Advanced Audio Processing: Exercise 01

Data Handling:
Data I/O, feature extraction, & data feeding

1. Introduction

The goal of this lab exercise is to familiarize you with the processes of reading files from the disk, extracting features, and feeding the features to an algorithm. This exercise consists of three parts, all of them equally contributing to the assessment of this exercise (i.e. each part gets one point for successful completion). The first part is about input/output (I/O) operations with audio data. The second is about extracting features from the audio data and the third is about creating the process of feeding the data to an algorithm (e.g. a neural network). All these three processes described in the corresponding parts, can be grouped under the term "data handling", which is the focus of this exercise. From you, it is required to implement different tasks for each of the three parts and submit your solutions to the web page for the Advanced Audio Processing course in Moodle.

- ★ At the end of this exercise, at least the following files should be returned: answers.py, dataset_class.py, file_io.py, plotting.py and main_script.py.
- ★ In each of the tasks, you will be asked to implement some functions and write them inside one of the aforementioned files. The execution of each function should be included in the main_script.py file, as suggested by the comments in the template (provided in exercise_01.zip in Moodle). The main script should run automatically without errors (the teaching assistant will not debug your code).
- ★ Please place all files in a folder named as E1_firstname_lastname_studentnumber and upload it to the Moodle page as a .zip file.

2. File I/O with Python and audio data (1 point)

In this exercise, we will focus on reading multiple audio files and specifically audio files of waveform audio file format (WAVE) [1], usually indicated by the ".wav" extension.

2.1. Path handling (0.5 points)

The goal of this task is to familiarize yourself with built-in Python packages that are used for path handling and common mistakes that can be encountered when trying to access multiple files.

If somebody wants to programmatically access a file on a hard disk drive, then one must create a string representation of the location of the file. For example, the path "a/b.txt" points to the file "b.txt" which is under the directory "a". In this exercise, you will try two built-in Python libraries to deal with directories.
Do the following:

1. Create a directory where you will place the code files and the audio files for this task (e.g. "exercise_01_task_1").
2. Place the files **file_io.py** and **audio.zip**, which you can find at the Moodle page of the exercise, in the new directory.

3. Expand the archive **audio.zip**, and the directory audio will be created. It should contain nine audio files.
4. Finally, use the functions inside the **file_io.py** file in order to read the contents of the audio. Try both **get_files_from_dir_with_os()** and **get_files_from_dir_with_pathlib()** and compare the results.
5. Answer the following questions in comments in the **main_script.py** file (in the section 2.1):

a) What is the difference of the paths that are returned by each function?
b) What is the difference in the values returned by each function?
c) Using the **sorted()** function in Python, sort the paths. What do you observe in the ordering of the paths?
d) Without changing the conceptual name of the files (e.g. a file named 0001.wav can be changed to 001.wav but not to 0002.wav), how can you fix what you observed from the ordering of the files?

2.2. Reading audio files (0.5 points)

In order to read the data from WAVE files using Python, one can use different packages and libraries. A few of them are:

1. The wave built-in package [2],
2. The io.wavefile package of the scipy library [3], and
3. The librosa library and specifically the load function [4].

From the above three options we will use the third one.

For this task you are required to load each of the files from the audio directory and print the length in seconds of each file in the console. Place the code that implements the loading of the data in the body of the get_audio_file_data() function in the **answers.py** file.

For this task, do the following:

1. Import the librosa library
2. To get the paths of the audio files, use one of the functions from the previous task. Make sure the path is used correctly (e.g., it needs to be a string).
3. Read the documentation of librosa.load [4] to make sure you understand how the function works. You have to focus on the path and sr input arguments.
4. Use librosa.load() to load each audio file from the audio directory (you can do that in a loop). Pay attention to the output of the function (it returns a tuple).
5. To get the length of the file in seconds, take the length of the audio in samples (hint: a.shape[-1]) and divide it by the sampling rate.
6. Use the print() function to display the length.

3. Extracting features from audio data (1 point)

One of the most common operations when dealing with audio processing is to extract some features from raw audio data. These extracted features can be used for further processing of

the audio signal. There are many different audio features, today we're going to focus on the most popular ones – mel-scaled energies.

3.1. Extraction of mel-band energies (0.5 points)

In this task, you're asked to implement a function calculating mel-scaled energies from an audio file. Place the implemented function extract_mel_band_energies() in answers.py.

Do the following:

1. Load the audio file **ex1.wav**, use the functions implemented in previous tasks.
2. Extract the mel-band features. You have two options: use the librosa function feature.melspectrogram [5], which will do the job for you or implement it yourself (see chapter 5 at the end of this file). In both cases, use a Hanning window with the length of 0.02 seconds and 50% hop size and extract 40 mel bands.
3. Print the shape of the extracted features by using the **.shape()** method.
4. Use the functions in the **plotting.py** file to plot the audio signal and the extracted features.
5. Answer the following questions (put them in the comment in the corresponding section in main_script.py):
    a) Can you recognise which is the dimension of the array of the features that holds the different mel-bands?
    b) What would be the other dimension?
    c) Is there any connection between the length of the corresponding audio file and the extracted features? If yes, what connection? If not, why is there not a connection?
    d) Qualitatively explain the figure of the mel-band energies, in relation to the figure of the audio.

3.2. Serializing the extracted features (0.5 points)

There are many ways to serialize features. The two most common are: the first one, using the pickle built-in package from Python and the second, using the serialization capabilities of the numpy library. Usually, when one wants to serialize only numpy arrays, the second way is preferred. But, when one wants to serialize both features and annotations for the features (e.g. these feature vectors are annotated as "class 1"), then the first way (i.e. using pickle) is preferred.

In this task we will use the pickle package to serialize data. You will extract mel-band energies from all files in the audio directory, and associate the features from each audio file with either number 0 or 1 (choosing randomly), which will simulate a class that is assigned to the features (e.g. class 0 or class 1). Then, you will serialize the features with their associated class. Your code, implementing the serialization, has to be placed in the body of the serialize_features_and_classes function() in the **answers.py** file.

Do the following:

1. Loop over the files in the audio directory and extract mel-band energies for each of them.
2. For each file, pick randomly 0 or 1 to associate the file with a “simulated class”.
3. Combine the feature vector and its assigned class into a Python dictionary (using keys: “features” and “class”).
4. In serialize_features_and_classes() function, serialize the dictionary using the pickle.dump() method (check the documentation for details).

4. Data loaders: Feeding data to PyTorch modules (1 point)

In this task we will focus on creating a data loader, using the functions and classes provided by the PyTorch library. You will implement all the necessary code in order to have a fully functioning process of loading the data and iterating through your dataset, getting at each iteration the input and output (i.e. target values) data for your algorithm (e.g., a neural network). All the code will be based on the torch.utils.data package [8].

4.1. Creation of dataset class (0.5 points)

In order to have the data loading functionality, one first has to create a subclass of the dataset class [9]. You are provided with the file **dataset_class.py**, in which there is already the skeleton for the dataset subclass, called MyDataset.
Do the following:

1. Modify the input arguments of the __init__ method, so it can take as an input argument the directory where your dataset files (i.e. the serialized dictionaries) are. Use one of the functions in the **file_io.py** file to get the paths from all the serialized files. Implement the loading functionality using the load function from the pickle package. Store the loaded dictionaries in a list.
2. Make the __len__ method to return the amount of your dictionaries.
3. Finally, make the __getitem__ to return a tuple, consisting of the features and the class for each example. In order to create batches from the elements of a Dataset, Pytorch needs them to be of equal size. This can be achieved in many ways, but for this exercise, we can just compute the temporal average of the spectrograms so the features of every element are just a vector with the average energy of every Mel band.

Note that this approach (loading the features of every element of the dataset at __init__) provides a faster access to the features but has a high memory cost. For bigger datasets this might not be feasible and you might need to save just the file names during the initialization of the dataset and load the features at __getitem__ when they’re actually needed.

4.2. Iterating over the dataset (0.5 points)
Having your subclass of the dataset class, now is the time to iterate over your dataset. To do this, you will have to use the data loader from PyTorch [10].

Do the following:

1. Modify the function dataset_iteration in **answers.py,** by initializing the data loader class inside the function. Initialize the class with the following parameters: drop_last=True, shuffle=True, and batch_size=3. The function itself should also take those as an argument.
2. Iterate over the dataset via the data loader class and print its content for data example (e.g., print information about the type and size of the output values).
3. Experiment with the values of the arguments drop_last, shuffle, and batch_size and verify their functionality.

5. Supplementary part to 3.1
Mel-band energies feature extraction (optional, only if you want to implement the mel-band features yourself)

1. Calculate the Short-time Fourier Transform of the signal (hint: librosa.stft()).
2. Calculate the power spectrum ESD (hint: in Python a**2 is equal to a^2):

$$ESD = |STFT(x)|^2 , \quad (1)$$

3. Calculate the mel-scaled filters using the function filters.mel() from librosa. Check the documentation to understand how it works. You can set the number of filters to 40. Make sure the argument n_fft is the same as when calculating the STFT.
4. Apply the mel filters to the power spectrum:

$$MBE = dot(ESD, M), \quad (2)$$

where dot(·) is the dot product function (hint: **numpy.dot()**)

References

[1] https://en.wikipedia.org/wiki/WAV
[2] https://docs.python.org/3.7/library/wave.html
[3] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.io.wavfile.read.html
[4] https://librosa.org/doc/main/generated/librosa.load.html
[5] https://librosa.org/doc/main/generated/librosa.feature.melspectrogram.html
[6] http://librosa.org/doc/main/generated/librosa.stft.html
[7] https://librosa.org/doc/main/generated/librosa.filters.mel.html
[8] torch.utils.data — PyTorch 1.10.1 documentation
[9] https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset
[10] https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader