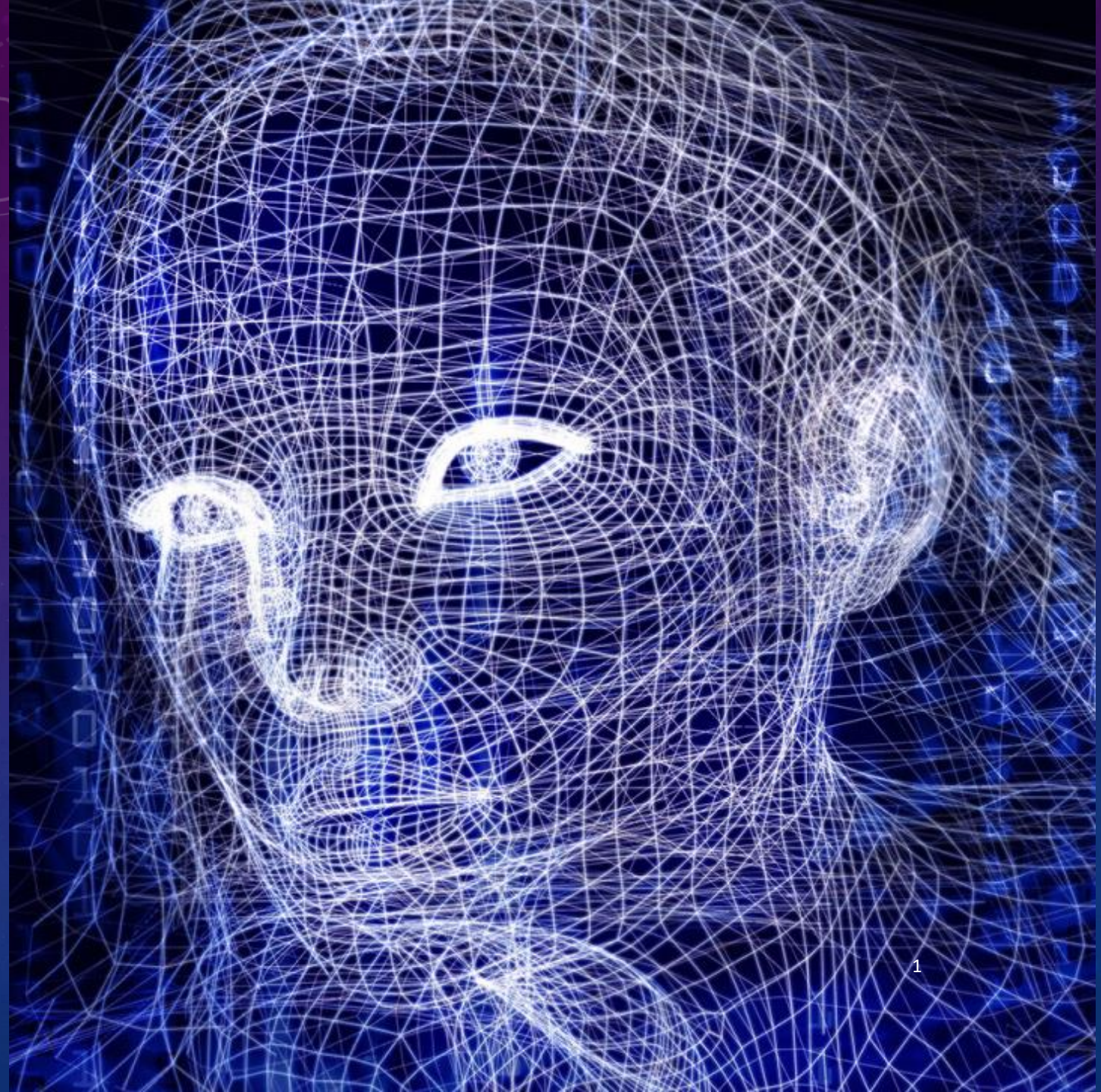
LECTURE SERIES FOR COMPUTER VISION

Generative adversarial network

徐繼聖

Gee-Sern Jison Hsu

National Taiwan University of Science and Technology



What we have learned so far

- Convolutional Neural Networks
  - Convolution and Deconvolution
  - Forward Pass
- Autoencoder
  - Encoder
  - Decoder
  - Variational Encoder-Decoder

Contents - Generative Adversarial Networks

- Concept of generating Data -> Generator
- Two Player Game
  - Example
  - Schemata
  - Transfer to CV
- Challenges of GANs
- Applications of GANs
  - DCGAN (Deep Convolutional Generative Adversarial Network)
  - CycleGAN Introduction, Example and Exercise

We already learned to generate data with VAEs

Pros of VAEs:

- Principle approach to generative models
- Allows inference of the conditional Probability

Cons of VAEs:

- Maximizes lower bound of likelihood: okay, but not as good evauluation as PixelCNN
- Samples are blurrier and lower quality

→ Solution? We can increase the sample quality with GANs!

So far..

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_\theta(x) = \prod_{i=1}^{n} p_\theta(x_i | x_1, ..., x_{i-1})$$

VAEs define intractable density function with latent **z**:

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

But: Cannot optimize directly, derive and optimize lower bound on likelihood instead

What if we give up on explicitly modeling density, and just want ability to sample?

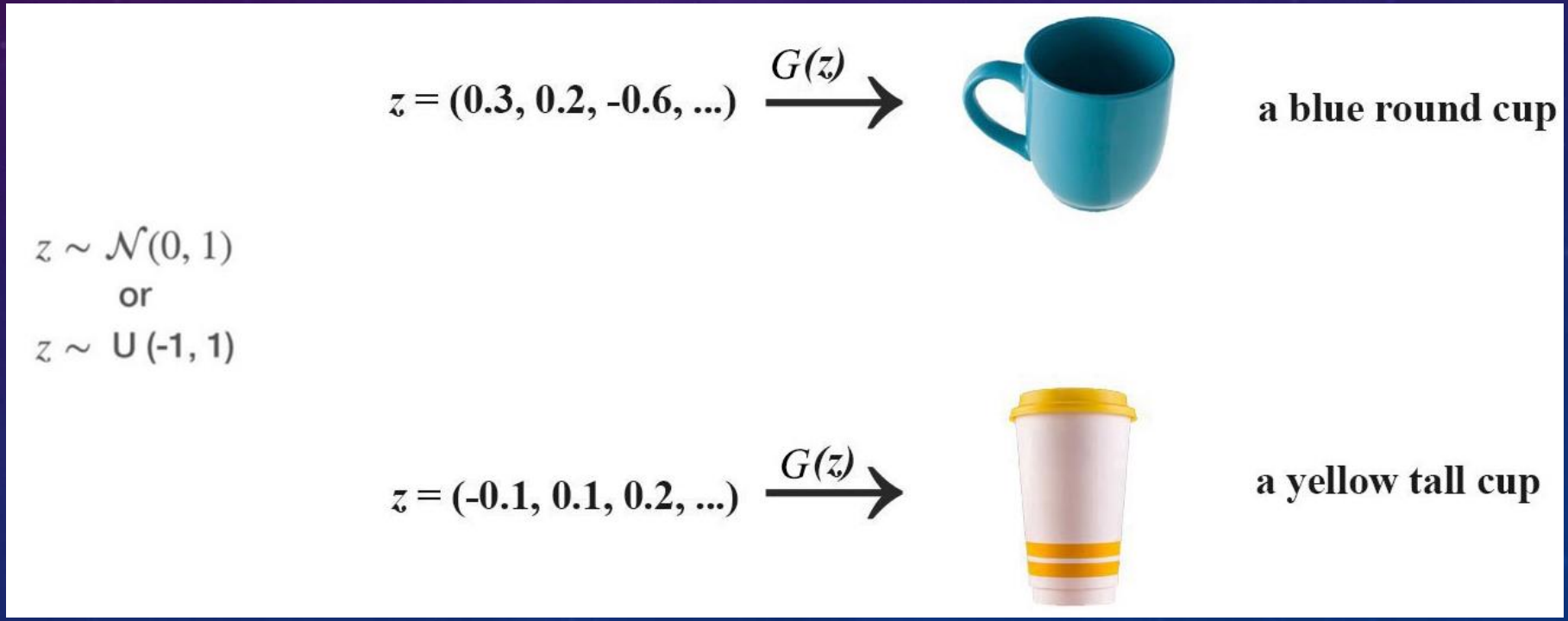
GANs: don't work with any explicit density function!

Instead, take game-theoretic approach: learn to generate from training distribution through 2-player game

What is a Generator?

GAN composes of two deep networks, the **generator,** and the **discriminator**.

1. Sample some noise z using a normal or uniform distribution
2. With z as an input, we use a generator G to create an image x $\left(x = G(z)\right)$.

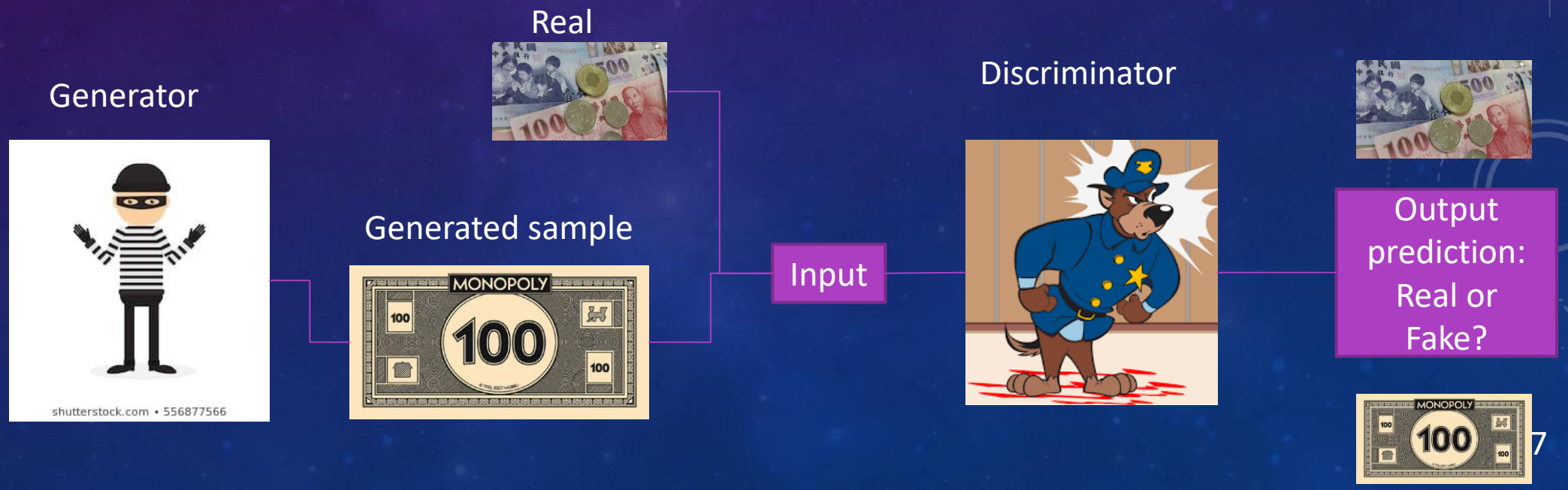


Training GANs: Two-Player game Example

- **Generator Network:** try to fool the discriminator by generating real-looking images
- **Discriminator Network:** try to distinguish between real and fake images

Example: Criminal Faking money (Generator),

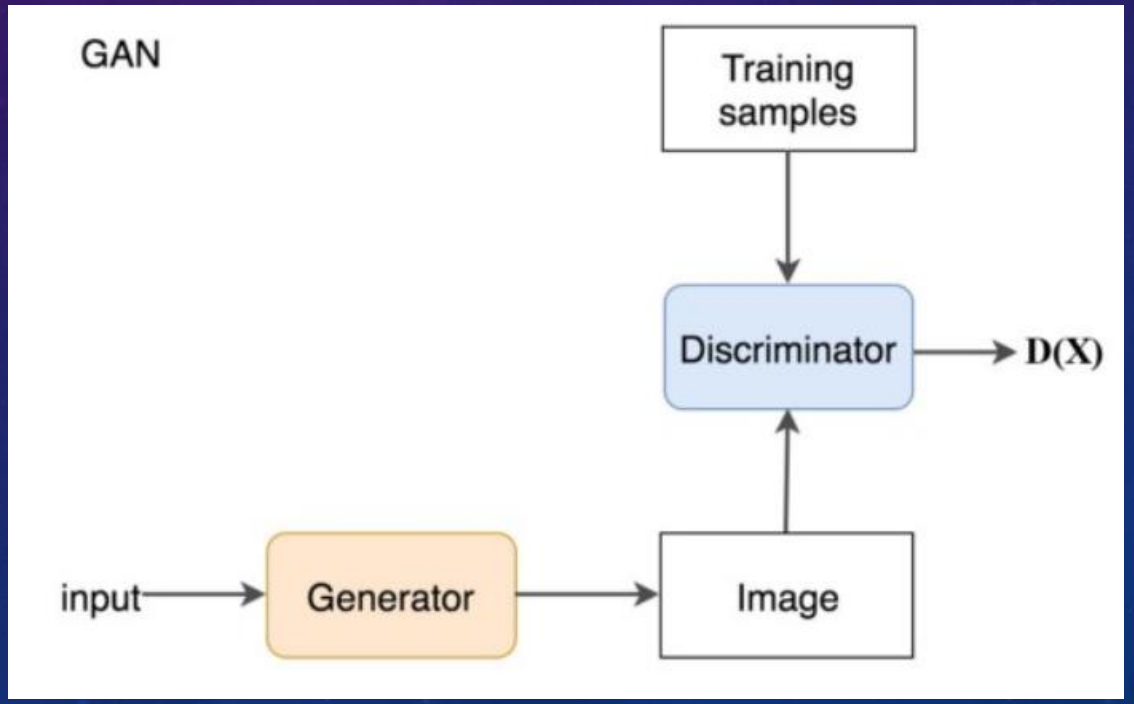
the police has to distinguish (Discriminator)



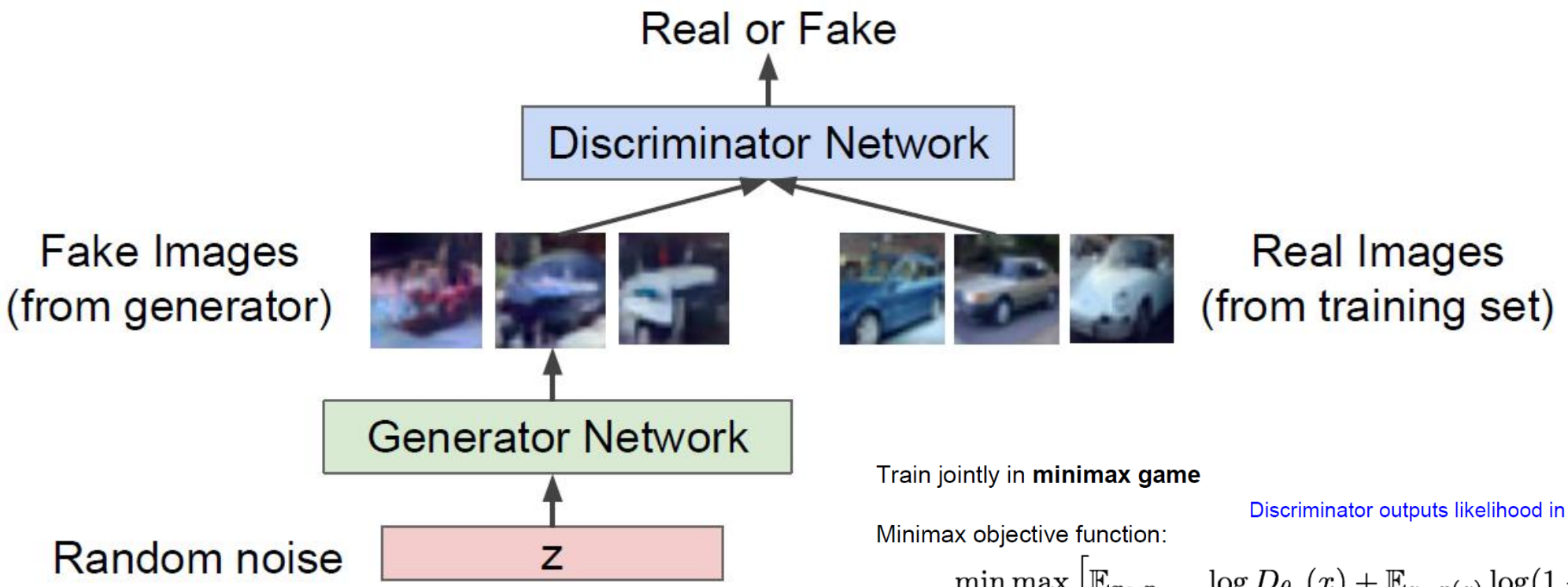
Training GANs: Two-Player game Schematic

By training with real images and generated images,

- GAN builds a discriminator to distinguish whether the discriminator input is real or generated.
- Then the discriminator will output a value D(x) to estimate the chance that the input is real.



Transfer to Computer vision



Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

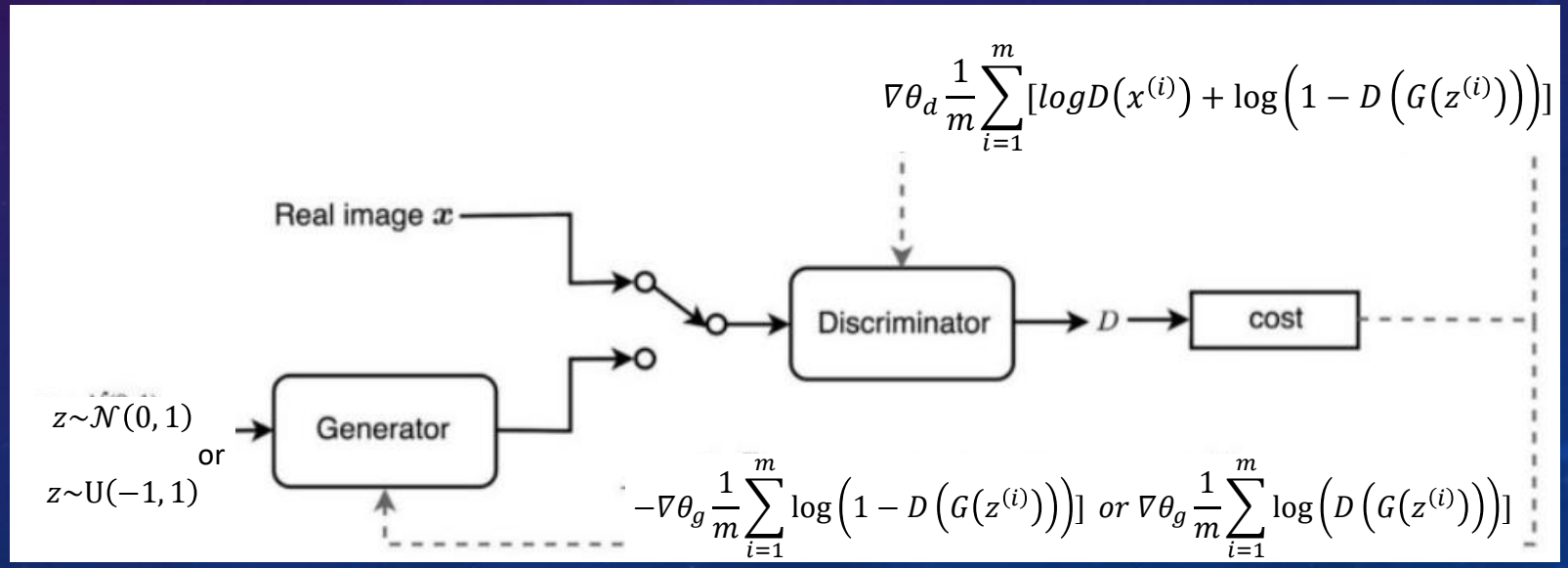
Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Objective function and gradients

- GAN is defined as a minimax game with the following objective function.

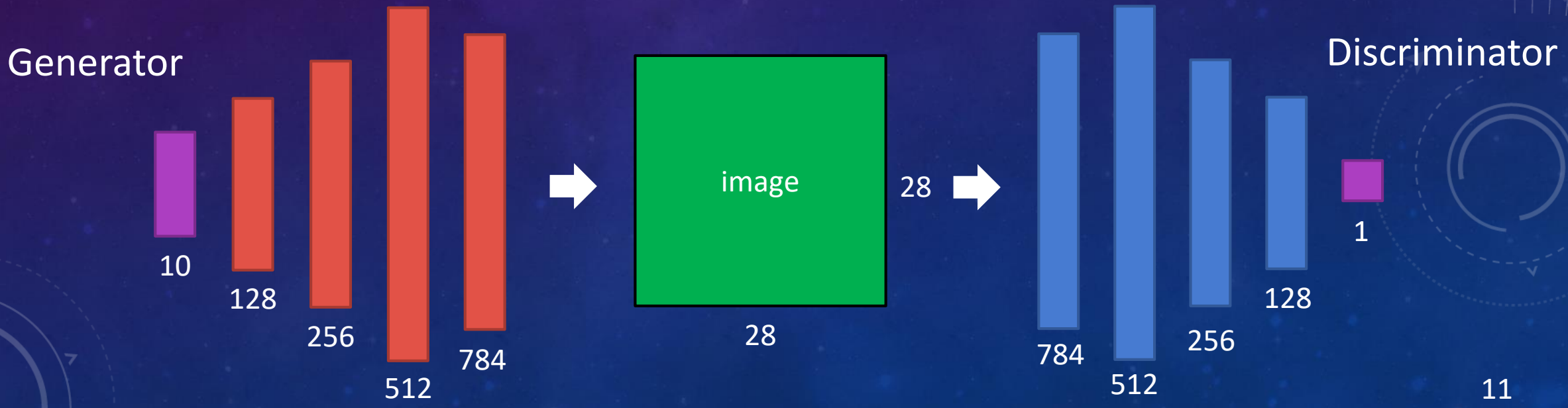
$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P_{data}(x)}[logD(x)] + \mathbb{E}_{z\sim P_z(z)}[\log(1 - D(G(z))]$$

- The diagram below summarizes how we train the discriminator and the generator using the corresponding gradient.



Example 4-1: Generative Adversarial Network

- Please download the "exercise4.1_ GAN.ipynb" on Moodle.
- Upload the "exercise4.1_ GAN.ipynb" to the Google Colab.
- Follow the sample code to understand the data flow of the generative adversarial network.
  - The generator aims to generate a novel image from a noise input.
  - The discriminator aims to distinguish the generated image from the real one.



Example 4-1: Generative Adversarial Network

Define the hyper-parameter and load the training data

```
train_epoch = 50
batch_size = 64
noise_size = 100
lr = 2e-4

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

dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=img_transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=False)
```

← Define the hyperparameter

← Download the Mnist dataset to the folder ‘./data’

Example 4-1: Generative Adversarial Network

Define the generator and discriminator

```
class generator(nn.Module):
    def __init__(self, input_size=100, n_class = 28*28):
        super(generator, self).    init    ()
        self.fc1 = nn.Linear(input_size, 256)
        self.fc2 = nn.Linear(self.fc1.out_features, 512)
        self.fc3 = nn.Linear(self.fc2.out_features, 1024)
        self.fc4 = nn.Linear(self.fc3.out_features, n_class)
        self.tanh = nn.Tanh()
```

Define the generator

```
    def forward(self, input):
        x = F.leaky_relu(self.fc1(input), 0.2)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.tanh(self.fc4(x))
        return x
```

```
class discriminator(nn.Module):
    def __init__(self, input_size=28*28, n_class=1):
        super(discriminator, self).__init__()
        self.fc1 = nn.Linear(input_size, 1024)
        self.fc2 = nn.Linear(self.fc1.out_features, 512)
        self.fc3 = nn.Linear(self.fc2.out_features, 256)
        self.fc4 = nn.Linear(self.fc3.out_features, n_class)
        self.sigmoid = nn.Sigmoid()
    def forward(self, input):
        x = F.leaky_relu(self.fc1(input), 0.2)
        x = F.dropout(x, 0.3)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = F.dropout(x, 0.3)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = F.dropout(x, 0.3)
        x = self.sigmoid(self.fc4(x))
        return x
```

Define the discriminator

Example 4-1: Generative Adversarial Network

Define the loss function

```
G = generator(input_size=noise_size, n_class=28*28)
D = discriminator(input_size=28*28, n_class=1)
```

Build a model

```
if torch.cuda.is_available():
    G.cuda()
    D.cuda()

print(G)
print(D)
```

```
BCE_loss = nn.BCELoss()
```

← Use “binary cross-entropy” as loss function

```
D_optimizer = torch.optim.Adam(D.parameters(), lr=lr, betas=(0.5, 0.999))
G_optimizer = torch.optim.Adam(G.parameters(), lr=lr, betas=(0.5, 0.999))
```

Example 4-1: Generative Adversarial Network

Start to training the "discriminator" D

```
# train discriminator D
    D.zero_grad()
    x_ = x_.view(-1, 28 * 28)
    mini_batch = x_.size()[0]
    y_real_ = torch.ones(mini_batch)
    y_fake_ = torch.zeros(mini_batch)
    x_, y_real_, y_fake_ = Variable(x_),
          Variable(y_real_), Variable(y_fake_)
    if torch.cuda.is_available():
       x_ = x_.cuda()
       y_real_ = y_real_.cuda()
       y_fake_ = y_fake_.cuda()
    D_result = D(x_)
    D_real_loss = BCE_loss(D_result, y_real_)
    D_real_score = D_result
```

Flatten the real images

Discriminate the real images are real or fake

Generate the noises

```
z_ = torch.randn((mini_batch, noise_size))
z_ = Variable(z_)
if torch.cuda.is_available():
   z_ = z_.cuda()
G_result = G(z_)

D_result = D(G_result)
D_fake_loss = BCE_loss(D_result, y_fake_)
D_fake_score = D_result

D_train_loss = D_real_loss + D_fake_loss

D_train_loss.backward()
D_optimizer.step()
```

Generate the images by the noises

Example 4-1: Generative Adversarial Network

Start to training the “generator” G

```
# train generator G
    G.zero_grad()
                                Generate the noises
    z_ = torch.randn((mini_batch, noise_size))
    y_ = torch.ones(mini_batch)

    z_, y_ = Variable(z_), Variable(y_)
    if torch.cuda.is_available():
        z_ = z_.cuda()
        y_ = y_.cuda()          Generate the images by the noises
    G_result = G(z_)
    D_result = D(G_result)      Discriminate the fake images are real or fake
    G_train_loss = BCE_loss(D_result, y_)
    G_train_loss.backward()
    G_optimizer.step()
```

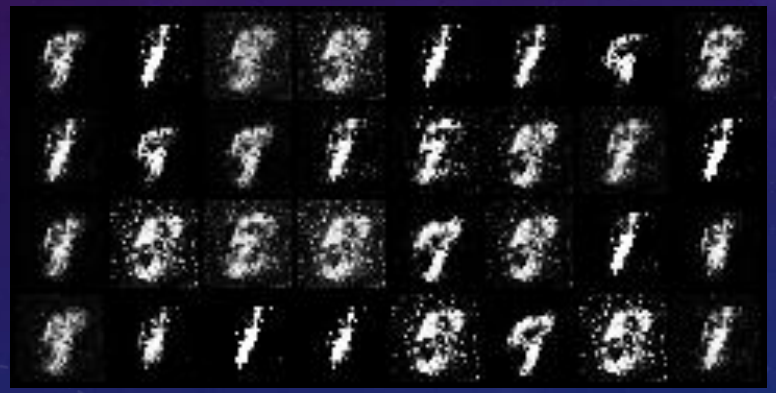
Example 4-1: Generative Adversarial Network

```
if epoch % 1 == 0:
    pic = to_img(G_result.cpu().data)
    save_image(pic, './gan_img/output_{}.png'.format(epoch))
```

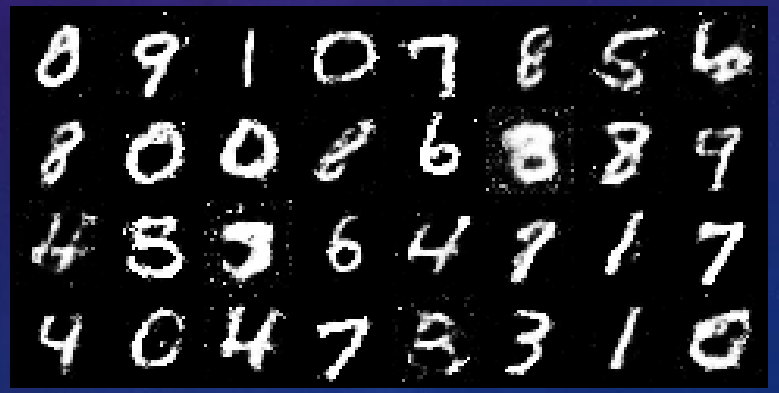
Save the output images

- Result:

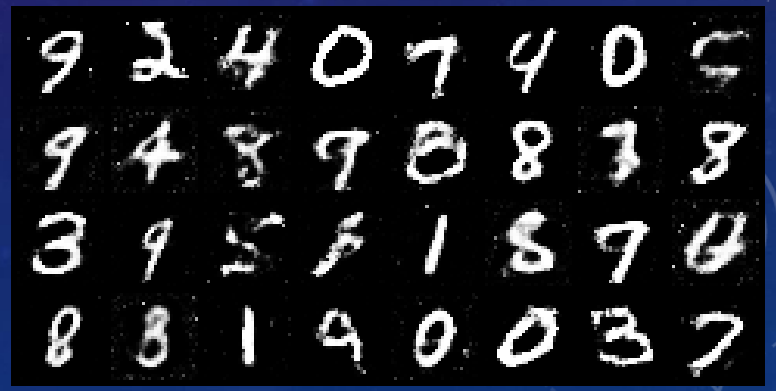
Epoch 1 :



Epoch 30 :



Epoch 50 :



~30 mins

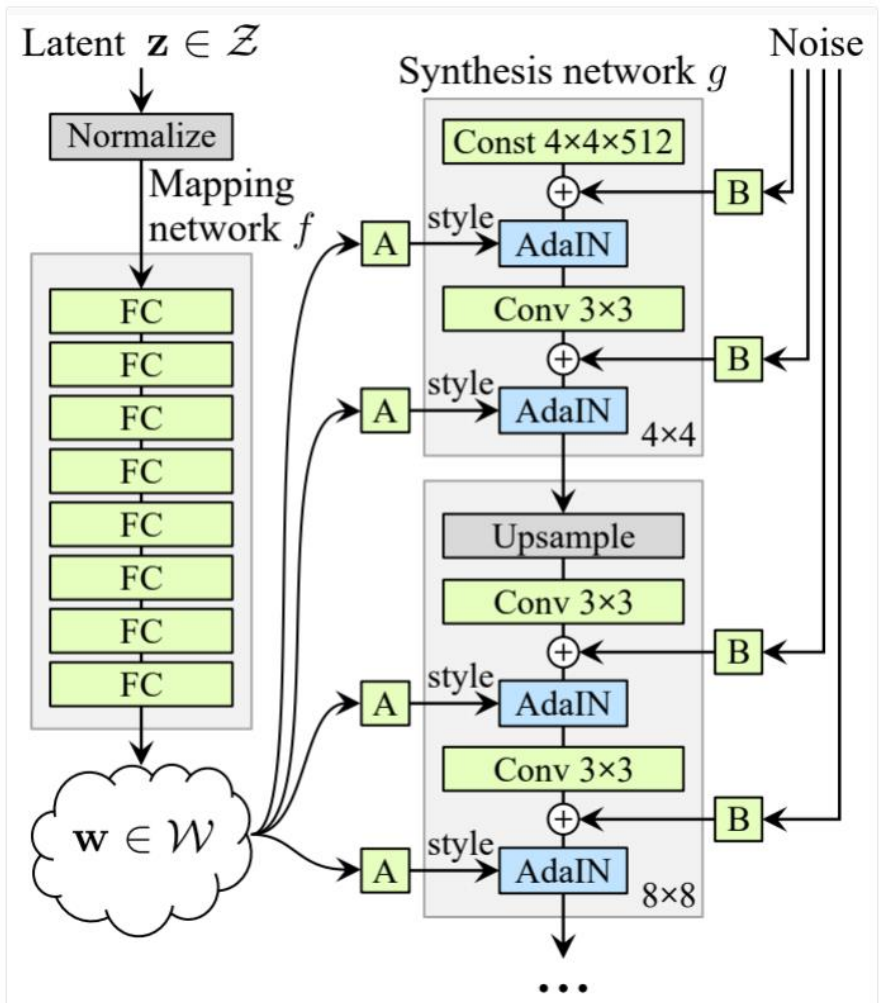
Lets be Discriminator for once: Are these faces real or fake?

Source A: gender, age, hair length, glasses, pose

Source B: everything else

Result of combining A and B

Answer: All Faces are generated by StyleGAN



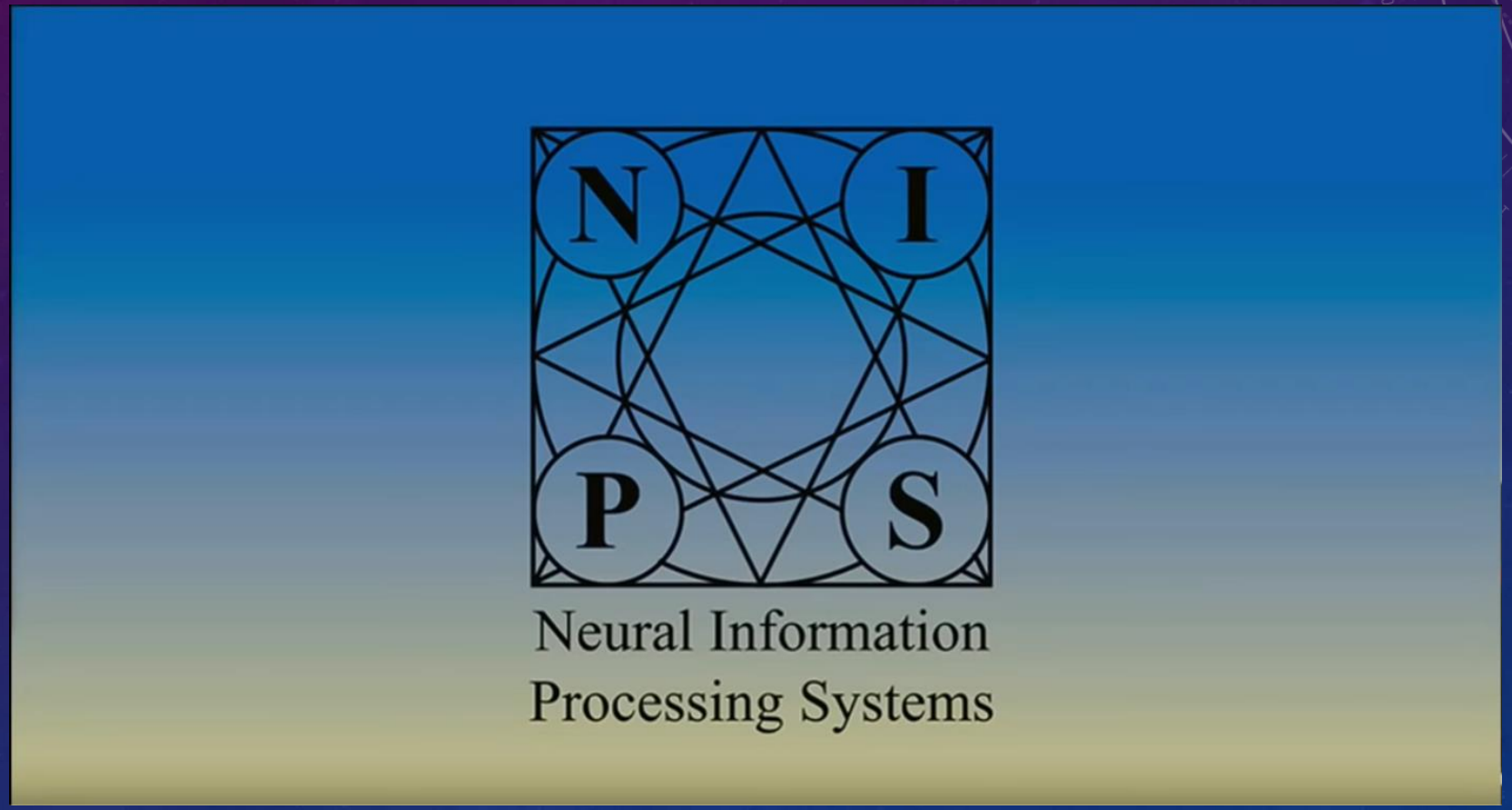
Paper: A Style-Based Generator Architecture for Generative Adversarial Networks

Release Date: March2019

Authors: Karras, Laine, Aila

Published in: CVPR 2019

Introduction to GANs, NIPS 2016
Ian Goodfellow, Inventor of GANs, OpenAI



https://www.youtube.com/watch?v=9JpdAg6uMXs [31:24]

However GANs come with problems

Many GAN models suffer the five major problems:

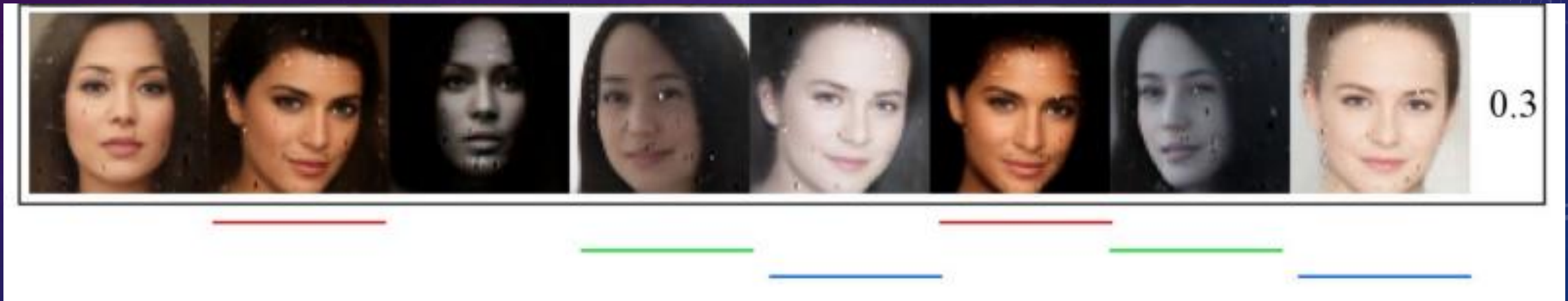
1. **Non-convergence**: the model parameters oscillate, destabilize and never converge,
2. **Mode collapse**: the generator collapses which produces limited varieties of samples,
3. **Diminished gradient**: the discriminator gets too successful that the generator gradient vanishes and learns nothing,
4. Unbalance between the generator and discriminator causing **overfitting**, and
5. **Highly sensitive** to the hyperparameter selections.

Further read by Jonathan Hui:

https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

Why mode collapse in GAN?

- Mode collapse is one of the hardest problems to solve in GAN.
- The images below with the same underlined color look similar and the mode starts collapsing.



Modified from source

Why mode collapse in GAN?

- The objective of the GAN generator is to create images that can fool the discriminator D the most.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

- Let's consider one extreme case where G is trained extensively without updates to D. The generated images will converge to find the optimal image x^* that fools D the most.

  In this extreme, x^* will be independent of the noise z.

$$x^* = argmax_x D(x)$$

- The mode collapses to a single point. The gradient associated with z approaches zero.

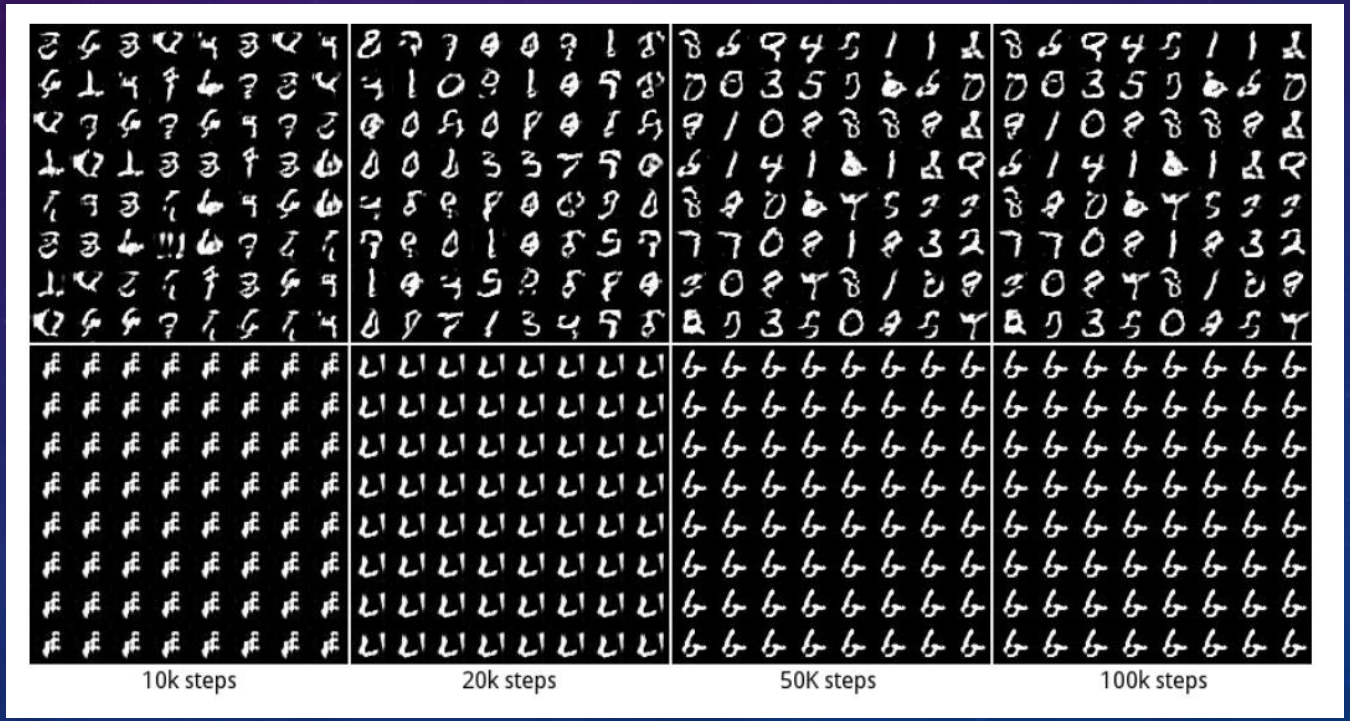
$$\frac{\partial J}{\partial z} \approx 0$$

Why mode collapse in GAN?

- When we restart the training in the discriminator, the most effective way to detect generated images is to detect this single mode.
  - Since the generator desensitizes the impact of z already, the gradient from the discriminator will likely push the single point around for the next most vulnerable mode.
  - The generator produces such an imbalance of modes in training that it deteriorates its capability to detect others.
  - The both networks are overfitted and the model will not converge.

Mode

- In MNIST, there are **10 major modes** from digit ‘0’ to digit ‘9’.
- The samples below are generated by two different GANs.
- The top row produces all 10 modes while the second row creates a single mode only “6”.
- This problem is called **mode collapse** when only a few modes of data are generated.



Source

Research is working on General Design Architectures for GANs

Generator is an upsampling network with fractionally-strided convolutions

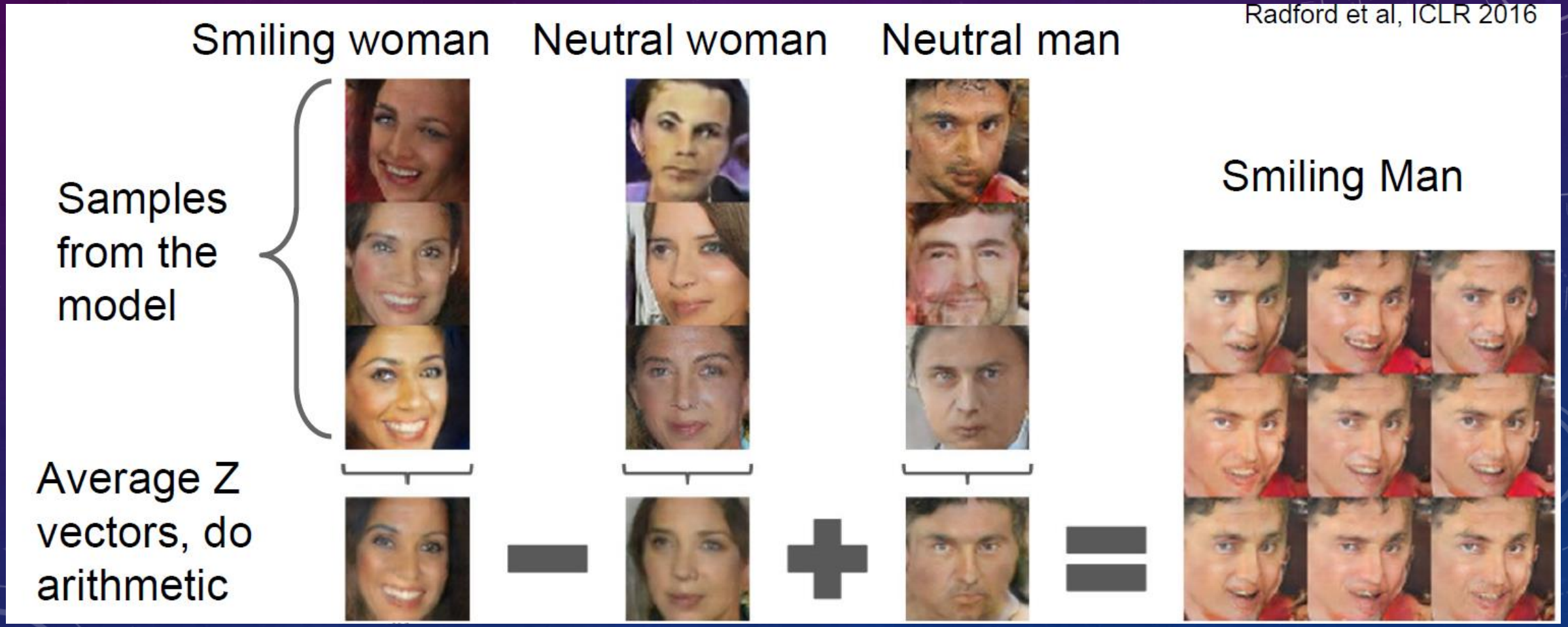
Discriminator is a convolutional network

How to make the GAN stable?

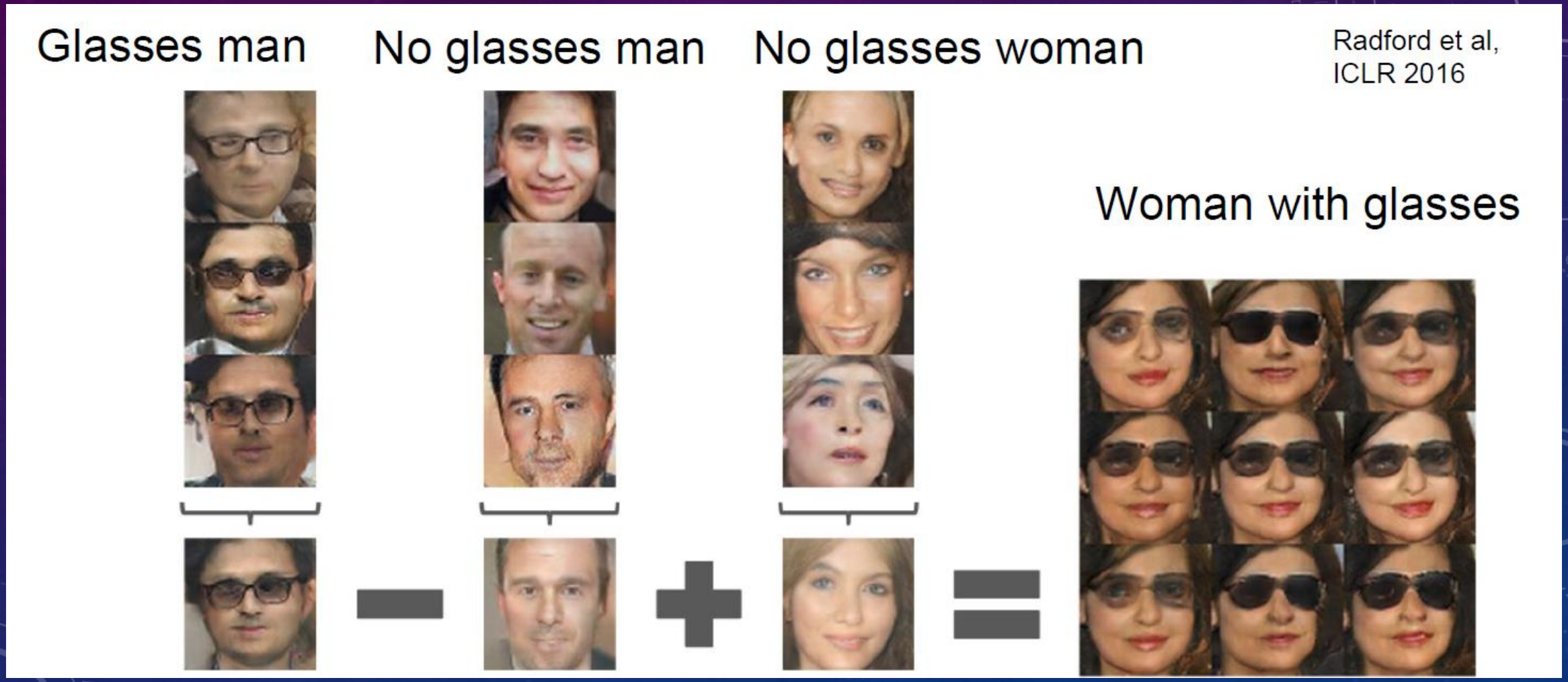
- Replace pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator)
- Batchnorm as a normalization method
- Remove FC layers
- Use ReLU activation in generator for all layers except for the output, which uses Tanh
- Use LeakyReLU activation in discriminator for all layers

By Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016

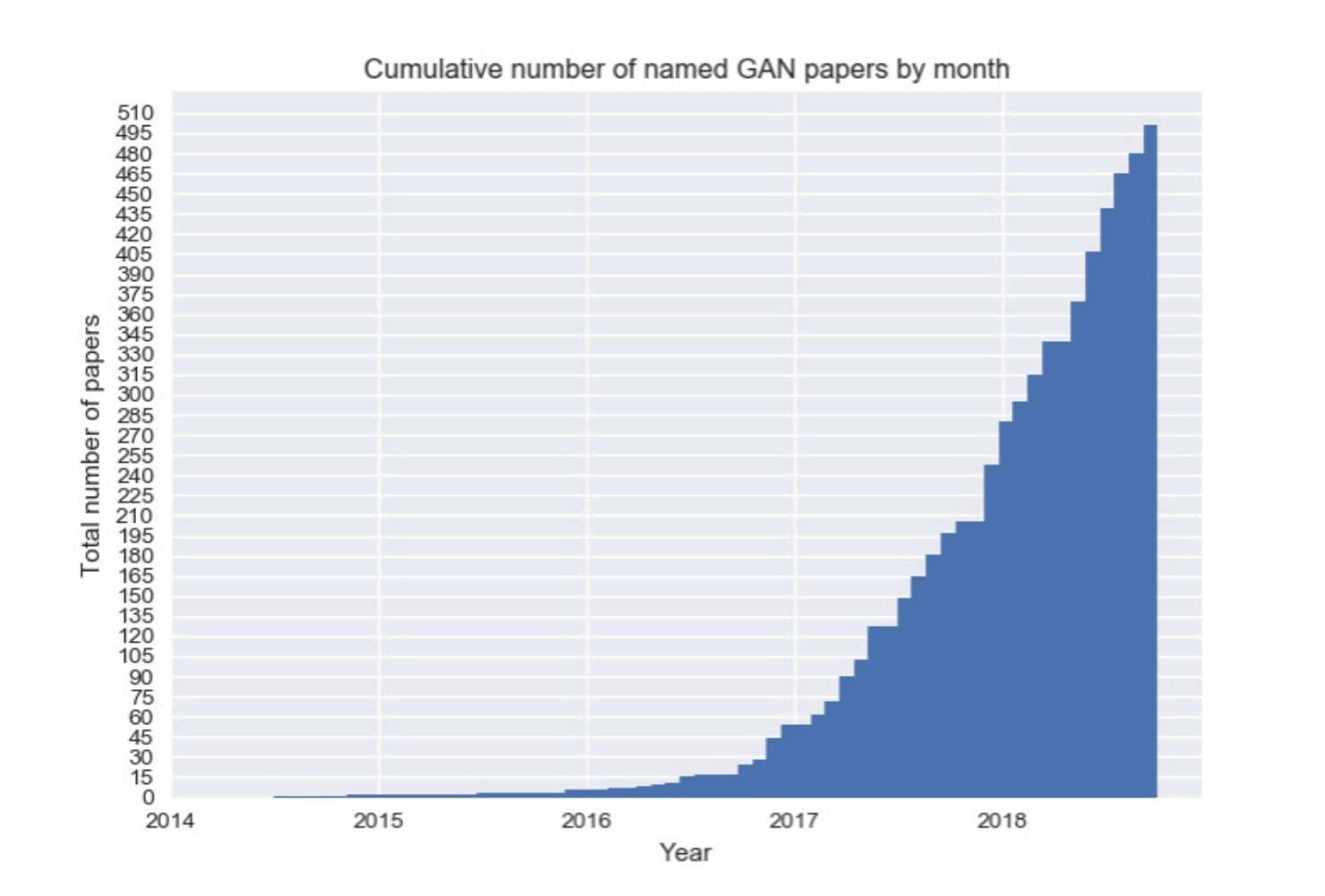
GAN: Vector Math to interpret



GAN: Vector Math to interpret



2017: Explosion of GANs (GAN Zoo)



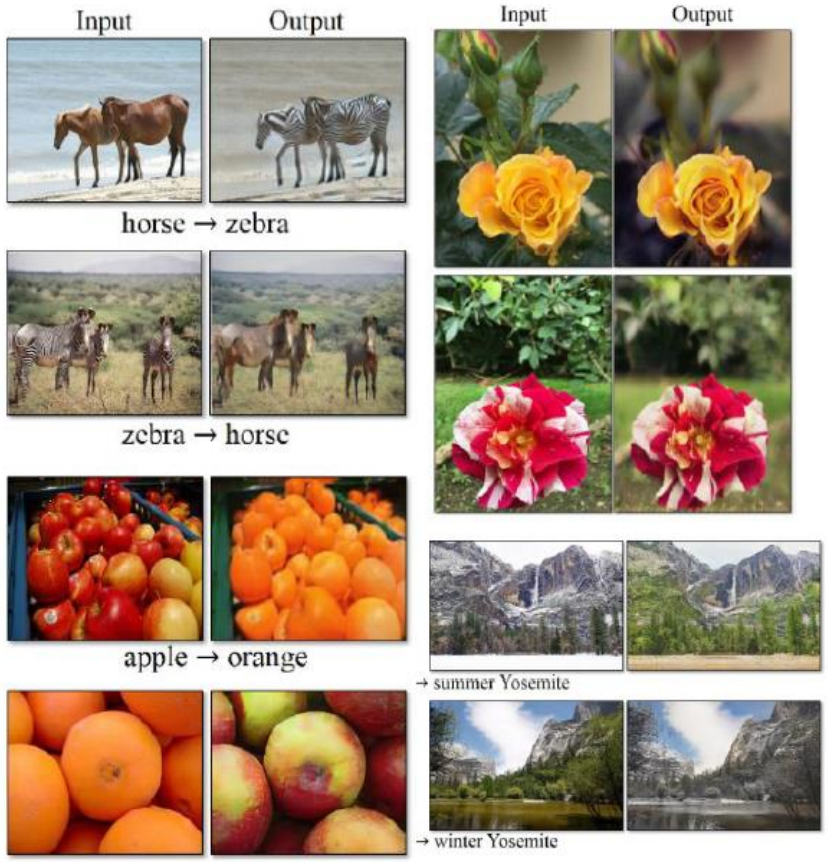
Github excerpt from letter A

https://github.com/hindupuravinash/the-gan-zoo

3D-ED-GAN - Shape Inpainting using 3D Generative Adversarial Network and Recurrent Convolutional Networks
•3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling (github)
•3D-IWGAN - Improved Adversarial Systems for 3D Object Generation and Reconstruction (github)
•3D-PhysNet - 3D-PhysNet: Learning the Intuitive Physics of Non-Rigid Object Deformations
•3D-RecGAN - 3D Object Reconstruction from a Single Depth View with Adversarial Learning (github)
•ABC-GAN - ABC-GAN: Adaptive Blur and Control for improved training stability of Generative Adversarial Networks (github)
•ABC-GAN - GANs for LIFE: Generative Adversarial Networks for Likelihood Free Inference
•AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
•acGAN - Face Aging With Conditional Generative Adversarial Networks
•ACGAN - Coverless Information Hiding Based on Generative adversarial networks
•acGAN - On-line Adaptative Curriculum Learning for GANs
•ACtuAL - ACtuAL: Actor-Critic Under Adversarial Learning
•AdaGAN - AdaGAN: Boosting Generative Models
•Adaptive GAN - Customizing an Adversarial Example Generator with Class-Conditional GANs
•AdvEntuRe - AdvEntuRe: Adversarial Training for Textual Entailment with Knowledge-Guided Examples
•AdvGAN - Generating adversarial examples with adversarial networks
•AE-GAN - AE-GAN: adversarial eliminating with GAN
•AE-OT - Latent Space Optimal Transport for Generative Models
•AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
•AF-DCGAN -

GANs have a wide range of application!

Source->Target domain transfer



CycleGAN. Zhu et al. 2017.

Text -> Image Synthesis

this small bird has a pink breast and crown, and black primaries and secondaries.

this magnificent fellow is almost all black with a red crest, and white cheek patch.

Reed et al. 2017.

Sketch to image
Pix2pix. Isola 2017

GANs have a wide range of application!

LSGAN, Zhu 2017.

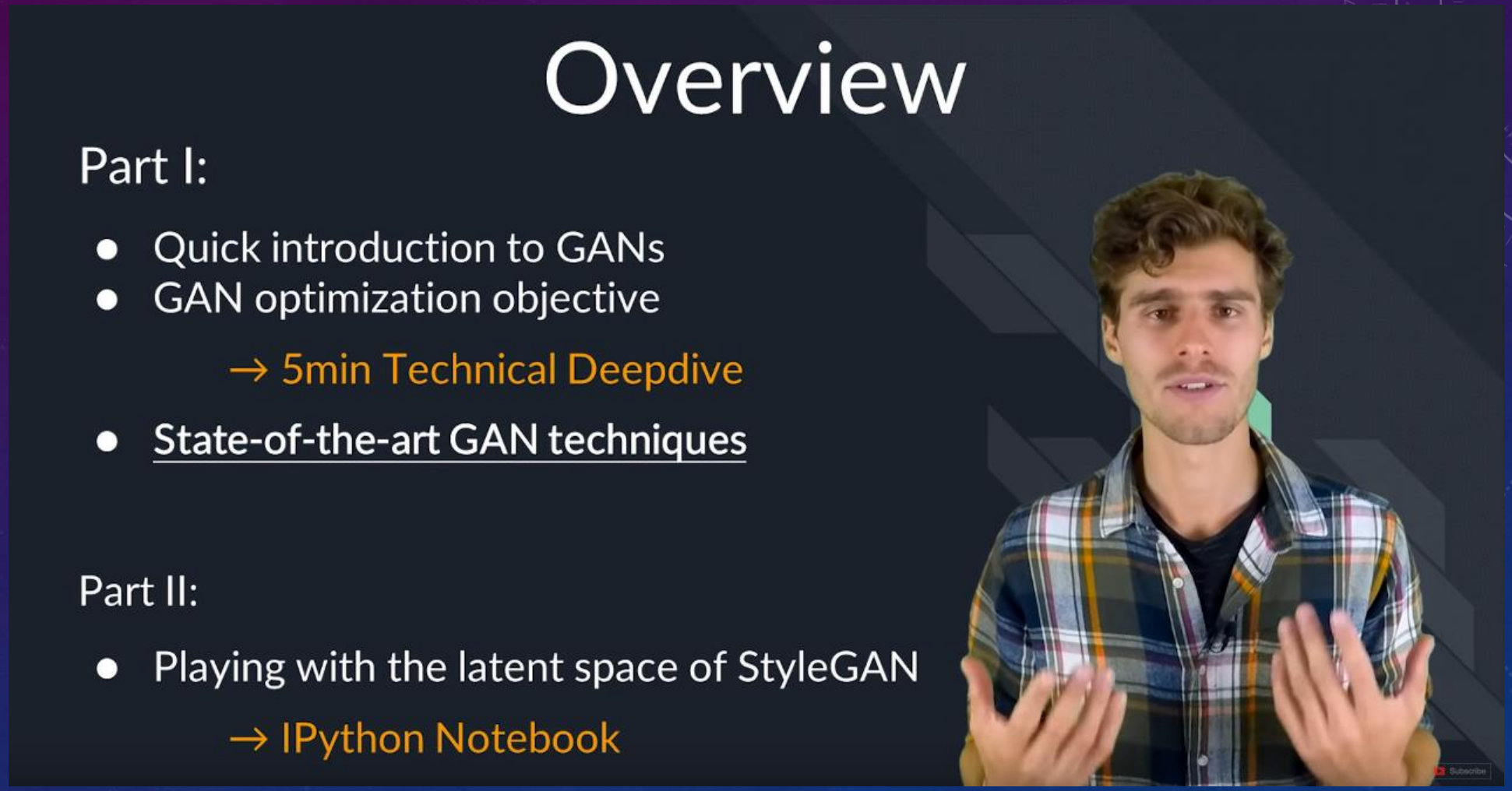
Wasserstein GAN, Arjovsky 2017. Improved Wasserstein GAN, Gulrajani 2017.

Progressive GAN, Karras 2018.

GANs are improving quickly: Progressive GAN

https://www.youtube.com/watch?v=XOxxPcy5Gr4&t=70s [6:10]

Face editing with Generative Adversarial Networks

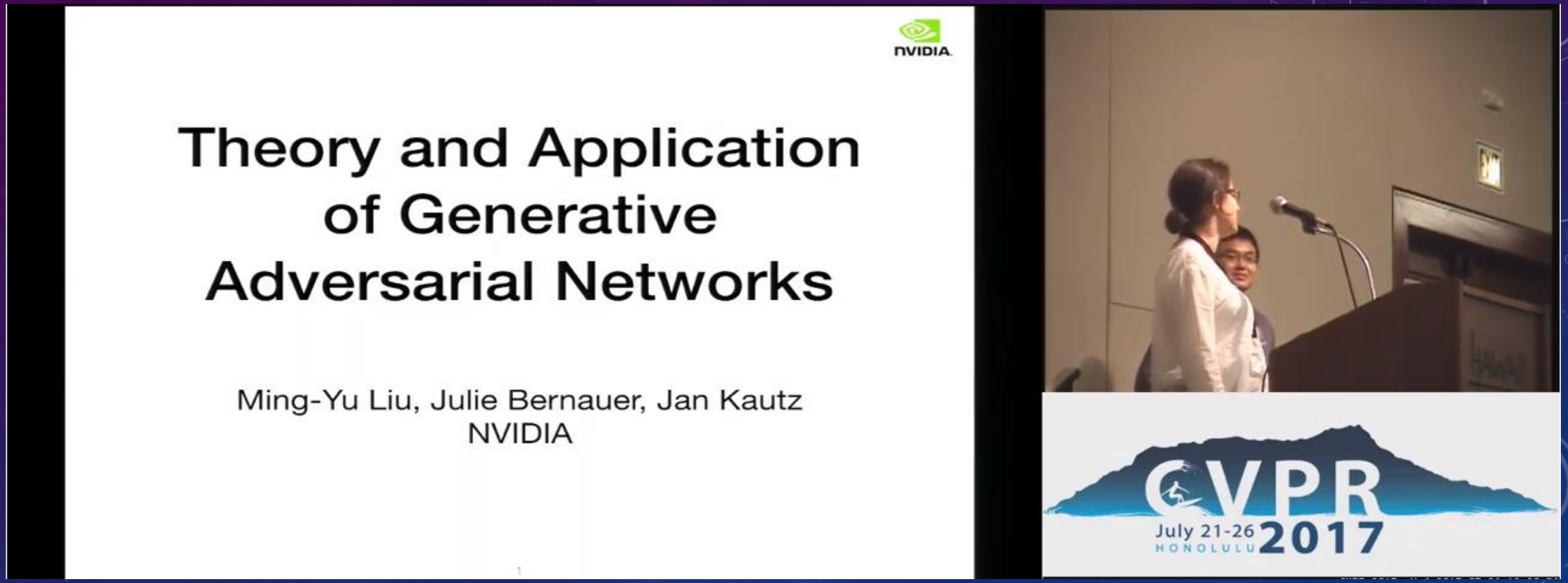


https://www.youtube.com/watch?v=dCKbRCUyop8 [25:26]

How to train a GAN, NIPS 2016 | Soumith Chintala, Facebook AI Research

https://www.youtube.com/watch?v=myGAju4L7O8 [31:58]

Tutorial : Theory and Application of Generative Adversarial Network



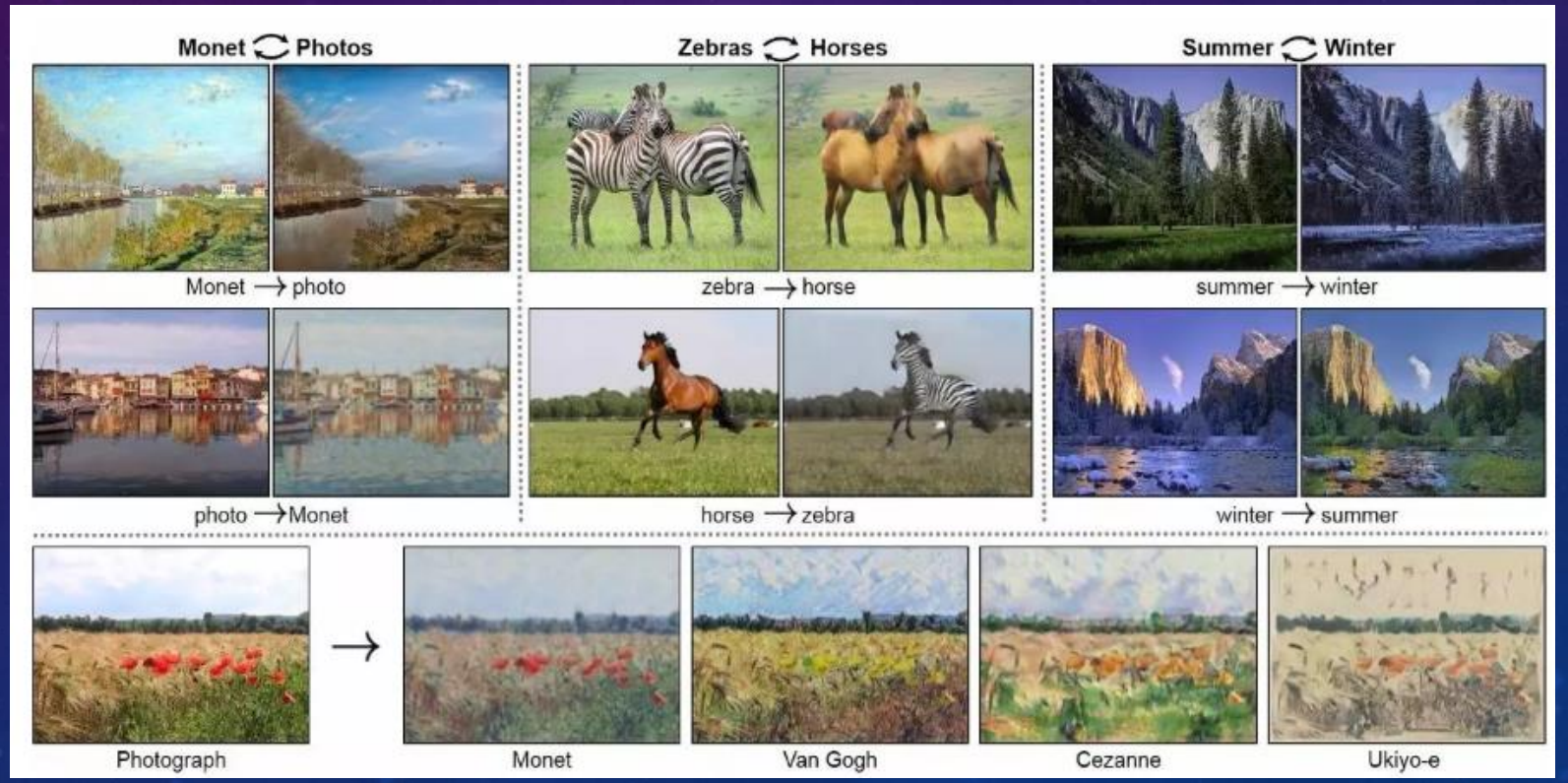
https://www.youtube.com/watch?v=KudkR-fFu_8 [2:53:05]

GAN — CycleGAN
An Amazing Tool to play and get to know the power of GANs

https://github.com/vanhuyz/CycleGAN-TensorFlow

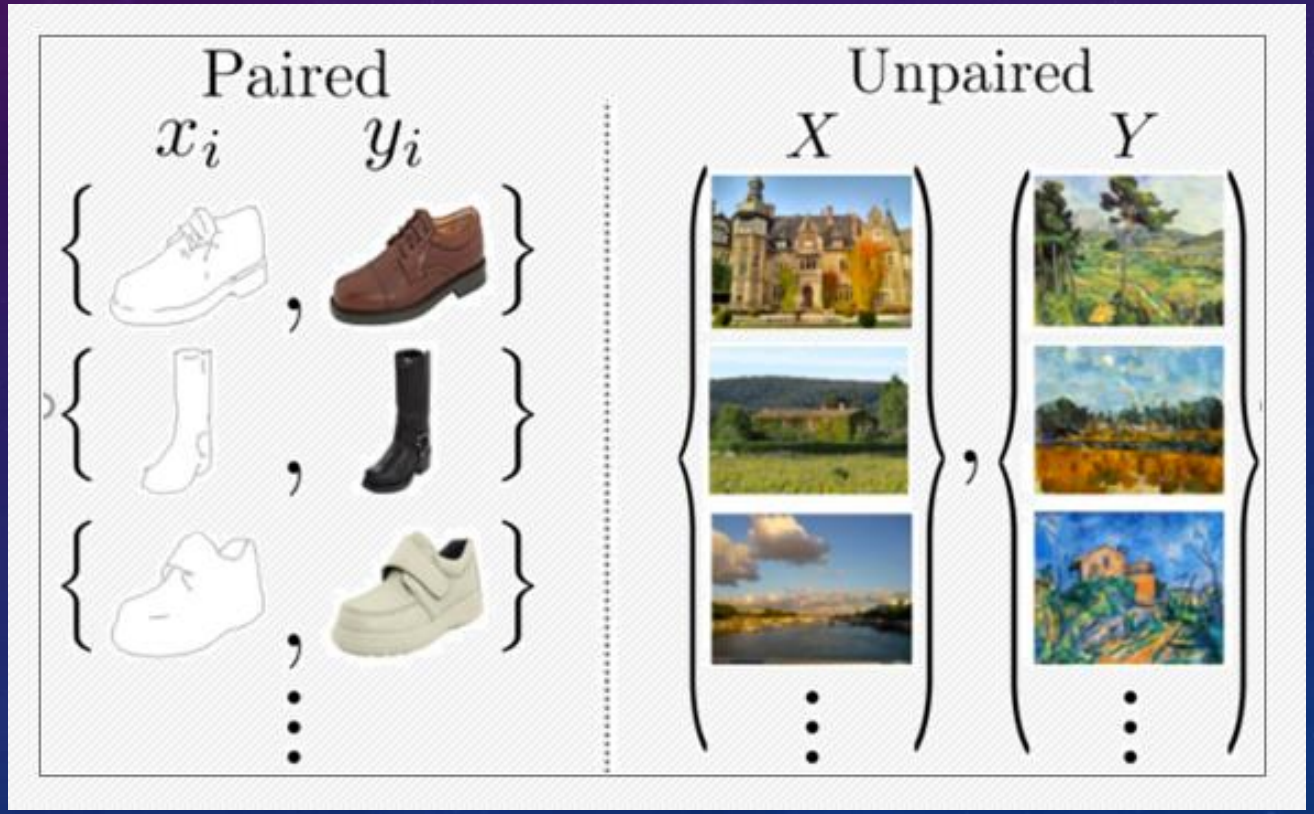
Background

- CycleGAN is an image style conversion technology proposed by Jun-Yan Zhu et al. The idea is to implement image style conversion without paired training data. cycleGAN can make different painter-style paintings into photos, turn summer into winter, turn horses into zebras, etc.

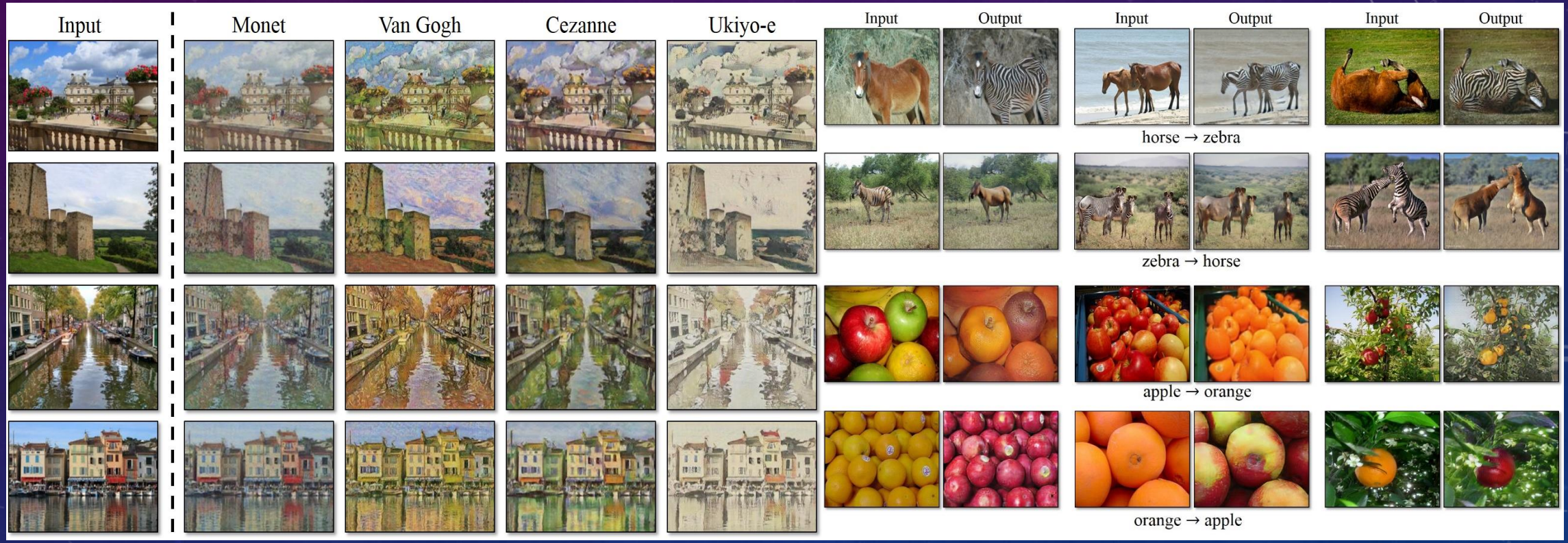


Background

- Pix2pix uses Conditional GAN to train a paired data set to achieve Cross Domain Image conversion; but it is not easy to generate a paired image. CycleGAN breaks this limitation. Achieve conversion between unpaired image sets.

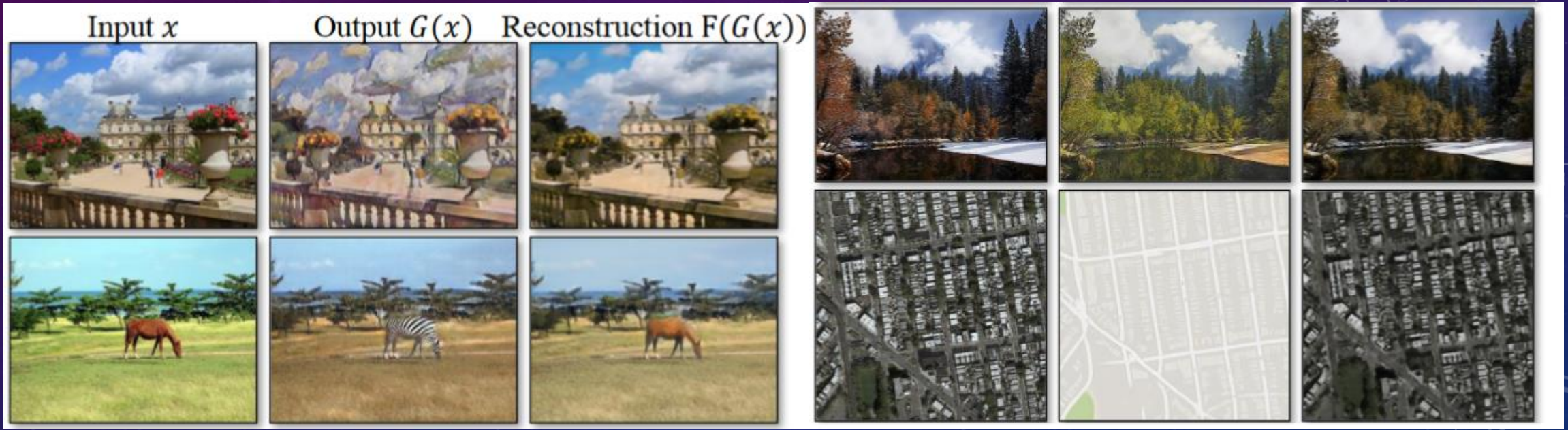


CycleGAN



Source : https://junyanz.github.io/CycleGAN/

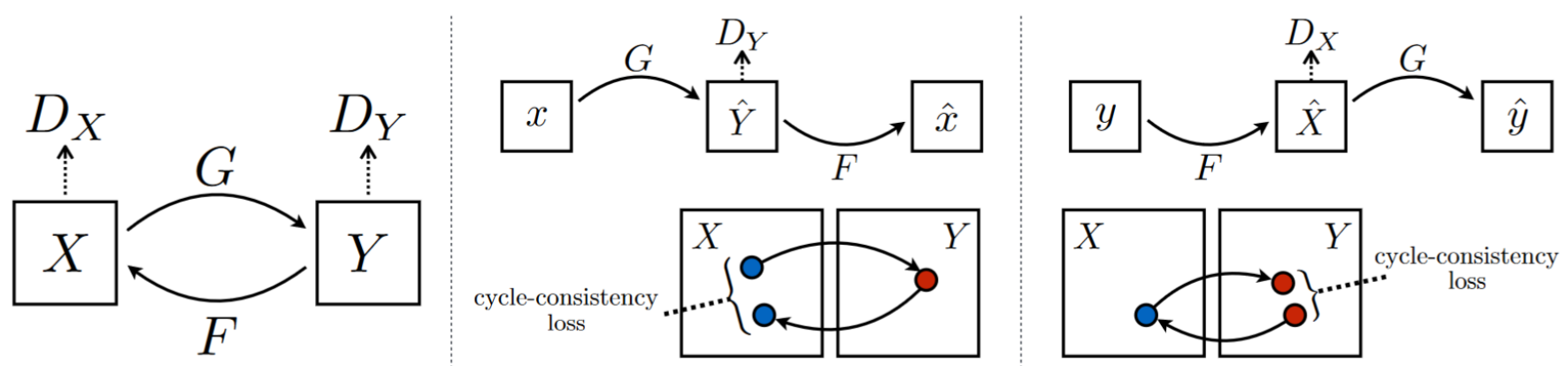
CycleGAN



The cycle transfer of CycleGAN

CycleGAN

- CycleGAN does the style transfer task with GAN structure. The main idea of the masterpiece is to utilize a full cycle of transferring.
- For CycleGAN, the two domains of data, X and Y, are both "style" and "content" at the same time, which means that the style changing takes two sides, from X to Y , and from Y to X.
- For the sake of improving quality, a **cycle-consistency loss** is proposed as below.



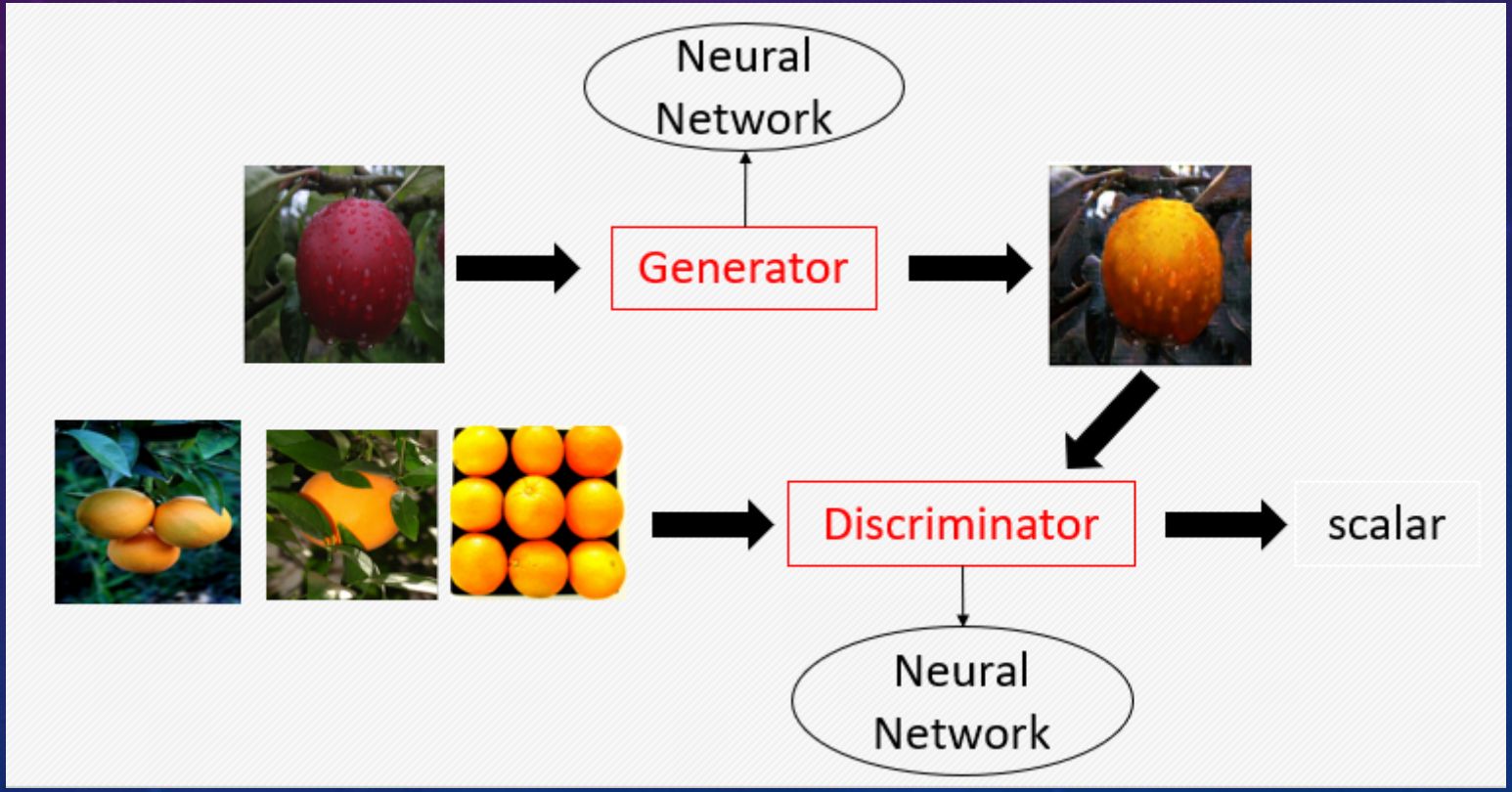
https://arxiv.org/pdf/1703.10593.pdf

CycleGAN

https://www.youtube.com/watch?v=AxrKVfjSBiA

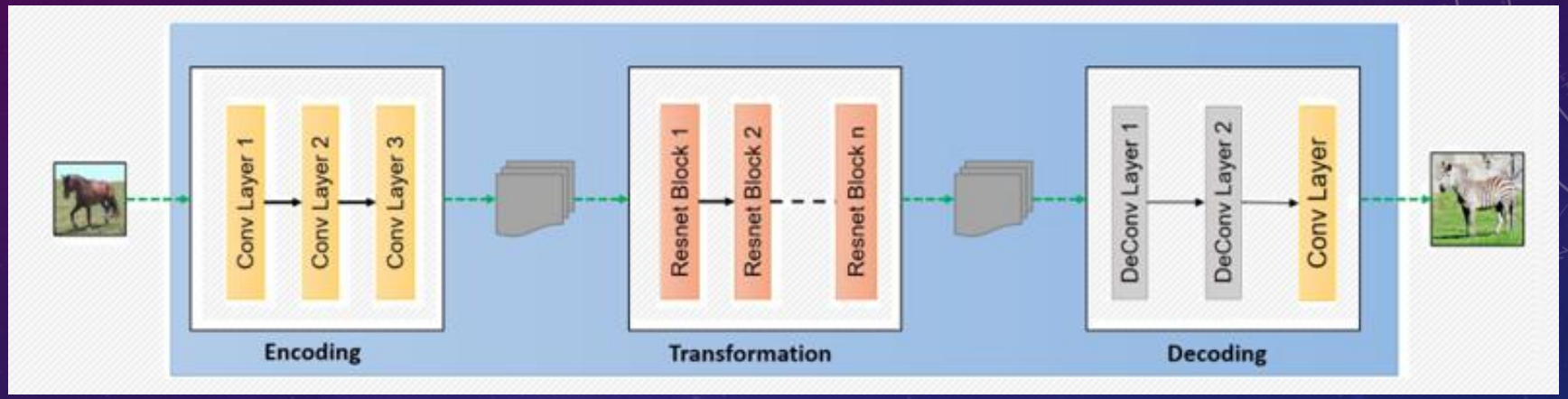
Background

- If there is no paired image, using the normal Conditional GAN architecture, the image generated by the trained network generator (X->Y) in the target domain will only be one of the images in the target distribution, and cannot be restricted. Correspondence with the original image.

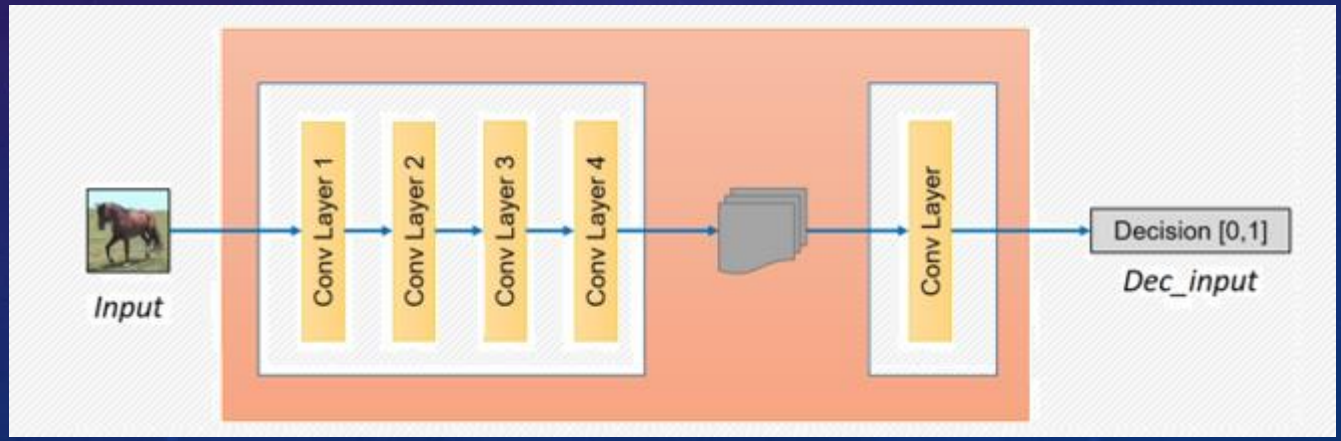


Background

Generator

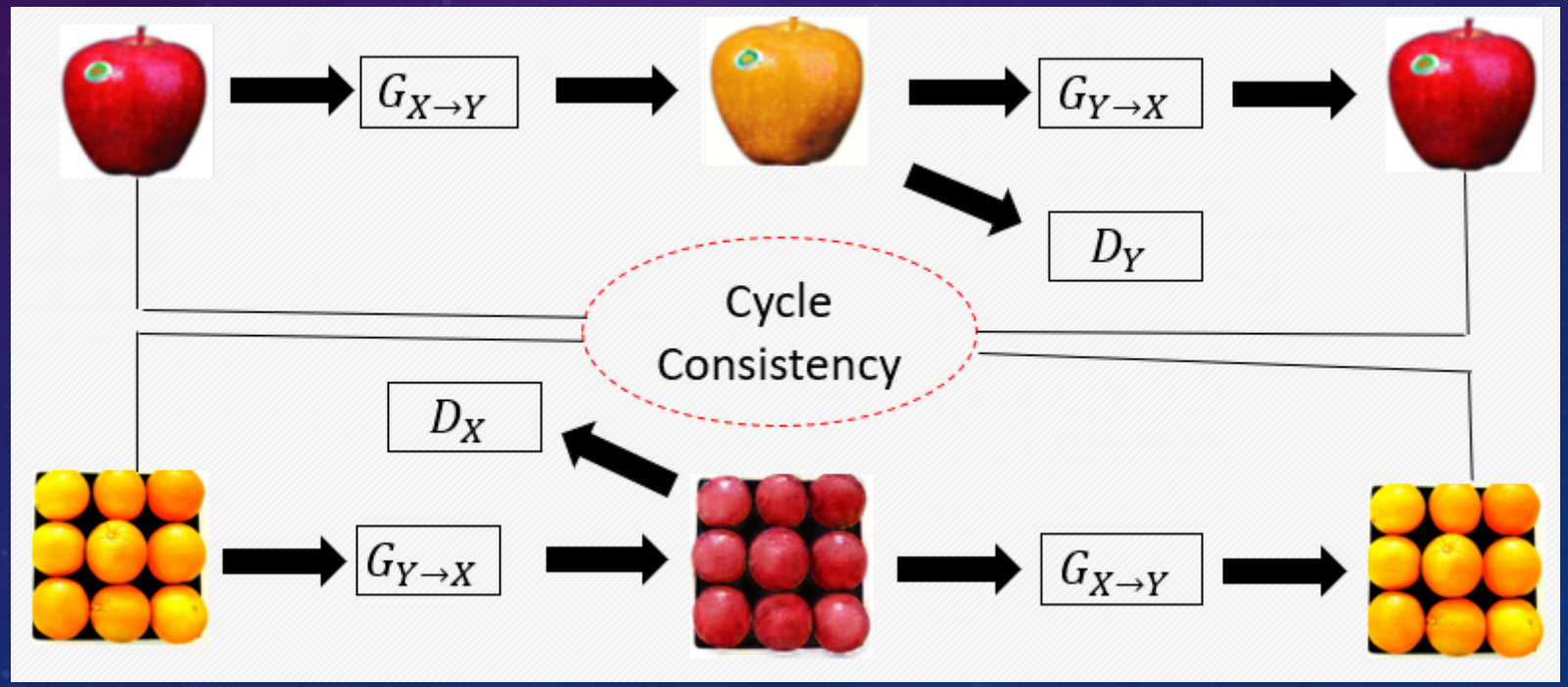


Discriminator



Background

- The author has introduced the concept of Cycle Consistency, adding another Generator (Y->X) to the original architecture, and converting the generated target image back to the original source domain, and limiting the converted map as close as possible to the original image, the corresponding image can be generated in the target domain.



Cycle GAN apple ↔ orange

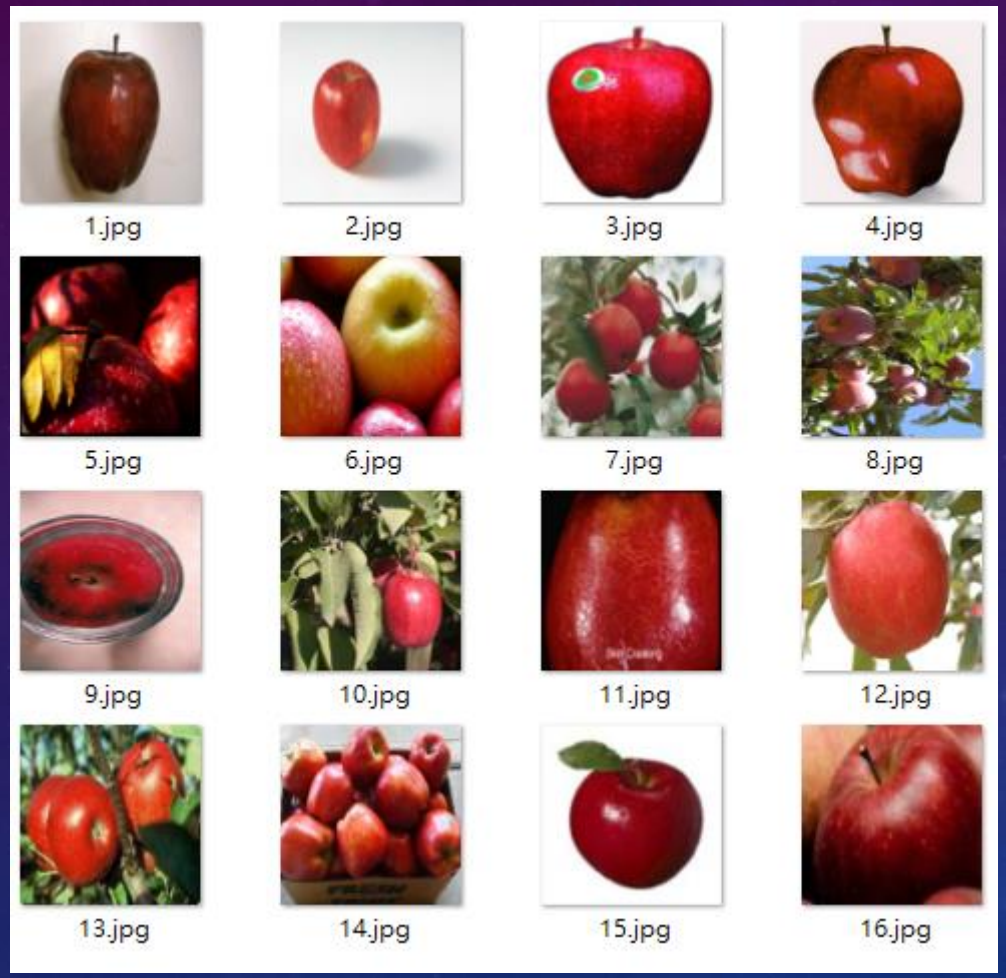
apple -> orange

Input	Output	Input	Output	Input	Output
					

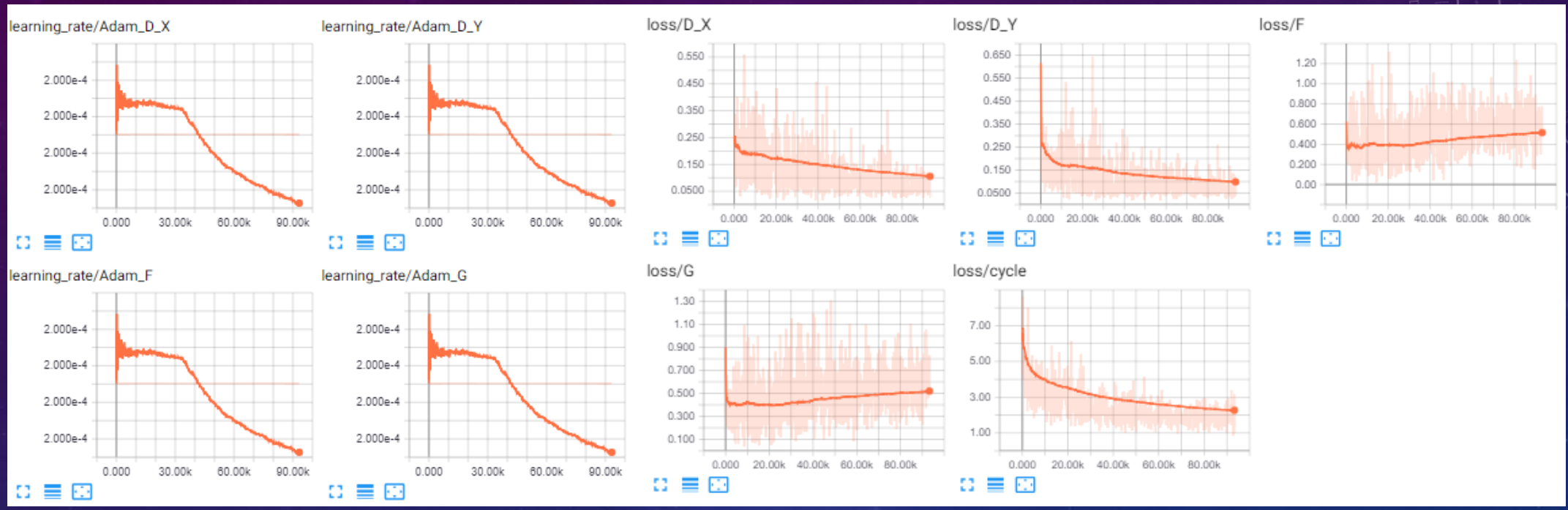
orange -> apple

Input	Output	Input	Output	Input	Output
					

apple ↔ orange



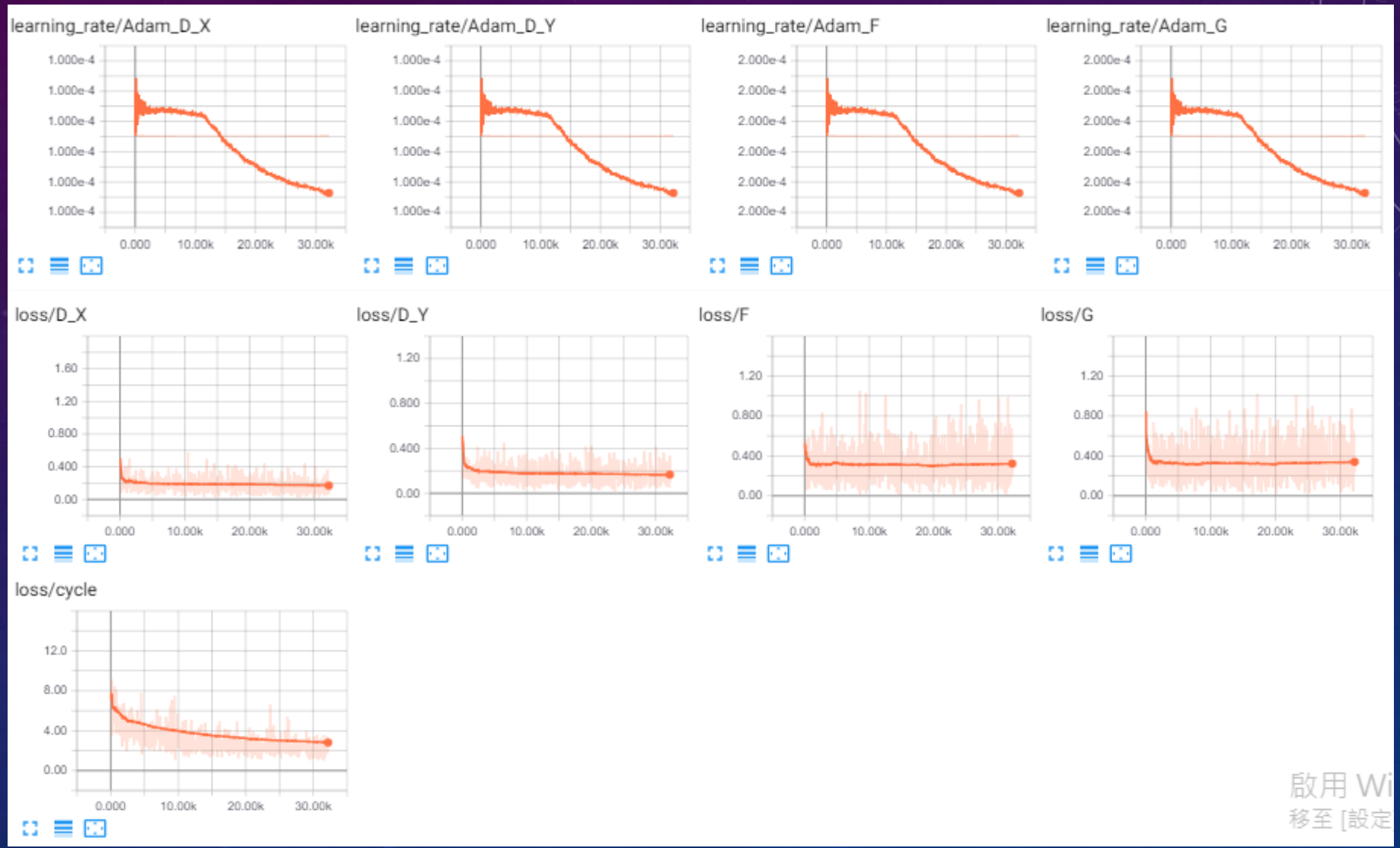
apple ↔ orange



problems: too many steps, gradient dissipation, learning rate

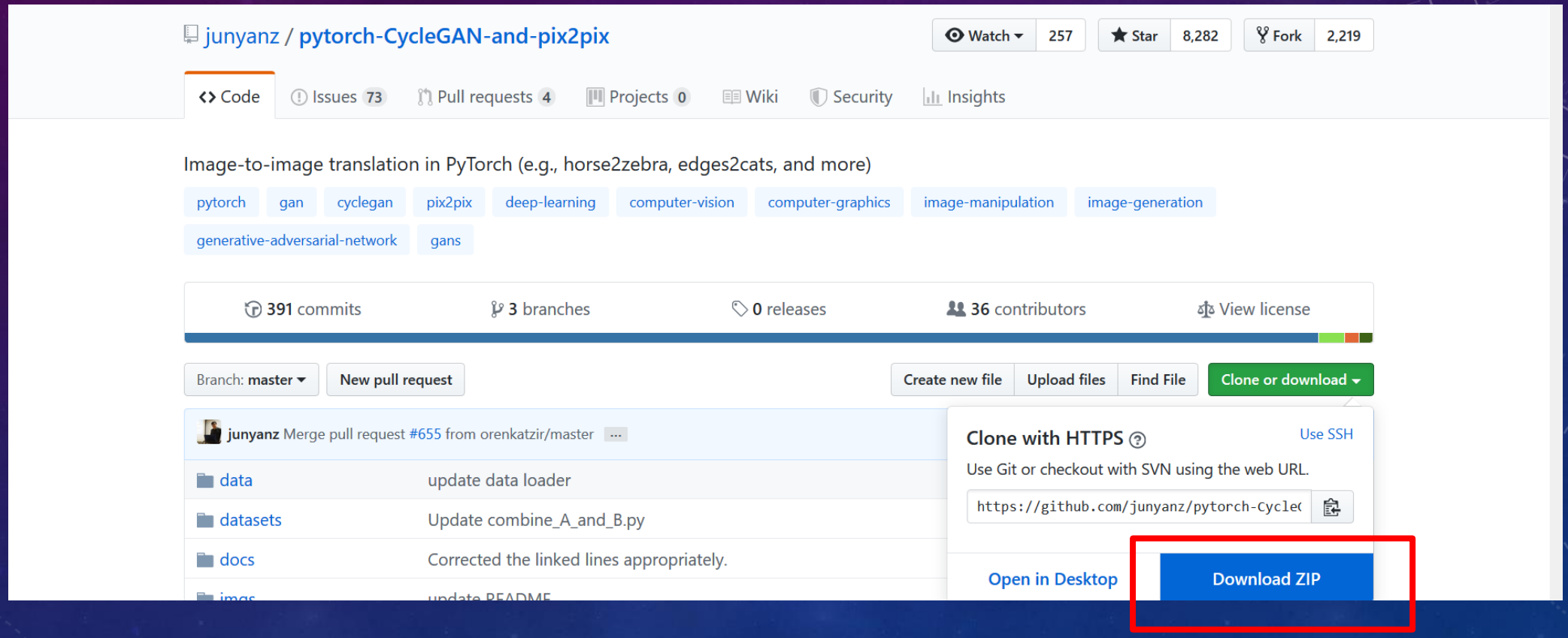
apple ↔ orange

apple ↔ orange



In Class Example and Exercise for CycleGAN

CycleGAN

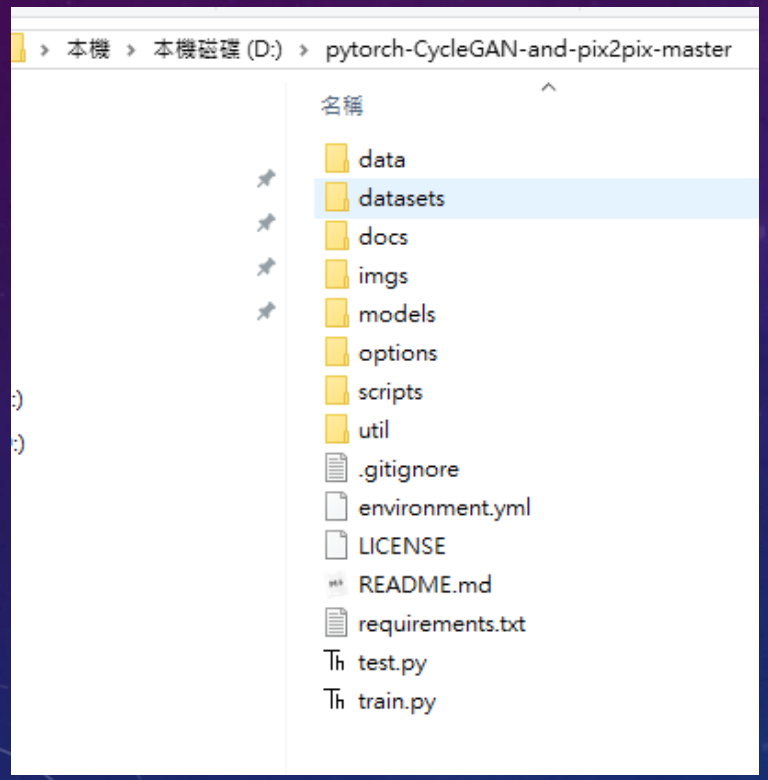


1. Please visit https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

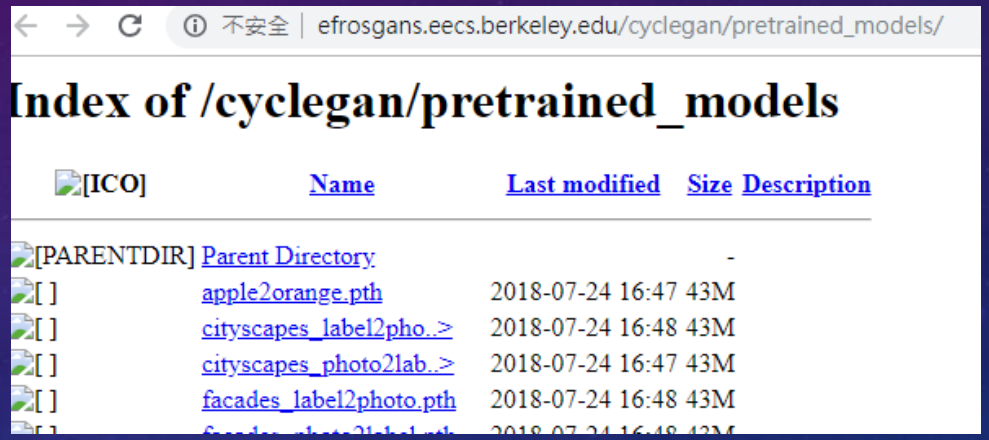
2. Download the zip file of the codes

CycleGAN

3.Unzip your code and put it in the directory desired.



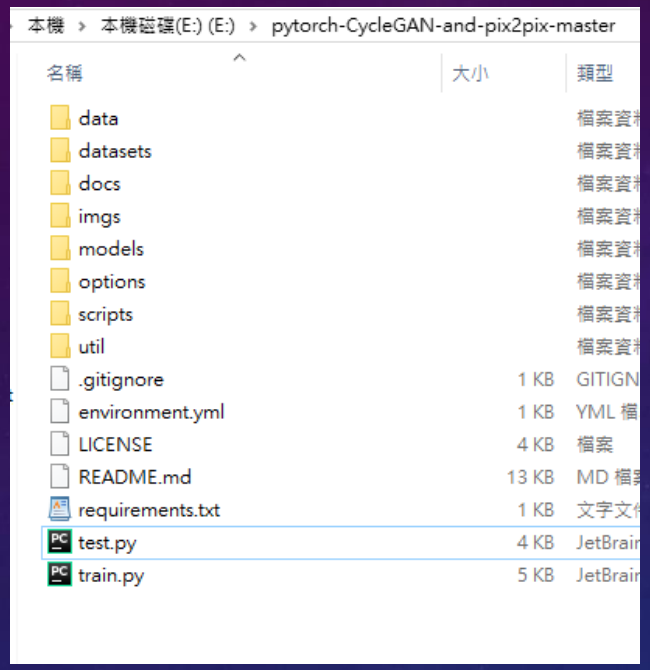
4. Visit http://efrosgans.eecs.berkeley.edu/cyclegan/pretrained_models/ for pretrained models



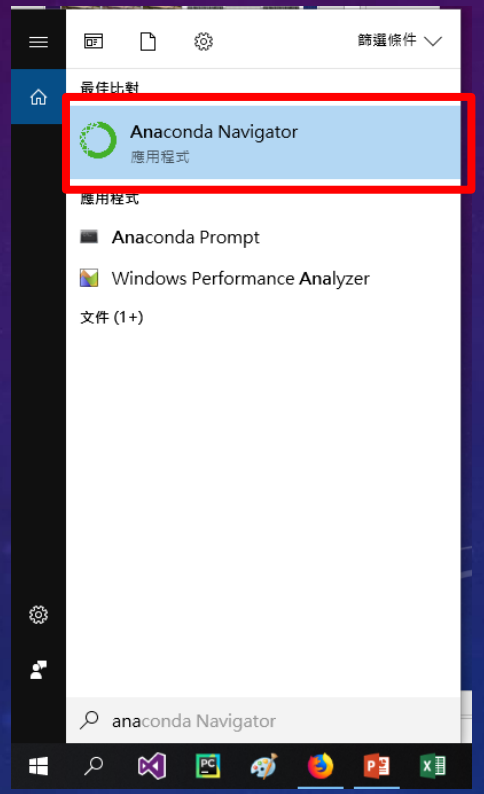
Let’s take appe2orange for example.

CycleGAN

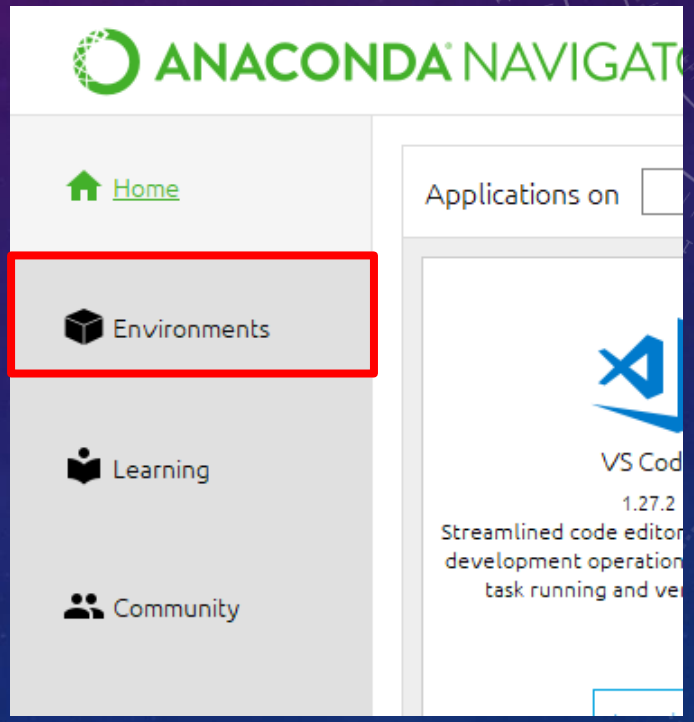
1. Extract your files to D:/



2. Open your Anaconda



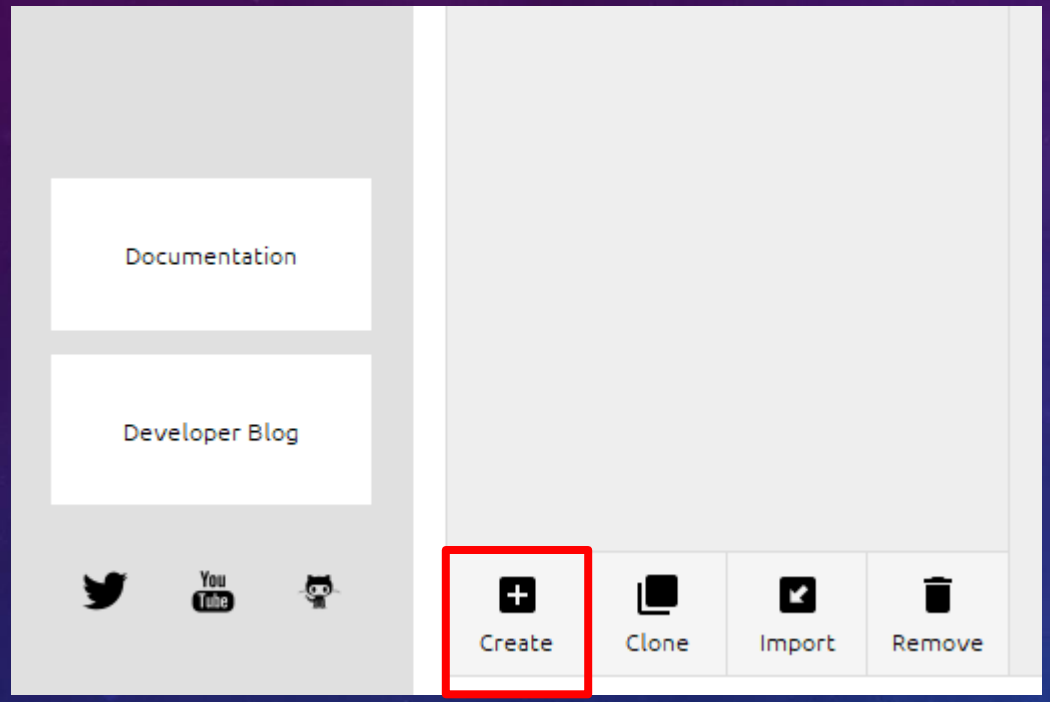
3. Click Environment



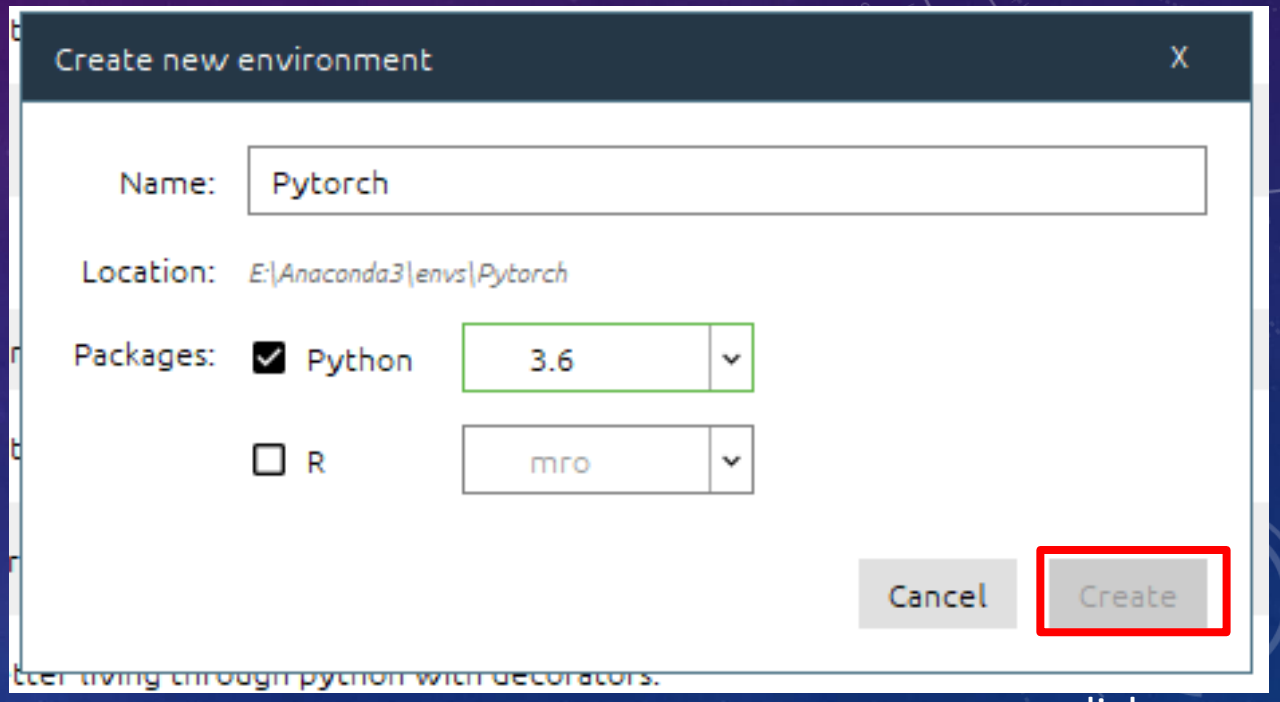
http://efrosgans.eecs.berkeley.edu/cyclegan/pretrained_models/

CycleGAN

4.Create a new environment.



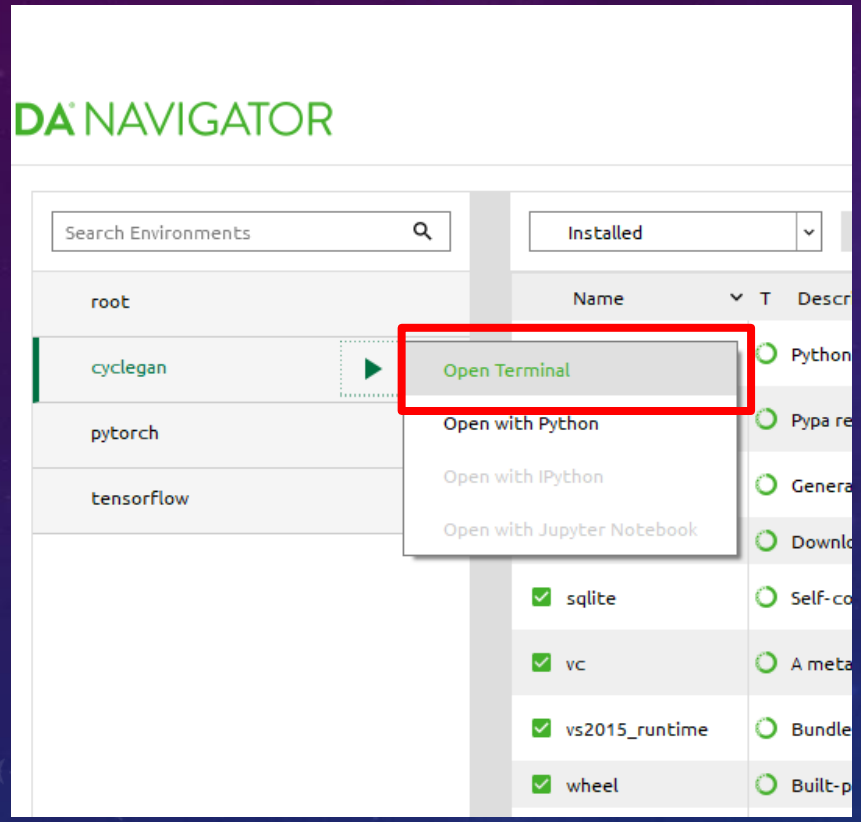
click



click

CycleGAN

5. After building steps, open your terminal.



6. Enter the folder where you put the codes

```
(C:\Anaconda3\envs\cyclegan) C:\Users\c_user>D:
(C:\Anaconda3\envs\cyclegan) D:\>cd pytorch-CycleGAN-and-pix2pix-master
(C:\Anaconda3\envs\cyclegan) D:\pytorch-CycleGAN-and-pix2pix-master>
```

7. Enter the command
"conda install pytorch=0.4.1 cuda80 –c pytorch"

```
pytorch-CycleGAN-and-pix2pix-master>conda install pytorch=0.4.1 cuda80 -c pytorch
```

8. Enter "y" for the permission

```
Proceed ([y]/n)? y
```

CycleGAN

Scipy : Please install the version of 1.2.1

```
pytorch-CycleGAN-and-pix2pix-master>pip install scipy==1.2.1
```

And other packages

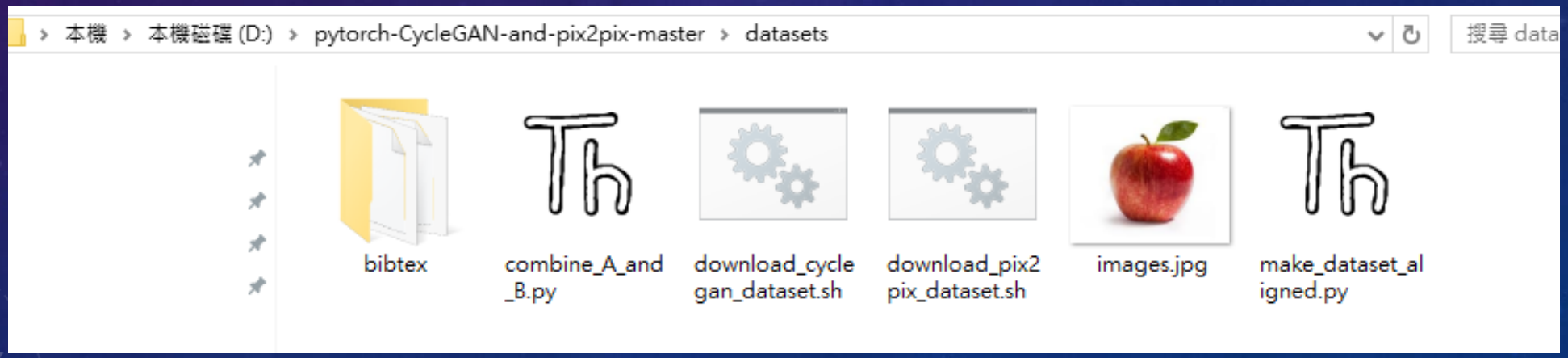
```
(C:\Anaconda3\envs\cyclegan) D:\pytorch-CycleGAN-and-pix2pix-master>pip install dominate pillow
```

```
(C:\Anaconda3\envs\cyclegan) D:\pytorch-CycleGAN-and-pix2pix-master>pip install torchvision
```

CycleGAN

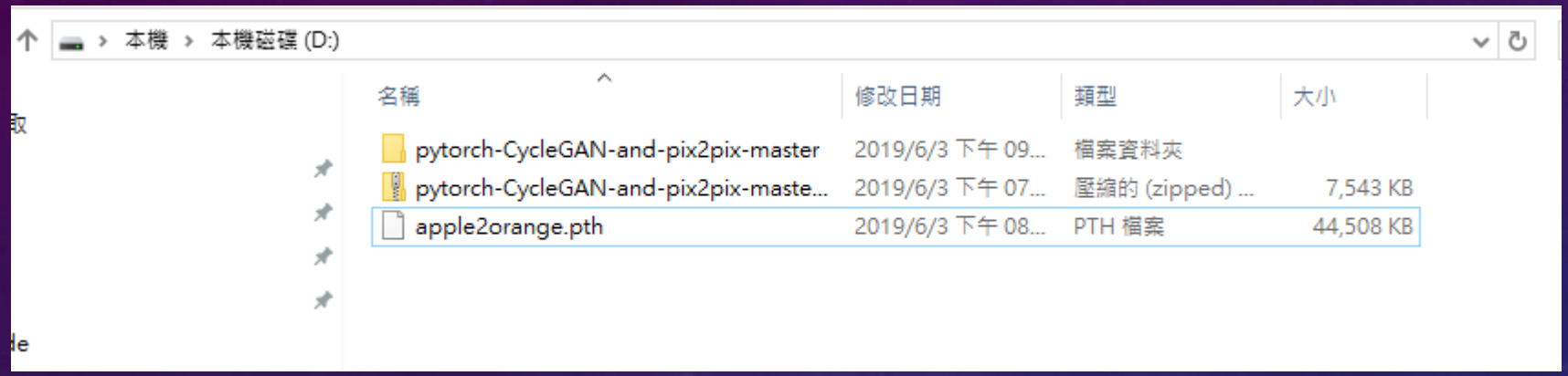
We will take “apple2orange” for example.

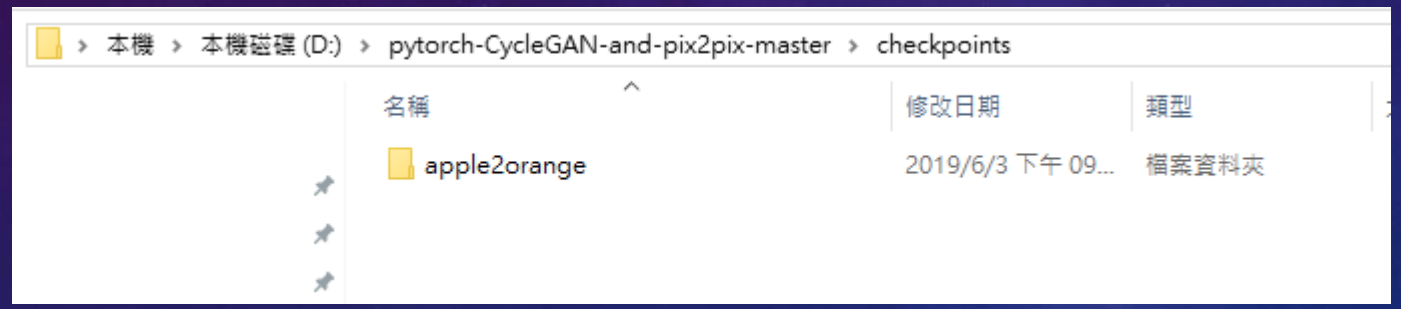
Please put your picture here



The code will find the “.jpg” file by itself.

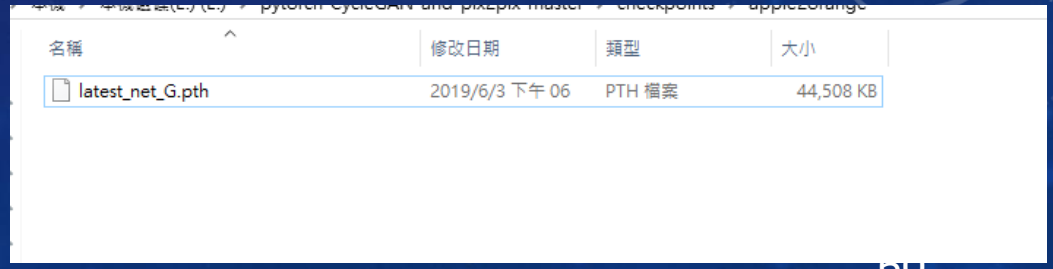
CycleGAN





1. Make the directory in <CycleGAN dir>/checkpoint/apple2orange
2. Copy it and paste it to <CycleGAN dir>/checkpoint/apple2orange/

Then , rename it as “latest_net_G.pth”



CycleGAN

- Enter the command <python test.py --dataroot datasets --model test --name apple2orange --no_dropout>

```
D:\pytorch-CycleGAN-and-pix2pix-master>python test.py --dataroot datasets --model test --name apple2orange --no_dropout
```

And you will get your results in the folder (pytorch-CycleGAN-and-pix2pix-master\results\apple2orange\test_latest\images)

CycleGAN

- We can also train a model on our own.
- We can obtain the datasets for demos at https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/
- After download the datasets, follow the commands provided on the github.

https://github.com/vanhuyz/CycleGAN-TensorFlow

Summary: What have we learned about GANs today

- Don't work with an explicit density function
- Take game-theoretic approach: learn to generate from training distribution through 2-player game
- Pros: Beautiful, state-of-the-art samples!
- Cons: Trickier / more unstable to train
- Active areas of research: Better loss functions, more stable training (Wasserstein GAN, LSGAN, many others)
- Networks you learned: CycleGAN (in depth) and many other applications of GAN

CH.4 APPENDIX

Appendix

- Unstable Gradients
- Nash equilibrium
- KL-divergence
- JS-divergence
- Basic Operations in GAN Networks (Pooling, Deconv, etc.)
- DCGAN
- Normalization (Group, Batch, Layer, etc.)

Unstable gradients

- An alternative cost function is proposed by the original GAN paper to address the gradient vanishing problem.

$$\nabla_{\theta_g} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \rightarrow 0 \quad \textit{change to} \quad \nabla_{\theta_g} - \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$$

- The corresponding gradient according to a different research paper from Arjovsky is:

$$\mathbb{E}_{z\sim p(z)}\left[-\nabla_\theta \log D^*(g_\theta(z))|_{\theta=\theta_0}\right] = \nabla_\theta\left[KL(\mathbb{P}_{g_\theta}\|\mathbb{P}_r) - 2JSD(\mathbb{P}_{g_\theta}\|\mathbb{P}_r)\right]|_{\theta=\theta_0}$$

  - It includes a reverse KL-divergence term .
  - Arjovsky uses it to explain why GAN has higher quality but less diverse image comparing to generative models based on KL-divergence.
  - But the gradients fluctuate and cause instability to the model.

Unstable gradients

- Arjovsky freezes the generator and trains the discrminator continuously.

  The gradient for the generator starts increasing with larger variants.

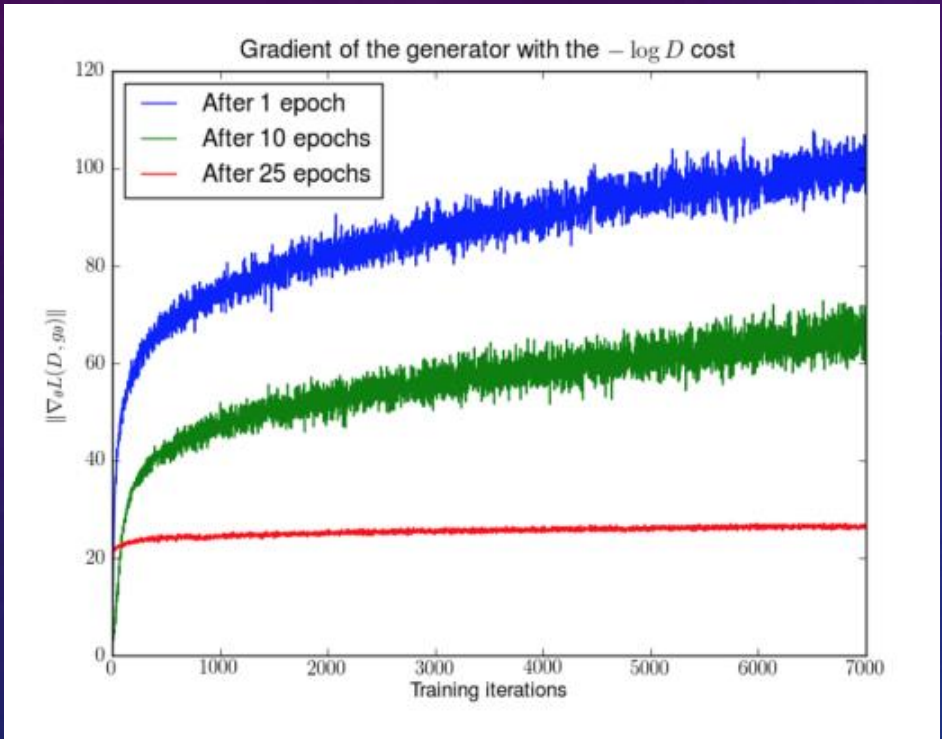


Figure 3: First, we trained a DCGAN for 1, 10 and 25 epochs. Then, with the generator fixed we train a discriminator from scratch and measure the gradients with the $-\log D$ cost function. We see the gradient norms grow quickly. Furthermore, the noise in the curves shows that the variance of the gradients is also increasing. All these gradients lead to updates that lower sample quality notoriously.

Source

- The first GAN generator's objective function has vanishing gradients and the alternative cost function has fluctuating gradients that cause instability to the models.

Nash equilibrium

- GAN is based on the zero-sum game (minimax).
- In game theory, the GAN model converges when the discriminator and the generator reach a Nash equilibrium. This is the optimal point for the minimax equation below.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P_r(x)}[logD(x)] + \mathbb{E}_{z\sim P_z(z)}[\log(1 - D(G(z))]$$

Nash equilibrium

- Consider two player A and B which control the value x and y respectively. Player A wants to maximize the value xy while B wants to minimize it.

$$\min_B \max_A V(D,G) = xy$$

The Nash equilibrium is x=y=0. This is the only state that any opponents' actions will not change the game outcome.

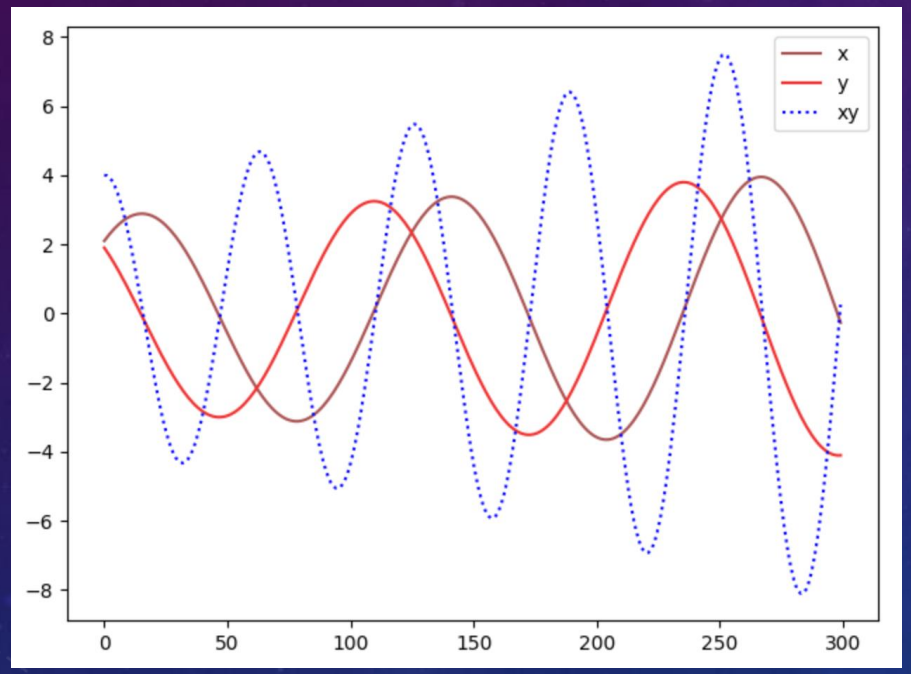
- We can use the gradient descent to find the Nash equilibrium easily.

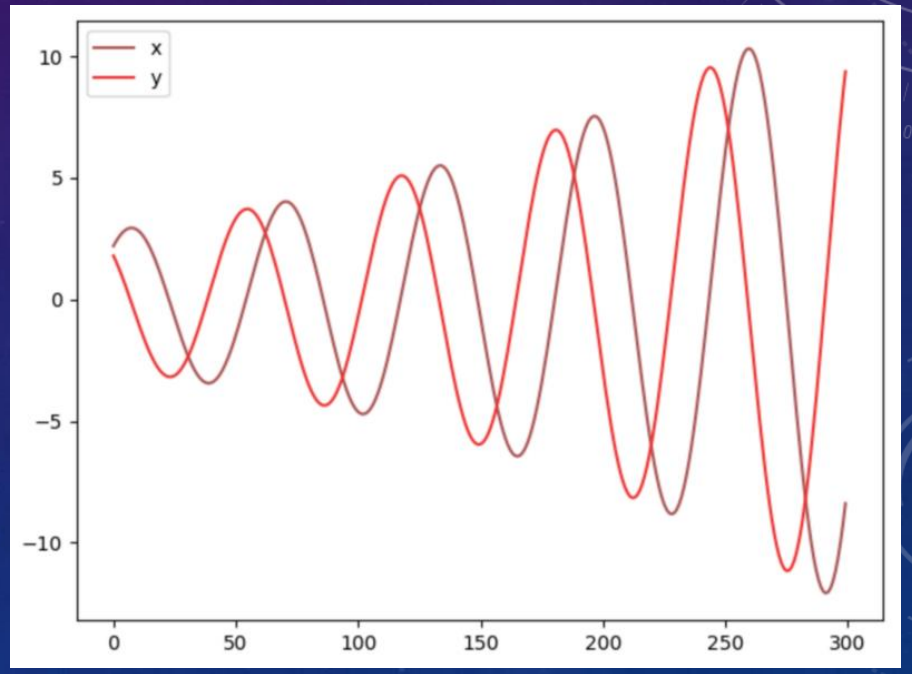
$$\Delta x = \alpha \frac{\partial(xy)}{\partial(x)} \qquad \Delta y = -\alpha \frac{\partial(xy)}{\partial(y)}$$

where α is the learning rate.

Nash equilibrium

- We can find that the solution does not converge in the training iterations.
- After increasing the learning rate or training the model longer, we can see the parameters x, y is unstable with big swings.





Cost functions may not converge using gradient descent in a minimax game.

Generative Model with Kullback-Leibler (KL) Divergence

- Before GAN, many generative models create a model θ that maximizes the **Maximum Likelihood Estimation** (MLE).

  i.e. finding the best model parameters that fit the training data the most.

$$\hat{\theta} = \arg\max_{\theta} \prod_{i=1}^{N} p(x_i|\theta)$$

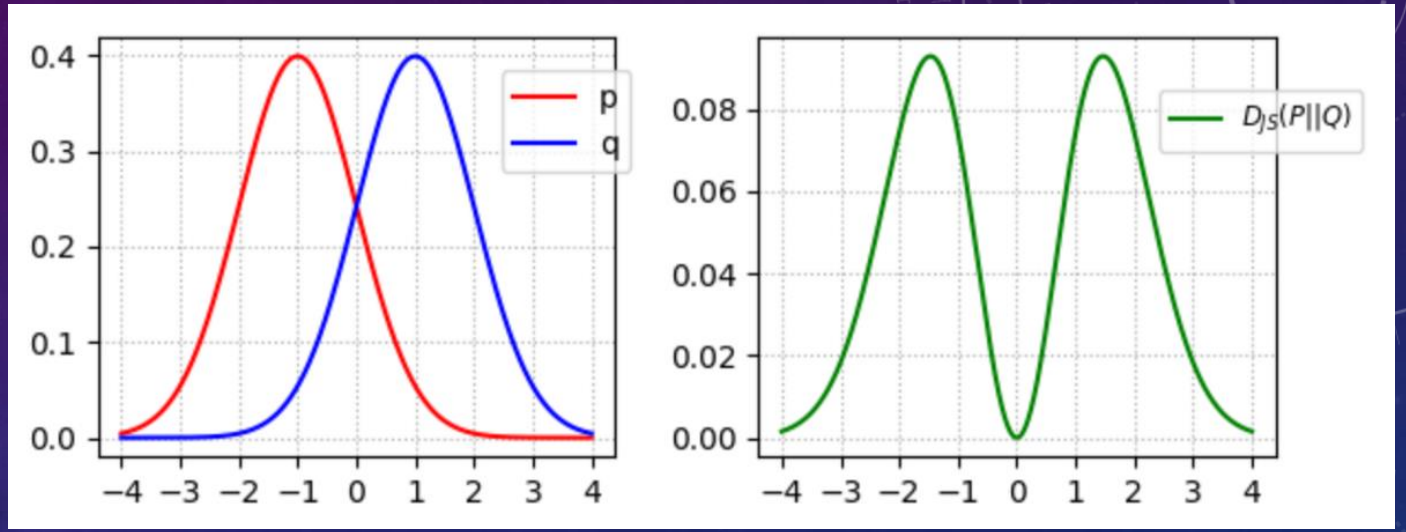
- This is the same as minimizing the **KL-divergence** KL(p,q) which measures how the probability distribution q diverges from the expected probability distribution p.

$$D_{KL}(p||q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

JS-Divergence

- JS-divergence is defined as:

$$D_{JS}(p||q) = \frac{1}{2}D_{KL}(p||\frac{p+q}{2}) + \frac{1}{2}D_{KL}(q||\frac{p+q}{2})$$



- JS-divergence is symmetrical , so it will penalize poor images badly.
  (when p(x)→ 0 and q(x) > 0)
- In GAN, if the discriminator is optimal, the generator's objective function becomes :

$$\min_G V(D^*, G) = 2D_{JS}(p_r \| p_g) - 2\log 2$$

Proof (GAN optimal point)

- Find the optimal value for V:

$$\min_G V(D^*, G) = \int_x \left(p_r(x) \log D^*(x) + p_g(x) \log(1 - D^*(x)) \right) dx$$
$$= \int_x \left(p_r(x) \log \frac{p_r(x)}{p_r(x) + p_g(x)} + p_g(x) \log \frac{p_g(x)}{p_r(x) + p_g(x)} \right) dx$$

$$D_{JS}(p_r \| p_g) = \frac{1}{2} D_{KL}(p_r || \frac{p_r + p_g}{2}) + \frac{1}{2} D_{KL}(p_g || \frac{p_r + p_g}{2})$$
$$= \frac{1}{2} \left(\int_x p_r(x) \log \frac{2p_r(x)}{p_r(x) + p_g(x)} dx \right) + \frac{1}{2} \left(\int_x p_g(x) \log \frac{2p_g(x)}{p_r(x) + p_g(x)} dx \right)$$
$$= \frac{1}{2} \left(\log 2 + \int_x p_r(x) \log \frac{p_r(x)}{p_r(x) + p_g(x)} dx \right) +$$
$$\frac{1}{2} \left(\log 2 + \int_x p_g(x) \log \frac{p_g(x)}{p_r(x) + p_g(x)} dx \right)$$
$$= \frac{1}{2} \left(\log 4 + \min_G V(D^*, G) \right)$$

$$\min_G V(D^*, G) = 2D_{JS}(p_r \| p_g) - 2 \log 2$$

Proof (GAN optimal point)

- From the equation below, the optimal point for V is when $p = q$. (when D_JS = 0)

$$\min_G V(D^*, G) = 2D_{JS}(p_r \| p_g) - 2\log 2$$

With $p = q$, the optimal value for D and V is

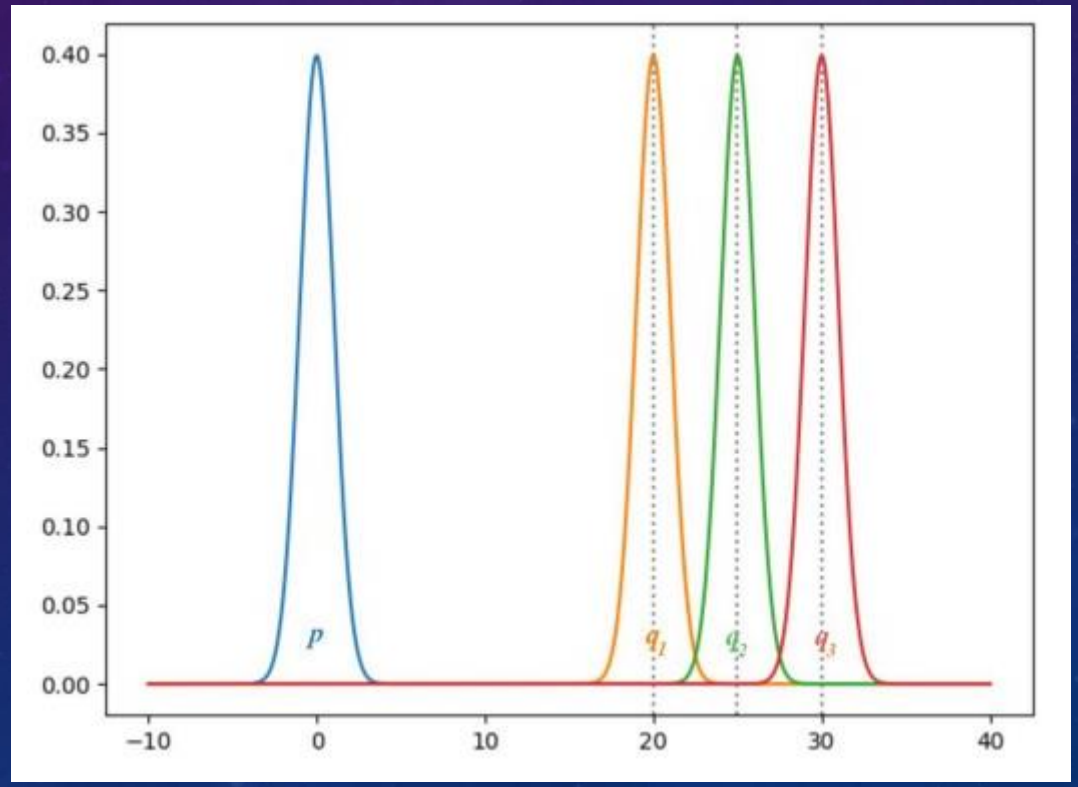
$$D^*(x) = \frac{p}{p+q} = \frac{1}{2}$$

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_r(x)}[\log \frac{1}{2}] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - \frac{1}{2})]$$
$$= -2\log 2$$

Vanishing gradients in JS-Divergence

- What happens to the JS-divergence gradient when the data distribution q of the generator's images does not match with the ground truth p for the real images.
- For example, p and q are Gaussian distributed and the mean of p is zero.

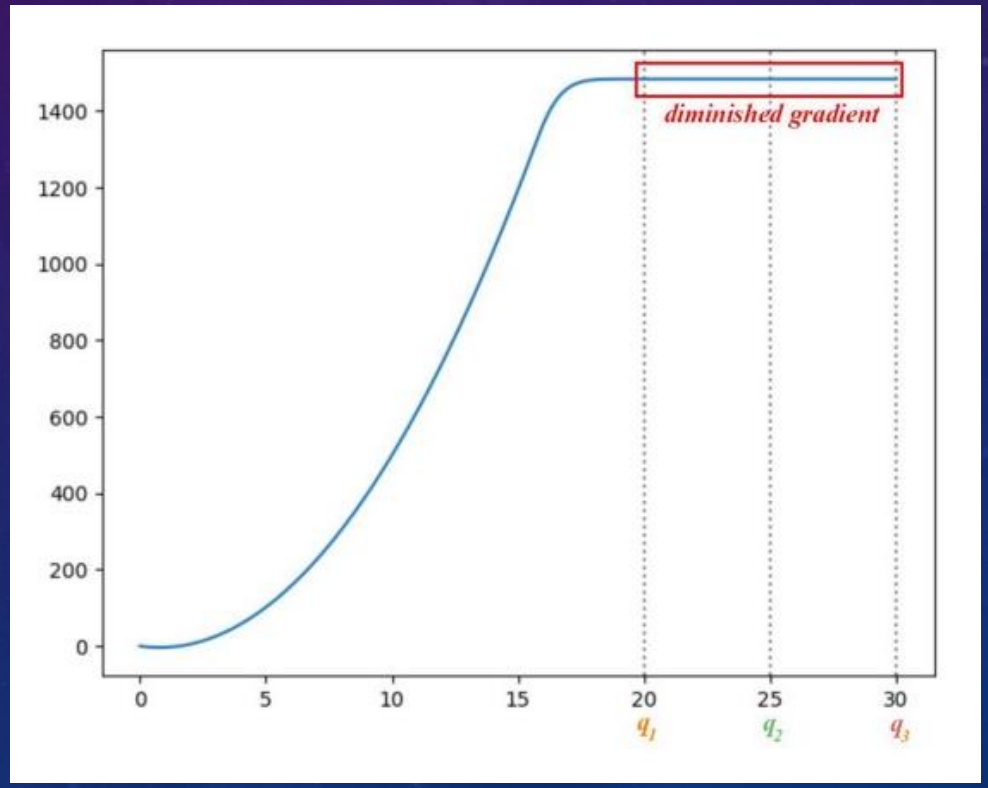
  We consider that q with different means to study the gradient of JS(p, q)



Vanishing gradients in JS-Divergence

- We plot the JS-divergence JS(p, q) between p and q with means of q ranging from 0 to 30.
- As shown below, the gradient for the JS-divergence vanishes from $q1$ to $q3$.

The GAN generator will learn extremely slow to nothing when cost is saturated in those regions.



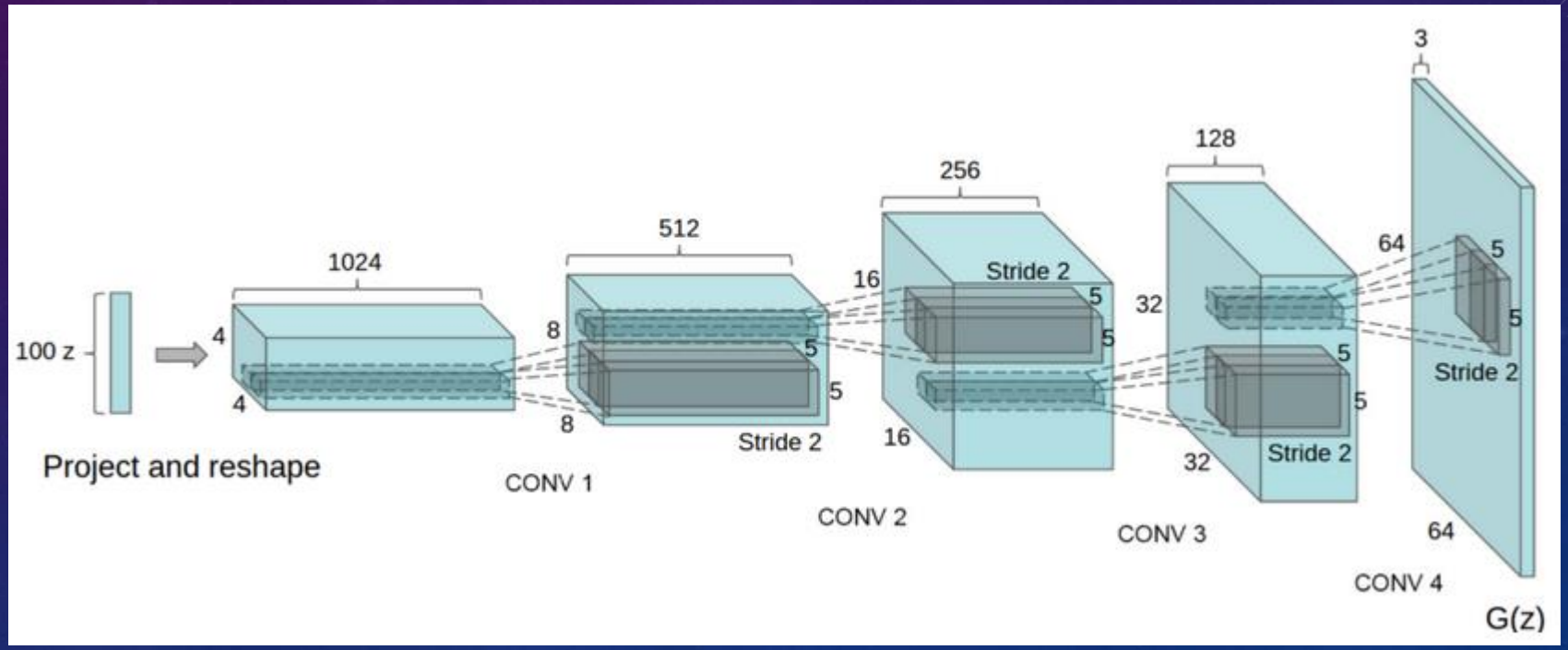
GAN — DCGAN (Deep convolutional generative adversarial networks)

Jonathan Hui

https://medium.com/@jonathan_hui/gan-dcgan-deep-convolutional-generative-adversarial-networks-df855c438f

DCGAN

One of the most popular network design for GANs (or originated from) is DCGAN.



DCGAN

It gets rid of max-pooling which destroys spatial information and hurts the image quality. Its main design includes :

- Replace all max pooling with convolutional stride,
- Use transposed convolution for upsampling,
- Eliminate fully connected layers, and
- Use Batch normalization **BN**. (Video shows the effects of **BN**)

Batch normalization is one measure

- Batch normalization is a method we can use to normalize the inputs of each layer in order to fight the internal covariate shift problem.

- During training time, a batch normalization layer does the following:

  1. Calculate the mean and variance of the layers input.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{Batch mean}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad \text{Batch variance}$$

Batch normalization

2. Normalize the layer inputs using the previously calculated batch statistics.

$$\overline{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

3. Scale and shift in order to obtain the output of the layer.

$$y_i = \gamma \overline{x_i} + \beta$$

Notice that γ and β are learned during training along with the original parameters of the network.

Batch normalization

Therefore, if each batch had *m* samples and there where *j* batches:

$$E_x = \frac{1}{m}\sum_{i=1}^{j}\mu_B^{(i)} \qquad \text{Inference mean}$$

$$Var_x = \left(\frac{m}{m-1}\right)\frac{1}{m}\sum_{i=1}^{j}\sigma_B^{2(i)} \qquad \text{Inference variance}$$

$$y = \frac{\gamma}{\sqrt{Var_x + \epsilon}}x + \left(\beta + \frac{\gamma E_x}{\sqrt{Var_x + \epsilon}}\right) \qquad \text{Inference scaling/shifting}$$

Other forms of Normalization

C= Channel axis

N = batch axis

H, W = spatial axis

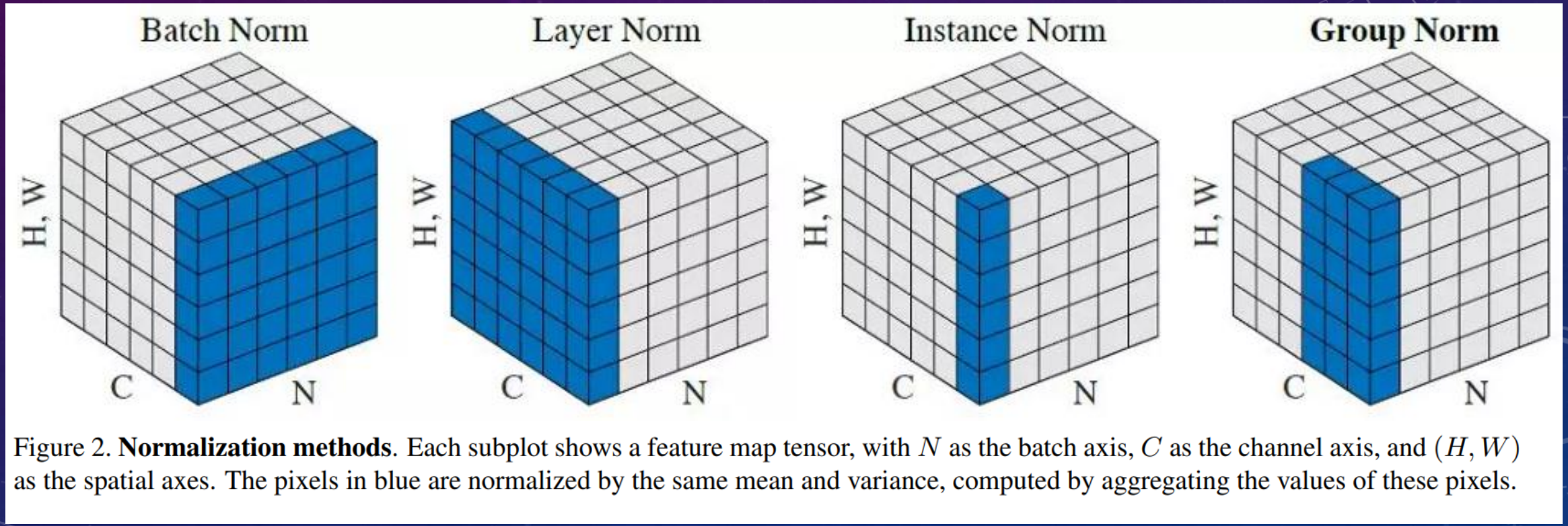


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

From the Paper "Group Normalization" by Yuxin Wu and Kaiming He, June 2018

https://arxiv.org/pdf/1803.08494.pdf