# Adversarial learning

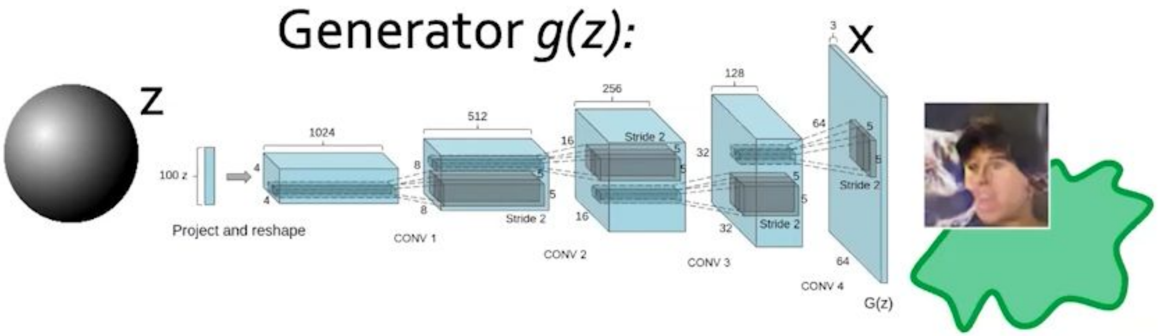
GAN

# Как поставлена задача

Есть: набор(рапределение) картинок, из которого мы можем брать сэмплы

Нужно: научиться генерировать новые сэмплы

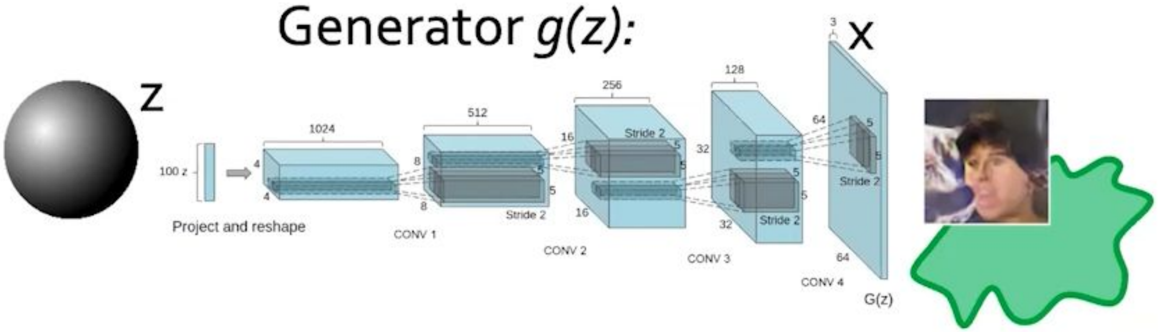
# Generative Adversarial Network




[Goodfellow et al. 14]

На вход получает вектор, на выходе картинка

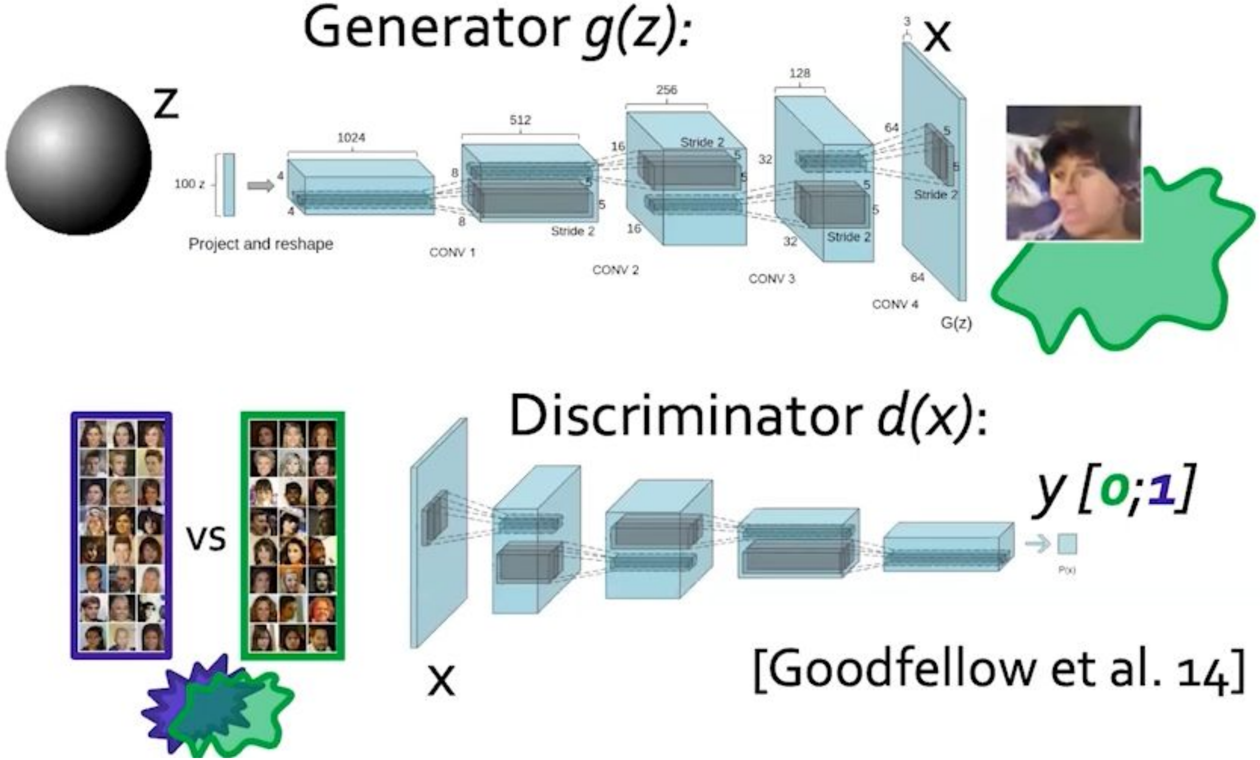
# Generative Adversarial Network



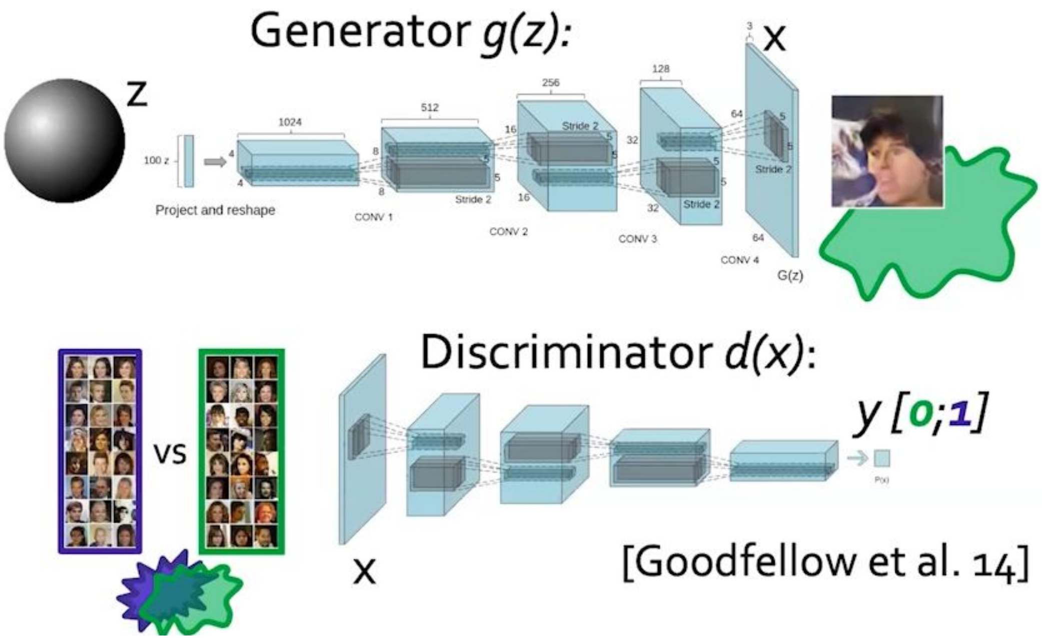

[Goodfellow et al. 14]

Какие могут быть архитектуры у генератора?

# Generative Adversarial Network



# Generative Adversarial Network




Дискриминатор обучается вместе с генератором, учится отличать реальные сэмплы от сгенерированных.

# Обучение моделируется игрой

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

Маленькими буквами d, g обозначены параметры дискриминатора, генератора

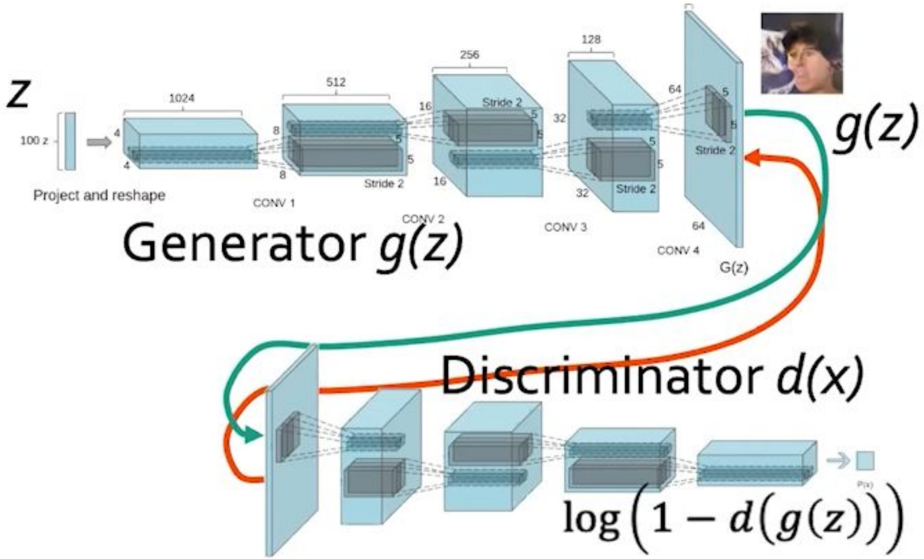
X - распределение картинок

Z - априорное нормальное распределение

# Обучение моделируется игрой

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x\sim X}[\log d(x)] + \mathbb{E}_{z\sim Z}[\log(1 - d(g(z)))]$$



# Обучение моделируется игрой

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

Какая часть относится к генератору?

# Обучение моделируется игрой

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

Какая часть относится к генератору? Почему 1-d(g(z))?

# Обучение моделируется игрой

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

Какая часть относится к генератору? Почему 1-d(g(z))?

Рассматриваем проблему с другой точки зрения, когда генератор стремится максимизировать вероятность того, что изображения являются реальными, вместо того, чтобы минимизировать вероятность того, что изображение является сгенерированным.

# The problem of perfect generator

$$\min_g \mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

$$\max_g \mathbb{E}_{z \sim Z}[\log d(g(z))]$$

What is the optimal generator for a fixed discriminator?

# Perfect discriminator with fixed generator

### 4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider the optimal discriminator $D$ for any given generator $G$.

**Proposition 1.** *For $G$ fixed, the optimal discriminator $D$ is*

$$D^*_G(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \tag{2}$$

*Proof.* The training criterion for the discriminator D, given any generator $G$, is to maximize the quantity $V(G, D)$

$$\begin{aligned} V(G, D) &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x}))dx + \int_z p_{\boldsymbol{z}}(\boldsymbol{z}) \log(1 - D(g(\boldsymbol{z})))dz \\ &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x}))dx \end{aligned} \tag{3}$$

For any $(a, b) \in \mathbb{R}^2 \setminus \{0, 0\}$, the function $y \rightarrow a \log(y) + b \log(1 - y)$ achieves its maximum in $[0, 1]$ at $\frac{a}{a+b}$. The discriminator does not need to be defined outside of $Supp(p_{\text{data}}) \cup Supp(p_g)$, concluding the proof. □

link

# Perfect discriminator with fixed generator

## Optimal discriminator

Let us derive optimal *d(x)* given $p_{\text{data}}(x),\ p_{\text{gen}}(x)$

$$\min_g \max_d V(d,g) =$$

$$\mathbb{E}_{x\sim X}[\log d(x)] + \mathbb{E}_{z\sim Z}[\log(1-d(g(z)))]$$

$$\frac{\delta V(d,g)}{\delta d(x)} = \frac{\delta[p_{\text{data}}(x)\log d(x) + p_{\text{gen}}(x)\log(1-d(x))]}{\delta d(x)}$$

$$= \frac{p_{\text{data}}(x)}{d(x)} - \frac{p_{\text{gen}}(x)}{1-d(x)} \qquad = 0$$

$$(1-d(x))p_{\text{data}}(x) - d(x)p_{\text{gen}}(x) = 0$$

$$d(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{gen}}(x)}$$

d(x) будет стремится к ½ если мы научились генерировать правдоподобные картинки

# The problem of perfect generator

$$\min_g \mathbb{E}_{x\sim X}[\log d(x)] + \mathbb{E}_{z\sim Z}[\log(1 - d(g(z)))]$$

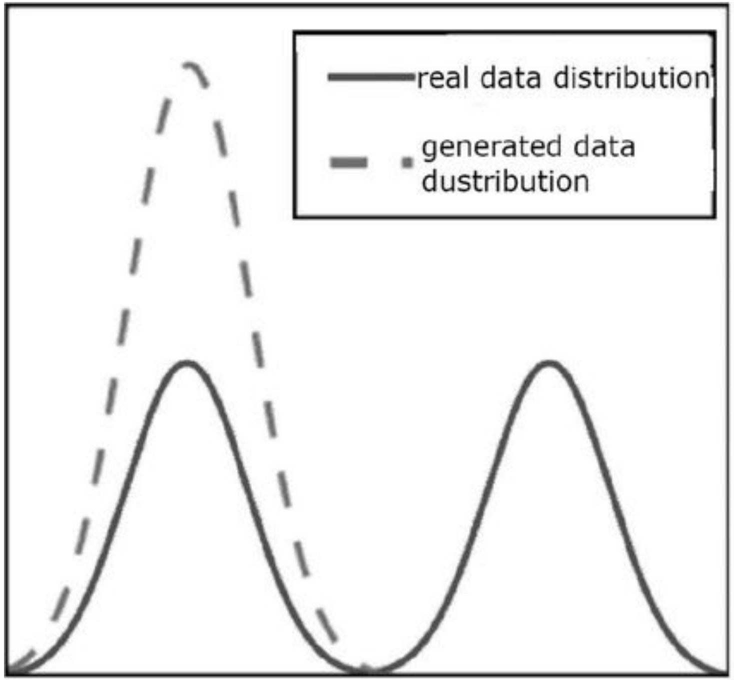
$$\max_g \mathbb{E}_{z\sim Z}[\log d(g(z))]$$

What is the optimal generator for a fixed discriminator?

Answer: $g(z) = \arg\max_x d(x)$

Main source of instability ("mode collapse").

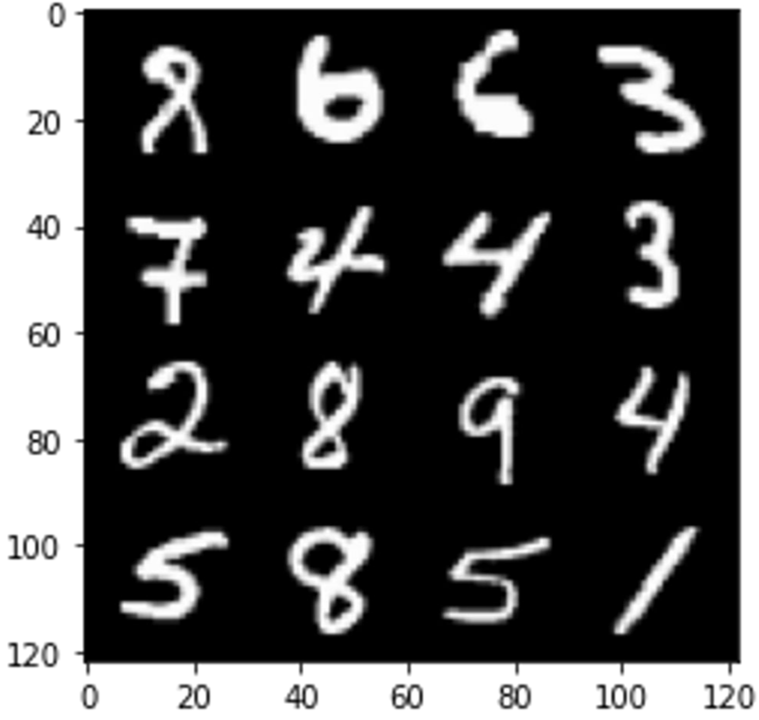
# Mode collapse



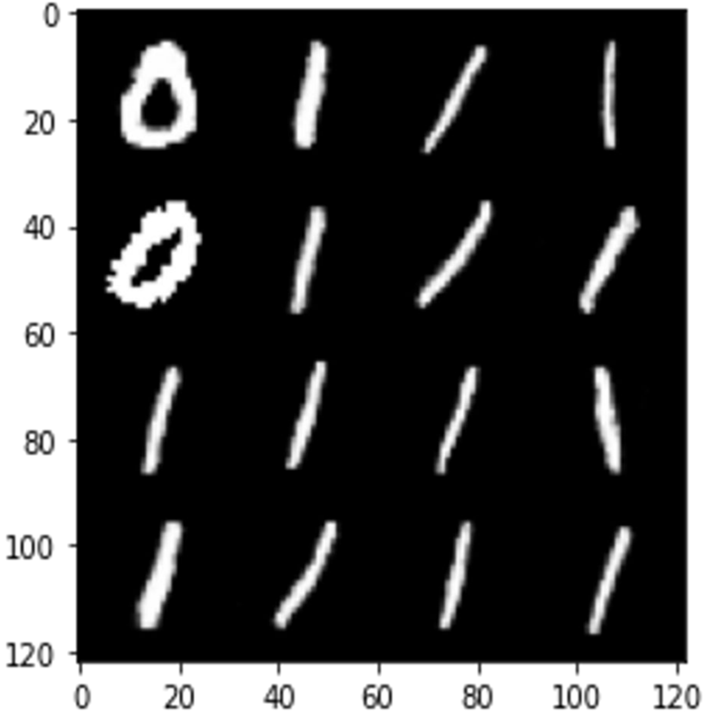

(a)

(b)

# Mode collapse




Train Data Point




GAN Generated Data Point

# One more problem

$$\min_g \max_d V(d, g) =$$
$$\mathbb{E}_{x\sim X}[\log d(x)] + \mathbb{E}_{z\sim Z}[\log(1 - d(g(z)))]$$

Problem: generator gets *vanishing gradient* when the discriminator is much smarter.

Discriminator learns faster, and it can stop training of the generator because of vanishing gradients. (at the end of the discriminator usually stays sigmoid with really small gradient and the tails).

# One more problem: solution

$$\min_g \max_d V(d, g) =$$

$$\mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

Problem: generator gets *vanishing gradient* when the discriminator is much smarter.
Therefore, non-zero sum formulation is popular:

$$\max_d \mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

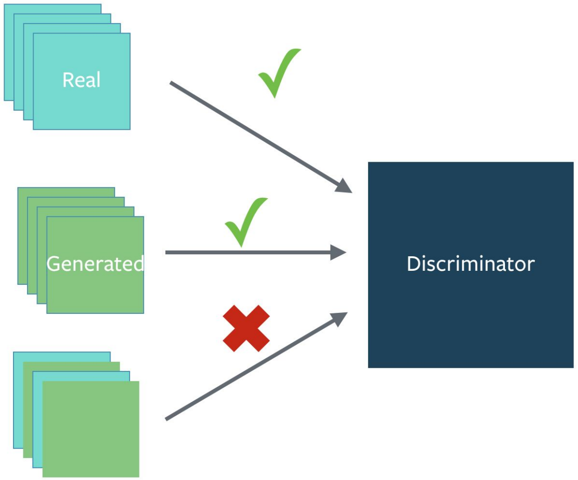
$$\max_g \mathbb{E}_{z \sim Z}[\log d(g(z))]$$

*[Goodfellow et al. 14]*

# GAN hacks

https://github.com/soumith/ganhacks

- Avoid Sparse Gradients: ReLU, MaxPool
- LeakyReLU = good (in both G and D)
- Construct different mini-batches for real and fake,
- i.e. each mini-batch needs to contain only all real images or all generated images.
- Dont sample from a Uniform distribution
- Sample from a gaussian distribution
- In GAN papers, the loss function to optimize G is min (log 1-D), but in practice folks practically use max log normalize the images between -1 and 1
- Tanh as the last layer of the generator output



TODO какие проблемы у loss’a

# Other losses

## Probability distances

Яндекс

- Kullback–Leibler divergence

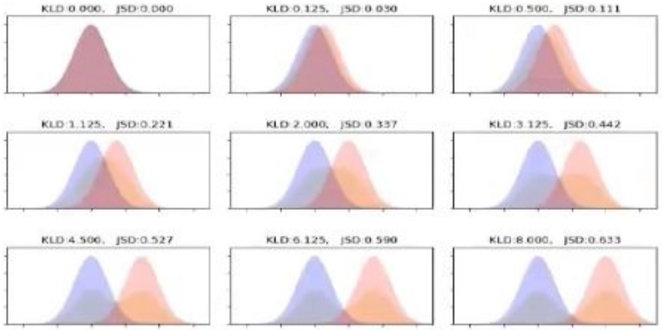
$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$

- Jensen–Shannon divergence

$$\mathrm{JSD}(P \parallel Q) = \frac{1}{2} D(P \parallel M) + \frac{1}{2} D(Q \parallel M),$$

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

Solution: use other probability distances!
[M. Arjovsky at al, 2017]




**JS** : symmetric, -> **ln2** if distributions are totally mismatched
**KL**: asymmetric -> **inf** if distributions are totally mismatched

# Other losses. EMD

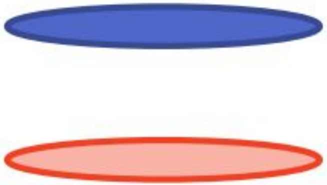
## Alignment through EMD

$$\min_g W(X \| g(Z))$$

Earth mover distance (aka Wasserstein-1 distance):

$$W(X\|Y) = \inf_{\gamma \in \Pi(X,Y)} \mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$$

All random variables over
with marginal distributions X and Y

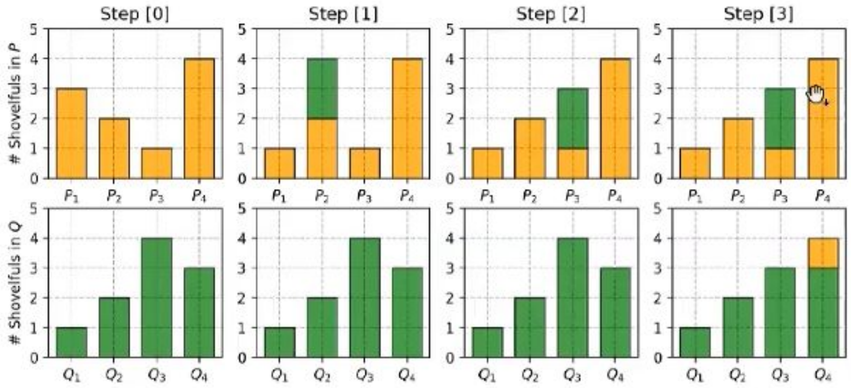


[Arjovsky et al 2017]

# EMD

## Earth mover distance

Яндекс




In order to change P to look like Q, as illustrated in Fig. 7, we:

- First move 2 shovelfuls from P1 to P2 => (P1,Q1) match up.
- Then move 2 shovelfuls from P2 to P3 => (P2,Q2) match up.
- Finally move 1 shovelfuls from Q3 to Q4 => (P3,Q3) and (P4,Q4) match up.

the Earth Mover's distance is W=5

# Wasserstein GAN

$$\min_g W(X \| g(Z))$$

$$W(X\|Y) = \inf_{\gamma \in \Pi(X,Y)} \mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$$

Kantorovich-Rubinstein duality:

$$W(X\|Y) = \sup_{\|f\|_L \le 1} \left(\mathbb{E}_{x\sim X}[f(x)] - \mathbb{E}_{x\sim Y}[f(x)]\right)$$

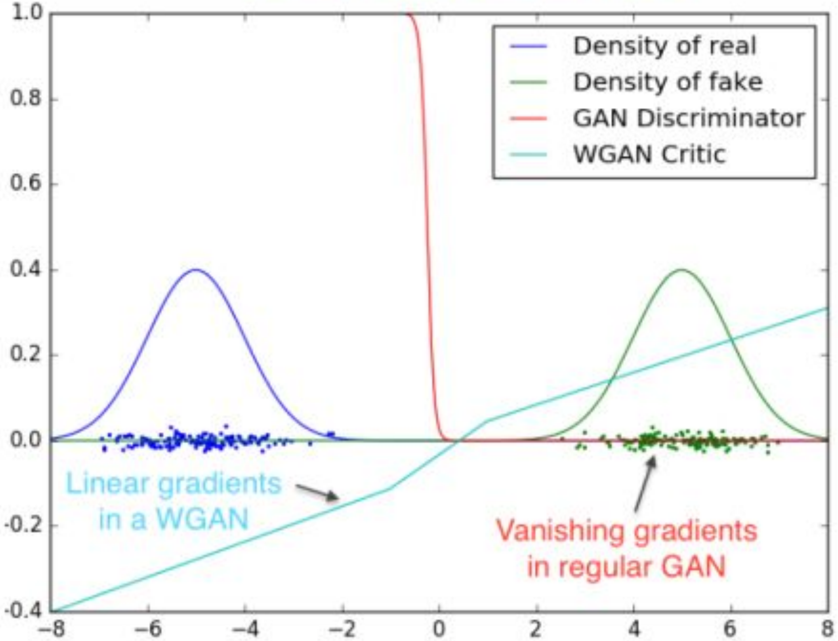
Wasserstein GAN [Arjovsky et al 2017]:

$$\min_g \max_{\|f\|_L \le 1} \left(\mathbb{E}_{x\sim X}[f(x)] - \mathbb{E}_{z\sim Z}[f(g(z))]\right)$$

(compare with the original GAN):

$$\min_g \max_d \mathbb{E}_{x\sim X}[\log d(x)] + \mathbb{E}_{z\sim Z}[\log(1-d(g(z)))]$$

# WGAN vs GAN

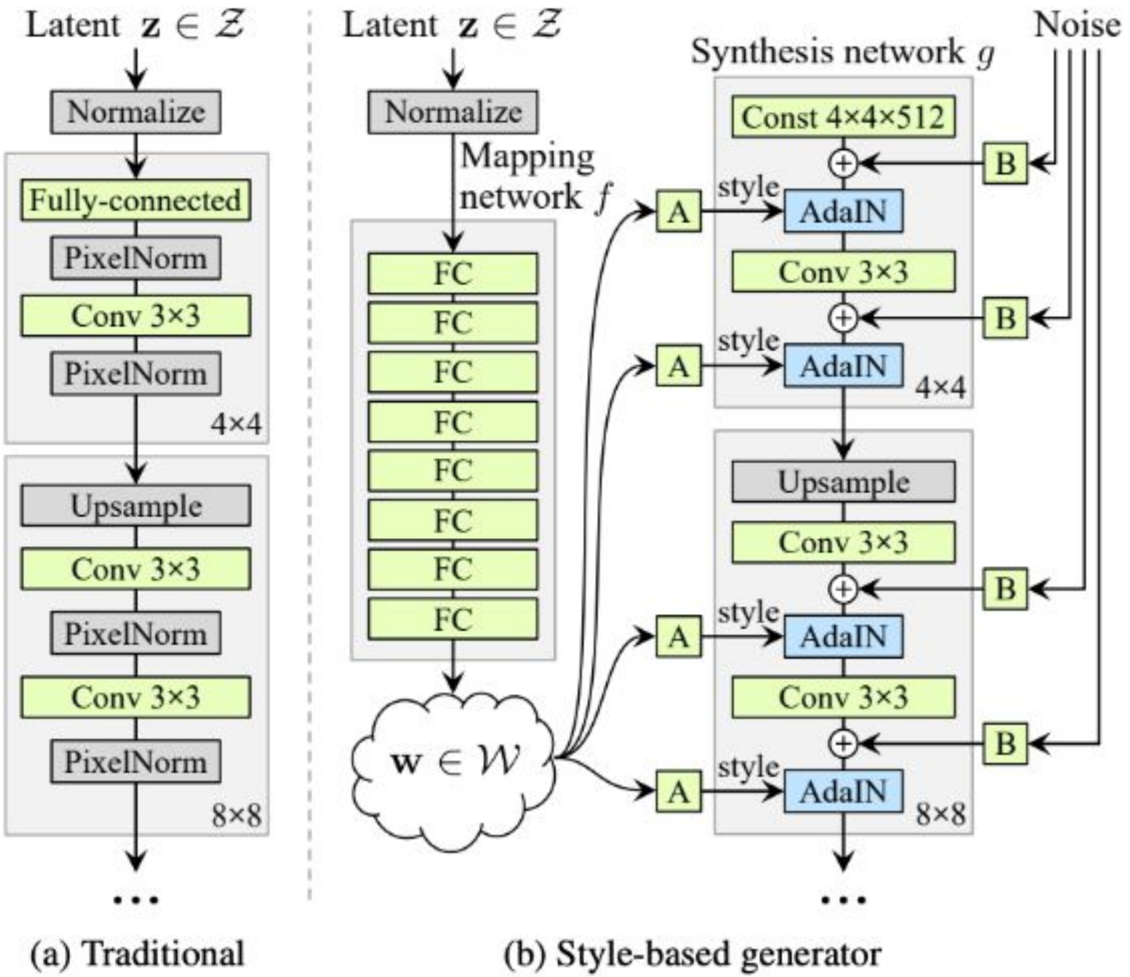



$$\max_{\|f\|_L \leq 1} \left( \mathbb{E}_{x \sim X}[f(x)] - \mathbb{E}_{z \sim Z}[f(g(z))] \right)$$

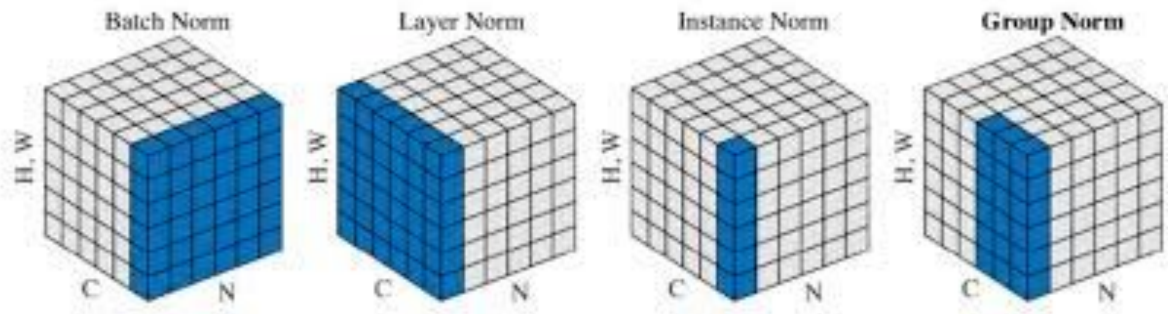
$$\max_{d} \mathbb{E}_{x \sim X}[\log d(x)] + \mathbb{E}_{z \sim Z}[\log(1 - d(g(z)))]$$

# StyleGAN

Figure 1. While a traditional generator [30] feeds the latent code though the input layer only, we first map the input to an intermediate latent space $\mathcal{W}$, which then controls the generator through adaptive instance normalization (AdaIN) at each convolution layer. Gaussian noise is added after each convolution, before evaluating the nonlinearity. Here "A" stands for a learned affine transform, and "B" applies learned per-channel scaling factors to the noise input. The mapping network $f$ consists of 8 layers and the synthesis network $g$ consists of 18 layers — two for each resolution ($4^2 - 1024^2$). The output of the last layer is converted to RGB using a separate $1 \times 1$ convolution, similar to Karras et al. [30]. Our generator has a total of 26.2M trainable parameters, compared to 23.1M in the traditional generator.



# AdaIN




```python
def get_mean_std(x, epsilon=1e-5):
    axes = [1, 2]

    # Compute the mean and standard deviation of a tensor.
    mean, variance = tf.nn.moments(x, axes=axes, keepdims=True)
    standard_deviation = tf.sqrt(variance + epsilon)
    return mean, standard_deviation


def ada_in(style, content):
    """Computes the AdaIn feature map.

    Args:
        style: The style feature map.
        content: The content feature map.

    Returns:
        The AdaIN feature map.
    """
    content_mean, content_std = get_mean_std(content)
    style_mean, style_std = get_mean_std(style)
    t = style_std * (content - content_mean) / content_std + style_mean
    return t
```

spaces to 512, and the mapping $f$ is implemented using an 8-layer MLP, a decision we will analyze in Section 4.1. Learned affine transformations then specialize $\mathbf{w}$ to *styles* $\mathbf{y} = (\mathbf{y}_s, \mathbf{y}_b)$ that control adaptive instance normalization (AdaIN) [27, 17, 21, 16] operations after each convolution layer of the synthesis network $g$. The AdaIN operation is defined as

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}, \qquad (1)$$

where each feature map $\mathbf{x}_i$ is normalized separately, and then scaled and biased using the corresponding scalar components from style $\mathbf{y}$. Thus the dimensionality of $\mathbf{y}$ is twice the number of feature maps on that layer.

# StyleGAN

# Are GANs perfect latent models?

Real

Sample

PRGAN reconstructions

[thanks to ShahRukh Athar]

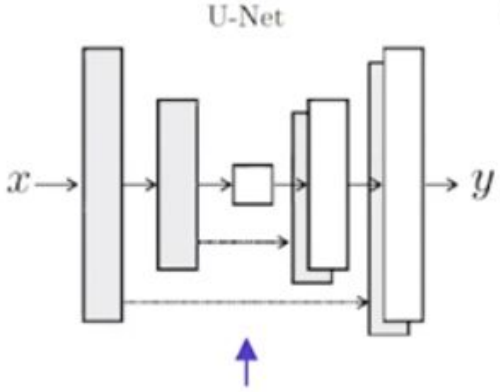
$$g(\arg\min_z \|g(z) - x\|^2)$$

- Reconstructing even training set image may lead to a large gap

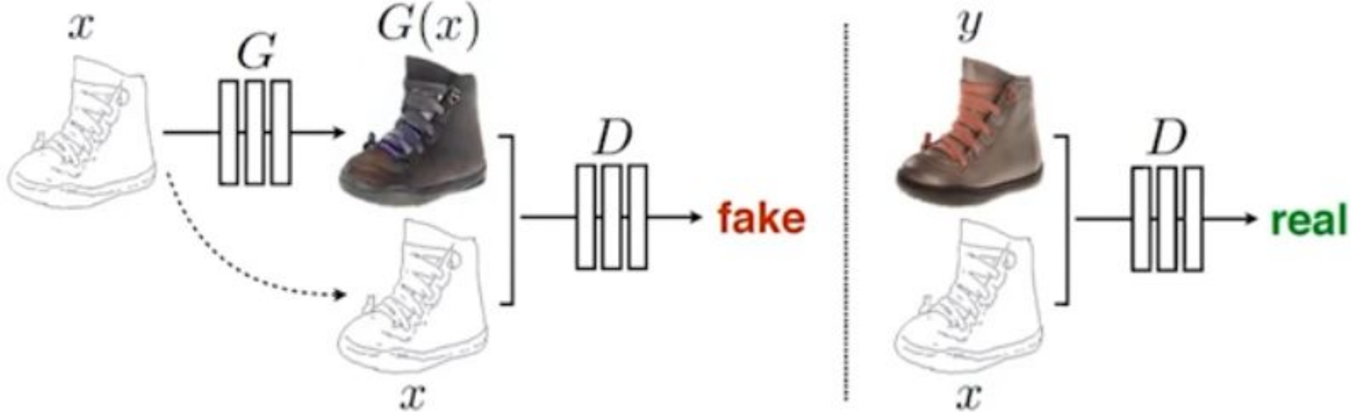
# Pix2pix: an image-conditional GAN


[Isola et al. 2017]

# Pix2pix: a conditional GAN

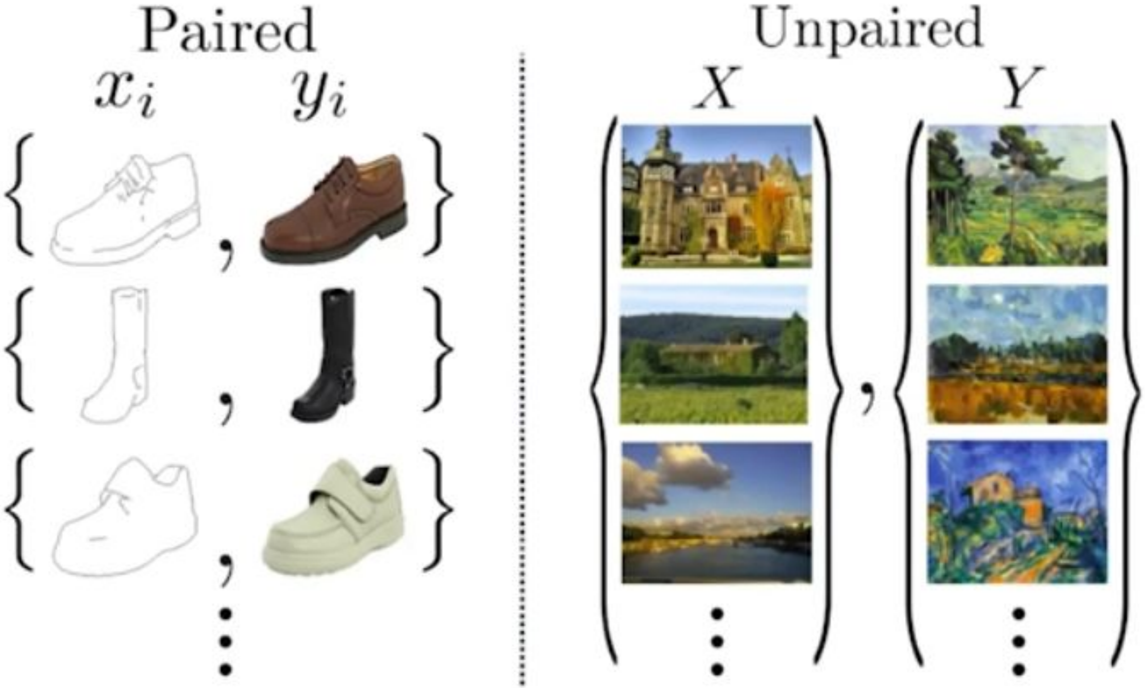



- Use a combination of L1-loss and adversarial loss
- Adversarial loss is computed for patches ("PatchGAN")



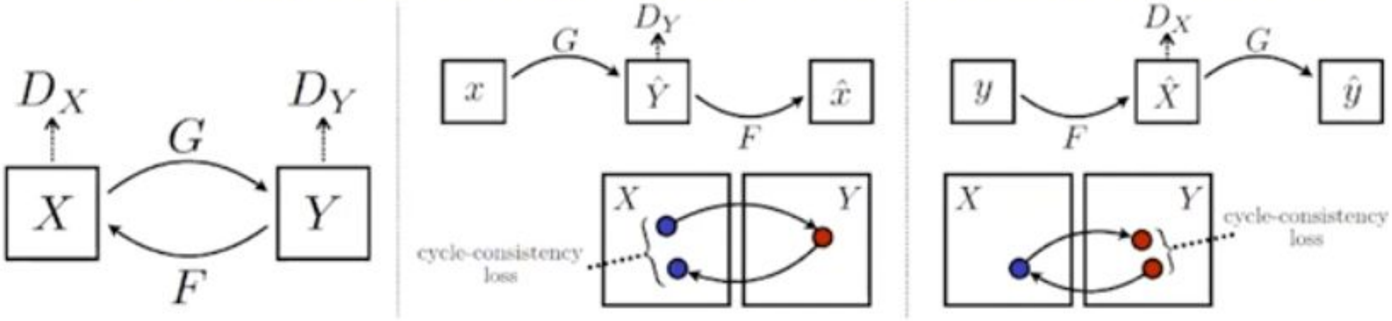

[Isola et al. 2017]

# Unaligned image translation: CycleGAN




[Zhu et al. ICCV 2017]

# Unaligned image translation: CycleGAN



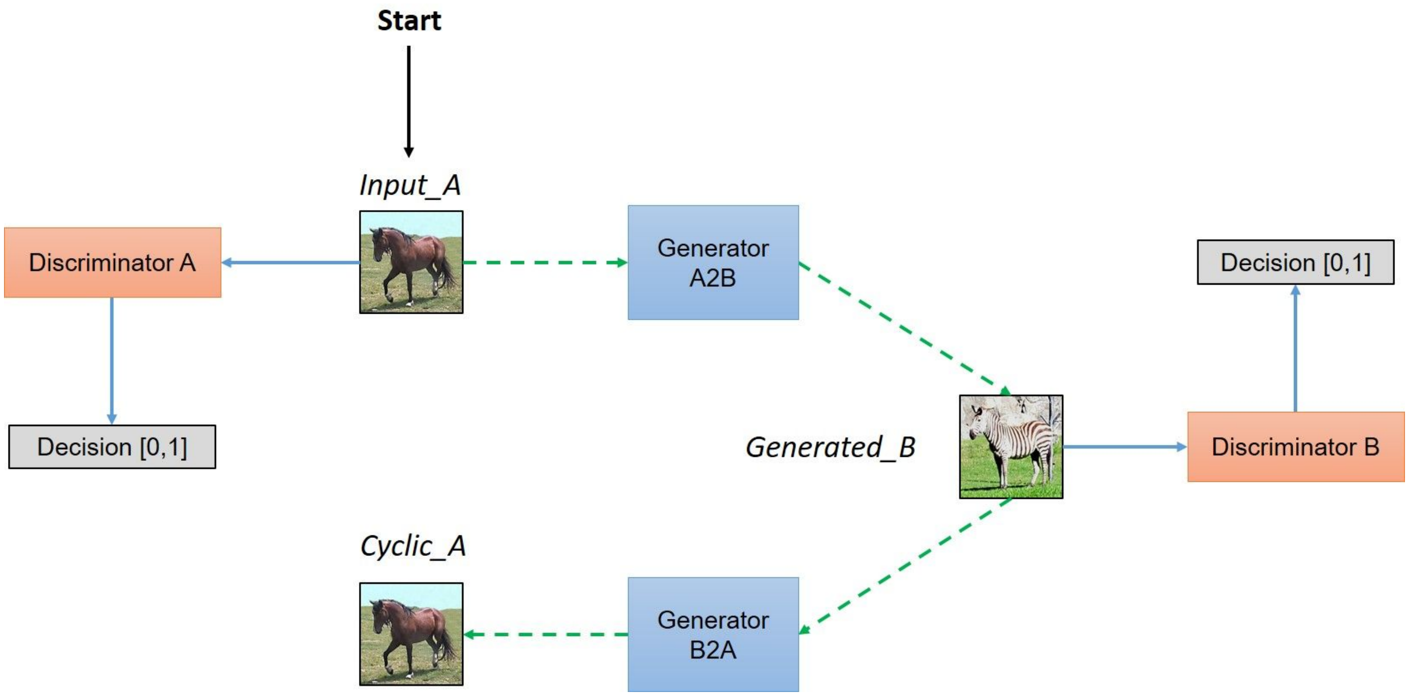

$$L_{\text{adv}} = \mathbb{E}_{x \sim X}\left[\sum_p -\log D_y(G(x)[p])\right]$$

$$L_{\text{cycle}} = \mathbb{E}_{x \sim X}\left[\|F(G(x)) - x\|_1\right]$$

$$L_{\text{identity}} = \mathbb{E}_{y \sim Y}\left[\|G(y) - y\|^2\right]$$

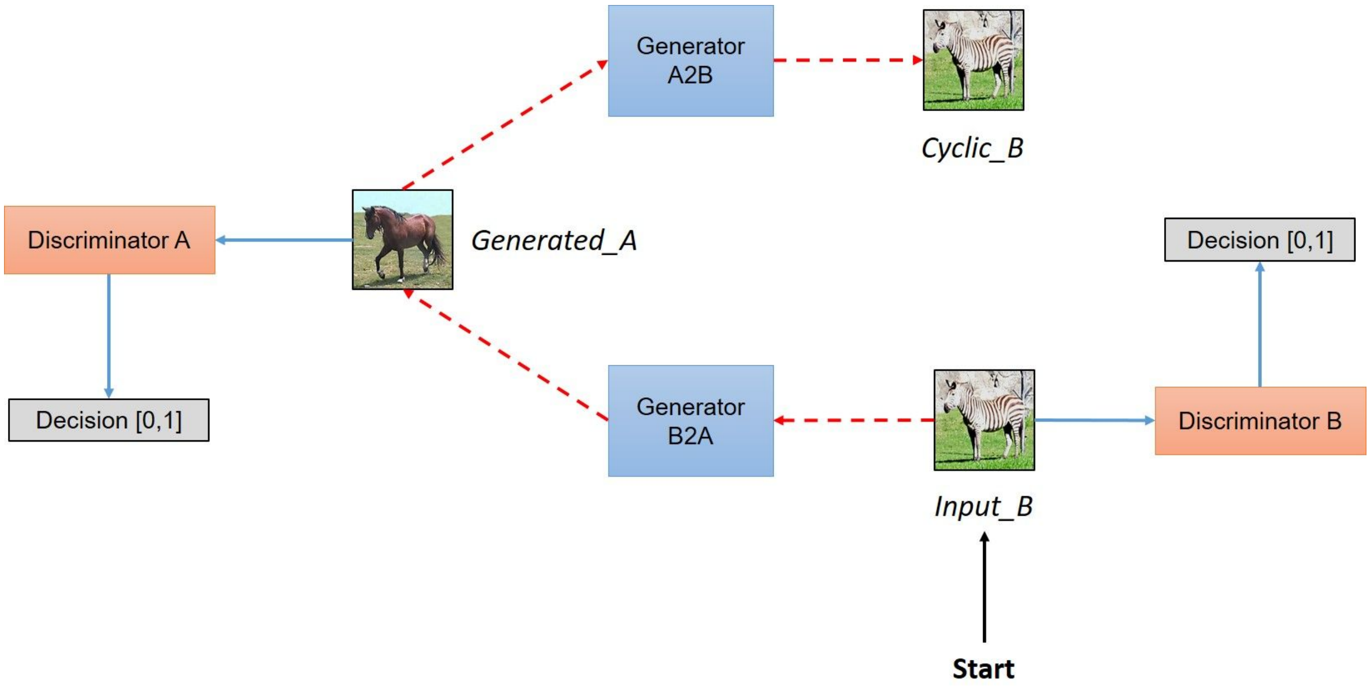
[Zhu et al. ICCV 2017]

# CycleGAN



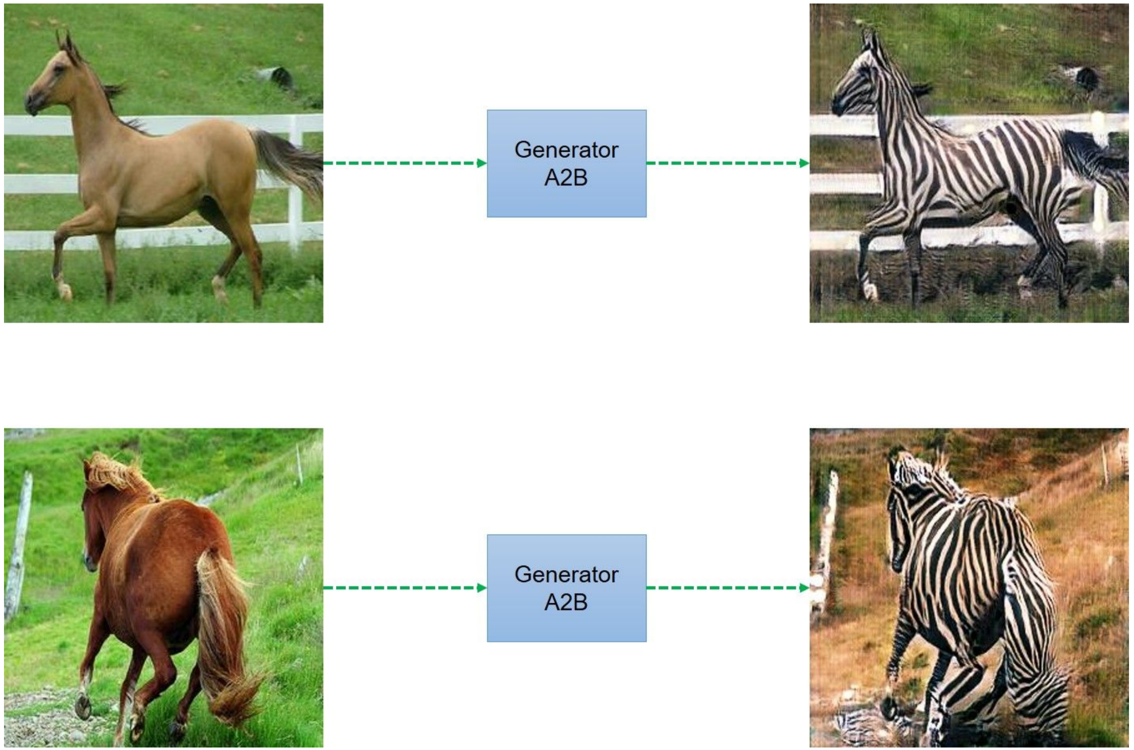
Start
Input_A
Discriminator A
Generator
A2B
Decision [0,1]
Decision [0,1]
Generated_B
Discriminator B
Cyclic_A
Generator
B2A

# CycleGAN



Generator A2B
Cyclic_B
Discriminator A
Generated_A
Decision [0,1]
Decision [0,1]
Generator B2A
Discriminator B
Input_B
Start

# CycleGAN



# CycleGAN loss

By now we have two generators and two discriminators
design the loss function in a way which accomplishes c
function can be seen having four parts:

1. Discriminator must approve all the original images o
corresponding categories.

2. Discriminator must reject all the images which are ge
corresponding Generators to fool them.

3. Generators must make the discriminators approve a
images, so as to fool them.

4. The generated image must retain the property of orig
we generate a fake image using a generator say
GeneratorA→BGeneratorA→B then we must be able t
original image using the another generator
GeneratorB→AGeneratorB→A - it must satisfy cyclic-c

***Part 1***

Discriminator must be trained such that recommendation for images from category A must be as close to 1, and vice versa for discriminator B. So Discriminator A would like to minimize $(Discriminator_A(a) - 1)^2$ and same goes for B as well. This can be implemented as:

```
D_A_loss_1 = tf.reduce_mean(tf.squared_difference(dec_A,1))
D_B_loss_1 = tf.reduce_mean(tf.squared_difference(dec_B,1))
```

***Part 2***

Since, discriniator should be able to distinguish between generated and original images, it should also be predicting 0 for images produced by the generator, i.e. Discriminator A wwould like to minimize $(Discriminator_A(Generator_{B\to A}(b)))^2$. It can be calculated as follow:

```
D_A_loss_2 = tf.reduce_mean(tf.square(dec_gen_A))
D_B_loss_2 = tf.reduce_mean(tf.square(dec_gen_B))

D_A_loss = (D_A_loss_1 + D_A_loss_2)/2
D_B_loss = (D_B_loss_1 + D_B_loss_2)/2
```

***Generator loss***

Generator should eventually be able to fool the discriminator about the authencity of it's generated images. This can done if the recommendation by discriminator for the generated images is as close to 1 as possible. So generator would like to minimize $(Discriminator_B(Generator_{A\to B}(a)) - 1)^2$ So the loss is:

```
g_loss_B_1 = tf.reduce_mean(tf.squared_difference(dec_gen_A,1))
g_loss_A_1 = tf.reduce_mean(tf.squared_difference(dec_gen_A,1))
```

***Cyclic loss***

And the last one and one of the most important one is the cyclic loss that captures that we are able to get the image back using anot
generator and thus the difference between the original image and the cyclic image should be as small as possible.

```
cyc_loss = tf.reduce_mean(tf.abs(input_A-cyc_A)) + tf.reduce_mean(tf.abs(input_B-cyc_B))
```

The complete generator loss is then:

```
g_loss_A = g_loss_A_1 + 10*cyc_loss
g_loss_B = g_loss_B_1 + 10*cyc_loss
```

# GAN metrics. FID vs IS

```python
def calculate_inception_score(p_yx, eps=1E-16):
    # calculate p(y)
    p_y = expand_dims(p_yx.mean(axis=0), 0)
    # kl divergence for each image
    kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
    # sum over classes
    sum_kl_d = kl_d.sum(axis=1)
    # average over images
    avg_kl_d = mean(sum_kl_d)
    # undo the logs
    is_score = exp(avg_kl_d)
    return is_score
```

## Inception score

The inception score is calculated by first using a pre-trained Inception v3 model to predict the class probabilities for each generated image.

*Images that contain meaningful objects should have a conditional label distribution p(y|x) with low entropy.*

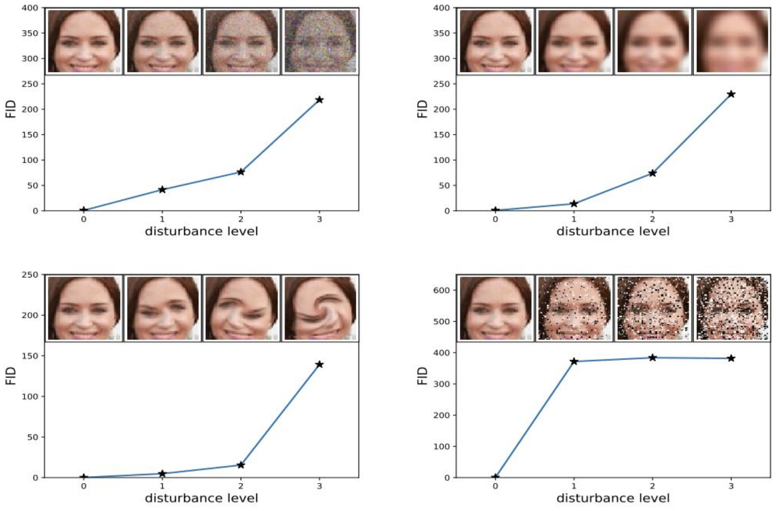
The entropy is calculated as the negative sum of each observed probability multiplied by the log of the probability. The intuition here is that large probabilities have less information than small probabilities.

*Moreover, we expect the model to generate varied images, so the marginal integral p(y|x = G(z))dz should have high entropy.*

Calculating the divergence between two distributions is written using the “||” operator, therefore we can say we are interested in the KL divergence between C for conditional and M for marginal distributions or:

- KL divergence = p(y|x) * (log(p(y|x)) – log(p(y)))

# GAN metrics. FID vs IS




## Frechet Inception Distance

This output layer has 2,048 activations, therefore, each image is p
called the coding vector or feature vector for the image.

The FID score is then calculated using the following equation taken from the paper:

- d^2 = ||mu_1 – mu_2||^2 + Tr(C_1 + C_2 – 2*sqrt(C_1*C_2))

The “*mu_1*” and “*mu_2*” refer to the feature-wise mean of the real and generated images, e.g. 2,048 element vectors where each element is the mean feature observed across the images.

The *C_1* and *C_2* are the covariance matrix for the real and generated feature vectors, often referred to as sigma.