MACHINE LEARNING III

Practice II: Deep GANs

Allison Black
Dennis Pedersen
Eleonora Jimenez
Georgy Eremin
Leonard Valencia
Rayan Alghamdi

2021 | JANUARY **TEAM C**

TABLE OF CONTENTS

1.1 High-level idea of Generative Adversarial Nets

Generative Adversarial Nets Overview

Invented in 2014 by a Stanford graduate Ian J. Goodfellow, generative adversarial networks (GANs) is a subclass of unsupervised machine learning frameworks. The main use case of these networks is image manipulation and generation which could be applied in many different fields: drug discovery, art, privacy maintenance, music generation and others [1]. GANs are based on two competitive models – the discriminator and generator – which work cooperatively to consistently improve the overall network ability to capture and copy variation within a dataset [2]. This model cooperation is not straightforward though. The discriminator tries to help generator to create images comparable to the real sample (penalizing the generator for wrong images) whereas the generator tries to fake the discriminator (minimizing the cost function). The whole process could be considered as a zero sum game – one model's gain is the other model's loss.

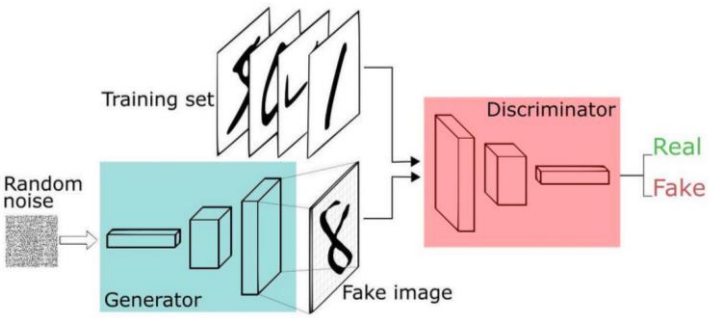


Figure 1: GANs visualisation

Key Features

- Generate enough data when lacking data
- No human supervision
- Real data sample is not seen by the generator
- Low computational cost

Discriminator

Discriminative algorithms perform binary classification for the input data. Given a set of features, those algorithms predict a category that the data point belongs to. For instance, discriminator algorithms can be trained to identify if a specific email is spam or not based on its content. The classification decision is mathematically expressed by the conditional probability concept: the probability of data input referring to a specific label given the features it contains [3]. Discriminator in GANs is a convolutional neural network. Its main function is to identify if generated images are comparable to the real sample.

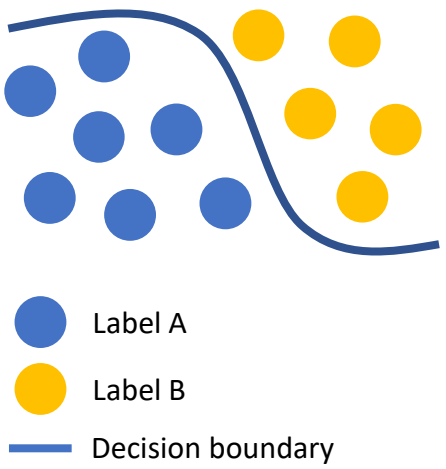


Figure 2: Discriminator visualisation

$$p\,(y|x) = probability\ of\ y\ (label)\ given\ x\ (features)$$

Generator

Generative algorithms perform feature prediction for the input data. Given a set of labels, those algorithms predict features that the data point might have. It comes as no surprise that this algorithms are completely opposite to discriminative algorithms. For instance, generative algorithms can be trained to identify how likely specific features are, assuming that the email is spam. The feature prediction is also mathematically expressed by the conditional probability concept: the probability of data input containing specific features given the label it has. Instead of decision boundary, generative algorithms learn the distribution of individual classes. Generator in GANs is de-convolutional neural network. Its main function is to fool discriminator by generating images comparable to the real dataset.

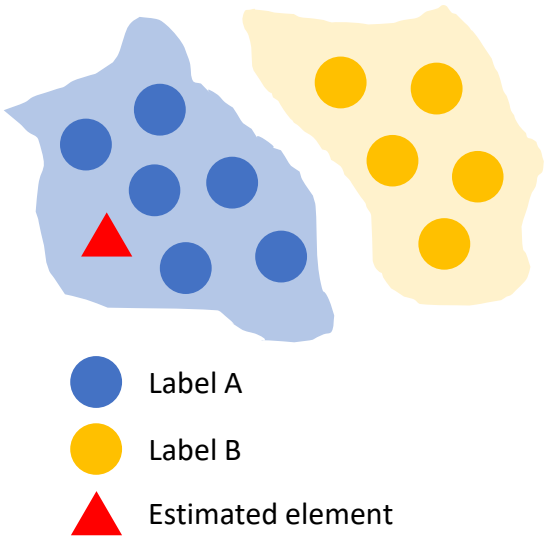


Figure 3: Generator visualisation

$$p\,(x|y) = \ probability\ of\ x\ (features)\ given\ y\ (labels)$$

Generative Adversarial Networks Compilation

Finally, discriminative and generative algorithms are combined to generate accurate high quality content. Firstly, generator consumes randomly generated numbers (noise) and returns an image. This image, alongside with images from actual dataset, is fed into the discriminator. The discriminator takesboth real and fake images and classifies them. The classification process is done through the probability calculation – a number between 0 and 1. If the output of the discriminator is 1, the processed image is classified as authentic. On the other hand, 0 means that the image is fake. The main point of the model is having the double feedback loop for discriminator and generator. Discriminator's cost function is connected to the real dataset (true images) whereas generator's cost function is connected to the feedback from the discriminator. Those cost functions are opposing each other (as stated before – zero sum game). Being a CNN, discriminator is taking an image and downsamples it to produce a probability of the image having a specific label. A generator, which represents a DNN, takes a vector of random noise and upsamples it into an image. What is reasonable to emphasize that the generator does not see any images from a real sample. The only feedback the generator is provided with is the numeric output from the discriminator about the fakery of a newly generated image.

Generative Adversarial Networks latest trends

The problem of GANs is their training difficulty. The reason is that there are two cost functions that are optimized during training. Unfortunately, it's challenging to simultaneously improve both functions as the win of one is the loss of the other. This problem becomes more difficult if there is a need to work with high-quality pictures. As stated in the NVIDIA article, there is a trade-off between the image resolution and potential variation captured. Moreover, while working with higher resolution, it's harder to generate images which are compared to the real images, resulting into the gradient problem. The solution NVIDIA suggested is the progressive growth of both the generator and discriminator, adding new layers in the training progress [4].

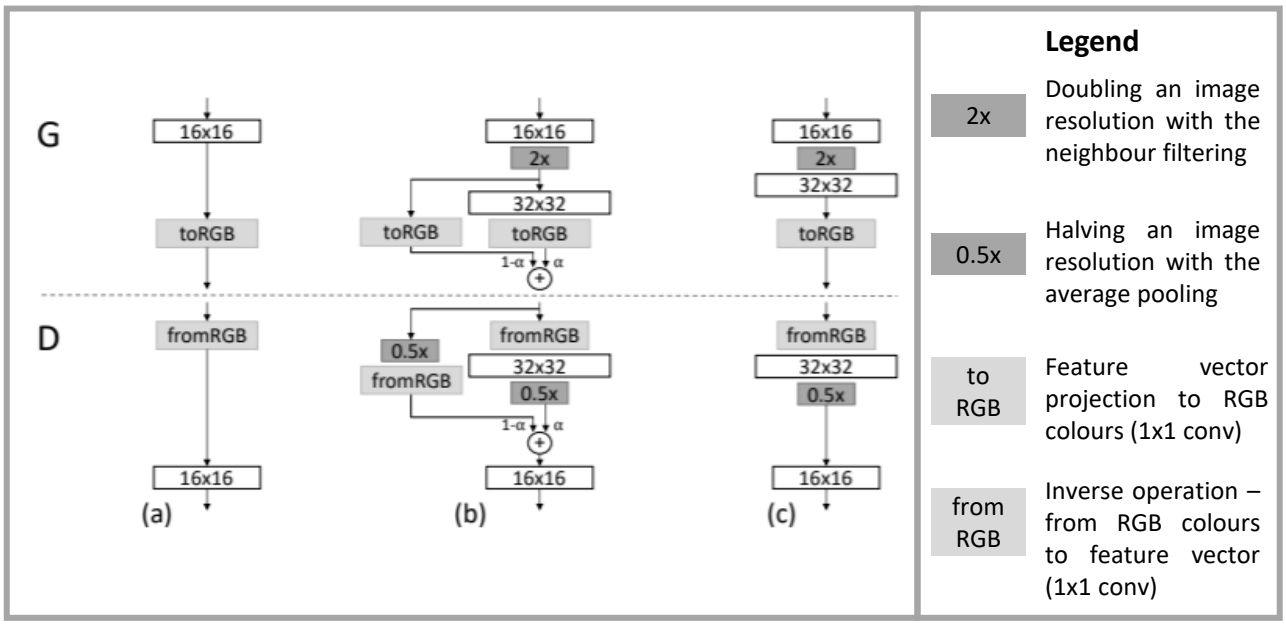


Figure 4: GAN resolution transition to capture high resolution images

As shown on the diagram (see Figure 4), there is a smooth transition from the 16x16 image (a) to the 32x32 image (c) with a transition layer (b). Same layers could be stacked upon each other to get high-quality images (1024x1024 resolution). It turned out that this type of model architecture was straightforward, fastest to train and provided the best performance.

What seemed particularly impressive is that NVIDIA has also found a way to increase GAN's variation through mini batching without adding new parameters. To solve the limited capacity of GANs in capturing subset variation of the training datasets, feature statistics are computed across a mini batch (not only from individual images as before). This implementation was done through adding an additional layer at the end of the discriminator which produces an array of statistics [5]. Capturing standard deviation, this array is averaged and used as an additional feature (see Figure 5).

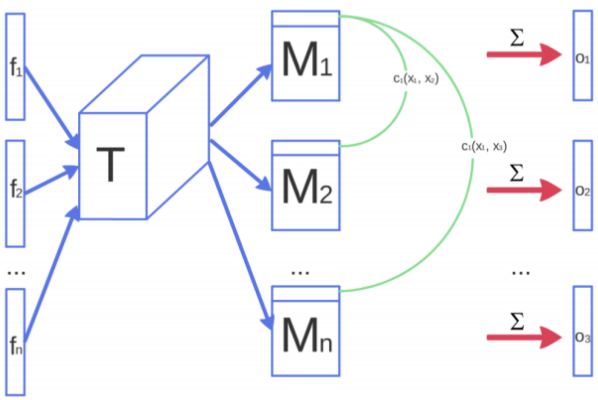


Figure 5: Mini batching discrimination

1.2 Generator & Discriminator inputs and expected outputs

Generator Technical Overview

Before any training is performed, the generator's input is random noise. The random noise introduced to the generator can follow any distribution as it has been shown not to have a significant impact [6]. This untrained generator model creates a fake image (see Figure 6), which given the random input does not resemble any recognizable image. Hence, the generator's output before training is a fake random image.

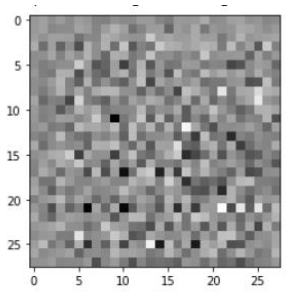


Figure 6: Untrained generator output

The generator model (Figure 7) receives as mentioned a random input, which in this case is in the form of a vector which data follows a normal (Gaussian) distribution. This is done by taking the dense layer which then is upsampled by the Conv2DTranspose layers until the image size is reached. The desired image size (output) is 28x28x1, thus a two-dimensional image. As seen, the generator model is also normalizing its input with batch_normalization. Since its training parameter is set to true, it will normalize the inputs with the mean and variance from the current batch [7]. All layers, apart from the output layer are activated through 'LeakyReLU'. 'LeakyReLU' tries to mitigate the issue around 'dying' ReLU, where the weights might be updated to deactivate a neuron permanently. The output layer is activated through 'tanh' which ensures that the output value is compressed into the range of -1 to +1 with a centre around zero [8].

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12544)	1254400
batch_normalization (BatchNo	(None, 12544)	50176
leaky_re_lu (LeakyReLU)	(None, 12544)	0
reshape (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose (Conv2DTran	(None, 7, 7, 128)	819200
batch_normalization_1 (Batch	(None, 7, 7, 128)	512
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_1 (Conv2DTr	(None, 14, 14, 64)	204800
batch_normalization_2 (Batch	(None, 14, 14, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_2 (Conv2DTr	(None, 28, 28, 1)	1600

Figure 7: Generator model summary

The generator's loss function is used for backpropagation through the discriminator and the generator to obtain the new gradients (Figure 5). Those gradients are then used to change the weights of the generator, essentially the input for the generator after training, in order to produce new fake images for the discriminator to classify. The loss function for the generator quantifies the generators performance in terms of 'tricking' the discriminator into classifying a fake image as real. Hence, the discriminator classifications will be compared with the fake images to an array of ones (1s) [9].

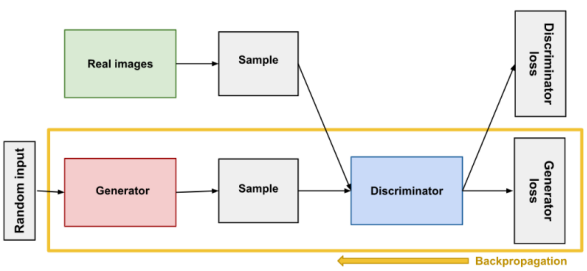


Figure 8: Generator training visualisation

Discriminator Technical Overview

The discriminator is a classifier trying to distinguish between real and fake images. Before training, the discriminator receives a real and a fake image as its two inputs and classifies each of them as either real or fake as its output. In the given notebook (01_dcgan_mnist.ipynb) the discriminator model will produce positive numbers for real images and negative numbers for fake images.

```
print(decision)
```

```
tf.Tensor([[-0.00253091]], shape=(1, 1), dtype=float32)
```

Figure 9: Discriminator output example

As the first image generated (see Figure 6) clearly is a fake image, the output before training for the discriminator is a negative number as shown in Figure 9.

The discriminator model is similar to the generator model, and also setup as a sequential model (see Figure 10). The input is a set of two-dimensional images (28x28x1). These images are downsampled through the Conv2D layers to its dense output which is either a negative or positive number. The dropout layers are used to generalize the output of precedent layers, reducing overfitting.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	1664
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	204928
leaky_re_lu_4 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 1)	6273

Figure 10: Discriminator model summary

The loss function for the discriminator computes the discriminator performance in terms of recognizing real and fake images. The real images are compared to an array of ones (1s), whereas the fake images are compared to an array of zeros (0s), cf. Appendix 1. In a way it can be said that the generator and discriminator compete against each other, where the generator's loss function decreases when it is able to 'fool' the discriminator. The discriminator will classify a fake image as real and even though this is a wrong outcome for the discriminator, it will help the generator to improve [10]. After training the input is the same for the discriminator. It will continue to receive real and fake images as its input. Similar to the generator, the discriminator also uses backpropagation to improve its weights and will use those for the next iteration in order to improve its classification performance after training (see Figure 11).

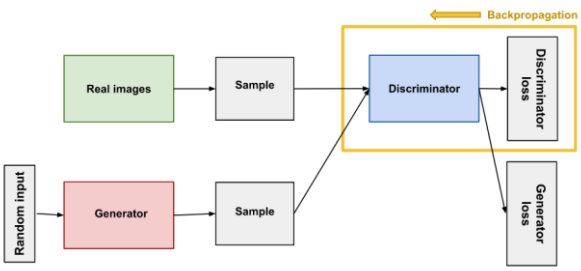


Figure 11: Discriminator training visualisation

The first part of the report (Question 1) covered the theoretical description of GANs framework. This part was devoted to a more technical description of each network's component. The full compilation of technical layers can be found in the cf. Appendix 2 [11].

1.3 Train & Train_step function explanations

Train function explanation

1. The first step of the train function is to define the for loop to ensure that the function runs the number of desired epochs as well as the desired image batches. In this example the number of epochs has been set to 50 earlier in the code (cf. Appendix 3). Defining start = time.time() allows the train function to calculate the time it takes to run each epoch, which is displayed later in the code. This line of code is not critical for the functionality, rather for timing analysis of training which is critically important in the current model.

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

    for image_batch in dataset:
      train_step(image_batch)
```

2. Afterwards, for each epoch and for each batch, the image is cleared and a new image generated. The 'display.clear_output(wait=True)' part enables updating the image in the loop. At the next stage the images are generated by the generator, and the epoch number is increased by 1. The model also includes a 'seed' as input in order to create the image.

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

3. The model is saved every 15 epochs. It should be noted that the number of epochs after which the model saves the result can be different from 15. Also, the code prints the time it takes for each epoch to run. Even though the train model works without printing the time per epoch it is useful information in terms of usability and scalability.

```
# 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))
```

4. As a final step the last epoch is generated and its images saved.

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

1.3 Train & Train_step function explanations

Train_step function explanation

1. The first step is to define the input noise. In this case tf.random.normal produces random values from a normal distribution. As discussed in section 1.2, it would have been possible to choose a different distribution as well without any significant impact on the end result of the model. As seen, noise is also defined in such a way that it takes the desired batch size and noise dim into account which were set to 256, and 100 respectively.

```
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
```

2. In this next part, the GradientTape() function allows us to create a customized training loop. Specifically, it creates multiple gradients at the same time and has the following components: model architecture, loss function, optimizer and step function. At this stage it is established which output is real (*real_output*) and which one is fake (*fake_output*). For the *real_output* variable, the discriminator function input is the set of real images. Conversely, the discriminator for the fake output receives the generated images by the generator as input. Then, the two loss functions are defined according to the outputs. The generator loss merely takes the fake output, which was created by the generator, into account whereas the discriminator loss is defined from both the real and the fake outputs.

```
with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
  generated_images = generator(noise, training=True)

  real_output = discriminator(images, training=True)
  fake_output = discriminator(generated_images, training=True)
  gen_loss = generator_loss(fake_output)
  disc_loss = discriminator_loss(real_output, fake_output)
```

3. Both gradients make use of the tf.GradientTape() function which is an essential part used for backpropagation. Using tf.GradienTape() enables TensorFlow to automatically calculate the gradient using the 'recorded' operations with the so called reverse mode differentiation [12]. The tf.GradientTape() inputs are the previously defined generator and discriminator losses for the generator and discriminator gradients respectively.

```
gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
```

4. As a final step an optimizer is defined and applied. The optimizer takes previously defined gradients of loss from the generator as well as the discriminator into account. Optimizers ensure the model gradients/weights are updated according to the loss functions. Hence, optimizers are paramount for the train_step function to work as intended.

```
generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

This part contains the step by step explanation of *train* and *train_step* functions. Full models of both functions could be found in the cf. Appendix 4-5.

GENERATE CIFAR10-IMAGES

2.1 Generator model code

As a foundation basis, we have taken the first notebook (01_dcgan_mnist.ipynb) and understood that the code requires specific modifications. Firstly, we investigated the required output of the generator model (which is 32x32 RGB). This is displayed within our code: we have changed the output shape of the final layer (1). *None* is used to establish the batch size, followed by the output shape 32x32x3. Next, we move to the start of the function definition. We establish the initial dense layer which takes the 8x8x256 as an input (2). Numbers were chosen due to the fact that 8 is a multiple of 32. The third number in the input – 256 – represents the number of filters in convolution and was chosen because of the primary image scale definition. After *BatchNormalization* and *LeakyReLU* layers (which were discussed in previous parts of the report), there was a need to reshape the layer output to add the value for the batch size (3). Steps (4) and (5) portray the deconvolution layer, where image size is upsampling and the number of filters is decreasing. In both steps we used the standard 5x5 kernel size, padding and no bias vector. Strides are increasing due to the image size increase. Lastly, we changed the number of filters to 3 (RGB) to fulfil the criteria of the model output (6).

```
def make_generator_model():
    # TODO1: Put your code here
    model = tf.keras.Sequential()                                                    (2)
    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)))                                           (3)
    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)                                   (4)
    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)                                  (5)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', use_bias=False, \   (6)
                                     activation='tanh'))
    assert model.output_shape == (None, 32, 32, 3)                                   (1)
    return model
```

2.2 Discriminator model code

Similarly to the previous part, we have used the same notebook (01_dcgan_mnist.ipynb) as our basis. The discriminator takes the image, produced by the generator, and downsamples it to a single value, concluding if the image is fake or not. Therefore, the output of the final dense layer is only 1 value (1). Then, it is important to match the output of the generator with the input of the discriminator (2). In this layer it is necessary to change the number of convolutional filters. This number increases throughout sequential layers in the network due to the fact that we perform deconvolution and downsampling the image (3-5). As before, we are using the standard 5x5 kernel with 2-width and length strides to move through the image. The dropout layer is added after each deconvolution + leaky relu layers to reduce the overfitting of the discriminator. Our team has added one more layer of deconvolution (compared to the first notebook) with 1-width and length strides as it turned out to produce better image quality (5).

```
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]))
    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.Conv2D(256, (5, 5), strides=(1, 1), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model
```

Combining both neural networks, we have established the following training parameters:

- buffer size = 50000
- batch size = 256
- epochs = 100

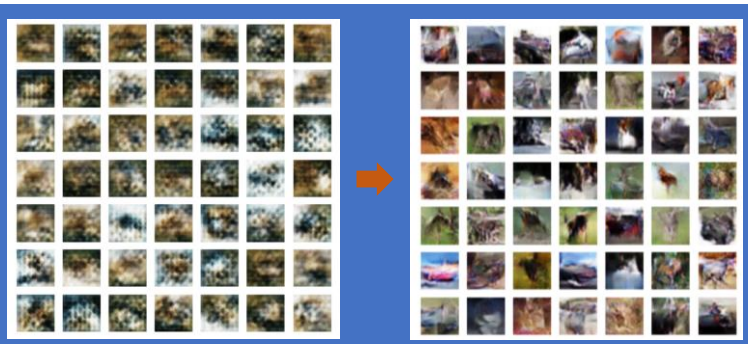


Figure 12: CIFAR10 GAN training progress

To sum up, starting from the random noise vector, our GAN was able to improve through out the process and produce recognisable images (see Figure 12). The whole training progress is represented in the gif attached to this report.

3.1 Conditional GAN overview

Theoretical cGAN overview

Even though GANs proved to be an exceptional combination with creative and effective outputs, there are still drawbacks to be considered. The main weak point of traditional GANs is the lack of control over their output. For instance, from all of the different labels it produces only several of them (without generating all labels) [13]. A conditional generative adversarial network (cGAN) is a kind of GAN which as the name suggests includes a conditional way of generating fake images by the generator [14]. CGANs include similar building blocks as the traditional GANs discussed in the first section of the report: a generator with random noise input, a discriminator that takes real and generated images as an input, and a loss function for both models. The difference is the additional one hot encoded input of image labels in both the generator and discriminator (see Figure 13). This implementation allows us to control the output of the model, creating targeted images for each label (or creating only one specific label). For instance, when working with the Fashion MNIST dataset our group was able to implement the model that produces each label of clothing in a separate column.

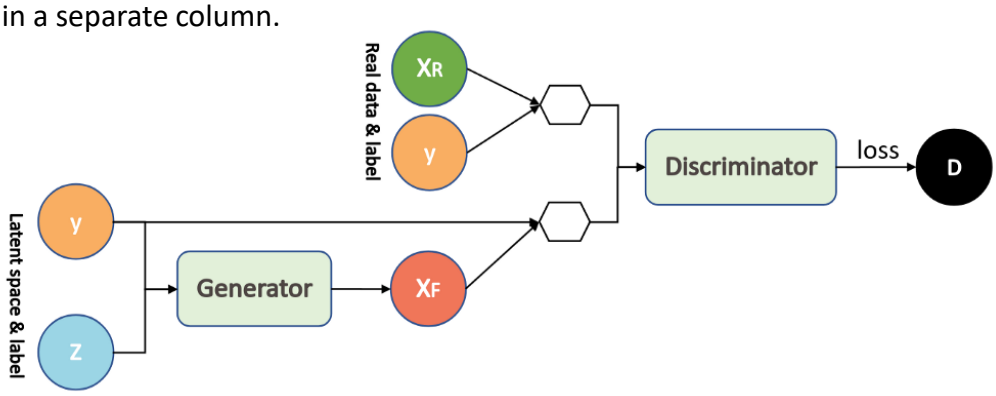


Figure 13: cGAN visualisation with additional one hot encoded label inputs

Technical cGAN overview

From a technical perspective, cGANs have several differences compared to traditional GANs. As discussed in the theoretical cGAN overview, label input leads to multiclass classification which emphasizes the use of the conditional probability concept. There are several ways to incorporate the desired condition into the generator and discriminator. It is considered good practice to make use of an embedded layer which is followed by a connected layer. The connected layer should be activated linearly in order to scale the embedding to the desired size of the image. It must then be concatenated into the model as an additional channel/feature map. The full compilation of cGAN's technical layers can be found in the cf. Appendix 6. Since the cGAN has multiple inputs, as the desired condition is added, it is recommended to use the functional API instead of the sequential API used for the GAN model in the previous sections. The main advantage of the functional API usage is the flexibility of input and output definition as well as the ability to connect models with shared layers [15].

3.2 Conditional GAN for the Fashion MNIST dataset

Generator model

Admittedly, the proposed blog *"How to Develop a Conditional GAN (cGAN) From Scratch"* by Jason Brownlee helped us to build the cGAN for the Fashion MNIST dataset. Yet, it still required several coding corrections from our team to adjust it to further notebook sections. Firstly, we have defined the generator model. As our notebook is built in the way that the optimizer and the cost function are defined in a separate section, we had to delete the *complie* method from the generator definition (1). We also defined function variables outside of the function itself: *latent_dim* = 100, *n_classes* = 10. This was done to match the noise and label input when the model is compiled. The definition of other layers are left untouched as they proved to be working well.

```
def make_generator_model(latent_dim, n_classes):

  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  gen = Dense(n_nodes)(in_lat)
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((7, 7, 128))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(merge)
  gen = LeakyReLU(alpha=0.2)(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # output
  out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model
```

1

3.2 Conditional GAN for the Fashion MNIST dataset

Discriminator model

Considering the discriminator model, our group has also implemented several changes. In steps (1) and (2), we have manually established the input image dimensions with linear activation and the input shape (*in_shape* = (28,28,1)). Afterwards, similarly to the generator, we have deleted the *compile* method at the end of the model definition (3). Lastly, we had to decide between the 'sigmoid' or 'tanh' activation function (4). Following a piece of advice in the instruction, we have tried to implement the 'tanh' activation into the discriminator model. However, during the training process we received black squares as the cGAN output. We then decided to attempt the cGAN training with the 'sigmoid' activation in the discriminator which proved to be a working strategy. The whole training progress is represented in gif attached to this report.

```
def make_discriminator_model(n_classes):
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # scale up to image dimensions with linear activation
  n_nodes = 28 * 28
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((28, 28, 1))(li)
  # image input
  in_image = Input(shape=in_shape)
  # concat label as a channel
  merge = Concatenate()([in_image, li])
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(merge)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output
  out_layer = Dense(1, activation='sigmoid')(fe)
  # define model
  model = Model([in_image, in_label], out_layer)
  return model
```

1 2 4 3

REFERENCES

1. 6 Unique GANs Use Cases (2019). Retrieved from: https://medium.com/@ODSC/6-unique-gans-use-cases-24cab2aa924d
2. What are Generative Adversarial Networks (GANs) and how do they work? (2018). Retrieved from: https://www.youtube.com/watch?v=-Upj_VhjTBs
3. Generative Artificial Intelligence (2018). Retrieved from: https://www.youtube.com/watch?v=PhCM3qoRZHE
4. Progressive Growing Of GANs For Improved Quality, Stability, And Variation (2018). Retrieved from: https://arxiv.org/pdf/1710.10196.pdf
5. Improved Techniques for Training GANs (2016). Retrieved from: https://arxiv.org/pdf/1606.03498v1.pdf
6. Generative Adversarial Networks – GAN Anatomy – The Generator. Retrieved from: https://developers.google.com/machine-learning/gan/generator
7. TensorFlow Core v2.4.0 – Python – BatchNormalization. Retrieved from: https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization
8. CS231n Convolutional Neural Networks for Visual Recognition. Retrieved from: https://cs231n.github.io/neural-networks-1/
9. Deep Convolutional Generative Adversarial Network. Retrieved from: https://www.tensorflow.org/tutorials/generative/dcgan
10. GAN Lab. Retrieved from: https://poloclub.github.io/ganlab/
11. Generative Adversarial Network. Retrieved from: https://slideplayer.com/slide/15353689/
12. Introduction to Gradients and Automatic Differentiation. Retrieved from: https://www.tensorflow.org/guide/autodiff
13. Deep Learning 33: Conditional Generative Adversarial Network (C-GAN): Coding in Google Colab (2019). Retrieved from: https://www.youtube.com/watch?v=7Tlk3Gql-Wg
14. How to Develop a Conditional GAN (cGAN) From Scratch (2019). Retrieved from: https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/
15. How to Use the Keras Functional API for Deep Learning (2020). Retrieved from: https://machinelearningmastery.com/keras-functional-api-deep-learning/

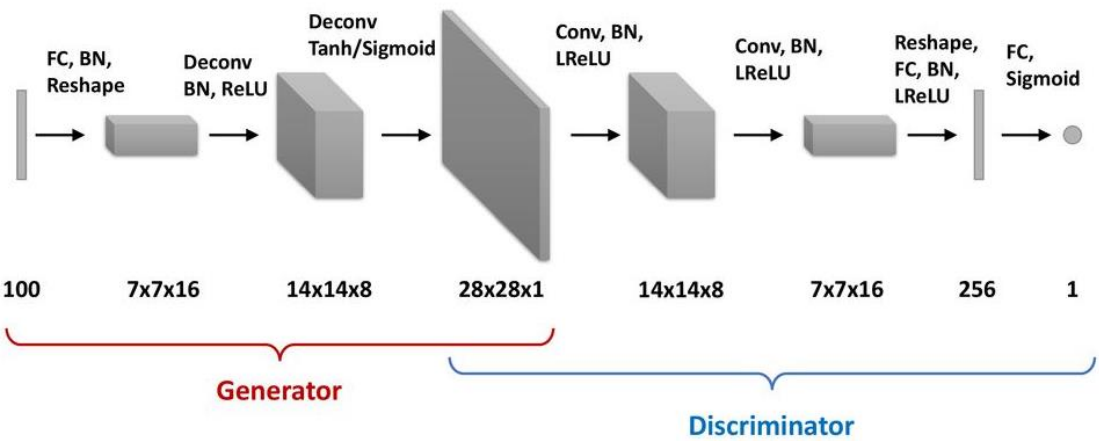
APPENDIX

1 *Appendix 1: Discriminator and generator loss functions*

```
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
```

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

2 *Appendix 2: Full compilation of GAN technical layers*



3 *Appendix 3: Definition of the training loop parameters*

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

APPENDIX

4 *Appendix 4: Full train function code*

```
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)
```

5 *Appendix 5: Full train_step function code*

```
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)
      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

6 *Appendix 6: Full compilation of cGAN technical layers*