**A Detailed Explanation of GAN with Implementation Using Tensorflow and Keras**

Table of Contents

1. What is a GAN?
2. How does GAN work?
3. Amazing Applications of GAN
4. Implementing a Toy GAN
5. Summary

What is a GAN?

Let us take an analogy to understand this concept:

Let’s say you are a football player and unfortunately you are not good at facing *ACME United.*What would you do to overcome this? You would simply ask to watch videos of ACME United playing to get better at it. You would also observe the teams who are good at facing ACME United. You probably keep practicing and learning from your mistakes. You would repeat this step until you become the **BEST** at facing ACME United. A similar concept can be incorporated in GANs.

The more you face ACME United, the better you’ll become at facing ACME United. But wait, to win more against ACME United, you **must have the players who can win against *ACME United* more frequently.**

**Simply, for getting a powerful hero (generator), we need a more powerful opponent (discriminator)!**

Now, let us understand it technically.

Generative Adversarial Networks(GAN in short) is an advancement in the field of Machine Learning which is capable of generating new data samples including Text, Audio, Images, Videos, etc. using previously available data. GANs consist of two Artificial Neural Networks or Convolution Neural Networks models namely **Generator** and **Discriminator**which are trained against each other (and thus *Adversarial*). We’ll discuss more this in the following section.

**How does GAN Work?**

As we’ve discussed that GANs consists of two ANN or CNN models:

1. Generator Model: Used to generate new images which look like real images.
2. Discriminator Model: Used to classify images as real or fake.

Let us understand each separately.

*Note: For simplicity, we’ll consider the Image Generation application to understand the GANs. Similar concepts can be applied to other applications.*

**The Generator Model**

The Generator Model generates new images by taking a fixed size random noise as an input. Generated images are then fed to the Discriminator Model.

The main goal of the Generator is to fool the Discriminator by generating images that look like real images and thus makes it harder for the Discriminator to classify images as real or fake.

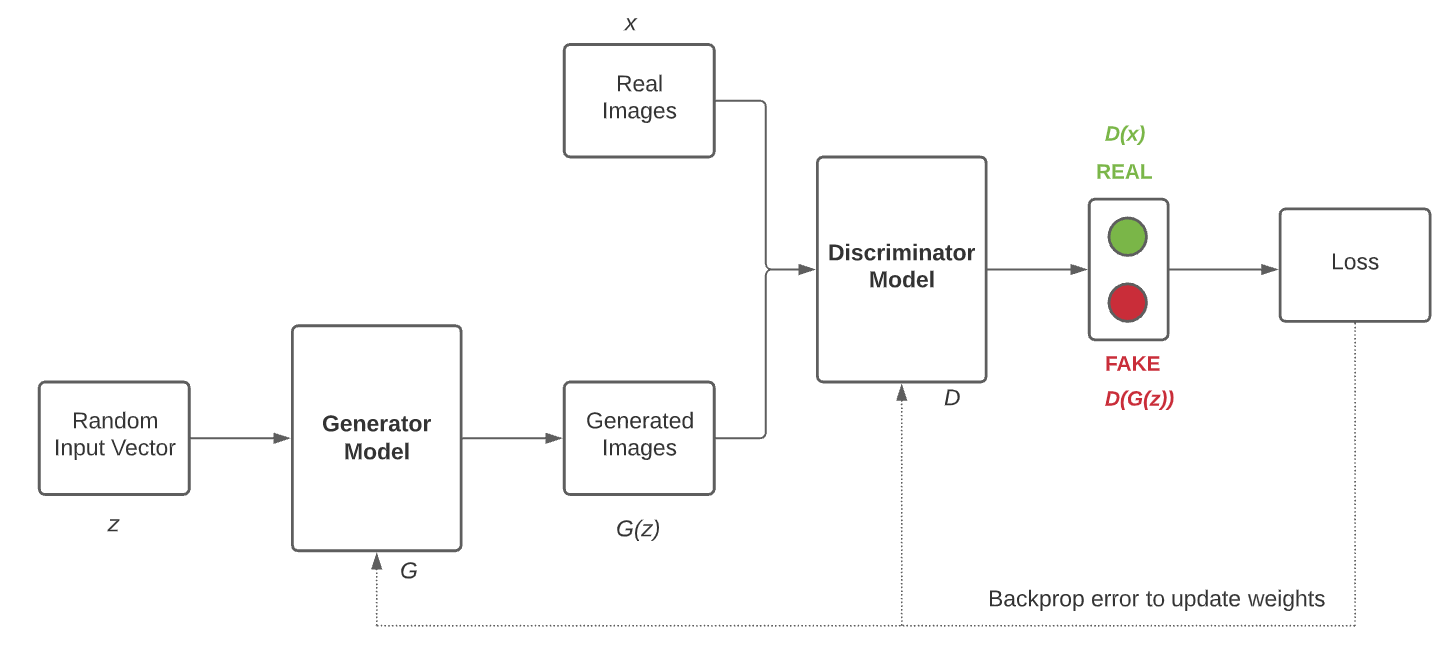
**The Discriminator Model**

The Discriminator Model takes an image as an input (generated and real) and classifies it as real or fake.

Generated images come from the Generator and the real images come from the training data.

The discriminator model is the simple binary classification model.

Now, let us combine both the architectures and understand them in detail.



The Generator Model *G* takes a random input vector *z* as an input and generates the images *G(z)*. These generated images along with the real images *x* from training data are then fed to the Discriminator Model *D*. The Discriminator Model then classifies the images as real or fake. Then, we have to measure the loss and this loss has to be back propagated to update the weights of the Generator and the Discriminator.

When we are training the Discriminator, we must freeze the Generator and back propagate errors to only update the Discriminator.

When we are training the Generator, we must freeze the Discriminator and back propagate errors to only update the Generator.

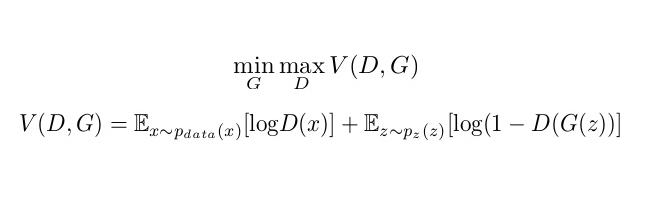
Thus, the Generator Model and the Discriminator Model getting better and better at each epoch.

We have to stop training when it attains the Nash Equilibrium or *D(x) = 0.5 for all x*. In simple words, **when the generated images look almost like real images.**

Let us introduce some notations to understand the loss function of the GANs.

|  |  |
| --- | --- |
| G | Generator Model |
| D | Discriminator Model |
| z | Random Noise (Fixed size input vector) |
| x | Real Image |
| G(z) | Image generated by Generator (Fake Image) |
| pdata(x) | Probability Distribution of Real Images |
| pz(z) | Probability Distribution of Fake Images |
| D(G(z)) | Discriminator’s output when the generated image is an input |
| D(x) | Discriminator’s output when the real image is an input |

The fight between the Generator Model and the Discriminator Model can be expressed mathematically as:



*Note: The term Ex~pdata(x)[log D(x)] can be read as****E of log(D(x)) when x is sampled from pdata(x)****and similar for the second term.*

As we can see in the equation, the Generator wants to minimize the *V(D, G)*whereas the  Discriminator wants to maximize the *V(D, G)*. Let us understand both terms:

1. Ex~pdata(x)[log D(x)]: Average log probability of D when real image is input.
2. Ez~pz(z)[log(1 – D(G(z)))]: Average log probability of D when the generated image is input.

Let us understand the equation by thinking from the Generator’s and the Discriminator’s perspectives separately.

**Discriminator’s perspective**

The Discriminator wants to maximize the loss function *V(D, G)*by correctly classifying real and fake images.

The first term suggests that the Discriminator wants to make D(x) as close to 1 as possible, i.e. correctly classifying real images as real.

The second term suggests that the Discriminator wants to make D(G(x)) as close to 0 as possible, i.e. correctly classifying fake images as fake and thus maximize the term eventually (1 – smaller number will result in a larger number). *Note: Probability lies in the range of 0-1.*

**Thus, The Discriminator tries to maximize both the terms.**

**Generator’s perspective**

The Generator wants to minimize the loss function *V(D, G)*by generating images that look like real images and tries to fool the Discriminator*.*

The second term suggests that the Generator wants to make D(G(z)) as close to 1 (instead of 0) as possible and thus minimize the term eventually (1 – larger number will result in a smaller number). So that the Discriminator fails and **misclassifies** the images.

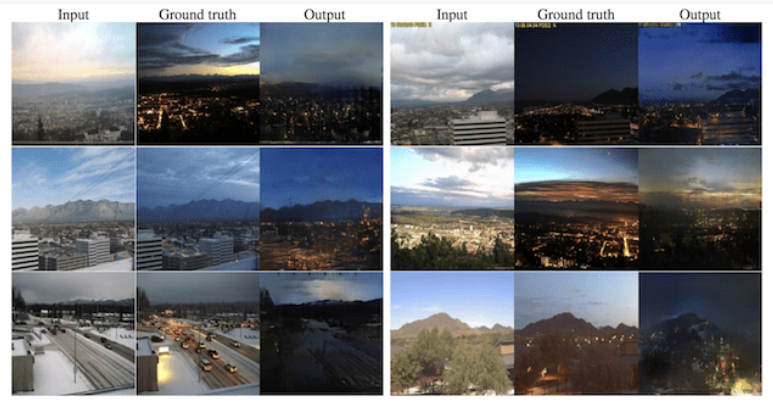
**Thus, The Generator tries to minimize the second term.**

Amazing Applications of GAN

Let us discuss some amazing applications of GANs other than image generation.

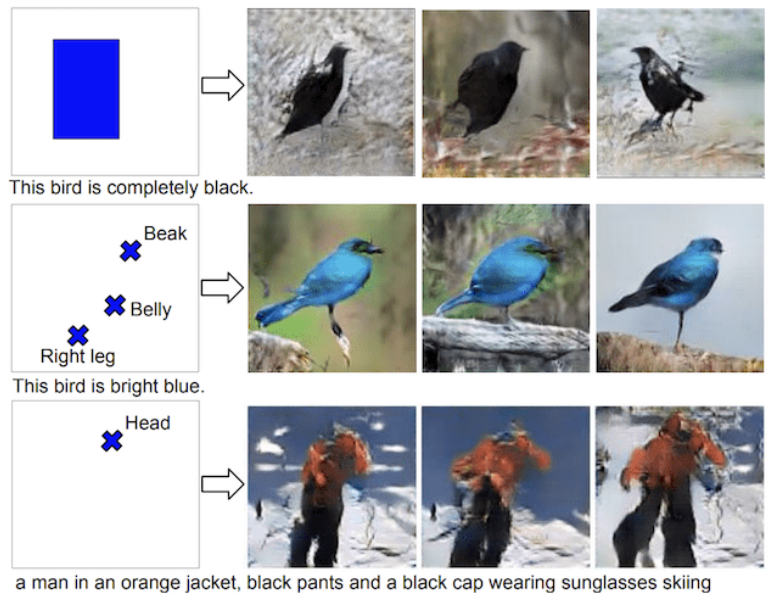
**Image to Image Translation**

 Phillip Isola, et al. in [this](https://arxiv.org/abs/1611.07004) paper demonstrates GANs as many images to image translation tasks.



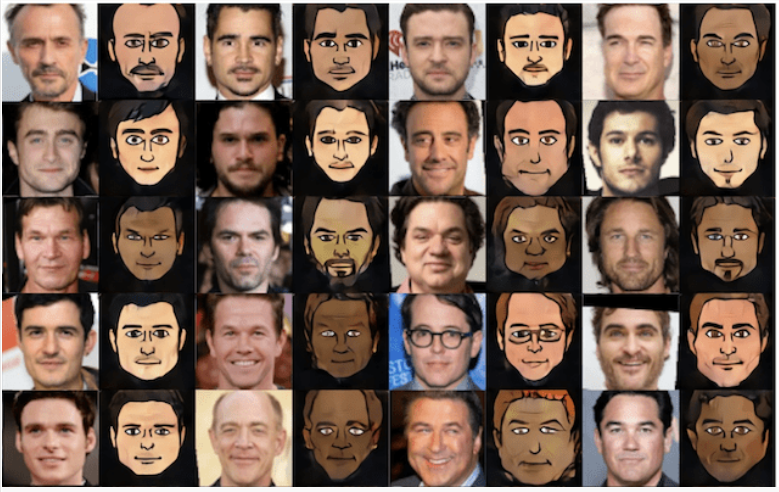
Text to Image Translation

Scott Reed, et al. in [this](https://arxiv.org/abs/1610.02454) paper, demonstrates a way to generate images from text.



Photos to Emojis

Yaniv Taigman, et al. in [this](https://arxiv.org/abs/1611.02200) paper used GANs to translate photos to emojis.



There are many more applications of GAN such as [Image Editing](https://arxiv.org/abs/1611.06355), [Face Aging](https://ieeexplore.ieee.org/document/8296650), [3D Object Generation](https://arxiv.org/abs/1610.07584), etc.

**Implementing a Toy GAN**

So, Now we’ve got a clear idea about the GANs. Let’s start implementing it using **Tensorflow** and **Keras.**

We’ll begin by Importing Necessary Libraries, considering you’ve installed all the necessary libraries already.

Importing Libraries

from numpy import zeros, ones, expand\_dims, asarray

from numpy.random import randn, randint

from keras.datasets import fashion\_mnist

from keras.optimizers import Adam

from keras.models import Model, load\_model

from keras.layers import Input, Dense, Reshape, Flatten

from keras.layers import Conv2D, Conv2DTranspose, Concatenate

from keras.layers import LeakyReLU, Dropout, Embedding

from keras.layers import BatchNormalization, Activation

from keras import initializers

from keras.initializers import RandomNormal

from keras.optimizers import Adam, RMSprop, SGD

from matplotlib import pyplot

import numpy as np

from math import sqrt

*Loading Datasets*

(X\_train, \_), (\_, \_) = fashion\_mnist.load\_data()

X\_train = X\_train.astype(np.float32) / 127.5 - 1

X\_train = np.expand\_dims(X\_train, axis=3)

print(X\_train.shape)

We are only loading the features of train data as we do not require the labels. Then we are dividing each pixel value by 127.5 and subtracting it from 1 to have pixel values in the range of -1 to 1. Finally, the X\_train shape is (60000, 28, 28, 1).

Some Necessary Functions

def generate\_latent\_points(latent\_dim, n\_samples):

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

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

return z\_input

We are using the above function to generate latent points of the shape n\_samplesxlatent\_dim(100 in our case).

def generate\_real\_samples(X\_train, n\_samples):

ix = randint(0, X\_train.shape[0], n\_samples)

X = X\_train[ix]

y = ones((n\_samples, 1))

return X, y

The above function helps us to generate *n* real samples with 1 as a label, i.e. real image.

def generate\_fake\_samples(generator, latent\_dim, n\_samples):

z\_input = generate\_latent\_points(latent\_dim, n\_samples)

images = generator.predict(z\_input)

y = zeros((n\_samples, 1))

return images, y

The above function helps us to generate *n* fake samples using the generator with 0 as a label, i.e. fake image.

def summarize\_performance(step, g\_model, latent\_dim, n\_samples=100):

X, \_ = generate\_fake\_samples(g\_model, latent\_dim, n\_samples)

X = (X + 1) / 2.0

for i in range(100):

pyplot.subplot(10, 10, 1 + i)

pyplot.axis('off')

pyplot.imshow(X[i, :, :, 0], cmap='gray\_r')

filename2 = 'model\_%04d.h5' % (step+1)

g\_model.save(filename2)

print('>Saved: %s' % (filename2))

This function helps us to summarize the performance. This includes generating a fake sample, plotting it, and finally saving the model.

def save\_plot(examples, n\_examples):

for i in range(n\_examples):

pyplot.subplot(sqrt(n\_examples), sqrt(n\_examples), 1 + i)

pyplot.axis('off')

pyplot.imshow(examples[i, :, :, 0], cmap='gray\_r')

pyplot.show()

The above function helps us to plot the results. We’ll use this to plot the generated images by the  Generator in later stages.

Model Building

def define\_discriminator(in\_shape=(28, 28, 1)):

init = RandomNormal(stddev=0.02)

in\_image = Input(shape=in\_shape)

fe = Flatten()(in\_image)

fe = Dense(1024)(fe)

fe = LeakyReLU(alpha=0.2)(fe)

fe = Dropout(0.3)(fe)

fe = Dense(512)(fe)

fe = LeakyReLU(alpha=0.2)(fe)

fe = Dropout(0.3)(fe)

fe = Dense(256)(fe)

fe = LeakyReLU(alpha=0.2)(fe)

fe = Dropout(0.3)(fe)

out = Dense(1, activation='sigmoid')(fe)

model = Model(in\_image, out)

opt = Adam(lr=0.0002, beta\_1=0.5)

model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])

return model

discriminator = define\_discriminator()

We are using couple of Dense, Flatten and Dropout layers with leaky relu as an activation function in hidden layers and sigmoid in the final layer, adam as an optimizer and binary cross-entropy as a loss function as the discriminator’s task is to perform the binary classification.

def define\_generator(latent\_dim):

init = RandomNormal(stddev=0.02)

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

gen = Dense(256, kernel\_initializer=init)(in\_lat)

gen = LeakyReLU(alpha=0.2)(gen)

gen = Dense(512, kernel\_initializer=init)(gen)

gen = LeakyReLU(alpha=0.2)(gen)

gen = Dense(1024, kernel\_initializer=init)(gen)

gen = LeakyReLU(alpha=0.2)(gen)

gen = Dense(28 \* 28 \* 1, kernel\_initializer=init)(gen)

out\_layer = Activation('tanh')(gen)

out\_layer = Reshape((28, 28, 1))(gen)

model = Model(in\_lat, out\_layer)

return model

generator = define\_generator(100)

We are using a couple of Dense layers to define the generator model with again leaky relu as an activation function in hidden layers and tanh in the final layer. The generated images *G(z)* will be of the shape 28x28x1.

def define\_gan(g\_model, d\_model):

d\_model.trainable = False

gan\_output = d\_model(g\_model.output)

model = Model(g\_model.input, gan\_output)

opt = Adam(lr=0.0002, beta\_1=0.5)

model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])

return model

gan\_model = define\_gan(generator, discriminator)

We are freezing the discriminator, providing *z* as input and *D(G(z))* as an output to our model. We are using adam as an optimizer and binary cross-entropy as a loss function.

Model Training

def train(g\_model, d\_model, gan\_model, X\_train, latent\_dim, n\_epochs=100, n\_batch=64):

bat\_per\_epo = int(X\_train.shape[0] / n\_batch)

n\_steps = bat\_per\_epo \* n\_epochs

for i in range(n\_steps):

X\_real, y\_real = generate\_real\_samples(X\_train, n\_batch)

d\_loss\_r, d\_acc\_r = d\_model.train\_on\_batch(X\_real, y\_real)

X\_fake, y\_fake = generate\_fake\_samples(g\_model, latent\_dim, n\_batch)

d\_loss\_f, d\_acc\_f = d\_model.train\_on\_batch(X\_fake, y\_fake)

z\_input = generate\_latent\_points(latent\_dim, n\_batch)

y\_gan = ones((n\_batch, 1))

g\_loss, g\_acc = gan\_model.train\_on\_batch(z\_input, y\_gan)

print('>%d, dr[%.3f,%.3f], df[%.3f,%.3f], g[%.3f,%.3f]' % (i+1, d\_loss\_r,d\_acc\_r, d\_loss\_f,d\_acc\_f, g\_loss,g\_acc))

if (i+1) % (bat\_per\_epo \* 1) == 0:

summarize\_performance(i, g\_model, latent\_dim)

This function helps us to train the generator and the discriminator. To train the Discriminator, it first generates real samples, updates the discriminator’s weights, generates fake samples, and then updates the discriminator’s weights again. To train the Generator, it first generates latent points, **generates labels as 1 to fool the discriminator,**and then updates the generator’s weights. Finally, the function summarizes the performance of the model after some steps.

latent\_dim = 100

train(generator, discriminator, gan\_model, X\_train, latent\_dim, n\_epochs=20, n\_batch=64)

We are finally calling the train function with 100 random samples, 20 epochs, and 64 as batch size.

Generating Samples Using GAN

model = load\_model('model\_18740.h5')

latent\_dim = 100

n\_examples = 100

latent\_points = generate\_latent\_points(latent\_dim, n\_examples)

X = model.predict(latent\_points)

X = (X + 1) / 2.0

save\_plot(X, n\_examples)

We are just loading the latest saved model, generating latent points, using the loaded model for prediction, and plotting the results.

Generated Images



The generated images aren’t quite clear, right? Because we haven’t used Convolution layers in our model. Try it on your own and see the results.