Understanding audio data and spectrogram features

Since my work will be related to waveforms and audio track, I decided to read something about it.

In audio processing, it's common to use `.wav` audio codec, which is lossless (without compression at all). Each Python audio library gives you some metadata about the recording, such as the sampling rate, that is how many captures per second you have. The duration of the track can be obtained dividing the length of the sample by the samplinge rate. Since audio is an array of amplitudes, we can plot it in the time domain.

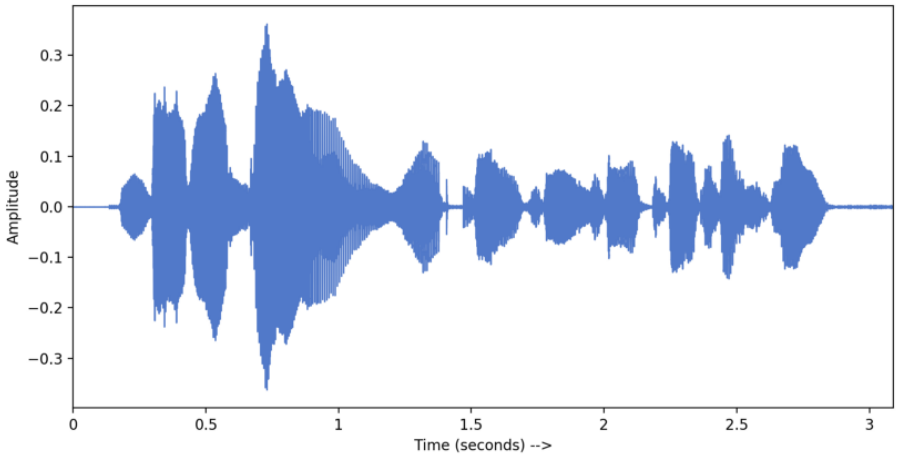


Figure 1: Time domain

When you plot the audio, you get some knowledge about the loudness of the audio recording, but to understand it you need to transform it in the frequency domain.

We can make this transformation by using the FFT algorithm. Usually, human speech has the main frequency components between 0 and 1KhZ.

The frequency domain gives us information about the frequencies of the track, but nothing about the time, so that if we have a sentence we can't say which word comes first. A spectrogram is a *visual represenation of frequencies of a given signal with time*. One axis represents time, the second one frequency, while the color of each point is the aplitude of the observed frequency at a particular time.

Usually, we need to break the audio signal into smaller frames (windows) and calculate DFT for each windows. Windows are in order and overlapped, so that we won't lose any frequency. For a typical speech recognition task, the window's size is between 20ms and 30ms (humans can't speak more than one phoneme in this windows, so that we can recognize all of them).

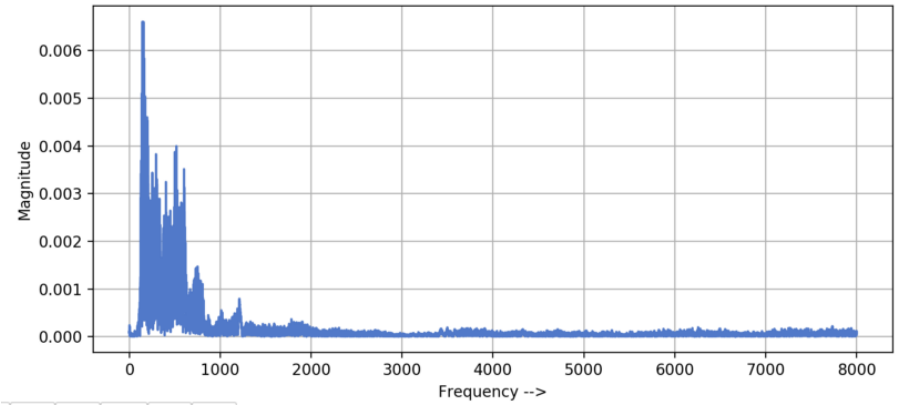


Figure 2: Frequency domain

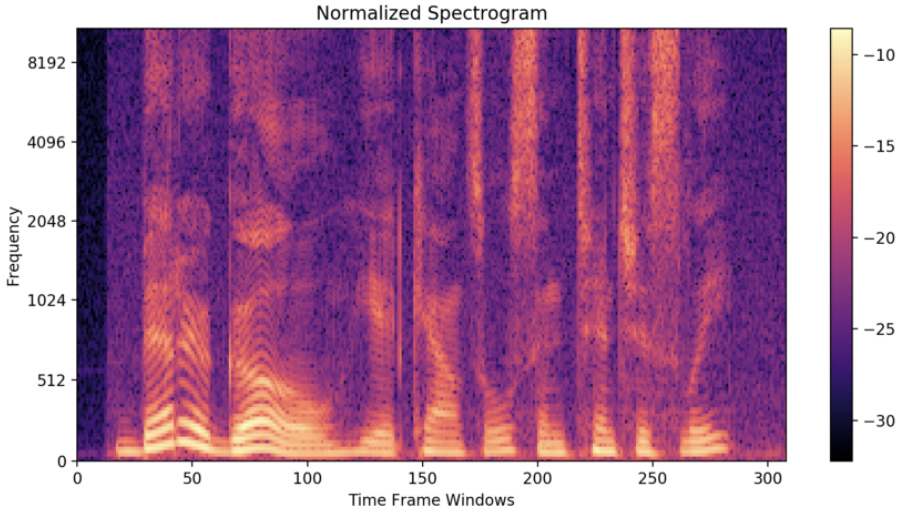


Figure 3: Spectrogram

A spectrogram shows frequencies in linear scale but our ear can discriminate lower frequencies more than higher frequencies. So, we transform the spectrogram's amplitudes from linear scale to Mel scale. Mel scale aims to mimic the non-linear human ear perception of sound. The resultant spectrogram is called Mel Spectrogram. The conversion into the mel scale is performed using mel filters. Frequencies on the linear scale are multiplied with mel filters to get frequencies on the mel scale.

We also perceive loudness on a logarithmic scale. So, we transform amplitude into the decibel scale, getting the melscale.

There are two main ways to build a simple NN to work with audio:

1. You can build it using 1d convolutional layers on the raw audio track;
2. You can build it using 2d convolutional layers on the spectrogram of the track.

The second one is interesting as it makes the neural network more explainable (dealing with images is always better for explainability), but some resources [4] demonstrate that deep neural networks with many convolutional layers can outperform networks based on log-mel scale.

References: [1], [2], [3], [4]

Dataset

The dataset I've found is [1]. It consits of 35 basic english words repeated thousands of times. They are chose for many different reasons: there are all the one-digit numbers, some usual words and some commands to make robots and drones move, such as *forward* and *follow*. Here is the list of words, followed by their number of occurances.

Word	Number of Utterances
Backward	1,664
Bed	2,014
Bird	2,064
Cat	2,031
Dog	2,128
Down	3,917
Eight	3,787
Five	4,052
Follow	1,579
Forward	1,557
Four	3,728
Go	3,880
Happy	2,054
House	2,113
Learn	1,575
Left	3,801
Marvin	2,100
Nine	3,934
No	3,941
Off	3,745
On	3,845
One	3,890
Right	3,778
Seven	3,998
Sheila	2,022
Six	3,860
Stop	3,872
Three	3,727
Tree	1,759
Two	3,880
Up	3,723
Visual	1,592
Wow	2,123
Yes	4,044
Zero	4,052

Unfortunately, there are no tracks under a lables as *unknown*. Another exepriment we could work on is choosing some of the words below and make the target words, while the others are *unknown*. That would be more coherent to the "*ehi*

siri" idea.

The advantage of using an already built dataset is obvious: words pronunciation are different from each other, there are many kinds of data augmentation in it (for instance, some background noise) and useful API to start working on it.

Here is a snippet of the code to work with the dataset:

```
from torchaudio.datasets import SPEECHCOMMANDS
import os


class SubsetSC(SPEECHCOMMANDS):
    def __init__(self, subset: str = None):
        super().__init__("./", download=True)

        def load_list(filename):
            filepath = os.path.join(self._path, filename)
            with open(filepath) as fileobj:
                return [os.path.normpath(os.path.join(self._path, line.strip())) for line in

        if subset == "validation":
            self._walker = load_list("validation_list.txt")
        elif subset == "testing":
            self._walker = load_list("testing_list.txt")
        elif subset == "training":
            excludes = load_list("validation_list.txt") + load_list("testing_list.txt")
            excludes = set(excludes)
            self._walker = [w for w in self._walker if w not in excludes]


# We don't need the validation!
train_set = SubsetSC("training")
test_set = SubsetSC("testing")
```

Each element has some information about the track. For instance, you get the waveform (which you can plot using `matplotlib`), the sample_rate (16KhZ), the labels (which word is pronunced) and some IDs for the speaker and for the track into the dataset.

There are some basics transformation we can apply to facilitate the training:

- We can downsample the audio for faster processing, for instance from 16KHz to 8KHz. In this case we are loosing some information about the audio, but it's still good enough to analyze it;
- We can reduce the number of channels, if there are more than one. All the tracks in this case are single channels; otherwise, we could mix the channels through an average, or simply delete one of them.

```
new_sample_rate = 8000
sample_rate     = 16000
transform = torchaudio.transforms.Resample(orig_freq=sample_rate,
                                           new_freq =new_sample_rate)
```

Since each words needs an ID, we can use its position inside an array made of the sorted words. This is done defining the array `labels`, and the two simple functions `label_to_index` and `index_to_label`.

Then we can create the two loaders using the DataLoader class in `torch.utils.data`.

References: [1], [2]

The architecture

The architecture is based on [1]. The paper is quite old (2016, which was much time ago for machine learning), but I think it can be useful to dive into the topic. I started studying machine learning with the "School in AI", and I still have to master some basic concepts of it, such as the convolutional layers. Furthermore, the paper itself says that it outperforms networks based on spectrogram of audio.

The network I adopted is M5, which is implemented in Keras in [2].

```
class M5(nn.Module):
    def __init__(self, n_input=1, n_output=35, stride=16, n_channel=32):
        super().__init__()
        self.conv1 = nn.Conv1d(n_input, n_channel, kernel_size=80, stride=stride)
        self.bn1 = nn.BatchNorm1d(n_channel)
        self.pool1 = nn.MaxPool1d(4)
        self.conv2 = nn.Conv1d(n_channel, n_channel, kernel_size=3)
        self.bn2 = nn.BatchNorm1d(n_channel)
        self.pool2 = nn.MaxPool1d(4)
        self.conv3 = nn.Conv1d(n_channel, 2 * n_channel, kernel_size=3)
        self.bn3 = nn.BatchNorm1d(2 * n_channel)
        self.pool3 = nn.MaxPool1d(4)
        self.conv4 = nn.Conv1d(2 * n_channel, 2 * n_channel, kernel_size=3)
        self.bn4 = nn.BatchNorm1d(2 * n_channel)
        self.pool4 = nn.MaxPool1d(4)
        self.fc1 = nn.Linear(2 * n_channel, n_output)

    def forward(self, x):
        # x.shape = (N, 1, 8000)
        x = self.conv1(x)
        # x.shape = (N, 32, 496)
        x = self.bn1(x)
        # x.shape = (N, 32, 496)
        x = F.relu(x)
```

```
        # x.shape = (N, 32, 496)
        x = self.pool1(x)
        # x.shape = (N, 32, 124)
        x = self.conv2(x)
        # x.shape = (N, 32, 122)
        x = self.bn2(x)
        # x.shape = (N, 32, 122)
        x = F.relu(x)
        # x.shape = (N, 32, 122)
        x = self.pool2(x)
        # x.shape = (N, 32, 30)
        x = self.conv3(x)
        # x.shape = (N, 64, 28)
        x = self.bn3(x)
        # x.shape = (N, 64, 28)
        x = F.relu(x)
        # x.shape = (N, 64, 28)
        x = self.pool3(x)
        # x.shape = (N, 64, 7)
        x = self.conv4(x)
        # x.shape = (N, 64, 5)
        x = self.bn4(x)
        # x.shape = (N, 64, 5)
        x = F.relu(x)
        # x.shape = (N, 64, 5)
        x = self.pool4(x)
        # x.shape = (N, 64, 1)
        x = F.avg_pool1d(x, x.shape[-1])
        # x.shape = (N, 64, 1)
        x = x.permute(0, 2, 1)
        # x.shape = (N, 1, 64)
        x = self.fc1(x)
        # x.shape = (N, 1, 35)
        x = F.log_softmax(x, dim=2)
        # x.shape = (N, 1, 35)
        return x
```

Let's see how each layer changes the dimension of the input.

- `nn.Conv1d`: Input: $(C_{\text{in}}, L_{\text{in}})$; Output: $(C_{\text{out}}, L_{\text{out}})$ where

$$C_{\text{out}} = \text{nChannel}$$

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} + 2 \cdot \text{padding} - \text{dilation} \cdot (\text{kernelSize} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

- `nn.BatchNorm1d`: Output has the same shape as the Input.
- `nn.MaxPool1d`: Input: (C, L_{in}); Output: (C, L_{out}) where

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} - \text{kernelSize}}{\text{kernelSize}} + 1 \right\rfloor$$

- `F.log_softmax`: It is equivalent to a softmax followed by a logarithm (`log(softmax(x))`), but numerically more stable. The `dim` attribute tells on which dimension the operation is done.

The input of the network is a waveform of shape `(1, 8000)` (the first dimension should be the batch size, but we can ignore it). Using the above expressions, we can write the shape of `x` after each layer of the network. The shape of the output is clearly `(1, 35)`, corresponding to the number of different labels.

Using some simple functions, the number of parameters to learn is 26915.

References: [1], [2].

Training

I'm going to use Adam as optimizer, since it's the same used in [1]. The learning rate, as an hyperparameter of the network, may change the overall accuracy: I'm going to do some tests, so that I can choose the base value.

```
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

The loss function used is the Negative Log-Likelihood funciton as, after some tests, it outperforms the Cross-Entropy.

We do the train for `n_epoch` times; as the training goes on, we print some information about the loss function. In particular, as we have saved all the losses in a list, we can show them into a plot.

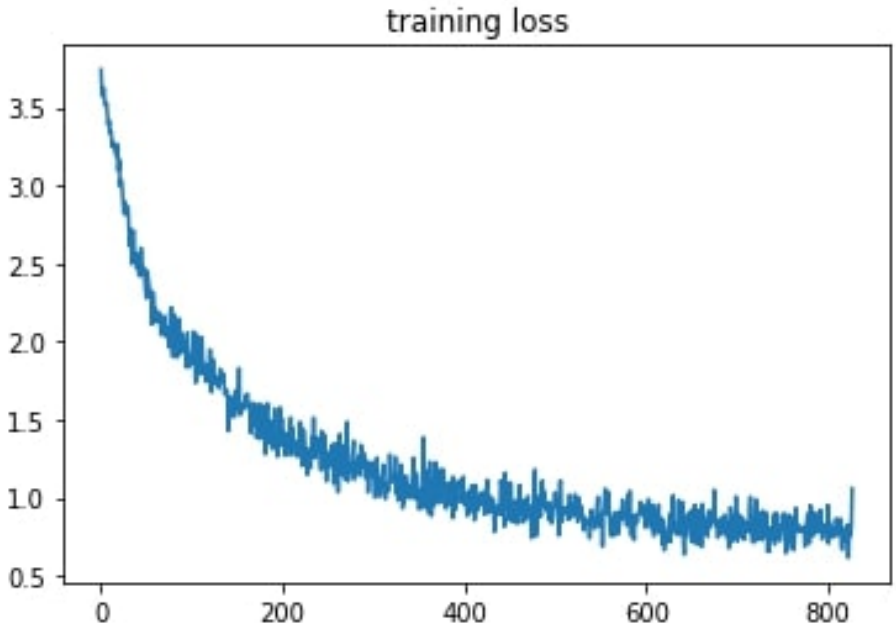
We also test the network on the `test_loader` at the end of each epoch, so that we can see the overall result (this score has no impact on the following epochs).

Some tests show us that two epochs are enough for the network to converge.

References: [1]

Results

Here is a plot of the loss function decreasing after two epochs:

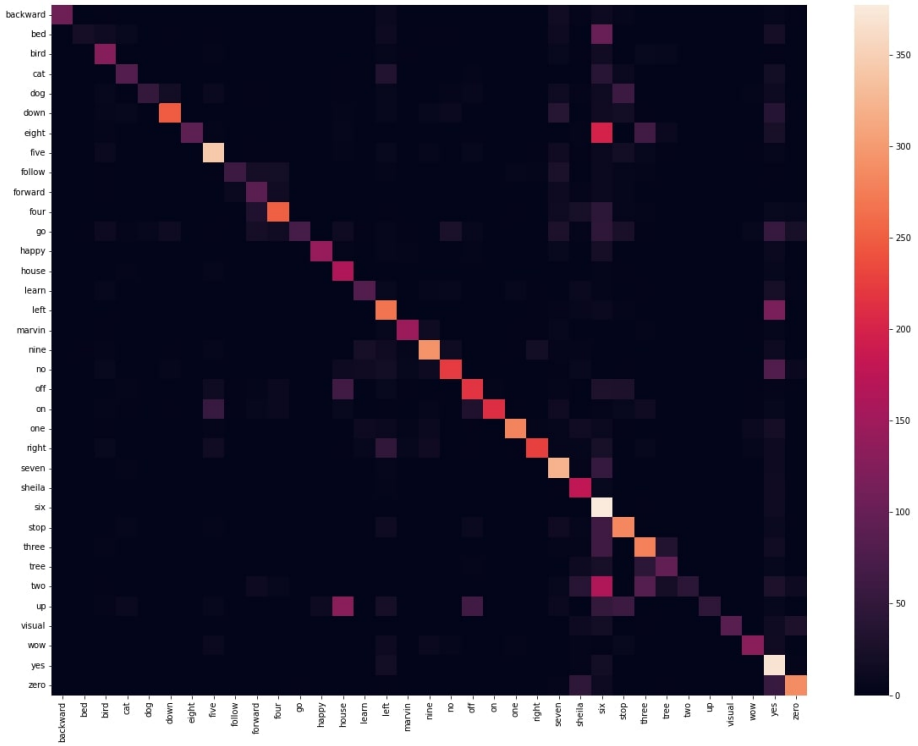


The accuracy of the network is 71%. This result is not excellent, but we need to consider some limitations of the work:

- We haven't used any particular technique to make the network *stronger*. We have many layers inside it, and, as we know, there may be many problems related to gradient descent.
- Some words may be similar to each other, making harder for the network to guess them.

Since the labels were 35, a random guess on the label has 2.86% of probability to be correct. As our percentage is way higher, we can say our network works properly.

Here is an heatmap of the classification:


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
350
300
250
200
150
100
50
0