Inside the Generative Adversarial Networks (GAN) architecture

Generative Adversarial Networks (**GANs**), represent a shift in architecture design for deep neural networks. There are several advantages to using this architecture: it generalizes with limited data, conceives new scenes from small datasets, and makes simulated data look more realistic. These are important topics in deep learning because many techniques today require large amounts of data. Using this new architecture, it's possible to drastically reduce the amount of data needed to complete these tasks. In extreme examples, these types of architectures can use 10% of the data needed for other types of deep learning problems. In this post we will understand what's inside a GAN architecture in detail.

This post is an excerpt taken from the book by Packt Publishing titled Generative Adversarial Networks Cookbook *written by Josh Kalin. In this book, you will learn different use cases involving DCGAN, Pix2Pix, and so on.*

Architecture structure basics

How do I build a GAN? There are a few principal components to the construction of this network architecture. First, we need to have a method to produce neural networks easily, such as Keras or PyTorch (using the TensorFlow backend).

How to build a GAN using an analogy

The classic analogy is the counterfeiter (generator) and FBI agent (discriminator). The counterfeiter is constantly looking for new ways to produce fake documents that can pass the FBI agent's tests. Let's break it down into a set of goals:

1. **Counterfeiter (generator) goal**: Produce products so that the cop cannot distinguish between the real and fake ones
2. **Cop (discriminator) goal**: Detect anomalous products by using prior experience to classify real and fake products

How GAN implementation works

Now, enough with the analogies, right? Let's restructure this into a game-theory-style problem-the minimax problem from the first GAN implementation. The following steps illustrate how we can create this type of problem:

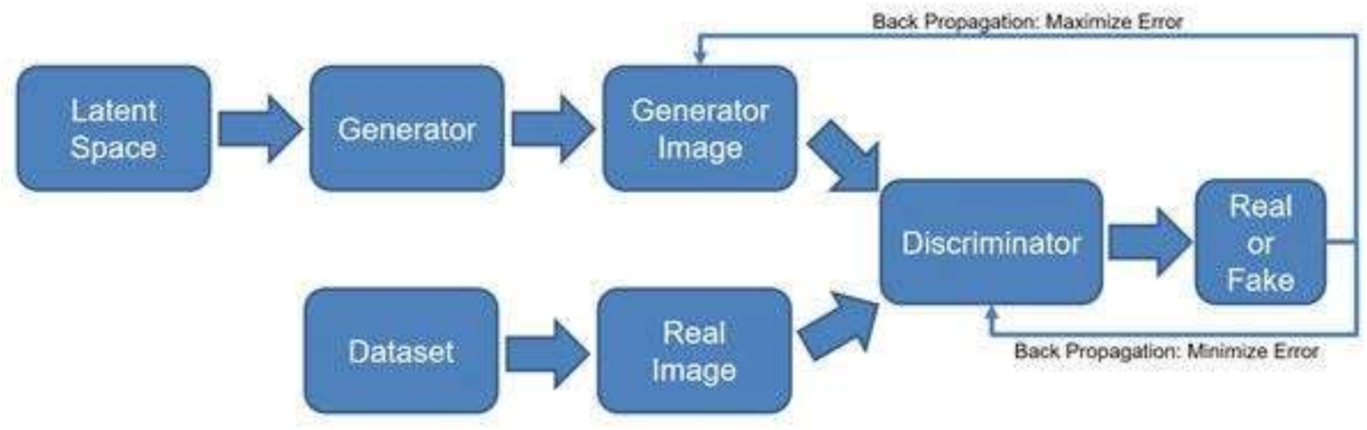
· **Generator goal**: Maximize the likelihood that the discriminator misclassifies its output as real

· **Discriminator goal**: Optimize toward a goal of 0.5, where the discriminator can't distinguish between real and generated images

Note

The Minimax Problem (sometimes called MinMax) is a theory that focuses on maximizing a function at the greatest loss (or vice versa). In the case of GANs, this is represented by the two models training in an adversarial way. The training step will focus on minimizing the error on the training loss for the generator while getting as close to 0.5 as possible on the discriminator (where the discriminator can’t tell the difference between real and fake).

In the GAN framework, the generator will start to train alongside the discriminator; the discriminator needs to train for a few epochs prior to starting the adversarial training as the discriminator will need to be able to actually classify images. There’s one final piece to this structure, called the loss function. The loss function provides the stopping criteria for the **Generator** and **Discriminator** training processes. Given all of these pieces, how do we structure these pieces into something we can train? Check out the following diagram:



A high-level description of the flow of the Generative Adversarial Network, showing the basic functions in block format

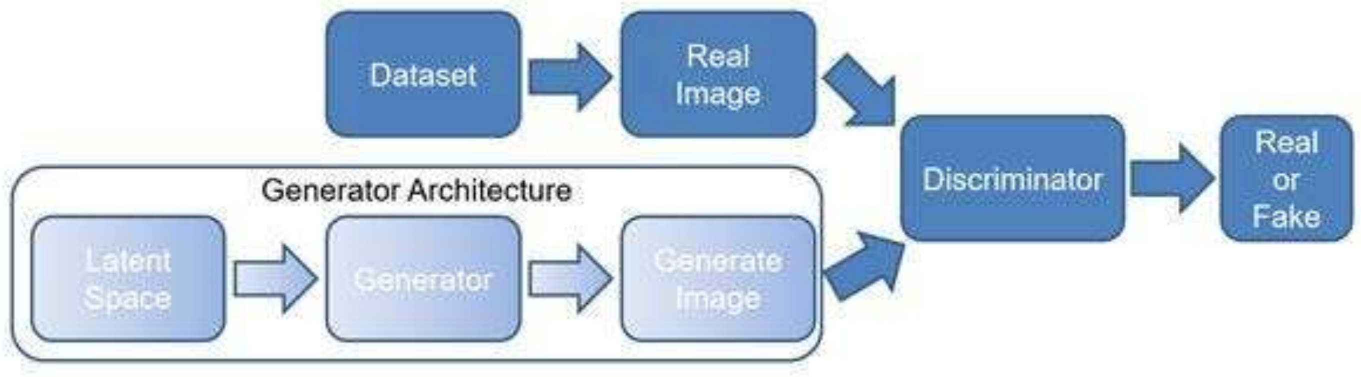
With this architecture, it’s time to break each piece into its component technology: generator, discriminator, and loss function. There will also be a section on training and inference to briefly cover how to train the model and get data out once it is trained.

Basic building block — generator

It’s important to focus on each of these components to understand how they come together. For each of these sections, I’ll be highlighting the architecture pieces to make it more apparent.

Generator architecture

The following diagram represents the important pieces of the generator:



The generator components in the architecture diagram: latent space, generator, and image generation by the generator

The focus in the diagram ensures that you see the core piece of code that you’ll be developing in the generator section.

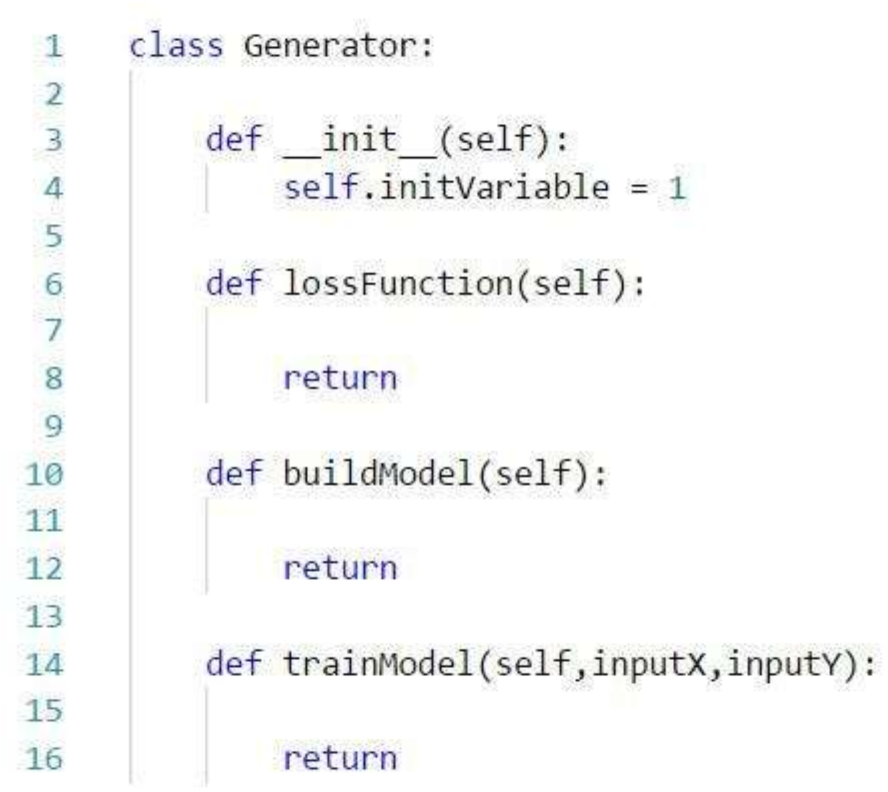
Here are a few steps to describe how we create a generator conceptually:

1. First, the generator samples from a latent space and creates a relationship between the latent space and the output
2. We then create a neural network that goes from an input (latent space) to output (image for most examples)
3. We’ll train the generator in an adversarial mode where we connect the generator and discriminator together in a model (every generator and GAN recipe in this book will show these steps)
4. The generator can then be used for inference after training

How a generator works

Each of these building blocks is fairly unique, but the generator is arguably the most important concept to understand. Ultimately, the generator will produce the images or output that we see after this entire training process is complete. When we talk about training GANs, it refers directly to training the generator. As we mentioned in a previous section, the discriminator will need to train for a few epochs prior to beginning the training process in most architectures or it would never complete training.

For each of these sections, it is important to understand the structure of the code we'll start building through the course of this book. In each chapter, we're going to define classes for each of the components. The generator will need to have three main functions within the class:



```
class Generator:

    def __init__(self):
        self.initVariable = 1

    def lossFunction(self):

        return

    def buildModel(self):

        return

    def trainModel(self,inputX,inputY):

        return
```

Class template for developing the generator — these represent the basic components we need to implement for each of our generator classes

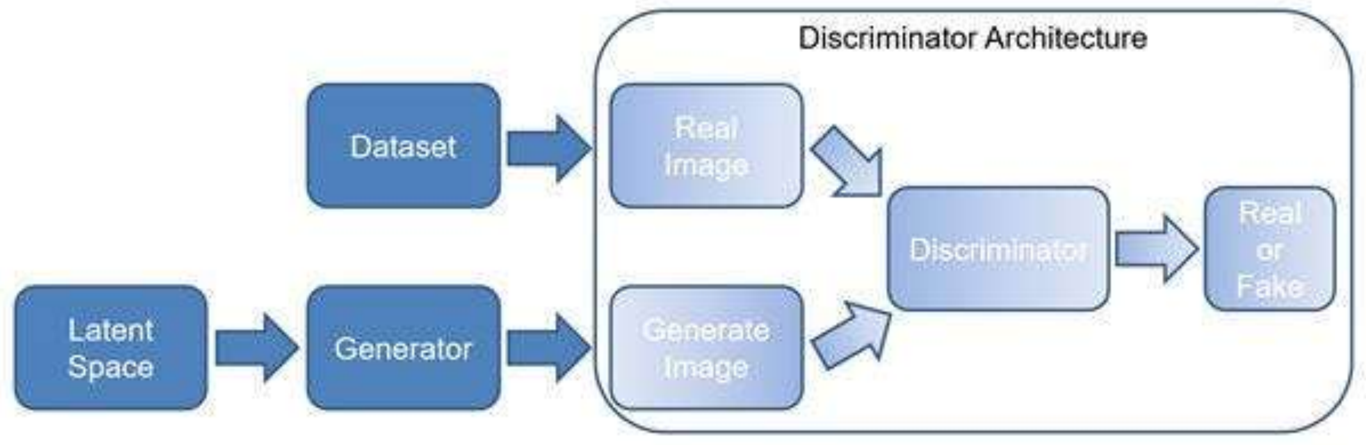
The loss function will define a custom loss function in training the model (if needed for that particular implementation). The *buildModel* function will construct the actual model of the given neural network. Specific training sequences for a model will go inside this class though we'll likely not use the internal training methods for anything but the discriminator.

Basic building block — discriminator

The generator generates the data in the GAN architecture, and now we are going to introduce the Discriminator architecture. The discriminator is used to determine whether the output of the generator and a real image are real or fake.

Discriminator architecture

The discriminator architecture determines whether the image is real or fake. In this case, we are focused solely on the neural network that we are going to create:



The basic components of the discriminator architecture

The discriminator is typically a simple **Convolution Neural Network** (**CNN**) in simple architectures. In our first few examples, this is the type of neural network we'll be using.

Here are a few steps to illustrate how we would build a discriminator:

1. First, we'll create a convolutional neural network to classify real or fake (binary classification)
2. We'll create a dataset of real data and we'll use our generator to create fake dataset
3. We train the discriminator model on the real and fake data
4. We'll learn to balance training of the discriminator with the generator training — if the discriminator is too good, the generator will diverge

How a discriminator works

So, why even use the discriminator in this case? The discriminator is able to take all of the good things we have with discriminative models and act as an adaptive loss function for the GAN as a whole. This means that the discriminator is able to adapt to the underlying distribution of data. This is one of the reasons that current deep learning discriminative models are so successful today — in the past, techniques relied too heavily on directly computing some heuristic on the underlying data distribution. Deep neural networks today are able to adapt and learn based on the distribution of the data, and the GAN technique takes advantage of that.

Ultimately, the discriminator is going to evaluate the output of the real image and the generated image for authenticity. The real images will score high on the scale initially, while the generated images will score lower. Eventually, the discriminator will have trouble distinguishing between the generated and real images. The discriminator will rely on building a model and potentially an initial loss function.