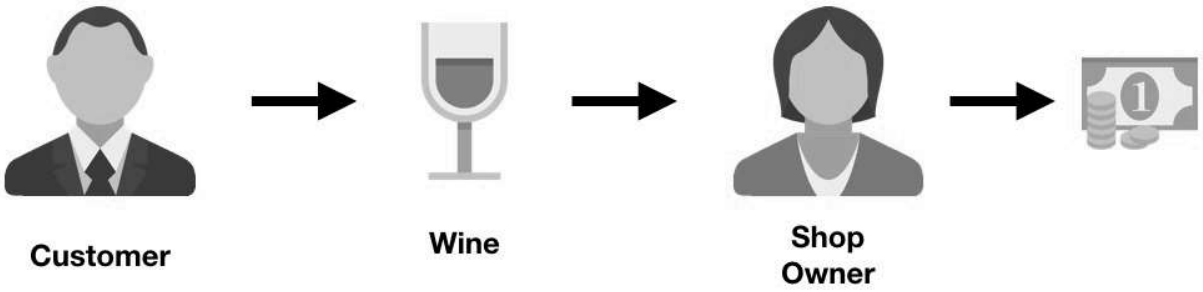
Demystifying Generative Adversarial Nets (GANs)

Analogy

The easiest way to understand what GANs are is through a simple analogy:

Suppose there is a shop which buys certain kinds of wine from customers which they will later resell.



However, there are nefarious customers who sell fake wine in order to get money. In this case, the shop owner has to be able to distinguish between the fake and authentic wines.

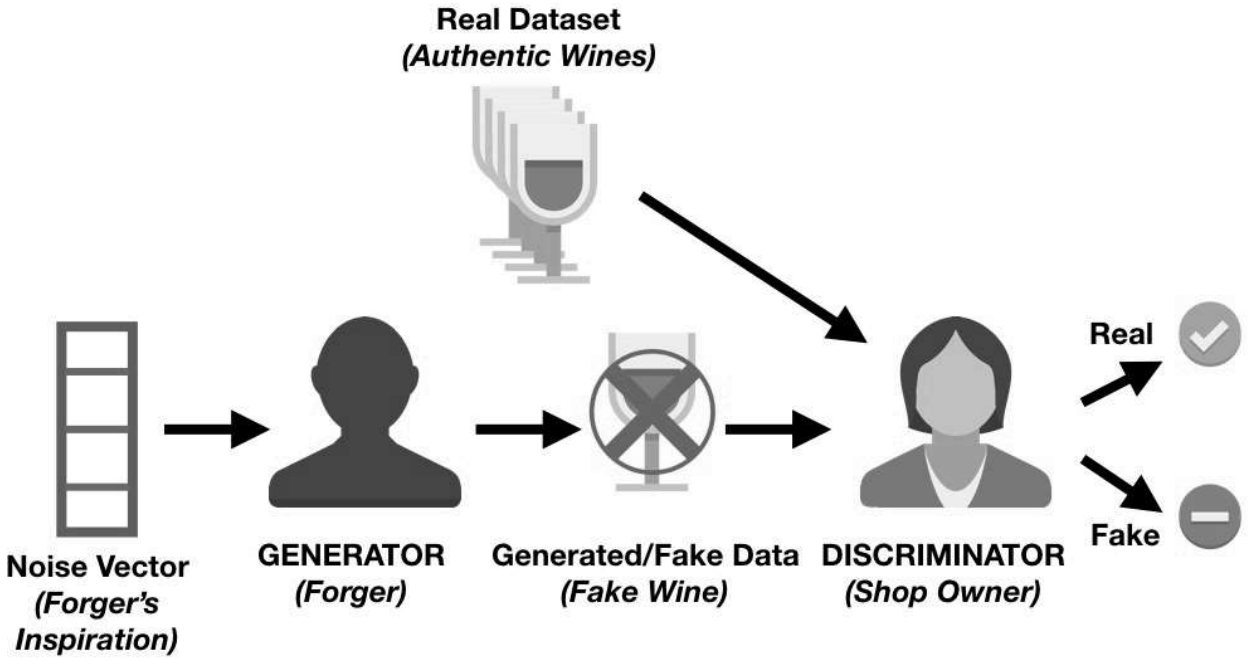
You can imagine that initially, the forger might make a lot of mistakes when trying to sell the fake wine and it will be easy for the shop owner to identify that the wine is not authentic. Because of these failures, the forger will keep on trying different techniques to simulate the authentic wines and some will eventually be successful. Now that the forger knows that certain techniques got past the shop owner's checks, he can start to further improve the fake wines based on those techniques.

At the same time, the shop owner would probably get some feedback from other shop owners or wine experts that some of the wines that she has are not original. This means that the shop owner would have to improve how she determines whether a wine is fake or authentic. The goal of the forger is to create wines that are indistinguishable from the authentic ones, and the goal of the shop owner is to accurately tell if a wine is real or not.

This back-and-forth competition is the main idea behind GANs.

Components of a Generative Adversarial Network

Using the example above, we can come up with the architecture of a GAN.



There are two major components within GANs: **the generator and the discriminator.**

The shop owner in the example is known as a discriminator network and is usually a convolutional neural network (since GANs are mainly used for image tasks) which assigns a probability that the image is real.

The forger is known as the generative network, and is also typically a convolutional neural network (with deconvolution layers). This network takes some noise vector and outputs an image. When training the generative network, it learns which areas of the image to improve/change so that the discriminator would have a harder time differentiating its generated images from the real ones.

The generative network keeps producing images that are closer in appearance to the real images while the discriminative network is trying to determine the differences between real and fake images. The ultimate goal is to have a generative network that can produce images which are indistinguishable from the real ones.

A Simple Generative Adversarial Network with Keras

Now that you understand what GANs are and the main components of them, we can now begin to code a very simple one. We will use Keras and if you are not familiar with this Python library you should read this tutorial before you continue. This tutorial is based on the GAN developed here.

The first thing you would need to do is install the following packages** via pip**

keras

matplotlib

tensorflow

tqdm

We will use matplotlib for plotting, tensorflow as the Keras backend library and tqdm to show a fancy progress bar for each epoch (iteration).

The next step is to create a Python script. In this script, we first need to import all the modules and functions we will use. An explanation of each will be given as they are used.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm


from keras.layers import Input
from keras.models import Model, Sequential
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import LeakyReLU, PReLU, ELU

from keras.datasets import mnist
from keras.optimizers import Adam
from keras import initializers
```

Code Explaination:-

This code imports necessary libraries and modules for building a deep learning model using Keras.

• os module provides a way of interacting with the operating system.

• numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

• matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

• tqdm is a progress bar library with good support for nested loops and Jupyter/IPython notebooks.

• keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.

• The code then imports specific modules from Keras, including Input, Model, Sequential, Dense, Dropout, LeakyReLU, mnist, Adam, and initializers.

• These modules are used to build and train a deep learning model for image classification on the MNIST dataset.

You now want to set some variables:

```
# Let Keras know that we are using tensorflow as our backend engine
os.environ["KERAS_BACKEND"] = "tensorflow"

# To make sure that we can reproduce the experiment and get the same results
np.random.seed(10)

# The dimension of our random noise vector.
random_dim = 100
```

This code snippet sets up some initial configurations for a machine learning model using Keras and TensorFlow as the backend engine.

• The first line sets the environment variable "**KERAS_BACKEND**" to "**tensorflow**", which tells Keras to use TensorFlow as the backend engine.

• The second line sets a random seed of **10 for NumPy**, which ensures that the random numbers generated during the training process are reproducible and consistent across different runs of the code.

• The third line sets the dimension of the random **noise vector to 100**, which will be used as input to the generator model in a Generative Adversarial Network (GAN) architecture.

Before we start building the discriminator and generator, we first gather and pre-process the data. We'll use the popular MNIST dataset which has a set of images of single digits ranging from 0 to 9.

```
def load_minst_data():
    # load the data
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # normalize our inputs to be in the range[-1, 1]
    x_train = (x_train.astype(np.float32) - 127.5)/127.5
    # convert x_train with a shape of (60000, 28, 28) to (60000, 784) so we have
    # 784 columns per row
    x_train = x_train.reshape(60000, 784)
    return (x_train, y_train, x_test, y_test)
```

Code Explaination:

This code defines a function called load_minst_data() that loads the MNIST dataset using the **mnist.load_data() function**.

• The data is split into training and testing sets, with the training set consisting of x_train and y_train, and the testing set consisting of x_test and y_test.

• The input data is then normalized to be in the range of [-1, 1] by subtracting 127.5 from each pixel value and dividing by 127.5.

• This is done to ensure that the input data has a mean of 0 and a standard deviation of 1, which can help improve the performance of machine learning models.

• Finally, the x_train data is reshaped from a 3D array with dimensions (60000, 28, 28) to a 2D array with dimensions (60000, 784) so that each row represents a single image with 784 columns (28 x 28 pixels).

• The function then returns a tuple containing the normalized training and testing data, as well as their corresponding labels.

• Note that this code uses the NumPy library, which provides support for large, multi-dimensional arrays and matrices, as well as a large collection of mathematical functions to operate on these arrays.

Note that the mnist.load_data() is part of Keras and allows you to easily import the MNIST dataset into your workspace.

Now, we create our generator and discriminator networks.

We here used the Adam optimizer for both networks.

For both the generator and discriminator, we will create a neural network with three hidden layers with the activation function being the Leaky Relu.

We also add dropout layers for the discriminator to improve its robustness on unseen images.

```
# You will use the Adam optimizer
def get_optimizer():
    return Adam(lr=0.0002, beta_1=0.5)

def get_generator(optimizer):
    generator = Sequential()
    generator.add(Dense(256, input_dim=random_dim, kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(784, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return generator

def get_discriminator(optimizer):
    discriminator = Sequential()
    discriminator.add(Dense(1024, input_dim=784, kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(256))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(1, activation='sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return discriminator
```

Code Explaination:-

This code defines three functions: **get_optimizer(), get_generator(), and get_discriminator().**

• The get_optimizer() function returns an instance of the Adam optimizer with a learning rate of 0.0002 and a beta_1 value of 0.5.

• The get_generator() function creates a Sequential model for a generator neural network.

• It has four dense layers with 256, 512, 1024, and 784 neurons respectively.

• The input dimension of the first layer is a variable called random_dim, which is not defined in this code snippet.

• The kernel initializer for the first layer is a random normal distribution with a standard deviation of 0.02.

• Each dense layer is followed by a LeakyReLU activation function with a slope of 0.2.

• The last dense layer has a tanh activation function.

• The generator is compiled with binary cross-entropy loss and the optimizer passed as an argument.

• The get_discriminator() function creates a Sequential model for a discriminator neural network.

• It has four dense layers with 1024, 512, 256, and 1 neurons respectively.

• The input dimension of the first layer is 784, which is the same as the output dimension of the generator's last layer.

• The kernel initializer for the first layer is a random normal distribution with a standard deviation of 0.02.

• Each dense layer is followed by a LeakyReLU activation function with a slope of 0.2 and a dropout layer with a rate of 0.3.

• The last dense layer has a sigmoid activation function.

• The discriminator is compiled with binary cross-entropy loss and the optimizer passed as an argument.

It is finally time to bring the generator and discriminator together!

```
def get_gan_network(discriminator, random_dim, generator, optimizer):
    # We initially set trainable to False since we only want to train either the
    # generator or discriminator at a time
    discriminator.trainable = False
    # gan input (noise) will be 100-dimensional vectors
    gan_input = Input(shape=(random_dim,))
    # the output of the generator (an image)
    x = generator(gan_input)
    # get the output of the discriminator (probability if the image is real or not)
    gan_output = discriminator(x)
    gan = Model(inputs=gan_input, outputs=gan_output)
    gan.compile(loss='binary_crossentropy', optimizer=optimizer)
    return gan
```

Code Explaination:-

This code defines a function called get_gan_network that creates a **Generative Adversarial Network (GAN) model**.

The function takes four arguments: discriminator: a pre-trained discriminator model

random_dim: an integer representing the dimensionality of the input noise vector

generator: a pre-trained generator model

optimizer: the optimizer used to train the GANThe function first sets the trainable attribute of the discriminator model to **False,** since we only want to train either the generator or discriminator at a time.

• Next, the function creates an input layer for the GAN model with the specified **random_dim shape**.

• It then passes this input through the generator model to produce an output image.

• The output image is then passed through the discriminator model to get the probability of whether the image is real or fake.

• This output is set as the output of the GAN model.

• Finally, the GAN model is compiled with a binary cross-entropy loss function and the specified optimizer.

• The function returns the compiled GAN model.

For completeness, you can create a function which will save your generated images every 20 epochs. Since this is not at the core of this lesson, you do not need to fully understand the function.

```
# Create a wall of generated MNIST images
def plot_generated_images(epoch, generator, examples=100, dim=(10, 10), figsize=(10, 10)):
    noise = np.random.normal(0, 1, size=[examples, random_dim])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(examples, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('gan_generated_image_epoch_%d.png' % epoch)
```

Code Explaination:-

This code defines a function called plot_generated_images that takes in four arguments: **epoch, generator, examples, dim, and figsize.**

• The epoch argument is an integer that represents the current epoch of the GAN training process.

• The generator argument is a Keras model that generates images.

The examples argument is an integer that represents the number of images to generate and plot.

• The dim argument is a tuple that represents the dimensions of the plot grid.

• The figsize argument is a tuple that represents the size of the plot.

• Inside the function, the **np.random.normal function** is used to generate random noise with a mean of 0 and standard deviation of 1. This noise is then passed to the generator model to generate examples number of images.

• The generated images are then reshaped to have a shape of (examples, 28, 28).

• The **plt.figure function** is used to create a new figure with the specified figsize.

• Then, a loop is used to plot each generated image on a subplot of the plot grid.

• The **plt.imshow function** is used to display the image, with the interpolation argument set to 'nearest' and the cmap argument set to 'gray_r' to display the image in grayscale.

• The **plt.axis function** is used to turn off the axis labels.

• Finally, the **plt.tight_layout function** is used to adjust the spacing between the subplots, and the plt.savefig function is used to save the plot as a PNG file with a filename that includes the current epoch number.

You have now coded the majority of your network. All that remains is to train this network and take a look at the images that you created.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

from keras.layers import Input
from keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Dropout, LeakyReLU
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import legacy as legacy_optimizers
from tensorflow.keras import initializers

# Let Keras know that we are using tensorflow as our backend engine
os.environ["KERAS_BACKEND"] = "tensorflow"

# To make sure that we can reproduce the experiment and get the same results
np.random.seed(10)

# The dimension of our random noise vector.
random_dim = 100

def load_mnist_data():
    # load the data
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # normalize our inputs to be in the range[-1, 1]
    x_train = (x_train.astype(np.float32) - 127.5) / 127.5
    # convert x_train with a shape of (60000, 28, 28) to (60000, 784) so we have
    # 784 columns per row
    x_train = x_train.reshape(60000, 784)
    return (x_train, y_train, x_test, y_test)

# You will use the legacy Adam optimizer
def get_optimizer():
    return legacy_optimizers.Adam(lr=0.0002, beta_1=0.5)

def get_generator(optimizer):
    generator = Sequential()
    generator.add(Dense(256, input_dim=random_dim, kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(512))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(784, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return generator

def get_discriminator(optimizer):
    discriminator = Sequential()
    discriminator.add(Dense(1024, input_dim=784, kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(256))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))

    discriminator.add(Dense(1, activation='sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return discriminator

def get_gan_network(discriminator, random_dim, generator, optimizer):
```

```
        # We initially set trainable to False since we only want to train either the
        # generator or discriminator at a time
        discriminator.trainable = False
        # gan input (noise) will be 100-dimensional vectors
        gan_input = Input(shape=(random_dim,))
        # the output of the generator (an image)
        x = generator(gan_input)
        # get the output of the discriminator (probability if the image is real or not)
        gan_output = discriminator(x)
        gan = Model(inputs=gan_input, outputs=gan_output)
        gan.compile(loss='binary_crossentropy', optimizer=optimizer)
        return gan

    # Create a wall of generated MNIST images
    def plot_generated_images(epoch, generator, examples=100, dim=(10, 10), figsize=(10, 10)):
        noise = np.random.normal(0, 1, size=[examples, random_dim])
        generated_images = generator.predict(noise)
        generated_images = generated_images.reshape(examples, 28, 28)

        plt.figure(figsize=figsize)
        for i in range(generated_images.shape[0]):
            plt.subplot(dim[0], dim[1], i+1)
            plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
            plt.axis('off')
        plt.tight_layout()
        plt.savefig('gan_generated_image_epoch_%d.png' % epoch)
        plt.close()

    def train(epochs=1, batch_size=128):
        # Get the training and testing data
        x_train, y_train, x_test, y_test = load_mnist_data()
        # Split the training data into batches of size 128
        batch_count = x_train.shape[0] // batch_size

        # Build our GAN network
        adam = get_optimizer()
        generator = get_generator(adam)
        discriminator = get_discriminator(adam)
        gan = get_gan_network(discriminator, random_dim, generator, adam)

        for e in range(1, epochs+1):
            print('-'*15, 'Epoch %d' % e, '-'*15)
            for _ in tqdm(range(batch_count)):
                # Get a random set of input noise and images
                noise = np.random.normal(0, 1, size=[batch_size, random_dim])
                image_batch = x_train[np.random.randint(0, x_train.shape[0], size=batch_size)]

                # Generate fake MNIST images
                generated_images = generator.predict(noise)
                X = np.concatenate([image_batch, generated_images])

                # Labels for generated and real data
                y_dis = np.zeros(2*batch_size)
                # One-sided label smoothing
                y_dis[:batch_size] = 0.9

                # Train discriminator
                discriminator.trainable = True
                discriminator.train_on_batch(X, y_dis)

                # Train generator
                noise = np.random.normal(0, 1, size=[batch_size, random_dim])
                y_gen = np.ones(batch_size)
                discriminator.trainable = False
                gan.train_on_batch(noise, y_gen)

            if e == 1 or e % 20 == 0:
                plot_generated_images(e, generator)

    if __name__ == '__main__':
        train(50, 128)
```