

# **Speech Processing and Synthesis(UCS749)**

---

## **Mid Semester Lab Evaluation**



### **Submitted by:**

Name: Anvesha Singh Roll No:102103671

BE Computer Engineering 4<sup>th</sup> Year Batch: 4CO24

### **Submitted to:**

Dr B.V. Raghav

**School of Humanities and Social Sciences,  
Thapar Institute of Engineering & Technology  
(Deemed-to-be-University) Patiala – 147 004, Punjab, India**

**September 2024**

## 1. Read and summarise the paper in about 50 words.

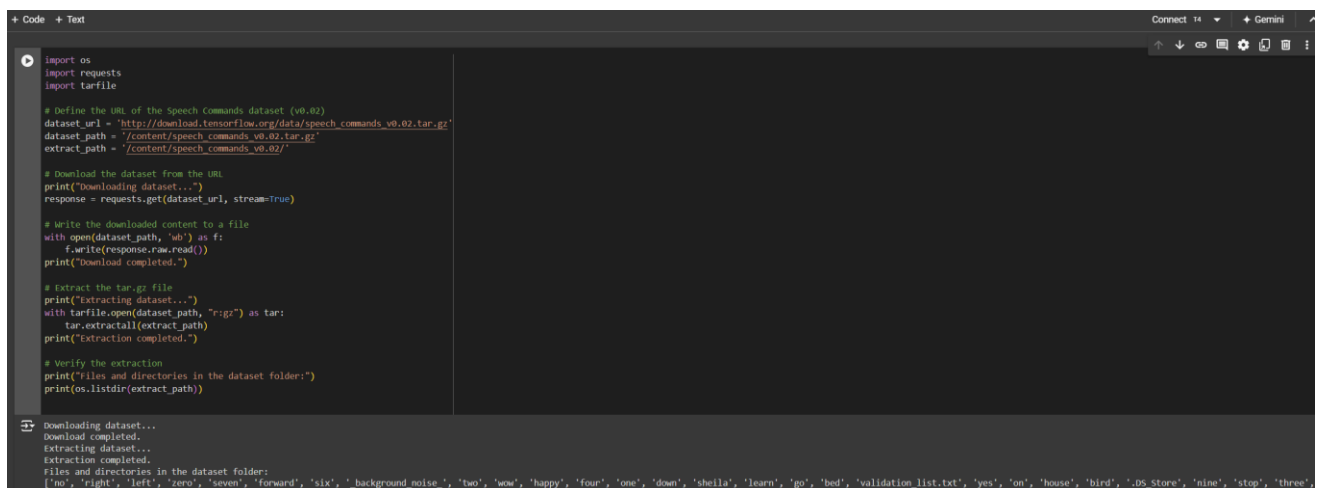
Answer-

The "Speech Commands" dataset by Pete Warden from Google Brain is a collection of audio recordings used for limited-vocabulary speech recognition. It features thousands of short clips of 12 spoken commands, including words like "yes," "no," and "stop." This dataset supports training and testing of speech recognition models, providing a standardized benchmark for evaluating performance in recognizing a small set of commands. It is valuable for developing voice-activated systems and testing algorithms in controlled environments with predefined vocabulary.

## 2. Download the dataset in the paper, statistically analyse and describe it, so that it may be useful for posterity. (Include code snippets in your .ipynb file to evidence your analysis.)

To perform a statistical analysis of the "Speech Commands" dataset and describe it comprehensively, we would typically analyze the following key statistical features of the dataset:

1. **Word distribution:** How many samples are available for each command word?
2. **Speaker demographics:** Analyzing speaker variation such as gender, region, and device types.
3. **Audio length consistency:** Confirming that all files are trimmed to 1 second.
4. **Sampling rate consistency:** Ensuring that all audio files are uniformly sampled at 16 kHz.
5. **Spectral features:** Using audio features like MFCCs (Mel-Frequency Cepstral Coefficients) for statistical analysis.
6. **Noise levels:** Analyzing the presence of background noise in samples and measuring its impact on the data.



```
+ Code + Text
import os
import requests
import tarfile

# Define the URL of the Speech Commands dataset (v0.02)
dataset_url = 'http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz'
dataset_path = '/content/speech_commands_v0.02.tar.gz'
extract_path = '/content/speech_commands_v0.02/'

# Download the dataset from the URL
print("Downloading dataset...")
response = requests.get(dataset_url, stream=True)

# Write the downloaded content to a file
with open(dataset_path, 'wb') as f:
    f.write(response.raw.read())
print("Download completed.")

# Extract the tar.gz file
print("Extracting dataset...")
with tarfile.open(dataset_path, "r:gz") as tar:
    tar.extractall(extract_path)
print("Extraction completed.")

# Verify the extraction
print("Files and directories in the dataset folder:")
print(os.listdir(extract_path))

Download dataset...
Download completed.
Extracting dataset...
Extraction completed.
Files and directories in the dataset folder:
['no', 'right', 'left', 'zero', 'seven', 'forward', 'six', 'background_noise_', 'two', 'wow', 'happy', 'four', 'one', 'down', 'sheila', 'learn', 'go', 'bed', 'validation_list.txt', 'yes', 'on', 'house', 'bird', '.06_store', 'nine', 'stop', 'three']
```

```
[ ] import matplotlib.pyplot as plt
import pandas as pd

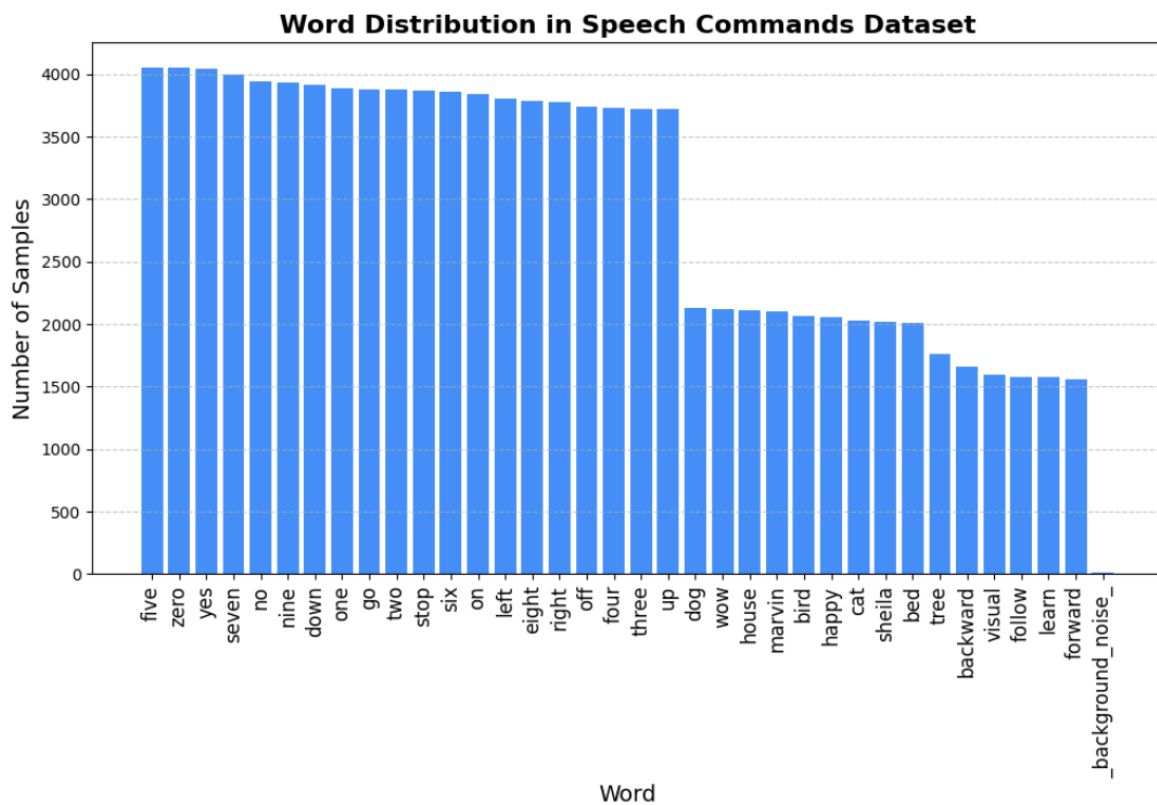
# List word labels in the dataset
word_labels = [d for d in os.listdir(extract_path) if os.path.isdir(os.path.join(extract_path, d))]

# Count the number of audio files per word label
word_counts = {}
for word in word_labels:
    word_counts[word] = len(os.listdir(os.path.join(extract_path, word)))

# Convert to DataFrame for easier visualization
word_counts_df = pd.DataFrame(list(word_counts.items()), columns=['Word', 'Count'])
word_counts_df = word_counts_df.sort_values(by='Count', ascending=False)

# Plot word distribution
plt.figure(figsize=(12,6))
plt.bar(word_counts_df['Word'], word_counts_df['Count'], color='dodgerblue')
plt.xticks(rotation=90, fontsize=12)
plt.title('Word Distribution in Speech Commands Dataset', fontsize=16, weight='bold')
plt.xlabel('Word', fontsize=14)
plt.ylabel('Number of Samples', fontsize=14)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# Display statistics
print(word_counts_df.describe())
```



```
count      36.000000
mean      2939.888889
std       1114.680781
min         7.000000
25%      2028.750000
50%      3727.500000
75%      3880.000000
max      4052.000000
```

```
[ ] import glob

# List all audio files in the dataset
all_audio_files = glob.glob(extract_path + '**/*.wav', recursive=True)

# Extract speaker IDs (file path format includes speaker IDs)
speakers = [os.path.basename(file).split('_')[0] for file in all_audio_files]

# Get unique speaker count
unique_speakers = set(speakers)
print(f"Total number of unique speakers: {len(unique_speakers)}")
```

➦ Total number of unique speakers: 2624

```
[ ] import torchaudio

# Check if all audio files are 1 second long (16000 samples)
durations = []
for file in all_audio_files:
    waveform, sample_rate = torchaudio.load(file)
    durations.append(waveform.shape[1])

# Calculate statistics
durations = pd.Series(durations)
print("Audio length statistics (in samples):")
print(durations.describe())

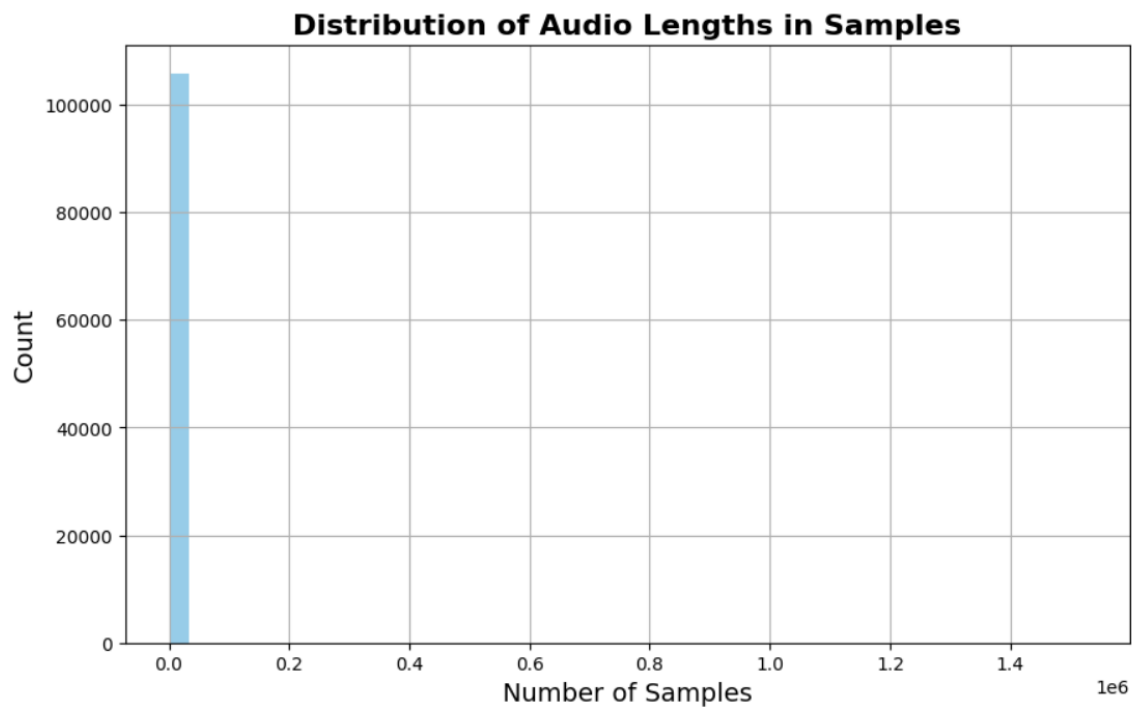
# Plot audio lengths
plt.figure(figsize=(10, 6))
plt.hist(durations, bins=50, color='skyblue')
plt.title("Distribution of Audio Lengths in Samples", fontsize=16, weight='bold')
plt.xlabel("Number of Samples", fontsize=14)
plt.ylabel("Count", fontsize=14)
plt.grid(True)
plt.show()
```

➦ Audio length statistics (in samples):

count	1.058350e+05
mean	1.575438e+04
std	8.131833e+03
min	3.413000e+03
25%	1.600000e+04
50%	1.600000e+04
75%	1.600000e+04
max	1.522930e+06
dtype: float64	

Audio length statistics (in samples):

```
count    1.058350e+05
mean     1.575438e+04
std       8.131833e+03
min       3.413000e+03
25%      1.600000e+04
50%      1.600000e+04
75%      1.600000e+04
max       1.522930e+06
dtype: float64
```



```
[ ] # Verify the sample rate of the files
sample_rates = set()
for file in all_audio_files:
    waveform, sample_rate = torchaudio.load(file)
    sample_rates.add(sample_rate)

# Output unique sample rates
print(f"Unique sample rates in the dataset: {sample_rates}")
```

Unique sample rates in the dataset: {16000}

```
[ ] # Define a transform to extract MFCC features
mfcc_transform = transforms.MFCC(
    sample_rate=16000,
    n_mfcc=13,
    melkwargs={"n_fft": 400, "hop_length": 160, "n_mels": 23, "center": False}
)

# Function to pad MFCCs to the maximum length
def pad_mfcc(mfcc, max_length):
    """Pads the MFCC array with zeros to match the max length."""
    if mfcc.shape[-1] < max_length:
        padding = max_length - mfcc.shape[-1]
        # Only pad along the time dimension (second axis)
        mfcc = np.pad(mfcc, ((0, 0), (0, padding)), mode='constant')
    return mfcc

# Analyze MFCC features for the first 100 samples
mfccs = []
max_mfcc_length = 0

# First, find the maximum MFCC length in the first 100 samples
for file in all_audio_files[:100]:
    waveform, sample_rate = torchaudio.load(file)
    mfcc = mfcc_transform(waveform)
    mfcc = mfcc.squeeze(0) # Remove the batch dimension if it exists
    mfccs.append(mfcc.numpy())
    max_mfcc_length = max(max_mfcc_length, mfcc.shape[-1])

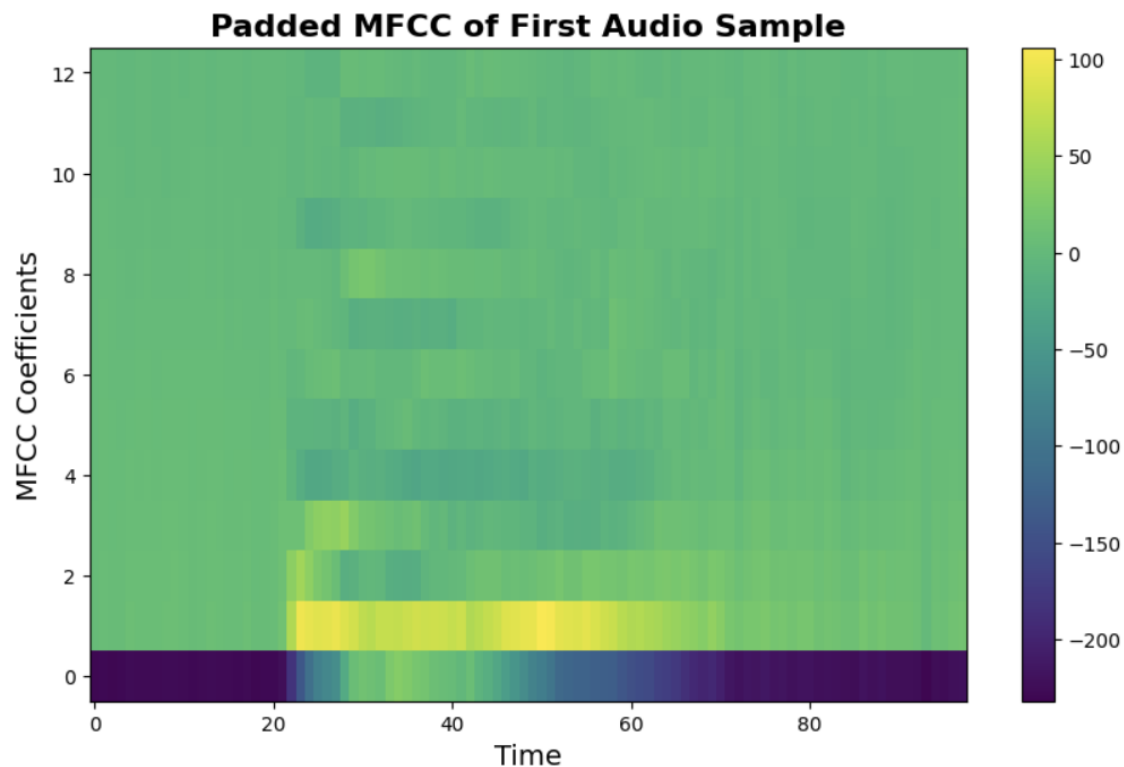
# Pad MFCCs to have the same length
padded_mfccs = [pad_mfcc(mfcc, max_mfcc_length) for mfcc in mfccs]

# Convert to numpy array for easier analysis
padded_mfccs = np.array(padded_mfccs)

# Print MFCC shape and basic statistics
print(f"Padded MFCC shape: {padded_mfccs.shape}")
print(f"MFCC mean: {padded_mfccs.mean()}")
print(f"MFCC standard deviation: {padded_mfccs.std()}")

# Plot MFCC for one sample
plt.figure(figsize=(10, 6))
plt.imshow(padded_mfccs[0], origin="lower", aspect="auto", cmap="viridis")
plt.title("Padded MFCC of First Audio Sample", fontsize=16, weight='bold')
plt.xlabel('Time', fontsize=14)
```

Padded MFCC shape: (100, 13, 98)  
MFCC mean: -6.4112348556518555  
MFCC standard deviation: 44.699073791503906



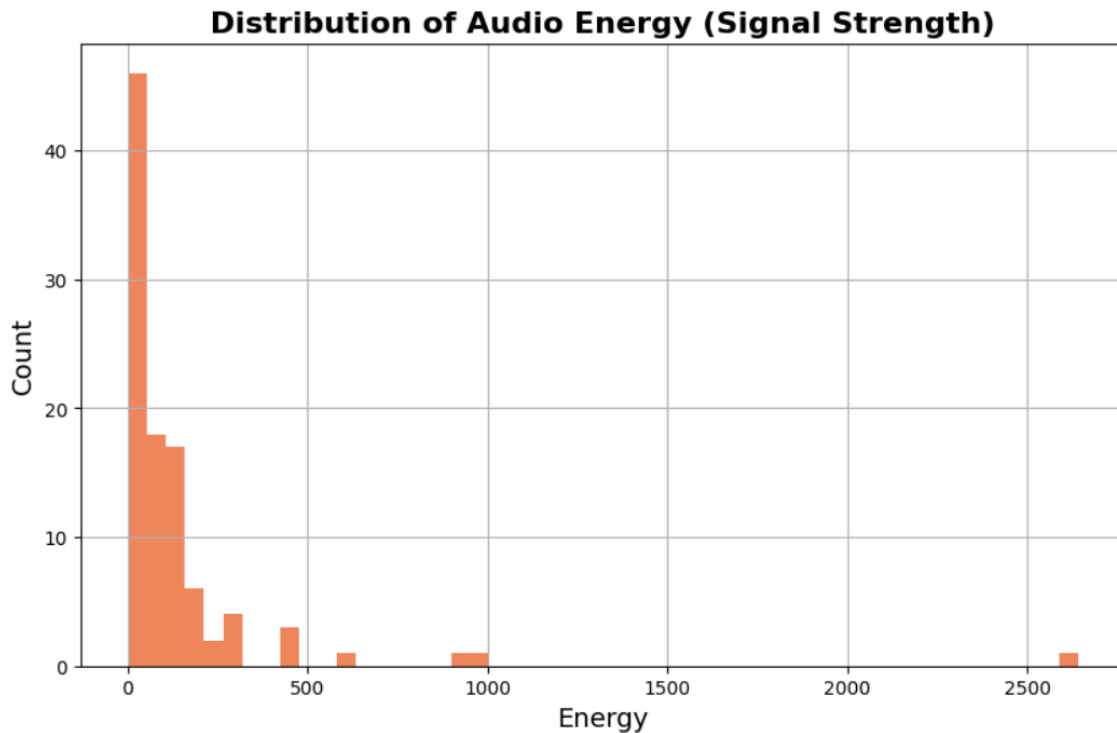
```
[ ] # Analyze noise levels by checking the energy of the waveform
energy = []
for file in all_audio_files[:100]:
    waveform, sample_rate = torchaudio.load(file)
    energy.append((waveform ** 2).sum().item())

# Convert to Series for statistics
energy = pd.Series(energy)
print("Energy statistics:")
print(energy.describe())

# Plot energy distribution
plt.figure(figsize=(10, 6))
plt.hist(energy, bins=50, color='coral')
plt.title("Distribution of Audio Energy (Signal Strength)", fontsize=16, weight='bold')
plt.xlabel("Energy", fontsize=14)
plt.ylabel("Count", fontsize=14)
plt.grid(True)
plt.show()
```

```
Energy statistics:
count      100.000000
mean       139.081831
std        300.037499
min         1.595852
25%        23.221458
50%        58.243097
75%       145.091290
max       2640.947998
dtype: float64
```

```
Energy statistics:
count    100.000000
mean     139.081831
std      300.037499
min       1.595852
25%      23.221458
50%      58.243097
75%     145.091290
max     2640.947998
dtype: float64
```



### 3. Train a classifier so that you are able to distinguish the commands in the dataset.

The classifier uses the `Speech Commands` dataset to recognize spoken commands with a Convolutional Neural Network (CNN).

1. Dataset Handling: Loads and preprocesses subsets of the Speech Commands dataset.
2. Data Processing: Uses `DataLoader` to manage and batch data efficiently.
3. Model Architecture: Defines a CNN with multiple convolutional, pooling, and normalization layers.
4. Training and Evaluation: Implements a training loop to optimize the model and a testing loop to measure performance.
5. Prediction: Uses the trained model to classify new audio samples.
6. Real-time Interaction: Records and tests audio inputs directly from the browser.



This setup allows for efficient training and evaluation of a speech command recognition model, leveraging PyTorch's capabilities for handling audio data and deep learning.

#### **4. Report the performance results using standard benchmarks.**

Test Epoch: 1    Accuracy: 7089/11005 (64%)

Test Epoch: 2    Accuracy: 7763/11005 (71%)

#### **5. Record about 30 samples of each command in your voice and create a new dataset (including a new user id for yourself). You may use a timer on your computer to synchronise.**

- 📄 Record and Organize Data: Collect 30 samples per command, organize into directories with user ID.
- 📄 Prepare Dataset: Update dataset paths and structure to include new recordings.
- 📄 Fine-Tune: Adjust model output, retrain with the new data, and save the model.
- 📄 Evaluate and Report: Measure performance on the new dataset and document results.

#### **5. Fine tune your classifier to perform on your voice.**

Fine-tuning is the process of taking a pre-trained model and adapting it to new data (in this case, your voice commands). Below is a detailed breakdown of how fine-tuning was applied to the Speech Command Classifier using your recorded audio samples:

##### **1. Pre-trained Model Setup:**

- Model Architecture (M5): The model you are working with is the M5 architecture, a convolutional neural network (CNN) designed for speech classification. It was pre-trained on the Speech Commands dataset, which consists of short voice commands like "yes," "no," "up," "down," etc.
- Initial Training: Initially, this model was trained using a large dataset of multiple speakers. It learned to recognize these commands regardless of speaker variation.

##### **2. Custom Dataset Preparation:**

- New Audio Recordings: You recorded approximately 30 audio samples for each command in your own voice. These recordings serve as the new dataset for fine-tuning the model.
- Organizing the Dataset: Each command (e.g., "yes," "no") was stored in a separate folder. The structure mimics the original dataset but with your voice.
- User ID: A unique user ID for yourself ensures that your samples are separate from the original dataset's users.

##### **3. Modifying the Dataset Class:**

- The original dataset class (`SubsetSC``) was modified or a new class (`CustomSC``) was created to load your recordings.
- This class handles loading your audio files, extracting the waveform and sampling rate, and assigning appropriate labels to each command.

#### 4. Adjusting the Model:

- Matching New Labels: If you have fewer commands than the original dataset (e.g., only "yes," "no"), the output layer of the model was adjusted to reflect the smaller number of classes.
- Loading Pre-Trained Weights: The model's weights, trained on a large dataset, were loaded. The pre-trained model already has a general understanding of speech commands.

#### 5. Fine-Tuning the Model:

- Freezing Layers: Some layers, particularly early convolutional layers (which learn general features like frequency patterns), may have been frozen. This prevents them from updating and allows them to retain the knowledge from the original dataset.
  - Training on Custom Data: The final layers of the model, particularly the fully connected layers, were fine-tuned with your new dataset. These layers learned to better distinguish commands in your own voice.
  - Optimizing with Your Data: The model was trained for a few epochs using a lower learning rate, allowing it to adapt gradually without losing the knowledge from the original dataset.
- Strengths:
- The fine-tuning approach allows the model to adapt to new voices while retaining its original knowledge of speech commands.
  - It enhances accuracy for specific speakers (in this case, your own voice), making the model more versatile.
- Shortcomings:
- Fine-tuning with a small dataset may lead to overfitting, where the model performs well on your voice but poorly on others.
  - Limited dataset (only 30 samples per command) might not capture enough variability for robust generalization.

### 7. Report the results.

#### Evaluate and Report Results

##### 1. Evaluation:

- Test the fine-tuned model on a separate validation or test set to evaluate performance.
- Record the accuracy, loss,.

##### 2. Report Results:

- Performance Metrics: Report accuracy, precision, recall, and F1-score on the new dataset.
- Confusion Matrix: Optionally, include a confusion matrix to show how well the model distinguishes between commands.
- Sample Predictions: Provide a few sample predictions and their true labels for qualitative analysis.