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 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.

- ★ Please upload to the Moodle the following files: answers.py, dataset_class.py and file_io.py .

The rest of the document is organized as follows. In Section 2 is the description of the tasks for the I/O operations. Section 3 holds the tasks for the feature extraction part of the exercise. Finally, in Section 4 is the description of the tasks for data feeding in an algorithm.

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

Reading and writing files (i.e. file I/O) from/to an external memory (e.g. hard disk drive) is a common task in programming, and Python has its own tools that can implement such functionality. 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)

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. This string representation of the location is called the path of the file, and the path is said to point to the specific file. For example, the path "a/b.txt" points to the file "b.txt" which is under the directory "a". To deal with paths, Python has two built-in packages. The first is "os" and the second is "pathlib". For this task, you will have to use the functions inside the file **file_io.py** and the files in the **audio.zip** archive. You will have to expand the archive file, and then use the functions to read the contents of the newly created directory. The goal of this task is to familiarize yourself with built-in packages from Python that are used for path handling and common mistakes that can be encountered when trying to access multiple files.

Specifically, first create a directory where you will place the code files and the audio files for this task (e.g. “exercise_01_task_1”). In the new directory, place the files **file_io.py** and **audio.zip**, which you can find at the Moodle page of the exercise. Then, expand the archive **audio.zip**, and the directory audio will be created. The audio directory has nine short audio files in it. Finally, you will have to use both of the functions inside the **file_io.py** file in order to read the contents of the audio and answer to the following questions in comments in the **answers.py** file:

1. What is the difference of the paths that are returned by each function?
2. What is the difference of the values returned by each function?
3. Using the sorted function in Python, sort the paths. What do you observe in the ordering of the paths?
4. 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)

Having the paths of different files is a first step in acquiring the data in the files. The next step is to actually read the data that are in the files, from where each path is pointing to. In order to read the data from WAVE files using Python, one can employ 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, as the most widely used in research from the Audio Research Group of Tampere University.

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 at the console. To do this, you will have to import the librosa library and use the librosa.load [4] function. To get the paths of the audio files, you will have to use one of the functions from the previous task. The goal of this task is to familiarize yourself with the process of reading online documentation of programming libraries, and loading audio files using a widely employed library for processing audio files.

Choose first which function you will use to get all the paths of audio files. If you choose the os-based function, make sure that you will do the needed additions to the paths. If you choose the pathlib-based function, then you will have to use the built-in function str to turn each Path object to string (e.g. str(Path('a'))). Then, go to the online documentation of librosa.load [4], and read the documentation. You have to focus on the path and sr input arguments. The sampling rate argument (i.e. sr) of the librosa.load function can be set to None or set to the sampling rate of the audio files (which is 44100).
Pay attention to what librosa.load returns! It returns a tuple. The first argument of the tuple is the data of the audio file and the second is the sampling rate at which the returned audio

data are sampled (to be used when using the native sampling rate option at the librosa.load function). Finally, use the librosa.load to load each audio file from the audio directory, get its amount of samples (i.e. its length), and divide that amount by the sampling rate of the data. The audio data is returned as a numpy array from the librosa.load function. To get the length of the data one should use: a.shape[-1], where a is the numpy array returned from the librosa.load function.

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. Make sure that your code works!

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. Then, these extracted features can be used for further processing of the audio signal. Although there are quite many options for feature extraction, lately it can be observed that most published methods in machine listening employ mel-band energies.

In this exercise you will learn how to calculate mel-band energies using the short-time fast Fourier transform (STFT) of an audio signal and the triangular mel-scale filters. You will extract mel-band energies from multiple files and then serialize the features (i.e. export them to the hard disk drive).

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

In order to extract mel-band energies from an audio signal, you have first to load the file with the audio data. Then, you have to calculate the STFT of that signal, calculate the energy spectral density (ESD), and use the mel-scale filters to scale the ESD. The result of the scaling is the mel-band energies of the signal. To do these processes in Python, you can use the liborsa library and specifically the stft [5] and filters.mel [6] functions.

For this task you have to implement the code that will do the above processes. To load the audio file, you will have to use your get_audio_file_data, developed in the previous task. To get the paths of the files, you will have to use one of the functions in the **file_io.py** file. For the calculation of the STFT you can use a frame length of 0.02 seconds, with 50% hop, and the Hanning window. You can use 40 mel filters. For this task you will use the file **ex1.wav** and not the files in the audio directory.

Specifically, first create the path of the file **ex1.wav** either using directly the os.path.join function or the pathlib.Path class. Then, load the audio data using your function get_audio_file_data that you developed in the previous step. From the loaded data, first extract the STFT using the stft function of the librosa library. You can check the necessary arguments at the online documentation of the stft function, and you will have to alter the n_fft and hop_length arguments. Then, calculate the ESD by

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

where | · | is the absolute operator and *STFT* is the STFT. The power operator can be implemented using the double star, e.g. 4**2 is the 4^2 or if a is an numpy array, then a**2 is the power of two for each of the elements of a. Afterwards, calculate the mel-scale filters M using the function `filters.mel` from librosa library. Again, check the online documentation of the function. You will have to alter the `n_mels` and `n_fft` arguments. Apply the mel-scale filters by

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

where dot(·) is the dot product function. Then, print to the console the shape of the extracted features. You can use the `numpy.dot` function for the dot product operation. You can get the shape of the extracted features by using the `.shape` method. Finally, plot on the screen the audio signal and the extracted features using the functions in the **plotting.py** file, and answer to the following questions:

1. Can you recognise which is the dimension of the array of the features that holds the different mel-bands?
2. What would be the other dimension?
3. 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?
4. Qualitatively explain the figure of the mel-band energies, in relation to the figure of the audio.

Your code has to be inserted at the body of the `extract_mel_band_energies` function at the answers.py file and your answers as comments under the `extract_mel_band_energies` function.

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.

In this exercise, you will again follow the process to load the data from the “audio” directory. Then, you will extract mel-band energies for each audio file loaded, using the function that you developed in the previous task. You will unify the extracted features of each file and either a 0 or an 1, in a dictionary with keys: “features” and “class”. The key “features” will have as values the features and the key “class” the class (i.e. the 0 or the 1). You will serialize the dictionary using the `pickle.dump` method. Your code, implementing the above,

has to be placed in the body of the serialize_features_and_classes function in the **answers.py** file. Finally, you will have to answer (in a descriptive but short manner) what should be done if you wanted to assign a different class to each feature vector. Your answer should be in comment format, below the serialize_features_and_classes function.

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

Feeding data in an algorithm might sound simple, but there are many details that have to be taken care of. For example, shuffling of the data, dealing with equal size batches, e.g if the data are given batch by batch to the algorithm, chances are that the batch size will not divide the amount of data (e.g. 10 files and a batch size of 3). What will happen with the last examples, not enough to create a batch? Also, an input/output operation is among the most costly (time-wise) operations in a modern-day computer. An advanced way of feeding data to an algorithm could have implemented a multi-process-based way of reading and feeding the data. That is, when one process is reading the data from the hard disk, another is feeding previously read data to the algorithm. Fortunately, most modern libraries for machine learning have functions and methods that effectively treat all the above.

In this task we will focus on creating a data loader, using the functions and classes provided in 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 [7].

4.1. Creation of dataset class (1 point)

In order to have the data loading functionality, one first has to create a subclass of the dataset class [8]. To do that, one has to subclass the dataset class, and implement the following functions:

- the __init__ method, in order to provide initialization functionality (e.g. read the files of the dataset),
- the __len__ method, in order to make the class able to answer to how many data/examples has (the __len__ is the function that is called when one does len() in Python), and
- the __getitem__ method, in order to make the class return data when used like “dataset class[10]”, e.g. what would be the 10th example that the object dataset class has to return?

You are provided with the file **dataset_class.py**, in which there is already the skeleton for the dataset subclass, called MyDataset. What is required from you for this task is to implement the above mentioned functionalities. For this, you will use the serialized files that you created in the previous task, plus all the functions that you have implemented in this exercise.

Specifically, firstly modify the input arguments to 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. Followingly, make the __len__ method to return the amount of your dictionaries. Finally, make the __getitem__ to return a tuple, consisting of the features and the class for each example.

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 [9]. You will have to use an object of your dataset class and specify the behavior of the data loader, regarding its arguments drop_last, shuffle, and batch_size.
From the function dataset_iteration in **answers.py** file, choose the following values for the arguments drop_last, shuffle, and batch_size: drop_last=True, shuffle=True, and batch_size=3. Experiment with the values of the arguments drop_last, shuffle, and batch_size and verify their functionality.

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] http://librosa.org/doc/main/generated/librosa.stft.html
[6] https://librosa.org/doc/main/generated/librosa.filters.mel.html
[7] torch.utils.data — PyTorch 1.10.1 documentation
[8] https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset
[9] https://pytorch.org/docs/stable/data.html?highlight=dataloader#torch.utils.data.DataLoader