

**Title:** Deep learning for speech and respiratory audio inference on edge devices

A project report presented to  
the faculty of  
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment  
of the requirements for the degree  
Master of Science in Electrical Engineering and Computer Science

by

Taiman A. Siddiqui

Advised by

Savas Kaya  
Professor, Electrical Engineering and Computer Science

## Abstract

This project is focused on the deployment of optimized deep learning models on edge devices to achieve lower latency, greater scalability, remote operability, higher security and less bandwidth usage for cases such as speech command inference and health monitoring in medical applications. As such, I explored the optimization of neural network architectures and model training parameters as well as audio processing parameters to detect keywords for speech command inference within resource constraints of edge devices. Among the keyword spotting models I explored, the convolutional neural network (CNN) based model had the best average classification accuracy for detecting 2 to 12 keywords. The CNN based keyword spotting model had relatively good average classification accuracies of between 93.2 to 90.6 % for detecting 6 to 12 keywords compared to other keyword spotting models within memory and compute constraints of edge devices. I deployed the CNN keyword spotting model on a number of edge devices with varying memory and compute capacities. The keyword spotting performance of the CNN model translated well to the ESP-EYE and Arduino NANO BLE boards. In case of the ESP-EYE board, the CNN model's correct classifications is only 0.6 to 1.7 % lower in terms of average classification accuracy for detecting between 2 to 12 keywords on a desktop PC using the identical model and parameters. Thus, I demonstrated the feasibility of achieving reasonably good keyword spotting performance for limited-vocabulary speech command inference on edge devices using my CNN based model to alleviate costs of speech command inference on the cloud. I implemented a similar deep learning pipeline to detect the presence of adventitious crackles and wheezes in breathing audio recorded from multiple chest locations. This is because the deployment of such a pipeline on edge devices may enable remote and automatic detection of crackles and wheezes in respiratory audio, which would be beneficial for the detection and monitoring of respiratory disorders. I used a hybrid convolutional and recurrent (CRNN) based neural network pipeline to detect crackles, wheezes and normal breathing in respiratory cycles. I calculated the precision of the pipeline in detecting crackles, wheezes and normal breathing in respiratory cycles recorded from different chest locations. The CRNN pipeline had the best precision results for classifying audio recorded from the anterior chest, with precision values of 86, 89 and 92 % for detecting crackles, wheezes and normal breathing in respiratory cycles. In the next phase of the project, with the proper clinical protocols and permissions, the CRNN model can be deployed on edge devices to monitor the percentage of crackles and wheezes detected in certain durations of breathing audio recorded from the anterior chest. This can potentially enable reliable detection of abnormal sounds in breathing that are symptomatic of respiratory conditions using low-cost and off-line monitors when medical specialists or network are not accessible.

## Table of Contents

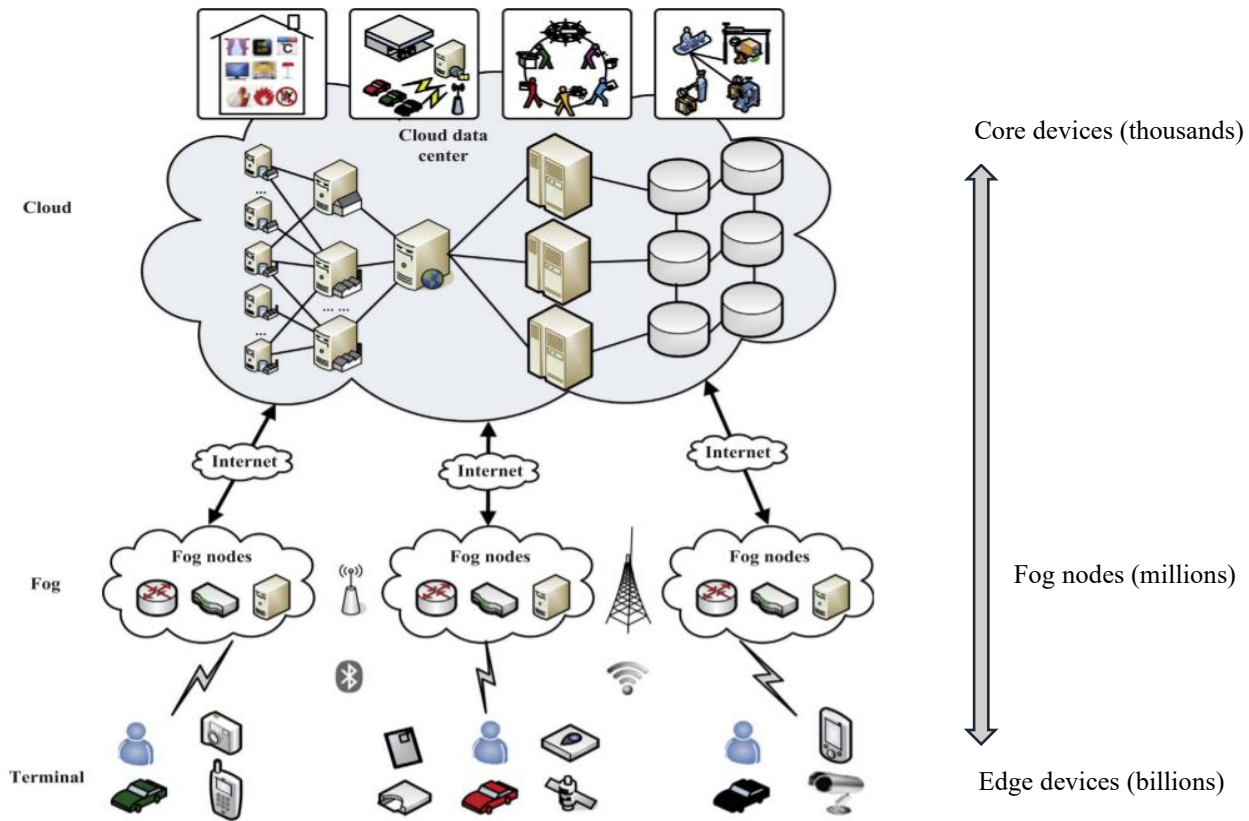
	Page
Abstract.....	2
Chapter 1: Introduction .....	4
Chapter 2: Background.....	7
Keyword Spotting (KWS) System .....	7
Microcontroller Systems (MCUs).....	8
Chapter 3: Keyword Spotting on MCUs .....	9
Convolutional Neural Network (CNN).....	11
Recurrent Neural Network (RNN) .....	13
Convolutional Recurrent Neural Network (CRNN) .....	14
Model Quantization.....	14
Chapter 4: KWS Evaluation results .....	16
Desktop computer evaluation.....	16
MCU deployment.....	20
Chapter 5: Respiratory Audio Classification .....	23
Chapter 6: Conclusion .....	29
References .....	31

## 1. Introduction:

In modern autonomous systems, machine learning (ML) has many applications in a wide spectrum of uses, from computer vision to language modeling. Desirable performance of deep learning models, i.e. accurate or reliable predictions from given input, can often come at the expense of high computation costs. The larger a deep learning model, the greater it's learning capacity and therefore it's predictive performance. As such, deep learning networks often consist of multiple computational layers wherein computations are performed with a large number of parameters in each layer per inference. The computational capacity required of a model increases with the size and/or dimensions of its input data. Deep learning models therefore have large computational needs for a wide range of practical applications. GPU servers in cloud systems or data centers are often required to meet such computational demands.

For consumer-grade ML tools such as monitoring the status of critical equipment, video surveillance or audio recognition, inference is expected to be done near real-time. Often such use cases require models to run on streaming data, and hence require continual or frequently recurring inferences. To achieve favorable results in such cases, it is desirable to optimize models and/or inference methods to minimize latency and resource usage while maintaining predictive reliability. Inferencing in the cloud introduces the following challenges for the use cases mentioned above: (a) It is costly to transfer data to the cloud for real-time or frequent inference (b) data transfer between the edge and cloud incurs latency and relies on connectivity (c) data transfer from the edge to the cloud introduces scalability issues with the increase in the number of connected devices, and (d) there exists privacy and security risks associated with sending data to the cloud.

Edge computing is a distributed computing framework which brings computation and data storage close to the location where it is required to improve response time and save bandwidth as well as energy. Though edge computing can potentially address or mitigate the mentioned cloud computing challenges, resource constraints on the edge means it can be difficult, if not unfeasible, to meet or outsource computational requirements of deep learning networks at the edge. However, given the vastly larger number of edge devices as compared to remote data servers across varied applications, the potential deployment of computational tasks on the edge can lead to more desirable and efficient allocation of resources. Hence, the optimal deployment of computational/machine-learning networks with edge IoT has become a relevant research area.



**Figure 1:** Illustration of the hierarchichal structure of interconnected computing [17].

Premised on optimizing the deployment of ML models on edge computing frameworks, the main focus of my project is the optimized implementation of voice command or audio recognition systems on edge platforms. Voice has turned into a pervasive method to manage and interact with tech devices, starting from its initial adoption in smartphones and smart speakers to its use now in smartwatches, home appliances and much more. Devices with voice command interfaces are expected to continue rising sharply over the next few years, given the advantages gained from controlling technology with speech. A study by Juniper Research has found there will be 8 billion digital voice assistants by the end of 2023, up from an estimated 2.5 billion at the end of 2018 [1]. Smartphones are the largest digital voice assistant platform by volume owing to the Google Assistant and Siri. Juniper Research predicts the fastest growing voice-automated devices over the next 5 years to be TVs, speakers and wearables respectively. Voice assistant applications can be wide-ranging; smart domestic devices range from TVs and speakers to appliances such as lighting and heating. Voice assistant technology is not only expected to become more sophisticated but will be integrated into a larger range of devices and applications. With more reliable implementations, they may also simplify or completely eliminate the need for hardware interfaces in many consumer products, saving cost and improving user experience.

Given the fast-growing adoption of voice assistants and their wide-varying use cases, the aforementioned connectivity, latency, scalability and security issues with cloud usage are

becoming increasingly significant for such applications. As such, my project is particularly focused on optimizing the implementation of speech recognition models for digital voice assistants with a very limited vocabulary using edge computing. To this end, my primary objective is the deployment of voice command recognition on edge devices for a specified set of localized tasks. Localized (local) tasks are those for which remote connectivity is not a requirement and real-time responsiveness is desired. In the context of smart home assistants, local tasks can include switching of lights, changing room temperature, etc. If a home voice assistant model localizes a set of tasks that require low latency or no internet connectivity to the edge and gets deployed at scale, it would substantially alleviate cloud and bandwidth usage costs/issues for local tasks, thereby improving efficiency.

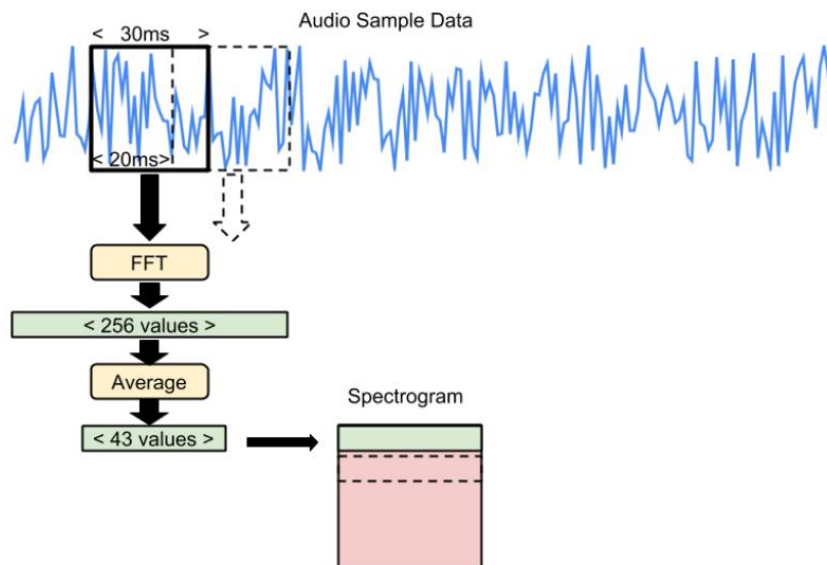
In order to localize a set of tasks to the edge, I developed a keyword spotting model optimized to perform inference for a specified group of keywords on edge platforms. The selected group of keywords can be used to recognize localized commands. Keyword spotting (*KWS*) is generally run on low-power edge devices to trigger full speech recognition on the cloud. Given *KWS* models run continuously, keyword triggers are usually limited to few words, such as ‘Ok Google’, to minimize power usage. Such a trigger word/phrase can be used to activate an edge-based speech command model to detect local commands before activating cloud-based inference if required. The keyword set of the edge-based speech command model can be configured to detect a set of local commands. For instance, room temperature can be controlled by local commands, where the keyword combination ‘increase temperature’ can be used to increment room temperature. Considering the resource constraints of edge devices, sets of 8-12 words were used to evaluate *KWS* models for edge deployment. *KWS* models were deployed on edge platforms of varying resource constraints. The potential expansion of keyword sets for edge *KWS* was also assessed.

The optimal deployment/migration of machine learning on the edge can potentially facilitate performance gains in other use cases where localized inference is desirable. The healthcare industry in particular has application space for ML on the edge. For instance, intensive care is an area which could benefit from edge-based ML as real-time processing and feedback on data is important to maintain critical physiological parameters. Wearable or embedded healthcare technology is another area where on-device AI has applicability. Cloud-based ML for the mentioned healthcare use cases can incur latency where real-time feedback is important or essential, create connectivity issues for critical operations or in remote locations, and incur high usage costs for frequent inference operations. As such, the latter focus of my project is on the implementation of deep learning on the edge for wearable/embedded healthcare, and I specifically developed a respiratory audio classifier for potential monitoring of respiratory health on edge hardware.

## 2. Background:

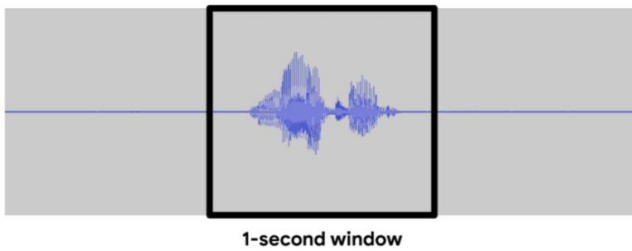
### 2.1 Keyword Spotting (KWS) System

The KWS system implemented consists of a feature extraction from raw audio signals into spectrograms that are fed as input to a neural network-based classifier to detect speech commands. The feature extractor module converts each second of audio to a 2D array of features that represents a spectrogram. Feature extraction is performed using Mel-frequency cepstral coefficients (MFCC), a traditional technique used to obtain speech features for deep learning-based speech recognition that translates time-domain speech signal into a set of frequency-domain spectral coefficients. In my KWS implementation, each 30-millisecond (ms) slice/sample of audio is translated into 40 frequency buckets using MFCC. Each 2D spectrogram is built by moving the 30-ms sample window forward with 20-ms stride until it covers a full second of audio. The resulting audio spectrogram consists of 49 consecutive 30-ms slices of audio containing 40 frequency buckets with an overlap of 10-ms between audio slices, and is ready to be passed into a model for inference.

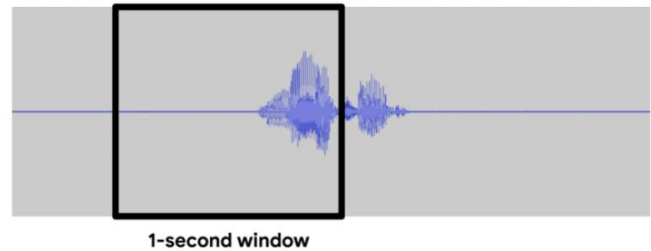


**Figure 2:** Diagram of audio signals being processed into spectrograms [2].

The KWS model outputs a set of probabilities for the presence of a keyword in a second of audio, where a keyword is detected if its probability is greater than a defined threshold. However, given keyword recognition is done on a continuous stream of audio, KWS will not be reliable if result of a single inference is taken as the classification result.



**Figure 3.a:** The word “noted” being captured



**Figure 3.b:** Part of the word “noted” being captured

The KWS model is applied repeatedly at fixed offsets of 20-ms, resulting in multiple inferences per second. Therefore, a spoken word or portions of it may appear in multiple one-second windows. To improve the reliability in detecting a spoken word, a command recognizer class is used which calculates the average score for each word over a number of inferences and detects a keyword if it’s score meets a confidence threshold.

For instance, Figures 3.a-b below show two one-second inference windows at different points of a waveform of the word “noted” being spoken. If the inference window in Figure 3.b captures the first syllable “no” in “noted”, a KWS model trained to detect the word “no” will likely interpret a high probability of “no” being spoken in this window. However, in this case the command recognizer class would average the score for the word “no” over a number of successive inference windows, which would likely result in a low score for the word “no”.

## 2.2 Microcontroller Systems (MCUs)

A typical microcontroller system (MCU) consists of a processor core and on-chip memory in the form of an SRAM block and an embedded flash. The application binary, usually preloaded into the non-volatile flash, is loaded into the SRAM at startup and the processor runs the application using the SRAM as the main data memory. Therefore, the size of the SRAM limits the memory size that the program can use.

Apart from the memory footprint, execution speed (i.e. operations per second) is a major limiting factor for running neural networks on microcontrollers. Most microcontrollers are designed for embedded applications which primarily target low cost and high power-efficiency, and therefore have low throughput for compute-intensive workloads as in the case of neural networks. Some microcontrollers have integrated DSP instructions, such as the Cortex M-4 and Coretx M-7, that can accelerate low-precision computation when running neural networks.



NN size	NN memory limit	Ops/inference limit
Small (S)	80 KB	6 MOps
Medium (M)	200 KB	20 MOps
Large (L)	500 KB	80 MOps

Table 1: Neural network constraints for KWS models considered in [3], assuming 10 inferences per second and 8-bit weights/activations.

Given memory footprint and execution time are the two main considerations when designing and optimizing neural networks for deployment on microcontrollers, the authors of [3] have derived three sets of constraints for running KWS neural networks based on typical microcontroller configurations which target small, medium and large microcontroller systems. The three sets of constraints are given in Table 1 and are derived factoring in resource requirements for other tasks such as OS, I/O, etc.

### 3. Keyword Spotting on MCUs

This section delves into the exploration of neural network architectures and parameters for the optimization and subsequent deployment of keyword spotting models on edge platforms.

#### **KWS dataset:**

I use the Google speech commands dataset [4] to develop and deploy KWS models. The dataset consists of 65K 1-second long utterances of 30 words by thousands of different people with each utterance containing only one word. I use keyword sets of 5-12 words to evaluate performance and potential of models for edge-based KWS. Given that words with similar pronunciations are harder to differentiate for classification, I use two types of keyword sets; one with a high percentage (60-70%) of words having one or more similarly pronounced words in the set, and another with an arbitrary selection of words. For a given number of keywords, I evaluate model performance on different combinations of each type of keyword set (with a total of between 5 to 7 combinations) and ensure no more than about 40% of keywords in each set have fewer than 5 characters to test model robustness. All model pipelines are developed using Google’s Tensorflow framework [5] with standard cross-entropy loss function and the Adam optimizer [6].

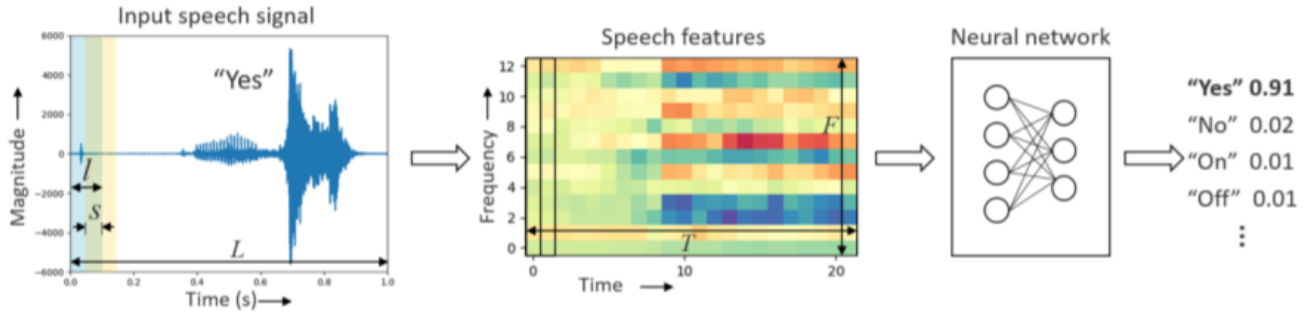


Figure 4: Illustration of the keyword spotting pipeline implemented [3]

### Feature extraction parameters:

As shown in Fig. 2, a spectrogram of  $T \times F$  features is extracted from the input signal where  $T$  is the number of audio frames and  $F$  is the number of MFCC features per audio frame. The number of these extracted features impact model size, number of operations and accuracy, and the key feature extraction parameters causing these impacts are the number of features per audio frame ( $F$ ) and the frame stride ( $S$ ).  $F$  impacts the number of weights in fully connected and recurrent layers, but not in convolutional layers as weight kernels of fixed sizes are used in convolutional layers. Both  $F$  and  $S$  impact the number of operations per inference and therefore classification performance. I use 40 MFCC features per frame ( $F$ ), 20-ms frame stride ( $S$ ) and 30-ms long speech frames to evaluate models. Making  $T \times F$  smaller, i.e., decreasing  $F$  and/or increasing  $S$  adversely impacted classification performance while doing the opposite increased the number of operations per inference without notable improvement in classification performance across evaluated models.

### Training parameters:

I use a train:test:evaluation split ratio is 80:10:10 for each keyword, making for a large proportion of training data as well as representative testing and evaluation data. In addition to keywords, models are trained to recognize an ‘unknown’ label and a ‘silence’ label. The ‘unknown’ label is used to classify speech falling outside the model’s keyword set as unknown and is trained using a mix of words from the speech dataset not included in the model’s keyword set. The ‘silence’ label is used to detect the absence of speech. An equal proportion of data is used to train and test each target label of a model (including the unknown and silence labels). For instance, if there are a total of 8 training labels, each label would be trained with 12.5 (100/8) % of the total training data. Both the unknown and silence labels are given the same proportion of training data as each keyword label to provide models reasonable number of training examples for differentiating between unknown words/audio, silence and known words.

Models are trained with a batch size of 100 and training data is augmented with background noise and random time shifts of up to 100-ms to resemble realistic environments and introduce distortion. Models are trained with an initial learning rate of  $1 \times 10^{-3}$ , which is reduced to  $1 \times 10^{-4}$  in the latter training phase to finetune model convergence as performance improvement begins to flatten. The number of training steps is increased with the number of training labels, with a total of 18 to 20k steps used to train models with 5 keyword labels. Trained model size, number of operations, classification accuracy and streaming performance on MCUs are evaluated.

The following provides an overview of the neural network architectures I explored and implemented for keyword spotting:

### **3.1) Convolutional neural network (CNN):**

Convolutional layers are the main building blocks of this type of network. A convolutional layer consists of kernels/filters of 2D weights applied on an (2D) input feature matrix across one or more channels in specified strides, resulting in feature maps locating and quantizing different features in the input space. An application of a convolutional filter is the dot product of the filter with a corresponding region of the input, and an activation function is applied to the resulting output value, typically ReLU activation. Batch normalization [7] can be applied before or after the activation function. A convolutional layer can be followed by a pooling layer which downsizes the convolutional layer's output feature map. Thus, CNNs enable the detection of highly differentiated input features for tasks such as image classification through the parallel learning of multiple convolutional filters. CNNs can efficiently capture the local temporal and spectral correlations in the feature spectrograms extracted from speech.

My CNN implementation for KWS is depicted in Figure 5. I explore the following model hyper parameters for the CNN model – convolutional layers: number of kernels, kernel size/stride, activation, pooling layers: pooling size/stride, dropout, and fully-connected layers. I evaluate my CNN model against a baseline CNN model provided in TensorFlow's speech recognition framework for keyword spotting [8]. The baseline CNN model has high test accuracy across all evaluated keyword sets (94-97%) but is large, using 940,000 weights parameters and taking more than 50 million FLOPS per inference. Thus, the baseline CNN cannot operate within the memory and compute constraints outlined in Table 1. I explored CNN model architecture and parameters to optimize KWS accuracy within the outlined MCU constraints.

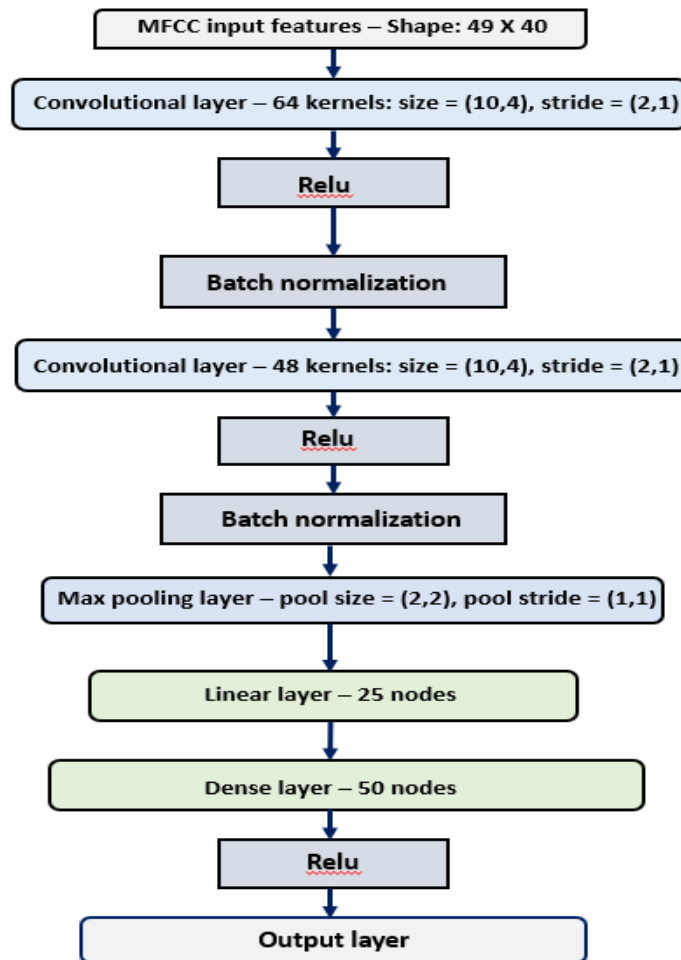


Figure 5: Model architecture of CNN

As seen in Figure 5, the MFCC input features are fed into two successive convolutional layers, enabling a higher level of feature differentiation than would be the case with one such layer. A third successive convolutional layer adds to model size without notable gain in classification performance. The number and dimensions of kernels for both convolutional layers were selected by searching and scaling for optimal configurations using the baseline CNN as reference. I use a larger number of kernels (64) in the first convolutional layer than the second one (48) to maximize information extracted from the input features. For a given kernel stride, the larger the kernel size (height X width), the fewer the number of convolutional operations but larger the number of weights in the convolutional layer. I experimented with kernel widths between 3 and 5 and kernel heights between 3 and 10 and selected kernel height X width of 10 X 4 for both convolutional layers after iteratively performing exhaustive search of kernel dimensions. Increasing the kernel stride or using a pooling layer after a convolutional layer down-samples the convolutional feature map, which reduces the number of weights in subsequent layers and can improve model performance by reducing overfitting or complexity. I use a kernel stride of 2 X 1 (height X width) in the convolutional layers and a maximum pooling layer with pool size 2 X 2 after the second convolutional layer.

Larger kernel strides or pooling sizes adversely impacted classification performance for the relatively larger keyword sets, likely because less information gets retained from the audio input given the compactness of the model. ReLU activation is applied to both convolutional layers.

The pooling layer connects to a linear layer with 25 nodes, followed by a fully connected (dense) layer with 50 nodes. The dense layer serves to improve learning capacity and the linear layer projects the pooling layer's output to a small number of nodes to reduce model size. The sizes of the linear and dense layers are selected with the aim of optimizing trade-off between model accuracy and size. The dense layer maps to the output layer which contains the model's logits/scores for its target labels. Dropout is applied after convolutional and dense layers during training to avoid overfitting.

### 3.2) Recurrent Neural Network (RNN):

RNNs are a class of neural networks that can capture sequential dependencies in data by using previous output as input to update its current state. During training, a recurrent layer updates a fixed set of weights at sequence step 't' using its current input and output from sequence step 't-1' by means of hidden state vector or 'gates' which encapsulate information from prior sequence steps.

RNNs thus capture temporal or sequential dependencies in input signals using its 'gating' mechanism to represent previous network states. RNNs have shown superior performance in many sequences modeling tasks, including speech recognition [9] and language modeling [10]. The model architecture of a recurrent layer as applied to my use case is shown in Figure 6. As seen in Figure 6, the recurrent layer operates for  $T$  time steps, where the input to the RNN at each time step  $t$  is the spectral feature vector  $f_t \in R^F$  concatenated with the previous time step output  $h_{t-1}$ . The RNN cell in the figure can be an LSTM cell [11] or a gated recurrent unit (GRU) cell [12]. Since recurrent layers don't use multiple weight filters, they tend to have fewer parameters than convolutional layers.

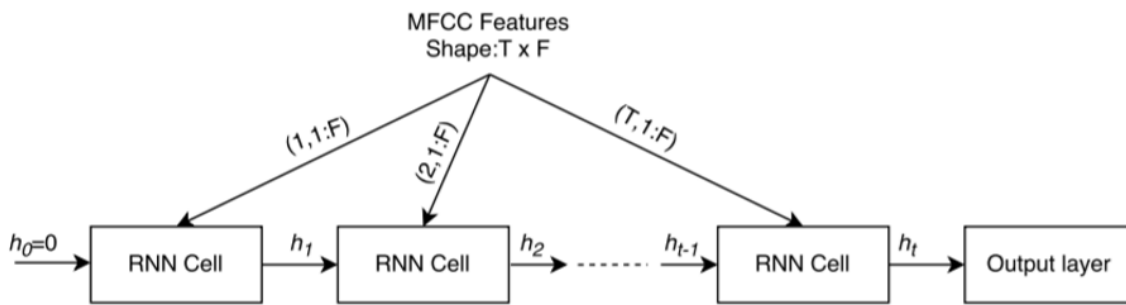


Figure 6: Recurrent layer architecture

I explore the size and number of recurrent layers as well as the use of dense layers for an optimized RNN implementation. Increasing the number of cells/nodes in a recurrent layer improves feature learning capacity, while stacking multiple recurrent layers enable a higher-level representation of features. I elected to use 2 LSTM layers with 100 cell per layer and the ‘sigmoid’ activation function in the recurrent layers. I use a dense layer with 50 nodes following the recurrent layers, which connects to the output layer. Model hyperparameters were selected with consideration to accuracy, computation and size trade-offs.

### **3.3) Convolutional Recurrent Neural Network (CRNN):**

Convolution recurrent neural network [13] incorporates both convolutional and recurrent layers to utilize the benefits of both types of networks. It can capture local temporal correlations using convolutional layers and global temporal correlations using recurrent layers in speech features. Figure 7 illustrates the CRNN model I implemented for KWS after hyperparameter and architecture exploration with convolutional, recurrent, and fully connected layers. As seen in Figure 7, the MFCC spectrogram is fed into a convolutional layer to create feature maps of the speech input, followed by a recurrent layer to capture the global dependencies in the feature maps. The recurrent layer connects to a dense layer to enable further performance improvement, followed by the output layer containing the classification logits.

### **3.4) Model Quantization:**

All trained models were quantized for deployment on MCUs. Quantization is the compression of any combination of model weights, activations, input or output to lower precision values in order to reduce model size and runtime latency. However, the quantization of a model can degrade its accuracy. I applied dynamic range quantization on models, which is a post-training quantization method provided by the Tensorflow Lite converter [14]. In dynamic range quantization, trained model weights are statically quantized from floats to 8-bit integers and ‘dynamic range’ operators dynamically quantize activations to 8-bits based on their range and then perform computations with 8-bit weights and activations.

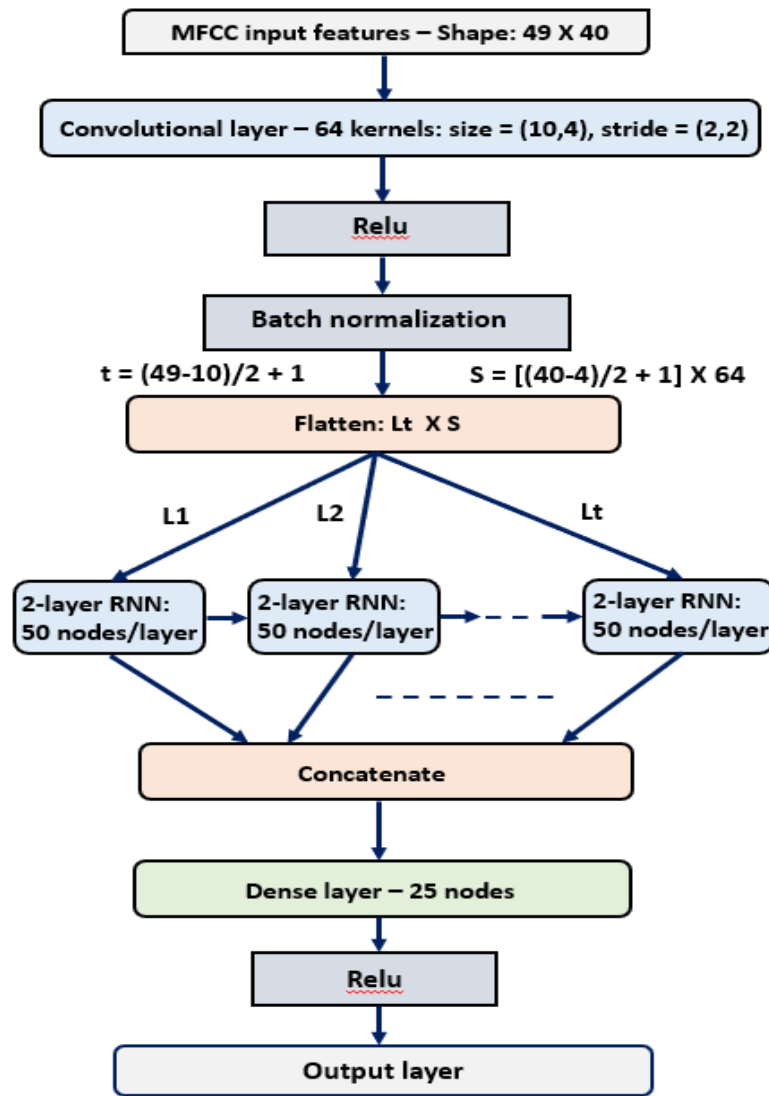


Figure 7: Model architecture of CRNN

Dynamic range quantization compresses my trained models to fit within the resource constraints of target edge devices, with relatively little degradation in classification performance as compared to full integer quantization. Full integer quantization makes all model math integer quantized and therefore enables further latency improvements and reduction in model size than dynamic range quantization, but with substantial degradation in classification performance. Quantization results were factored into my hyperparameter selection process for KWS models. For instance, I increased the number of convolutional kernels in my CNN model to improve its learning capacity in response to degradation in test classification accuracy of the CNN model after quantization.

#### 4. KWS Evaluation Results:

Size, average accuracy and operations per inference of my quantized models are given in Table 1, where the average accuracy is calculated for keyword sets of size 6 to 12. As seen in the table, the CNN-1 model has the best accuracy results across all tested keyword sets among the models, likely because the CNN-1 architecture best captures both spatial and temporal correlations in the speech features. The RNN model takes more number of operations compared to the CNN, as it's recurrent layers are more compute intensive than the CNN's convolutional layers,. The CRNN model has better accuracy results than the RNN, probably because it exploits the characteristics of both convolutional and recurrent layers. While there is no major difference in size between the 3 models in Table 2 and the CRNN model takes less operations per inference than the CNN, I deploy the CNN models on edge MCUs given its superior classification performance within the medium and large resource bounds in Table 1.

The CNN-1 model has between 327000-329000 weight parameters depending on the number of output labels. The size of the quantized CNN-1 model has size of about 331 KB, including weights, activations, I/O and MFCC features. I use a smaller version of the CNN-1 model, labelled as CNN-s, to implement KWS for word sets with 5 or fewer words as comparatively good performance can be attained with less compute capabilities than CNN-1 for such keyword sets. The CNN-s model is optimized to perform KWS for 5 or fewer words on the two smaller target edge devices used in this work, both of which cannot fit CNN-1 and fall in the medium size category in Table 1. The CNN-s model has about 174000 weight parameters and quantized size of about 156 KB.

Desktop computer evaluation:

I present evaluation results of running KWS models on a desktop computer with sufficient memory and compute capacities to run the models in this section

NN Architecture	Average Accuracy				Memory	Ops/inference
	KWS-6	KWS-8	KWS-10	KWS-12		
CNN-1	93.20%	92.40%	91.80%	90.60%	331 KB	22.4M
RNN	91.60%	90.30%	88.60%	87.10%	328 KB	25.2M
CRNN	92.30%	91.10%	89.70%	88.20%	335 KB	20.2M

Table 2: Comparison of quantized model size, accuracy, and operations/inference across keyword sets of size 6-12, where KWS-6 denotes keyword sets of size 6, KWS-8 denotes keyword sets of size 8 and so on.



Average accuracies of CNN models across keyword test sets of size 2 to 12 are shown in Table 2. Results for keywords sets of upto 5 words are from the CNN-s model and results for keywords sets with more than 5 words are from running inference with CNN-1. ‘KWS-Sim’ in the table refers to keyword sets with a high percentage of similarly pronounced words (as described in Section 3) and ‘KWS-Reg’ refers to keywords sets with an arbitrary selection of words.

As seen in Tables 2 and 3, the CNN models have reasonably reliable KWS performance given compute constraints, with the average test accuracies across tested keyword sets ranging between 93.3-90.6% depending on the size of the keyword set. The drop in accuracies for ‘KWS-Sim’ as compared to ‘KWS-Reg’ are reasonably small, with a maximum drop in accuracy of 1.5% in case of 12 keywords, which demonstrates robustness of models for difficult classification tasks. There is steady but no unreasonably sharp decline in accuracies with the increase in number of keywords.

Figure 8 shows a comparison of average KWS accuracies of the baseline CNN model with my CNN-1 and CRNN models (my RNN model is not included in the comparison as it has worser KWS performance than the CRNN). The baseline CNN has accuracies ranging from about 97.5 to 94 % across keyword sets of size 2 to 12. The drop in average test accuracies from the baseline to CNN-1 are reasonably small, ranging between 3 to 3.4 % over evaluated keyword sets, given the size and number of operations per inference of the baseline CNN are more than twice of those of the CNN-1 model. The CRNN has comparable KWS performance with CNN-1 for keyword sets of upto 5 words, but the CNN-1 has notably better KWS accuracies for keyword sets of more than 5 words.

	Average keyword spotting test accuracy of CNN models						
No. of keywords	2	4	5	6	8	10	12
KWS-Sim	92.7%	91.3%	90.7%	92.6%	91.7%	91.2%	89.8%
KWS-Reg	93.6%	92.6%	92.1%	93.7%	92.8%	92.2%	91.3%

Table 3: Average test accuracies of CNN models across keywords sets. CNN-2 results are displayed for 5 or less keywords while CNN-1 results are displayed for 6 or greater keywords.

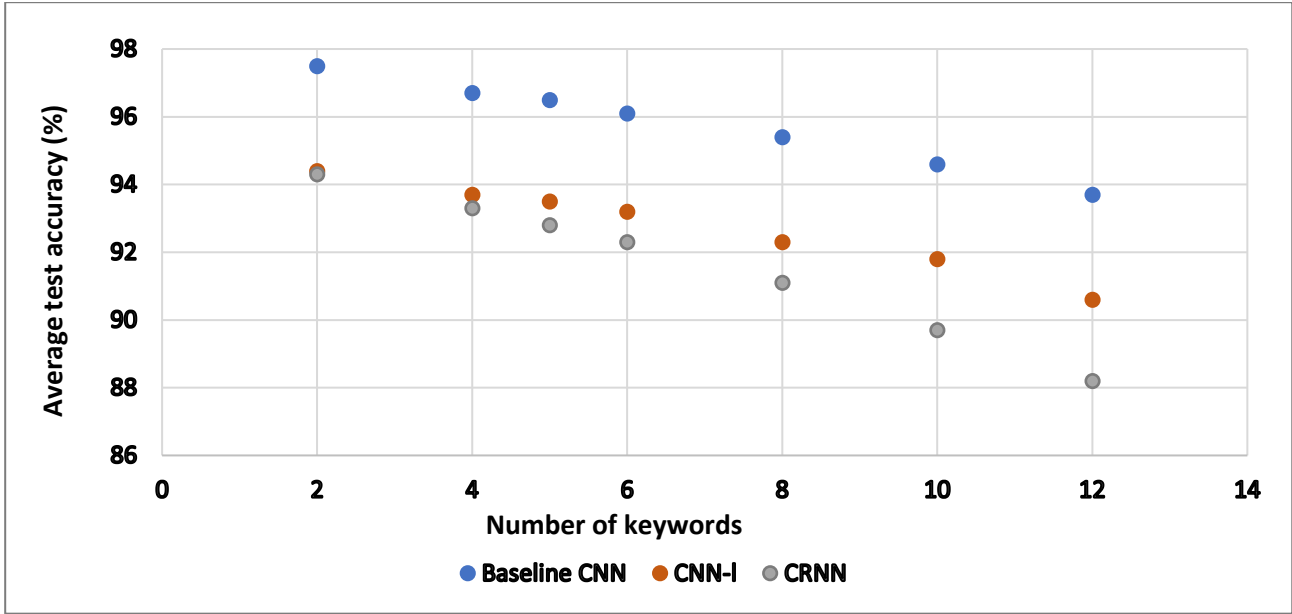


Figure 8: Average test accuracies of CNN-1, CNN-2 and Baseline CNN across different number of keywords.

Despite the steady decline in KWS performance with the increase in number of keywords, the CNN-I model produces reasonably good KWS performance across evaluated numbers of keywords, with the average KWS accuracy for a given number of keywords greater than or equal to 90.5%. My CNN architecture produced optimal results among evaluated models, hence I present evaluation results of deploying my CNN-I and CNN-s models on edge devices belonging to the large and medium categories defined in Table 1 respectively.

The authors of [3] have also performed neural network architecture exploration to optimize keyword spotting models for edge deployment using the Google speech commands dataset. Figure 9 shows the comparison of their KWS models with my corresponding models for 10 keywords overlayed with medium (M) and large (L) memory and compute bounding boxes for neural networks from Table 1. In order to enable inference on the edge for a reasonable or desirable number of speech commands, a model trained to detect a sufficient number of keywords (which can potentially be more than 12 words) needs to be deployed. As such, I compare models between the M and L boxes in Figure 9 as KWS performance on the edge of evaluated models within the M box is considerably worse compared to those between M and L for keyword sets of more than 6 words. This is because 6 or fewer words can be insufficient to cover a desirable range of local speech commands for my use case.

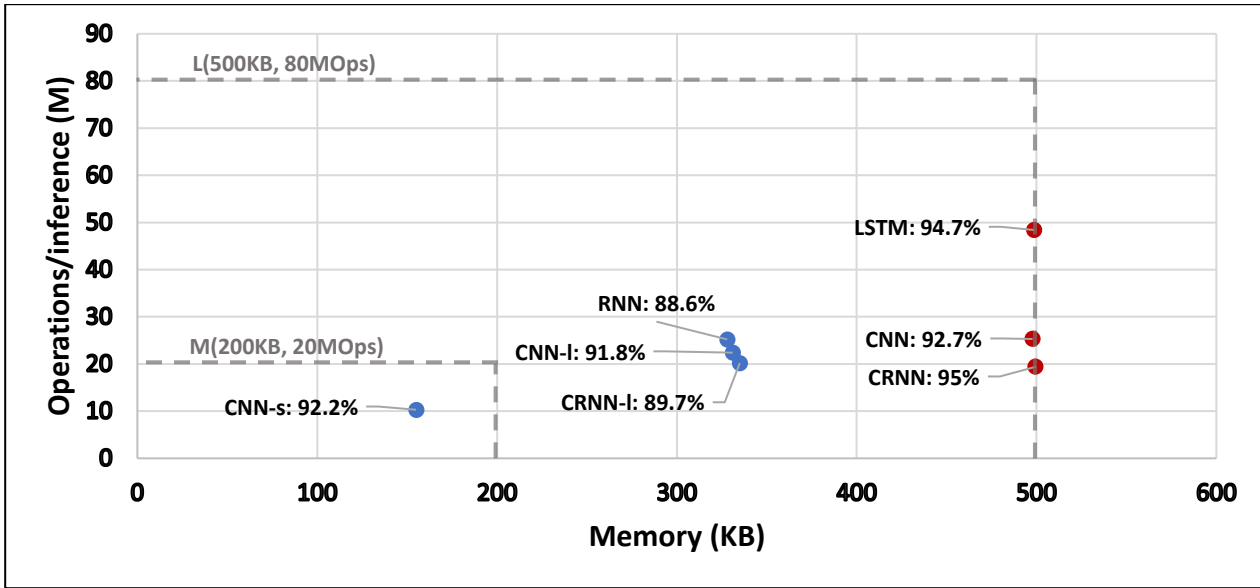


Figure 9: Memory, operations/inference and average test accuracies of KWS models. Models from this work are shown in blue while models from [3] are shown in red.

In context of Figure 9, the optimization of a model would involve bringing it closer to the origin while maintaining good/high KWS accuracy. As seen in Figure 9, my CNN-l, CRNN and RNN are well within the large memory and compute bounds, with the CNN-l having the best average classification accuracy and the CRNN taking the least number of computations among them. The CRNN from [3] has the best classification accuracy among the models in Figure 9, with 3.2% higher accuracy than CNN-l. While the large models from [3] have higher accuracies than CNN-l, they are at the edge of the L memory limit and between 167-170 KB larger than CNN-l. Among the models from [3] in Figure 9, the CNN and CRNN takes about 3 million more and fewer computations than CNN-l respectively, while the RNN takes more than twice the number of computations of CNN-l. Hence, there is scope to reduce if not close the gap in KWS accuracies between my models and models from [3], which is between 0.9 to 3.2 % in the case of CNN-l, with further hyperparameter exploration within L bounds given the substantial differences in memory and/or compute requirements between the models from [3] and CNN-l.

It should be noted that the models from [3] were not evaluated on a range of 5 or more keyword sets comprising of similarly pronounced and arbitrary words with no more than 40% of words having fewer than 5 characters as was the case with my models. If KWS accuracies of the models from [3] were averaged across keyword datasets similarly to my model evaluations, their accuracy results can be expected to drop, in which case the CNN-l can potentially gain similar or better KWS performance within L bounds compared to one or more of the models from [3]. The models from [3] were only trained

NN Architecture	Test accuracy	Memory	Operations/inference
CNN-1 [15]	90.70%	556 KB	76.02 MOps
CNN-2 [15]	84.60%	149 KB	1.46 MOps
CRNN [13]	87.80%	298 KB	5.85 MOps

Table 4: Neural network accuracy on Google speech commands dataset from literature. CNN-1, CNN-2 are *cnn-trad-fpool3*, *cnn-one-fstride4* architectures from [15].

on 10 keywords and would exceed the L memory limit if trained on 12 or more keywords, whereas there is scope to improve KWS accuracies of my models within L bounds for 12 if not more words. The authors of [3] did not provide evaluation results of deploying their models on edge platforms, hence it is not known how their models' performances would translate to edge MCUs, whereas I evaluated CNN-1's performance on edge devices. Test accuracy of CNN-s for keywords sets of 5 words is shown in Figure 9. CNN-s is well within M bounds and is meant to provide reasonable KWS performance for small keyword sets in my use case.

Table 4 shows test accuracy of other KWS models from literature trained on the Google speech commands dataset [4]. As seen in the table, the CNN-1 has lower test accuracy and much larger memory and compute requirements than CNN-2. The CNN-2 and the CRNN models require less memory and compute resources but with lower test accuracies than CNN-1.

MCU deployment:

I deployed KWS models on the edge development boards shown in Table 5. The Arduino Nano BLE has SRAM of 256 KB, and hence can only fit small to medium sized KWS models as per Table 1. The Sparkfun Edge has full clock speed of 48 MHz, so it cannot support the operation speeds of the large KWS models. I deployed CNN-s on the Arduino and Sparkfun boards to run keyword spotting for 5 or fewer words. The STM32 and ESP-EYE boards both have sufficient memory and operational speeds to run the CNN-1 model, although the STM32 board has just about enough SRAM to fit CNN-1 while the ESP-EYE has SRAM much larger than 500 KB and can therefore fit any large size KWS model. I deployed CNN-1 on the STM32 and ESP-EYE boards to run keyword inference for up to 12 words.


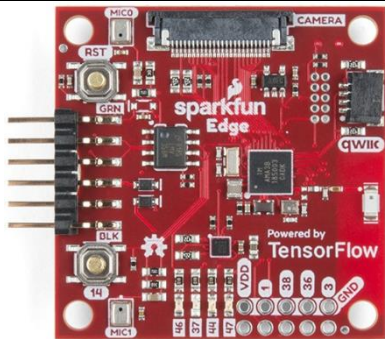
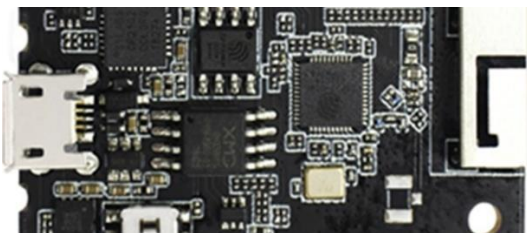
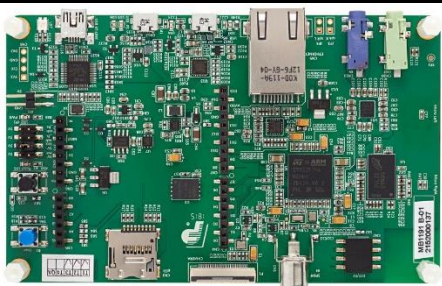
Edge development board	SRAM	Operating speed	Images (not to scale)
Arduino Nano BLE	256KB	64MHz	
SparkFun Edge	384KB	48MHz	
ESP-EYE	8MB	240 MHz	
STM32F746G-DISCO	340KB	216MHz	

Table 5: SRAM and CPU clock frequency of edge development boards used in this work to run KWS.

To evaluate streaming KWS performance of my CNN models on the edge, I generated test scripts of keywords and unknown words given a keyword set to be spoken into the microphones of edge development boards running the CNN models. The scripts are 3 minutes long each, with intervals between words ranging between 1 to 5 seconds. I generated test scripts for each keyword set used to obtain the average test accuracy results of models shown in Figure 8 and had the scripts spoken by 4 people with different accents comprising of 2 females and 2 males (including myself) to compare

model performance before and after edge deployment. Each person spoke out the test script into the microphones of the edge devices shown in Table 5 in realistic environments with varying levels of background noise. Atleast three test scripts of were recorded for each volunteer for a given number of keywords. Model classification results for each spoken test script were recorded and compared with the groundtruth, which are the correct labels corresponding to the respective test script.

To obtain predictions on streaming audio data on the edge, the model is applied repeatedly at fixed offsets in time and the inference results averaged over a specified window. As mentioned, this ensures the entirety of a word is captured in streaming audio if the sampling rate is sufficiently high. After searching for optimal streaming audio signal processing parameters , I set the length of time to average inference results over to 1000 ms, the suppression window which stops subsequent word detections after one is found to 500 ms, and the detection threshold which controls how high the average score of a model label must be to classify it as detected to 0.7.

Figure 10 shows the streaming KWS performance of my CNN models on edge platforms. The figure shows the percentage of correctly matched classifications (i.e., recall) across all spoken test scripts for a given number of keywords, including classifications for unknown words and silence. Misclassifications in this case would include incorrect labels and false negatives for spoken words and false positives for keywords. The CNN- is run on the Arduino Nano and Sparkfun boards to detect 2 to 5 keywords while the CNN-1 is run on the ESP-EYE and STM32 boards to detect 2 to 12 keywords.

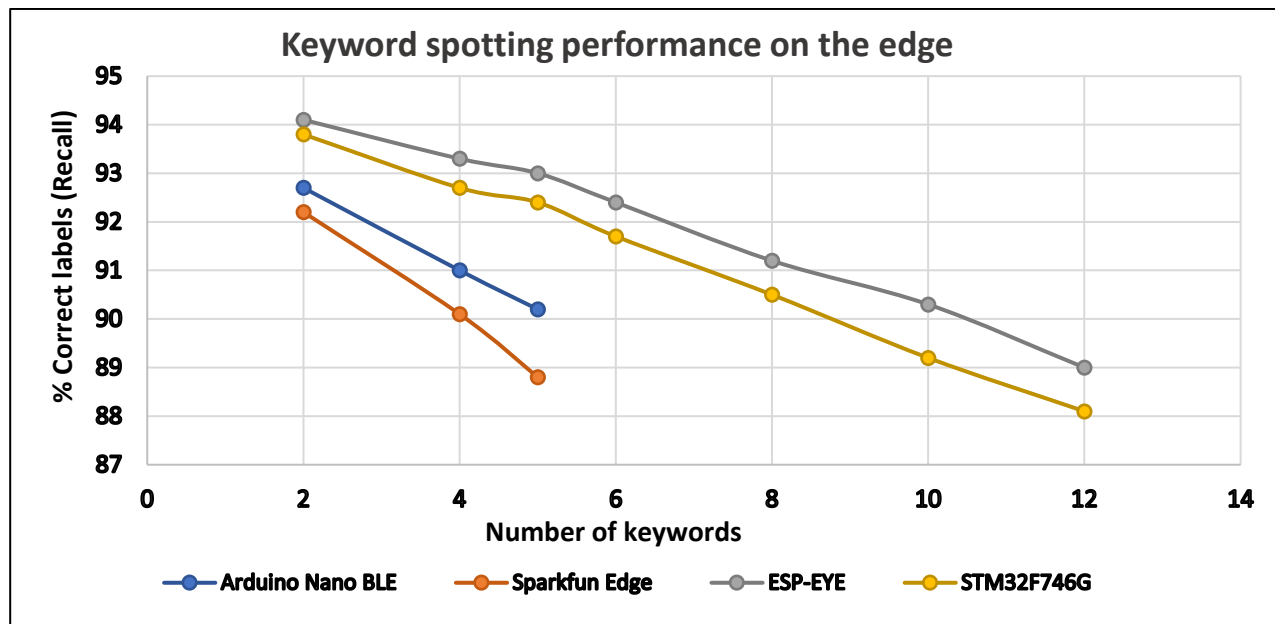


Figure 10: Classification performance of CNN models on edge platforms for keyword spotting of 2 to 12 words.

The CNN-l's classification accuracy on the ESP-EYE for a given number of keywords in Figure 10 is reasonably close to its average test accuracy shown in Figure 8 for the same number of keywords. CNN-l's recall when running on the ESP-EYE is between 0.6 to 1.7 % lower than its respective test accuracy when running on a desktop PC (shown in Figure 8) for detecting 2 to 12 keywords, while the corresponding difference in CNN-l's recall on the STM32 board and its average test accuracy on a desktop PC is greater, between 0.6 to 2.5 %. Some misclassifications on the edge devices can be attributed to errors or discrepancies that occur in practice as opposed to classification errors by the KWS model, such as the edge microphone failing to effectively capture audio as it is relatively low-power or due to human error. The CNN-l's KWS performance translates well to the ESP-EYE as the board has specialized audio capturing and processing units for speech recognition AI as well as sufficient memory and operating speed to run the CNN-l models. CNN-l's KWS performance is somewhat poorer on the STM32 board compared to the ESP-EYE. This is likely because the STM32 board does not have specialized audio processing hardware like the ESP-EYE and it's memory is near capacity when running CNN-l models, making it more limited in data processing capacity and therefore more prone to errors when running CNN-l compared to the ESP-EYE.

The CNN-s has good classification performance running on the Arduino NANO with respect to its average test accuracies, with its recall on the Arduino NANO being between 0.5 to 1.5 % lower than its respective average test accuracy on a desktop PC for detecting 2 to 5 keywords. CNN-s has poorer classification performance running on the Sparkfun board than on the Arduino NANO, with an increasing difference in the recall accuracies between the two boards with increase in number of keywords. The Sparkfun board becomes more prone to errors and/or missed inferences with increasing model complexity when running CNN-s due to its low operating speed. The Arduino NANO is therefore a good candidate to run low-power KWS for a small number of upto 5 or 6 keywords.

## **5. Respiratory Audio Classification:**

There is potential utility in deploying machine learning on the edge for healthcare use cases, including remote/embedded health monitoring. As such, I implemented a deep learning pipeline similar to the pipeline I used for keyword spotting to classify breathing sounds for respiratory health monitoring on edge devices. Respiratory sounds are important indicators of respiratory health, and are indicative of the presence of respiratory disorders. For example, a wheezing respiratory sound is a common sign of

the presence of an obstructive airway disease like asthma. Respiratory sound can be digitally recorded using a digital stethoscope or other recording techniques, which opens up the possibility of using machine learning to automatically diagnose or monitor respiratory disorders like asthma, bronchiolitis and pneumonia, to name a few. As such, the use of edge technology to record and classify respiratory audio can potentially enable patients to remotely and automatically monitor their respiratory health, which will be beneficial for the diagnosis and treatment of respiratory disorders.

I used the Respiratory Sound database originally compiled to support the scientific challenge organized at Int. Conf. on Biomedical Health Informatics (ICBHI) in 2017 [16] to develop my respiratory audio classification pipeline. The Respiratory Sound Database was created by two research teams in Portugal and Greece and includes 920 annotated recordings containing respiratory cycles taken from 126 patients. There are a total of about 5.5 hours of recordings containing 6898 respiratory cycles which contain either crackles, wheezes, both crackles and wheezes, or normal breathing. The recordings consist of both clean respiratory sounds as well as noisy audio that simulate real life conditions. The patients span all age groups, i.e, children, adults and the elderly, with each patient having one of a comprehensive range of respiratory disorders or no respiratory disorder.

Table 6 shows the provided demographic and respiratory audio recording information as well as the list of respiratory health conditions of patients in the Respiratory Sound Database [16]. Audio samples were collected from 126 patients, numbered 101 to 226, and demographic information listed in the table was collected for each patient. For each patient, audio samples of varying lengths between 10 to 90 seconds were collected from six chest different locations, with each audio sample obtained in either single-channel or multi-channel acquisition mode using one of the recording equipments listed in the table. Each patient had been diagnosed with one of the 8 respiratory health conditions listed in the table prior to the recording of their audio samples.



Each audio sample is annotated with the start and end of respiratory cycles, as well as the presence/absence of crackles and presence/absence of wheezes in respiratory cycles. My objective was to implement a deep learning pipeline to detect whether breathing sound has crackling, wheezing or is normal. As such, I segmented the respiratory cycles from all samples to four categories: ones containing only crackles, ones containing only wheezes, ones containing both crackles and wheezes and ones that are normal. Then, I split data in each of the four categories into training and test datasets in the

Demographic information	Audio recording information		Respiratory health diagnoses
Patient number	Patient number:	101, 102,...,226	COPD: Chronic Obstructive Pulmonary Disease
Age	Chest location:	a. Trachea (Tc)	LRTI: Lower Respiratory Tract Infection
Sex		b. Anterior left (Al)	URTI: Upper Respiratory Tract Infection
Adult BMI (kg/m2)		c. Anterior right (Ar)	Asthma
Child Weight (kg)		d. Posterior left (Pl)	Bronchiectasis
Child Height (cm)		e. Posterior right (Pr)	Bronchiolitis
		f. Lateral left (Ll)	Pneumonia
		g. Lateral right (Lr)	Healthy
	Acquisition mode:	a. sequential/single channel (sc)	
		b. simultaneous/multi channel (mc)	
	Recording equipment:	a. AKG C417L Microphone (AKGC417L)	
b. 3M Littmann Classic II SE Stethoscope (LittC2SE			
c. 3M Litmmann 3200 Electronic Stethoscope (Litt3200)			
d. WelchAllyn Meditron Master Elite Electronic Stethoscope (Meditron)			

Table 6: Provided demographic and audio recording information and list of respiratory health diagnoses of patients in the Respiratory Sound Database [16]

train:validation:test ratio 75:5:20, with no overlap in patients and audio samples between the datasets and a similar distribution of age groups in each dataset.

Figure 11 shows the time and frequency domain characteristics of normal, crackling and wheezing lung sound cycles. As seen in the figure, the three types of lung sounds have distinctive characteristics. A wheeze is a continuous and adventitious waveform lasting around 250 ms or more and has a lower frequency range than a crackle. A crackle is discontinuous and explosive in nature, usually lasting for 1 to 10ms and has a distinctly wide frequency range up to 2000 Hz. Normal breathing has a lower frequency range than either a crackle or wheeze. Hence, a deep learning pipeline similar to the keyword spotting pipelines I implemented can be used to obtain a spectrogram of a respiratory cycle and classify whether the cycle is normal, has crackles, or has wheezes.

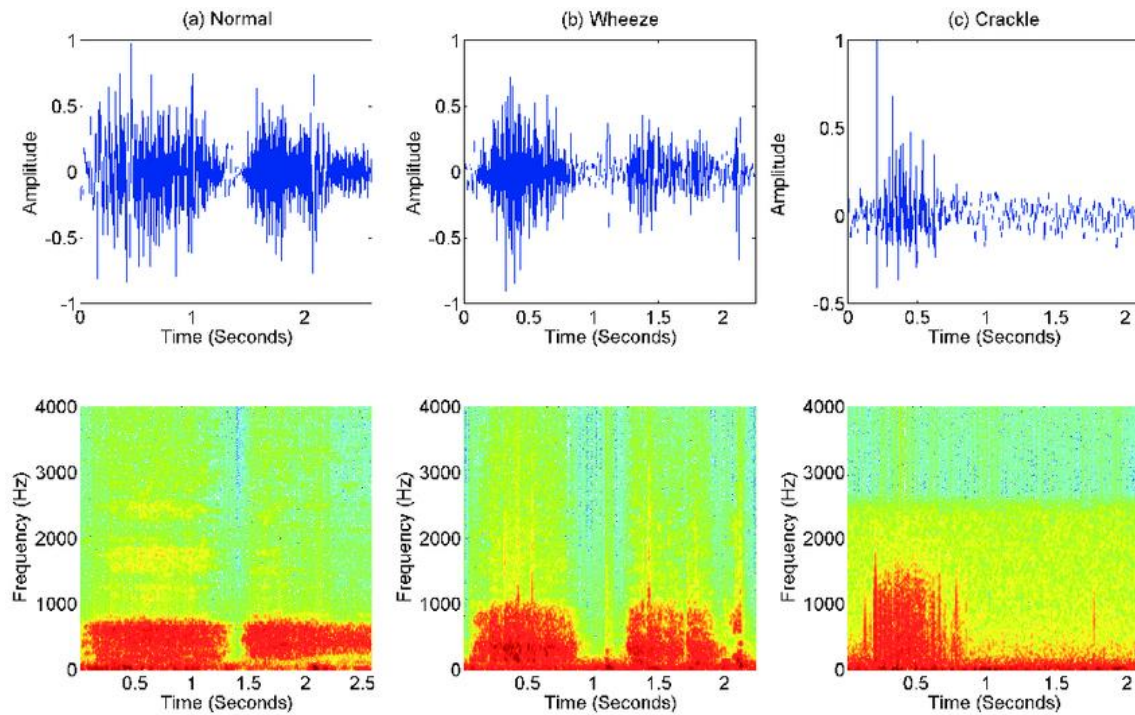


Figure 11: Time-domain and spectrogram characteristics of (a) normal, (b) wheeze and (c) crackle lung sound cycle. [18]

I featurized respiratory cycles to mel-spectrograms having a frame width of 20ms and a frame stride of 10ms with 26 frequency bands per frame fed as input to the deep learning classifier. I use a smaller number of frequency bands for respiratory audio compared to that for keyword spotting given the lower range of frequencies of breathing sound compared to that of speech. The chest location and acquisition mode of each recording were also appended as a feature column to its mel-spectrograms. I use the CRNN architecture I implemented for keyword spotting illustrated in Figure 7 to classify respiratory cycles, as it yielded better classification results than the CNN or RNN architectures described in Sections 3.1 and 3.2 respectively. However, I use 48 kernels in the convolutional layer and 20 nodes in the linear layer of my CRNN model for respiratory audio inference after exploration for optimal model parameters. The CRNN model is trained for 15k steps with an initial learning rate of  $1 \times 10^{-3}$  that reduced to  $1 \times 10^{-4}$  for the last 3k steps to finetune model convergence.

I calculated the precision in detecting crackles, wheezes and normal breathing in test data of respiratory cycles containing only crackling, only wheezing, and only normal breathing respectively. Precision, or the true positive rate, is calculated as follows:

$$\text{Precision} = \frac{T_P}{T_P + F_P},$$

Where  $T_p$  and  $F_p$  are the number of true positives and false positives respectively. In this case, the primary objective is the detection of the persistent presence of adventitious crackles and/or wheezes as opposed to the detection of every occurrence of adventitious crackles and/or wheezes in breathing. Hence, precision is used as the performance metric rather than accuracy to classify respiratory cycles, given crackles and wheezes were detected by my CRNN pipeline in every original audio recording from [16] at least 15 seconds long that contained crackles and wheezes respectively.

Respiratory cycle	Trachea	Anterior	Posterior	Lateral
Crackle	84%	86%	81%	82%
Wheeze	86%	89%	83%	86%
Normal	90%	92%	89%	90%

Table 7: Precision results of CRNN pipeline in detecting crackles, wheezes or normal respiratory cycles in audio recordings from different chest locations obtained from [16].

Table 7 shows precision results of the CRNN pipeline in detecting crackles, wheezes and normal respiratory cycles in test data from each of the four different audio recording locations of the chest. Each precision result provided in the table is calculated for respiratory cycles in test data from both the left and right sides of the corresponding chest location containing: only crackles, only wheezes, or only normal breathing. As seen in the table, the CRNN pipeline has the best precision results in detecting crackles, wheezes and normal respiratory cycles in recordings from the Anterior chest area. The quantized CRNN pipeline used in this case can fit in large edge devices as defined in Table 1. Hence, it is worth exploring means to deploy the CRNN model on edge devices to detect adventitious crackles or wheezes in respiratory cycles recorded over different durations from the anterior chest, given such an implementation can potentially enable remote and automatic detection or monitoring of respiratory disorders.

### **Future work:**

So far, we have been able to establish the necessary ML algorithms and parameters for the KWS on edge devices. We also showed that we can utilize this framework to process and classify chest sounds for respiratory abnormalities such as wheezing and crackles. However, the logistical elements required to fully implement a CRNN pipeline on edge devices for detecting the presence of adventitious crackles or wheezes in respiratory audio could not be obtained for this work yet. For this to happen, firstly, we would ideally need an even larger database for training and improving the model. Then we should also develop standardized means of capturing and transmitting respiratory audio to the CRNN pipeline running on an edge device and an adequate pool of patients. Such clinical work would also require specialized IRB protocols in place. Hence, first and foremost, the current work should be expanded to include additional data from other respiratory sound libraries to further improve the model. Next, following the necessary permissions and recording protocols, we should explore means to effectively deploy the CRNN pipeline on edge devices for tests on real patients. We also intend to explore additional signal processing and algorithmic approaches to more accurately detect the presence of co-existing crackling and wheezing in a respiratory cycle. Moreover, other ailments that express themselves on the chest sounds such as rhonchi or stridor and can be also considered subsequently. The CRNN model's classification performance for different demographic groups can be analyzed in future work to investigate any possible impact of demography on model performance. Further exploration of deep learning model architectures and parameters can also be done to see if we can improve upon the classification performance of the CRNN pipeline on respiratory audio recordings from one or more chest locations or types of microphones.

## 6. Conclusion:

The deployment of neural networks on edge MCUs rather than the cloud can enable performance gains in terms of lower latency, remote operability, lower bandwidth usage and better security for various applications. This may also improve the design of control and command inferences for digital systems and medical applications such as monitoring the health of patients or equipment. As such, I explored the optimization of deep learning networks to detect keywords for speech command inference on edge devices. I specifically explored neural network architectures, parameters to obtain MFCC input features from audio for keyword spotting models as well as neural network model training parameters in order to deploy optimized keyword spotting pipelines on edge devices. I quantized my keyword spotting models dynamically [14] from 32-bit floating point to 8-bit fixed point precision to fit the models within memory and compute bounds of edge MCUs. Among the keyword spotting models I explored, my quantized CNN models has the best average classification accuracies for detecting 2 to 12 keywords. The CNN-l model has relatively good KWS performance compared to other models within large edge device constraints [3], with average test accuracies between 93.2 to 90.6 % for detecting 6 to 12 keywords. The CNN-s model provides reasonably good KWS performance for detecting 2 to 5 keywords within medium edge device constraints [3]. I deployed the CNN-l and CNN-s models to large and medium size edge devices respectively. The CNN-l model's KWS performance translates well to the ESP-EYE edge board, with the model's percentage of correct classifications when running on the ESP-EYE between 0.6 to 1.7 % lower than its average classification accuracy when running on a desktop PC for detecting 2 to 12 keywords. The CNN-s model's KWS performance translates well to Arduino NANO BLE board for detecting 2 to 5 keywords. Hence, I demonstrated the feasibility of deploying my CNN models on edge MCUs to detect a reasonable range of keywords for on-device speech command inference.

I developed a deep learning pipeline similar to my CRNN pipeline for KWS to detect the presence of adventitious crackles or wheezes in breathing audio within resource constraints of edge MCUs. This is because such an implementation may enable remote and automatic monitoring of respiratory health on edge devices, which would be beneficial in the diagnosis and treatment of respiratory disorders. I elected to use a hybrid convolutional and recurrent based neural network model (CRNN) after exploration for optimal deep learning network parameters to detect crackles, wheezes and normal breathing in respiratory audio. I calculated the precision of my CRNN model in detecting crackles, wheezes and normal breathing in respiratory cycles recorded from

four different chest areas [16]. The model's best precision results were for classifying audio recordings from the Anterior chest area, with precision values of 86, 89 and 92 % for detecting crackles, wheezes and normal breathing respectively. As such, it is worth exploring means to deploy the CRNN pipeline on edge devices to determine the presence of crackles and wheezes over different durations of breathing audio recorded from the Anterior chest area. This will potentially enable novel and low-cost tools to remotely and automatically detect adventitious crackles and wheezes in breathing audio.

## References:

- [1] Juniper Research. “Digital Voice Assistants in Use to Triple to 8 Billion by 2023” Driven by Smart Home Devices”. Web blog post. 12 February 2018. Available at: [web URL](#).
- [2] Pete Warden, Daniel Situnayake. “TinyML machine learning with TensorFlow Lite on Arduino and ultra-low power microcontrollers”. O'Reilly Media Inc. December 2019.
- [3] Zhang, Yundong & Suda, Naveen & Lai, Liangzhen. (2017). “Hello Edge: Keyword Spotting on Microcontrollers”. [arXiv:1711.07128v3](#). 14 Feb 2018.
- [4] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. [arXiv:1804.03209v1](#). April 2018.
- [5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. [arXiv:1603.04467v2](#). 16 Mar 2016.
- [6] [Diederik P. Kingma](#), [Jimmy Ba](#). “Adam: A Method for Stochastic Optimization”. [arXiv:1412.6980v9](#). January 2017.
- [7] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. International Conference on Machine Learning, volume 37, pages 448–456, July 2015.
- [8] Tensorflow. “Simple audio recognition: Recognizing keywords”. Web URL: [https://www.tensorflow.org/tutorials/audio/simple\\_audio](https://www.tensorflow.org/tutorials/audio/simple_audio)
- [9] Ha şim Sak, Andrew Senior, and Françoise Beaufays. “Long short-term memory recurrent neural network architectures for large scale acoustic modeling”. Fifteenth Annual Conference of the International Speech Communication Association, 2014.
- [10] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernock`y, and Sanjeev Khudanpur. “Recurrent neural network based language model”. Interspeech, volume 2, page 3, 2010.
- [11] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. Neural computation, 9(8):1735–1780, 1997.
- [12] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning phrase representations using rnn encoder-decoder for statistical machine translation”. [arXiv:1406.1078v3](#). September 2014.
- [13] Sercan O Arik, Markus Kliegl, Rewon Child, Joel Hestness, Andrew Gibiansky, Chris Fougner, Ryan Prenger, and Adam Coates. “Convolutional recurrent neural networks for small-footprint keyword spotting”. [arXiv:1703.05390v3](#). July 2017.
- [14] TensorFlow. “TensorFlow Lite converter”. Web URL: <https://www.tensorflow.org/lite/convert/>
- [15] Tara N Sainath and Carolina Parada. “Convolutional neural networks for small-footprint keyword spotting”. Sixteenth Annual Conference of the International Speech Communication Association, 2015.
- [16] Rocha, Bruno Miguel Machado et al. “A Respiratory Sound Database for the Development of Automated Classification.” BHI 2017 (2017).
- [17] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, Tie Qiu. “Survey on fog computing: architecture, key technologies, applications and open issues”. Journal of Network and Computer Applications, Volume 98, Pages 27-42, Figure 1, 2017.
- [18] Sengupta, Nandini & Sahidullah, Md & Saha, Goutam. “Lung sound classification using cepstral-based statistical features”. Computers in Biology and Medicine, Volume 75, Pages 118-129, 2016.