Python project

Jacques BOONAERT-LEPERS - Anthony FLEURY - Guillaume LOZENGUEZ

octobre 2025

1 Project objectives

The objectives of this project are divided in two parts. The first functionalities to be achieved are the following:

- determining the note played by a musical instrument (whatever the instrument)
- recognize the instrument among a given set.

the second set of objectives, to be considered as the achievement of the project, is the following:

- as a first step, given an audio file containing the music performed by a single instrument, return the class of the instrument and the sequence of notes played.
- as a final step, given an audio file containing the music performed by several instruments (let's say 3), return the classes of each playing instruments and the sequence of notes played by each of them...
- **to evaluate the performances of your classifier, it will be tested on a short extract of classical music (about 60 s duration wave file). The winner is the one that provides the "partitions" that fit the music the best !**

The following subsections describe the dataset to be used and the scripts provided as starting points.

1.1 Musical dataset

The dataset to be used can be found at the following URL:
https://www.kaggle.com/datasets/abdulvahap/music-instrunment-sounds-for-classification?resource=download
as described, it is structured as 28 folders, one for each of the considered instruments. Each of these folders, in turn, contains a variable quantity of audio files using the **wave** format. The duration of the wave files is targeted to a 3 **s**

duration. Be aware the a file is not made of a single note played (most of the time, it contains a sequence of two notes, a least). They must be considered as sets of "acoustic" signatures of an instrument, reason why they are a valuable choice for training a classifier whose aim is to distinguish musical instruments among each others. As you can see, this structure are relatively close to the one adopted for the MNIST database, so that, adapting this musical dataset to a categorical "look and feel" (just as MNIST) should be an easy task for you.

1.2 data pre-treatment

So-called "wave" files deal with sound as a temporal signal. In brief, if you do not consider "cookery" such as data compression and of all the related stuff, they contain the sequence of samples that represent a sound signal $s(t)$ sampled at a given frequency F_e, leading to a sequence $\{s\left(\frac{k}{T_e}\right)\}_{0\leq k\leq N-1}$, N being the number of samples.
A prior intuition is that this raw temporal representation if not necessary the best approach to exhibit the most important properties of the sound signal. Then, it can be a good thing to change it to something else. One can imagine using a pure frequency-based representation (for instance, by computing the Fourier transform of the initial signal).
One of the drawback of these new representation if that the way the frequency evolves over time gets hard to decode (it is, most of the time, embedded in the low frequencies of the spectrum).
Finally, a good trade-off is to implement a signal representation that takes advantage of both of the temporal and the frequency-based representation (such as the Wiegner transform). In our case, we will use a sliding temporal window (**Hamming**) in which we compute the **Discrete Fourier Transform**, providing us the spectrum of the signal for that specific position of the temporal window. This last is shifted by a few samples "to the right" (at least 1), and the new spectrum is computed, again and again, until the end of the temporal signal is reached: the so-called obtained **spectrogram** is a very convenient way to express how the signal frequency components evolve with time, that may be considered as a pillar of musical signal.
Another interest of this type of representation is that it make sounds become visible... To be clear, at each position of the sliding temporal window, we get the spectrum of the signal within. If we take into account the magnitude of the spectrum (the same approach would also be valuable with the argument), it can be converted to gray levels or shades of colors (using a *colormap*), the lighter points being the location of the stronger frequencies of the spectrum.
As an illustration, figure 1 page 3 and figure 2 page 3 respectively show the spectrogram of a violin and a vibraphone. The shapes of the lines composing these spectrograms show interesting characteristics that make them recognizable with our eyes only: the violin spectrum is made of thin lines producing small sinusoidal waves, while the vibraphone spectrogram shows lines that spread up and

down. The figures has been obtained using the following code (that you can

Figure 1: spectrogram of a 3 s duration violin sound

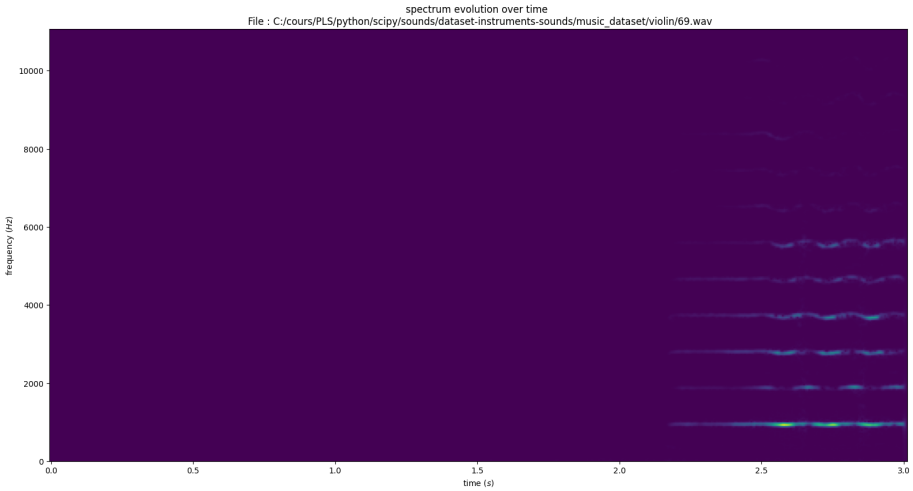
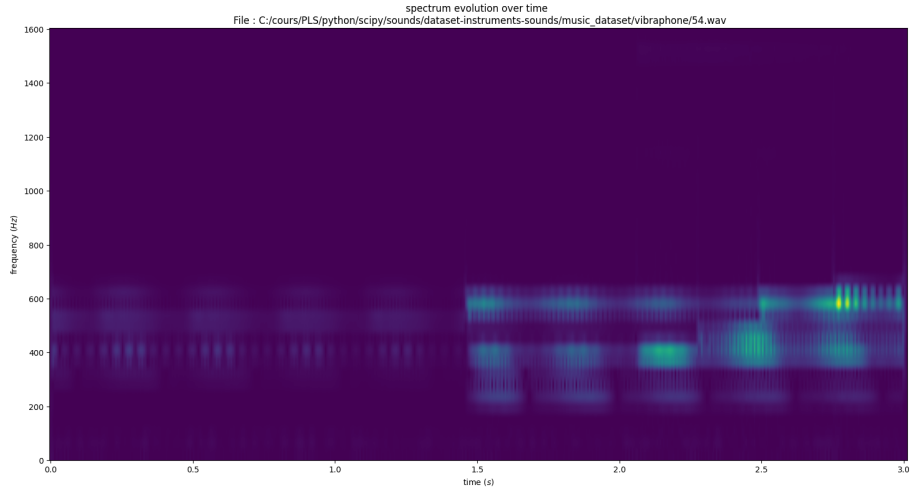


Figure 2: spectrogram of a 3 s duration vibraphone sound



also download from the shared folder whose URL is indicated in this document) :

```
#=============================================
# example of code that build spectrograms from
# WAVE files
#=============================================
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import ShortTimeFFT
from scipy.signal.windows import gaussian
from scipy.signal.windows import hamming
from scipy.io import wavfile
import simpleaudio as sa
import time
import os
#-----------
# constants
#-----------
lstINSTRUMENTS_CLASSES = [
      "acoustic_Guitar",
      "banjo"
      "bass_Guitar",
      "clarinet",
      "cowbell",
      "cymbals",
      "dobro",
      "drum_set",
      "electro_guitar",
      "floor_tom",
      "flute",
      "harmonica",
      "harmonium",
      "hi_hats",
      "horn",
      "keyboard",
      "mandolin",
      "organ",
      "piano",
      "saxophone",
      "shakers",
      "tambourine",
      "trumpet",
      "ukulele",
      "vibraphone",
      "violin"
]
```

```
# the following string is the location of the dataset :
szDATASET_DIR = 'C:/cours/PLS/python/scipy/sounds/dataset
    -instruments-sounds/music_dataset'
szINSTRUMENT = 'Violin'
#########
# help :
#########
def usage( szPgmName):
    print(szPgmName + "<instrument number><sample number
        ><outfile>")
    print("where <instrument number> is the integer
        corresponding to a given instrument")
    print("and <sample number> is the number of the
        sequence to be processed (in brief : wave file
        name")
    print("without the .wav extension")
    print("the correspondancy between intruments and
        instruments number is the following:")
    for i in range(len(lstINSTRUMENTS_CLASSES)):
        print(str(i)+ "---->" + lstINSTRUMENTS_CLASSES[i])
#########################################################
# build the complete file access path to the wave
# file given the instrument and the sample number
# IN :
#       iInstrument : instrument number
#       iSample     : sample number
# OUT :
#       szFileName
#########################################################
def BuildFilePath2Wave( iInstrument, iSample):
    szFileName = szDATASET_DIR   +  '/' +
        lstINSTRUMENTS_CLASSES[iInstrument] + '/' + str(
        iSample) + '.wav'
    return szFileName
#########################################################
# process a wave file to generate a npy array
    corresponding to a correlogram
# IN :
#       szFileName              : path to the WAVE file
#       width = 512             : size of the hamming window
#       stride = 16             : value of the hamming shift
    in number of samples
#       bDisplay = False        : display the spectrogram if
    True
#       bPlaySound = False      : play the wave file if True
# OUT :
```

```
#        npaSpgm      : spectrogram
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
def BuildSpectrogram( szFileName, width = 512, stride =
    128, bDisplay = False, bPlaySound = True  ):
    freq, wvData = wavfile.read( szFileName)
    N = wvData.shape[0]
    print("Sampling period = " + str(freq)  + "Hz -
        number of samples = " + str(N) )
    w = hamming(width, sym=True)  # symmetric Gaussian
        window
    SFT = ShortTimeFFT(w, hop=stride, fs=freq, mfft=width
        , scale_to='magnitude')
    Sx = SFT.stft(wvData)  # perform the STFT
    #-----------------------------------------
    # display if required
    if bDisplay:
        plt.imshow(abs(Sx), origin='lower', aspect='auto'
            ,
                extent=SFT.extent(N), cmap='viridis')
        plt.xlabel('time ($s$)')
        plt.ylabel('frequency ($Hz$)')
        plt.title('spectrum evolution over time \n File :
            ' + szFileName)
        plt.show()
    #-------------------------------------------
    #---------------------------
    # play sound if required
    if bPlaySound:
        wave_obj = sa.WaveObject.from_wave_file(
            szFileName)
        play_obj = wave_obj.play()
        time.sleep(N/freq + 0.1)
    #---------------------------
    return Sx
##################################################
# entry point :
argc = len(os.sys.argv)
if argc != 4:
    usage( os.sys.argv[0])
else:
    iInstrument    = eval(os.sys.argv[1])
    iSample        = eval(os.sys.argv[2])
    szFile         = BuildFilePath2Wave( iInstrument,
        iSample)
    npaSpectro     = BuildSpectrogram(szFile, bDisplay =
        True, bPlaySound= True)
```

```
szOutFileName = os.sys.argv[3]+ '.npy'
print('saving spectrogram to' + szOutFileName + '
    ....')
np.save(szOutFileName, npaSpectro)
print('done.')
```

1.3 a few directions for the classification task

As suggested in the last subsection, it appears that based on a adequate pretreatment (such as the production of a spectrogram), sounds produced by musical instruments can look like images that seem to embed sufficient "visual clues" to allow us to distinguish an instrument from another.
Assuming this hypothesis is correct, one can think using deep classifiers originally designed for image processing, to classify the sounds produced by musical instruments, using the following pipeline:

- **(1)**: extract a subpart of a wave file.
- **(2)**: build the corresponding **spectrogram**.
- **(3)**: apply this spectrogram to the input of a **classifier**.
- **(4)**: retrieve the output of the classifier.

The classifier's output must indicate the class label of the instrument that should be associated to the sound data presented at it's input.

1.3.1 building the spectrograms

The **BuildSpectrogram** function provided in the previous **python** script can help you for the pretreatment step (spectrogram construction). It must be kept in mind that the return variable **Sx** is an **numpy.array** of **complex** values. Each column corresponds to a given instant (that corresponds to the temporal shift of the hamming window inside which the FFT is computed). Then, the magnitude of a given row provides the *magnitude spectrum* at the considered instant (the *complex argument* would provide the *phase spectrum* for that row, then in turn, for that instant).
When calling this function, care must be taken regarding the value passed for the **width** and **stride** arguments:

- **width**: a rather large hamming window will provide a rather small frequency resolution, allowing to detect the slow evolution of the sound signal. As a counterpart, it can "blur" the visibility of sounds produced with an high dynamic. In other words, narrow hamming windows are good for detecting percussion while larger windows are a correct choice for instruments that produce low frequencies. If you feel it is too hard to decide for a single value, a good strategy could be to compute different spectrograms for the same sound signal, each of them characterized by different widths

of the hamming window (let's say 3, for instance: one small, to be good at detecting transient signals with an high dynamics, one intermediate for the "average" dynamics, and a large one to take into account low frequencies. These 3 spectrograms can then be stacked together exactly the same way as a color images, made of R, G and B color planes).

- **stride**: this represents the shift (in number of samples) applied to the hamming window for each **FFT** computation. A value of 1 will produce the evolution for each time step but will take almost forever to produce the calculation for a maybe unnecessary increase in the precision of the signal description (without telling about the bunch of memory the storage of the result will require...). A value strictly equal to the hamming window width will produce no overlaps between the temporal window, leading to no "coupling" between the successive FFTs (owing to the fact they do not share a part of their input samples) and a much faster calculation. Nevertheless, it can be desirable to introduce a part of samples dependency between successive spectra by specifying a stride value smaller than the width (that is the choice we made in the provided sample code).

- **for both** of these two parameters, always take values that can be expressed as a power of 2 (*i.e* 512, 128, *etc.*) so that the FFT algorithm can be employed the most efficient way.

Remark: *As you will certainly experience, the use of the* ***play()*** *method to actually produce sound from the* ***wave*** *files can lead to a brutal end of the script execution without a warning or any kind of message. As a consequence, avoid it when writing code designed for important processing!*

1.3.2 setting up the dataset

As explained in the previous sections, we first have to address a classification problem. That is, given a 3 s duration wave file, the classifier must output the label of the instrument that plays, assuming in this first step that there is only one musical instrument involved. Each instrument is labeled using an integer (from 0 to 27). Then, in terms of neural network structure, the input is made of the spectrogram (or made of a stack of spectrograms if you decide to use multiple values of the hamming window's width for a given input sound) and the output is made of a layer of 28 neurons, each one providing the "probability" of membership of that particular sound to the class of musical instrument it is attached to.
Except the fact MNIST deals with "true" images, this problem really looks like the one you addressed with Anthony in a previous session. As a consequence, structuring your dataset the same way seems to be a reasonable approach: for instance, you can create 28 folders named with numbers from 0 to 27, each of them containing the sample spectrograms you can build on the basis of the **BuildSpectrogram** function (in conjonction with the **glob()** function from the **glob** library, writing the code that automates the exploration of the sound files

hierarchy to produce such a dataset structure can be made an efficient way...).

1.3.3 designing the classifier

The choice of the deep classifier to be used is your, but you have to write the code that :

- construct the deep network structure *from scratch* using**tensorflow / keras** (**no transfer learning of pre-trained deep networks from the internet allowed...**
- write the code that train your network from **your** dataset.

Just as an example, the following code generate a residual deep classifier named **RESNET-34** using **keras** and **tensor flow** that may do the job (not tested in that situation, at your own risk):

```
#=============================================
# buiding a RESNET 34 deep network
# from scratch
#=============================================
import numpy
import os
import tensorflow as tf
from functools import partial
# model's paremeters :
npaDEFAULT_INPUT_SHAPE  = [224, 224,3]  # change this to
    adapt to the actual size of the input
iNB_CLASSES             = 10            # number of
    classes to discriminate
# default definition for 2D convolutional layers :
DefaultConv2D = partial(tf.keras.layers.Conv2D,
    kernel_size = 3, strides = 1, padding = "same",
    kernel_initializer = "he_normal", use_bias = False)
# definition of a residual unit :
class ResidualUnit(tf.keras.layers.Layer):
    #............
    # constructor
    #............
    def __init__(self, filters, strides = 1, activation =
        "relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(
            activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
```

```
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters,kernel_size = 1,
                    strides =  strides),
                tf.keras.layers.BatchNormalization()
          ]
    #..............................................
    # get the output of the residual unit given its input
    #..............................................
    def call( self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
#&&&&&&&&&&&&&&&&&&&&&&&
# making of the complete network :
# IN :
#       npaInputShape : shape of the input
#       iNbClasses    : number of classes to discriminate
#&&&&&&&&&&&&&&&&&&&&&&&&
def BuildRESNET34( npdInputShape = npaDEFAULT_INPUT_SHAPE
    , iNbClasses = iNB_CLASSES):
    model = tf.keras.Sequential([
        DefaultConv2D(64, kernel_size = 7, strides=2,
            input_shape = npaDEFAULT_INPUT_SHAPE),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Activation("relu"),
        tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
            padding="same"),
    ])
    prev_filters = 64
    for filters in [64] * 3 + [128] * 4 + [256] * 6 +
        [512] * 3:
        strides = 1 if filters == prev_filters else 2
        model.add(ResidualUnit(filters, strides=strides))
        prev_filters = filters
    model.add(tf.keras.layers.GlobalAvgPool2D())
    model.add(tf.keras.layers.Flatten())
```

```
    model.add(tf.keras.layers.Dense(iNB_CLASSES,
        activation="softmax"))
    return( model )
##########################
# entry point :
m1 = BuildRESNET34()
print(m1.summary())
```

2 Tasks to achieve

This last section recalls the tasks you have to try to achieve at the end of the project:

- given any of the sample wave files, provide the note played at a given instant. *indication: this functionality does not imply machine learning, because it can be driven on the basis of a "simple" frequency analysis*
- given any of the sample wave files, provide the name of the instrument that produce the sound. *indication: machine learning can be useful at this point*
- given any of the sample wave files, give the name of the instrument that produces the sound and the note played every $\frac{1}{4}$ s. A this step, we still assume that only one instrument is playing (we keep the original dataset unchanged).
- taking into account 5 different instruments (let's say **Violin**, **piano**, **cymbals**, **vibraphone** and **flute**), produce a new dataset that melts the sounds of these instruments. The classifier must output the combination of the instrument playing together.
- **challenging :** same question, but the classifier must also be able to indicate every $\frac{1}{4}$ s the note played by every instrument involved.