Advanced Audio Processing: Exercise 02

Audio classification with convolutional neural networks

1 Introduction

The goal of this lab exercise is to show you how to perform classification using audio data. This exercise consists of two parts, though only one is graded. The first part (which is not graded) will be an introduction on how to use PyTorch for creating deep neural networks, and specifically on how to use loss functions and optimizers, and do the back-propagation. The second part is about creating an audio classification system, that predicts a label for the whole audio signal used as an input. For the implementation of this exercise you will have to use the PyTorch deep learning library[1], and the functionality developed at the previous lab exercise[2]. For any case, you will also be given an implementation of the code asked for the previous exercise. When you have to use code developed at the previous exercise, you are free to use either the code provided or the one that you developed at the previous exercise. Chances are that your learning experience will be maximized if you use the provided code for the previous exercise, only as a guideline in order to optimize (i.e. correct or make better) the code that **you developed** at the previous exercise. It is OK, and encouraged, to use your code from the previous exercise. It is also OK to use the provided code for this exercise. You should not import and use any other code that implements needed functionality for this exercise.

All code will be implemented in Python programming language, version 3.7, following directions from the PEP 8 (the official style guide for Python programming language)[3] and conforming to the application programming interface (API) of PyTorch library for deep neural networks. The tasks in this exercise utilize two non built-in packages of Python. The first is `librosa`[4] and the second PyTorch. If there is a need to install the packages, one can check at the provided corresponding webpages of each package. The rest of the document is organized as follows. In Section 2 is an introduction on how one can develop deep neural networks using PyTorch, and Section 3 holds the description of the tasks for classification using convolutional neural networks (CNNs).

2 Deep Neural Networks with PyTorch (0 points)

PyTorch is a modern deep learning library, based on the `Torch`[5] and `Chainer`[6] 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 here only to serve as an introduction for the next Sections in this exercise. For further reading and more in-depth information, you can check the online tutorials[7] and documentation[8] of PyTorch.

[1] `https://pytorch.org`
[2] `https://moodle.tuni.fi/mod/assign/view.php?id=218411`
[3] `https://www.python.org/dev/peps/pep-0008/`
[4] `http://librosa.github.io`
[5] `http://torch.ch`
[6] `https://chainer.org`
[7] `https://pytorch.org/tutorials/`
[8] `https://pytorch.org/docs/stable/index.html`

2.1 Deep neural networks with PyTorch (0 points)

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. Specifically, 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 an complete system/architecture) in PyTorch are sub-classes of the `torch.nn.Module`[9] (`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 on 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 is 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.

2.2 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

[9] `\https://pytorch.org/docs/stable/nn.html?highlight=module#torch.nn.Module`

`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[10].

To better understand the above, you will have again 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.

3 Predicting a Label for an Audio Signal with Convolutional Neural Networks (3.0 points)

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[11], music genre recognition[12], and acoustic scene classification[13]. 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 task 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, process it using the functions developed in the previous lab exercise, and develop a DNN system based on CNNs, in order to classify if an audio signal is speech or music, and do the training and testing of your system.

3.1 Getting and initializing the data (1.0 point)

The dataset that you will use is a subset of the "Music Speech" dataset, developed by G. Tzanetakis[14]. 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. One for training your system, one for validating your system, and one for testing your system. You will have to use the three splits to develop, optimize, and test the performance of your system, according to what you have been taught in the lectures of the course. Specifically, you will have to:

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 archive "music_speech_subset.zip" for the Moodle page of the lab exercise, and expand it in the `root` directory. This will create the corresponding directory "music_speech_subset", containing the following three directories:
   (a) training
   (b) validation
   (c) testing
3. Create three different directories, again called "training", "validation", and "testing", in the `root` directory.
4. Set up the proper parameters/arguments for the functions developed in the previous lab exercise, and extract and serialize the features from the directories in "music_speech_subset" to the

[10] `https://pytorch.org/docs/stable/notes/cuda.html`
[11] `https://www.kaggle.com/c/freesound-audio-tagging-2019`
[12] `https://www.crowdai.org/challenges/www-2018-challenge-learning-to-recognize-musical-genre`
[13] `http://dcase.community/challenge2019/task-acoustic-scene-classification`
[14] `http://marsyas.info/downloads/datasets.html`

corresponding ones at the `root` directory. For example, the features extracted from the audio files in "music_speech_subset/training" should be placed in the "root/training", where "root" is the `root` directory.

5. Set up three different data loaders, one for each group of features, i.e. training, validation, and testing. For keeping track of the predictions of your system, the testing dataloader should not shuffle the data.

Your code should be placed in a file called `getting_and_init_the_data_<your_name>.py`, where at the `<your_name>` part, you will have to fill your name (e.g.: `getting_and_init_the_data_kostas_drossos.py`). The file has to be submitted for the assessment of the present task.

3.2 Setting-up the CNNs (1 point)

Creating a DNN system and choosing appropriate hyper-parameters requires a series of experiments and experience and knowledge about the task, the DNNs, and the data that will be used. In this task you will be focusing on implementing a CNN-based system that works and you will be doing a small series of experiments to optimize the hyper-parameters of your system. To do so, you will have to do the following:

1. Create a class, extending the `Module` class of PyTorch, and give it a reasonable name (e.g. "MyCNNSystem").
2. Make the `__init__` method to set up two layers of CNNs, each layer followed by a rectified linear unit, a batch normalization process, and a max pooling process. You should have the parameters for each layer to be defined using input argument to the `__init__` method. As an example you can see the provided code at Section 2.
3. Add a linear layer, that will act as your classifier, using output features (i.e. `out_features`) equal to 1.
4. Add, where appropriate, dropout with a probability defined at the `__init__` method.
5. Implement the `forward` method, which will have as an input a PyTorch tensor, process it according to your DNN, and return the output of your linear layer.
6. Test your class using random data, created with the help of `torch.rand`.

All of your code should be placed in a file having the same name as your class, but using the snake notation. For example, if your class is called "MyCNNSystem" then your file should be called `my_cnn_system_<your_name>.py`. The file should be placed in the `root` directory and submitted with any other file of this exercise.

To calculate the output dimensions of the CNNs and the max pooling operations (if you use such an operation), you can use the following equation (also available online at the documentation of the CNN classes at PyTorch website[15]):

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

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

where H is the height and W the width of the input $\cdot_{\text{in}}$ and output $\cdot_{\text{out}}$ of the CNN, and $K \in \mathbb{Z}^2$, $P \in \mathbb{Z}^2$, $S \in \mathbb{Z}^2$, and $D \in \mathbb{Z}^2$ 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 bands 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.

[15] `https://pytorch.org/docs/stable/nn.html#conv2d`

3.3 Training and testing (1 point)

After the set-up of a DNN-based system, it is the time of optimizing its parameters (i.e. training). Different decisions have to be made, for example how many channels for the CNNs, what would be a good learning rate for the optimizer, and how many CNN blocks (i.e. CNN, non-linearity, normalization, sub-sampling) will be better to use. For answering all such questions, the DNN-based system has to be trained with the training data and with a set of choices for the hyper-parameters, and then evaluate its performance on a split of the data that is reserved specifically for this purpose, i.e. the validation data. At the end, the best performing system (using the training and validation splits) is chosen and its performance is assessed using the testing data.

For this task you will use the code developed in the previous exercise and tasks and you will perform the training of your system, using some set of hyper-parameters. You are free to experiment with the hyper-parameters, keeping those that result to the best performance. To do so, you will have to implement the following:

1. Set-up the training loop for a maximum amount of epochs. You can start using 100 as maximum amount of epochs.
2. At the end of every epoch, validate your model.
3. Implement the logging of the losses and early stopping.
4. (Optional) When the training is done, alter the hyper-parameters and assess the performance of your system on the validation data.
5. Do the testing using the testing data.

Again, you can have as an example the code provided for the tasks in Section 2. Please note that the focus of the present task is not to get the best performing system. Instead, the focus is to implement correctly a typical audio classification scenario, where you will tune the hyper-parameters of your model. Your code has to be placed in a file called `training_1<your_name>.py` and submitted along with any other files of this exercise.