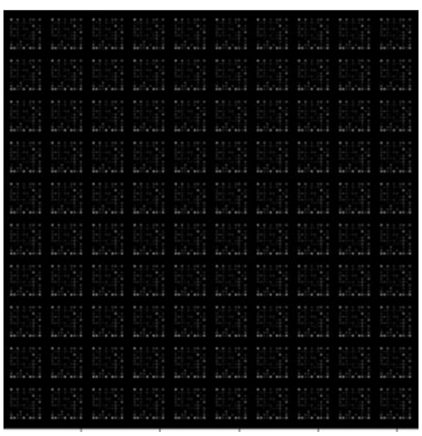
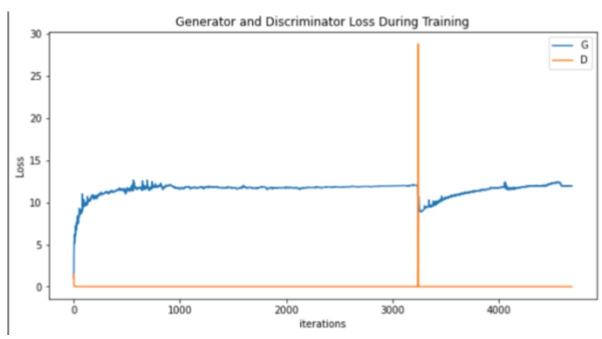
***Note the beginning is a little longish. I was hoping you could skim through it. Otherwise, you could read page 1,2,6 10 on.**

I want to start by experimenting with large G and D and see if I can maintain fragile balance. (This would help me if I chose to experiment later in the assignment with Celeb datasets.) Here is the new network.

```
class Generator(nn.Module):
  #may even add another don layuer
    def __init__(self,latenSpace_dim):
      super().__init__()

      self.generator_cnn = nn.Sequential(

        nn.ConvTranspose2d(100,256,4,stride=1, padding= 0, bias=False),
        nn.BatchNorm2d(num_features=256),
        nn.ReLU(True),

        nn.ConvTranspose2d(256,256,4,stride=2, padding= 1, bias=False),
        nn.BatchNorm2d(num_features=256),
        nn.ReLU(True),

        nn.ConvTranspose2d(256,128,4,stride=2, padding= 1,bias=False),
        nn.BatchNorm2d(num_features=128),
        nn.ReLU(True))

      self.generator_cnn2 = nn.Sequential(
        nn.ConvTranspose2d(128,32,1,stride =1, padding =2,bias=False),
        nn.BatchNorm2d(num_features = 32),
        nn.ReLU(True),
        # state size 32 x 32.

        nn.ConvTranspose2d(32, 1, 1, stride=2, padding=1,bias=False),
        nn.Tanh()
        )
```

```
class Descriminator(nn.Module):
    def __init__(self):
        super().__init__()
        #stat (variable) descriminator_cnn1: Sequential
        self.descriminator_cnn1 = nn.Sequential(
          nn.Conv2d(1,32,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=32),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn2 = nn.Sequential(
          nn.Conv2d(32,32,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=32),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn3 = nn.Sequential(
          nn.Conv2d(32, 64, 4, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=64),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(64,1,4, stride =2, padding = 1),
          nn.Sigmoid()
        )
```

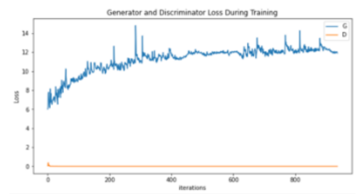




Initial vanishing gradient problem

It seems that the generator does not have ample gradients to update its weights. This occurs after only a few iterations. There may be two explanations: G is overly good making D overly wrong in turn causing G to lose incentive to do any better. On the other hand, D is overly good not allowing G to learn.
By looking at Dloss which converges rapidly to 0, It is clear we lie in the second case where G doesn't learn.

```
[0/938]     Loss_D: 0.4678  Loss_G: 0.3535
[50/938]    Loss_D: 0.0027  Loss_G: 0.9304
[100/938]   Loss_D: 0.0005  Loss_G: 0.9628
[150/938]   Loss_D: 0.0002  Loss_G: 0.9775
[200/938]   Loss_D: 0.0001  Loss_G: 0.9822
```



reduce Batch size – didn't solve the issue. Tried reintroducing the leaky Relu in the generator→didn't work. I modified the loss function to satisfy the MSGAN model. This didn't work either. The saturation loss didn't helped either. G doesn't learn.
I strengthen G by increasing the number of feature maps. → doesn't work.
Reduce discriminator learning rate→ doesn't work.

I realize that changes of the network structure including activations layers, size of kernels, addition/removal of linear vs convolutional layers/ learning rate/batch size and lastly different loss functions have not solved in any noticeable way the early convergence of D to zero.

One-sided label smoothing
I penalized the discriminator by setting the labeling of D on the True data to be 0.9. We essentially penalize any prediction with a confidence of above 0.9 for true labels. The literature claims that this is supposed to help greedy parametrization as in the case we have seen in class. However, I hope it will solve the early convergence of D. Findings: This only created an asymptotic bound of the Dloss curve. The early convergence subsists. Also I have a precntimnt that setting the true label to 0.9 will also impairs the leaning of G.

```
[200/938]  Loss_D: 0.3253  Loss_G: 9.8688
[250/938]  Loss_D: 0.3253  Loss_G: 9.7356
[300/938]  Loss_D: 0.3258  Loss_G: 10.2244
[350/938]  Loss_D: 0.3254  Loss_G: 10.1844
[400/938]  Loss_D: 0.3254  Loss_G: 10.4686
[450/938]  Loss_D: 0.3254  Loss_G: 10.6534
[500/938]  Loss_D: 0.3253  Loss_G: 10.8749
[550/938]  Loss_D: 0.3252  Loss_G: 10.8299
[600/938]  Loss_D: 0.3271  Loss_G: 10.9693
[650/938]  Loss_D: 0.3254  Loss_G: 10.9852
```

```
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
Loss_D: 1.0064  Loss_G: 0.6931
```

After much reading I realize that the choice of loss function should only come after a careful consideration of the structure of the model. After the loss function has been chosen, one should play with the hyper parameters to get the best results. Here is a graph that essentially shows that there is no one predominating loss function and that each is sensitive to the parametrization and dataset. Note that this is consistent with my results.

Name	Value Function
GAN	$L_D^{GAN} = E[\log(D(x))] + E[\log(1 - D(G(z)))]$ $L_G^{GAN} = E[\log(D(G(z)))]$
LSGAN	$L_D^{LSGAN} = E[(D(x) - 1)^2] + E[D(G(z))^2]$ $L_G^{LSGAN} = E[(D(G(z)) - 1)^2]$
WGAN	$L_D^{WGAN} = E[D(x)] - E[D(G(z))]$ $L_G^{WGAN} = E[D(G(z))]$ $W_D \leftarrow clip\_by\_value(W_D, -0.01, 0.01)$
WGAN_GP	$L_D^{WGAN\_GP} = L_D^{WGAN} + \lambda E[(\lvert\nabla D(\alpha x - (1 - \alpha G(z)))\rvert - 1)^2]$ $L_G^{WGAN\_GP} = L_G^{WGAN}$
DRAGAN	$L_D^{DRAGAN} = L_D^{GAN} + \lambda E[(\lvert\nabla D(\alpha x - (1 - \alpha x_p))\rvert - 1)^2]$ $L_G^{DRAGAN} = L_G^{GAN}$
CGAN	$L_D^{CGAN} = E[\log(D(x,c))] + E[\log(1 - D(G(z),c))]$ $L_G^{CGAN} = E[\log(D(G(z),c))]$
infoGAN	$L_{D,Q}^{infoGAN} = L_D^{GAN} - \lambda L_I(c,c')$ $L_G^{infoGAN} = L_G^{GAN} - \lambda L_I(c,c')$
ACGAN	$L_{D,Q}^{ACGAN} = L_D^{GAN} + E[P(class = c \mid x)] + E[P(class = c \mid G(z))]$ $L_G^{ACGAN} = L_G^{GAN} + E[P(class = c \mid G(z))]$
EBGAN	$L_D^{EBGAN} = D_{AE}(x) + \max(0, m - D_{AE}(G(z)))$ $L_G^{EBGAN} = D_{AE}(G(z)) + \lambda \cdot PT$
BEGAN	$L_D^{BEGAN} = D_{AE}(x) - k_t D_{AE}(G(z))$ $L_G^{BEGAN} = D_{AE}(G(z))$ $k_{t+1} = k_t + \lambda(\gamma D_{AE}(x) - D_{AE}(G(z)))$

	MNIST	FASHION	CIFAR	CELEBA
MM GAN	9.8 ± 0.9	29.6 ± 1.6	72.7 ± 3.6	65.6 ± 4.2
NS GAN	6.8 ± 0.5	26.5 ± 1.6	58.5 ± 1.9	55.0 ± 3.3
LSGAN	7.8 ± 0.6*	30.7 ± 2.2	87.1 ± 47.5	53.9 ± 2.8*
WGAN	6.7 ± 0.4	21.5 ± 1.6	55.2 ± 2.3	41.3 ± 2.0
WGAN GP	20.3 ± 5.0	24.5 ± 2.1	55.8 ± 0.9	30.0 ± 1.0
DRAGAN	7.6 ± 0.4	27.7 ± 1.2	69.8 ± 2.0	42.3 ± 3.0
BEGAN	13.1 ± 1.0	22.9 ± 0.9	71.4 ± 1.6	38.9 ± 0.9
VAE	23.8 ± 0.6	58.7 ± 1.2	155.7 ± 11.6	85.7 ± 3.8

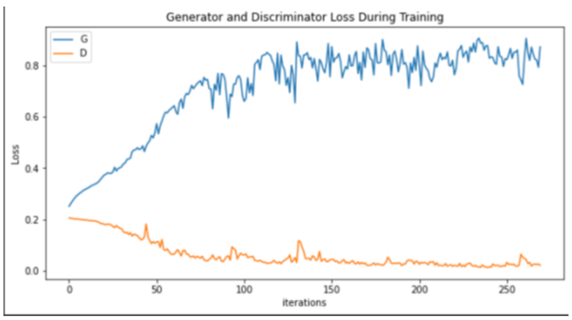
I will therefore go back again and rebuilt my network in a symmetric (allowing constant asymmetry). Here is the new network.

```
class Generator(nn.Module):
  #may even add another don layuer
    def __init__(self,latenSpace_dim):
      super().__init__()
      self.generator_cnn = nn.Sequential(

        nn.ConvTranspose2d(100,Gfm * 8,4,stride=1, padding= 0, bias=False),
        nn.BatchNorm2d(num_features=Gfm * 8),#256
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm * 8,Gfm * 4,4,stride=2, padding= 1, bias=False),
        nn.BatchNorm2d(num_features=Gfm * 4),
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm * 4,Gfm * 2,4,stride=2, padding= 1,bias=False),
        nn.BatchNorm2d(num_features=Gfm * 2),
        nn.LeakyReLU(0.2, inplace=True))

      self.generator_cnn = nn.Sequential(

        nn.ConvTranspose2d(Gfm * 2,Gfm,1,stride =1, padding =2,bias=False),
        nn.BatchNorm2d(num_features = Gfm),
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm, 1, 1, stride=2, padding=1,bias=False),
        nn.Tanh()
        )

    def forward(self, x):
      x = self.generator_cnn(x)
      x = self.generator_cnn1(x)
      return x
```

```
class Descriminator(nn.Module):
    def __init__(self):
        super().__init__()
        #state size 28 x 28 x 1
        self.descriminator_cnn1 = nn.Sequential(
          nn.Conv2d(1,Dfm *8,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm *8),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn2 = nn.Sequential(
          nn.Conv2d(Dfm *8,Dfm *4,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm *4),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn3 = nn.Sequential(
          nn.Conv2d(Dfm *4, Dfm *2, 4, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm *2),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm *2, Dfm, 4, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm,1,3, stride =2, padding = 1),
          nn.Sigmoid()
```


```
[0/10][0/938]   Loss_D: 0.6938  Loss_G: 0.7072
[0/10][50/938]  Loss_D: 0.5976  Loss_G: 1.0611
[0/10][100/938] Loss_D: 0.4032  Loss_G: 1.4304
[0/10][150/938] Loss_D: 0.3989  Loss_G: 1.5299
[0/10][200/938] Loss_D: 0.4857  Loss_G: 1.0773
[0/10][250/938] Loss_D: 0.3610  Loss_G: 1.4039
[0/10][300/938] Loss_D: 0.3748  Loss_G: 1.6351
[0/10][350/938] Loss_D: 0.3347  Loss_G: 2.1114
[0/10][400/938] Loss_D: 0.2694  Loss_G: 1.8896
[0/10][450/938] Loss_D: 0.2717  Loss_G: 1.9058
[0/10][500/938] Loss_D: 0.2306  Loss_G: 2.3556
[0/10][550/938] Loss_D: 0.2078  Loss_G: 2.6332
[0/10][600/938] Loss_D: 0.3619  Loss_G: 1.7813
```

For our new network, the BCELoss is unstable. Further the loss value initiates at a suspiciously low value. I changed the ratio of feature maps between G and D. I updated Gfm from 32 to 64 and Dfm from 32 to 28. I also changed the batch size back to 126. (As long as there is no early convergence of D I don't think there is a difference in batch size. And I think that a network that operates with a larger batch size is a strength. Here are the results:



- Here we can almost see a zoomed in version of previous graph. See how Gloss is almost inversely symmetric to Dloss. As D learns, G's task becomes harder. This is exactly our problem I surmise.

This again didn't work. Lets play around with the learning rate of G and D. I will set it with a ratio of 3:1 respectively. I will also switch back to the original Binary cross entropy loss BCELoss. Here are the results.

```
[0/469]     Loss_D: 0.6955  Loss_G: 0.6883
[50/469]    Loss_D: 0.6290  Loss_G: 0.9659
[100/469]   Loss_D: 0.5314  Loss_G: 0.8661
[150/469]   Loss_D: 0.5935  Loss_G: 0.9008
[200/469]   Loss_D: 0.4660  Loss_G: 1.2802
[250/469]   Loss_D: 0.4864  Loss_G: 1.2802
[300/469]   Loss_D: 0.4355  Loss_G: 1.2500
[350/469]   Loss_D: 0.3210  Loss_G: 1.6436
[400/469]   Loss_D: 0.3145  Loss_G: 1.7970
[450/469]   Loss_D: 0.2796  Loss_G: 2.0663
[0/469]     Loss_D: 0.2747  Loss_G: 2.1323
[50/469]    Loss_D: 0.2243  Loss_G: 2.2839
[100/469]   Loss_D: 0.2095  Loss_G: 2.6682
[150/469]   Loss_D: 0.1944  Loss_G: 2.9545
[200/469]   Loss_D: 0.1886  Loss_G: 3.1824
[250/469]   Loss_D: 0.1839  Loss_G: 3.4133
[300/469]   Loss_D: 0.1827  Loss_G: 3.5942
[350/469]   Loss_D: 0.1992  Loss_G: 3.7633
```

- At the beginnning we see a more stable training. Although D overwhelms G we still see a gradual convergence and a less aggressive vanishing gradients. Something I didn't have on many occasions. Note that this is still a failure.

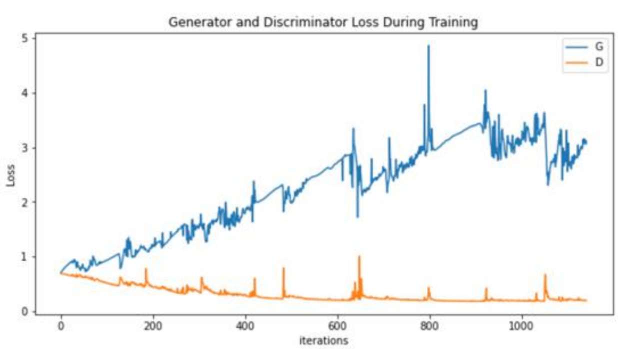
I have many options. I could try to reintroduce the FC layer in the generator as was instructed in the exercise. I could also try to introduce gaussian noise to the input of D. I could also train G multiple times for each train of D.

After all this experimenting, I realize that trying to make G stronger doesn't have the intended result. Maybe we should try to restrict D instead. Here is the new network:

```
class Descriminator(nn.Module):
    def __init__(self):
        super().__init__()
        #state size 28 x 28 x 1
        self.descriminator_cnn1 = nn.Sequential(
          nn.Conv2d(1,Dfm ,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn2 = nn.Sequential(
          nn.Conv2d(Dfm,Dfm *4,4,stride =2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm *4),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn3 = nn.Sequential(
          nn.Conv2d(Dfm *4, Dfm *2, 4, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm *2),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm *2, Dfm, 4, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=Dfm),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm,1,3, stride =2, padding = 1)
          # ,nn.Sigmoid() no need if I use BCEloss-logit
```

```
batch_size = 256
LATEN_DIM = 100
EPOCH = 5
lr_D = 0.002
lr_G = 0.008
#nets feature map
Gfm = 64
Dfm = 32
```

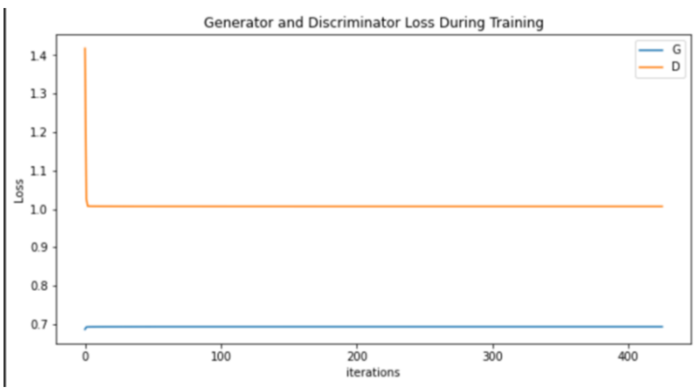
```
    def __init__(self,latenSpace_dim):
      super().__init__()
      self.generator_cnn = nn.Sequential(

        nn.ConvTranspose2d(100,Gfm * 4,4,stride=1, padding= 0, bias=False),
        nn.BatchNorm2d(num_features=Gfm * 4),
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm * 4,Gfm * 4,4,stride=2, padding= 1, bias=Fals
        nn.BatchNorm2d(num_features=Gfm * 4),
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm * 4,Gfm * 2,4,stride=2, padding= 1,bias=False
        nn.BatchNorm2d(num_features=Gfm * 2),
        nn.LeakyReLU(0.2, inplace=True))

      self.generator_cnn1 = nn.Sequential(

        nn.ConvTranspose2d(Gfm * 2,Gfm,1,stride =1, padding =2,bias=False),
        nn.BatchNorm2d(num_features = Gfm),
        nn.LeakyReLU(0.2, inplace=True),

        nn.ConvTranspose2d(Gfm, 1, 1, stride=2, padding=1,bias=False),
        nn.Tanh()
```




```
[0/5][0/235]     Loss_D: 0.6936  Loss_G: 0.7093
[0/5][50/235]    Loss_D: 0.6265  Loss_G: 0.7900
[0/5][100/235]   Loss_D: 0.5159  Loss_G: 0.9359
[0/5][150/235]   Loss_D: 0.5013  Loss_G: 1.1394
[0/5][200/235]   Loss_D: 0.4382  Loss_G: 1.2025
[1/5][0/235]     Loss_D: 0.3807  Loss_G: 1.3114
[1/5][50/235]    Loss_D: 0.3927  Loss_G: 1.6847
[1/5][100/235]   Loss_D: 0.3424  Loss_G: 1.6495
[1/5][150/235]   Loss_D: 0.2982  Loss_G: 1.7229
[1/5][200/235]   Loss_D: 0.2538  Loss_G: 2.0314
[2/5][0/235]     Loss_D: 0.2338  Loss_G: 2.2546
[2/5][50/235]    Loss_D: 0.2616  Loss_G: 2.1938
[2/5][100/235]   Loss_D: 0.2106  Loss_G: 2.5689
[2/5][150/235]   Loss_D: 0.2005  Loss_G: 2.8477
[2/5][200/235]   Loss_D: 0.2423  Loss_G: 2.2918
[3/5][0/235]     Loss_D: 0.2251  Loss_G: 2.5785
[3/5][50/235]    Loss_D: 0.2209  Loss_G: 2.6335
[3/5][100/235]   Loss_D: 0.2278  Loss_G: 3.3418
[3/5][150/235]   Loss_D: 0.1892  Loss_G: 3.2224
[3/5][200/235]   Loss_D: 0.1833  Loss_G: 3.4295
[4/5][0/235]     Loss_D: 0.2123  Loss_G: 2.8071
[4/5][50/235]    Loss_D: 0.1921  Loss_G: 3.1049
[4/5][100/235]   Loss_D: 0.1893  Loss_G: 3.2779
[4/5][150/235]   Loss_D: 0.2008  Loss_G: 2.9861
[4/5][200/235]   Loss_D: 0.2019  Loss_G: 3.1143
```

Although the training process seems stable. G didn't manage to learn. This is extenuating. Do you know why I get it wrong every time? Is my tuning to harsh? What would you do? Is the Linear layer of at the beginning of G important? DCGANS do not use although it is heavy. Before I give up, I'll try to strive for more symmetry. Ill reduce the batch size to 64, update Dfm = Gfm =128, update D_lr = G_lr = 0.002. I will also try a substantially larger epoch number of 20, just to be sure that G doesn't somehow recover and that Dloss is always low regardless. Findings: G again didn't learn.

I will not give up. I will do two things. I'll first try to introduce some noise in the real images D receive. This essentially weakens D. A strong D behaves like a step function, thus there's no useful gradient signal for updating G. I read that introduction of noise in the input of D is equivalent to a gradient penalty. It is interesting whether R1 regularization would be suitable here.
Note that D_loss_fake converges faster than D_loss_real (see below paper). Thus, the vanishing of Dloss is due to the convergence of Dloss_fake. This is an indication that we should introduce noise there. However, I decided to introduce noise everywhere regardless of if real of fake.



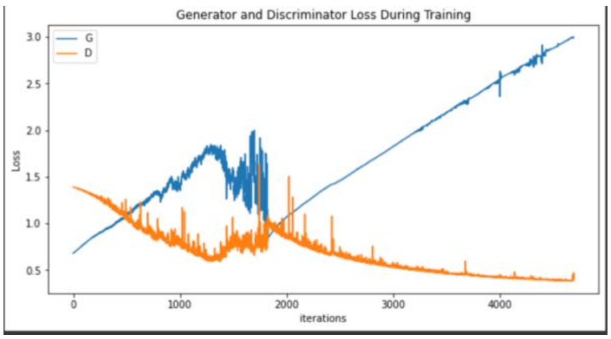
- This obviously didn't work. I do not understand why the losses are constant. Here D fails completely.

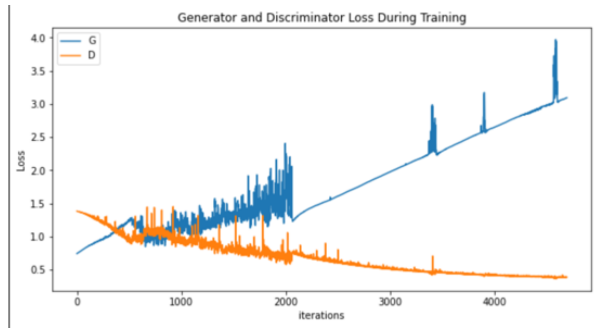
Here is the relevant code:

```
class GaussianNoise(nn.Module):                         # Try
    def __init__(self, std=0.1, decay_rate=0):
        super().__init__()
        self.std = std
        self.decay_rate = decay_rate

    def decay_step(self):
        self.std = max(self.std - self.decay_rate, 0)

    def forward(self, x):
        # if self.training:
        return x + torch.empty_like(x).normal_(std=self.std)
        # else:
        #     return x
```

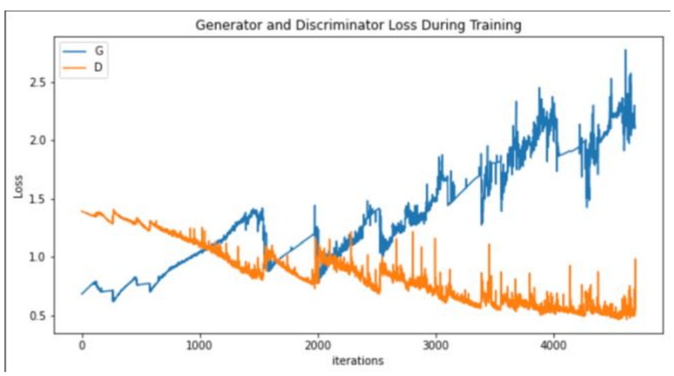
```
class Descriminator(nn.Module):
    def __init__(self,Dfm,std,std_decay_rate): #do I need to pu
        super().__init__()
        #state size 28 x 28 x 1
        self.descriminator_cnn1 = nn.Sequential(
          nn.Conv2d(1,Dfm ,4,stride =2, padding = 1))
          # nn.BatchNorm2d(num_features=Dfm),
          # nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn2 = nn.Sequential(
          nn.Conv2d(Dfm,Dfm *2,4,stride =2, padding = 1),
          nn Loading... 2d(num_features=Dfm *2),
          nn.LeakyReLU(0.2, inplace=True))

        self.descriminator_cnn3 = nn.Sequential(
          nn.Conv2d(Dfm *2, Dfm *4, 4, stride=2, padding = 2),
          nn.BatchNorm2d(num_features=Dfm *4),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm *4, Dfm*8, 4, stride=2, padding = 2),
          nn.BatchNorm2d(num_features=Dfm*8),
          nn.LeakyReLU(0.2, inplace=True),

          nn.Conv2d(Dfm*8,1,4, stride =2, padding = 1),
          GaussianNoise(std, std_decay_rate),          ###does
          nn.Sigmoid() #no need if I use BCEloss-logit
```

The above has failed. I will therefore use a restart scheme to facilitate the convergence of G and impair the one of D. See [paper](). What I did here is use the CosineAnnealingLR scheduler. I attempted to restart the learning rate of only D which will obviously affect G. Here are the results.



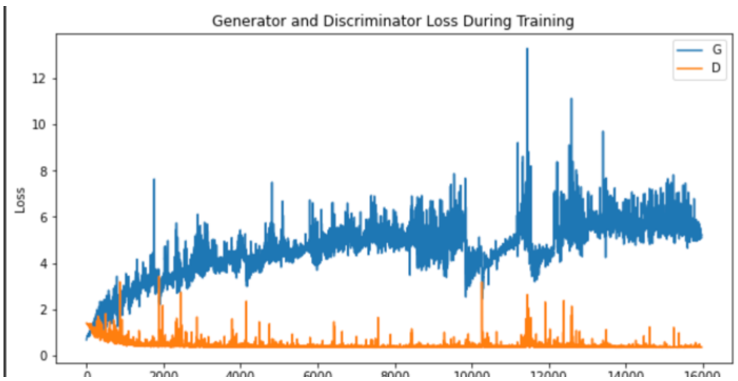


We can clearly see an overfit curve after 2000 iterations. Lets update Gfm = 64, Dfm = 28. Lets also restrict D by setting back the true labels at 0.9(see above). Lets G_lr = 0.002, D_lr = 0.0002. Here are the results:

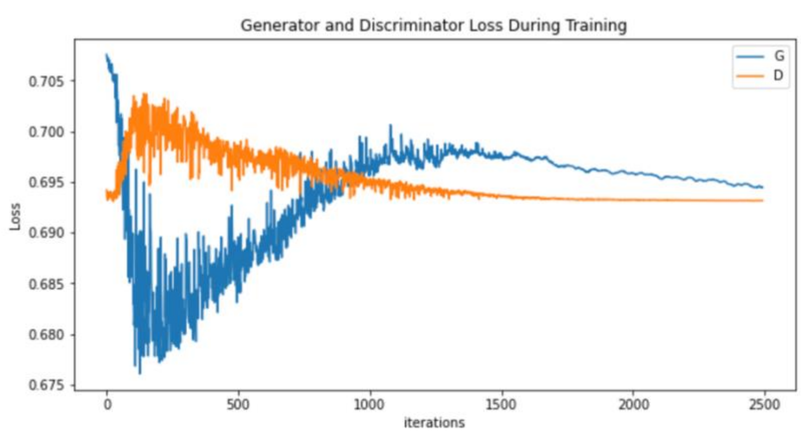


This is a better graph than before. Here we can clearly see the cos pattern effects on the lr and consequently on the loss curves. We also solved the vanishing gradient at the start of the training. However, we still get an underfitted graph further on. We need to make G even stronger. Its is worrying that G couldn’t produce any pictures throughout the training. There seems to be something wrong. I gather that if D is too weak, Gloss will look fine yet G wont learn a thing, where at the moment D learns G starts overfitting.

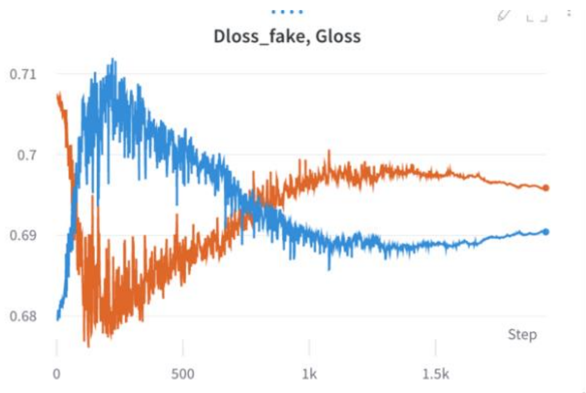
Ill introduce a different scheduler CosineAnnealingWarmRestarts with a factor of 10 between min and max learning rates. Ill update back Dfm =32 and Gfm=64. Ill introduce back a linear layer in G with an additional convolutional layer. Hopefully increasing Dfm to 32 will ensure that D learns.

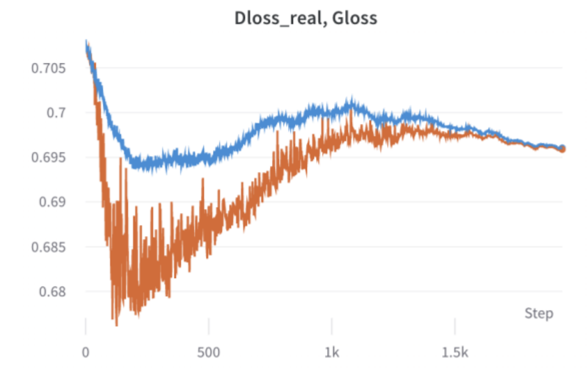


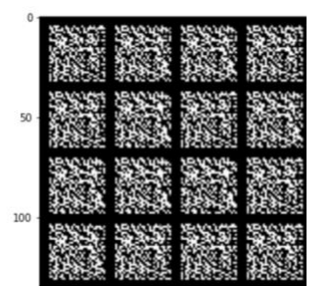
I review my code to check for mistakes. I also upgraded to the Wanbd interface to get better visualization of the losses. I updated the network: Gfm =128,Dfm=32. I also changed the learning schedule to CyclicLR with a min_lr = 0.00002 and max_lr of 0.002. Note the min_lr is very low while the max_lr is exactly the learning rate of G. I hope that this will sufficiently impair D to allow G to learn. Here are the results:



Finally, I was able to achieve a good convergence of the losses! Here are plots of Gloss compared with the real and fake Dlosses. Its nice that we can see the obvious symmetry in the curves Gloss and Dloss_fake.

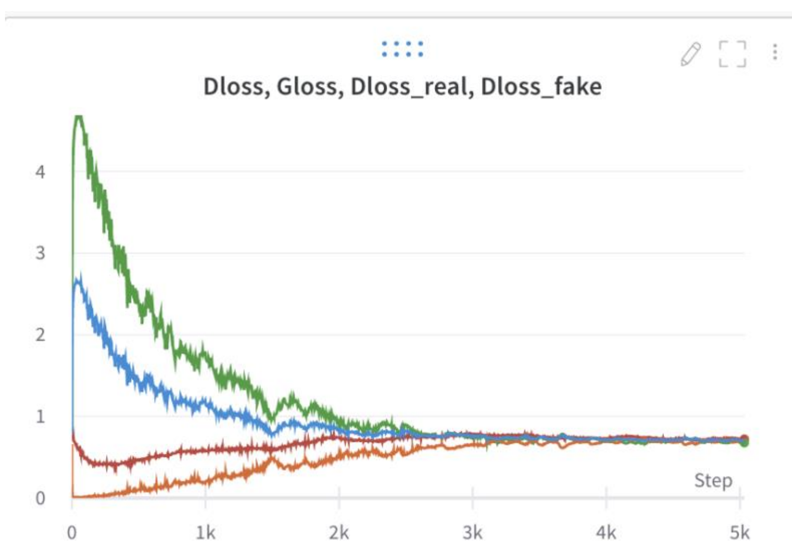




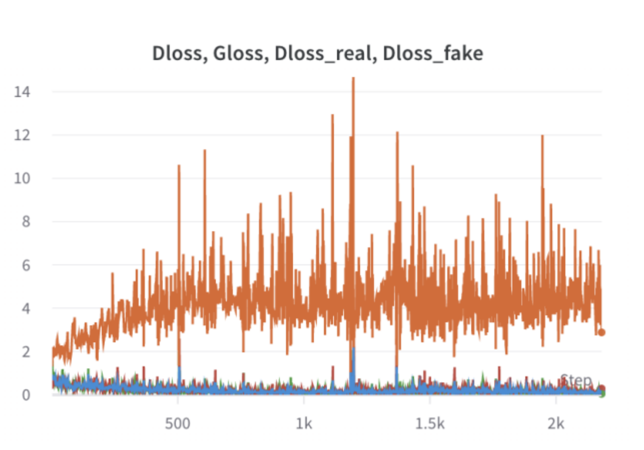


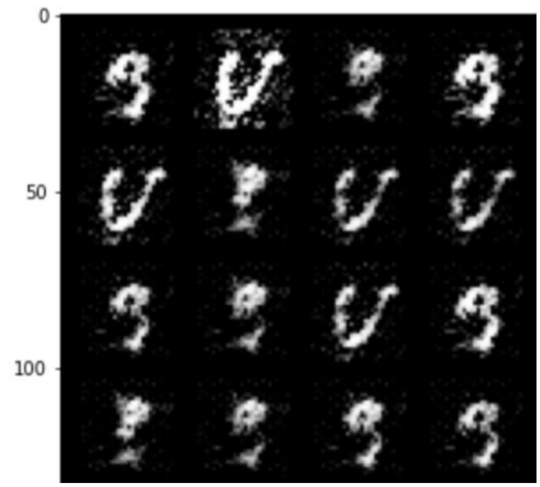
The problem however is that G still doesn't learn. Maybe D must learn better. If I didn't know any better, I would have upscaled the parameters and keep the same ratio. However, since we didn't get any learning of G, I think it best to downscale the network's activation maps while keeping the same ratio. Here I used network 2. See code. Here are the results:

First, I don't understand how come Gloss is lower than Dloss through the training. I expected Gloss to be higher than Dloss (at around 1-3).
After a couple of tried I realize that this is because I set the ground truth = 0.9 and used the schedule on D. G still didn't learn. (G in orange)



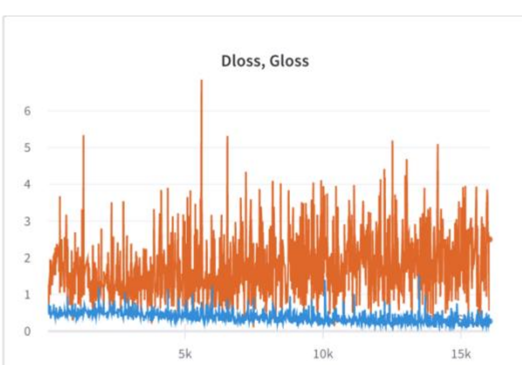
I removed the scheduler and set the ground truth label = 1. We're back to the early convergence of D. However, G did learn! was the scheduler the problem? Here I learn that even if Dloss is low throughout the training, as long as there is no vanishing gradient scenario, it is ok.



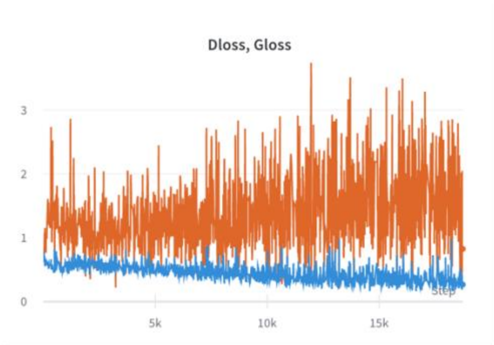


I come to realize that I need to conduct more research on schedulers before using them. Ill drop the scheduler and also simplify the network to rather concentrate on a larger epoch number for training. I also played with different learning rates of D while keeping G_lr =0.002. The saturation loss as well as the MSE loss performances are shown below. I set True label = 1, Batch_size = 128. This is my last network. See assignment code for more details (network 2).

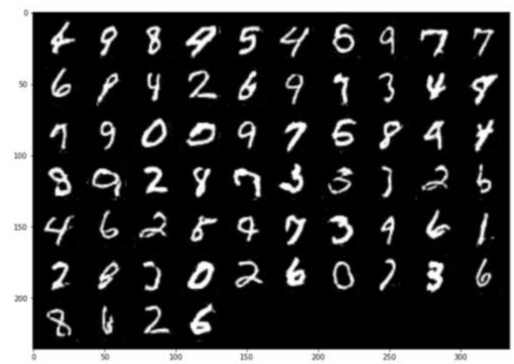
LR_G = LR_D = 0.002



LR_G =0.002, LR_D = 0.001

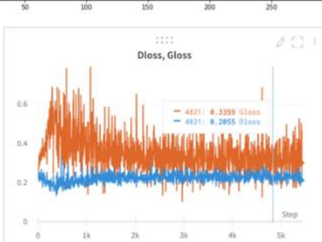


LR_G =0.002, LR_D = 0.0005

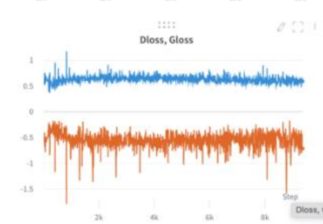


There doesn't seem to be a noticeable difference in performance when I change D's learning rate.

LR_G =0.002, LR_D = 0.0005, Loss=MSE



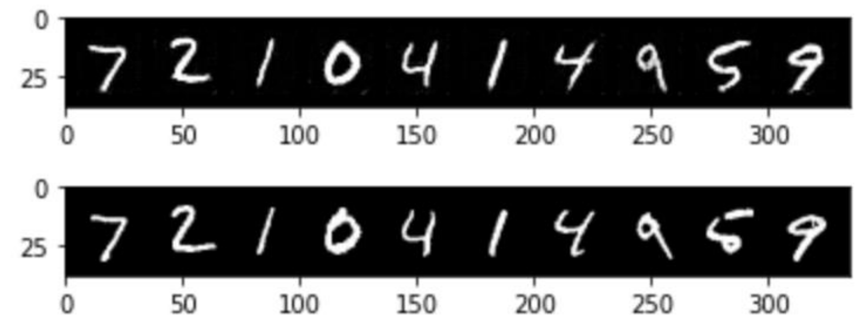
LR_G =0.002, LR_D = 0.0005, Loss=saturation

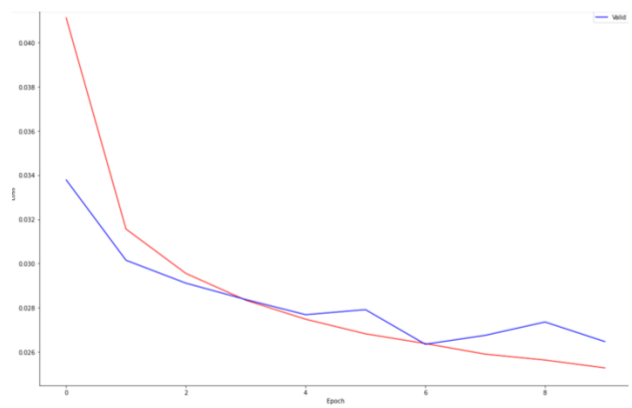


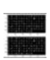
MODEL-INVERSION:

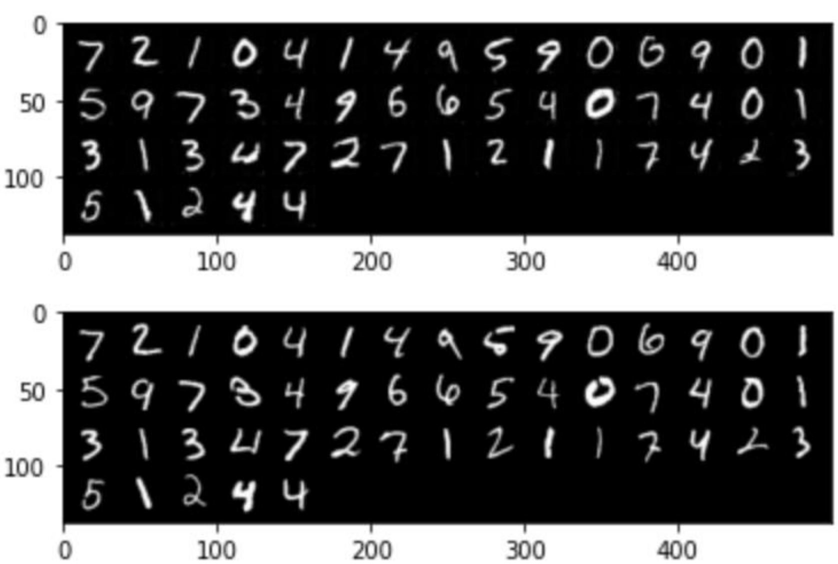
Here I used the original trained generator from the GAN. I then trained an encoder to turn an image to its laten space code. I then fed this code into G and got a new generated image. We calculate the loss between the input of E and the output of G. Although it is hard to see when training on the MNIST data, there are limitations that may affect the model inversion.

Since we take a trained generator, what we are essentially doing is training our encoder to produce a laten space that is best for our trained G. That is, if E is not strong enough, the mapping of the laten space will be impaired and so is the performance of G. This will affect the final inversion. Further, there are two common mode collapses (that I know of) that can occur. The first is when G is trained too well whereby mapping different laten codes to the same set of images. We will be in the lookout for this. The second case occurs when G is limited in the “features” that it can generate. This occurs when G only learns a “knit graph” of the original data distribution. I think that Gan inversion is a great tool to evaluate the second mode collapse. Mathematically, the distribution entailed by G doesn’t span the entire target distribution (We have seen in class the example of the bars that disappear in the generated image).









Note the reconstructed images are on the top. Here we can see the second case of mode collapse. If we look carefully, we notice that G doesn’t know how to reconstruct the horizontal line that passes through the number 7. This is exiting.

The first mode collapse scenario doesn’t seem to occur here as we can see many different generated forms and directions for the same number.

I am happy with the results.

Wasserstein Auto-Encoders (AEs)

I.

I recuperated the AE from the last exercise. The reconstruction loss is the normal AE loss.(MSE in our case). Here we define the Wasserstein distance based on the set of functions F which contains the fist, the second, and forth moments (monomials to be precise). That is, mean, std, Kurtosis.

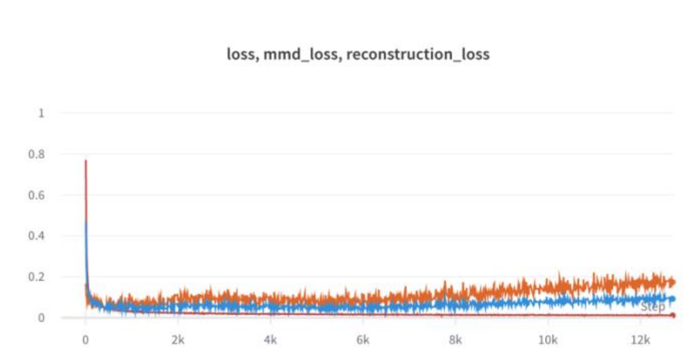
I encountered an interesting problem resulting in my MMD loss exploding exponentially. This was solved by zeroing the gradients before calculating the loss. I wonder what part of not zeroing the gradients results in this exponential growth. Here is the code:

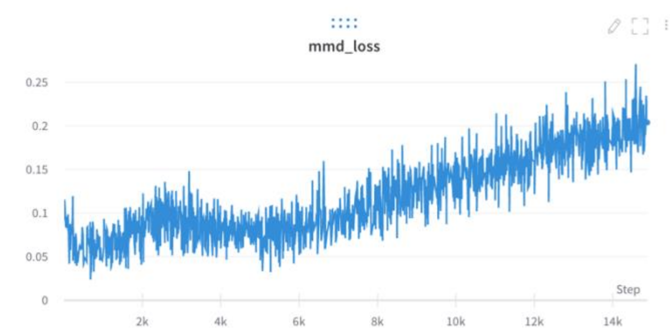
```
def train_WAE():

  wandb.init(project="WAE", entity="matancoo",settings=dict(start_method='thread'))

  encoder.train()
  decoder.train()
  for epoch in range(EPOCH):
    for i, data in enumerate(trainloader, 0):
      images, true_labels = data
      encoder.zero_grad() #The mmd_loss explodes if not
      decoder.zero_grad()
      #Forward pass
      encoded_images = encoder(images)
      print(encoded_images.shape)
      print(encoded_images.squeeze().shape)
      noise = torch.randn(images.shape[0], LATEN_DIM) #maybe need gaussian distribu here instead
      mmd_loss = calculate_mmd_loss(encoded_images.squeeze(),noise)
```

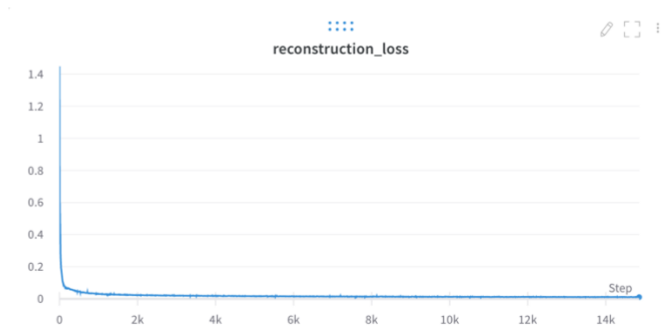
Its hard to give serious a qualitative analysis here. This is the disadvantage of working with the MNIST dataset.

Note that here the training of the AE was faster and there is no mode collapse (see horizontal line on 7). This is the advantage of being a reconstructing network rather than an adversarial one.

I am satisfied with the results.

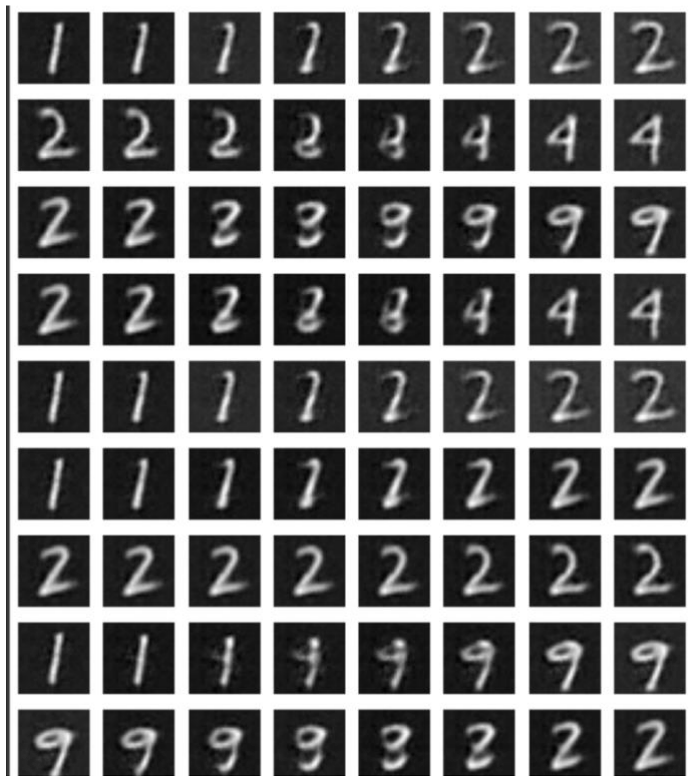






II.

First, I think that the problem with interpolation as a qualitative assessment of the laten space is that it is not very scientific and that as the dataset becomes simpler the assessment losses its claimed efficacity. This being said, we can see some smoothness/structure in our pictures/laten space. We arrived with little training to the same or even more “smoothness” in our interpolated images than our previous exercise AE. I think the fact that our laten space is “structured” around the gaussian distribution (we limited ourselves to 3 moments) is the major reason for the results. The simple AE’s enshrined laten space is unstructured and so the random sample of noise in the laten space loses its semantic meaning which translates to less relation between semantic meaningful points in the laten space or in other words, less smoothness. An interesting point is that the black background of the images lightens through the interpolation process. This seems consistent with the mmd loss which takes averages and higher moments. (Right?). If the background wasn’t constant (not only black) I would have expected it to be blurry.



Theoretical Questions:

Q1) The glow setting uses the following approach:

Sample $z \sim N(0,1)$ and

- Defining a new R.V $x \doteq M_\theta(z)$, which by twicking θ becomes our target variable.

Invertible.

To sample from the distribution entailed by data x_i we assume $M_\theta \in Gl_n(\mathbb{R})$.

Allowing the following formulation:

$$P(x) = \left| \frac{\nabla M^{-1}(x_i)}{\nabla x} \right| P_z(M^{-1}(x))$$

Note M_θ is a generative reversible model. This mean that it can make EXACT laten-vector inferences (since Reversible)

THis allows us to calculate the exact log likelihood. of The data since we can get the exact distribution in laten space $x_i \sim P_\theta$.

THUS The Glow setting calculates $\max_\theta \sum_{i=1} \log P_\theta(x_i) =$

From our equation above:

$$= \max_\theta \left\{ \log P_z(z) + \sum_{i=1}^{k} \log \left| dh_{k-1}^i / dh_k^i \right| \right\} \quad \text{where } M = h_1 \circ \cdots \circ h_k \quad \text{For } z = M_\theta^{-1}(x_i)$$

θ defines weights in $h_1 \ldots h_k$.

$R^?$

The change of variable also occurs in the encoder of auto-encoders Yet the encoder itself is not reversible thus $P(x)$ cannot be optained as above.

GANS do not even have an encoder.

and thus do not have an identity relation between the target distribution and the latent space. (note that GAN inversion although Partially working) is still not the same thing

Lastly in the Glo setting a function is drawn from the data to the latent space. by allocating a learnt latent code for each data point.

Yet this in does mean that there is a one-to-one relation between the latent space and the data.

continuous discrete

decoder in Glo

$\hookrightarrow$ we can't draw random $z \sim P_z$ and expect $D_\theta(z) \sim P_{data}$.

There is no Analytic function that knows How to map noise to a signal. Since there is no meaning is the special space of the noise.

So here in Glo we change the z_i. but we do not shape in a continuous manner the latent space in the way Glow does. which would be essential to generalize the data X.

(2) The gradients of D with respect to G will vanish: Here is the Loss of D.

$$J^D = E_{x\sim P_{data}}\Big[\underbrace{\log\left(D_{\theta_D}(x)\right)}_{\text{constant when we derive one to } \theta_G}\Big] + E_z\left[\log\left(1 - D_{\theta_D}\left(G_{\theta_G}(z)\right)\right)\right]$$

Applying log we can calculate the gradient of the Loss J^D.

$$\nabla_{\theta_G} J^D = E_z\left[-D'_{\theta_G}\left(G_{\theta_G}(z)\right) \cdot \frac{1}{1 - D_{\theta_D}\left(G_{\theta_G}(z)\right)}\right]$$

Since D_θ becomes a step function lets evaluate the above:

$$D_\theta\left(G_\theta(z)\right) \rightarrow 1 \Rightarrow \nabla_{\theta_G} J^D = -\infty$$ exploding

$$D_\theta\left(G_\theta(z)\right) \rightarrow 0 \Rightarrow \nabla_{\theta_G} J^D = 0$$ vanishing Gradient

Q3) Since we run once on D and once on G it is obvious that the two Problems are equivalent $\min_G \max_D \equiv \max_D \min_G$ given that there exist an equilibrium.

a) Min-G $V(D,G)$ tries to minimize the # of times D is right in its classification of real and fake.

b) Max-D $\left(\text{Min-G } V(D,G)\right)$ tries to maximize upon D_θ - That is, Finds θ such that D_θ Maximize the right classifications given Fixed converged $G_{\theta min}$.

Note that this, in simpler words is just training a classifier which is I think an easy task even for a limited NN. (It is a supervised training environment).

c) In the classical Problem $\min_G \max_D V(D,G)$ if D fails the G_{min} wont necessarily be the intended one. That is, G will not achieve Perfect generalization.

In the new-Practicle-equivalent Problem $\max_D \min_G V(D,G)$, For a Fixed $G_{\theta min}$, The problem becomes $\max_D V(D, G_{\theta min})$. Note that Here we would expect several iteration of Gradient descent of G For each of D. However as we know, there is a 1:1 ratio between G's GD and D's GD. Since the training of D is easier than of G (see why above) I would expect Vanishing gradients of D.

In the previous formulation the unsymmetry in the training complexity between G and D was ballanced by the 1:1 ratio of GD.
In the second formulation it is not the case !!