Short Programming Project
Spoken Language Identification

Documentation and Specification

Robbin de Groot (s3376508)
Supervisor: Kerstin Bunte
February, 2020

Contents

1 Introduction

Spoken language identification is the process of determining the human spoken language from audio. This document will outline an attempt at implementing such an identifier, and the observations and conclusions that can be drawn from it. We will describe the resources used and the basis on which decisions were made in detail in the sections below.

2 Project Specification and Requirements

Before we define the implementation, it is important we define the project and its scope. This project will consist of 2 separate, integrated applications. A Convolutional Neural Network (CNN) and a Website. Each of them were developed separately and as such, the network can be extracted for external use. Below, we will further define the development plan and the components that make up the project.

2.1 Development plan

Before the start of development, we define how we plan to develop all components. The project is developed according to a waterfall model. Since this project is not primarily software oriented, we will not be working using agile/SCRUM methods. To keep track of progress and make a planning, a Trello board was made, along with a table indicating the time reserved for every step of the process. This table was included in the proposal document and will hence not be repeated here. According to this plan, we will start development.

2.2 CNN Classifier

A neural network is used to classify audio samples into languages. The input consists of audio samples in `mp3` format, the output is one of 6 well-known spoken languages: Dutch, English, German, French, Italian and Spanish. In addition, only male samples were used, as the addition of female voices adds more variation in the data. We applied these restrictions to save on computation time and complexity, as we are not in the possession of large-scale computational resources. From this specification, we can design a network architecture.

The network architecture is heavily inspired by a paper by Sarthak, Shukla, and Mittal [9]. Their paper outlines several methods towards classifying spoken language. We opt for the 1-dimensional method using a standard CNN architecture. More on this in section 4 on page 4.

2.3 Dataset

The network is trained on data provided by the Mozilla Common Voice project [4]. Common Voice is a free, crowd-sourced project that collects samples of spoken voices by human participants. This dataset is large enough to train a small network. The crowd-source aspect comes with some "bad clips" that potentially yield poor results [2]. The alternative would be a dataset by Topcoder [10], which was considerably smaller, but more diverse in the languages offered. This made us decide on Mozilla Common Voice, as it is better suited and designed for the problem.

2.4 Website

We use the website to obtain extra, custom training data for the network as well as to test the network capabilities. The interface will allow users to record audio and send it to a server for classification. The server may save the sample to use for re-training later on, as this data better suits the application. We will elaborate on this choice in section 5 (page 5).

3 Technology Stack

The following resources were used in the building of this project

- **PyTorch/LibTorch**: a `C++` frontend of the LibTorch framework for Machine Learning and Neural Networks. [5]
- **OpenMP3**: a `C++` library for decoding `mp3` files into frames and metadata. [1]
- **Flask**: a `Python` microframework for developing website and web applications. [8]
- **PyDub**: a `Python` library based on `ffmpeg` that aids in audio processing and conversion.
- **Bootstrap**: a `CSS` collection of stylesheets that make developing of mobile first, cross browser websites easier and better-looking.
- **Mozilla Common Voice**: an audio dataset for a large range of languages. [4]

4 Network Implementation

4.1 Description and Construction

To classify the languages, a convolutional neural network (CNN) was used. In this document, we will not elaborate on the structure of this kind of network. CNNs are commonly used for pattern recognition and classification, as they save on network size required for a deep-MLP. Hence we choose this kind of network. We base our implementation and structure on a paper by Sarthak, Shukla, and Mittal [9].

The first major design decision is the input shape. `mp3` files are split into 'frames', short, equally sized segments of (usually) 1152 samples. Depending on the sample rate – typically 44100 Hz – the length of a frame varies. Our reference paper uses inputs of $10s$ [9]. However, due to complexity constraints mentioned in section 2, we use $2.5s$ to reduce the complexity and memory usage.
We also consider stereo or mono samples. We choose for mono sound as it, once again, reduces complexity and because we don't want the network to 'learn' a sense of positioning. To explain, stereo audio will pick up a left and right sound. We do not want the network to adjust for this asymmetry. Therefore, we only use mono audio, converting stereo to mono when applicable.

To determine the input shape, we need to analyse the structure of the input. The aforementioned paper mentions that based on the Nyquist-Shannon theorem, the useful frequencies (0-300 Hz) are all captured when sampling a signal at 8 kHz. Therefore, we resample the samples (again, sampled at 44100 Hz) at just over 8 kHz, by taking every 5th sample from a given file. This nets a sample rate of $\frac{44100}{5} = 8820$ Hz.
We will now calculate the number of inputs such that we know the input shape. A `mp3` frame contains 1152 samples, at 44100 Hz. This means that a frame is $\frac{1152}{44100} \approx 0.026s$ long. We mentioned that we aim for a total input length of around $2.5s$, therefore, we will need $\frac{2.5}{0.026} \approx 95.7 \approx 96$ frames. Given that we 'compress' each sample by 5, we have

$$\frac{96 \cdot 1152}{5} = 22118.4$$

samples in each input. Since we compress by simply taking every 5th sample of a file, the calculated value is rounded down. We therefore have 22118 samples per input.

The network size is a parameter we can play with. As mentioned before, we largely copy the input shape from our reference paper. We reduce the amount of filters by a factor of 8. Initially, a reduction of 4 was applied, however this yielded a severely overfit network (Train accuracy of $\sim 95\%$, test accuracy of $\sim 23\%$). A further reduction that 8, say 16 or even more, has not been tried, but is worth considering for smaller amounts of data. At this point, however, the amount of channels gets small to a point where the network may not be able to detect enough patterns. In the end, a reduction factor of 8 was chosen, the results of which can be found below.

The final structure of the network is presented in Table 1. We present to this network a Tensor of shape $(1, 21188)$. 1D convolution filters will be presented in the following format $< \text{channels}, \text{size}, \text{stride} >$. Every convolution layer is followed by a ReLU output function layer, then by a batch normalisation layer.

Layer Type	Kernel Shape	Output Shape
1D Conv	$< 16, 3, 3 >$	$(16, 7372)$
1D Conv	$< 16, 3, 1 >$	$(16, 7370)$
1D MaxPool	$< 3 >$	$(16, 2456)$
1D Conv	$< 16, 3, 1 >$	$(16, 2454)$
1D MaxPool	$< 3 >$	$(16, 818)$
1D Conv	$< 32, 3, 1 >$	$(32, 816)$
1D MaxPool	$< 3 >$	$(32, 272)$
1D Conv	$< 32, 3, 1 >$	$(32, 270)$
1D MaxPool	$< 3 >$	$(32, 90)$
1D Conv	$< 64, 3, 1 >$	$(64, 88)$
1D MaxPool	$< 88 >$	$(64, 1)$
Fully Connected	$< 64, 6 >$	$(6, 1)$
Softmax	$< 6 >$	$(6, 1)$

Table 1: Structure of a CNN used to classify spoken language as samples of input size $(1, 22118)$.

4.2 Training

The network was trained using the *K fold cross validation* training method, on 2000 `mp3` files per language. That is 12000 files combined. The K value used was 10, which yielded a sufficiently large test subset. We choose this method as it allows us to observe the performance of the network during training. We used Adam as the optimizer, with an annealing learning rate between $1e-4$ and $1e-5$ (and using PyTorch defaults everywhere else) [6]. The loss was calculated using the *Natural Log Loss* [7], which perfectly complements the output we require.

4.3 Evaluation

The performance of the network is evaluated by running the classifier on a test set. This test set does not intersect with the training set. The excluded set is used calculate the loss of the network and to test the accuracy. The results are plotted in graphs found in figures 1 and 2 respectively.
Testing the accuracy of a network on a set that the network has 'seen' can give the impression of making a good fit, but does not show of overfitting. For this reason, we use the aforementioned test set. This set has been manually tested in between some epochs. This means we have no intermediate data of the results of these tests, only the final data. The final, average natural log loss over the train set is 0.155 (standard deviation of 0.33), and the classification accuracy is 95.66%. The average loss over the test set is 4.24 (standard deviation of 5.54), and the classification accuracy of 39.16%.

5 Server Implementation

Since the main goal of this project is focused on the network, the server is kept minimal in complexity. This is also the reason why we opt for a website over a mobile app of dedicated software. Websites make cross platform development trivial, as all OS and device specific functionality is implemented in the browser software. A website is also easy to deploy and maintain in comparison, as this is similar compared to how it is tested.

The server is implemented as a Flask website/webserver that allows for quick-and-dirty implementation.
The website aesthetic is also kept at a minimum. It features a "start" and "stop" button to start and stop audio recording, an audio-player to listen to the recording, and a line of text displaying the status of the classifier.

The classifier is build to a binary executable which can easily be run by the `Python`-based server back-end. The recorded sample is sent to the server in `ogg` format (by HTML convention). The back-end does some pre-processing (noise reduction) and converts the file to `mp3`. The classifier is called on the newly created

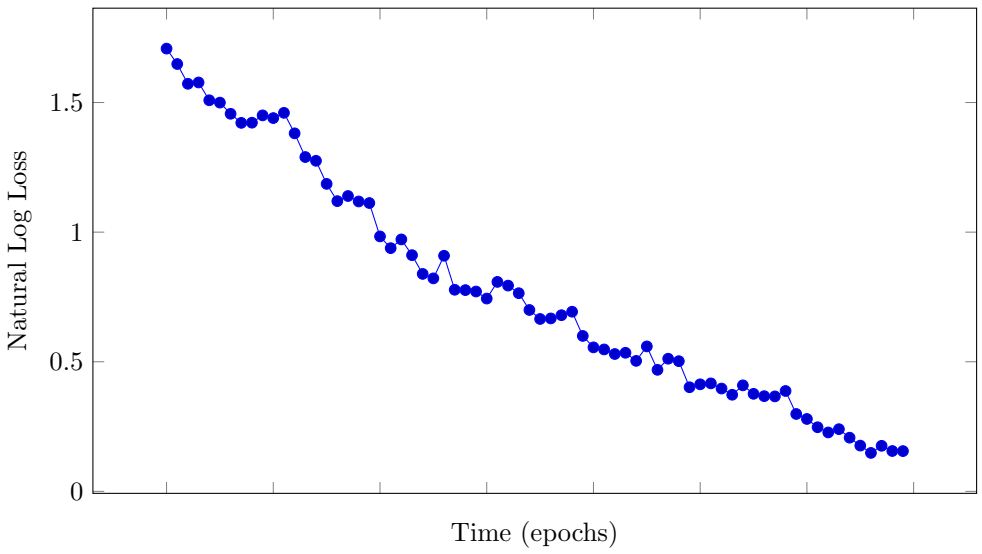

Figure 1: Average value of the natural log loss function of the output of the classifier over time trained (excluded group).

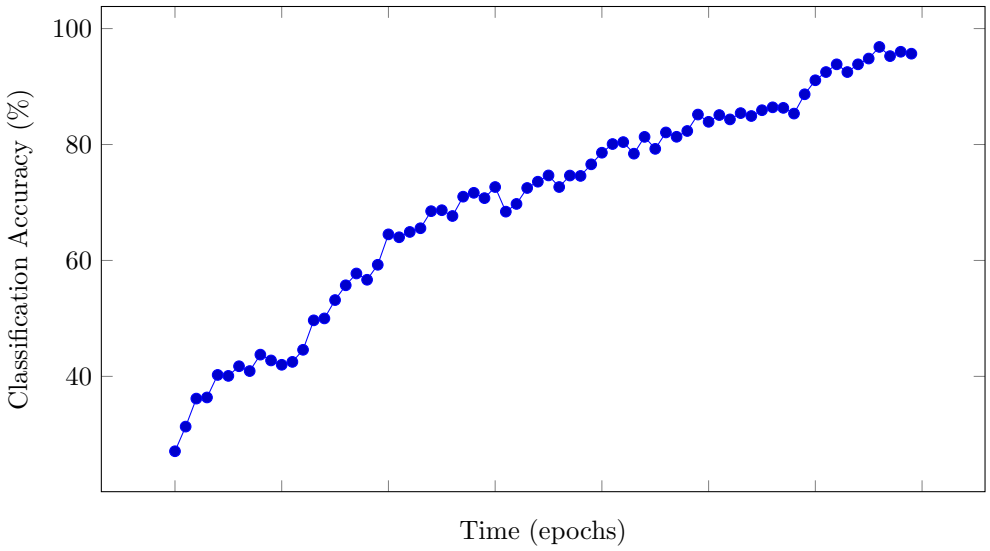

Figure 2: Training classification accuracy of the classifier over time trained.

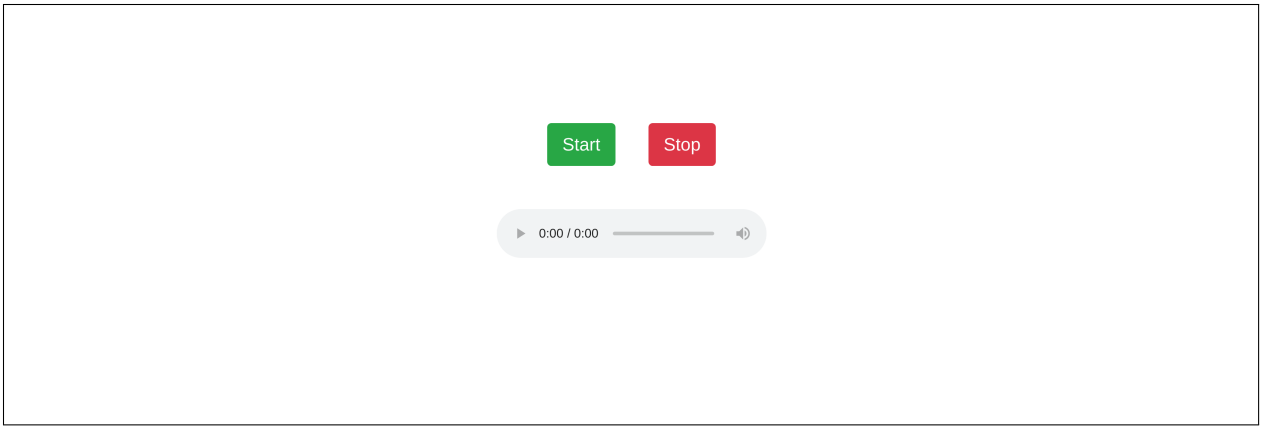

Figure 3: Screenshot of the website taken before recording an audio clip.

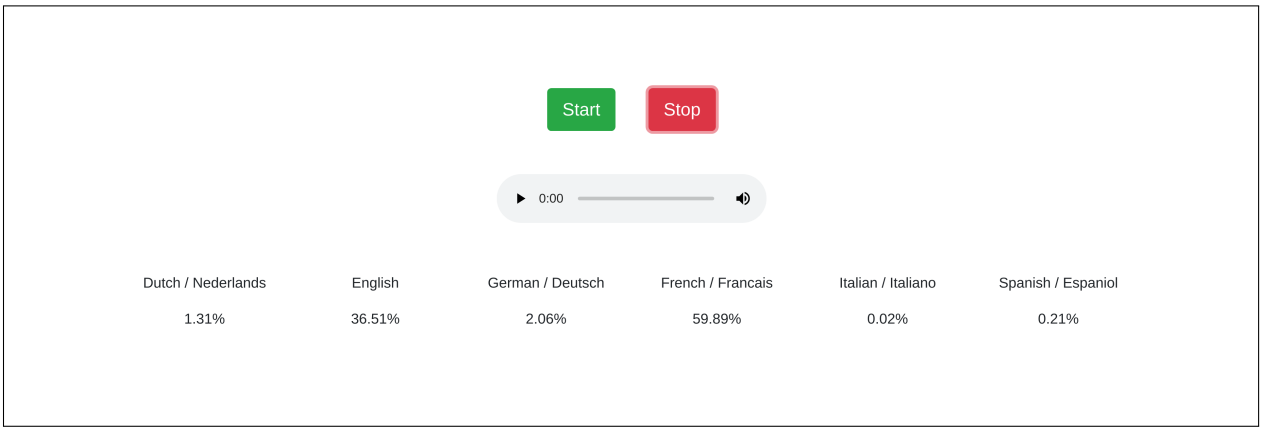

Figure 4: Screenshot of the website taken after recording an audio clip, showing the classification results made by the CNN.

`mp3` file which outputs a classification. The output is delegated back to the website for pretty-printing. An example UI before and after a run can be seen in figures 3 and 4.

6 Conclusions

From the results obtained after training a network of this category, we may conclude that SLI is a hard problem. This is not surprising, as even human brains need practice to recognise languages too. A small sized network is therefore hard to train to great accuracy (say $> 99\%$). For the application, simply approaching this value is acceptable. A small network has less learning capacity and is therefore more prone to underfitting. Therefore, in practice, a larger network is adviced.

Getting proper training data is a problem. Mozilla Common Voice comes a long way, but pre-processes the data and allows participants to send in multiple tracks. This means not all data is unrelated. Getting new training data from the same source as it will be used in is better for the application, but sadly, the scope of the project did not allow for this luxury. Training a network on existing datasets is useful in the start of deployment, as from its use, the application may collect more relevant data, which it can in turn use for retraining.

All in all, this project forms a basis for potentially applied uses of SLI in other applications. It is not of much use as a direct port, but based on the findings in this paper, a well suited, integrated application feature may

be written. A hard problem like SLI requires careful planning, and therefore, this paper should be used as a proof-of-concept.

The source code for this project can be found in a public Github repository [3].

7 Reflection

This project is not the first machine learning oriented project I have done. Many of the activities and principles involved (training strategies, familiarity with frameworks) are therefore not foreign to me. However, this project is the first where I needed to consider the application of my network. The problem of classifying languages is hard, and requires a larger network than presented to be done in practical context. However, due to limitations, certain compromises needed to be made, which requires rethinking of certain decisions.

The network went through many iterations. A first trial was done with a recurrent CNN, which would in theory converge to a solution. The idea was that the temporal dimension plays a role in human speech. However, this did not turn out to be a viable method, as the network did not have a large section of audio to train on each iteration. This is but one of many examples where a iterative development proved useful, even though a waterfall development cycle was planned.
An unforeseen problem was deployment. The server on which the network is run has a different processor architecture than the development system. This means the binary is incompatible and does not execute. Because of limited resources, the program had to be cross-compiled. To add insult to injury, the machine learning library, LibTorch, does not have a version for that processor architecture, meaning we have to compile the source of LibTorch, too.
Unexpected circumstances like this add days, if not weeks to development time. The takeaway is that a time buffer should be in place when working with a waterfall method in case deadlines are missed.

References

[1] audioboy77. *OpenMP3*. URL: `https://github.com/audioboy77/OpenMP3` (visited on 11/13/2019).

[2] user: dabinat. *Mozilla Common Voice Forum: Identifying bad clips in dataset releases*. URL: `https://discourse.mozilla.org/t/identifying-bad-clips-in-dataset-releases/43474/4` (visited on 11/17/2019).

[3] Robbin de Groot (s3376508). *Spoken Language Identification project repository*. URL: `https://github.com/RobbinDG/SLI-App` (visited on 02/10/2020).

[4] Mozilla. *Common Voice*. URL: `https://voice.mozilla.org/en` (visited on 11/17/2019).

[5] *PyTorch*. URL: `https://pytorch.org/` (visited on 11/12/2019).

[6] *PyTorch Adam Optimizer reference*. URL: `https://pytorch.org/docs/stable/%5C_modules/torch/optim/adam.html` (visited on 01/20/2020).

[7] *PyTorch Natural Log Loss reference*. URL: `https://pytorch.org/docs/stable/nn.functional.html?highlight=nll#torch.nn.functional.nll_loss` (visited on 01/22/2020).

[8] Armin Ronacher. *Python Flask*. URL: `https://palletsprojects.com/p/flask/` (visited on 11/12/2019).

[9] Sarthak, Shikhar Shukla, and Govind Mittal. "Spoken Language Identification using ConvNets". In: *arXiv:1910.04269* (Oct. 2019).

[10] Topcoder. *Marathon Match, problem: SpokenLanguages2 (dataset)*. URL: `https://community.topcoder.com/longcontest/?module=ViewProblemStatement&rd=16555&pm=13978` (visited on 11/19/2019).