

Generative Adversarial Networks

Jaegul Choo (주재걸)

Korea University

<https://sites.google.com/site/jaegulchoo/>

Slides made by my student, Yunjey Choi

<https://github.com/yunjey>



- Namju Kim. Generative Adversarial Networks (GAN)

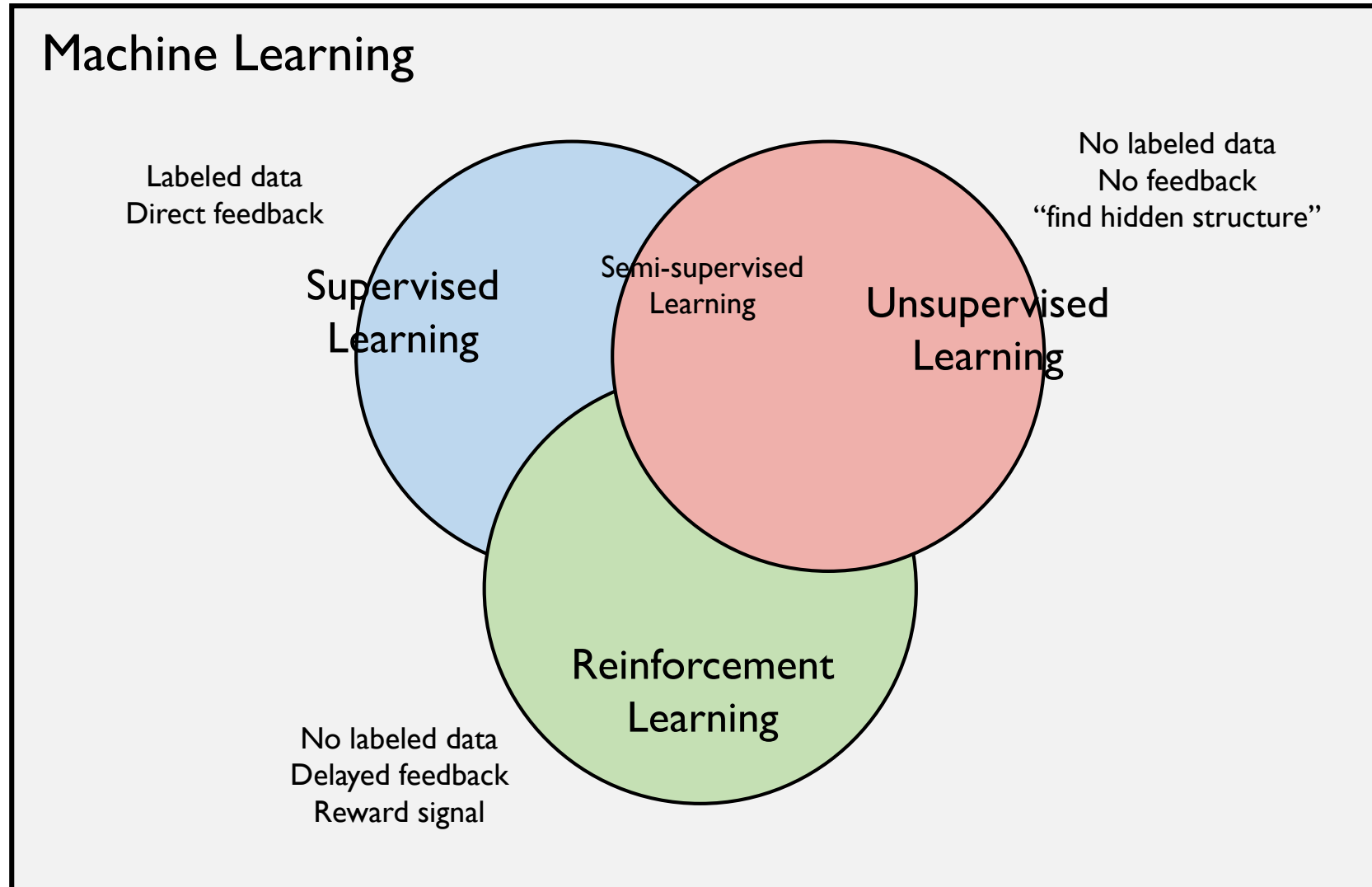
<https://www.slideshare.net/ssuser77ee21/generative-adversarial-networks-70896091>

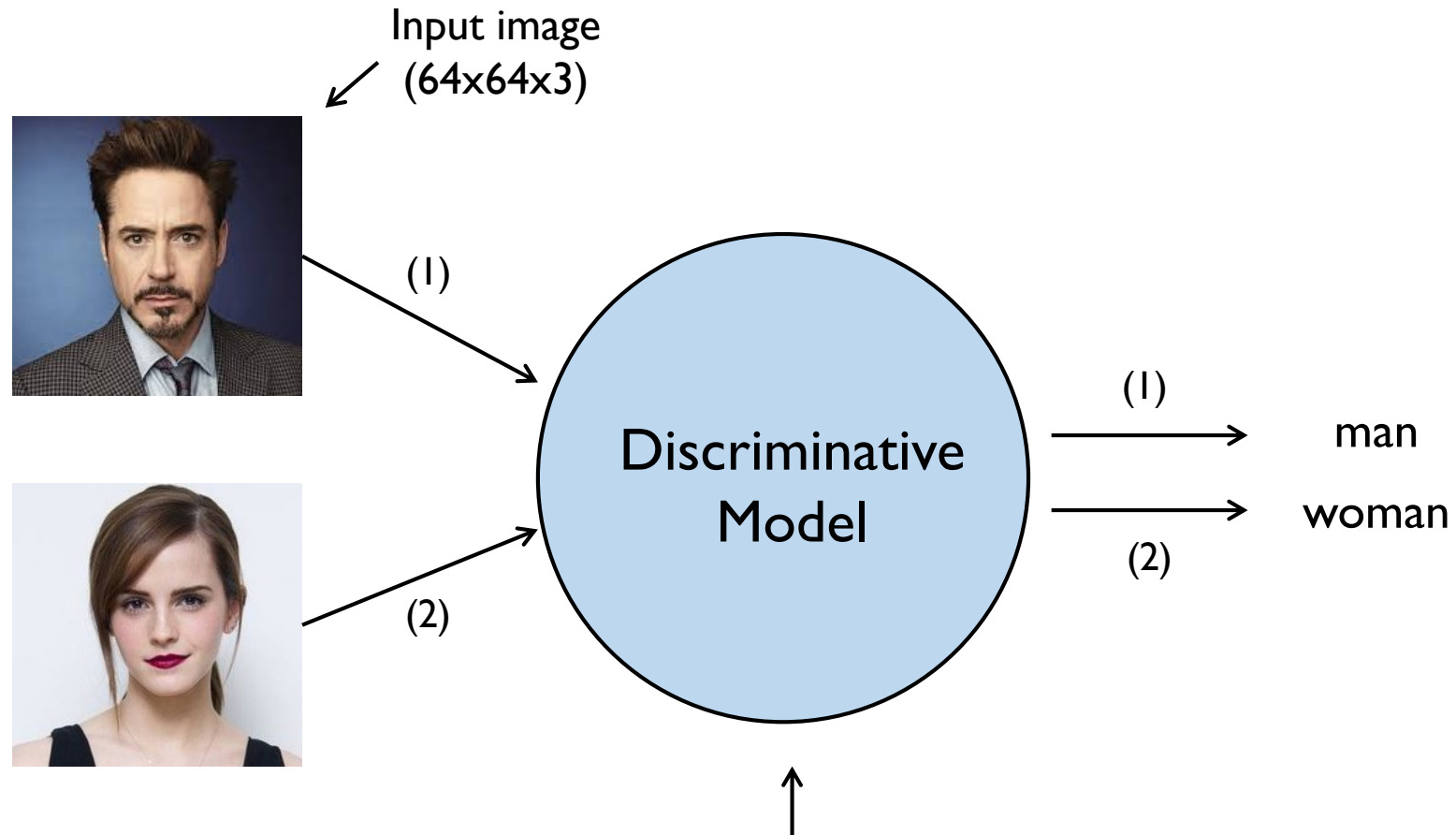
- Taehoon Kim. 지적 대화를 위한 깊고 넓은 딥러닝

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

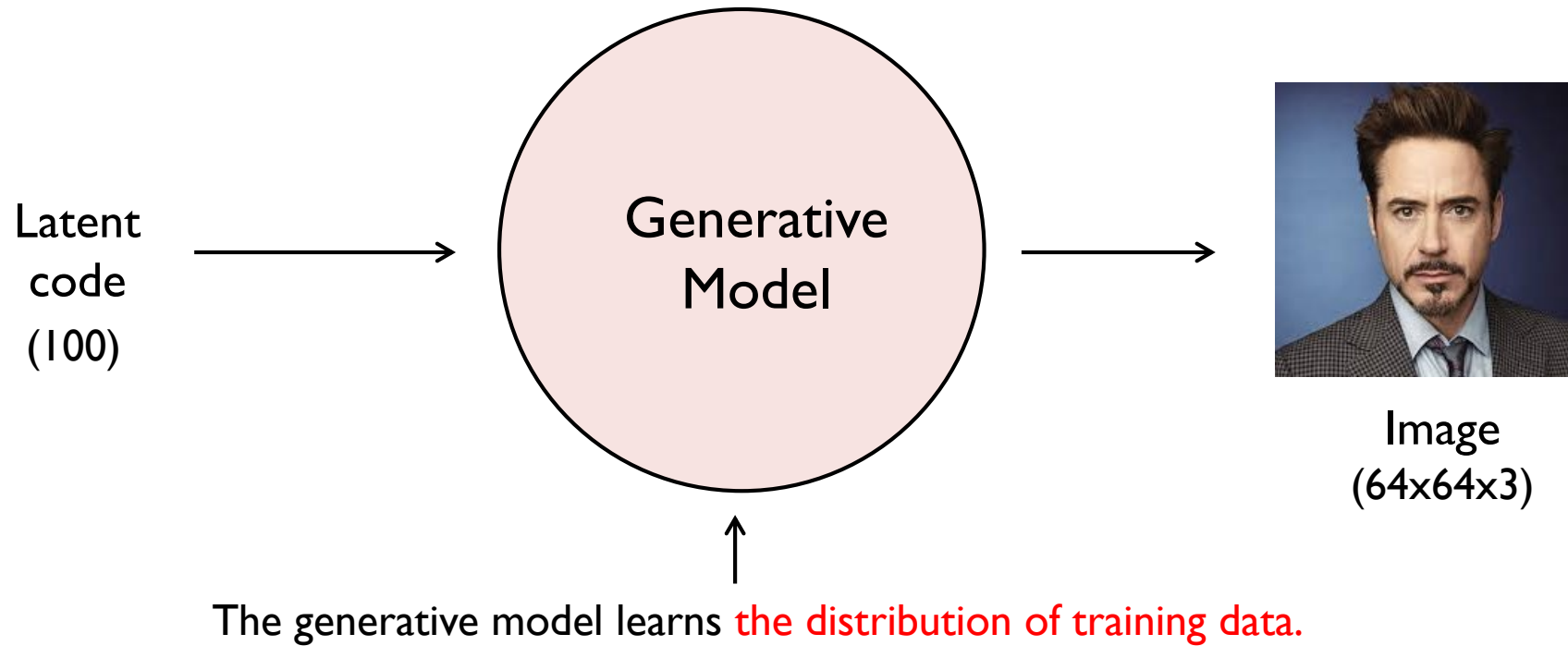
01 Introduction







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



Probability Distribution



Introduction



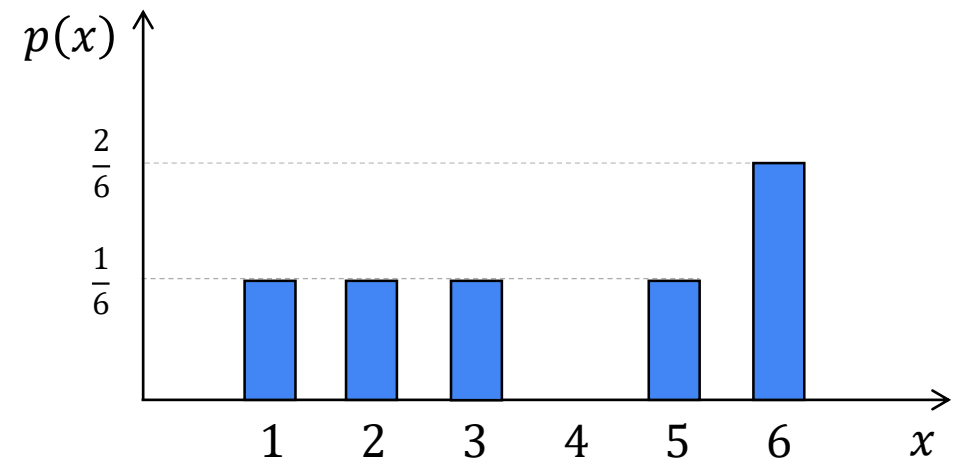
Probability Basics (Review)



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





What if x is actual images in the training data?

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

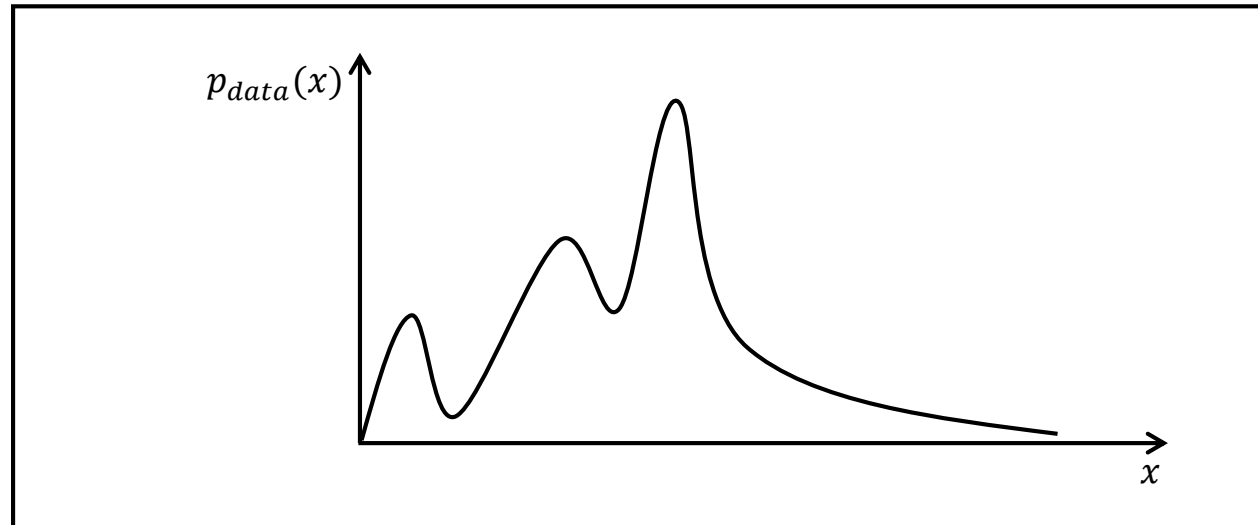




Probability density function



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



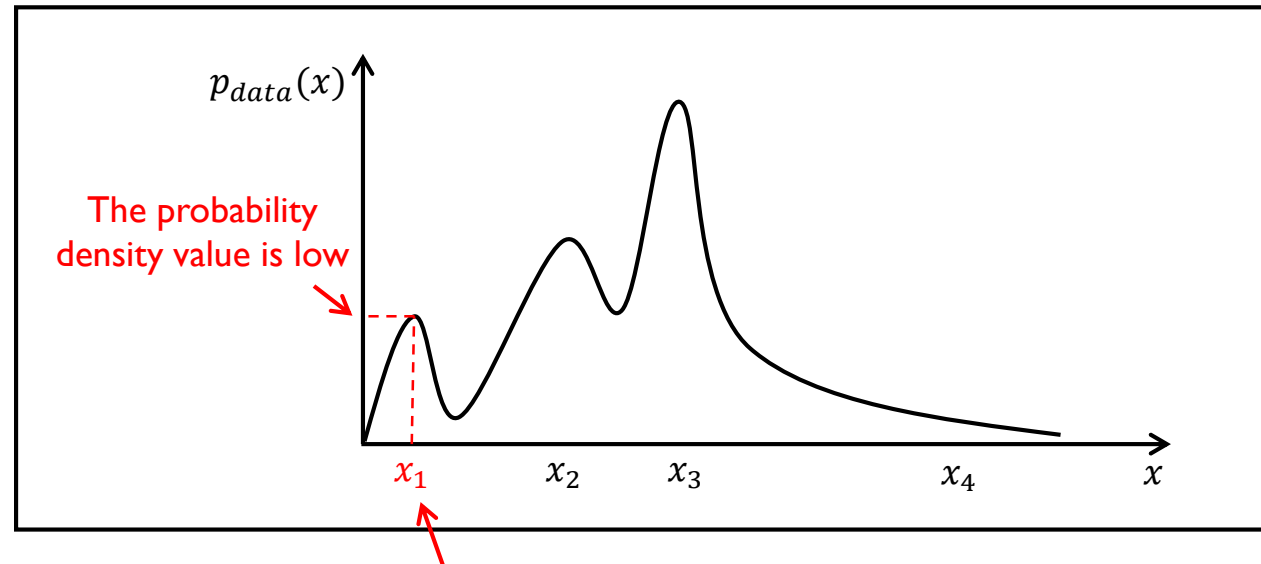
Probability Distribution



Introduction



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

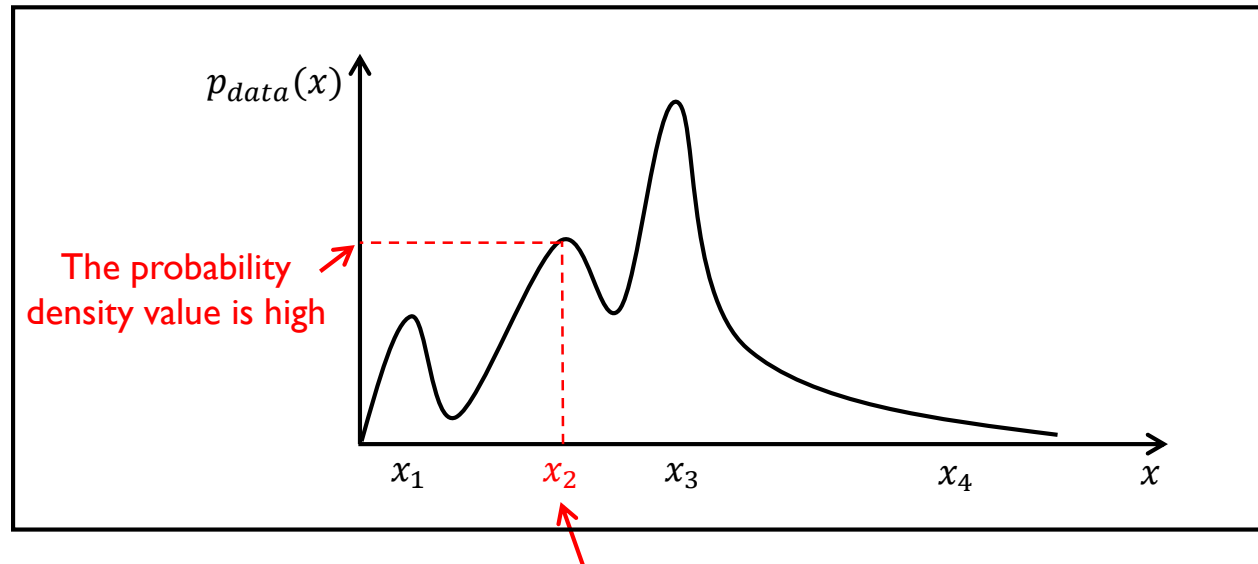


x_1 is a 64x64x3 high dimensional vector
representing **a man with glasses**.





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

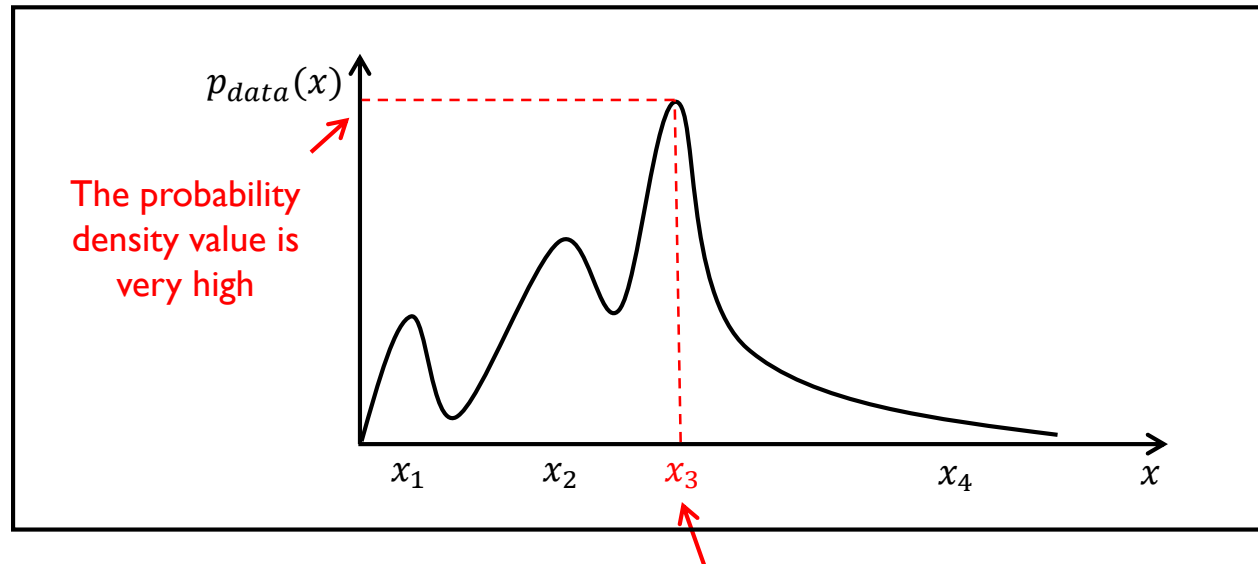


x_2 is a 64x64x3 high dimensional vector representing **a woman with black hair**.





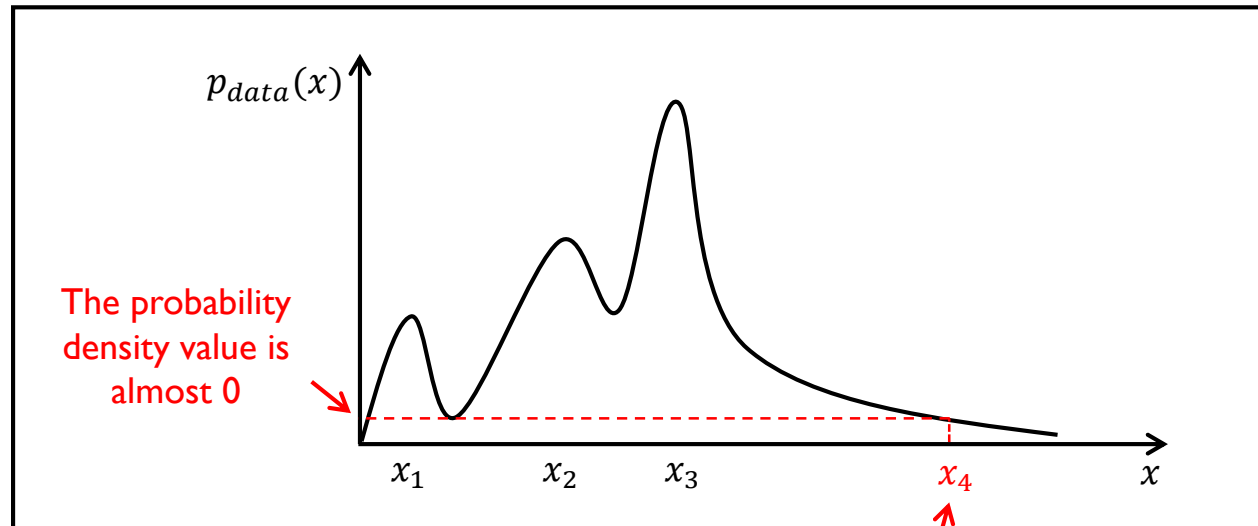
Our dataset may contain very many images of **women with blonde hair**.



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



Our dataset may not contain **these strange images**.



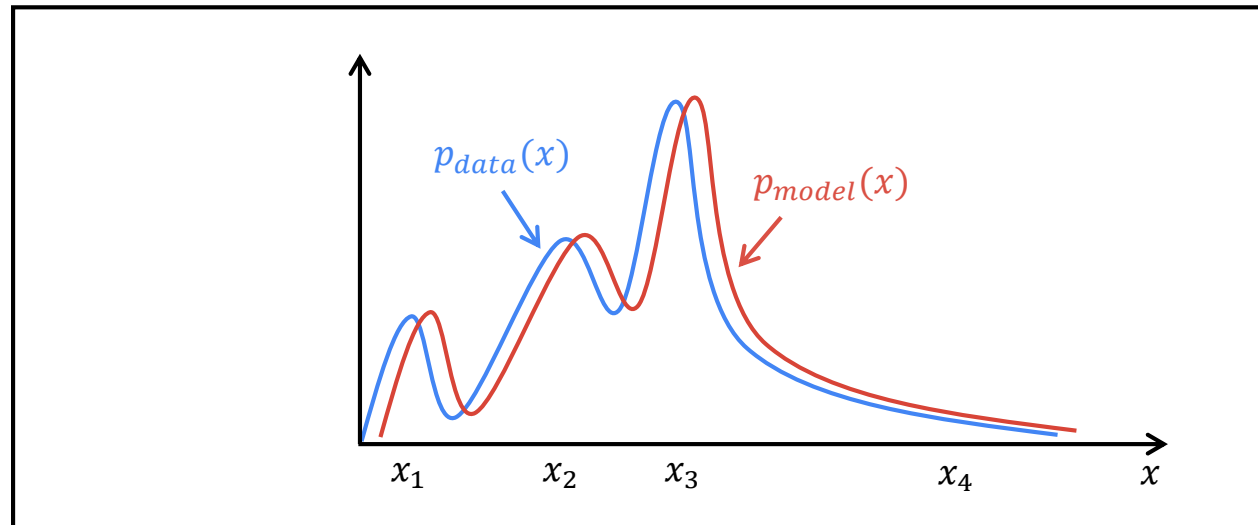
x_4 is an 64x64x3 high dimensional vector representing **very strange images**.



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

Distribution of images generated by the model

Distribution of actual images



02

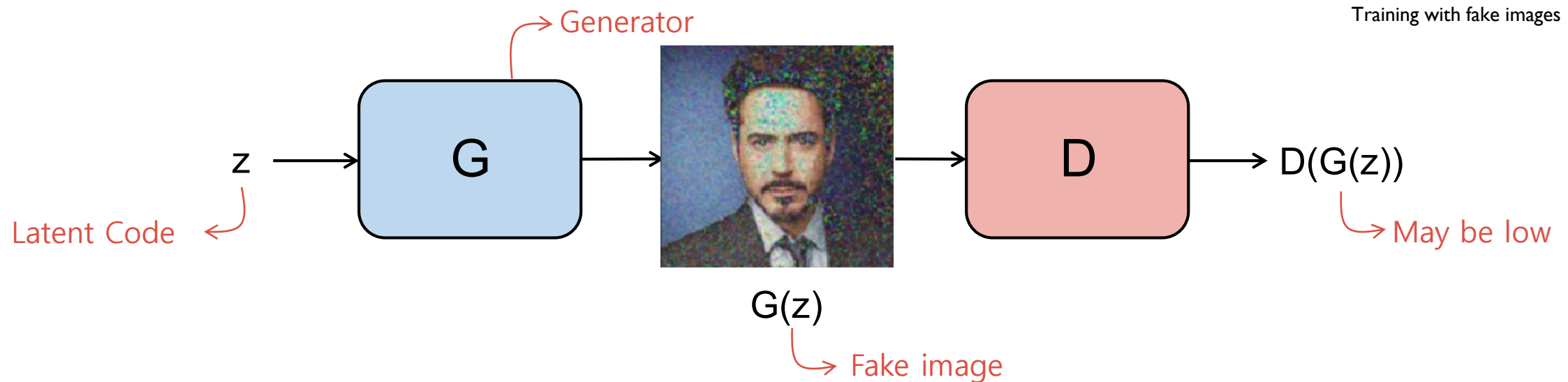
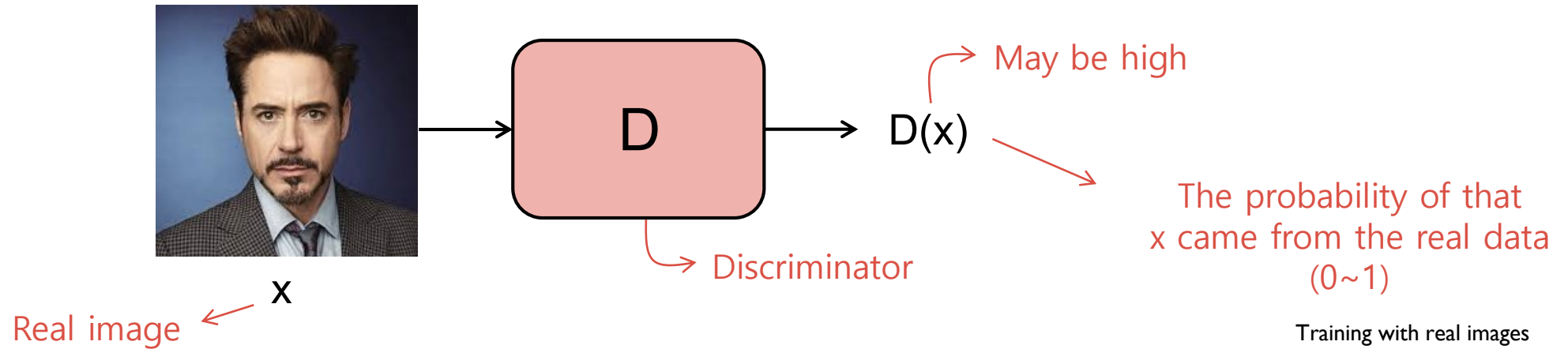
Generative Adversarial Networks



Intuition in GAN



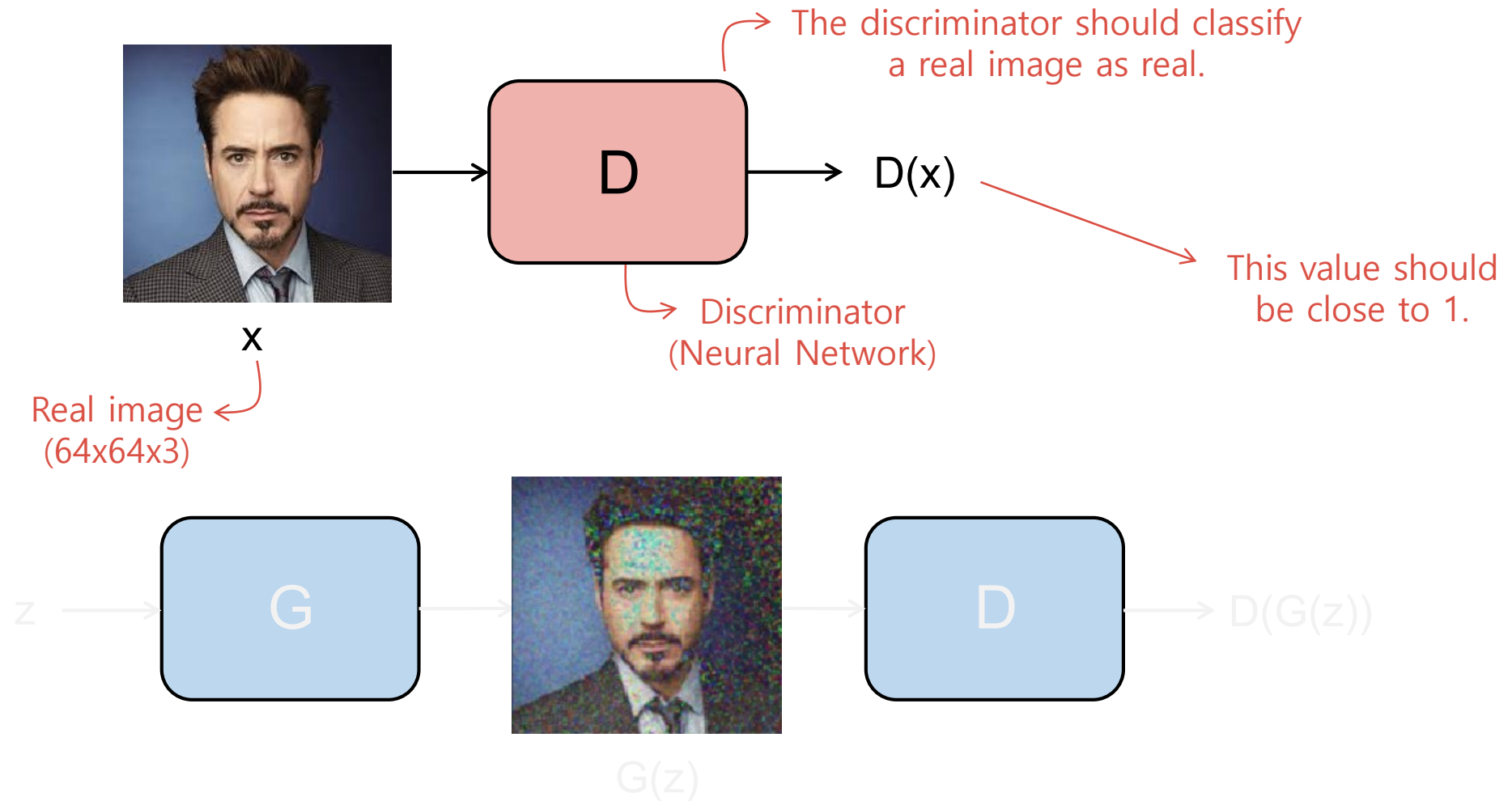
GANs



Intuition in GAN



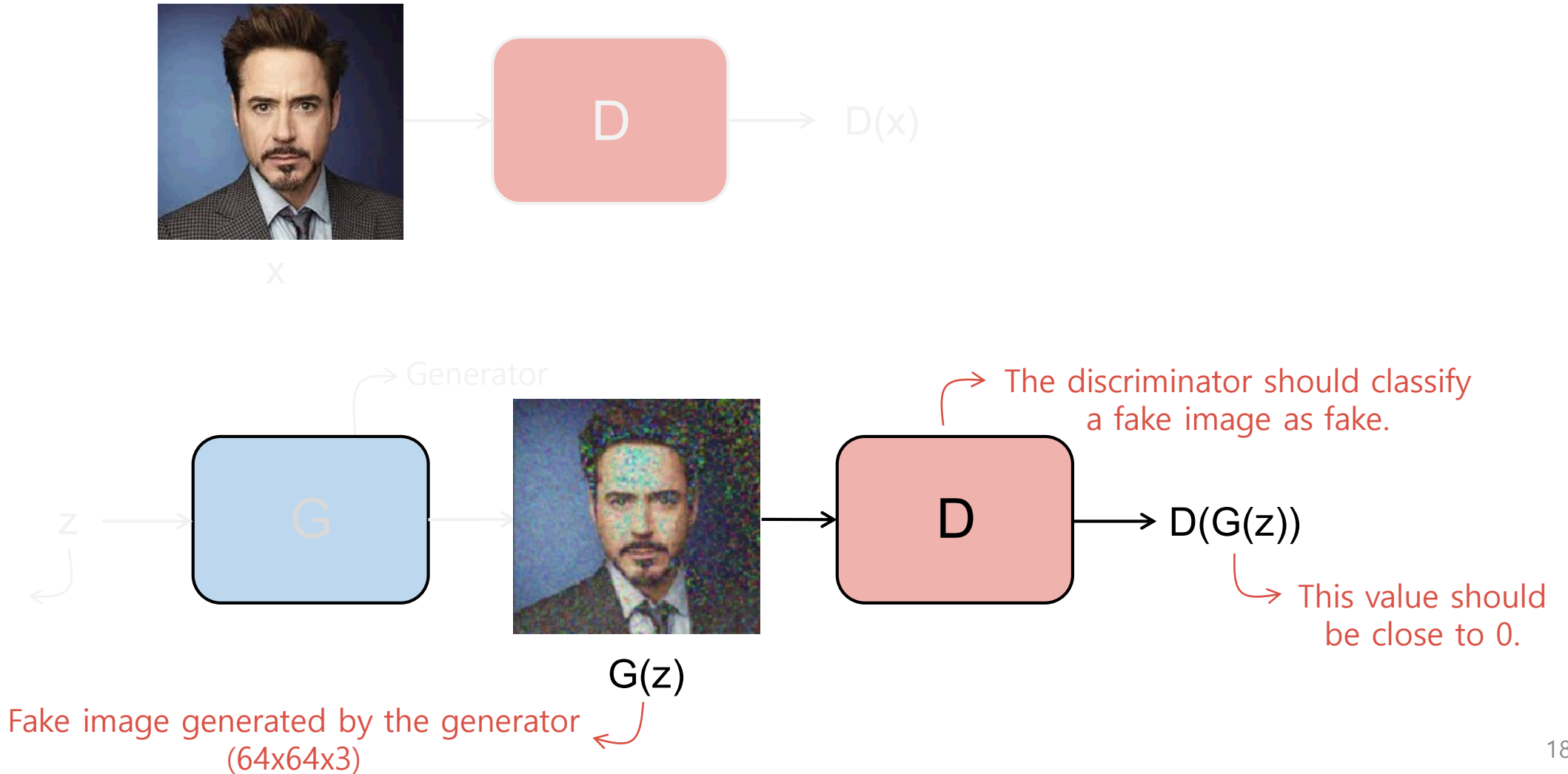
GANs



Intuition in GAN



GANs



Intuition in GAN



GANs



x

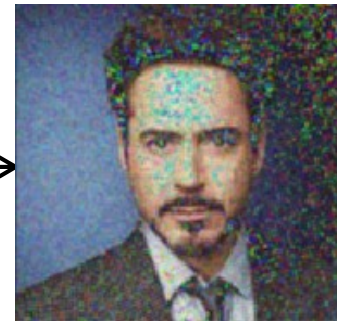


$D(x)$

Generator
(Neural Network)

z

G



$G(z)$

D

$D(G(z))$

Latent Code
(100)

The generator should create an image
that is indistinguishable from real to
deceive the discriminator

Generated image
(64x64x3)

This value should
be close to 1.

Objective Function of GAN



GANs



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 $V(D, G)$

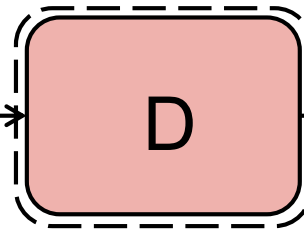
Maximum when $D(x) = 1$

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

Objective function



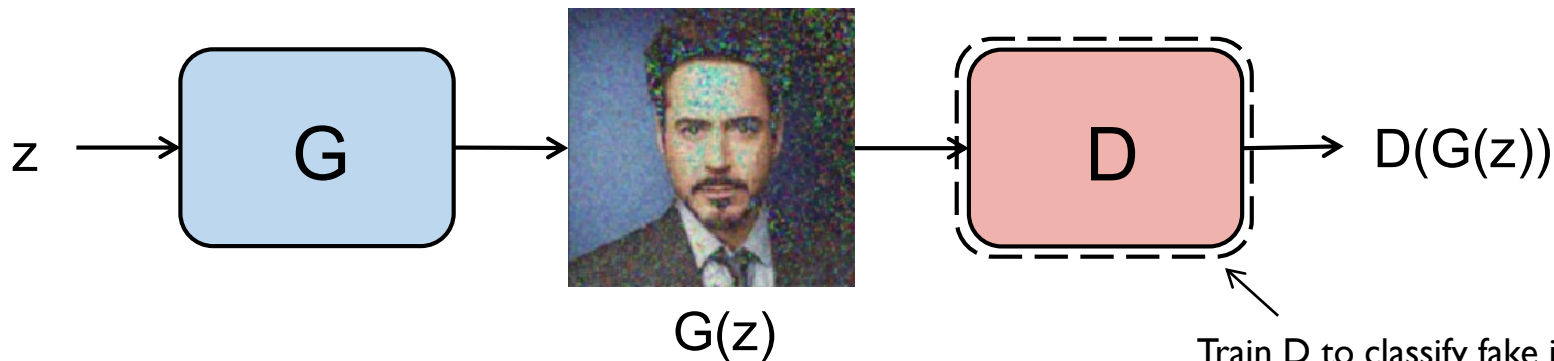
x



$D(x)$

Train D to classify real images as real

Training with real images



Objective Function of GAN



GANs



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

G is independent of this part

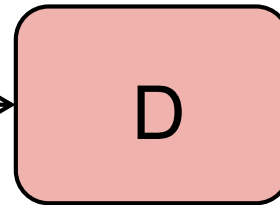
G should minimize $V(D, G)$

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

Objective function

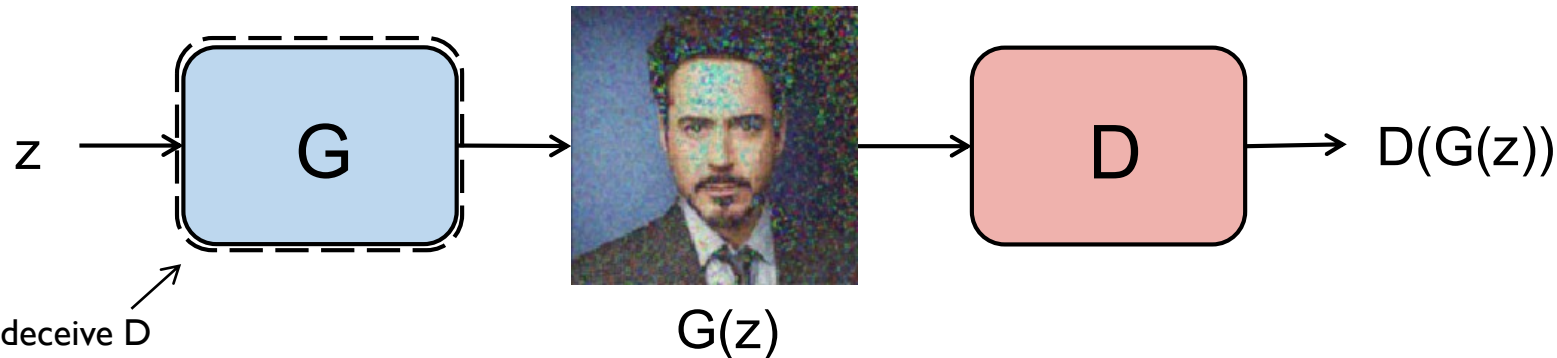


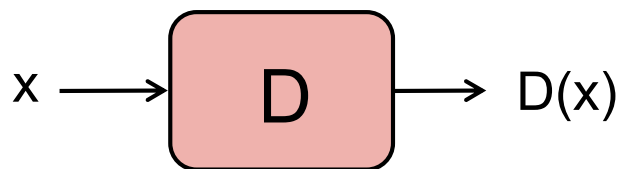
x



$D(x)$

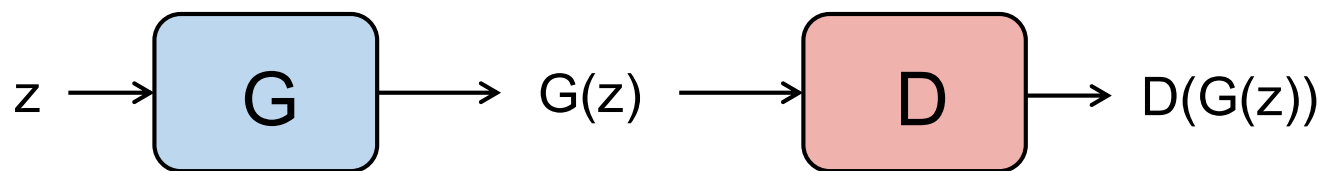
Training with real images





Training with real images

Training with fake images



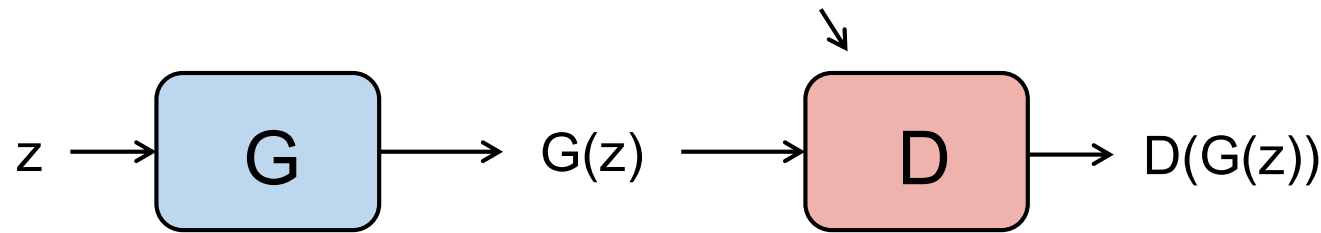
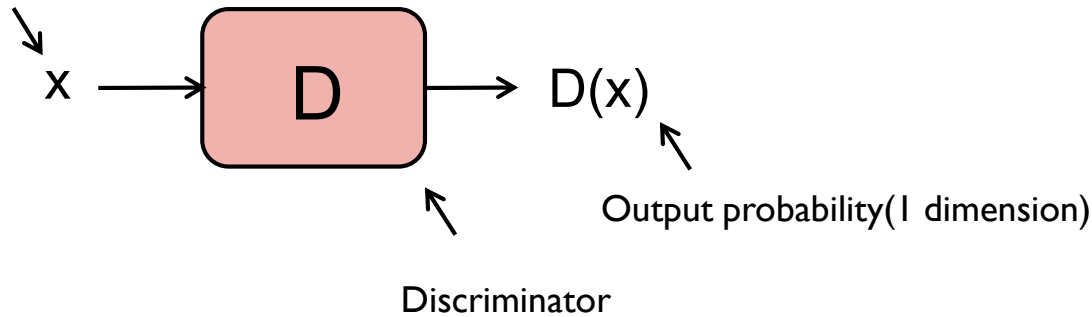
```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```



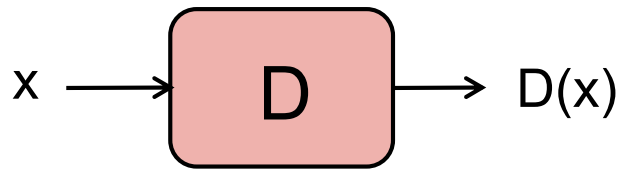
Define the discriminator

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

Assume x is MNIST (784 dimension)

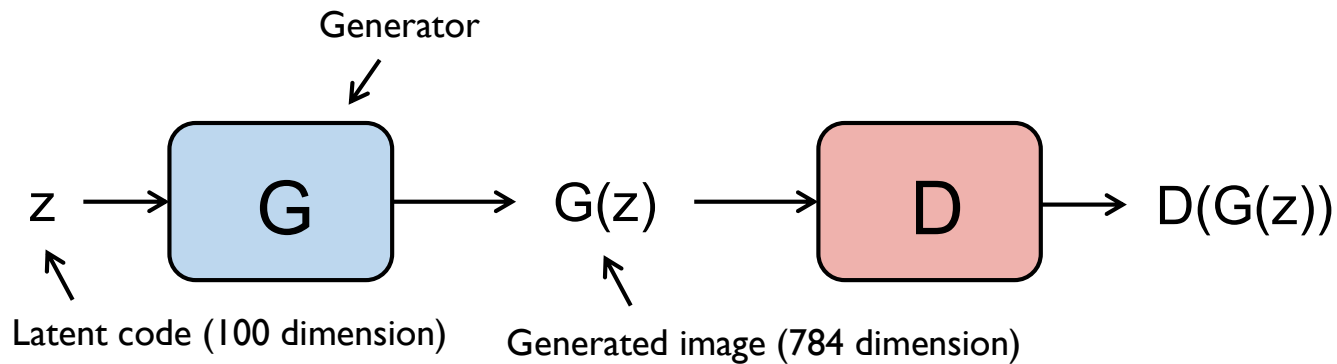


```
1 import torch
2 import torch.nn as nn
3
4
5
6
7
8
9
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```

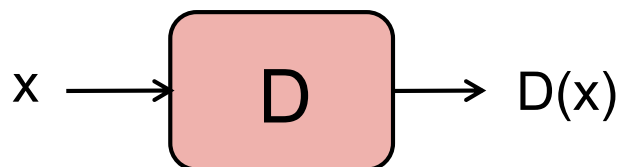


Define the generator

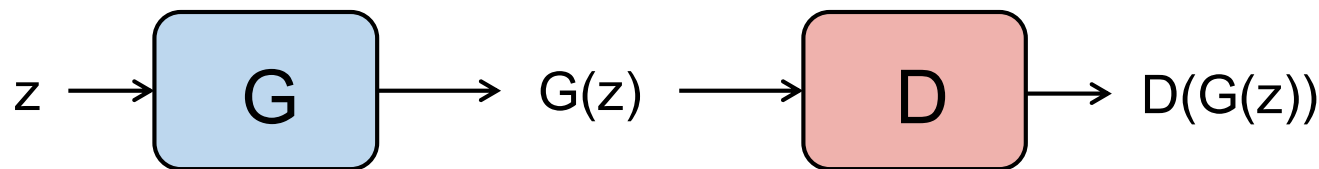
input size: 100
hidden size: 128
output size: 784



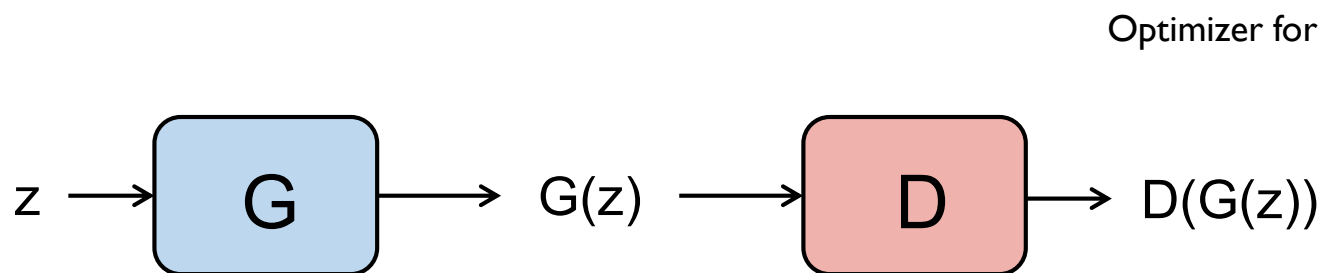
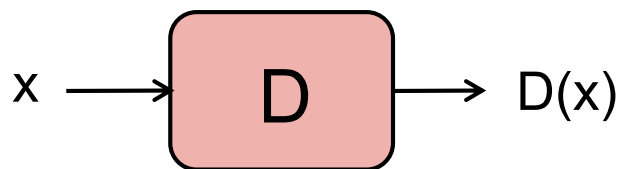
```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11
12
13
14
15
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```

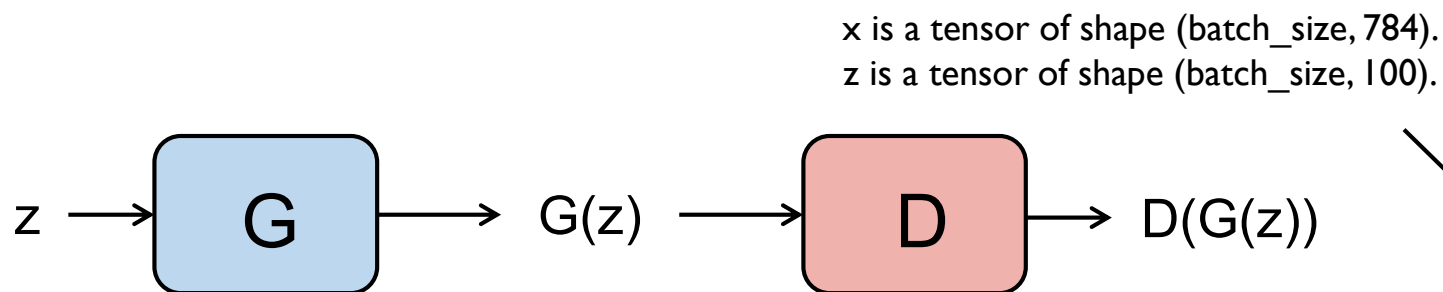
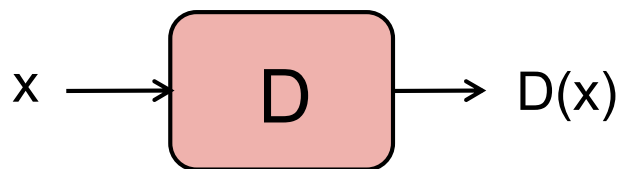
Binary Cross Entropy Loss $(h(x), y)$
 $-y \log h(x) - (1 - y) \log(1 - h(x))$



```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```



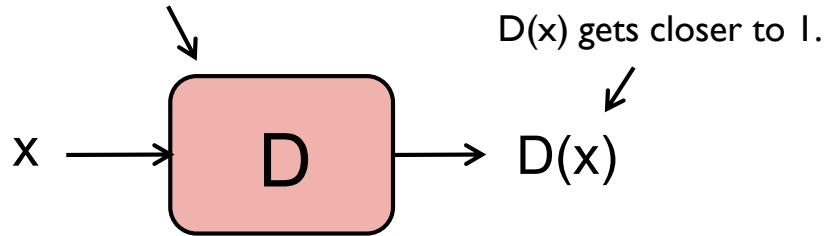
```
1  import torch
2  import torch.nn as nn
3
4
5  D = nn.Sequential(
6      nn.Linear(784, 128),
7      nn.ReLU(),
8      nn.Linear(128, 1),
9      nn.Sigmoid())
10
11  G = nn.Sequential(
12      nn.Linear(100, 128),
13      nn.ReLU(),
14      nn.Linear(128, 784),
15      nn.Tanh())
16
17  criterion = nn.BCELoss()
18
19  # Training loop
20
21  # Assume x be real images of shape (batch_size, 784)
22  # Assume z be random noise of shape (batch_size, 100)
23
24  while True:
25      # train D
26      loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
27      loss.backward()
28      d_optimizer.step()
29
30      # train G
31      loss = criterion(D(G(z)), 1)
32      loss.backward()
33      g_optimizer.step()
```



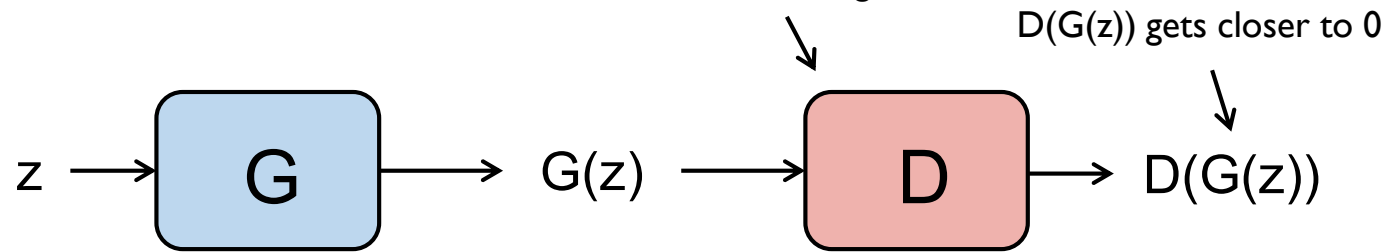
```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22
23
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```



Train the discriminator
with real images

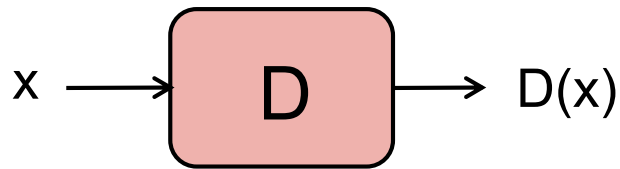


Train the discriminator
with fake images

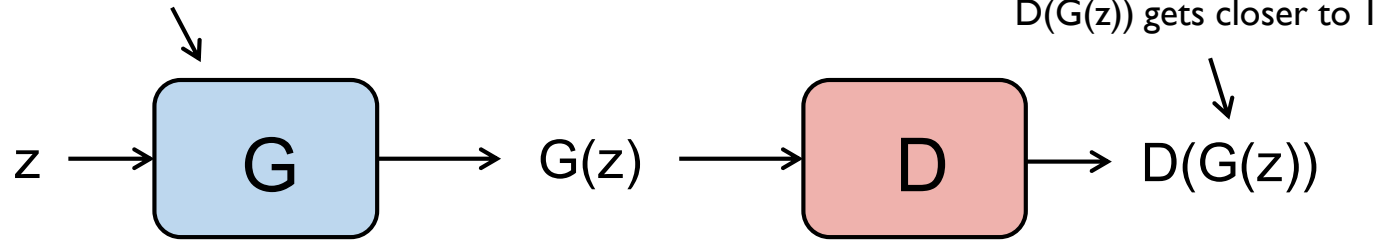


Forward, Backward and Gradient Descent

```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27
28
29
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```

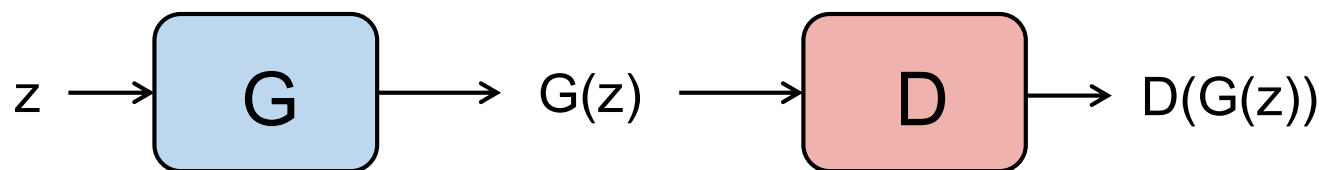
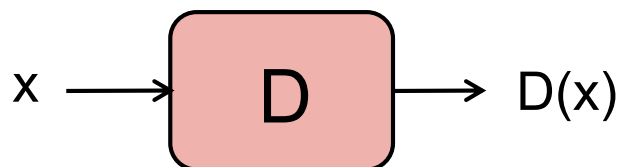


Train the generator
to deceive the discriminator



```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
```

Forward, Backward and Gradient Descent →



The complete code can be found here

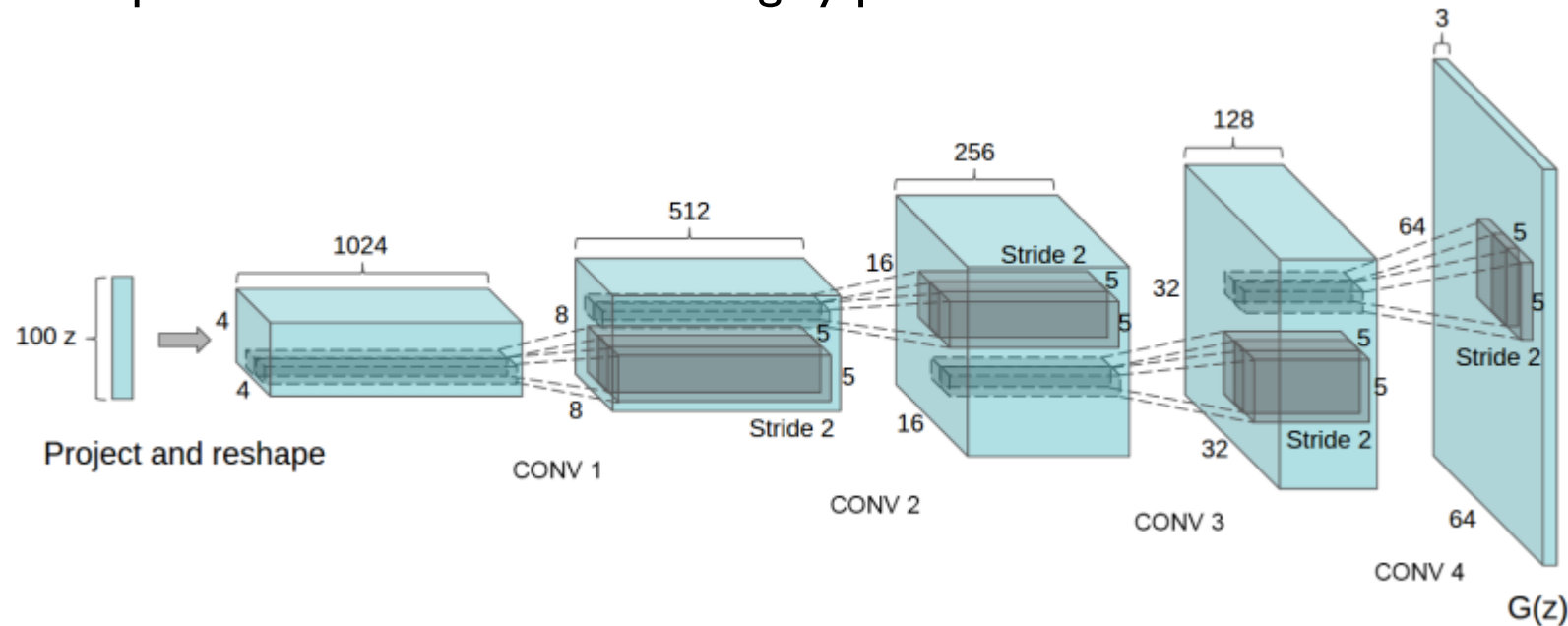
<https://github.com/yunjey/pytorch-tutorial>

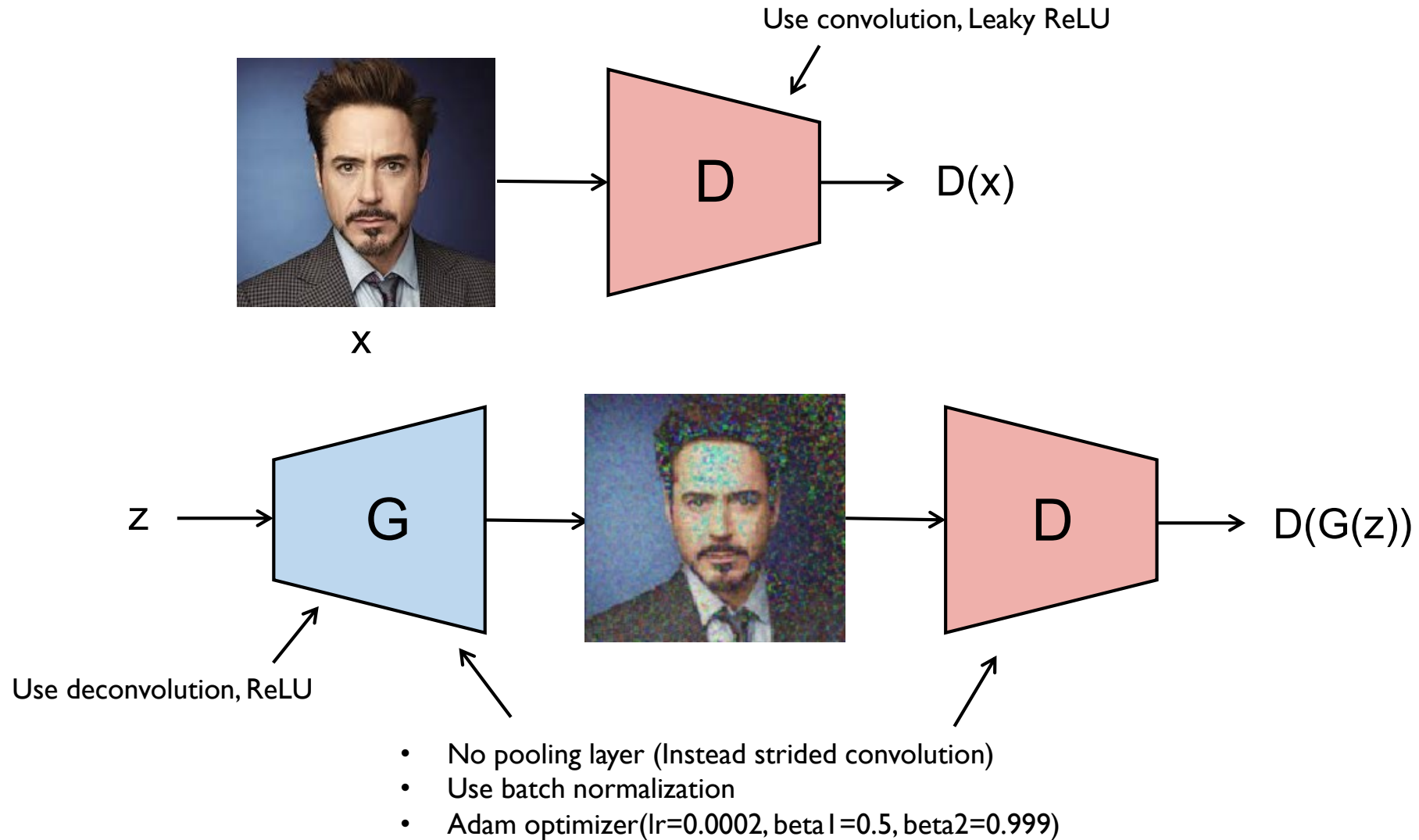
```
1 import torch
2 import torch.nn as nn
3
4
5 D = nn.Sequential(
6     nn.Linear(784, 128),
7     nn.ReLU(),
8     nn.Linear(128, 1),
9     nn.Sigmoid())
10
11 G = nn.Sequential(
12     nn.Linear(100, 128),
13     nn.ReLU(),
14     nn.Linear(128, 784),
15     nn.Tanh())
16
17 criterion = nn.BCELoss()
18
19 d_optimizer = torch.optim.Adam(D.parameters(), lr=0.01)
20 g_optimizer = torch.optim.Adam(G.parameters(), lr=0.01)
21
22 # Assume x be real images of shape (batch_size, 784)
23 # Assume z be random noise of shape (batch_size, 100)
24
25 while True:
26     # train D
27     loss = criterion(D(x), 1) + criterion(D(G(z)), 0)
28     loss.backward()
29     d_optimizer.step()
30
31     # train G
32     loss = criterion(D(G(z)), 1)
33     loss.backward()
34     g_optimizer.step()
```



- Deep Convolutional GAN(DCGAN), 2015

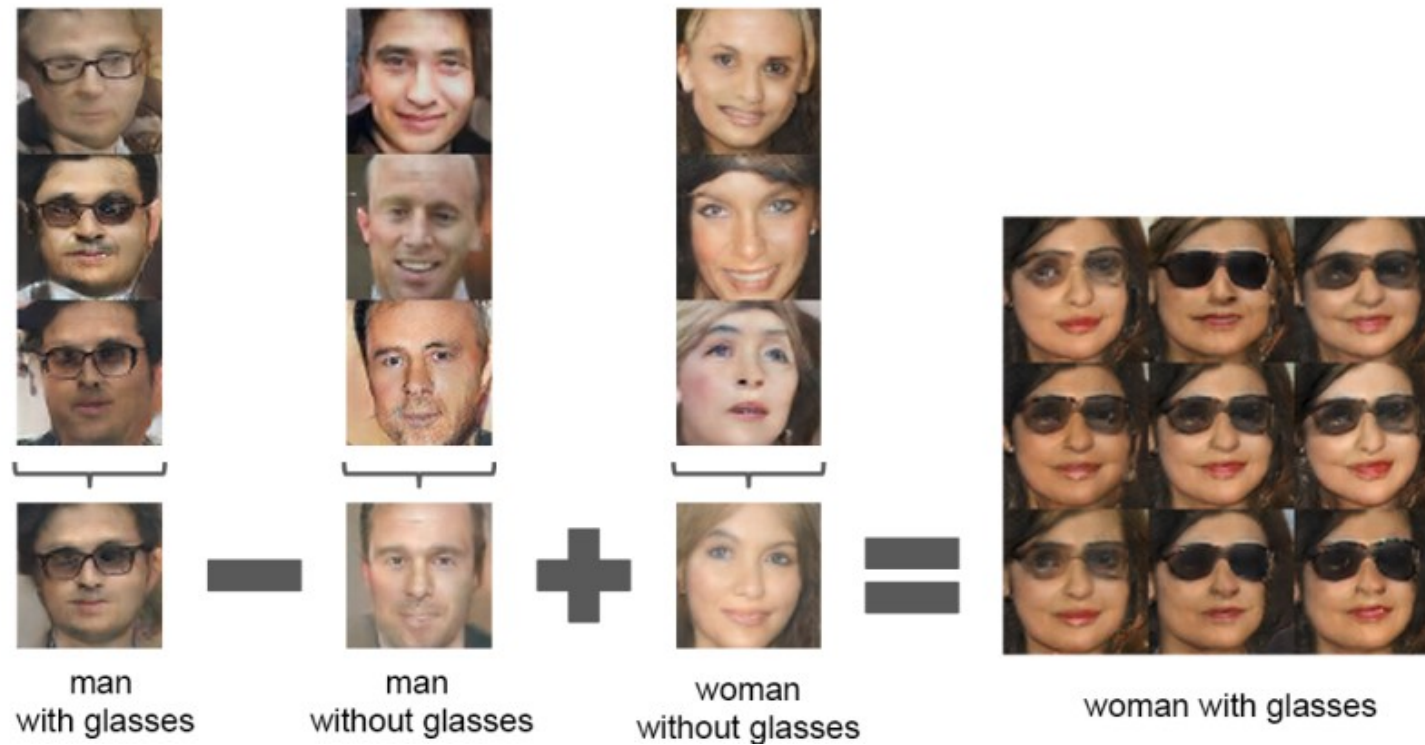
The authors present a model that is still highly preferred.







- Latent vector arithmetic



04

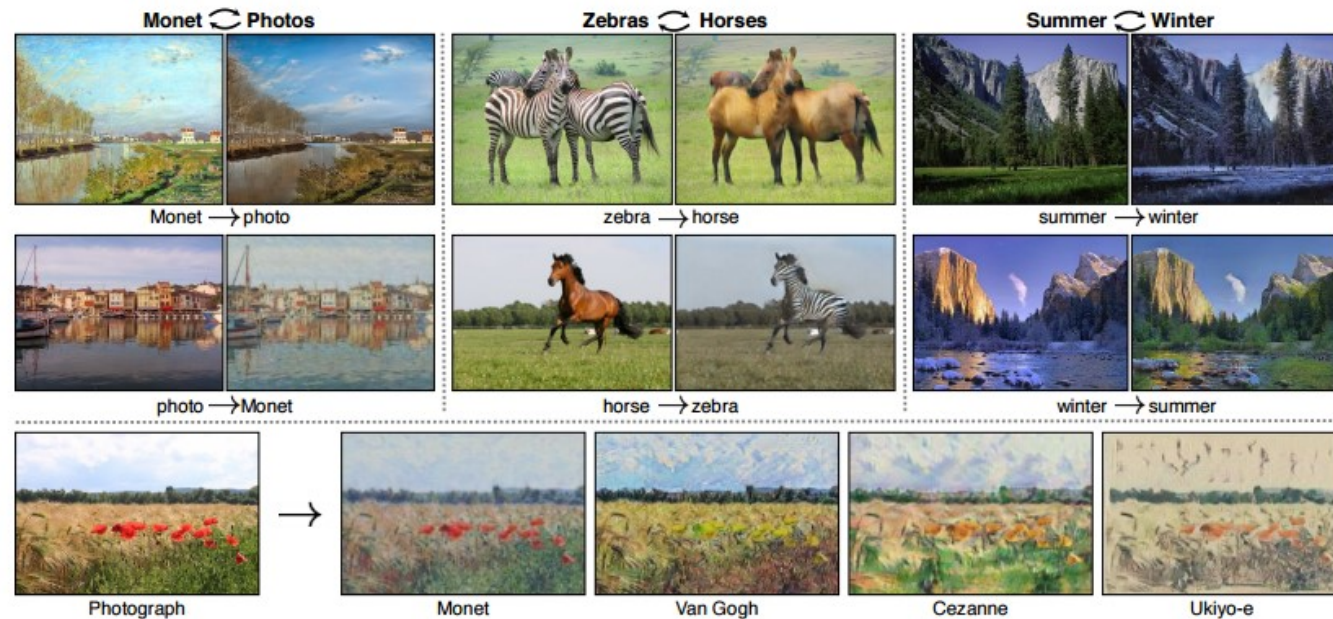
Extensions of GAN





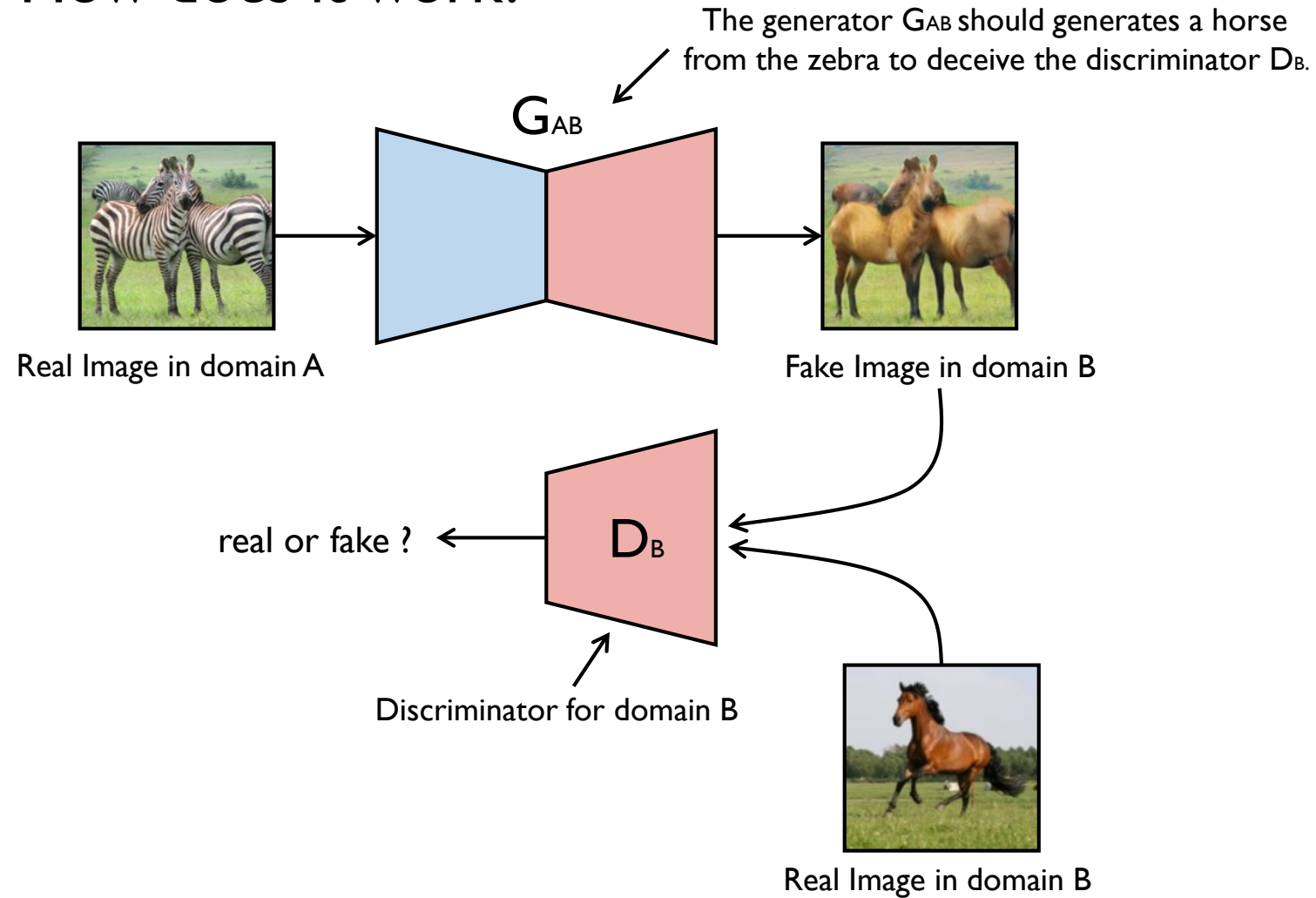
- CycleGAN: Unpaired Image-to-Image Translation

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



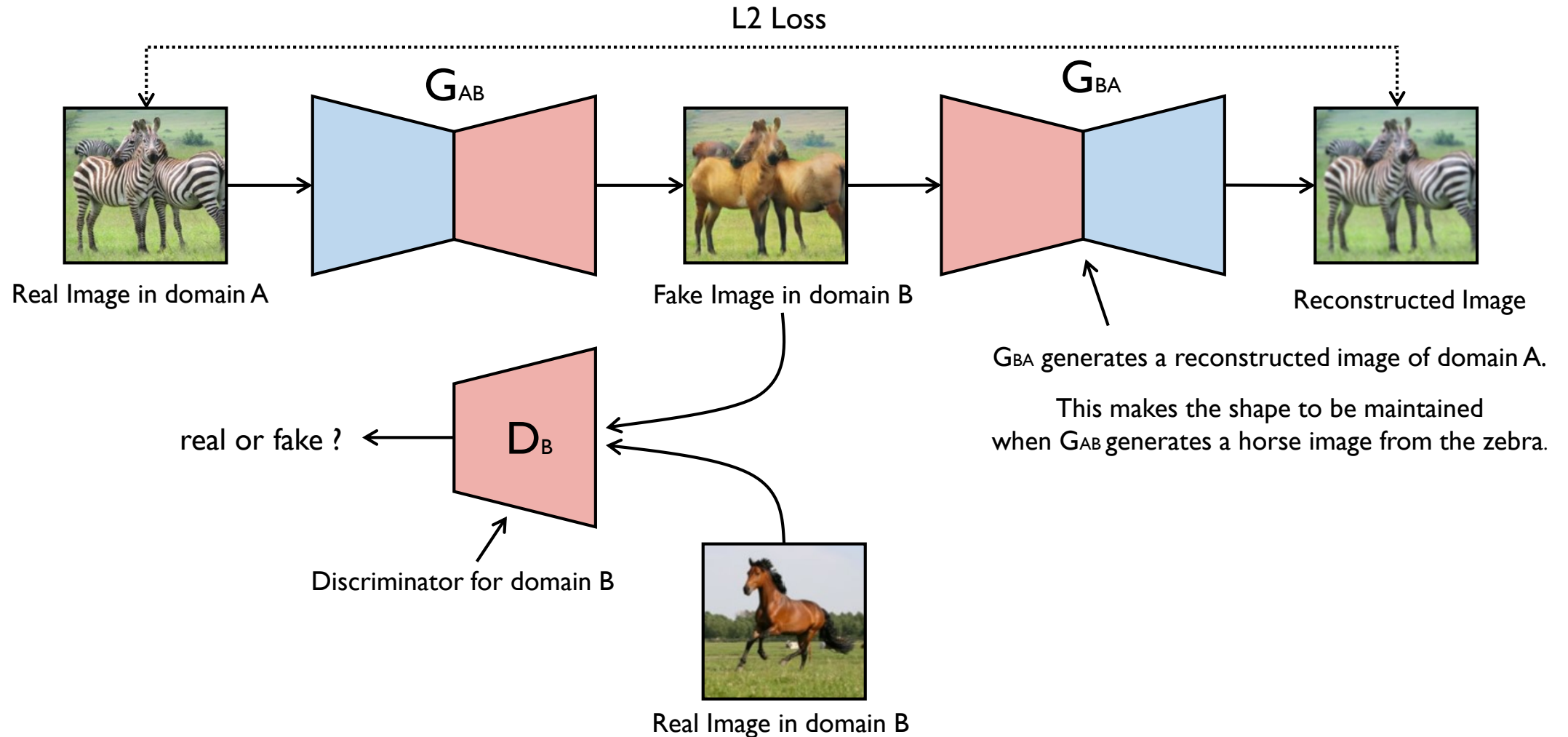


- How does it work?



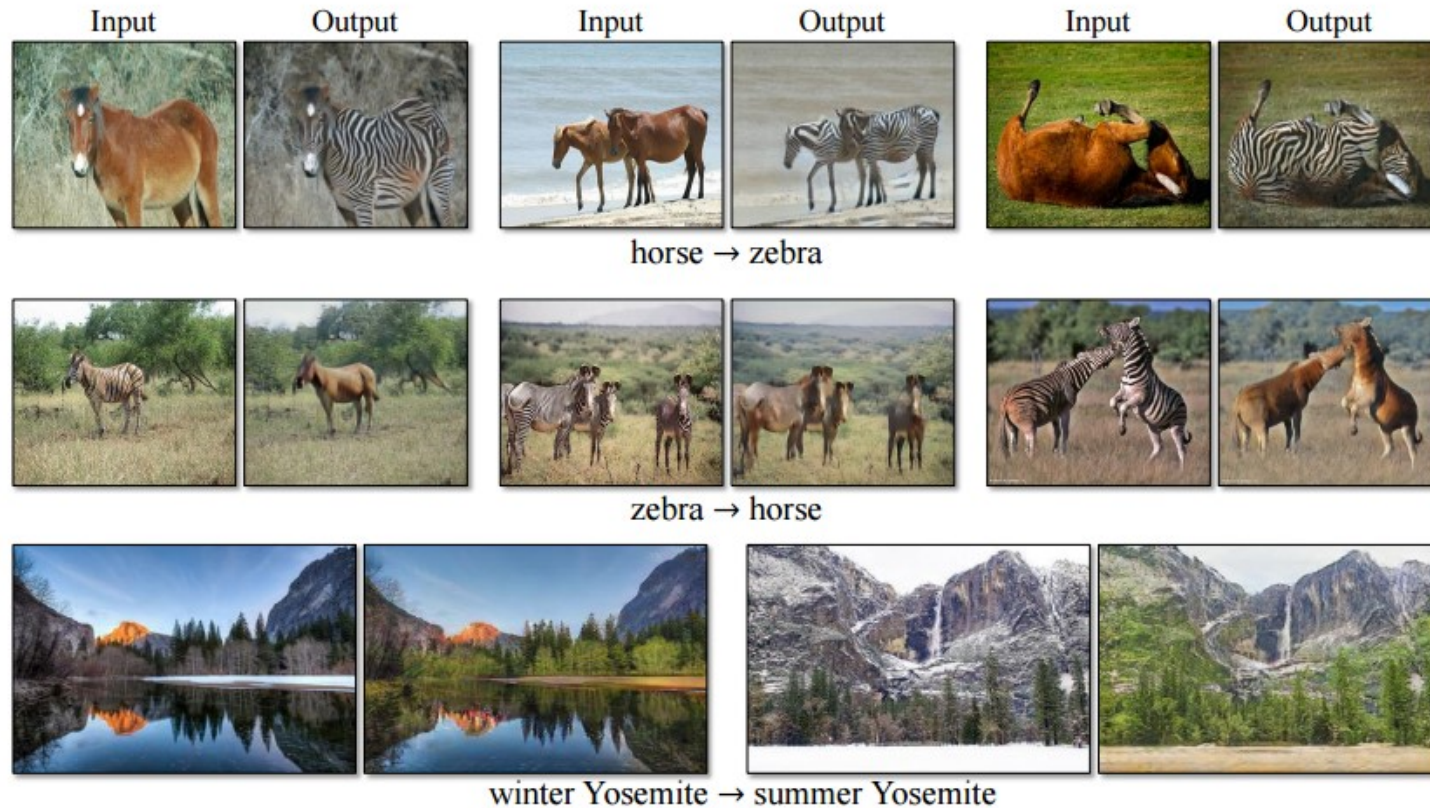


- How does it work?





- Results





- Results

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



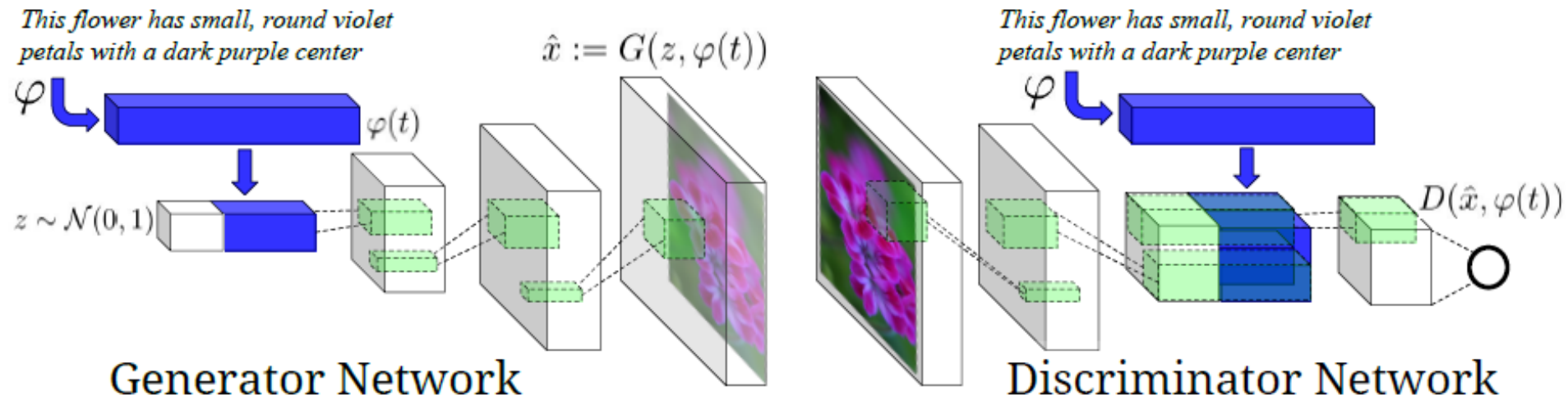
SVHN-to-MNIST



MNIST-to-SVHN

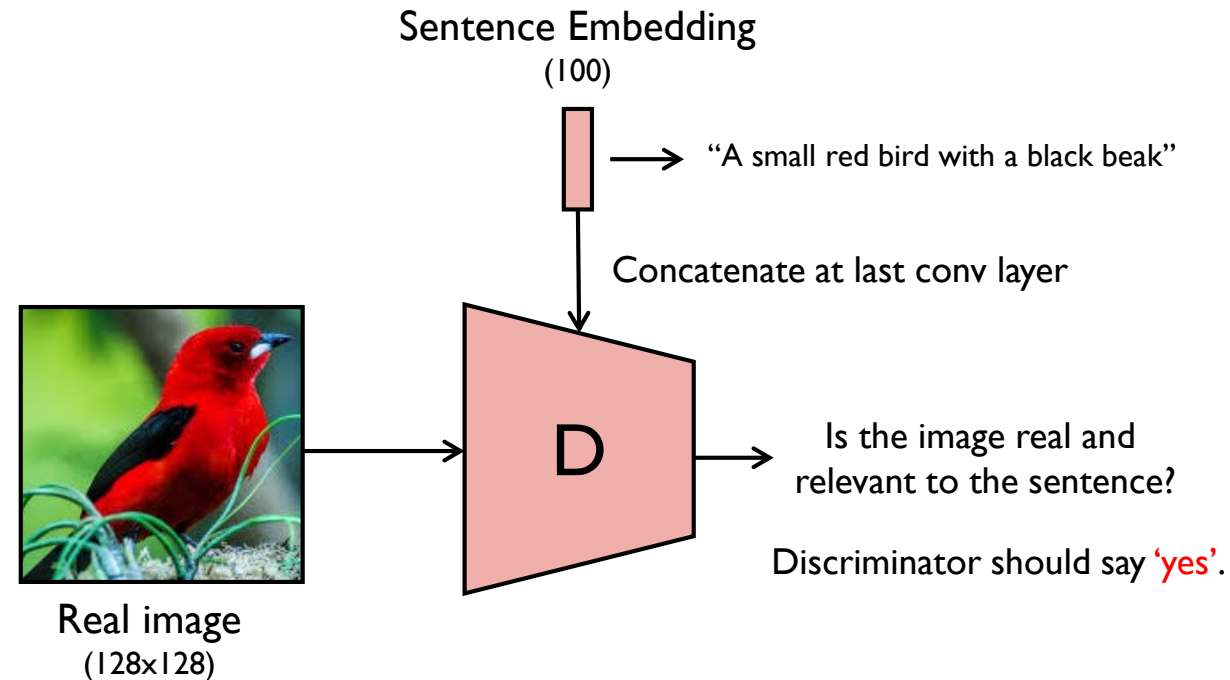


- **Generative Adversarial Text to Image Synthesis, 2016**
presents a novel model architecture that generates an image from the text.



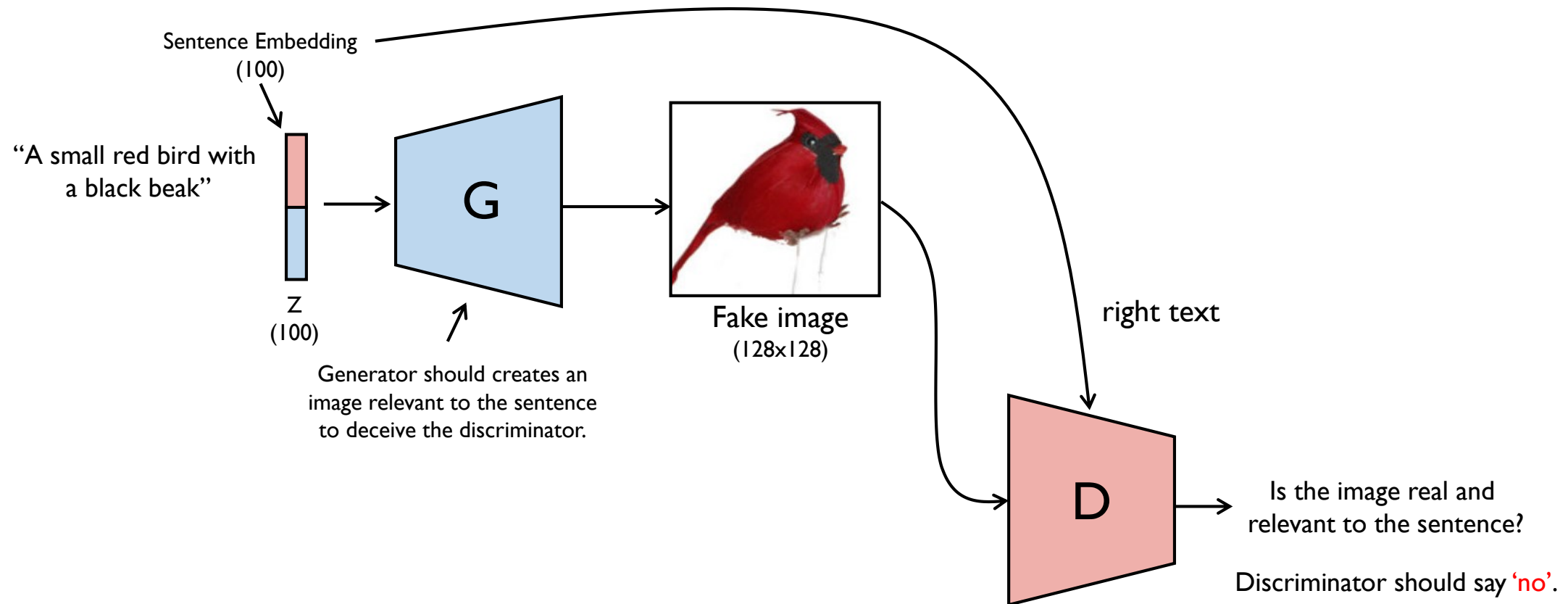


- Training with (real image, right text)



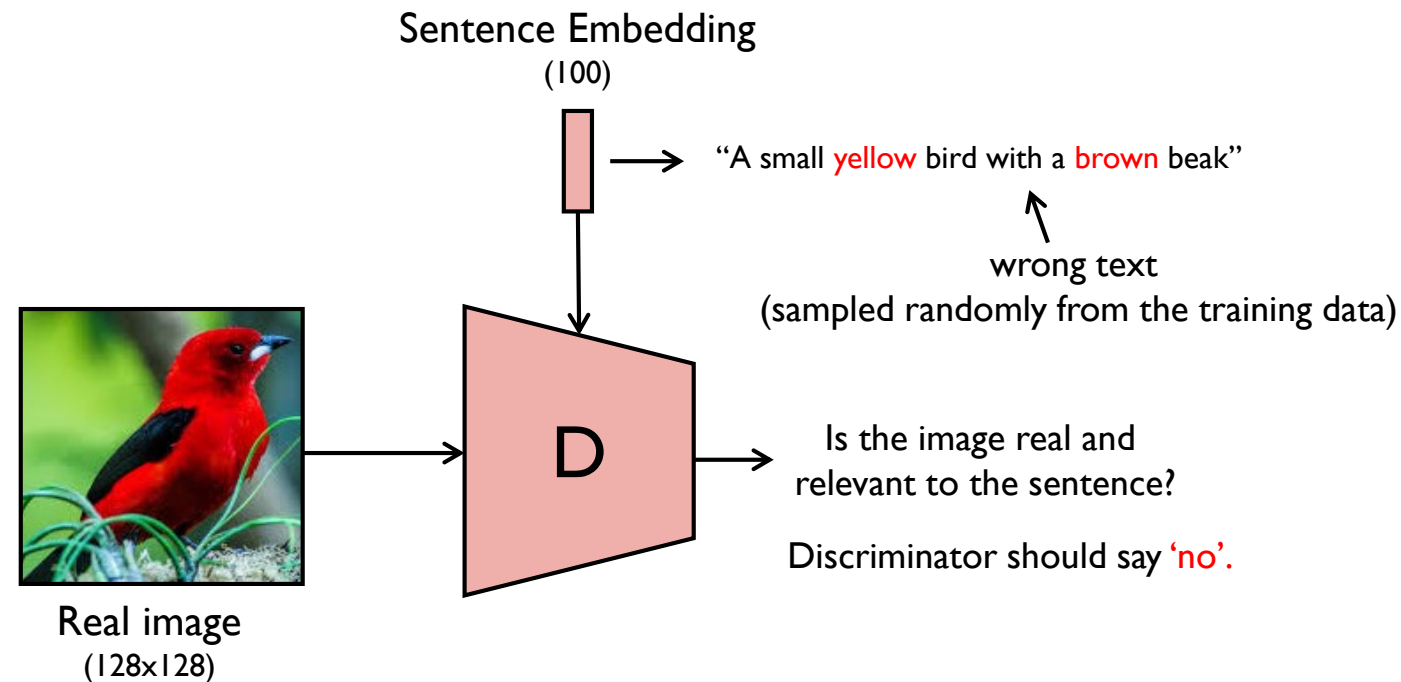


- Training with (**fake image**, right text)



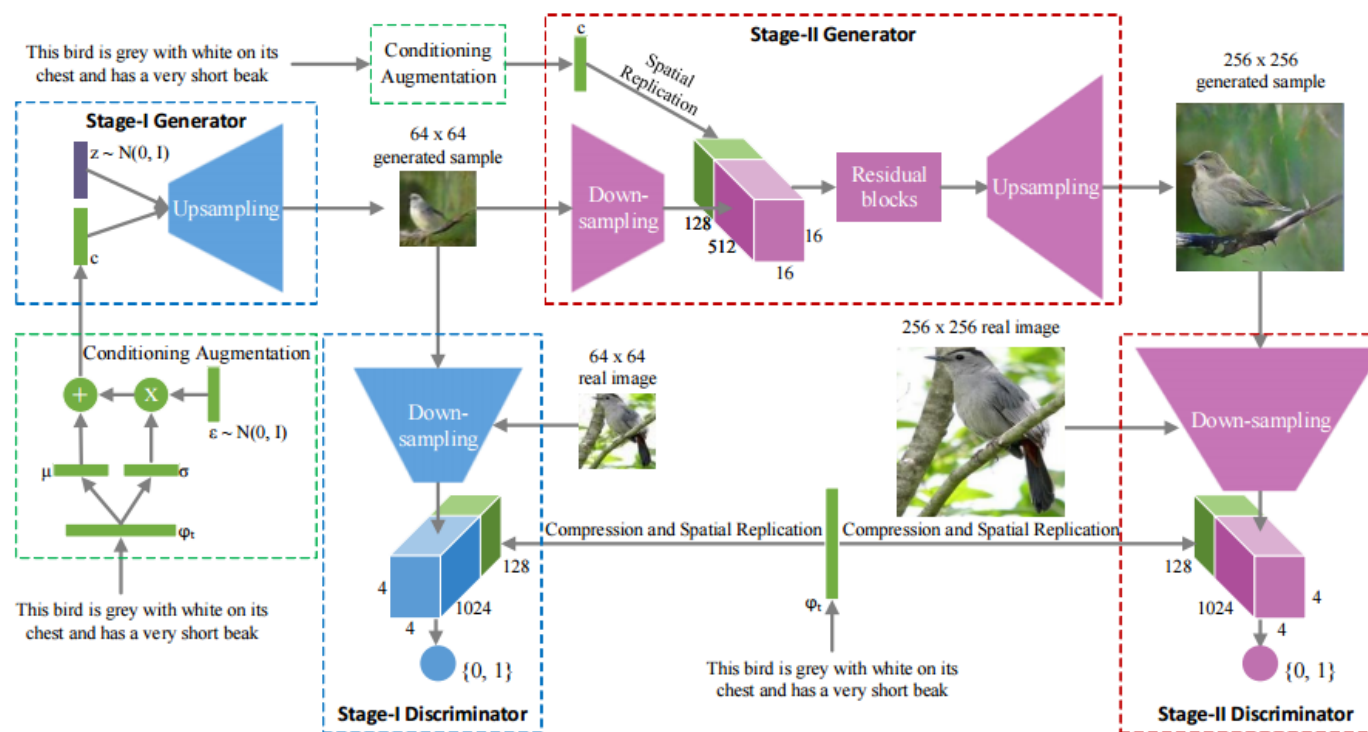


- Training with (real image, **wrong text**)





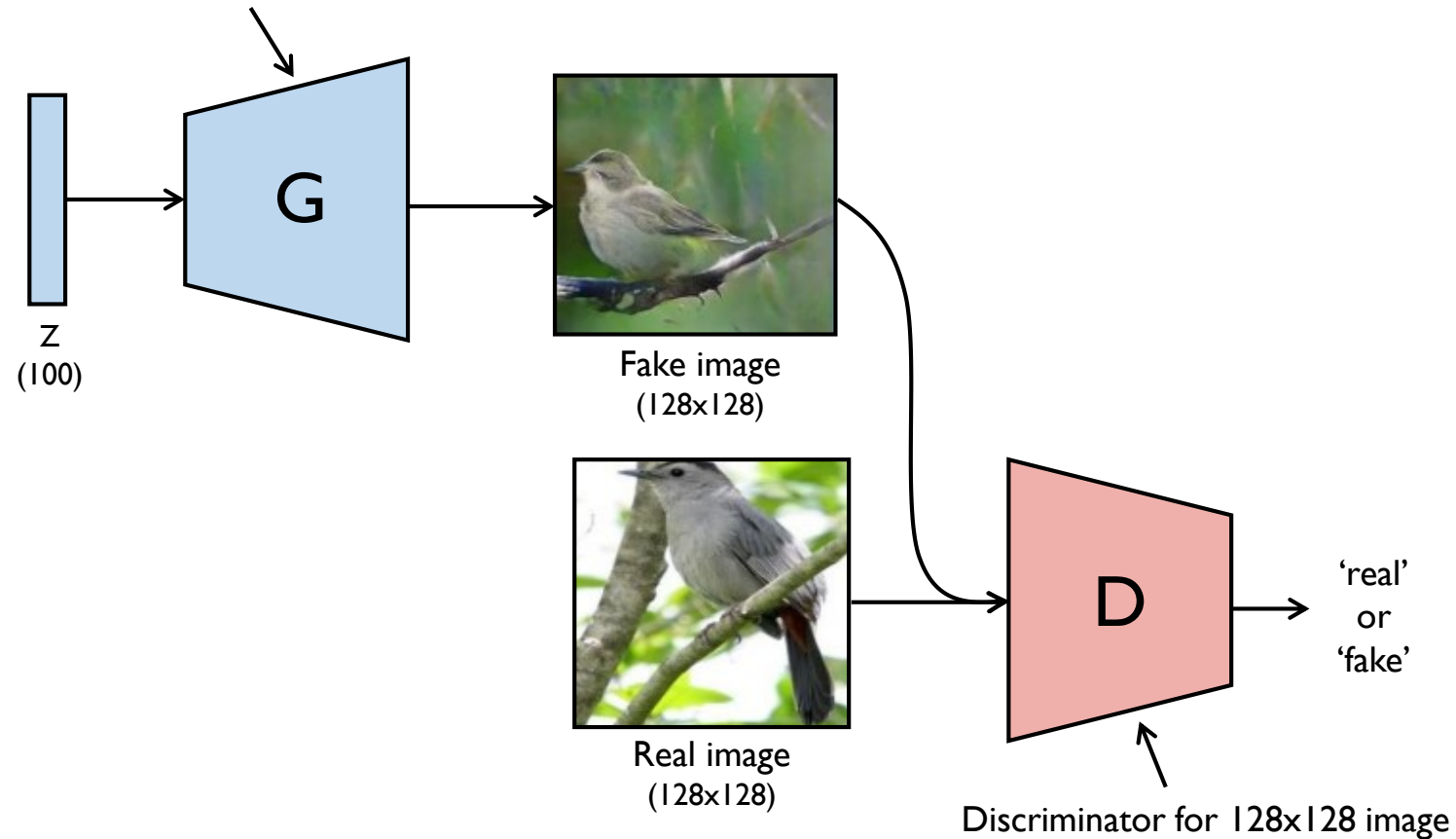
- StackGAN: Text to Photo-realistic Image Synthesis





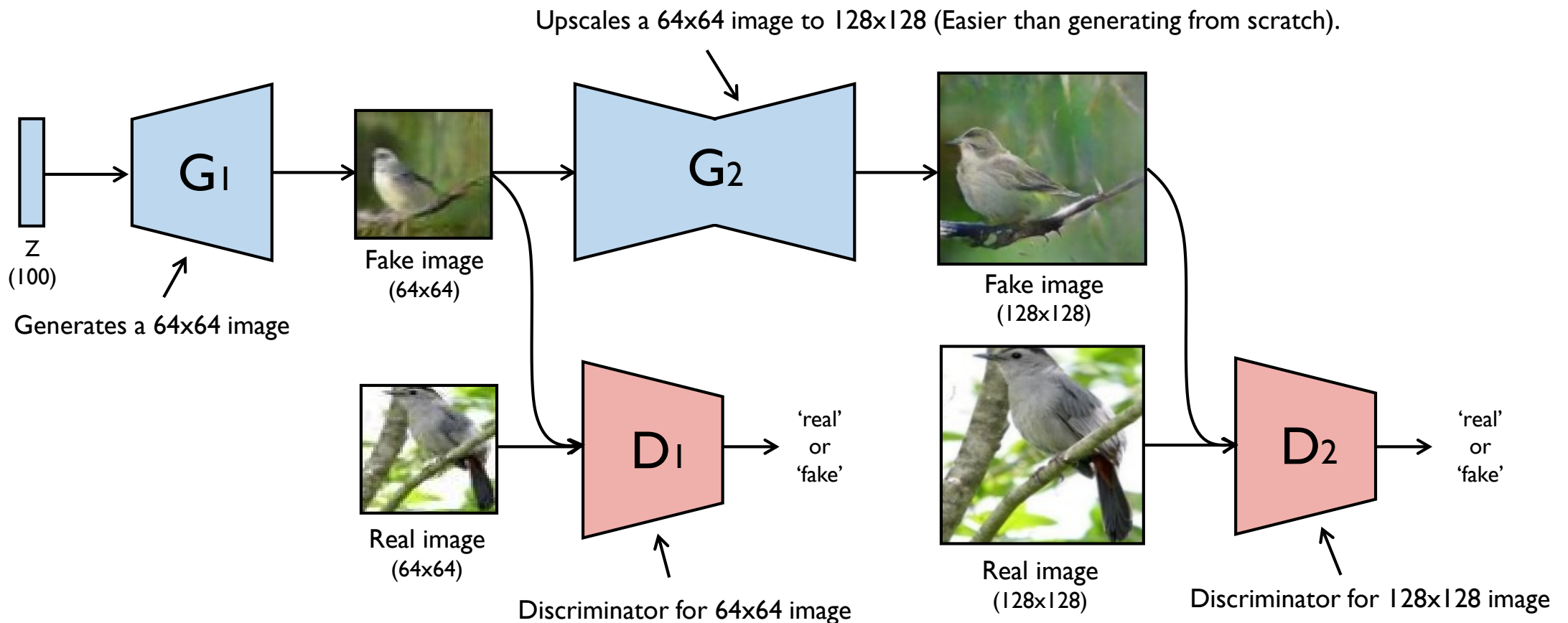
- Generating 128x128 from scratch

Generates a 128x128 image from scratch (not guarantee good result)





- Generating 128x128 from 64x64



Thank you

