Technical Documentation: Audio Classification for Aircraft Sounds

September 4, 2025

1 Introduction

This document provides comprehensive technical documentation for a Python script designed for multi-class audio classification. The system's primary goal is to analyze audio clips of aircraft and classify them based on various attributes, including engine type, number of engines, and fuel type. The core methodology involves converting raw audio signals into a visual representation known as a Mel spectrogram, which is then used as input to a Convolutional Neural Network (CNN) for classification.

The script is modular, allowing for training different classification tasks by simply changing a single variable. This documentation will explain the purpose and function of each code block and its role in the overall machine learning pipeline.

2 Dataset Loading and Task Configuration

The initial section of the script handles the loading of the dataset metadata and defines the classification tasks. The system is designed to classify audio into four categories:

1. `is_aircraft`: A binary classification task to determine if the audio contains an aircraft or not.
2. `engtype`: A multi-class classification task to identify the engine type (Turbofan, Turboprop, Piston, Turboshaft).
3. `engnum`: A multi-class classification task to identify the number of engines (1, 2, or 4).
4. `fueltype`: A binary classification task to identify the fuel type (Kerosene or Gasoline).

- `df = pd.read_csv('dataset/sample_meta.csv')`: This line loads a CSV file containing metadata about the audio clips. This metadata is crucial as it contains the filename, class labels, and data fold (`train`, `val`, `test`) for each recording.
- `AUDIO_DIR = 'dataset/audio/audio'`: Defines the base directory where all audio files are stored.
- `TARGET_COLUMNS`: A dictionary mapping user-friendly task names (`'is_aircraft'`, `'engtype'`, etc.) to their corresponding column names in the metadata DataFrame. This structure allows for easy switching between different classification problems.
- `CURRENT_TASK`: A variable that dictates which classification task the script will execute. The script's behavior, from label preparation to model architecture, is dynamically adjusted based on the value of this variable.

3 Feature Extraction and Preprocessing

The core data preparation stage of the pipeline involves converting raw audio signals into Mel spectrograms and preparing them for the neural network.

The process is governed by several pre-defined constants that ensure a consistent output shape for the neural network. These include:

- `SR`: Sample rate (`22050` Hz)
- `DURATION`: Segment length (`5` seconds)
- `SAMPLES_PER_SEGMENT`: Total samples per segment (`SR * DURATION`)
- `N_FFT`: FFT window size (`2048`)
- `HOP_LENGTH`: Hop size (`1024`)
- `N_MELS`: Number of Mel bins (`128`)
- `EXP_INPUT_SHAPE`: Expected spectrogram shape `(128, 109)`

The `audio_to_spectrogram` function takes an audio file, segments it into `5`-second clips, and generates a Mel spectrogram for each segment using `librosa.feature.melspectrogram`. It includes validation for the sample rate and zero-padding to ensure a uniform output shape. The `normalise_array` function then applies a min-max scaling to convert spectrogram values to a range between 0 and 1, a crucial step for optimal neural network training. Finally, the `preprocess` wrapper function automates the entire feature extraction pipeline for a list of audio files, returning the features (`X`) and labels (`y`) as NumPy arrays.

3.1 Mathematical Foundations of the Mel Spectrogram

The conversion of an audio signal into a Mel spectrogram involves several key mathematical transformations. The process begins by dividing the time-domain signal into short, overlapping frames and performing a spectral analysis on each frame.

3.1.1 Discrete Fourier Transform (DFT)

The process begins with the **Short-Time Fourier Transform (STFT)**, which is an application of the **Discrete Fourier Transform (DFT)**. The DFT transforms a time-domain signal into the frequency domain. For a discrete signal $x[n]$ of length N, its DFT, $X[k]$, is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}$$

where:

- $x[n]$ is the audio signal sample.
- k is the frequency bin.
- N is the number of samples in the segment (the `n_fft` parameter).

3.1.2 Power Spectrum

The power spectrum, $P[k]$, is calculated from the magnitude of the STFT output. For a given frequency bin k, it's simply the squared magnitude:

$$P[k] = |X[k]|^2$$

3.1.3 Mel Scale

The human ear perceives pitch on a non-linear scale. Low frequencies are perceived with much higher resolution than high frequencies. The **Mel scale** approximates this logarithmic perception. The formula to convert frequency f (in Hz) to the Mel scale is:

$$m = 2595 \cdot \log_{10}(1 + \frac{f}{700})$$

The `librosa` function creates a series of overlapping triangular filters in the frequency domain, with their center frequencies and bandwidths spaced according to the Mel scale. The power from the linear frequency bins is then summed within each Mel filter.

3.1.4 Decibel (dB) Conversion

The final step is converting the power spectrum to a **decibel scale**. This is a logarithmic scale that compresses the large range of power values into a more manageable range. The formula used is:

$$dB_{power} = 10 \cdot \log_{10}(P/P_{ref})$$

The final output is a 2D array where each value represents the loudness (in dB) of a specific Mel frequency band at a specific point in time.

4 Model Training and Evaluation

4.1 Label Preparation

The `prepare_labels_and_weights` function handles the crucial step of preparing the labels for the selected classification task.

- **Multi-class tasks**: It uses `sklearn.preprocessing.LabelEncoder` to convert categorical labels (e.g., 'Turbofan', 'Piston') into numerical integers (`0`, `1`, `2`, ...).
- **Class Weights**: It computes balanced class weights using `sklearn.utils.class_weight.co` This is essential for handling imbalanced datasets (where some classes have significantly fewer examples than others) and preventing the model from becoming biased towards the majority class.

4.2 Dataset Splitting

The script uses a pre-defined folding system from the metadata to split the dataset into training, validation, and test sets.

- **Training Set**: Folds 1, 2, 3, and 4.
- **Validation Set**: Fold 5.
- **Test Set**: 'test' fold.

This split is applied to either all recordings (for `is_aircraft`) or only aircraft recordings (for multi-class tasks), ensuring the model is evaluated on unseen data.

4.3 Model Architecture (`build_model`)

This function defines the Convolutional Neural Network (CNN) architecture. The model is constructed as a `Sequential` stack of layers.

- `Conv2D`: The primary convolutional layers. These layers apply a set of learnable filters (kernels) to the input spectrogram. The `(3, 3)` kernel size means each filter is 3x3 pixels, allowing it to detect local patterns such as changes in frequency or timbre. The number of filters

increases with each layer (32, 64, 128), enabling the network to learn progressively more complex and abstract features. The `relu` activation function ($f(x) = \max(0, x)$) introduces non-linearity, allowing the model to learn more complex relationships.

- `BatchNormalization`: This layer normalizes the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. This technique helps to stabilize and accelerate the training process by preventing internal covariate shift.

- `MaxPooling2D`: This layer performs spatial downsampling. It takes the maximum value from a 2x2 window, effectively reducing the dimensions of the feature maps and making the model more robust to minor shifts or translations in the input.

- `Dropout`: A regularization technique to prevent overfitting. It randomly sets a fraction of input units to 0 at each update during training. This forces the model to learn more robust features that are not dependent on specific neurons, thereby improving its generalization ability.

- `Flatten`: This layer transforms the 2D output of the final convolutional block into a 1D vector. This is necessary because the subsequent `Dense` layers expect a 1D input.

- `Dense`: These are fully connected layers. Every neuron in a dense layer is connected to every neuron in the preceding layer. They are used to perform the final classification based on the features extracted by the convolutional layers.

- `output_units` **and** `activation`: The final `Dense` layer is the output layer.

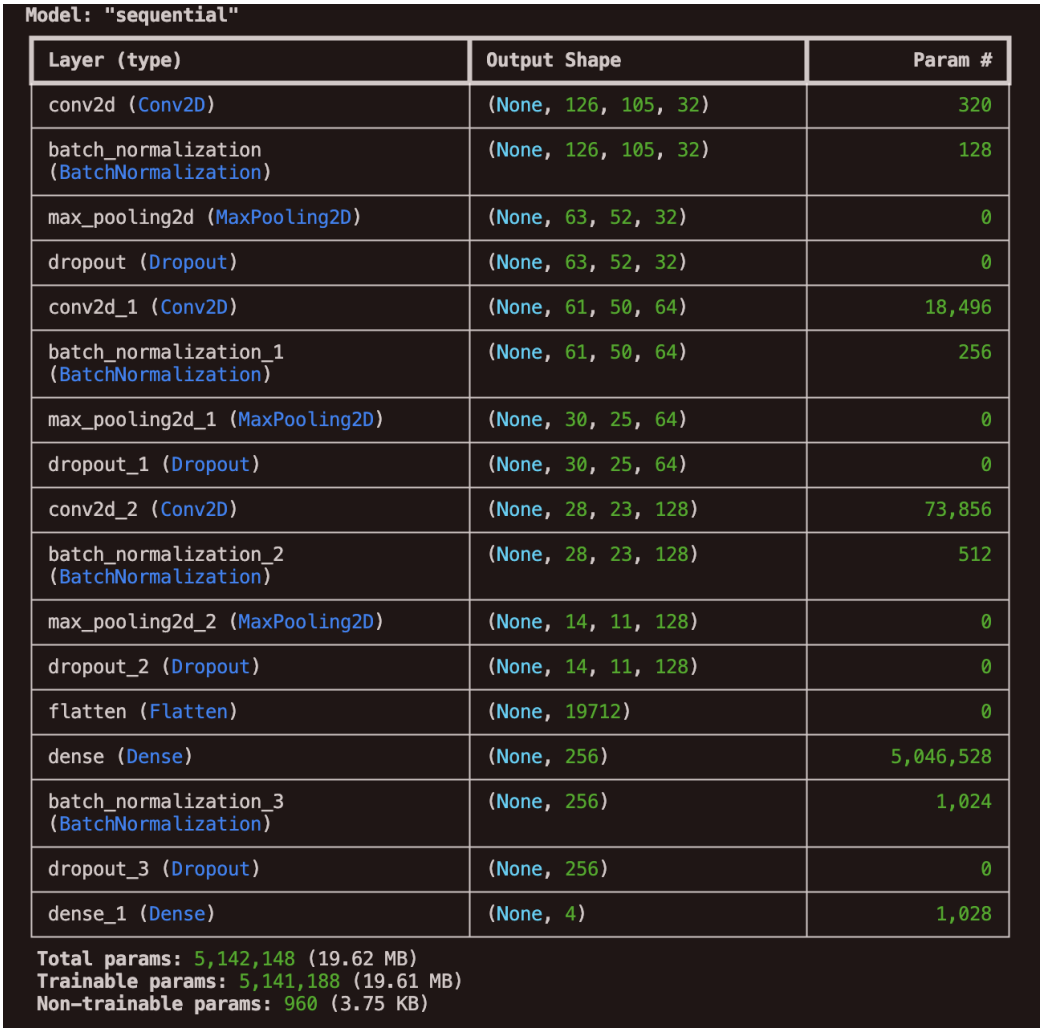
  - For **binary classification** (`is_aircraft` or `fueltype`), `output_units` is 1 and the `activation` is `sigmoid`, which outputs a probability score between 0 and 1.
  - For **multi-class classification** (`engtype` or `engnum`), `output_units` is equal to the number of classes, and the `activation` is `softmax`, which outputs a probability distribution over the classes.

The model is compiled with the `adam` optimizer and the appropriate loss function (`binary_crossentropy` for binary or `sparse_categorical_crossentropy` for multi-class).

4.4 Training and Evaluation

- `model.fit`: The model is trained using the prepared training and validation data.
  - `epochs=50`: The number of training epochs.
  - `batch_size=32`: The number of samples per gradient update.
  - `class_weight=class_weight_dict`: Ensures the model learns from all classes effectively.
  - `callbacks`:
    - `EarlyStopping`: Stops training if the validation loss doesn't improve for 10 epochs, preventing overfitting.
    - `ReduceLROnPlateau`: Reduces the learning rate if the validation loss plateaus, helping the model converge more effectively.
- **Evaluation**: After training, the model is evaluated on the held-out `X_test` dataset.
- **Classification Report**: The `sklearn.metrics.classification_report` provides a detailed breakdown of precision, recall, and F1-score for each class.
- **Confusion Matrix**: `sklearn.metrics.confusion_matrix` provides a visual table of the model's predictions versus the true labels.

A Model Architecture



Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 105, 32)	320
batch_normalization (BatchNormalization)	(None, 126, 105, 32)	128
max_pooling2d (MaxPooling2D)	(None, 63, 52, 32)	0
dropout (Dropout)	(None, 63, 52, 32)	0
conv2d_1 (Conv2D)	(None, 61, 50, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 61, 50, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 30, 25, 64)	0
dropout_1 (Dropout)	(None, 30, 25, 64)	0
conv2d_2 (Conv2D)	(None, 28, 23, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 28, 23, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 14, 11, 128)	0
dropout_2 (Dropout)	(None, 14, 11, 128)	0
flatten (Flatten)	(None, 19712)	0
dense (Dense)	(None, 256)	5,046,528
batch_normalization_3 (BatchNormalization)	(None, 256)	1,024
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 4)	1,028

Total params: 5,142,148 (19.62 MB)
Trainable params: 5,141,188 (19.61 MB)
Non-trainable params: 960 (3.75 KB)

Figure 1: Neural Network Architecture