به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	فاطمه جلیلی - سالار صفردوست
شماره دانشجویی	۸۱۰۱۹۹۳۹۸ - ۸۱۰۱۹۹۴۵۰
تاریخ ارسال گزارش	۱۴۰۲/۱۱/۱۰

فهرست

پاسخ ۱. Control VAE

۱-۱. مقدمه

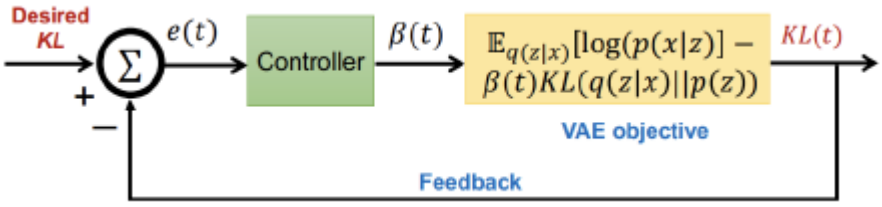
تابع هزینه‌ی مربوط به ساختار **VAE** دارای دو ترم **reconstruction** و **KL** می‌باشد. با کاهش دادن ترم اول هدف داریم تا میزان شباهت ورودی و خروجی را افزایش دهیم و با کاهش ترم دوم به دنبال آن هستیم تا اطلاعات مشترک ورودی و خروجی را کاهش دهیم.

اما مشکل این روش برای آموزش شبکه این است که معمولا کنترلی روی **KL** وجود ندارد و میزان شباهت خروجی‌ها به ورودی‌ها چندان در کنترل ما نیست.

مقاله‌ی معرفی شده سعی دارد با اضافه کردن ضریب بتای متغیر برای **KL** در تابع هزینه، سعی کند این مقدار را با الهام از سیستم‌های کنترلی(**PI**) در مقداری ثابت نگه دارد.

برای پیاده‌سازی این عملیات، تابعی تعریف شده است که مقدار هدف **KL** را می‌گیرد و با توجه به بتاهای سابق سعی می‌کند مقدار بتا در تابع هزینه را به گونه‌ای تعیین کند که در مرحله‌ی بعد به مقدار هدف نزدیک شویم.

سیستم کنترلی **PI** تعریف شده از یک سیستم حلقه بسته تشکیل شده است که هر سری مقدار **KL** خوانده شده را از مقدار **KL** کنونی کم می‌کند(**e**) و مقدار به دست آمده را به تابع ما می‌دهد تا مقدار جدید بتا را بگیرد.



حلقه‌ی کنترلی بسته برای کنترل مقدار KL

۱-۲. پیاده سازی VAE

```
[8]  class VAE(nn.Module):
       def __init__(self, dim_z):
         super(VAE, self).__init__()
         self.encoder_conv = nn.Sequential(
             nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.Conv2d(32, 32, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.Conv2d(64, 256, kernel_size=4),
             nn.ReLU()
         )
         self.FC1 = nn.Linear(256, 256)
         self.mean = nn.Linear(256, dim_z)
         self.logvar = nn.Linear(256, dim_z)
         self.FC2 = nn.Linear(dim_z, 256)
         self.decoder_conv = nn.Sequential(
             nn.ReLU(),
             nn.ConvTranspose2d(256, 64, kernel_size=4),
             nn.ReLU(),
             nn.ConvTranspose2d(64, 64, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.ConvTranspose2d(32, 32, kernel_size=4, stride=2, padding=1),
             nn.ReLU(),
             nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1)
         )
       def forward(self, x):
         x = self.encoder_conv(x)
         x = x.reshape(x.size(0), -1)
         x = self.FC1(x)
         means = self.mean(x)
         logvars = self.logvar(x)
         stds = torch.exp(0.5*logvars)
         z = stds*torch.randn_like(stds)+means
         z = self.FC2(z)
         z = z.reshape(z.size(0), -1, 1, 1)
         x = self.decoder_conv(z)
         return x, means, stds, z
```

کلاس تعریف شده‌ی VAE

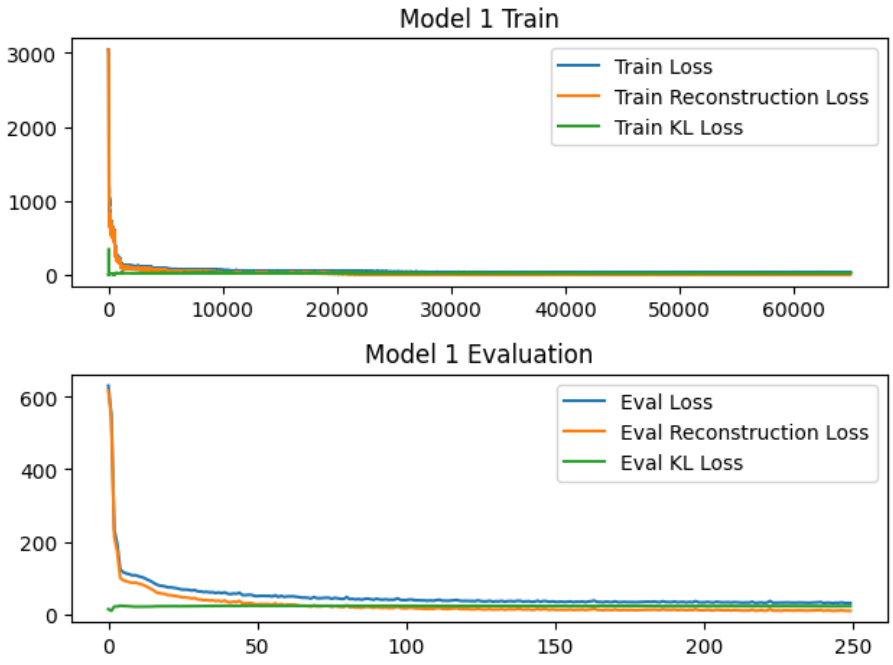
کلاس مربوط به **VAE** به شکل بالا تعریف شده است که در لایه‌ی مربوط به خروجی پنهان، دو خروجی ۱۰ بعدی برای میانگین‌ها و لگاریتم واریانس‌ها وجود دارد.

```python
device = torch.device('cuda' if torch.cuda.is_available else 'cpu')

dim_z = 10
model1 = VAE(dim_z).to(device)
train_imgs = train_imgs.to(device)
eval_imgs = eval_imgs.to(device)
test_imgs = test_imgs.to(device)

epochs = 250
batch_size = 64
learning_rate = 1e-4
train_data_loader = DataLoader(train_imgs, batch_size=batch_size, shuffle=True)
optimizer = optim.Adam(model1.parameters(), lr=learning_rate)

model1_train_reconstruction_loss_list = []
model1_train_KL_loss_list = []
model1_train_loss_list = []

model1_eval_reconstruction_loss_list = []
model1_eval_KL_loss_list = []
model1_eval_loss_list = []
for epoch in range(epochs):
  for i, in_imgs in enumerate(train_data_loader):
    out_imgs, means, stds, z = model1(in_imgs)

    reconstruction_loss = torch.sum(torch.mean(binary_cross_entropy_with_logits(out_imgs, in_imgs, reduction='none'), dim=0))
    KL_loss = torch.mean(0.5*torch.sum((stds.pow(2))+(means.pow(2))-2*stds.log()-1, dim=1))
    loss = reconstruction_loss+KL_loss
    model1_train_reconstruction_loss_list.append(reconstruction_loss.cpu().detach().numpy())
    model1_train_KL_loss_list.append(KL_loss.cpu().detach().numpy())
    model1_train_loss_list.append(loss.cpu().detach().numpy())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  print(f'Epoch {epoch+1}/{epochs}\t-> Train Loss: \t\t{loss:.2f}\t, Train Reconstruction Loss: \t\t{reconstruction_loss:.2f}\t, Train KL Loss: \t{KL_loss:.5f}')

  out_imgs, means, stds, z = model1(eval_imgs)
  reconstruction_loss = torch.sum(torch.mean(binary_cross_entropy_with_logits(out_imgs, eval_imgs, reduction='none'), dim=0))
  KL_loss = torch.mean(0.5*torch.sum((stds.pow(2))+(means.pow(2))-2*stds.log()-1, dim=1))
  loss = reconstruction_loss + KL_loss
  model1_eval_reconstruction_loss_list.append(reconstruction_loss.cpu().detach().numpy())
  model1_eval_KL_loss_list.append(KL_loss.cpu().detach().numpy())
  model1_eval_loss_list.append(loss.cpu().detach().numpy())

  print(f'\t\t   Evaluation Loss: \t{loss:.2f}\t, Evaluation Reconstruction Loss: \t{reconstruction_loss:.2f}\t, Evaluation KL Loss: \t{KL_loss:.5f}\n')
```

آموزش شبکه‌ی VAE با میانگیری از lossها روی هر batch

مقدار بتا برای **KL** برابر یک و ثابت در نظر گرفته شد.

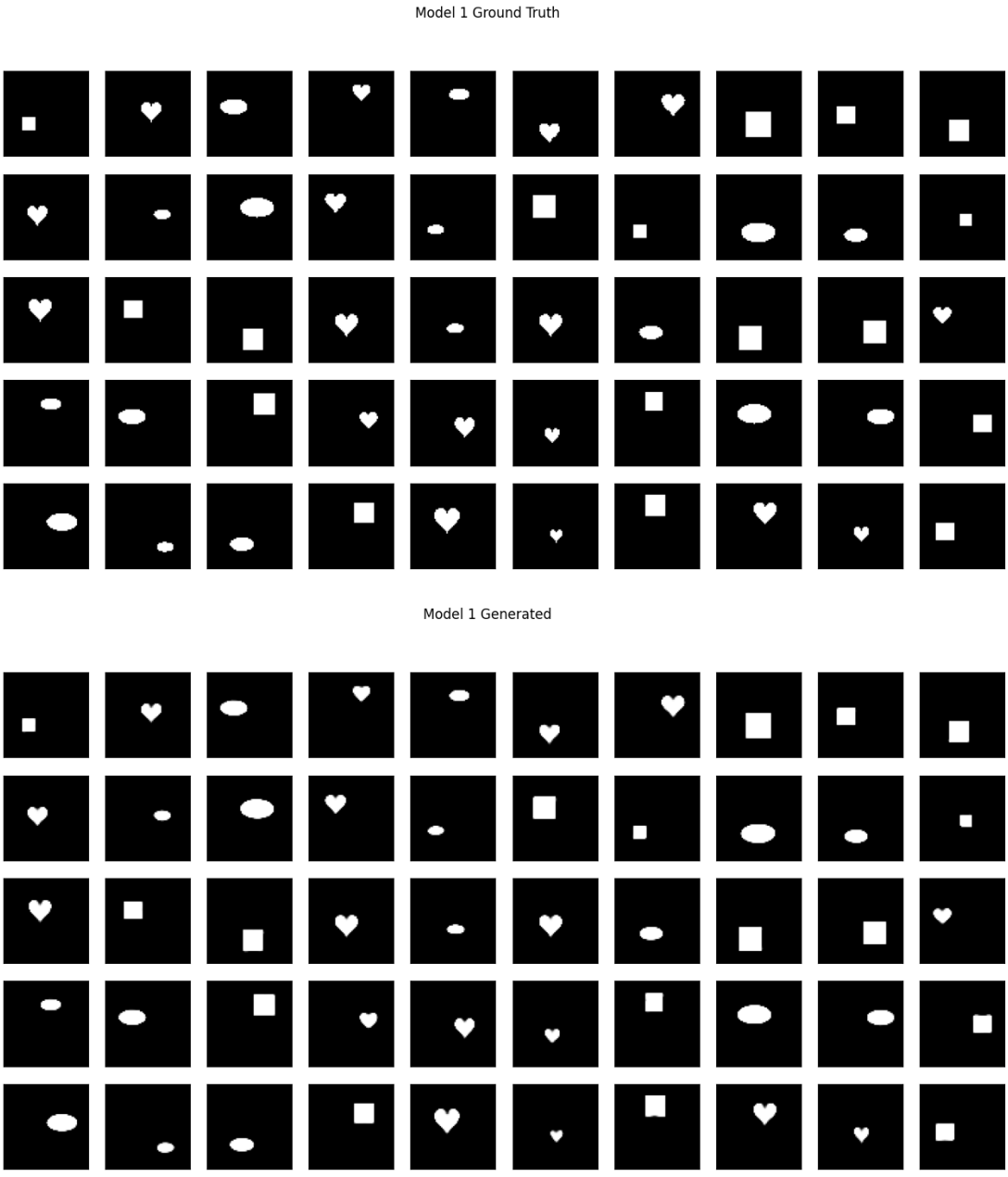
۱-۳. ارزیابی مدل VAE



نمودار مربوط به reconstruction loss، KL loss و total loss

همانگونه که مشخص است، هر سه مقدار نمایش داده شده روند نزولی داشته‌اند، البته مقدار **KL** در ابتدای کار کمی پرش داشته ولی این پرش کوتاه مدت بوده و در نهایت مقدار کمی برای این معیار ثبت شده است.

به طور کلی با توجه به اینکه هدف ما کم کردن تابع هدف بدون عوض کردن ضرایب در هر ایپاک بوده است، مقادیر مربوط به دو خطا کم و کمتر شده‌اند تا به مقدار نهایی برسند. البته پرش ناگهانی **KL** در ابتدای آموزش را می‌توان به خاطر آن دانست که مقدار اولیه‌ی آن بسیار کم بوده است و به همین خاطر گرادیان ایجاد شده تقریبا کامل به سمتی بوده که خطای **reconstruction** را کاهش دهد.



ورودی‌ها و خروجی‌های متناظر شبکه‌ی VAE پس از آموزش

در شکل‌های بالا کاملاً مشخص است که شباهت بالایی میان ورودی‌ها و خروجی‌ها وجود دارد، چرا که **KL Loss** بسیار کوچک شده است.

معیار **FID(Frechet Inception Distance)** برای به دست آوردن میزان کیفیت عکس‌های تولید شده توسط یک شبکه استفاده می‌شود. برای به دست آوردن این مقدار باید مجموعه‌ای از عکس‌های واقعی و عکس‌های تولیده شده را به آن بدهیم تا عددی مثبت را به عنوان خروجی به ما بدهد. این عدد هر چقدر به ۰ نزدیکتر باشد نشان‌گر نزدیکی بیشتر عکسای تولید شده به عکس‌های واقعی می‌باشد و به طور کلی مقادیر زیر ۵۰ مقادیر بسیار خوبی برای عکس‌های تولید شده توسط یک مدل می‌باشد.

این معیار توزیع‌های مربوط به عکس‌های تولیدی را با توزیع‌های مربوط به عکس‌های واقعی مقایسه می‌کند.

$$d_F(\mu, \nu) := \left(\inf_{\gamma \in \Gamma(\mu,\nu)} \int_{\mathbb{R}^n \times \mathbb{R}^n} \|x - y\|^2 \, \mathrm{d}\gamma(x, y) \right)^{1/2}$$

فرمول ریاضی

برای محاسبه معمولا دو دسته از تصویر را به یک شبکه مانند **Inception V3** می‌دهند و از روی خروجی‌ها مقادیر واریانس و میانگین را به دست می‌آورند. سپس از روی فرمول زیر مقدار **FID** را محاسبه می‌کنند. (در این فرمول مقادیر بدون پسوند مربوط به توزیع فیچرهای عکس‌های تولیدی است و مقادیر با پسوند مربوط به توزیع فیچرهای عکس‌های واقعی می‌باشد.

$$\mathrm{FID} = ||\mu - \mu_w||_2^2 + \mathrm{tr}(\Sigma + \Sigma_w - 2(\Sigma^{1/2}\Sigma_w\Sigma^{1/2})^{1/2}).$$

این معیار با فرض امکان توصیف کردن

```
Downloading: "https://github.com/mseitzer/pytorch-fid/releases/download/fid_weights/pt_inception-2015-12-05-6726825d.pth"
100%|██████████| 91.2M/91.2M [00:01<00:00, 72.6MB/s]
100%|██████████| 20/20 [00:18<00:00,  1.05it/s]
100%|██████████| 20/20 [00:19<00:00,  1.02it/s]
Model 1 Generated Images FID: 29.41410906522333
```

مقدار FID محاسبه شده برای ۵۰۰۰ عکس واقعی و تولید شده

۱-۴. پیاده سازی Control VAE

```
device = torch.device('cuda' if torch.cuda.is_available else 'cpu')

dim_z = 10
model3 = VAE(dim_z).to(device)
train_imgs = train_imgs.to(device)
eval_imgs = eval_imgs.to(device)
test_imgs = test_imgs.to(device)

epochs = 250
batch_size = 64
learning_rate = 1e-4
train_data_loader = DataLoader(train_imgs, batch_size=batch_size, shuffle=True)
optimizer = optim.Adam(model3.parameters(), lr=learning_rate)

model3_train_reconstruction_loss_list = []
model3_train_KL_loss_list = []
model3_train_loss_list = []

model3_eval_reconstruction_loss_list = []
model3_eval_KL_loss_list = []
model3_eval_loss_list = []

I = 0
beta = 0
beta_min = 1
beta_max = 50
Kp = 0.01
Ki = 0.001
desired_KL = 14
beta_list = []
for epoch in range(epochs):
  for i, in_imgs in enumerate(train_data_loader):
    out_imgs, means, stds, z = model3(in_imgs)

    reconstruction_loss = torch.sum(torch.mean(binary_cross_entropy_with_logits(out_imgs, in_imgs, reduction='none'), dim=0))
    KL_loss = torch.mean(0.5*torch.sum((stds.pow(2)+(means.pow(2))-2*stds.log()-1, dim=1))
    beta, I = Beta_Calculator(desired_KL, KL_loss.cpu().detach().numpy(), Kp, Ki, beta, I, beta_max, beta_min)
    loss = reconstruction_loss+beta*KL_loss
    beta_list.append(beta)
    model3_train_reconstruction_loss_list.append(reconstruction_loss.cpu().detach().numpy())
    model3_train_KL_loss_list.append(KL_loss.cpu().detach().numpy())
    model3_train_loss_list.append(loss.cpu().detach().numpy())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  print(f'Epoch {epoch+1}/{epochs}\t-> Train Loss: \t\t{loss:.2f}\t, Train Reconstruction Loss: \t\t{reconstruction_loss:.2f}\t, Train KL Loss: \t{KL_loss:.5f}')

  out_imgs, means, stds, z = model3(eval_imgs)
  reconstruction_loss = torch.sum(torch.mean(binary_cross_entropy_with_logits(out_imgs, eval_imgs, reduction='none'), dim=0))
  KL_loss = torch.mean(0.5*torch.sum((stds.pow(2)+(means.pow(2))-2*stds.log()-1, dim=1))
  loss = reconstruction_loss + KL_loss
  model3_eval_reconstruction_loss_list.append(reconstruction_loss.cpu().detach().numpy())
  model3_eval_KL_loss_list.append(KL_loss.cpu().detach().numpy())
  model3_eval_loss_list.append(loss.cpu().detach().numpy())
```

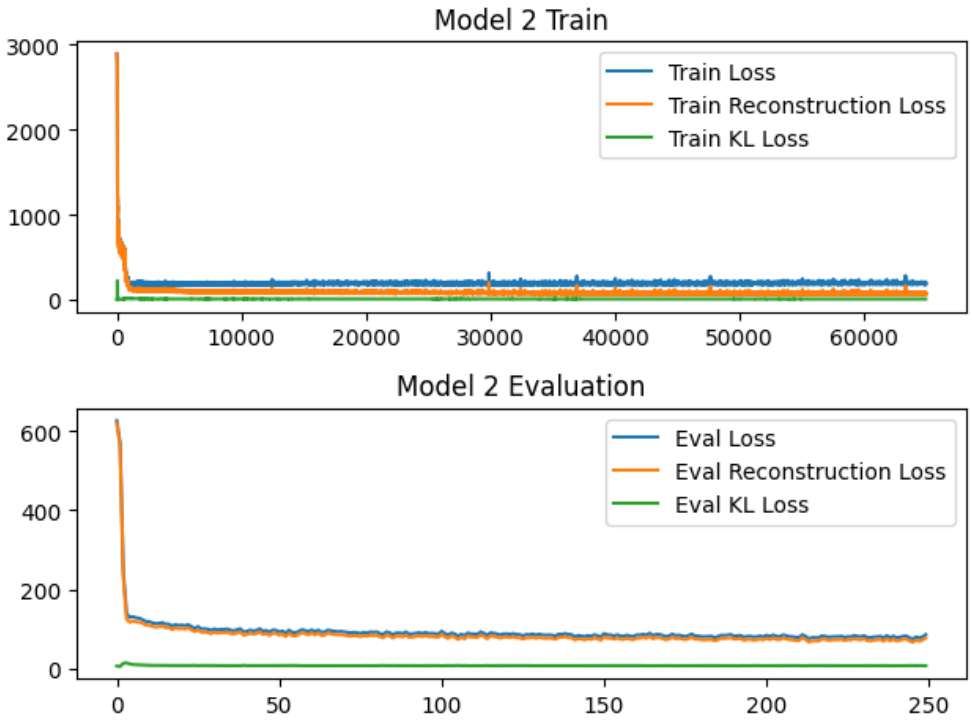
پیاده‌سازی CVAE و آموزش آن با پارامترهای مد نظر در صورت سوال

در این مرحله، الگوریتم مدنظر در مقاله پیاده‌سازی شد و در هر ایتریشن با فراخوانی تابع مرتبط با الگوریتم مقاله به نام **Beta_Calculator** مقدار جدید بتا به دست می‌آید.

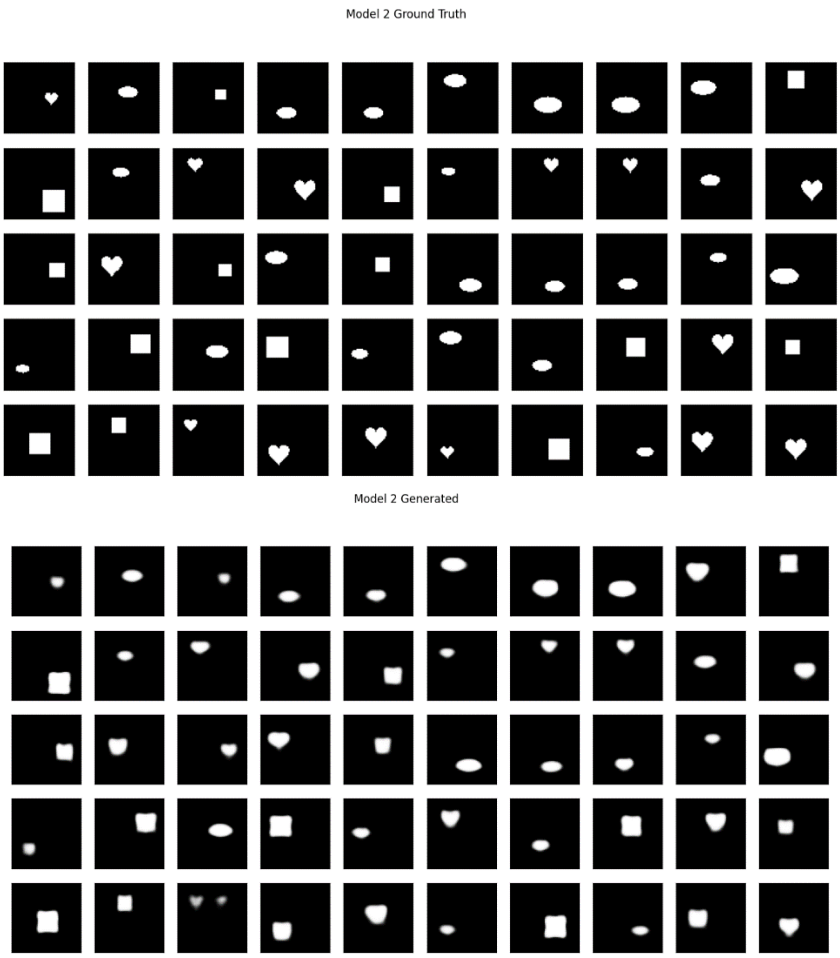
```
def Beta_Calculator(deisred_KL, KL_loss, Kp, Ki, beta, I, beta_max, beta_min):
  e = deisred_KL-KL_loss
  P = Kp/(1+np.exp(e))
  if(beta>=beta_min and beta<=beta_max):
    I_new = I-Ki*e
  else:
    I_new = I
  beta_new = P+I_new+beta_min
  if(beta_new>beta_max):
    beta_new = beta_max
  elif(beta_new<beta_min):
    beta_new=beta_min
  return beta_new, I_new
```

تابع Beta_Calculater

الگوریتم به این صورت است که در هر تکرار مقدار **I** که به گونه‌ای منفی انتگرال اورها می‌باشد به همراه بتای حال حاضر و سایر ضرایب به تابع داده می‌شوند تا با پیاده سازی **PI** غیرخطی روی آن‌ها مقدار جدید **I** و بتا محاسبه شود. مقادیر بتای ماکسیمم و مینیمم برای آن هستند تا از تغییرات ناگهانی در بتا جلوگیری کنیم و پایداری را بیشتر کنیم.



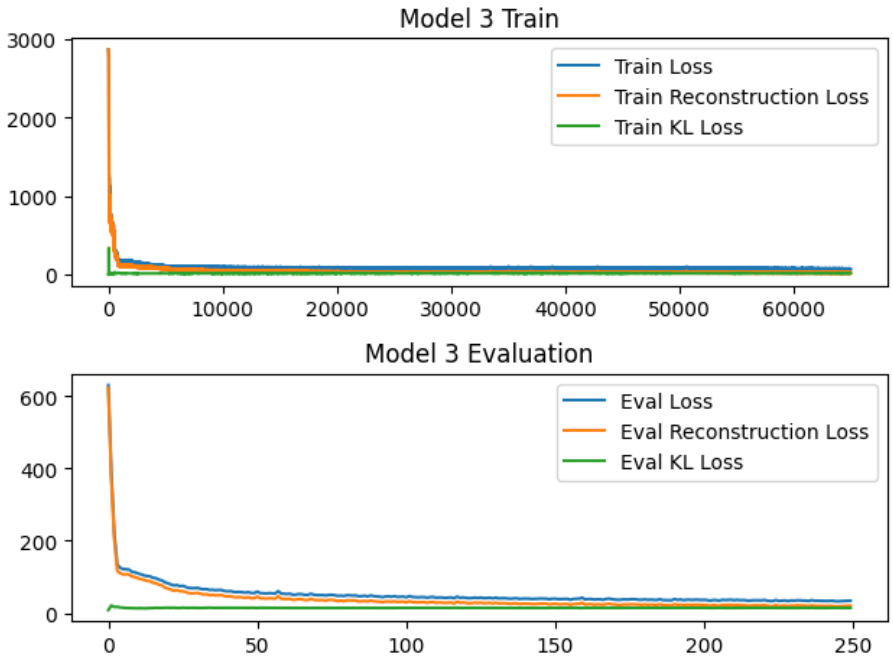
توابع هزینه برای desired_KL=8



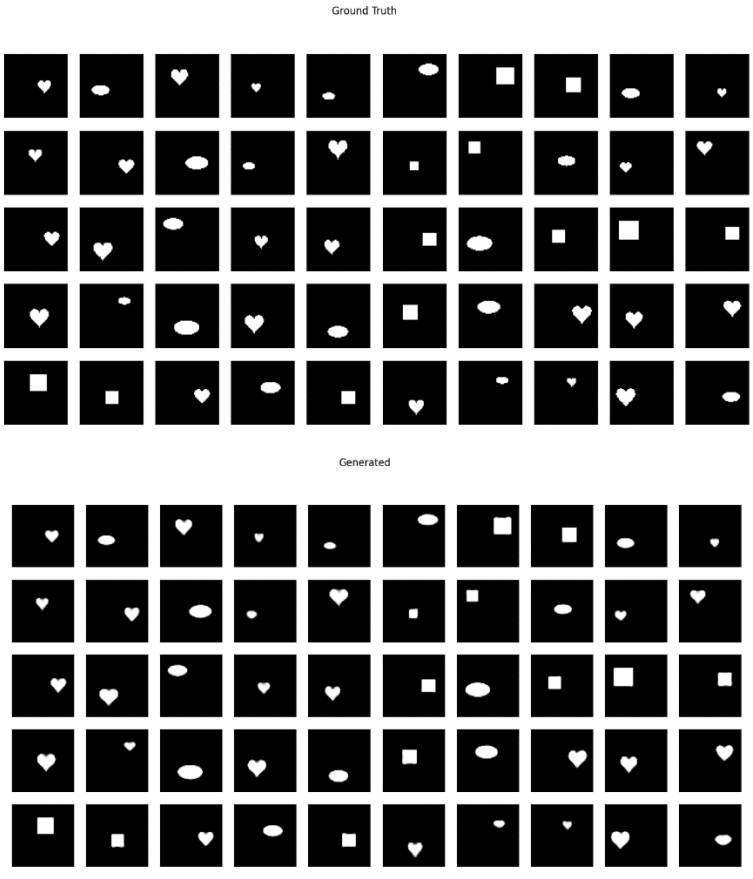
خروجی‌های شبکه به ازای desired_KL=8

```
100%|██████████| 20/20 [00:19<00:00,  1.05it/s]
100%|██████████| 20/20 [00:19<00:00,  1.02it/s]
Model 2 Generated Images FID: 117.71792515739791
```

معیار FID برای desired_KL=8



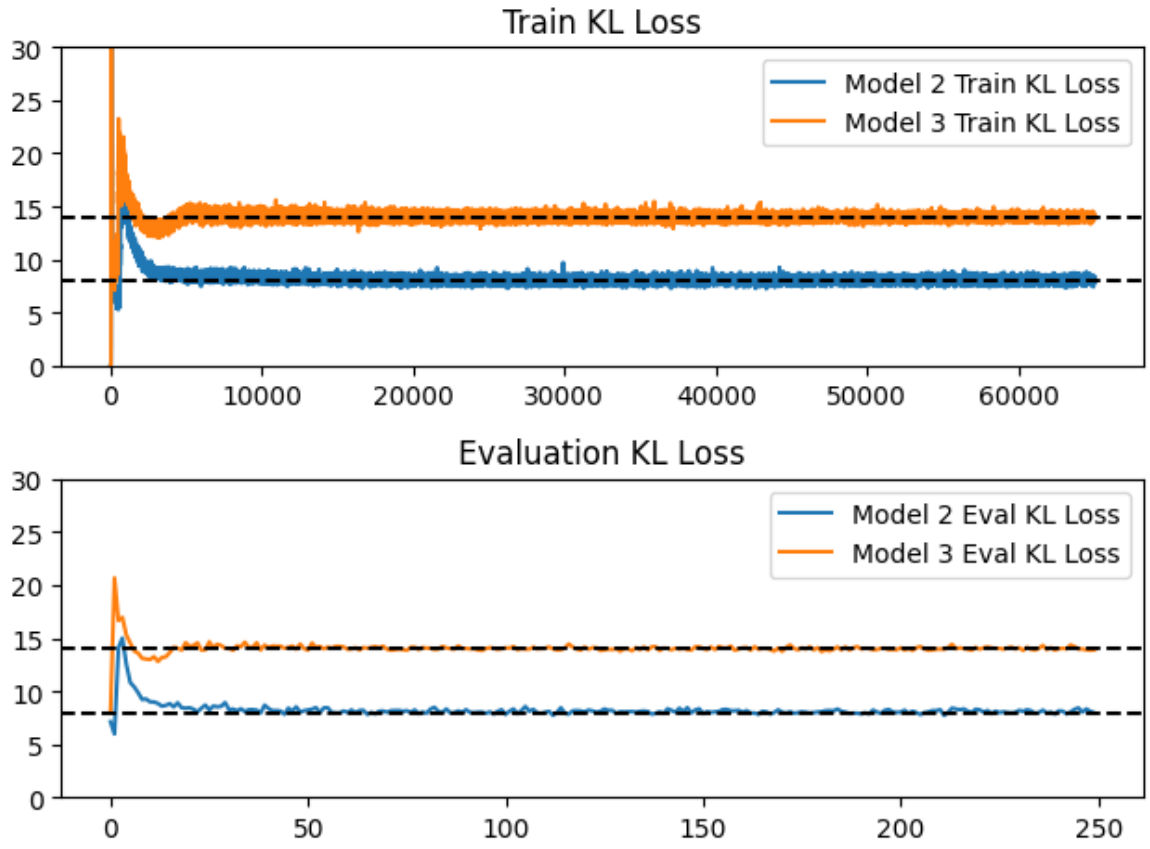
توابع هزینه برای desired_KL=14



خروجی‌های شبکه به ازای desired_KL=14

```
100%|██████████| 20/20 [00:18<00:00,  1.05it/s]
100%|██████████| 20/20 [00:19<00:00,  1.02it/s]
Model 3 Generated Images FID: 66.57572985468784
```

معیار FID برای desired_KL=14



نمایش مقادیر KL برای هر کدام از دو مدل

در خروجی‌های مدل دیده می‌شود با کاهش **KL** به کاهش شباهت میان عکس‌های واقعی و تولید شده می‌رسیم و همچنین معیار **FID** را افزایش می‌دهیم.

همچنین قابل مشاهده است که در هر دو حالت به خوبی توانسته‌ایم معیار **KL** در مقدار مد نظر کنترل کنیم.

پاسخ ۲. معرفی Generative Adversarial Networks(GANs)

۲-۱. آموزش مدل GAN بر روی دیتاست MNIST

مشابه معماری های داده شده با تکمیل قسمت های علامت سوال مطابق عکس که در ادامه آمده است

```
class discriminatorNet(nn.Module):
    def __init__(self, d_input_dim = 1):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(d_input_dim, 32, kernel_size = 4, stride = 2, padding = 1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size = 4, stride = 2, padding = 1),
            nn.ReLU(),
            reshape(64 * 7 * 7),
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 1),
        )
    def forward(self, d):
        return self.model(d)

class generatorNet(nn.Module):
    def __init__(self, g_input_dim = 64, g_output_dim = 1):
        super().__init__()
        self.g_input_dim = g_input_dim
        self.model = nn.Sequential(
            nn.Linear(g_input_dim, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Linear(512, 64 * 7 * 7),
            nn.BatchNorm1d(64 * 7 * 7),
            nn.ReLU(),
            reshape(64, 7, 7),
            nn.PixelShuffle(2),
            nn.Conv2d(16, 32, kernel_size = 3, padding =  1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.PixelShuffle(2),
            nn.Conv2d(8, g_output_dim, kernel_size = 3, padding = 1),
            nn.Tanh()
        )
    def forward(self, g):
        return self.model(g)
```

شبکه های **generator** و **discriminator** را تعریف می کنیم :

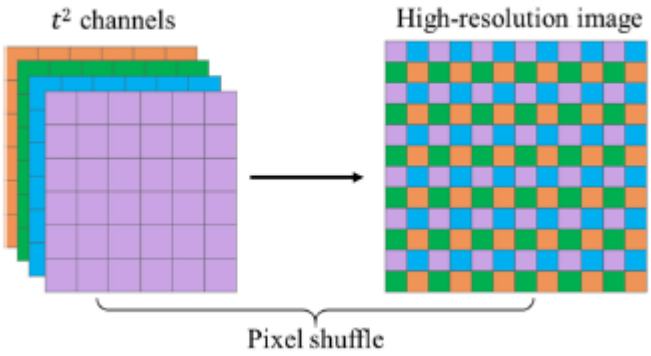
برای مطابقت خروجی بین لایه ها یک لایه **reshape** هم قبل **PixelShuffle** در **generator** و یک لایه قبل **Linear** در **discriminator** اضافه کردیم.

همچنین همانطور که دیده می شود مقادیر علامت سوال اول $64 \times 7 \times 7$ تعیین شد تا باکس های ۷ در ۷ با عمق ۶۴ بسازیم تا زمانی که ۲ بار از pixelshuffle با r=2 استفاده می کنیم همان سایز خروجی ۲۸ در ۲۸ بشود که سایز عکس های دیتاست ما است. مقدار علامت سوال دوم ۱ برابر خروجی generator و مقدار علامت سوال سوم هم مشابه علامت سوال اول $64 \times 7 \times 7$ تعیین شد.

۲-۱-۱. پیاده سازی

پرسش ۱

عملگر PixelShuffle یک تنسور چند کاناله را با باز چینش پیکسل های کانال های آن تبدیل به یک تنسور با تعداد کانال های کمتر اما با ابعاد بزرگ تر می کند، به بیان دقیق تر این عملگر ، درایه های یک تنسور با ابعاد $(*, C \times r^2, H, W)$ را به فرم $(*, C, H \times r, W \times r)$ تبدیل می کند



این عملگر در کاربرد ساخت یک مدل با وظیفه Super-Resolution ارائه شد و عملکرد آن برای شبکه معرفی شده در مقاله ای به نام Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network بهینه کردن فرآیند Convolution می باشد. همانطور که در ساختار شبکه پیشنهادی صورت پروژه دیده می شود ، از این عملگر درست قبل از لایه های Convolution استفاده شده است تا تعداد کانال ها را کاهش داده و به نوعی افزایش تعداد کانال های ناشی از لایه Convolution را جبران نماید. بدین ترتیب می توان به رزولوشن های بالاتر و عکس های با کیفیت تری از نظر وضوح لبه ها و جزییات تصویر رسید.

تابع خطای مورد نیاز را با استفاده از تابع binary_cross_entropy_with_logits مطابق زیر پیاده سازی می کنیم:

```
def discriminator_loss(logits_real, logits_fake):
    loss_pos = bce_loss(logits_real, torch.ones_like(logits_real))
    loss_neg = bce_loss(logits_fake, torch.zeros_like(logits_fake))
    loss = loss_pos + loss_neg
    return loss

def generator_loss(logits_fake):
    loss = bce_loss(logits_fake, torch.ones_like(logits_fake))
    return loss
```

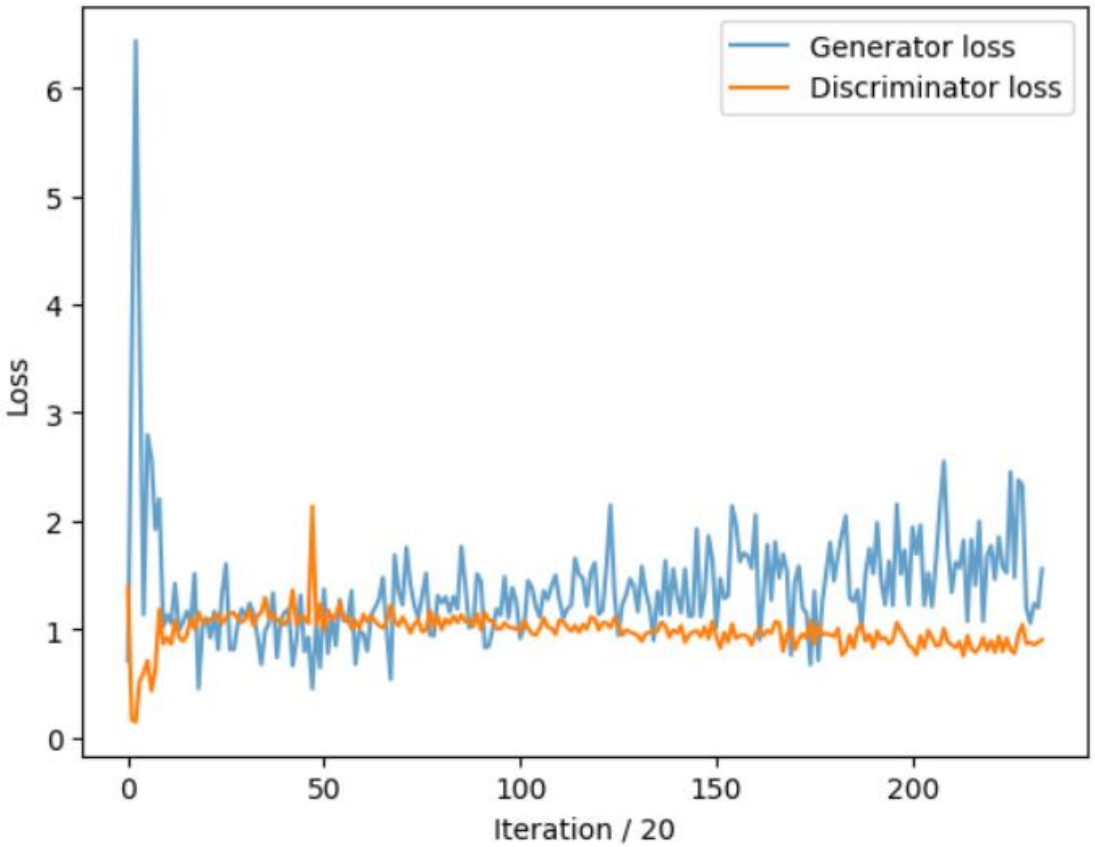
پرسش ۲

با استفاده از سعی و خطا برای ۱۰ ایپاک با batch_size=128 و مقادیر ابر پارامتر های مربوط به optimizer Adam مطابق زیر مدل را آموزش دادیم:

```
def get_optimizer(model):
    optimizer = optim.Adam(
        model.parameters(),
        lr=1e-3,
        betas=(0.5, 0.999)
    )
    return optimizer
```

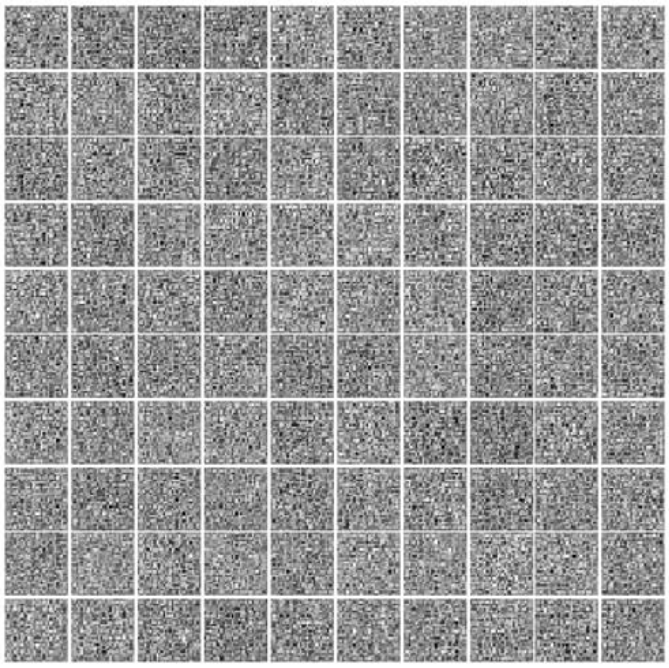
ارزیابی مدل

نمودار لاس ها :



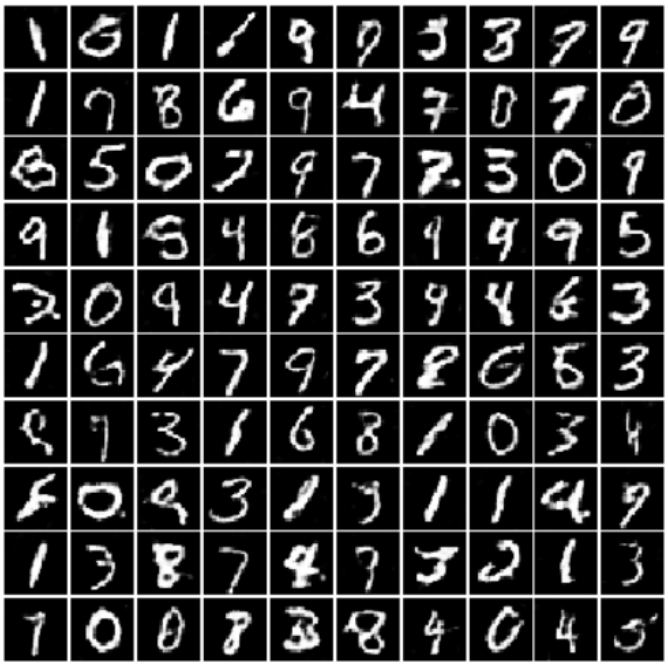
خروجی مدل به ازای **iteration =0** که در واقع صرفا همان نویز اولیه تولید شده می باشد:

```
iteration: 0
```



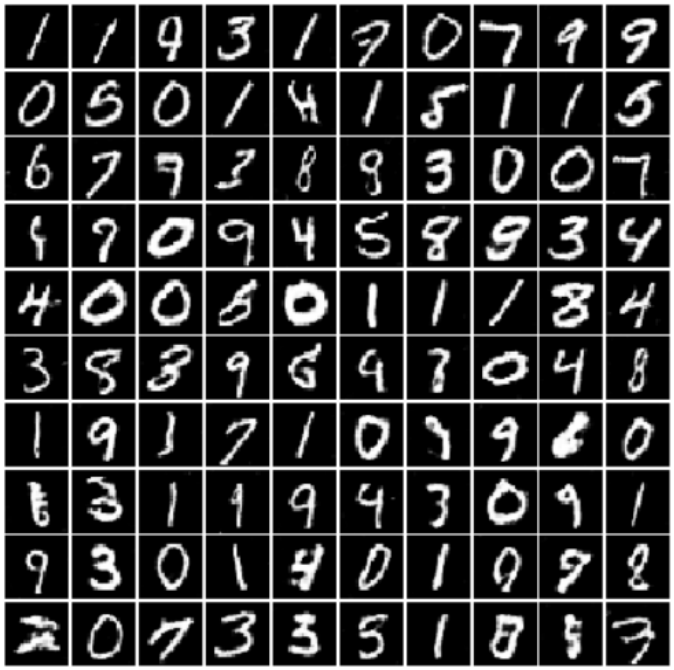
خروجی مدل به ازای مراحل میانی :

iteration: 1404



خروجی مدل به ازای مرحله ی نهایی:

iteration: 4212



با سیو کردن خروجی مدل و با استفاده از عکس های تست معیار **FID** را با استفاده از تابع **calculate_fid_given_paths** معیار **FID** را محاسبه می کنیم:

```
100%|██████████| 200/200 [00:38<00:00,  5.23it/s]
100%|██████████| 100/100 [00:19<00:00,  5.08it/s]
FID Score: 15.0679
```

همانطور که دیده می شود نتیجه حاصل شده بسیار مطلوب است منتها نمودار **discriminator loss** جهش های زیادی دارد که در مدل هایی که در ادامه پیاده سازی می کنیم از مشکل بهتر می شود ، به دلیل ساده بود دیتاست این مدل **GAN** اولیه به خوبی روی آن عمل می کند و نسبت به مدل های بعدی سریع تر به نتیجه هم می رسد منتها برای تولید عکس هایی با جزییات بیش تر به دنبال این هستیم که **discriminator loss** پایدار باشد.

۲-۲. مدل Wasserstein GAN

الف)

WGAN با استفاده از تابع هزینه **Wasserstein** و روش نرمال‌سازی وزن‌ها، مشکلات ناپدید شدن گرادیان و عدم استقرار در آموزش **GAN** را حل می‌کند و بهبودهای قابل توجهی در کیفیت تولید تصاویر و پایداری آموزش مدل‌ها ارائه می‌دهد.

مشکل **Gradient Vanishing** به این معنی است که در طول آموزش، ناحیه‌هایی از فضای نمونه‌ها وجود دارند که گرادیان‌ها به شدت کوچک یا صفر می‌شوند، که باعث کندی و عدم استقرار در فرآیند آموزش می‌شود. **WGAN** از معیار فاصله **Wasserstein** برای محاسبه فاصله بین توزیع واقعی داده‌ها و توزیع مصنوعی مدل استفاده می‌کند. این معیار فاصله برای تخمین دقیق‌تر و پایدارتر کردن آموزش **GAN** مورد استفاده قرار می‌گیرد. برای این منظور، **WGAN** از تابع هزینه **Wasserstein** استفاده می‌کند که علاوه بر برطرف کردن مشکل ناپدید شدن گرادیان، مزایای دیگری نیز دارد.

تابع هزینه **Wasserstein**، به جای استفاده از تابع هزینه ساده‌تری مانند تابع هزینه **Jensen-Shannon** یا دیورژانس **Kullback-Leibler**، از یک تابع هزینه متقارن و پیوسته استفاده می‌کند. این تابع هزینه باعث می‌شود که گرادیان‌ها در هر نقطه از فضا به طور متوسط مناسب باشند و در نتیجه، مشکل گرادیان منتقلی کاهش می‌یابد.

علاوه بر این، **WGAN** از روش نرمال‌سازی وزن‌ها **Weight Clipping** برای محدود کردن مقادیر وزن‌ها استفاده می‌کند. این روش به طور مستقیم باعث کاهش متغیریت در توزیع وزن‌ها می‌شود و به عنوان یک روش ساده برای تضمین پایداری آموزش **GAN** مورد استفاده قرار می‌گیرد.

ب)

معماری شبکه های **generator** و **discriminator** تغییری نمی کند منتها تغییرات دیگری در آموزش و لاس ها می دهیم که در ادامه اشاره می شود:

مطابق مقاله لاس ها را به صورت زیر تعریف می کنیم:

```
def critic_loss(logits_real, logits_fake):
  return -torch.mean(logits_real) + torch.mean(logits_fake)

def generator_loss(logits_fake):
  return -torch.mean(logits_fake)
```

همچنین **optimizer** خود را به **RMSProp** مطابق با پارامترهای مشابه شکل زیر تغییر می دهیم(این **optimizer** نشان داد بهتر روی **WGAN** عمل می کند و کانورج می شود)

```
def get_optimizer(model):
  return torch.optim.RMSprop(model.parameters(), lr = 1e-4)
```

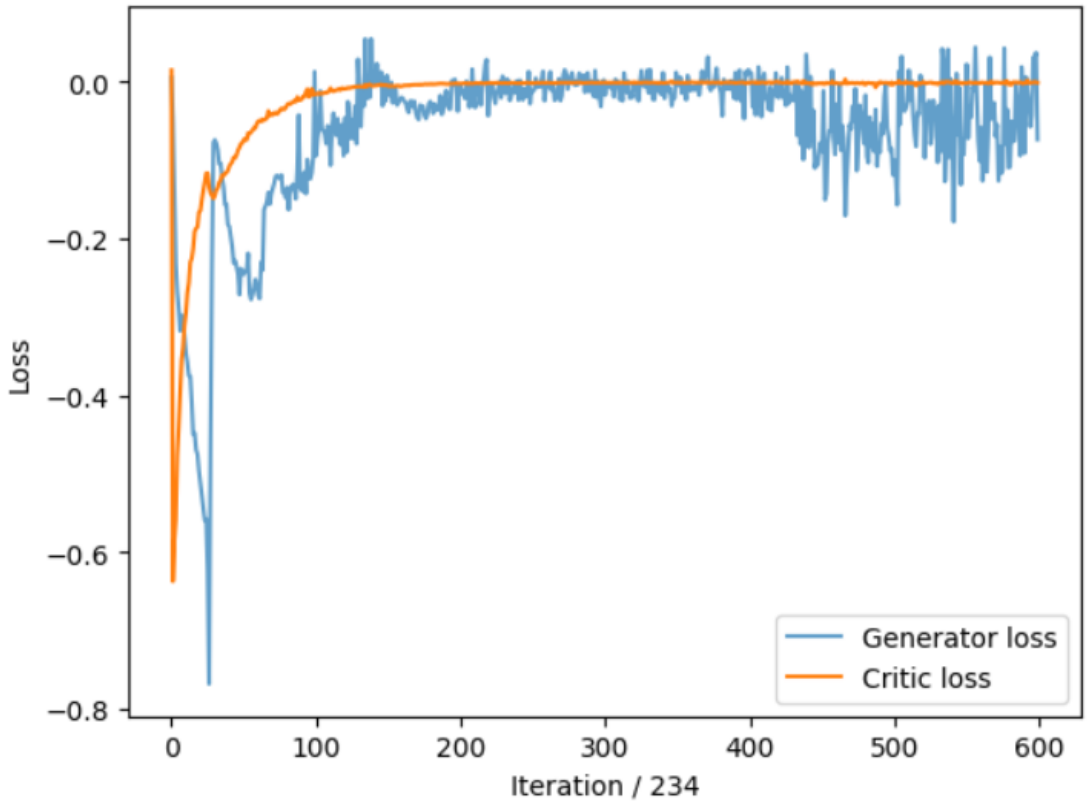
همچنین هر ۲۰ بار که **Critic** آپدیت می شود یک بار **Generator** را آپدیت می کنیم هم مدل سریع تر ترین می شود و هم بهتر کانورج می شود.

علاوه بر این از **Weight Clipping** روی وزن ها **Critic** استفاده می کنیم:

```
for p in C.parameters():
        p.data.clamp_(-0.005, 0.005)
```

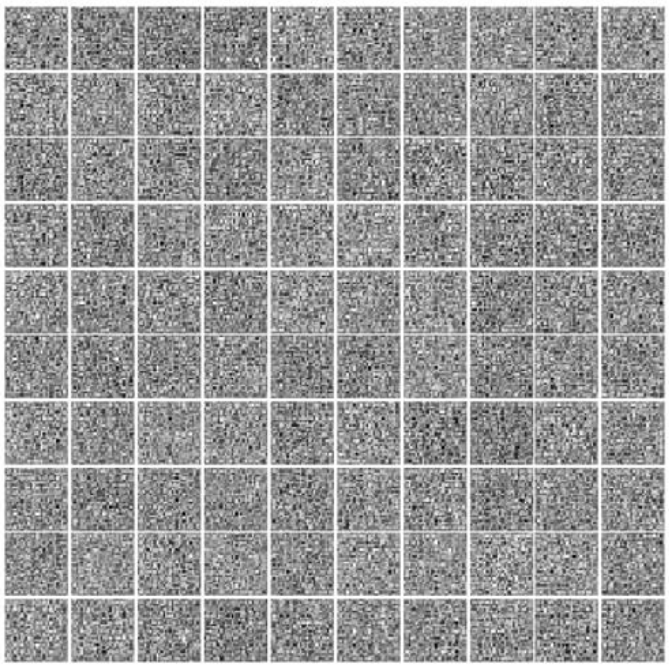
ارزیابی مدل :

نمودار لاس ها پس از ۳۰۰ ایپاک :



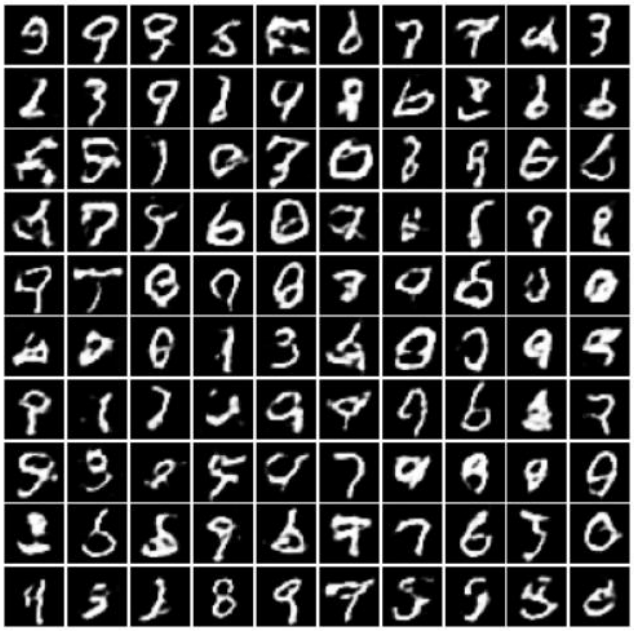
خروجی مدل به ازای **iteration =0** که در واقع صرفا همان نویز اولیه تولید شده می باشد:

```
iteration: 0
```



خروجی مدل به ازای مراحل میانی :

iteration: 84240



خروجی مدل به ازای مرحله ی نهایی:

iteration: 135720

با سیو کردن خروجی مدل و با استفاده از عکس های تست معیار **FID** را با استفاده از تابع **calculate_fid_given_paths** معیار **FID** را محاسبه می کنیم:

```
100%|██████████| 200/200 [00:40<00:00,  4.92it/s]
100%|██████████| 100/100 [00:19<00:00,  5.03it/s]
FID Score: 35.3114
```

همانطور که دیده می شود نتیجه حاصل شده همچنان مطلوب است همچنین نمودار **discriminator loss** بسیار منظم تر شده است و پایدار است، همچنین مطابق شکل با نمودار **generator loss** نسبت به مدل قبلی بهتر کانورج می شوند اما متاسفانه آموزش این مدل در مقایسه با حالت قبل زمان بسیار بیش تر و ایپاک های زیاد تری لازم دارد تا به خروجی مطلوب برسد ، اگر زمان کافی و زیرساخت های پردازشی بیش تر در اختیار باشد این مدل عملکرد بهتری از مدل اولیه نشان می دهد منتها به دلیل محدودیت پردازشی **colab** ما نتوانستیم بیش تر از ۳۰۰ ایپاک مدل را آموزش بدهیم.

۲-۲. مدل Self-Supervised GAN

۲-۳-۱. Generator

شبکه generator را مشابه معماری داده شده مطابق شکل زیر می سازیم:

```
class G_Residual(nn.Module):
  def __init__(self, in_channels, out_channels=256):
    super(G_Residual, self).__init__()
    self.Block1 = nn.Sequential(
        nn.BatchNorm2d(in_channels),
        nn.ReLU(),
        nn.Upsample(scale_factor=2, mode='nearest'),
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
    )
    self.Block2 = nn.Upsample(scale_factor=2, mode='nearest')
  def forward(self, x):
    block1 = self.Block1(x)
    block2 = self.Block2(x)
    return block1 + block2

class Generator(nn.Module):
  def __init__(self):
    super(Generator, self).__init__()
    self.model = nn.Sequential(
        nn.Linear(128, 256*4*4),
        reshape(256, 4, 4),
        nn.BatchNorm2d(256),
        G_Residual(256, 256),
        G_Residual(256, 256),
        G_Residual(256, 256),
        nn.ReLU(),
        nn.Conv2d(256, 1, kernel_size=3, stride=1, padding=1),
        nn.Tanh()
    )
  def forward(self, x):
    return self.model(x)
```

۲-۳-۲. Discriminator

شبکه discriminator هم مطابق معماری داده شده مشابه شکل زیر می سازیم:

```
class D_Residual(nn.Module):
  def __init__(self, in_channels, out_channels, start=False, down_sampling=False):
    super(D_Residual, self).__init__()
    self.start = start
    self.down_sampling = down_sampling

    self.relu = nn.ReLU()
    self.Block1 = nn.Sequential(
        nn.utils.spectral_norm(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)),
        nn.ReLU(),
        nn.utils.spectral_norm(nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)),
    )
    self.Block2 = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0)
    )
    self.avgpool2d = nn.AvgPool2d(kernel_size=2, stride=2, padding=1)
  def forward(self, x):
    if(self.start):
      block1 = self.relu(self.Block1(x))
    else:
      block1 = self.Block1(self.relu(x))
    if(self.down_sampling):
      block1 = self.avgpool2d(block1)
      x = self.avgpool2d(x)
    block2 = self.Block2(x)
    return block1 + block2

class Discriminator(nn.Module):
  def __init__(self):
    super(Discriminator, self).__init__()
    self.model = nn.Sequential(
        D_Residual(1, 128, start=True, down_sampling=True),
        D_Residual(128, 128, down_sampling=True),
        D_Residual(128, 128),
        D_Residual(128, 128),
        nn.ReLU()
    )
    self.FC1 = nn.Linear(128, 1)
    self.FC2 = nn.Linear(128, 4)
  def forward(self, x):
    x = self.model(x)
    x = torch.sum(x, dim=(2,3))
    realfake_logits = self.FC1(x)
    rotation_logits = self.FC2(x)
    return realfake_logits, rotation_logits
```

توابع لاس را با استفاده از cross entropy (چون ۴ کلاس برای چرخش داریم) مطابق شکل زیر تعریف می کنیم:

```python
def discriminator_realfake_loss(logits_real, logits_fake):
  loss_pos = bce_loss(logits_real, torch.ones_like(logits_real))
  loss_neg = bce_loss(logits_fake, torch.zeros_like(logits_fake))
  loss = loss_pos + loss_neg
  return loss

def generator_realfake_loss(logits_fake):
  loss = bce_loss(logits_fake, torch.ones_like(logits_fake))
  return loss

def discriminator_rotation_loss(logits_real, logits_fake, batch_size, device):
  ce_loss = nn.CrossEntropyLoss()
  labels = torch.zeros(logits_real.shape[0])
  for i in range(batch_size):
    labels[i*4] = 0
    labels[i*4+1] = 1
    labels[i*4+2] = 2
    labels[i*4+3] = 3
  labels = labels.to(torch.long).to(device)
  loss = ce_loss(logits_real, labels) + ce_loss(logits_fake, labels)
  return loss

def generator_rotation_loss(logits_fake, batch_size, device):
  ce_loss = nn.CrossEntropyLoss()
  labels = torch.zeros(logits_fake.shape[0])
  for i in range(batch_size):
    labels[i*4] = 0
    labels[i*4+1] = 1
    labels[i*4+2] = 2
    labels[i*4+3] = 3
  labels = labels.to(torch.long).to(device)
  loss = ce_loss(logits_fake, labels)
  return loss
```

دوباره از optimizer Adam با پارامتر هایی که در شکل زیر آمده است استفاده می کنیم:

```python
def get_optimizer(model):
  optimizer = optim.Adam(
      model.parameters(),
      lr=6e-6,
      betas=(0.9, 0.99)
  )
  return optimizer
```

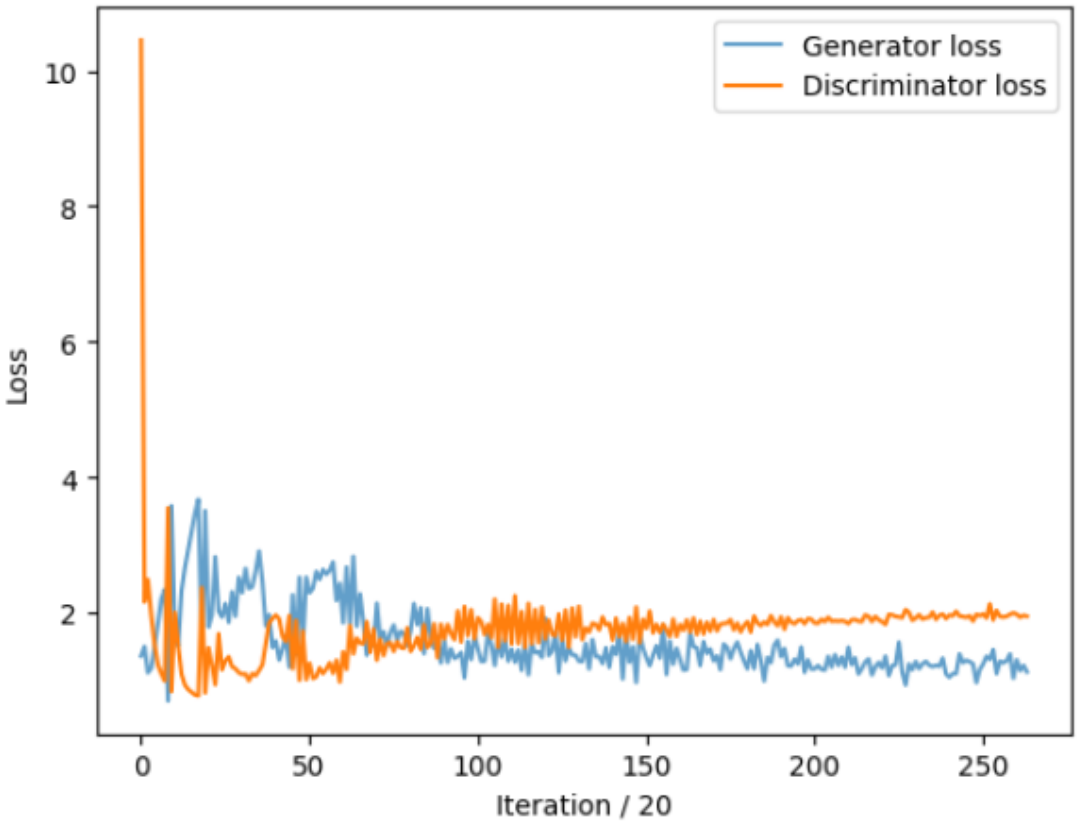
الفا و بتا مربوط به SSGAN را هم مطابق زیر تعیین کردیم:

```
alpha = 0.25
beta = 0.25
```

به دلیل زمان زیادی که آموزش شبکه می گرفت از ۱ چهارم دیتا استفاده کردیم.

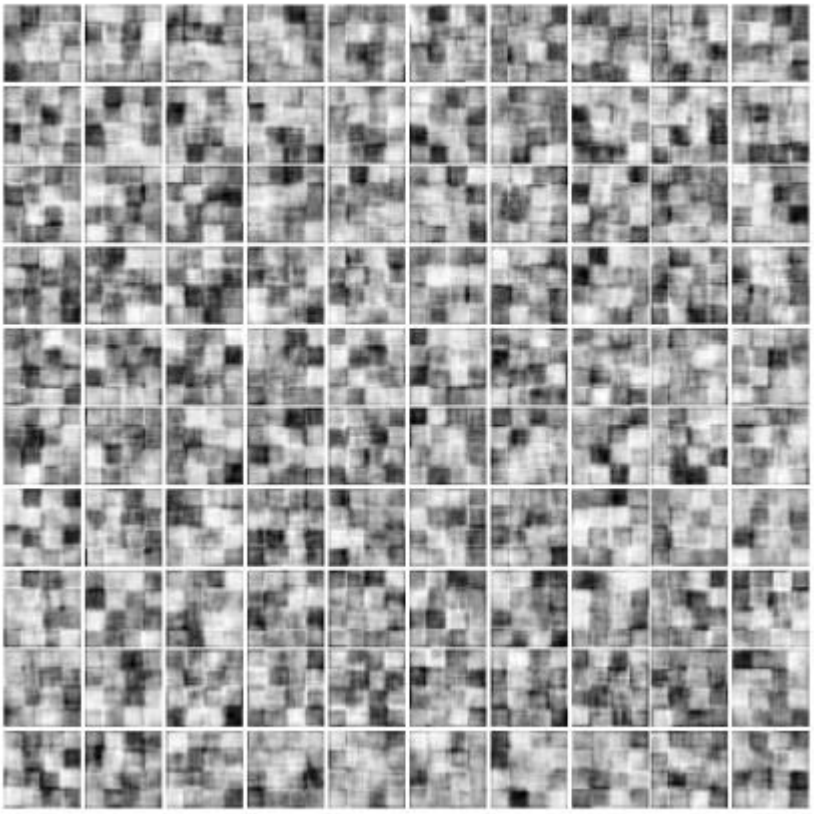
ارزیابی مدل :

نمودار لاس ها پس از ۴۵ ایپاک :



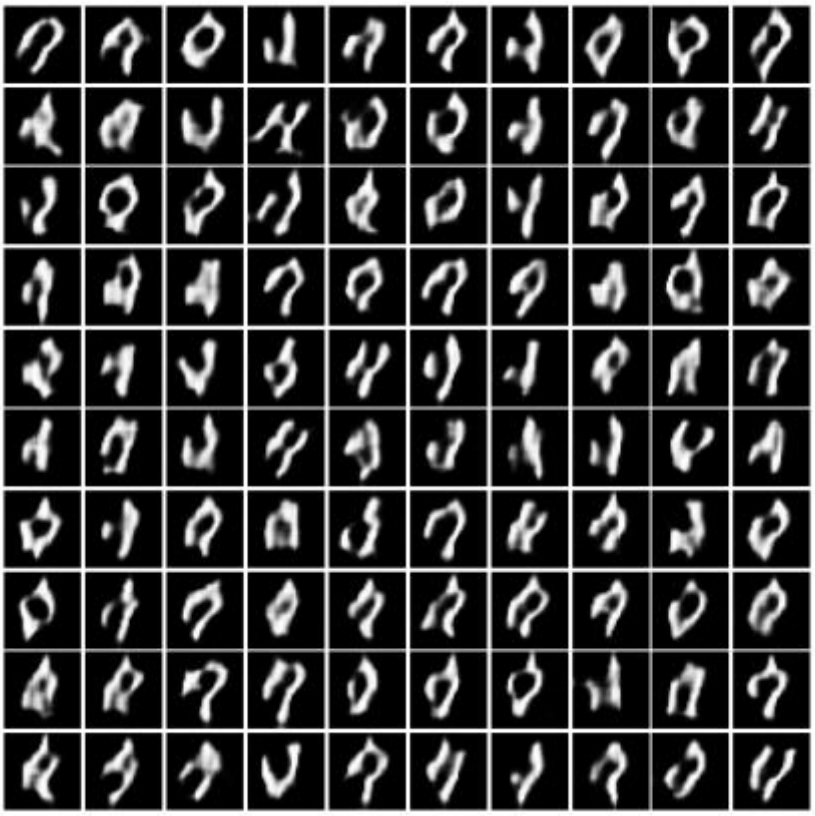
خروجی مدل به ازای **iteration =0** که در واقع صرفا همان نویز اولیه تولید شده می باشد:

```
epoch: 0
```



خروجی مدل به ازای مراحل میانی :

```
epoch: 20
```



خروجی مدل به ازای مرحله ی نهایی:

epoch: 44

همانطور که دیده می شود مطابق نمودار **loss** مدل به خوبی در حال کانورج شدن است منتها فرصت و ایپاک بیش تری لازم دارد تا به خروجی مطلوب برسد و ما فرصت نکردیم بیش از این مدل را آموزش دهیم. اگر زمان کافی و زیرساخت های پردازشی بیش تر در اختیار باشد این مدل عملکرد بهتری از مدل اولیه نشان می دهد.