Keyword Spotting & Automated Censorship

COMP 576 Final Project Report

Ben Harris
Rice University
6100 Main St, Houston, TX 77005
bbh3@rice.edu

Kyle Manning
Rice University
6100 Main St, Houston, TX 77005
ksm9@rice.edu

Abstract

In this paper, we explore Keyword Spotting (KWS) - a field that has seen an explosion of use with the introduction of devices such as Apple's Siri, Amazon Alexa, Google Home, etc. We consider 2 models to identify whether keywords appear in a short, single-word audio sample. Then, we explored the process of audio splitting on words such that in the future, one could easily apply KWS to the task of censorship.

1. Introduction

Keyword Spotting is the task that attempts to identify whether a set of predetermined words (the keywords) are contained within an audio sample. For example, the word "Siri" for an iPhone, "Hey Alexa" to wake up an Amazon Echo, or a set such as the voice commands for these devices.

For our task, we seek to classify 1-second of audio into one of 11 different categories - the clip is labeled as either one of the 10 keywords, or not containing any keyword. In order to achieve this goal, we used the following models: the first option considered was modeling on raw waveform data using a 1-dimensional ConvNet. Our second model first transformed raw waveform data to a frequency representation, then transformed and extracted features at certain time intervals by using Mel-frequency Cepstrum Coefficients (MFCC's - see figure below)[1]. We then used a 2D convolution network to convolve over this coefficient.

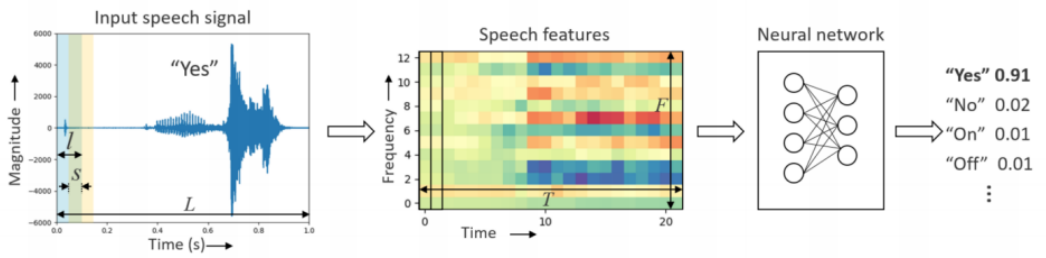


Figure 1: Keyword spotting pipeline.

[1]Image used is from [2]

2. Background

In recent years, deep learning has broached Hidden Markov Models (HMM's) as the best choice for this task. A typical KWS pipeline involves extracting features, then using a variety of neural nets for the task. While approaches involving HMM's are generally accurate, they're difficult to train and are more computationally expensive than deep learning models for prediction. One model that could be used for this task is a Recurrent Neural Network (RNN). Though accurate, these too suffer from being too expensive to use in tasks that require low latency (i.e. censoring/ KWS). See [1] for more background on different models. Thus, we consider Convolutional Neural Networks (CNN's) for this task.

3. Data

We used the "Google Speech Commands" dataset [2]. This dataset contains 105,000 WAVE files of 34 different words (see figure below). Each utterance is a second or less.

Word	Number of Utterances
Backward	1,664
Bed	2,014
Bird	2,064
Cat	2,031
Dog	2,128
Down	3,917
Eight	3,787
Five	4,052
Follow	1,579
Forward	1,557
Four	3,728
Go	3,880
Happy	2,054
House	2,113
Learn	1,575
Left	3,801
Marvin	2,100
Nine	3,934
No	3,941
Off	3,745
On	3,845
One	3,890
Right	3,778
Seven	3,998
Sheila	2,022
Six	3,860
Stop	3,872
Three	3,727
Tree	1,759
Two	3,880
Up	3,723
Visual	1,592
Wow	2,123
Yes	4,044
Zero	4,052

Figure 1: How many recordings of each word are present in the dataset [2]

As mentioned in [2], the most common speech commands used are "Yes", "No", "Up", "Down",

[2] Taken from [1]

"Left", "Right", "On", "Off", "Stop", and "Go"; these formed our keyword set, while the other words form our set of unknowns.

4. Models

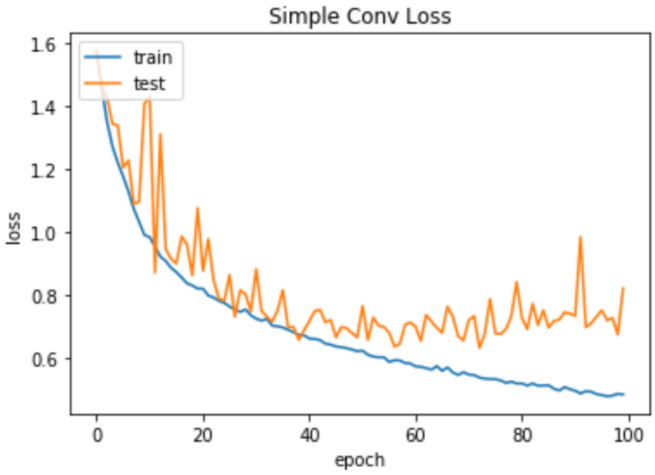
1-Dimensional ConvNet on Raw Waveform

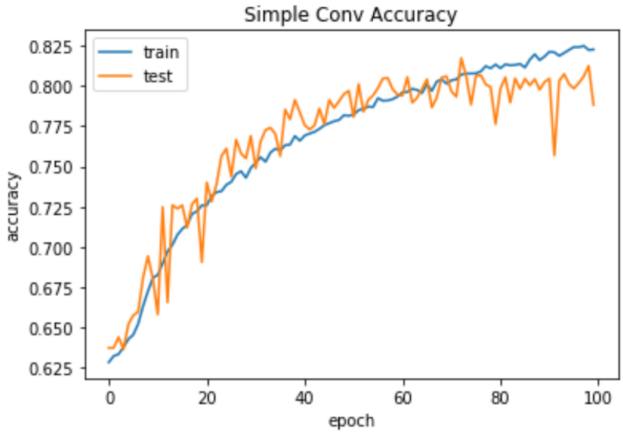
As previously mentioned, we modeled two different representations of the audio data. While modeling on the raw waveforms, we used the following convolutional network:

1. 1D Conv Layer with 8 filters and a stride of 5, ReLu, MaxPool of size 2, Dropout
2. 1D Conv Layer with 16 filters and a stride of 3, ReLu, MaxPool of size 2, Dropout
3. 1D Conv Layer with 32 filters and a stride of 5, ReLu, MaxPool of size 2, Dropout
4. 1D Conv Layer with 64 filters and a stride of 4, ReLu, MaxPool of size 2, Dropout
5. 1D Conv Layer with 128 filters and a stride of 3, ReLu, MaxPool of size 2, Dropout
6. Flatten
7. Dense256 , ReLu, Dropout
8. Dense 128, ReLu, Dropout
9. Dense 11, Softmax

We use categorical cross-entropy loss and used with Adam Optimizer with a learning rate of 0.001. Note the accuracy and loss curves below - we achieved mediocre results at a peak validation accuracy of 81.25%. The training was accomplished over the course of 100 epochs.

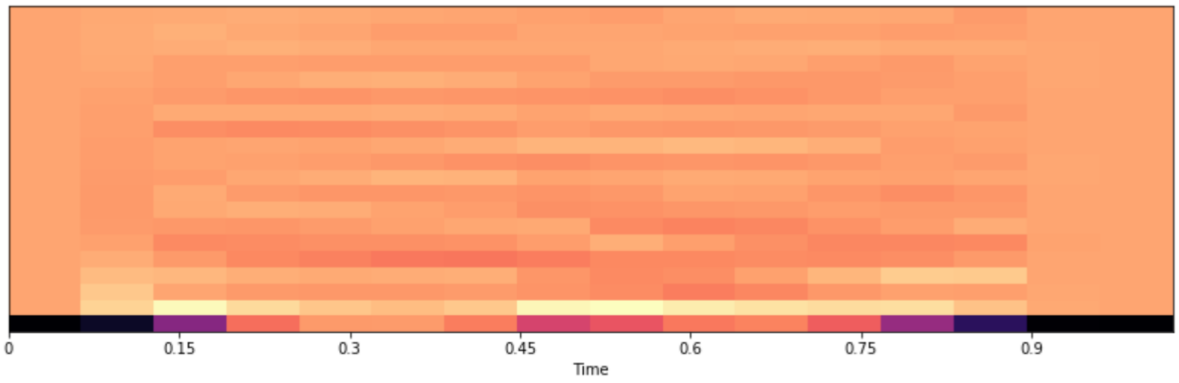
Accuracy and Loss over Training (1D ConvNet)





2-Dimensional LeNet5 on MFCCs

Our other model used features extracted from the raw audio. This involved MFCC samples at 16 different time steps within the 1second utterances - see figure below.

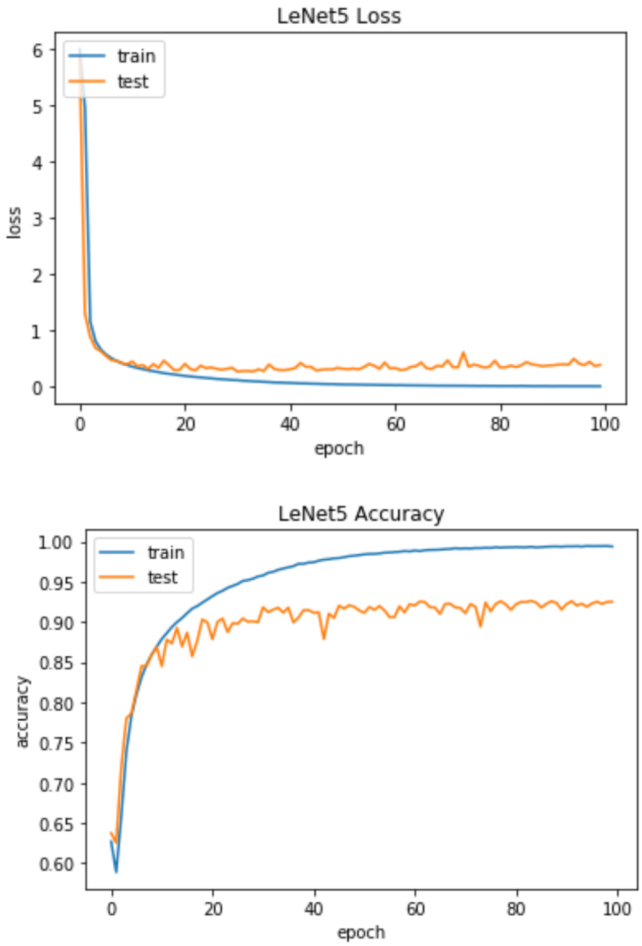


This idea was heavily influenced by the paper [1]. However, we considered a slightly deeper model; though we experimented with different filter sizes and stride setups, including a layout mentioned in [1] our best results were achieved from the classic LeNet; see model layout below :

1. 2D Conv Layer with 32 filters, filter size of (5, 5), ReLu, MaxPool of (2, 2)
2. 2D Conv Layer with 64 filters, filter size of (5, 5), ReLu, MaxPool of (2, 2)
3. Flatten
4. Dense 1024, ReLu
5. Dropout
6. Dense 11, Softmax

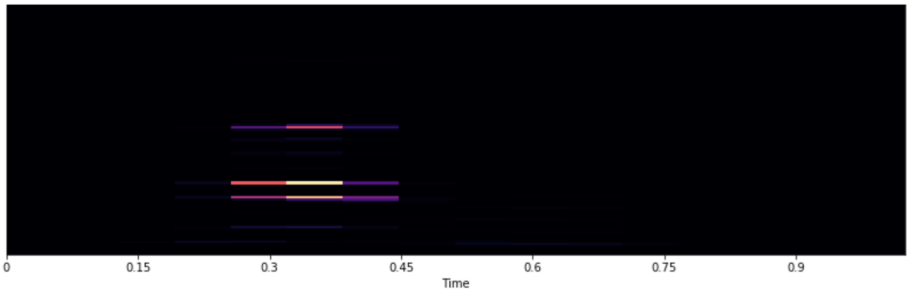
Consider the accuracy and loss plots below; we obtained an admirable accuracy of 92.52%. We also chose to use the RMSprop optimizer with the learning rate of 0.0001 and decay rate of 1e-6; ADAM tended to get stuck in local maxima, even with varying learning rates.

Accuracy and Loss over Training (2D LeNet5)



Spectrograms

We also explored implementing a model with a spectrogram as input. A spectrogram is a visual representation of the frequency as it varies with time and looks quite similar to an MFCC. Ultimately, after some quick visualization, we found that the simplicity of words, sampling rates, and .wav filetype caused spectrograms created from our data to be mostly blank data. With this in mind (and content with the accuracy received from the MFCC's), we chose not to build a model with spectrograms as input.



5. Conclusion & Discussion

Considering the simplicity of our models, our results show promise. We attribute this success to our feature extraction techniques. This is useful in a real world setting for a variety of reasons - considering censoring, for example, the simplicity of our model lends itself to perform well as a low latency keyword identifier.

For the task of automated censorship, we needed to isolate the individual words for predictions in our models. While we explored writing custom algorithms to split on the valleys times in the noise data, ultimately (based on time considerations) we implemented Google Clouds' speech-to-text API to timestamp and then used the librosa library to cut out and pad words.

After running this script -- `cut_speech.py` -- to isolate the words, we can feed the resultant .wav files through the model. If the model determines them to be a keyword, we can use librosa to replace the time with a beep wav file of the same frequency and time length.

Some future work could be done in terms lowering the preprocessing overhead. This would enable the model to work in a background setting more easily. Based on the format of our models, our automated censorship is also completely dependent on the word splitting capabilities. Improvements in those algorithms would drastically improve the ease of implementation for a deep-learning based censor.

Perhaps more importantly, we believe that with appropriate datasets, these results could be extended to include difficulties such as longer speech segment length (with potential for multiple words), background noise perturbation, and of course different sets of words.

References

[1] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” arXiv preprint arXiv:1804.03209, 2018.

[2] T. Sainath and C. Parada. Convolutional neural networks for small-footprint keyword spotting. In Proc. Interspeech, 2015.

[3] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello edge: Keyword spotting on microcontrollers,” arXiv preprint arXiv:1711.07128, 2017.