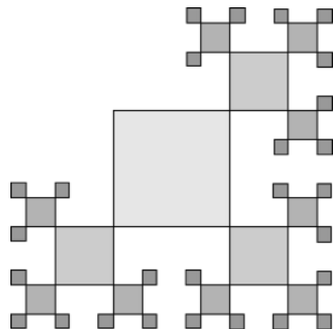
INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

NETWORK STRUCTURE DOES MATTER!

- As discussed in the previous module, 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 convolutional neural network (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.

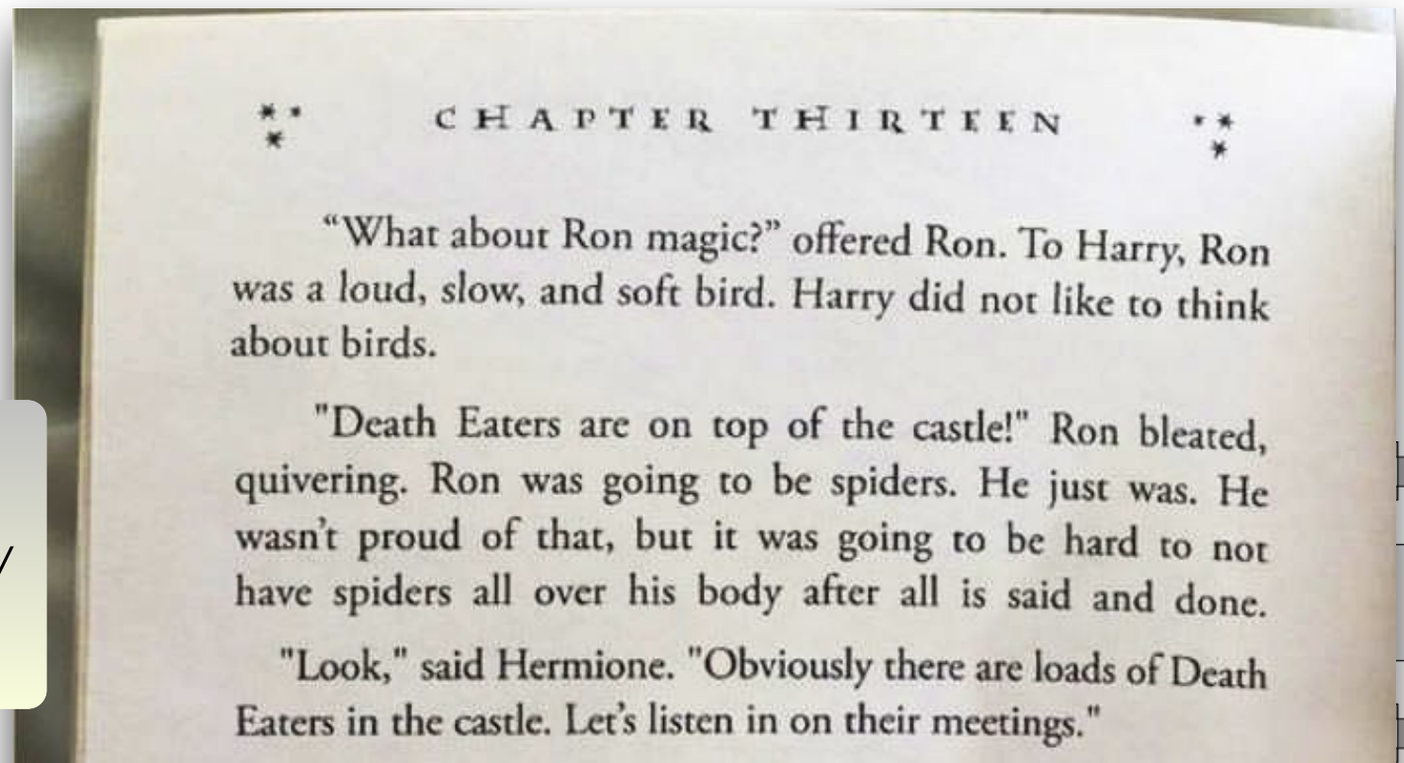
Let's quickly comment on some modern networks which has been developed for different topics!



OTHER POPULAR 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.

A Harry Potter chapter "written" by AI program…



CHAPTER THIRTEEN

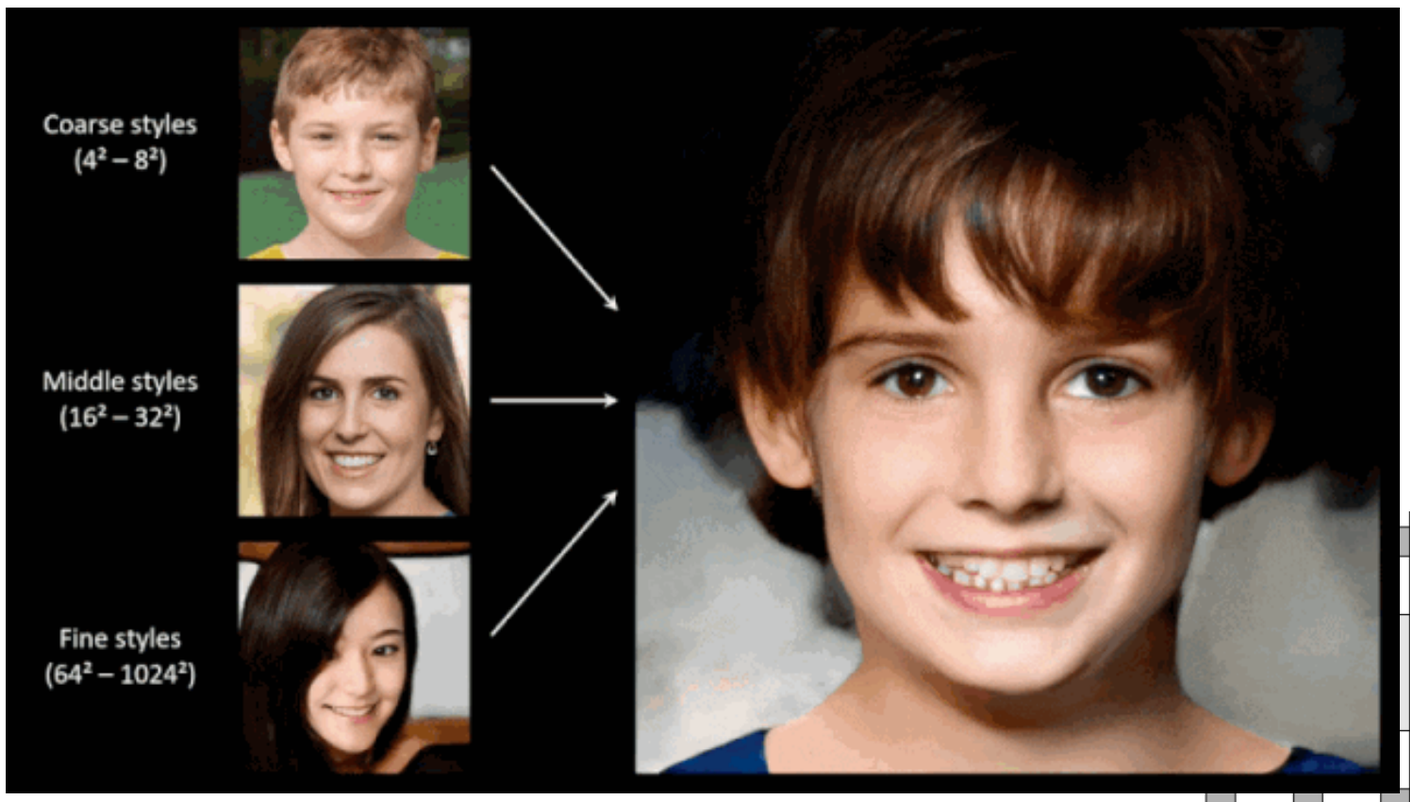
"What about Ron magic?" offered Ron. To Harry, Ron was a loud, slow, and soft bird. Harry did not like to think about birds.

"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 Eaters in the castle. Let's listen in on their meetings."

OTHER POPULAR DEEP NETWORKS & IDEAS (II)

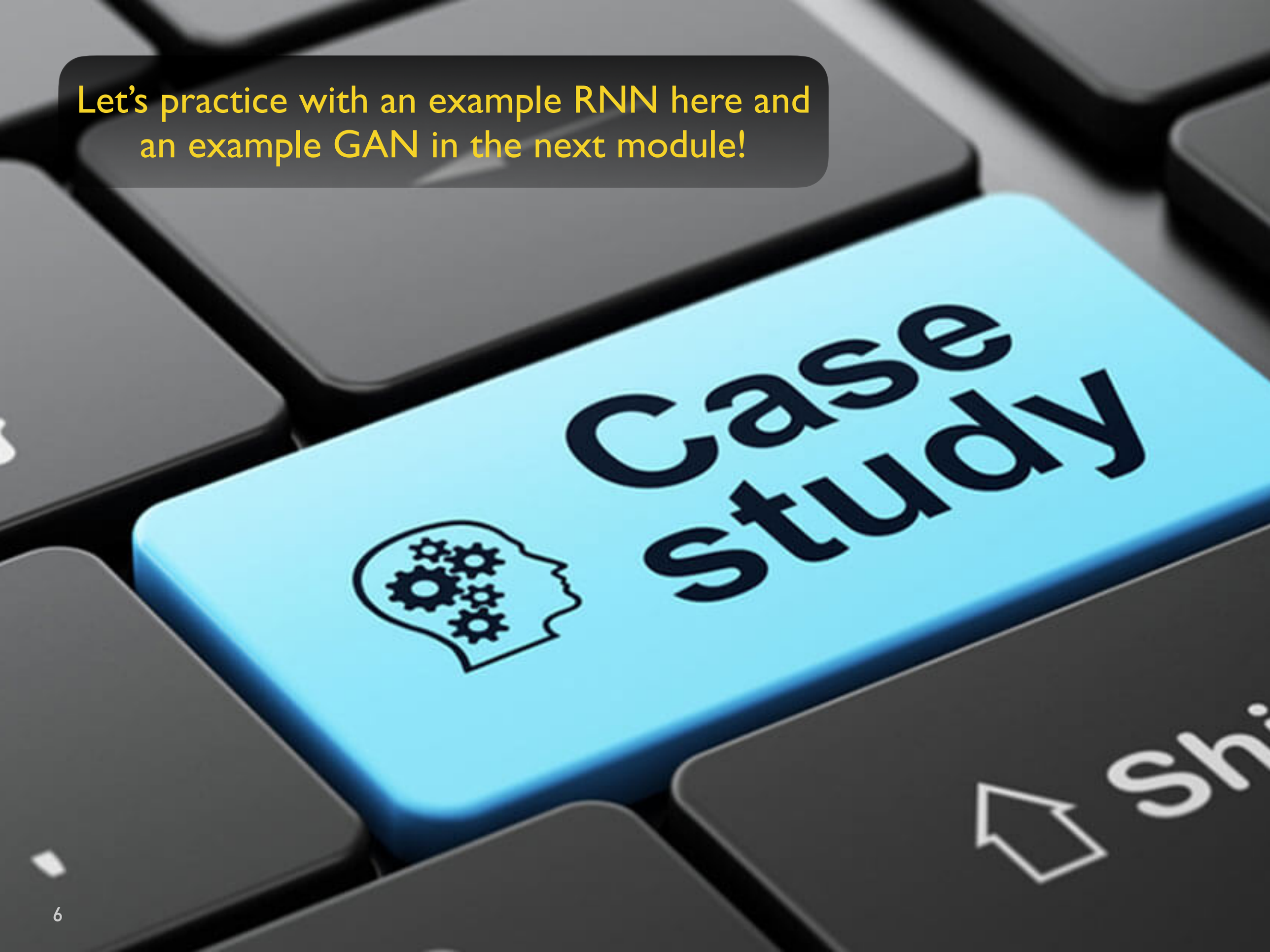
- **Generative adversarial network (GAN)**:
  - The basic structure of GAN is to have two network "fighting" with each other: one is to find "fake" images out of the pool, another one is to generate fake images.
  - Once it has been trained, you can use the generator to produce lots of "nearly true"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



OTHER POPULAR DEEP NETWORKS & IDEAS (III)

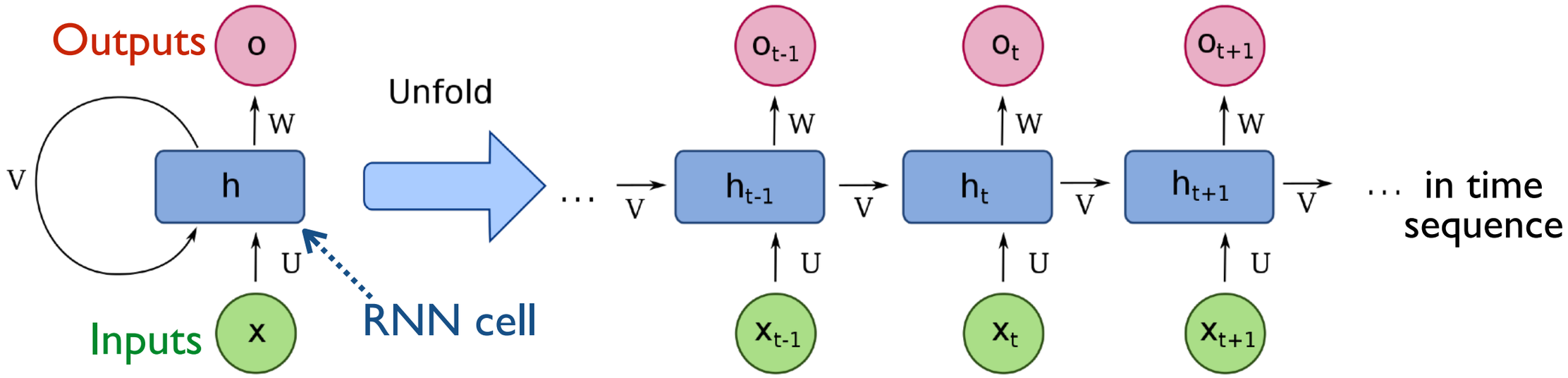
- **Reinforcement Learning (RL)**:
  - In our example network, the required responses of our model are relatively simple (just which digit, 0-9). But in many problems, for example, playing chess, this is not a simple task as no clear classification of good/bad labels.
  - Then the reinforcement learning 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!

Let's practice with an example RNN here and an example GAN in the next module!



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?

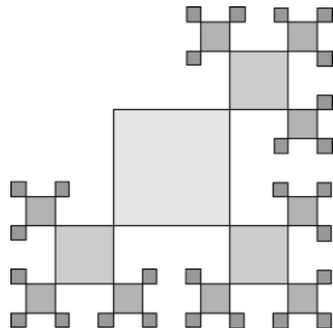
- Let's take an analogy, by reading/examination the following short story (*suppose you are using a NN to process an article*):

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

- If there is a 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:

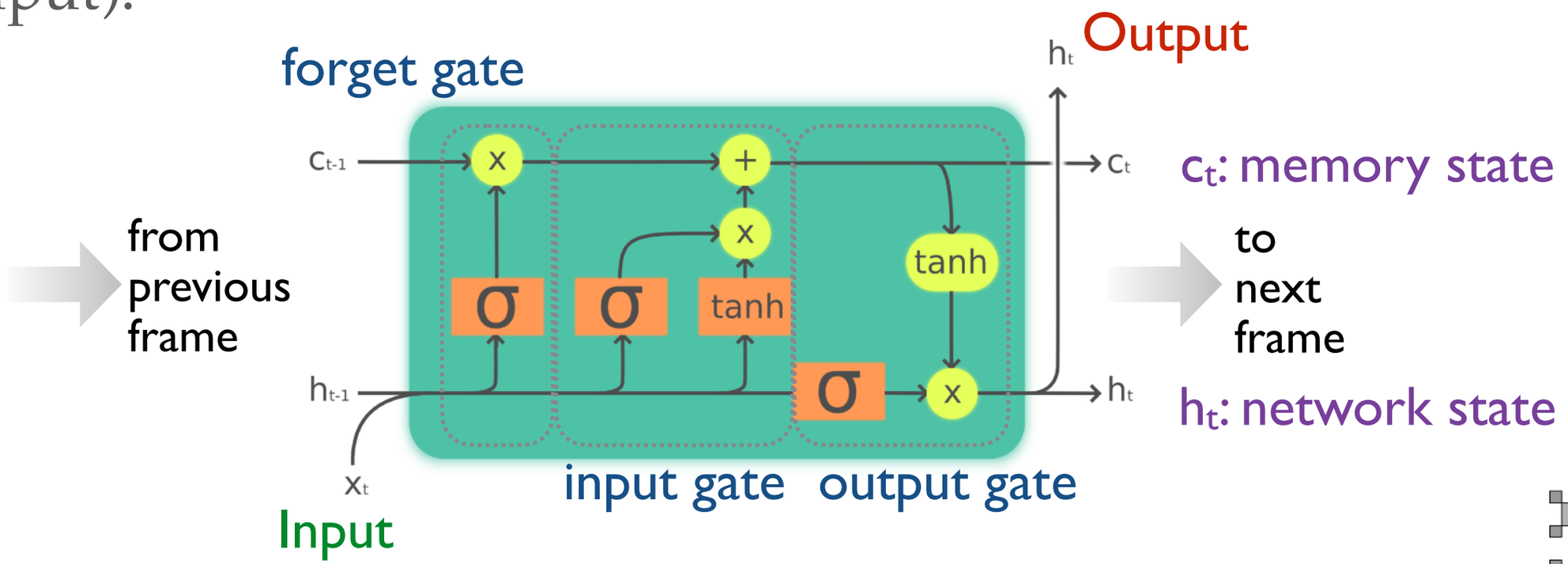
  June was born in France. (…a long story and blah-blah…) Surely, she can still speak nearly perfect French.

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).



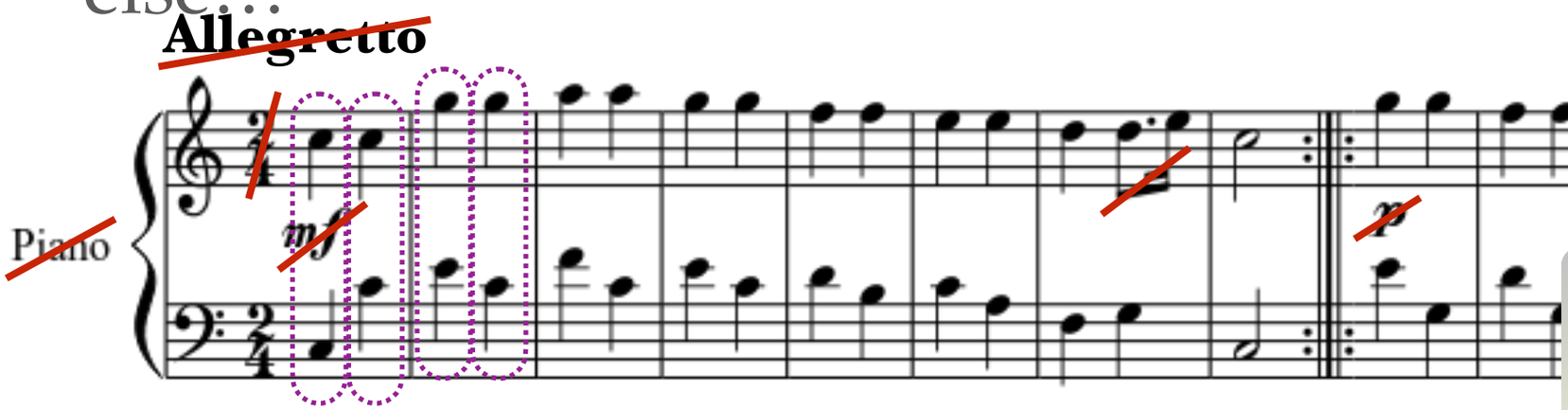
LONG SHORT-TERM MEMORY (II)

- 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".

partial **midi_phraser.py**

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

def decode_midi(filename, maxnotes = 0):
  mid_in, notes = MidiFile(filename),[]
  for track in mid_in.tracks: ← loop over "tracks"
    sum_of_ticks, pool = 0, []
    for msg in track: ← loop over "messages"
      sum_of_ticks += msg.time ← count "ticks"
      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
```

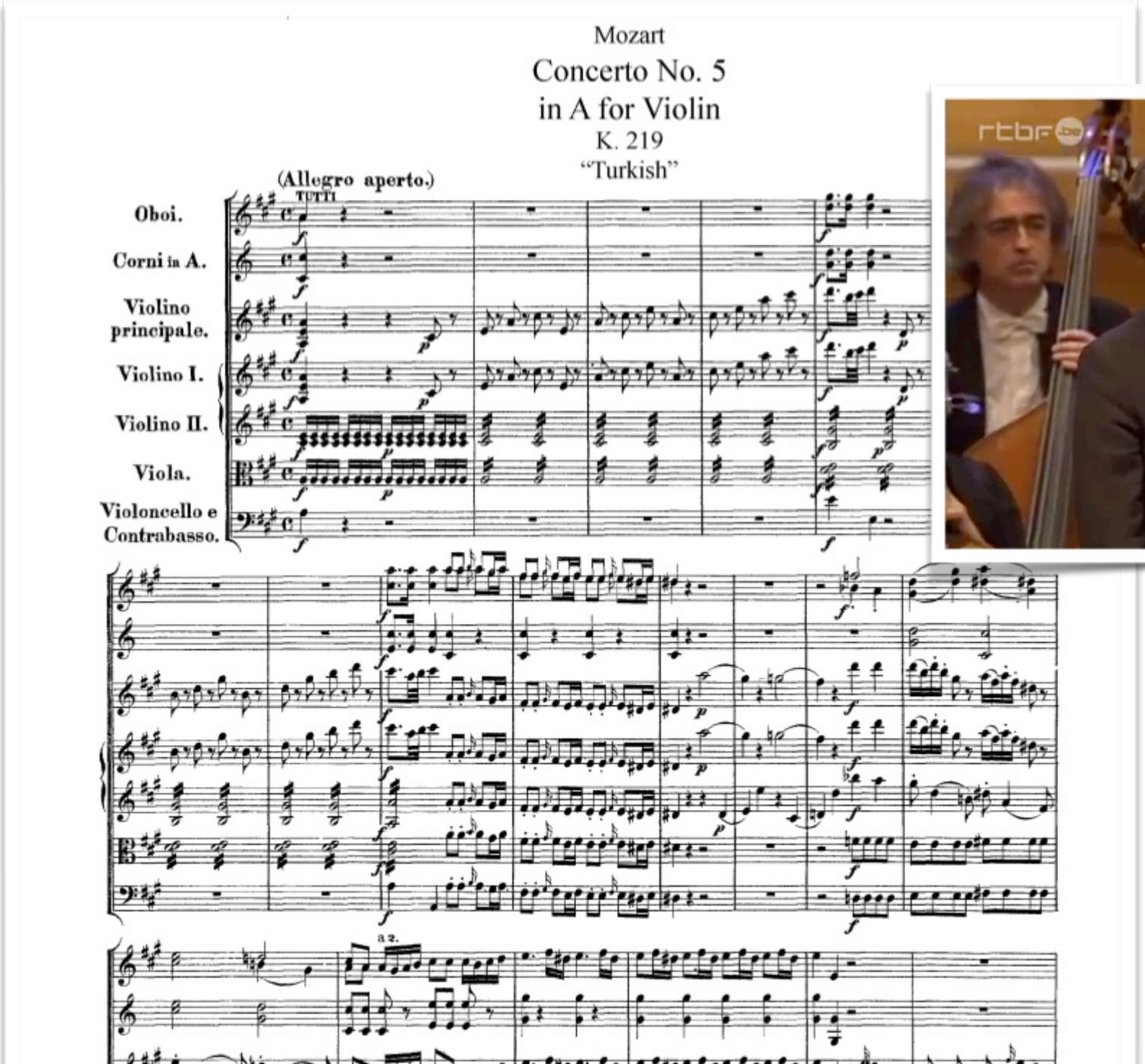
↑ interpret "note on/off" messages

↑ output the "chords" as a list of strings

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**.

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!*)
- The frequency for each pitch can be calculated by

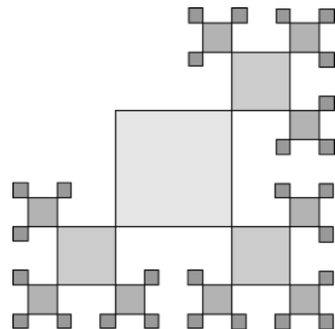
$$f_m = 2^{\frac{m-69}{12}} \times 440 \text{ Hz}$$

ex_ml9_1.py

```
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)
```

```
#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!



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?

(not too bad?)

- Let's simply **pack it back to a MIDI file** and check if the music still sounds like a Mozart concerto?

partial midi_phraser.py

```
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)
        delay = 120  ← no idea about the duration 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)
```

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:

partial ex_ml9_2.py

```
. . . . . .
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])
. . . . . .
```

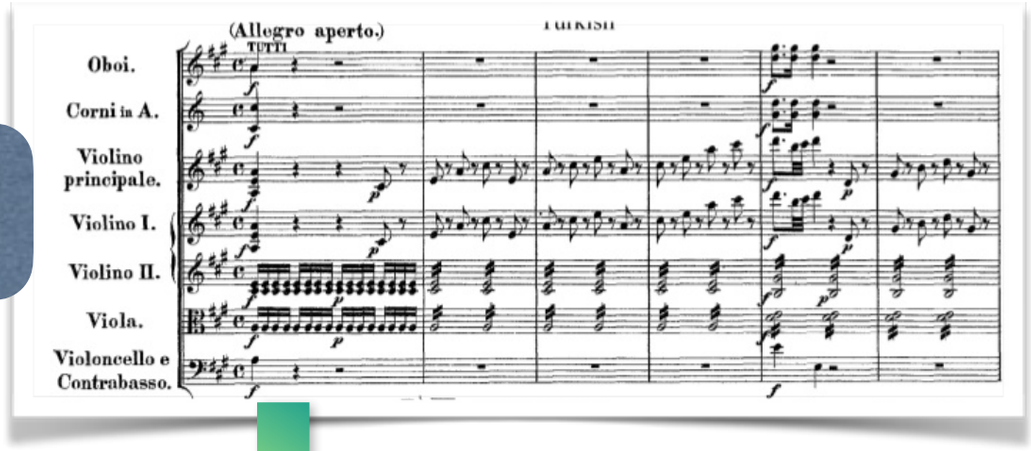
```
Total # of chords: 792
[100] ==> 0
[33, 45, 61, 64, 69, 81] ==> 1
[33, 45, 61, 64, 69] ==> 2
[36, 48, 60, 80] ==> 3
. . . . . .
```

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

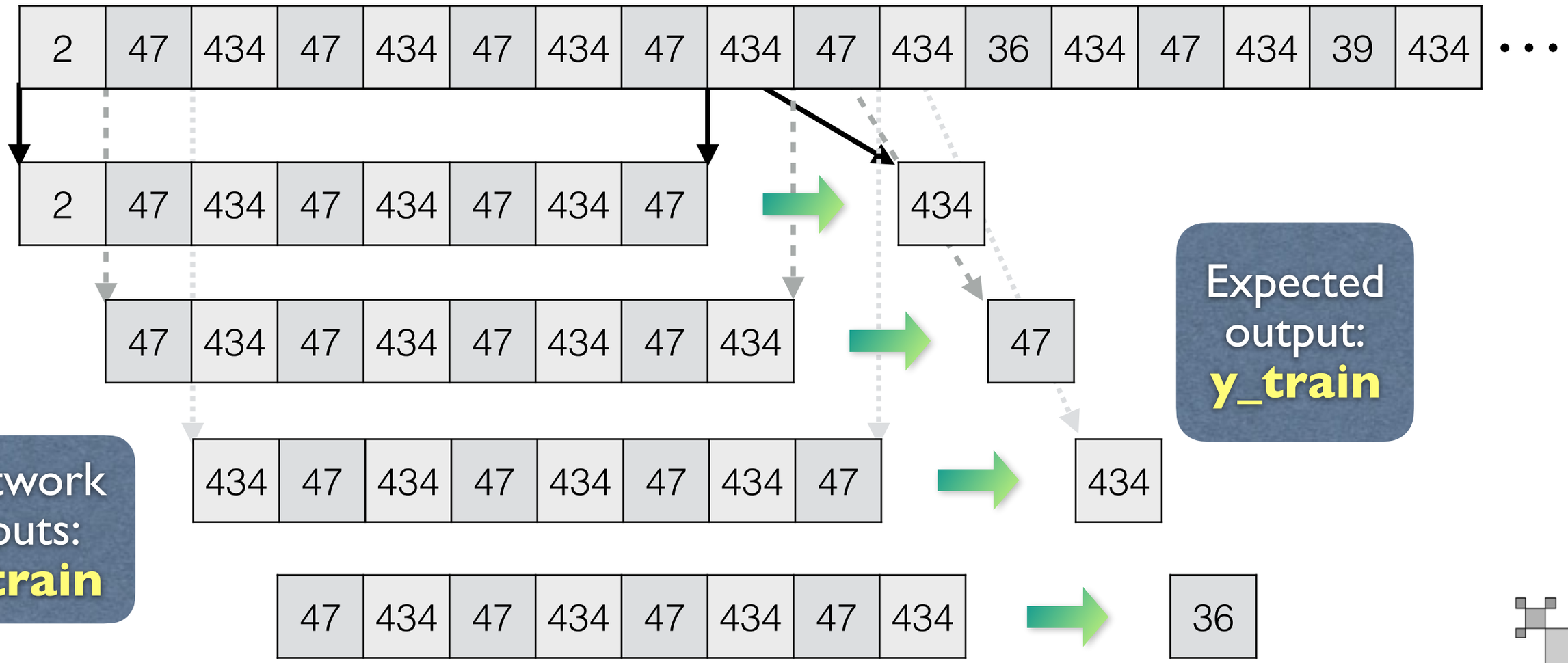
```
[33, 45, 61, 64, 69], [45, 49, 52], [57], [45, 49, 52], [57], [45, 49,
52], [57], [45, 49, 52], [57], [45, 49, 52], [57], [45, 49, 52, 61],
[57], [45, 49, 52], [57], [45, 49, 52, 64], [57], . . . . . .
```

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 tensorflow.keras.layers import LSTM, Dropout, Dense
from tensorflow.keras.layers import Activation, Input, Embedding
from tensorflow.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-ex03.h5')
```

Prepare x_train, y_train

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

Layers of LSTM

← softmax + x-entropy

partial ex_ml9_3.py

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: 4.3044
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!

partial ex_ml9_3a.py

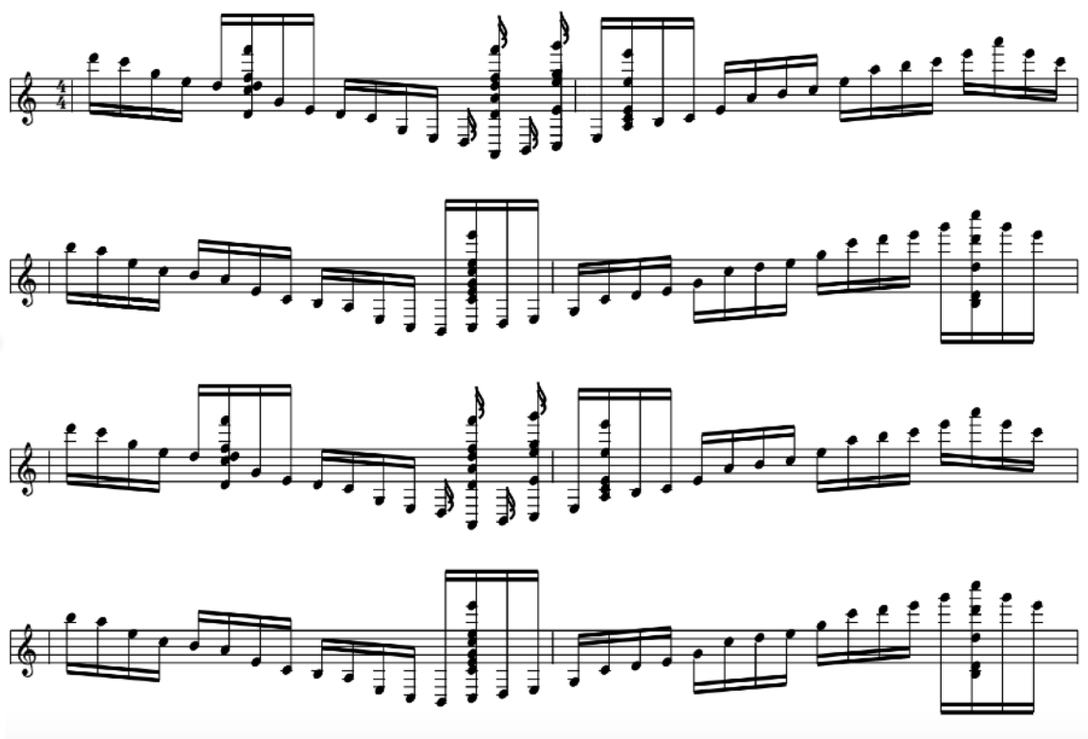
```
. . . . . .
model.load_weights('weights-ex03.h5')

x_test = np.array([np.random.randint(0,n_chords,length)])
result = []                                          ↑ seed of the song
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)
```

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 2 notes with highest pitches
    all_data.append(data)

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

partial ex_ml9_4.py

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…

partial ex_ml9_4.py

```
. . . . . .
for data_idx, data in enumerate(all_data):
    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))
model.add(LSTM(256))
. . . . . .
model.fit(x_train, y_train, epochs=150, batch_size=64)
model.save_weights('weights-ex04.h5')
```

← loop over 9 input MIDI files

↖ enlarged network

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, AGAIN

There is another commonly used "beam search", you can try to implement it, too!

- 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 *= 1.20
    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, T=%.2f' % (seq,result[-1],temperature))

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

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

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

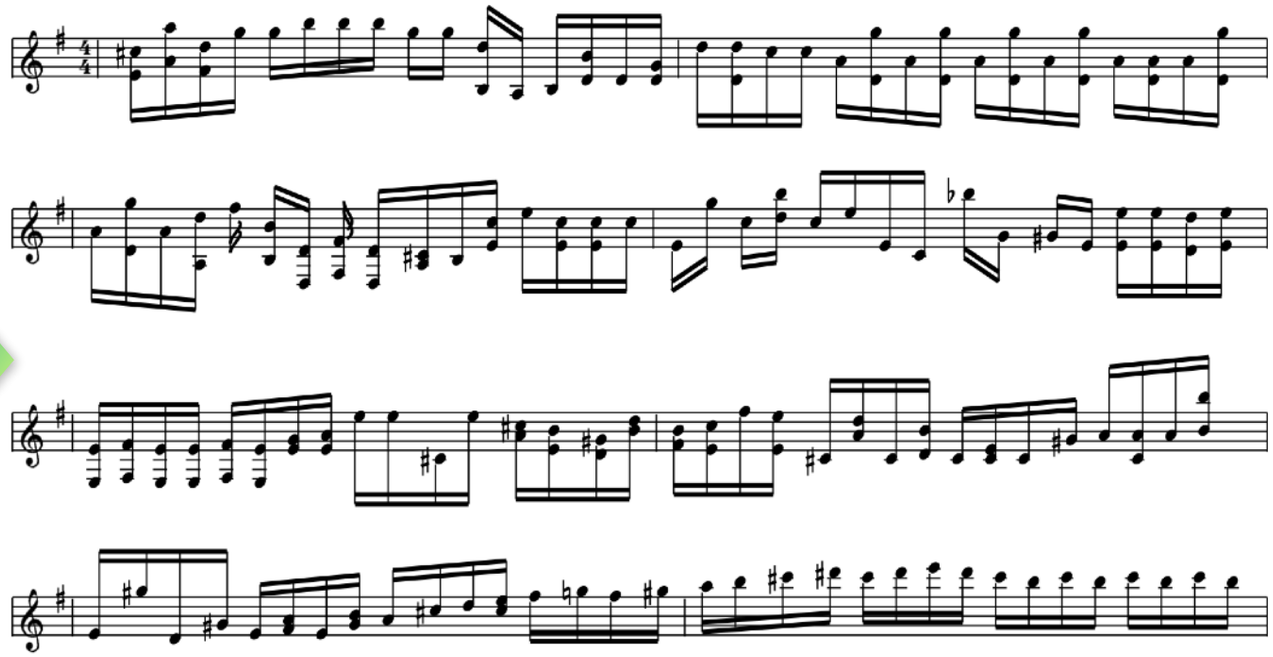
we are using the "probability" interpretation of the softmax function + rescaling by the temperature

partial ex_ml9_4a.py

MUSIC GENERATION, AGAIN (II)

- 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
. . . . . .
```



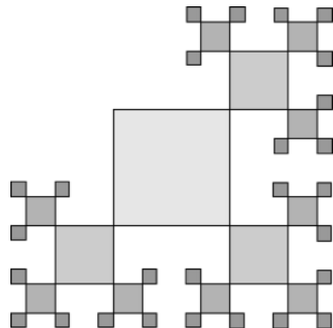
It sounds not too bad?
But obvious not-so-Mozart!

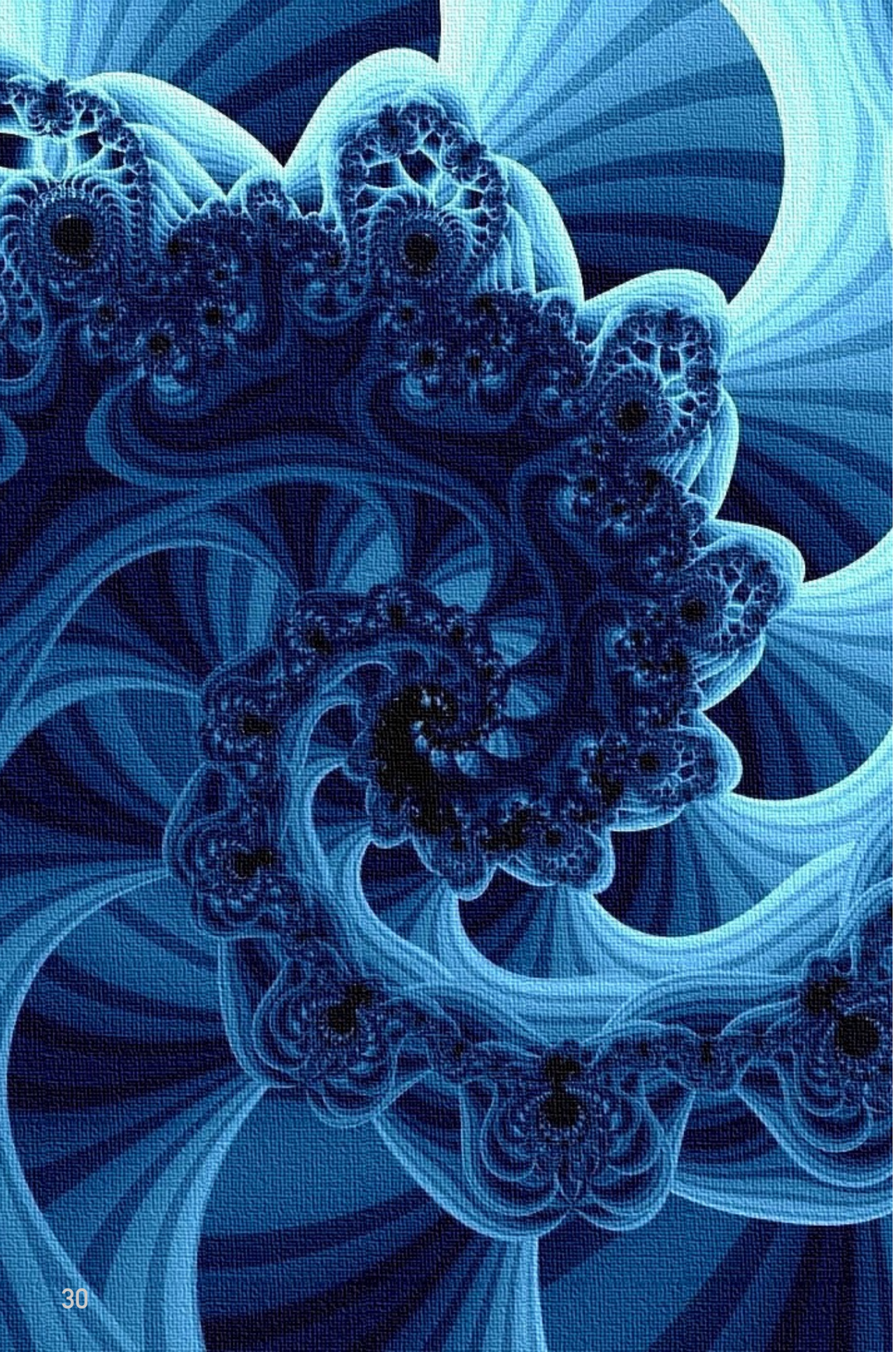
Trial #1

Trial #2

COMMENT: MUSIC GENERATION WITH 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





MODULE SUMMARY

- In this module we introduced another well-known deep structure learning models, RNN and LSTM, and implemented a simple code to generate music data.
- Next module we will introduce another famous and very interesting model, the Generative adversarial network (GAN) model.