DEEP LEARNING

LECTURE 6: GENERATIVE ADVERSARIAL NETWORKS

goorm

KAIST AI
Graduate School of AI

DAVIAN
Data and Visual Analytics Lab

Reference Slides

- Namju Kim. Generative Adversarial Networks (GAN)

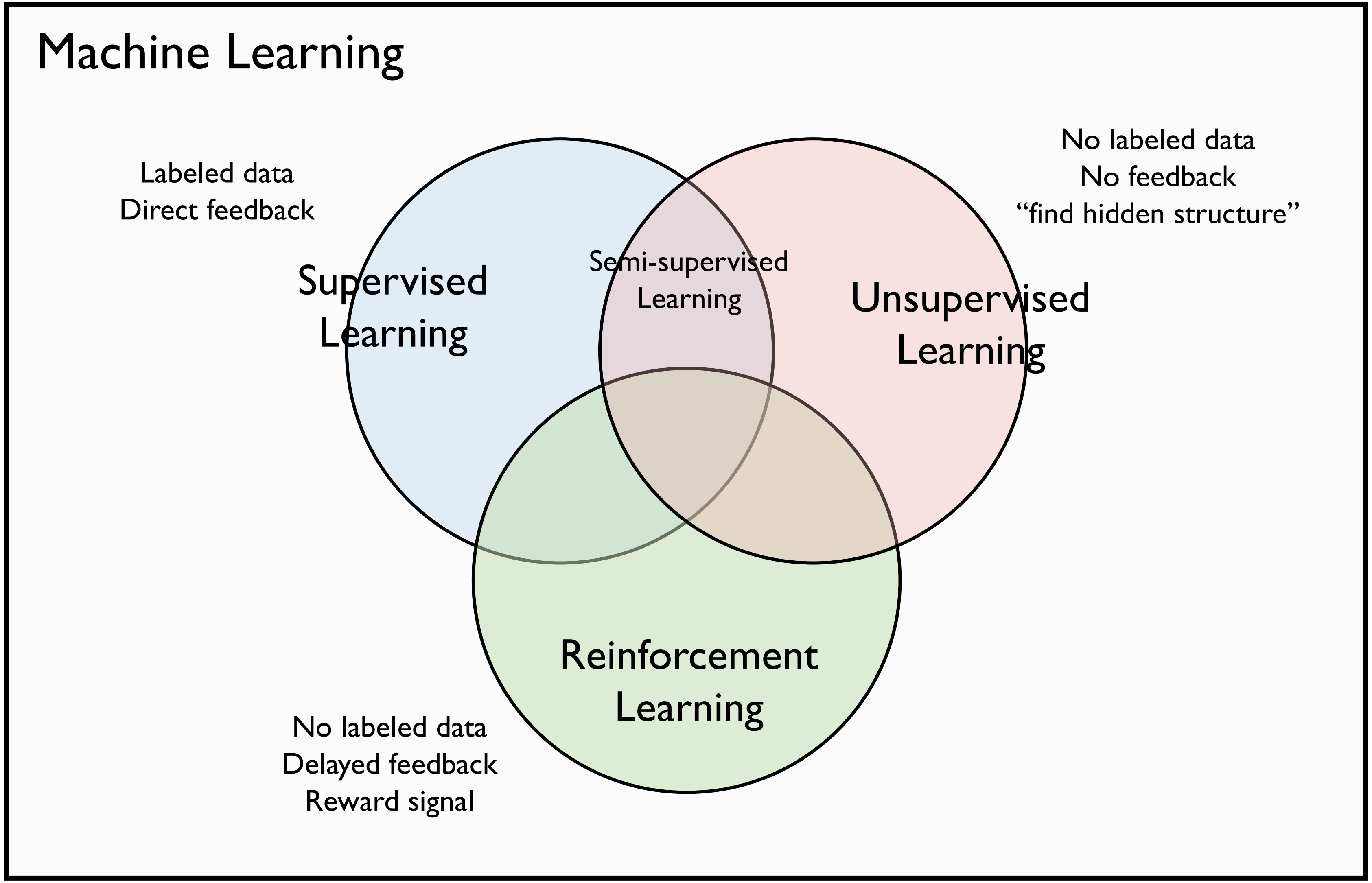
  https://www.slideshare.net/ssuser77ee21/generative-adversarial-networks-70896091

- Taehoon Kim. 지적 대화를 위한 깊고 넓은 딥러닝

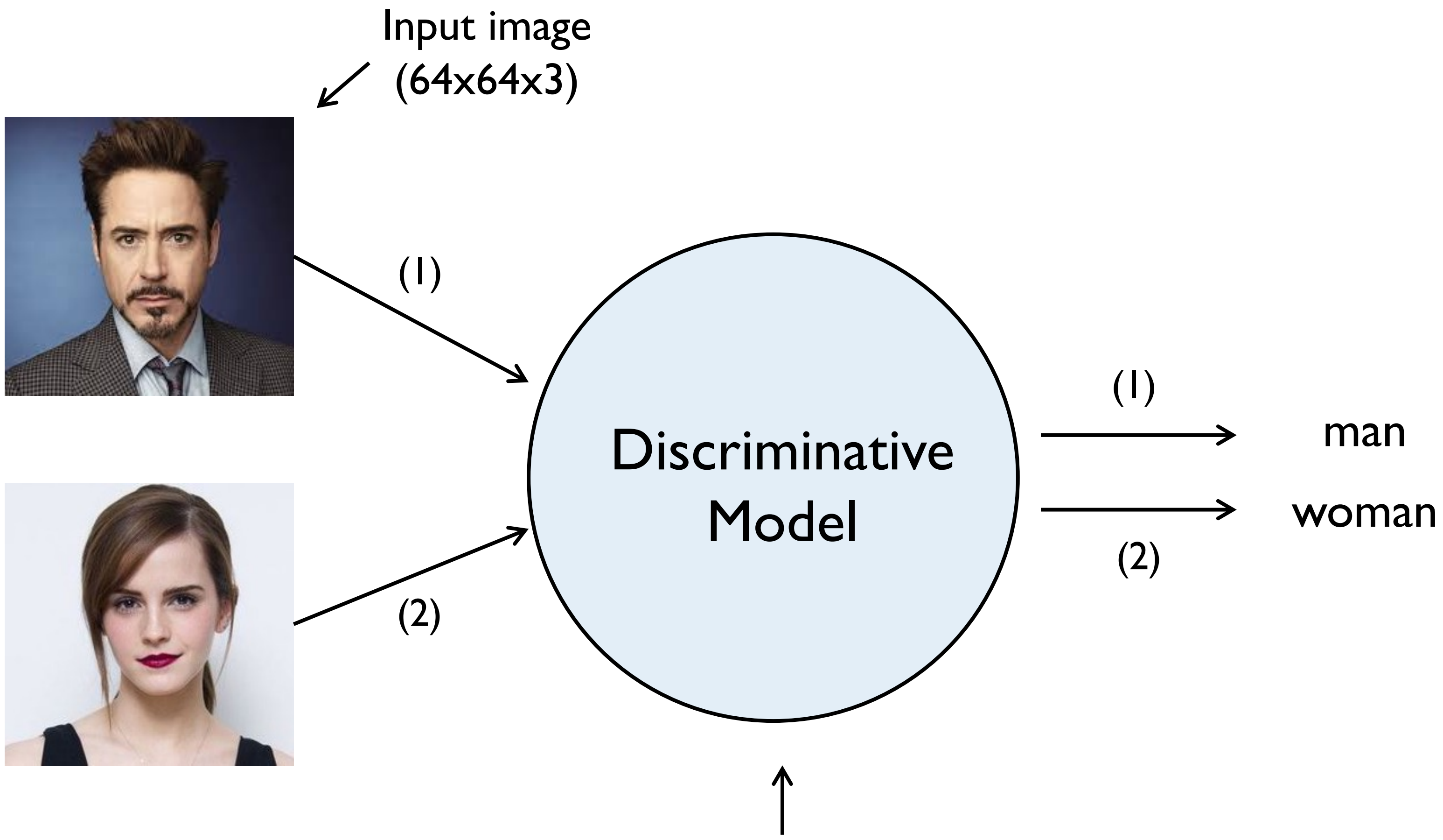
  https://www.slideshare.net/carpedm20/ss-63116251

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.

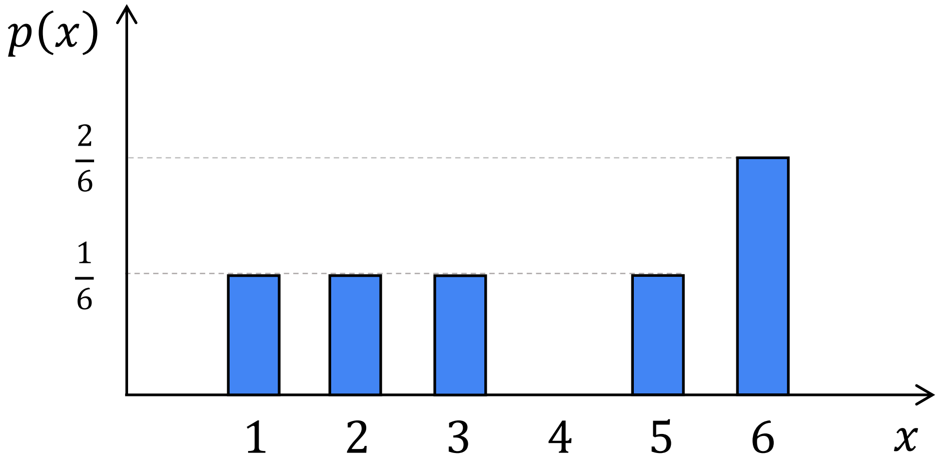
Probability Distribution

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

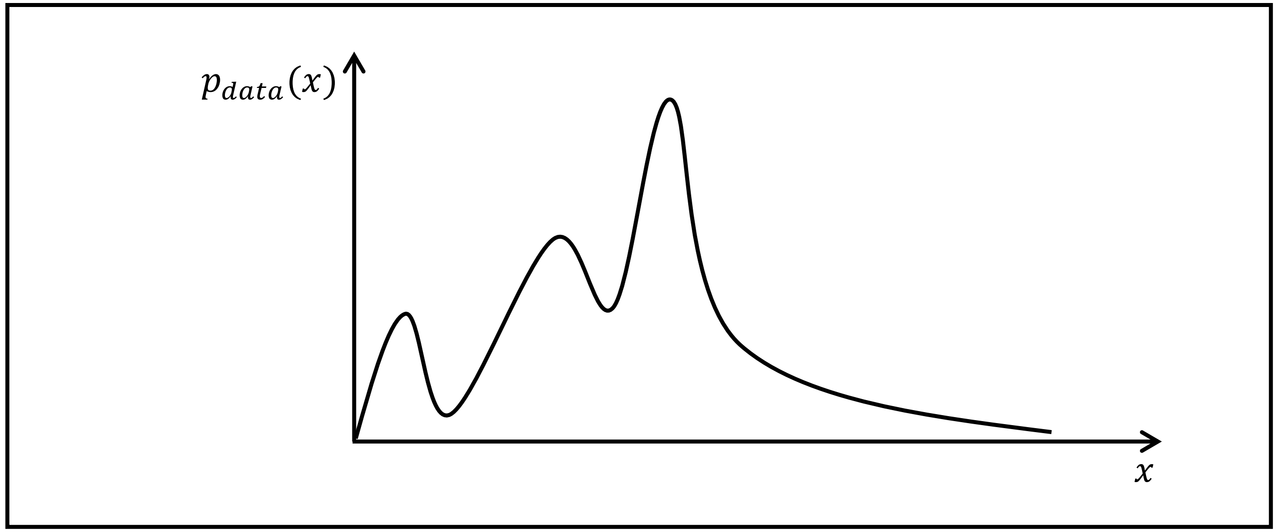
What if x is actual images in the training data?

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

Probability Distribution

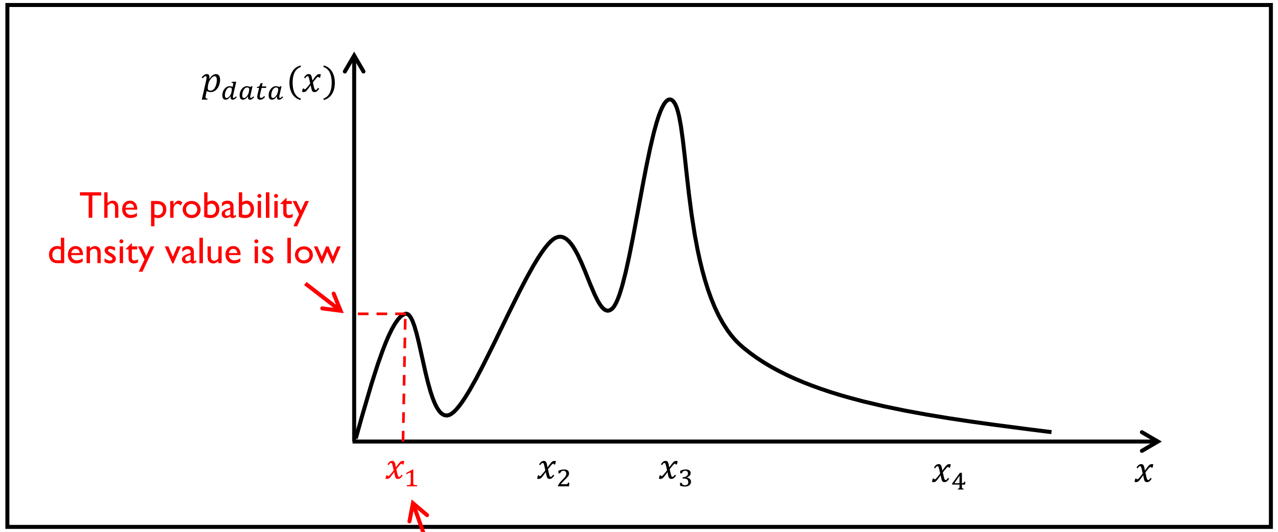
Probability density function

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



Probability Distribution

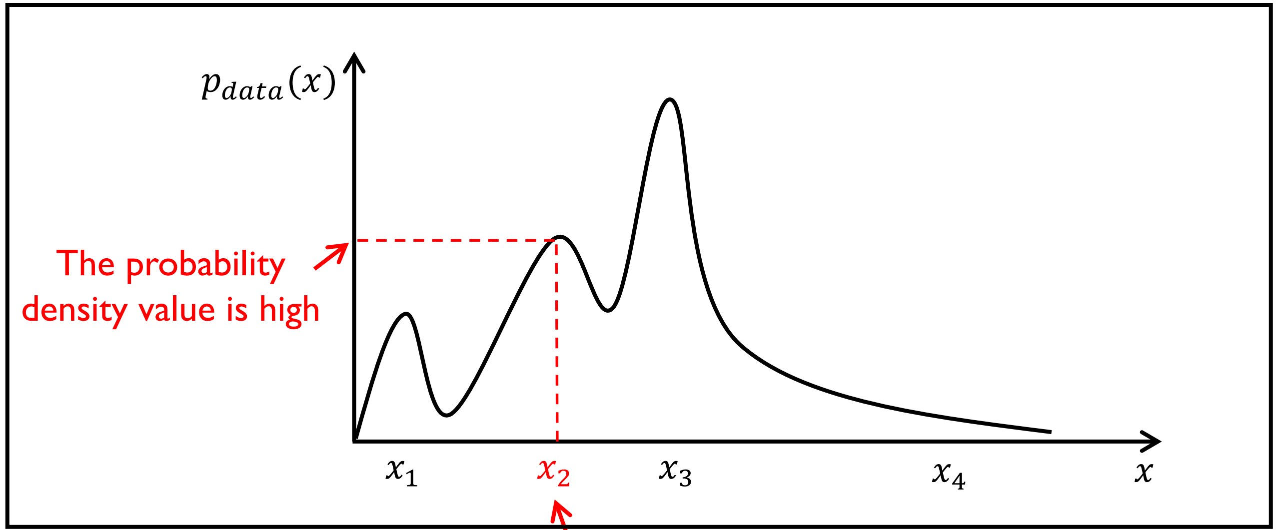
Let's take an example with human face image dataset.
Our dataset may contain few images of men with glasses.



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

Probability Distribution

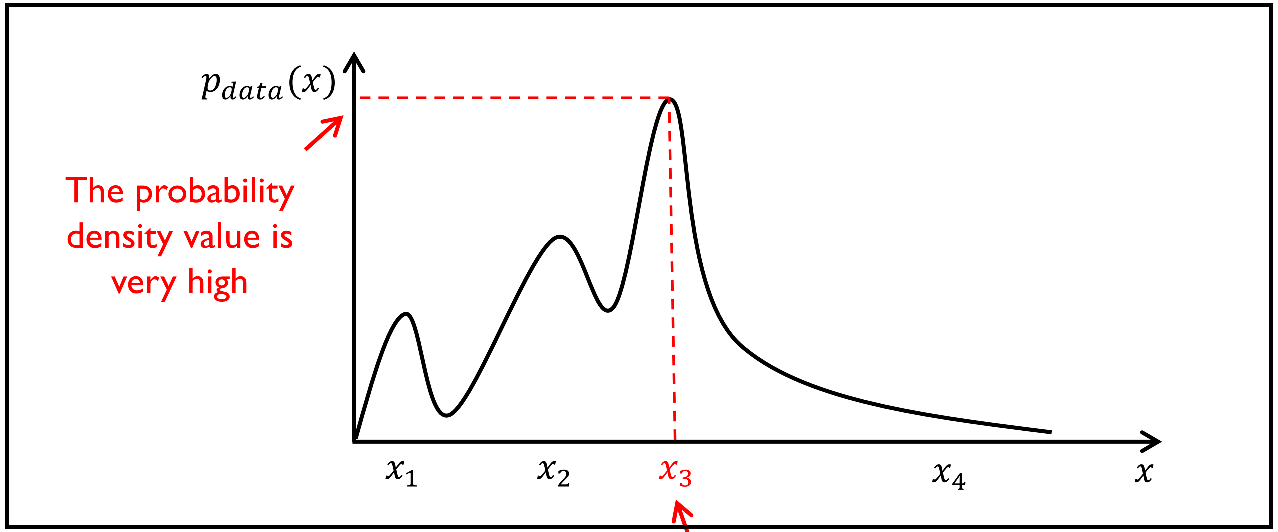
Our dataset may contain many images of women with black hair.



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

Probability Distribution

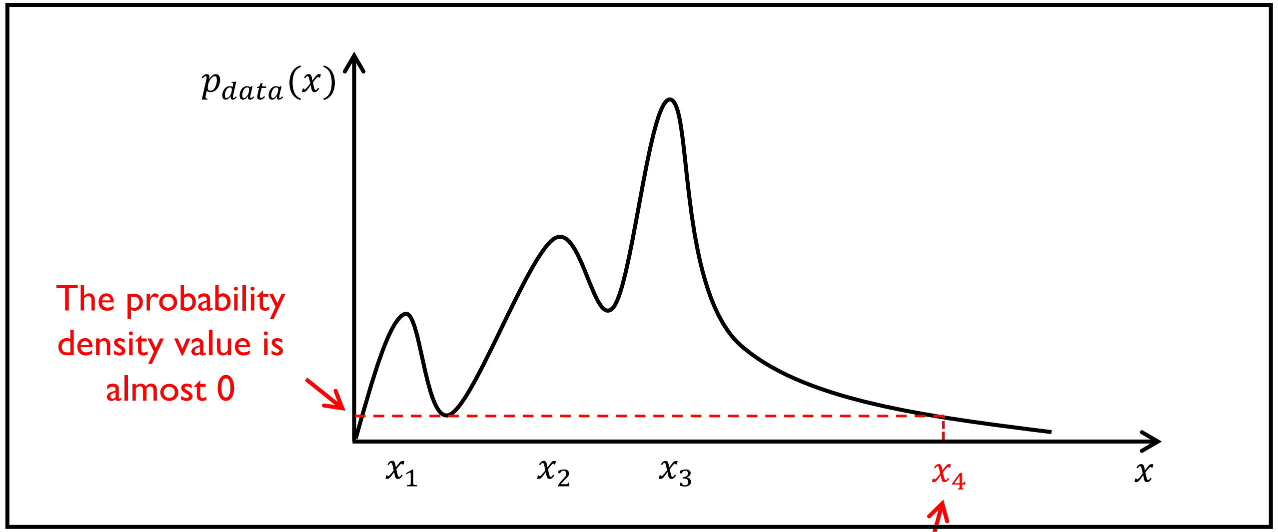
Our dataset may contain very many images of women with blonde hair.



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

Probability Distribution

Our dataset may not contain these strange images.



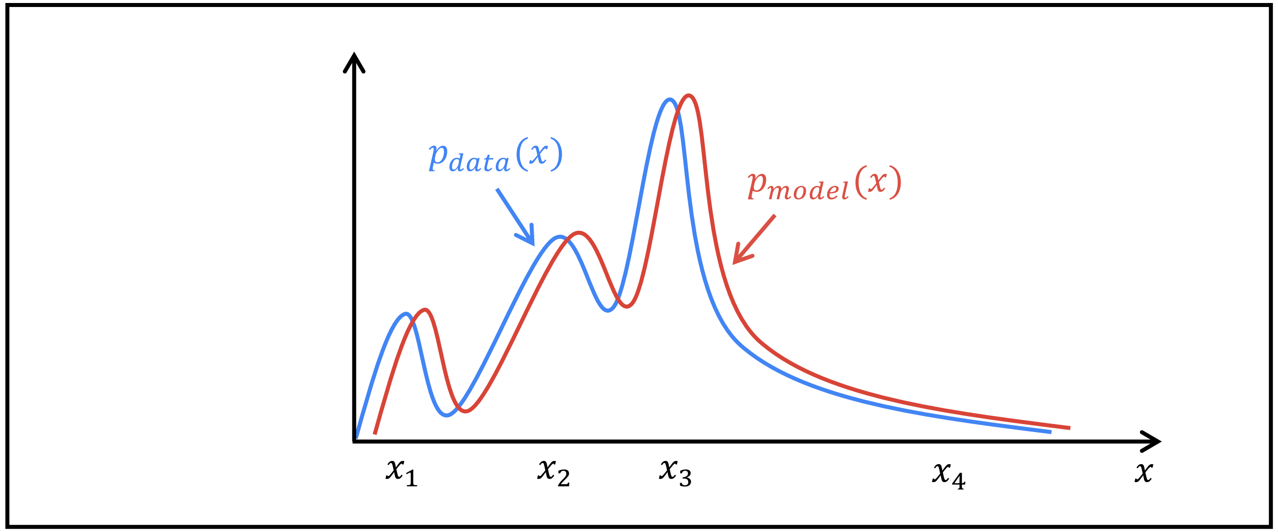
x_4 is an 64x64x3 high dimensional vector representing very strange images.

Probability Distribution

Distribution of images generated by the model

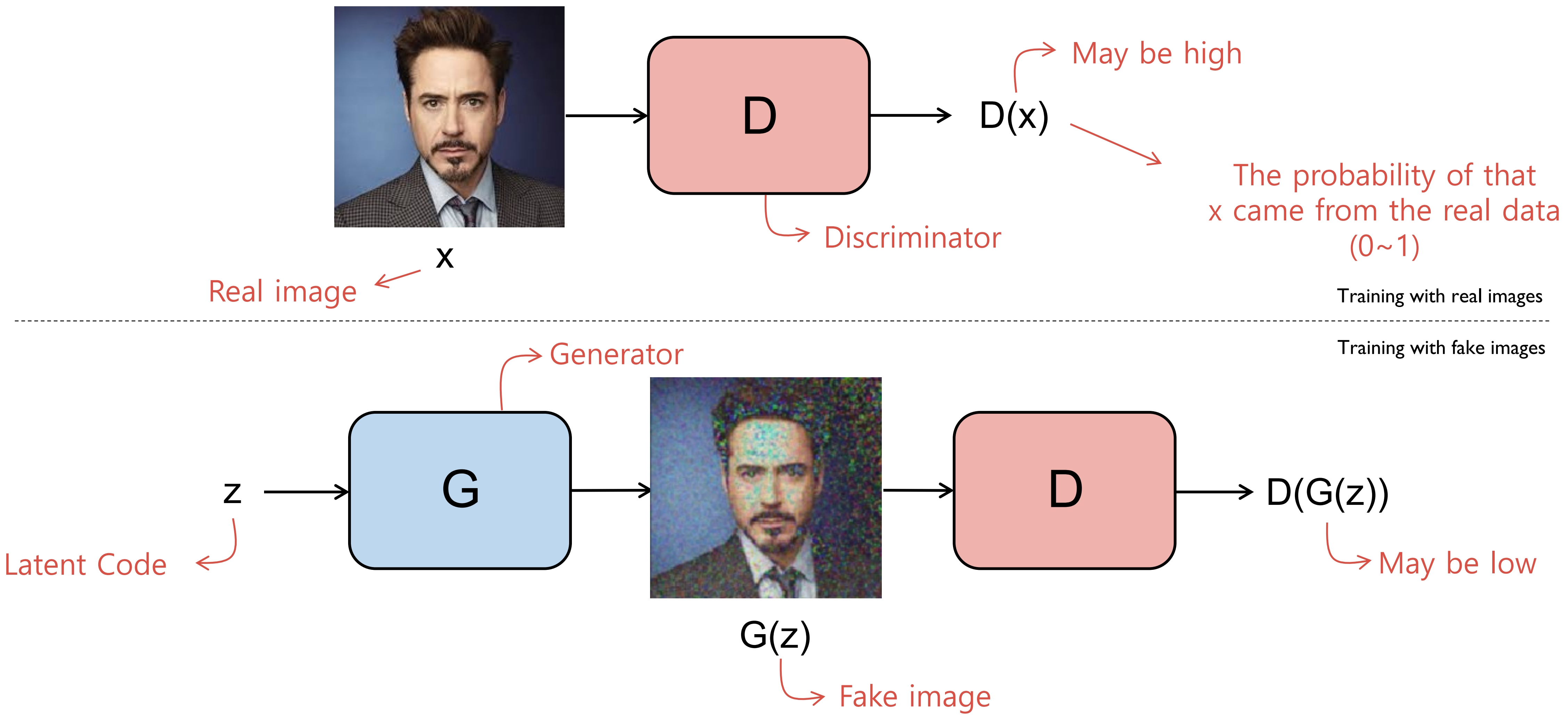
The goal of the generative model is to find a $p_{model}(x)$ that approximates $p_{data}(x)$ well.

Distribution of actual images

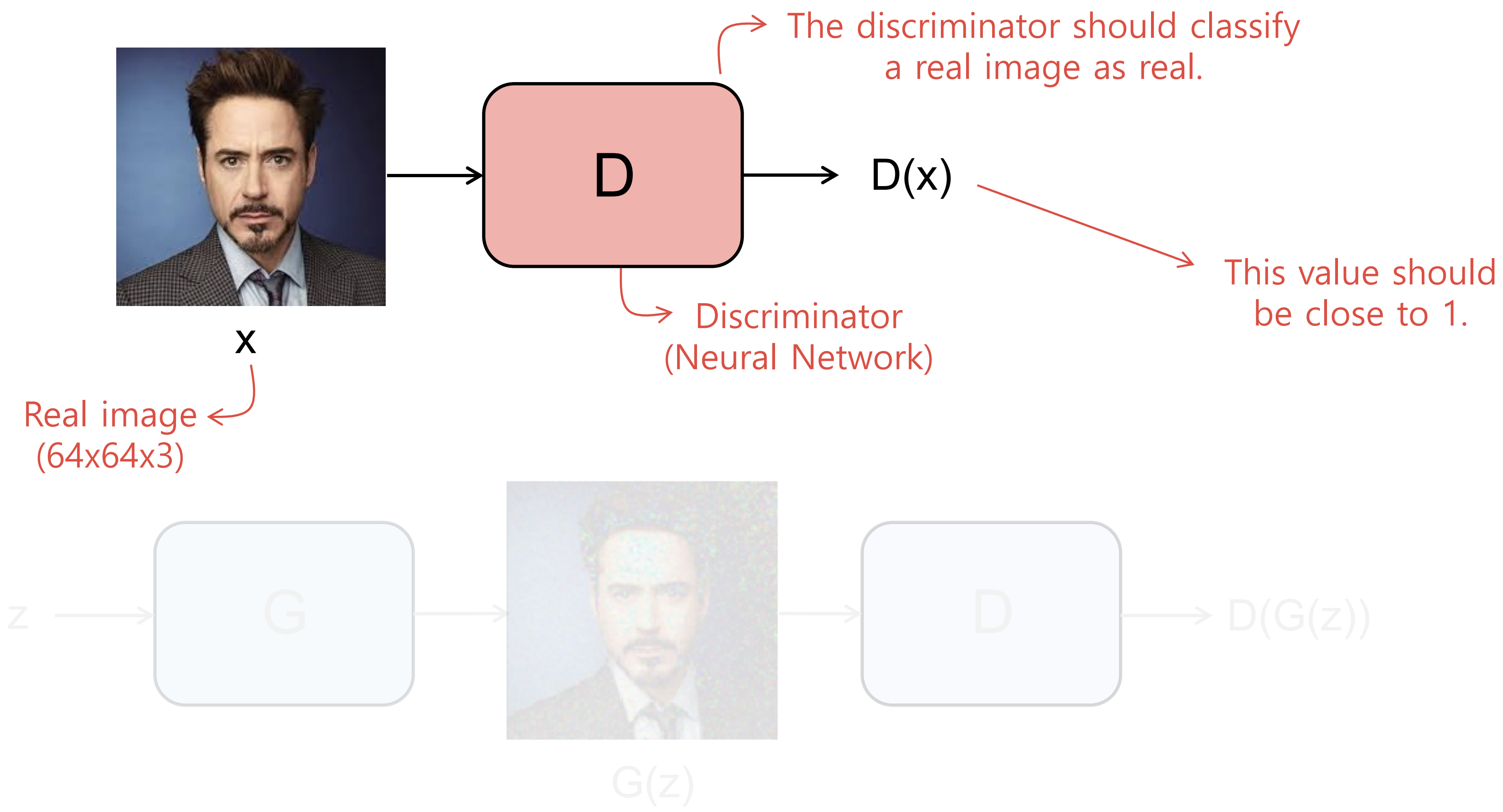


02
Generative Adversarial Networks

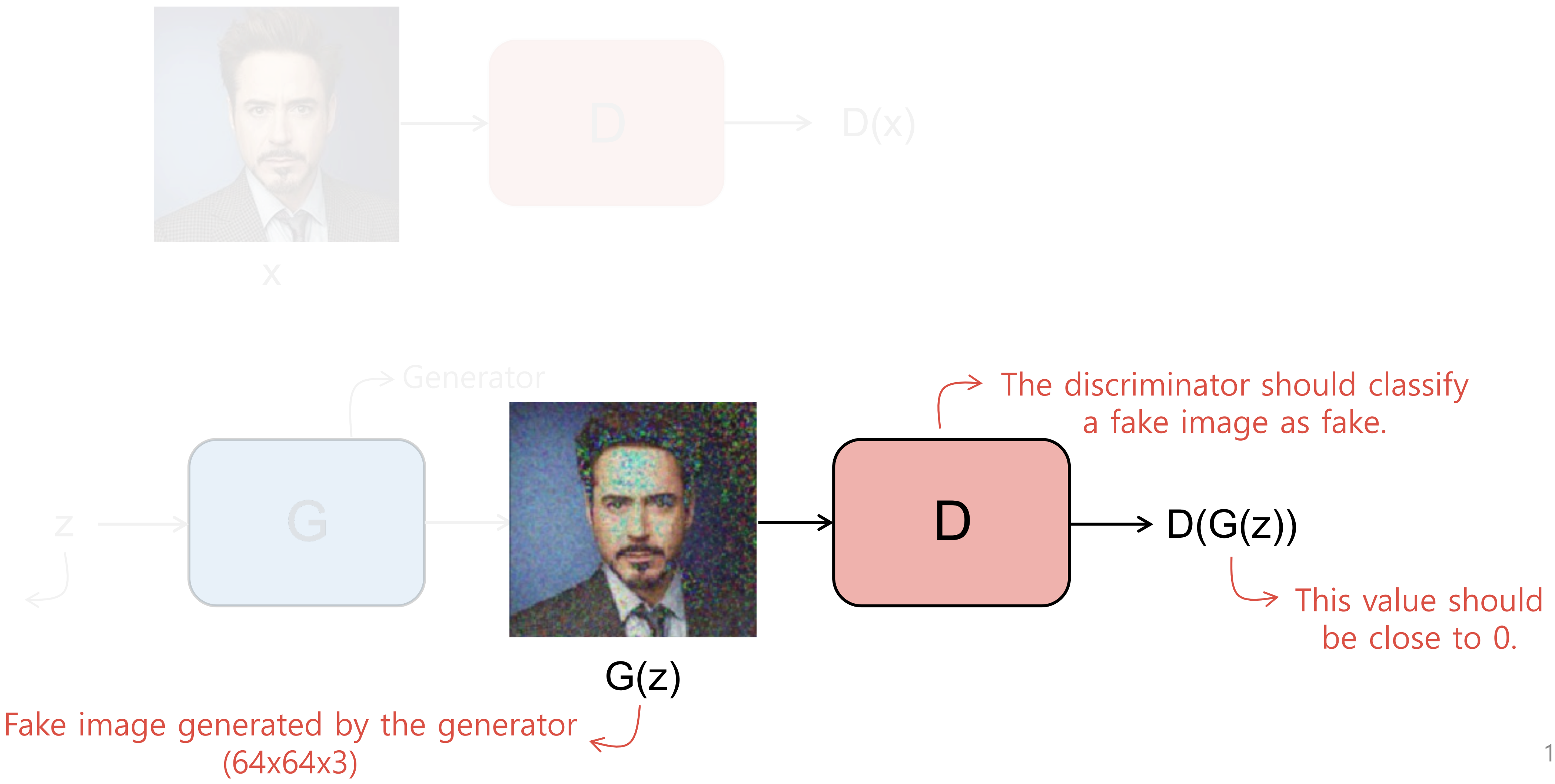
Intuition in GAN



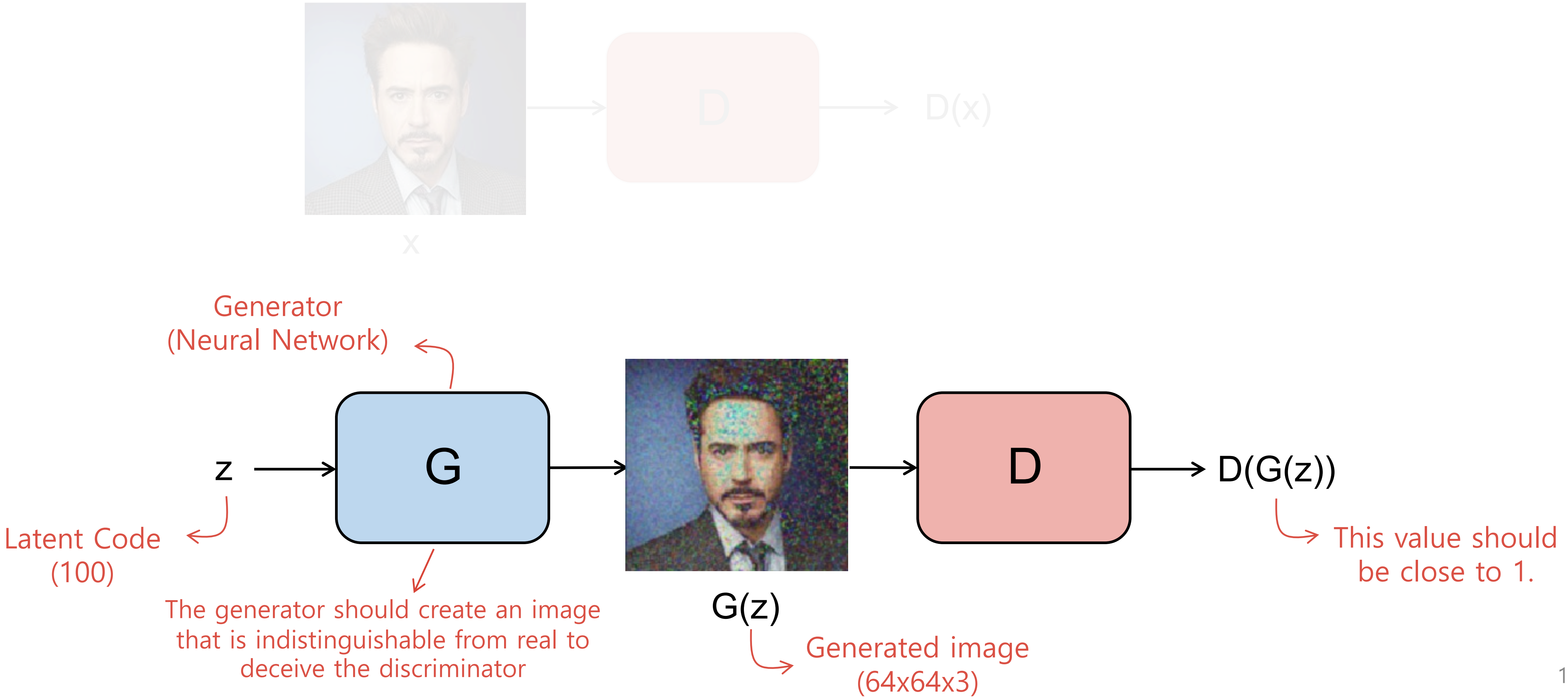
Intuition in GAN



Intuition in GAN



Intuition in GAN



Objective Function of GAN

Sample x from real data distribution

Sample latent code z from Gaussian distribution

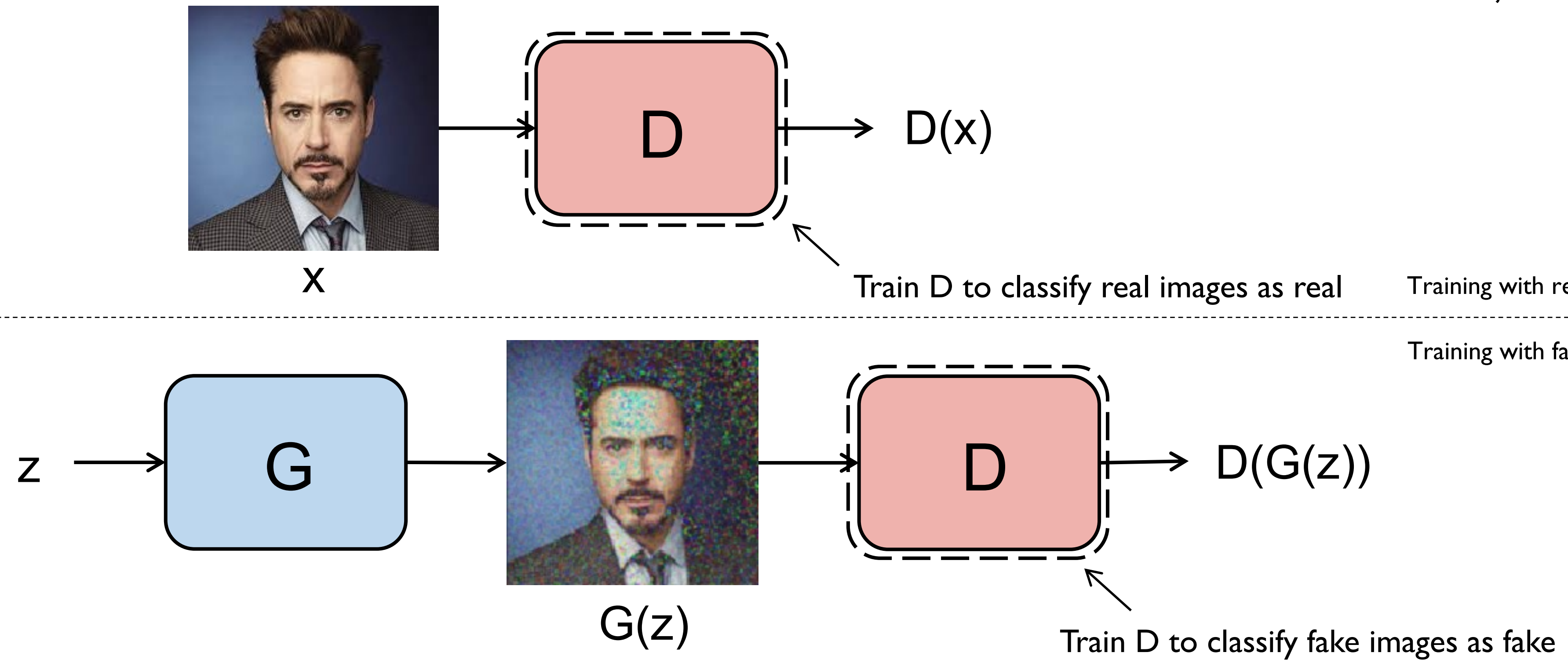
$$\min_{G}\max_{D} V(D,G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

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

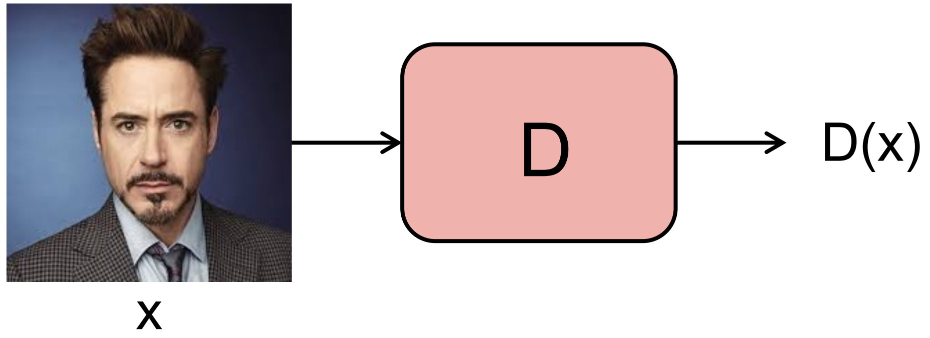
G is independent of this part

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

G should minimize $V(D,G)$

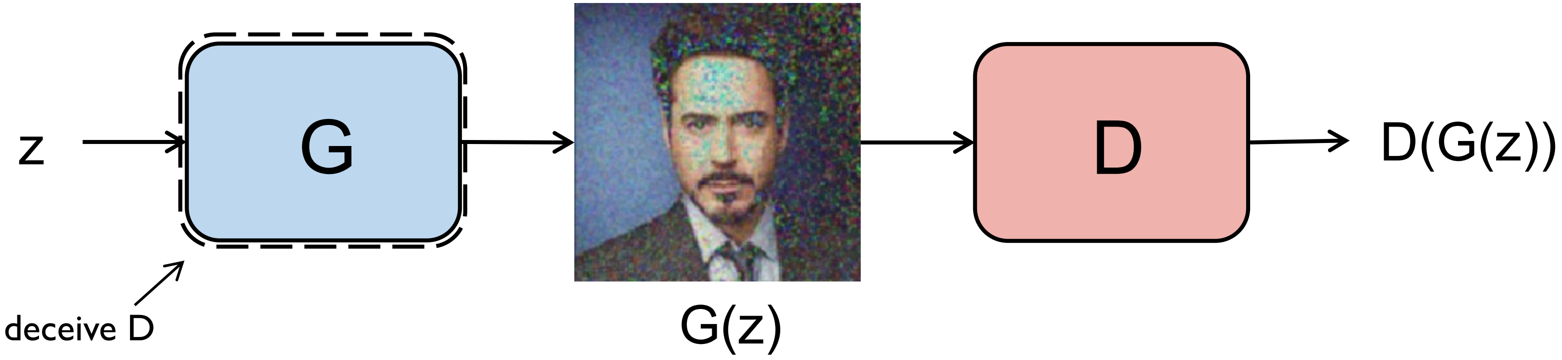
Minimum when $D(G(z)) = 1$

Objective function



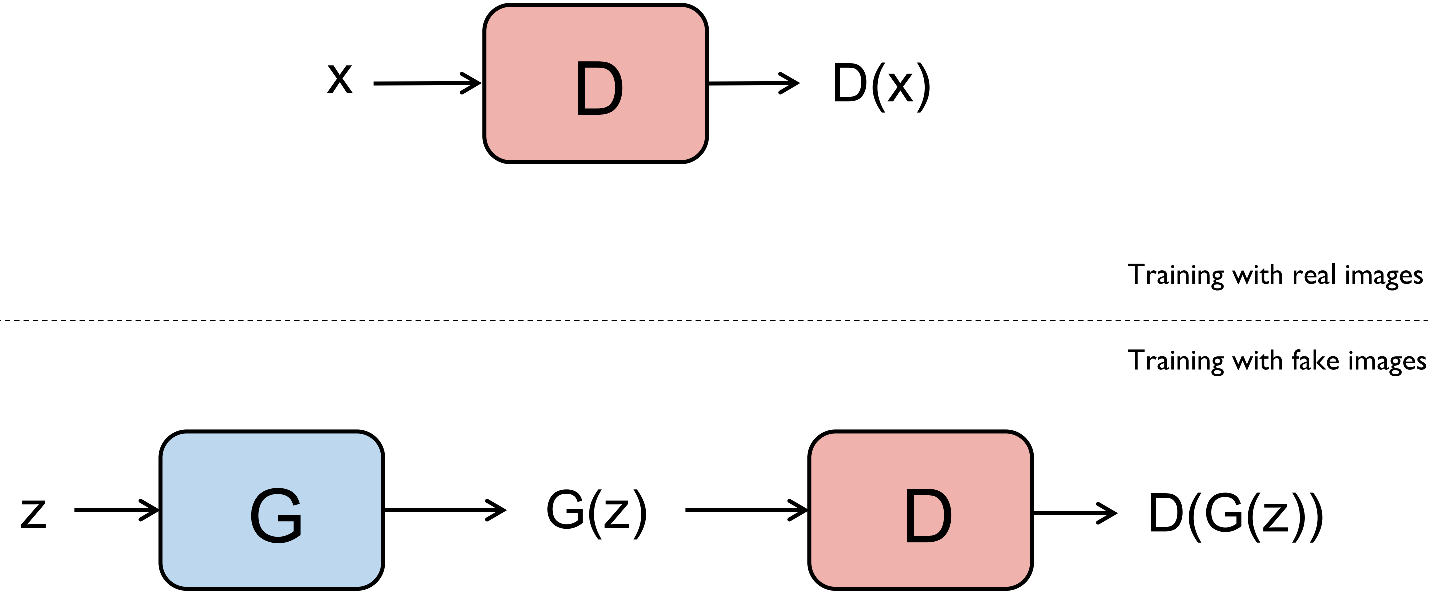
Training with real images

Training with fake images



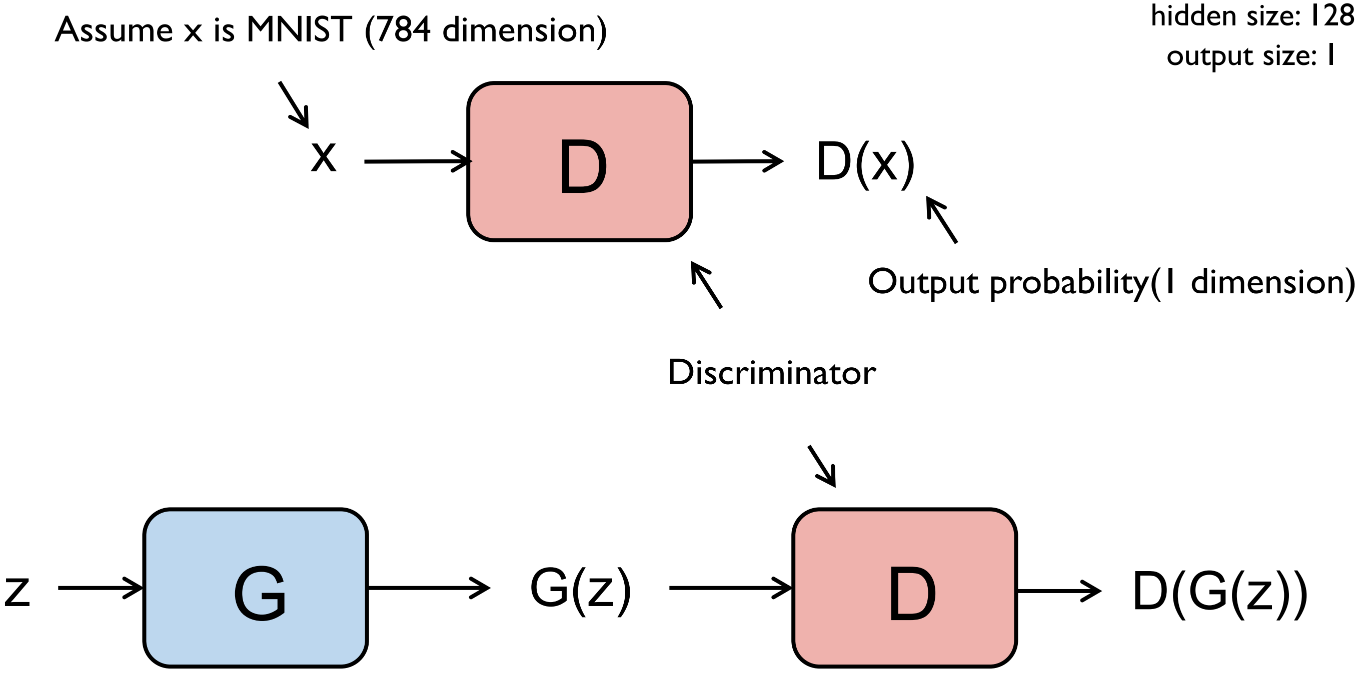
Train G to deceive D

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()
```

PyTorch Implementation

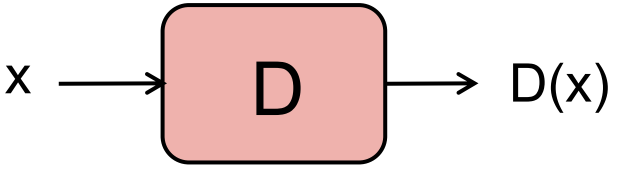


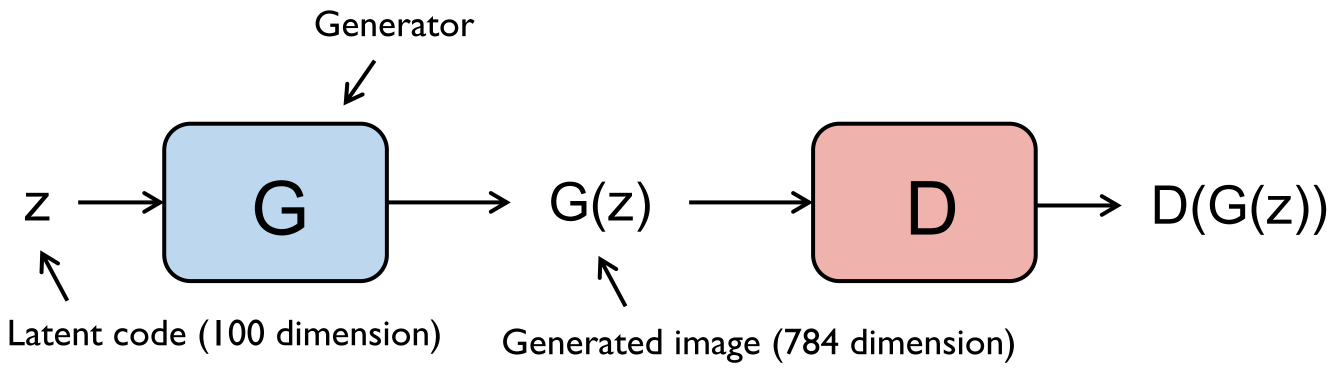
Define the discriminator

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



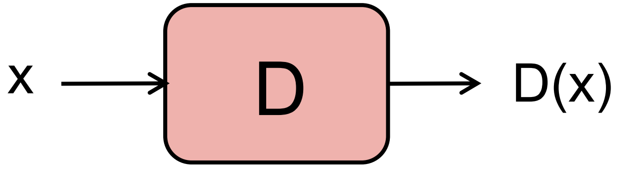


Define the generator

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

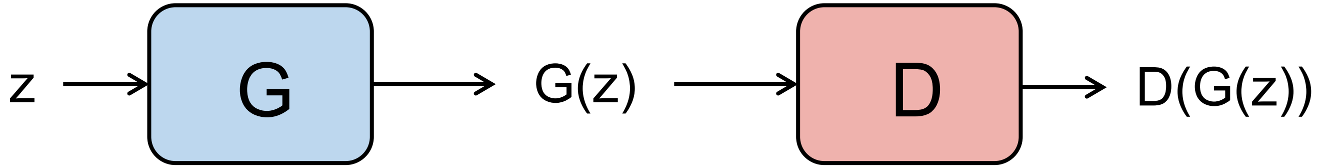
```python
import torch
import torch.nn as nn


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



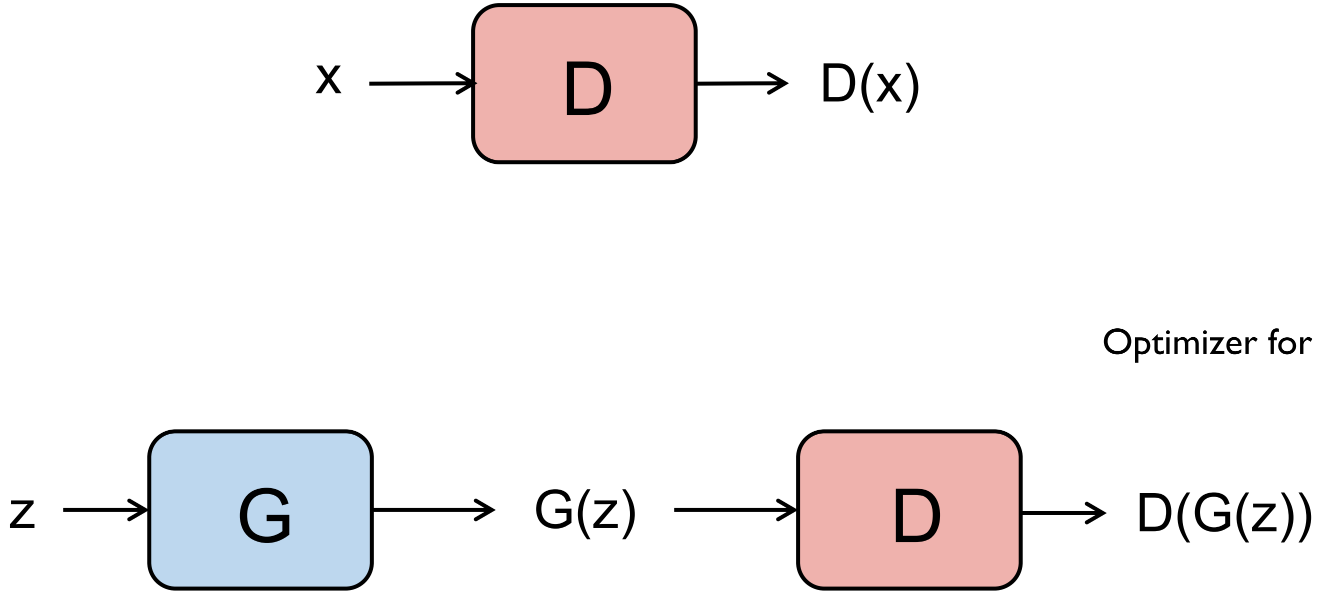
Binary Cross Entropy Loss $(h(x), y)$

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




```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()
```

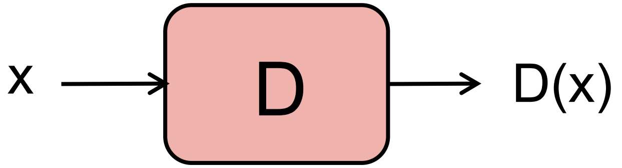
PyTorch Implementation



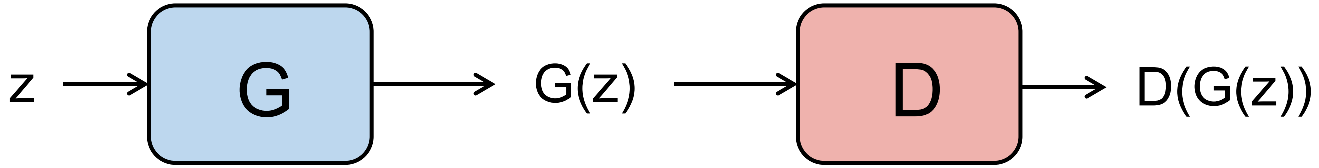
Optimizer for D and G

```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

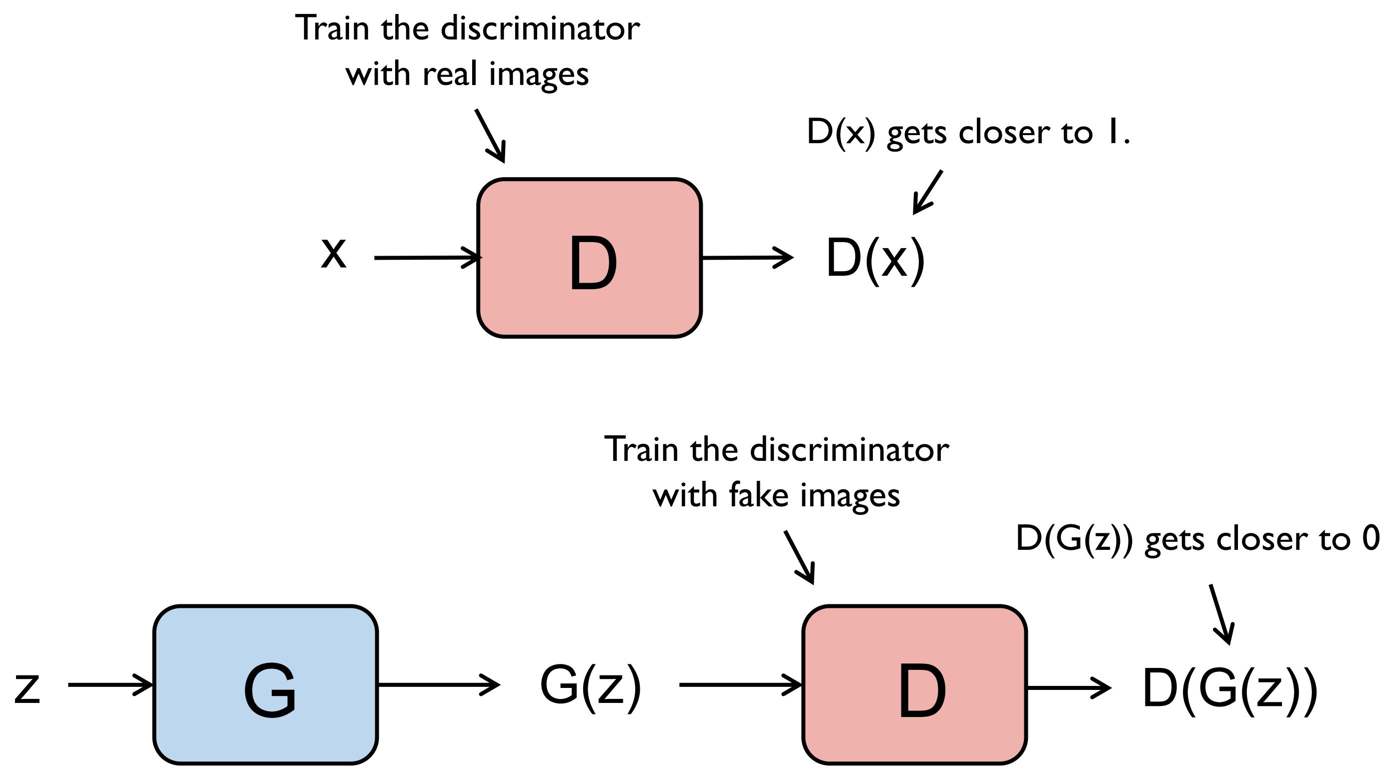


x is a tensor of shape (batch_size, 784).
z is a tensor of shape (batch_size, 100).




```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()
```

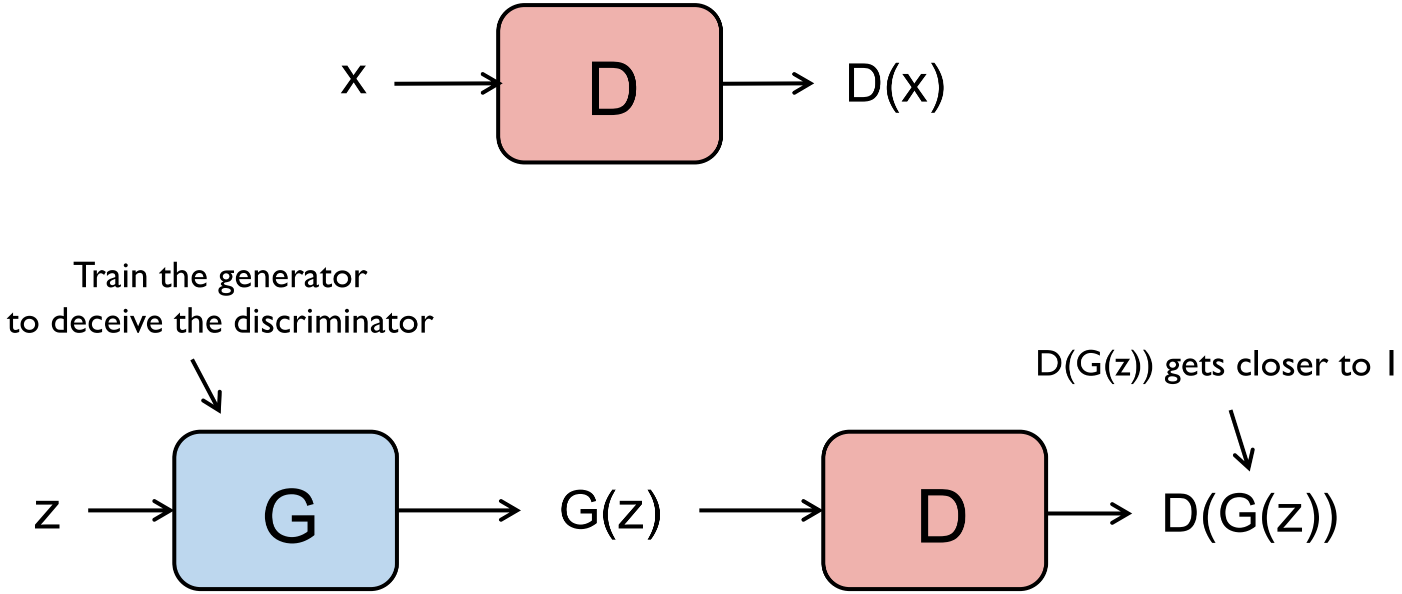
PyTorch Implementation



Forward, Backward and Gradient Descent

```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()
```

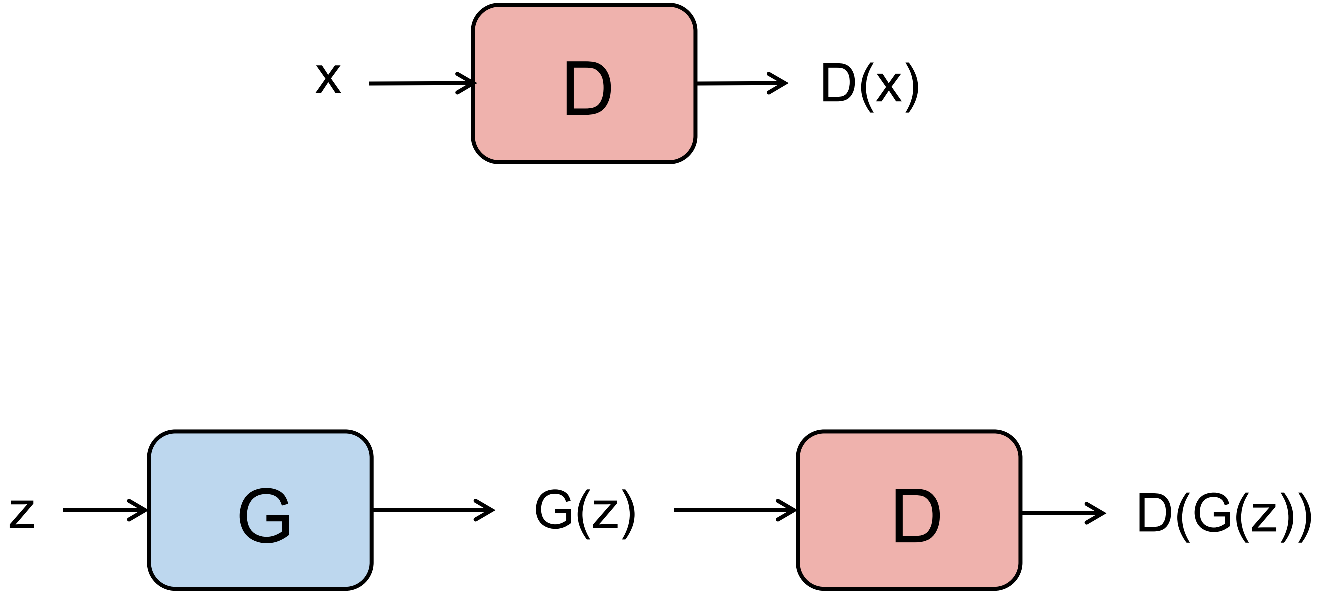
PyTorch Implementation




```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()
```

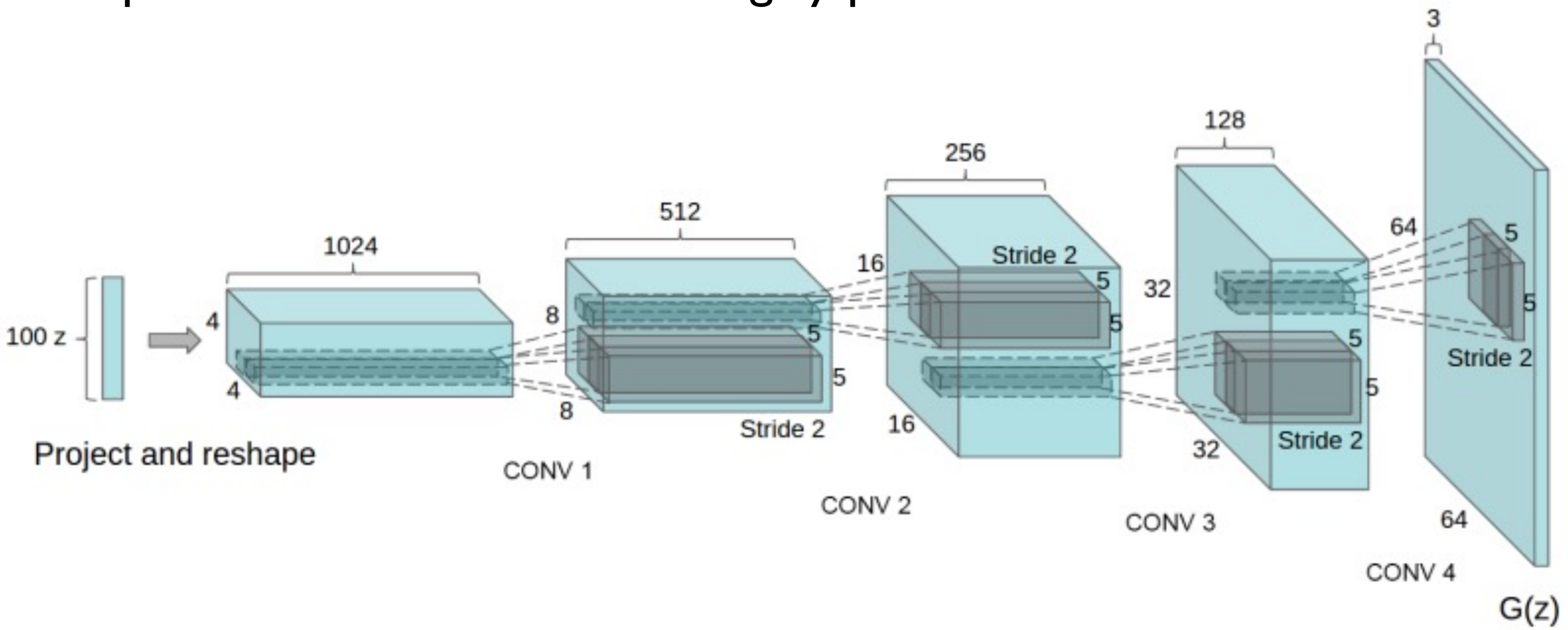
The complete code can be found here

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

DCGAN

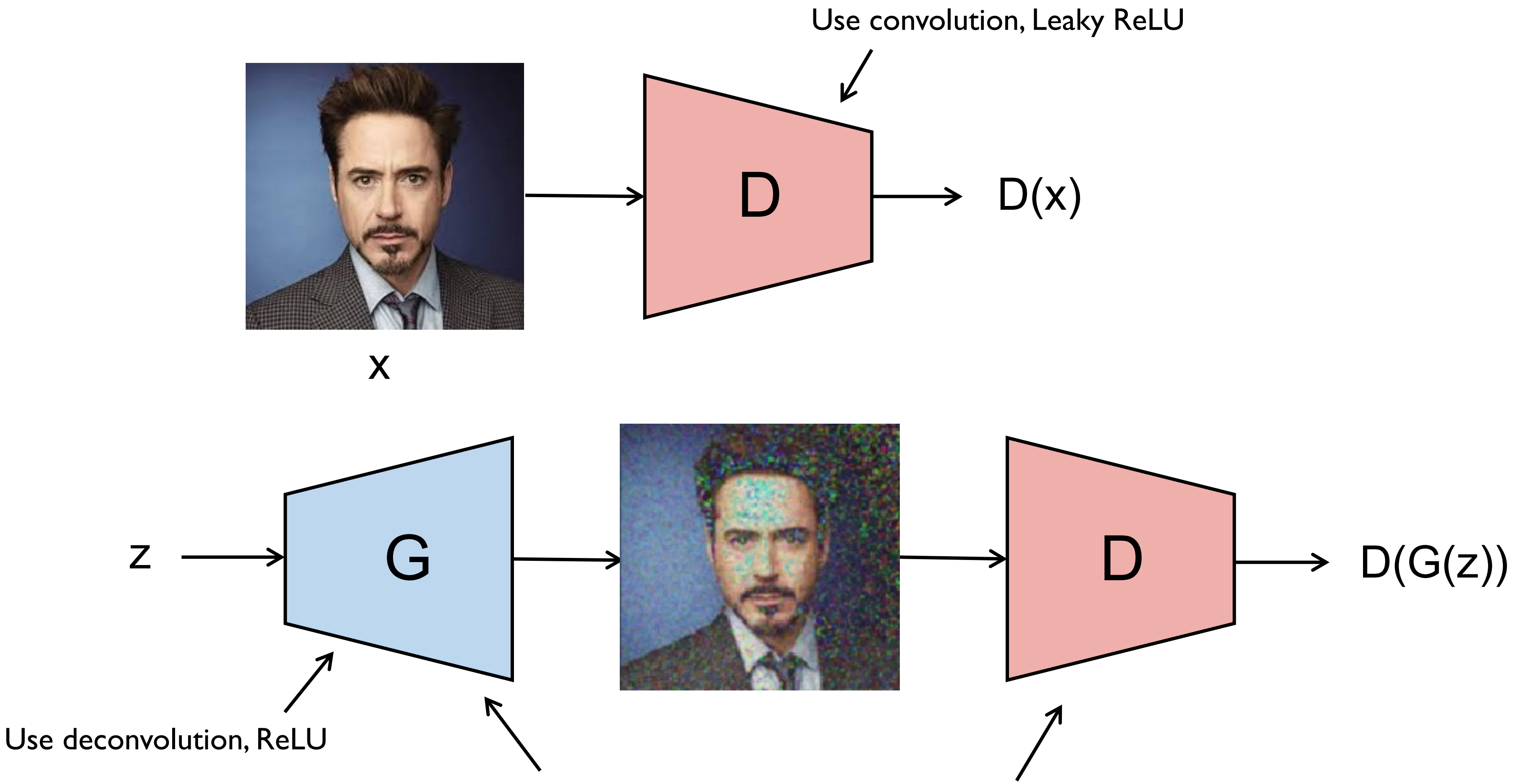
- Deep Convolutional GAN(DCGAN), 2015

  The authors present a model that is still highly preferred.



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

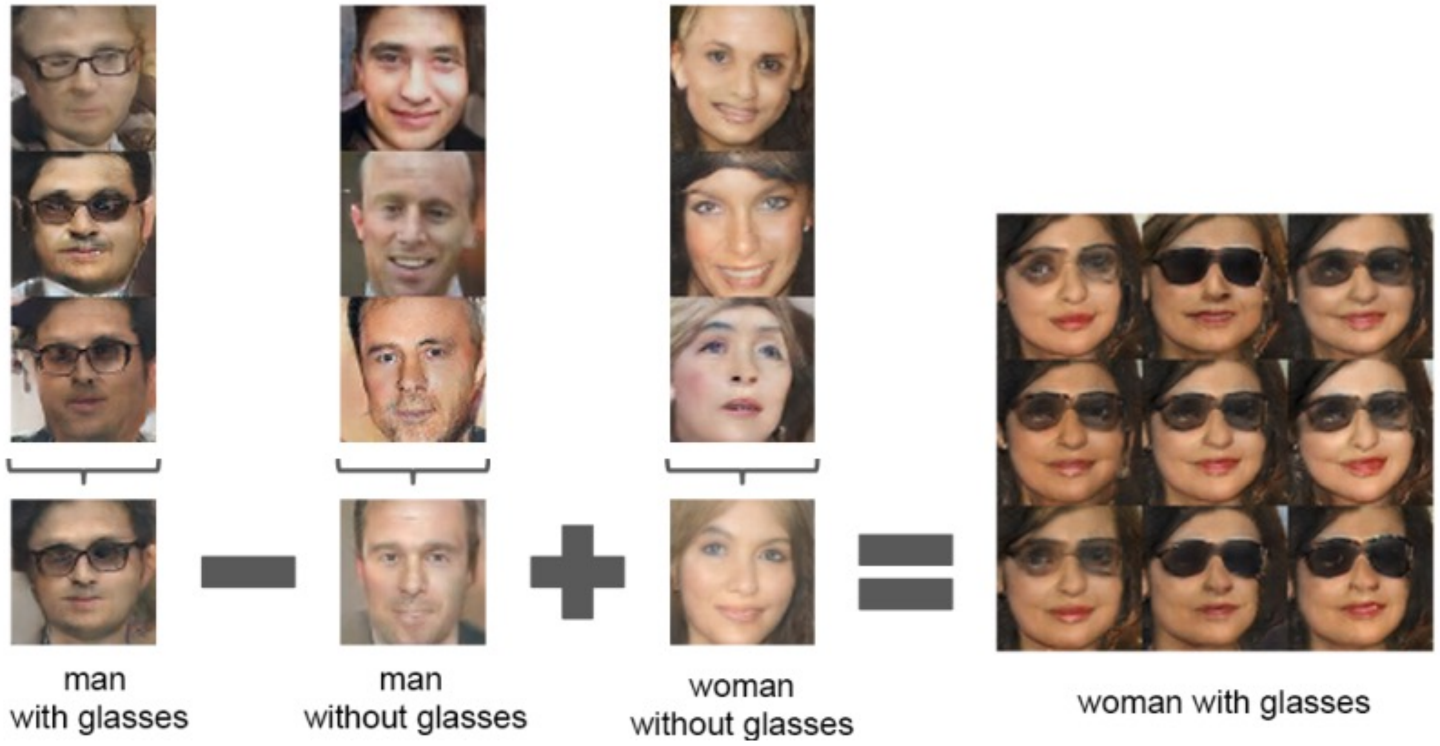
DCGAN



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

DCGAN

- Latent vector arithmetic

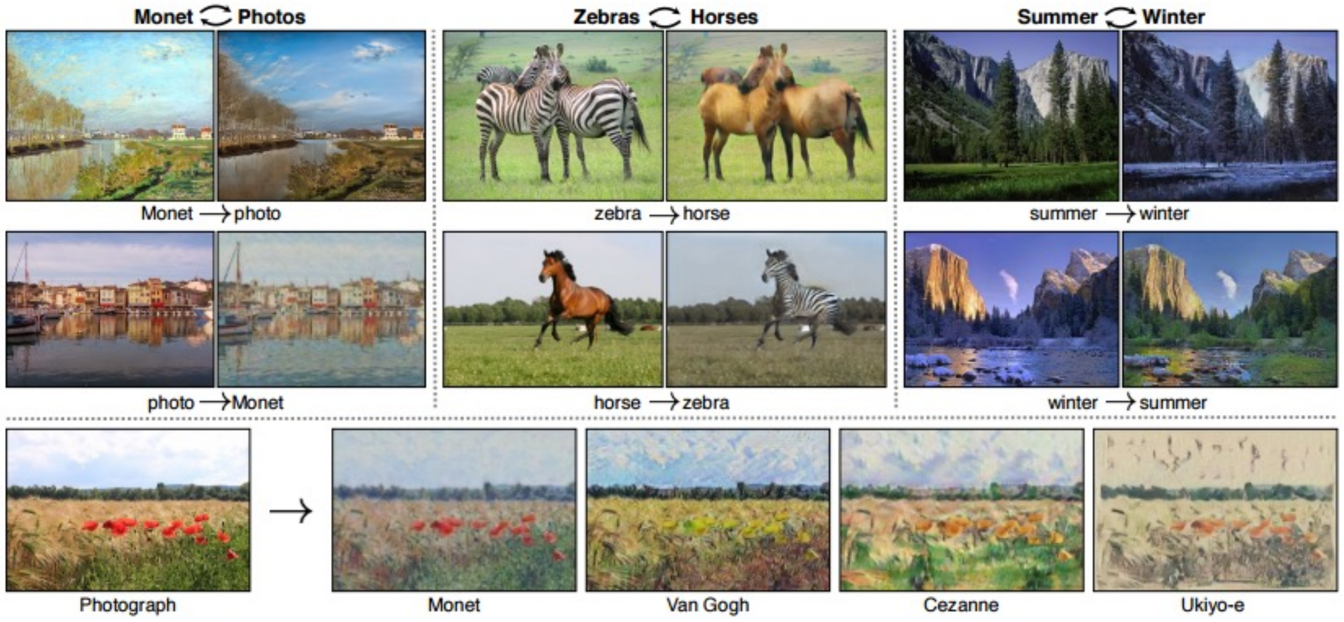


Radford et al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015

04
Extensions of GAN

- CycleGAN: Unpaired Image-to-Image Translation

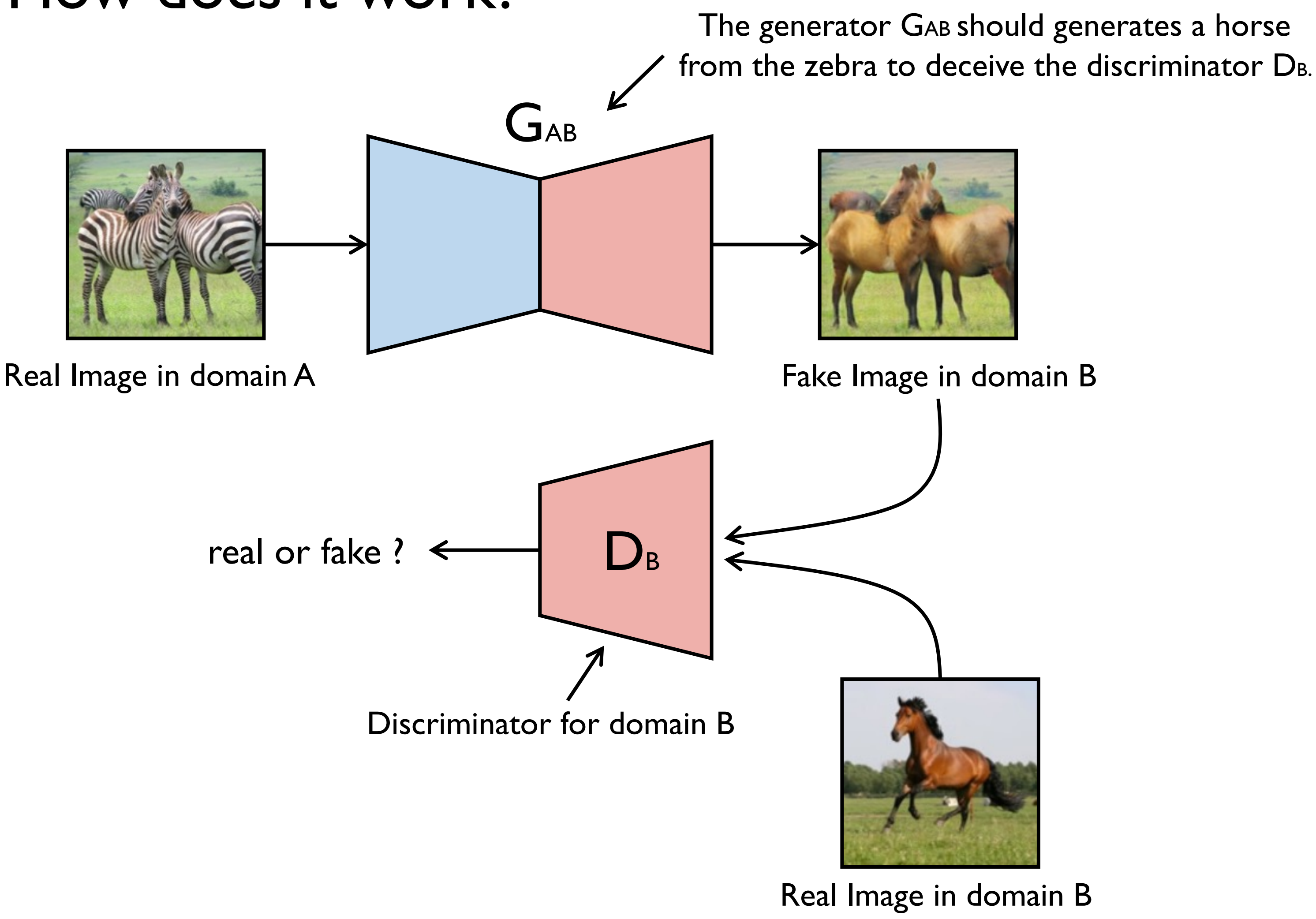
  presents a GAN model that transfer an image from a source domain A to a target domain B in the absence of paired examples.



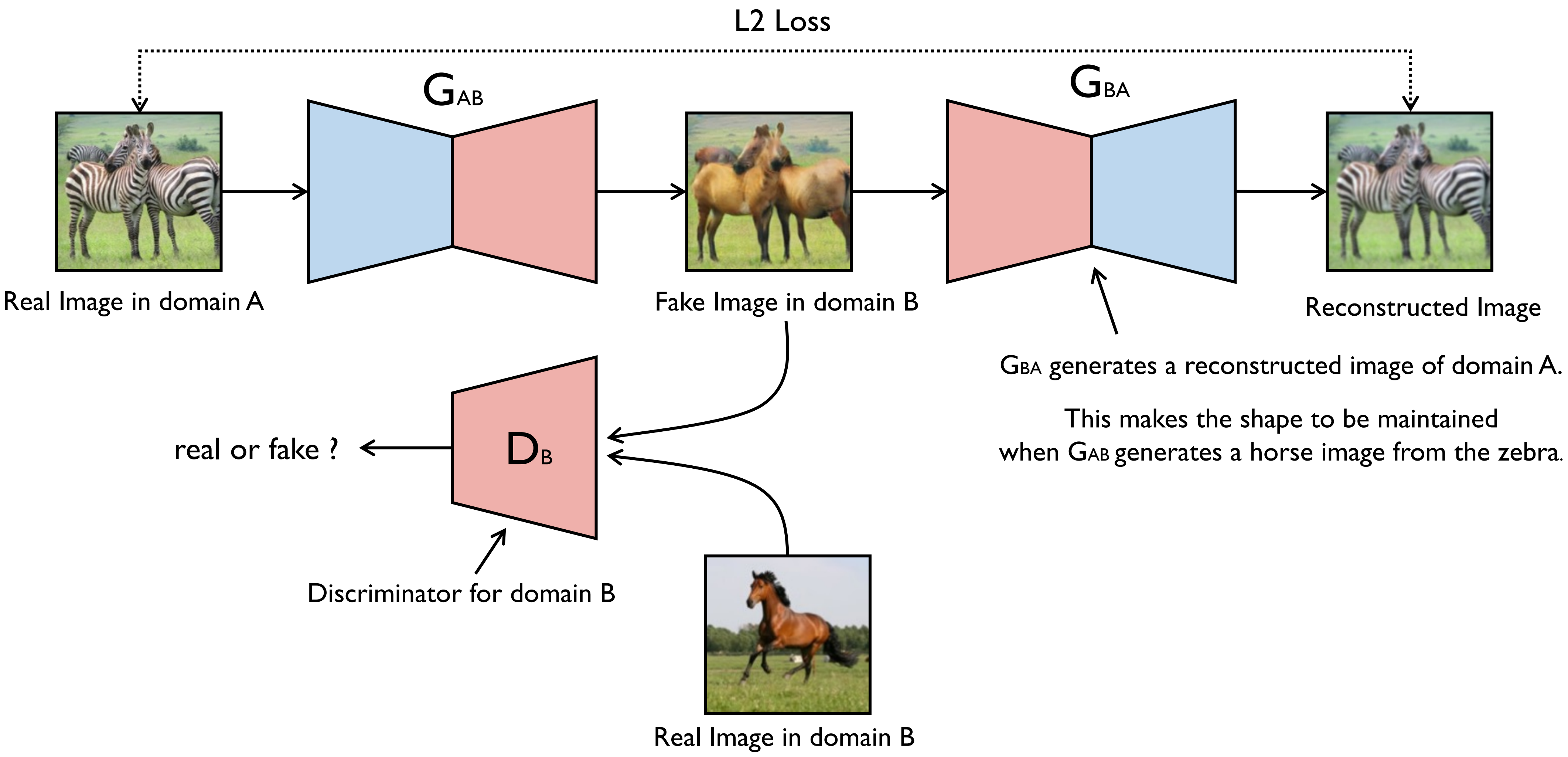
Jun-Yan Zhu et al. Unpaired Image-to-Image Translation using Cycle Consistent Adversarial Networks, 2017

CycleGAN

- How does it work?

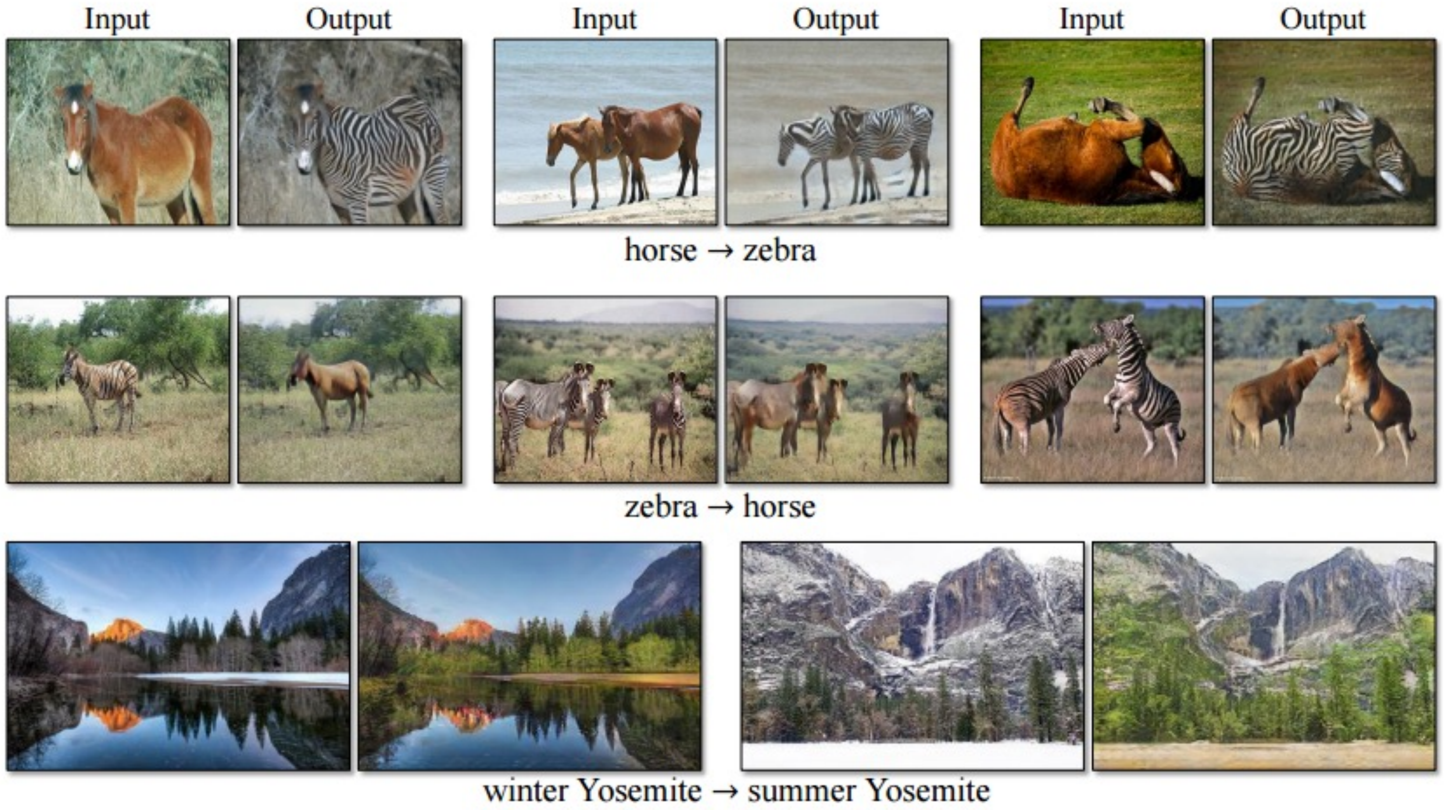


- How does it work?



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.

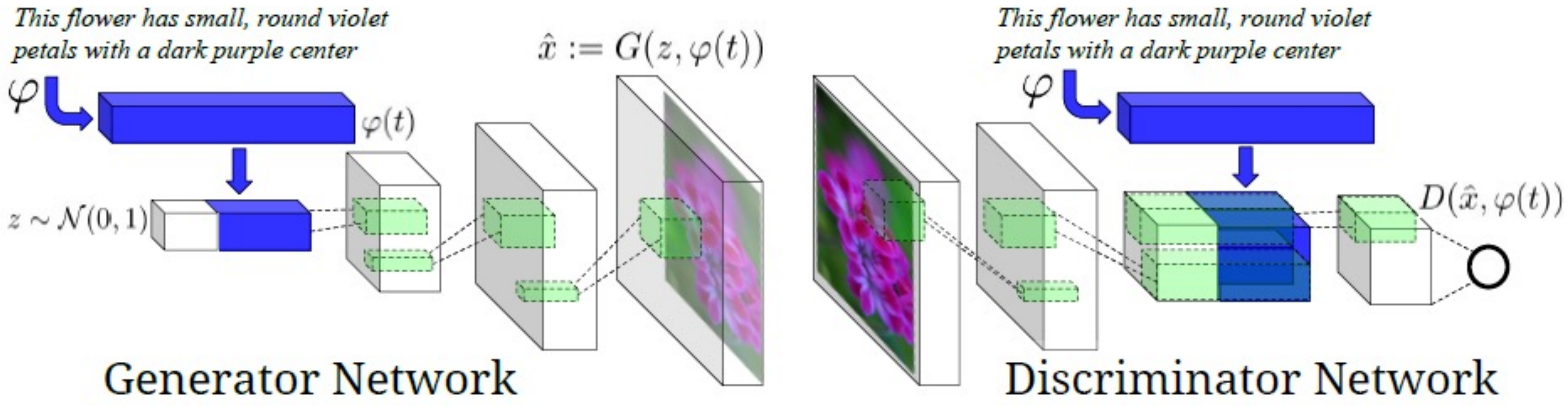
SVHN-to-MNIST

MNIST-to-SVHN

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

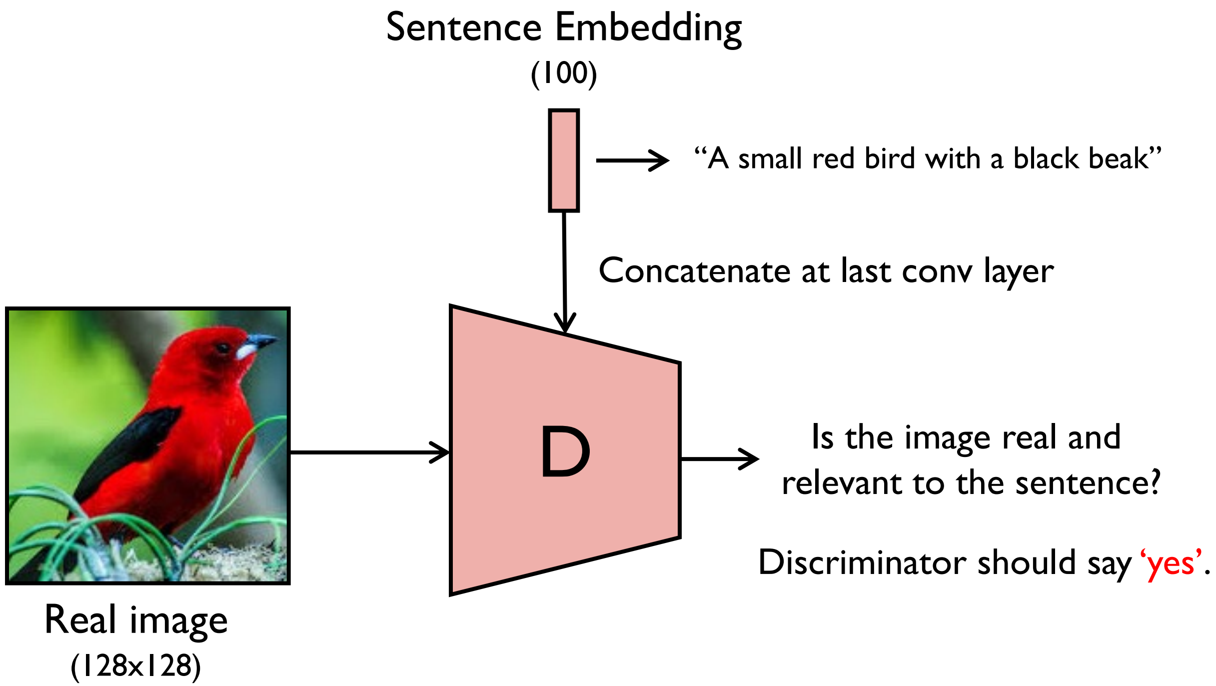
- Generative Adversarial Text to Image Synthesis, 2016

  presents a novel model architecture that generates an image from the text.



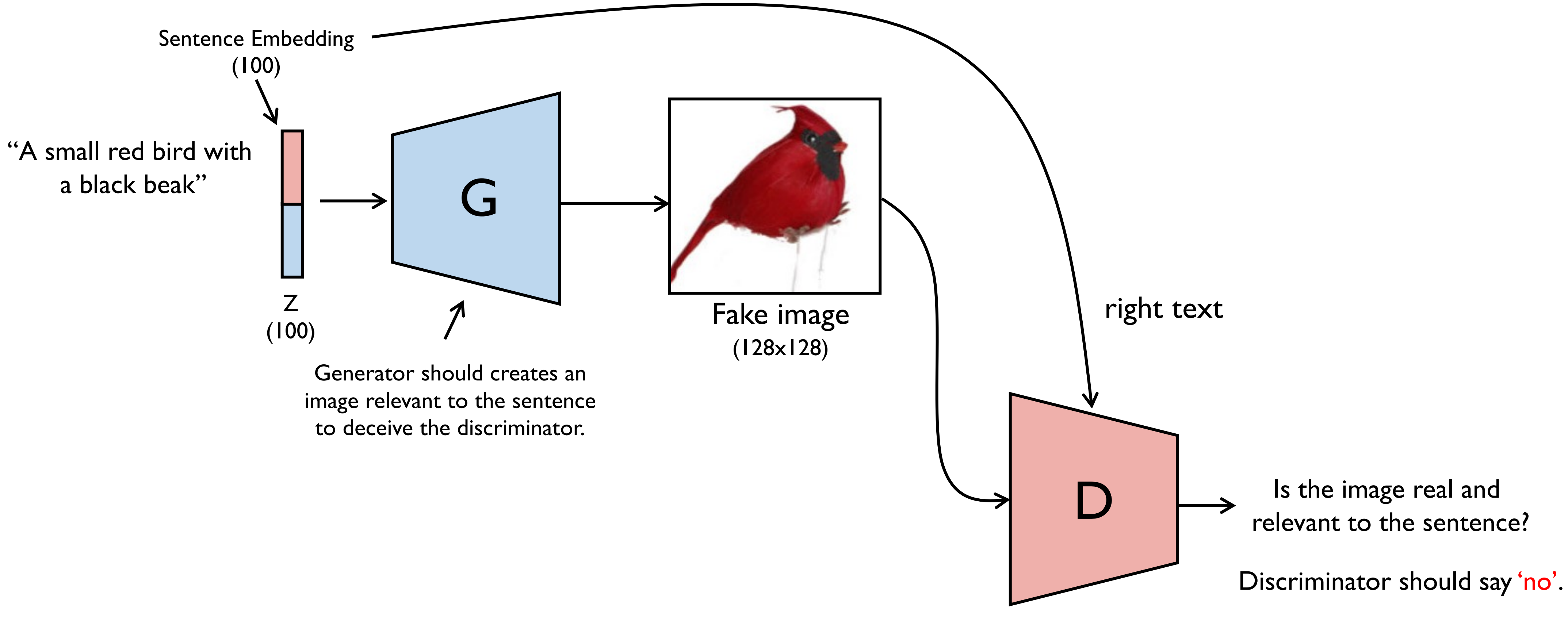
Scott Reed et al. Generative Adversarial Text to Image Synthesis, 2016

- Training with (real image, right text)



Text2Image

- Training with (fake image, right text)

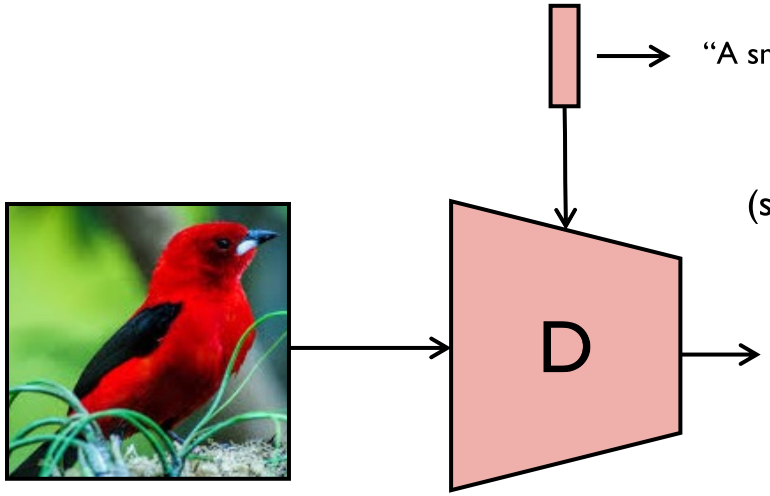


Text2Image

Extensions

- Training with (real image, wrong text)

Sentence Embedding
(100)



"A small yellow bird with a brown beak"

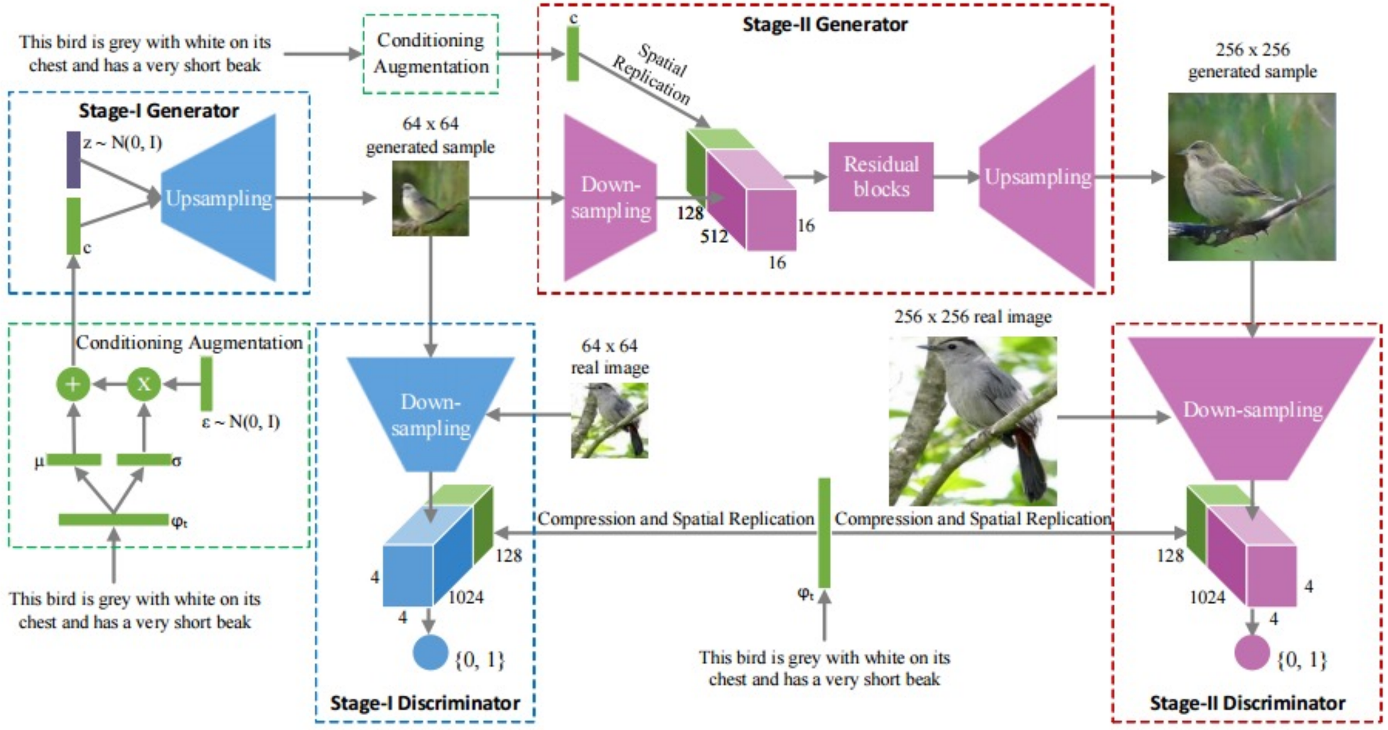
wrong text
(sampled randomly from the training data)

Is the image real and
relevant to the sentence?

Discriminator should say 'no'.

Real image
(128x128)

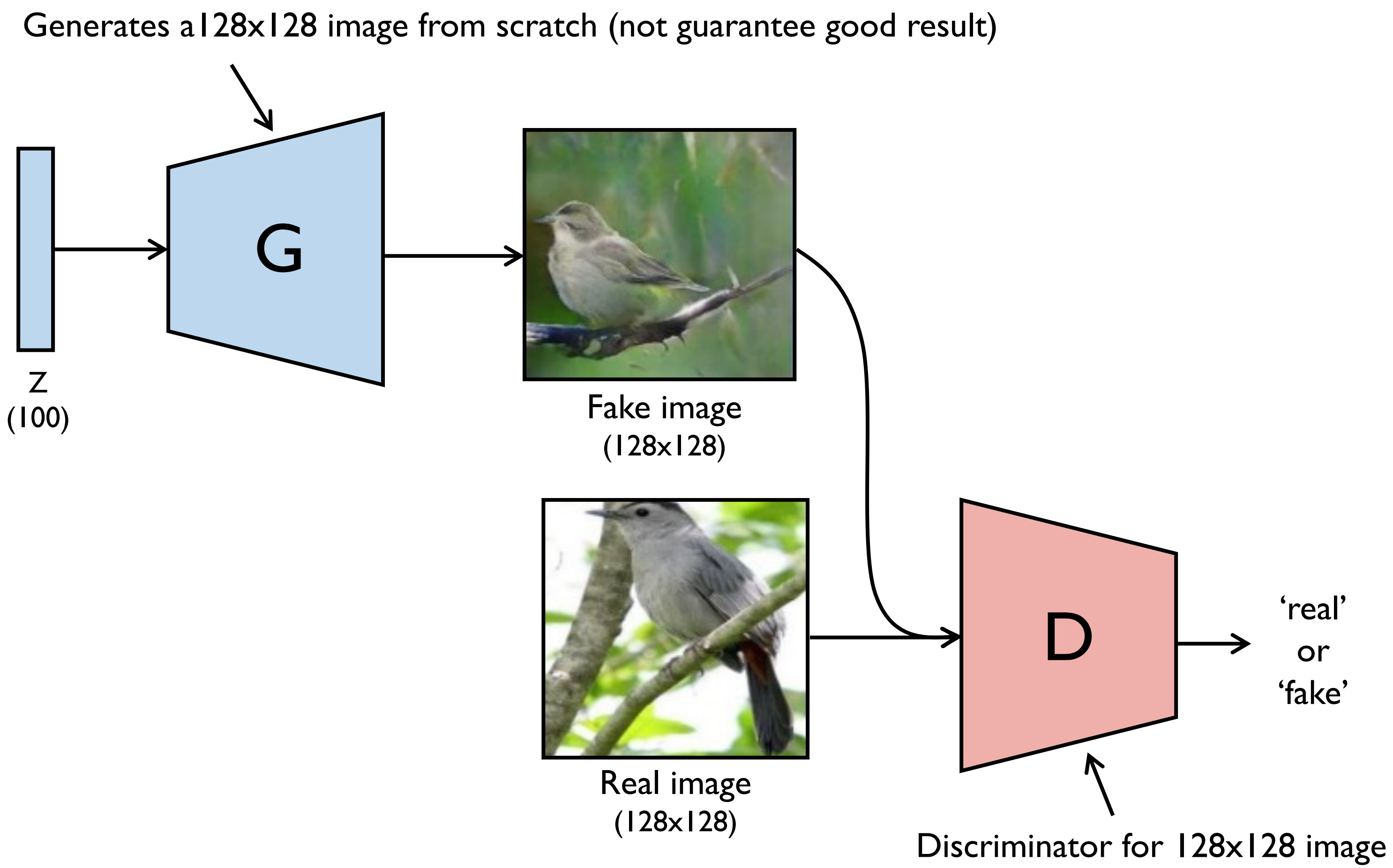
- StackGAN: Text to Photo-realistic Image Synthesis



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

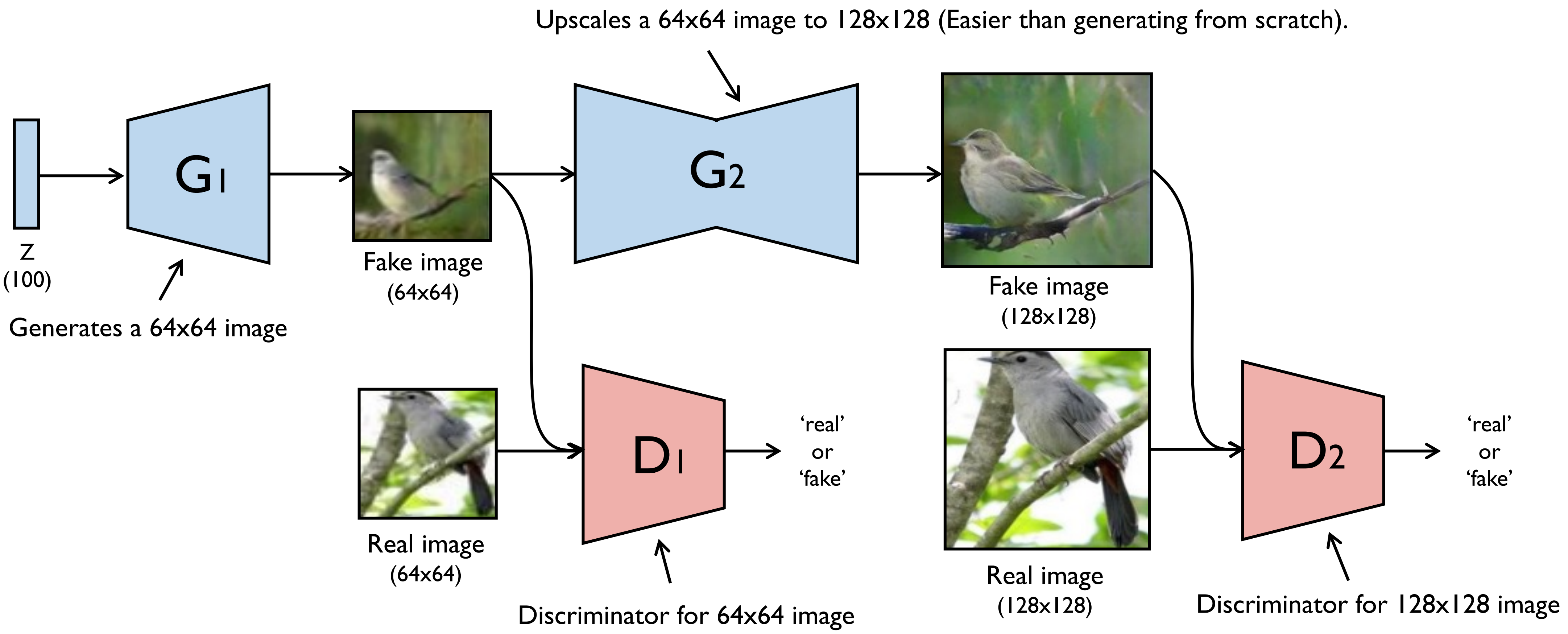
StackGAN

- Generating 128x128 from scratch



StackGAN

- Generating 128x128 from 64x64



Thank you