Group Project 2 – Group D

Machine Learning III

MBD 2020

Alexandre Bouamama, Tarek ElNoury, Guillermo Germade, Alain Grullón, Roberto Picón, Rebecca Rosser
IE HST

Summary

1. Generating MNIST data

[2 point] TODO 1. Explain the high-level idea of Generative Adversarial Nets.

The Generative Adversarial Nets or GANs is a Machine Learning technique. It relies on the competition of two nets within one same framework. The two nets are named “generator” and “discriminator”. The generator is a type of convolutional neural net whose role is to create (to “generate”) new instances of an object. For example, if we want to classify cat images, this net will try to create new images of cats. On the other side, the “discriminator” is a neural net which works on determining the authenticity of the object presented (To follow the same example, this net asks the question: Is this a true cat image?). During the training process, both entities are competing against each other, which in returns sharpens and improves their respective behaviours. This particular feature is called backpropagation. On a high-level, this competition between both nets enables the creation of more and more convincing images and detections features at each iteration.

[1 point] TODO 2. Both the generator and discriminator are convolutional neural nets. Which are the inputs and expected outputs of both of them before and after training?

To explain the different inputs and outputs, we will present the following figure that we elaborated using the website lucidchart.com:

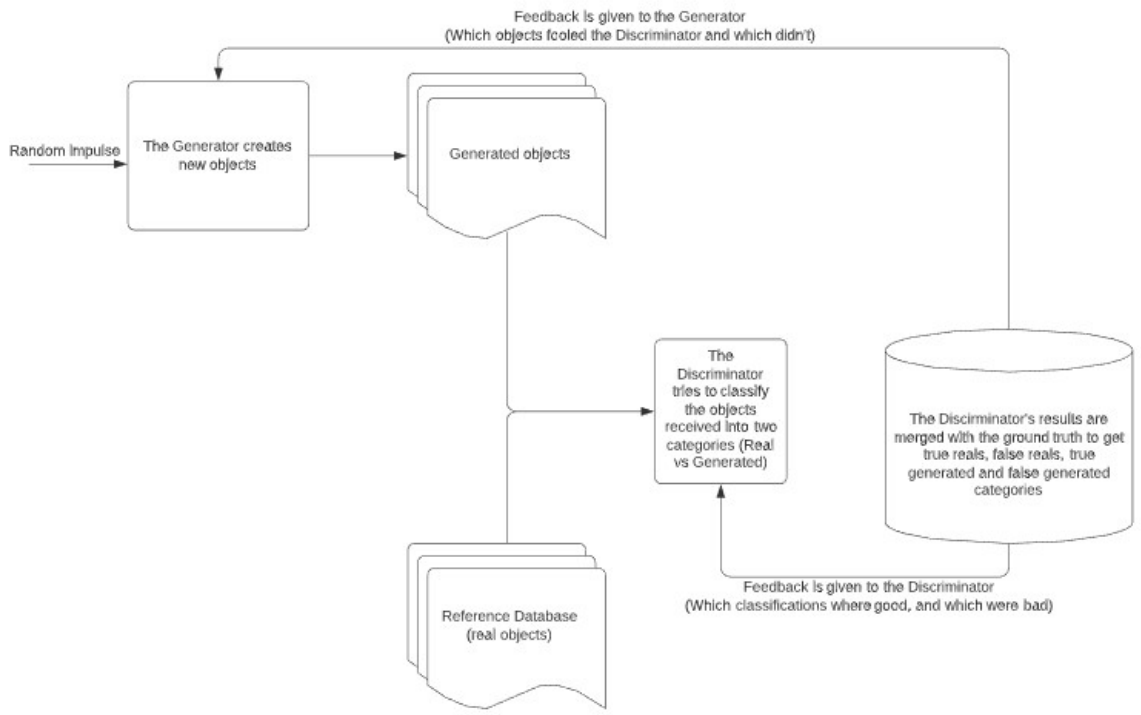

Figure 1: High-level chart of the inputs and outputs at each step of the GAN framework

As you can see from the chart, the cold-start problem is solved by giving a random impulse to the generator so that it creates the first set of random objects. Then, those objects are merged with reference objects (true images of cats, to follow our previous example) and the discriminator tries to classify them into Real or Generated objects. The result is then used as feedback to the Discriminator and the Generator (to take into account the falsely categorized data).

[1 point] TODO 3. The core functions are train and its subfunction train_step. Explain step by step what they are doing.

Since GANs are composed of two nets that are intertwined nets, it is of capital importance to understand each step of the training process to avoid a misallocation of weights, for example. The train function is at the core of this step, and to understand it, we first need to dig into how its subfunction *train_step* is built.

```
4 def train_step(images):
5     noise = tf.random.normal([BATCH_SIZE, noise_dim])
6 
7     with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
8       generated_images = generator(noise, training=True)
9 
10      real_output = discriminator(images, training=True)
11      fake_output = discriminator(generated_images, training=True)
12      gen_loss = generator_loss(fake_output)
13      disc_loss = discriminator_loss(real_output, fake_output)
14 
15    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
16    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
17 
18    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
19    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

Figure 2: train_step subfunction

The *train_step* subfunction starts with the random *noise* variable that will be used for the impulse. The GradientTape() step is then used to track each step of the classification of the data to ensure that the backpropagation is made accordingly. This is why the next step is to define the discriminator for both real and fake outputs (real_output and fake_output). The next two lines represent the loss of the generator when the output is fake (gen_loss) and the loss of the discriminator what the output is real and fake (disc_loss). The gradients of both the generator and the discriminator and then calculated and applied to the two neural nets. Now that we explained the *train_step* subfunction, we can look at the *train* function.

```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as we go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
```

Figure 3: train function

The *train* function first of all goes through all the epochs to keep track of the time spent using the *start* variable to lock in the starting time of the training process. The function loops through each image of the dataset and uses it as input for the *train_step* subfunction. The next step is to generate all the images of the current result and save (log) the model after each 15th epoch. Finally, the function prints the time used for the epoch and prints the final result.

2. Generate CIFAR10-images

[2.5 point] TODO 1. Complete the code for the Generator model. def make_generator_model() Note that now you have color images, so the output should be 32, 32, 3!

```python
def make_generator_model():
    # TODO1: Put your code here
    model = tf.keras.Sequential()

    model.add(layers.Dense(8*8*256, use_bias=False, input_shape=(100,)))

    model.add(layers.BatchNormalization())

    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((8, 8, 256)))

    assert model.output_shape == (None, 8, 8, 256)

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

    assert model.output_shape == (None, 8, 8, 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, 16, 16, 64) #upsampling to 16 (8*2)

    model.add(layers.BatchNormalization())

    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))

    assert model.output_shape == (None, 32, 32, 3) #upsampling to 32 (16*2)

    return model
```

Figure 4: Code of the generator model

[2,5 points] TODO 2. Complete the code for the Discriminator model.

```python
def make_discriminator_model():
    # TODO2: Put your code here
    model = tf.keras.Sequential()

    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',input_shape=[32, 32, 3])) #changing to the right shape

    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(3))

    return model
```

Figure 5: Code of the discriminator model

3. CONDITIONAL GENERATIVE ADVERSARIAL NETWORKS

[1 point] Explain what a Conditional Generative Adversarial network is.

The main issue of GANs is that you cannot tell your framework to give you a specific output (let's say an image of a Siamese cat to follow the same example as before). To answer to this problem, researchers have found the solution, and it is the CGAN or Conditional Generative Adversarial Network. The only difference with the traditional GAN framework is that there is a label condition added. This enables the network to produce an object conditioned on a class label.[1] This added level of specificity enables a way wider ranges of uses for the framework.

[+1 extra point] Create your own Conditional Generative Adversarial Network to generate conditioned samples in the Fashion Mnist dataset.

You can see the network in the notebook 03_cdcgan_fashion_mnist_template_GroupD.ipynb

[1] Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. arXiv 2014. arXiv preprint arXiv:1411.1784