**What are GANs?**

Some time ago, I showed you how to create a simple **Convolutional Neural Network (ConvNet)** for satellite imagery classification using **Keras**. ConvNets are not the only cool thing you can do in Keras, they are actually just the tip of an iceberg. Now,I think it’s about time to show you something more!

Before we start, I will recommend that you review my two previous posts ([Ship recognition in satellite imagery part I](https://appsilon.com/ship-recognition-in-satellite-imagery-part-i/) and [part II](https://appsilon.com/ship-recognition-in-satellite-imagery-part-ii/)) if you haven’t already.

Okay, so what are GANs?

**Generative adversarial networks**, or GANs, were introduced in 2014 by [Ian Goodfellow](https://arxiv.org/abs/1406.2661). They are **generative algorithms** comprised of two deep neural networks “playing” against each other. To fully understand GANs, we have to first understand how the generative method works.

Let’s go back to our [ConvNet](https://appsilon.com/ship-recognition-in-satellite-imagery-part-ii/) for satellite imagery classification. As you remember, our task looked like this:

We wanted to predict class (ship or non-ship). To be more specific, we wanted to find the probability that the image belongs to the specific class, given the image. Each image was composed of a set of pixels that we were using as features/inputs. Mathematically, we were using a set of features, X (pixels), to get the conditional probability of Y (class) given X (pixels):

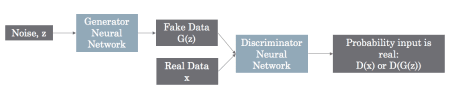
*p(y|x)*

This is an example of a **discriminative** algorithm. Generative algorithms, on the other hand, do the complete opposite. Using our example, assuming that the class of an image is “ship,” what should the image look like? More precisely, what value should each pixel have? This time, we’re generating the distribution of X (pixels) given Y (class):

*p(x|y)*

Now that we know how the generative algorithms work, we can dive deeper into GANs.

Like I said previously, GANs are composed of two deep neural networks. The first network is called the **generator**, and it’s basically responsible for creating new instances of data from random noise. The second network is called **discriminator**, and it “judges” if the data generated by the generator is real or fake by comparing it to real data.

  
*Note that I’m not saying that those are ConvNets or Recurrent Neural Networks. There are many different variations of GANs and depending on the task, we will use different networks to build our GAN. For example, later on, we will use* ***Deep*** ***Convolutional*** ***Generative Adversarial Networks*** ***(DCGAN)*** *to generate new satellite imagery.*

**DCGAN in R**

To build a GAN in R, we have to first build a generator and discriminator. Then, we will join them together. We want to create DCGAN for satellite imagery where the generator network will take random noise as input and will return the new image as an output.

image\_height <- 80 # Image height in pixels

image\_width <- 80 # Image width in pixels

image\_channels <- 3 # Number of color channels - here Red, Green and Blue

noise\_dim <- 80 # Length of gaussian noise vector for generator input

# Setting generator input as gaussian noise vector

generator\_input <- layer\_input(shape = c(noise\_shape))

# Setting generator output - 1d vector will be reshaped into an image array

generator\_output <- generator\_input %>%

layer\_dense(units = 64 \* image\_height / 4 \* image\_width / 4) %>%

layer\_activation\_leaky\_relu() %>%

layer\_reshape(target\_shape = c(image\_height / 4, image\_width / 4, 64)) %>%

layer\_conv\_2d(filters = 128, kernel\_size = 5, padding = "same") %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d\_transpose(filters = 128, kernel\_size = 4, strides = 2, padding = "same") %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d\_transpose(filters = 256, kernel\_size = 4, strides = 2, padding = "same") %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 256, kernel\_size = 5, padding = "same") %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 256, kernel\_size = 5, padding = "same") %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = image\_channels, kernel\_size = 7, activation = "tanh", padding = "same")

# Setting up the model

generator <- keras\_model(generator\_input, generator\_output)

The discriminator will take a real or generated image as input and return the probability of the image’s authenticity, indicating if the image was real or not.

# Setting discriminator input as an image array

discriminator\_input <- layer\_input(shape = c(image\_height, image\_width, image\_channels))

# Setting discriminator output - the probability that image is real or not

discriminator\_output <- discriminator\_input %>%

layer\_conv\_2d(filters = 256, kernel\_size = 4) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 256, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 128, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 128, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 128, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 128, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_conv\_2d(filters = 128, kernel\_size = 2, strides = 2) %>%

layer\_activation\_leaky\_relu() %>%

layer\_flatten() %>%

layer\_dropout(rate = 0.3) %>%

layer\_dense(units = 1, activation = "sigmoid")

# Setting up the model

discriminator <- keras\_model(discriminator\_input, discriminator\_output)

As previously stated, both networks are “playing” against each other. The discriminator’s task is to distinguish real and fake images, and the generator has to create new data (which is an image in this case) that will indistinguishable from real data. Because the discriminator is returning probabilities, we can use [binary cross-entropy](https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/) as the loss function.

discriminator %>% compile(

optimizer = optimizer\_rmsprop(

lr = 0.0006,

clipvalue = 1.0,

decay = 1e-7

),

loss = "binary\_crossentropy"

)

Before we merge our two networks into a GAN, we will freeze the discriminator weights so that they won’t be updated when the GAN is trained. Otherwise, this would cause the discriminator to return “true” value for each image we pass into it. Instead, we will train networks separately.

freeze\_weights(discriminator)

gan\_input <- layer\_input(shape = c(noise\_shape))

gan\_output <- discriminator(generator(gan\_input))

gan <- keras\_model(gan\_input, gan\_output) gan %>% compile(

optimizer = optimizer\_rmsprop(

lr = 0.0003,

clipvalue = 1.0,

decay = 1e-7

),

loss = "binary\_crossentropy"

)

# Training the GAN doesn't follow the simplicity as we could experience while working with Convolutional Networks. In simplification, we have to train both networks separately in a loop.

for(i in 1:1000) {

# TRAIN THE DISCRIMINATOR

# TRAIN THE GAN

# You can find full code of the training process for similar example in https://www.manning.com/books/deep-learning-with-r

}

If you want to learn more about GANs and Keras, I would encourage that you [read Deep Learning with R](https://www.manning.com/books/deep-learning-with-r). It’s a great place to start your adventure with Keras and deep learning.

**Results**

I’ve checked a few architectures of my GAN, and below, you will find some of the results.

We can see that the generator is learning how to create some simple “ship-like” shapes. All of them share the same orientation as the ship, water hue, and so on.  We can also see what happens when a GAN is over-trained because we’re getting some really abstract pictures.

The results are limited for two reasons. First of all, we worked on a really small sample size. Secondly, we should try out many different architectures of neural networks. In this example, I was working on my local machine, but using a cluster of machines over a longer period of time would likely give us much better results.

