به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

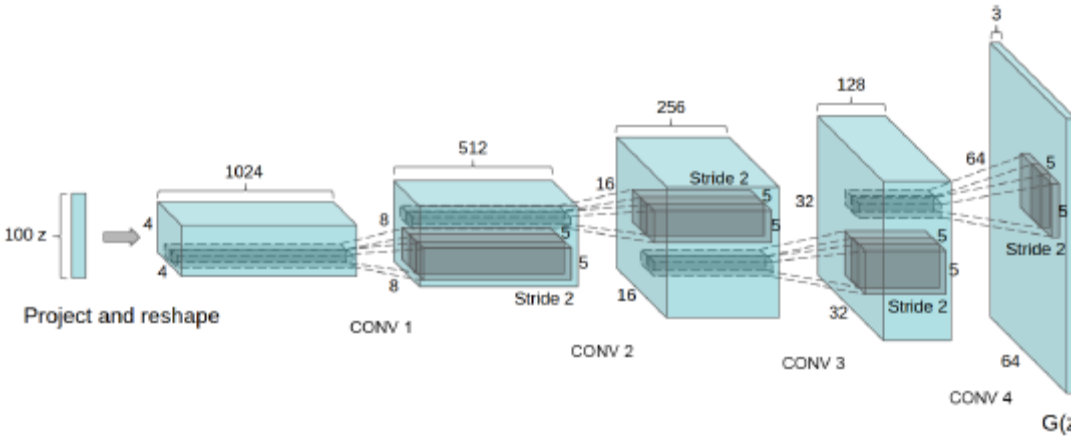
نام و نام خانوادگی	آرمان فروزش – مهسا ندافی قهنوییه
شماره دانشجویی	810100490 - 111946
تاریخ ارسال گزارش	۱۴۰۱.11.7

فهرست

شکل‌ها

پاسخ 1. شبکه های مولد تخاصمی کانولوشنال عمیق

در این بخش با توجه به مقاله ی داده شده دو شبکه ی Discriminator و Generator را با توجه به شکل زیر توسط را مدل سازی کردیم.



شکل 1.1 معماری شبکه ی **Generator**

کدهای مربوطه:

- discriminator

```
def define_discriminator(in_shape=(64,64,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(3, (5,5), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization())
  # downsample
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization())
  # downsample
  model.add(Conv2D(256, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization())
  # downsample
  model.add(Conv2D(512, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(BatchNormalization())
  # downsample
  model.add(Conv2D(1024, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
```

```
  model.add(BatchNormalization())
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['
accuracy'])
  return model

X = define_discriminator()
X.summary()
```

```
Model: "sequential_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_37 (Conv2D)           (None, 64, 64, 3)         228

leaky_re_lu_61 (LeakyReLU)   (None, 64, 64, 3)         0

batch_normalization_16 (Bat  (None, 64, 64, 3)         12
chNormalization)

conv2d_38 (Conv2D)           (None, 32, 32, 128)       9728

leaky_re_lu_62 (LeakyReLU)   (None, 32, 32, 128)       0

batch_normalization_17 (Bat  (None, 32, 32, 128)       512
chNormalization)

conv2d_39 (Conv2D)           (None, 16, 16, 256)       819456

leaky_re_lu_63 (LeakyReLU)   (None, 16, 16, 256)       0

batch_normalization_18 (Bat  (None, 16, 16, 256)       1024
chNormalization)

conv2d_40 (Conv2D)           (None, 8, 8, 512)         3277312

leaky_re_lu_64 (LeakyReLU)   (None, 8, 8, 512)         0

batch_normalization_19 (Bat  (None, 8, 8, 512)         2048
chNormalization)

conv2d_41 (Conv2D)           (None, 4, 4, 1024)        13108224

leaky_re_lu_65 (LeakyReLU)   (None, 4, 4, 1024)        0

batch_normalization_20 (Bat  (None, 4, 4, 1024)        4096
chNormalization)

flatten_7 (Flatten)          (None, 16384)             0

dropout_5 (Dropout)          (None, 16384)             0

dense_18 (Dense)             (None, 1)                 16385

=================================================================
Total params: 17,239,025
Trainable params: 17,235,179
Non-trainable params: 3,846
```

generator •

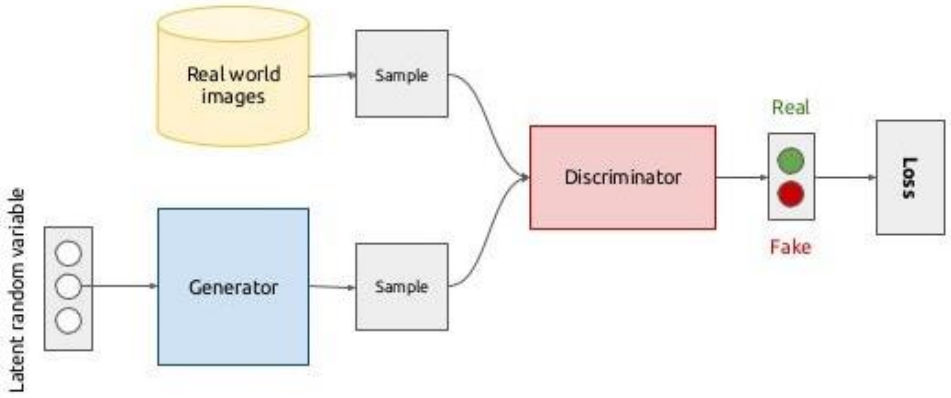
```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 1024 * 4 * 4
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(Reshape((4, 4, 1024)))
  # upsample to 8x8
  model.add(Conv2DTranspose(512, (5,5), strides=(2,2), activation='re
lu', padding='same'))
  model.add(BatchNormalization())
  # upsample to 16x16
  model.add(Conv2DTranspose(256, (5,5), strides=(2,2), activation='re
lu', padding='same'))
  model.add(BatchNormalization())
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (5,5), strides=(2,2), activation='re
lu', padding='same'))
  model.add(BatchNormalization())
  # upsample to 64x64
  model.add(Conv2DTranspose(3, (5,5), strides=(2,2), activation='relu
', padding='same'))
  model.add(BatchNormalization())
  # output layer
  model.add(Conv2D(3, (5,5), activation='tanh', padding='same'))
  model.add(BatchNormalization())
  return model

X = define_generator(100)
X.summary()
```

```
Model: "sequential_23"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_19 (Dense)            (None, 16384)             1654784

 reshape_10 (Reshape)        (None, 4, 4, 1024)        0

 conv2d_transpose_35 (Conv2D  (None, 8, 8, 512)        13107712
 Transpose)

 batch_normalization_21 (Bat  (None, 8, 8, 512)        2048
 chNormalization)

 conv2d_transpose_36 (Conv2D  (None, 16, 16, 256)      3277056
 Transpose)

 batch_normalization_22 (Bat  (None, 16, 16, 256)      1024
 chNormalization)

 conv2d_transpose_37 (Conv2D  (None, 32, 32, 128)      819328
 Transpose)

 batch_normalization_23 (Bat  (None, 32, 32, 128)      512
 chNormalization)

 conv2d_transpose_38 (Conv2D  (None, 64, 64, 3)        9603
 Transpose)

 batch_normalization_24 (Bat  (None, 64, 64, 3)        12
 chNormalization)

 conv2d_42 (Conv2D)          (None, 64, 64, 3)         228

 batch_normalization_25 (Bat  (None, 64, 64, 3)        12
 chNormalization)

=================================================================
Total params: 18,872,319
Trainable params: 18,870,515
Non-trainable params: 1,804
_________________________________________________________________
```

- gan

```
# define the combined generator and discriminator model, for updating
 the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
```

```
  opt = Adam(learning_rate=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```



شکل 1.2 نحوه ی کار شبکه های **GAN**

تمامی نکات گفته شده در مقاله که در قسمت زیر آورده شده است در طراحی رعایت شده است:

Architecture guidelines for stable Deep Convolutional GANs

_ Replace any pooling layers with strided convolutions (discriminator) and fractional-strided

convolutions (generator).

_ Use batchnorm in both the generator and the discriminator.

_ Remove fully connected hidden layers for deeper architectures.

_ Use ReLU activation in generator for all layers except for the output, which uses Tanh.

_ Use LeakyReLU activation in the discriminator for all layers.

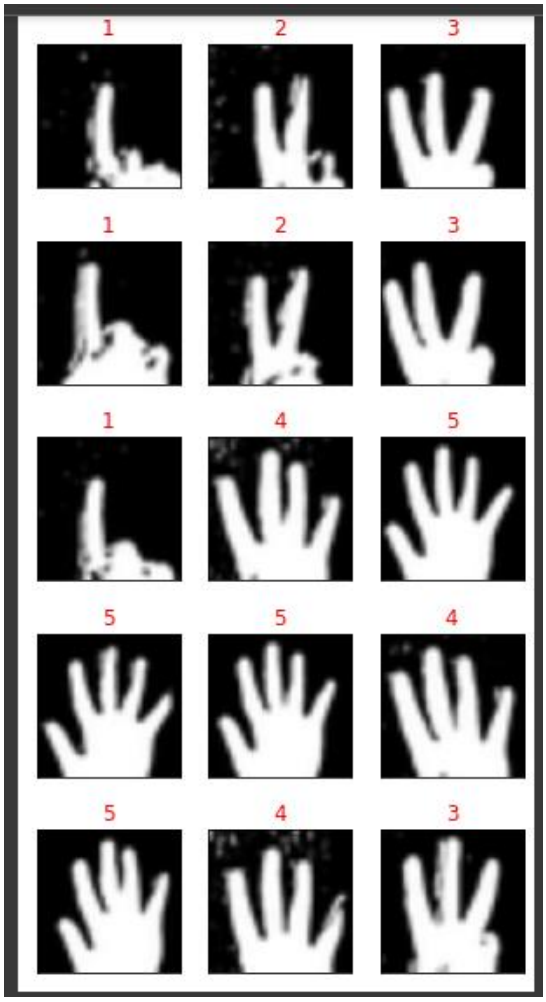
1-1 پیاده سازی مولد تصویر با استفاده از شبکه های مولد تخاصمی کانولوشنال عمیق

در این بخش در ابتدا دیتاست داده شده فراخوانی میشود و در 5 کلاس مجزا در دو متغیر X,Y ذخیره میشود:

```
X_data = []
Y_data = []
```

```
path = "/content/drive/MyDrive/Colab Notebooks/Dataset/Class "
for i in range(5):
  files = glob.glob(path + str(i+1) + "/*.png")
  for myFile in files:
    image = cv2.imread(myFile,0)
    reshapeimg = cv2.resize(image, dsize=(64, 64))
    # gry to rgb for create 3 channel
    finalimg = cv2.cvtColor(reshapeimg, cv2.COLOR_GRAY2BGR)
    X_data.append (finalimg)
    Y_data.append (i+1)

X_data = np.array(X_data)
Y_data = np.array(Y_data)

print('X_data shape:', X_data.shape)
print('Y_data shape:', Y_data.shape)
```

نمونه هایی از این دیتاست در تصویر زیر دیده میشود:



شکل 1-3 نمونه هایی از دیتاست

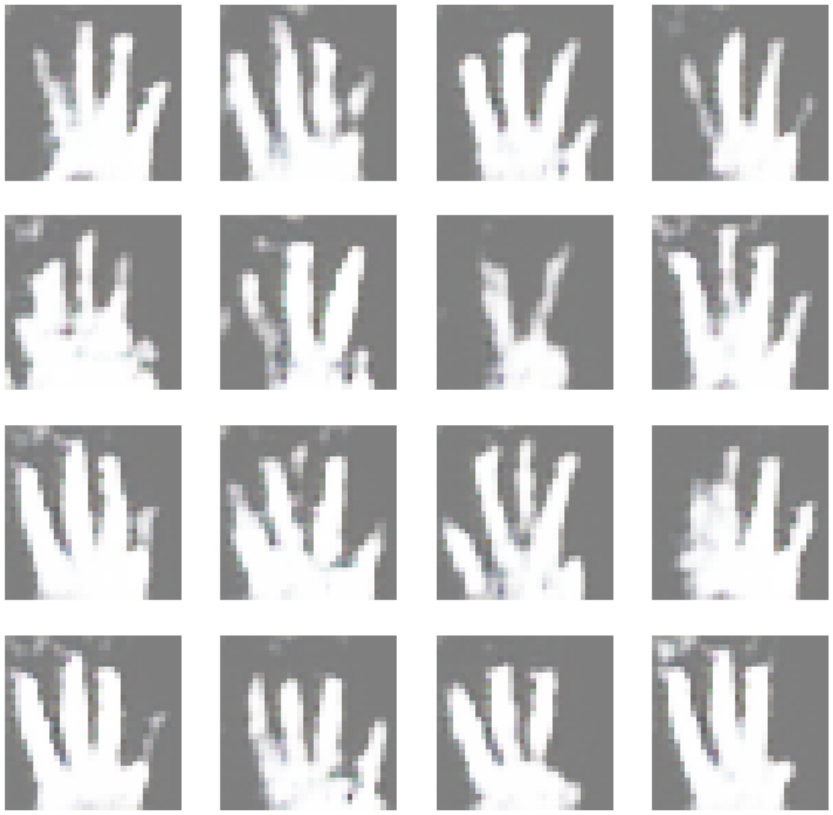
برای نرمال کردن تصاویر آنها را تقسیم بر 255 کرده ایم تا به بازه ی 0و1 انتقال یابد.

با تعداد ایپاک 100 با استفاده از بهینه ساز Adam با نرخ یادگیری 0.001 و تابع loss از نوع binary_crossentropy برای generator و نرخ یادگیری 0.0002 برای gan به آموزش شبکه پرداختیم.

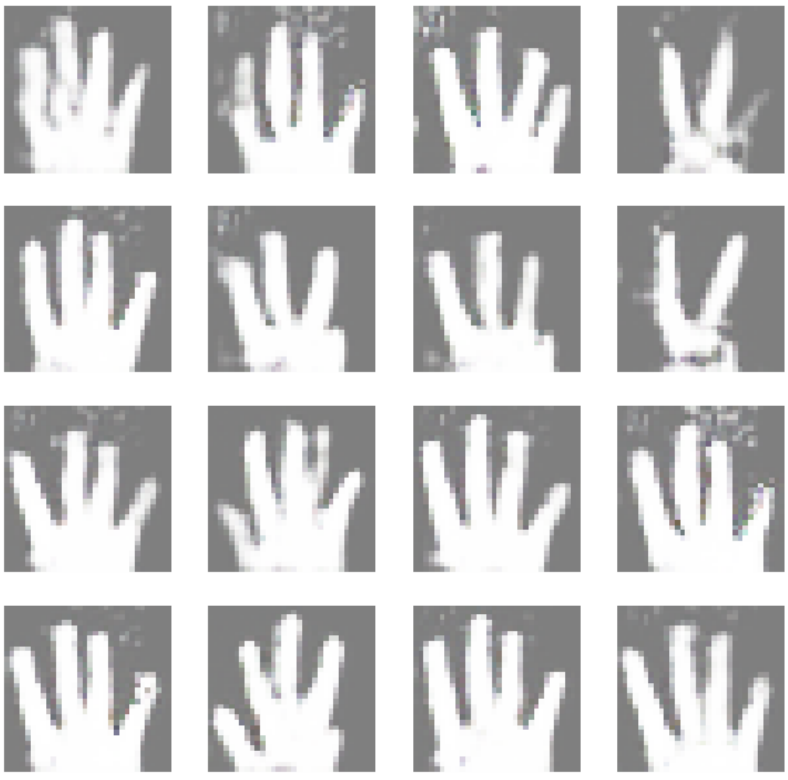
شبکه در ایپاک 191:

```
1/1 [==============================] - 0s 12ms/step
>191, 4/15, d1=0.49225, d2=0.67100, g=2.46105
1/1 [==============================] - 0s 18ms/step
>191, 5/15, d1=0.12174, d2=0.28137, g=4.35990
1/1 [==============================] - 0s 17ms/step
>191, 6/15, d1=0.95709, d2=0.19908, g=2.45913
1/1 [==============================] - 0s 12ms/step
>191, 7/15, d1=0.14320, d2=0.26865, g=3.01067
1/1 [==============================] - 0s 19ms/step
>191, 8/15, d1=0.22362, d2=0.24671, g=3.08059
1/1 [==============================] - 0s 20ms/step
>191, 9/15, d1=0.30650, d2=0.31031, g=2.44660
1/1 [==============================] - 0s 15ms/step
>191, 10/15, d1=0.22527, d2=0.39460, g=2.86209
1/1 [==============================] - 0s 13ms/step
>191, 11/15, d1=0.61106, d2=0.28917, g=3.05799
1/1 [==============================] - 0s 13ms/step
>191, 12/15, d1=0.33189, d2=0.35590, g=2.64632
1/1 [==============================] - 0s 13ms/step
>191, 13/15, d1=0.36011, d2=0.22498, g=2.92824
1/1 [==============================] - 0s 17ms/step
>191, 14/15, d1=0.39849, d2=0.20287, g=2.50966
1/1 [==============================] - 0s 14ms/step
>191, 15/15, d1=0.15504, d2=0.36151, g=3.01201
4/4 [==============================] - 0s 6ms/step
>Accuracy real: 88%, fake: 97%
```

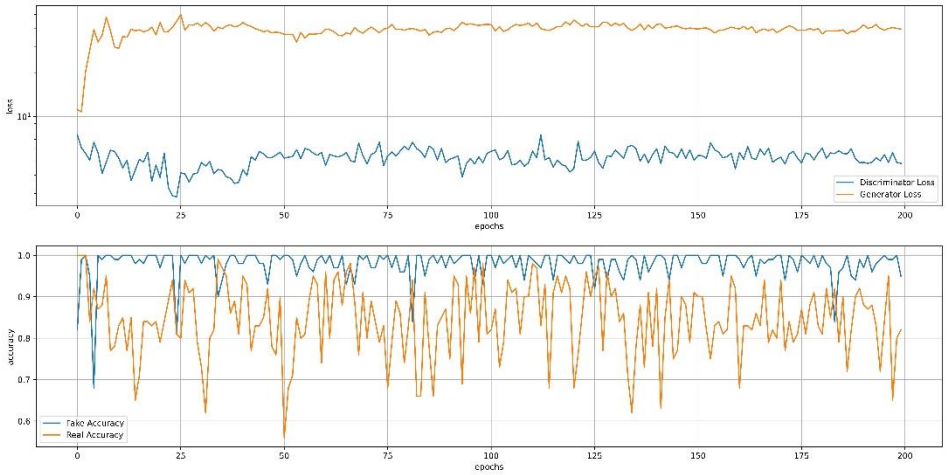
2-1 ارزیابی شبکه



شکل4-1 نتایج شبکه ی **DCGAN** در ایپاک 100



شکل 1.5 نتایج شبکه ی **DCGAN** در ایپاک 200



شکل 1.6 نتایج **loss** و **accuracy** شبکه ی **DCGAN**

توجه شود که با توجه به مدل شبکه که تصاویر را بصورت RGB میگیرد خروجی نیز به رنگ خاکستری و RGB است برای رفع این مشکل میتوان تصاویر خروجی را به حالت grayscale تبدیل کرد.

3-1 پایدارسازی شبکه

به طور کلی آموزش شبکه GAN کاری سخت می باشد، دلیل آن نیز این است که دو شبکه برای آموزش دیدن با یکدیگر در حال رقابت هستند و ممکن است یکی از دو شبکه غالب شود و عملکرد شبکه دیگر نادیده گرفته شود.

یکی از مشکلاتی که ممکن است رخ دهد این است که بلوک G به جای پیدا کردن یک نقطه تعادل، در سیکل تولید یکسری نمونه های مشخص در خروجی بیافتد. یکی دیگر از مشکلات که به نام mode collapse شناخته می شود، این است که چند ورودی مختلف در بلوک G به یک خروجی مشخص نگاشت شود.

به طور کلی کار تحلیلی زیادی بروی چگونگی پایداری سازی این شبکه ها انجام نگرفته است. ولی در طول زمان و با توجه تجربه به دست آمده از تحقیقات و مطالعات به دست آمده در این زمینه به یکسری نکات پی برده شده که باعث بهبود پایداری شبکه می گردد. این نکات در بخش اول سوال آورده شده است.

علاوه بر موارد بالا استفاده در مقاله DCGAN به استفاده از بهینه ساز Adam با پارامترهای نرخ یادگیری 0.0002 و همچین بتا یک 0.5 به جای 0.9 توصیه شده است.

همچنین استفاده از class label هم می تواند خروجی شبکه را تا حد زیادی بهبود ببخشد.(احتمالا از mode collapse تا حدی جلوگیری میکند.)

جهت پایدارسازی شبکه های GAN دوتا از ساده ترین کارها و اولین پیشنهادات ی که داده میشود، به این منظور ابتدا به نحوه پیاده سازی این دو تکنیک و جزئیات هر کدام از روشها ی One-sided label Smoothingو همچنین Add Noise میپردازیم و در نهایت با اصلاح شبکه DCGAN طراحی شده ،موارد خواسته شده در زی ر را بررسی میکنیم.

- **One-sided label smoothing :**

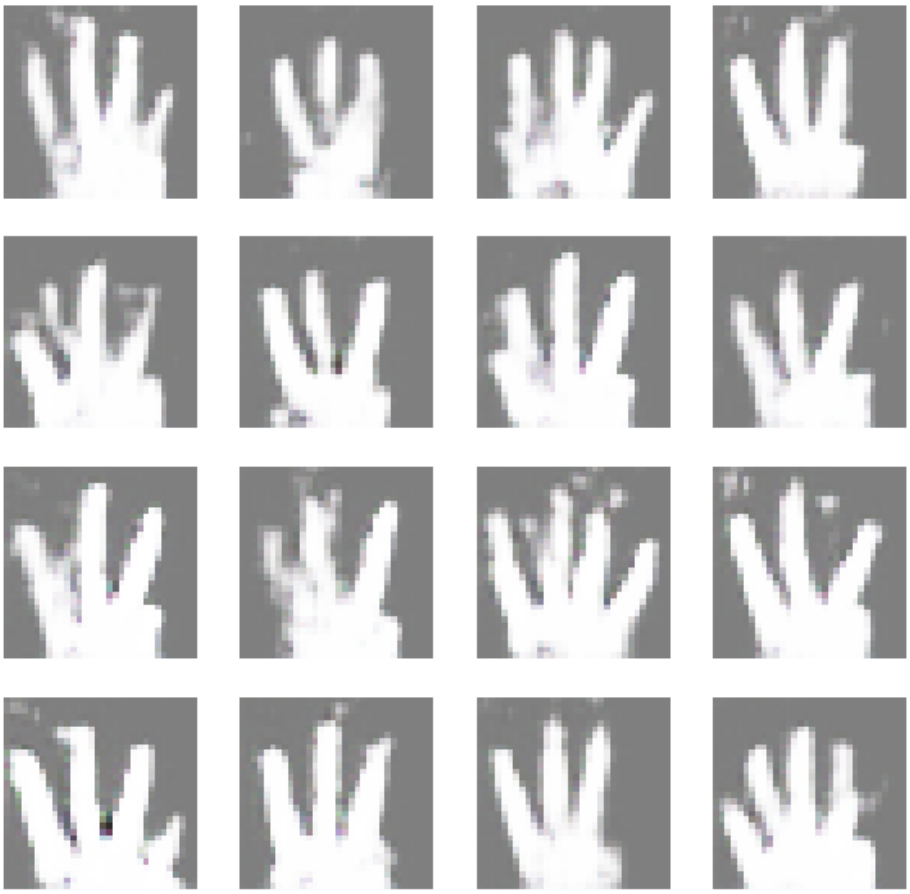
شبکه های عمیق(overconfidence اعتماد بیش از حد) متحمل میشوند به عنوان مثال، از ویژگی های بسیار کمی برای طبقه بندی یک شی استفاده می کند .برای کاهش مشکل، یادگیری عمیق از رگولیشن و dropout بر ای جلوگیر ی از overconfidence استفاده می کند.در GAN ، اگر تمایزکننده به مجموعه کوچکی از ویژگیها برای تشخیص تصاویر واقعی وابسته باشد، مولد ممکن است این ویژگیها را فقط برای بهره بردن از تمایزکننده تولید کند .بهینه سازی ممکن است بیش از حد حریص باشد و هیچ سود طولانی مدتی به همراه نداشته باشد .

برای جلوگیری از این مشکل، زما نی که پیشبینی هر تصویر واقعی از 0.9 فراتر رود، متمایزکننده را جریمه می کنیم(D (تصوی ر واقعی .) 0.9 <) این کار با تنظیم مقدار برچسب هدف ما به جا ی 1.0 روی 0.9 انجام می شود.

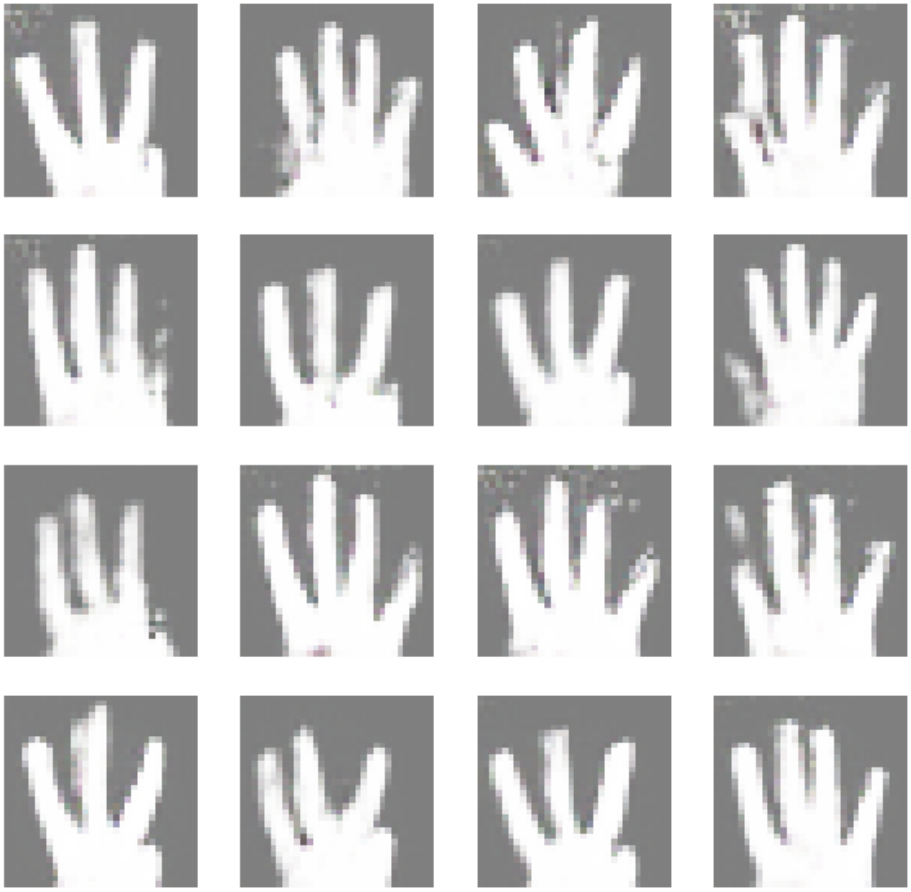
روش دیگر برای دستیابی به همین هدف هموارسازی برچسب است که درک و پیاده سازی آن حتی ساده تر است :اگر مجموعه برچسب برای تصاویر واقعی 1 باشد، آن را به مقدار کمتر ی مانند 0.9 تغییر می دهیم.

تغییرات کد بصورت زیر ایجاد میشود:

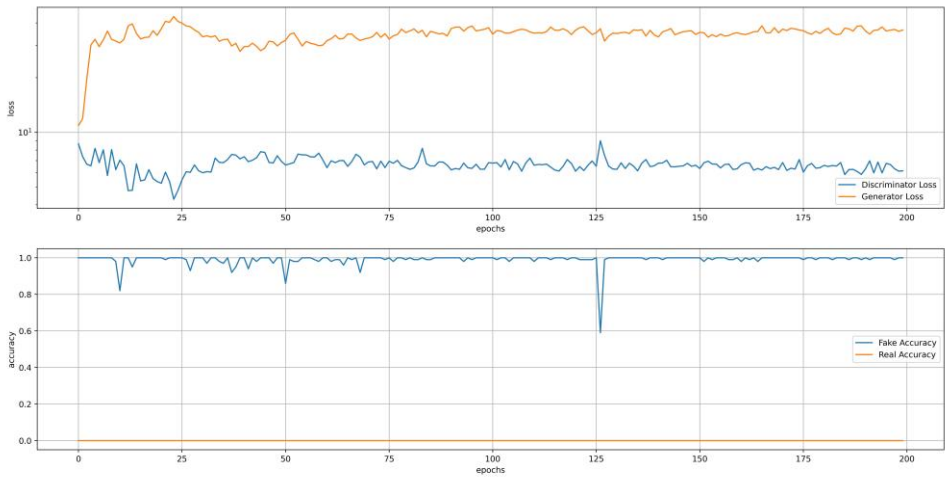
```
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = np.randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = np.ones((n_samples, 1))*0.9   # One sided label smoothing
  return X, y
```



شکل 1.7 نتایج شبکه ی **DCGAN** با **One sided label smoothing** در ایپاک 100



شکل 1.8 نتایج شبکه ی **DCGAN** با **One sided label smoothing** در ایپاک 200



شکل 1.9 نتایج **loss** و **accuracy** شبکه ی **DCGAN** با **One sided label smoothing**

همانطور مشاهده میشود نمودار دقت در داده های فیک کمتر نوسان دارد و در ایپاک دویستم تصویر دقیق تری با رزولوشن بهتر در دست داریم.

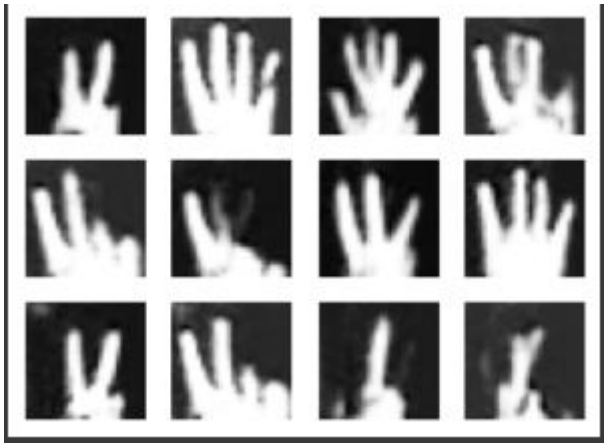
- **افزودن Noise**

دشوارتر کردن آموزش متمایزکننده برای پایداری به طور کلی مفید است. یکی از شناخته شده ترین روشها برای افزایش پیچیدگی آموزش تفکیک کننده، افزودن نویز به داد ه های واقعی و مصنوعی به عنوان مثال تصاویر تولید شده توسط مولد است. در دنیای ریاضی این باید کار کند زیرا به پایداری توزیع دادههای دو شبکه رقیب کمک میکند.

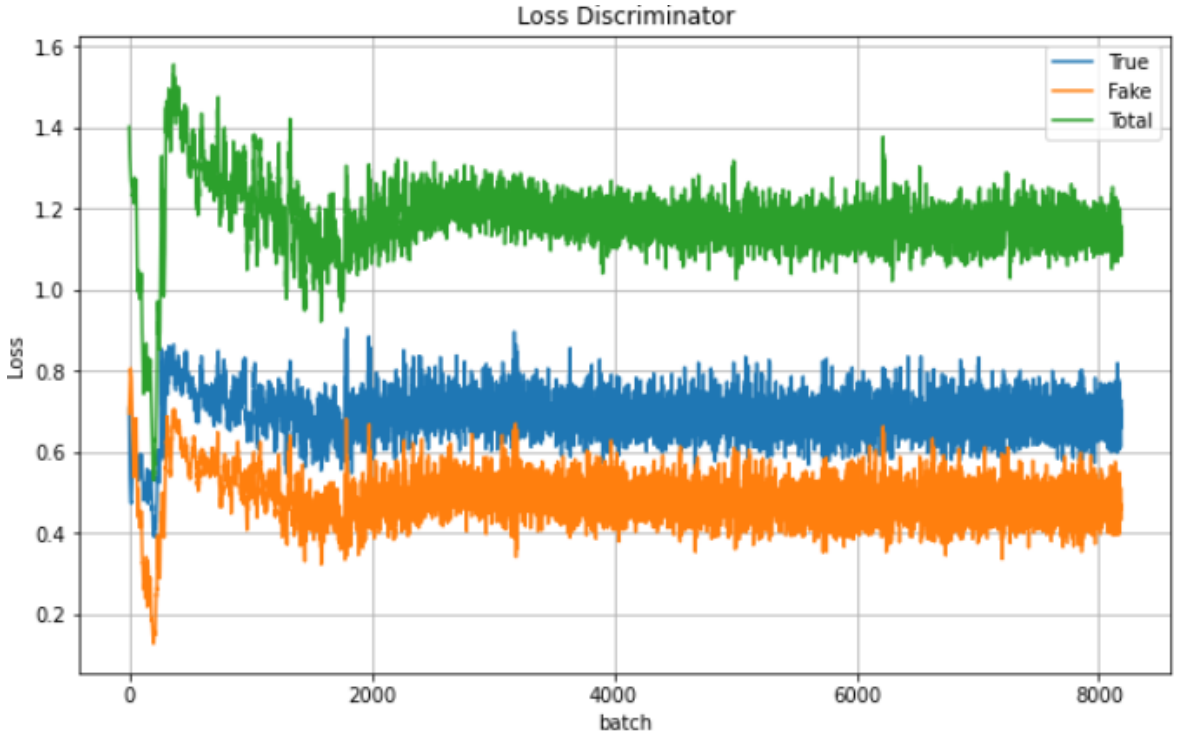
روند افزودن نویز:

```
generator = generator_model()
noise = tf.random.normal([1, 100])
generated_img = generator(noise, training=False)

plt.imshow(generated_img[0,:,:,0], cmap='gray')

discriminator = discriminator_model()
decision_output = discriminator(generated_img)
```

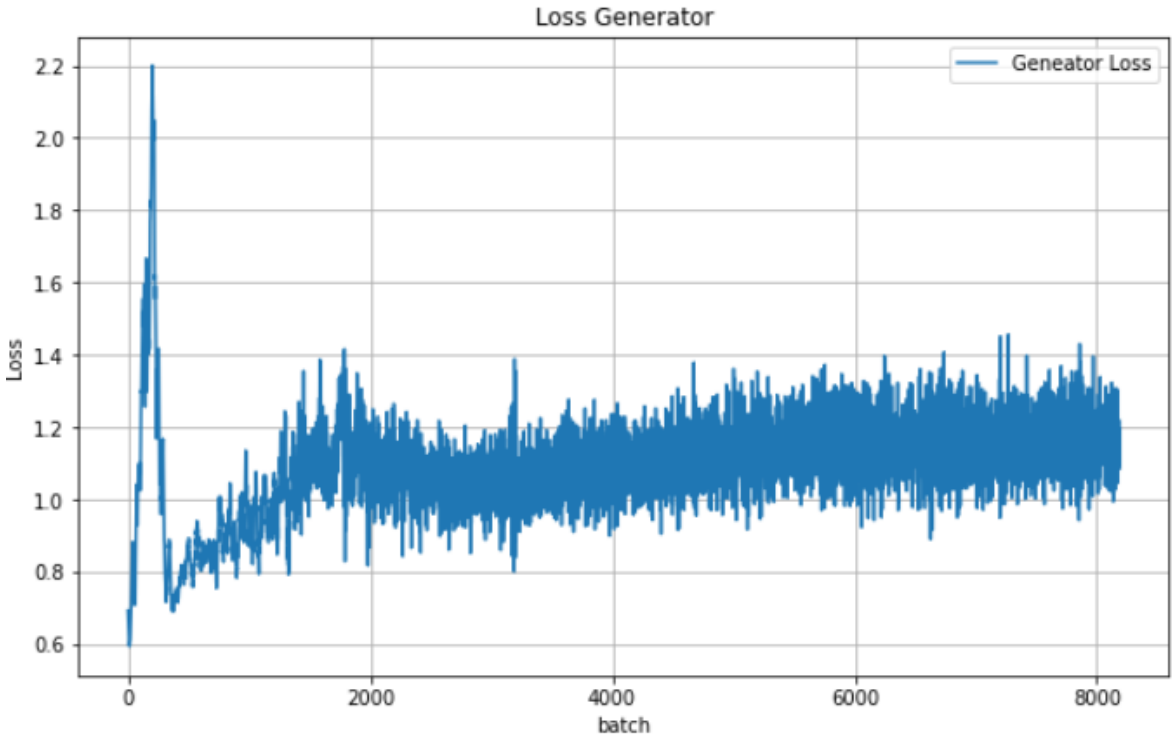
نتایج:



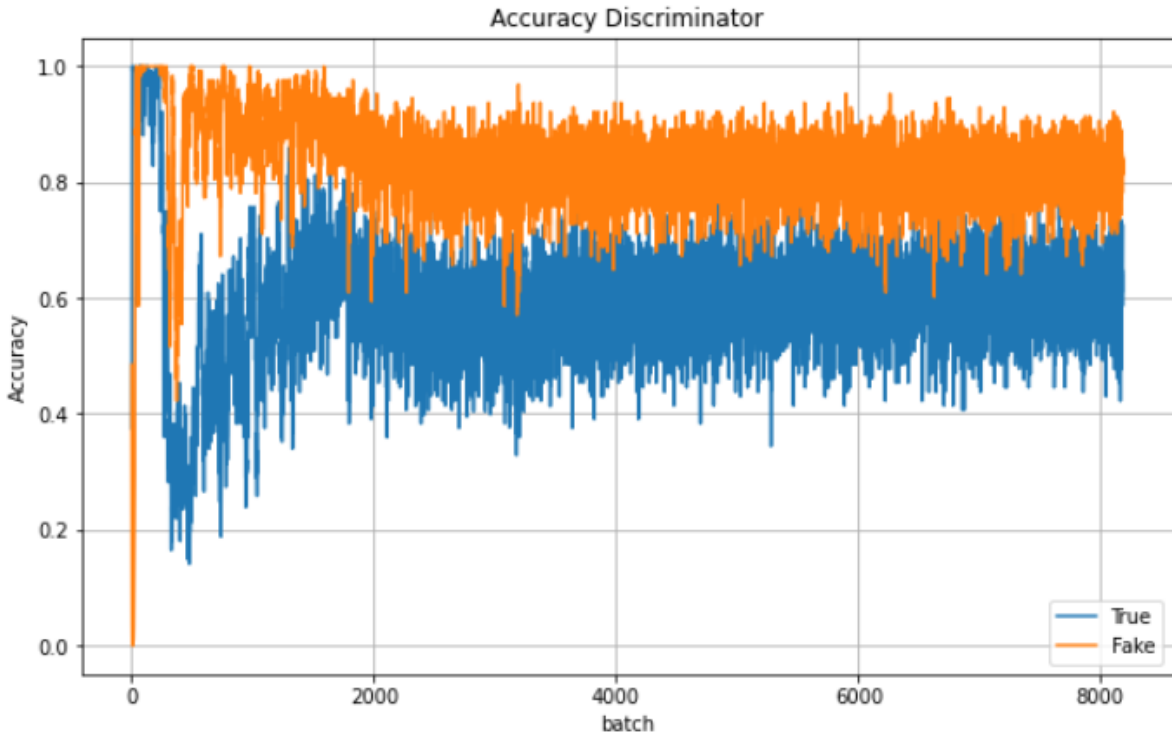
شکل 1.10تصاویر حاصل بعد از افزودن نویز به DCGAN



شکل 1.11 نمودار **Loss** متخاصم



شکل **1.12** نمودار **loss** مولد



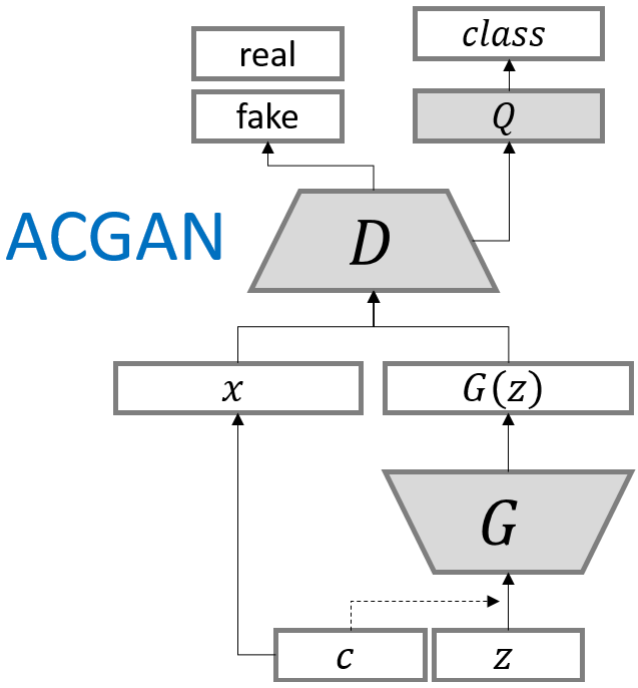
شکل 1.13 نمودار دقت

شاهد بهبود کیفیت شبکه ها و میزان جدال بیشتر بین انها در نمودار ها هستیم اما باید دقت شود که دقت داده های فیک بیشتر بوده است.

پاسخ 2. شبکه متخاصم مولد طبقه‌بند کمکی و شبکه Wasserstein

2-۱. شبکه متخاصم مولد طبقه‌بند کمکی

تفاوت اصلی شبکه‌ی AC-GAN که مخفف auxiliary classifier GAN است، با مدل های دیگر GAN این است که در این شبکه بخش discriminator علاوه بر اینکه باید عکس های real و fake را تشخیص دهد، باید بتواند کلاس عکس ورودی را نیز تعیین کند. علاوه بر آن در بخش generator نیز ورودی این بخش، علاوه بر نویز، کلاس مورد نظر برای تولید عکس نیز میباشد. شماتیک کلی این شبکه در شکل1.2 قابل مشاهده است.



شکل 1.2 ساختار مدل **ACGAN**

همانگونه که مشاهده میشود در این مدل، بخش discriminator دو خروجی (real-fake و کلاس) دارد و همینطور بخش generator نیز دو ورودی (نویز و کلاس) دارد.

در بخش generator با توجه به اینکه لیبل موردنظر به عنوان ورودی به این بخش وارد میشود، باعث بهبود عملکرد سیستم در تولید عکس شده و پروسه تولید عکس را پایدار میکند. در این شبکه تابع هدف از دو بخش زیر تشکیل میشود.

$$L_c = E[log(P(C = c|X_{real})] + E[log(P(C = c|X_{fake})]$$

$$L_s = E[log(P(S = real|X_{real})] + E[log(P(S = fake|X_{fake})]$$

در نهایت در فرایند آموزش، بخش discriminator به دنبال ماکزیمم کردن $L_c + L_s$ و بخش generator به دنبال ماکزیمم کردن $L_c - L_s$ میباشد.

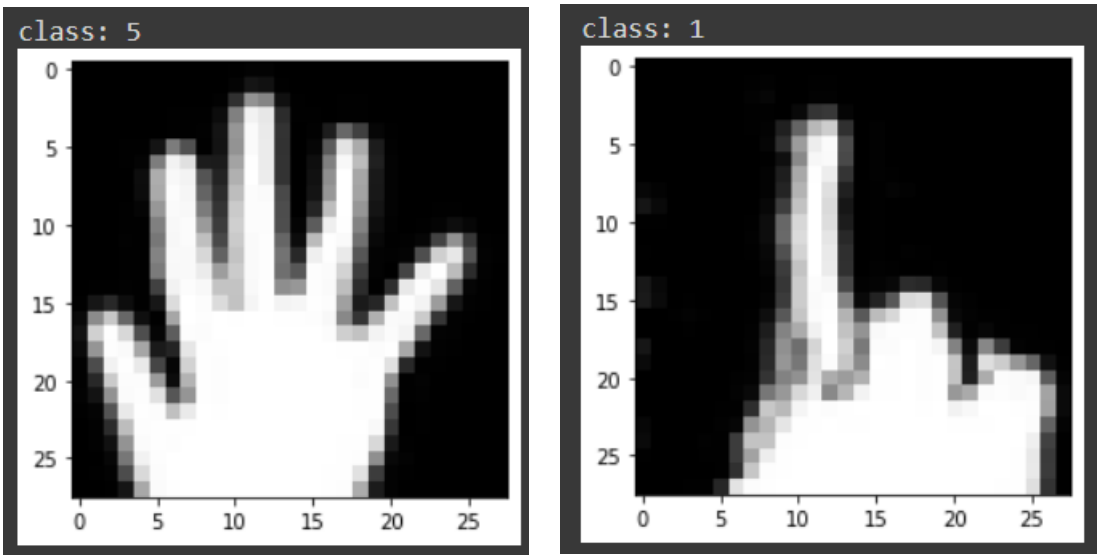
پیاده‌سازی:

برای پیاده‌سازی مدل AC-GAN برروی دیتاست داده شده، علاوه بر داده های تصویری، نیاز به کلاس داده ها نیز داریم. به همین دلیل با استفاده از کد شکل 2.2 دو آرایه X و Y میسازیم که به ترتیب شامل تصاویر و لیبل های مربوطه هستند.

```
X_data = []
Y_data = []

path = "/content/drive/MyDrive/NNDL/HW6/Dataset/Class "
for i in range(5):
  files = glob.glob(path + str(i+1) + "/*.png")
  for myFile in files:
    image = cv2.imread(myFile,0)
    res = cv2.resize(image, dsize=(28, 28))
    # image = cv2.IMREAD_GRAYSCALE (myFile)
    X_data.append (res)
    Y_data.append (i+1)

X_data = np.array(X_data)
Y_data = np.array(Y_data)
print('X_data shape:', X_data.shape)
print('Y_data shape:', Y_data.shape)
```

```
X_data shape: (1005, 28, 28)
Y_data shape: (1005,)
```

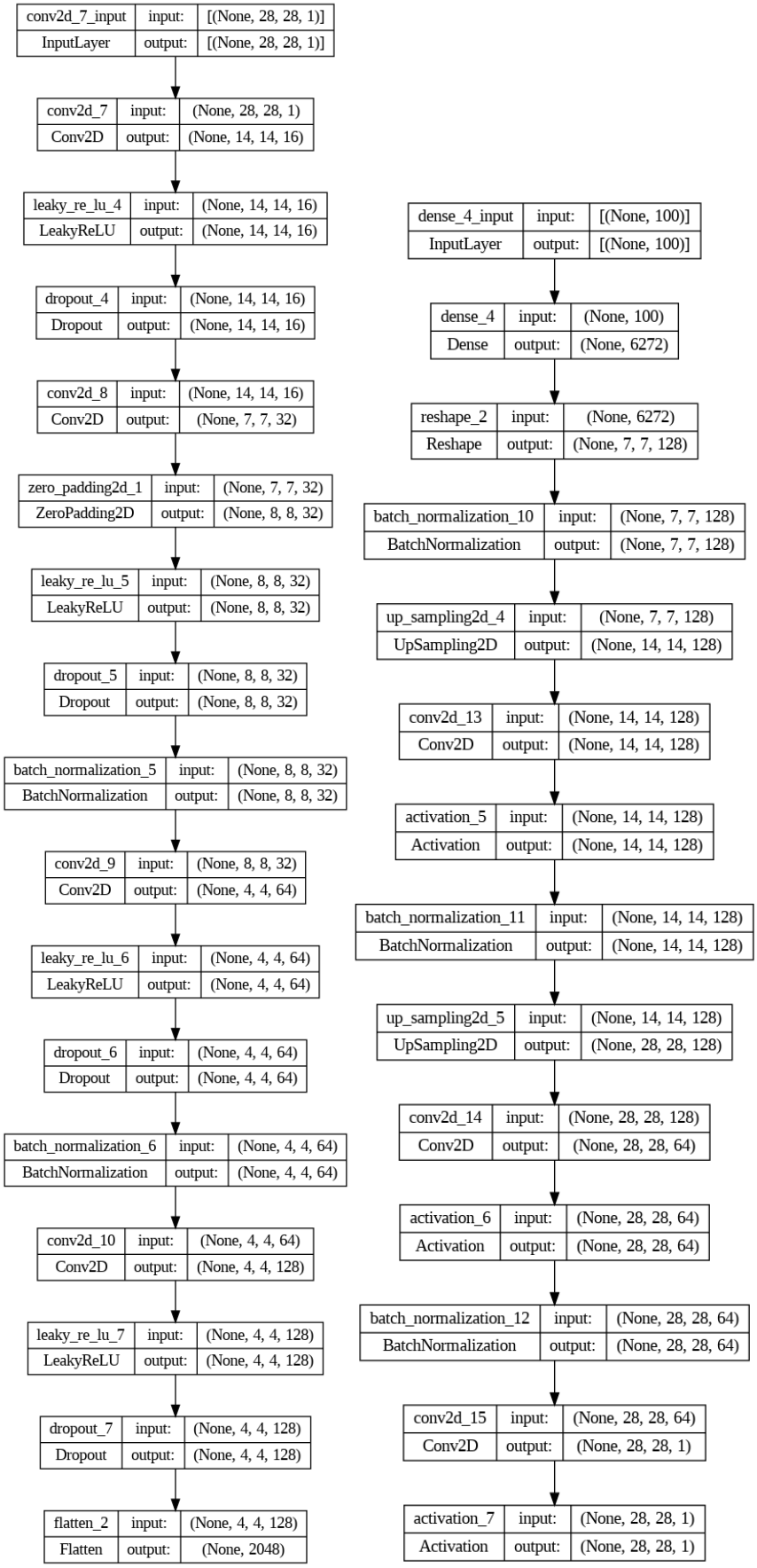
شکل 2.2 کد مربوط به پردازش دیتا



شکل3.2 نمونه تصاویر دیتاست و کلاس مربوطه

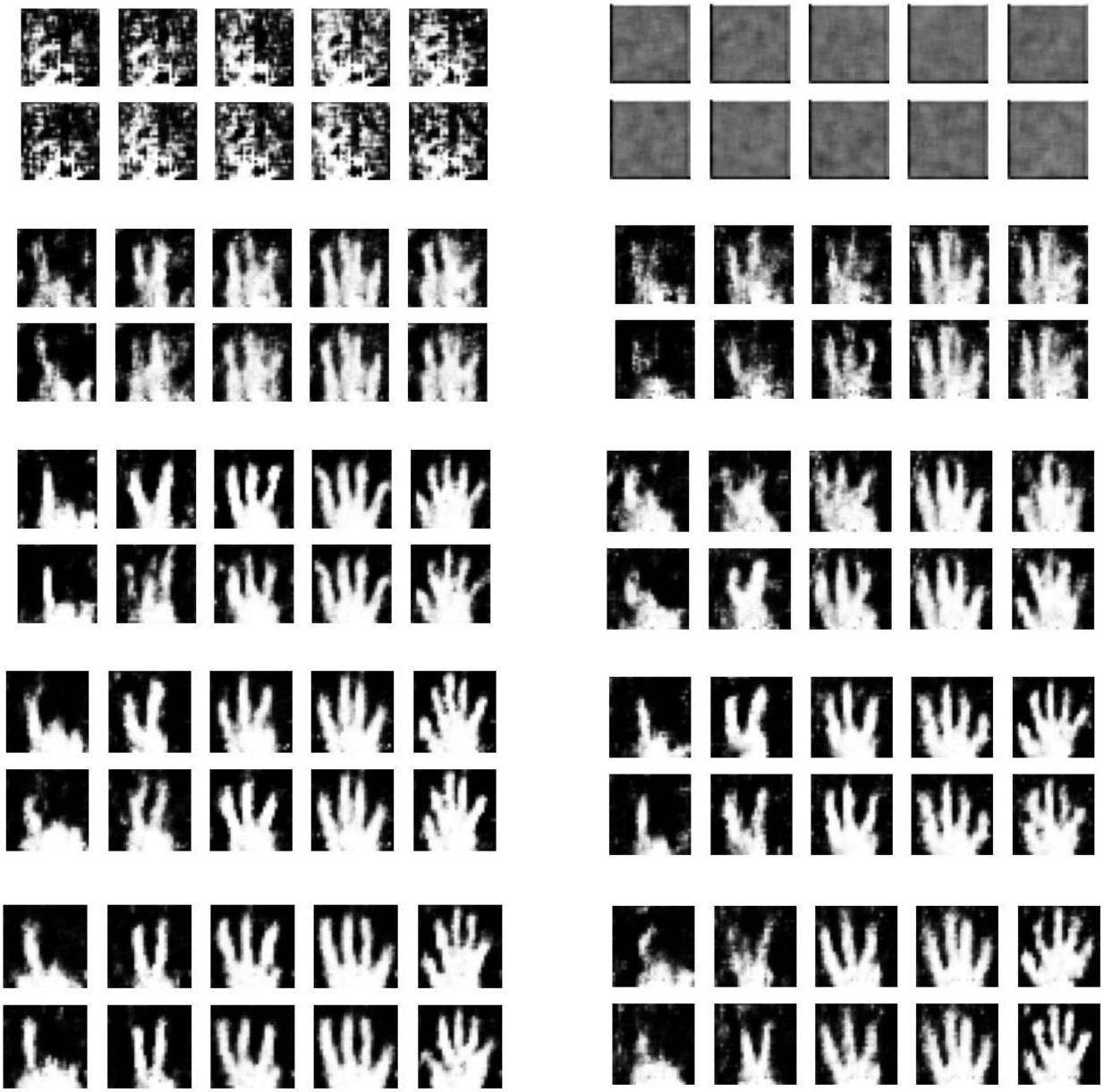
در شکل 4.2 ساختار بخش های generator (شکل سمت راست) و discriminator (شکل سمت چپ) نمایش داده شده است. در این شبیه سازی سایز نویز ورودی 100 در نظر گرفته شده است و همانگونه که مشاهده میشود سایز خروجی شبکه generator با ورودی discriminator مشابه است. در هردو مدل از بهینه ساز Adam با نرخ یادگیری 0.0002 و از تابع هزینه sparse categorical crossentropy استفاده شده است. سپس این شبکه را با batch_size =8 به اندازه 2001 ایپاک آموزش میدهیم و نتایج خروجی بخش generator را هر 200 ایپاک ذخیره میکنیم تا با درک بهتری روند تولید خروجی را ببینیم.

در نهایت پس از اتمام فرایند آموزش، نمودار loss را برای دو بخش generator و discriminator را رسم میکنیم که در شکل 6.2 قابل مشاهده است.

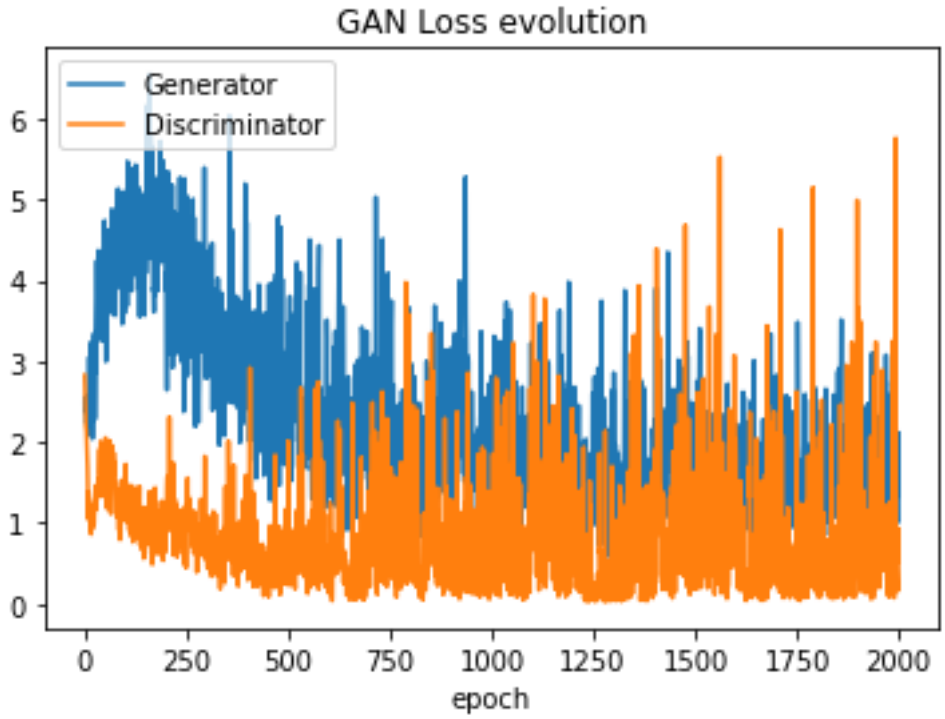

conv2d_7_input | input: | [(None, 28, 28, 1)]
InputLayer | output: | [(None, 28, 28, 1)]
conv2d_7 | input: | (None, 28, 28, 1)
Conv2D | output: | (None, 14, 14, 16)
leaky_re_lu_4 | input: | (None, 14, 14, 16)
LeakyReLU | output: | (None, 14, 14, 16)
dropout_4 | input: | (None, 14, 14, 16)
Dropout | output: | (None, 14, 14, 16)
conv2d_8 | input: | (None, 14, 14, 16)
Conv2D | output: | (None, 7, 7, 32)
zero_padding2d_1 | input: | (None, 7, 7, 32)
ZeroPadding2D | output: | (None, 8, 8, 32)
leaky_re_lu_5 | input: | (None, 8, 8, 32)
LeakyReLU | output: | (None, 8, 8, 32)
dropout_5 | input: | (None, 8, 8, 32)
Dropout | output: | (None, 8, 8, 32)
batch_normalization_5 | input: | (None, 8, 8, 32)
BatchNormalization | output: | (None, 8, 8, 32)
conv2d_9 | input: | (None, 8, 8, 32)
Conv2D | output: | (None, 4, 4, 64)
leaky_re_lu_6 | input: | (None, 4, 4, 64)
LeakyReLU | output: | (None, 4, 4, 64)
dropout_6 | input: | (None, 4, 4, 64)
Dropout | output: | (None, 4, 4, 64)
batch_normalization_6 | input: | (None, 4, 4, 64)
BatchNormalization | output: | (None, 4, 4, 64)
conv2d_10 | input: | (None, 4, 4, 64)
Conv2D | output: | (None, 4, 4, 128)
leaky_re_lu_7 | input: | (None, 4, 4, 128)
LeakyReLU | output: | (None, 4, 4, 128)
dropout_7 | input: | (None, 4, 4, 128)
Dropout | output: | (None, 4, 4, 128)
flatten_2 | input: | (None, 4, 4, 128)
Flatten | output: | (None, 2048)
dense_4_input | input: | [(None, 100)]
InputLayer | output: | [(None, 100)]
dense_4 | input: | (None, 100)
Dense | output: | (None, 6272)
reshape_2 | input: | (None, 6272)
Reshape | output: | (None, 7, 7, 128)
batch_normalization_10 | input: | (None, 7, 7, 128)
BatchNormalization | output: | (None, 7, 7, 128)
up_sampling2d_4 | input: | (None, 7, 7, 128)
UpSampling2D | output: | (None, 14, 14, 128)
conv2d_13 | input: | (None, 14, 14, 128)
Conv2D | output: | (None, 14, 14, 128)
activation_5 | input: | (None, 14, 14, 128)
Activation | output: | (None, 14, 14, 128)
batch_normalization_11 | input: | (None, 14, 14, 128)
BatchNormalization | output: | (None, 14, 14, 128)
up_sampling2d_5 | input: | (None, 14, 14, 128)
UpSampling2D | output: | (None, 28, 28, 128)
conv2d_14 | input: | (None, 28, 28, 128)
Conv2D | output: | (None, 28, 28, 64)
activation_6 | input: | (None, 28, 28, 64)
Activation | output: | (None, 28, 28, 64)
batch_normalization_12 | input: | (None, 28, 28, 64)
BatchNormalization | output: | (None, 28, 28, 64)
conv2d_15 | input: | (None, 28, 28, 64)
Conv2D | output: | (None, 28, 28, 1)
activation_7 | input: | (None, 28, 28, 1)
Activation | output: | (None, 28, 28, 1)

شکل4.2 ساختار **Discriminator** و **Generator**

نتایج خروجی شبکه مولد را در شکل 5.2 مشاهده میکنید.



شکل5.2 خروجیهای شبکه مولد هر 200 ایپاک



شکل6.2 نمودار **loss** مدل **ACGAN**

همانگونه که مشاهده شد، نتایج بسیار خوبی بدست آوردیم و generator توانست از نویز به تصاویر بسیار شبیه به دیتاست تولید کند و از روی نمودار loss هم میبینیم که loss هردو بخش به خوبی کاهش یافته و میبینیم که همگرا نیز شده اند و با مشکل همگرایی نیز روبرو نیستیم.

2-2. شبکه متخاصم مولد Wasserstein

این مدل، مدل تغییر یافته شبکه DCGAN محسوب میشود. این مدل برای برطرف کردن مشکلاتی همچون Diminished Gradient و Mode Collapse که در DCGAN وجود داشت، از لاس Wasserstein استفاده میکند. با استفاده از این لاس جدید بخش discriminator به جای اینکه یک خروجی 0-1 به عنوان اینکه عکس ورودی fake یا real است بدهد، یک مقدار عددی بین 0و1 به عنوان خروجی میدهد، به همین دلیل است که در شبکه WGAN، بخش discriminator را critic نیز مینامند. به همین دلیل نیاز است که در معماری DCGAN در لایه آخر discriminator، از تابع sigmoid استفاده نشود و وزن های discriminator را clip میکنیم تا که مدل پایدار بماند. و به جای استفاده از Adam که در DCGAN به کار برده میشود، از بهینه ساز های بدون مومنتوم همچون RMSProp استفاده میکنیم. این الگوریتم در مقاله مربوطه نیز به همین صورت ذکر شده است که در شکل زیر مشاهده میشود.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

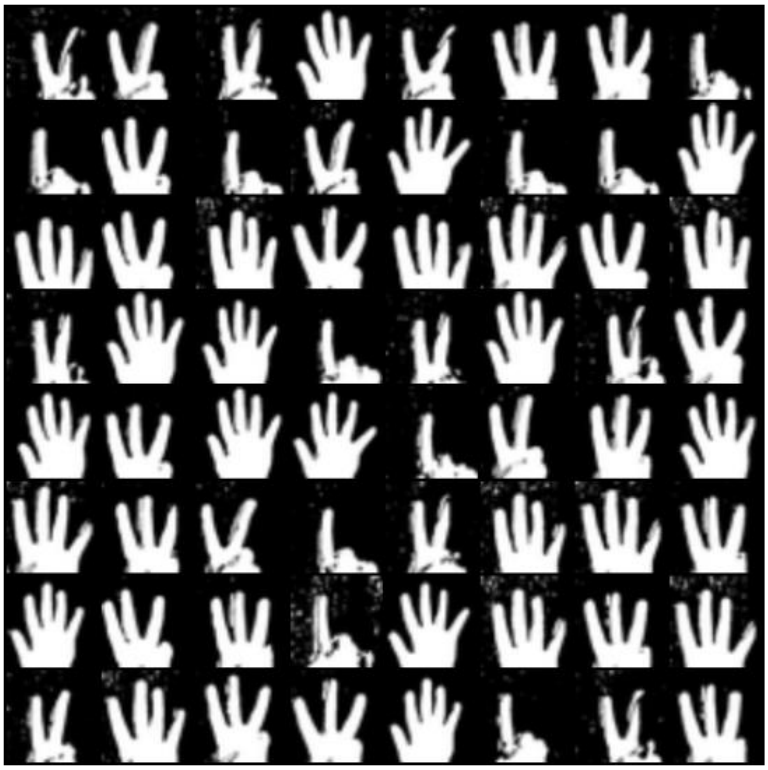
شکل 7.2 الگوریتم **WGAN**

در نهایت پس از استفاده از دیتالودر(با بهره بردن از دستور ImageFolder) که در شکل 8.2 قابل مشاهده است و انجام پیش پردازش روی داده ها، فرایند آموزش WGAN را آغاز میکنیم.

```
transform = transforms.Compose([
    transforms.Resize(IMG_SIZE),
    transforms.CenterCrop(IMG_SIZE),
    transforms.ToTensor(),
    transforms.Normalize(*norm,inplace=True),
])

def unnorm(images, means, stds):
    means = torch.tensor(means).reshape(1,3,1,1)
    stds = torch.tensor(stds).reshape(1,3,1,1)
    return images * stds + means


dataset = datasets.ImageFolder(root="/content/drive/MyDrive/NNDL/HW6/Dataset",transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2)

def show_batch(data_loader):
    for images, labels in data_loader:
        plt.subplots(figsize=(12, 10))
        plt.axis("off")
        raw_images = unnorm(images, *norm)
        plt.imshow(make_grid(raw_images[:BATCH_SIZE], nrow=8).permute(1, 2, 0).clamp(0,1))
        break

show_batch(dataloader)
```

شکل 7.2 الگوریتم **WGAN**

تعدادی از داده های ورودی شبکه به عنوان real، در شکل 8.2 قابل مشاهده است.



شکل8.2 داده های **real**

ساختار شبکه generator بصورت زیر است:

```python
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.cnn = nn.Sequential(
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        return self.cnn(x)

generator = Generator()
generator.to(DEVICE)
```

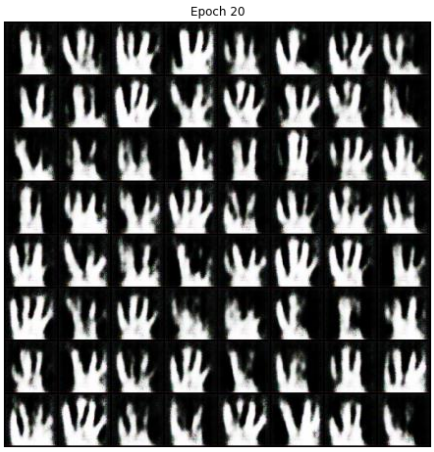
شکل9.2 ساختار **generator** در **WGAN**

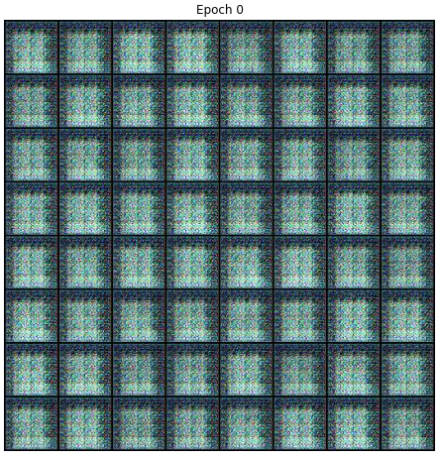
و ساختار discriminator نیز بصورت زیر میباشد:

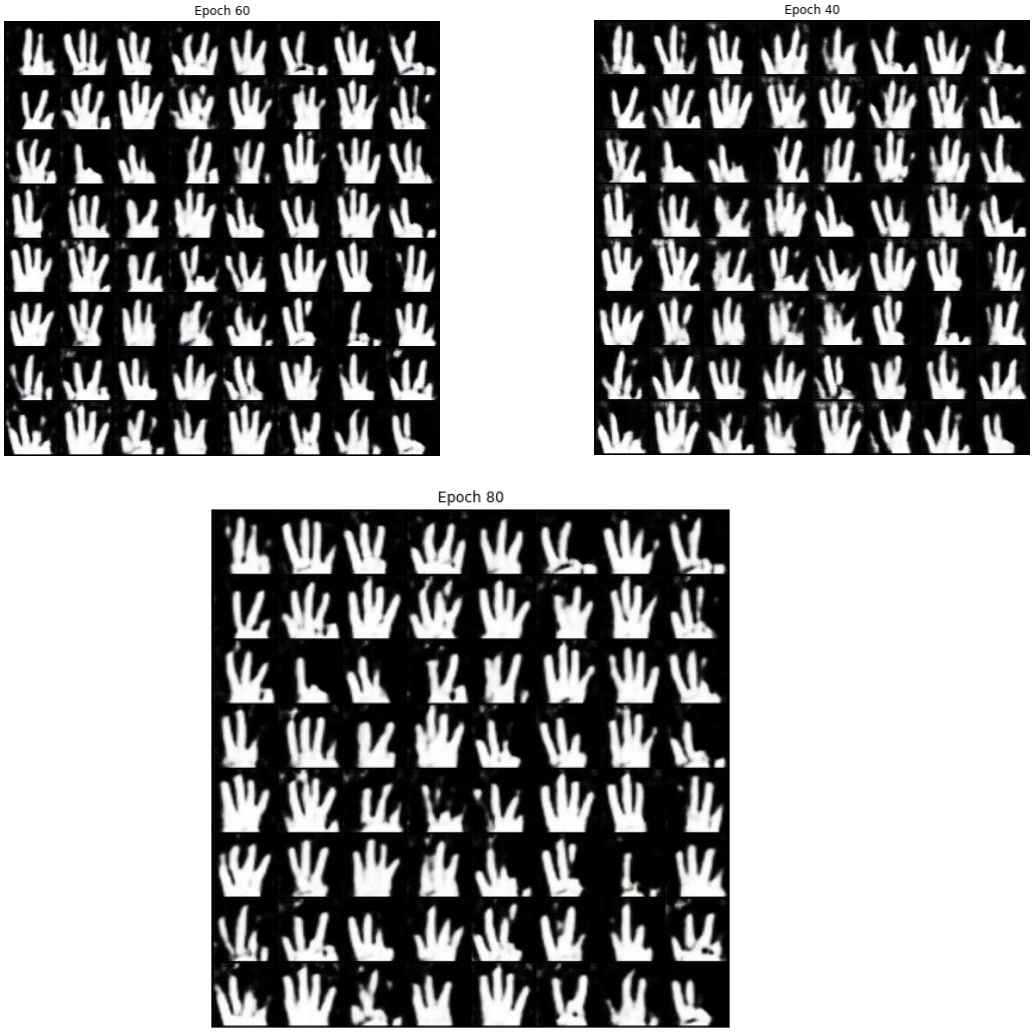
```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False),
#             nn.Sigmoid()
        )

```

شکل 10.2 ساختار **discriminator** در **WGAN**

همانگونه که میدانیم در شبکه های GAN ابتدا discriminator و سپس generator را آموزش میدهیم. با استفاده از این روند و تعداد ایپاک 80، به نتایج مناسب دست پیدا کردیم. سپس خروجی شبکه generator را به ازای هر 20 ایپاک ذخیره کردیم، که نتایج بصورت زیر میباشد.

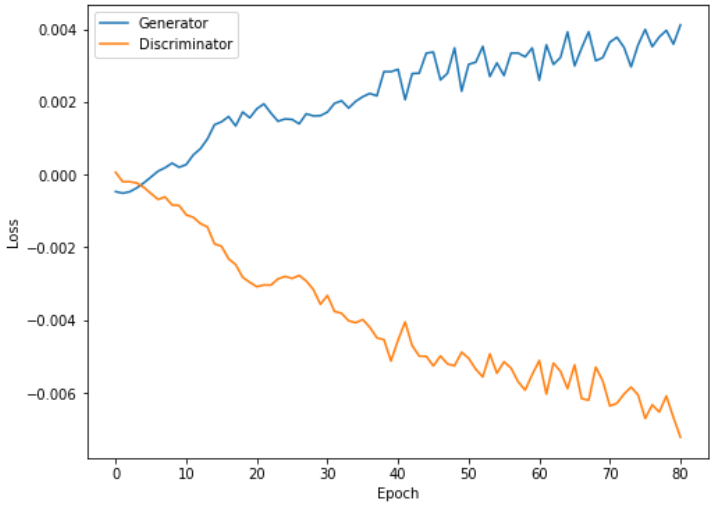






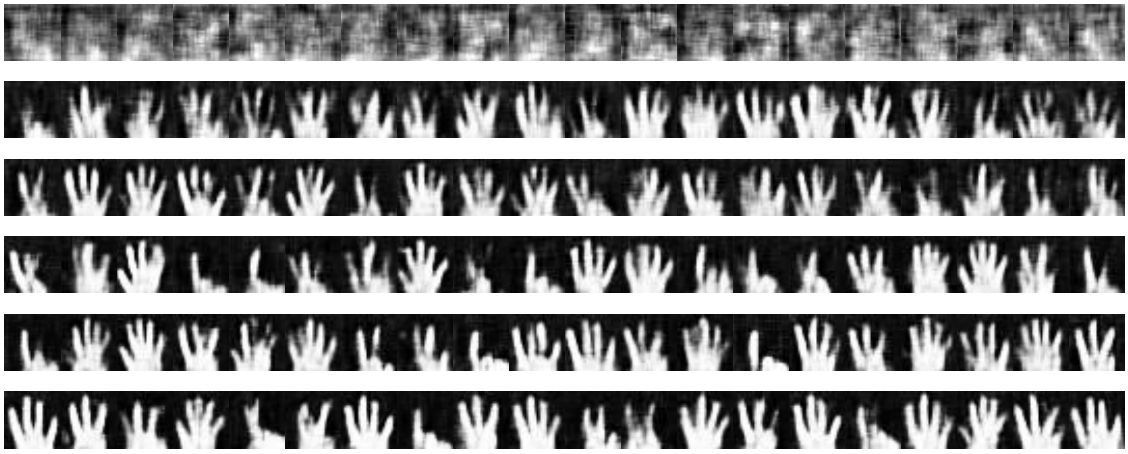
شکل11.2 نتایج مدل **WGAN**

میبینیم که مدل به خوبی آموزش دیده و با افزایش تعداد ایپاک ها، generator توانسته خروجی با کیفیت تری از نویز ورودی خود تولید کند. نمودار loss دو شبکهgenerator و discriminator نیز در شکل 12.2 قابل مشاهده است که دلالت بر همین موضوع دارد و میبینیم که هردو شبکه به خوبی آموزش دیده اند.



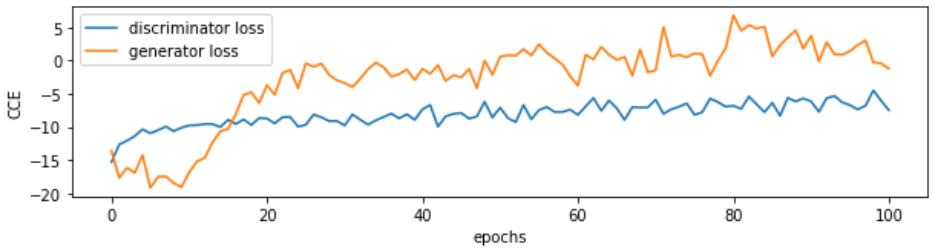
شکل 12.2 نمودار **loss** در **WGAN**

یکی از مشکلات این مدل همانگونه که مشاهده میشود این است که به خوبی همگرا نشده است. این مشکل را میتوان با روش ارائه شده در مقاله و استفاده از gradient penalty به جای weight clipping برطرف کرد. در این شبکه میتوانیم از بهینه ساز Adam نیز استفاده کنیم که باعث افزایش سرع شبکه نیز میشود. نتایج این شبکه را در شکل زیر میتوان مشاهده کرد.



شکل 13.2 نتایج مدل **WGAN-GP**

. نمودار loss دو شبکهgenerator و discriminator در مدل WGAN-GP نیز در شکل زیر دیده میشود که میبینیم همگرایی در این مدل بهبود یافته و نتایج نیز با توجه به شکل 13.2 بهتر هستند.



شکل 14.2 نمودار loss در WGAN-GP