Techniques to Improve Small-Footprint Keyword Spotting

Dhananjay Deshpande
Data Science Institute
Columbia University
dcd2139@columbia.edu

Wonkyung Kim
Computer Science, GS
Columbia University
wk2294@columbia.edu

Saaya Yasuda
Data Science, GS
Columbia University
sy2569@columbia.edu

Abstract

We explored different techniques to improve small-footprint Keyword Spotting Task (KWS) in Speech-to-Text, primarily for mobile device and IoT applications. This means the models need to be kept small while achieving high accuracy. DNN, CNN, and CRNN have been investigated earlier to improve the accuracy and to reduce the model complexity. After building a preprocessing pipeline on Cloud, experimenting with various model structures, and tuning parameters based on past work, our CRNN model with GRU and CE loss achieved a 95.4% accuracy which is 4.77% higher than the Kaggle winner's 91.06% accuracy, with a testing/prediction time of 0.25 seconds.

1. Introduction

Mobile/IoT voice platforms are becoming ubiquitous. In the last few years, we started witnessing many voice-activated speakers and assistants (e.g., Amazon Echo/Alexa and Google Now) becoming popular in the market. The eMarketer forecast in May 2017 noted that 35.6 million American consumers use a voice-activated assistant device at least once a month [1], and Juniper Research reported in November 2017 that over 70 million households will own at least one of the smart speakers in their home by 2022, with the total number of 175 million installed devices in the market[6].

Given the popularity of voice-activated smart speakers, common interest is to recognize keywords or wake words from human speech, such as "Hey Siri" and "Alexa." This task is known as Keyword Spotting (KWS) [14]. The way in which smart speakers and mobile devices work is that the devices listen and detect the wake words continuously and act on commands that come after the wake words. In order to do the task, a pre-trained model is deployed to these devices.

Multiple services (e.g., IBM Watson and Google Speech API) are now available and provide different levels of service for speech recognition. However, building a speech-to-text model for this application has a unique set of constraints such as limited CPU/memory of the speakers and mobile devices. Thus this task is called by multiple papers a "small-footprint" keyword spotting task. There have been multiple attempts at improving the performance at this task as outlined in the Current Research section (1.1). In this paper, we intended to research ways to further improve the accuracy while keeping a low model complexity.

The remainder of this paper is organized as follows: Section 1 outlines background, our problem and goal, and any constraints such as dataset and environment. Section 2 provides the overview of the methodology we used to approach this problem as well as details on implementations other than the model structures. Section 3 details the model structures for each type of models created for this project. Section 4 provides the results, and section 5 summarizes our findings as well as suggestions for the future work.

1.1. Current Research

Given the practical use cases, multiple companies and research teams have attempted to challenge and improve on a small-footprint keyword spotting task over the last few years. Current research on this area is based on DNN (Microsoft, 2015)[13], CNN (Google, 2015)[14], RNN (University of Lubeck, 2015)[8], and CRNN (Mindori and Baidu, 2016-2017)[9] [11] architectures with features such as filter bank and MFCC. [10] [7] According to the CRNN paper by Baidu from 2017 [11] which was the latest at the time of this project, DNN did not provide optimal performance, and CNN with cross entropy (CE) loss [14] provided a better performance. While RNN with Connectionist Temporal Classification (CTC) loss [8] and CRNN with CTC loss [9] did not provide improvement, CRNN with CE loss [11] provided improvements (but this was for recognizing one label).

Since the start of the project, there have been two additional papers recently published on this topic: Xiomi experimented with various CRNN models (March 29, 2018) [12] and Amazon used a DNN-HMM decoder model (April 19, 2018) [15]. Both papers focused on reducing the false rejection rate while keeping the model size small.

In April 2017, Google released an open-source speech command dataset for Tensorflow [3]. Following this, Google Brain hosted a Kaggle competition in from November 2017 to January 2018 [2] as a testbed for trying various techniques. Tensorflow's official audio recognition tutorial [3] uses two types of CNN architectures with CE loss, replicating the models proposed in Google's 2015 paper [14].

1.2. Problem Statement

In speech processing, keyword spotting (KWS) task deals with the identification of keywords in utterances. Our problem is to maximize the accuracy of keyword identification given resource constraints on CPU/memory of mobile/IoT devices.

1.3. Goal

In this project, we will build an algorithm that can recognize 10 simple commands from audio data (”yes”, ”no”, ”up”, ”down”, ”left”, ”right”, ”on”, ”off”, ”stop”, and ”go”). Given the current research findings, we start from Tensorflow/Google's CNN models with CE loss, replicate the CRNN structure of the latest paper[11], and further explore and find the most efficient model with the highest possible accuracy. Our goal is to propose a better network structure for this KWS task.

1.4. Dataset

We used the Speech Commands Dataset[3], an open source audio dataset created by Google and Tensorflow in 2017. The dataset contains 65,000 wave files of one-second-long voice recordings, and 30 words are pronounced by thousands of people who contributed to the dataset through the Google AIY website. Each folder in the dataset represents a word and contains wave files for the word. Following the Kaggle competition's rule, We focused on identifying the ten words (”yes”, ”no”, ”up”, ”down”, ”left”, ”right”, ”on”, ”off”, ”stop”, and ”go”) and considered the other 20 words as unknown/silence. Kaggle website has a more detailed description of the dataset. [2]

1.5. Environment

We initially tested most models using the local environment. After the data pipeline and preprocessing are built for running the models in Google Cloud to facilitate and shorten the training process, we picked best performing models from the local environment and ran on Google Cloud to optimize the parameters. At the very end, the credits ran out, so some of the results are partial and/or supplemented by local results. We will discuss more on this in the results section.

1.6. Evaluation Criteria

For comparing models for a small-footprint KWS task with a limited CPU resource, we need to consider the combination of accuracy measure and efficiency measure. Accuracy calculation is based on a confusion matrix (i.e., true positive and true negative over total data). An efficiency measure of our choice is the test time to classify the test set and the number of parameters in the model. The number of files in the test set is 6500. For the test time, to keep it consistent, we measured it in the local environment on a specific laptop (2.3 GHz Intel Core i5 processor; 8GB 2133 MHz LPDDR3 memory; MacOS Sierra) for the models we tried, and for some of the models, we also recorded this number in Google Cloud.

For accuracy, We originally aimed for a 90% test accuracy for 65,000 .wav audio files. For reference, the Kaggle winner of the TensorFlow Speech Recognition Challenge has the accuracy of 91.06%, and the Bronze-level winners range from 88.79% to 89.568%.

2. Method

2.1. Method

Our approach to this problem was to first take the Tensorflow's CNN models and run it locally, replicate the CRNN model from Baidu's CRNN paper (no code-base was available), and create different model structures that may perform better than the two papers' structures. Each local training took about 20-24 hours.

Then we built the preprocessing pipeline using Google Cloud Dataflow and Apache Beam and ran a few of the best-performing models via Google Cloud ML Engine. Advantages of using Google Cloud are: 1) separation of preprocessing, training, and prediction steps; 2) high performance (training 5X faster); and 3) ability to store and share results. In the end, we optimized parameters for the models we chose.

In the local environment, we tried the following models with various parameters and batch sizes, with both Gradient Descent and Adam optimizers, and with a dropout:

- CNN (traditional 2 layer model)
- CNN (low latency 1 layer model)
- CRNN with GRU
- CRNN with LSTM
- RCNN with GRU

Based on the local performance, in Google Cloud, we tried optimizing the following models with a 10% added noise to the background of the data:

- CNN (traditional 2 layer model and 3 layer model)

- CRNN with GRU
- RCNN with GRU

Details of the model structures are described in the Model Architectures section.

2.2. Pre-Processing Pipeline

Google Cloud Dataflow is a fully managed service for transforming and enriching data. It allows data preprocessing and augmentation and is based on the use of Apache Beam. Apache Beam is an advanced unified programming model to construct processing pipelines on data. We used Apache Beam 2.4 with Google Cloud Dataflow for this project to construct the data pipeline.

Google Cloud does not have an example Dataflow and training pipeline for audio data. Our code was built based on the framework of the Cloud ML sample for the flowers (image) dataset, and it was built to be able to contribute back to Cloud ML.

After splitting the data, our preprocessing code converts WAV audio files into mel-frequency cepstral coefficients (MFCCs) fingerprints and save them as TFRecord files.

2.3. Codebase

The codebase can be found in `https://github.com/dhananjay-deshpande/speech-commands-recognition` [4]. This includes the work on splitting data, Cloud ML Dataflow Pipeline for data preprocessing using Apache Beam 2.4. Tensorflow 1.6 was used for training and prediction.

We used another Github repository for running the models locally [4]. It is based on Tensorflow tutorial's codebase in Github [5] which contains CNN models. We constructed the CRNN and RCNN models in models.py. Most of these models are slightly modified and included in our Cloud Github repository.

3. Model Architectures

In this section, the model structures are described. For all the models, we used a CE loss function rather than a CTC loss function. This is largely because a CTC loss function is heavier and Baidu's paper [11] succeeded in improving the results by using a CE loss function.

The labels used in the model tables mean the following:
Convolutional layer (CNN):

- NC: Number of filters.
- LT & LF: Kernel/filter sizes.
- ST & SF: Strides.

Recurrent layer (RNN):

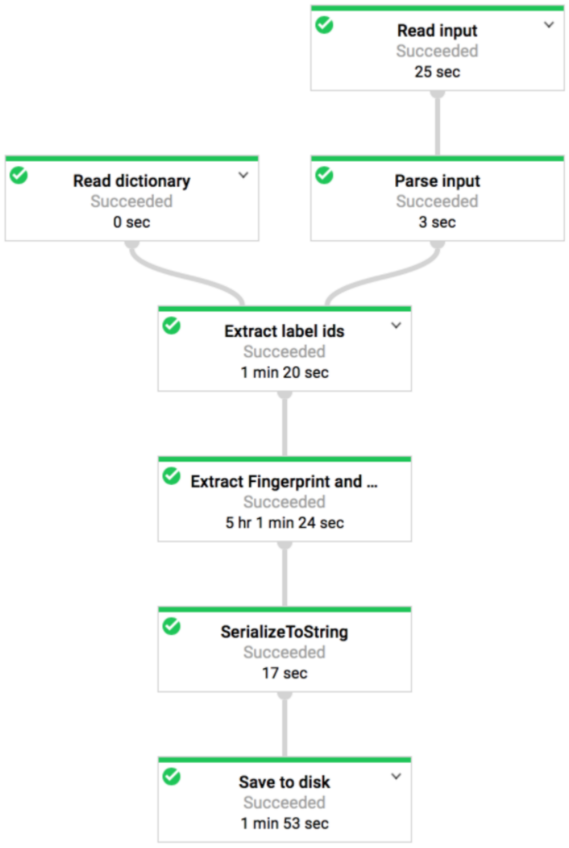
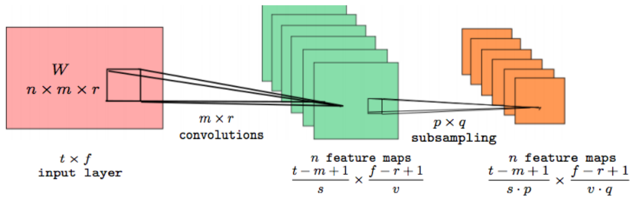


Figure 1: Preprocessing pipeline output from Google Cloud Dataflow



W
$n \times m \times r$
$t \times f$ input layer
$m \times r$ convolutions
n feature maps $\frac{t-m+1}{s} \times \frac{f-r+1}{v}$
$p \times q$ subsampling
n feature maps $\frac{t-m+1}{s \cdot p} \times \frac{f-r+1}{v \cdot q}$

Figure 2: Diagram showing a typical convolutional network architecture [14]

- R: Number of recurrent layers.
- NR: Number of hidden units.

Fully connected layer (DNN):

- FC: Number of units in fully connected layer.

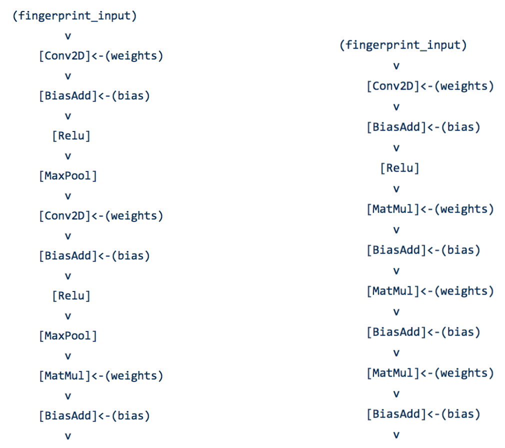


Figure 3: Left: Traditional CNN (cnn-trad-fpool3); and right: low latency CNN (cnn-one-fstride4)

3.1. CNN Models

Most of the CNN models we tried come from Google's CNN paper from 2015 [14]. There are two main models: "cnn-trad-fpool3" which provided the best accuracy in the paper and "cnn-one-fstride4" which provided the lowest number of multiplications in the model (thus efficient). For those model structures, see the Table 1 and 2 respectively. The main differences between the two models are the number of convolutional layers and the max-pooling process. The cnn-one-fstride4 model omitted a CNN layer and max-pooling since it makes the model more complex by having a large number of multiplications (see figure 2). Figure 3 also shows the model structure for these two models in more detail.

We ran 10000-20000 iterations locally with batch sizes 100 and 500, a dropout rate of 0.5, and both Gradient Descent optimizer or Adam optimizer. They tended to have converged within 10000 iterations, and while the test accuracy wasn't as high as other models, the testing time was much lower compared to the models using any recurrent layer.

Based on this preliminary result, the question for the CNN model in particular was whether we can improve the accuracy while maintaining the low testing time, by adding a layer(s). Thus we created another model with an extra layer based on the traditional CNN with max-pooling (since the accuracy was better than the low latency model's). See the Table 3 for this model.

3.2. CRNN Models

Our next model of choice is the CRNN model, which is expected to have the better performance than CNN for a speech recognition task [11].

Figure 4 shows the architecture of the CRNN model from Baidu's 2017 CRNN paper [11]. A CRNN can be described as a modified CNN by replacing the last convo-

2-layer Traditional CNN (cnn-trad-fpool3)				
type	NC	(LT, LF)	(ST, SF)	Pooling Size
CNN	64	(20, 8)	(1, 1)	(1, 3)
CNN	64	(10, 4)	(1, 1)	(1, 1)
DNN	128	N/A	N/A	N/A

Table 1: Traditional CNN model (cnn-trad-fpool3)

1-layer low latency CNN (cnn-one-fstride4)				
type	NC	(LT, LF)	(ST, SF)	Pooling Size
CNN	186	(32, 8)	(1, 4)	N/A
DNN	128	N/A	N/A	N/A
DNN	128	N/A	N/A	N/A

Table 2: low latency CNN model (cnn-one-fstride4)

3-layer Traditional CNN				
type	NC	(LT, LF)	(ST, SF)	Pooling Size
CNN	64	(20, 8)	(1, 1)	(1, 3)
CNN	64	(10, 4)	(1, 1)	(1, 3)
CNN	64	(5, 2)	(1, 1)	(1, 1)
DNN	128	N/A	N/A	N/A

Table 3: Traditional CNN model - 3 layers

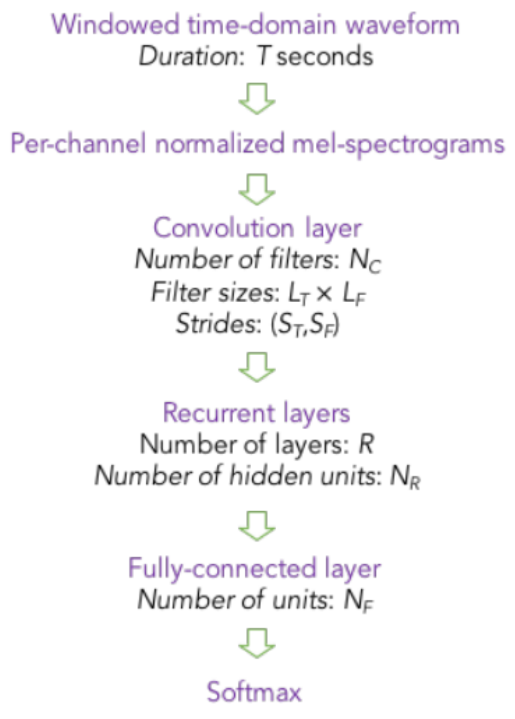


Figure 4: End-to-end CRNN architecture for KWS. [11]

CRNN-GRU 1 (best parameters of [11])					
type	NC	(LT, LF)	(ST, SF)	R	NR
CNN	32	(20, 5)	(8, 2)	N/A	N/A
RNN	N/A	N/A	N/A	2	32
DNN	64	N/A	N/A	N/A	N/A

Table 4: CRNN-GRU Model 1

CRNN-GRU 2					
type	NC	(LT, LF)	(ST, SF)	R	NR
CNN	64	(8, 20)	(1, 2)	N/A	N/A
RNN	N/A	N/A	N/A	2	128
DNN	256	N/A	N/A	N/A	N/A

Table 5: CRNN-GRU Model 2

CRNN-GRU 3					
type	NC	(LT, LF)	(ST, SF)	R	NR
CNN	32	(20, 5)	(4, 2)	N/A	N/A
RNN	N/A	N/A	N/A	2	64
DNN	128	N/A	N/A	N/A	N/A

Table 6: CRNN-GRU Model 2

lutional layer with an RNN layer. A potential drawback of CNN is that they cannot model the context over the entire frame without wide filters or great depth. Adopting an RNN for aggregating the features enables the networks to take the global structure into account while local features are extracted by the remaining convolutional layers. As we can see in Figure 4, overall architecture is similar to CNN except the fact that there are both convolutional layers and recurrent layers. The main difference from CNN is that the outputs of the convolutional layer are fed once more to bi-directional recurrent layers (GRU or LSTM). And the outputs of the recurrent layers are given to the fully connected layer.

We developed two types of CRNN model using a GRU layer and LSTM layer. In a small-scale test in the local environment, LSTM did not perform as well as the GRU model in terms of accuracy. This result aligns with Baidu's CRNN paper [11]. Based on this and the fact that LSTM is less efficient than GRU, we decided to optimize the GRU model in Cloud rather than the LSTM model. Similarly, in the local environment, we tried both Gradient Descent and Adam optimizers. Given the performance and following the paper [11], we decided to use Adam in the Cloud environment. The dropout rate was consistently at 0.5.

Table 4-6 show the architecture of the CRNN models we ran on the Cloud. CRNN-GRU 1 uses the best parameters from Baidu's CRNN paper [11].

RCNN-GRU 1					
type	NC	(LT, LF)	(ST, SF)	R	NR
RNN	N/A	N/A	N/A	2	32
CNN	32	(20, 5)	(8, 2)	N/A	N/A
DNN	64	N/A	N/A	N/A	N/A

Table 7: RCNN-GRU Model 1

RCNN-GRU 2					
type	NC	(LT, LF)	(ST, SF)	R	NR
RNN	N/A	N/A	N/A	2	128
CNN	32	(20, 5)	(8, 2)	N/A	N/A
DNN	128	N/A	N/A	N/A	N/A

Table 8: RCNN-GRU Model 2

RCNN-GRU 3					
type	NC	(LT, LF)	(ST, SF)	R	NR
RNN	N/A	N/A	N/A	3	8
CNN	128	(20, 5)	(8, 2)	N/A	N/A
DNN	32	N/A	N/A	N/A	N/A

Table 9: RCNN-GRU Model 3

RCNN-GRU 4					
type	NC	(LT, LF)	(ST, SF)	R	NR
RNN	N/A	N/A	N/A	1	16
CNN	64	(10, 3)	(8, 2)	N/A	N/A
DNN	32	N/A	N/A	N/A	N/A

Table 10: RCNN-GRU Model 4

3.3. RCNN Model

In addition to the conventional CRNN format where a convolutional layer is placed before a recurrent layer, we experimented by placing the recurrent layer before a convolutional layer. We call this RCNN. The model architectures are shown in the table 7-10. They all used Adam optimizer and a dropout rate of 0.5.

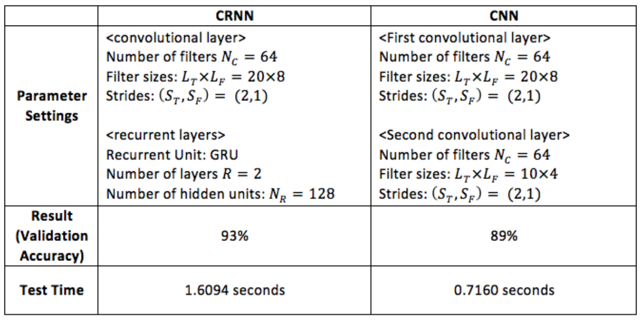
3.4. Data Augmentation

As described in the previous section, after Google Dataflow pipeline was set up, we added 10% background noise to the data. For all models run on Cloud, the data was augmented this way. It uses the data from "_background_noise_" folder, and this was also recommended by the Kaggle competition's Data page to improve the training performance.

4. Results

4.1. Results in the Local Environment

We ran various forms of CNN and CRNN models in the local environment at the small scale, picked the best performing models to run with a large number of iterations, and achieved the results below. The model ran for roughly 24 hours each (i.e. about 20,000 iterations). As we expected, the CRNN model worked better than the CNN model with the similar parameter settings, but the CRNN model took more time to classify the test data (See test time in the chart below).



	CRNN	CNN
Parameter Settings	<convolutional layer> Number of filters $N_C = 64$ Filter sizes: $L_T \times L_F = 20\times8$ Strides: $(S_T, S_F) = (2,1)$ <recurrent layers> Recurrent Unit: GRU Number of layers $R = 2$ Number of hidden units: $N_R = 128$	<First convolutional layer> Number of filters $N_C = 64$ Filter sizes: $L_T \times L_F = 20\times8$ Strides: $(S_T, S_F) = (2,1)$ <Second convolutional layer> Number of filters $N_C = 64$ Filter sizes: $L_T \times L_F = 10\times4$ Strides: $(S_T, S_F) = (2,1)$
Result (Validation Accuracy)	93%	89%
Test Time	1.6094 seconds	0.7160 seconds

4.2. Results in the Cloud Environment

Overall, the CRNN-GRU3 model performed the best if we consider the balance between the accuracy and efficiency. CNN models gave a shorter testing time and lower number of parameters, but the testing accuracy was not high enough. RNN models performed well in terms of accuracy, but the testing time is slightly longer and there are too many parameters. See the Results in Table 11.

Note that the Google Cloud credits ran out at the end, so some training iterations did not finish. For those runs, the result as of the checkpoint is listed.

After training is done, we also tested with the data with our own voice. The best model was able to predict the word correctly.

5. Conclusion and Future Work

5.1. Conclusion

After building a preprocessing pipeline on Cloud, experimenting with various model structures, and tuning parameters based on past work, our CRNN-GRU3 model with Adam optimizer, 0.5 dropout rate, 10% data augmentation with background noise, and CE loss achieved a 95.4% accuracy. This is 4.77% higher than Google's Speech Recognition Kaggle competition winner's 91.06% accuracy. While the model's number of parameters are not as low as CNN models, the testing/prediction time for the model was 0.25 seconds.

5.2. Future Work

With more Google Cloud credits, we could iterate more models in a structured way. We would like to explore the way to minimize the number of model parameters. Another possibility is to deploy the model to an Android device as described in the Tensorflow tutorial [3] and use the real-time voice data.

Model	Optimizer	Iter	Batch	Train Acc	Test Acc	# Param	Testing Time
Low latency CNN	Gradient	10000	500	90.8	69.2	949,866	
Low latency CNN	Adam	10000	500	93.2	68.3	2,849,592	0.07 sec
CRNN-GRU1	Adam	20000	100	92.0	83.0	719,946	0.17 sec
CRNN-GRU2	Adam	5709	500	97.0	95.0	13,955,562	1.05 sec
CRNN-GRU3	Adam	20000	500	98.2	95.4	2,843,274	0.25 sec
RCNN-GRU1	Adam	10000	500	95.4	92.3	19,016,510	0.52 sec
RCNN-GRU2	Adam	10000	500	96.2	94.3	75,977,342	1.39 sec
RCNN-GRU3	Adam	10000	500	90.8	89.8	5,566,382	0.23 sec
RCNN-GRU4	Adam	10000	500	89.0	90.4	8,621,394	0.34 sec
Trad CNN	Adam	2118	500	91.7	66.0	2,780,586	2.23 sec
Trad CNN 3 layers	Adam	2049	500	90.6	63.9	1,221,738	2.18 sec

Table 11: Results

References

[1] Alexa, say what?! voice-enabled speaker usage to grow nearly 130 https://www.emarketer.com/Article/Alexa-Say-What-Voice-Enabled-Speaker-Usage-Grow-Nearly-130-This-Year/1015812.

[2] Google speech command kaggle competition. https://www.kaggle.com/c/tensorflow-speech-recognition-challenge.

[3] Google speech data set. https://www.tensorflow.org/versions/master/tutorials/audio_recognition.

[4] Team 19 deep learning project ml pipeline link. https://github.com/dhananjay-deshpande/speech-commands-recognition.

[5] Tensorflow speech commands example. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands.

[6] Voice-enabled smart speakers to reach 55report. https://techcrunch.com/2017/11/08/voice-enabled-smart-speakers-to-reach-55-of-u-s-households-by-2022-says-report/.

[7] H. Fayek. Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between. http://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html, 2016.

[8] K. Hwang, M. Lee, and W. Sung. Online keyword spotting with a character-level recurrent neural network, Dec 2015.

[9] C. Lengerich and A. Hannun. An end-to-end architecture for keyword spotting and voice activity detection, Nov 2016.

[10] M. Neumann and N. T. Vu. Attentive convolutional neural network based speech emotion recognition: A study on the impact of input features, signal length, and acted speech, 2017.

[11] R. C. J. H. A. G. C. F. R. P. A. C. Sercan O. Arik, Markus Kliegl. Convolutional recurrent neural networks for small-footprint keyword spotting, March 2017.

[12] C. Shan, J. Zhang, Y. Wang, and L. Xie. Attention-based end-to-end models for small-footprint keyword spotting, March 2018.

[13] V. Sindhwani, T. N. Sainath, and S. Kumar. Structured transforms for small-footprint deep learning, Oct 2015.

[14] C. P. Tara N. Sainath. Convolutional neural networks for small-footprint keyword spotting. http://www.iscaspeech.org/archive/interspeech_2015/papers/i15_1478.pdf, 2015.

[15] M. Wu, S. Panchapagesan, M. Sun, J. Gu, R. Thomas, S. N. P. Vitaladevuni, B. Hoffmeister, and A. Mandal.