

Assignment 3 - MNIST with GANs

2021320117 Bae Minseong

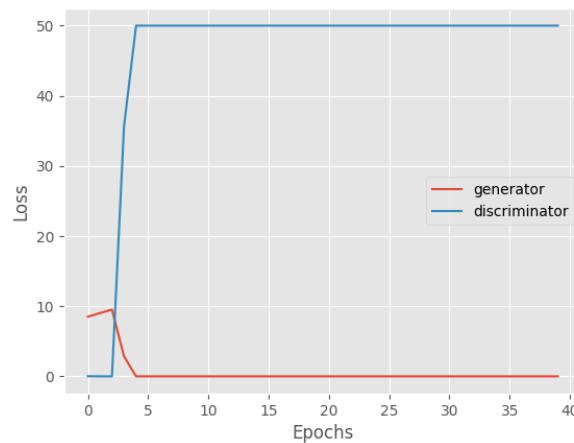
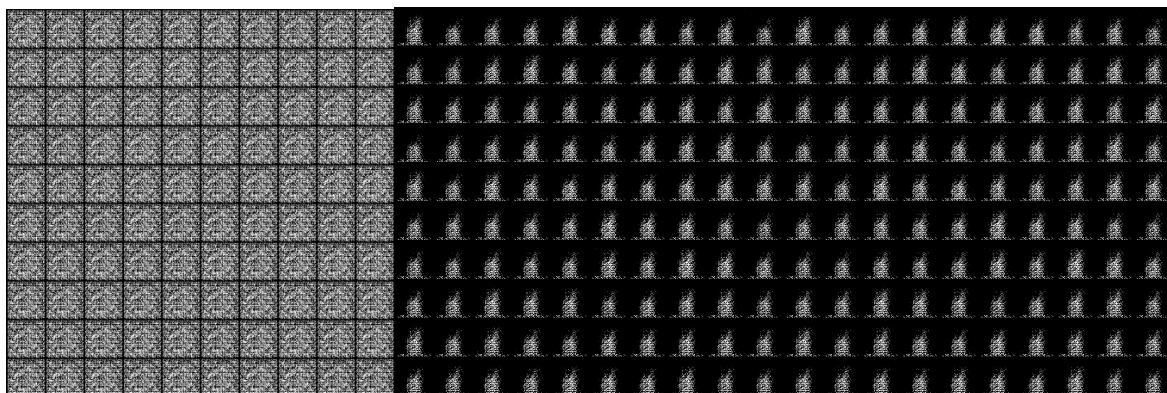
우선, 해당 과제를 진행하면서 제작한 모든 visualization 결과 및 model의 checkpoint를 아래 드라이브 주소에서 확인할 수 있으며, 이 중 필요한 일부를 보고서에 이미지 형태로 첨부하였다.

(<https://drive.google.com/drive/folders/1Hy-6hlxJuXIFc7BFF9iEWSj1jrsRcFKI?usp=sharing>)

1. Learn a GAN (deconv) with 64x64

a. Hyperparameter tuning & Analysis

수업 Official Github (<https://github.com/mlvlab/data303>)의 “Image_Generation_by_GAN.ipynb” 파일을 Google Colab 상에서 실행시켜 다음과 같은 결과를 얻을 수 있었다. Default hyperparameter는 lr = 2e-4, batch_size = 512, latent_dims = 100 이다. 먼저, 아래는 40 epoch 의 학습 과정에서 동일한 latent vector에 대해 각각 0, 20, 40 epoch에서의 결과를 나타낸 그림이며, 아래의 그래프는 loss curve이다. Discriminator loss가 4 epoch에서부터 0이 되어버리면서 Generator의 loss가 50이 되어버려 학습이 원활하게 진행되지 않은 모습이다. 이를 통해 discriminator의 loss가 학습 초기부터 지나치게 낮은 경우 학습이 정상적으로 진행되기 어렵다는 사실을 유추해볼 수 있다.



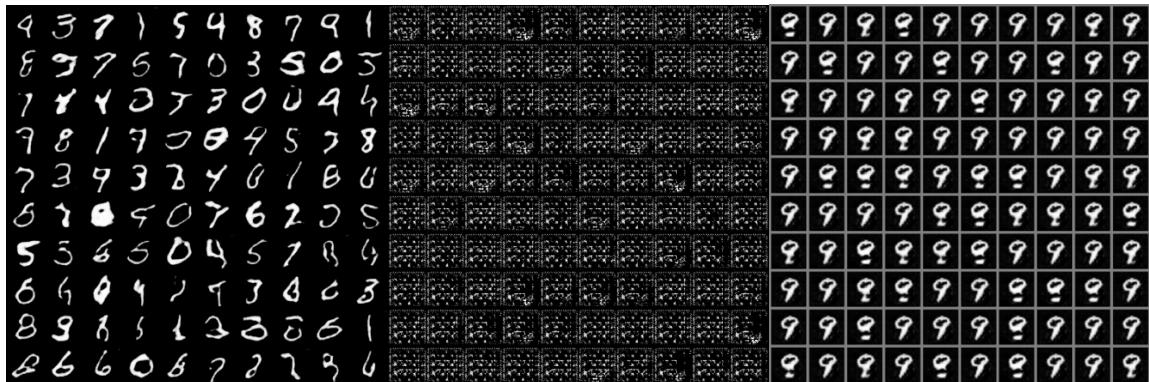
해당 결과를 참고하여 다음과 같이 hyperparameter 값을 바꿔가며 hyperparameter tuning 과정을 진행하였다.

lr	batch_size	latent_dims	Description
2e-4	512	100	Default, bad result
2e-4	512	64	Mode collapse, bad after 9 epochs

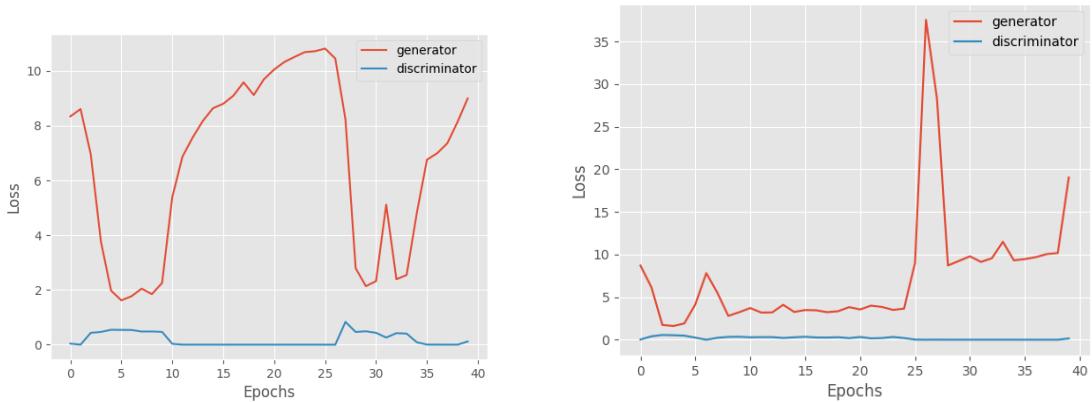
2e-4	512	256	Best
2e-4	64	100	Bad, mode collapse, similar loss curve to default
2e-4	128	100	Quite good, but suddenly unstable at 17 ~ 27 epochs
2e-4	256	100	Mode collapse, bad after 30 epochs
1e-4	512	100	Bad, similar loss curve to default
1e-3	512	100	Bad, similar loss curve to default
2e-4	128	256	Mode collapse, Bad after 10 epochs
2e-4	64	256	Bad, similar loss curve to default
2e-4	64	64	Bad, similar loss curve to default

대체로 default hyperparameter 를 기준으로 latent_dims 를 증가시키거나 batch_size 를 감소시켰을 때 더 좋은 결과가 나왔다. 다만 이를 너무 지나치게 변화시킬 경우 학습이 원활하게 진행되지 않거나 mode collapse 에 빠지는 경우가 많았고, learning rate 를 변화시키는 것은 큰 의미가 없으며 오히려 default 와 같이 discriminator loss 가 아예 0 이 되버리는 현상이 더 빠르게 일어나버렸다.

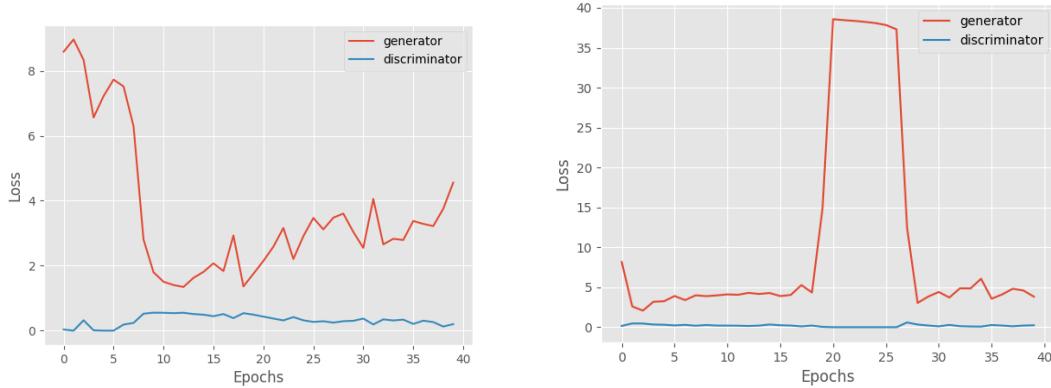
또한, 많은 경우에서 mode collapse 가 발생함을 확인할 수 있었는데, 이 경우 loss curve 에서 discriminator 의 loss 가 낮은 상황에서 generator 의 loss 가 갑작스럽게 증가하는 구간이 존재하는 것을 확인할 수 있었다. 예를 들어 (2e-4, 512, 64)의 경우 10 epoch 까지는 학습이 어느 정도 정상적으로 진행되며 숫자와 비슷한 형태의 결과를 생성해내다가 갑작스럽게 다음 epoch 에서 generator 의 loss 가 증가하며 결과가 이상해지고 최종적으로는 mode collapse 가 일어나는 것을 확인할 수 있었다. 아래 그림이 각각 10/11 epoch 에서의 생성 결과와 40 epoch 후 최종 결과를 나타낸 그림이다.



Loss curve 자체도 굉장히 unstable 한 모습이다. 왼쪽은 (2e-4, 512, 64), 오른쪽은 (2e-4, 256, 100)의 loss curve 이다. 오른쪽 hyperparameter 도 학습이 잘 진행되다가 generator loss 가 갑작스럽게 증가하는 25 epoch 이후 구간에서 갑자기 생성 결과가 이상해지면서 최종적으로 mode collapse 와 유사한 형태가 일어남을 볼 수 있었다.



다양한 hyperparameter setting 들 중 좋은 결과물을 보여준 hyperparameter setting 은 (2e-4, 512, 256)과 (2e-4, 128, 100)이었다. 각각의 모델의 loss curve 를 살펴보면 왼쪽의 (2e-4, 512, 256)의 경우 generator 의 loss 가 큰 폭으로 감소하면서 discriminator 의 loss 가 증가하는 추세를 보이다가, 10 epoch 부근부터 generator loss 가 다시 진동하면서 증가하고 discriminator 의 loss 가 감소하는 추세를 보인다. 추후에 64x64 size 로 학습을 진행한 InfoGAN 학습에서도 이와 같은 loss curve 의 구조가 나올 시에 좋은 결과가 나오는 것을 확인할 수 있었다. 오른쪽의 (2e-4, 128, 100)의 경우 초반부터 generator 와 discriminator 모두 매우 낮은 loss 를 유지하면서 큰 변화가 없다가, 17 ~ 27 epoch 부근에서 갑작스럽게 generator 의 loss 가 증가하며 mode collapse 가 일어나는 경우와 유사한 추세를 보이다가 다시 generator loss 가 감소하며 stable 해지는 모양을 보였다. 해당 학습 과정의 경우 hyperparameter 가 좋았다고 보기보다는 initialization 이 잘 되면서 빠르게 좋은 모델 파라미터를 찾은 것이 아닐까 추측되어진다. 따라서 두 hyperparameter 세팅 모두 결과가 잘 나왔으나 둘 중에서 왼쪽의 경우가 조금 더 안정적으로 학습되었다고 판단하여 best hyperparameter 로 해당 값들을 설정하였으며, visualization 을 진행하였다.



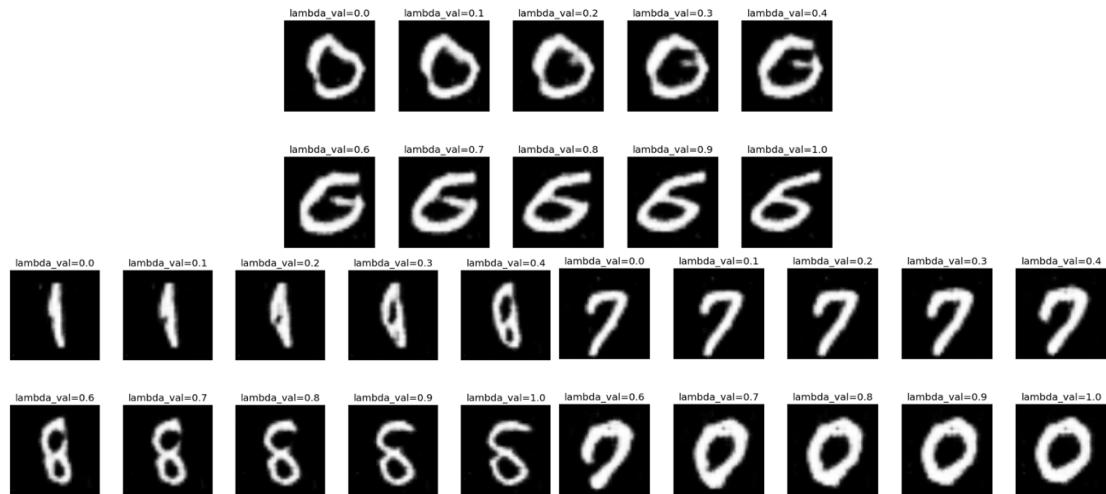
b. Visualize Generated Samples

아래 그림은 임의의 latent vector 100 개로 최종 visualization 을 진행한 결과물이다.



c. Interpolation

아래 그림들은 latent space 상에서 두 개의 random 한 sample에 대해 interpolation을 진행한 결과물이다. lambda 값을 조금씩 조정함에 따라 lambda 값이 0 일때의 숫자에서 1 일때의 숫자로 점차 변화해가는 듯한 모습을 볼 수 있다. 특히 두 번째 그림의 경우 lambda = 0 일때 1과 같은 모양을 한 숫자가 8 과 같은 모양이 되며 두 개의 구멍이 생기면서 최종적으로 lambda = 1 일때 8 과 유사한 숫자가 되는 모습을 볼 수 있다.



d. Visualize generated images using the same latent vectors across epochs

아래 그림들은 40 epoch 동안 학습이 진행되면서 생성한 동일한 latent vector에 대한 generated image를 visualization한 결과물이다. 각 image는 0, 10, 20, 30, 40 epoch의 학습 이후의 generated image이다. imageio 라이브러리를 활용해 gif로도 제작했다. (https://drive.google.com/file/d/18_RIXHeijtQ_8uZzQbTcOnjAUh_zX15/view?usp=sharing)



2. Learn a GAN (deconv) with 28x28

a. Implementation

제공된 코드는 64×64 를 기준으로 구현이 되어 있었기 때문에 이를 MNIST 의 original size 인 28×28 에 맞추어 구현하기 위해 몇 가지를 수정하였다. 우선, img_transform 에서 이제는 64×64 로 resizing 을 할 필요가 없기 때문에 해당 부분을 삭제해주고, Generator 와 Discriminator 를 구현하는 과정에서 28×28 로 차원이 변경된 것을 맞춰주기 위해 ConvTranspose2d 함수와 Conv2d 함수를 수정해주고 그에 맞추어 forward 함수까지 수정해주었다.

```

import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST

img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = MNIST(root='./data/MNIST', download=True, train=True, transform=img_transform)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

test_dataset = MNIST(root='./data/MNIST', download=True, train=False, transform=img_transform)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

class Generator(nn.Module):
    def __init__(self, d=128):
        super(Generator, self).__init__()
        self.deconv1 = nn.ConvTranspose2d(latent_dims, d*4, 4, 1, 0) # 1x1 => 4x4
        self.deconv1_bn = nn.BatchNorm2d(d*4)
        self.deconv2 = nn.ConvTranspose2d(d*4, d*2, 4, 1, 0) # 4x4 => 7x7
        self.deconv2_bn = nn.BatchNorm2d(d*2)
        self.deconv3 = nn.ConvTranspose2d(d*2, d, 4, 2, 1) # 7x7 => 14x14
        self.deconv3_bn = nn.BatchNorm2d(d)
        self.deconv4 = nn.ConvTranspose2d(d, 1, 4, 2, 1) # 14x14 => 28x28

    def forward(self, input):
        # x = F.relu(self.deconv1(input))
        x = F.relu(self.deconv1_bn(self.deconv1(input)))
        x = F.relu(self.deconv2_bn(self.deconv2(x)))
        x = F.relu(self.deconv3_bn(self.deconv3(x)))
        x = torch.tanh(self.deconv4(x))

        return x

class Discriminator(nn.Module):
    def __init__(self, d=128):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(1, d, 4, 2, 1)
        self.conv2 = nn.Conv2d(d, d*2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(d*2)
        self.conv3 = nn.Conv2d(d*2, d*4, 4, 1, 0)
        self.conv3_bn = nn.BatchNorm2d(d*4)
        self.conv4 = nn.Conv2d(d*4, 1, 4, 1, 0)

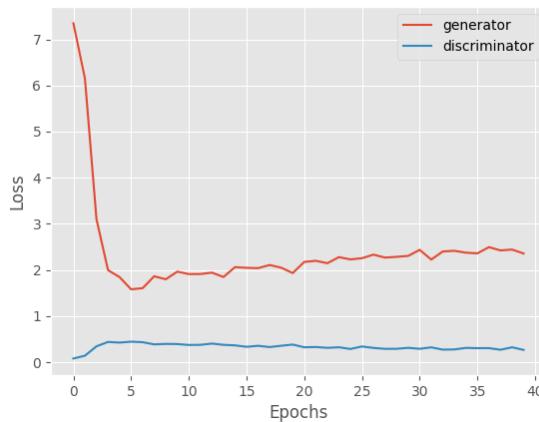
    def forward(self, input):
        x = F.leaky_relu(self.conv1(input), 0.2)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = torch.sigmoid(self.conv4(x))

        return x

```

b. Hyperparameter tuning & Analysis

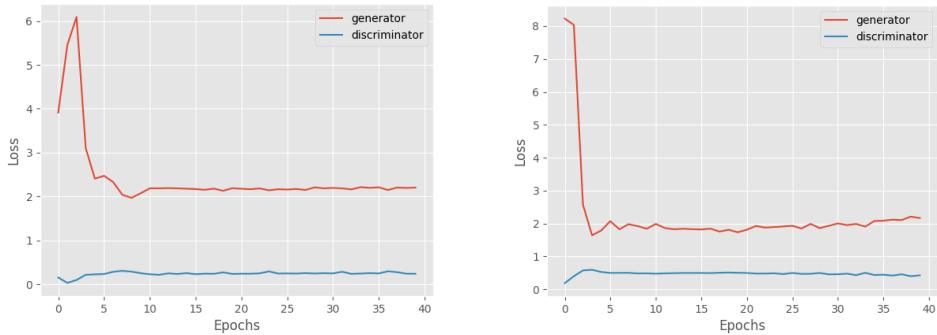
Default hyperparameter 는 lr = 2e-4, batch_size = 512, latent_dims = 100 이다. 먼저, 아래는 40 epoch 의 학습 과정에서 동일한 latent vector 에 대해 각각 0, 20, 40 epoch 에서의 결과를 나타낸 그림이며, 아래의 그래프는 loss curve 이다. 약 5 epoch 부근까지 generator loss 가 급격하게 감소하면서 그 이후 두 loss function 모두 매우 stable 해지며 학습이 안정적으로 진행된 모습이다. (https://drive.google.com/file/d/1bqW77F8A_hRH-54ZWBy8tldpc8Fz--lu/view?usp=sharing)



이후 진행된 hyperparameter tuning 에서도 대부분의 hyperparameter 에서 비슷한 형태의 loss curve 와 좋은 결과물이 생성되었으며, 시도해본 hyperparameter 는 다음과 같다.

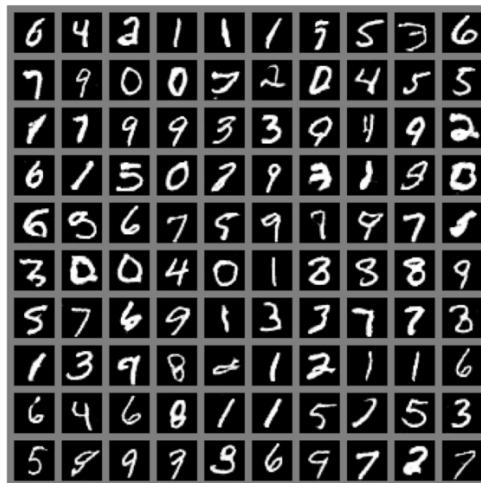
lr	batch_size	latent_dims	Description
2e-4	512	100	Good
2e-4	256	100	Similar loss curve to default, also good
2e-4	128	100	Good, Generator 의 loss 가 매우 빠르게 증가하기 시작, loss 값도 더 큼
2e-4	1024	100	Good, Loss 값이 떨어진 후 거의 수렴
2e-4	512	64	Best 이지만 사실 다른 모델들과 거의 유사, 시각화된 결과로 판단
2e-4	512	200	Good
1e-4	512	100	Good
1e-3	512	100	Good
5e-5	512	100	Good, 초반에 조금 unstable, 좀 더 느리게 수렴
5e-5	1024	100	Good, 초반에 조금 unstable, 좀 더 느리게 수렴

왼쪽과 오른쪽의 loss curve 는 각각 default hyperparameter 에서 learning rate 를 $5e-5$ 와 $1e-3$ 으로 변경한 경우의 loss curve 인데, 왼쪽의 경우 loss curve 가 조금 더 최저점을 찍는 시점이 느려졌고 초반의 학습이 unstable 해졌으며, 오른쪽의 경우 최저점을 찍는 시점이 빨라진 것을 볼 수 있다.



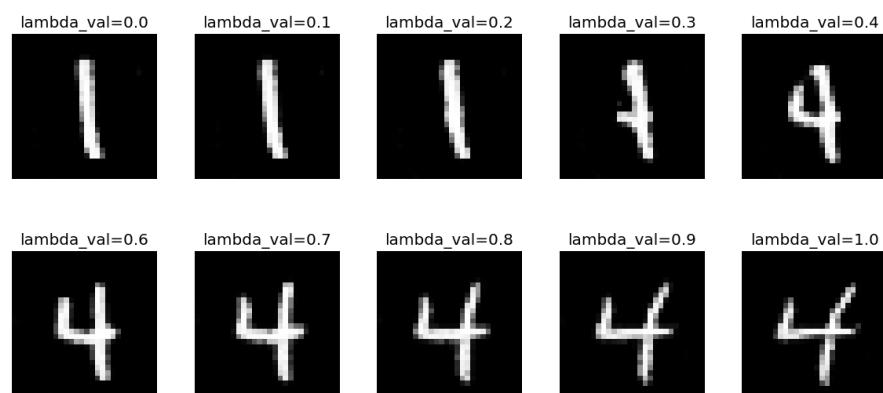
b. Visualize Generated Samples

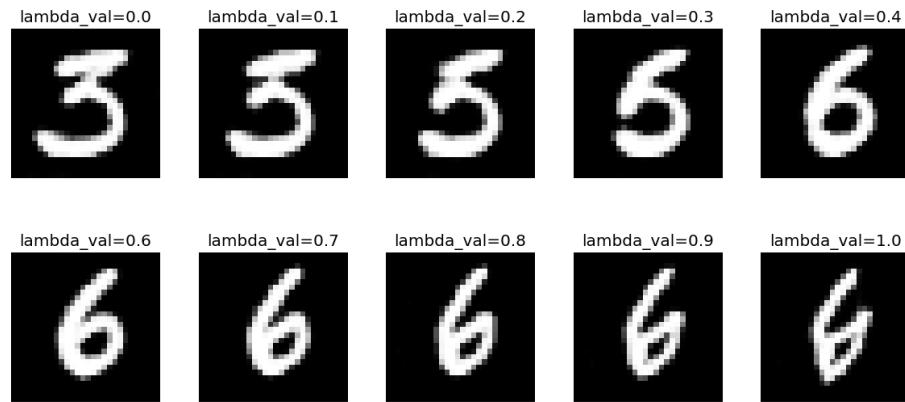
아래 그림은 임의의 latent vector 100 개로 최종 visualization 을 진행한 결과물이다.



c. Interpolation

아래 그림들은 latent space 상에서 두 개의 random 한 sample 에 대해 interpolation 을 진행한 결과물이다. lambda 값을 조금씩 조정함에 따라 lambda 값이 0 일때의 숫자에서 1 일때의 숫자로 점차 변화해가는 듯한 모습을 볼 수 있다.





d. Visualize generated images using the same latent vectors across epochs

아래 그림들은 40 epoch 동안 학습이 진행되면서 생성한 동일한 latent vector에 대한 generated image를 visualization한 결과물이다. 각 image는 0, 10, 20, 30, 40 epoch의 학습 이후의 generated image이다. imageio 라이브러리를 활용해 gif로도 제작했다. (<https://drive.google.com/file/d/15FPb-yHhUjX6wbTgvJKEDfDK67eHiNci/view?usp=sharing>)



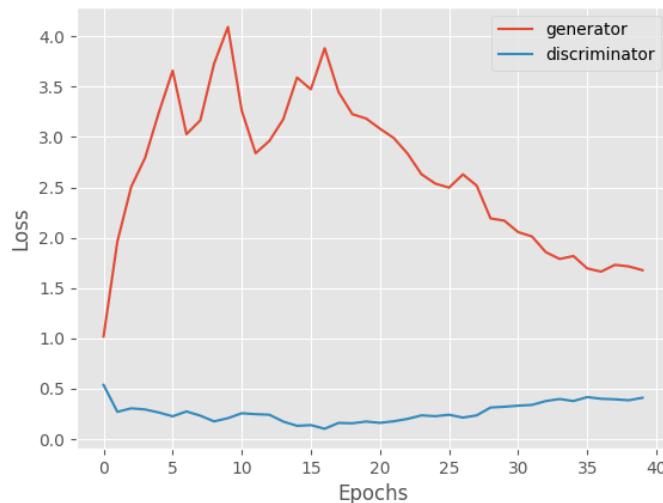
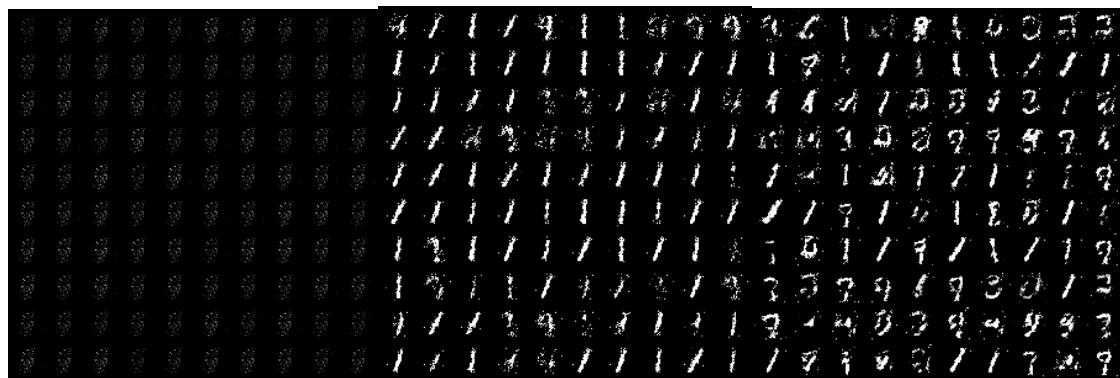
3. Learn a GAN (MLP)

a. Implementation

MLP로 구현한 GAN의 경우 MNIST의 original size인 28x28에 맞추어 구현하였으며, Generator와 Discriminator를 구현하는 과정에서 ConvTranspose2d 함수와 Conv2d 함수 대신 nn.Linear를 활용하였고 그에 맞추어 forward 함수까지 수정해주었다. 또한, Discriminator의 forward 과정에서 dropout을 추가하여 discriminator가 지나치게 분류를 잘해 학습이 원활하게 진행되지 않는 것을 방지해주었다. 해당 코드는 <https://debuggercafe.com/vanilla-gan-pytorch/>의 코드를 참고하였다.

b. Hyperparameter tuning & Analysis

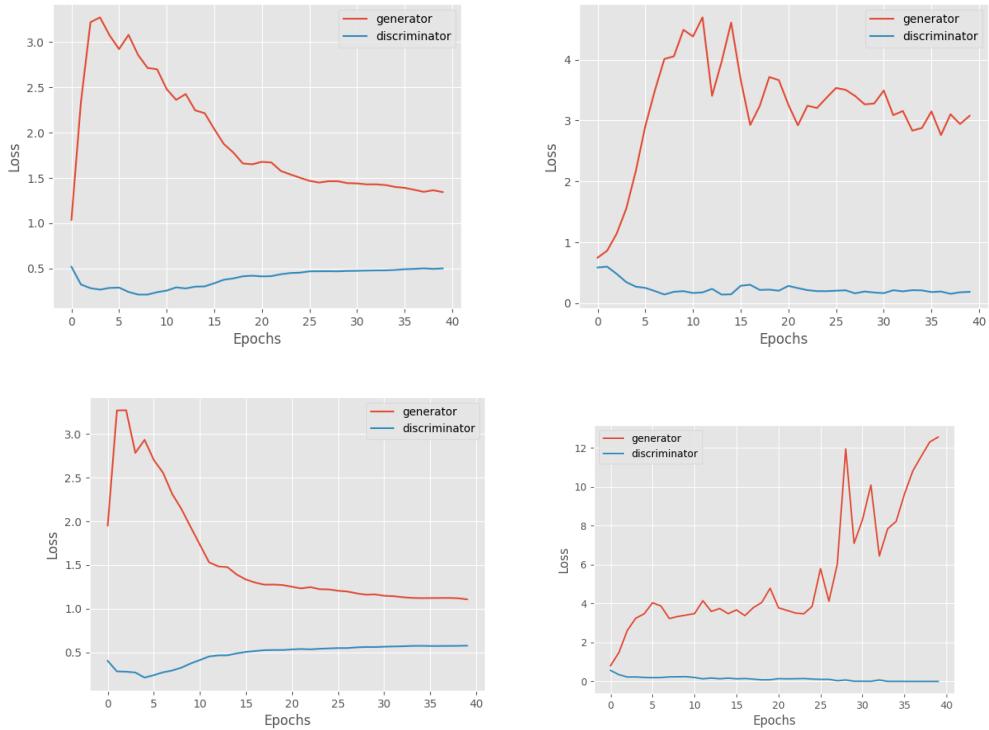
Default hyperparameter는 lr = 2e-4, batch_size = 512, latent_dims = 100이다. 먼저, 아래는 40 epoch의 학습 과정에서 동일한 latent vector에 대해 각각 0, 20, 40 epoch에서의 결과를 나타낸 그림이며, 아래의 그래프는 loss curve이다. 학습 초반부, discriminator loss는 감소하지만 generator loss가 급격하게 증가하면서 매우 불안정하게 진동함을 확인할 수 있고 이에 따라 mode collapse와 같이 1과 비슷한 형태만 generate하는 것을 확인할 수 있다. 점차 generator loss가 감소하고 discriminator loss가 조금씩 증가하면서 1이 아닌 다른 숫자의 형태도 조금씩 만들어내는 것을 확인할 수 있고, 이를 통해 epoch 수가 40보다 더 많이 진행되었더라면 좀 더 좋은 결과를 얻을 수 있지 않았을까 유추해볼 수 있다.
(<https://drive.google.com/file/d/1P8DDl15ySHpVHGnOgN2K-qa8RefiMAd/view?usp=sharing>)



해당 결과를 참고하여 다음과 같이 hyperparameter 값을 바꿔가며 hyperparameter tuning 과정을 진행하였다.

lr	batch_size	latent_dims	Description
2e-4	512	100	Default, bad result, mode collapse
2e-4	256	100	Similar but more stable than default
2e-4	1024	100	Mode collapse, unstable, converge to high loss
2e-4	128	100	Stable, converge to low loss
2e-4	512	200	Similar but more stable than default
2e-4	512	64	Unstable, increasing generator loss
1e-4	512	100	Mode collapse, unstable
1e-3	512	100	Similar to default but faster
5e-5	512	100	Mode collapse, increasing generator loss

아래 4 개의 loss curve 는 각각 default setting 에서 batch size 를 256, 1024, 128 로 바꿨을 때의 loss curve 와 latent_dims 를 64 로 바꿨을 때의 loss curve 이다. Batch size 를 감소시켰을 때 loss 가 좀 더 stable 한 양상을 띠고 있다. 특히, 1024 의 경우 discriminator loss 는 0 에 가까운 값이면서 generator loss 가 감소하지 않고 진동하면서 3 정도의 높은 값에 수렴하는 것을 볼 수 있다. 그에 비해 256 의 경우 generator loss 가 점차 감소하면서 discriminator loss 가 점차 증가하는 모습을 보이고 있으며, 128 은 이에 더불어 값이 감소하며 거의 수렴하는 형태를 띠고 있어 256 보다 훨씬 더 안정적으로 모델이 학습되고 있음을 유추해볼 수 있다. latent_dims 가 200 으로 증가한 경우에는 첫 번째와 거의 유사한 형태의 loss curve 를 보였고, 64 로 감소한 경우 discriminator loss 가 0 에 도달하면서 generator loss 가 큰 폭으로 증가하며 mode collapse 현상이 발생하였다. 이에 가장 안정적인 학습이 이루어진 hyperparameter 인 (2e-4, 128, 100)을 택했다. 실제로 해당 hyperparameter 로 100 epoch 까지 학습을 시켜본 결과 loss 가 40 epoch 에서의 값과 거의 비슷한 값으로 수렴해 일정한 값을 유지하는 것을 볼 수 있었다.



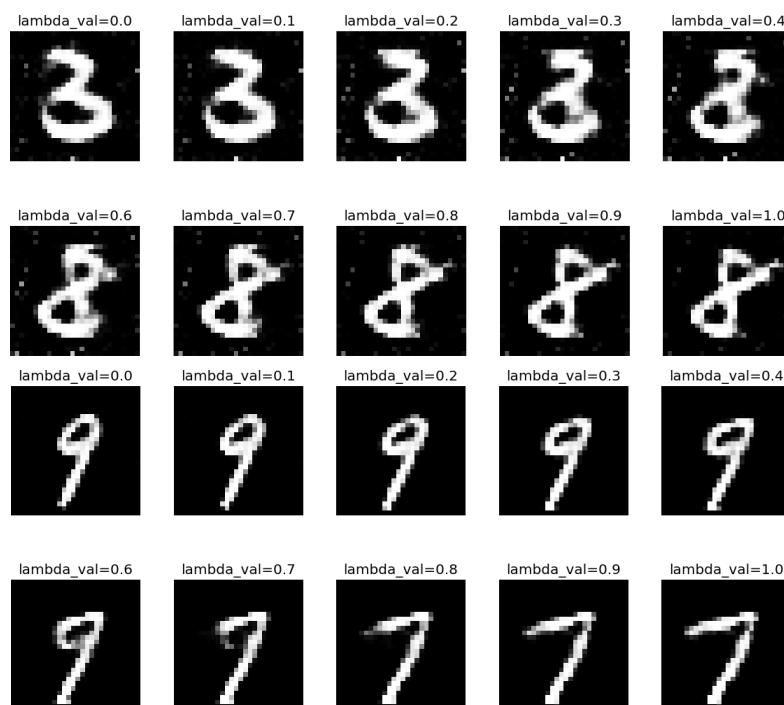
b. Visualize Generated Samples

아래 그림은 임의의 latent vector 100 개로 최종 visualization 을 진행한 결과물이다.



c. Interpolation

아래 그림들은 latent space 상에서 두 개의 random 한 sample 에 대해 interpolation 을 진행한 결과물이다. lambda 값은 조금씩 조정함에 따라 lambda 값이 0 일때의 숫자에서 1일때의 숫자로 점차 변화해가는 듯한 모습을 볼 수 있다.



d. Visualize generated images using the same latent vectors across epochs

아래 그림들은 40 epoch 동안 학습이 진행되면서 생성한 동일한 latent vector에 대한 generated image를 visualization한 결과물이다. 각 image는 0, 10, 20, 30, 40 epoch의 학습 이후의 generated image이다. imageio 라이브러리를 활용해 gif로도 제작했다. (https://drive.google.com/file/d/1VO3_utaSrAQeGeQ60GPa-ap5G2brmZg2/view?usp=sharing)



4. InfoGAN

a. Implementation

해당 64x64 InfoGAN 코드의 구현은 <https://github.com/Natsu6767/InfoGAN-PyTorch> 의 28x28 InfoGAN 구현 코드를 참고하였다. InfoGAN을 구현하기 위해 먼저 generator와 discriminator의 구현을 변경하였다.

Generator의 경우 latent vector의 dimension에 추가로 c1, c2, c3의 dimension 총합인 10+1+1=12를 더해주었고, 1번에서 구현했던 64x64 deconv GAN의 구현에 맞춰 nn.ConvTranspose2d 함수의 parameter를 조정해주었다.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.deconv1 = nn.ConvTranspose2d(latent_dims + 12, 1024, 4, 1, 0)
        self.deconv1_bn = nn.BatchNorm2d(1024)
        self.deconv2 = nn.ConvTranspose2d(1024, 256, 4, 2, 1)
        self.deconv2_bn = nn.BatchNorm2d(256)
        self.deconv3 = nn.ConvTranspose2d(256, 128, 4, 2, 1)
        self.deconv3_bn = nn.BatchNorm2d(128)
        self.deconv4 = nn.ConvTranspose2d(128, 64, 4, 2, 1)
        self.deconv4_bn = nn.BatchNorm2d(64)
        self.deconv5 = nn.ConvTranspose2d(64, 1, 4, 2, 1)

    def forward(self, input):
        x = F.relu(self.deconv1_bn(self.deconv1(input)))
        x = F.relu(self.deconv2_bn(self.deconv2(x)))
        x = F.relu(self.deconv3_bn(self.deconv3(x)))
        x = F.relu(self.deconv4_bn(self.deconv4(x)))
        x = torch.tanh(self.deconv5(x))

    return x
```

Discriminator 의 경우 generator 에서 정의한 nn.ConvTranspose2d 함수의 parameter 에 맞춰 Conv2d 의 parameter 를 조정해주었고, InfoGAN 논문에 따라 Leaky ReLU 함수의 parameter 를 0.1로 조정해주었다.

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 4, 2, 1)
        self.conv2 = nn.Conv2d(64, 128, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(256)
        self.conv4 = nn.Conv2d(256, 1024, 8, 1, 0)
        self.conv4_bn = nn.BatchNorm2d(1024)

    def forward(self, input):
        x = F.leaky_relu(self.conv1(input), 0.1)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.1)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.1)
        x = F.leaky_relu(self.conv4_bn(self.conv4(x)), 0.1)

    return x
```

또한, InfoGAN 에서는 code 들의 discriminator network 인 Q network 를 구현해야한다. 이 Q network 는 discriminator D 와 대부분의 architecture 를 공유하고 있고, 오직 head 부분만 차이가 있기 때문에 공통 부분을 Discriminator class 로 정의하고 D 와 Q 각각의 head 를 DHead 와 QHead 라는 class 로 구현해주었다. D 의 head 의 경우 일반 discriminator 와 동일하게 convolution layer 를 한 번 거친 이후 sigmoid function 으로 처리해주었고 Q 의 head 의 경우 논문의 구현과 유사하게 convolution layer 와 batch normalization 을 한 번씩 수행하고 Leaky ReLU 를 거쳐서 FC layer 를 통해 최종적으로 0~9 까지의 discrete 한 class 인 c1 을 나타내는 10x1 logit vector 와 c2 / c3 를 나타내는 Gaussian distribution 의 mean 과 standard deviation 2x1 vector 를 얻게 된다. 논문에서의 구현에 맞추어 standard deviation 엔 exp 함수를 적용시켜 양수 값만 나오도록 구현한다.

```
class DHead(nn.Module):
    def __init__(self):
        super(DHead, self).__init__()
        self.conv = nn.Conv2d(1024, 1, 1, 1, 0)

    def forward(self, input):
        return torch.sigmoid(self.conv(input))

class QHead(nn.Module):
    def __init__(self):
        super(QHead, self).__init__()
        self.conv1 = nn.Conv2d(1024, 64, 1, 1, 0)
        self.conv1_bn = nn.BatchNorm2d(64)

        self.conv_discrete = nn.Conv2d(64, 10, 1) # 10 discrete units (c1)
        self.conv_mean = nn.Conv2d(64, 2, 1) # mean of Gaussian for continuous codes (c2, c3)
        self.conv_stdev = nn.Conv2d(64, 2, 1) # standard deviation of Gaussian for continuous codes (c2, c3)

    def forward(self, input):
        x = F.leaky_relu(self.conv1_bn(self.conv1(input)), 0.1)

        discrete_values = self.conv_discrete(x).squeeze()
        mean = self.conv_mean(x).squeeze()
        exp_stdev = torch.exp(self.conv_stdev(x).squeeze())

    return discrete_values, mean, exp_stdev
```

InfoGAN 의 loss function 과 optimization 과정에 맞게 generator 의 parameter 와 함께 Q network 의 head 에 해당하는 부분의 parameter 를 선언하고, discriminator 의 parameter 와 함께 discriminator head 에 해당하는 부분의 parameter 를 선언해주어야 한다. 또한, 구현 과정에서 논문과 같은 형태로 generator 와 discriminator 의 learning rate 를 따로 설정해주었다.

```

gen_optimizer = torch.optim.Adam(params=[{'params': generator.parameters()}, {'params': recognition_head.parameters()}], lr=learning_rate_gen, betas=(
disc_optimizer = torch.optim.Adam(params=[{'params': discriminator.parameters()}, {'params': discriminator_head.parameters()}], lr=learning_rate_disc,
# set to training mode
generator.train()
discriminator.train()

```

$$\min_{G,Q} \max_D V_{\text{InfoGAN}}(D, G, Q) = V(D, G) - \lambda L_I(G, Q)$$

추후 visualization 을 위한 고정된 latent vector 를 만들기 위해 다음과 같이 latent vector 를 만드는 코드를 수정한다. 100 개의 latent vector 에 0~9 까지를 나타내는 10x1 one-hot vector 를 10 개씩 총 100 개를 생성하여 concat 하고, c2 와 c3 의 경우 각각 Unif(-1, 1)로부터 sampling 하여 random 한 값을 부여해 concat 해 100 x (latent_dims + 12) 차원의 latent vector 를 만들어낸다.

```

gen_latent = torch.randn(100, latent_dims, 1, 1, device=device)

# creating c1
idx = np.arange(10).repeat(10)
c1 = torch.zeros(100, 1, 10, device=device)
c1[torch.arange(0, 100), 0, idx] = 1.0
c1 = c1.view(100, -1, 1, 1)
print(c1.shape)

gen_latent = torch.cat((gen_latent, c1), dim=1)

# creating c2 and c3 ~ Unif(-1,1)
c2 = torch.rand(100, 1, 1, 1, device=device) * 2 - 1
c3 = torch.rand(100, 1, 1, 1, device=device) * 2 - 1

print(c2.shape, c3.shape)

gen_latent = torch.cat((gen_latent, c2), dim=1)
gen_latent = torch.cat((gen_latent, c3), dim=1)

```

학습 과정에서도 마찬가지로 batch size 에 맞추어 latent vector 에 random 한 10x1 one-hot vector 를 생성하여 concat 하고, c2 와 c3 의 경우 각각 Unif(-1, 1)로부터 sampling 하여 random 한 값을 부여해 concat 해 (batch_size) x (latent_dims + 12) 차원의 latent vector 를 만들어낸다.

```

# c1 vector
c1 = torch.zeros(batch_size, 1, 10, device=device)
idx = np.random.randint(10, size=batch_size)
c1[torch.arange(0, batch_size), 0, idx] = 1.0
c1 = c1.view(batch_size, -1, 1, 1)
latent = torch.cat((latent, c1), dim=1)

# c2/c3 vector
c2 = torch.rand(batch_size, 1, 1, 1, device=device) * 2 - 1
latent = torch.cat((latent, c2), dim=1)

c3 = torch.rand(batch_size, 1, 1, 1, device=device) * 2 - 1
latent = torch.cat((latent, c3), dim=1)

fake_image_batch = generator(latent)

```

이후 real image 와 fake image 를 discriminator 와 discriminator 의 head model 에 넣어 binary CE loss 를 통해 discriminator의 loss를 계산하여 discriminator를 학습시킨다. 그리고 Q network의 학습을 위해 target index와 Q head를 통해 계산된 logit 의 CE loss 를 계산하여 c1에 대한 loss 를 계산하고, Unif(-1, 1)에서 sampling 된 c2, c3 의 값과 Q head를 통해 계산된 c2 와 c3 의 Gaussian distribution 의 mean 과 standard deviation 값 사이의 Gaussian NLL loss 를 이용하여 c2 와 c3 에 대한 loss 를 계산한다. 이후 fake image 에 대한 loss 와 latent code 들에 대한 loss 를 모두 더해 generator 의 loss 를 계산하고 generator 를 학습시킨다.

```
real_pred = discriminator_head(discriminator(image_batch)).squeeze()
fake_pred = discriminator_head(discriminator(fake_image_batch.detach())).squeeze()
disc_loss = F.binary_cross_entropy(real_pred, label_real) + F.binary_cross_entropy(fake_pred, label_fake)

disc_optimizer.zero_grad()
disc_loss.backward()
disc_optimizer.step()

# train generator to output an image that is classified as real

discriminator_output = discriminator(fake_image_batch)
fake_pred = discriminator_head(discriminator_output).squeeze()

# train recognition network (Q)
logits, mean, exp_var = recognition_head(discriminator_output)

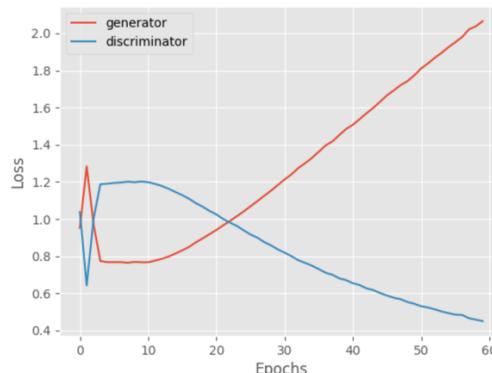
target = torch.LongTensor(idx).to(device)
discrete_loss_func = nn.CrossEntropyLoss()
discrete_loss = discrete_loss_func(logits[:, 0:10], target)

continuous_loss_func = nn.GaussianNLLLoss()
continuous_loss = continuous_loss_func(latent[:, (latent_dims + 10):].view(-1, 2), mean, exp_var) * 0.1

gen_loss = F.binary_cross_entropy(fake_pred, label_real) + discrete_loss + continuous_loss
```

InfoGAN 학습을 위한 hyperparameter는 다음과 같았다. 논문에서는 28x28로 구현했기 때문에 완벽히 hyperparameter가 같을 수 없었고, 처음 논문과 완전히 동일한 hyperparameter 로 학습시켰을 때 discriminator 의 loss 가 너무 낮아 학습이 잘 진행되지 않았다. 이를 막기 위해, 논문에서의 latent dimension 이었던 62 를 50 으로 줄이고, 2e-4 였던 discriminator 의 learning rate 를 1e-5 로 줄여서 학습을 진행했다. 아래의 loss curve 는 해당 setting 으로 60 epoch 학습을 진행했을 때의 Loss curve 인데, discriminator loss 는 계속 감소하고 generator loss 는 계속 증가하는 형태로 학습이 안정적으로 진행되면서 최종적으로 좋은 결과물을 얻을 수 있었다.

```
[ ] latent_dims = 50 # dimension of noize vector (z)
num_epochs = 40
batch_size = 64
learning_rate_disc = 1e-5
learning_rate_gen = 1e-3
use_gpu = True
```



b. Reproduction of Figure 2 (a), (c), (d) in InfoGAN

--	--

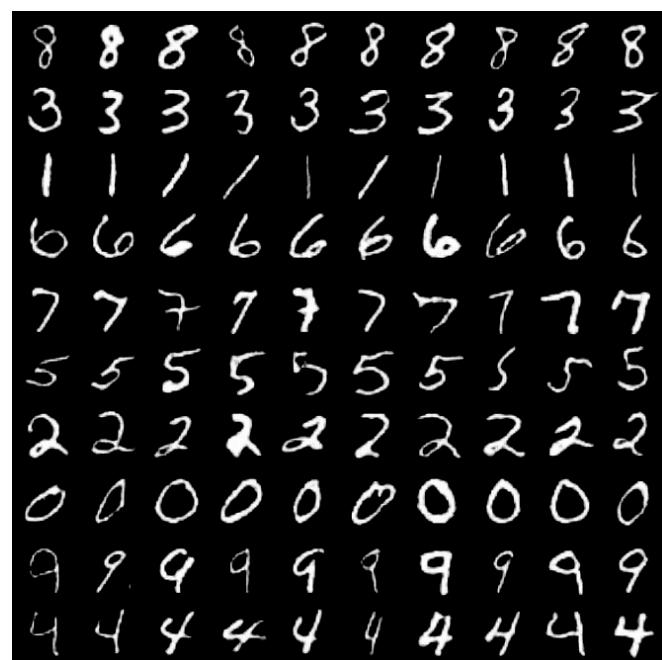
(a) Varying c_1 on InfoGAN (Digit type)(b) Varying c_1 on regular GAN (No clear meaning)

--	--

(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation)(d) Varying c_3 from -2 to 2 on InfoGAN (Width)

위 그림은 InfoGAN 논문에 수록된 Figure 2로 c_1, c_2, c_3 를 변경시킴에 따라 model 0이 generate 하는 image의 number type, rotation, width 가 바뀌는 것을 보여주는 그림이다.

이를 reproduce 하기 위해 먼저 c_1 을 변경해보자. 우리는 학습 과정에서 어떤 one-hot vector 가 어떤 digit type에 배정되었는지 알 수 없기 때문에 visualization을 통해 이를 유추해야 한다. 사전에 선언했던 fixed latent code를 통해 visualization을 해본 결과, 다음과 같은 image를 얻을 수 있다. 이에 따라 우리는 $[1, 0, 0, \dots, 0]$ 이 8, $[0, 1, 0, \dots, 0]$ 이 3에 배정되었다는 사실을 알 수 있고, 이러한 방식으로 나머지 숫자들도 유추해보면 10개의 one-hot vector가 각각 어떤 숫자에 대응되는지를 알 수 있다.



이를 통해 다음과 같은 방식으로 index 를 만들어 50 개의 one-hot vector 를 선언하고 같은 column 에는 같은 숫자가 나오도록 c1 을 선언하고 c2, c3 는 random 한 값을 sampling 하여 같은 row 에서는 같은 값을 갖게 만들면 아래와 같이 Figure 2 (a)와 매우 유사한 그림을 만들어낼 수 있다. 심지어 원래 논문에서는 7 의 중심부에 가로 선이 있는 경우 9 와 잘 구분하지 못했는데, 해당 모델의 경우 그러한 경우도 7 로 잘 구분해내는 것을 볼 수 있다.

```
# figure (a) : changing c1

latent = torch.randn(50, latent_dims, 1, 1, device=device)
print(latent.shape)

idx = np.tile(np.array([7, 2, 6, 1, 9, 5, 3, 4, 0, 8]), 5)
c1 = torch.zeros(50, 1, 10, device=device)
c1[torch.arange(0, 50), 0, idx] = 1.0
c1 = c1.view(50, -1, 1, 1)
print(c1.shape)

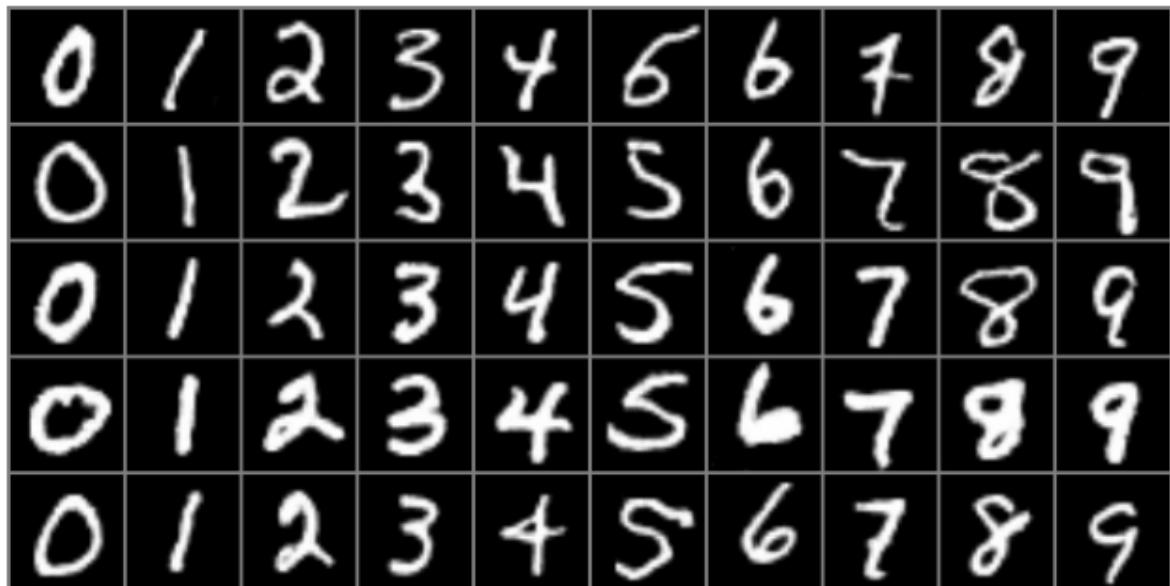
latent = torch.cat((latent, c1), dim=1)

# same c2 and c3 for a row
c2_c3 = np.repeat(torch.rand(5, 2, 1, 1, device=device) * 2 - 1, 10, axis = 0)
print(c2_c3.shape)

latent = torch.cat((latent, c2_c3), dim=1)
print(latent.shape)
print(latent.dtype)

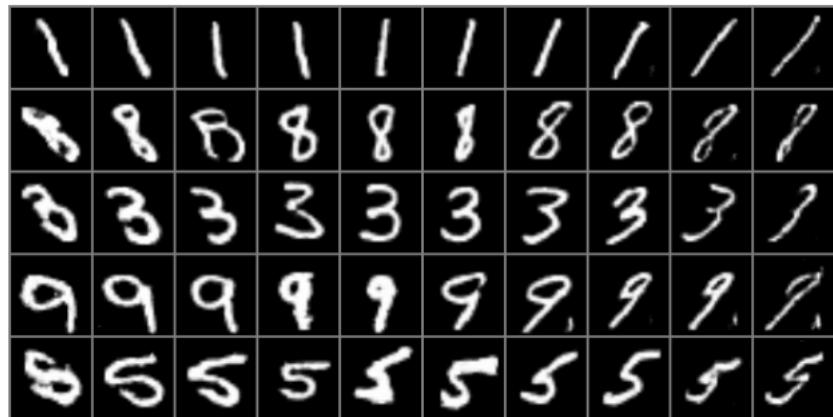
fake_image_batch = generator(latent).cpu()

fig, ax = plt.subplots(figsize=(8, 8))
show_image(make_grid(fake_image_batch.data[:50], 10))
ax.axis('off')
plt.show()
```

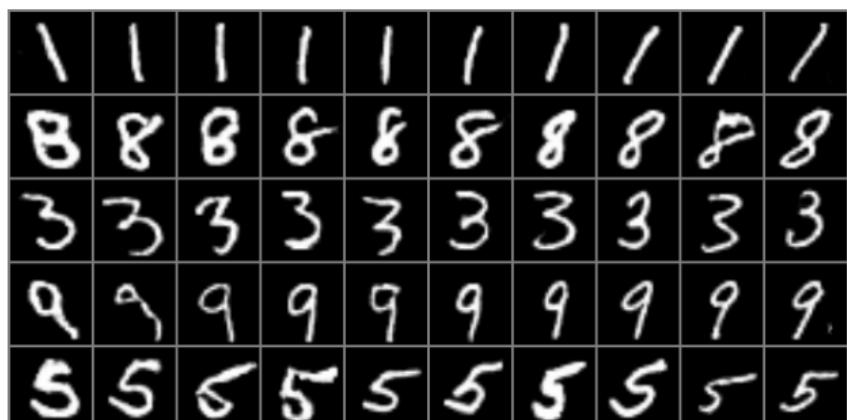


(Reproduced version of Figure 2. (a))

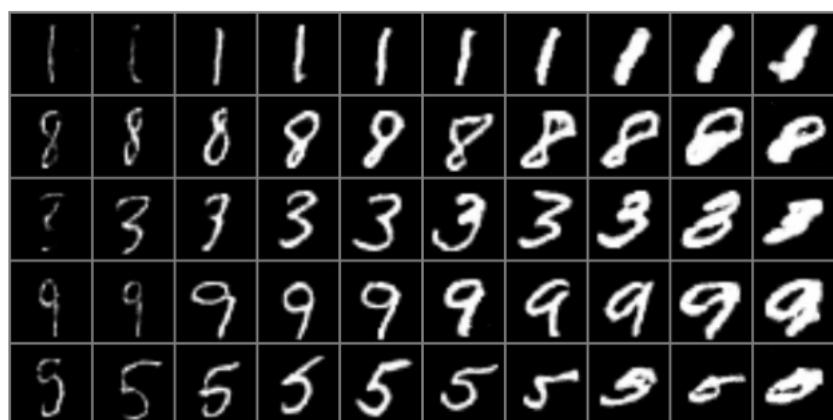
이와 비슷한 방식으로 c2 와 c3 를 -1 부터 1 까지의 범위 내에서 조정해보면 Figure 2.(c)와 (d)와 매우 유사한 그림을 얻을 수 있었다. 원래 논문에서는 generalization 을 위해 -2 부터 2 까지의 범위 내에서 균등하게 조정하는데, 논문 image 의 resolution 은 28x28 이라 64x64 기준으로 학습된 이 모델에서는 숫자 형태가 조금 이상해지는 경우가 있었다. 하지만 c2와 c3 자체의 특성인 rotation과 width는 매우 잘 나타내는 것을 볼 수 있기 때문에, c2와 c3 가 각각 rotation과 width 를 나타내도록 잘 학습되었다는 것을 알 수 있다.



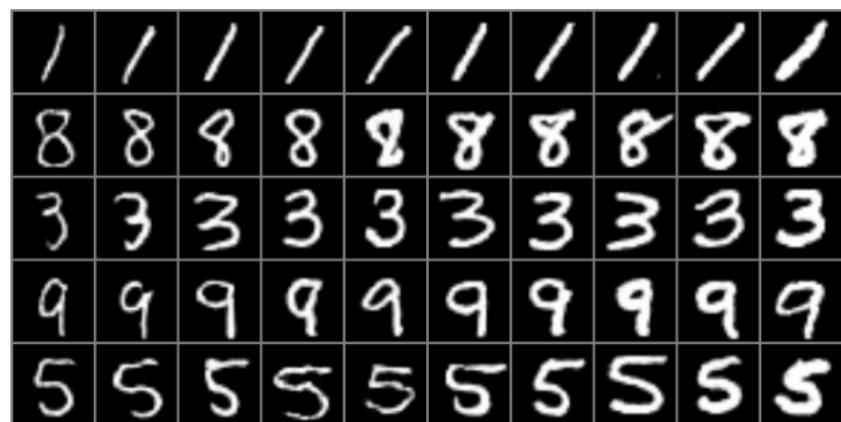
(Reproduced version of Figure 2. (c), from -2 to 2)



(Reproduced version of Figure 2. (c), from -1 to 1)



(Reproduced version of Figure 2. (d), from -2 to 2)



(Reproduced version of Figure 2. (d), from -1 to 1)