EchoPlay
Voice Controlled Music Player

1st Ilo Chen
Electrical Engineering
National Taiwan University
b11901173@ntu.edu.tw

2nd Justin Yi
Electrical Engineering
National Taiwan University
b11901150@ntu.edu.tw

***Abstract*—We present a voice-command controlled, low power, bluetooth music speaker system to showcase IoT edge processing. Edge nodes employ data preprocessing techniques to efficiently extract and send audio data features in real-time. The central server runs lightweight ML models and uses a simple state machine to parse commands. The system is highly flexible and can be modularized for different applications.**

I. Introduction

Our system uses an STM32 B-L475E-IOT01A board as the IoT node for recording, preprocessing, and Wifi transmission. We use a Raspberry Pi 5 as the central server for Wifi receival, music playback, and keyword prediction. Keyword prediction is done through two lightweight (1Mb), shallow, fully-connected neural networks. This paper introduces the techniques we used throughout the system. In the second section, we detail the techniques used on the STM32 board for recording and preprocessing as well as the environment setup and Wifi Transmission. In the third section, we explain the steps we took to pretrain our ML models to make them robust to noise which includes custom dataset generation. In the fourth section, we outline the setup of our Raspberry Pi 5 and how we integrate different components into one structure.

II. STM32 IoT Node

The STM32 board is a versatile microcontroller; however, its primary limitation lies in its restricted memory size of only 128 KB. Consequently, we must adopt creative strategies to manage memory efficiently while storing and processing data. Additionally, the real-time requirements of the system compel us to utilize parallel processing techniques effectively. This section details our solutions in three parts, focusing on audio recording, audio preprocessing, and audio transmission.

A. Audio Recording and DMA Setup

The recorded audio is intended for feature extraction and keyword spotting, necessitating continuous recording at a sufficiently high frequency. We opted to record at 8000 Hz for 1 second and store the results in an array. To address the constraints of limited memory, we divided this large array into eight smaller sub-arrays and implemented circular DMA to handle the data.

Identify applicable funding agency here. If none, delete this.

Using this approach, raw audio data is stored in a 1024-element array in memory, which is repeatedly transferred into the larger audio buffer. This strategy takes advantage of the time DMA spends loading half of the array to preprocess the other half before it is stored in the buffer. As a result, the audio recordings are processed concurrently with the recording operation, significantly reducing system latency.

B. Audio Preprocessing for STFT

To capture both temporal and frequency characteristics of the audio data, we compute its spectrogram using the Short-Time Fourier Transform (STFT), defined as:

$$X(t,\omega) = \int_{-\infty}^{\infty} x(\tau) w(\tau - t) e^{-j\omega\tau} d\tau \tag{1}$$

Here, the spectrogram is derived by calculating the magnitude squared of the STFT:

$$S(t,\omega) = |X(t,\omega)|^2 \tag{2}$$

Since sending a 2-dimensional spectrogram array can be resource-intensive, we leverage the locality of the STFT calculation by selecting an optimal hamming window size and overlap. Specifically, we use an FFT size of 256 and a window overlap of 50%. This configuration allows for efficient sequential preprocessing of data as it is recorded.

To perform these calculations, we use the ARM-CMSIS DSP library's RFFT functions, which are optimized for STM32 devices. By ensuring the chosen FFT size and overlap complete the required calculations before the DMA cycles back to the same memory segment, we guarantee both efficiency and accuracy in the audio preprocessing stage.

C. Audio Transmission with TCP Server

To mitigate packet loss, the recorded audio is transmitted as 32 smaller recordings, each comprising 256 samples, instead of sending the full recording as a single data block. This chunking approach allows the server to detect and recover from packet loss more effectively, reducing the overall impact of a misdelivery.

This strategy not only ensures the reliable transmission of data but also decreases the cost and complexity associated with handling packet loss. The robust design facilitates seamless communication, even in network environments prone to instability.

III. Machine Learning Model

To efficiently recognize and classify voice commands sent from the STM32 into nine distinct keywords—"pop," "classic," "hits," "rock," "start," "pause," "next," "back," and "choose"—we employ a machine learning model. However, training a lightweight model capable of accurately classifying such a large number of keywords is challenging. To address this, we divide the task into two separate keyword spotting models: one dedicated to identifying music genres and the other focused on playback commands.

Let's start by explaining how we collected the dataset, followed by an overview of the model training process.

A. *Data Collection*

To efficiently classify each keyword, we need at least a thousand samples per keyword. However, our target keywords are not included in the Google Speech Commands dataset. Without an easy way to generate these samples automatically, we would face the daunting task of recording them manually. To address this, we decided to use a pre-trained Text-To-Speech (TTS) model from the TTS 0.22.0 library, which offers a diverse set of approximately 109 different speakers.

An additional advantage of using the TTS model is the ability to standardize our dataset to match the STM32's audio specifications: an 8kHz sampling frequency and a duration of 1 second per sample. This ensures consistency between our training data and the real-world deployment environment.

The TTS model provides us with 109 unique speakers, but this falls far short of the 1,000 samples required per keyword. To bridge this gap, we applied data augmentation techniques to each sample, including:

- Pitch shifting
- Time stretching
- Adding noise
- Adding reverberation
- Adjusting volume

By randomly applying these techniques, we generated 10 augmented samples per speaker, significantly expanding our dataset.

We also observed that the Google Speech Commands dataset includes various background noises, such as: white noise, pink noise, brown noise, human chatter, music, and other environmental sounds (e.g., running water, machinery noise).

To make our dataset as diverse as the Google Speech Commands dataset, we recorded our own environmental sounds using the STM32. These noise recordings were then randomly selected and mixed with our speech command samples at varying signal-to-noise ratios (SNR), ensuring a more robust and realistic dataset.

The two figures below showcase the audio waveforms of an original voice sample and its augmented version, highlighting the differences introduced by the applied augmentation.

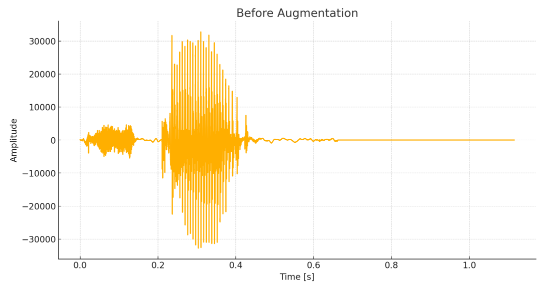


Fig. 1. Waveform of `Start_p257.wav` file.

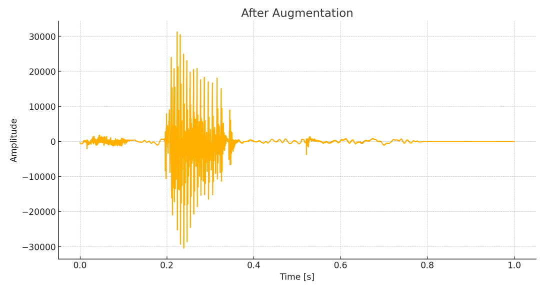


Fig. 2. Waveform of `Start_p257_variation7.wav` file.

B. *Model training process*

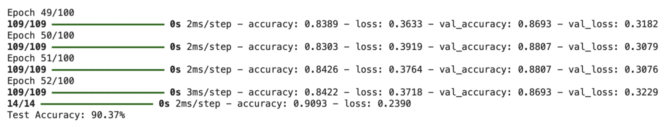
Preprocessing is a crucial step in any machine learning workflow. Proper preprocessing ensures that the model learns effectively and generalizes well to unseen data.In our preprocessing pipeline, we transform raw audio waveforms into a format suitable for machine learning model training by extracting meaningful frequency information.With preprocessing complete, let's turn our attention to the architecture of our model.

1) Input Layer : Our model begins with an Input Layer that accepts audio features represented by a vector of size (129,). These 129 features are derived from the frequency spectrum of the preprocessed audio using Short-Time Fourier Transform (STFT).
2) Regularization with Dropout : Immediately after the input layer, the model applies a Dropout layer with a dropout rate of 30%. Dropout randomly deactivates 30% of the neurons during training, which helps prevent overfitting by forcing the network to learn more robust features instead of relying on specific neurons.
3) Dense Layers (Fully Connected Layers) : The model then passes data through a series of Dense (fully connected) layers, each followed by a Dropout layer to maintain regularization.
    - First Dense Layer (128 neurons, ReLU activation)
    - Dropout Layer (30%)
    - Second Dense Layer (64 neurons, ReLU activation)
    - Dropout Layer (30%)
    - Third Dense Layer (32 neurons, ReLU activation)
    - Dropout Layer (30%)
4) Output Layer (4 neurons, Softmax activation) : The

final layer has 4 neurons, corresponding to the four target classes. A Softmax activation function converts the output into a probability distribution, where each neuron represents the likelihood of the input belonging to one of the four categories.

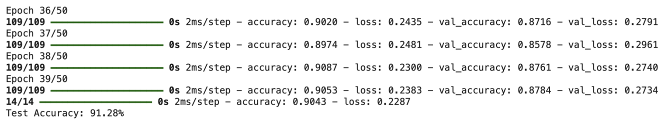
5) Compilation Step : The model is compiled with the following settings:
   - Optimizer: Adam
   - Loss Function: sparse categorical crossentropy
6) Training Process: The model is trained using the fit method with the following key configurations:
   - Epochs: 100 – The model iterates over the dataset up to 100 times.
   - Batch Size: 32 – The dataset is split into batches of 32 samples for each training iteration.
   - Early Stopping Callback: Stops training early if the validation performance does not improve for 5 consecutive epochs and restores the best-performing model weights.
7) Evaluation: After training, the model is evaluated on a separate test dataset to measure its final performance. The evaluation returns the loss and accuracy, providing insight into how well the model generalizes to unseen data.

Our model achieved final test accuracies of 91.28% and 90.37%, which we find satisfactory, especially when the audio samples are recorded in a quiet environment.



```
Epoch 49/100
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8389 - loss: 0.3633 - val_accuracy: 0.8693 - val_loss: 0.3182
Epoch 50/100
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8303 - loss: 0.3919 - val_accuracy: 0.8807 - val_loss: 0.3079
Epoch 51/100
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8426 - loss: 0.3764 - val_accuracy: 0.8807 - val_loss: 0.3076
Epoch 52/100
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8422 - loss: 0.3718 - val_accuracy: 0.8693 - val_loss: 0.3229
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9093 - loss: 0.2390
Test Accuracy: 90.37%
```

Fig. 3. Test accuracy of Genre Spotting model.



```
Epoch 36/50
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9020 - loss: 0.2435 - val_accuracy: 0.8716 - val_loss: 0.2791
Epoch 37/50
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8974 - loss: 0.2481 - val_accuracy: 0.8578 - val_loss: 0.2961
Epoch 38/50
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9087 - loss: 0.2300 - val_accuracy: 0.8761 - val_loss: 0.2740
Epoch 39/50
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9053 - loss: 0.2383 - val_accuracy: 0.8784 - val_loss: 0.2734
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9043 - loss: 0.2287
Test Accuracy: 91.28%
```

Fig. 4. Test accuracy of Command Spotting model.

IV. Raspberry Pi Server

A. *Finite state machine*

The FSM operates in a cyclic flow between two primary states: “Command” and “Genre”. It begins in the “Genre” state, where it prepares or manages genre-specific music operations. Once the STM32 sends data to the Raspberry Pi (RPI) and the ML model successfully identifies the genre, the FSM transitions to the “Command” state.

In the “Command” state, the system listens for incoming instructions from the TCP client, determining the appropriate next steps. If the ML model interprets a command as “Choose”, the FSM seamlessly transitions back to the “Genre” state to execute the instruction or carry out genre-related tasks.

This cyclic exchange between receiving commands and performing tasks ensures a smooth, efficient, and well-organized operational flow, enabling the system to respond dynamically to user inputs while maintaining clarity in task execution.

B. *YouTube playback*

Without downloading the file, yt_dlp extracts a direct audio stream URL from YouTube links, which are thoughtfully organized into genre-specific playlists. Each playlist represents a unique musical category, such as classical, pop, or rock. To prepare for playback, the system preloads a VLC player with the first track from each playlist.

These players are then arranged into a player list, functioning as a control hub for the FSM. This setup allows the system to retain the playback state for each playlist. When switching between genres, the player can seamlessly resume from where it last paused, ensuring a smooth transition without losing track of progress. This design creates a responsive and uninterrupted listening experience across different playlists.