به نام خدا

دانشگاه تهران

دانشکده مهندسی

برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	فاطمه نائینیان – محمد عبائیانی
شماره دانشجویی	810198432 - 810198479
تاریخ ارسال گزارش	1401-11-7

فهرست

پاسخ ۱ – شبکه های مولد تخاصمی کانولوشنال عمیق

(1-1

همانطور که میدانیم شبکه های GAN از دو بخش generator و discriminator تشکیل شده است. حال باید این دو بخش را پیاده سازی کنیم. ابتدا داده های موجود در پوشه ها را میخوانیم و برچسب های آنها را مشخص میکنیم. برای اینکار ابتدا یک دیتاست از ادرس داده ها درست میکنیم سپس همه عکس ها را در غالب یک لیست ذخیره میکنیم. مراحل preprocess نیز به این صورت است که میخواهیم فرمت را float32 میکنیم و داده ها را منهای 127.5 و تقسیم بر 127.5 میکنیم تا همه داده ها بین 1 و 1- قرار بگیرد.

بخش discriminator وظیفه دارد تا برچسب واقعی یا ساختگی بودن به عکس ها بزند و اگر واقعی تشخیص دهد خروجی 1 و اگر ساختگی تشخیص دهد خروجی 0 را میدهد.

سپس با توجه به مقاله بخش discriminator شامل دو لایه کانوولوشن، maxpool و دو لایه fully connected است که با sigmoid طبقه بندی میکند.

```
def disc():
  discriminator = Sequential()
  discriminator.add(Conv2D(64, (5, 5), padding='same', activation=LeakyReLU(), input_shape=(32, 32, 1)))
  discriminator.add(MaxPooling2D(pool_size=(2, 2)))
  discriminator.add(Conv2D(128, (5, 5), activation=LeakyReLU()))
  discriminator.add(MaxPooling2D(pool_size=(2, 2)))
  discriminator.add(Flatten())
  discriminator.add(Dense(1024))
  discriminator.add(Activation('tanh'))
  discriminator.add(Dense(1))
  discriminator.add(Activation('sigmoid'))
  return discriminator
```

شکل 1: تعریف **discriminator**

بخش generator وظیفه دارد تا با گرفتن نویز، یک تصویر ساختگی تولید کند. به گونه ای که بتواند discriminator را به نوعی گول بزند تا داده های ساختگی و واقعی از هم غیر قابل تفکیک شوند.

در generator نیز سه لایه کانوولوشن و سه لایه upsampling و دو لایه fully connected خواهیم داشت که اینکار معادل همان deconvoloution است.

```python
def gen():
  generator = Sequential()
  generator.add(Dense(1024,input_dim=100))
  generator.add(Activation('tanh'))
  generator.add(Dense(1024*4*4))
  generator.add(BatchNormalization())
  generator.add(Activation('tanh'))
  generator.add(Reshape((4, 4, 1024), input_shape=(1024*4*4,)))
  generator.add(UpSampling2D(size=(2, 2)))
  generator.add(Conv2D(256, (5, 5), padding='same'))
  generator.add(Activation('tanh'))
  generator.add(UpSampling2D(size=(2, 2)))
  generator.add(UpSampling2D(size=(2, 2)))
  generator.add(Conv2D(64, (5, 5), padding='same'))
  generator.add(Activation('tanh'))
  generator.add(Conv2D(1, (5, 5), padding='same'))
  generator.add(Activation('tanh'))
  return generator
```

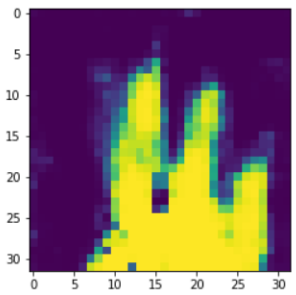
شکل 2: تعریف **generator**

مدل از توالی generator و discriminator به دست می اید. حال مدل را بر روی داده ها اجرا میکنیم و نتایج را مشاهده میکنیم.

```python
generator = gen()
discriminator = disc()
DCGAN =Sequential([generator,discriminator])
discriminator.compile(optimizer='adam',loss='binary_crossentropy')
discriminator.trainable = False
DCGAN.compile(optimizer='adam',loss='binary_crossentropy')
```

شکل 3: ترکیب **generator** و **discriminator** برای ساخت **DCGAN**

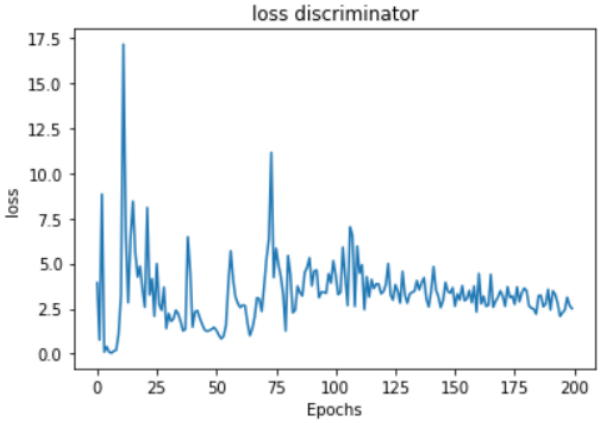
به تعداد 200 ایپاک و با batch size = 100 مدل را ران میکنیم. اینکار توسط یه حلقه for انجام می شود که در هر مرحله ابتدا discriminator را با batch مدنظر از داده های واقعی اموزش میدهیم. سپس اموزش ان را با داده های ساختگی که توسط generator تولید شده ادامه میدهیم. سپس اموزش را متوقف میکنیم سپس با کمک DCGAN که در ان discriminator ثابت است و فقط generator تغییر میکند را اموزش میدهیم. یعنی در واقع فقط generator را اموزش میدهیم. در هر مرحله loss و accuracy را ذخیره میکنیم.



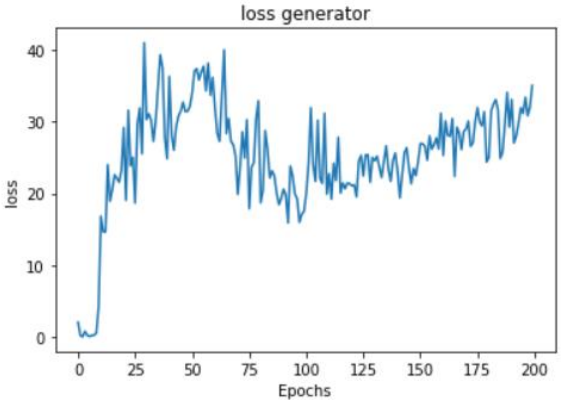
شکل 4: یک نمونه خروجی برای نویز رندوم

2-1)

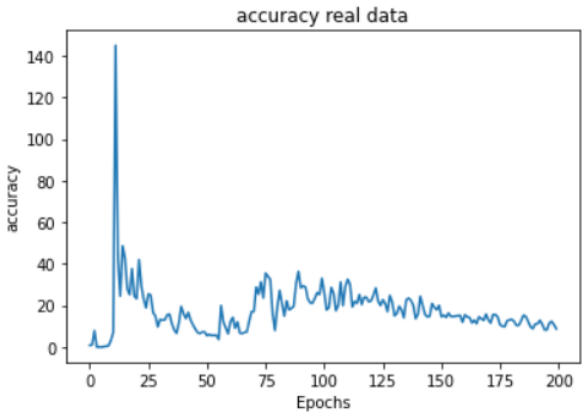
در هرمرحله به ازای هر batch ، loss و accuracy به دست می اید که در نهایت loss کل ایپاک برابر مجموع loss ها و accuracy کل ایپاک برابر میانگین انها خواهد شد. یکی از نشانه های ناپایدار بودن این مدل را در نمودار های زیر میبینیم به گونه ای که تابع evaluate نتوانسته همگرا شود و مقدار بیشتر از 100 برای دقت خروجی داده است. این مشکل در بخش بعدی رفع می شود.



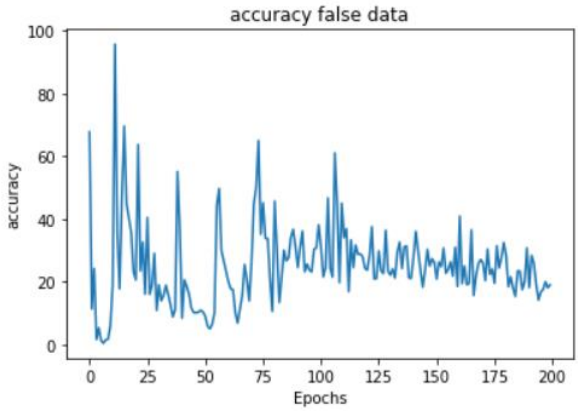
شکل 5: نمودار **loss** برای **discriminator**



شکل 6: نمودار **loss** برای **generator**



شکل 7: نمودار دقت برای داده های اصلی



شکل 8: نمودار دقت برای داده های ساختگی

(3-1

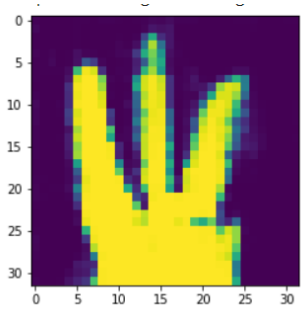
One-sided label smoothing : در این روش به جای اینکه برچسب ها را 0 و 1 کنیم با اعدادی مثل 0.1 یا 0.9 برچسب ها را مشخص میکنیم. با این روش GAN ها در حالت تخاصمی برچسب های smooth تری را به discriminator میدهد که سبب کاهش ریسک و loss می شود به دلیل اینکه discriminator به صورت حریصانه ای برچسب میزند و این موضوع به GAN اسیب زیادی میزند. این برچسب ها دقت هایی را نشان میدهند که برای مثال 0.8 و 0.9 میتواند متعلق به کلاس 1 باشد و 0.2 و 0.3 کلاس 0 است. برای پیاده سازی این روش برچسب های کلاس ها را از 0 و 1 به 0.1 و 0.9 تبدیل میکنیم.

Add noise : در شبکه های GAN یکی از عوامل عدم پایداری به بینهایت میل کردن

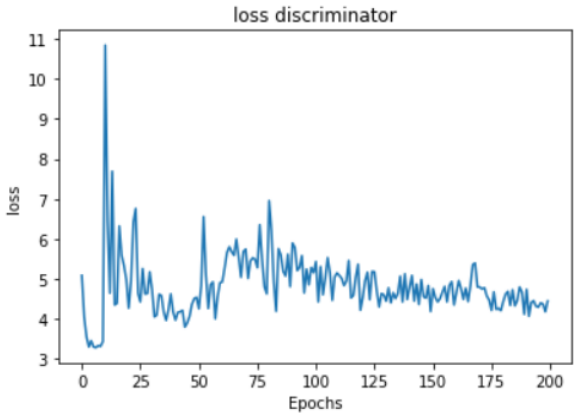
$$2\log 2 - 2JSD(\mathbb{P}_r \| \mathbb{P}_g)$$

است. به همین دلیل اگر داده های اموزش discriminator یا همان داده های حقیقی را با کمک یک نویز نرمال یعنی میانگین صفر و انحراف معیار یونیفرم بین 0 و 0.1 جمع کنیم، به پایداری سیستم و شبکه کمک میکند و جلوی به بینهایت میل کردن مقادیر و ناپایداری را میگیرد.

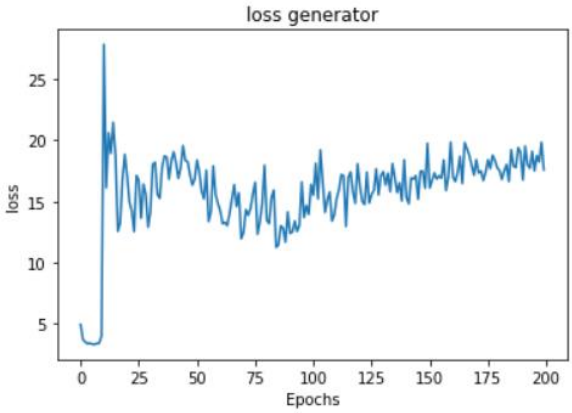
حال با توجه به این تغییرات به نتایج زیر میرسیم.



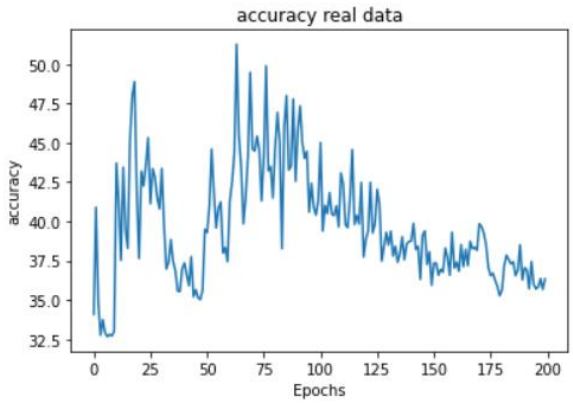
شکل 9: یک نمونه تولید شده توسط مدل به ازای داده های اصلی همراه با نویز



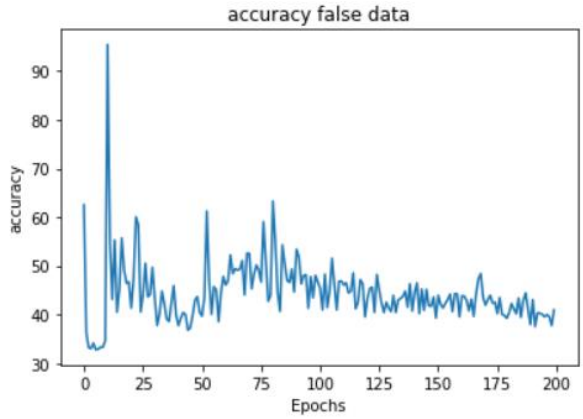
شکل 10: نمودار **loss** برای **discriminator** همراه با نویز



شکل 11: نمودار **loss** برای **generator** همراه با نویز



شکل 12: نمودار دقت برای داده های اصلی همراه با نویز



شکل 13: نمودار دقت برای داده های ساختگی همراه با نویز

از مقایسه دو حالت میفهمیم که با پایدار کردن مدل، دقت و loss نیز اوضاع بهتری پیدا میکنند و دقت برای داده های ساختگی به طور میانگین حدود 5 درصد افزایش، و حدود 10 درصد برای داده های اصلی افزایش یافته است. Loss نیز مقدار چشم گیری کاهش یافته است.

پاسخ ۲ – شبکه متخاصم مولد طبقه بند کمکی و شبکه Wasserstein

2-1)

شبکه های ACGAN در بخش generator علاوه بر ورودی نویز، برچسب را نیز میگیرد. این ویژگی شبکه های ACGAN را بسیار مشهور کرده است. همانند بخش قبلی generator یک داده ساختگی تولید میکند و سپس ان را به discriminator میدهد. این بار discriminator علاوه بر اینکه ساختگی یا اصلی بودن داده را تشخیص میدهد، برای ان برچسب نیز تعیین میکند. بنابراین شبکه میتواند تلاش کند تا داده ای تولید کند که برچسب ان را ما مشخص کرده ایم.

فرایند ورودی گرفتن همانند بخش قبل است اما باید تابع generator به گونه ای تغییر کند تا برچسب نیز در ان اثر داشته باشد. چون 5 کلاس داریم، برچسب ها را به صورت one hot در می اوریم. سپس ورودی generator به صورت concat ورودی نویز و برچسب خواهد بود. خروجی آن نیز عکسی با ابعاد مد نظر خواهد بود.

Layer (type)	Output Shape	Param #	Connected to
input_17 (InputLayer)	[(None, 100)]	0	[]
input_16 (InputLayer)	[(None, 1)]	0	[]
dense_21 (Dense)	(None, 24576)	2482176	['input_17[0][0]']
embedding_5 (Embedding)	(None, 1, 50)	250	['input_16[0][0]']
activation_15 (Activation)	(None, 24576)	0	['dense_21[0][0]']
dense_20 (Dense)	(None, 1, 64)	3264	['embedding_5[0][0]']
reshape_11 (Reshape)	(None, 8, 8, 384)	0	['activation_15[0][0]']
reshape_10 (Reshape)	(None, 8, 8, 1)	0	['dense_20[0][0]']
concatenate_5 (Concatenate)	(None, 8, 8, 385)	0	['reshape_11[0][0]', 'reshape_10[0][0]']
conv2d_transpose_10 (Conv2DTranspose)	(None, 16, 16, 192)	1848192	['concatenate_5[0][0]']
batch_normalization_20 (BatchNormalization)	(None, 16, 16, 192)	768	['conv2d_transpose_10[0][0]']
activation_16 (Activation)	(None, 16, 16, 192)	0	['batch_normalization_20[0][0]']
conv2d_transpose_11 (Conv2DTranspose)	(None, 32, 32, 1)	4801	['activation_16[0][0]']
activation_17 (Activation)	(None, 32, 32, 1)	0	['conv2d_transpose_11[0][0]']

شکل 14: مدل **generator**

حال به سراغ discriminator می رویم. این تابع عکس تولید شده توسط generator را میگیرد و یک خروجی ان برای تشخیص ساختگی یا اصلی بودن داده استفاده می شود و 5 خروجی ان کلاس متعلق به تصویر را پیش بینی میکند. برای پیش بینی ساختگی یا اصلی بودن از تابع sigmoid استفاده می شود و برای پیش بینی کلاس مربوطه از softmax استفاده می شود.

Layer (type)	Output Shape	Param #	Connected to
input_18 (InputLayer)	[(None, 32, 32, 1)]	0	[]
conv2d_20 (Conv2D)	(None, 16, 16, 32)	320	['input_18[0][0]']
leaky_re_lu_20 (LeakyReLU)	(None, 16, 16, 32)	0	['conv2d_20[0][0]']
dropout_20 (Dropout)	(None, 16, 16, 32)	0	['leaky_re_lu_20[0][0]']
conv2d_21 (Conv2D)	(None, 16, 16, 64)	18496	['dropout_20[0][0]']
batch_normalization_21 (BatchNormalization)	(None, 16, 16, 64)	256	['conv2d_21[0][0]']
leaky_re_lu_21 (LeakyReLU)	(None, 16, 16, 64)	0	['batch_normalization_21[0][0]']
dropout_21 (Dropout)	(None, 16, 16, 64)	0	['leaky_re_lu_21[0][0]']
conv2d_22 (Conv2D)	(None, 8, 8, 128)	73856	['dropout_21[0][0]']
batch_normalization_22 (BatchNormalization)	(None, 8, 8, 128)	512	['conv2d_22[0][0]']
leaky_re_lu_22 (LeakyReLU)	(None, 8, 8, 128)	0	['batch_normalization_22[0][0]']
dropout_22 (Dropout)	(None, 8, 8, 128)	0	['leaky_re_lu_22[0][0]']
conv2d_23 (Conv2D)	(None, 8, 8, 256)	295168	['dropout_22[0][0]']
batch_normalization_23 (BatchNormalization)	(None, 8, 8, 256)	1024	['conv2d_23[0][0]']
leaky_re_lu_23 (LeakyReLU)	(None, 8, 8, 256)	0	['batch_normalization_23[0][0]']
dropout_23 (Dropout)	(None, 8, 8, 256)	0	['leaky_re_lu_23[0][0]']
flatten_5 (Flatten)	(None, 16384)	0	['dropout_23[0][0]']
dense_22 (Dense)	(None, 1)	16385	['flatten_5[0][0]']
dense_23 (Dense)	(None, 5)	81925	['flatten_5[0][0]']

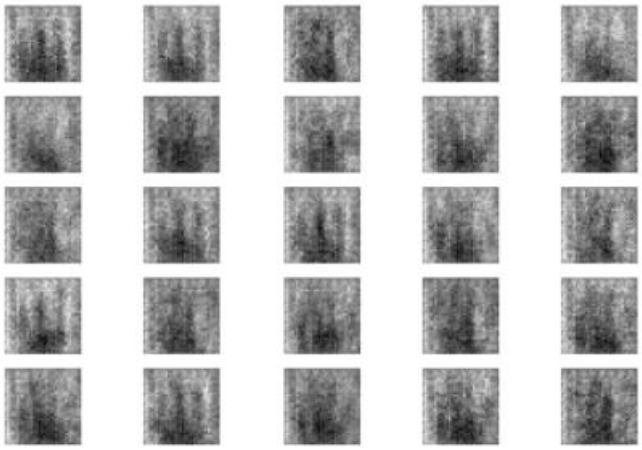
Total params: 487,942
Trainable params: 487,046
Non-trainable params: 896

شکل 15: مدل **discriminator**

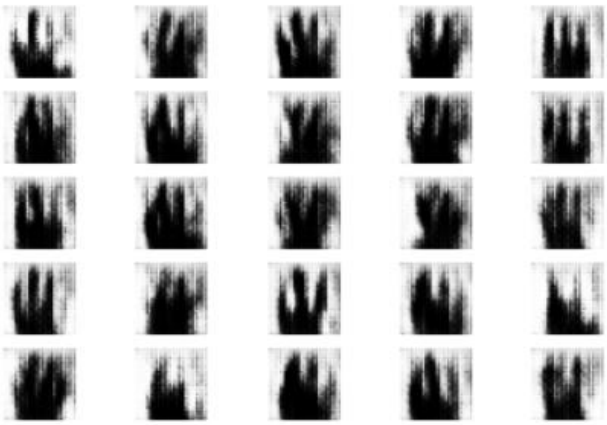
همانطور که میدانیم ACGAN از ترکیب generator و discriminator تشکیل می شود. پس ان را تشکیل میدهیم و به ازای اپتیمایز adam(lr=0.0002, beta_1=0.5) و loss = binary_crossentropy مدل را کامپایل میکنیم.

همانند بخش قبل به تعداد ایپاک دلخواه به ازای batch های مختلف مدل را اجرا میکنیم. در هر batch یک دسته ای از داده های اصلی را جدا میکنیم و دسته ای داده نویز تولید میکنیم و برچسب ها را مشخص میکنیم. برای این کار ها دو تابع generate_fake_sample و generate_real_sample استفاده میکنیم. generate_real_sample به تعدادی batch از مجموعه اصلی دیتا جدا میکند. generate_fake_sample ابتدا با کمک generate_latent_points تعدادی نویز تولید میکند و سپس ان را به generator میدهد و داده های ساختگی را تولید میکند. توجه شود که در هر سه تابع برچسب ها نیز تولید می شود. در generate_real_sample برچسب اصلی بودن انها 1 می شود و برچسب کلاس های انها نیز به همراه داده ها خروجی داده می شود. در generate_fake_sample برچسب اصلی بودن انها صفر می شود و برچسب هایی به صورت رندوم بین کلاس 0 تا 4 به هر داده نویز اختصاص داده می شود.

حال به ازای batch = 100 و 200 ایپاک مدل را اموزش میدهیم. همانند بخش قبل هنگامی که generator اموزش میبیند discriminator ثابت است و هنگامی که discriminator اموزش میبیند، generator ثابت است.



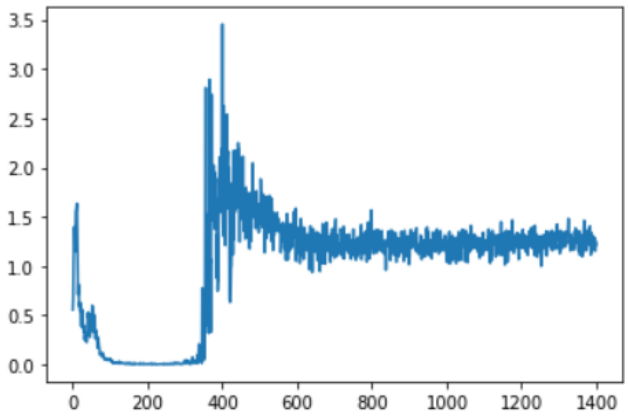
شکل 16: تصاویر **generate** شده در ایپاک اول



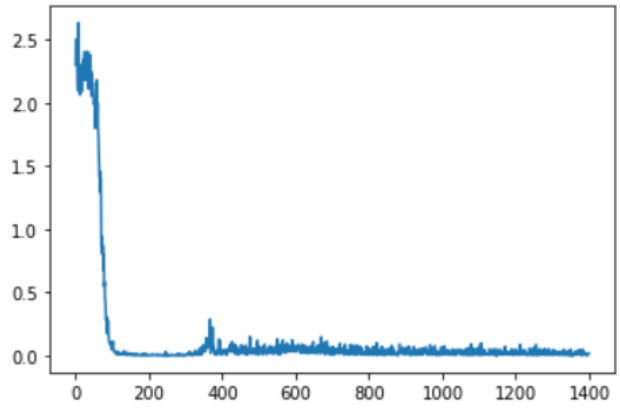
شکل 17: تصاویر **generate** شده در ایپاک 100

شکل 18: تصاویر **generate** شده در ایپاک 200

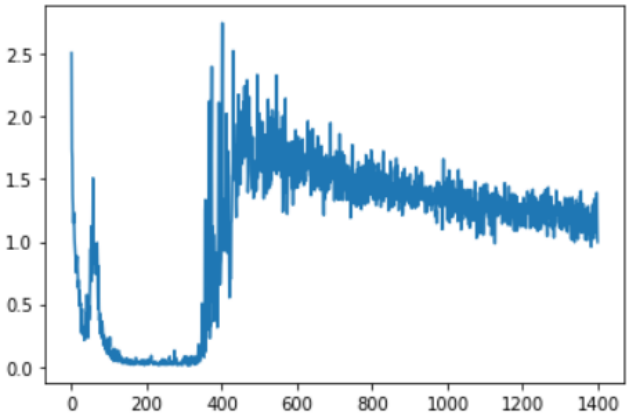
میبینیم مدل توانسته تا حد خوبی کلاس ها را تفکیک کند اما بعد از گذشت 200 ایپاک هنوز خطای خیلی زیادی دارد و بنظر می اید به ازای ایپاک های خیلی زیاد ممکن است مدل همگرا شود. مدل برای 1 و 2 بیشتر از سایر کلاس ها مشکل و خطا دارد. به دلیل کمبود وقت امکان اجرای مدل بر روی ایپاک های بیشتر را نداشته ایم.



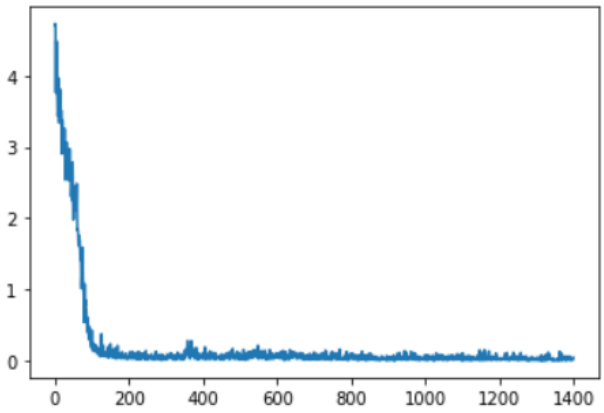
شکل 19: نمودار **loss** برای **generator** به ازای تفکیک ساختگی یا اصلی بودن



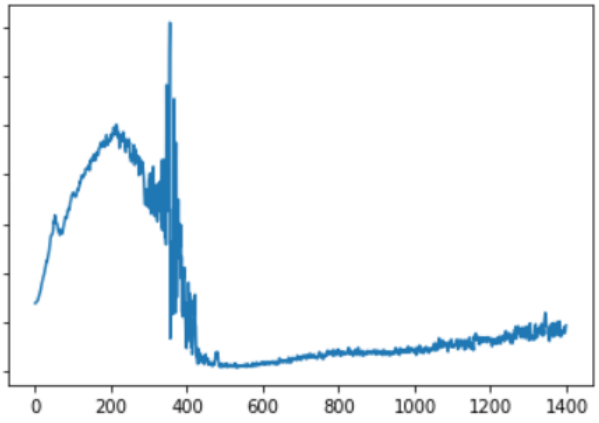
شکل 20: نمودار **loss** برای **generator** به ازای تفکیک کلاس ها و برچسب ها



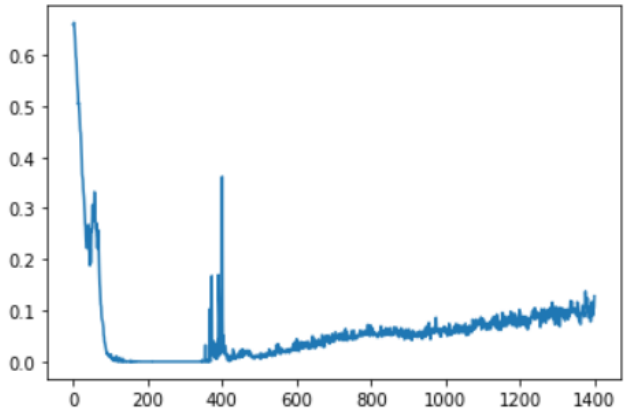
شکل 21 : نمودار **loss** برای **discriminator** به ازای تفکیک ساختگی یا اصلی بودن



شکل 22 : نمودار **loss** برای **discriminator** به ازای تفکیک کلاس ها و برچسب ها

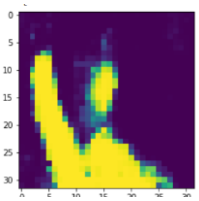


شکل 23: نمودار دقت برای **generator**

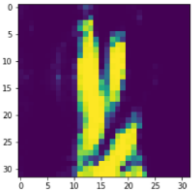


شکل 24: نمودار دقت برای **discriminator**

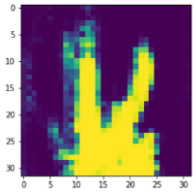
حال در نهایت به ازای 5 نویز و 5 برچسب مختلف از مدل خروجی میگیریم تا ان را مقایسه کنیم.



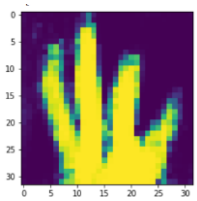
شکل 25: خروجی مدل به ازای برچسب 1



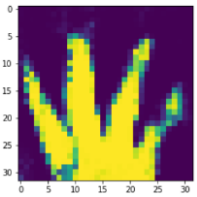
شکل 26: خروجی مدل به ازای برچسب 2



شکل 27: خروجی مدل به ازای برچسب 3



شکل 28: خروجی مدل به ازای برچسب 4



شکل 29: خروجی مدل به ازای برچسب 5

مجدد مشاهده میشود که مدل خطای زیادی دارد و نیازمند تغییراتی در تعداد ایپاک و نوع generator و discriminator است.

(2-2

در بخش قبل تابع loss استفاده شده binary_crossentropy بود و در این بخش میخواهیم از گرادیان پنالتی استفاده کنیم. دلیل اینکار این است که هنگام استفاده از binary_crossentropy ، vanishing gradient رخ میدهد که سبب می شود تا مدل توانایی لازم را در پیش بینی و تشخیص همه کلاس ها را نداشته باشد و فقط بتواند یک کلاس را تشخیص دهد. ولی در روش Wasserstein توزیع عکس اصلی و ساختگی بیشتر نزدیک می شوند. این loss به صورت زیر تعریف می شود:

$$loss\ Wasserstein = E\big(D(real) - D(fake)\big) = K.mean\big(y_{true} * y_{pred}\big)$$

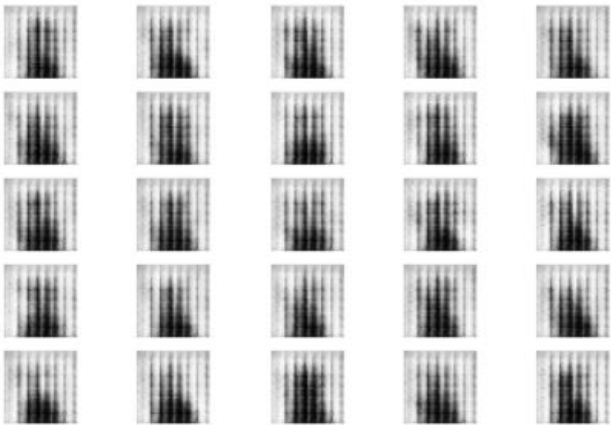
```python
import keras.backend as K

def wasserstein_loss( y_true, y_pred):
        y_true = float(y_true)
        y_pred = float(y_pred)
        return K.mean(y_true * y_pred)
```

شکل 30: نحوه پیاده سازی **loss wasserstein**

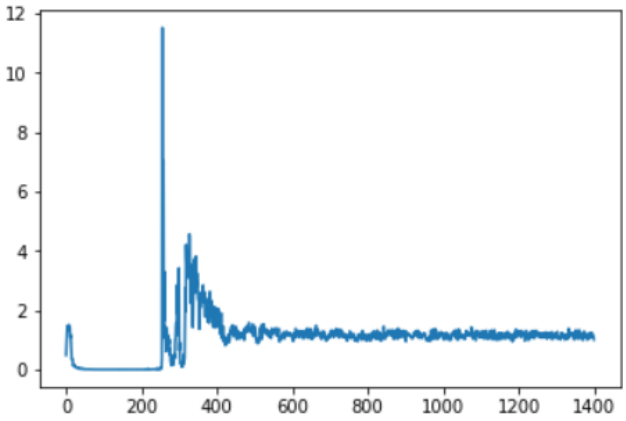
در این روش discriminator سعی میکند تا مقدار ان را افزایش دهد (یعنی تشخیص دهد کدام ساختگی و کدام اصلی است) و از طرفی generator سعی میکند تا ان را کاهش دهد (یعنی عکس هایی تولید کند که توانایی گول زدن discriminator را داشته باشد).

همه مراحل همانند بخش قبل اجرا میشوند و فقط از این تابع هزینه به جای تابع هزینه قبلی استفاده می شود.

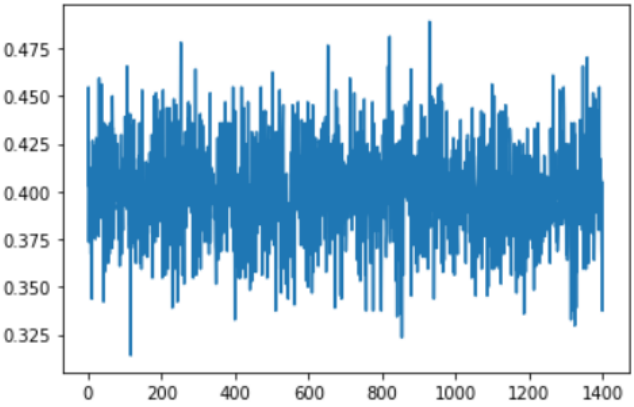


شکل 31: خروجی مدل جدید در ایپاک های ابتدایی

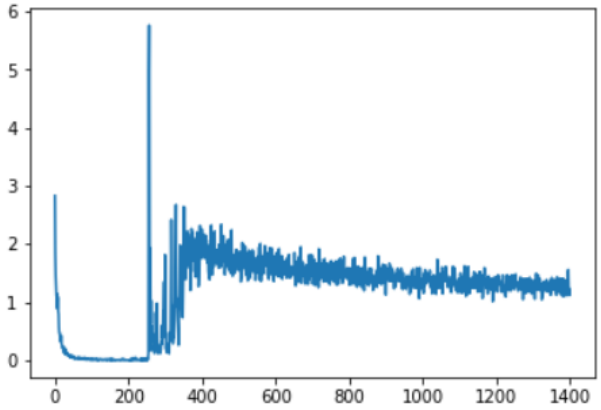
شکل 32: خروجی مدل جدید در ایپاک های نهایی



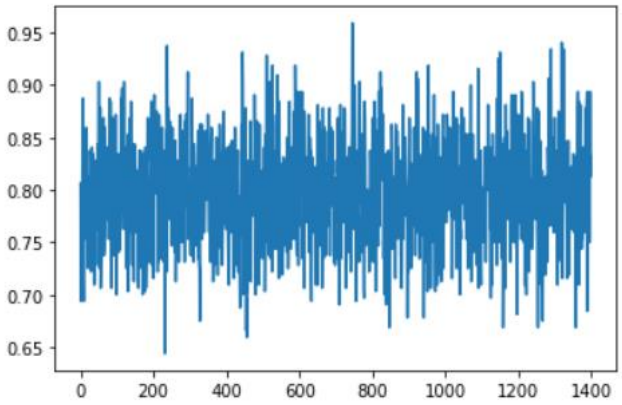
شکل 33: نمودار **loss** جدید برای **generator** به ازای تفکیک ساختگی یا اصلی بودن



شکل 34 : نمودار **loss** جدید برای **generator** به ازای تفکیک کلاس ها و برچسب ها



شکل 35: نمودار **loss** جدید برای **discriminator** به ازای تفکیک ساختگی یا اصلی بودن



شکل 36: نمودار **loss** جدید برای **discriminator** به ازای تفکیک کلاس ها و برچسب ها