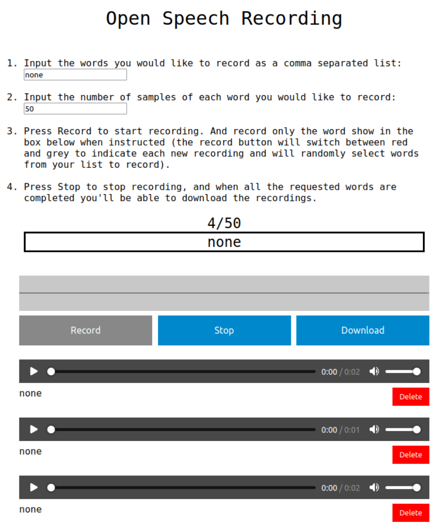
Wake-Word Detection with TinyML

December 30, 2023

1 Introduction

This project focuses on developing an audio analysis system capable of recognizing absolutist keywords, potentially indicative of mental health conditions. The objective is to create a low-power, efficient system for monitoring daily mental health language markers, using specific keywords like ”never”, ”none”, ”all”, ”must”, and ”only”.

2 Experiment



Open Speech Recording

1. Input the words you would like to record as a comma separated list: none
2. Input the number of samples of each word you would like to record: 50
3. Press Record to start recording. And record only the word show in the box below when instructed (the record button will switch between red and grey to indicate each new recording and will randomly select words from your list to record).
4. Press Stop to stop recording, and when all the requested words are completed you'll be able to download the recordings.

4/50

none

Figure 1: Speech recording of the keyword ”none” using the Open Speech recording Tool

This project involved the collection of new audio data, a critical aspect of TinyML application development. I focused on five specific absolutist keywords: “never”, ”none”, ”all”, ”must”, and ”only”. Utilizing the Open Speech Recording tool, at least 50 recordings per keyword were gathered, totaling a minimum of 250 samples. These recordings, initially in “.ogg” format, were converted to “.wav” format for compatibility. The collected dataset was then integrated with the Speech Commands dataset from Pete Warden, resulting in a comprehensive and diverse dataset for training the machine learning model. This approach ensured the inclusion of a variety of real-world audio scenarios, including different background noise levels. Additionally, the project encouraged collaboration and data sharing among students, further enriching the dataset's diversity and size.

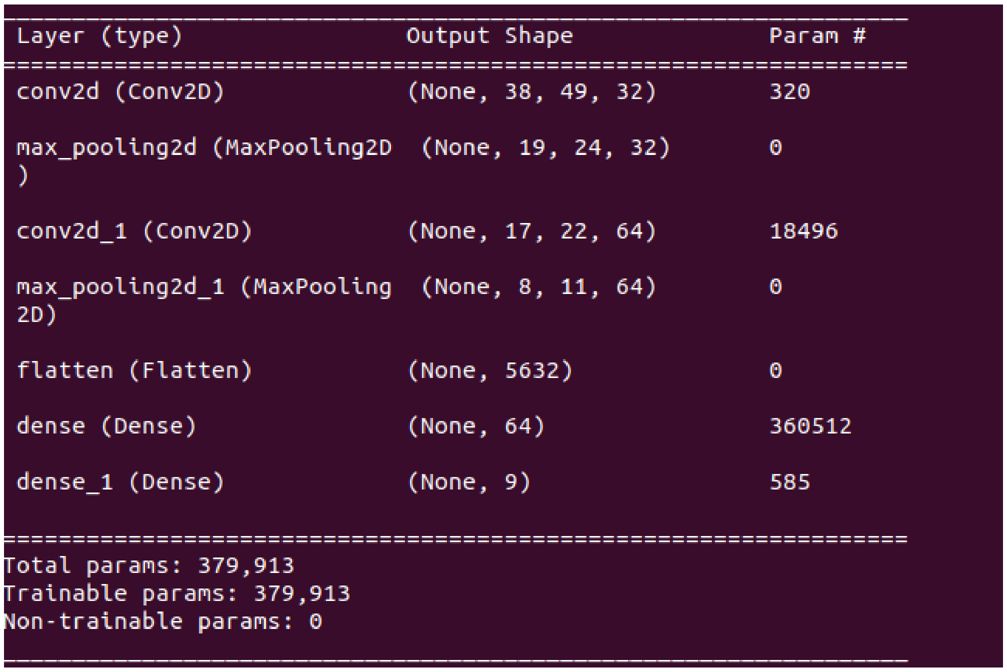
3 Algorithm

In this project, I developed a machine learning model focusing on the recognition of specific keywords in audio data using spectrograms and convolutional neural networks (CNNs).

3.1 Data Processing and Feature Extraction

- **Spectrogram Generation:** Each audio file was converted into a spectrogram using the librosa library. Spectrograms were generated by applying a Mel-scale filter to the Fourier transform of the audio signal.
- **Data Padding:** The generated spectrograms were padded to ensure uniformity in dimensions across all samples.

3.2 Model Architecture



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 38, 49, 32)        320

 max_pooling2d (MaxPooling2D  (None, 19, 24, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 17, 22, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 8, 11, 64)        0
 2D)

 flatten (Flatten)           (None, 5632)              0

 dense (Dense)               (None, 64)                360512

 dense_1 (Dense)             (None, 9)                 585

=================================================================
Total params: 379,913
Trainable params: 379,913
Non-trainable params: 0
_________________________________________________________________
```

Figure 2: CNN Model

- **Input Layer:** Accepts the reshaped spectrograms.

- **Convolutional Layer 1:** Conv2D layer with 32 filters, followed by a MaxPooling2D layer.
- **Convolutional Layer 2:** Conv2D layer with 64 filters, followed by another MaxPooling2D layer.
- **Flatten Layer:** Converts the 2D feature maps into a 1D vector.
- **Dense Layer 1:** Comprises 64 neurons with ReLU activation.
- **Dense Layer 2:** Another set of 64 neurons with ReLU activation.
- **Output Layer:** A dense layer with softmax activation, outputting the probability distribution over the target classes.

3.2.1 Training Process

The model was trained with the Adam optimizer and sparse categorical cross-entropy loss function. A validation split was used during training to monitor performance and mitigate overfitting. Post-training, the model was quantized and converted to TensorFlow Lite format for deployment on low-power devices. This approach aligns with the project's objective of developing an embedded system for real-time audio analysis.

3.2.2 Training Parameters

- **Optimizer**: Adam. A popular optimization algorithm in deep learning, known for its efficiency with sparse gradients and adaptive learning rate.
- **Loss Function**: Sparse Categorical Crossentropy. Suitable for multi-class classification where targets are integers.
- **Metrics**: Accuracy. Used to measure the model's performance.
- **Number of Epochs**: 10. Each epoch represents a complete pass of the training dataset through the neural network.
- **Validation Data**: Used for evaluating model performance after each epoch. Consists of *test_spectrograms* and *test_labels*.
- **Batch Size**: Typically 32 (default in Keras). Determines the number of samples processed before the model is updated.

3.3 Deployment on Arduino

The deployment of the TensorFlow Lite model on the Arduino platform involved the following steps:

1. **Initialization:** Set up the Arduino environment and imported necessary TensorFlow Lite libraries.

2. **Model Loading:** Loaded the pre-trained TensorFlow Lite model into the Arduino's memory.
3. **Interpreter Setup:** Created a `MicroInterpreter` with specific operation resolvers.
4. **Memory Allocation:** Allocated memory for input, output, and intermediate arrays in a defined tensor arena.
5. **Input Processing:** Prepared to process input audio signals using feature providers and recognizers.
6. **Main Loop Execution:** Continuously processed audio data, generating spectrograms and running model inference.
7. **Command Recognition:** Analyzed model output to recognize commands using the `RecognizeCommands` class.
8. **Response Handling:** Executed actions based on recognized commands.
9. **Profiling (Optional):** Measured and reported inference cycle times for performance optimization.

I modified the `micro_speech` example to incorporate my model, ensuring compatibility and optimal functioning. Necessary adjustments were made to accommodate the specific requirements of my model while keeping most of the original code unchanged.

4 Results & Analysis

The training process yielded promising results, demonstrating high accuracy in keyword spotting. The key outcomes are as follows:

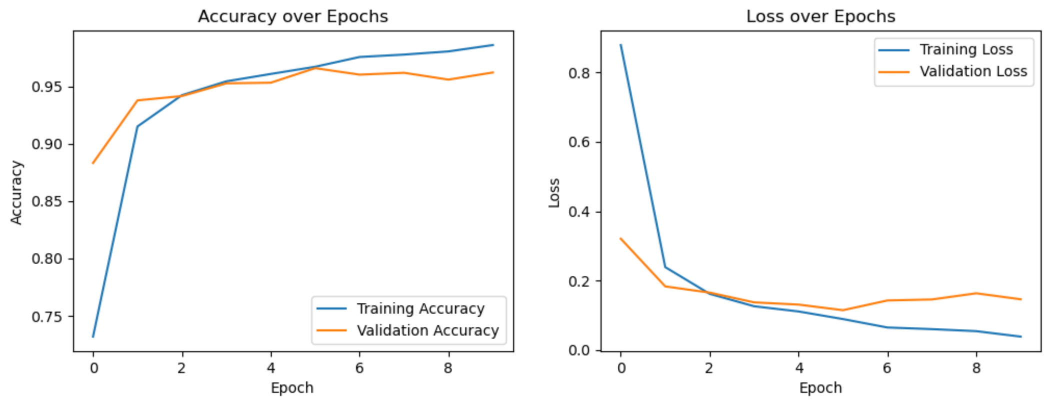
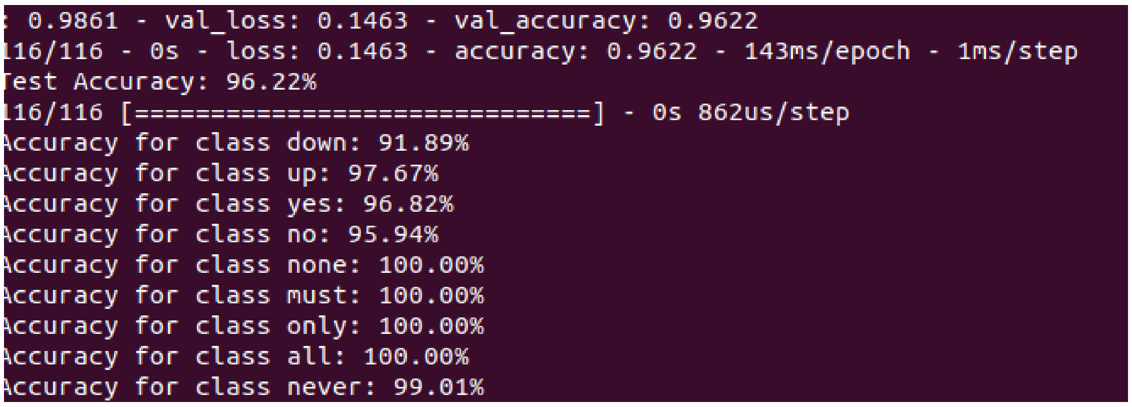


Figure 3: Accuracy and Loss plots for the CNN model.

- The model achieved a **training accuracy of 96.22%**, indicating robust learning and generalization capabilities.

- In the **test set evaluation**, the model maintained a **high accuracy of 96.22%**, reflecting its effectiveness on unseen data.
- The **class-wise accuracies** were particularly impressive, with most classes achieving near-perfect recognition rates. Specifically:
  - ‘down‘: 91.89%
  - ‘up‘: 97.67%
  - ‘yes‘: 96.82%
  - ‘no‘: 95.94%
  - ‘none‘, ‘must‘, ‘only‘, ‘all‘: 100.00%
  - ‘never‘: 99.01%

These results indicate the model’s effective differentiation between various keywords, including challenging ones, and underscore its potential for real-world applications in voice-controlled systems.



```
: 0.9861 - val_loss: 0.1463 - val_accuracy: 0.9622
116/116 - 0s - loss: 0.1463 - accuracy: 0.9622 - 143ms/epoch - 1ms/step
Test Accuracy: 96.22%
116/116 [==============================] - 0s 862us/step
Accuracy for class down: 91.89%
Accuracy for class up: 97.67%
Accuracy for class yes: 96.82%
Accuracy for class no: 95.94%
Accuracy for class none: 100.00%
Accuracy for class must: 100.00%
Accuracy for class only: 100.00%
Accuracy for class all: 100.00%
Accuracy for class never: 99.01%
```

Figure 4: Accuracy of the CNN model.

4.0.1 Real-time Prediction Performance

The deployment of the model for real-time audio keyword recognition revealed significant insights. Initially, the model displayed high prediction accuracy in a simplified context with only two classes: ‘yes‘ and ‘no‘. This high accuracy underscored the model’s capability in a constrained environment.

However, as the complexity increased with the number of classes expanding to 17, including ‘silence‘, ‘unknown‘, ‘left‘, ‘right‘, ‘off‘, ‘stop‘, ‘go‘, ‘on‘, and others, a noticeable decline in prediction accuracy was observed. This reduction can be attributed to the increased challenge in distinguishing between a larger set of keywords. Additionally, environmental factors such as background noise played a significant role in impacting the model’s performance.

This scenario highlights the need for models to be robust against noise and capable of handling a wide range of classes in real-world applications. The insights from this analysis are crucial for further refining the model and improving its real-time prediction capabilities.

5 Discussion and Summary

5.1 Summary of Results

The project achieved a remarkable test accuracy of 96.22

5.2 Challenges and Difficulties

The main challenges included dealing with a large number of classes and managing environmental noise in real-time predictions. The complexity of distinguishing between multiple classes significantly affected accuracy.

5.3 Future Resolutions

To improve, future iterations could focus on enhancing noise robustness and implementing more advanced data augmentation techniques. Also, exploring deeper or more complex neural network architectures may yield better results in differentiating between a larger set of keywords.

5.4 Project Difficulties

The most difficult part was ensuring the model's efficiency and accuracy in real-time scenarios, especially given the limitations of computational resources on microcontrollers.

5.5 Improvements and Real-Time Accuracy Expectations

Improvements could include integrating a more diverse dataset and refining the feature extraction process. Although the real-time prediction accuracy was lower than test accuracy, it provided valuable insights into the model's practical applicability and areas for enhancement.