Rapidly Produced Datasets Against a CNN

Brent Redmon, David Crowe

Abstract—With the exception of SpeechCommands and SpokenVerbs, datasets consisting of single spoken words are hard to come by and can be very difficult and time consuming to assemble. Especially when acquiring such a dataset through means of volunteering, we deal with not only the issue of slow growth, but we also must trust that our volunteers are speaking the right words with respect to their associated labels. In this paper, we will discuss WordEnsemble, an application that we have developed that allows for the rapid creation of datasets consisting of single words through the LingoJam TTS (Text to Speech) service. We then test such a dataset against a pre-trained model provided by Tensorflow and wrap the service in a front end web application. All of the code can be found at https://git.txstate.edu/btr26/CS4347_VGG16_Project.git

Index Terms—Rapid prototyping, minimum viable product, Convolutional Neural Network, spectrogram

1 Introduction

SINCE the advent of natural language processing, Text-To-Speech (TTS) services have not only improved immensely, but they have also become widely accessible to the general public. Upon searching Google for ”text to speech online”, you will quickly see many services that offer free text to speech utilization, such as NaturalReaders, TTSReader, FromTextToSpeech, and many others, some of which offer the user the ability to download a snippet of audio containing the word that they have requested be translated into speech. There are other services, such as Google's Text-To-Speech API, Microsoft Azure Text to Speech API, and IBM Watson Text-To-Speech, which offer voice synthesis that is less robotic and more like a regular person. These services, however, are paid services, and therefore may not be as accessible to someone who wants to rapidly prototype a model against a single word dataset with limited resources. We discuss means of utilizing free resources such that a high enough quality dataset could be generated.

2 Methodology

The Selenium API offers a user the ability to automate tasks through a web browser. Such a tool is often used when software developers need to rapidly test functionality on their webpage without hiring a monkey tester. This tool, along with our WordEnsemble, is another means for software developers with limited resources to automate tasks and generate a higher level of production without the aid of expensive outside resources. We utilize this API to gather our dataset.

2.1 Input

Given a dictionary of words D, we grab each word D_i and feed them through the Selenium API. A Chrome browser window will appear and go to LingoJam's RobotVoiceGenerator service, where D_i is automatically typed into the website's text box. The website also allows the user to change the pitch of the generated word, and save the word to disk. We also include two hyper parameters, N and P, where N is the number of samples of audio to download, and P is how many units we increment the pitch per time a sample is downloaded.

2.2 Caveats

It is important to note that there is a limitation in the way that Google Chrome stores files of the same name. For files 1 through n with the

same name, Chrome will save the files as *file (1)*, *file (2)*, *file (2)*, and so on. However, once we get to *file (100)*, Chrome cannot save any more files past that, so we must rename the files in such a way that we can not only bypass this policy, but also so that we are very unlikely to name two files the same name which would hinder the flow of automation.

We perform this operation as such: For every 50 files that we download, we change our directory into our Downloads folder and pseudo-cryptographically rename each file. I say pseudo because there really is nothing cryptographic about it, we will merely be generating a random string and assigning that name to the file. The word that we generate will consist of thirty characters, and each set of 3 characters will consist of 2 randomly selected letters using the *string.ascii_letters* function, and a random number between 0 and 1000. That way, our chances of naming two files the same are drastically reduced.

It is also the case that we are currently only utilizing one website for all of our speech data, and although we generate words at different pitches, the underlying voice is still the same, which is a generic robot male voice. There are TTS services, such as Google's, that implement a RESTful interface rather than a physical download. This allows for us to take advantage of the HTTP protocol and cURL to download our files, rather than Selenium, which is a lot heavier and not as efficient. Free services, however, do not have a RESTful implementation, but offer a wider range of voices than LingoJam. Because of this, we are subject to the fact that many of these free services, with the exception of LingoJam, use AJAX methods of displaying links to download sound files. This means that if I were to make a POST request to a TTS processing server, my response would include all the HTML on the page *except* for the link to download my sound file, which has not been created yet.

This bane is what fuels the need for more research to be applied to this software suite, not only to make it free from the dependence of Selenium, but also such that we can take advantage of a pipelined POST, where data continues to flow through the pipe until we have the file link that we desire. Only then could we use cURL to make a GET request to the downloaded link and save the file to disk.

2.3 Output

In the working directory for our project, we have a directory called *dataset*, which will hold all of the words that we download. For each word that is downloaded and renamed, we continue our bash script which will create a folder in the *dataset* folder and name it the same name as the word that we downloaded. Then, we will automatically move each file in the *Downloads* folder into this newly created folder. Once we have downloaded all of our words and placed them in their correct folders, we will have a resulting directory tree that resembles the following structure:

```
/
└── dataset
    ├── get
    │   ├── AE61...o896.wav
    │   ├── IO42...a899.wav
    │   ├── xx20...f753.wav
    │   └── ...
    ├── thing
    │   ├── ba42...X469.wav
    │   ├── Ih44...L583.wav
    │   ├── RC12...u918.wav
    │   └── ...
    ├── are
    │   ├── HW20...U469.wav
    │   ├── qO95...A617.wav
    │   ├── zy50...t707.wav
    │   └── ...
    └── many
        ├── aI81...fW85.wav
        ├── gE65...P319.wav
        ├── xM83...G534.wav
        └── ...
```

For the model that we will test on, which was created by the Tensorflow team at Google, this is the structure that we must abide by.

2.4 Additional Functionality

In our software package, we also have included a script that also utilizes the Selenium API. It

scrapes the internet for the 1000 most commonly used words in the English language, which can be a great test set of words if a user wants a fast way to acquire a large set of words to train this model on.

3 Contribution of this work

There are many instances of companies that utilize speech recognition through their telephone system. The goal of this is typically to eliminate the need for a phone operator, who's job is to redirect callers to the appropriate departments depending on what they need. In the event of a startup company needing a system such as this, our software solution could be a viable option for them if they want a rapidly prototyped minimum viable product at no cost, at least in terms of training the model, rather than purchasing a model from another company.

4 Attempts to train the model

We took a few approaches to try to train a model that would accept our dataset, and for each model that we tried and modified, we increased our performance.

4.1 LSTM Recurrent Neural Network

Our first attempt to train this model was with an RNN that utilized a first layer LSTM that analyzed the sequence of bytes of sound entering the system. Much in the same way that the sentence "the deer ate the grass" is much different from "the grass ate the deer", we hypothesized that if we could train a recurrent neural network to understand that these two sentences are different, then we could train this same RNN to understand the sequence of phenomes that comprise a word.

Preprocessing

The Scipy library offers a function called *wavread*, which allows for a user to easily read and extract the data from a *wav* file. Among the data structures that it returns, one is a memmap integer array that represents each sample of sound from the file. Since we are dealing with 16 bit sampled *wav* files, this integer array can consist of numbers between -2^{16} to $2^{16} - 1$. Therefore, in order for our gradient descent function to converge without the possibility of overflow errors, we must normalize our data between 0 and 1. To do this, we use the normalization formula, which is defined to be:

$$z_i = \frac{x - min(x) * (smax - smin)}{max(x) - min(x)} + smin$$

where *smin* = 0 and *smax* = 255. We then took every *wav* file and converted them into png images. Once these images were created, we loaded these images into an array of integers as well as their associated label, and concatenated them into an array of tuples, which we shuffled so that it would be easier to split into training and testing data. Our model expected our samples x to be in the form of a two dimensional array, so we also took advantage of the numpy *reshape* function to transform all of our data samples into two dimensional arrays. Additionally, we concatenate zeros to the end of the array until we reach 44100 samples.

Model Structure

The input layer consists of a Long Short-Term Memory layer with 128 nodes and expects a two dimensional array of 210x210, which comes out to 44,100 features/samples. Since we are dealing with audio that is sampled at 22050 samples per second, each of our audio clips come out to two seconds each. The rest of the layers continue as follows:

1) LSTM layer with 128 nodes
2) Dense layer with 32 nodes
3) Dense layer with 64 nodes
4) Dense layer with 32 nodes
5) Dense layer with 2 nodes and a *softmax* activation function

At the time that the model was built, we were only testing on two words, which is why the last layer only has two output nodes. With the exception of the last layer, all other layers consisted of a *relu* activation function with twenty percent dropout. We used *sparse categorical crossentropy* as our loss function in conjunction with the *Adam* optimizer. Our *learning rate* α was 1e-5 and our *decay* value was 1e-4.

Results

With this technique, the results were not satisfactory. As our model trained, our training accuracy stayed absolutely constant, and our loss only changed by a few decimal points per epoch. In addition to our accuracy staying constant, it was low as well and usually stayed between 38% to 50%.

Epilogue to the first model

Python has a built in library called *wave*, which, like SciPy, also allows for the easy reading and writing of *wav* files. The difference with this library is that instead of the library returning back a memmap array of integers, it returns a string of bytes that represent the sounds in the wav file. If you have a string of bytes β, you can turn this string of bytes into an array in Python by typing:

byte_array = list(β)

We thought that by bypassing the need to normalize our data, we could create a pixel array that was more representative of our data samples, and thus could be more easily identified by our model. But unfortunately, this technique did not solve the issue of our accuracy staying constant.

4.2 Spectrographic Analysis With a CNN

The second attempt to train a model on our dataset involved image recognition on spectrograms, which is a three dimensional representation of sound consisting of frequency and decibel range as they change along the time domain. By convolving images of two second spectrograms such that we can extract their features, we hypothesized that we could successfully convert an audio recognition problem into an image recognition problem.

Preprocessing

For each sound file that was in our collection of words, we needed to convert them into spectrograms such that we could perform image analysis on them. This was achieved using the spectrogram creation tool in the *SciPy* library. The images were saved as *png* images of size 620x480.

Model Structure

Much in the same way that we had to prepare the previous model for a specific size of input, we had to do the same for our convolutional neural network. Each image would translate into an array of pixels of size 620x480.

Our sequential convolutional neural network consisted of layers that were very similar to our recurrent neural network:

1) Dense layer with 64 nodes
2) Dense layer with 32 nodes
3) Dense layer with 32 nodes
4) Flatten layer
5) Dense layer with two output nodes.

Much like the recurrent neural network, we were training this model on only two words. We calculated loss against *categorical crossentropy* in conjunction with the Adam optimizer with a learning rate of .0001. We achieved an accuracy of roughly 43% across 100 epochs. We did not train against more epochs because our accuracy continually fell as we trained. Therefore, we needed to restructure our model.

We decided to pursue training of a VGG16 model provided by Keras, which originally was trained with weights as provided by ImageNet. However, we were not able to load the pretrained weights into the model and therefore had to train from scratch. Our accuracy was even worse than the shallow convolutional neural network, which converged to around 23%.

Epilogue to the second model

5 Final Model

The final model that we used was a pre-trained model provided by Tensorflow, who's sole purpose was to train a dataset much to the likes of ours. It uses the same underlying principle of analyzing spectrograms rather than sound, much like how we originally proposed. However, the problem with our spectrograms was the fact that they did not have much variance between each graph. There was also a lot of wasted space in the picture between the white

space, x and y axis labels, and the majority of the graph which did not have a lot of valuable data (Fig. 1).

Fig. 2 shows an example of how our pre structured Tensorflow library generates the spectrograms that it trains itself on. Rather than the spectrogram containing axis information and wasted graph space, we are able to not only contain the picture to just the graph itself, but we also take more advantage of the space to color in pixels. This allows our spectrograms to become more unique from each other, and therefore can be identified by the model easier.

On this model with our dataset of 5 words (get, post, put, delete, patch), we were able to achieve 100% accuracy over 1400 epochs. Our accuracy over time and loss over categorical cross entropy is depicted in figures 3 and 4.

6 Wrapping our project

In this section, we discuss the ways in which we were able to wrap our project into a final product such that it could be compiled and ran given the right dependencies.

6.1 The Flask Micro-framework

Flask is a Python micro-framework that allows a user to rapidly produce a web server with routes and API endpoints that can be queried very rapidly. This was the framework that we housed our files in, which served as a medium of processing input files up against our trained model. Our backend consisted of one GET API and one POST API, and the HTML page that we had for our front end was rendered with the Jinja2 Templating Engine and prototyped using Bootstrap Studio.

6.2 Using the Web Service

Once the Flask server is launched, the user can access their localhost on port 5000, which will take them to the front end web service. They will be presented with the option to record a sound file and upload a sound file. In order for a user to make a prediction of a sound, they must record themselves saying a word, and save that file to their hard disk. Then, they can use the upload feature to select the file from their hard disk, and upload it to the Flask server, which will automatically run the file up against the model that we have trained. Once a prediction is determined, the server will serve the home page, but this time with a model that displays the prediction results.

6.3 External Dependencies

It is important to note that the sound that LingoJam generates is sampled at 22050hz on one channel. This is quite different from most built in computer microphones, which typically sample at 44100hz on two channels, or stereo. This conflicts with our model, because our model expects the kinds of files that it was trained on. To attempt to solve this, we used built in Python modules that allow for audio to be downsampled to lower frequencies. However, this changes the status of the file into a compressed file, which Python cannot read and interpret correctly.

To mitigate this, we use a command line interface called Sox, which is called from within our prediction algorithm. It can down sample and convert stereo files into mono files without changing its compression status, thus allowing it to still be manipulable in Python. So even though a user's microphone may be able to sample at very high frequencies, these sounds will always be downsampled to 22050 hz on one channel such that our model can understand it.

7 Conclusion

Though we were able to achieve fantastic accuracy on our generated dataset, we believe that these numbers can be misleading. As stated in the beginning, the dataset that we generate does not have a lot of variety as to the kinds of voices that it contains. It only has a male robotic voice, which is obviously not representative of the population. This is why though our accuracy numbers are very good, the model can depict what a user is saying only some of the time.

We believe that if we were able to combine the resources of more TTS services and funnel

them into a single dataset, we might be able to come closer to creating a single word audio dataset that is more representative of the collective voices in our population.

Fig. 1. Spectrogram using built in Python3 libraries

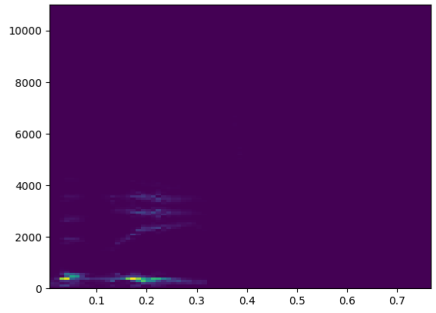

Fig. 2. Spectrogram built from Tensorflow

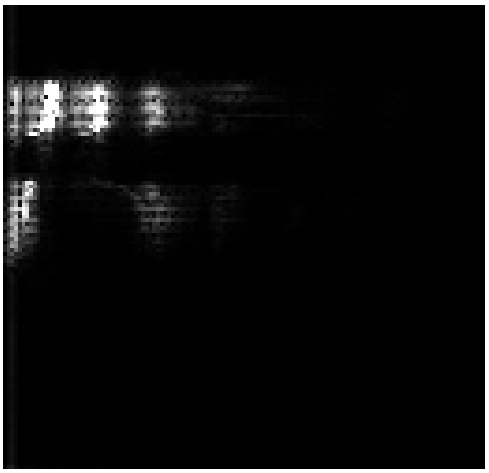

Fig. 3. Categorical Cross Entropy Loss

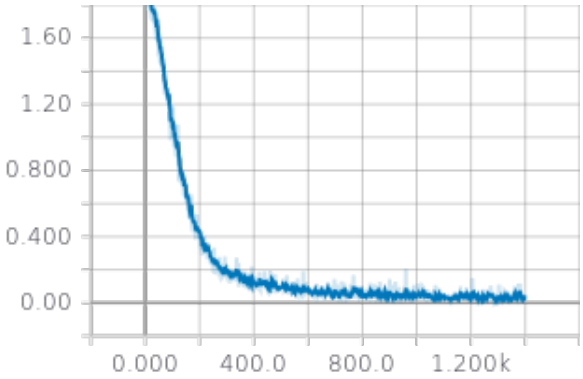

Fig. 4. Accuracy over 1400 Epochs