Agenda:

1. Generate new anime characters
2. Understand the architecture of Generative Adverserial Network(GAN)
3. Applications of GAN

Problem Statement:

`You are working as a Data Scientist at Kyoto Animation, a Japanese animation studio`

- In 2020, about 179 new animes were released and total more than 4505 anime have been released.
- Anime industry wants to develop an automated system to generate newer anime characters.

How would a human create new anime characters ?

1. Character Profile: Choose the artistic style, skin tone, hairstyle and gesture.
2. Rough Character Sketches
3. Developing the Character Design
4. Coloring an Anime Character

Task: Generate new anime character images

How can we achieve this?

- We can provide samples of anime images to our deep learning model and make it generate similar images.

Have we encountered a similar problem before?

- `No`

Can we use any computer vision or CNN algorithm learned so far to solve this problem?

- `Yes`, we can use some concepts similar to Autoencoder or U-Net architecture, where the output is the generated anime images.

Are Autoencoders sufficient or do we need to change something?

- `No`, Autoencoders have following limitations:

> 1. Their goal is to learn how to reconstruct the input-data.
> 2. They tend to overfit and they suffer from the vanishing gradient problem.
> 3. Less variation, Generated images are too similar to the input data and no new designs

Let's have a look at a possible solution

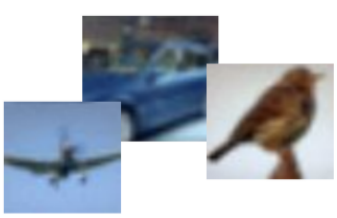
How can we achieve this?

- `Through Generative Models`

What are Generative models?

> Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

- **More concretely, these models learn the distribution of the given training data & it generates new samples of data of the same distribution.**



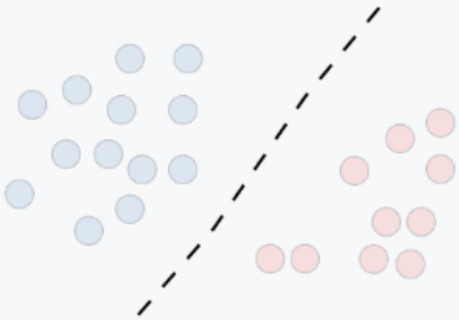
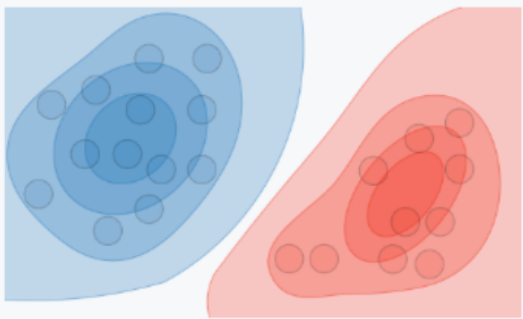
Training data ~ $p_{data}(x)$

Generated samples ~ $p_{model}(x)$

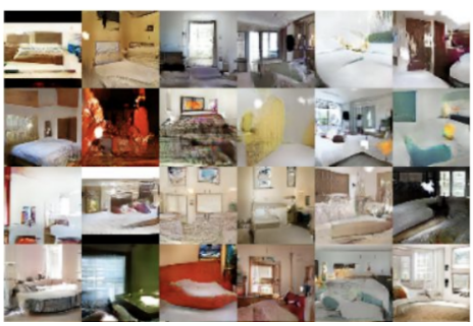
Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

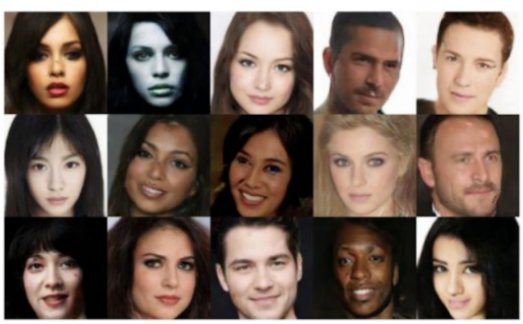
Discriminative vs Generative models?

- Discriminative models draw boundaries in the data space, while generative models try to model how data is placed throughout the space.
- A generative model focuses on explaining how the data was generated, while a discriminative model focuses on predicting the labels of the data.

	Discriminative model	**Generative model**
Goal	Directly estimate $P(y\|x)$	Estimate $P(x\|y)$ to then deduce $P(y\|x)$
What's learned	Decision boundary	Probability distributions of the data
Illustration		
Examples	Regressions, SVMs	GDA, Naive Bayes

- Generative Models can generate realistic samples for artwork, super-resolution, colorization, etc.





- We will be using GANS(Generative Adverserial Network) for generating new anime

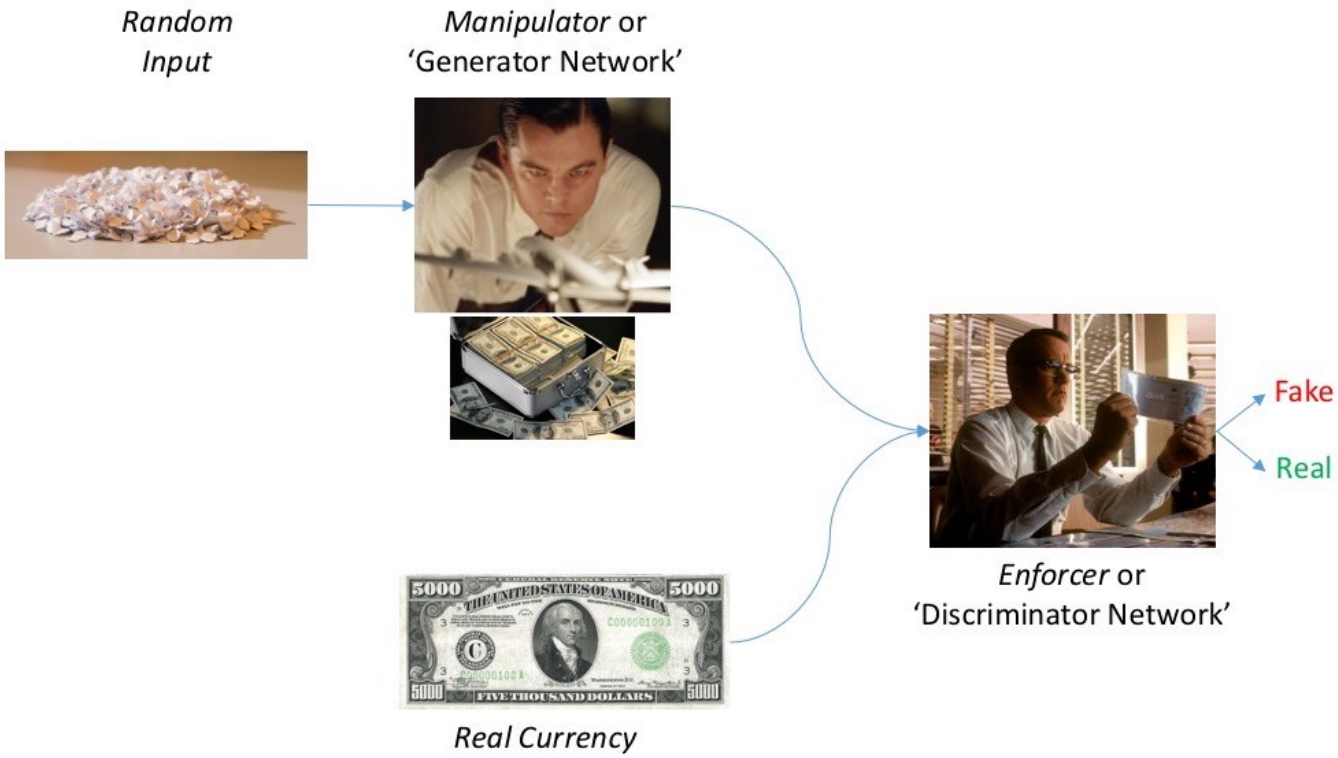
How to generate better quality images?

Can we make machines learn-to-teach themselves?

- Yes, this is doable.

```
What if we pitted two neural networks against each other?
```

```
Let us solve this using an analogy
```



- Setting off the perfect cat-and-mouse game.
- Imagine a quintessential movie where two estranged brothers embrace opposing philosophies in life. One starts a fresh underworld operation printing fake currencies as a 'manipulator', and the other enrols in a bureau to set up a new division that detects counterfeits as an 'enforcer'.
- To start with, lets say that the 'manipulator' in underworld starts with a disadvantage of knowing nothing about what original currencies look like. The 'enforcer' in the bureau knows just basics of how few real currencies look. And then the game begins.

- The manipulator starts printing, but the initial fakes are terrible. It doesn't need even a trained eye to detect the counterfeits, and promptly every single one of them is detected by the enforcer.

- The manipulator is industrious and keeps churning out fakes, while also learning what didn't work in previous attempts. By sheer magnitude of experimentation with fakes & some feedback, the quality of counterfeits slowly starts inching up (of course, assuming the operation is not shut down!)

- Eventually, the manipulator starts getting a few random counterfeits right and this goes undetected by the enforcer. So, its learning time on the other side and the enforcer takes lessons on detecting these smarter counterfeits.

- With the enforcer getting smarter, the counterfeits are detected again. The manipulator has no choice but to upgrade the counterfeiting operation to create more genuine-looking fakes.

- This continuous game of cat-and-mouse continues, and ends up making experts out of both the manipulator and enforcer. So much so that the counterfeits are indistinguishable from the genuine ones, and also the detection of such ingenious fakes becomes almost uncanny.

```
You get the drift. And this, is the underlying concept of Generative
Adversarial Networks (GANs)
```

Quiz -1

GANs are composed of two models that compete against each other in order to achieve equilibrium

(a) True

(b) False

Ans : (a) True

- We could train one model to generate the data and another model to verify the quality of the generated data.
- This approach is known as Generative Adversarial Networks (GANs)

Introducing GANs

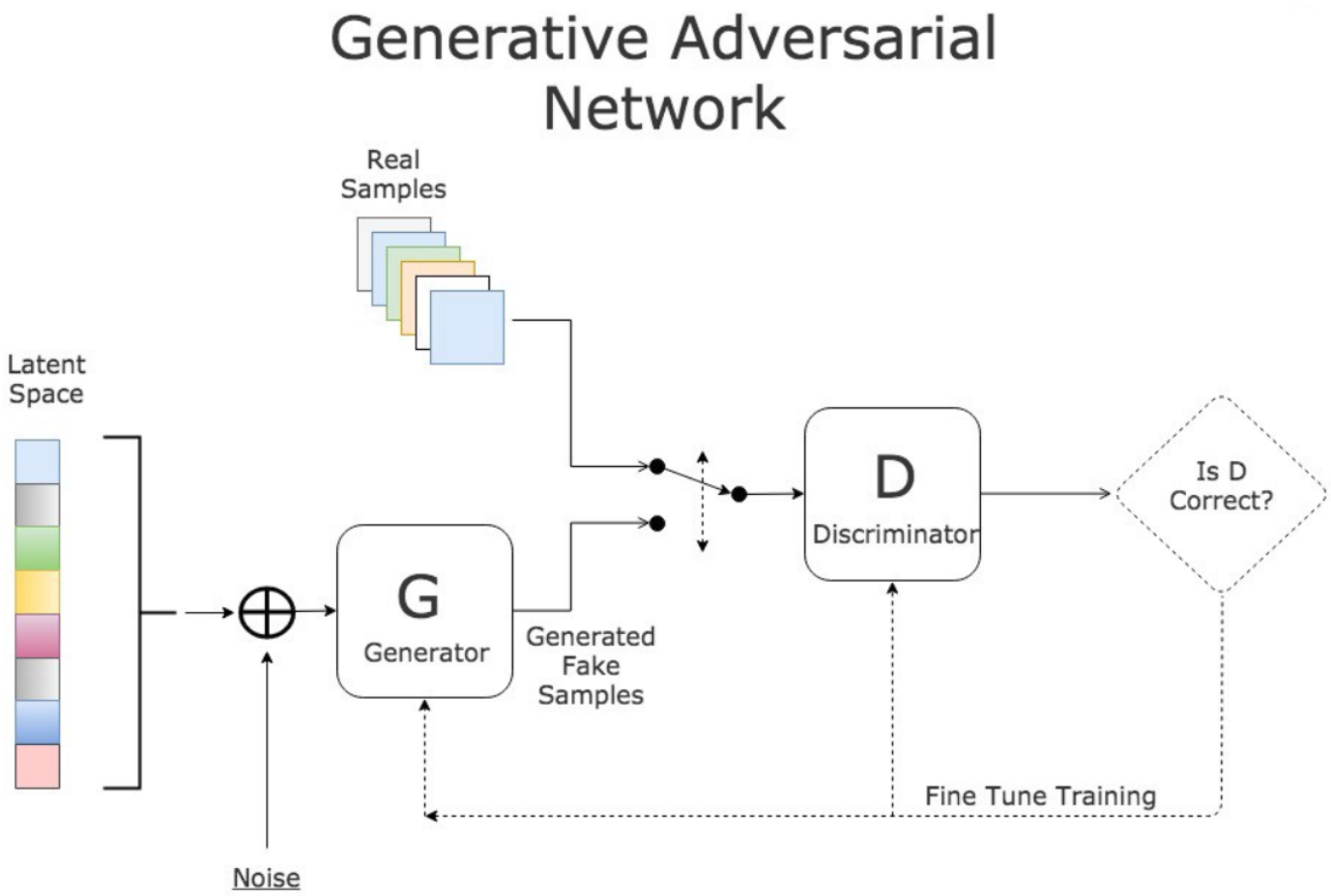
- Both the manipulator and enforcer are models, a variant of the Deep learning neural networks.

1. The manipulator is called the 'Generator network'. The job of the Generator is to produce fake image that looks real so that it can fool the Discriminator.

2. The enforcer is the 'Discriminator network'. The job of Discriminator is to become better and better in disinguishing real and fake image.

3. By pairing two models against each other as adversaries, we set them up for a healthy competition. Each tries mastering its own job across thousands of iterations, with no manual intervention.

4. This way both the generator and the discriminator get better at doing their jobs.



Note: Generative modeling is an unsupervised learning task, so the images do not have any labels.

- In this lecture, we'll train a GAN to generate images of anime characters.
- Generative models can create plausible new data of their own.
- But the results were often not very good: computer-generated images tended to be blurry or have errors like missing features.
- GANs are extremely sensitive to hyperparameters, activation functions and regularization.

How does the data look?

Does the dataset have any labels?

- No, since we are going to work with unsupervised machine learning, we won't be needing any labels to train our network.

```
# Load the Drive helper and mount
from google.colab import drive
drive.mount('/content/drive')
```

```
---------------------------------------------------------------------------
MessageError                                Traceback (most recent call last)
<ipython-input-1-1dbcf436c2dc> in <module>
      1 # Load the Drive helper and mount
      2 from google.colab import drive
----> 3 drive.mount('/content/drive')

                                  3 frames
/usr/local/lib/python3.7/dist-packages/google/colab/_message.py in
read_reply_from_input(message_id, timeout_sec)
    100         reply.get('colab_msg_id') == message_id):
    101       if 'error' in reply:
--> 102         raise MessageError(reply['error'])
    103       return reply.get('data', None)
    104

MessageError: Error: credential propagation was unsuccessful
```

SEARCH STACK OVERFLOW

```
!gdown 1tkKn01cnF3MH7-8mQzIay7ShMcgdZXF7
```

```
Downloading...
From: https://drive.google.com/uc?id=1tkKn01cnF3MH7-8mQzIay7ShMcgdZXF7
To: /content/animefacedataset.zip
100% 418M/418M [00:01<00:00, 254MB/s]
```

```
# unzip the dataset in local directory
!unzip '/content/animefacedataset.zip' -d '/content/animefacedataset'
```

```
  inflating: /content/animefacedataset/content/animefacedataset/images/28263
  inflating: /content/animefacedataset/content/animefacedataset/images/11733
  inflating: /content/animefacedataset/content/animefacedataset/images/47643
  inflating: /content/animefacedataset/content/animefacedataset/images/12792
  inflating: /content/animefacedataset/content/animefacedataset/images/48652
  inflating: /content/animefacedataset/content/animefacedataset/images/51581
  inflating: /content/animefacedataset/content/animefacedataset/images/4762_
  inflating: /content/animefacedataset/content/animefacedataset/images/53747
  inflating: /content/animefacedataset/content/animefacedataset/images/28736
  inflating: /content/animefacedataset/content/animefacedataset/images/38450
  inflating: /content/animefacedataset/content/animefacedataset/images/25232
  inflating: /content/animefacedataset/content/animefacedataset/images/53444
  inflating: /content/animefacedataset/content/animefacedataset/images/41840
  inflating: /content/animefacedataset/content/animefacedataset/images/6038
```

```
inflating: /content/animefacedataset/content/animefacedataset/images/24648
inflating: /content/animefacedataset/content/animefacedataset/images/33313
inflating: /content/animefacedataset/content/animefacedataset/images/16740
inflating: /content/animefacedataset/content/animefacedataset/images/33568
inflating: /content/animefacedataset/content/animefacedataset/images/3501_
inflating: /content/animefacedataset/content/animefacedataset/images/61776
inflating: /content/animefacedataset/content/animefacedataset/images/48729
inflating: /content/animefacedataset/content/animefacedataset/images/35588
inflating: /content/animefacedataset/content/animefacedataset/images/29847
inflating: /content/animefacedataset/content/animefacedataset/images/27413
inflating: /content/animefacedataset/content/animefacedataset/images/46513
inflating: /content/animefacedataset/content/animefacedataset/images/20049
inflating: /content/animefacedataset/content/animefacedataset/images/7252_
inflating: /content/animefacedataset/content/animefacedataset/images/28748
inflating: /content/animefacedataset/content/animefacedataset/images/12339
inflating: /content/animefacedataset/content/animefacedataset/images/25458
inflating: /content/animefacedataset/content/animefacedataset/images/2436_
inflating: /content/animefacedataset/content/animefacedataset/images/14414
inflating: /content/animefacedataset/content/animefacedataset/images/47759
inflating: /content/animefacedataset/content/animefacedataset/images/33984
inflating: /content/animefacedataset/content/animefacedataset/images/17029
inflating: /content/animefacedataset/content/animefacedataset/images/18439
inflating: /content/animefacedataset/content/animefacedataset/images/13021
inflating: /content/animefacedataset/content/animefacedataset/images/16606
inflating: /content/animefacedataset/content/animefacedataset/images/32645
inflating: /content/animefacedataset/content/animefacedataset/images/38533
inflating: /content/animefacedataset/content/animefacedataset/images/15039
inflating: /content/animefacedataset/content/animefacedataset/images/23353
inflating: /content/animefacedataset/content/animefacedataset/images/62159
inflating: /content/animefacedataset/content/animefacedataset/images/8745_
inflating: /content/animefacedataset/content/animefacedataset/images/34366
inflating: /content/animefacedataset/content/animefacedataset/images/61852
inflating: /content/animefacedataset/content/animefacedataset/images/8556_
inflating: /content/animefacedataset/content/animefacedataset/images/27537
inflating: /content/animefacedataset/content/animefacedataset/images/7486_
inflating: /content/animefacedataset/content/animefacedataset/images/51600
inflating: /content/animefacedataset/content/animefacedataset/images/6532_
inflating: /content/animefacedataset/content/animefacedataset/images/46697
inflating: /content/animefacedataset/content/animefacedataset/images/16572
inflating: /content/animefacedataset/content/animefacedataset/images/12891
inflating: /content/animefacedataset/content/animefacedataset/images/43024
inflating: /content/animefacedataset/content/animefacedataset/images/56980
inflating: /content/animefacedataset/content/animefacedataset/images/11433
inflating: /content/animefacedataset/content/animefacedataset/images/12421
```

How many samples in the data?

```
import os
import gdown
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt

image_dimensions = (64, 64)
batch_size = 256
```

```
# set path to the dataset
dataset_path = "./animefacedataset"

dataset = keras.preprocessing.image_dataset_from_directory(
    dataset_path, label_mode=None, image_size=image_dimensions, batch_size=batch_si
)

# scaling images to -1 to 1
dataset = dataset.map(lambda x: (x - 127.5) / 127.5)
```

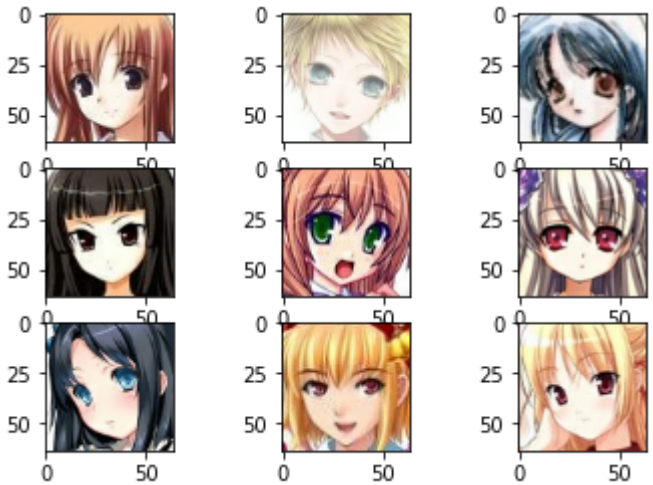
```
Found 63565 files belonging to 1 classes.
```

Is the data enough?

What is the dimension of each sample image?

- Let's visualize some samples and check the image dimension

```
from matplotlib import pyplot

# Display grid of images from dataset
def display_images(total=9): # default total images to display = 9
    num=total
    for x in dataset:
        pyplot.subplot(330 + 1 + total - num)
        plt.imshow(( (x.numpy()*0.5 + 0.5) * 255).astype("int32")[0]) #pyplot.imsho
        num-=1
        if not num:
            break

display_images()
```




```
def display_single(dataloader):
    for x in dataloader:
        plt.axis("off")
```

```
            print("Image Dimensions: ",x.numpy().shape)
            plt.imshow(((x.numpy()*0.5 + 0.5) * 255).astype("int32")[0])
            break

display_single(dataset)
```

```
Image Dimensions:  (256, 64, 64, 3)
```

```
import numpy as np
from sklearn.utils import shuffle
import time
import cv2
from tqdm.notebook import tqdm
from PIL import Image
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.layers.core import Activation
from keras.layers.core import Flatten, Dropout
from keras.layers import Input, merge
from keras.layers.pooling import MaxPooling2D
from keras.layers.convolutional import Conv2D, Conv2DTranspose
import matplotlib.pyplot as plt
import keras.backend as K
from keras.initializers import RandomNormal

img_shape = (64, 64, 3)
```

Double-click (or enter) to edit

How does a discriminative model work?

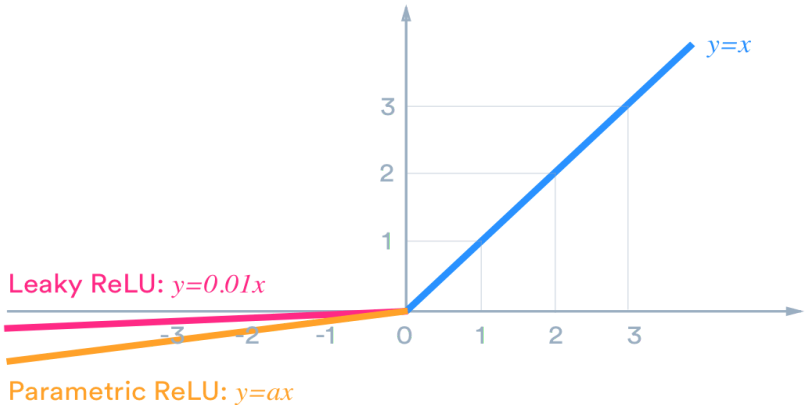
- The discriminator takes an image as input, and tries to classify it as "real" or "generated".
- Just like any other binary classification model, the output of the discriminator is a single number between 0 and 1, which can be interpreted as the probability of the input image

being real i.e. picked from the original dataset.

- It's like any other neural network.
- We'll use a convolutional neural networks (CNN) which outputs a single number output for every image.
- We'll use stride of 2 to progressively reduce the size of the output feature map.
- In discriminator we found the Leaky ReLU is used except at the output layer where Sigmoid is used.

What is leaky ReLU ?

- Different from the regular ReLU function, Leaky ReLU allows the pass of a small gradient signal for negative values. As a result, it makes the gradients from the discriminator flow stronger into the generator.



Quiz-2

Why do we require LeakyReLU in discriminator ?

> (a) It will help the discriminator to distinguish between real and fake image

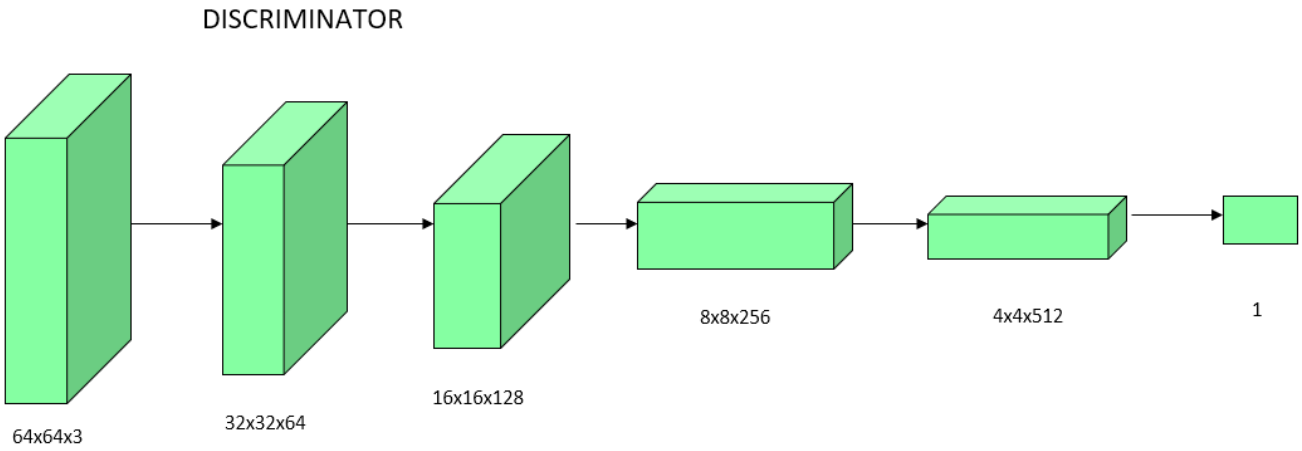
> (b) It will help the generator to learn as gradient will flow always from the discriminator

> (c) It is computationally inexpensive

> (d) It will saturate

Ans : (b) It will help the generator to learn as gradient will flow always from the discriminator. If ReLU was used then the Generator would learn only from the positive gradients but in Leaky ReLU the Generator will learn from both positive and negative gradient

Architecture of Discriminator




```
def get_disc_normal(image_shape=(64,64,3)):
    image_shape = image_shape

    dropout_prob = 0.4

    #kernel_init = RandomNormal(mean=0.0, stddev=0.01)
    kernel_init = 'glorot_uniform'

    dis_input = Input(shape = image_shape)

    discriminator = Conv2D(filters = 64, kernel_size = (4,4), strides = (2,2), padd
    discriminator = LeakyReLU(0.2)(discriminator)
    #discriminator = MaxPooling2D(pool_size=(2, 2))(discriminator)

    #discriminator = Dropout(dropout_prob)(discriminator)
    discriminator = Conv2D(filters = 128, kernel_size = (4,4), strides = (2,2), pad
    discriminator = BatchNormalization(momentum = 0.5)(discriminator)
    discriminator = LeakyReLU(0.2)(discriminator)
    #discriminator = MaxPooling2D(pool_size=(2, 2))(discriminator)

    #discriminator = Dropout(dropout_prob)(discriminator)
    discriminator = Conv2D(filters = 256, kernel_size = (4,4), strides = (2,2), pad
    discriminator = BatchNormalization(momentum = 0.5)(discriminator)
    discriminator = LeakyReLU(0.2)(discriminator)
    #discriminator = MaxPooling2D(pool_size=(2, 2))(discriminator)

    #discriminator = Dropout(dropout_prob)(discriminator)
    discriminator = Conv2D(filters = 512, kernel_size = (4,4), strides = (2,2), pad
    discriminator = BatchNormalization(momentum = 0.5)(discriminator)
    discriminator = LeakyReLU(0.2)(discriminator)
    #discriminator = MaxPooling2D(pool_size=(2, 2))(discriminator)

    discriminator = Flatten()(discriminator)

    #discriminator = MinibatchDiscrimination(100,5)(discriminator)
    discriminator = Dense(1)(discriminator)
    discriminator = Activation('sigmoid')(discriminator)

    discriminator_model = Model(dis_input, discriminator)
```

```
    discriminator_model.summary()
    return discriminator_model


discriminator = get_disc_normal()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 64, 64, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 64)        3136

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 64)        0

 conv2d_1 (Conv2D)           (None, 16, 16, 128)       131200

 batch_normalization (BatchN  (None, 16, 16, 128)      512
 ormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 16, 16, 128)       0

 conv2d_2 (Conv2D)           (None, 8, 8, 256)         524544

 batch_normalization_1 (Batc  (None, 8, 8, 256)        1024
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 8, 8, 256)         0

 conv2d_3 (Conv2D)           (None, 4, 4, 512)         2097664

 batch_normalization_2 (Batc  (None, 4, 4, 512)        2048
 hNormalization)

 leaky_re_lu_3 (LeakyReLU)   (None, 4, 4, 512)         0

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 1)                 8193

 activation (Activation)     (None, 1)                 0

=================================================================
Total params: 2,768,321
Trainable params: 2,766,529
Non-trainable params: 1,792
_________________________________________________________________
```

Quiz-2

Why does discriminator does not use the GAP(Global Average Pooling) followed by FC layers after the Conv Layer ?

> (a) Because we can pass input image of any shape if FC layers are not used

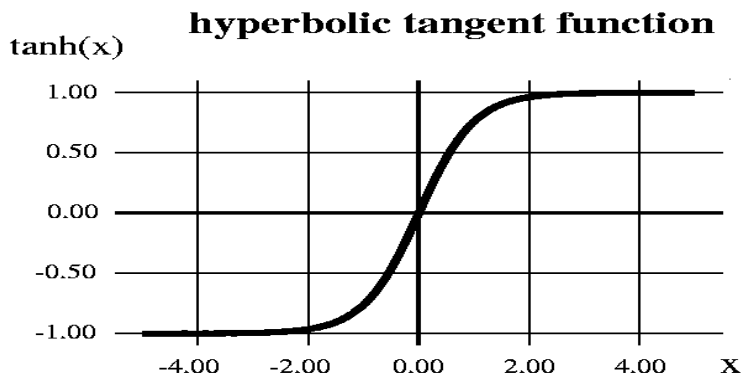
> (b) To reduce the no of parameters as FC layers will increase the number of parameters

> (c) GAP increase the model stability but it hurts the convergence of GAN

Ans: (c) GAP increase the model stability but it hurts the convergence of GAN. Refer Page 3 of DCGAN paper Click to know more

How does a generator model work?

- The input to the generator is typically a vector or a matrix of random numbers (referred to as a latent tensor) which is used as a seed for generating an image.
- The generator will convert a latent tensor of shape (128, 1, 1) into an image tensor of shape 3 x 64 x 64.
- To achieve this, we'll use the Conv2DTranspose layer from Keras, which is performed as a transposed convolution (also referred to as a deconvolution).
- In generator, ReLU is used except at the output layer where tanh activation function is used



Quiz-3

Why was the real image standardized to a range of [-1,1] ?

> (a) The outputs of the tanh activation lie in the range [-1,1] so thats why we have applied the similar transformation to the images in the training dataset.

> (b) Standardization of image to a range of [-1,1] is better than range of [0,1]

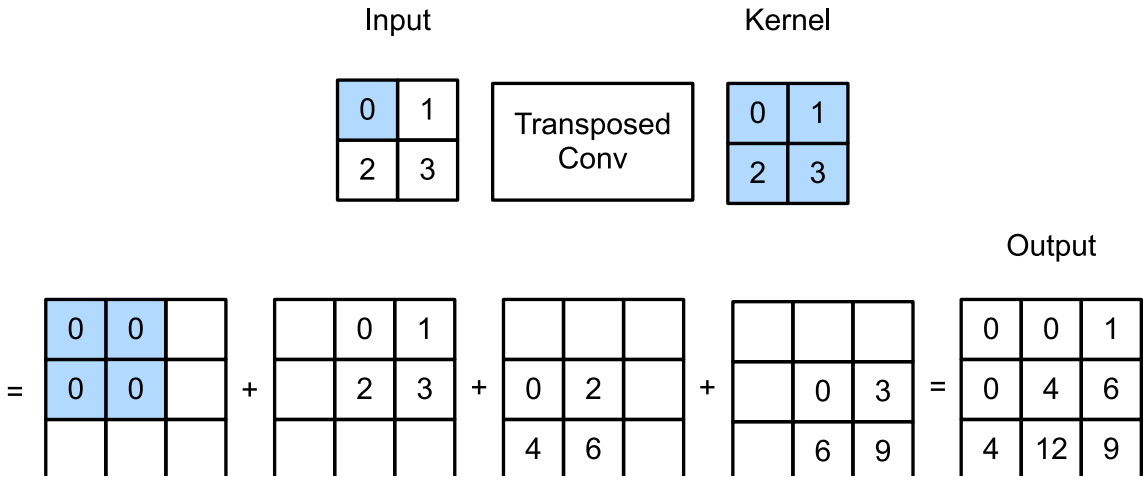
> (c) Image standardized to a range of [-1,1] will be zero centered

> (d) We could have standardized image to range of [-1,1] or [0,1]. Both will work equally good

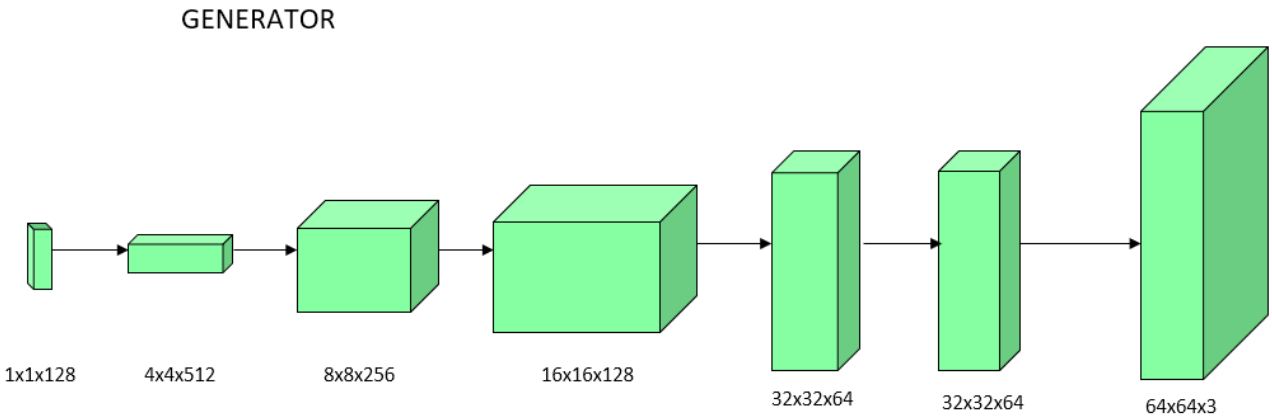
Ans (a) The outputs of the tanh activation lie in the range [-1,1] so thats why we have applied the similar transformation to the images in the training dataset.

So how does a deconvolution layer works?

- A deconvolution is a mathematical operation that reverses the effect of convolution.
- For example, lets take input matrix of shape 2x2 and kernel of shape 2x2. We'll get the output matrix of shape 3x3.



Architecture of Generator



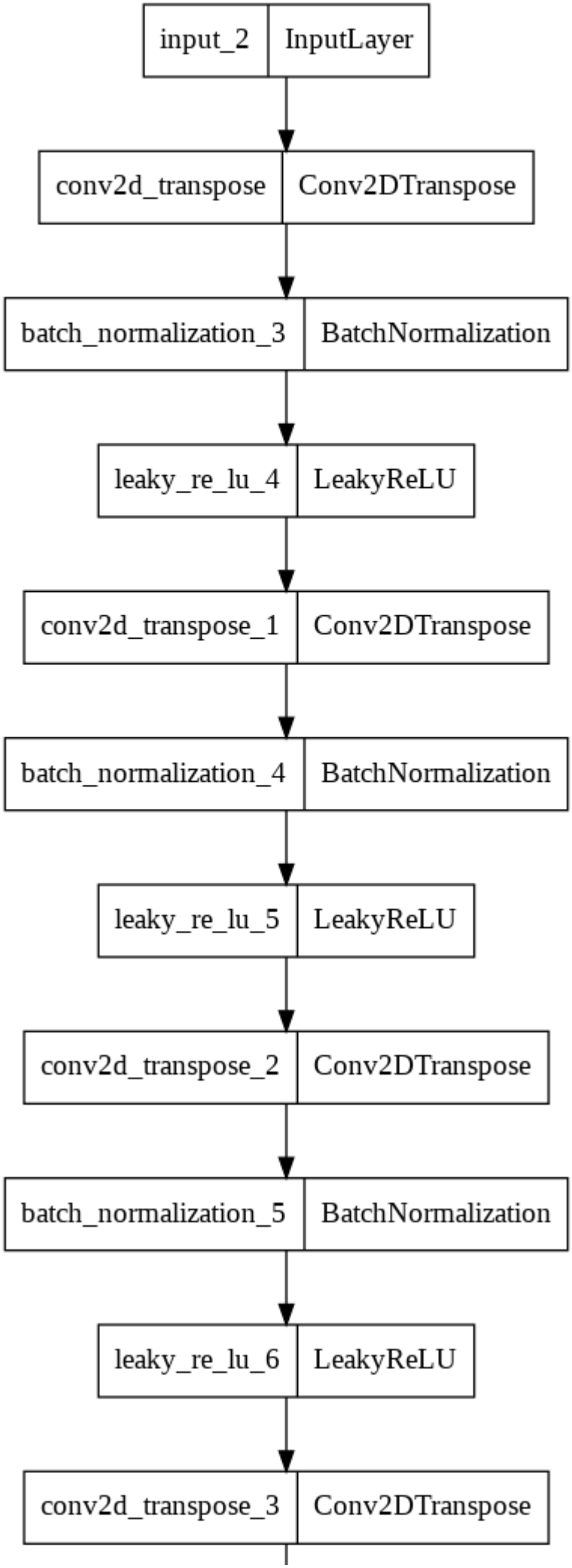

```
latent_dim = 128

def get_gen_normal(noise_shape = (1,1,128)):
    noise_shape = noise_shape
    """
    Changing padding = 'same' in the first layer makes a lot fo difference!!!!
    """
    #kernel_init = RandomNormal(mean=0.0, stddev=0.01)
    kernel_init = 'glorot_uniform'

    gen_input = Input(shape = noise_shape) #if want to directly use with conv layer
    #gen_input = Input(shape = [noise_shape]) #if want to use with dense layer next

    generator = Conv2DTranspose(filters = 512, kernel_size = (4,4), strides = (1,1)
    generator = BatchNormalization(momentum = 0.5)(generator)
    generator = LeakyReLU(0.2)(generator)
```

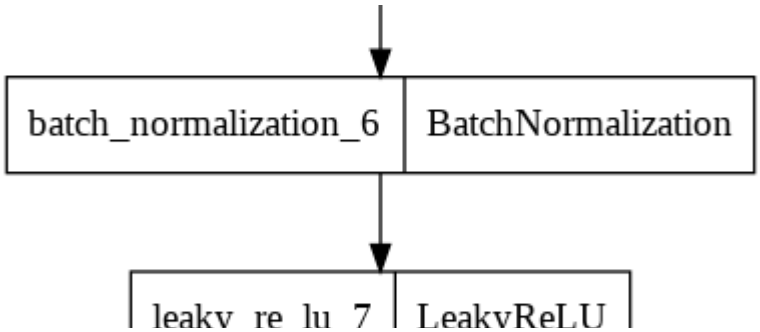
```
    generator = Conv2DTranspose(filters = 256, kernel_size = (4,4), strides = (2,2)
    generator = BatchNormalization(momentum = 0.5)(generator)
    generator = LeakyReLU(0.2)(generator)

    generator = Conv2DTranspose(filters = 128, kernel_size = (4,4), strides = (2,2)
    generator = BatchNormalization(momentum = 0.5)(generator)
    generator = LeakyReLU(0.2)(generator)

    generator = Conv2DTranspose(filters = 64, kernel_size = (4,4), strides = (2,2),
    generator = BatchNormalization(momentum = 0.5)(generator)
    generator = LeakyReLU(0.2)(generator)

    generator = Conv2D(filters = 64, kernel_size = (3,3), strides = (1,1), padding
    generator = BatchNormalization(momentum = 0.5)(generator)
    generator = LeakyReLU(0.2)(generator)

    generator = Conv2DTranspose(filters = 3, kernel_size = (4,4), strides = (2,2),
    generator = Activation('tanh')(generator)

    generator_model = Model(gen_input, generator)
    generator_model.summary()

    return generator_model


generator = get_gen_normal()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 1, 1, 128)]       0

 conv2d_transpose (Conv2DTra  (None, 4, 4, 512)        1049088
 nspose)

 batch_normalization_3 (Batc  (None, 4, 4, 512)        2048
 hNormalization)

 leaky_re_lu_4 (LeakyReLU)   (None, 4, 4, 512)         0

 conv2d_transpose_1 (Conv2DT  (None, 8, 8, 256)        2097408
 ranspose)

 batch_normalization_4 (Batc  (None, 8, 8, 256)        1024
 hNormalization)

 leaky_re_lu_5 (LeakyReLU)   (None, 8, 8, 256)         0

 conv2d_transpose_2 (Conv2DT  (None, 16, 16, 128)      524416
 ranspose)

 batch_normalization_5 (Batc  (None, 16, 16, 128)      512
 hNormalization)

 leaky_re_lu_6 (LeakyReLU)   (None, 16, 16, 128)       0
```

```
conv2d_transpose_3 (Conv2DT  (None, 32, 32, 64)       131136
ranspose)

batch_normalization_6 (Batc  (None, 32, 32, 64)       256
hNormalization)

leaky_re_lu_7 (LeakyReLU)   (None, 32, 32, 64)        0

conv2d_4 (Conv2D)           (None, 32, 32, 64)        36928

batch_normalization_7 (Batc  (None, 32, 32, 64)       256
hNormalization)

leaky_re_lu_8 (LeakyReLU)   (None, 32, 32, 64)        0

conv2d_transpose_4 (Conv2DT  (None, 64, 64, 3)        3075
ranspose)

activation_1 (Activation)   (None, 64, 64, 3)         0

=================================================================
Total params: 3,846,147
Trainable params: 3,844,099
Non-trainable params: 2,048
_________________________________________________________________
```

```
tf.keras.utils.plot_model(generator)
```


input_2 InputLayer
conv2d_transpose Conv2DTranspose
batch_normalization_3 BatchNormalization
leaky_re_lu_4 LeakyReLU
conv2d_transpose_1 Conv2DTranspose
batch_normalization_4 BatchNormalization
leaky_re_lu_5 LeakyReLU
conv2d_transpose_2 Conv2DTranspose
batch_normalization_5 BatchNormalization
leaky_re_lu_6 LeakyReLU
conv2d_transpose_3 Conv2DTranspose



How to train GANs?

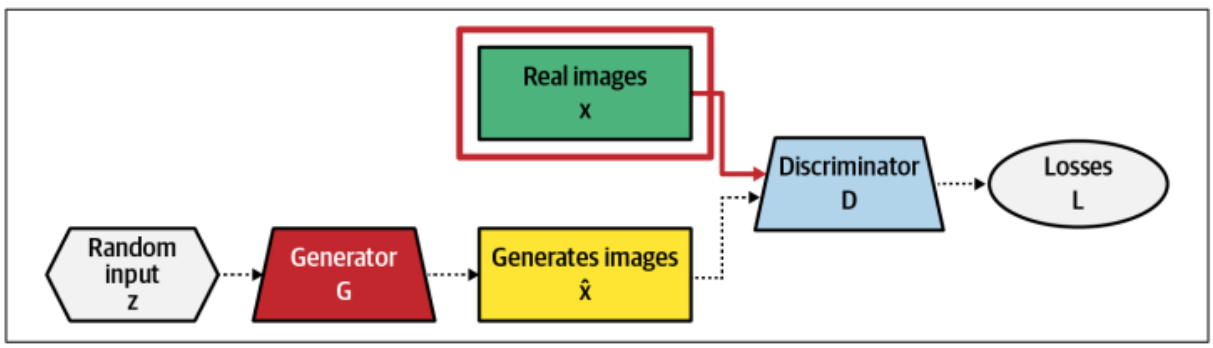
Discriminator Training

- Since the discriminator is a binary classification model, we can use the `binary cross entropy loss function` to quantify how well it is able to differentiate between real and generated images.
- The binary cross entropy loss is defined as:

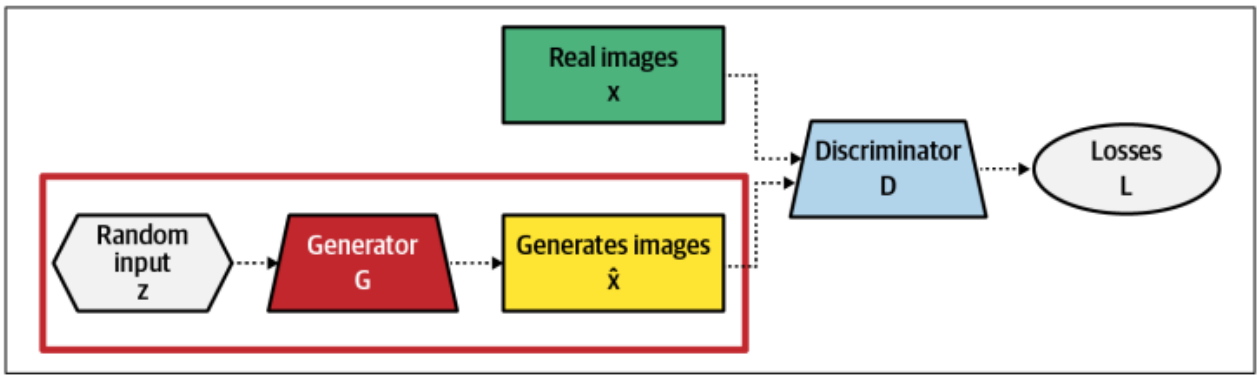
$$-\frac{1}{N}\sum_{i=1}^{N} y_i log(h_\theta(x_i)) + (1 - y_i) log(1 - h_\theta(x_i))$$

Here are the steps involved in training the discriminator:

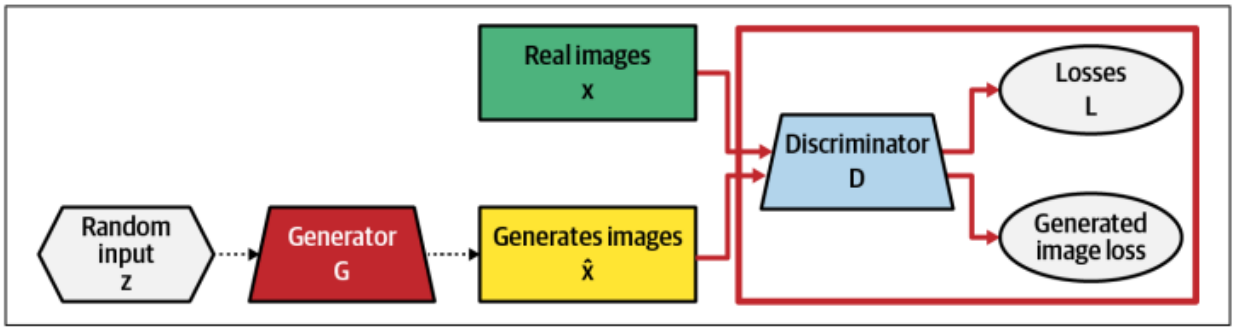
- We expect the discriminator to output 1 if the image was picked from the real Anime Face dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.



- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.



- Finally we add the two losses and use the overall loss to perform gradient descent to adjust the weights of the discriminator.



> *It's important to note that we don't change the weights of the generator model while training the discriminator (opt_d only affects the discriminator.parameters())*

Discriminator loss

> 1. Z: Noise vector (Dimension of the noise vector is a hyperparameter).
> 2. G(Z): Output of the generator when given the noise vector Z.
> 3. X: Real Training data.
> 4. D(G(Z)): Output of the discriminator when given fake generated data or G(Z).
> 5. D(X): Output of the discriminator when given real training data from X.

- The discriminator takes either X or G(Z). Note that the discriminator is nothing but a binary classifier so, we label D(X) as 1 and D(G(Z)) as 0.
- We want our discriminator to label all D(X) as 1 and all D(G(Z)) as 0. So how can we achieve this?
    1. Given, real image as input X , the predicted value is $D(X)$ and the label is 1. Therefore the loss function is,
        - $L(D(X), 1) = 1.logD(X) + (1 - 1).\, log(1 - D(X)) = logD(X)$
        - The discriminator should maximize Log(D(X)), and as Log is a monotonic function so log(D(X)) will automatically get maximized if the discriminator maximized D(X).
    2. Given, $G(Z)$ as input to Discriminator, the predicted value is $D(G(Z))$ and the label is 0. Therefore the loss function is,
        - $L(D(G(Z)), 0) = 0.logD(G(Z)) + (1 - 0).\, log(1 - D(G(Z))) = log(1 -$
        - The discriminator needs to maximize log(1 − D(G(Z))), which means it must have to minimize D(G(Z)).
- So, the loss function for the discriminator (for a single sample) becomes,

> $$max[logD(X) + log(1 - D(G(Z))]$$

- Now, the loss function of the discriminator over a batch is,

> $$max[E_{(X\sim P(X))}[logD(X)] + E_{Z\sim P(Z)}[log(1 - D(G(Z)))]]$$

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

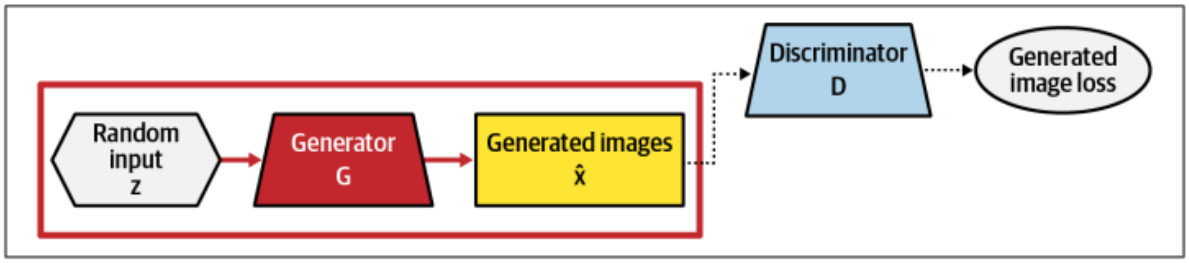
- **The following equation is maximized for training the discriminator**

> $$\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$$

> **Note:** that, D(X) and D(G(Z)) both are probability values and **both of them lie in between 0 and 1.**

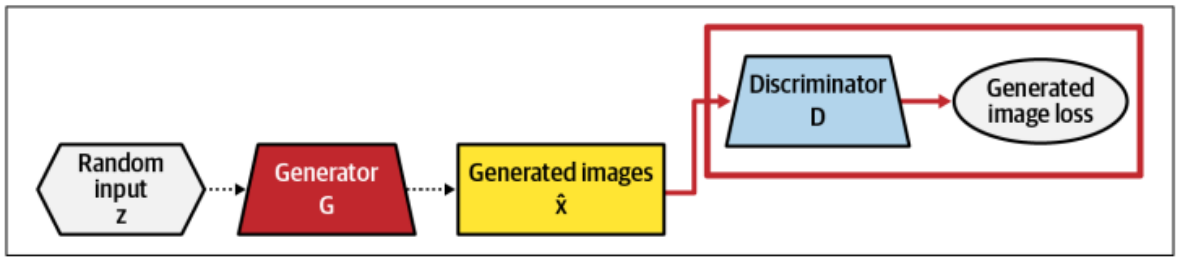
Generator Training

- Since the outputs of the generator are images, it's not obvious how we can train the generator.
- This is where we employ a rather elegant trick, which is to use the discriminator as a part of the loss function.
- Here's how it works:
  - We generate a batch of images using the generator, pass the into the discriminator.



  - We calculate the loss by setting the target labels to 1 i.e. real. We do this because the generator's objective is to "fool" the discriminator.

- We use the loss to perform gradient descent i.e. change the weights of the generator, so it gets better at generating real-like images to "fool" the discriminator.



Generator loss

- The Generator needs to fool the discriminator by generating images as real as possible. This means the generator should generate such G(Z) which if we pass through the discriminator will label is as 1.
- So, discriminator wants to make D(G(Z)) equal to 1 and generator wants to make D(G(Z)) equal to 0.Thus the loss function is,
  - $L(D(G(Z)), 0) = log(1 - D(G(Z)))$
- The above one is for just one sample. Over a batch, it will be,
  - $L(D(G(Z)), 0) = E_{Z \sim P(Z)}[log(1 - D(G(Z)))]$
- The generator will minimize the above loss function and to minimize the above the generator must maximize D(G(Z)). Now, it is very clear that **the discriminator wants to minimize D(G(Z)) and the generator wants to maximize D(G(Z)).**
- Understand that, the generator is never going to see any real data but for completeness, the generator loss function can be written as follows!

> $$min[E_{(X \sim P(X))}[logD(X)] + E_{(Z \sim P(Z))}[log(1 - D(G(Z)))]]$$

> **Note:** the generator has no control over the first term so the **generator will only minimize the second term.**

- **The following equation is minimized for training the generator:**
  - $\min_G \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(Z^i)))$.
- **Or, we can maximize the following for training the generator:**
  - $\max_G \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

Quiz-4

For training the GAN generator, which of the following is used :

1. $\max_G \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$

2. $\min_G \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(Z^i)))$

> (a) 1

> (b) 2

> (c) Either 1 or 2

> (d) None of the above

Ans (a) 1 .Although both 1 and 2 are same, but for training 1 is better for gradients to flow. Click for more details

---

Full Training Loop

- Assume that, D is the parameters of the Discriminator and G is the parameters of the generator. So, we can write the loss function as,
  - $\min_G \max_D [E_{(X \sim P(X))}[logD(X)] + E_{(Z \sim P(Z))}[log(1 - D(G(Z)))]]$
- **This means the discriminator parameters (defined by D) will maximize the loss function and the generator parameters (defined by G) will minimize the loss function.**
- The adversarial loss can be optimized by gradient descent. But while training a GAN **we do not train the generator and discriminator simultaneously**, while training the generator we freeze the discriminator and vice-versa!

Do you expect that GANs training is as smooth as we train other deep learning models?

- Since two deep learning models are in a constant fight. One model might overpower the other, which can degrade the quality of generated images
- For GANs, we expect the generator's loss to reduce over time, without the discriminator's loss getting too high.

Algorithm

for number of training iterations **do**

- Sample minibatch of *m* noise samples $\{z^1, ..., z^m\}$ from noise prior p_g(z).
- Sample minibatch of *m* examples $\{x^1, ..., x^m\}$ from data generating distribution p_{data}(x).
- Update the discriminator by ascending its stochastic gradient:
  - $\nabla_{\Theta_d} \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$

- Sample minibatch of *m* noise samples $\{z^1, \ldots\ldots, z^m\}$ from noise prior p_g(z).
- Update the generator by ascending its stochastic gradient:
  - $\nabla_{\Theta_g} \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

end for

```
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]

    def train_step(self, real_images):
        # Sample random points in the latent space
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.late

        # Decode them to fake images
        generated_images = self.generator(random_latent_vectors)

        # Combine them with real images
        combined_images = tf.concat([generated_images, real_images], axis=0)

        # # Assemble labels discriminating real from fake images
        # labels = tf.concat(
        #     [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0
        # )

        # Assemble labels discriminating real from fake images
        labels = tf.concat(
            [tf.zeros((batch_size, 1)), tf.ones((batch_size, 1))], axis=0
        )

        # Add random noise to the labels - important trick!
        # labels += 0.05 * tf.random.uniform(tf.shape(labels))

        # Train the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
```

```
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space
        random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.late

        # # Assemble labels that say "all real images"
        # misleading_labels = tf.zeros((batch_size, 1))
        # Assemble labels that say "all real images"
        misleading_labels = tf.ones((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            predictions = self.discriminator(self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weight

        # Update metrics
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```

The function save_img_batch will save the batch of images generated by GANS.

```
import matplotlib.gridspec as gridspec

def save_img_batch(img_batch,img_save_dir):
    img_batch.numpy()
    plt.figure(figsize=(4,4))
    gs1 = gridspec.GridSpec(4, 4)
    gs1.update(wspace=0, hspace=0)
    rand_indices = np.random.choice(img_batch.shape[0],16,replace=False)
    #print(rand_indices)
    for i in range(16):
        #plt.subplot(4, 4, i+1)
        ax1 = plt.subplot(gs1[i])
        ax1.set_aspect('equal')
        rand_index = rand_indices[i]
        image = img_batch[rand_index, :,:,:]
        fig = plt.imshow(image)
        plt.axis('off')
        fig.axes.get_xaxis().set_visible(False)
        fig.axes.get_yaxis().set_visible(False)
    plt.tight_layout()
```

```
    plt.savefig(img_save_dir,bbox_inches='tight',pad_inches=0)
    plt.show()
```

- GANS are very difficult to train. So we will be monitoring the output of the GANS generator after every epoch and save the batch of generated images

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=16, latent_dim=128, file_writer=None):
        self.num_img = num_img
        self.latent_dim = latent_dim
        self.file_writer = file_writer
        self.random_latent_vectors = tf.random.normal(shape=(self.num_img, 1, 1, se
        # Directory to save generated outputs after each epoch
        self.path = './generate_per_epoch'
        if not os.path.exists(self.path):
            os.mkdir(self.path)


    def on_epoch_end(self, epoch, logs=None):
        # random_latent_vectors = tf.random.normal(shape=(self.num_img, 1, 1, self.
        generated_images = self.model.generator(self.random_latent_vectors)
        generated_images = (generated_images*0.5 + 0.5)
        # generated_images.numpy()
        save_img_batch(generated_images, self.path + '/' + 'generated_img_%03d.png'

        # Convert to image and log
        with self.file_writer.as_default():
            tf.summary.image("Epoch end generated data", generated_images, step=epo

        # img = keras.preprocessing.image.array_to_img(generated_images)
        # img.save("generated_img_%03d.png" % (epoch))
        # for i in range(self.num_img):
        #     img = keras.preprocessing.image.array_to_img(generated_images[i])
        #     img.save("generated_img_%03d_%d.png" % (epoch, i))
```

- You need to save the logs in the tensorboard so that it can be viewed and loaded any time

```
training_log_dir = './logs/training_logs'
epoch_end_logdir = './logs/epoch_end_logs'
```

Quiz-5

Normaly while training a neural network we use Adam optimizer with a default $\beta 1$ = 0.9 and $\beta 2$ = 0.99 but while training GAN we set $\beta 1$ to 0.5 . What is the reason behind this:

> (a) We can use the default value of $\beta 1$

> (b) To help the discriminator distinguish between real and fake image

> (c) To achieve more stability in training the GAN

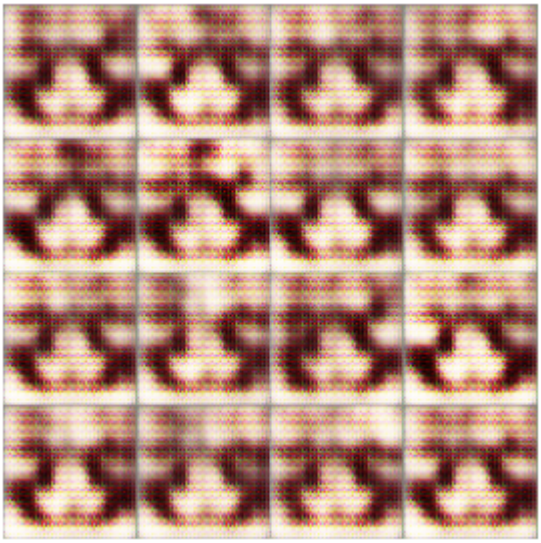
Ans: (c) To achieve more stability in training the GAN . Refer to section DETAILS OF ADVERSARIAL TRAINING in the paper

```
epochs = 60  #  try ~100 epochs
latent_dim = 128
# training_log_dir = "./drive/MyDrive/Datasets - DSML/IntroToGANs/logs/training_log

# Sets up a timestamped log directory.
# epoch_end_logdir = "./drive/MyDrive/Datasets - DSML/IntroToGANs/logs/epoch_end_lo
# Creates a file writer for the log directory.
file_writer = tf.summary.create_file_writer(epoch_end_logdir)

gan = GAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.00015, beta_1=0.5),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=training_log_dir, his
```

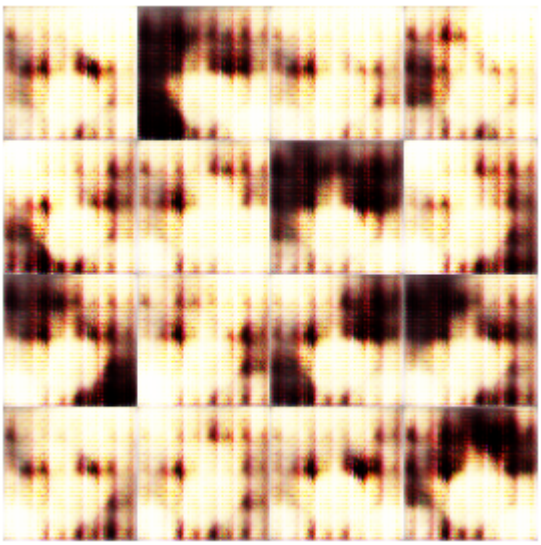
- Start training the GANS and observe the generated output after every epoch

```
%%time
history = gan.fit(
                    dataset, epochs=epochs,
                    callbacks=[GANMonitor(num_img=16, latent_dim=latent_dim, file_w
                               tensorboard_callback])
```

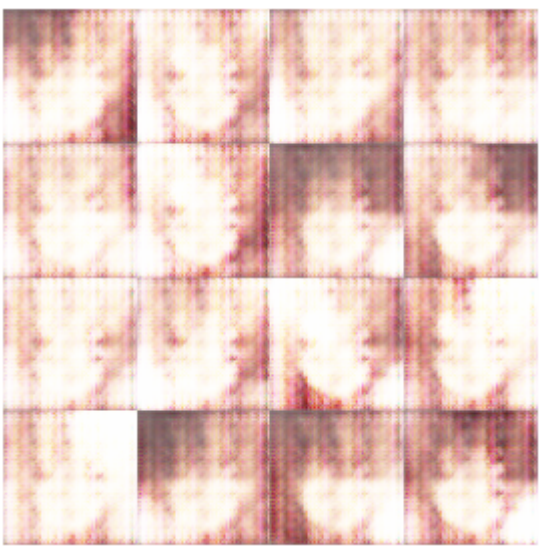
```
Epoch 1/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2191 - g_loss
```



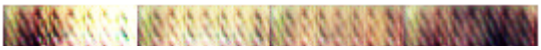
```
249/249 [==============================] - 107s 357ms/step - d_loss: 0.2191
Epoch 2/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2485 - g_loss
```



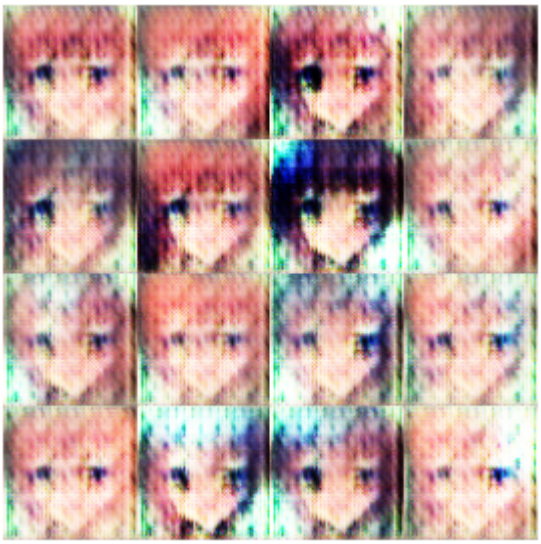
```
249/249 [==============================] - 92s 367ms/step - d_loss: 0.2485 -
Epoch 3/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2252 - g_loss
```



```
249/249 [==============================] - 94s 373ms/step - d_loss: 0.2252 -
Epoch 4/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1893 - g_loss
```

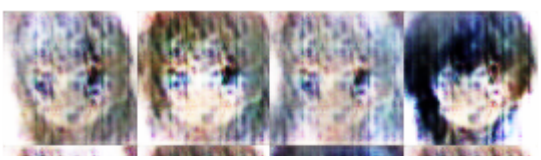


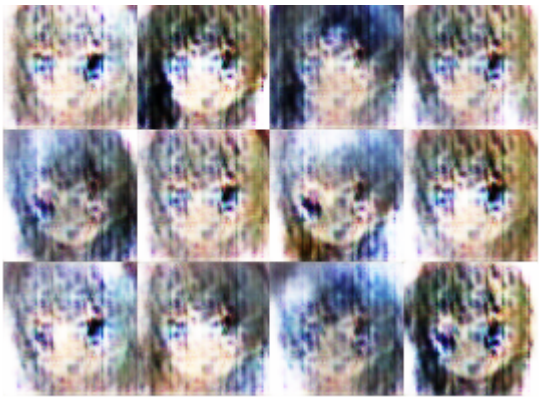
```
249/249 [==============================] - 94s 374ms/step - d_loss: 0.1893 -
Epoch 5/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1768 - g_loss
```



```
249/249 [==============================] - 96s 385ms/step - d_loss: 0.1768 -
Epoch 6/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1785 - g_loss
```

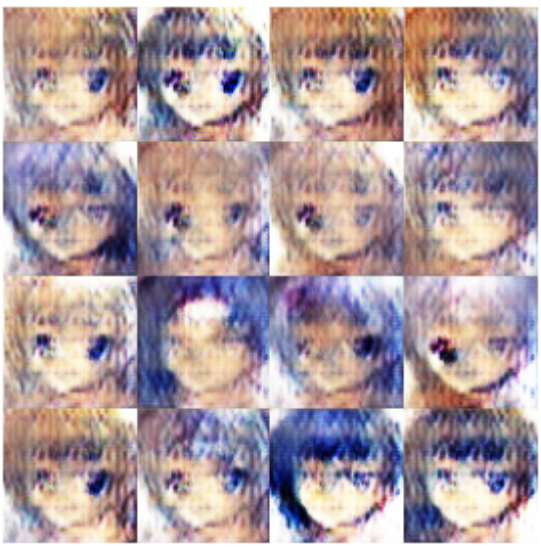
```
249/249 [==============================] - 96s 385ms/step - d_loss: 0.1785 -
Epoch 7/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1689 - g_loss
```



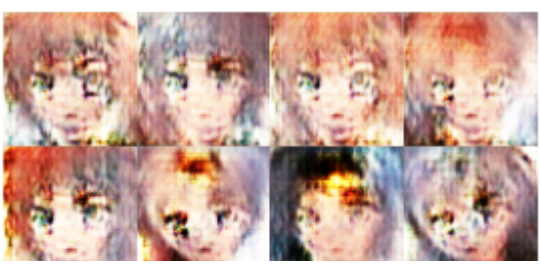


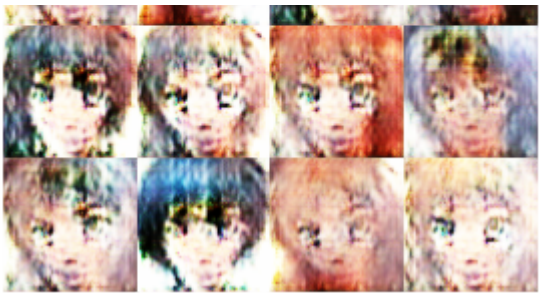
```
249/249 [==============================] - 95s 380ms/step - d_loss: 0.1689 -
Epoch 8/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1653 - g_loss
```

```
249/249 [==============================] - 97s 386ms/step - d_loss: 0.1653 -
Epoch 9/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1615 - g_loss
```



```
249/249 [==============================] - 95s 379ms/step - d_loss: 0.1615 -
Epoch 10/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1686 - g_loss
```

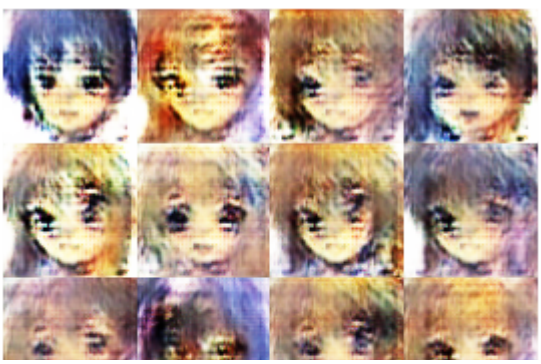




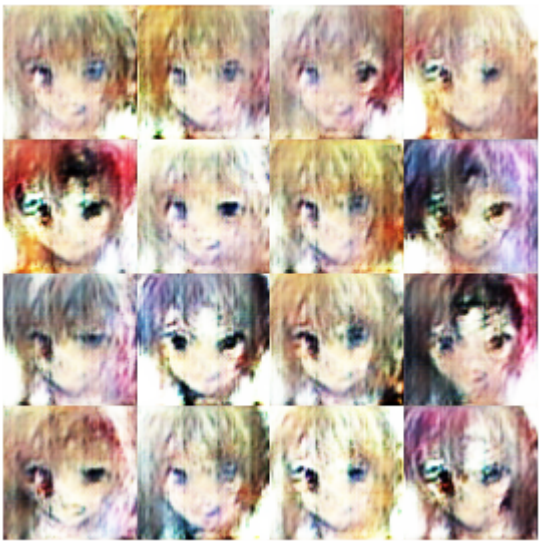
```
249/249 [==============================] - 95s 378ms/step - d_loss: 0.1686 -
Epoch 11/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1663 - g_loss
```

```
249/249 [==============================] - 95s 377ms/step - d_loss: 0.1663 -
Epoch 12/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1687 - g_loss
```

```
249/249 [==============================] - 95s 379ms/step - d_loss: 0.1687 -
Epoch 13/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1903 - g_loss
```

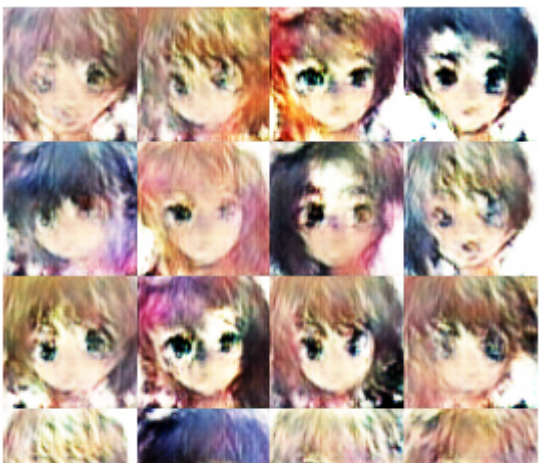


```
249/249 [==============================] - 95s 377ms/step - d_loss: 0.1903 -
Epoch 14/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2062 - g_loss
```



```
249/249 [==============================] - 97s 386ms/step - d_loss: 0.2062 -
Epoch 15/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1711 - g_loss
```

```
249/249 [==============================] - 97s 387ms/step - d_loss: 0.1711 -
Epoch 16/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1705 - g_loss
```



```
249/249 [==============================] - 97s 385ms/step - d_loss: 0.1705 -
Epoch 17/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1711 - g_loss
```

```
249/249 [==============================] - 97s 385ms/step - d_loss: 0.1711 -
Epoch 18/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1749 - g_loss
```

```
249/249 [==============================] - 95s 380ms/step - d_loss: 0.1749 -
Epoch 19/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1989 - g_loss
```

```
249/249 [==============================] - 95s 378ms/step - d_loss: 0.1989 -
Epoch 20/60
249/249 [==============================] - ETA: 0s - d_loss: 0.1988 - g_loss
```

```
249/249 [==============================] - 95s 377ms/step - d_loss: 0.1988 -
Epoch 21/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2158 - g_loss
```

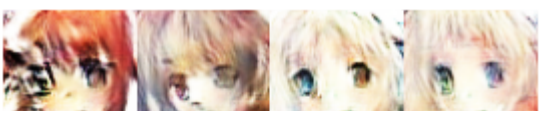
```
249/249 [==============================] - 95s 377ms/step - d_loss: 0.2158 -
Epoch 22/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2001 - g_loss
```

```
249/249 [==============================] - 95s 377ms/step - d_loss: 0.2001 -
Epoch 23/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2112 - g_loss
```

```
249/249 [==============================] - 97s 388ms/step - d_loss: 0.2112 -
Epoch 24/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2190 - g_loss
```

```
249/249 [==============================] - 97s 385ms/step - d_loss: 0.2190 -
Epoch 25/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2415 - g_loss
```

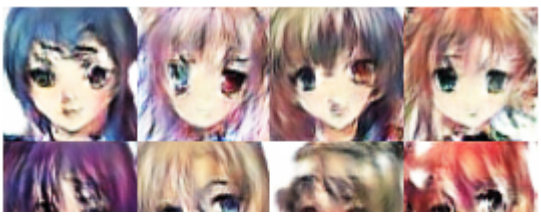
```
249/249 [==============================] - 96s 381ms/step - d_loss: 0.2415 -
Epoch 26/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2186 - g_loss
```

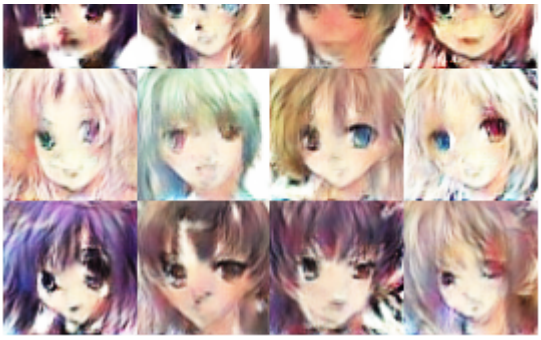


```
249/249 [==============================] - 96s 382ms/step - d_loss: 0.2186 -
Epoch 27/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2248 - g_loss
```

```
249/249 [==============================] - 99s 393ms/step - d_loss: 0.2248 -
Epoch 28/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2419 - g_loss
```

```
249/249 [==============================] - 98s 388ms/step - d_loss: 0.2419 -
Epoch 29/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2350 - g_loss
```





```
249/249 [==============================] - 97s 386ms/step - d_loss: 0.2350 -
Epoch 30/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2416 - g_loss
```

```
249/249 [==============================] - 96s 382ms/step - d_loss: 0.2416 -
Epoch 31/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2457 - g_loss
```

```
249/249 [==============================] - 95s 380ms/step - d_loss: 0.2457 -
Epoch 32/60
249/249 [==============================] - ETA: 0s - d_loss: 0.2595 - g_loss
```