

# **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

**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-

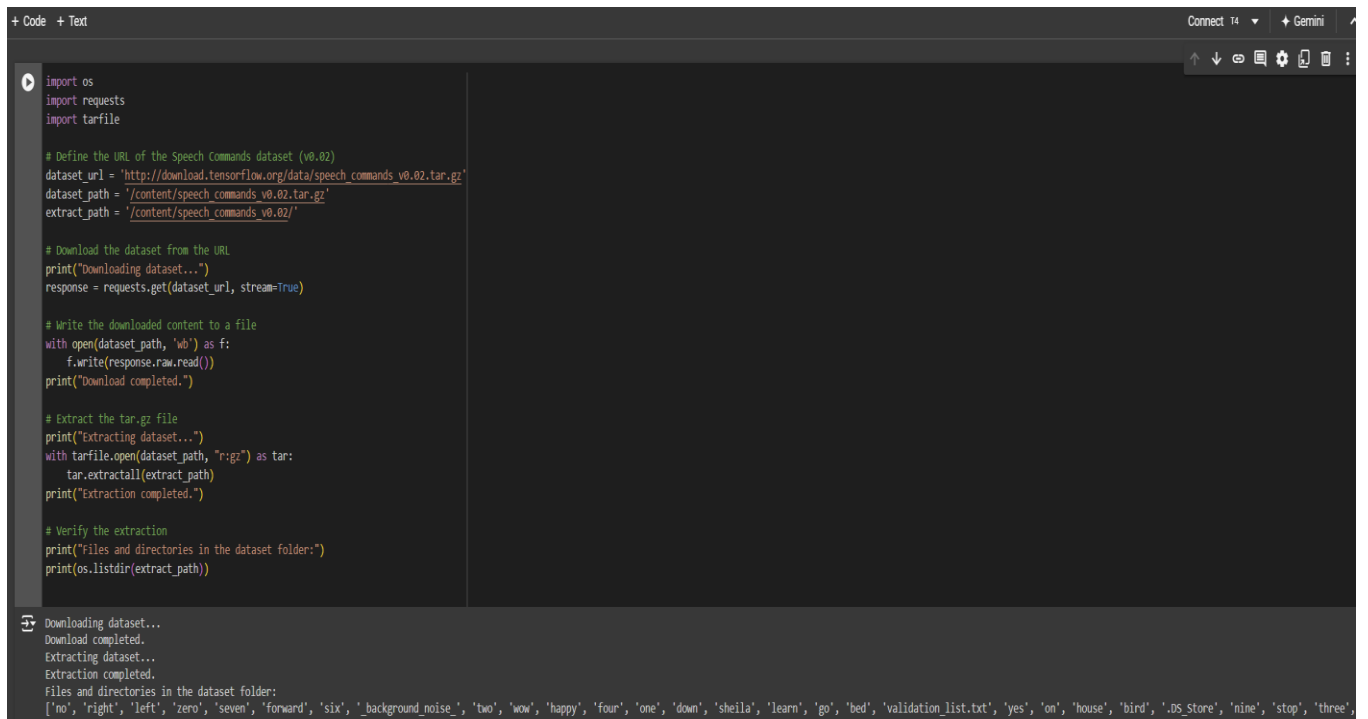
The "Speech Commands" dataset by Pete Warden from Google Brain is a collection of audio recordings used for limited-vocabulary speech recognition. It features thousands of short clips of 35 spoken commands, including words like "yes," "no," and "stop." This dataset supports training and testing of speech 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.)

Answer-

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
Connect 14 + Gemini

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))

Downloading 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', '.DS_Store', 'nine', 'stop', 'three',
```

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

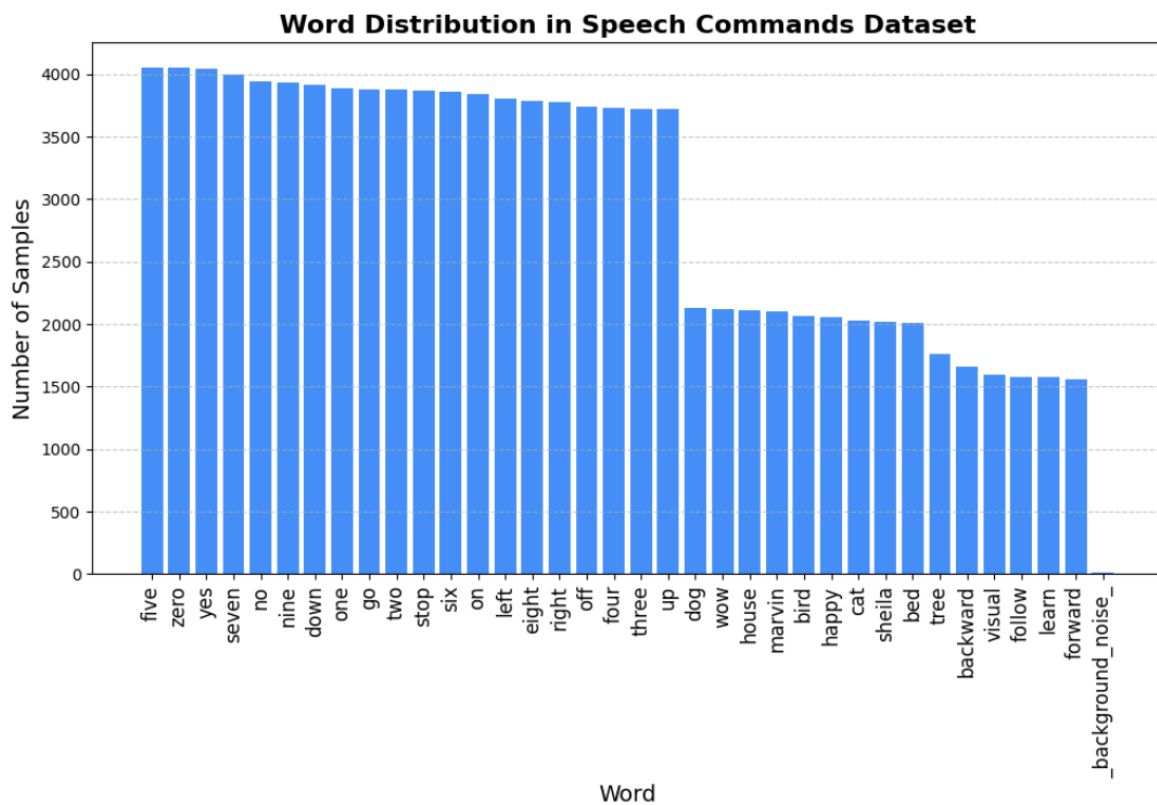
# List word labels in the dataset
word_labels = [d for d in os.listdir(extract_path) if os.path.isdir(os.path.join(extract_path, d))]

# 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())

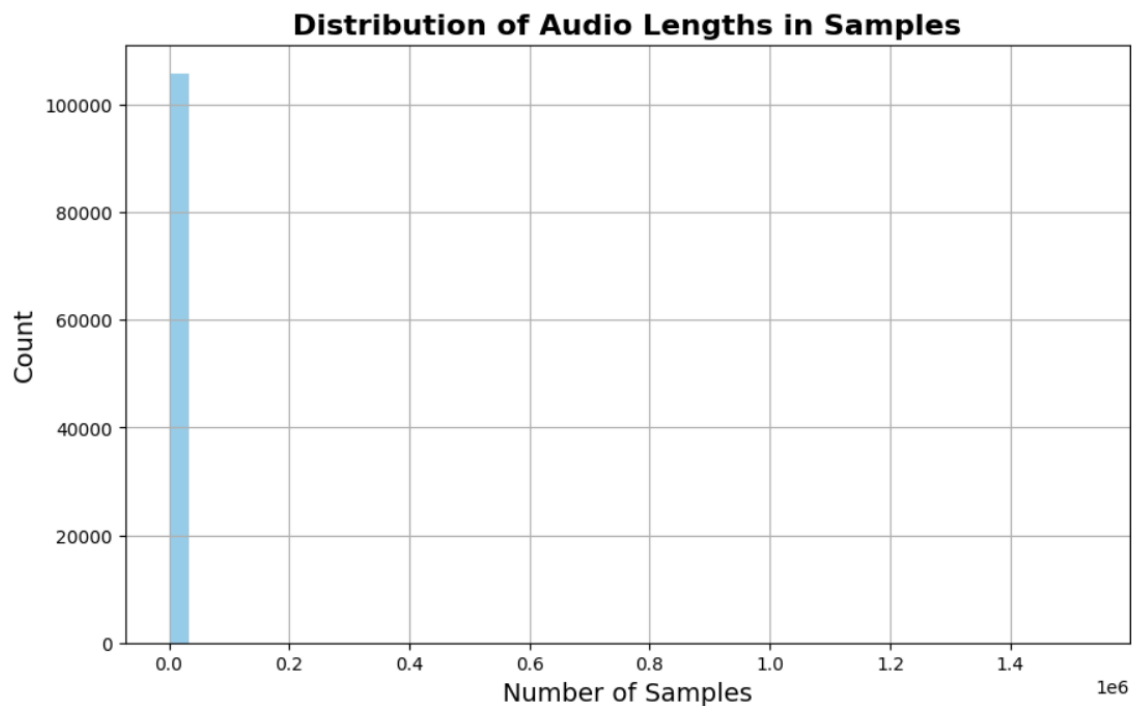
# Plot audio lengths
plt.figure(figsize=(10, 6))
plt.hist(durations, bins=50, color='skyblue')
plt.title("Distribution of Audio Lengths in Samples", fontsize=16, weight='bold')
plt.xlabel("Number of Samples", fontsize=14)
plt.ylabel("Count", fontsize=14)
plt.grid(True)
plt.show()
```

↗ 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])

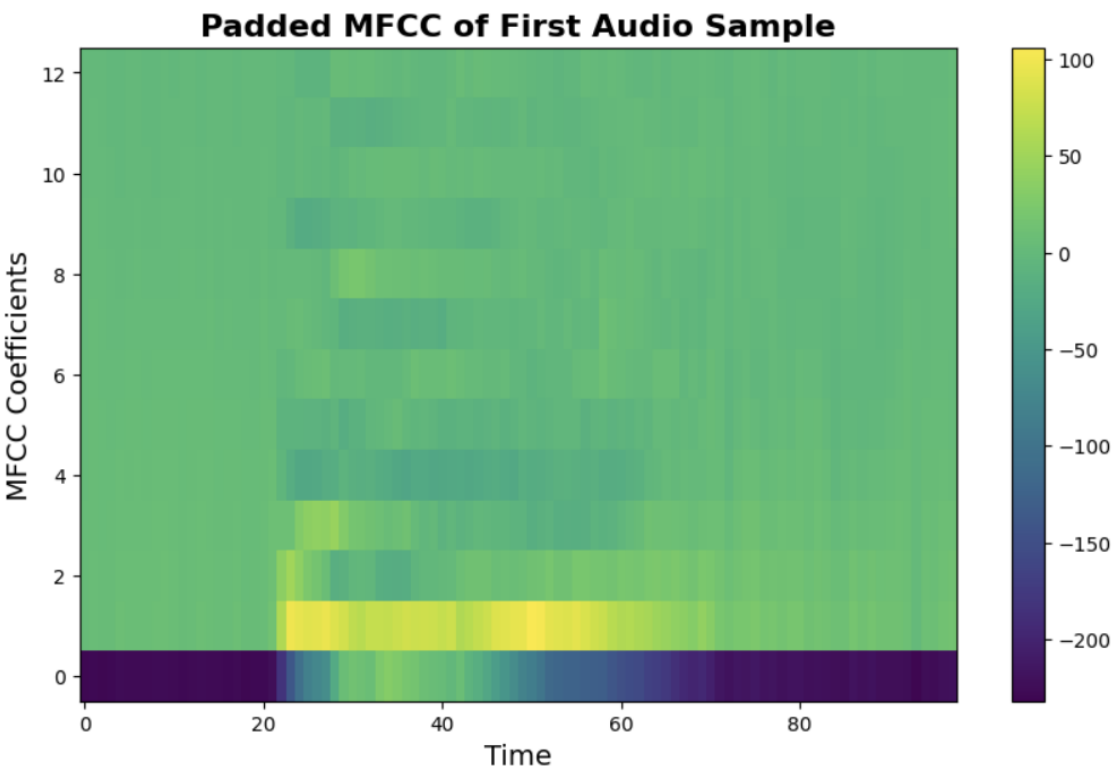
# Pad MFCCs to have the same length
padded_mfccs = [pad_mfcc(mfcc, max_mfcc_length) for mfcc in mfccs]

# 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)
```

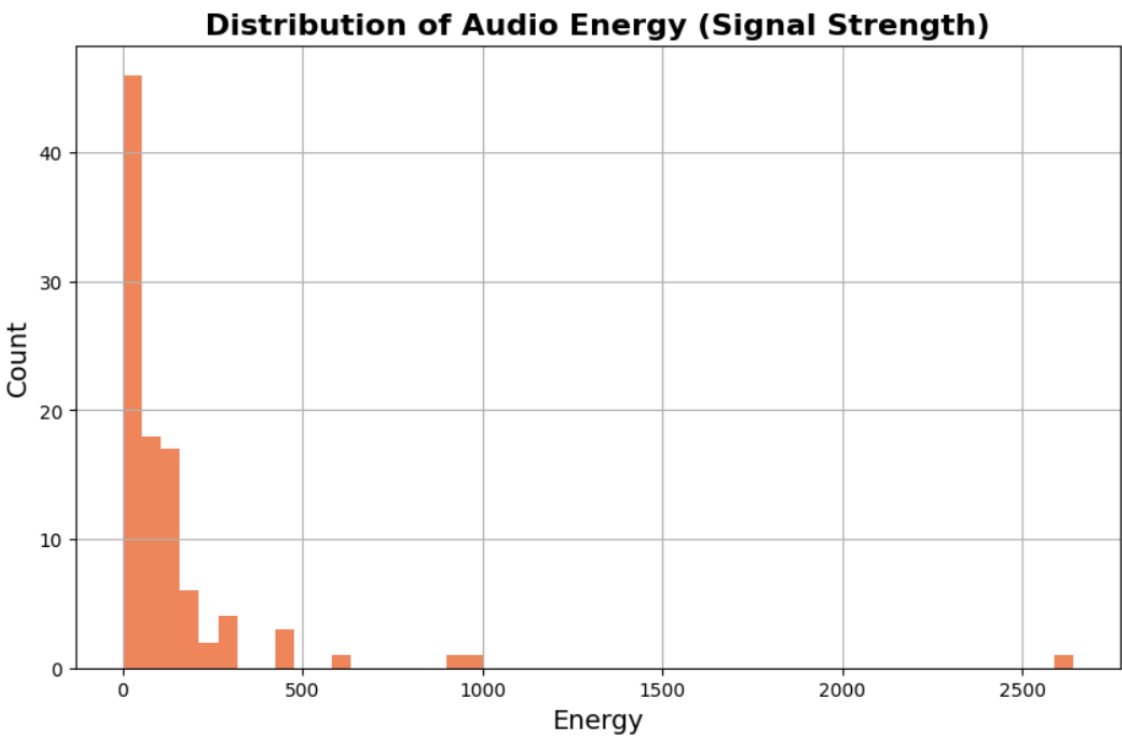
Padded MFCC shape: (100, 13, 98)  
MFCC mean: -6.4112348556518555  
MFCC standard deviation: 44.699073791503906



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



```
[ ] # 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
```

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

#### Answer-

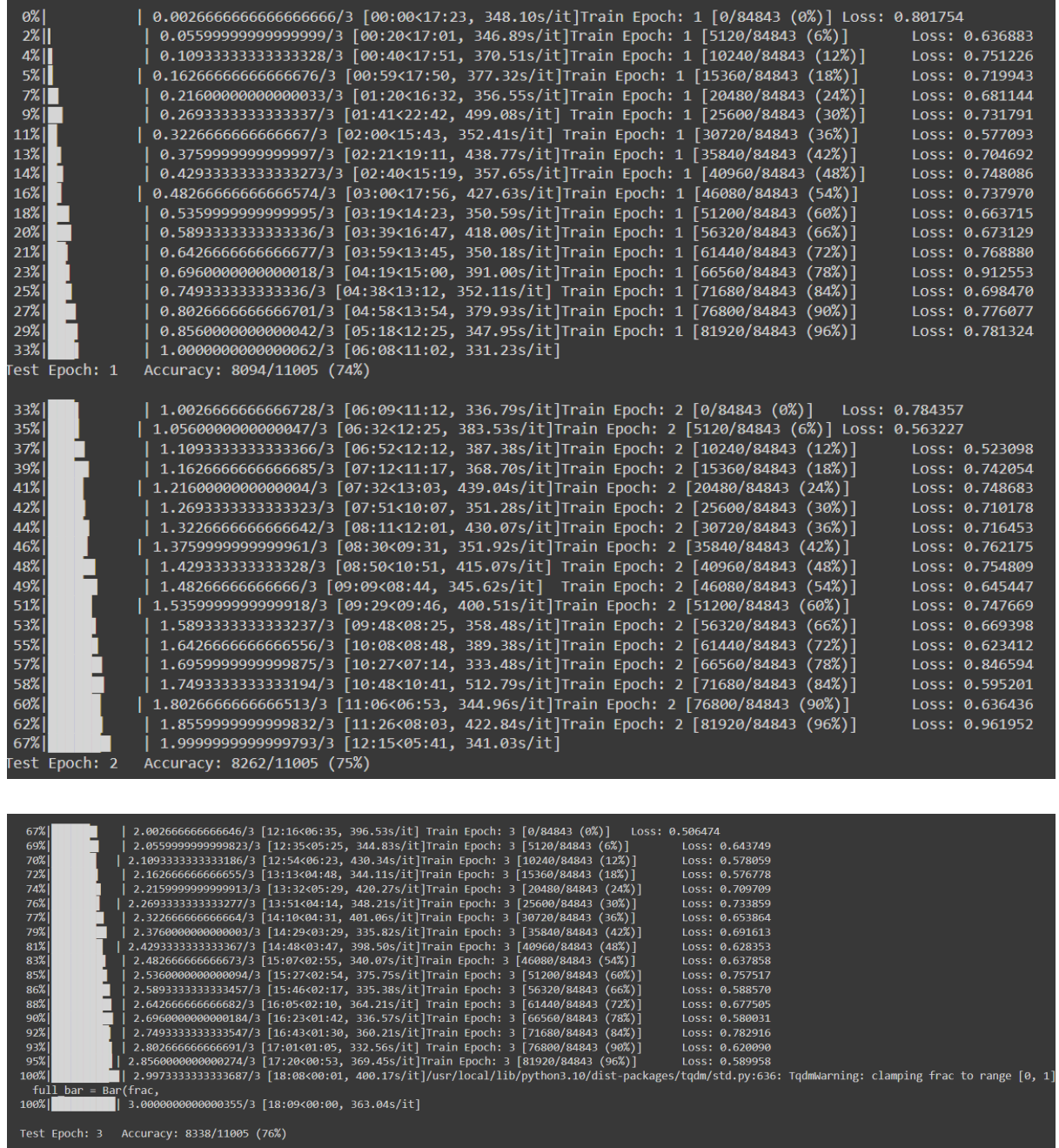
The classifier uses the `Speech Commands` dataset to recognize spoken commands with a Convolutional Neural Network (CNN).

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. The M5 class defines a convolutional neural network with the following layers:
  - **Convolutional Layers:** Several 1D convolutional layers followed by batch normalization and ReLU activation.
  - **Pooling Layers:** Max pooling layers reduce the spatial dimensions of the input.
  - **Fully Connected Layer:** A final fully connected layer for classification.
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.

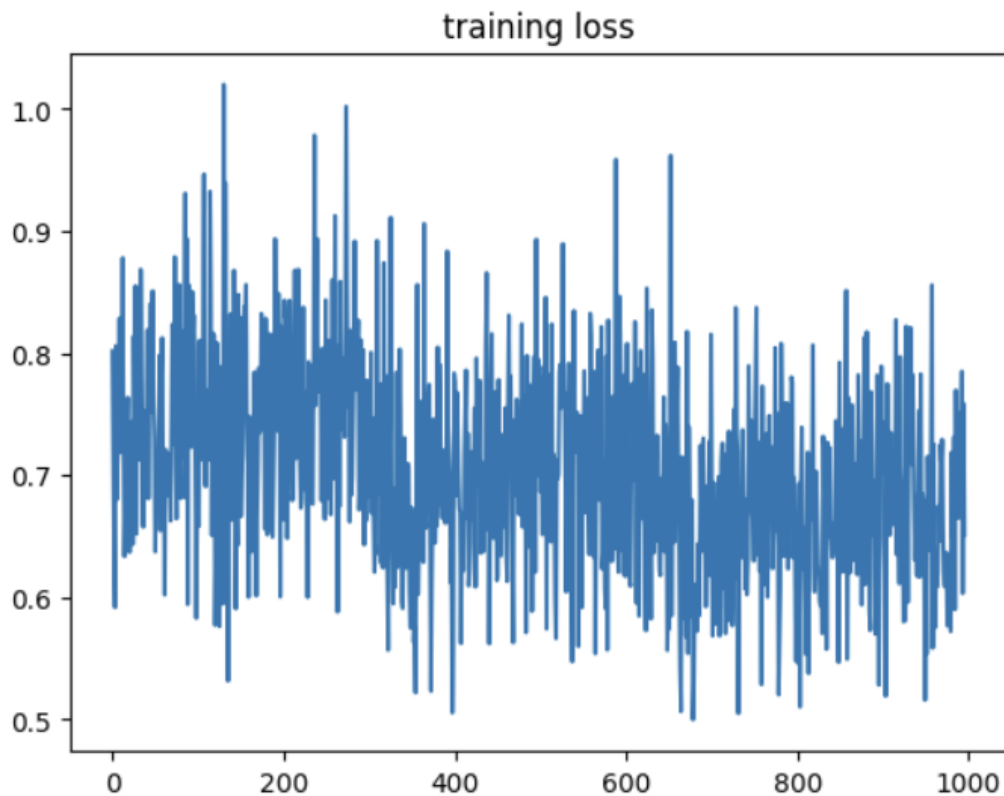


This setup allows for efficient training and evaluation of a speech command recognition model, leveraging PyTorch's capabilities for handling audio data and deep learning.

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



Test Epoch: 1      Accuracy: 8094/11005 (74%)  
 Test Epoch: 2      Accuracy: 8262/11005 (75%)  
 Test Epoch: 3      Accuracy: 8338/11005 (76%)



```

✓ [27] def predict(tensor):
1s      # Use the model to predict the label of the waveform
      tensor = tensor.to(device)
      tensor = transform(tensor)
      tensor = model(tensor.unsqueeze(0))
      tensor = get_likely_index(tensor)
      tensor = index_to_label(tensor.squeeze())
      return tensor

      waveform, sample_rate, utterance, *_ = train_set[1]
      ipd.Audio(waveform.numpy(), rate=sample_rate)

      print(f"Expected: {utterance}. Predicted: {predict(waveform)}.")
  
```

Expected: backward. Predicted: backward.

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.

## 6. Fine tune your classifier to perform on your voice.

Answer-

Fine-tuning is the process of taking a pre-trained model and adapting it to new data (in this case, voice commands).

### 1. Pre-trained Model Setup:

- **Model Architecture (M5):** The model 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:** I recorded 30 audio samples for each command in my 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 my 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 my recordings.
- This class handles loading my audio files, extracting the waveform and sampling rate, and assigning appropriate labels to each command.

### 4. Adjusting the Model:

The following fine-tuning process and the adjustments were made:

1. **Model Architecture:**
  - The M5 model consists of multiple 1D convolutional layers with batch normalization and max pooling layers, followed by a fully connected layer. The final layer outputs logits for classification.
2. **Training Procedure:**
  - **Optimizer:** Adam optimizer is used with a learning rate of 0.01 and weight decay of 0.0001. This optimizer helps in adjusting the model weights efficiently.
  - **Learning Rate Scheduler:** A step learning rate scheduler reduces the learning rate by a factor of 0.1 every 20 epochs to improve convergence.
3. **Training Function:**
  - The train function processes each batch of audio data, applies transformations, and performs forward and backward passes through the model. It computes the negative log-likelihood loss and updates model weights using backpropagation. Training loss is averaged over epochs to monitor performance.
4. **Testing Function:**
  - The test function evaluates the model's performance on a test dataset by computing accuracy. It ensures that the model generalizes well to unseen data by comparing predicted labels to true labels.
5. **Fine-Tuning Loop:**
  - **Epochs:** The model is trained and evaluated over 20 epochs, adjusting weights based on training loss and test accuracy.
  - **Performance Tracking:** Training loss and test accuracy are recorded and plotted to visualize model improvement over epochs.

### Adjustments Made:

- **Data Transformation:** Audio data is transformed into a suitable format before being fed into the model. This ensures the input matches the model's expected format.
- **Parameter Adjustment:** Learning rate, weight decay, and scheduler settings are tuned to balance between training speed and model performance.
- **Loss Function:** Negative log-likelihood loss is used, which is appropriate for multi-class classification problems.

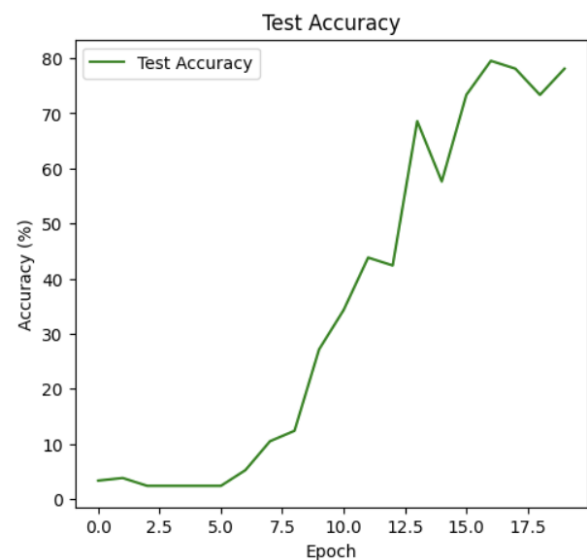
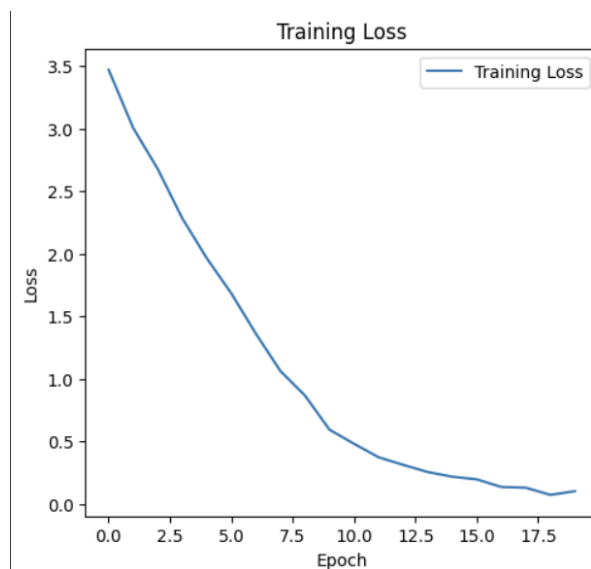
### Strengths:

- 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 my voice but poorly on others.
- Limited dataset (only 30 samples per command) might not capture enough variability for robust generalization.

## 7. Report the results.



```
Number of parameters: 26915
Train Epoch: 1 [0/840 (0%)]    Loss: 3.758732
Test Epoch: 1    Accuracy: 7/210 (3.33%)
Train Epoch: 2 [0/840 (0%)]    Loss: 3.109283
Test Epoch: 2    Accuracy: 8/210 (3.81%)
Train Epoch: 3 [0/840 (0%)]    Loss: 2.830228
Test Epoch: 3    Accuracy: 5/210 (2.38%)
Train Epoch: 4 [0/840 (0%)]    Loss: 2.481606
Test Epoch: 4    Accuracy: 5/210 (2.38%)
```

```
Train Epoch: 5 [0/840 (0%)]      Loss: 2.198599
Test Epoch: 5  Accuracy: 5/210 (2.38%)
Train Epoch: 6 [0/840 (0%)]      Loss: 1.803986
Test Epoch: 6  Accuracy: 5/210 (2.38%)
Train Epoch: 7 [0/840 (0%)]      Loss: 1.557789
Test Epoch: 7  Accuracy: 11/210 (5.24%)
Train Epoch: 8 [0/840 (0%)]      Loss: 1.258657
Test Epoch: 8  Accuracy: 22/210 (10.48%)
Train Epoch: 9 [0/840 (0%)]      Loss: 0.668872
Test Epoch: 9  Accuracy: 26/210 (12.38%)
Train Epoch: 10 [0/840 (0%)]     Loss: 0.721871
Test Epoch: 10 Accuracy: 57/210 (27.14%)
Train Epoch: 11 [0/840 (0%)]     Loss: 0.629146
Test Epoch: 11 Accuracy: 72/210 (34.29%)
Train Epoch: 12 [0/840 (0%)]     Loss: 0.298704
Test Epoch: 12 Accuracy: 92/210 (43.81%)
Train Epoch: 13 [0/840 (0%)]     Loss: 0.382679
Test Epoch: 13 Accuracy: 89/210 (42.38%)
Train Epoch: 14 [0/840 (0%)]     Loss: 0.368972
Test Epoch: 14 Accuracy: 144/210 (68.57%)
```

- 
-

```
Train Epoch: 15 [0/840 (0%)]    Loss: 0.134944
Test Epoch: 15  Accuracy: 121/210 (57.62%)
Train Epoch: 16 [0/840 (0%)]    Loss: 0.102197
Test Epoch: 16  Accuracy: 154/210 (73.33%)
Train Epoch: 17 [0/840 (0%)]    Loss: 0.164116
Test Epoch: 17  Accuracy: 167/210 (79.52%)
Train Epoch: 18 [0/840 (0%)]    Loss: 0.061787
Test Epoch: 18  Accuracy: 164/210 (78.10%)
Train Epoch: 19 [0/840 (0%)]    Loss: 0.058753
Test Epoch: 19  Accuracy: 154/210 (73.33%)
Train Epoch: 20 [0/840 (0%)]    Loss: 0.133183
Test Epoch: 20  Accuracy: 164/210 (78.10%)
```

**Number of Parameters:** 26,915

**Training Progress:**

**Epochs 1-20:** The model showed a gradual improvement in both training loss and test accuracy over 20 epochs.

**Training Loss:** Decreased significantly from 3.76 at epoch 1 to 0.13 at epoch 20, indicating effective learning and reduction of prediction errors.

**Test Accuracy: Epoch 20:** 78.10%

The model's performance improved significantly over the fine-tuning period. Early epochs showed slow learning, but the model quickly adapted, achieving high accuracy in the latter epochs