CEN598 - Final Project - Wind Speed Estimation Using Closely Spaced Microphones

Sarwan Shah[1]

Abstract— **This project aimed to create a machine learning (ML) model that is capable of estimating the wind speed based on the audio data from two closely spaced microphones. The project also demonstrates the ability to create an experimental setup for gathering and processing audio data.**

INTRODUCTION

It was shown in [3][7] that when two microphones are placed closely together and wind strikes the structure of the microphone it leads to the generation of turbulent flow that is captured as wind noise at frequencies below 50 Hz. This noise can be correlated between the audio data of the two microphones and this correlation can be used to provide an estimate of the wind speed.

While this correlation can be mathematically estimated using the Corcos model [1], a model for fluid flow, it is nonetheless a simplification and an estimation, as it is difficult to precisely turbulent flow such as the aforementioned with a closed-form mathematical model. Thus, this presents a great opportunity to explore this problem from a machine-learning perspective.

RELATED WORKS

There is only one research paper [7] has explored this approach using machine learning and none of them have explored it from an embedded system's perspective to the best of the author's knowledge.

The hypothesis is that a machine learning model will be able to capture this non-linear relationship caused by wind noise in the two microphones and provide an estimate of the wind speed.

This has larger implications as meteorological data is always sparse and installing new automatic weather stations (AWS) is expensive and comes with logistical challenges. If wind speed, being one of the most important meteorological parameters could be estimated using microphones only then this allows for manufacturing of equipment like the AWS' significantly cheaper and easier to deploy given the smaller footprint. This could help increase the density of the meteorological data globally by making such equipment more accessible, especially in developing countries that lack a strong technological infrastructure while being largely impacted by climate change.

Thus, this sets our motivation for pursuing this project. In the next few sessions, we will go over our system design, our data collection methodology, design choices, and finally our results, after which we will conclude.

SYSTEM DESIGN

Experimental Setup

A significant part of this project was being able to appropriately gather the required experimental data. In this section, we explore the design choices made to ensure a robust experimental setup for

Since we are dealing with audio data, which had to sample more than one sensor (at least two microphones), a powerful microprocessor was needed to be able to gather this data. For this purpose, the ESP32 by Espressif Systems was chosen due to its fast processor and big SRAM (320 KB), which enabled sufficient room for sampling multiple sensors and making calculations such as the FFT over large audio samples. Additionally, the ESP32 also comes coupled with I2S peripherals, which are an industry standard for audio-related communication between devices. Thus, these factors made an ESP32 an excellent choice for our project.

The choice of microphone was the INMP441 [10] MEMs microphone which was chosen because of its sensitivity, which would enable the capturing of low-frequency noise. The original goal was to address the people using an array of 4-closely spaced microphones, but it was identified the I2S peripheral on the ESP32 was limited to only one stereo input. Thus, only two microphones - fig. 3 - could be coupled with the ESP32 using a single channel by reading them independently as left and right channels. The configuration for this is shown in the fig. 1.

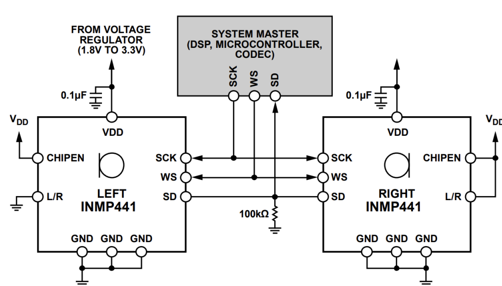

Fig. 1. Left and right channel setup configuration of two INMP441 on the ESP32 [10]

A hall-effect anemometer was used as a control for the experiment to measure speed and provide corresponding labels for the sample and was sampled by maintaining a simple pulse counter on an interrupt. Additionally, a DHT11 sensor was also used to measure the temperature and humidity, with the idea that it could help address sensitivity issues of the microphone, which can for some microphones change with temperature, humidity, and pressure.

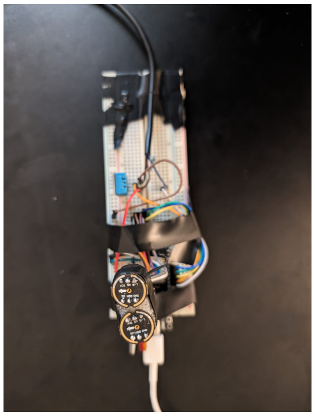

Fig. 2. Microphone array used for capturing wind noise

Lastly, a desk fan with 10 distinct speed settings was used to generate winds with speeds between approximately $\tilde{0}$ to 12 m/s, which were blown across the two-microphone array and the anemometer immediately behind it. See figure **??**.

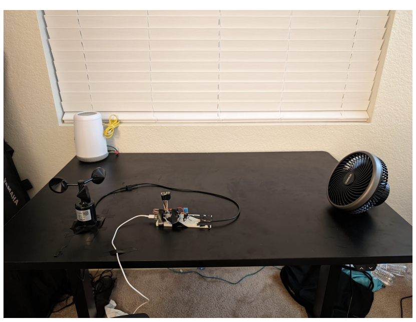

Fig. 3. Microphone array used for capturing wind noise

Data Collection

Using the aforementioned experimental setup 5000 samples of data were gathered. The complete breakdown of samples can be seen in table I. Each sample was one second long (made to match the sampling rate of the hall-effect anemometer) and the data was sampled for 60 seconds at each speed setting. This ensured the creation of a balanced dataset.

The direction of the wind was also varied by varying the position of the microphones while keeping the anemometer and fan in a fixed position. The goal is the possibility to explore whether directional information could be extracted from the audio data or not.

Direction	Samples
North	1600
North East	600
North West	600
East	600
West	600
North	600
South East	600
South West	600

TABLE I
DATA SAMPLES COLLECTED USING EXPERIMENTAL SETUP

On the ESP32 the audio was sampled at 1 kHz and 16-bit after experimenting with 8 kHz and 16 kHz, but failing to transfer that data to the computer due to limits with the Baud-rate with the ESP32. A Python script was written on the PC using multiple threading to accommodate the stream of incoming data from the ESP32 and the data in comma-separated '.txt' files.

TRAINING & EVALUATION

Before using the data to train a model a few steps around pre-processing the data were performed. These involved normalizing the data using min-max standardization. Furthermore, as a feature engineering step, rather than directly inputting the raw audio data from the two microphones into the model, the data was converted into spectrograms. This is an image-like representation that gives insights into the frequency components present in the signal over time.

Furthermore, it was ensured that the frequency components in the spectrograms be limited to below 51 Hz to reduce chances of background noise or the noise from the fan (approximately 100 Hz) interfering or bias the audio data.

The data was divided into training and test sets. The training data was further divided into training and validation sets which were used for the actual training of the model. These proportions are expressed percentage of the overall data below in table II. It is worth noting that only 'North' directional data samples were used for the linear regression problem. The data gathered in other directions is to address the classification problem for future work.

Data Set	%	Samples
Training	70	1100
Validation	20	300
Testing	11	180

TABLE II
DATA SAMPLES GATHERED

This data split was then used to train and test a convolutional neural network (CNN) model. Multiple iterations were performed in which the layers and weights were modified, added, and removed until the lowest loss and error were achieved. To address issues over over-fitting dropout layers were introduced and the model complexity was kept at a minimum, as in general over-fitting was being experienced. The summary for the model is shown in fig.**??** below.

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resizing_1 (Resizing)       (1200, 32, 32, 2)         0

 normalization_4 (Normaliza  (1200, 32, 32, 2)         5
 tion)

 conv2d_26 (Conv2D)          (1200, 32, 32, 32)        288

 conv2d_27 (Conv2D)          (1200, 32, 32, 64)        8256

 max_pooling2d_15 (MaxPooli  (1200, 16, 16, 64)        0
 ng2D)

 dropout_20 (Dropout)        (1200, 16, 16, 64)        0

 flatten_8 (Flatten)         (1200, 16384)             0

 dense_16 (Dense)            (1200, 32)                524320

 dropout_21 (Dropout)        (1200, 32)                0

 dense_17 (Dense)            (1200, 1)                 33

=================================================================
Total params: 532902 (2.03 MB)
Trainable params: 532897 (2.03 MB)
Non-trainable params: 5 (24.00 Byte)
_________________________________________________________________
```

Fig. 4. Model Summary

While the model seemed to perform some learning, but generally was found to saturate beyond a certain point. The loss and mean absolute error for our training phase are shown in the figures 5 & 6 below.

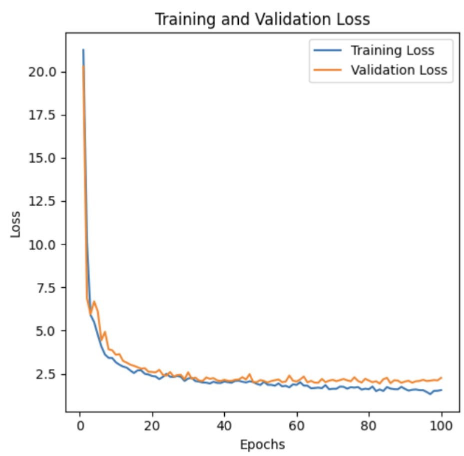

Fig. 5. Training and Validation Loss

Results & Discussion

It can be inferred from the spectrograms in fig. **??** which are for different speeds across the two microphones there is the presence of a difference in the frequency component in the two different cases that is contributing to learning. However, it seems that either there is a lack of data or the need for making these features more available to accommodate better learning with the model and achieve better results.

Given more time, we are confident that exploring aspects of the feature engineering component of this project could allow us to significantly improve the accuracy of our results on a convolutional neural network.

Through this project, we were able to appreciate the effort and investment that goes into data-gathering, especially when working with audio devices on embedded systems, and learned the various considerations that need to be made when choosing sampling rates, bit rates, and a number of devices.

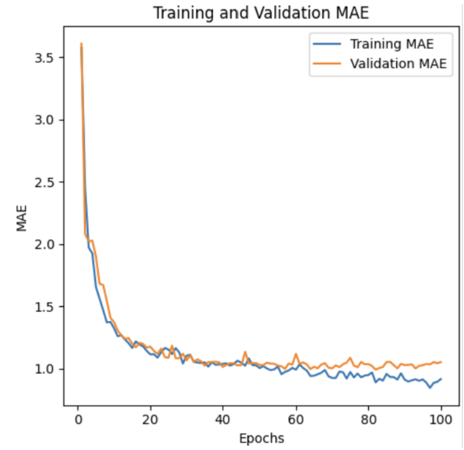

Fig. 6. Training and Validation Mean Absolute Error

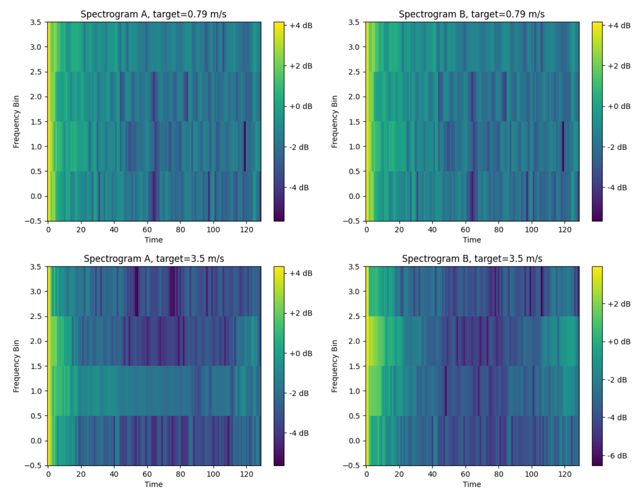

Fig. 7. Caption

We could unfortunately not run inference in the interest of time, as most of our time was consumed by addressing challenges around sampling stereo audio data and training the model.

Please note that the codes for data collection, extraction, and model training can be found in the appendix.

References

[1] Corcos GM. The structure of the turbulent pressure field in boundary-layer flows. Journal of Fluid Mechanics. 1964;18(3):353-378. doi:10.1017/S002211206400026X

[2] Henry E. Bass, Richard Raspet, John O. Messer; Experimental determination of wind speed and direction using a three microphone array. J. Acoust. Soc. Am. 1 January 1995; 97 (1): 695–696. https://doi.org/10.1121/1.412293

[3] F. Douglas Shields; Low-frequency wind noise correlation in microphone arrays. J. Acoust. Soc. Am. 1 June 2005; 117 (6): 3489–3496. https://doi.org/10.1121/1.1879252

[4] C. M. Nelke and P. Vary, "Measurement, analysis and simulation of wind noise signals for mobile communication devices," 2014 14th International Workshop on Acoustic Signal Enhancement (IWAENC), Juan-les-Pins, France, 2014, pp. 327-331, doi: 10.1109/IWAENC.2014.6954312.

[5] Oleg A. Godin, Vladimir G. Irisov, Mikhail I. Charnotskii; Passive acoustic measurements of wind velocity and sound speed in air. J. Acoust. Soc. Am. 1 February 2014; 135 (2): EL68–EL74. https://doi.org/10.1121/1.4862885

[6] Mirabilii, Daniele Habets, Emanuël. (2018). Simulating Multi-Channel Wind Noise Based on the Corcos Model. 10.1109/IWAENC.2018.8521302.

[7] D. Mirabilii, K. K. Lakshminarayana, W. Mack and E. A. P. Habets, "Data-Driven Wind Speed Estimation Using Multiple Microphones," ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Barcelona, Spain, 2020, pp. 576-580, doi: 10.1109/ICASSP40776.2020.9054381.

[8] Daniele Mirabilii, Emanuel A. P. Habets; Pseudo-spectrum based methods for estimating the wind speed and direction based on closely spaced microphone signals. Proc. Mtgs. Acoust. 8 May 2023; 51 (1): 022003. https://doi.org/10.1121/2.0001737
bibitemc9 Features - TDK invensense, https://invensense.tdk.com/wp-content/uploads/2015/02/INMP441.pdf (accessed Dec. 7, 2023).

Appendix I
Data Collection Code

```
#include <driver/i2s.h>
#include <Arduino.h>
#include "DHT_Async.h"

// Define pins for I2S communication
#define I2S_WS 21
#define I2S_SD 22
#define I2S_SCK 23

// Define pins and type for DHT sensor
#define DHT_SENSOR_TYPE DHT_TYPE_11
static const int DHT_SENSOR_PIN = 20;
DHT_Async dht_sensor(DHT_SENSOR_PIN,
    DHT_SENSOR_TYPE);

// Variables for debouncing interrupt
unsigned long lastDebounceTime = 0; // the
    last time the output pin was toggled
unsigned long debounceDelay = 1000; // the
    debounce time; increase if the output
    flickers
int pinInterrupt = 19;
int Count = 0;

// I2S configuration
#define I2S_PORT I2S_NUM_0
#define bufferLen 1024

float temperature_ = 0.0;
float humidity_ = 0.0;
float windSpeed = 0.0;
int sampleNumber = 0;

// Double the buffer size for stereo data
int16_t sBuffer[bufferLen * 2];

// Function to install I2S driver
void i2s_install() {
  const i2s_config_t i2s_config = {
    .mode = i2s_mode_t(I2S_MODE_MASTER |
        I2S_MODE_RX),
    .sample_rate = 1000,
    .bits_per_sample =
        I2S_BITS_PER_SAMPLE_16BIT,
    .channel_format =
        I2S_CHANNEL_FMT_RIGHT_LEFT, // Set to
        LEFT for left channel
    .communication_format =
        I2S_COMM_FORMAT_I2S,
    .intr_alloc_flags = 0,
    .dma_buf_count = 8,
    .dma_buf_len = bufferLen,
    .use_apll = false
  };

  i2s_driver_install(I2S_PORT, &i2s_config,
      0, NULL);
}

// Function to set I2S pins
void i2s_setpin() {
  const i2s_pin_config_t pin_config = {
    .bck_io_num = I2S_SCK,
    .ws_io_num = I2S_WS,
```

```
    .data_out_num = I2S_PIN_NO_CHANGE,
    .data_in_num = I2S_SD
  };

  i2s_set_pin(I2S_PORT, &pin_config);
}

void setup() {
  Serial.begin(115200);
  Serial.println(" ");
  delay(10000);
  pinMode(pinInterrupt, INPUT_PULLUP);// set the interrupt pin
  attachInterrupt(digitalPinToInterrupt(pinInterrupt), onChange, FALLING);

  // Initialize and start I2S
  i2s_install();
  i2s_setpin();
  i2s_start(I2S_PORT);
  delay(100);
  Serial.println("SAMPLE,WIND_SPEED,MIC_A,MIC_B");
  delay(500);
}

void loop() {
  // Debounce and calculate wind speed
  if ((millis() - lastDebounceTime) > debounceDelay) {
    windSpeed = (float)(Count * 8.75)/(float)100.0;
    lastDebounceTime = millis();
    sampleNumber += 1;
    Count = 0;
  }

  // Read from I2S and print data
  size_t bytesIn = 0;
  esp_err_t result = i2s_read(I2S_PORT, &sBuffer, bufferLen * 2, &bytesIn, portMAX_DELAY);

  if (result == ESP_OK) {
    int16_t samples_read = bytesIn / 4; // Each sample is 16 bits (2 bytes)
    if (samples_read > 0) {
      for (int16_t i = 0; i < samples_read; i += 2) {
        Serial.print(sampleNumber);
        Serial.print(",");
        Serial.print(windSpeed);
        Serial.print(",");
        Serial.print(sBuffer[i]);
        Serial.print(",");
        Serial.println(sBuffer[i + 1]);
      }
    }
  }
}

// Function to measure temperature and humidity using DHT sensor
static bool measure_environment(float *temperature, float *humidity) {
    static unsigned long measurement_timestamp = millis();

    // Measure once every four seconds
    if (millis() - measurement_timestamp > 1000ul) {
        if (dht_sensor.measure(temperature, humidity)) {
            measurement_timestamp = millis();
            return (true);
        }
    }

    return (false);
}

// Interrupt handler for pin change
void onChange() {
  if (digitalRead(pinInterrupt) == LOW )
    Count++;
}
```

Appendix II
Data Extraction Code

```
import serial
import time
import threading

start_time = time.time() * 1000

def elapsed_milliseconds():
    return round((time.time() * 1000) - start_time)

def read_serial(serial_port, buffer):
    while True:
        serial_data = serial_port.readline().decode('utf-8').strip()
        timestamped_data = f"{elapsed_milliseconds()},{serial_data}"
        buffer.append(timestamped_data)

def write_to_file(buffer, output_file_path):
    while True:
        time.sleep(1) # Adjust the sleep duration based on your requirements
        with open(output_file_path, 'a') as output_file:
            lines_to_write = buffer[:]
            buffer.clear()
            for line in lines_to_write:
                output_file.write(line + '\n')

if __name__ == "__main__":
    serial_port = serial.Serial('COM6', 115200) # Replace with your actual port and baud rate
    output_file_path = 'North.txt'
    data_buffer = []

    try:
        serial_thread = threading.Thread(target=read_serial, args=(serial_port, data_buffer), daemon=True)
        serial_thread.start()

        write_thread = threading.Thread(target=write_to_file,
```

```
        args=(data_buffer,
        output_file_path), daemon=True)
    write_thread.start()

    serial_thread.join()
    write_thread.join()

except KeyboardInterrupt:
    serial_port.close()
    print('Serial port closed.')

except Exception as e:
    print(f"An error occurred: {e}")

finally:
    serial_port.close()
```

Appendix III
Training, Validation, & Testing Code

```
# -*- coding: utf-8 -*-
"""CEN598-FinalProject.ipynb

Automatically generated by Colaboratory.

Original file is located at
   https://colab.research.google.com/drive/1lT1zN45ZpuKETSb2ZhWOToeRfyG9qJe7
"""

# Import necessary libraries
import pandas as pd
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn.model_selection import
    train_test_split

# Import specific components from TensorFlow
    and Keras
from tensorflow.keras import layers
from tensorflow.keras import models
from keras.models import Model
from tensorflow.keras import Sequential
from keras.layers import Input, Dense,
    Conv2D, MaxPooling2D, Flatten, Dropout,
    Resizing, Normalization, Conv1D

# Define the file path for the data
url = 'North.txt'

# Read the data from the specified CSV file
    into a pandas DataFrame
df = pd.read_csv(url)

# Drop the '4606' column from the DataFrame
df = df.drop(columns=['4606'])

# Display the first few rows of the DataFrame
print(df.head())

# Group the DataFrame by the 'SAMPLE' column
grouped = df.groupby('SAMPLE')

# Extract data groups for each 'SAMPLE' into
    a list
data = [group for name, group in grouped]

# Initialize lists to store targets and
    features
targets = []
features = []

# Iterate over each group of data
for sample_df in data:
    # Extract the 'WIND_SPEED' values and
        calculate the mean
    target = sample_df['WIND_SPEED'].values
    targets.append(np.mean(target))

    # Extract 'MIC_A' and 'MIC_B' values for
        each 'SAMPLE'
    pair = []
    feature_columns = ['MIC_A', 'MIC_B']
    for column in feature_columns:
        pair.append(sample_df[f'{column}'].values)
    features.append(pair)

# Initialize new lists to filter features
    and targets based on length conditions
newFeatures = []
newTargets = []

# Iterate through each set of features
for i in range(len(features)):
    # Check if the lengths of both 'MIC_A'
        and 'MIC_B' are 768
    if (len(features[i][0]) == 768 and
        len(features[i][1]) == 768):
        # If the condition is met, add the
            features and targets to the new
            lists
        newFeatures.append(features[i])
        newTargets.append(targets[i])

# Update the original features and targets
    lists with the filtered data
features = newFeatures
targets = newTargets

# Perform min-max normalization on each
    nested array
min_values = np.min(features, axis=2,
    keepdims=True)
max_values = np.max(features, axis=2,
    keepdims=True)

# Normalize the features using min-max
    normalization
features = (features - min_values) /
    (max_values - min_values)

# Print the maximum value in the targets
    array
print(np.max(targets))

# Initialize lists to store spectrograms for
    each feature column
spectrogramsA = []
spectrogramsB = []

# Function to compute spectrogram from audio
    data
```

```
def compute_spectrogram(audio_data,
    sample_rate=512, max_frequency=51):
   # Convert audio data to a TensorFlow
      constant with float32 data type
   audio_data = tf.constant(audio_data,
      dtype=tf.float32)

   # Compute short-time Fourier transform
      (STFT) of the audio data
   spectrogram = tf.signal.stft(audio_data,
      frame_length=255, frame_step=129)

   # Calculate the magnitude of the
      spectrogram
   spectrogram = tf.abs(spectrogram)

   # Apply logarithmic scaling to the
      spectrogram to enhance features
   spectrogram = tf.math.log(spectrogram +
      1e-9)

   # Add an additional axis to the
      spectrogram to give it tensor-like
      shape
   spectrogram = spectrogram[..., tf.newaxis]

   return spectrogram

# Iterate through each set of features
for feature in features:
   # Define the indices of the feature
      columns
   feature_columns = [0, 1]

   # Iterate through each feature column
   for column in feature_columns:
      # Extract audio data from the feature
         column
      audio_data = feature[column]

      # Compute the spectrogram for the
         audio data
      spec = compute_spectrogram(audio_data)

      # Append the computed spectrogram to
         the appropriate list based on the
         column index
      if column == 0:
         spectrogramsA.append(spec)
      else:
         spectrogramsB.append(spec)

# Display spectrogram for sample 20 in
    'spectrogramsA'
spec = np.squeeze(spectrogramsA[20].numpy())
plt.imshow(spec, aspect='auto',
    origin='lower')
plt.title(f'Spectrogram A,
    target={targets[20]} m/s') # Set the
    title with the corresponding target value
plt.xlabel('Time')
plt.ylabel('Frequency Bin')
plt.colorbar(format='%+2.0f dB')
plt.show()

# Display spectrogram for sample 20 in
    'spectrogramsB'
spec = np.squeeze(spectrogramsB[20].numpy())
plt.imshow(spec, aspect='auto',
    origin='lower')
plt.title(f'Spectrogram B,
    target={targets[20]} m/s') # Set the
    title with the corresponding target value
plt.xlabel('Time')
plt.ylabel('Frequency Bin')
plt.colorbar(format='%+2.0f dB')
plt.show()

# Display spectrogram for sample 750 in
    'spectrogramsA'
spec = np.squeeze(spectrogramsA[750].numpy())
plt.imshow(spec, aspect='auto',
    origin='lower')
plt.title(f'Spectrogram A,
    target={targets[750]} m/s') # Set the
    title with the corresponding target value
plt.xlabel('Time')
plt.ylabel('Frequency Bin')
plt.colorbar(format='%+2.0f dB')
plt.show()

# Display spectrogram for sample 750 in
    'spectrogramsB'
spec = np.squeeze(spectrogramsB[750].numpy())
plt.imshow(spec, aspect='auto',
    origin='lower')
plt.title(f'Spectrogram B,
    target={targets[750]} m/s') # Set the
    title with the corresponding target value
plt.xlabel('Time')
plt.ylabel('Frequency Bin')
plt.colorbar(format='%+2.0f dB')
plt.show()

# Initialize a list to store correlation
    spectrograms
correlation_spectrogram = []

# Iterate through pairs of spectrograms from
    'spectrogramsA' and 'spectrogramsB'
for specA, specB in zip(spectrogramsA,
    spectrogramsB):
   # Ensure that both spectrograms have the
      same shape for correlation
   assert specA.shape == specB.shape,
      "Spectrograms must have the same
      shape for correlation"

   # Flatten and compute cross-correlation
      between the two spectrograms
   cross_corr =
      np.correlate(np.array(specA).flatten(),
      np.array(specB).flatten(),
      mode='full')

   # Normalize the cross-correlation values
   normalized_cross_corr = cross_corr /
      np.max(np.abs(cross_corr))

   # Append the normalized cross-correlation
      to the list
   correlation_spectrogram.append(normalized_cross_corr)

# Convert the list of correlation
    spectrograms to a numpy array
```

```
correlation_spectrogram =
    np.array(correlation_spectrogram)

# Display the correlation spectrogram
plt.imshow(correlation_spectrogram,
    aspect='auto', cmap='jet',
    origin='lower')
plt.title('Correlation Spectrogram')
plt.xlabel('Time lag')
plt.ylabel('Sample index')
plt.colorbar()
plt.show()

# Concatenate spectrograms from
    'spectrogramsA' and 'spectrogramsB'
    along the last axis
spectrogram_pair_train =
    tf.concat([spectrogramsA,
    spectrogramsB], axis=-1)

# Split the data into training and testing
    sets
X_train = spectrogram_pair_train[:1200]
X_test = spectrogram_pair_train[1200:]
y_train = np.array(targets[:1200])
y_test = np.array(targets[1200:])

# Display the shapes of the training data
    and labels
print(X_train.shape)
print(y_train.shape)

# Create a TensorFlow dataset for the
    training set
train_dataset =
    tf.data.Dataset.from_tensor_slices((X_train,
    y_train))
# Shuffle the dataset, create batches, and
    prefetch for optimization
train_dataset =
    train_dataset.shuffle(buffer_size=len(spectrogram_pair_train)).batch(batch_size=256).prefetch(tf.dat

# Create a TensorFlow dataset for the
    testing set
test_dataset =
    tf.data.Dataset.from_tensor_slices((X_test,
    y_test))
# Create batches and prefetch for
    optimization
test_dataset =
    test_dataset.batch(batch_size=256).prefetch(tf.data.experimental.AUTOTUNE)

# Defining the model
model = Sequential()
model.add(Resizing(32,32))
model.add(Normalization())

model.add(Conv2D(32, (2, 2), padding='same',
    activation = 'relu', input_shape =
    X_train.shape))

model.add(Conv2D(64, (2, 2), padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3)) # Adding dropouts to
    avoid overfitting

model.add(Flatten())
```

```
model.add(Dense(32, activation = 'relu'))
model.add(Dropout(0.3))

model.add(Dense(1))

model.build(input_shape=X_train.shape)
model.summary()

epochs = 100 #Compiling model
model.compile(optimizer='adam', loss =
    'mean_squared_error', metrics=['mae'])

history = model.fit(train_dataset, epochs =
    epochs, validation_data = test_dataset,
    verbose=1) # Learning a model

train_loss = history.history['loss']
train_mae = history.history['mae']

val_loss = history.history['val_loss']
val_mae = history.history['val_mae']

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(range(1, epochs + 1), train_loss,
    label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss,
    label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(1, epochs + 1), train_mae,
    label='Training MAE')
plt.plot(range(1, epochs + 1), val_mae,
    label='Validation MAE')
plt.title('Training and Validation MAE')
plt.xlabel('Epochs')
plt.ylabel('MAE')
plt.legend()

plt.tight_layout()
plt.show()
```