

# Deep Learning Practical 6

## Implementing CycleGAN

### 1 Overview

Our sixth deep learning practical will be an interesting one and will guide you through implementing a larger deep learning project than we have seen so far in our demos. In the recent lecture, we learned how Generative Adversarial Networks (GAN) work and we even looked at various improvements, variants and advanced applications. It would be good to familiarise yourself with the lecture material before going through this practical. We also went through a couple of practical examples in the lecture, which you might want to look at, since this practical assumes basic knowledge of adversarial training:

- [🔗 Simple GAN Example](#)
- [🔗 Conditional GAN Example](#)

In this practical, we are going to be focusing on a very interesting application of GANs, which is CycleGAN. This work, originally published in a [🔗](#) paper entitled “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, enables translation between two “domain” of images. You may have seen the seminal image showing zebras having been converted to horses. Of course, this is made possible despite the fact that there is no training dataset that goes from zebras to horses with exactly the same pose and in exactly the same environment. Even David Attenborough can’t really get animals to pose for aligned photographs. The most impressive thing about CycleGAN is that it makes “unpaired” image-to-image translation possible.



This practical will walk you through implementing CycleGAN. It is not necessary for you to have read the paper, but it will be helpful even if you do it after the practical itself. In this practical, we will be covering:

- a method of dataloading for unpaired translation
- designing large networks
- iterative training of multiple networks

The answer is provided for you in the form of a Jupyter Notebook. It would be good to try to complete the practical while resisting the urge to look at the answer but of course if you feel like it will help you understand things better, that is the primary objective here. You can find the answer on Github, which you should be able to download or directly import into Google Colab if you need:

- [🔗 Answer to the practical](#)

### 2 Preparing the Dataset

As we mentioned earlier, this unpaired image translation approach will mean we can go from one set of images to another set of images and of course these two sets of images do not need to have the same number of images in them. But we are going to make our lives a bit easier and use the fantastic [🔗](#) AckBinks dataset! Let’s start building up our code.

First, let’s import what we need:

```
import torch
import torch.nn as nn
import torch.nn.functional as fn
import torchvision

import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print(f"Device is {device}!")
```

Now, it is time to download the dataset. We have done this before if you remember from Lecture 2:

```
!wget -q -O AckBinks.zip https://github.com/atapour/dl-pytorch/blob/main/2.Datasets/AckBinks/AckBinks.zip?raw=true

!unzip -q AckBinks.zip

!rm AckBinks.zip
```

Now that we have the dataset downloaded, we can start defining the transforms. Using [Compose](#), make sure the images are [Resized](#) to  $32 \times 32$ , [RandomHorizontalFlip](#) is applied to all images as a form of augmentation and the images are converted to torch tensors via [ToTensor](#).

When your transforms are ready, create the dataset object using [ImageFolder](#) pointing to where the dataset is stored, like so:

```
dataset = torchvision.datasets.ImageFolder('AckBinks/train', transform)
```

Now that we have the overall dataset, we can start separating the Jar Jar Binks and Admiral Ackbar images. An excellent way to do this, of course, would be to create your own custom dataset class. You can have a look at this [tutorial](#) later but for now we are going to use the next few lines to create two separate datasets for Jar Jar and the good Admiral. Make sure you understand what is happening in the following lines. Consult the [documentation](#) if you need to.

```
# Admiral Ackbar images:
ackbar_idx = torch.tensor(dataset.targets) == 0

# Jar Jar Binks images:
jar_jar_idx = torch.tensor(dataset.targets) == 1

jar_jar_dataset = torch.utils.data.dataset.Subset(dataset, np.where(jar_jar_idx == 1)[0])
ackbar_dataset = torch.utils.data.dataset.Subset(dataset, np.where(ackbar_idx == 1)[0])
```

Now use [DataLoader](#) to create two separate dataloaders called `ackbar_loader` and `jar_jar_loader`. Use a `batch_size` of 1 here for simplicity. Though higher batch size will work, a batch of 1 is cleaner for smaller datasets even though it is not very efficient.

When you have the dataloaders sorted, try to visualise a few of the images from each set to make sure you have done things correctly. We have previously done this in most of the demos we have looked at so you can take a look at source code in our previous examples or try your own creative methods.

### 3 Network Architectures

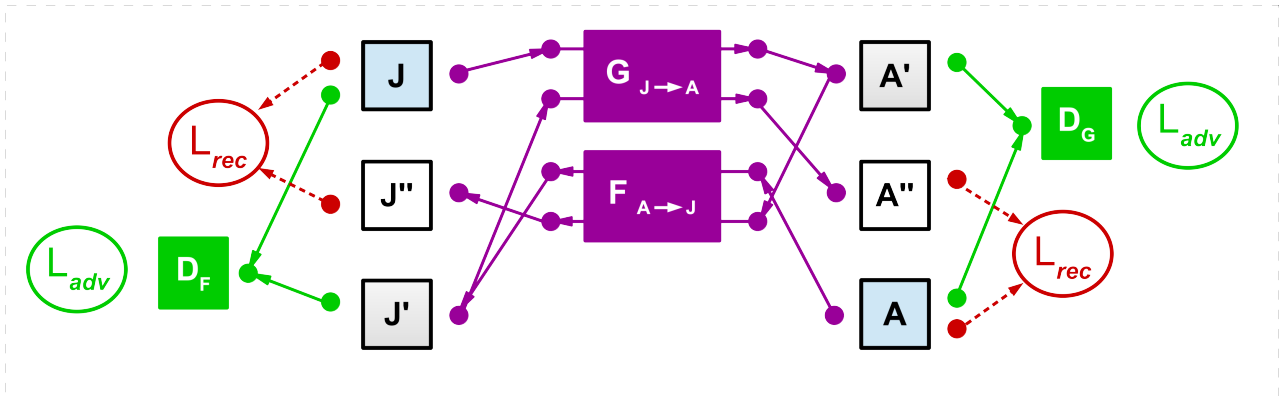
Having the data sorted, now is the time to define the networks we are going to use. CycleGAN uses two generators and two discriminators. The overall model needs two generator models: one generator ( $G$ ) for generating images for the first domain (A for Ackbar) and the second generator ( $F$ ) for generating images for the second domain (J for Jar Jar).

- Domain J  $\rightarrow$  Generator  $G \rightarrow$  Domain A
- Domain A  $\rightarrow$  Generator  $F \rightarrow$  Domain J

Each generator has a corresponding discriminator model ( $D_F$  and  $D_G$ ). The discriminator model takes real images from each domain and generated images from the corresponding generator to predict whether they are real or fake.

- Domain J  $\rightarrow$  Discriminator  $D_F \rightarrow$  [Real/Fake]
- Domain A  $\rightarrow$  Generator  $F \rightarrow$  Discriminator  $D_F \rightarrow$  [Real/Fake]
- Domain A  $\rightarrow$  Discriminator  $D_G \rightarrow$  [Real/Fake]
- Domain J  $\rightarrow$  Generator  $G \rightarrow$  Discriminator  $D_F \rightarrow$  [Real/Fake]

The following diagram might make things a bit simpler to follow. You will see that “cyclic consistency” is an important part of the process (denoted by  $L_{rec}$ ), e.g., a real image of the Admiral is converted into a fake image of Jar Jar, which is then converted back into a fake image of the Admiral. The generated fake image must of course match the original real image used as the input. This cyclic consistency should hold both ways. You will hopefully see this in more details as we develop the code further but ask questions if something is not clear.



But first things first. Let’s make the architectures of our networks. For details of the architectures of the generators and the discriminators, you can refer to the original paper or the [code](#) released by the authors. We won’t be sticking to the exact architectures used in the paper and will simplify things somewhat but have tried to use building blocks that will hopefully help you in your future deep learning journey.

## Discriminators

We will first build the architecture of our discriminators. Note that the architecture defined in this exercise will not follow the architecture of CycleGAN exactly.

```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()

        # ...

    def forward(self, input):
        # ...

Dg = discriminator().to(device)
Df = discriminator().to(device)
```

Our discriminators will have 5 layers:

- The **first layer** will be a [Conv2d](#) that receives a 3-channel image as the input, produce 64 feature channels as its output with a `kernel_size` of 4, `stride` of 2 and `padding` of 1. This layer will have a [leaky\\_relu](#) as its non-linearity with `negative_slope = 0.2`.
  - The **second layer** will be a [Conv2d](#) that receives the output of the previous layer and produces 128 feature channels as its output with a `kernel_size` of 4, `stride` of 2 and `padding` of 1. The layer should be followed by an [InstanceNorm2d](#). [Various types of normalisation](#) are now a staple of deep learning architectures. Instance normalisation normalises across each channel in each training sample. This [blog post](#) might help you understand normalisation techniques a bit better.
- The second layer will also have a [leaky\\_relu](#) as its non-linearity with `negative_slope = 0.2`.

- The **third layer** will also be a [Conv2d](#) that receives the output of the previous layer and produces 256 feature channels as its output with a `kernel_size` of 4, `stride` of 2 and `padding` of 1.  
The layer should be followed by an [InstanceNorm2d](#) and should have a [leaky\\_relu](#) as its non-linearity with `negative_slope = 0.2`.
- The **fourth layer** will also be a [Conv2d](#) that receives the output of the previous layer and produces 512 feature channels as its output with a `kernel_size` of 4, `stride` of 1 and `padding` of 1.  
The layer should be followed by an [InstanceNorm2d](#) and should have a [leaky\\_relu](#) as its non-linearity with `negative_slope = 0.2`.
- The **fifth layer** will be the “head” of the network. It will be a [Conv2d](#) that receives the output of the previous layer and produces a single neuron as its output with a `kernel_size` of 4, `stride` of 1 and `padding` of 1. As this is the output, no normalisation is needed.  
The final output should then be passed through a [sigmoid](#). Note that if you are using the [BCELoss](#), you won’t need the Sigmoid here since the PyTorch implementation of [BCELoss](#) already has the Sigmoid. Since the answer will not use the BCELoss, we will add the sigmoid layer here.

## Generators

We will now build the architecture of our generators. Note that the architecture defined in this exercise will not follow the architecture of CycleGAN exactly. The architecture of our generator will be much deeper and contains many more layers than our discriminator. We also use [ReflectionPad2d](#), which as the name suggests enables padding the outer edges with their own reflection.

The definition of the layers is provided for you but you should complete the `forward` function based on the instructions. The decoder part of the generator will use [ConvTranspose2d](#).

```
class generator(nn.Module):
    def __init__(self):
        super(generator, self).__init__()

        self.r1 = nn.ReflectionPad2d(3)
        self.conv1 = nn.Conv2d(3, 32, kernel_size=7, stride=1)
        self.in1 = nn.InstanceNorm2d(32)

        self.r2 = nn.ReflectionPad2d(1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2)
        self.in2 = nn.InstanceNorm2d(64)

        self.r3 = nn.ReflectionPad2d(1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=2)
        self.in3 = nn.InstanceNorm2d(128)

        self.r4 = nn.ReflectionPad2d(1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3)
        self.in4 = nn.InstanceNorm2d(128)

        self.r5 = nn.ReflectionPad2d(1)
        self.conv5 = nn.Conv2d(128, 128, kernel_size=3)
        self.in5 = nn.InstanceNorm2d(128)

        self.r6 = nn.ReflectionPad2d(1)
        self.conv6 = nn.Conv2d(128, 128, kernel_size=3)
        self.in6 = nn.InstanceNorm2d(128)

        self.r7 = nn.ReflectionPad2d(1)
        self.conv7 = nn.Conv2d(128, 128, kernel_size=3)
        self.in7 = nn.InstanceNorm2d(128)

        self.r8 = nn.ReflectionPad2d(1)
        self.conv8 = nn.Conv2d(128, 128, kernel_size=3)
        self.in8 = nn.InstanceNorm2d(128)

        self.r9 = nn.ReflectionPad2d(1)
        self.conv9 = nn.Conv2d(128, 128, kernel_size=3)
        self.in9 = nn.InstanceNorm2d(128)

        self.r10 = nn.ReflectionPad2d(1)
        self.conv10 = nn.Conv2d(128, 128, kernel_size=3)
        self.in10 = nn.InstanceNorm2d(128)

        self.r11 = nn.ReflectionPad2d(1)
```

```

self.conv11 = nn.Conv2d(128, 128, kernel_size=3)
self.in11 = nn.InstanceNorm2d(128)

self.r12 = nn.ReflectionPad2d(1)
self.conv12 = nn.Conv2d(128, 128, kernel_size=3)
self.in12 = nn.InstanceNorm2d(128)

self.r13 = nn.ReflectionPad2d(1)
self.conv13 = nn.Conv2d(128, 128, kernel_size=3)
self.in13 = nn.InstanceNorm2d(128)

self.r14 = nn.ReflectionPad2d(1)
self.conv14 = nn.Conv2d(128, 128, kernel_size=3)
self.in14 = nn.InstanceNorm2d(128)

self.r15 = nn.ReflectionPad2d(1)
self.conv15 = nn.Conv2d(128, 128, kernel_size=3)
self.in15 = nn.InstanceNorm2d(128)

self.uconv16 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1)
self.in16 = nn.InstanceNorm2d(64)

self.uconv17 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1)
self.in17 = nn.InstanceNorm2d(32)

self.r18 = nn.ReflectionPad2d(3)
self.conv18 = nn.Conv2d(32, 3, kernel_size=7, stride=1)
self.in18 = nn.InstanceNorm2d(3)




```

```
def forward(self, input):
```

```
    # ...
```

```
G = generator().to(device)
F = generator().to(device)
```

The forward pass through the generator should go through the following layers:

- **Layer 1:** conv1 with  ReflectionPad2d (r1) and  InstanceNorm2d (in1) and  leaky\_relu with negative\_slope = 0.2, like so:

```
x = fn.leaky_relu(self.in1(self.conv1(self.r1(input))), negative_slope=0.2)
```

- **Layer 2:** conv2 with padding r2 and norm in2 and leaky\_relu with negative\_slope = 0.2.
- **Layer 3:** conv3 with padding r3 and norm in3 and leaky\_relu with negative\_slope = 0.2.
- **Layer 4:** conv4 with padding r4 and norm in4 and leaky\_relu with negative\_slope = 0.2.
- **Layer 5:** conv5 with padding r5 and norm in5 and leaky\_relu with negative\_slope = 0.2.

Layers 4 and 5 should form a residual block. Create a residual connection from Layer 3 to Layer 5. If you remember, this can simply be done by summing the output of Layer 3 and the output of Layer 5 before sending the result to Layer 6.

- **Layer 6:** conv6 with padding r6 and norm in6 and leaky\_relu with negative\_slope = 0.2.
- **Layer 7:** conv7 with padding r7 and norm in7 and leaky\_relu with negative\_slope = 0.2.

Layers 6 and 7 should form a residual block. Create a residual connection from Layer 5 to Layer 7.

- **Layer 8:** conv8 with padding r8 and norm in8 and leaky\_relu with negative\_slope = 0.2.
- **Layer 9:** conv9 with padding r9 and norm in9 and leaky\_relu with negative\_slope = 0.2.

Layers 8 and 9 should form a residual block. Create a residual connection from Layer 7 to Layer 9.

- **Layer 10:** conv10 with padding r10 and norm in10 and leaky\_relu with negative\_slope = 0.2.
- **Layer 11:** conv11 with padding r11 and norm in11 and leaky\_relu with negative\_slope = 0.2.

Layers 10 and 11 should form a residual block. Create a residual connection from Layer 9 to Layer 11.

- **Layer 12:** conv12 with padding r12 and norm in12 and leaky\_relu with negative\_slope = 0.2.
- **Layer 13:** conv13 with padding r13 and norm in13 and leaky\_relu with negative\_slope = 0.2.

Layers 12 and 13 should form a residual block. Create a residual connection from Layer 11 to Layer 13.

- **Layer 14:** conv14 with padding r14 and norm in14 and leaky\_relu with negative\_slope = 0.2.
- **Layer 15:** conv15 with padding r15 and norm in15 and leaky\_relu with negative\_slope = 0.2.

Layers 14 and 15 should form a residual block. Create a residual connection from Layer 13 to Layer 15.

- **Layer 16:** Transpose convolution `uconv16` with instance norm `in16` and `leaky_relu` as the non-linearity with `negative_slope = 0.2`.
- **Layer 17:** Transpose convolution `uconv17` with instance norm `in17` and `leaky_relu` as the non-linearity with `negative_slope = 0.2`.
- **Layer 18:** Ourput layer consisting of `conv18` with reflection padding `r18` and norm `in18` and `leaky_relu` with `negative_slope = 0.2`.

The final output should then be passed through a [↗](#) `sigmoid` to make sure the values are between zero and one. Note that the original architecture uses a [↗](#) `Tanh`, and we only use a Sigmoid for simplicity and easier processing for visualisation.

After we have the model architectures, we can set up our optimisers and get ready for the main training loop:

```
G_optim = torch.optim.Adam(G.parameters(), lr=0.0002)
F_optim = torch.optim.Adam(F.parameters(), lr=0.0002)
Dg_optim = torch.optim.Adam(Dg.parameters(), lr=0.0001)
Df_optim = torch.optim.Adam(Df.parameters(), lr=0.0001)
```

## 4 Main Training Loop

Now for the main event, the training loop. A lot of the code will be provided for you but since we are operating within loops, make sure you get the python indentation right.

Let's start by setting up a few things:

```
epoch = 0
epochs = 2000
step = 0

G.train()
F.train()
Dg.train()
Df.train()
```

We are going to train the models for 2000 epochs (a number that is arbitrarily chosen - you may need to train the pipeline much longer for any reasonable results but we aren't really looking for any fantastic results here). We will use a nested loop for training. The outer loop will enforce the number of epochs we are going to train the model for:

```
while (epoch < epochs):
    ...
```

In the inner loop, we are going to extract the batch data. If you remember, we created two dataloaders, one for each of our image domains. Now how do we read images from each of these? Given that our two sets of data contain the same number of images, you may want to look at [↗](#) `zip()` in Python.

No matter how you extract the data, read the images and put them in two variables, one for each domain, `ack_real`, which would hold real images of the good Admiral Ackbar, and `jar_real`, which should hold real images of Jar Jar Binks. Make sure the images are on `cuda` as well.

Given that we are basically training GANs, we will need real and fake labels for our discriminators:

```
label_fake = torch.zeros(1)[0].to(device)
label_real = torch.ones(1)[0].to(device)
```

With everything sorted, we are ready to start training the first cycle.

### Training the First Cycle ( $G$ and $D_G$ ) - Jar Jar to Ackbar

Right, we are getting to the fine details here. We are going to train the discriminator first and in order to do that, we will need a fake Admiral Ackbar image. Pass the `jar_real` through the generator `G` and place the output in `ack_fake`. Make sure you [↗](#) `detach()` since we are only going to train  $D_g$  and not  $G$ .

We are going to train the discriminator twice, once with the real image and once with the fake image. So pass `ack_fake` and `ack_real` through  $D_g$  and calculate the loss. You can use [↗](#) `BCELoss` as the loss for the discriminators. In the answer, I have used the Mean Squared Error, just to show what is possible. But feel free

to experiment with both. Note that you will need to remove the Sigmoid from the end of the discriminator if you are going to use the [🔗 BCELoss](#).

Take the proper “`step`”s and train the discriminator. Make sure you use the `label_real` as the ground truth for the real image and `label_fake` as the ground truth for the fake image.

We are now ready to start training the generator  $G$ . To train the generator, we will need a fake image again. Pass the `jar_real` through the generator  $G$  and place the output in `ack_fake`. Make sure you **don't** `detach()` this time since we are indeed going to train  $G$ . Once you obtain `ack_fake`, pass it through the discriminator and calculate the loss. Make sure you use the `label_real` as the ground truth of our fake image since we want to train the generator to actually fool the discriminator - i.e., we want the fake image to look real...!

An important part of CycleGAN is enforcing its cyclic consistency. Now that we have trained the generator using our discriminator, we are going to train it using the “cyclic” reconstruction loss. In order to do this, you will pass `ack_real` through the generator  $F$  and then pass the output through  $G$ . If cyclic consistency holds, then the output of  $G(F(\text{ack\_real}))$  should be exactly the same as `ack_real`. So, we shall minimise the distance between  $G(F(\text{ack\_real}))$  and `ack_real` as part of our loss for training the generator. Use the Mean Absolute Error as the loss function for this.

Given that there are two loss components for training the generator, we can weight these losses. In the answer, I have made the cyclic loss 10 times stronger than the adversarial loss, but this weighting coefficient is a hyperparameter that needs to be tuned depending on the data and the network architectures. This concludes the training of our first cycle.

## Training the Second Cycle ( $F$ and $D_F$ ) - Ackbar to Jar Jar

You can probably tell where this is going. The training of the second cycle is exactly the same as that of the first. The only difference being that we are going the other direction so you should be able to follow the same basic procedure used to train  $G$  and  $D_G$  in order to train  $F$  and  $D_F$ .

This should conclude one training iteration of our model. To make sure things are going well, you can visualise images and print out losses. The answer will use the following code snippet to print out loss values and show the images every 200 iterations.

```
if step % 200 == 0:
    print(f'Epoch: {epoch} — Step: {step}')
    print(f'Discriminator Dg loss: {loss_Dg.item()} — Generator G loss: {loss_G.item()}')
    print(f'Discriminator Df loss: {loss_Df.item()} — Generator F loss: {loss_F.item()}')

    vis1 = torch.stack([ack_real[0].cpu().detach(), jar_fake[0].cpu().detach()])
    vis2 = torch.stack([jar_real[0].cpu().detach(), ack_fake[0].cpu().detach()])
    vis = torch.cat([vis1, vis2], dim=2)

    plt.imshow(torchvision.utils.make_grid(vis).permute(1, 2, 0), cmap=plt.cm.binary)
    plt.show()
step += 1
```

Obviously, this is just an exercise and based on very small dataset, we are not going to get fantastic results. You should be able to roughly see the outline of Jar Jar looking like the Admiral and vice versa if you squint really really hard after a few hundred iterations but not much beyond that. When you are ready, take a look at the [🔗 Answer](#) and make sure you understand the overall process and implementation details.