Advanced Audio Processing: Exercise 03

Audio detection with convolutional and recurrent neural networks

1 Introduction

In the previous lab exercise you implemented an audio classification system using convolutional neural networks (CNNs), assigning one label to a whole audio signal. But, there are quite many times where we are concerned with identifying the activities of classes in shorter audio segments, effectively allowing us to identify the onset and offset times of the corresponding activities of classes (i.e. to perform detection). To do so, the predicted activities of classes have to be in a fine time resolution, usually not less than the time resolution of the employed short-time Fourier transform (STFT) for the feature extraction (e.g. 0.02 seconds). CNNs can be employed for detection, by using proper hyper-parameters that do not alter the time dimension of the input audio features. What typical CNNs lack though, is the ability to model long term temporal dependencies, useful for detection of activities of classes that exhibit inter- and intra-temporal structures at the input audio features. For this reason, one can use recurrent neural networks (RNNs), which are known to model such long term structures. In this lab exercise you will get familiar with employing RNNs for audio detection. You will use the implementation of two RNNs that are widely employed in all systems that are using RNNs, the gated recurrent units (GRUs) and the long short-term memory (LSTM) RNN. The dataset that will be employed is a subset of the freely available TUT Real Life 2017 sound event detection dataset [1]. In all tasks, you are free to use either a CPU or a graphics processing unit (GPU) for implementing the necessary calculations. Additionally, you will get familiar with formulating the target values for sequences and multi-class/multi-label problems.

★ Please upload to the Moodle the following files: **rnn_example.py** (for section 2), **sed_data_loading.py**, **crnn_system.py** and **crnn_training.py** (for section 3).

The rest of the document is organized as follows. In Section 2 you will implement a simple RNN-based DNNs, using both GRUs and LSTM RNNs, and see the k-hot encoding scheme. Section 3 holds the description of the tasks for performing sound event detection using stacked CNNs and RNNs.

2 Recurrent neural networks in PyTorch (1.5 points)

Most (if not all) deep learning libraries support RNNs and PyTorch is not an exception. In this task you will get familiar with how RNNs are handled in PyTorch and how one formulates the encoding of target values in a multi-class/multi-label scenario. You will see both schemes that PyTorch offers for RNNs and you will implement a dummy case of an RNN-based DNN in order to get familiar with how sequences can be used and with multi-class/multi-label problems in PyTorch.

2.1 RNNs as black-box modules (1 point)

In Pytorch, RNNs come in two schemes. As a sub-class of RNNBase or RNNCellBase (both classes are in torch.nn). In both cases, an RNN gets as an input T sequences of F features, e.g. $\boldsymbol{X} \in R^{T \times F}$. In this task you will focus on the case where RNNs are a sub-class of torch.nn.RNNBase and you will implement three different dummy RNN-based DNNs, using dummy data. Specifically, you have to:

1. Create a two layer RNN-based DNN, using T = 64 and F = 8. The dimensionality of the first layer should be H_1 = 4 and of the second H_2 = 2.
2. Add a linear layer at the end, used as a regressor and with output dimensionality $Z = 2$. Check the range of values that the random generator has in PyTorch, and use the proper non-linearity at the end of the linear layer.
3. Create some dummy data for input and target values, using the above defined variables. Both input and target values should be continuous (floating point numbers).
4. Do a training (only training, no validation or testing) using the function torch.nn.MSELoss [2] and 100 epochs. For every epoch, display the mean loss in the console (you are free to reuse code from previous exercise). Note that you should not use only the last output of the RNN, but the whole sequence.

Implement the above for the RNNBase-based GRU [3] and LSTM [4]. Using the Python debugger and the online documentation of RNNs in PyTorch, observe and comment on the differences of the input/output variables of GRUs and LSTMs (e.g. do all RNN types output the same amount of hidden-to-hidden vectors?).

(Optional)
Although using the above classes for RNNs (i.e. subclasses of the RNNBase) offer convenience, there are limited possibilities for the development of advanced methods. A more flexible option is the RNNCellBase, however it is more complex and needs more lines of code. Pick one of the above-used types of RNN, and implement the above for the RNNCell-based GRU and LSTM (i.e. GRUCell [5] or LSTMCell [6]). Using the Python debugger and the online documentation of RNNs in PyTorch, observe and comment on the differences of the different types of RNNs (i.e. RNNBase versus RNNCellBase types). For example, could you use a condition statement (i.e. if/else) in the loop of the RNNBase type RNNs? Could you use a condition statement in the loop of the RNNCellBase type RNNs?

2.2 *k*-hot encoding of target values (0.5 points)

Up to now you have seen binary classification problems (i.e. predicting 0 or 1). In this task you will be focusing on a multi-label/multi-class problem, employing sequences. You will use one of the RNN-based DNNs that you implemented above (of your choice), but now your target values will be multi-class/multi-label. This means that each input feature vector could have been assigned multiple classes.

To deal with the encoding of the multi-class/multi-label annotations, usually one employs what is called k-hot encoding. k-hot encoding is an extension of the one-hot encoding, where a class c is represented as a vector $\mathbf{v} \in \{0, 1\}^C$, with C to be the number of classes and

$$v_i = \begin{cases} 1, & if\ i = c, \\ 0, & otherwise \end{cases}$$

In a k-hot encoding scenario, there can be multiple elements of $\mathbf{v}$ equal to 1, signifying the simultaneously active classes. In this task you have to:

1. Employ one of the RNN-based DNN systems that you developed in the above tasks (any type is fine), altering the dimensionality of your classifier to C = 4.
2. Modify your dummy data in order to have as target values a vector of length C, initialized with 0's and with random amount of 1's. That is, you will simulate a case where a feature vector can have one or more active classes, from four available.
3. Use the BCEWithLogitsLoss [7] class for calculating the loss (pay attention to the documentation and especially to the part about not using a non-linearity).
4. Do the training process for your DNN (one of the above that you developed), using the new multi-label/multi-class dummy data and for 100 epochs (again, print to console the mean loss for every epoch).

For sections 2.1, 2.2 and 2.3 the corresponding code and corresponding answers to each question should be placed in the file **rnn_example.py** (please organize your code in functions, with proper/descriptive naming), which you will submit. Any text answers (i.e. not code) should be as comments in the Python file.

3 Audio detection (1.5 points)

One of the most prominent audio detection tasks is the sound event detection (SED). At SED, extracted features $\boldsymbol{X} \in R^{T \times F}$ from audio signals, are given as an input to an SED system, the system processes its input, and outputs its predictions of activities of classes $\hat{\mathbf{Y}} \in \{0, 1\}^{T \times C}$, where T is the number of time-frames (obtained after windowing the input signal), F is the number of features (e.g. log mel-band energies, usually set as $F = 40$), and C is the number of classes (C depends on the dataset). In the following tasks, you will implement an SED system, using stacked CNN and RNN layers and a subset of the freely available dataset TUT Real Life 2017. The focus of the tasks is not to get a high performing system, but have a working system. Firstly, you will create the code that will handle your data, then you will develop your SED system, and finally you will implement the training, validation, and testing processes.

3.1 Loading and handling the data (0.25 points)

For this task you will be provided a dataset, similar to what was provided to you at exercise 2. You will have to set-up your dataset and data loaders, using the provided splits. You should follow these steps:

1. Download the file **sed_dataset.zip** from Moodle and expand it. You will have three directories (i.e. training, validation, and testing), similar to exercise 2.
2. Set-up your data loader, so you can use all three splits, similar to exercise 2. You can use your code or the code provided in the previous exercises.

You should submit all your code (even the code that you have developed in previous exercises). Code developed in this section should be in a file called **sed_data_loading.py**.

3.2 Implementing the DNN system (1 point)

For the DNN system that you will implement, you will use your CNN-based system from the previous exercise and one RNN layer. You can choose freely between all available types, though a GRU (either GRU or GRUCell) is recommended. Specifically, you will have to:

1. Use one to three CNN blocks, carefully selecting hyper-parameters (e.g. kernel shapes) in order not to affect the time dimension of your data.
2. After the CNN blocks, reshape your data (with torch.reshape [8]) in order to have the dimensions *batch_size* × *time-steps* × *features*. The output of a CNN is *batch_size* × *channels-conv* × *time-steps* × *channel-features*. To do the reshaping, you first have to use the reshape function to alter the order of dimensions and then use the view function [9]. The order of the dimensions should be *batch_size*× *time-steps*× *channel-features* × *channels-conv*, and then use the view function to have the final shape of *batch_size* × *time-steps* × *channel-features**channels-conv.
   For example, if input to convolutional layer has shape of (N, 500,40) with N= batch size, 500 time_steps, and 40 log mel-band features, and we use a Conv2d layer with in_channels=1 and out_channels=64 (number of convolutional filters), then the output shape is (N, 64, 500,40). So, we first need to reshape it to (N, 500, 40, 64) and then use view function to change it to (N, 500, 40*64).
3. Append an RNN after your CNN blocks and give to it as an input the reshaped data.
4. Append a classifier after your RNN in order to perform classification (do not use a non-linearity, for more info see PyTorch documentation). Also, PyTorch automatically shares the weights of a linear layer through time, if the input to the linear layer is a sequence.
5. The output dimensionality of your classifier should be equal to the amount of classes that will be supported (i.e. recognized). You can check that amount from your data, by loading one example and checking the target values (which are in a k-hot encoding fashion).

You should submit all your code (even the related code that you have developed in previous exercises). Code developed in this section should be in a file called **crnn_system.py**.

3.3 Training, validation, and testing processes (0.25 points)

Finally, you have to implement the training, validation, and testing processes of your system, using the BCEWithLogitsLoss class [7] from PyTorch and any part of your or the provided code from the previous exercise(s). You will have to:

1. Set-up the loop over epochs (e.g. 300 epochs).
2. Set-up the early stopping process, similar to exercise 2.
3. Log training, validation, and testing losses and print them to console (where appropriate).

Place the code for section 3.3 in the file **crnn_training.py** .

References

[1] https://zenodo.org/record/814831#.YfL_EPexXu0
[2] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html#torch.nn.MSELoss
[3] https://pytorch.org/docs/stable/generated/torch.nn.GRU.html#torch.nn.GRU
[4] https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM
[5] https://pytorch.org/docs/stable/generated/torch.nn.GRUCell.html#torch.nn.GRUCell
[6] https://pytorch.org/docs/stable/generated/torch.nn.LSTMCell.html#torch.nn.LSTMCell
[7] https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html#torch.nn.BCEWithLogitsLoss
[8] https://pytorch.org/docs/stable/torch.html?highlight=reshape#torch.reshape
[9] https://pytorch.org/docs/stable/tensors.html?highlight=view#torch.Tensor.view