Name: Aanish Verma

Roll No: 102115224

Section: 4CO23 (NC7)

TRAINING THE CLASSIFIER

Audio Preprocessing and Spectrogram Generation

This script preprocesses audio files from a dataset of commands for further analysis or model input. It starts by defining the path to the dataset and filtering out directories representing different commands (labels). The script uses librosa to load and preprocess audio files, converting them into a fixed-length Mel spectrogram.

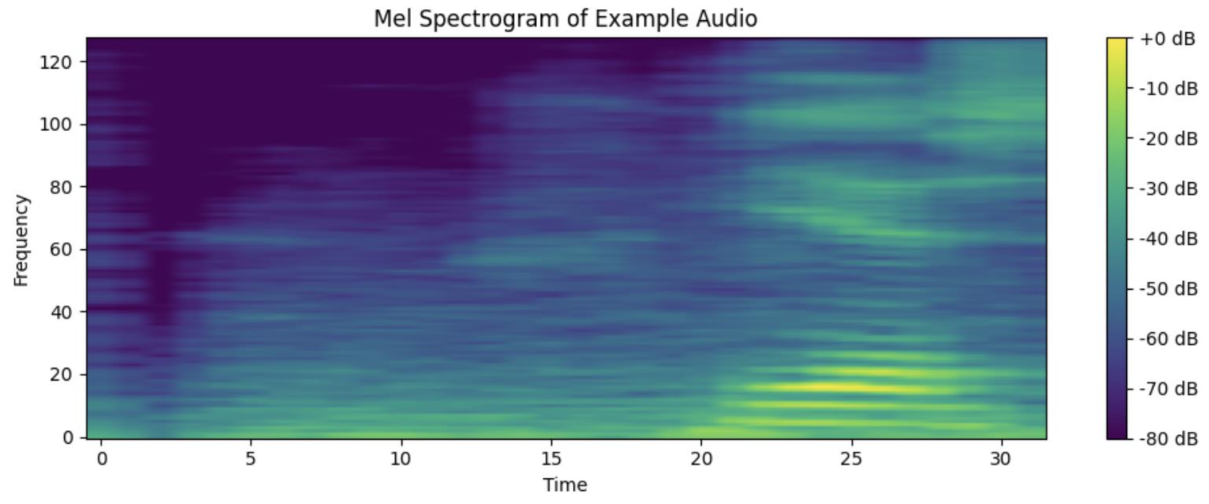
Key steps include:

- Loading and normalizing audio to a fixed length (16,000 samples).
- Padding or truncating audio to maintain consistency across samples.
- Converting audio to a Mel spectrogram and transforming it to the decibel (dB) scale.
- Adding an extra channel dimension for compatibility with convolutional neural networks (CNNs).

An example file is processed, and the resulting spectrogram's shape is printed.

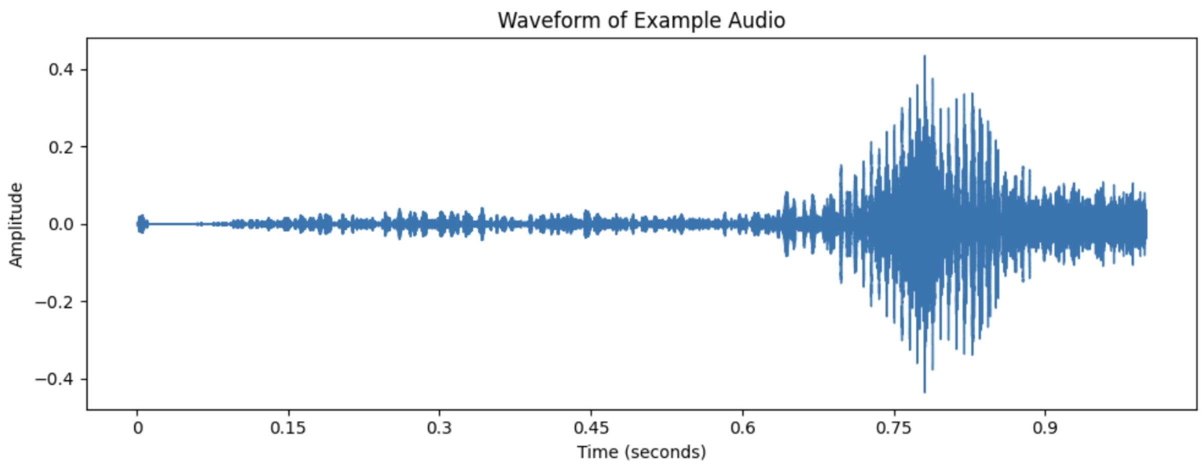
Mel Spectrogram Plot

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.



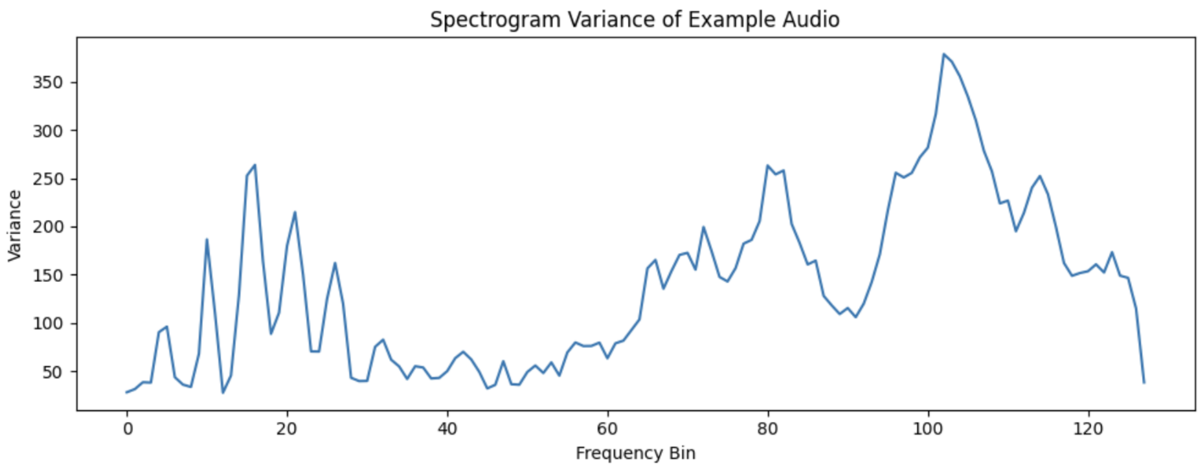
Waveform Plot

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 uses **LabelEncoder** from scikit-learn to convert string labels (commands) into numerical values, which are more suitable for machine learning models.

Key steps include:

- Initializing a LabelEncoder instance.
- Fitting the encoder on the training labels and transforming them into integer-encoded values.
- Using the same encoder to transform the validation labels for consistency.

By encoding labels, the script ensures that both the training and validation sets have corresponding numerical values for their respective string labels.

Loading and Splitting Audio Dataset

This script loads an audio dataset from a directory structure and splits it into training and validation sets. The dataset consists of .wav files organized into subdirectories, where each folder represents a different label (command).

Key steps include:

- **Loading the dataset:** The load_dataset() function iterates through the subdirectories in the dataset folder, appending each file's path and its corresponding label (folder name) to the dataset and labels lists.
- **Splitting the dataset:** Using train_test_split, the dataset is randomly split into training and validation sets, with an 80/20 ratio, while ensuring the label distribution remains consistent using stratify.

The script finally prints the number of samples in the training and validation sets.

Dataset Analysis: Number of Audio Files per Command

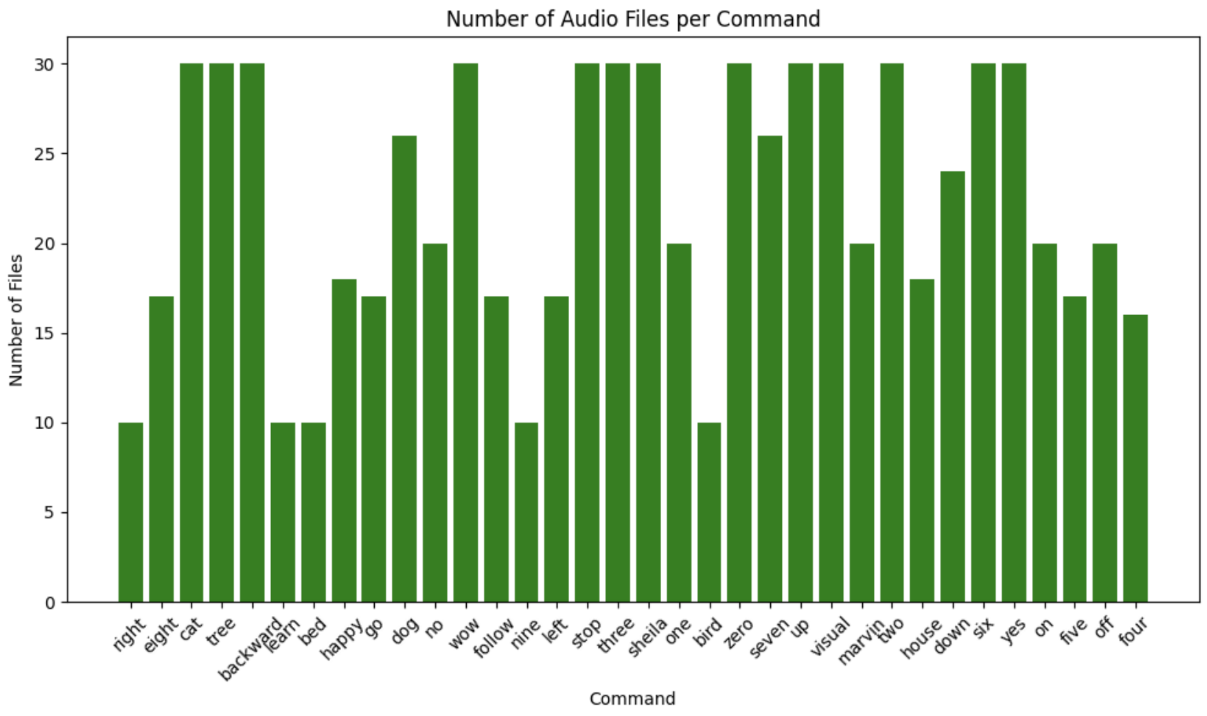
This analysis examines the distribution of audio files across different commands (labels) in the dataset. Each command represents a category of audio, and the number of .wav files in each command folder is counted and visualized.

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 well-balanced 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 for spectrograms of shape (128, 32, 1)—128 Mel-frequency bands, 32-time steps, and 1 channel.
    - Two convolutional layers (Conv2D) with 32 and 64 filters, both followed by MaxPooling2D layers, 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 is used to learn higher-level features.
    - The final output layer has a number of units equal to the number of commands (determined by len(label_encoder.classes_)), with a softmax activation function for multi-class classification.
2. **Compilation:**
    - The model is compiled using the Adam optimizer, sparse categorical cross-entropy 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:

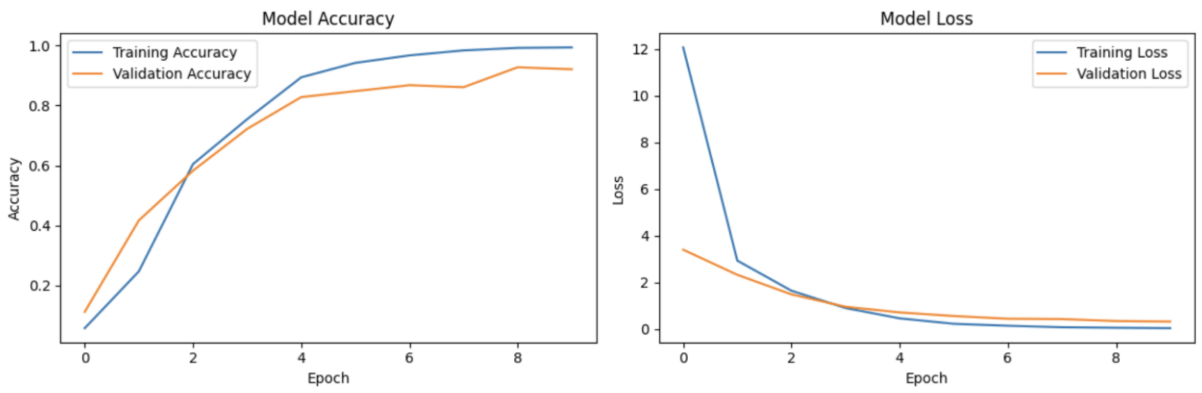
1. **Training:**
    - The model.fit() function is used to train the model.
    - train_ds[0] and train_ds[1] provide the training data (spectrograms) and labels, respectively.
    - The training is conducted over 10 epochs.
    - The model's performance is validated using val_ds[0] and val_ds[1] for validation data and labels.

This setup allows the CNN model to learn from the training data and evaluate 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.