Name: AKSHAT MEHRA

Roll No: 102115231

Section: 4CO23 (NC7)

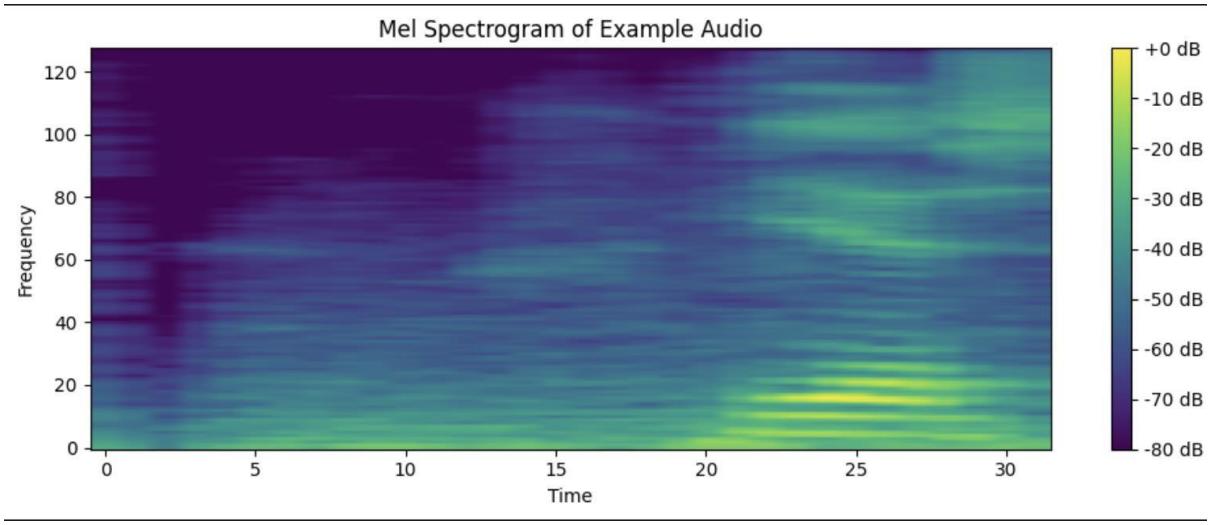
TRAINING THE CLASSIFIER

Audio Preprocessing and Spectrogram Generation

This script processes audio files from a command dataset to prepare them for analysis or use as model input. It begins by specifying the dataset path and selecting directories that represent different command labels. Using librosa, the script loads and preprocesses the audio files, converting them into fixed-length Mel spectrograms.

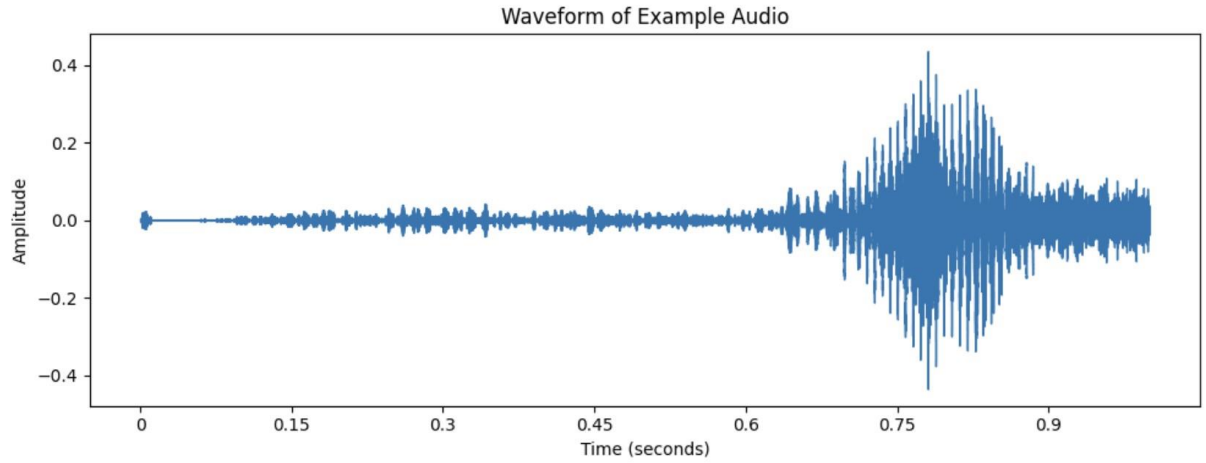
Key steps include:
• Loading and normalizing the audio to a fixed length of 16,000 samples.
• Padding or trimming audio to ensure uniform sample length.
• Converting the audio into a Mel spectrogram and applying a decibel (dB) scale transformation.
• Adding an additional channel dimension to make it compatible with CNNs.
Finally, an example audio file is processed, and the shape of the resulting spectrogram is displayed.***Mel Spectrogram Plot***

The Mel spectrogram plot is a visual representation of the power spectrum of sound frequencies. It is an essential feature in most audio-based machine learning tasks.



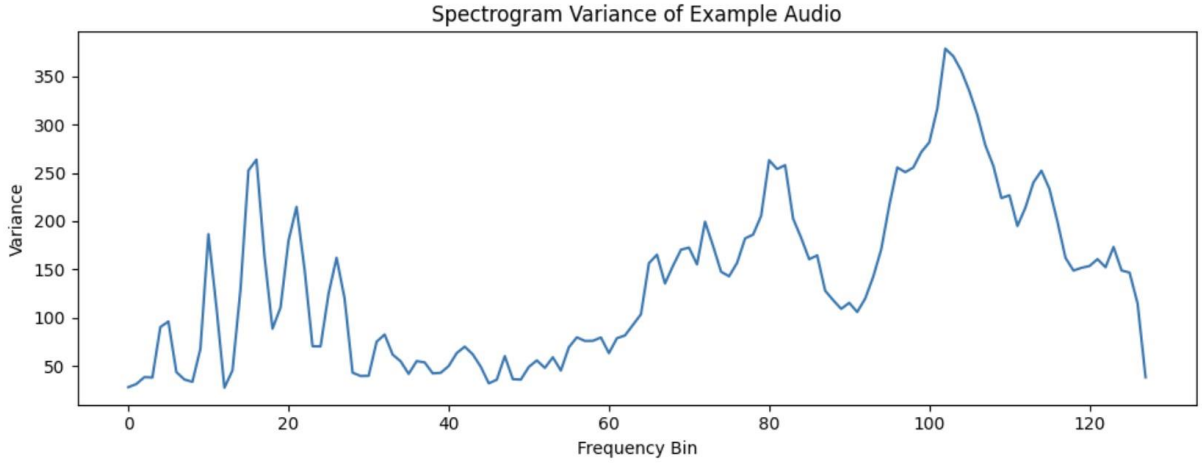
Waveform Plot

The waveform plot shows the raw audio signal's amplitude over time. This is the unprocessed audio that will later be transformed into spectrograms for feature extraction.



Spectrogram Variance Plot

The spectrogram variance plot shows the variance across frequency bins over time, revealing how much the audio signal changes dynamically. This can be particularly useful for analyzing the variation in the audio content.



Label Encoding for Training and Validation Labels

This code snippet utilizes LabelEncoder from scikit-learn to transform string labels (commands) into numeric values, making them more appropriate for machine learning models.

Key steps include:
• Initializing a LabelEncoder instance.
• Fitting the encoder to the training labels and converting them into integer-encoded values.
• Applying the same encoder to the validation labels to ensure consistency.
By encoding the labels, the code ensures that both the training and validation datasets have corresponding numeric representations of their string labels.

Loading and Splitting Audio Dataset

This script loads an audio dataset from a directory and divides it into training and validation sets. The dataset contains .wav files organized in subdirectories, with each folder representing a distinct label (command).

Key steps include:
• Loading the dataset: The load_dataset() function traverses the subdirectories, adding each file's path and its associated label (folder name) to the dataset and label lists.
• Splitting the dataset: Using train_test_split, the dataset is randomly split into training and validation sets in an 80/20 ratio, while maintaining consistent label distribution through stratification.
Finally, the script prints the number of samples in both the training and validation sets.The script finally prints the number of samples in the training and validation sets.

Dataset Analysis: Number of Audio Files per Command

This analysis examines the distribution of audio files across different commands (labels) in the dataset. Each command represents a category of audio, and the number of .wav files in each command folder is counted and visualized.

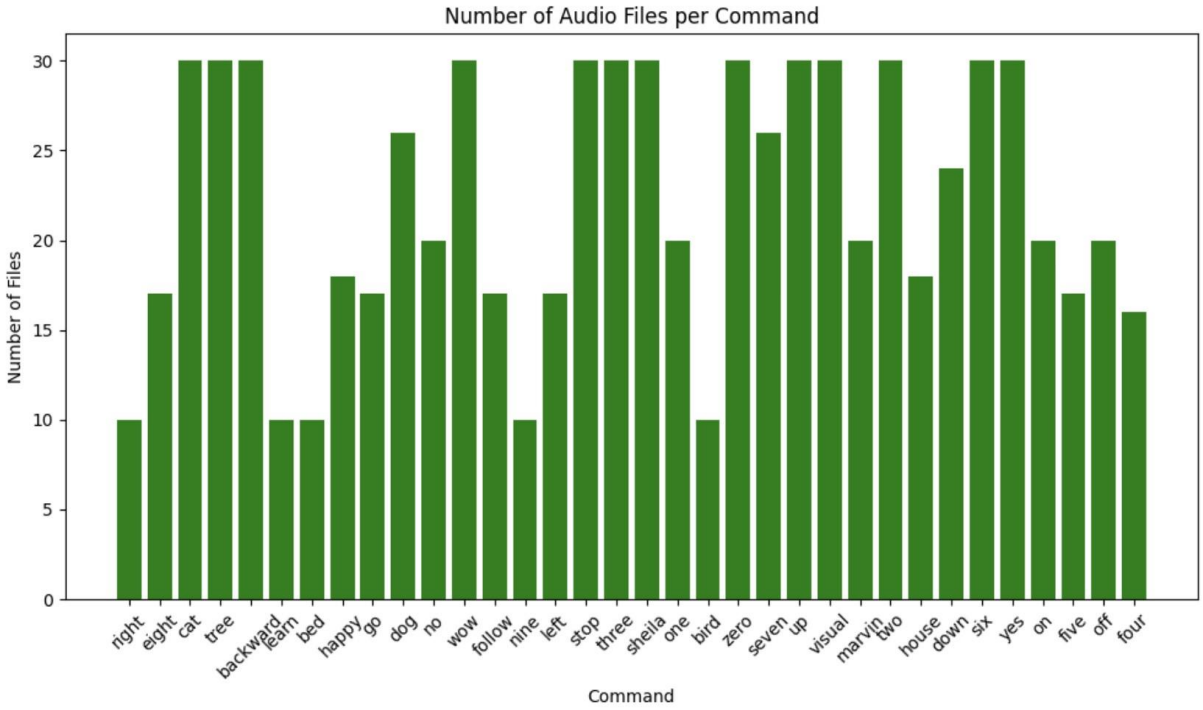
Bar Plot: Number of Audio Files per Command

The bar plot below shows the number of audio files available for each command in the dataset. The y-axis represents the number of files, while the x-axis lists the commands. This visualization helps in understanding the balance of data across different commands. A wellbalanced dataset is critical for training robust machine learning models, as it reduces the risk of bias towards commands with more data.

Key observations from the plot:

- Commands like "eight", "tree", "sheila", and "stop" have the highest number of samples (around 30 files).

- Some commands like "backward", "lean", "nine", and "four" have relatively fewer samples (less than 15).
- Ensuring balanced data for each command can lead to more accurate models.



Audio Data Preprocessing and Dataset Preparation

This script preprocesses audio files by converting them into spectrograms and encoding their labels for machine learning tasks. The workflow involves several key steps:

1. **Audio Decoding:**
    - The decode_audio() function reads the raw audio file and decodes it into a waveform using TensorFlow's tf.audio.decode_wav().
2. **Spectrogram Generation:**
    - The get_spectrogram() function converts the waveform into a spectrogram using Short-Time Fourier Transform (STFT) via TensorFlow's tf.signal.stft(). The absolute value of the spectrogram is then taken for further processing.
3. **Label Extraction:**
    - The get_label() function extracts the label (command) from the file path by splitting the path and taking the second last part (which corresponds to the folder name).
4. **Preprocessing Pipeline:**

- The preprocess() function applies the audio preprocessing and label encoding for each file. It generates a spectrogram from the audio file and encodes the label using the previously fitted LabelEncoder.

5. **Dataset Preparation:**
    - The train_ds and val_ds lists are created by applying the preprocess() function to all training and validation files.
    - The lists are then converted into NumPy arrays for both spectrograms and labels, making them ready for model input.

This code prepares the audio data for training and evaluation in a machine learning pipeline.

CNN Model for Audio Command Classification

This script defines and compiles a Convolutional Neural Network (CNN) for classifying audio commands, with the following steps:

1. **Model Architecture:**
    - The model starts with an input layer designed to accept spectrograms of shape (128, 32, 1)—representing 128 Mel-frequency bands, 32 time steps, and 1 channel.
      • Two convolutional layers (Conv2D) with 32 and 64 filters, each followed by MaxPooling2D layers, are used to extract spatial features from the spectrogram.
      • A Flatten layer converts the 2D feature maps into a 1D vector.
      • A dense hidden layer with 128 units and ReLU activation helps capture higher-level features.
      • The final output layer contains as many units as there are commands (determined by len(label_encoder.classes_)), with a softmax activation for multi-class classification.
2. **Compilation:**
    - The model is compiled using the Adam optimizer, sparse categorical crossentropy loss (since the labels are integer-encoded), and accuracy as the evaluation metric.
3. **Model Summary:**
    - The model's architecture and parameter details are printed with the model.summary() method, providing an overview of the network layers and parameters.

This CNN model is ready for training on the preprocessed spectrogram data.

Model Training with Processed Audio Data

This script trains the previously defined CNN model using the preprocessed audio data:

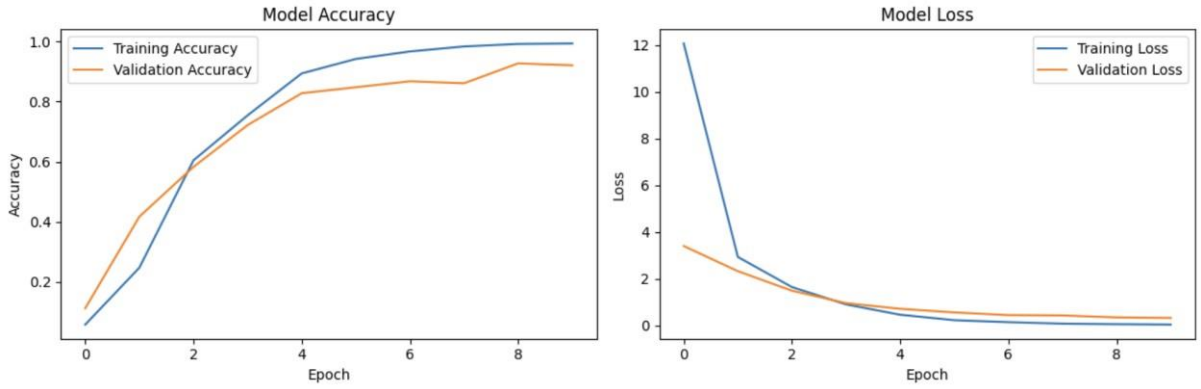
1. **Training:**

- The model.fit() function is used to train the model.
  • train_ds[0] and train_ds[1] supply the training data (spectrograms) and corresponding labels.
  • The model is trained for 10 epochs.
  • Validation is performed using val_ds[0] and val_ds[1], which contain the validation data and labels.
  This process enables the CNN model to learn from the training data and assess its performance on the validation set after each epoch.

Model Evaluation and Training History Visualization

This script evaluates the performance of the trained CNN model and visualizes the training process:

1. **Accuracy Calculation:**
   - The print_accuracy() function computes and prints the accuracy of the model on both the training and validation datasets.
   - It uses model.predict() to obtain predictions, then calculates the predicted labels and compares them to the true labels to compute accuracy.
2. **Accuracy and Loss Visualization:**
   - **Training Accuracy and Validation Accuracy:** Plots the accuracy of the model over epochs for both training and validation datasets. ○ **Training Loss and Validation Loss:** Plots the loss over epochs for both datasets.
   - plt.tight_layout() ensures that the subplots are neatly arranged, and plt.show() displays the plots.



These visualizations help in assessing the model's performance and detecting any signs of overfitting or underfitting during training.