GROUP D

MACHINE LEARNING ASSIGNMENT II

Spring 2020

IE Masters in Business Analytics and Big Data

Prof. Javier Gonzalez Dominguez

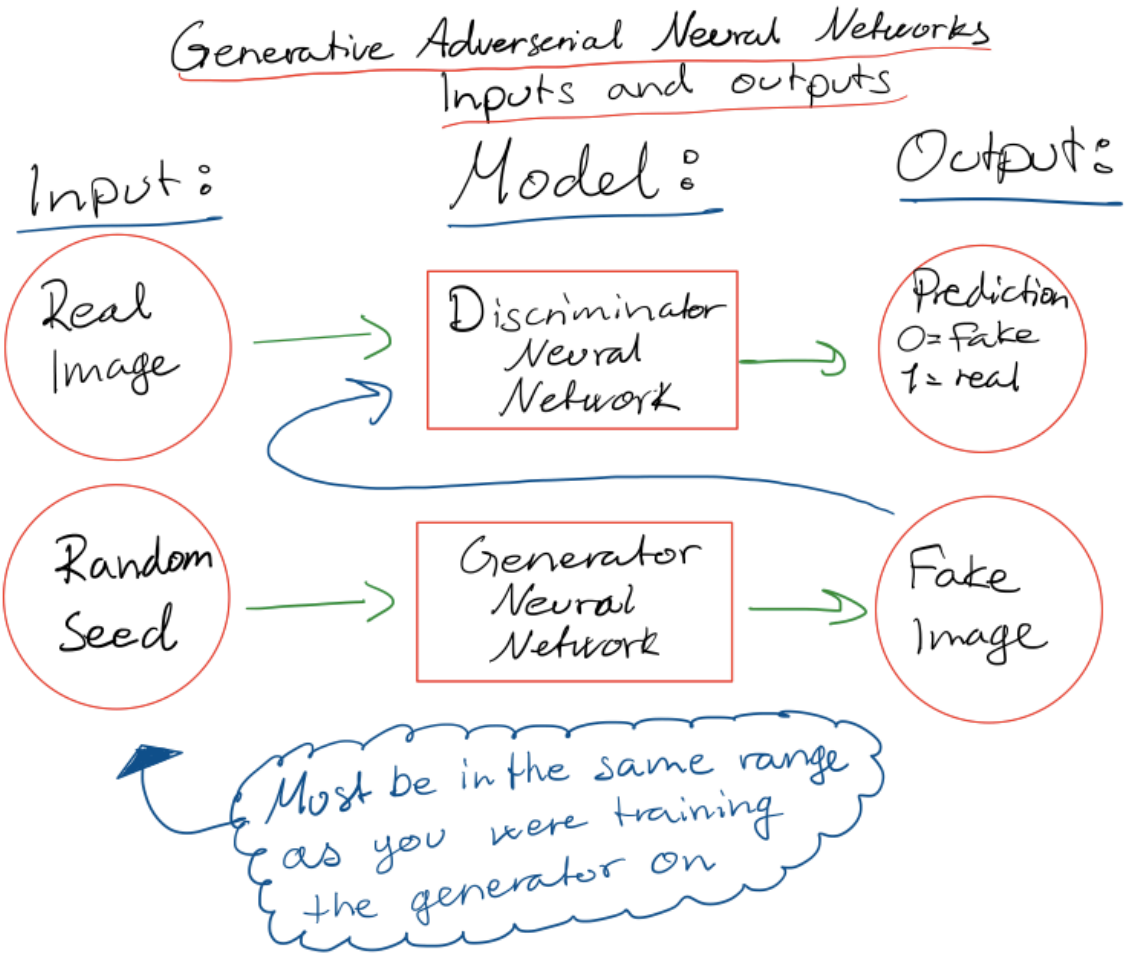
1. Generating MNIST data

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

Generative Adversarial Networks (GANs) was invented by Ian Goodfellow in 2014 and is one of the most brilliant ideas in the last 10 years within the field of Machine Learning. The general idea is to generate images, but now we are using them for many other things. The model uses variational autoencoders for the Generator to create new images, while it uses the Discriminator to classify the generated images from real images.

We have two models combined to one in GANs, where we call one model the "Generator", and the other one the "Discriminator", both in the category of Convolutional Neural Networks (CNN). The first, our Generator, does what the name says, it generates. What does it generate? Images, or to be exact, fake images to fool the discriminator to believe that the images we are feeding it is real. The generator never sees a real image, it rather gets noise as input, where the model does its best to create imitations of the data. The discriminator on the other hand is trained on real images, and when you feed it fake images, it either classifies them as real or fake, with binary values of ones and zeros. What is most important about the combination of these two models into one, is that the error obtained in the discriminator is passed to the generator, where the generator uses this as feedback on how to do a better job, working its best to reduce the error, and create what would be classified as real images. However, instead of classifying the image as real, it is more like a min-max game, where the discriminator will not be able to distinguish between real and fake images after the training of the whole architecture and the output of the discriminator will be 0.5. When that happens, we have successfully created a model that cannot distinguish between real and fake images, because the images created by the Generator has gotten so good that the Discriminator can't see the difference anymore.

[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?

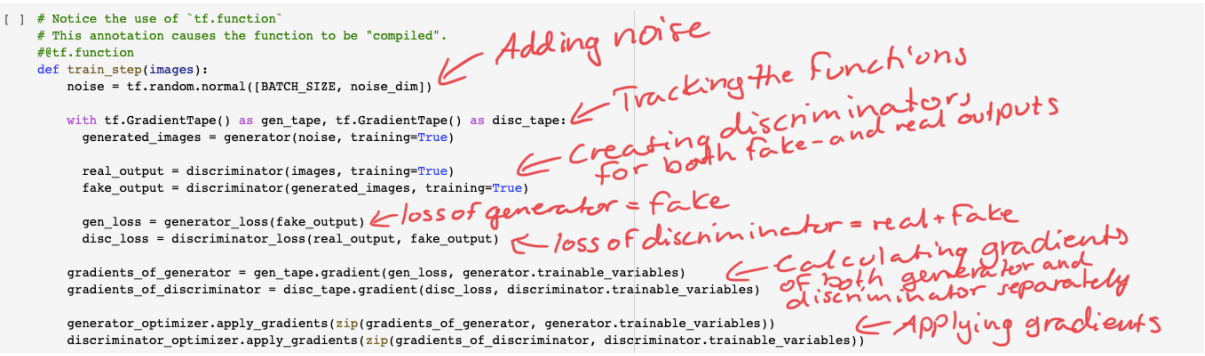


In the training of GANs, we need to make sure to distinguish between the two different CNNs we are building. The different models have very different tasks, and are working against each other, so that is why the drawing above looks so different for the two of them. In the Generator, we don't train the model on any false cases, and its objective is to fool the Discriminator. It takes random seeds as input, generates an image, and this image is then fed to the Discriminator together with real images. It is now up to the Discriminator to classify these as real- or fake images. This number output is used for the loss function, where it calculates the backpropagation based on the real input classified as ones. This is just one step, and we get more to detail in this with the 3rd question below. The Discriminator on the other hand is trained on both real- and fake images (generated by the Generator) that it classifies as either fake or real. The loss of the fake input and the loss of the real input is used to improve the Discriminator, which we will get into detail in the 3rd question.

After training, we hope that both our models have become experts in their tasks, as the Generator gets better at creating pictures that look real, the Discriminator gets better at separating them. However, at some point if we are doing it right, the Discriminator will no longer be able to see the difference between them, and the output we get will be closer to 0.5 where it reaches equilibrium.

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

The training for these types of Neural Networks (NN) are very different than for what we have seen earlier in the course. Since we are training two different models, but want to integrate them, we need to be careful about where each weight is modified, because modifying the wrong network could lead to wrong outputs, or even better outputs based on where we do the mistake. Better outputs in this case does not necessarily mean a better model, as we do want the generator to create new images based on the seed, and we do want the discriminator to not be able to tell the real images apart from the fake images. How are the *train* and *train_step* functions created? To get to the core function *train*, we first have to define the subfunction *train_step*.



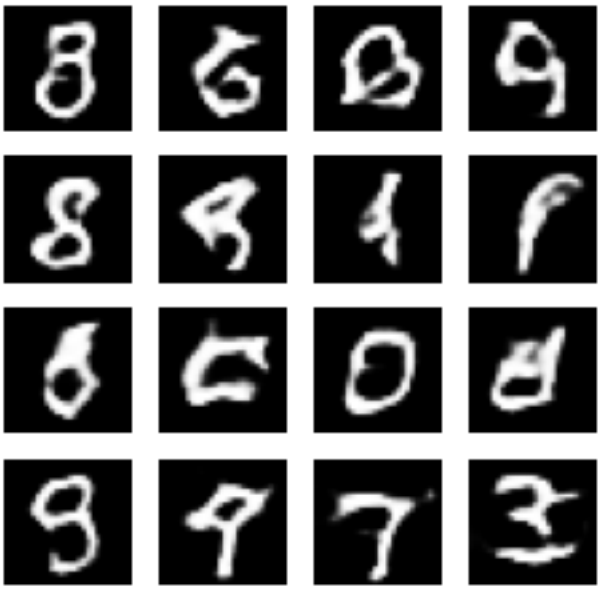
In *train_step* we are first introducing noise through random seeds based on our previous set batch size and noise dimensions. We go on to the *GradientTape()* that is used for the purpose of separating the derivates of the two NNs. What this is doing, is that it is tracking all the functions going through the NN, before it does backpropagation to figure out the derivates for each NN. We move on to creating separate discriminators for the real- and the fake outputs to be able to create the loss functions for both NN. The loss of the generator is only based on the fake outputs, while for the discriminator the loss of both the fake- and the real images is used for its loss function. This illustrates what we discussed in the previous question about training the different models. Next step is to calculate the gradients of both the generator and discriminators, again separately. Here we use the previously defined gradient tapes of both types and keeping them separate to avoid any crossover. We finish the *train_step* by applying the gradients for both our two types of NNs.

After we have defined our *train_step* function, we move to a simpler function to finish the setup of our training, of course implementing our more advanced *train_step* into it.

```
[ ]  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)
```

looping through number of epochs
keeping track of time spent
looping through each image
cleaning output
generates images of the current result
saving every 15th epoch
printing time used for the epoch
printing final result

We start by going through the number of epochs that we defined and keep track of the time spent here. Then we loop through for every epoch each image batch in our dataset by applying the *train_step* we defined above. We continue to produce images for the GIF as we go, where the images will be printed for every epoch, where we in the next code define that it will only be saved for every 15th epoch. These images are then saved into a checkpoint file that we later can download to our computer. While it's training, it will print for every new iteration of the entire dataset the time spent, which in our case was around 36 seconds with GPU from Google Collaboratory. Once the model is finished training, it will clear the output before it prints the image based on the model we used, number of epochs and the random seeds. Below you can see the image it printed for us:



2. Generate CIFAR10-images

[2.5 point] TODO 1. Complete the code for the Generator model.

```
def make_generator_model()
```

```
[ ] def make_generator_model():
        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) # 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, 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)
        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)

        return model
```

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

```
def make_generator_model():
```

```
[ ]  def make_discriminator_model():
         model = tf.keras.Sequential()
         model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                 input_shape=[32, 32, 3]))
         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
```

3. STAND ON THE SHOULDERS OF GIANTS

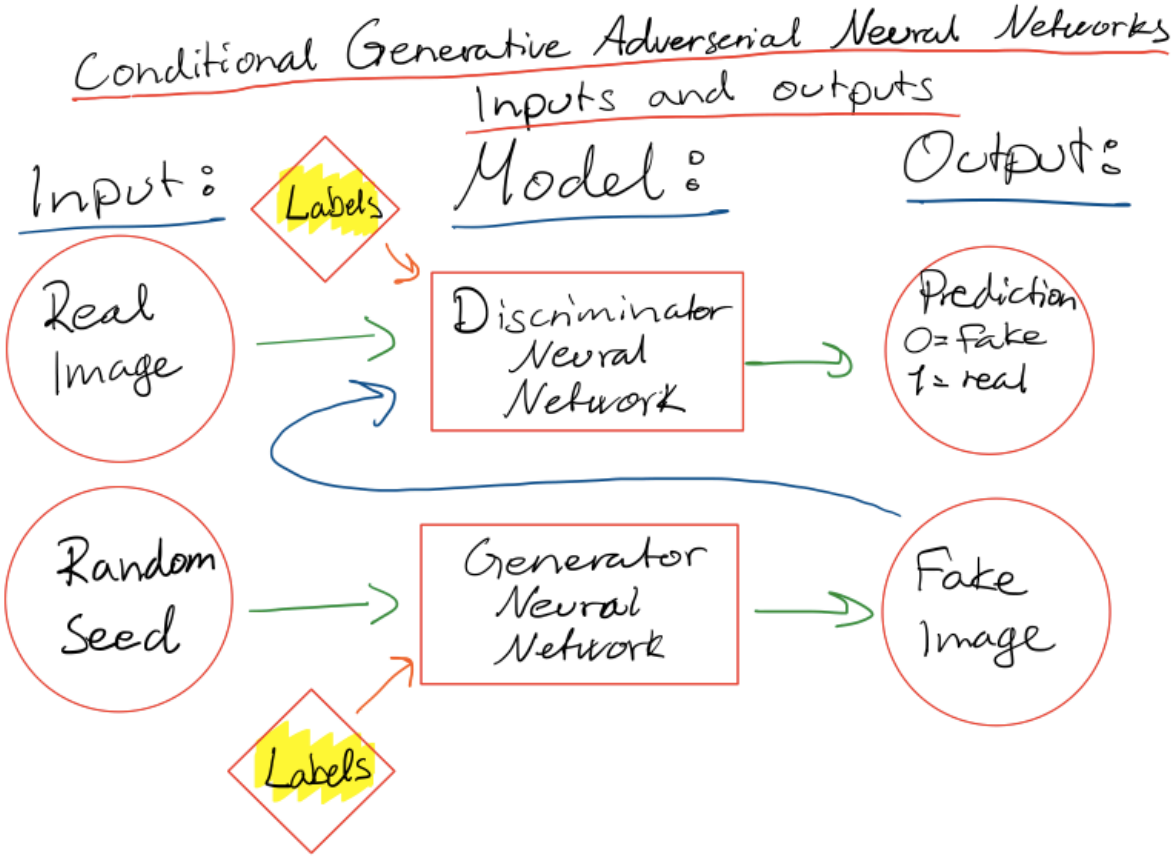
Conditional Generative adversarial networks.

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

The drawback of GANs is that you don't have control over the output. For example, in MNIST dataset, generated images may not represent all of 10 classes of clothes that GANs generated upon from, also we will see different classes all together in the output grid. So, if we want to control the output of GANs, such as to show only shoes or only shorts in all the output grids, in other words, if we want the GANs to produce a controlled output, Conditional GANs are needed.

For example, to get only one class of clothes as an output we introduce label condition representing classes of clothes to the GANs. By including the label condition to the GANs, we are able to control the output so that GANs generates only certain class/clothes.

In the below picture, we have included the condition into our GAN model, where we feed the label to both the Generator and the Discriminator for the output to generate images.



As you can see, CGANs architecture is very similar to GANs with exception of additional conditional layer.

[+2 extra points] Create your own Conditional Generative Adversarial Network to generate conditioned samples in the Fashion Mnist dataset.

For this exercise everything can be found in the notebook named "03_CDCGAN_FASSION_MNIST.ipynb". However, to make it a little bit easier to review, we have included both the Generator and Discriminator we defined below:

```
[ ] def make_generator_model(n_classes):

        # Inputs
        inputs = tf.keras.Input(shape=(1,))

        # Embedding to include categorical input
        x = layers.Embedding(n_classes, 50)(inputs)

        # linear multiplication
        x = layers.Dense(7*7, use_bias=False)(x)

        # Reshaping the additional channel
        x = layers.Reshape((7, 7, 1))(x)

        # image generator input
        lat = tf.keras.Input(shape = (100,))

        # Adding the foundation for 7x7 image
        generate = layers.Dense(128*7*7, use_bias=False)(lat)
        generate = layers.LeakyReLU()(generate)
        generate = layers.Reshape((7, 7, 128))(generate)

        # concatenating the channels as one channel
        merge = layers.Concatenate()([generate, x])

        # Upsampling to 14x14
        generate = layers.Conv2DTranspose(filters = 128, kernel_size = (4, 4), strides = (2, 2), padding='same')(merge)
        generate = layers.LeakyReLU()(generate)

        # Upsampling to 28x28
        generate = layers.Conv2DTranspose(filters = 64, kernel_size = (4, 4), strides = (2, 2), padding = "same")(generate)
        generate = layers.LeakyReLU()(generate)

        # Outputs
        outputs = layers.Conv2D(filters = 1, kernel_size = (7, 7), padding = "same", activation = "tanh")(generate)

        # Defining our model
        model = tf.keras.Model(inputs=[lat, inputs], outputs=outputs)

        return model
```

```
[ ] def make_discriminator_model(n_classes, in_shape = (28 , 28, 1)):

        # Inputs
        inputs = tf.keras.Input(shape=(1,))

        # Embedding to include categorical input
        x = layers.Embedding(n_classes, 50)(inputs)

        # Scaling to input dimension using linear activation
        x = layers.Dense(in_shape[0] * in_shape[1], use_bias=False)(x)

        # Reshaping the additional channel
        x = layers.Reshape((in_shape[0], in_shape[1], 1))(x)

        # Adding the image
        image_input = tf.keras.Input(shape = in_shape)

        # concatenating the channels as one channel
        merge = layers.Concatenate()([image_input, x])

        # downsampling
        fe = layers.Conv2D(filters = 128, kernel_size = (3,3), strides=(2,2), padding="same")(merge)
        fe = layers.LeakyReLU()(fe)

        # downsampling
        fe = layers.Conv2D(filters = 128, kernel_size = (3,3), strides=(2,2), padding="same")(fe)
        fe = layers.LeakyReLU()(fe)

        # flattening the feature maps
        fe = layers.Flatten()(fe)

        # adding dropout
        fe = layers.Dropout(0.3)(fe)

        # outputs
        outputs = layers.Dense(units = 1)(fe)

        # defining our model
        model = Model(inputs = [image_input, inputs], outputs = outputs)

        return model
```