Published in Towards Data Science · Follow

Matthew Stewart, PhD Researcher · Follow
May 13, 2019 · 23 min read

GANs vs. Autoencoders: Comparison of Deep Generative Models

Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are your new best friend.

"Generative Adversarial Networks is the most interesting idea in the last 10 years in Machine Learning." — **Yann LeCun, Director of AI Research at Facebook AI**

Part 1 of this tutorial can be found here:

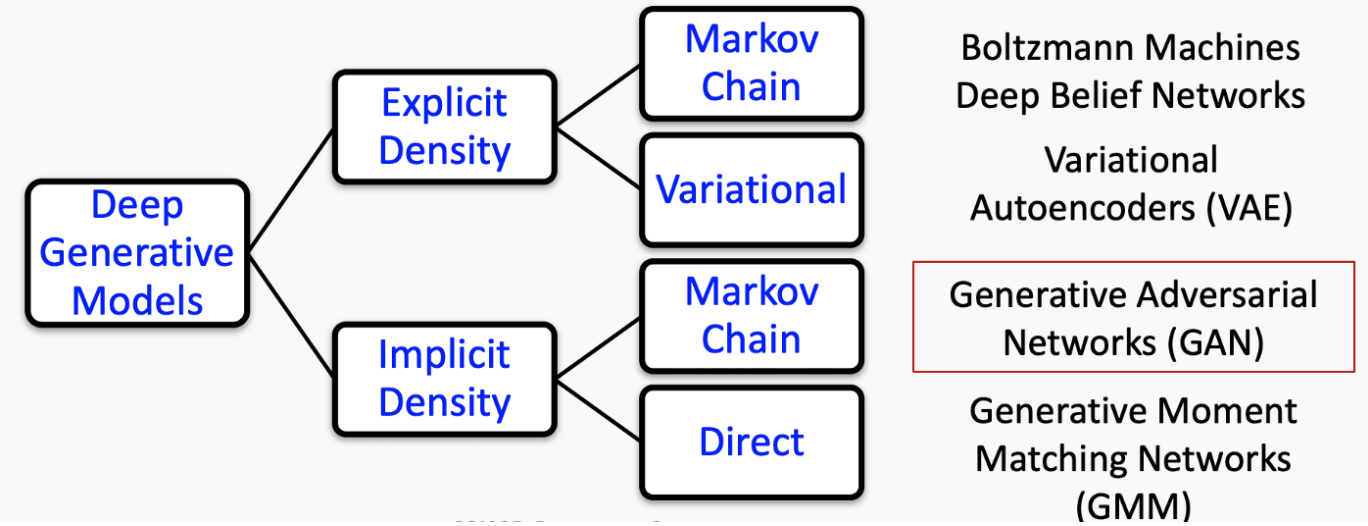
Introduction to Turing Learning and GANs
Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are…
towardsdatascience.com

Part 2of this tutorial can be found here:

Advanced Topics in GANs
Want to turn horses into zebras? Make DIY anime characters or celebrities? Generative adversarial networks (GANs) are…
towardsdatascience.com

These articles are based on lectures taken at Harvard on AC209b, with major credit going to lecturer Pavlos Protopapas of the Harvard IACS department.

This is the third part of a three-part tutorial on creating deep generative models specifically using generative adversarial networks. This is a natural extension to the previous topic on variational autoencoders (found here). We will see that GANs are typically superior as deep generative models as compared to variational autoencoders. However, they are notoriously difficult to work with and require a lot of data and tuning. We will also examine a hybrid model of GAN called a VAE-GAN.



- VAE for CelebA Dataset
- DC-GAN for CelebA Dataset
- DC-GAN for Anime Dataset
- VAE-GAN for Anime Dataset

I strongly recommend the reader to review at least part 1 of the GAN tutorial, as well as my variational autoencoder walkthrough before going further, as otherwise, the implementation may not may much sense to the reader.

All related code can now be found in my GitHub repository:

Let's begin!

VAE for CelebA Dataset

The CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations. The images in this dataset cover large pose variations and background clutter. CelebA has large diversities, large quantities, and rich annotations, including

- 10,177 number of identities,
- 202,599 number of face images, and
- 5 landmark locations, 40 binary attributes annotations per image.

You can download the dataset from Kaggle here:

The first step is to import all our necessary functions and extract the data.

Imports

```
import shutil
import errno
import zipfile
import os
import matplotlib.pyplot as plt
```

Extract Data

```
zip_ref.extractall()
zip_ref.close()
```

Custom Image Generator

This step is likely something most readers have not used before. Due to the huge size of our data, it may not be possible to load the dataset into the memory of your Jupyter Notebook. This is a pretty normal problem to have when working on large datasets.

A workaround for this is to use a stream generator, which streams batches of data (images in this case) into memory sequentially, thereby limiting the amount of memory that is required for the function. The caveat to this is that they are a bit complicated to understand and code, as they require a reasonable understanding of computer memory, GPU architecture, etc.

```
# data generator
# source from https://medium.com/@ensembledme/writing-custom-keras-generators-fe815d992c5a
from skimage.io import imread

def get_input(path):
    """get specific image from path"""
    img = imread(path)
    return img

def get_output(path, label_file = None):
    """get all the labels relative to the image of path"""
    img_id = path.split('/')[-1]
    labels = label_file.loc[img_id].values
    return labels

def preprocess_input(img):
    # convert between 0 and 1
    return img.astype('float32') / 127.5 -1

def image_generator(files, label_file, batch_size = 32):
    while True:

        batch_paths = np.random.choice(a = files, size = batch_size)
        batch_input = []
        batch_output = []

        for input_path in batch_paths:

            input = get_input(input_path)
            input = preprocess_input(input)
            output = get_output(input_path, label_file = label_file)
            batch_input += [input]
            batch_output += [output]
        batch_x = np.array(batch_input)
        batch_y = np.array(batch_output)

        yield batch_x, batch_y

def auto_encoder_generator(files, batch_size = 32):
    while True:
        batch_paths = np.random.choice(a = files, size = batch_size)
        batch_input = []
        batch_output = []

        for input_path in batch_paths:
            input = get_input(input_path)
            input = preprocess_input(input)
            output = input
            batch_input += [input]
            batch_output += [output]
        batch_x = np.array(batch_input)
        batch_y = np.array(batch_output)

        yield batch_x, batch_y
```

For more information on writing custom generators in Keras, a good article to check out is the one I referenced in the above code:

Load the Attribute Data

Not only do we have images for this dataset, but each image also has a list of attributes corresponding to aspects of the celebrity. For example, there are attributes describing whether the celebrity is wearing lipstick, or a hat, whether they are young or not, whether they have black hair, etc.

```
# now load attribute

# 1.A.2
import pandas as pd
attr = pd.read_csv('list_attr_celeba.csv')
attr = attr.set_index('image_id')

# check if attribute successful loaded
attr.describe()
```

Finish Making the Generator

Now we finish making the generator. We set the image name length to 6 since we have a 6 digit number of images in our dataset. This section of code should make sense after reading the custom Keras generator article.

```
import numpy as np
from sklearn.model_selection import train_test_split

IMG_NAME_LENGTH = 6

file_path = "img_align_celeba/"
img_id = np.arange(1,len(attr.index)+1)
img_path = []
for i in range(len(img_id)):
    img_path.append(file_path + (IMG_NAME_LENGTH - len(str(img_id[i])))*'0' + str(img_id[i]) + '.jpg')

# pick 80% as training set and 20% as validation set
train_path = img_path[:int((0.8)*len(img_path))]
val_path = img_path[int((0.8)*len(img_path)):]

train_generator = auto_encoder_generator(train_path,32)
val_generator = auto_encoder_generator(val_path,32)
```

We can now pick three images and check that attributes make sense.

```
fig, ax = plt.subplots(1, 3, figsize=(12, 4))
for i in range(3):
    ax[i].imshow(get_input(img_path[i]))
    ax[i].axis('off')
    ax[i].set_title(img_path[i][-10:])
plt.show()

attr.iloc[:3]
```



	5_o_Clock_Shadow	Arched_Eyebrows	Attractive	Bags_Under_Eyes	Bald	Bangs	Big_Lips
image_id							
000001.jpg	-1	1	1	-1	-1	-1	-1
000002.jpg	-1	-1	-1	1	-1	-1	-1
000003.jpg	-1	-1	-1	-1	-1	-1	1

Three random images along with some of their attributes.

Building and Training a VAE Model

First, we will create and compile a Convolutional VAE Model (including encoder and decoder) for the celebrity faces dataset.

More Imports

```
from keras.models import Sequential, Model
from keras.layers import Dropout, Flatten, Dense, Conv2D, MaxPooling2D, Input, Reshape, UpSampling2D,
InputLayer, Lambda, ZeroPadding2D, Cropping2D, Conv2DTranspose, BatchNormalization
from keras.utils import np_utils, to_categorical
from keras.losses import binary_crossentropy
from keras import backend as K,objectives
from keras.losses import mse, binary_crossentropy
```

Model Architecture

Now we can create and make a summary of the model.

```
b_size = 128
n_size = 512
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape = (n_size,) , mean = 0, stddev = 1)
    return z_mean + K.exp(z_log_sigma/2) * epsilon

def build_conv_vae(input_shape, bottleneck_size, sampling, batch_size = 32):

    # ENCODER
    input = Input(shape=(input_shape[0],input_shape[1],input_shape[2]))
    x = Conv2D(32,(3,3),activation = 'relu', padding = 'same')(input)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2,2), padding ='same')(x)
    x = Conv2D(64,(3,3),activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2,2), padding ='same')(x)
    x = Conv2D(128,(3,3), activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2,2), padding ='same')(x)
```

```
    shape = K.int_shape(x)
    flatten_1 = Flatten()(x)
    dense_1 = Dense(bottleneck_size, name='z_mean')(flatten_1)
    z_mean = BatchNormalization()(dense_1)
    flatten_2 = Flatten()(x)
    dense_2 = Dense(bottleneck_size, name ='z_log_sigma')(flatten_2)
    z_log_sigma = BatchNormalization()(dense_2)
    z = Lambda(sampling)([z_mean, z_log_sigma])
    encoder = Model(input, [z_mean, z_log_sigma, z], name = 'encoder')

    # DECODER
    latent_input = Input(shape=(bottleneck_size,), name = 'decoder_input')
    x = Dense(shape[1]*shape[2]*shape[3])(latent_input)
    x = Reshape((shape[1],shape[2],shape[3]))(x)
    x = UpSampling2D((2,2))(x)
    x = Cropping2D([[0,0],[0,1]])(x)
    x = Conv2DTranspose(256,(3,3), activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    x = UpSampling2D((2,2))(x)
    x = Cropping2D([[0,1],[0,1]])(x)
    x = Conv2DTranspose(128,(3,3), activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    x = UpSampling2D((2,2))(x)
    x = Cropping2D([[0,1],[0,1]])(x)
    x = Conv2DTranspose(64,(3,3), activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    x = UpSampling2D((2,2))(x)
    x = Conv2DTranspose(32,(3,3), activation = 'relu', padding = 'same')(x)
    x = BatchNormalization()(x)
    output = Conv2DTranspose(3,(3,3), activation = 'tanh', padding ='same')(x)
    decoder = Model(latent_input, output, name = 'decoder')

    output_2 = decoder(encoder(input)[2])
    vae = Model(input, output_2, name ='vae')
    return vae, encoder, decoder, z_mean, z_log_sigma

vae_2, encoder, decoder, z_mean, z_log_sigma = build_conv_vae(img_sample.shape, n_size, sampling,
batch_size = b_size)
print("encoder summary:")
encoder.summary()
print("decoder summary:")
decoder.summary()
print("vae summary:")
vae_2.summary()
```

Define the VAE Loss

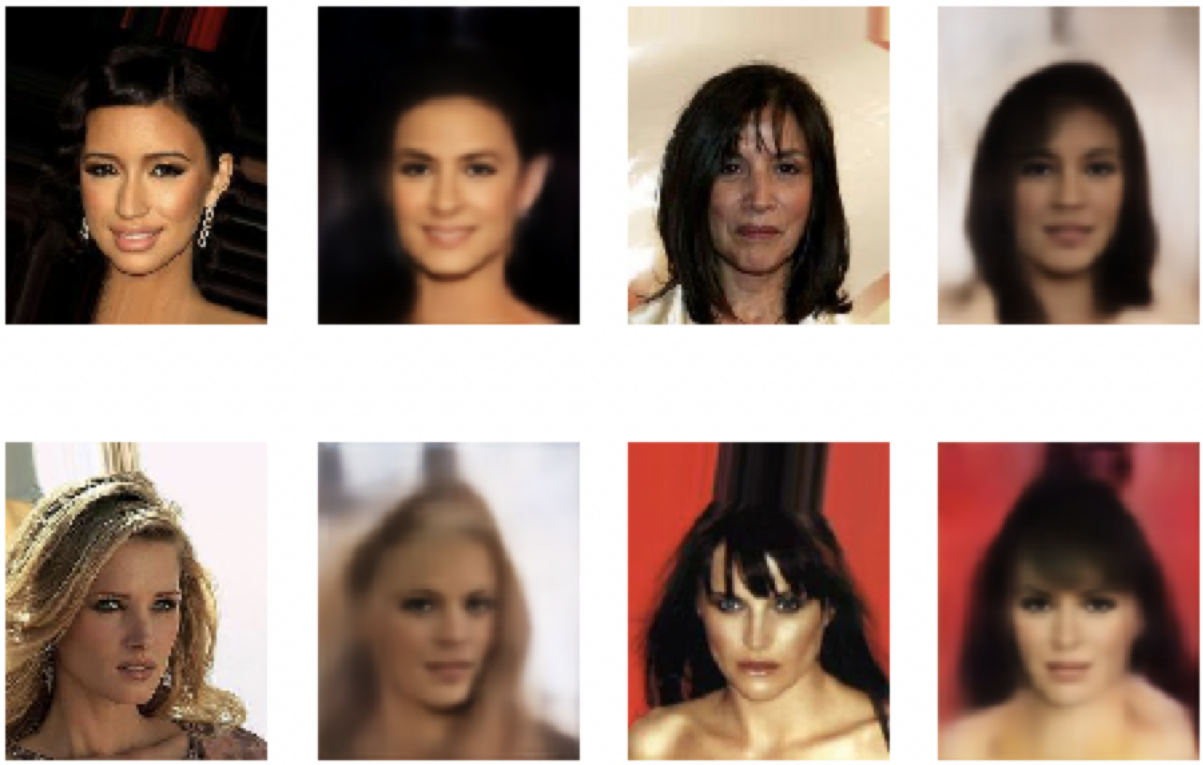
```
def vae_loss(input_img, output):
    # Compute error in reconstruction
    reconstruction_loss = mse(K.flatten(input_img) , K.flatten(output))

    # Compute the KL Divergence regularization term
    kl_loss = - 0.5 * K.sum(1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma), axis = -1)

    # Return the average loss over all images in batch
    total_loss = (reconstruction_loss + 0.0001 * kl_loss)
    return total_loss
```

Compile the Model

```
vae_2.compile(optimizer='rmsprop', loss= vae_loss)
encoder.compile(optimizer = 'rmsprop', loss = vae_loss)
decoder.compile(optimizer = 'rmsprop', loss = vae_loss)
```

Train the Model

```
import random
x_test = []
for i in range(64):
    x_test.append(get_input(img_path[random.randint(0,len(img_id))]))
x_test = np.array(x_test)
figure_Decoded = vae_2.predict(x_test.astype('float32')/127.5 -1, batch_size = b_size)
figure_original = x_test[0]
figure_decoded = (figure_Decoded[0]+1)/2
for i in range(4):
    plt.axis('off')
    plt.subplot(2,4,1+i*2)
    plt.imshow(x_test[i])
    plt.axis('off')
    plt.subplot(2,4,2 + i*2)
    plt.imshow((figure_Decoded[i]+1)/2)
    plt.axis('off')
plt.show()
```



Random samples from training set compared to their VAE reconstruction.

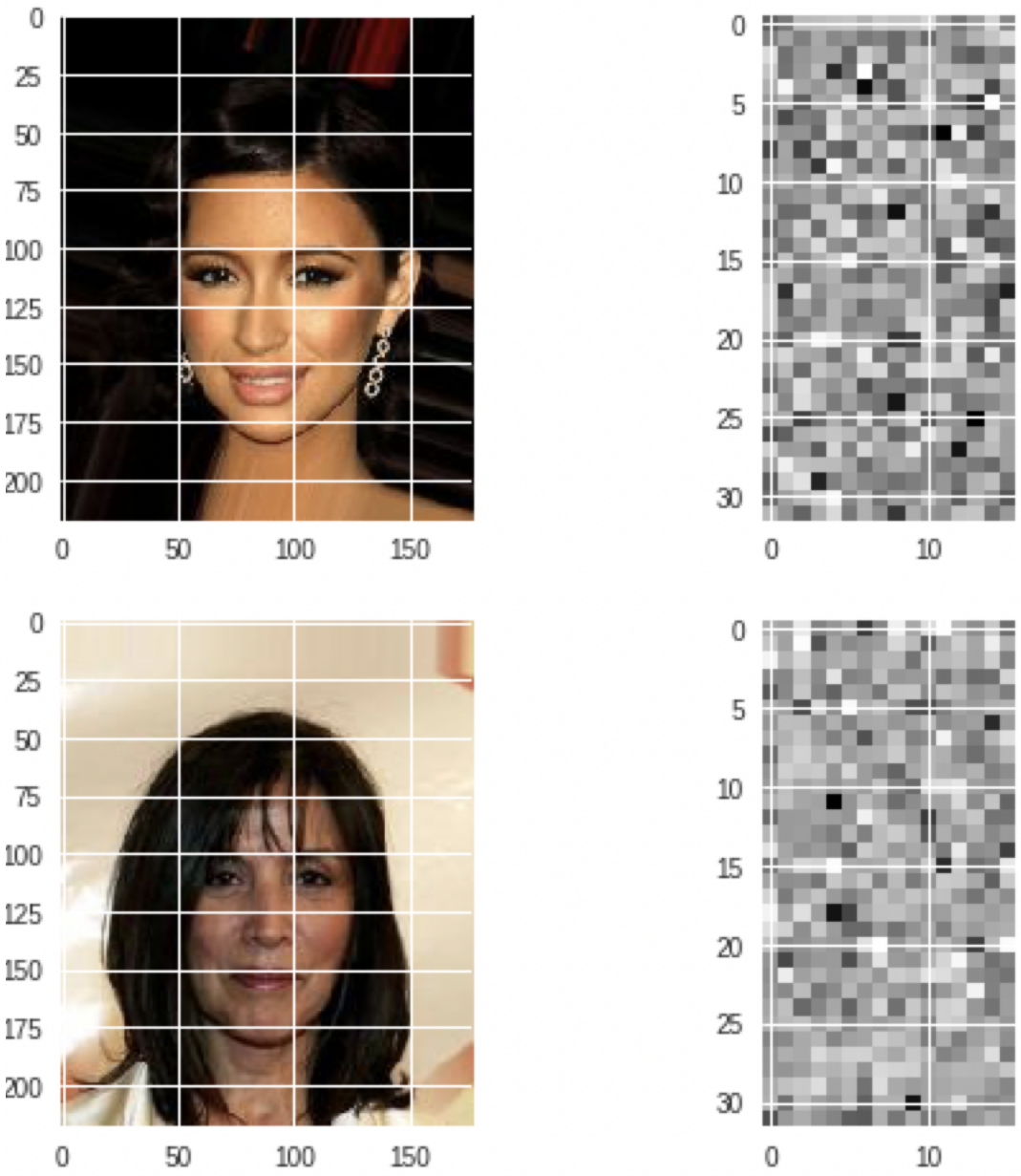
Notice that the reconstructed images share similarities with the original versions. However, the new images are a bit blurry, which is a known phenomenon of VAEs. This has been hypothesized to be due to the fact that variational inference optimizes a lower bound to the likelihood, not the actual likelihood itself.

Latent Space Representation

We can choose two images with different attributes and plot their latent space representations. Notice that we can see some differences between the latent codes, which we might hypothesize as explaining the differences between the original images.

```
# Choose two images of different attributes, and plot the original and latent space of it

x_test1 = []
for i in range(64):
    x_test1.append(get_input(img_path[np.random.randint(0,len(img_id))]))
x_test1 = np.array(x_test)
x_test_encoded = np.array(encoder_predict(x_test1/127.5-1, batch_size = b_size))
```

```
plt.figure(figsize=(8, 8))
plt.subplot(2,2,1)
plt.imshow(figure_original_1)
plt.subplot(2,2,2)
plt.imshow(Encoded1)
plt.subplot(2,2,3)
plt.imshow(figure_original_2)
plt.subplot(2,2,4)
plt.imshow(Encoded2)
plt.show()
```



Sampling from Latent Space

We can randomly sample 15 latent codes and decode them to generate new celebrity faces. We can see from this representation that the images generated by our model is of great similar styles with those images in our training set and it is also of good reality and variations.

```
# We randomly generated 15 images from 15 series of noise information

n = 3
```

```
    for j in range(5):
        z_sample = np.random.rand(1,512)
        x_decoded = decoder.predict([z_sample])
        figure[i * digit_size1: (i + 1) * digit_size1,
               j * digit_size2: (j + 1) * digit_size2,:] = (x_decoded[0]+1)/2

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```

So it seems that our VAE model is not particularly good. With more time and better selection of hyperparameters and so on, we would probably have achieved a better result than this.

Now let us compare this result to a DC-GAN on the same dataset.

DC-GAN on CelebA Dataset

Since we have already set up the stream generator, there is not too much work to do to get the DC-GAN model up and running.

```
# Create and compile a DC-GAN model, and print the summary

from keras.utils import np_utils
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Activation, Flatten, LeakyReLU,\
      BatchNormalization, Conv2DTranspose, Conv2D, Reshape
from keras.layers.advanced_activations import LeakyReLU
```

```
from scipy.misc import imresize

def generator_model(latent_dim=100, leaky_alpha=0.2, init_stddev=0.02):

    g = Sequential()
    g.add(Dense(4*4*512, input_shape=(latent_dim,),
                kernel_initializer=RandomNormal(stddev=init_stddev)))
    g.add(Reshape(target_shape=(4, 4, 512)))
    g.add(BatchNormalization())
    g.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    g.add(Conv2DTranspose(256, kernel_size=5, strides=2, padding='same',
                kernel_initializer=RandomNormal(stddev=init_stddev)))
    g.add(BatchNormalization())
    g.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    g.add(Conv2DTranspose(128, kernel_size=5, strides=2, padding='same',
                kernel_initializer=RandomNormal(stddev=init_stddev)))
    g.add(BatchNormalization())
    g.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    g.add(Conv2DTranspose(3, kernel_size=4, strides=2, padding='same',
                kernel_initializer=RandomNormal(stddev=init_stddev)))
    g.add(Activation('tanh'))
    g.summary()
    #g.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0001, beta_1=0.5), metrics=['accuracy'])
    return g


def discriminator_model(leaky_alpha=0.2, init_stddev=0.02):

    d = Sequential()
    d.add(Conv2D(64, kernel_size=5, strides=2, padding='same',
               kernel_initializer=RandomNormal(stddev=init_stddev),
               input_shape=(32, 32, 3)))
    d.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    d.add(Conv2D(128, kernel_size=5, strides=2, padding='same',
               kernel_initializer=RandomNormal(stddev=init_stddev)))
    d.add(BatchNormalization())
    d.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    d.add(Conv2D(256, kernel_size=5, strides=2, padding='same',
               kernel_initializer=RandomNormal(stddev=init_stddev)))
    d.add(BatchNormalization())
    d.add(Activation(LeakyReLU(alpha=leaky_alpha)))
    d.add(Flatten())
    d.add(Dense(1, kernel_initializer=RandomNormal(stddev=init_stddev)))
    d.add(Activation('sigmoid'))
    d.summary()
    return d

def DCGAN(sample_size=100):
    # Generator
    g = generator_model(sample_size, 0.2, 0.02)

    # Discriminator
    d = discriminator_model(0.2, 0.02)
    d.compile(optimizer=Adam(lr=0.001, beta_1=0.5), loss='binary_crossentropy')
    d.trainable = False
    # GAN
    gan = Sequential([g, d])
    gan.compile(optimizer=Adam(lr=0.0001, beta_1=0.5), loss='binary_crossentropy')

    return gan, g, d
```

The above code is just for the architecture of the generator and discriminator network. Comparing this method of coding the GAN to that which I did in part 2 is a good idea, you can see this one is less clean and we did not define global parameters, so there are many places we could have potential errors.

Now we define a bunch of functions to make our life easier, these are mostly just for the preprocessing and plotting of images to help us in analyzing the network output.

```
def load_image(filename, size=(32, 32)):
    img = plt.imread(filename)
    # crop
    rows, cols = img.shape[:2]
```

```
    img = imresize(img, size)
    return img

def preprocess(x):
    return (x/255)*2-1

def deprocess(x):
    return np.uint8((x+1)/2*255)

def make_labels(size):
    return np.ones([size, 1]), np.zeros([size, 1])

def show_losses(losses):
    losses = np.array(losses)

    fig, ax = plt.subplots()
    plt.plot(losses.T[0], label='Discriminator')
    plt.plot(losses.T[1], label='Generator')
    plt.title("Validation Losses")
    plt.legend()
    plt.show()

def show_images(generated_images):
    n_images = len(generated_images)
    cols = 5
    rows = n_images//cols

    plt.figure(figsize=(8, 6))
    for i in range(n_images):
        img = deprocess(generated_images[i])
        ax = plt.subplot(rows, cols, i+1)
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])
    plt.tight_layout()
    plt.show()
```

Train the Model

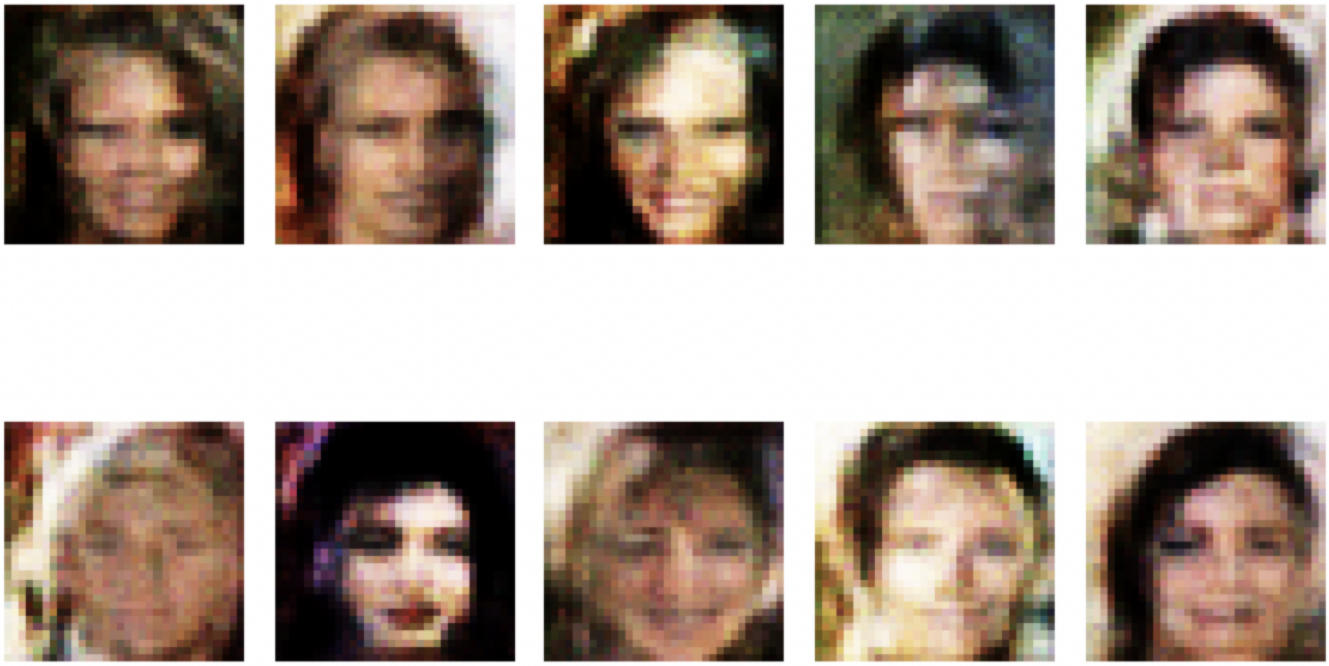
We now define the training function. As we did before, notice that we switch between setting the discriminator to be trainable and untrainable (we did this implicitly in part 2).

```
def train(sample_size=100, epochs=3, batch_size=128, eval_size=16, smooth=0.1):

    batchCount=len(train_path)//batch_size
    y_train_real, y_train_fake = make_labels(batch_size)
    y_eval_real,  y_eval_fake  = make_labels(eval_size)

    # create a GAN, a generator and a discriminator
    gan, g, d = DCGAN(sample_size)

    losses = []

    for e in range(epochs):
        print('-'*15, 'Epoch %d' % (e+1), '-'*15)
        for i in tqdm_notebook(range(batchCount)):

            path_batch = train_path[i*batch_size:(i+1)*batch_size]
            image_batch = np.array([preprocess(load_image(filename)) for filename in path_batch])

            noise = np.random.normal(0, 1, size=(batch_size, noise_dim))
            generated_images = g.predict_on_batch(noise)

            # Train discriminator on generated images
            d.trainable = True
            d.train_on_batch(image_batch, y_train_real*(1-smooth))
            d.train_on_batch(generated_images, y_train_fake)

            # Train generator
            d.trainable = False
            g_loss=gan.train_on_batch(noise, y_train_real)
```

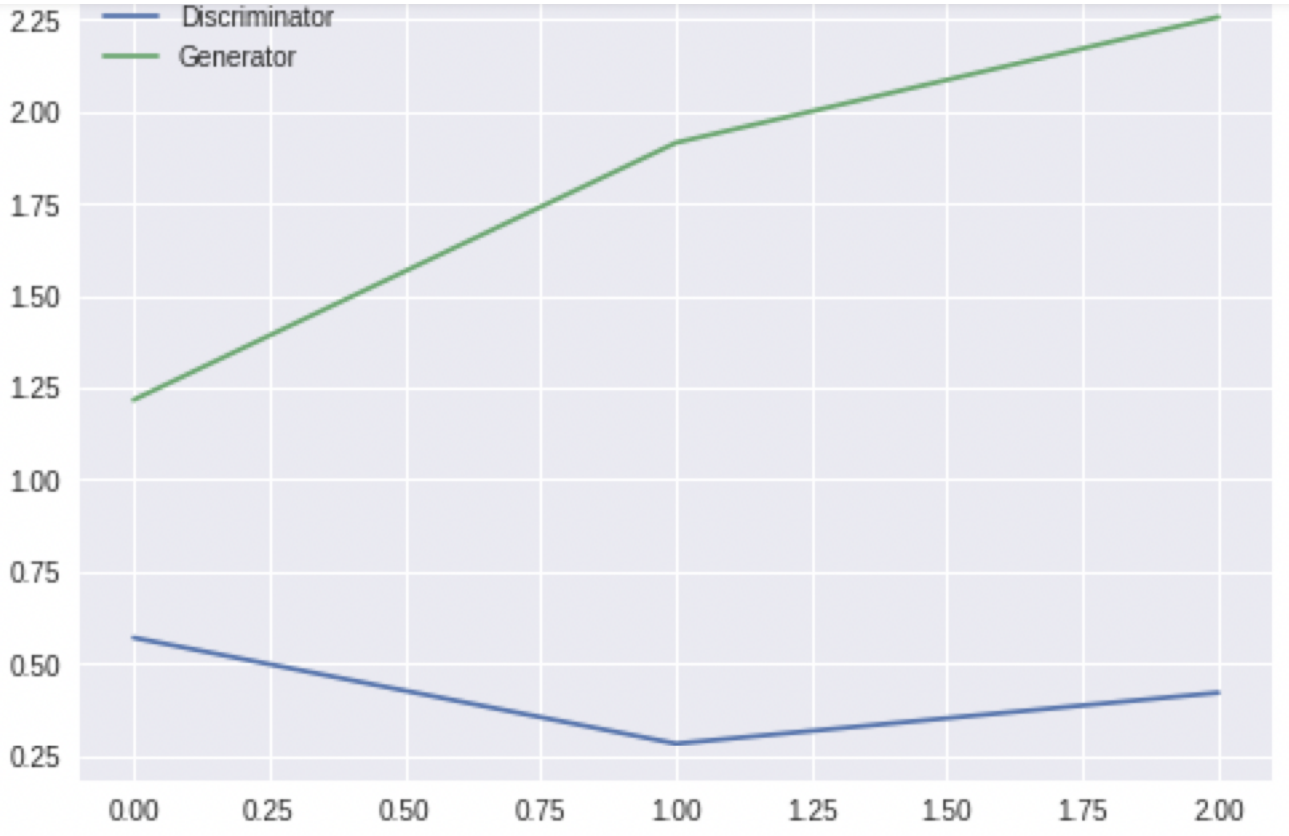
```
            noise = np.random.normal(loc=0, scale=1, size=(eval_size, sample_size))
            x_eval_fake = g.predict_on_batch(noise)

            d_loss  = d.test_on_batch(x_eval_real, y_eval_real)
            d_loss += d.test_on_batch(x_eval_fake, y_eval_fake)
            g_loss  = gan.test_on_batch(noise, y_eval_real)

            losses.append((d_loss/2, g_loss))

            print("Epoch: {:>3}/{} Discriminator Loss: {:>6.4f} Generator Loss: {:>6.4f}".format(
                e+1, epochs, d_loss, g_loss))

            show_images(x_eval_fake[:10])

        # show the result
        show_losses(losses)
        show_images(g.predict(np.random.normal(loc=0, scale=1, size=(15, sample_size))))
        return g

    noise_dim=100
    train()
```

The output of this function will give us the following output for each epoch:

```
Epoch:   1/3 Discriminator Loss: 1.1399 Generator Loss: 1.2161

<Figure size 576x432 with 10 Axes>
```



It will also plot our validation losses for the discriminator and generator.

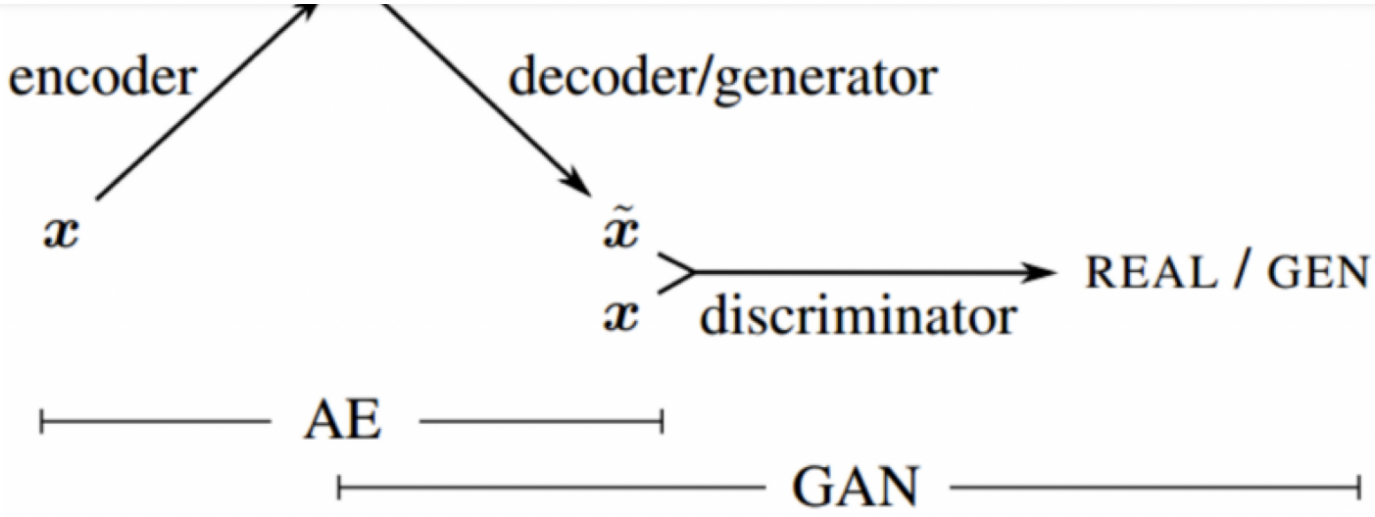


The generated images look reasonable. Here we can see that our model performed adequately, though the quality of images is not so good as those in the training set (since we reshaped the images to become smaller and made them more blurry than the original ones). However, they are vivid enough to create valid faces, and these faces are close enough to reality. Also, compared with images produced by VAE, the images are more creative and real-looking.

So it seems that the GAN performs superior in this circumstance. Now let us try a new dataset and see how well a GAN can perform compared to a hybrid variant, the VAE-GAN.

Anime Dataset

In this section, we will aim to generate faces in the same style as the Anime dataset using a GAN, as well as another special form of GAN, a VAE-GAN. The term VAE-GAN was first used by Larsen et. al in their paper "Autoencoding beyond pixels using a learned similarity metric". VAE-GAN models differentiate themselves from GANs in that their **generators are variation autoencoders**.



VAE-GAN architecture. Source: https://arxiv.org/abs/1512.09300

First, we will focus on the DC-GAN. The Anime dataset consists of over 20K anime faces in the form of 64x64 images. We will also need to create another Keras Custom Data Generator. A link to the dataset can be found here:

Mckinsey666/Anime-Face-Dataset

🖼 A collection of high-quality anime faces. Contribute to Mckinsey666/Anime-Face-Dataset development by creating an...

github.com

DC-GAN on Anime Dataset

The first thing we need to do is create anime directory and download the data. This can be done from the link above. It is always good practice to check the data before moving ahead, so we do this now.

```
from skimage import io
import matplotlib.pyplot as plt

filePath='anime-faces/data/'
imgSets=[]

for i in range(1,20001):
    imgName=filePath+str(i)+'.png'
    imgSets.append(io.imread(imgName))

plt.imshow(imgSets[1234])
plt.axis('off')
plt.show()
```

We now create and compile our DC-GAN model.

```
# Create and compile a DC-GAN model

from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Activation, \
    Flatten, LeakyReLU, BatchNormalization, Conv2DTranspose, Conv2D, Reshape
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D
from keras.optimizers import Adam, RMSprop,SGD
from keras.initializers import RandomNormal
```

```
from PIL import Image
from tqdm import tqdm_notebook

image_shape = (64, 64, 3)
#noise_shape = (100,)
Noise_dim = 128
img_rows = 64
img_cols = 64
channels = 3

def generator_model(latent_dim=100, leaky_alpha=0.2):
    model = Sequential()

    # layer1 (None,500)>>(None,128*16*16)
    model.add(Dense(128 * 16 * 16, activation="relu", input_shape=(Noise_dim,)))

    # (None,16*16*128)>>(None,16,16,128)
    model.add(Reshape((16, 16, 128)))

   # (None,16,16,128)>>(None,32,32,128)
    model.add(UpSampling2D())
    model.add(Conv2D(256, kernel_size=3, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    #(None,32,32,128)>>(None,64,64,128)
    model.add(UpSampling2D())

    # (None,64,64,128)>>(None,64,64,64)
    model.add(Conv2D(128, kernel_size=3, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    # (None,64,64,128)>>(None,64,64,32)

    model.add(Conv2D(32, kernel_size=3, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    # (None,64,64,32)>>(None,64,64,3)
    model.add(Conv2D(channels, kernel_size=3, padding="same"))
    model.add(Activation("tanh"))

    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0001, beta_1=0.5), metrics=['accuracy'])
    return model

def discriminator_model(leaky_alpha=0.2, dropRate=0.3):
    model = Sequential()

    # layer1 (None,64,64,3)>>(None,32,32,32)
    model.add(Conv2D(32, kernel_size=3, strides=2, input_shape=image_shape, padding="same"))
    model.add(LeakyReLU(alpha=leaky_alpha))
    model.add(Dropout(dropRate))

    # layer2 (None,32,32,32)>>(None,16,16,64)
    model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))

    # model.add(ZeroPadding2D(padding=((0, 1), (0, 1))))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=leaky_alpha))
    model.add(Dropout(dropRate))

    # (None,16,16,64)>>(None,8,8,128)
    model.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(dropRate))

    # (None,8,8,128)>>(None,8,8,256)
    model.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(dropRate))

     # (None,8,8,256)>>(None,8,8,64)
```

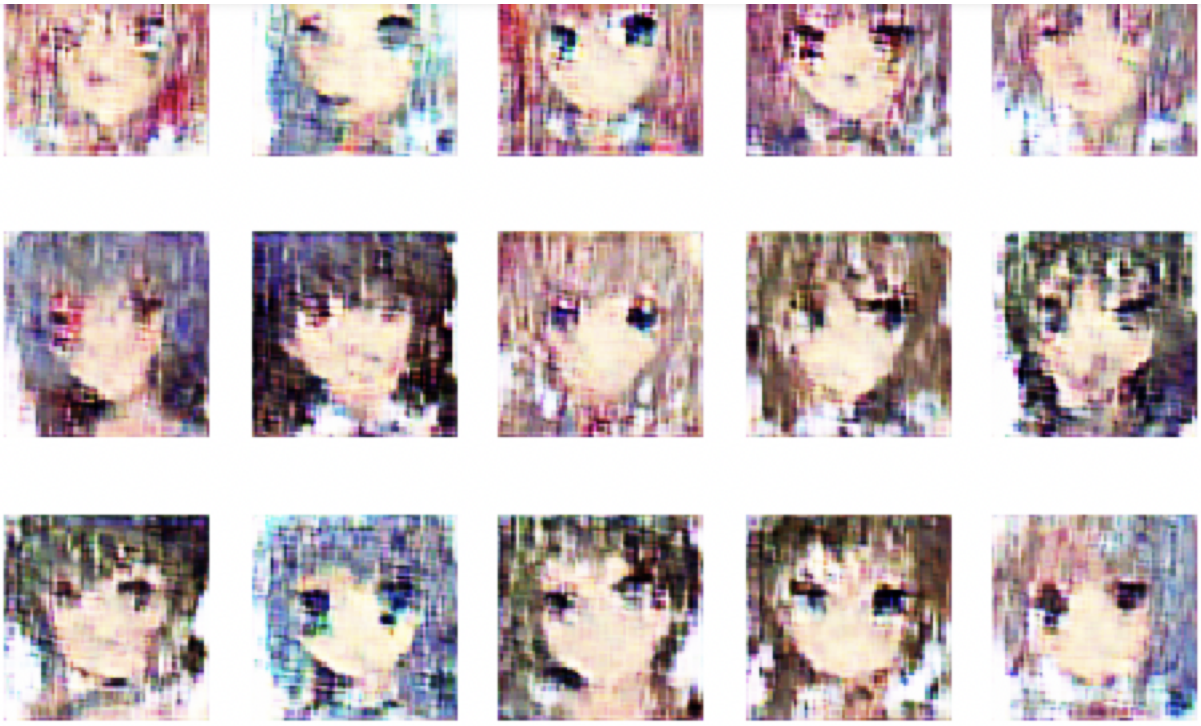
```
    # (None,8,8,64)
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    model.summary()

    sgd=SGD(lr=0.0002)
    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0001, beta_1=0.5), metrics=['accuracy'])
    return model

def DCGAN(sample_size=Noise_dim):
    # generator
    g = generator_model(sample_size, 0.2)

    # discriminator
    d = discriminator_model(0.2)
    d.trainable = False
    # GAN
    gan = Sequential([g, d])

    sgd=SGD()
    gan.compile(optimizer=Adam(lr=0.0001, beta_1=0.5), loss='binary_crossentropy')
    return gan, g, d

def get_image(image_path, width, height, mode):
    image = Image.open(image_path)
    #print(image.size)

    return np.array(image.convert(mode))

def get_batch(image_files, width, height, mode):
    data_batch = np.array([get_image(sample_file, width, height, mode) \
                           for sample_file in image_files])
    return data_batch

def show_imgs(generator,epoch):
    row=3
    col = 5
    noise = np.random.normal(0, 1, (row * col, Noise_dim))
    gen_imgs = generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(row, col)
    #fig.suptitle("DCGAN: Generated digits", fontsize=12)
    cnt = 0

    for i in range(row):
        for j in range(col):
            axs[i, j].imshow(gen_imgs[cnt, :, :, :])
            axs[i, j].axis('off')
            cnt += 1

    #plt.close()
    plt.show()
```

We can now train the model on the Anime dataset. We will do this in two different ways, the first will involve training the discriminator and generator with a 1:1 proportion of training times.

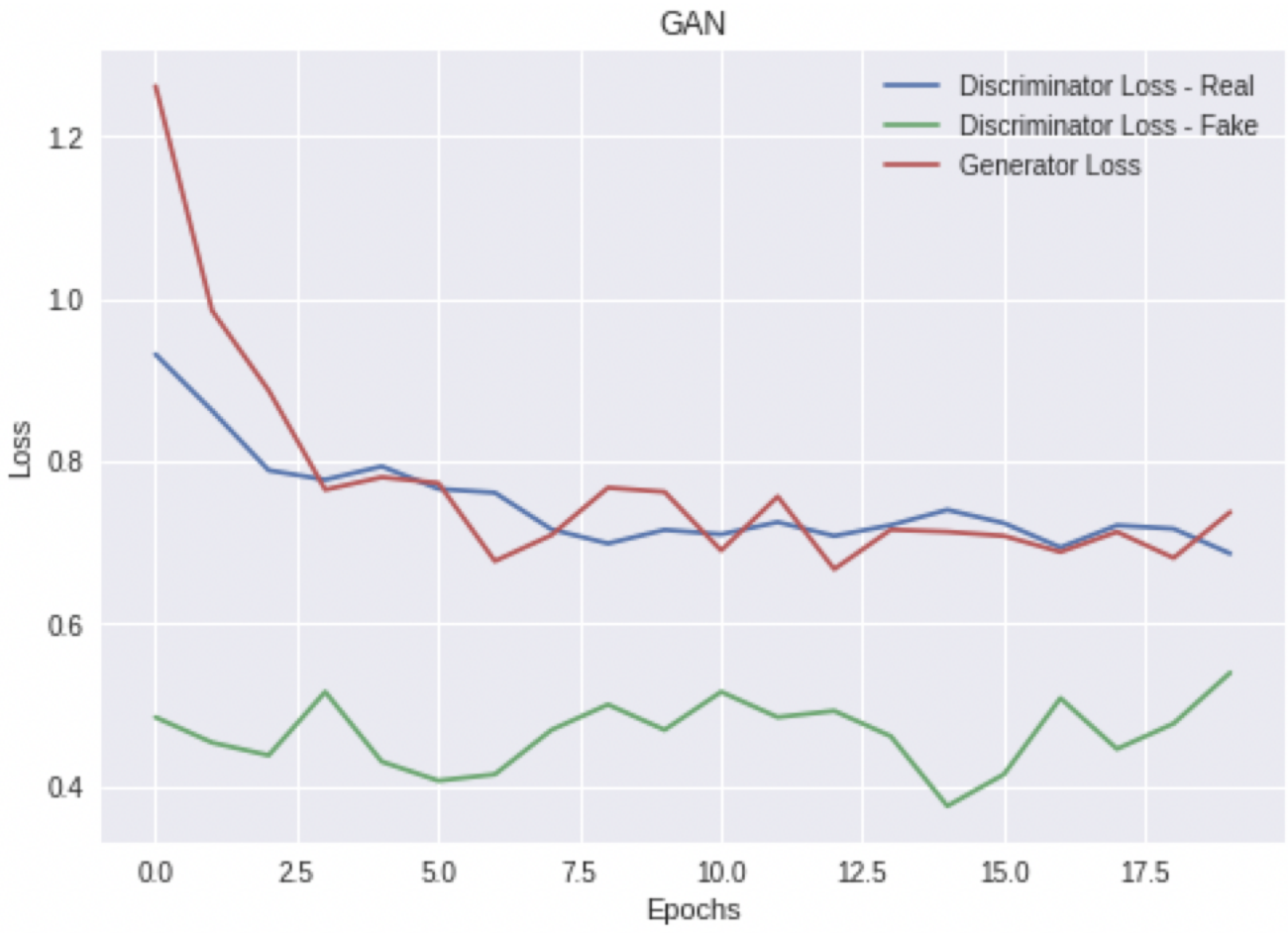
```
# Training the discriminator and generator with the 1:1 proportion of training times

def train(epochs=30, batchSize=128):
    filePath = r'anime-faces/data/'

    X_train = get_batch(glob.glob(os.path.join(filePath, '*.png'))[:20000], 64, 64, 'RGB')
    X_train = (X_train.astype(np.float32) - 127.5) / 127.5

    halfSize = int(batchSize / 2)
    batchCount=int(len(X_train)/batchSize)
```

```
        for i in tqdm_notebook(range(batchCount)):
            index = np.random.randint(0, X_train.shape[0], halfSize)
            images = X_train[index]

            noise = np.random.normal(0, 1, (halfSize, Noise_dim))
            genImages = generator.predict(noise)

            # one-sided labels
            discriminator.trainable = True
            dLossR = discriminator.train_on_batch(images, np.ones([halfSize, 1]))
            dLossF = discriminator.train_on_batch(genImages, np.zeros([halfSize, 1]))
            dLoss = np.add(dLossF, dLossR) * 0.5
            discriminator.trainable = False

            noise = np.random.normal(0, 1, (batchSize, Noise_dim))
            gLoss = gan.train_on_batch(noise, np.ones([batchSize, 1]))

        dLossReal.append([e, dLoss[0]])
        dLossFake.append([e, dLoss[1]])
        gLossLogs.append([e, gLoss])

        dLossRealArr = np.array(dLossReal)
        dLossFakeArr = np.array(dLossFake)
        gLossLogsArr = np.array(gLossLogs)


        # At the end of training plot the losses vs epochs
        show_imgs(generator, e)

    plt.plot(dLossRealArr[:, 0], dLossRealArr[:, 1], label="Discriminator Loss - Real")
    plt.plot(dLossFakeArr[:, 0], dLossFakeArr[:, 1], label="Discriminator Loss - Fake")
    plt.plot(gLossLogsArr[:, 0], gLossLogsArr[:, 1], label="Generator Loss")
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('GAN')
    plt.grid(True)
    plt.show()


    return gan, generator, discriminator

GAN,Generator,Discriminator=train(epochs=20, batchSize=128)
train(epochs=1000, batchSize=128, plotInternal=200)
```

The output will now start printing a series of anime characters. They are very grainy at first, and over time gradually become more and more pronounced.



We will also get a plot of our generator and discriminator loss functions.



Now we will do the same but with different training times for the discriminator and generator to see what the effect has been.

Before moving forward, it is good to save the weights of the model somewhere so that you do not need to run the entire training

```
discriminator.save_weights('/content/gdrive/My
Drive/discriminator_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
gan.save_weights('/content/gdrive/My Drive/gan_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
generator.save_weights('/content/gdrive/My Drive/generator_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
```

To load the weights:

```
discriminator.load_weights('/content/gdrive/My
Drive/discriminator_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
gan.load_weights('/content/gdrive/My Drive/gan_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
generator.load_weights('/content/gdrive/My Drive/generator_DCGAN_lr0.0001_deepgenerator+proportion2.h5')
```

Now we move onto the second network implementation without worrying about saving over our previous network.

```
# Train the discriminator and generator separately and with different training times

def train(epochs=300, batchSize=128, plotInternal=50):
    gLoss = 1
    filePath = r'anime-faces/data/'

    X_train = get_batch(glob.glob(os.path.join(filePath,'*.png'))[:20000],64,64,'RGB')
    X_train=(X_train.astype(np.float32)-127.5)/127.5
    halfSize= int (batchSize/2)

    dLossReal=[]
    dLossFake=[]
    gLossLogs=[]


    for e in range(epochs):
        index=np.random.randint(0,X_train.shape[0],halfSize)
        images=X_train[index]

        noise=np.random.normal(0,1,(halfSize,Noise_dim))
        genImages=generator.predict(noise)

        if e < int(epochs*0.5):
            #one-sided labels
            discriminator.trainable=True
            dLossR=discriminator.train_on_batch(images,np.ones([halfSize,1]))
            dLossF=discriminator.train_on_batch(genImages,np.zeros([halfSize,1]))
            dLoss=np.add(dLossF,dLossR)*0.5
            discriminator.trainable=False

            cnt = e

            while cnt > 3:
                cnt = cnt - 4

            if cnt == 0:
                noise=np.random.normal(0,1,(batchSize,Noise_dim))
                gLoss=gan.train_on_batch(noise,np.ones([batchSize,1]))

        elif e>= int(epochs*0.5) :
            cnt = e

            while cnt > 3:
                cnt = cnt - 4

            if cnt == 0:
                #one-sided labels
                discriminator.trainable=True
                dLossR=discriminator.train_on_batch(images,np.ones([halfSize,1]))
                dLossF=discriminator.train_on_batch(genImages,np.zeros([halfSize,1]))
                dLoss=np.add(dLossF,dLossR)*0.5
                discriminator.trainable=False
```

```
            dLossReal.append([e,dLoss[0]])
            dLossFake.append([e,dLoss[1]])
            gLossLogs.append([e,gLoss])

            if e % plotInternal == 0 and e!=0:
                show_imgs(generator, e)


            dLossRealArr= np.array(dLossReal)
            dLossFakeArr = np.array(dLossFake)
            gLossLogsArr = np.array(gLossLogs)

            chk = e

            while chk > 50:
                chk = chk - 51

            if chk == 0:
                discriminator.save_weights('/content/gdrive/My
    Drive/discriminator_DCGAN_lr=0.0001,proportion2,deepgenerator_Fake.h5')
                gan.save_weights('/content/gdrive/My
    Drive/gan_DCGAN_lr=0.0001,proportion2,deepgenerator_Fake.h5')
                generator.save_weights('/content/gdrive/My
    Drive/generator_DCGAN_lr=0.0001,proportion2,deepgenerator_Fake.h5')
            # At the end of training plot the losses vs epochs
        plt.plot(dLossRealArr[:, 0], dLossRealArr[:, 1], label="Discriminator Loss - Real")
        plt.plot(dLossFakeArr[:, 0], dLossFakeArr[:, 1], label="Discriminator Loss - Fake")
        plt.plot(gLossLogsArr[:, 0], gLossLogsArr[:, 1], label="Generator Loss")
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.title('GAN')
        plt.grid(True)
        plt.show()


        return gan, generator, discriminator

    gan, generator, discriminator = DCGAN(Noise_dim)
    train(epochs=4000, batchSize=128, plotInternal=200)
```

Let us compare the output of these two networks. By running the line:

```
show_imgs(Generator)
```

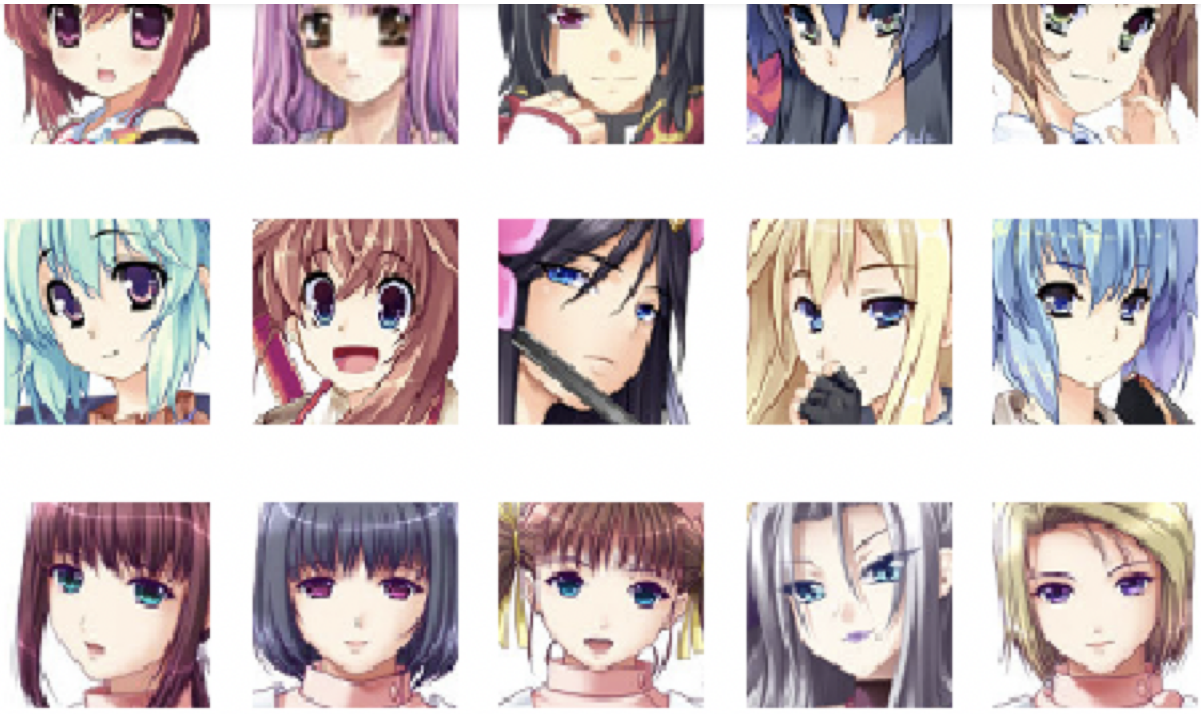
the network will output some images from the generator (this is one of the functions we defined earlier).

Generated images from 1:1 training of discriminator vs. generator.

Now let's check the second model.

Generated images from the second network with different training times for the discriminator and generator.

We can see that the details of the generated images are improved and the texture of them are slightly more detailed. However, in comparison to the training images they are still sub-par.

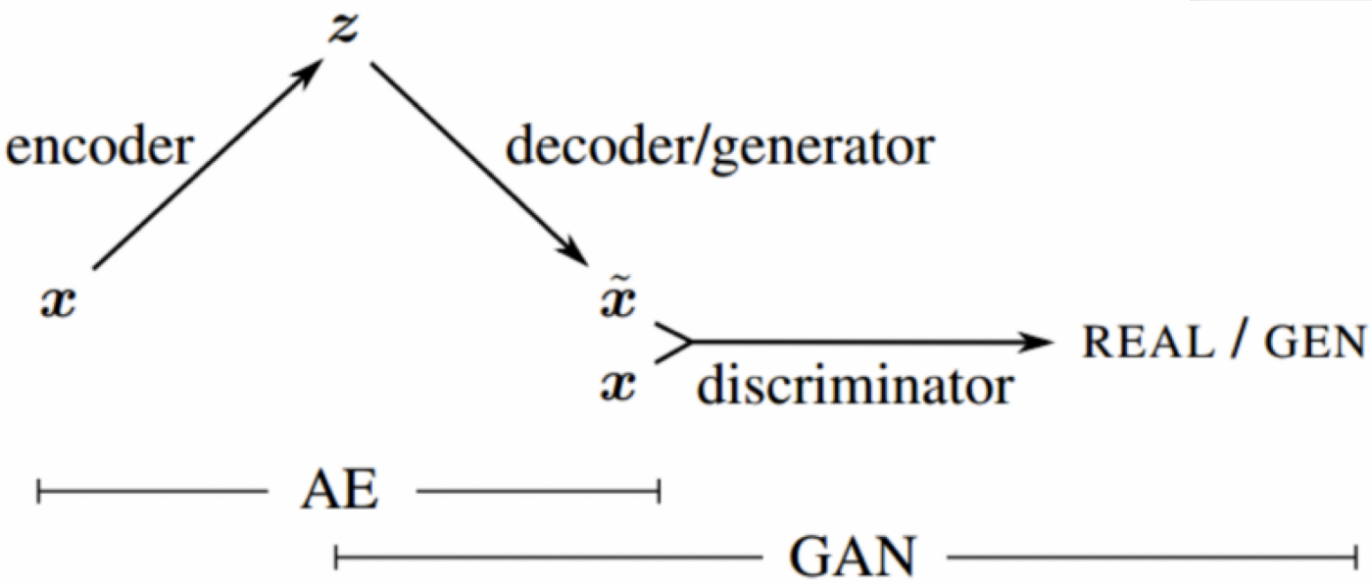


Training images from Anime dataset.

Perhaps the VAE-GAN will perform better?

VAE-GAN on Anime Dataset

To reiterate what I said previously about the VAE-GAN, the term VAE-GAN was first used by Larsen et. al in their paper "Autoencoding beyond pixels using a learned similarity metric". VAE-GAN models differentiate themselves from GANs in that their **generators are variation autoencoders**.



VAE-GAN architecture. Source: https://arxiv.org/abs/1512.09300

First we need to create and compile the VAE-GAN and make a summary for each of the networks (this is a good way to simply check the architecture).

```
Reshape,MaxPooling2D,UpSampling2D,InputLayer, Lambda
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D
from keras.optimizers import Adam, RMSprop,SGD
from keras.initializers import RandomNormal
import numpy as np
import matplotlib.pyplot as plt
import os, glob
from PIL import Image
import pandas as pd
from scipy.stats import norm
import keras
from keras.utils import np_utils, to_categorical
from keras import backend as K
import random
from keras import metrics
from tqdm import tqdm

# plotInternal
plotInternal = 50

#######
latent_dim = 256
batch_size = 256
rows = 64
columns = 64
channel = 3
epochs = 4000
# datasize = len(dataset)

# optimizers
SGDop = SGD(lr=0.0003)
ADAMop = Adam(lr=0.0002)
# filters
filter_of_dis = 16
filter_of_decgen = 16
filter_of_encoder = 16

def sampling(args):
    mean, logsigma = args
    epsilon = K.random_normal(shape=(K.shape(mean)[0], latent_dim), mean=0., stddev=1.0)
    return mean + K.exp(logsigma / 2) * epsilon

def vae_loss(X , output , E_mean, E_logsigma):
        # compute the average MSE error, then scale it up, ie. simply sum on all axes
  reconstruction_loss = 2 * metrics.mse(K.flatten(X), K.flatten(output))

        # compute the KL loss
  kl_loss = - 0.5 * K.sum(1 + E_logsigma - K.square(E_mean) - K.exp(E_logsigma), axis=-1)

  total_loss = K.mean(reconstruction_loss + kl_loss)

  return total_loss


def encoder(kernel, filter, rows, columns, channel):
    X = Input(shape=(rows, columns, channel))
    model = Conv2D(filters=filter, kernel_size=kernel, strides=2, padding='same')(X)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*2, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*4, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*8, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Flatten()(model)

    mean = Dense(latent_dim)(model)
```

```
def decgen(kernel, filter, rows, columns, channel):
    X = Input(shape=(latent_dim,))

    model = Dense(2*2*256)(X)
    model = Reshape((2, 2, 256))(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = Activation('relu')(model)

    model = Conv2DTranspose(filters=filter*8, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = Activation('relu')(model)

    model = Conv2DTranspose(filters=filter*4, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = Activation('relu')(model)

    model = Conv2DTranspose(filters=filter*2, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = Activation('relu')(model)

    model = Conv2DTranspose(filters=filter, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = Activation('relu')(model)

    model = Conv2DTranspose(filters=channel, kernel_size=kernel, strides=2, padding='same')(model)
    model = Activation('tanh')(model)

    model = Model(X, model)
    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0001, beta_1=0.5), metrics=['accuracy'])
    return model

def discriminator(kernel, filter, rows, columns, channel):
    X = Input(shape=(rows, columns, channel))

    model = Conv2D(filters=filter*2, kernel_size=kernel, strides=2, padding='same')(X)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*4, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*8, kernel_size=kernel, strides=2, padding='same')(model)
    model = BatchNormalization(epsilon=1e-5)(model)
    model = LeakyReLU(alpha=0.2)(model)

    model = Conv2D(filters=filter*8, kernel_size=kernel, strides=2, padding='same')(model)

    dec = BatchNormalization(epsilon=1e-5)(model)
    dec = LeakyReLU(alpha=0.2)(dec)
    dec = Flatten()(dec)
    dec = Dense(1, activation='sigmoid')(dec)

    output = Model(X, dec)
    output.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5), metrics=
['accuracy'])

    return output


def VAEGAN(decgen,discriminator):
    # generator
    g = decgen

    # discriminator
    d = discriminator
    d.trainable = False
    # GAN
    gan = Sequential([g, d])

#     sgd=SGD()
    gan.compile(optimizer=Adam(lr=0.0001, beta_1=0.5), loss='binary_crossentropy')
    return g, d, gan
```

```
    #print(image.size)

    return np.array(image.convert(mode))

def show_imgs(generator):
    row=3
    col = 5
    noise = np.random.normal(0, 1, (row*col, latent_dim))
    gen_imgs = generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(row, col)
    #fig.suptitle("DCGAN: Generated digits", fontsize=12)
    cnt = 0

    for i in range(row):
        for j in range(col):
            axs[i, j].imshow(gen_imgs[cnt, :, :, :])
            axs[i, j].axis('off')
            cnt += 1

    #plt.close()
    plt.show()
```

The parameters of the generator will be affected by both the GAN and VAE training.

```
# note: The parameters of the generator will be affected by both the GAN and VAE training


G, D, GAN = VAEGAN(decgen(5, filter_of_decgen, rows, columns, channel),discriminator(5, filter_of_dis,
rows, columns, channel))

# encoder
E = encoder(5, filter_of_encoder, rows, columns, channel)
print("This is the summary for encoder:")
E.summary()


# generator/decoder
# G = decgen(5, filter_of_decgen, rows, columns, channel)
print("This is the summary for dencoder/generator:")
G.summary()


# discriminator
# D = discriminator(5, filter_of_dis, rows, columns, channel)
print("This is the summary for discriminator:")
D.summary()


D_fixed = discriminator(5, filter_of_dis, rows, columns, channel)
D_fixed.compile(optimizer=SGDop, loss='mse')

# gan
print("This is the summary for GAN:")
GAN.summary()

# VAE
X = Input(shape=(rows, columns, channel))

E_mean, E_logsigma, Z = E(X)

output = G(Z)
# G_dec = G(E_mean + E_logsigma)
# D_fake, F_fake = D(output)
# D_fromGen, F_fromGen = D(G_dec)
# D_true, F_true = D(X)

# print("type(E)",type(E))
# print("type(output)",type(output))
# print("type(D_fake)",type(D_fake))
```

```
VAE.summary()
```

In the below cell we begin training our model. Note that we use the previous method to train the discriminator and GAN and VAE for different lengths of time. We emphasize the training of the discriminator in the first half of the training process and we train the generator more in the second half because we want to improve the quality of output images.

```
# We train our model in this cell

dLoss=[]
gLoss=[]
GLoss = 1
GlossEnc = 1
GlossGen = 1
Eloss = 1

halfbatch_size = int(batch_size*0.5)

for epoch in tqdm(range(epochs)):
    if epoch < int(epochs*0.5):
        noise = np.random.normal(0, 1, (halfbatch_size, latent_dim))
        index = np.random.randint(0,dataset.shape[0], halfbatch_size)
        images = dataset[index]

        latent_vect = E.predict(images)[0]
        encImg = G.predict(latent_vect)
        fakeImg = G.predict(noise)

        D.Trainable = True
        DlossTrue = D.train_on_batch(images, np.ones((halfbatch_size, 1)))
        DlossEnc = D.train_on_batch(encImg, np.ones((halfbatch_size, 1)))
        DlossFake = D.train_on_batch(fakeImg, np.zeros((halfbatch_size, 1)))

#         DLoss=np.add(DlossTrue,DlossFake)*0.5

        DLoss=np.add(DlossTrue,DlossEnc)
        DLoss=np.add(DLoss,DlossFake)*0.33
        D.Trainable = False

        cnt = epoch

        while cnt > 3:
            cnt = cnt - 4

        if cnt == 0:
            noise = np.random.normal(0, 1, (batch_size, latent_dim))
            index = np.random.randint(0,dataset.shape[0], batch_size)
            images = dataset[index]
            latent_vect = E.predict(images)[0]

            GlossEnc = GAN.train_on_batch(latent_vect, np.ones((batch_size, 1)))
            GlossGen = GAN.train_on_batch(noise, np.ones((batch_size, 1)))
            Eloss = VAE.train_on_batch(images, None)
            GLoss=np.add(GlossEnc,GlossGen)
            GLoss=np.add(GLoss,Eloss)*0.33
        dLoss.append([epoch,DLoss[0]])
        gLoss.append([epoch,GLoss])

    elif epoch >= int(epochs*0.5):
        cnt = epoch
        while cnt > 3:
            cnt = cnt - 4

        if cnt == 0:
            noise = np.random.normal(0, 1, (halfbatch_size, latent_dim))
            index = np.random.randint(0,dataset.shape[0], halfbatch_size)
            images = dataset[index]

            latent_vect = E.predict(images)[0]
            encImg = G.predict(latent_vect)
            fakeImg = G.predict(noise)

            D.Trainable = True
            DlossTrue = D.train_on_batch(images, np.ones((halfbatch_size, 1)))
```

```
#             DLoss=np.add(DlossTrue,DlossEnc)
#             DLoss=np.add(DLoss,DlossFake)*0.33
            D.Trainable = False

        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        index = np.random.randint(0,dataset.shape[0], batch_size)
        images = dataset[index]
        latent_vect = E.predict(images)[0]

        GlossEnc = GAN.train_on_batch(latent_vect, np.ones((batch_size, 1)))
        GlossGen = GAN.train_on_batch(noise, np.ones((batch_size, 1)))
        Eloss = VAE.train_on_batch(images, None)
        GLoss=np.add(GlossEnc,GlossGen)
        GLoss=np.add(GLoss,Eloss)*0.33

        dLoss.append([epoch,DLoss[0]])
        gLoss.append([epoch,GLoss])

    if epoch % plotInternal == 0 and epoch!=0:
        show_imgs(G)


    dLossArr= np.array(dLoss)
    gLossArr = np.array(gLoss)

#     print("dLossArr.shape:",dLossArr.shape)
#     print("gLossArr.shape:",gLossArr.shape)

    chk = epoch

    while chk > 50:
        chk = chk - 51

    if chk == 0:
        D.save_weights('/content/gdrive/My Drive/VAE discriminator_kernalsize5_proportion_32.h5')
        G.save_weights('/content/gdrive/My Drive/VAE generator_kernalsize5_proportion_32.h5')
        E.save_weights('/content/gdrive/My Drive/VAE encoder_kernalsize5_proportion_32.h5')


    if epoch%20 == 0:
        print("epoch:", epoch + 1,"  ", "DislossTrue loss:",DlossTrue[0],"D accuracy: ",100* DlossTrue[1],
"DlossFake loss:", DlossFake[0],"GlossEnc loss:",
          GlossEnc, "GlossGen loss:",GlossGen, "Eloss loss:",Eloss)
#     print("loss:")
#     print("D:", DlossTrue, DlossEnc, DlossFake)
#     print("G:", GlossEnc, GlossGen)
#     print("VAE:", Eloss)

print('Training done,saving weights')
D.save_weights('/content/gdrive/My Drive/VAE discriminator_kernalsize5_proportion_32.h5')
G.save_weights('/content/gdrive/My Drive/VAE generator_kernalsize5_proportion_32.h5')
E.save_weights('/content/gdrive/My Drive/VAE encoder_kernalsize5_proportion_32.h5')


print('painting losses')
# At the end of training plot the losses vs epochs
plt.plot(dLossArr[:, 0], dLossArr[:, 1], label="Discriminator Loss")
plt.plot(gLossArr[:, 0], gLossArr[:, 1], label="Generator Loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('GAN')
plt.grid(True)
plt.show()
print('end')
```

If you are planning on running this network, beware that the training process takes a REALLY long time. I would not attempt this unless you have access to some powerful GPUs or are willing to run the model for an entire day.

Now our VAE-GAN training is complete, we can check to see how our output images look and compare them to our previous GANs.

```
# In this cell, we generate and visualize 15 images.
```

We can see that in this implementation of VAE-GAN, we got a nice model which can generate images that are clear and of a similar style to the original images. Our VAE-GAN can create images more robustly and this can be done without extra noise of the anime faces. However, the competence of generalization of our model is not very good, it seldom changes the manner or sex of the character, so this is a point that we could try to improve.

Final Comments

It is not necessarily clear that any one of the models is better than the others, and none of these methods have been optimized properly so it is difficult to make a comparison.

This is still an active area of research, so if you are interested I recommend getting yourself stuck in and try and use GANs within your own work to see what you can come up with.

I hope you have enjoyed this trilogy of articles on GANs and now have a much better idea of what they are, what they can do, and how to make your own.

Thank you for reading!

Newsletter

For updates on new blog posts and extra content, sign up for my newsletter.

Further Reading

Run BigGAN in COLAB:

- https://www.jessicayung.com/explaining-tensorflow-code-for-a-convolutional-neural-network/
- https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html
- https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
- https://github.com/tensorlayer/srgan
- https://junyanz.github.io/CycleGAN/ https://affinelayer.com/pixsrv/
- https://tcwang0509.github.io/pix2pixHD/

Influential Papers:

- DCGAN https://arxiv.org/pdf/1511.06434v2.pdf
- Wasserstein GAN (WGAN) https://arxiv.org/pdf/1701.07875.pdf
- Conditional Generative Adversarial Nets (CGAN) https://arxiv.org/pdf/1411.1784v1.pdf
- Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks (LAPGAN) https://arxiv.org/pdf/1506.05751.pdf
- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network (SRGAN) https://arxiv.org/pdf/1609.04802.pdf
- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (CycleGAN) https://arxiv.org/pdf/1703.10593.pdf
- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets https://arxiv.org/pdf/1606.03657
- DCGAN https://arxiv.org/pdf/1704.00028.pdf
- Improved Training of Wasserstein GANs (WGAN-GP) https://arxiv.org/pdf/1701.07875.pdf
- Energy-based Generative Adversarial Network (EBGAN) https://arxiv.org/pdf/1609.03126.pdf
- Autoencoding beyond pixels using a learned similarity metric (VAE-GAN) https://arxiv.org/pdf/1512.09300.pdf
- Adversarial Feature Learning (BiGAN) https://arxiv.org/pdf/1605.09782v6.pdf
- Stacked Generative Adversarial Networks (SGAN) https://arxiv.org/pdf/1612.04357.pdf
- StackGAN++: Realistic Image Synthesis with Stacked Generative Adversarial Networks https://arxiv.org/pdf/1710.10916.pdf
- Learning from Simulated and Unsupervised Images through Adversarial Training (SimGAN) https://arxiv.org/pdf/1612.07828v1.pdf