Deep Learning Based Respiratory Sound Classifier

Aditya Dasari
2023101051
IIIT Hyderabad

Sudarshan Nikhil
2023101094
IIIT Hyderabad

Jai Aakash Gopalakrishnan
2023101098
IIIT Hyderabad

Aditya Nair
2023111029
IIIT Hyderabad

***Abstract*—Respiratory diseases are among the leading causes of mortality worldwide, requiring effective diagnostic methods. Traditional auscultation techniques rely on subjective assessment by trained professionals and are prone to errors. This work proposes a portable system for detecting and classifying lung sounds using ESP32, a digital microphone, and a convolutional neural network (CNN). Lung sounds captured with the ESP32 and INMP-441 microphone are preprocessed through filtering techniques to extract clean respiratory audio. The resulting sounds are classified into categories such as wheeze, stridor, rhonchi, and crackles using a pre-trained CNN model, achieving reliable results for real-time diagnostics.**

I. Introduction

Respiratory diseases such as asthma, bronchitis, and pneumonia significantly impact global health and socio-economic conditions. Prompt diagnosis is critical to preventing irreversible complications. While auscultation remains a widely used method, its reliance on human expertise introduces subjectivity and variability. This research develops a system to automate lung sound detection, leveraging machine learning to classify sounds with high accuracy, sensitivity, and specificity.

Our system integrates audio capture, preprocessing, and classification on a portable platform. The primary contributions of this work are:

- A cost-effective, portable solution for lung sound classification.
- Implementation of filtering techniques to improve signal quality.
- Deployment of a CNN trained on Mel spectrograms to achieve state-of-the-art classification accuracy.

II. Related Work

Automated lung auscultation has been explored in previous studies using machine learning techniques. Traditional methods relied on support vector machines (SVMs) or k-nearest neighbors (KNNs) for classifying adventitious respiratory sounds. Recent advancements have introduced convolutional neural networks (CNNs) for feature extraction from spectrograms. However, these models often require high computational power, limiting their portability. This study builds on these works by integrating lightweight hardware and robust models into a single portable solution.

III. System Overview

A. Hardware Setup

1) Components Used: The hardware setup consists of:

- **ESP32**: A low-power microcontroller that processes and transmits audio.
- **INMP-441 Microphone**: A digital MEMS microphone providing high-quality mono audio.
- **Stethoscope**: Used for precise audio localization of respiratory sounds.

2) Challenges and Integration: The integration of the stethoscope and INMP-441 microphone posed significant challenges. Initial attempts to place the microphone near the stethoscope's membrane resulted in substantial sound loss and noise interference. After several iterations, we devised a solution:

- **Modified Stethoscope Membrane**: By carefully creating holes in the membrane aligned with the INMP-441's pin structure, we securely embedded the microphone within the stethoscope.
- **Enhanced Recording Clarity**: This setup minimized sound loss, providing clear recordings of lung and heart sounds while preserving the stethoscope's acoustic amplification properties.

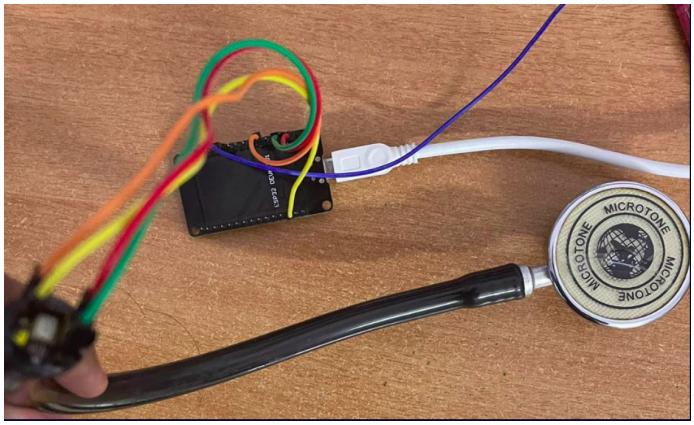

Fig. 1. Stethescope and INMP-441 Integration

B. Software Workflow

1) Audio Capture and Transmission: The ESP32 records audio at a 16 kHz sampling rate, capturing chunks of 1-second duration. The audio is transmitted to a Flask server over HTTP, where it is appended into a .raw file.

2) Conversion and Preprocessing: The .raw file is converted into a .wav file using the `convert_raw_to_wav.py` script. This process ensures compatibility with filtering and classification stages.

3) Heart Sound Filtering: Heart sounds, ambient noise, and artifacts are filtered using Gaussian filtering, parametric equalization, and bandpass filters. The processed audio contains only the desired lung sounds.

4) Feature Extraction: The cleaned audio is transformed into a Mel spectrogram using Librosa. Spectrograms provide a visual representation of audio signals across time and frequency, crucial for effective classification by CNN models.

5) Classification: The spectrograms are input into a CNN trained on the HF Lung V1 dataset. The model predicts respiratory conditions, categorizing sounds into normal, wheeze, stridor, rhonchi, or crackles.

IV. Audio Acquisition

Using the ESP32 and INMP-441 microphone, mono audio data is captured at 16 kHz. Each chunk of 1-second duration is streamed to a Flask server for further processing.

We decided to transmit audio in 1-second chunks for several key reasons tied to the limitations of the ESP32 microcontroller and the need for efficient real-time processing:

- **Memory Constraints:** The ESP32 has limited RAM (approximately 520 KB), which makes it challenging to buffer large amounts of audio data. If we were to send continuous audio, the system would quickly run out of memory. By splitting the data into 1-second chunks, we ensure that the microcontroller's memory capacity isn't exceeded, allowing it to handle each chunk independently without performance issues.
- **Processing Power:** The ESP32's processing capabilities are limited compared to more powerful systems. Sending smaller chunks of audio reduces the computational load. Each 1-second chunk is easier to process and transmit, ensuring that the system can handle the real-time requirements of lung sound detection.
- **Bandwidth Efficiency:** Transmitting large, continuous audio streams could overwhelm the available Wi-Fi bandwidth, especially in environments with varying network conditions. Sending smaller chunks of 1-second audio helps mitigate this issue by ensuring that the transmission is more stable and less prone to packet loss or delays.
- **Real-Time Performance:** By sending 1-second chunks, the system is able to process and classify each segment independently, reducing overall latency. This setup ensures continuous, real-time audio processing without significant delays between recording and classification.

This approach allows the ESP32 to function within its constraints while providing a responsive and stable system for lung sound detection.

V. Heart Sound Filtering

A. Gaussian Filtering

Purpose: The initial step in the filtering process is to reduce high-frequency noise and smooth out the signal.
Method: The `gaussian_filter` function applies a Gaussian filter to the audio data. A Gaussian filter is a low-pass filter that smooths the signal by averaging nearby values with a weight determined by the Gaussian distribution.
Parameters:

- `sigma = 25`: This defines the standard deviation of the Gaussian distribution. A higher sigma smooths the signal more, removing higher-frequency noise at the cost of potentially distorting finer details in the audio.
- `truncate = 6`: This parameter controls the extent of the filter's kernel. It specifies how many standard deviations from the mean to consider when creating the filter. A value of 6 ensures that the filter captures a significant portion of the signal's characteristics.

Effect: This method helps to smooth out rapid fluctuations in the signal, typically caused by high-frequency noise and irrelevant artifacts, while preserving the main features of the lung sounds.

B. Parametric Equalization (Parametric EQ)

Purpose: After Gaussian filtering, parametric equalization is applied to adjust specific frequency bands, improving the clarity of lung sounds and further suppressing unwanted frequencies.
Method: The `ParametricEQ` class from the yodel package is used to apply an equalizer with adjustable frequency bands. Parametric equalizers allow us to modify the amplitude of selected frequency bands without affecting other parts of the audio spectrum.
Parameters:

- `Number of Bands (num_bands)`: This specifies how many frequency bands we want to modify. In this case, two bands are selected.
- `Band Frequencies (band)`: The chosen frequency bands are 75 Hz and 241 Hz. These frequencies are selected based on the characteristics of the lung sounds we are trying to emphasize and the heart sound frequencies we aim to reduce.
- `Q Factor (Qfactor)`: This parameter controls the width of the frequency band affected by the equalizer. A lower Q-factor (e.g., 0.40) indicates a wider band, which means more frequencies will be affected by the equalizer.
- `Gain (db_gain)`: The db_gain values specify how much to boost or cut each frequency band. In this case, the 75 Hz band is boosted by 16 dB to highlight important lung sound frequencies, while the 241 Hz band is cut by 24 dB to suppress certain frequencies that could interfere with lung sound clarity.

Effect: This equalization step is used to amplify desired lung sound frequencies while attenuating heart sound frequencies, improving the overall quality and distinguishability of lung sounds from heartbeats.

C. Bandpass Filtering (Highpass + Lowpass)

Purpose: The bandpass filter is designed to isolate the relevant frequency range for lung sounds, typically between 50 Hz and 270 Hz. This is done by first removing low-frequency noise (heart sounds) and high-frequency noise (ambient or

external interference).

Method: The bandpass filter is created by combining two filters: a highpass filter to remove low-frequency components (such as heartbeats) and a lowpass filter to remove high-frequency noise.

Highpass Filter:

- `Cutoff Frequency`: 50 Hz is chosen as the low-frequency cutoff to remove unwanted low-frequency noise, including heartbeats, which typically occur in lower frequency ranges.
- `Order`: l_order = 6 determines the steepness of the filter's cutoff. A higher order results in a sharper transition between the passband (allowed frequencies) and the stopband (filtered frequencies).

Lowpass Filter:

- `Cutoff Frequency`: 270 Hz is selected as the high-frequency cutoff to suppress any high-frequency noise that falls above the target frequency range for lung sounds.
- `Order`: h_order = 12 indicates a relatively steep filter, ensuring that frequencies above 270 Hz are attenuated efficiently.

Effect: The combination of these filters creates a narrower frequency band that isolates the desired lung sound frequencies while attenuating both heart sounds and high-frequency noise.

D. Normalization and Amplification

Purpose: After filtering, the audio signal is typically left with reduced amplitude due to the filtering process. Normalization is applied to amplify the audio to its original range while preventing distortion.

Method: The `normalize_int16` function ensures that the filtered audio is scaled back into the valid range of 16-bit signed integers (-32768 to 32767), which is standard for WAV audio formats.

Effect: This step ensures that the final audio output has a full dynamic range without introducing clipping or distortion.

VI. Lung Sound Filtering

In this system, lung sound filtering plays a crucial role in improving the accuracy of respiratory sound classification by removing unwanted heart sounds and ambient noise. The goal is to extract the lung sounds clearly and without interference, making them suitable for subsequent analysis by the CNN model. Below is a detailed breakdown of the techniques used for filtering lung sounds in this pipeline:

A. Gaussian Filtering for Lung Sounds

Purpose: The first step in the filtering process is to smooth the lung sound signal and reduce high-frequency noise. However, unlike the heart sound filtering, this Gaussian filtering is applied more gently (using a milder sigma value) to avoid distorting the key features of the lung sounds.

Method: The `gaussian_filter` function from the audio processing module is used to apply a Gaussian filter to the audio data. A smaller sigma value (`sigma_lung = 5`) ensures that the filter has a light touch, removing high-frequency noise but preserving the essential details of the lung sounds.

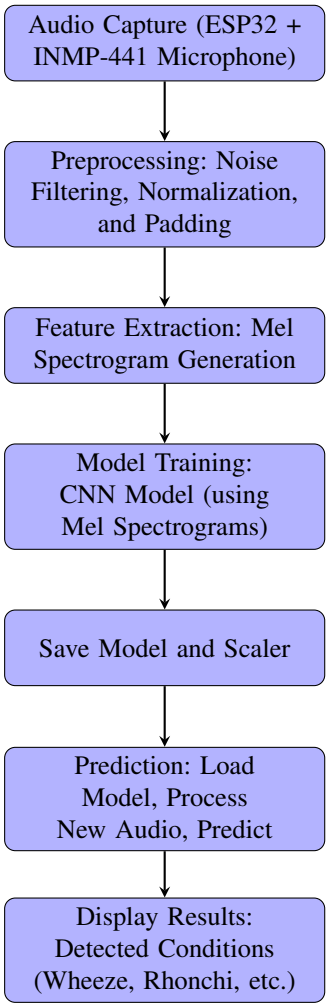

Fig. 2. Lung Sound Classification Workflow

Effect: This gentle filtering smooths out minor fluctuations or sharp noises, which could be caused by ambient interference or other non-respiratory sounds that aren't crucial for lung sound classification.

B. Bandpass Filtering to Focus on Lung Frequencies

Purpose: After smoothing the audio, a bandpass filter is applied to focus on the frequencies most relevant to lung sounds, typically in the 100 Hz to 2000 Hz range. This helps to attenuate both low-frequency heart sounds and high-frequency noise that do not belong to the lung sound spectrum.

Method:

- **Bandpass Filter Parameters:** The bandpass filter is designed to allow frequencies between 100 Hz and 2000

Hz to pass through while attenuating lower and higher frequencies.

- **Butterworth Filter:** The `butter` function from `scipy.signal` is used to create a Butterworth bandpass filter with a cutoff frequency range of 100 Hz to 2000 Hz and an order of 6. The higher the order, the sharper the transition between the passband and stopband, resulting in a more selective filter.
- **Application:** The `filtfilt` function is used to apply the filter to the Gaussian-smoothed lung sound data.

Effect: The bandpass filter isolates the most important frequency range for lung sounds, ensuring that both heart sounds (which typically occur below 100 Hz) and high-frequency interference (such as ambient noise) are reduced, leaving mostly relevant lung sounds in the desired frequency range.

C. Heart Sound Extraction Using Heart Filtering

Purpose: One of the most critical steps in isolating lung sounds is removing the heart sounds, as they can significantly interfere with the clarity of respiratory sounds. This is particularly challenging because heart sounds share some frequency overlap with lung sounds.

Method: To extract heart sounds, we use a combination of Gaussian filtering and parametric equalization (as described in the heart filtering process). These steps are designed to identify and isolate the frequency characteristics of heart sounds.

- **Gaussian Filtering:** A stronger Gaussian filter is applied to the original audio data to highlight the lower-frequency components associated with heartbeats.
- **Parametric Equalization:** The parametric equalizer is configured to enhance frequencies typically associated with heartbeats (around 30 Hz to 100 Hz), boosting their presence while leaving the lung sounds relatively unchanged.

Effect: This method allows us to extract the heart sounds from the original audio by focusing on the frequency range that is characteristic of heartbeats, thus separating it from the lung sounds.

D. Subtraction of Heart Sounds from Lung Audio

Purpose: After isolating the heart sounds, the next step is to subtract them from the lung sound signal, leaving behind only the lung sounds.

Method: The extracted heart sounds are subtracted from the bandpass-filtered lung signal. This works because the heart sounds are typically present in a specific frequency range, which can be effectively isolated and removed using the previously applied filters.

Effect: By subtracting the heart sounds from the lung signal, we eliminate interference caused by heartbeats, ensuring that the lung sounds are isolated more clearly for further classification.

E. Why Other Techniques Were Rejected

Noise Gating: One of the methods considered for separating heart sounds from lung sounds was noise gating, which works by setting a threshold that distinguishes background noise from actual signals. However, we found that this approach did not perform well in this context because heart sounds and lung sounds are often of similar amplitude, making it difficult to differentiate between the two using amplitude-based thresholds alone.

Limitations:

- **Amplitude Similarity:** The heart sound and lung sound amplitudes can overlap significantly, especially in certain frequencies. This makes noise gating unreliable, as it would fail to cleanly separate the heart sounds without cutting into the lung sounds.
- **Performance:** In practice, the use of noise gating resulted in inconsistent performance, as it couldn't properly isolate the heart sounds in all cases, leading to suboptimal results.

Amplitude as a Measure for Filtering: Another approach we considered was using amplitude measurements to filter heart sounds. However, we discarded this idea because amplitude-based filtering is not always effective when the heart sounds are close in amplitude to the lung sounds. The complex overlap in the frequency spectra of heart and lung sounds means that purely amplitude-based approaches would either leave heart sounds intact or remove important lung sound components, making them unsuitable for reliable heart sound subtraction.

VII. Feature Extraction

Feature extraction is a crucial step in transforming raw audio data into a format suitable for classification by machine learning models. In this project, Mel spectrograms are used as the primary feature for classifying lung sounds. Mel spectrograms offer a time-frequency representation of the audio, capturing both the spectral content and temporal variations that are essential for distinguishing between different types of respiratory sounds.

A. Mel Spectrograms

The Mel scale is used to represent the frequencies of the audio in a way that approximates human auditory perception, with a higher resolution in the lower frequencies and less emphasis on the higher frequencies. This makes it ideal for analyzing lung sounds, which are typically concentrated in the lower to mid-frequency ranges. The Mel spectrograms allow the system to efficiently capture important features while minimizing computational complexity.

B. Generation of Mel Spectrograms

The Mel spectrograms are generated through the following steps:

1) **Fourier Transform:** The raw audio data is first transformed from the time domain into the frequency domain

using the Fast Fourier Transform (FFT). This process breaks the signal into its constituent frequencies, allowing us to observe how the frequency content of the audio changes over time.

2) **Mel Scale Transformation:** After obtaining the frequency spectrum, the frequencies are mapped onto the Mel scale, which is designed to reflect the non-linear way humans perceive sound. This transformation focuses on the frequencies most relevant to lung sounds.
3) **Logarithmic Scaling:** The amplitude of each frequency bin is converted to decibels (dB) to make the spectrogram more perceptually relevant, as the human ear responds to sound intensity logarithmically.
4) **Windowing:** The audio is divided into short overlapping segments (frames), and each frame is transformed into a spectrogram. This allows the system to capture the evolving characteristics of the sound over time.

The resulting Mel spectrogram captures the essential features of lung sounds, making it well-suited for classification tasks. These spectrograms are then fed into the convolutional neural network (CNN) for further analysis and classification.

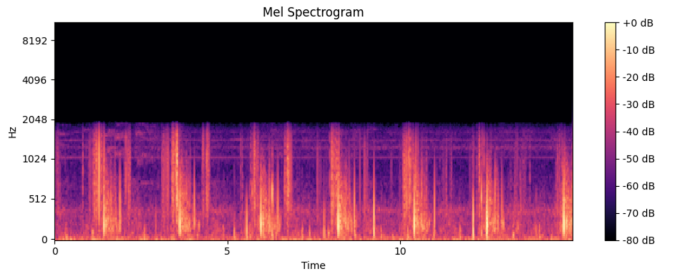

Fig. 3. Mel Spectrogram for Crackles

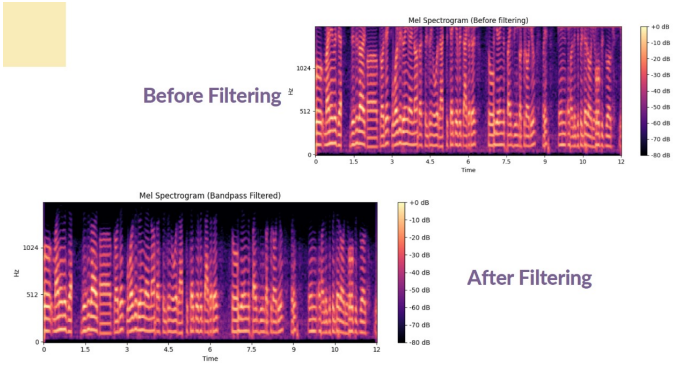

Fig. 4. Sample Mel Spectrogram Before and After Filtering

VIII. Lung Sound Classification and Prediction

The process of classifying lung sounds involves training a deep learning model and using it to make predictions on new audio samples. The following sections outline the steps involved in the training and prediction pipeline, which includes feature extraction, model training, and prediction.

A. Data Preprocessing and Feature Extraction

Before training the model, the raw audio data is processed into features that can be effectively used by the machine learning model. The primary feature extracted from the audio is the Mel spectrogram, which is a time-frequency representation of the audio signal.

1) Mel Spectrogram Generation: The `audio_to_mel_spectrogram` method in the `LungSoundClassifier` and `LungSoundPredictor` classes uses the Librosa library to generate Mel spectrograms from the raw audio. The steps involved are:

1) **Audio Loading:** The audio is loaded with a fixed duration of 15 seconds. If the audio is shorter than the required duration, it is padded with zeros to ensure consistency in input size.
2) **Fourier Transform:** The audio is transformed into the frequency domain using the Short-Time Fourier Transform (STFT), which splits the audio signal into overlapping frames.
3) **Mel Scale Transformation:** The frequency scale is transformed from linear to the Mel scale to better match human auditory perception. The Mel scale emphasizes lower frequencies, which are more important for detecting respiratory sounds like wheezes or crackles.
4) **Logarithmic Scaling:** The amplitude of each frequency bin is converted to decibels (dB) to make the spectrogram more perceptually relevant.
5) **Spectrogram Visualization:** The generated Mel spectrogram is plotted for visual inspection, with time on the x-axis, Mel frequency bands on the y-axis, and color intensity representing the amplitude of the frequencies.

B. Model Architecture and Training

Once the features are extracted, they are used to train a Convolutional Neural Network (CNN) for classification. The model architecture consists of several layers designed to learn spatial features from the Mel spectrogram images.

1) Model Architecture: The model is built using the Sequential API from Keras and includes the following key layers:

- **Convolutional Layers:** These layers (`Conv2D`) are responsible for learning spatial patterns from the Mel spectrogram images. The first convolutional layer has 32 filters, followed by 64 and 128 filters in subsequent layers. Each convolutional layer uses ReLU (Rectified Linear Unit) activation to introduce non-linearity.
- **Max-Pooling Layers:** After each convolutional layer, a max-pooling layer is applied to reduce the spatial dimensions of the spectrogram and highlight the most important features.
- **Dropout Layers:** Dropout regularization is used after each pooling layer to prevent overfitting. The dropout rate is set to 25% for most layers, and 50% for the fully connected layers.
- **Fully Connected Layers:** After the convolutional and pooling layers, the model flattens the output and passes

it through a fully connected layer with 256 units and ReLU activation.
- **Output Layer:** The final layer uses sigmoid activation to output probabilities for each of the four classes: wheeze, stridor, rhonchi, and crackles. The model predicts whether each of these conditions is present or not, using a binary cross-entropy loss function.

2) Model Training: During training, the following steps are carried out:

- **Data Loading:** The training and testing datasets are loaded and preprocessed. Each audio file is converted to a Mel spectrogram, and its corresponding label is retrieved from a text file containing the class labels.
- **Normalization:** The Mel spectrograms are flattened into 1D arrays and normalized using a StandardScaler. This ensures that the model receives inputs with a consistent range of values, which helps with convergence during training.
- **Training:** The model is trained for 20 epochs with a batch size of 32. The training process includes validation on a separate test dataset to monitor the model's performance.
- **Model Saving:** After training, the model weights are saved to a file (`lung_sound_classifier.h5`), and the scaler is saved to another file (`scaler.pkl`). These files are used for making predictions on new data.

The dataset used for training the model, the *HF_Lung_V1 Dataset*, is available online [2].

C. Prediction on New Data

Once the model is trained, it can be used to classify lung sounds from new audio recordings. The `LungSoundPredictor` class handles the prediction process.

1) Predicting with the Trained Model: The `predict` method performs the following steps to predict the condition of the lung sounds:

- **Audio Preprocessing:** The audio file is loaded and converted into a Mel spectrogram using the same process as in training. This ensures that the input is consistent with the features the model was trained on.
- **Feature Normalization:** The Mel spectrogram is flattened into a 1D array and then normalized using the scaler that was saved during training. This step ensures that the input data is scaled appropriately before being passed into the model.
- **Model Prediction:** The model processes the normalized Mel spectrogram, and outputs probabilities for each class. The probabilities indicate the likelihood that the given audio corresponds to each of the possible lung conditions (wheeze, stridor, rhonchi, crackles).
- **Thresholding:** A threshold of 0.5 is applied to the output probabilities. If a class probability is greater than or equal to 0.5, that class is considered to be present in the audio. If no class meets this threshold, the system predicts that the audio is normal.

2) Prediction Example: For example, when given an audio file (`steth_20190626_15_12_23.wav`), the model will output a list of detected conditions, such as:

```
Detected conditions: ['wheeze', 'rhonchi']
```

This means that the model has classified the audio as containing both wheeze and rhonchi sounds.

D. Model Evaluation and Performance

The model's performance is evaluated using standard classification metrics such as accuracy, precision, recall, and F1 score. The use of multi-label classification allows the model to simultaneously predict multiple conditions (e.g., both wheeze and rhonchi), which is important for diagnosing respiratory diseases that may present with more than one symptom.

E. Conclusion

The lung sound classification pipeline uses deep learning techniques to automate the process of diagnosing respiratory conditions from acoustic signals. By employing a Convolutional Neural Network (CNN), the model can effectively classify different lung sound conditions based on their Mel spectrogram representations. The use of preprocessing steps like Mel spectrogram generation and normalization ensures that the model can work effectively on raw audio data, providing a robust solution for real-time lung sound analysis.

IX. Results

A. Performance Metrics

X. Challenges Faced

A. Hardware Integration

Issue: Integrating the INMP-441 microphone with the stethoscope resulted in significant sound loss and poor audio quality.
Solution: Modified the stethoscope membrane by puncturing holes to align with the microphone pins, improving sound capture clarity.

B. Real-time Processing

Issue: Processing continuous audio in real-time was challenging due to the ESP32's limited memory and processing power.
Solution: Transmitted audio in 1-second chunks to ensure stable processing and transmission, avoiding memory overload and latency issues.

C. Audio Quality and Noise Handling

Issue: Heart sounds interfered with lung sound classification, and noise gating methods were ineffective due to similar amplitudes.
Solution: Employed advanced filtering techniques (Gaussian filtering, parametric equalization, and bandpass filtering) to isolate lung sounds and reduce interference.

XI. Code

```
#include <HTTPClient.h>
#include <WiFi.h>

#include "driver/i2s.h" // Include I2S driver

// Wi-Fi credentials
const char* ssid = "Sudarshan M34";
const char* password = "Warriors";

// Server URL
const char* serverURL = "http
    ://192.168.216.221:5000/upload";

unsigned long prev_time;

// I2S pins
#define I2S_SCK 14 // Bit clock (BCLK)
#define I2S_SD 32 // Serial data (DOUT)
#define I2S_WS 15 // Word select (LRCL)

// Configuration for 1-second audio chunk
#define SAMPLE_RATE 16000 // Set sample rate to 16
    kHz for the INMP441
#define CHUNK_DURATION 1 // 1 second duration
#define CHUNK_SIZE (SAMPLE_RATE * CHUNK_DURATION) //
     Number of samples for 1 second

#define SEC2MILLIS 1000

int16_t audioBuffer[CHUNK_SIZE]; // Buffer to store
    1 second of audio

void setup() {
    Serial.begin(115200);
    WiFi.begin(ssid, password);

    // Wait for Wi-Fi connection
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.println("Connecting to WiFi...");
    }
    Serial.println("Connected to WiFi");

    // I2S configuration for INMP441
    i2s_config_t i2s_config = {
        .mode = i2s_mode_t(I2S_MODE_MASTER |
            I2S_MODE_RX), // Master mode, receive
        .sample_rate = SAMPLE_RATE, // 16kHz sample
            rate
        .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT
            , // 16-bit per sample
        .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
             // Mono input (only left channel)
        .communication_format = i2s_comm_format_t(
            I2S_COMM_FORMAT_I2S), // Standard I2S
            format
        .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1, //
             Interrupt level 1
        .dma_buf_count = 8, // Number of DMA buffers
        .dma_buf_len = 1024, // Length of each DMA
            buffer
        .use_apll = false, // Not using APLL
    };

    // I2S pin configuration
    i2s_pin_config_t pin_config = {
        .bck_io_num = I2S_SCK, // BCLK pin
        .ws_io_num = I2S_WS, // LRCL pin
        .data_out_num = I2S_PIN_NO_CHANGE, // No
            data output
        .data_in_num = I2S_SD // DOUT pin (serial
            data input)
    };

    // Install and start the I2S driver
    i2s_driver_install(I2S_NUM_0, &i2s_config, 0,
        NULL);
    i2s_set_pin(I2S_NUM_0, &pin_config);

    // Set the I2S clock
    i2s_set_clk(I2S_NUM_0, SAMPLE_RATE,
        I2S_BITS_PER_SAMPLE_16BIT, I2S_CHANNEL_MONO)
        ;

    prev_time = millis();
}

void loop() {
    unsigned long curr_time = millis();

    if ((curr_time - prev_time) > (CHUNK_DURATION *
        SEC2MILLIS)) {
      prev_time = curr_time;

      size_t bytes_read;

      // Capture 1 second of audio data from the
          INMP441 microphone
      i2s_read(I2S_NUM_0, (void*)audioBuffer, sizeof
          (audioBuffer), &bytes_read, portMAX_DELAY)
          ;

      Serial.printf("Read %d bytes of audio.\n",
          bytes_read);

      // Send the 1-second audio data via HTTP
      HTTPClient http;
      http.begin(serverURL);
      http.addHeader("Content-Type", "application/
          octet-stream");

      // Convert buffer to bytes and send
      uint8_t* byteData = (uint8_t*)audioBuffer;
      int httpResponseCode = http.POST(byteData,
          bytes_read);


      if (httpResponseCode > 0) {
          Serial.printf("Data sent, response code: %
              d\n", httpResponseCode);
      }

      else {
          Serial.printf("Error in sending data: %s\n
              ", http.errorToString(httpResponseCode
              ).c_str());
      }

      http.end(); // End the connection
    }

    // delay(1000); // Wait 1 second before
        capturing and sending again
}
```

Listing 1. ESP-to-Local Audio Recording

```
import numpy as np
from scipy.signal import butter, filtfilt
from yodel import filter as ft

import audio_processing as ap

def extract_heart_audio(audio_data, sample_rate):
    """
```

```
    Extracts heart sounds from the given audio data
        using Gaussian filtering,
    parametric equalization, and bandpass filtering.

    Parameters:
        audio_data (numpy.ndarray): The input audio
            data array.
        sample_rate (int): The sample rate of the
            audio data.

    Returns:
        numpy.ndarray: The extracted heart sounds
            without normalization.
    """
    # Apply Gaussian filter to smooth the audio
        signal
    sigma = 25
    gauss_audio = ap.gaussian_filter(audio_data,
        sigma, truncate=6)

    # Parametric Equalization to enhance heart sound
         frequencies
    num_bands = 2
    band = [75, 241]
    Qfactor = [0.40, 0.40]
    db_gain = [16.0, -24.0]

    parameq = ft.ParametricEQ(sample_rate, num_bands
        )
    parameq.set_band(0, band[0], Qfactor[0], db_gain
        [0])
    parameq.set_band(1, band[1], Qfactor[1], db_gain
        [1])
    parameq_audio = np.empty_like(gauss_audio)
    parameq.process(gauss_audio, parameq_audio)

    # Highpass filter to remove frequencies below
        heart sounds
    lowcutoff = 50
    l_order = 6
    bl, al = butter(l_order, lowcutoff, btype='
        highpass', fs=sample_rate)
    parameq_hp_audio = filtfilt(bl, al,
        parameq_audio)

    # Lowpass filter to remove frequencies above
        heart sounds
    highcutoff = 270
    h_order = 12
    bh, ah = butter(h_order, highcutoff, btype='
        lowpass', fs=sample_rate)
    parameq_bp_audio = filtfilt(bh, ah,
        parameq_hp_audio)

    # Return the extracted heart sounds without
        normalization
    heart_audio = parameq_bp_audio

    return heart_audio

def extract_heart_and_normalize(audio_data,
    sample_rate):
    """
    Extracts heart sounds from the input WAV file,
        normalizes them,
    and saves them to the output WAV file.

    Parameters:
        input_filename (str): The path to the input
            WAV file.
        output_filename (str): The path to the
            output WAV file.
    """
    # Load the audio recording
    # audio_data, sample_rate = ap.read_wav_audio(
        input_filename)

    # Extract heart sounds
    heart_audio = extract_heart_audio(audio_data,
        sample_rate)

    # Normalize the heart audio
    normalized_heart_audio = ap.normalize_int16(
        heart_audio)

    # Save the normalized heart audio to a WAV file
    # ap.save_wav_audio(normalized_heart_audio,
        sample_rate, output_filename)

    return normalized_heart_audio
```

Listing 2. Heart Filtering

```
import heart_filtering as hf
import numpy as np
from scipy.signal import butter, filtfilt

import audio_processing as ap

def lung_filter(audio_data, sample_rate):
    """
    Processes the lung audio data by applying
        Gaussian filtering,
    bandpass filtering, and heart sound subtraction.

    Parameters:
        audio_data (numpy.ndarray): The input audio
            data array.
        sample_rate (int): The sample rate of the
            audio data.

    Returns:
        numpy.ndarray: The cleaned lung sound audio
            data.
    """
    # Apply a lighter Gaussian filter to the lung
        sounds
    sigma_lung = 5  # Lighter sigma for lung sounds
    gauss_lung_audio = ap.gaussian_filter(audio_data
        , sigma=sigma_lung, truncate=6)

    # Filter ambient noise to focus on lung sounds
        (100 Hz to 2000 Hz)
    lowcut = 100
    highcut = 2000
    order = 6
    b, a = butter(order, [lowcut, highcut], btype='
        bandpass', fs=sample_rate)
    lung_bandpassed = filtfilt(b, a,
        gauss_lung_audio)

    # Extract heart sounds from the original audio
        data
    heart_sounds = hf.extract_heart_audio(audio_data
        , sample_rate)

    # Subtract the heart sounds from the lung
        bandpassed signal
    cleaned_lung = lung_bandpassed - heart_sounds

    return cleaned_lung
```

Listing 3. Lung Filtering

GitHub Repository

The complete code for this project is available at the following GitHub repository: DL-Respiratory-Sound-Classifier.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our professor, Abhishek Srivastava, for his invaluable guidance, encouragement, and support throughout the course of this project. His expertise and insights have been instrumental in shaping our work.

We also extend our heartfelt thanks to our teaching assistants, Ms. Santhoshini and Mr. Srikar, for their constant assistance and feedback, which greatly enhanced the quality of our work.

CONTRIBUTIONS

- Aditya Nair worked on the workflow and designed the heart filtering process, and assisted in fine-tuning of lung filtering, using various filtering methods.
- Jai Aakash worked on hardware integration, focusing on integrating the stethoscope and microphone to ensure effective audio capture, and creating and testing the lung filter workflow.
- Sudarshan Nikhil worked on capturing audio from the microphone and ESP32, and along with Aditya Dasari performed preliminary filtering to determine the appropriate filtering techniques for heart and lung sounds.
- Aditya Dasari was responsible for training and testing the CNN model for classifying lung sounds. Sudarshan Nikhil implemented a GUI for easy use of the model, and assisted with fine-tuning the model to improve its performance.

REFERENCES

[1] A. S. Edakkadan and A. Srivastava, "Deep Learning Based Portable Respiratory Sound Classification System," *2023 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 129-133, DOI: 10.1109/APCCAS60141.2023.00039, 2023.

[2] TechSupportHF, "`HF_Lung_V1 Dataset`" Available: https://gitlab.com/techsupportHF/HF_Lung_V1.