Audio Command Classification Problem

Ali Mousazadeh
Politecnico di Torino
Student id: s308284
s308284@studenti.polito.it

***Abstract*—This report introduces a possible approach to the audio command classification problem. The proposed method consists of extracting the Mel spectrogram of each audio sample, then feeding it to a Long-Short Term Memory layer stacked on top of two one-dimensional convolutional layers. This method vastly outperforms the naive baseline and reaches an accuracy of 98.9% on the evaluation data.**

I. Problem overview

The proposed competition is a classification problem on a provided dataset consisting of audio recordings of speech commands, alongside other data including their age, gender, native and current language spoken, and self-reported fluency. The task is to classify the command spoken by each person. The dataset has 7 different possible commands: *activate music, change language, deactivate lights, decrease heat, decrease volume, increase heat, increase volume*. We will map these labels to integers from 0 to 6 for easier inspection. Although the dataset does include an *action* and *object* column, upon further inspection we see that only these 7 combinations are present. The dataset is divided into two parts:

- a *developement* set, containing 9854 labeled recordings.
- an *evaluation* set, containing 1455 recordings without labels.

We will use the development set to build a classification model and then test the model's accuracy on the evaluation dataset on the leaderboard.

The distribution of labels in this dataset is somewhat unbalanced, as we see in Fig. 1. We will explain the balancing solution we tried in the model selection section.

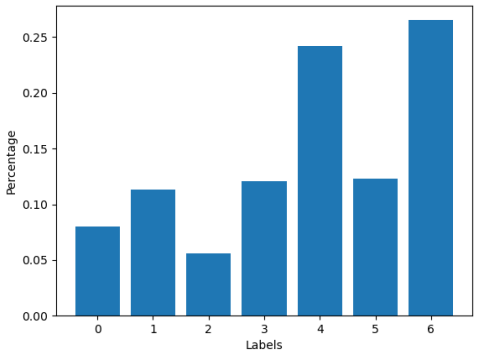


Fig. 1. Distribution of labels in the development dataset.

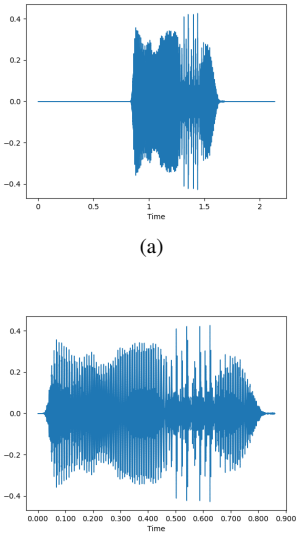
Most of the audio signals are between 2-3 seconds. There are some audio signals over 10 seconds, but there are few of them in the dataset.

II. Proposed approach

A. Preprocessing

In this section, we discuss the preprocessing methods used for feature extraction.

Before extracting higher-level features from the audio files, we inspect several of them in the time domain. We see that many audio files have a period of silence at the beginning and the end. Therefore we use *librosa.effects.trim* from the *librosa* library in Python to trim the audio signals. This function receives the audio signal and a decibel threshold, and it trims the audio signal accordingly. Fig. 2 demonstrates an example of this trimming effect. We use the cut-off decibel threshold of 25. This is a bit over the commonly used values of 15-20 decibels, but we have chosen 25 because preserving as much of the audio as possible is our primary objective, and the model will have no problem learning from an audio signal with a handful of zeros at both ends. With the same philosophy, we have not downsampled the sampling rate of 22050 Hz of the audio. Trimming will also reduce the number of features that we'll have to consider later.



(a)

(b)

Fig. 2. Example of an audio signal before (a) and after (b) trimming.

After applying the trimming effect to the audio signals, we use *librosa.feature.melspectrogram* to extract the Mel spectrograms. This function does a Fourier transform of sliding windows on the audio signal, measuring the strength of each frequency inside the window. It produces a 2-dimensional frequency-time plot representing the strength of frequencies over time. It also receives some parameters that will affect the produced spectrogram, including the number of Mel bins, max frequency, size, and hop length of the sliding windows. These could be considered as some of the possible hyperparameters that need tuning, however, they are much easier to tune compared to hyperparameters regarding the model. In practice, we tried different values for these parameters and picked the ones that would produce the clearest image containing the most information. The parameters that were changed from the default settings were $n_{mels} = 200$, $f_{max} = 8192$ Hz, and $hop_{length} = 32$. A higher number of Mel frequency bins increases the resolution of the frequency domain, while a lower number for the hop length increases the resolution of the time domain. While typically the value for max frequency is chosen to be half the sampling rate, we chose a lower value of 8192 Hz because, upon closer inspection of various Mel spectrograms, we saw that the top section of the plot corresponding to higher frequencies had negligible power. After computing the Mel spectrogram, *librosa.power_to_db* was used to logarithmically scale the data, since the human perception of different frequencies is logarithmic. Fig. 3 shows an example of the Mel spectrogram with the parameters described in this section after scaling.

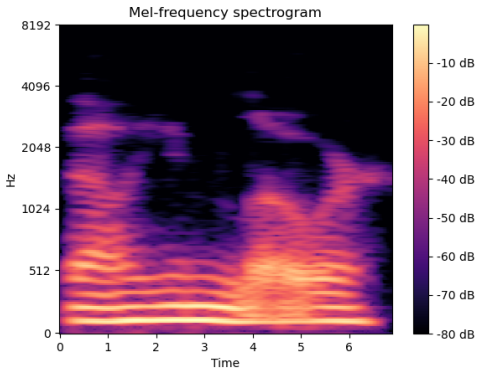


Fig. 3. Scaled Mel spectrogram of an audio signal in the development dataset with $n_{mels} = 200$, $f_{max} = 8192$ Hz, and $hop_{length} = 32$.

Since the Mel spectrogram has a variable length in its second dimension, some sort of preprocessing would need to be conducted to fix the length of the second dimension. Either by padding all the data to the same length, or taking advantage of averaging or feature compression on each individual audio's spectrogram. However, all these methods sacrifice or dilute the available information in order to enable the use of a simpler model. To overcome this limitation, we take advantage of convolutional neural networks in combination with Long-Short Term Memory layers which are well suited to the task of variable-length samples. We will discuss the selected model in the next section.

It should also be noted that non-audio features were not used in the final model, as they seemed to offer little or no improvement on the evaluation accuracy.

B. Model selection

In order to balance the Cross-Entropy loss function given the fact that label distributions are unbalanced, we flip these weights such that the labels with a lower presence get an increased weight in the cost function and the labels with a higher presence get a lower weight. By this adjustment, the model will learn to pay equal attention to all labels.

We take advantage of the **Pytorch** library and a CUDA-enabled RTX 3070 GPU for training. We have used two convolutional-batch normalization-max pooling layers, the output of which is fed to a *Long-Short Term Memory* (**LSTM**) [1] layer. We use the convolutional and max pooling layers to downsample the input data in the time domain. However, the pooling process is not done aggressively, so as to leave the time domain big enough for the LSTM to process. Fig. 4 demonstrates the steps of the proposed model in processing a batch of input.

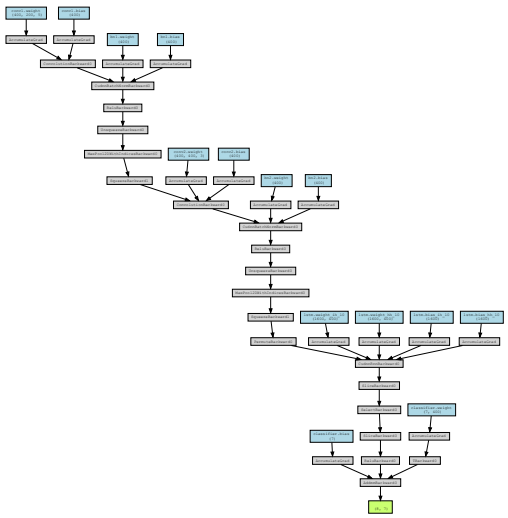


Fig. 4. The proposed model.

C. Hyperparameters tuning

In this section, we discuss the steps taken in order to tune the hyperparameters of the model.

For some of these hyperparameters such as batch size and optimization algorithm, rules of thumb and overall guidelines were used. Lower batch size will take less advantage of the underlying vectorization processes and the parallelism abilities of the GPU and slow down the learning process. However, it is also known to improve model generalization. As a balance between training time and generalization power, a batch size of 8 was selected after multiple tests on the evaluation dataset.

The AdamW [2] optimization algorithm was used to minimize the cross-entropy loss function, as it is an improvement to the Adam [3] algorithm and it is shown to better generalize to unseen data.

Max pooling layers were chosen instead of average pooling due to the background of the spectrogram being black. In such cases where the absence of information is marked with low values, average pooling dilutes the information while max pooling will select the most informative part of the input. The size of kernels, stride, number of channels, and number of neurons inside the LSTM were all tuned on the evaluation data.

Normally it would be better to divide the development dataset into train and validation and tune the hyperparameters on the validation data. However, for this particular task, the leaderboard score was the main tool for tuning. The reason for this decision was that upon many tests, the data split into development and evaluation did not seem to be random. Normally, one would expect the accuracy for train, validation, and testing to go from higher to lower in this order. However, in many iterations, the leaderboard accuracy would be higher than the validation accuracy. This could mean that the evaluation dataset is somewhat "easier" in a sense compared to the development dataset. Since the final goal was to get the highest evaluation accuracy possible, in this particular case, dividing the development data into train and validation for turning was not the best approach, despite it being a good approach in general for many tasks.

For this same peculiar phenomenon, rather than employing methods to avoid overfitting such as early stopping or regularization, the model was permitted to train for an absurdly huge epoch number of 100. However, an exponential decay learning rate scheduler was used to limit the changes in the model in the later epochs, by reducing the learning rate by 25% every 5 epochs.

III. Results

In this section, we will discuss the performance of the selected model explained in the previous section after tuning its hyperparameters, demonstrated in Fig. 4.

The model reaches the accuracy of 100% on the development data and the accuracy of 98.9% on the evaluation data.

IV. Discussion

Another way to improve the accuracy of the evaluation dataset could be to stack the predictions of multiple CNNs before feeding them to the LSTM layer. Or taking advantage of residual layers after the LSTM for added computing power.

It should be noted that in order to streamline the tuning phase of the project, the audio loading, trimming, and preprocessing parts were done in different notebooks and their results were saved in the format of Pandas DataFrames on the disk. In this way upon restarting the machine or kernel, the already preprocessed features could be loaded into the training notebook very fast which increased the speed of the training iterations significantly. This allowed for rapid tuning of the hyperparameters and the model. However, the preprocessed data stored on the drive is several gigabytes big in size. This was only doable because the datasets for this task were relatively small. Otherwise, the audio files would have to be loaded and processed one by one during training. A cache system could be implemented to increase the speed of training despite individual loads and preprocesses done, however simply processing everything in bulk and saving it on the disk was the more convenient choice for this task.

Since it is infeasible to upload the processed data on the portal and it is a requirement that all the code be merged into a single notebook, I should note that the preprocessing and loading parts of the code would take a long time to run on a less powerful machine, since these parts are done in bulk at the start of the code rather than during the training process. If requested, I can share the preprocessed DataFrames in some other way most convenient for the reviewer.

It should also be noted that due to the random initialization of the model weights and random permutations of the data in each batch, the accuracy of the evaluation dataset may differ very slightly from the leaderboard score if retrained, which should be negligible. The state dictionary of the model with the highest accuracy will also be provided alongside the report, and the predictions can be checked for reproducibility if desired.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.