**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.

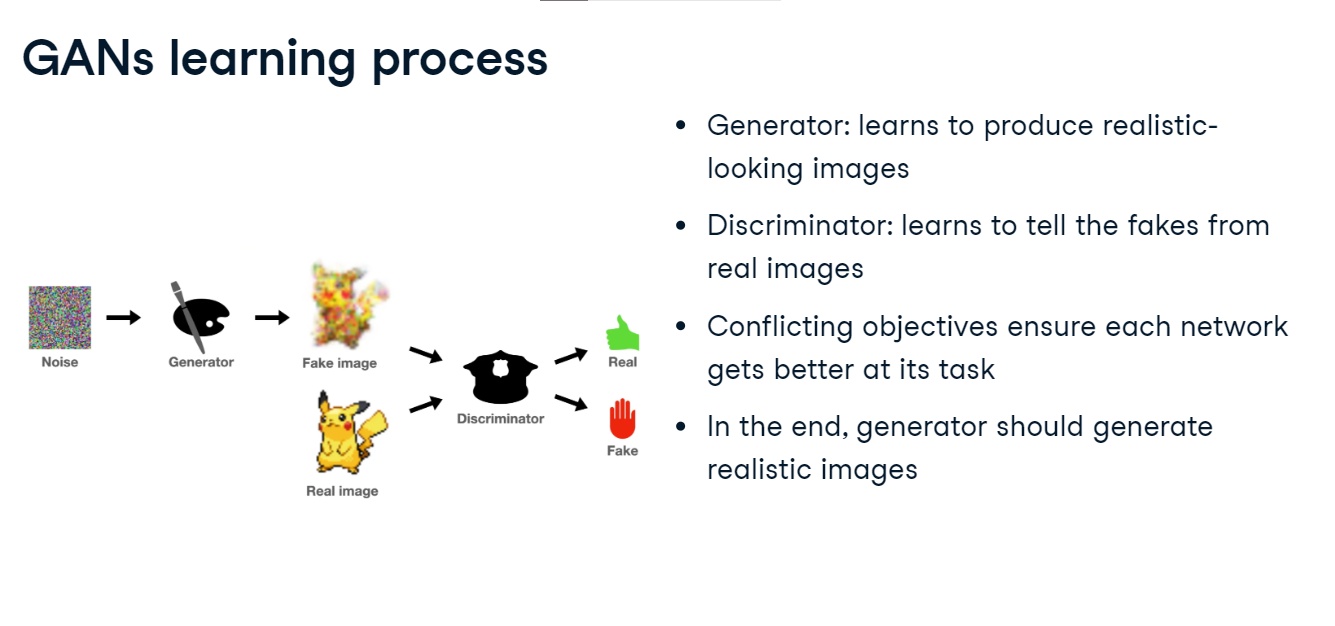
The Generator model receives random noise as input, and produces an image, in our case, a Pokemon sprite. The noise is a tensor of random values drawn from a standard normal distribution.

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.

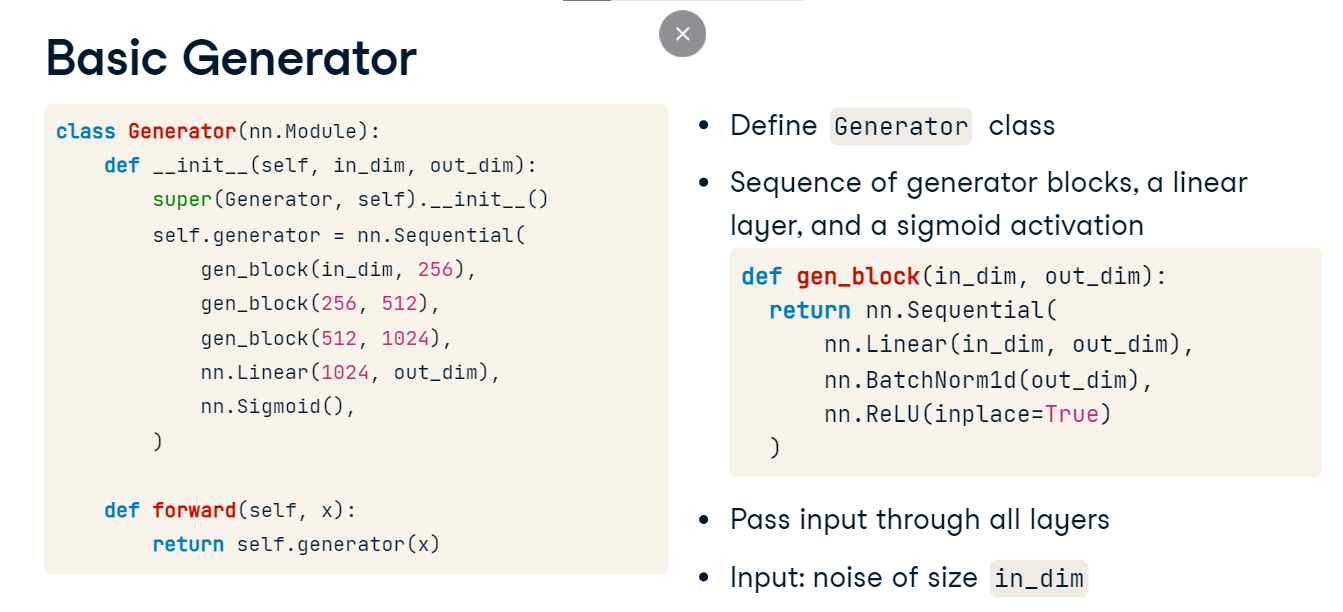
**GANs learning process**

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.



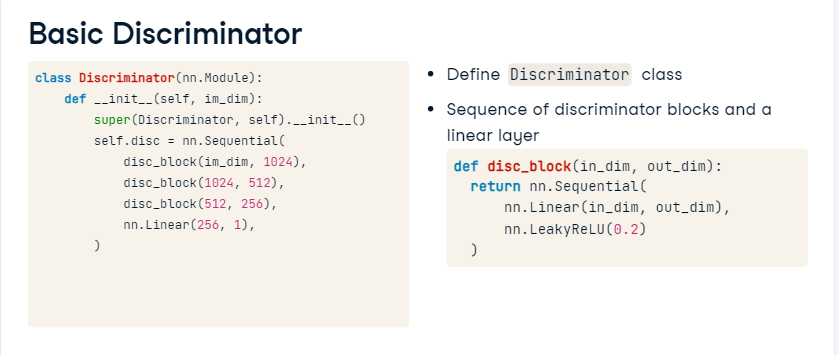
**Basic Generator**

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**

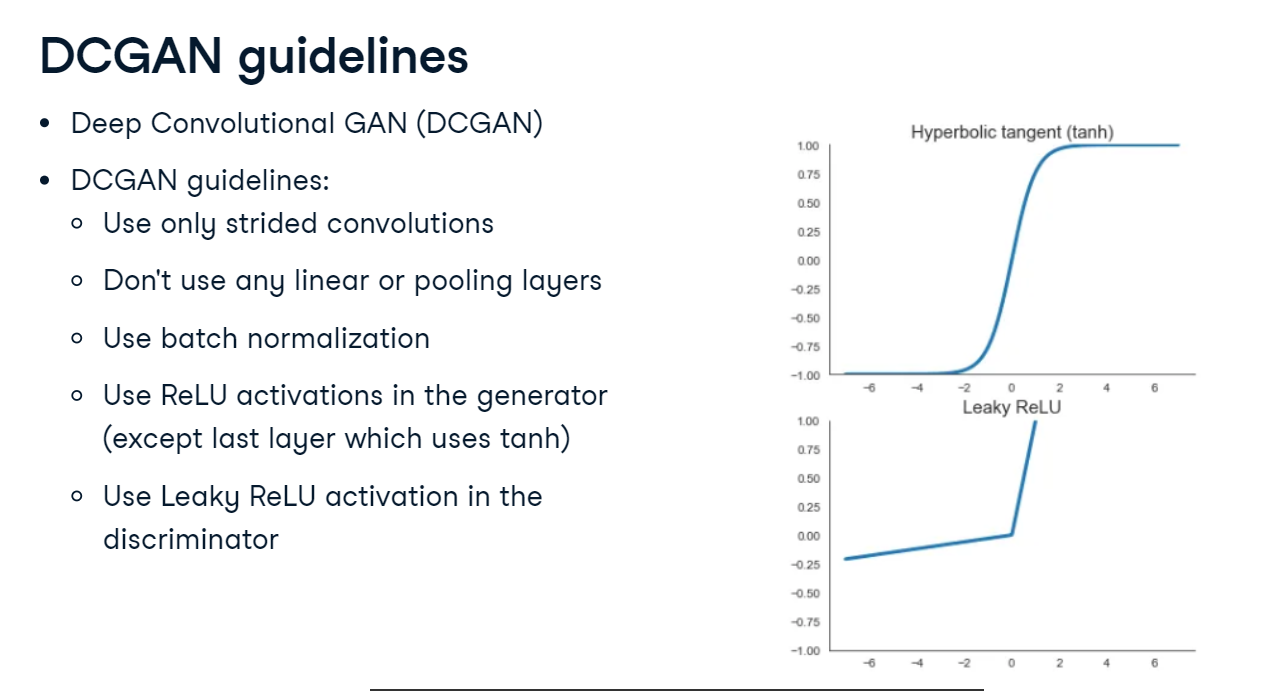
Convolutional layers provide better results when processing images than basic linear layers. Let's learn to use them in GANs!

**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.

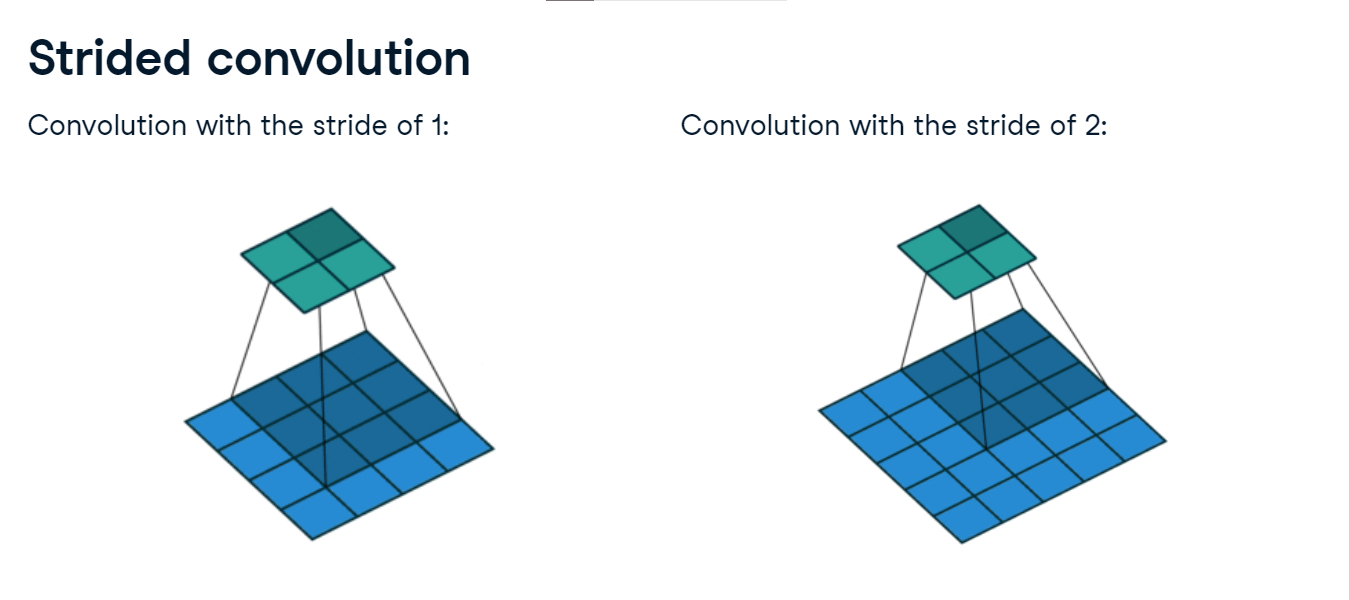
**DCGAN guidelines**

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.



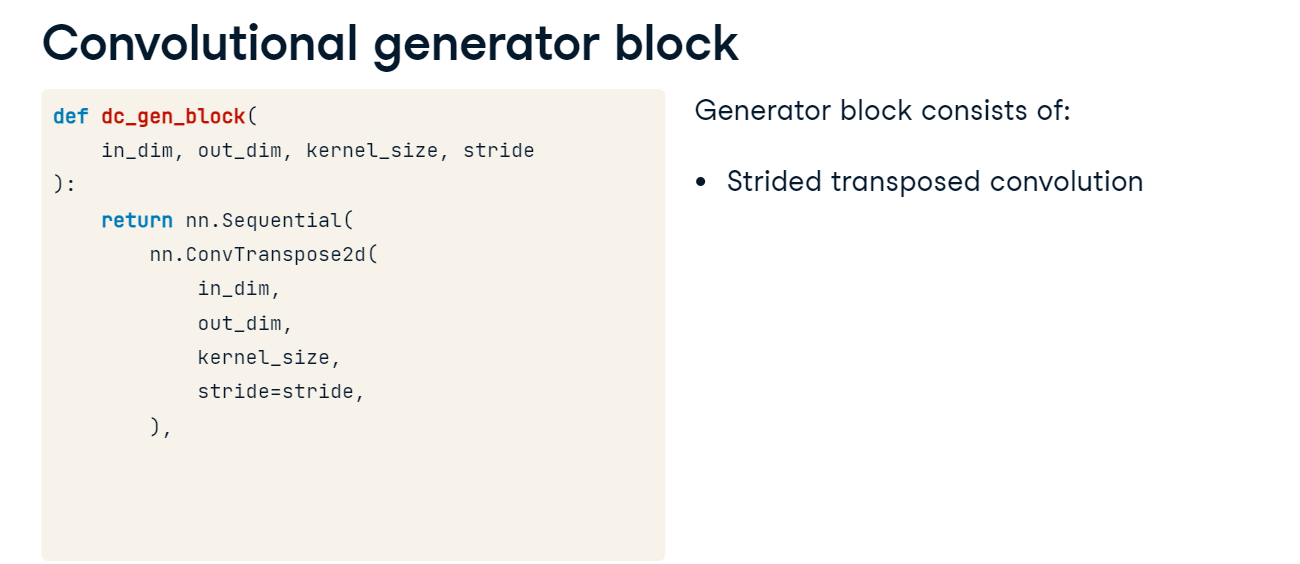
**Strided convolution**

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.

****

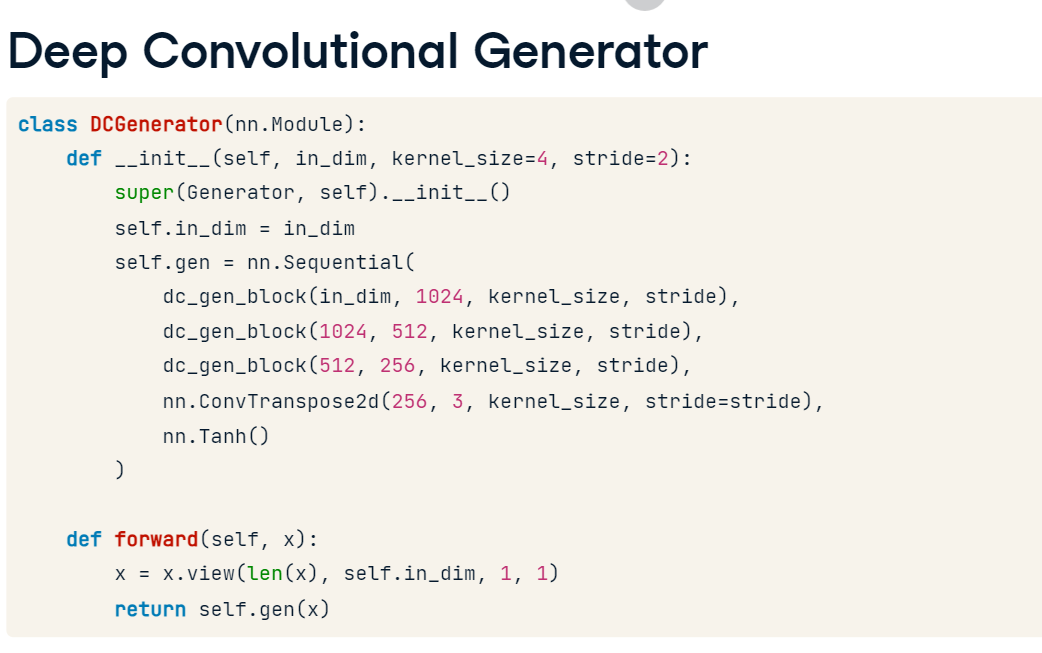
**Convolutional generator block**

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.



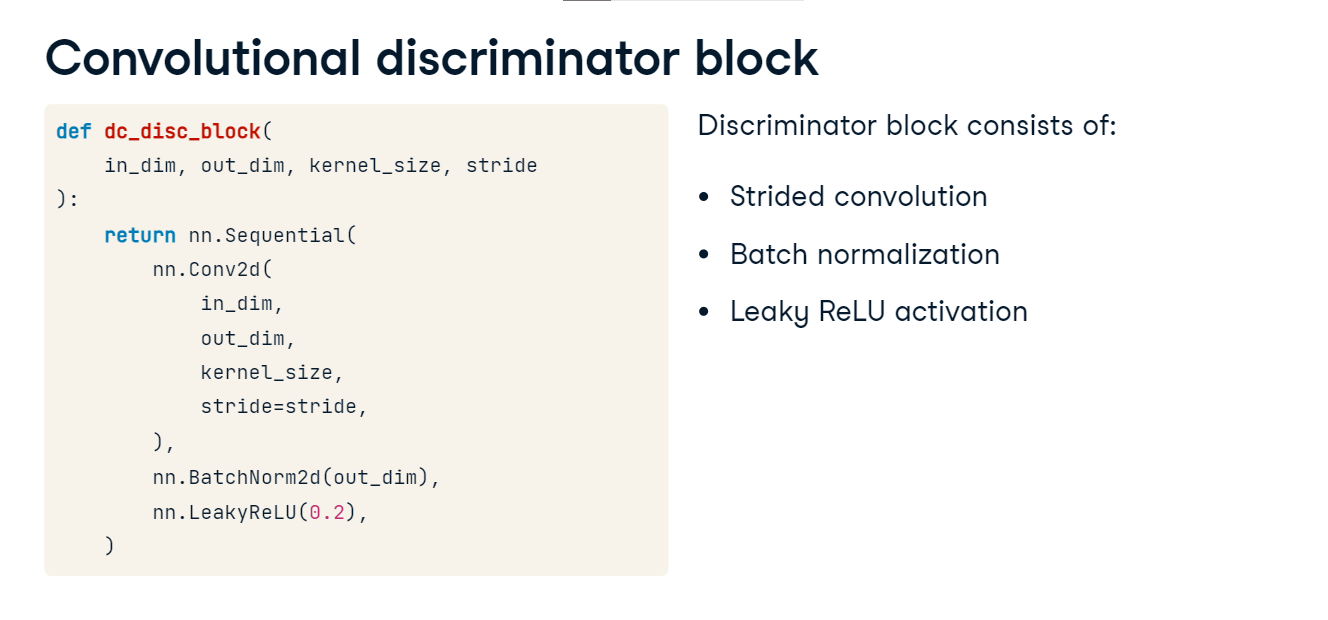
**Deep Convolutional Generator**

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.

****

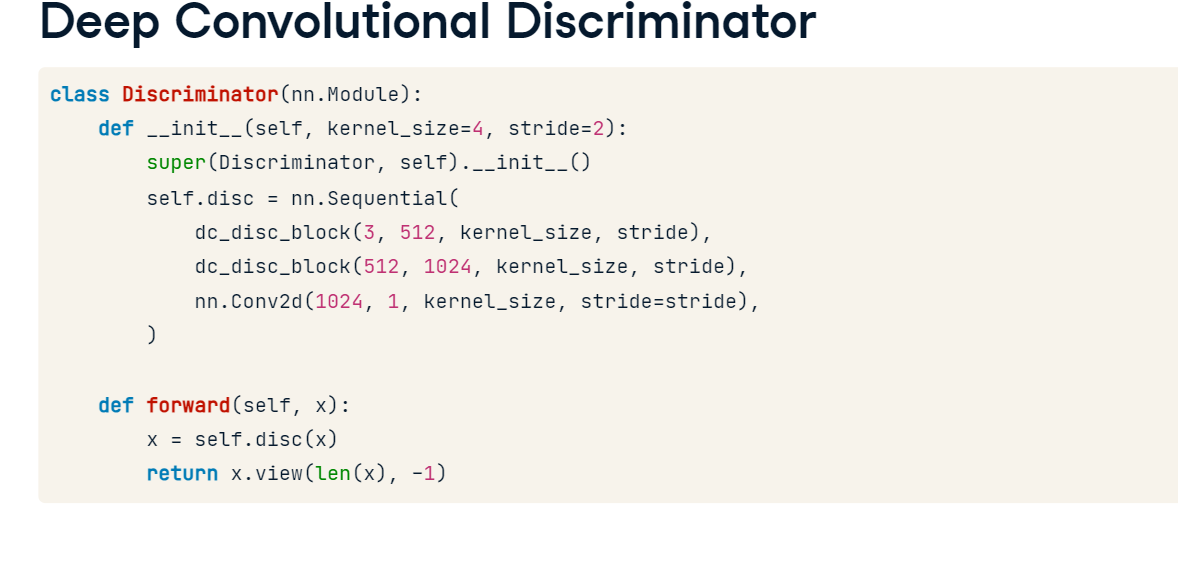
**Convolutional discriminator block**

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.



**Training GANs**

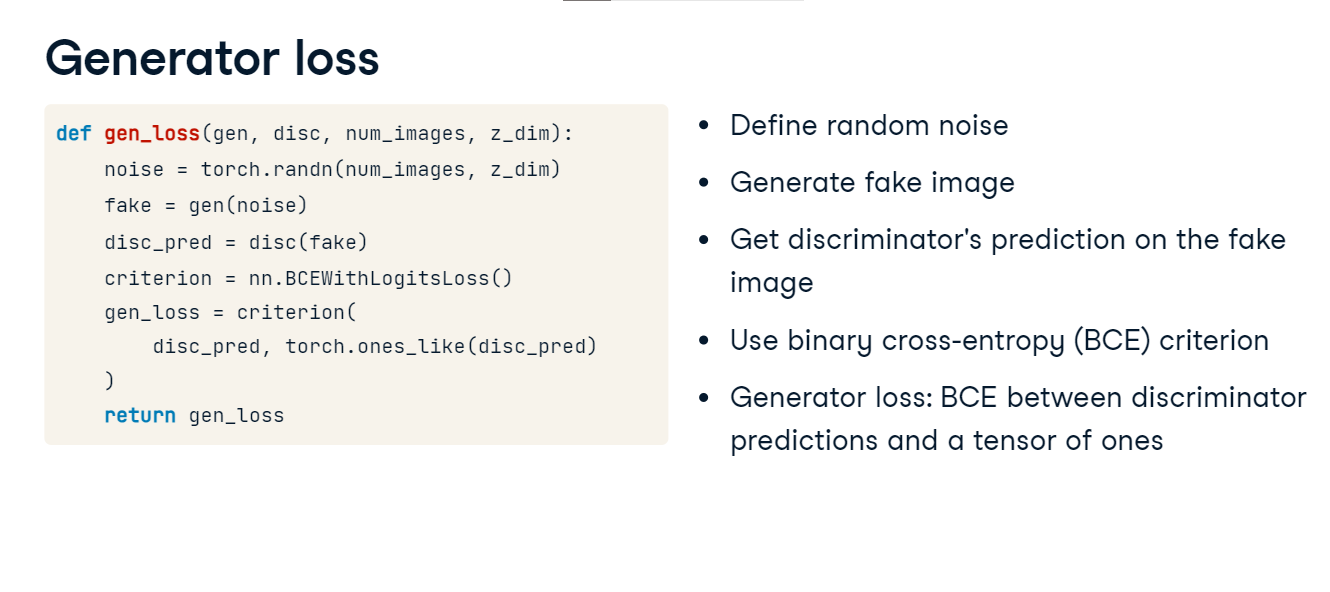
It's finally training time! But before we run the training loop, we need to define loss functions for the models.

**Generator objective**

Let's think about the generator's loss function first. Recall that generator's objective is to create such fake images that would fool the discriminator into classifying them as real. The key idea is to use the discriminator to inform us about the generator's quality. We will use the generator to produce some fake images and give them to the discriminator to classify. If it misclassifies them as real (label one), the generator is doing a good job and its loss will be small. If it correctly recognizes them as fake (label zero), generator loss will be large.

**Generator loss**

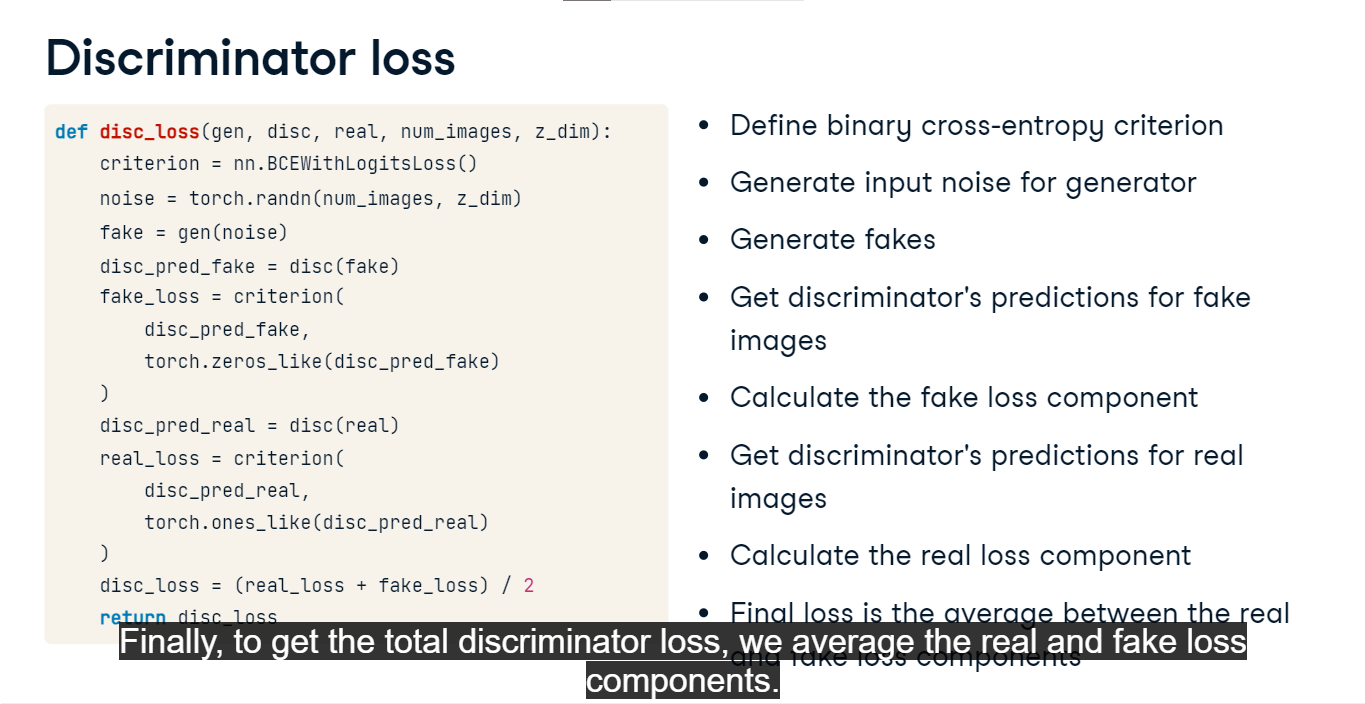
Let's see it in code. We define the function called gen-underscore-loss to compute generator loss. First, we define random noise as input for the generator. The noise tensor is of shape num-underscore-images, which corresponds to the batch size, by z-underscore-dim, the noise size. Then, we pass the noise to the generator to produce fake images which we then pass to the discriminator to classify. We define binary cross-entropy criterion to measure generator's performance. From the generator's perspective, it's desired that the discriminator classifies the fakes as real images which have the label one. Therefore, the generator loss is binary cross-entropy between the discriminator predictions for fakes, and the tensor of ones of the same shape, which we create with torch-dot-ones-underscore-like.



**Discriminator objective**

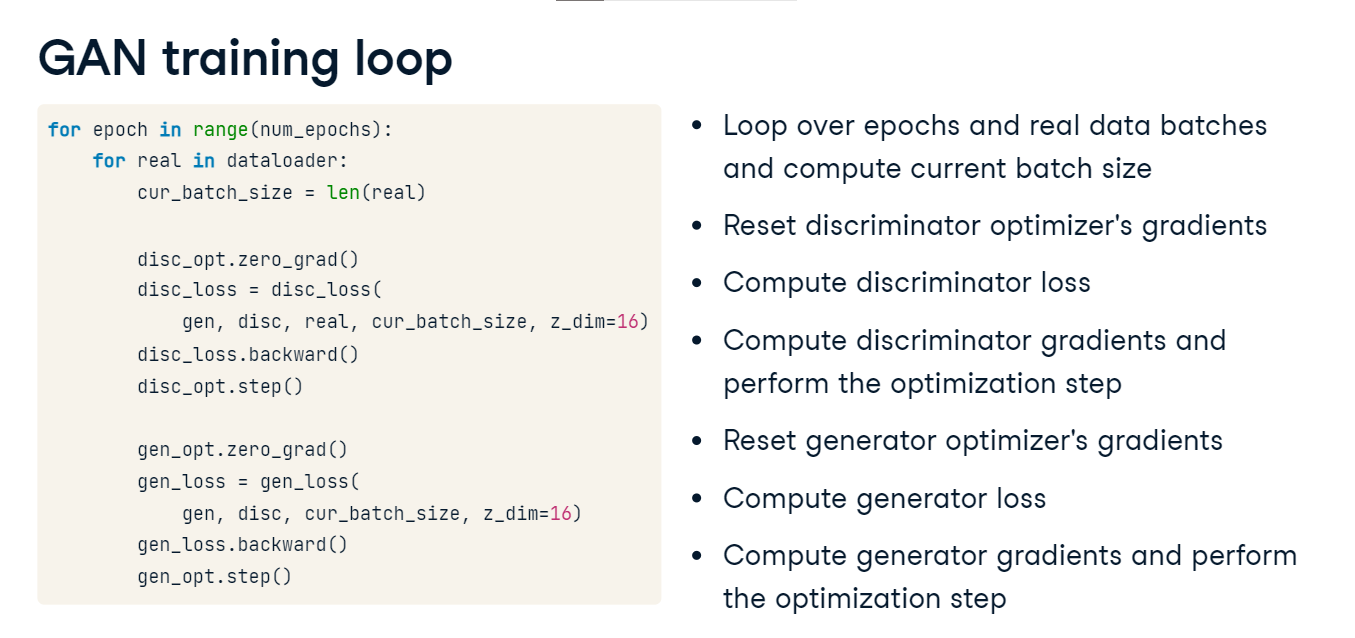
Let's turn to the discriminator now. Recall its objective is to correctly classify fakes and real images. To evaluate its loss, we will pass it some generator outputs to see if it correctly recognizes them as fake, or label zero. We will also pass it some real images from the training data expecting them to be classified as one, or real. Let's take a look at the code.

**Discriminator loss**

We define the disc-underscore-loss function to compute discriminator loss. We will use the binary cross-entropy criterion again. First, we produce random noise as generator input like before to feed it to the generator and obtain fake images which are then passed to the discriminator. This way we get disc-underscore-pred-underscore-fake, the discriminator's predictions for the fake images. Next, we pass those predictions alongside a tensor of zeros to the criterion. This fake loss component will be larger when discriminator's prediction for fakes are real. Then, we use the discriminator to classify a sample of real images from the training data. We then compute the real loss component by passing these predictions together with a tensor of ones to the criterion. This loss component is large when the discriminator misclassifies real images as fake. Finally, to get the total discriminator loss, we average the real and fake loss components.

**GAN training loop**

Let's define the GAN training loop! We iterate over epochs and real data batches from a pre-defined dataloader. For each batch, we compute the current batch size. We start with the discriminator. We reset its gradients by calling zero-dot-grad on the discriminator optimizer, which has been pre-defined, for example as an Adam optimizer. Next, we compute the discriminator loss using the function from before and call the backward method on the optimizer to compute the gradients. Then, we perform the optimization step by calling the step method of the optimizer. We then repeat the process for the generator: reset the gradients, compute the loss using our custom function, and compute gradients, and perform the optimization step.



**Evaluating GANs**

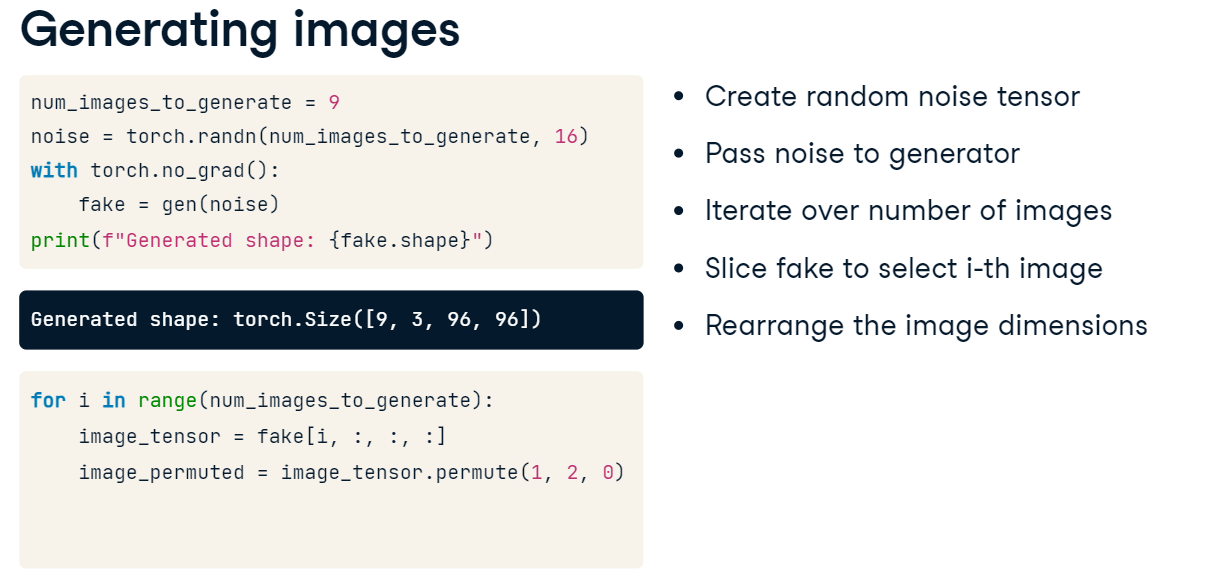
**Generating images**

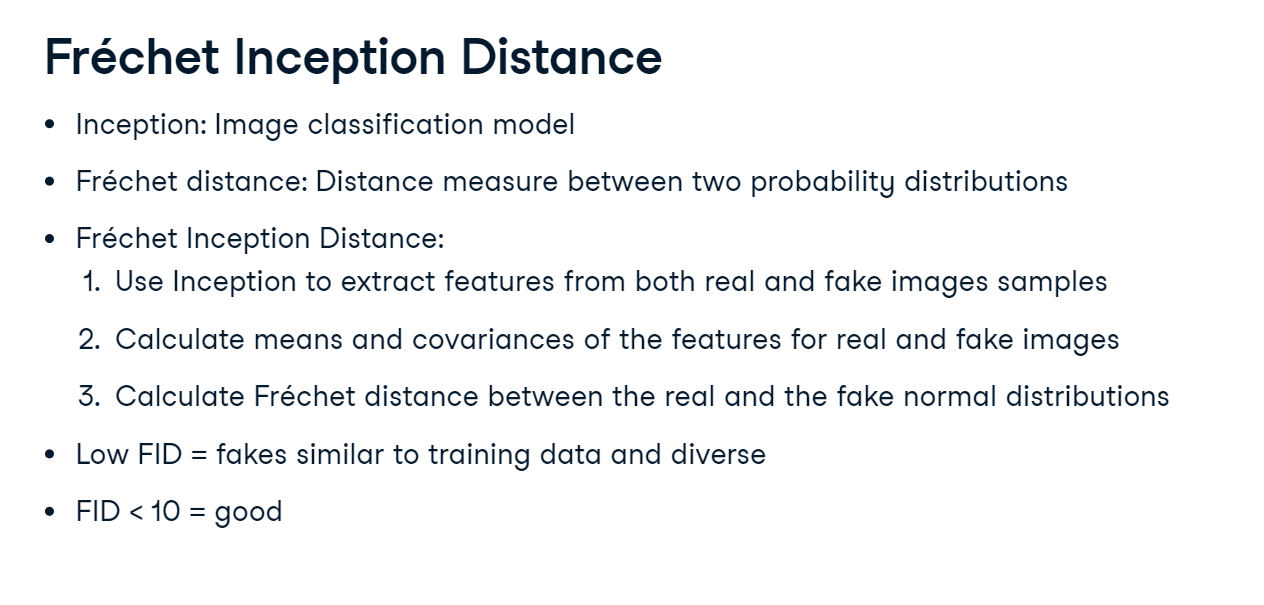
Since GANs produce visual outputs, the first step in evaluating a GAN is to inspect the images it generates. Let's do that! We will generate nine images. First, we create a random noise tensor of shape 9 by 16, where 16 is the noise input size we used during training. Then, with no gradients calculation, we pass the noise to gen, the generator, to obtain fake images. We can plot their shape. We have 9 images, each of 3 color channels and size 96 by 96 pixels. Let's visualize them. We iterate over the number of images to generate. For each, we extract the i-th image by slicing fake with square brackets and taking only the i-th element in the first dimension. Next, in order to visualize the image, we must rearrange its dimensions from color channel, height, width to height, width, color channel. We do that by calling the permute method on the image tensor and passing it the desired order of dimensions: 1, 2, and 0. Finally, we can plot the images.

**GAN generations**

Not bad! They do look a lot like pokemons from the training data. Some make the impression like they might be missing an eye or a leg, but in general, they are okay. But can we have a more precise evaluation method than this visual inspection?

**Fréchet Inception Distance**

A metric commonly used to evaluate GANs is the Fréchet Inception Distance, or FID for short. To understand how it works, we must first mention two related concepts: Inception and Fréchet Distance. Inception is a popular image classification model, while Fréchet distance is a distance measure between two probability distributions. Back to the Fréchet Inception Distance. FID uses a pre-trained Inception model to extract features from both the generated and real images. The extracted features are then used to calculate the mean and covariance for both sets of images (generated and real). These statistics encapsulate the distribution of features across the images. Finally, the FID is calculated using the Fréchet distance between the distributions of real and fake images, each parametrized with the mean and covariance calculated before. A lower FID score suggests that the distributions of generated and real i****mages are closer in the feature space, indicating that the generated images are more similar to the training data and more diverse. While there are no specific guidelines for interpreting the FID scores, typically values below 10 and considered good.



**FID in PyTorch**

Let's compute FID in PyTorch. We start by importing FrechetInceptionDistance from torchmetrics-dot-image-dot-fid. We instantiate the metric passing it feature equals 64 as argument. This means that we want to use the sixty-fourth layer of the inception model for feature extraction but a different one can be used, too. Next, we update the metrics with the sample of fake images. To do this, we call the update method of the metric we defined and pass it the fake images. However, we first need to convert the pixel values to integers between 0 and 255, and the GAN has given us floats between 0 and 1. To fix that, we multiply the image tensor by 255 and call dot-to-torch-uint8 on it as we pass it to the update method. We also set real equals false to indicate we are passing fake images. Then, we perform a similar update with real images, this time passing real equals true. Finally, we call the compute method on the metric to get its value. Seven-point-five is pretty low, indicating high-quality and diverse generations.

