Generative Adversarial Networks (GANs) - A Gentle Introduction

Umberto Michelucci (mailto:umberto.michelucci@toelt.ai)

January 25, 2022

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]. 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 paper 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 paper I will describe 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 at `https://adl.toelt.ai`.

1 A Story about Fake Paintings

Imagine that there a painter (let's call her Susie) that wants to learn how to paint like Van Gogh. She does not now how but she is willing to learn. She finds a friend (let's call her Marie), that is willing to help her and they decide to proceed in this way. Susie will paint fake Van Gogh painting as good as she can, and Marie will try to criticize and tell Susie what she did wrong as best as she can. To improve, Marie will try to learn to recognize real Van Gogh painting as best as she can so that she can give better feedback to Susie. They go on in this fashion as long as possible. Susie will get better and her painting will be more and more difficult to distinguish from a real Van Gogh painting for Susie. But she will also learn and therefore becomes better at spotting small details that will give the fakes away. If they continue to do this long enough,

[1] Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). Generative Adversarial Nets (PDF). Proceedings of the International Conference on Neural Information Processing Systems (NIPS 2014). pp. 2672–2680.

Susie will become very good at painting like Van Gogh, and Marie will become very good at spotting fakes.
GANs work exactly on this principle. Susie and Marie will be replaced by neural networks: called a generator and a discriminator respectively, that will be trained with backpropagation. One network (the generator) will try to produce *fakes* as best as possible (fakes in this context means realistic examples of some kind, for example realistic faces, realistic images, etc.), and the second (the discriminator) will try to determine if the output of the generator is fake or not and give feedback.

2 Introduction to GANs

The best way for us to understand how GANs work in detail, is to base our discussion on the diagram in Figure 1. After having understood what is going on under the hood, we will look at how to implement GANs in Keras.

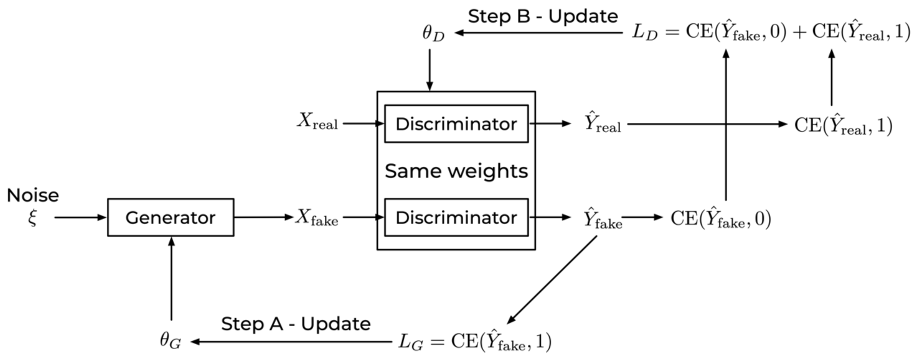
Step B - Update
θ_D
$L_D = \mathrm{CE}(\hat{Y}_{\text{fake}}, 0) + \mathrm{CE}(\hat{Y}_{\text{real}}, 1)$
X_{real}
Discriminator
$\hat{Y}_{\text{real}}$
$\mathrm{CE}(\hat{Y}_{\text{real}}, 1)$
Same weights
Noise
ξ
Generator
X_{fake}
Discriminator
$\hat{Y}_{\text{fake}}$
$\mathrm{CE}(\hat{Y}_{\text{fake}}, 0)$
Step A - Update
θ_G
$L_G = \mathrm{CE}(\hat{Y}_{\text{fake}}, 1)$

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

2.1 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] X_{fake}, while the Discriminator has the goal of classifying an input *X* as fake or real. Imagine again the example we discussed at the beginning: the Generator (Susie) is trying to produce paintings of Van Gogh. And the Discriminator (Marie) gives a judgement if the painting that Susie has produced looks real or not. They are new to this, so they decide to learn this together. Susie produces a painting. Marie examines it and gives some suggestions to Susie. Every now and then, Marie also trains with some real Van Gogh paintings to get better in spotting errors from Susie. This process is repeated many times, until Susie is so good to fool Marie. At this point Susie can paint like Van Gogh,

[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.

can produce many fakes, and get rich by selling her fake paintings[3]. This process that we just described, is depicted in Figure 1. Let's see how our story translates in the language of neural networks.
The generator gets as input a noise vector $\xi \in \mathbb{R}^k$ 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 $k = 100$. The Generator (George) take the random vector and generate a fake observation X_{fake} (as you can see in Figure 1). The output X_{fake} will have the same dimension of the observations contained in the training dataset X_{real} (in this example Van Gogh paintings). If for example if X_{real} are 1000x1000 pixels images in color, then X_{fake} will also be a 1000x1000 color image. Now is the Discriminator's (Anna's) turn. It gets as input a X_{real} (or X_{fake}) and produces a one-dimensional output $\hat{Y}$ (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 $\xi \in \mathbb{R}^k$ of k numbers is generated from a normal distribution.
2. Using this ξ the Generator gives as output an X_{fake}. The discriminator is used two times: one with a real input (X_{real}) and one with the X_{fake} generated in the previous step.
3. Two loss functions are calculated: $L_G = CE(Y_{fake}, 1)$ and $L_D = CE\,(Y_{real}, 1) + CE(X_{fake}, 0)$
4. 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 L_G will be done only with respect to the trainable parameters of the Generator, while minimizing L_D 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.

2.2 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[4]. 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()
```

[3]Of course, we are not encouraging anyone to become dishonest. Is just a story to let you understand GANs...

[4] 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.

```
  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
```

Listing 1: A function to create the generator 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 ξ that is, in our example, a 100-dimensional vector of random numbers generated from a normal distribution. In Figure 2 you can see a better visualization of the network. There 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 X_{fake} 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
```

Listing 2: A function to create the discriminator 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 3 you can see the 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. 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 L_D:

```
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
```

Listing 3: A function to calculate the discriminator loss.

After having defined

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Listing 4: The Keras cross-entropy loss function.

You will remember that we will need the X_{fake} (this will be the variable `fake_output`) and the X_{real} (the variable `real_output`) to train the discriminator. The Generator loss function L_G is defined analogously

```
def generator_loss(fake_output):
  return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Listing 5: A function to calculate the generator loss.

For L_G, as you will remember from the previous section, we only need X_{fake}. 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)
```

Listing 6: The optimizers that will be used in the custom training loop.

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))
```

Listing 7: A function to calculate the generator loss.

Let us summarize the steps:

1. We first calculate X_{fake}:
   `generated_images = generator(noise, training=True)`
2. We calculate $\hat{Y}_{real}$:
   `real_output = discriminator(images, training=True)`
3. We calculate $\hat{Y}_{fake}$:
   `fake_output = discriminator(generated_images, training=True)`
4. We calculate L_G:
   `gen_loss = generator_loss(fake_output)`
5. We calculate L_D:
   `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)
```

Listing 8: A function to calculate the generator loss.

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))
```

Listing 9: A function to calculate the generator loss.

At this point the only thing left, is to perform those steps enough times to get the network to learn. By comparing Figure 1 and this code, you should be able to immediately see how this GAN is implemented. In Figure 14 you can see some digits generated by a generator when trained on the MNIST dataset. 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)
```

Listing 10: A function to calculate the generator loss.

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.

2.2.1 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 X_{fake} 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.

3 Conditional GANs

Now let's turn our attention to conditional GANs (CGANs). CGANs work almost the same as we have described in this paper. 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 5 you can see an updated diagram explaining the training (Figure 5 updated).
The main thing that we need to change to achieve this, are the architectures of the two networks. In Figure 6 and 7 you can see example architectures of two networks: a Generator and a Discriminator respectively.
From Figures 6 and 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])
```

Listing 11: First layers of the generator network.

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 ξ as before but a random vector **and** a class label, that you will use to choose the X_{real} 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)
```

Listing 12: A function to calculate the generator loss.

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 paper 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).

You should study them to really understand what is going on with CGANs. Remember the goal of this paper 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.

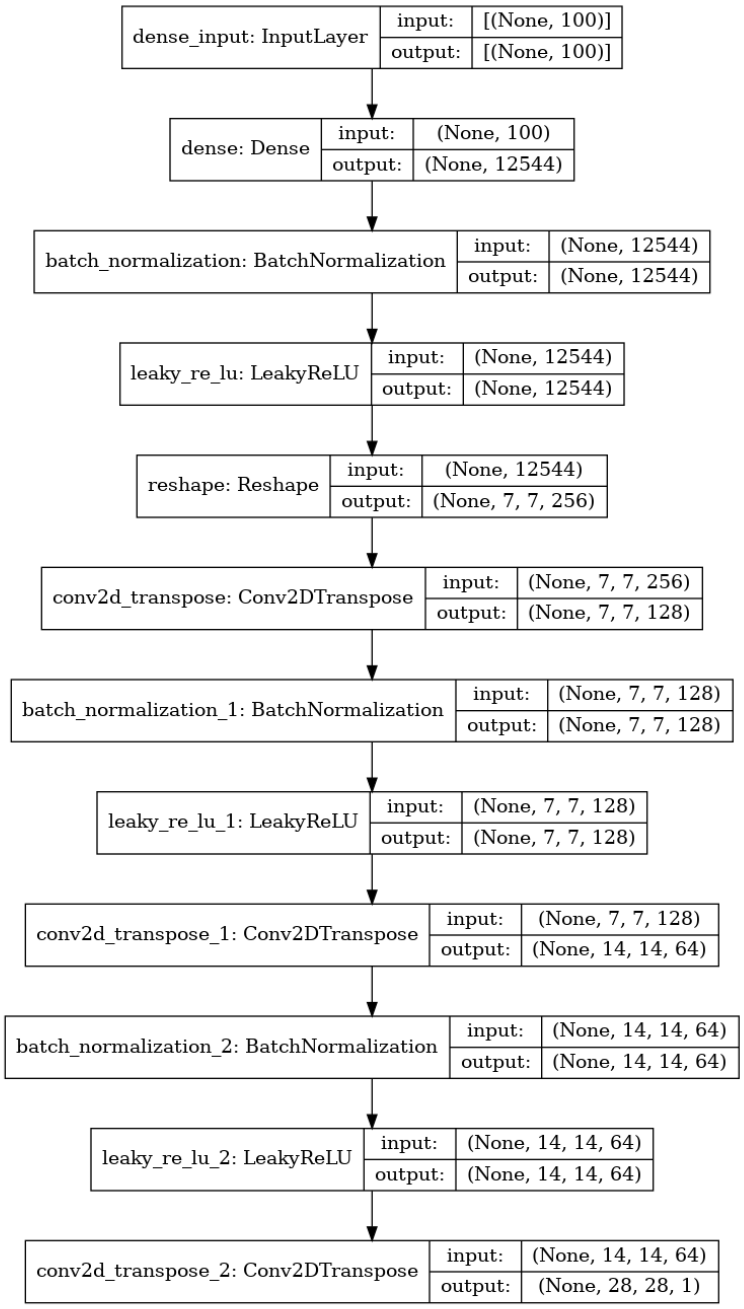
dense_input: InputLayer | input: [(None, 100)] | output: [(None, 100)]
dense: Dense | input: (None, 100) | output: (None, 12544)
batch_normalization: BatchNormalization | input: (None, 12544) | output: (None, 12544)
leaky_re_lu: LeakyReLU | input: (None, 12544) | output: (None, 12544)
reshape: Reshape | input: (None, 12544) | output: (None, 7, 7, 256)
conv2d_transpose: Conv2DTranspose | input: (None, 7, 7, 256) | output: (None, 7, 7, 128)
batch_normalization_1: BatchNormalization | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
leaky_re_lu_1: LeakyReLU | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
conv2d_transpose_1: Conv2DTranspose | input: (None, 7, 7, 128) | output: (None, 14, 14, 64)
batch_normalization_2: BatchNormalization | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
leaky_re_lu_2: LeakyReLU | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
conv2d_transpose_2: Conv2DTranspose | input: (None, 14, 14, 64) | output: (None, 28, 28, 1)

Figure 2: the Generator neural network architecture.

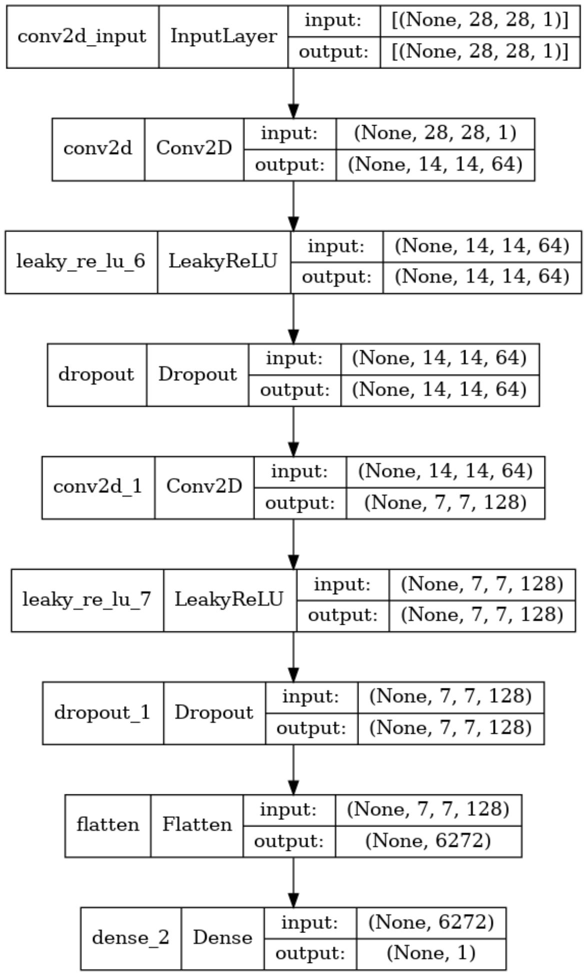
conv2d_input | InputLayer | input: | [(None, 28, 28, 1)]
output: | [(None, 28, 28, 1)]
conv2d | Conv2D | input: | (None, 28, 28, 1)
output: | (None, 14, 14, 64)
leaky_re_lu_6 | LeakyReLU | input: | (None, 14, 14, 64)
output: | (None, 14, 14, 64)
dropout | Dropout | input: | (None, 14, 14, 64)
output: | (None, 14, 14, 64)
conv2d_1 | Conv2D | input: | (None, 14, 14, 64)
output: | (None, 7, 7, 128)
leaky_re_lu_7 | LeakyReLU | input: | (None, 7, 7, 128)
output: | (None, 7, 7, 128)
dropout_1 | Dropout | input: | (None, 7, 7, 128)
output: | (None, 7, 7, 128)
flatten | Flatten | input: | (None, 7, 7, 128)
output: | (None, 6272)
dense_2 | Dense | input: | (None, 6272)
output: | (None, 1)

Figure 3: the Discrimnator neural network architecture.

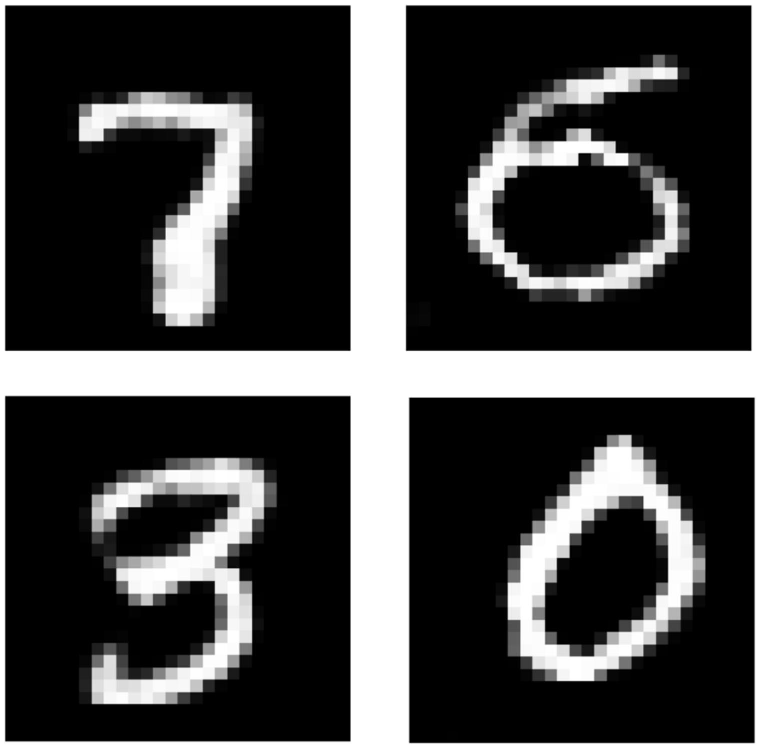

Figure 4: 4 examples of digits generated by the Generator network that is described in the text. The digits do not exist in the dataset, and have been "created" by the neural network.

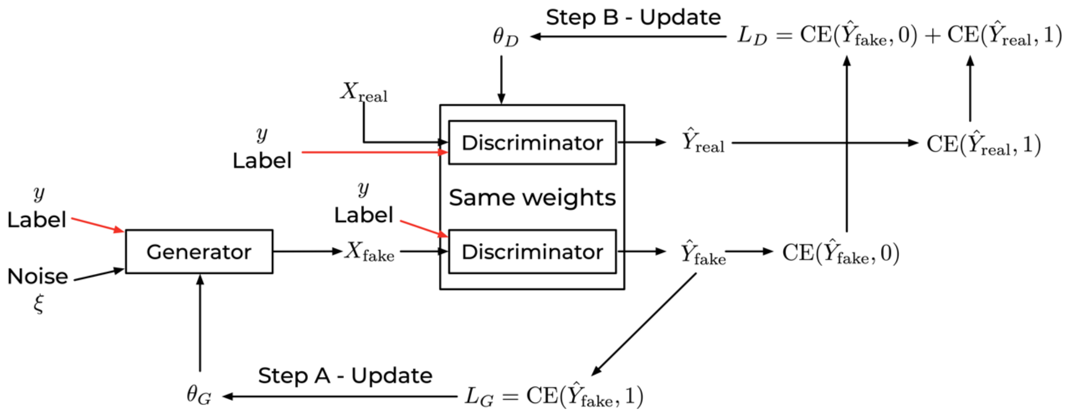

Figure 5: the training of CGAN system. In red is highlighted the role of the label that make possible for the Generator to create fake examples of specific classes.

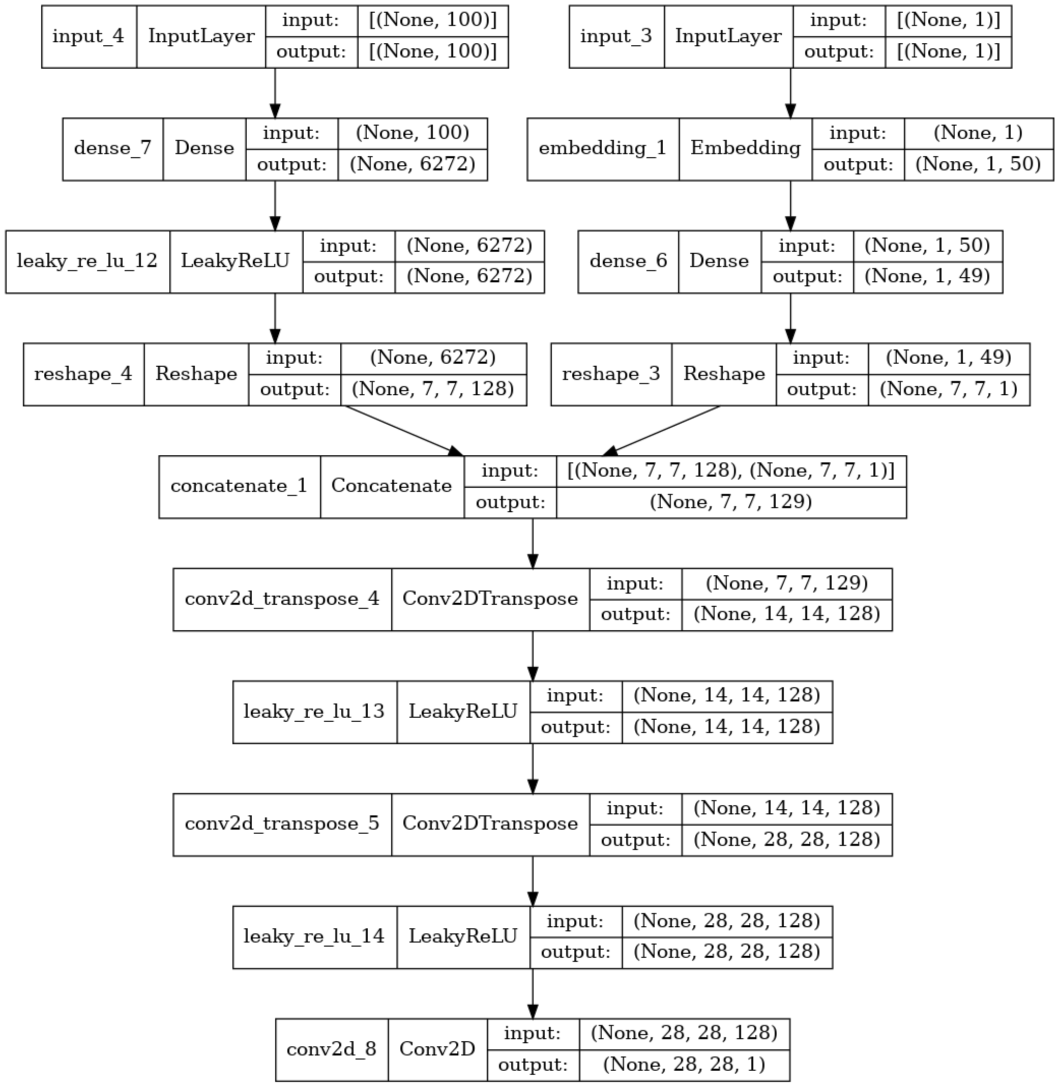
input_4 | InputLayer | input: [(None, 100)] | output: [(None, 100)]
input_3 | InputLayer | input: [(None, 1)] | output: [(None, 1)]
dense_7 | Dense | input: (None, 100) | output: (None, 6272)
embedding_1 | Embedding | input: (None, 1) | output: (None, 1, 50)
leaky_re_lu_12 | LeakyReLU | input: (None, 6272) | output: (None, 6272)
dense_6 | Dense | input: (None, 1, 50) | output: (None, 1, 49)
reshape_4 | Reshape | input: (None, 6272) | output: (None, 7, 7, 128)
reshape_3 | Reshape | input: (None, 1, 49) | output: (None, 7, 7, 1)
concatenate_1 | Concatenate | input: [(None, 7, 7, 128), (None, 7, 7, 1)] | output: (None, 7, 7, 129)
conv2d_transpose_4 | Conv2DTranspose | input: (None, 7, 7, 129) | output: (None, 14, 14, 128)
leaky_re_lu_13 | LeakyReLU | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
conv2d_transpose_5 | Conv2DTranspose | input: (None, 14, 14, 128) | output: (None, 28, 28, 128)
leaky_re_lu_14 | LeakyReLU | input: (None, 28, 28, 128) | output: (None, 28, 28, 128)
conv2d_8 | Conv2D | input: (None, 28, 28, 128) | output: (None, 28, 28, 1)

Figure 6: The Generator network architecture for CGAN.

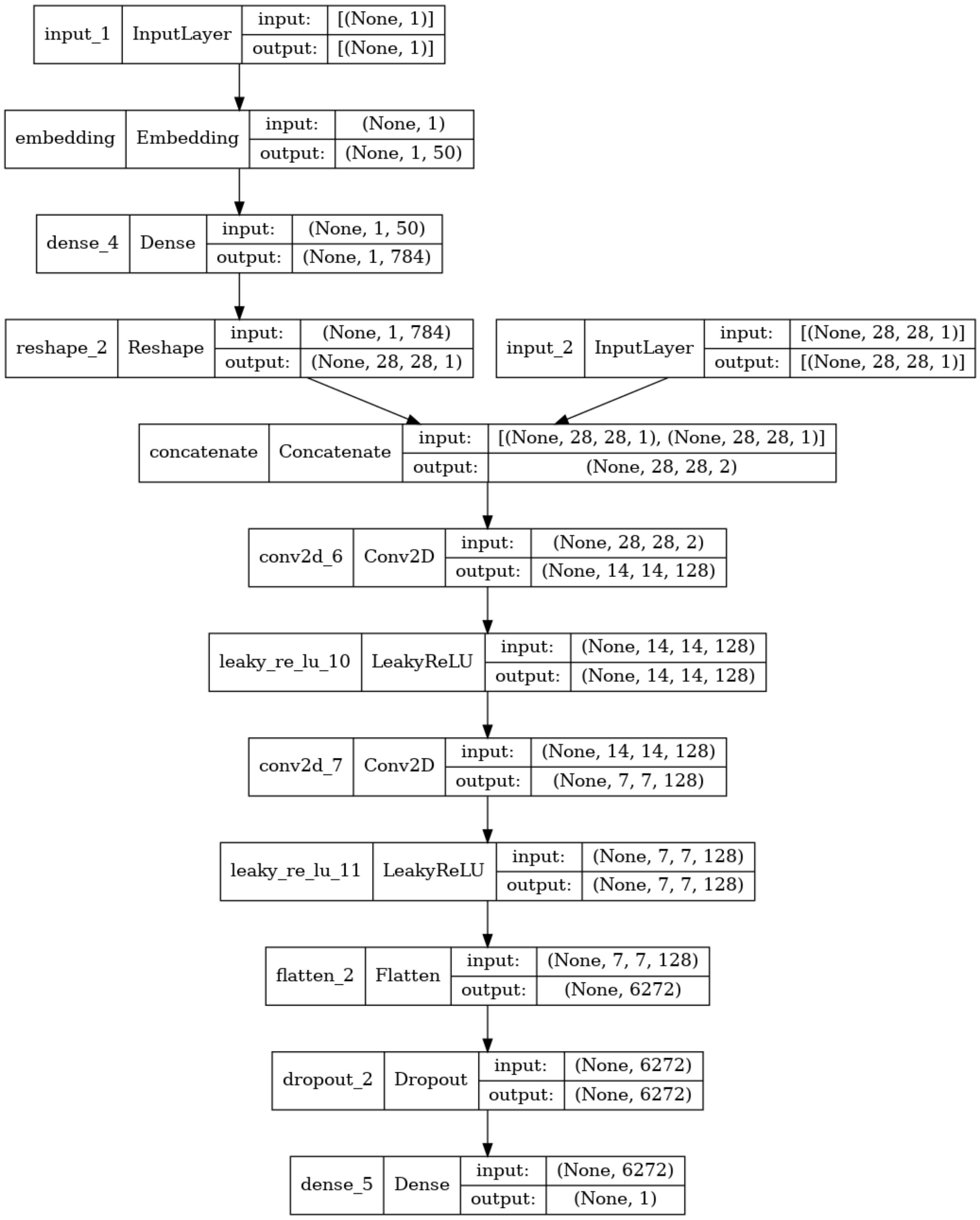
input_1 | InputLayer | input: [(None, 1)] | output: [(None, 1)]
embedding | Embedding | input: (None, 1) | output: (None, 1, 50)
dense_4 | Dense | input: (None, 1, 50) | output: (None, 1, 784)
reshape_2 | Reshape | input: (None, 1, 784) | output: (None, 28, 28, 1)
input_2 | InputLayer | input: [(None, 28, 28, 1)] | output: [(None, 28, 28, 1)]
concatenate | Concatenate | input: [(None, 28, 28, 1), (None, 28, 28, 1)] | output: (None, 28, 28, 2)
conv2d_6 | Conv2D | input: (None, 28, 28, 2) | output: (None, 14, 14, 128)
leaky_re_lu_10 | LeakyReLU | input: (None, 14, 14, 128) | output: (None, 14, 14, 128)
conv2d_7 | Conv2D | input: (None, 14, 14, 128) | output: (None, 7, 7, 128)
leaky_re_lu_11 | LeakyReLU | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
flatten_2 | Flatten | input: (None, 7, 7, 128) | output: (None, 6272)
dropout_2 | Dropout | input: (None, 6272) | output: (None, 6272)
dense_5 | Dense | input: (None, 6272) | output: (None, 1)

Figure 7: the Discriminator network architecture for a CGAN.