

بسم الله الرحمن الرحيم

نام دانشجو: سید محمد علی رضایی

شماره دانشجویی: 400131020

استاد درس: دکتر صفابخش

گزارش تمرین شماره 7

کدهای گزارش در فایل زیپ موجود می باشد.

سوال یک :

اغلب شبکه های مولد تقابلی حداقل به طور کلی بر پایه معماری DCGAN هستند. معماری مدل GAN شامل دو زیر مدل است: یک مدل مولد برای تولید نمونه های جدید و یک مدل متمایزگر برای دسته بندی نمونه های تولید شده در دو دسته واقعی از دامنه و جعلی که توسط مدل مولد ساخته شده است. مولد: مدلی که برای تولید مثال های قابل باور جدید از دامنه مسأله مورد استفاده قرار می گیرد. متمایزگر: مدلی که برای دسته بندی مثال ها به عنوان واقعی (از دامنه) یا جعلی (ساخته شده توسط مدل مولد) مورد استفاده قرار می گیرد. 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. فضای برداری با عنوان فضای پنهان (Latent Space) یا یک فضای برداری متشکل از متغیرهای پنهان (Latent Variables) نامیده می شود. متغیرهای پنهان (Latent Variables) یا (Hidden Variables) متغیرهایی هستند که برای دامنه حائز اهمیت هستند اما به طور مستقیم قابل مشاهده نیستند. معمولاً از متغیرهای پنهان یا فضای پنهان با عنوان تصویر (Projection) یا فشرده سازی توزیع داده یاد می شود. این مورد، فضای پنهانی است که فشرده سازی یا مفهوم سطح بالا از داده های خام مشاهده شده مانند توزیع داده های ورودی را فراهم می کند. در GAN، 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.

مدل متمایزگر، نمونه ای (Example) 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 طبیعی (و به خوبی درک شده) است. پس از فرایند آموزش، مدل متمایزگر کنار گذاشته می شود، زیرا مدل مولد است که جذابیت دارد. گاهی اوقات، مولد را می توان باز هدف گذاری کرد، زیرا به خوبی یاد گرفته است که ویژگی ها را در دامنه مسأله استخراج کند. برخی یا تمام لایه های استخراج ویژگی را می توان در کاربردهای یادگیری انتقال (Transfer Learning) با استفاده از همان داده ها یا داده های مشابه استفاده کرد.

مدل سازی مولد از جمله مسائل یادگیری نظارت نشده است. 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

نمونه‌ها را آماده می‌کند و این دسته، همراه با مثال‌هایی از دامنه، برای متمایزگر فراهم و به عنوان واقعی (real) یا جعلی (fake) دسته‌بندی می‌شوند. مولد سپس به روز رسانی می‌شود تا در تمایز بین نمونه‌های واقعی و جعلی در دور بعدی بهتر عمل کند. مهم‌تر آنکه، مولد نیز بر پایه اینکه متمایزگر با نمونه تولید شده به اندازه کافی خوب گول خورده است یا خیر نیز به روز رسانی می‌شود.

با توجه به توضیحات فوق، آموزش شبکه مشابه یک بازی MIN_Max است، بدین صورت که مولد و متمایزگر به صورت توأم آموزش داده می‌شوند اما به صورت نوبتی، یعنی در ابتدا متمایزگر آموزش می‌بیند و پس چند دور آموزش پارامترهای آن ثابت شده و این بار مولد آموزش می‌بیند و تابع هدف این شبکه به صورت زیر می‌باشد.

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \underbrace{\log D_{\theta_d}(x)}_{\text{Discriminator output for real data } x} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data } G(z)}) \right]$$

در این رابطه متمایزگر به دنبال بیشینه و مولد به دنبال کمینه کردن این تابع می‌باشند.

سوال دو :

عملکرد لایه‌ی معکوس کانولوشن، که آن را دکانولوشن نیز می‌نامند، دقیقاً عکس عملکرد کانولوشن استاندارد است، بدین صورت که اگر ما یک تصویر را به عنوان ورودی داشته باشیم و با استفاده از فیلترها یک نقشه ویژگی تولید کرده باشیم حال در دکانولوشن به دنبال عکس این اتفاق هستیم، و با استفاده از نقشه ویژگی می‌خواهیم تصویری را در خروجی داشته باشیم. یا به عبارتی دیگر با استفاده از یک نقشه ویژگی در ورودی به دنبال نقشه ویژگی در خروجی هستیم که ابعاد بیشتری داشته باشد. مشابه عملیاتی که در کانولوشن عادی داشتیم padding and stride به صورت فرضی در نظر گرفته می‌شوند یعنی بر اساس این دو ما از روی تصویر به نقشه ویژگی که ورودی لایه دکانولوشن است رسیده‌ایم. پس با استفاده از این دو به دنبال پیدا کردن نقشه ویژگی بزرگ‌تر می‌باشیم. یعنی اگر خروجی دکانولوشن را بر روی آن‌ها با استفاده از padding and stride که تعریف کرده‌ایم پیاده کنیم به ورودی خواهیم رسید.

نحوه محاسبه مقدار خروجی لایه دکانولوشن : در ابتدا بین هر مقدار در نقشه ویژگی صفر می‌گذاریم که این عمل سایز ورودی را به

$$p^{\wedge} = \text{Input_shape} * 2 - 1 \quad \text{تبدیل می‌کند. سپس با استفاده از رابطه } p^{\wedge} = \text{kernel size} - p - 1 \text{ و}$$

اندازه کرنل موجود کانولوشن استاندارد را اجرا کرده و خروجی را بدست می‌آوریم که این خروجی نهایی لایه دکانولوشن می‌باشد.

سوال ۳ :

تابع استاندارد هزینه برای شبکه GAN، به اصطلاح ان را minimax loss می‌گویند که رابطه ان مطابق زیر می‌باشد:

$$E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$$

که در این رابطه مولد به دنبال کمینه کردن این تابع و متمایزگر به دنبال بیشینه کردن این تابع می‌باشد همانند بازی minmax

همچنین تابع استاندارد هزینه برای شبکه Gan را نیز می‌توان به دو قسمت برای شبکه مولد و متمایزگر تقسیم بندی کرد:

برای بخش متمایزگر :

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log(1 - D(G(\mathbf{z}^{(i)}))) \right]$$

تمایزگر به دنبال بیشینه کردن این تابع می‌باشد در زمانی که بین داده‌های واقعی و داده‌های مجازی تولید شده توسط مولد نتواند وجه تمایزی قائل شود.

تابع هزینه برای بخش مولد که مولد به دنبال کمینه کردن آن می باشد عبارت است از :

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D \left(G \left(\mathbf{z}^{(i)} \right) \right) \right)$$

تابع هزینه مولد از روی مقدار قدرت دسته بندی بخش متمایزگر بدست می‌آید بدین گونه که مولد به دنبال گمراه کردن متمایزگر می‌باشد و سعی در تولید داده‌های مجازی را به گونه ای دارد که متمایزگر آن‌ها را واقعی تلقی کند.

سوال ۴ :

پیاده سازی شبکه DCGAN :

در ابتدا به فراخوانی دیتا ست موجود و پیش پردازش های لازم بر روی آن می پردازیم :

```
1 from logging import info
2 setattr(tfds.image_classification.cats_vs_dogs, '_URL', "https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA77B919F/kagglecatsanddogs_93628.png")
3 train, validation, test = tfds.load('cats_vs_dogs', split=['train[:70%]', 'train[70%:80%]', 'train[80%:]'], shuffle_files=True, as_supervised=True)
```

با توجه به این که سایز تصاویر ورودی در دیتا ست متفاوت می باشد با استفاده از بلاک کد زیر سایز تصاویر را برای کل دیتا ست ثابت کرده و سپس بر ۲۵۵ تقسیم می کنیم تا مقدار عددی آن بین ۰ تا ۱ تغییر کند.

```
1 def normalization(img, label):
2     temp = (tf.image.resize(img, (output_dim, output_dim))) / 255
3     return temp
4 output_dim = 56
5 train = train.map(normalization)
```

مجموعه داده در دسترس را به مجموعه آموزش و اعتبار سنجی و تست تقسیم بندی کردیم.

ساختار داده های آموزش به صورت زیر است:

```
[ ] 1 train.shape
(16283, 56, 56, 3)
```

که در این جا ما ۱۶۲۸۳ تصویر سه کاناله با ابعاد ۵۶ در ۵۶ داریم.

نمایش کوچکی از تصاویر موجود در دیتا ست:

همان طور که مشاهده می شود تصاویر مربوط به دو کلاس تصویر از سگ ها و گربه ها می باشد. که در این تمرین قصد داریم با استفاده

از این تصاویر در شبکه مولد رقابتی به تولید تصاویری مجازی شبیه به تصاویر موجود در این دیتا ست کنیم.



ساختار شبکه DCGAN از دو بخش مولد و متمایزگر تشکیل شده است در بخش مولد در ابتدا با استفاده از بردار نویز ورودی و لایه‌های دکانولوشن به تولید تصاویری مجازی می‌نماییم سپس این تصاویر را به عنوان مقدار ورودی به متمایزگر می‌دهیم. در بخش متمایزگر که خود دارای دو ورودی است: اول ورودی را از مولد می‌گیرد و دوم آن که ورودی را از داده‌های واقعی دریافت می‌کند و سعی می‌کند با استفاده از داده‌های واقعی تشخیص دهد که داده‌های ورودی از بخش مولد واقعی هستند یا خیر و در اصل بخش مولد به دنبال گمراه کردن بخش متمایزگر می‌باشد.

پیاده سازی بخش اول شبکه(قسمت مولد):

این ساختار بر اساس ساختار معرفی شده در کتاب یادگیری عمیق نویسنده : فرانسوا شوله می‌باشد.

```
1 latent_dim = 56
2 height = 56
3 width = 56
4 channels = 3
5 generator_input = keras.Input(shape=(latent_dim,))
6 x = layers.Dense(128 * 28 * 28)(generator_input)
7 x = layers.LeakyReLU()(x)
8 x = layers.Reshape((28, 28, 128))(x)
9 x = layers.Conv2D(256, 5, padding='same')(x)
10 x = layers.LeakyReLU()(x)
11 x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
12 x = layers.LeakyReLU()(x)
13 x = layers.Conv2D(256, 5, padding='same')(x)
14 x = layers.LeakyReLU()(x)
15 x = layers.Conv2D(256, 5, padding='same')(x)
16 x = layers.LeakyReLU()(x)
17 x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
18 generator = keras.models.Model(generator_input, x)
19 generator.summary()
```

تصویر زیر خروجی هر لایه و تعداد کل پارامترهای قابل یادگیری این مولد را نشان می‌دهد.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 56)]	0
dense (Dense)	(None, 100352)	5720064
leaky_re_lu (LeakyReLU)	(None, 100352)	0
reshape (Reshape)	(None, 28, 28, 128)	0
conv2d (Conv2D)	(None, 28, 28, 256)	819456
leaky_re_lu_1 (LeakyReLU)	(None, 28, 28, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 56, 56, 256)	1048832
leaky_re_lu_2 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_1 (Conv2D)	(None, 56, 56, 256)	1638656
leaky_re_lu_3 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_2 (Conv2D)	(None, 56, 56, 256)	1638656
leaky_re_lu_4 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_3 (Conv2D)	(None, 56, 56, 3)	37635
=====		
Total params: 10,903,299		
Trainable params: 10,903,299		
Non-trainable params: 0		

در این ساختار ورودی یک داده رندوم با ابعاد (۵۶،۱) خواهد بود و در خروجی یک تصویر با ابعاد (۵۶،۵۶،۳) خواهیم داشت.

تعداد کل پارامترهای قابل یادگیری در این مولد در حدود ۱۱ میلیون می‌باشد. که در این مولد از لایه‌های کانولوشن برای استخراج ویژگی و همچنین از لایه دکانولوشن برای انجام عمل up sampling استفاده شده است.


```

1 discriminator_input = layers.Input(shape=(height, width, channels))
2 x = layers.Conv2D(128, 3)(discriminator_input)
3 x = layers.LeakyReLU()(x)
4 x = layers.Conv2D(128, 4, strides=2)(x)
5 x = layers.LeakyReLU()(x)
6 x = layers.Conv2D(128, 4, strides=2)(x)
7 x = layers.LeakyReLU()(x)
8 x = layers.Conv2D(128, 4, strides=2)(x)
9 x = layers.LeakyReLU()(x)
10 x = layers.Flatten()(x)
11 x = layers.Dropout(0.4)(x)
12 x = layers.Dense(1, activation='sigmoid')(x)
13 discriminator = keras.models.Model(discriminator_input, x)
14 discriminator.summary()
15 discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-8)
16 discriminator.compile(optimizer=discriminator_optimizer, loss='binary_crossentropy')

```

ورودی این شبکه در اصل خروجی بخش مولد می‌باشد پس ورودی آن به صورت (۵۶،۵۶،۳) خواهد بود و در لایه‌های مخفی آن با استفاده از کانولوشن دو بعدی با تعداد متفاوت کرنل و استفاده از تابع فعال ساز leaky RELU، ورودی را به یک بردار یک بعدی به اندازه ۳۲۰۰ تبدیل کرده و در نهایت هم با استفاده از یک لایه dense با تابع فعال ساز سیگموید به عنوان یک مسئله دسته بندی که دو کلاس حقیقی و مجازی به این بررسی می‌پردازیم که آیا تصاویر واقعی بوده اند یا مجازی.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 56, 56, 3)]	0
conv2d_4 (Conv2D)	(None, 54, 54, 128)	3584
leaky_re_lu_5 (LeakyReLU)	(None, 54, 54, 128)	0
conv2d_5 (Conv2D)	(None, 26, 26, 128)	262272
leaky_re_lu_6 (LeakyReLU)	(None, 26, 26, 128)	0
conv2d_6 (Conv2D)	(None, 12, 12, 128)	262272
leaky_re_lu_7 (LeakyReLU)	(None, 12, 12, 128)	0
conv2d_7 (Conv2D)	(None, 5, 5, 128)	262272
leaky_re_lu_8 (LeakyReLU)	(None, 5, 5, 128)	0
flatten (Flatten)	(None, 3200)	0
dropout (Dropout)	(None, 3200)	0
dense_1 (Dense)	(None, 1)	3201
Total params: 793,601		
Trainable params: 793,601		
Non-trainable params: 0		

حال نوبت به ساخت مدل GAN مورد نظر خود رسیده است، در این بخش ابتدا پارامترهای متمایزگر را به توصیه کتاب freeze میکنیم،

به این علت که اگر این کار را در ابتدا انجام ندهیم آن وقت متمایزگر سعی بر این دارد تا تمامی داده ها را real دسته بندی کند.

ورودی مدل گن مورد نظر هم ابعاد با ورودی بخش مولد خواهد بود و خروجی آن، حاصل مقدار خروجی متمایزگر به ازاء ورودی از بخش مولد میباشد.

```
1 discriminator.trainable = False
2 gan_input = keras.Input(shape=(latent_dim,))
3 gan_output = discriminator(generator(gan_input))
4 gan = keras.models.Model(gan_input, gan_output)
5 gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
6 gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

Model: "model_3"

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 56)]	0
model (Functional)	(None, 56, 56, 3)	10903299
model_1 (Functional)	(None, 1)	793601

=====
Total params: 11,696,900
Trainable params: 10,903,299
Non-trainable params: 793,601

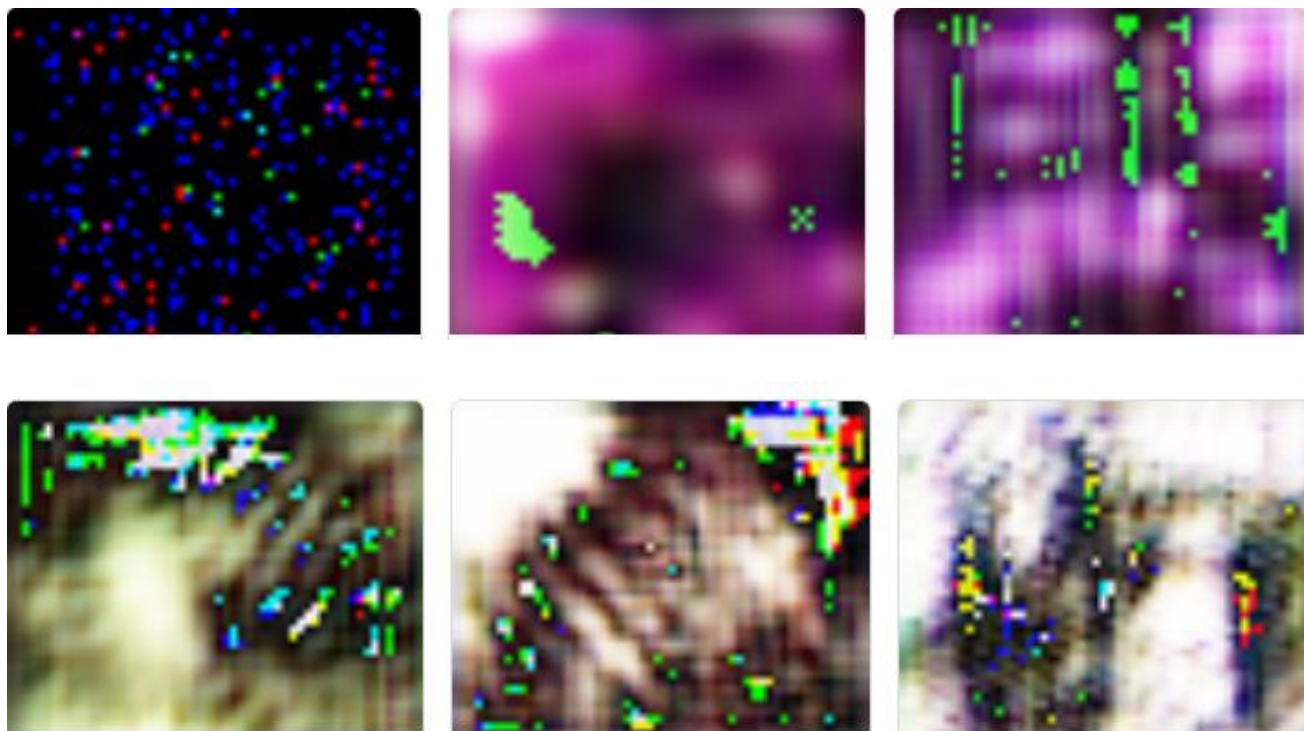
با توجه به ساختار فوق و پارامترهای تنظیم شده شروع به آموزش شبکه و تولید تصاویر با ۱۰۰۰ دور آموزش می‌کنیم، توجه شود به ازاء هر ۱۰۰ دور آموزش یک تصویر را در خروجی نمایش گرفته ایم.

در این آموزش ملاک ارزیابی تابع loss برای مولد و مولد میباشد در این جا به دنبال این هستیم تا مقدار تابع هزینه متمایزگر را افزایش دهیم و مقدار تابع هزینه مولد کاهش پیدا کند.

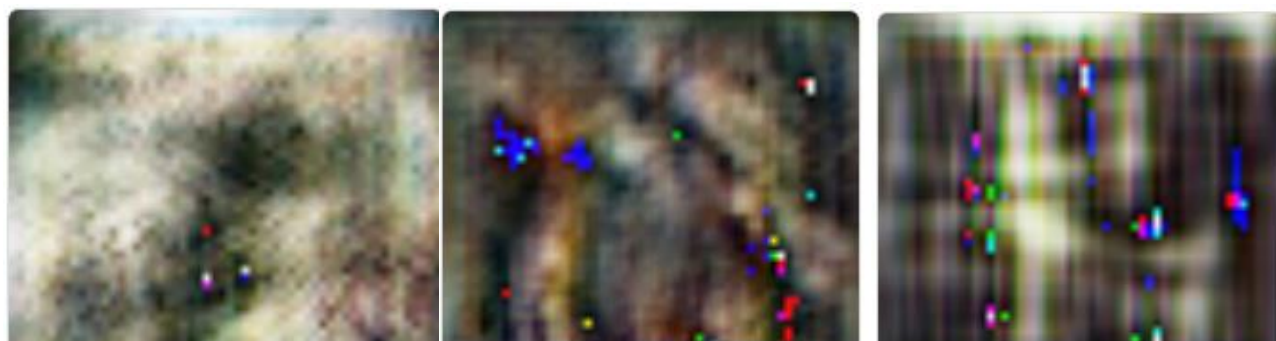
پس از ده دور آموزش مقدار loss برای متمایزگر و مولد برابر است با :

```
discriminator loss: 0.696391224861145
adversarial loss: 0.7145578265190125
```

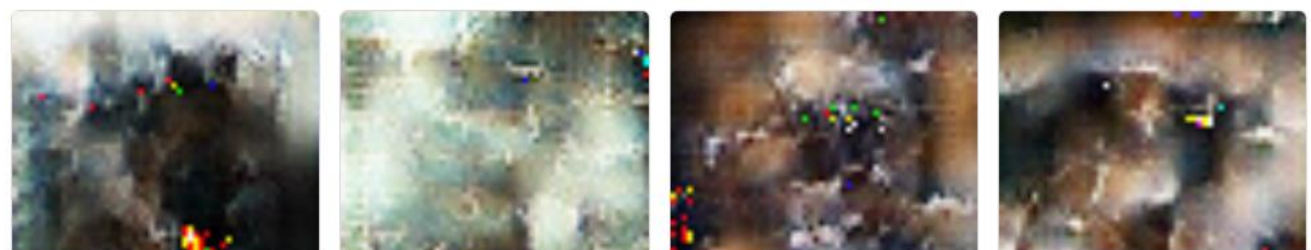
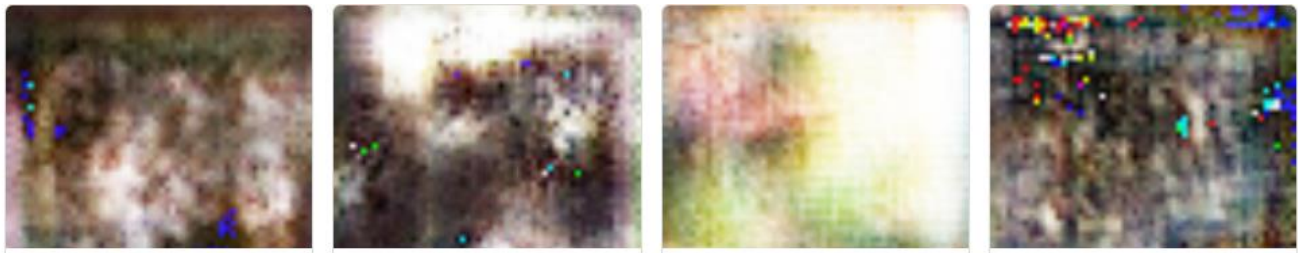
در زیر نمایش برخی از تصاویر تولید شده را داریم :




همان طور که مشاهده می شود تصاویر به شدت در ابتدا نویزی بوده است اما با گذشت روند آموزش تا حدودی توانسته شبکه تا تصویری از حاله یک سگ یا گربه را نمایش دهد مانند :




حال این بار شبکه را با تعداد دور بیشتری آموزش می دهیم: (۱۰۰۰۰) :

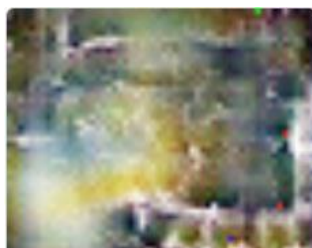


 generated4800.png

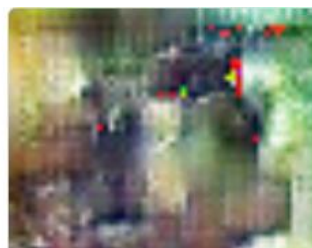
 generated4900.png

 generated5000.png

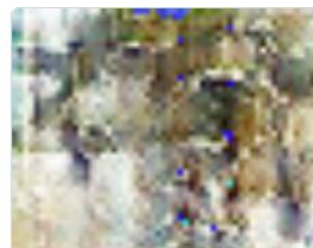
 generated5100.png



 generated5600.png



 generated5700.png



 generated5800.png



 generated5900.png

پس از حدود ۱۰۰۰۰ دور آموزش مقدار تابع هزینه برای مولد و متمایزگر برابر است با :

```
discriminator loss: 0.7021788358688354
adversarial loss: 0.2691323161125183
```

حال این بار تعداد لایه های موجود در بخش مولد شبکه مذکور را افزایش می دهیم ولی برای جلوگیری از افزایش چشم گیر پارامتر ها و محدودیتی که در اجرای شبکه به خاطر سخت افزار داریم تعداد پارامتر ها رو کاهش دادیم و در اصل شبکه را عمیق تر و کشیده تر کردیم و پهنای شبکه را کاهش دادیم :

input_8 (InputLayer)	[(None, 56)]	0
dense_7 (Dense)	(None, 12544)	715008
leaky_re_lu_37 (LeakyReLU)	(None, 12544)	0
reshape_7 (Reshape)	(None, 14, 14, 64)	0
conv2d_29 (Conv2D)	(None, 14, 14, 64)	102464
leaky_re_lu_38 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_30 (Conv2D)	(None, 14, 14, 128)	204928
leaky_re_lu_39 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_7 (Conv2DT ranspose)	(None, 28, 28, 128)	262272
leaky_re_lu_40 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_transpose_8 (Conv2DT ranspose)	(None, 56, 56, 256)	524544
leaky_re_lu_41 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_31 (Conv2D)	(None, 56, 56, 256)	1638656
leaky_re_lu_42 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_32 (Conv2D)	(None, 56, 56, 256)	1638656
leaky_re_lu_43 (LeakyReLU)	(None, 56, 56, 256)	0
conv2d_33 (Conv2D)	(None, 56, 56, 3)	37635

شبکه را با ۵۰۰ دور آموزش می‌دهیم :



generated400.png



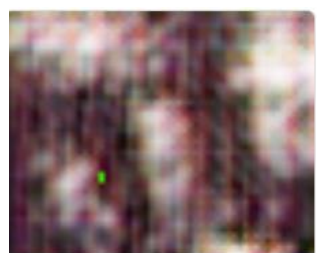
generated500.png



generated600.png



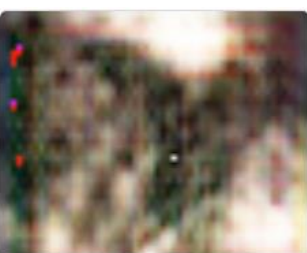
generated700.png



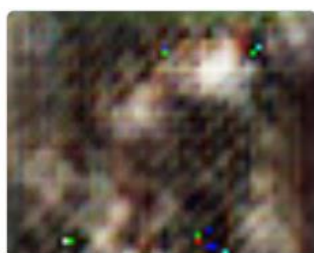
generated800.png



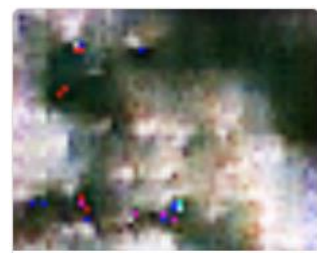
generated900.png



generated1000.png



generated1100.png



generated4400.png



generated4500.png



generated4600.png



generated4700.png

افزایش عمق شبکه با توجه به ساختار پیچیده شبکه به یادگیری شبکه کمک میکند اما در اینجا چون پارامترها رو نسبت به قبل کاهش

داده ام پس تفاوت چندانی در خروجی تصاویر نداشته ایم .

مقدار loss در ۵۰۰ دور آخر آموزش برای مولد و متمایزگر به شرح زیر است:

```
discriminator loss: 0.6715945601463318
adversarial loss: 0.8526541590690613
discriminator loss: 0.6699644923210144
adversarial loss: 1.4593076705932617
discriminator loss: 0.6498201489448547
adversarial loss: 0.8045040369033813
discriminator loss: 0.6847512125968933
adversarial loss: 0.8304470777511597
discriminator loss: 0.7782961130142212
adversarial loss: 0.6089445352554321
discriminator loss: 0.7337358593940735
adversarial loss: 1.0661375522613525
```

حال در این قسمت به پیاده سازی شبکه FCGAN می پردازیم :

ساختار این شبکه مشابه ساختار شبکه قبل می باشد و تفاوتی که در این ساختار دارد به جای استفاده از لایه های کانولوشنی در بخش

مول از لایه های تمام متصل استفاده شده است با توجه به بلاک کد زیر این ساختار را پیاده سازی می کنیم :

FCGAN

```
[ ] 1 optimizer = Adam(0.0002, 0.5)
    2 def create_generator():
    3     generator = Sequential()
    4
    5     generator.add(Dense(256, input_dim=56))
    6     generator.add(LeakyReLU(0.2))
    7
    8     generator.add(Dense(512))
    9     generator.add(LeakyReLU(0.2))
    10
    11    generator.add(Dense(1024))
    12    generator.add(LeakyReLU(0.2))
    13
    14    generator.add(Dense(56*56*3, activation='tanh'))
    15    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    16    return generator
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	14592
leaky_re_lu (LeakyReLU)	(None, 256)	0
dense_1 (Dense)	(None, 512)	131584
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
dense_2 (Dense)	(None, 1024)	525312
leaky_re_lu_2 (LeakyReLU)	(None, 1024)	0
dense_3 (Dense)	(None, 9408)	9643200

```
=====
Total params: 10,314,688
Trainable params: 10,314,688
Non-trainable params: 0
```


مولد این شبکه دارای یک ساختار sequential می‌باشد، ورودی این شبکه یک بردار تصادفی تولیدی می‌باشد که ساختار آن به صورت (۵۶،۱) می‌باشد. این ورودی به یک لایه تمام متصل با ۲۵۶ نرون و تابع فعال ساز leaky_relu وارد می‌شود و پس از آن از چند لایه تمام متصل، مطابق با تصویر خلاصه ساختار شبکه عبور کرده و در نهایت ساختاری به صورت (۱، ۹۴۰۸) را به خود می‌گیرد و این خروجی برای ورودی مرحله متمایزگر استفاده خواهد شد.

بخش متمایزگر شبکه FCGAN :

در این ساختار ورودی هم اندازه با خروجی مرحله مولد بوده است و از لایه‌های تمام متصل استفاده کرده‌ایم که تابع فعال ساز آن‌ها leaky_relu می‌باشد، در انتها نیز از یک لایه تمام متصل با یک نرون برای دسته بندی مسئله به عنوان کلاس واقعی یا مجازی با تابع فعال ساز سیگموید استفاده شده است.

```
1 def create_discriminator():
2     discriminator = Sequential()
3
4     discriminator.add(Dense(1024, input_dim=56*56*3))
5     discriminator.add(LeakyReLU(0.2))
6
7     discriminator.add(Dense(512))
8     discriminator.add(LeakyReLU(0.2))
9
10    discriminator.add(Dense(256))
11    discriminator.add(LeakyReLU(0.2))
12
13    discriminator.add(Dense(1, activation='sigmoid'))
14    discriminator.compile(loss='binary_crossentropy', optimizer = optimizer)
15    return discriminator
```

شکل زیر ساختار و تعداد پارامترهای قابل یادگیری این بخش را نمایش می‌دهد:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 1024)	9634816
leaky_re_lu_3 (LeakyReLU)	(None, 1024)	0
dense_5 (Dense)	(None, 512)	524800
leaky_re_lu_4 (LeakyReLU)	(None, 512)	0
dense_6 (Dense)	(None, 256)	131328
leaky_re_lu_5 (LeakyReLU)	(None, 256)	0
dense_7 (Dense)	(None, 1)	257
Total params: 10,291,201		
Trainable params: 10,291,201		
Non-trainable params: 0		

پس از ساخت این دو مرحله حال نوبت به ساخت مدل GAN مربوط به این دو بخش می‌باشد

تصور زیر قطعه کد و خلاصه ساختار این شبکه را نمایش می‌دهد :

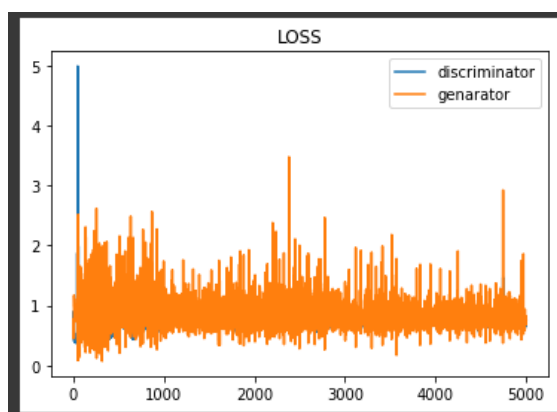
```
1 fc_gan_input = keras.layers.Input(shape=(56,))
2 fc_fake_image = fc_generated(fc_gan_input)
3 fc_gan_output = fc_discriminator(fc_fake_image)
4 fc_gan =keras.models.Model(fc_gan_input, fc_gan_output)
5 fc_gan.compile(loss='binary_crossentropy', optimizer=optimizer)
6 fc_gan.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 56)]	0
sequential (Sequential)	(None, 9408)	10314688
sequential_1 (Sequential)	(None, 1)	10291201
Total params: 20,605,889		
Trainable params: 10,314,688		
Non-trainable params: 10,291,201		

حال شبکه مذکور را با ۵۰۰۰ دور آموزش می‌دهیم و برخی از تصاویر تولید شده توسط این شبکه را نمایش می‌دهیم:

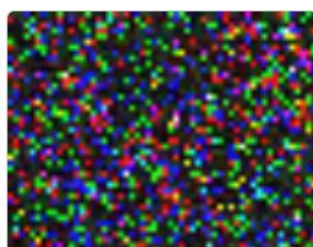
در ابتدا تصاویر تولیدی دارای نویز و کاملاً رندوم هستند اما رفته رفته کیفیت تصویر بهتر شده گرچه شبکه نتوانسته تصاویر سگ و گربه را برای ما تولید کند اما باز هم از تصاویر اولیه به مرور فاصله گرفته ایم تعداد دور آموزش بیشتر، امکان بهبود در این خروجی را به ارمغان خواهد آورد اما محدودیت سخت افزاری مانع از انجام آن است و هم چنین مقدار **loss** برای مولد و متمایزگر نیز برای ۵۰۰ تلاش آخر به شرح زیر است:



```
discriminator loss: 0.6868487000465393
adversarial loss: 0.6692709922790527
discriminator loss: 0.7149016857147217
adversarial loss: 0.911675751209259
discriminator loss: 0.7059229016304016
adversarial loss: 0.8899266123771667
discriminator loss: 0.7236690521240234
adversarial loss: 0.742743968963623
discriminator loss: 0.6892371773719788
adversarial loss: 0.8788919448852539
```



fc_gan_generated0.png



fc_gan_generated100.png



fc_gan_generated200.png



fc_gan_generated300.png



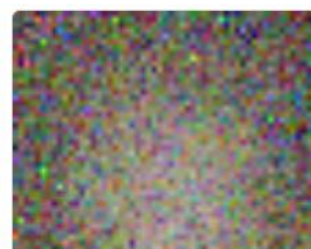
fc_gan_generated400.png



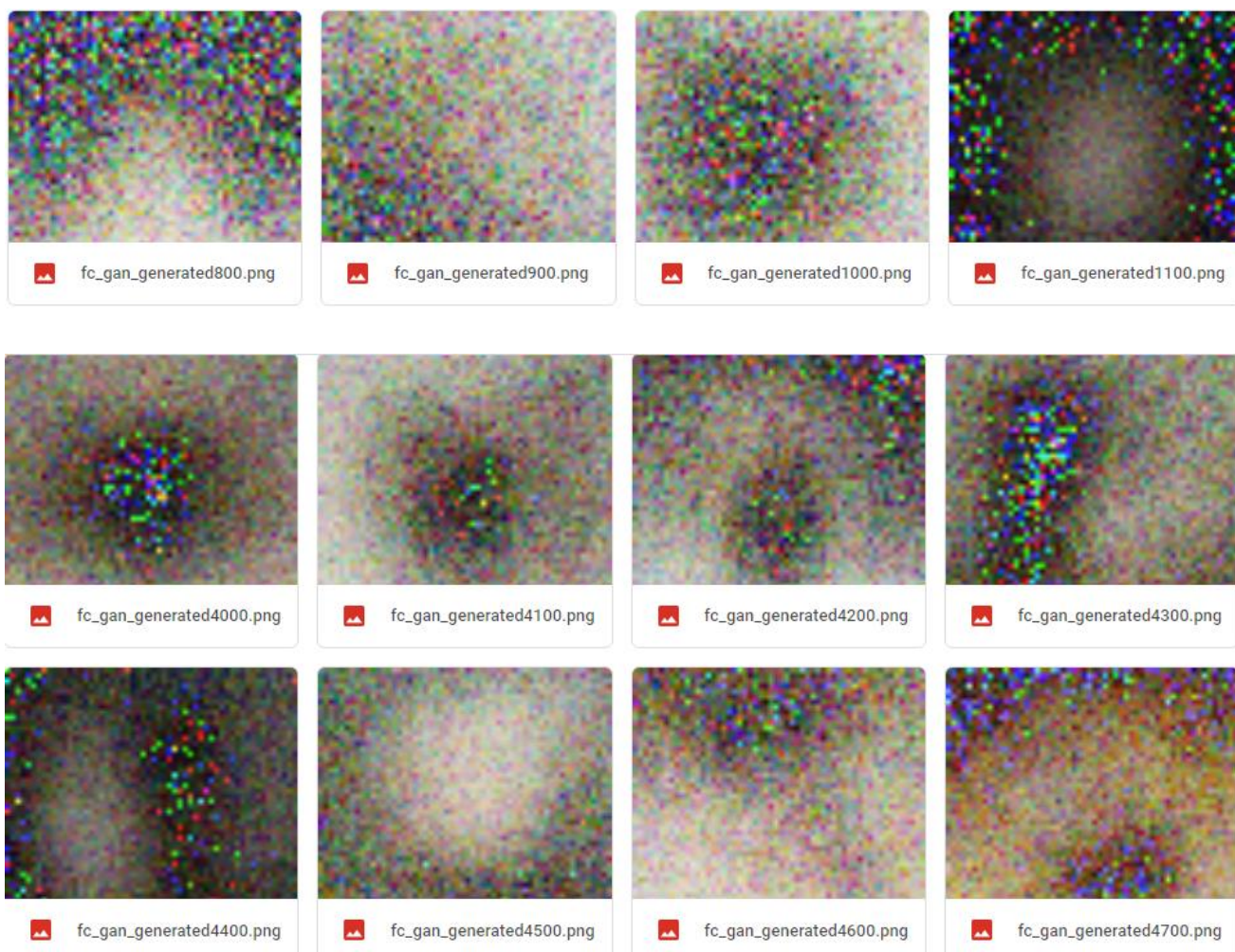
fc_gan_generated500.png



fc_gan_generated600.png



fc_gan_generated700.png



همان طور که مشاهده می شود تصاویر تولید شده توسط شبکه DCGAN نسبت به این شبکه بسیار مطلوب تر می باشد.

حال با توجه به سوال قسمت ۴ تعداد لایه های موجود در ساختار شبکه مولد را افزایش می دهیم:

کاهش لایه ها با توجه به این که در هر لایه ویژگی هایی از تصاویر استخراج می شود و در مرحله بعد این ویژگی ها با هم ترکیب می شوند در چنین حالتی کاهش تعداد لایه ها عملکرد و راندمان مدل را کاهش خواهد داد که علت آن نیز میتوان به این اشاره کرد که مسئله به شدت پیچیده است و نیاز به لایه های بیشتر برای ترکیب و استخراج این ویژگی ها داریم پس به افزایش لایه ها می پردازیم :

عمق بخش مولد را مطابق با بلاک کد زیر افزایش داده ایم :

```
optimizer = Adam(0.0002, 0.5)
def create_generator():
    generator = Sequential()

    generator.add(Dense(128, input_dim=56))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(256))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(512))
    generator.add(LeakyReLU(0.2))

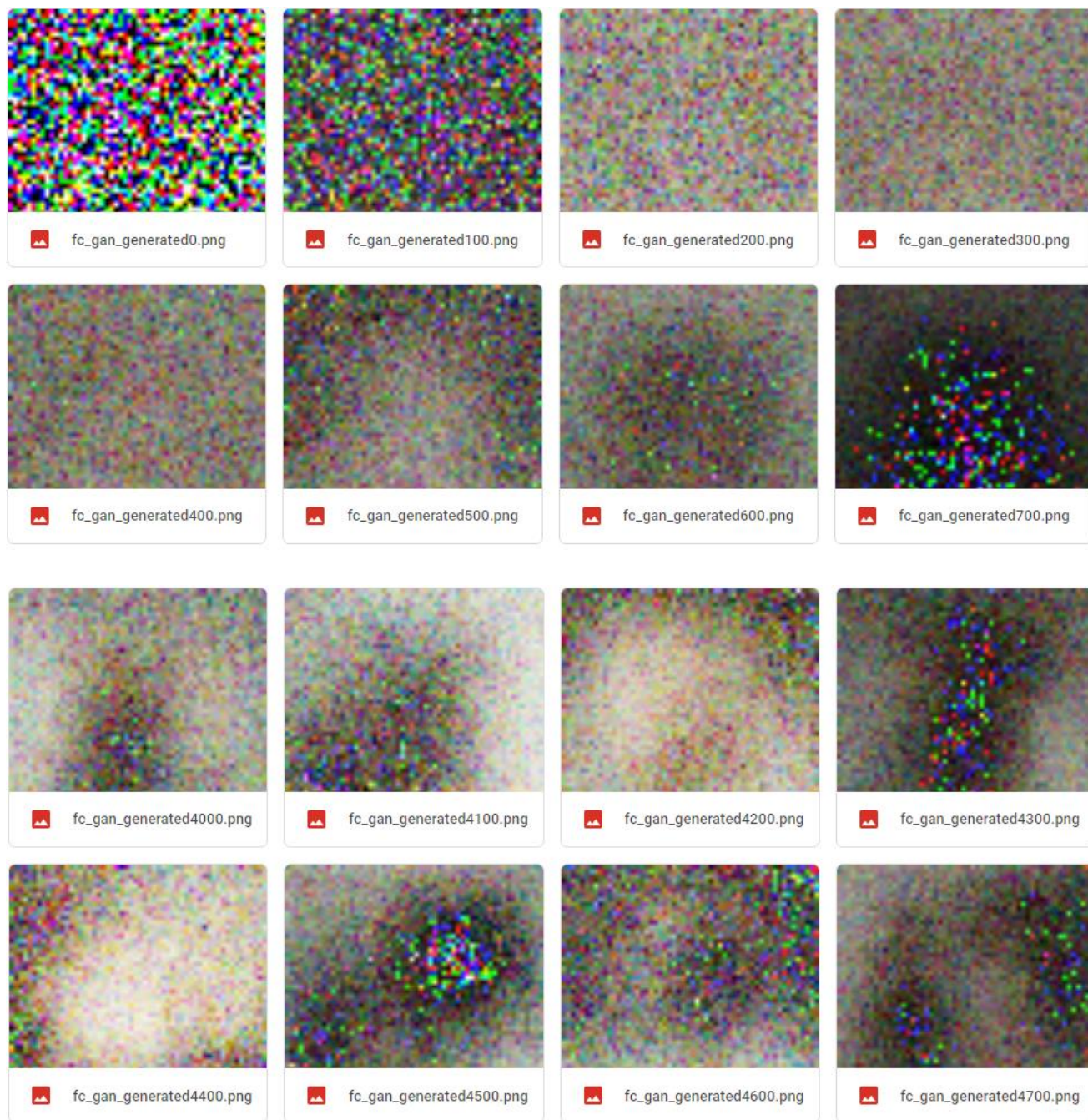
    generator.add(Dense(1024))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(2048, input_dim=56))
    generator.add(LeakyReLU(0.2))

    generator.add(Dense(56*56*3, activation='tanh'))
    generator.compile(loss='binary_crossentropy', optimizer=optimizer)
    return generator
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	7296
leaky_re_lu (LeakyReLU)	(None, 128)	0
dense_1 (Dense)	(None, 256)	33024
leaky_re_lu_1 (LeakyReLU)	(None, 256)	0
dense_2 (Dense)	(None, 512)	131584
leaky_re_lu_2 (LeakyReLU)	(None, 512)	0
dense_3 (Dense)	(None, 1024)	525312
leaky_re_lu_3 (LeakyReLU)	(None, 1024)	0
dense_4 (Dense)	(None, 2048)	2099200
leaky_re_lu_4 (LeakyReLU)	(None, 2048)	0
dense_5 (Dense)	(None, 9408)	19276992
=====		
Total params: 22,073,408		
Trainable params: 22,073,408		
Non-trainable params: 0		

حال با این ساختار و افزایش عمق داده شده شبکه را با ۵۰۰۰ iteration آموزش می‌دهیم :



در این بخش با ۶ لایه تمام متصل آموزش شبکه را انجام داده‌ایم در حالی که در بخش قبل با ۴ لایه تمام متصل در بخش مولد شبکه را آموزش داده بودیم.

مقدار loss برای این آزمایش برای ۵۰۰ تلاش آخر برابر است با :

```
discriminator loss: 0.6442545652389526
generator loss: 0.7324432730674744
discriminator loss: 0.6376244425773621
generator loss: 0.9575151205062866
discriminator loss: 0.7244008183479309
generator loss: 0.7943868637084961
discriminator loss: 0.6616056561470032
generator loss: 0.9422572255134583
discriminator loss: 0.6860424876213074
generator loss: 1.1594419479370117
```

سوال ۵ :

برای برطرف کردن این مشکل چندین راه کار می‌توان ارائه داد در ابتدا می‌توان با اضافه کردن نویز به برچسب داده‌ها و تصاویر، فرایند آموزش متمایزگر را کندتر کرد، به این ترتیب مولد فرصت کافی برای آموزش را بدست می‌آورد. علاوه بر آن می‌توان در زمان تولید بردار تصادفی برای ورودی مولد برای تولید تصاویر مجازی از یک توزیع گوسی استفاده کنیم به این صورت حالت رندوم بودن این ورودی‌ها تا حدودی کنترل خواهد شد. استفاده از تکنیک نرمال سازی `drop out` برای خروجی لایه‌های متمایزگر منجر به کاهش قدرت و کاهش سرعت همگرایی سریع آن خواهد شد، علاوه بر این روش‌های گفته شده تکنیک‌های دیگری مانده استفاده از تابع فعال ساز `leaky_relu` برای لایه‌های مخفی و تابع فعال ساز `tanh` برای لایه خروجی متمایزگر برای برطرف کردن این مشکل پیش‌نهاد شده است.

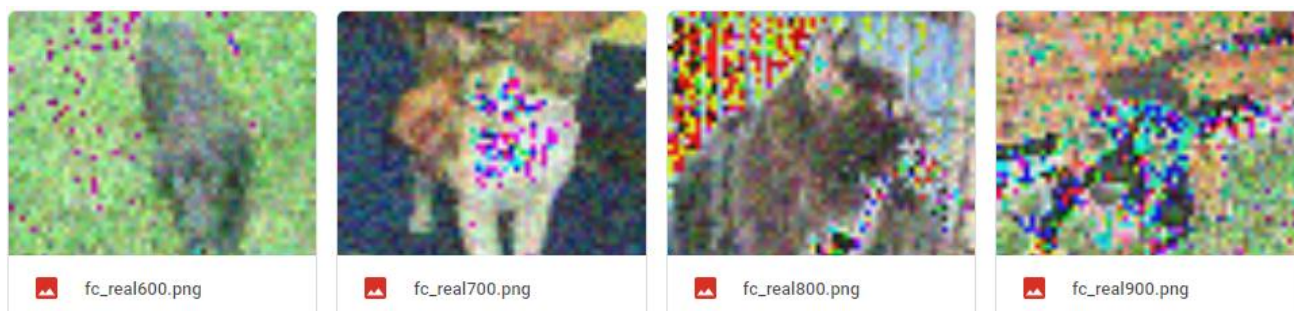
سوال ششم :

برای این بخش با استفاده از بلاک کد زیر یک مقدار تصادفی از یک توزیع نرمال را به تصاویر ورودی اعمال می‌کنیم:

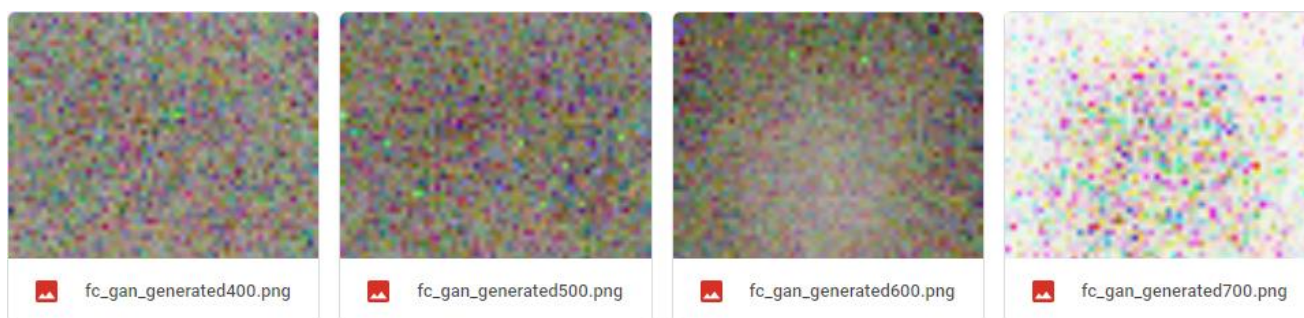
```
real_images += noise_strength * np.random.random(real_images.shape)
```

حال شبکه `fcgan` را اینبار با استفاده از این تکنیک آموزش می‌دهیم.

در شکل زیر نمایشی چند از تصاویر واقعی نویزی شده را نمایش می‌دهیم :



نمایش تصاویر مجازی تولید شده توسط شبکه :



مقدار loss برای مولد و متمایزگر به شرح زیر است :

در این مرحله از آزمایش شدت نویز وارد شده به تصاویر ورودی را ۰.۳ در نظر گرفته بودیم حال در آزمایش بعدی این شدت نویز را کاهش می‌دهیم و نتایج را گزارش می‌کنیم.

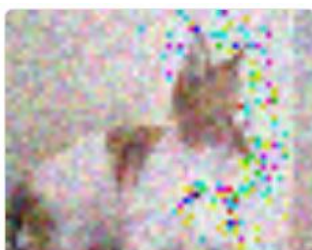
```
discriminator loss: 0.7852710485458374
generator loss: 0.6939018964767456
discriminator loss: 1.7990589141845703
generator loss: 0.09396077692508698
discriminator loss: 0.8400224447250366
generator loss: 2.061497449874878
discriminator loss: 0.6968239545822144
generator loss: 0.9524517059326172
discriminator loss: 0.7948616743087769
generator loss: 1.2786743640899658
discriminator loss: 0.7705121040344238
generator loss: 1.3314908742904663
discriminator loss: 0.6213079690933228
generator loss: 1.2016721963882446
discriminator loss: 0.7010551691055298
generator loss: 1.1202499866485596
discriminator loss: 0.6932101249694824
generator loss: 0.8902467489242554
discriminator loss: 0.697436511516571
generator loss: 0.9297569990158081
```


این بار شدت نویز را برابر با ۰.۰۳ در نظر می‌گیریم که مقداری به مراتب کمتر نسبت به قبل می‌باشد :

تصاویر واقعی نویزی شده:



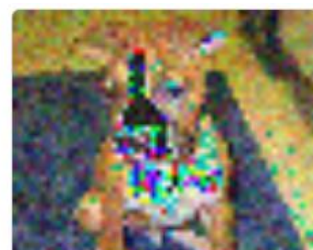
fc_real200.png



fc_real300.png



fc_real400.png



fc_real500.png



fc_real600.png



fc_real700.png

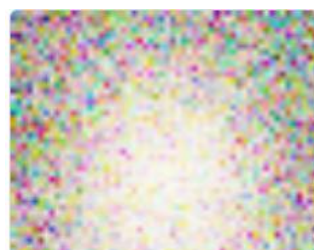


fc_real800.png



fc_real900.png

تصاویر مجازی تولید شده :



fc_gan_generated400.png



fc_gan_generated500.png



fc_gan_generated600.png

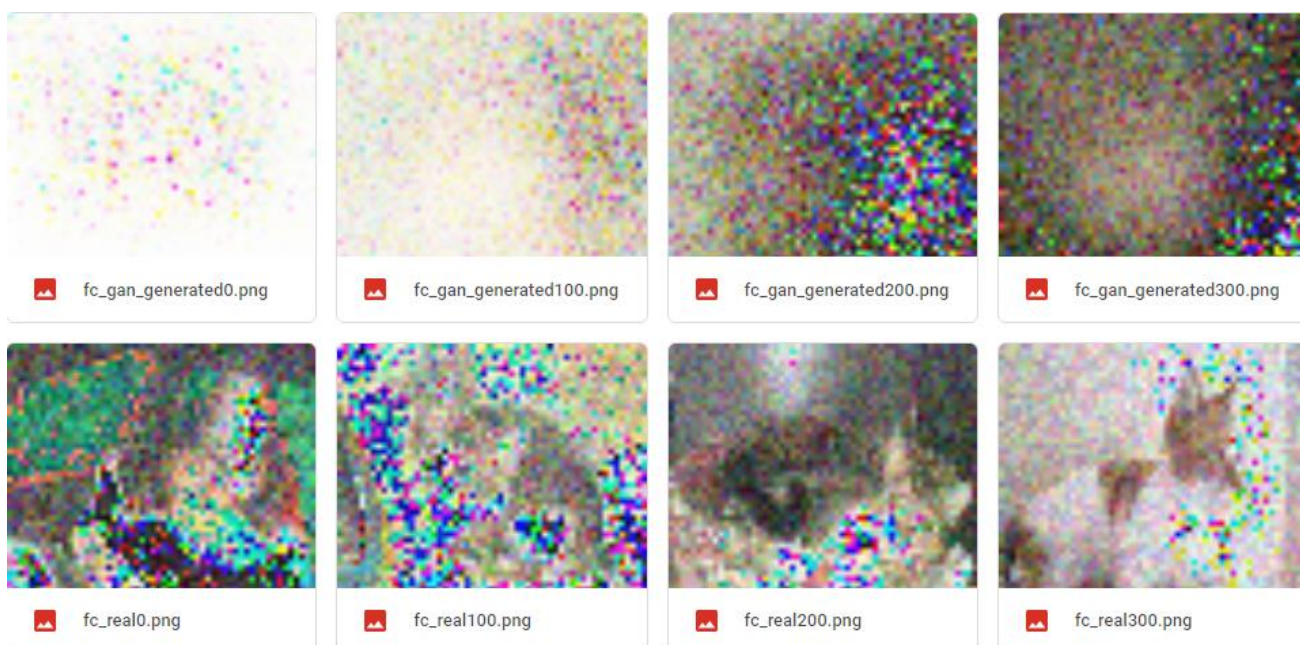


fc_gan_generated700.png

نمایش مقدار loss برای متمایزگر و مولد :

```
discriminator loss: 1.1296844482421875
generator loss: 1.3602049350738525
discriminator loss: 0.7029432058334351
generator loss: 0.6712161302566528
discriminator loss: 0.6416916847229004
generator loss: 1.4172911643981934
discriminator loss: 0.6895214319229126
generator loss: 0.7280462980270386
discriminator loss: 0.6981480717658997
generator loss: 0.8965033292770386
discriminator loss: 0.6892391443252563
generator loss: 0.9174615144729614
discriminator loss: 0.6927448511123657
generator loss: 0.7740777730941772
discriminator loss: 0.6516323089599609
generator loss: 1.5698360204696655
discriminator loss: 0.7402685880661011
generator loss: 1.0757347345352173
discriminator loss: 0.6858730316162109
generator loss: 0.6050978899002075
```

حال این بار دوباره شدت نویز را کاهش می‌دهیم (۰.۰۰۳):



```

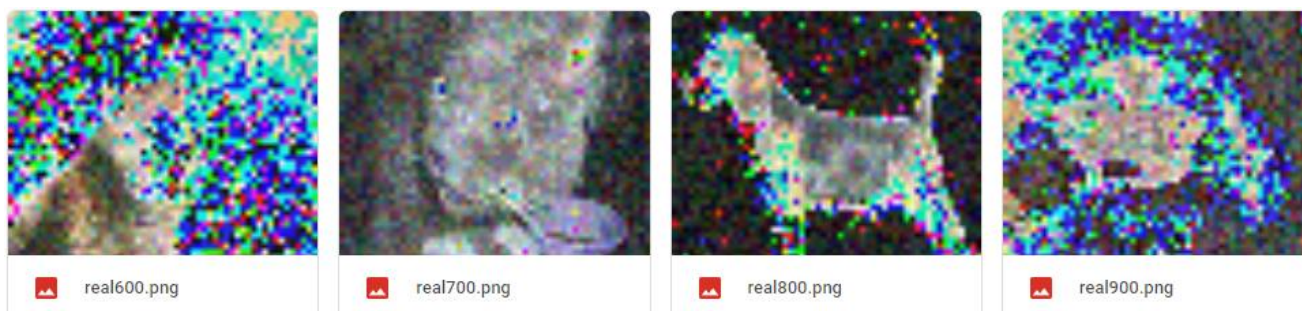
discriminator loss: 0.6816281080245972
generator loss: 0.7230402231216431
discriminator loss: 0.679690957069397
generator loss: 0.6828842759132385
discriminator loss: 0.7056088447570801
generator loss: 0.8030721545219421
discriminator loss: 0.6906868815422058
generator loss: 0.8272153735160828
discriminator loss: 0.8022278547286987
generator loss: 0.9826341867446899
discriminator loss: 0.6922503709793091
generator loss: 0.8530882000923157
discriminator loss: 0.7162622213363647
generator loss: 0.6171309947967529
discriminator loss: 0.6925622820854187
generator loss: 0.8688672780990601
discriminator loss: 0.6985726356506348
generator loss: 0.852975070476532
discriminator loss: 0.6399454474449158
generator loss: 0.7893959283828735

```

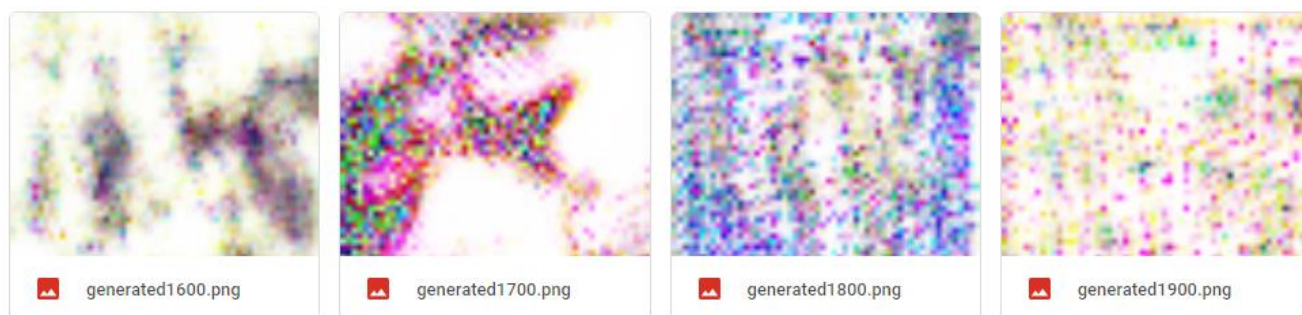
حال مجدد شبکه dc را پیاده سازی نموده و این تکنیک را بر روی آن اجرا میکنیم :

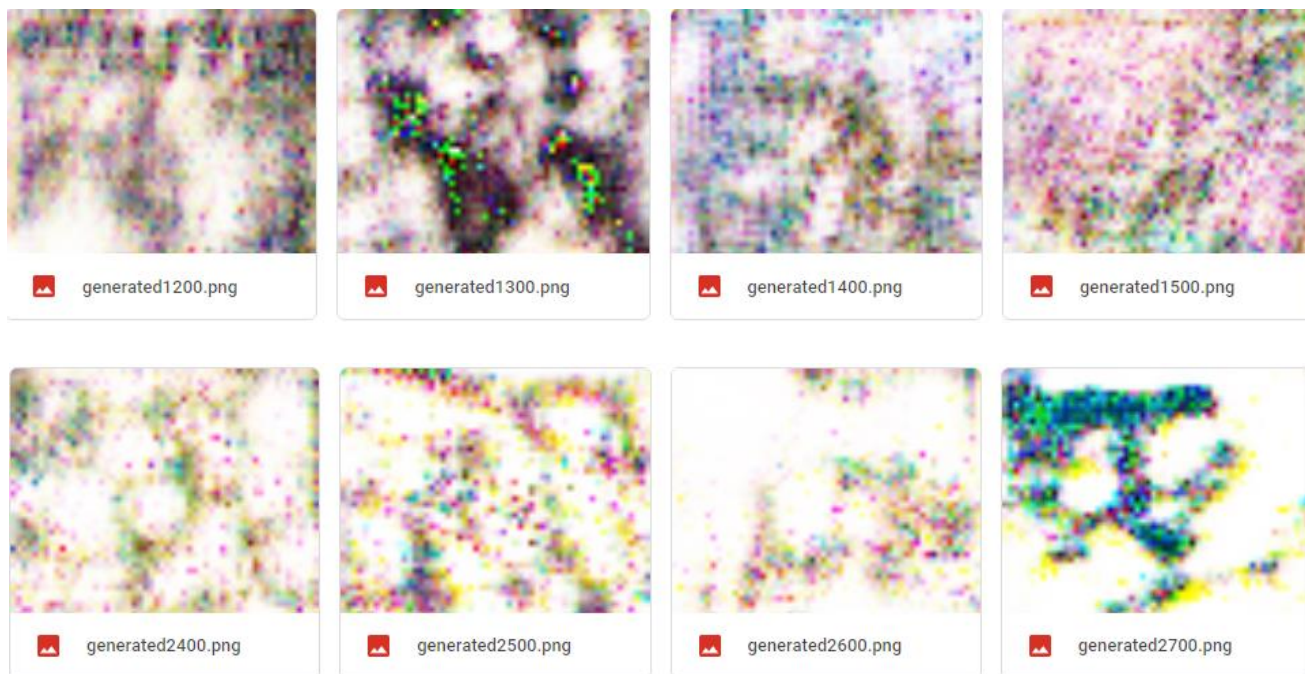
شبکه را با ۵۰۰۰ دور آموزش، و شدت نویز ۰.۳ برای تصاویر ورودی آموزش می‌دهیم :

تصاویر واقعی نویزی شده:



نمونه‌ای از تصاویر مصنوعی تولید شده :



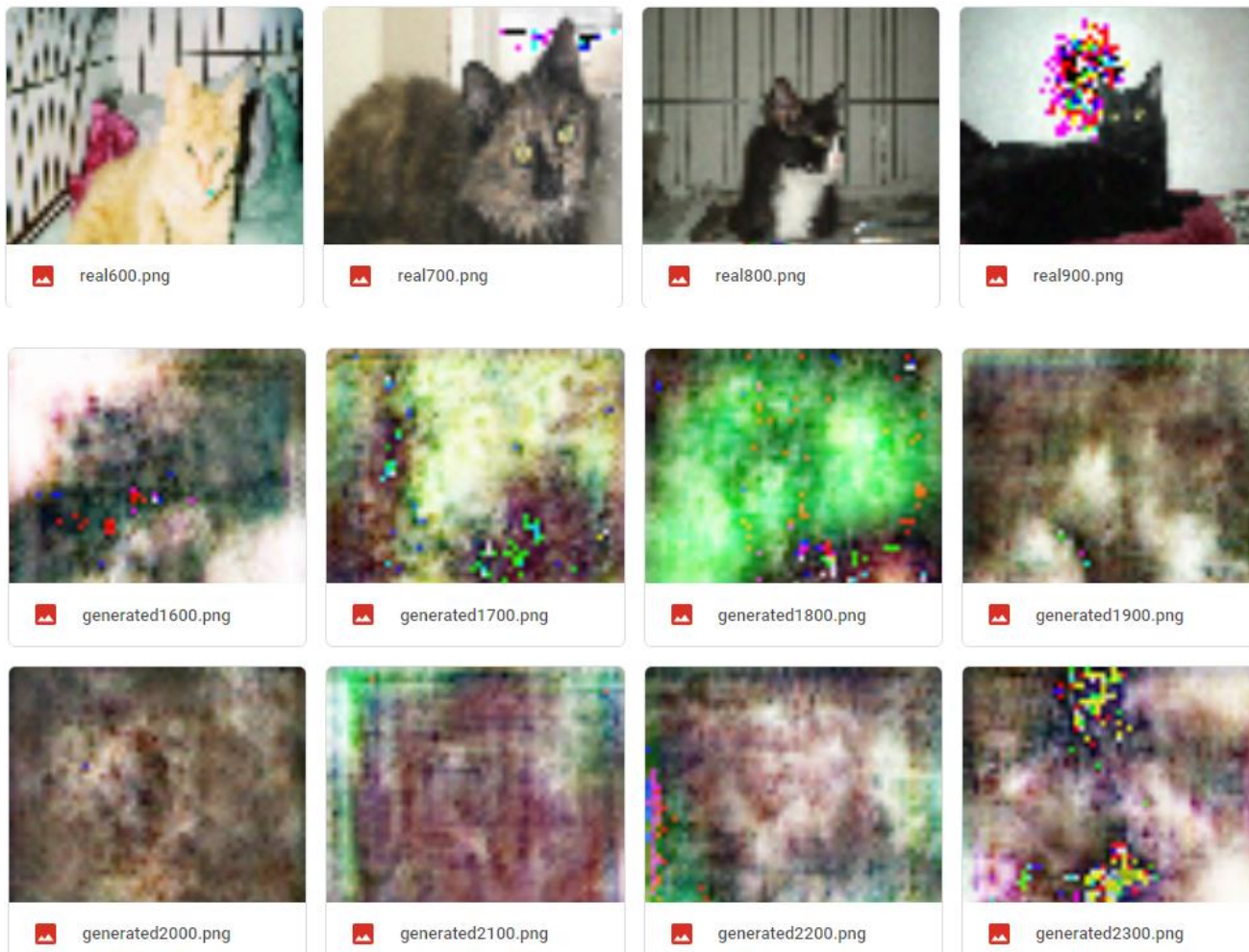


نمایش مقدار loss برای مولد و متمایزگر در ۷۰۰ تلاش آخر :

مشاهده می‌کنیم مقدار loss برای مولد افزایش داشته است در حالی که به دنبال کاهش آن و در پی آن به دنبال افزایش مقدار loss برای متمایزگر هستیم که این مقدار نیز رو به کاهش است.

```
discriminator loss: 0.3894652724266052
adversarial loss: 1.0991790294647217
discriminator loss: 0.9463289976119995
adversarial loss: 0.023417722433805466
discriminator loss: 0.3910316824913025
adversarial loss: 1.1541746854782104
discriminator loss: 0.13035130500793457
adversarial loss: 0.9748051762580872
discriminator loss: 0.39560386538505554
adversarial loss: 1.7616281509399414
discriminator loss: 0.3786468505859375
adversarial loss: 2.2249300479888916
discriminator loss: 0.1361270695924759
adversarial loss: 2.420480251312256
discriminator loss: 0.7451449632644653
adversarial loss: 8.535809516906738
discriminator loss: 0.36344441771507263
adversarial loss: 4.251267910003662
discriminator loss: 0.11038486659526825
adversarial loss: 4.972512722015381
```

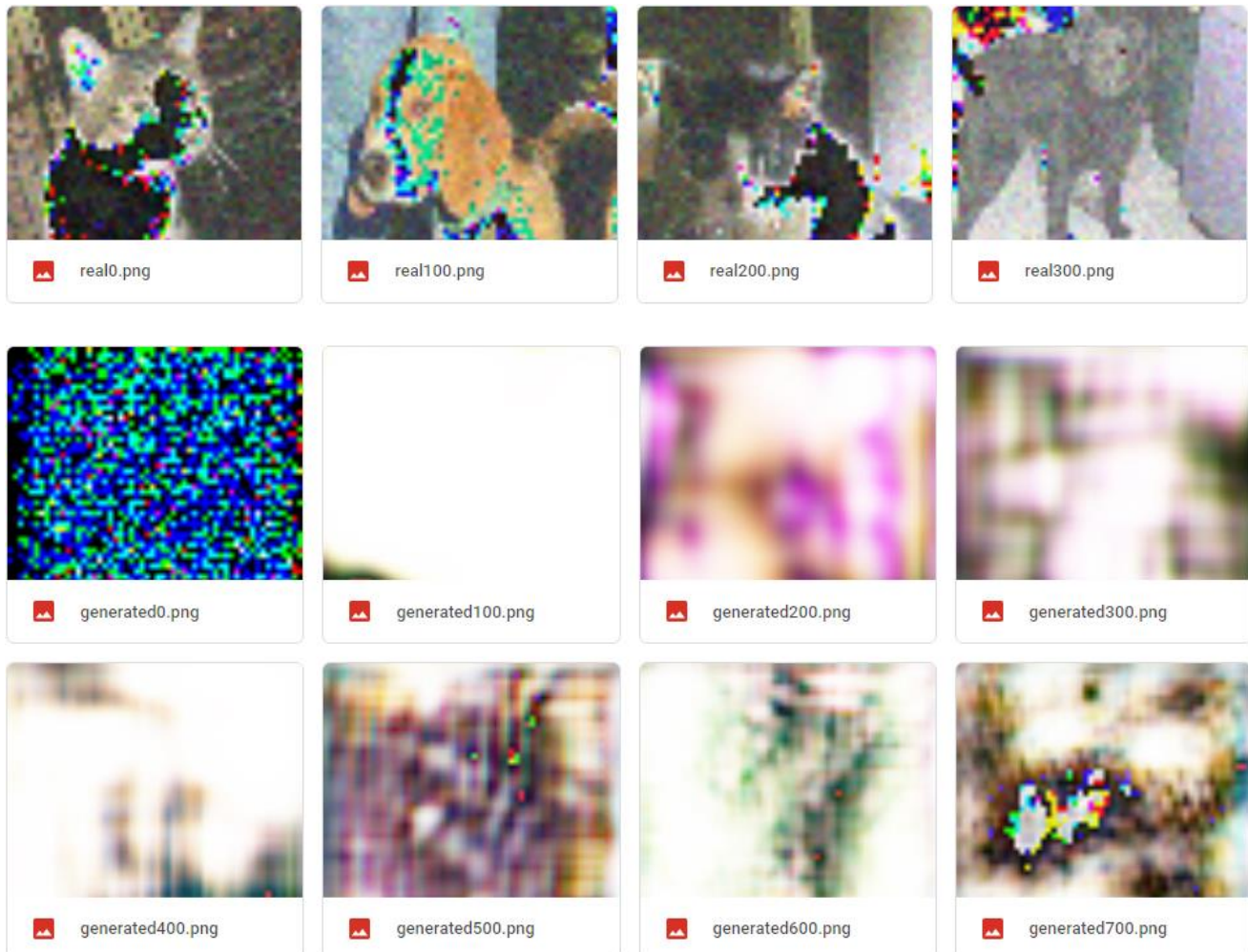
حال این بار شدت نویز را کاهش می‌دهیم و به ۰.۰۳ می‌رسانیم:



با توجه به مقادیر loss برای مولد و متمایزگر نسبت به حالت قبل این شدت نویز تاثیرات مطلوب تری را به ارمغان آورده است و با حدود ۳۰۰۰ دور آموزش به نظر می‌رسد loss برای هر دو به همگرایی رسیده است.

```
discriminator loss: 0.7036124467849731
adversarial loss: 0.7771774530410767
discriminator loss: 0.698095440864563
adversarial loss: 0.7437461018562317
discriminator loss: 0.7064720392227173
adversarial loss: 0.7367696762084961
discriminator loss: 0.6980293393135071
adversarial loss: 0.7947947382926941
discriminator loss: 0.7067005038261414
adversarial loss: 0.8152325749397278
discriminator loss: 0.6951620578765869
adversarial loss: 0.7495230436325073
discriminator loss: 0.6737141609191895
adversarial loss: 0.8552572131156921
discriminator loss: 0.6887260675430298
adversarial loss: 0.7115283012390137
discriminator loss: 0.6924344301223755
adversarial loss: 0.769952654838562
```

شدت نویز را ۰.۱ در نظر می‌گیریم :



افزایش شدت نویز ورودی منجر به گمراه کردن شبکه می‌شود و داده‌های مورد آموزش شبکه را عملاً تغییر داده کاهش این مقدار شدت نویز نیز تا جایی مفید خواهد بود که تصویر خیلی به تصاویر اولیه نزدیک نشده باشد پس باید با ازمون و خطا مقدار مطلوب را بدست آورد در این آزمایش شدت ۰.۰۳ خیلی تاثیر زیادی بر روی داده‌های ورودی وارد نمی‌کند پس می‌توان کران بالا را ۰.۰۳ در نظر گرفت و کران پایین را ۰.۳ ؛ به نظر می‌رسد مقدار ۰.۱ نتایج بهتری را همراه داشته باشد.

```
discriminator loss: 0.7582845687866211
adversarial loss: 0.6194726228713989
discriminator loss: 0.6648997068405151
adversarial loss: 0.7267957925796509
discriminator loss: 0.7207430005073547
adversarial loss: 0.813130259513855
discriminator loss: 0.6842560172080994
adversarial loss: 0.6990841627120972
discriminator loss: 0.7024919390678406
adversarial loss: 0.8341562151908875
discriminator loss: 0.7012232542037964
adversarial loss: 0.8396605253219604
discriminator loss: 0.6688764691352844
adversarial loss: 1.2046091556549072
discriminator loss: 0.6800538301467896
adversarial loss: 0.7817224860191345
discriminator loss: 0.710930585861206
adversarial loss: 0.6390248537063599
```

سوال هفتم :

اساس عملکرد و ساختار شبکه fcgan و dcgan، هر دو با استفاده از یک مولد، به تولید تصاویر با استفاده از یک بردار تصادفی در ورودی می باشد. در ساختار مولد برای شبکه fc با استفاده از لایه های تمام متصل عمل up sampling را انجام می دهیم اما در شبکه DC با استفاده از لایه های کانولوشنی و لایه های کانولوشنی معکوس به تولید و استخراج ویژگی در داده های خود می پردازیم. همان طور که می دانیم لایه های کانولوشنی با توجه به ماهیتی که دارند برای پردازش و استخراج ویژگی برای تصاویر نسبت به لایه های تمام متصل مناسب تر می باشند، و برای تولید تصاویر بهتر است از شبکه DC استفاده کنیم تا شبکه fc. همچنین در تصاویر تولید شده در قسمت های قبل نیز مشاهده کردیم که تصاویر شبکه کانولوشنی به مراتب کیفیت بالاتری نسبت به شبکه تمام متصل داشته اند.

سوال هشتم:

نمایش ۱۰۰ تصویر ساخته شده با استفاده از شبکه DC_GAN :

این تصاویر با استفاده از شبکه مذکور با ۲۰۰۰ دور آموزش و ساختاری به شکل زیر تولید شده‌اند:

Model: "model_2"

Layer (type)	Output Shape	Param #
=====		
input_3 (InputLayer)	[(None, 56)]	0
model (Functional)	(None, 56, 56, 3)	5124163
model_1 (Functional)	(None, 1)	793601
=====		
Total params: 5,917,764		
Trainable params: 5,124,163		
Non-trainable params: 793,601		

