Advanced Audio Processing: Exercise 02

Audio classification with convolutional neural networks

Introduction

The goal of this lab exercise is to show you how to perform classification on audio data using convolutional neural networks (CNNs). You are asked to create an audio classification system that predicts one label for each input audio signal.

A typical task in modern audio signal processing is to get an audio signal and predict a class for it. Examples of such tasks are audio tagging [1], music genre recognition [2], and acoustic scene classification [3]. The usual approach is to extract some features from an audio signal, and give the extracted features as an input to an audio classification system.

In this exercise you will see an example of audio tagging, focusing on the discrimination between music and speech. You will use a subset of a freely available dataset for the speech versus music classification. In task 1 you will process data using the functions developed in the previous lab exercise (1.0 point). In task 2, you will design a DNN system based on CNNs to classify if a given audio signal is speech or music (1.0 point). And finally in task 3, you will do the training and testing of your system (1.0 point).

If you are not familiar with how to develop deep neural networks in PyTorch you may check the "background" section at the end of this document (see also the provided example).

★ For this exercise you need to prepare and return the following files:
- **data_loader.py**
- **cnn_system.py**
- **cnn_training.py**

Please submit them along with the files from the previous exercise that are needed to run the code for this exercise (i.e., **file_io.py**, **getting_and_init_the_data.py**, and **dataset_class.py**) .

★ Please place all files in the root folder named as "**E2_firstname_lastname_studentnumber**" and upload it to the Moodle page as a .zip file.

★ Please check the required versions of Python and PyTorch from the Moodle front page "Details about exercises".

Task 1. Getting and initializing the data (1 point)

The dataset that you will use is a subset of the "Music Speech" dataset, developed by G. Tzanetakis [4]. The subset of the dataset is provided at the Moodle page of this lab exercise. For the needs of this task, the subset is divided in three splits:

- Training split to develop your model.
- Validation split to optimize your model.
- Test split to measure the performance of your system.

For this task you have to follow below steps:

1. Create a directory to host the code and various files for the present exercise. This directory will be called root directory for the rest of this document.
2. Get the file "**E2_template.zip**" from the Moodle page and extract all files within that into the root directory.
3. Get the archive "**music_speech_dataset.zip**" from the Moodle page of the lab exercise, and expand it in the root directory. This will create the corresponding directory "music_speech_dataset", containing the following three directories:
   (a) training
   (b) validation
   (c) testing
4. You can find the functions to extract the features at the following files:
   - file_io.py
   - dataset_class.py
   - getting_and_init_the_data.py

   Functions to load the data are in the file called **getting_and_init_the_data.py**, which makes use of the class MyDataset in the file **dataset_class.py**.
5. Next, you are going to set up three different data loaders, one for each group of features, i.e. training, validation, and testing. For that, open the file "**data_loader.py**" and implement the missing sections in the "load_data" function that are denoted by question marks.
6. A template for calling the "load_data" function on the "training" set is provided inside the main function. Repeat this for the "validation" and the "testing" splits by implementing the missing lines inside the "main" function. Note that for keeping track of the predictions of your system, the testing data loader should not shuffle the data.
7. Finally, to test your code, run the "main" function which should print the size of each split.

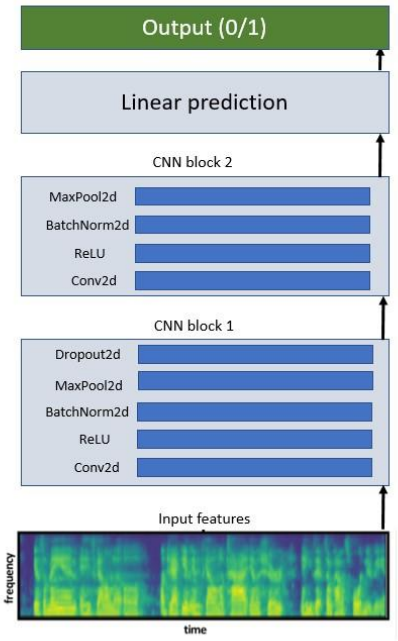

Figure 1. The CNN model to be implemented as an “audio classification” system (Task 2).

Task 2. Setting-up the CNN model (1 point)

In this task you will be focusing on implementing a CNN-based system. To do so, you will have to do the following:

1. Open the file “**cnn_system.py**” which includes a template for implementing your CNN model. Make the init method to set up two layers of CNNs (CNN block 1 and CNN block 2 in Figure 1), where each convolutional layer is followed by a rectified linear unit, a batch normalization process, and a max pooling process. Use the function “Sequential” to create the blocks. You should have the parameters for each layer to be defined using input argument to the $\_\_\text{init}\_\_$ method. As an example you can see the provided code “**dnn_pytorch_example.py**”.
2. Add a dropout with a probability defined at the init method at the end of the CNN block_1.
3. After the second CNN block, add a linear layer that will act as your classifier (output feature dimension is equal to 1).
4. Implement the forward method, which will have as an input a PyTorch tensor, process it according to your CNN, and return the output of your linear layer.
5. Implement the missing lines inside the “main” function to build the CNN model.
6. Finally test your code by running the “main” function that is provided to test your class using random data.

To calculate the output dimensions of the CNNs and the max pooling operations you can use the following equation (also available online at the documentation of the CNN classes at PyTorch website [5]):

$$H_{out} = \frac{H_{in} + 2 \cdot P[1] - D[1] \cdot (K[1] - 1) - 1}{S[1]} + 1 \quad (1)$$

$$W_{out} = \frac{W_{in} + 2 \cdot P[2] - D[2] \cdot (K[2] - 1) - 1}{S[2]} + 1 \quad (2)$$

where H is the height and W the width of the input $_{\cdot in}$ and output $_{\cdot out}$ of the CNN, and $K \in Z^2$, $P \in Z^2$, $S \in Z^2$, and $D \in Z^2$ are the kernel shape, padding, stride, and dilation (respectively) of the CNN. Please note that in the online documentation of PyTorch, the index of K, P, S, and D is zero based, in order to match the indexing of Python. In the context of exercise 2, the input to the CNN will be the log-mel band energies with $H_{\text{in}} = 40$ (i.e. log-mel band energies) and $W_{\text{in}} = T$, where T is the amount of frames of your audio.

Task 3. Training and testing (1 point)

After the set-up of a CNN-based system, you need to optimize its parameters. For that, different decisions have to be made. For example, how many CNN blocks (i.e. CNN, non-linearity, normalization, sub-sampling) to use, how many channels to use in each convolution layer, and what would be a good learning rate for the optimizer. For answering all such questions, the CNN-based system has to be trained with the training data and with a set of choices for the hyper-parameters, and then its performance has to be evaluated on

the validation data. At the end, the best performing system is chosen and its performance is assessed using the testing data.

For this task you are free to experiment with the hyper-parameters, keeping those that give as result the best performance. To do so, you will have to implement the following:

1. Open the file "**cnn_training.py**" which includes a template for training your CNN system.
1. Define the optimizer as Adam and give it model parameters and a learning rate.
2. Define the loss function as Binary Cross Entropy [6].
3. Implement the missing parts from training, validation and testing loops.
4. Finally, test your code by running the main function.
5. Experiment with different model parameters such as number of convolutional filters, and kernel sizes of convolutional and pooling layers. Report your observations as comments at the end of the "**cnn_training.py**" file.

Please note that the focus of the present exercise is not to get the best performing system. Instead, the focus is to correctly implement a typical audio classification scenario, where you will tune the hyper-parameters of your model.

Background:

PyTorch is a modern deep learning library, based on the Torch [7] and Chainer [8] libraries (for Lua and Python programming languages, respectively). PyTorch is free and open source, and actively developed and maintained by Facebook. In this Section you can find introductory information on how you can develop deep neural networks (DNNs) architectures and custom DNN layers, how to use different loss functions, and how you can use a graphics processing unit (GPU) to do all the necessary computations.

The tasks in this Section will not be graded and are only to serve as an introduction for this exercise. For further reading and more in-depth information, you can check the online tutorials [9] and documentation [10] of PyTorch.

Deep neural networks with PyTorch

Deep neural networks (DNNs) in PyTorch are handled in a dynamic fashion, using standard Python statements and functionality, and the same base class for all DNN layers. PyTorch allows for re-definition of the computational graph of the DNN (i.e. the series of computations) at each iteration. This allows the usage of conditions and easy implementation of advanced algorithms, compared to many other DNN libraries. All DNN layers and DNNs (i.e. multiple layers, organized in a complete system/architecture) in PyTorch are sub-classes of the $\mathrm{torch.nn.Module}$ [11] (Module) class. Consequently, all DNN layers expose the same application programming interface (API), as sub-classes of the same class. This allows for easy usage and replacement of different layers.

Additionally, DNN architectures are (or should be) implemented as sub-classes of the same Module class, also exposing the same API as all DNN layers in PyTorch. This allows for easy creation of custom layers and fast testing of different architectures. Apart from all the above, using the same Module class as the base class, allows for a unified way of controlling the calculations and handling of gradient. In this section you will get familiar with how you can define your DNN layers or DNNs, and how you can use a graphics processing unit (GPU) for the necessary calculations, using PyTorch.

At the provided file **dnn_pytorch_example.py**, is an example of a DNN using three linear layers, with non-linearities and dropout after each of them. For the completeness of the example, the hyper-parameters of the layers are defined using the input arguments to the __init__ function. The forward pass of the DNN is defined at the forward method. Again for the completeness of the example, it is assumed that the DNN can get two different input and return two different outputs. At the training loop, you can also see the usage of a loss function and an optimizer. Finally, the backward pass is handled automatically from PyTorch.
In the same **dnn_pytorch_example.py**, there is a main function, which will create some random data and run some epochs of the DNN in the same file. In order to understand how PyTorch handles DNNs, you have to do the following:

- Observe the creation of the object of the DNN. See if you are familiar with creating objects of classes in Python.
- Observe what is the usual information that is handled in the __init__ method, and how the layers are defined.
- Put a breakpoint at the first line of the forward method of the class and see when it is called and how the data are processed. You will have to use the debugger of your integrated development environment (IDE) and, specifically, the step-over and step-into functionalities (when needed).
- See how the loss function is handled and how the losses are logged for printing them after each epoch.
- Check how the optimizer is initialized and used, especially the .zero_ grad() method. The .zero_ grad() is a peculiarity of PyTorch, and what it does is that it zeroes out the previous gradients of your system that are logged from your optimizer.
- Finally, check how the update of the parameters is performed with the optimizer.

Using GPU with PyTorch

For speeding the calculations needed for a DNN, one can use a GPU instead of the CPU. To do this in PyTorch, one has only to transfer the variables to the GPU. Then, all operations with the involved variables will be performed on the GPU. Every tensor (e.g. data or parameters of a DNN) in PyTorch can be transferred from CPU to GPU, or vice-versa, by using the method .to(). This method gets as an input a string that signifies the device that will be used for the tensor. For example, the statement a = torch.Tensor([1, 2, 3]).to('cpu') will create a tensor, transfer it to the CPU, and assign it to the variable "a". We can transfer the variable (or the tensor) to the GPU by issuing the statement a.to('cuda'). Though, any operation between tensors that are not at the same device (i.e. CPU or GPU) will raise an exception from PyTorch. The above stands true for any tensor, for example a tensor holding the input data to the DNN.

Additionally, any object of the class Module can transfer its parameters to the GPU or CPU by using its method to(). By using this method, the object will automatically transfer to the specified device all of its parameters, even if these parameters are members of other objects of class Module in it. For more information about using different devices, you can check the online documentation of PyTorch [12].

To better understand the above, you will again have to go through the code in the file used in the above task, **dnn_pytorch_example.py**, and uncomment the lines where the method to() is used. Then, using the functionality of your IDE "Jump to definition", find the definition of the to() method and observe how the semantics of the devices are handled. Finally, check how the device semantics are handled for the data.

References

[1] https://www.kaggle.com/c/freesound-audio-tagging-2019
[2] https://www.crowdai.org/challenges/www-2018-challenge-learning-to-recognize-musical-genre
[3] http://dcase.community/challenge2019/task-acoustic-scene-classification
[4] http://marsyas.info/downloads/datasets.html
[5] https://pytorch.org/docs/stable/nn.html#conv2d
[6] https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html
[7] http://torch.ch
[8] https://chainer.org
[9] https://pytorch.org/tutorials/
[10] https://pytorch.org/docs/stable/index.html
[11] https://pytorch.org/docs/stable/nn.html?highlight=module#torch.nn.Module
[12] https://pytorch.org/docs/stable/notes/cuda.html