Classifying Monosyllabic Audio Samples into Yoruba Phonemes: Neural Networks

CPSC 381 - Final Project Report
Prof. Alex Wong
Muyi Aghedo

Introduction

Currently, there is a gap in the current state of automatic speech recognition (ASR) systems for the Yoruba language, which presents a unique challenge within ASR due to a) its relative lack of a large corpus, and b) its tonality, which even native speakers can have trouble distinguishing between. While my project doesn't directly address either of these issues, this class has encouraged me to try my hand at training a natural language model, and I personally couldn't have found a more applicable avenue than doing so for the Yoruba language.

The state of my project explores training a neural network model to classify audio samples of vowels, as pronounced in the Yoruba language, to their correct vowels. I used PyTorch to design my network architecture and utilize my GPU, and hopefully, this project could be the first step in a longer journey to crafting a speech recognition system for Yoruba.

Data

All of my data is self collected. Because an audio corpus doesn't really exist specifically for my use case, and few thorough or labeled audio corpora exist in general for the Yoruba language, I did what I could to collect data on my own.

To do so, I developed a Flask app to record my own voice and then prepared a Firebase table to store the binary information of the audio files as a byteString. Browsers automatically record most audio media in .webm format, so my byte-strings follow that format.

For each vowel ("a", "á", "à", "e", "é", "è", "ẹ", "ẹ́", "ẹ̀","i", "í", "ì", "o", "ó", "ò", "ọ", "ọ́", "ọ̀", "u", "ú", "ù"), I recorded 17 - 18 audio samples of my saying the word as my training dataset. I realized that there were likely too many classifications and not enough data points per classifier, so I combined all diacritically-marked 'a' sounds under an 'a' label, 'e' sounds under 'e', and so forth. This made about 51 - 54 data points per my new vowel classification, with some vowels, like 'e' and 'o' having up to 100 data points. I split this data randomly into a training and testing set, which I loaded separately into their own torch DataLoaders.

Honestly, I still didn't believe I had nearly enough data to make a strong classification model still. Were I to attempt this project again, I would try to synthesize more data by adding noise to my current recordings, clipping the audio samples at different lengths or 'moving' the sample around different possible paddings. I could even try another approach, classifying using a convolutional neural network after converting the audio samples into a cool-looking image called a ‘Mel spectrogram’.

Methodology

I had collected my data and began writing code to hear it and convert them into numpy arrays when I realized my first issue: my audio samples had different raw audio data sizes because they were different lengths, but for a neural network to train, I would need a uniform input size across all audio sample inputs to my network. My workaround to this issue was padding. I padded every sample so that it was the same length as my largest sample, simply adding trailing zeros at the end of the audio data array until the necessary length was reached. I found out that tensorflow had a preprocessing function, pad_sequences, that did this, but when I moved off Colab and to my local machine, I realized Tensorflow didn’t yet support Python 3.13, so I had to write the function myself.

My second issue was choosing a network architecture, and honestly, I didn’t do much of changing my architecture often once decided simply because of how long (3 - 4 hours) it took for me to train 10000 epochs. I used a simple feed-forward neural network with an input size of 95040, three hidden layers of sizes 500, 1000, and 100, respectively, with 5 outputs (one per class/logit). Between the layers, I added ReLU activation functions.

For my optimizer, I used stochastic gradient descent, primarily because I was unsure of what other optimizers would perform better, and I used cross-entropy loss as my loss function, since I’m doing multi-classification. I added L1 regularization, because of how many features my inputs had.

Implementation Details

I mentioned a lot of the preprocessing in my methodology, aside from padding the raw audio data, I should mention needing to convert the original .webm byte string downloaded from Firebase into the raw audio data I needed. I used pydub’s AudioSegment library to read those bytes and return a numpy ndarray of just the audio, without the file headers and filetype-specific artifacts.

Otherwise, I arbitrarily set my learning rate to 0.001, added no weight decay, and set a batch size of about a quarter of the entire training dataset, and multiplied regularization by a very small lambda (0.0001).

In my initial attempts to train, my loss was becoming exceptionally large, and so I did some research and learned to clip my gradients, my weight changes, so they wouldn’t go crazy and make my loss function go out-of-whack. I’m still not completely sure why, but that one step completely saved my model.

Results

There isn’t much of a benchmark to test this model against, so I personally consider success if my model was able to get over 60% on my training set, and something over 50% on my test.

My actual results were surprising, almost scary: in about 6 epochs, my training dataset was already 100% accurate, and when testing the model with my test set, it was also 100% accurate…

On this note, I’m sure something fishy might be happening, but I conjecture that simply because my datasets aren’t big enough, it’s really easy to fit. Moreover, my test dataset isn’t really adversarial in any way, so it likely ‘feels’ just the same to the model.