2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 3-4: Deep Structured Learning

Kai-Feng Chen
National Taiwan University

We shall go deeper this time!

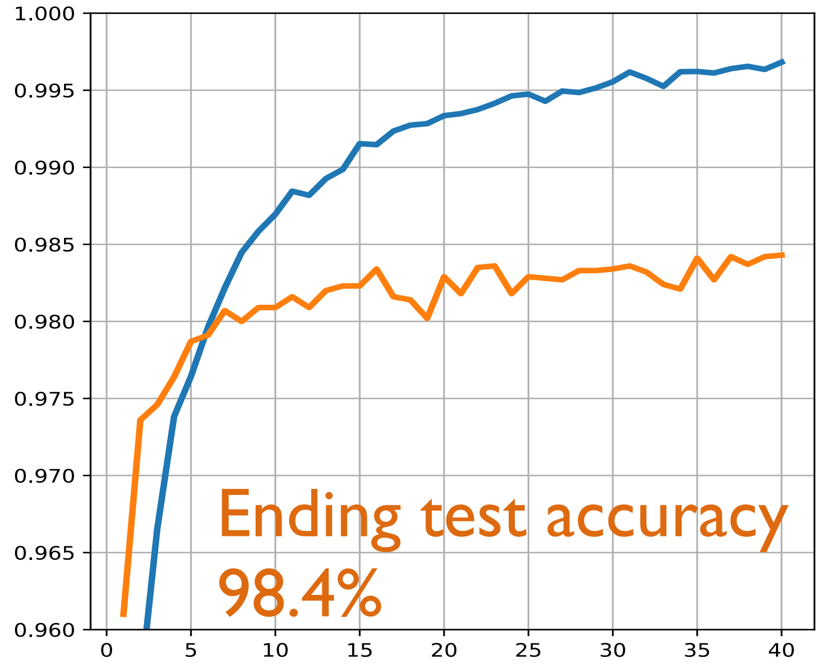
RECALL FROM THE LAST LECTURE…(AGAIN^2!)

- Last lecture we started to optimize our network with quite a few tricks that are commonly introduced nowadays, including a different choices of **loss function**, reducing the overtraining issue by introducing the **regularization** or **dropout**, or trying a different activation functions like **ReLU** which does not suffer from the slow learning problem as the classical sigmoid function.
- In the end we try to introduce a larger, complex network, with some of the tricks enabled. We were able to reach the best testing accuracy of 98.5%! Note this just reached the same performance as the best nonlinear SVM can do.
- Can we still further improve it by introducing a *deeper* network, or a different structure, such as the **convolutional neural network**?

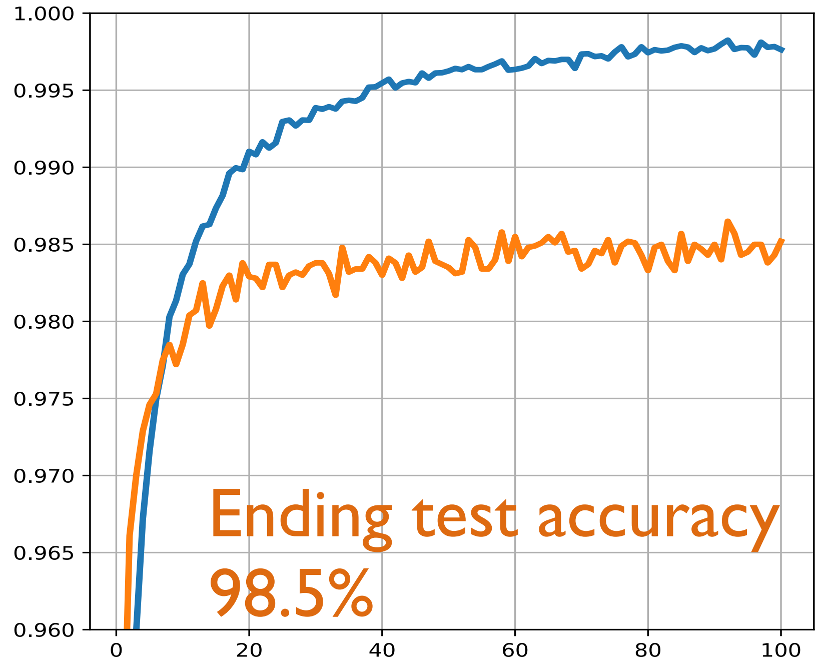
RECALL FROM THE LAST LECTURE…(AGAIN^2!)

- We have tested the network with more and more hidden layers and turns out to be hard to improve it further.
- The intrinsic problem is that the gradients are unstable with deeper network. The classical network may still work better but an effective training becomes difficult.

784-(256x4)-10
train for 40 epochs



784-(256x8)-10
train for 100 epochs



Using a **network with different structure** may resolve the problem, further improve the performance!

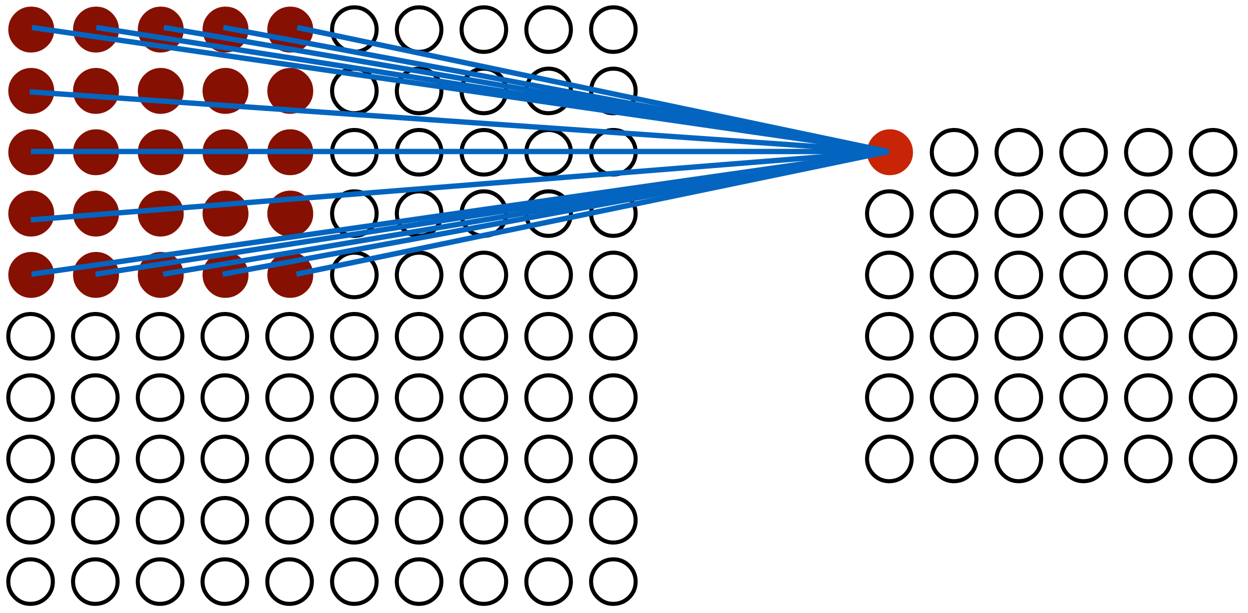
Here comes the Convolutional Neural Network…

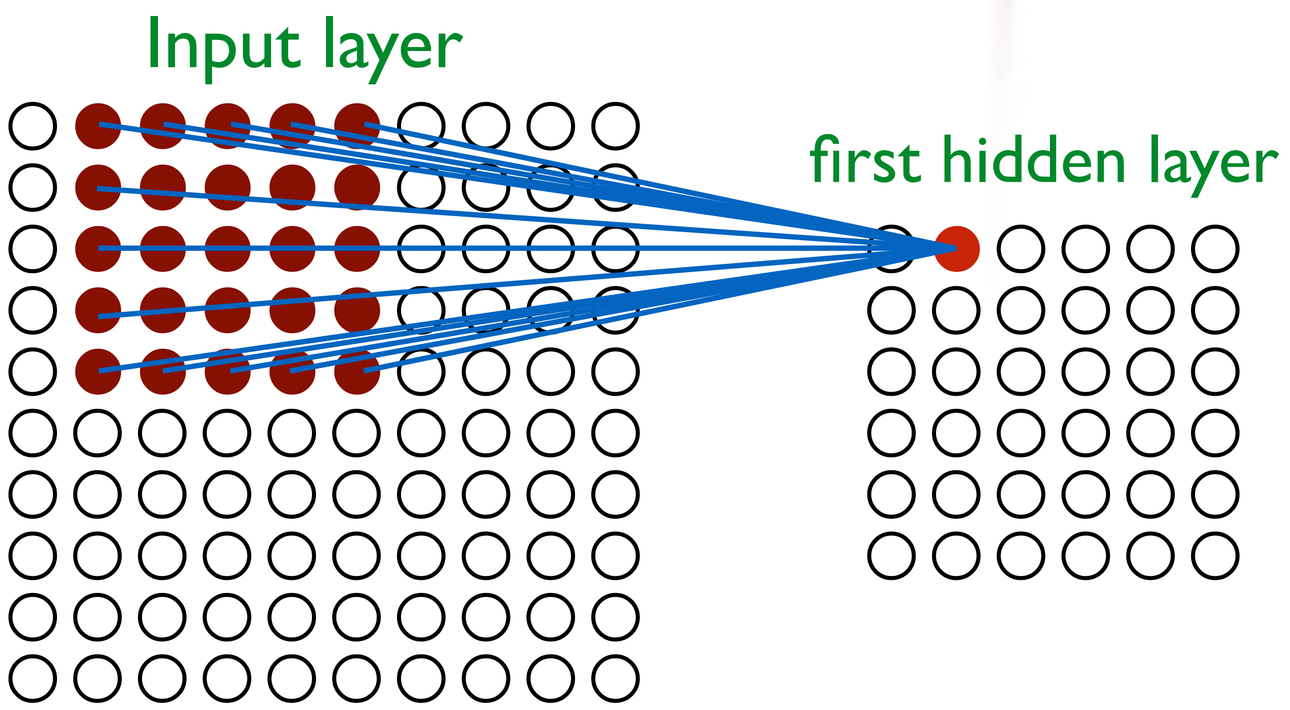
CONVOLUTIONAL NETWORK

- Up to now we are using a network first by “reshape” of the input 28×28 pixels into a flat input of 784 neurons. Although it works rather well but we do not take into account the nature of images in fact. **The local information (*of adjacent pixels*) is lost**.
- The convolutional networks use a special architecture which is particularly well-adapted to image recognition. The architecture of convolutional network makes the training of deep, multi-layer networks easier.
- There are several ideas introduced for the convolutional neural networks to be discussed in the following slides: **local receptive fields**, **shared weights**, and the **pooling**.

LOCAL RECEPTIVE FIELDS

- In a typical convolutional network, the input layer is encoded in the following structure. For example, instead of fully connected network, one only has the first 5×5 block of neurons being connected to one neuron in the first hidden layer, and next 5×5 block connected to the second neuron…



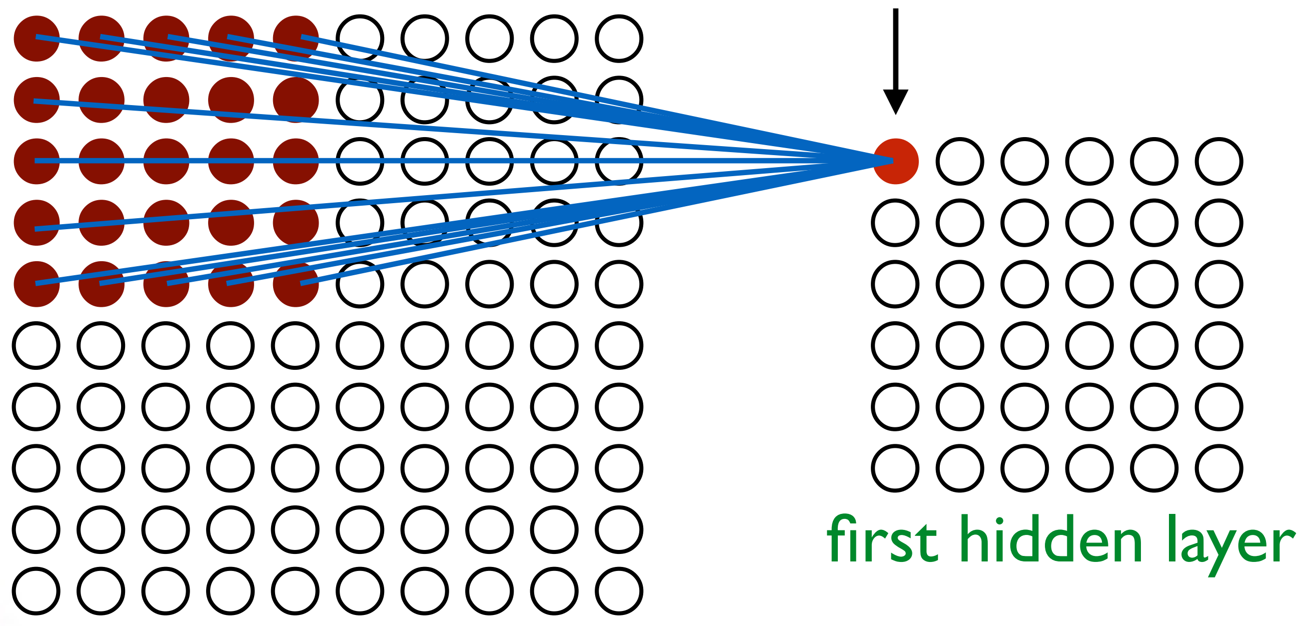


If we have 28×28 as the input image, and with a 5×5 local representative field, the first hidden layer will be 24×24.

SHARED WEIGHTS/BIAS

- The second important feature is that the local representative fields have a shared weights/bias through out the whole first hidden layer. e.g. the same 5×5 weights and a common bias are shared by all of the neurons on the first hidden layer.

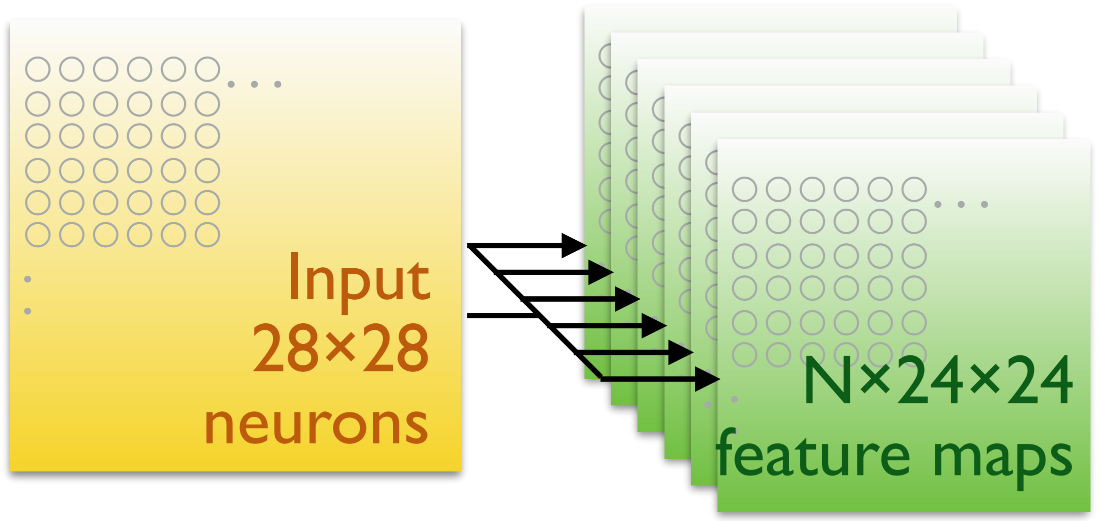
$$\sigma\left(b+\sum_{i=0}^{4}\sum_{j=0}^{4} w_{ij} v_{m+i,n+j}\right)$$

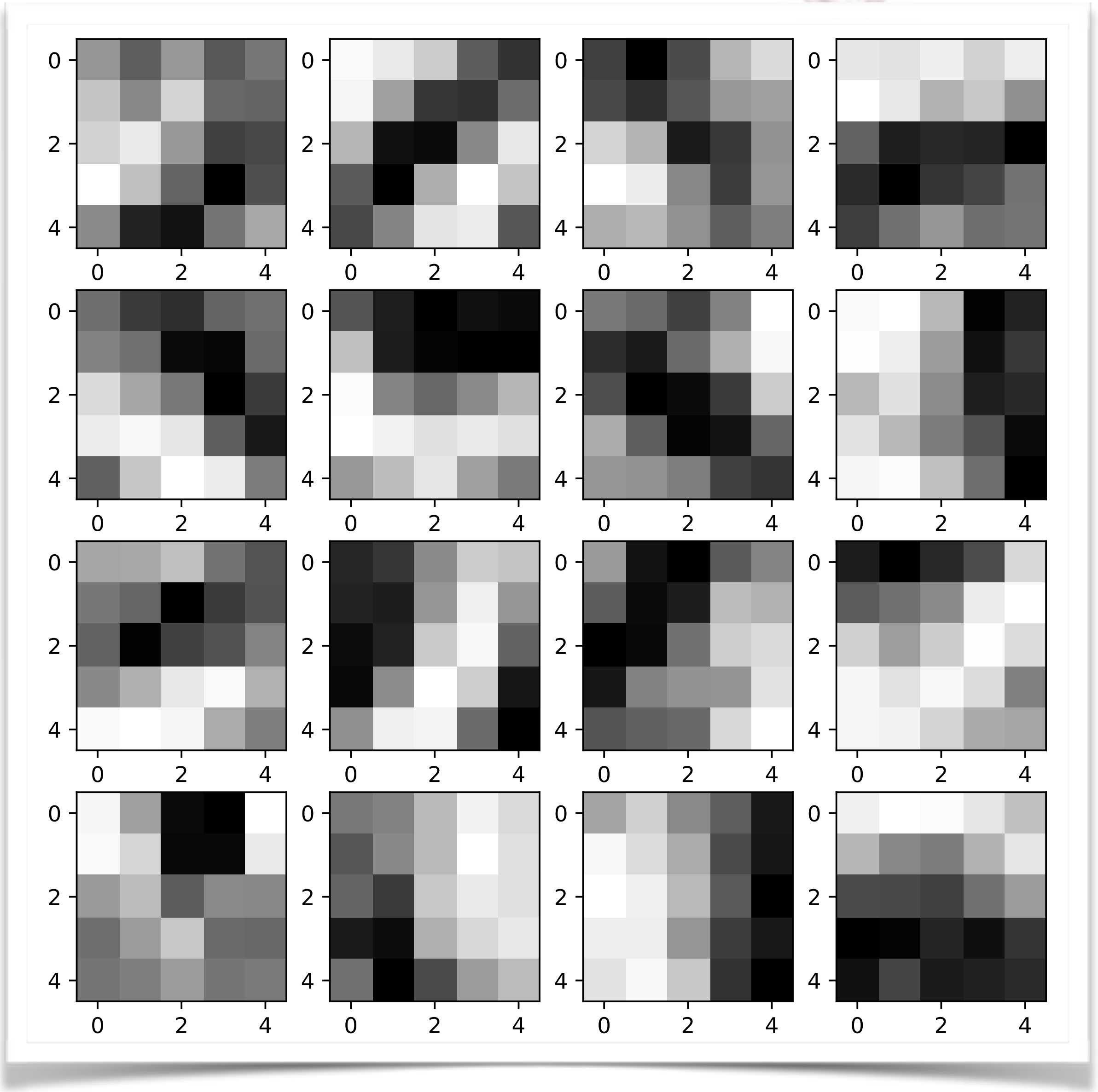


  - This means all of the neurons of the hidden layer can *detect exactly the same feature*.
  - The map from the input layer to the hidden layer is usually called a **feature map**.
  - A feature map only keep **25 weights and 1 bias**!
  - The shared weights/bias are often said to define a **kernel** or a **filter**.

FEATURE MAPS

- And it is very common to build multiple feature maps, i.e.



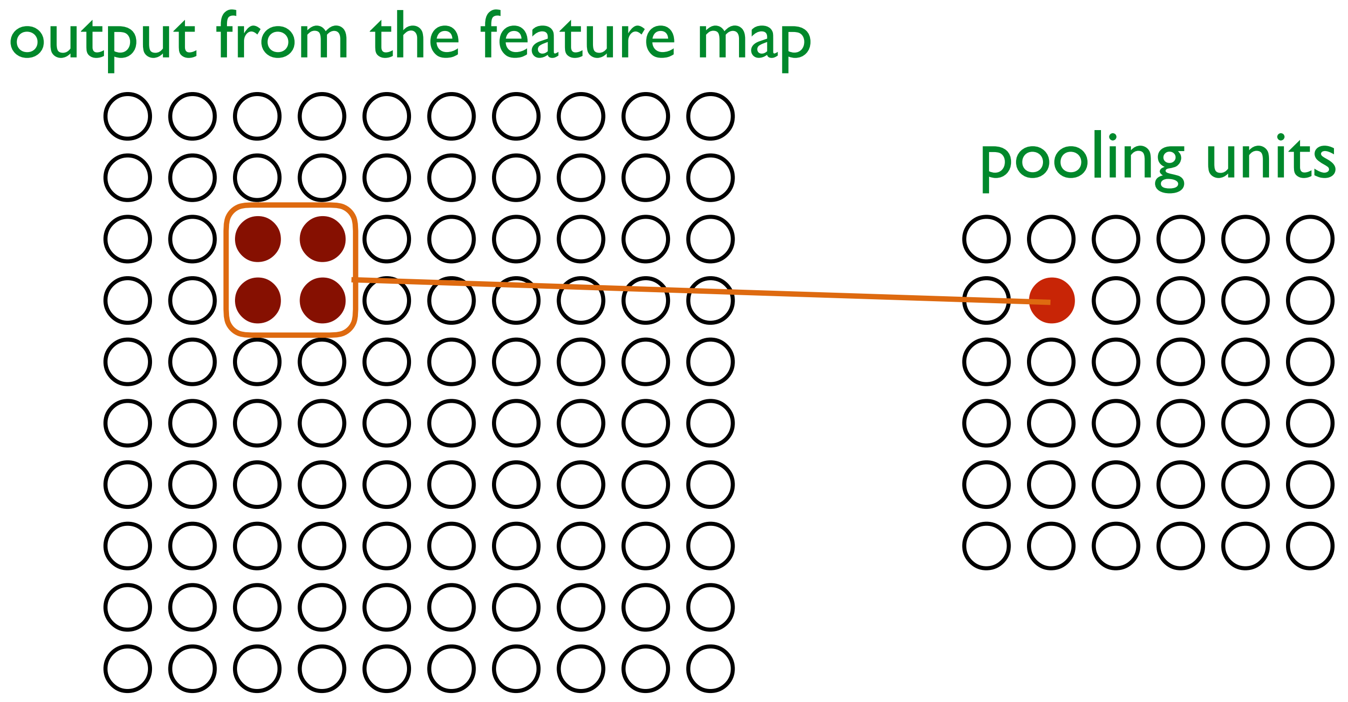


– For example here are the trained 16 feature maps (or kernels/filters) in the next example.

– Basically each map supposes to pick up a different feature from the input images!

POOLING LAYERS

- In addition to the convolutional layers, a pooling layer is usually added right after them. A pooling layer is to simplify the information from the convolutional layer, for example a 2×2 pooling layer shrink the input 24×24 feature map into a 12×12 units:

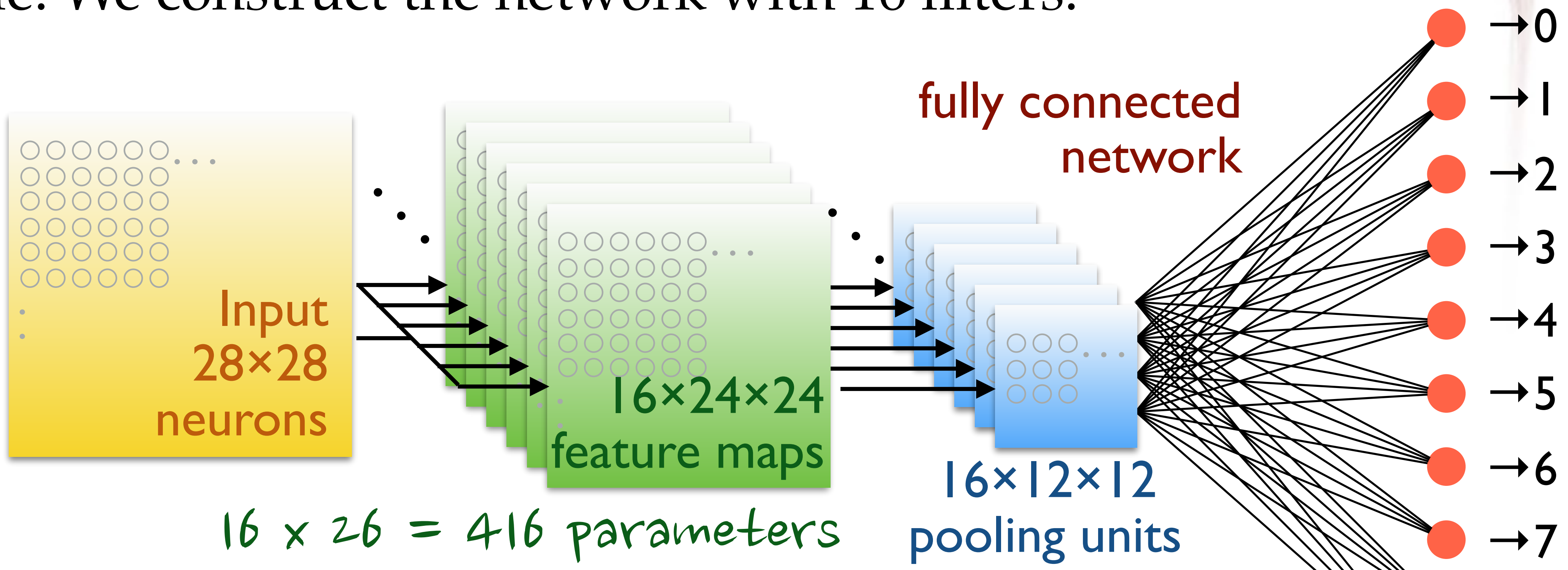


Usually this is applied to each feature map output layer

  - **Max-pooling**: simply outputs the maximum activation value in input region.
  - **L2 pooling**: take the square root of the quadrature sum of the activations.
  - No additional weight/bias but just condensing information from the convolutional layer.

PUT ALL TOGETHER: CONVOLUTIONAL NETWORK

- Here we just draw the structure of a typical convolutional network. And it will be implemented in our upcoming example code. We construct the network with 16 filters:



Although you may think this is a complicated model, but in fact the total # of parameters are much smaller than our previous example, only **23,466** weights/bias!

PUT ALL TOGETHER (II)

- Easy implementation with Keras:

Just the model discussed in the previous page!

```
. . . . . .
from keras.models import Sequential
from keras.layers import *
from keras.optimizers import Adadelta

model = Sequential()
model.add(Reshape((28,28,1), input_shape=(28,28)))
model.add(Conv2D(16, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))           ⇑ 5x5 convolutional layer
model.add(Flatten())                     ⇑ 2x2 pooling layer
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=Adadelta(),
              metrics=['accuracy'])
. . . . . .
```

l304-example-01.py (partial)

PUT ALL TOGETHER (III)

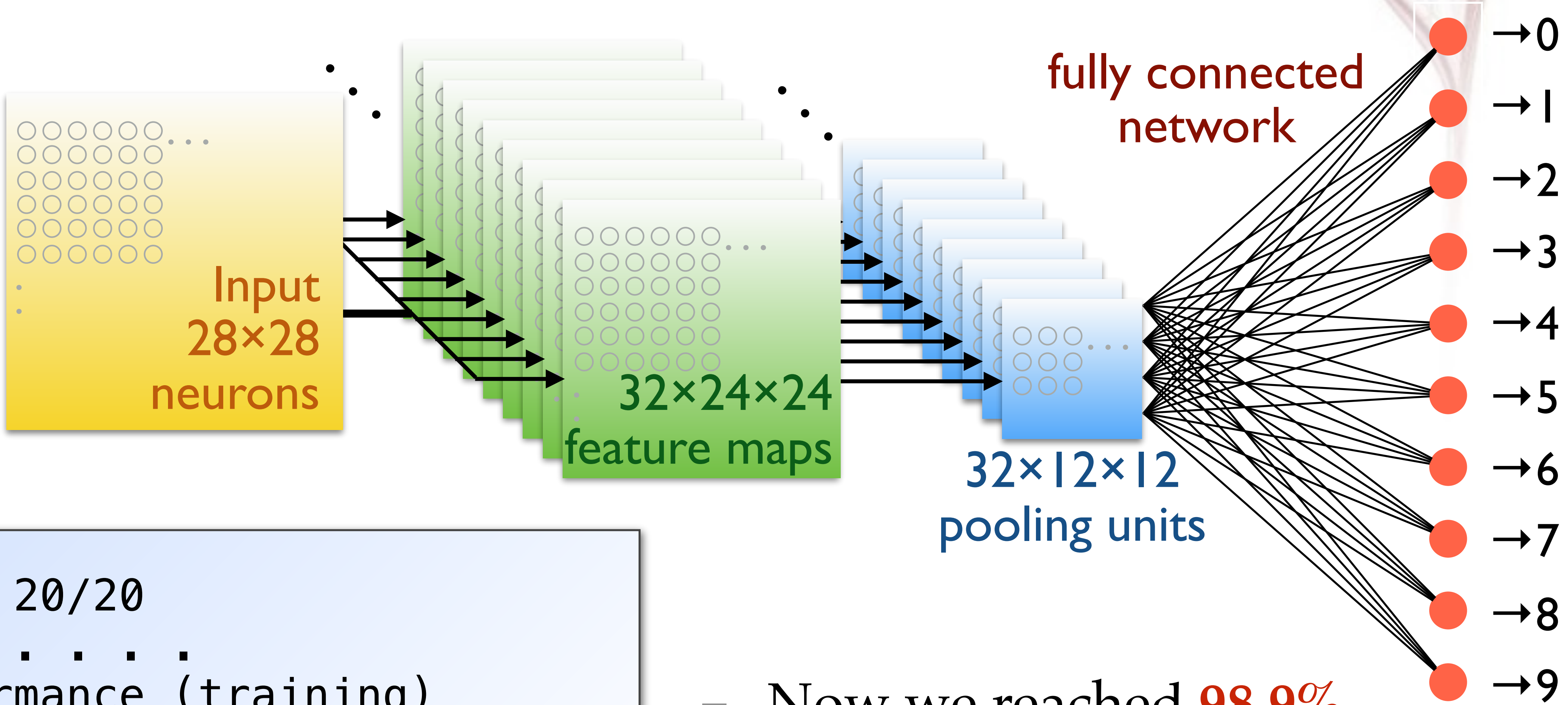
- And we can reach a very good performance already:

```
. . . . . .
Epoch 20/20
60000/60000 [===========] 13s 217us/step - loss: 0.0363 - acc: 0.9890
- val_loss: 0.0371 - val_acc: 0.9874
Performance (training)
Loss: 0.02537, Acc: 0.99267
Performance (testing)
Loss: 0.03712, Acc: 0.98740
```

- A testing accuracy of 98.7% reached, only 126 images are mis-identified. Remember we only put a layer of convolutional network and **# of parameters is reduced by a factor of 28** comparing to the previous flat 784-512-512-10 network!
- Can we do even better? *Let's try to add more layers!*

HOW ABOUT ADDING MORE FEATURES MAPS?

- Let's just double the feature maps? Can we improve the model?

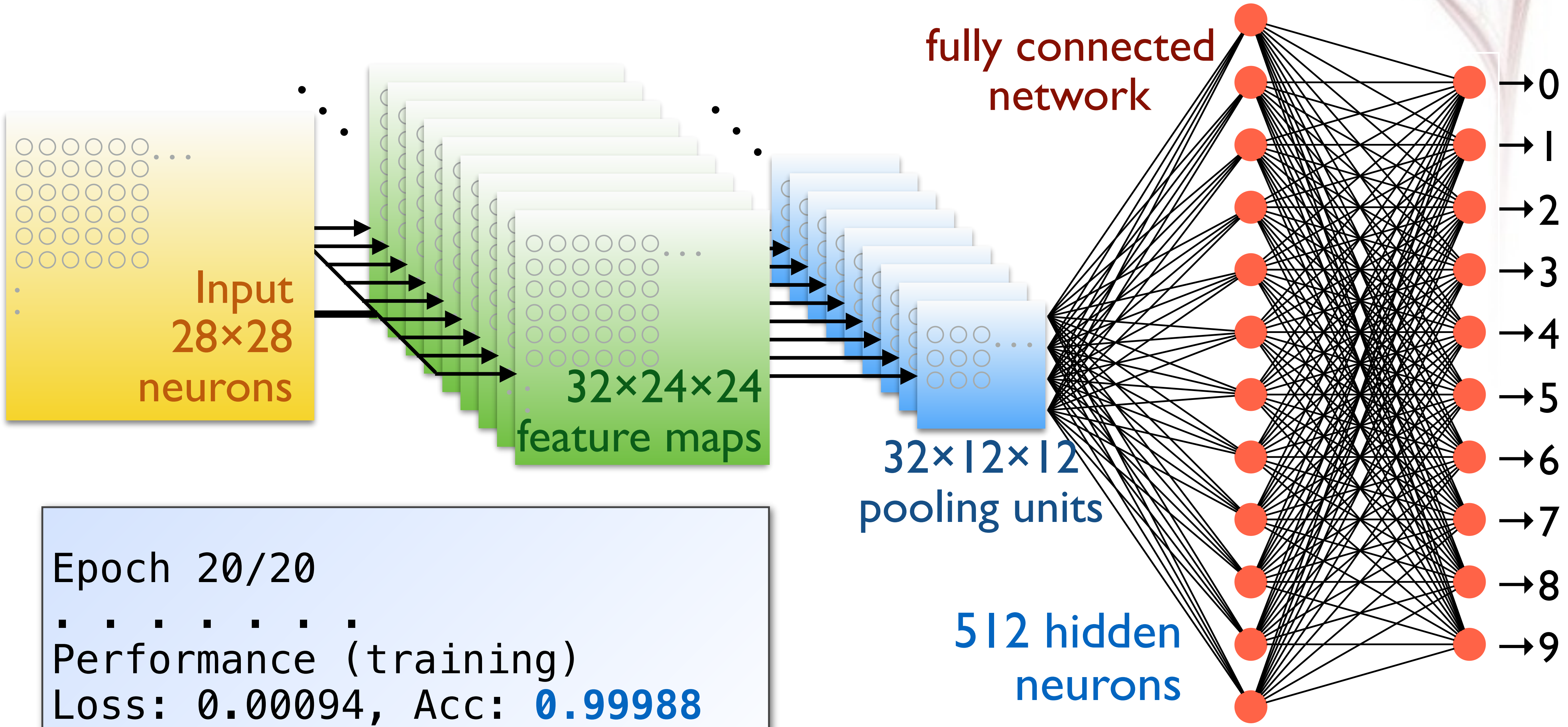


```
Epoch 20/20
. . . . . . .
Performance (training)
Loss: 0.01816, Acc: 0.99518
Performance (testing)
Loss: 0.03244, Acc: 0.98900
```

  - Now we reached **98.9%** test accuracy, only 110 digits are wrongly tagged!

ADD ANOTHER HIDDEN FULLY CONNECTED LAYER?

- Let's add another fully connected layer and see the performance?



```
Epoch 20/20
. . . . . . .
Performance (training)
Loss: 0.00094, Acc: 0.99988
Performance (testing)
Loss: 0.02896, Acc: 0.99230
```

  – Now we go beyond **99.2%**!

DOUBLED LAYERS!

- Let's config our model by two convolution+pooling layers, and two fully connected layers. Then see how good can we do here?

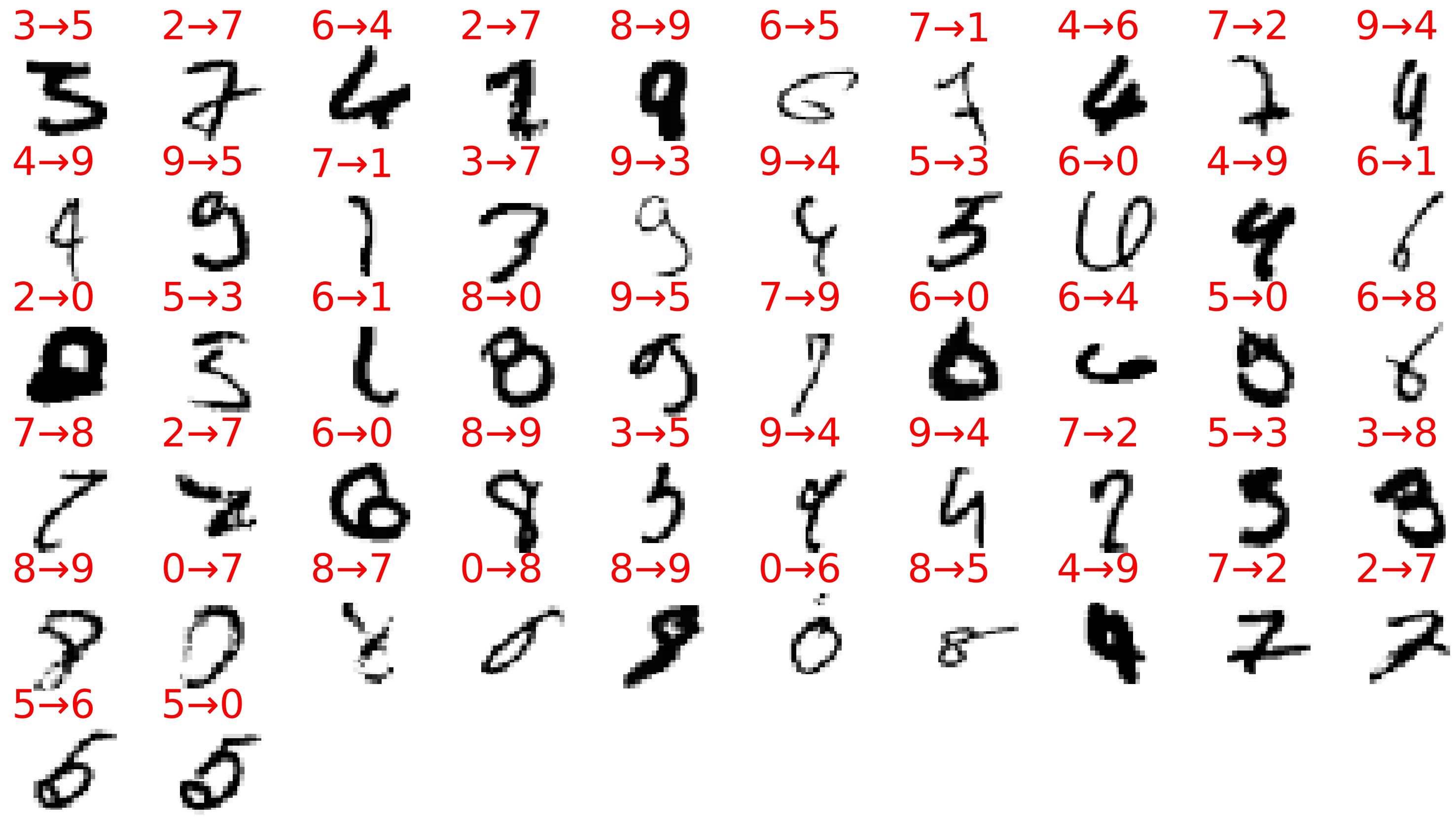
```
. . . . . .
model = Sequential()
model.add(Reshape((28,28,1), input_shape=(28,28)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
. . . . . .
```

1304-example-01a.py (partial)

```
Performance (training)
Loss: 0.00167, Acc: 0.99960
Performance (testing)
Loss: 0.01988, Acc: 0.99480
```

– Now we can almost reach **99.5%**!

DOUBLED LAYERS! (II)



- Now we only have 52 wrongly tagged images (0.52% failed).
- Some of them are also difficult for real humans!
- Remember the best trained network (*world record*) is with 0.21% failure rate. Still rooms to be improved!

The convolutional neural network is a kind of deep network **good for image recognition**!

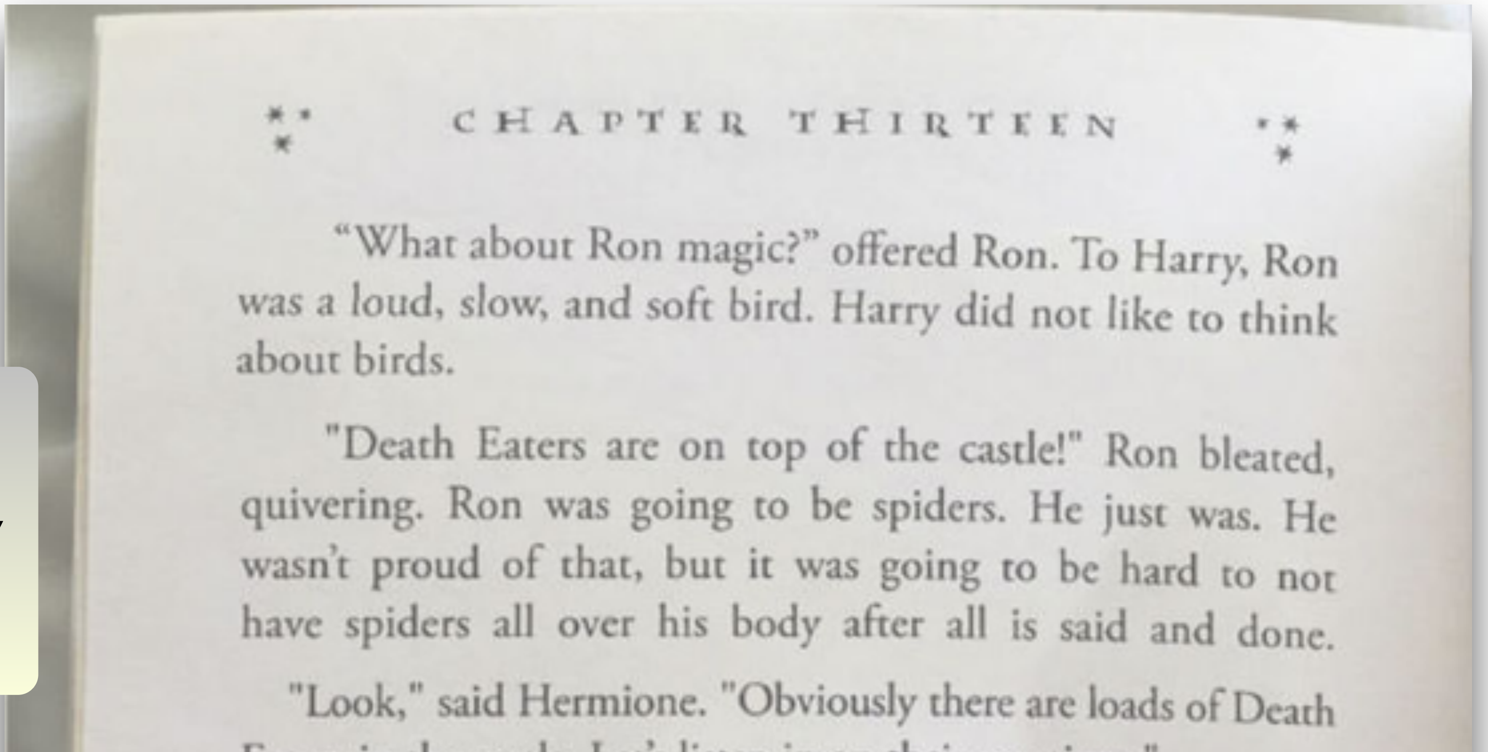
STRUCTURE DOES MATTER

- It is very interesting that *by changing the structure of network,* it contains a smaller number of tunable parameters, but also boost the performance. This is due to the structure makes the network easier to train and can reach a very good performance within a **limited training time**.
- In fact, by using classical multilayers of network, the performance can be as good as CNN but the training can take a very long time and a lot of tricks need to be adopted.
- On the other hand, CNN is good for image recognition, but for other topics, one may want to introduce a different structure, or even different concepts to have a powerful ML program.

Let's quickly comment on some modern networks which has been developed for different topics!

OTHER DEEP NETWORKS & IDEAS

- **Recurrent neural network (RNN):**
  - Up to now our network has a fixed flow throughout the training, but what will happen if we allow the network to vary itself along with time sequence?
  - Unlike feedforward neural network, RNN can use their internal state to process a sequence of inputs. This gives RNN a good approach to the unsegmented data, for example, language/speech recognition.



CHAPTER THIRTEEN

"What about Ron magic?" offered Ron. To Harry, Ron was a loud, slow, and soft bird. Harry did not like to think about birds.

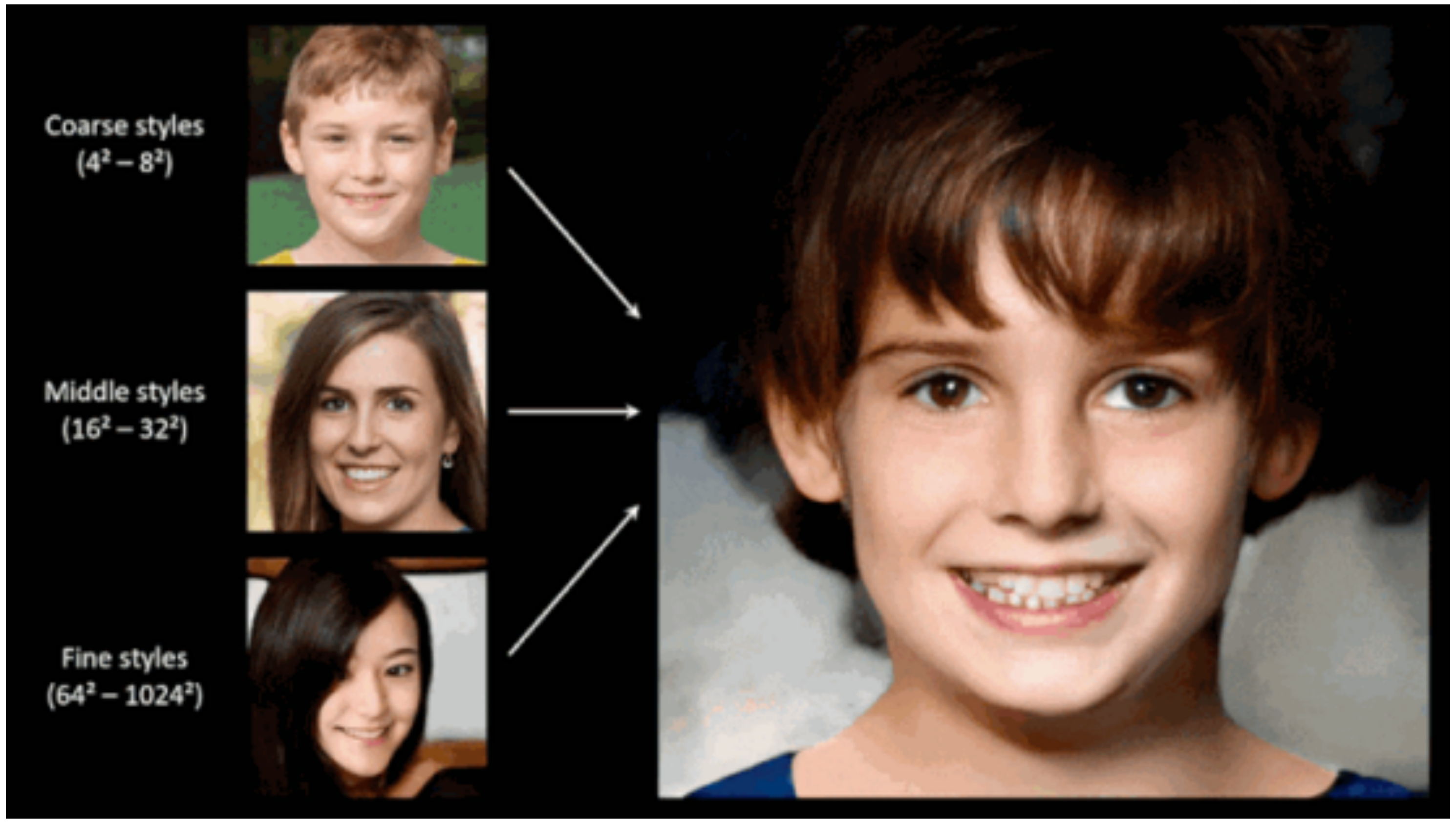
"Death Eaters are on top of the castle!" Ron bleated, quivering. Ron was going to be spiders. He just was. He wasn't proud of that, but it was going to be hard to not have spiders all over his body after all is said and done.

"Look," said Hermione. "Obviously there are loads of Death

A Harry Potter chapter "written" by AI program…

OTHER DEEP NETWORKS & IDEAS (II)

- **Generative adversarial network (GAN):**
  - The basic structure of GAN is to have two network "fighting" with each other: one is to find "fake" images out of the pool, another one is to generate fake images.
  - Once it has been trained, you can use the generator to produce lots of "nearly true" fake images, e.g. photo of a person who never exists in the real world, or convert your doodle to a fancy graph!



OTHER DEEP NETWORKS & IDEAS (III)

- **Reinforcement Learning (RL):**
  - In our example network, the required responses of our model are relatively simple (just which digit, 0-9). But in many problems, for example, playing chess, this is not a simple task as no clear classification of good/bad labels.
  - Then the reinforcement learning is a kind of idea to build the environment for your program to learn how to survive by itself (only give it a goal to reach, e.g. beating the opponent, getting higher scores etc). *Let the environment to be the teacher*.
  - A famous example is the **AlphaGoZero**, which is trained without any prior knowledge of Go, but just let to figure out how to play Go by itself!

INTERMISSION

- It is very interesting to see what are he feature maps looked like exactly (an example has been shown in an earlier slide), since the feature maps are kind of direct demonstration how the CNN "look" at the input images.
- This can be carried out by adding the following short code to the end of training (following the model in `l304-example-01.py`):

```
. . . . . . .
fig = plt.figure(figsize=(8,8), dpi=80)
for i in range(16):
    plt.subplot(4,4,i+1)
    w = model.layers[1].get_weights()
    plt.imshow(w[0][:,:,0,i], cmap='Greys')
plt.show()
```

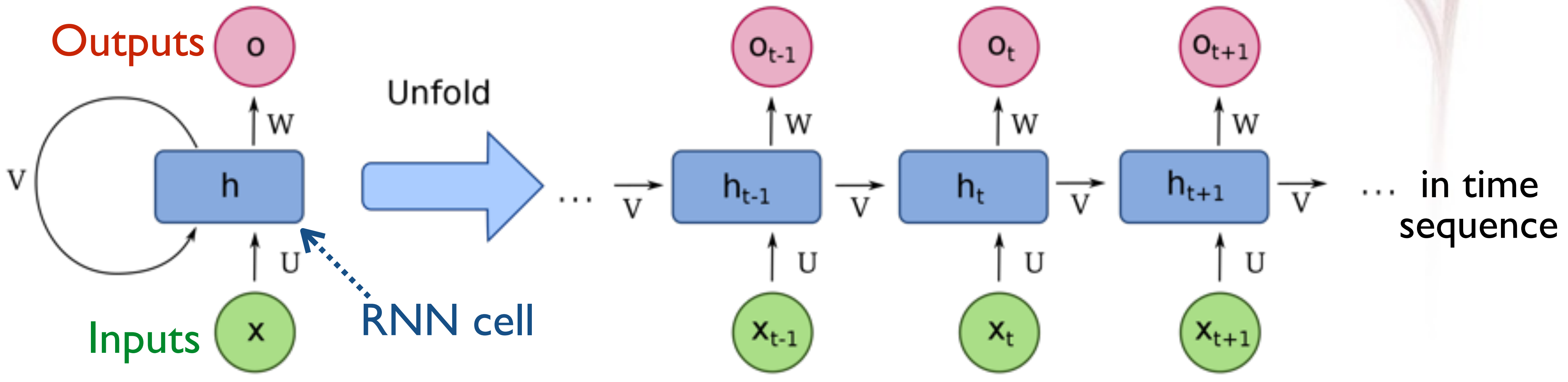
l304-example-01b.py (partial)

You may try it now!

Let's play with an example RNN and an example GAN here!

VANILLA RNN

- Classical ("Vanilla") RNN has a structure to connect the information from the previous time frame to the next, in addition to the regular inputs:



- Ideally the information can be **passed to next time frame**, but in practical when training a vanilla RNN using back-propagation, the gradients which are back-propagated can easily "vanish" (*the network tends to remember only recent frames*) or "explode".
- At least the vanish gradient problem can be resolved by adding **"memory"** capability.

WHY A MEMORY CELL IS IMPORTANT?

- Let's take an analogy, by reading/examination the following short story (*suppose you are using a NN to process an article*):

  June was born in France. (…a long story and blah-blah…) Surely, she can still speak nearly perfect French.

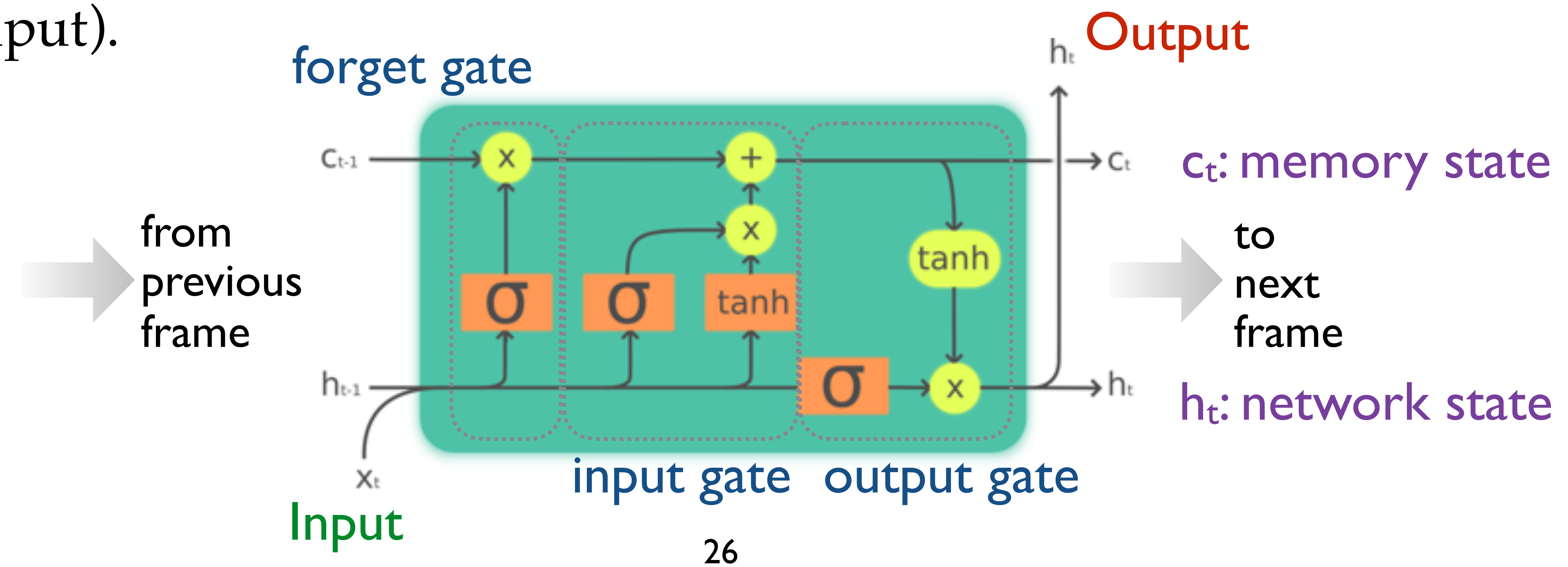
- If there an memory cell, the important information (*such as born in France*) can be kept and eventually it can build up a connection with the *French speaking capability* in the end. But if a classic RNN is deployed, the information given in the earlier lines will fade out with time sequence due to the vanish gradient problem:

  June was born in France. (…a long story and blah-blah…) Surely, she can still speak nearly perfect French.

It will be difficult to connect the key information of the article with vanish gradients.

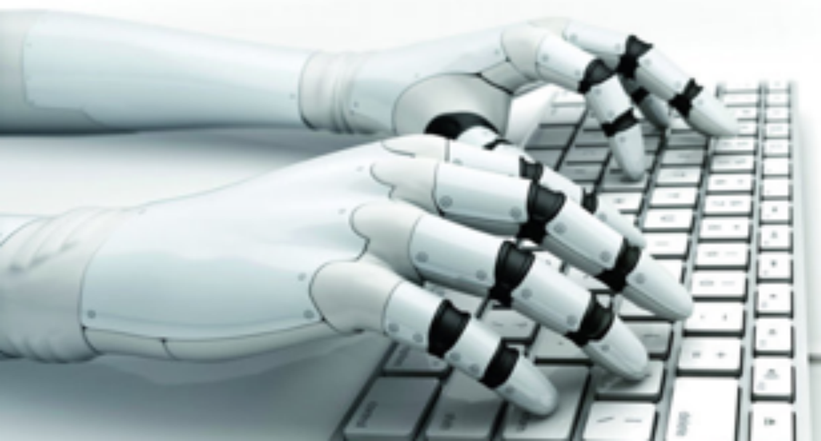
LONG SHORT-TERM MEMORY

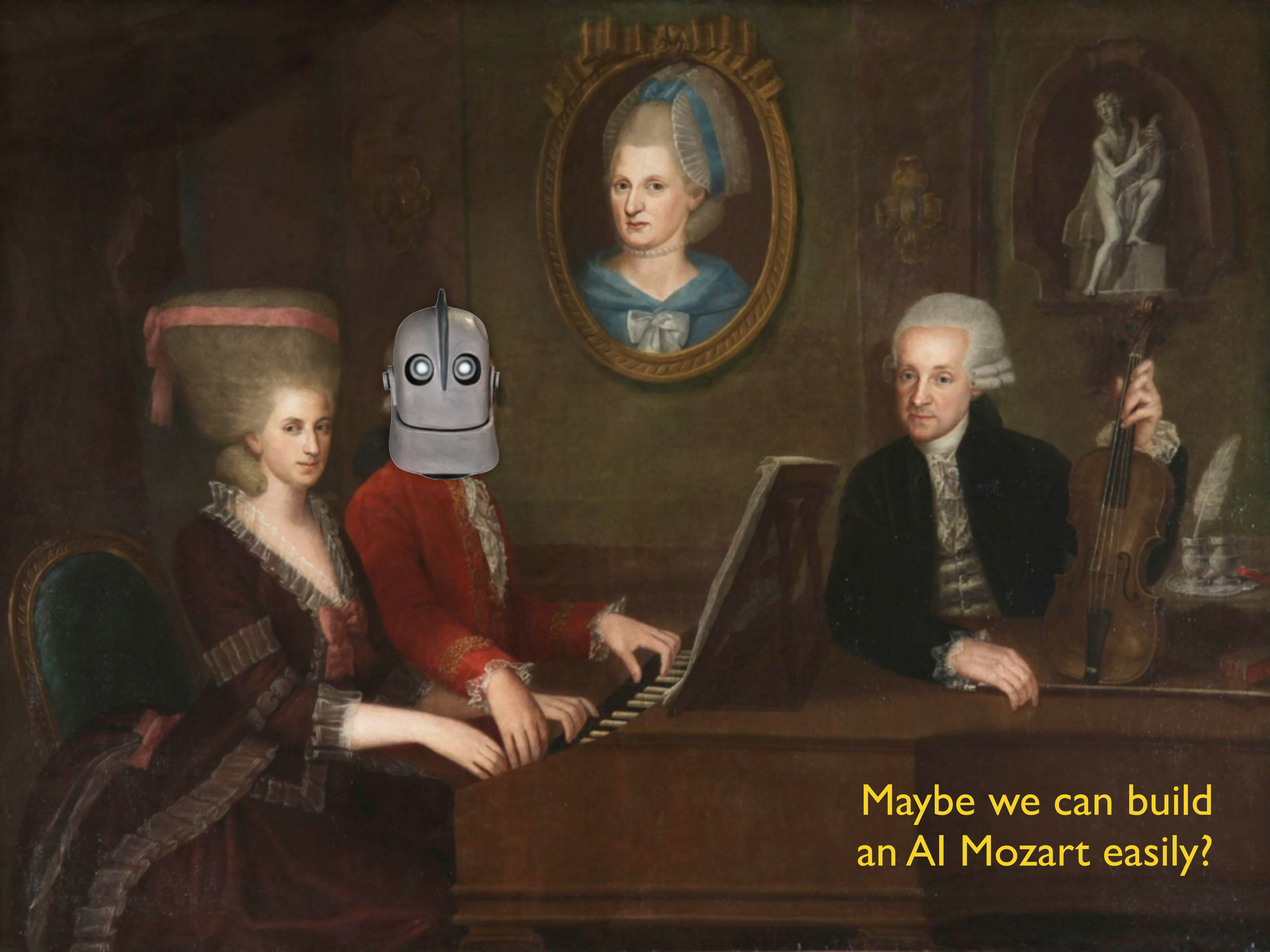
- **Long short-term memory (LSTM)** is a kind of recurrent neural network architecture. It has the capability to train long-term dependencies. It was first introduced by Hochreiter & Schmidhuber in 1997 and it is widely used in many different places nowadays.
- The key idea is to replace the classical RNN unit with the LSTM unit, which consists of a memory cell + 3 “gates” (forget/input/ouput).



LONG SHORT-TERM MEMORY (II)

- With such a structure, it will be easier for the network to remember a long sequence of data, and keep/remember the key information.
- It can be used to do language processing, music processing, as far as we can convert the "words" or "notes" into input data.
- With a trained model it can be also used to generate articles (as the so-called "AI writer") or music ("AI composer").
- For our amusement, let's practice a simple **LSTM model with music data**, and see if our simple model can remember (being trained) and **generate a nice piece of music** or not!

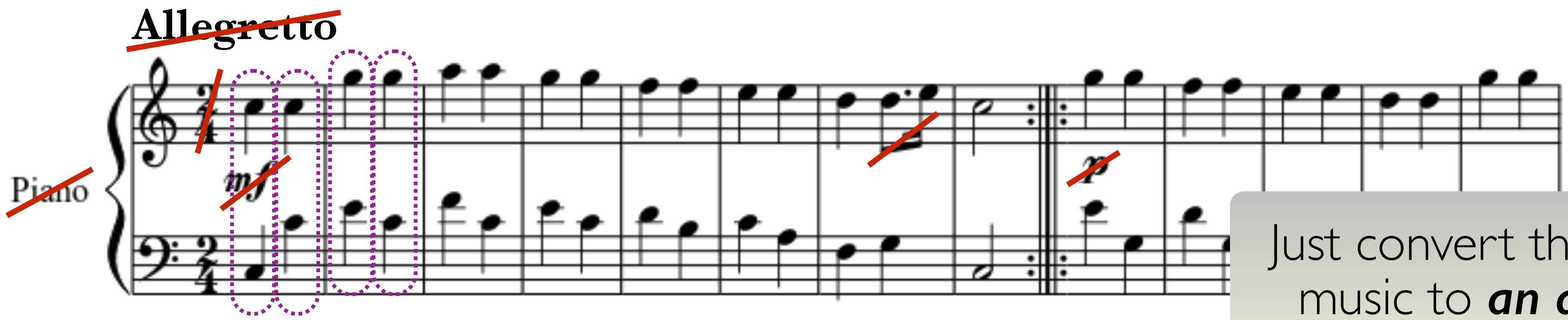




Maybe we can build an AI Mozart easily?

MUSIC DATA: DECODING

- In fact it should not be too difficult to convert the music data (from a MIDI file or so) into a sequence of data.
- But a full song can be quite complicated! Let's give up some of the information at the first place — the instrument, volume, and tempo, tonality.
- There are still pitches, duration, delay, etc. Just focus on the **CHORD/NOTE** only for now and forget about everything else…



→ **C3+C5, C4+C5, E4+G5, C4+G5, …**

Just convert the sheet music to ***an article***, where a "word" contains a "chord/note".

MUSIC DATA: DECODING

Not going into the details how to phrase a MIDI file, just show you a piece of code which can analyze the track and produce a list "chords" with a tool named **mido**.

```python
from mido import MidiFile, MidiTrack, Message, MetaMessage

def decode_midi(filename, maxnotes = 0):

    mid_in = MidiFile(filename)
    notes = []
    for track in mid_in.tracks:
        sum_of_ticks = 0
        pool = []
        for msg in track:
            sum_of_ticks += msg.time
            if msg.type=='note_on':
                for p in pool:
                    if p[1]==msg.channel and p[2]==msg.note:
                        if sum_of_ticks-p[0]>0: notes.append([p[0], p[2], sum_of_ticks-p[0]])
                        pool.remove(p)
                        break
                else: pool.append([sum_of_ticks, msg.channel, msg.note])
            if msg.type=='note_off':
                for p in pool:
                    if p[1]==msg.channel and p[2]==msg.note:
                        if sum_of_ticks-p[0]>0: notes.append([p[0], p[2], sum_of_ticks-p[0]])
                        pool.remove(p)
                        break
        for p in pool:
            if sum_of_ticks-p[0]>0: notes.append([p[0], p[2], sum_of_ticks-p[0]])

    notes = np.array(notes)
    ticks = np.unique(notes[:,0])

    pack = []
    for idx in range(len(ticks)-1):
        notes_at_ticks = np.unique(notes[notes[:,0]==ticks[idx]], axis=0)
        chord = str([p for p in notes_at_ticks[-maxnotes:,1]])
        pack.append(chord)
    return pack
```

⇐ loop over "tracks"

⇐ loop over "messages"

⇐ count "ticks"

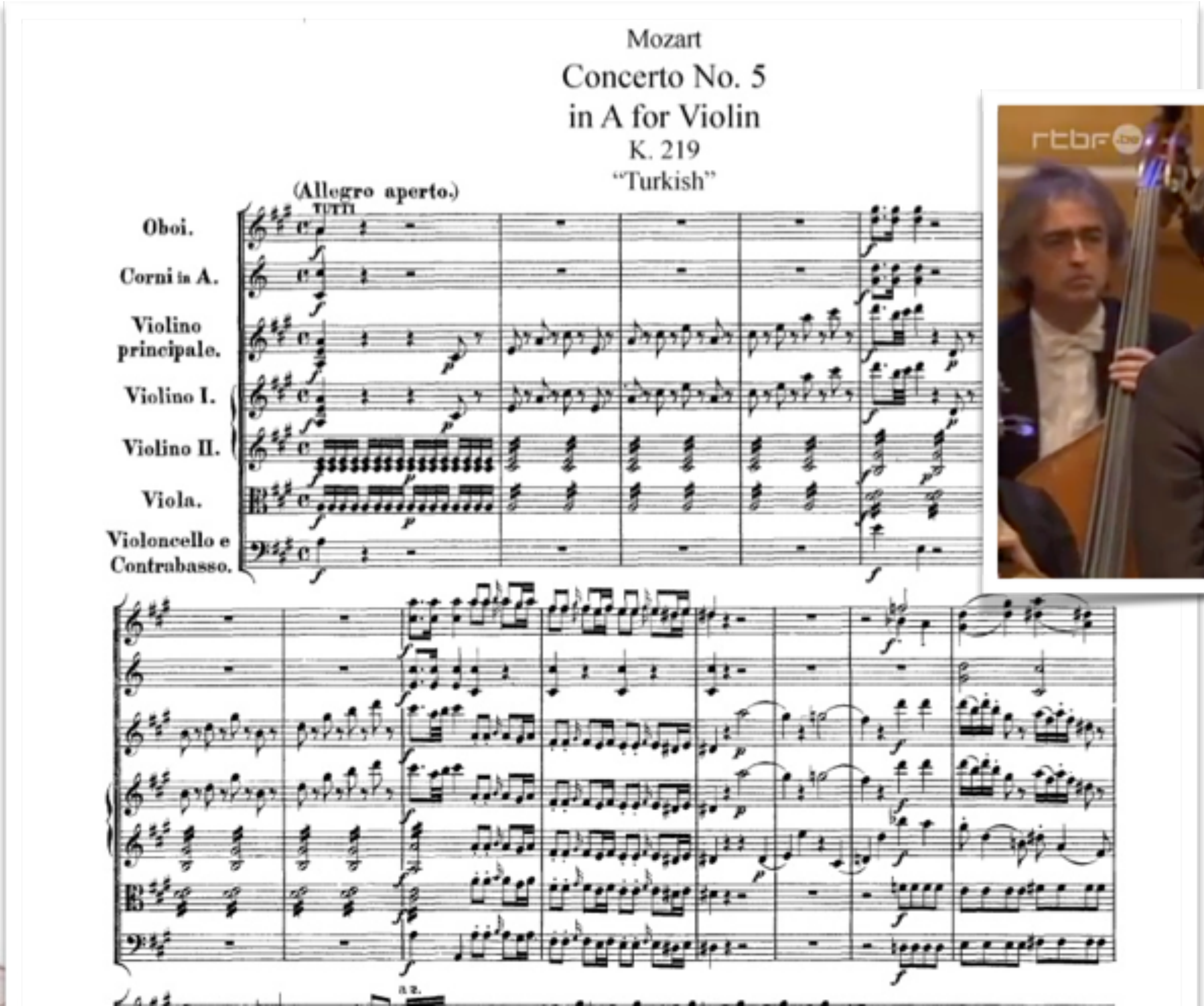
interpret "note on/off" message

output the "chords"
⇐ as a list of strings

midi_phraser.py (partial)

DECODING TEST

- Let's test this "decoding" with **Mozart's Violin Concerto No. 5**:



Surely not from the real music but from an existing **MIDI file**…

DECODING TEST (II)

- Decoding from a MIDI file (*not the original concerto but a rearranged version for violin & piano, but it does not matter here!*)

```python
from midi_phraser import *

data = decode_midi('mozk219a.mid')

for idx, chord in enumerate(data):
    print('#%d: %s' % (idx,chord))

encode_midi('test.mid', data)
```

l304-example-02.py

```
#0: [33, 45, 61, 64, 69]
#1: [45, 49, 52]
#2: [57]
#3: [45, 49, 52]
#4: [57]
#5: [45, 49, 52]
#6: [57]
#7: [45, 49, 52]
#8: [57]
#9: [45, 49, 52]
#10: [57]
#11: [45, 49, 52, 61]
#12: [57]
#13: [45, 49, 52]
#14: [57]
#15: [45, 49, 52, 64]
#16: [57]
. . . . . .
```

These are the **pitch numbers** suppose to be played at the same time!

- The frequency for each pitch can be calculated by

$$f_m = 2^{\frac{m-69}{12}} \times 440 \text{ Hz}$$

DECODING+ENCODING

- The question is — are we giving up too much information (*remember we already dropped the duration, delay. etc!*) at the first place and the music does not sound like a song anymore?
- Let's simply pack it back to a MIDI file and check if the music still sounds like a Mozart concerto?

(not too bad?)

```
def encode_midi(filename, data, tempo_set=500000):
  mid_out = MidiFile()
  track = MidiTrack()
  mid_out.tracks.append(track)                  ⇓ set to 'Harp'
  track.append(Message('program_change', program=46, time=0))
  track.append(MetaMessage('set_tempo', tempo=tempo_set, time=0))
  for pack in data:
    chord = eval(pack)   no idea about the duration
    delay = 120        ⇐ of each note, just set to 120
    for pit in chord:
      track.append(Message('note_on', note=pit, velocity=64, time=0))
    track.append(Message('note_off', note=chord[0], velocity=64, time=delay))
    for pit in chord[1:]:
      track.append(Message('note_off', note=pit, velocity=64, time=0))
  mid_out.save(filename)
```

midi_phraser.py (partial)

PREPARE THE DATA FOR OUR NETWORK

- In other to feed the music data we just extracted from MIDI file, there is still one more step to map the chords to an index number.
- This can be carried out with a small piece of code like this:

```
data = decode_midi('mozk219a.mid')

all_chords = sorted(set(data))
n_chords = len(all_chords)
chords_to_idx = dict((v, i) for i,v in enumerate(all_chords))
idx_to_chords = dict((i, v) for i,v in enumerate(all_chords))

print('Total # of chords:',n_chords)
for key in chords_to_idx:
    print(key,'==>',chords_to_idx[key])
```

l304-example-03.py (partial)

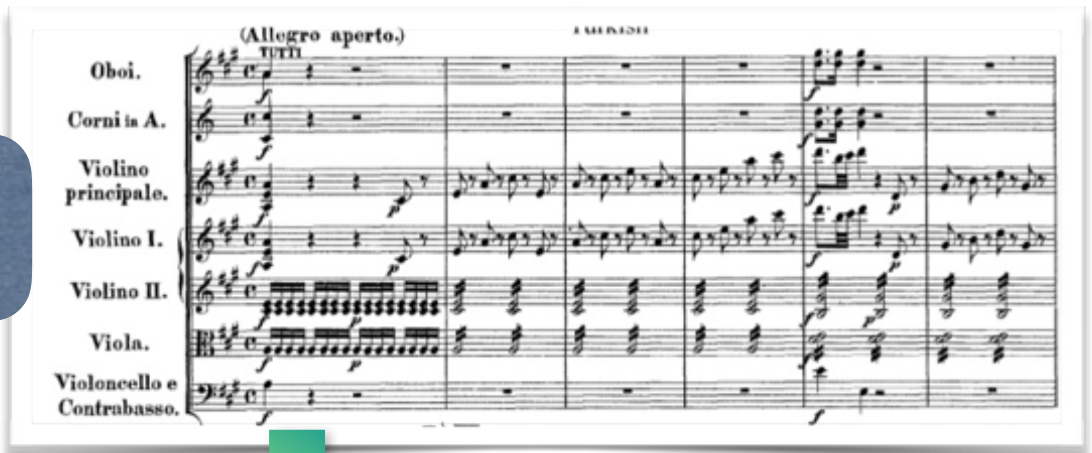
```
Total # of chords: 792
[100] ==> 0
[33, 45, 61, 64, 69, 81] ==> 1
[33, 45, 61, 64, 69] ==> 2
[36, 48, 60, 80] ==> 3
. . . . . .
```

By introducing such a dictionary, we can further "encode" the music data into a sequence of integers!

PREPARE THE DATA FOR OUR NETWORK (II)

- This would allowed us to convert the input music data into a very compact sequence of numbers:

sheet music / MIDI file



list of chords

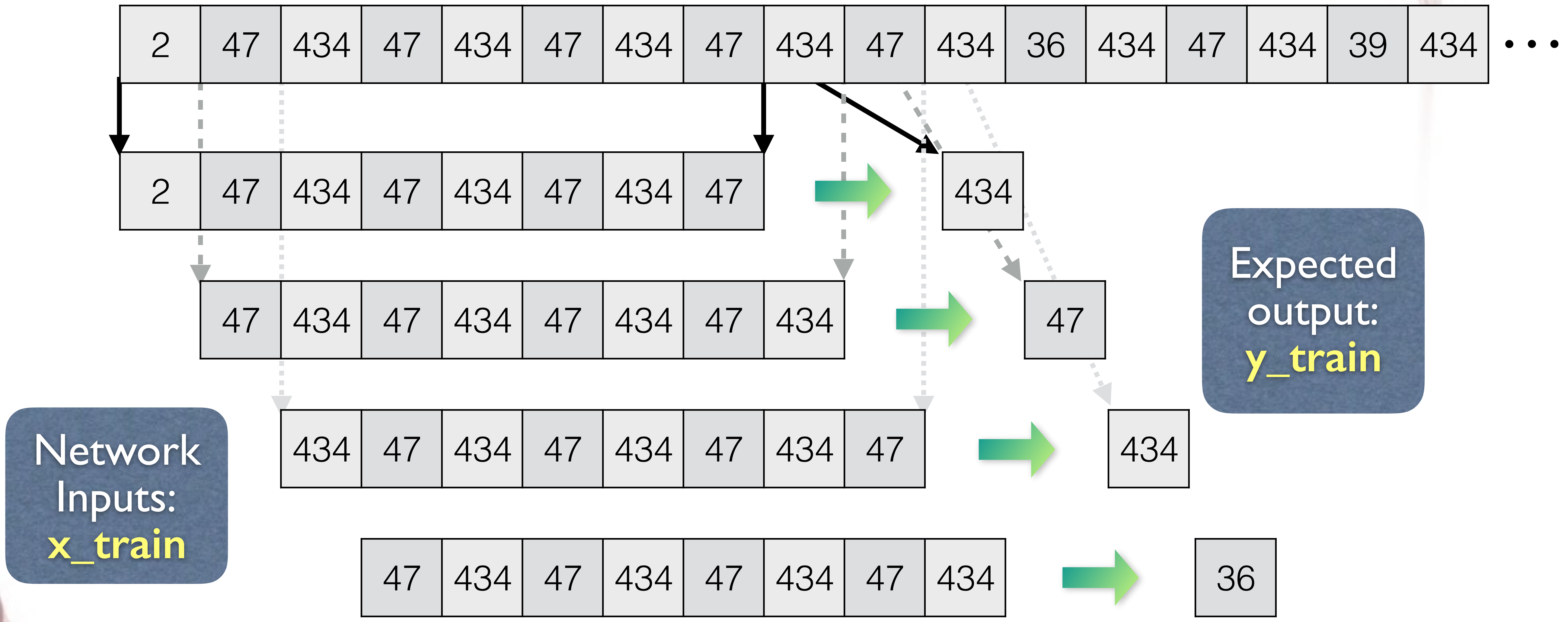
```
[33, 45, 61, 64, 69], [45, 49, 52], [57], [45, 49, 52], [57], [45, 49,
52], [57], [45, 49, 52], [57], [45, 49, 52], [57], [45, 49, 52, 61],
[57], [45, 49, 52], [57], [45, 49, 52, 64], [57], . . . . . .
```

Encoded data

2	47	434	47	434	47	434	47	434	47	434	36	434	47	434	39	434	• • •

INPUTS & EXPECTED OUTPUTS

- The key point is to let the network to **PREDICT** the upcoming note (chord) based on a sequence of input data. For example:



Put all together: unpack the data, create the dictionary, prepare training data, create LSTM model, and training…

```
length = 128
x_train, y_train = [], []
for idx in range(len(data)-length):
    sequence = data[idx:idx+length]
    next = data[idx+length]
    x_train.append([chords_to_idx[s] for s in sequence])
    y = np.zeros(n_chords)
    y[chords_to_idx[next]] = 1.
    y_train.append(y)
x_train, y_train = np.array(x_train), np.array(y_train)
from keras.layers import LSTM, Dropout, Dense
from keras.layers import Activation, Input, Embedding
from keras.models import Sequential, Model
model = Sequential()
model.add(Embedding(n_chords, 128, input_length=length))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(128))
model.add(Dropout(0.3))
model.add(Dense(n_chords))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
model.fit(x_train, y_train, epochs=200, batch_size=64)
model.save_weights('weights-ex04.h5')
```

Prepare x_train, y_train

"Embedding" layer for converting the input integers into dense vectors ⇐

Layers of LSTM

⇐ softmax + x-entropy

l304-example-04.py (partial)

TEST WITH A "SIMPLER" SONG

- Well, it turns out the Mozart concerto is rather difficult to train. Let's test the code with a simpler song, e.g. the **Prelude from the Final Fantasy** game series.

```
Total # of chords: 104    ⇐ simpler & shorter…
Total # of notes: 831
Epoch 1/200
703/703 [===============] - 20s 29ms/step - loss: 4.3
Epoch 2/200
703/703 [===============] - 15s 22ms/step - loss: 4.0379
Epoch 3/200
703/703 [===============] - 16s 22ms/step - loss: 3.9926
Epoch 4/200
. . . . . .
Epoch 199/200
703/703 [===============] - 16s 22ms/step - loss: 0.2531
Epoch 200/200
703/703 [===============] - 16s 22ms/step - loss: 0.2658
```

Here are the input MIDI…

MUSIC GENERATION

- Now let's try to use the trained model to generate some music!
- The key idea is to load the model (*instead of training*), and use a **random sequence as a "seed"** to feed into the network. Translate the network output back to the selected chord, and encode it back as a MIDI file. Done!

```
. . . . . .
model.load_weights('weights-ex04.h5')

x_test = np.array([np.random.randint(0,n_chords,length)])  ⇐ seed of the song
result = []
for seq in range(512):
    y_test = model.predict(x_test, verbose=0)[0]
    idx = np.argmax(y_test)  ⇐ let's pick up a chord based on the output
    result.append(idx_to_chords[idx])
    print('#%d: %s' % (seq,result[-1]))

    x_test[:,:-1] = x_test[:,1:]  ⇐ "rolling" the inputs
    x_test[:,-1] = idx

encode_midi('test.mid', result)
```

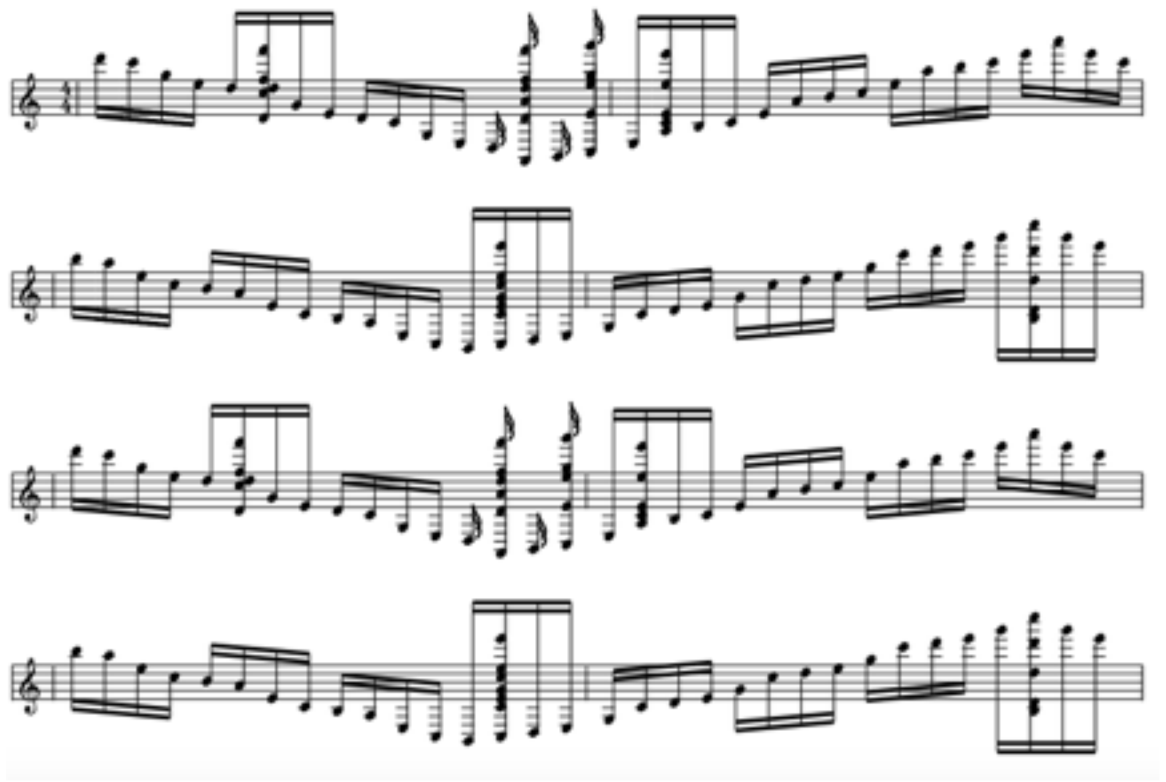
l304-example-04a.py (partial)

MUSIC GENERATION (II)

- This is what we can get:

```
#0: [86]
#1: [84]
#2: [79]
#3: [76]
#4: [74]
#5: [62, 72, 74, 77, 89]
#6: [67]
#7: [64]
#8: [62]
#9: [60]
#10: [55]
#11: [52]
#12: [50]
#13: [45, 45, 62, 69, 74, 77, 89]
#14: [47]
#15: [48, 64, 76, 79, 91]
#16: [52]
#17: [57, 60, 60, 64, 76, 88]
#18: [59]
#19: [60]
#20: [64]
. . . . . .
```

but it sounds just like repeating the input song...



Generated music w/ obvious structure!

COMMENT

- This test clearly shows the capability of RNN/LSTM, which can **"remember"** a given time-sequence data!
- But obviously, by training the network with only one song, it simply 100% remember the tune and repeat it as output — typical overtraining.
- Another problem is the selected song has a very distinct structure. When we just pick up the chord with highest score (this algorithm is usually called as **"greed search"**), it simply loops over the same tune. Not very optimal for music generation which requires some "variation" effect.
- Let's improve the whole situation by switch back to our dear Mozart concertos…

Simply include more songs, and a different way of music generation!

INCLUDE MULTIPLE SONGS AT ONES…

- Let's include all **Mozart violin concerto No. 3/4/5** times 3 movements into the pool!
- It is simple to add more MIDI files, but it also become very complicated (*too many different chords*) in the end.
- To be simplified (*as for this lecture*), we only take the **highest two notes** from each chord to reduce the combinations. This also gives a higher chance to "mix" the training data.

```
sources = ['mozk216a.mid','mozk216b.mid','mozk216c.mid',
           'mozk218a.mid','mozk218b.mid','mozk218c.mid',
           'mozk219a.mid','mozk219b.mid','mozk219c.mid']
all_data = []
for src in sources:
    data = decode_midi(src, 2)⇐ only keep the highest 2 nodes
    all_data.append(data)

all_chords = sorted(set([s for data in all_data for s in data]))
. . . . . .
```

1304-example-05.py (partial)

INCLUDE MULTIPLE SONGS AT ONES (II)…

- Surely, we also need a larger network to have better trained performance, given the complicity of the input data…

```
. . . . . .
for data_idx, data in enumerate(all_data):   ⇐ loop over 9 input MIDI files
    print('Song',data_idx,'- # of notes:',len(data))
    for idx in range(len(data)-length):
        sequence = data[idx:idx+length]
        next = data[idx+length]
. . . . . .
x_train, y_train = np.array(x_train), np.array(y_train)
print('Total # of training samples:',len(x_train))
. . . . . .
model.add(LSTM(256, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(256, return_sequences=True))
model.add(Dropout(0.3))                        ⇐ enlarged network
model.add(LSTM(256))
. . . . . .

model.fit(x_train, y_train, epochs=150, batch_size=64)
model.save_weights('weights-ex05.h5')
```

1304-example-05.py (partial)

This training will take a lot of time! You may want to get my trained weight file and skip this. It took me 48 hours on 12 CPUs…

MUSIC GENERATION (III)

- In order to avoid repeating/looping, instead of the greed search, here we just introduce a "temperature-controlled" **random search**.

```
. . . . . .
x_test = np.array([np.random.randint(0,n_chords,length)])
result = []
temperature=0.5
for seq in range(512):
    y_test = model.predict(x_test, verbose=0)[0]

    repeats = [np.all(x_test[:,-n:]==x_test[:,-n*2:-n]) for n in [2,3,4]]
    if np.any(repeats): temperature *= 1.15
    else: temperature *= 0.95
    temperature = min(max(temperature, 0.2),5.0)

    y_test = y_test**(1./temperature)
    idx = np.random.choice(range(n_chords),p=y_test/y_test.sum())
    result.append(idx_to_chords[idx])
    print('#%d: %s' % (seq,result[-1]))

    x_test[:,:-1] = x_test[:,1:]
    x_test[:,-1] = idx

encode_midi('test.mid', result, 375000)
```

my own test code to raise the temperature if there are too many repeating/looping notes.

we are using the "probability" interpretation of the softmax function + rescaling by the temperature

l304-example-05a.py (partial)

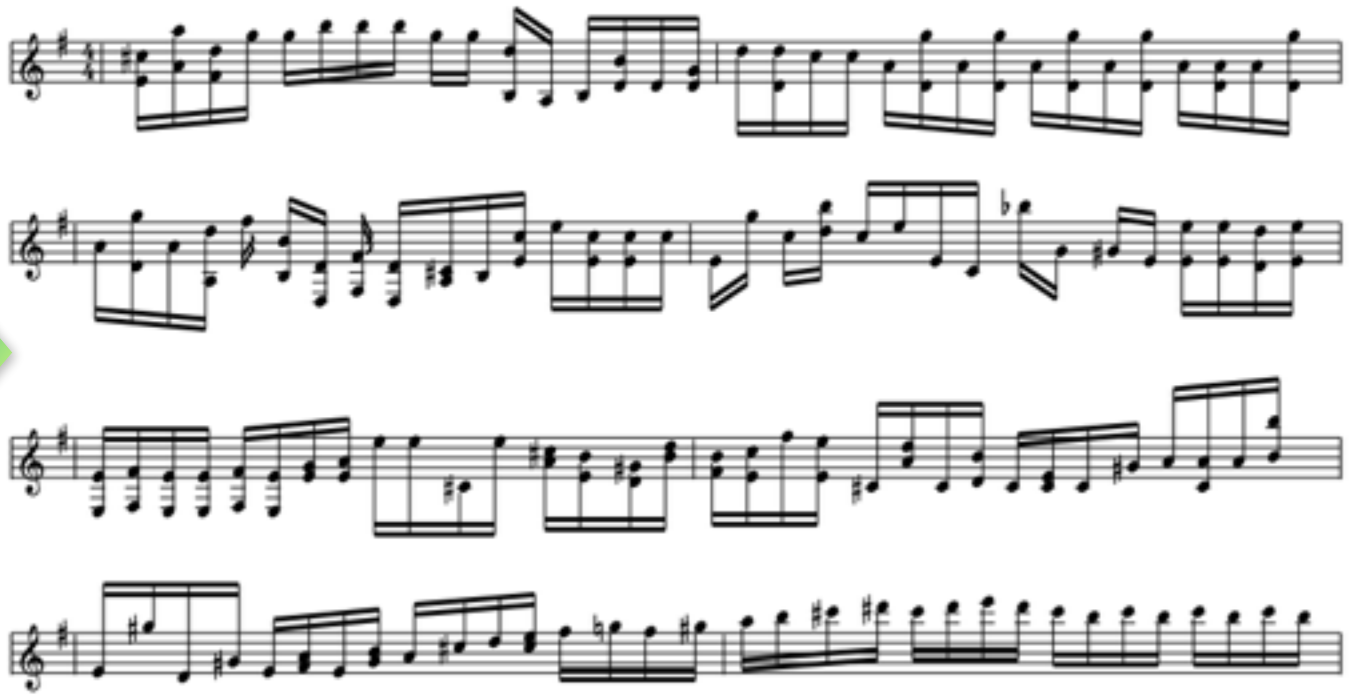
MUSIC GENERATION (IV)

- This is what we can get:

```
#0: [64, 73], T=0.47
#1: [69, 81], T=0.45
#2: [66, 74], T=0.43
#3: [79], T=0.41
#4: [79], T=0.39
#5: [83], T=0.37
#6: [83], T=0.35
#7: [83], T=0.33
#8: [79], T=0.32
#9: [79], T=0.30
#10: [59, 74], T=0.28
#11: [57], T=0.27
#12: [59], T=0.26
#13: [62, 71], T=0.24
#14: [62], T=0.23
#15: [62, 67], T=0.22
#16: [74], T=0.21
#17: [62, 74], T=0.20
#18: [72], T=0.20
#19: [72], T=0.20
#20: [69], T=0.20
. . . . . . .
```

It sounds not too bad?
But obvious not-so-Mozart!

Trial #1　　Trial #2



COMMMENT: MUSIC GENERATION W/ RNN

- Generating the music with RNN is kind of fun!
- But surely we still have a lot of room for improvement —
  - We shall not drop the rhythm!
  - One shall separate tune generation and chord matching! Otherwise we are only generating the notes that have been used by Mozart…
  - Better selected data, better trained model, etc…
- Leave all these points for your own study. Or you can check out the projects which has been developed so far:
  - Magenta (*this is the actual project behind the "Bach doodle"*): https://magenta.tensorflow.org
  - AIVA (*this is a commercial product*): https://www.aiva.ai

INTERMISSION

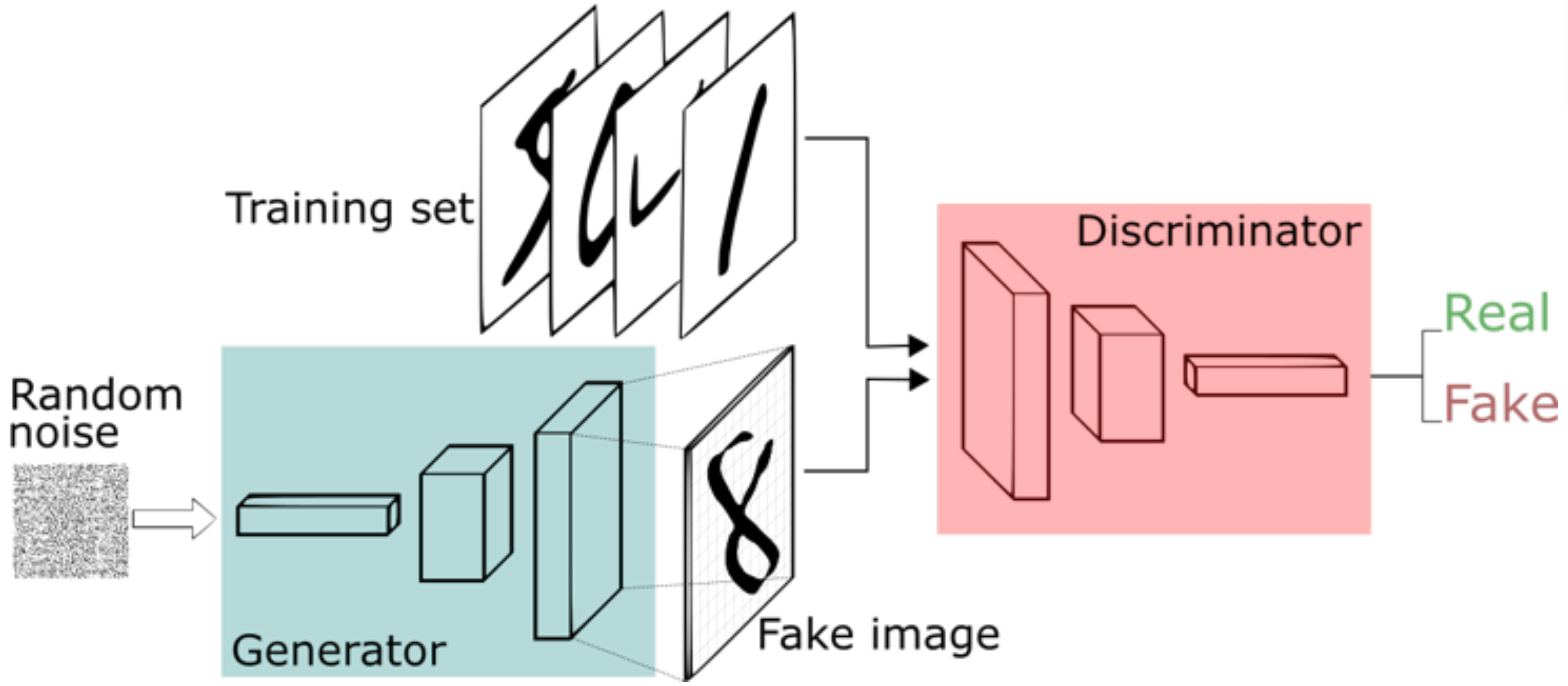
- You may want to change the generation rules (**greed search**, **random search**) in l304-example-04a.py and l304-example-05a.py and see if you are able to come up with a different tune?
  - There is another commonly introduced "**beam search**", you can try to implement one!
- Surely, by replacing the training music data, the situation will change dramatically. You may try to replace the input with your own favorite song and see if you are able to come up with something different?

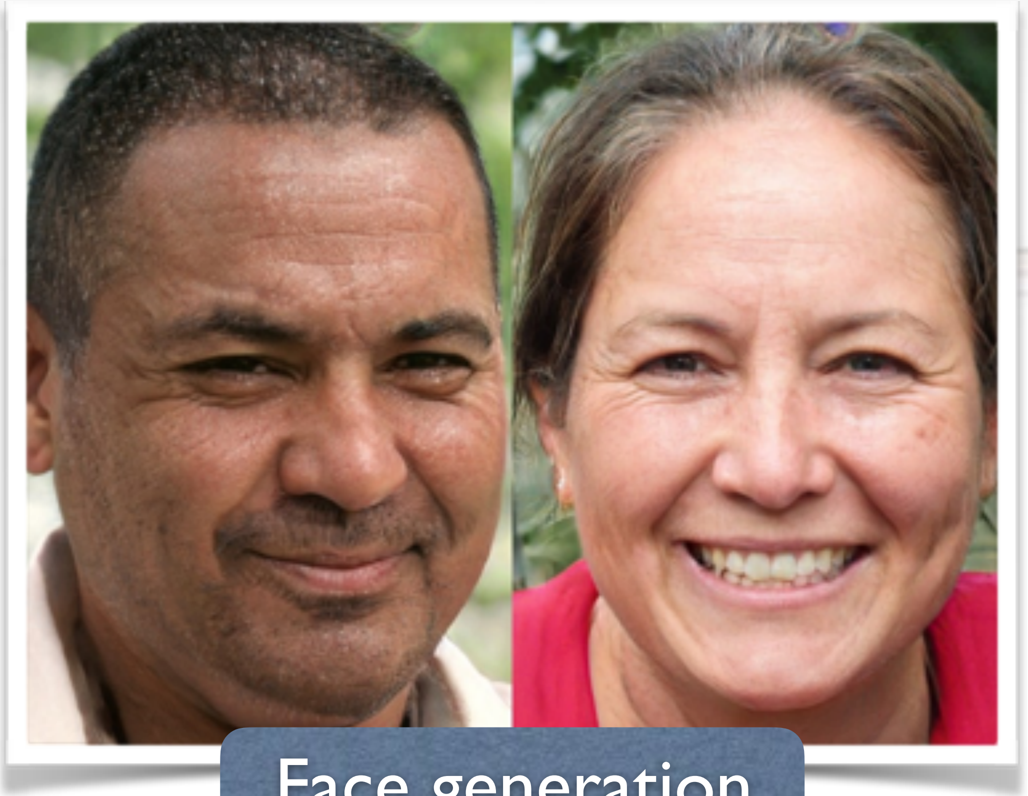
GENERATIVE ADVERSARIAL NETWORK

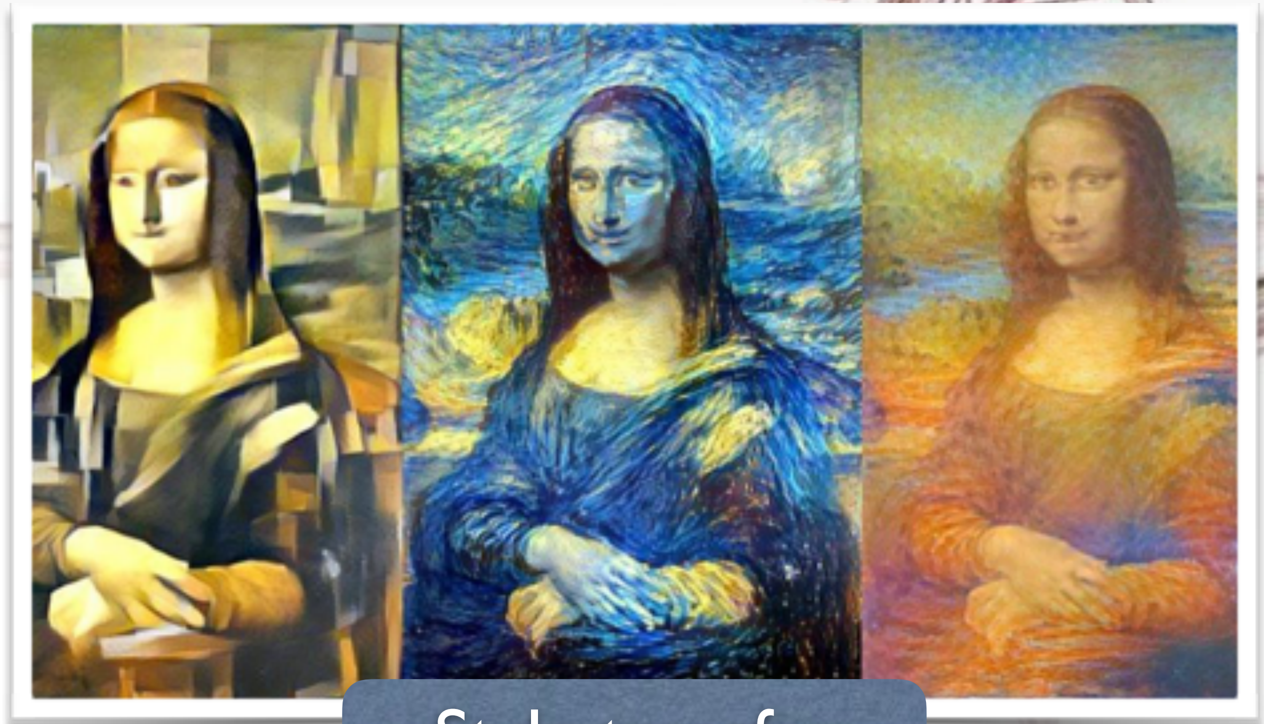
- The name "GAN", or the **Generative Adversarial Network**, was first introduced by Ian Goodfellow in 2014. It is a very interesting idea and became extremely popular in recent years.
- As we already slightly mentioned earlier, the key setup is to have two networks training against each other:
  - **discriminative 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!

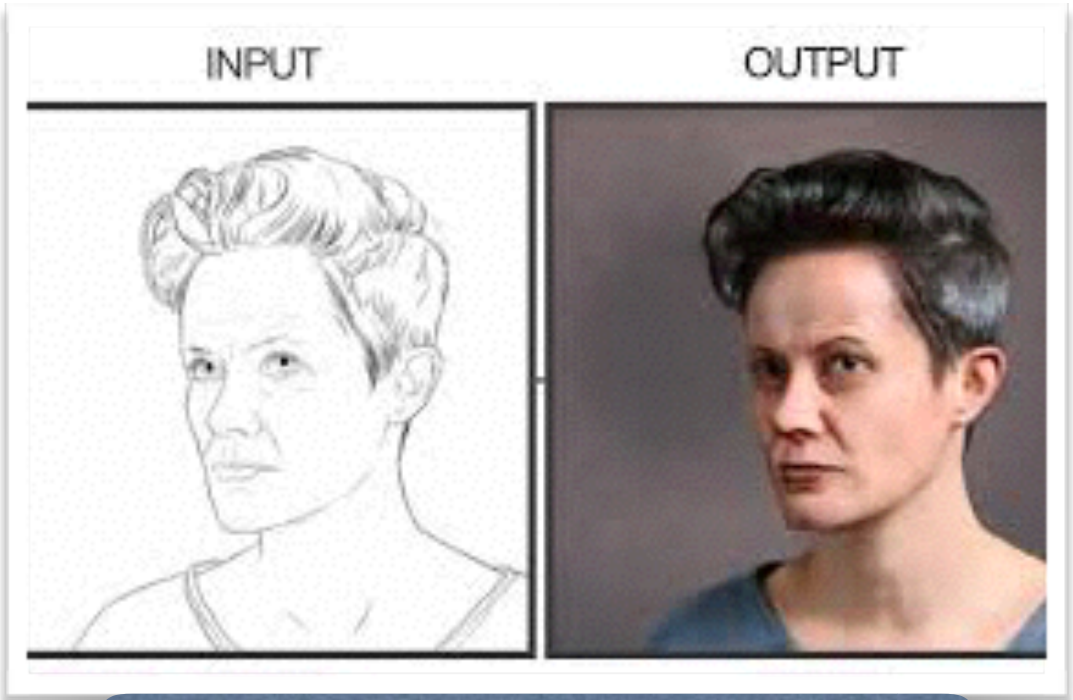
GENERATIVE ADVERSARIAL NETWORK (II)

- 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!




Face generation


Style transfer



Convert doodle to photo

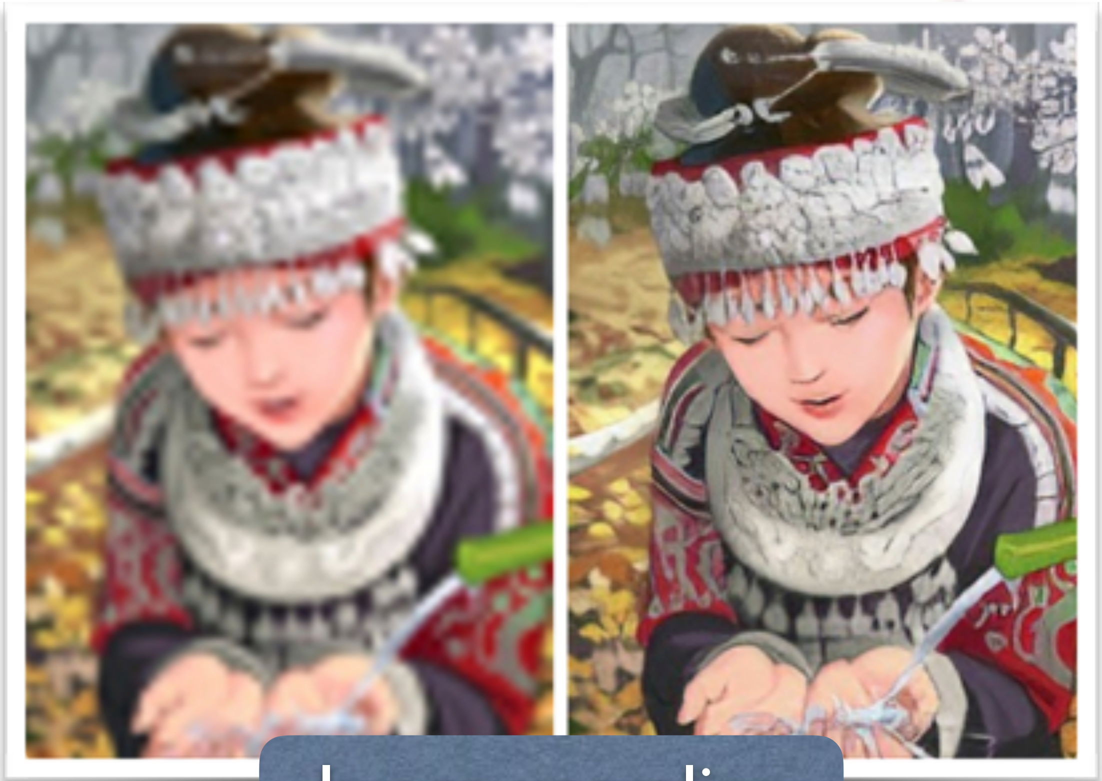


Image upscaling

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

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!

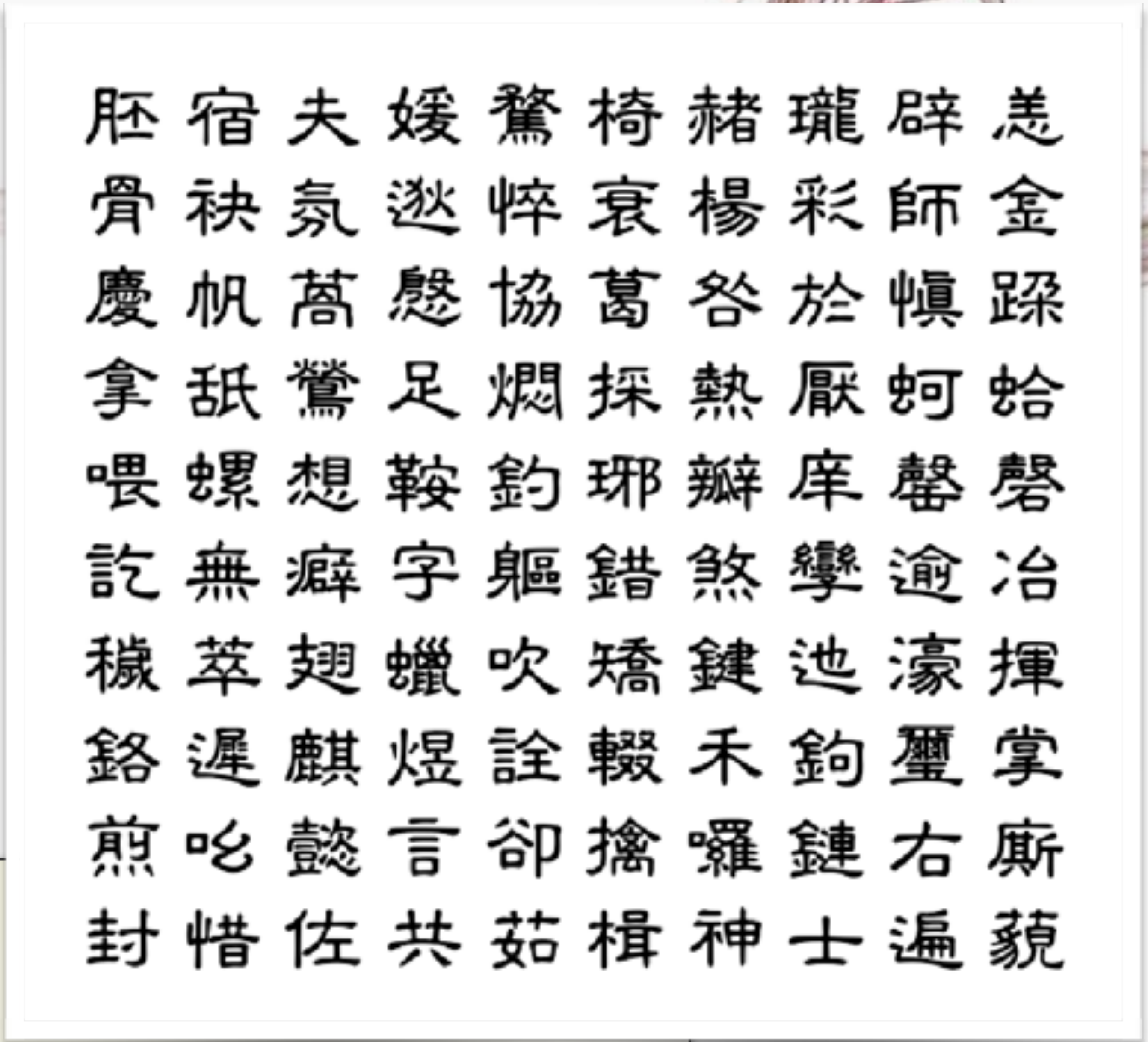
How about an AI Cangjie?

FONT DATA

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

```python
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()
```

l304-example-06.py



Yes, these are **clerical scripts**!

CONSTRUCT A VANILLA GAN

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

```
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 keras.layers import Input, Dense, Reshape
from keras.layers import BatchNormalization, LeakyReLU
from keras.models import Sequential, Model
from 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(Dense(256))
discriminator.add(LeakyReLU())
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 nodes

l304-example-07.py (partial)

CONSTRUCT A VANILLA GAN (II)

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

```
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

```
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)

l304-example-07.py (partial)

CONSTRUCT A VANILLA GAN (III)

- Manual training steps: ask the discriminator to separate real / fake images; ask the generator to generate cheat the discriminator.

```
. . . . . .
batch_size = 32
for epoch in range(20001):
  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)
  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
  noise = np.random.randn(batch_size, latent_size)
  gen_loss = combined.train_on_batch(noise, np.ones((batch_size,1)))
  
  print("Epoch: %d, discriminator(loss: %.3f, acc.: %.2f%%), generator(loss: %.3f)" %
        (epoch, dis_loss[0], dis_loss[1]*100., gen_loss))
```

real images from input data;
fake images from generator

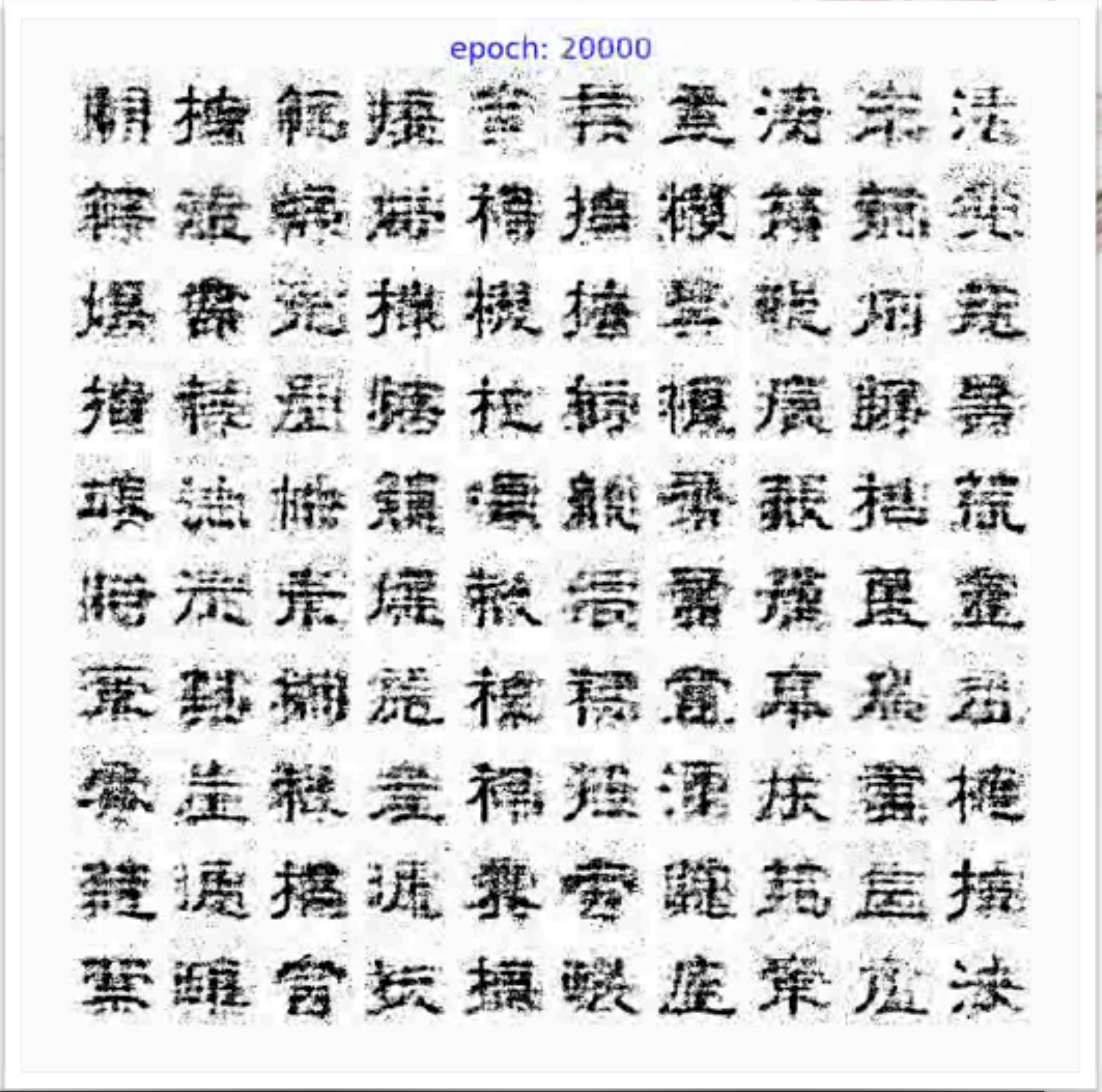
Training the discriminator
with real & fake images

training generator

l304-example-07.py (partial)

RESULTS OF TRAINING

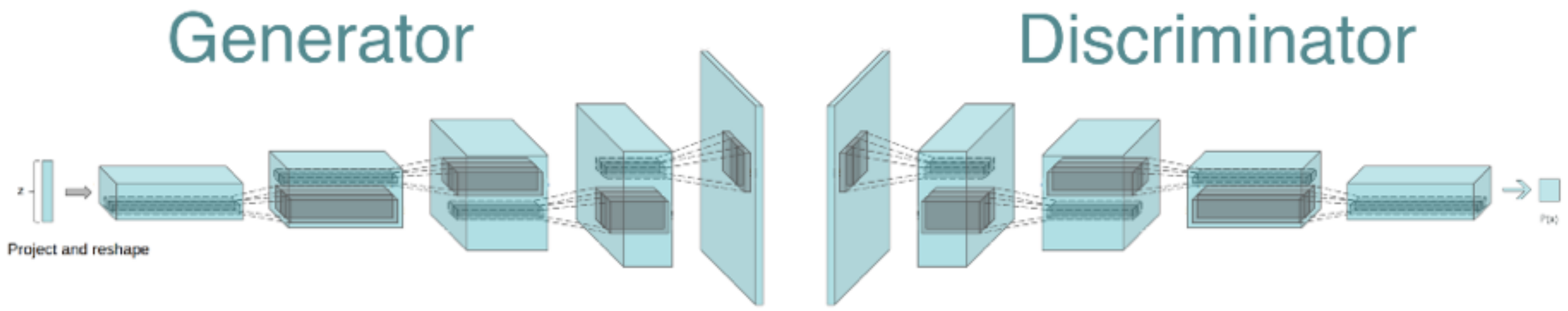
- It does generate some images which may "look like" Chinese characters (*although one has to read them from a long distance*).
- **Surely none of them is really readable!**



```
Epoch: 0, discriminator(loss: 0.710, acc.: 39.06%), generator(loss: 0.720)
......
Epoch: 100, discriminator(loss: 0.012, acc.: 100.00%), generator(loss: 4.221)
......
Epoch: 15000, discriminator(loss: 0.102, acc.: 96.88%), generator(loss: 4.796)
......
Epoch: 20000, discriminator(loss: 0.111, acc.: 96.88%), generator(loss: 5.076)
```

GAN+CNN = DCGAN

- Well, 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

- Need to replace the **discriminator**:

```
. . . . . . .
from keras.layers import Input, Dense, Reshape
from keras.layers import BatchNormalization, LeakyReLU
from keras.layers import Conv2D, Flatten, UpSampling2D
from keras.models import Sequential, Model
from keras.optimizers import Adam

discriminator = Sequential()
discriminator.add(Reshape(img_shape+(1,), input_shape=img_shape))
discriminator.add(Conv2D(32, kernel_size=6, strides=2))
discriminator.add(LeakyReLU())
discriminator.add(Conv2D(64, kernel_size=4, strides=2))
discriminator.add(BatchNormalization())
discriminator.add(LeakyReLU())
discriminator.add(Conv2D(128, kernel_size=4, strides=1))
discriminator.add(BatchNormalization())
discriminator.add(LeakyReLU())
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)×3 ⇒ 1 binary node

l304-example-08.py (partial)

CONSTRUCT A DCGAN (II)

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

```
generator = Sequential()
generator.add(Dense(14*14*64, input_dim=latent_size,
activation='relu'))
generator.add(Reshape((14,14,64)))
generator.add(UpSampling2D())
generator.add(Conv2D(64, kernel_size=3, activation='relu'))
generator.add(BatchNormalization())
generator.add(UpSampling2D())
generator.add(Conv2D(64, kernel_size=3, activation='relu'))
generator.add(BatchNormalization())
generator.add(Conv2D(1, kernel_size=3, 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 ⇒ (up sampling⇒conv)×2 ⇒ conv ⇒ image

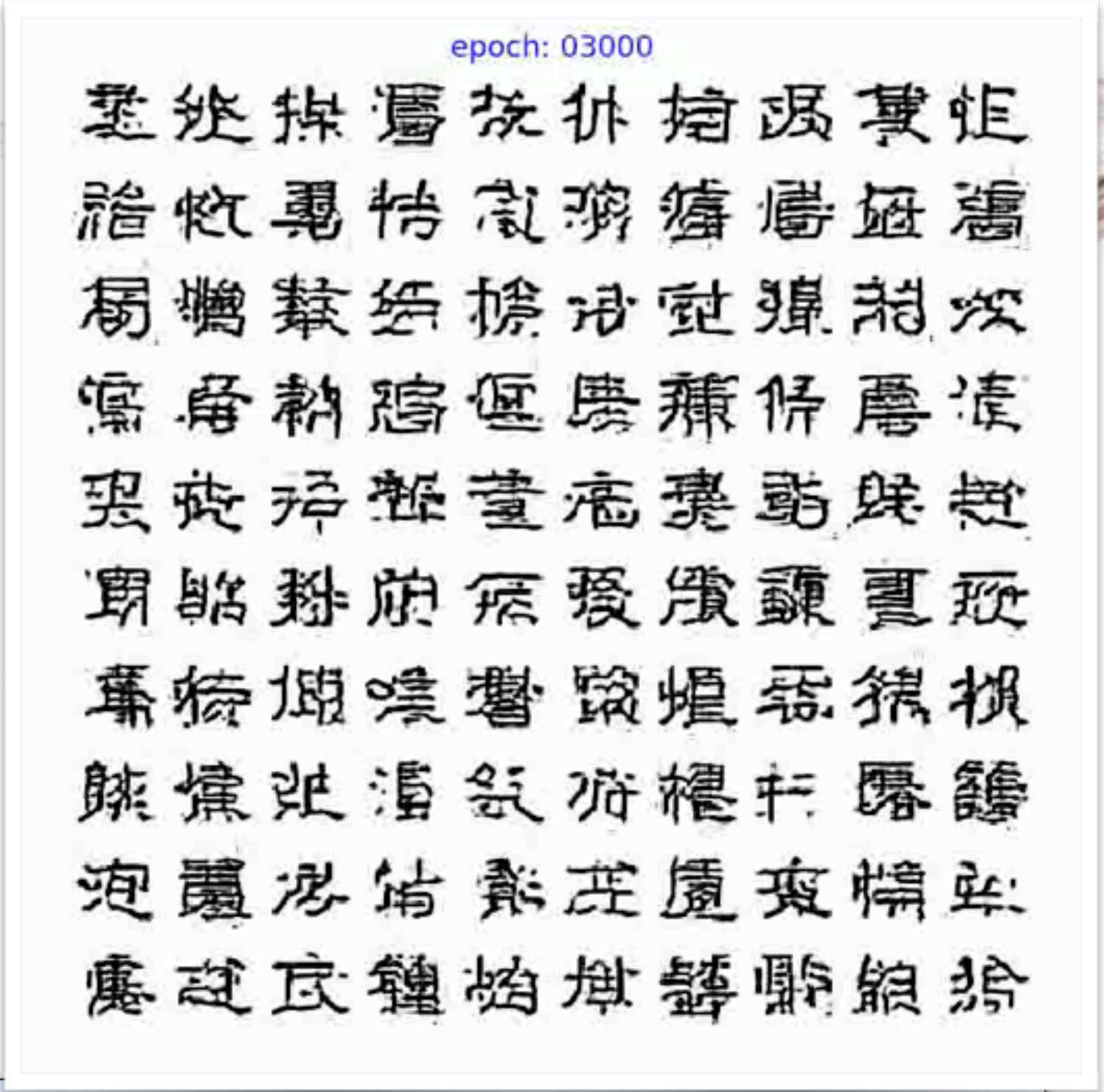
⇐ combined model is the same

l304-example-08.py (partial)

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: 0.932, acc.: 42.19%), generator(loss: 0.426)
......
Epoch: 200, discriminator(loss: 0.466, acc.: 85.16%), generator(loss: 2.104)
......
Epoch: 2000, discriminator(loss: 0.086, acc.: 99.22%), generator(loss: 3.668)
......
Epoch: 3000, discriminator(loss: 0.092, acc.: 100.00%), generator(loss: 4.371)
```

COMMMENT

- 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!

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

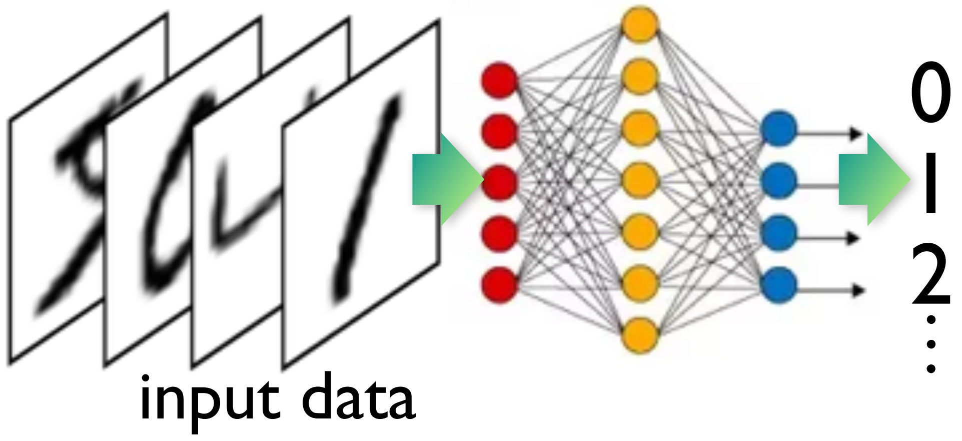
FINAL COMMMENT: PHYSICIST'S ML

- 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.

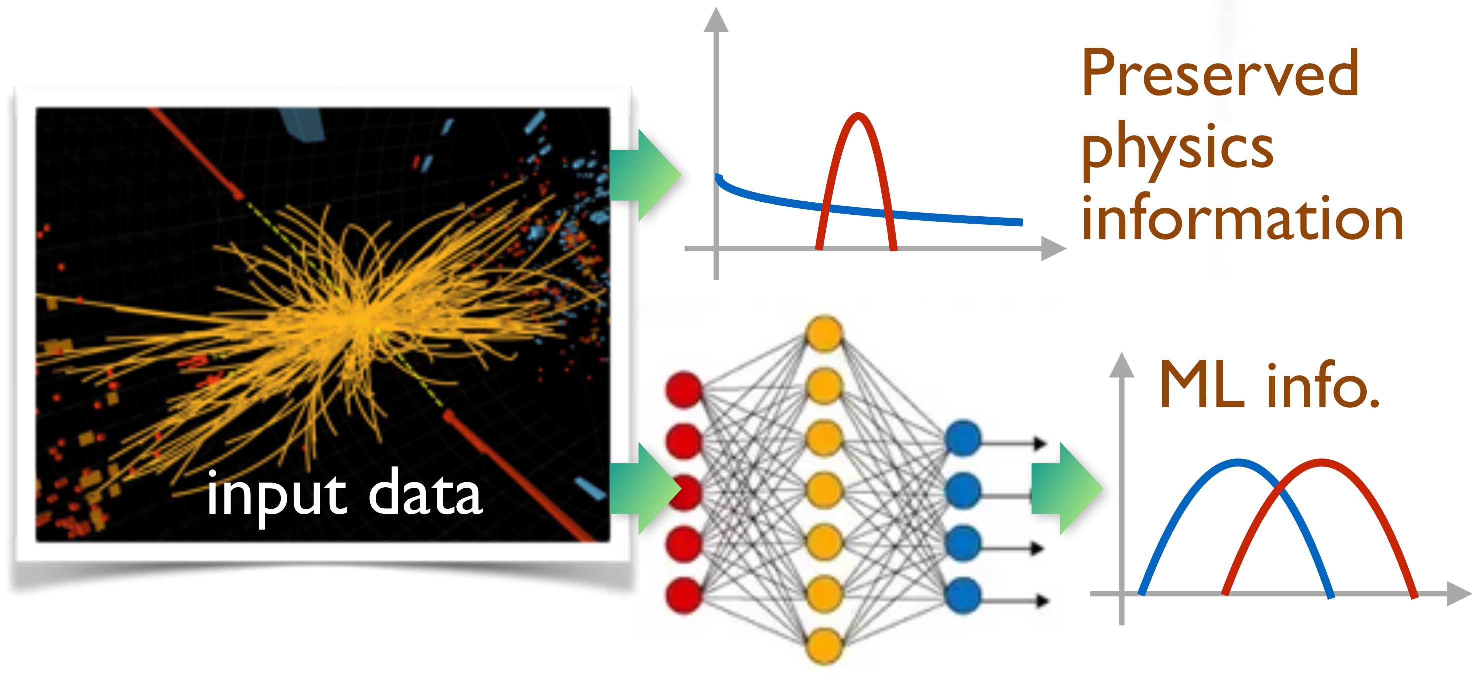
FINAL COMMMENT: PHYSICIST'S ML (II)

- 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

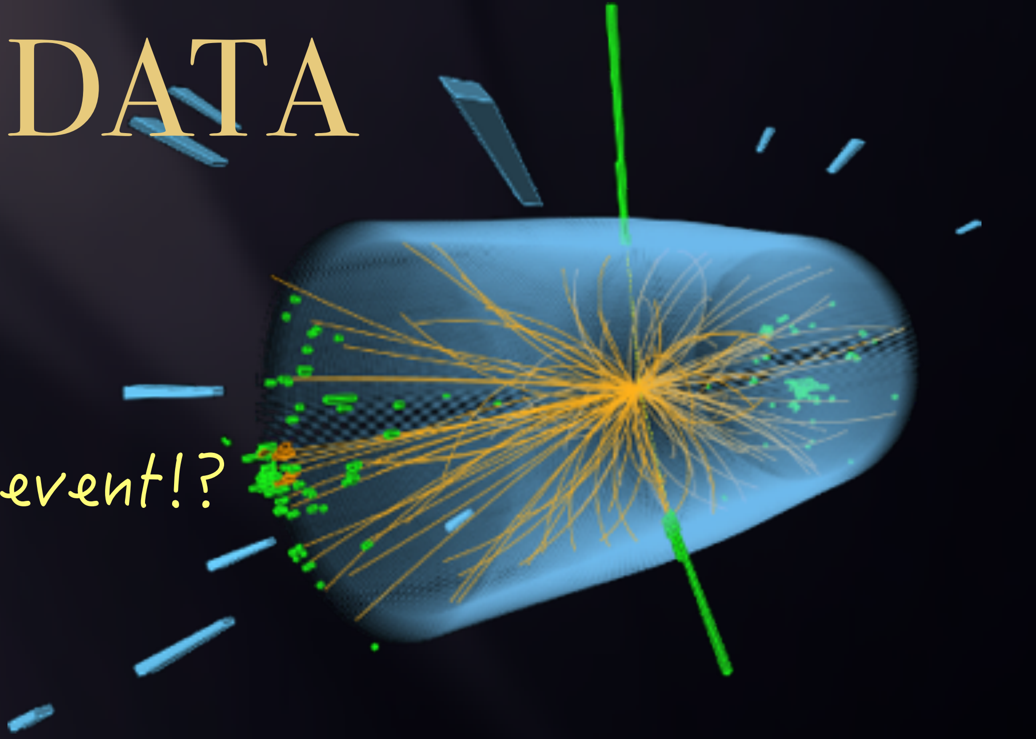
A nice kitty!

A cat maybe??

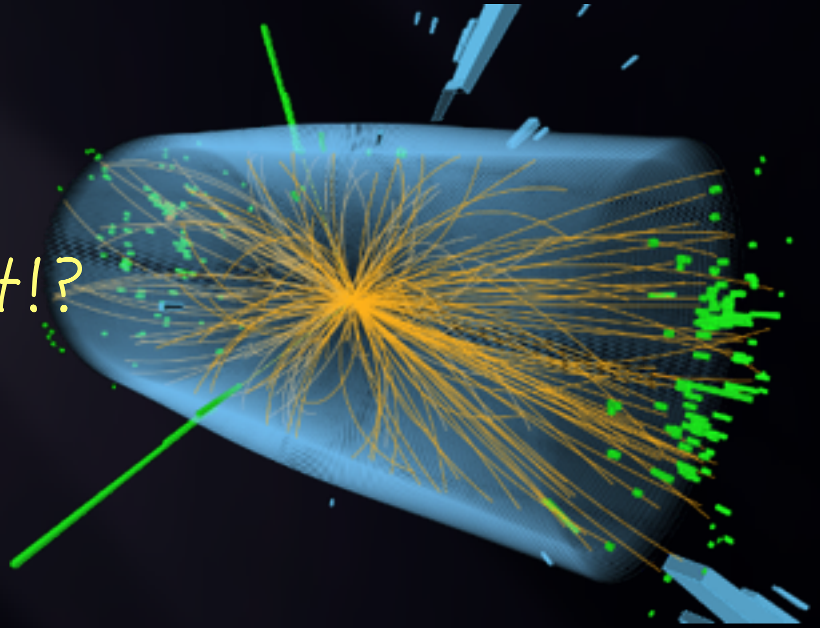
It's a toast, right?

One can tell by eyes quickly

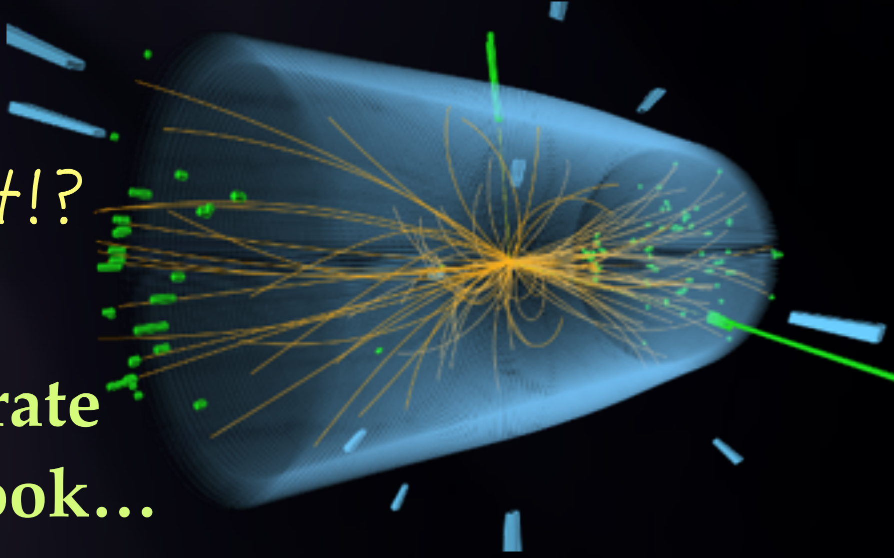
HEP DATA



A Higgs event!?



A Higgs event!?



A Higgs event!?

Cannot separate by the first look…

HANDS-ON SESSION

- **Practice data:**
  There is a data of 2 classes, stored in the `l304practice.npz` file (can be downloaded from CEIBA or the lecture web). The following piece of code can be used to load it:

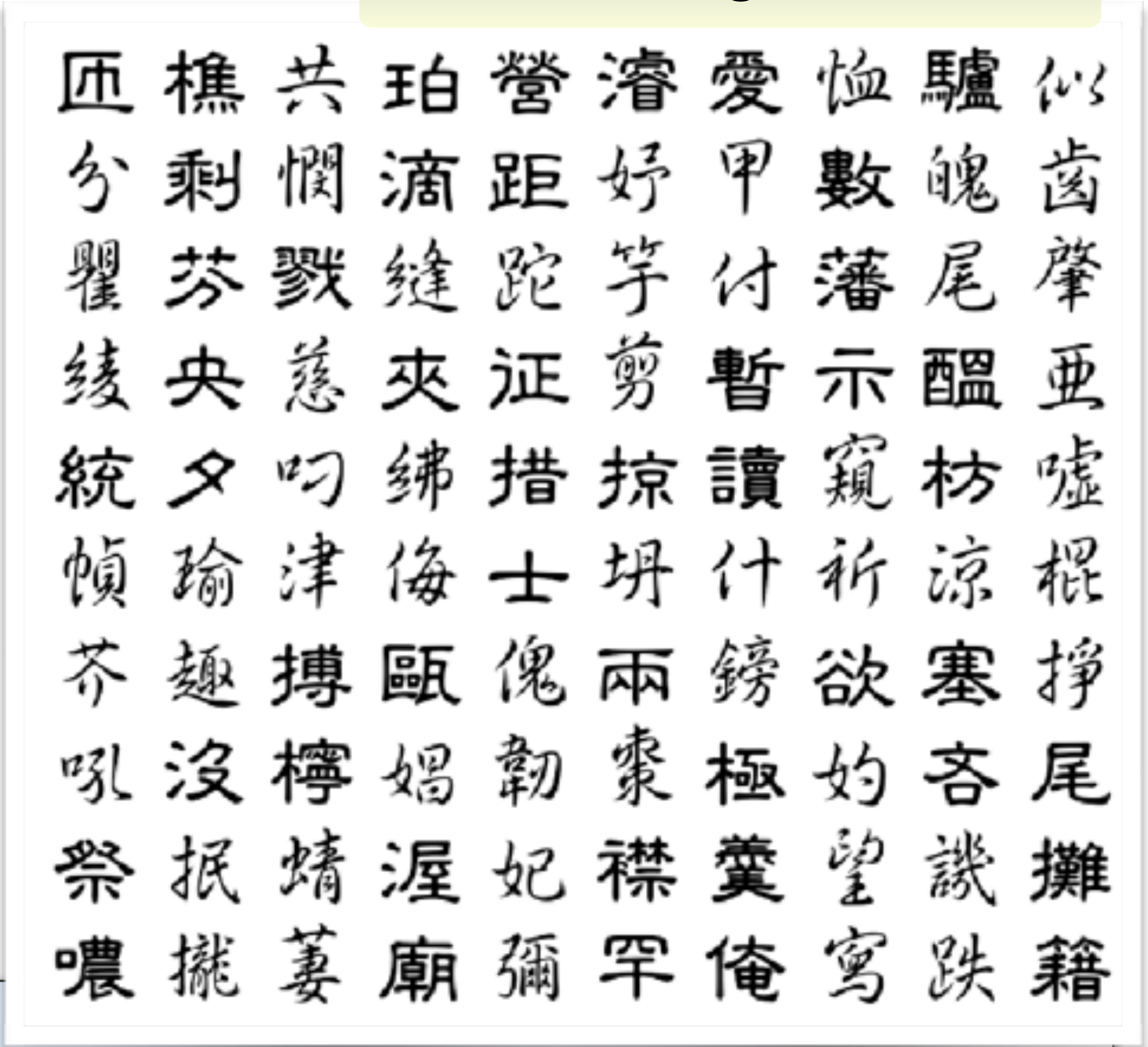
```python
import numpy as np

data = np.load('l304practice.npz')
x_train = data['x_train']
y_train = data['y_train']
x_test = data['x_test']
y_test = data['y_test']
```

  The `x_train, y_train` contains 6400 samples, and `x_test, y_test` contains 3216 samples.

HANDS-ON SESSION

- The **x_train** and **x_test** data contains the images (also 48×48) as a mixture of two different scripts of Chinese characters.
- The **y_train** and **y_test** data contains the label: **1 = clerical script**, **2 = semi-cursive script**.

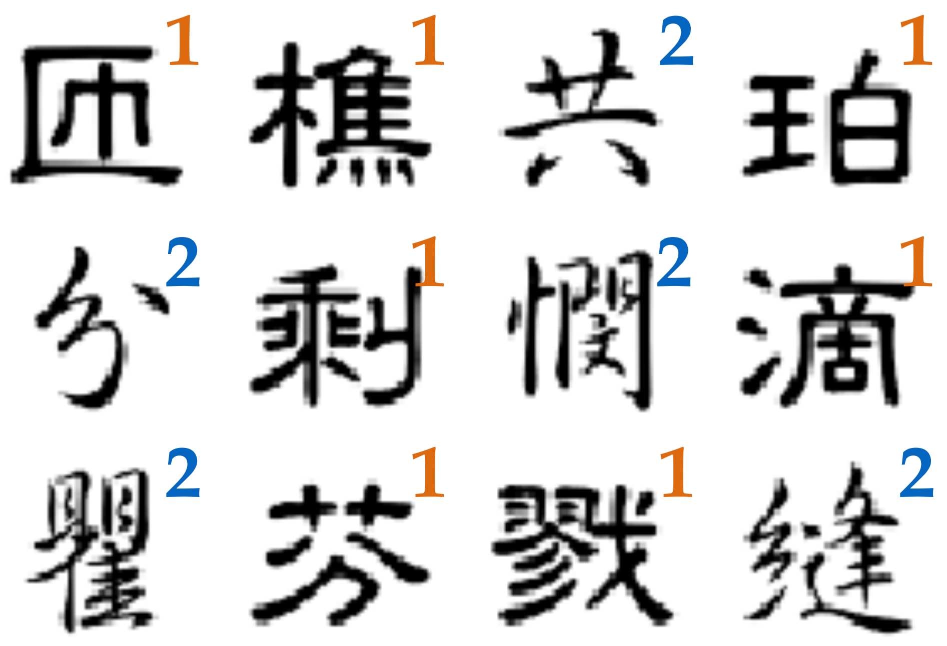
First 100 images in x_train



```
>>> print (y_train[:100])
[1 1 2 1 1 1 1 2 1 2 2 1 2 1 1 2 2 1 2 2 2 1 1 2 2 2 2 1 2 2 2 1 2 1 1 2 1
 1 1 2 1 1 2 2 1 1 1 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 1 1 2 1 2 1 1 2 2 1 1 2
 2 2 1 2 1 1 1 2 2 1 2 1 1 2 2 1 1 2 2 1 2 1 1 2 2 1]
```

HANDS-ON SESSION

- **Practice 01:**
  Take the `l304-example-01.py` (or `l304-example-01a.py`) as a template code, replace the MNIST data with the data we just provided, see if you can construct a CNN model to separate the two different scripts of Chinese characters?



```
Performance (training):
0.xxxxx
Performance (testing):
0.yyyyy
```

HANDS-ON SESSION

- **Practice 02:**
  Take the `l304-example-08.py` (or `l304-example-07.py`) as a template code, replace the input data with the **x_train** images, and ㄎ used to train a DCGAN(or GAN) model.
- See if we can come up with a new style of Chinese font by mixing clerical and semi-cursive scripts? Although we do not expect to generate any readable fonts…
- This will take a long time on your laptop, you may want to run it on a better PC or at least find a power plug first…

行書 + 隸書 =?