**Introduction to GANs**

**General Adversarial Networks**

This is a picture of a cat. However, it is quite a special cat in that it does not exist. This image comes from the website thesecatsdonotexist-dot-com where we can find infinitely many non-existent cats. They were all artificially generated using a technique known as Generative Adversarial Networks, or GANs for short. GANs are generative models able to create completely new data samples similar to the training data they are given.

**Pokemon Sprites Dataset**

Throughout this chapter, we will be working with the Pokemon Sprites dataset, available from the PokeAPI. It consists of about 1300 sprite images of Pokemons, animal-like creatures from a popular Japanese video game. Our goal is to use GANs to generate completely new Pokemons!

**GANs architecture**

Let's discuss how GANs work. Their architecture contains a neural network called a generator. We can think of it as a fraudster trying to produce forged paintings.

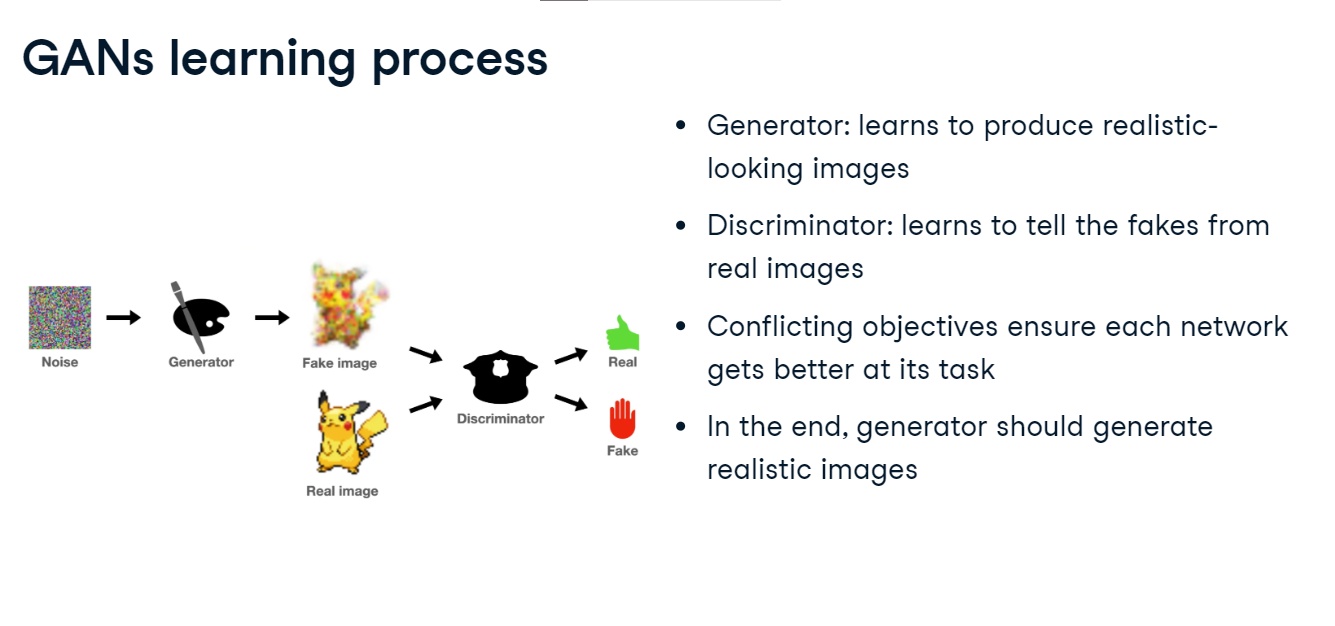
The Generator model receives random noise as input, and produces an image, in our case, a Pokemon sprite. The noise is a tensor of random values drawn from a standard normal distribution.

At this point, a second neural network called the discriminator enters the scene. We can think of the discriminator as the police officer attempting to catch art forgers.

Its job is to distinguish between real and fake images.

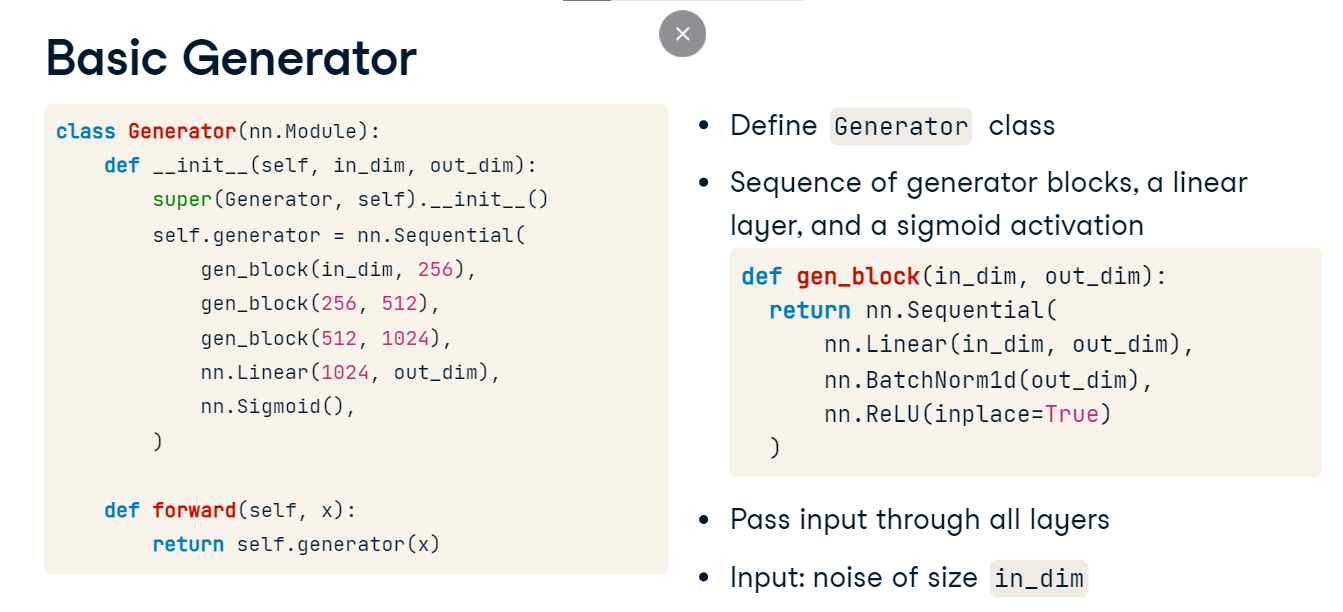
**GANs learning process**

The generator and the discriminator are trained in tandem but with conflicting objectives. This is referred to as adversarial training. The generator learns to produce realistic-looking images that would fool the discriminator, while the discriminator learns to tell the increasingly better fakes from real images. These conflicting goals of the two networks should ensure that each would gradually become better at its task during training. In the end, the generator will hopefully be able to generate realistic images.



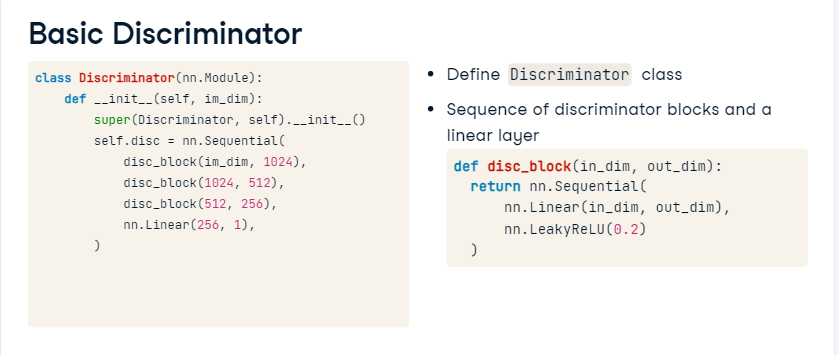
**Basic Generator**

Let's build a basic generator. We start by defining the Generator class. In the init method, we define a sequential network consisting of three generator blocks produced using a custom gen-block function. Each generator block is a linear layer followed by batch normalization and ReLU activation. Notice how with each block we increase the size of the feature maps to go from the small input noise to the large output image. The specific numbers of neurons in each layer are chosen arbitrarily here. After the generator blocks, we append a linear layer and a sigmoid activation. In the forward method, we pass the input through the sequential network we defined. This generator will take as input a random noise vector of size in-dim, and produce the output image of size out-dim.



**Basic Discriminator**

Let's turn our attention to the discriminator now. The concept is quite similar. We start by defining the Discriminator class. Next, we define the sequential network, this time consisting of discriminator blocks created using a custom disc-underscore-block function. Each discriminator block consists of a single linear layer followed by a leaky ReLU activation. Notice how the first discriminator block maps the input to size 1024, while all the subsequent blocks decrease the size of the feature map, until we arrive at a single number in the last linear layer. In the forward method, we pass the input through all the layers. This discriminator will take the image of size in-underscore-dim as input, and will produce the output of size 1: a single prediction whether the input is a real or a fake image.



**Deep Convolutional GAN**

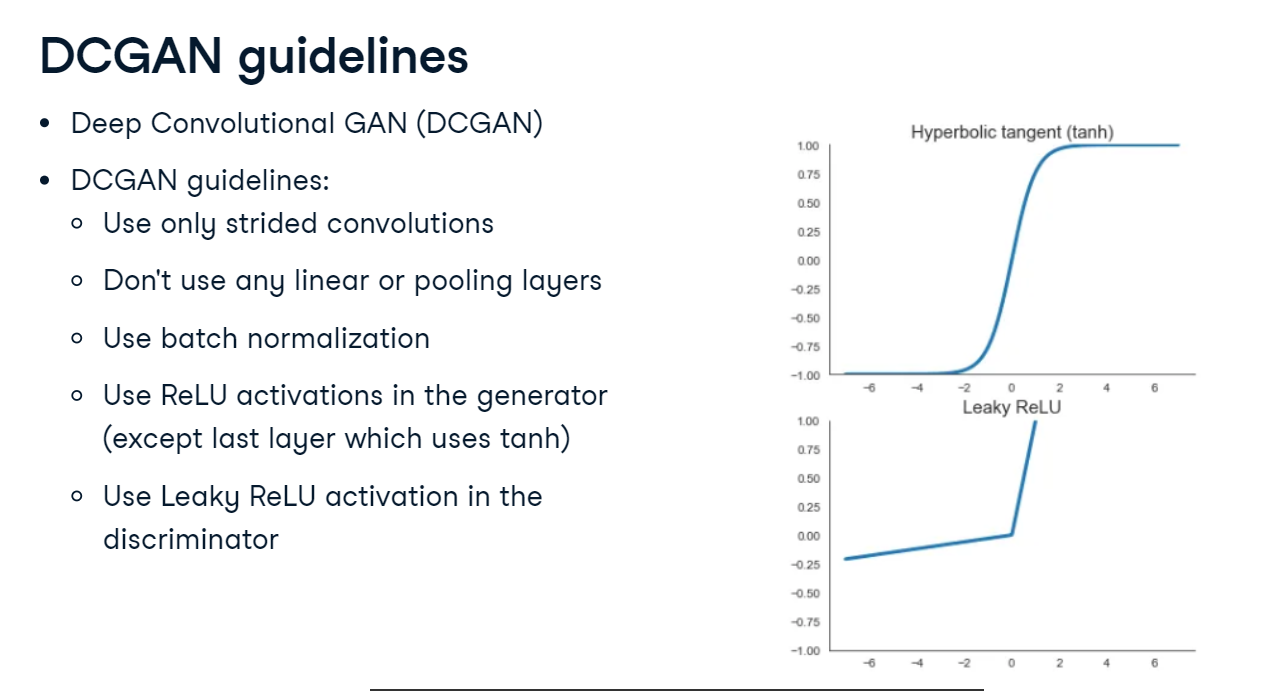
Convolutional layers provide better results when processing images than basic linear layers. Let's learn to use them in GANs!

**Deep Convolutional GAN intuition**

To make a GAN more effective for image data, we could replace the linear layers in the discriminator with convolutional layers. In the generator, in order to upsample feature maps, we could use transposed convolutional layers which we have already seen in the U-Net architecture for semantic segmentation. Unfortunately, it's not that simple. Training GANs is often unstable, and simply swapping linear layers for convolutions is not enough, as more adjustments are needed.

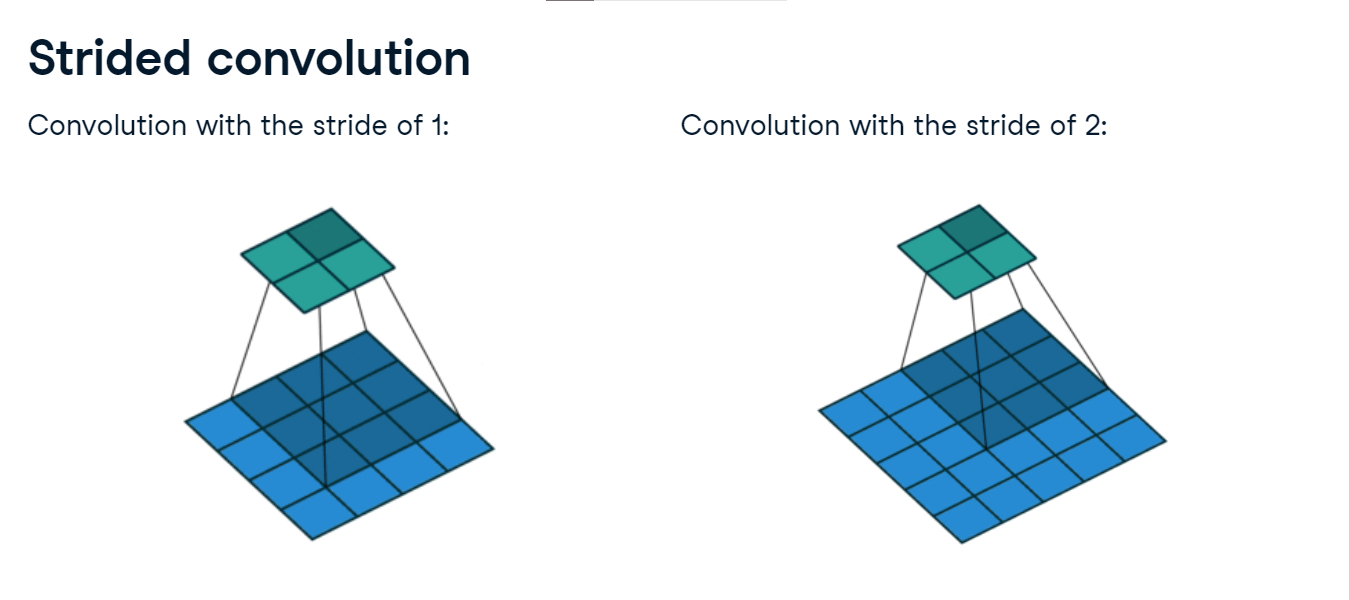
**DCGAN guidelines**

Deep Convolutional GAN, or DCGAN for short, is a famous GAN architecture making use of convolutions. In order to stabilize the training, DCGAN authors suggest following some guidelines. Only strided convolutions are used, which we will discuss shortly. There are no linear or pooling layers, but batch normalization is employed after the convolutions. In the generator, ReLU activations are applied, except for the final layer which uses a tanh activation. Throughout the discriminator, Leaky ReLU activations are used. We will see how to implement these guidelines in practice in a moment. First, let's discuss strided convolutions.



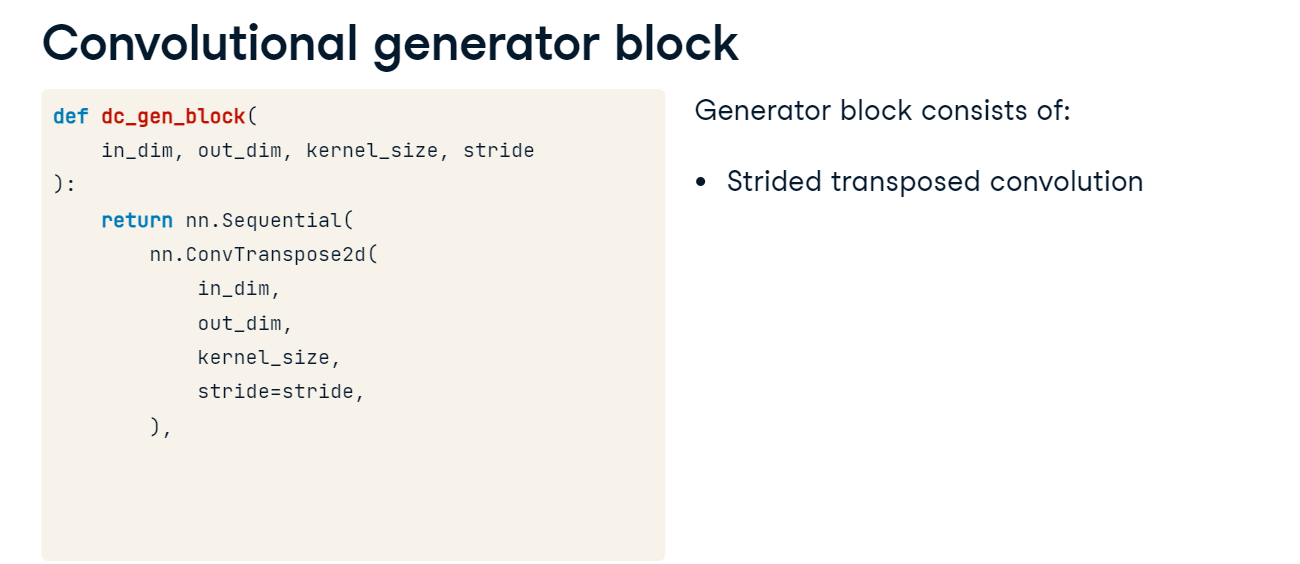
**Strided convolution**

A typical convolution is of stride one. This means that as the kernel slides over the input feature map, it shifts by one pixel at a time. Convolutions with any stride above one are referred to as strided. With a stride of two, for example, the kernel shifts two pixels at a time, both left and down. In PyTorch, we can set the stride of a convolution by passing the stride argument to nn-dot-Conv2d.

****

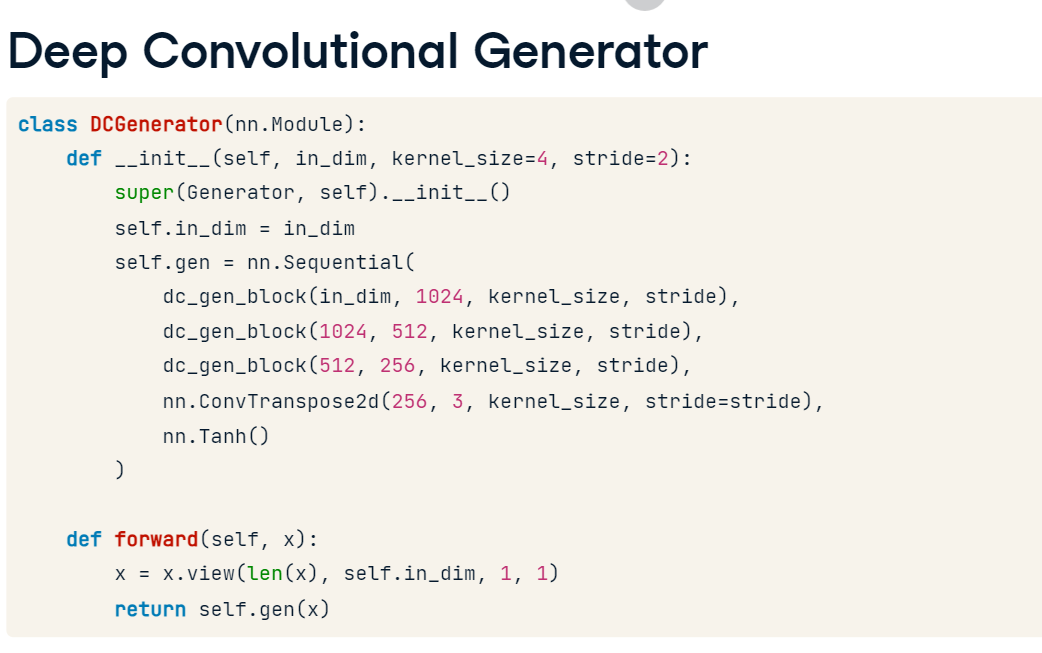
**Convolutional generator block**

Just like with the basic GAN before, we will use custom generator and discriminator block functions to define our GAN. The generator block will consist of a transposed convolution to which we will pass a stride parameter, a batch normalization layer, and a ReLU activation.



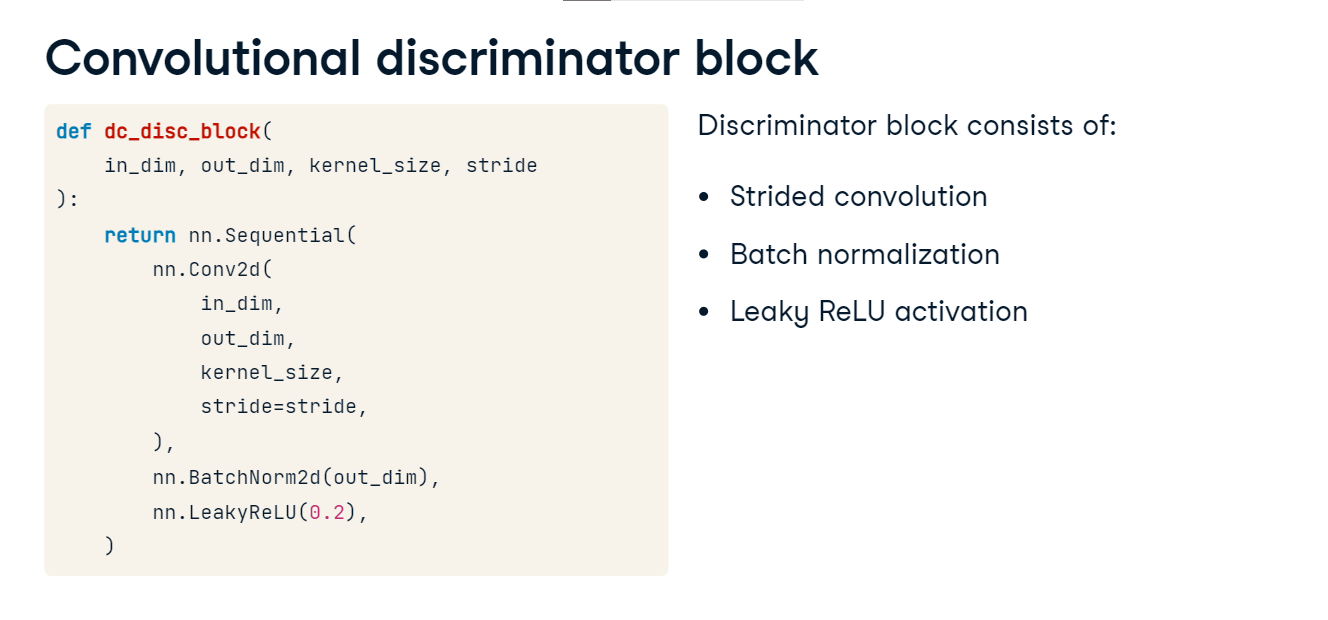
**Deep Convolutional Generator**

Let's define the generator. As arguments, it accepts the input noise size, in-underscore-dim, kernel size, and stride. In the init method, we define a sequential block consisting of three generator blocks followed by a transposed convolution that produces three feature maps, corresponding to the three color channel of the generated image. Finally, we add a tanh activation. In the forward method, before we pass the input to the generator's sequential block, we reshape it with the view method. We make it a tensor of shape len-x, which corresponds to the batch size, by the size of the input noise, by one by one. This reshaping converts the one-dimensional noise vector into a shape compatible with the subsequent convolutional layers in the network.

****

**Convolutional discriminator block**

Let's look at the discriminator now. First, we define the custom dc-disc-block function. The discriminator block will consist of a strided convolution, a batch norm layer, and a leaky ReLU activation.



**Deep Convolutional Discriminator**

As usual, in the init method, we define the sequential block. It consists of two discriminator blocks we have defined earlier, followed by a convolutional layer that produces the output of size one. Recall this corresponds to the discriminator's prediction of whether its input is a real or a fake image. In the forward method, we pass the input to the discriminator's sequential block. Before we return its output, we reshape it with the view method to len-x, corresponding to the batch size by -1 in order to flatten the output of the convolutional layer.

