HCMUTE

Artificial Neural Network

Dr. Trần Vũ Hoàng

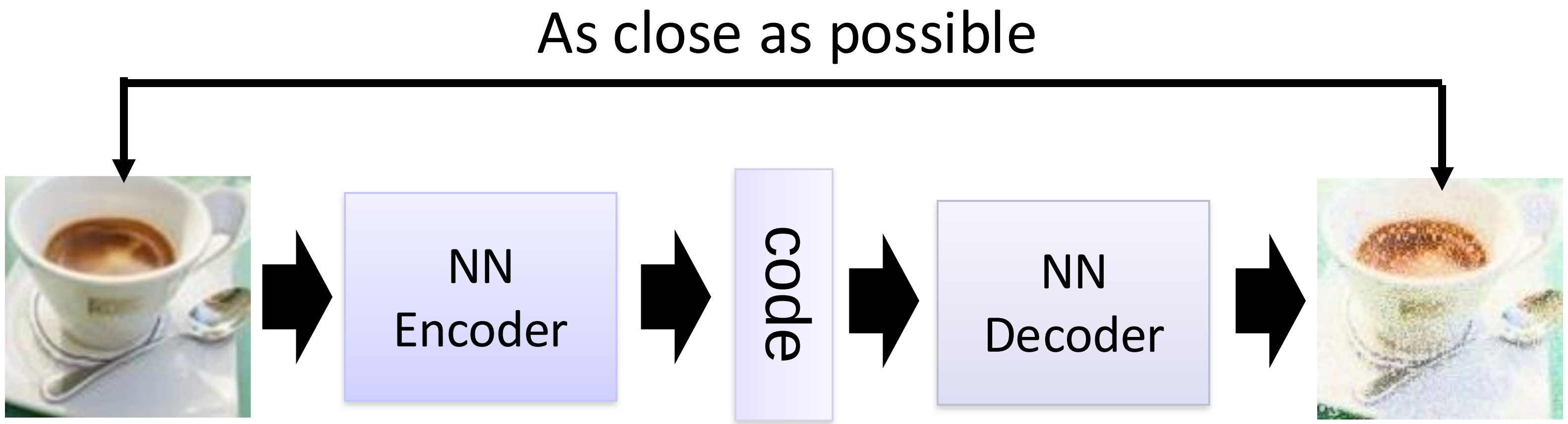
Generative Adversarial Network

- GAN was first introduced by Ian Goodfellow et al in 2014
- Have been used in generating images, videos, poems, some simple conversation.
- Note, image processing is easy (all animals can do it), NLP is hard (only human can do it).
- This co-evolution approach might have far-reaching implications. Bengio: this may hold the key to making computers a lot more intelligent.
- Ian Goodfellow:

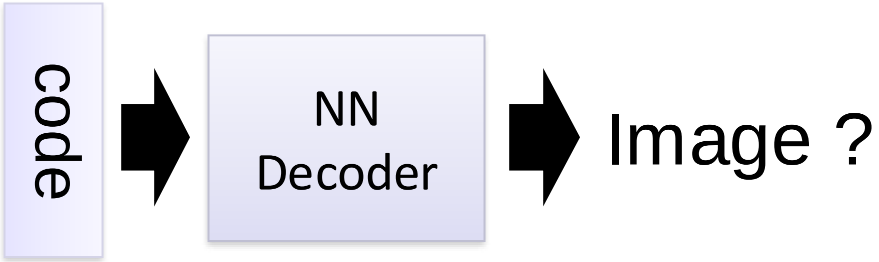
  https://www.youtube.com/watch?v=YpdP_0-IEOw
- Radford, (generate voices also here)

  https://www.youtube.com/watch?v=KeJINHjyzOU
- Tips for training GAN: https://github.com/soumith/ganhacks

Autoencoder

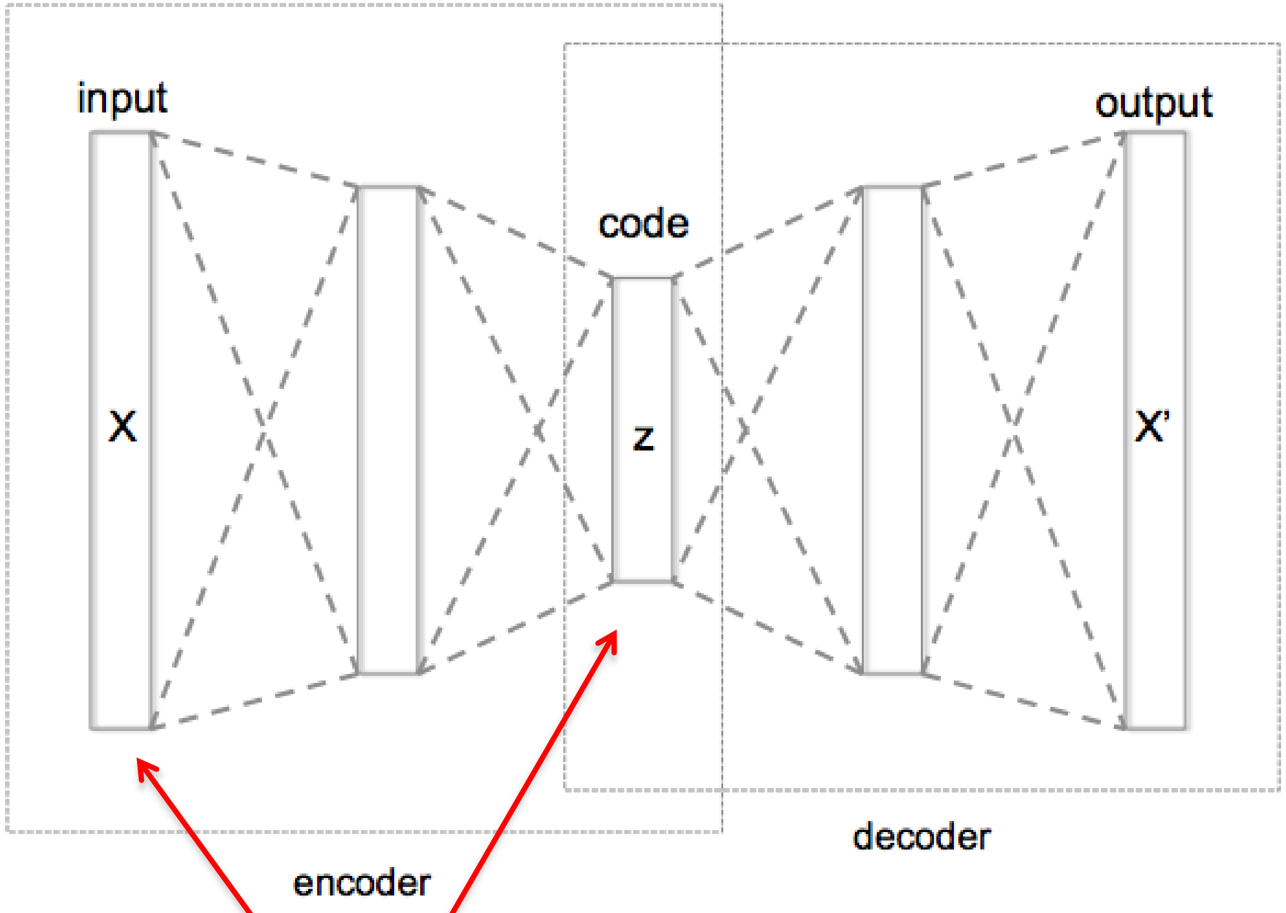


Randomly generate a vector as code



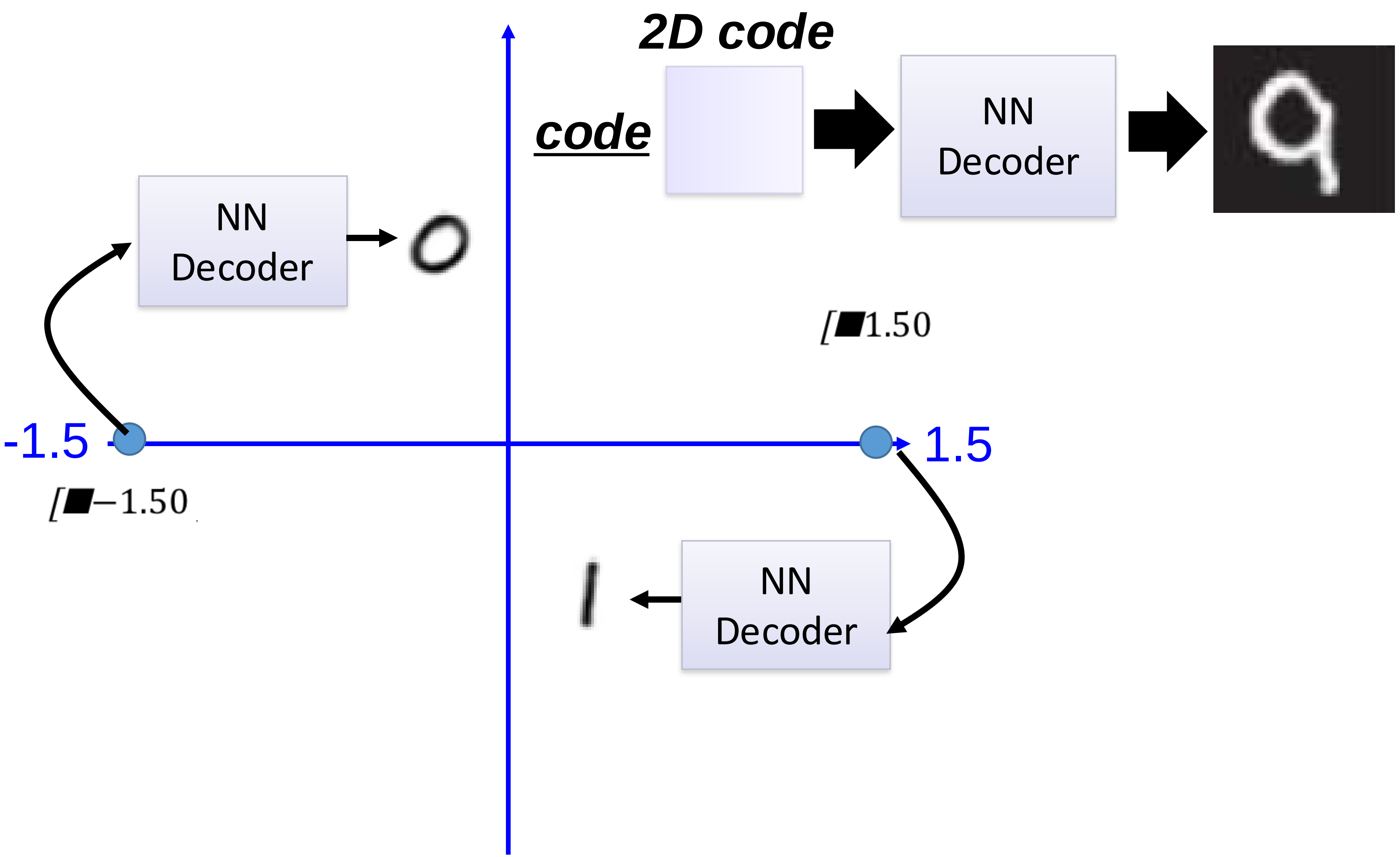
Autoencoder with 3 fully connected layers

Training: model.fit(X,X)
Cost function: $\Sigma_{k=1..N} (x_k - x'_k)^2$

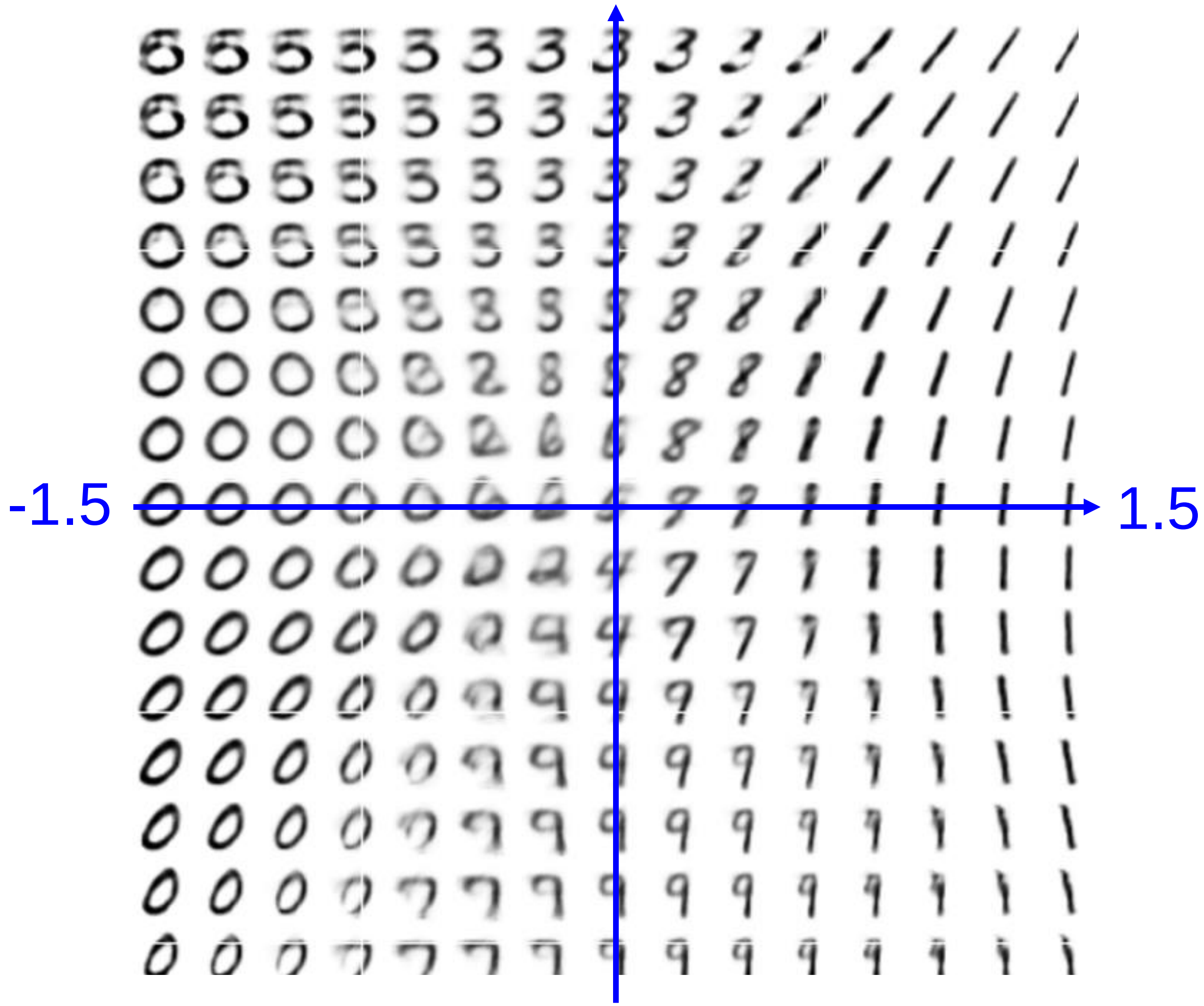


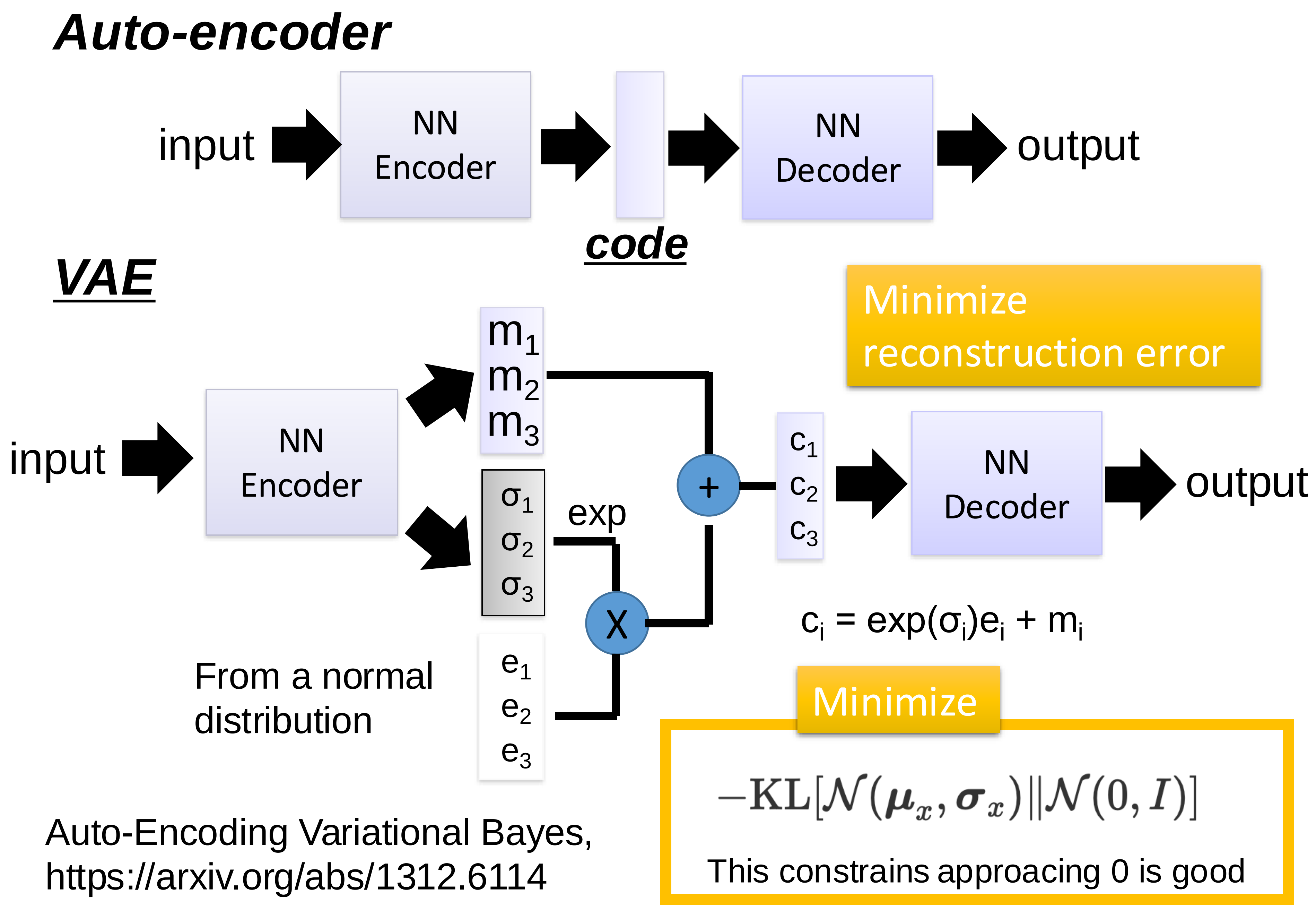
Large → small, learn to compress

Auto-encoder



Auto-encoder




Auto-encoder
input
NN Encoder
code
NN Decoder
output
VAE
input
NN Encoder
m_1 m_2 m_3
σ_1 σ_2 σ_3
exp
From a normal distribution
e_1 e_2 e_3
X
+
c_1 c_2 c_3
NN Decoder
output
Minimize reconstruction error
$c_i = \exp(\sigma_i)e_i + m_i$
Minimize
$-\mathrm{KL}[\mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x) \| \mathcal{N}(0, I)]$
This constrains approcing 0 is good
Auto-Encoding Variational Bayes,
https://arxiv.org/abs/1312.6114

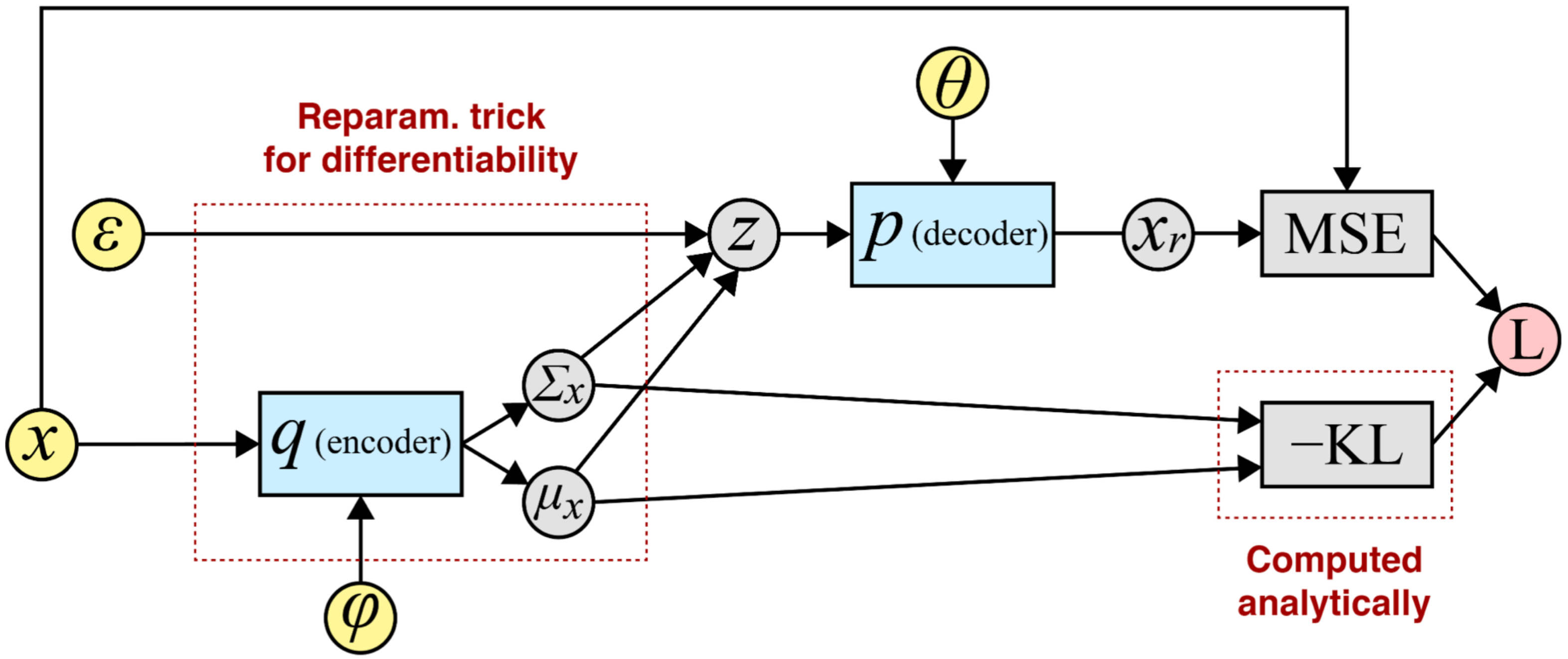
VAE

$$\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x = M(\mathbf{x}), \Sigma(\mathbf{x})$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$$

$$\mathbf{z} = \boldsymbol{\epsilon}\boldsymbol{\sigma}_x + \boldsymbol{\mu}_x$$

$$\mathbf{x}_r = p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$$



$\text{recon. loss} = \text{MSE}(\mathbf{x}, \mathbf{x}_r)$	Compute reconstruction loss
$\text{var. loss} = -\text{KL}[\mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x) \| \mathcal{N}(0, I)]$	Compute variational loss
$\text{L} = \text{recon. loss} + \text{var. loss}$	Combine losses

Why re-parameter?

Let's say we want to take the gradient w.r.t. θ of the following expectation,

$$\mathbb{E}_{p(z)}[f_\theta(z)]$$

where p is a density. Provided we can differentiate $f_\theta(z)$, we can easily compute the gradient:

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] &= \nabla_\theta \Big[\int_z p(z) f_\theta(z) dz \Big] \\ &= \int_z p(z) \Big[\nabla_\theta f_\theta(z) \Big] dz \\ &= \mathbb{E}_{p(z)} \Big[\nabla_\theta f_\theta(z) \Big]\end{aligned}$$

Why re-parameter?

what happens if our density *p* is also parameterized by θ?

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] &= \nabla_\theta \Big[\int_z p_\theta(z) f_\theta(z) dz \Big] \\
&= \int_z \nabla_\theta \Big[p_\theta(z) f_\theta(z) \Big] dz \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \int_z p_\theta(z) \nabla_\theta f_\theta(z) dz \\
&= \underbrace{\int_z f_\theta(z) \nabla_\theta p_\theta(z) dz}_{\text{What about this?}} + \mathbb{E}_{p_\theta(z)} \Big[\nabla_\theta f_\theta(z) \Big]
\end{aligned}
$$

Why re-parameter?

Re-parameter:

$$\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$$

$$\mathbf{z} = g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \mathbf{x})$$

$$\mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{z})}[f(\mathbf{z}^{(i)})] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(g_{\theta}(\boldsymbol{\epsilon}, \mathbf{x}^{(i)}))]$$

$$\begin{aligned}
\nabla_{\theta}\mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{z})}[f(\mathbf{z}^{(i)})] &= \nabla_{\theta}\mathbb{E}_{p(\boldsymbol{\epsilon})}[f(g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \mathbf{x}^{(i)}))] && (1) \\
&= \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\theta}} f(g_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}, \mathbf{x}^{(i)}))] && (2) \\
&\approx \frac{1}{L}\sum_{l=1}^{L} \nabla_{\boldsymbol{\theta}} f(g_{\boldsymbol{\theta}}(\epsilon^{(l)}, \mathbf{x}^{(i)})) && (3)
\end{aligned}$$

Entropy

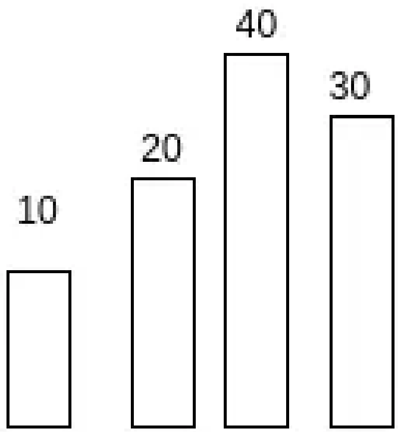
- $averageBit = p_1 * n_1 + p_2 * n_2 + \ldots + p_n * n_n$

n_i: #bits used to encode signal with probability of occurrence p_i

- With N types of signal, we need $n = log_2 N$ bits to encode
- assuming all signals have the same probability of occurrence: p = 1/N

$$n_p = log_2 N = log_2 \frac{1}{p} = -log_2 p$$

- Entropy = $\sum_i^n p_i * n_i$ = -$\sum_i^n p_i * log_2 p_i$



Weather distribution: sunny - rainy - cloudy -snowy

sunny - 0
rainy - 1
cloudy - 10
snow: 11

Encode Table 1

sunny - 11
rainy - 01
cloudy - 0
snow: 1

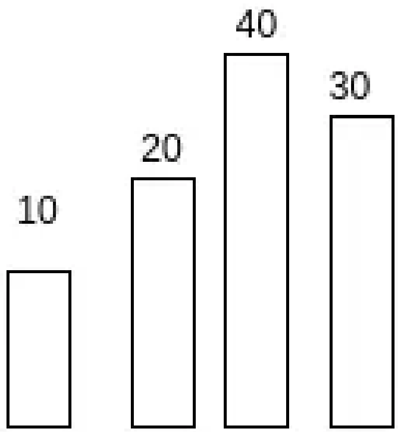
Encode Table 2

Cross entropy

- 2022: Q={0.1,0.3,0.4,0.2}
- In beginning of 2023: use Q to encode the signals
- In the end of 2023: re-statistics, we obtain: P = {0.11,0.29,0.41,0.19}
- The average bit used in 2023:

CrossEntropy(P,Q) = $-\sum_{i}^{n} p_i * log_2 q_i$

- Minimum if P==Q



Weather distribution: sunny - rainy - cloudy -snowy

sunny - 0
rainy - 1
cloudy - 10
snow: 11

Encode Table 1

sunny - 11
rainy - 01
cloudy - 0
snow: 1

Encode Table 2

KL (Kullback-Leibler) divergence

- Discrete:

$$D_{KL}(P||Q) = \Sigma_i P(i)\log[P(i)/Q(i)]$$

- Continuous:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x)\log\,[p(x)/q(x)]$$

- Explanations:

Entropy: $-\ \Sigma_i P(i)\log P(i)$ - expected code length (also optimal)

Cross Entropy: $-\ \Sigma_i P(i)\log Q(i)$ – expected coding length using optimal code for Q

$D_{KL} = \Sigma_i P(i)\log[P(i)/Q(i)] = \Sigma_i P(i)[\log P(i) - \log Q(i)]$, extra bits

$JSD(P||Q) = ½\ D_{KL}(P||M) + ½\ D_{KL}(Q||M)$, $M = ½\ (P+Q)$, symmetric KL

* JSD = Jensen-Shannon Divergency

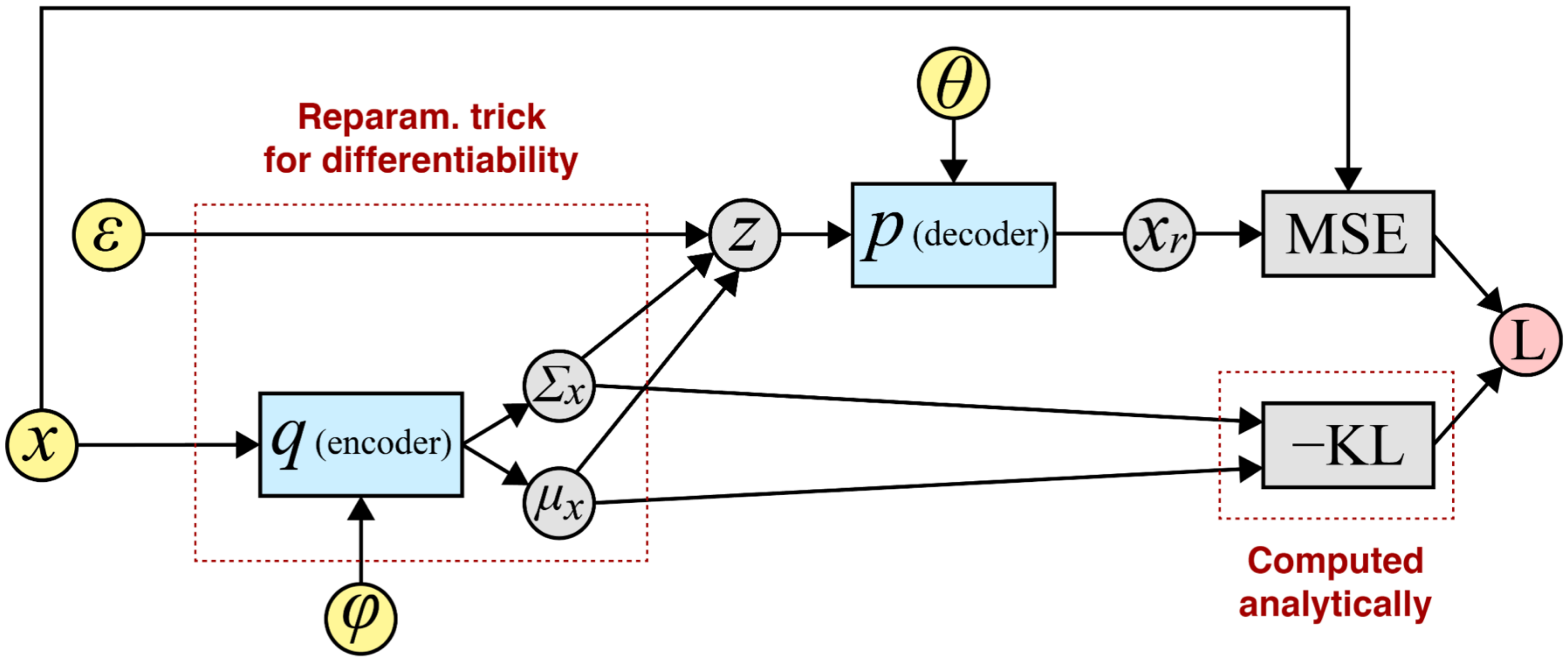
Why KL divergence?

- Bayes' rule:

P(x|z).P(z) = P(z|x).P(x)

- KL-divergence term encourages the approximate posterior P(z|x) to be close to the prior P(z)

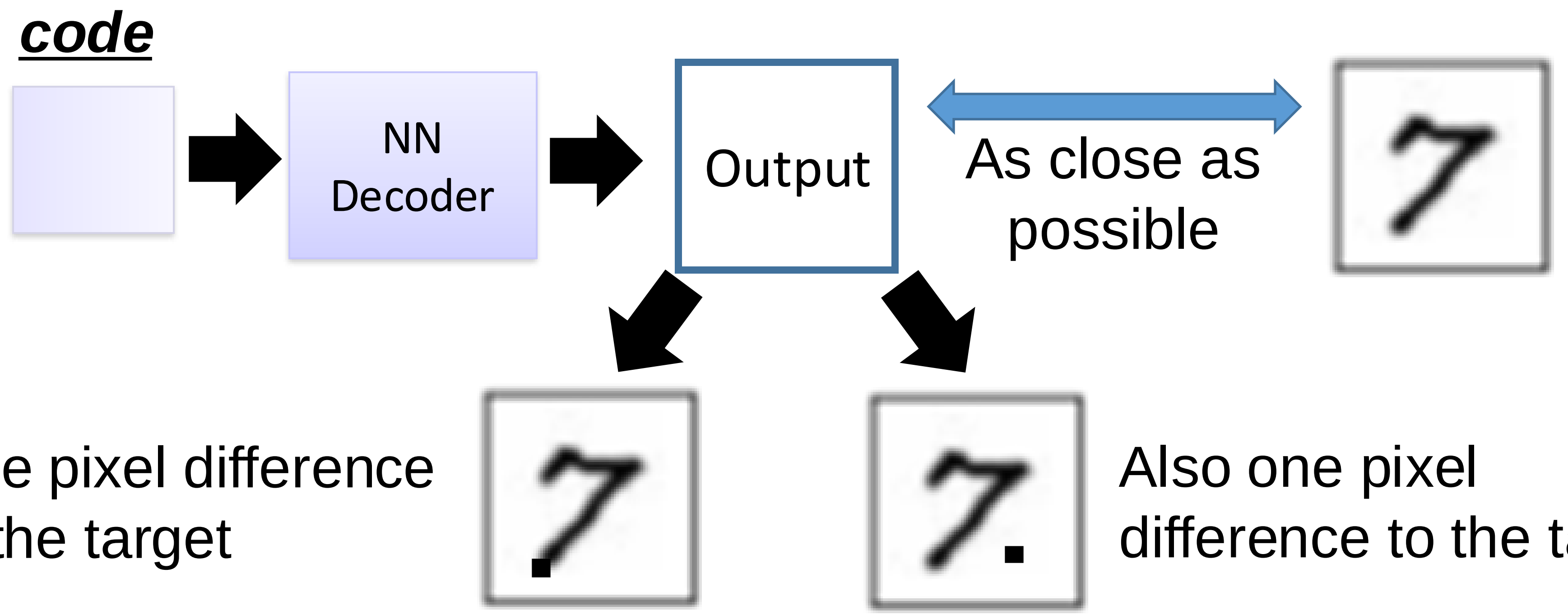
→P(x|z) = P(x)



$$\mathcal{L}^B = -\mathrm{KL}[\overbrace{q_\phi(\mathbf{z} \mid \mathbf{x}^{(i)})}^{\text{Encoder}} \| \overbrace{p_\theta(\mathbf{z})}^{\text{Fixed}}] + \frac{1}{L}\sum_{l=1}^{L} \log \overbrace{p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} \mid \mathbf{z}^{(l)})}^{\text{Decoder}}$$

Problems of VAE

- It does not really try to simulate real images



One pixel difference
to the target

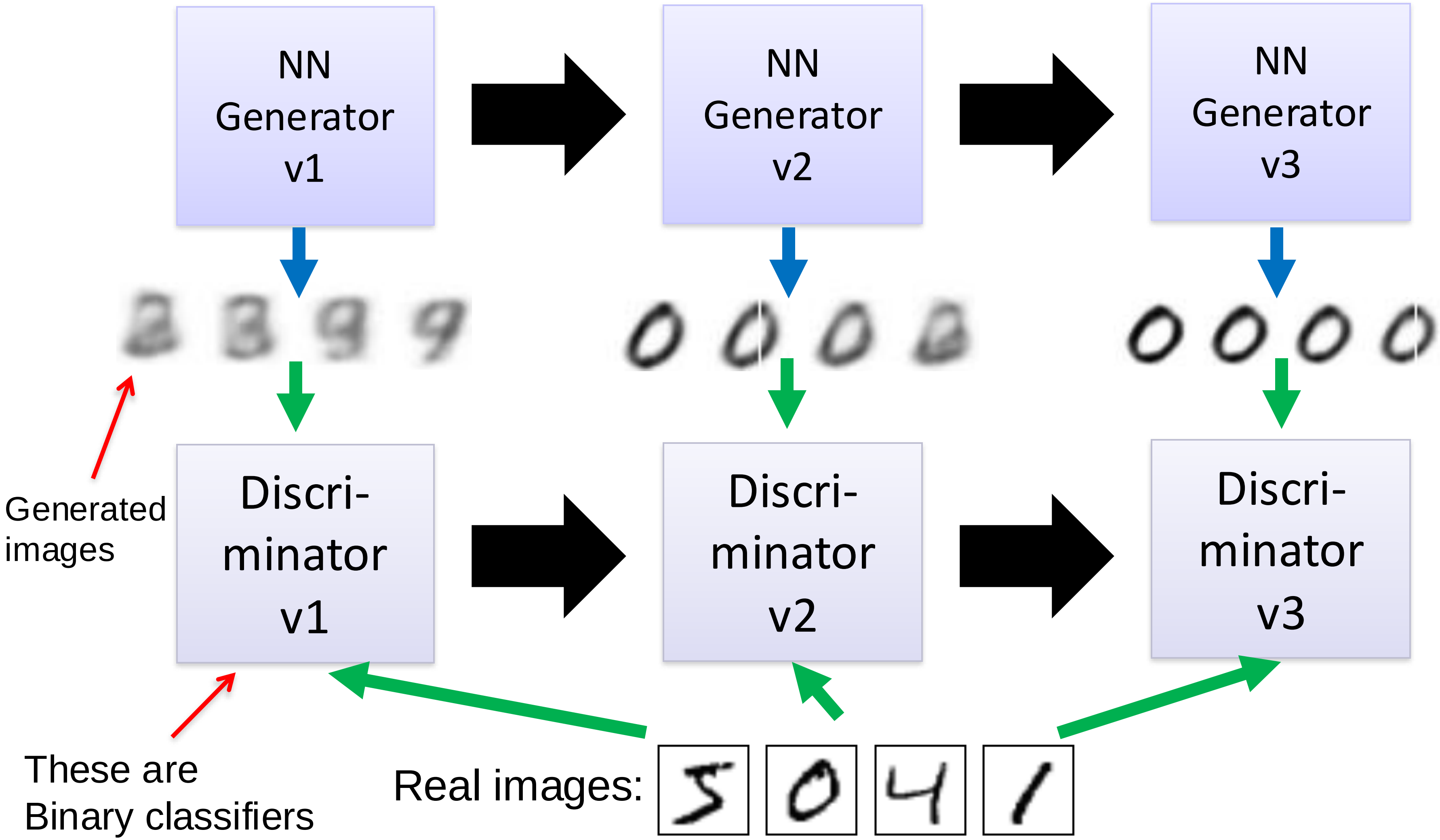
Also one pixel
difference to the target

Realistic

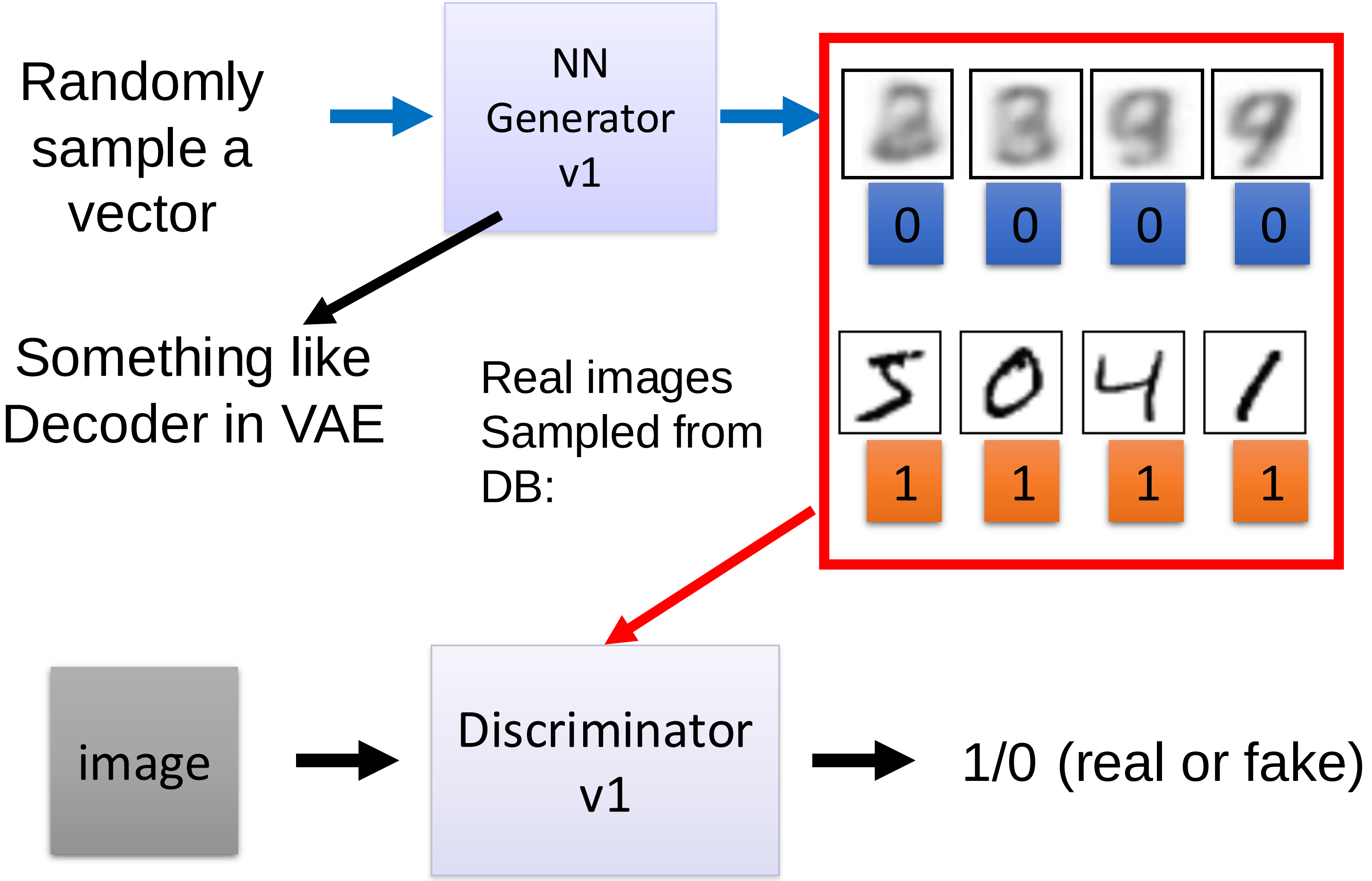
Fake

VAE treats these the same

Gradual and step-wise generation



GAN – Learn a discriminator



GAN – Learn a generator

Updating the parameters of generator

➡ The output be classified as “real” (as close to 1 as possible)

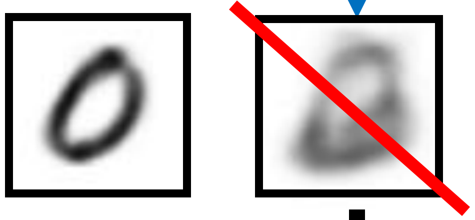
Generator + Discriminator = a network

Using gradient descent to update the parameters in the generator, but fix the discriminator

Randomly sample a vector

Train this

NN Generator v2 ~~v1~~



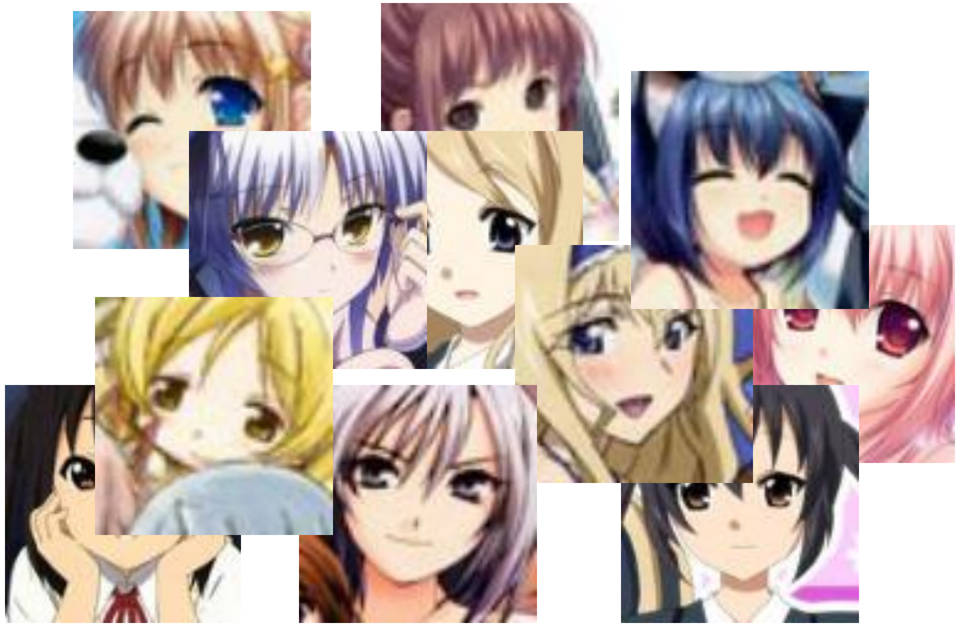
They have Opposite objectives

Do not Train This

Discriminator v1

1.0 ~~0.13~~

Generating 2^{nd} element figures



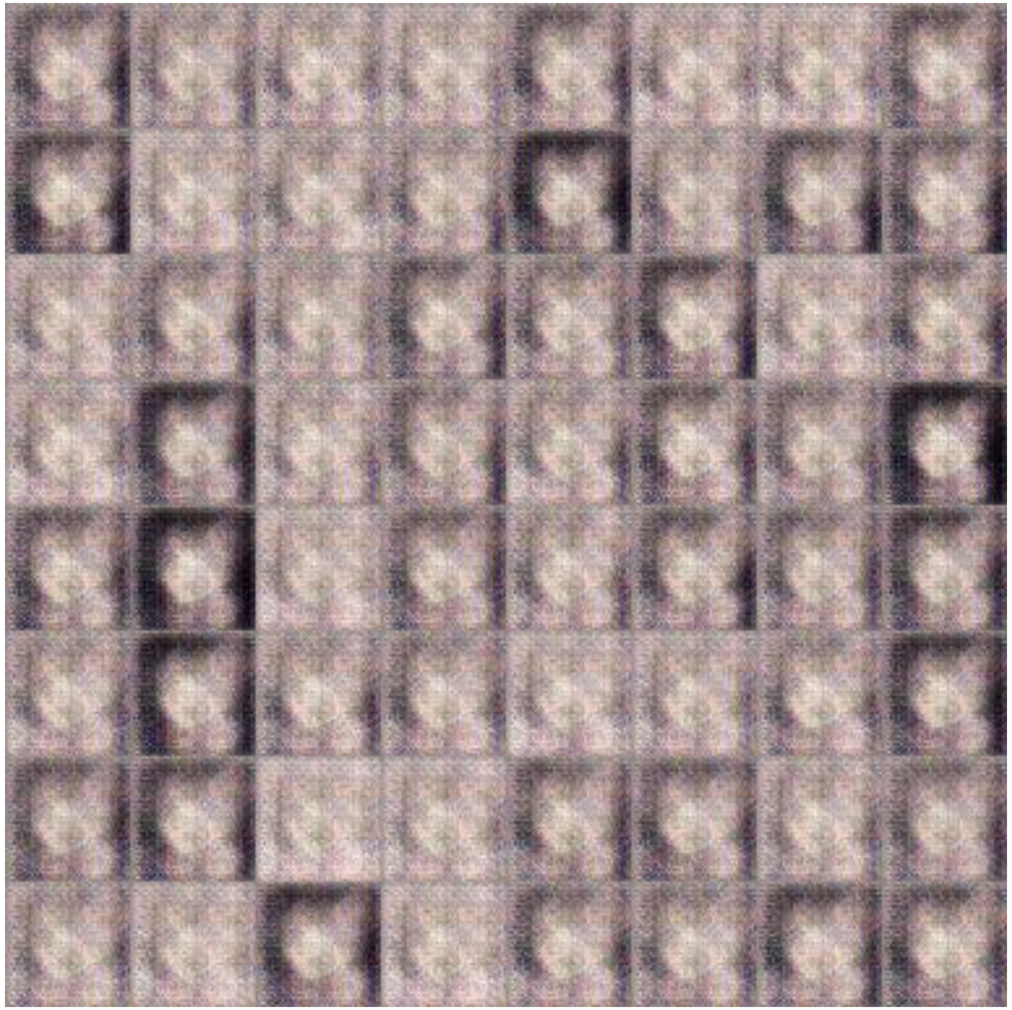
You can use the following to start a project (but this is in Chinese):

Source of images: https://zhuanlan.zhihu.com/p/24767059
From Dr. HY Lee's notes.

DCGAN: https://github.com/carpedm20/DCGAN-tensorflow

GAN – generating 2nd element figures

100 rounds



This is fast, I think you can use your CPU

GAN – generating 2nd element figures

1000 rounds

GAN – generating 2nd element figures


2000 rounds

GAN – generating 2nd element figures

5000 rounds

GAN – generating 2nd element figures

10,000 rounds

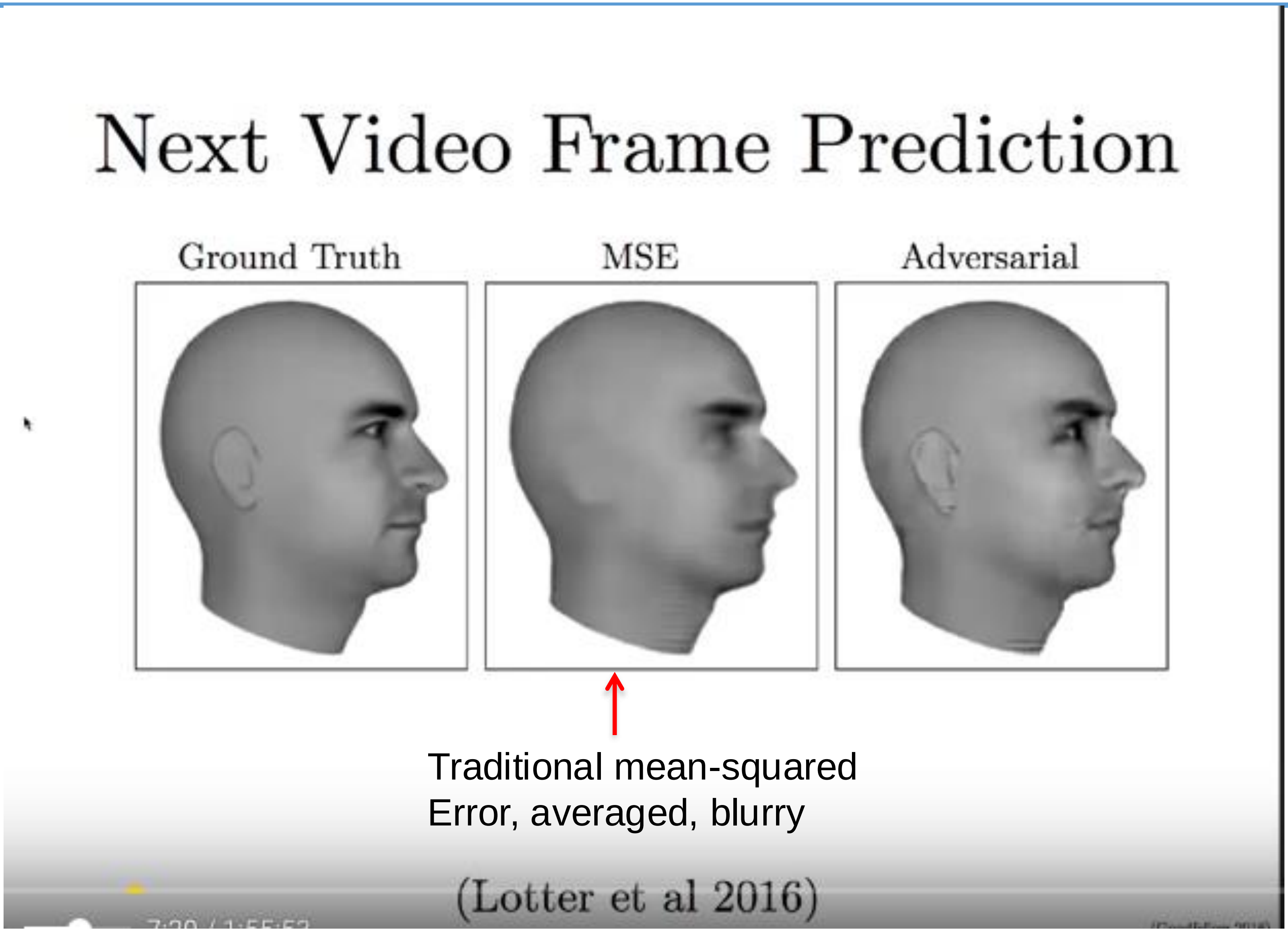
GAN – generating 2nd element figures

20,000 rounds

GAN – generating 2^{nd} element figures

50,000 rounds

Next few images from Goodfellow lecture



Traditional mean-squared
Error, averaged, blurry

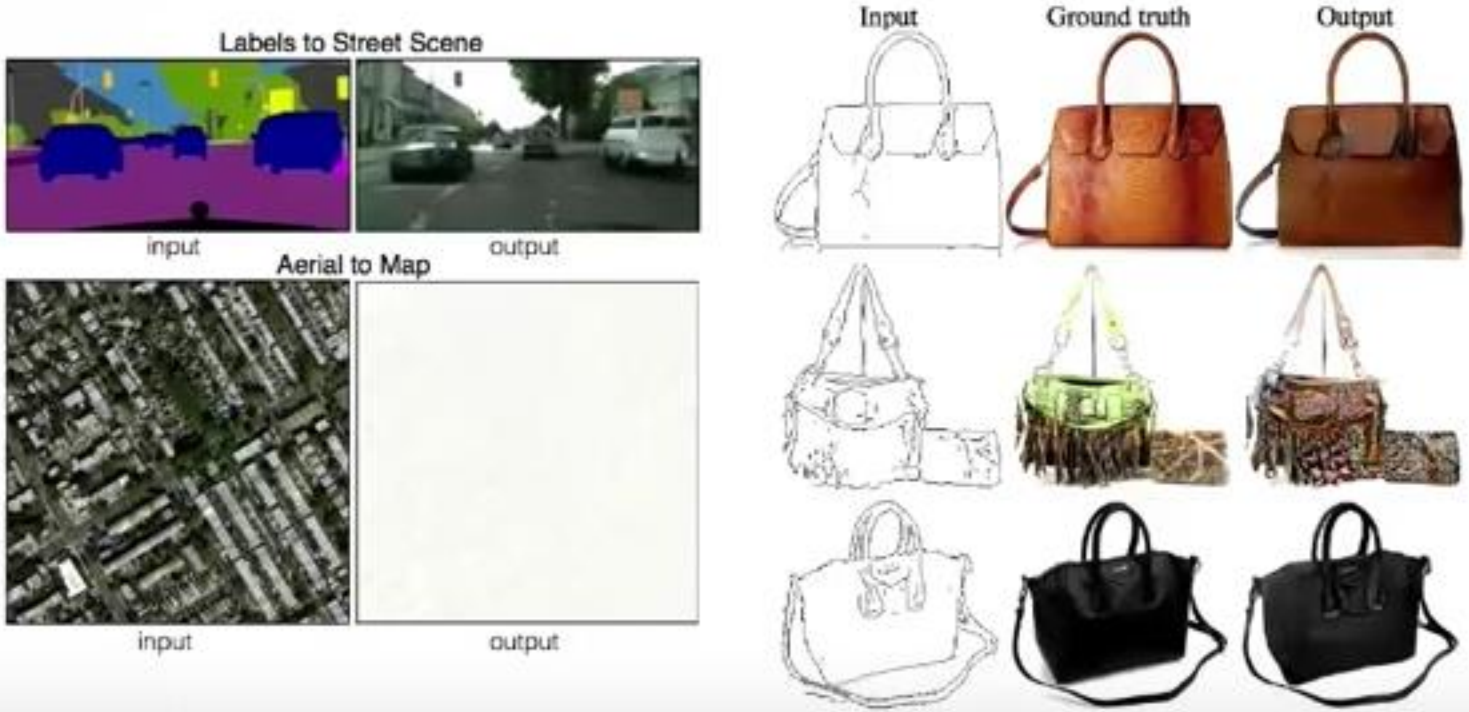
HCMUTE

Single Image Super-Resolution

(Ledig et al 2016)

Last 2 are by deep learning approaches.

Image to Image Translation



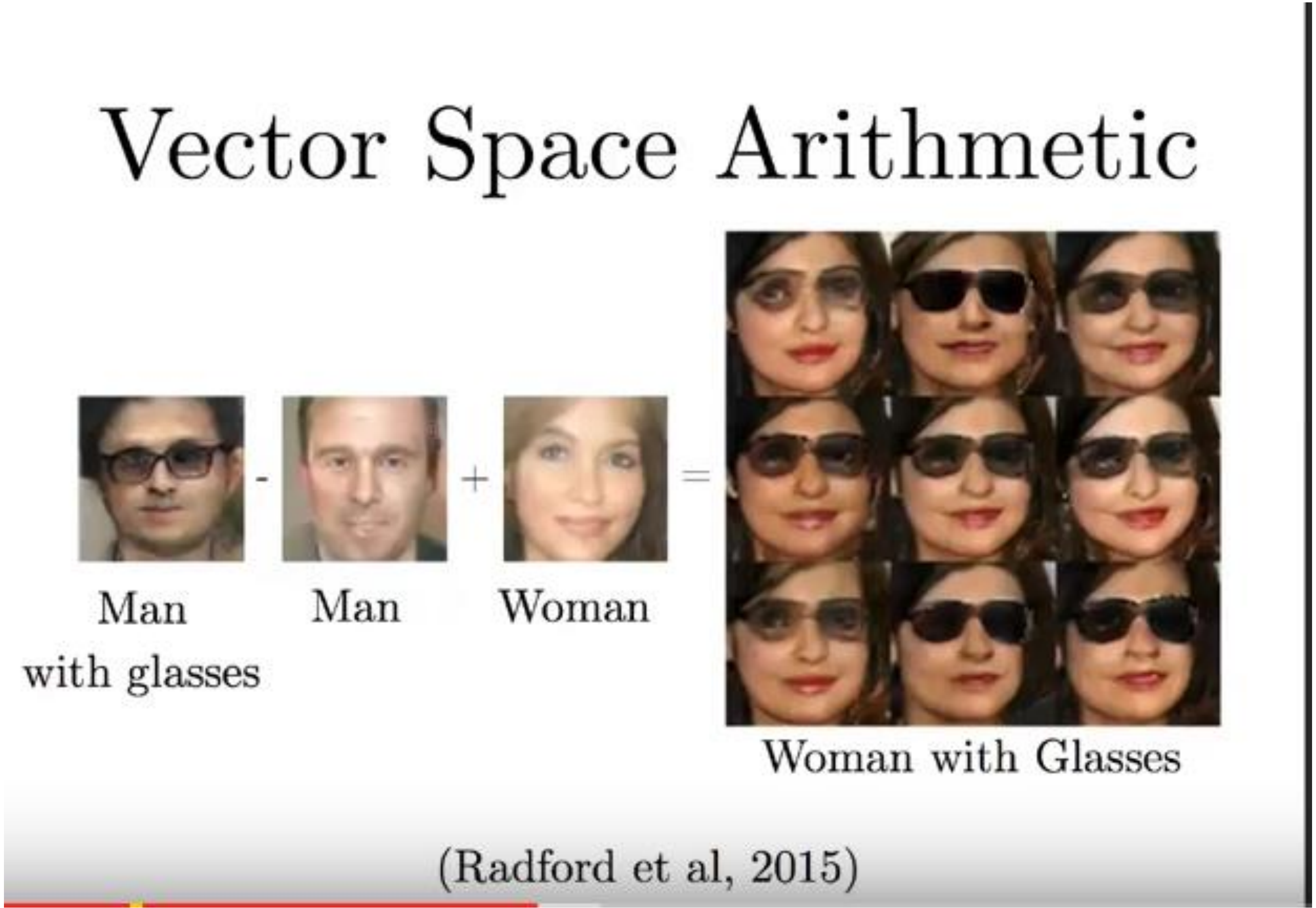
(Isola et al 2016)

(Goodfellow 2016)

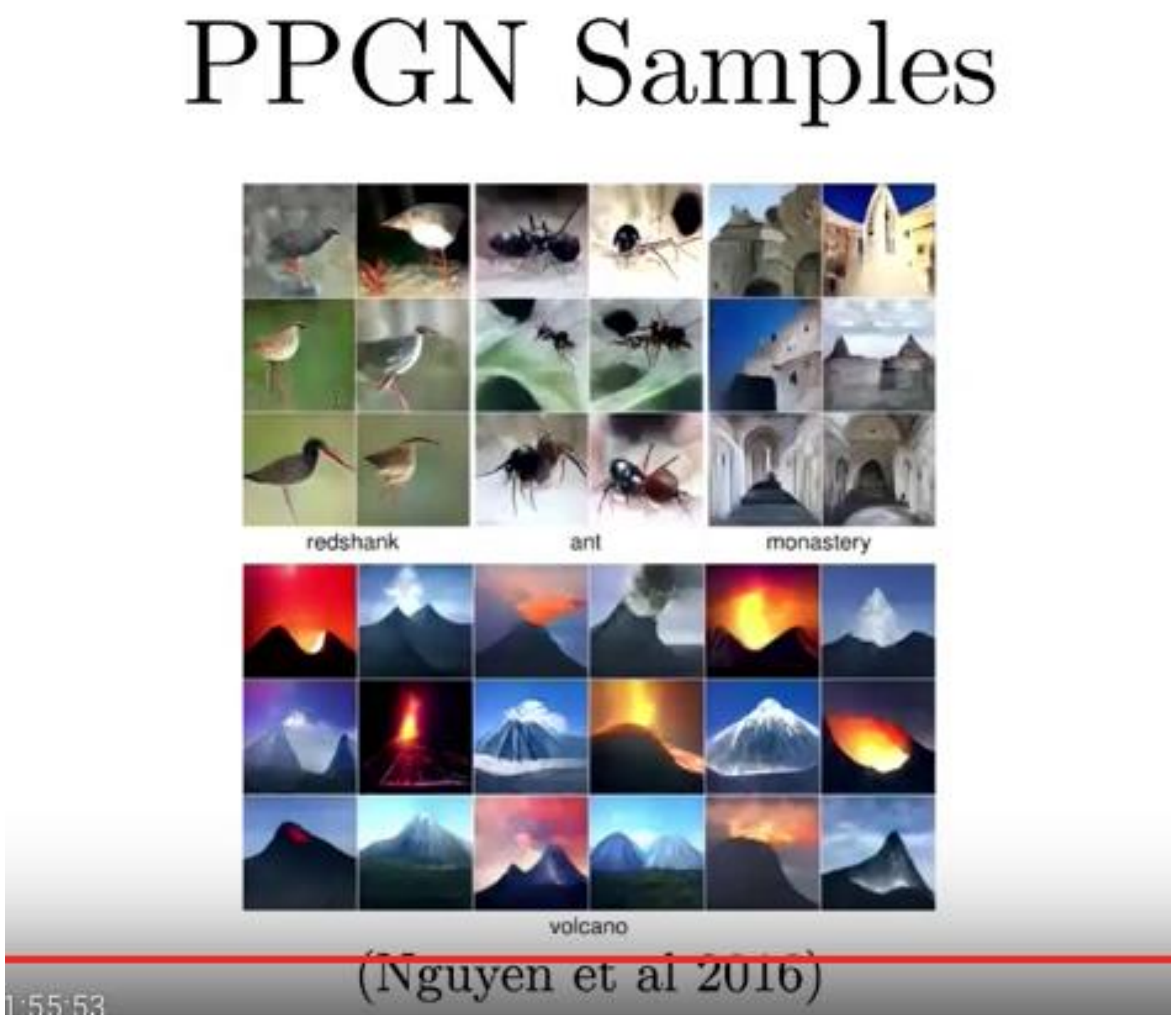
DCGANs for LSUN Bedrooms

(Radford et al 2015)

Similar to word embedding (DCGAN paper)



256x256 high resolution pictures
by Plug and Play generative network



From natural language to pictures

PPGN for caption to image

oranges on a table next to a liquor bottle

(Nguyen et al 2016)

Basic Idea of GAN

- Generator G
  - G is a function, input z, output x
  - Given a prior distribution $P_{prior}(z)$, a probability distribution $P_G(x)$ is defined by function G
- Discriminator D
  - D is a function, input x, output scalar
  - Evaluate the "difference" between $P_G(x)$ and $P_{data}(x)$
- In order for D to find difference between P_{data} from P_G, we need a cost function V(G,D):

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

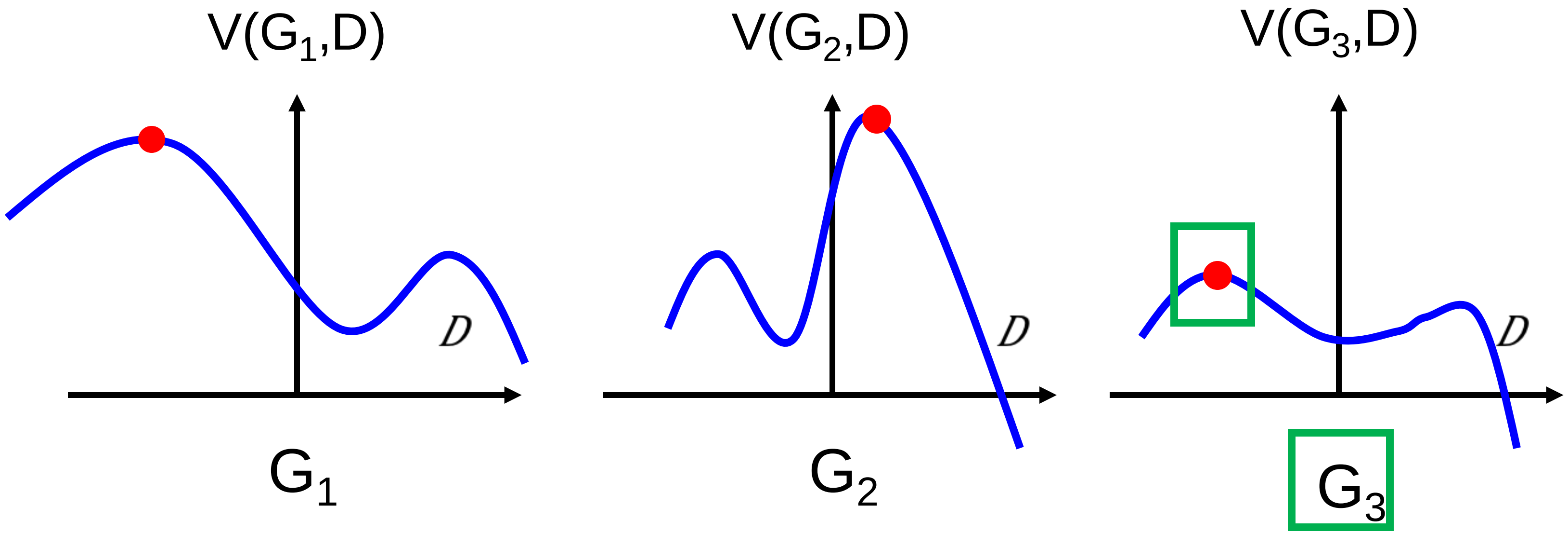
Basic Idea

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

Pick JSD function: $V = E_{x\sim P\_data}[\log D(x)] + E_{x\sim P\_G}[\log(1-D(x))]$

Given a generator G, $\max_D V(G,D)$ evaluates the "difference" between P_G and P_{data}

Pick the G s.t. P_G is most similar to P_{data}



$\text{Max}_D V(G,D)$, $G^* = \arg\min_G \max_D V(G,D)$

- Given G, what is the optimal D* maximizing

$$V = E_{x\sim P\_data}[\log D(x)] + E_{x\sim P\_G}[\log(1-D(x))]$$
$$= \Sigma\,[\,P_{data}(x)\log D(x) + P_G(x)\log(1-D(x)\,]$$

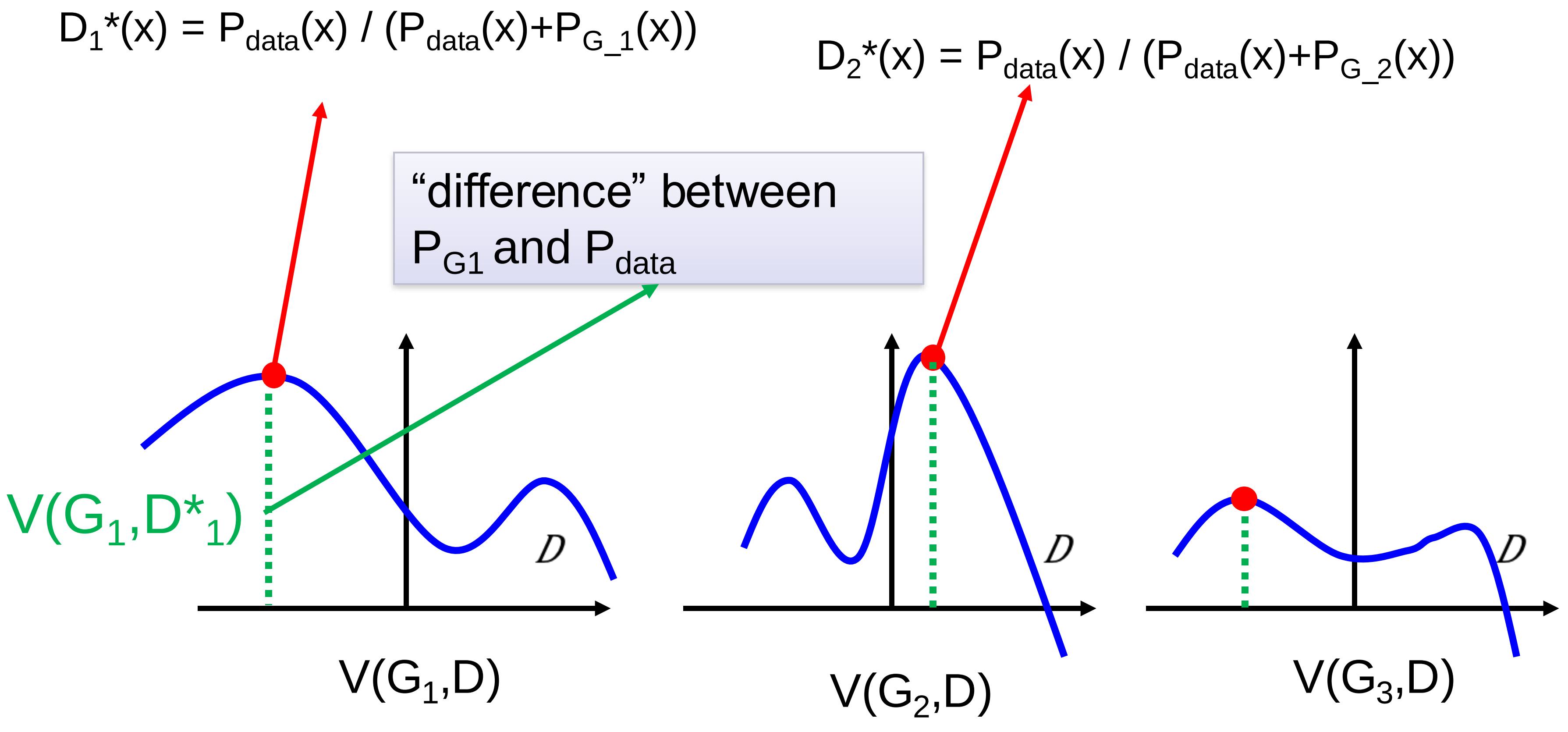
$$\text{Thus: } D^*(x) = P_{data}(x) / (P_{data}(x)+P_G(x))$$

Assuming D(x) can have any value here

- Given x, the optimal D* maximizing is:

$$f(D) = a\log D + b\log(1-D) \Rightarrow D^* = a/(a+b)$$

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



$\max_D V(G,D)$

$V = E_{x\sim P\_data}[\log D(x)] + E_{x\sim P\_G}[\log(1-D(x))]$

$\max_D V(G,D)$

$= V(G,D^*)$, where $D^*(x) = P_{data} / (P_{data} + P_G)$, and

$1-D^*(x) = P_G / (P_{data} + P_G)$

$= E_{x\sim P\_data} \log D^*(x) + E_{x\sim P\_G} \log (1-D^*(x))$

$\approx \Sigma [P_{data} (x) \log D^*(x) + P_G(x) \log (1-D^*(x))]$

$= -2\log 2 + 2\, JSD(P_{data} || P_G)$,

$JSD(P||Q)$ = Jensen-Shannon divergence

$= ½\, D_{KL}(P||M)+ ½\, D_{KL}(Q||M)$

where $M= ½ (P+Q)$.

$D_{KL}(P||Q) = \Sigma P(x) \log P(x) /Q(x)$

Summary:

$$V = E_{x\sim P\_data}[\log D(x)] + E_{x\sim P\_G}[\log(1-D(x))]$$

- Generator G, Discriminator D
- Looking for G* such that

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

- Given G, $\max_D V(G,D)$

$$= -2\log 2 + 2JSD(P_{data}(x) \,||\, P_G(x))$$

- What is the optimal G? It is G that makes JSD smallest = 0:

$$P_G(x) = P_{data}(x)$$

Algorithm

Initialize θ_d for D and θ_g for G

HCMUTE

- In each training iteration

Ian Goodfellow comment: this is also done once

Learning D

Repeat k times

- Sample m examples $\{x^1, x^2, \ldots x^m\}$ from data distribution $P_{data}(x)$
- Sample m noise samples $\{z^1, \ldots, z^m\}$ from a simple prior $P_{prior}(z)$
- Obtain generated data $\{x^{*1}, \ldots, x^{*m}\}$, $x^{*i} = G(z^i)$
- Update discriminator parameters θ_d to maximize
  - $V' \approx 1/m \, \Sigma_{i=1..m} \log D(x^i) + 1/m \, \Sigma_{i=1..m} \log(1 - D(x^{*i}))$
  - $\theta_d \leftarrow \theta_d + \eta \Delta V'(\theta_d)$ (gradient ascent)

Learning G

Only Once

- Sample another m noise samples $\{z^1, z^2, \ldots z^m\}$ from the prior $P_{prior}(z)$, $G(z^i) = x^{*i}$
- Update generator parameters θ_g to minimize

$$V' = \cancel{1/m \, \Sigma_{i=1..m} \log D(x^i)} + 1/m \, \Sigma_{i=1..m} \log(1 - D(x^{*i}))$$

$$\theta_g \leftarrow \theta_g - \eta \Delta V'(\theta_g) \quad \text{(gradient descent)}$$

Objective Function for Generator in Real Implementation

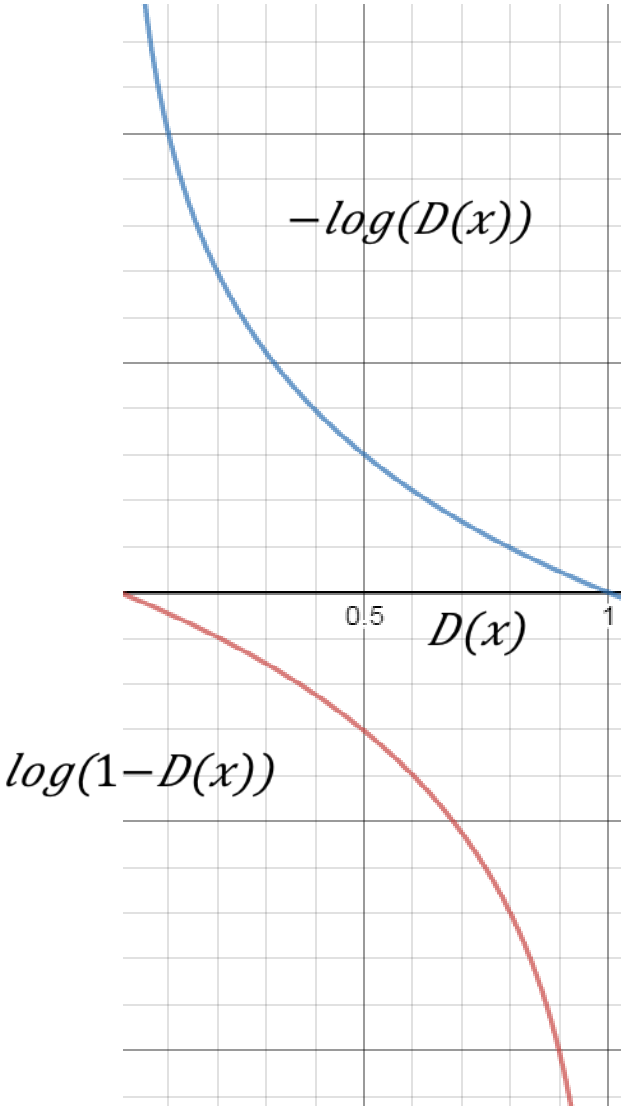
$V = \cancel{E_{x\sim P\_data}[\log D(x)]}$

$+ E_{x\sim P\_G}[\log(1-D(x))]$

Training slow at the beginning

$V = E_{x\sim P\_G}[-\log(D(x))]$

Real implementation:
label x from P_G as positive



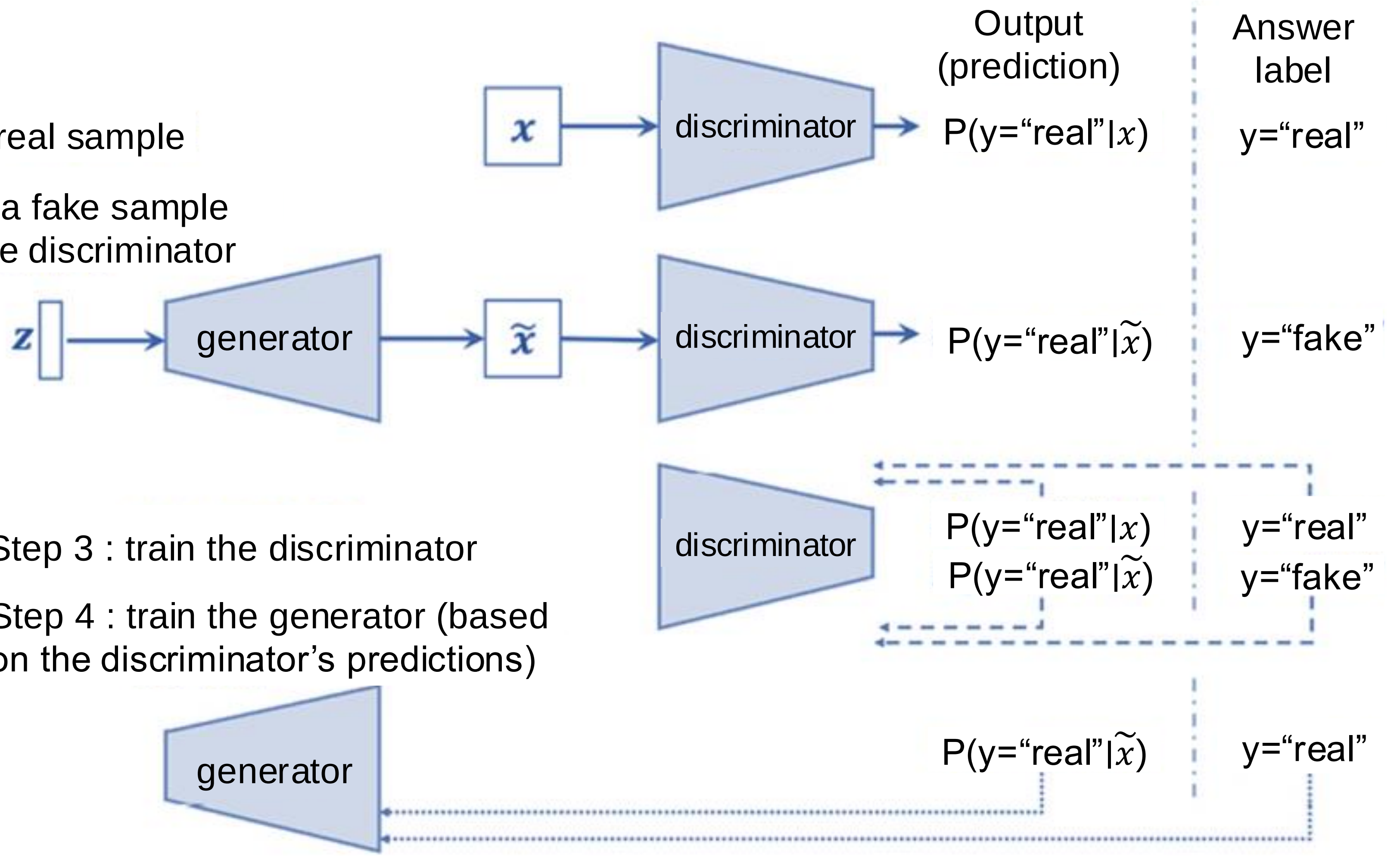
GAN training process

Step 1 : deliver a real sample

Step 2 : generate a fake sample and deliver it to the discriminator

Step 3 : train the discriminator

Step 4 : train the generator (based on the discriminator’s predictions)

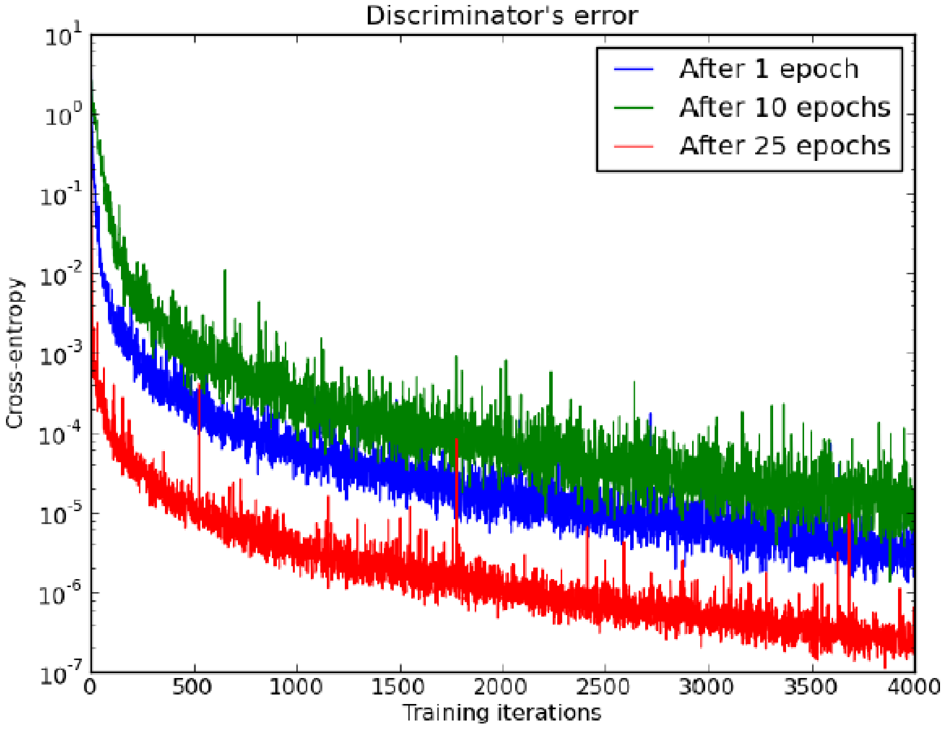


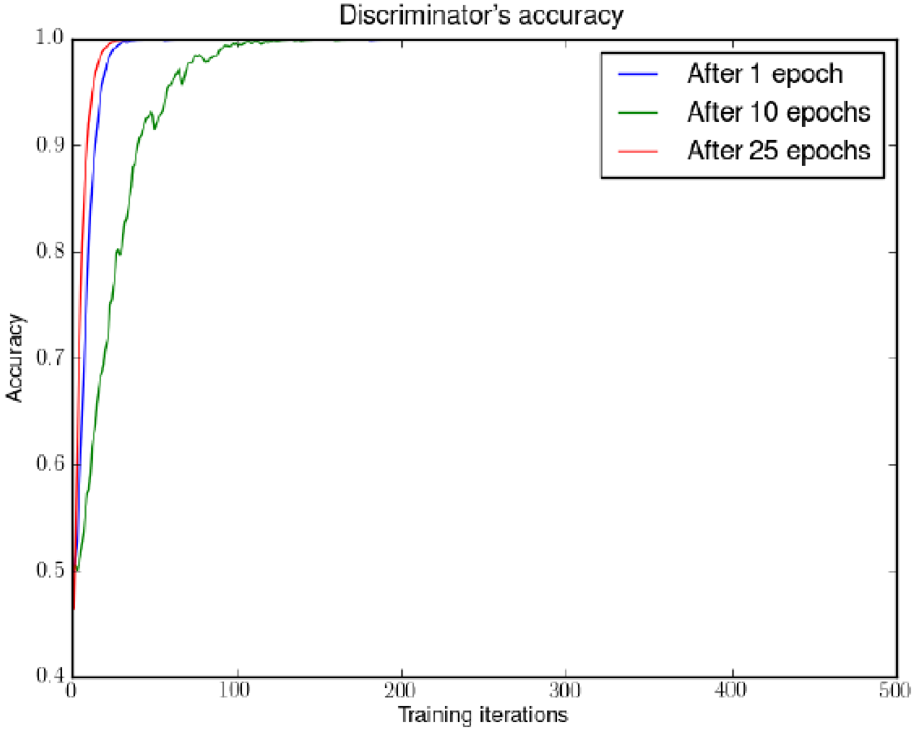
Some issues in training GAN

M. Arjovsky, L. Bottou, Towards principled methods for training generative adversarial networks, 2017.

Evaluating JS divergence

Discriminator is too strong: for all three Generators, JSD = 0



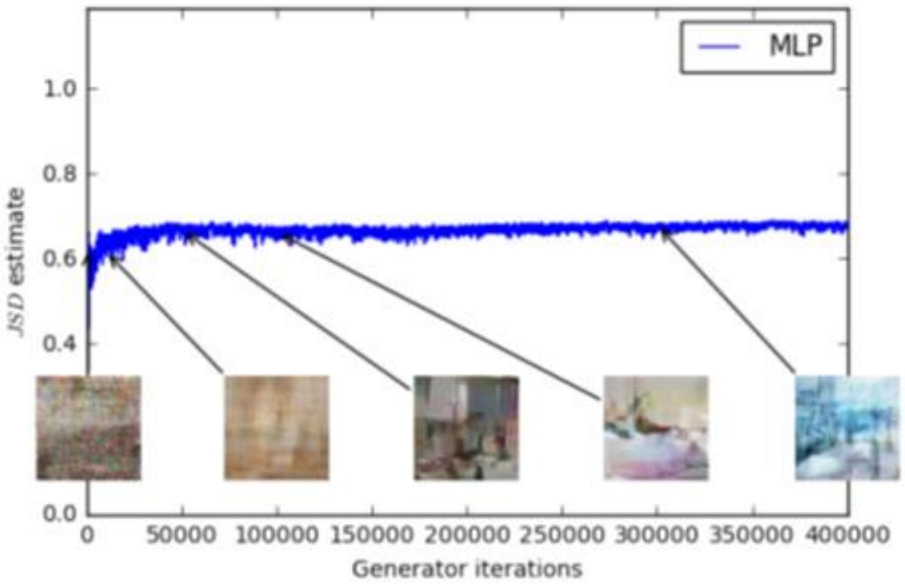


Martin Arjovsky, Léon Bottou, Towards Principled Methods for Training Generative Adversarial Networks, 2017, arXiv preprint

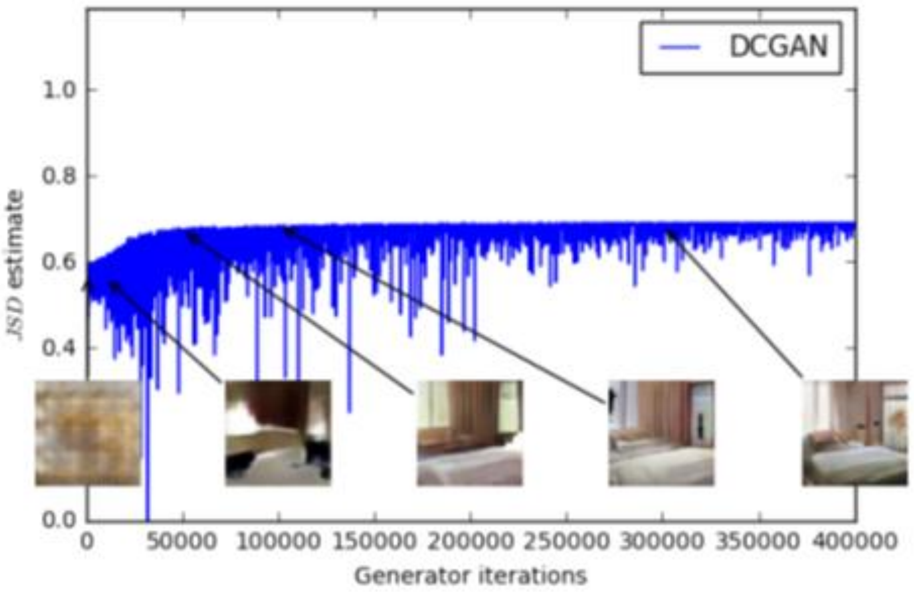
Evaluating JS divergence

https://arxiv.org/abs/1701.07875

- JS divergence estimated by discriminator telling little information



Weak Generator



Strong Generator

Discriminator

1 for all positive examples

0 for all negative examples

$$V = E_{x\sim P\_data}[\log D(x)] + E_{x\sim P\_G}[\log(1-D(x))]$$
$$= 1/m\ \Sigma_{i=1..m} \log D(x^i) + 1/m\ \Sigma_{i=1..m} \log(1-D(x^{*i}))$$

$$\max_D V(G,D) = -2\log 2 + 2\ JSD(P_{data} \,||\, P_G)$$

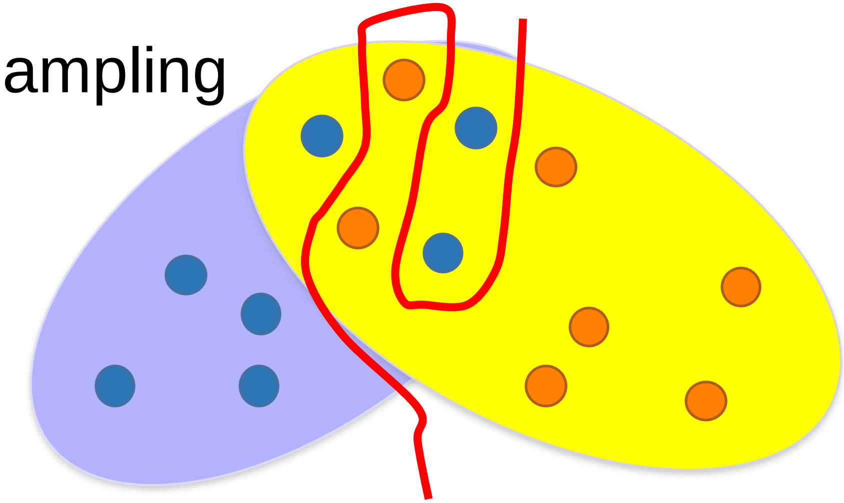
= 0

log 2 when P_{data} and P_G differ completely

Reason 1. Approximate by sampling

Weaken your discriminator?

Can weak discriminator compute JS divergence?



Discriminator

GAN implementation estimation

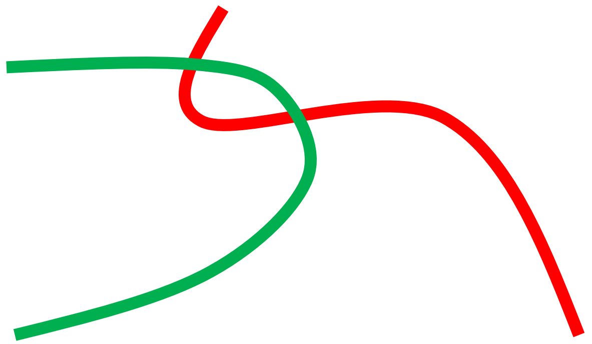
$$V = E_{x\sim P\_data}\,[\overbrace{\log D(x)}^{1}] + E_{x\sim P\_G}[\overbrace{\log(1-D(x))}^{0}]$$
$$= 1/m\ \Sigma_{i=1..m} \log D(x^i) + 1/m\ \Sigma_{i=1..m} \log(1-D(x^{*i}))\ \approx 0$$

$$\max_D V(G,D) = -2\log 2 + \underbrace{2\ JSD(P_{data}\ ||\ P_G\ )}_{\log 2}\ \boxed{= 0}$$

Theoretical estimation

Reason 2. the nature of data

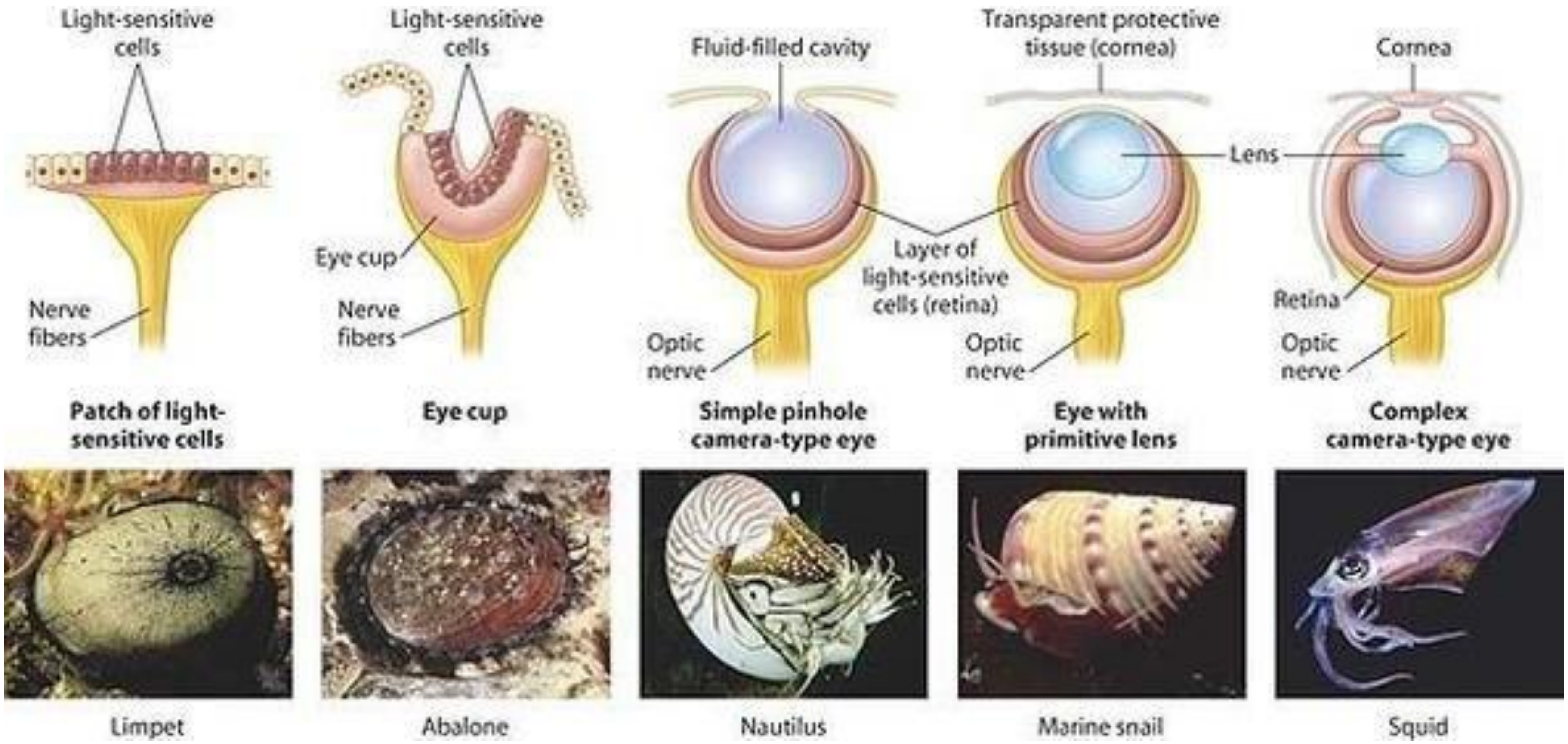
$P_{data}(x)$ and $P_G(x)$ have very little overlap in high dimensional space



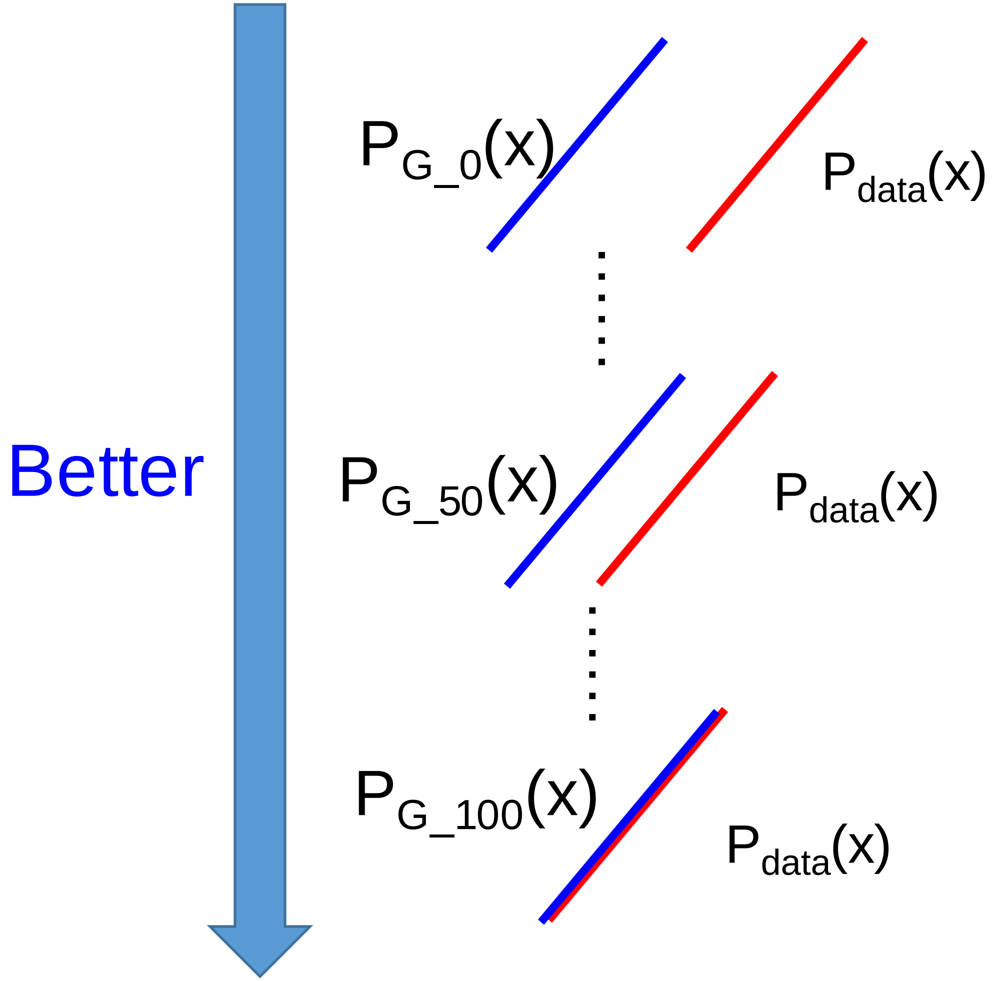
Evolution

http://www.guokr.com/post/773890/

Better



Evolution needs to be smooth:



$JSD(P_{G\_0} \| P_{data}) = \log 2$

$JSD(P_{G\_50} \| P_{data}) = \log 2$

Not really better ……

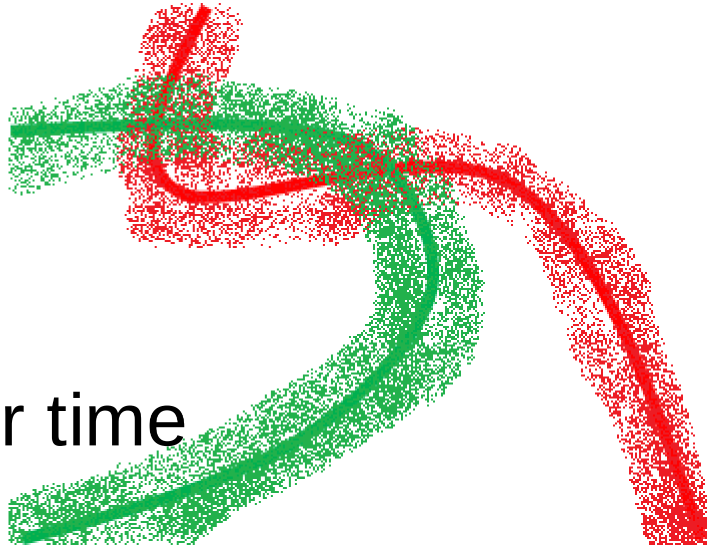
$JSD(P_{G\_100} \| P_{data}) = 0$

One simple solution: add noise

- Add some artificial noise to the inputs of discriminator
- Make the labels noisy for the discriminator

Discriminator cannot perfectly separate real and generated data

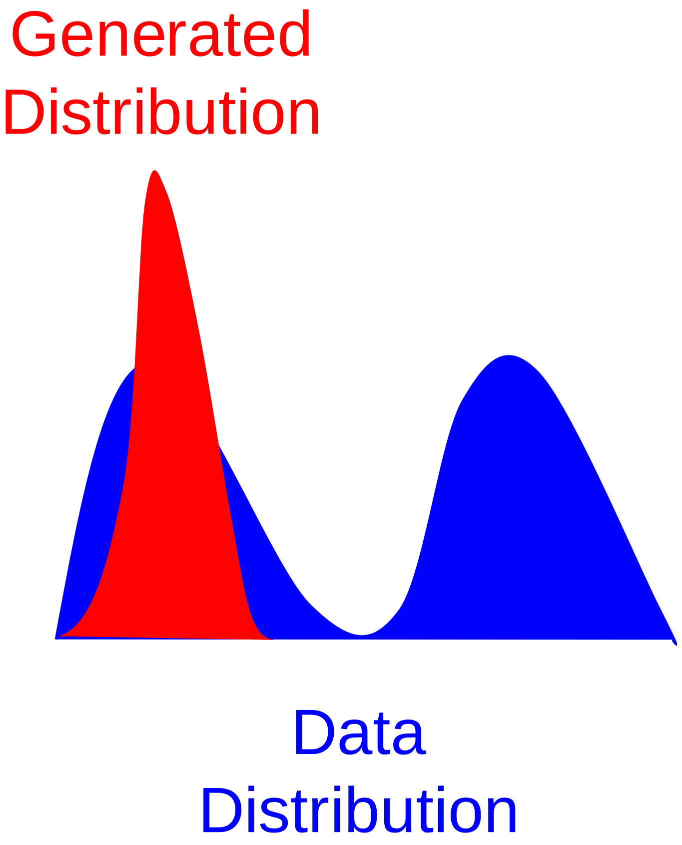
$P_{data}(x)$ and $P_G(x)$ have some overlap



Noises need to decay over time

Mode Collapse

Converge to same faces

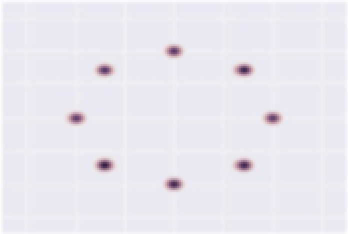


Sometimes, this is hard to tell since
one sees only what’s generated, but not what’s missed.

Mode Collapse Example

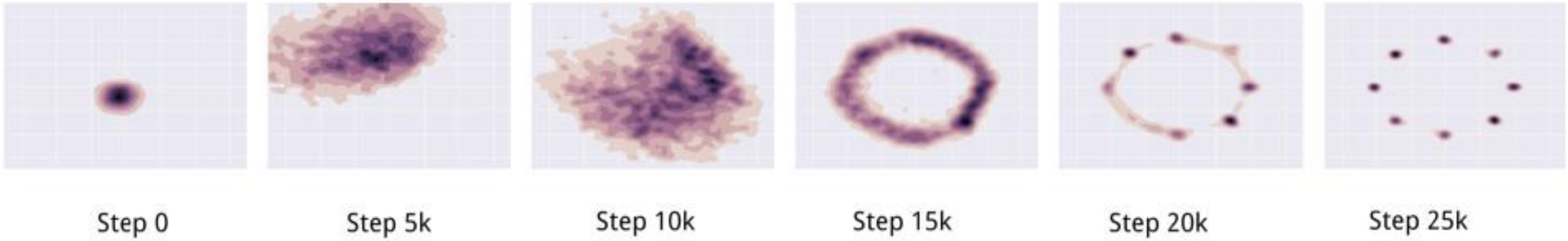
8 Gaussian distributions:

P_{data}

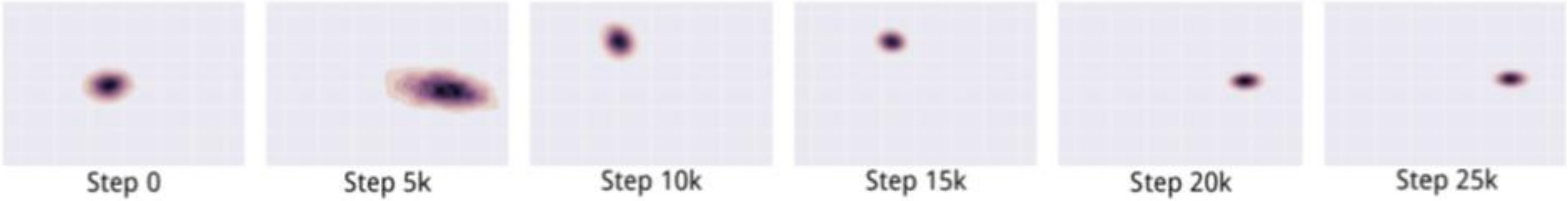


Target

What we want …



In reality …



HCMUTE

Algorithm WGAN

- In each training iteration

Ian Goodfellow comment: this is also done once

Learning D

Repeat k times

- Sample m examples $\{x^1, x^2, \ldots x^m\}$ from data distribution $P_{data}(x)$
- Sample m noise samples $\{z^1, \ldots, z^m\}$ from a simple prior $P_{prior}(z)$
- Obtain generated data $\{x^{*1}, \ldots, x^{*m}\}$, $x^{*i}=G(z^i)$
- Update discriminator parameters θ_d to maximize
  - $V' \approx \Sigma_{i=1..m} \cancel{\log} D(x^i) + 1/m\, \Sigma_{i=1..m} \cancel{\log}(1-D(x^{*i}))$
  - $\theta_d \leftarrow \theta_d + \eta \Delta V'(\theta_d)$ (gradient ascent plus weight clipping)

Learning G

Only Once

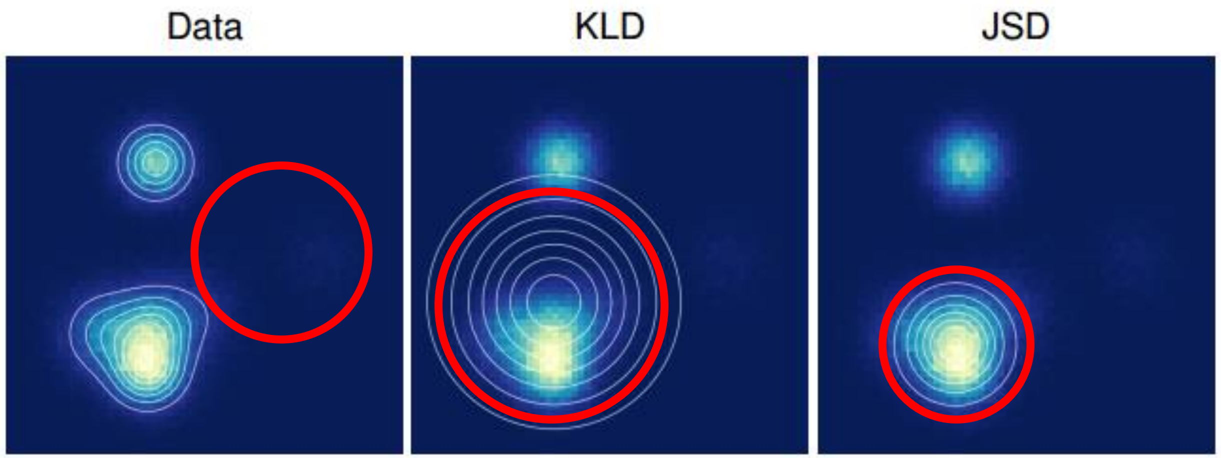
- Sample another m noise samples $\{z^1, z^2, \ldots z^m\}$ from the prior $P_{prior}(z)$, $G(z^i)=x^{*i}$
- Update generator parameters θ_g to minimize

$$V' = \cancel{1/m\, \Sigma_{i=1..m} \log D(x^i)} + 1/m\, \Sigma_{i=1..m} \cancel{\log}(1-D(x^{*i}))$$

$$\theta_g \leftarrow \theta_g - \eta \Delta V'(\theta_g) \quad \text{(gradient descent)}$$

Experimental Results

- Approximate a mixture of Gaussians by single mixture



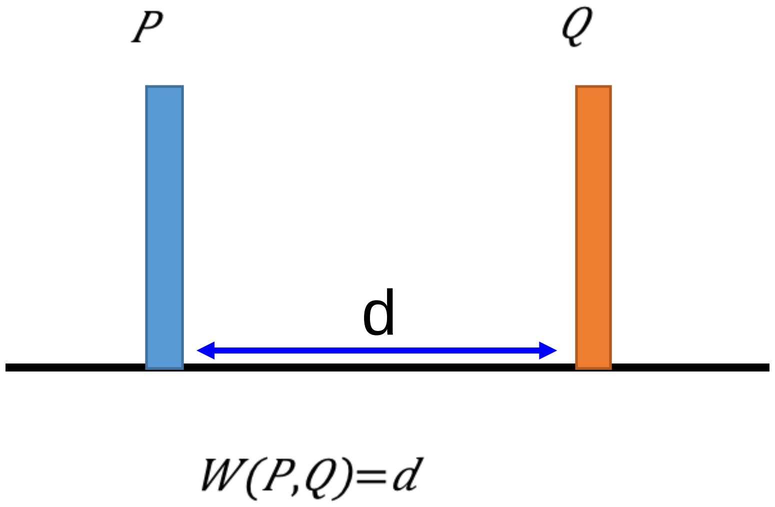
train \ test	KL	KL-rev	JS	Jeffrey	Pearson
KL	**0.2808**	0.3423	0.1314	0.5447	0.7345
KL-rev	0.3518	**0.2414**	0.1228	0.5794	1.3974
JS	0.2871	0.2760	**0.1210**	0.5260	0.92160
Jeffrey	0.2869	0.2975	0.1247	**0.5236**	0.8849
Pearson	0.2970	0.5466	0.1665	0.7085	**0.648**

WGAN Background

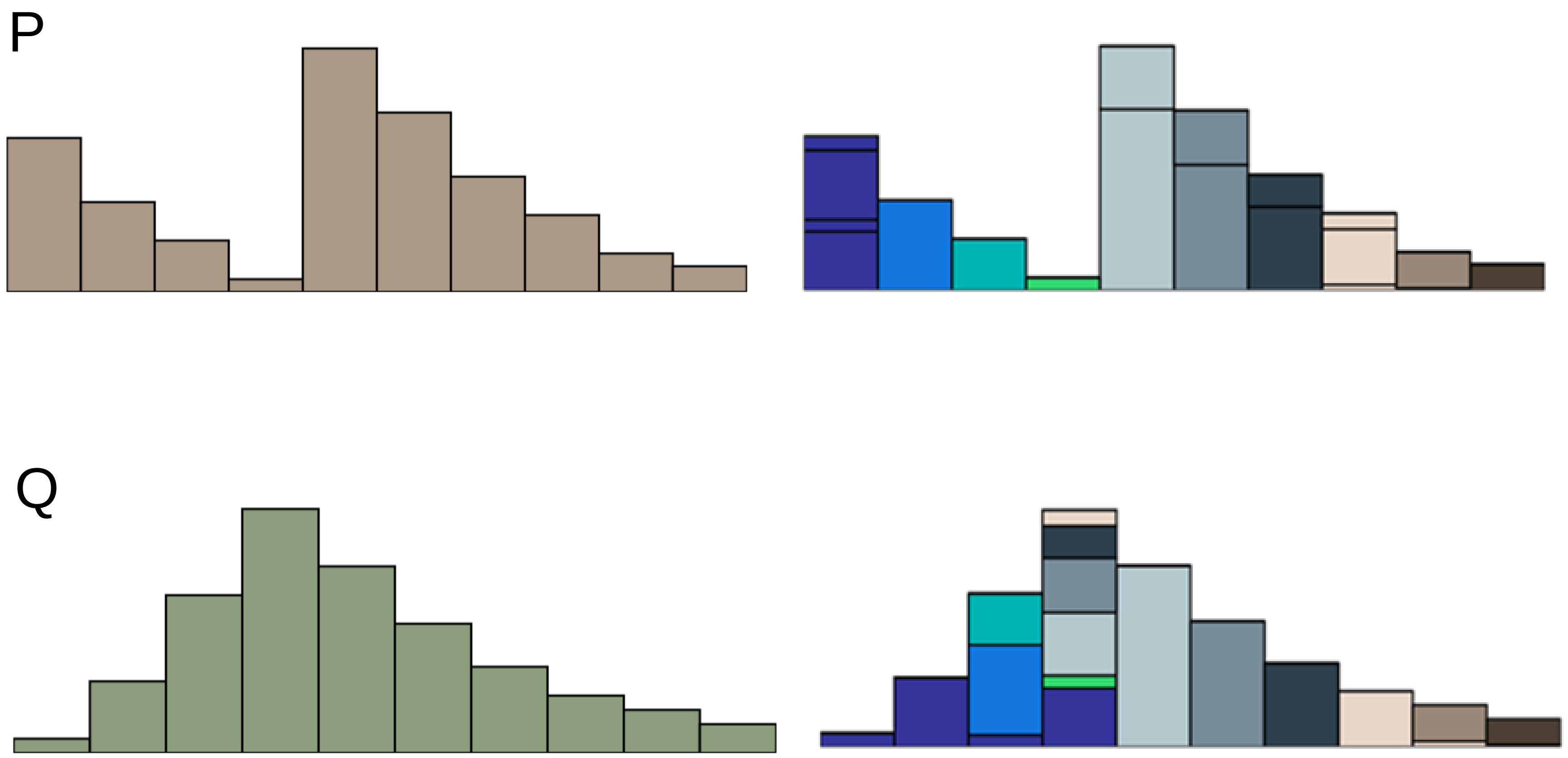
- We have seen that JSD does not give GAN a smooth and continuous improvement curve.
- We would like to find another distance which gives that.
- This is the Wasserstein Distance or earth mover’s distance.

Earth Mover's Distance

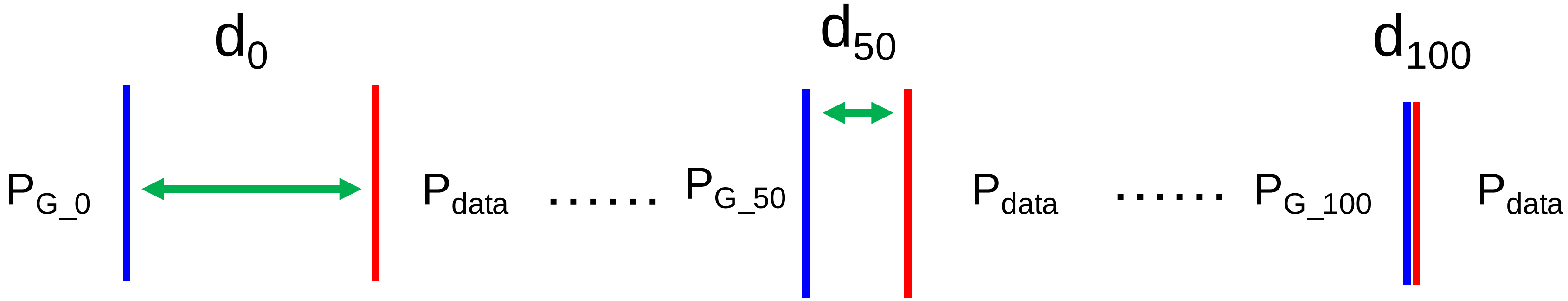
- Considering one distribution P as a pile of earth (total amount of earth is 1), and another distribution Q (another pile of earth) as the target
- The "earth mover's distance" or "Wasserstein Distance" is the average distance the earth mover has to move the earth in an optimal plan.



Earth Mover’s Distance: best plan to move



JS vs Earth Mover's Distance



$JS(P_{G\_0}, P_{data}) = \log 2$ $JS(P_{G\_50}, P_{data}) = \log 2$ $JS(P_{G\_100}, P_{data}) = 0$

$W(P_{G\_0}, P_{data})=d_0$ $W(P_{G\_50}, P_{data})=d_{50}$ $W(P_{G\_100}, P_{data})=0$

Explaining WGAN

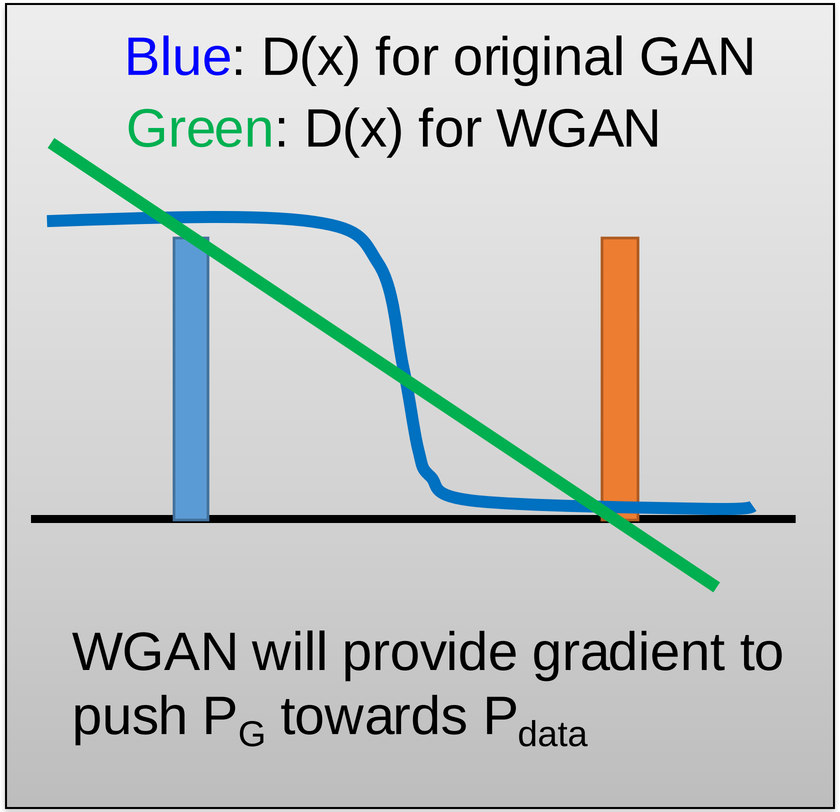
- Let W be the Wasserstein distance.

$W(P_{data}, P_G) = \max_{D \text{ is 1-Lipschitz}}[E_{x\sim P\_data} D(x) - E_{x\sim P\_G} D(x)]$

Where a function f is a
k-Lipschitz function if

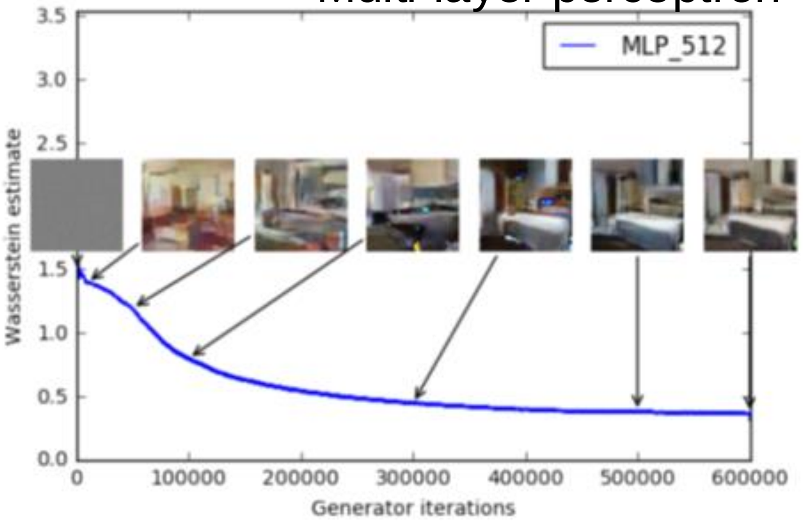
$\|f(x_1) \quad - \quad f(x_2) \quad \| \qquad \|$

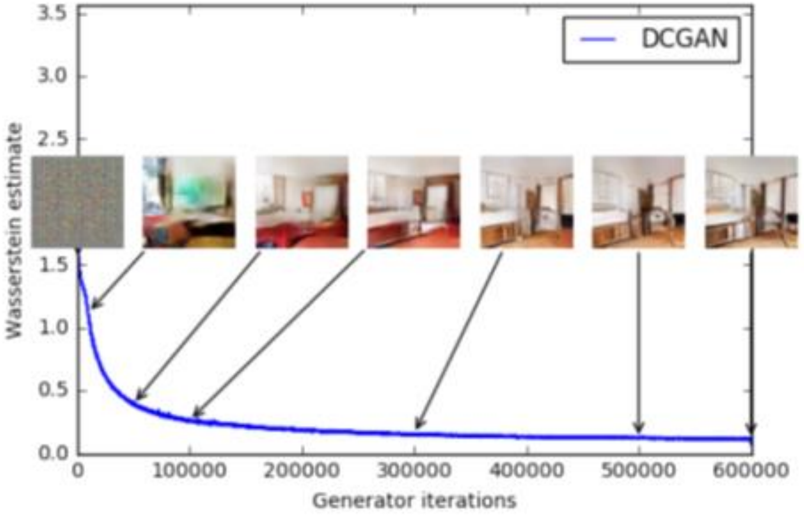
How to guarantee this?
Weight clipping: for all
parameter updates, if w>c
Then w=c, if w<-c, then w=-c.

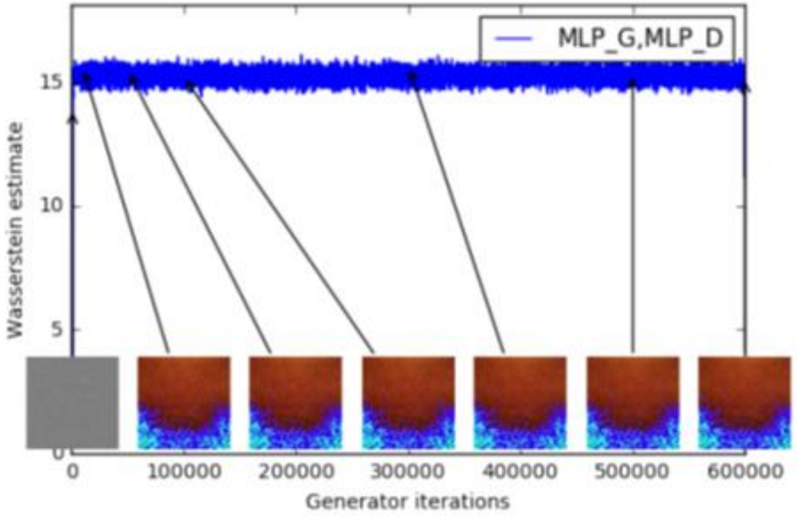


Earth Mover Distance Examples:

Multi-layer perceptron

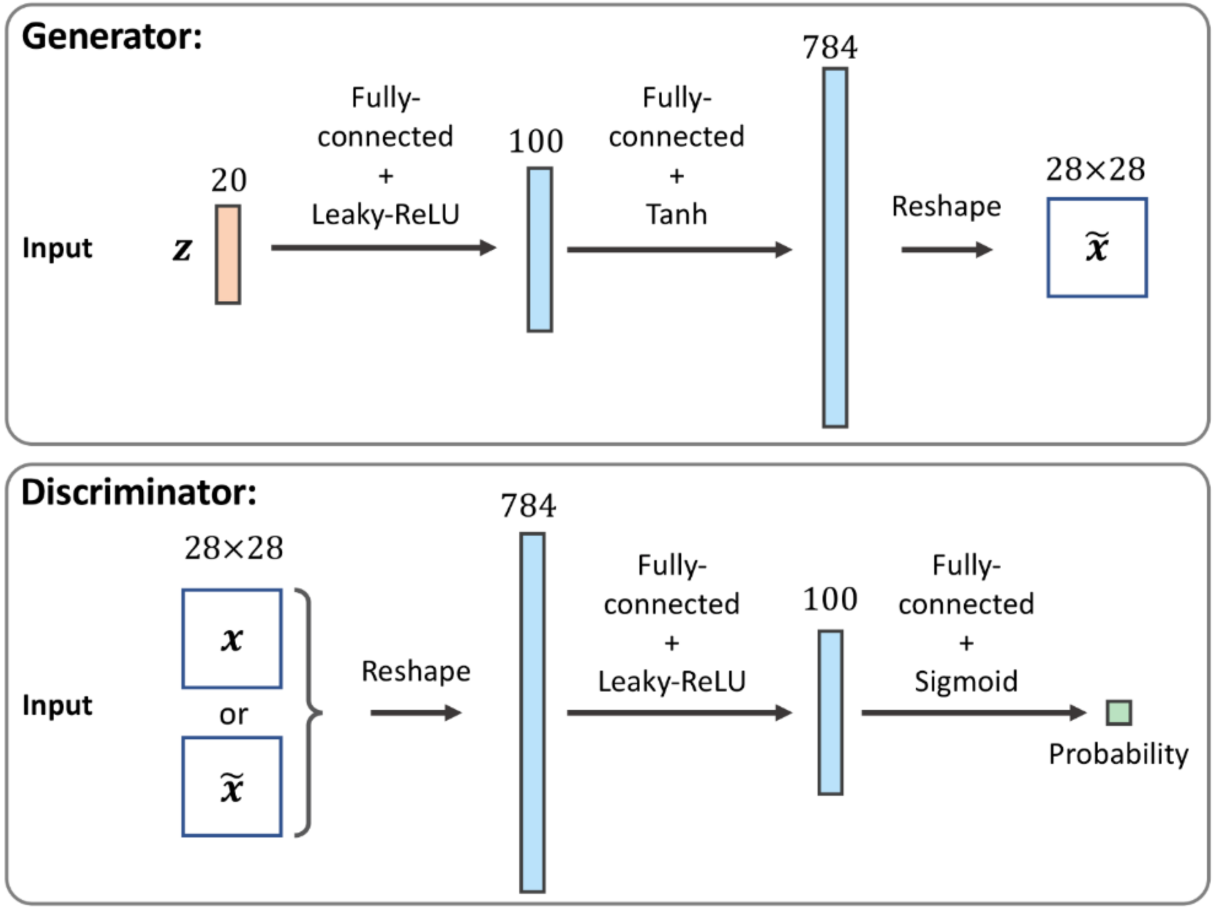






Generative Adversarial Networks Exercise

- Let's build the first GAN model's generator and discriminator as a fully connected neural network with one or more hidden layers



Generative Adversarial Networks Exercise

- Let's define the two helper functions for the two neural networks.

```
[5] import tensorflow as tf
    import tensorflow_datasets as tfds
    import numpy as np
    import matplotlib.pyplot as plt
```

```
## Define the generator function:
def make_generator_network(
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=784):
    model = tf.keras.Sequential()
    for i in range(num_hidden_layers):
        model.add(
            tf.keras.layers.Dense(
                units=num_hidden_units,
                use_bias=False)
            )
        model.add(tf.keras.layers.LeakyReLU())

    model.add(tf.keras.layers.Dense(
        units=num_output_units, activation='tanh'))
    return model
```

```
## Define the discriminator function:
def make_discriminator_network(
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=1):
    model = tf.keras.Sequential()
    for i in range(num_hidden_layers):
        model.add(tf.keras.layers.Dense(units=num_hidden_units))
        model.add(tf.keras.layers.LeakyReLU())
        model.add(tf.keras.layers.Dropout(rate=0.5))

    model.add(
        tf.keras.layers.Dense(
            units=num_output_units,
            activation=None)
        )
    return model
```

Generative Adversarial Networks Exercise

- Building generator and discriminator neural network

```python
image_size = (28, 28)
z_size = 20
mode_z = 'uniform'  # 'uniform' vs. 'normal'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)

gen_model = make_generator_network(
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))

gen_model.build(input_shape=(None, z_size))
gen_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               2000
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 784)               79184
=================================================================
Total params: 81,184
Trainable params: 81,184
Non-trainable params: 0
_________________________________________________________________
```

Generative Adversarial Networks Exercise

- Building generator and discriminator neural network

```
disc_model = make_discriminator_network(
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)

disc_model.build(input_shape=(None, np.prod(image_size)))
disc_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 100)               78500
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dropout (Dropout)            (None, 100)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 101
=================================================================
Total params: 78,601
Trainable params: 78,601
Non-trainable params: 0
```

Generative Adversarial Networks Exercise

- Define test dataset
  - The range of pixel values of the synthetic image is (-1, 1) since the output layer of the generator uses the tanh activation function.
  - pixel intensity range: [0, 1], Multiply 2 and subtract 1 to adjust the pixel intensity range to [-1, 1]
  - create a random vector z based on a random distribution

```python
mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
mnist = mnist_bldr.as_dataset(shuffle_files=False)

def preprocess(ex, mode='uniform'):
    image = ex['image']
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.reshape(image, [-1])
    image = image*2 - 1.0
    if mode == 'uniform':
        input_z = tf.random.uniform(
            shape=(z_size,), minval=-1.0, maxval=1.0)
    elif mode == 'normal':
        input_z = tf.random.normal(shape=(z_size,))
    return input_z, image
```

Generative Adversarial Networks Exercise

- Let's examine the dataset object:

```
[8] mnist_trainset = mnist['train']

    print('before preprocessing: ')
    example = next(iter(mnist_trainset))['image']
    print('dtype: ', example.dtype, ('minimum: {}, maximum: {}'.format(np.min(example), np.max(example))))

    mnist_trainset = mnist_trainset.map(preprocess)

    print('after preprocessing: ')
    example = next(iter(mnist_trainset))[0]
    print('dtype: ', example.dtype, ('minimum: {}, maximum: {}'.format(np.min(example), np.max(example))))

    before preprocessing:
    dtype:  <dtype: 'uint8'> minimum: 0, maximum: 255
    after preprocessing:
    dtype:  <dtype: 'float32'> minimum: -0.8737728595733643, maximum: 0.9460210800170898
```

Generative Adversarial Networks Exercise

HCMUTE

- In the following code, let's print the input vector and the image array shape by extracting a batch:

```
mnist_trainset = mnist_trainset.batch(32, drop_remainder=True)
input_z, input_real = next(iter(mnist_trainset))
print('input-z -- shape:', input_z.shape)
print('input-real -- shape:', input_real.shape)
```

```
input-z -- shape: (32, 20)
input-real -- shape: (32, 784)
```

Generative Adversarial Networks Exercise

- let’s run the front-propagation computation of the generator and discriminator to understand the general data flow

```
g_output = gen_model(input_z)
print('Generator output -- shape:', g_output.shape)

d_logits_real = disc_model(input_real)
d_logits_fake = disc_model(g_output)
print('discriminator (real) -- shape:', d_logits_real.shape)
print('discriminator (fake) -- shape:', d_logits_fake.shape)
```

```
Generator output -- shape: (32, 784)
discriminator (real) -- shape: (32, 1)
discriminator (fake) -- shape: (32, 1)
```

Generative Adversarial Networks Exercise

- Training GAN model

```
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)

## Generator loss
g_labels_real = tf.ones_like(d_logits_fake)
g_loss = loss_fn(y_true=g_labels_real, y_pred=d_logits_fake)
print('Generator loss:  {:.4f}'.format(g_loss))
```

```
Generator loss:  0.6961
```

$$V\left(\theta^{(D)}, \theta^{(G)}\right) = \mathbb{E}_{z \sim p_z(z)}[\log[D(G(z))]]$$

Generative Adversarial Networks Exercise

- Training GAN model

```
## Discriminator loss
d_labels_real = tf.ones_like(d_logits_real)
d_labels_fake = tf.zeros_like(d_logits_fake)

d_loss_real = loss_fn(y_true=d_labels_real, y_pred=d_logits_real)
d_loss_fake = loss_fn(y_true=d_labels_fake, y_pred=d_logits_fake)
print('Discriminator loss:  real {:.4f}, fake {:.4f}'
        .format(d_loss_real.numpy(), d_loss_fake.numpy()))
```

```
Discriminator loss:  real 1.7901, fake 0.6969
```

$$V\left(\theta^{(D)}, \theta^{(G)}\right) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log[1 - D(G(z))]]$$

Generative Adversarial Networks Exercise

- Final Training (1/6)

```python
import time


num_epochs = 100
batch_size = 64
image_size = (28, 28)
z_size = 20
mode_z = 'uniform'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)
np.random.seed(1)
```

```python
if mode_z == 'uniform':
    fixed_z = tf.random.uniform(
        shape=(batch_size, z_size),
        minval=-1, maxval=1)
elif mode_z == 'normal':
    fixed_z = tf.random.normal(
        shape=(batch_size, z_size))


def create_samples(g_model, input_z):
    g_output = g_model(input_z, training=False)
    images = tf.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0
```

Generative Adversarial Networks Exercise

- Final Training (2/6)

```
## Prepare dataset
mnist_trainset = mnist['train']
mnist_trainset = mnist_trainset.map(
    lambda ex: preprocess(ex, mode=mode_z))

mnist_trainset = mnist_trainset.shuffle(10000)
mnist_trainset = mnist_trainset.batch(
    batch_size, drop_remainder=True)
```

Generative Adversarial Networks Exercise

- Final Training (3/6)

```python
## Prepare model
with tf.device(device_name):
    gen_model = make_generator_network(
        num_hidden_layers=gen_hidden_layers,
        num_hidden_units=gen_hidden_size,
        num_output_units=np.prod(image_size))
    gen_model.build(input_shape=(None, z_size))

    disc_model = make_discriminator_network(
        num_hidden_layers=disc_hidden_layers,
        num_hidden_units=disc_hidden_size)
    disc_model.build(input_shape=(None, np.prod(image_size)))
```

```python
## Loss function and optimizer
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
g_optimizer = tf.keras.optimizers.Adam()
d_optimizer = tf.keras.optimizers.Adam()
```

Generative Adversarial Networks Exercise

- Final Training (4/6)

```python
all_losses = []
all_d_vals = []
epoch_samples = []

start_time = time.time()
for epoch in range(1, num_epochs+1):
    epoch_losses, epoch_d_vals = [], []
    for i,(input_z,input_real) in enumerate(mnist_trainset):

        ## Calculate the loss of generator.
        with tf.GradientTape() as g_tape:
            g_output = gen_model(input_z)
            d_logits_fake = disc_model(g_output, training=True)
            labels_real = tf.ones_like(d_logits_fake)
            g_loss = loss_fn(y_true=labels_real, y_pred=d_logits_fake)

        # Calculate the gradient of g_loss.
        g_grads = g_tape.gradient(g_loss, gen_model.trainable_variables)

        # Optimizer: Apply gradients.
        g_optimizer.apply_gradients(
            grads_and_vars=zip(g_grads, gen_model.trainable_variables))
```

Generative Adversarial Networks Exercise

- Final Training (5/6)

```
## Calculate the loss of discriminator.
with tf.GradientTape() as d_tape:
    d_logits_real = disc_model(input_real, training=True)

    d_labels_real = tf.ones_like(d_logits_real)

    d_loss_real = loss_fn(
        y_true=d_labels_real, y_pred=d_logits_real)

    d_logits_fake = disc_model(g_output, training=True)
    d_labels_fake = tf.zeros_like(d_logits_fake)

    d_loss_fake = loss_fn(
        y_true=d_labels_fake, y_pred=d_logits_fake)

    d_loss = d_loss_real + d_loss_fake

## Calculate the gradient of d_loss.
d_grads = d_tape.gradient(d_loss, disc_model.trainable_variables)
```

HCMUTE

Generative Adversarial Networks Exercise

- Final Training (6/6)

```
        ## Opmizer: Apply gradients
        d_optimizer.apply_gradients(
            grads_and_vars=zip(d_grads, disc_model.trainable_variables))

        epoch_losses.append(
            (g_loss.numpy(), d_loss.numpy(),
             d_loss_real.numpy(), d_loss_fake.numpy()))

        d_probs_real = tf.reduce_mean(tf.sigmoid(d_logits_real))
        d_probs_fake = tf.reduce_mean(tf.sigmoid(d_logits_fake))
        epoch_d_vals.append((d_probs_real.numpy(), d_probs_fake.numpy()))
    all_losses.append(epoch_losses)
    all_d_vals.append(epoch_d_vals)
    print(
        'epoch {:03d} | time  {:.2f} min | Average loss >>'
        ' Generator/Discriminator {:.4f}/{:.4f} [Discriminator-Real]: {:.4f} Discriminator-Fake: {:.4f}]'
        .format(
            epoch, (time.time() - start_time)/60,
            *list(np.mean(all_losses[-1], axis=0))))
    epoch_samples.append(
        create_samples(gen_model, fixed_z).numpy())
```

```
epoch 001 | time 1.23 min | Average loss >>Generator/Discriminator 3.0232/0.3027 [Discriminator-Real: 0.0320 Discriminator-Fake: 0.2707]
epoch 002 | time 2.01 min | Average loss >>Generator/Discriminator 4.9142/0.3012 [Discriminator-Real: 0.0941 Discriminator-Fake: 0.2071]
epoch 003 | time 2.77 min | Average loss >>Generator/Discriminator 3.9093/0.6338 [Discriminator-Real: 0.2792 Discriminator-Fake: 0.3546]
epoch 004 | time 3.54 min | Average loss >>Generator/Discriminator 1.9743/0.9225 [Discriminator-Real: 0.4597 Discriminator-Fake: 0.4628]
epoch 005 | time 4.90 min | Average loss >>Generator/Discriminator 2.2160/0.7707 [Discriminator-Real: 0.4182 Discriminator-Fake: 0.3525]
epoch 006 | time 5.66 min | Average loss >>Generator/Discriminator 1.9331/0.8516 [Discriminator-Real: 0.4704 Discriminator-Fake: 0.3812]
epoch 007 | time 6.41 min | Average loss >>Generator/Discriminator 1.7033/0.9588 [Discriminator-Real: 0.5177 Discriminator-Fake: 0.4411]
```

Generative Adversarial Networks Exercise

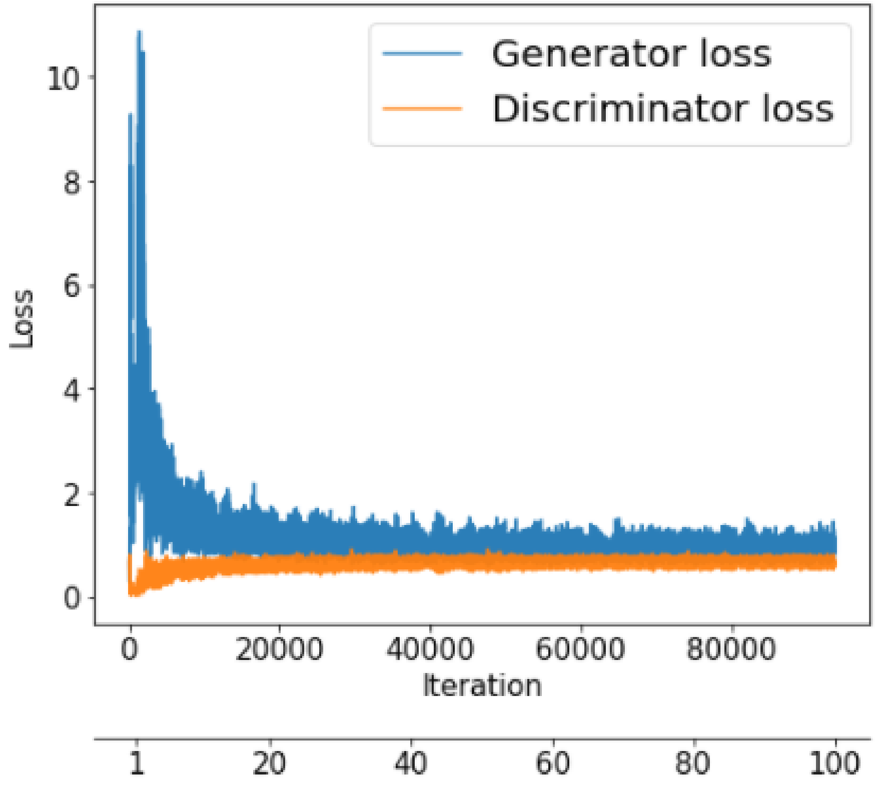
- Also, it is helpful to print the average probability of the real and fake samples computed by the discriminator for each iteration.
- If this probability is close to 0.5, the discriminator cannot distinguish between real and fake images.

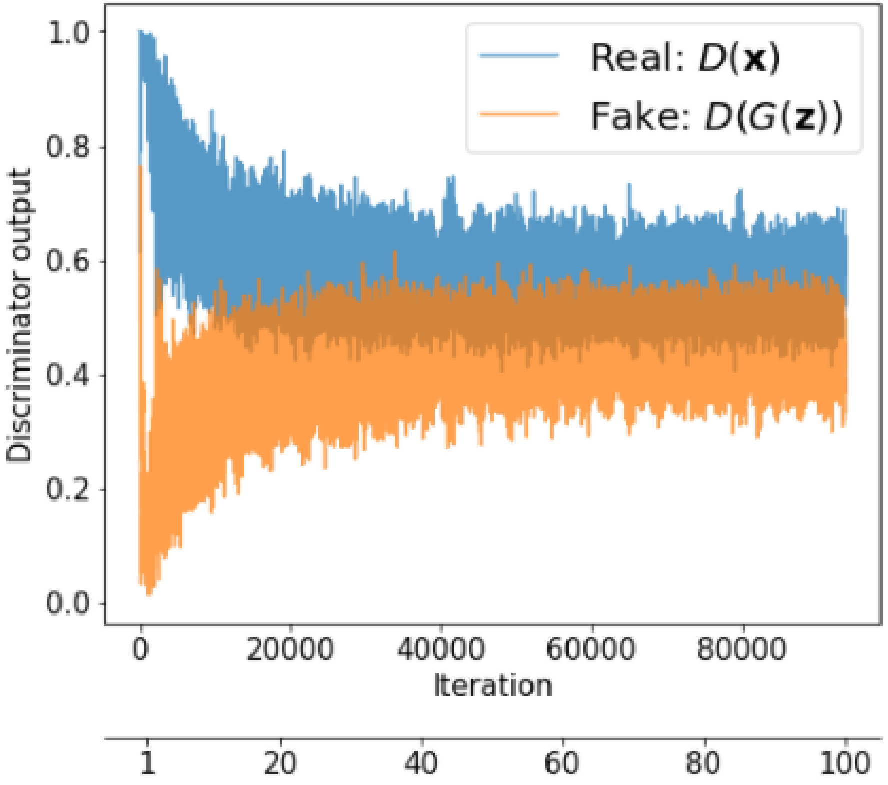
```python
import itertools


fig = plt.figure(figsize=(16, 6))

  ## Loss graph
ax = fig.add_subplot(1, 2, 1)
g_losses = [item[0] for item in itertools.chain(*all_losses)]
d_losses = [item[1]/2.0 for item in itertools.chain(*all_losses)]
plt.plot(g_losses, label='Generator loss', alpha=0.95)
plt.plot(d_losses, label='Discriminator loss', alpha=0.95)
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

epochs = np.arange(1, 101)
epoch2iter = lambda e: e*len(all_losses[-1])
epoch_ticks = [1, 20, 40, 60, 80, 100]
newpos = [epoch2iter(e) for e in epoch_ticks]
ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)
```

```python
  ## Print discriminator
ax = fig.add_subplot(1, 2, 2)
d_vals_real = [item[0] for item in itertools.chain(*all_d_vals)]
d_vals_fake = [item[1] for item in itertools.chain(*all_d_vals)]
plt.plot(d_vals_real, alpha=0.75, label=r'Real: $D(\mathbf{x})$')
plt.plot(d_vals_fake, alpha=0.75, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)

plt.show()
```

Generative Adversarial Networks Exercise

- Also, it is helpful to print the average probability of the real and fake samples computed by the discriminator for each iteration.
- If this probability is close to 0.5, the discriminator cannot distinguish between real and fake images.

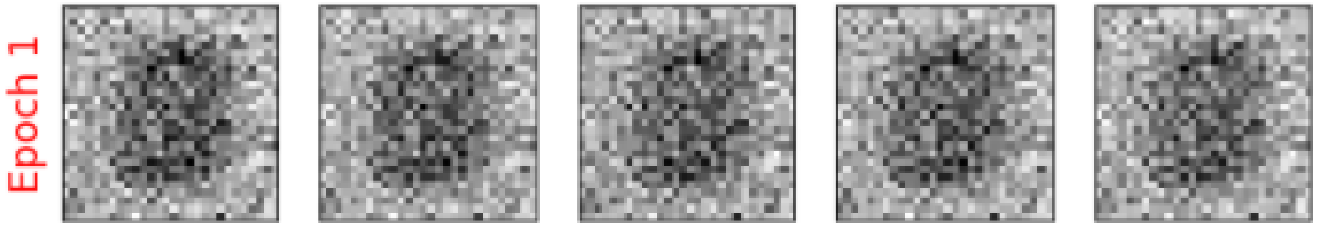




Generative Adversarial Networks Exercise

- Training GAN model

In [25]:
```
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, 'Epoch {}'.format(e),
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)

        image = epoch_samples[e-1][j]
        ax.imshow(image, cmap='gray_r')

plt.show()
```



Generative Adversarial Networks Exercise

- Training GAN model