IMPERIAL COLLEGE LONDON
DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Embedded Machine Learning Ecosystem

Authors' Names:	***Authors' Emails:***
Andreas Floros	af2318@ic.ac.uk
Bharat Kumar	bkc17@ic.ac.uk
Hussain Kurabadwala	hk2018@ic.ac.uk
Manginas Vasileios	vm3218@ic.ac.uk
Stacey Wu	zw12618@ic.ac.uk

Contents

1. Introduction and Motivation

Over the past few years, there has been an immense rise in Machine Learning popularity, and we are seeing more and more applications every day. As environmental consciousness comes into focus, low-cost & energy-efficient ML devices are set to power the next generation, creating a high demand for ML models on embedded devices. It is projected that by 2030 a $1 trillion market will emerge for embedded ML devices [1].

As more and more industries, like manufacturing, research and development, and finance move towards automation, the growth of the embedded ML industry is imminent. Tiny ML in particular has seen many exciting breakthroughs in recent years, with more and more capable and intelligent hardware fueling large strides in this field. This in turn has lead to more on-device analytics and inferences. Hence, embedded devices will be able to interact with the cloud, further increasing the current scope of the industry [2].

Despite its aforementioned rise in popularity, incorporating ML is still hard for beginners. Audio applications are no exception, where non-experts face a range of technical challenges. This may include finding or making datasets, implementing the right audio pre-processing methods, creating and training ML models, converting models to a format that can be read by their microcontroller, and finally setting up an Arduino environment to run the trained model on specific hardware. Currently, no such platform exists that is able to combine all the above within one application. Therefore, we have the challenge to make this a reality.

2. Client Specifications

As per the specifications given by our client, ARM, the main task of this project was to build a framework upon which it would be much easier to create and deploy audio classification models on a microcontroller. The final aim is therefore to make embedded machine learning for audio classification more accessible, especially to people who might not necessarily have the underlying technical knowledge to face and resolve the many issues that arise in creating such a system. To that end, ARM provided us with the following specifications:

Baseline project:

- Build an audio classification model and deploy it to an Arduino to run inference.

Stretch goals:

- Build User Interface
- Select different datasets / create new dataset
- Implement different audio processing methods
- Configure different models / implement auto-ML for finding better models
- Directly interface with Arduino
- Automate the pipeline to the largest degree possible

The microcontroller required by the client for deployment and inference was the Arduino Nano 33 BLE Sense [3]. It features embedded sensors, such as a PDM microphone, and a powerful CPU running at 64 MHz, making this board suitable for running Machine Learning models. However, the device has a small program memory, with 1MB of Flash Memory and 256KB SRAM.

3. High Level Overview

Based on the client requirements, our product would be aimed towards non-experts who want to try building their own audio classifiers. Therefore, we started by creating a high level overview of the components that should be part of our solution.

The platform is constructed on two pipelines, one for each of the devices used, the device training the model (which from now on will be referred to as the Python-side), and the device running inference (which from now on will be referred to as the Arduino-side). Fig.1 depicts a flowchart with the tasks present on both of these pipelines.

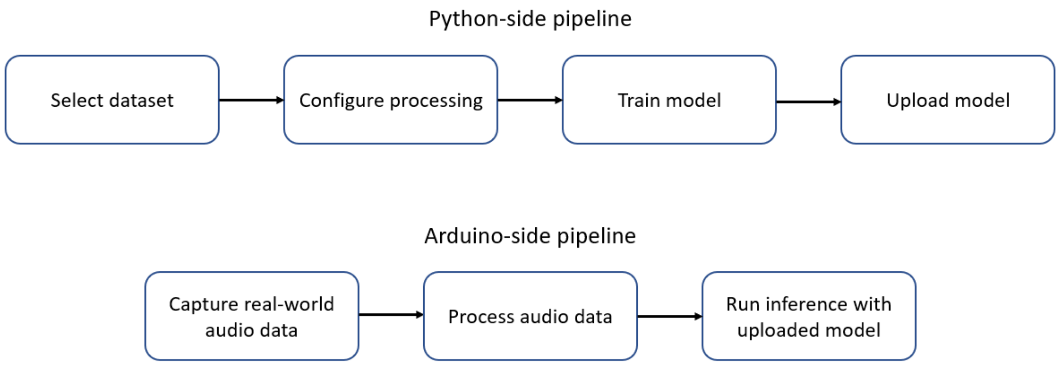


Figure 1. Python- and Arduino-side pipelines

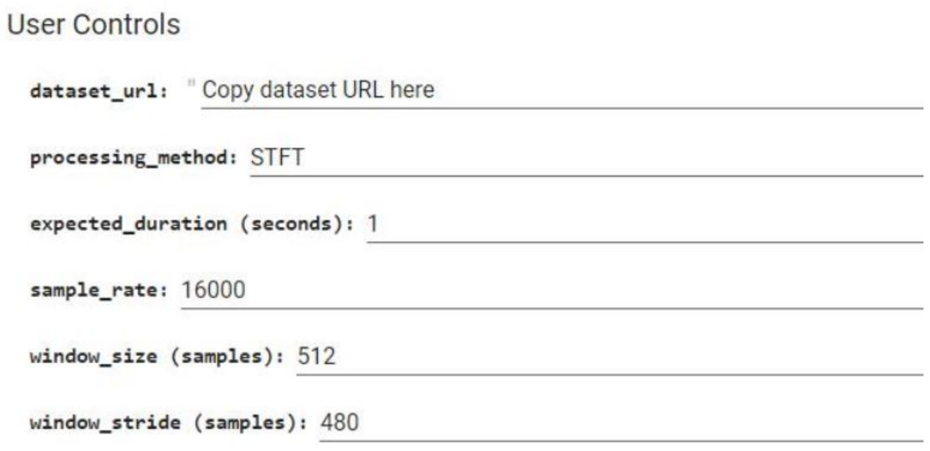
The Python-side consists of four main sections: Dataset selection, data processing for the chosen dataset, model training and conversion, and model deployment. Our product aims to guide the user through all of these steps. The Arduino-side is tasked with capturing real-world audio through the on-board microphone, processing that data in the exact same way as the Python-side, and finally running inference on it using the uploaded model, all in real time.

4. Concept Selection and Timeline

Given the high level overview, two main options were considered for the implementation of the baseline project. The first was utilizing cloud services, such as Google Colab, for hosting the entire Python-side pipeline, whereas the second option was to run all scripts locally.

Initially, all Python scripts were written in Google Colab in order to make use of the processing power of the GPUs and minimise runtime. We chose TensorFlow as the framework for the Machine Learning models due to existing infrastructure for running inference on embedded devices, including the chosen Arduino board. Fig.2 shows an early version of a front-end created for Colab to make it easier to edit the values within the code.

Although Colab provided a lot of benefits, there were several disadvantages to it. One of the main problems was that Colab is unable to access the local directories of the user's device. Consequently, it wasn't possible to interface to the Arduino board. Any generated models or scripts would need to be manually downloaded and then uploaded through the Arduino IDE, defeating one of the main goals of the platform, to automate the pipeline to the largest degree possible. Furthermore, the only available dataset source was through a web address downloading it into google drive, which Colab had access to. Thus, the dataset would have to be downloaded every time if the user did not want to store it on their Google Drive permanently. The user interface also had very limited options.


User Controls

dataset_url: " Copy dataset URL here

processing_method: STFT

expected_duration (seconds): 1

sample_rate: 16000

window_size (samples): 512

window_stride (samples): 480

Figure 2. Initial User Interface made on Colab

For these reasons, we decided to move away from cloud services, and instead run everything locally. Although this meant not utilizing cloud GPUs, we would still be able to run the entire pipeline on our local CPU [1]. While this increased runtime, this was not drastic since the machine learning models under consideration were relatively small. Additionally, we decided to construct a User Interface (UI) as the front-end of the Python-side pipeline. This provided the benefit of giving the users access to more features, such as dataset selection, different processing methods, and control over model-related parameters. Finally, it also allowed access to local files, solving several of the problems mentioned above. Combining all this led to the platform being a more complete product. The next section will look at the development of this solution in more technical detail.

[1]The users may also be able to run model training on their local GPU. This however, requires a setup so that TensorFlow is able to utilize the hardware.

5. Concept Development

5.1. Python-side Pipeline

Having decided on the concept design, we proceed to describe the main features of our user interface. Each of the four blocks of the pipeline, as mentioned in section 3, will be examined in detail in the following sections.

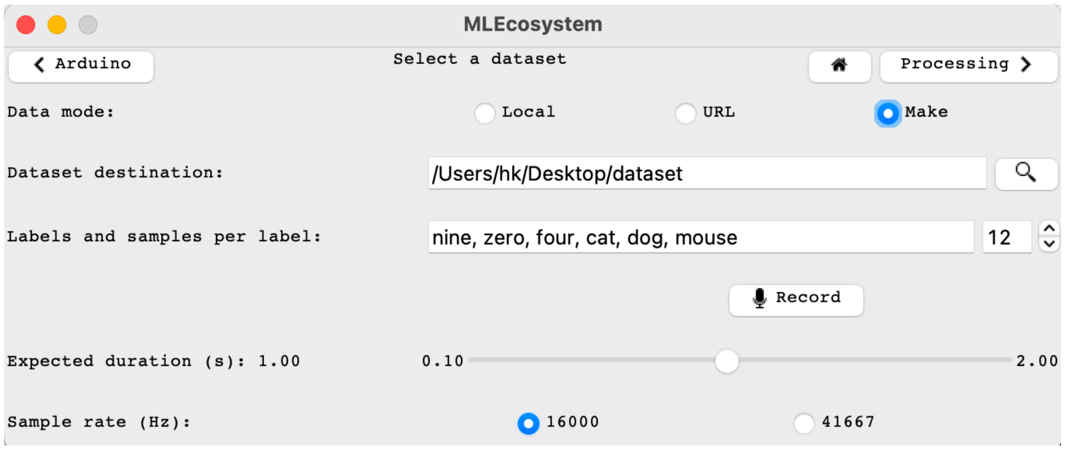
5.1.1 Dataset Selection

We provide three options for the source of the audio dataset:

a) Local: The user browses for a folder in their local machine, in which a dataset already exists.

b) URL: The user inputs a web address which downloads a compressed file (we currently support .zip and .tar files), as well as a dataset destination folder in their local machine. We proceed to save the compressed file at the provided path, and then extract the contents of the file into that folder.

c) Make: By using the microphone on the Arduino board we provide the option of recording a new dataset from within the UI. The user inputs all the labels they will be performing multiclass classification on, as well as the number of tracks they will be recording for each class. After specifying the location where they wish to save their own dataset and pressing the "Record" button on the UI, the user is guided through the recording process, at the end of which they have created their own dataset. It should be noted that the UI is able to detect existing class folders and can append recordings to an existing dataset. Hence, the user does not have to record all the samples for a label at once.



Data mode: Local URL [✓ Make]

Dataset destination: /Users/hk/Desktop/dataset

Labels and samples per label: nine, zero, four, cat, dog, mouse 12

Expected duration (s): 1.00

Sample rate (Hz): [✓ 16000] 41667

Figure 3. Dataset page of the UI

It is important to note here that our platform relies on a particular dataset structure. Namely, we require that the dataset folder contains *only* one folder for each of the classes, with the folder name being identical to the class name. We also require that within each of the folders, *only* the audio files for that class exist. This is required from the users in the "Local" and "URL" dataset options, but handled automatically in the "Make" case. For the "Make" case, the destination path must be an empty folder or a folder that contains other folders with only audio files in them.

The dataset page also includes input fields for important dataset parameters: expected duration of the audio tracks and sample rate. The expected duration is used to ensure that all tracks are the same size, achieved by either truncating if longer than expected, or zero-padding if shorter than expected, while the sample rate specifies the frequency at which we sample the audio data when loading a file. The range of values provided for the expected duration is limited by the space constraints on the Arduino board used during this project, while the distinct available values for the sample rate are defined by the values supported by the Arduino microphone library.

5.1.2 Data Processing

We offer three different processing methods: Averaging, Short Time Fourier Transfrom (STFT) and Windowed Root Mean Square (WRMS). Additionally, we provide the option of tuning the window size and the window stride for the latter two methods (the former operates on fixed slices of 32 samples). An example for STFT is shown in Fig.4 [2] where the chosen window size is 30ms and the window stride is 20ms. The processing is identical for WRMS with the exception that we calculate the RMS of all samples in the window instead of taking FFT of the windowed signal.

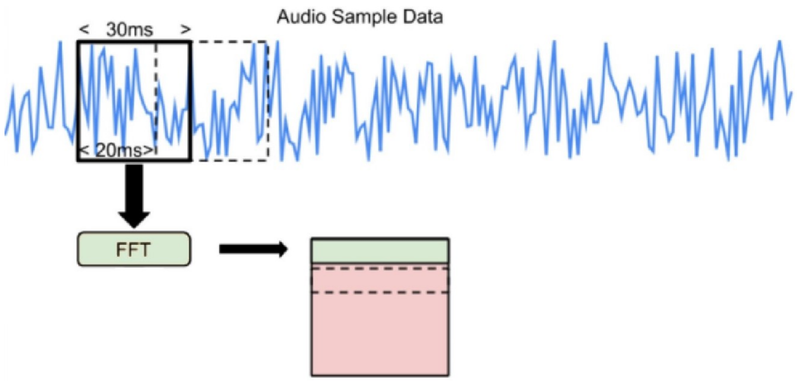
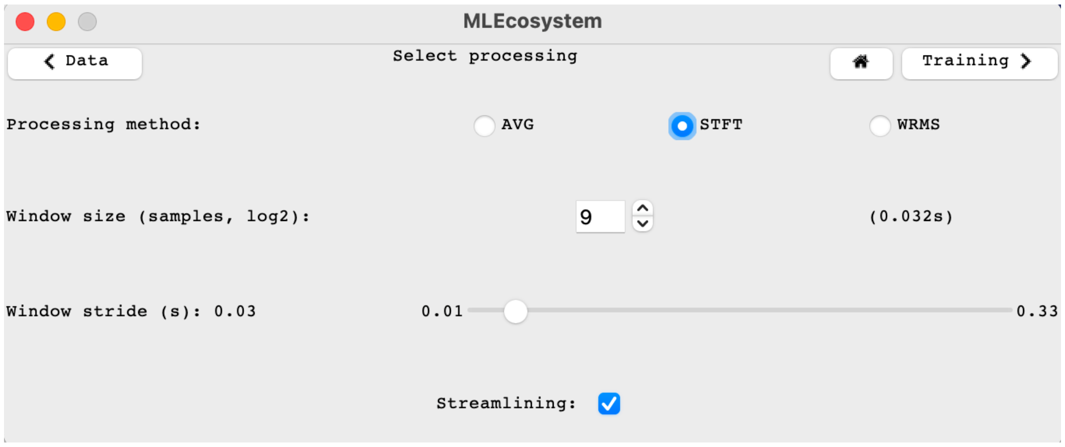


Figure 4. Example of STFT processing with a window size of 30ms and a window stride of 20ms

We also provide the users with a default processing method and parameters which allow beginners to get the most out of their training time. We have taken care to ensure that our approach is robust by limiting the parameters to reasonable values such that training will always go smoothly.



Processing method: AVG [✓ STFT] WRMS

Window size (samples, log2): 9 (0.032s)

Window stride (s): 0.03 0.01 0.33

Streamlining: ✓

Figure 5. Processing page of the UI

An important feature included in this page is the option of streamlining. This refers to a restructuring in the flow of data within the pipeline. While normally the entire dataset would be processed and subsequently trained on, during

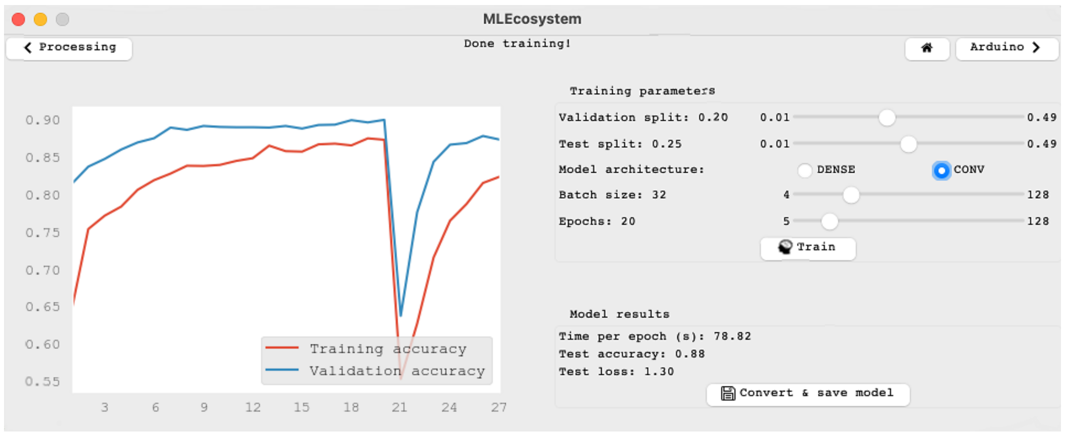
[2] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro/examples/micro_speech/train

streamlining processing and training occur for one batch a time. This enables the processing of larger datasets, as well as the use of processing configurations which result in very large feature maps, both of which would otherwise potentially lead to RAM depletion. While this approach is very memory efficient, it adds a considerable overhead and leads to an increase in training time. We therefore recommend this be opted for only when necessary.

5.1.3 Model Training and Conversion

Having configured the audio signal processing, we are ready to train a Machine Learning model. Up until now, this project has utilized Neural Networks as the hypothesis class, with which we attempt to tackle the learning problem of multiclass classification.

Prior to training, the user is prompted to set various Machine Learning-related fields. We have allowed for control over dataset allocation for validation and testing, batch size, number of epochs, and a choice between a Dense and a Convolutional model architecture. During training, our UI creates a real-time plot of model performance, updating training and validation accuracy curves at the end of every epoch. At the end of training we evaluate the model on the test set, reporting a test accuracy and loss.



Done training!

Training parameters

Validation split: 0.20	0.01	0.49
Test split: 0.25	0.01	0.49
Model architecture:	DENSE	[✓ CONV]
Batch size: 32	4	128
Epochs: 20	5	128

Model results

Time per epoch (s): 78.82
Test accuracy: 0.88
Test loss: 1.30

Figure 6. Training page of the UI

Since the model we are training is going to be deployed on a microcontroller, where size constraints are an inherent limitation, we also perform model quantization to reduce the model size. Quantization refers to converting model parameters to types with reduced precision, such as from a 32-bit value to an 8-bit value [4]. This 4x reduction in model size, however, does not come without a price, which in this case is a very significant drop in performance during inference. To minimize this drop, we utilize quantization-aware training [5]. This refers to the fact that after completing the specified number of epochs in training, we proceed to train for an additional number of epochs during which we simulate quantized inference. While this at first leads to a large drop in accuracy, as can clearly be seen in the figure above, the model attempts to adapt to the new quantized setting, leading to a reduction in the performance drop that will occur during inference.

Finally, this page handles the conversion and saving of the trained model. At first, we convert the trained model to a TensorFlow Lite (.tflite) file [6], and subsequently to a C header (.h) file, a format which is readable by the Arduino. To this header file we also append lines containing several parameters that must be communicated to the Arduino for running inference. An example of such lines is given above. The header file containing the model as well as the parameters is then saved in a user-specified location.

```
// ADDED LINES START
constexpr char* processing_method = "stft";
constexpr double expected_duration = 1.0;
constexpr int sample_rate = 16000;
constexpr int window_size_samples = 512;
constexpr int window_stride_samples = 480;

const char* RESPONSES[] = {"no", "yes"};
unsigned short NUM_RESPONSES = 2;
// ADDED LINES END
```

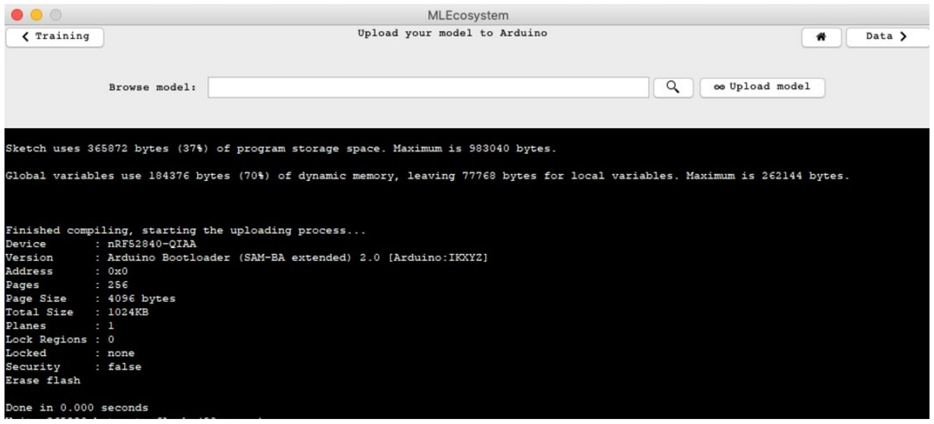
Figure 7. Parameter passing to the arduino via the model header file

5.1.4 Model Deployment

The final stage of the pipeline consists of choosing a saved model (which at this point is a header (.h) file), and then compiling and uploading an Arduino script which will be able to run inference using the selected model. It is important to note that we do not require that the chosen model file contains a model that has just been trained, converted, and saved. Therefore, the user is able to upload any model which they have created and saved in the past.

Normally, the user would be required to go through a list of manual steps, namely move the Arduino file to an appropriate location, include the necessary header files, open the Arduino IDE, select the communication port at which the board is located, and compile and upload their Arduino inference script. Besides the number of required manual steps, this process can also be buggy and time-consuming, due to the fact that the IDE faces multiple connection errors and is usually not able to connect to the Arduino board. To minimise the issues in this part of the pipeline, a novel approach to compile and upload the script was established.

The UI makes use of the Arduino-CLI [7], a Command Line Interface that can be used to communicate to the board and upload any script to it. Through this, the UI is able to automatically detect all necessary information, such as the communication port or the name of the board, and subsequently compile and upload the script. After obtaining this information, we place the selected model in the same folder as the Arduino file we are uploading, which has been custom made by us. This results in a fully automated process, for which the user solely has to choose the model to upload. The UI page behind which this process occurs is shown in the figure below. In case the CLI has not been installed before on the user's device, the UI also detects that and automatically installs it on their system. This whole process is compatible with Windows, MacOS and Linux.



```
Sketch uses 365872 bytes (37%) of program storage space. Maximum is 983040 bytes.

Global variables use 184376 bytes (70%) of dynamic memory, leaving 77768 bytes for local variables. Maximum is 262144 bytes.


Finished compiling, starting the uploading process...
Device       : nRF52840-QIAA
Version      : Arduino Bootloader (SAM-BA extended) 2.0 [Arduino:IKXYZ]
Address      : 0x0
Pages        : 256
Page Size    : 4096 bytes
Total Size   : 1024KB
Planes       : 1
Lock Regions : 0
Locked       : none
Security     : false
Erase flash

Done in 0.000 seconds
```

Figure 8. Arduino page of the UI

5.2. Arduino-side Pipeline

While the focus of this report up until now has been the pipeline on the side of the user's computer, we must also consider that all of the above features need to be compatible with the Arduino board in order for deployment to be successful. The microcontroller has strict requirements both in terms of memory and processing power. We therefore had to come up with efficient and optimised algorithms to replicate the Python-side.

We begin by describing our approach to reliable audio reception. The microcontroller is collecting samples at 16kHz or 41.667kHz, meaning that computations need to be as quick as possible. If that is not the case, the built in PDM[3] callback, the interrupt running whenever new samples appear at the microphone's internal buffer, will be changing the stored audio data during runtime of other functions, leading to unpredictable and potentially incorrect processing. Aside from optimising data processing, a strategy was devised for extracting audio data as efficiently as possible, with minimum computations. The algorithm is shown in Fig.9. For compactness and clarity, we present the snippet pseudo-code:

```
1   if samplesRead > 0:
2
3     updateBuffer(sampleBuffer, samplesRead)
4
5     if bufferIsCorrupted:
6       corruptionCounter -= samplesRead;
7       if corruptionCounter <= 0:
8         bufferIsCorrupted = false
9
10    if not bufferIsCorrupted:
11      meanSquare = calculateMeanSquare(sampleBuffer) // over the first fourth of the buffer
12      audioIsLoud = (meanSquare > THRESHOLD)
13
14    if audioIsLoud and not bufferIsCorrupted:
15      processing()
16      runInference()
17      bufferIsCorrupted = true
18      corruptionCounter = SAMPLE_BUFFER_SIZE
```

Figure 9. Audio sampling algorithm

The above is a high level description of the inference script's main loop. Beginning at line 1, the loop only runs if there are new samples available (collected from the PDM callback). If that is the case, the new samples are stored in our buffer which has length *SAMPLE_BUFFER_SIZE*[4].

The *bufferIsCorrupted* boolean is used as a check to determine if the buffer contains valid audio. The *corruptionCounter* variable denotes the number of samples that need to be renewed before the buffer is no longer corrupted.

In the case that the buffer contains valid audio, lines 10 to 12 perform a test to see if the audio is loud and hence potentially meaningful. The test is done by using the mean square of the data and comparing it to a tuned threshold value[5].

[3]Pulse Density Modulation. The board has a built in PDM microphone.
[4]Equivalent to the expected duration multiplied by the sample rate (refer to the Dataset Selection section).
[5]Note that the mean square is evaluated only over the first fourth of the buffer. This is equivalent to saying that we only deem the audio meaningful if the user has just finished speaking.

Lines 14 to 18 are responsible for processing the audio and running inference. Note that this only happens if the audio is loud, as per line 12, and if the the buffer contains valid audio. Since processing and running inference are potentially time consuming tasks, it is likely that within that time frame the PDM callback will be triggered more than once. Because the main loop is busy with the aforementioned tasks, this means that the *updateBuffer* function will not be called right after new samples are obtained. This leads to buffer corruption. Hence, after inference we declare that the buffer is corrupted and we set the *corruptionCounter* to it's maximum possible value.

Having explained the main loop of the Arduino script, it remains to optimise lines 15 and 16, i.e. processing and running inference. For processing we use the CMSIS DSP library[8] and in particular its optimised FFT[6]. For running inference we use Tensorflow's microcontroller library[9].

6. Testing

To evaluate the performance of our system, we ran a series of tests using the Speech Commands dataset [10]. The tests were carried out using the default processing configuration of the UI, STFT with a window size of 512 samples and a window stride of 480 samples. For the model we used the default convolutional architecture, as shown in Fig.10 below.

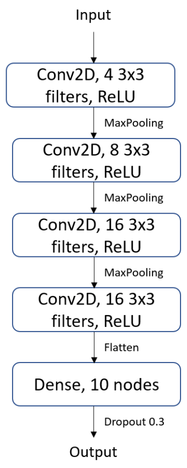

Input
Conv2D, 4 3x3 filters, ReLU
MaxPooling
Conv2D, 8 3x3 filters, ReLU
MaxPooling
Conv2D, 16 3x3 filters, ReLU
MaxPooling
Conv2D, 16 3x3 filters, ReLU
Flatten
Dense, 10 nodes
Dropout 0.3
Output

Figure 10. Model used for testing

Our testing consisted of using the same model to classify datasets of different sizes, where size was defined as the number of different classes within the dataset. Starting from 2 words and incrementing to 8 words, we generated 5 distinct datasets for each of the sizes, choosing classes randomly from the 36 Speech Commands classes. For example, when testing for 3 words, we would randomly obtain 3 of the 36 Speech Commands classes and make a dataset from them. Doing this 5 five times results in 5 different 3-word-long datasets.

After having obtained the datasets for a particular size, we trained the aforementioned model and subsequently obtained a test accuracy for each. We then recorded the minimum, maximum, and average of these 5 trials. It should be noted that we use recognition accuracy on the test set as the model performance metric, since it is the most commonly used metric for audio applications.

[6]Fast Fourier Transform.

Fig.11 shows the results of the above process. For each of the distinct dataset sizes we plot the minimum, maximum, and average of the 5 trials as mentioned above, as well as the performance of a random model.

We can make the following observations:

- Test accuracy decreases as the dataset size increases.
- After 4 words, the training and validation curves for accuracy start decreasing. This implies that the models are underfitting.
- Despite the above points, our fixed model always manages to perform better than random guessing. Initially the margin is substantial, but decreases as the dataset size increases.

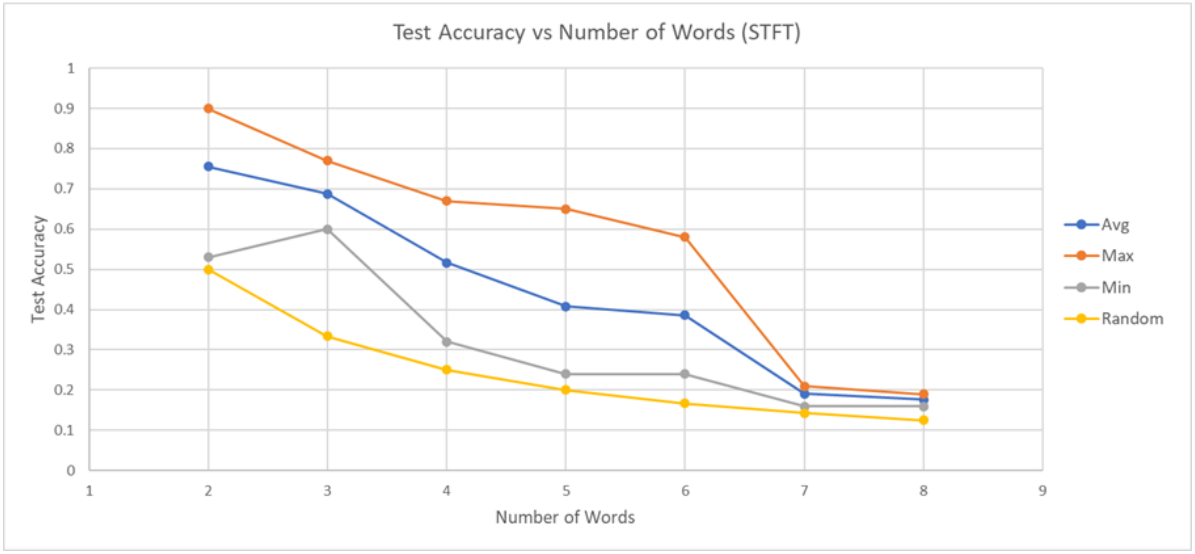


Figure 11. Accuracy vs Number of words (STFT)

Given the above, training with multiple words leads to poor results. This is due to the model not being complex enough. Unfortunately this issue is to some extent unavoidable given the memory limitations on the Arduino side. Solutions to this problem are actively being worked on at the moment (See Section 9 - AutoKeras).

7. Ethical, Legal and Sustainability Concerns

7.1. Ethical

The Arduino BLE Sense 33 board has a built-in microphone, which can potentially lead to some privacy concerns. For example, after the user makes their own dataset, the script to capture microphone data keeps recording until another script is uploaded to the Arduino. Unless the user decides to upload a model, the Arduino will continue recording audio if plugged in. To prevent unnecessary recording, a dummy script is uploaded to the Arduino board to stop the microphone use.

More generally, these concerns are minimized by the fact that the audio data collected by our system is only saved in the user's computer, rather than communicated to cloud services or to other users. However, this doesn't protect against all possible issues, since others may adapt our product to be used in one of the following unethical ways:

- Audio recording consent: The audio samples can be recorded without user permission, infringing on data consent rights.
- Data interpretation: Using audio datasets may lead to model bias. For example, if the audio datasets supplied were from a group of people with similar accents, the model will not perform well when running inference with voice with a different accent. Concerns of racism or other types of discrimination may arise, in which case, the dataset may need to be re-collected to reflect a better variety.
- Data collection: The audio samples collected from the microcontroller and classification results might be sold to data analytics company without user consent to target ads for certain products. For example, if the users want to identify the dog barking sounds, the data will be collected and the user will be advertised dog food and products.

7.2. Legal

- Use of libraries: All libraries used in the project are either standard libraries or open-source libraries, in which case there are clear instructions describing which libraries are used and how to install them in README file.
- Intellectual Property Rights: All of the code was contributed by members of the team and the team holds the copyright of all code produced. The team has decided to release all source code produced as part of the project to the public under the permissive Apache license.

7.3. Sustainability

The project was designed to be user-friendly and sustainable in the long run.

- Long term maintenance: The project has many stretch goals and features to be implemented in the future. However, the baseline project which allows users to train a few words with different audio processing methods and upload the trained model through the interface are documented in a README file on GitHub. The code is commented to improve readability and allow enthusiasts to add features if needed.
- Energy saving: The microcontroller used in the project can be powered with a 3.3V supply. With the onboard features, it is a powerful yet energy efficient micro-controller for TinyML while providing decent performance for audio classification.

8. Project Management

Project management was especially important when developing this platform because all group members were situated in different locations and time-zones. To ensure constant progress, short 30-minute meetings were held everyday to analyse progress of the previous day and set the agenda for that day. Additionally, client meetings were held every Friday to update the client on the progress and get their feedback and advice on that week's technical issues. An additional team meeting was held after this to discuss the agenda for the upcoming week.

8.1. Meeting minutes and recordings

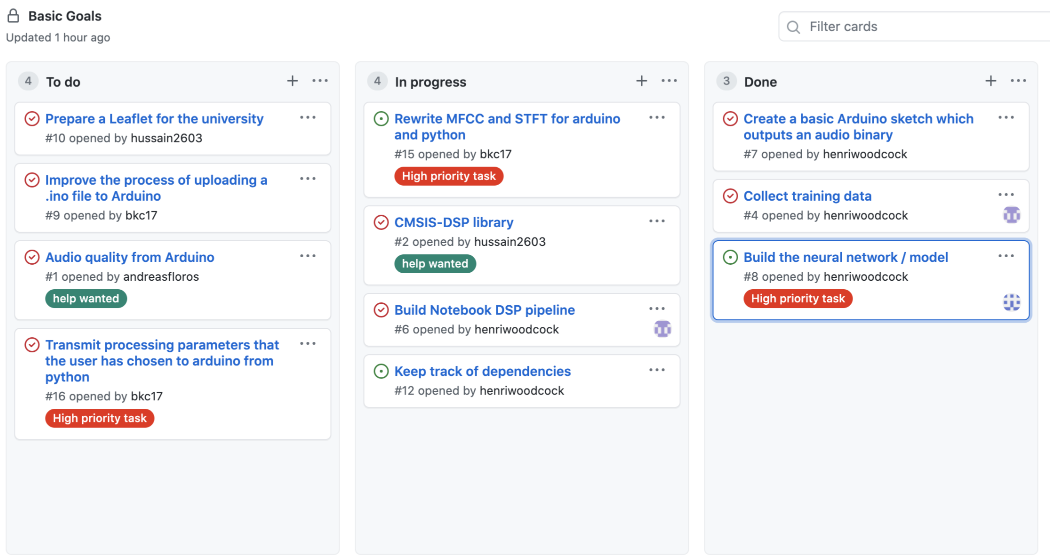
Minutes taken during all the meetings were especially helpful in setting the agenda. Meetings were mostly held on Microsoft Teams and constant communication was also maintained on WhatsApp.

8.2. GitHub

We chose GitHub for sharing the code with all the group members, due to the benefits provided by version-control. GitHub issues were opened with relevant tags when a member encountered technical difficulties, so that members of the client team could provide support. A README file was maintained regularly to provide an intuitive introduction to our project, showcase the UI operation using GIFs, and explain all the different aspects of the code. This is particularly useful to all the Users and also for the enthusiasts who want to add more features to the platform on their own. It should be noted that the majority of our work was done on a private ARM repository which is now archived. For our current implementation and updates please click on our public repository: Project EMILY

8.3. Kanban Board

Kanban board was created on the GitHub page to keep track of the progress of the project. Through Kanban we were able to separate the development of the platform into different tasks and assign each to different people. This helped with division of labour and made the responsibilities clearer to the whole team.



Basic Goals
Updated 1 hour ago
Filter cards

4 To do
- Prepare a Leaflet for the university
  #10 opened by hussain2603
- Improve the process of uploading a .ino file to Arduino
  #9 opened by bkc17
- Audio quality from Arduino
  #1 opened by andreasfloros
  help wanted
- Transmit processing parameters that the user has chosen to arduino from python
  #16 opened by bkc17
  High priority task

4 In progress
- Rewrite MFCC and STFT for arduino and python
  #15 opened by bkc17
  High priority task
- CMSIS-DSP library
  #2 opened by hussain2603
  help wanted
- Build Notebook DSP pipeline
  #6 opened by henriwoodcock
- Keep track of dependencies
  #12 opened by henriwoodcock

3 Done
- Create a basic Arduino sketch which outputs an audio binary
  #7 opened by henriwoodcock
- Collect training data
  #4 opened by henriwoodcock
- Build the neural network / model
  #8 opened by henriwoodcock
  High priority task

Figure 12. A snippet of Kanban board in week 4

9. Future Work

As with any project with time constraints, some features that were under development were not completed. Below we have listed these features, as well as other future goals for the platform:

- Add more models to the system or extend support for use of AutoKeras. The current system only gives limited flexibility of choosing two fixed models. Adding more model choices improves this flexibility and the users can look through a wider range of models to choose one that fits their purpose. Implementing Autokeras would allow the platform to automatically come up with a model given certain constraints and this may be a good option for users who are not experts in Machine Learning or those who don't know much about model choices/parameters.
- Include more advanced data pre-processing methods, such as, Mel-Frequency Cepstral Coefficients(MFCC), Meta-Feature Extractor(MFE), etc. These methods are known to be some of the best feature extraction

methods for audio data. Implementing more complicated data pre-processing methods on micro-controller efficiently is challenging at the moment due to memory and processing power constraints.

- Implement data augmentation for better generalisation of the model. In case of insufficient audio samples, data augmentation can prevent the model from overfitting and therefore provide better generalisation to test data.
- Introduce more hyper-parameter choices such as different optimisers or number and type of layers. These choices will give experienced users more freedom and allow them to obtain a better model.
- Train a general model using a large dataset and allow users to use the model and finetune it for their use case by supplying a smaller dataset, rather than train the model from scratch.
- Extend support to other Embedded ML boards and operating systems to expand the user base.
- Improve the appearance of the User Interface.

10. Conclusion

With the rising popularity of TinyML, embedded machine learning has a huge potential market. Building the embedded machine learning ecosystem for users from different technical backgrounds is therefore crucial. At the moment, there are very few companies and projects dedicated to building that ecosystem. This project managed to build a foundation for creating and deploying audio classifiers on a microcontroller. By implementing all aspects of the pipeline, from selecting a dataset to uploading the model to the Arduino, we have created a beginner-friendly, all-in-one platform.

However, we recognise there are still some limitations to our product. We can only run audio classification on a relatively small dataset, as mentioned in Section 6, and there is scope for improvement in several aspects of our platform, as discussed in Section 9. The field of TinyML is still in its early stages, and more work is required to optimise the ecosystem. Our public GitHub repository, Project EMILY, contains more technical details, a guide for setting up and using the platform, as well as information on contributing.

References

[1] Krishna Rangasayee. *2021: The Year of ML Scaling at the Embedded Edge*. 2021. URL: https://sima.ai/technology/2021-the-year-of-ml-scaling-at-the-embedded-edge/.

[2] Evan Schuman and Lauren Horwitz. *Predictions for Embedded Machine Learning for IoT in 2021*. URL: https://www.iotworldtoday.com/2020/12/10/predictions-for-embedded-machine-learning-for-iot-in-2021/.

[3] ArduinoStore. *Arduino Board: Arduino Nano 33 BLE Sense*. URL: https://store.arduino.cc/arduino-nano-33-ble-sense.

[4] Tensorflow. *Post-training quantization*. URL: https://www.tensorflow.org/model_optimization/guide/quantization/post_training.

[5] Tensorflow. *Quantization-aware training*. URL: https://www.tensorflow.org/model_optimization/guide/quantization/training.

[6] Tensorflow. *Tensorflow Lite*. URL: https://www.tensorflow.org/lite.

[7] Arduino. *Arduino-CLI*. URL: https://www.arduino.cc/pro/cli.

[8] ARM. *CMSIS Software Library*. URL: https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html.

[9] Tensorflow. *Tensorflow Lite for Microcontrollers*. URL: https://www.tensorflow.org/lite/microcontrollers.

[10] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. arXiv: 1804.03209 [cs.CL].