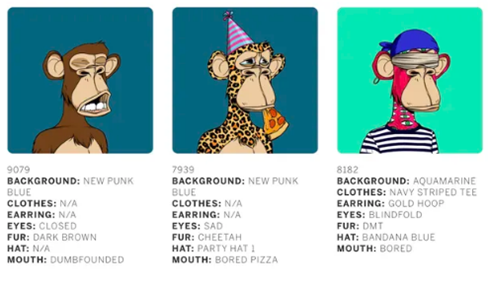
**PGGANs for NFTs**

**Introduction**

Lately, there has been a lot of buzz surrounding NFTs (non-fungible tokens), which is essentially a fancy acronym for electronic art. This art can sell for millions of dollars, including some well-known Ape NFTs selling for over $24 million.



Example Ape NFTs that sold at auction

What makes NFTs so unique? Well, there are various reasons. For a buyer, it’s the concept of owning an original copy of the artwork that may or may not go viral on the internet that could drive up its value, all while having peace of mind knowing that there is built-in authentication and certification using similar blockchain software as bitcoin. For artists, this gives them another place other than an auction house or a gallery to sell their artwork, increasing their chances of having their work recognized.

Since you most likely can’t afford to purchase a well-known NFT, you’re probably wondering “why should I even care for electronic art that can sell for as much as classic artworks you’d find in museums at the Louvre?”. This is where artificial intelligence comes in. If you’re not a talented artist and don’t have the money to get in on the action, you can build your own neural network model (for free) that can generate artwork for you.

Now that we’ve caught your attention, you’re probably wondering how this works. Well, you can use a variation of a GAN neural network called a PGGAN, which we’ll now discuss in more detail.

**What is a PGGAN and How is it Different From a GAN?**

GANs (generative adversarial networks) are a type of neural network model that generates unique images that are trained on a set of other images. It has two components — a generator (used to create the image) and a discriminator (used to determine if the image is “real”). Now, if you’re unfamiliar with the nitty-gritty of how GANs work, we suggest you check out one of our previous [articles](https://medium.com/@siraj.hatoum/gan-hyperparameter-tuning-with-keras-tuner-81e00ad1d6be), which focuses on improving GANs based on this [article](https://medium.com/dc-gan/how-to-generate-realistic-pictures-with-deep-convolutional-gans-328beb40c14), which creates realistic images that don’t exist.

Although this is super cool, there are some drawbacks to this type of model. The first is that it is difficult for GANs to generate high-resolution images because it needs to learn not only how to generate a large image, but also the small details. This can be a problem when you’re creating your NFT because you want it to be detailed and appealing to look at.

The second problem with the GAN is that trying to produce such a large, high-quality image will require a lot of memory. Since you’re most likely using your laptop, this will significantly slow down the rate at which you’re capable of generating your NFTs (further slowing down your path to becoming both an artistic genius and a millionaire).

To counter this memory problem, the third issue comes with trying to reduce the batch size for model training. Although reducing the batch size reduces the memory required for training, this may lead to the training process becoming unstable, further producing low-quality images. In the end, nobody wants to buy a pixelated image.

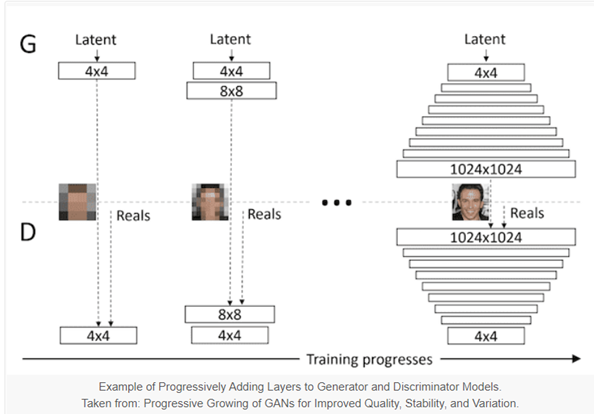
This is where PGGANs come to the rescue on your money-making journey. PGGANs, otherwise known as progressive growing generative adversarial networks, solve these problems by using an extension of the original GAN, that can produce large high quality and high-resolution images by progressively adding layers to the input image. The next section will discuss this framework in more detail.

**PGGAN Framework**

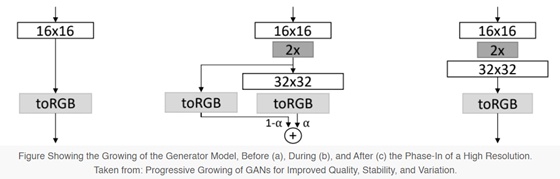
Like a regular GAN, PGGANs have a generator and a discriminator. When training begins, you first start with a single convolution layer in the discriminator and the generator. When looking at the framework example below, the first column starts with the first layer having a 4x4 image. To train at this scale, you downsample the training images (also in size 4x4) and use them for training.

After a few iterations, you then introduce an additional convolutional layer in the generator and discriminator, as can be seen in the image in the second column, where the added discriminator is an 8x8 image.

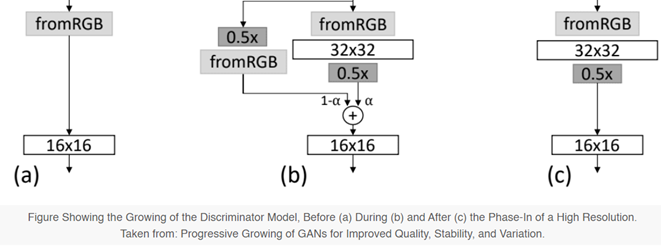
You can continue this progressive addition of convolutional layers until you have your desired output image (last column in the image below). By growing the model progressively, it’s capable of first learning the high-level details that come with the small image, followed by more precise details that come with the larger higher resolution image.



In particular, the higher resolution layers are progressively phased into both the discriminator and generator through a weighted sum of two neural network pathways. In the case of the generator, the first pathway connects a new neural network block preceded by an upsampling layer to the higher resolution pathway and the second pathway directly connects the upsampling layer to the higher resolution output layer. During training, the weight is shifted from the second pathway through the first pathway by increasing the weighted alpha to be closer to 1. At the end of the training, the second pathway is removed.



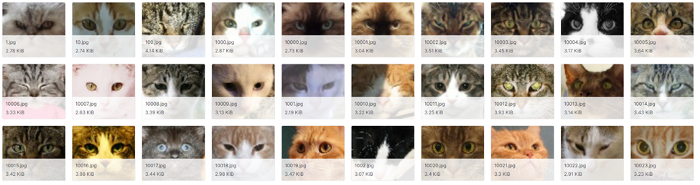
The discriminator has similar pathways and processes as the generator. The only difference is that the first pathway adds an input layer that supports higher resolution images along with a new neural network block and a downsampling output layer to support the rest of the network. And the second pathway directly downsamples the images from the new high-resolution input layer and connects this downsampling layer to the rest of the old network. Similarly, during training more weight is given to the first pathway until the second pathway is completely removed.



Now that you have a better understanding of how the PGGAN works, we can move forward with an example of how to code this model.

**Preprocessing**

In honour of Nyan cat (which sold for almost $600k), we will be using a dataset from [Kaggle](https://www.kaggle.com/datasets/spandan2/cats-faces-64x64-for-generative-models?fbclid=IwAR2s0__nzs6pF2D8FAzzzORDtjtAYAtWSW_rCEcxwouKMRbuveRNbOQVwFU) that contains 15.7k images of cat faces of size 64x64 to demonstrate how the PGGAN works.



Prior to diving into the model, there are some “usual suspect” steps that we need to cover first, which are setting up and cleaning the data. First, we import the necessary python packages.

# importing librariesimport pandas as pd   
import numpy as np  
import matplotlib.pyplot as plt  
from tensorflow import keras   
from PIL import Image  
import os  
from PIL import ImageFile  
ImageFile.LOAD\_TRUNCATED\_IMAGES = True  
from google.colab import files  
from math import sqrt  
from numpy import zeros  
from numpy import ones  
from skimage.transform import resize

Next, we create two functions to convert our dataset into a 3-dimensional (for each RGB channel) NumPy array of pixels.

# load an image as an rgb numpy array

def load\_image(filename):

  # load image from file

  image = Image.open(filename)

  # convert to RGB, if needed

  image = image.convert('RGB')

  # convert to array

  pixels = np.asarray(image)

  return pixels

# convert dataset into rgb numpy array

def dataset\_creation(directory):

  art = []

  for filename in os.listdir(directory):

    # load the image

    pixels = load\_image(directory + filename)

    # store

    art.append(pixels)

    print(len(art))

  return np.asarray(art)

data = dataset\_creation(directory)

cleaned\_data = np.savez\_compressed('cleaned\_data.npz', data)

And last, but not least, we normalize the values to be between -1 and 1. Normalization is important, especially within neural networks, to ensure that the input images are all within a similar scale for model training.

# Normalize the dataset to be between -1 and 1

def load\_real\_samples(filename):

  # load dataset

  data = np.load(filename)

  # extract numpy array

  X = data['arr\_0']

  # convert from ints to floats

  X = X.astype('float32')

  # scale from [0,255] to [-1,1]

  X = (X - 127.5) / 127.5

  return X

training\_set = load\_real\_samples('cleaned\_data.npz')

# ****Building the Model****

Now that the data has been processed, we can move forward with the model. Before diving into defining the functions needed to create and grow the discriminator and generator, we need to define custom layers that are key to PGGANs.

We create a WeightedSum layer by subclassing the Add layer. This layer is used to phase in the higher resolution layers into both the lower resolution discriminator and generator.

class WeightedSum(keras.layers.Add):

  # init with default value

  def \_\_init\_\_(self, alpha=0.0, \*\*kwargs):

    super(WeightedSum, self).\_\_init\_\_(\*\*kwargs)

    self.alpha = keras.backend.variable(alpha, name='ws\_alpha')

  # output a weighted sum of inputs

  def \_merge\_function(self, inputs):

    # only supports a weighted sum of two inputs

    assert (len(inputs) == 2)

    # ((1-a) \* input1) + (a \* input2)

    output = ((1.0 - self.alpha) \* inputs[0]) + (self.alpha \* inputs[1])

    return output

We create a MinibatchStdev layer by subclassing the Layer function. This layer is applied to the output block of the discriminator and provides summary statistics of the batch of activations. It essentially helps the generator create a batch of samples with realistic statistics through backpropagation.

class MinibatchStdev(keras.layers.Layer):

  # initialize the layer

  def \_\_init\_\_(self, \*\*kwargs):

    super(MinibatchStdev, self).\_\_init\_\_(\*\*kwargs)

  # perform the operation

  def call(self, inputs):

    # calculate the mean value for each pixel across channels

    mean = keras.backend.mean(inputs, axis=0, keepdims=True)

    # calculate the squared differences between pixel values and mean

    squ\_diffs = keras.backend.square(inputs - mean)

    # calculate the average of the squared differences (variance)

    mean\_sq\_diff = keras.backend.mean(squ\_diffs, axis=0, keepdims=True)

    # add a small value to avoid a blow-up when we calculate stdev

    mean\_sq\_diff += 1e-8

    # square root of the variance (stdev)

    stdev = keras.backend.sqrt(mean\_sq\_diff)

    # calculate the mean standard deviation across each pixel coord

    mean\_pix = keras.backend.mean(stdev, keepdims=True)

    # scale this up to be the size of one input feature map for each sample

    shape = keras.backend.shape(inputs)

    output = keras.backend.tile(mean\_pix, (shape[0], shape[1], shape[2], 1))

    # concatenate with the output

    combined = keras.backend.concatenate([inputs, output], axis=-1)

    return combined

  # define the output shape of the layer

  def compute\_output\_shape(self, input\_shape):

    # create a copy of the input shape as a list

    input\_shape = list(input\_shape)

    # add one to the channel dimension (assume channels-last)

    input\_shape[-1] += 1

    # convert list to a tuple

    return tuple(input\_shape)

Next, PGGANs do not use Batch Normalization. Instead, they use Pixel Normalization which normalizes each pixel in activation maps to unit length. Again, this layer is created by subclassing the layer class.

class PixelNormalization(keras.layers.Layer):

  # initialize the layer

  def \_\_init\_\_(self, \*\*kwargs):

    super(PixelNormalization, self).\_\_init\_\_(\*\*kwargs)

  # perform the operation

  def call(self, inputs):

    # calculate square pixel values

    values = inputs\*\*2.0

    # calculate the mean pixel values

    mean\_values = keras.backend.mean(values, axis=-1, keepdims=True)

    # ensure the mean is not zero

    mean\_values += 1.0e-8

    # calculate the sqrt of the mean squared value (L2 norm)

    l2 = keras.backend.sqrt(mean\_values)

    # normalize values by the l2 norm

    normalized = inputs / l2/

    return normalized

  # define the output shape of the layer

  def compute\_output\_shape(self, input\_shape):

    return input\_shape

Lastly, we implement Wasserstein loss in our model as our loss function and we define a function for it.

# calculate wasserstein loss

def wasserstein\_loss(y\_true, y\_pred):

  return keras.backend.mean(y\_true \* y\_pred)

Now we are finally getting into building the framework for the generator and discriminator. There are many ways both frameworks can be set up. In our case, we are defining every updated iteration of the models. This means that we build a different Keras model for both the discriminator and generator at every growth step of our PGGAN. These models will share the same weights and layers.

The first part is the generator framework, which will be used to generate new images of cats that do not exist based on random noise. The framework is composed of two functions; The define\_generator function creates a base model (4x4) that is updated through the add\_generator\_block function. This latter function defines two models for each resolution. A “phasing in” model with the two pathways and weighted sum layer and a fully updated model with the same weights and layers but without the old layer and weighted sum layer. A fully updated model is required for fine-tuning the weights before fading into the next higher resolution layers. The resolution is increased by increasing the resolution of the output layer.

1. *Updating the model: add\_generator\_block function*

def add\_generator\_block(old\_model):

  # weight initialization

  init = keras.initializers.RandomNormal(stddev=0.02)

  # weight constraint

  const = keras.constraints.max\_norm(1.0)

  # get the end of the last block

  block\_end = old\_model.layers[-2].output

  # upsample, and define new block

  upsampling = keras.layers.UpSampling2D()(block\_end)

  g = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(upsampling)

  g = PixelNormalization()(g)

  g = keras.layers.LeakyReLU(alpha=0.2)(g)

  g = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(g)

  g = PixelNormalization()(g)

  g = keras.layers.LeakyReLU(alpha=0.2)(g)

  # add new output layer

  out\_image = keras.layers.Conv2D(3, (1,1), padding='same', kernel\_initializer=init, kernel\_constraint=const)(g)

  # define model

  model1 = keras.Model(old\_model.input, out\_image)

  # get the output layer from old model

  out\_old = old\_model.layers[-1]

  # connect the upsampling to the old output layer

  out\_image2 = out\_old(upsampling)

  # define new output image as the weighted sum of the old and new models

  merged = WeightedSum()([out\_image2, out\_image])

  # define model

  model2 = keras.Model(old\_model.input, merged)

  return [model1, model2]

1. Defining base model (4x4) and automating the update of the model for higher resolution: add\_generator\_block function

# define generator models

def define\_generator(latent\_dim, n\_blocks, in\_dim=4):

  # weight initialization

  init = keras.initializers.RandomNormal(stddev=0.02)

  # weight constraint

  const = keras.constraints.max\_norm(1.0)

  model\_list = list()

  # base model latent input

  in\_latent = keras.layers.Input(shape=(latent\_dim,))

  # linear scale up to activation maps

  g  = keras.layers.Dense(128 \* in\_dim \* in\_dim, kernel\_initializer=init, kernel\_constraint=const)(in\_latent)

  g = keras.layers.Reshape((in\_dim, in\_dim, 128))(g)

  # conv 4x4, input block

  g = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(g)

  g = PixelNormalization()(g)

  g = keras.layers.LeakyReLU(alpha=0.2)(g)

  # conv 3x3

  g = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(g)

  g = PixelNormalization()(g)

  g = keras.layers.LeakyReLU(alpha=0.2)(g)

  # conv 1x1, output block

  out\_image = keras.layers.Conv2D(3, (1,1), padding='same', kernel\_initializer=init, kernel\_constraint=const)(g)

  # define model

  model = keras.Model(in\_latent, out\_image)

  # store model

  model\_list.append([model, model])

  # create submodels

  for i in range(1, n\_blocks):

    # get prior model without the fade-on

    old\_model = model\_list[i - 1][0]

    # create new model for next resolution

    models = add\_generator\_block(old\_model)

    # store model

    model\_list.append(models)

  return model\_list

The second part is the discriminator framework, where we created two similar functions to the generator function. The define\_discriminator function defines the base model (4x4 resolution) and recursively updates the model for higher resolution through the add\_discriminator\_block function. The resolution is increased by updating the input layer of the discriminator.

1. *Updating the model: add\_discriminator\_block*

def add\_discriminator\_block(old\_model, n\_input\_layers=3):

  # weight initialization

  init = keras.initializers.RandomNormal(stddev=0.02)

  # weight constraint

  const = keras.constraints.max\_norm(1.0)

  # get shape of existing model

  in\_shape = list(old\_model.input.shape)

  # define new input shape as double the size

  input\_shape = (in\_shape[-2]\*2, in\_shape[-2]\*2, in\_shape[-1])

  in\_image = keras.layers.Input(shape=input\_shape)

  # define new input processing layer

  d = keras.layers.Conv2D(128, (1,1), padding='same', kernel\_initializer=init, kernel\_constraint=const)(in\_image)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  # define new block

  d = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(d)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  d = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(d)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  d = keras.layers.AveragePooling2D()(d)

  block\_new = d

  # skip the input, 1x1 and activation for the old model

  for i in range(n\_input\_layers, len(old\_model.layers)):

    d = old\_model.layers[i](d)

  # define straight-through model

  model1 = keras.Model(in\_image, d)

  # compile model

  model1.compile(loss=wasserstein\_loss, optimizer=keras.optimizers.Adam(lr=0.00001, beta\_1=0, beta\_2=0.99, epsilon=10e-8))

  # downsample the new larger image

  downsample = keras.layers.AveragePooling2D()(in\_image)

  # connect old input processing to downsampled new input

  block\_old = old\_model.layers[1](downsample)

  block\_old = old\_model.layers[2](block\_old)

  # fade in output of old model input layer with new input

  d = WeightedSum()([block\_old, block\_new])

  # skip the input, 1x1 and activation for the old model

  for i in range(n\_input\_layers, len(old\_model.layers)):

    d = old\_model.layers[i](d)

  # define straight-through model

  model2 = keras.Model(in\_image, d)

  # compile model

  model2.compile(loss=wasserstein\_loss, optimizer=keras.optimizers.Adam(lr=0.00001, beta\_1=0, beta\_2=0.99, epsilon=10e-8))

  return [model1, model2]

1. Defining base model (4x4) and automating the update of the model for higher resolution: define\_discriminator function

# define the discriminator models for each image resolution

def define\_discriminator(n\_blocks, input\_shape=(4,4,3)):

  # weight initialization

  init = keras.initializers.RandomNormal(stddev=0.02)

  # weight constraint

  const = keras.constraints.max\_norm(1.0)

  model\_list = list()

  # base model input

  in\_image = keras.layers.Input(shape=input\_shape)

  # conv 1x1

  d = keras.layers.Conv2D(128, (1,1), padding='same', kernel\_initializer=init, kernel\_constraint=const)(in\_image)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  # conv 3x3 (output block)

  d = MinibatchStdev()(d)

  d = keras.layers.Conv2D(128, (3,3), padding='same', kernel\_initializer=init, kernel\_constraint=const)(d)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  # conv 4x4

  d = keras.layers.Conv2D(128, (4,4), padding='same', kernel\_initializer=init, kernel\_constraint=const)(d)

  d = keras.layers.LeakyReLU(alpha=0.2)(d)

  # dense output layer

  d = keras.layers.Flatten()(d)

  out\_class = keras.layers.Dense(1)(d)

  # define model

  model = keras.Model(in\_image, out\_class)

  # compile model

  model.compile(loss=wasserstein\_loss, optimizer=keras.optimizers.Adam(lr=0.00001, beta\_1=0, beta\_2=0.99, epsilon=10e-8))

  # store model

  model\_list.append([model, model])

  # create submodels

  for i in range(1, n\_blocks):

    # get prior model without the fade-on

    old\_model = model\_list[i - 1][0]

    # create new model for next resolution

    models = add\_discriminator\_block(old\_model)

    # store model

    model\_list.append(models)

  return model\_list

GANs are trained recursively. In particular, the generator is only trained after the discriminator has finished training at every epoch. When the generator undergoes training, fake images are generated and passed on to the discriminator as “real” images. A loss is then calculated when the discriminator classifies these images are fake and backpropagated to update the weights of the generator. During this phase, the discriminator’s weights are not trainable. This process is implemented using the define\_composite function below.

# define composite models for training generators via discriminators

def define\_composite(discriminators, generators):

  model\_list = list()

  # create composite models

  for i in range(len(discriminators)):

    g\_models, d\_models = generators[i], discriminators[i]

    # straight-through model

    d\_models[0].trainable = False

    model1 = keras.Sequential()

    model1.add(g\_models[0])

    model1.add(d\_models[0])

    model1.compile(loss=wasserstein\_loss, optimizer=keras.optimizers.Adam(lr=0.00001, beta\_1=0, beta\_2=0.99, epsilon=10e-8))

    # fade-in model

    d\_models[1].trainable = False

    model2 = keras.Sequential()

    model2.add(g\_models[1])

    model2.add(d\_models[1])

    model2.compile(loss=wasserstein\_loss, optimizer=keras.optimizers.Adam(lr=0.00001, beta\_1=0, beta\_2=0.99, epsilon=10e-8))

    # store

    model\_list.append([model1, model2])

  return model\_list

# ****Model Training****

The last step is to train our PGGAN. We created various functions to be used in training. The first function allows us to randomly sample pictures from our dataset which will be useful to train the discriminator.

def generate\_real\_samples(dataset, n\_samples):

  # choose random instances

  ix = np.random.randint(0, dataset.shape[0], n\_samples)

  # select images

  X = dataset[ix]

  # generate class labels

  y = ones((n\_samples, 1))

  return X, y

The second function generates points in latent space as input for the generator function we created previously. This will be useful when training both the discriminator and generator.

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

  # generate points in the latent space

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

  # reshape into a batch of inputs for the network

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

  return x\_input

The third function takes the generator function and a latent point generated by the previously defined function to generate fake images. A label of 1 (real image) is associated with the generated images for when the generator is being trained.

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

  # generate points in latent space

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

  # predict outputs

  X = generator.predict(x\_input)

  # create class labels

  y = -ones((n\_samples, 1))

  return X, y

The fourth function is used to update the alpha value on each instance of the weighted sum layer we defined earlier.

# update the alpha value on each instance of WeightedSum

def update\_fadein(models, step, n\_steps):

  # calculate current alpha (linear from 0 to 1)

  alpha = step / float(n\_steps - 1)

  # update the alpha for each model

  for model in models:

    for layer in model.layers:

      if isinstance(layer, WeightedSum):

        keras.backend.set\_value(layer.alpha, alpha)

The fifth function is used to scale images to a preferred size. This function will be used to change the resolution of our dataset during training to be compatible with each stage of the discriminator and generator.

def scale\_dataset(images, new\_shape):

  images\_list = list()

  for image in images:

    # resize with nearest neighbor interpolation

    new\_image = resize(image, new\_shape, 0)

    # store

    images\_list.append(new\_image)

  return np.asarray(images\_list)

The sixth function is used to save the generator models and training results to specified paths.

images\_dir = '/content/gdrive/MyDrive/Colab Notebooks/PGGAN/generated\_images\_4'

models\_dir = '/content/gdrive/MyDrive/Colab Notebooks/PGGAN/generated\_models\_4'

def summarize\_performance(status, g\_model, latent\_dim, n\_samples=25):

  # devise name

  gen\_shape = g\_model.output\_shape

  name = '%03dx%03d-%s' % (gen\_shape[1], gen\_shape[2], status)

  # generate images

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

  # normalize pixel values to the range [0,1]

  X = (X - X.min()) / (X.max() - X.min())

  # plot real images

  square = int(sqrt(n\_samples))

  for i in range(n\_samples):

    plt.subplot(square, square, 1 + i)

    plt.axis('off')

    plt.imshow(X[i])

  # save plot to file

  filename1 = 'plot\_%s.png' % (name)

  plt.savefig(f"{images\_dir}/"+filename1)

  plt.close()

  # save the generator model

  filename2 = 'model\_%s.h5' % (name)

  g\_model.save(f"{models\_dir}/"+filename2)

  # shutil.copy(,models\_dir)

  print('>Saved: %s and %s' % (filename1, filename2))

The seventh and eighth functions are used to define the training process, including training the generator and the discriminator. The train\_epochs function defines the training procedure for each epoch iteration. This first implies randomly sampling a batch of real images and generating a batch of fake images to train the discriminator. It then involves using the generated fake images to train the generator through the composite model. The function also prints the loss of the discriminator on the real and fake data as well as the loss of the generator respectively.

The train function runs the train\_epochs function for a specified number of epochs and growth stages. In particular, the function first trains the base model then the higher resolution models by (1) scaling the dataset to the preferred size then (2) training and saving (only generator) the associated “phasing in” model and fine-tuned models.

# defining training functions

def train\_epochs(g\_model, d\_model, gan\_model, dataset, n\_epochs, n\_batch, latent\_dim, fadein=False):

  # calculate the number of batches per training epoch

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

  # calculate the number of training iterations

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

  # calculate the size of half a batch of samples

  half\_batch = int(n\_batch / 2)

  # manually enumerate epochs

  for i in range(n\_steps):

    # update alpha for all WeightedSum layers when fading in new blocks

    if fadein:

      update\_fadein([g\_model, d\_model, gan\_model], i, n\_steps)

    # prepare real and fake samples

    X\_real, y\_real = generate\_real\_samples(dataset, half\_batch)

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

    # update discriminator model

    d\_loss1 = d\_model.train\_on\_batch(X\_real, y\_real)

    d\_loss2 = d\_model.train\_on\_batch(X\_fake, y\_fake)

    # update the generator via the discriminator's error

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

    y\_real2 = ones((n\_batch, 1))

    g\_loss = gan\_model.train\_on\_batch(z\_input, y\_real2)

    # summarize loss on this batch

    print('>%d, d1=%.3f, d2=%.3f g=%.3f' % (i+1, d\_loss1, d\_loss2, g\_loss))

# train the generator and discriminator

def train(g\_models, d\_models, gan\_models, dataset, latent\_dim, e\_norm, e\_fadein, n\_batch):

  # fit the baseline model

  g\_normal, d\_normal, gan\_normal = g\_models[0][0], d\_models[0][0], gan\_models[0][0]

  # scale dataset to appropriate size

  gen\_shape = g\_normal.output\_shape

  scaled\_data = scale\_dataset(dataset, gen\_shape[1:])

  print('Scaled Data', scaled\_data.shape)

  # train normal or straight-through models

  train\_epochs(g\_normal, d\_normal, gan\_normal, scaled\_data, e\_norm[0], n\_batch[0],latent\_dim)

  summarize\_performance('tuned', g\_normal, latent\_dim)

  # process each level of growth

  for i in range(1, len(g\_models)):

    # retrieve models for this level of growth

    [g\_normal, g\_fadein] = g\_models[i]

    [d\_normal, d\_fadein] = d\_models[i]

    [gan\_normal, gan\_fadein] = gan\_models[i]

    # scale dataset to appropriate size

    gen\_shape = g\_normal.output\_shape

    scaled\_data = scale\_dataset(dataset, gen\_shape[1:])

    print('Scaled Data', scaled\_data.shape)

    # train fade-in models for next level of growth

    train\_epochs(g\_fadein, d\_fadein, gan\_fadein, scaled\_data, e\_fadein[i], n\_batch[i],latent\_dim, True)

    summarize\_performance('faded', g\_fadein, latent\_dim)

    # train normal or straight-through models

    train\_epochs(g\_normal, d\_normal, gan\_normal, scaled\_data, e\_norm[i], n\_batch[i], latent\_dim)

    summarize\_performance('tuned', g\_normal, latent\_dim)

For our scenario, we specified 5 growth phases (five resolution steps from 4x4 to 64x64), with a latent dimension of 100, a descending list of a specified number of batches, and an ascending list of a number of epochs. We decided to have smaller batches as resolution increases to speed up training. To compensate, we ensured to have more epochs for training higher resolution models.

# number of growth phases, e.g. 5 == [4, 8, 16, 32, 64]

n\_blocks = 5

# size of the latent space

latent\_dim = 100

# # define models

d\_models = define\_discriminator(n\_blocks)

# define models

g\_models = define\_generator(latent\_dim, n\_blocks)

# define composite models

gan\_models = define\_composite(d\_models, g\_models)

# Print data info

print('Loaded Dataset info:', training\_set.shape)

# # train model

n\_batch = [16, 16, 16, 8, 4]

# # 10 epochs == 500K images per training phase

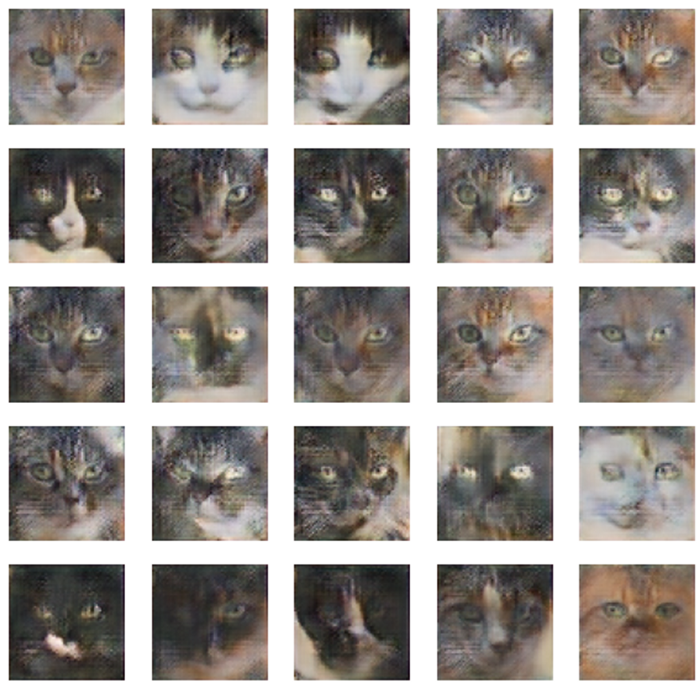
n\_epochs = [5, 8, 8, 10, 10]

train(g\_models, d\_models, gan\_models, training\_set, latent\_dim, n\_epochs, n\_epochs, n\_batch)

Once you have all of this, you are ready to train the model!

**Results**

After training, you can start generating your own images!



Images of Cats Generated from the Model

And there you have it! You’re now one step closer to becoming a millionaire all from a few lines of code.