991

Face Generation with GANs

Sepideh Bahrami 960122680003
Tahere Hemmati 960122680017
Hesam Ghadimi 960122681003

آشنایی با Generative Adversarial Networks (GANs)

واژه ی generative در GAN ها در واقع دسته ای از مدل های آماری را توصیف می کند که در تقابل با مدل های discriminative هستند.به طور کلی در GAN ها دو شبکه ی عصبی وجود دارند که به نوعی در حال رقابت با همدیگر هستند یکی از این شبکه ها Generator و دیگری Discriminator نامیده می شوند. که به طور خلاصه کار های زیر را انجام می دهند.

- Generator : generator یاد می گیرد که داده های قابل قبول تولید کند. نمونه های تولید شده به نمونه های آموزشی منفی برای discriminator تبدیل می شوند.
- Discriminator : discriminator آموخته است که داده های جعلی ژنراتور را از داده های واقعی تشخیص دهد. discriminator تولید کننده را به دلیل تولید نتایج غیرقابل قبول مجازات می کند.

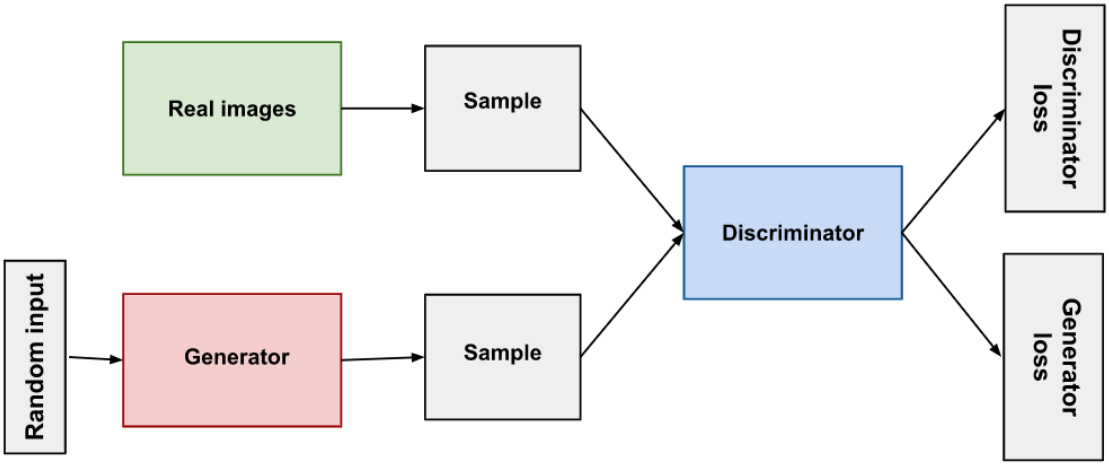
هنگامی که آموزش شروع می شود ، generator داده های جعلی تولید می کند و discriminator سریع می آموزد که جعلی است برای فهم بیشتر این موضوع به شکل زیر توجه کنیم:

با پیشرفت آموزش، generator به تولید خروجی نزدیک می شود که می تواند discriminator را گول بزند:

سرانجام، اگر آموزش generator به خوبی پیش برود، discriminator در تشخیص تفاوت واقعی و جعلی بدتر می شود. شروع به طبقه بندی داده های جعلی به عنوان واقعی می کند و از دقت آن کاسته می شود. که باز این موضوع هم در شکل زیر آورده شده است.

شکل (۱)

شـــکل زیر کل ســـیســـتم را نشـــان می دهد و همانطور که در شـــکل هم مشـــاهده می شـــود generator و discriminator هر دو شبکه عصبی هستند. خروجی generator مستقیماً به ورودی discriminator متصل می شود. از طریق backpropagation ، discriminator (discriminator classification) سیگنالی را فراهم می کند که generator از آن برای به روزرسانی وزن خود استفاده می کند.



شکل (۲)معماری کلی شبکه‌های GAN

توضیحات مربوط به پروژه‌ی face generation

در این پروژه از DCGAN که مخفف Deep Convolutional Generative Adversarial Network اســـت برای face generation استفاده شده است. DCGAN ها هم بسیار به شبکه‌های GAN معمولی شباهت دارند ولی در این شبکه‌های GAN از Deep Convolutional layers به جای Fully Connected Layers استفاده شده است. و شبکه‌های کانولوشنالی هم به طورکلی correlation areas را در یک تصویر مشخص می‌کنند.

جزئیات پیاده سازی و معماری شبکه و توضیحات مربوط به بخش‌های مختلف کد

- **موارد موردنیاز قبل از شروع اجرای پروژه**

بـــرای اجـــرای ایـــن پـــروژه ابتـــدا ســـه پوشـــه بـــه نام‌هـــای gifs، plots و training_samples بایـــد در قســـمت content گوگـــل کولـــب ســـاخته شـــوند تـــا در رونـــد اجـــرای برنامـــه بـــه تـــدریج مـــوارد مربـــوط بـــه هـــر پوشه در آن‌ها قرار بگیرند.

- **Import کردن کتابخانه‌ها**

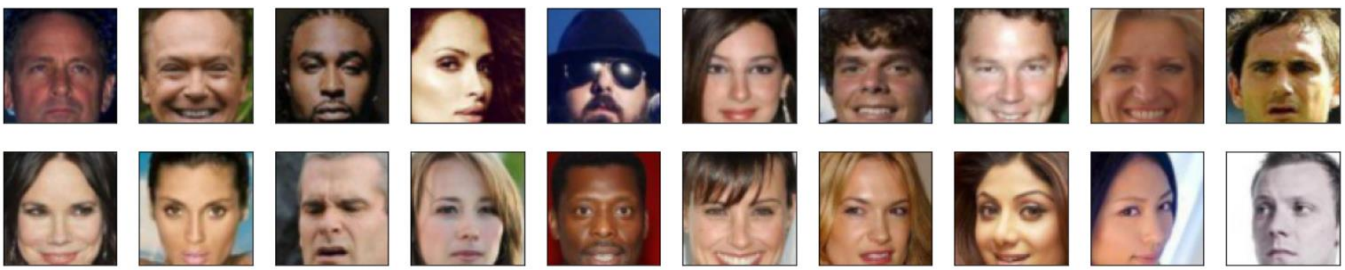
در این بخش از پروژه که ابتدایی‌ترین بخش آن است همه‌ی کتابخانه‌های لازم را import می‌کنیم. لازم به ذکر است که اگر این پروژه در گوگل کولب ران شود نیاز به نصب هیچ کتابخانه و فریم ورکی نیست و فقط باید کتابخانه‌های موجود در تصویر زیر import شوند.

```
# importing libraries
import torch
from torchvision import datasets
from torchvision import transforms
import pickle as pkl
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
# Important libraries for creating models.
import torch.nn as nn
import torch.nn.functional as F
from collections import OrderedDict
import torch.optim as optim
# Importing libraries for creating gifs
import matplotlib.animation as animation
import torchvision.utils as vutils
import imageio
import os
```

همه‌ی کتابخانه‌های لازم برای اجرای پروژه‌ی Face Generation

- **دیتاست استفاده شده:**

در این پروژه از دیتاست CelebA استفاده شده است. این دیتاست از مجموعه‌ای شامل ۱۰۰۰۰ هویت و به طور کلی ۲۰۰۰۰۰ عکس تشکیل شده است. که سایز این عکس‌ها در اصل ۱۶۰ × ۱۶۰ پیکسل است، ولی در این پروژه از ورژن rescale شده‌ی آن که سایز ۶۴ × ۶۴ پیکسل دارند استفاده شده است. تعداد عکسی که در این پروژه مورد استفاده قرار گرفته است ۳۲۶۰۰ عدد است. در شکل(۳) تصاویری از عکس‌های متعلق به این دیتاست آورده شده است.



شکل(۳) عکس‌هایی از دیتاست CelebA

- **دانلود Dataset این پروژه**

دیتاست استفاده شده در این پروژه را می‌توان از طریق لینک زیر و به ترتیبی که در تصویر نشان داده شده است دانلود و سپس از حالت زیپ خارج کرد. پس از خارج کردن آن از حالت زیپ شده پوشه‌ای به نام processed-cellba-small در بخش content تشکیل می‌شود.

لینک دانلود دیتاست این پروژه

```
1 !wget "https://s3.amazonaws.com/video.udacity-data.com/topher/2018/November/5be7eb6f_processed-celeba-small/processed-celeba-small.zip"
```

```
-2021-01-07 18:31:20--  https://s3.amazonaws.com/video.udacity-data.com/topher/2018/November/5be7eb6f_processed-celeba-small/processed-celeba-small.zip
esolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.96.173
onnecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.96.173|:443... connected.
TTP request sent, awaiting response... 200 OK
ength: 247694507 (236M) [application/zip]
aving to: ‘processed-celeba-small.zip’

rocessed-celeba-sm 100%[===================>] 236.22M  56.1MB/s    in 4.2s

021-01-07 18:31:24 (56.5 MB/s) - ‘processed-celeba-small.zip’ saved [247694507/247694507]
```

```
1 !unzip "/content/processed-celeba-small.zip"
```

- **Data Preprocessing**

در این بخش بر طبق تصویر آمده در زیر ابتدا مسیردیتاها معلوم می‌کنیم و سپس بوسیله‌ی تابع fn_trainloder آنها را لود می‌کنیم. و درمراحل بعد بخش‌های مربوط به حاضر سازی دیتا برای استفاده به شکل تنسور و یا تغییر سایز آن و از این دست کارها را اعمال می‌کنیم.

```
1 data_dir = '/content/processed_celeba_small/celeba'
```

```
1 def fn_Dataloader(batch_size, image_size, data_dir='/content/processed_celeba_small/celeba'):#real name get_dataloader
2   """
3   batch_size : The size of each batch(the number of images in a batch).
4   img_size: The square size of the image data (x, y).
5   data_dir: Directory where image data is located
6   Return Data that shuffles and batches tensor images.
7   """
8   data_transform = transforms.Compose([transforms.Resize(image_size), transforms.ToTensor()])
9   train_data = datasets.ImageFolder(data_dir, transform= data_transform)
10  train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=0 )
11
12  return train_loader
13
```

- **rescale کردن دیتای ورودی**

با استفاده از تابع make-scale دیتای ورودی که عکس هستند را به محدوده‌ی (1 ,1-) می‌بریم تا با عکس‌های تولید شده توسط generator که خروج یافته از تابع Tanh هستند و در همین محدوده قرار دارند هم scale باشند.

```python
def make_scale(t_img, f_range=(-1, 1)):
  #real name scale
  min, max = f_range
  t_img = t_img * (max - min) + min

  return t_img

```

```python
# Checking if scaling has occured
img = images[0]
scaled_image = make_scale(img)
print('Min: ', scaled_image.min())
print('Max: ', scaled_image.max())
```

```
Min:  tensor(-1.)
Max:  tensor(0.9529)
```

تابع make_scale

- **توضیحات راجع به معماری کلی DCGAN به کار رفته در این پروژه**

به نظر میرسد توجه به موارد زیر در رابطه با GAN ها و به طور جزئی DCGAN ها که در این پروژه مورد استفاده قرار گرفته‌اند به درک بهتر ساختار معماری دو شبکه‌ی عصبی generator و discriminator بسیار کمک خواهد کرد.

۱. در DCGAN ها از convolutional layers برای discriminator و Transposed convolutional layers برای generator ها استفاده شده است.
۲. به جای pooling layers ها که در convolutional neural networks رایج هستند، از strided convolutions در discriminator و fractional strided در generator ها استفاده شده است. دلیل این موضوع این است که در GAN ها با این جایگزینی به شبکه اجازه می‌دهیم تا فضای downsampling در discriminator ها و upsampling در generator ها توسط خود شبکه‌ها یادگیری شود.
۳. از btchnorm ها هم در discriminator ها و هم در generator ها استفاده شده است. btchnorm ها باعث می شوند که نتیجه‌ای که بدست می‌آید در بازه ی استاندار با میانگین صفر و واریانس ۱ قرار گرفته و فرایند یادگیری stable پیش برود. و استفاده از آن ها در همه ی لایه های هر دو discriminator و generator به غیر از لایه ی اخر generator و لایه ی اول discriminator توصیه می‌شود.
۴. در DCGAN ها Fully connected hidden layer ها از deeper architecture ها حذف شده اند.
۵. یکی از activation function هایی که دراین معماری به کار رفته است RelU activation function نام دارد که یکی از دلایل استفاده از آن این است که این تابع فعالساز راه حلی در مقابل مسئله‌ی Vanishing gradient است. البته لازم به ذکر است که از RelU ها در generator ها و از ReakyRelU ها که ورژنی از RelU ها هستند که محدوده‌های کمتر از صفر را هم حساب

می‌کنند در discriminator ها استفاده می‌شود. همینطور در همه‌ی لایه‌های generator از این تابع استفاده می‌شود به غیر از لایه‌ی آخر که تابع فعالساز آن Tanh است.

- **پیاده سازی Generative Model**

همانطور که در تصویر کد مربوط به این بخش آمده است generator از ۵ لایه تشکیل شده است که همه‌ی لایه‌ها به غیر از آخری از strided convolutional transpose ، batch norm و ReLU تشکیل شده است. و لایه‌ی آخر نیز از strided convolutional transpose و Tanh تشکیل شده است.ورودی generator یک بردار latent است، که از یک توزیع نرمال استاندارد گرفته شده است. خروجی یک عکس RGB ۳× ۶۴ × ۶۴ است. strided conv transpose به بردار latent این اجازه را می‌دهد تا به حجمی با همان shape تصویر تبدیل شود.

```
class Generator(nn.Module):

    def __init__(self, z_dim, hidden_dim):

        super(Generator, self).__init__()
        self.hidden_dim = hidden_dim
          # Build the neural network
        self.generator = nn.Sequential(
            self.make_generator_block( z_dim, hidden_dim * 8, stride = 1, padding= 0 ),
            self.make_generator_block( hidden_dim * 8, hidden_dim * 4),
            self.make_generator_block( hidden_dim * 4, hidden_dim * 2),
            self.make_generator_block( hidden_dim * 2, hidden_dim),
            self.make_generator_block( hidden_dim , 3,final_layer=True),)

    def make_generator_block(self, input_channels, output_channels, kernel_size=4, stride=2, padding = 1, bias = False, final_layer=False):
        # Build the neural block
        if not final_layer:
            return nn.Sequential( nn.ConvTranspose2d(input_channels, output_channels, kernel_size,  stride, padding , bias),
                    nn.BatchNorm2d(output_channels),
                    nn.ReLU(inplace=True),)
        else:
            return nn.Sequential(nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride, padding, bias),
                nn.Tanh(),)


    def forward(self, x):
        out = self.generator(x)
        return out
print(Generator(100,64))
```

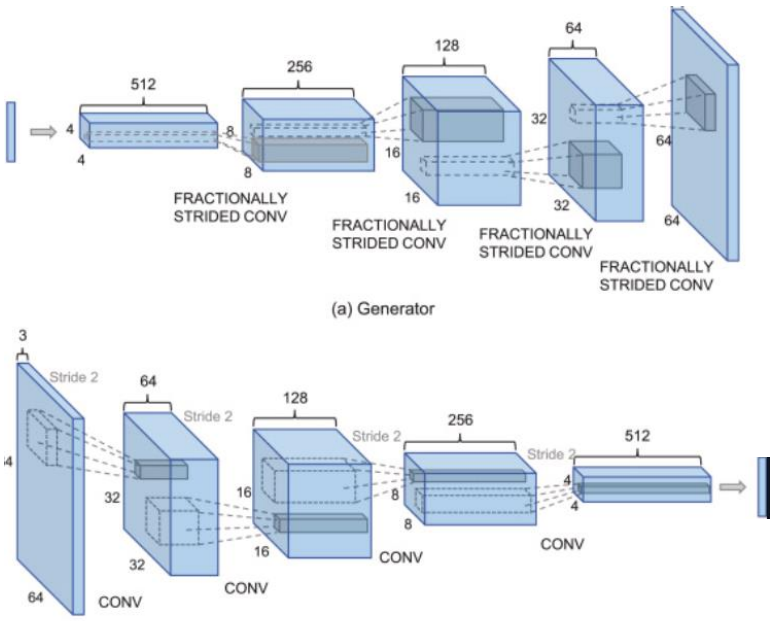
تصویر مربوط به پیاده سازی معماری generator

- **پیاده سازی Discriminative Model**

همانطور که در تصویر کد مربوط به این بخش آمده است discriminator از ۵ لایه تشکیل شده است که همه‌ی لایه‌ها به غیر ازاولی و آخری از strided convolution ، batch norm و از تابع فعالساز LeakyReU تشکیل شده اند و شیب این تابع فعالساز در این پروژه برابر ۰,۲ قرار داده شده است. در لایه‌ی اول batch norm و در لایه‌ی آخر تابع فعاساز نمی‌آید دلیل این موضوع هم

این است که در بخش Loss function از BCELoss استفاده شده است که خودش ترکیبی از دو تابع sigmoid activation function و Cross Entropy Loss است. پس چون در اینجا هم تابه سیگوید را داریم لزومی به تکرار این تابع در لایه‌ی آخر discriminator نیست. ورودی یک تصویر ورودی ۳× ۶۴ × ۶۴ است و خروجی یک احتمال است که نشان دهنده‌ی این است که ورودی از توزیع واقعی داده است یا نه.

```
lass Discriminator(nn.Module):
  def __init__(self, hidden_dim):
      super(Discriminator, self).__init__()
      self.discriminator = nn.Sequential(
          self.make_discriminator_block( 3, hidden_dim, first_layer=True),
          self.make_discriminator_block( hidden_dim, hidden_dim * 2),
          self.make_discriminator_block( hidden_dim * 2, hidden_dim * 4),
          self.make_discriminator_block( hidden_dim * 4, hidden_dim * 8),
          self.make_discriminator_block( hidden_dim * 8 , 1,final_layer=True),)

  def make_discriminator_block(self, input_channels, output_channels, kernel_size=4, stride=2, padding = 1,dilation=True,groups=1, bias = False, first_layer=False,
      if (not(final_layer or first_layer)):
          return nn.Sequential(
              nn.Conv2d(input_channels, output_channels, kernel_size, stride, padding,dilation, groups, bias),
              nn.BatchNorm2d(output_channels),
              nn.LeakyReLU(0.2, inplace=True))

      elif first_layer:
          return nn.Sequential(
              nn.Conv2d(input_channels, output_channels, kernel_size, stride, padding, dilation, groups, bias),
              nn.LeakyReLU(0.2, inplace=True))

      else:
          return nn.Sequential(
              nn.Conv2d(input_channels, output_channels,kernel_size,stride=1,padding =0,dilation=True, groups=1, bias=False)
              )

  def forward(self, x):

      out = self.discriminator(x)
          return out
print(Discriminator(64))
```



شکل معماری‌های مربوط به discriminator و generator

- **مقداردهی اولیه‌ی وزن‌ها (Weight Initialization):**

تمام وزن های مدل این پروژه به طور تصـــادفی از یک توزیع نرمال با میانگین صـــفر، واریانس ۰,۰۲ مقداردهی شـــده‌اند. تابع initial_weights_normal یک مدل اولیه را به عنوان ورودی در نظر می گیرد و تمام لایه های convolutional، convolutional-transpose و batch normalization layers را مقداردهی می‌کند تا معیارهای این پروژه را برآورده کند. در زیر تصویرکدهای مربوط به این تابع آورده شده است.

```
from torch.nn import init
def initial_weights_normal(network):
    classname = network.__class__.__name__
    if hasattr(network, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear') != -1):
        init.normal_(network.weight.data, 0.0, 0.02)

        if hasattr(network, 'bias') and network.bias is not None:
            init.constant_(network.bias.data, 0.0)

    elif classname.find('BatchNorm2d') != -1:
        init.normal_(network.weight.data, 1.0, 0.02)
        init.constant_(network.bias.data, 0.0)
```

تصویر تابع initial_weights_normal

- **تابع build_network**

در این تابع هم دو شـــئ از کلاس‌های discriminator و generator به نام‌های D و G می‌ســـازیم و مقدار دهی اولیه‌ی وزن‌ها را هم اعمال می‌کنیم تا در مراحل بعدی، کد آنها را مورد استفاده قرار بدهیم.

```
def build_network(d_conv_dim, g_conv_dim, z_size, verbose = False):
    D = Discriminator(d_conv_dim)
    G = Generator(z_dim=z_size, hidden_dim=g_conv_dim)
    D.apply(initial_weights_normal)
    G.apply(initial_weights_normal)
    if verbose:
        print(D)
        print()
        print(G)
    return D, G
```

تصویر تابع build_network

- **استفاده از GPU برای فرایند یادگیری**

همانطور که در تصویر قطعه کد زیر مشخص است فرایند یادگیری را بر روی GPU منتقل می‌کنیم.

```
import torch
train_on_gpu = torch.cuda.is_available()
if not train_on_gpu:
    print('No GPU found. Please use a GPU to train your neural network.')
else:
    print('Training on GPU!')
```

- **تابع هزینه و بهینه ساز (Loss Functions and Optimizers)** :

در این پروژه به جای discriminator و generator از D و G به عنوان شئ این کلاس‌ها آورده شده است. که بوسیله‌ی loss function و optimizer پیشرفت آنها در یادگیری را بررسی می‌کنیم.

در این پروژه از Binary Cross Entropy loss function یا(BCELoss) ا ستفاده شده ا ست.که درواقع از ترکیب تابع فعال ساز sigmoid و Cross Entropy Loss ت شکیل شده ا ست. باینری در ابتدای نام این تابع به دلیل این است که از این نوع تابع در مسائل classifier باینری استفاده می‌شود.

- $V(x;\omega)$ = discriminator (excluding final activation)
- $G(z;\theta)$ = generator
- $\sigma(v)$ = sigmoid activation function
- $\mathcal{L}_{BCE}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$ = binary cross entropy loss

Discriminator update, "real" examples $x \sim X_{data}$:

$$\mathcal{L}_{real}(x) = \mathcal{L}_{BCE}(\sigma(V(x;\omega)), 1) = -\log(\sigma(V(x;\omega)))$$

Discriminator update, "fake" examples $x \sim G(Z;\theta)$:

$$\mathcal{L}_{fake}(x) = \mathcal{L}_{BCE}(\sigma(V(x;\omega)), 0) = -\log(1-\sigma(V(x;\omega)))$$

Generator update:

$$\mathcal{L}_{gen}(z) = \mathcal{L}_{BCE}(\sigma(V(G(z;\theta);\omega)), 1) = -\log(\sigma(V(G(z;\theta);\omega)))$$

در نهایت لیبل دیتای واقعی را برابر ۱ و لیبل دیتای غیرواقعی را برابر صفر قرار دادیم. این لیبل ها به هنگام محاسبه‌ی تابع هزینه برای D وG استفاده می‌شوند. درنهایت دو بهینه ساز یا optimizer از نوع Adam با

نـرخ یـادگیری ۰,۰۰۰۲ و متغیـر $\beta 1$ برابـر ۰,۵ بـرای بهینـه سـازی تـابع هزینـه اسـتفاده کـردیم. در تصـویر زیـر کـدهای مربـوط بـه دو تـابعی کـه بـرای مجاسـبه‌ی Loss هـا مـورد اسـتفاده قـرار می‌گیرنـد و در زیـر توضـیحات خلاصه‌ی هرکدام آورده شده است.

- **real_loss** : ایـن تـابع احتمـال نزدیـک بـودن خروجـی discriminator را بـه توزیـع دیتـای واقعـی بررسـی می‌کنـد. در آن از BCELoss اسـتفاده شـده اسـت.در آرگومـان ایـن تـابع متغیـری بـه نـام smooth وجـود دارد کـه بـرای اعمـال ایـده‌ی **One-sided label smoothing** اسـت کـه در سـال ۲۰۱۶ در مقالـه‌ی NIPS 2016 Tutorial: Generative Adversarial Networks مطرح شد و از ایـن قـرار اسـت کـه بـه نظـر می‌رسـد محاسـبه‌ی loss دیتاهـای واقعـی بـا یـک بـردار هـم سایز با تعـداد دیتـای اصـلی حـاوی مقـادیر کمتـر از یـک مـثلا ۰,۹ مـوثر تـر عمـل می‌کنـد چـون از رفتـار تعمـیم افراطـی discriminator جلـوگیری می‌کنـد. ولـی loss دیتاهـای fake بـا بـرداری از یـک ها مقایسه و محاسبه می‌شود. (با فرمول‌هایی که در صفحه‌ی قبل آورده شده است.)
- **fake_ loss** : این تابع هم برای محا سبه‌ی احتمال نزدیک بودن به تابع fake ا ست. و درآن از BCELoss برای محاسبه‌ی loss استفاده شده است. موارد مربوط به پیاده سازی آن در تصاویر زیر آورده شده است.

```
def real_loss(D_out, smooth = False):

    _batch_size = D_out.size(0)
    # label smoothing
    if smooth:
        # smooth, real labels = 0.9
        labels = torch.ones(_batch_size)*0.9
    else:
        labels = torch.ones(_batch_size) # real labels = 1
    # move labels to GPU if available
    if train_on_gpu:
        labels = labels.cuda()
    # binary cross entropy with logits loss
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

```
def fake_loss(D_out):

    _batch_size = D_out.size(0)
    labels = torch.zeros(_batch_size) # fake labels = 0
    if train_on_gpu:
        labels = labels.cuda()
    criterion = nn.BCEWithLogitsLoss()
    # calculate loss
    loss = criterion(D_out.squeeze(), labels)
    return loss
```

- **تابع train**

تابعی اســت که کدهای مربوط به training ، discriminator و generator و محاســبه‌ی loss مربوط به هرکدام از شــبکه‌ها در هر Epoch را دربرگرفته اســت. (توضــیحات کامل و دقیق این بخش در ویدیوی مربوط به ارائه‌ی این پروژه آورده شده است.) تصاویر زیر کدهای مربوط به این تابع هستند.

```
def train(D, G, n_epochs,d_optimizer, g_optimizer, batch_size, z_size=100, img_size=64 ,save_samples=True):


    celeba_train_loader = fn_Dataloader(batch_size, img_size)

    # used for print_msg
    epoch_len = len(str(n_epochs))
    batches = len(celeba_train_loader)
    batches_len = len(str(batches))
    # print 10 times per epoch
    print_every = batches // 10

    if train_on_gpu:
        D.cuda()
        G.cuda()

    # keep track of loss and generated, "fake" samples
    samples = []
    losses = []
    avg_losses = []
    d_losses = []
    g_losses = []

    # Get some fixed data for sampling. These are images that are held
    # constant throughout training, and allow us to inspect the model's performance
    sample_size = 64
    fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size, 1, 1))
    fixed_z = torch.from_numpy(fixed_z).float()
    # move z to GPU if available
```

```
    if train_on_gpu:
        fixed_z = fixed_z.cuda()

    # epoch training loop
    for epoch in range(1, n_epochs + 1):

        # batch training loop
        for batch_i, (real_images, _) in enumerate(celeba_train_loader):

            _batch_size = real_images.size(0)
            real_images = make_scale(real_images)
```

در ســطر آخر میبینیم که پس از لود کردن دیتا آن را توســط تابع make_scale به بازه‌ی (1 ,1-) میبیریم تا با خروجی generator که هردو به عنوان ورودی به discriminator داده می‌شود هم scale باشند.

```
#*****************************************TRAIN THE NETWORKS***********************************


d_optimizer.zero_grad()

# 1. Train the discriminator on real and fake images

# Compute the discriminator losses on real images
if train_on_gpu:
    real_images = real_images.cuda()

D_real = D(real_images)
d_real_loss = real_loss(D_real, smooth=True)

# Generate fake images
z = np.random.uniform(-1, 1, size=(_batch_size, z_size, 1, 1))
z = torch.from_numpy(z).float()
# move x to GPU, if available
if train_on_gpu:
    z = z.cuda()
fake_images = G(z)

# Compute the discriminator losses on fake images
D_fake = D(fake_images)
d_fake_loss = fake_loss(D_fake)
```

```
# add up loss and perform backprop
d_loss = d_real_loss + d_fake_loss
d_loss.backward()
d_optimizer.step()

# 2. Train the generator with an adversarial loss

g_optimizer.zero_grad()

# Generate fake images
z = np.random.uniform(-1, 1, size=(_batch_size, z_size, 1, 1))
z = torch.from_numpy(z).float()
if train_on_gpu:
    z = z.cuda()
fake_images = G(z)

# Compute the discriminator losses on fake images
# using flipped labels!
D_fake = D(fake_images)
g_loss = real_loss(D_fake) # use real loss to flip labels

# Add first loss before training starts
if batch_i == 0 and epoch == 1:
    losses.append((d_loss.item(), g_loss.item()))

# perform backprop
g_loss.backward()
g_optimizer.step()
```

```
#*****************************************END OF training part*********************************
```

تصاویر بالا بخشی از تابع train و درواقع اون قسمتی است که کدهای مربوط به training ، discriminator و generator است. بقیه‌ی بخش‌ها مربوط به ذخیره‌سازی loss ها و موارد حساب شدهدر طی فرایند یادگیری است.

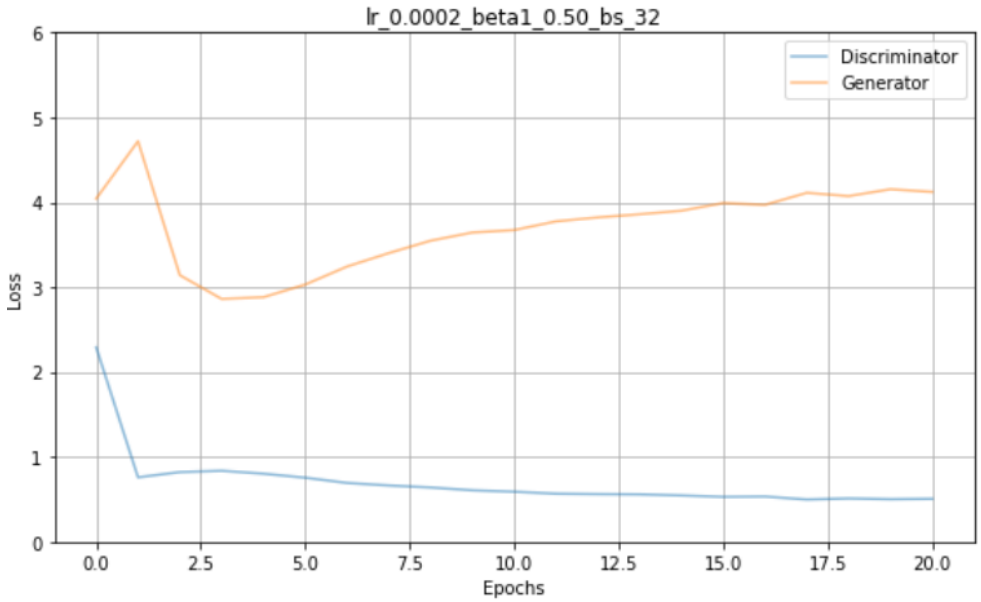
- **بخش پایانی تنظیمات پارامتر ها :**

مدل استفاده شده در این پروژه در ۲۰ Epoch آموزش دیده است. batch size در این پروژه ۳۲ است. نرخ یادگیری بر روی ۰,۰۰۰۲ و بتا یک بر روی ۰,۵ تنظیم شده ا ست. که در این مورد برا ساس مقاله ای که در سال ۲۰۱۵ به نام Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks انتشار پیدا کرد الگو گیری شده است. در تصویر زیر کد مربوط به تنظیم پارامتر آورده شده است.

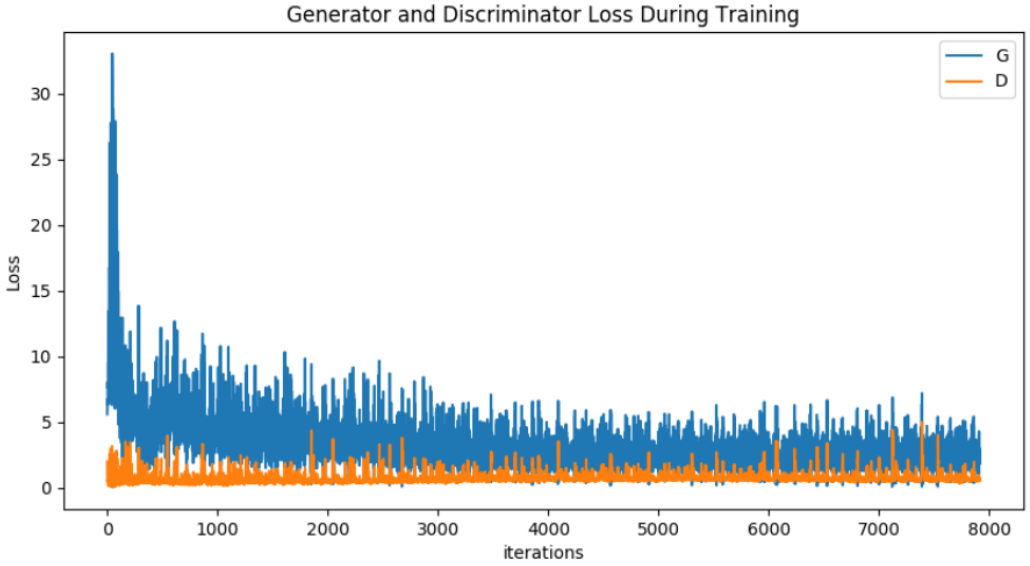
```
# Define hyperparameters
img_s = 64

d_hidden_dim = 64
g_hidden_dim = 64
z_s = 100

beta2 = 0.999 # default value
# set number of epochs
n_epochs = 20
# set batch size
batch_s = 32
# set learning rate
lr = 0.0002
# set beta1
beta1 = 0.5
```

- **نمودار مربوط به Loss ها :**

در ادامه‌ی کولب نوت بوک کدهای مربوط به visualize کردن نمودارهای مربوط به loss ها و ساختن گیف از تصاویر تولید شده در Epoch های مختلف داریم. که نتایج این کدها را به ترتیب در تصایر زیر آورده شده است.



توجه . به دلیل اینکه تعداد ایپاک‌ها پایین بوده منحنی‌های مربوط به D Loss و G Loss چندان همگرا نشده‌اند ولی در تعداد بالا ایپاک انتظار می‌رود به شکلی مانند شکل زیر دست بیابند.



شکل بالا نمونه‌هایی از sample هایی که generator این پروژه ساخته است. و در تصویر مقابل هم گیفی که از ایپاک‌های مختلف انجام این پروژه ساخته شده آورده شده است.