Single word speech regognition using an RNN with Long Short-Term Memory

Brent Redmon, David Crowe

November 18, 2018

1 Introduction

Though much of the data in the world is unstructured, that does not mean that we cannot uncover similar patterns or trends within such unstructured data. Consider the sentence, "The deer ate the grass." If we were to switch around a few words such that it reads, "The grass ate the deer", we will see that these two sentences have widely differing meanings and connotations. Though language would be considered in the realm of unstructured data, that is not to say that there are no grammar rules or words that carry higher associations with some words over others. Breaking this down into a more microcosmic level, the very order of the sounds that we utter determines individual words, and by recognizing certain resonance and waveform patterns, we can understand sound input on a word by word basis. That is what this project aims to achieve, to recognize speech on a singular word basis.

2 Methodologies

This section aims to convey our methods of gathering our dataset, preprocessing it, building our model, and feeding the preprocessed data through the model.

2.1 Data Collection

The Selenium API allows for quick and easy browser automation and is used as the tool with which we were able to acquire our dataset. Imagine we were to train our model on the words 'get', 'thing', 'are', and 'many'. For each word in our dictionary, we tell our browser to go to the RobotVoiceGenerator website, which allows you to type in a word, play the word as a robot would say it, and also be able to change the pitch and speed that it's spoken. We wanted to have a wide representation of frequencies per word that we train our model with, so we instructed Selenium to perform this flow of operations for every word:

1. Type in the word in the text box.

2. Play the sound. Unless you play the sound for each word, then the file that you download for that word has no size.
3. Click on the download link to download a .wav file of the word
4. Increase the pitch of the word by 5 units of incrementation

When you download a stream of files from this website, they will download as *audio.wav*, *audio (1).wav* , *audio (2).wav* and so on. However, the caveat with this technique is that Google Chrome will only let you download 100 files with the same name. After that, it will prompt you with a message asking if you would like to add this file to your downloads folder. This breaks the flow of automation, so to get around this, we change the names of all the files starting with the word *audio* to a random cryptographic name so that the likelihood of any two files being named the same is very little.

Once all the files for a word are downloaded, we manually move them into our project folder. We are working to make this an automated process, but for the time being while we are training on such few words, we move each file into our working directory in this structure:

```
/
└── dataset
    ├── get
    │   ├── AE615st277kt746Sa732lR722Md70kO326Jb36wY996ho896.wav
    │   ├── IO423eO574RU742OP630SR479fq622QA398zI189Jn24za899.wav
    │   ├── xx204ME445pF246Jc918vN864GS503Ax209iz209Dk245of753.wav
    │   └── ...
    ├── thing
    │   ├── ba420O05Ha996ev932mw709Fd13eU200TH57er670RX469.wav
    │   ├── Ih446ZQ808hx146Nf972dh848Xl979XC339lk508Mb702qL583.wav
    │   ├── RC120lA314kb780Bh842jR360do743BQ698bT298Rf760xu918.wav
    │   └── ...
    ├── are
    │   ├── HW202PV433ZX70Gp867Ul652Gd833zZ381WQ414RN776oU469.wav
    │   ├── qO957DU692zZ111xw4KC100eU35eQ7ND736qF304lA617.wav
    │   ├── zy503hm798VN610WO590it659AR705Mz337qH29Ny276Qt707.wav
    │   └── ...
    └── many
        ├── aI816oL239wJ251tO147YX63SC926wo322YB839Mi758fW85.wav
        ├── gE656DH818qE434dQ599BZ560Ys552bz278re771aw255oP319.wav
        ├── xM836dj31DK578lt944cC942NZ68Xw61qk762kQ723UG534.wav
        └── ...
```

2.2 Preprocessing

The SciPy python library has a module called *wavfile* which allows for easy wav file manipulation. When you open a wav file with SciPy, the module returns back the sampling frequency of the audio file and a one dimensional numpy memmap integer array, which represents the individual sounds in the file. The duration of time that it takes to speak a word varies for each word, resulting in audio files with differing durations. To combat this, we set our standard to be two seconds per audio file, as most words do not take more than two seconds to say. Relying on the following equation, we can determine how much to extend an audio file such that it reaches two seconds.

$$seconds = frames/float(rate)$$

where the number of frames is equal to the length of the resulting memmap array, and the rate is the sampling rate, or number of samples recorded in the span of one second, which for our files is 22050 samples per second. Given this, if we want a clip of audio input with a duration of two seconds, we could need 44100 samples.

$$2 = frames/float(22050)$$

$$frames = 2 * float(22050)$$

$$frames = 44100$$

For each memmap array, we need to insert zeros at the end until we reach 44100 frames, resulting in a two second duration.

Next, we normalize the memmap between values of -1 and 1. We first much calculate the maximum and minimum signed values that could exist in the array, their average, and the range. Then we can calculate a normalized value with the following equation:

$$normalizedValue = (currentNum - average)/range$$

The Tensorflow LSTM model excepts to be fed a three dimensional array of values, so for each memmap we need to reshape it into a two dimensional array and then concatenate them to our overall dataset. We achieve this with the built in numpy *reshape* function, which expects two arguments: the array we are reshaping, and the dimensions we want to reshape it into. Another benefit of making our audio clips two seconds long is that 44100 happens to be a square number, so we can transform these memmaps into 210 x 210 squares.

2.3 Training the model

We plan to feed our dataset through a model consisting of a 128 node LSTM input layer with relu activation and 20% dropout. The next two layers will be dense layers, one with 64 nodes and the other with 32 nodes. Both will also have 20% dropout and a relu activation function. The final output layer will be a softmax dense layer with as many nodes as there are words that we are training our model on. We will optimize with the Adam optimizer and calculate the loss with categorical cross entropy over 100 epochs. Most likely, this model will not be complex enough to handle the large inputs, so we will alter the model as needed.

3 Post Training

To take advantage of the model we have just trained, we will be implementing a lightweight web based front end using the Flask web server micro-framework and bootstrap to make our page responsive. A user will be able to click a button, speak into their microphone, and have their voice be evaluated against the model such that it will predict what the user just said, and then display the prediction to the screen.