Wasserstein GAN (WGAN)

Arjovsky, M., Chintala, S., & Bottou, L. (2017, July). Wasserstein generative adversarial networks. In *International conference on machine learning* (pp. 214-223). PMLR.

- The *Total Variation* (TV) distance

$$\delta(\mathbb{P}_r, \mathbb{P}_g) = \sup_{A \in \Sigma} |\mathbb{P}_r(A) - \mathbb{P}_g(A)| \ .$$

- The *Kullback-Leibler* (KL) divergence

$$KL(\mathbb{P}_r \| \mathbb{P}_g) = \int \log\left(\frac{P_r(x)}{P_g(x)}\right) P_r(x) d\mu(x) \ ,$$

where both $\mathbb{P}_r$ and $\mathbb{P}_g$ are assumed to admit densities with respect to a same measure μ defined on $\mathcal{X}$.[2] The KL divergence is famously assymetric and possibly infinite when there are points such that $P_g(x) = 0$ and $P_r(x) > 0$.

- The *Jensen-Shannon* (JS) divergence

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r \| \mathbb{P}_m) + KL(\mathbb{P}_g \| \mathbb{P}_m) \ ,$$

where $\mathbb{P}_m$ is the mixture $(\mathbb{P}_r + \mathbb{P}_g)/2$. This divergence is symmetrical and always defined because we can choose $\mu = \mathbb{P}_m$.

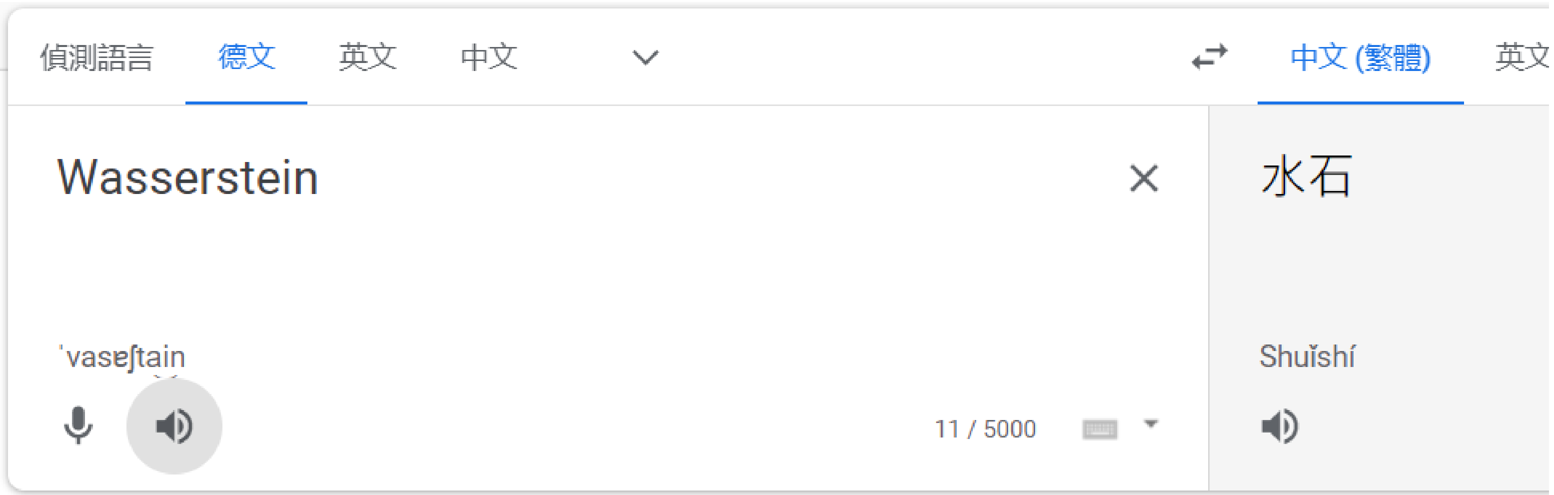
- The *Earth-Mover* (EM) distance or Wasserstein-1

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}\big[\, \|x - y\| \,\big] \ , \quad (1)$$

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ is the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$. Intuitively, $\gamma(x, y)$ indicates how much "mass" must be transported from x to y in order to transform the distributions $\mathbb{P}_r$ into the distribution $\mathbb{P}_g$. The EM distance then is the "cost" of the optimal transport plan.

Arjovsky, M., Chintala, S., & Bottou, L. (2017, July). Wasserstein generative adversarial networks. In *International conference on machine learning* (pp. 214-223). PMLR.

The name "Wasserstein distance" was coined by R. L. Dobrushin in 1970, after learning of it in the work of Russian mathematician Leonid Vaseršteĭn 1969, however the metric was first defined by Leonid Kantorovich in The Mathematical Method of Production Planning and Organization (Russian original 1939) in the context of optimal transport planning of goods and materials. Some scholars thus encourage use of the terms "Kantorovich metric" and "Kantorovich distance". Most English-language publications use the German spelling "Wasserstein" (attributed to the name "Vaseršteĭn" being of German origin).



https://en.wikipedia.org/wiki/Wasserstein_metric

JS divergence is not suitable

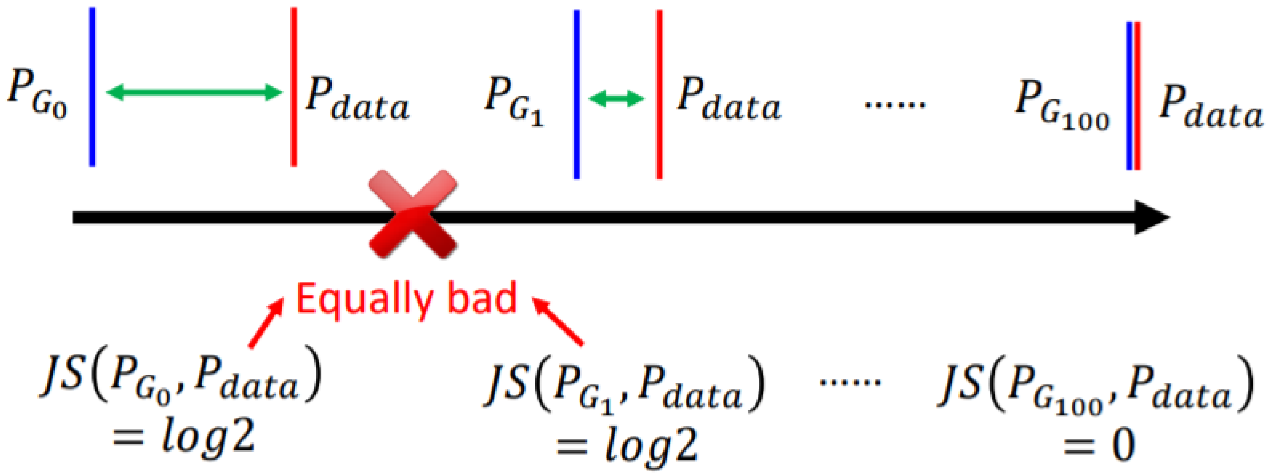
- In most cases, P_G and P_{data} are not overlapped.

JS divergence is log2 if two distributions do not overlap.

Intuition: If two distributions do not overlap, binary classifier achieves 100% accuracy

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

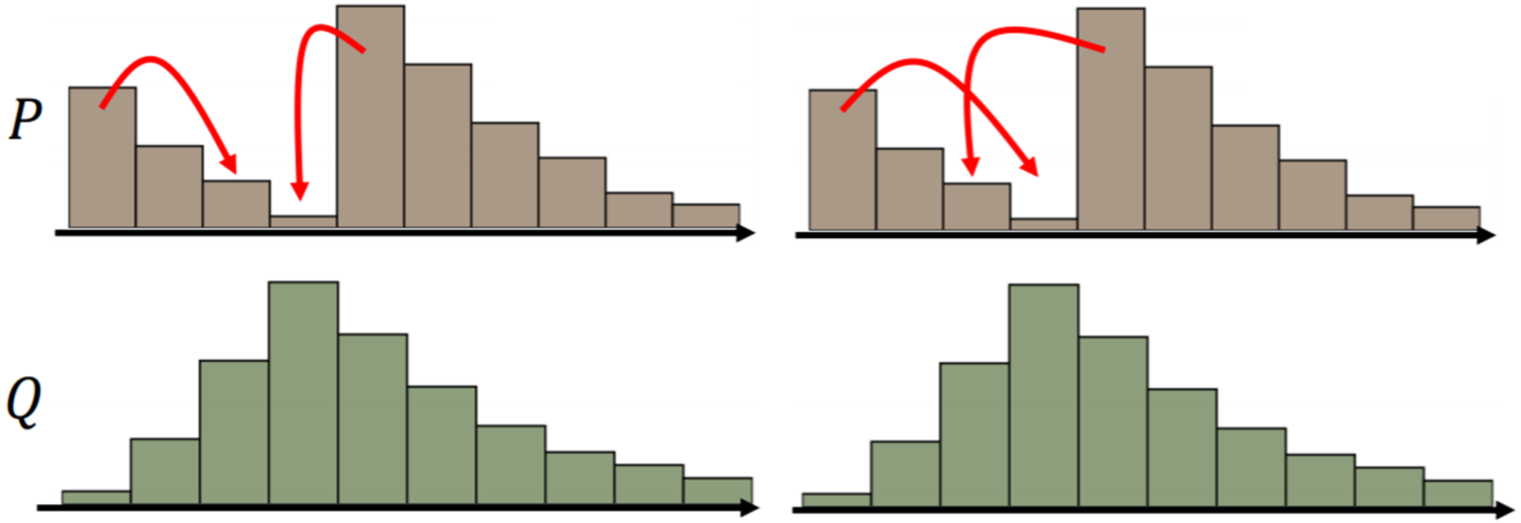
$2JSD(P_{data}||P_G)$ Jensen-Shannon divergence

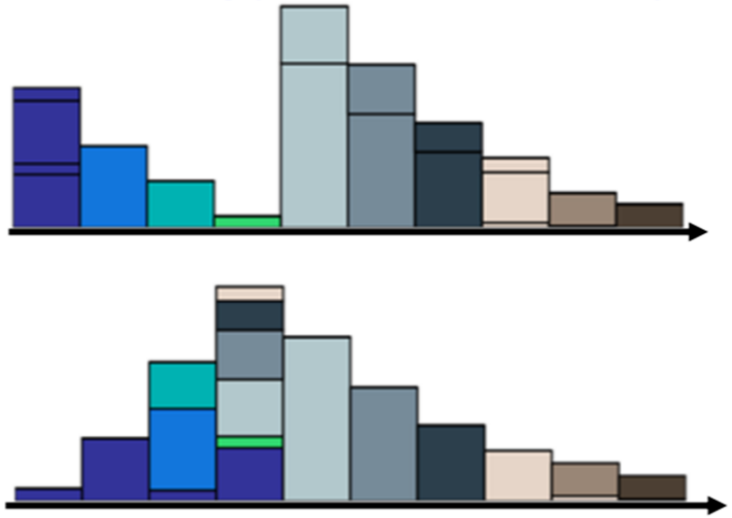


Reference: 李弘毅 GAN Lecture 6 (2018) https://youtu.be/3JP-xuBJsyc

W-GAN: Earth move's distance

- Considering one distribution P as a pile of earth, and another distribution Q as the target
- The average distance the earth mover has to move the earth.



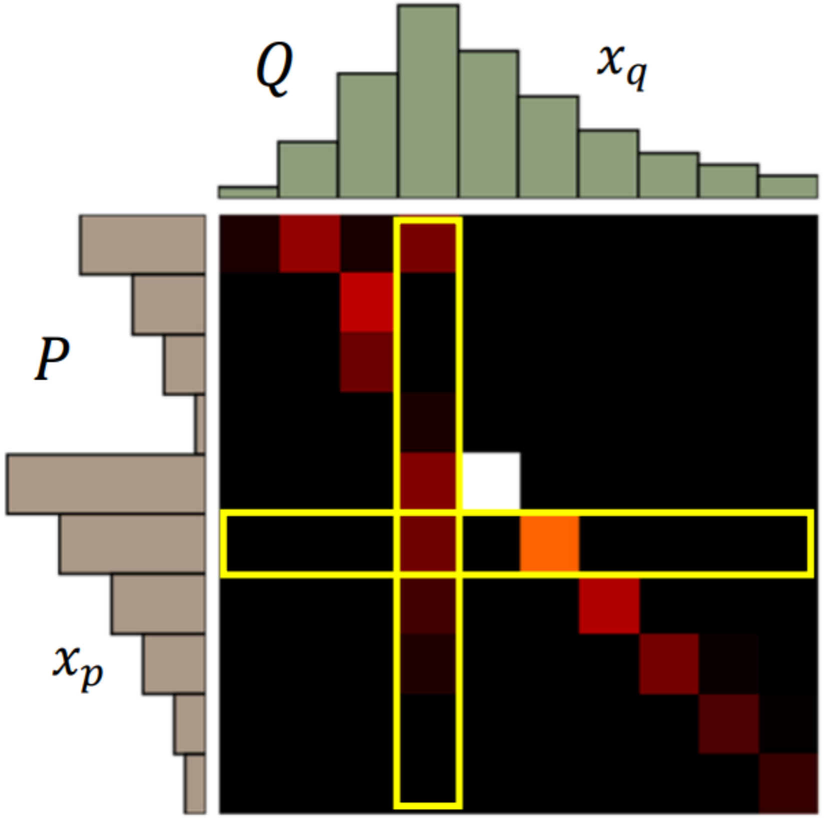


Reference: 李弘毅 GAN Lecture 6 (2018) https://youtu.be/3JP-xuBJsyc

W-GAN: Earth move's distance

A "moving plan" is a matrix

The value of the element is the amount of earth from one position to another.



moving plan γ

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

The discriminator that measures Wasserstein distance can be trained by adding Lipschitz continuity constraint to the objective function

GAN

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

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

WGAN

$$D^* = arg \max_{D \in 1-Lipschitz} V(D,G)$$

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

The discriminator that measures Wasserstein distance can be trained by adding Lipschitz continuity constraint to the objective function

Lipschitz continuity

From Wikipedia, the free encyclopedia

In mathematical analysis, **Lipschitz continuity**, named after Rudolf Lipschitz, is a strong form of uniform continuity for functions. Intuitively, a Lipschitz continuous function is limited in how fast it can change: there exists a real number such that, for every pair of points on the graph of this function, the absolute value of the slope of the line connecting them is not greater than this real number; the smallest such bound is called the *Lipschitz constant* of the function (or *modulus of uniform continuity*). For instance, every function that has bounded first derivatives is Lipschitz continuous.[1]

https://en.wikipedia.org/wiki/Lipschitz_continuity

The discriminator that measures Wasserstein distance can be trained by adding Lipschitz continuity constraint to the objective function

Evaluate wasserstein distance between P_{data} and P_G

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

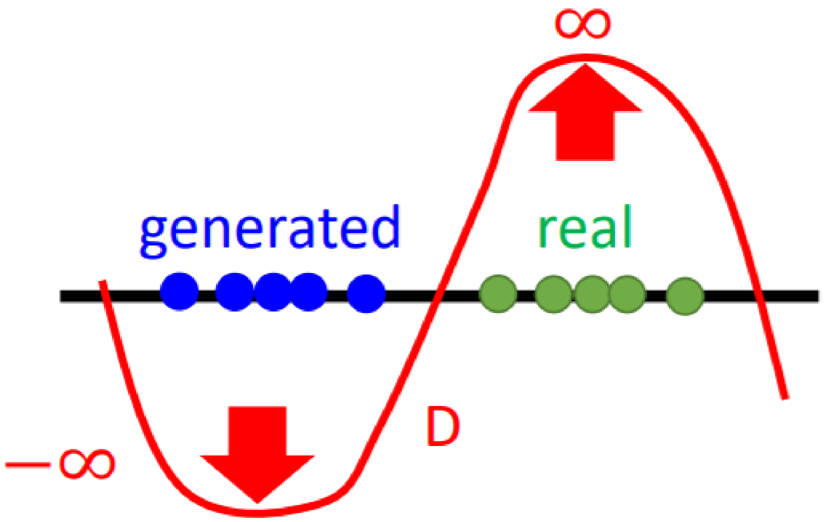
D has to be smooth enough.

Lipschitz Function

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

Output change   Input change

K=1 for "$1 - Lipschitz$"



Use weight clipping to solve the constrained optimization

$$D^* = arg \max_{D \in 1-Lipschitz} V(D,G)$$

Weight Clipping [Martin Arjovsky, et al., arXiv, 2017]

Force the parameters w between c and -c

After parameter update, if w > c, w = c;

if w < -c, w = -c

Use weight clipping to solve constrained optimization in PPO

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

W-GAN

Algorithm of WGAN

No sigmoid for the output of D

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

Weight clipping / Gradient Penalty …

Learning G

Only Once

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

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

Update discriminator parameters θ_d to maximize

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

Sample another m noise s … } from prior P … (z)

Weight clipping / Gradient Penalty …

GAN

```
[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), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)
    real_score = torch.mean(real_preds).item()

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

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)
    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
```

WGAN

```
# Pass real images through discriminator
real_preds = discriminator(real_images)
real_loss = torch.mean(real_preds)
```

```
# Pass fake images through discriminator
fake_preds = discriminator(fake_images)
fake_loss = torch.mean(fake_preds)

# Update discriminator weights
loss = fake_loss - real_loss
```

```
# Parameter(Weight) Clipping for K-Lipshitz constraint
for p in discriminator.parameters():
  p.data.clamp_(-0.01, 0.01)
return loss.item()
```

No sigmoid for the output of D

GAN

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

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    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=False),
    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=False),
    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=False),
    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=False),
    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=False),
    # out: 1 x 1 x 1

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

WGAN

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

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    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=False),
    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=False),
    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=False),
    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=False),
    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=False),
    # out: 1 x 1 x 1

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

- $\tilde{V} = \cancel{\frac{1}{m}\sum_{i=1}^{m} logD(x^i)} - \frac{1}{m}\sum_{i=1}^{m} D\left(G(z^i)\right)$

GAN

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

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

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

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

    return loss.item()
```

WGAN

```python
# Try to fool the discriminator
preds = discriminator(fake_images)
loss = -torch.mean(preds)
```

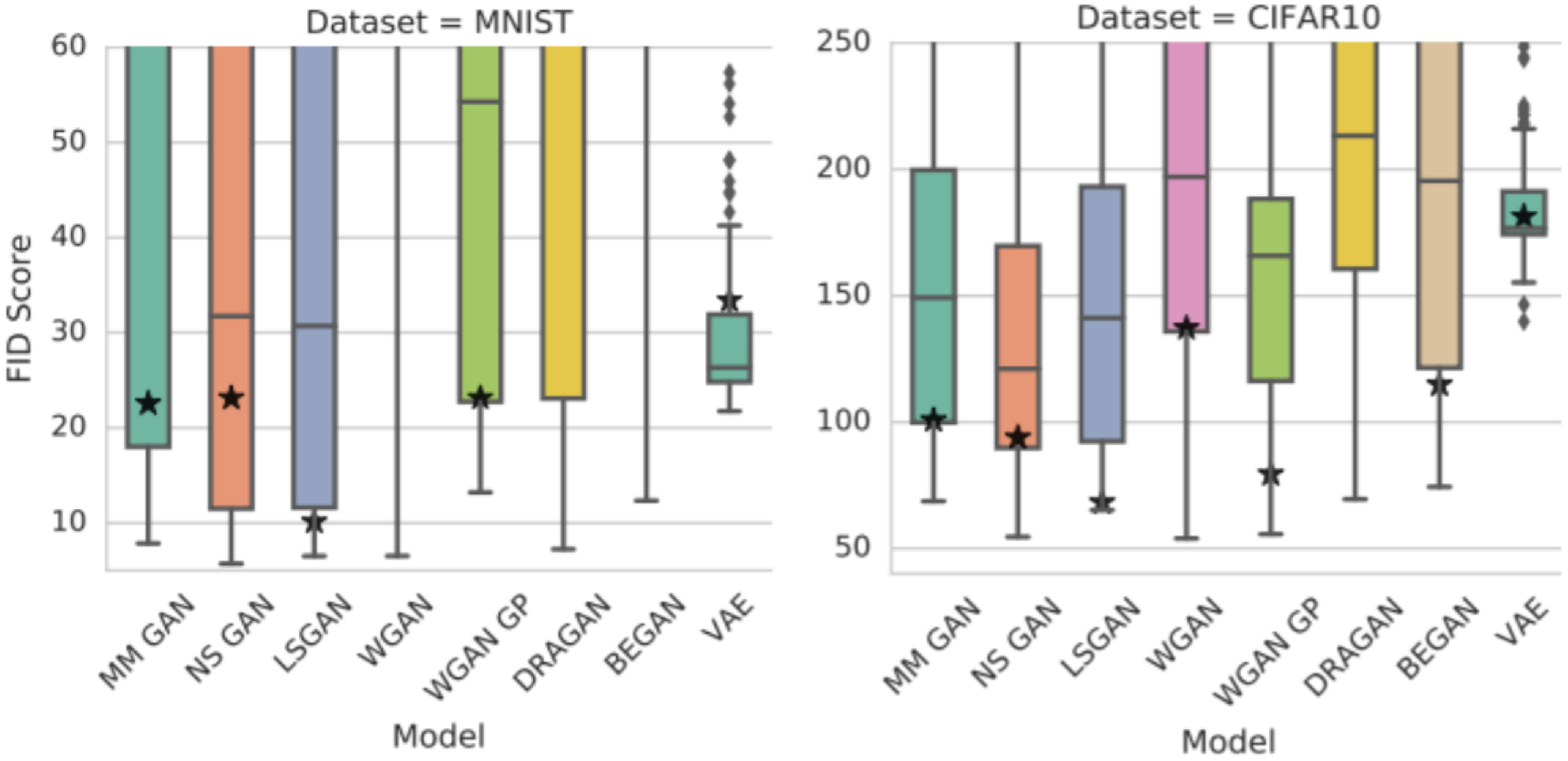
Practice

- Open “8.2. WGAN.ipynb"

HW7 (2)

- Use your own images, e.g., facial expression to train a WGAN.
- 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.

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

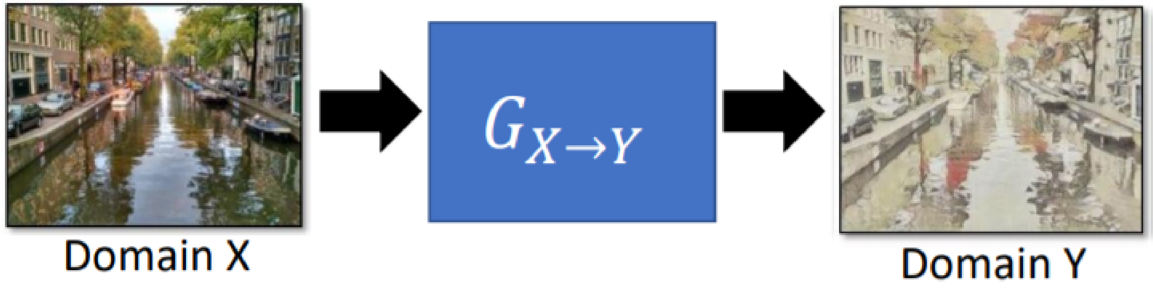


Reference: 李弘毅 GAN Lecture 1 (2018): https://youtu.be/DQNNMiAP5lw

Unsupervised conditional generation

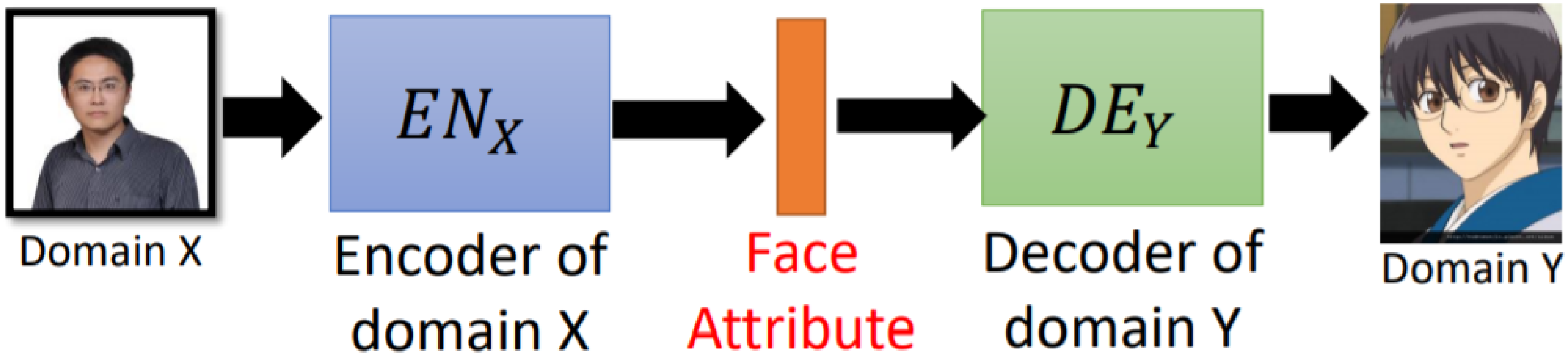
Two approaches for unsupervised conditional generation

- Approach 1: Direct Transformation



For texture or color change

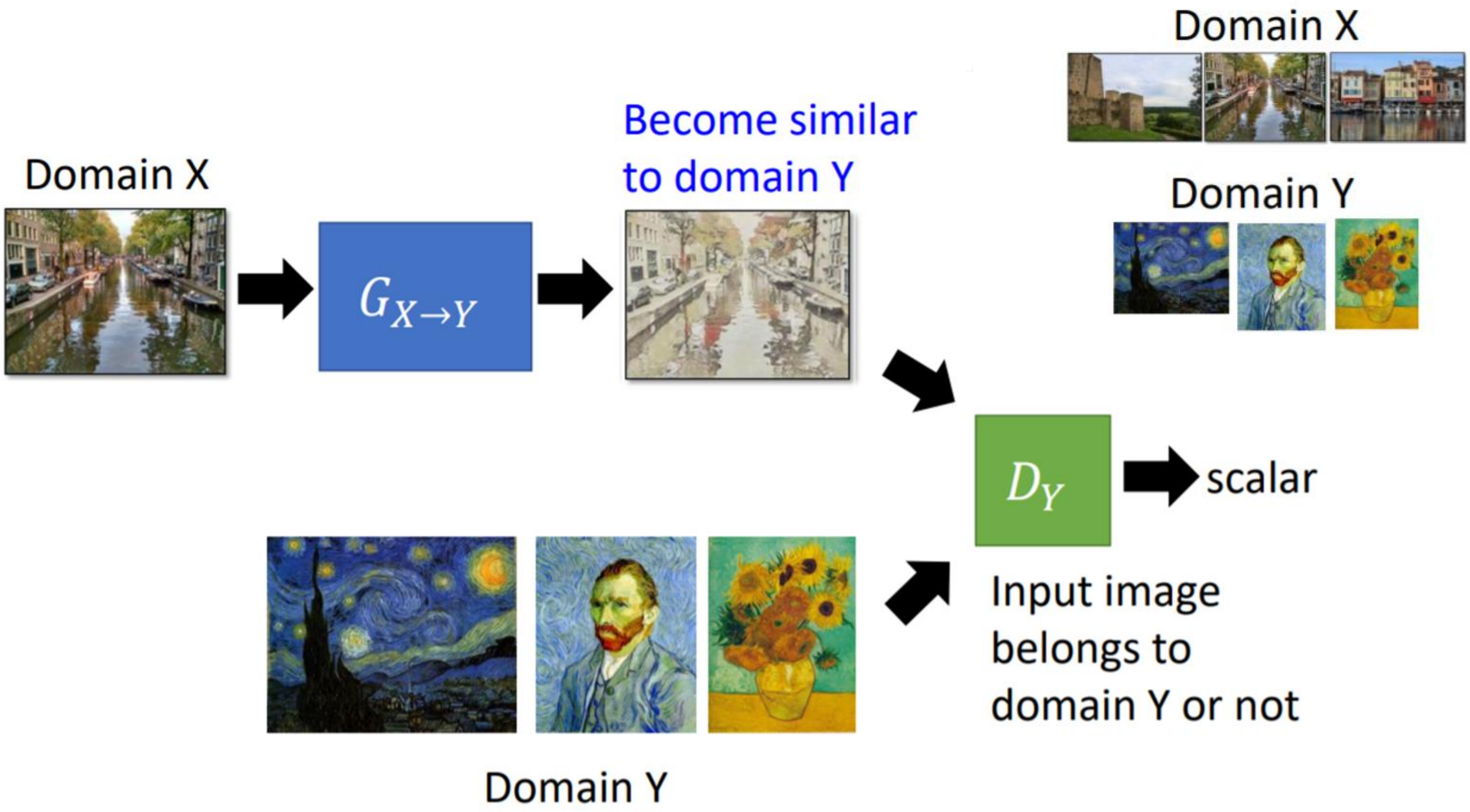
- Approach 2: Projection to Common Space



Larger change, only keep the semantics

Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

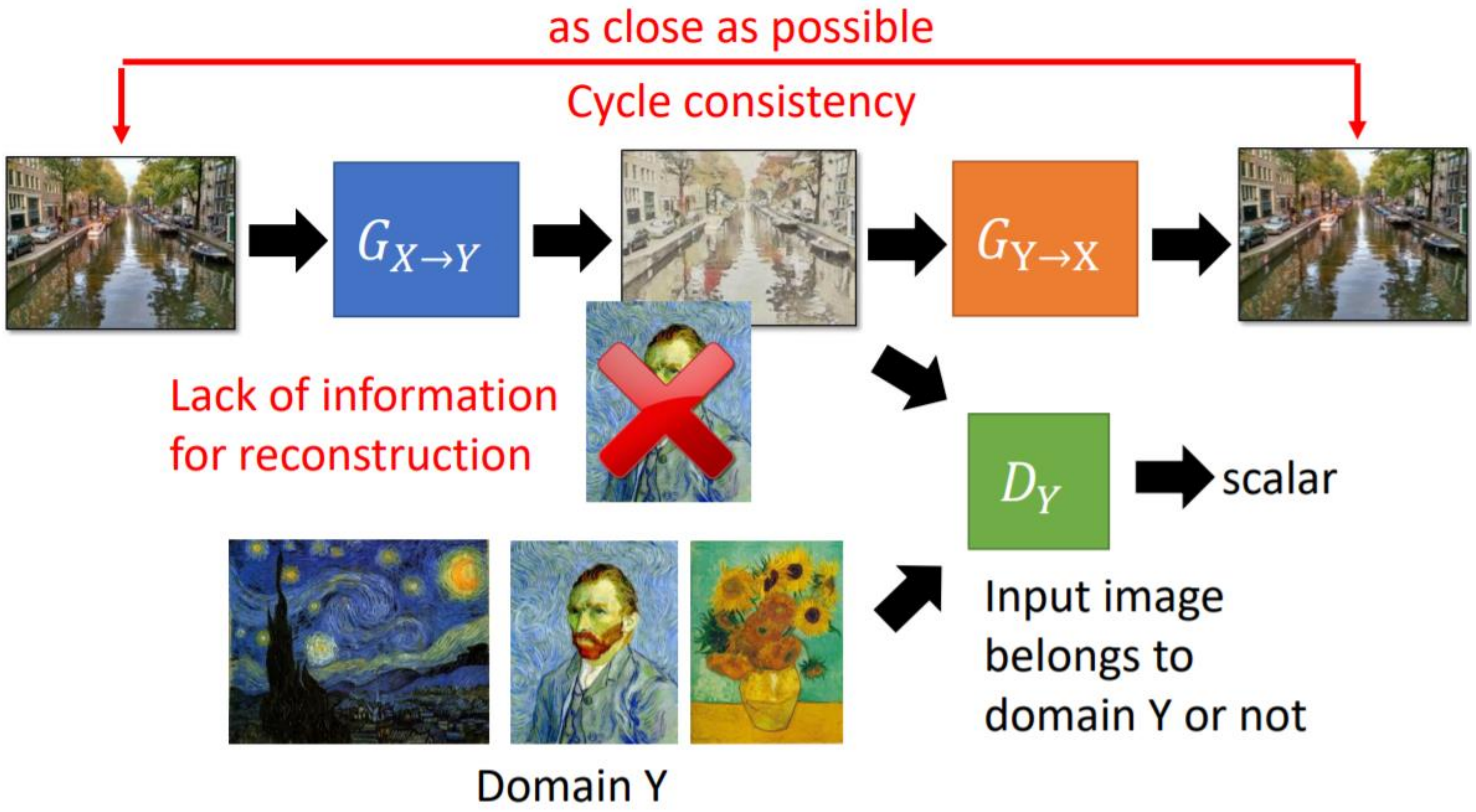
Direct transform



Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

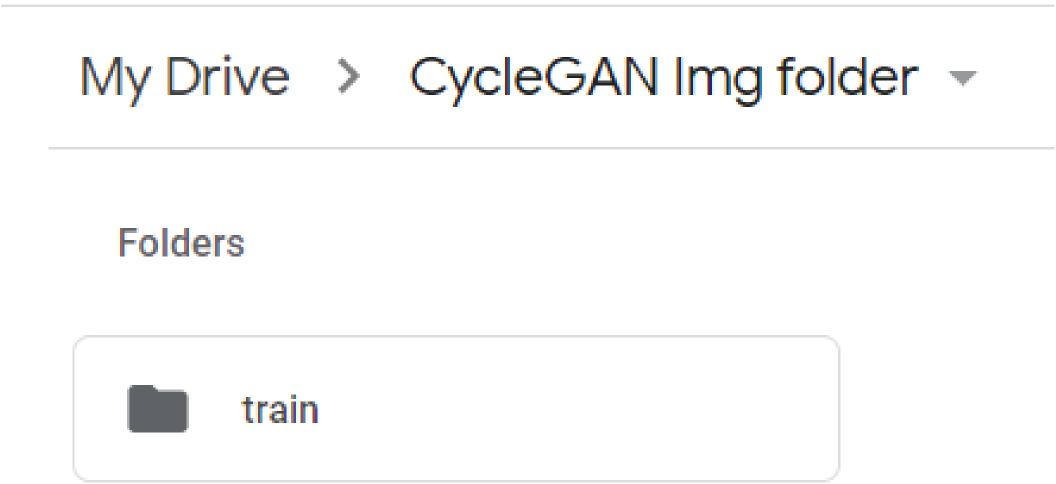
Cycle GAN

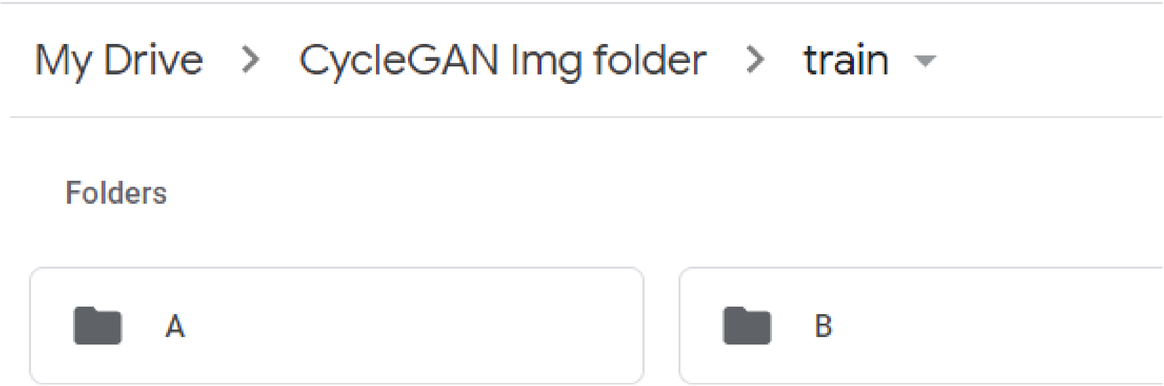
[Jun-Yan Zhu, et al., ICCV, 2017]



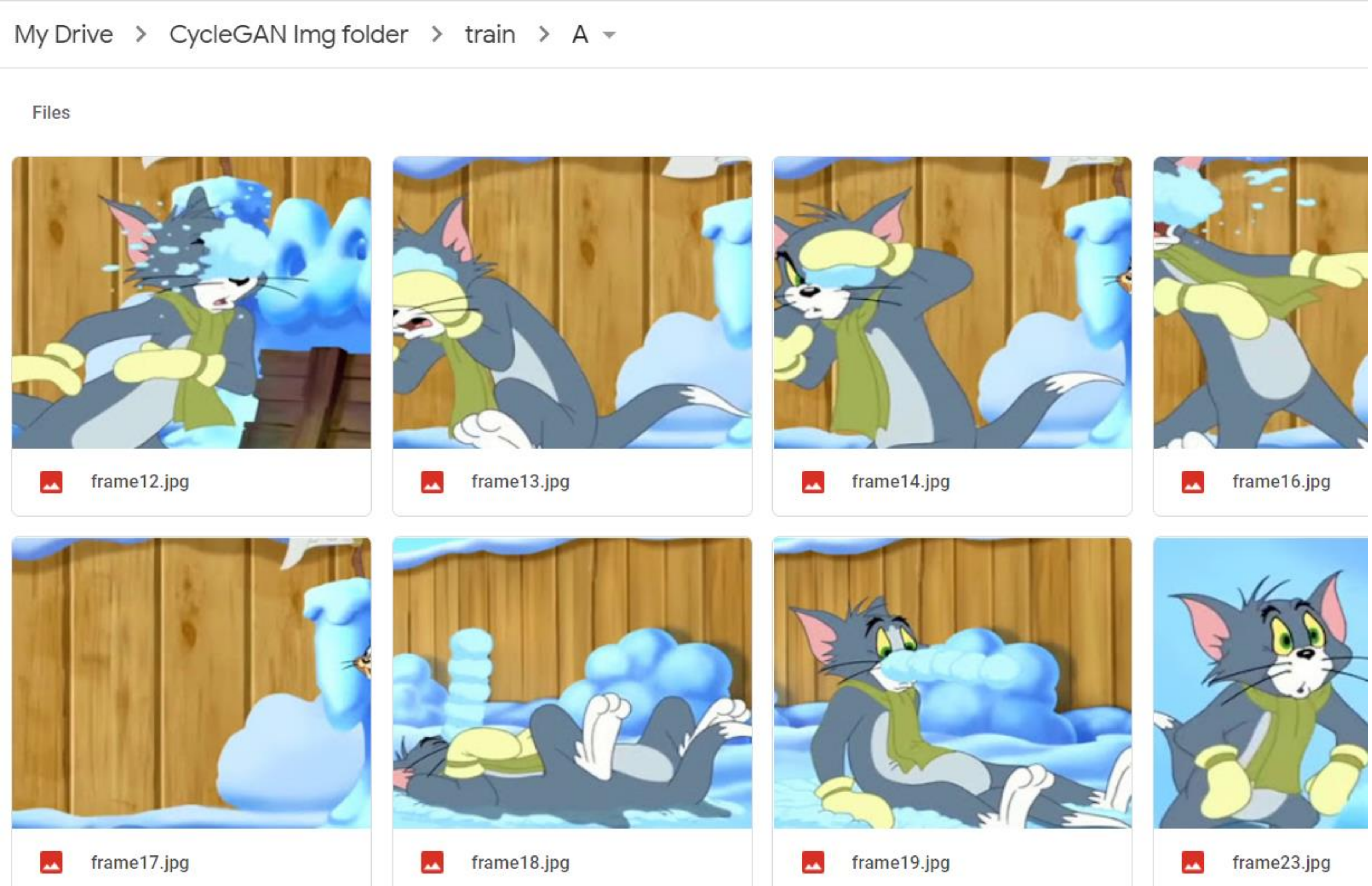
Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

Prepare training images for CycleGAN

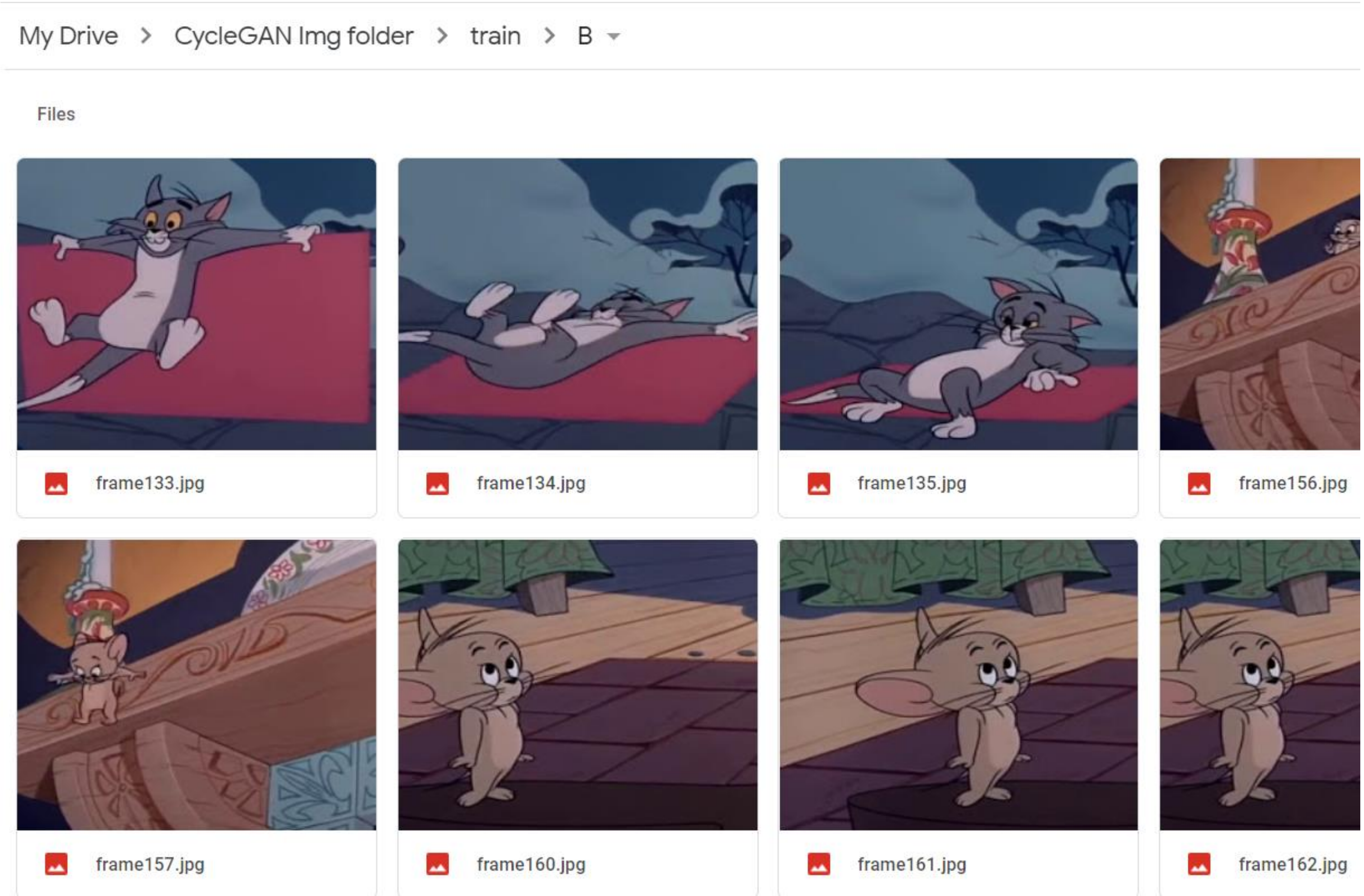




Prepare training images for CycleGAN



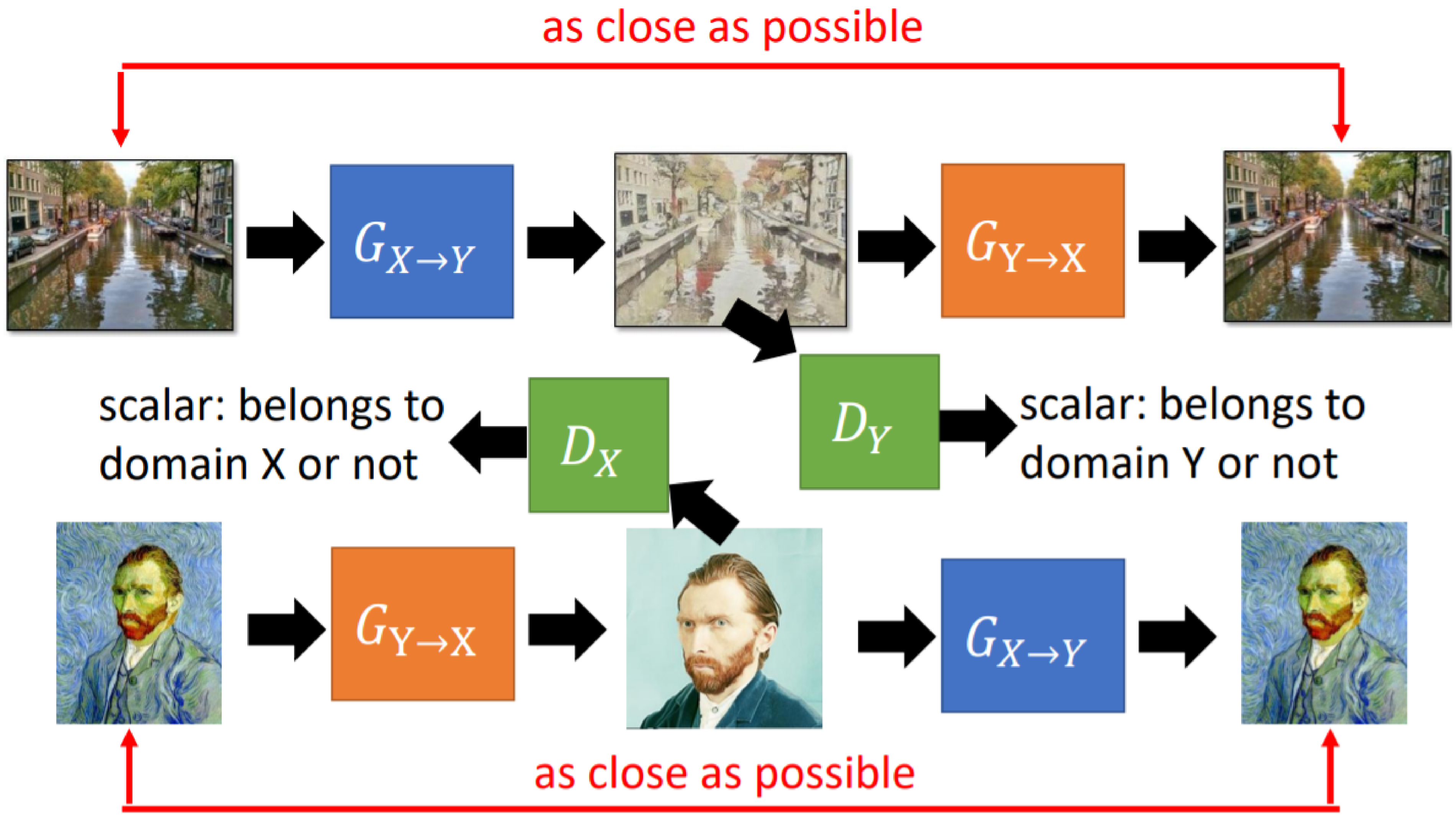
Prepare training images for CycleGAN



Prepare training images for CycleGAN

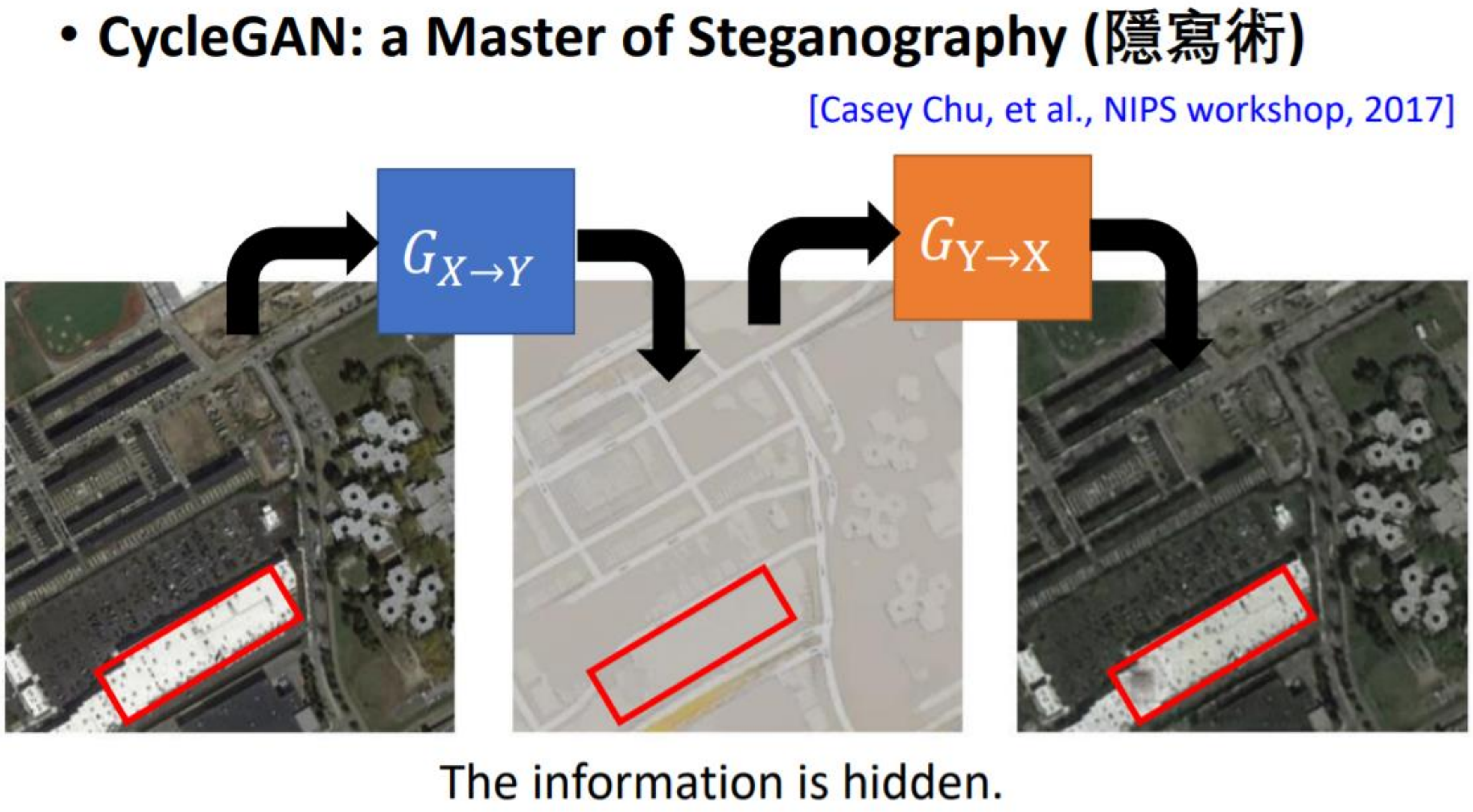
```
[21] lr = 0.0001
     epochs = 5 # change to larger number, 50000, for real training
     decay_epochs = 2 #change to 100
```

Two-way cycle GAN



Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

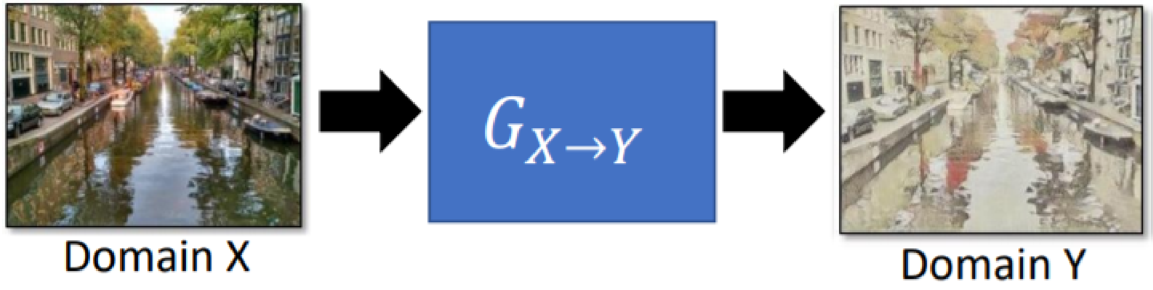
Cycle GAN will hide information in the input and display it again in the output



Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

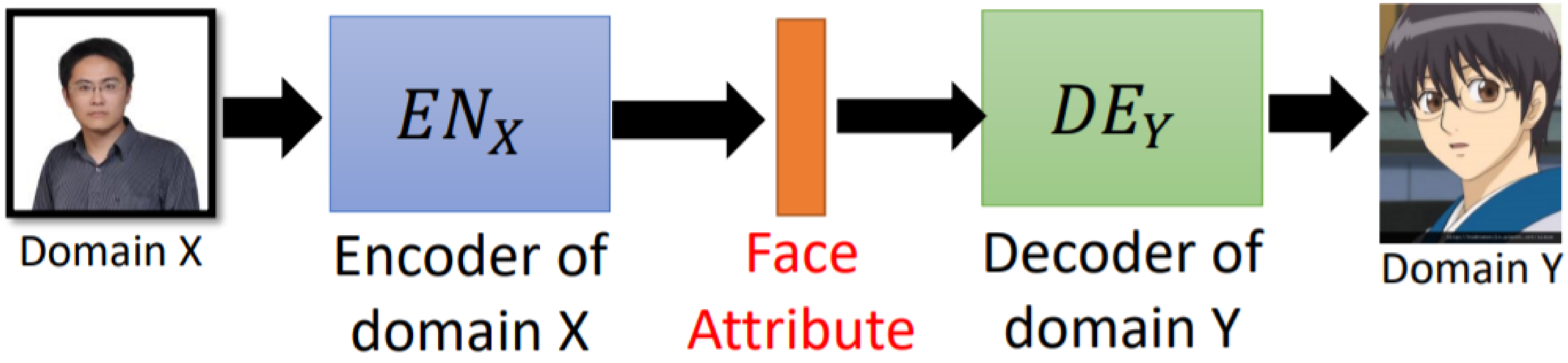
Projection to common space

- Approach 1: Direct Transformation



For texture or color change

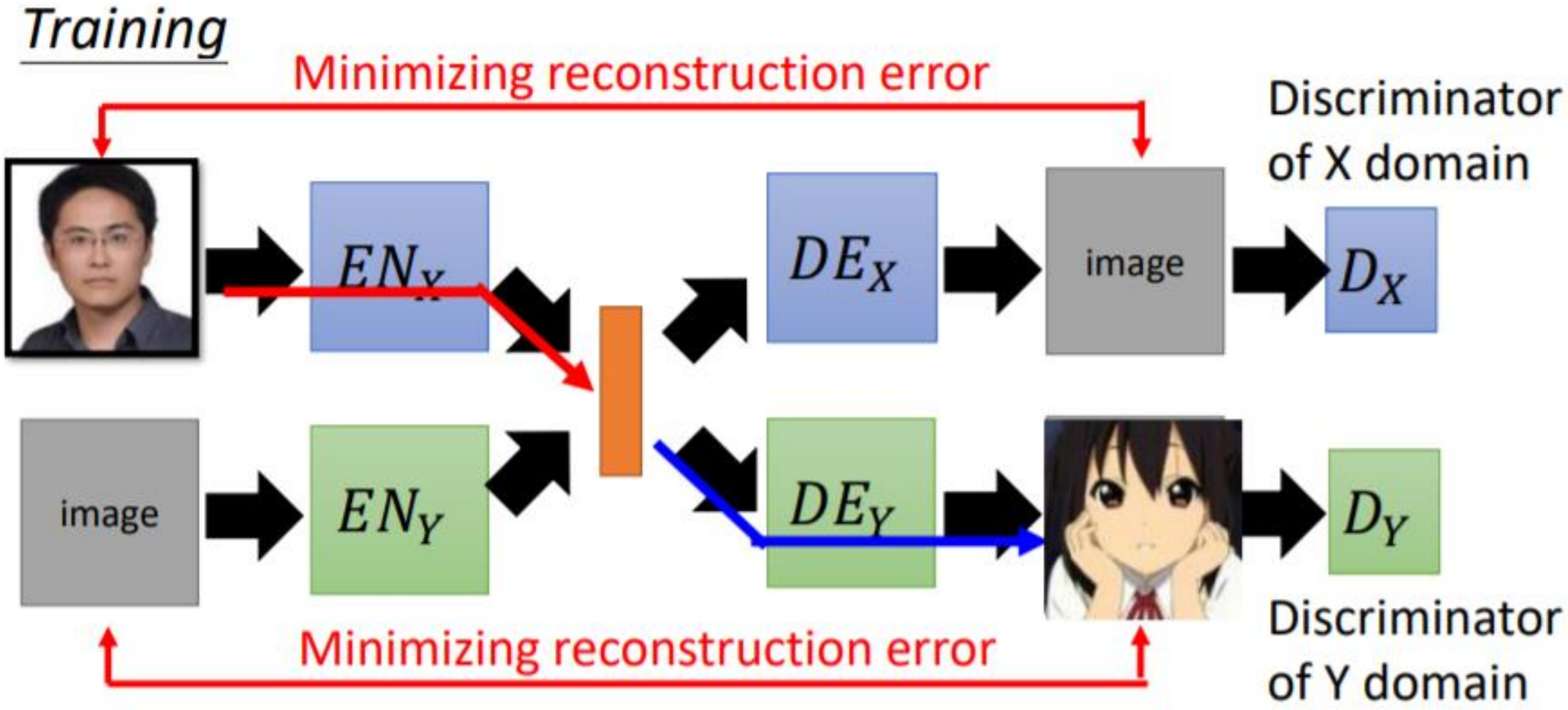
- Approach 2: Projection to Common Space



Larger change, only keep the semantics

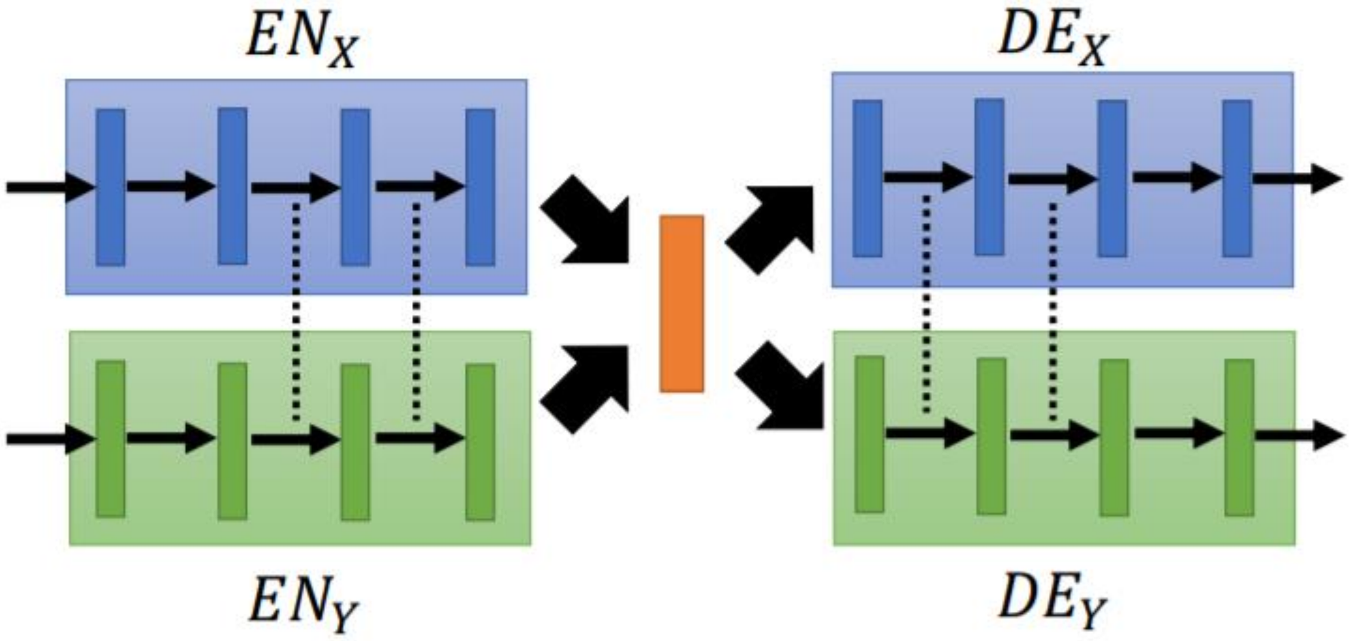
Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

VAE enhanced GAN



Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

Approach 1 – Sharing parameters to tie them together



Sharing the parameters of encoders and decoders

Couple GAN[Ming-Yu Liu, et al., NIPS, 2016]

UNIT[Ming-Yu Liu, et al., NIPS, 2017]

Reference: 李弘毅 GAN Lecture 3 (2018): https://youtu.be/-3LgL3NXLtI

Approach 2 – domain discriminator