Lab4: InfoGAN

Department of Computer Science, NCTU

TA: Alan(李懿倫)、Jonson(黃正昕)

Important Date

- Experiment Report Submission Deadline: 12/29 11:59 a.m
- Demo date: 12/29
- Zip all files in one file
  - Report (.pdf)
  - Source code (.py)
- Name it like 「DLP_LAB4_yourID_name.zip」
  - ex: DLP_LAB4_409551015_李懿倫.zip
  - Email it to jshuang.cs09@nycu.edu.tw with subject MTK_DLP_LAB4_yourID_name

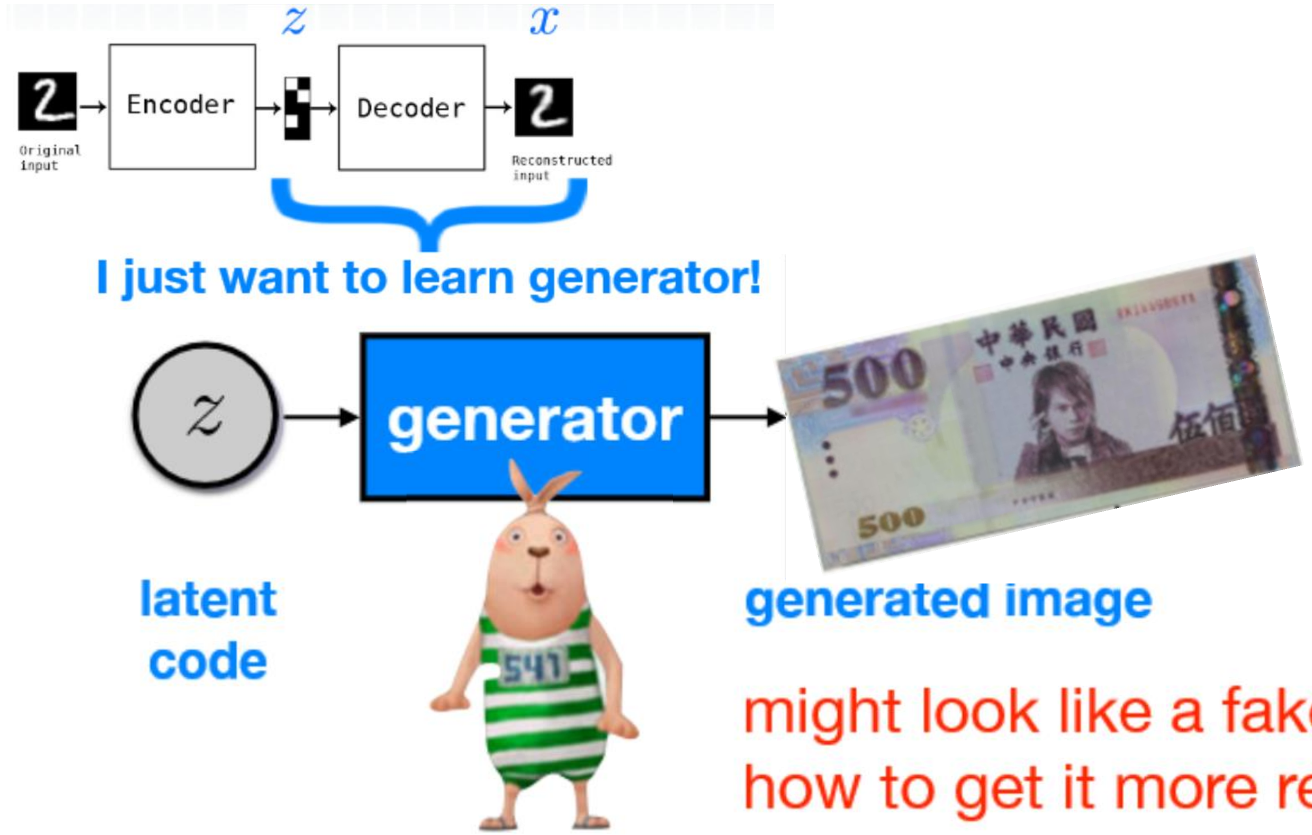
Lab Objective

- In this lab, you need to implement InfoGAN for generation of handwritten digit images.

- Handwritten digit images generation
  - Noise + fixed latent code -> generate handwritten images of the same digit but different appearance.

e.g. noise + fixed latent code of digit ‘0’ ->

GAN (Generative Adversarial Networks)



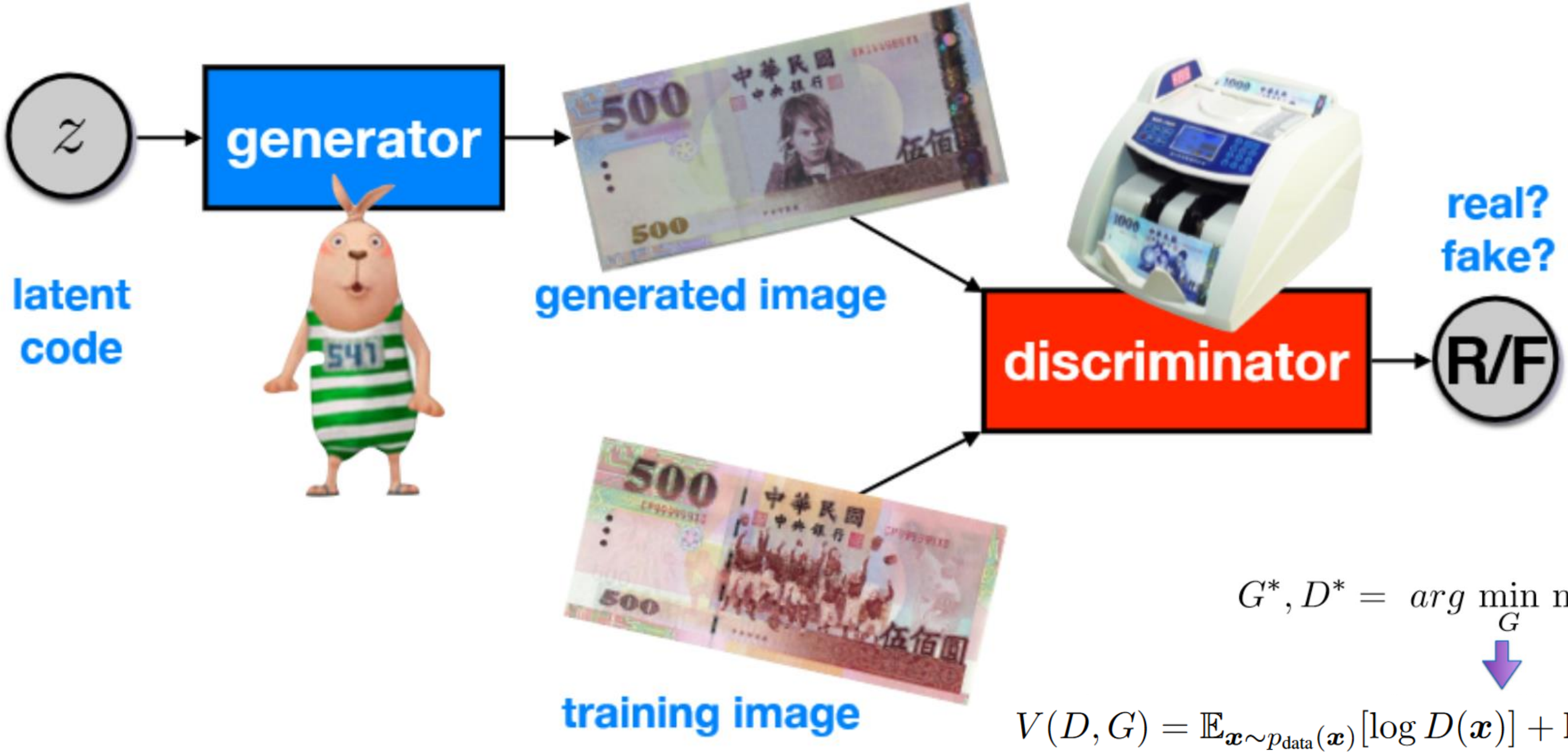
might look like a fake image,
how to get it more realistic?

☞impose **adversarial loss** on **data distribution**

GAN (Generative Adversarial Networks)

generator: try to generate more realistic images to cheat discriminator

discriminator: try to distinguish whether the image is generated or real



$$G^*, D^* = arg \min_G \max_D V(D, G)$$

$$V(D,G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Vanilla GAN Loss

Adversarial Loss

$$L_D = -\log\big(D(x)\big) - \log\big(1 - D\big(G(z)\big)\big)$$

$$\begin{cases} L_G = \log\big(1 - D\big(G(z)\big)\big) \\ L_G = -\log\big(D\big(G(z)\big)\big) \end{cases}$$

$$G^*, D^* = arg\ \min_G \max_D V(D, G)$$

$$\Downarrow$$

$$V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

GAN-Train the Discriminator

- Minimize $L_D = \boxed{-\log\big(D(x)\big)} - \log(1 - D\big(G(z)\big))$
  - Passing a batch of real data through D and calculate $\log\big(D(x)\big)$
  - nn.BCELoss()$(D(x), 1)$

```
## Train D with all-real batch
optimD.zero_grad()
# Get a batch of real data
real_data = data[0].to(device)
b_size = real_data.size(0)
label = torch.full((b_size,), real_label, device=device)
# Forward real batch to D
output = netD(real_data)
# Calculate loss on all-real batch
loss_real = criterionD(output, label)
# Calculate the gradients for real batch
loss_real.backward()
```

GAN-Train the Discriminator

- Minimize $L_D = -\log(D(x)) \boxed{-\log(1 - D(G(z)))}$
  - Sample a noise z and construct a fake data $G(z)$
  - Passing a batch of fake data through D and calculate $\log(1 - D(G(z)))$
  - nn.BCELoss()($D(G(z)), 0$)

```
## Train D with all-fake batch
# Generate batch of latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=device)
# Generate fake image batch with Generate
fake_data = netG(noise)
label.fill_(fake_label)
# Classify all fake batch with D
output = netD(fake_data.detach())
# Calculate D's loss on the all-fake batch
loss_fake = criterionD(output, label)
# Calculate the gradients for fake batch
loss_fake.backward()
# Update parameters
optimD.step()
```

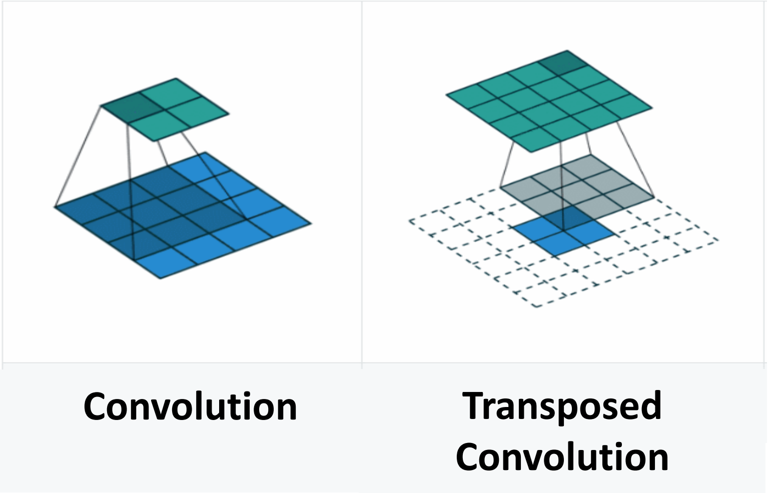
GAN-Train the Generator

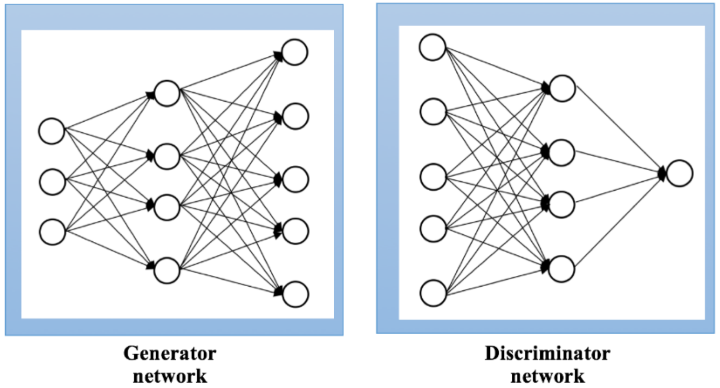
- Minimize $L_G = -\log(D(G(z)))$
  - Passing the fake data through D and calculate $\log(D(G(z)))$
  - nn.BCELoss()$(D(G(z)), 1)$

```
## Train G
optimG.zero_grad()
# label for generator should be real_label -> fooling D
label.fill_(real_label)
# Since we just updated D, perform another forward path of all-fake batch through D
output = netD(fake_data)
# Calculate G's loss based on this output
loss_G = criterionD(output, label)
# Calculate gradient for G
loss_G.backward()
# Update parameters
optimG.step()
```

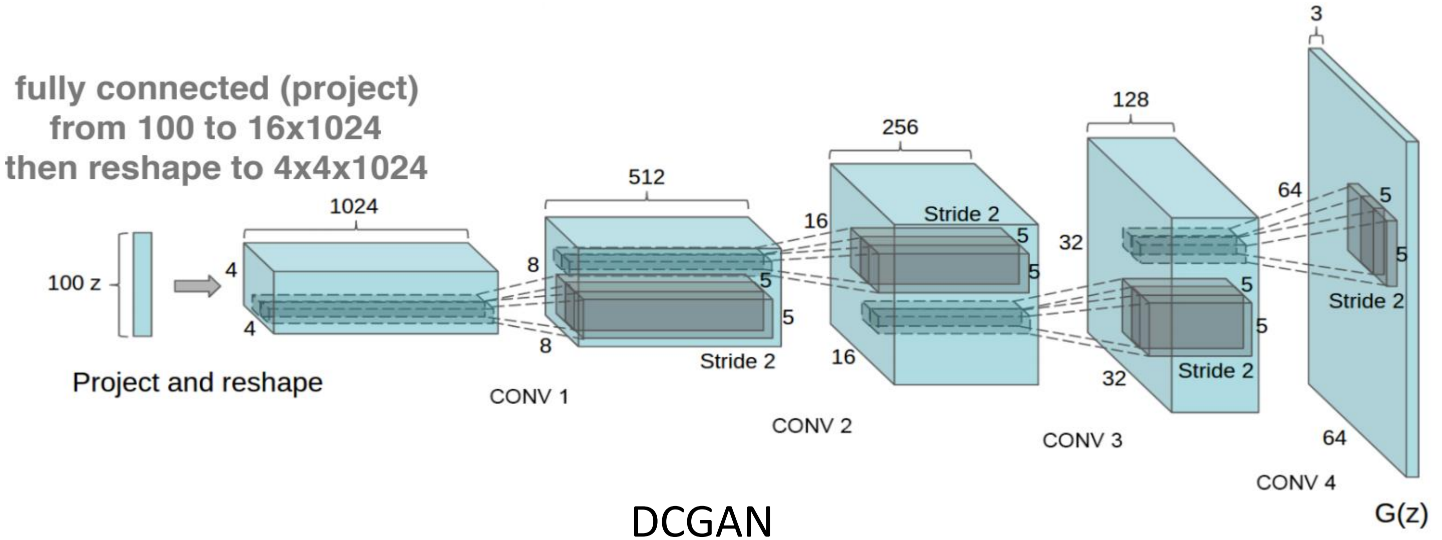
From GAN to DCGAN

- Replace fully connected layers with convolutions
- Use batch normalization after each layer





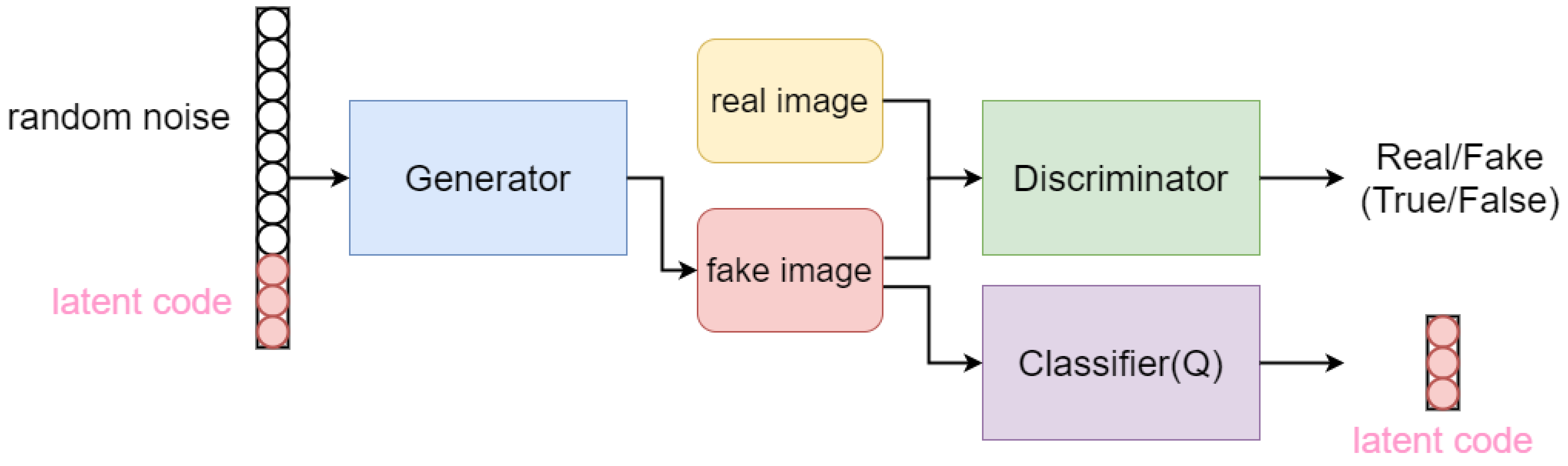
GAN



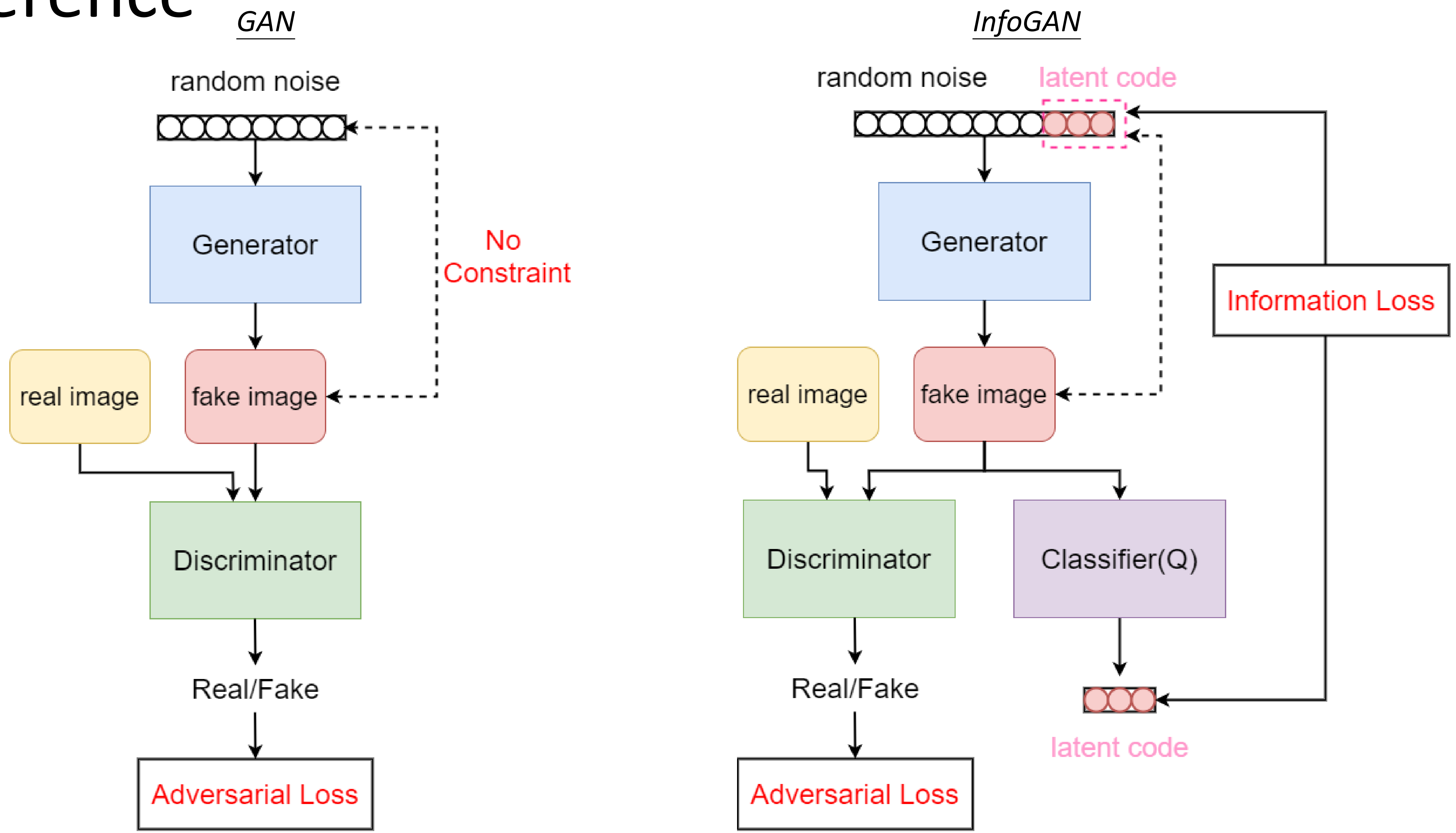
DCGAN

InfoGAN

- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets



Difference


GAN
random noise
Generator
No
Constraint
real image
fake image
Discriminator
Real/Fake
Adversarial Loss
InfoGAN
random noise
latent code
Generator
Information Loss
real image
fake image
Discriminator
Classifier(Q)
Real/Fake
latent code
Adversarial Loss

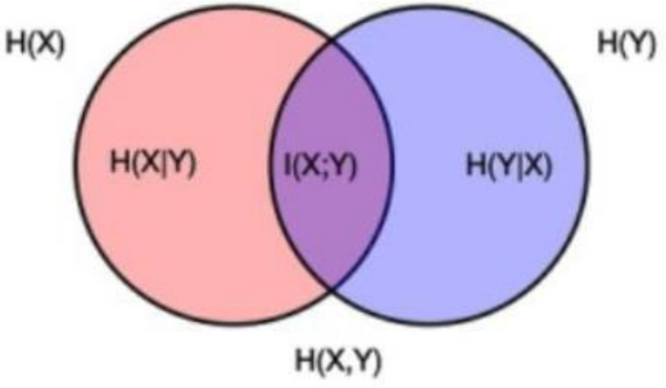
InfoGAN

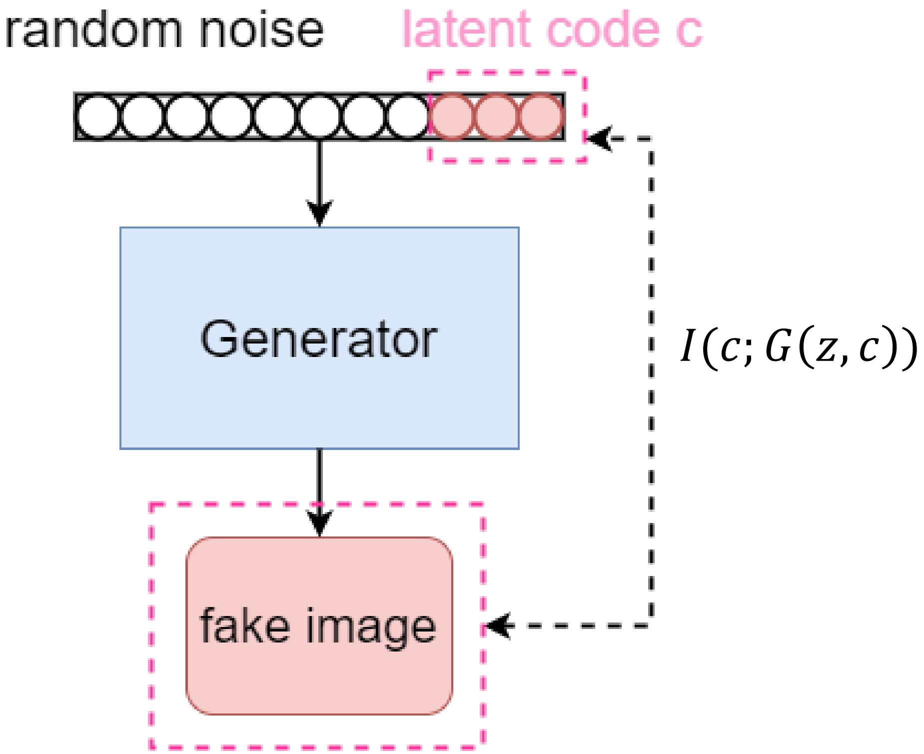
- Mutual Information:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

$$\boxed{\mathrm{H(Y)} = -\sum_{y} \log p(y) p(y)}$$

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$





InfoGAN-prof

$$I(c;G(z,c)) = H(c) - H(c|G(z,c))$$

$$= E_{x\sim G(z,c)}\left[E_{c'\sim P(c|x)}[logP(c'|x)]\right] + H(c)$$

$$= E_{x\sim G(z,c)}\left[D_{KL}(P(\cdot|x)||Q(\cdot|x)) + E_{c'\sim P(c|x)}[logQ(c'|x)]\right] + H(c)$$

$$\geq E_{x\sim G(z,c)}\left[E_{c'\sim P(c|x)}[logQ(c'|x)]\right] + H(c)$$

$$= \boldsymbol{E_{c\sim P(c),x\sim G(z,c)}[logQ(c|x)] + H(c)}$$

fixed value

$$L_I(G,Q) = c \cdot logQ\big(G(z,c)\big) + \boxed{H(c)}$$

$$maximize\ I\big(c;G(z,c)\big) = maximize\ L_I(G,Q) = maximize\ c \cdot \log Q\big(G(z,c)\big) = \boxed{minimize - c \cdot \log Q\big(G(z,c)\big)}$$

Cross Entropy Loss

InfoGAN Loss

Adversarial Loss

$$L_D = -\log\big(D(x)\big) - \log(1 - D\big(G(z)\big))$$

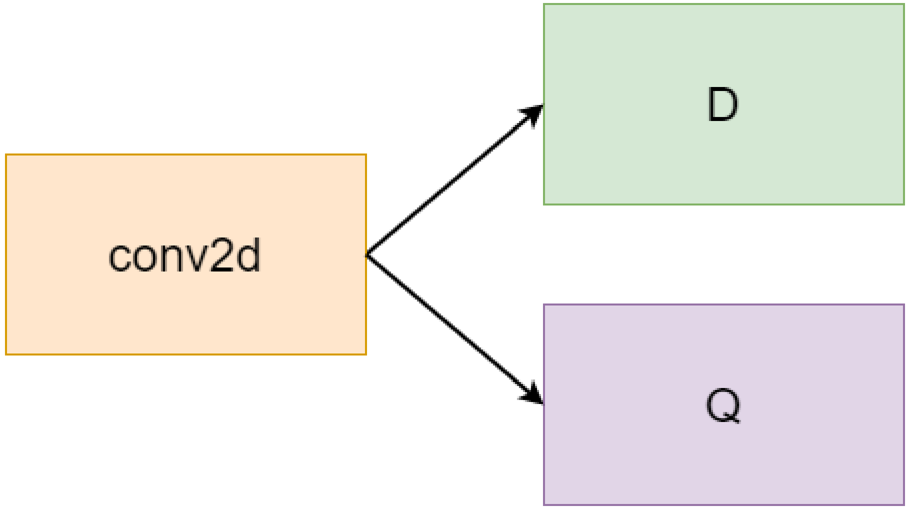
$$\begin{cases} L_G = \log(1 - D\big(G(z)\big)) \\ L_G = -\log(D\big(G(z)\big)) \end{cases}$$

Information Loss

$$L_I(Q, G) = -c \cdot \log(Q(G(z, c)))$$

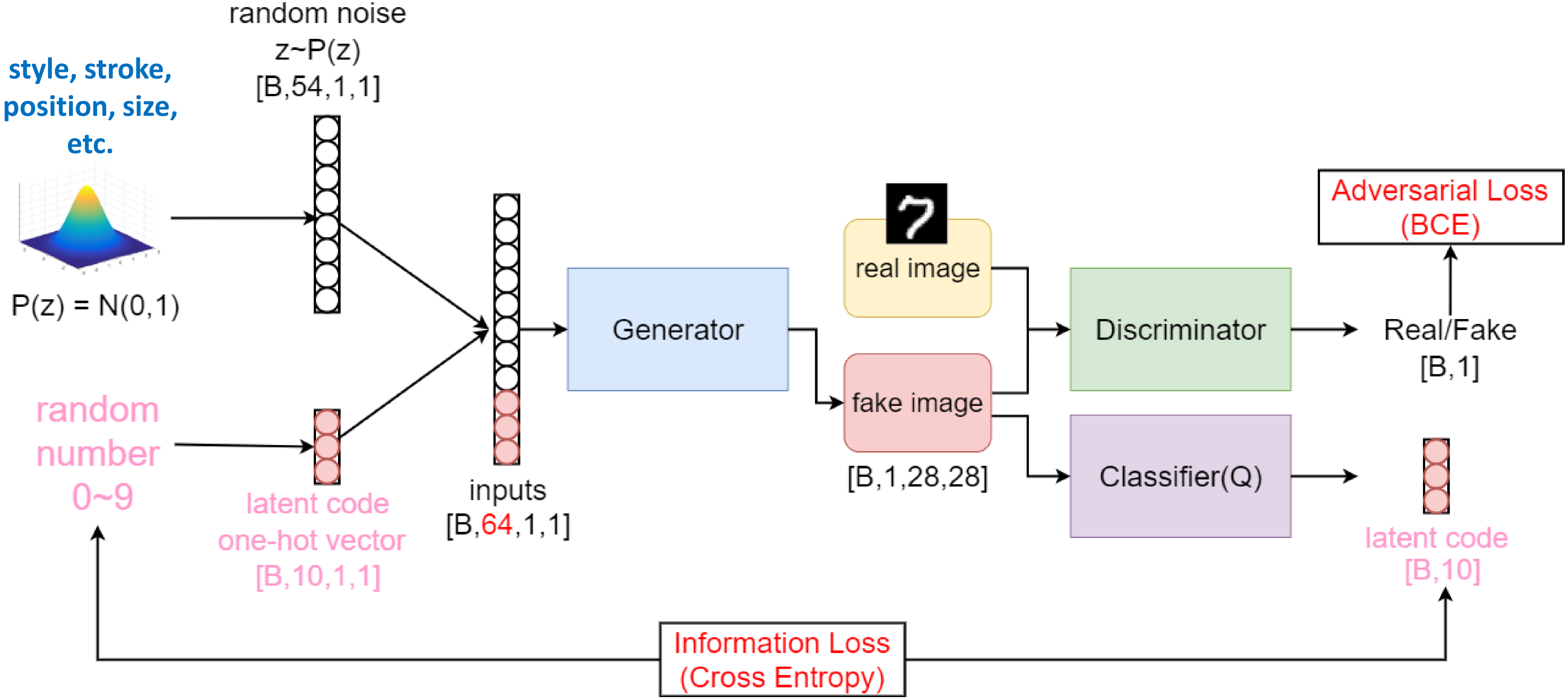
InfoGAN (share-layer)

- In practice, D and Q will share conv layers to share the information.




```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        # shared layer of discriminator
        self.main = nn.Sequential(
            #################
            #  To Do
            #################
        )

        # Dsicriminator branch
        self.D = nn.Sequential(
            #################
            #  To Do
            #################
        )

        # Info branch
        self.Q = nn.Sequential(
            #################
            #  To Do
            #################
        )


    def forward(self, x):
        h = self.main(x)
        real_or_fake = self.D(h).view(-1,1)
        info = self.Q(h).squeeze()

        return real_or_fake, info
```

InfoGAN-MNIST digits generation



Lab Requirement

- Modify the DCGAN architecture to InfoGAN.
  - Implement the model architecture in model.py
- Implement training procedure
  - Implement the training procedure in train.py
  - Adopt traditional generator and discriminator loss.
  - Maximize the mutual information between generated images and discrete one-hot vector.
- Show the generated images of digits.
- Plot the generator loss, discriminator loss and information loss during training.
- Implement the evaluation procedure.

Modify model.py

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(64, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.01)
    (6): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.01)
    (9): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): LeakyReLU(negative_slope=0.01)
    (12): ConvTranspose2d(64, 1, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (13): Sigmoid()
  )
)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.1, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.1, inplace)
    (5): Conv2d(128, 256, kernel_size=(7, 7), stride=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.1, inplace)
  )
  (D): Sequential(
    (0): Conv2d(256, 1, kernel_size=(1, 1), stride=(1, 1))
    (1): Sigmoid()
  )
)
```

```python
class Generator(nn.Module):
    def __init__(self, in_dim=64):
        super(Generator, self).__init__()

        # Generator
        self.main = nn.Sequential(
            #################
            #   To Do
            #################
        )
```

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        # shared layer of discriminator
        self.main = nn.Sequential(
            #################
            #   To Do
            #################
        )

        # Dsicriminator branch
        self.D = nn.Sequential(
            #################
            #   To Do
            #################
        )

        # Info branch
        self.Q = nn.Sequential(
            #################
            #   To Do
            #################
        )
```

Training procedure

- Update D network
  - Clear gradient
  - Real Loss
    - Input real data into D to get prediction
    - Calculate real_loss = $-\log(D(x))$ and backward
  - Fake Loss
    - Sample noise and generate one-hot vector, and concatenate them as the generator input
    - Pass the generator input into Generator to get fake data, and input fake data into D to get prediction
    - Calculate fake_loss = $-\log(1 - D(G(z)))$ and backward
  - Update parameters

Training procedure

- Update G and Q network
  - Clear gradient
  - Generator Loss
    - Input fake data into D again to get prediction and latent vector
    - Calculate G_loss = $-\log(D(G(z)))$
  - Information Loss
    - Calculate info_loss = $\max(L_I(G,Q))$ = $\mathrm{m}in(-c \cdot \log(Q(G(z,c))))$
  - Total Loss
    - Calculate loss = $G\_loss\ +\ \lambda \cdot info\_loss$ and backward
  - Update parameters

Hyper-parameters

- noise size = 54
- latent code size = 10
- total epochs = 50
- optimizer : Adam
- learning rate for G and Q: $1e^{-3}$
- learning rate for D: $2e^{-4}$
- Info loss weight: 0.25

You can adjust the hyper-parameters according to your own ideas.

Expected output

- 10 numbers (latent vectors) with 10 different styles (noises).

10 styles

10 numbers

Demo

- Please implement the evaluation procedure
  - Load pretrained model weights

```
# load model
checkpoint = torch.load(os.path.join(opt.output_str, 'checkpoint/model_{}.pt'.format(epoch)))
netG = checkpoint['netG']
netD = checkpoint['netD']
```

  - Generate output with 10 numbers x 10 styles

```
# generate data for evaluation procedure
fix_noise = torch.randn(10, opt.dim_noise, 1, 1, device=device).repeat_interleave(10, dim=0)
y = torch.arange(10).repeat(10)
y_cat = to_categorical(y, opt.dim_dis_latent)
val_data_inputs = torch.cat([fix_noise, y_cat], dim=1)
```

- While demo, you should run the evaluation and show your result to TA

Scoring Criteria

- Report (50%)
  - A. Introduction (10%)
  - B. Experiment Setups (20%)
    - How did you implement Info GAN
      - Adversarial loss
      - Maximizing mutual information
    - Which loss function of generator (refer to page.6) did you use? What’s the difference?
  - C. Results and discussion (20%)
    - Results of your samples
    - Training loss curves

Scoring Criteria

- Demo (50%)
  - Generate 10 numbers with 10 different styles (noises). A column is correct if there are ≥ 7 correct numbers in that column, and your total score is decided according to the total number of correct columns. (30%)
    - 10 correct columns ------ 100%
    - 9 correct columns ------ 90%
    - ≥ 7 correct columns ------ 80%
    - ≥ 5 correct columns ------ 60%
    - Otherwise ------ 0%
  - Questions (20%)