

# آموزش مدل GAN روی دیتاست tf\_flowers

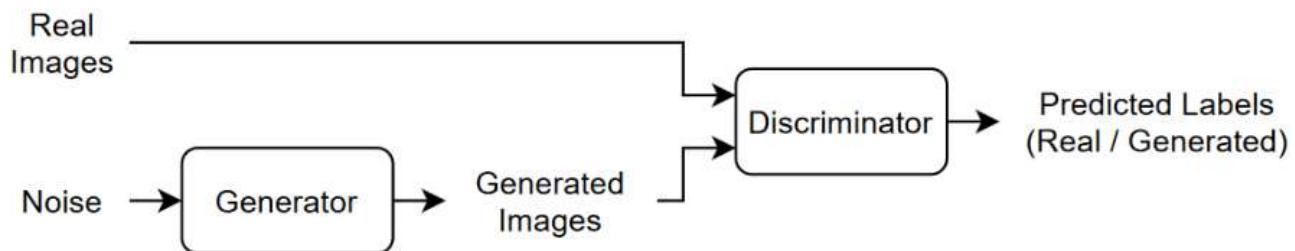
در این پروژه از شبکه GAN برای تولید تصاویری از انواع گل های مختلف استفاده می شود. برای آموزش شبکه از دیتاست استفاده می شود که حاوی 3670 تصویر از انواع گل ها می باشد. جزئیات پروژه در مورد نوع شبکه طراحی شده و پیش پردازش های مورد نیاز در ادامه به طور کامل شرح داده شده است.

به طور کلی شبکه GAN، نوعی شبکه عصبی است که در صورت آموزش مناسب، می تواند داده های مشابه داده های واقعی تولید کند. این شبکه از دو بخش مولد / Generator و متمايزگر / Discriminator تشکيل شده است.

- یک بردار تصادفی در ورودی تولید می کند و تلاش می کند این بردار را به داده های مشابه با داده های واقعی تبدیل کند.
- یک داده را در ورودی دریافت می کند و تلاش می کند جعلی یا واقعی بودن آن را تشخیص دهد.

در واقع دو بخش Generator و Discriminator به صورت رقابتی عمل می کنند. به عبارت دیگر Generator تلاش می کند داده هایی تولید کند که بسیار مشابه داده های واقعی باشند به گونه ای که Discriminator نتواند جعلی بودن آن ها را تشخیص دهد و در عین حال Discriminator تلاش می کند بتواند به طور دقیق داده های جعلی را از داده های واقعی تشخیص دهد.

تصویر زیر ساختار کلی یک شبکه GAN را نشان می دهد:



در این گزارش در 8 بخش مختلف به شرح زیر، فرایند آموزش یک شبکه GAN برای تولید تصاویری از انواع گل شرح داده شده است:

1. پیش پردازش دیتاست
2. ذخیره سازی و بارگذاری تصاویر در قالب tfrecord
3. طراحی شبکه GAN - مدل اول
4. طراحی شبکه GAN - مدل دوم
5. توابع کمکی برای رسم تصاویر و نمودارها
6. تنسوربورد
7. آموزش مدل
8. نتایج اجرا

# 1. پیش‌پردازش دیتاست

## بارگذاری دیتاست

ابتدا دیتاست با استفاده از کد زیر بارگذاری می‌شود:

```
#-- Load DS --
ds, ds_info = tfds.load(
    'tf_flowers',
    with_info=True,
    as_supervised=True,
)

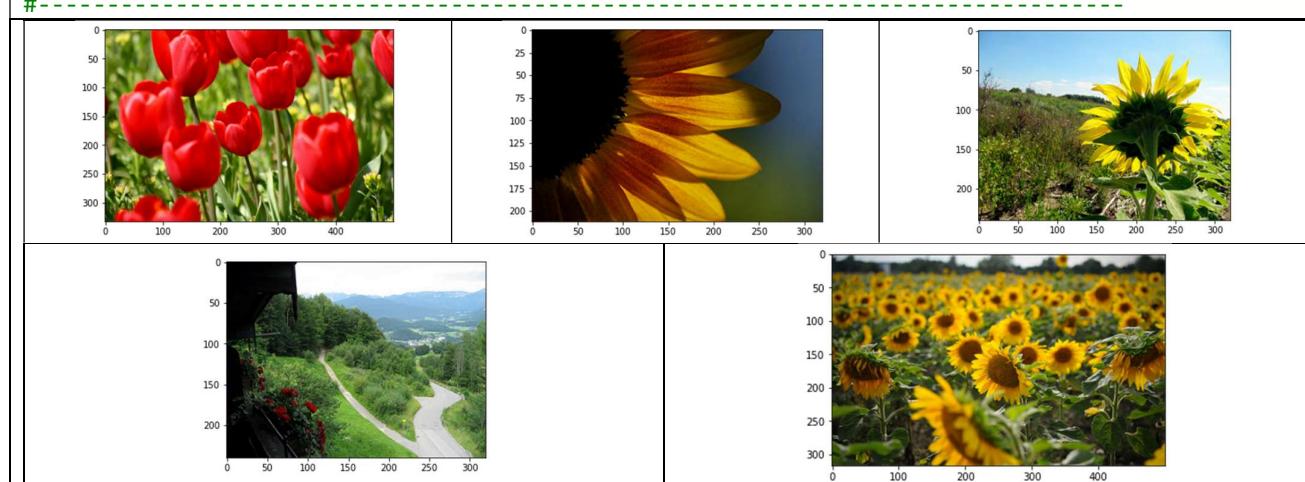
training_set = ds['train']
#--
```

این دیتاست شامل 3670 تصویر از 5 نوع گل متفاوت است. گل‌ها در این دیتاست برچسب‌گذاری شده‌اند اما با توجه به اینکه در این پروژه صرفاً تولید گل مهم است و گل خاصی مد نظر نیست، نیازی به استفاده از برچسب‌ها نیست و در ادامه، فقط از تصاویر موجود در این دیتاست استفاده خواهیم کرد.

## نمایش چند نمونه از تصاویر دیتاست

پنج تصویر ابتدایی دیتاست را با استفاده از کد زیر نمایش می‌دهیم:

```
#-- Plot a few First Images --
for image, label in training_set.take(5):
    image = image.numpy()
    plt.figure()
    plt.imshow(image, cmap=plt.cm.binary)
    plt.show()
#--
```



## استخراج تصاویر از دیتاست

همانطور که اشاره شد، در این پروژه فقط به تصاویر موجود در دیتاست نیاز داریم. کد زیر تصاویر را استخراج می‌کند و در لیستی با عنوان images قرار می‌دهد:

```
-- Extract Images from DS --
images = []
for image, label in training_set.take(-1):
    images.append(image.numpy())

print(len(images))
#-----
3760
```

## افزونه‌سازی تصاویر

به منظور آموزش بهتر مدل لازم است عمل افزونه‌سازی روی تصاویر موجود در دیتاست انجام شود. بدین منظور از کتابخانه استفاده شده و 5 حالت از افزونه‌سازی انتخاب شده است.

روی هر تصویر حداقل 3 افزونه‌سازی انجام می‌شود که به طور تصادفی از 5 حالت مورد نظر، انتخاب می‌شود.  
در نهایت لیست augmented\_images شامل تصاویر اصلی و تصاویر افزونه‌سازی شده می‌باشد.

```
-- Image Augmentation --
-- list of augmentations --
transform = A.Compose([
    A.RandomRotate90(),
    ATranspose(),
    A.ShiftScaleRotate(shift_limit=0.08, scale_limit=0.5, rotate_limit=5, p=.8),
    A.Blur(blur_limit=7),
    A.GridDistortion(),
])

-- Run a Maximum of 3 Augmentations on each Image --
augmented_images = images.copy()
max_aug = 3
for img in images:
    n_aug = random.randint(1, max_aug)
    for i in range(n_aug):
        aug_img = transform(image=img)[ 'image' ]
        augmented_images.append(aug_img)

print(len(augmented_images))
#-----
10997
```

## یکسان‌سازی اندازه تصاویر موجود در دیتاست

تصاویر موجود در این دیتاست، اندازه‌های مختلفی دارند. کد زیر اندازه 5 تصویر ابتدایی را نشان می‌دهد:

```
-- Show Images Size --
for i in range(5):
```

```

print('Image {} shape: {}'.format(i+1, images[i].shape))
#-----
Image 1 shape: (333, 500, 3)
Image 2 shape: (212, 320, 3)
Image 3 shape: (240, 320, 3)
Image 4 shape: (240, 320, 3)
Image 5 shape: (317, 500, 3)

```

جهت آموزش شبکه لازم است، تصاویر موجود در دیتابست، اندازه یکسانی داشته باشند. در این پروژه از تصاویر رنگی با اندازه 64\*64 استفاده می‌شود و اندازه همه تصاویر به 64\*64 تغییر می‌کند:

```

IMG_ROW, IMG_COL, IMG_CHANNEL = 64, 64, 3
#-- Resize Image -----
def Resize_Image(image):
    image = tf.image.resize(image, (IMG_ROW, IMG_COL))/255.0
    return image
#-----

#-- Resize All Images --
training_images = list(map(Resize_Image, augmented_images))

#-- Show Images Size --
for i in range(5):
    print('Image {} shape: {}'.format(i+1, training_images[i].shape))
#-----
Image 1 shape: (64, 64, 3)
Image 2 shape: (64, 64, 3)
Image 3 shape: (64, 64, 3)
Image 4 shape: (64, 64, 3)
Image 5 shape: (64, 64, 3)

```

## 2. ذخیره‌سازی و بارگذاری تصاویر در قالب tfrecord

### ذخیره‌سازی تصاویر در قالب tfrecord

در این مرحله همه تصاویر پردازش شده در مراحل پیشین، در قالب یک فایل به نام images.tfrecords ذخیره می‌شود. برای این منظور از کتابخانه numpy2tfrecord استفاده می‌شود. هر رکورد به صورت یک دیکشنری ساخته می‌شود که کلید آن 'img' و مقدار آن یک ارایه سه بعدی 64\*64\*3 حاوی مقادیر پیکسل‌های تصویر می‌باشد.

```

#-- Save Images as tfrecord --
#-- Set path and file name --
path_tfrecords = 'images.tfrecords'

#-- convert all images to tfrecords --
samples = []
with Numpy2TFRecordConverter(path_tfrecords) as converter:
    for img in training_images:
        img = img.numpy()
        img = img.astype(np.float32)
        sample = {"img": img}

```

```

samples.append(sample)

converter.convert_list(samples)
#-
```

## بارگذاری تصاویر از فایل tfrecord

به منظور آموزش مدل با استفاده از تصاویر ذخیره شده در قالب tfrecord لازم است این فایلها decode شوند.

تابع Read\_tfrecord هر رکورد موجود را Decode میکند و محتوای آن را در قالب تصویری با اندازه 3\*64\*64 بازمیگرداند:

```

--> Read each tfrecord and decode it to an image -----
def Read_tfrecord(sample):
    tfrecord_format = (
        {'img': tf.io.FixedLenFeature([IMG_ROW,IMG_COL,IMG_CHANNEL], tf.float32),})
    sample = tf.io.parse_single_example(sample, tfrecord_format)
    image = sample['img']

    return image
#-
```

تابع Load\_Dataset با استفاده از تابع Read\_tfrecord همه رکوردهای موجود در فایل را decode میکند و در قالب یک MapDataset بازمیگرداند.

```

--> Load DS from tfrecords file -----
def Load_Dataset(filename):
    ignore_order = tf.data.Options()
    ignore_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filename)
    dataset = dataset.with_options(ignore_order)
    dataset = dataset.map(partial(Read_tfrecord))

    return dataset
#-
```

در نهایت با استفاده از توابع فوق فایل images.tfrecords میشود و تصاویر در X\_train ذخیره میشوند.

```

--> Load DS from tfrecords and Save as Numpy Array -----
dataset = Load_Dataset(path_tfrecords)
print(type(dataset))

ds_size = len(list(dataset))

X_train = np.zeros((ds_size,IMG_ROW,IMG_COL,IMG_CHANNEL))

index = 0
for image in dataset.take(-1):
    X_train[index] = image
    index += 1
```

```

np.random.shuffle(X_train)
print(X_train.shape)
#-----
<class 'tensorflow.python.data.ops.dataset_ops.MapDataset'>
(10992, 64, 64, 3)

```

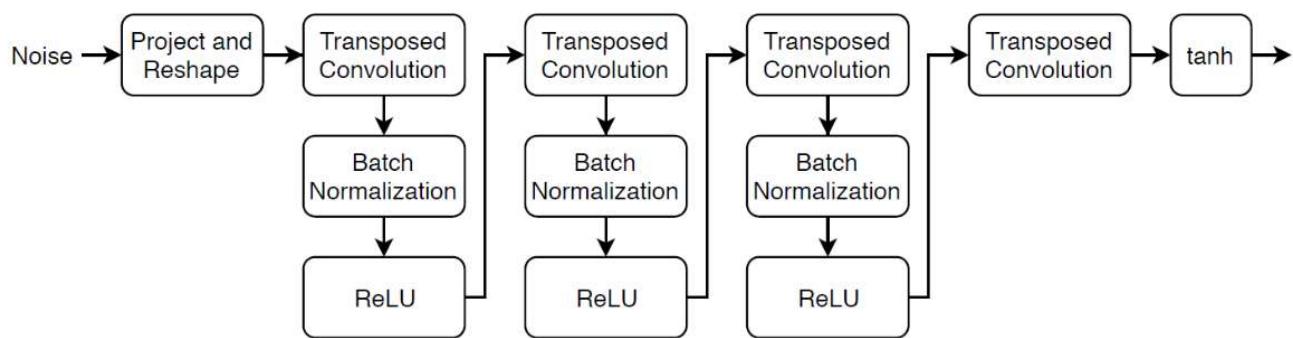
### 3. طراحی شبکه GAN - مدل اول

در بخش اول از مدل طراحی شده در لینک زیر استفاده شده است:

<https://www.mathworks.com/help/deeplearning/ug/train-generative-adversarial-network.html>

#### Generator مدل

در مدل اول، ساختار شبکه generator به صورت زیر می باشد:



به طور کلی این شبکه یک بردار تصادفی در اندازه 100 را به عنوان ورودی دریافت می‌کند و در خروجی یک تصویر  $64 \times 64 \times 3$  تولید می‌کند. کد زیر این شبکه را تعریف می‌کند.

```

NOISE_VEC_DIM = 100
OPTIMIZER = optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
FILTER_SIZE = 5
N_FILTERS= 64
def Create_Generator_1():

    generator = Sequential()
    #-----

    d = 4
    generator.add(layers.Dense(d*d*512,
                               input_shape=(NOISE_VEC_DIM, ),
                               name='gen_dense'))
    generator.add(layers.Reshape((d, d, 512),
                               name='gen_reshape'))
    print(generator.output_shape)
    #-----

    generator.add(layers.Conv2DTranspose(4*N_FILTERS,
                                         (FILTER_SIZE, FILTER_SIZE),
                                         name='gen_Con2DTrans_1'))
    generator.add(layers.BatchNormalization(name='gen_BatchNorm_1'))

```

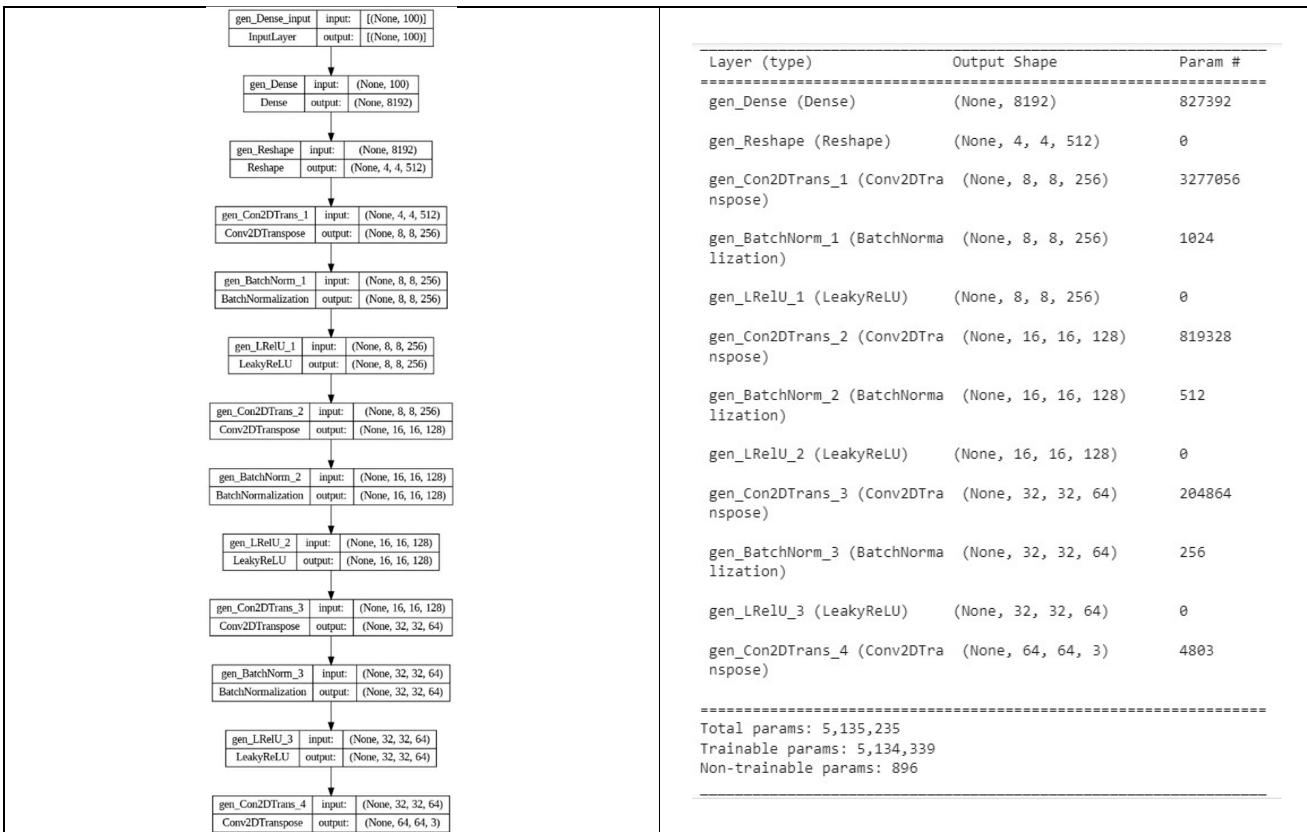
```

generator.add(layers.LeakyReLU(name='gen_LReLU_1'))
print(generator.output_shape)
#-----#
generator.add(layers.Conv2DTranspose(2*N_FILTERS,
                                    (FILTER_SIZE, FILTER_SIZE),
                                    strides=(2, 2),
                                    padding='same',
                                    name='gen_Con2DTrans_2'))
generator.add(layers.BatchNormalization(name='gen_BatchNorm_2'))
generator.add(layers.LeakyReLU(name='gen_LReLU_2'))
print(generator.output_shape)
#-----#
generator.add(layers.Conv2DTranspose(N_FILTERS,
                                    (FILTER_SIZE, FILTER_SIZE),
                                    strides=(2, 2),
                                    padding='same',
                                    name='gen_Con2DTrans_3'))
generator.add(layers.BatchNormalization(name='gen_BatchNorm_3'))
generator.add(layers.LeakyReLU(name='gen_LReLU_3'))
print(generator.output_shape)

#-----#
generator.add(layers.Conv2DTranspose(3,
                                    (FILTER_SIZE, FILTER_SIZE),
                                    strides=(2, 2),
                                    padding='same',
                                    activation='tanh',
                                    name='gen_Con2DTrans_4'))
print(generator.output_shape)
#-----#
generator.compile(loss='binary_crossentropy', optimizer=OPTIMIZER)
#-----#
return generator

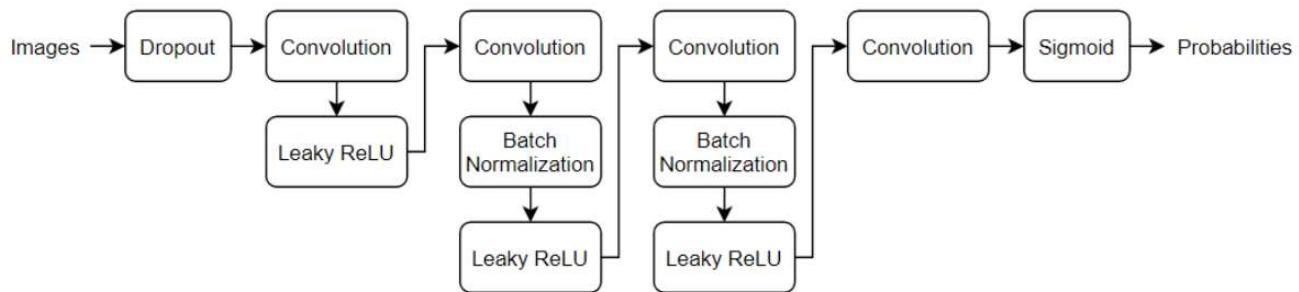
```

شکل زیر ساختار این شبکه را نشان می‌دهد. همانطور که مشخص است این شبکه یک بردار به اندازه 100 را در ورودی دریافت می‌کند و مرحله به مرحله آن را تبدیل به یک ماتریس 3 $\times$ 64 $\times$ 64 (یک تصویر رنگی با اندازه 64 $\times$ 64) می‌کند.



## مدل Discriminator

ساختار شبکه discriminator به صورت زیر می‌باشد:



به طور کلی، این شبکه یک تصویر در اندازه  $64 \times 64 \times 3$  را در ورودی دریافت می‌کند و جعلی یا واقعی بودن آن را در خروجی تشخیص می‌دهد. کد زیر این شبکه را تعریف می‌کند:

```

--> Discriminator 1 <-
def Create_Discriminator_1():
    discriminator = Sequential()
    #-----#
    discriminator.add(layers.InputLayer(input_shape=INPUT_SIZE,
                                         name = 'disc_Input'))
    discriminator.add(layers.Dropout(DROPOUT_PROB,
                                     name= 'disc_Dropout_1'))
    print(discriminator.output_shape)
    #-----#
  
```

```

discriminator.add(layers.Conv2D(N_FILTERS,
                               (FILTER_SIZE, FILTER_SIZE),
                               strides=(2, 2),
                               padding='same',
                               name='disc_Conv_1'))
discriminator.add(layers.LeakyReLU(SCALE,
                                   name = 'disc_LReLU_1'))
print(discriminator.output_shape)
#-----

discriminator.add(layers.Conv2D(2*N_FILTERS,
                               (FILTER_SIZE, FILTER_SIZE),
                               strides=(2, 2),
                               padding='same',
                               name='disc_Conv_2'))
discriminator.add(layers.BatchNormalization(name='BatchNorm_2'))
discriminator.add(layers.LeakyReLU(SCALE,
                                   name = 'disc_LReLU_2'))
print(discriminator.output_shape)
#-----

discriminator.add(layers.Conv2D(4*N_FILTERS,
                               (FILTER_SIZE, FILTER_SIZE),
                               strides=(2, 2),
                               padding='same',
                               name='disc_Conv_3'))
discriminator.add(layers.BatchNormalization(name='BatchNorm_3'))
discriminator.add(layers.LeakyReLU(SCALE,
                                   name = 'disc_LReLU_3'))
print(discriminator.output_shape)
#-----

discriminator.add(layers.Conv2D(8*N_FILTERS,
                               (FILTER_SIZE, FILTER_SIZE),
                               strides=(2, 2),
                               padding='same',
                               name='disc_Conv_4'))
discriminator.add(layers.BatchNormalization(name='BatchNorm_4'))
discriminator.add(layers.LeakyReLU(SCALE,
                                   name = 'disc_LReLU_4'))
print(discriminator.output_shape)
#-----
```

discriminator.add(layers.Flatten(name='disc\_Flatten'))  
discriminator.add(layers.Dropout(0.4, name='disc\_Dropout'))

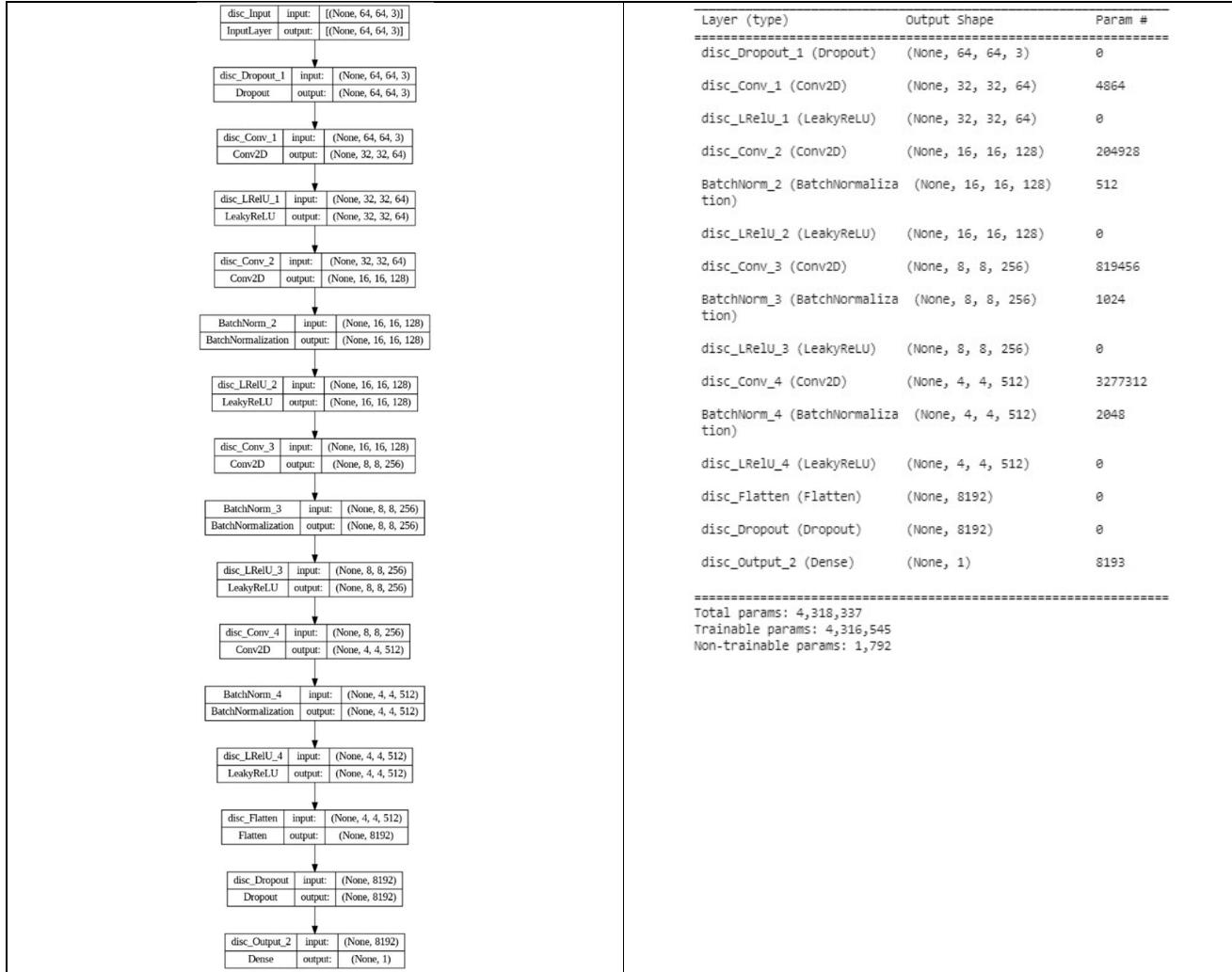
discriminator.add(layers.Dense(1,
 activation='sigmoid',
 input\_shape=(IMG\_COL, IMG\_ROW, IMG\_CHANNEL),
 name='disc\_Output\_2'))

```

discriminator.compile(loss='binary_crossentropy', optimizer=OPTIMIZER)
#-----#
return discriminator
#-----#

```

جزئیات لایه‌های این شبکه در شکل‌های زیر نشان داده شده است:



## GAN مدل

در نهایت با ترکیب دو مدل Generator و Discriminator یک شبکه GAN ایجاد می‌شود. کد زیر این شبکه را تعریف می‌کند:

```

#-- Create GAN--#
def Create_GAN(discriminator , generator):

    discriminator.trainable = False

    gan_input = layers.Input(shape=(NOISE_VEC_DIM,), name='gan_Input')
    fake_image = generator(gan_input)

    gan_output = discriminator(fake_image)

```

```

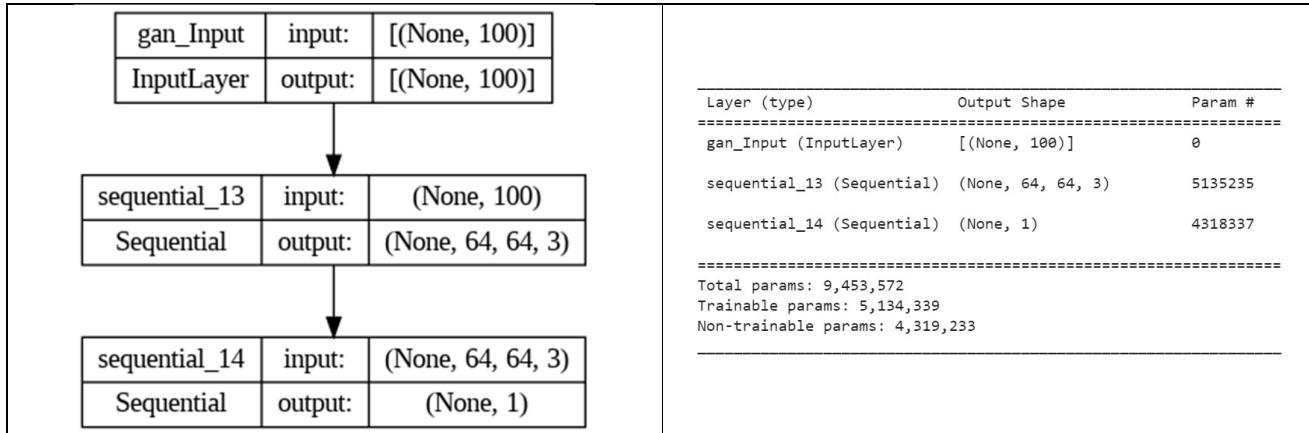
gan = Model(gan_input, gan_output)

gan.compile(loss='binary_crossentropy', optimizer=OPTIMIZER)

return gan
#-

```

شکل زیر ساختار این شبکه را نشان می‌دهد.



## 4. طراحی شبکه GAN – مدل دوم

در بخش دوم، از یک مدل متفاوت با مدل تعریف شده در لینک آموزش پروژه استفاده شده است. جزئیات این مدل در ادامه شرح داده می‌شود.

### Generator مدل

در اینجا هم مدل Generator یک بردار تصادفی با اندازه 100 را به عنوان ورودی دریافت می‌کند و پس از عبور از لایه‌های مختلف آن را به یک ماتریس سه بعدی  $64 \times 64 \times 3$  (یعنی یک تصویر رنگی با اندازه  $64 \times 64$ ) تبدیل می‌کند.

کد زیر این شبکه را تعریف می‌کند:

```

NOISE_VEC_DIM = 100
OPTIMIZER = optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
#-- Generator 2 -----
def Create_Generator_2():
    generator = Sequential()
    #
    d = 16
    generator.add(layers.Dense(d*d*256,
                               kernel_initializer=RandomNormal(0, 0.02),
                               input_dim=NOISE_VEC_DIM,
                               name = 'gen_dense'))

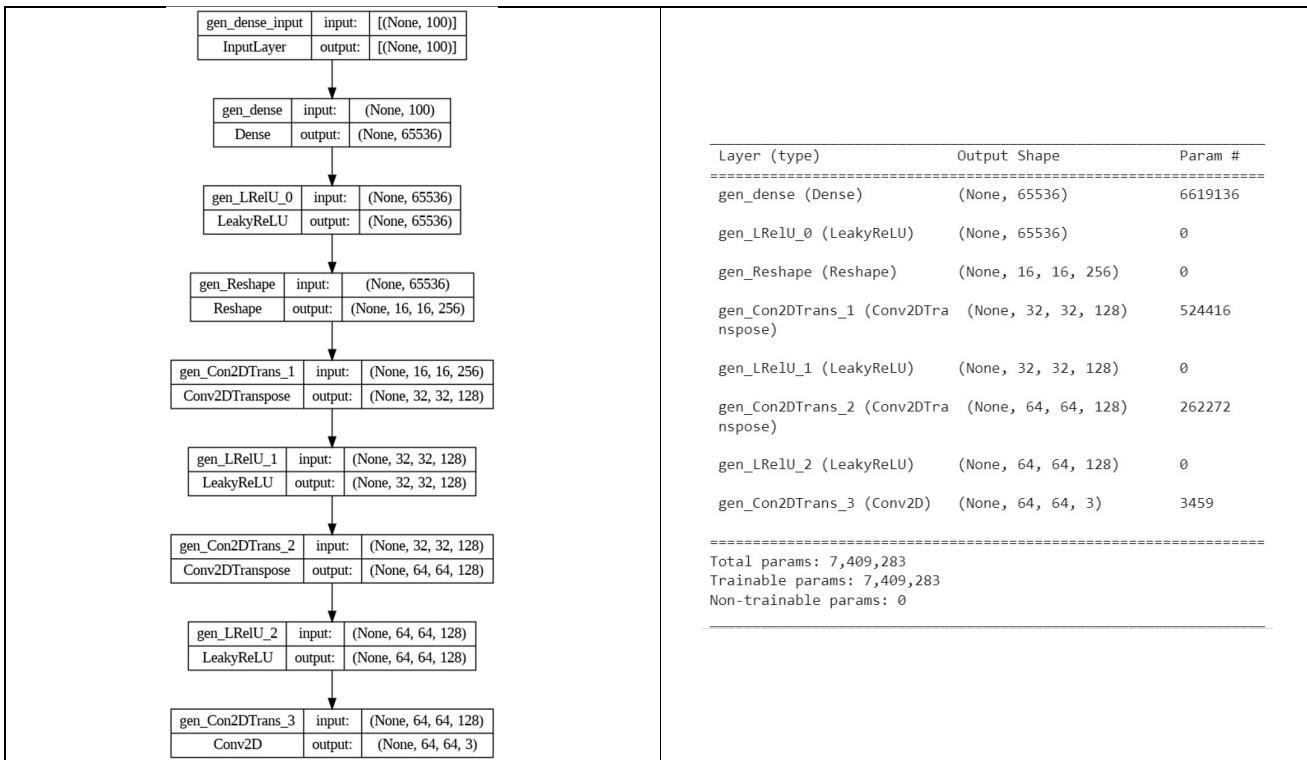
```

```

generator.add(layers.LeakyReLU(0.2,
                               name = 'gen_LReLU_0'))
print(generator.input_shape)
#-----#
generator.add(layers.Reshape((d, d, 256),
                               name='gen_Reshape'))
print(generator.output_shape)
#-----#
generator.add(layers.Conv2DTranspose(128, (4, 4),
                                     strides=2,
                                     padding='same',
                                     kernel_initializer=RandomNormal(0, 0.02),
                                     name = 'gen_Con2DTrans_1'))
generator.add(layers.LeakyReLU(0.2,
                               name = 'gen_LReLU_1'))
print(generator.output_shape)
#-----#
generator.add(layers.Conv2DTranspose(128, (4, 4),
                                     strides=2,
                                     padding='same',
                                     kernel_initializer=RandomNormal(0, 0.02),
                                     name = 'gen_Con2DTrans_2'))
generator.add(layers.LeakyReLU(0.2,
                               name = 'gen_LReLU_2'))
print(generator.output_shape)
#-----#
generator.add(layers.Conv2D(IMG_CHANNEL, (3, 3),
                           padding='same',
                           activation='tanh',
                           kernel_initializer=RandomNormal(0, 0.02),
                           name = 'gen_Con2DTrans_3'))
print(generator.output_shape)
#-----#
generator.compile(loss='binary_crossentropy', optimizer=OPTIMIZER)
#-----#
return generator

```

جزئیات ساختار و لایه‌های این شبکه در تصویر زیر نشان داده شده است:



## Discriminator مدل

این شبکه یک تصویر در اندازه  $64 \times 64$  را در ورودی دریافت می‌کند و جعلی یا واقعی بودن آن را در خروجی تشخیص می‌دهد. کد زیر این شبکه را تعریف می‌کند:

```

print(discriminator.output_shape)
#-----



discriminator.add(layers.Conv2D(128,
                               (3, 3),
                               strides=2,
                               padding='same',
                               kernel_initializer=RandomNormal(0, 0.02),
                               name= 'disc_Conv_3'))
discriminator.add(layers.LeakyReLU(0.2,
                                   name= 'disc_LReLU_3'))
print(discriminator.output_shape)
#-----



discriminator.add(layers.Conv2D(256,
                               (3, 3),
                               strides=2,
                               padding='same',
                               kernel_initializer=RandomNormal(0, 0.02),
                               name= 'disc_Conv_4'))
discriminator.add(layers.LeakyReLU(0.2,
                                   name= 'disc_LReLU_4'))
print(discriminator.output_shape)
#-----



discriminator.add(layers.Flatten(name='disc_Flatten'))
discriminator.add(layers.Dropout(0.4,
                                name = 'disc_Dropout'))
print(discriminator.output_shape)
#-----



discriminator.add(layers.Dense(1,
                               activation='sigmoid',
                               input_shape=(IMG_COL, IMG_ROW, IMG_CHANNEL),
                               name = 'disc_Output'))
print(discriminator.output_shape)
#-----



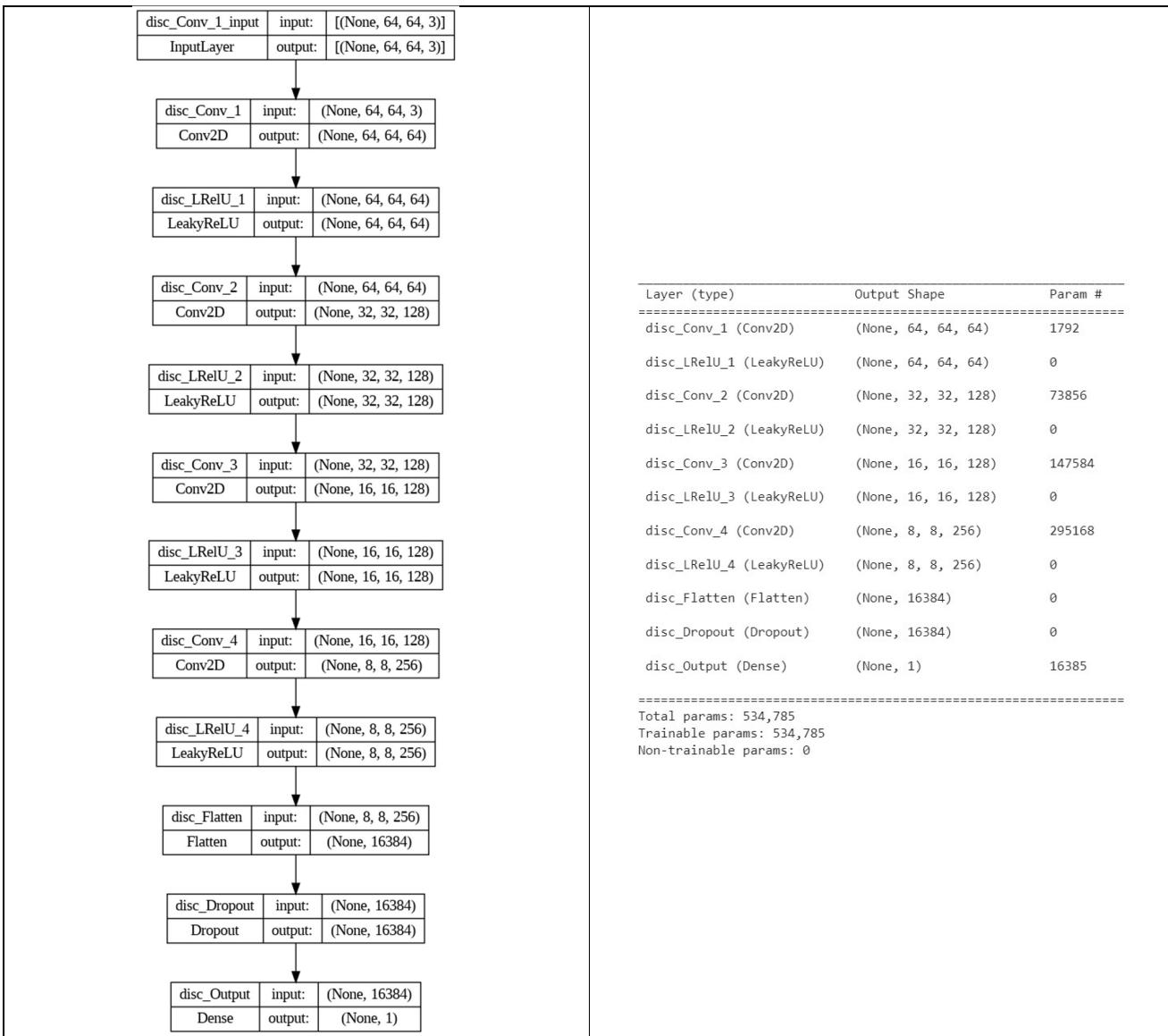
discriminator.compile(loss='binary_crossentropy', optimizer=OPTIMIZER)
#-----



return discriminator

```

جزئیات ساختار لایه‌های این شبکه در تصویر زیر نشان داده شده است:



## GAN مدل

در نهایت با ترکیب دو مدل Generator و Discriminator یک شبکه GAN ایجاد می‌شود. کد زیر این شبکه را تعریف می‌کند:

```
--- Create GAN ---
def Create_GAN(discriminator , generator):

    discriminator.trainable = False

    gan_input = layers.Input(shape=(NOISE_VEC_DIM,) , name='gan_Input')
    fake_image = generator(gan_input)

    gan_output = discriminator(fake_image)

    gan = Model(gan_input, gan_output)

    gan.compile(loss='binary_crossentropy' , optimizer=OPTIMIZER)
```

```
    return gan
#-----
```

## 5. توابع کمکی برای رسم تصاویر و نمودارها

### نمایش تصاویر تولید شده توسط Generator

تابع زیر در انتهای هر epoch مشخص، یک بردار تصادفی را به مدل Generator می‌دهد و تصویر تولید شده توسط آن را نمایش می‌دهد. (در واقع این تصویر لیستی از بردارهای تصادفی را به generator ارسال می‌کند و به ازای هر بردار تصویر تولید شده متناظر با آن را رسم می‌کند).

علاوه بر این در همین کد، تصاویر تولید شده در فایل log ذخیره می‌شوند تا در تنسوربورد نمایش داده شوند. (توضیحات مربوط به تنسوربورد در بخش 6 ذکر شده است)

```
--- Plot Images-----
def Plot_Images(noise, size_fig, generator, epoch):

    --- get generated images by generator ---
    generated_images = generator.predict(noise)

    --- create a figure ---
    plt.figure(figsize=size_fig)

    for i, image in enumerate(generated_images):
        plt.subplot(size_fig[0], size_fig[1], i+1)
        if IMG_CHANNEL == 1:
            image = image.reshape((IMG_ROW, IMG_COL))
            plt.imshow(image, cmap='gray')
        else:
            image = image.reshape((IMG_ROW, IMG_COL, IMG_CHANNEL))
            plt.imshow(image)

    Write_Log_Images(image , epoch)

    plt.axis('off')

    plt.tight_layout()
    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch+1))
    plt.show()
#-----
```

### رسم نمودار خطی

تابع زیر میزان خطای مدل‌های Generator و Discriminator در طی آموزش را رسم می‌کند.

```
--- Plot Losses-----
def Plot_Loss(d_losses, g_losses):
```

```

plt.rcParams.update({'font.size': 22})

d_losses = np.round(d_losses , 3)
g_losses = np.round(g_losses , 3)

x_plot = range(1, len(d_losses)+1)

plt.figure(figsize=(7,4))

plt.plot(x_plot, d_losses , label= 'discriminator')

plt.plot(x_plot , g_losses, label = 'generator')

plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.grid()
plt.show()

```

## 6. تنسوربورد

### فعالسازی تنسوربورد

برای فعالسازی تنسوربورد در محیط Colab ابتدا دستور زیر اجرا می‌شود:

```
%load_ext tensorboard
```

سپس شیئی از نوع Tensorboard ایجاد و پارامترهای مورد نیاز تنظیم می‌شوند:

```

log_path = '/tmp/my_logs'

#-- Clear any logs from previous runs --
!rm -rf ./tmp/my_logs

#-- Create the TensorBoard --
tensorboard = TensorBoard(
    log_dir= log_path,
    histogram_freq=0,
    write_graph=True,
)

#-- set gan for tensroboard model --
tensorboard.set_model(gan)
#-----
```

### ذخیره داده‌های مورد نظر در فایل log

به منظور ذخیره مقادیر خطا در حین فرایند آموزش و تصاویر تولید شده در حین آموزش از دوتابع زیر استفاده می‌شود:

```

#-- Write Losses in Log file for Tensorboard -----
def Write_Log_Loss(names, logs, batch_no):
    writer = SummaryWriter()
    for name, value in zip(names, logs):
        writer.add_scalar(name, value, batch_no)
    writer.flush()
#-----
#-- Save images in Log file for Tensorboard -----
def Write_Log_Images(image , epoch):
    writer = SummaryWriter()
    writer.add_image('fake_image',image,epoch+1, dataformats='HWC')
    writer.close()
#-----

```

این دو تابع در حین آموزش مدل فراخوانی می‌شوند.

## اجرای تنسوربورد

برای اجرای تنسوربورد و مشاهده نتایج دستور زیر در محیط Colab اجرا می‌شود:

```
%tensorboard --logdir /tmp/my_logs
```

## 7. آموزش مدل

ابتدا یک شبکه GAN با ترکیب دو مدل Generator و Discriminator ایجاد می‌شود:

```

#-- Create GAN Model 1 -----
#-- Create generator --
generator = Create_Generator_1()

#-- Models Summery--generator --
plot_model(generator, show_shapes=True,
            show_layer_names=True, to_file='generator.png')
Image(retina=True, filename='generator.png')
generator.summary()

#-- Create discriminator --
discriminator = Create_Discriminator_1()

#-- Models Summery--discriminator --
plot_model(discriminator, show_shapes=True,
            show_layer_names=True, to_file='discriminator.png')
Image(retina=True, filename='discriminator.png')
discriminator.summary()

#-- Create GAN --
gan = Create_GAN(discriminator, generator)

```

```

-- Models Summery--gan --
plot_model(gan, show_shapes=True,
           show_layer_names=True, to_file='gan.png')
Image(retina=True, filename='gan.png')
gan.summary()
#-----

```

سپس فرایند آموزش روی مدل ایجاد شده، اجرا می‌شود:

```

-- Train --
d_losses = []
g_losses = []

for epoch in range(EPOCHS):
    for batch in range(STEPS_PER_EPOCH):

        #-- Create random vector --
        noise = np.random.normal(0, 1, size=(BATCH_SIZE, NOISE_VEC_DIM))

        #-- Generate fake images by generator --
        fake_x = generator.predict(noise)

        #-- Get real images
        real_x = X_train[np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)]

        #-- Concat real images and fake images --
        x = np.concatenate((real_x, fake_x))

        #-- set true output for real images and fake images --
        disc_y = np.zeros(2*BATCH_SIZE)
        disc_y[:BATCH_SIZE] = 0.9

        #-- train discriminator --
        d_loss = discriminator.train_on_batch(x, disc_y)

        #-- set true output for generator --
        y_gen = np.ones(BATCH_SIZE)

        #-- train gan --
        g_loss = gan.train_on_batch(noise, y_gen)

        #-- Save Logs for tensorboard on each batch--
        Write_Log_Loss(['g_loss'], [g_loss], batch)
        Write_Log_Loss(['d_loss'], [d_loss], batch)

        #-- save losses for each epoch--
        d_losses.append(d_loss)
        g_losses.append(g_loss)

        #-- Print losses for each epoch
        print(f'Epoch: {epoch + 1} \t Discriminator Loss: {d_loss} \t\t Generator Loss: {g_loss}')

```

```

-- Plot Generated Images by Generator for each epoch --
if epoch==0 or (epoch+1%10)==0:
    noise = np.random.normal(0, 1, size=(25, NOISE_VEC_DIM))
    Plot_Images(noise, (5, 5), generator, epoch)

-- Save weights for each epoch --
if epoch==0 or (epoch+1%100)==0:
    weights_file = 'gan_weights_{:04d}.h5'.format(epoch+1)
    gan.save_weights(weights_file)

Plot_Loss(d_losses, g_losses)

```

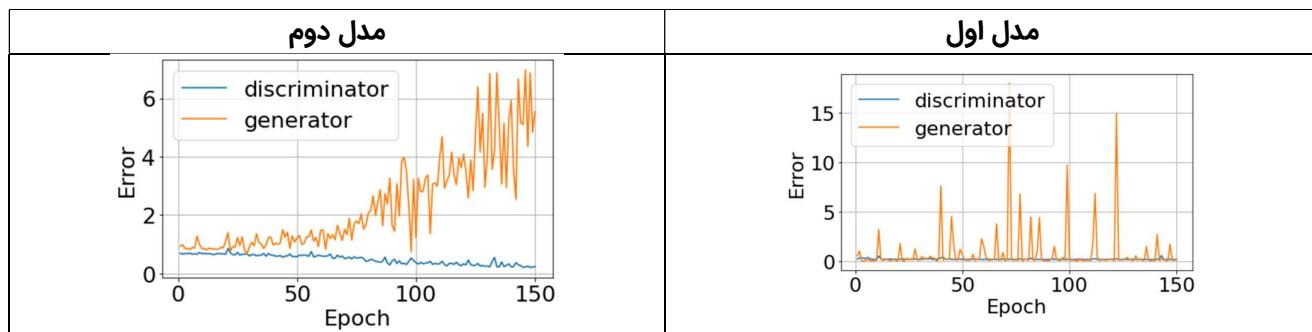
## 8. نتایج اجرا - مدل اول

در این بخش نتایج حاصل از اجرای هر دو مدل نشان داده شده است.

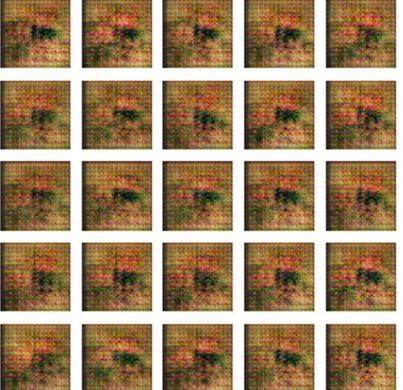
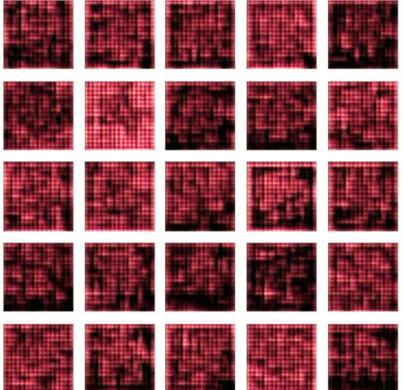
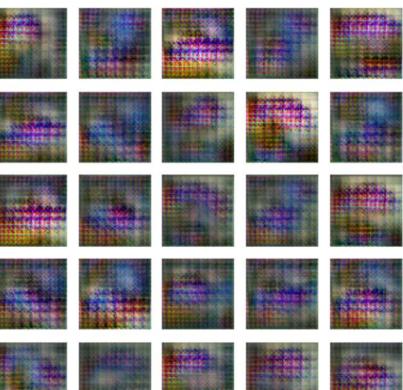
همانطور که در تصاویر زیر مشخص است به طور کلی مدل دوم (که ساختار آن در بخش 4 شرح داده شد) عملکرد بهتری نسبت به مدل اول (که ساختار آن در بخش 3 شرح داده شد) دارد. علاوه بر اینکه میزان خطای Generator در مدل دوم کمتر است، تصاویر تولید شده نیز کیفیت بهتری دارند. البته بر اساس نمودار خطای حدود 60 تکرار برای آموزش این مدل کافی است و پس از آن مدل دچار overfit میشود که این در تصاویر تولید شده نیز مشخص است و تصاویر بعد از تکرار 60 دچار افت کیفیت شده اند.

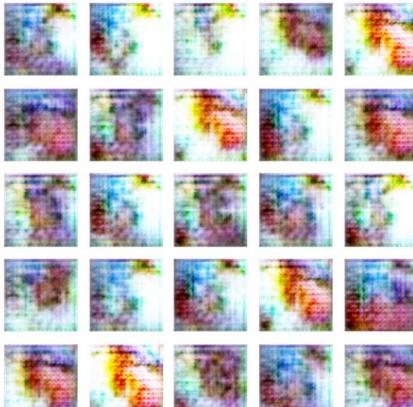
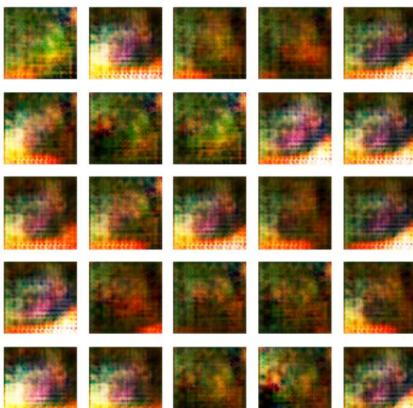
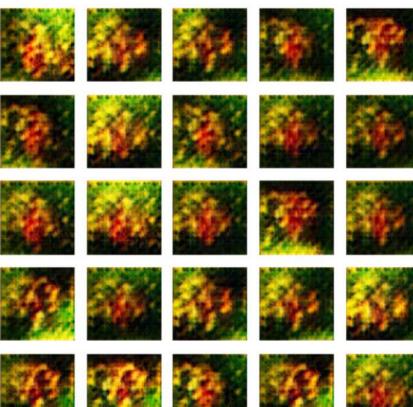
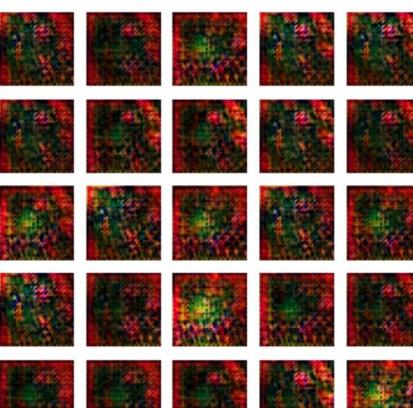
البته متناسبانه هیچ یک از مدل، در دستیابی به کیفیت مورد نظر و تولید تصاویر نسبتاً واضح از انواع گلهای، عملکرد خیلی خوبی نداشتند.

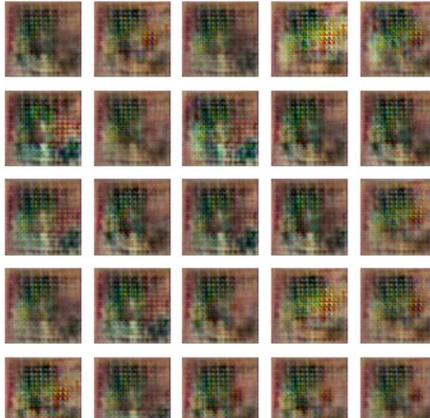
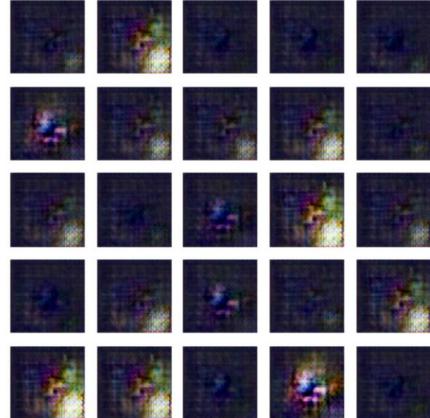
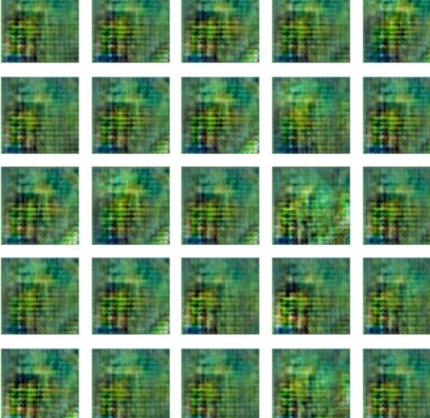
**نمودار خطای**

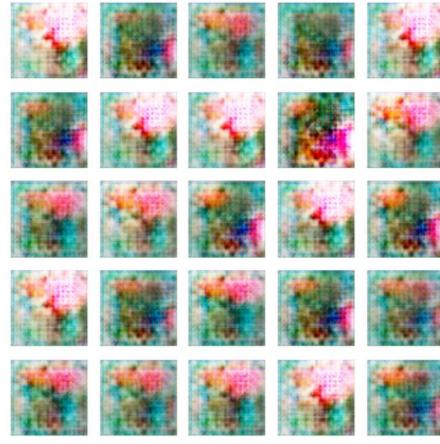
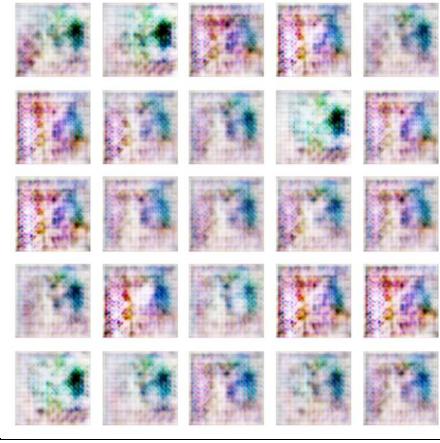
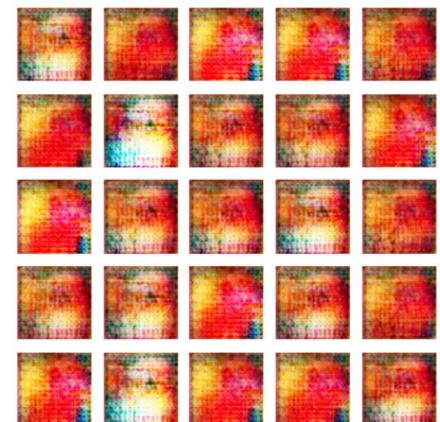


### تصاویر تولید شده در epoch های مختلف

epoch	مدل اول	مدل دوم
1		
10		
20		
30		

		40
		50
		60
		70

		80
		90
		100
		110

		120
		130
		140
		150