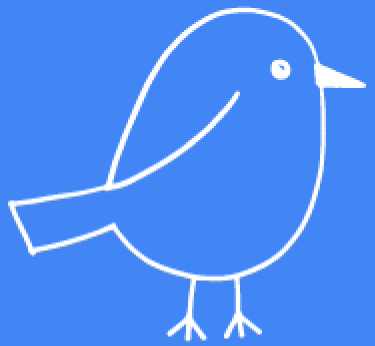
Generative Adversarial Networks



Yunjey Choi
Korea University
DAVIAN LAB

Speaker Introduction

yunjey

Add a bio

Edit profile

Korea University

Seoul, Korea

yunjey.choi@gmail.com

B.S. in Computer Science & Engineering at Korea University

M.S. Student in Computer Science & Engineering at Korea University (Current)

Interest: Deep Learning, TensorFlow, PyTorch

GitHub Link: https://github.com/yunjey

Referenced Slides

- Namju Kim. Generative Adversarial Networks (GAN)

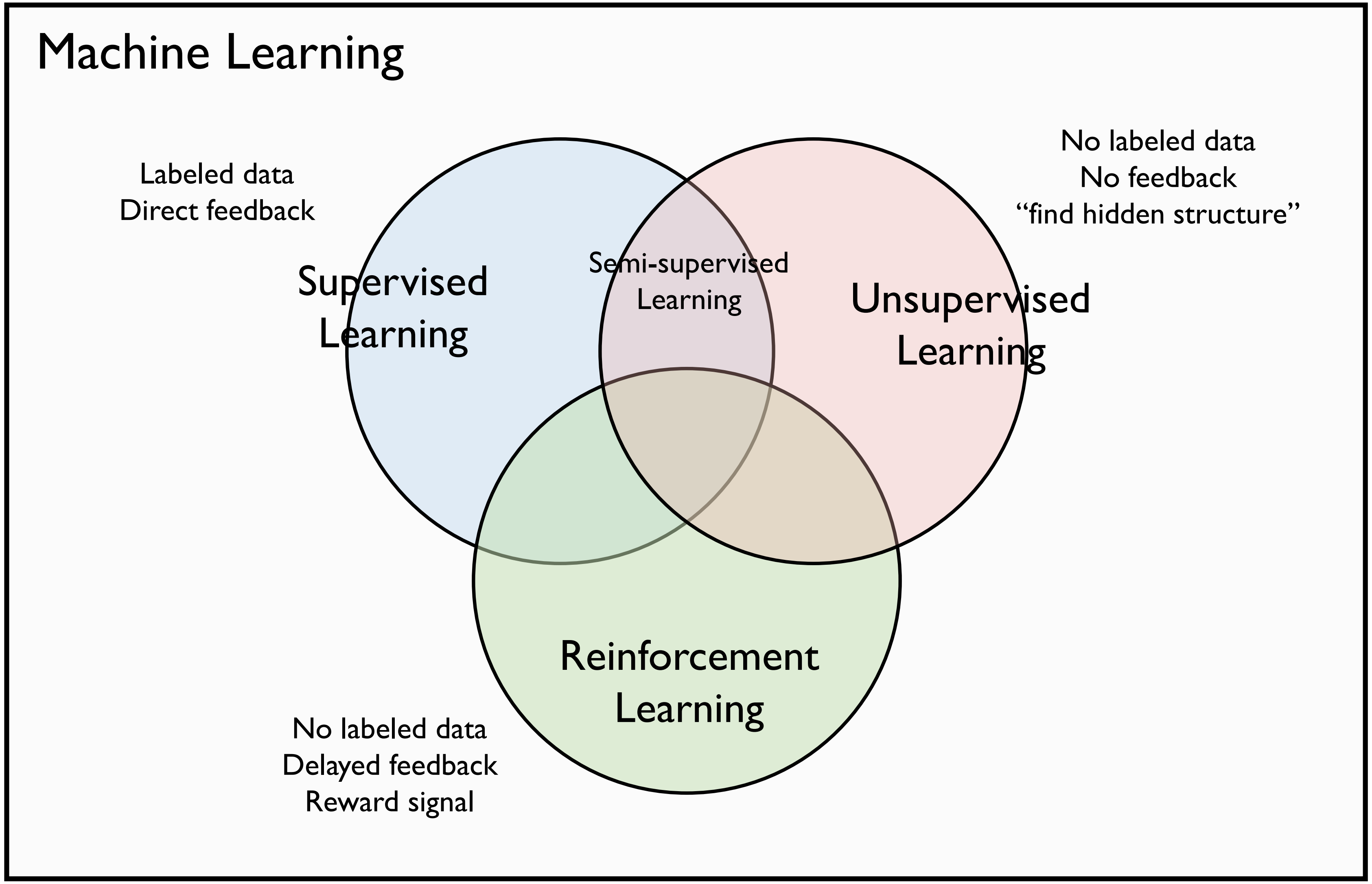
  https://www.slideshare.net/ssuser77ee21/generative-adversarial-networks-70896091

- Taehoon Kim. 지적 대화를 위한 깊고 넓은 딥러닝

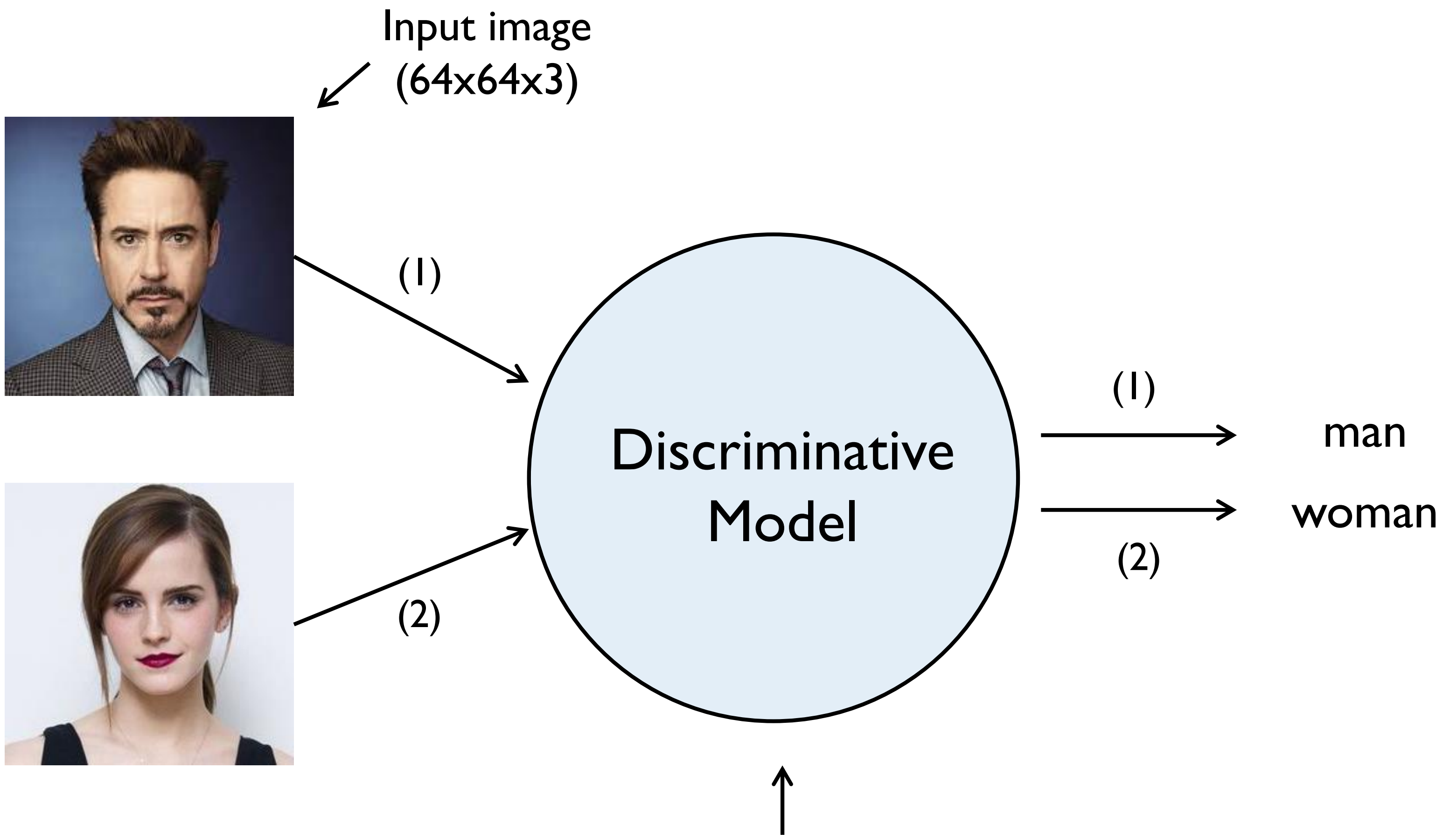
  https://www.slideshare.net/carpedm20/ss-63116251

01
Introduction

Branches of ML



Supervised Learning



The discriminative model learns how to classify input to its class.

Unsupervised Learning

Latent code (100)

Generative Model

Image (64x64x3)

The generative model learns the distribution of training data.

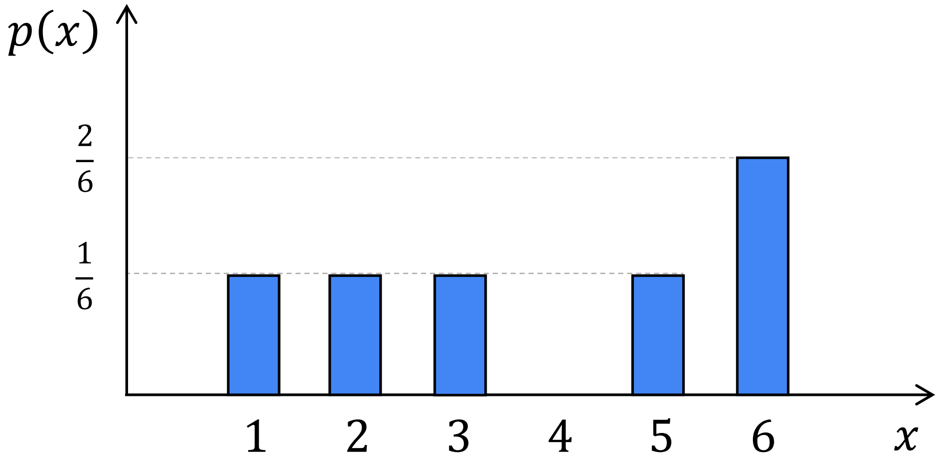
Probability Distribution

Probability Basics (Review)

Random variable

X	1	2	3	4	5	6
$P(X)$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{0}{6}$	$\frac{1}{6}$	$\frac{2}{6}$

Probability mass function



Probability Distribution

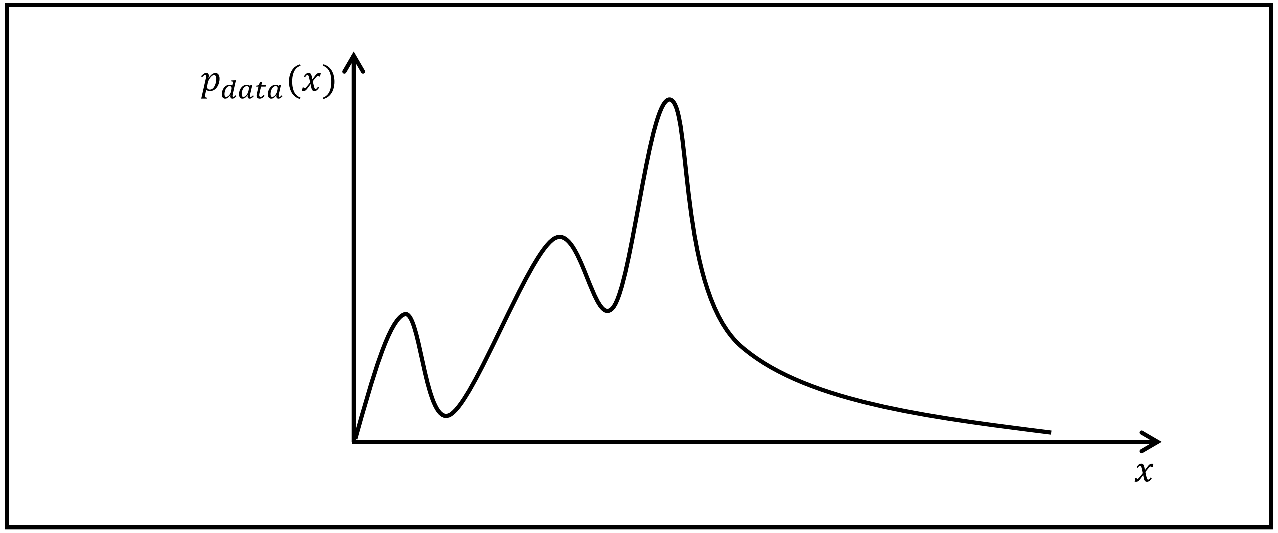
What if x is actual images in the training data?

At this point, x can be represented as a (for example) 64x64x3 dimensional vector.

Probability Distribution

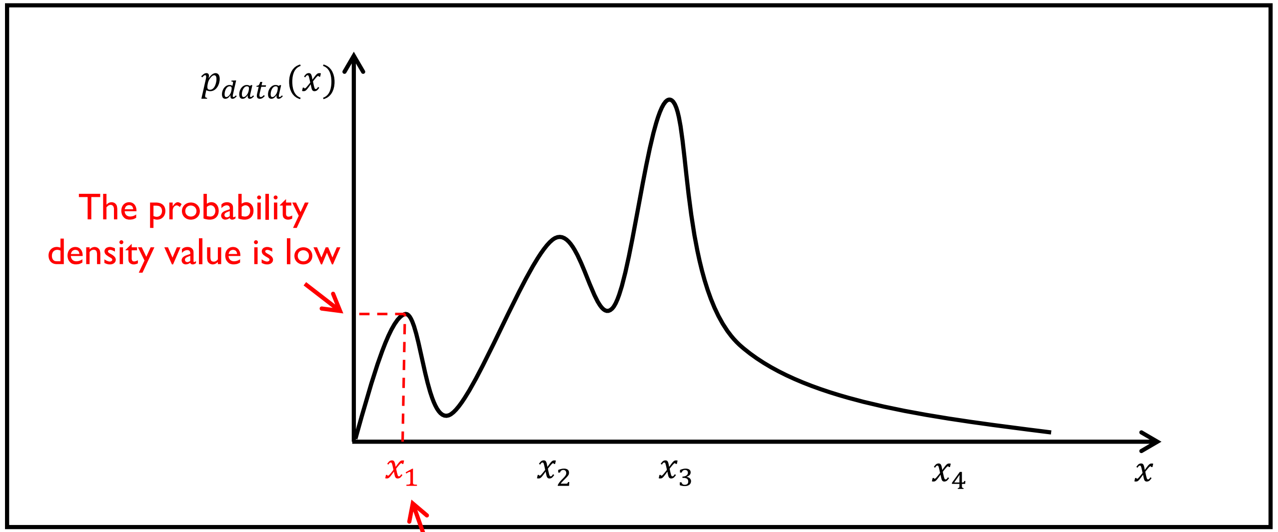
Probability density function

There is a $p_{data}(x)$ that represents the distribution of actual images.



Probability Distribution

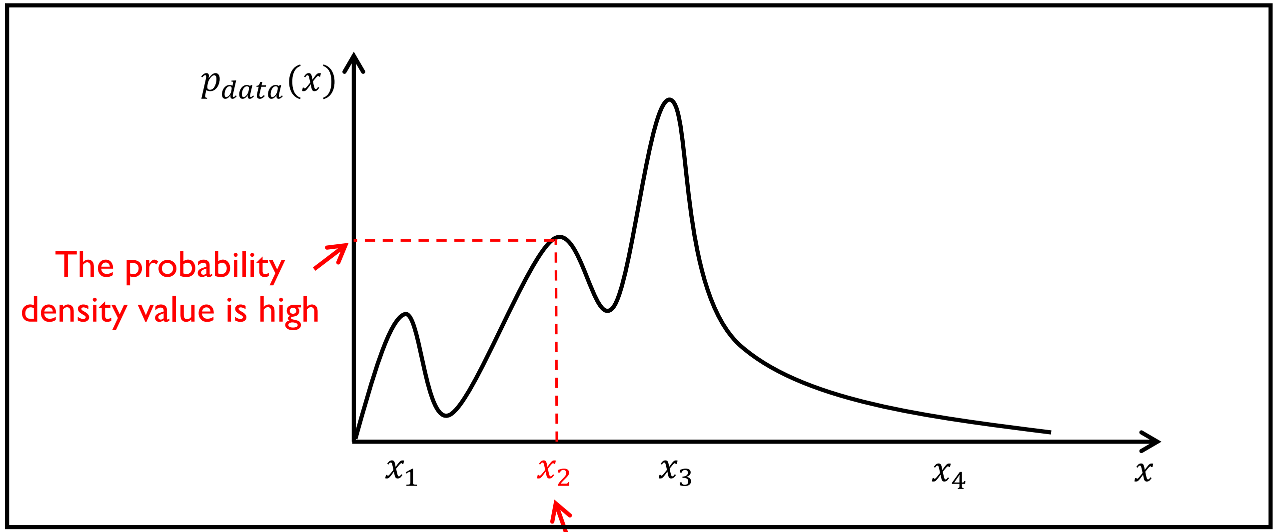
Let's take an example with human face image dataset.
Our dataset may contain few images of men with glasses.



x_1 is a 64x64x3 high dimensional vector representing a man with glasses.

Probability Distribution

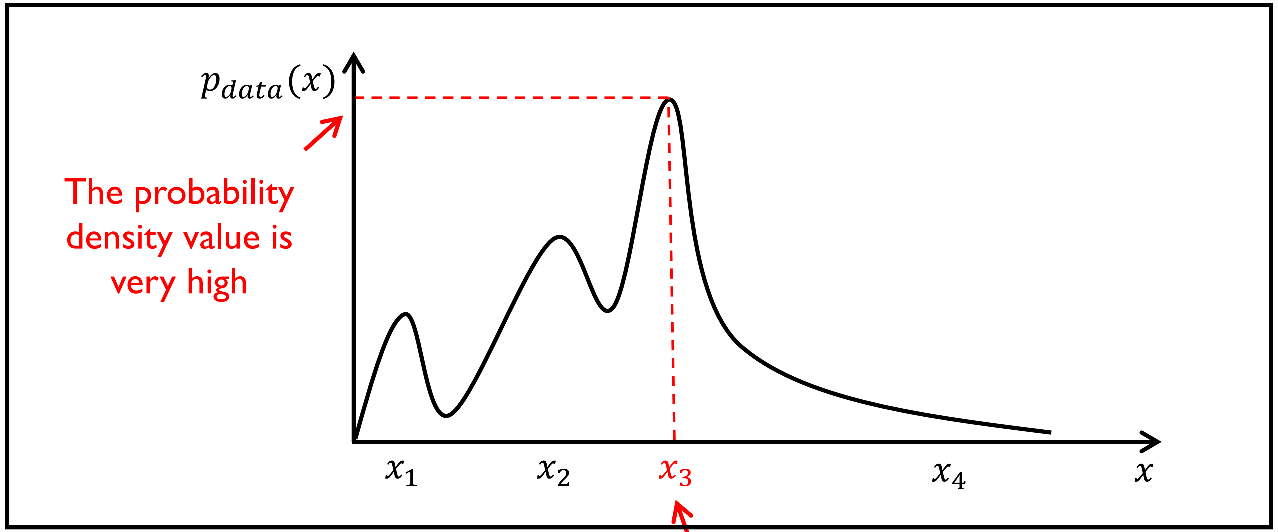
Our dataset may contain many images of women with black hair.



x_2 is a 64x64x3 high dimensional vector representing a woman with black hair.

Probability Distribution

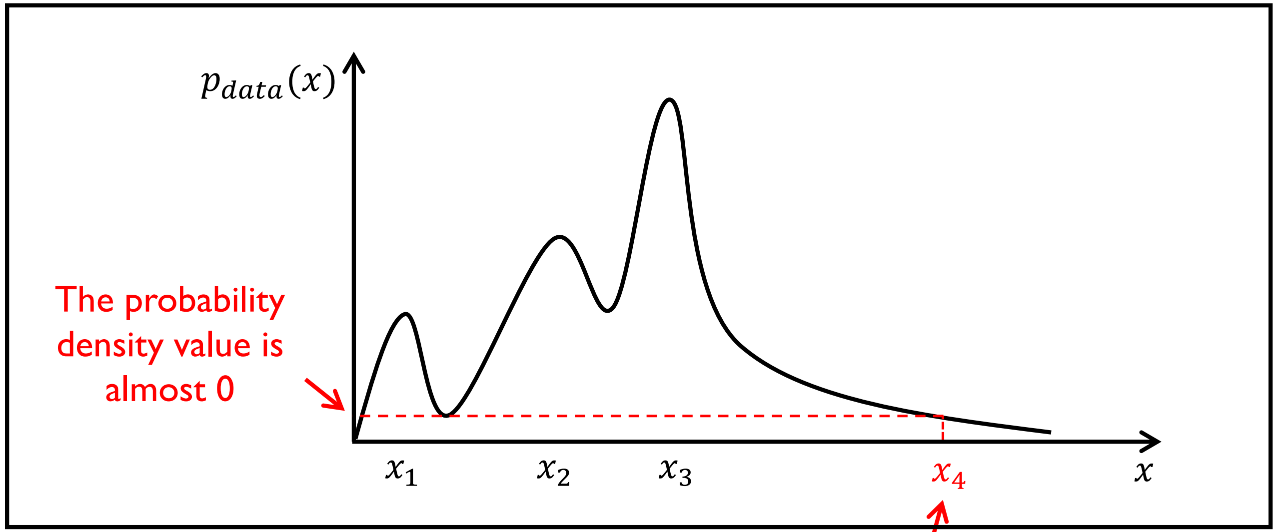
Our dataset may contain very many images of women with blonde hair.



x_3 is a 64x64x3 high dimensional vector representing a woman with blonde hair.

Probability Distribution

Our dataset may not contain these strange images.



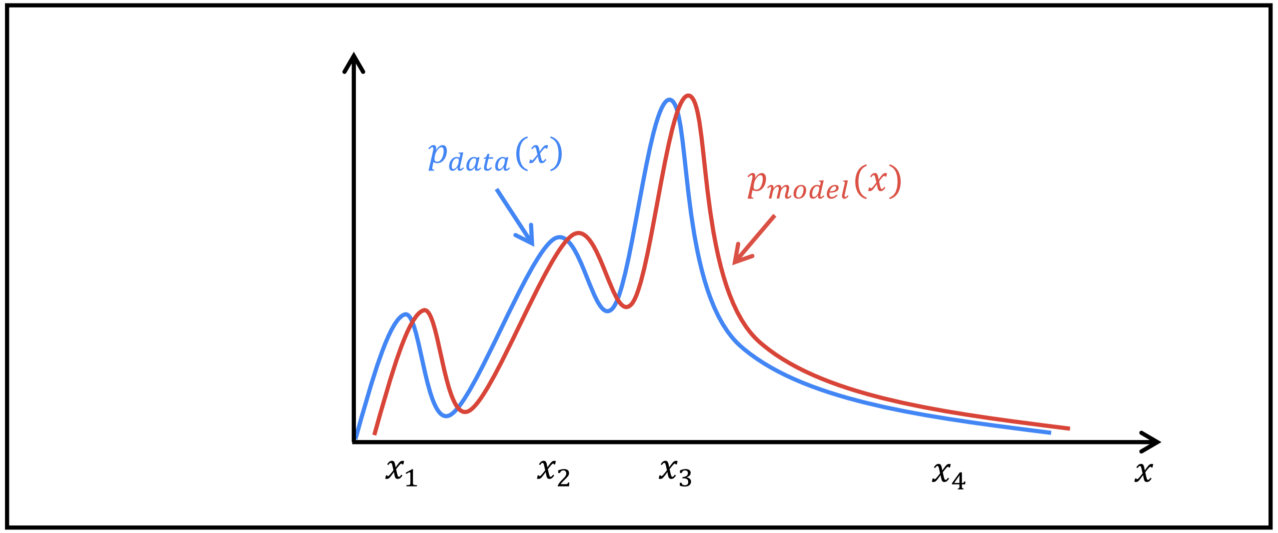
x_4 is an 64x64x3 high dimensional vector representing very strange images.

Probability Distribution

Distribution of images generated by the model

The goal of the generative model is to find a $p_{model}(x)$ that approximates $p_{data}(x)$ well.

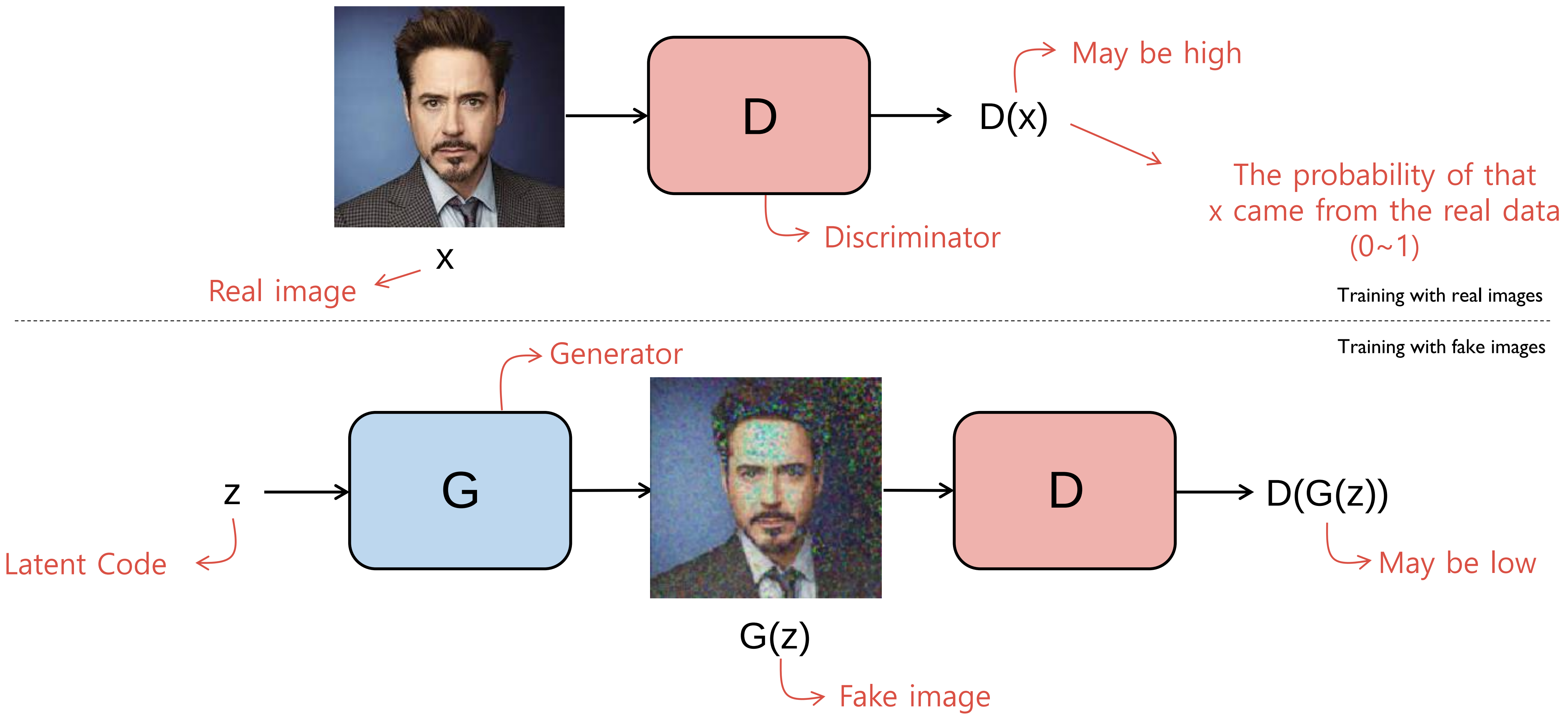
Distribution of actual images



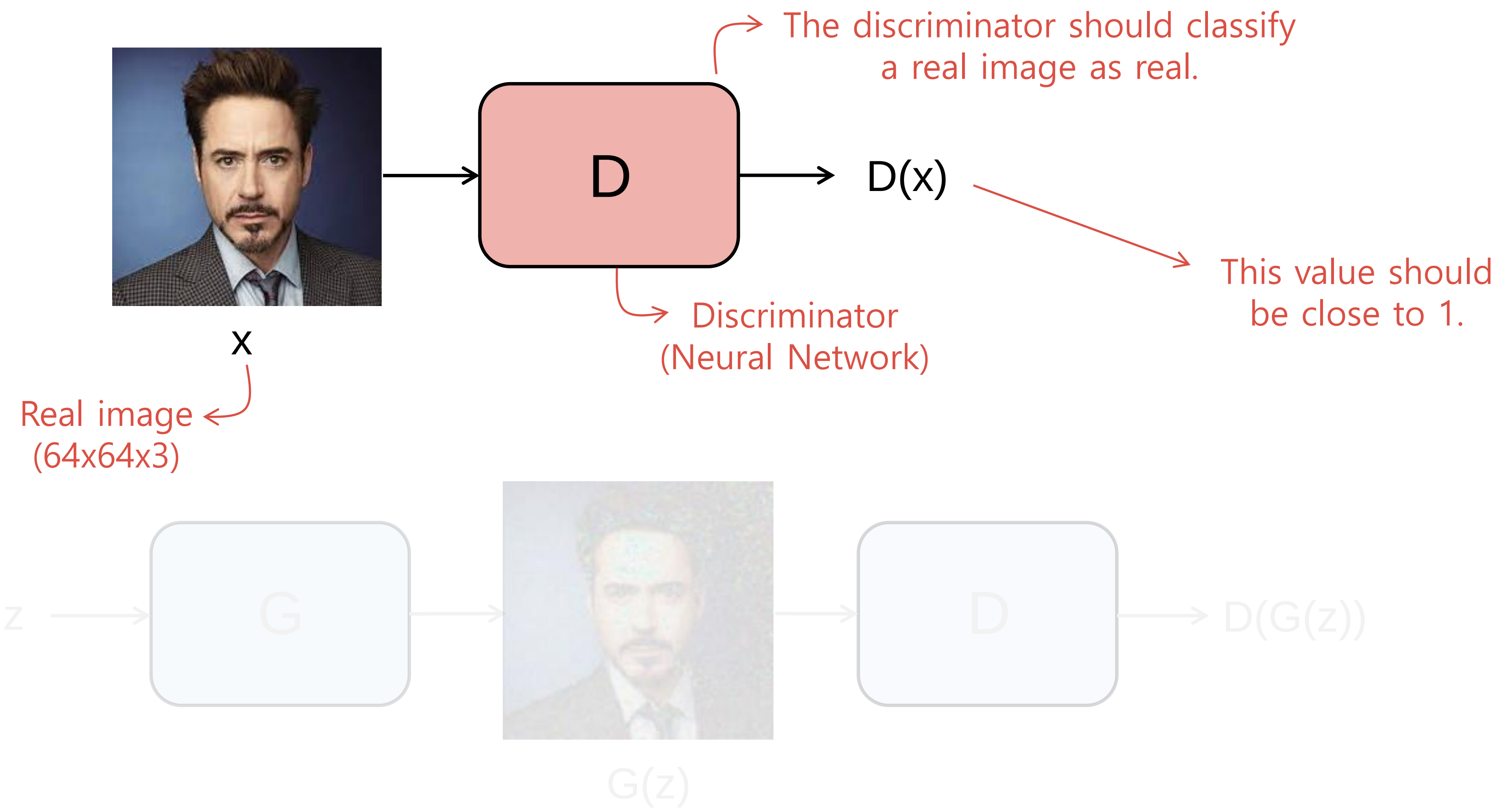
02

Generative Adversarial Networks

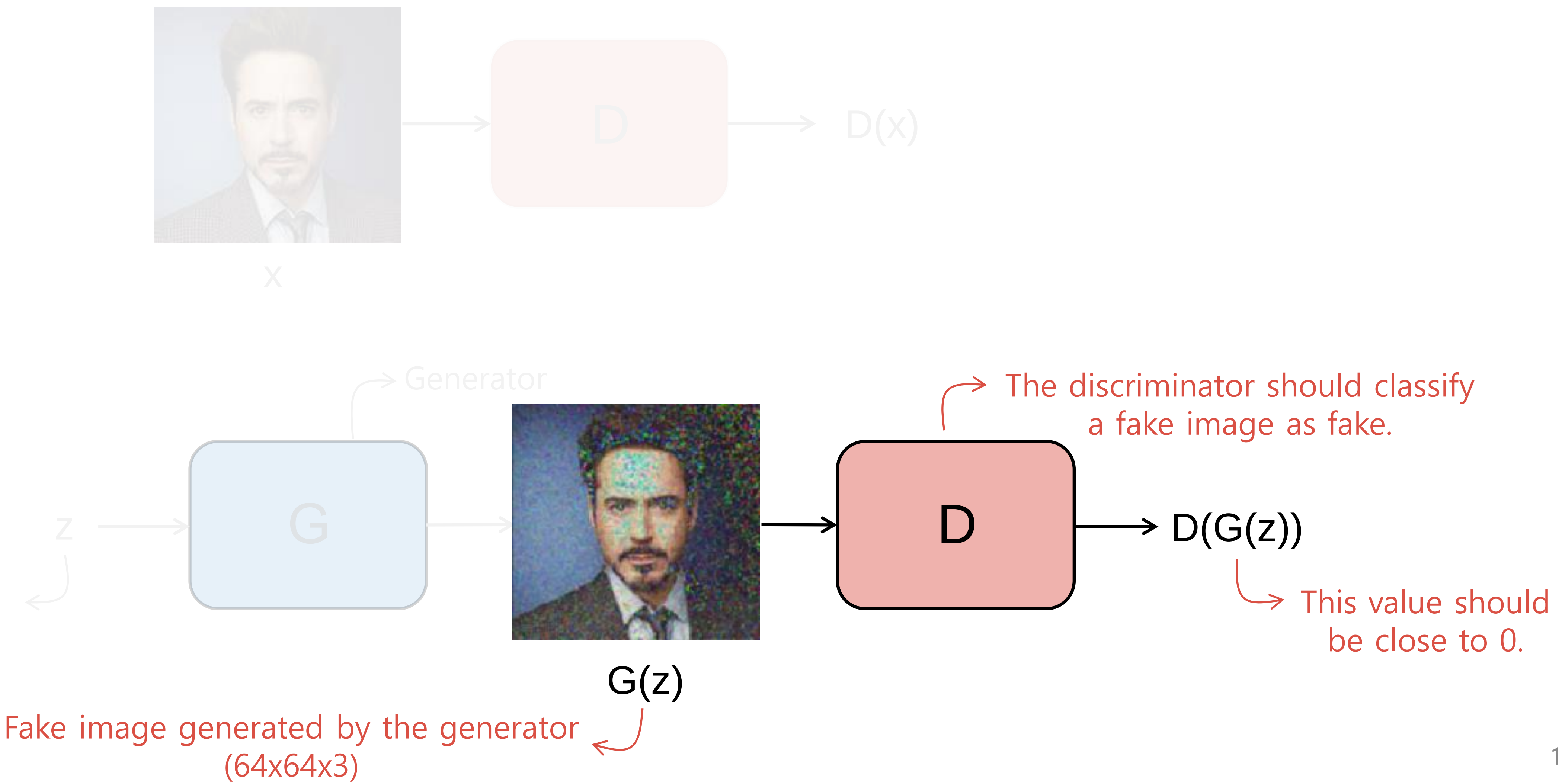
Intuition in GAN



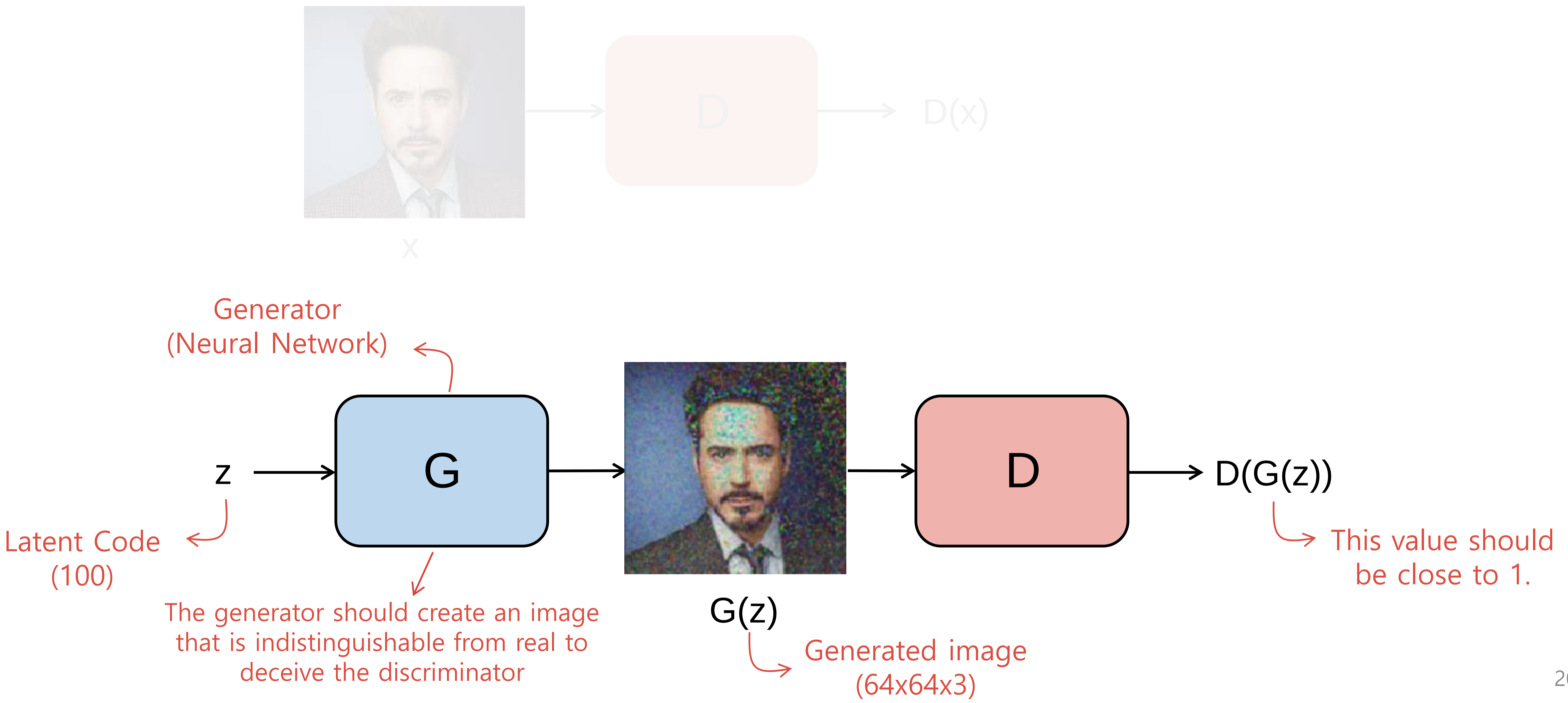
Intuition in GAN



Intuition in GAN



Intuition in GAN



Objective Function of GAN

Sample x from real data distribution

Sample latent code z from Gaussian distribution

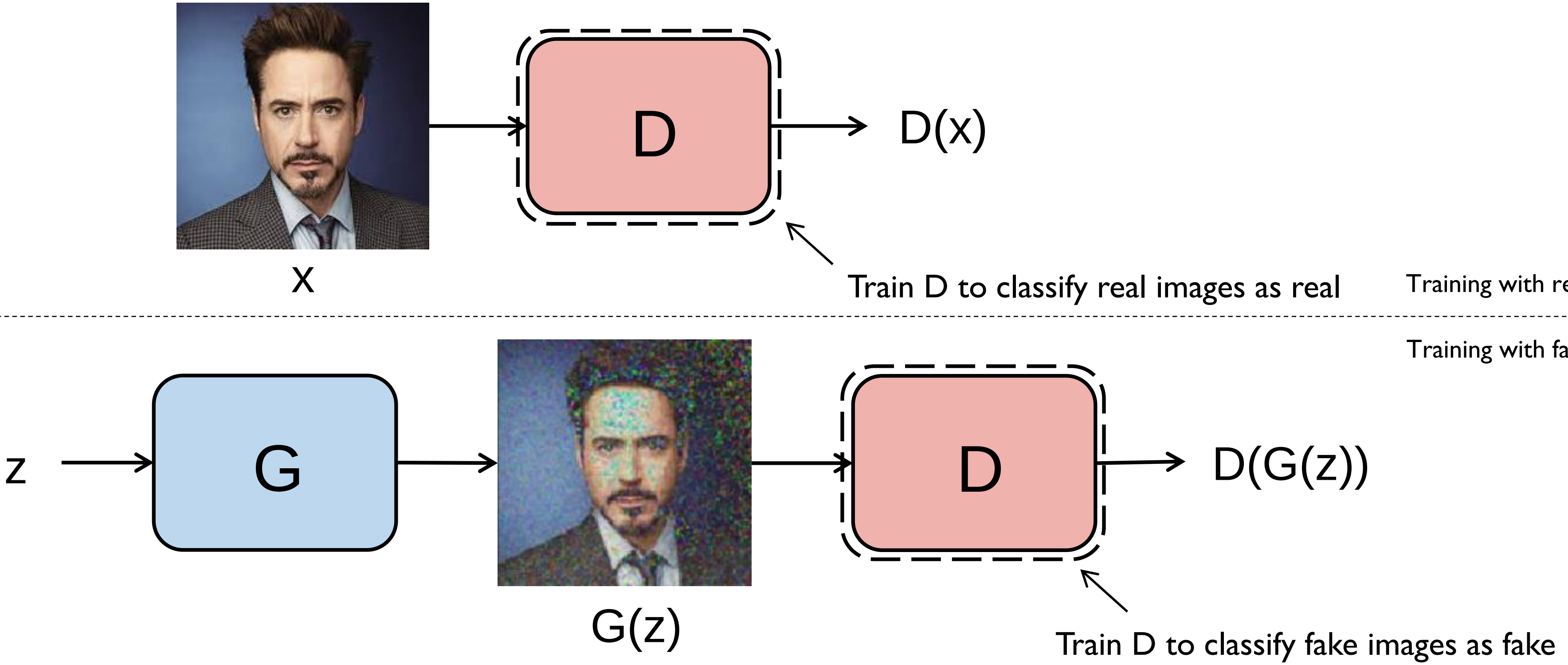
$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

D should maximize $V(D,G)$

Maximum when $D(x) = 1$

Maximum when $D(G(z)) = 0$

Objective function



Training with real images

Training with fake images

Objective Function of GAN

GANs

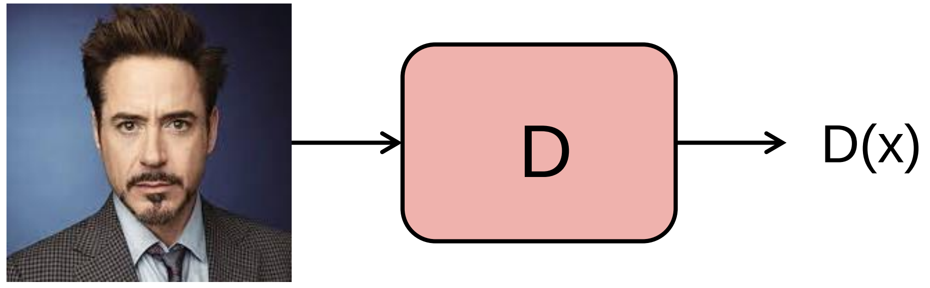
G is independent of this part

$$\min_G \max_D V(D,G) = \cancel{E_{x\sim p_{data}(x)}[\log D(x)]} + E_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

G should minimize $V(D,G)$

Minimum when $D(G(z)) = 1$

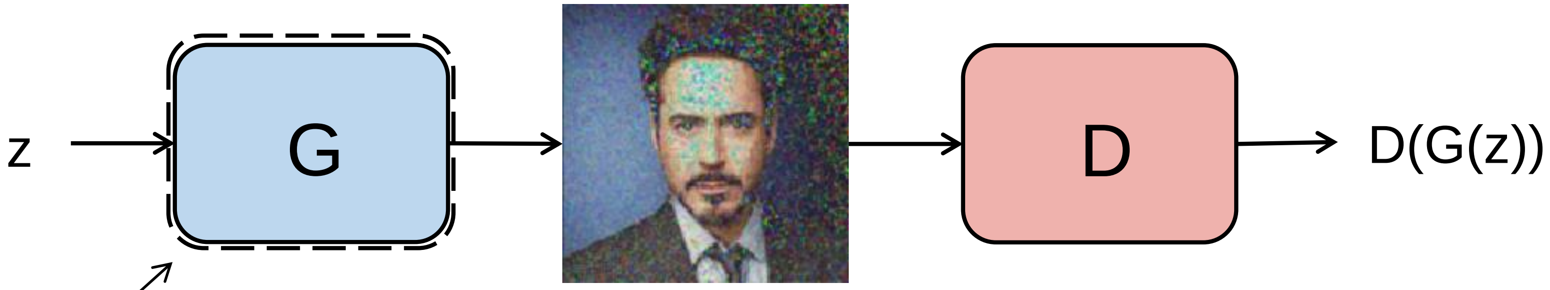
Objective function



x

Training with real images

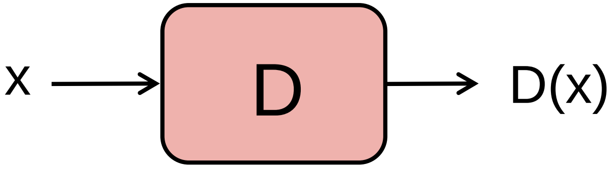
Training with fake images



Train G to deceive D

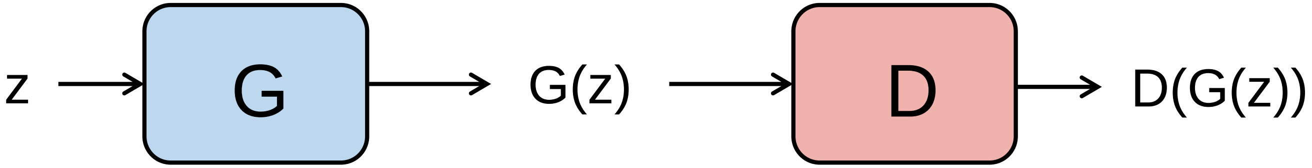
G(z)

PyTorch Implementation



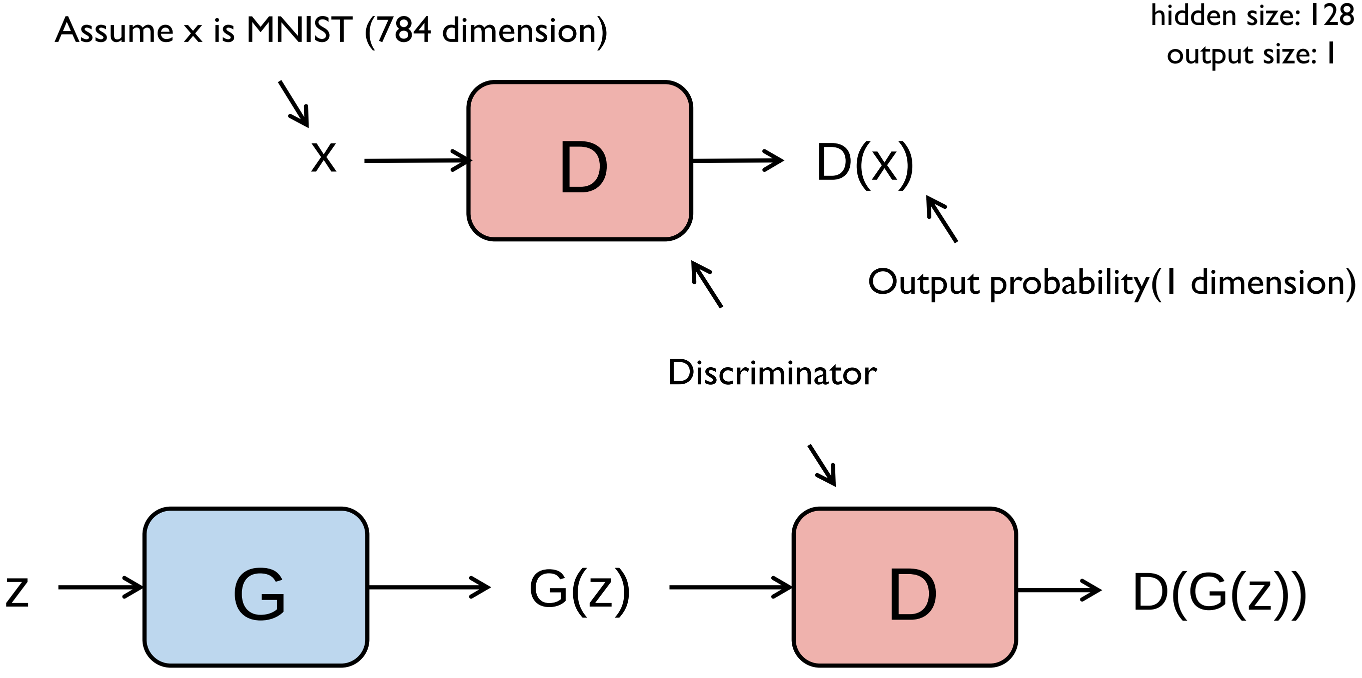
Training with real images

Training with fake images




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

PyTorch Implementation

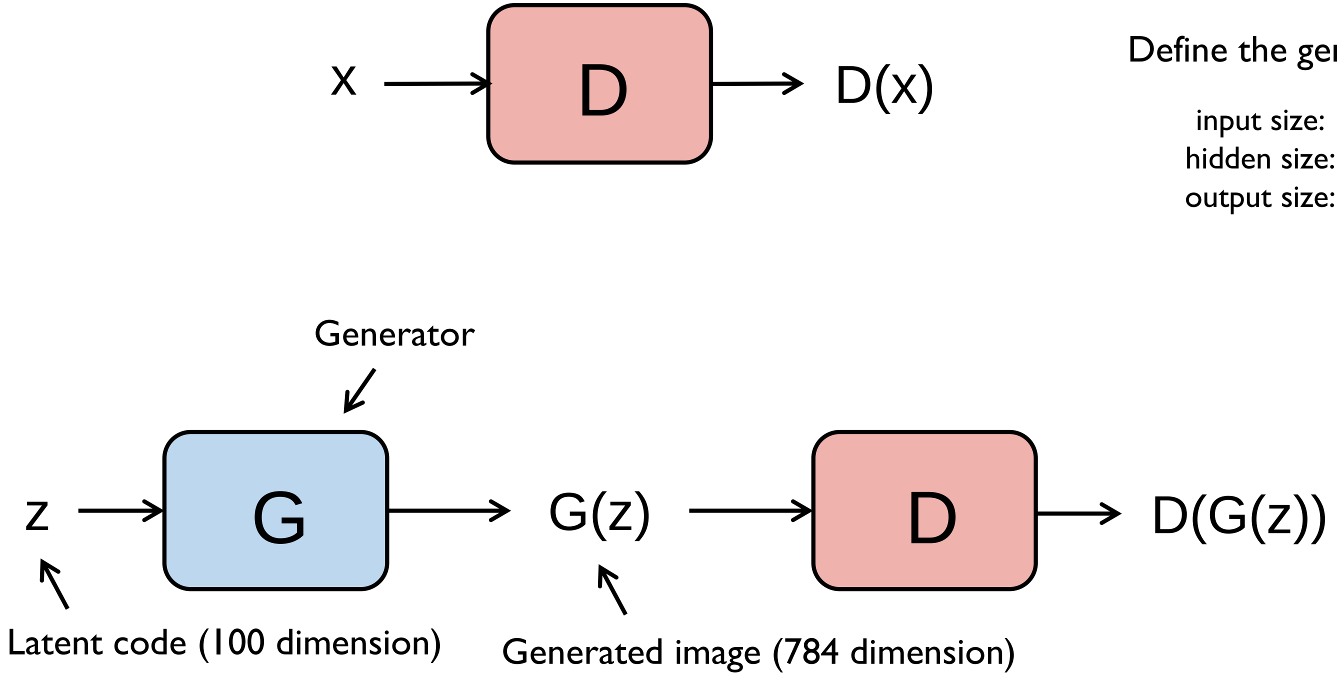


Define the discriminator

input size: 784
hidden size: 128
output size: 1

```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

PyTorch Implementation

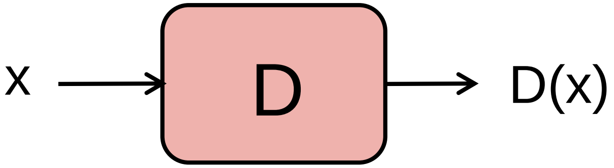


Define the generator

input size: 100
hidden size: 128
output size: 784

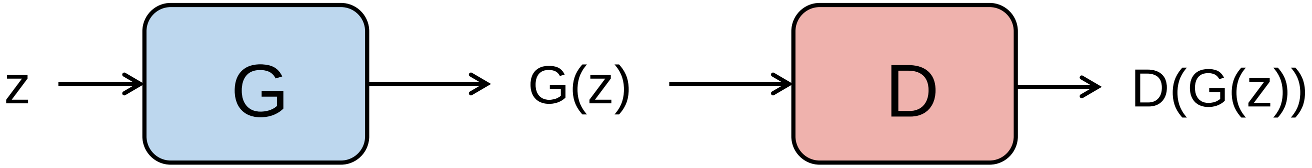
```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

PyTorch Implementation



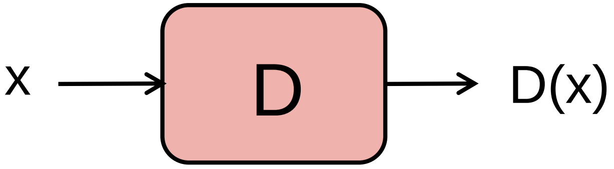
Binary Cross Entropy Loss $(h(x), y)$

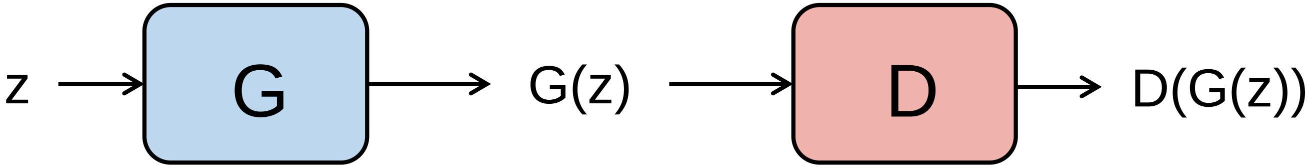
$$-y\log h(x) - (1-y)\log(1-h(x))$$




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

PyTorch Implementation

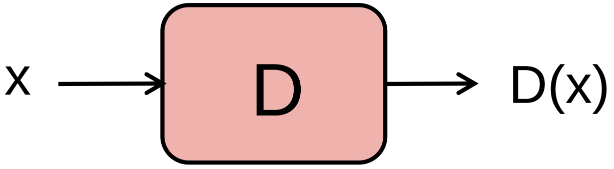




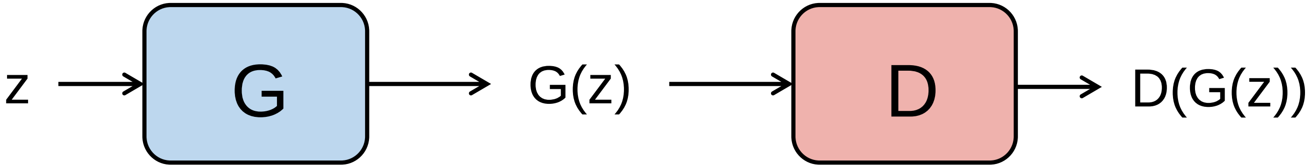
Optimizer for D and G

```
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

PyTorch Implementation



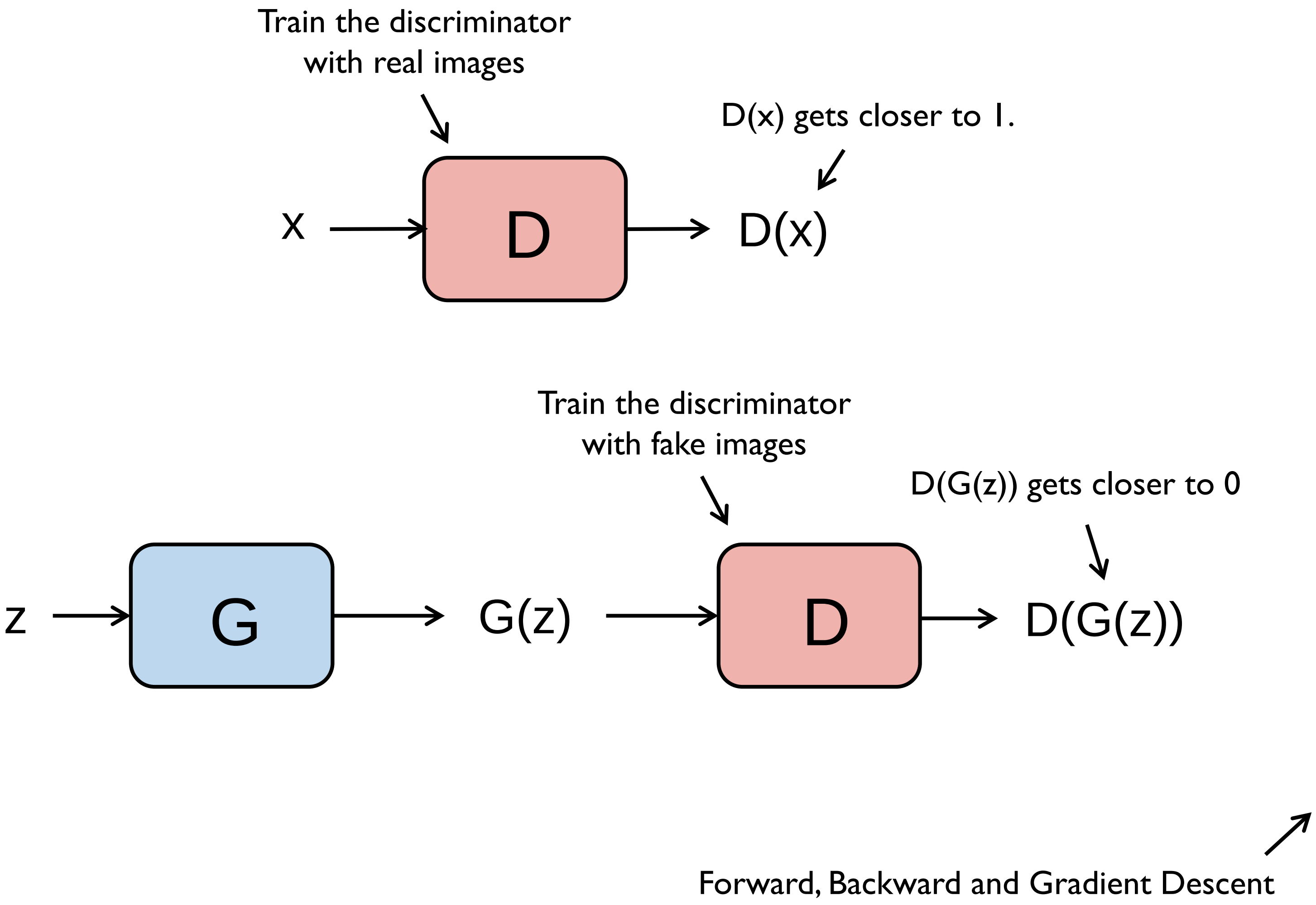
x is a tensor of shape (batch_size, 784).
z is a tensor of shape (batch_size, 100).




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

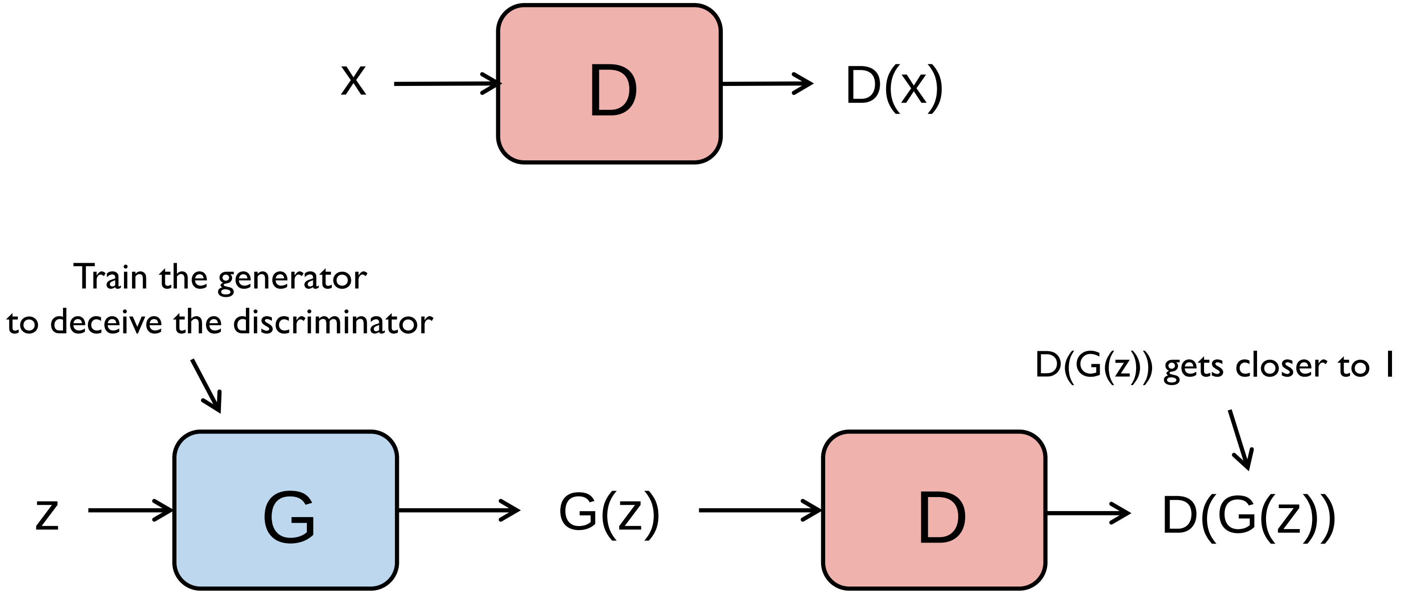
PyTorch Implementation

GANs




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

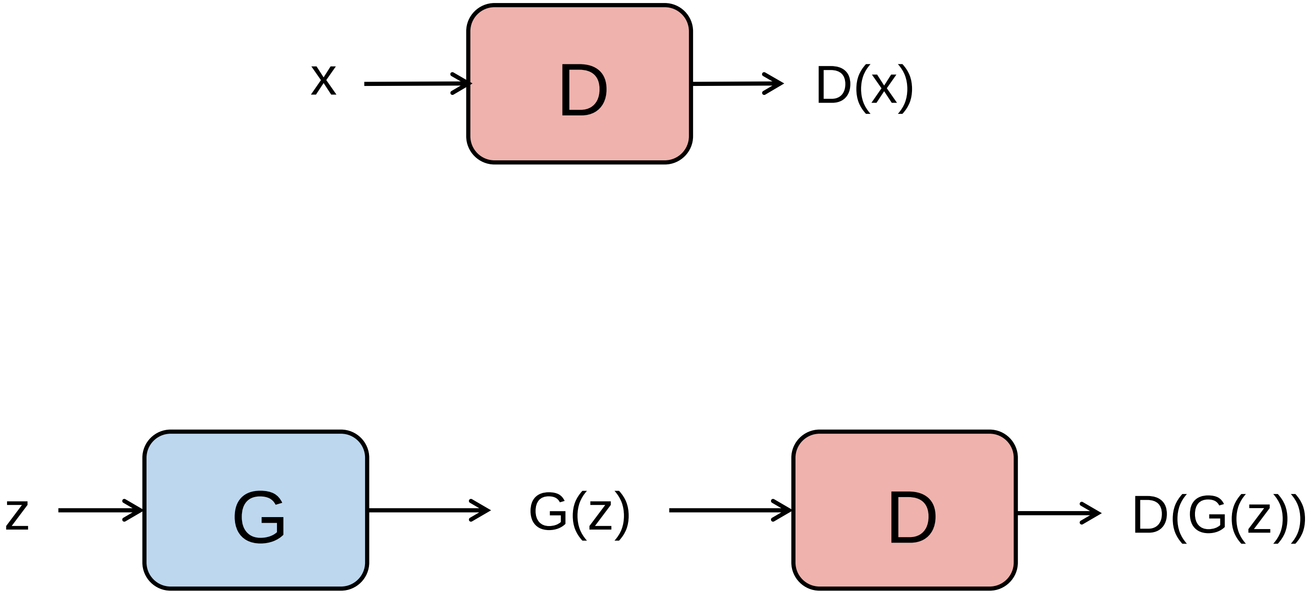
PyTorch Implementation




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

Forward, Backward and Gradient Descent →

PyTorch Implementation




```python
import torch
import torch.nn as nn


D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

criterion = nn.BCELoss()

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)

# Assume x be real images of shape (batch_size, 784)
# Assume z be random noise of shape (batch_size, 100)

while True:
    # train D
    loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
    loss.backward()
    d_optimizer.step()

    # train G
    loss = criterion(D(G(z)), 1)
    loss.backward()
    g_optimizer.step()
```

The complete code can be found here

https://github.com/yunjey/pytorch-tutorial

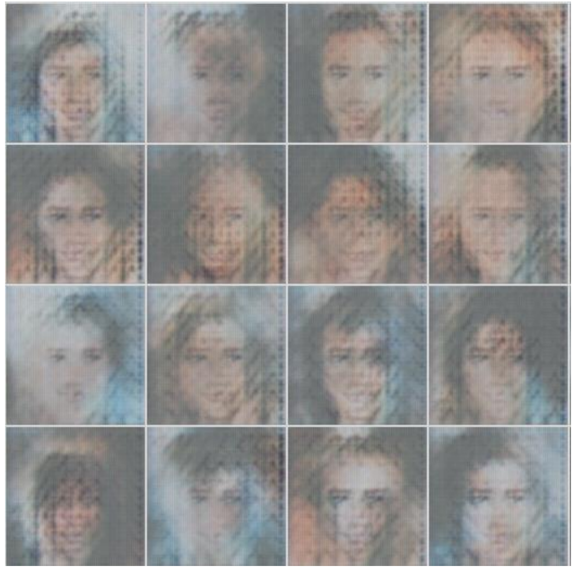
Non-Saturating Game

$$\min_G E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

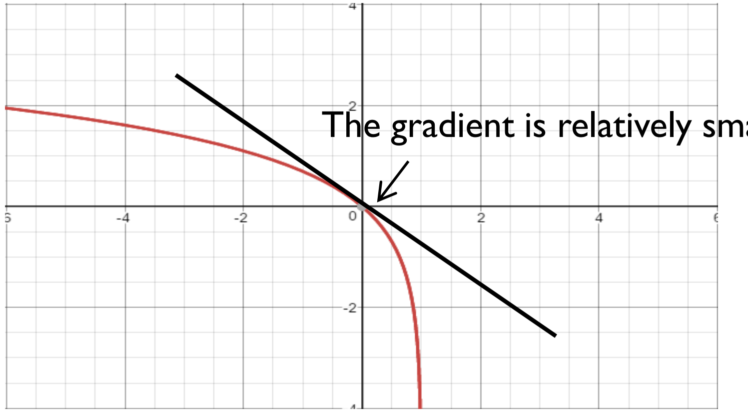
Objective function of G

At the beginning of training, the discriminator can clearly classify the generated image as fake because the quality of the image is very low.

This means that D(G(z)) is almost zero at early stages of training.



Images created by the generator at the beginning of training



The gradient is relatively small at D(G(z))=0

$$y = \log(1 - x)$$

Non-Saturating Game

$$\cancel{\min_G E_{z\sim p_z(z)}[\log(1 - D(G(z))]}$$

Modification (heuristically motivated)

$$\max_G E_{z\sim p_z(z)}[\log D(G(z))]$$

```
# tensorflow
tf.losses.sigmoid_cross_entropy()

# pytorch
nn.BCELoss()
```

- Practical Usage

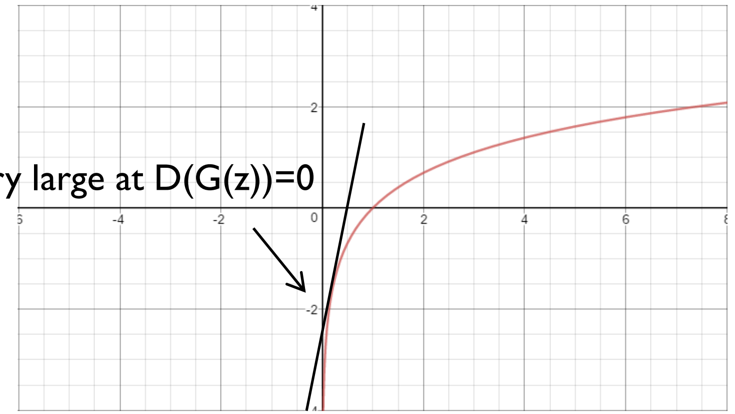
Use binary cross entropy loss function with fake label (1)

$$\min_G E_{z\sim p_z(z)}[-y \log D(G(z)) - (1-y)\log(1 - D(G(z))]$$

$$\Big\downarrow\; y = 1$$

$$\min_G E_{z\sim p_z(z)}[-\log D\big(G(z)\big)]$$

The gradient is very large at D(G(z))=0



$y = \log(x)$

Theory in GAN

GANs

- **Why does GANs work?**

Because it actually minimizes the distance between the real data distribution and the model distribution.

$$\min_G \max_D V(D,G) \xrightarrow{\text{same}} \min_{G,D} JSD(p_{data}||p_g)$$

Objective function of GANs

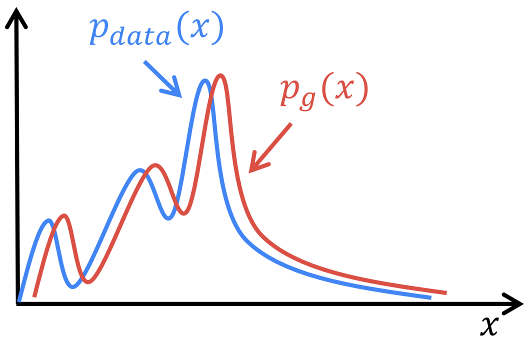
$$E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Jenson-Shannon divergence

$$JSD(P||Q) = \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M)$$

$$where\ M = \frac{1}{2}(P+Q)$$

KL Divergence



Please see Appendix for details

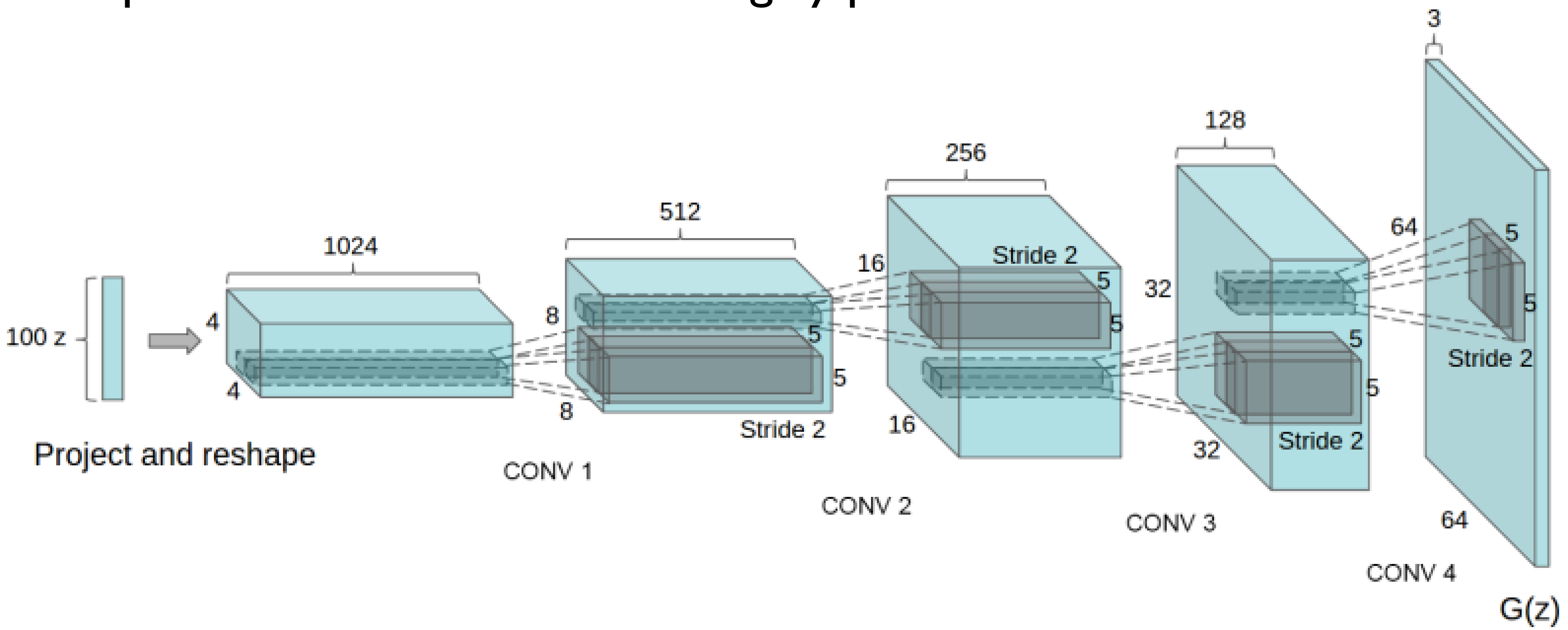
03

Variants of GAN

DCGAN

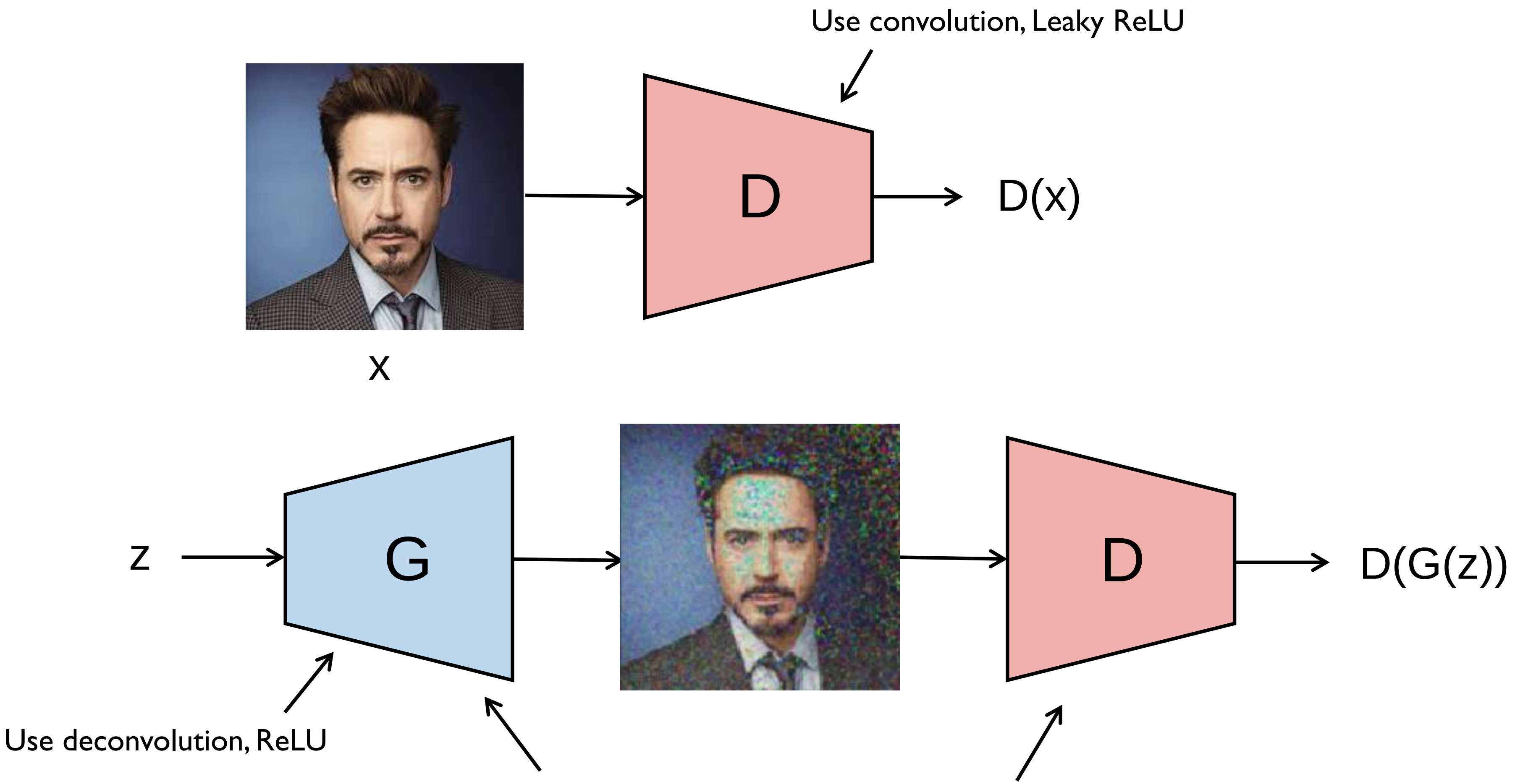
- Deep Convolutional GAN(DCGAN), 2015

The authors present a model that is still highly preferred.



Radford et al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015

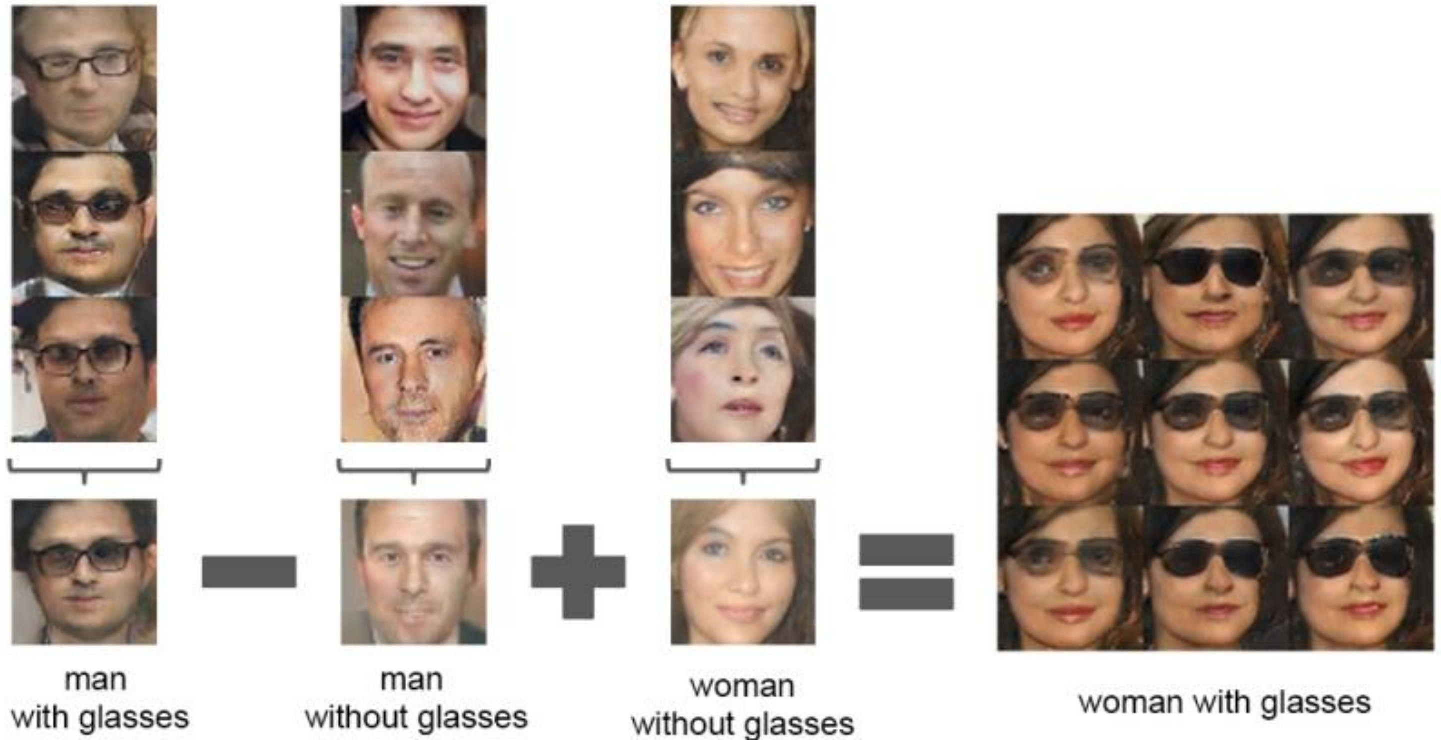
DCGAN



- No pooling layer (Instead strided convolution)
- Use batch normalization
- Adam optimizer(lr=0.0002, beta1=0.5, beta2=0.999)

DCGAN

- Latent vector arithmetic

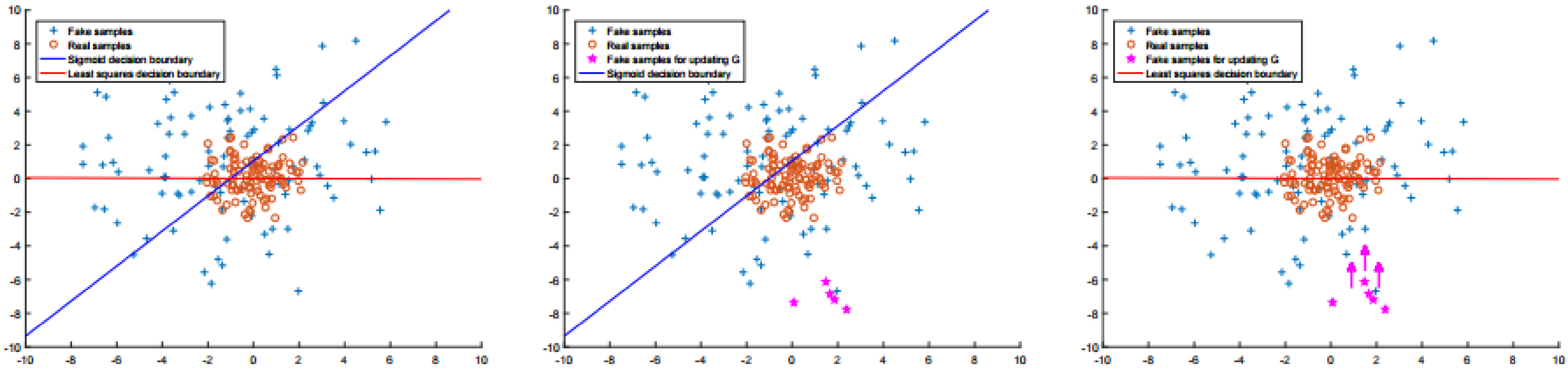


Radford et al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015

LSGAN

- Least Squares GAN (LSGAN)

Proposed a GAN model that adopts the least squares loss function for the discriminator.



Xudong Mao et al. Least Squares Generative Adversarial Networks, 2016

LSGAN

Vanilla GAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = - torch.mean(torch.log(D(x))) - torch.mean(torch.log(1 - D(G(z))))

# Loss of G
G_loss = - torch.mean(torch.log(D(G(z))))
```

Remove sigmoid non-linearity in last layer →

LSGAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1))


G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = torch.mean((D(x) - 1)**2) + torch.mean(D(G(z))**2)

# Loss of G
G_loss = torch.mean((D(G(z)) - 1)**2)
```

LSGAN

Vanilla GAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = - torch.mean(torch.log(D(x))) - torch.mean(torch.log(1 - D(G(z))))

# Loss of G
G_loss = - torch.mean(torch.log(D(G(z))))
```

Generator is the same as original

LSGAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1))


G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = torch.mean((D(x) - 1)**2) + torch.mean(D(G(z))**2)

# Loss of G
G_loss = torch.mean((D(G(z)) - 1)**2)
```

LSGAN

Vanilla GAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = - torch.mean(torch.log(D(x))) - torch.mean(torch.log(1 - D(G(z))))

# Loss of G
G_loss = - torch.mean(torch.log(D(G(z))))
```

Replace cross entropy loss to least squares loss (L2)

LSGAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1))


G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = torch.mean((D(x) - 1)**2) + torch.mean(D(G(z))**2)

# Loss of G
G_loss = torch.mean((D(G(z)) - 1)**2)
```

D(x) gets closer to 1
D(G(z)) gets closer to 0
(same as original)

LSGAN

Vanilla GAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1),
    nn.Sigmoid())

G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = - torch.mean(torch.log(D(x))) - torch.mean(torch.log(1 - D(G(z))))

# Loss of G
G_loss = - torch.mean(torch.log(D(G(z))))
```

Replace cross entropy loss to least squares loss (L2)

LSGAN

```python
D = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 1))


G = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Linear(128, 784),
    nn.Tanh())

# Loss of D
D_loss = torch.mean((D(x) - 1)**2) + torch.mean(D(G(z))**2)

# Loss of G
G_loss = torch.mean((D(G(z)) - 1)**2)
```

D(G(z)) gets closer to 1 (same as original)

LSGAN

- Results (LSUN dataset)

(a) Church outdoor.

(b) Dining room.

(c) Kitchen.

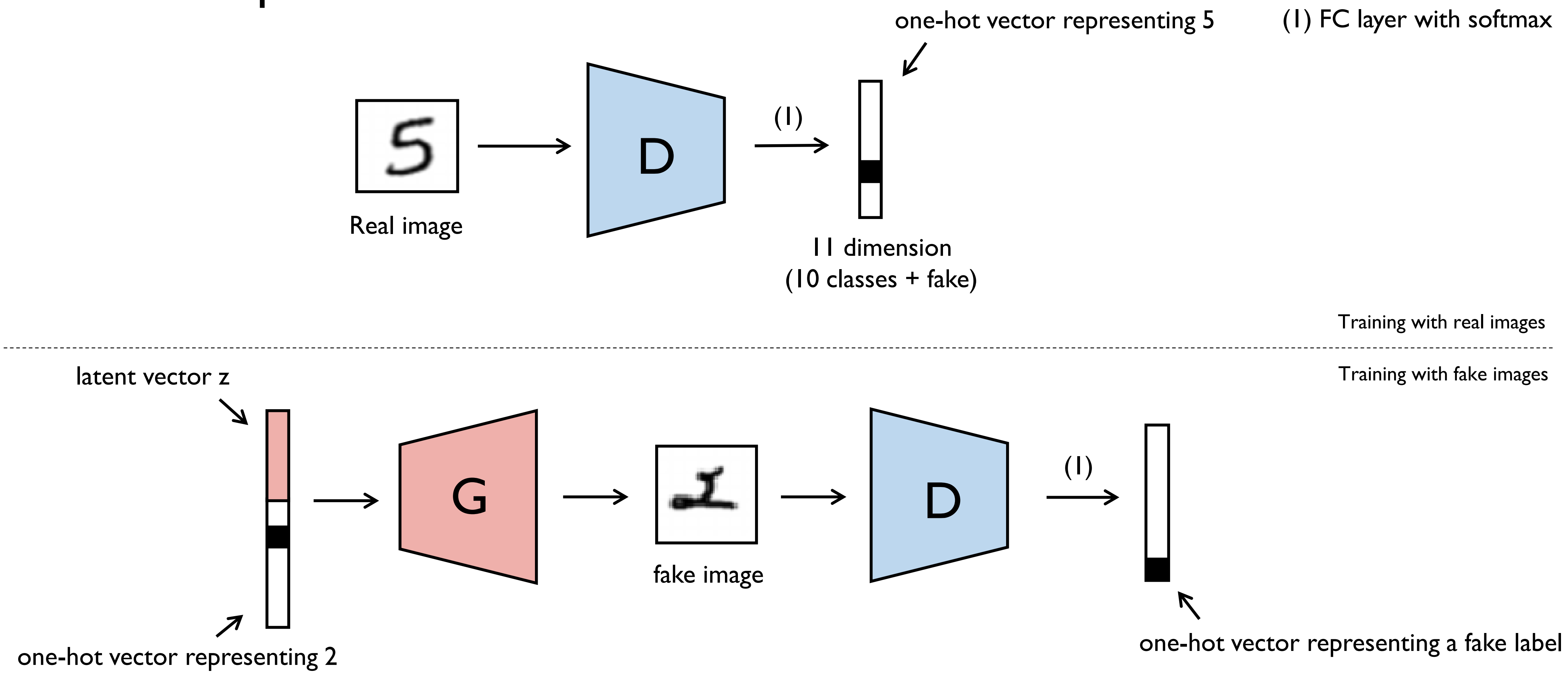
Xudong Mao et al. Least Squares Generative Adversarial Networks, 2016

LSGAN

- Results (CelebA)

SGAN

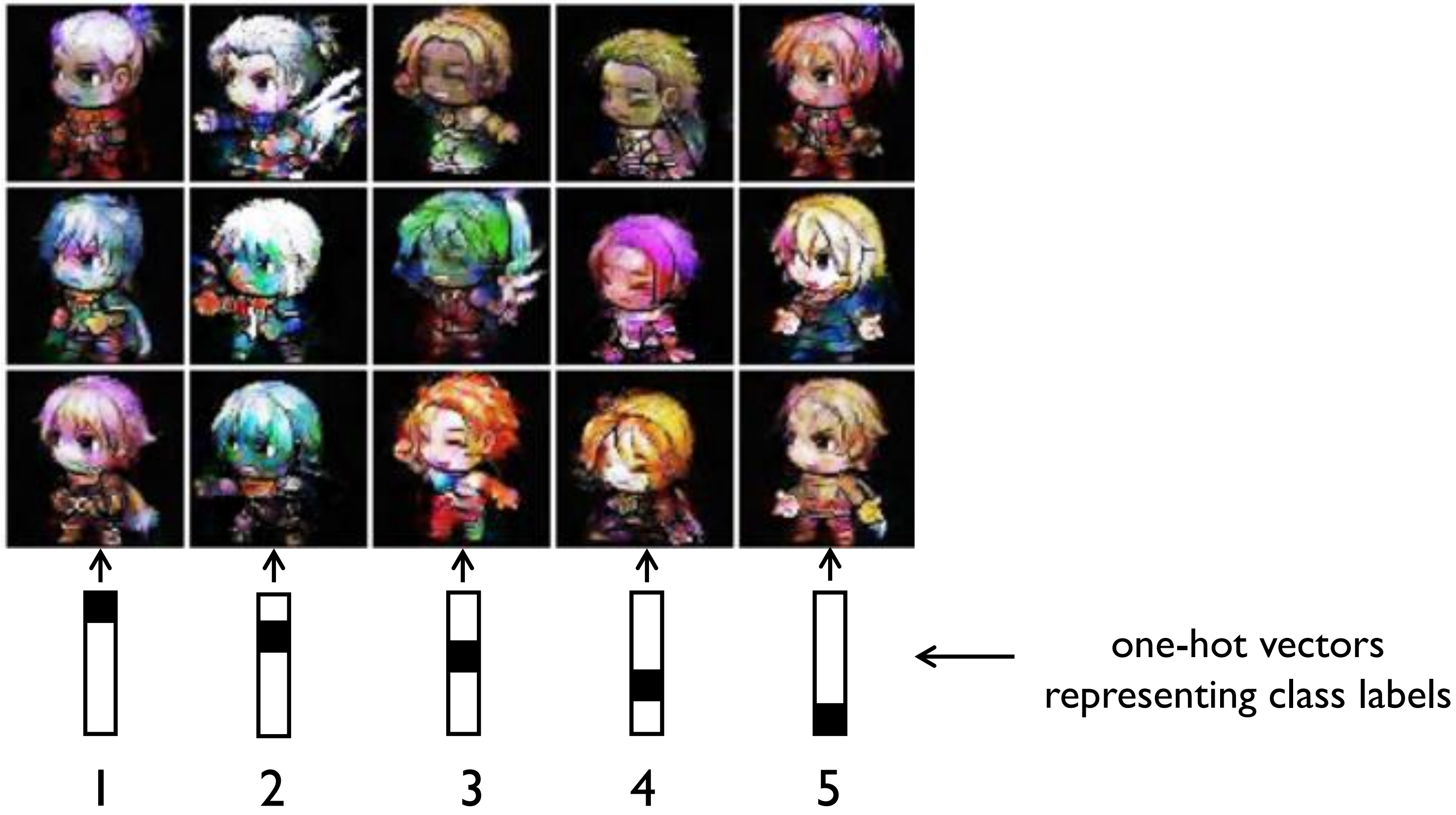
- Semi-Supervised GAN



Augustus Odena et al. Semi-Supervised Learning with Generative Adversarial Netwoks, 2016

- Results (Game Character)

The generator can create an character image that takes a certain pose.

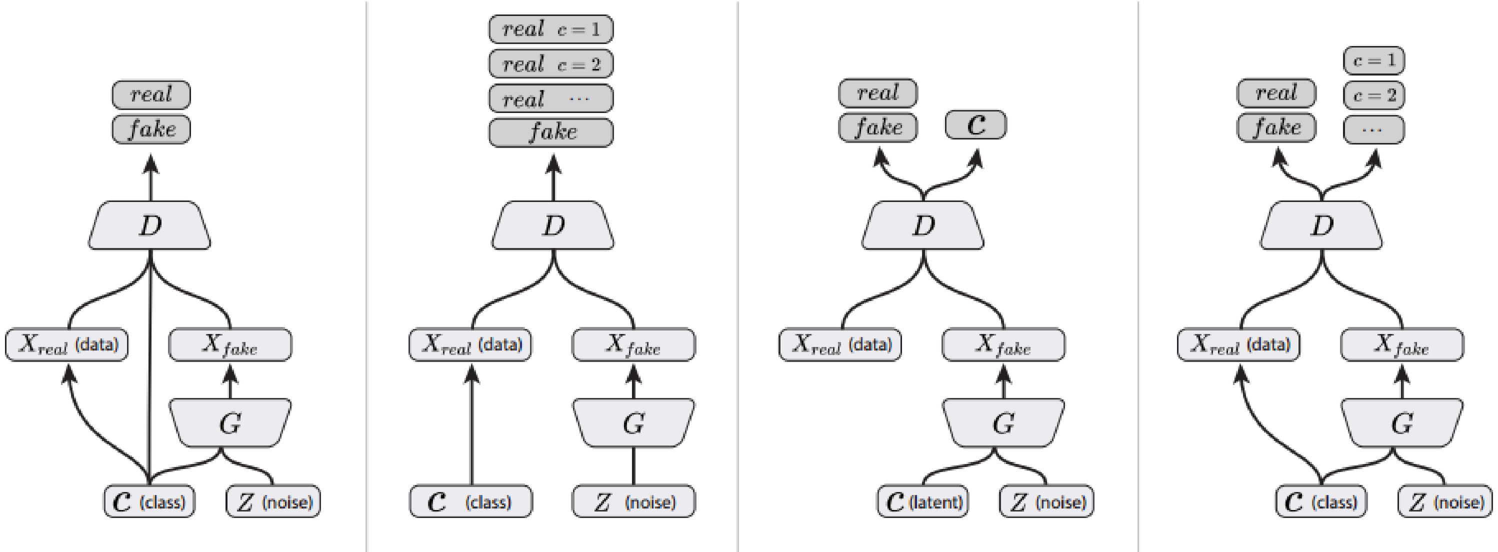


The code will be available soon: https://github.com/yunjey

ACGAN

- Auxiliary Classifier GAN(ACGAN), 2016

  Proposed a new method for improved training of GANs using class labels.



Conditional GAN
(Mirza & Osindero, 2014)

Semi-Supervised GAN
(Odena, 2016; Salimans, et al., 2016)

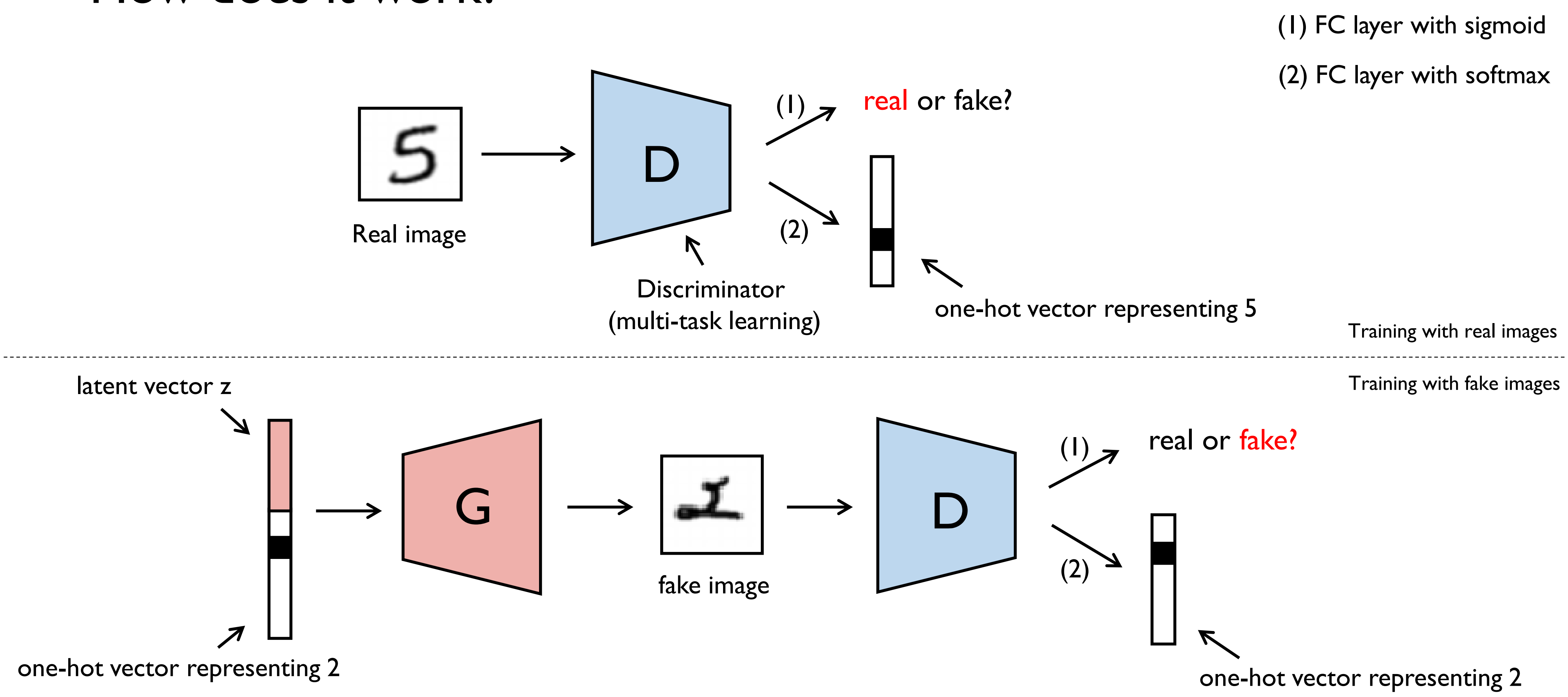
InfoGAN
(Chen, et al., 2016)

AC-GAN
(Present Work)

Augustus Odena et al. Conditional Image Synthesis with Auxilary Classifier, 2016

ACGAN

- How does it work?

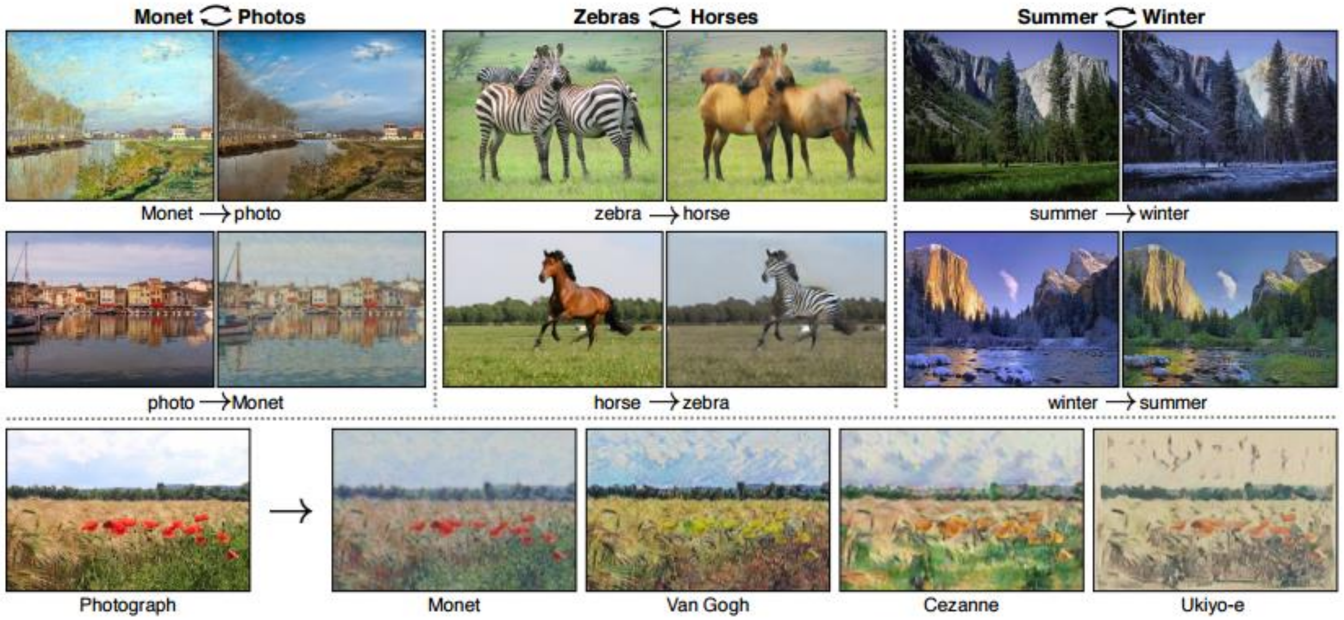


04

Extensions of GAN

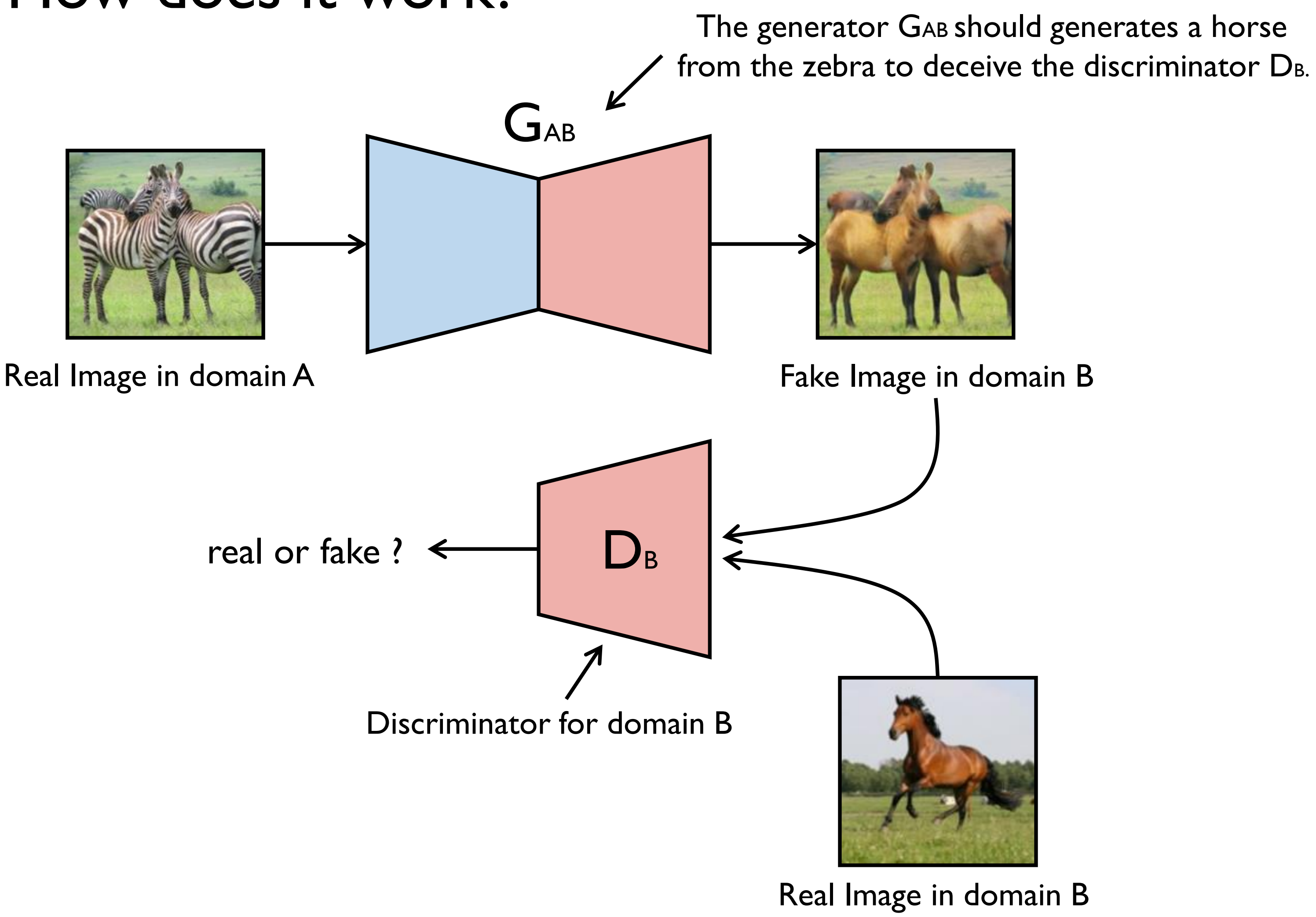
- CycleGAN: Unpaired Image-to-Image Translation

  presents a GAN model that transfer an image from a source domain A to a target domain B in the absence of paired examples.



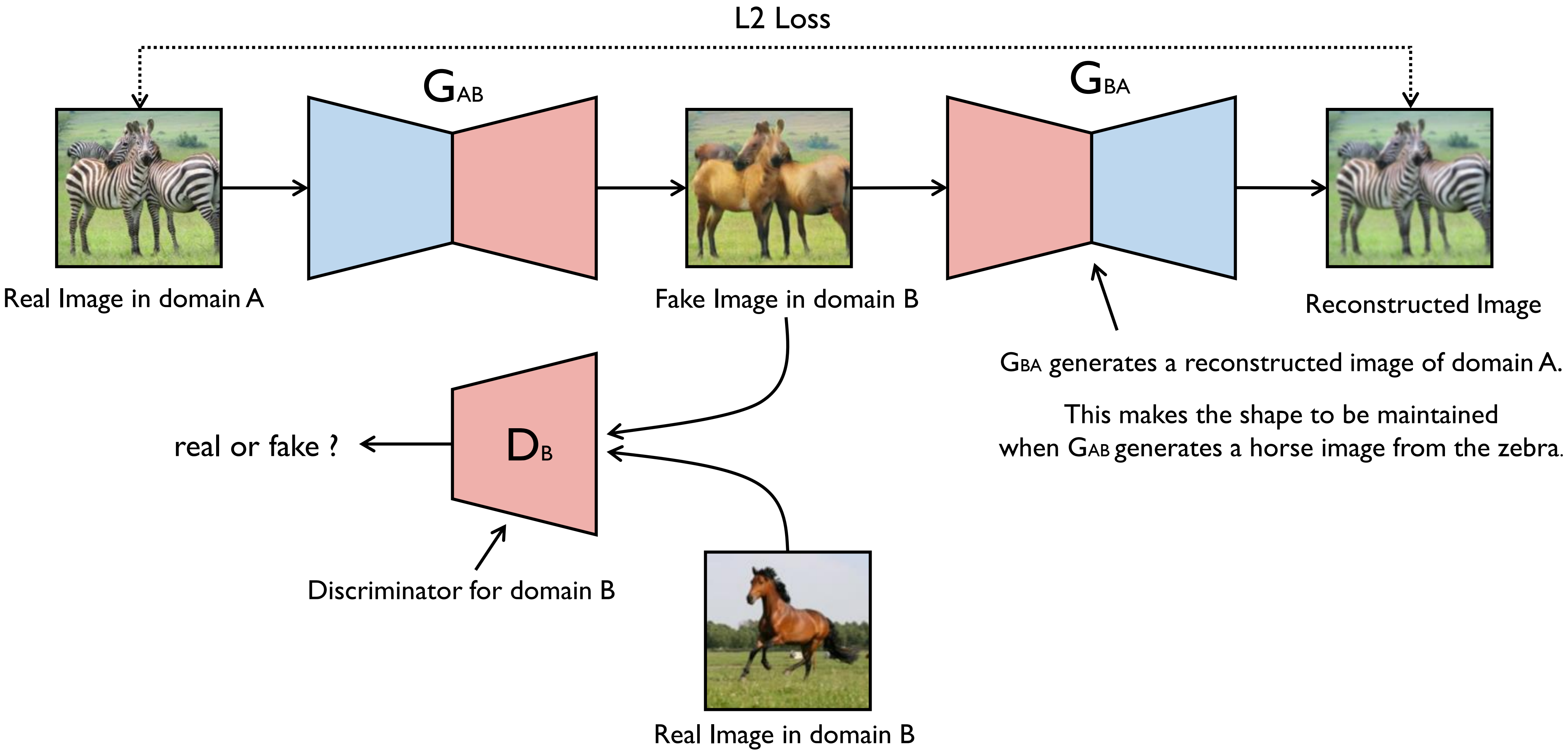
Jun-Yan Zhu et al. Unpaired Image-to-Image Translation using Cycle Consistent Adversarial Networks, 2017

- How does it work?



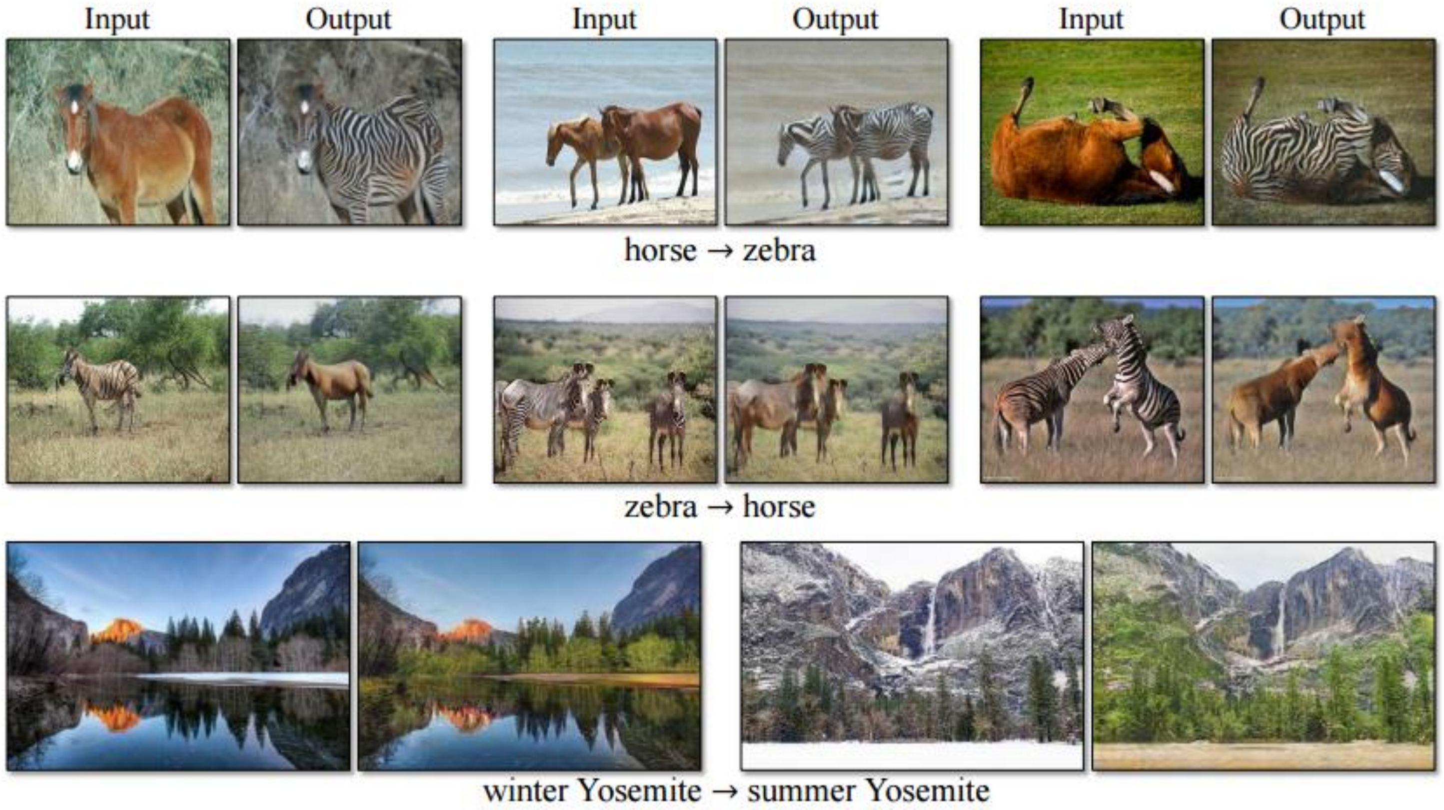
CycleGAN

- How does it work?



CycleGAN

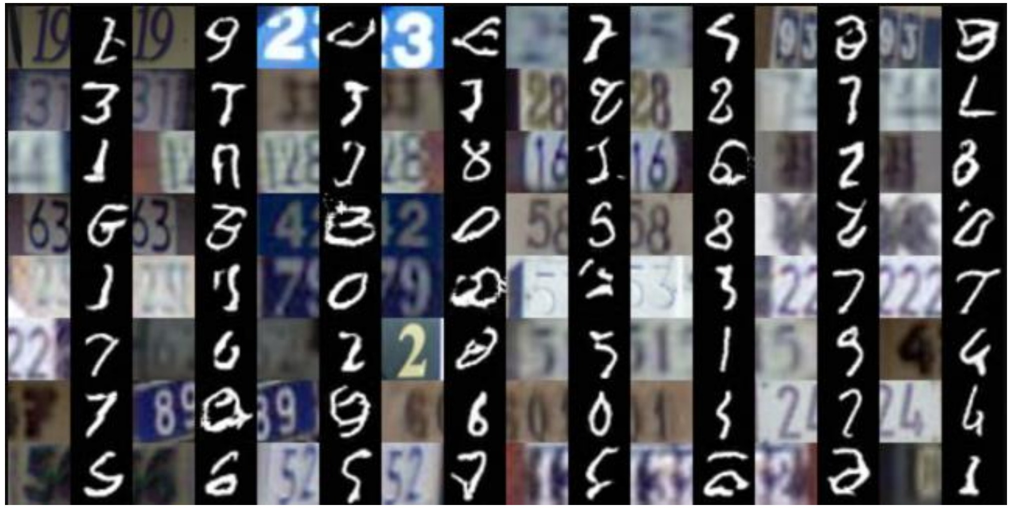
- Results



Jun-Yan Zhu et al. Unpaired Image-to-Image Translation using Cycle Consistent Adversarial Networks, 2017

- Results

Odd columns contain real images and even columns contain generated images.

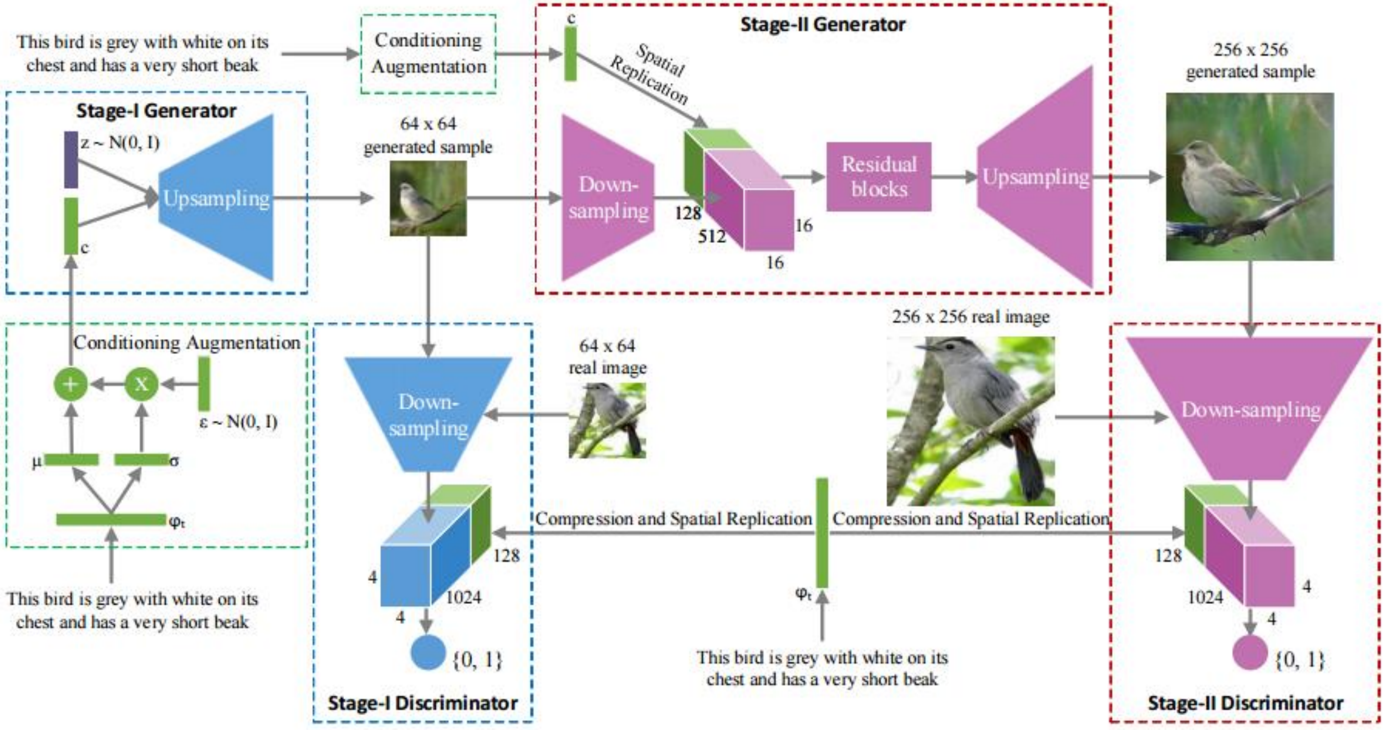


SVHN-to-MNIST

MNIST-to-SVHN

https://github.com/yunjey/mnist-svhn-transfer

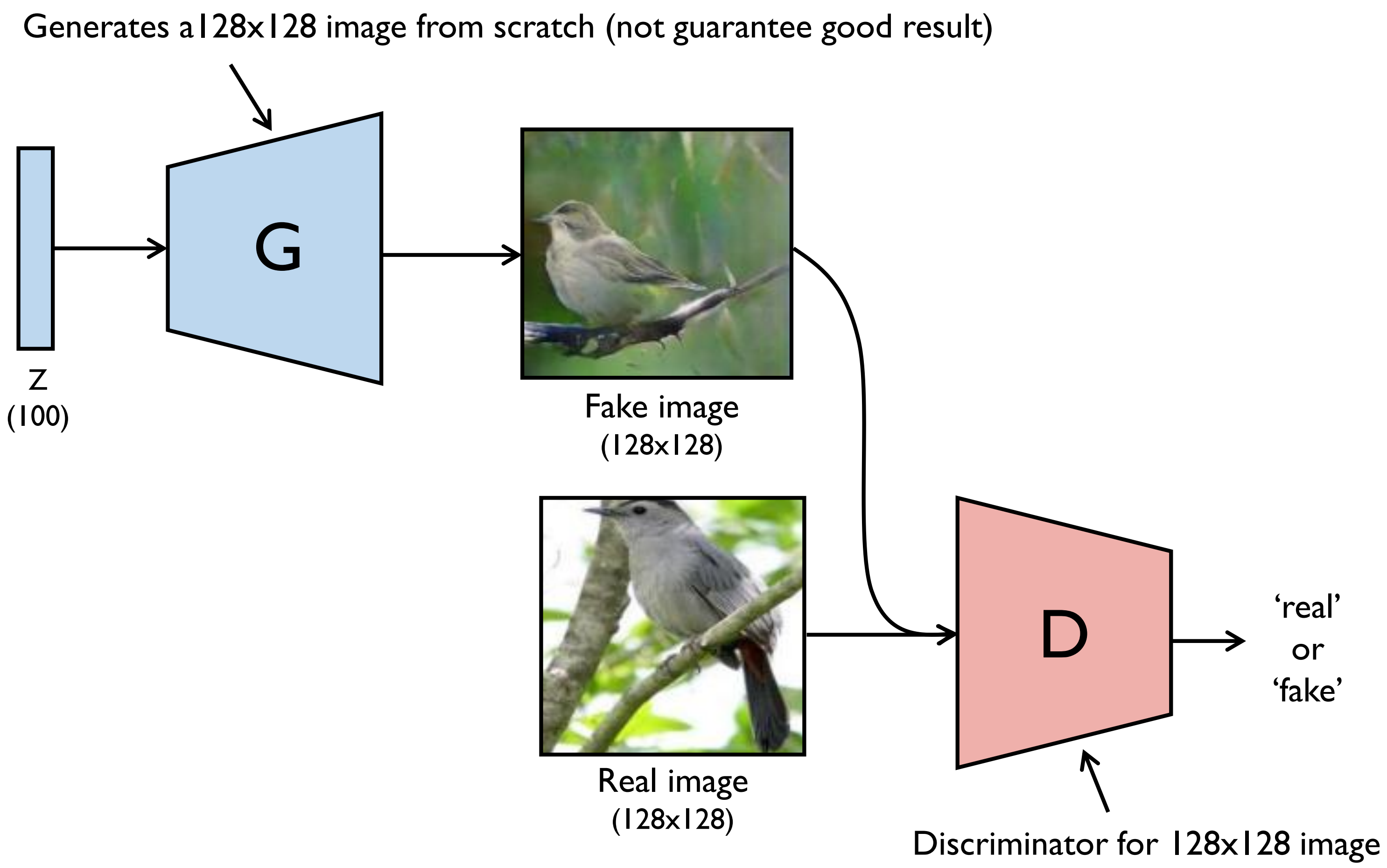
- StackGAN: Text to Photo-realistic Image Synthesis



Han Zhang et al. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks, 2016

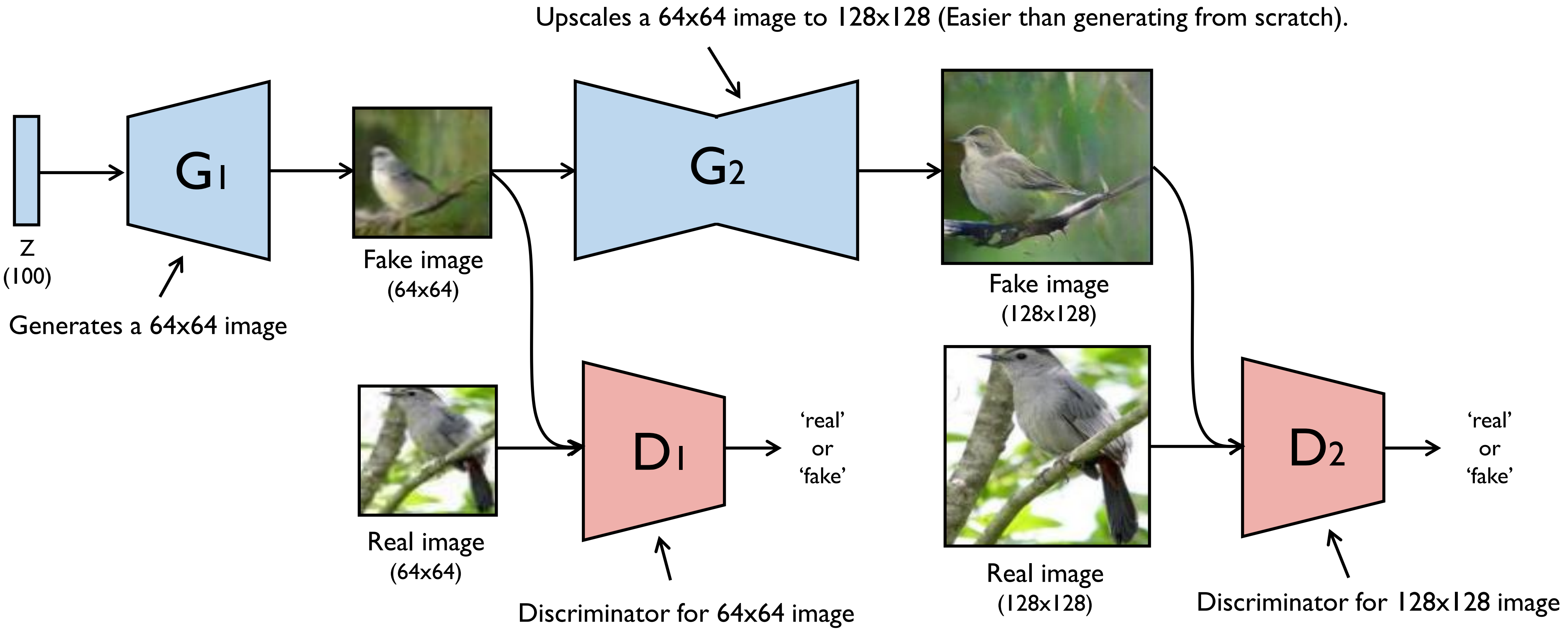
StackGAN

- Generating 128x128 from scratch

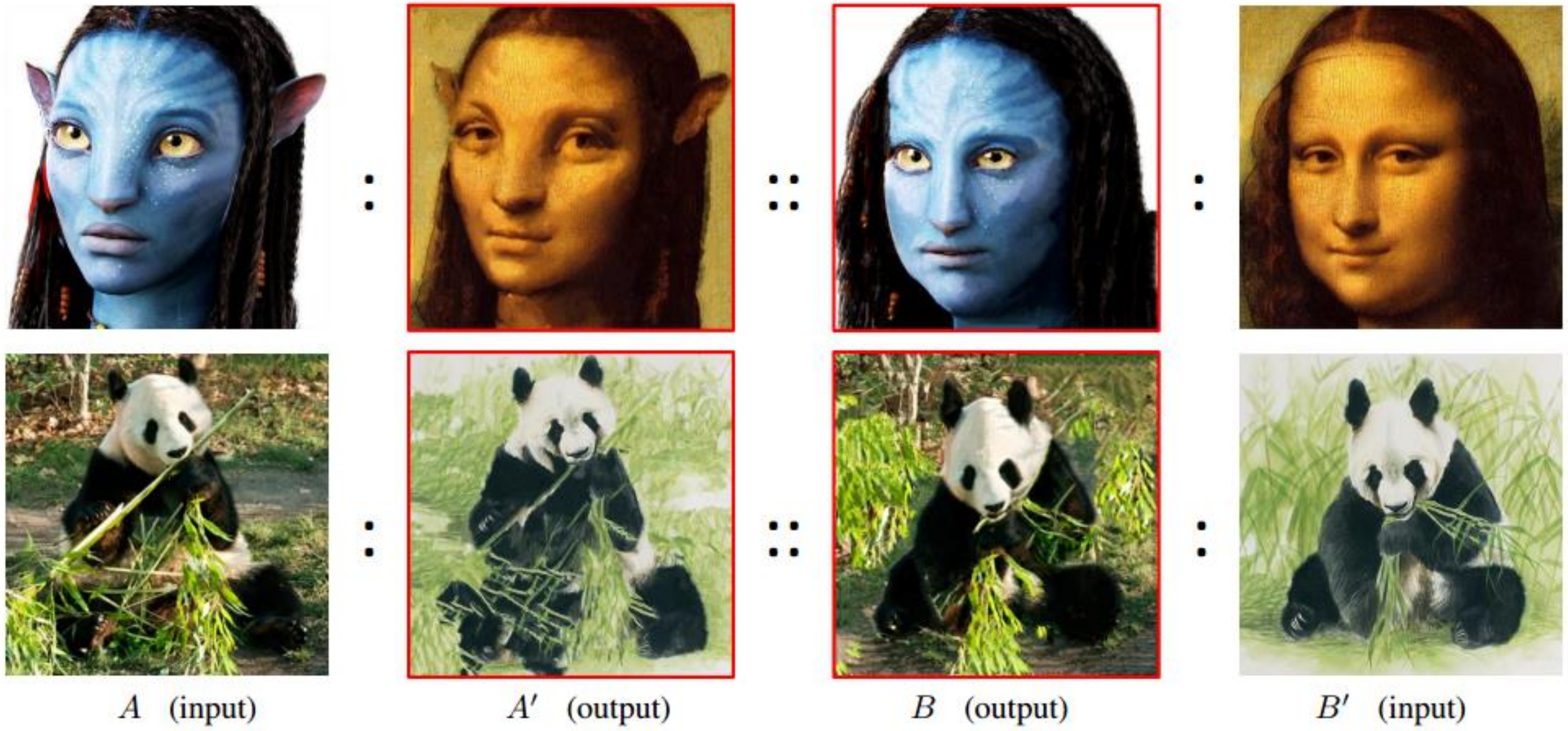


StackGAN

- Generating 128x128 from 64x64

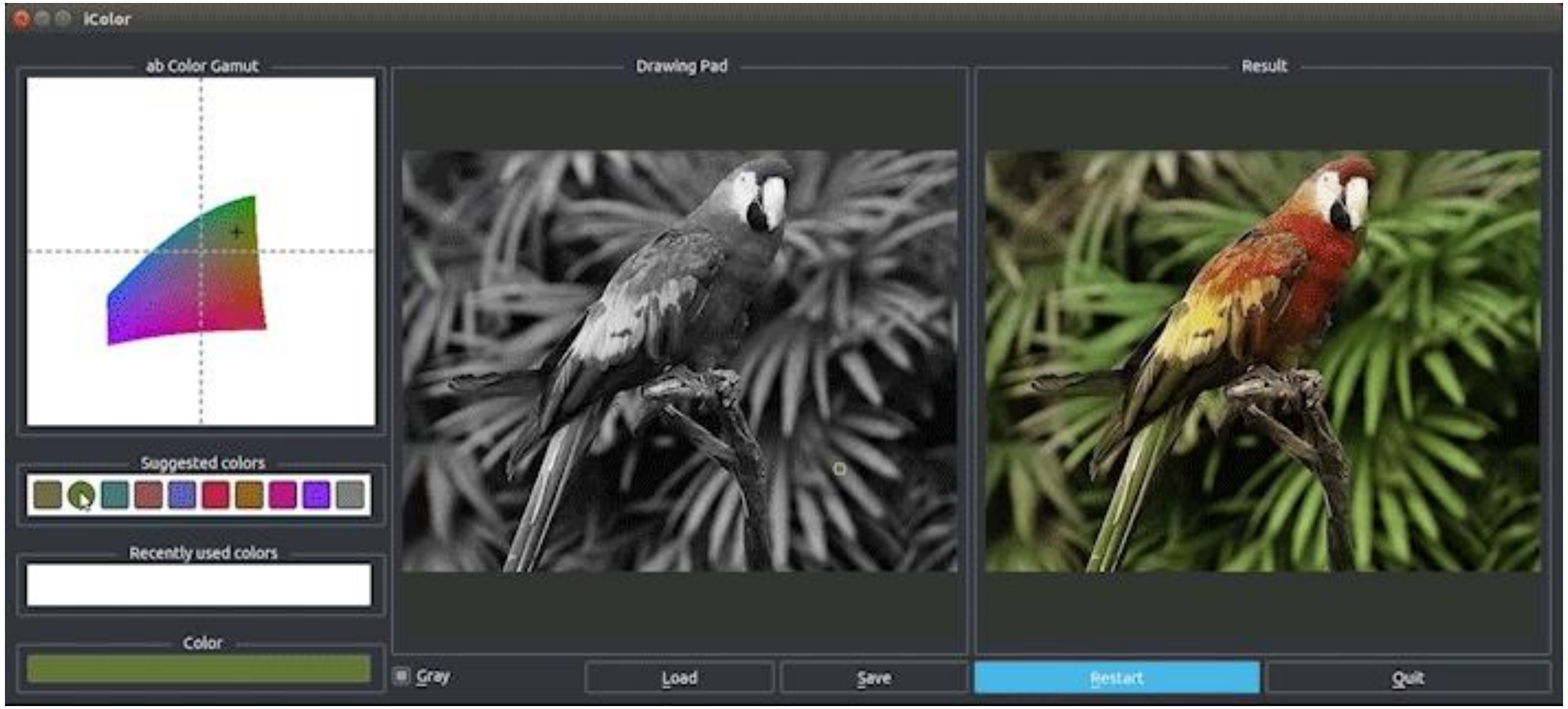


- Visual Attribute Transfer



Jing Liao et al. Visual Attribute Transfer through Deep Image Analogy, 2017

- User-Interactive Image Colorization



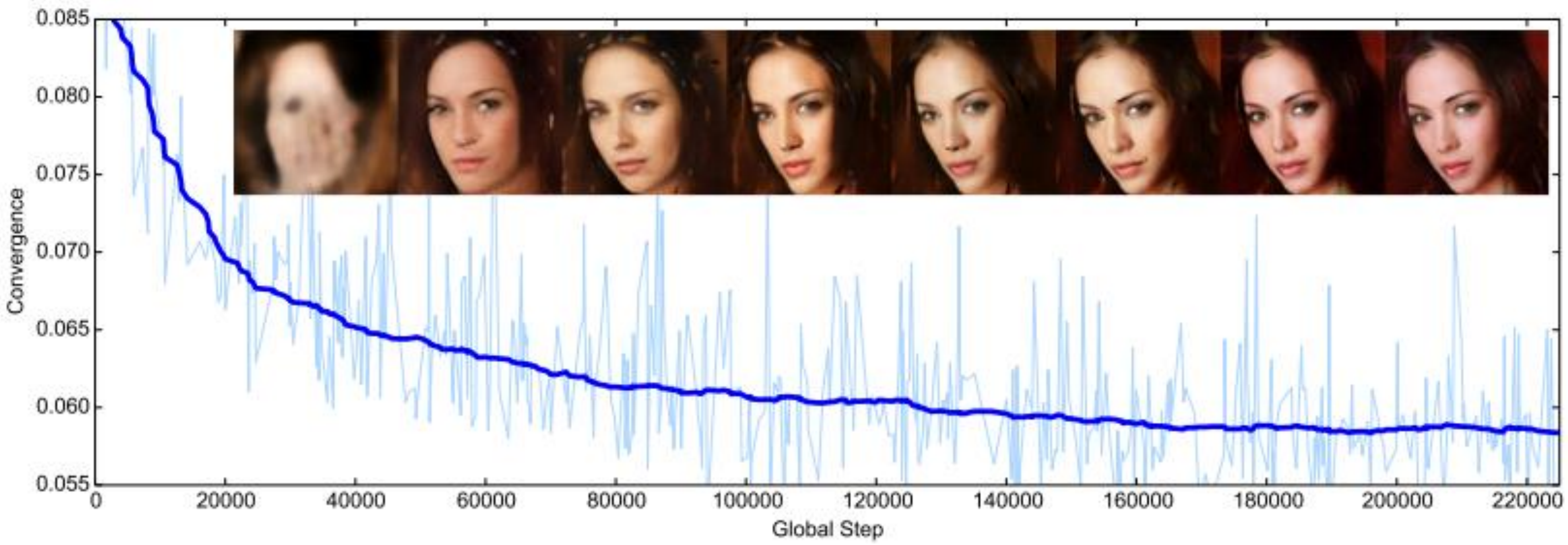
Richard Zhang et al. Real-Time User-Guided Image Colorization with Learned Deep Prioirs, 2017

05
Future of GAN

Convergence Measure

- Boundary Equilibrium GAN (BEGAN)

$$\mathcal{M}_{global} = \mathcal{L}(x) + |\gamma\mathcal{L}(x) - \mathcal{L}(G(z_G))|$$



David Berthelot et al. BEGAN: Boundary Equilibrium Generative Adversarial Networks, 2017

Convergence Measure

- Reconstruction Loss

$$\mathcal{L}_{\text{rec}}(G, X) = \frac{1}{m} \sum_{i=1}^{m} \min_{z} ||G(z) - x^{(i)}||^2$$

G(z)

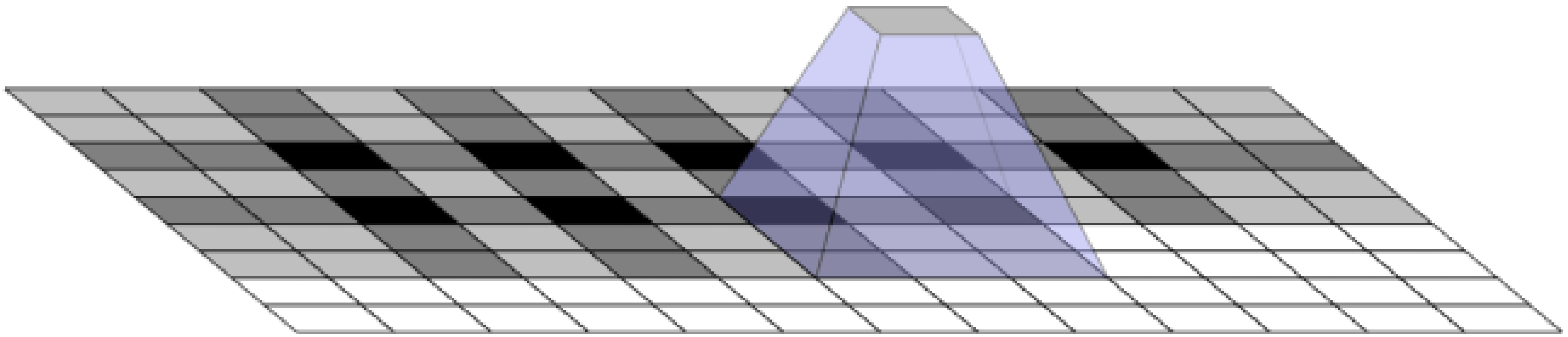
X

Test images

Sitao Xiang. On the effect of Batch Normalization and Weight Normalization in Generative Adversarial Network, 2017

Better Upsampling

- Deconvolution Checkboard Artifacts



http://distill.pub/2016/deconv-checkerboard/

Better Upsampling

- Deconvolution vs Resize-Convolution

Deconv in last two layers.
Other layers use resize-convolution
Artifacts of frequency 2 and 4.

Deconv only in last layer.
Other layers use resize-convolution
Artifacts of frequency 2.

All layers use resize-convolution.
No artifacts.

http://distill.pub/2016/deconv-checkerboard/

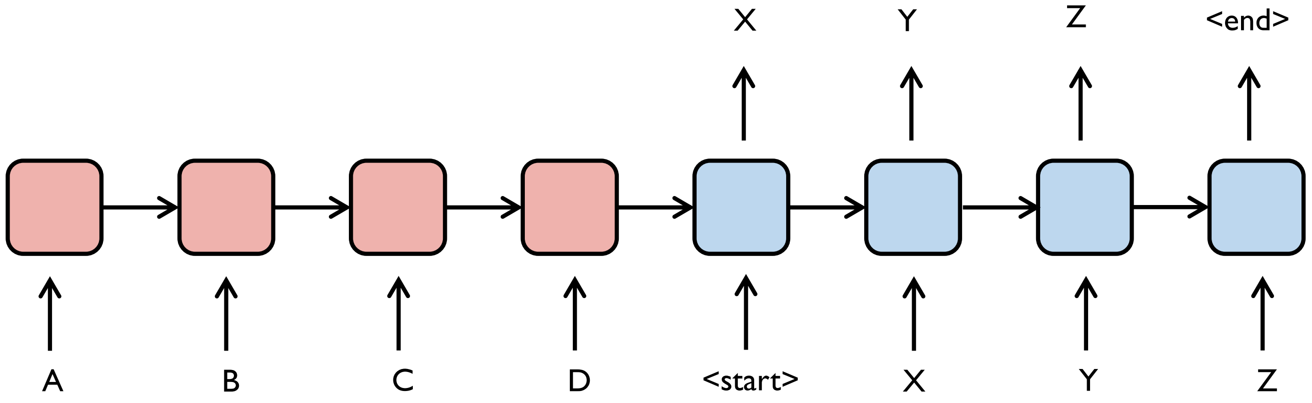
GAN in Supervised Learning

Extensions

- Machine Translation (Seq2Seq)

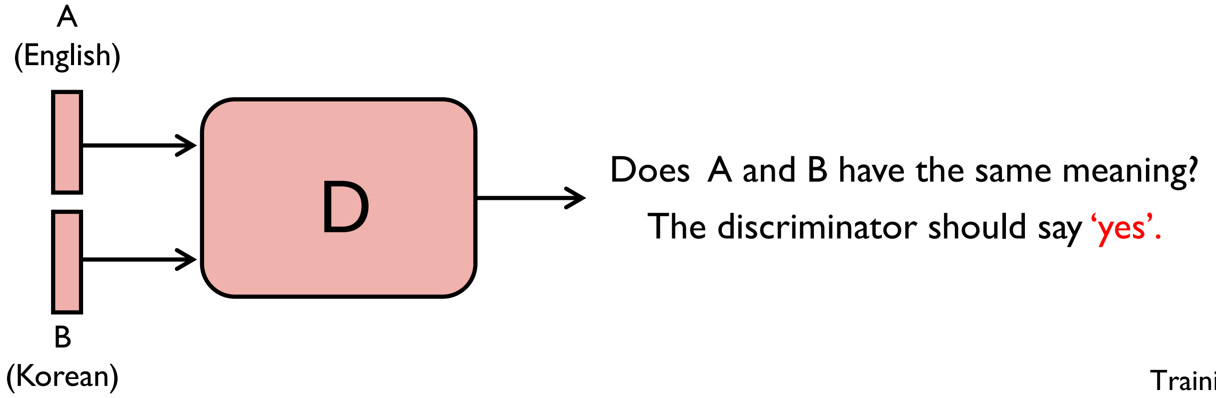
Should ‘ABCD’ be translated to ‘XYZ’?

Tackling the supervised learning



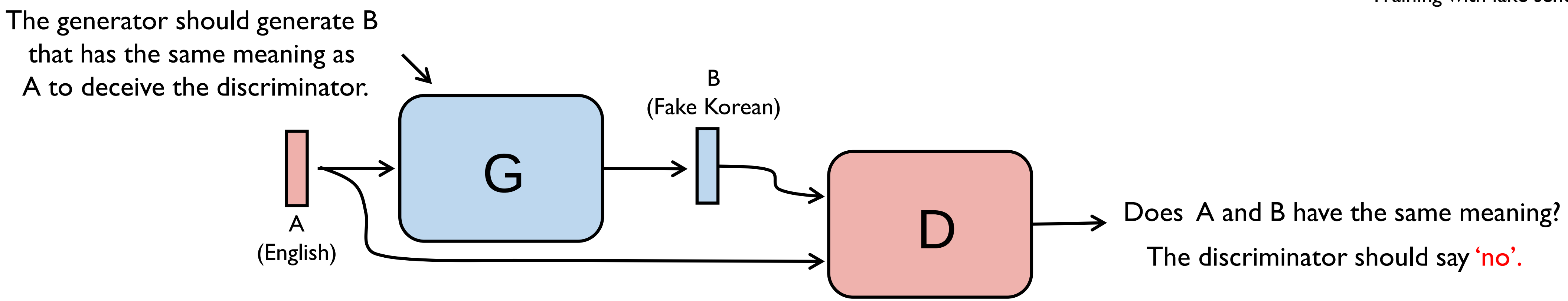
GAN in Supervised Learning

- Machine Translation (GANs)



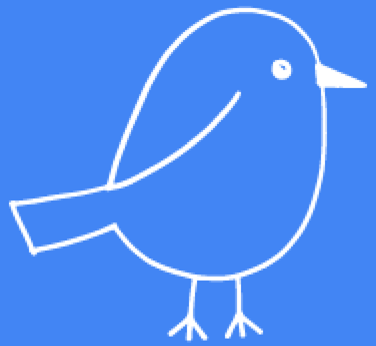
Training with real sentences

Training with fake sentences



Lijun Wu et al. Adversarial Neural Machine Translation, 2017

Thank you



06
Appendix

Theory in GAN

high dimensional vector (e.g. 64 × 64)

low dimensional vector (e.g. 100)

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

Objective function of GANs

real data distribution

Gaussian distribution

Theory in GAN

$$\cancel{\min_G} \max_D V(D,\cancel{G}) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Fix G to make it to a function of D

$$D^*(x) = arg \max_D V(D) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Optimal D

Get D when $V(D)$ is maximum

Theory in GAN

GANs

$$\cancel{\min_G} \max_D V(D, \cancel{G}) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Fix G to make it to a function of D

$$D^*(x) = \arg\max_D V(D) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

$$= E_{x\sim p_{data}(x)}[\log D(x)] + E_{x\sim p_g(x)}[\log(1 - D(x))]$$

vector of 64×64 *dimension*

sampling x from p_g

instead of sampling z from p_z

vector of 100 dimension

model G distribution for high dimensional vector (e.g. 64×64*)*

Theory in GAN

GANs

$$\cancel{\min_G} \max_D V(D, \cancel{G}) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Fix G to make it to a function of D

$$D^*(x) = arg \max_D V(D) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

sampling x from p_g
instead of sampling z from p_z

$$= E_{x\sim p_{data}(x)}[\log D(x)] + E_{x\sim p_g(x)}[\log(1 - D(x))]$$

Definition of Expectation

$$E_{x\sim p(x)}[f(x)] = \int_x p(x) f(x) dx$$

$$= \int_x p_{data}(x) \log D(x)\, dx + \int_x p_g(x) \log\big(1 - D(x)\big)\, dx$$

Integrate for all possible x

Theory in GAN

GANs

$$\cancel{\min_G} \max_D V(D,\cancel{G}) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

Fix G to make it to a function of D

$$D^*(x) = \arg\max_D V(D) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z))]$$

sampling x from p_g
instead of sampling z from p_z

$$= E_{x\sim p_{data}(x)}[\log D(x)] + E_{x\sim p_g(x)}[\log(1 - D(x))]$$

Definition of Expectation

$$E_{x\sim p(x)}[f(x)] = \int_x p(x)f(x)dx$$

$$= \int_x p_{data}(x)\log D(x)\,dx + \int_x p_g(x)\log\big(1 - D(x)\big)\,dx$$

Basic property of Integral

$$= \int_x p_{data}(x)\log D(x) + p_g(x)\log\big(1 - D(x)\big)\,dx$$

Theory in GAN

GANs

$$\cancel{\min_G} \max_D V(D, \cancel{G}) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

Fix G to make it to a function of D

$$D^*(x) = arg \max_D V(D) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

sampling x from p_g
instead of sampling z from p_z

$$= E_{x \sim p_{data}(x)}[\log D(x)] + E_{x \sim p_g(x)}[\log(1 - D(x))]$$

Definition of Expectation

$$E_{x \sim p(x)}[f(x)] = \int_x p(x) f(x) dx$$

$$= \int_x p_{data}(x) \log D(x)\, dx + \int_x p_g(x) \log(1 - D(x))\, dx$$

Basic property of Integral

$$= \int_x \boxed{p_{data}(x) \log D(x) + p_g(x) \log(1 - D(x))} dx$$

Now we need to find $D(x)$ which makes the funtion inside integral maximum.

Theory in GAN

$$D^*(x) = \underset{D}{arg\,max}\, V(D)$$

$$= \underset{D}{arg\,max}\;\; p_{data}(x) \log D(x) + p_g(x) \log\big(1 - D(x)\big)$$

$The\ funtion\ inside\ integral$

Theory in GAN

GANs

$$D^*(x) = \arg\max_D V(D)$$

$$= \arg\max_D \; p_{data}(x)\log D(x) + p_g(x)\log\big(1 - D(x)\big)$$

Substitute $a = p_{data}(x),\; y = D(x),\; b = p_g(x)$

$$a\log y + b\log(1 - y)$$

Theory in GAN

GANs

$$D^*(x) = \underset{D}{arg\,max}\, V(D)$$

$$= \underset{D}{arg\,max}\; p_{data}(x) \log D(x) + p_g(x) \log\big(1 - D(x)\big)$$

Substitute $a = p_{data}(x),\ y = D(x),\ b = p_g(x)$

$$a \log y + b \log(1 - y)$$

$$\frac{a}{y} + \frac{-b}{1-y} = \frac{a - (a+b)y}{y(1-y)}$$

Differentiate with respect to $D(x)$ *using* $\frac{d}{dx} \log f(x) = \frac{f'(x)}{f(x)}$

Note that $D(x)$ *can not affect to* $p_{data}(x)$ *and* $p_g(x)$.

Theory in GAN

$$D^*(x) = \arg\max_D V(D)$$

$$= \arg\max_D \; p_{data}(x)\log D(x) + p_g(x)\log\big(1 - D(x)\big)$$

Substitute $a = p_{data}(x),\ y = D(x),\ b = p_g(x)$

$$a\log y + b\log(1 - y)$$

Differentiate with respect to $D(x)$ *using* $\frac{d}{dx}\log f(x) = \frac{f'(x)}{f(x)}$

Note that $D(x)$ *can not affect to* $p_{data}(x)$ *and* $p_g(x)$.

$$\frac{a}{y} + \frac{-b}{1-y} = \frac{a-(a+b)y}{y(1-y)}$$

Find the point where the derivative value is 0 *(local extreme).*

$$\frac{a-(a+b)y}{y(1-y)} = 0$$

It has a maximum value when $y = \frac{a}{a+b}$

Note that the local maximum is the global maximum when there are no other local extremes.

Theory in GAN

GANs

$$D^*(x) = \arg\max_D V(D)$$

$$= \arg\max_D \; p_{data}(x) \log D(x) + p_g(x) \log(1 - D(x))$$

Substitute $a = p_{data}(x),\; y = D(x),\; b = p_g(x)$

$$a \log y + b \log(1 - y)$$

Differentiate with respect to $D(x)$ *using* $\frac{d}{dx} \log f(x) = \frac{f'(x)}{f(x)}$

Note that $D(x)$ *can not affect to* $p_{data}(x)$ *and* $p_g(x)$.

$$\frac{a}{y} + \frac{-b}{1-y} = \frac{a-(a+b)y}{y(1-y)}$$

Find the point where the derivative value is 0 *(local extreme).*

$$\frac{a-(a+b)y}{y(1-y)} = 0$$

It has a maximum value when $y = \frac{a}{a+b}$

Substitute $a = p_{data}(x),\; y = D(x),\; b = p_g(x)$

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Theory in GAN

GANs

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

$$\begin{aligned}
V(D^*,G) &= E_{x\sim p_{data}}[\log D^*(x)] + E_{x\sim p_g}[\log(1 - D^*(x))] \\
&= \int_x p_{data}(x)\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{p_g(x)}{p_{data}(x)+p_g(x)}dx \\
&= -log4 + log4 + \int_x p_{data}(x)\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{p_g(x)}{p_{data}(x)+p_g(x)}dx \\
&= -log4 + \int_x p_{data}(x)\log\frac{2\cdot p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{2\cdot p_g(x)}{p_{data}(x)+p_g(x)}dx \\
&= -log4 + KL(p_{data}||\frac{p_{data}+p_g}{2}) + KL(p_g||\frac{p_{data}+p_g}{2}) \\
&= -log4 + 2\cdot JSD(p_{data}||p_g)
\end{aligned}$$

Theory in GAN

GANs

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

Optimal D

$$V(D^*,G) = E_{x\sim p_{data}}[\log D^*(x)] + E_{x\sim p_g}[\log(1-D^*(x))]$$

G should minimize

$$= \int_x p_{data}(x)\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{p_g(x)}{p_{data}(x)+p_g(x)}dx$$

$$= -log4 + log4 + \int_x p_{data}(x)\log\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{p_g(x)}{p_{data}(x)+p_g(x)}dx$$

$$= -log4 + \int_x p_{data}(x)\log\frac{2\cdot p_{data}(x)}{p_{data}(x)+p_g(x)}dx + \int_x p_g(x)\log\frac{2\cdot p_g(x)}{p_{data}(x)+p_g(x)}dx$$

$$= -log4 + KL(p_{data}||\frac{p_{data}+p_g}{2}) + KL(p_g||\frac{p_{data}+p_g}{2})$$

$$= -log4 + 2\cdot JSD(p_{data}||p_g)$$

Optimizing $V(D,G)$ is same as minimizing $JSD(p_{data}||p_g)$

G should minimize