

پروژه یک: آموزش مدل GAN با زبان پایتون و کتابخانه Tensorflow

۱- شبکه های GAN:

در شبکه های GAN دو شبکه عصبی، در یک بازی با یکدیگر به رقابت می پردازند. با بهره گیری از یک مجموعه داده آموزشی این روش می آموزد تا داده های جدیدی مشابه با مجموعه آموزش تولید کند. یک شبکه GAN که روی مجموعه داده تصاویر آموزش داده شده باشد، می تواند تصاویر جدیدی را تولید کند حداقل به طور سطحی توسط ناظر انسانی قابل باور باشند.

شبکه های GAN راهکاری هوشمندانه برای آموزش دادن یک مدل مولد هستند. آن ها این کار را با قاب بندی مساله به عنوان یک مساله یادگیری نظارت شده با دو زیر مدل انجام می دهند. این دو زیر مدل عبارتند از «مدل مولد» (Generator Model) که برای تولید نمونه های جدید آموزش داده می شود و «مدل متمایزگر» (Discriminator Model) که تلاش می کند تا نمونه ها را به عنوان نمونه واقعی (از دامنه) یا جعلی (تولید شده) دسته بندی کند.

۲.۱- مدل مولد

مدل مولد یک بردار تصادفی با طول ثابت را به عنوان ورودی دریافت و نمونه ها را در دامنه تولید می کند. بردار به طور تصادفی از یک توزیع گوسی برگرفته شده و بردار برای دانه دادن به فرایند مولد مورد استفاده قرار می گیرد. پس از آموزش دادن، نقاط در این فضای برداری چندبُعدی متناظر با نقاط در دامنه مسأله خواهند بود و یک ارائه فشرده از توزیع داده ها را ارائه می کنند. در GAN ها، مدل مولد، به نقاط در فضای پنهان معنا می بخشد؛ به طوری که نقاط جدیدی که برگرفته از فضای پنهان هستند را می توان برای مدل مولد به عنوان ورودی قرار داد و از آن ها برای تولید خروجی های جدید و متفاوت استفاده کرد.

۳.۱- مدل متمایزگر

مدل متمایزگر، نمونه ای از دامنه را به عنوان ورودی (واقعی یا تولید شده) دریافت و برچسب کلاس دودویی را واقعی یا جعلی پیش بینی می کند. مثال های واقعی از مجموعه داده آموزش می آیند. مثال های تولید شده خروجی مدل مولد هستند. متمایزگر یک مدل دسته بندی است. پس از فرایند آموزش، مدل متمایزگر کنار گذاشته می شود، زیرا مدل مولد است که جذابیت دارد.

۴.۱- شبکه های GAN به عنوان یک بازی دو بازیکنی

یکی از خصوصیات هوشمندانه معماری GAN آموزش دادن مدل مولد به عنوان یک مساله یادگیری نظارت شده قاب بندی شده است. دو مدل مولد و متمایزگر، با یکدیگر آموزش می بینند. مولد، دسته ای از نمونه ها را آماده می کند و این دسته، همراه با مثال هایی از دامنه، برای متمایزگر فراهم و به عنوان واقعی یا جعلی دسته بندی می شوند. مولد سپس به روز رسانی می شود تا در تمایز بین نمونه های واقعی و جعلی در دور بعدی بهتر عمل کند. مهم تر آنکه، مولد نیز بر پایه اینکه متمایزگر با نمونه تولید شده به اندازه کافی خوب گول خورده است یا خیر نیز به روز رسانی می شود.

۲- پیاده سازی

برای پیاده سازی از کتابخانه تنسوفلو و کراس استفاده شده است. کدهای این پروژه بر روی محیط گوگل کولب اجرا شد.

در ابتدای پروژه کتابخانه‌ها و فریم‌ورک‌های مورد نیاز را ایمپورت می‌کنیم:

```
import tensorflow_datasets as tfds
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.preprocessing.image import array_to_img
from tensorflow.keras.callbacks import Callback
import matplotlib.pyplot as plt
import numpy as np
import datetime
import os
```

برای اینکه بتوانیم نتایج مدل و عکس‌های تولید شده را ذخیره کنیم گوگل درایو را به محیط کولب متصل می‌کنیم:

```
from google.colab import drive
drive.mount('/content/drive')
```

TensorBoard ابزاری برای ارائه اندازه‌گیری‌ها و تجسم‌های مورد نیاز در جریان کار یادگیری ماشین است. ردیابی معیارهای آزمایش مانند از دست دادن و دقت، تجسم نمودار مدل، پیش‌بینی جاسازی‌ها در فضای با ابعاد پایین‌تر و موارد دیگر را امکان‌پذیر می‌کند. افزونه تنسوربرد به محیط کولب لود می‌کنیم:

```
%load_ext tensorboard
```

لاگ‌هایی که ممکن است از اجرای قبلی باقی مانده باشد را پاک می‌کنیم:

```
!rm -rf ./logs/
```

هایپرپارامترهای مدل را طبق help متلب تنظیم می‌کنیم:

```
BATCH_SIZE = 128
IMG_SIZE = 64

EPOCHS = 500
noise_dim = 100
```

۱.۲ ساخت دیتاست

حال باید دیتاست مورد نیاز برای آموزش مدل را آماده کنیم. برای این کار چندین روش مختلف وجود دارد. یک روش دانلود مستقیم و برداشتن از روی دیسک، دیگری استفاده از `tf.keras.utils.get_file` برای دانلود دیتاست و قرار گرفتن آن بر روی محیط کولب است.

```
flowers = tf.keras.utils.get_file(
    'flower_photos',
```

```
'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
    untar=True)
```


اگر از این روش استفاده کنیم برای افزودگی دیتاست به طریق زیر می توان عمل کرد:

```
img_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255,
    vertical_flip=True,
    horizontal_flip=True,
    rotation_range=20)
```

که ابتدا یک شی برای افزودگی ایجاد می شود سپس با `tf.data.Dataset.from_generator` می توان اقدام به ساخت دیتاست نمود.

```
ds = tf.data.Dataset.from_generator(
    lambda: img_gen.flow_from_directory(flowers,
        target_size=(IMG_SIZE, IMG_SIZE),
        batch_size=BATCH_SIZE),
    output_types=(tf.float32, tf.float32),
    output_shapes=([BATCH_SIZE, IMG_SIZE, IMG_SIZE, 3], [BATCH_SIZE, 5])
)
```

اما اگر به سایت تنسورفلو برای `ImageDataGenerator` مراجعه کنید با پیام منسوخ شدن این روش برای کدهای جدید مواجه می شوید:

 **Deprecated:** `tf.keras.preprocessing.image.ImageDataGenerator` is not recommended for new code. Prefer loading images with `tf.keras.utils.image_dataset_from_directory` and transforming the output `tf.data.Dataset` with preprocessing layers. For more information, see the tutorials for [loading images](#) and [augmenting images](#), as well as the [preprocessing layer guide](#).

البته می توان افزودگی را خودمان به صورت دستی بدون استفاده از توابع تنسورفلو که مختص افزودگی کل دیتاست هستند، هم انجام بدهیم:

```
ds = ds.map(lambda image, label: (tf.image.flip_left_right(image), label))
ds = ds.map(lambda image, label: (tf.image.adjust_contrast(image, 0.5), label))
ds = ds.map(lambda image, label: (tf.image.adjust_brightness(image, delta=0.2), label))
ds = ds.map(lambda image, label: (tf.image.flip_up_down(image), label))
```

انتخاب ما در این پروژه برای بارگیری تصاویر استفاده از `tensorflow_datasets` است. که با استفاده از `tfds.load` دیتاست نامبرده را در یک `tf.data.Dataset` بارگذاری می کند.

```
data, info = tfds.load(name="tf_flowers", split="train",
```

```
as_supervised=True, with_info=True)
```

برای تغییر اندازه عکس‌های دیتاست و تغییر اسکیل پیکسل‌ها از لایه پیش‌پردازش کراس استفاده می‌کنیم:

```
resize_and_rescale = tf.keras.Sequential([
    layers.Resizing(IMG_SIZE, IMG_SIZE),
    layers.Rescaling(1./255)])
```

layers.Rescaling اندازه تصاویر را به بازه صفر تا یک می‌برد.

برای افزودگی تصاویر طبق help مطلب از از لایه پیش‌پردازش کراس استفاده می‌کنیم:

```
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2),])
```

دو راه برای استفاده از این لایه‌های پیش‌پردازش وجود دارد.

گزینه ۱ این است که لایه‌های پیش‌پردازش را بخشی از مدل خود کنیم. در این مورد دو نکته مهم وجود دارد که باید به آن توجه کنیم:

۱. افزایش داده روی دستگاه، همزمان با بقیه لایه‌ها اجرا می‌شود و از GPU بهره می‌برد.

۲. وقتی مدل خود را با استفاده از model.save ذخیره کنیم، لایه‌های پیش‌پردازش به همراه بقیه مدل ذخیره می‌شوند.

اگر بعداً این مدل را اجرا کنیم، به طور خودکار تصاویر را استاندارد می‌کند.

گزینه ۲: لایه‌های پیش‌پردازش را به مجموعه داده خود اعمال کنیم. با این رویکرد، از Dataset.map برای ایجاد مجموعه داده‌ای استفاده می‌کنیم که دسته‌هایی از تصاویر افزون شده را به دست می‌دهد. در این حالت افزودگی تصاویر به صورت ناهمزمان در CPU اتفاق می‌افتد.

```
def prepare(ds):
    ds = ds.map(lambda x, y: (resize_and_rescale(x)),
                num_parallel_calls=AUTOTUNE)

    ds = ds.cache()
    ds = ds.shuffle(BUFFER_SIZE)

    ds = ds.batch(BATCH_SIZE)

    ds = ds.map(lambda x: (data_augmentation(x, training=True)),
                num_parallel_calls=AUTOTUNE)

    ds.prefetch(buffer_size=AUTOTUNE)

    return ds
```

در ابتدا تغییر اندازه عکس و اسکیل پیکسل‌ها را به تمام دیتاست اعمال می‌کنیم و در آخرین مرحله از لایه‌های افزودنی استفاده می‌کنیم.

Dataset.cache: تصاویر را پس از بارگیری از دیسک در اولین اپوک در حافظه نگه می‌دارد. این تضمین می‌کند که مجموعه داده در حین آموزش مدل به گلوگاه تبدیل نشود.

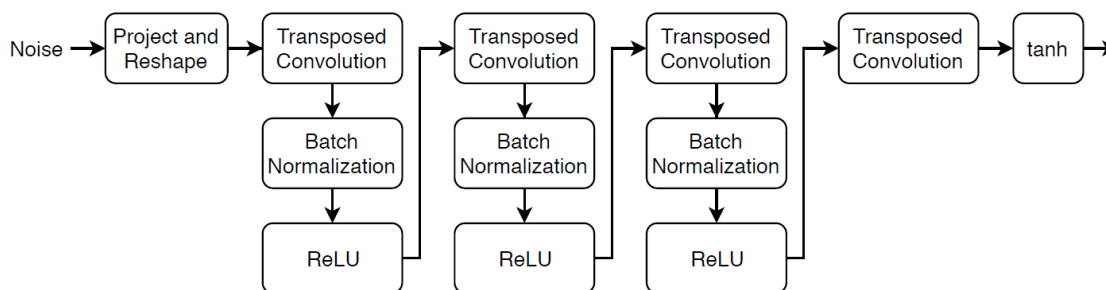
Dataset.prefetch: زمانی که GPU در حال کار بر روی forward / backward propagation در دسته فعلی است، ما می‌خواهیم که CPU دسته بعدی داده را پردازش کند تا بلافاصله آماده شود. همیشه در طول آموزش استفاده می‌شود. به این همپوشانی مصرف کننده / تولید کننده گفته می‌شود که در آن مصرف کننده GPU و تولید کننده CPU است. با tf.data می‌توان این کار را با یک فراخوانی ساده database.prefetch انجام دهیم و مطمئن شویم که همیشه یک دسته از داده‌ها آماده است. کار آماده سازی دیتاست را با فراخوانی تابع بالا به اتمام می‌رسانیم:

```
ds = prepare(data)
```

۲.۲- ساخت مدل‌ها:

۱.۲.۲- ساخت مدل Generator:

معماری این مدل help مطلب به صورت تصویر زیر ترسیم شده است:



این شبکه بردارهای تصادفی با اندازه ۱۰۰ را با استفاده از عملیات پروژکشن و تغییر شکل به آرایه‌های ۴ در ۴ در ۵۱۲ تبدیل می‌کند. آرایه‌های به دست آمده را با استفاده از یک سری لایه‌های کانولوشن و لایه‌های ReLU به آرایه‌های ۶۴ در ۶۴ در ۳ ارتقا می‌دهد. برای لایه کانولوشن انتقالی نهایی، سه فیلتر به سه کانال RGB تصاویر تولید می‌شود. در انتهای شبکه یک لایه tanh قرار می‌گیرد.

تمامی موارد فوق در کدزنی اعمال شده‌است هرچند در تمامی منابع که مشاهده کردم از LeakyRelu به عنوان تابع فعال‌ساز استفاده می‌شد. تنها تفاوت با توضیحات مطلب این است که اندازه کرنل از ۵ تا ۴ برای جلوگیری از کاهش مصنوعات شطرنجی در تصاویر تولید شده است که در مقاله Deconvolution and Checkerboard Artifacts به آن اشاره می‌شود. این به این دلیل است که اندازه کرنل ۵ بر گام ۲ تقسیم نمی‌شود، بنابراین راه حل این است که به جای ۵ از اندازه هسته ۴ استفاده کنیم. برای دسترسی به این مقاله می‌توانید به آدرس <https://distill.pub/2016/deconv-checkerboard> مراجعه فرمایید.

```
def generator_model():  
    model = tf.keras.Sequential()
```

```

model.add(layers.Dense(4*4*512, input_shape=(noise_dim,)))
model.add(layers.Reshape((4, 4, 512)))
assert model.output_shape == (None, 4, 4, 512)

model.add(layers.Conv2DTranspose(256, (4, 4), strides=(2, 2), padding='same'))
assert model.output_shape == (None, 8, 8, 256)
model.add(layers.BatchNormalization())
model.add(layers.ReLU())

model.add(layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
assert model.output_shape == (None, 16, 16, 128)
model.add(layers.BatchNormalization())
model.add(layers.ReLU())

model.add(layers.Conv2DTranspose(64, (4, 4), strides=(2, 2), padding='same'))
assert model.output_shape == (None, 32, 32, 64)
model.add(layers.BatchNormalization())
model.add(layers.ReLU())

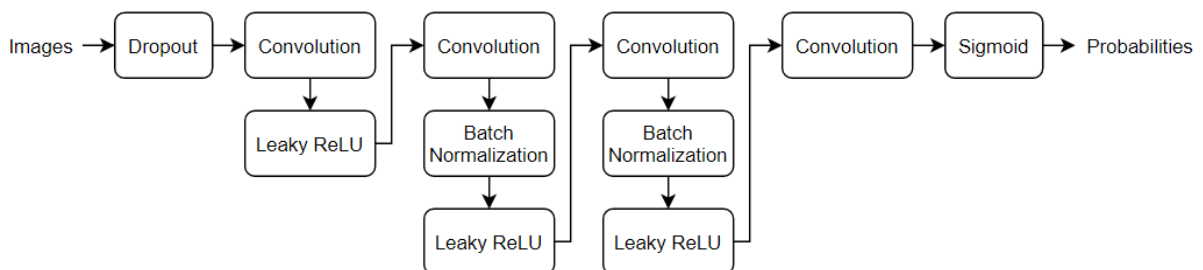
model.add(layers.Conv2DTranspose(3, (filterSize, filterSize), strides=(2,
2), padding='same', use_bias=False, activation='tanh'))
assert model.output_shape == (None, 64, 64, 3)

return model

```

۲.۲.۲- ساخت مدل Discriminator

معماری این مدل help متلب به صورت تصویر زیر ترسیم شده است:



در این تصویر ۴ لایه کانولوشن نشان داده شده است. اما با توجه ریزتر به کد متلبی که در help درج شده است در آن ۵ لایه کانولوشن وجود دارد.

```

imageInputLayer(inputSize,Normalization="none")
dropoutLayer(dropoutProb)
#1

```

```

convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same")
leakyReluLayer(scale)
#2
convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same")
batchNormalizationLayer
leakyReluLayer(scale)
#3
convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same")
batchNormalizationLayer
leakyReluLayer(scale)
#4
convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same")
batchNormalizationLayer
leakyReluLayer(scale)
#5
convolution2dLayer(4,1)
sigmoidLayer];

```

ما هم از ۵ لایه کانولوشنی استفاده می کنیم:

این شبکه تصاویر ۶۴ در ۶۴ در ۳ را می گیرد و با استفاده از یک سری لایه های کانولوشن با batchNormalization و leakyRelu، پیش بینی را برمی گرداند. برای لایه های کانولوشن، فیلترهای ۵ در ۵ را با تعداد فیلترهای فزاینده برای هر لایه مشخص می کنیم. همچنین یک گام ۲ و padding را تعیین می کنیم. برای leakyRelu، مقیاس ۰.۲ را تعیین می کنیم. برای خروجی احتمالات در محدوده [۰،۱]، یک لایه کانولوشن با یک فیلتر ۴ در ۴ و به دنبال آن یک لایه سیگموئید قرار می دهیم.

```

def discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape = [IMG_SIZE, IMG_SIZE, 3]))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(512, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Conv2D(1, (4, 4))

```

```

model.add(layers.Flatten())
model.add(layers.Dense(1, activation='sigmoid'))

return model

```

چون در هنگام آموزش مدل خودمان نویز اضافه می‌کنیم لایه dropout اول را حذف کردیم.

۳.۲- ساخت حلقه آموزش دو مدل به صورت همزمان:

در ابتدا اپتیمايز Adam و تابع BinaryCrossentropy (برای دسته‌بندی دودویی) را برای آموزش مدل تعیین می‌کنیم:

```

g_opt = tf.keras.optimizers.Adam(1e-4)
d_opt = tf.keras.optimizers.Adam(1e-4)
loss_fn = BinaryCrossentropy()

```

کلاس GAN را تعریف می‌کنیم که از Model کراس ارث بری می‌کند. مقادیری که تعریف کردیم به compile می‌فرستیم.

```

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

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

```

در متد train_step که در زمان fit کردن مدل برای آموزش فراخوانی می‌شود نحوه محاسبه توابع خطا برای هر دو شبکه را مشخص می‌کنیم.

ابتدا برای آموزش شبکه discriminator یک batch از تصاویری که generator تولید می‌کند را با اعداد تصادفی بدست می‌آوریم. بعد این تصاویر را به تصاویر اصلی الحاق می‌کنیم. برای دو دسته تصویر لیبل تعیین می‌کنیم. برای تصاویر جعلی برچسب یک و برای تصاویر اصلی برچسب صفر را اختصاص می‌دهیم. مقدار کمی نویز برای آموزش بهتر شبکه به برچسب اضافه می‌کنیم. خروجی شبکه discriminator را برای این تصاویر بدست می‌آوریم و اشتباهات شبکه را نسبت به برچسب داده شده محاسبه می‌کنیم و گرادینان را بر وزن‌های شبکه اعمال می‌کنیم.


```
def train_step(self, real_images):
    batchsize = tf.shape(real_images)[0]
    generated_images=self.generator(tf.random.normal((batchsize, self.latent_dim)
    ))
    combined_images = tf.concat([generated_images, real_images], axis=0)

    labels = tf.concat([tf.ones((batchsize, 1)),tf.zeros((batchsize, 1))],axis=0)
    labels += 0.05 * tf.random.uniform(tf.shape(labels))

    with tf.GradientTape() as tape:
        predictions = self.discriminator(combined_images)
        d_loss = self.loss_fn(labels, predictions)

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

برای آموزش مدل generator یک batch از تصاویری که generator تولید می‌کند را با اعداد تصادفی بدست می‌آوریم و به عنوان ورودی discriminator قرار می‌دهیم.

لیبل این دسته از تصاویر که generator تولید کرده را برابر صفر (تصاویر واقعی) در نظر می‌گیریم، به این شکل است که مدل generator آموزش می‌بیند. برای آموزش شبکه discriminator ما برای تصاویر واقعی برچسب صفر در نظر گرفتیم، برای آموزش discriminator ما این تصاویر تولید شده را به عنوان تصاویر واقعی در نظر می‌گیریم. به نوعی شبکه generator هر بار که discriminator تصور کند که این تصاویر واقعی است پاداش می‌گیرد. اشتباهات شبکه را نسبت به برچسب داده شده محاسبه می‌کنیم و گرادینان را بر وزن‌های شبکه اعمال می‌کنیم.

```
random_latent_vectors = tf.random.normal(shape=(batchsize, self.latent_dim))
misleading_labels = tf.zeros((batchsize, 1))

with tf.GradientTape() as tape:
    predictions = self.discriminator(self.generator(random_latent_vectors))
    g_loss = self.loss_fn(misleading_labels, predictions)

grads = tape.gradient(g_loss, self.generator.trainable_weights)
self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))
```

یک کلاس می‌سازیم که از Callback ارث بری می‌کند تا تصاویر تولید شده توسط مدل و خود مدل را در هر ۱۰۰ اپوک ذخیره کند.

```
class ModelMonitor(Callback):

    def __init__(self, num_img=3, latent_dim=noise_dim):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):

        if epoch % 100 == 0:
            random_latent_vectors = tf.random.uniform((self.num_img, self.latent_dim))
            generated_images = self.model.generator(random_latent_vectors)
            generated_images *= 255
            generated_images.numpy()

            for i in range(self.num_img):
                img = array_to_img(generated_images[i])
                img.save(os.path.join(checkpoint_dir + "/images", f'generated_img_{epoch}_{i}.png'))

            self.model.generator.save(checkpoint_dir + f'/weights/generator_{epoch}.h5')
```

یک شی از کلاس GAN می‌سازیم و مدل‌ها را به آن می‌دهیم. برای کامپایل هم مقادیر تعریف شده را به تابع می‌دهیم:

```
gan = GAN(discriminator, generator, noise_dim)
gan.compile(g_opt, d_opt, loss_fn)
```

برای رهگیری تمام مشخصات مدل در حین آموزش یک شی از callbacks.TensorBoard می‌سازیم:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
```

و در نهایت آموزش مدل با که ساخته‌ایم:

```
hist = gan.fit(ds, epochs=EPOCHS, callbacks=[ModelMonitor(), tensorboard_callback])
```

برای مشاهده تنسوربورد:

```
%tensorboard --logdir logs/fit
```