GAN and CycleGAN

Reporter: Liu Jian

Outline

- Generative Adversarial Networks
  - Why do we need generative models?
  - The GAN framework
  - More details about GAN

- CycleGAN
  - CycleGAN network architecture and details
  - Some discussions about loss function, especially cycle consistency loss
  - Some successful and failure experimental cases

Why do we need generative models?

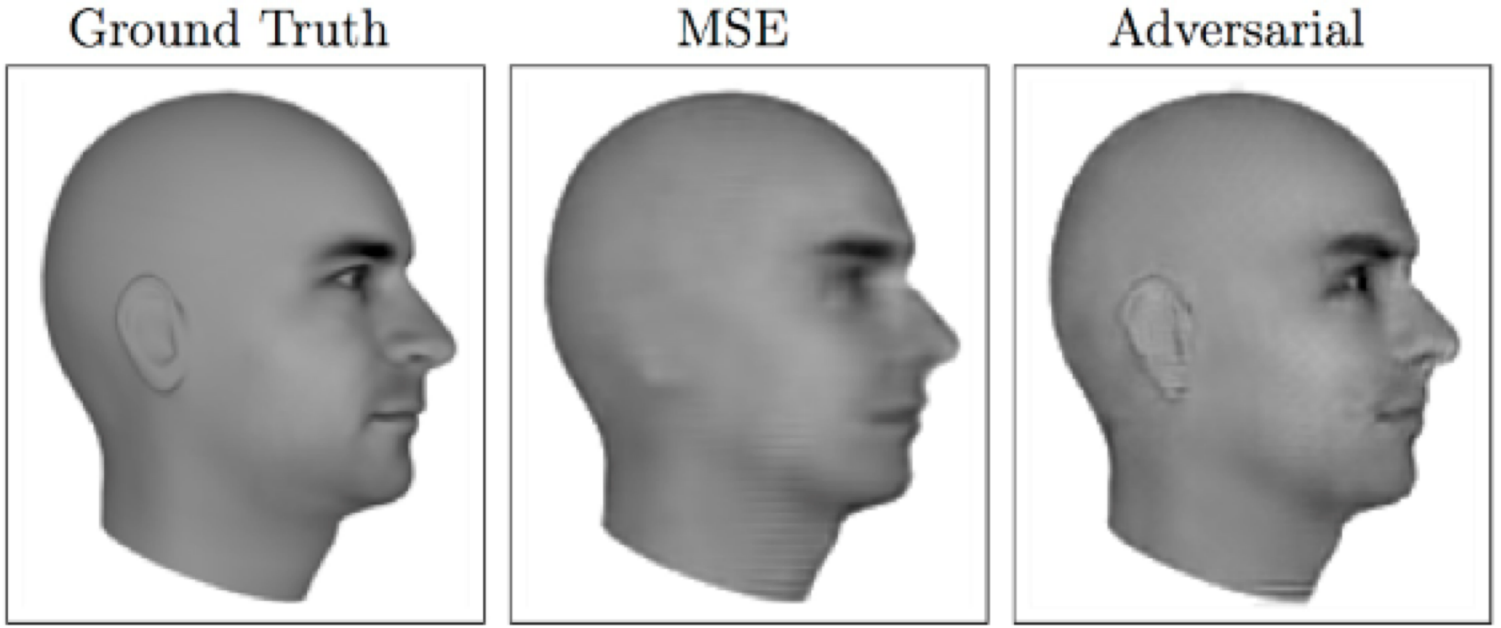
- Training and sampling from generative models is an excellent test of our ability to represent and manipulate high-dimensional probability distributions.

- Generative models can be incorporated into reinforcement learning in several ways.

- Generative models can be trained with missing data and can provide predictions on inputs that are missing data.
  - Improved Techniques for Training GANs

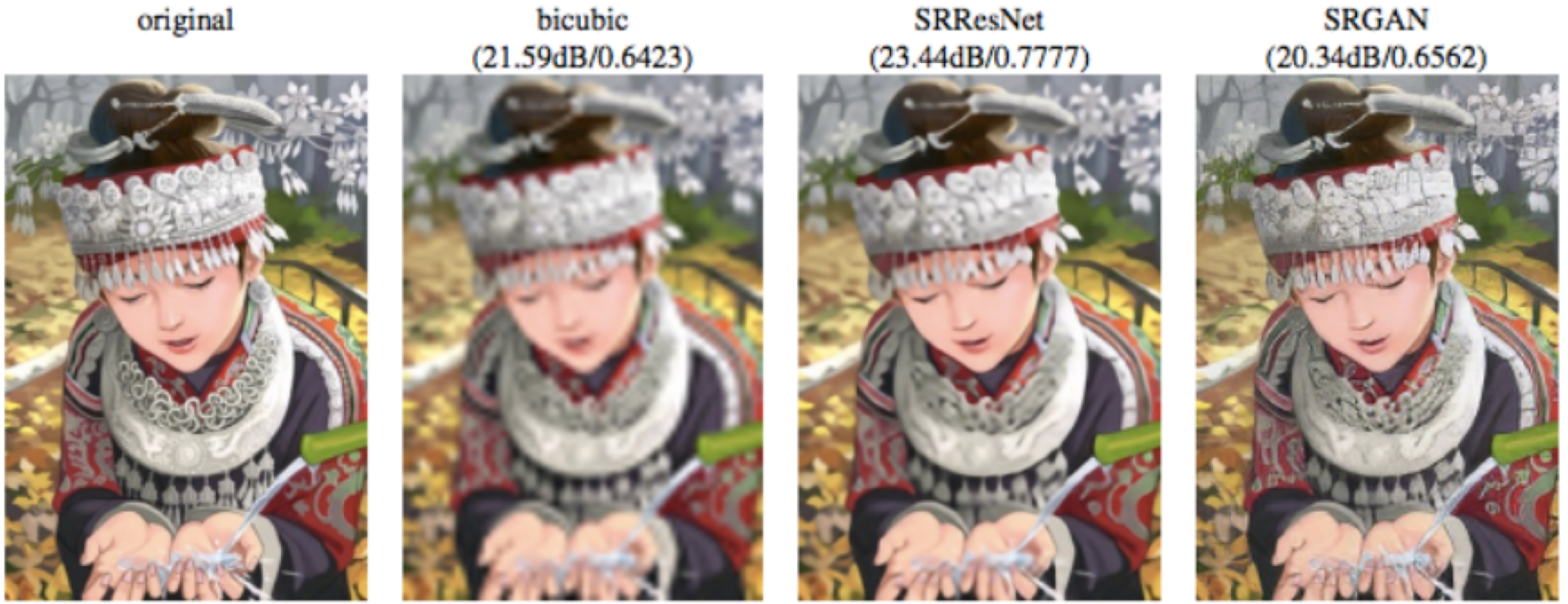
Why do we need generative models?

- Generative models, and GANs in particular, enable machine learning to work with multi-modal outputs.
  - ➢ multi-modal：Tutorial on Multimodal Machine Learning



Why do we need generative models?

- Finally, many tasks intrinsically require realistic generation of samples from some distribution.
  - ➢ Single image super-resolution



Why do we need generative models?

- Image-to-image translation applications can convert aerial photos into maps or convert sketches to images.

Many kinds of GAN…

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

ACGAN

BGAN

CGAN

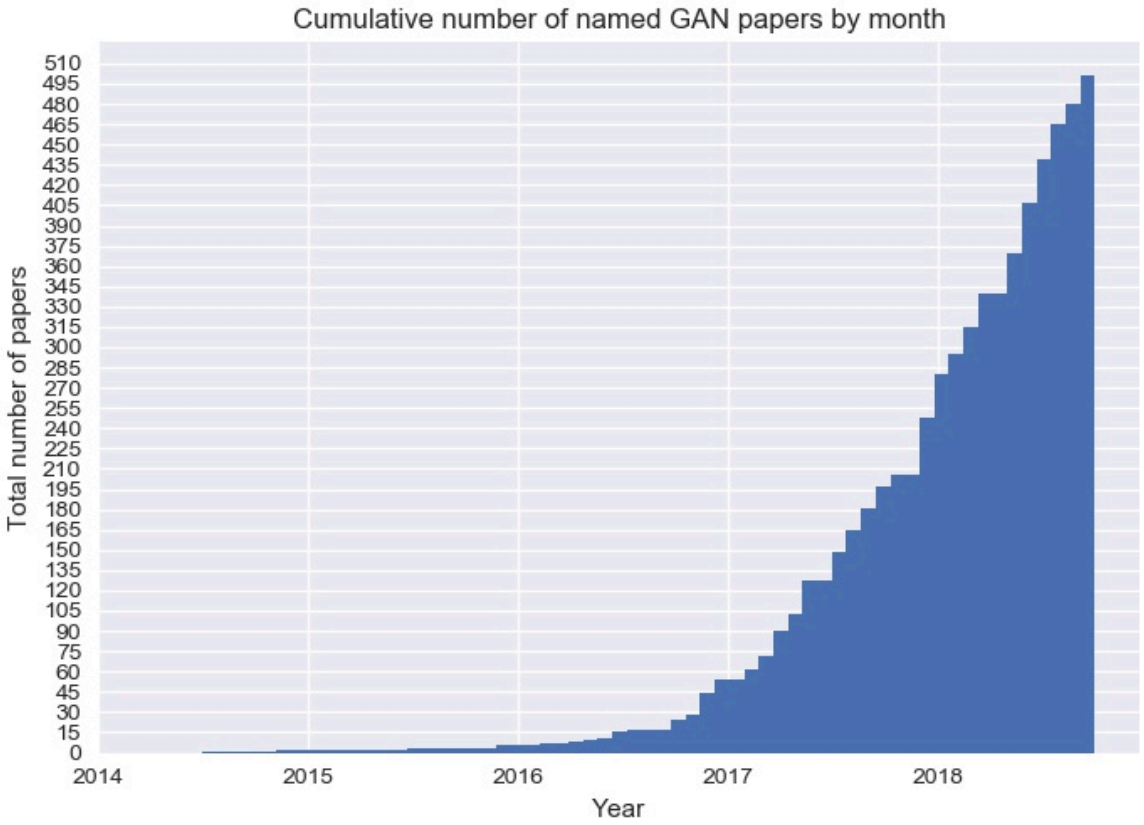
CycleGAN

DCGAN

Info GAN

WGAN

⋮



Every week, new GAN papers are coming out and it's hard to keep track of them all, not to mention the incredibly creative ways in which researchers are naming these GANs!

The GAN framework

- The basic idea of GANs is to set up a game between two players.
  - One of them is called the generator. The generator creates samples that are intended to come from the same distribution as the training data.
  - The other player is the discriminator. The discriminator examines samples to determine whether they are real or fake.
- The discriminator learns using traditional supervised learning techniques, dividing inputs into two classes (real or fake).
- The generator is trained to fool the discriminator.

The GAN framework

- The two players in the game are represented by two functions, each of which is differentiable both with respect to its inputs and with respect to its parameters.
  - The discriminator is a function D that takes x and G(z) as input and uses $\theta^{(D)}$ as parameters.
  - The generator is defined by a function G that takes z as input and uses $\theta^{(G)}$ as parameters.
- Both players have cost functions that are defined in terms of both players' parameters.
  - The discriminator wishes to minimize $V^{(D)}(\theta^{(D)}, \theta^{(G)})$ and must do so while controlling only $\theta^{(D)}$.
  - The generator wishes to minimize $V^{(G)}(\theta^{(D)}, \theta^{(G)})$ and must do so while controlling only $\theta^{(G)}$.
- If both models have sufficient capacity, then the Nash equilibrium of this game corresponds to the G(z) being drawn from the same distribution as the training data, and D(x) = 1/2 for all x.

Loss Function

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

This is a maximum and minimum optimization problem. Here we optimize D and optimize G. In essence, it is two optimization problems. After the target formula is disassembled, the following two formulas are obtained:

- **Optimize D:**

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

- **Optimize G:**

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

Algorithm

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

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

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

A simple example to implement GAN：gan_tensorflow

```python
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import os

# 初始化参数时使用的xavier_init函数
def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)        # 初始化标准差
    return tf.random_normal(shape=size, stddev=xavier_stddev) # 返回初始化的结果

# X表示真实样本(即真实的手写数字)
X = tf.placeholder(tf.float32, shape=[None, 784])

# D_W1表示使用xavier方式初始化的判别器的D_W1参数，是一个784行128列的矩阵
D_W1 = tf.Variable(xavier_init([784, 128]))
# D_b1表示全零方式初始化的判别器的bias参数，是一个长度为128的向量
D_b1 = tf.Variable(tf.zeros(shape=[128]))

# D_W2表示使用xavier方式初始化的判别器的D_W2参数，是一个128行1列的矩阵
D_W2 = tf.Variable(xavier_init([128, 1]))
# D_b2表示全零方式初始化的判别器的bias参数，是一个长度为1的向量
D_b2 = tf.Variable(tf.zeros(shape=[1]))

# theta_D表示判别器的可训练参数集合
theta_D = [D_W1, D_W2, D_b1, D_b2]

# Z表示生成器的输入(在这里是噪声)，是一个N列100行的矩阵
Z = tf.placeholder(tf.float32, shape=[None, 100])
# G_W1表示使用xavier方式初始化的生成器的G_W1参数，是一个100行128列的矩阵
G_W1 = tf.Variable(xavier_init([100, 128]))
# G_b1表示全零方式初始化的生成器的bias参数，是一个长度为128的向量
G_b1 = tf.Variable(tf.zeros(shape=[128]))

# G_W2表示使用xavier方式初始化的生成器的G_W2参数，是一个128行784列的矩阵
G_W2 = tf.Variable(xavier_init([128, 784]))
# G_b2表示全零方式初始化的生成器的bias参数，是一个长度为784的向量
G_b2 = tf.Variable(tf.zeros(shape=[784]))

# theta_G表示生成器的可训练参数集合
theta_G = [G_W1, G_W2, G_b1, G_b2]
```

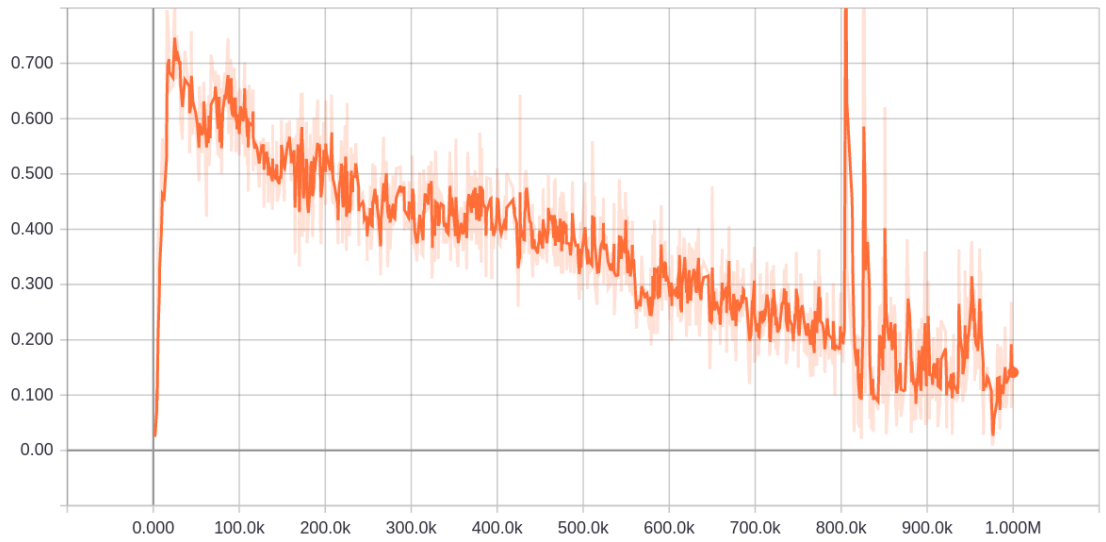
```python
# 生成维度为[m, n]的随机噪声(从均匀分布中随机采样)作为生成器G的输入
def sample_Z(m, n):
    return np.random.uniform(-1., 1., size=[m, n])

# 生成器，噪声z的维度为[N, 100]
def generator(z):
    # 输入的随机噪声乘以G_W1矩阵加上偏置G_b1(相当于全连接层)，G_h1维度为[N, 128]
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
    # G_h1乘以G_W2矩阵加上偏置G_b2(相当于全连接层)，G_log_prob维度为[N, 784]
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    # G_log_prob经过一个sigmoid函数，G_prob维度为[N, 784]
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob

# 判别器，真实样本x的维度为[N, 784]
def discriminator(x):
    # 输入乘以D_W1矩阵加上偏置D_b1，D_h1维度为[N, 128]
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
    # D_h1乘以D_W2矩阵加上偏置D_b2，D_logit维度为[N, 1]
    D_logit = tf.matmul(D_h1, D_W2) + D_b2
    # D_logit经过一个sigmoid函数，D_prob维度为[N, 1]
    D_prob = tf.nn.sigmoid(D_logit)

    return D_prob, D_logit


G_sample = generator(Z)
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)


# 保存图片时使用的plot函数
def plot(samples):
    fig = plt.figure(figsize=(4, 4))
    gs = gridspec.GridSpec(4, 4)
    gs.update(wspace=0.05, hspace=0.05)

    for i, sample in enumerate(samples):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(sample.reshape(28, 28), cmap='Greys_r')

    return fig
```

A simple example to implement GAN：gan_tensorflow

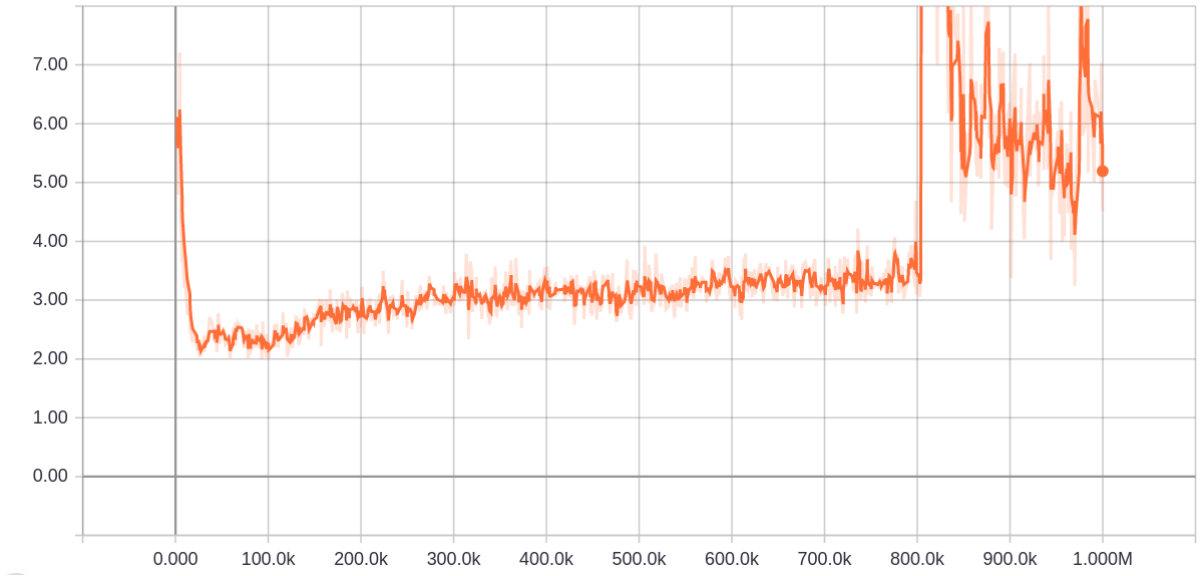
```
# 对判别器对真实样本的判别结果计算误差(将结果与1比较)
D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_real, labels=tf.ones_like(D_logit_real)))
# 对判别器对虚假样本(即生成器生成的手写数字)的判别结果计算误差(将结果与0比较)
D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_fake, labels=tf.zeros_like(D_logit_fake)))
# 判别器的误差
D_loss = D_loss_real + D_loss_fake
# 生成器的误差(将判别器返回的对虚假样本的判别结果与1比较)
G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_fake, labels=tf.ones_like(D_logit_fake)))

# 判别器的优化器
D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
# 生成器的优化器
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)

# 训练时的batch_size
mb_size = 128
# 生成器输入的随机噪声的列的维度
Z_dim = 100

mnist = input_data.read_data_sets('../../MNIST_data', one_hot=True)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

if not os.path.exists('out/'):
    os.makedirs('out/')

i = 0

for it in range(1000000):
    if it % 1000 == 0:
        samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})

        fig = plot(samples)
        plt.savefig('out/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
        i += 1
        plt.close(fig)

    X_mb, _ = mnist.train.next_batch(mb_size)

    _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z: sample_Z(mb_size, Z_dim)})
    _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_Z(mb_size, Z_dim)})

    if it % 1000 == 0:
        print('Iter: {}'.format(it))
        print('D loss: {:.4}'. format(D_loss_curr))
        print('G_loss: {:.4}'.format(G_loss_curr))
        print()
```

Results：gan_result

d_loss

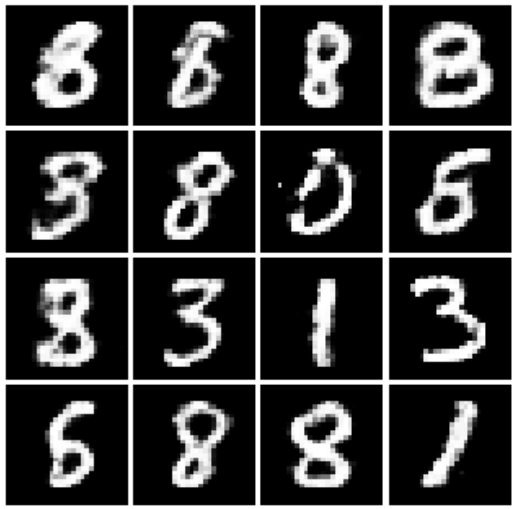


g_loss

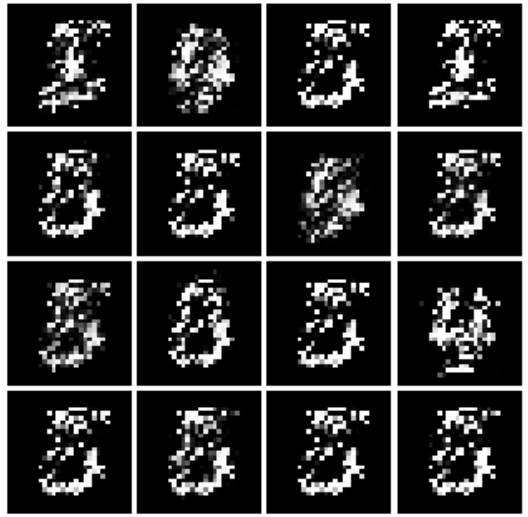


After training 6k times

After training 31.6w times



After training 85.7w times

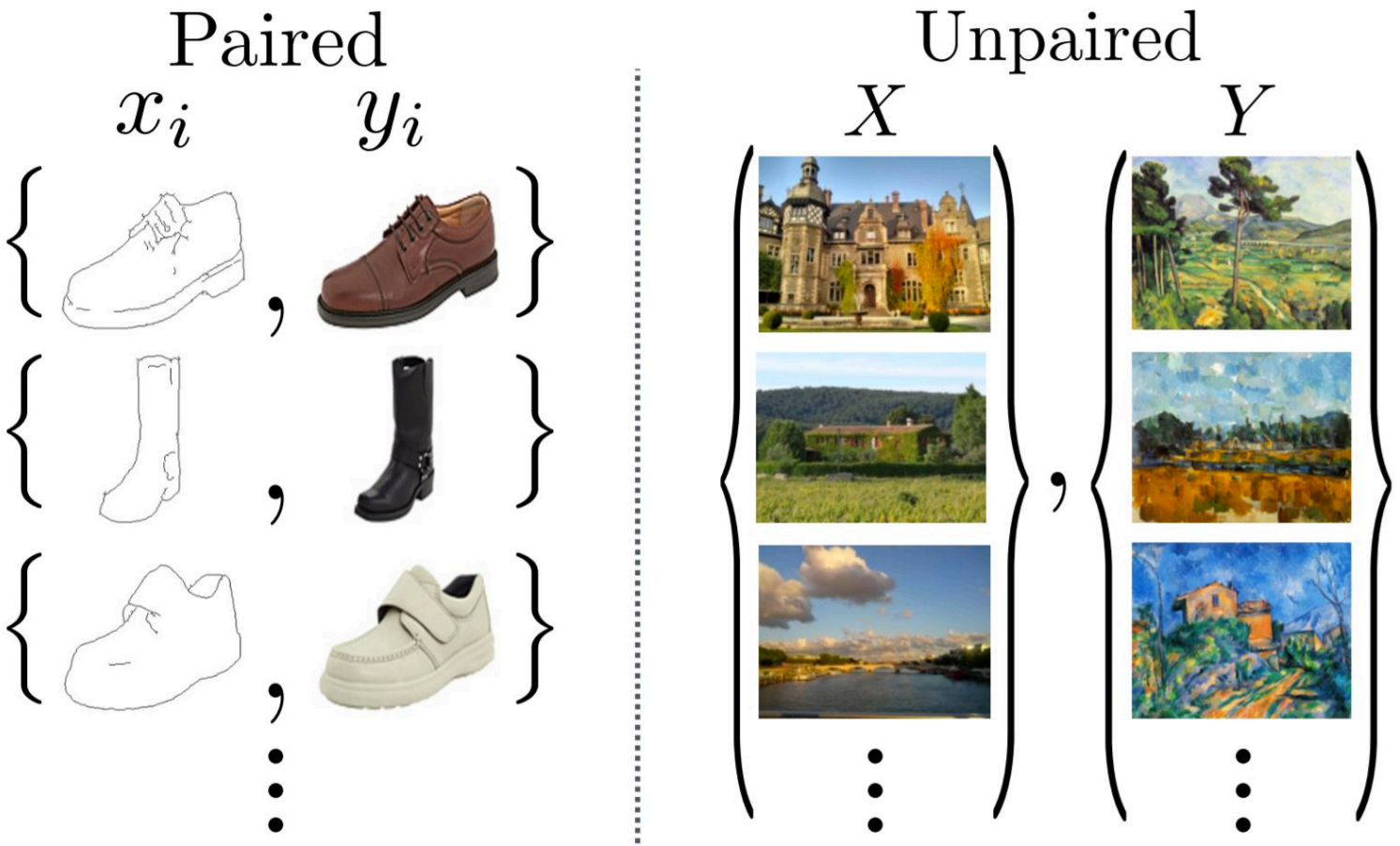


CycleGAN —— Some examples of style transfer

convert a photo into the style of Van Gogh or Picasso

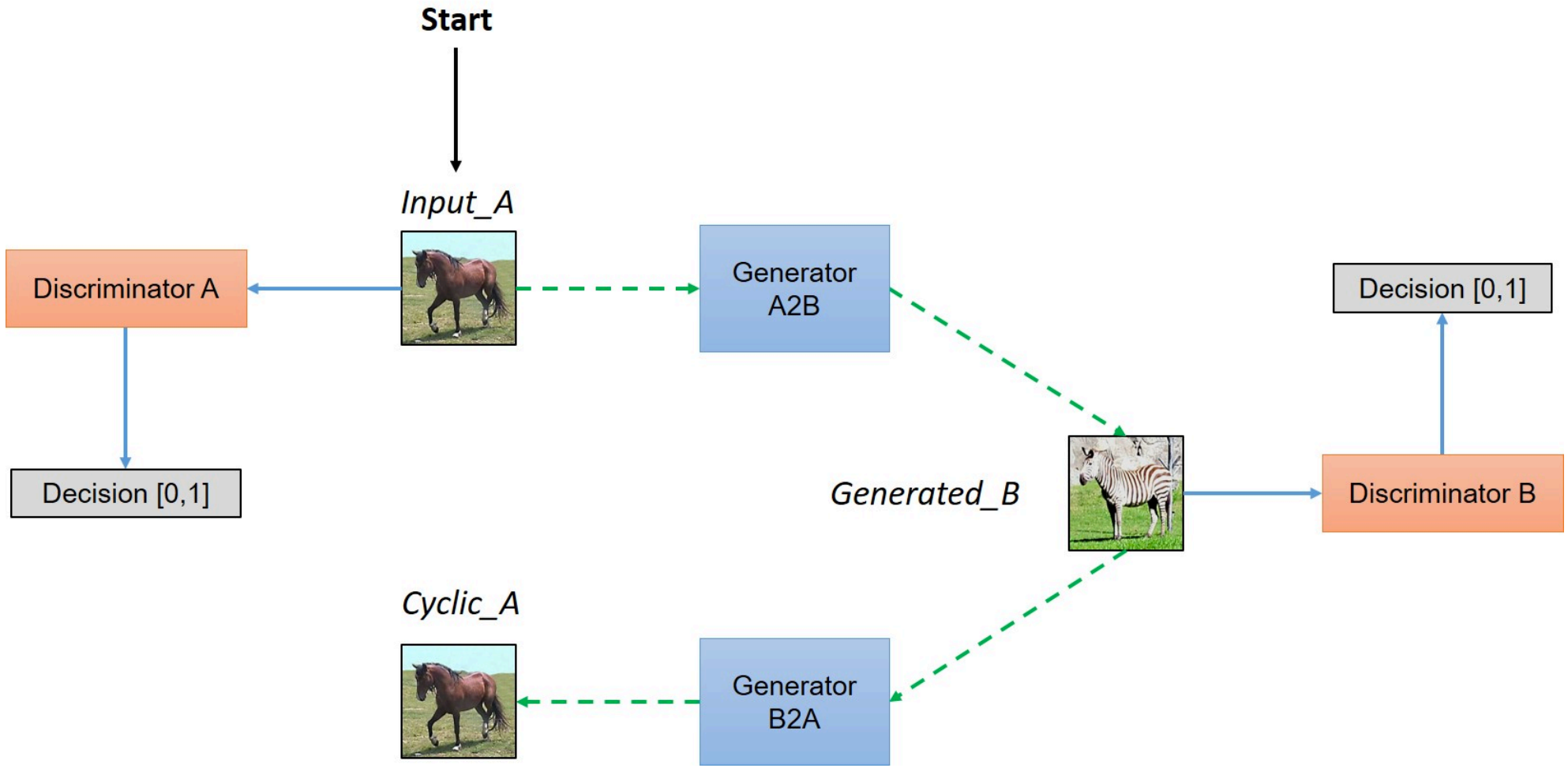
put a smile on Agent 42's face with the virally popular Faceapp

CycleGAN —— Motivation

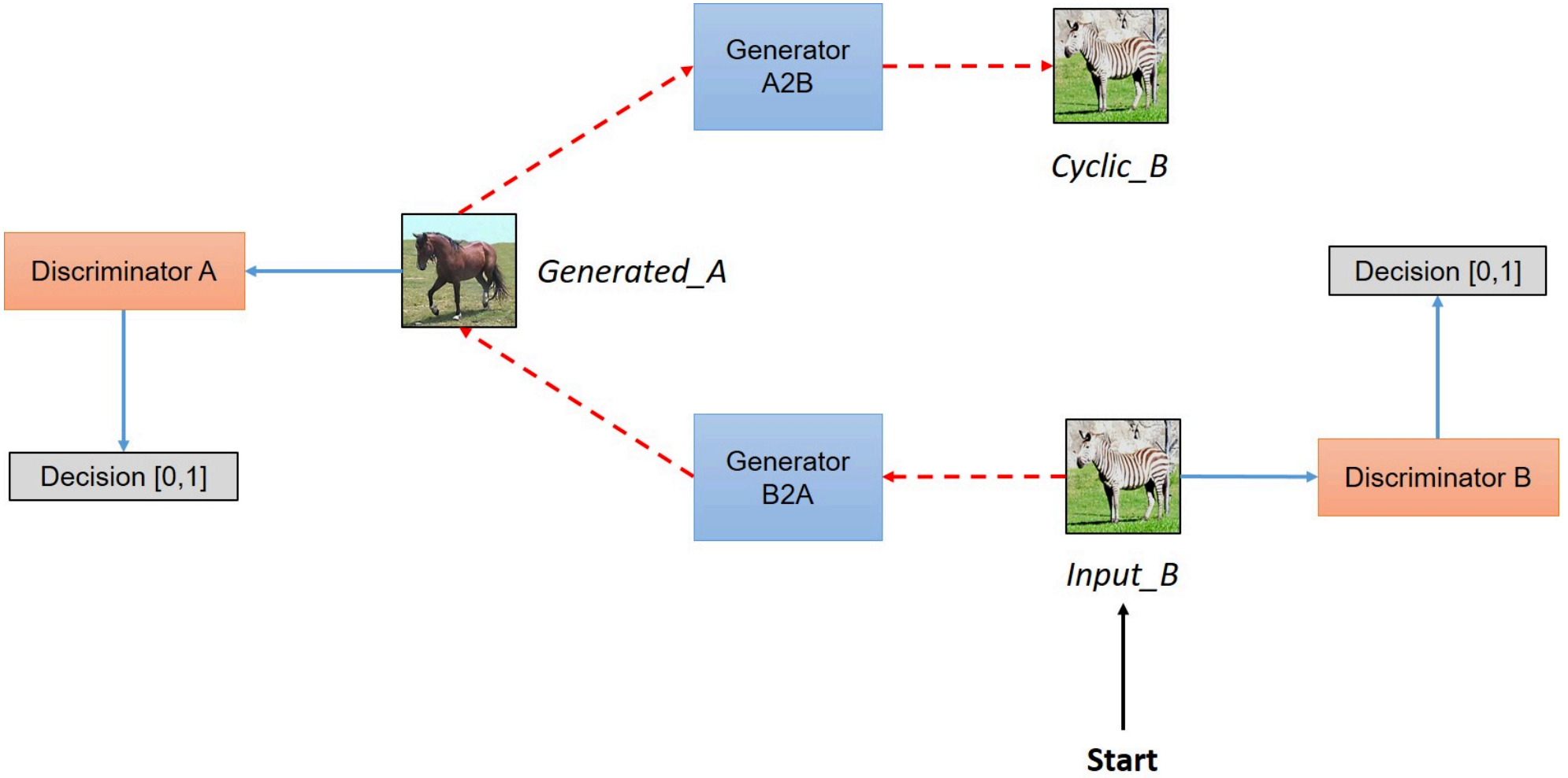


Pix2Pix is a practical image translation algorithm. The Pix2Pix framework is based on GAN and requires paired training data. However, obtaining paired training data can be difficult and expensive. So proposed the CycleGAN framework.

CycleGAN Network Architecture

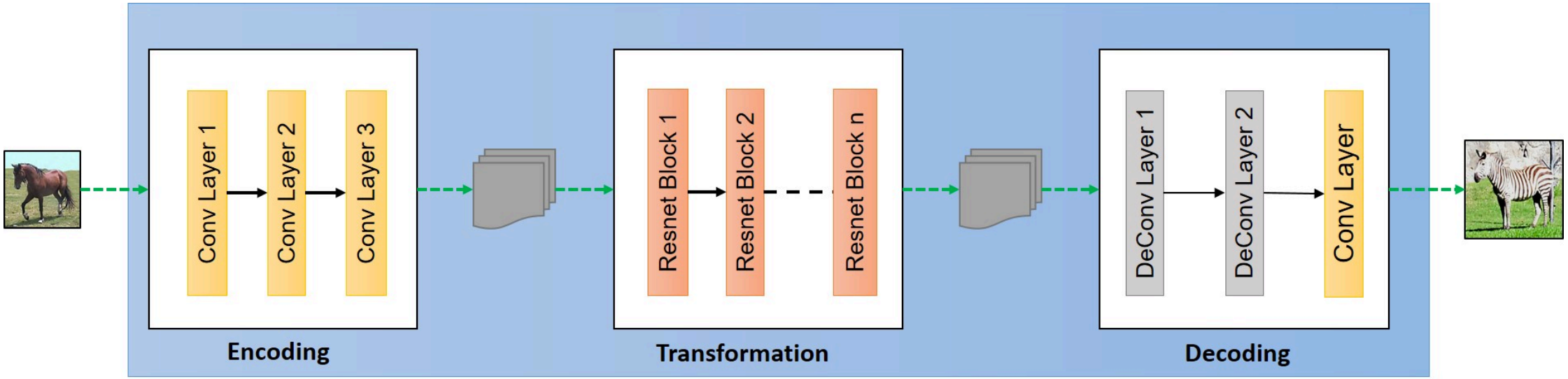


CycleGAN Network Architecture



Building the generator

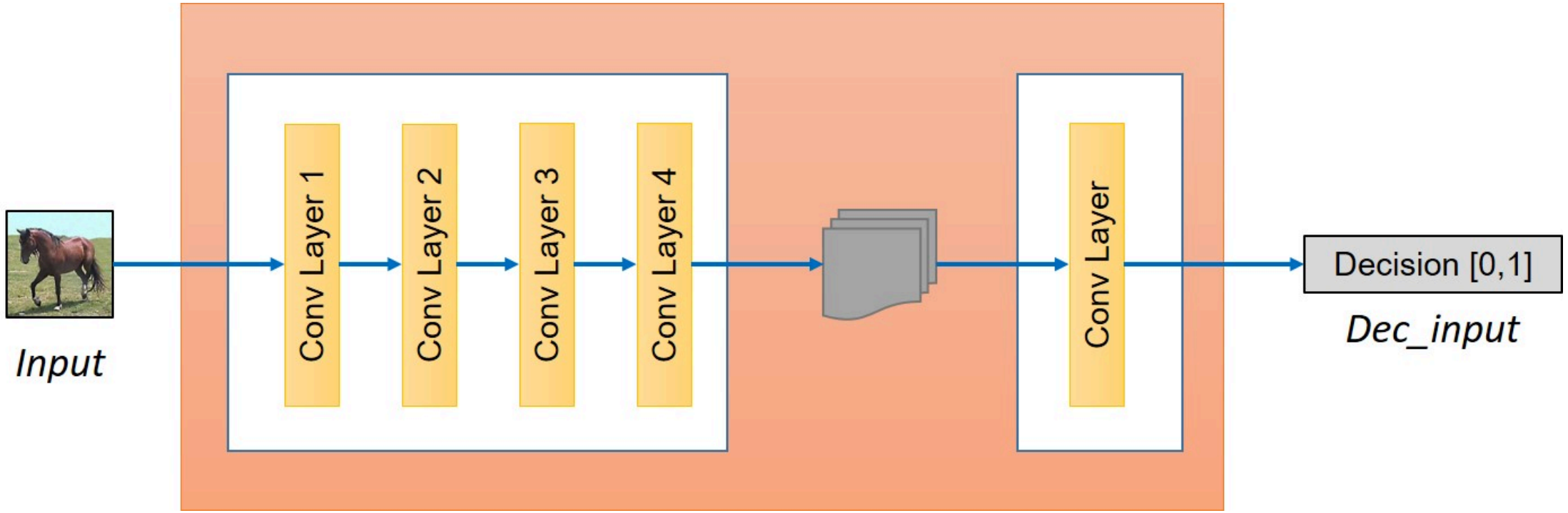
High level structure of Generator can be viewed in the following image.



The generator have three components:
1.Encoder
2.Transformer
3.Decoder

Building the discriminator

The discriminator would take an image as an input and try to predict if it is an original or the output from the generator. It is simply a convolution network.



Building the model

Before getting to loss funtion let us define the base and see how to take input, construct the model.

```
input_A = tf.placeholder(tf.float32, [batch_size, img_width, img_height, img_layer], name="input_A")
input_B = tf.placeholder(tf.float32, [batch_size, img_width, img_height, img_layer], name="input_B")
```

These placeholders will act as input while defining our model as follow.

```
gen_B = build_generator(input_A, name="generator_AtoB")
gen_A = build_generator(input_B, name="generator_BtoA")
dec_A = build_discriminator(input_A, name="discriminator_A")
dec_B = build_discriminator(input_B, name="discriminator_B")

dec_gen_A = build_discriminator(gen_A, "discriminator_A")
dec_gen_B = build_discriminator(gen_B, "discriminator_B")
cyc_A = build_generator(gen_B, "generator_BtoA")
cyc_B = build_generator(gen_A, "generator_AtoB")
```

Loss Function

- Adversarial Loss for the mapping function $\boldsymbol{G}$: $\boldsymbol{X} \rightarrow \boldsymbol{Y}$ and its discriminator

$$L_{GAN}(G, D_Y, \mathrm{X}, \mathrm{Y}) = E_{y \sim p_{data}(y)}[log D_Y(y)] + E_{x \sim p_{data}(x)}[log(1 - D_Y(G(x))]$$

- Adversarial Loss for the mapping function $\mathbf{F}$: $\boldsymbol{X} \rightarrow \boldsymbol{Y}$ and its discriminator

$$L_{GAN}(\mathrm{F}, D_X, \mathrm{Y}, \mathrm{X}) = E_{x \sim p_{data}(x)}[log D_X(x)] + E_{y \sim p_{data}(y)}[log(1 - D_X(F(y))]$$

- Cycle Consistency Loss

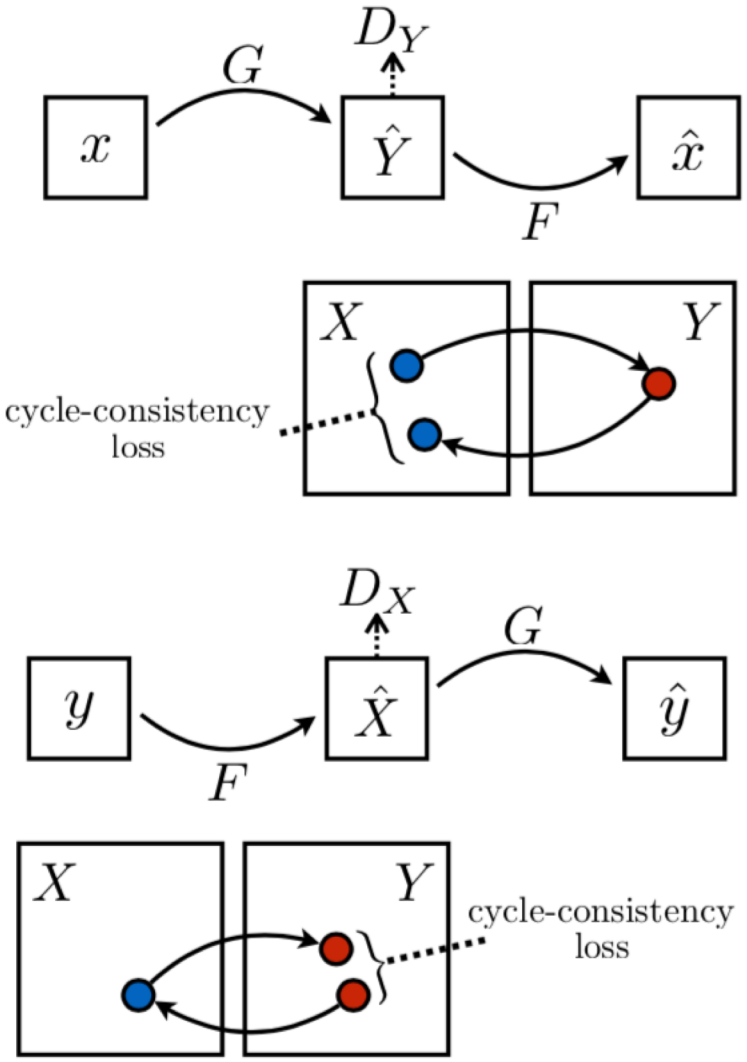
$$L_{cyc}(G, F) = E_{x \sim p_{data}(x)}[\, \|F(G(x)) - x\|_1 \,] + E_{y \sim p_{data}(y)}[\, \|G(F(y)) - y\|_1 \,]$$

- The full objective is:

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F)$$

where λ controls the relative importance of the two objectives.

What is Cycle Consistency Loss?



forward cycle-consistency loss: $\boldsymbol{x \rightarrow G(x) \rightarrow F(G(x)) \approx x}$

backward cycle-consistency loss: $\boldsymbol{y \rightarrow F(y) \rightarrow G(F(y)) \approx y}$

Why do we need Cycle Consistency Loss?

- Quoting from the original paper:

Adversarial training can, in theory, learn mappings G and F that produce outputs identically distributed as target domains Y and X respectively. However, with large enough capacity, a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution.
To further reduce the space of possible mapping functions, we argue that the learned mapping functions should be cycle-consistent:

➢ for each image x from domain X, the image translation cycle should be able to bring x back to the original image, i.e. $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$. We call this forward cycle consistency.

➢ for each image y from domain Y , G and F should also satisfy backward cycle consistency:

$$y \rightarrow F(y) \rightarrow G(F(y)) \approx y$$

- In short, the reason for artificially introducing cycle consistency loss is reducing the size of the mapping function solution space, so that mapping function can map an individual input x_i to a desired output y_i

Further thinking: Is Cycle Consistency Loss always work?

- Many-to-many mappings of CycleGAN.
  - Augmented CycleGAN: Learning Many-to-Many Mappings from Unpaired Data

- CycleGAN introduces Cycle Consistency Loss, which guarantees one-to-one, but also limits diversity. In fact, $A \rightarrow B \rightarrow A$ does not necessarily need to be restored to exactly the same A.
  - Toward Multimodal Image-to-Image Translation

- Some related work
  - Unsupervised Attention-guided Image to Image Translation
  - Unsupervised Image-to-Image Translation Using Domain-Specific Variational Information Bound
  - Image-to-image translation for cross-domain disentanglement

Some Training Details

- replace the negative log likelihood objective by a least square loss, This loss performs more stably during training and generates higher quality results. $L_{GAN}(\mathrm{G}, D_Y, \mathrm{X}, \mathrm{Y})$ then becomes:

$$L_{LSGAN}(\mathrm{G}, D_Y, \mathrm{X}, \mathrm{Y}) = E_{y \sim p_{data}(y)}[(D_Y(y) - 1)^2] + E_{x \sim p_{data}(x)}[D_Y(G(x))^2]$$

$L_{GAN}(\mathrm{F}, D_X, \mathrm{Y}, \mathrm{X})$ also made the same change

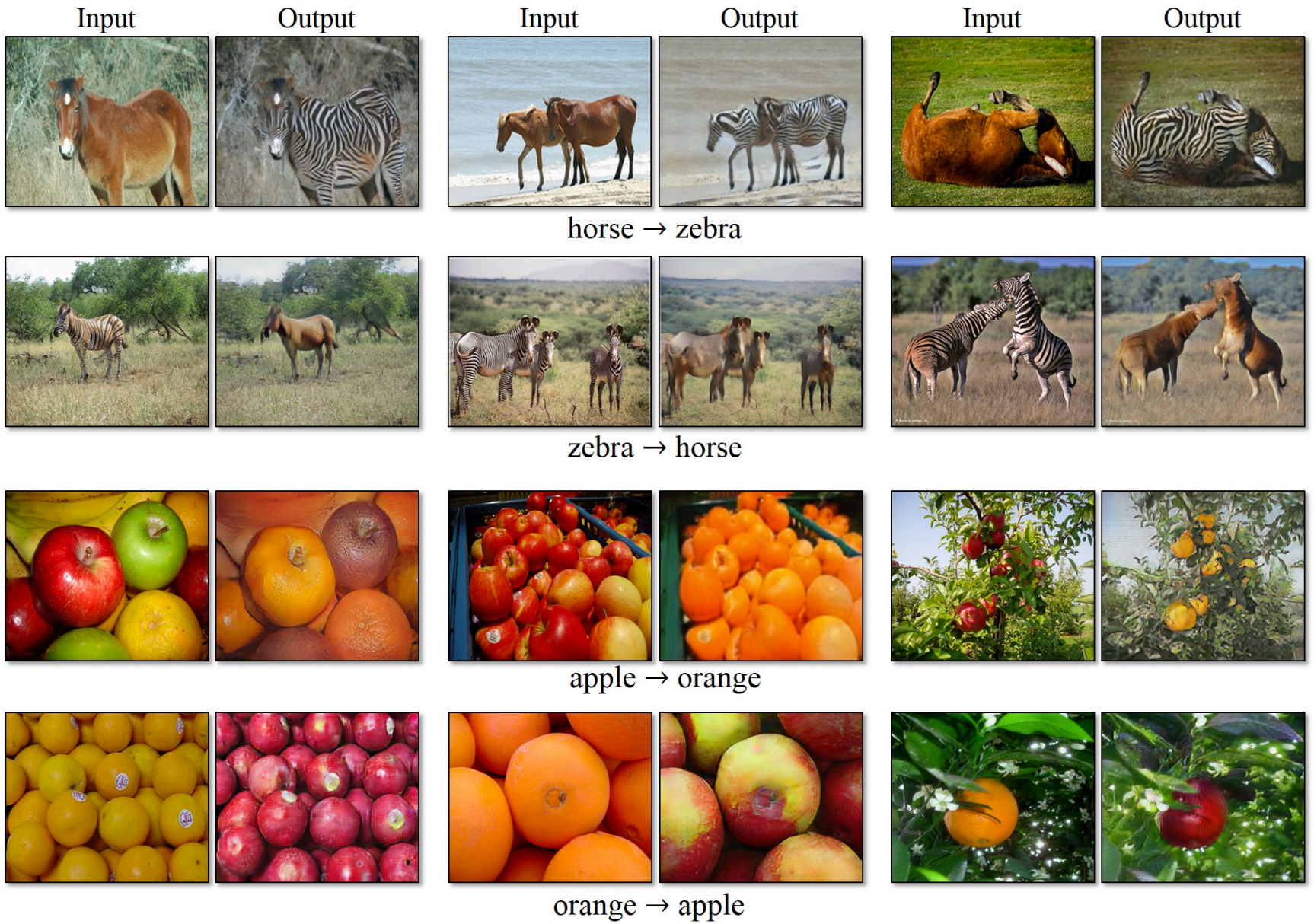
- to reduce model oscillation, then update the discriminators D_X and D_Y using a history of generated images rather than the ones produced by the latest generative networks.
  - This idea comes from: Learning from Simulated and Unsupervised Images through Adversarial Training

- For painting→photo, the paper find that it is helpful to introduce an additional loss to encourage the mapping to preserve color composition between the input and output. It regularize the generator to be near an identity mapping when real samples of the target domain are provided as the input to the generator:

$$L_{identity}(\mathrm{G}, F) = E_{y \sim p_{data}(y)}[\, \|G(y) - y\|_1 \,] + E_{x \sim p_{data}(x)}[\, \|F(x) - x\|_1 \,]$$

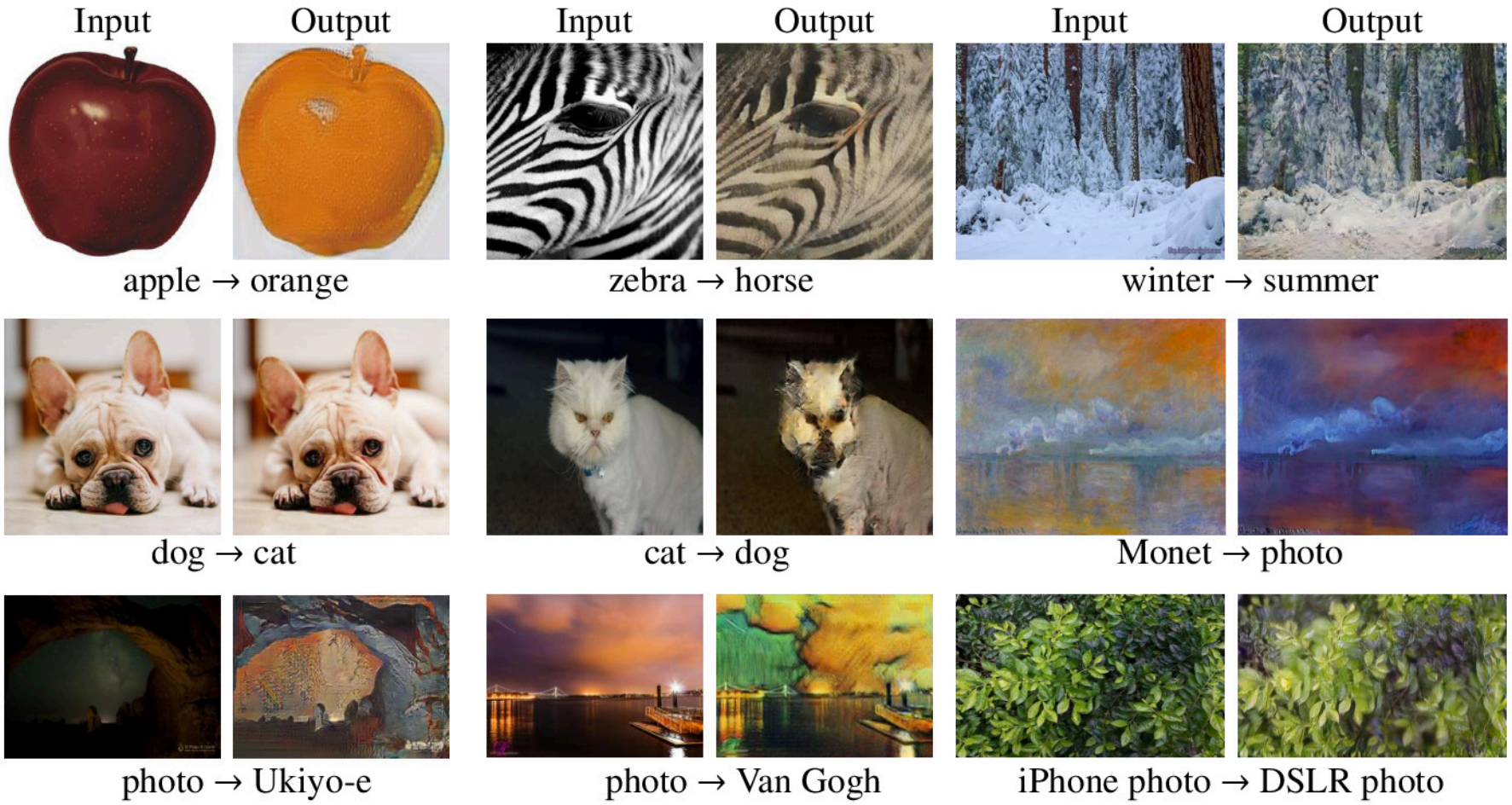
  - This idea comes from:Unsupervised Cross-Domain Image Generation

Results: Some Successful Cases



These are examples of cross domain image transfer - we want to take an image from an input domain D_i and then transform it into an image of target domain D_t without necessarily having a one-to-one mapping between images from input to target domain in the training set.

Results: Some Failure Cases



Quoting from the original paper:
We have also explored tasks that require geometric changes, with little success. For example, on the task of $dog \leftrightarrow cat$ transfiguration, the learned translation degenerates into making minimal changes to the input. Handling more varied and extreme transformations, especially geometric changes, is an important problem for future work.

Thanks!