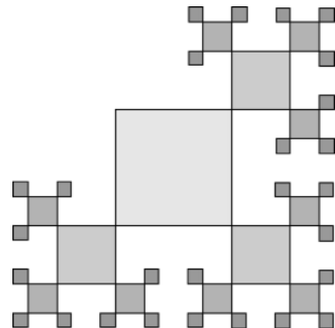
INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

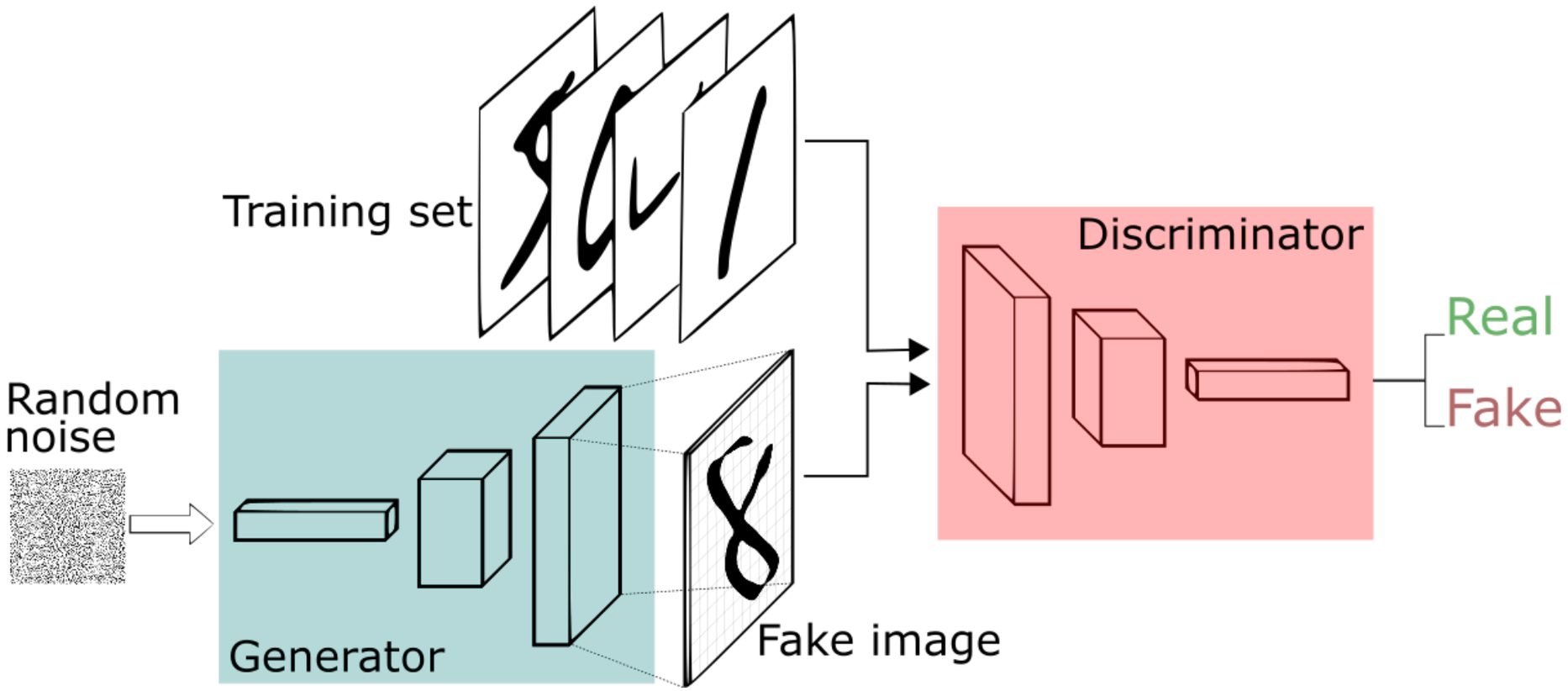
GENERATIVE ADVERSARIAL NETWORK: INTRODUCTION

- The name "GAN", or the **Generative Adversarial Network**, was first introduced by Ian Goodfellow in 2014. It is a very interesting idea and became extremely popular in recent years.
- As we already slightly mentioned earlier, the key setup is to have two networks training against each other:
  - **Discriminative network** — trained to distinguish the data produced by the generator from the true data.
  - **Generative network** — trained to map from a latent space to a data distribution of interest; objective is to increase the error rate (*to fool*) of the discriminator.
- GAN is a kind of **unsupervised learning**, e.g. no needs of labeling data by human beings!



GENERATIVE ADVERSARIAL NETWORK: INTRODUCTION (II)

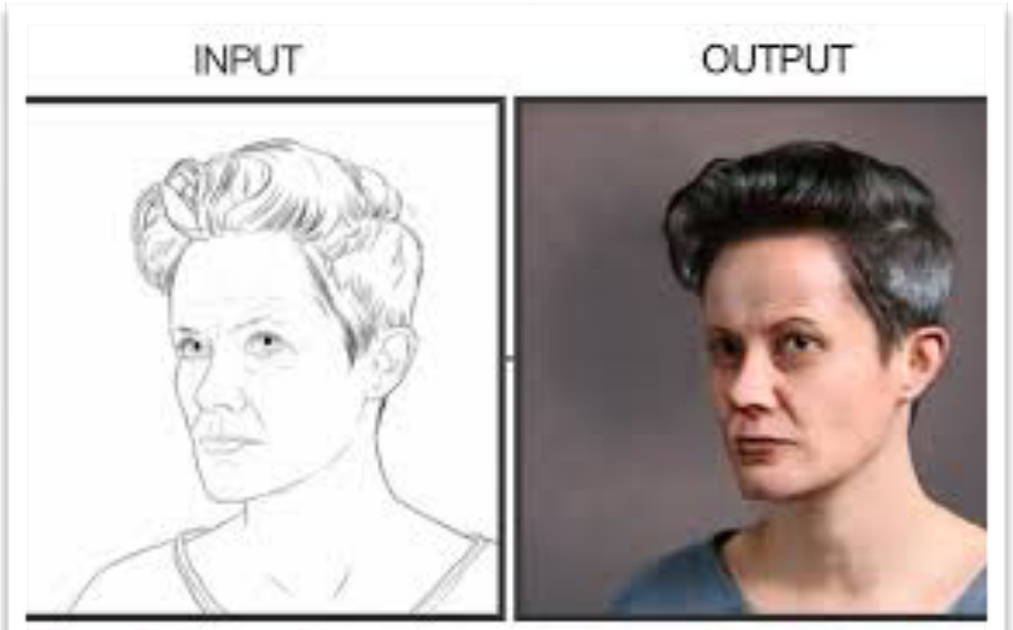
- The typical GAN network structure is arranged as following. The generator and discriminator can be classical MLP or convolutional network or any other variations.
- If one replace the input noise with some other stuff (e.g. a doodle, etc), it can be used to convert/modify images!



Many fancy stuffs you heard recently may all related to this type of network!

Face generation

Style transfer



Convert doodle to photo

Image upscaling

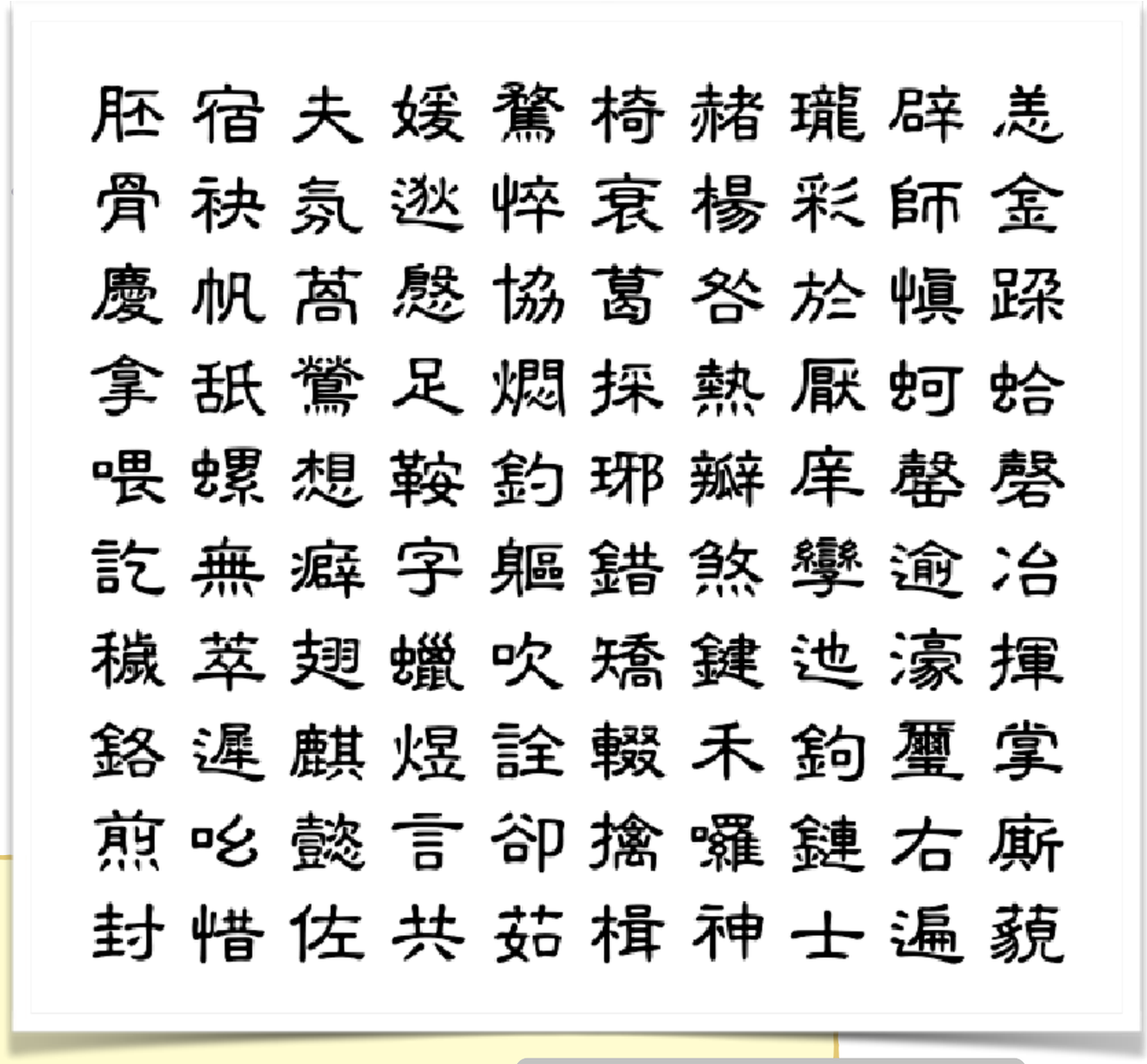
IMAGE GENERATION WITH GAN

- Let's practice image generation with a very simple GAN setup. All we need to do is to prepare a collection of images, train the generator and discriminator, and use the generator to produce some fake images.
- One can simply collect some nice photos, drawings, or whatever data to do such a practice in fact!
- In the following example, we are going to ask GAN to generate some **Chinese characters** which does not exist so far!

How about an AI Cangjie?

FONT DATA

- The given font_data.npy file stores the images (48×48) of commonly used 4808 characters, defined by MOE!
- Randomly pick up 100 characters and show!



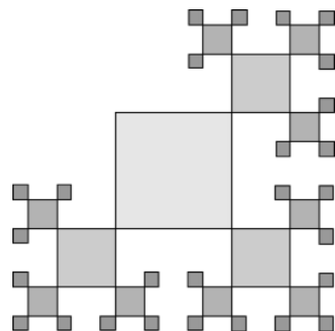
```
import numpy as np
import matplotlib.pyplot as plt

data = np.load('font_data.npy')

fig = plt.figure(figsize=(10,10), dpi=80)
plt.subplots_adjust(0.05,0.05,0.95,0.95,0.1,0.1)
for i in range(100):
    plt.subplot(10,10,i+1)
    plt.axis('off')
    plt.imshow(data[np.random.randint(4808)], cmap='Greys')
plt.show()
```

ex_ml10_1.py

Yes, these are **clerical scripts**!



CONSTRUCT A VANILLA GAN

- Construct a classical neural network as the **discriminator**, with input = image / output = binary classifier:

partial ex_ml10_2.py

```
x_train = np.load('font_data.npy')
x_train = x_train/127.5-1.
```

← loading images and scale to ±1

```
latent_size = 128
img_shape = (48,48)

from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam

discriminator = Sequential()
discriminator.add(Reshape((np.prod(img_shape),),input_shape=img_shape))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))
discriminator.add(Dense(1, activation='sigmoid'))
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(0.0002, 0.5),
                      metrics=['accuracy'])
```

Discriminator Model:
image ⇒ 512 ⇒ 256 ⇒ 1 node

← sigmoid + binary x-entropy

CONSTRUCT A VANILLA GAN (II)

- **Generator** can be constructed also with a classical network, input = latent array (noise) / output = image:

partial ex_ml10_2.py

```
generator = Sequential()
generator.add(Dense(256, input_dim=latent_size))
generator.add(LeakyReLU())
generator.add(BatchNormalization())
generator.add(Dense(512))
generator.add(LeakyReLU())
generator.add(BatchNormalization())
generator.add(Dense(1024))
generator.add(LeakyReLU())
generator.add(BatchNormalization())
generator.add(Dense(np.prod(img_shape), activation='tanh'))
generator.add(Reshape(img_shape))
```

Generator Model:
noise ⇒ 256 ⇒ 512 ⇒ 1024 ⇒ image

recommended to use tanh
↓ for generator output

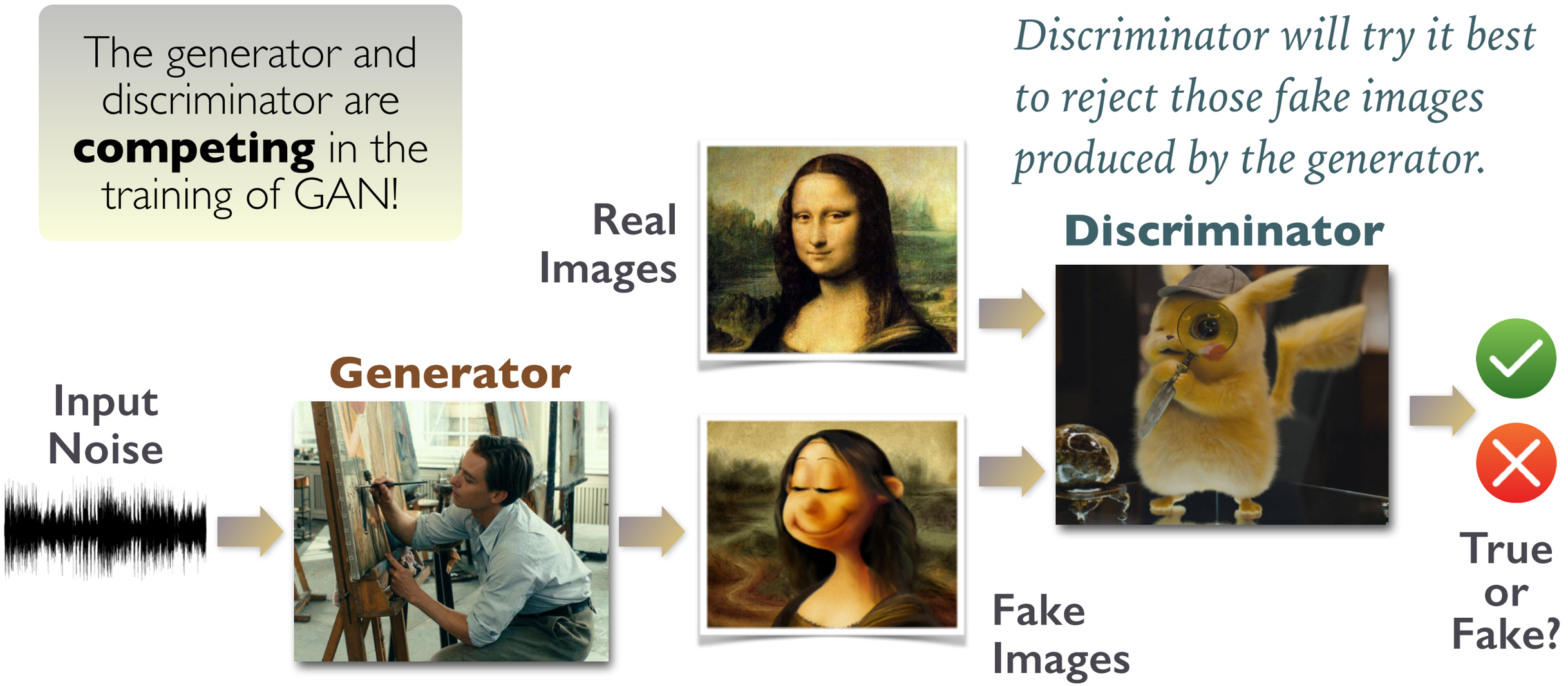
```
noise = Input(shape=(latent_size,))
img = generator(noise)
discriminator.trainable = False
validity = discriminator(img)
combined = Model(noise, validity)
combined.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```

← combined model: noise input,
binary classifier output
(disable training for discriminator part)

GENERATOR VS DISCRIMINATOR

The generator and discriminator are **competing** in the training of GAN!

Discriminator will try it best to reject those fake images produced by the generator.



Generator will try it best to convert the input noise to be the nearly true fake images. Training objective = confuse the discriminator as much as possible.

GAN TRAINING

- Training steps in manual : ask the discriminator to separate real/fake images; ask the generator to cheat the discriminator.

partial ex_ml10_2.py

```
. . . . . .
noise_test = np.random.randn(100, latent_size)
batch_size = 64
for epoch in range(20001):
```

real images from input data;
fake images from generator

```
    imgs_real = x_train[np.random.randint(0, len(x_train), batch_size)]

    noise = np.random.randn(batch_size, latent_size)
    imgs_fake = generator.predict(noise)
```

Training the discriminator with real & fake images

```
    dis_loss_real = discriminator.train_on_batch(imgs_real, np.ones((batch_size,1)))
    dis_loss_fake = discriminator.train_on_batch(imgs_fake, np.zeros((batch_size,1)))
    dis_loss = np.add(dis_loss_real,dis_loss_fake)*0.5
```

training generator

```
    noise = np.random.randn(batch_size, latent_size)
    gen_loss = combined.train_on_batch(noise, np.ones((batch_size,1)))

    print("Epoch: %d, disc(loss: %.3f, acc.: %.2f%%), gen(loss: %.3f)" % \
          (epoch, dis_loss[0], dis_loss[1]*100., gen_loss))
```

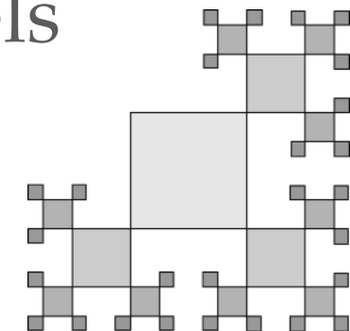
RESULTS OF TRAINING

- It does generate some images which may "look like" Chinese characters (*although one has to read them from a long distance*).
- **Surely none of them is really readable!**


```
Epoch: 0, discriminator(loss: 0.881, acc.: 44.53%), generator(loss: 0.818)
. . . . . .
Epoch: 100, discriminator(loss: 0.022, acc.: 100.00%), generator(loss: 5.088)
. . . . . .
Epoch: 1000, discriminator(loss: 0.625, acc.: 68.75%), generator(loss: 0.985)
. . . . . .
Epoch: 10000, discriminator(loss: 0.555, acc.: 74.22%), generator(loss: 1.262)
. . . . . .
Epoch: 20000, discriminator(loss: 0.465, acc.: 77.34%), generator(loss: 1.422)
```

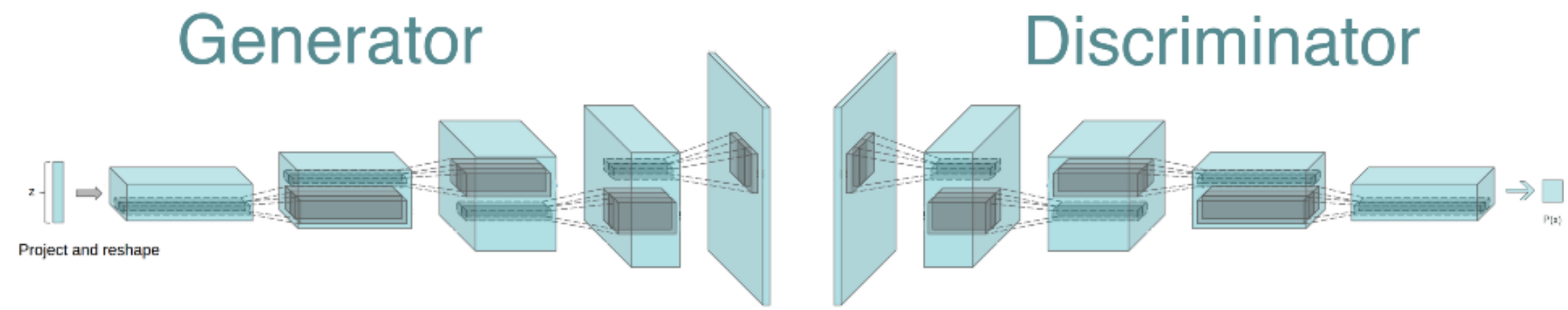
COMMENTS/TIPS

- You may note the training of discriminator will not end up with a high accuracy (nearly 100% or so).
- This is actually good for GAN — if the accuracy is high for discriminator, it means the generator is ill-trained, and have no way to "cheat" the discriminator.
- Ideally **the discriminator and generator should "grow" together** (*gradually improve both network as epochs*). If one of the models becoming too powerful (in most of the cases the discriminator is much more effective), the GAN training may fail and end up with noisy images.
- So the trick is to keep the balance between the two models during the training process.



GAN+CNN = DCGAN

- We do understand the convolutional network can be outperforming for image processing problems.
- If one replace the discriminator with a convolutional network, and use a "deconvolution" network for the generator, it might be more powerful than a vanilla GAN?
- This is the basic idea of **Deep Convolutional GAN**, or **DCGAN**. It adds convolutional layers for scaling up/down, and without max pooling and fully connected layers.



CONSTRUCT A DCGAN

- Have to replace the **discriminator** with CNN:

partial ex_ml10_3.py

```
. . . . . .
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam

discriminator = Sequential()
discriminator.add(Reshape(img_shape+(1,), input_shape=img_shape))
discriminator.add(Conv2D(64, kernel_size=5, strides=2, padding='same'))
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))
discriminator.add(Conv2D(128, kernel_size=5, strides=2, padding='same'))
discriminator.add(LeakyReLU())
discriminator.add(Dropout(0.3))
discriminator.add(Flatten())
discriminator.add(Dense(1, activation='sigmoid'))
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(0.0002, 0.5),
                      metrics=['accuracy'])
```

Discriminator Model:
image ⇒ (conv)x2 ⇒ 1 binary node

CONSTRUCT A DCGAN (II)

- The **Generator** has to be replaced as well:

partial ex_ml10_3.py

```
generator = Sequential()
generator.add(Dense(12*12*256, input_dim=latent_size))
generator.add(BatchNormalization())
generator.add(LeakyReLU())
generator.add(Reshape((12,12,256)))
generator.add(Conv2DTranspose(128, kernel_size=5, strides=1, padding='same'))
generator.add(BatchNormalization())
generator.add(LeakyReLU())
generator.add(Conv2DTranspose(64, kernel_size=5, strides=2, padding='same'))
generator.add(BatchNormalization())
generator.add(LeakyReLU())
generator.add(Conv2DTranspose(1, kernel_size=5, strides=2, padding='same', \
              activation='tanh'))
generator.add(Reshape(img_shape))

noise = Input(shape=(latent_size,))
img = generator(noise)
discriminator.trainable = False
validity = discriminator(img)
combined = Model(noise, validity)
combined.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```

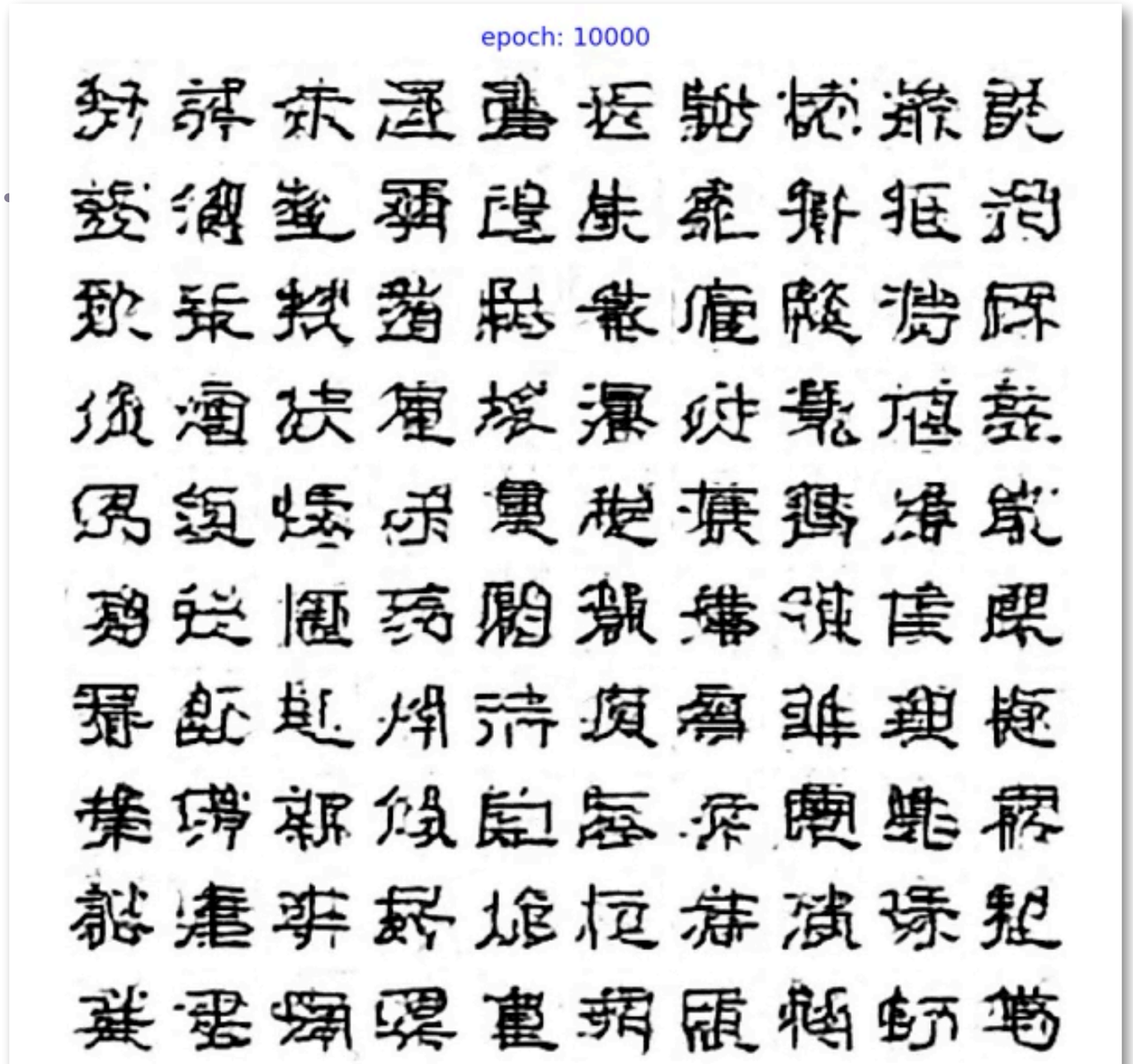
Generator Model:
noise ⇒ (Conv2D)x3 ⇒ image

← combined model is the same

all other parts are the same as the previous example!

RESULTS OF TRAINING (II)

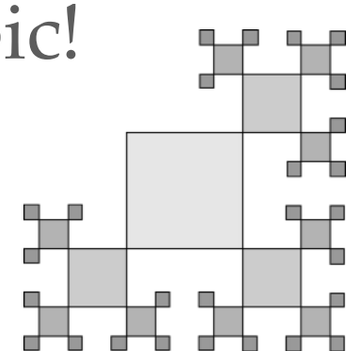
- Using DCGAN seems to have "smoother" fonts comparing to the previous vanilla GAN.
- **As expected none of them is really readable, *still*!**




```
Epoch: 0, discriminator(loss: 0.680, acc.: 33.59%), generator(loss: 0.642)
. . . . . .
Epoch: 100, discriminator(loss: 0.000, acc.: 100.00%), generator(loss: 0.000)
. . . . . .
Epoch: 1000, discriminator(loss: 0.022, acc.: 100.00%), generator(loss: 0.037)
. . . . . .
Epoch: 5000, discriminator(loss: 0.392, acc.: 82.81%), generator(loss: 2.061)
. . . . . .
Epoch: 10000, discriminator(loss: 0.427, acc.: 81.25%), generator(loss: 2.092)
```

COMMENT (AGAIN)

- There are far more interesting applications constructed based on the idea of GAN, as we already introduced some of the typical (famous) use cases earlier.
- Many of them do have example implementations. The following git directory contains many example code based on Keras: https://github.com/eriklindernoren/Keras-GAN
- If you are not satisfied with this, you may want to check the the **GAN Zoo** (*well, there might be too many!*): https://github.com/hindupuravinash/the-gan-zoo
- You may be able to think of a smart way of using such a network structure to resolve the problems of your own research topic!



JUST TRY IT OUT!

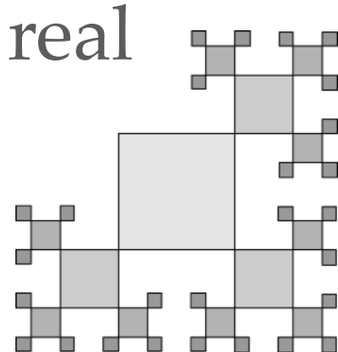
- GAN is one of the most interesting ideas that have been developed during past years, you should really give it a try by inserting a different set of images and see how good you can reach! (*note: machine learning is a super fast developing field, there are far too many papers produced nowadays!*).
- One thing you can already play is to mix two different font styles, e.g. clerical and semi-cursive scripts (also provided on the web)? And see what you can get, although we do not expect to generate any readable fonts…

行書＋隸書＝?

Let's discuss a little bit regarding the interplay between ML and (Particle) Physics!

FINAL COMMENT: PHYSICIST'S ML

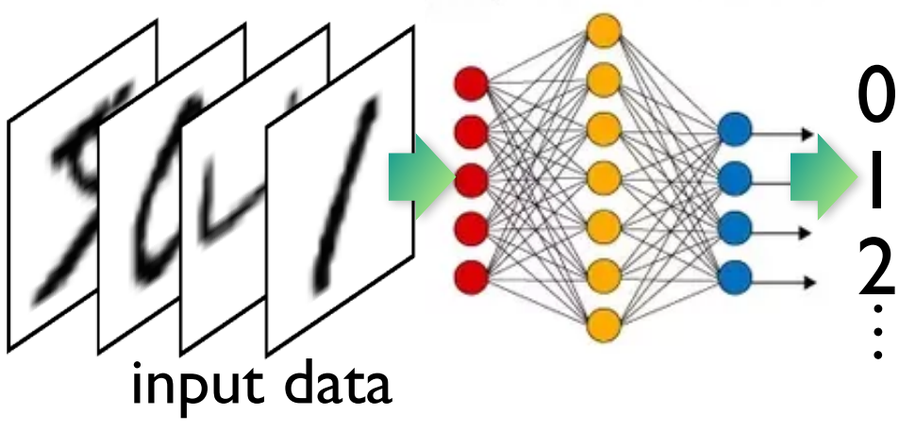
- Physicists also use a lot of ML to solve the problems found in the experiments or theories. But what are the core difference between a physicists' problem and a generic problem?
- Surely I cannot comment for everyone — but at least I can say the *particle physicists* have a rather different prospective regarding ML tools comparing to generic users.
- The key point of particle physicists' ML is about its **statistical interpretation**: we do not just concern about if your ML tool is working or not, we also worry about ***how correct it performs***. e.g. even if you know the accuracy of your network is **99.5%**, we also want to know the error of this value, e.g. **99.5±0.XX%**, and also the performance difference between the ideal situation and and real application.



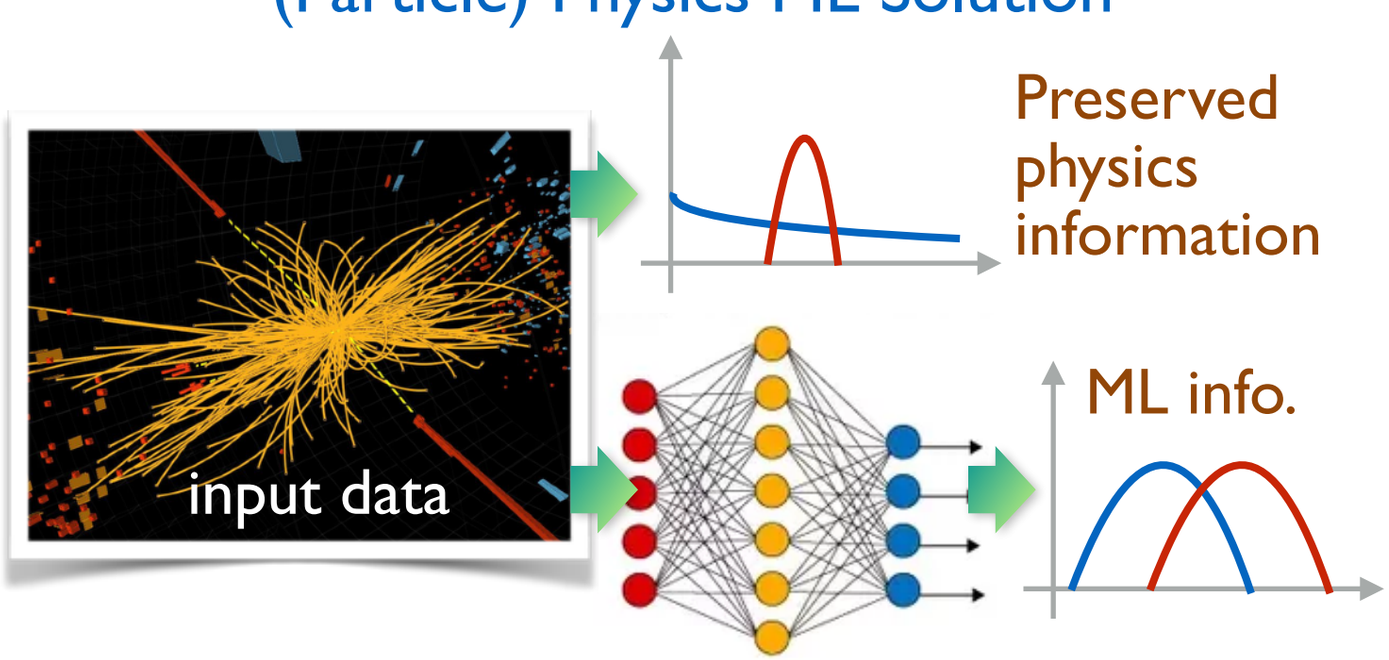
FINAL COMMENT: PHYSICIST'S ML (II)

- So unlike the generic problem (*e.g. image recognition, etc.*), we need to find a way to **preserve the information** and still use it to present physics results, instead of just dump everything into the network. i.e.

Generic ML Solution



(Particle) Physics ML Solution



So the (particle) physics ML solution is generally weaker than the generic ML due to lack of key information in ML. But we use it to do further **statistical analysis** afterwards.

PIX2PIX EXAMPLE

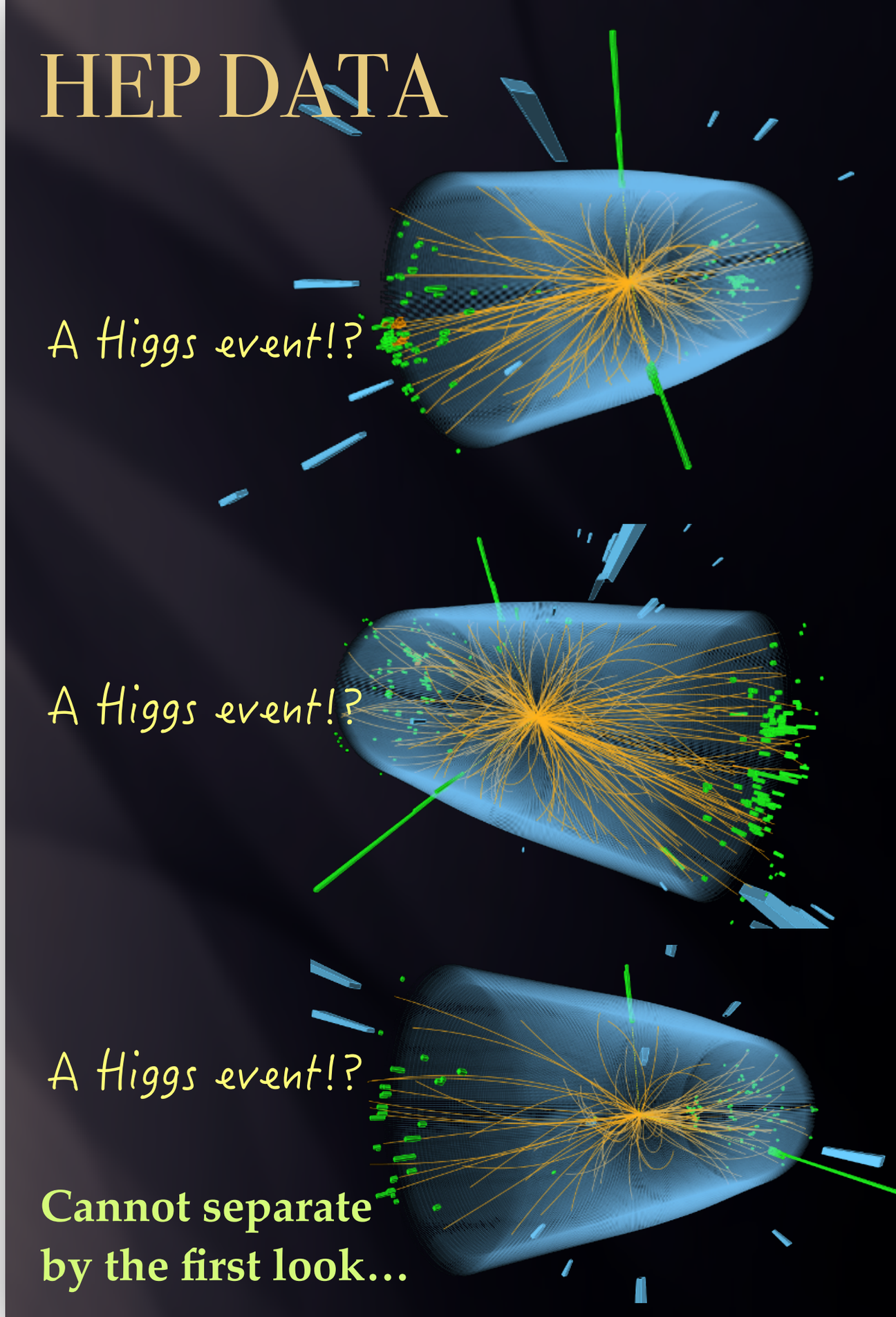
A nice kitty!

A cat maybe??

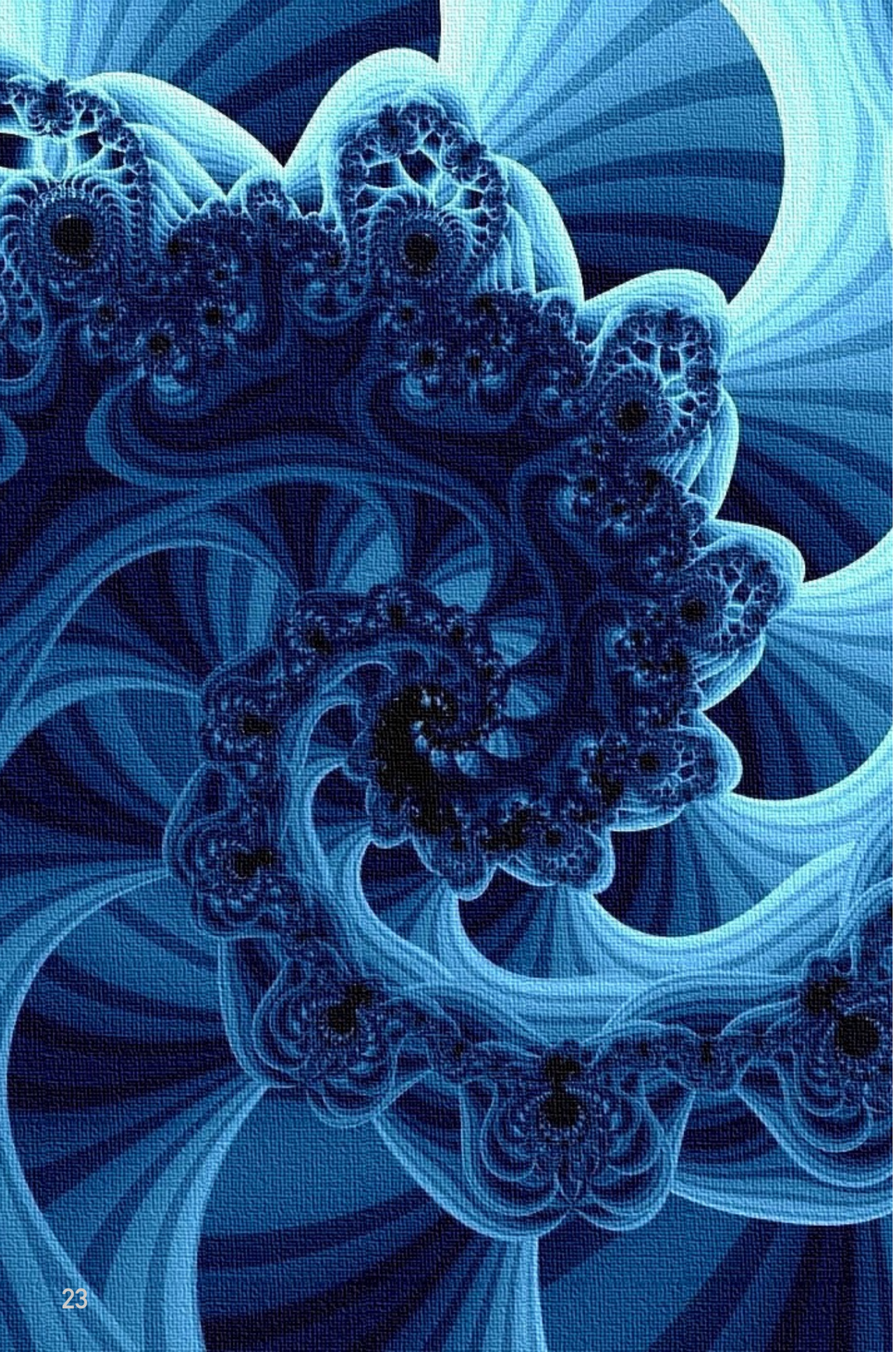
It's a toast, right?

One can tell by eyes quickly

HEP DATA



Cannot separate by the first look…



MODULE SUMMARY

- In this module we introduced the very famous and very interesting model, the Generative adversarial network (GAN) model. We have asked our simple GAN model to generate some Chinese characters which does not exist at all.