# setup

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import os

import IPython

from IPython.display import display

import numpy as np

import PIL.Image

import pandas as pd

import six

import tensorflow as tf

import tensorflow\_hub as hub

def imgrid(imarray, cols=8, pad=1):

pad = int(pad)

assert pad >= 0

cols = int(cols)

assert cols >= 1

N, H, W, C = imarray.shape

rows = int(np.ceil(N / float(cols)))

batch\_pad = rows \* cols - N

assert batch\_pad >= 0

post\_pad = [batch\_pad, pad, pad, 0]

pad\_arg = [[0, p] for p in post\_pad]

imarray = np.pad(imarray, pad\_arg, 'constant')

H += pad

W += pad

grid = (imarray

.reshape(rows, cols, H, W, C)

.transpose(0, 2, 1, 3, 4)

.reshape(rows\*H, cols\*W, C))

return grid[:-pad, :-pad]

def imshow(a, format='png', jpeg\_fallback=True):

a = np.asarray(a, dtype=np.uint8)

if six.PY3:

str\_file = six.BytesIO()

else:

str\_file = six.StringIO()

PIL.Image.fromarray(a).save(str\_file, format)

png\_data = str\_file.getvalue()

try:

disp = display(IPython.display.Image(png\_data))

except IOError:

if jpeg\_fallback and format != 'jpeg':

print ('Warning: image was too large to display in format "{}"; '

'trying jpeg instead.').format(format)

return imshow(a, format='jpeg')

else:

raise

return disp

class Generator(object):

def \_\_init\_\_(self, module\_spec):

self.\_module\_spec = module\_spec

self.\_sess = None

self.\_graph = tf.Graph()

self.\_load\_model()

@property

def z\_dim(self):

return self.\_z.shape[-1].value

@property

def conditional(self):

return self.\_labels is not None

def \_load\_model(self):

with self.\_graph.as\_default():

self.\_generator = hub.Module(self.\_module\_spec, name="gen\_module",

tags={"gen", "bsNone"})

input\_info = self.\_generator.get\_input\_info\_dict()

inputs = {k: tf.placeholder(v.dtype, v.get\_shape().as\_list(), k)

for k, v in self.\_generator.get\_input\_info\_dict().items()}

self.\_samples = self.\_generator(inputs=inputs, as\_dict=True)["generated"]

print("Inputs:", inputs)

print("Outputs:", self.\_samples)

self.\_z = inputs["z"]

self.\_labels = inputs.get("labels", None)

def \_init\_session(self):

if self.\_sess is None:

self.\_sess = tf.Session(graph=self.\_graph)

self.\_sess.run(tf.global\_variables\_initializer())

def get\_noise(self, num\_samples, seed=None):

if np.isscalar(seed):

np.random.seed(seed)

return np.random.normal(size=[num\_samples, self.z\_dim])

z = np.empty(shape=(len(seed), self.z\_dim), dtype=np.float32)

for i, s in enumerate(seed):

np.random.seed(s)

z[i] = np.random.normal(size=[self.z\_dim])

return z

def get\_samples(self, z, labels=None):

with self.\_graph.as\_default():

self.\_init\_session()

feed\_dict = {self.\_z: z}

if self.conditional:

assert labels is not None

assert labels.shape[0] == z.shape[0]

feed\_dict[self.\_labels] = labels

samples = self.\_sess.run(self.\_samples, feed\_dict=feed\_dict)

return np.uint8(np.clip(256 \* samples, 0, 255))

class Discriminator(object):

def \_\_init\_\_(self, module\_spec):

self.\_module\_spec = module\_spec

self.\_sess = None

self.\_graph = tf.Graph()

self.\_load\_model()

@property

def conditional(self):

return "labels" in self.\_inputs

@property

def image\_shape(self):

return self.\_inputs["images"].shape.as\_list()[1:]

def \_load\_model(self):

with self.\_graph.as\_default():

self.\_discriminator = hub.Module(self.\_module\_spec, name="disc\_module",

tags={"disc", "bsNone"})

input\_info = self.\_discriminator.get\_input\_info\_dict()

self.\_inputs = {k: tf.placeholder(v.dtype, v.get\_shape().as\_list(), k)

for k, v in input\_info.items()}

self.\_outputs = self.\_discriminator(inputs=self.\_inputs, as\_dict=True)

print("Inputs:", self.\_inputs)

print("Outputs:", self.\_outputs)

def \_init\_session(self):

if self.\_sess is None:

self.\_sess = tf.Session(graph=self.\_graph)

self.\_sess.run(tf.global\_variables\_initializer())

def predict(self, images, labels=None):

with self.\_graph.as\_default():

self.\_init\_session()

feed\_dict = {self.\_inputs["images"]: images}

if "labels" in self.\_inputs:

assert labels is not None

assert labels.shape[0] == images.shape[0]

feed\_dict[self.\_inputs["labels"]] = labels

return self.\_sess.run(self.\_outputs, feed\_dict=feed\_dict)

#Select a model { run: "auto" }

model\_name = "S3GAN 256x256 10% labels (FID 8.8, IS 130.7)" # @param ["S3GAN 256x256 10% labels (FID 8.8, IS 130.7)", "S3GAN 128x128 2.5% labels (FID 12.6, IS 48.7)", "S3GAN 128x128 5% labels (FID 8.4, IS 74.0)", "S3GAN 128x128 10% labels (FID 7.6, IS 90.3)", "S3GAN 128x128 20% labels (FID 6.9, IS 98.1)"]

models = {

"S3GAN 256x256 10% labels": "https://tfhub.dev/google/compare\_gan/s3gan\_10\_256x256/1",

"S3GAN 128x128 2.5% labels": "https://tfhub.dev/google/compare\_gan/s3gan\_2\_5\_128x128/1",

"S3GAN 128x128 5% labels": "https://tfhub.dev/google/compare\_gan/s3gan\_5\_128x128/1",

"S3GAN 128x128 10% labels": "https://tfhub.dev/google/compare\_gan/s3gan\_10\_128x128/1",

"S3GAN 128x128 20% labels": "https://tfhub.dev/google/compare\_gan/s3gan\_20\_128x128/1",

}

module\_spec = models[model\_name.split(" (")[0]]

print("Module spec:", module\_spec)

tf.reset\_default\_graph()

print("Loading model...")

sampler = Generator(module\_spec)

print("Model loaded.")

#Sampling { run: "auto" }

num\_rows = 1 # @param {type: "slider", min:1, max:16}

num\_cols = 4 # @param {type: "slider", min:1, max:16}

noise\_seed = 53 # @param {type:"slider", min:0, max:100, step:1}

label\_str = "951) lemon" # @param ["-1) Random", "0)

num\_samples = num\_rows \* num\_cols

z = sampler.get\_noise(num\_samples, seed=noise\_seed)

label = int(label\_str.split(')')[0])

if label == -1:

labels = np.random.randint(0, num\_classes, size=(num\_samples))

else:

labels = np.asarray([label] \* num\_samples)

samples = sampler.get\_samples(z, labels)

imshow(imgrid(samples, cols=num\_cols))

#Interpolation { run: "auto" }

num\_samples = 1 # @param {type: "slider", min: 1, max: 6, step: 1}

num\_interps = 3 # @param {type: "slider", min: 2, max: 10, step: 1}

noise\_seed\_A = 17 # @param {type: "slider", min: 0, max: 100, step: 1}

noise\_seed\_B = 0 # @param {type: "slider", min: 0, max: 100, step: 1}

label\_str = "1) goldfish, Carassius auratus" # @param ["0) tench]

def interpolate(A, B, num\_interps):

alphas = np.linspace(0, 1, num\_interps)

if A.shape != B.shape:

raise ValueError('A and B must have the same shape to interpolate.')

return np.array([((1-a)\*A + a\*B)/np.sqrt(a\*\*2 + (1-a)\*\*2) for a in alphas])

def interpolate\_and\_shape(A, B, num\_interps):

interps = interpolate(A, B, num\_interps)

return (interps.transpose(1, 0, \*range(2, len(interps.shape)))

.reshape(num\_samples \* num\_interps, -1))

label = int(label\_str.split(')')[0])

labels = np.asarray([label] \* num\_samples \* num\_interps)

z\_A = sampler.get\_noise(num\_samples, seed=noise\_seed\_A)

z\_B = sampler.get\_noise(num\_samples, seed=noise\_seed\_B)

z = interpolate\_and\_shape(z\_A, z\_B, num\_interps)

samples = sampler.get\_samples(z, labels)

imshow(imgrid(samples, cols=num\_interps))

#Discriminator

disc = Discriminator(module\_spec)

batch\_size = 4

num\_classes = 1000

images = np.random.random(size=[batch\_size] + disc.image\_shape)

labels = np.random.randint(0, num\_classes, size=(batch\_size))

disc.predict(images, labels=labels)