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 Berkeley AI Research (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 Implementing Artificial Networks with Tensorflow (IANNWTF) by two essential building blocks, which basically depend on each other. [1]

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. [1]

As a second building block, the simple Generative Adversarial Network (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]

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

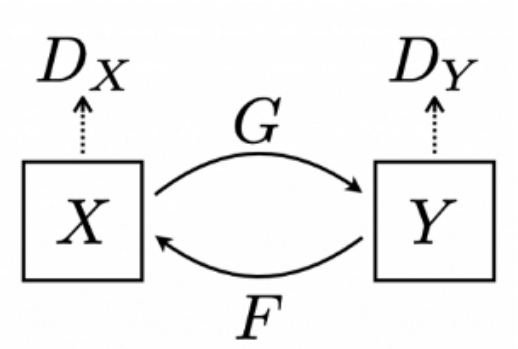

Figure 1. Mapping functions of the Cycle-GAN [1]

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 [1].

Generative Adversarial Networks 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 adveresarial loss. [2, 3]

Cycle-GANs extend this concept, in terms of unpaired image-to-image translation. As mentioned in the introduction, the choosen 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 ($G : X \rightarrow Y$) and vice versa ($F : Y \rightarrow X$). This principle is shown in Figure 1. [1]

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 Figure 2. [1]

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 GANs. 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 behavior is similar the other way round. This is described mathematically in [4] 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 [2]. In [2], 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* und *Discriminator B*. This architecture is shown simplified in Figure 3.

1.3 Dataset

In addition to the architecture, a suitable dataset 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. The library `tensorflow_dataset` provides a dataset specifically for Cycle-GANs including apple and orange images. This one is called *cycle_gan/apple2orange*. The dataset contains two domains A and B each for training and test data. Table 1 shows the four parts of the default configuration with the number of examples included. The examples each consist of image-label pairs. The dataset has a total size of 74,82 MiB and can therefore still be loaded and processed in a manageable time for this project. [5]

2. Implementation

The implementation is performed using the learned three steps, pre-processing, training and test from the module IANNWTF [6, 7]. In addition, the methods mentioned in the section

Table 1. Structure of the standard configuration split of the Tensorflow dataset *cycle_gan/apple2orange* [5]

index	examples
'testA'	266
'testB'	248
'trainA'	995
'trainB'	1019

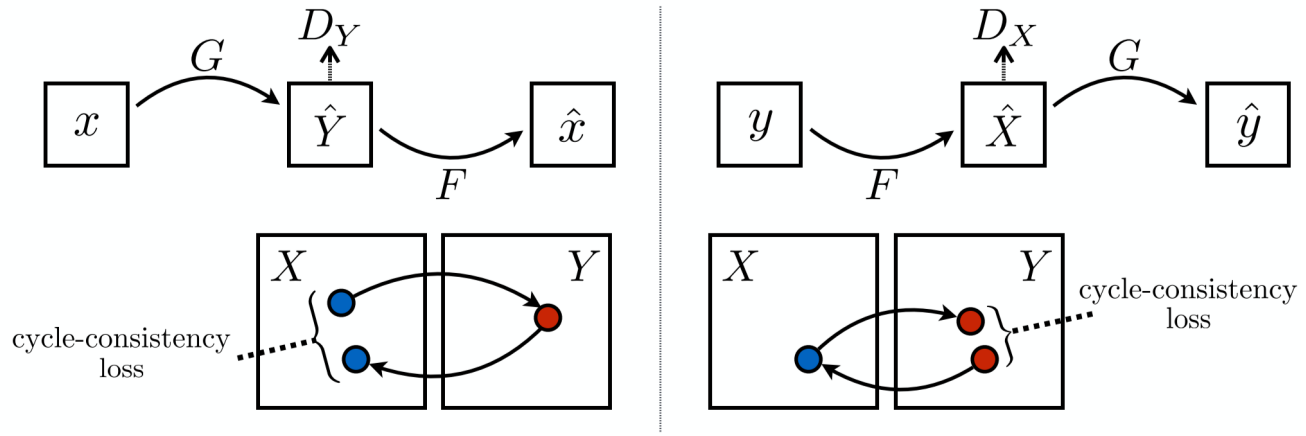

Figure 2. Cycle Consistency Losses in the back and forward translation [1]

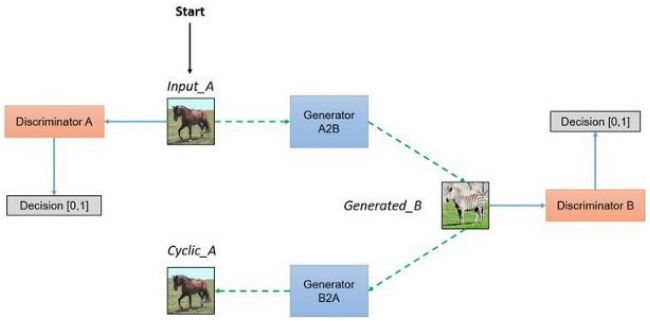

Figure 3. A simplified architecture of a Cycle-GAN [2]

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` is imported as `tf` and `tensorflow_datasets` imported as `tfds`. In addition to these, the `tensorflow_addons`, imported as `tfa`, and the `matplotlib.pyplot`, imported as `plt`, are also needed for further implementation. How this implementation is realized and the different libaries 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 Figure 4. [2]

The **Encoder** extracts the feature of the original input image, for which convolutional layers are normally used [2]. In `Cycle_GAN_Generator`, the encoder also consists of exactly three convolutional layers, as also described in Figure 4. The Keras class `Conv2D` is used for this. As activation function Rectified Linear Units (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 [8], and is therefore suitable for the dataset used for the reimplementation. The pseudocode is described below:

```
self.conv_1 = tf.keras.layers.Conv2D(...)
self.instance_norm_1 = tfa.layers.
    InstanceNormalization(...)
self.activ_1 = tf.keras.layers.ReLu()

self.conv_2 = tf.keras.layers.Conv2D(...)
self.instance_norm_2 = tfa.layers.
    InstanceNormalization(...)
self.activ_2 = tf.keras.layers.ReLu()

self.conv_3 = tf.keras.layers.Conv2D(...)
self.instance_norm_3 = tfa.layers.
    InstanceNormalization(...)
self.activ_3 = tf.keras.layers.ReLu()
```

The **Transformer** is needed to relate closely spaced features extracted from the input image by the previous encoder, [2]. And 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 as follows:

```
self.conv_1 = tf.keras.layers.Conv2D(...)
self.instance_norm_1 = tfa.layers.
    InstanceNormalization(...)
self.activ_1 = tf.keras.layers.ReLu()

self.conv_2 = tf.keras.layers.Conv2D(...)
self.instance_norm_2 = tfa.layers.
    InstanceNormalization(...)
self.concat = tf.keras.layers.Concatenate()
```

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

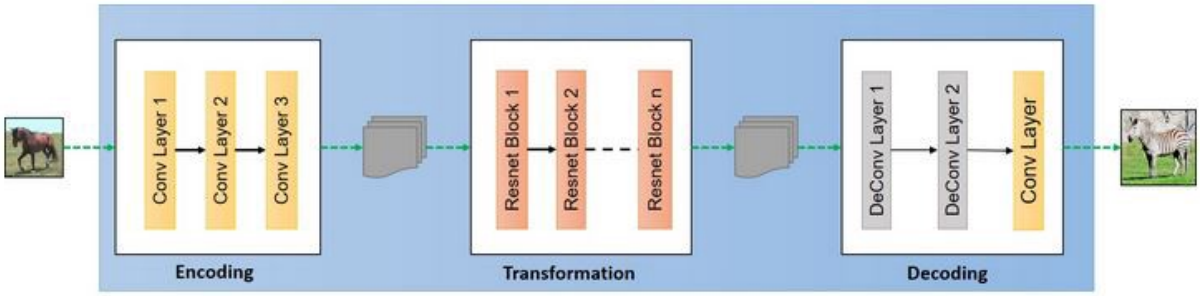

Figure 4. The high-level structure of the Cycle-GANs generator [2]

on the instance variable `n_resnet`, which is set to 6 by default.

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 [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 pseudocode is as follows:

```
self.transp_conv_1 = tf.keras.layers.
    Conv2DTranspose(...)
self.instance_norm_4 = tfa.layers.
    InstanceNormalization(...)
self.activ_4 = tf.keras.layers.ReLu()

self.transp_conv_2 = tf.keras.layers.
    Conv2DTranspose(...)
self.instance_norm_5 = tfa.layers.
    InstanceNormalization(...)
self.activ_5 = tf.keras.layers.ReLu()

self.conv_4 = tf.keras.layers.Conv2D(...)
self.instance_norm_6 = tfa.layers.
    InstanceNormalization(...)
self.output_image = tf.keras.activations.tanh
```

Discriminator Class also inherits from the Keras class `Layer`. In this reimplementation the class is just simple network, including different layers. Since it has an image as input and a simple decision vector as output, `Conv2D` layers, are associated with `InstanceNormalization` and different activation functions such as `ReLu` and `LeakyReLu`, see following pseudocode. The structure of the discriminator is shown in Figure 5. [2]

```
self.conv_1 = tf.keras.layers.Conv2D(...)
??
self.activ_1 = tf.keras.layers.LeakyReLU(...)

self.conv_2 = tf.keras.layers.Conv2D(...)
self.instance_norm_2 = tfa.layers.
    InstanceNormalization(...)
self.activ_2 = tf.keras.layers.LeakyReLU(...)

self.conv_3 = tf.keras.layers.Conv2D(...)
self.instance_norm_3 = tfa.layers.
    InstanceNormalization(...)
self.activ_3 = tf.keras.layers.LeakyReLU(...)

self.conv_4 = tf.keras.layers.Conv2D(...)
self.instance_norm_4 = tfa.layers.
    InstanceNormalization(...)
self.activ_4 = tf.keras.layers.LeakyReLU(...)

self.patch_output = tf.keras.layers.Conv2D(...)
```

2.2 Pre-Processing

In addition to creating the network architecture, the dataset must also be preprocessed. Since this is a Tensorflow dataset, it can be loaded more easily with the `tfds.load` function. When the dataset is loaded, it is also directly split into test and training data 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 *128x128*. The `image.resize` function of the Tensorflow library is used for this. After normalization and reshaping the tensor to *128x128x3*, the dataset is shuffled and prefetched. Batching is omitted to facilitate the later definition of real samples in training. The pseudocode shows all preprocessing steps.

```
train_data, test_data = tfds.load(...)

train_data = tf.image.resize each train_image to
    [128, 128]
train_data = 2 * (each train_image / 255) -1)
train_data = tf.reshape each train_image_tensor to
     [128, 128, 3]
train_data = train_data.shuffle(...).prefetch(...)

test_data = tf.image.resize each test_image to
    [128, 128]
test_data = 2 * (each test_image / 255) -1)
test_data = tf.reshape each test_image_tensor to
    [128, 128, 3]
test_data = test_data.shuffle(...).prefetch(...)
```

1. Warum shuffel 32?
2. warum prefetch 32?
3. Warum bilinear ge resized?

2.3 Training

1. Welche Schritte gibt es im Training? Generator/Discriminator
2. Welche Klassen/Funktionen wurden hierfür dann geschrieben?
3. Wo sind relevanten Werte gewählt worden? Warum wurden sie so gewählt?(z.B. Numbers of epoch?)

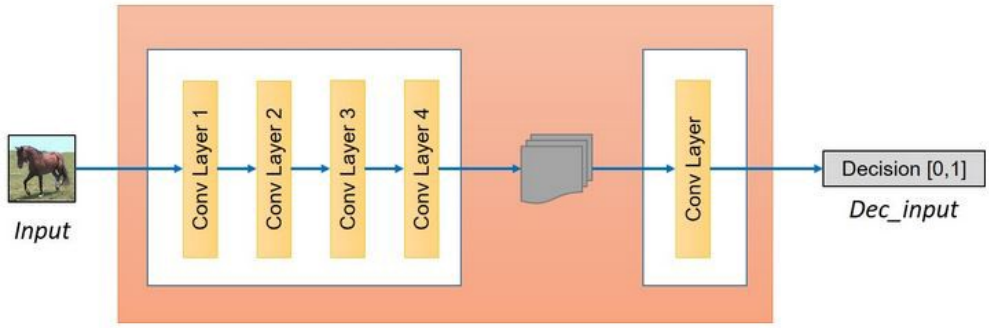

Figure 5. The high-level structure of the Cycle-GANs discriminator [2]

2.4 Test

1. Welche Schritte gibt es im Test? Generator/Discriminator, einfacher Forward Step?
2. Ausgabe/Ergebnis anzeigen, wie umgesetzt?

3. Evaluation

1. Wir haben improvements anhand folgender zwei paper vorgenommen

3.1 Improved Paper 1

3.2 Improved Paper 2

4. Results

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

5. Limitations and Discussion

1. Datensatz/Bildgröße
2. Dauer des Trainings und beanspruchung der rechenzeit

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] 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.

[5] 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.

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

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

[8] Samual Marks; Gabriel de Marmiesse; Sean Morgan; Tzu-Wei Sung; Moritz Kroeger; Kaixi Hou; David Honzatko; Aaron Mondal; Aakash Kumar Nain; Dimitry Nikitin; Autoih. tfa.layers.instancenormalization. https://www.tensorflow.org/addons/api_docs/python/tfa/layers/InstanceN
01 2021. Creative Commons Attribution 4.0 License.