celebGAN2

November 7, 2022

[ ]:
```
# basic imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

from tensorflow.keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt

import os
import io

import datetime

%load_ext tensorboard
```

[ ]:
```
# Set base Parameters
IMAGE_HEIGHT = 64
IMAGE_WIDTH = 64
BATCH_SIZE = 256
```

[ ]:
```
# Load Dataset from storage
AUTOTUNE = tf.data.AUTOTUNE
dataset = tf.keras.utils.image_dataset_from_directory("dataset/preprocessed",
  image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
  batch_size=BATCH_SIZE)
normalization_layer = tf.keras.layers.Rescaling(1./127.5, offset=-1)
dataset = dataset.map(lambda x, y: (normalization_layer(x), y))
```

[ ]:
```
# Define generator model
def make_generator(latent_vector_shape, dense_shape):
    latent_input = layers.Input(shape=latent_vector_shape)
    gen = layers.Dense(4*4*dense_shape)(latent_input)
    gen = layers.ReLU()(gen)
    gen = layers.Reshape((4,4,dense_shape))(gen)
    gen = layers.Dropout(0.2)(gen)
    gen = layers.Conv2DTranspose(dense_shape, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/2, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/4, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/8, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)

    out = layers.Conv2D(3, (4, 4), strides=(1,1), padding="same",␣
 ↪activation='tanh')(gen)

    model: keras.Model = keras.Model(latent_input, out)
    print(model.output_shape)
    assert model.output_shape == (None, 64, 64, 3)
    return model

generator = make_generator(128, 256)
generator.summary()
```

[ ]:
```
#define discriminator model
def make_discriminator(input_shape):
    image_input = layers.Input(input_shape)

    disc = layers.Conv2D(8, (3, 3))(image_input)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.BatchNormalization()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)


    disc = layers.Conv2D(16, (3, 3))(disc)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)
    disc = layers.Dropout(0.3)(disc)

    disc = layers.Conv2D(32, (3, 3))(disc)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)

    disc = layers.Flatten()(disc)
    disc = layers.Dropout(0.3)(disc)
    disc = layers.Dense(32)(disc)
    out = layers.Dense(1)(disc)

    model = keras.Model(image_input, out)

    return model

discriminator = make_discriminator((64, 64, 3))
discriminator.summary()
```

[ ]:
```
# define GAN model
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim=128,␣
 ↪disc_extra_steps=3):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_steps = disc_extra_steps

    def compile(self, d_optimizer, g_optimizer, d_loss_fn, g_loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.d_loss_fn = d_loss_fn
        self.g_loss_fn = g_loss_fn

    def train_step(self, data):
        images, _ = data
        #calculate bacth size of current batch
        batch_size = tf.shape(images)[0]

        for i in range(self.d_steps):
            #generate new latent vector
            latent_vector = tf.random.normal(shape=(batch_size, self.
 ↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
 ↪tape
                # this allows backpropagation and automatic taking of the␣
 ↪derivative
                generated_images = self.generator(latent_vector, training=True)
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)

                #flip images randomly to inttroduce variety
                flipped_images = tf.image.random_flip_left_right(images)
                prediction_real = self.discriminator(flipped_images,␣
↪training=True)

                #calculate discriminator loss
                d_loss = self.d_loss_fn(prediction_real, prediction_fake)
            #calculate discriminator gradients
            d_gradients = gt.gradient(d_loss, self.discriminator.
↪trainable_variables)
            #apply gradients using Adam
            self.d_optimizer.apply_gradients(zip(d_gradients, self.
↪discriminator.trainable_variables))

            #generate new latent vector for generator training
            latent_vector = tf.random.normal(shape=(batch_size, self.
↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
↪tape
                # this allows backpropagation and automatic taking of the␣
↪derivative
                generated_images = self.generator(latent_vector, training=True)
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)
                #calculate generator loss
                g_loss = self.g_loss_fn(prediction_fake)

            #calculate gradients and apply them using Adam
            g_gradients = gt.gradient(g_loss, self.generator.
↪trainable_variables)
            self.g_optimizer.apply_gradients(zip(g_gradients, self.generator.
↪trainable_variables))

        return {"d_loss": d_loss, "g_loss": g_loss}
```

[ ]:
```
# define Loss functions
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=True)
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

#initalize optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

[ ]:
```
# Enable Checkpoint saving if training gets interrupted
checkpoint_dir = 'celebGAN2/training_checkpoints'
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
manager = tf.train.CheckpointManager(checkpoint, checkpoint_dir, max_to_keep=5)
```

[ ]:
```
log_dir = "celebGAN2/logs/fit/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
img_log_dir = "celebGAN2/logs/images/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,␣
 ↪histogram_freq=1)

file_writer = tf.summary.create_file_writer(img_log_dir)

# Method converting matplotlib figures to images usable by tensorboard
# Source https://www.tensorflow.org/tensorboard/image_summaries
def plot_to_image(figure):
    """Converts the matplotlib plot specified by 'figure' to a PNG image and
    returns it. The supplied figure is closed and inaccessible after this call.
 ↪"""
    # Save the plot to a PNG in memory.
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    # Closing the figure prevents it from being displayed directly inside
    # the notebook.
    plt.close(figure)
    buf.seek(0)
    # Convert PNG buffer to TF image
    image = tf.image.decode_png(buf.getvalue(), channels=4)
    # Add the batch dimension
    image = tf.expand_dims(image, 0)
    return image

# Save Image every epoch
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=16, latent_dim=100, start_epoch=0, seed=None):
        self.num_img = num_img
        if not seed == None:
            self.seed = seed
        else:
            self.seed = tf.random.normal(shape=(num_img, latent_dim))
        self.start_epoch = start_epoch

    def on_epoch_end(self, epoch, logs=None):
        generated_images = self.model.generator(self.seed, training=False)
        generated_images = (generated_images * 127.5) + 127.5
        generated_images = generated_images.numpy()

        fig = plt.figure(figsize=(4, 4))

        #generate a subplot
        for i in range(generated_images.shape[0]):
            plt.subplot(4, 4, i+1)
            plt.imshow(generated_images[i, :, :, :].astype("int32"))
            plt.axis('off')

        #save to harddrive
        plt.savefig(os.path.join("celebGAN2/", "images/",'image_at_epoch_{:04d}.
 ↪png'.format(self.start_epoch+epoch)))
        with file_writer.as_default():
            tf.summary.image("Output", plot_to_image(fig), step=epoch)

# Save Checkpoint every 2 epochs
class GANSaver(keras.callbacks.Callback):
    def __init__(self, manager, num_epochs=15):
        self.num_epochs = num_epochs
        self.manager = manager

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.num_epochs == 0:
            self.manager.save()


ckp = GANSaver(manager, 2)
```

```
[ ]: gan = GAN(discriminator, generator, latent_dim=128, disc_extra_steps=1)
     gan.compile(discriminator_optimizer, generator_optimizer, discriminator_loss,␣
      ↪generator_loss)
```

```
[ ]: #restore latest state of training
     if manager.latest_checkpoint:
         checkpoint.restore(manager.latest_checkpoint)
         latest_epoch = int(manager.latest_checkpoint.split('-')[1])
```

```
    last_epoch = latest_epoch * 2
    print ('Latest checkpoint of epoch {} restored!!'.format(last_epoch))
else:
    last_epoch = 0
    print ('No latest checkpoint found!')
#initialize image saver with start epoch variable to keep existing images after␣
 ↪restart
ick = GANMonitor(num_img=16, latent_dim=128, start_epoch=last_epoch)

#start tensorboard
%tensorboard --logdir celebGAN2/logs
#train model on the dataset for 100 epochs
gan.fit(dataset, epochs=100, batch_size=256, callbacks=[ick, ckp,␣
 ↪tensorboard_callback])
```

[ ]:
```
# save model to harddrive
manager.save()
# store seed in variable to keep faces consistent when rerun
seed = ick.seed
```

[ ]:
```
#generate 100 sample images in batches
images = 25
predictions = np.empty([100,64,64,3])
for i in range(4):
    seed = tf.random.normal([images, 128])
    label_seed = np.random.randint(0,2, images)
    pred = generator(seed, training=False).numpy()
    predictions[25*i:25*(i+1), :, :, :] = pred

print(predictions.shape)
figsize = 10
fig = plt.figure(figsize=(figsize, figsize))
for i in range(predictions.shape[0]):
    plt.subplot(figsize, figsize, i+1)
    plt.imshow((predictions[i, :, :, :]*127.5+127.5).astype("int32"))
    plt.axis('off')
plt.show()
```

[ ]:
```
# output the Generator and Discriminator as Image
tf.keras.utils.plot_model(generator, "celebGAN/Generator.png", show_shapes=True)
tf.keras.utils.plot_model(discriminator, "celebGAN/Discriminator.png",␣
 ↪show_shapes=True)
```

[ ]:
```
# save Generator as h5 model to use in e.g. matlab
generator.save("celebGAN/generator_model/generator.h5")
```