**11**

**Progressive Growing Generative Adversarial Networks**

GANs are effective at generating crisp synthetic images, but are limited in size to about 64 × 64 pixels. A **Progressive Growing GAN** is an extension of the GAN that enables training generator models to generate large high-quality images up to about 1024 × 1024 pixels (as of this writing). The approach has proven effective at generating high-quality synthetic faces that are startlingly realistic.

The key innovation of Progressive Growing GANs is the incremental increase in the size of images output by the generator. By generating small images at the beginning of training and gradually adding convolutional layers to both generator and discriminator models, larger and larger images with finer and finer resolution are generated.

The technique begins with a low resolution vector as input. It continues by progressively growing the generator and discriminator by adding new layers so the model can increasingly learn fine details during the training process. The technique also speeds and stabilizes training while generating images of unprecedented quality.

# Latent Space Learning

**Latent space** is a representation of compressed data in which similar data points are closer together in space. Latent space is useful for learning data features and finding simpler representations of data for analysis. The idea behind creating a latent space is to compress reality so vector math works. Latent space can also be referred to as latent vector space.

When we observe our world, we see a vast landscape of pixels (or observed pixel space). But how can we hope to learn from such a giant canvas of data? One solution is to create a latent space that compresses an observed pixel space into manageable pixel images. For instance, to teach a model to learn human faces we begin by taking (or using existing) pictures of them. We then convert the pictures into a set of pixel images. So each face is represented by a set of pixels. Now that we have a latent representation, we can apply calculus and vector arithmetic on image pixels to teach a learning model the essence of human faces (at least from a pixel image perspective). Substantively, the latent space for our experiment is a compressed pixel space representation culled from observed pixel spaces of actual human faces.

In an observed pixel space, there may be no immediate similarity between any two images. But mapping a pixel space to a latent space compresses images to be much closer together so we can more easily learn about such images.

A generative model in the GAN architecture learns to map points in a latent space for image generation. It **takes a point from the latent space as input and applies vector arithmetic to generate a new image. A series of points can also be created on a linear path between two points in the latent space to create multiple generated images. In practice, a** generative model effectively uses its latent space representation to interpolate between points in its latent space with the goal of deriving meaningful and targeted effects from its generated images. **But the l**atent space only has meaning as it applies to the generative model being trained. That is, every learning experiment has its own latent space.

Notebooks for chapters are located at the following URL:

https://github.com/paperd/deep-learning-models

We present two Progressive Growing GAN experiments. The first experiment generates images from a pre-trained model. The second experiment creates a custom training loop to learn a target image from an initial generated image. Begin setting up the Colab ecosystem by importing the main TensorFlow library and instantiating the GPU.

# Import the TensorFlow Library

Import the library and alias it as **tf**:

import tensorflow as tf

Aliasing the TensorFlow library as tf is common practice.

# GPU Hardware Accelerator

To vastly speed up processing, use the GPU available from the Google Colab cloud service. Colab provides a free Tesla K80 GPU of about 12 GB. It’s very easy to enable the GPU in a Colab notebook:

1. click *Runtime* in the top left menu
2. click *Change runtime type* from the drop-down menu
3. choose *GPU* from the *Hardware accelerator* drop-down menu
4. click *SAVE*

**Note:** The GPU must be enabled in *each* notebook. But, it only has to be enabled once.

Verify that the GPU is active:

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

**Note:** If you get the error name 'tf' is not defined, re-execute the code to import the TensorFlow library!

# Create Environment for Experiments

Both experiments expect packages, libraries, and functions for image and animation display.

## Install Packages for Creating Animations

Install imageio, scikit-image, and TensoFlow docs packages for animations:

!pip -q install imageio

!pip -q install scikit-image

!pip install -q git+https://github.com/tensorflow/docs

## Install Libraries

Install the logging module on top of standard logging:

from absl import logging

The logging module is from the Abseil Python package, which provides various libraries for building Python applications.

Install image processing libraries:

import imageio

import PIL.Image

import matplotlib.pyplot as plt

from IPython import display

from skimage import transform

The imageio library provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. The Image module is used to represent a PIL image. The Python Imaging Library (PIL) library adds support for opening, manipulating, and saving many different image file formats. The plt library is used for displaying images. The display module is a public API for display tools in IPython. IPython is an interactive shell built with Python. The transform module is used for image processing.

Import other requisite libraries:

import numpy as np

import tensorflow\_hub as hub

from tensorflow\_docs.vis import embed

import time

NumPy is a Python library used for working with scalars, arrays, and matrices. The hub module allows access to TensorFlow Hub, which is a repository of trained machine learning models. The embed module is used to embed animation in a notebook. The time library provides many ways of representing time in code such as objects, numbers, and strings.

# Create Functions for Image Display

Create a function to display an image with the PIL library:

def display\_image(image):

  image = tf.constant(image)

  image = tf.image.convert\_image\_dtype(image, tf.uint8)

  return PIL.Image.fromarray(image.numpy())

Create a function to display an image:

def show\_image(image):

  plt.imshow(image)

  plt.axis('off')

  plt.show()

The imshow() function is used to display data as an image.

Create a function to display animation as shown in Listing 11-1.

def animate(images):

  images = np.array(images)

  converted\_images = np.clip(

      images \* 255, 0, 255).astype(np.uint8)

  imageio.mimsave('./animation.gif', converted\_images)

  return embed.embed\_file('./animation.gif')

Listing 11-1. Function to Display Animation

# Create Latent Space Dimensions

Latent space is useful for learning data features and finding simpler representations of data for analysis. Humans have an understanding of a broad range of topics and the events belonging to those topics. Latent space aims to provide a similar understanding for a computer model through a quantitative spatial representation. So latent space is really just a measurable spatial representation of a compressed reality for a specific training experiment.

Compressing a dataset into latent space helps a model better understand the observed data because the model deals with much smaller variations than it would with the entire dataset. So the model learns from a smaller space than if it had to learn from the actual observed pixel space.

The terms high dimensional and low dimensional help define how specific or general the kinds of features we want our latent space to learn and represent. High dimensional latent space is sensitive to more specific features of the input data, but can sometimes lead to overfitting when there isn't sufficient training data. Low dimensional latent space aims to capture the most important features (or aspects) required to learn and represent the input data.

Set a high dimensional latent space of 512 because the pre-trained model we use for this experiment was trained on this latent space:

latent\_dim = 512

The pre-trained model maps from a 512-dimensional latent space to images. We can retrieve latent dimensions from the module.structured\_input\_signature method if we don't know beforehand the module we are using. We show how to do this later in the chapter.

# Set Verbosity for Error Logging

To see logging errors:

logging.set\_verbosity(logging.ERROR)

# Image Generation Experiment

We use the progan-128 pre-trained model to generate realistic celebrity images. The progan-128 model is a Progressive GAN trained on CelebA for 128 x 128 pixel images. It maps from a 512-dimensional latent space to images. During training, the latent space vectors are sampled from a normal distribution.

The module takes a tensor (Tensor(tf.float32, shape=[?, 512]) that represents a batch of latent vectors as input and outputs a tensor (Tensor(tf.float32, shape=[?, 128, 128, 3]) that represents a batch of RGB images. The original model is trained on a GPU for 636,801 steps with a batch size 16.

CelebA (CelebFaces Attributes Dataset) is a large-scale face attributes dataset containing more than 200,000 celebrity images. Each image has 40 attribute annotations. Images in this dataset cover large pose variations and background clutter. CelebA also has large diversities, large quantities, and rich annotations including:

\* 10,177 number of identities

\* 202,599 number of face images

\* 5 landmark locations

\* 40 binary attributes annotations per image

CelebA can be employed as the training and test sets for computer vision tasks such as face attribute recognition, face detection, landmark (or facial part) localization, and face editing & synthesis.

## Create a Function to Interpolate Hypersphere

Create a function to interpolate the space between vectors in the latent space as shown in Listing 11-2.

def interpolate\_hypersphere(v1, v2, num\_steps):

  v1\_norm = tf.norm(v1)

  v2\_norm = tf.norm(v2)

  v2\_normalized = v2 \* (v1\_norm / v2\_norm)

  vectors = []

  for step in range(num\_steps):

    interpolated =\

      v1 + (v2\_normalized - v1) \*\

      step / (num\_steps - 1)

    interpolated\_norm = tf.norm(interpolated)

    interpolated\_normalized =\

      interpolated \* (v1\_norm / interpolated\_norm)

    vectors.append(interpolated\_normalized)

  return tf.stack(vectors)

Listing 11-2. Function to Interpolate the Hypersphere

The function initiates latent space interpolation between two randomly initialized vectors. It interpolates between vectors that are non-zero and don't both lie on a line going through the origin and returns the normalized interpolated vectors to the calling environment. **Image interpolation** is the process of generating in-between images from a sequence of images.

The function begins by creating two Euclidean normed vectors v1 and v2. It then normalizes v2 to have the same norm as v1. It continues by interpolating between the two vectors on the hypersphere (or latent space) to produce a set of vectors based on the number of interpolation steps.

A **hypersphere** is a four-dimensional analog of a sphere. Although a sphere exists in 3D-space, its surface is two-dimensional. Similarly, a hypersphere has a three-dimensional surface that curves into 4D-space. Our universe could be the hypersurface of a hypersphere.

## Load the Pre-Trained Model

Set a global random seed to maintain reproducibility:

tf.random.set\_seed(7)

Set the seed value. Use any number that you wish. But use the same number on all experiments for reproducibility.

Load the pre-trained Progressive GAN (progran-128):

hub\_model = hub.load(

    'https://tfhub.dev/google/progan-128/1')\

    .signatures['default']

Get output shapes:

hub\_model.output\_shapes

The Progressive GAN (progan-128) is trained on CelebA using image size of 128 x 128 x 3 pixel-images.

Get dimensions for the latent space:

hub\_model.structured\_input\_signature

The progran-128 model maps from a 512-dimensional latent space to images. During training, latent space vectors are sampled from a normal distribution.

## Generate and Display an Image

A new image can be generated from a random point in the latent space. Begin by creating a random normal vector in the latent space. Continue by feeding the vector to progran-128. The pre-trained model identifies the closest vector in the latent space to the random vector and generates an image from the closest vector. Algorithmically, progran-128 identifies the closest latent vector by minimizing the overall distance between the real and generated distribution.

Create a function to find the closest vector in the latent space:

def get\_module\_space\_image():

  vector = tf.random.normal([1, latent\_dim])

  image = hub\_model(vector)['default'][0]

  return image

The function creates a random normal vector between 1 and 512 (the size of our latent space). It then uses progran-128 to generate an image from the closest latent vector to the random latent vector we created.

Display a generated image:

generated\_image = get\_module\_space\_image()

display\_image(generated\_image)

Not bad! The pre-trained model generates a relatively realistic image from a vector drawn from the latent space.

## Create a Function to Generate Multiple Images

The function creates two random vectors, interpolates the space between them in the latent space, and uses progran-128 to generate images as shown in Listing 11-3.

def interpolate\_between\_vectors(steps):

  v1 = tf.random.normal([latent\_dim])

  v2 = tf.random.normal([latent\_dim])

  vectors = interpolate\_hypersphere(v1, v2, steps)

  interpolated\_images = hub\_model(vectors)['default']

  return interpolated\_images

Listing 11-3. Function to Generate Interpolated Images

The function creates two random normal variables from the 512-dimensional latent space. It then creates a tensor with n steps of interpolation between v1 and v2. It ends by using progran-128 to generate n images based on the tensor. Since the tensor consists of n steps of interpolation, generated images are along a line of interpolation bounded by the closest vectors in the latent space between v1 and v2.

## Create an Animation

Create an animation based on n generated images:

interpolation\_steps = 100

interpolated\_images = interpolate\_between\_vectors(

    interpolation\_steps)

animate(interpolated\_images)

Pretty amazing! With the help of progran-128, we create an animation based on interpolation between two vectors in the latent space closest to the two random vectors we created. The number of steps makes a big difference in the animation process! The higher the number of steps, the more interpolated images are created between the vectors in the latent space. So experiment with the number of steps to see what happens. But don’t use too big of an n-size because you may run out of memory!

**Tip:** Keep interpolation steps relatively low to avoid a memory crash! Although Colab is a cloud service, the free version is pretty limited in the amount of allocated RAM to a notebook.

## Display Interpolated Image Vectors

Display interpolated image vectors to deconstruct the image generation process.

Get the number of interpolated images in the latent space bounded by random vectors v1 and v2:

num\_imgs = len(interpolated\_images)

num\_imgs

We set interpolation steps to 100 so we have 100 interpolated images.

Display the initial interpolated image:

show\_image(interpolated\_images[0])

Display the final interpolated image:

show\_image(interpolated\_images[num\_imgs - 1])

So the animation begins with the initial image and morphs into the final image because interpolated vectors are along a line in the latent space bounded by random vectors v1 and v2.

Create a function to display generated image vectors from the latent space as shown in Listing 11-4.

def generated\_images(images, cols, rows):

  columns, rows = cols, rows

  ax = []

  fig = plt.figure(figsize=(20, 20))

  for i in range(columns\*rows):

    img = images[i].numpy()

    ax.append(fig.add\_subplot(rows, columns, i+1))

    plt.imshow(img)

    plt.axis('off')

Listing 11-4. Function to Display Generated Image

Display interpolated images:

generated\_images(interpolated\_images, 10, 10)

It’s fascinating to see how the model creates a series of images that morph from the initial to final one based on interpolated vectors from the latent space! So a learning model can create a series of vectors **on a linear path between two points in the latent space as we demonstrated with our animation.**

## Display Multiple Images from a Single Vector

Create a function to return a latent vector and image:

def get\_vector():

  vector = tf.random.normal([1, latent\_dim])

  image = hub\_model(vector)['default'][0]

  return vector, image

The function returns a random normal latent vector and an image generated by progran-128 from the random latent vector. Each time the function is executed, images are different because each latent vector is generated from a random vector, fed into progran-128, and generated.

Display:

for \_ in range(2):

  latent\_vector, image = get\_vector()

  print (latent\_vector[0][0:3])

  show\_image(image)

For each loop cycle, a slice from each latent vector and its corresponding image is displayed. The two images are dissimilar and have no relationship because we create a latent vector close to a random vector for each loop cycle. But there is no path between the vectors because we didn’t create a series of latent vectors on a linear path bounded by two random vectors! We only interpolated a single vector from the latent space for each loop cycle.

Display several images as shown in Listing 11-5.

rows, cols = 2, 2

plt.figure(figsize=(10, 10))

for i in range(rows\*cols):

  plt.subplot(rows, cols, i + 1)

  plt.imshow(get\_module\_space\_image())

  plt.axis('off')

Listing 11-5. Display Several Images Based on a Single Random Vector

Again, images are dissimilar and have no relationship because the function creates a latent vector close to a random latent vector each time it is invoked!

## Create a Target Latent Vector from an Uploaded Image

As we’ve already demonstrated, we can create a random normal latent vector and generate a new image from it. Alternatively, we can create a vector from an uploaded image vector and generate a new image from it.

Import the requisite library:

from google.colab import files

Create a function to get an uploaded image from your local drive:

def upload\_image():

  uploaded = files.upload()

  image = imageio.imread(uploaded[list(uploaded.keys())[0]])

  return transform.resize(image, [128, 128])

The function use files.upload() from the google.colab library to enable file uploads to the notebook. It uses imread() from the imageio library to read the image from a local drive. The function returns a resized image in the form expected by progran-128 using transform.resize() from the skimage library.

**Note:** Be sure to have an image on your local drive to upload.

Get an image from your local drive:

local\_image = upload\_image()

Click Choose Files to select the image you want to load from your local drive.

Display the image:

display\_image(local\_image)

Create a generated image based on the local image vector:

vector = tf.dtypes.cast(local\_image, tf.float32)

generated\_image = hub\_model(vector)['default'][0]

display\_image(generated\_image)

Instead of creating a random normal latent vector to generate an image, create a vector from an uploaded image to generate an image. We do need to convert the local image tensor to a float tensor. Continue by generating an image from the float tensor with the help of progran-128. End by displaying the image. The progran-128 model generates a new image based on what it learned from CelebA. So a generated image resembles a human face regardless of what image is uploaded because progran-128 learned on human faces! Although deep learning is amazing, it is still limited to what it is trained upon (at least for the time being).

# Custom Loop Learning Experiment

The objective of a supervised learning experiment is to predict the correct label for newly presented input data. The model learns how to connect input features to a predicted outcome in a very precise manner. Although GANs are unsupervised experiments, we can create a pseudo-supervised experiment by creating a series of latent vectors on a linear path bounded by two randomly generated vectors. We identify one of the generated vectors as the feature vector and the other as the predicted (or target) image. A **latent vector** is one that is drawn from the latent space that cannot be accessed or manipulated during training. The idea is similar to how feedforward neural networks cannot manipulate values output by hidden layers.

We train the model by defining a loss function between the feature vector and target image, and use gradient descent on the feature vector to find variable values that minimize the loss. Training begins by defining a loss function between the feature image and target image. A custom training loop enables a gradient descent algorithm and the loss function to find variable values that minimize the loss between the feature vector and target.

## Create the Feature Vector

Generate a seed for reproducibility and create a feature vector:

seed\_value = 777

tf.random.set\_seed(seed\_value)

feature\_vector = tf.random.normal([1, latent\_dim])

Change the seed to generate a different image. Be careful to use the same seed value to maintain reproducibility between experiments.

**Note:** We set a different random seed for this experiment. We could have kept the global random seed that we set in first experiment. But we decided to separate the seeds for the two experiments.

The feature vector can be thought of as the initial vector because the model learns through gradient steps how to proceed from an initial latent vector to the final target image.

Verify that the feature vector was drawn from the latent space:

feature\_vector.shape

The feature vector is drawn from the 512-dimensional latent space as indicated by shape 1 x 512. So the feature vector is a 1-dimensional vector with 512 elements.

## Display an Image from the Feature Vector

Display an image from the feature vector using progran-128:

display\_image(hub\_model(feature\_vector)['default'][0])

## Create the Target Image

Create the target image from the latent space:

target\_image = get\_module\_space\_image()

Verify its shape:

target\_image.shape

The shape of the target tensor is 128 x 128 x 3, which is the expected shape for progran-128.

Display the target image:

display\_image(target\_image)

## Create a Function to Find the Closest Latent Vector

The function defines a loss function between the feature vector and target image. It uses gradient descent to find variable values that minimize loss as shown in Listing 11-6.

def find\_closest\_latent\_vector(

initial\_vector, target\_image,

num\_optimization\_steps,

steps\_per\_image, loss\_alg):

images = []

losses = []

vector = tf.Variable(initial\_vector)

optimizer = tf.optimizers.Adam(learning\_rate=0.01)

loss\_fn = loss\_alg

for step in range(num\_optimization\_steps):

if (step % 100)==0:

print()

print('.', end='')

with tf.GradientTape() as tape:

image = hub\_model(vector.read\_value())['default'][0]

if (step % steps\_per\_image) == 0:

images.append(image.numpy())

target\_image\_difference = loss\_fn(

image, target\_image[:,:,:3])

regularizer = tf.abs(tf.norm(vector) - np.sqrt(latent\_dim))

loss = target\_image\_difference + regularizer

losses.append(loss.numpy())

grads = tape.gradient(loss, [vector])

optimizer.apply\_gradients(zip(grads, [vector]))

return images, losses

Listing 11-6. Custom Training Loop Function

The function accepts the feature vector, number of steps, steps per image, and loss algorithm. It then initializes variables including the optimizer and loss function. It also converts the feature vector to a tf.Variable. A tf.Variable is a tensor whose value can be changed during training.

The function continues with a custom training loop. For custom loop learning, TensorFlow provides the tf.GradientTape API for automatic differentiation. TensorFlow records relevant operations executed inside the context of a tf.GradientTape onto a virtual tape. TensorFlow then uses the virtual tape to compute the gradients of a recorded computation using reverse mode differentiation.

**Automatic Differentiation** (AD) is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. We use AD to compute the gradient of a computation with respect to some inputs (e.g., tf.Variables). AD implements two distinct modes. Forward mode differentiation traverses the chain rule from inside out. Reverse mode differentiation traverses the chain rule from outside to inside. Automatic differentiation is useful for implementing machine learning algorithms such as backpropagation for training neural networks.

For an excellent resource on automatic differentiation, peruse:

https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation

To differentiate automatically, TensorFlow needs to remember what operations happen in what order during the forward pass. Then, during the backward pass, TensorFlow traverses this list of operations in reverse order to compute gradients.

The gradient tape uses the pre-trained model to generate an image, finds the space between the feature image and target image, uses regularization to get more realistic images, calculates loss, applies the gradient descent algorithm, and optimizes the gradients. Once training is completed, the function returns an array of generated images and an array of losses calculated during training.

Since latent vectors are sampled from a normal distribution, we can get more realistic images if we regularize the length of the latent vector to the average length of all vectors from this distribution. We implement regularization with the regularizer variable in the gradient tape section of the function.

## Create the Loss Function

Create the loss function algorithm:

reduction = tf.keras.losses.Reduction.SUM

mae\_loss\_algorithm = tf.losses.MeanAbsoluteError(reduction)

Use Mean Absolute Error (MAE) reduction for the loss function algorithm. **MAE** measures the difference between two continuous variables.

Generally, the tf.losses.MeanAbsoluteError API computes the mean of absolute differences between labels and predictions. In our experiment, it computes the mean absolute difference between latent vectors and the actual target.

## Train the Model

Clear previous model sessions:

tf.keras.backend.clear\_session()

**Tip:** When running multiple training sessions in a notebook, it is a good idea to clear previous model sessions prior to training a model with the tf.keras.backend.clear\_session API.

Invoke the training function:

num\_optimization\_steps = 200

steps\_per\_image = 5

mae\_images, mae\_loss = find\_closest\_latent\_vector(

    feature\_vector, target\_image, num\_optimization\_steps,

    steps\_per\_image, mae\_loss\_algorithm)

Tweak optimization steps and steps per image to see the impact on the visualization.

## Training Loss

Plot loss:

plt.plot(mae\_loss)

fig = plt.ylim([0, max(plt.ylim())])

Calculate the final loss with MAE reduction:

MAE\_loss = mae\_loss[num\_optimization\_steps - 1]

MAE\_loss

## Animate

Create an animation from the images generated from training:

animate(np.stack(mae\_images))

## Compare Learned Images to the Target

Get number of generated images:

len(mae\_images)

Get image type:

type(mae\_images[0])

Create a function to display learned images as shown in Listing 11-7.

def closest\_latent\_images(faces, rows, cols):

  fig = plt.figure(1, (20., 40.))

  for i in range(40):

    plt.subplot(10, 4, i+1)

    plt.imshow(faces[i])

    plt.axis('off')

Listing 11-7. Function to Display Generated Images

Display closest latent images between the feature and target:

closest\_latent\_images(mae\_images, 10, 4)

During each step of the training loop, the function generates a new image by leveraging the pre-trained weights from progan-128. It then computes the loss between the newly generated image and the target. Gradually, through gradient descent and loss minimization techniques images become more and more similar to the target. It's magic!

Contrast the first generated image against the target:

display\_image(np.concatenate(

    [mae\_images[0], target\_image], axis=1))

Show how well the model performed:

display\_image(np.concatenate(

    [mae\_images[-1], target\_image], axis=1))

Grab 'mae\_images[-1]' to display the final generated image.

To get larger images, use the other display function:

show\_image(np.concatenate(

    [mae\_images[-1], target\_image], axis=1))

Not bad at all!

## Try a Different Loss Function Algorithm

Use a MSE instead of MAE:

reduction = tf.keras.losses.Reduction.SUM

mse\_loss\_algorithm = tf.losses.MeanSquaredError(reduction)

**Mean Squared Error** (MSE) measures the average of the squares of the errors (average squared difference) between the estimated values and the target. The tf.losses.MeanSquaredError API computes the mean of squares of errors between labels and predictions.

Clear previous model sessions:

tf.keras.backend.clear\_session()

Train the model with MSE reduction:

num\_optimization\_steps = 200

steps\_per\_image = 5

mse\_images, mse\_loss = find\_closest\_latent\_vector(

    feature\_vector, target\_image, num\_optimization\_steps,

    steps\_per\_image, mse\_loss\_algorithm)

Plot loss:

plt.plot(mse\_loss)

fig = plt.ylim([0, max(plt.ylim())])

Calculate the final loss with MSE reduction:

MSE\_loss = mse\_loss[num\_optimization\_steps - 1]

MSE\_loss

MSE loss is much lower.

Display animation:

animate(np.stack(mse\_images))

Compare the final generated image with the target image:

display\_image(np.concatenate(

    [mse\_images[-1], target\_image], axis=1))

Compare to MAE reduction:

display\_image(np.concatenate(

    [mae\_images[-1], target\_image], axis=1))

## Create a Target from an Uploaded Image

Instead of creating a target image with progran-128 from a latent vector, create a target image with progran-128 from an uploaded image vector.

Generate a seed and create an initial feature vector from the latent space:

seed\_value = 0

tf.random.set\_seed(seed\_value)

feature\_vector = tf.random.normal([1, latent\_dim])

Notice that we use a different seed value. Of course, you can use any seed value you wish. But always use the same seed value across experiments for reproducibility.

Grab an image from a local drive and display:

uploaded\_image = upload\_image()

display\_image(uploaded\_image)

Convert the uploaded image to float32 for TensorFlow consumption:

uploaded\_vector = tf.dtypes.cast(uploaded\_image, tf.float32)

display\_image(uploaded\_vector)

**Note:** The uploaded tensor is not the target image because it was not drawn from the latent space. The target image is created from progran-128 based on the uploaded vector!

Use progran-128 to generate the target image from the vector we just created from the uploaded image:

uploaded\_target = hub\_model(uploaded\_vector)['default'][0]

display\_image(uploaded\_target)

So the uploaded image is not the target image. It is just a vector (once we convert the image to a float tensor) that progran-128 uses to create a target.

Create a loss algorithm with MSE reduction:

reduction = tf.keras.losses.Reduction.SUM

loss\_algorithm = tf.losses.MeanSquaredError(reduction)

We use MSE reduction, but you can substitute MAE reduction if you wish.

Clear previous model sessions:

tf.keras.backend.clear\_session()

Train:

num\_optimization\_steps = 300

steps\_per\_image = 5

mse\_images, mse\_loss = find\_closest\_latent\_vector(

    feature\_vector, uploaded\_target, num\_optimization\_steps,

    steps\_per\_image, loss\_algorithm)

We trained the model on more optimization steps to generate a better facsimile of the target image.

Calculate the final loss:

MSE\_loss = mse\_loss[num\_optimization\_steps - 1]

MSE\_loss

Animate:

animate(np.stack(mse\_images))

Compare the final generated image to the target:

display\_image(np.concatenate(

    [mse\_images[-1], uploaded\_target], axis=1))

Not bad. We can increase the number of optimization steps to generate a more realistic image. But be careful because setting the number of steps too high might compromise available RAM.

## Create a Target from a Google Drive Image

Instead of grabbing an image from a local drive, grab it from Google Drive. We create a new feature vector, but you can use the one we created for the uploaded image exercise if you wish.

Generate a seed and create an initial feature vector from the latent space:

seed\_value = 0

tf.random.set\_seed(seed\_value)

feature\_vector = tf.random.normal([1, latent\_dim])

Mount Google Drive:

from google.colab import drive

drive.mount('/content/gdrive')

Click on the URL, choose a Gmail account, copy the authorization code, paste it into the text box, and click the Enter button on your keypad.

Get and display the image:

from PIL import Image

p1 = 'gdrive/My Drive/Colab Notebooks/'

p2 = 'images/honest\_abe.jpeg'

path = p1 + p2

img\_path = path

gdrive\_image = Image.open(img\_path)

plt.axis('off')

\_ = plt.imshow(gdrive\_image)

Create a path to the image on Google Drive. Open the image with the path and display. Be sure that the image is in the Colab Notebooks directory!

**Note:** Any image (or file) that you wish to load into a Colab notebook must be in the 'Colab Notebooks' directory (or a subdirectory inside the 'Colab Notebooks' directory) on your Google Drive. The 'Colab Notebooks' directory is automatically created the first time you save a Colab notebook. All Colab notebooks are saved to the 'Colab Notebooks' directory.

Create a function to convert the Google Drive image to a TensorFlow consumable tensor:

def reformat(img, size):

  img = tf.keras.preprocessing.image.img\_to\_array(img) / 255.

  img = tf.image.resize(img, size)

  return img

The Keras utility converts a PIL Image instance to a NumPy array and resizes it to the expected size of progran-128, which is 128 x 128 x 3.

Invoke the function:

img\_size = (128, 128)

gdrive\_vector = reformat(gdrive\_image, img\_size)

gdrive\_vector.shape

Display the image:

display\_image(gdrive\_vector)

**Note:** The uploaded tensor is not the target image because it was not drawn from the latent space. The target image is created from progra-128 based on the uploaded tensor!

Use progran-128 to generate the target image from the vector we just created from the Google Drive image:

gdrive\_target = hub\_model(gdrive\_vector)['default'][0]

display\_image(gdrive\_target)

Again, the image loaded from Google Drive is not the target image. It is just a vector (once we convert the image to a NumPy array) based on the Google Drive image that progran-128 uses to create a target image.

Create a loss algorithm with MSE reduction:

reduction = tf.keras.losses.Reduction.SUM

loss\_algorithm = tf.losses.MeanSquaredError(reduction)

We use MSE reduction, but MAE reduction should also work.

Clear previous model sessions:

tf.keras.backend.clear\_session()

Train:

num\_optimization\_steps = 300

steps\_per\_image = 5

mse\_images, mse\_loss = find\_closest\_latent\_vector(

    feature\_vector, gdrive\_target, num\_optimization\_steps,

    steps\_per\_image, loss\_algorithm)

We trained the model on more optimization steps to generate a better facsimile of the target image.

Calculate the final loss:

MSE\_loss = mse\_loss[num\_optimization\_steps - 1]

MSE\_loss

Animate:

animate(np.stack(mse\_images))

Compare the final generated image to the target:

display\_image(np.concatenate(

    [mse\_images[-1], gdrive\_target], axis=1))

## Create a Target from a Wikimedia Commons Image

Wikimedia Commons is an online repository of free-use images, sounds, other media, and JSON files. It is a project of the Wikimedia Foundation. Wikimedia Commons only accepts free content such as images and other media files not subject to copyright restrictions that would prevent them being used by anyone, anytime, for any purpose.

Generate a seed and create a feature vector:

seed\_value = 0

tf.random.set\_seed(seed\_value)

feature\_vector = tf.random.normal([1, latent\_dim])

Grab an image:

p1 = 'http://upload.wikimedia.org/wikipedia/commons/'

p2 = 'd/de/Wikipedia\_Logo\_1.0.png'

URL = p1 + p2

im = imageio.imread(URL)

im.shape

Import the requisite library for converting an image tensor to a NumPy array:

from keras.preprocessing.image import img\_to\_array

Convert the image to a NumPy array and display:

img\_array = img\_to\_array(im)

print(img\_array.dtype)

print(img\_array.shape)

plt.imshow(tf.squeeze(img\_array))

fig = plt.axis('off')

The Keras utility converts a PIL Image instance to a NumPy array so we can display the image.

Resize the image for progran-128 consumption:

wiki\_vector = tf.image.resize(img\_array, (128, 128))

plt.imshow(tf.squeeze(wiki\_vector))

fig = plt.axis('off')

Resize the image to the expected size of progran-128 and display it.

Create the target:

wiki\_target = hub\_model(wiki\_vector)['default'][0]

display\_image(wiki\_target)

Create a loss algorithm with MSE reduction:

reduction = tf.keras.losses.Reduction.SUM

loss\_algorithm = tf.losses.MeanSquaredError(reduction)

Clear previous model sessions:

tf.keras.backend.clear\_session()

Train:

num\_optimization\_steps = 300

steps\_per\_image = 5

mse\_images, mse\_loss = find\_closest\_latent\_vector(

    feature\_vector, wiki\_target, num\_optimization\_steps,

    steps\_per\_image, loss\_algorithm)

Calculate the final loss:

MSE\_loss = mse\_loss[num\_optimization\_steps - 1]

MSE\_loss

Animate:

animate(np.stack(mse\_images))

Compare:

display\_image(np.concatenate(

    [mse\_images[-1], wiki\_target], axis=1))

Not bad.

Although we generate pretty realistic images and facsimiles of a target image, much more robust models and training algorithms are needed to consistently generate images that are indistinguishable from actual ones.

# Latent Vectors and Image Arrays

The progran-128 module generates a new image from either a latent vector of size (1, 512) or float vector of size 128 x 128 x 3. A latent vector accepted by progran-128 is a 1-dimensional vector of size 512. A float vector accepted by progran-128 is 128 x 128 x 3-pixel vector.

The progran-128 module is an image generator based on the TensorFlow re-implementation of Progressive GANs. It maps from a 512-dimensional latent space to images. During training, latent space vectors are sampled from a normal distribution.

According to the documentation, progran-128 takes an input tensor with datatype float32 tensor and shape (?, 512). The input tensor to progran-128 represents a batch of latent vectors. The output from progran-128 is a float tensor with shape (?, 128, 128, 3), which represents a batch of RGB images. We can also generate a new image from an image array, which is not included in the documentation.

To view progran-128 documentation, peruse:

https://tfhub.dev/google/progan-128/1

## Generate a New Image from a Latent Vector

Create a random normal vector from the latent space:

random\_normal\_latent\_vector = tf.random.normal([1, latent\_dim])

random\_normal\_latent\_vector.shape

The tf.random.normal API outputs random values from a normal distribution. So the new vector consists of 512 randomly drawn values from a normal distribution based on our latent space.

Convert the tensor to NumPy to enable inspection:

rnlv = random\_normal\_latent\_vector.numpy()

len(rnlv[0])

Inspect some elements from the NumPy array:

for i, element in enumerate(rnlv[0]):

  if i < 5:

    print (element)

  else: break

Each element in the new vector represents a latent dimension (or latent variable) that cannot be directory observed, but can be assumed to exist. Since latent dimensions exist, they can be used to explain patterns of variation in observed variables. In our experiment, observed variables represent CelebA images. So we can feed progran-128 the new vector to generate a new image.

Create a float output tensor from the latent space with progran-128:

float\_output\_tensor = hub\_model(

    random\_normal\_latent\_vector)['default'][0]

float\_output\_tensor.shape

Display the float output tensor as an image:

display\_image(generated\_image)

So progran-128 generates a 128 x 128 x 3 image from a latent vector.

## Generate a New Image from an Image Vector

Get an image from Google Drive:

p1 = 'gdrive/My Drive/Colab Notebooks/'

p2 = 'images/honest\_abe.jpeg'

path = p1 + p2

img\_path = path

abe\_image = Image.open(img\_path)

plt.axis('off')

\_ = plt.imshow(abe\_image)

Convert the JPEG image to an image vector of the appropriate datatype and size:

img\_size = (128, 128)

abe\_vector = reformat(abe\_image, img\_size)

abe\_vector.shape

Display a slice from the vector:

abe\_vector[0][0].numpy()

The vector is definitely not from the latent space!

Generate a new image from the vector we just created:

image\_from\_abe\_vector = hub\_model(abe\_vector)['default'][0]

display\_image(image\_from\_abe\_vector)

So progran-128 generates a 128 x 128 x 3 image from an image vector!