Generative Deep Learning

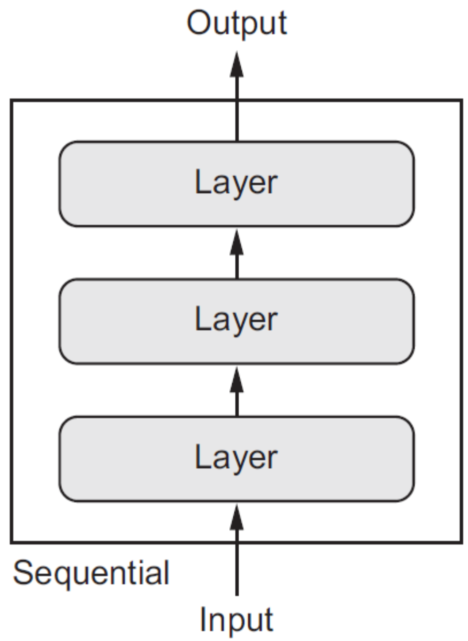
Outline

- **Keras functional API**
- **Variational autoencoders (VAE)**
- **Understanding generative adversarial networks (GAN)**

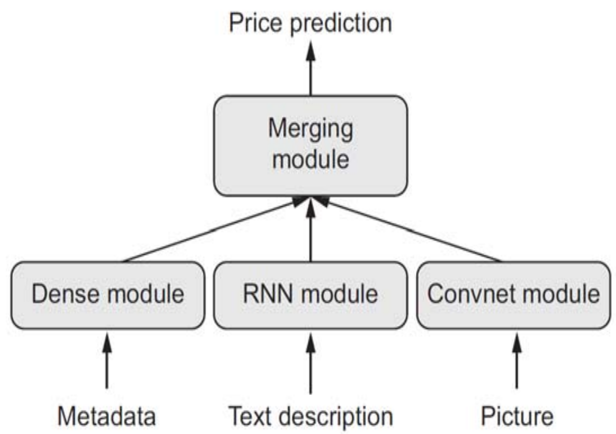
Keras functional API

The Keras functional API

- The *Sequential* model makes the assumption that the network has exactly **one input** and exactly **one output**, and that it consists of a linear stack of layers.
- Some networks require
  - several independent inputs,
  - multiple outputs, and
  - **internal branching** between layers that makes them look like ***graphs*** of layers rather than linear stacks of layers.

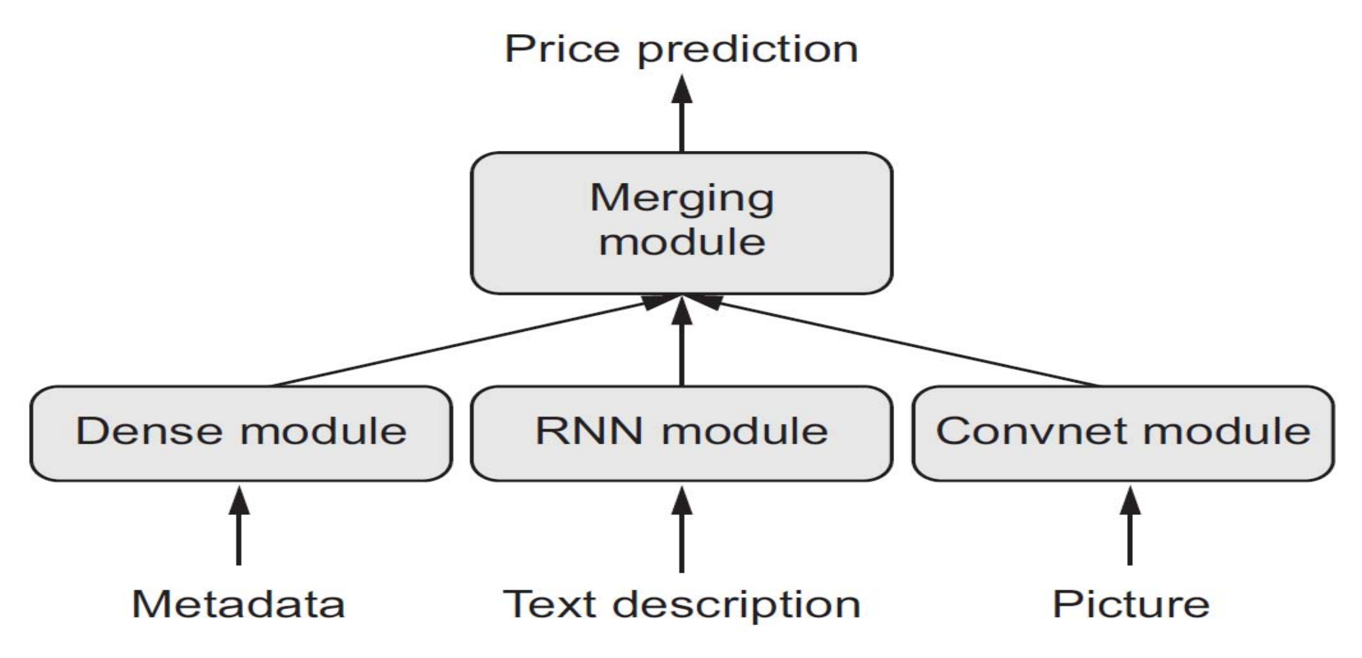


- ***Multimodal* inputs** they merge data coming from different input sources, processing each type of data using different kinds of neural layers.
  - Predict market price of a **second-hand piece** of clothing
  - inputs: user-provided **metadata** (such as the item's brand, age, and so on), a user-provided **text description**, and a **picture** of the item.



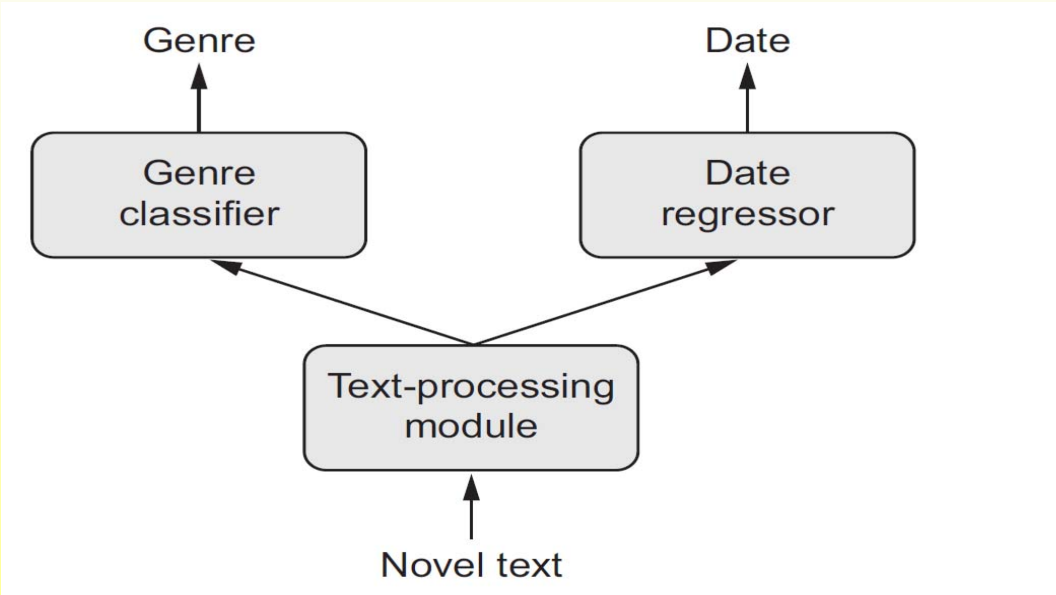
A multi-input model

- But how can you use all three at the same time?
  - A naive approach would be to **train three separate models** and then do a weighted average of their predictions. But this may **be suboptimal**, because the information extracted by the models may be **redundant**.
  - A better way is to *jointly* learn a more accurate model of the data by using a model that can see all available input modalities simultaneously: a model with three input branches.



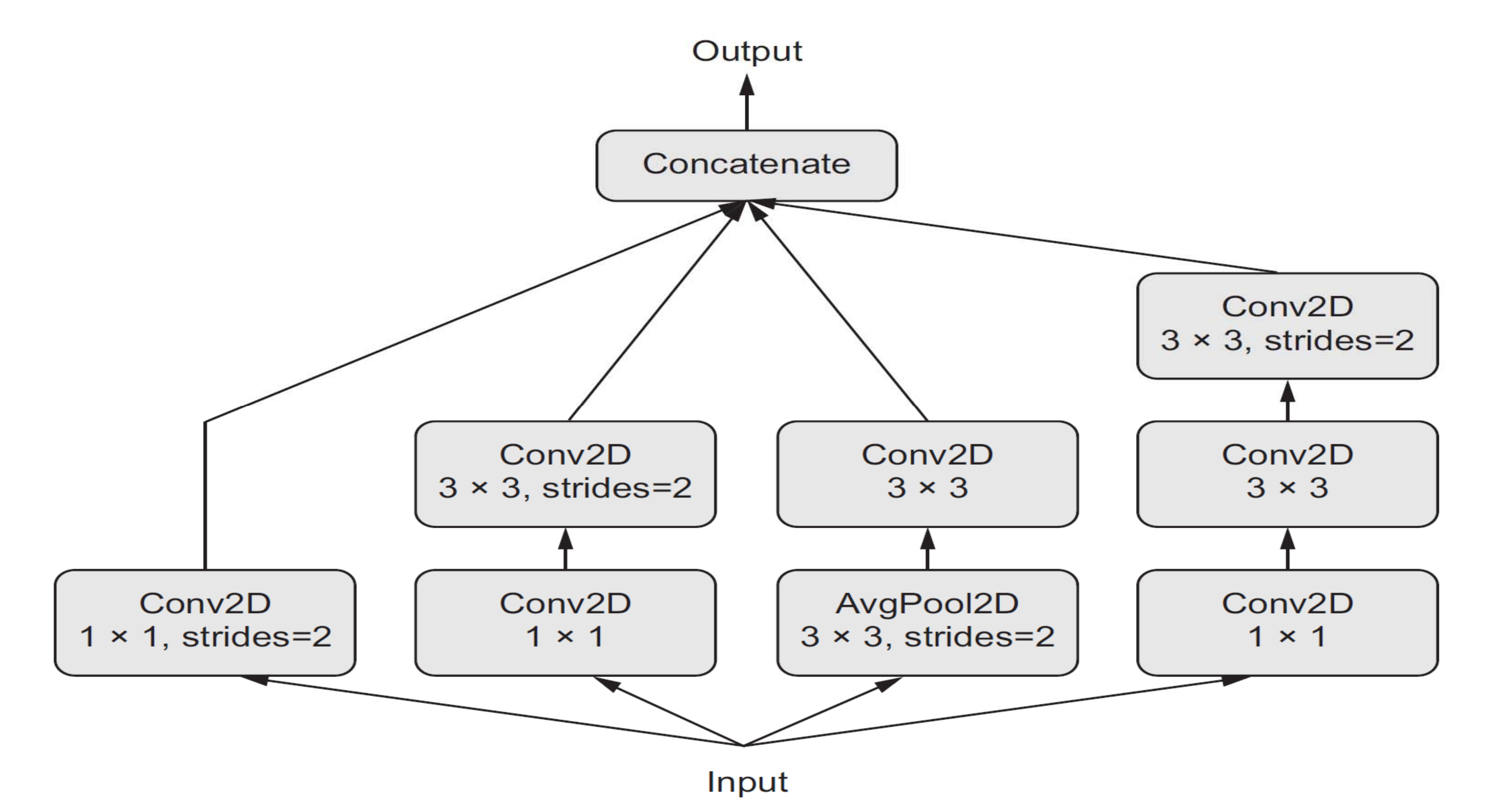
A multi-output (or multihead) model

- Given the text of a novel or short story, you might want to automatically **classify** it by genre (類型) (such as romance or thriller) but also **predict the** approximate date it was written.
- Of course, you could train two separate models: one for the genre and one for the date.
- But because these attributes aren’t statistically independent, you could build a better model by learning to jointly predict both genre and date at the same time.
- Such a joint model would then have two outputs, or *heads*.
- **Co-relation issue**



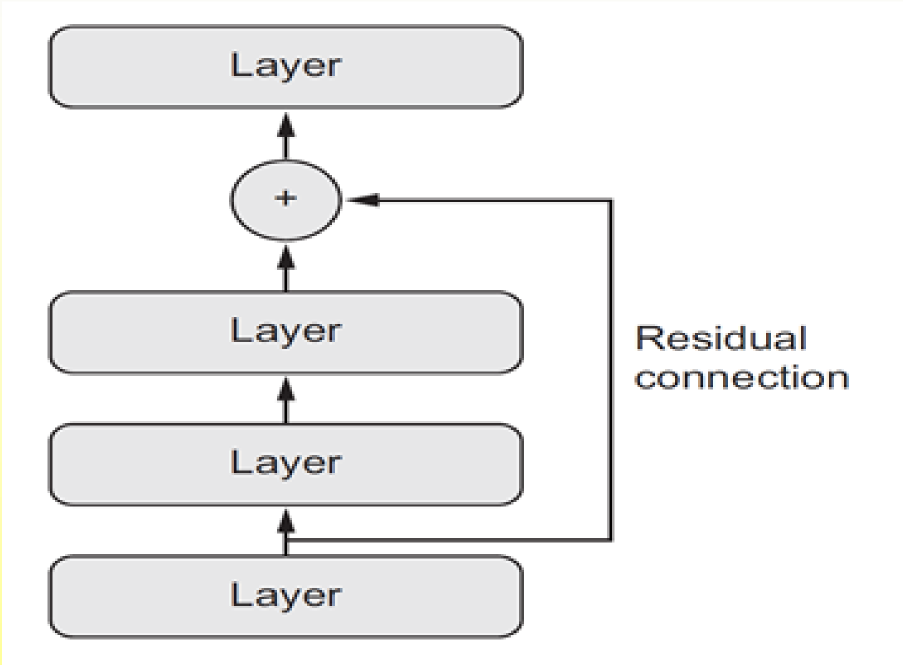
Inception modules

- Christian Szegedy et al., “Going Deeper with Convolutions,” Conference on Computer Vision and Pattern Recognition (2014), https://arxiv.org/abs/1409.4842.



ResNet family of networks

- Adding ***residual connections*** to a model, which started with the ResNet family of networks (developed by He et al. at Microsoft)
- A residual connection consists of reinjecting previous representations into the downstream flow of data by adding a past output tensor to a later output tensor which **helps prevent information loss along the data-processing flow**.
- There are many other examples of such **graph-like networks.**



Introduction to the functional API

- But there's another far more general and flexible way to use Keras: the ***functional API***.
- In the functional API, you **directly manipulate tensors**, and you **use layers as *functions*** that take tensors and return tensors.

```
from keras import Input, layers

input_tensor = Input(shape=(32,))            <- A tensor

dense = layers.Dense(32, activation='relu')  <- A layer is a function.

output_tensor = dense(input_tensor)          <- A layer may be called on a
                                                tensor, and it returns a tensor.
```

A simple Sequential model and its equivalent in the functional API:

Sequential model

```
from keras.models import Sequential, Model
from keras import layers
from keras import Input

seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Sequential model, which you already know about

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = Model(input_tensor, output_tensor)

model.summary()
```

Its functional equivalent

functional API

The Model class turns an input tensor and output tensor into a model.

Let's look at it!

model.summary()

```
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
input_1 (InputLayer)         (None, 64)                0
________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
________________________________________________________________
dense_3 (Dense)              (None, 10)                330
================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
________________________________________________________________
```

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = Model(input_tensor, output_tensor)

model.summary()
```

Its functional equivalent

functional API

The Model class turns an input tensor and output tensor into a model.

Let's look at it!

Model object

- Keras retrieves every layer involved in going from input_tensor to output_tensor, bringing them together into a graph-like data structure — a **Model**.
- Of course, the reason it works is that **output_tensor was obtained by repeatedly transforming input_tensor.**

```
>>> unrelated_input = Input(shape=(32,))
>>> bad_model = model = Model(unrelated_input, output_tensor)

RuntimeError: Graph disconnected: cannot
obtain value for tensor
➥Tensor("input_1:0", shape=(?, 64), dtype=float32) at layer "input_1".
```

Compiling, Training, or Evaluating

- compiling, training, or evaluating such an instance of Model, the API is the same as that of Sequential:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')   <- Compiles the model

import numpy as np                                                    <- Generates dummy Numpy data to train on
x_train = np.random.random((1000, 64))
y_train = np.random.random((1000, 10))

model.fit(x_train, y_train, epochs=10, batch_size=128)                <- Trains the model for 10 epochs

score = model.evaluate(x_train, y_train)                              <- Evaluates the model
```

Variational Autoencoders (VAEs)

變分自編碼器

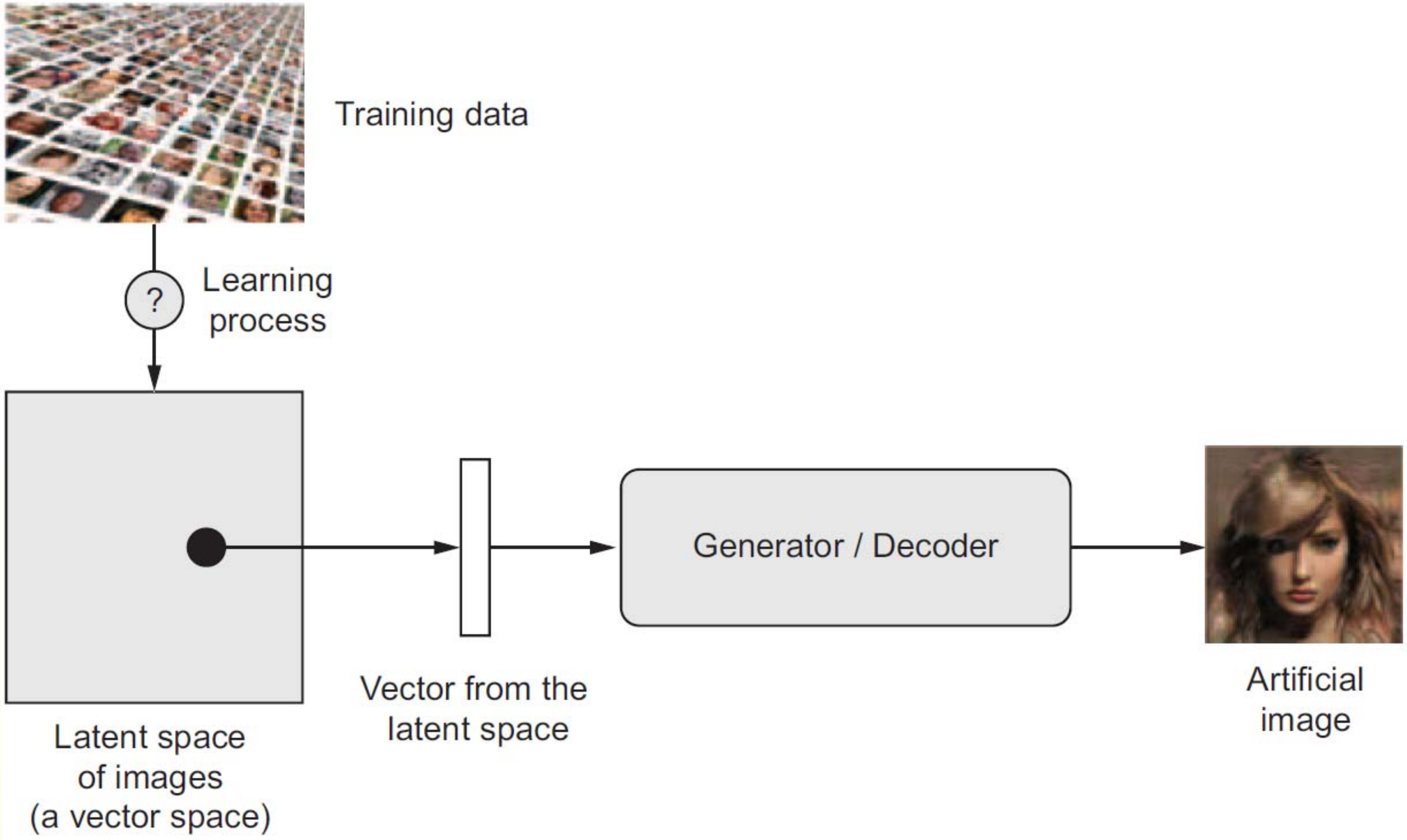
Generating images with Variational Autoencoders

- Sampling from a **latent space (潛在空間)** of images to create entirely new images or edit existing ones is currently the most popular and successful application of creative AI.
- We'll review some high-level concepts pertaining to image generation, alongside implementations details relative to the two main techniques in this domain: ***variational autoencoders*** **(VAEs)** and ***generative adversarial networks*** **(GANs) (生成對抗網路)**.
- The techniques we present here **aren't** specific to images — you could develop latent spaces of **sound**, **music**, or even **text**, using GANs and VAEs—but in practice, the most interesting results have been obtained with pictures, and that's what we focus on here.

Sampling from latent spaces of images

- The key idea of image generation is to develop a low-dimensional *latent space* of representations (which naturally is **a vector space**) where any point can be **mapped to a realistic-looking image**.
- The module capable of realizing this mapping, taking as **input a latent point and outputting an image (a grid of pixels)**, is called a ***generator*** (in the case of **GANs**) or a ***decoder*** (in the case of **VAEs)**.
- Once such a latent space has been developed, you can sample points from it, either deliberately (有意的) or at random, and, by mapping them to image space, generate images that have never been seen before.

Learning a **latent vector** space of images, and using it to sample new images



VAES vs. GANs

- **VAEs** are great for learning latent spaces that are well structured (結構良好的), where specific **directions** encode a meaningful **axis** of variation in the data.
- **GANs** generate images that can potentially be highly **realistic (逼真)**, but the latent space they come from may not have as much structure and continuity.

A continuous space of faces generated by Tom White using VAEs

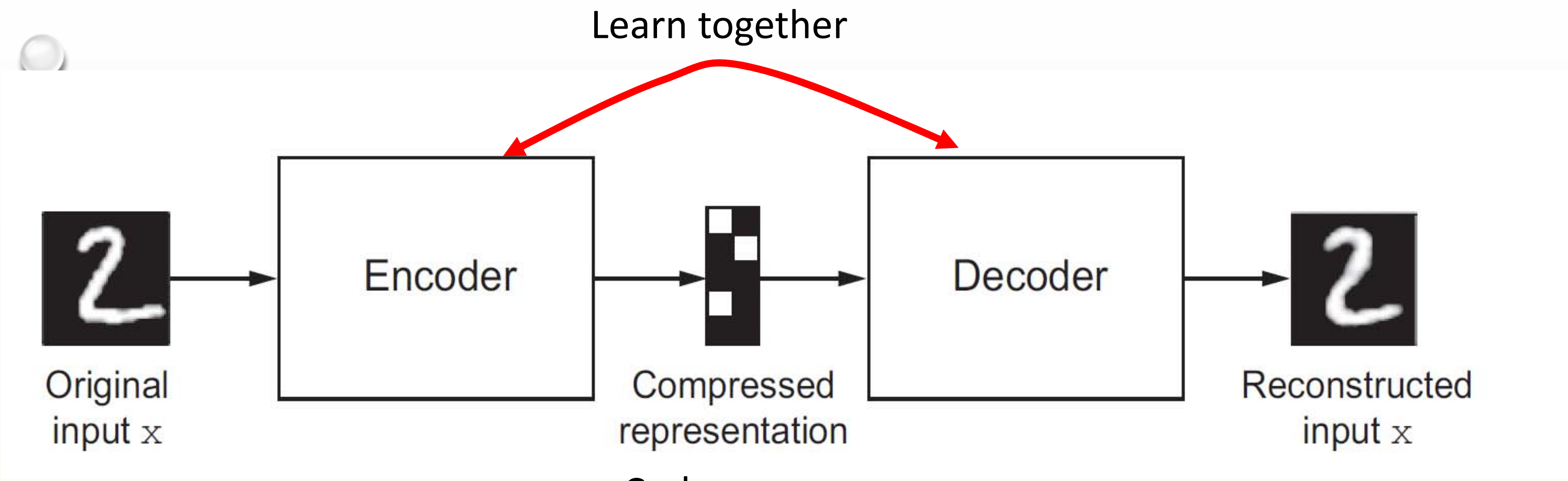
Concept vectors for image editing

- ***Concept vector*** **(概念向量)**
  - given a latent space of representations, or an embedding space, certain directions in the space may encode interesting axes of variation in the original data.
- In a latent space of images of faces, there may be a *smile vector* s, such that if latent point z is the embedded representation of a certain face, then latent point z + s is the embedded representation of the **same face, smiling**.
- Once you've identified such a vector, it then becomes possible to **edit images by projecting them into the latent space**, moving their representation in a meaningful way, and then decoding them back to image space.

Smile face example

- There are concept vectors for essentially any independent dimension of variation in image space—in the case of faces, you may discover vectors for adding sunglasses to a face, removing glasses, turning a male face into as female face, and so on.
- VAE on CelebA dataset

Auto-encoder (AE) and Decoder



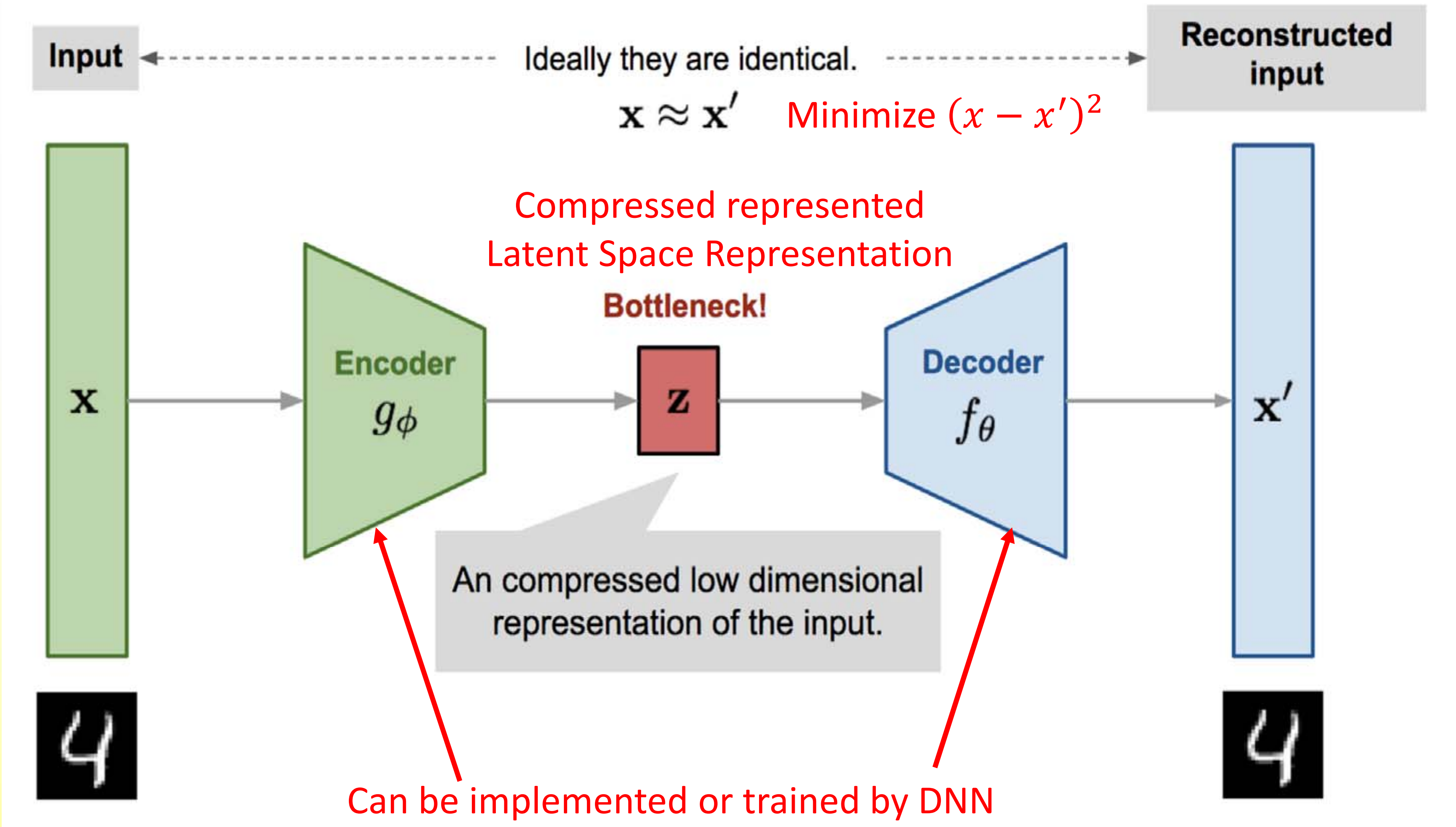
28*28=784

Code
Size 30 <784

28*28=784

- Application
  - Text retrieval
    - Document->vector (or code) using **bag-of-word** method
  - Image retrieval
    - Image-> vector

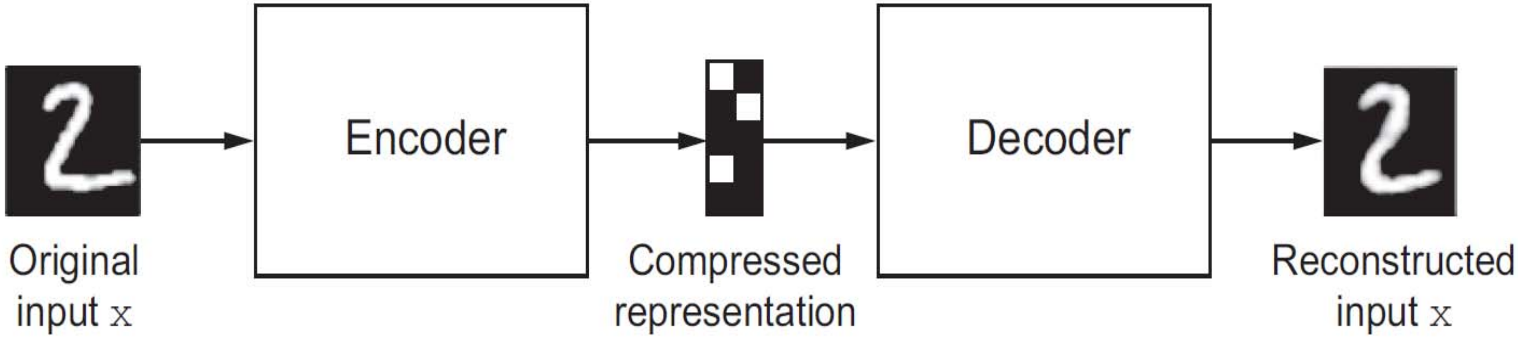
Autoencoder (自動編碼器)



Autoencoder

- A classical image autoencoder takes an image, maps it to a **latent vector space** via an **encoder module**, and then decodes it back to an output with the same dimensions as the original image, via a decoder modules.
- It’s then trained by using as target data the *same images* as the input images, meaning the autoencoder learns to reconstruct the original inputs.
- By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more-or-less interesting latent representations of the data.
- Most commonly, you’ll constrain the code to be low-dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

Autoencoder



- **Drawback**
  - Classical autoencoders don't lead to particularly useful or nicely structured latent spaces.
  - They're not much good at compression.
  - For these reasons, they have largely fallen out of fashion.
- VAEs
  - augment autoencoders with a little bit of **statistical magic** that **forces them to learn continuous, highly structured latent spaces**.
  - They have turned out to be a powerful tool for **image generation**.

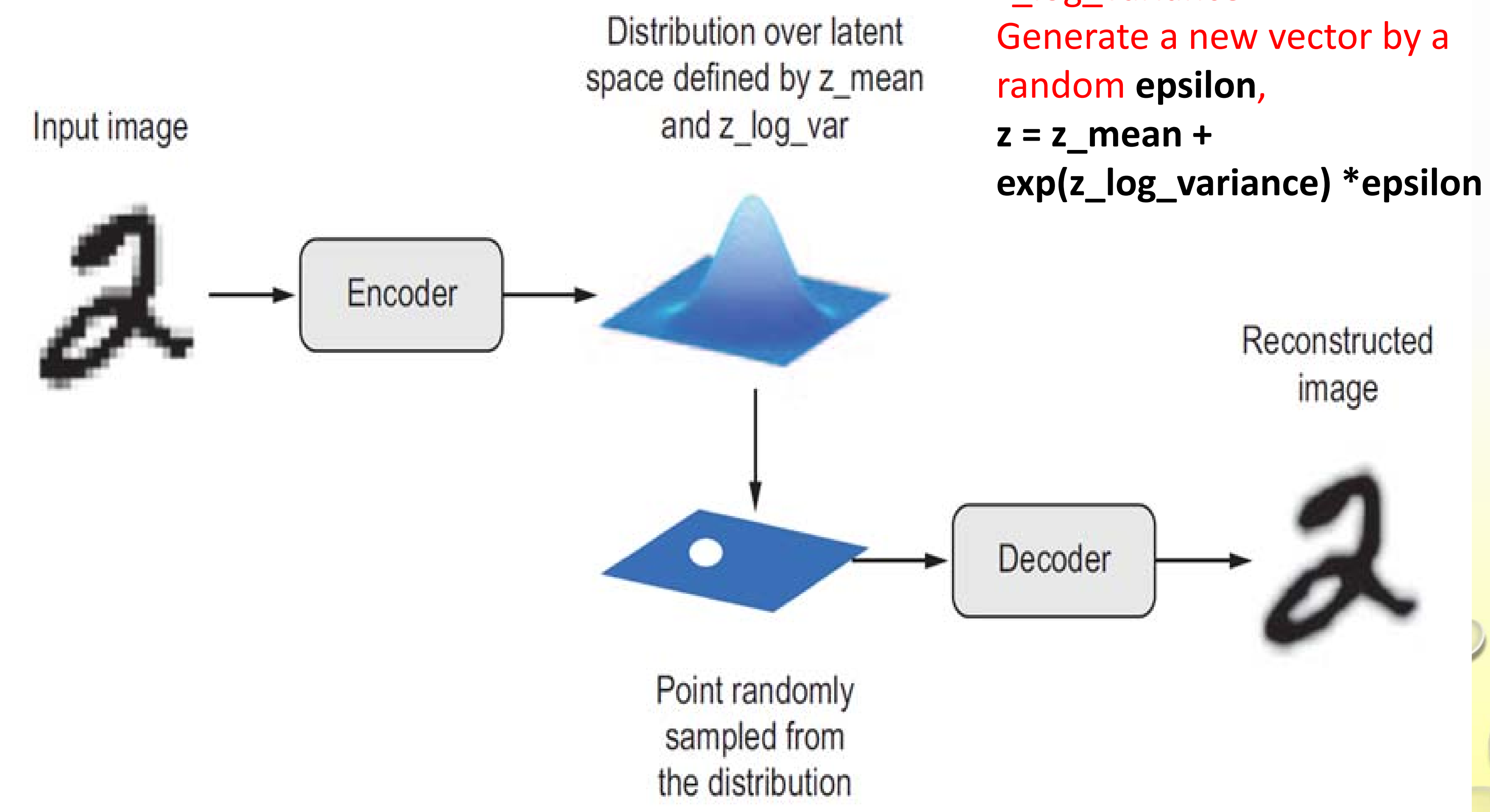
Variational Autoencoders (VAE)

- **Variational AutoEncoders (VAE)**
  - Discovered by
    - Kingma and Welling in December 2013 and
    - Rezende, Mohamed, and Wierstra in January 2014.
  - A kind of generative model that's especially appropriate for the task of image editing via **concept vectors**.
  - VAE
    - a type of network that aims to encode an input to a **low-dimensional latent space** and then decode it back—that mixes ideas from deep learning with Bayesian inference.
  - **Unsupervised learning**
  - **Internal Representation**
  - 可以看做是對輸入的資料做壓縮(維度限制)或是加入雜訊到輸入資料

Variational Autoencoder (VAE)

- A VAE, instead of compressing its input image into a fixed code in the latent space, **turns the image into the parameters of a statistical distribution: a mean and a variance**.
- Essentially, this means you're **assuming the input image has been generated by a statistical process**, and that the randomness of this process should be taken into accounting during encoding and decoding.
- The VAE then uses the **mean and variance parameters** to randomly sample one element of the distribution, and decodes that element back to the original input.
- The **stochasticity** of this process improves robustness and forces the latent space to encode meaningful representations everywhere: **every point sampled in the latent space is decoded to a valid output**.

VAE Example



Two vectors z_mean, z_log_variance
Generate a new vector by a random **epsilon**,
z = z_mean + exp(z_log_variance) *epsilon

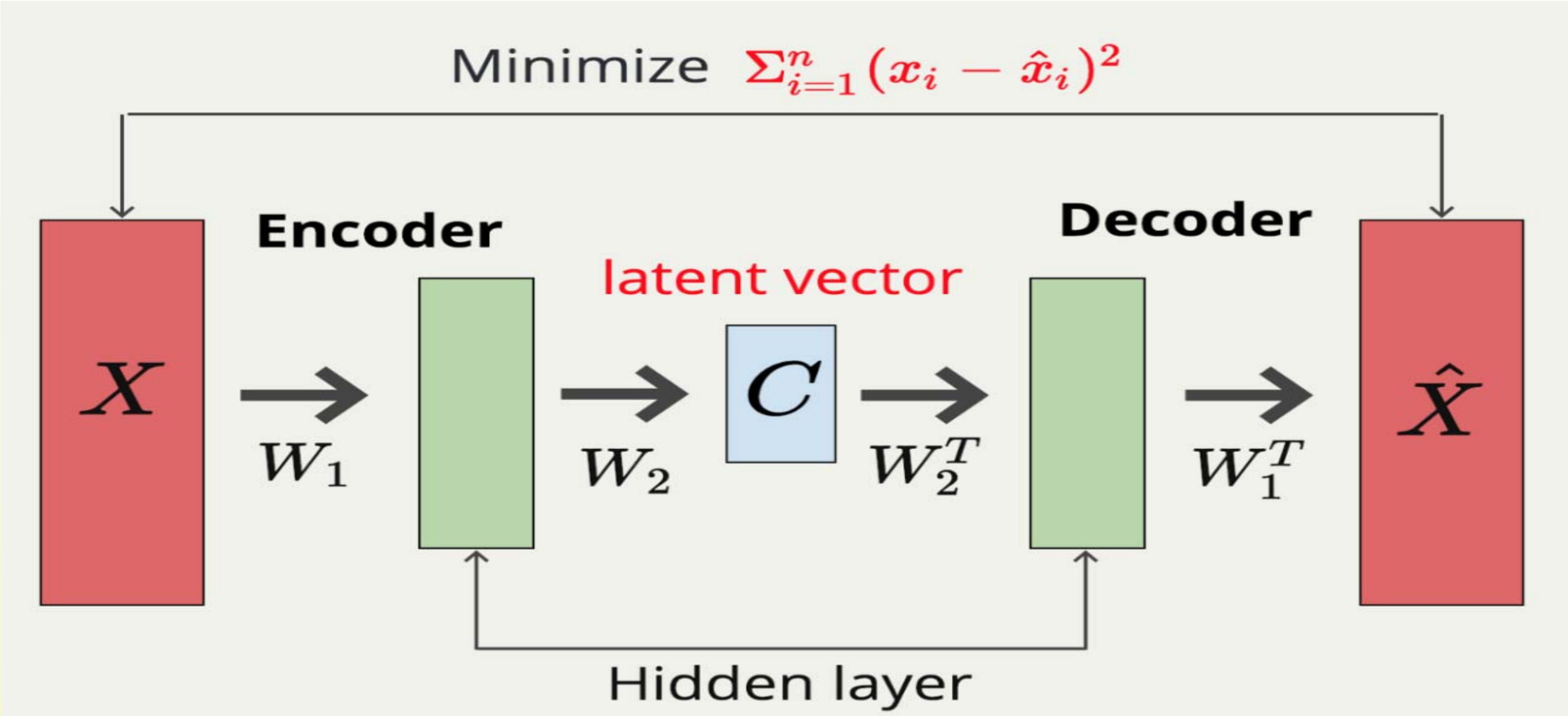
VAE Process

- In technical terms, here's how a VAE works:
  - An encoder module turns the input samples input_img into two parameters in a latent space of representations, **z_mean** and **z_log_variance**.
  - You randomly sample a point **z** from the latent normal distribution that's assumed to generate the input image, via **z = z_mean + exp(z_log_variance) *epsilon**, where epsilon is a random tensor of small values.
  - A decoder module maps this point in the latent space back to the original input image.

- Because epsilon is random, the process ensures that every point that's close to the latent location where you encoded **input_img (z-mean)** can be decoded to something similar to **input_img**, thus forcing the latent space to be continuously meaningful.
- **Any two close points in the latent space will decode to highly similar images.**

VAE Training

- 為了取得中間的code輸入表示，我們將code輸入到解碼器decoder中，得到一個輸出，如果這個輸出與輸入input很像的話，那我們就可以相信這個中間向量code跟輸入是存在某種關係的，也就是存在某種映射。
- 那麼這個中間向量code就可以作為輸入的一個特徵向量。我們通過調整encoder和decoder的參數，使得輸入和最後的輸出之間的**誤差最小**，這時候code就是輸入input的一個表示。



Loss functions

- The parameters of a VAE are trained via two loss functions:
  - a *reconstruction loss (重建損失)* that forces the decoded samples to match the initial inputs, and
  - a *regularization loss (常規化損失)* that helps learn well-formed latent spaces and reduce overfitting to the training data.
- Let's quickly go over a Keras implementation of a VAE.
- Schematically, it looks like this:

```
z_mean, z_log_variance = encoder(input_img)

z = z_mean + exp(z_log_variance) * epsilon

reconstructed_img = decoder(z)

model = Model(input_img, reconstructed_img)
```

Encodes the input into a mean and variance parameter

Draws a latent point using a small random epsilon

Decodes z back to an image

Instantiates the autoencoder model, which maps an input image to its reconstruction

VAE encoder (Keras API model)

```
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np
```

```
import keras
from keras import layers
# from keras import backend as K #網路建議
from tensorflow.keras import backend as K
from keras.models import Model
import numpy as np
```

```
img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2
```

Dimensionality of the latent space: a 2D plane

潛在空間維度

```
input_img = keras.Input(shape=img_shape)
```

(?, 28, 28, 1)

```
x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
```

(?, 28, 28, 32)

```
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
```

(?, 14, 14, 64)

```
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
```

(?, 14, 14, 64)

```
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
```

(?, 14, 14, 64)

```
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)
```

(?, 32)

```
z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
```

The input image ends up being encoded into these two parameters.

Sampling

- Next is the code for using **z_mean** and **z_log_var,** the parameters of the statistical distribution assumed to have produced **input_img**, to generate a latent space point **z**.
- Here, you wrap some arbitrary code (built on top of Keras backend primitives) into a **Lambda layer**.
- In Keras, everything needs to be a layer, so code that isn't part of a built in layer should be wrapped in a **Lambda** (or in a custom layer).
- 只是想對流經該層的數據做個變換，而這個變換本身沒有什麼需要學習的參數，那麼直接用Lambda Layer。

Listing 8.24 Latent-space-sampling function

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
```

VAE decoder

```
decoder_input = layers.Input(K.int_shape(z)[1:])
```
Input where you'll feed z

```
x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)
```
Upsamples the input

```
x = layers.Reshape(shape_before_flattening[1:])(x)

x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)
```
Uses a Conv2DTranspose layer and Conv2D layer to decode z into a feature map the same size as the original image input

Reshapes z into a feature map of the same shape as the feature map just before the last Flatten layer in the encoder model

```
decoder = Model(decoder_input, x)

z_decoded = decoder(z)
```
Instantiates the decoder model, which turns "decoder_input" into the decoded image

Applies it to z to recover the decoded z

Compute the VAE loss

```
class CustomVariationalLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)
        kl_loss = -5e-4 * K.mean(
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x

y = CustomVariationalLayer()([input_img, z_decoded])
```

You implement custom layers by writing a call method.

You don't use this output, but the layer must return something.

Calls the custom layer on the input and the decoded output to obtain the final model output

Training the VAE

Listing 8.27 Training the VAE

```
from keras.datasets import mnist

vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
```

Show the result

Listing 8.28 Sampling a grid of points from the 2D latent space and decoding them to images

```
import matplotlib.pyplot as plt
from scipy.stats import norm

n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()
```

You'll display a grid of 15 × 15 digits (255 digits total).

Transforms linearly spaced coordinates using the SciPy ppf function to produce values of the latent variable z (because the prior of the latent space is Gaussian)

Repeats z multiple times to form a complete batch

Reshapes the first digit in the batch from 28 × 28 × 1 to 28 × 28

Decodes the batch into digit images

Result

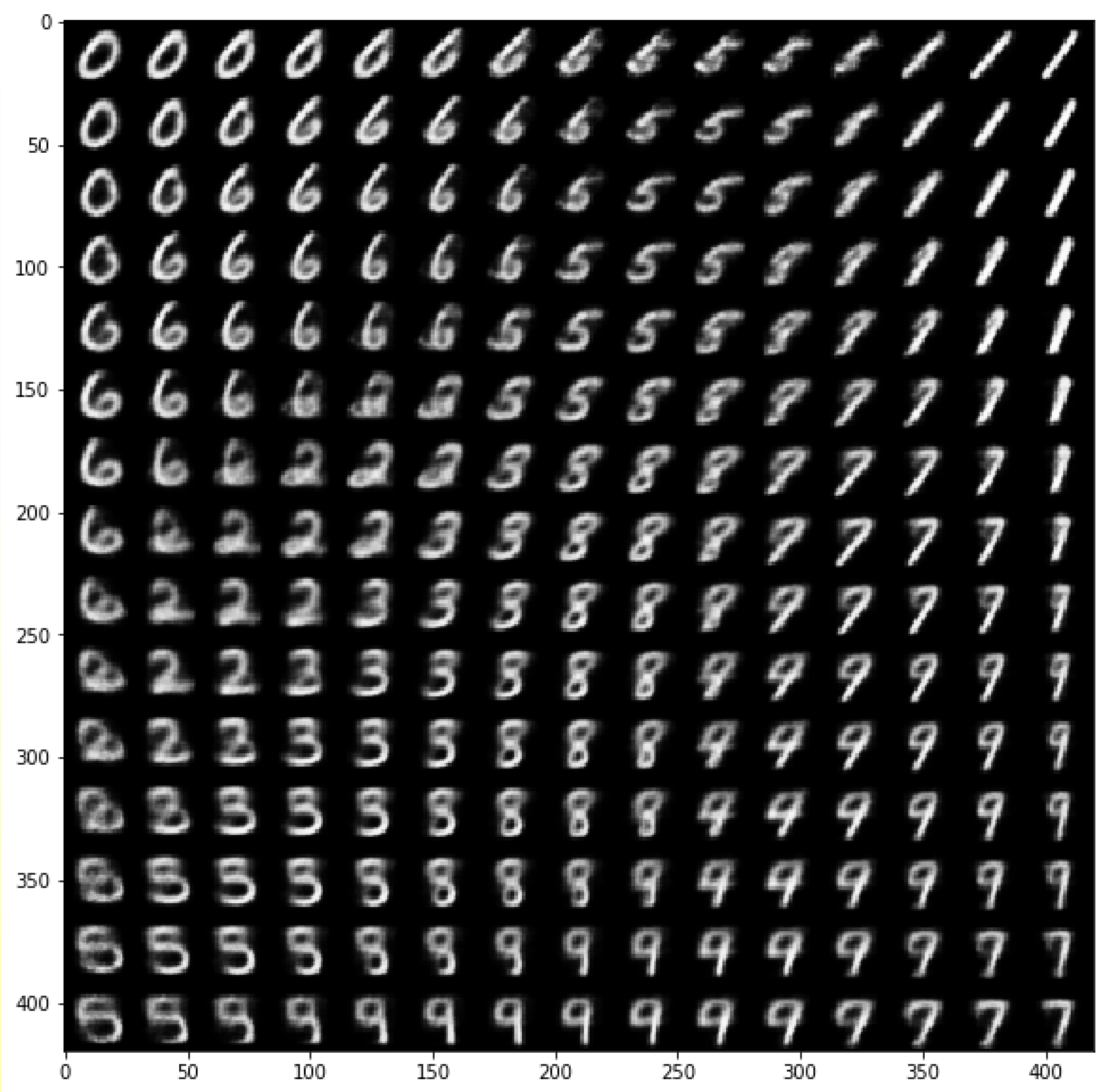
```
In [14]: runfile('E:/深度學習範例程式/F9379/ch8/8_23.py', wdir='E:/深度學習範例程式/F9379/ch8')
(None, 28, 28, 1)
(None, 14, 14, 64)
(None, 14, 14, 64)
(None, 32)
(None, 2)
(None, 12544)
(None, 14, 14, 64)
(None, None, None, 32)
(None, None, None, 1)
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_12 (InputLayer)        (None, 2)                 0
_________________________________________________________________
dense_7 (Dense)              (None, 12544)             37632
_________________________________________________________________
reshape_1 (Reshape)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 32)        18464
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 1)         289
=================================================================
Total params: 56,385
Trainable params: 56,385
Non-trainable params: 0
```

```
(None, 28, 28, 1)
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_11 (InputLayer)           (None, 28, 28, 1)    0
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 28, 28, 32)   320         input_11[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 14, 14, 64)   18496       conv2d_5[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 14, 14, 64)   36928       conv2d_6[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 14, 14, 64)   36928       conv2d_7[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 12544)        0           conv2d_8[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 32)           401440      flatten_2[0][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 2)            66          dense_4[0][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 2)            66          dense_4[0][0]
__________________________________________________________________________________________________
lambda_2 (Lambda)               (None, 2)            0           dense_5[0][0]
                                                                 dense_6[0][0]
__________________________________________________________________________________________________
model_1 (Model)                 (None, 28, 28, 1)    56385       lambda_2[0][0]
__________________________________________________________________________________________________
custom_variational_layer_1 (Cus [(None, 28, 28, 1),  0           input_11[0][0]
                                                                 model_1[1][0]
==================================================================================================
Total params: 550,629
Trainable params: 550,629
Non-trainable params: 0
```

Result

```
C:\ProgramData\Anaconda3\lib\site-packages\keras\engine\training_utils.py:819: UserWarning: Output
custom_variational_layer_1 missing from loss dictionary. We assume this was done on purpose. The fit and evaluate
APIs will not be expecting any data to be passed to custom_variational_layer_1.
  'be expecting any data to be passed to {0}.'.format(name))
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 49s 820us/step - loss: 0.2156 - val_loss: 0.1978
Epoch 2/10
60000/60000 [==============================] - 50s 828us/step - loss: 0.1950 - val_loss: 0.1915
Epoch 3/10
60000/60000 [==============================] - 51s 845us/step - loss: 0.1908 - val_loss: 0.1895
Epoch 4/10
60000/60000 [==============================] - 52s 859us/step - loss: 0.1884 - val_loss: 0.1890
Epoch 5/10
60000/60000 [==============================] - 53s 877us/step - loss: 0.1866 - val_loss: 0.1861
Epoch 6/10
60000/60000 [==============================] - 53s 877us/step - loss: 0.1853 - val_loss: 0.1878
Epoch 7/10
60000/60000 [==============================] - 53s 888us/step - loss: 0.1843 - val_loss: 0.1850
Epoch 8/10
60000/60000 [==============================] - 53s 876us/step - loss: 0.1834 - val_loss: 0.1843
Epoch 9/10
60000/60000 [==============================] - 53s 875us/step - loss: 0.1828 - val_loss: 0.1833
Epoch 10/10
60000/60000 [==============================] - 54s 903us/step - loss: 0.1823 - val_loss: 0.1848
```

Trained result



Application of VAE

- **DNN Model pretrained weight**
  - 為讓模型找到一個較好的起始值。
- **Image segmentation**
  - 影像切割 **or face detection**
- **Video to Text**
  - image caption，想當然爾的就用到 sequence to sequence 模型拉，input data 是一堆的照片，output 則是描述照片的一段文字，在這邊的sequence to sequence 模型，我們會使用 LSTM + Conv net 當作 encoder & decoder
- **Image Retrieval**
- **Anomaly Detection**

Generative Adversarial Networks (GANs)

生成對抗網路

The Creativity of NNs

- Traditional NNs are supervised learning models
  - Learn knowledge from labeled data
  - The knowledge is based on the label
  - Do not have the creativity

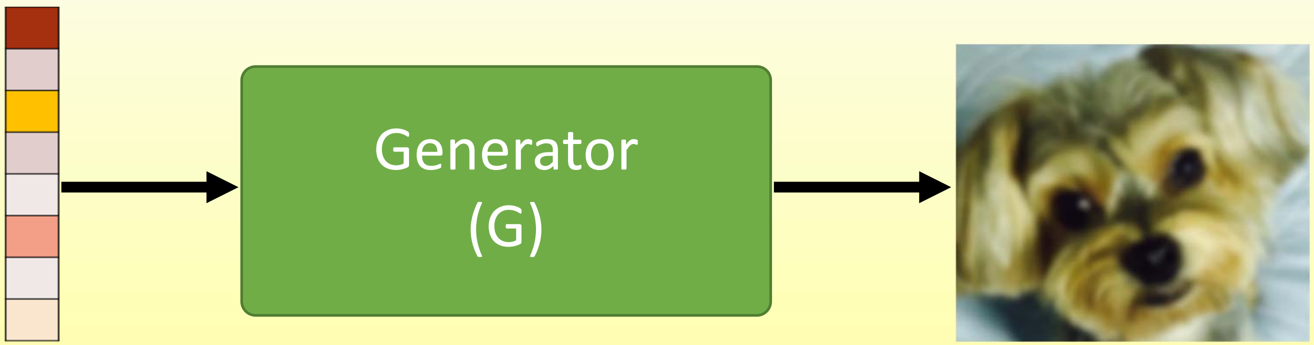
- How about **creating data** by machines themselves?
  - Auto-encoder
    - The decoder in auto-encoder

Generative Adversarial Networks (GAN)

- **GANs** (生成對抗網路), introduced in 2014 by Goodfellow et al., are an alternative to VAEs for learning latent spaces of images.
- They enable the generation of fairly realistic synthetic images by forcing the generated images to be statistically almost indistinguishable from real ones.
- GAN example
  - A **forger (偽造者)** trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos and shows them all to an art dealer.
  - The **art dealer (藝品經銷商)** makes an **authenticity** assessment for each painting and gives the forger feedback about what makes a Picasso look like a Picasso.
  - The forger goes back to his studio to prepare some new fakes. As times goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes.
  - In the end, they have on their hands some excellent fake Picassos.

The Generator in GANs

- ***Generator network*** **(生成器網路)**
  - Takes as input a **random vector** (a random point in the latent space), and decodes it into a synthetic image.
  - The generator network is trained to **be able to fool the discriminator network (鑑別器網路)**, and thus it evolves toward generating increasingly realistic images as training goes on: artificial images that look indistinguishable from real ones, to the extent that it’s impossible for the discriminator network to tell the two apart.

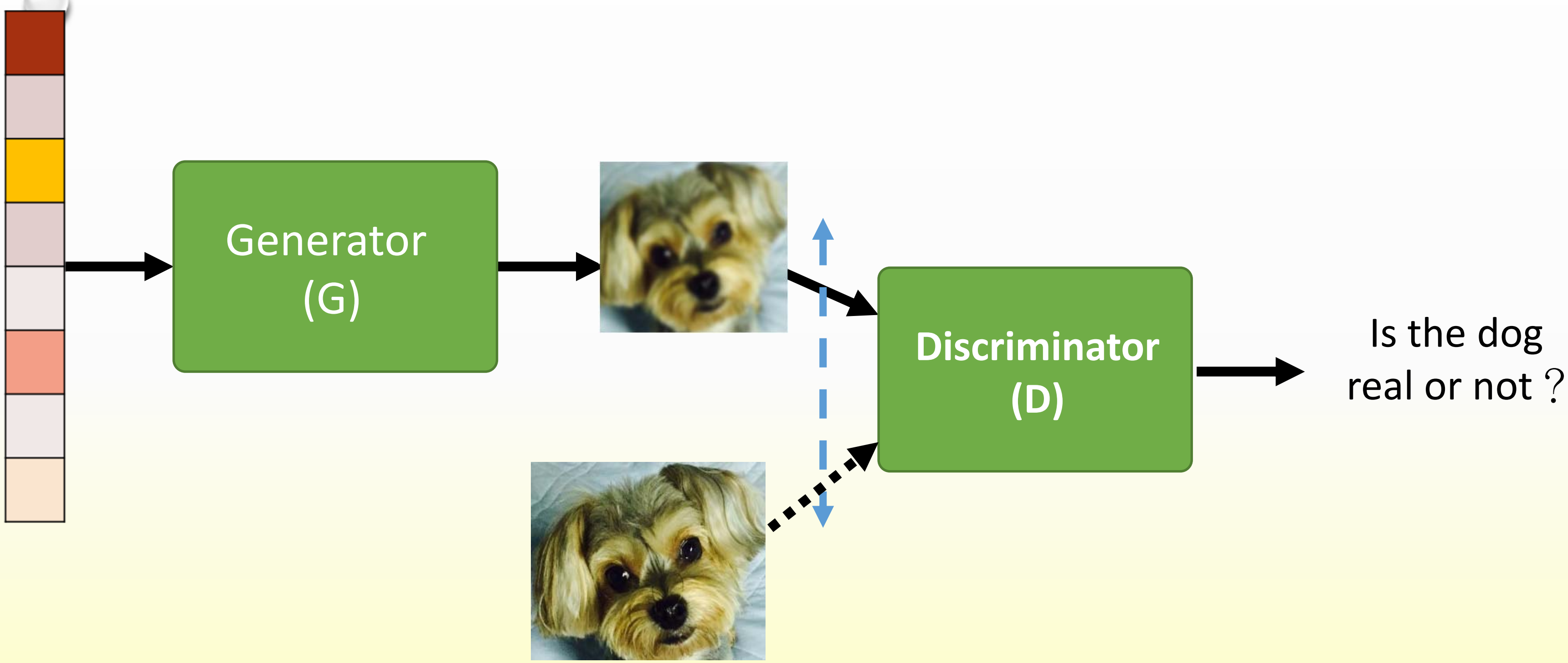


Representation

Is Generated Data Real Enough?

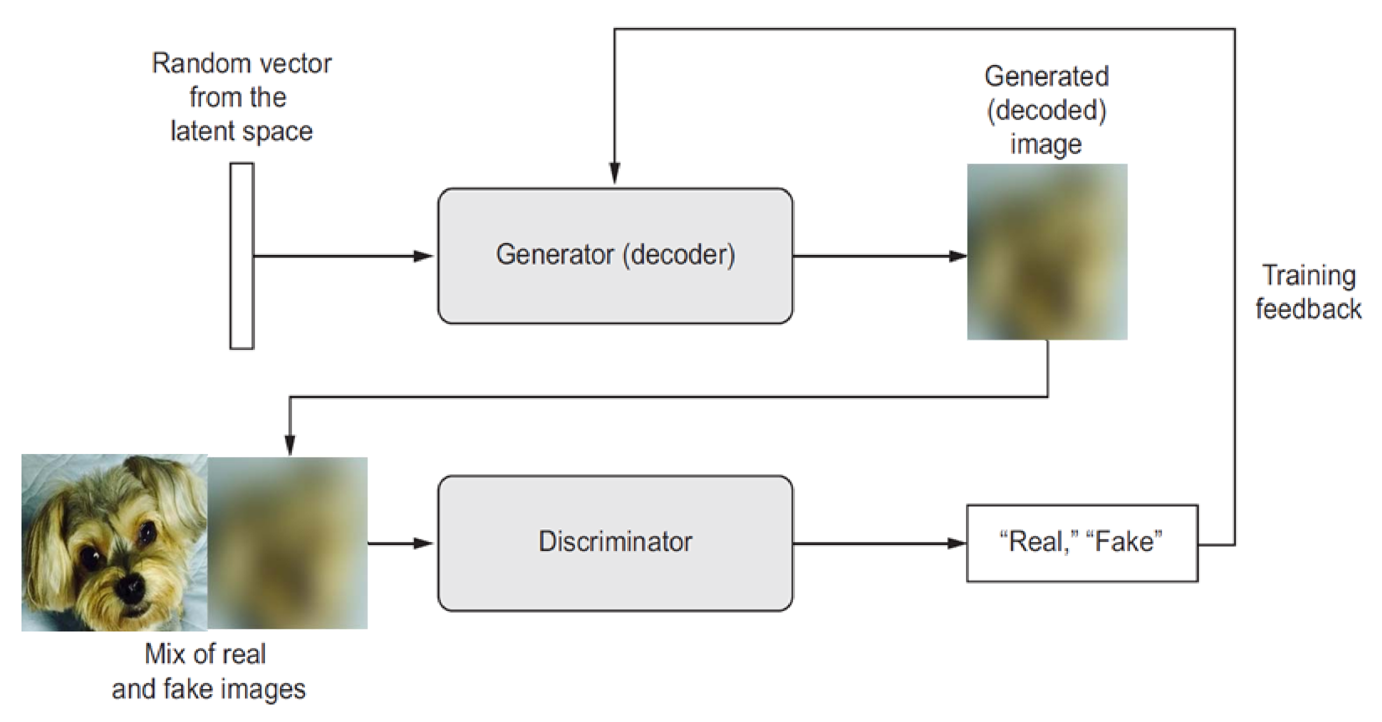
- **Discriminator network (or adversary)** 鑑別器網路
  - Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.
  - Discriminator is **constantly adapting** to the gradually improving capabilities of the generator, **setting a high** bar of realism for the generated images.

A Brief Overview of GANs



The Training Objective and flow chart

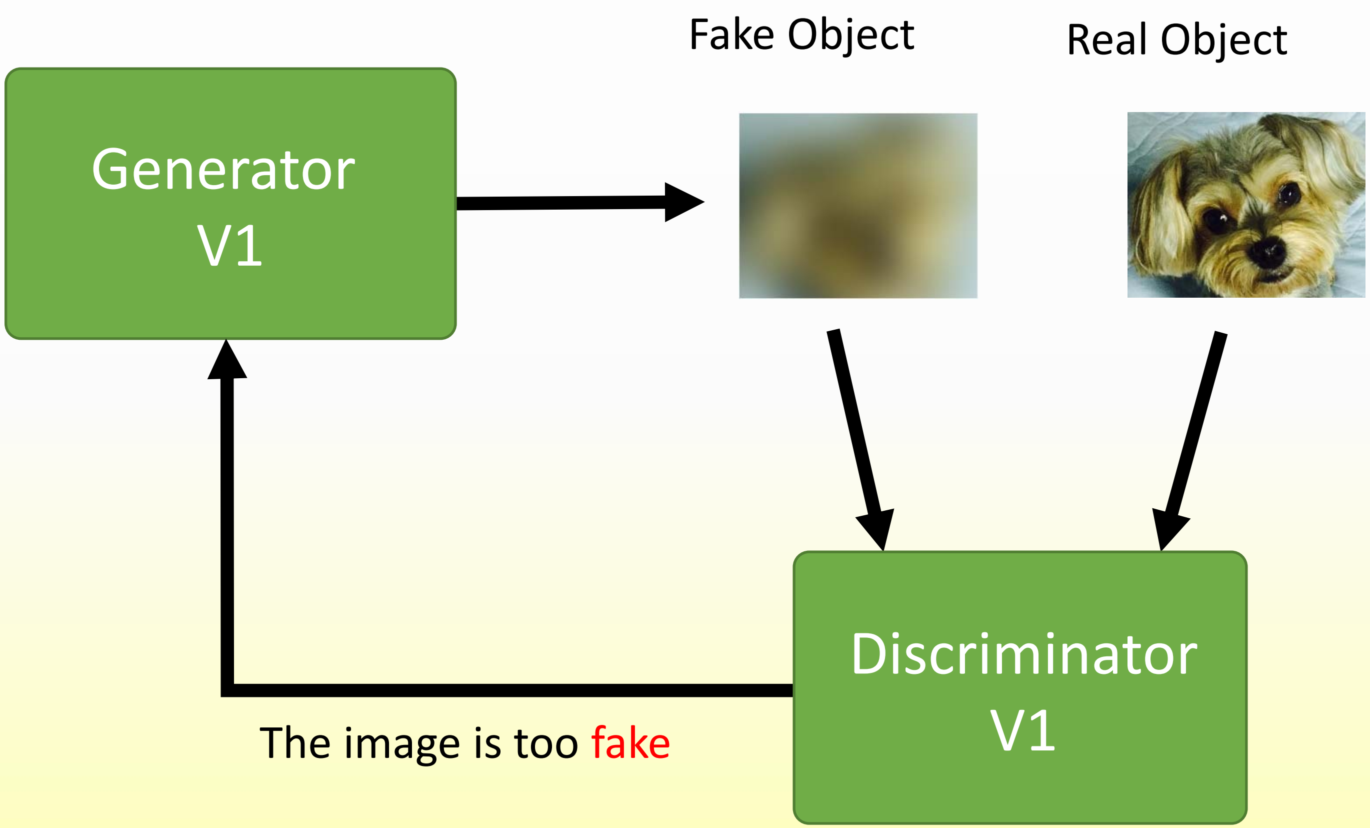
- Generator
  - Generate the object as similar as the real object
- Discriminator
  - Correctly distinguish the fake objects from true ones



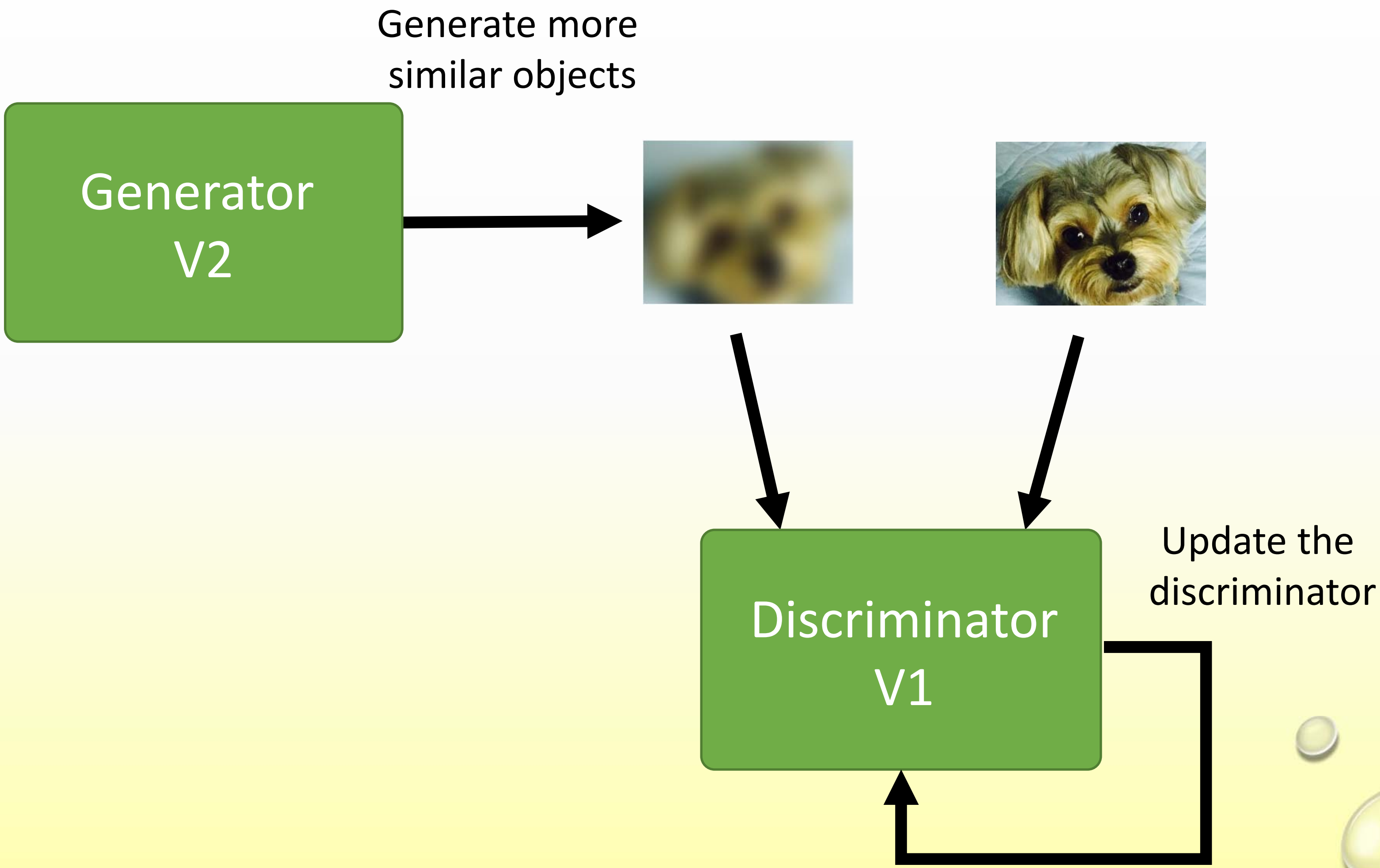
Training GAN

- Remarkably, a GAN is a system where the optimization minimum isn't fixed. Normally, gradient descent consists of rolling down hills in a static loss landscape.
- But with a GAN, every step taken down the hill changes the entire landscape a little.
- It's a dynamic system where the optimization process is seeking **not a minimum**, but an equilibrium (平衡) between two forces.
- For this reason, GANs are **notoriously difficult to train** —getting a GAN to work requires lots of careful tuning of the model architecture and training parameters.

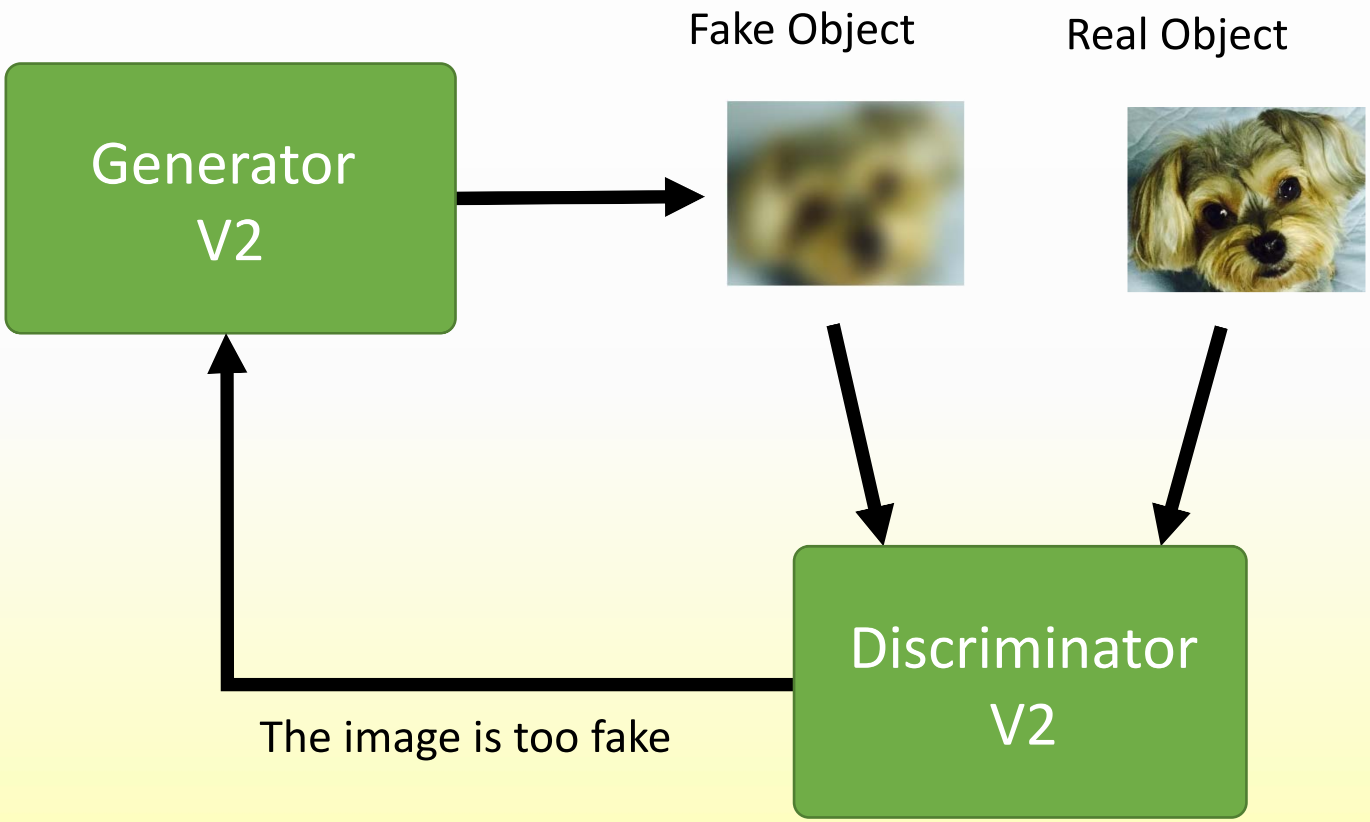
The Training Process of GANs



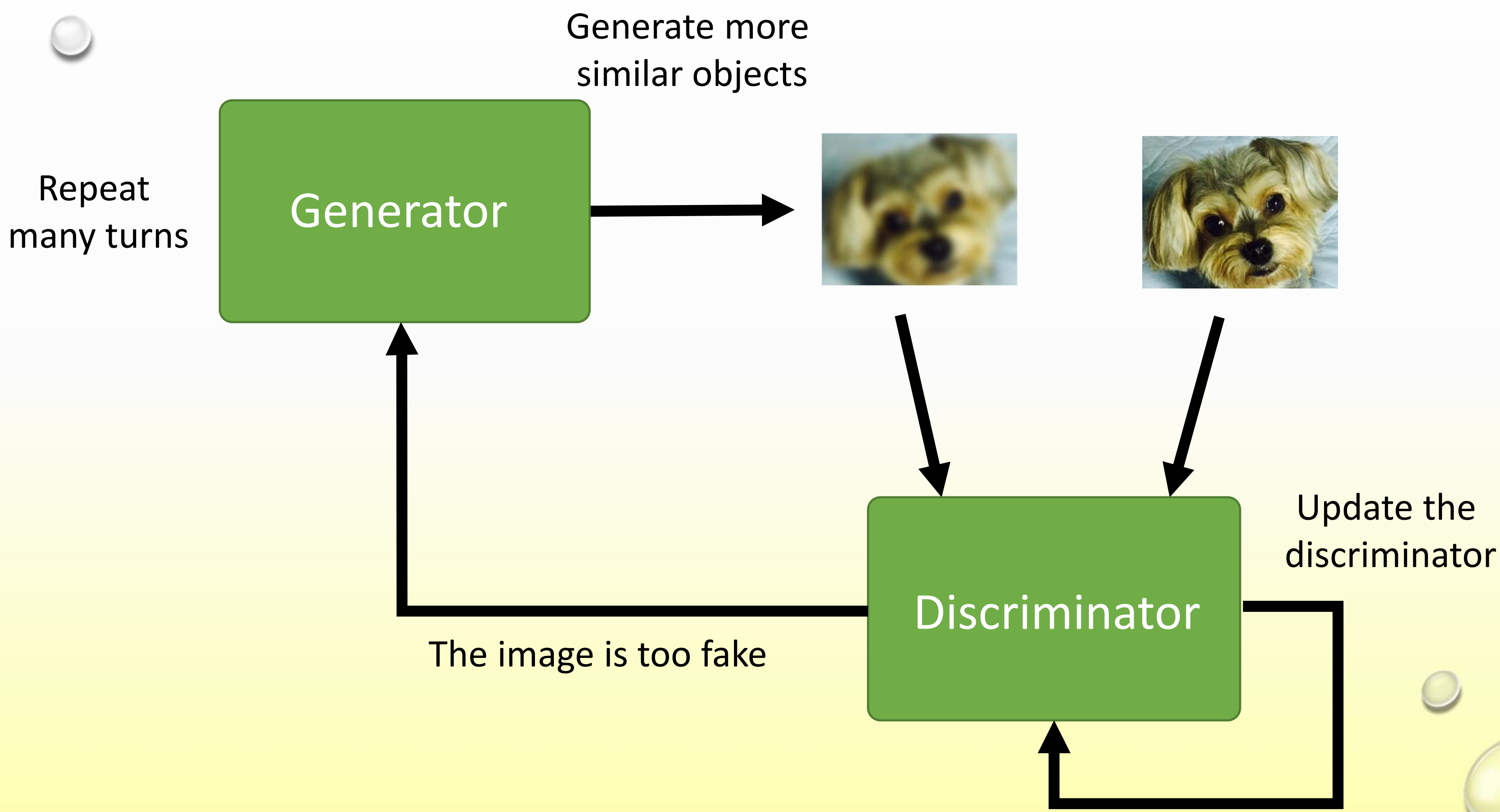
The Training Process of GANs



The Training Process of GANs



The Training Process of GANs



How to Train GANs

- For every iteration
  - **We train the discriminator first**
    - By a given generator
    - Train **many rounds** to create a powerful discriminator
  - Then we train the generator
    - By a given discriminator
    - Only **one iteration** to prevent overfitting

Training On Discriminators

- The discriminator is a **binary classifier**
  - Classify if the input is **real or not**

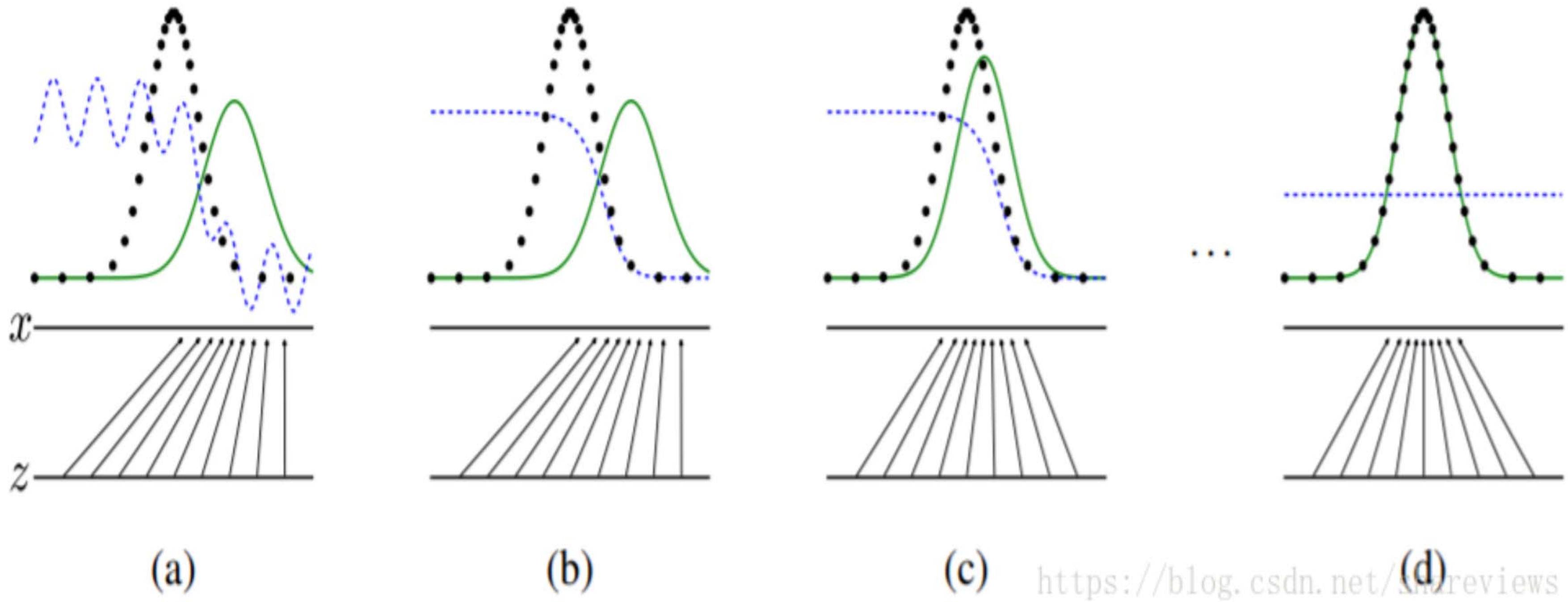
- Use cross-entropy as the error of the discriminator
  - **Back-propagation** to train discriminator
  - Lower cross-entropy is better

Training On Generators

- The goal of the generator is to generate the data similar to the real data
  - Let the discriminator be unable to classify the fake objects well

- Using the **cross-entropy** of the discriminator as the error
  - Back-propagation to train the generators
  - **Higher** cross-entropy is better

GANs和很多其他模型不同，GANs在訓練時需要同時執行兩個優化演算法，我們需要為discriminator和generator分別定義一個優化器，一個用來來最小化discriminator的損失，另一個用來最小化generator的損失。即loss = d_loss + g_loss



黑色虛線是真實資料的高斯分佈，綠色的線是生成網路學習到的偽造分佈，藍色的線是判別網路判定為真實圖片的概率，標x的橫線代表服從高斯分佈x的取樣空間，標z的橫線代表服從均勻分佈z的取樣空間。從上圖中可以看出，經過多次迭代，可以看出生成模型(Generator)學習了從z的空間到x的空間的對映關係。簡單來說就是生成模型(Generator)和原始資料集的特徵近似相同，訓練工作就結束了，生成模型(Generator)生成的資料已經假假真真不可辨識了。

A schematic GAN implementation

- The specific implementation is a ***deep convolutional GAN* (DCGAN):** a GAN where the generator and discriminator are **deep convnets**. In particular, it uses a **Conv2DTranspose** layer for image upsampling in the generator.
- You'll train the GAN on images from CIFAR10, a dataset of **50,000** 32×32 RGB images belonging to **10** classes (5,000 images per class).
- To make things easier, you'll only use images belonging to the class "**frog**."

GAN scheme/generator

- A **generator network** maps vectors of **shape** (latent_dim,) to images of shape (32, 32, 3).

```
import keras
from keras import layers
import numpy as np

latent_dim = 32
height = 32
width = 32
channels = 3

generator_input = keras.Input(shape=(latent_dim,))  # 建立輸入張量, shape = (?, 32)

# 將輸入轉換成 16×16 128 層次 (channel) 的張量
x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)
print(x.shape) # (?, 16, 16, 128)

# 加入卷積層
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
print(x.shape) # (?, 16, 16, 256)
```

GAN scheme/Generator

```
# 向上取樣成 32x32
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)
print(x.shape)  # 雖然從 shape 看不出來,
                # 但從待會的 model.summary() 可以看出 shape=(None, 32, 32, 256)

# 使用更多的卷積
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)


x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x) # 產生 32x32 3 層次的特徵圖 (CIFAR10圖像的形狀)
generator = keras.models.Model(generator_input, x)   # 實例化生成器模型，將 shape=(latent_dim, )的輸入對應成 shape=(32,32,3) 的圖像
generator.summary()
```

Generator summary()

```
(None, 16, 16, 128)
(None, 16, 16, 256)
(None, None, None, 256)
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 32)                0
_________________________________________________________________
dense_9 (Dense)              (None, 32768)             1081344
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32768)             0
_________________________________________________________________
reshape_2 (Reshape)          (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_104 (Conv2D)          (None, 16, 16, 256)       819456
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 32, 32, 256)       1048832
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_105 (Conv2D)          (None, 32, 32, 256)       1638656
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_106 (Conv2D)          (None, 32, 32, 256)       1638656
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_107 (Conv2D)          (None, 32, 32, 3)         37635
=================================================================
Total params: 6,264,579
Trainable params: 6,264,579
Non-trainable params: 0
```

A bag of tricks

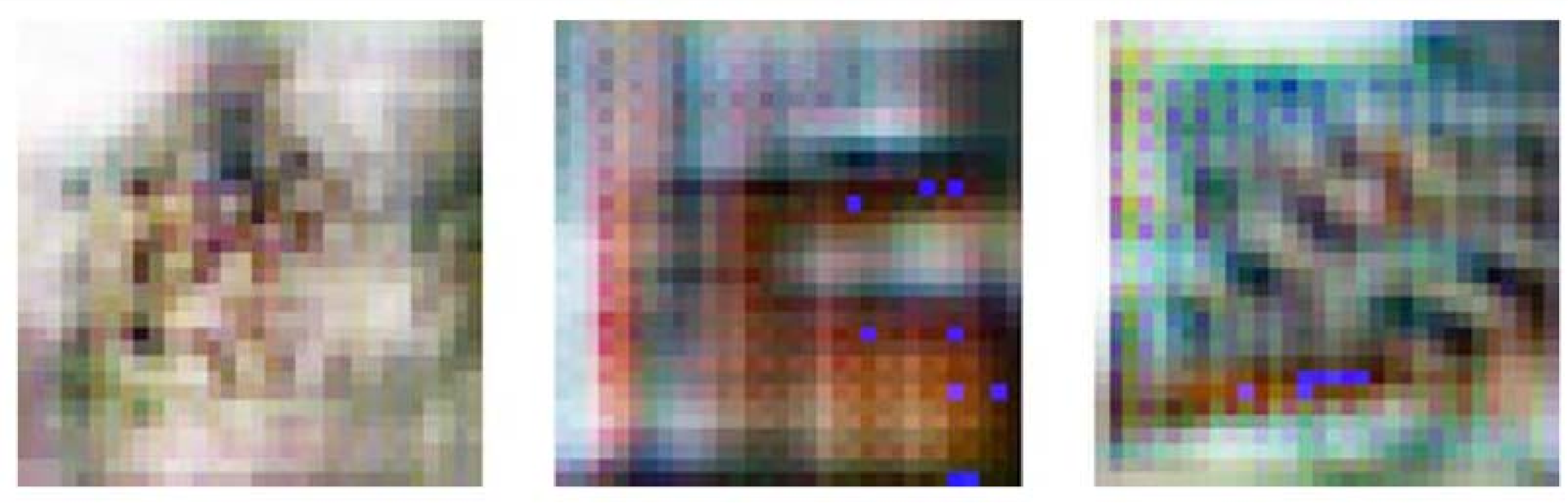
- We use *tanh* as the last activation in the generator, instead of sigmoid, which is more commonly found in other types of models.
- We sample points from the latent space using a ***normal distribution*** (**Gaussian distribution**), not a uniform distribution.
- Stochasticity is good to induce robustness. Because GAN training results in a dynamic equilibrium, GANs are likely to get stuck in all sorts of ways. Introducing randomness during training helps prevent this.
- We introduce randomness in two ways:
  - by using **dropout** in the discriminator and
  - by adding **random noise** to the labels for the discriminator.

A bag of tricks

- **Sparse gradients** can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs.
- Two things can induce gradient sparsity:
  - **max pooling operations** and **ReLU activations**.
- Instead of max pooling, we recommend using **strided** convolutions for downsampling, and we recommend using a **LeakyReLU** layer instead of a **ReLU** activation.
- It's similar to ReLU, but it relaxes sparsity constraints by allowing small negative activation values.

A bag of tricks

- In generated images, it's common to see checkerboard artifacts caused by unequal coverage of the pixel space in the generator. To fix this, we use a **kernel size** that's divisible by the stride size whenever we use a strided **Conv2DTranpose** or **Conv2D** in both the generator and the discriminator.



GAN/ discriminator

- A **discriminator network** maps images of shape **(32, 32, 3)** to a **binary score** estimating the probability that the image is real.

```
# 建立圖片尺寸的輸入張量, 其 shape=(?, 32, 32, 3)
discriminator_input = layers.Input(shape=(height, width, channels))

x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
print(x.shape) # 經過一連串的卷積 shape=(?, 2, 2, 12
                                                              # 學習速率
x = layers.Flatten()(x) # 拉平, 其 shape=(?, 512)             # 在優化過程中使用梯度遞減(依設定值)
                                                              # 為了穩定訓練過程，使用學習速率衰減
                  # 隨機丟棄 40 % 的神經元
x = layers.Dropout(0.4)(x)  # 一個重要的技巧：一個丟

x = layers.Dense(1, activation='sigmoid')(x)     # 分類層
print(x.shape) # 最終輸出 shape=(?, 1)

# 實例化鑑別器模型，其將 (32,32,3) 輸入圖片轉換為二元分類決策 (假/真)
discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()

discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008,      #
                                                   clipvalue=1.0,  #
                                                   decay=1e-8)     #
discriminator.compile(optimizer=discriminator_optimizer,
                      loss='binary_crossentropy')
```

Discriminator summary()

```
(None, 2, 2, 128)
(None, 1)
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_108 (Conv2D)          (None, 30, 30, 128)       3584
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 30, 30, 128)       0
_________________________________________________________________
conv2d_109 (Conv2D)          (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_110 (Conv2D)          (None, 6, 6, 128)         262272
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 6, 6, 128)         0
_________________________________________________________________
conv2d_111 (Conv2D)          (None, 2, 2, 128)         262272
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 2, 2, 128)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 513
=================================================================
Total params: 790,913
Trainable params: 790,913
Non-trainable params: 0
_________________________________________________________________
```

The adversarial network

- Finally, you'll set up the GAN, which chains the generator and the discriminator.
- When trained, this model will move the generator in a direction that improves its ability to fool the discriminator. This model turns latent-space points into a classification decision—"fake" or "real"—and it's meant to be trained with labels that are always "these are real images."
- So, training gan will update the weights of generator in a way that makes discriminator more likely to predict "real" when looking at fake images.
- It's very important to note that you set the discriminator to be frozen during training **(non-trainable):** its weights won't be updated when training gan.
- If the discriminator weights could be updated during this process, then you'd be training the discriminator to always predict "real," which isn't what you want!

GAN Design

- A **gan** network chains the generator and the discriminator together: **gan(x) = discriminator(generator(x)).**
- Thus this gan network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.

```
discriminator.trainable = False # 將鑑別器權重設定為不可訓練 (僅適用於ga

gan_input = keras.Input(shape=(latent_dim,))        # 建立 GAN 的輸入
gan_output = discriminator(generator(gan_input))    # 將輸入張量送入生
gan = keras.models.Model(gan_input, gan_output)     # 實例化 GAN 模型
gan.summary()

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004,
                                         clipvalue=1.0,
                                         decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

GAN summary()

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         (None, 32)                0
_________________________________________________________________
model_3 (Model)              (None, 32, 32, 3)         6264579
_________________________________________________________________
model_4 (Model)              (None, 1)                 790913
=================================================================
Total params: 7,055,492
Trainable params: 6,264,579
Non-trainable params: 790,913
_________________________________________________________________
```

GAN Train

- You train the **discriminator** using examples of real and fake images along with “real”/“fake” labels, just as you train any regular **image-classification model**.
- To train the **generator,** you use the gradients of the generator’s weights with regard to the loss of the gan model.
  - This means, at every step, **you move the weights of the generator in a direction that makes the discriminator more likely to classify as “real” the images decoded by the generator**.
  - In other words, you train the generator to fool the discriminator.

Code example

```
import os
from keras.preprocessing import image

# 載入 CIFAR10 資料
(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data()

# 選擇青蛙圖像 (類別6)
x_train = x_train[y_train.flatten() == 6]

# 標準化 (正規化) 資料
x_train = x_train.reshape(
    (x_train.shape[0],) + (height, width, channels)).astype('float32') / 255.

iterations = 10000
batch_size = 20
# 指定要儲存生成圖像的位置
save_dir = 'gan_images'
start = 0
```

```python
for step in range(iterations):  # 進行 10000 步
    # 在潛在空間中取樣隨機的點
    random_latent_vectors = np.random.normal(size=(batch_size,
                                                    latent_dim))

    # 產生成假圖像
    generated_images = generator.predict(random_latent_vectors)

    # 將假圖片與真實圖像相混合
    stop = start + batch_size
    real_images = x_train[start: stop]
    combined_images = np.concatenate([generated_images, real_images])

    # 分配標籤，以從假圖像中辨別真的
    labels = np.concatenate([np.ones((batch_size, 1)),
                             np.zeros((batch_size, 1))])
    # 在標籤中增加隨機雜訊，這是一個重要技巧！
    labels += 0.05 * np.random.random(labels.shape)

    d_loss = discriminator.train_on_batch(combined_images, # 訓練鑑別器
                                          labels)
```

```python
# 在潛在空間中取樣隨機的點
random_latent_vectors = np.random.normal(size=(batch_size,
                                               latent_dim))

# 分配標籤說 "這些都是真實圖像" (這是謊言!)
misleading_targets = np.zeros((batch_size, 1))

# 訓練生成器 (透過 gan 模型，其中鑑別器權重被凍結)
a_loss = gan.train_on_batch(random_latent_vectors,
                            misleading_targets)

start += batch_size
if start > len(x_train) - batch_size:
  start = 0

if step % 100 == 0:          # 每 100 步儲存並和繪製結果
    gan.save_weights('gan.h5')  # 儲存模型權重

    # 印出衡量指標
    print('discriminator loss at step %s: %s' % (step, d_loss))
    print('adversarial loss at step %s: %s' % (step, a_loss))

    # 儲存一個生成的圖像
    img = image.array_to_img(generated_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir, 'generated_frog' + str(step) + '.png'))

    # 儲存一個真實圖像以進行比較
    img = image.array_to_img(real_images[0] * 255., scale=False)
    img.save(os.path.join(save_dir, 'real_frog' + str(step) + '.png'))
```

How to train your DCGAN

- Draw random points in the latent space (random noise).
- Generate images with generator using this random noise.
- Mix the generated images with real ones.
- Train discriminator using these mixed images, with corresponding targets: either “real” (for the real images) or “fake” (for the generated images).
- Draw new random points in the latent space.
- Train gan using these random vectors, with targets that all say “these are real images.” This updates the weights of the generator (only, because the discriminator is frozen inside gan) to move them toward getting the discriminator to predict “these are real images” for generated images: this trains the generator to fool the discriminator.

Console 1/A

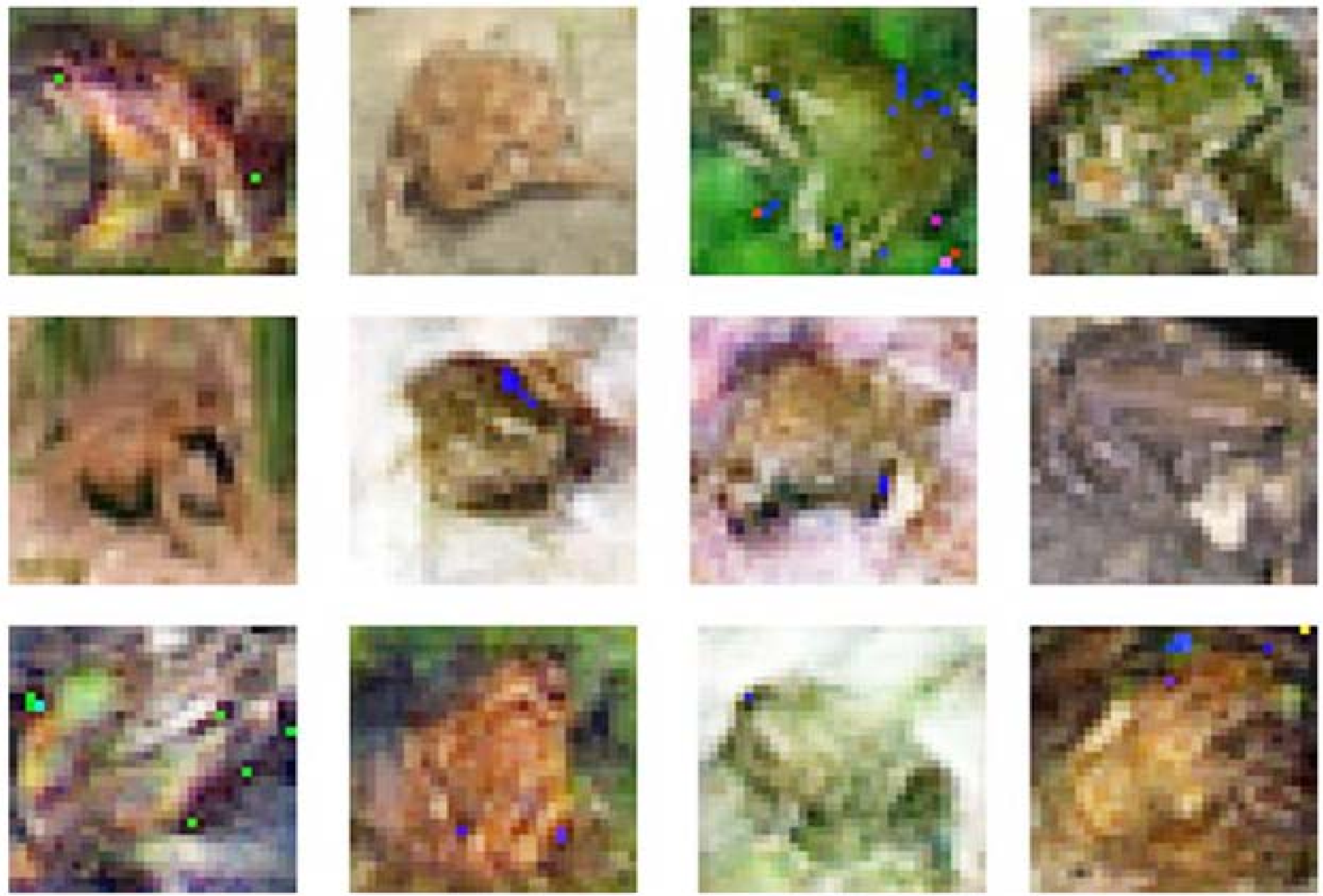
```
C:\ProgramData\Anaconda3\lib\site-packages\ker
weights and collected trainable weights, did y
  'Discrepancy between trainable weights and c
discriminator loss at step 100: 0.75582993
adversarial loss at step 100: 0.9504241
discriminator loss at step 200: 0.68841046
adversarial loss at step 200: 0.8984643
discriminator loss at step 300: 0.69808036
adversarial loss at step 300: 0.77540207
discriminator loss at step 400: 0.7073922
adversarial loss at step 400: 0.74175984
discriminator loss at step 500: 0.67492497
adversarial loss at step 500: 0.74553394
discriminator loss at step 600: 0.69969815
adversarial loss at step 600: 0.7383934
discriminator loss at step 700: 0.6963583
adversarial loss at step 700: 0.7510997
discriminator loss at step 800: 0.80721235
adversarial loss at step 800: 0.92534447
discriminator loss at step 900: 0.68276036
adversarial loss at step 900: 1.0949743
discriminator loss at step 1000: 0.69614303
adversarial loss at step 1000: 0.74313784
```

```
discriminator loss at step 1100: 0.70450234
adversarial loss at step 1100: 0.7281759
discriminator loss at step 1200: 0.6860279
adversarial loss at step 1200: 0.7312754
discriminator loss at step 1300: 0.83236706
adversarial loss at step 1300: 0.7858817
discriminator loss at step 1400: 0.7036462
adversarial loss at step 1400: 0.776518
discriminator loss at step 1500: 0.69471437
adversarial loss at step 1500: 0.7433877
discriminator loss at step 1600: 0.6979415
adversarial loss at step 1600: 0.7757813
discriminator loss at step 1700: 0.6832381
adversarial loss at step 1700: 0.7464247
discriminator loss at step 1800: 0.71137893
adversarial loss at step 1800: 0.76255256
discriminator loss at step 1900: 0.7179365
adversarial loss at step 1900: 0.9277687
discriminator loss at step 2000: 0.6733109
adversarial loss at step 2000: 2.926025
```

Console 1/A

```
adversarial loss at step 8800: 1.1112775
discriminator loss at step 8900: 0.667642
adversarial loss at step 8900: 0.5758585
discriminator loss at step 9000: 0.67467844
adversarial loss at step 9000: 0.8679792
discriminator loss at step 9100: 0.66247123
adversarial loss at step 9100: 0.77195966
discriminator loss at step 9200: 0.6461853
adversarial loss at step 9200: 0.78622264
discriminator loss at step 9300: 0.70442784
adversarial loss at step 9300: 2.220649
discriminator loss at step 9400: 0.6587831
adversarial loss at step 9400: 0.7392591
discriminator loss at step 9500: 0.6787297
adversarial loss at step 9500: 0.7856676
discriminator loss at step 9600: 0.66123104
adversarial loss at step 9600: 0.7972778
discriminator loss at step 9700: 0.6988957
adversarial loss at step 9700: 0.8783563
discriminator loss at step 9800: 0.668928
adversarial loss at step 9800: 0.79789555
discriminator loss at step 9900: 0.69468796
adversarial loss at step 9900: 0.58311665
```

Result

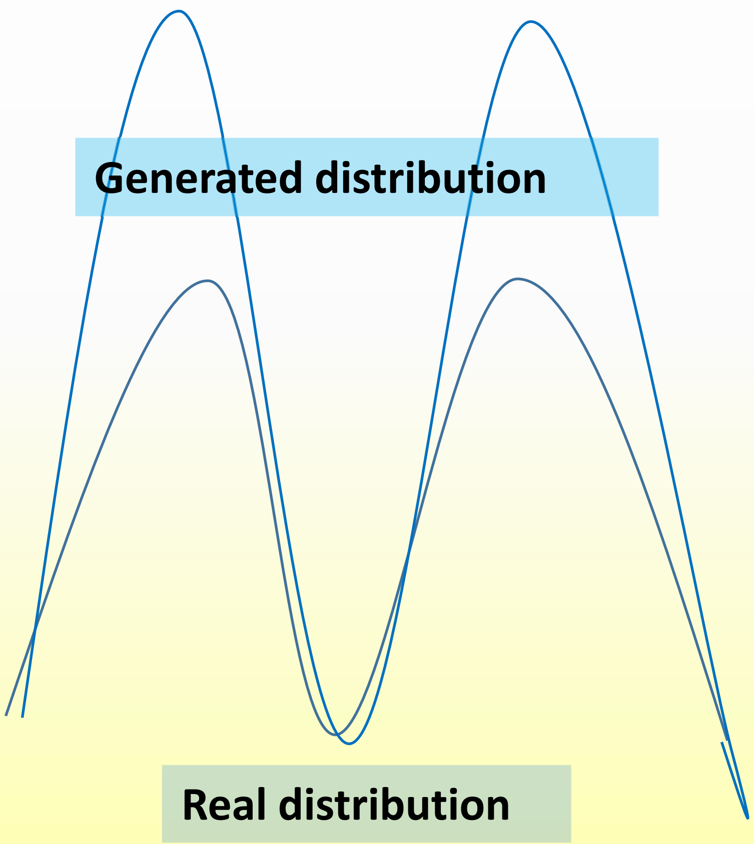


The Problems in GANs

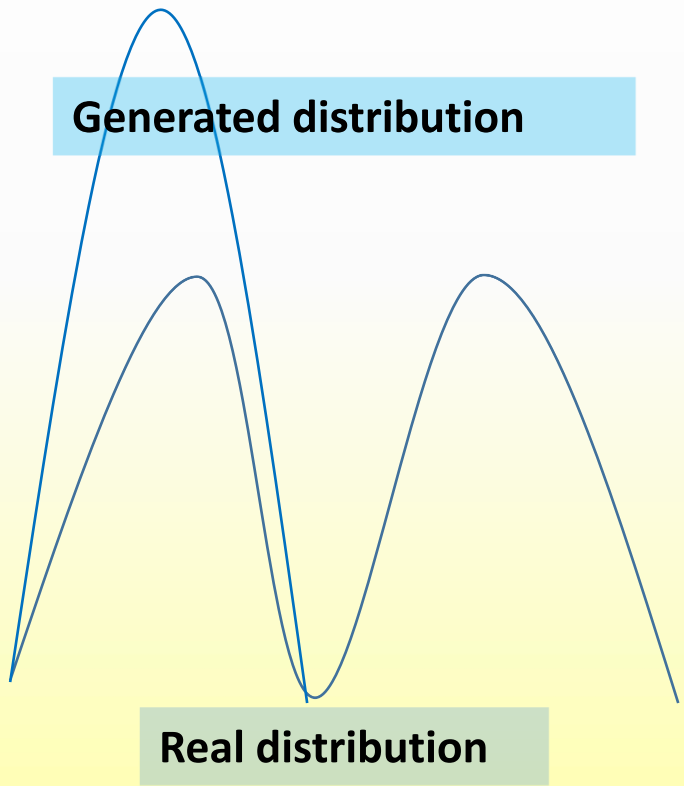
- The balance between generator and discriminator
- Mode collapse
- Gradient vanishing

Mode Collapse

We hope that



In real case

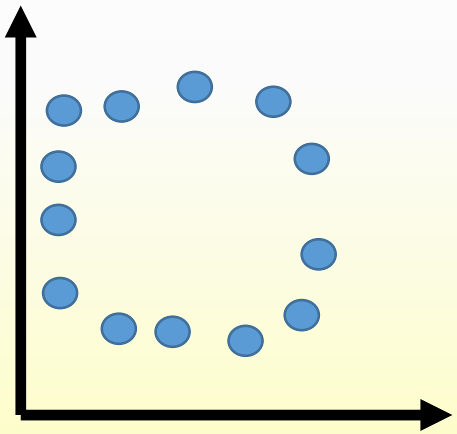


Mode Collapse

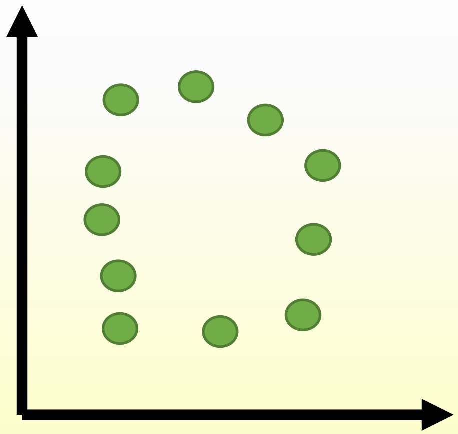
- The generator do not generate different kinds of objects
  - Because it will lead KL or JS divergence larger, which means the loss will be larger
- The generator tend to generate similar objects with the same distribution
  - Which can lead to the smallest loss
- The generated object become very similar
  - Lose the diversity of the generated objects

Another Example for Mode Collapse

Real distribution



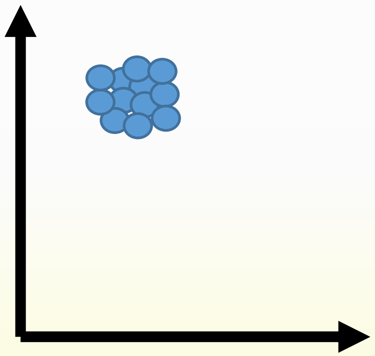
Ideal generated distribution



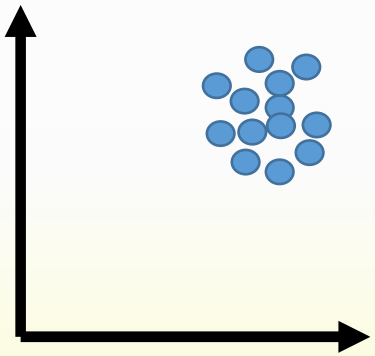
Another Example for Mode Collapse

Actually……

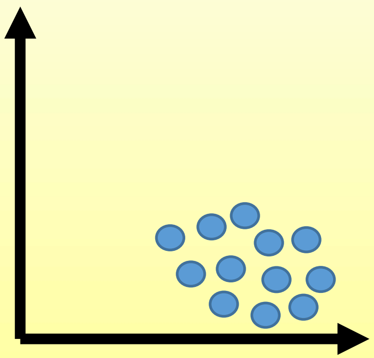
100th iteration



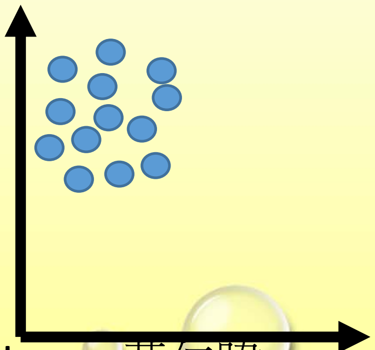
500th iteration

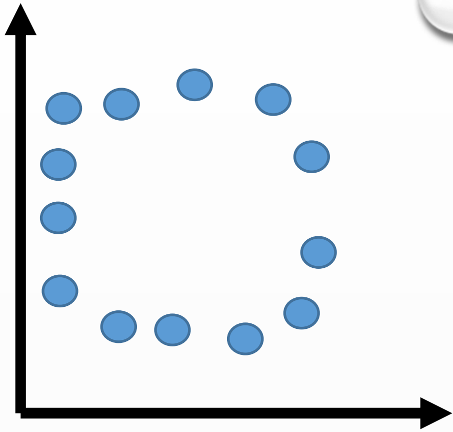


1000th iteration



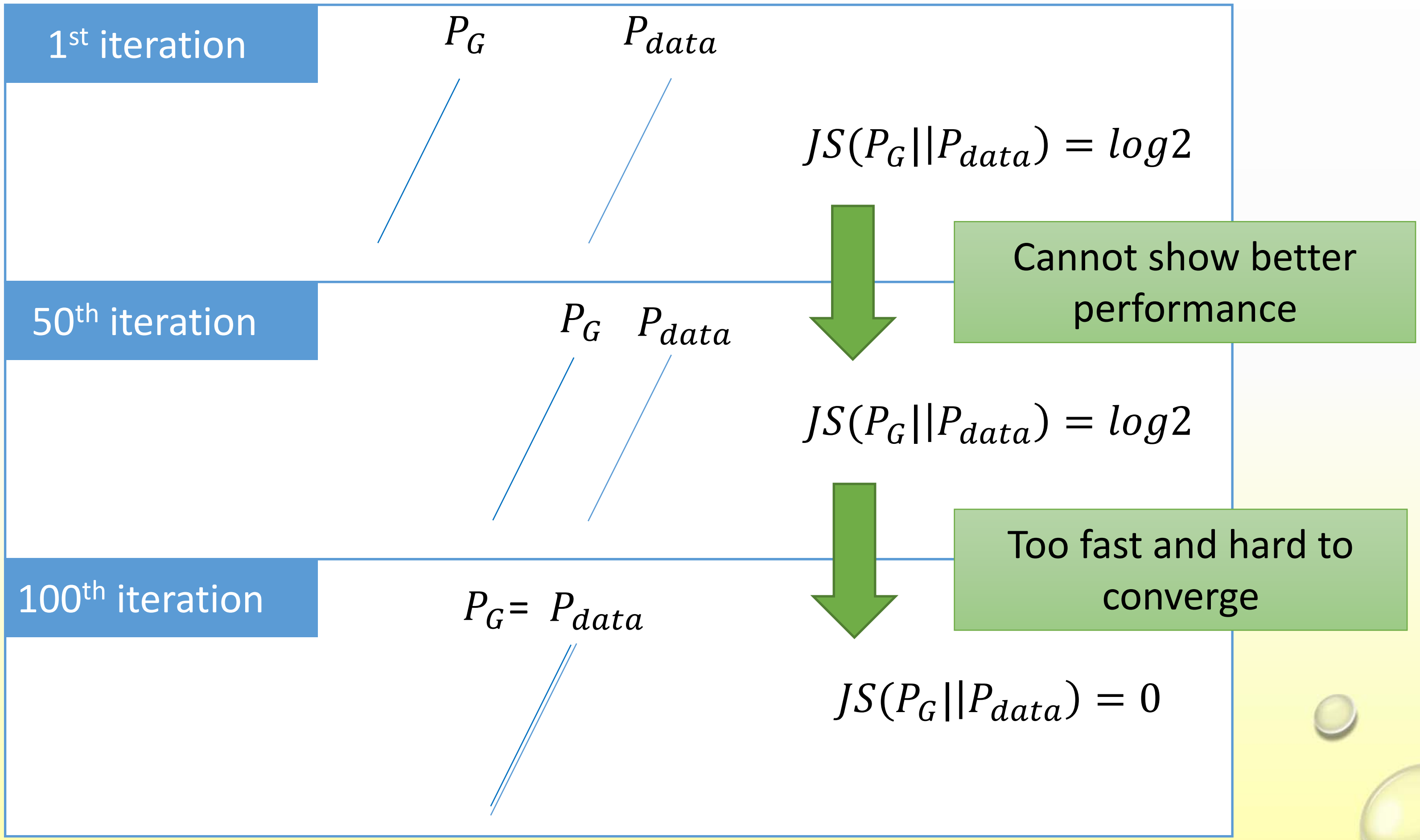
1500th iteration





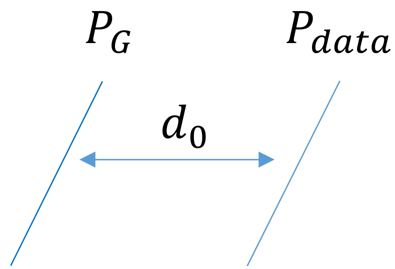
An infinite loop……
Cannot converge

Gradient Vanishing



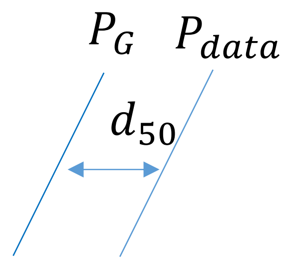
How about Other Distance Measurement?

1st iteration



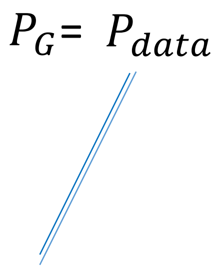
$$W(P_G||P_{data}) = d_0$$

50th iteration



$$W(P_G||P_{data}) = d_{50} < d_0$$

100th iteration

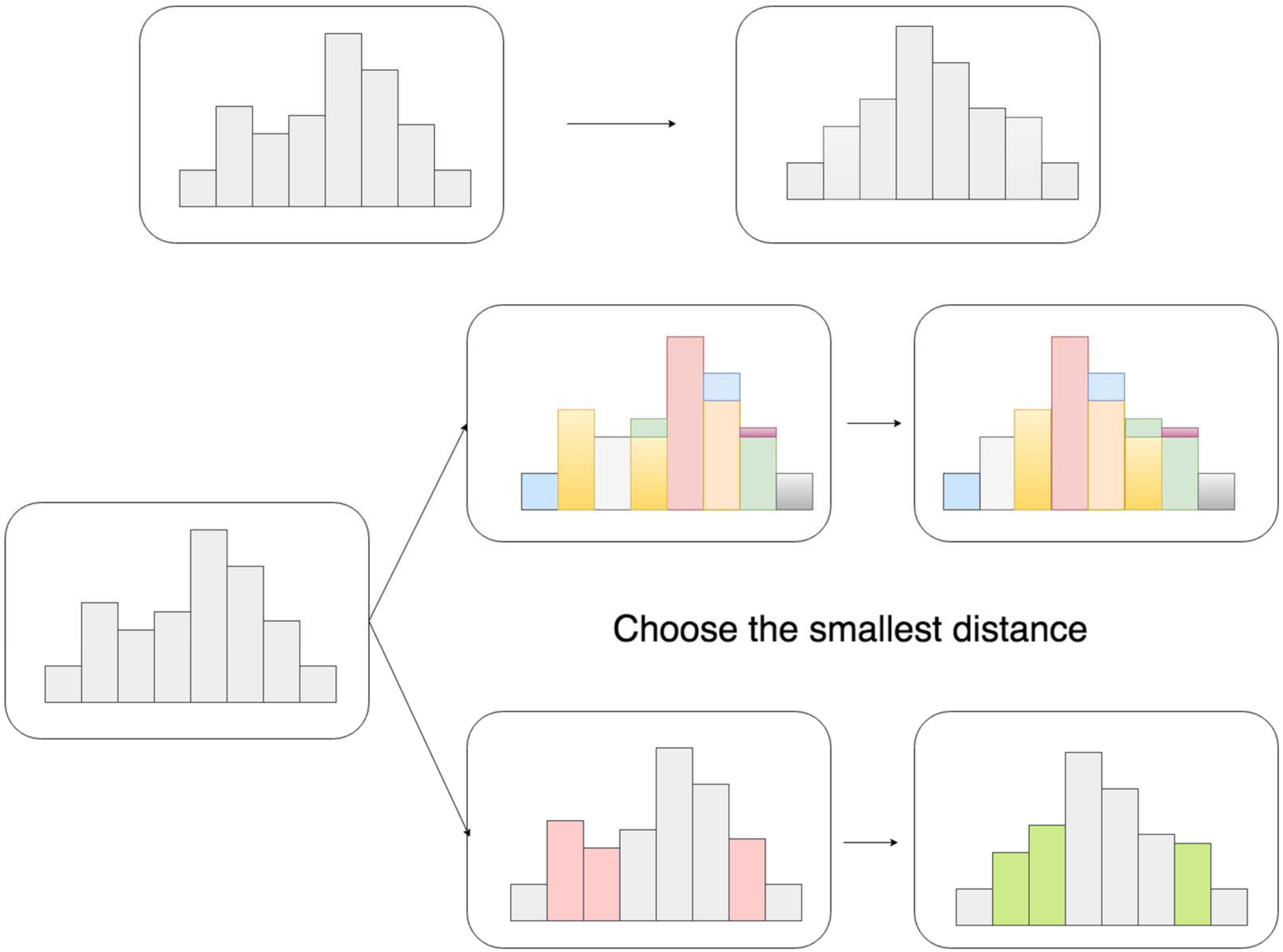


$$W(P_G||P_{data}) = 0$$

Wasserstein Distance

- Also known as Earth Mover Distance
- The distance from a distribution to another distribution
- More simplify
  - Two distributions are places
  - There is a pile of earth on the first distribution
  - We want to move the earth to the second distribution
  - The average distance is Earth Mover Distance

Earth Mover Distance



Wasserstein Generative Adversarial Network (WGAN)

- Conquer the problems of GAN
  - The balance on generator and discriminator
  - Mode collapse
  - Gradient vanishing

The Modifications in WGAN

- Using Wasserstein distance instead of KL divergence or JS divergence

- Removing the sigmoid activation in the output layer
- Using real error instead log error
- Weight clipping
  - If $w < -c$, let $w = -c$
  - If $w > c$, let $w = c$
- Use SGD instead of momentum based optimization

How to Design a Good GAN

The Notions and Suggestions

- The performance of generator and discriminator
  - The discriminator should not be very strong
- The diversity of inputs and outputs
  - Especially the diversity of outputs
  - To prevent mode collapse
- Consider to use WGAN instead of GANs
  - Prevent the problems in GANs