. In the case of the paper, however, a network is developed that is trained based on two domains. As training input, the network is fed a domain of, for example, summer images. The training output then contains a domain of winter images. The network learns the different properties of summer and winter based on these image domains. [**?**]

As a second building block, the simple **GAN!** (**GAN!**) as presented in **IANNWTF!** is extended into a so-called Cycle-**GAN!**. The name is based on the use of the Cycle Consistency Loss in the training phase. We reimplemented and tested such a Cycle-**GAN!** based on the paper. [**?**]

1. Methods

As a transfer performance, our goal is not to teach our network the same translation as described in the paper. Therefore, we decided to optimize our network to perform a translation from apple images to orange images and vice versa. For this, we had to select a suitable dataset, as well as to work out the methodological basics of the network architecture from the paper. In the following, these basics will be explained.

1.1 Architectures

As the basis of the network architecture, the **GAN!** already known through the **IANNWTF!** module is extended to the Cycle-**GAN!** known through the paper [**?**].

GAN!s basically exist out of two neural networks, which more or less battle against each other, the *Generator* and *Discriminator*. The Generator takes the input and generates a fake image out of it, while the Discriminator tries to discriminate between generated and true images. Both parts of the network are trained individually from each other. The loss used in training is the adversarial loss. [**?**, **?**]

Cycle-GAN!s extend this concept, in terms of unpaired image-to-image translation. As mentioned in the introduction, the chosen paper uses domains of input D_x and output images D_y. The Generator is trained two main functions. Generating images from domain D_x to images of domain D_y

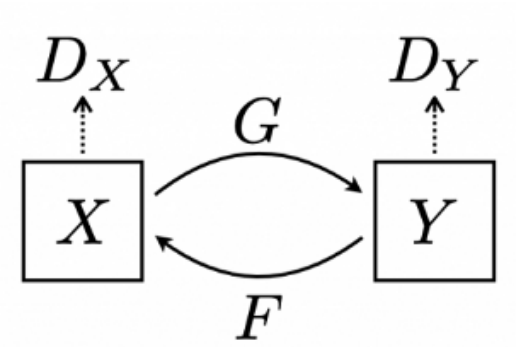

Figure 1. Mapping functions of the Cycle-**GAN!** [**?**]

$(G : X \rightarrow Y)$ and vice versa $(F : Y \rightarrow X)$. This principle is shown in **??**. [**?**]

On the basis of this it becomes clear that a simple back and forward translation can lead to the fact that the back translation does not refer to exactly the original input image, but to another image within the input image domain. This is where the cycle consistency loss comes into play. This one is used in addition to the adversarial loss so that the network learns to return to the original image during a back and forward translation. The cycle consistency losses for the two mapping functions are shown in the **??**. [**?**]

1.2 Formulation

In order to better understand the function and subsequent implementation of the losses, the mathematical background will be discussed in a bit more detail.

Adversarial Loss is already used in training of standard **GAN!**s. It arises from the fact that the generator G and discriminator D work against each other. If the generator generates particularly good fake images that the discriminator can no longer identify as such, the loss of the discriminator automatically increases. On the other hand, the loss of the generator decreases. The loss behaviour is similar the other way round. This is described mathematically in [**?**] 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 additionally to the adversarial loss in the Cycle-**GAN!**. The idea has already been briefly explained before, that a generator G translates an image *Input_A* from the domain X into an image B of the domain Y. In the backward translation of generator F, an image *Cyclic_A* from domain X will be created from the image B of domain Y. The difference between image *Input_A* and *Cyclic_A* forms the cycle consistency loss [**?**]. In [**?**], the cycle consistency loss is described 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 Cycle-**GAN!** is built from two generators, *Generator A2B* and *Generator B2A*, and two discriminators, *Discriminator A* and *Discriminator B*. This architecture is shown simplified in **??**.

Identity Loss is also introduced in the paper [**?**]. The loss is a regularization technology for the generator, to provide a better performs if the input is already near to the predicted output image. Mathematically the identity loss is defined in the paper [**?**] as follows:

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

1.3 Datasets

In addition to the architecture, a suitable datasets also had to be selected. On the one hand, it should be versatile, but on the other hand, it must not contain too large images so that the computing time is not too high.

Tensorflow provides different dataset specifically for Cycle-**GAN!**s including apple and orange images as well as photo and Monet images. These ones are called *cycle_gan/apple2orange* and *cycle_gan/monet2photo*. The datasets contain two domains A and B each for training and test data. **??** shows the four parts of the default configuration with the number of examples included in each dataset. Each example consist of image-label pairs. [**?**]

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

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

Kaggle provides another dataset *arnaud58/selfie2anime*. In order to make a further comparison, another dataset from a different library was deliberately selected. This one contains images of anime and images of women selfies. The examples included in this dataset are shown in **??**. [**?**]

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

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

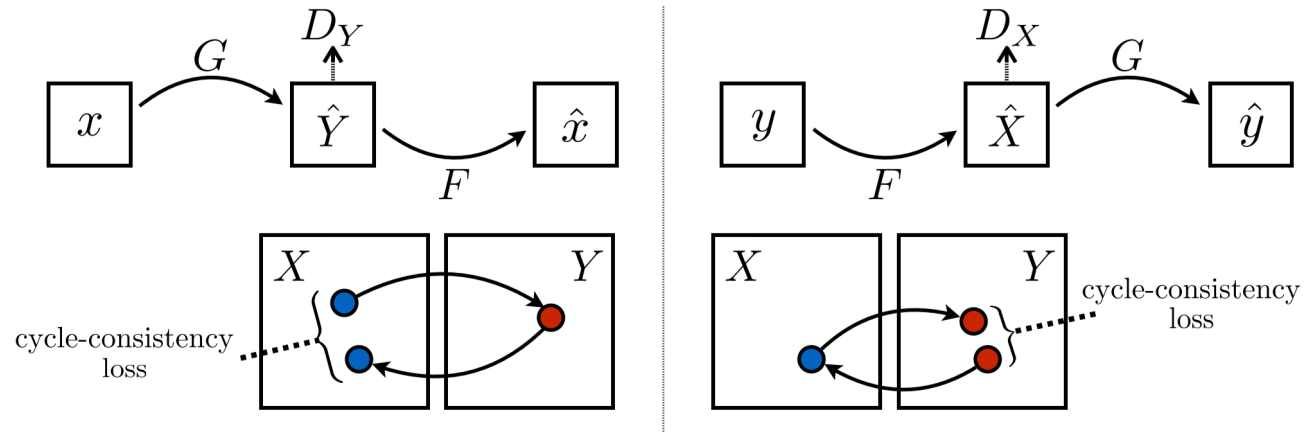

Figure 2. Cycle Consistency Losses in the back and forward translation [?]

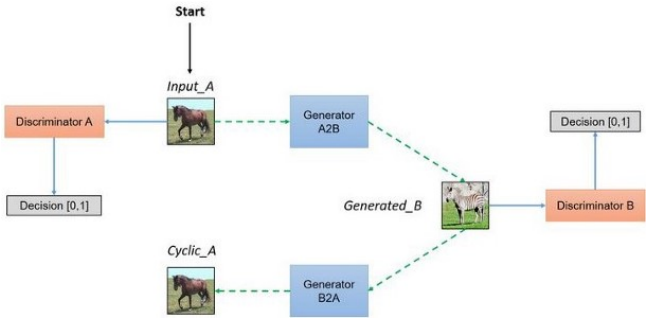

Figure 3. A simplified architecture of a Cycle-**GAN!** [?]

2. Implementation

The implementation is performed using the learned three steps, pre-processing, training and test from the module **IANNWTF!** [?, ?]. In addition, the methods mentioned in the section before are used to develop the Cycle-**GAN!** architecture. Furthermore, other functions of the Tensorflow and Keras libraries are used in all parts of the implementation. Therefore, `tensorflow`, `tensorflow_datasets`, `tensorflow_addons` and `matplotlib.pyplot` are imported. How the implementation is realized and the different libraries are used, is now explained in more detailed.

2.1 Network Architecture

The architecture is built as explained in section methods before. A class `Cycle_GAN_Generator` and a class `Cycle_GAN_Discriminator` are programmed. Two instances of each of the two classes are then created, the `generator_A_to_B` and `generator_B_to_A`, and the `discriminator_A` and `discriminator_B`.

Generator Class inherits from the Keras class `Layer`. The class in general is build up of three parts, the *Encoder*, *Transformer* and *Decoder*. Each part consists again out of different layers. The structure of the generator is shown in **??**. [?]

The **Encoder** extracts the feature of the original input image, for which convolutional layers are normally used [?]. In `Cycle_GAN_Generator`, the encoder also consists of exactly three convolutional layers, as also described in **??**. The Keras class `Conv2D` is used for this. As activation function **ReLu!** (**ReLu!**) is selected, for which the Keras class `ReLu` of the same name is used. Furthermore, the Keras class `InstanceNormalization` is used, which is a special form of group normalization, see [?], and is therefore suitable for the dataset used for the reimplementation. The normalization contains the argument axis, which is set to -1, which prevents the features per feature map from being normalized. So that the training does not exceed the time frame of the project, only a quarter of the original filter sizes of the paper are used in each `Conv2D` layer. All other values for `kernel_size`, `strides`, `activation`, `padding` and the `kernel_initializer` are taken identically from the specifications of the paper. The specifications and values for the three `Conv2D` layers are shown in **??**. [?]

Table 3. Values of all convolutional layer specifications in the encoder part of the class `Cycle_GAN_Generator` [?].

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 **Transformer** is needed to relate closely spaced features extracted from the input image by the previous encoder [?]. And this is exactly what the blocks of **ResNet!**s (**ResNet!**s), like shown in **??**, are needed for. Every **ResNet!** is build up by two `Conv2D` layers, two layers of `InstanceNormalizati` 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.

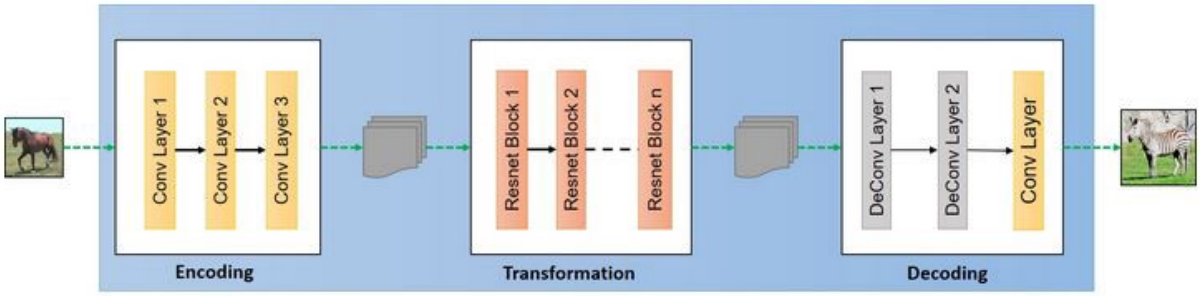

Figure 4. The high-level structure of the Cycle-**GAN!**s generator [**?**]

All values for all layers are taken from the paper. The convolutional layer specifications are shown in **??**.

Table 4. Values of the convolutional layer specifications in the transformer part of each **ResNet!** block of the class `ResNet_Block` [**?**].

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

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

The **Decoder** now leads back to an output image. For this, the low-level features are worked out with the help of the transpose convolution, see [**?**]. To achieve this, the Keras class `Conv2DTranspose` is used in this part. Finally, as shown in **??**, another `Conv2D` is used, in conjunction with an activation function of the Keras class `tanh`. The values of the transposed convolution layer specifications is found in **??**, for the convolutional layer specifications in **??**.

Discriminator Class also inherits from the Keras class `Layer`. In this reimplementation the class isn't a simple discriminator class, but 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 values for the layers and there specifications are taken from the paper and are shown in **??**. The original `filters` size here again is divided by four in the reimplementation to reduce training time.

Table 5. Values of the transposed convolutional layer specifications in the decoder part of the class `Cycle_GAN_Generator` [**?**].

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 specifications in the decoder part of the class `Cycle_GAN_Generator` [**?**].

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

After first convolutional layer they didn't deliberately used `InstanceNormalization`. Last layer is a convolution to produce an output of 1-dimension. The structure of the discriminator is also shown in **??**. [**?**]

2.2 Pre-Processing

In addition to creating the network architecture, the dataset must also be preprocessed. In order to implement easier switching between datasets and comparison versions, in addition to the reimplementation of the paper, different mechanisms were included. For switching between datasets, additional variables (`is_selfie2anime_dataset`, `is_apple2orange_dataset`, `is_monet2photo_dataset`) were implemented to select the dataset to be pre-processed. At the beginning, these variables are used to select which

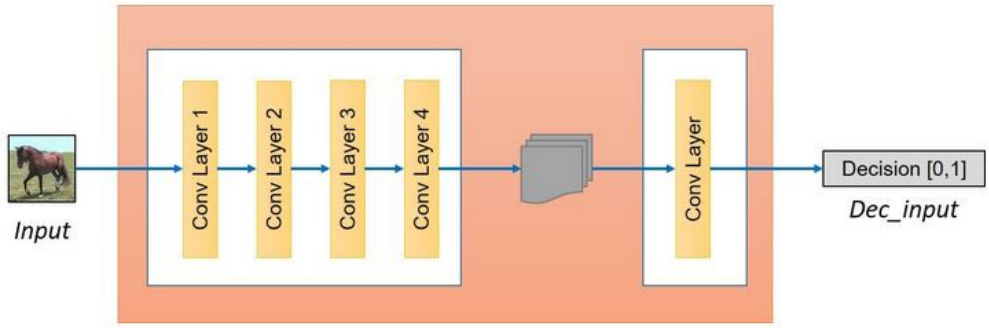

Figure 5. The high-level structure of the Cycle-**GAN!**s discriminator [**?**]

Table 7. Values of the convolutional layer specifications in the discriminator class `Cycle_GAN_Discriminator`[**?**].

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

dataset is to be loaded, by setting it `true`. Only one dataset can be loaded at a time. If several dataset arguments are set to true, the Kaggle dataset is preferred to the Tensorflow datasets due to the `if-elif-elif` query, and the *apple2orange* dataset is preferred to the *monet2photo* dataset.

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

Moreover, since two different sources for the three datasets are used, there are a few 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 directly split into test and training data, as well as in 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 arguments of the different pre-processing steps are taken directly from the paper. [**?**]

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 **??**, see [**?**], 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 randomly taken from the domains again for each training step. Afterwards the training for the `generator_A_to_B` takes place and then for the generator `generator_B_to_A`. Here for, the function `generator_train_step` was programmed, in which the adversarial loss (`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, this was used to calculate the training loss in the reimplementation for all datasets. The training loss is then used in this function to update the generator parameters. To update the gradients of the generators, the loss function *MeanAbsoluteError* is used. To update the generator parameters the optimizer *Adam* is used. [**?**, **?**, **?**]

In the second step of each training step, the discriminators are trained. Since the discriminator is to identify the output image of the generator, the image from the selected image-label pair of domains A and B is fed to the respective generator before training and the result image is returned (function: `generate_fake_dataset`). However, the discriminator is now not simply passed this training image. 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 yet filled with 50 images. As soon as the pool is filled with 50 images, a random image from the buffer is exchanged against the current training image when the function is called. The exchanged image is then given to the discriminator as input. So it can happen that the discriminator gets an image, where it is easier for the discriminator to distinguish a generated image from a real image, because the image was generated by a worse generator version. This prevents, after [**?**], the oscillation of **GAN!**s. 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. [**?**, **?**, **?**]

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 an Excel spreadsheet in Google Drive. The evaluation is then carried out with these values from these lists.

2.4 Test

As in the training, this time 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, as in the training. However, this time 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 were programmed in addition to the reimplementation. Like mentioned before, also the test losses are saved into Excel files in the Google Drive. 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 Cycle**GAN!** 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 **??**.

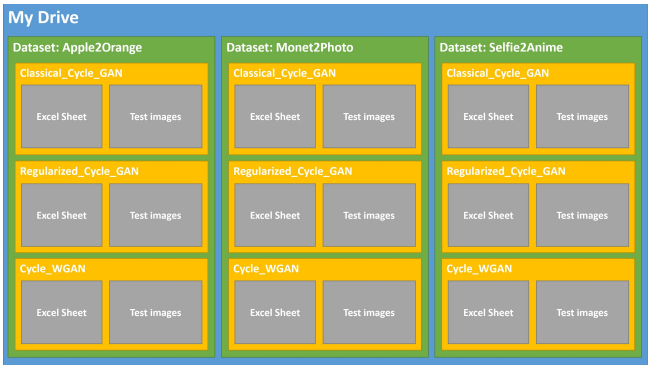

Figure 6. The folder structure of storing data in the Google Drive from the Colab script.

3. Evaluation

After the pure reimplementation of the Cycle-**GAN!**, two further improvements are implemented. All three variants are then compared with each other. The implementation of the two improved variants is explained in more detail in the following two subsections.

3.1 Regularized ResNet! Blocks

The first idea is taken from the original paper, see [**?**], and continued. Here also the Cycle-**GAN!** was already tested with options for the regularization. In this variant a further class for the **ResNet!** block, the `Regularized_ResNet_Block`, was programmed. This contains additionally a 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*, then the default **ResNet!** is used during initialization. Is this argument set to *true* during initialization, the Cycle-**GAN!** 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 instant of the discriminator class the argument `kernel_regularizer` with a L2 loss regularizer is introduced with the convolutional layers. In the standard version of the Cycle-**GAN!** this is also initialized, but not used. In the

training and test therefore also an argument, `Regularized_Cycle_GAN`, is used, in order to control the computation of the loss dependent on it. If the argument is set to *true*, the L2 loss for the discriminator is calculated and used for parameter adjustment.

3.2 Cycle WassersteinGAN!

Furthermore, another method is introduced, according to [**?**]. Here, instead of proceeding according to a simple **GAN!**, this is replaced by a **WGAN!** (**WGAN!**). This leads to the fact that first of all instead of the optimizer *Adam*, the optimizer ***RMSProp! (RMSProp!)*** is used. In addition, the weights in the discriminator are clipped, which also leads to the fact that the discriminator now has to be trained five times more than the generator. Furthermore, the *sigmoid* activation is removed from the output layer of the discriminator. To implement all of this, a new class, `Cycle_WGAN_Discriminator`, is programmed for the **WGAN!** discriminator.

So that the network can also be included in the same lean colab notebook, the argument `Cycle_WGAN` is introduced. If the argument is set to *true*, then a Cycle-**WGAN!** is created, in the other case a standard Cycle-**GAN!**. Here then also the optimizer is initialized instead of as *Adam*, as ***RMSProp!***.

4. Results

4.1 Training vs. Test

1. Welche Ergebnisse erzielt das Training?
2. Welches Ergebnis erzielt der Test?

4.2 Comparison of datasets

1. Welche Ergebnisse erzielt apple2orange?
2. Welche Ergebnisse erzielt monet2photo?
3. Welche Ergebnisse erzielt selfie2anime?
4. Welche Ergebnisse zielen sie im vergleich zu einander?

4.3 Comparison of network variations

1. Welche Ergebnisse erzielt standard cycleGAN?
2. Welche Ergebnisse erzielt regularized cycle GAN?
3. Welche Ergebnisse erzielt wasserstein cycle gan?
4. Welche Ergebnisse zielen sie im vergleich zu einander?

5. Limitations and Discussion

1. Datensatz/Bildgröße
2. Dauer des Trainings und beanspruchung der rechenzeit
3. noch besser wäre es wenn die 5 bilder noch zwischen den Unterschiedlichen GAN varianten gleich wäre, aber auf grund von shuffel ist das nicht so einfach und da das notepad immer neu gestartet wird
4. besser test und training gleichviele abspeichern oder noch öfter, aber das wäre zu zeit intensiv