Snoring Detection on a Microcontroller

Adriana Rotaru
Computer Science Department
Harvard University
Cambridge, MA, USA
adrianarotaru@college.harvard.edu

Jiayu Yao
Computer Science Department
Harvard University
Cambridge, MA, USA
jiy328@g.harvard.edu

Kelly Zhang
Computer Science Department
Harvard University
Cambridge, MA, USA
kellywzhang@seas.harvard.edu

***Abstract*—Snoring is related to a common medical condition that can lead to many serious health issues including diabetes, stroke, and depression. Due to the negative health impacts of such medical condition, it is crucial for people to know whether they snore and understand their snoring patterns and triggers of snoring. In this paper, we propose a bed-side snoring detection program on microcontroller device that automatically identifies snoring sounds. Our device extracts spectrograms from real time audio data and applies a Convolutional Neural Network (CNN) to classify whether an audio sample contains snoring sounds or not. In our work, we investigate different pre-processing methods, including Fast Fourier Transforms (FFT) and Mel-frequency cepstral coefficients (MFCC), as well as different neural network model architectures. We evaluate different approaches in terms of accuracy and model size. We deploy our models on a microcontroller with 1MB of Flash with minimal power consumption that can be run continuously. The best model we deploy has accuracy of 96.86%, which is comparable to that of existing snoring detection models. However, our model is of size 18,712 bytes which is over 500 times smaller than other models in the literature (e.g. 9.8MB of [7]).**

***Index Terms*—Snoring, MFCC, FFT, Arduino, Tensorflow LITE/Micro**

I. Introduction

A. Motivation and Background

A study conducted by the National Commission on Sleep Disorders Research found out that more than 40 million Americans suffer from various types of sleep disorders [12]. Snoring is one of the most common symptoms of Obstructive Sleep Apnea (OSA), which is a breathing sleep disorder that is found across different genders and ages. OSA disturbs sleep and often disrupts the balance of oxygen and carbon dioxide in the body. If OSA goes without treatment, it can lead to dangerous daytime drowsiness and even serious health conditions such as cardiovascular issues, diabetes, stroke, and depression [9]. Additionally, a patient's snoring pattern can be critical for informing their diagnosis and treatment, which makes detecting snoring patterns an important problem.

We are interested in developing a bed-side microcontroller device that can detect snoring sounds and monitor people's snoring patterns while sleeping. While one could use a device like a phone for snoring audio detection, to do so, the user would have to ensure that the phone is charged and the program is turned on before going to sleep, which can be inconvenient. Unlike many other snoring detection devices, with a microcontroller device placed by the bed, the user does not have to remember to turn on the device or to ensure it is charged or attached to their body, which can be inconvenient and uncomfortable [7]. Our microcontroller device is inexpensive and can be battery powered without requiring frequent charging or battery replacement because of low power consumption.

Moreover, using a microcontroller offers additional privacy protection since prediction is performed on device. The advantage of on device prediction is that the raw audio sounds will not be required to be sent to the phone, computer, or cloud, which increases the privacy risk of sending unprotected information to a device that could be hacked or misplaced. The microcontroller would only send the sequence of snoring predictions to another device, like a phone or computer, which can then be used to monitor the snoring or analyze the snoring pattern. With an accurate snoring prediction model, the user can check their snoring frequency and with additional analysis, the users can even learn if snoring is related to other behaviors (e.g. drinking alcohol could make snoring worse). Additionally, the microcontroller could be equipped with an external sensor that could prompt the person to change position when snoring is detected during sleeping.

B. Our Contribution

In this work, we describe the process of developing and deploying a snoring detection model onto a microcontroller. Our snoring detection model can run continuously on a microcontroller with high accuracy and small memory and power consumption. Compared to other work on audio snoring detection [7], our snoring detection model can be deployed on much smaller devices with high accuracy that is comparable to that of larger models. Finally, we also discuss the challenges associated with training and deploying tiny Machine Learning (tinyML) models. Our work can act as a case study for others trying to deploy machine learning models onto microcontrollers.

II. Related Work

In the previous literature, snoring detection models often utilize vibration information of the snoring patters and rely on mechanical methods[13]. In this project we will focus on snoring detection by extracting spectrum features from the raw audio and performing prediction task with deep learning models.

Fast Fourrier Transform (FFT) is a frequently used technique in audio processing for feature extraction. In addition to FFT, more complicated preprocessing methods like Mel frequency cepstral (MFFC) from Khan or deep spectrum feature extraction from Amiriparian et al. are commonly used in signal processing. The advantage of FFT is faster on device execution and greater compatibility with pre-processing methods developed for different devices. Previous works have found out that the optimal frequencies needed for snoring detection differ from those optimal for speech detection. For example, Agrawal et al. found that median frequency for palatal (velum) snoring is 137 Hz and the median frequency for tongue based snoring is located at 1,243 Hz. Additionally, Qian et al. performed snoring sound classification by fusing different acoustic features and found that a combination of upper and lower level frequencies to be amongst the best performing. Given that previous works use MFCC or spectrum-based methods that amplify certain frequencies over others, in our work we experiment with multiple types of preprocessing methods. For example, a simple FFT allows extraction of features that reflect a wider variety of snoring frequencies in the dataset and minimizes latency due to pre-processing compared to MFCC.

Existing snoring detection works have been primarily based on deep neural networks, including CNNs and Recurrent Nerural Networks (RNNs). Although these models have high accuracies of 91-98.40 %, as shown in Table I, many do not report model sizes and the ones that do are too large to fit on many microcontroller devices.

TABLE I: Comparison with other work

Author	**Model Architecture**	**Accuracy**
R.Nonaka, et. al [8]	Logistic Regression with auditory features	97.30%
E.Dafna, et. al. [4]	AdaBoost classifier with spectral features	98.40%
H.Romero, et. al.[11]	Deep learning with bottleneck features	91.11%
T.Emoto, et. al [5]	Deep NN	75.10 %
S.Amiriparian [2]	Convolutional NN	92.50%
Arsenali, et. al [3]	Recurrent Neural Network with MFCC	95.00%
Khan [7]	CNN with MFCC	96.00%
Proposed	Tiny Convolutional NN	96.86%

(Note that the accuracies reported here are for various datasets. In our work, we compute accuracies on the dataset used by Khan [7].)

In our work, we experiment with different model architectures and compare them in terms of size, accuracy, and on-device performance. Our baseline models are based on those from the MicroSpeech Keyword Spotting Project of TensorflowLITE Micro library, which are modified to adapt to the binary classification of snoring and no-snoring. Comparing these models with another baseline CNN introduced by Khan, our best deployed model has similar accuracy and is more than 500 times smaller in size (9.8 MB vs. 18 KB).

III. Methods

A. Dataset and Pre-processing

Collecting snoring audio data can be quite difficult, since deep neural network models require many training examples. Moreover, to train a model that is applicable to many different people, we require snoring data from people with a wide range of snoring conditions, age, gender, vocal features, etc. In this work we use the dataset constructed by [7], which is publically available here on Kaggle. The dataset includes data from people of various ages and genders. The data set also includes non-snoring sounds that are representative of disturbances in a sleeping environments, such as baby cries, wind sounds, TV noises, etc. The data samples are 1000 one-second sound clips, with 500 samples for each of the 2 classes: snoring versus non-snoring. Among the 500 snoring samples, 363 samples consist of snoring sounds of children, adult men and adult women without any background sound. The remaining 137 samples consist of snoring clips having a background of non-snoring sounds [7]. In order to improve model performance, we augmented the training dataset with additional samples of silence, background noise, and human speech [14].

In order to make accurate inferences from the data, appropriate pre-processing methods have to be applied for feature extraction. We selected the pre-processing methods to be consistent with the variability in the audio data, the model's data format requirements, and the compatibility of the libraries with the device, as well as to minimize the pre-processing time on device. For the pre-processing, the clips, which were initially at a sample rate of 48 kHz, were converted to 16kHz using the PyDub Audiosegment library, in order to minimize the size of the inputs (spectrograms) to the model without sacrificing on audio quality. To convert the clips to spectrograms, the audio files were traversed by 30ms windows with 20ms jumps (strides), as shown in Figure [1] below [15]. Each of the 49 frames are mapped to a row in the spectrogram, so there are 49 rows in the final spectrogram. FFT with a bin count of 512 was applied to each of the 30ms frames, resulting into an array of 256 values. The values were averaged in groups of 6, leading to 43 values, which were then reduced to 40. The resulting spectrogram is of size 49x40.

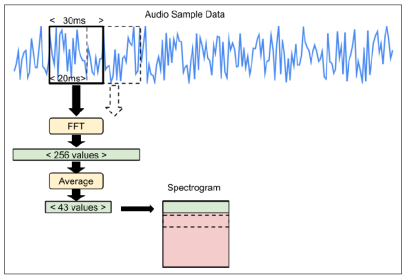
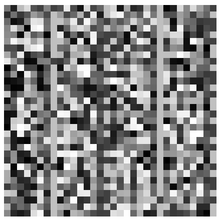
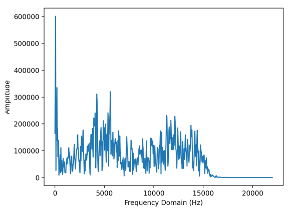


Fig. 1: **Audio Processing Example**. *Source*: *"TinyML" by Pete Warden and Daniel Situnayake [15]*

In addition to experimenting with FFT preprocessing, we also tried using a customized MFCC algorithm for preprocessing, which was adapted from Khan et. al [7]. For this method, the audio signals were sliced into 32 frames of 30ms each and the power spectrum, as shown in Figure [2b], was computed using the SpeechPy library. The MFCC had 10 filterbanks placed uniformly at 100Hz intervals from 100Hz to 1000Hz, 22 filterbanks placed uniformly for frequency above 1000Hz. This resulted in a 2D 32x32 (n_filters x n_frames) array of features that was converted to a gray image as in Figure [2a].


(a) Spectrogram



(b) Power Spectrum

B. Hardware and Sensors

We deploy our snoring detection model onto a **Arduino Nano 33 BLE nRF52840** board, which is a 32-bit ARM® Cortex™-M4 CPU running at 64 MHz. The device has 1MB of CPU Flash Memory and allows deployment for models as large as 100KB after quantization.

C. Software Framework

The speech commands example of the TensorFlowLITE library was adapted for training and the micro speech deployment framework, from the Wake Words assignment, was modified for deployment.

For training, four classification labels were used, the same as in the micro_speech example, which includes a silence and an unknown category. We used more than two classes during the training because merging data with dissimilar patterns (e.g. silence versus no-snoring) affects the model performance. The *silent_percentage* and *unknown_percentage_frequency* were set to 25% to make the four classes balanced, the *background_frequency* was set to 0.8 and *background_volume_frequency* to 0.1 (see details in train_snoring_model.ipynb). We set the pre-processing flag to *micro* and experimented with all the baseline models under the *model_architecture* flag. As written in the speech_commands framework, quantization was used to reduce the model sizes.

The deployment framework was modified based changes we made to the micro_speech experimental framework. For example, we modified the model architectures and model arena sizes, and also tuned deployment prediction sensitivity parameters, like the prediction threshold and the voting mechanism to predict. For deployment, the *unknown* and *silence* classes were mapped to *no_snoring* because we only want to distinguish snoring from all other classes, as done in previous work [7]. (See all modifications made to the code under all_summary.md).

D. Model Architecture

Given the strict memory constraints of the microcontroller we wanted to deploy on, we experimented with many different model architectures and compared models in terms of their sizes and test prediction accuracy. All the models were trained using the *momentum* optimizer and with a *cross-entropy* loss function. Given the audio classification task and the spectrograms input formats, most of the models we tested were CNN-based. The primary work we compared to was the Khan CNN model of size 9.8 MB and accuracy of 96.00%. Since the model of Khan is too large to fit on the microcontroller, we were interested in developing a model with similar accuracy, that was much smaller. We tested the following baseline models proposed in the open source speech_commands code:

- **low_latency:** a depthwise convolutional 2D layer, three (128, 128, 4) fully connected layers.
- **tiny_conv:** a depthwise convolutional 2D layer, a 4-units fully connected layer
- **single_fc:** a single 4-units fully connected layer
- **tiny_embedding_conv:** a depthwise convolutional 2D layer, a convolutional 2D layer, a 4-units fully connected layer
- **low_latency_svdf:** a singular value decomposition filters layer
- **conv:** a depthwise convolutional 2D layer, a Maxpool 2D, a 4-units fully connected layer

Additionally, we proposed two additional models, which were modifications of the baseline models, but aimed to better trade-off accuracy and model size. The following custom models were proposed:

- **tiny_conv(256,128):** a depthwise convolutional 2D layer, three (256,128,4) fully connected layers.
- **tiny_conv(128, 128):** a depthwise convolutional 2D layer, three (128, 128, 4) fully connected layers

For a detailed description of all the different model architectures, see the models section.

E. Deployment Framework

We modified the default deployment code of micro_speech to reflect our specific classification task and to adapt our detection intervals to reflect that of the periodicity of typical snoring patterns. We then deploy the model based on the micro-speech example from the Tensorflow Micro library, which contains the FFT preprocessing and the inference code. The deployment code is modified to the binary classification task of differentiating between snoring vs. non-snoring audio clips.

In snoring audio detection, the detection intervals are critical because people tend to snore with specific time patterns. Snoring often occurs cyclically with four second intervals [6]. In addition, some snoring patterns can be irregular, lasting longer than others. To reflect these real-life situations, we have tuned the parameters for the recognition window in the deployment code. For instance, reducing the *average_window_duration_ms* increases the sensitivity of the prediction and addresses the issue of frequent misses due to irregular snoring patterns. Additionally, reducing the *detection_threshold* and the *minimum_count* both increase the sensitivity of the model and increase the false positive rate while decreasing the false negative rate. Finally, tuning the *suppression_ms* parameter, which is the time for which further recognitions are suppressed, to match the separation between the snores at 4 seconds would be reflective of real-life snoring patterns.

IV. Results

A. Models comparison

The first aspect we experimented with was the pre-processing method. Specifically we compared a customized MFCC pre-processing method with a simple FFT pre-processing (as shown in preprocessing.py script). We trained the same CNN model on the dataset with each of the different pre-processing methods. We found that the model trained on data pre-processed with FFT had 5-6% lower accuracy compared to that trained on the data pre-processed with the MFCC method. Additionally, we experimented with the pre-processing methods developed in TensorFlow AudioOps. Specifically we tried a baseline CNN models with the *–micro* (which sets the preprocessing to FFT) and *–mfcc* pre-processing methods. In these experiments, the float point model with MFCC performed better than that with FFT (100% vs 96.86%). However, the quantized model with MFCC had unusually low accuracy at 68%. We conjecture that the input pre-processed with MFCC suffers from accuracy loss during INT8 quantization. From Figure 3, we see that the input pre-processed with MFCC is more concentrated, which indicates more bits are required for lossless decoding.

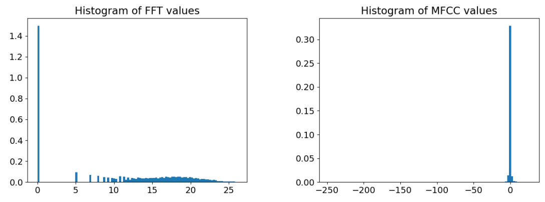


Fig. 3: Histogram of input after different pre-processing. Input pre-processed with MFCC is highly concentrated around 0 while input pre-processed with FFT is more spread out.

Another disadvantage of MFCC is that it takes much longer to run, which can affect the latency of the model. Latency is important in snoring detection because it affects how frequently the model predicts—a high latency model can only predict at longer intervals—which can be sub-optimal if snoring patterns occur at a different rate. Additionally, low latency would be important if the snoring detection model was connected to device that could prompt the user to change sleeping positions when snoring is detected. We also had many challenges integrating the MFCC pre-processing code into the deployment code because the default deployment code used FFT and adjusting the pre-processing would require significant coding in C++. For these reasons, we focused on using FFT pre-processing for all the models we deployed.

Each model was trained with *silence* and *unknown* categories in addition to the snoring and no_snoring categories, as a way to augment data with background noise and silence. So to be able to make our models comparable to those proposed in [7], we mapped 4 categories into 2 categories (silence and unknown classes are mapped to no_snoring) when computing the testing accuracies using the results in the confusion matrix. The accuracies of all the models that were tested are shown below.

TABLE II: Summary of baseline models performance

Model	**Test Accuracy (with mapping to 2 classes)**	**Size (bytes)**
low_latency	98.11%	884,360
tiny_conv	**96.86%**	**18,712**
tiny_conv with MFCC	100.00%	18,712
single_fc	**89.93%**	**8,872**
tiny_embedding_conv	89.93%	8,864
low_latency_svdf	98.11%	37,449
conv	97.48%	308,168
tiny_conv (256,128)	**98.11%**	**1,062,816**
tiny_conv (128,64)	**98.11%**	**525,216**

The best performing baseline models in testing that could be deployed on Arduino were the tiny convolutional (tiny_conv) and the single fully connected (single_fc) ones with accuracies of 96.86% and 89.93% respectively. The convolutional model encountered a memory corruption issue (see the On Device Comparison section for more details) and the low latency models was too large to fit on the microcontroller. The low latency singular value decomposition model could not be deployed because TensorFlowLITE has no support for SVDF.

Given the accuracy, size tradeoffs and compatibility limitations, the single_fc and tiny_conv models were deployed for comparison. Since tiny_conv had a great performance both offline and on device, we explored variations of it found in the tiny_conv_custom folder. The proposed models tiny_conv (256,128) and tiny_conv (128,64) were modifications of the tiny_conv model, obtained by adding two more fully connected (FC) layers to it. The reason for adding the FC layers was our previous observation of unusual high performance of the single fully connected layer model [II] on a fairly complex 2D dataset. The models with the highest in accuracy of 98.11% were also much larger than the baseline, with sizes 525kB and 1000kB respectively. Looking at these 2 models with (256,

128) and (128, 64) fully connected layers, it turns out that doubling the number of neurons in the fully connected layer makes the size of the model explode and doesn't significantly change the accuracy. One takeaway from this result is that for snoring detection, at least on this dataset, when the test accuracy reaches at least 95%, increasing the model's accuracy requires a disproportionately large increase in the model size.

One analysis we performed to understand the relationship between model size and model accuracy was to start with the baseline tiny_conv model and add an additional fully connected layer of various sizes. How these variations in the final fully connected layer size affect accuracies and model size are displayed in the following figure:

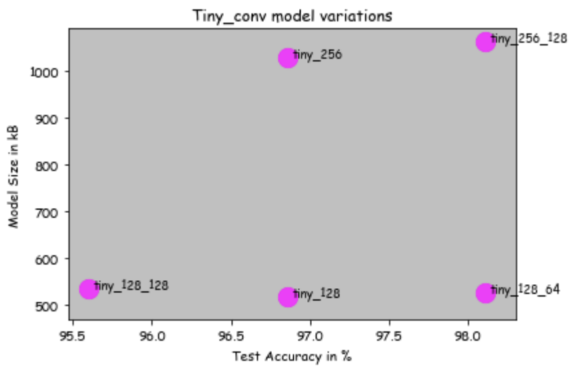


Fig. 4: Performance and model size tradeoff of variations of the tiny_conv model. *The numbered labels correspond to the number of neurons in the fully connected layers that were added to the existing baseline tiny_conv. For example, tiny_128_64 corresponds to 2 fully connected layers of 128 and 64 added to the baseline model.*

B. On Device Comparison

The top performing models (tiny_conv (128,64), low_latency, conv) threw a memory corruption error when deployed. The same type of error arose when deploying the 2 custom models, despite maximizing the use of the arena_size, which should allow the model to fit on Flash (we were able to deploy similar sized models). We found this memory corruption error was commonly encountered by others when deploying a newly trained model with the same speech commands framework. Based on the github issue raised recently, this problem is currently an unsolved TensorFlow problem: https://github.com/tensorflow/tensorflow/issues/39938. Due to these challenges, we primarily tested the tiny_conv baseline in deployment. Additionally, the lack of support of SVDFs on TensorFlowLITE made it impossible to test the low_latency_svdf model, a model with the good size-accuracy balance.

We found that the sensitivity parameters used for prediction in deployment dramatically impacted the real world performance of our model. Under the default sensitivity deployment parameters, we found that our tiny convolutional model never detected snoring (no false negatives). However, after adjusting the sensitivity parameters we were able to get this model to perform well in practice, achieving a minimal number of false negatives and almost no false negatives. Specifically, prediction deployment algorithm uses a voting mechanism for predictions made in some past window of time; we adjusted the detection threshold as well as the number of votes needed within the time window for the code to trigger a classification. For more details, see our deployment demo.

V. Discussion

A. Takeaways

In this study, we found out that the development and the deployment in tinyML is non-trivial. Given that there was an open source method for wake word detection in tensorflow lite, we thought it would be relatively straightforward to adapt the setup for snoring detection. However, we ran into many unexpected challenges. For example, we discovered that small changes in the pre-processing procedure would require significant changes in raw C++ code used for the pre-processing procedure on device. Time-wise, we were not able to modify the original FFT pre-processing to the MFCC pre-processing that is used in other snoring detection work.

Moreover, we found that the best performing models off device were generally also the best performing models on device—however, all models had slightly lower accuracy in deployment (empirically) than when testing offline on the computer. Some reasons between offline and deployment accuracy is that there may be small differences between the training and real world testing data distributions. For example, the distance between the device and the source of the sound also affects the on device performance significantly. We also found that the deployment performance was significantly impacted by the hyperparameters for prediction including prediction threshold, the amount of time between predictions, the window of past predictions used for voting, etc. Based on our experience, we advocate for more thorough and systematized real world testing in order to truly make fair comparisons on the task we care about.

Finally, we want to point out that this project can serve as a case study for developers seeking to integrate data processing, modeling and deployment architectures into a real life TinyML application. The issues that can arise due to incompatibility or unsupported pre-processing methods can lead to sacrificing performance. We were not able to deploy many of our top performing models due to memory corruption issues encountered and a lack of support for MFCC pre-processing on device, and as a result our final deployed model had much lower accuracy than could be deployed on such a device. In summary, while it was easy to train many models in the TensorFlowLite framework, we encountered many problems regarding the reliability and flexibility of the open source deployment code that inhibited us from deploying many of the models we trained. Therefore it is essential for people to better develop more flexible and reliable tools to help developers throughout the entire training to deploying pipeline in order to advance the field of TinyML.

B. Future Directions

In terms of future work, some more immediate next steps would be to deploy the MFCC pre-processing step on device and to collect additional snoring sound data for training. Khan reported 53ms for the MFCC pre-processing step on the RasperryPi device [7]. If MFCC can be integrated into our pipeline, it will take longer on the Arduino board, which has to be addressed in the implementation. Additionally, in future work, we could develop a more systematic method for deploying a multitude of models for testing, with an automatic parametrization, instead of manually tweaking the parameters, as well as for testing models on device. In terms of improving model performance, while additional effort could be put in developing a better deep neural network architecture, we believe that the most gains in performance will be obtained from changing the pre-processing and collecting more data that is specific to the use case (e.g. specific environments and located at reasonable distance from bed as in a real use case).

Regarding more long term work, this snoring model could be personalized by having the model trained on device with a small subset of data which includes the user's own snoring/non-snoring audio data. This could be achieved by using transfer learning with warm start neural network training. Additionally the application of the snoring detection model could be expanded towards prevention. A sensor could be integrated with the device (ex: arm cuff that vibrates), which would wake up the user to prompt them to change their sleeping position when they snore. For such a use case, it would be crucial to ensure the false positive rate to be be low because frequently disrupting the users' sleep may cause them to quit the application. Finally, logs of snoring data could also be combined with additional health data collected through other sensors (e.g. heart rate, step count, ect.) to help users learn if there are any other triggers to snoring.

C. Ethical Considerations

Overall, we believe that our application sufficiently protects privacy. Our application maintains constraints on the flow of information because all predictions are made on a microcontroller device, meaning all audio data is discarded after prediction. Our application reduces the attributes of the data by only sending the sequence of snoring and non-snoring predictions to a device connected to the internet, like a phone or computer, which is designated by the user.

Regarding context-relative norms, the context of snoring and bedroom data is that no one besides those present in the room or house, can hear the sounds in the room. Our application (as it currently stands) maintains the autonomy of the user, by default, because the only people who have access to the predictions from the microcontroller are the people who have access to the room and have linked their phone or computer to the microcontroller device. Since the user has full autonomy over sharing and holding the flow of information, there are no ethical issues related to autonomy infringement or risks with informational leaking that arises from the use of the device. The user has full control over the flow of snoring information and is free to choose to disclose it to their doctor. The user is also not dependent on this tool, which excludes concerns related to power relations imbalance. There are many alternatives on the market for snoring detection, which makes this proposed tool an additional support mechanism, rather than an indispensable one for people with snoring conditions.

References

[1] S Agrawal et al. "Sound frequency analysis and the site of snoring in natural and induced sleep." In: *Clin Otolaryngol Allied Sci* 27.3 (2002), pp. 162–6.

[2] Shahin Amiriparian et al. "Snore Sound Classification Using Image-Based Deep Spectrum Features." In: *INTERSPEECH*. Vol. 434. 2017, pp. 3512–3516.

[3] B. Arsenali et al. "Deep Learning Features for Robust Detection of Acoustic Events in Sleep-disordered Breathing." In: *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Honolulu, HI, USA* (2018), pp. 328–331.

[4] Dafna E., Tarasiuk A., and Zigel Y. "Automatic Detection of Whole Night Snoring Events Using Non-Contact Microphone." In: *PLoS* 8.e84139 (2013).

[5] T. Emoto et al. "Detection of sleep breathing sound based on artificial neural network analysis." In: *Biomed.Signal Process* 41 (2018), pp. 81–89.

[6] Mesquita J. "All night analysis of time interval between snores in subjects with sleep apnea hypopnea syndrome." In: *Medical and biological engineering and computing* (2012), p. 1.

[7] Tareq Khan. "A Deep Learning Model for Snoring Detection and Vibration Notification Using a Smart Wearable Gadget". In: *Electronics* 8.9 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8090987. URL: https://www.mdpi.com/2079-9292/8/9/987.

[8] R. Nonaka et al. "Automatic snore sound extraction from sleep sound recordings via auditory image modeling." In: *Biomed.Signal Process* 27 (2016), pp. 7–14.

[9] *Obstructive Sleep Apnea - Causes and Symptoms*. Oct. 2020. URL: https://www.sleepfoundation.org/sleep-apnea/obstructive-sleep-apnea.

[10] K. Qian et al. "Classification of the Excitation Location of Snore Sounds in the Upper Airway by Acoustic Multi-Feature Analysis." In: *IEEE Transactions on Biomedical Engineering* (2016).

[11] H.E. Romero et al. "Deep Learning Features for Robust Detection of Acoustic Events in Sleep-disordered Breathing." In: *ICASSP 2019—2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, UK* (2019), pp. 810–814.

[12] T Roth. “An overview of the report of the national commission on sleep disorders research”. In: *European Psychiatry* 10 (1995). Satellite Symposium to the AEP Congress, 109s–113s. ISSN: 0924-9338. DOI: https://doi.org/10.1016/0924-9338(96)80091-5. URL: http://www.sciencedirect.com/science/article/pii/0924933896800915.

[13] Hangsik Shin and Jaegeol Cho. “Unconstrained snoring detection using a smartphone during ordinary sleep”. In: *Biomedical engineering online* 13.1 (2014), p. 116.

[14] Pete Warden. “Speech commands: A dataset for limited-vocabulary speech recognition”. In: *arXiv preprint arXiv:1804.03209* (2018).

[15] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*. O’Reilly Media Inc., 2020.

VI. Appendices

A. System and Code Description

A clear description of how to use your system and how to generate the output you discussed in the write-up.The teaching staff must be able to run your system. Use references from the Supplementary folder layout

Data

The Snoring Dataset can be downloaded from here. The *_background_noise_* samples were added from the speech_commands dataset.

Data Pre-processing

All the separate MFCC and FFT pre-processing algorithms files are in the preprocessing folder.

1) You can apply MFCC or FFT on the dataset by giving the right names to the *save_dir* and the *dir* variables and passing the right argument for the preprocessing method to the main function.
2) To downgrade the sample rate of the audio samples: *python3 downgrade_sample_rate.py*, making sure that the dataset *dir* is set and that the folder layout in the dataset is the following:

```
Snoring_Dataset
├─ 0 (no-snoring wavs)
│
├─ 1 (snoring wavs)
│
└─ _background_noise_
```

Model Training

1) Training baseline models
   Run the train_snoring_model.ipynb in Colab or Jupyter, setting the directory name of the dataset, *PREPROCESS* to micro, and the model flag *MODEL_ARCHITECTURE* to *tiny_conv*, *single_fc*, *low_latency*, etc. Make sure that you have access to tensorflow (the train.py, freeze.py in speech_commands are unchanged).

2) Training the additional models
   - the CNN from proposed Khan can be trained on the MFCC pre-processed data (see the data processing instructions above). The backbone of the model is found in the models/snoring-CNN folder.
   - To compare the performance of the CNN on MFCC data with the FFT data, run the *baselines.ipynb* after running the *preprocessing.py* script which will generate the preprocessed dataset. These files are found in the CNN model folder

3) To train the customized tiny_conv models, refer to the tiny_conv_custom folder. Replace the *models.py* script in the downloaded tensorflow/../examples/speech_commands folder with the script in the tiny_conv_custom folder, which includes the function for creating the custom models. The summary.md in the tiny_conv_custom folder includes all the results from training the models, as well as the .cc and .tflite models.

Model Deployment

Deployment video here.
Microspeech code modifications:

1) *recognize_commands.h*
   Change the sensitivity parameters in the RecognizeCommands() object as follows
   - tflite::ErrorReporter* error_reporter,
   - int32_t average_window_duration_ms = 800,
   - uint8_t detection_threshold = 200,
   - int32_t suppression_ms = 1500,
   - int32_t minimum_count = 2
2) *micro_speeech.ino*
   For models above 18kB size set *constexpr int kTensorArenaSize = 100 * 1024;*

3) *micro_features_model.cpp*
   Add the model.cc and the size

4) *micro_features_micro_model_settings.h*
   Set *constexpr int kCategoryCount = 4;* . Keep the unknown index = 1, silence index = 0

5) *micro_features_micro_model_settings.cpp*
   Set *const char* kCategoryLabels[kCategoryCount] = { "silence", "no_snoring", "snoring", "no_snoring", };*

6) *arduino_command_responder.cpp*
   digitalWrite(LEDG, LOW); // Green for snoring
   digitalWrite(LEDR, LOW); // Red for no snoring

B. Division of Work

A list of each project participant and their contributions to the project. If this varies significantly from the project proposal, provide a brief explanation.

1) Kelly Zhang:

- Writing: introduction section, discussion section, results deploying on device, editing
- Presentation: I helped edit the presentation script
- Deployment: I was the primary person in charge of deployment. We initially started with a two class prediction framework (modifying the original wake words code), which I created. I filmed the deployment video and experimented with the sensitivity parameters in deployment.

2) Jiayu Yao:

- Writing: Model comparison, proofread, editing
- Models: Pre-processing; implemented some baseline models and all custimized models; investigated MFCC versus FFT pre-processing difference.
- 5min Presentation

3) Adriana Rotaru: Focused on data pre-processing and modeling and deployment parameters tuning.

a) Writing: related work, references, methods, discussion sections; the supplementary and system and code description. Computed all the diagrams.

b) Data Preprocessing:

- Implemented the customized MFCC pre-processing in the preprocessing folder.
- Implemented a customized FFT algorithm separate from the one in the speech commands, to compare the performance of the models with FFT preprocessing versus that of models with MFCC.
- Downgraded the sample rate of the audio files to match it to the sample rate expected by the deployment code, while preserving the quality of the audio signal.

c) Modeling:

- Trained, tested and compared the 6 baseline models.
- Build the CNN proposed by Khan [7], as shown in Figure [7].

d) Deployment:

- Deployed the tiny_conv and the customized tiny_conv models.
- Set the deployment framework with all the appropriate modifications.
- Tuned the sensitivity parameters for recognition on device for the tiny_conv model.
- Investigated the memory corruption error in deployment, by coordinating with Pete Warden.

Since I trained the baseline models, I had a better grasp of the model features, so I took on part the deployment, modifying the framework to adapt it to the snoring classification. After discussing with Pete on the deployment issue with predictions being made too fast and sporadically, I tuned the sensitivity parameters solving the issue.

VII. Supplementary

A. Folder Layout

- a summary about each folder in the submission

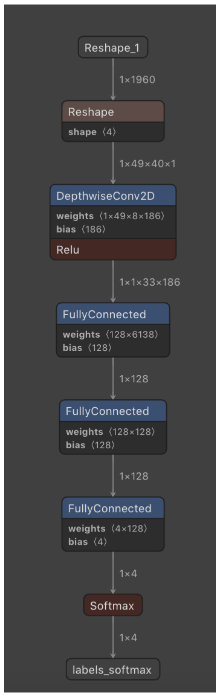
```
project
├─ baseline_models
│  ├─ all_summary.md
│  ├─ tiny_conv**
│  ├─ low_latency
│  ├─ low_latency_svdf
│  ├─ tiny_embedding_conv
│  ├─ single_fc**
│  └─ snoring-CNN
├─ train_snoring_model.ipynb
├─ CNN model
│  ├─ .ipynb training script
│  ├─ neutron models
│  ├─ processed_data
│  └─ utils.py
├─ preprocessing
│  ├─ preprocessing.py
│  └─ downgrade_sample_rate.py
└─ tiny_conv_custom
   ├─ summary.md
   ├─ .tflite and .cc models
   └─ models.py
```

** models that were deployed

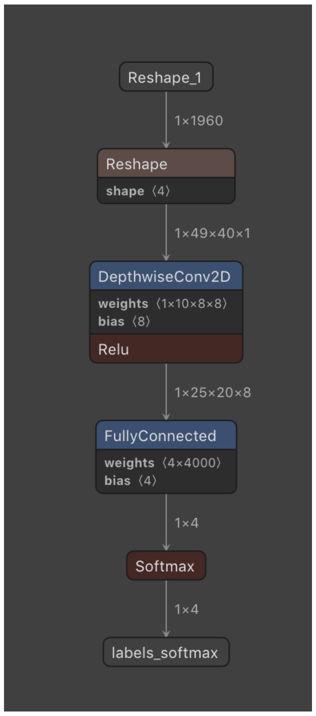
B. Diagrams and Code snippets

CNN model

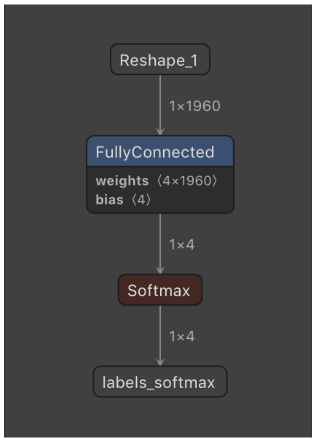
Baseline Models



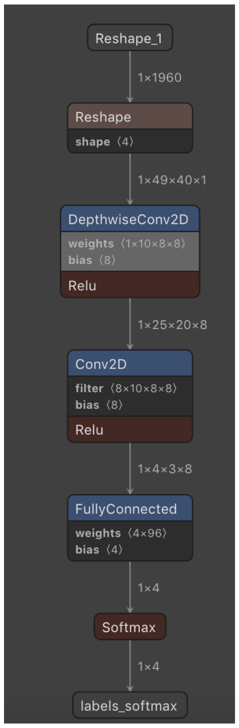
(a) low_latency



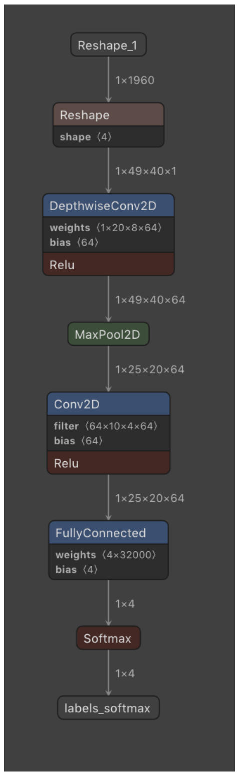
(b) tiny_conv



(c) single_fc

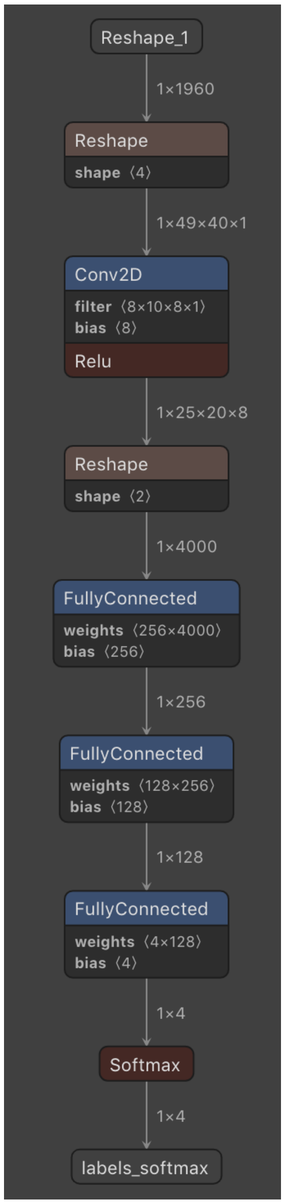


(d) tiny_embedding

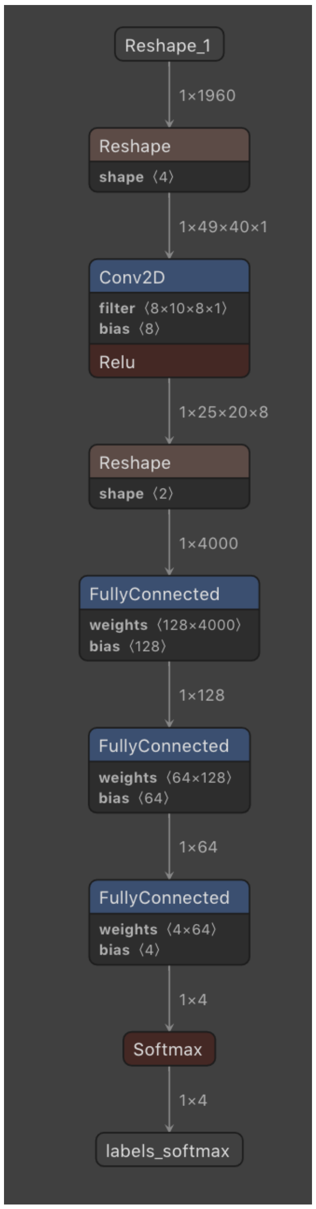


(e) conv

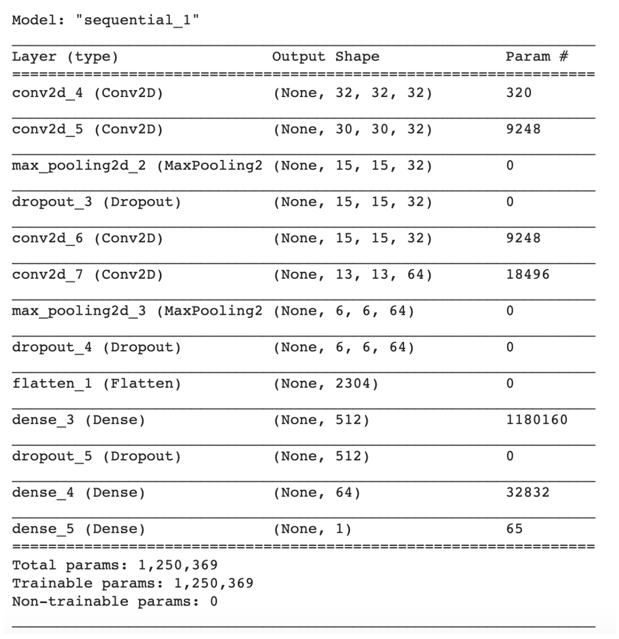
Proposed Models



(a) tiny_conv_256_128



(b) tiny_conv_128_64



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 32, 32, 32)        320
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 15, 15, 32)        9248
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_4 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                32832
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,250,369
Trainable params: 1,250,369
Non-trainable params: 0
_________________________________________________________________
```

Fig. 7: **CNN model of the Khan snoring paper** [7].