Machine learning for vison and multimedia
Keyword Spotting with Pytorch

Keyword spotting (KWS) is a typical problem defined in the context of speech processing. A special case of KWS is the case of wake word detection that is commonly used by personal digital assistants as Alexa and Siri to “wake up” when their command is spoken. This is not the case of speech recognition, that needs much more complex systems and structures, but this approach can be used to identify a set of commands to operate on a personal device.

The objective of this lab is to experiment with a workflow for training a simple network that can learn how to detect know commands in one second of audio, e.g., 'up', 'down', 'left', 'right', 'stop', 'go', 'yes', 'no'.

Some of the steps described in this document are implemented in the notebook provided with the lab material. Activities to be performed by the student are identified with the keyword “Task”.

Note. The notebook modifies the Pytorch tutorial on Speech Command Classification with Torchaudio by using both Waveform and Mel Spectrogram as input to the Network Model and, optionally, using audio feature augmentation.

Preliminaries

Before starting, review or have ready the following material:

- Lesson "14. Machine and Deep Learning for Audio" slides and recording
- Pytorch tutorial on Audio Feature Extractions to compute and represent the Mel Spectrogram https://pytorch.org/audio/main/tutorials/audio_feature_extractions_tutorial.html#sphx-glr-tutorials-audio-feature-extractions-tutorial-py
- Pytorch tutorial on Audio Data Augmentation to extend the set of samples fo the training https://pytorch.org/audio/stable/tutorials/audio_data_augmentation_tutorial.html
- Pytorch tutorial on Speech Command Classification https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html

Dataset

We use torchaudio to download and represent the dataset. Here we use SpeechCommands, which is a datasets of 35 commands spoken by different people. The dataset **SPEECHCOMMANDS** is a **torch.utils.data.Dataset** version of the dataset. In this dataset, all audio files are about 1 second long (and so about 16000 time frames long).

The actual loading and formatting steps happen when a data point is being accessed, and torchaudio takes care of converting the audio files to tensors. If one wants to load an audio file directly instead, torchaudio.load() can be used. It returns a tuple containing the newly created tensor along with the sampling frequency of the audio file (16kHz for SpeechCommands).

Going back to the dataset, here we create a subclass that splits it into standard training, validation, testing subsets.

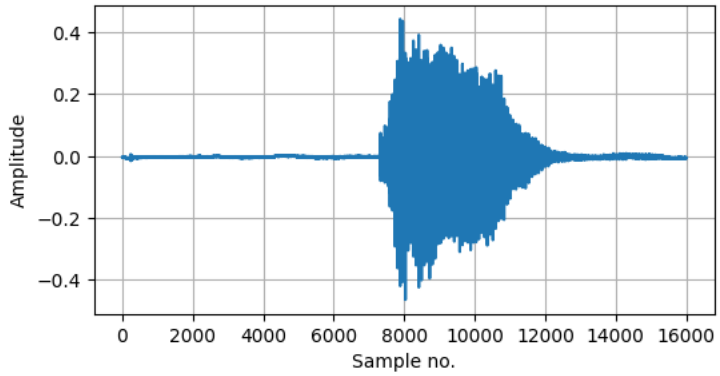
Note. In order to speed up the learning process, we restrict the number of different commands to a subset of the original thirtyfive commands: ['backward', 'bed', 'bird', 'cat', 'dog', 'down', 'eight', 'five', 'follow', 'forward', 'four', 'go', 'happy', 'house', 'learn', 'left', 'marvin', 'nine', 'no', 'off', 'on', 'one', 'right', 'seven', 'sheila', 'six', 'stop', 'three', 'tree', 'two', 'up', 'visual', 'wow', 'yes', 'zero'] (see the commands set).

```
commands = {'up', 'down', 'left', 'right', 'stop', 'go', 'yes', 'no'}
```

Exploring the data

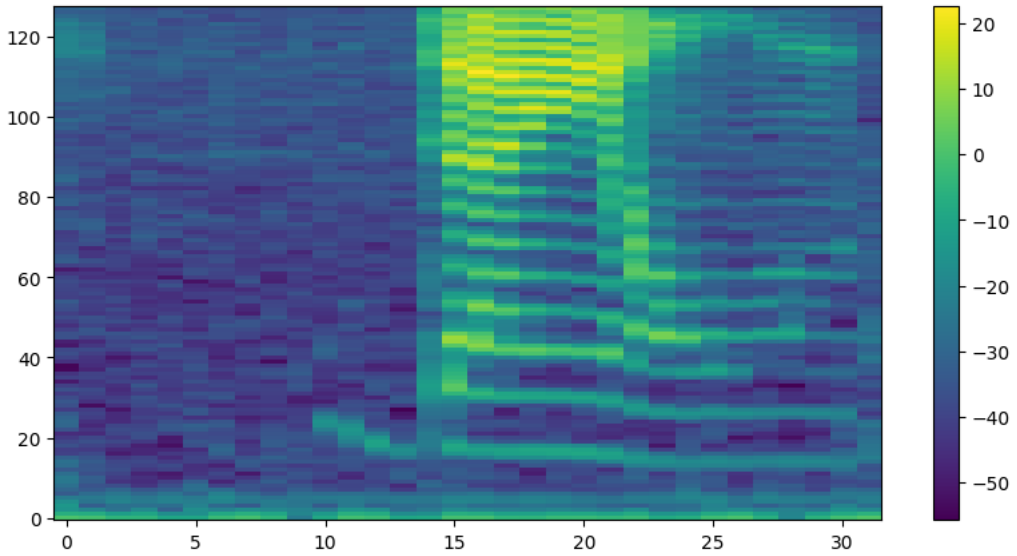
A data point in the SPEECHCOMMANDS dataset is a tuple made of a waveform (the audio signal), the sample rate, the utterance (label), the ID of the speaker, the number of the utterance.

Task: Waveform Plot. Select a sample from the dataset (e.g., idx=1), plot the waveform and print its shape. Listen to the same sample with "ipd.Audio(<waveform>, <rate>)"



Task: Mel Spectrogram. Use *Torchaudio transforms* to transform the sample data in an image, i.e., the Mel Spectrogram. First just use the default parameters, then follow the Pytorch tutorial on Audio Feature Extraction and experiment with different *win_length*: the frame length, *hop_length*: the "stride" length, *n_fft*: the number of frequency bins, *n_mels*: the number of mel bands. A possible result is as shown in the following figure.

- Q: How many samples (power of two) is a window of about 10ms? Why 10ms are often used?
- Q: What is the common ratio between the window length and the hop size?
- Q: How many FFT bins or Mel bands are commonly used in deep learning networks? See some examples in the slides, note if the numbers depend on the sampling rate. Is the same value used for both of them or not?
- H: Remember to compute the spectrogram values in dB (i.e., log scale).



Formatting the data

Functions label_to_index and index_to_label are used to map (and encode) each word with its index in the list of labels.

To turn a list of data point made of audio recordings and utterances into two batched tensors for the model, we implement a collate function which is used by the PyTorch DataLoader that allows us to iterate over a dataset by batches. Please see the documentation for more information about working with a collate function.

Each sequence is eventually padded with zeros so that tensors in a batch will have all the same length.

Generic functions for training and testing the network model

Now let's define a *generic* training function that will feed our training data into the model and perform the backward pass and optimization steps. The network will then be tested after each epoch to see how the accuracy varies during the training.

The *generic* function receives as parameters:

- The **trasform** function to apply to the data (or None).
- The **criterion** function for computing the loss.
- The **optimizer** to be used during the process.

Now that we have a *generic* training function, we need to make a *generic* **testing function** to evaluate the networks accuracy. We will set the model to eval() mode and then run inference on the test dataset. Calling eval() sets the training variable in all modules in the network to false. Certain layers like batch normalization and dropout layers behave differently during training so this step is crucial for getting correct results.

Define also a *generic* **predictor** to be used in the evaluation phase**.**

Phase 1. Model definition, training, and testing (Mel Spectrogram)

Task: Data pre-processing Mel Spectrogram Transform. Starting from the data visualization section where the Mel Spectrogram of a sample has been computed, build *transform pipeline* to convert the audio samples to the Mel spectrogram representation.

- H: check the shape of the 2D array before using it as input to the model, i.e. the number of bins (values) both in the time and frequency domain.

Task. 2D spectrogram based network Model. Define a convolutional neural network (CNN) starting from the architecture implemented in Lab n.03 for the fashion MINST dataset and adapting its structure to the data at hand.

Training and testing.

Task: training and testing. Train and test the network model. Assign to the model, transform, and criterion variables the classes or functions to be used for the specific model at hand.

- Q: Do the result change using the *default* MelSpectrogram transform or the *customized* one with different window, fft, and mel parameters?
- Repeat the processing, training and test the results. Consider changing the input representation, the network architecture, or the network hyper-parameters
- Plot the training loss versus the number of iterations.
- Look at the prediction performance and annotate the results for the comparison with the 1D model in Phase 2.

Phase 2. Model definition, training, and testing (end-to-end)

As a first example we will use **the raw audio data** and a 1D convolutional neural network. Usually more advanced transforms are applied to the audio data, however CNNs can also be used to process the raw data.

Task: End-to-end 1D network model. Build a network model with architecture defined as M5 in the paper Very deep convolutional neural networks for raw waveforms. The figure below shows a graphical representation of the M3 architecture and the structure of the additional layers for building the M5 architecture.

An important aspect of models processing raw audio data is the receptive field of their first layer's filters. This model's first filter is length 80 so when processing audio sampled at 8kHz the receptive field is around 10ms (and at 4kHz, around 20 ms). This size is similar to speech processing applications that often use receptive fields ranging from 20ms to 40ms.

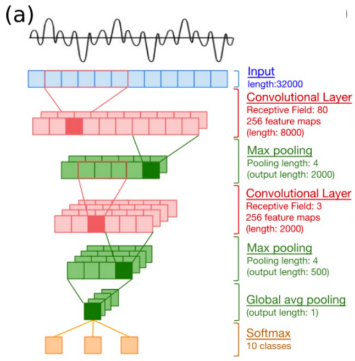


Fig. 1: (a) The model architecture of M3 (Table 1). The input audio is represented by a single feature map (or channel). In each convolutional layer a feature map encodes activity level of the associated convolutional kernel. Note that the number of feature maps doubles as temporal resolution decreases by factor of 4 in the max pooling layers, capped by a global average pooling. Note that a reduction by factor of 4 in our max pooling layers is equal to a 2D max pooling with stride (2x2) used in many vision networks.

M3 (0.2M)	M5 (0.5M)	M11 (1.8M)	M18 (3.7M)	M34-res (4M)
Input: 32000x1 time-domain waveform				
[80/4, 256]	[80/4, 128]	[80/4, 64]	[80/4, 64]	[80/4, 48]
Maxpool: 4x1 (output: 2000 × n)				
[3, 256]	[3, 128]	[3, 64] × 2	[3, 64] × 4	$\begin{bmatrix} 3, 48 \\ 3, 48 \end{bmatrix} \times 3$
Maxpool: 4x1 (output: 500×n)				
	[3, 256]	[3, 128] × 2	[3, 128] × 4	$\begin{bmatrix} 3, 96 \\ 3, 96 \end{bmatrix} \times 4$
	Maxpool: 4x1 (output: 125 × n)			
	[3, 512]	[3, 256] × 3	[3, 256] × 4	$\begin{bmatrix} 3, 192 \\ 3, 192 \end{bmatrix} \times 6$
	Maxpool: 4x1 (output: 32 × n)			
		[3, 512] × 2	[3, 512] × 4	$\begin{bmatrix} 3, 384 \\ 3, 384 \end{bmatrix} \times 3$
Global average pooling (output: 1 × n)				
Softmax				

Table 1: Architectures of proposed fully convolutional network for time-domain waveform inputs. M3 (0.2M) denotes a weight layers and 0.2M parameters. [80/4, 256] denotes a convolutional layer with receptive field 80 and 256 filters, with stride 4. Stride is omitted for stride 1 (e.g., [3, 256] has stride 1). [...] $\times k$ denotes stacked layers. Double layers in a bracket denotes a residual block and only occur in M34-res. Output size after each pooling is written as $m \times n$ where m is the size in time-domain and n is the number of feature maps and can vary across architectures. All convolutional layers are followed by batch normalization layers which are omitted to avoid clutter. Without fully connected layers, we do not use dropout [14] in these architectures.

Task: Data pre-processing. For the waveform, we define a **transform function** to downsample the audio (see torchaudio.transforms.Resample) for faster processing without losing too much of the classification power.

- Q: Which *reduced* sample rate can be used to preserve speech information?

We don't need to apply other transformations here. It is common for some datasets though to have to reduce the number of channels (say from stereo to mono) by either taking the mean along the channel dimension, or simply keeping only one of the channels. Since SpeechCommands uses a single channel for audio, this is not needed here.

Training and testing.

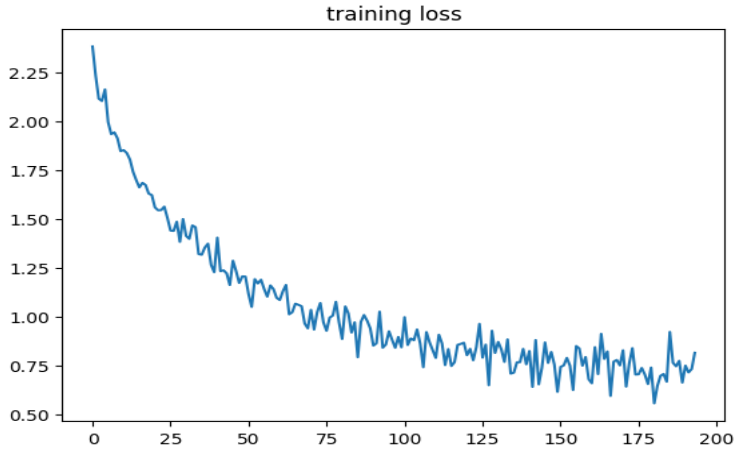
Finally, we can train and test the network. The network will be tested after each epoch to see how the accuracy varies during the training.

The network should be more than 65% accurate on the test set after 2 epochs, and 85% after 21 epochs.

Note. We will use the same optimization technique used in the paper, an Adam optimizer with weight decay set to 0.0001. If the number of epoch is greater than 20, an optimizer can be added in order to decrease the weight decay.

Task: training and testing. Train and test the network model. Assign to the model, transform, and criterion variables the classes or functions to be used for the specific model at hand.

- Plot the training loss versus the number of iterations.
- Look at the prediction performance and annotate the results for the comparison with the 2D model.



Audio data augmentation

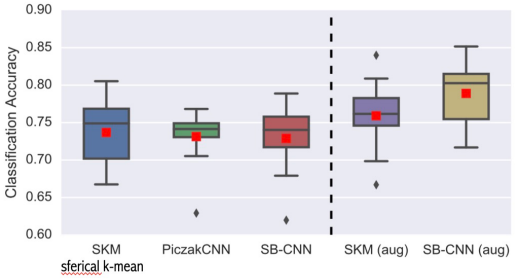
It is interesting to investigate what we can achieve with data augmentation, how does it affect the training and recognition, and how does it look (and sound) like.

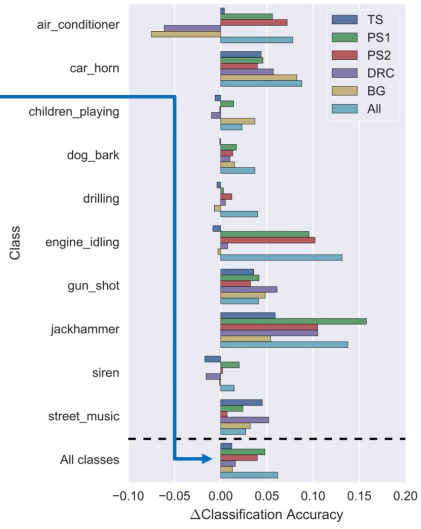
Torchaudio effects and transforms (https://pytorch.org/audio/stable/transforms.html) such as time stretch, frequency masking, and time masking (or pitch scale) are commonly used for data augmentation. See for example the Pytorch tutorial on Audio Data Augmentation and the Torchaudio Transforms documentation for examples on audio augmentation.

- H: Try to implement the Specaugment method proposed in the *"Specaugment: a simple data augmentation method for automatic speech recognition"* paper.
- Q: which are the common parameter ranges for the different transformation? See the course slides "14. Machine and Deep Learning for Audio" at page no. 82. E.g. time stretching factor from 0.81 to 1.23, etc.
- Q: Can data augmentation compensate for a reduction in the number of training data? Try to reduce the size of the dataset with and without data augmentation.

Effects of data augmentation

- Gain in data augmentation
  - TS: 0.81, 0.93, 1.07, 1.23
  - PS1: −2, −1, 1, 2 semitones
  - PS2: −3.5, −2.5, 2.5, 3.5
  - DRC: music, film, speech, radio
  - BG noise: weight [0.1–0.5] workers, traffic, people, park





Salamon et al., 2016. "Deep VNN and Data Augmentation for Environmental Sound Classification"

Audiomentations library: https://iver56.github.io/audiomentations/