Generative Adversarial Networks (GANs)

Generative Adversarial Networks, known as GANs, are in their most basic form, two neural networks that teach each other how to solve a specific task. It was invented by Goodfellow and colleagues in 2014[[1]](#footnote-1). Intuitively, the two networks help each other with the final goal of being able to generate new data that look like as the data used for training. For example, you may want to train a network to generate human faces that are as realistic as possible. In this case one network will generate human faces as good as it can, and the second network will criticize the results and tell the first network how to correct the faces. The two networks will learn from each other, so to speak. In this chapter we will look in detail how this works, and how to implement an easy example in Keras.

The goal of this chapter is to give you a basic understanding on how GANs work. Adversarial learning (of which GANs are a specific case) is a vast area of research and starts to be an advanced topic in deep learning. In this chapter we will investigate in detail how a basic GAN system work, and we will discuss, albeit in a shorter way, how conditional GANs function. Complete examples can be found, as usual, at <https://adl.toelt.ai>.

# Introduction to GANs

The best way for us to understand how GANs work, is to base our discussion on the diagram in Figure 11.1. After having understood what is going on under the hood, we will look at how to implement GANs in Keras.

## Training Algorithm for GANs

To build a GANs system, we need two neural networks: a *Generator* and a *Discriminator.* The Generator has the goal of producing a fake observation[[2]](#footnote-2) , while the Discriminator has the goal of classifying an input as fake or real. Imagine the following classical example: the Generator (let’s call him George) can be an art forger that is trying to produce paintings of some known painter, let’s say Van Gogh. And the Discriminator (let’s call her Anna) is an art critic that gives a judgement if the painting that George has produced looks real or not. They are new to this, so they decide to learn this together[[3]](#footnote-3). George produces a painting. Anna examines it and gives some suggestions to George. Every now and then, Anna also trains with some real Van Gogh paintings to get better in spotting errors from George. This process is repeated many times, until George is so good to fool Anna. At this point George can paint like Van Gogh, can produce many fakes, and get rich by selling his fake paintings[[4]](#footnote-4). This process that we just described, is depicted in Figure 1.11. Let’s see how our story translates in the language of neural networks.

Diagram

Description automatically generated

Figure 11-1: A diagram depicting all components and steps of a GAN setup.

The generator gets as input a noise vector taken from a normal distribution. The size of this vector is not fixed and can be chosen depending on the problem at hand. In the example we will discuss in this chapter we have chosen . The Generator (George) take the random vector and generate a fake observation (as you can see in Figure 1-11). The output will have the same dimension of the observations contained in the training dataset (in this example Van Gogh paintings). If for example if are 1000x1000 pixels images in color, then will also be a 1000x1000 color image. Now is the Discriminator’s (Anna’s) turn. It gets as input a (or ) and produces a one-dimensional output (the probability of the input of being real or fake). Basically, the discriminator is performing binary classification.

The steps of the training loop are described below.

1. A vector of numbers is generated from a normal distribution.
2. Using this the Generator gives as output an .
3. The discriminator is used two times: one with a real input () and one with the generated in the previous step.
4. Two loss functions are calculated: and
5. Via an optimizer (Adam, Momentum, etc.), the two loss functions are minimized sequentially (sometime for one step for the Generator, multiple steps in updating the weights for the Discriminator are run). Note that minimizing will be done only with respect to the trainable parameters of the Generator, while minimizing will be done only with respect to the trainable parameters of the Discriminator. Sometime for one step for the Generator, multiple steps for the Discriminator are carried out.

## A practical example with Keras and MNIST

Let’s see now a practical example of what we discussed in the previous section implemented with Keras and applied to the MNIST dataset[[5]](#footnote-5). As usual you can find the complete code on <https://adl.toelt.ai> so we will concentrate here only on the relevant parts of the code. In particular we will look at the 5 steps described at the end of the previous section and how to implement them. To start we will first need to create two neural networks: the Generator and the Discriminator. This can be done in the usual way. Nothing new here. For example

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(7\*7\*256, use\_bias=False, input\_shape=(100,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Reshape((7, 7, 256)))

assert model.output\_shape == (None, 7, 7, 256) # Note: None is the batch size

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

assert model.output\_shape == (None, 7, 7, 128)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

assert model.output\_shape == (None, 14, 14, 64)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

return model

The really important part of this network is the input shape: input\_shape=(100,). Remember that the Generator gets as input the random vector that is, in our example, a 100-dimensional vector of random numbers generated from a normal distribution. In Figure 11-2 you can see a better visualization of the network.

Table

Description automatically generated with medium confidence

Figure 11-2: the Generator neural network architecture.

In Figure 11-2 you can see how the random vector is transformed in increasingly larger images, until at the end, the expected 28x28 pixels image with one channel is obtained (this will be the we discussed in the previous section). The Discriminator can be created analogously, with standard Keras:

def make\_discriminator\_model():

model = tf.keras.Sequential()

model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',

input\_shape=[28, 28, 1]))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Flatten())

model.add(layers.Dense(1))

return model

that is a rather small network. The input will be an image 28x28 pixels in resolution and with just one channel (gray levels). The output is just the probability that the image is real and is achieved with one neuron layers.Dense(1).

In Figure 11-3 you can see the network architecture.

Diagram

Description automatically generated

Figure 11-3: the Discrimnator neural network architecture.

As discussed, we need to train the two networks in alternate fashion, so you will realize that the standard compile()/fit() approach will not be enough and we will need to develop our own custom training loop[[6]](#footnote-6). Before doing that, we need to define the loss functions that we need. This is not difficult, and we can start with the discriminator function :

def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

After having defined

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

You will remember that we will need the (this will be the variable fake\_output) and the (the variable real\_output) to train the discriminator. The Generator loss function is defined analogously

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

For , as you will remember from the previous section, we only need . At this point we are almost done. We need to define the optimizers (always using standard Keras functions)

generator\_optimizer = tf.keras.optimizers.Adam(1e-4)

discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

And now here is the custom training loop

def train\_step(images):

# Generation of the xi vector (random noise)

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:

# Calculation of X\_{fake}

generated\_images = generator(noise, training=True)

# Calculation of \hat Y\_{real}

real\_output = discriminator(images, training=True)

# Calculation of \hat Y\_{fake}

fake\_output = discriminator(generated\_images, training=True)

# Calculation of L\_G

gen\_loss = generator\_loss(fake\_output)

# Calculation of L\_D

disc\_loss = discriminator\_loss(real\_output, fake\_output)

# Calculation of the gradients of L\_G for backpropagation

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

# Calculation of the gradients of L\_D for backpropagation

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

# Applications of the gradients to update the weights generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

Let us summarize the steps:

* We first generate : generated\_images = generator(noise, training=True)
* Then : real\_output = discriminator(images, training=True)
* Then : fake\_output = discriminator(generated\_images, training=True)
* Then we define : gen\_loss = generator\_loss(fake\_output)
* Then : disc\_loss = discriminator\_loss(real\_output, fake\_output)

At this point we can evaluate the gradients:

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

and then apply them to update the trainable parameters of the two networks:

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

At this point the only thing left, is to perform those steps enough times to get the network to learn. By comparing Figure 11-1 and this code, you should be able to immediately see how this GAN is implemented. In Figure 11-4 you can see

Qr code

Description automatically generated with low confidence

Figure 11-4: 4 examples of digits generated by the Generator network that is described in the text. The digits do not exist in the dataset, and have been “created” by the neural network.

To generate images the only thing you need to do, is to feed the Generator with 100 random numbers. For example, with

noise = tf.random.normal([1, 100])

generated\_image = generator(noise, training=False)

Now be aware that due to the dimensions used in the code, if you want to extract the 28x28 image you need to use the code generated\_image[0, :, :, 0]. You can find the entire code at <https://adl.toelt.ai>. Try different networks, different number of epochs and so on to get a feeling on how such an approach can generate realistic images from a training dataset. Note that the approach we described learn from all the classes at the same time. For example, is not possible to ask the network to generate a specific digit. The Generator will simply randomly generate one digit. To be able to do this, we need to implement what is called “conditional” GANs. Those gets as input also the class labels and are able to generate examples from specific classes. If you want to try the code on your laptop keep in mind that training GANs is rather slow. If you do it on Google Colab and you use a GPU, one epoch may take up to 30 seconds or more. Keep that in mind. On a modern laptop without GPUs, one epoch may take up to 1.5-2 minutes.

### A note on Training

There is an important aspect on why the training is implemented in sequential fashion that we need to discuss. One could ask why we need to train the two networks in alternate fashion. Why cannot we train the Discriminator alone for example until it gets really good at distinguishing fakes and real images? The reason is very simple. Imagine that the Discriminator is really good. It will always spot the as fakes, and therefore the Generator will never be able to get better, as the Discriminator will never make any mistake. Therefore, the training, in such a situation would never be successful. In practice one of the biggest challenge when training GANs, is to make sure that the Generator and the Discriminator networks remains during the training at approximately the same skill level.

# Conditional GANs

Now let´s turn our attention to conditional GANs (CGANs). CGANs work the same as we have described in this chapter. The working idea is the same, with the difference that we will be able to specify from which class we want the Generator to create an image. In the MNIST example, we could tell the Generator that we want one fake digit one, for example. In Figure 11-5 you can see an updated diagram explaining the training (Figure 11-1 updated).

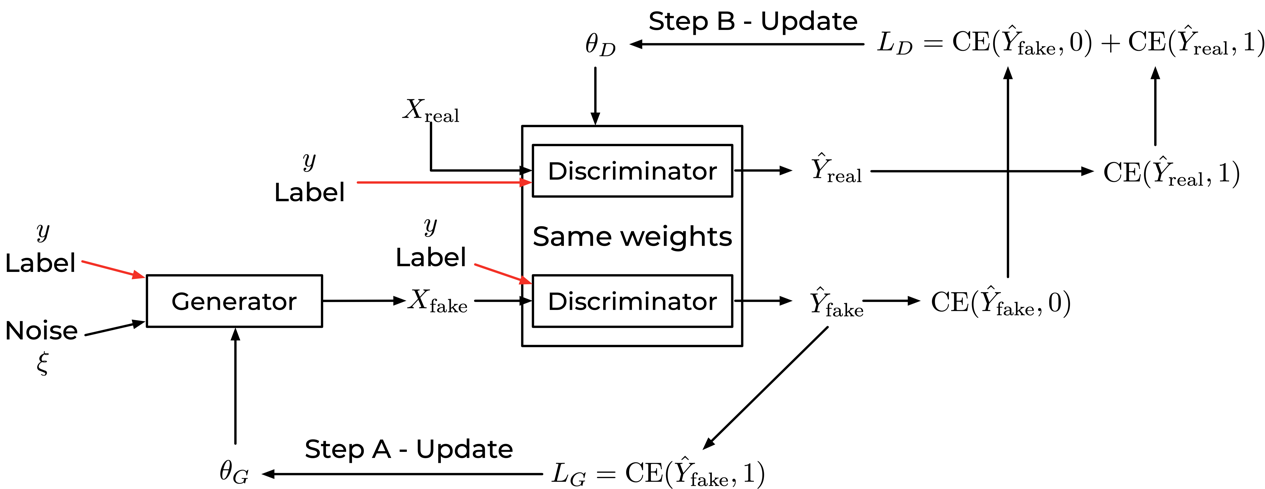


Figure 11-5: the training of CGAN system. In red is highlighted the role of the label that make possible for the Generator to create fake examples of specific classes.

The main thing that we need to change to achieve this, are the architectures of the two networks. In Figure 11-6 and 11-6 you can see example architectures of two networks: a Generator and a Discriminator respectively.

A picture containing text, receipt, screenshot

Description automatically generated

Figure 11-6: The Generator network architecture for CGAN.

Diagram, table

Description automatically generated

Figure 11-7: the Discriminator network architecture for a CGAN.

From Figures 11-6 and 11-7 you can immediately see that what has changed is that they now have an additional input: a one-dimensional tensor, that will be the class. The kind of networks that you see here can be easily implemented by using the functional Keras API. Just to give you an idea about how to build such networks, here are the first layers of the Generator network up until the merging of the two branches

input\_label = Input(shape=(1,))

emb = Embedding(n\_classes, 50)(input\_label)

n\_nodes = 7 \* 7

emb = Dense(n\_nodes)( emb)

emb = Reshape((7, 7, 1))(emb)

in\_lat = Input(shape=(latent\_dim,))

n\_nodes = 128 \* 7 \* 7

gen = Dense(n\_nodes)(in\_lat)

gen = LeakyReLU(alpha=0.2)(gen)

gen = Reshape((7, 7, 128))(gen)

merge = Concatenate()([gen, emb])

where you can see how flexible Keras Functional APIs are. Now when training the Generator network for example, you need to give as input not only a random vector as before but a random vector **and** a class label, that you will use to choose the to train the Discriminator. To generate the random noise and the labels you will use a code that will look like this

latend\_dim = 100

x\_input = randn(latent\_dim \* n\_samples)

z\_input = x\_input.reshape(n\_samples, latent\_dim)

labels = randint(0, n\_classes, n\_samples)

And you will give the Generator network the input as [z\_input, labels]. The variable n\_samples is simply the batch size you want to use. Now to analyze the complete code would make this chapter really long, really difficult to follow and really boring. Implementing a CGAN starts to be really advanced, and the best way of understanding is to go through a complete example. As usual, you will find one at <https://adl.toelt.ai> where you can check all the code. In the meanwhile, your best source of knowledge about CGANs are the two papers

Radford, Alec, Luke Metz, and Soumith Chintala.

"*Unsupervised representation learning with deep convolutional*

*generative adversarial networks*." arXiv preprint arXiv:1511.06434 (2015).

Mirza, Mehdi, and Simon Osindero. "*Conditional generative*

*adversarial nets*." arXiv preprint arXiv:1411.1784 (2014).

That you should study to really understand what is going on with CGANs. Remember the goal of this chapter is not to go into details about advanced GAN architectures, but to give you an initial understanding on how adversarial learning works. There are many advanced architectures and topics about GANs that we cannot cover in this book, since they would go well beyond the skill level of the average reader of this book. But with this chapter I hope I could give you an initial understanding on how GANs work and how easy is to implement them in Keras.

1. Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). [Generative Adversarial Nets](https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf) (PDF). Proceedings of the International Conference on Neural Information Processing Systems (NIPS 2014). pp. 2672–2680. [↑](#footnote-ref-1)
2. We use here the generic term observation. They could be fake faces, in case you are trying to build a system that generates realistic faces, or an aged version of a face for example. We call an observation one of the inputs in the training dataset. [↑](#footnote-ref-2)
3. How realistic would be that a forger and a critic work together is not the main point of the story. [↑](#footnote-ref-3)
4. Of course, we are not encouraging anyone to become dishonest. Is just a story to let you understand GANs… [↑](#footnote-ref-4)
5. At this point in the book, you should know the MNIST dataset very well. In case you don’t remember, it is a dataset with 70000 handwritten digits, save das gray-level images 28x28 pixel in resolution. [↑](#footnote-ref-5)
6. If you have never seen a custom training loop, you can check Appendix 2 where the basics of customizing Keras are explained. [↑](#footnote-ref-6)