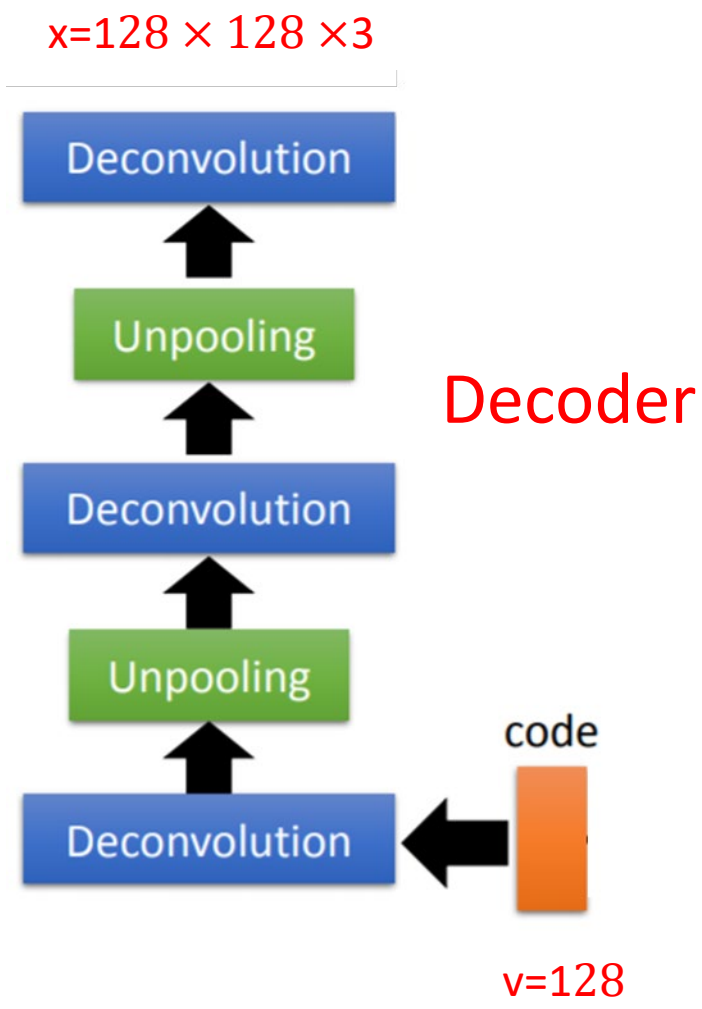
Generative Adversarial Network (GAN)

Practice

- Open “8.1. GAN.ipynb"

Generator

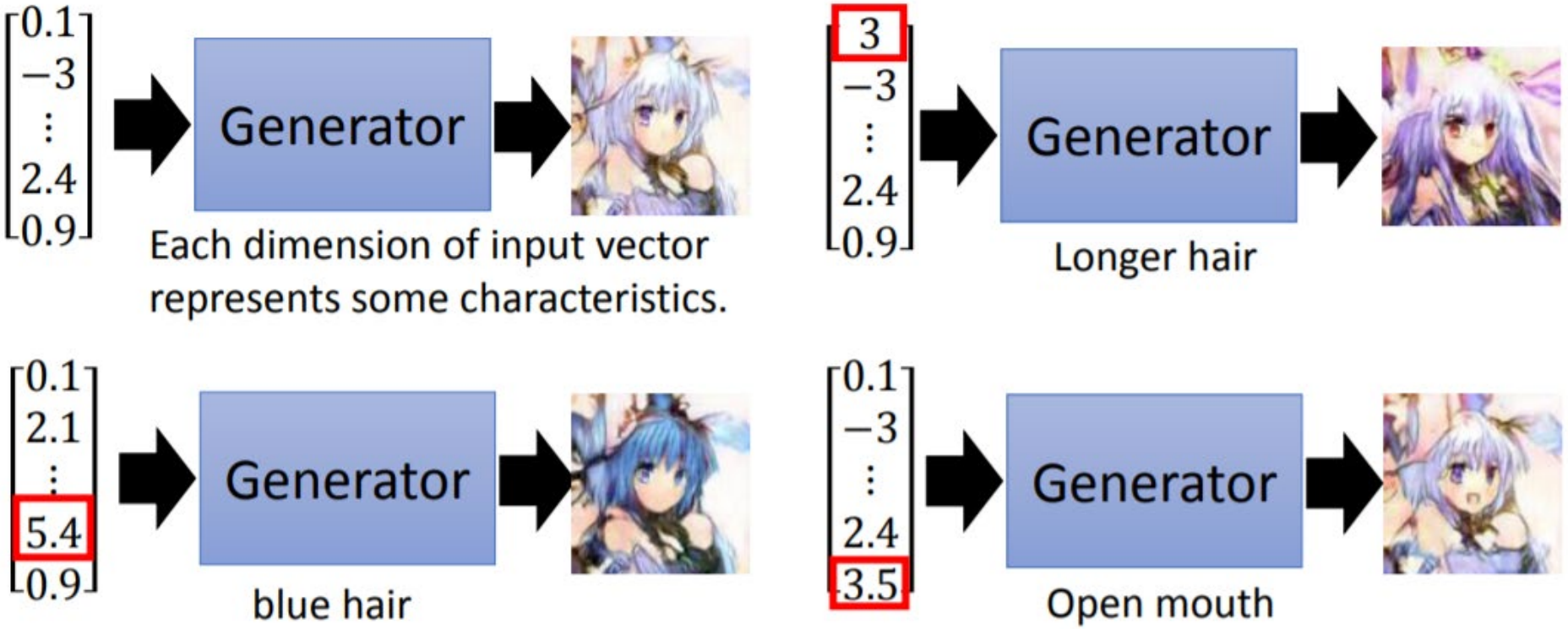


Decoder

[10]:
```
latent_size= 128
```

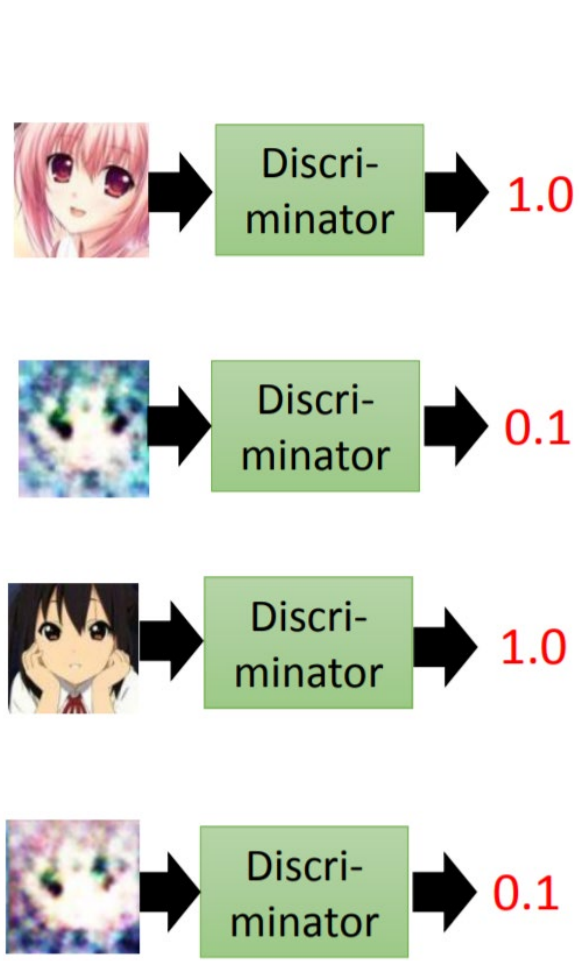
[11]:
```
generator = nn.Sequential(
    # in: latent_size x 1 x 1

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1,
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    # out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    # out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1,
    nn.BatchNorm2d(32),
    nn.ReLU(True),
    # out: 32 x 64 x 64

    nn.ConvTranspose2d(32, 3, kernel_size=4, stride=2, padding=1,
    nn.Tanh()
    # out: 3 x 128 x 128
)
```

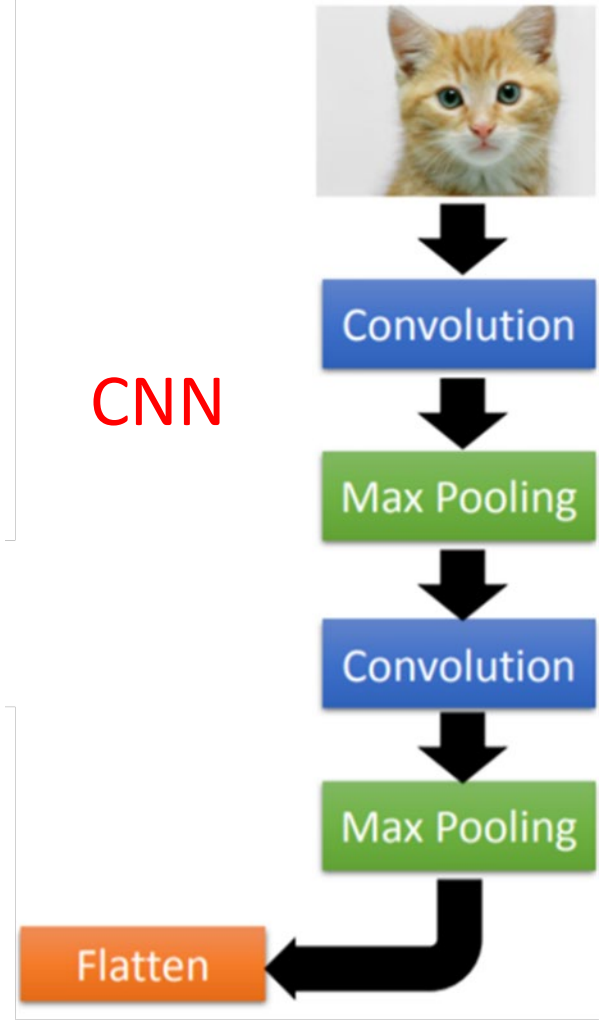
Each dimension in the feature vector represent a property of the output image



Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw

Discriminator





Reference: 李弘毅 GAN Lecture 1 (2018)

[15]:
```
discriminator = nn.Sequential(
    # in: 3 x 128 x 128

    nn.Conv2d(3, 64, kernel_size=4, stride=2, p
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 64 x 64 x 64

    nn.Conv2d(64, 128, kernel_size=4, stride=2,
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 32 x 32

    nn.Conv2d(128, 256, kernel_size=4, stride=2
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 16 x 16

    nn.Conv2d(256, 512, kernel_size=4, stride=2
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 8 x 8

    nn.Conv2d(512, 1024, kernel_size=4, stride=
    nn.BatchNorm2d(1024),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 1024 x 4 x 4

    nn.Conv2d(1024, 1, kernel_size=4, stride=1,
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid())
```

Step1 – Fix G and train D

- Initialize generator and discriminator G D
- In each training iteration:

Step 1: Fix generator G, and update discriminator D

```
[12]: generator.to(device)
```

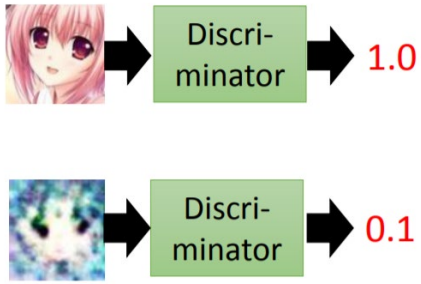
```
[16]: discriminator.to(device)
```

```
[35]: for epoch in range(epochs):
        if(epoch % 10 ==0):
          print(epoch, end=",")
        for real_images, _ in train_dl:
          # Train discriminator
          loss_d, real_score, fake_score = train_discriminator(real_images.
          # Train generator
          loss_g = train_generator(opt_g)
```

Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw

Train discriminator *D*

- Sample m examples $\{x^1, x^2, \dots, x^m\}$ from database
- Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from a distribution
- Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}$, $\tilde{x}^i = G(z^i)$



$D(x^i) \to 1,\ \log D(x^i) \to 0$

$D(\tilde{x}^i) \to 0,\ \log(1 - D(\tilde{x}^i)) \to 0$

log(0.xxx) <0

Update discriminator parameters θ_d to maximize

- $\tilde{V} = \frac{1}{m}\sum_{i=1}^{m} logD(x^i) + \frac{1}{m}\sum_{i=1}^{m} log\left(1 - D(\tilde{x}^i)\right)$
- $\theta_d \leftarrow \theta_d + \eta\nabla\tilde{V}(\theta_d)$

[19]:

```
def train_discriminator(real_images, opt_d):
    # Clear discriminator gradients
    opt_d.zero_grad()

    # Pass real images through discriminator
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0)
    real_loss = F.binary_cross_entropy(real_preds
    real_score = torch.mean(real_preds).item()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size,
    fake_images = generator(latent.to(device))

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds
    fake_score = torch.mean(fake_preds).item()

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item(), real_score, fake_score
```

Review – Cross entropy

Measures the differences between the true probability p_i and the predicted probability q_i

$$H(p,q) = -\sum_i p_i ln(q_i)$$

A	B	C	D	E	F	G	H	I	J	K
z1	z2	y-hat	p1	p2	EXP(z1)	EXP(z2)	F+G	q1	q2	-(P1*LN(Q1)+P2*LN(Q2))
-0.018	0.0855	0	1	0	0.982	1.089	2.071	0.474	0.526	0.74624
-0.0244	0.0741	0	1	0	0.976	1.077	2.053	0.475	0.525	0.74361
-0.0187	0.085	0	1	0	0.981	1.089	2.070	0.474	0.526	0.74634
-0.0258	0.0687	1	0	1	0.975	1.071	2.046	0.476	0.524	0.64701
-0.0267	0.0617	1	0	1	0.974	1.064	2.037	0.478	0.522	0.64992
										0.70662

Step2 – Fix D and train G

Step 2: Fix discriminator D, and update generator G

Generator learns to "fool" the discriminator

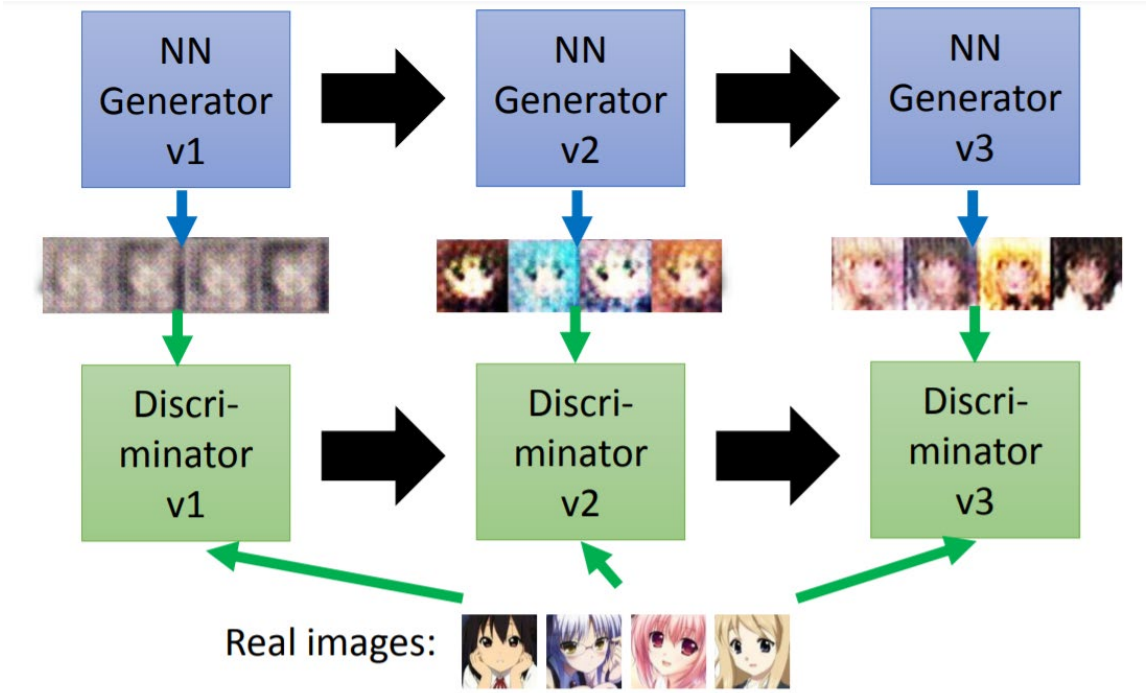
- Sample m noise samples $\{z^1, z^2, \ldots, z^m\}$ from a distribution
- Update generator parameters θ_g to maximize
  - $\tilde{V} = \frac{1}{m}\sum_{i=1}^{m} log\left(D\left(G\left(z^i\right)\right)\right)$
  - $\theta_g \leftarrow \theta_g - \eta \nabla \tilde{V}(\theta_g)$

```
[35]: for epoch in range(epochs):
        if(epoch % 10 ==0):
          print(epoch, end=",")
        for real_images, _ in train_dl:
          # Train discriminator
          loss_d, real_score, fake_score =
          # Train generator
          loss_g = train_generator(opt_g)
```

```
[28]: def train_generator(opt_g):
        # Clear generator gradients
        opt_g.zero_grad()

        # Generate fake images
        latent = torch.randn(batch_size, latent_size,
        fake_images = generator(latent)

        # Try to fool the discriminator
        preds = discriminator(fake_images)
        targets = torch.ones(batch_size, 1, device=dev
        loss = F.binary_cross_entropy(preds, targets)

        # Update generator weights
        loss.backward()
        opt_g.step()

        return loss.item()
```

An evolution process

```
[35]: for epoch in range(epochs):
        if(epoch % 10 ==0):
          print(epoch, end=",")
        for real_images, _ in train_dl:
          # Train discriminator
          loss_d, real_score, fake_score = train_discriminator(real_images
          # Train generator
          loss_g = train_generator(opt_g)
```



Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw

Visualize fake images generated by *G* during the evolution process

[17]:
```
sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)
```

[34]:
```
fixed_latent = torch.randn(64, latent_size,
# used to generate saved images
```

```
if(epoch % 50 ==0):
  # Log losses & scores (last batch)
  print("Epoch [{}/{}], loss_g: {:.4f}, loss_d:
      epoch+1, epochs, loss_g, loss_d, real_sc
  # Save generated images
  save_samples(epoch+start_idx, fixed_latent, s
```

[18]:
```
def save_samples(index, latent_tensors, show=True):
  fake_images = generator(latent_tensors)
  fake_fname = 'generated-images-{0:0=4d}.png'.format(index)
  save_image(denorm(fake_images), os.path.join(sample_dir, f
  print('Saving', fake_fname)
  if show:
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(fake_images.cpu().detach(), nrow=8).
```

8.1. GAN.ipynb

檔案 編輯 檢視畫面 插入 執行階段 工具 說明 無法儲存變更

共用

+ 程式碼 + 文字 複製到雲端硬碟

RAM 磁碟 編輯

檔案

..

gdrive

generated

generated-images-0001.png

generated-images-0051.png

generated-images-0101.png

generated-images-0151.png

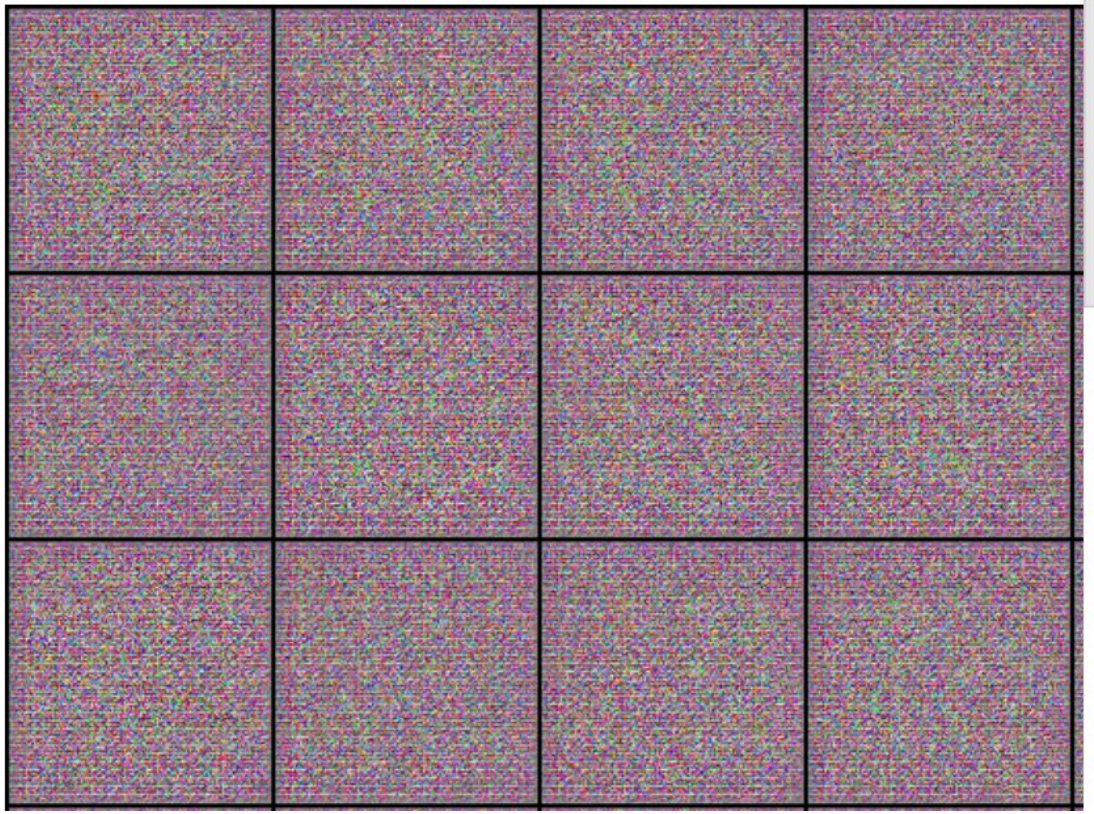
generated-images-0201.png

sample_data

```
        # Log losses & scores (l
        print("Epoch [{}/{}], loss_
              epoch+1, epochs, lo
        # Save generated images
        latent = torch.randn(batch_
        save_samples(epoch+start_idx,
        #save_samples(epoch+start_idx
```

```
0, Epoch [1/1200], loss_g: 5.6803, los
Saving generated-images-0001.png
10, 20, 30, 40, 50, Epoch [51/1200], loss_
Saving generated-images-0051.png
60, 70, 80, 90, 100, Epoch [101/1200], los
Saving generated-images-0101.png
110, 120, 130, 140, 150, Epoch [151/1200],
Saving generated-images-0151.png
160, 170, 180, 190, 200, Epoch [201/1200],
Saving generated-images-0201.png
210, 220,
```

generated-images-0201.png generated-images-0001.png



8.1. GAN.ipynb

檔案 編輯 檢視畫面 插入 執行階段 工具 說明 無法儲存變更

共用

+ 程式碼 + 文字 複製到雲端硬碟

RAM 磁碟 編輯

檔案

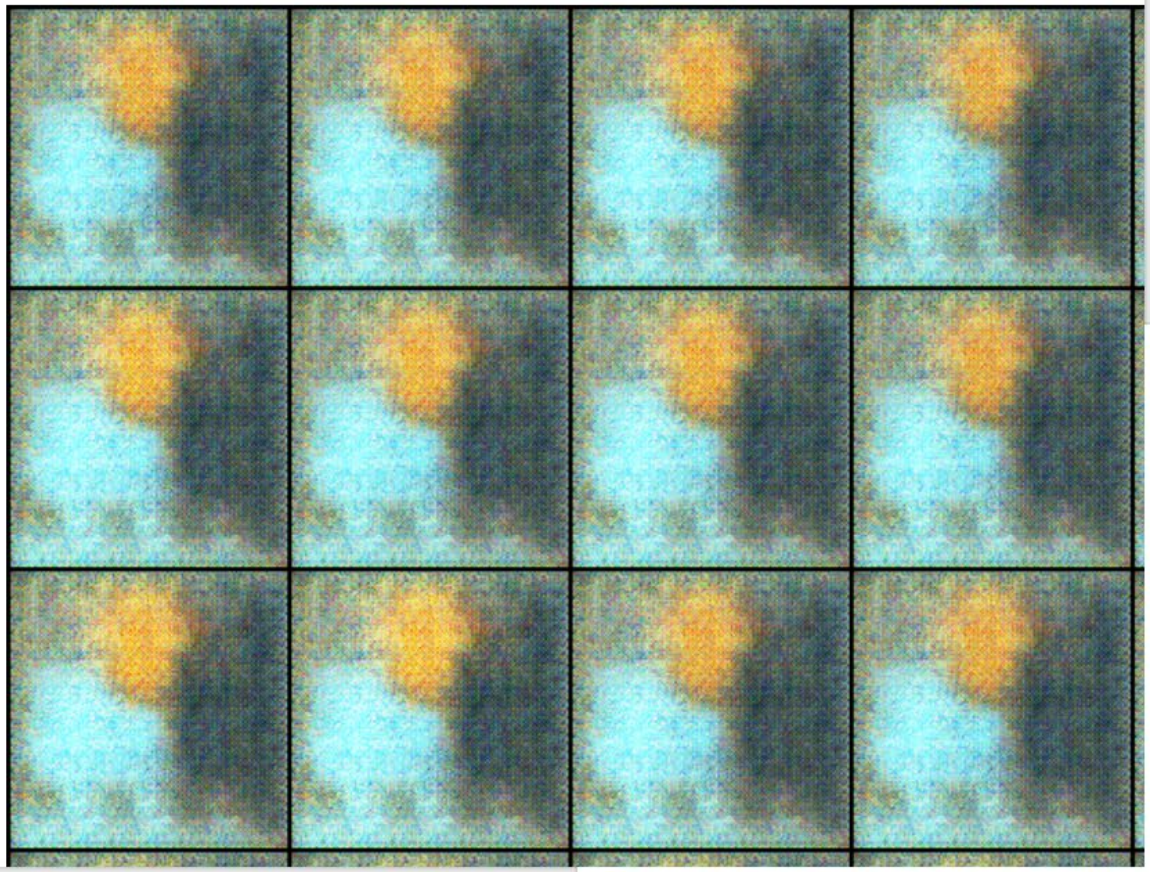
..
gdrive
generated
generated-images-0001.png
generated-images-0051.png
generated-images-0101.png
generated-images-0151.png
generated-images-0201.png
generated-images-0251.png
sample_data

磁碟 29.48 GB 可用

```
Saving generated-images-0001.png
10, 20, 30, 40, 50, Epoch [51/1200], loss_
Saving generated-images-0051.png
60, 70, 80, 90, 100, Epoch [101/1200], los
Saving generated-images-0101.png
110, 120, 130, 140, 150, Epoch [151/1200],
Saving generated-images-0151.png
160, 170, 180, 190, 200, Epoch [201/1200],
Saving generated-images-0201.png
210, 220, 230, 240, 250, Epoch [251/1200],
Saving generated-images-0251.png
260, 270, 280,
```

[ ]

generated-images-0251.png



8.1. GAN.ipynb

檔案 編輯 檢視畫面 插入 執行階段 工具 說明 無法儲存變更

共用

檔案

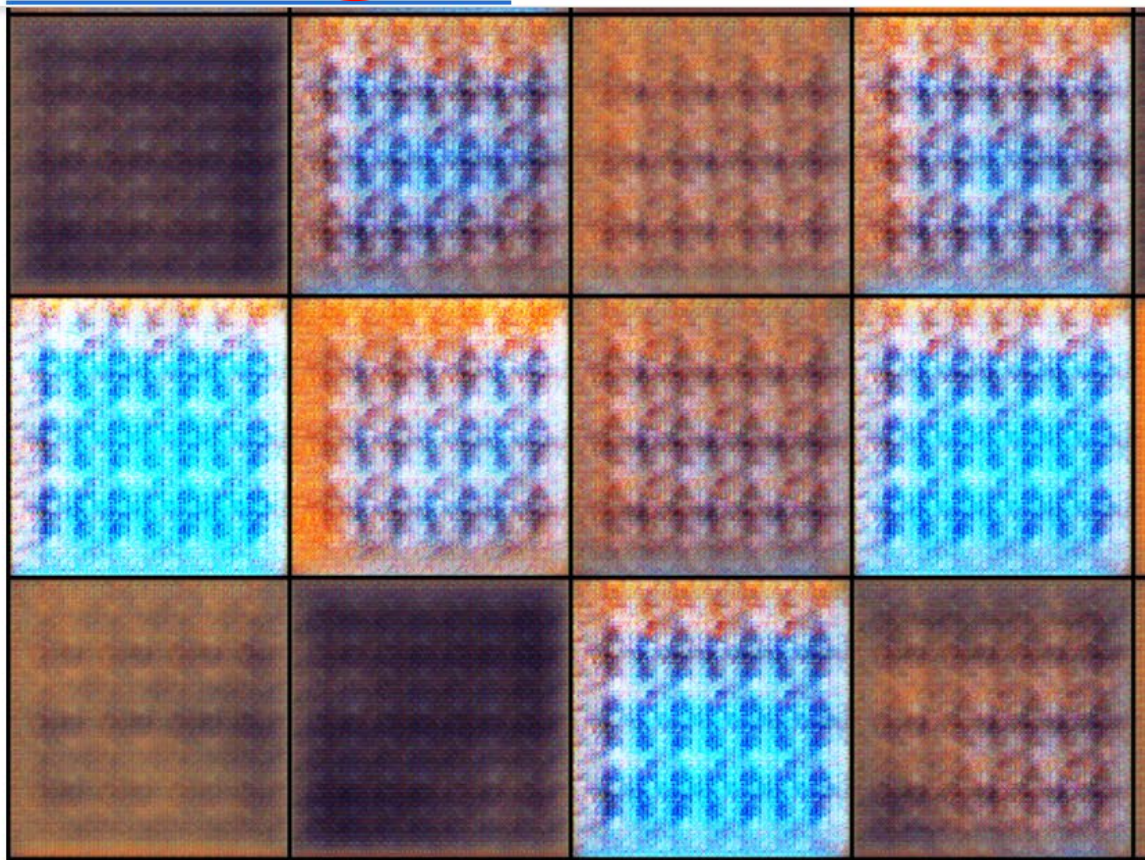
..
gdrive
generated
generated-images-0001.png
generated-images-0051.png
generated-images-0101.png
generated-images-0151.png
generated-images-0201.png
generated-images-0251.png
generated-images-0301.png
sample_data

+ 程式碼 + 文字 複製到雲端硬碟

RAM 磁碟 編輯

```
Saving generated-images-0001.png
10, 20, 30, 40, 50, Epoch [51/1200], loss_
Saving generated-images-0051.png
60, 70, 80, 90, 100, Epoch [101/1200], los
Saving generated-images-0101.png
110, 120, 130, 140, 150, Epoch [151/1200],
Saving generated-images-0151.png
160, 170, 180, 190, 200, Epoch [201/1200],
Saving generated-images-0201.png
210, 220, 230, 240, 250, Epoch [251/1200],
Saving generated-images-0251.png
260, 270, 280, 290, 300, Epoch [301/1200],
Saving generated-images-0301.png
310, 320, 330, 340,
```

generated-images-0301.png



Epoch =41201

34151

28651

25901

23201

16401

15551

14301

Save the generated images as a video

[37]:
```
img_array = []
for filename in files:
  img = cv2.imread(filename)
  height, width, layers = img.shape
  size = (width,height)
  img_array.append(img)

out = cv2.VideoWriter('GANTrainingVideo.avi',cv2.VideoWriter_fourcc(*'DIVX'), 15, size)

for i in range(len(img_array)):
    out.write(img_array[i])
out.release()
```

Tom & Jerry video https://youtu.be/uDEGITFmhiQ

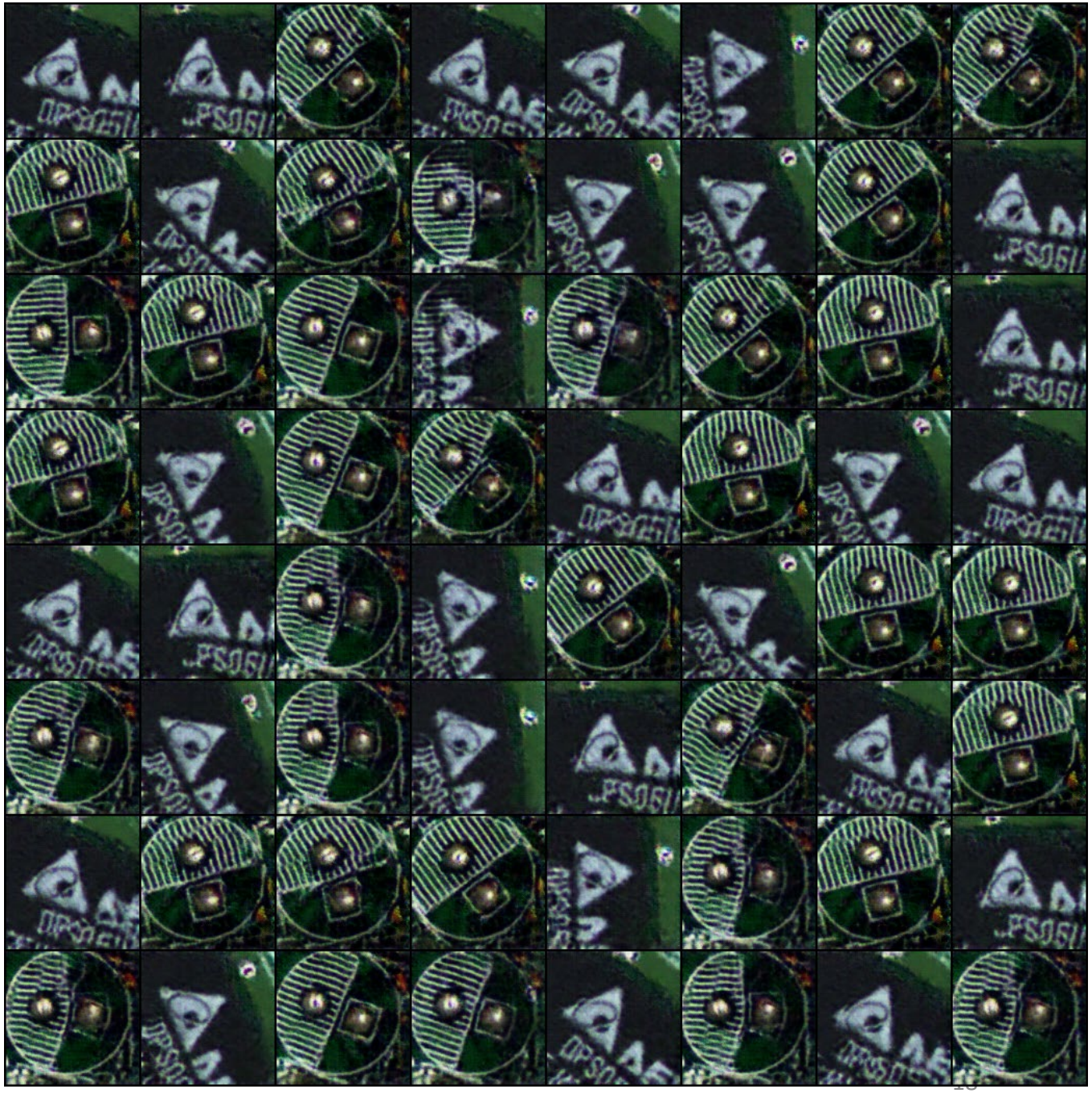
HW7 (1)

- Use your own images, e.g., facial expression to train a GAN.
- Show the generated images.
- Try latent vectors like (all 1), (all 0.5), (all 0.3), (0, 0, 1, 0, ….), (one dimension goes from 0 to 1 and other dimensions fixed), … to see the generated images.

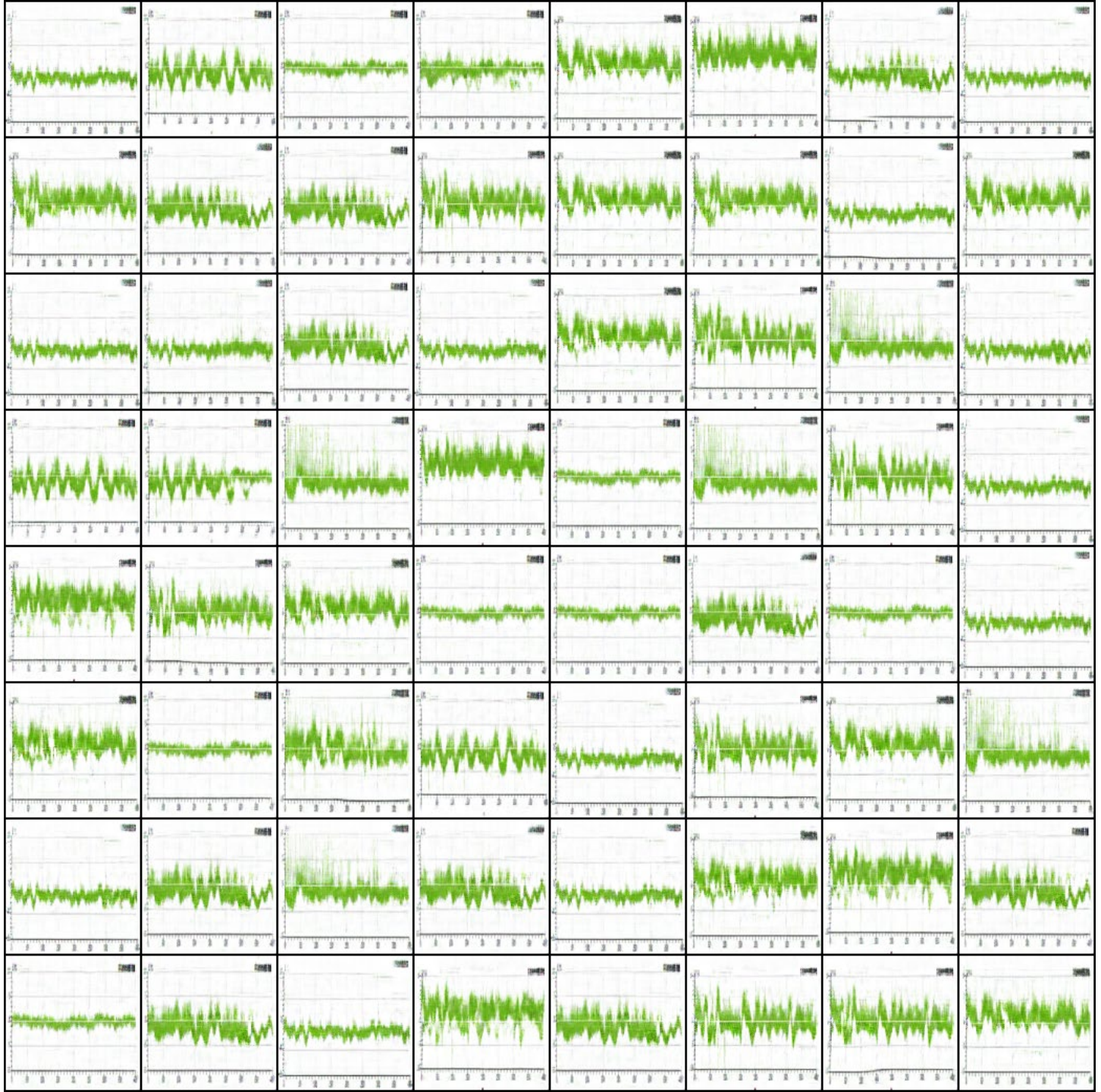
Images generated from *G* after 5651 epochs

Image size = 180x180x3
Class 1 = 100, Class 2 = 100
Latent vector size = 128
batch=32

Run on 2080Ti, takes about 15 min.



Images-19951

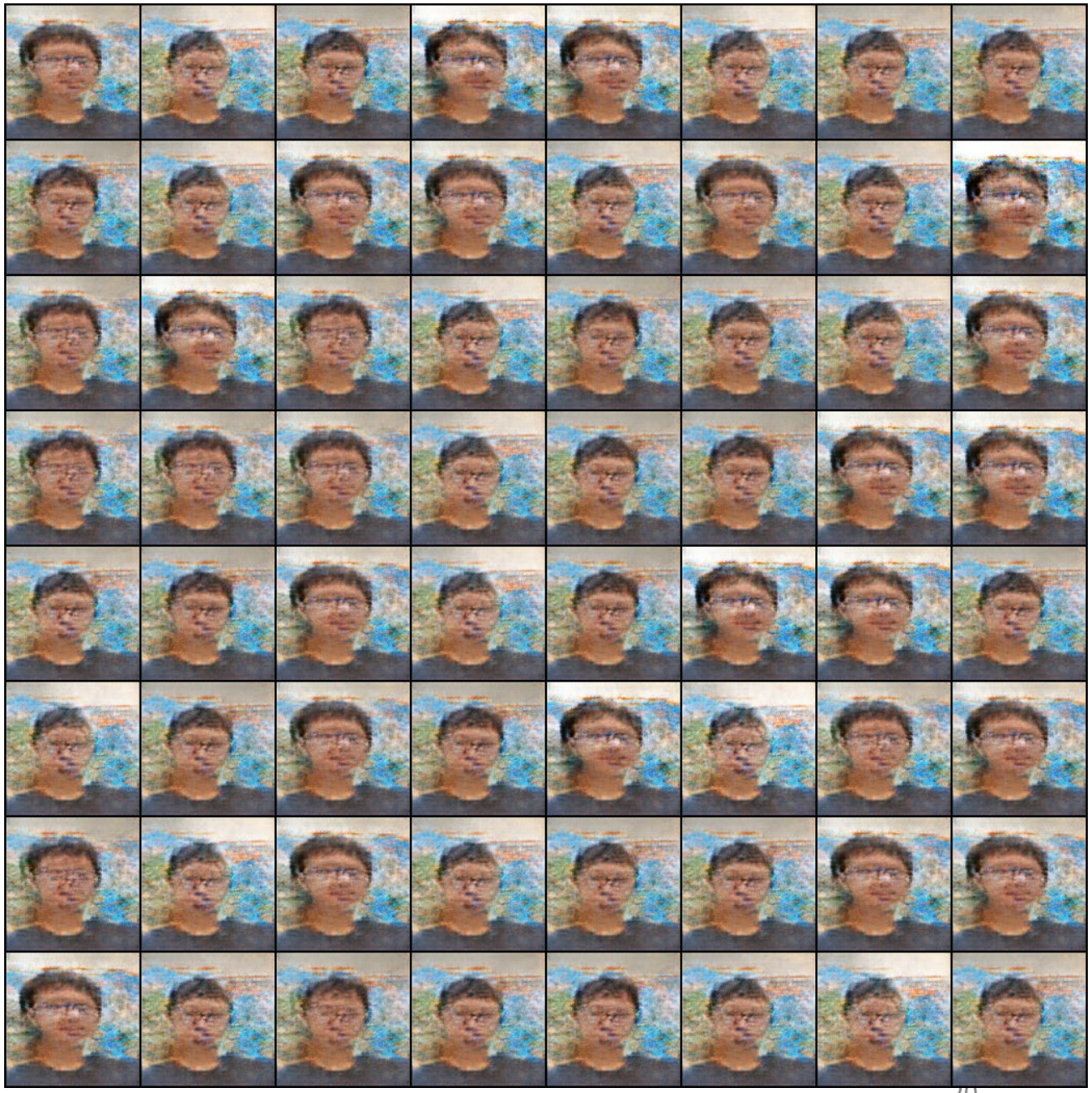


1071228 陳仁慶

Images generated from *G* after 2100 epochs

Image size = 128x128x3
Class 1 = 10, Class 2 = 10
Latent vector size = 128
batch=128

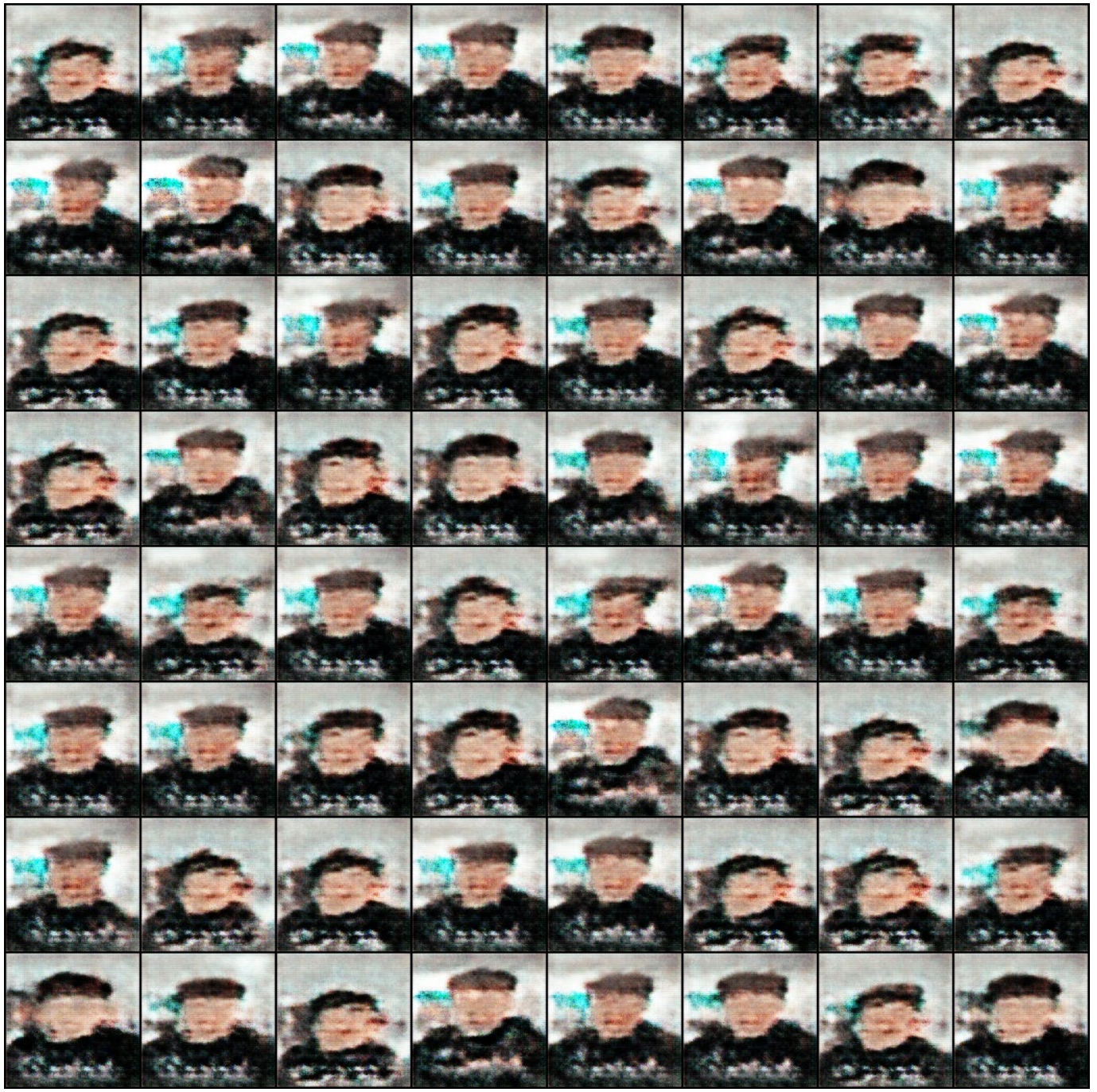
Run on Colab Tesla K80, takes about 120 min.



Images generated from *G* after 3051 epochs

Image size = 128x128x3
Class 1 = 10, Class 2 = 10
Latent vector size = 128
batch=128

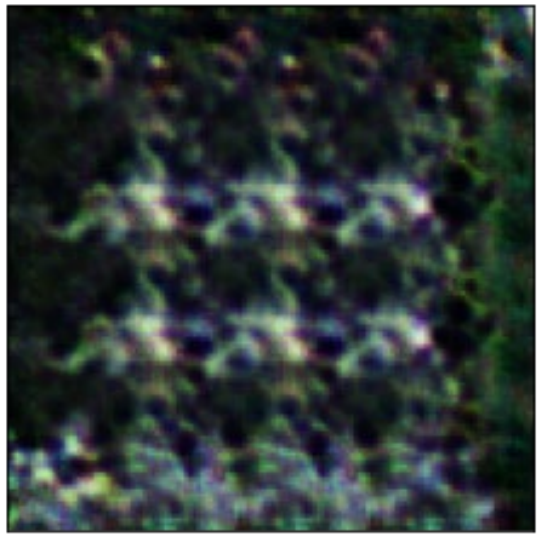
Run on Colab Telsa T4, takes about 60 min.



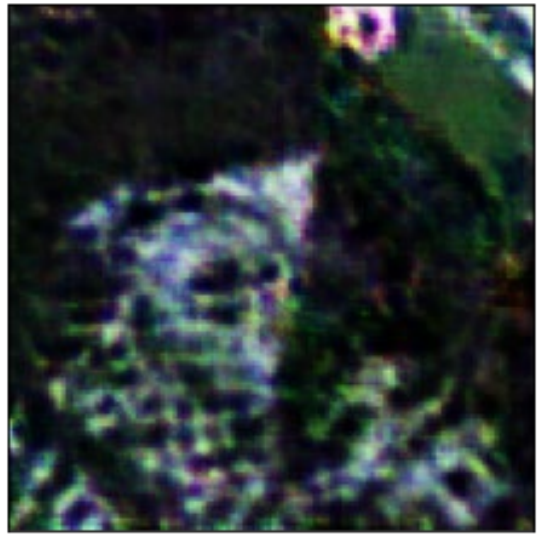
1061221 林大洋

Input vector: [1, 1, 1...1]

[0, 0, 0...0]



[-1, -1, -1...-1]



```
x_ones = torch.ones(1, latent_size, 1, 1)
x_negs = -torch.ones(1, latent_size, 1, 1)
x_zeros = torch.zeros(1, latent_size, 1, 1)
```

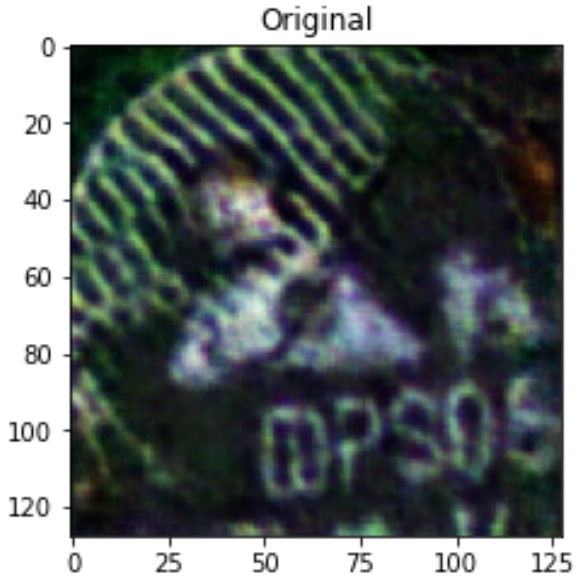
```
fake_image = denorm(generator(x_ones.to(device)).detach().cpu()[0].permute(1, 2, 0))
print('Image size:',fake_image.shape)
plt.figure(figsize=(4, 4))
plt.imshow(fake_image.numpy().astype(float))
plt.xticks([])
plt.yticks([])
```

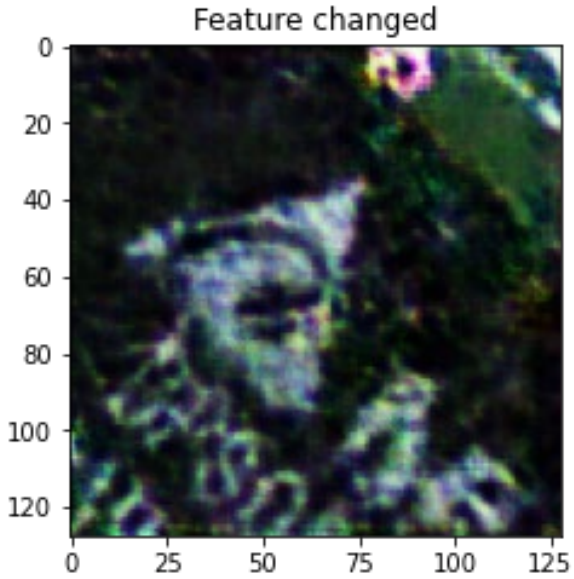
```
Image size: torch.Size([128, 128, 3])
```

```
# Change a feature from the feature vector
feature_index = 3
new_ones = torch.ones(1, latent_size, 1, 1)
fake1 = denorm(generator(new_ones.to(device)).detach().cpu()[0].permute(1, 2, 0)).numpy().astype(float)
new_ones[:, feature_index, :, :] = 50
fake2 = denorm(generator(new_ones.to(device)).detach().cpu()[0].permute(1, 2, 0)).numpy().astype(float)
print('Fake image size: ', fake1.shape)
fig, ax = plt.subplots(1, 2, figsize=(8, 4))
ax[0].imshow(fake1)
ax[1].imshow(fake2)
ax[0].set_title('Original')
ax[1].set_title('Feature changed')
```

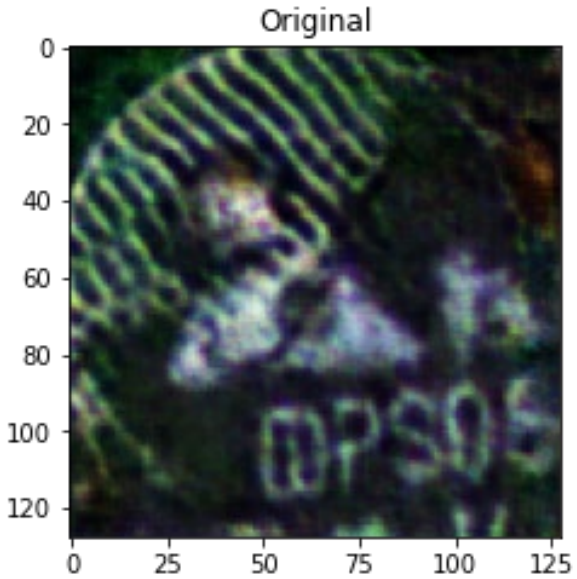
```
Fake image size:  (128, 128, 3)
```

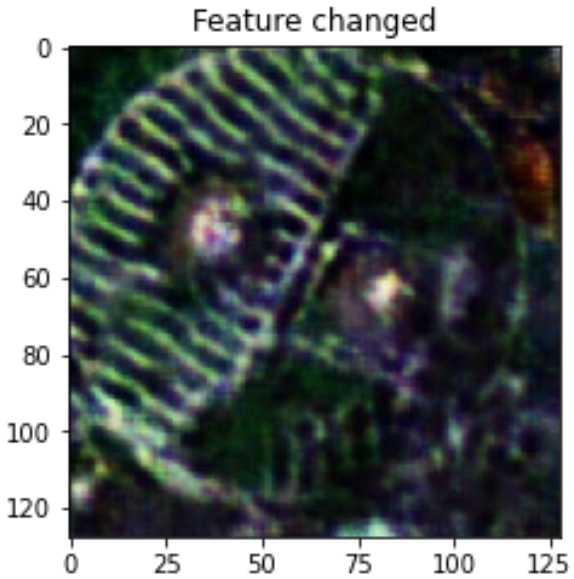
Feature 3=50





Feature 123=-50



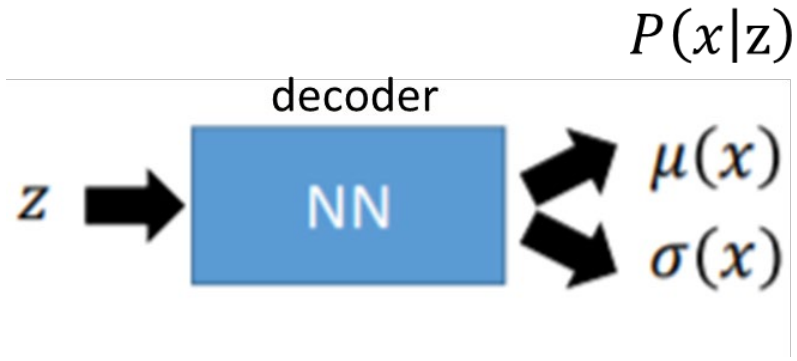


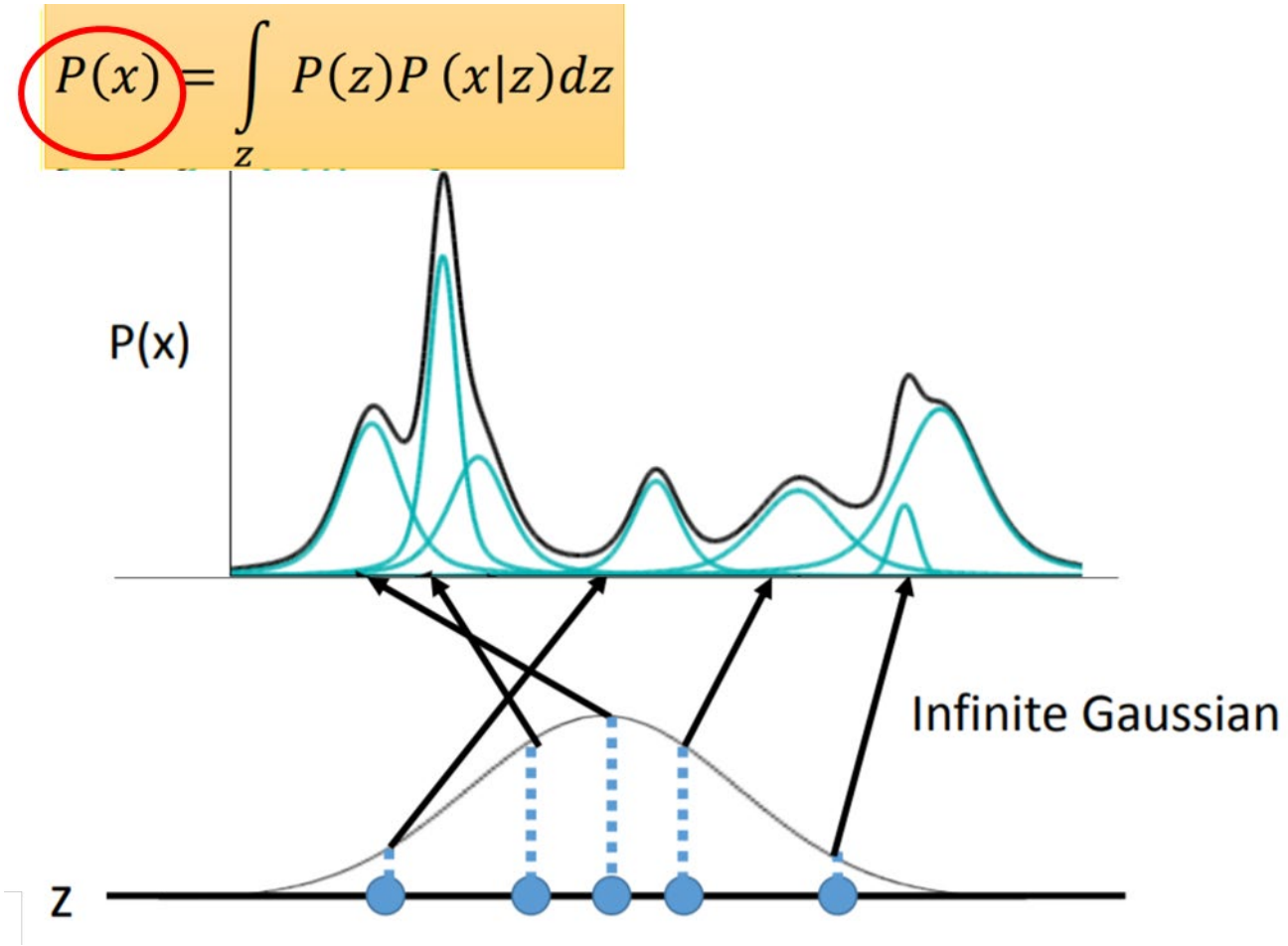
Theory behind GAN, VAE, PPO

The concept of deep generative model

1. Use probability distribution model to model the image generation process or the state-action generation process.
2. With this probability model, we want to maximize the likelihood to observe the training data or observe the trajectories with higher long-term rewards.
3. Maximize the likelihood can be achieved by minimizing the KL divergence.
3. Use DNN to represent a more general probability model.
4. Use DNN to represent a more general divergence measure.

(VAE) 1. Modelling the image generation process as a mixed Gaussian distribution model





Reference: 李弘毅 ML Lecture 18 https://youtu.be/8zomhgKrsmQ

(VAE) 2. Given the image generation model, we want to maximize the likelihood to observe the training images

Maximizing Likelihood

$$P(x) = \int_z P(z)P(x|z)dz$$

P(z) is normal distribution

$x|z \sim N(\mu(z), \sigma(z))$

$\mu(z), \sigma(z)$ is going to be estimated

$$L = \sum_x logP(x)$$

Maximizing the likelihood of the observed x

$$L = p(x^1) \times p(x^2) \times p(x^3) \times \cdots p(x^m) = \prod_{i=1,\ldots,m} P(x^i)$$

Reference: 李弘毅 ML Lecture 18 https://youtu.be/8zomhgKrsmQ

(VAE) 3. Maximize the likelihood can be achieved by minimizing KL divergence

Max. L_b can be done by min. $KL(q(z|x)||P(z))$ and max. $\int_z q(z|x)logP(x|z)dz$. That is why loss = KLD + MSE (x, $\hat{x}$)

Minimize

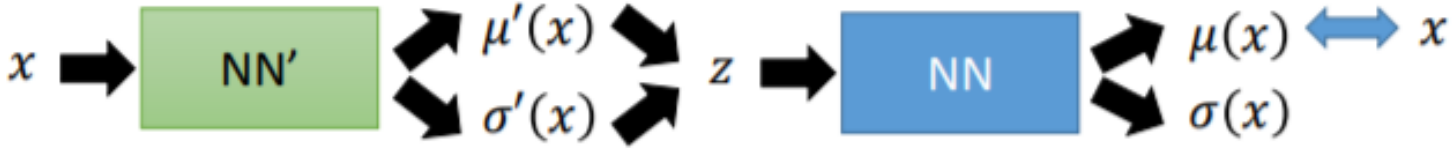
$$\sum_{i=1}^{3}(exp(\sigma_i) - (1+\sigma_i) + (m_i)^2)$$

Minimizing $KL(q(z|x)||P(z))$

(Refer to the Appendix B of the original VAE paper)

Maximizing

$$\int_z q(z|x)logP(x|z)dz = E_{q(z|x)}[logP(x|z)]$$



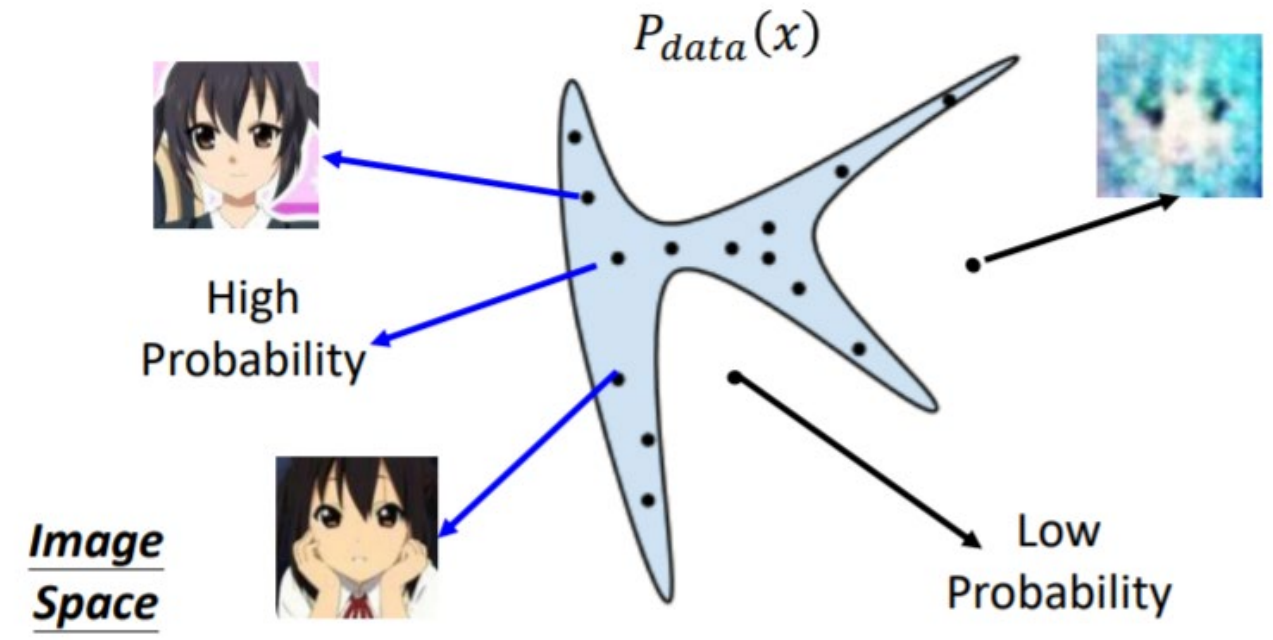
by min. $MSE(x, \hat{x})$

Reference: 李弘毅 ML Lecture 18 https://youtu.be/8zomhgKrsmQ

(GAN) 1. Modelling the generator as a probability distribution model P_G

X: an image (a high-dimensional vector)

- We want to find data distribution $P_{data}(x)$



Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

(GAN) 2. Given the image generation model, we want to maximize the likelihood to observe the training images

- Given a data distribution $P_{data}(x)$ (We can sample from it.)
- We have a distribution $P_G(x;\theta)$ parameterized by θ
  - We want to find θ such that $P_G(x;\theta)$ close to $P_{data}(x)$
  - E.g. $P_G(x;\theta)$ is a Gaussian Mixture Model, θ are means and variances of the Gaussians

Sample $\{x^1, x^2, \ldots, x^m\}$ from $P_{data}(x)$

We can compute $P_G(x^i;\theta)$

Likelihood of generating the samples

$$L = \prod_{i=1}^{m} P_G(x^i;\theta)$$

Find θ^* maximizing the likelihood

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

(GAN) 3. Maximize the likelihood can be achieved by minimizing the KL divergence

Maximum likelihood estimation = minimum KL divergence

$$\theta^* = arg\max_\theta \prod_{i=1}^{m} P_G(x^i;\theta) = arg\min_\theta KL(P_{data}||P_G)$$

$$\theta^* = arg\max_\theta \prod_{i=1}^{m} P_G(x^i;\theta) = arg\max_\theta log\prod_{i=1}^{m} P_G(x^i;\theta)$$

$$= arg\max_\theta \sum_{i=1}^{m} logP_G(x^i;\theta) \quad \{x^1, x^2, \dots, x^m\} \text{ from } P_{data}(x)$$

$$\approx arg\max_\theta E_{x\sim P_{data}}[logP_G(x;\theta)]$$

$$= arg\max_\theta \int_x P_{data}(x)logP_G(x;\theta)dx - \int_x P_{data}(x)logP_{data}(x)dx$$

$$= arg\min_\theta KL(P_{data}||P_G)$$

How to define a general P_G?

$$\int_x \mathrm{P}_{data}(x)logP_{data}(x)dx = logP_{data}(x)$$

$$D_{KL}(q||p) = \sum_{i=1}^{N} q(x_i)\log(\frac{q(x_i)}{p(x_i)})$$

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

(Classification) 1. Assuming the training data is generated from $y_1 = P_{w,b}(C_1|x) = \sigma(w \cdot x + b)$ 2. Maximize the likelihood to observe the training data

Training Data	x^1	x^2	x^3	……	x^N
	C_1	C_1	C_2		C_1

$$L(w,b) = f_{w,b}(x^1) f_{w,b}(x^2) \left(1 - f_{w,b}(x^3)\right) \cdots f_{w,b}(x^N)$$

$$-lnL(w,b) = lnf_{w,b}(x^1) + lnf_{w,b}(x^2) + ln\left(1 - f_{w,b}(x^3)\right) \cdots$$

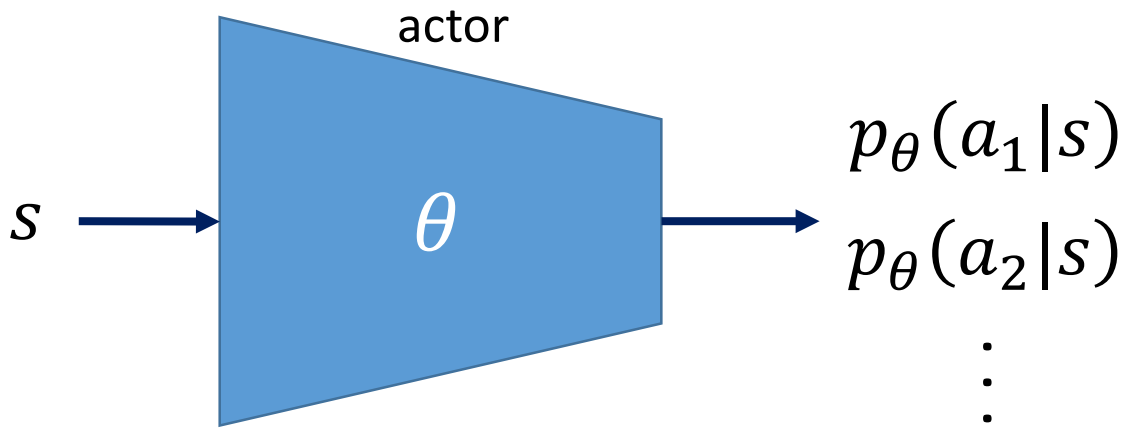
$\hat{y}^n$: 1 for class 1, 0 for class 2

$$= \sum_n -\left[\hat{y}^n lnf_{w,b}(x^n) + (1 - \hat{y}^n) ln\left(1 - f_{w,b}(x^n)\right)\right]$$

Cross entropy between two Bernoulli distribution

Reference: 李弘毅 ML Lecture 5 https://youtu.be/hSXFuypLukA

(PPO) 1. Modelling the state-action mapping network as $p_\theta(a|s)$



(PPO) 2. Given the probability model, we want to maximize the likelihood to observe trajectories with higher long-term rewards

$$\tau = (s_1, a_1, r_1, s_2, a_2, r_{2,} \cdots s_T, a_T)$$

$$p_\theta(\tau) = p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_2)p(s_3|s_2, a_2)\cdots$$

$$\bar{R}_\theta = \sum R(\tau)\, p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)] \qquad R(\tau) = \sum_{t=1}^{T} r_t$$

$$\max_\theta E[\bar{R}_\theta]$$

$$\nabla\bar{R}_\theta = \sum R(\tau)\, \nabla p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla \log p_\theta(\tau)] \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla \log p_\theta(\tau^n)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_t^n)$$

(PPO) 3. To facilitate off-policy sampling, we minimize the KL divergence

Off-policy

$$\nabla \bar{R}_\theta = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t)\nabla \log p_\theta(a_t^n|s_t^n)\right]$$

Loss function

$$J^{\theta'}(\theta) = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t)\right]$$

Proximal policy optimization (PPO)

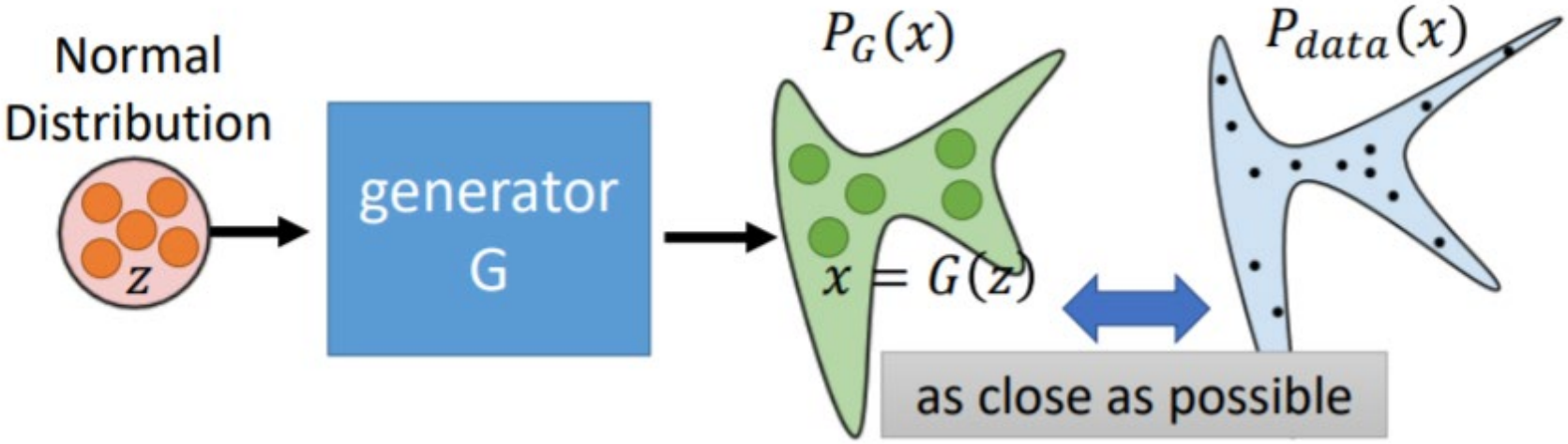
$$J_{PPO}^{\theta'}(\theta) = J^{\theta'}(\theta) - \beta KL(\theta,\theta')$$

$$J_{PPO2}^{\theta'}(\theta) = \sum_{(s_t,a_t)} min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right) A^{\theta'}(s_t,a_t)\right)$$

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

4. Use DNN to represent a more general probability model

- A generator G is a network. The network defines a probability distribution P_G



$$G^* = arg \min_G Div(P_G, P_{data})$$

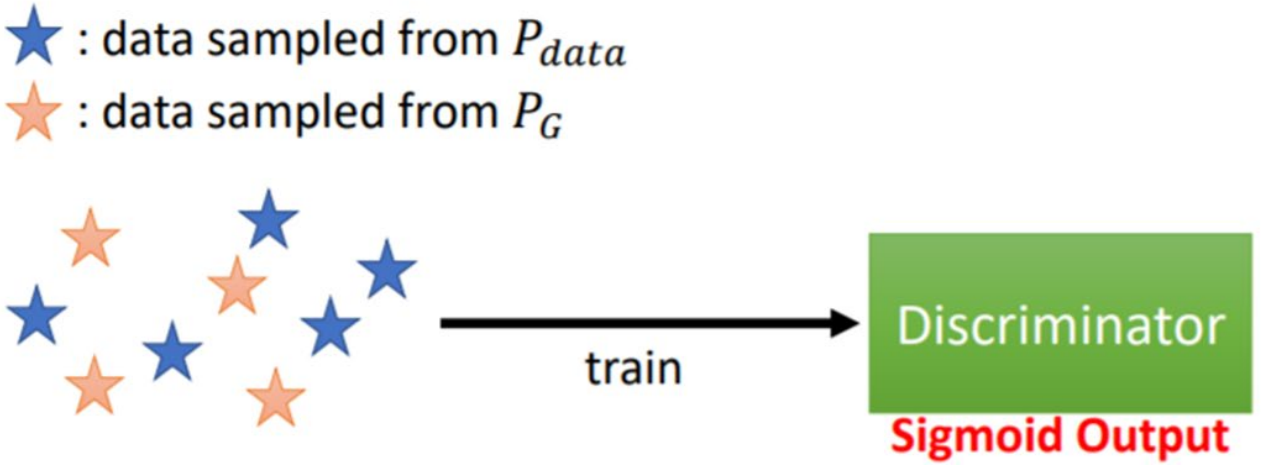
Divergence between distributions P_G and P_{data}

How to compute the divergence?

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

5. Use DNN to represent a more general divergence measure

How to compute the divergence when a NN is used to represent a more general probability model? – train a discriminator *NN* to compute the divergence



Example Objective Function for D

$$V(G,D) = E_{x\sim P_{data}}[logD(x)] + E_{x\sim P_G}[log(1-D(x))]$$

(G is fixed)

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

[Goodfellow, et al., NIPS, 2014]

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

The objective function for D is related to JS divergence

$$V(D,G) = E_{x\sim P_{data}}[logD(x)] + E_{x\sim P_G}[\log(1-D(x))]$$

- Given G, what is the optimal D* maximizing

$$V = E_{x\sim P_{data}}[logD(x)] + E_{x\sim P_G}[log(1-D(x))]$$

$$= \int_x P_{data}(x)logD(x)\,dx + \int_x P_G(x)log(1-D(x))\,dx$$

$$= \int_x [P_{data}(x)logD(x) + P_G(x)log(1-D(x))]\,dx$$

Assume that D(x) can be any function

- Given x, the optimal D* maximizing

$$P_{data}(x)logD(x) + P_G(x)log(1-D(x))$$

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

The objective function for D is related to JS divergence

$$V(D,G) = E_{x\sim P_{data}}[logD(x)] + E_{x\sim P_G}[\log(1-D(x))]$$

$$\max_D V(G,D) = V(G,D^*) \qquad D^*(x) = \frac{P_{data}(x)}{P_{data}(x)+P_G(x)}$$

$$= -2log2 + \int_x P_{data}(x) log \frac{P_{data}(x)}{\left(P_{data}(x)+P_G(x)\right)/2} dx$$

$$+ \int_x P_G(x) log \frac{P_G(x)}{\left(P_{data}(x)+P_G(x)\right)/2} dx$$

$$= -2\log2 + \mathrm{KL}\left(\mathrm{P_{data}} || \frac{\mathrm{P_{data}}+\mathrm{P_G}}{2}\right) + \mathrm{KL}\left(\mathrm{P_G} || \frac{\mathrm{P_{data}}+\mathrm{P_G}}{2}\right)$$

$$= -2log2 + 2JSD(P_{data}||P_G)$$ Jensen-Shannon divergence

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

The objective function for D is implemented as binary cross entropy

$$V(D,G) = E_{x\sim P_{data}}[logD(x)] + E_{x\sim P_G}[\log(1-D(x))]$$

Binary Classifier

D is a binary classifier with sigmoid output (can be deep)

$\{x^1, x^2, \ldots, x^m\}$ from $P_{data}(x)$ ➡ Positive examples

$\{\tilde{x}^1, \tilde{x}^2, \ldots, \tilde{x}^m\}$ from $P_G(x)$ ➡ Negative examples

Minimize Cross-entropy

Cross entropy:

$$C(f(x^n), \hat{y}^n) = -\left[\hat{y}^n ln f(x^n) + (1-\hat{y}^n) ln\left(1-f(x^n)\right)\right]$$

(GAN) Two adversarial neural networks *G* and *D*

(1) Train generator

$$G^* = arg \min_G Div(P_G, P_{data})$$

small divergence

★ : data sampled from P_{data}

★ : data sampled from P_G

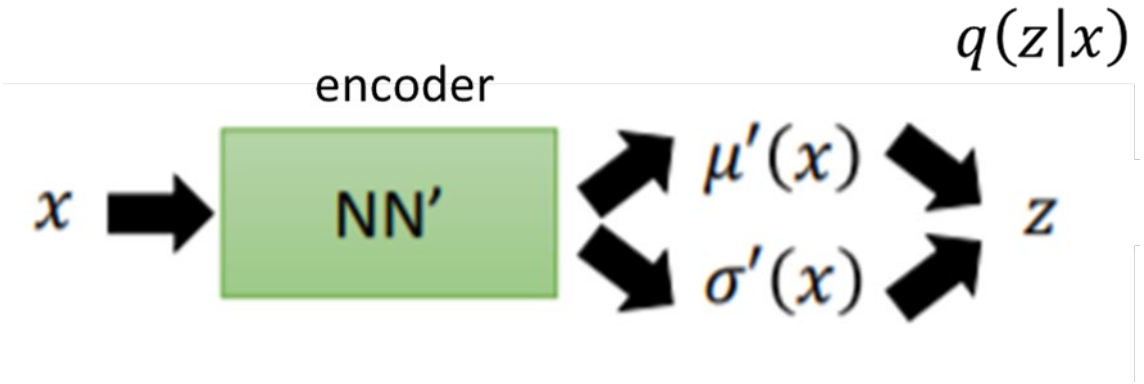
(2) Train discriminator

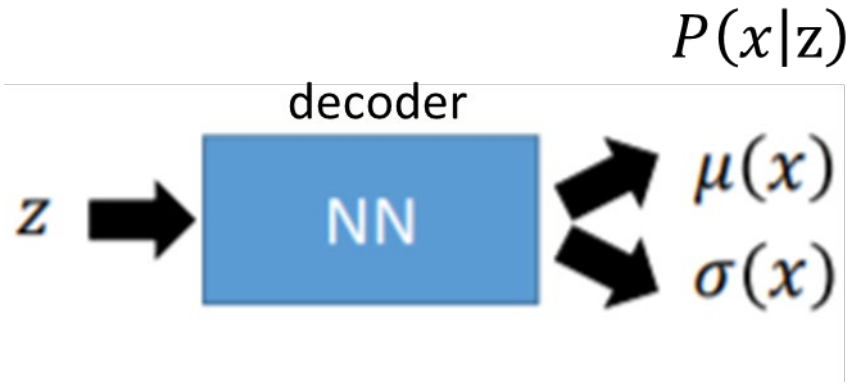
$$D^* = arg \max_D V(D, G)$$

large divergence

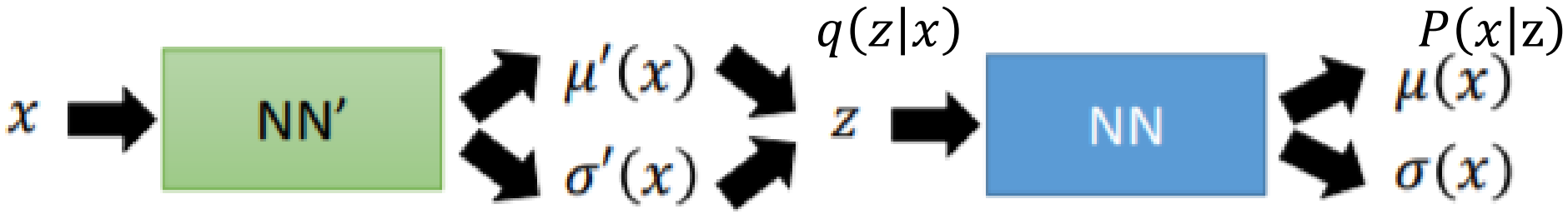
$$V(D,G) = E_{x \sim P_{data}}[logD(x)] + E_{x \sim P_G}[\log(1 - D(x))]$$

(VAE) Two sequential neural networks, one encode and one decode



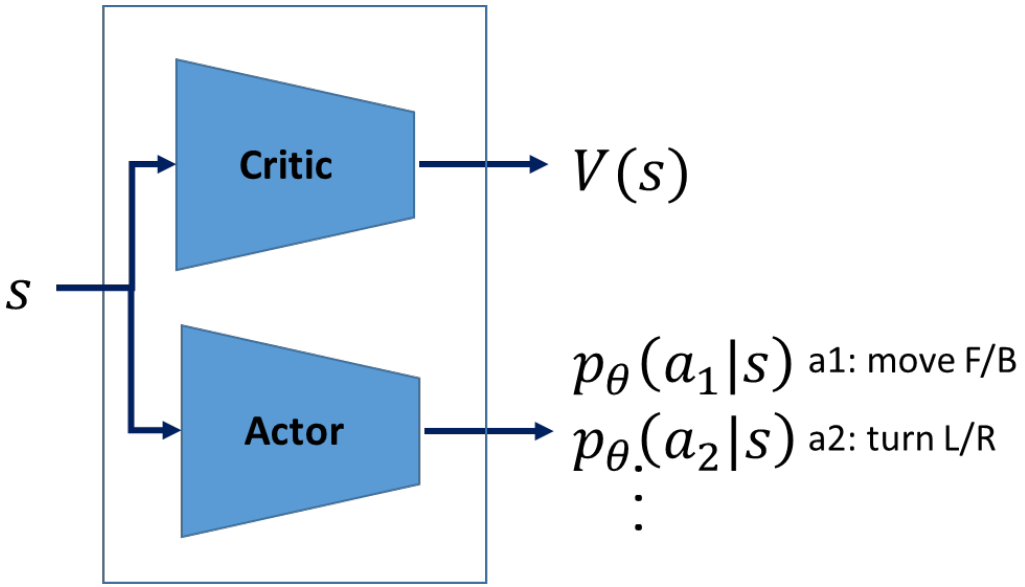


$$\mathrm{L} = MSE(x, \hat{x}) + KL(q(z|x)||P(z))$$



(PPO-AC) Two parallel neural networks, one actor and one critic

$$L = c_v L_v + L_\pi - \beta L_{reg}$$



(1) Actor – Learns the best actions (that can have maximum long-term rewards)

$$L_\pi = \sum_{(s_t,a_t)} min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right) A^{\theta'}(s_t,a_t)\right)$$

(2) Critic – Learns the expected value of the long-term reward.

$$L_v = MSE\ of\ (return - v)$$

Algorithm

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

- Given G_0
- Find D_0^* maximizing $V(G_0, D)$

$V(G_0, D_0^*)$ is the JS divergence between $P_{data}(x)$ and $P_{G_0}(x)$

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

$$V(D,G) = E_{x \sim P_{data}}[logD(x)] + E_{x \sim P_G}[\log(1 - D(x))]$$

- $\theta_G \leftarrow \theta_G - \eta \, \partial V(G, D_0^*)/\partial \theta_G$ ➡ Obtain G_1
- Find D_1^* maximizing $V(G_1, D)$

Decrease JS divergence(?)

$$G^* = arg \min_G Div(P_G, P_{data})$$

$V(G_1, D_1^*)$ is the JS divergence between $P_{data}(x)$ and $P_{G_1}(x)$

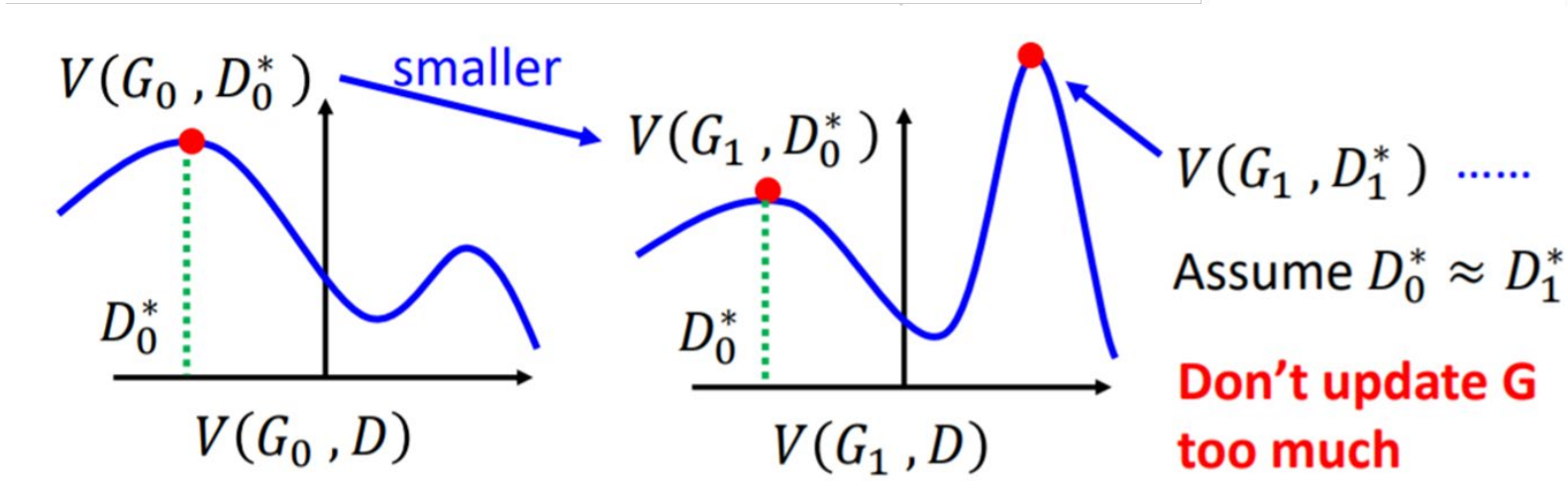
- $\theta_G \leftarrow \theta_G - \eta \, \partial V(G, D_1^*)/\partial \theta_G$ ➡ Obtain G_2
- ……

Decrease JS divergence(?)

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

Do not update G too much…

When we update G from G_0 to G_1, the divergence measurement function $V(G_0, D^*)$ will no longer fully represent the JS divergence between P_G and P_{data}



(PPO): We do not update $p_\theta(a_t|s_t)$ too much so that we can calculate $p_{\theta'}(a_t|s_t)$ off-policy

$$L_\pi = \sum_{(s_t,a_t)} min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right) A^{\theta'}(s_t,a_t)\right)$$

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

But we should update D as much as possible

- In each training iteration:

Learning D

Repeat k times

- Sample m examples $\{x^1, x^2, \dots, x^m\}$ from data distribution $P_{data}(x)$
- Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from the prior $P_{prior}(z)$
- Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}$, $\tilde{x}^i = G(z^i)$
- Update discriminator parameters θ_d to maximize
  - $\tilde{V} = \frac{1}{m}\sum_{i=1}^{m} logD(x^i) + \frac{1}{m}\sum_{i=1}^{m} log\left(1 - D(\tilde{x}^i)\right)$
  - $\theta_d \leftarrow \theta_d + \eta\nabla\tilde{V}(\theta_d)$

Learning G

Only Once

- Sample another m noise samples $\{z^1, z^2, \dots, z^m\}$ from the prior $P_{prior}(z)$
- Update generator parameters θ_g to minimize
  - $\tilde{V} = \cancel{\frac{1}{m}\sum_{i=1}^{m} logD(x^i)} + \frac{1}{m}\sum_{i=1}^{m} log\left(1 - D\left(G(z^i)\right)\right)$
  - $\theta_g \leftarrow \theta_g - \eta\nabla\tilde{V}(\theta_g)$

```
In [19]: def train_discriminator(real_images, opt_d):
           # Clear discriminator gradients
           opt_d.zero_grad()
```

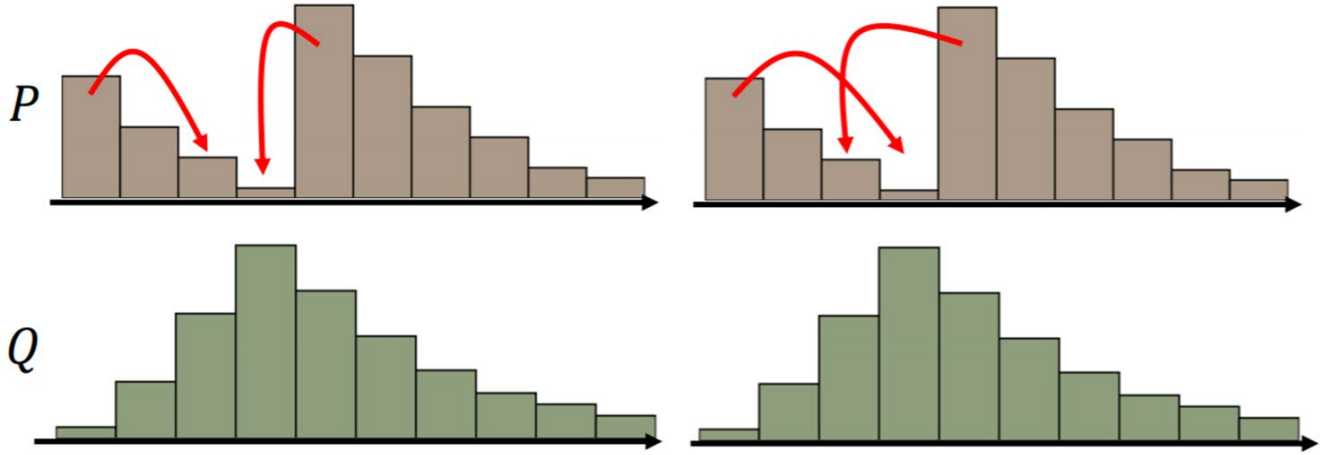
For loop for *k* times?

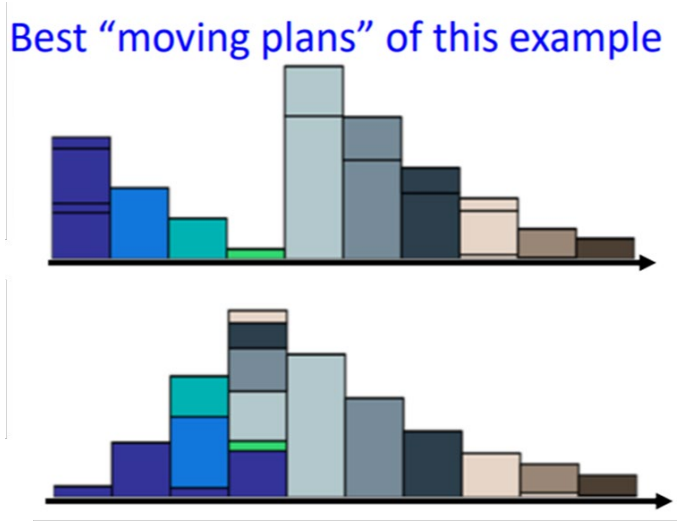
```
           # Pass real images through discriminator
           real_preds = discriminator(real_images)
           real_targets = torch.ones(real_images.size(0), 1, dev
           real_loss = F.binary_cross_entropy(real_preds, real_t
           real_score = torch.mean(real_preds).item()

           # Generate fake images
           latent = torch.randn(batch_size, latent_size, 1, 1, d
           fake_images = generator(latent.to(device))

           # Pass fake images through discriminator
           fake_targets = torch.zeros(fake_images.size(0), 1, de
           fake_preds = discriminator(fake_images)
           fake_loss = F.binary_cross_entropy(fake_preds, fake_t
           fake_score = torch.mean(fake_preds).item()

           # Update discriminator weights
           loss = real_loss + fake_loss
           loss.backward()
           opt_d.step()
           return loss.item(), real_score, fake_score
```

Reference: 李弘毅 GAN Lecture 4 (2018) https://youtu.be/DMA4MrNieWo

Wasserstein GAN (WGAN)

W-GAN: Earth move's distance



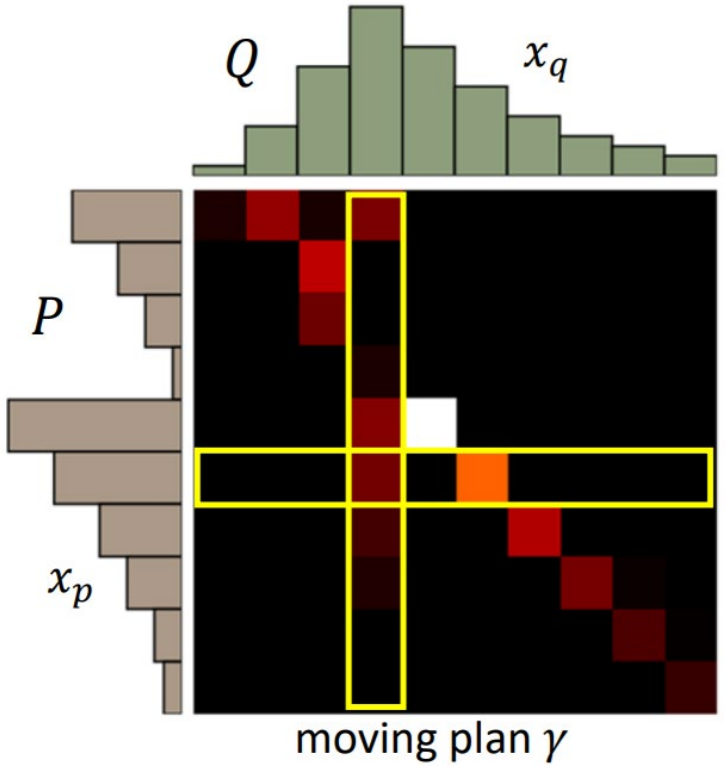


Average distance of a plan γ:

$$B(\gamma) = \sum_{x_p, x_q} \gamma(x_p, x_q) \|x_p - x_q\|$$

Earth Mover's Distance:

$$W(P, Q) = \min_{\gamma \in \Pi} B(\gamma)$$



Reference: 李弘毅 GAN Lecture 6 (2018) https://youtu.be/3JP-xuBJsyc

W-GAN

Evaluate wasserstein distance between P_{data} and P_G

$$V(G,D) = \max_{D \in 1-Lipschitz} \{E_{x \sim P_{data}}[D(x)] - E_{x \sim P_G}[D(x)]\}$$

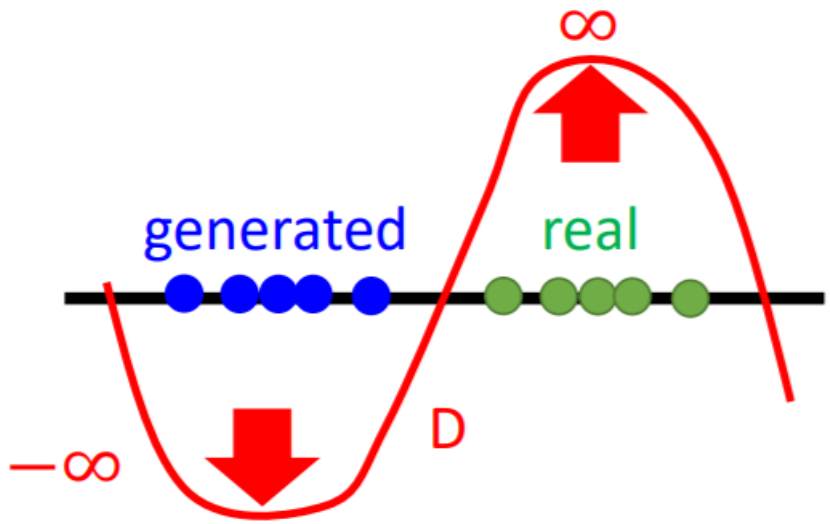
D has to be smooth enough.

Lipschitz Function

$$\|f(x_1) - f(x_2)\| \leq K\|x_1 - x_2\|$$

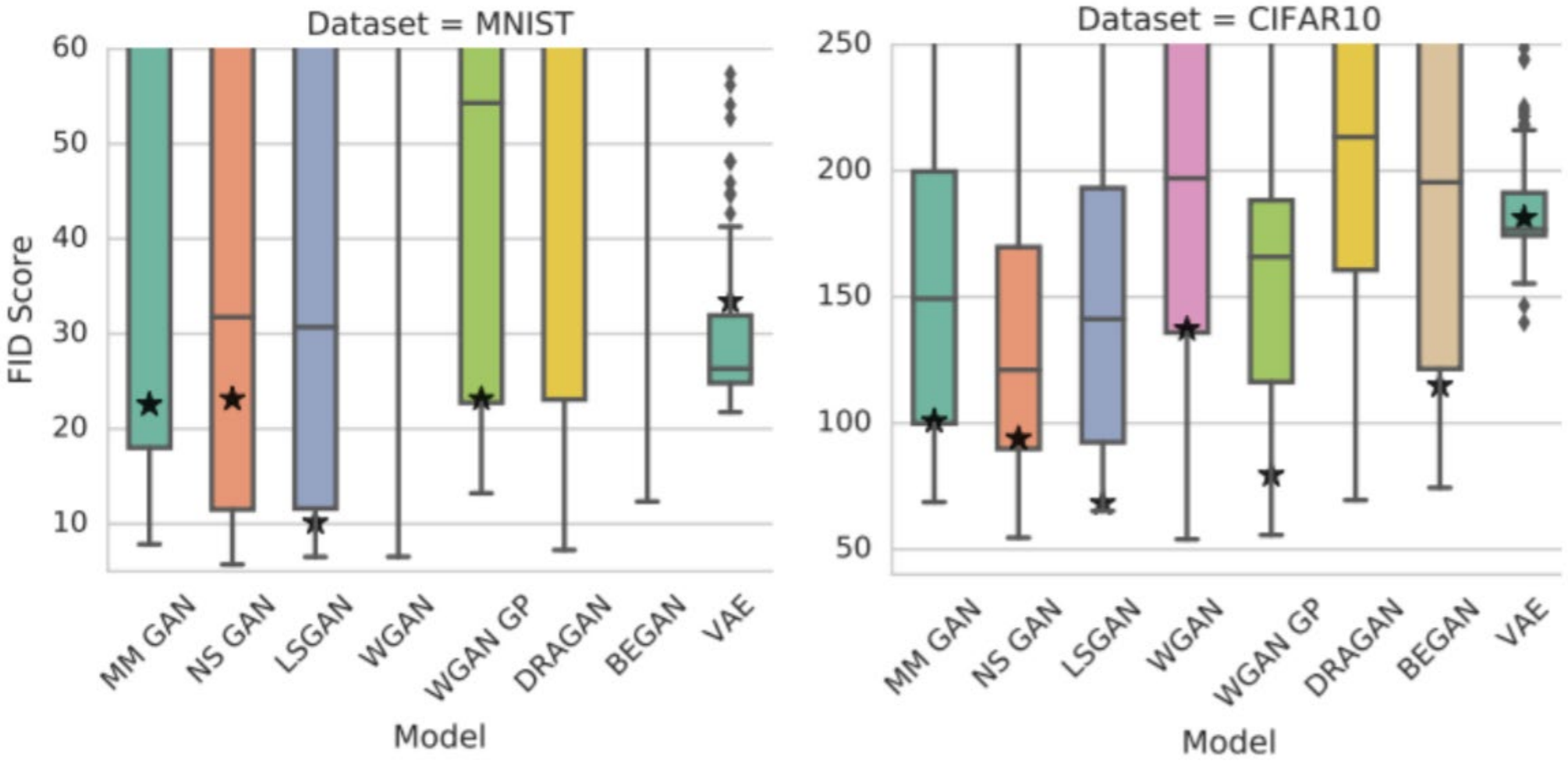
Output change / Input change

K=1 for "$1-Lipschitz$"



Reference: 李弘毅 GAN Lecture 6 (2018) https://youtu.be/3JP-xuBJsyc

GAN is sensitive to hyper-parameter tuning and its performance range is large. Different GANs' performances are similar



Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw