Generative Adversarial Network (GAN)

Practice

- Open “8.1. GAN.ipynb"

Generator takes a feature vector and generates an output image (using deconvolution to do up-sampling)

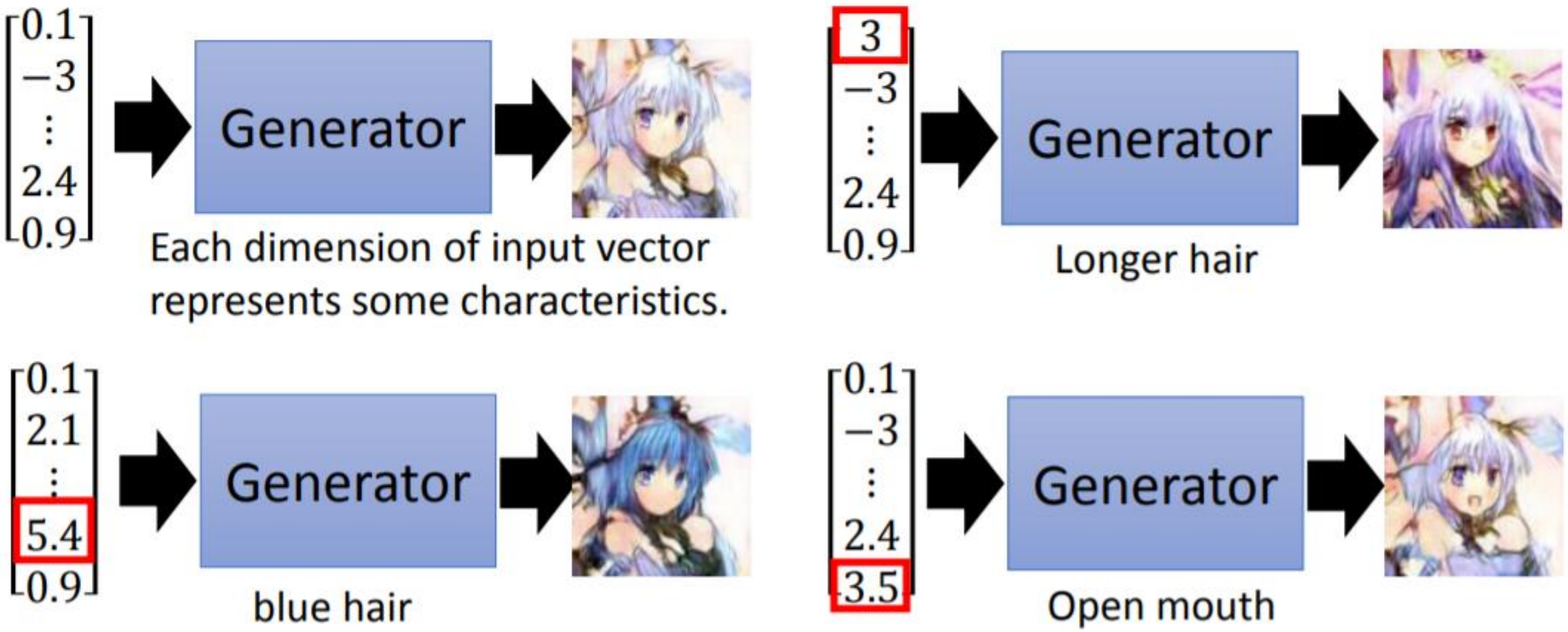
[10]:
```
latent_size= 128
```

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

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1, padding=0, bias=False)
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    # out: 128 x 16 x 16

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

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

```
[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, padding=0, bias=False)
          nn.BatchNorm2d(512),
```

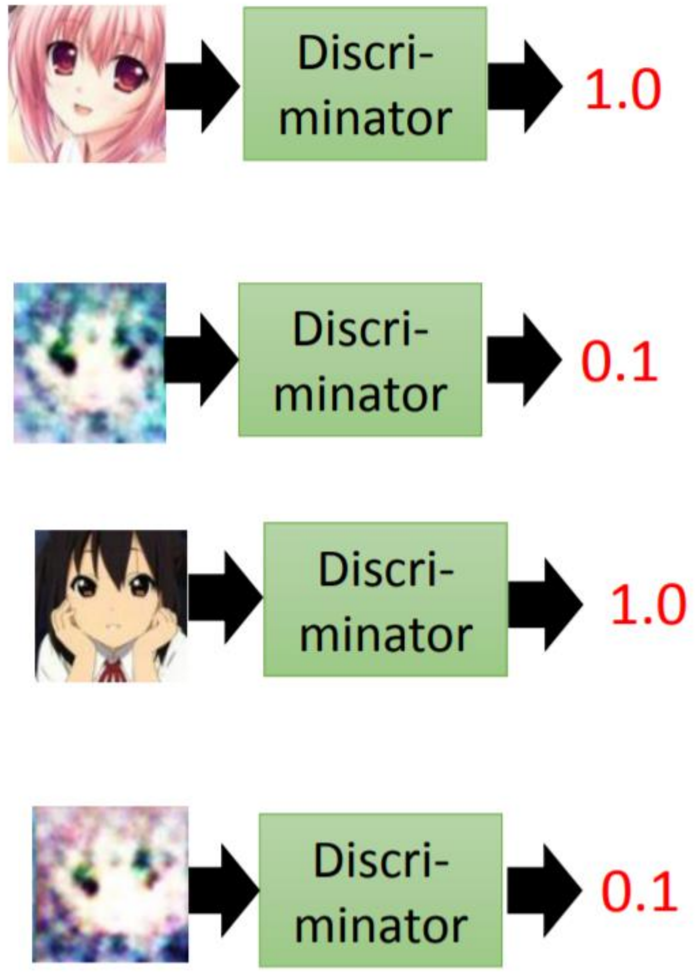


Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw

Discriminator takes an input image and tells whether it is a true image or not

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

          nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=Fals
          nn.BatchNorm2d(64),
          nn.LeakyReLU(0.2, inplace=True),
          # out: 64 x 64 x 64

          nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=Fa
          nn.BatchNorm2d(128),
          nn.LeakyReLU(0.2, inplace=True),
          # out: 128 x 32 x 32

          nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=F
          nn.BatchNorm2d(256),
          nn.LeakyReLU(0.2, inplace=True),
          # out: 256 x 16 x 16

          nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=F
          nn.BatchNorm2d(512),
          nn.LeakyReLU(0.2, inplace=True),
          # out: 512 x 8 x 8

          nn.Conv2d(512, 1024, kernel_size=4, stride=2, padding=1, bias=
          nn.BatchNorm2d(1024),
          nn.LeakyReLU(0.2, inplace=True),
          # out: 1024 x 4 x 4

          nn.Conv2d(1024, 1, kernel_size=4, stride=1, padding=0, bias=Fa
          # out: 1 x 1 x 1
```

Larger value means real, smaller value means fake



Reference: 李弘毅 GAN Lecture 1 (2018)

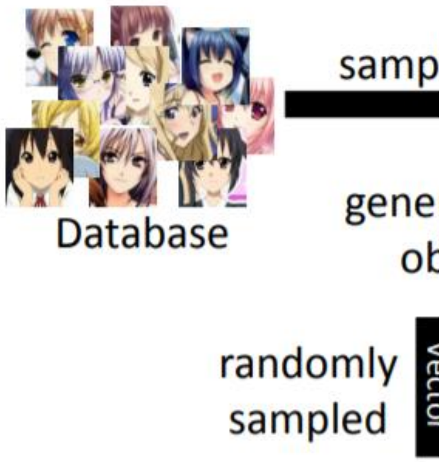
Step1 – Fix G and train D

- 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

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

Update discriminator parameters θ_d to maximize

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

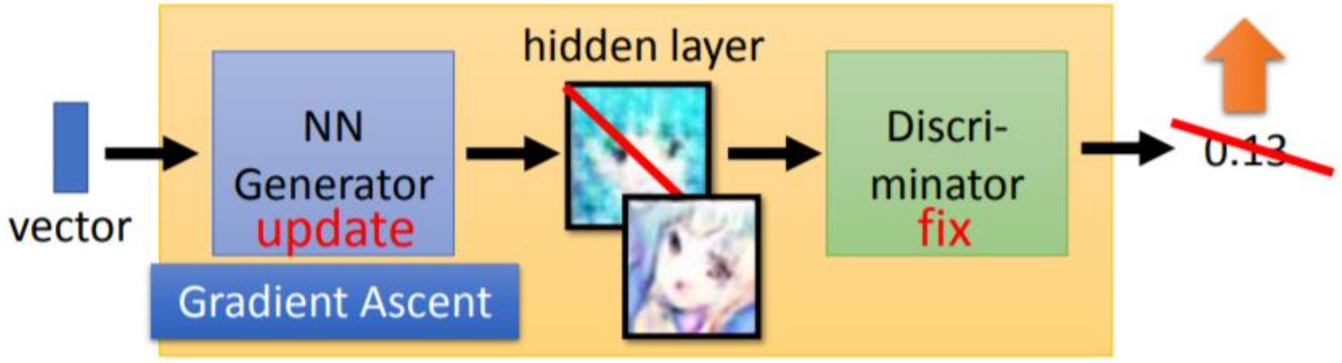
[19]:

```
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
```

Step2 – Fix D and train G

Step 2: Fix discriminator D, and update generator G

Generator learns to “fool” the discriminator



large network

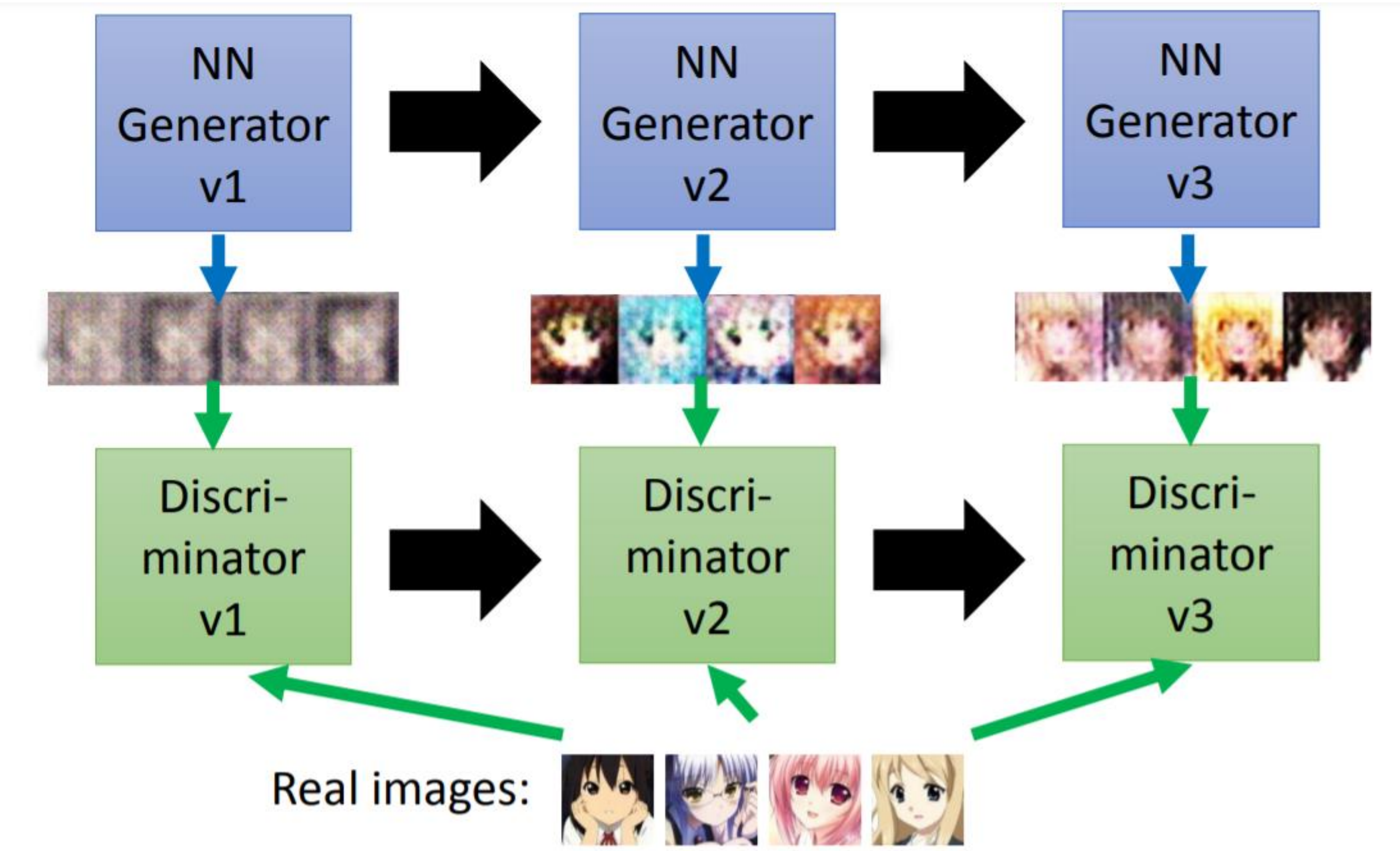
[35]:
```
for epoch in range(epochs):
  if(epoch % 10 ==0):
    print(epoch, end=",")
  for real_images, _ in train_dl:
    # Train discriminator
    loss_d, real_score, fake_score = train_discriminator(real_images.
    # Train generator
    loss_g = train_generator(opt_g)
```

Train generator

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

[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



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

Visualize the fake images from *G* during the evolution process

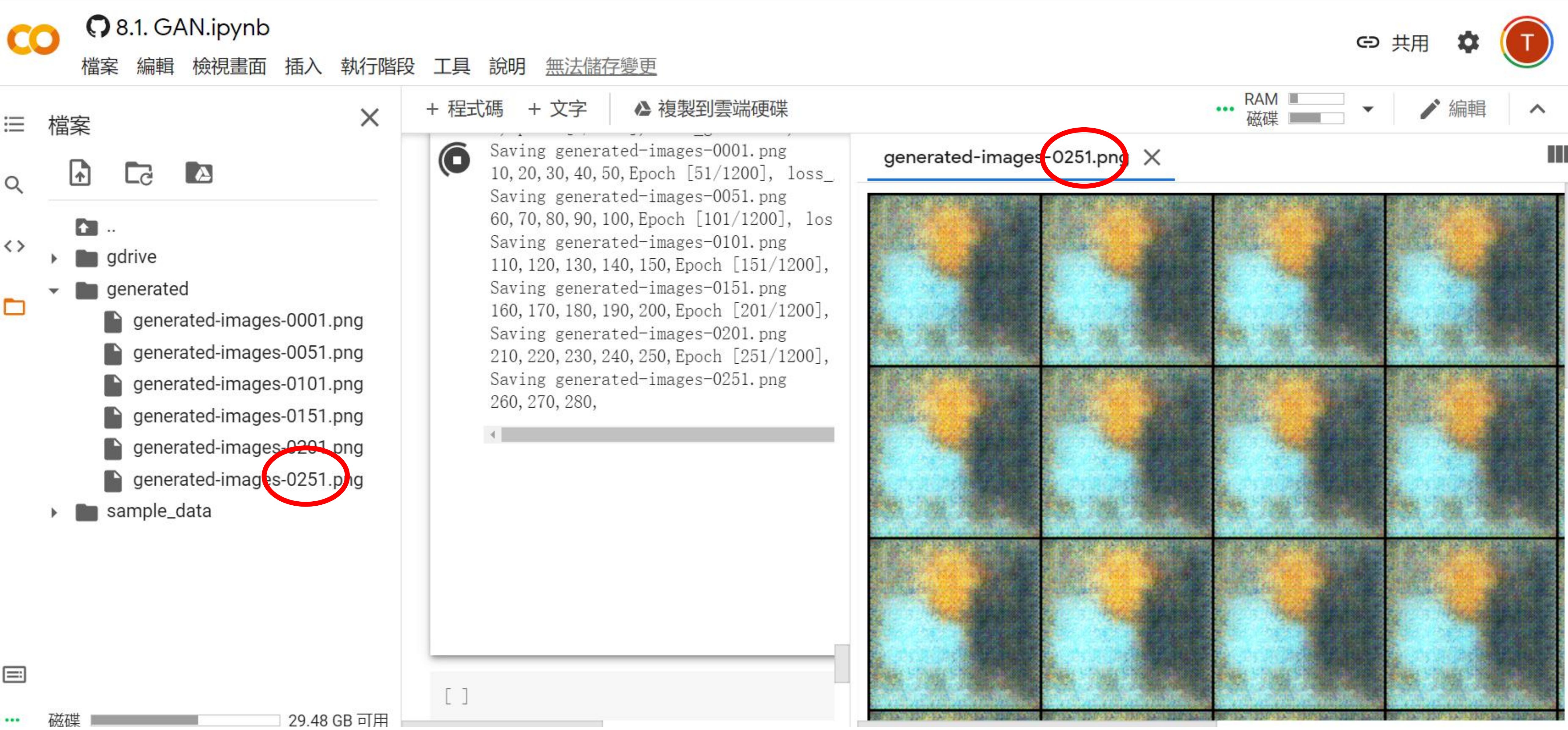
```
[17]: sample_dir = 'generated'
      os.makedirs(sample_dir, exist_ok=True)
```

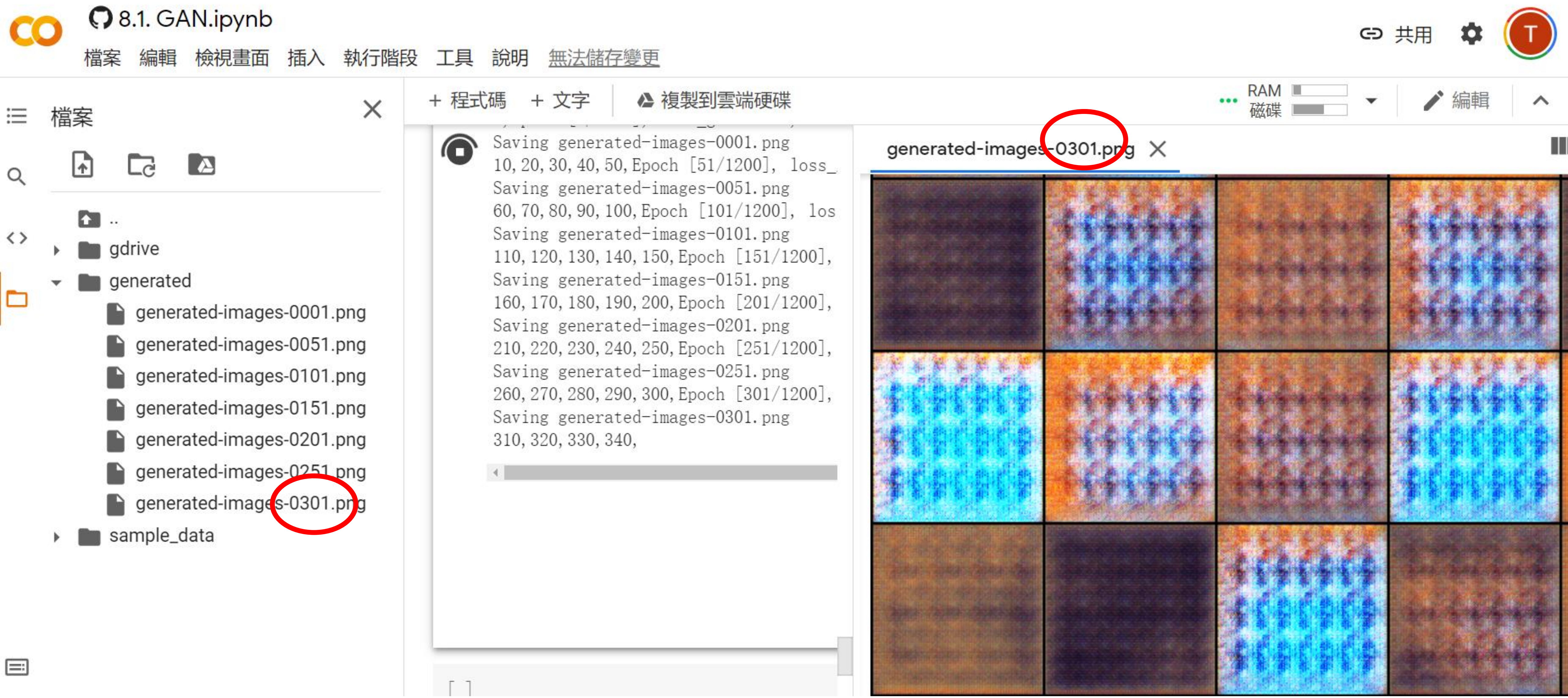
```
[34]: fixed_latent = torch.randn(64, latent_size,
      # used to generate saved images
```

```
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
T
檔案
+ 程式碼 + 文字
複製到雲端硬碟
RAM
磁碟
編輯
..
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

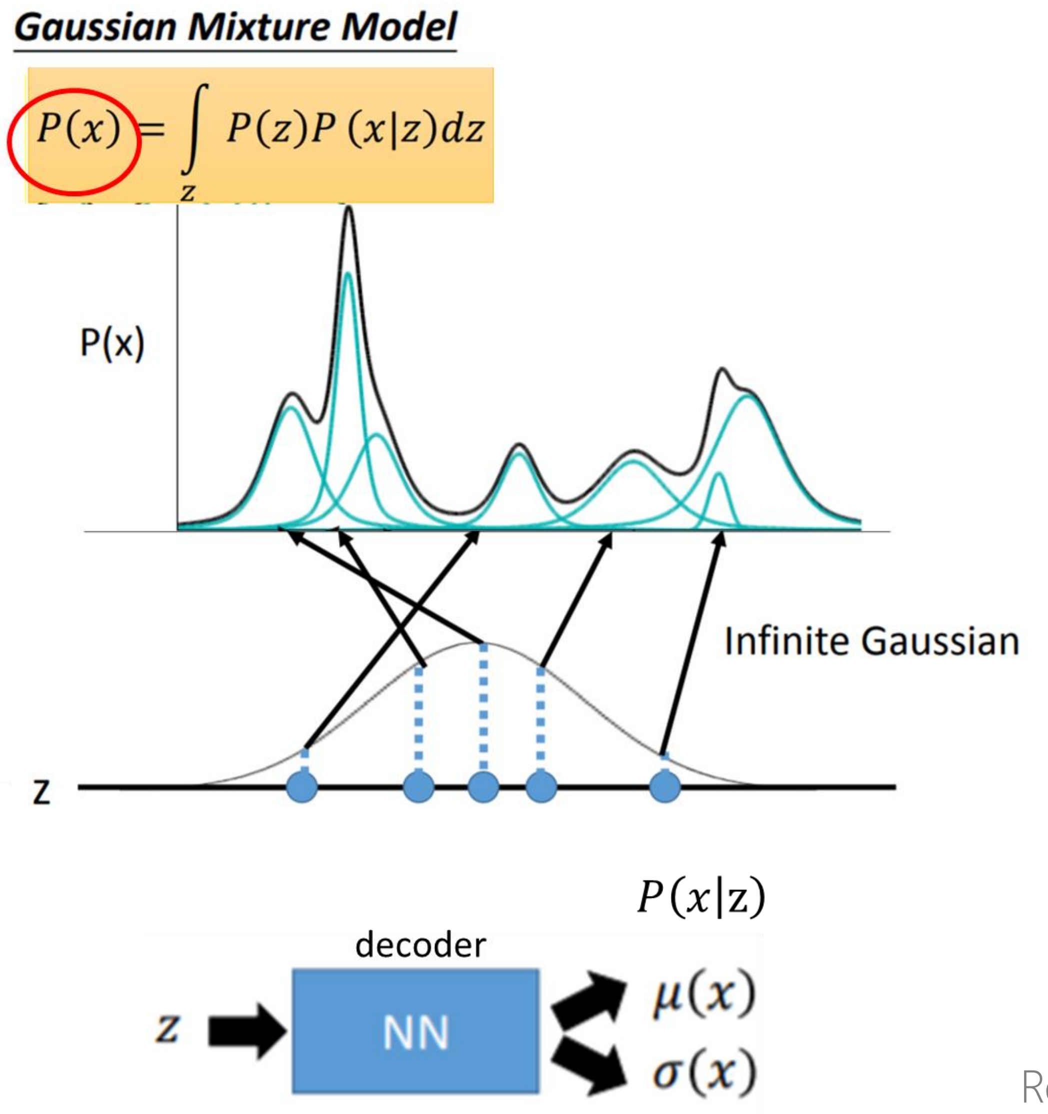

8.1. GAN.ipynb
檔案 編輯 檢視畫面 插入 執行階段 工具 說明 無法儲存變更
共用
+ 程式碼 + 文字 複製到雲端硬碟
RAM
磁碟
編輯
檔案
..
gdrive
generated
generated-images-0001.png
generated-images-0051.png
generated-images-0101.png
generated-images-0151.png
generated-images-0201.png
generated-images-0251.png
sample_data
磁碟 29.48 GB 可用
Saving generated-images-0001.png
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
T
+ 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
generated-images-0301.png
sample_data
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

Practice

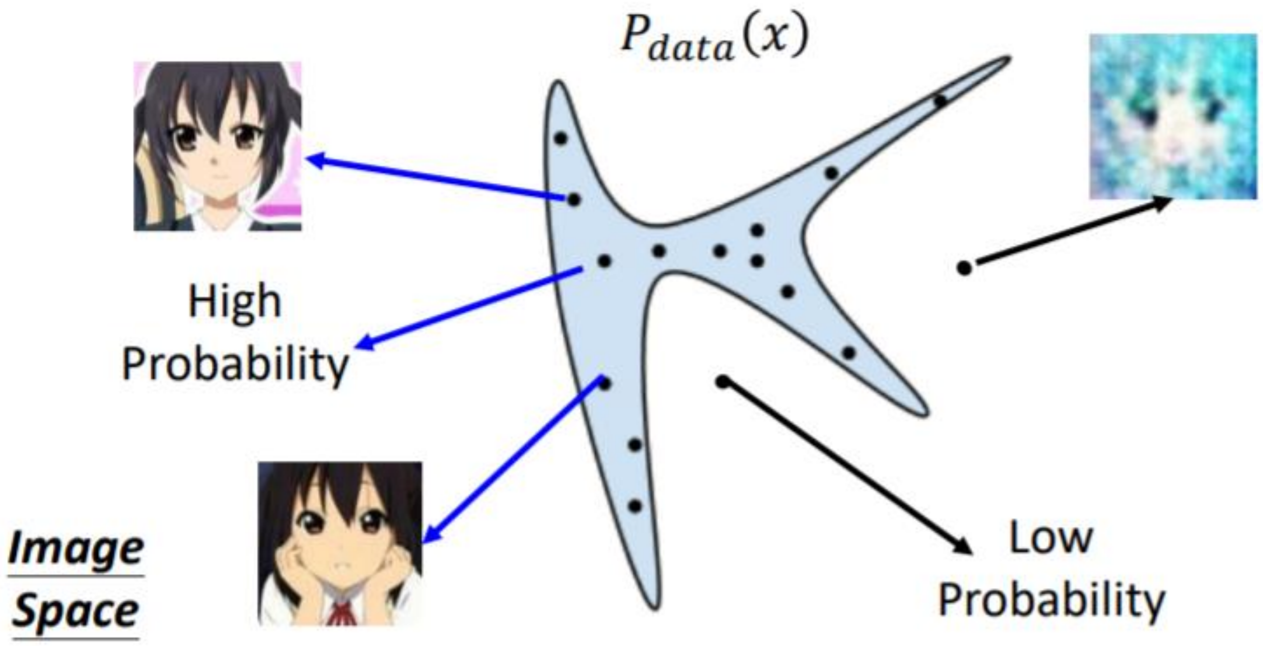
- Run “8.1. GAN.ipynb" in Colab. Try to generate Tom&Jerry

The generator can be modeled 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

Based on the distribution model P_G, we want to maximize the likelihood of observing the training images x_1, x_2, …, x_m

- 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

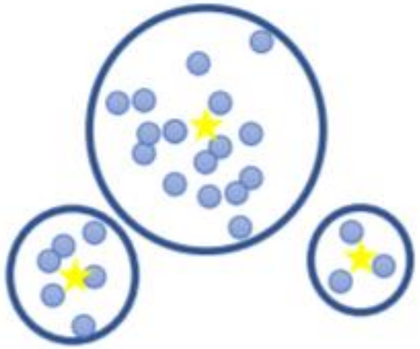
Sample $\{x^1, x^2, \dots, x^m\}$ from $P_{data}(x)$

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

Review: Based on the distribution model P_G, we want to maximize the likelihood of observing the training images x_1, x_2, x_3, …, x_m

HW6(2) – VAE

Maximizing Likelihood

P(z) is normal distribution

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

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

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

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

Maximizing the likelihood of the observed x

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

HW4, 5 – MLP, CNN classifier

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

Compare with RL PPO: Based on the distribution model $P_\theta(\tau)$, We want to maximize the likelihood of obtaining best rewards

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

See RL training video in my Github/Intelligent-Robot

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

$$R(\tau) = \sum_{t=1}^{T} r_t$$

The reward of doing this trajectory under current policy

$$\bar{R}_\theta = \sum R(\tau)\, p_\theta(\tau) = E_{\tau \sim p_\theta(\tau)}[R(\tau)]$$

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

Maximum likelihood estimation = minimum KL divergence

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

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

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

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

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

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

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

How to define a general P_G?

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

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

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

Compare with VAE and PPO: : Maximum likelihood estimation = minimum KL divergence

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

Variational AE

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

Minimize

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

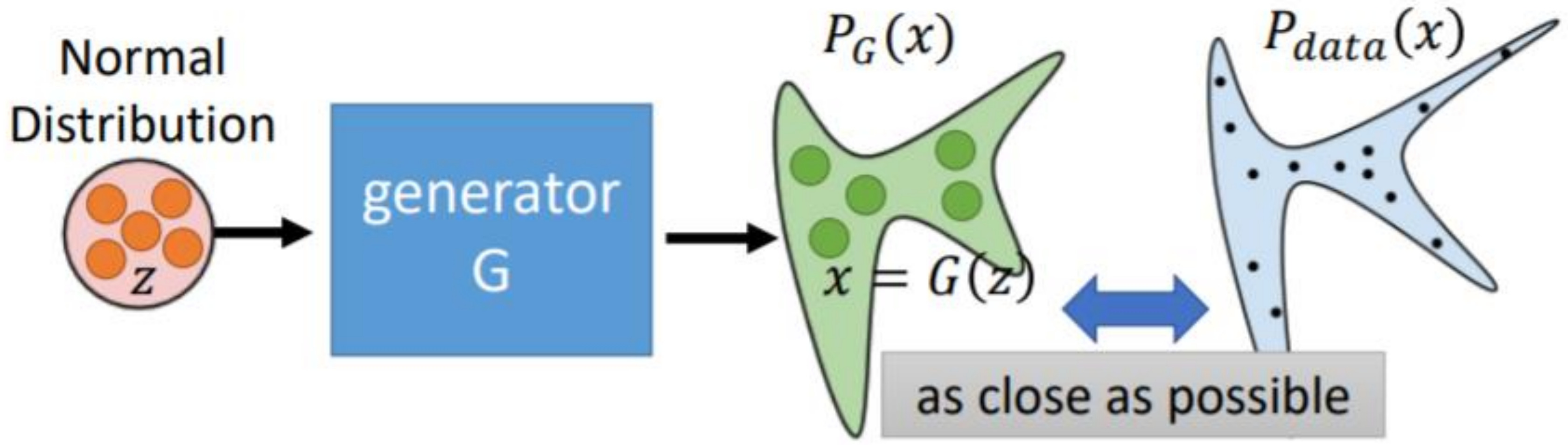
(Refer to the Appendix B of the original VAE paper)

Proximal policy optimization (PPO)

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

How to define a general P_G? – use a generator *NN*

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



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

Divergence between distributions P_G and P_{data}

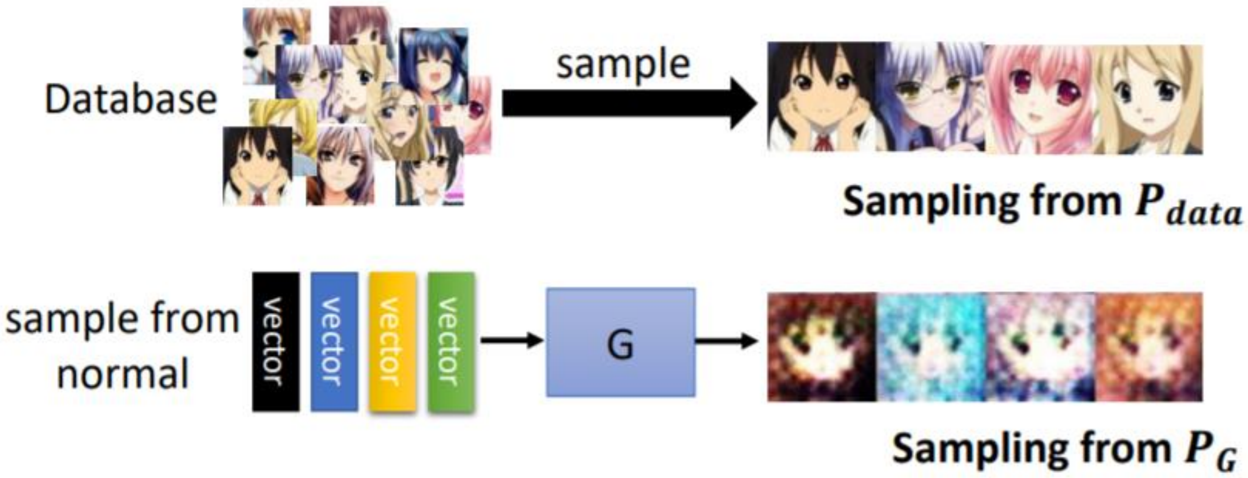
How to compute the divergence?

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

How to compute the divergence? – train a discriminator *NN* to compute the divergence

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

Although we do not know the distributions of P_G and P_{data}, we can sample from them.




```
[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)
```

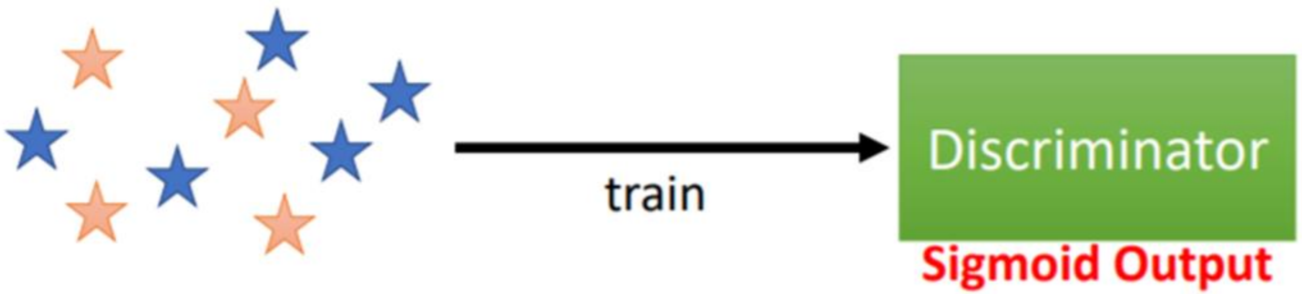
```
[19]: def train_discriminator(real_images, opt_d):
        # Clear discriminator gradients
        opt_d.zero_grad()

        # 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))
```

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

Train a discriminator *NN* to compute the divergence

★ : data sampled from P_{data}

☆ : data sampled from P_G



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: $D^* = arg\,\boxed{\max_D V(D,G)}$

[Goodfellow, et al., NIPS, 2014]

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

    # Pass real images through discrim
    real_preds = discriminator(real_im
    real_targets = torch.ones(real_ima
    real_loss = F.binary_cross_entropy
    real_score = torch.mean(real_preds

    # Generate fake images
    latent = torch.randn(batch_size, l
    fake_images = generator(latent.to(

    # Pass fake images through discrim
    fake_targets = torch.zeros(fake_im
    fake_preds = discriminator(fake_im
    fake_loss = F.binary_cross_entropy
    fake_score = torch.mean(fake_preds

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
```

Review: Loss function for binary classifier

Using the example objective function is exactly the same as training a binary classifier.

The discriminator is a binary classifier (logistic regression) to classify real vs fake

Cross entropy:

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

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

(G is fixed)

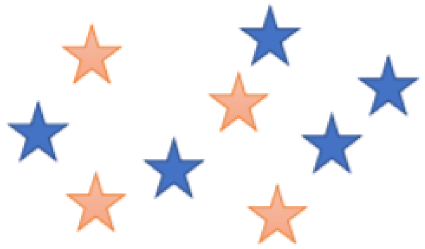
Adversarial neural networks G and D

Difficult to train!!

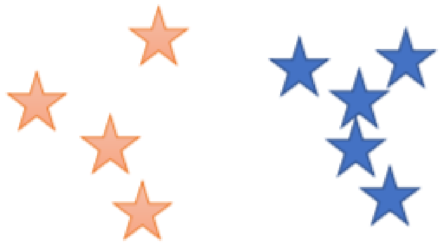
(1) Train generator – max. likelihood = min. KL divergence between P_G and P_{data}

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

(2) Train discriminator – max. divergence between P_G and P_{data}



small divergence



large divergence

★ : data sampled from P_{data}

☆ : data sampled from P_G

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

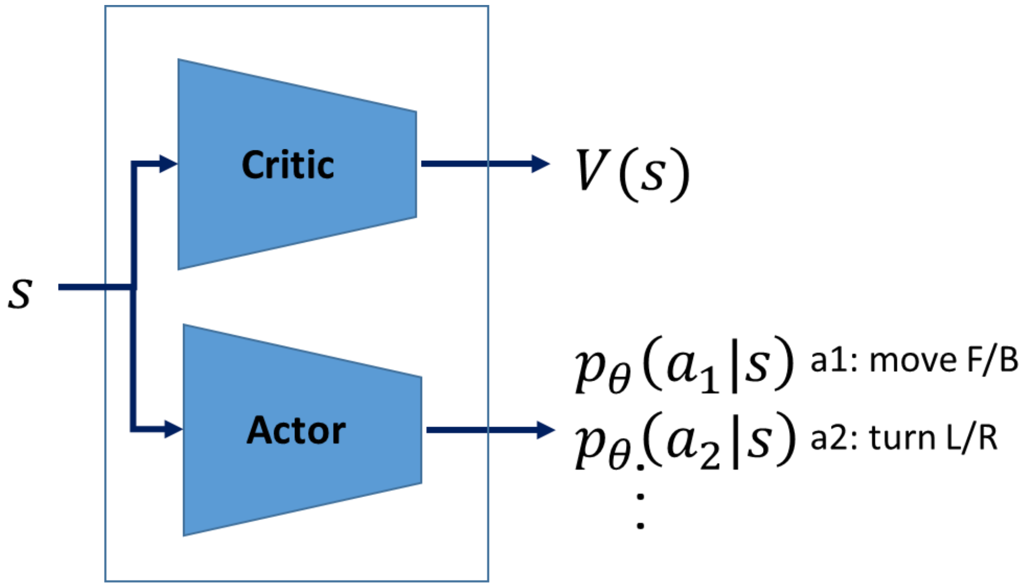
(G is fixed)

Training:

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

Compare with RL PPO: Train two neural networks actor and critic together

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



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

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

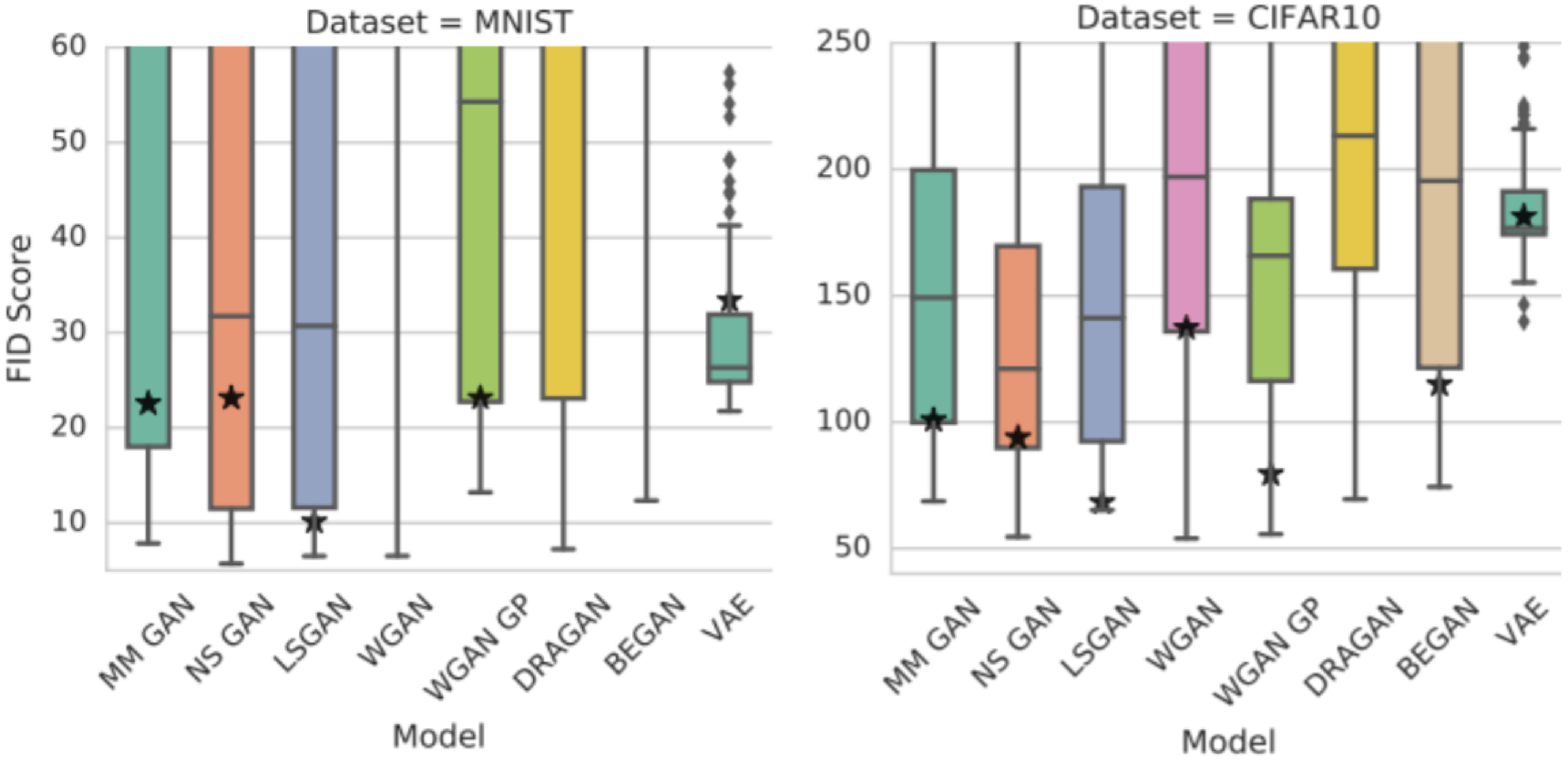
(2) Critic – Learns the expected value of the long-term reward.

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

The objective function of *D* is related to JS divergence

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

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