Generative neural networks have become a popular and active area of research and development. A huge amount of credit for this trend goes to a class of models that we are going to discuss in this chapter. These models are called **generative adversarial networks** (**GANs**) and were introduced in 2014. Ever since the introduction of the basic GAN model, various types of GANs have been, and are being, invented for different applications.

Essentially, a GAN is composed of two neural networks – a **generator** and a **discriminator**. Let's look at an example of the GAN that is used to generate images. For such a GAN, the task of the generator would be to generate realistic-looking fake images, and the task of the discriminator would be to tell the real images apart from the fake images.

In a joint optimization procedure, the generator would ultimately learn to generate such good fake images that the discriminator will essentially be unable to tell them apart from real images. Once such a model is trained, the generator part of it can then be used as a reliable data generator. Besides being used as a generative model for unsupervised learning, GANs have also proven useful in semi-supervised learning.

In the image example, for instance, the features learned by the discriminator model could be used to improve the performance of classification models trained on the image data. Besides semi-supervised learning, GANs have also proven to be useful in reinforcement learning, which is a topic that we will discuss in *Chapter 10, Deep Reinforcement Learning*.

A particular type of GAN that we will focus on in this chapter is the **deep convolutional GAN (DCGAN)**. A DCGAN is essentially an unsupervised **convolution neural network (CNN)** model. Both the generator and the discriminator in a DCGAN are purely *CNNs with no fully connected layers*. DCGANs have performed well in generating realistic images, and they can be a good starting point for learning how to build, train, and run GANs from scratch.

In this chapter, we will first understand the various components within a GAN – the generator and the discriminator models and the joint optimization schedule. We will then focus on building a DCGAN model using PyTorch. Next, we will use an image dataset to train and test the performance of the DCGAN model. We will conclude this chapter by revisiting the concept of style transfer on images and exploring the Pix2Pix GAN model, which can efficiently perform a style transfer on any given pair of images.

We will also learn how the various components of a Pix2Pix GAN model relate to that of a DCGAN model. After finishing this chapter, we will truly understand how GANs work and will be able to build any type of GAN model using PyTorch. This chapter is broken down into the following topics:

- Defining the generator and discriminator networks
- Training a DCGAN using PyTorch
- Using GANs for style transfer

Defining the generator and discriminator networks

As mentioned earlier, GANs are composed of two components – the generator and the discriminator. Both of these are essentially neural networks. Generators and discriminators with different neural architectures produce different types of GANs. For example, DCGANs purely have CNNs as the generator and discriminator. You can find a list of different types of GANs along with their PyTorch implementations at [9.1]

.

For any GAN that is used to generate some kind of real data, the generator usually takes random noise as input and produces an output with the same dimensions as the real data. We call this generated output **fake data**. The discriminator, on the other hand, works as a **binary classifier**. It takes in the generated fake data and the real data (one at a time) as input and predicts whether the input data is real or fake. *Figure 9 .1* shows a diagram of the overall GAN model schematic:

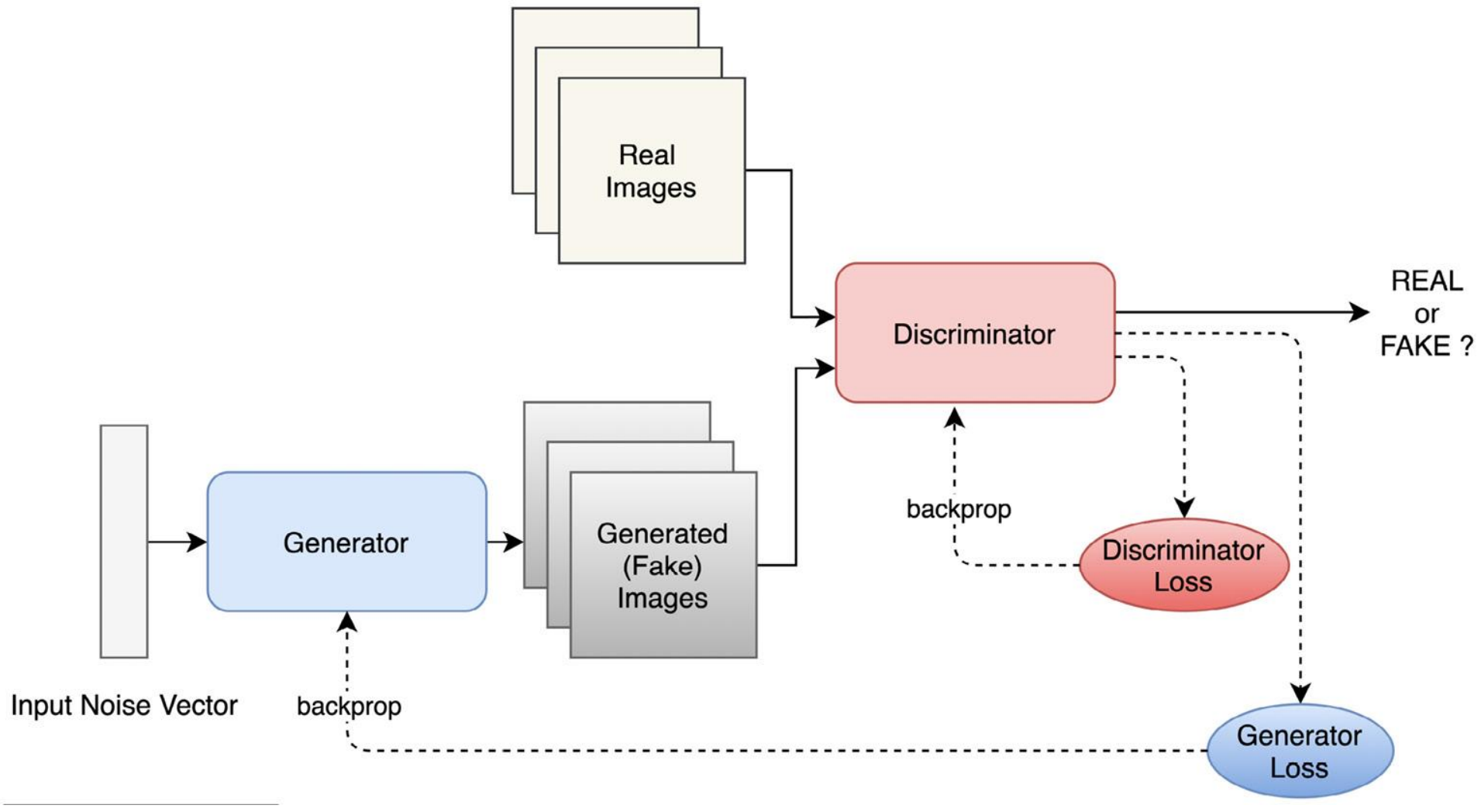


Figure 9 .1 – A GAN schematic

The discriminator network is optimized like any binary classifier, that is, using the binary cross-entropy function. Therefore, the discriminator model's motivation is to correctly classify real images as real and fake images as fake. The generator network has quite the opposite motivation. The generator loss is mathematically expressed as *-log(D(G(x)))*, where *x* is random noise inputted into the generator model, *G*; *G(x)* is the generated fake image by the generator model; and *D(G(x))* is the output probability of the discriminator model, *D*, that is, the probability of the image being real.

Therefore, the generator loss is minimized when the discriminator thinks that the generated fake image is real. Essentially, the generator is trying to fool the discriminator in this joint optimization problem.

In execution, these two loss functions are backpropagated alternatively. That is, at every iteration of training, first, the discriminator is frozen, and the parameters of the generator networks are optimized by backpropagating the gradients from the generator loss.

Then, the tuned generator is frozen while the discriminator is optimized by backpropagating the gradients from the discriminator loss. This is what we call joint optimization. It has also been referred to as being equivalent to a two-player Minimax game in the original GAN paper [9.2] .

Understanding the DCGAN generator and discriminator

For the particular case of DCGANs, let's consider what the generator and discriminator model architectures look like. As already mentioned, both are purely convolutional models. *Figure 9 .2* shows the generator model architecture for a DCGAN:

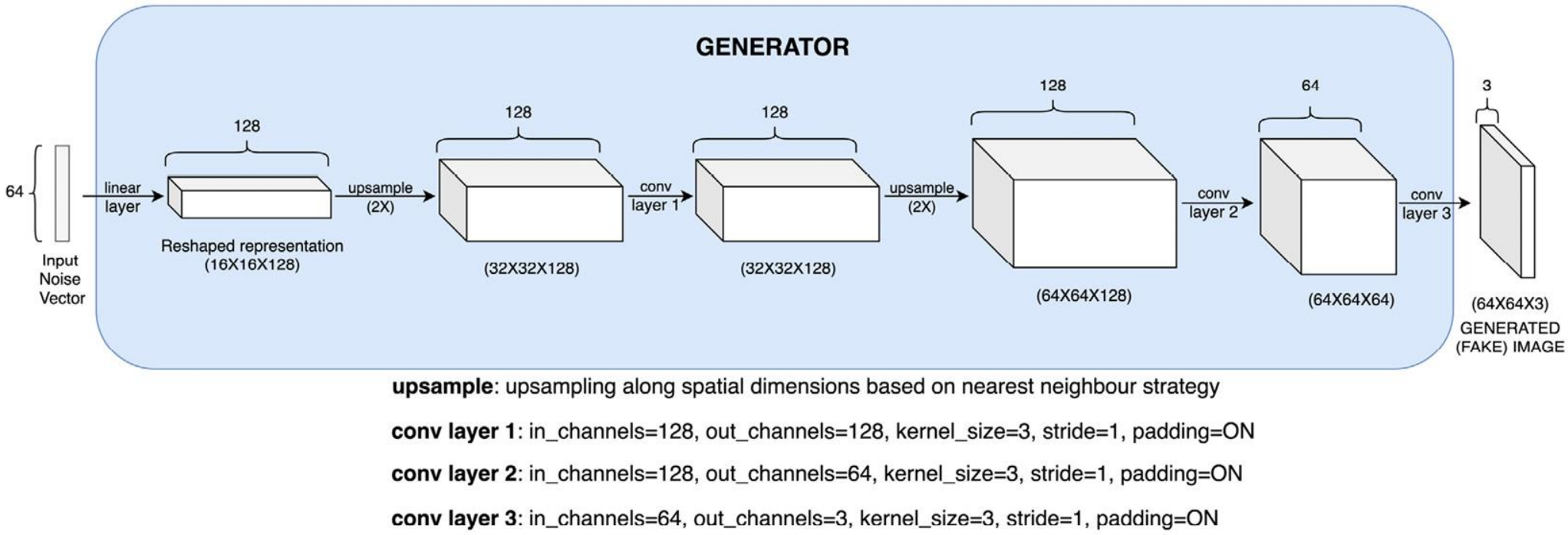


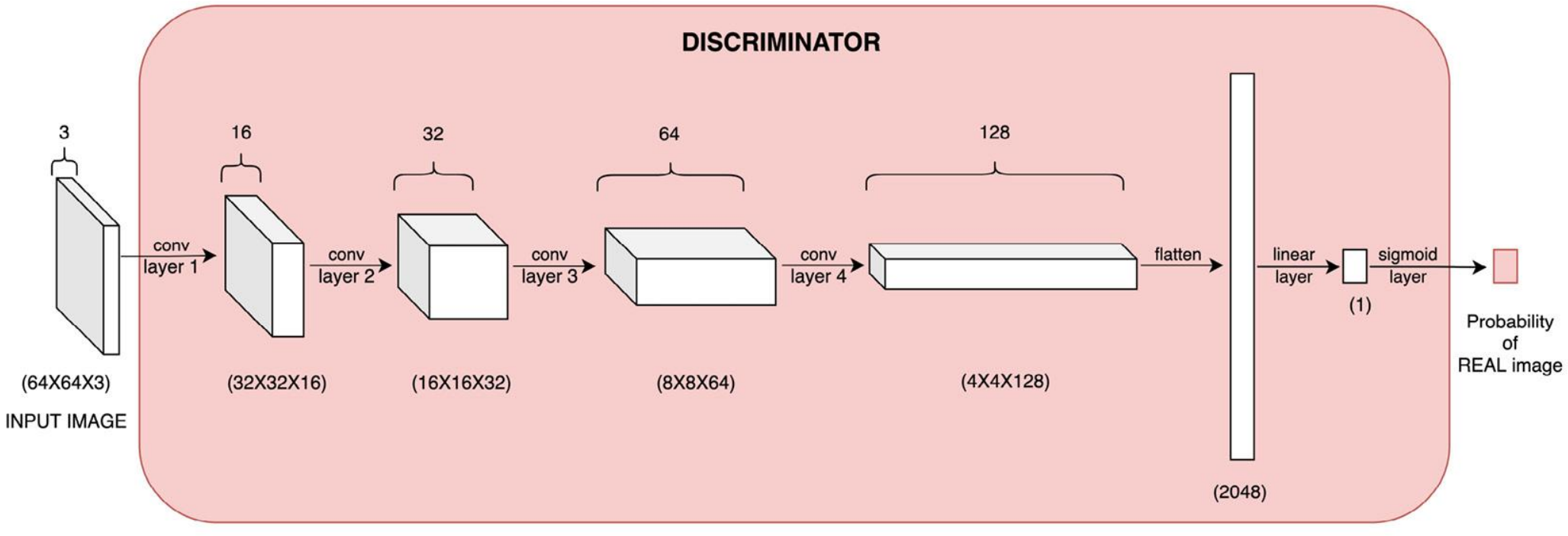
Figure 9 .2 – The DCGAN generator model architecture

First, the random noise input vector of size **64** is reshaped and projected into **128** feature maps of size **16x16** each. This projection is achieved using a linear layer. From there on, a series of upsampling and convolutional layers follow. The first upsampling layer simply transforms the **16x16** feature maps into **32x32** feature maps using the nearest neighbor upsampling strategy.

This is followed by a 2D convolutional layer with a **3x3** kernel size and **128** output feature maps. The **128 32x32** feature maps outputted by this convolutional layer are further upsampled to **64x64**-sized feature maps, which is followed by two **2D** convolutional layers resulting in the generated (fake) RGB image of size **64x64.**

Note
We have omitted the batch normalization and leaky ReLU layers to avoid clutter in the preceding architectural representation. The PyTorch code in the next section will have these details mentioned and explained.

Now that we know what the generator model looks like, let's examine what the discriminator model looks like. *Figure 9 .3* shows the discriminator model architecture:



conv layer 1: in_channels=3, out_channels=16, kernel_size=3, stride=2, padding=ON

conv layer 2: in_channels=16, out_channels=32, kernel_size=3, stride=2, padding=ON

conv layer 3: in_channels=32, out_channels=64, kernel_size=3, stride=2, padding=ON

conv layer 4: in_channels=64, out_channels=128, kernel_size=3, stride=2, padding=ON

Figure 9 .3 – The DCGAN discriminator model architecture

As you can see, a stride of **2** at every convolutional layer in this architecture helps to reduce the spatial dimension, while the depth (that is, the number of feature maps) keeps growing. This is a classic CNN-based binary classification architecture being used here to classify between real images and generated fake images.

Having understood the architectures of the generator and the discriminator network, we can now build the entire DCGAN model based on the schematic in *Figure 9 .1* and train the DCGAN model on an image dataset.

In the next section, we will use PyTorch for this task. We will discuss, in detail, the DCGAN model instantiation, loading the image dataset, jointly training the DCGAN generator and discriminator, and generating sample fake images from the trained DCGAN generator.

Training a DCGAN using PyTorch

In this section, we will build, train, and test a DCGAN model using PyTorch in the form of an exercise. We will use an image dataset to train the model and test how well the generator of the trained DCGAN model performs when producing fake images.

Defining the generator

In the following exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, you can refer to our github repository [9.3] :

1. First, we need to `import` the required libraries, as follows:

```
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torch.autograd import Variable
import torchvision.transforms as transforms
from torchvision.utils import save_image
from torchvision import datasets
```

In this exercise, we only need `torch` and `torchvision` to build the DCGAN model.

1. After importing the libraries, we specify some model hyperparameters, as shown in the following code:

```
num_eps=10
bsize=32
lrate=0.001
lat_dimension=64
image_sz=64
chnls=1
logging_intv=200
```

We will be training the model for `10` epochs with a batch size of `32` and a learning rate of `0.001`. The expected image size is *64x64x3*. `lat_dimension` is the length of the random noise vector, which essentially means that we will draw the random noise from a *64*-dimensional latent space as input to the generator model.

1. Now we define the generator model object. The following code is in direct accordance with the architecture shown in *Figure 9 .2*:

Copy Explain

```
class GANGenerator(nn.Module):
    def __init__(self):
        super(GANGenerator, self).__init__()
        self.inp_sz = image_sz // 4
        self.lin =   nn.Sequential(nn.Linear(lat_dimension, 128 * self.inp_sz ** 2))
        self.bn1 = nn.BatchNorm2d(128)
        self.up1 = nn.Upsample(scale_factor=2)
        self.cn1 = nn.Conv2d(128, 128, 3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(128, 0.8)
        self.rl1 = nn.LeakyReLU(0.2, inplace=True)
        self.up2 = nn.Upsample(scale_factor=2)
        self.cn2 = nn.Conv2d(128, 64, 3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64, 0.8)
        self.rl2 = nn.LeakyReLU(0.2, inplace=True)
        self.cn3 = nn.Conv2d(64, chnls, 3, stride=1, padding=1)
        self.act = nn.Tanh()
```

1. After defining the `_init_` method, we define the `forward` method, which is essentially just calling the layers in a sequential manner:

Copy Explain

```
    def forward(self, x):
        x = self.lin(x)
        x = x.view(x.shape[0], 128, self.inp_sz, self.inp_sz)
        x = self.bn1(x)
        x = self.up1(x)
        x = self.cn1(x)
        x = self.bn2(x)
        x = self.rl1(x)
        x = self.up2(x)
        x = self.cn2(x)
        x = self.bn3(x)
        x = self.rl2(x)
        x = self.cn3(x)
        out = self.act(x)
        return out
```

We have used the explicit layer-by-layer definition in this exercise as opposed to the `nn.Sequential` method; this is because it makes it easier to debug the model if something goes wrong.We can also see the batch normalization and leaky ReLU layers in the code, which are not mentioned in *Figure 9 .2*.

FAQ - Why are we using batch normalisation?
Batch normalization is used after the linear or convolutional layers to both fasten the training process and reduce sensitivity to the initial network weights.
FAQ - Why are we using leaky ReLU?
ReLU might lose all the information for inputs with negative values. A leaky ReLU set with a 0.2 negative slope gives 20% weightage to incoming negative information, which might help us to avoid vanishing gradients during the training of a GAN model.

Next, we will take a look at the PyTorch code to define the discriminator network.

Defining the discriminator

Similar to the generator, we will now define the discriminator model as follows:

1. Once again, the following code is the PyTorch equivalent for the model architecture shown in *Figure 9 .3*:

Copy Explain

```
class GANDiscriminator(nn.Module):
    def __init__(self):
        super(GANDiscriminator, self).__init__()
        def disc_module(ip_chnls, op_chnls, bnorm=True):
            mod = [nn.Conv2d(ip_chnls, op_chnls, 3, 2, 1), nn.LeakyReLU(0.2,
inplace=True),
                   nn.Dropout2d(0.25)] if bnorm:
                mod += [nn.BatchNorm2d(op_chnls, 0.8)]
            return mod
        self.disc_model = nn.Sequential(
            *disc_module(chnls, 16, bnorm=False),
            *disc_module(16, 32),
            *disc_module(32, 64),
            *disc_module(64, 128),
        )
        # width and height of the down-sized image
        ds_size = image_sz // 2 ** 4
        self.adverse_lyr = nn.Sequential(nn.Linear(128 * ds_size ** 2, 1),
nn.Sigmoid())
```

First, we have defined a general discriminator module, which is a cascade of a convolutional layer, an optional batch normalization layer, a leaky ReLU layer, and a dropout layer. In order to build the discriminator model, we repeat this module

sequentially four times – each time with a different set of parameters for the convolutional layer.

The goal is to input a 64x64x3 RGB image and to increase the depth (that is, the number of channels) and decrease the height and width of the image as it is passed through the convolutional layers.

The final discriminator module's output is flattened and passed through the adversarial layer. Essentially, the adversarial layer fully connects the flattened representation to the final model output (that is, a binary output). This model output is then passed through a sigmoid activation function to give us the probability of the image being real (or not fake).

1. The following is the `forward` method for the discriminator, which takes in a 64x64 RGB image as input and produces the probability of it being a real image:

```
    def forward(self, x):
        x = self.disc_model(x)
        x = x.view(x.shape[0], -1)
        out = self.adverse_lyr(x)
        return out
```

1. Having defined the generator and discriminator models, we can now instantiate one of each. We also define our adversarial loss function as the binary cross-entropy loss function in the following code:

```
# instantiate the discriminator and generator models
gen = GANGenerator()
disc = GANDiscriminator()
# define the loss metric
adv_loss_func = torch.nn.BCELoss()
```

The adversarial loss function will be used to define the generator and discriminator loss functions later in the training loop. Conceptually, we are using binary cross-entropy as the loss function because the targets are essentially binary – that is, either real images or fake images. And, binary cross-entropy loss is a well-suited loss function for binary classification tasks.

Loading the image dataset

For the task of training a DCGAN to generate realistic-looking fake images, we are going to use the well-known `MNIST` dataset which contains images of handwritten digits from 0 to 9. By using `torchvision.datasets`, we can directly download the `MNIST` dataset and create a `dataset` and a `dataloader` instance out of it:

Copy Explain

```
# define the dataset and corresponding dataloader
dloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "./data/mnist/", download=True,
        transform=transforms.Compose(
            [transforms.Resize((image_sz, image_sz)),
             transforms.ToTensor(), transforms.Normalize([0.5], [0.5])]),),
batch_size=bsize, shuffle=True,)
```

Here is an example of a real image from the `MNIST` dataset:

Figure 9. 4 – A real image from the MNIST dataset

Dataset citation
[LeCun et al., 1998a] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1999.
Yann LeCun (Courant Institute, NYU) and Corinna Cortes (Google Labs, New York) hold the copyright of the MNIST dataset, which is a derivative work from the original NIST datasets. The MNIST dataset is made available under the terms of the Creative Commons Attribution-Share Alike 3.0 license.

So far, we have defined the model architecture and the data pipeline. Now it is time for us to actually write the DCGAN model training routine, which we will do in the following section.

Training loops for DCGANs

In this section, we will train the DCGAN model:

1. **Defining the optimization schedule:** Before starting the training loop, we will define the optimization schedule for both the generator and the discriminator. We will use the `Adam` optimizer for our model. In the original DCGAN paper [9. 4] , the *beta1* and *beta2* parameters of the Adam optimizer are set to *0.5* and *0.999*, as opposed to the usual *0.9* and *0.999*.

We have retained the default values of *0.9* and *0.999* in our exercise. However, you are welcome to use the exact same values mentioned in the paper for similar results:

Copy Explain

```
# define the optimization schedule for both G and D
opt_gen = torch.optim.Adam(gen.parameters(), lr=lrate)
opt_disc = torch.optim.Adam(disc.parameters(), lr=lrate)
```

1. **Training the generator:** Finally, we can now run the training loop to train the DCGAN. As we will be jointly training the generator and the discriminator, the training routine will consist of both these steps – training the generator model and training the discriminator model – in an alternate fashion. We will begin with training the generator in the following code:

Copy Explain

```
os.makedirs("./images_mnist", exist_ok=True)
for ep in range(num_eps):
    for idx, (images, _) in enumerate(dloader):
        # generate ground truths for real and fake images
        good_img = Variable(torch.FloatTensor(images.shape[0], 1).fill_(1.0),
requires_grad=False)
        bad_img = Variable(torch.FloatTensor(images.shape[0], 1) .fill_(0.0),
requires_grad=False)
        # get a real image
        actual_images = Variable(images.type(torch.FloatTensor))
        # train the generator model
        opt_gen.zero_grad()
        # generate a batch of images based on random noise as input
        noise = Variable(torch.FloatTensor(np.random.normal(0, 1, (images.shape[0],
lat_dimension))))
        gen_images = gen(noise)
        # generator model optimization - how well can it fool the discriminator
        generator_loss = adv_loss_func(disc(gen_images), good_img)
        generator_loss.backward()
        opt_gen.step()
```

In the preceding code, we first generate the ground truth labels for real and fake images. Real images are labeled as `1`, and fake images are labeled as `0`. These labels will serve as the target outputs for the discriminator model, which is a binary classifier.

Next, we load a batch of real images from the MINST dataset loader, and we also use the generator to generate a batch of fake images using random noise as input.

Finally, we define the generator loss as the adversarial loss between the following:

i) The probability of realness of the fake images (produced by the generator model) as predicted by the discriminator model.

ii) The ground truth value of `1`.

Essentially, if the discriminator is fooled to perceive the fake generated image as a real image, then the generator has succeeded in its role, and the generator loss will be low. Once we have formulated the generator loss, we can use it to backpropagate gradients along the generator model in order to tune its parameters.

In the preceding optimization step of the generator model, we left the discriminator model parameters unchanged and simply used the discriminator model for a forward pass.

1. **Training the discriminator:** Next, we will do the opposite, that is, we will retain the parameters of the generator model and train the discriminator model:

Copy Explain

```
        # train the discriminator model
        opt_disc.zero_grad()
        # calculate discriminator loss as average of mistakes(losses) in confusing
real images as fake and vice versa
        actual_image_loss = adv_loss_func(disc(actual_images), good_img)
        fake_image_loss = adv_loss_func(disc(gen_images.detach()), bad_img)
        discriminator_loss = (actual_image_loss + fake_image_loss) / 2
        # discriminator model optimization
        discriminator_loss.backward()
        opt_disc.step()
        batches_completed = ep * len(dloader) + idx
        if batches_completed % logging_intv == 0:
            print(f"epoch number {ep} | batch number {idx} | generator loss =
{generator_loss.item()} \
            | discriminator loss = {discriminator_loss.item()}")
            save_image(gen_images.data[:25], f"images_mnist/{batches_completed}.png",
nrow=5, normalize=True)
```

Remember that we have a batch of both real and fake images. In order to train the discriminator model, we will need both. We define the discriminator loss simply to be the adversarial loss or the binary cross entropy loss as we do for any binary classifier.

We compute the discriminator loss for the batches of both real and fake images, keeping the target values at `1` for the batch of real images and at `0` for the batch of fake images. We then use the mean of these two losses as the final discriminator loss, and use it to backpropagate gradients to tune the discriminator model parameters.

After every few epochs and batches, we log the model's performance results, that is, the generator loss and the discriminator loss. For the preceding code, we should get an output similar to the following:

```
epoch number 0 | batch number 0 | generator loss = 0.683123 | discriminator loss = 0.693203
epoch number 0 | batch number 200 | generator loss = 5.871073 | discriminator loss = 0.032416
epoch number 0 | batch number 400 | generator loss = 2.876508 | discriminator loss = 0.288186
epoch number 0 | batch number 600 | generator loss = 3.705342 | discriminator loss = 0.049239
epoch number 0 | batch number 800 | generator loss = 2.727477 | discriminator loss = 0.542196
epoch number 0 | batch number 1000 | generator loss = 3.382538 | discriminator loss = 0.282721
epoch number 0 | batch number 1200 | generator loss = 1.695523 | discriminator loss = 0.304907
epoch number 0 | batch number 1400 | generator loss = 2.297853 | discriminator loss = 0.655593
epoch number 0 | batch number 1600 | generator loss = 1.397890 | discriminator loss = 0.599436
                                                    ⋮
epoch number 10 | batch number 3680 | generator loss = 1.407570 | discriminator loss = 0.409708
epoch number 10 | batch number 3880 | generator loss = 0.667673 | discriminator loss = 0.808560
epoch number 10 | batch number 4080 | generator loss = 0.793113 | discriminator loss = 0.679659
epoch number 10 | batch number 4280 | generator loss = 0.902015 | discriminator loss = 0.709771
epoch number 10 | batch number 4480 | generator loss = 0.640646 | discriminator loss = 0.321178
epoch number 10 | batch number 4680 | generator loss = 1.235740 | discriminator loss = 0.465171
epoch number 10 | batch number 4880 | generator loss = 0.896295 | discriminator loss = 0.451197
epoch number 10 | batch number 5080 | generator loss = 0.690564 | discriminator loss = 0.285500
```

Figure 9. 5 – DCGAN training logs

Notice how the losses are fluctuating a bit; that generally tends to happen during the training of GAN models due to the adversarial nature of the joint training mechanism. Besides outputting logs, we also save some network-generated images at regular intervals. *Figure 9. 6* shows the progression of those generated images along the first few epochs:

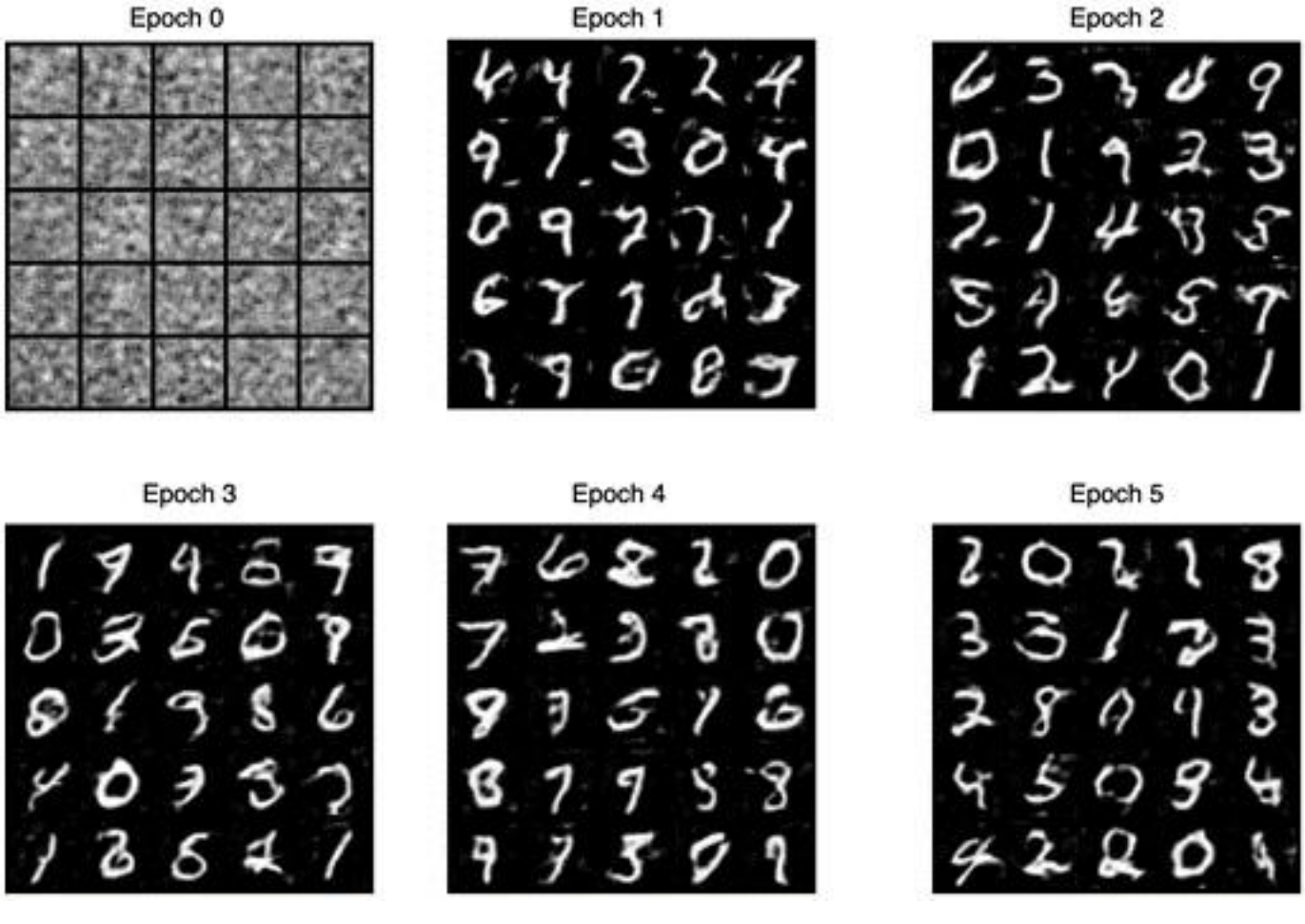


Figure 9. 6 – DCGAN epoch-wise image generation

If we compare the results from the later epochs to the original MNIST images in *Figure 9. 4*, it looks like the DCGAN has learned reasonably well how to generate realistic-looking fake images of handwritten digits.

That is it. We have learned how to use PyTorch to build a DCGAN model from scratch. The original DCGAN paper has a few nuanced details, such as the normal initialization of the layer parameters of the generator and discriminator models, using specific *beta1* and *beta2* values for the Adam optimizers, and more. We have omitted some of those details in the interest of focusing on the main parts of the GAN code. You are encouraged to incorporate those details and see how that changes the results.

Additionally, we have only used the `MNIST` database in our exercise. However, we can use any image dataset to train the DCGAN model. You are encouraged to try out this model on other image datasets. One popular image dataset that is used for DCGAN training is the celebrity faces dataset [9. 5] .

A DCGAN trained with this model can then be used to generate the faces of celebrities who do not exist. *ThisPersonDoesntExist* [9. 6] is one such project that generates the faces of humans that do not exist. Spooky? Yes. That is how powerful DCGANs and GANs, in general, are. Also, thanks to PyTorch, we can now build our own GANs in a few lines of code.

In the next and final section of this chapter, we will go beyond DCGANs and take a brief look at another type of GAN – the `pix2pix` model. The `pix2pix` model can be used to generalize the task of style transfer in images and, more generally, the task of image-to-image translation. We will discuss the architecture of the `pix2pix`

model, its generator and discriminator, and use PyTorch to define the generator and discriminator models. We will also contrast Pix2Pix with a DCGAN in terms of their architecture and implementation.

Using GANs for style transfer

So far, we have only looked at DCGANs in detail. Although there exist hundreds of different types of GAN models already, and many more are in the making, some of the well-known GAN models include the following:

GAN

DCGAN

Pix2Pix

CycleGAN

SuperResolutionGAN (SRGAN)

Context encoders

Text-2-Image

LeastSquaresGAN (LSGAN)

SoftmaxGAN

WassersteinGAN

Each of these GAN variants differ by either the application they are catering to, their underlying model architecture, or due to some tweaks in their optimization strategy, such as modifying the loss function. For example, SRGANs are used to enhance the resolution of a low-resolution image. The CycleGAN uses two generators instead of one, and the generators consist of ResNet-like blocks. The LSGAN uses the mean square error as the discriminator loss function instead of the usual cross-entropy loss used in most GANs.

It is impossible to discuss all of these GAN variants in a single chapter or even a book. However, in this section, we will explore one more type of GAN model that relates to both the DCGAN model discussed in the previous section and the neural style transfer model discussed in *Chapter 8 , Neural Style Transfer* .

This special type of GAN generalizes the task of style transfer between images and, furthermore, provides a general image-to-image translation framework. It is called **Pix2Pix**, and we will briefly explore its architecture and the PyTorch implementation of its generator and discriminator components.

Understanding the pix2pix architecture

In *Chapter 8 , Neural Style Transfer*, you may recall that a fully trained neural style transfer model only works on a given pair of images. Pix2Pix is a more general model that can transfer style between any pair of images once trained successfully. In fact, the model goes beyond just style transfer and can be used for any image-to-image translation application, such as background masking, color palette completion, and more.

Essentially, Pix2Pix works like any GAN model. There is a generator and a discriminator involved. Instead of taking in random noise as input and generating an image, as shown in *Figure 9. 1*, the generator in a `pix2pix` model takes in a real image as input and tries to generate a translated version of that image. If the task at hand is style transfer, then the generator will try to generate a style-transferred image.

Subsequently, the discriminator now looks at a pair of images instead of just a single image, as was the case in *Figure 9. 1*. A real image and its equivalent translated image is fed as input to the discriminator. If the translated image is a genuine one, then the discriminator is supposed to output *1*, and if the translated image is generated by the generator, then the discriminator is supposed to output *0*. *Figure 9. 7* shows the schematic for a `pix2pix` model:

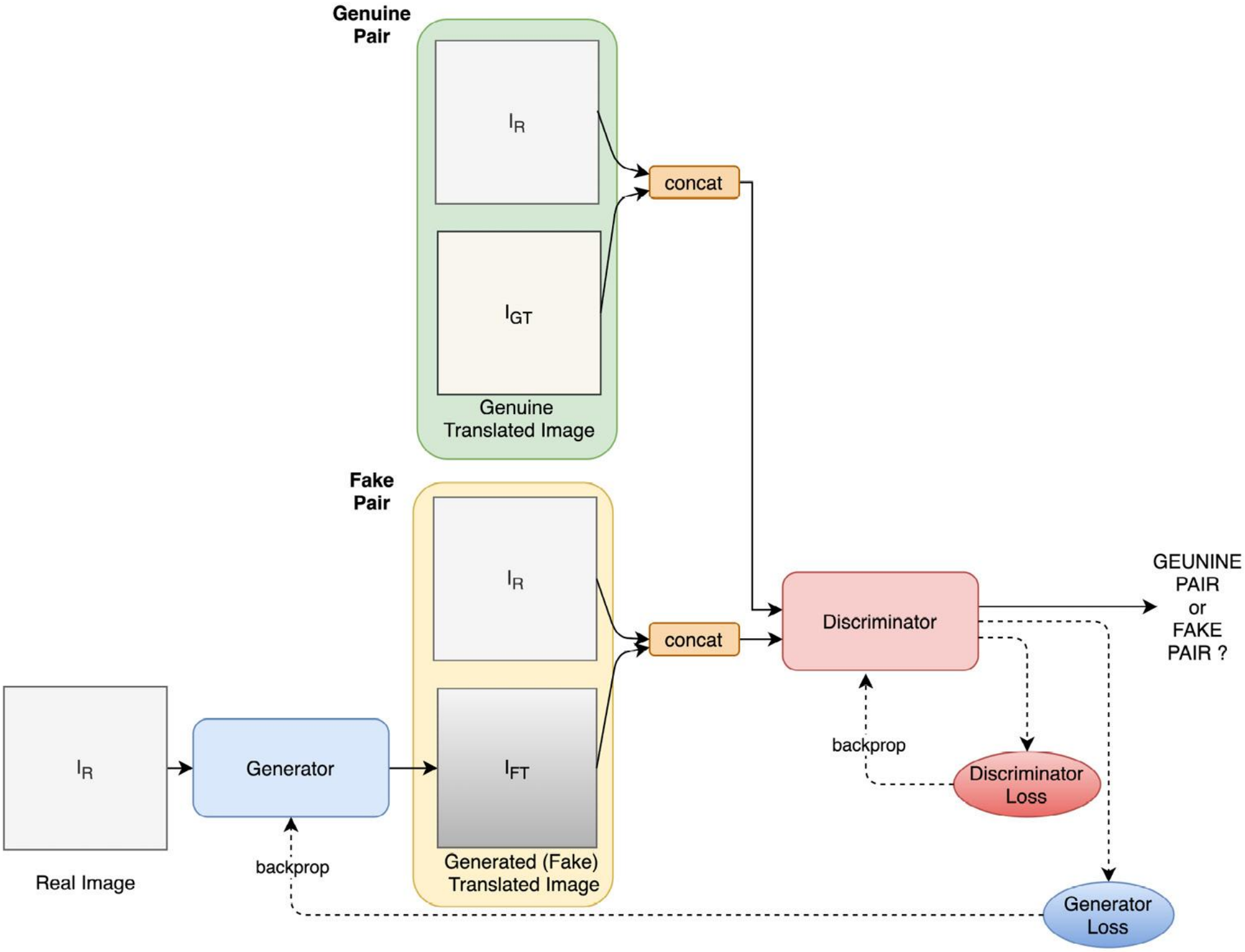


Figure 9. 7 – A Pix2Pix model schematic

Figure 9. 7 shows significant similarities to *Figure 9. 1*, which implies that the underlying idea is the same as a regular GAN. The only difference is that the real or fake question to the discriminator is posed on a pair of images as opposed to a single image.

Exploring the Pix2Pix generator

The generator sub-model used in the `pix2pix` model is a well-known CNN used for image segmentation – the **UNet**. *Figure 9. 8* shows the architecture of the UNet, which is used as a generator for the `pix2pix` model:

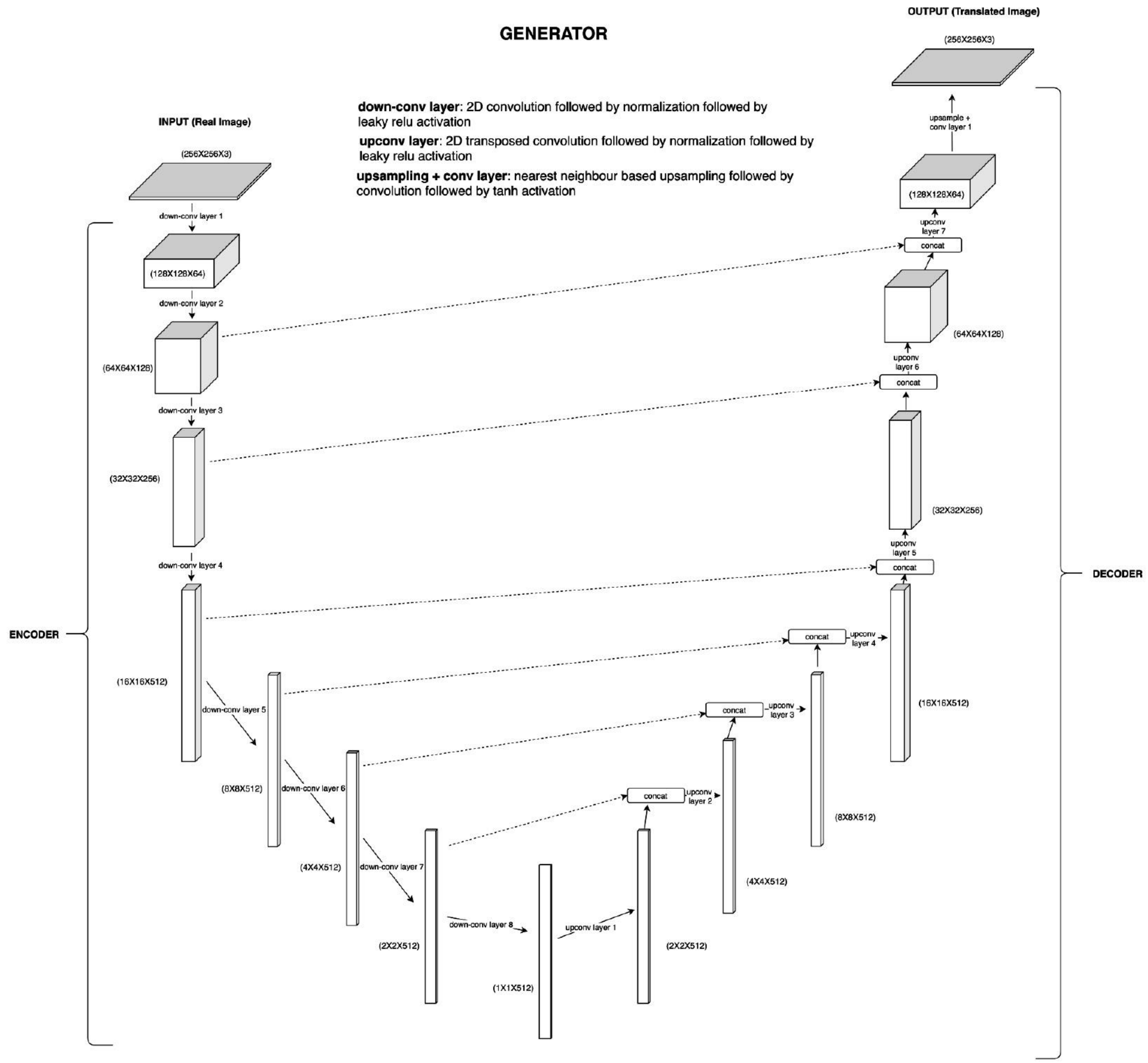


Figure 9. 8 – The Pix2Pix generator model architecture

Firstly, the name, UNet, comes from the *U* shape of the network, as is made evident from the preceding diagram. There are two main components in this network, as follows:

From the upper-left corner to the bottom lies the encoder part of the network, which encodes the **256x256** RGB input image into a **512**-sized feature vector.

From the upper-right corner to the bottom lies the decoder part of the network, which generates an image from the embedding vector of size **512**.

A key property of UNet is the **skip connections**, that is, the concatenation of features (along the depth dimension)from the encoder section to the decoder section, as shown by the dotted arrows in *Figure 9.8*

FAQ - Why do we have encoder-decoder skip-connections in U-Net?
Using features from the encoder section helps the decoder to better localize the high-resolution information at each upsampling step.

Essentially, the encoder section is a sequence of down-convolutional blocks, where each down-convolutional block is itself a sequence of a 2D convolutional layer, an instance normalization layer, and a leaky ReLU activation. Similarly, the decoder section consists of a sequence of up-convolutional blocks, where each block is a sequence of a 2D-transposed convolutional layer, an instance normalization layer, and a ReLU activation layer.

The final part of this UNet generator architecture is a nearest neighbor-based upsampling layer, followed by a 2D convolutional layer, and, finally, a `tanh` activation. Let's now look at the PyTorch code for the UNet generator:

1. Here is the equivalent PyTorch code for defining the UNet-based generator architecture:

Copy Explain

```
class UNetGenerator(nn.Module):
    def __init__(self, chnls_in=3, chnls_op=3):
        super(UNetGenerator, self).__init__()
        self.down_conv_layer_1 = DownConvBlock(chnls_in, 64, norm=False)
        self.down_conv_layer_2 = DownConvBlock(64, 128)
        self.down_conv_layer_3 = DownConvBlock(128, 256)
        self.down_conv_layer_4 = DownConvBlock(256, 512, dropout=0.5)
        self.down_conv_layer_5 = DownConvBlock(512, 512, dropout=0.5)
        self.down_conv_layer_6 = DownConvBlock(512, 512, dropout=0.5)
        self.down_conv_layer_7 = DownConvBlock(512, 512, dropout=0.5)
        self.down_conv_layer_8 = DownConvBlock(512, 512, norm=False, dropout=0.5)
        self.up_conv_layer_1 = UpConvBlock(512, 512, dropout=0.5)
        self.up_conv_layer_2 = UpConvBlock(1024, 512, dropout=0.5)
        self.up_conv_layer_3 = UpConvBlock(1024, 512, dropout=0.5)
        self.up_conv_layer_4 = UpConvBlock(1024, 512, dropout=0.5)
        self.up_conv_layer_5 = UpConvBlock(1024, 256)
        self.up_conv_layer_6 = UpConvBlock(512, 128)
        self.up_conv_layer_7 = UpConvBlock(256, 64)
        self.upsample_layer = nn.Upsample(scale_factor=2)
        self.zero_pad = nn.ZeroPad2d((1, 0, 1, 0))
        self.conv_layer_1 = nn.Conv2d(128, chnls_op, 4, padding=1)
        self.activation = nn.Tanh()
```

As you can see, there are 8 down-convolutional layers and 7 up-convolutional layers. The up-convolutional layers have two inputs, one from the previous up-convolutional layer output and another from the equivalent down-convolutional layer output, as shown by the dotted lines in *Figure 9. 7*.

1. We have used the `UpConvBlock` and `DownConvBlock` classes to define the layers of the UNet model. The following is the definition of these blocks, starting with the `UpConvBlock` class:

Copy Explain

```
class UpConvBlock(nn.Module):
    def __init__(self, ip_sz, op_sz, dropout=0.0):
        super(UpConvBlock, self).__init__()
        self.layers = [
            nn.ConvTranspose2d(ip_sz, op_sz, 4, 2, 1),
            nn.InstanceNorm2d(op_sz), nn.ReLU(),]
        if dropout:
            self.layers += [nn.Dropout(dropout)]
    def forward(self, x, enc_ip):
        x = nn.Sequential(*(self.layers))(x)
        op = torch.cat((x, enc_ip), 1)
        return op
```

The transpose convolutional layer in this up-convolutional block consists of a 4x4 kernel with a stride of 2 steps, which essentially doubles the spatial dimensions of its output compared to the input.

In this transpose convolution layer, the 4x4 kernel is passed through every other pixel (due to a stride of 2) in the input image. At each pixel, the pixel value is multiplied with each of the 16 values in the 4x4 kernel.

The overlapping values of the kernel multiplication results across the image are then summed up, resulting in an output twice the length and twice the breadth of the input image. Also, in the preceding `forward` method, the concatenation operation is performed after the forward pass is done via the up-convolutional block.

1. Next, here is the PyTorch code for defining the `DownConvBlock` class:

```
class DownConvBlock(nn.Module):
    def __init__(self, ip_sz, op_sz, norm=True, dropout=0.0):
        super(DownConvBlock, self).__init__()
        self.layers = [nn.Conv2d(ip_sz, op_sz, 4, 2, 1)]
        if norm:
            self.layers.append(nn.InstanceNorm2d(op_sz))
        self.layers += [nn.LeakyReLU(0.2)]
        if dropout:
            self.layers += [nn.Dropout(dropout)]
    def forward(self, x):
        op = nn.Sequential(*(self.layers))(x)
        return op
```

The convolutional layer inside the down-convolutional block has a kernel of size 4x4, a stride of 2, and the padding is activated. Because the stride value is 2, the output of this layer is half the spatial dimensions of its input.

A leaky ReLU activation is also used for similar reasons as DCGANs – the ability to deal with negative inputs, which also helps with alleviating the vanishing gradients problem.

So far, we have seen the `__init__` method of our UNet-based generator. The `forward` method is pretty straightforward hereafter:

```
    def forward(self, x):
        enc1 = self.down_conv_layer_1(x)
        enc2 = self.down_conv_layer_2(enc1)
        enc3 = self.down_conv_layer_3(enc2)
        enc4 = self.down_conv_layer_4(enc3)
        enc5 = self.down_conv_layer_5(enc4)
        enc6 = self.down_conv_layer_6(enc5)
        enc7 = self.down_conv_layer_7(enc6)
        enc8 = self.down_conv_layer_8(enc7)
        dec1 = self.up_conv_layer_1(enc8, enc7)
        dec2 = self.up_conv_layer_2(dec1, enc6)
        dec3 = self.up_conv_layer_3(dec2, enc5)
        dec4 = self.up_conv_layer_4(dec3, enc4)
        dec5 = self.up_conv_layer_5(dec4, enc3)
        dec6 = self.up_conv_layer_6(dec5, enc2)
        dec7 = self.up_conv_layer_7(dec6, enc1)
        final = self.upsample_layer(dec7)
        final = self.zero_pad(final)
        final = self.conv_layer_1(final)
        return self.activation(final)
```

Having discussed the generator part of the `pix2pix` model, let's take a look at the discriminator model as well.

Exploring the Pix2Pix discriminator

The discriminator model, in this case, is also a binary classifier – just as it was for the DCGAN. The only difference is that this binary classifier takes in two images as inputs. The two inputs are concatenated along the depth dimension. *Figure 9. 9* shows the discriminator model's high-level architecture:

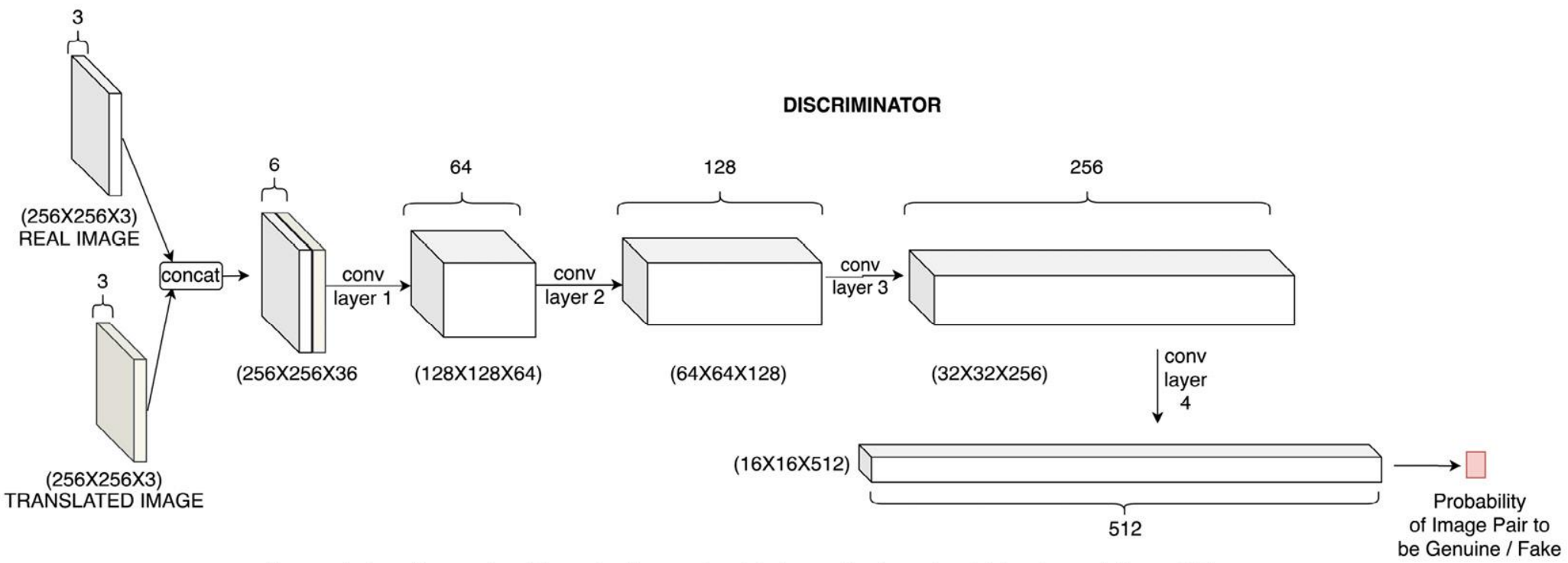


Figure 9. 9 – The Pix2Pix discriminator model architecture

It is a CNN where the last 3 convolutional layers are followed by a normalization layer as well as a leaky ReLU activation. The PyTorch code to define this discriminator model will be as follows:

```
class Pix2PixDiscriminator(nn.Module):
    def __init__(self, chnls_in=3):
        super(Pix2PixDiscriminator, self).__init__()
        def disc_conv_block(chnls_in, chnls_op, norm=1):
            layers = [nn.Conv2d(chnls_in, chnls_op, 4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(chnls_op))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
        self.lyr1 = disc_conv_block(chnls_in * 2, 64, norm=0)
        self.lyr2 = disc_conv_block(64, 128)
        self.lyr3 = disc_conv_block(128, 256)
        self.lyr4 = disc_conv_block(256, 512)
```

As you can see, the 4 convolutional layers subsequently double the depth of the spatial representation at each step. Layers 2, 3, and 4 have added normalization layers after the convolutional layer, and a leaky ReLU activation with a negative slope of 20% is applied at the end of every convolutional block. Finally, here is the `forward` method of the discriminator model class in PyTorch:

Copy Explain

```
    def forward(self, real_image, translated_image):
        ip = torch.cat((real_image, translated_image), 1)
        op = self.lyr1(ip)
        op = self.lyr2(op)
        op = self.lyr3(op)
        op = self.lyr4(op)
        op = nn.ZeroPad2d((1, 0, 1, 0))(op)
        op = nn.Conv2d(512, 1, 4, padding=1)(op)
        return op
```

First, the input images are concatenated and passed through the four convolutional blocks and finally led into a single binary output that tells us the probability of the pair of images being genuine or fake (that is, generated by the generator model). In this way, the `pix2pix` model is trained at runtime so that the generator of the `pix2pix` model can take in any image as input and apply the image translation function that it has learned during training.

The `pix2pix` model will be considered successful if the generated fake-translated image is difficult to tell apart from a genuine translated version of the original image.

This concludes our exploration of the `pix2pix` model. In principle, the overall model schematic for Pix2Pix is quite similar to that of the DCGAN model. The discriminator network for both of these models is a CNN-based binary classifier. The generator network for the `pix2pix` model is a slightly more complex architecture inspired by the UNet image segmentation model.

Overall, we have been able to both successfully define the generator and discriminator models for DCGAN and Pix2Pix using PyTorch, and understand the inner workings of these two GAN variants.

After finishing this section, you should be able to get started with writing PyTorch code for the many other GAN variants out there. Building and training various GAN models using PyTorch can be a good learning experience and certainly is a fun exercise. We encourage you to use the information from this chapter to work on your own GAN projects using PyTorch.

Summary

GANs have been an active area of research and development in recent years, ever since their inception in 2014. This chapter was an exploration of the concepts behind GANs, including the components of GANs, namely, the generator and the discriminator. We discussed the architectures of each of these components and the overall schematic of a GAN model.

In the next chapter, we will go a step further in our pursuit of generative models. We will explore how to generate image from text using cutting-edge deep learning techniques.

Previous Chapter

Next Chapter