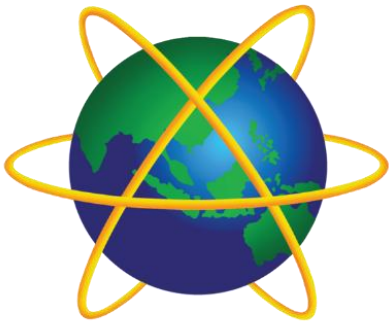

Individual Assignment

TECHNOLOGY PARK MALAYSIA

CT118-3-3-ODL

OPTIMISATION AND DEEP LEARNING

APU3F2502CS(DA)

HAND OUT DATE: 23rd June 2025

HAND IN DATE: 5th October 2025

NAME: SIM SAU YANG

TP NO: TP065596

LECTURER: Assoc. Prof. Dr. Raja Rajeswari A/P Ponnusamy

Table of Contents

Table of Figures

1 Introduction

This report studies the problem of **musical instrument sound classification** using deep learning approaches, including neural networks. The dataset selection, exploratory data analysis (EDA), preprocessing, and justification of suitable neural network models will be discussed, and then moving into implementation, which includes model development, optimization with hyperparameter tuning, and evaluation. In this study, an open-source dataset of labelled instrument sounds is used, and two predictive architectures will be proposed and justified, including **Convolutional Neural Network (CNN)** and a **Recurrent Neural Network (RNN) with Bidirectional Gated Recurrent Units (BiGRU)**. The overall objective is to identify the most effective deep learning approach for accurate instrument classification and show how optimization techniques can enhance model performance.

2 Dataset and Algorithm Selection

2.1 Dataset Selection

For this project, an open musical instrument sound dataset is selected from a public repository, which is Kaggle (source link: *https://www.kaggle.com/datasets/abdulvahap/music-instrunment-sounds-for-classification*). This dataset size is about 5GB, and it contains more than 42,000 audio clips of 28 distinct musical instruments, with each clip being a fixed-length 3-second recording of the instrument (Vahap, 2024). Some instruments include flute, trombone, organ, and more, which show a rich and diverse sample and enough data, which is appropriate for audio classification using deep models, and can reduce overfitting risk. Instrument recognition is a well-known problem in music information retrieval (MIR), where instrument recognition supports automatic annotation of music, content-based recommendation, and musical transcription (Han et al., 2016). There are also past studies using similar datasets, such as IRMAS or NSynth, to explore instrument classification techniques in reality (Borovčak & Babac, 2025).

By focusing on this dataset, the common distinctive frequency pattern and temporal dynamics of musical instrument sounds, such as the flute's harmonic spectrum and the drum's transient attack, can be turned into a digital signal for deep learning algorithms to exploit. This dataset's advantage is the uniform three-second window, which balances with computational tractability for GPU training and batching. However, the challenges that must be addressed in the

experimental design, such as class imbalance, can bias accuracy (can consider using F1-score), and data leakage can occur if near-duplicate clips or stems from a common source span both train and test (need to stratify splits carefully). In short, the dataset's scale, label diversity, and standardized duration make it a strong fit for deep learning approaches to instrument recognition for this project.

2.2 Predictive Neural Network Models Selection

The nature of the data is time-series audio signals, and two deep learning architectures are proposed, which are 2D CNN and RNN with BiGRU. These algorithms are chosen because they can handle specific requirements in audio analysis, like CNNs, which can extract local spectral features that are good at learning local and shift-tolerant patterns (Dorfler et al., 2017), while RNNs can operate on frame sequences that summarize longer-range temporal dynamics (Nandi, 2025).

2.2.1 CNN Model

CNN is a deep learning architecture originally developed for image recognition that applies convolutional filters to automatically extract hierarchical features from input data (Wieclaw, 2025). It becomes a suitable choice for the music instrument sound dataset because the audio signals can be transformed into log-Mel or STFT spectrogram images (Park & Lee, 2015). By converting each audio clip into a time-frequency representation spectrogram, the classification task becomes visually oriented, allowing the CNN to learn discriminative patterns from the images. Unlike traditional manual feature engineering, like extracting Mel-frequency Cepstral Coefficients (MFCCs) or spectral peaks, CNNs automatically learn the optimal features directly from the raw data (Kumar et al., 2023). In predominant instrument settings research by Han et al. (2016), CNNs have significantly improved over earlier baselines on real-world polyphonic audio, achieving higher F-scores and robustness to onset or type variation.

CNN-based methods have been used by many researchers and have gotten good results by leveraging spectrogram inputs. Giri and Radhitya (2024) trained CNN on a smaller Kaggle instrument dataset with classes including piano, violin, drum, and guitar, and achieved an accuracy of about 72%. Borovčak and Babac (2025) also found that deeper CNN architectures like ResNet and DenseNet can perform strongly on instrument recognition, by using the

IRMAS dataset, showing DenseNet121 had the highest F1-score of 0.62. Su (2023) trained the instrument dataset using various types of models, and CNN had the highest accuracy of 96.82%, performing better than ANN and RNN. In summary, CNN should be well-suited to this dataset because of its characteristic of converting the instrument classification into an image recognition task to show its strengths in spatial feature extraction.

2.2.2 RNN with BiGRU Model

The second proposed neural network model for this dataset is RNN, specifically BiGRU. RNN is a deep learning model designed for sequential data, where its recurrent connections allow the network to retain information from previous steps (Keary, 2025), and the Gated Recurrent Unit (GRU) is an improved variant that simplifies this process while effectively capturing temporal dependencies (*Bidirectional Gated Recurrent Unit*, 2024). So, it becomes a logical choice for raw audio or feature sequences over time. In this project, instead of treating the audio as an image for CNN, another reasonable way is to use it as a time series of acoustic feature vectors, like a sequence of MFCC frames or spectral embeddings. The GRU provides comparable performance on sequence tasks with faster training and less risk of overfitting (Sharma, 2025). Together with the bidirectionality, it is advantageous in offline clip classification because the model can condition both past and future frames, which not only rely on the earlier cues, but also consider later cues (*What Is BiGRU Explained?*, 2025) to capture temporal dynamics like attack-sustain-decay patterns or rhythm, which might be crucial for certain instrument sounds.

Past literature research by Niu (2022), who worked on music emotion recognition using the GTZAN and ISMIR2004 dataset, had achieved the highest accuracy of 74.5% using BiGRU with self-attention. For music genre classification on the GTZAN S2n1 dataset, using GRU can reach an accuracy of 95.81% (Ba et al., 2025). Another music genre recognition using spectrogram features is evaluated on the combination of CNN with various algorithms such as LSTM, BiLSTM, GRU, and BiGRU, and reported that the CNN+BiGRU variant gave the highest accuracy of 89.3% (Ashraf et al., 2023).

2.3 Similar System

Author	Research	Task	Dataset	Model	Result	Take note
Chen et al. (2025)	Instrument Sound Classification Using a Music-Based Feature Extraction Model Inspired by Mozart's Turkish March Pattern	Instrument classification	Kaggle Musical Instrument Sounds for Classification, same dataset as this study	custom music-based feature extraction model inspired by rhythmic and melodic patterns from Mozart's Turkish March, combined with a CNN classifier	98.76% accuracy on validation, which is improved over plain CNN about 96.8% and other ML baselines	Introduced a handcrafted feature pattern inspired by classical motifs to guide CNN learning. Very high accuracy
Giri and Radhitya (2024)	Musical Instrument Classification using Audio Features and CNN	Instrument classification	PPMI with 12 instruments, about 9k samples, 44.1 kHz/16-bit	MFCC features into a CNN	Validation accuracy is 71.8%, macro-F1 is 0.677	12-class set with MFCC-only pipeline, and performance is just okay.
Su (2023)	Instrument Classification Using Different Machine Learning and Deep Learning Methods	Instrument classification	Philharmonia, with 7 instruments that are curated from monophonic samples	Spectral features with CNN and RNN baselines	CNN has 96.82% accuracy, RNN 94.83% has accuracy	Use a small and clean dataset, CNN is mentioned as the fastest and most accurate in this setup.

Ashraf et al. (2023)	A Hybrid CNN and RNN Variant Model for Music Classification	Music genre classification (same pipeline)	GTZAN dataset with 10 genres, has 30-s clips, and is resampled/sliced for training	Mel-spectrograms or MFCCs, then CNN + RNN variants	Best with CNN+BiGRU on Mel-spec with 89.30% accuracy and F1 of 0.88	It shows that bidirectional GRU/LSTM performs best, and hybrid improves temporal modelling.

Table 1: Similar System

Overall, there are so many studies saying their model, including CNN and RNN-based architectures, are highly effective for musical instrument sound classification. Results across various datasets show differences in accuracy levels, particularly when combining handcrafted or hybrid features. The high performance achieved using the same Kaggle dataset in that research supports its suitability for this study's proposed algorithm, so let's move on to implementation to see and prove it.

2.4 Summary

Overall, the Kaggle musical instrument dataset provides a balanced combination of scale, diversity, and standardized formatting that is suitable for training deep learning models. Its relevance to music information retrieval tasks makes sure the findings are practically meaningful. The two selected models, CNN and RNN-BiGRU, have their own strengths. CNNs can perform well at extracting local spectral features from spectrograms, while BiGRUs can capture temporal dynamics across frames. Past research also shows the effectiveness of both models in music and audio classification. This selection allows the study to evaluate different perspectives on instrument recognition and identify optimized solutions for the problem.

3 Data Preparation

After identifying the problem and dataset, and having a clear direction on the selected models, the next step is to prepare the data so that it can be used effectively for deep learning. This section covers the process from the initial setup to the preprocessing of the audio files, including exploring the dataset through EDA to understand its distribution and characteristics. The raw audio needs to be transformed into suitable input formats and standardized to ensure consistency. Lastly, the dataset is split into training, validation, and testing sets, and it will be ready for implementation and model development.

3.1 Setup

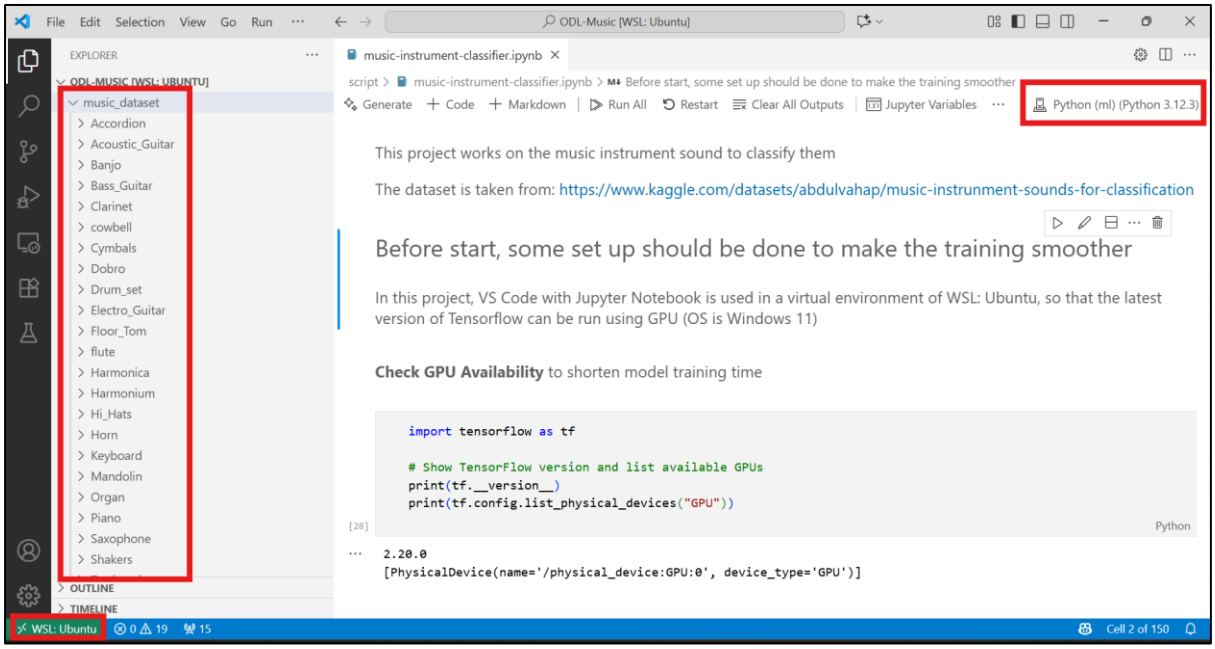

Figure 1: Set Up Programming Environment

This project is carried out using Visual Studio Code (VS Code) with Jupyter Notebook (.ipynb) files as the working environment. The kernel is configured through a Python virtual environment built on WSL Ubuntu (to enable GPU on TensorFlow). After that, the dataset is downloaded in .zip format from Kaggle and extracted into the project folder to make the audio files accessible for preprocessing. The environment is ready for coding and further data preparation tasks.

```
import tensorflow as tf

# Show TensorFlow version and list available GPUs
print(tf.__version__)
print(tf.config.list_physical_devices("GPU"))
```

```
2.20.0
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Figure 2: Check GPU Availability

Firstly, TensorFlow is imported to confirm that the most important deep-learning framework is installed and working. The TensorFlow version is printed. Next, TensorFlow lists physical GPU devices available to the kernel to verify that the system, including WSL, drivers, CUDA, and cuDNN, is correctly exposing a GPU to TensorFlow. The output shows that GPU acceleration is enabled. The purpose of this setup is to greatly speed up spectrogram processing and model training rather than using a CPU.

```
    # setting memory growth for GPUs
    for gpu in tf.config.list_physical_devices('GPU'):
        tf.config.experimental.set_memory_growth(gpu, True)
        print('GPU setup successful!')

GPU setup successful!
```

Figure 3: Enable GPU Memory Growth

This step configures TensorFlow to use GPU memory growth, which means it will only allocate memory as needed instead of reserving all at once. This helps to avoid out-of-memory issues.

3.2 Dataset Class Distribution and Balance

```
    from pathlib import Path
    import pandas as pd

    # Dataset root folder, each subfolder is presenting one class
    DATA_DIR = Path("../music_dataset/").resolve()

    # Collect subfolders
    class_dirs = sorted([p for p in DATA_DIR.iterdir() if p.is_dir()])

    # Build a dataframe table, class name and count
    df_counts = pd.DataFrame(
        [(d.name, sum(1 for _ in d.glob("*.wav"))) for d in class_dirs],
        columns=["class", "count"]
    ).sort_values("count", ascending=False).reset_index(drop=True)

    print(f"Total classes: {len(df_counts)}")
    print(f"Total wavs: {int(df_counts['count'].sum())}")

Total classes: 28
Total wavs: 42311
```

Figure 4: List Classes and Count Audio Files

Then, data exploration and processing begin. Firstly, the dataset folder is scanned, and each subfolder is treated as a class. The code then counts how many .wav files belong to each class and puts them into a data frame table. The output shows that the music instrument sound dataset consists of 28 classes with a total of 42,311 clips.

```
display(df_counts)
```

	class	count
0	flute	3719
1	Acoustic_Guitar	3654
2	Drum_set	3648
3	Bass_Guitar	3613
4	Accordion	3581
5	Banjo	2998
6	Trombone	2965
7	Mandolin	2458
8	Keyboard	2041
9	Organ	1442
10	Shakers	1357
11	Electro_Guitar	1316
12	Harmonium	1314
13	Horn	1258
14	Ukulele	790
15	Clarinet	634
16	Violin	630
17	cowbell	621
18	Piano	575
19	Tambourine	558
20	vibraphone	506
21	Trumpet	503
22	Dobro	487
23	Saxophone	454
24	Hi_Hats	444
25	Floor_Tom	406
26	Cymbals	208
27	Harmonica	131

Figure 5: List All Classes and File Count

This table shows how many audio clips each instrument class contains after scanning the dataset folders. It confirms there are 28 classes and 42,311 clips in total, and it also shows a clear class imbalance. For example, flutes and guitars have more than 3,600 clips, while the harmonica has only 131.

```
THRESHOLD = 1000 # minimum clips per class to keep

# Keep only classes with more than 1000 data
df_counts = (
    df_counts[df_counts["count"] > THRESHOLD]
    .sort_values("count", ascending=False)
    .reset_index(drop=True)
)

print(f"Classes with > {THRESHOLD} files: {len(df_counts)}")
```

```
Classes with > 1000 files: 14
```

Figure 6: Filter Classes with More Than 1000 Clips Only

To make the dataset more balanced and computationally manageable, only classes with more than 1,000 audio clips are retained, resulting in a reduced set of 14 instrument classes. This decision addresses the issue and limitation. Firstly, it helps to control training time and computational load, because using all 28 classes and over 42,000 clips can significantly increase resource requirements. Secondly, it prevents extreme class imbalance, because very small classes can bias the model towards larger classes and lower overall accuracy. The trade-off of this approach is that some instrument variety is lost, but focusing on larger classes ensures that the models learn from sufficient examples per class.

```
display(df_counts)
```

	class	count
0	flute	3719
1	Acoustic_Guitar	3654
2	Drum_set	3648
3	Bass_Guitar	3613
4	Accordion	3581
5	Banjo	2998
6	Trombone	2965
7	Mandolin	2458
8	Keyboard	2041
9	Organ	1442
10	Shakers	1357
11	Electro_Guitar	1316
12	Harmonium	1314
13	Horn	1258

Figure 7: List All Filtered Classes and File Count

The updated table has 14 instrument classes that remain after applying the threshold of 1,000 clips per class. Each of these categories now has a sufficient number of audio examples, ranging from 3,719 flute clips to around 1,258 horn clips. This refined dataset not only reduces the computational burden but also ensures that each class has enough training samples to allow the models to learn.

```
import matplotlib.pyplot as plt
import numpy as np

# df_counts: ["class","count"] for classes > 1000
labels = df_counts["class"].tolist()
values = df_counts["count"].to_numpy()

# Palette for both charts
cmap = plt.get_cmap("Set3")
colors = [cmap(i % cmap.N) for i in range(len(labels))]

# Put pie chart at left and bar chart at right
fig, (ax_pie, ax_bar) = plt.subplots(
    1, 2, figsize=(11, 4),
    gridspec_kw={"width_ratios": [1, 2.6]}
)

# Pie to show proportion per class
wedges, texts, autotexts = ax_pie.pie(
    values,
    labels=labels,
    autopct="%1.1f%%",
    startangle=90,
    colors=colors,
    wedgeprops={"linewidth": 0.5, "edgecolor": "white"},
    textprops={"fontsize": 9}
)
ax_pie.set_title("Class Distribution")
ax_pie.axis("equal")

# Bar to show absolute counts per class
x = np.arange(len(labels))
bars = ax_bar.bar(x, values, color=colors)
ax_bar.set_title("Class counts (> 2500 files)")
ax_bar.set_ylabel("clips")
ax_bar.set_xticks(x)
ax_bar.set_xticklabels(labels, rotation=45, ha="right")
ax_bar.bar_label(bars, labels=[str(v) for v in values], padding=3, fontsize=9)

# subtle grid for readability
ax_bar.grid(axis="y", linestyle=":", linewidth=0.6, alpha=0.7)

plt.tight_layout()
plt.show()
```

Figure 8: Plot Class Balance

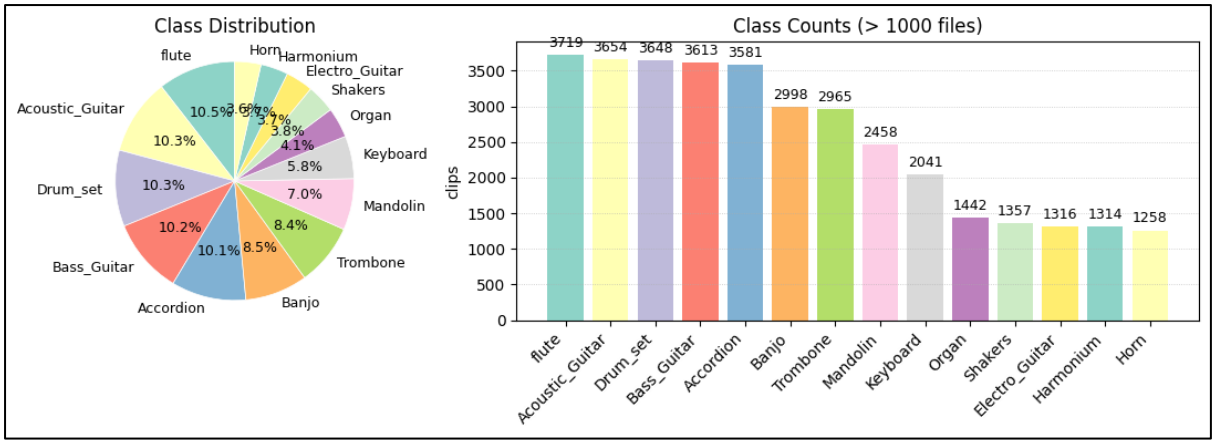

Figure 9: Class Balance Pie and Bar Chart

Figure 9: Class Balance Pie and Bar ChartFigure 9 summarizes the remaining 14 classes after filtering. The pie chart shows each instrument's proportion of the dataset, while the bar chart shows the absolute clip counts per class with value labels. They show a long-tailed distribution, with a few classes like flute, acoustic, bass guitar, drum set, and accordion dominating, while others like organ, shakers, harmonium, and horn are smaller.

3.3 Dataset Clips-Level Exploration and Validation

```
# Make one row per audio file with its class
selected_classes = df_counts["class"].tolist()

rows = []
for c in selected_classes:
    # Collect only .wav files
    for p in (DATA_DIR / c).glob("*.wav"):
        rows.append({"file": p.name, "class": c})

df = pd.DataFrame(rows)

print(f"Rows: {len(df)}")
df.head()
```

```
Rows: 35364
```

	file	class
0	947.wav	flute
1	3031.wav	flute
2	2616.wav	flute
3	2051.wav	flute
4	791.wav	flute

Figure 10: Expand Class Folders to File Level Table

The next step after getting to know basic information about the dataset is to flatten the folder structure into a file-level table so each row represents one audio clip and its class label. The script loops through the selected 14 class folders, collects only *.wav files, and builds a tidy data frame table with two columns, which are file (filename) and class (instrument), yielding 35,364 rows in total. This tabular view makes downstream work much easier, which will be mainly used for the following tasks.

```
import numpy as np
import pandas as pd, soundfile as sf
from tqdm.auto import tqdm

# Helper function: path to the audio file for a dataframe row
def path_of(row):
    return str((DATA_DIR / row["class"] / row["file"]).resolve())

# Ensure metadata columns exist
for col in ["sr", "channels", "duration"]:
    if col not in df.columns:
        df[col] = pd.Series(np.nan, index=df.index, dtype="float64")

# Only process rows with no metadata fields
mask = df["sr"].isna() | df["duration"].isna() | df["channels"].isna()

unreadable_count = 0
for i, row in tqdm(df.loc[mask].iterrows(), total=int(mask.sum())):
    p = path_of(row)
    try:
        # Read only header so no need to load the whole audio into memory
        with sf.SoundFile(p) as f:
            sr = float(f.samplerate)
            ch = float(f.channels)
            dur = float(len(f) / f.samplerate)
        df.at[i, "sr"] = sr
        df.at[i, "channels"] = ch
        df.at[i, "duration"] = dur
    except Exception:
        # Count files that cannot be read
        unreadable_count += 1

print(f"Unreadable: {unreadable_count}")
```

✓ 15.8s

100% 35364/35364 [00:15<00:00, 2320.98it/s]

Unreadable: 0

Figure 11: Read Basic Audio Metadata

Then, the file table is enriched with audio metadata without loading full waveforms. For each clip, it opens the file header with *soundfile* and records the sample rate (sr), channels, and duration (seconds) into new columns, using a fast loop with a progress bar. Only rows missing these fields are processed, and any unreadable files are counted for quality control. The output shows no unreadable files, so the dataset is clean. These fields verify assumptions, such as 3.0s length, consistent sampling rate, spot anomalies, and drive later preprocessing decisions before modelling.

```
df.head()
```

	file	class	sr	channels	duration
0	947.wav	flute	22050.0	1.0	3.0
1	3031.wav	flute	22050.0	1.0	3.0
2	2616.wav	flute	22050.0	1.0	3.0
3	2051.wav	flute	22050.0	1.0	3.0
4	791.wav	flute	22050.0	1.0	3.0

Figure 12: Preview of Expanded Audio Files Table

Figure 12 confirms the metadata is attached correctly for each clip. Each row lists the filename, its class, and basic audio info, including sample rate, mono audio, and duration.

```
display(df[['sr','channels','duration']].describe())
```

	sr	channels	duration
count	35364.0	35364.0	35364.0
mean	22050.0	1.0	3.0
std	0.0	0.0	0.0
min	22050.0	1.0	3.0
25%	22050.0	1.0	3.0
50%	22050.0	1.0	3.0
75%	22050.0	1.0	3.0
max	22050.0	1.0	3.0

Figure 13: View Clips Summary

This summary table shows the descriptive statistics of the audio metadata. All 35,364 clips have exactly the same values with a sample rate of 22,050 Hz, 1 audio channel (mono), and a duration of 3.0 seconds. The standard deviation is 0, and the min, max, and quartile values are identical. This confirms that the dataset is consistent in format. This uniformity is helpful because it means no extra preprocessing steps are needed to standardize the audio before further preprocessing for model training.

3.4 Sample Feature Extraction and Visualization

```
# Feature and processing settings

# Native sample rate to use throughout
sample_rate = 22050
# Fixed clip length so all tensors share the same time dimension
clip_seconds = 3.0
samples_per_clip = int(sample_rate * clip_seconds)

# STFT settings: 1024-point FFT (~46 ms at 22.05 kHz) balances time vs frequency detail
fft_size         = 1024
# Hop length (75% overlap is ~11.6 ms hop) gives smoother spectrograms and better transient capture
hop_length       = 256
# Mel-spectrogram resolution: 64 bins capture timbre detail without heavy compute
num_mel_bins     = 64
# MFCC dimensionality used for music/audio classification
num_mfcc         = 13
```

Figure 14: Set Feature and Preprocessing Constants

Before extracting features, several parameters are defined to ensure consistency across all clips and to balance time–frequency resolution for the models:

- **Sample rate (22,050 Hz):** The dataset already uses 22.05 kHz, so no resampling is needed. According to the Nyquist theorem, this captures frequencies up to 11,025 Hz, which comfortably covers the important timbral range of most instruments while keeping files lighter than those at 44.1 kHz.
- **Clip length (3.0 seconds):** Each file is exactly three seconds long, which has a uniform input length that does not need to perform padding or trimming. Then, the number of raw samples per clip can be calculated as:

  $$22{,}050 \times 3.0 = 66{,}150\ samples\ per\ clip$$

  The raw waveform for one clip, therefore, contains 66,150 samples, which is the time dimension used to convert into a sequence of frames for STFT, Mel, or MFCC.
- **FFT size (1,024 points):** The Fast Fourier Transform (FFT) size determines how long each time window is when breaking the audio into small frames for the Short-Time Fourier Transform (STFT). Choosing the right value is a trade-off between time resolution and frequency resolution (Campos, 2025). In this project, with a sample rate of 22,050 Hz, a 1024-point FFT window can cover:

  $$\frac{1{,}024}{22{,}050} \approx 0.046\ seconds\ (46\ ms)$$

  This is short enough to follow changes in instrument sounds, such as note attacks or percussive hits, but still long enough to resolve harmonic structures. While the frequency resolution is:

  $$\frac{22{,}050}{1{,}024} \approx 21.5\ Hz\ per\ frequency\ bin$$

This window length shows a balance between frequency resolution and time resolution.

- **Hop length (256 samples):** The hop length controls how much the windows overlap and therefore the time resolution of the spectrogram (*Choice of HOP Size | Spectral Audio Signal Processing*, n.d.). At 22,050 Hz, 256 samples:

$$\frac{256}{22{,}050} \approx 0.0116\ seconds\ (11.6\ ms)$$

The spectrogram updates every ~11.6 ms. Compared to the 1024-sample window size, this gives a 75% overlap ($1 - \frac{256}{1{,}024}$), which can help to avoid losing information between frames and create smoother spectrograms. So, the frames per 3-s clip is:

$$1 + \left[\frac{66{,}150 - 1{,}024}{256}\right] \approx 255\ frames$$

- **Mel-spectrogram bins (64):** After computing the STFT, the frequency axis (linear scale) is converted into the Mel scale, which better reflects human hearing sensitivity. With a 22,050 Hz sample rate, the Nyquist frequency is 11,025 Hz, so the Mel filterbank spans from 0 to 11 kHz. So, choosing 64 Mel bins means that the entire spectrum is compressed into 64 perceptually spaced frequency bands. It provides enough detail to represent harmonic structures, noise components, and spectral shape that define timbre.
- **MFCCs (13 coefficients):** The Mel-Frequency Cepstral Coefficients (MFCCs) compress the Mel-spectrogram into a low-dimensional representation that captures overall spectral shape. 13 coefficients are used because it is standard in speech and music classification tasks, as it balances expressiveness and computational cost (Kumar & Yadav, 2024).

In short, these feature settings ensure consistent inputs and strike a balance between time–frequency resolution, with Mel-spectrograms capturing timbre details and MFCCs providing a compact, widely used representation for instrument classification.

```python
import numpy as np, librosa, soundfile as sf
from IPython.display import Audio, display
import ipywidgets as widgets

# one sample per class
one_per_class = (
    df.sort_values(["class", "file"])
      .groupby("class", sort=True)
      .nth(1)
)

def load_mono_fixed(path, sr=sample_rate, length=samples_per_clip):
    # Read audio as float32 and average to mono if stereo
    x, in_sr = sf.read(path, dtype="float32", always_2d=False)
    if x.ndim > 1: x = x.mean(axis=1)
    if in_sr != sr: x = librosa.resample(x, orig_sr=in_sr, target_sr=sr)
    return x[:length] if len(x) >= length else np.pad(x, (0, length - len(x)))

# build widgets for label + audio player per class
items = []
for _, row in one_per_class.iterrows():
    x = load_mono_fixed(path_of(row))
    label = widgets.Label(value=row["class"])
    out = widgets.Output()
    with out:
        display(Audio(x, rate=sample_rate))
    items.append(widgets.VBox([label, out]))

# grid layout to show
grid = widgets.GridBox(
    children=items,
    layout=widgets.Layout(
        grid_template_columns=f"repeat({3}, 1fr)",
        grid_gap="10px 12px",
        align_items="flex-start",
        width="100%"
    )
)

display(grid)
```

Figure 15: Play One Sample per Class

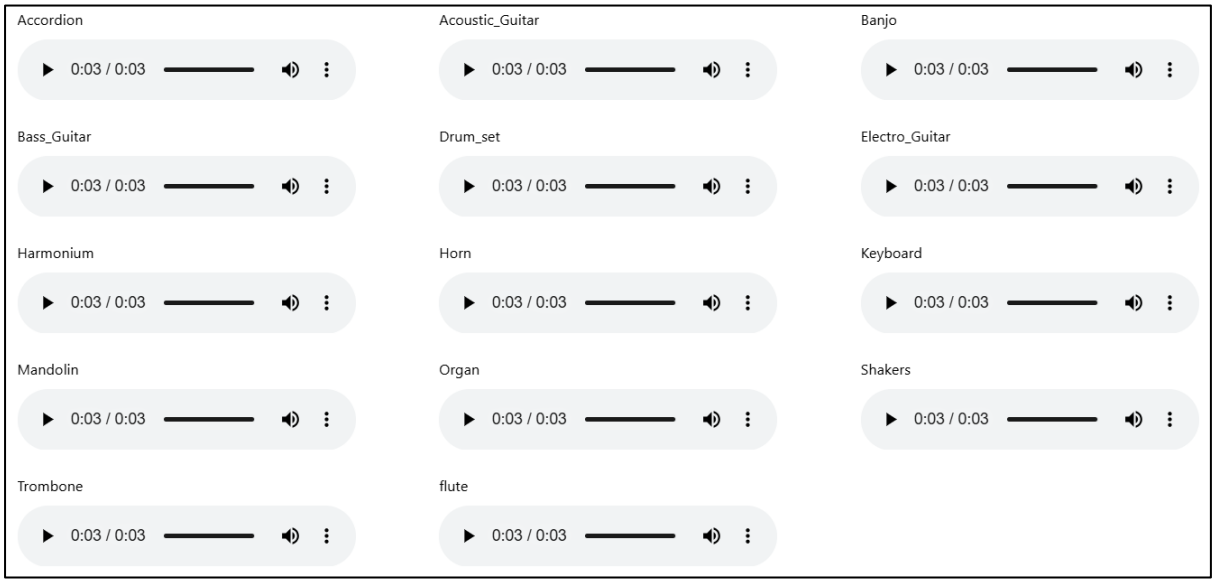

Figure 16: Sample Clip by Class

Next, A listening gallery is created with one representative clip per instrument class for quick verification and later plotting. A single file is selected from each class folder, then loaded as mono at 22,050 Hz to ensure a consistent reference. Each waveform is rendered as an interactive audio player labeled with its class, arranged in a simple grid layout.

```
# Collect features for one sample per class
# Each entry stores raw clip + several time/frequency-domain features
features = []  # list of dicts: {"class","x","freqs","fft_mag","S_db","mel_db","mfcc","chroma"}

for _, row in one_per_class.iterrows():
    x = load_mono_fixed(path_of(row))

    # FFT (global frequency content)
    X = np.fft.rfft(x) # one-sided FFT for real signals
    freqs = np.fft.rfftfreq(len(x), d=1 / sample_rate) # frequency axis (Hz)
    fft_mag = np.abs(X) # magnitude spectrum

    # STFT (time-frequency)
    stft = librosa.stft(x, n_fft=fft_size, hop_length=hop_length)
    S_power = np.abs(stft)**2 # power spectrogram
    S_db = librosa.power_to_db(S_power, ref=np.max) # dB scale for visualization

    # Log-Mel spectrogram (perceptual frequency scale)
    mel = librosa.feature.melspectrogram(
        y=x, sr=sample_rate, n_fft=fft_size,
        hop_length=hop_length, n_mels=num_mel_bins, power=2.0
    )
    mel_db = librosa.power_to_db(mel, ref=np.max)

    # MFCCs (compact timbre descriptor)
    mfcc = librosa.feature.mfcc(S=mel_db, n_mfcc=num_mfcc) # Using log-Mel (dB) as input to DCT for MFCCs

    # Chromagram (pitch class energy)
    chroma = librosa.feature.chroma_stft(S=np.abs(stft), sr=sample_rate)

    # Store result
    features.append({
        "class": row["class"], "x": x,
        "freqs": freqs, "fft_mag": fft_mag,
        "S_db": S_db, "mel_db": mel_db,
        "mfcc": mfcc, "chroma": chroma
    })
```

Figure 17: Precompute Diagnostic Features

The next step is to extract representative features from one sample of each instrument class, which covers both time-domain and frequency-domain representations that are commonly used in music information retrieval. The process includes:

- **Raw waveform (x):** The original three-second audio signal is kept as the baseline reference.
- **Fast Fourier Transform (FFT):** A one-sided FFT is applied to the waveform to calculate the global frequency content. The frequency axis (*freqs*) is constructed in Hertz, and the magnitude spectrum (*fft_mag*) is obtained. This shows which frequencies are present in the clip overall, but it does not show how they change over time.

- **Short-Time Fourier Transform (STFT):** STFT is calculated using the defined FFT size and hop length to capture how frequencies evolve. The squared magnitude gives the power spectrogram, which is then converted to decibel (dB) scale (*S_db*) for visualization, which can provide a detailed time-frequency map that shows note onsets, sustain, and transient patterns.
- **Log-Mel spectrogram:** The STFT is further mapped onto the Mel scale using 64 Mel filter bands (*mel_db*). This compresses the frequency axis into perceptually relevant bands. It becomes easier to distinguish instrument timbre in a way aligned with human hearing. The dB scaling highlights the relative intensity across frequencies.
- **MFCCs:** 13 Mel-Frequency Cepstral Coefficients are extracted from the log-Mel spectrogram. These coefficients summarize the spectral envelope of the sound. It provides a compact representation of timbre.
- **Chroma features:** Chromagram is computed from the STFT magnitude. This representation folds the spectrum into 12 bins corresponding to the musical pitch classes (C through B). It shows the relative energy of each pitch class over time.

Lastly, all these features are stored together in a dictionary alongside the class label, which can later be visualized side-by-side.

```
import math
import matplotlib.pyplot as plt

# 4 columns x enough rows to fit all classes
cols = 4
rows = math.ceil(len(features) / cols)
fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows), sharey=True)
axes = axes.ravel()

for i, f in enumerate(features):
    ax = axes[i]
    librosa.display.waveshow(f["x"], sr=sample_rate, ax=ax) # time-domain signal
    ax.set_title(f['class'], fontsize=9)
    ax.set_xlabel(""); ax.set_ylabel("")

for j in range(i+1, len(axes)): axes[j].axis("off")

fig.suptitle("Waveforms", y=1.02)
plt.tight_layout(); plt.show()
```

Figure 18: Plot Waveforms

After that, a grid of waveform plots is arranged for one per instrument class to give a quick visual check of the raw time-domain signals. Showing waveforms side by side helps verify that clips load correctly, are audible, and have similar amplitude ranges (no clipping or near-silence). A consistent y-axis and compact layout can also easily spot anomalies across classes briefly.

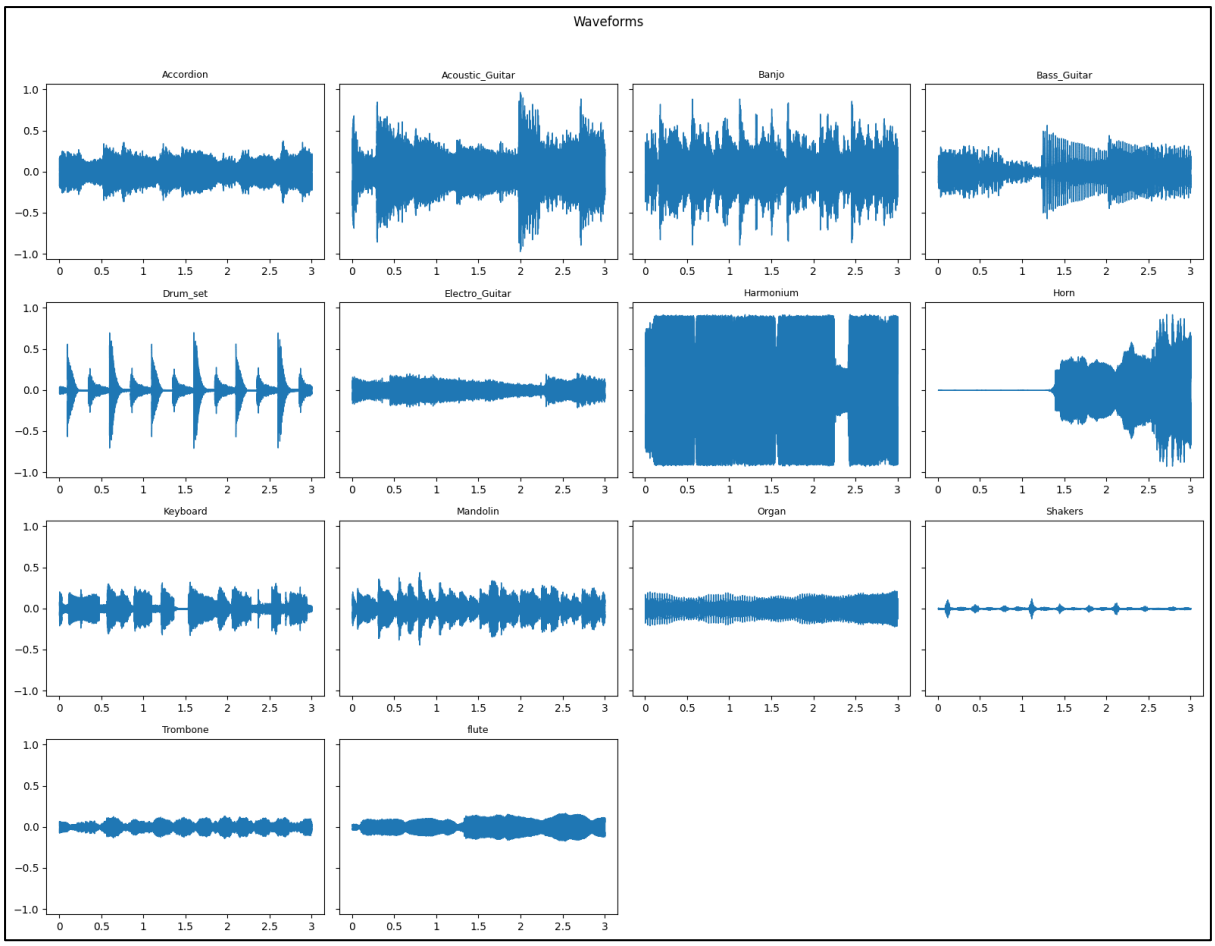

Figure 19: Sample Waveforms of Each Class

Figure 19 shows the waveform is successfully plotted, with one example from each of the 14 instrument classes. Each plot shows the flow of signal amplitude changes over the three-second clip. They have different temporal shapes and energy patterns across instruments. Percussive sounds such as the drum set or shakers display clear spikes or pulses, reflecting their sharp transients. Sustained instruments like the accordion and harmonium show more continuous energy across the full duration. Instruments like a horn or trombone show gradual build-ups in energy, while plucked strings like banjo or mandolin show repeated bursts corresponding to strums or picking. Overall, the visualization confirms that the dataset clips are consistent in length and amplitude range and shows the natural variety in the signal.

```
fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows))
axes = axes.ravel()

for i, f in enumerate(features):
    ax = axes[i]
    ax.plot(f["freqs"]/1000.0, f["fft_mag"]) # magnitude vs frequency (kHz)
    ax.set_title(f['class'], fontsize=9)
    ax.set_xlim(0, sample_rate/2000.0)  # show up to Nyquist (sr/2) in kHz
    ax.set_xlabel("kHz"); ax.set_ylabel("")

for j in range(i+1, len(axes)): axes[j].axis("off")

fig.suptitle("FFT Magnitude", y=1.02)
plt.tight_layout(); plt.show()
```

Figure 20: Plot FFT Magnitude

FFT magnitude spectrum is plotted for one clip from each instrument class to show how energy is distributed across frequency (x-axis in kHz) up to the Nyquist limit (sr/2). The display will show broad timbral differences. Comparing spectra across classes can help verify sensible frequency content, show which bands carry discriminative information, and motivate later choices for Mel scaling and model features.

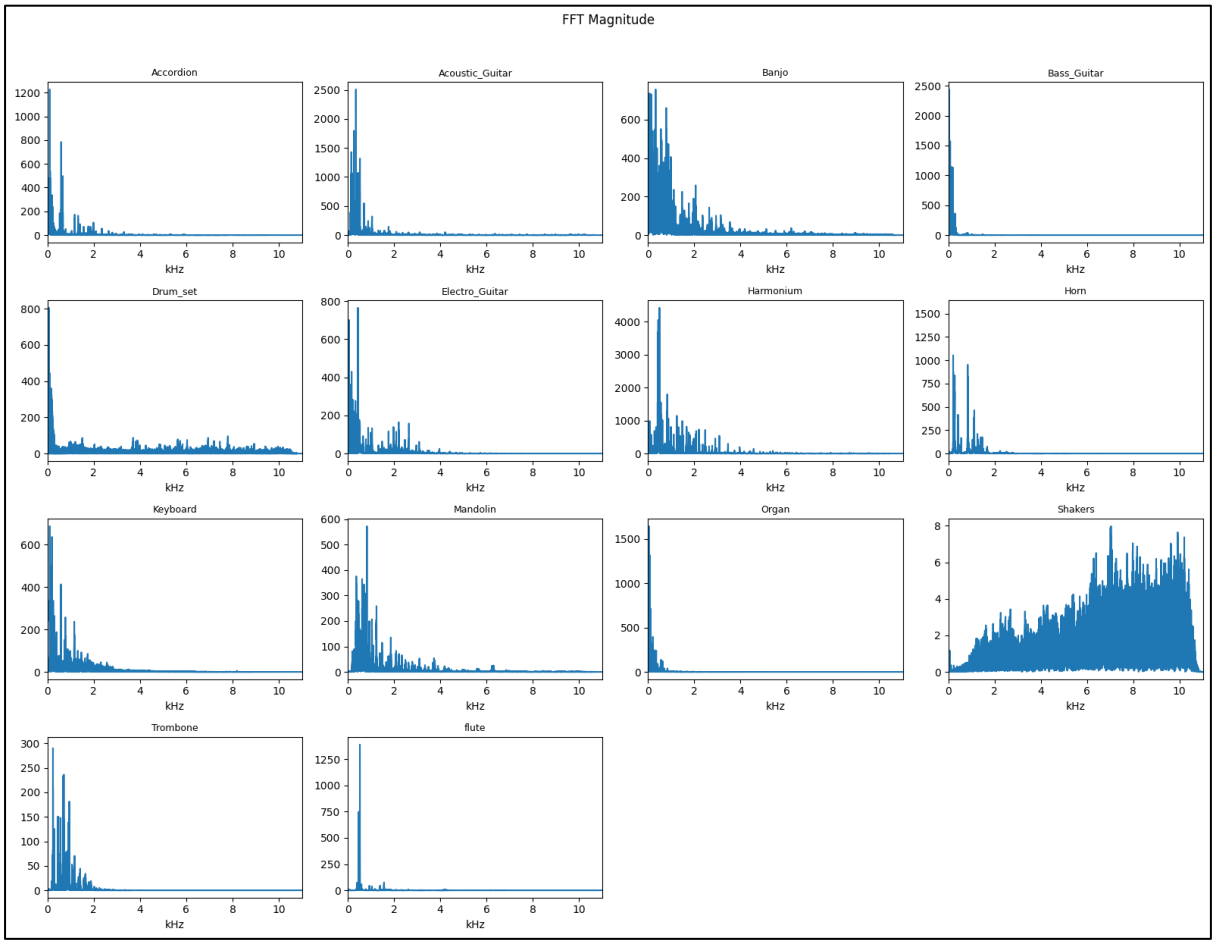

Figure 21: Sample FFT Magnitude of Each Class

Figure 21 shows the frequency spectra of one clip from each instrument class, about how their sound energy is spread across different frequencies. Instruments with clear pitches, such as flute, trombone, and acoustic guitar, display sharp peaks that represent their fundamental tones and harmonics. Sustained instruments like the accordion and harmonium show dense clusters of strong low-frequency harmonics, while the bass guitar is concentrated mostly in the lower range. In contrast, percussive instruments such as drum set and shakers produce much broader and noisier spectra, with shakers spreading energy widely into the higher frequencies. These differences become the unique characteristic of each instrument.

```
fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows))
axes = axes.ravel()

for i, f in enumerate(features):
    ax = axes[i]
    librosa.display.specshow(
        f["S_db"], # power spectrogram in dB
        sr=sample_rate,
        hop_length=hop_length,
        x_axis="time", # time on x-axis
        y_axis="linear", # linear frequency on y-axis
        ax=ax)
    ax.set_title(f['class'], fontsize=9)

for j in range(i+1, len(axes)): axes[j].axis("off")

fig.suptitle("STFT Spectrogram (power dB)", y=1.02)
plt.tight_layout(); plt.show()
```

Figure 22: Plot STFT Spectrogram

The next plot shows the STFT spectrograms (in decibels), with time on the x-axis and linear frequency on the y-axis. Each panel visualizes how energy changes across time and frequency, revealing onsets, sustains, and decays that are not visible in a single FFT. The decibel scaling can highlight strong components and suppress very low energy regions, making timbral patterns easier to compare across classes.

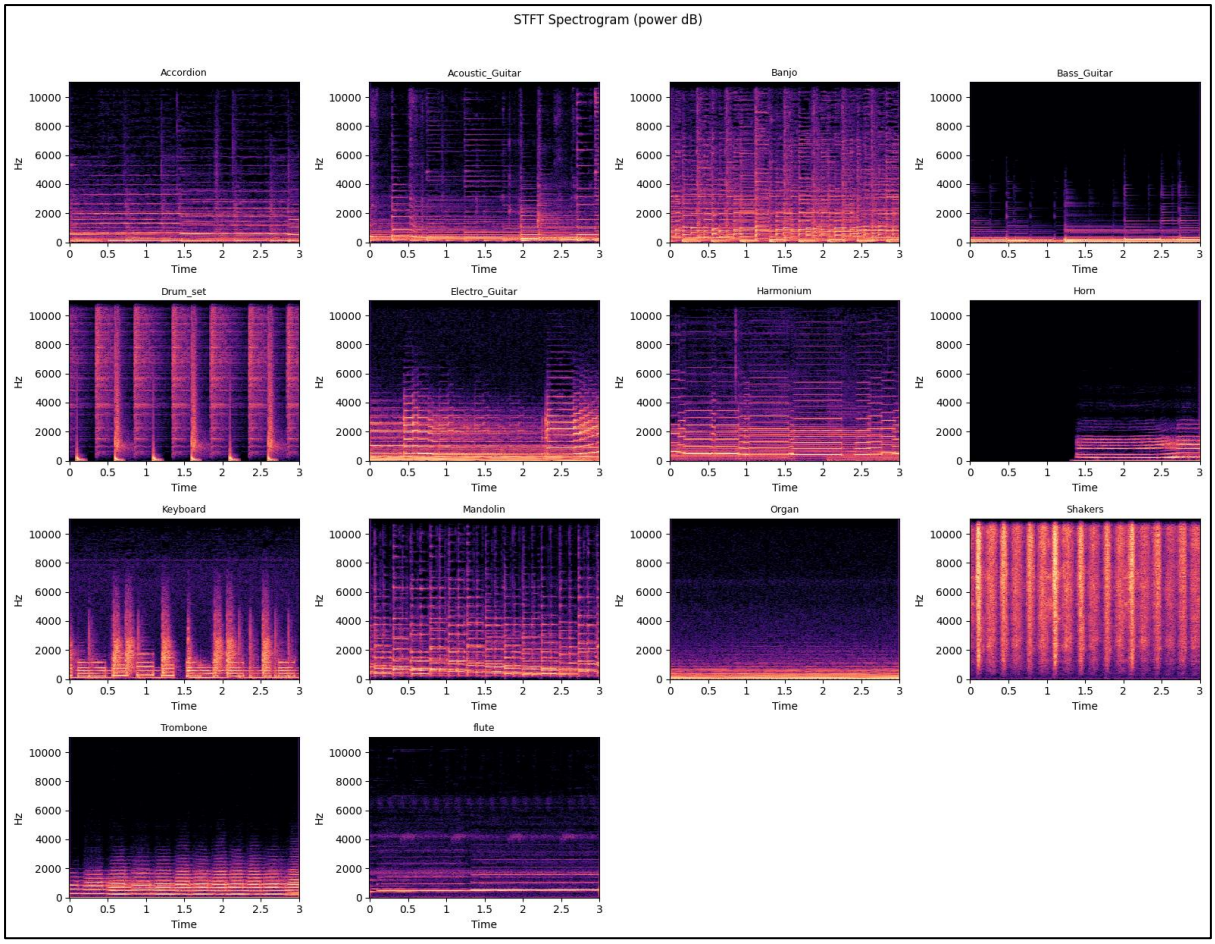

Figure 23: Sample STFT Spectrogram of Each Class

The STFT spectrograms show how sound energy changes over both time and frequency. Compared to the previous FFT plots, which only displayed the overall frequency content, the spectrograms reveal the dynamic patterns of the instruments. Sustained instruments such as accordion and harmonium show continuous horizontal bands of harmonics, while string instruments like banjo and mandolin exhibit repeated strumming patterns as dense vertical textures. Percussive sounds such as drum set and shakers appear as sharp bursts spread across a wide frequency range, reflecting their transient nature. Instruments with clear pitch, such as the flute and trombone, display distinct horizontal lines corresponding to their harmonics.

```
# maps linear FFT bins to Mel bands
mel_filterbank = librosa.filters.mel(
    sr=sample_rate,
    n_fft=fft_size,
    n_mels=num_mel_bins,
    norm=None)

plt.figure(figsize=(7, 3))
librosa.display.specshow(mel_filterbank, x_axis="linear", y_axis=None)

plt.title(f"Mel Filterbank (n_mels={num_mel_bins}, n_fft={fft_size}, sr={sample_rate})")
plt.xlabel("FFT bin"); plt.ylabel("Mel filter index")
plt.tight_layout()
plt.show()
```

Figure 24: Plot Mel Filterbank

After visualizing the STFT spectrograms, the Mel filterbank is used to convert linear FFT bins into perceptual Mel bands. Each row represents one Mel filter (there are 64), and each column corresponds to an FFT frequency bin from the 1024-point STFT at 22.05 kHz. The bright values are the weights applied to that bin. The overlapping shapes show how nearby FFT bins are pooled together, with narrower filters at low frequencies and wider filters at high frequencies.

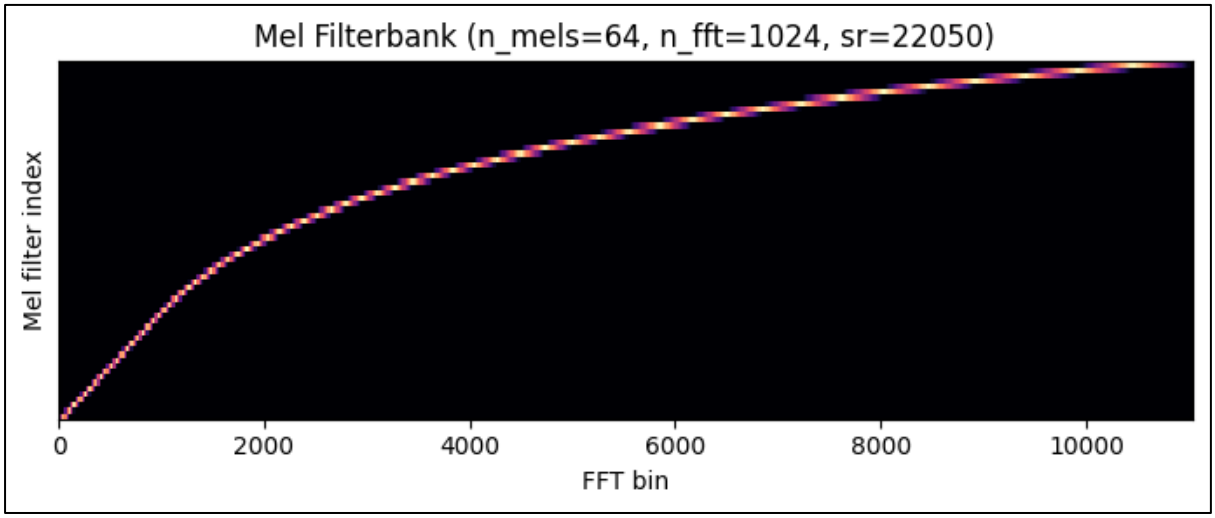

Figure 25: Mel Filterbank

The Mel filterbank reshapes the linear frequency bins from the FFT into 64 Mel bands. The filters are narrow and closely spaced at low frequencies, and wider at higher frequencies. By applying this filterbank, the raw spectrum can be transformed into a smoother representation.

```
# Frequency (Hz) for each FFT bin up to Nyquist
fft_bin_hz = np.linspace(0, sample_rate/2, 1 + fft_size//2)

plt.figure(figsize=(9,3))

# Draw each Mel filter response (rows of the filterbank)
for i in range(num_mel_bins):
    plt.plot(fft_bin_hz, mel_filterbank[i], lw=1.2)

plt.title(f"Mel-Spaced Filterbank ({num_mel_bins} filters)")
plt.xlabel("Frequency (Hz)"); plt.ylabel("Amplitude")
plt.ylim(0, 1.05)
plt.xlim(0, sample_rate/2) # 0 to Nyquist
plt.grid(ls=":", alpha=0.4)
plt.tight_layout()
plt.show()
```

Figure 26: Plot Mel-Spaced Filterbank

The next plot draws each Mel filter as a curve across frequency from 0 to 11 kHz, showing how the 64 filters are densely packed and narrow at low frequencies and wider and more spaced at high frequencies. The overlapping, triangular shapes indicate how energy from neighboring FFT bins is blended when forming Mel bands, which smooths small pitch shifts and reduces noise.

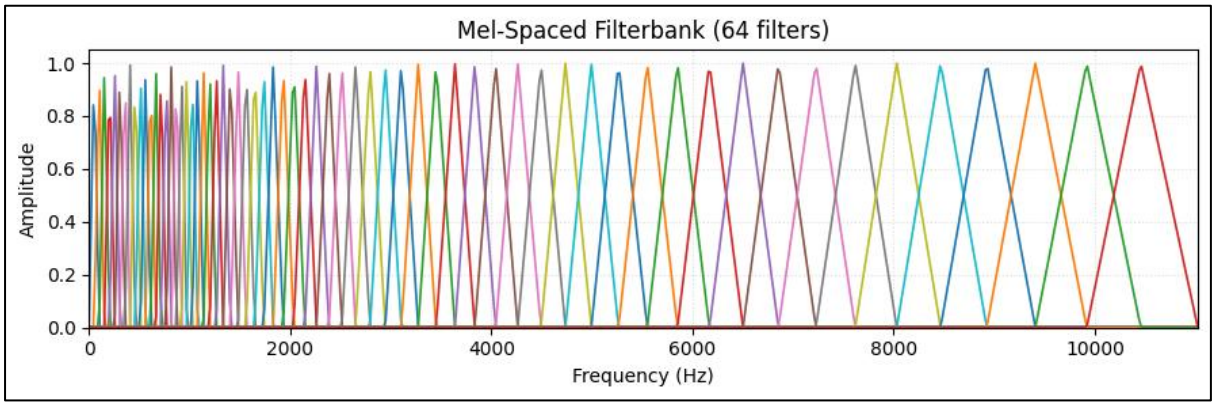

Figure 27: Mel-Spaced Filterbank

The 64 Mel filters are plotted, with each filter shaped like a triangle and overlapping with its neighbors, meaning that energy from adjacent frequency bins is shared smoothly between filters. At the lower frequencies, the filters are densely packed and narrow to reflect human hearing's higher sensitivity in this range, while at higher frequencies, the filters become broader and more widely spaced. This design compresses the detailed FFT spectrum into a perceptually scaled.

```
fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows))
axes = axes.ravel()

for i, f in enumerate(features):
    ax = axes[i]
    librosa.display.specshow(
        f["mel_db"], # log-Mel in dB
        sr=sample_rate,
        hop_length=hop_length,
        x_axis="time", # time on x-axis
        y_axis="mel", # Mel frequency on y-axis
        ax=ax
    )
    ax.set_title(f['class'], fontsize=9)

for j in range(i+1, len(axes)): axes[j].axis("off")

fig.suptitle("Log-Mel Spectrogram (dB)", y=1.02)
plt.tight_layout(); plt.show()
```

Figure 28: Plot Log-Mel Spectrogram

The log-Mel spectrograms are shown with time on the x-axis and Mel-scaled frequency on the y-axis. Unlike the linear STFT plots, the Mel scale groups frequencies the way human hearing perceives them. It produces smoother and more compact patterns that emphasize timbre and reveal class-specific textures. This view is important because log-Mel spectrograms are the standard input for CNN/RNN models in audio.

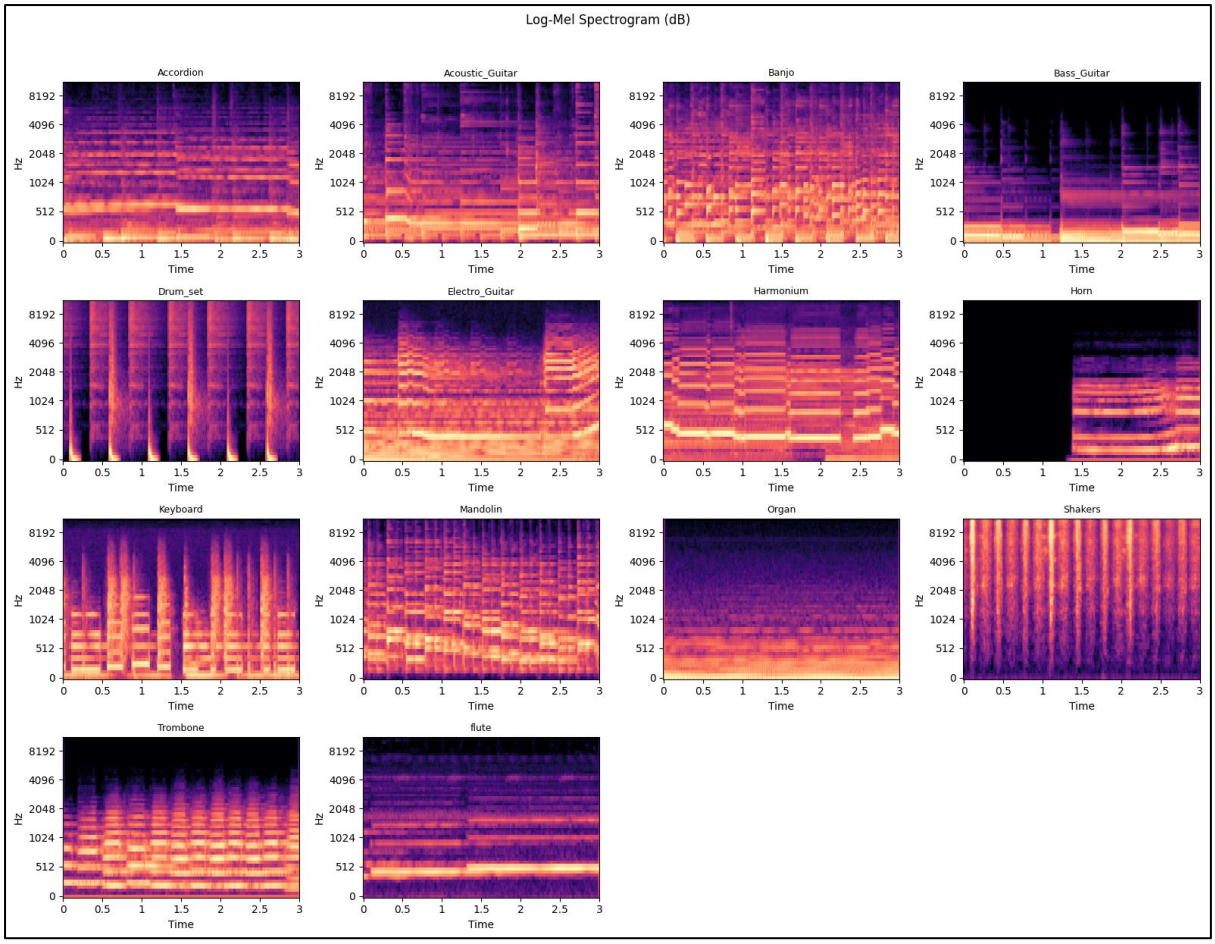

Figure 29: Sample Log-Mel Spectrogram of Each Class

The log-Mel spectrograms highlight how each instrument produces distinct timbral signatures once sound energy is mapped into a perceptual scale. Compared with the FFT magnitude and linear STFT spectrograms shown earlier, the Mel view condenses the frequency space, which makes patterns such as harmonic bands, percussive bursts, or strumming textures more visually separable across instruments. For example, the dense horizontal layers of harmonium contrast with the sharp vertical streaks of shakers, while the smoother harmonic lines of flute differ from the richer mid-frequency textures of mandolin.

```
fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows))
axes = axes.ravel()

for i, f in enumerate(features):
    ax = axes[i]
    # MFCC matrix over time (coefficients on y-axis, time on x-axis)
    librosa.display.specshow(
        f["mfcc"],
        x_axis="time",
        ax=ax)
    ax.set_title(f['class'], fontsize=9)

for j in range(i+1, len(axes)): axes[j].axis("off")

fig.suptitle(f"MFCCs (n={num_mfcc})", y=1.02)
plt.tight_layout(); plt.show()
```

Figure 30: Plot MFCCs

Next, the MFCC plots show how the spectral envelope of each instrument evolves over time, with MFCC index on the y-axis and time on the x-axis. Because MFCCs are a compact summary of the log-Mel spectrogram, they emphasize broad timbre shape while suppressing fine pitch detail and noise, making differences between classes easier to spot in a small matrix. Visualizing MFCCs alongside the Mel spectrograms confirms that both views capture complementary information, that Mel features show detailed time-frequency textures, and MFCCs distill them into a concise representation.

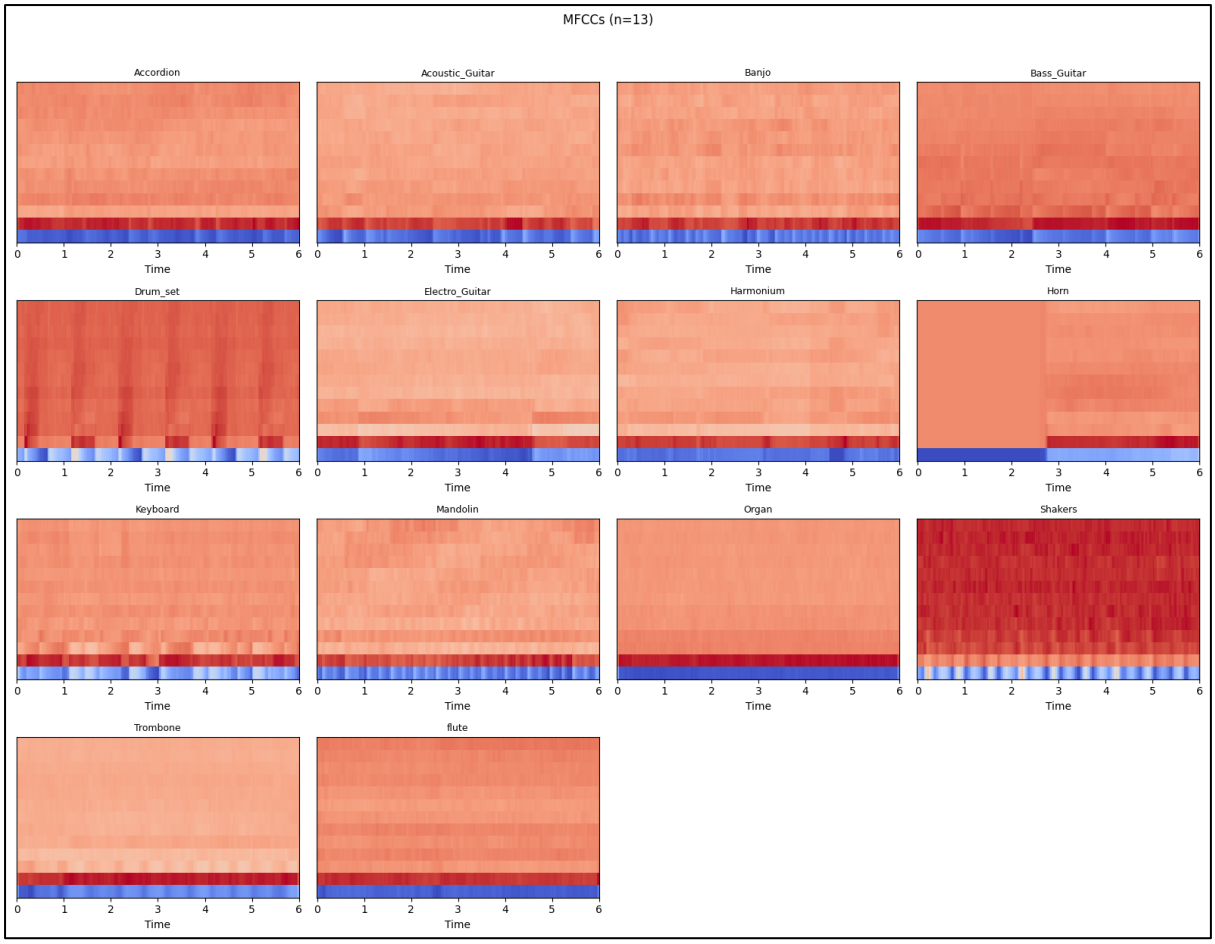

Figure 31: Sample MFCCs of Each Class

Figure 31 shows MFCCs with clear differences that can be noticed in how stable or varied the coefficient patterns are. Sustained instruments such as accordion, harmonium, and flute form relatively consistent bands, while percussive instruments like drum set and shakers create frequent changes across time, which reflect their sharp and irregular energy bursts. String instruments such as banjo and mandolin show moderate variation that matches strumming or plucking activity.

```
# Compute time-averaged log-Mel energies for each class sample
# Result: one vector of length num_mel_bins per class

# time-average log-mel per class to length become num_mel_bins
rows_list = []
for f in features:
    coeff = np.log(librosa.db_to_power(f["mel_db"]) + 1e-10).mean(axis=1)  # log energies
    rows_list.append({"class": f["class"], "coeff": coeff})

fbank_df = pd.DataFrame({
    "class": [r["class"] for r in rows_list],
    **{f"fb_{i}": [r["coeff"][i] for r in rows_list] for i in range(num_mel_bins)}
}).set_index("class")

# Heatmap: classes vs. Mel filter index
plt.figure(figsize=(10, max(4, 0.35*len(fbank_df))))
librosa.display.specshow(fbank_df.values, cmap="magma")
plt.title("Log Filter-Bank Coefficients (mean over time)")
plt.xlabel("Mel filter index")
plt.ylabel("Class")
plt.yticks(range(len(fbank_df.index)), fbank_df.index)
plt.colorbar(label="log energy")
plt.tight_layout()
plt.show()
```

Figure 32: Plot Log Filter-Bank Coefficients

Lastly, each class sample's log-Mel energy is summarized into a single timbral profile and visualized as a heatmap. For every clip, the log-Mel spectrogram is averaged over time, producing a 64-value vector (one value per Mel band) that reflects the clip's overall spectral envelope. These vectors are stacked into a table with classes as rows and Mel filter indices as columns, then plotted. The compact view is useful for spotting similarities that can confuse classification.

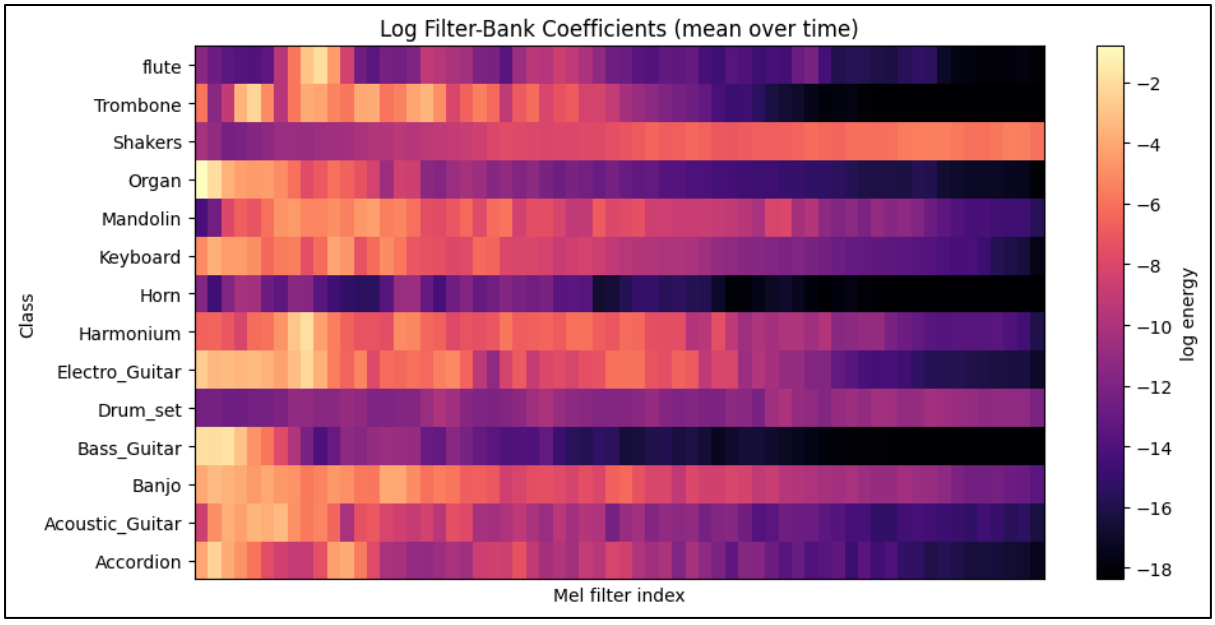

Figure 33: Sample Log Filter-Bank Coefficients of Each Class

Figure 33 heatmap shows the average log-Mel filter-bank energies. Each row represents an instrument, while the horizontal axis corresponds to Mel filter indices from low to high frequencies. Clear contrasts can be seen with bass-heavy instruments, such as bass guitar, which emphasize the lower Mel filters, while shakers highlight higher bands with more uniform energy across the spectrum. Sustained harmonic instruments like accordion and acoustic guitar display broader mid-frequency emphasis, and flute and trombone show stronger activity in specific low-to-mid ranges.

3.5 Stratified Split Dataset

```
from sklearn.model_selection import train_test_split

# build absolute paths once for convenience
df_all = df.copy()
df_all["path"] = df_all.apply(lambda r: str((DATA_DIR / r["class"] / r["file"]).resolve()), axis=1)

# Take 15% for final testing
trainval_df, test_df = train_test_split(
    df_all, test_size=0.15, stratify=df_all["class"], random_state=42
)

# From the remaining 85%, take 15/85 ≈ 17.647% as validation
# Resulting overall split ≈ 70% train / 15% val / 15% test
train_df, val_df = train_test_split(
    trainval_df, test_size=0.17647, stratify=trainval_df["class"], random_state=42
)
```

Figure 34: Stratified Train/Val/Test Split (70/15/15)

With the feature exploration complete and class imbalance considered, the dataset is split in a stratified way so that every subset keeps the same class proportions seen earlier. Absolute file paths are added for reliable loading, and a fixed *random_state=42* ensures the split is reproducible. First, 15% of the data is held out as a final test set that stays untouched until all modeling is finished to give an unbiased measure of generalization. From the remaining 85%, a further 17.647% is set aside for validation, which yields an overall 70/15/15 train/val/test split. The validation set is used for tasks like epoch, hyperparameter tuning, and early stopping, while stratification prevents rare classes from disappearing in smaller folds and keeps metrics like accuracy and macro-F1 comparable across splits.

```
# Build a table of per-class counts for each split
count_tbl = pd.DataFrame({
    "Train": train_df["class"].value_counts(),
    "Val":   val_df["class"].value_counts(),
    "Test":  test_df["class"].value_counts(),
}).fillna(0).astype(int).sort_index()

# add row total
count_tbl["Total"] = count_tbl.sum(axis=1)
count_tbl.loc["Total"] = count_tbl.sum(axis=0)

display(count_tbl)
```

class	Train	Val	Test	Total
Accordion	2507	537	537	3581
Acoustic_Guitar	2558	548	548	3654
Banjo	2098	450	450	2998
Bass_Guitar	2529	542	542	3613
Drum_set	2554	547	547	3648
Electro_Guitar	922	197	197	1316
Harmonium	920	197	197	1314
Horn	880	189	189	1258
Keyboard	1429	306	306	2041
Mandolin	1720	369	369	2458
Organ	1010	216	216	1442
Shakers	949	204	204	1357
Trombone	2075	445	445	2965
flute	2603	558	558	3719
Total	24754	5305	5305	35364

Figure 35: View Each Split Count

The output shows the number of audio clips from each instrument class after applying the stratified 70/15/15 split into training, validation, and test sets. Each class is represented proportionally across the three subsets, showing that stratification preserved the original balance observed during the EDA. The totals at the bottom show 24,754 clips for training, 5,305 for validation, and 5,305 for testing, with a combined dataset of 35,364 clips. This even distribution ensures that all models will have sufficient representation from each instrument class during training and fair evaluation on unseen data, which can reduce the risk of bias toward the more frequent categories.

```
# Sorted list of class names (stable order across runs)
class_names = sorted(train_df["class"].unique().tolist())

# Map: class name to integer index (0..num_classes-1)
class_to_idx = {c: i for i, c in enumerate(class_names)}

# Number of target classes
num_classes = len(class_names)
```

Figure 36: Class Index Mapping

After that, the class labels are organized into a consistent format for model training. A sorted list of class names is created to keep the order stable across different runs for reproducibility. Each class is then mapped to a unique integer index (0 to *num_classes* – 1). This forms a dictionary that converts text labels into numerical targets so that the models can work with them. Finally, the total number of classes is stored as *num_classes*, which will be used to define the model's output layer size.

```
# Mel frequency range (Hz)
fmin, fmax = 0.0, sample_rate/2

# Precompute Mel weight matrix once for TF ops (linear FFT bins to Mel bins)
mel_w = tf.signal.linear_to_mel_weight_matrix(
    num_mel_bins,
    1 + fft_size // 2,
    sample_rate,
    lower_edge_hertz=fmin,
    upper_edge_hertz=fmax,
    dtype=tf.float32,
)

def wav_to_logmel(path: tf.Tensor) -> tf.Tensor:
    # Read WAV file and decode to mono tensor
    audio_bin = tf.io.read_file(path)
    x, sr = tf.audio.decode_wav(audio_bin, desired_channels=1) # x: [T, 1], sr: scalar
    x = tf.squeeze(tf.cast(x, tf.float32), axis=-1) # [T]

    # STFT to power spectrogram
    S = tf.signal.stft(x, frame_length=fft_size, frame_step=hop_length, fft_length=fft_size)
    P = tf.math.real(S * tf.math.conj(S)) # [frames, 1+fft/2]

    # Linear to Mel, then take log for dynamic range compression
    M = tf.matmul(P, mel_w) # [frames, n_mels]
    M = tf.math.log(M + 1e-6) # log-Mel

    # Shape to [mels, frames, 1] for CNNs; ensure known mel dim for Keras
    M = tf.transpose(M, perm=[1, 0])[:, :, tf.newaxis]
    M = tf.ensure_shape(M, [num_mel_bins, None, 1]) # helpful for Keras shape inference
    return M
```

Figure 37: Convert WAV to Log-Mel Function

The next step is to build a TensorFlow input pipeline that converts each WAV file into a log-Mel spectrogram ready for the models. First, the Mel weight matrix is precomputed once by mapping linear FFT bins to 64 Mel bands over 0–11 kHz. For each path, the audio is read and

decoded to mono float. STFT value is computed to get a power spectrogram, which is projected onto Mel bands with the precomputed matrix and converted to log scale to compress dynamic range and emphasize timbre. Finally, the result is transposed to shape [mels, frames, 1]. This is a channel-last format for CNNs, with the Mel dimension fixed for clean Keras shape inference. This process turns raw file paths into consistent, model-ready features that match the previous EDA settings.

```
def make_ds_simple(dframe, batch=32, training=False, shuffle=1000):
    # Prepare file paths and integer labels
    paths  = dframe["path"].astype(str).tolist()
    labels = dframe["class"].map(class_to_idx).astype(np.int32).values

    ds = tf.data.Dataset.from_tensor_slices((paths, labels))
    if training:
        ds = ds.shuffle(shuffle, reshuffle_each_iteration=True)  # randomise batches during training

    # Map WAV to log-Mel tensor
    def _map(p, y):
        M = wav_to_logmel(p)                          # [mels, frames, 1]
        return tf.cast(M, tf.float32), tf.cast(y, tf.int32)

    # Parallel map, pad variable time axis, and prefetch for throughput
    ds = ds.map(_map, num_parallel_calls=tf.data.AUTOTUNE, deterministic=False)
    ds = ds.padded_batch(batch, padded_shapes=([num_mel_bins, None, 1], []))
    return ds.prefetch(tf.data.AUTOTUNE)

# Build datasets
train_ds = make_ds_simple(train_df, batch=32, training=True)
val_ds   = make_ds_simple(val_df,   batch=32, training=False)
test_ds  = make_ds_simple(test_df,  batch=32, training=False)
```

Figure 38: Build Log-Mel Feature Pipeline

With the previous function that converts a WAV file into a log-Mel spectrogram, next is to define *make_ds_simple* to turn the file paths and labels table into a TensorFlow dataset. Within this function, firstly, each file path is paired with its integer class label. The dataset for training will be shuffled so that batches are randomized at every epoch. Then, it maps each path using the earlier *wav_to_logmel* function to ensure every clip becomes a consistent [mels, frames, 1] tensor. These tensors and labels are batched together, and prefetching is enabled so data loading keeps up with model training. Finally, three datasets are built, including *train_ds*, *val_ds*, and *test_ds*, all according to the understanding of the preprocessing and EDA, and they are ready to be used in modelling. This pipeline ensures that raw audio is seamlessly transformed into model-ready features and can be reproduced.

3.6 Define Evaluation Function for Easy Use

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report

def plot_training_curves(history, metrics=("loss", "accuracy"), figsize=(10, 4), suptitle=None):
    # Handle Training History
    hist = history.history if hasattr(history, "history") else history

    # Plot metrics that have both train and val series
    show_metrics = [m for m in metrics if (m in hist and f"val_{m}" in hist)]
    if not show_metrics:
        # Fallback: try common keys
        keys = list(hist.keys())
        show_metrics = [k for k in ("loss", "accuracy") if k in keys and f"val_{k}" in keys]

    n = max(1, len(show_metrics))
    fig, axes = plt.subplots(1, n, figsize=figsize)
    axes = np.atleast_1d(axes)

    # Plot metrics
    for ax, m in zip(axes, show_metrics):
        ax.plot(hist[m], label="train")
        ax.plot(hist[f"val_{m}"], label="val")
        ax.set_title(m.capitalize())
        ax.set_xlabel("Epoch")
        ax.legend()

    if suptitle:
        fig.suptitle(suptitle)
    plt.tight_layout()
    plt.show()
```

Figure 39: Plot Training Curves Function

Before starting modelling, some evaluation helpers are defined to make life easier. The function *plot_training_curves* takes a Keras History or a plain dict that is stored in CSV in this scenario, and automatically plots the metrics that have both training and validation series, including training and validation loss, training and validation accuracy. It will first extract the history, check the available metrics, and then create side-by-side line charts over epochs, labeling training and validation so trends are easy to compare. This visual check helps see if there is overfitting, underfitting, or healthy learning. The function is designed to be flexible because it supports custom titles and figure size for all different models.

```python
def batch_predict(ds, model):
    # Iterate over a tf.data dataset and collect true/pred labels
    y_true, y_pred = [], []
    for X, y in ds:
        # If labels are one-hot, convert to integer indices
        y_np = y.numpy()
        if y_np.ndim > 1 and y_np.shape[-1] > 1:
            y_labels = np.argmax(y_np, axis=1)
        else:
            y_labels = y_np.reshape(-1)

        # Model forward pass
        p = model.predict(X, verbose=0)
        y_true.extend(y_labels.tolist())
        y_pred.extend(np.argmax(p, axis=1).tolist())
    return np.asarray(y_true, dtype=int), np.asarray(y_pred, dtype=int)

def plot_confusion(cm, class_names=None, normalize=False, cmap="Blues",
                   title="Confusion Matrix", xtick_rotation=45, ytick_rotation=0):
    # Row-normalize the confusion matrix to show per-class accuracy
    if normalize:
        with np.errstate(all="ignore"):
            cm = cm.astype("float") / cm.sum(axis=1, keepdims=True)
        cm = np.nan_to_num(cm)

    # Display plot
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
    disp.plot(cmap=cmap, values_format=".2f" if normalize else "d", colorbar=False)

    ax = plt.gca()
    ax.set_title(title)

    # Rotate ticks for readability
    for label in ax.get_xticklabels():
        label.set_rotation(xtick_rotation)
        label.set_horizontalalignment("right")
    for label in ax.get_yticklabels():
        label.set_rotation(ytick_rotation)

    plt.tight_layout()
    plt.show()
```

Figure 40: Test Model and Plot Confusion Matrix Function

To streamline the evaluation for the confusion matrix and classification report after training, two small helpers are defined first. *batch_predict* walks through a *tf.data* dataset, runs *model.predict* on each batch, and collects the true labels and the model's predicted class indices into NumPy arrays. It also handles both integer labels and one-hot labels, so outputs are consistent for later metrics. *plot_confusion* then takes a confusion matrix and renders it with readable axis labels and optional row-normalization to make it easy to read per-class accuracy and spot which instruments are most often confused with which others. Tick rotation and a clean color map improve readability.

```python
def evaluate_model(model, test_ds, class_names=None, model_name="Model", normalize_cm=False, num_classes=None, return_results=True):
    # Run predictions and gather ground-truth labels
    y_true, y_pred = batch_predict(test_ds, model)

    # Ensure confusion_matrix uses the full label set
    if num_classes is None:
        if class_names is not None:
            num_classes = len(class_names)
        else:
            num_classes = int(max(y_true.max(), y_pred.max())) + 1
    labels = np.arange(num_classes)

    # Overall accuracy
    acc = (y_true == y_pred).mean()
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    print(f"{model_name} Test accuracy: {acc:.4f}")

    # Confusion matrix plot
    title = f"{model_name} - Confusion Matrix"
    plot_confusion(cm, class_names=class_names, normalize=normalize_cm, title=title)

    # Classification report
    names_for_report = class_names if class_names is not None else [str(i) for i in labels]
    print("\nClassification report:\n")
    print(classification_report(y_true, y_pred, target_names=names_for_report, digits=3))

    # Can return raw results for further analysis
    if return_results:
        return {
            "y_true": y_true,
            "y_pred": y_pred,
            "accuracy": acc,
            "confusion_matrix": cm
        }
```

Figure 41: Evaluate Model Function

Then, a function is defined to make both helpers work together in a single evaluation function call. *evaluate_model* first calls *batch_predict* to generate the true and predicted labels from the provided test dataset. It then ensures the full label set is used, computes the overall test accuracy, and builds a confusion matrix to visualize class-wise performance with *plot_confusion*. Finally, it prints a detailed classification report that includes precision, recall, and F1-score for each instrument, which becomes the evaluation metrics for the model in this project. This single function can provide a direct and easy call to the model performance on test data.

4 Model Building and Evaluation

In this section, the focus is on building and evaluating two deep learning models, which are a CNN and an RNN with BiGRU. Each model will first be trained in a baseline form to test how well it responds to the dataset and to provide a starting point for comparison. From there, the models will be enhanced step by step, with hyperparameter tuning applied to find the best configuration to deliver strong and reliable performance for instrument sound classification.

4.1 CNN

4.1.1 Baseline CNN

```
from tensorflow.keras import Sequential, layers, optimizers

# Log-Mel input: [mels, frames, channels]
input_shape = (num_mel_bins, None, 1)

base_model_cnn = Sequential(name="base_model_cnn")
base_model_cnn.add(layers.Input(shape=(num_mel_bins, None, 1)))

# Block 1
base_model_cnn.add(layers.Conv2D(8, (3,3), padding="same", activation="relu"))
base_model_cnn.add(layers.MaxPooling2D(pool_size=(4,4)))

# Block 2
base_model_cnn.add(layers.Conv2D(8, (3,3), padding="same", activation="relu"))
base_model_cnn.add(layers.MaxPooling2D(pool_size=(4,4)))

# Classification head
base_model_cnn.add(layers.GlobalAveragePooling2D()) # reduce HxW to 1x1
base_model_cnn.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

# Compile with a simple optimizer and standard loss for int labels
base_model_cnn.compile(
    optimizer=optimizers.SGD(learning_rate=0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Figure 42: Build Baseline CNN Model

The modelling process starts with a baseline CNN. The network needs log-Mel spectrogram inputs in the shape *(mels, frames, channels) = (64, None, 1)*. So, the Mel axis acts like image height, and the time axis (frames) acts like width. Using *None* on the time dimension keeps the model compatible with variable frame counts, and the earlier preprocessing already shows consistent channel layout and Mel dimension for clean Keras shape inference.

In the baseline CNN, two convolutional blocks with pooling are the core feature extractor. Each *Conv2D* uses 8 filters with a 3×3 kernel, *padding="same"*, and ReLU activation. The 3×3 kernel captures local time-frequency patterns, while “same” padding preserves spatial size before pooling, so features are not lost at the edges. ReLU introduces non-linearity and trains stably in shallow baselines. The small filter count (8 only) for baselune reduces computation and overfitting risk, and makes the baseline sensitive to data quality and feature choices before adding capacity in later iterations, to test how CNN will perform. Each convolution is followed by *MaxPooling2D(pool_size=(4,4)),* which down-samples both the Mel and time axes. Pooling by 4 along frequency and time builds invariance to small pitch and timing shifts. With 64 Mel bins and roughly about 255 time frames per 3-second clip, two 4×4 pools can reduce the feature map by about 16 times on each axis (Mel is 64 to 4 and Time is about 255 to 16). This gives a compact map that still preserves coarse timbral and temporal structure.

A *GlobalAveragePooling2D* layer then collapses the remaining spatial map to a single feature vector by averaging each channel over all time-frequency positions. This choice keeps the parameter count low to mitigate overfitting and makes the classifier less sensitive to where in the clip a pattern occurs. It also naturally supports variable-length inputs, since the average is taken over whatever number of frames are present. The final *Dense(num_classes, activation="softmax")* layer gives a probability distribution over the instrument classes, directly matching the integer labels initiated earlier.

For optimization, the model is compiled with SGD at a learning rate of 0.01, *sparse_categorical_crossentropy* loss, and accuracy as the primary metric. Sparse cross-entropy fits the label format, and plain SGD provides a simple, interpretable baseline for learning dynamics.

Overall, this configuration is intentionally minimal. Small kernels to learn local spectro-temporal cues, moderate pooling to build invariance and reduce compute, global averaging to regularize and permit variable length, and a lightweight classifier head to expose how well the dataset and features alone support discrimination before moving into later enhancement.

```
base_model_cnn.summary()
```

Model: "base_model_cnn"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, None, 8)	80
max_pooling2d (MaxPooling2D)	(None, 16, None, 8)	0
conv2d_1 (Conv2D)	(None, 16, None, 8)	584
max_pooling2d_1 (MaxPooling2D)	(None, 4, None, 8)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 8)	0
Classifier (Dense)	(None, 14)	126

Total params: 790 (3.09 KB)

Trainable params: 790 (3.09 KB)

Non-trainable params: 0 (0.00 B)

Figure 43: Baseline CNN Model Summary

The model summary confirms the structure and size of the baseline CNN. The output shapes show how the input log-Mel spectrogram is gradually reduced through convolution and pooling. After the first convolution, the shape becomes *(None, 64, None, 8)*, meaning 64 Mel bins, a variable number of time frames, and 8 feature maps. Pooling reduces the frequency dimension from 64 to 16, and then to 4 after the second pooling, while the time dimension also shrinks proportionally. By the time the data reaches the global average pooling layer, each input clip is represented as a compact vector of length 8. The final dense classifier maps this vector into 14 outputs, one per instrument class.

The parameter counts are also very small, with only 790 total trainable parameters and no non-trainable weights. The purpose is to make the model fast to train and useful as a simple baseline to check how well the dataset alone supports classification. Since all parameters are trainable, the model has full flexibility within its limited size.

```
from tensorflow.keras.utils import plot_model
from IPython.display import Image, display

# Export a PNG of the model graph
plot_model(base_model_cnn, to_file="../output/model_architecture/base_model_cnn.png",
           show_shapes=True, show_layer_names=True,
           expand_nested=True, dpi=150)
display(Image(filename="../output/model_architecture/base_model_cnn.png", width=500))
```

Figure 44: Display Baseline CNN Model Architecture

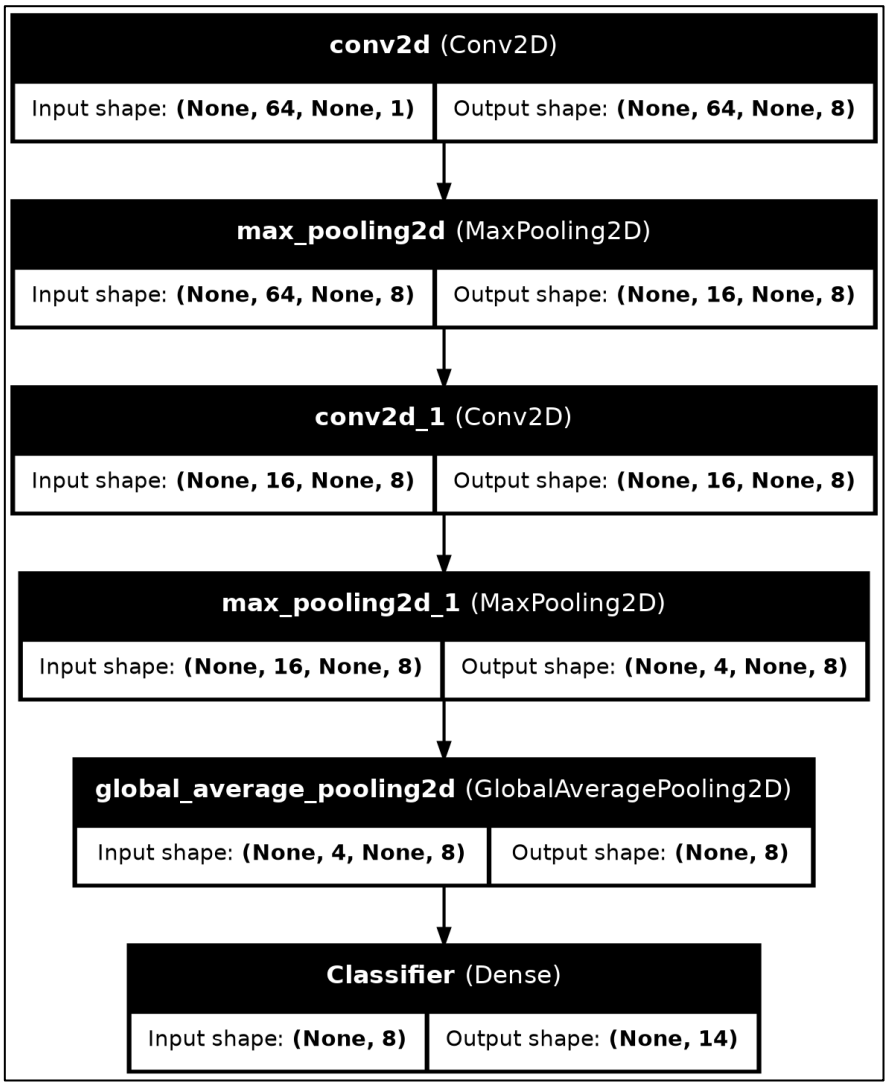

Figure 45: Baseline CNN Model Architecture

After the model is built, it is exported and displayed as a diagram that shows each layer and the corresponding input and output shapes to visually confirm the architecture. It is only for easy visualization and reporting because the full details of the structure are already explained.

```python
from tensorflow.keras import callbacks

# Log epoch-by-epoch metrics to save the footprint
log = callbacks.CSVLogger("../output/training_log/base_model_cnn.csv", append=False)

# Train baseline model on log-Mel inputs
history = base_model_cnn.fit(
    train_ds,
    validation_data=val_ds,
    epochs=30,
    callbacks=log,
    verbose=1
)
```

Figure 46: Train Baseline CNN

After the model is built, training begins using the prepared *train_ds* and *val_ds* datasets for 30 epochs as the baseline. *CSVLogger* callback is set to record epoch-by-epoch to a CSV file, which can be found back later. The call to *model.fit(...)* iterates over mini-batches from *train_ds*, updates the weights, and then evaluates on *val_ds* at the end of each epoch to monitor generalization and detect signs of overfitting. The *verbose=1* setting prints a progress bar and metrics each epoch for a quick view of learning dynamics so that CSV log here serves as the permanent record for analysis and reporting.

```
Epoch 1/30
2025-09-25 14:13:26.277222: I external/local_xla/xla/service/service.cc:163] XLA service 0x7a3e5c004580 initialized for
2025-09-25 14:13:26.277265: I external/local_xla/xla/service/service.cc:171]   StreamExecutor device (0): NVIDIA GeForce
2025-09-25 14:13:26.351728: I tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:269] disabling MLIR crash repr
2025-09-25 14:13:26.540562: I external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:473] Loaded cuDNN version 91300
 19/774 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.0486 - loss: 3.0003
I0000 00:00:1758780811.404415   49015 device_compiler.h:196] Compiled cluster using XLA!  This line is logged at most or
774/774 ━━━━━━━━━━━━━━━━━━━━ 20s 18ms/step - accuracy: 0.4072 - loss: 1.8161 - val_accuracy: 0.5874 - val_loss: 1.3150
Epoch 2/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.7098 - loss: 0.9458 - val_accuracy: 0.7877 - val_loss: 0.6965
Epoch 3/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.7795 - loss: 0.7054 - val_accuracy: 0.8136 - val_loss: 0.6085
Epoch 4/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.8013 - loss: 0.6153 - val_accuracy: 0.7749 - val_loss: 0.6174
Epoch 5/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.8233 - loss: 0.5425 - val_accuracy: 0.7813 - val_loss: 0.6489
```

Figure 47: Baseline CNN First 5 Epoch

```
Epoch 26/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9016 - loss: 0.3020 - val_accuracy: 0.8854 - val_loss: 0.3300
Epoch 27/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9033 - loss: 0.2981 - val_accuracy: 0.8961 - val_loss: 0.3135
Epoch 28/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 11s 14ms/step - accuracy: 0.9009 - loss: 0.2998 - val_accuracy: 0.9037 - val_loss: 0.2805
Epoch 29/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.9049 - loss: 0.2904 - val_accuracy: 0.9014 - val_loss: 0.2870
Epoch 30/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 10s 13ms/step - accuracy: 0.9039 - loss: 0.2912 - val_accuracy: 0.8893 - val_loss: 0.3291
```

Figure 48: Baseline CNN Last 5 Epoch

The training logs show that in the first few epochs, the model quickly improved from very low accuracy around 0.4 to 0.6 to above 0.8, with validation loss dropping. By the final 5 epochs,

training accuracy stabilized around 0.90 and validation accuracy around 0.88 to 0.90, showing that the baseline CNN learned useful patterns from the data, but still needs further evaluation to know the result.

```
# Plot training curves
plot_training_curves(history, metrics=("loss", "accuracy"), suptitle="Baseline CNN")
```

Figure 49: Plot Baseline CNN Training Curves

The previously defined evaluation function *plot_training_curves* is now used to visualize the baseline CNN's loss and accuracy evolved during training and validation.

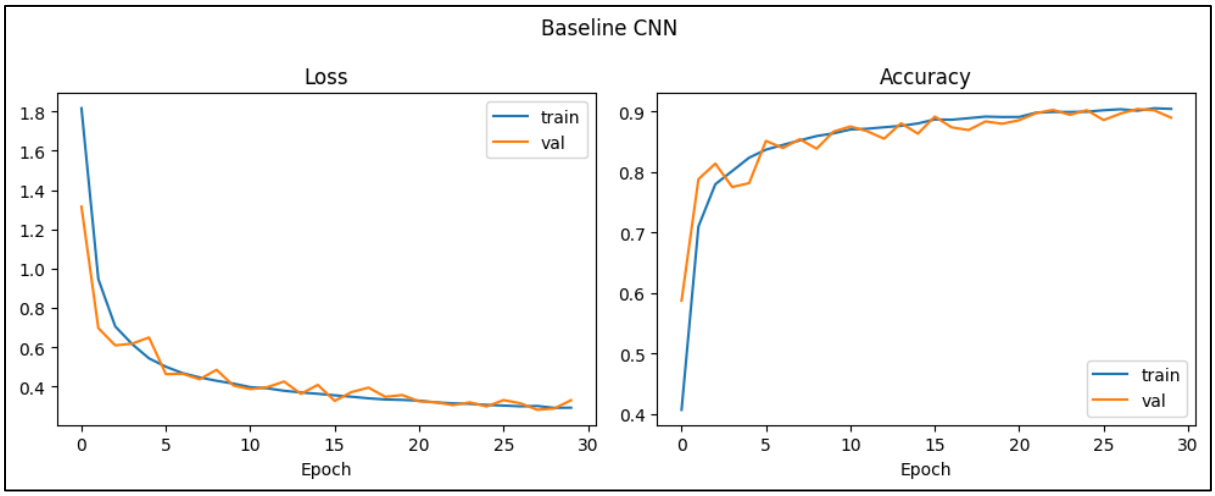

Figure 50: Baseline CNN Training Curves

Figure 50 shows the training curves of the baseline CNN across 30 epochs. The left side shows the loss decreases for both training and validation, meaning that the model is learning effectively without divergence. On the right, accuracy quickly rises from around 40% in the first epoch to over 90% by the end, with training and validation lines tracking closely together. The overlap between the two curves means that the model generalizes well and does not show strong signs of overfitting.

```
# Evaluate model, and plot confusion matrix and classification report
results = evaluate_model(
    model=base_model_cnn,
    test_ds=test_ds,
    class_names=class_names,
    model_name="Baseline CNN",
    normalize_cm=False
)
```

Figure 51: Plot Baseline CNN Confusion Matrix and Classification Report

The previously defined *evaluate_model* function is now also called to generate the confusion matrix and classification report for the baseline CNN on the test set.

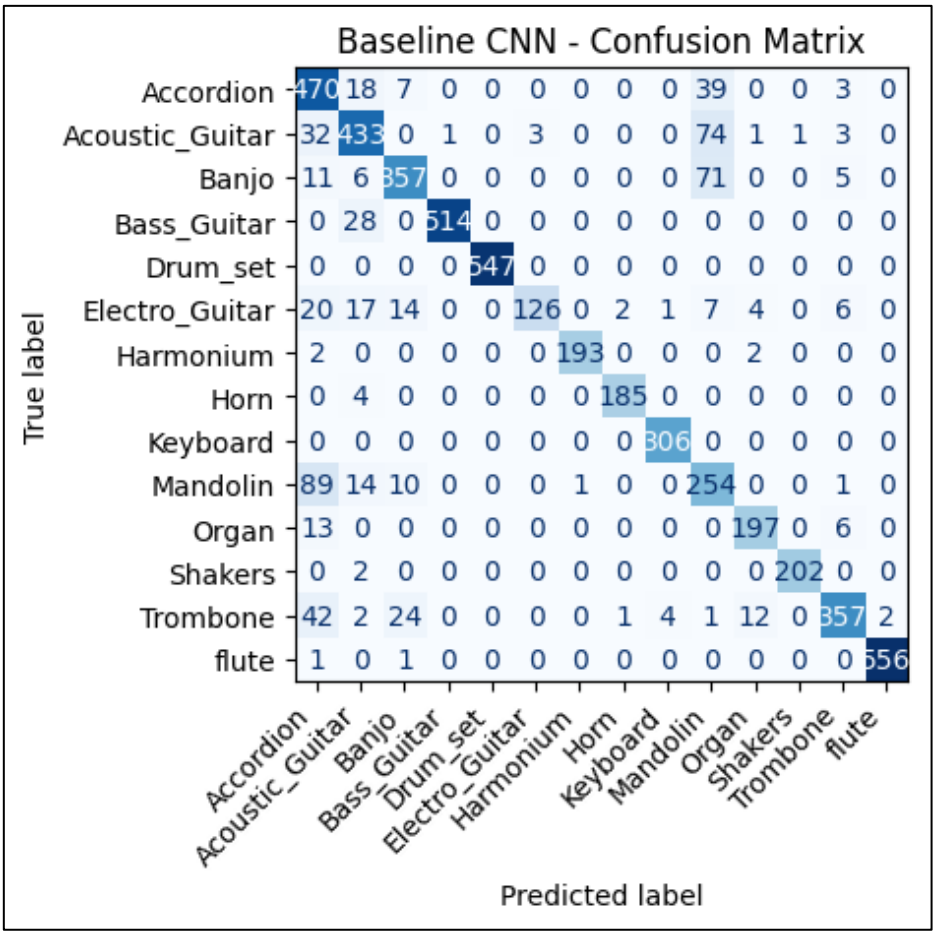

Figure 52: Baseline CNN Confusion Matrix

This confusion matrix in Figure 52 shows the baseline CNN classification performance on each instrument in the test set. The diagonal values represent correct predictions, and most of them are high. For example, flute, drum set, and shakers are almost perfectly classified, while accordion, acoustic guitar, and mandolin still show some confusion, often being misclassified as one another. Similarly, trombone and accordion sometimes overlap, which is reasonable because their timbre have similar harmonic structures.

```
Classification report:

                precision    recall  f1-score   support

     Accordion       0.691     0.875     0.772       537
Acoustic_Guitar      0.826     0.790     0.808       548
         Banjo       0.864     0.793     0.827       450
   Bass_Guitar       0.998     0.948     0.973       542
      Drum_set       1.000     1.000     1.000       547
 Electro_Guitar      0.977     0.640     0.773       197
      Harmonium      0.995     0.980     0.987       197
          Horn       0.984     0.979     0.981       189
      Keyboard       0.984     1.000     0.992       306
      Mandolin       0.570     0.688     0.623       369
         Organ       0.912     0.912     0.912       216
       Shakers       0.995     0.990     0.993       204
      Trombone       0.937     0.802     0.864       445
         flute       0.996     0.996     0.996       558

      accuracy                           0.885      5305
     macro avg       0.909     0.885     0.893      5305
  weighted avg       0.897     0.885     0.888      5305
```

Figure 53: Baseline CNN Classification Report

The classification report in Figure 53 gives a more detailed view of the baseline CNN's performance. The mode overall accuracy is 88.5%, to be honest, it is strong already for a simple model. Instruments such as drum set, flute, harmonium, keyboard, and shakers show near-perfect results with precision, recall, and F1-scores above 0.98. However, some instruments like accordion, acoustic guitar, banjo, mandolin, and trombone are more challenging, with lower recall or precision. The macro average F1-score of 0.893 shows the balanced performance across classes, while the weighted average of 0.888 accounts for class size. This indicates the baseline CNN can capture discriminative features but still struggles with instruments that share similar frequency patterns or timbres, which means it still has space for improvement.

```
# Save model
base_model_cnn.save("../output/model/base_model_cnn.keras")
```

Figure 54: Save Model

The trained model is saved as a *Keras*. The following models will also be saved for record purposes.

4.1.2 Enhanced CNN

```
from tensorflow.keras import Sequential, layers, optimizers

# Log-Mel input: [mels, frames, channels]
input_shape = (num_mel_bins, None, 1)

enhanced_model_cnn = Sequential(name="enhanced_model_cnn")
enhanced_model_cnn.add(layers.Input(shape=input_shape))

# Block 1
enhanced_model_cnn.add(layers.Conv2D(8, (3,3), padding="same")) # conv without inline activation
enhanced_model_cnn.add(layers.BatchNormalization())              # stabilize/accelerate training
enhanced_model_cnn.add(layers.ReLU())                            # non-linearity after BN
enhanced_model_cnn.add(layers.MaxPooling2D(pool_size=(2,2)))     # downsample (time x mel)

# Block 2
enhanced_model_cnn.add(layers.Conv2D(16, (3,3), padding="same"))
enhanced_model_cnn.add(layers.BatchNormalization())
enhanced_model_cnn.add(layers.ReLU())
enhanced_model_cnn.add(layers.MaxPooling2D(pool_size=(2,2)))

# Block 3
enhanced_model_cnn.add(layers.Conv2D(32, (3,3), padding="same"))
enhanced_model_cnn.add(layers.BatchNormalization())
enhanced_model_cnn.add(layers.ReLU())
enhanced_model_cnn.add(layers.Dropout(0.3))                      # regularize features

# Classification head
enhanced_model_cnn.add(layers.GlobalAveragePooling2D())          # squeeze spatial dims
enhanced_model_cnn.add(layers.Dense(32, activation="relu"))      # small dense bottleneck
enhanced_model_cnn.add(layers.Dropout(0.3))                      # regularize head
enhanced_model_cnn.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

# Compile with Adam and standard sparse CE for integer labels
enhanced_model_cnn.compile(
    optimizer=optimizers.Adam(learning_rate=3e-4),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Figure 55: Build Enhanced CNN Model

The modelling continues with an enhanced CNN that builds directly on the baseline design but adds capacity and regularization to improve accuracy and stability. The input shape for CNN is all the same log-Mel format *(64, None, 1)*. The main idea is to keep the baseline like local time-frequency filters and global average pooling, and base on this to improve the feature quality, preserve more detail, and control overfitting.

First, one more hidden layer block is added to this model. Each convolutional block now uses the sequence from *Conv2D* to *BatchNormalization* and to *ReLU* instead of a *Conv2D* with an inline activation. Batch normalization is used to stabilize the distribution of activations to make training less sensitive to initialization and learning rate, and speed up convergence. ReLU comes after normalization because normalization operates on the linear outputs of the

convolution before non-linearity is applied. The number of filters increases by block from 8 to 16 and then to 32. This gradual growth can let the network learn more channels of features as representations become more abstract.

Then, max pooling is designed to be gentler than before. The baseline used 4×4 pooling twice, but the enhanced model uses 2×2 pooling in the first two blocks and no pooling in the third block. This keeps more resolution on both axes. Then, a *Dropout(0.3)* is added after the third block and again in the classifier head. Dropout will randomly disable a fraction of features during training to reduce co-adaptation and help the model generalize to unseen data. This is important because the network now has more filters than the baseline, it is good, but meaning can also bring a higher risk of overfitting.

The classification head combines *GlobalAveragePooling2D* with a small *Dense(32, ReLU)* bottleneck before the final softmax. The 32-unit dense layer mixes the pooled channels to form a compact task-specific representation. The dropout after this layer can help to regularize the head for reducing overfitting.

Finally, the optimizer switches from SGD to Adam with a learning rate of 0.0003. Adam adapts the step size per parameter. Hopefully, it should provide smoother training on audio spectrograms, especially when batch normalization and deeper stacks are added to this model. The loss remains sparse categorical cross-entropy. Overall, the changes from the baseline CNN to the enhanced CNN model already increase the expressive power and add safeguards against overfitting.

```
enhanced_model_cnn.summary()
```

Model: "enhanced_model_cnn"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 64, None, 8)	80
batch_normalization (BatchNormalization)	(None, 64, None, 8)	32
re_lu (ReLU)	(None, 64, None, 8)	0
max_pooling2d_2 (MaxPooling2D)	(None, 32, None, 8)	0
conv2d_3 (Conv2D)	(None, 32, None, 16)	1,168
batch_normalization_1 (BatchNormalization)	(None, 32, None, 16)	64
re_lu_1 (ReLU)	(None, 32, None, 16)	0
max_pooling2d_3 (MaxPooling2D)	(None, 16, None, 16)	0
conv2d_4 (Conv2D)	(None, 16, None, 32)	4,640
batch_normalization_2 (BatchNormalization)	(None, 16, None, 32)	128
re_lu_2 (ReLU)	(None, 16, None, 32)	0
dropout (Dropout)	(None, 16, None, 32)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 32)	0
dense (Dense)	(None, 32)	1,056
dropout_1 (Dropout)	(None, 32)	0
Classifier (Dense)	(None, 14)	462

Total params: 7,630 (29.80 KB)

Trainable params: 7,518 (29.37 KB)

Non-trainable params: 112 (448.00 B)

Figure 56: Enhanced CNN Model Summary

Compared to the baseline CNN, which had only 790 trainable parameters, the enhanced model has about 7,518 trainable parameters, almost ten times more capacity. This increase is because of the deeper convolutional layers from 8 to 16 and then to 32 filters, and the added dense bottleneck layer. The 112 non-trainable parameters come from batch normalization layers, which store running statistics used during inference.

While the output shape spectrogram is gradually reduced from *(64, None, 8)* after the first convolution to *(16, None, 32)* before global average pooling, meaning frequency resolution is compressed, but the number of learned feature maps grows. The global average pooling at the end will reduce everything to a 32-length vector. This model is larger and deeper than the baseline.

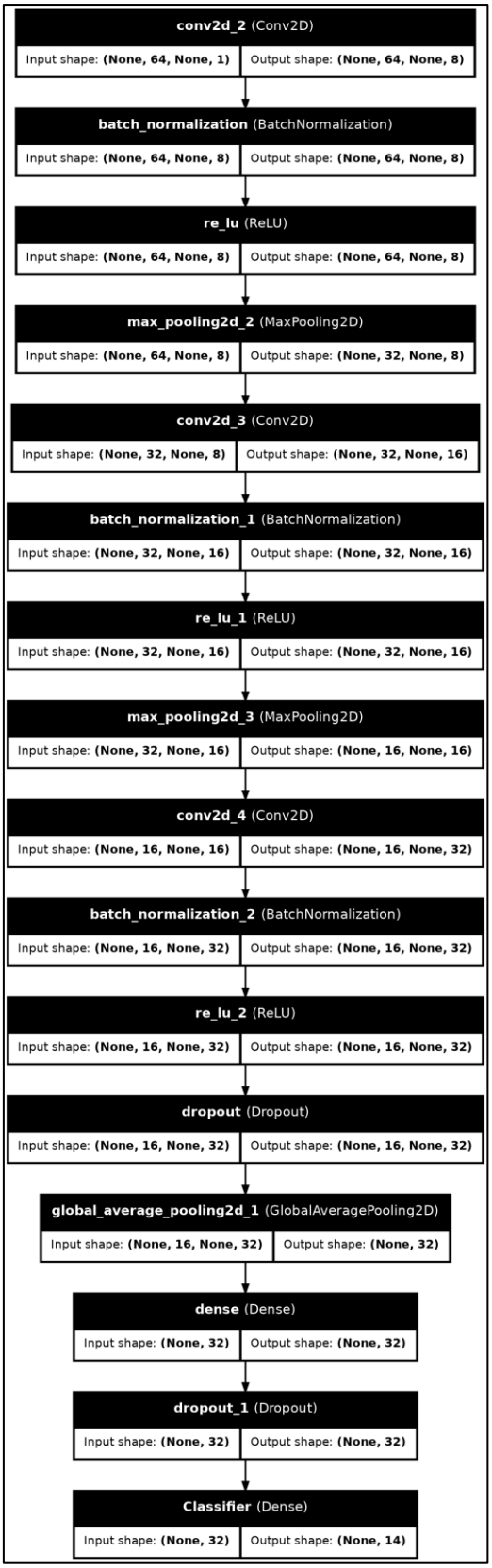

Figure 57: Enhanced CNN Model Architecture

Figure 57 is the diagram of the enhanced CNN architecture, purposely shown for flow visualization only.

```python
from tensorflow.keras import callbacks

# Stop when val accuracy stops improving; restore the best weights
es = callbacks.EarlyStopping(
    monitor="val_accuracy",
    mode="max",
    patience=6,
    restore_best_weights=True,
    verbose=1
)

# Reduce LR on validation loss plateau to fine-tune later epochs
rlr = callbacks.ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.5,
    patience=3,
    verbose=1
)

# Log epoch metrics to CSV for later plotting
log = callbacks.CSVLogger("../output/training_log/enhanced_model_cnn.csv", append=False)

# Train the enhanced CNN
history = enhanced_model_cnn.fit(
    train_ds,
    validation_data=val_ds,
    epochs=30,
    callbacks=[es, rlr, log],
    verbose=1
)
```

Figure 58: Train Enhanced CNN

The training process of the enhanced CNN is also enhanced. *EarlyStopping* is used to watch validation accuracy and stops training if it does not improve for 6 consecutive epochs. This can prevent overfitting when training accuracy is improving but validation is not, and saves time. *restore_best_weights=True* rolls the model back to the weights from the best validation accuracy, so final evaluation uses the strongest checkpoint rather than the last one seen.

Because models often learn quickly at first and then slow down, *ReduceLROnPlateau* monitors validation loss and halves the learning rate *(factor=0.5)* when improvement stalls for 3 epochs. Lowering the learning rate late in training can help the optimizer take smaller and more precise steps.

Same as baseline CNN, a CSVLogger records epoch-by-epoch metrics to a file, ensuring results can be found even if the editor is closed or the notebook session resets. The model then trains for up to 30 epochs with these callbacks active. This setup balances speed, stability, and generalization, theoretically, it can lead to better performance. Let's see the result.

```
Epoch 1/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 19s 18ms/step - accuracy: 0.4884 - loss: 1.6944 - val_accuracy: 0.7080 - val_loss: 1.1071 - learning_rate: 3.0000e-04
Epoch 2/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 12s 15ms/step - accuracy: 0.6722 - loss: 1.0420 - val_accuracy: 0.7953 - val_loss: 0.7135 - learning_rate: 3.0000e-04
Epoch 3/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.7409 - loss: 0.7994 - val_accuracy: 0.8639 - val_loss: 0.5441 - learning_rate: 3.0000e-04
Epoch 4/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 9s 12ms/step - accuracy: 0.7947 - loss: 0.6605 - val_accuracy: 0.8613 - val_loss: 0.4563 - learning_rate: 3.0000e-04
Epoch 5/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 12s 15ms/step - accuracy: 0.8222 - loss: 0.5633 - val_accuracy: 0.9167 - val_loss: 0.3415 - learning_rate: 3.0000e-04
```

Figure 59: Enhanced CNN First 5 Epoch

```
Epoch 15/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 17s 22ms/step - accuracy: 0.9104 - loss: 0.2792 - val_accuracy: 0.9551 - val_loss: 0.1568 - learning_rate: 3.0000e-04
Epoch 16/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 20s 25ms/step - accuracy: 0.9177 - loss: 0.2674 - val_accuracy: 0.9465 - val_loss: 0.1684 - learning_rate: 3.0000e-04
Epoch 17/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 17s 21ms/step - accuracy: 0.9194 - loss: 0.2545 - val_accuracy: 0.9514 - val_loss: 0.1518 - learning_rate: 3.0000e-04
Epoch 18/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 20s 26ms/step - accuracy: 0.9226 - loss: 0.2479 - val_accuracy: 0.9489 - val_loss: 0.1636 - learning_rate: 3.0000e-04
Epoch 19/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 13s 17ms/step - accuracy: 0.9262 - loss: 0.2399 - val_accuracy: 0.9374 - val_loss: 0.1688 - learning_rate: 3.0000e-04
Epoch 20/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 12s 16ms/step - accuracy: 0.9270 - loss: 0.2328 - val_accuracy: 0.9512 - val_loss: 0.1473 - learning_rate: 3.0000e-04
Epoch 21/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 15s 19ms/step - accuracy: 0.9273 - loss: 0.2293 - val_accuracy: 0.9538 - val_loss: 0.1519 - learning_rate: 3.0000e-04
Epoch 21: early stopping
Restoring model weights from the end of the best epoch: 15.
```

Figure 60: Enhanced CNN Last 5 Epoch

In the first five epochs, the enhanced CNN improved rapidly, with training accuracy rising from about 49% to 82% and validation accuracy rising from 71% to over 91%, while both losses also dropped. By the last five epochs, training accuracy stabilized around 92 to 93% and validation accuracy peaked at 95%. Training stopped early at epoch 21 due to the early stopping callback because no further improvement, which restored the weights from the best epoch, which is epoch 15.

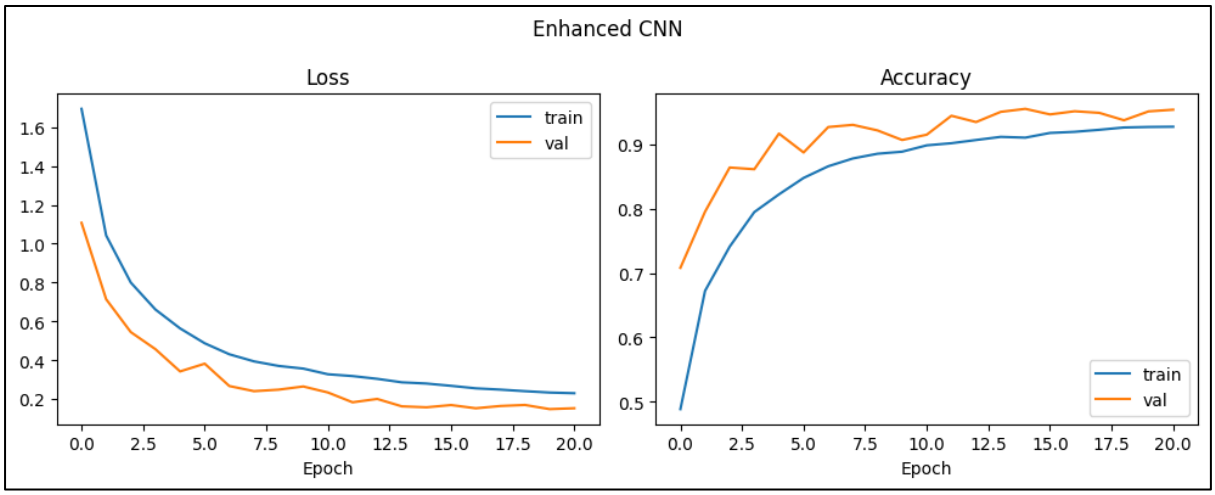

Figure 61: Enhanced CNN Training Curves

On the loss plot, both training and validation loss decrease steadily. Validation loss is slightly lower. On the accuracy plot, training accuracy gradually rises to around 92 and 93%, while validation accuracy reaches as high as 95% (as shown in the training log), which is okay in generalization without signs of overfitting.

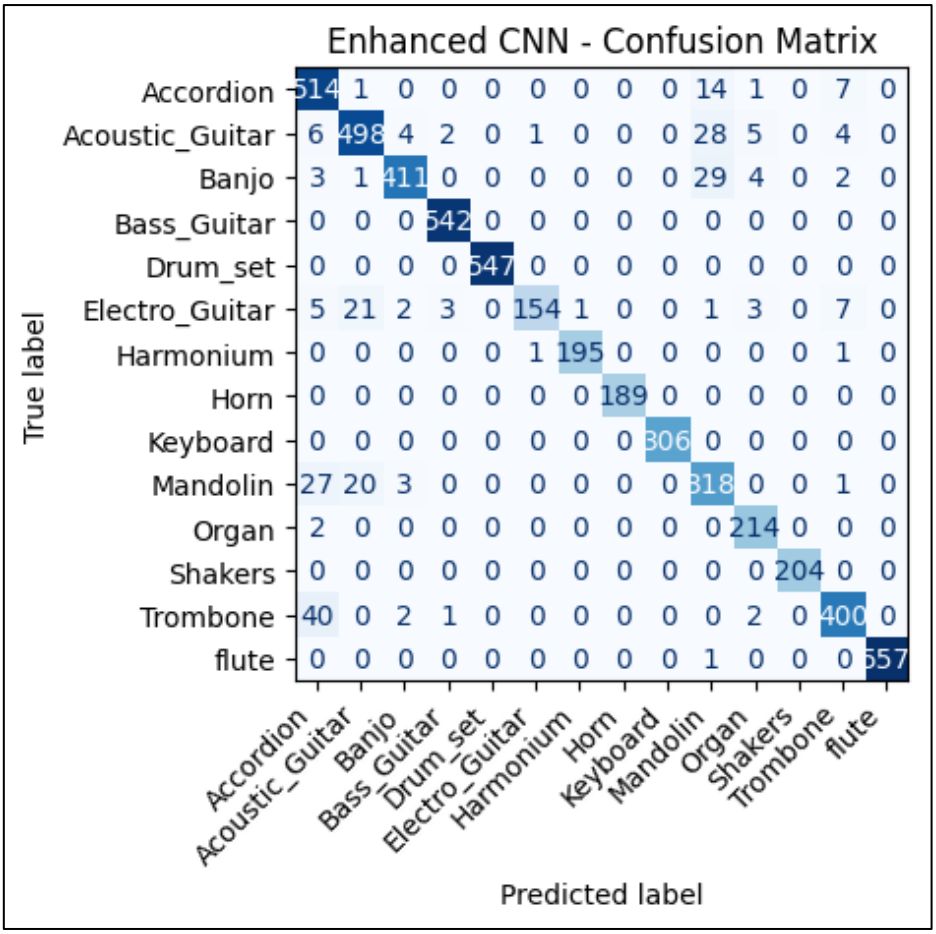

Figure 62: Enhanced CNN Confusion Matrix

Figure 62 shows that the model performs better across all instrument classes, with most predictions concentrated on the diagonal. Instruments such as the drum set, harmonium, horn, keyboard, shakers, and flute are almost correctly classified, with nearly no misclassifications. Others, such as the accordion, acoustic guitar, banjo, and trombone, still exhibit some confusion with similar instruments, but the errors are fewer compared to the baseline model. The mandolin and trombone also have clearer improvements in recognition.

Classification report:

	precision	recall	f1-score	support
Accordion	0.861	0.957	0.907	537
Acoustic_Guitar	0.921	0.909	0.915	548
Banjo	0.974	0.913	0.943	450
Bass_Guitar	0.989	1.000	0.994	542
Drum_set	1.000	1.000	1.000	547
Electro_Guitar	0.987	0.782	0.873	197
Harmonium	0.995	0.990	0.992	197
Horn	1.000	1.000	1.000	189
Keyboard	1.000	1.000	1.000	306
Mandolin	0.813	0.862	0.837	369
Organ	0.934	0.991	0.962	216
Shakers	1.000	1.000	1.000	204
Trombone	0.948	0.899	0.923	445
flute	1.000	0.998	0.999	558
accuracy			0.952	5305
macro avg	0.959	0.950	0.953	5305
weighted avg	0.954	0.952	0.952	5305

Figure 63: Enhanced CNN Classification Report

The enhanced CNN achieves 95.2% overall accuracy with a macro-F1 of 0.953 and weighted-F1 of 0.952. It is a clear gain over the baseline of 88.5% accuracy. Several classes are perfect, including drum set, horn, keyboard, shakers, flute, and bass guitar, which show F1 of 0.99 and 1.00, with both high precision and recall. The challenging classes also improve, accordion rises to F1 0.907, banjo to 0.943, trombone to 0.923, and mandolin to 0.837. The main remaining weakness is electro guitar, with a 0.782 recall. Overall, scores are strong across instruments. By adding depth, batch normalization, dropout, and the Adam optimizer to the CNN, the generalization is improved. This structure already shows strong performance, further improvement can still be achieved through hyperparameter tuning to find the best configuration.

4.1.3 Hyperparameter Tuning CNN

```
import os, json, itertools, gc, hashlib, time, numpy as np, tensorflow as tf
from tensorflow.keras import Sequential, layers, optimizers, regularizers, callbacks, backend as K
from sklearn.metrics import classification_report, confusion_matrix

# Reproducibility by fix seeds for TF and NumPy
def set_seed(seed=42):
    tf.random.set_seed(seed)
    np.random.seed(seed)

set_seed(42)

# Directory to save all trial logs and checkpoints
RUNS_DIR = "../output/model/cnn_grid_runs"
os.makedirs(RUNS_DIR, exist_ok=True)

# Reduce TensorFlow verbosity in console
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
```

Figure 64: Setup for CNN Hyperparameter Tuning

After the enhanced CNN is trained and evaluated, the next step is hyperparameter tuning to search for a better configuration for the best performance. Before it begins, the random seeds are fixed for TensorFlow and NumPy to keep results reproducible across runs. *RUNS_DIR* is created to store each trial's checkpoints and logs. Finally, TensorFlow's console verbosity is reduced so the tuning loop focuses on key metrics rather than runtime messages, which can crash the VS Code editor if too heavy.

```python
def build_model(input_shape, num_classes, p):
    # Optional L2 regularization
    l2 = regularizers.l2(p["l2_strength"]) if p["l2_strength"] > 0 else None
    ks = p["kernel_size"]   # kernel size tuple
    ps = p["pool_size"]     # pooling size tuple

    model = Sequential(name="enhanced_model_cnn")
    model.add(layers.Input(shape=input_shape))

    # Block 1
    model.add(layers.Conv2D(p["conv_filters"][0], ks, padding="same", kernel_regularizer=l2))
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())
    model.add(layers.MaxPooling2D(pool_size=ps))

    # Block 2
    model.add(layers.Conv2D(p["conv_filters"][1], ks, padding="same", kernel_regularizer=l2))
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())
    model.add(layers.MaxPooling2D(pool_size=ps))

    # Block 3
    model.add(layers.Conv2D(p["conv_filters"][2], ks, padding="same", kernel_regularizer=l2))
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())
    model.add(layers.Dropout(p["dropout_block3"]))

    # Head
    model.add(layers.GlobalAveragePooling2D())
    model.add(layers.Dense(p["dense_units"], activation="relu", kernel_regularizer=l2))
    model.add(layers.Dropout(p["dropout_head"]))
    model.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

    # Optimizer configured per trial
    opt = optimizers.Adam(learning_rate=p["learning_rate"])
    model.compile(
        optimizer=opt,
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model
```

Figure 65: CNN Model Factory

After that, a model factory that turns the enhanced CNN design into a flexible version for hyperparameter search is created. Instead of fixed settings, parameters are read from the dictionary *p*, such as the number of filters per block (*conv_filters*), the convolution kernel size (*kernel_size*), the pool size (*pool_size*), the dropout rates in block 3 and the head, the dense layer width (*dense_units*), optional L2 weight decay (*l2_strength*), and the Adam learning rate. The overall architecture mirrors the enhanced CNN, but each component can now be varied per trial to test different capacities, levels of regularization, and learning dynamics.

```
# 3 x 2 x 2 x 2 x 2 x 2 = 96 trials
param_grid = {
    "conv_filters": [(8,16,32), (16,32,64), (24,48,96)],  # depth per block
    "kernel_size": [(3,3)],                               # 3x3 convs (standard choice)
    "pool_size": [(2,2)],                                 # downsample after blocks
    "dropout_block3": [0.30, 0.40],                       # regularize last conv block
    "dense_units": [64, 128],                             # head width
    "dropout_head": [0.30, 0.50],                         # regularize dense head
    "learning_rate": [3e-4, 1e-4],                        # Adam LR candidates
    "l2_strength": [0.0, 1e-4],                           # weight decay (0 = off)
    "epochs": [30],                                       # fixed training budget per trial
    "batch_size": [32],                                   # keep memory use stable
}
```

Figure 66: CNN Hyperparameter Grid

Then, the hyperparameter grid is set:

- *conv_filters: (8,16,32), (16,32,64), (24,48,96):* Varies model capacity by scaling channels in each block. The smallest tuple is close to the enhanced baseline for good bias-variance balance, the middle increases representation power, and the largest tests whether richer feature banks capture subtle timbre differences without overfitting.
- *kernel_size: (3,3)*: It is fixed because 3×3 is a already strong default for spectrograms. Fixing this keeps the grid compact and focuses compute on other, more influential dimensions.
- *pool_size: (2,2)*: It is also fixed to maintain a gentle down-sampling schedule that preserves useful temporal and spectral detail seen to help in the enhanced model.
- *dropout_block3: 0.30, 0.40*: This tries to tune regularization right after the deepest convolutional block. Testing 0.30 and 0.40 can measure how much stochastic feature suppression helps generalization.
- *dense_units: 64, 128:* The dense units adjust the bottleneck width in the head to control how much the model can mix and recombine global features. 64 offers a compact head, which means fewer parameters and less overfitting, while 128 allows a richer combination.
- *dropout_head: 0.30, 0.50*: This regularizes the classifier head. Comparing 0.30 to 0.50 shows whether aggressive dropout improves validation scores or starts to underfit.
- *learning_rate: 3e-4, 1e-4 (Adam)*: Learning rate tunes the step size for Adam because it is a key factor in convergence speed and final validation performance. 3e-4 is a common strong default for spectrogram CNNs, while 1e-4 is a safer but slower alternative.

- *l2_strength: 0.0, 1e-4*L This tests weight decay as a complementary regularizer to dropout and BN. 1e-4 is a modest penalty that discourages overly large weights, while 0.0 means a no-decay baseline.
- *epochs: 30*: The epoch is fixed to ensure fair and comparable training budgets across trials. With callbacks, this is enough for the validation signal to get good configurations without excessive computation.
- *batch_size: 32:* It is fixed to keep memory use stable and maintain consistent gradient noise across trials.

Overall, the parameter search grid tries to focus on the more impactful factors and keep other aspects constant to control training conditions and, at the same time, balance the use of computational resources.

```
def dict_product(d):
    # Cartesian product over parameter lists to dicts
    keys = list(d.keys())
    for values in itertools.product(*[d[k] for k in keys]):
        yield dict(zip(keys, values))

def short_id_from_params(p):
    # Short, readable ID from params (stable across runs)
    key_str = json.dumps(p, sort_keys=True)
    return hashlib.md5(key_str.encode()).hexdigest()[:8]
```

Figure 67: Generate Parameter Sets and Unique Trial IDs

The next setup is a utility for preparing each trial in the hyperparameter search. The *dict_product* function takes the parameter grid and generates every possible combination as a dictionary to make it easy to loop through all trial settings in an organized way. The *short_id_from_params* function then creates a short and unique ID for each parameter set by hashing its contents. This ID will be used to name log files, checkpoints, or results without writing out the entire parameter dictionary.

```python
def train_one_trial(p, input_shape, num_classes, train_ds, val_ds):
    run_id = short_id_from_params(p)
    run_dir = os.path.join(RUNS_DIR, run_id)
    os.makedirs(run_dir, exist_ok=True)

    # Per-trial file paths
    csv_path = os.path.join(run_dir, "history.csv")
    ckpt_path = os.path.join(run_dir, "best.weights.h5")

    # Callbacks: CSV log, best-weight checkpoint (by val_accuracy), early stopping
    cbs = [
        callbacks.CSVLogger(csv_path, append=False),
        callbacks.ModelCheckpoint(
            ckpt_path,
            save_weights_only=True,
            monitor="val_accuracy",
            mode="max",
            save_best_only=True,
            verbose=0
        ),
        callbacks.EarlyStopping(
            monitor="val_accuracy",
            mode="max",
            patience=6,
            restore_best_weights=True,
            verbose=0
        ),
    ]

    # Build model for this param set
    model = build_model(input_shape, num_classes, p)

    # Train quietly (verbose=0) to avoid console crash from too many text
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=p["epochs"],
        batch_size=p["batch_size"],
        verbose=0,
        callbacks=cbs
    )

    # Ensure we evaluate the best checkpointed weights
    if os.path.exists(ckpt_path):
        model.load_weights(ckpt_path)

    # Validation metrics used for model selection
    val_loss, val_acc = model.evaluate(val_ds, verbose=0)

    # Save a compact JSON summary for later aggregation
    with open(os.path.join(run_dir, "summary.json"), "w") as f:
        json.dump({"params": p, "val_loss": float(val_loss), "val_accuracy": float(val_acc)}, f, indent=2)

    print(f"[{run_id}] val_acc={val_acc:.4f} | params={p}")
    return run_id, val_acc, val_loss, run_dir, model
```

Figure 68: Train, Evaluate, and Log One CNN Trial Function

The next part is to define a single trial of the CNN process of training, evaluating, and logging. Each parameter set is given a unique run ID and a directory where its results will be stored. Inside the created folder, the training history is saved, the best model weights that are based on validation accuracy are checkpointed, and early stopping ensures training halts when no further

improvement is seen. For every trial, the model is built using the parameter dictionary, trained silently to avoid the editor console crash, and then reloaded with the best weights if a checkpoint exists. After training, validation loss and accuracy are calculated as the key metrics for model selection. Finally, a summary file in JSON format is created, which records both the hyperparameters and results. This ensures that each trial is fully documented and can easily be found when reviewing results.

```
def count_trials(pg):
    # Multiply lengths of all value lists to get Cartesian product size
    total = 1
    for k, vals in pg.items():
        total *= len(vals)
    return total

TOTAL_TRIALS = count_trials(param_grid)
print(f"Total Trials: {TOTAL_TRIALS}")
```

```
Total Trials: 96
```

Figure 69: Count Total Trials

This step simply calculates and confirms that the parameter grid will generate a total of 96 trials for the hyperparameter search.

```
import sys, json, os

# Path to CSV that logs each trial's summary
LIVE_CSV = os.path.join(RUNS_DIR, "grid_summary_live.csv")

# Create the CSV with header
if not os.path.exists(LIVE_CSV):
    with open(LIVE_CSV, "w") as f:
        f.write("idx,run_id,val_accuracy,val_loss,params_json\n")
        f.flush(); os.fsync(f.fileno())
```

Figure 70: Prepare Live Summary CSV for Grid Runs

Before tuning starts, a live summary CSV file is prepared to keep track of all grid search trials. The file will record each trial's index, run ID, validation accuracy, validation loss, and parameter settings in JSON format. This ensures that every run is logged in a structured way, to prevent information loss if something undesirable situation happens.

```
# Input shape for log-Mel tensors and containers for results
input_shape = (num_mel_bins, None, 1)
results = []
best = {"val_acc": -1, "run_id": None, "run_dir": None, "params": None}
start_all = time.time()

# Iterate over all parameter combinations
for i, p in enumerate(dict_product(param_grid), start=1):
    K.clear_session(); gc.collect(); set_seed(42)
    t0 = time.time()
    model = None
    try:
        # Train one trial and collect metrics
        run_id, val_acc, val_loss, run_dir, model = train_one_trial(
            p, input_shape, num_classes, train_ds, val_ds
        )
        results.append({"run_id": run_id, "val_accuracy": float(val_acc), "val_loss": float(val_loss), "params": p})

        # Track best model so far and persist weights immediately
        if val_acc > best["val_acc"]:
            best.update({"val_acc": val_acc, "run_id": run_id, "run_dir": run_dir, "params": p})
            best_weights_path = os.path.join(RUNS_DIR, "best_overall.weights.h5")
            model.save_weights(best_weights_path)

        # single-line progress output
        p_show = {
            "filters": p["conv_filters"], "ks": p["kernel_size"], "ps": p["pool_size"],
            "drop3": p["dropout_block3"], "dense": p["dense_units"], "dropH": p["dropout_head"],
            "lr": p["learning_rate"], "l2": p["l2_strength"], "bs": p["batch_size"], "ep": p["epochs"],
        }
        elapsed = time.time() - t0
        line = f"[{i}/{TOTAL_TRIALS}] {run_id} val_acc={val_acc:.4f} val_loss={val_loss:.4f} | {p_show} | {elapsed:.1f}s"
        print(line); sys.stdout.flush()  # force print in notebook

        # Append to progress.log (text) with fsync
        with open(os.path.join(RUNS_DIR, "progress.log"), "a") as f:
            f.write(line + "\n"); f.flush(); os.fsync(f.fileno())

        # Append to live CSV with fsync
        with open(LIVE_CSV, "a") as f:
            params_json = json.dumps(p, sort_keys=True).replace(",", ";")
            f.write(f"{i},{run_id},{val_acc:.6f},{val_loss:.6f},{params_json}\n")
            f.flush(); os.fsync(f.fileno())

    finally:
        # Clean up model and free graph memory between trials
        if model is not None:
            del model
        K.clear_session(); gc.collect()

# Total wall-clock time for the entire grid
total_time = time.time() - start_all
```

Figure 71: Run CNN Grid Search

After all are set up, the tuning process begins by looping through every parameter combination in the grid. For each trial, the session is cleared, and seeds are reset to keep results consistent, then the model is built and trained using the *train_one_trial* function defined earlier. The validation accuracy and loss are recorded, and if the trial produces the best accuracy so far, its weights are saved as the current best model. A compact summary of the trial's key parameters and metrics is printed in console, while detailed logs are written to both a progress log file and the live CSV as backup record. Finally, memory is cleaned up after each run to avoid overload during long searches. This process ensures that all 96 trials are systematically trained, evaluated, and stored, with the best model configuration stored for further evaluation.

```
[350bd261] val_acc=0.9815 | params={'conv_filters': (8, 16, 32), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.3, 'dense_units': 64, 'dropout_head': 0.3, 'learning_rate': 0.0003, 'l2_strength': 0.0, 'epochs':
30, 'batch_size': 32}
[1/96] 350bd261 val_acc=0.9815 val_loss=0.0686 | {'filters': (8, 16, 32), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.3,
'dense': 64, 'dropH': 0.3, 'lr': 0.0003, 'l2': 0.0, 'bs': 32, 'ep': 30} | 407.0s
[791ee877] val_acc=0.9787 | params={'conv_filters': (8, 16, 32), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.3, 'dense_units': 64, 'dropout_head': 0.3, 'learning_rate': 0.0003, 'l2_strength': 0.0001, 'epochs':
30, 'batch_size': 32}
[2/96] 791ee877 val_acc=0.9787 val_loss=0.0922 | {'filters': (8, 16, 32), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.3,
'dense': 64, 'dropH': 0.3, 'lr': 0.0003, 'l2': 0.0001, 'bs': 32, 'ep': 30} | 404.9s
[4b38f5ee] val_acc=0.9627 | params={'conv_filters': (8, 16, 32), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.3, 'dense_units': 64, 'dropout_head': 0.3, 'learning_rate': 0.0001, 'l2_strength': 0.0, 'epochs':
30, 'batch_size': 32}
[3/96] 4b38f5ee val_acc=0.9627 val_loss=0.1377 | {'filters': (8, 16, 32), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.3,
'dense': 64, 'dropH': 0.3, 'lr': 0.0001, 'l2': 0.0, 'bs': 32, 'ep': 30} | 384.2s
[a4d3492f] val_acc=0.9576 | params={'conv_filters': (8, 16, 32), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.3, 'dense_units': 64, 'dropout_head': 0.3, 'learning_rate': 0.0001, 'l2_strength': 0.0001, 'epochs':
30, 'batch_size': 32}
[4/96] a4d3492f val_acc=0.9576 val_loss=0.1664 | {'filters': (8, 16, 32), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.3,
'dense': 64, 'dropH': 0.3, 'lr': 0.0001, 'l2': 0.0001, 'bs': 32, 'ep': 30} | 383.3s
[cbd31a47] val_acc=0.9742 | params={'conv_filters': (8, 16, 32), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.3, 'dense_units': 64, 'dropout_head': 0.5, 'learning_rate': 0.0003, 'l2_strength': 0.0, 'epochs':
30, 'batch_size': 32}
[5/96] cbd31a47 val_acc=0.9742 val_loss=0.0967 | {'filters': (8, 16, 32), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.3,
'dense': 64, 'dropH': 0.5, 'lr': 0.0003, 'l2': 0.0, 'bs': 32, 'ep': 30} | 366.7s
```

Figure 72: CNN First 5 Tuning Trials

```
[5e5e59c1] val_acc=0.9857 | params={'conv_filters': (24, 48, 96), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.4, 'dense_units': 128, 'dropout_head': 0.3, 'learning_rate': 0.0001, 'l2_strength': 0.0001,
'epochs': 30, 'batch_size': 32}
[92/96] 5e5e59c1 val_acc=0.9857 val_loss=0.0848 | {'filters': (24, 48, 96), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.4,
'dense': 128, 'dropH': 0.3, 'lr': 0.0001, 'l2': 0.0001, 'bs': 32, 'ep': 30} | 464.7s
[ab3ec169] val_acc=0.9891 | params={'conv_filters': (24, 48, 96), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.4, 'dense_units': 128, 'dropout_head': 0.5, 'learning_rate': 0.0003, 'l2_strength': 0.0, 'epochs':
30, 'batch_size': 32}
[93/96] ab3ec169 val_acc=0.9891 val_loss=0.0390 | {'filters': (24, 48, 96), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.4,
'dense': 128, 'dropH': 0.5, 'lr': 0.0003, 'l2': 0.0, 'bs': 32, 'ep': 30} | 584.1s
[1b9b5a1f] val_acc=0.9870 | params={'conv_filters': (24, 48, 96), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.4, 'dense_units': 128, 'dropout_head': 0.5, 'learning_rate': 0.0003, 'l2_strength': 0.0001,
'epochs': 30, 'batch_size': 32}
[94/96] 1b9b5a1f val_acc=0.9870 val_loss=0.0767 | {'filters': (24, 48, 96), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.4,
'dense': 128, 'dropH': 0.5, 'lr': 0.0003, 'l2': 0.0001, 'bs': 32, 'ep': 30} | 446.1s
[e2717ad7] val_acc=0.9827 | params={'conv_filters': (24, 48, 96), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.4, 'dense_units': 128, 'dropout_head': 0.5, 'learning_rate': 0.0001, 'l2_strength': 0.0, 'epochs':
30, 'batch_size': 32}
[95/96] e2717ad7 val_acc=0.9827 val_loss=0.0751 | {'filters': (24, 48, 96), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.4,
'dense': 128, 'dropH': 0.5, 'lr': 0.0001, 'l2': 0.0, 'bs': 32, 'ep': 30} | 390.8s
[4b5c99de] val_acc=0.9894 | params={'conv_filters': (24, 48, 96), 'kernel_size': (3, 3), 'pool_size': (2, 2),
'dropout_block3': 0.4, 'dense_units': 128, 'dropout_head': 0.5, 'learning_rate': 0.0001, 'l2_strength': 0.0001,
'epochs': 30, 'batch_size': 32}
[96/96] 4b5c99de val_acc=0.9894 val_loss=0.0670 | {'filters': (24, 48, 96), 'ks': (3, 3), 'ps': (2, 2), 'drop3': 0.4,
'dense': 128, 'dropH': 0.5, 'lr': 0.0001, 'l2': 0.0001, 'bs': 32, 'ep': 30} | 578.7s
```

Figure 73: CNN Last 5 Tuning Trials

The first and last five runs of the CNN hyperparameter tuning are shown in Figure 72 and Figure 73 confirm that the process is successfully executed, with each trial training as designed, logging results, and producing validation scores as expected. Different parameter combinations, such as filter depth, dropout rates, and learning rates, are tested. This shows that the grid search was successfully run for all 96 trials.

```
import pandas as pd

# Read live results, sort by validation accuracy (desc), and show top-10
df_results = pd.read_csv(LIVE_CSV)
df_results = df_results.sort_values("val_accuracy", ascending=False)
pd.set_option("display.max_rows", 12)
pd.set_option("display.max_colwidth", 60)

print("Top 10 Trials by Val Accuracy:")
display(df_results.head(10)[["idx","run_id","val_accuracy","val_loss"]].round(4))

print(f"Trials: {len(df_results)} | Elapsed: {total_time/60:.1f} min")
```

```
Top 10 Trials by Val Accuracy:
```

	idx	run_id	val_accuracy	val_loss
76	77	acb30593	0.9945	0.0221
40	41	ec01b23c	0.9921	0.0300
32	33	ab39039c	0.9913	0.0360
61	62	78040c1a	0.9913	0.0571
89	90	458afd3e	0.9913	0.0622
77	78	9f1a3709	0.9913	0.0658
82	83	de0762cd	0.9910	0.0382
68	69	029f04f9	0.9910	0.0365
75	76	5a79e648	0.9910	0.0656
60	61	49fad348	0.9908	0.0316

```
Trials: 96 | Elapsed: 742.0 min
```

Figure 74: CNN Hyperparameter Tuning Summary

Figure 74 shows the CNN hyperparameter tuning summary table is sorted by the best validation accuracy. Out of the 96 trials, the top-performing model achieved a validation accuracy of 99.45% with a very low loss of 0.0221, while several other trials also performed very well in the 99.0 to 99.2% range. This shows that the tuning process successfully explored the parameter space and found many high-performing configurations and with the best one. The overall runtime for all trials is 742 minutes, about 12 hours. The results show that the CNN can be optimized to reach higher accuracy, with the top few trials standing out.

```
    best = df_results.iloc[0]
    params_str = str(best["params_json"]).replace(";", ",")
    best_params = json.loads(params_str)

    print("\nBest hyperparameters:")
    print(json.dumps(best_params, indent=2, sort_keys=True))
```

```
Best hyperparameters:
{
  "batch_size": 32,
  "conv_filters": [
    24,
    48,
    96
  ],
  "dense_units": 128,
  "dropout_block3": 0.3,
  "dropout_head": 0.5,
  "epochs": 30,
  "kernel_size": [
    3,
    3
  ],
  "l2_strength": 0.0,
  "learning_rate": 0.0003,
  "pool_size": [
    2,
    2
  ]
}
```

Figure 75: Print CNN Best Hyperparameter

The best setup uses a *(24, 48, 96)* filter progression that has higher capacity than the enhanced baseline, 3×3 kernels with 2×2 pooling, a dense head of 128 units, dropout 0.3 after the deepest *conv* block, dropout 0.5 in the head, and no L2 weight decay. Training is done with Adam at 3e-4, batch size 32, for 30 epochs. This best-tuned model is moderately larger and produces the highest validation accuracy in the grid search.

```python
# Reload best model to plot curves and evaluate
import os, json, pandas as pd, numpy as np
import tensorflow as tf
from tensorflow.keras import backend as K
RUNS_DIR = "../output/model/cnn_grid_runs"
# Recover best params/run_id even if the session restarted
def _recover_best(RUNS_DIR, best_dict=None):
    # If current session tracked a best dict, use it
    if best_dict and best_dict.get("run_id"):
        return best_dict["run_id"], best_dict["params"]

    # Otherwise, read the leaderboard CSV (prefer final, fallback to live)
    grid_csv = os.path.join(RUNS_DIR, "grid_summary.csv")
    if not os.path.exists(grid_csv):
        grid_csv = os.path.join(RUNS_DIR, "grid_summary_live.csv")

    df = pd.read_csv(grid_csv)
    top = df.sort_values("val_accuracy", ascending=False).iloc[0]
    run_id = top["run_id"]

    # Params may be embedded as JSON or stored in the trial's summary.json
    if "params_json" in df.columns:
        params = json.loads(top["params_json"])
    else:
        with open(os.path.join(RUNS_DIR, run_id, "summary.json"), "r") as f:
            params = json.load(f)["params"]
    return run_id, params

best_run_id, best_params = _recover_best(RUNS_DIR, best)

# Build and load weights for the best model
best_model = build_model(input_shape, num_classes, best_params)
best_weights = os.path.join(RUNS_DIR, "best_overall.weights.h5")
best_model.load_weights(best_weights)
```

Figure 76: Reload Best Tuned CNN and Restore Weights

Next, the best CNN from the grid search is reloaded so its structure can be displayed and inspected, and evaluated again. First, *_recover_best* finds the top trial by reading the leaderboard CSV in the runs folder and retrieves its *run_id* plus the exact hyperparameters. With those parameters, *build_model* reconstructs the same architecture, and the previously saved best weights *(best_overall.weights.h5)* are loaded into it. The result is a restoration of the highest-performing tuned model, even after a fresh session, it is still ready for plotting curves and running final test evaluations.

```
best_model.summary()
```

```
Model: "enhanced_model_cnn"
```

Layer (type)	Output Shape	Param #
conv2d_18 (Conv2D)	(None, 64, None, 24)	240
batch_normalization_18 (BatchNormalization)	(None, 64, None, 24)	96
re_lu_18 (ReLU)	(None, 64, None, 24)	0
max_pooling2d_12 (MaxPooling2D)	(None, 32, None, 24)	0
conv2d_19 (Conv2D)	(None, 32, None, 48)	10,416
batch_normalization_19 (BatchNormalization)	(None, 32, None, 48)	192
re_lu_19 (ReLU)	(None, 32, None, 48)	0
max_pooling2d_13 (MaxPooling2D)	(None, 16, None, 48)	0
conv2d_20 (Conv2D)	(None, 16, None, 96)	41,568
batch_normalization_20 (BatchNormalization)	(None, 16, None, 96)	384
re_lu_20 (ReLU)	(None, 16, None, 96)	0
dropout_12 (Dropout)	(None, 16, None, 96)	0
global_average_pooling2d_6 (GlobalAveragePooling2D)	(None, 96)	0
dense_6 (Dense)	(None, 128)	12,416
dropout_13 (Dropout)	(None, 128)	0
Classifier (Dense)	(None, 14)	1,806

```
Total params: 67,118 (262.18 KB)

Trainable params: 66,782 (260.87 KB)

Non-trainable params: 336 (1.31 KB)
```

Figure 77: Best Tuned CNN Model Summary

Figure 77 shows the final tuned CNN architecture summary that achieved the best performance during hyperparameter search. Compared to the baseline and enhanced models, this best model uses larger filter sizes and a wider dense layer, which increases its capacity to learn detailed time-frequency patterns from the log-Mel spectrograms.

The network begins with three convolutional blocks, as it follows the enhanced CNN model design. The first convolutional layer applies 24 filters of size 3×3, followed by batch normalization and ReLU activation to stabilize training and add non-linearity. A max-pooling

step then reduces the Mel dimension to reduce complexity. The second block expands the filters to 48, repeating the same structure of normalization, activation, and pooling to allow the model to capture increasingly complex patterns. The third block further increases the filters to 96 and with dropout.

After the convolutional stages, a global average pooling layer condenses each feature map into a single value to summarize the learned features in a compact form without adding large numbers of parameters. This is followed by a dense layer with 128 units as a bottleneck, where the model combines and interprets global features. Another dropout is applied here for regularization before the final softmax classifier, which outputs probabilities across the 14 instrument classes.

Overall, this tuned model has 67,118 parameters. The combination of stronger filter banks, a wider dense layer, and carefully placed batch normalization and dropout layers makes this structure the best performer during the tuning process.

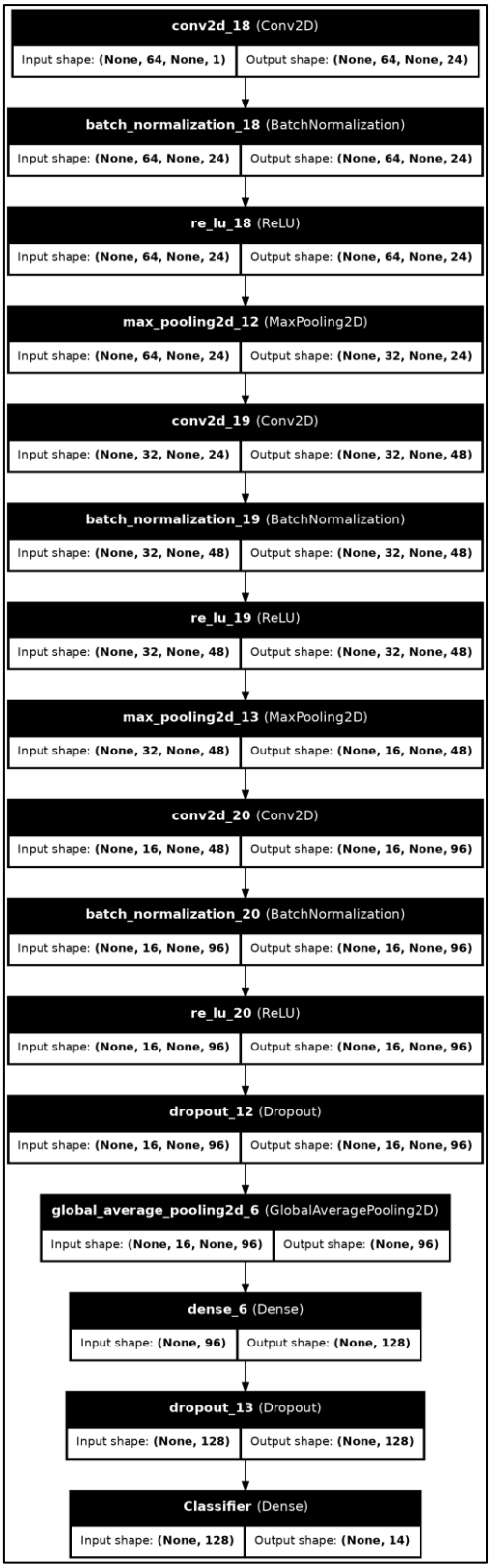

Figure 78: Best Tuned CNN Model Architecture

Figure 78 shows the architecture of the best-tuned CNN model, which follows the enhanced CNN structure but with the optimal parameters found during tuning. It is the final configuration that achieved the highest validation accuracy.

```python
# Recreate a History-like object from the best trial's history.csv for plotting
hist_path = os.path.join(RUNS_DIR, best_run_id, "history.csv")
if os.path.exists(hist_path):
    hist_df = pd.read_csv(hist_path)

    # Minimal shim that mimics Keras History structure
    class _SimpleHistory:
        pass
    history_for_plot = _SimpleHistory()

    # Convert CSV columns to lists; ignore 'epoch' if present
    hist_cols = [c for c in hist_df.columns if c != "epoch"]
    history_for_plot.history = {c: hist_df[c].tolist() for c in hist_cols}

# Plot training curves for the best tuned CNN
plot_training_curves(
    history_for_plot,
    metrics=("loss", "accuracy"),
    suptitle=f"Best Tuned CNN (run on id: {best_run_id})"
)
```

Figure 79: Recreate History for Best Tuned CNN

The training history from the best trial is reconstructed by reading its saved *history.csv* and wrapping the columns into a History-like object. With this, the previously defined evaluation functions, like *plot_training_curves,* can be reused to visualize the best model's loss and accuracy over epochs.

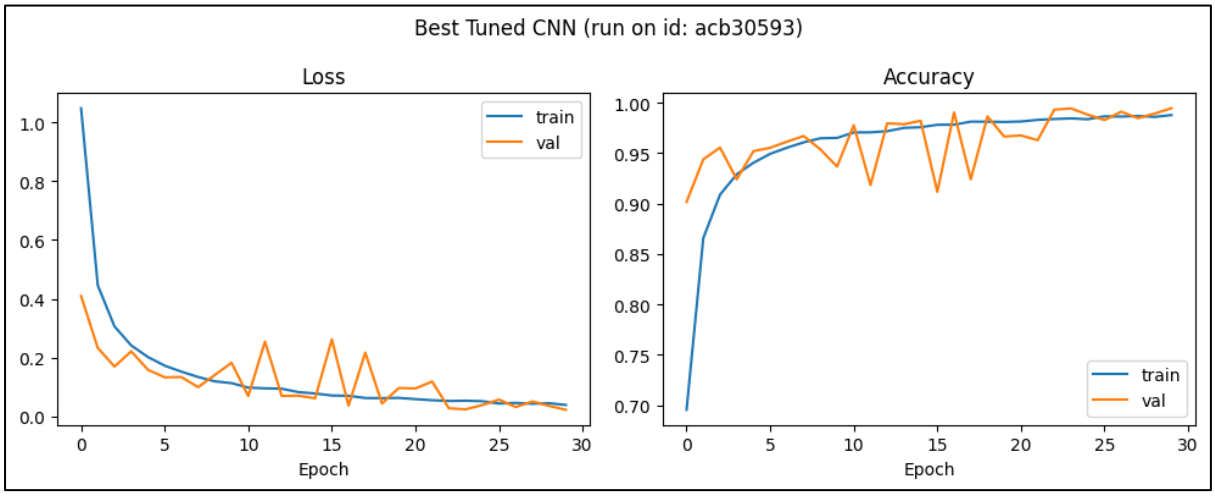

Figure 80: Best Tuned CNN Training Curves

The training curves of the best-tuned CNN show a very good performance among all trained CNNs. The loss plot drops quickly within the first few epochs for both training and validation, then continues to decrease steadily, and the fluctuations are smaller in validation loss. The accuracy plot also shows a quick rise, reaching above 95% within just a few epochs, and then stabilizing around 99% accuracy for both training and validation by the end of training. The close alignment of the two curves means that the model generalizes well, and there is no overfitting. The tuned hyperparameters give a balanced and effective configuration.

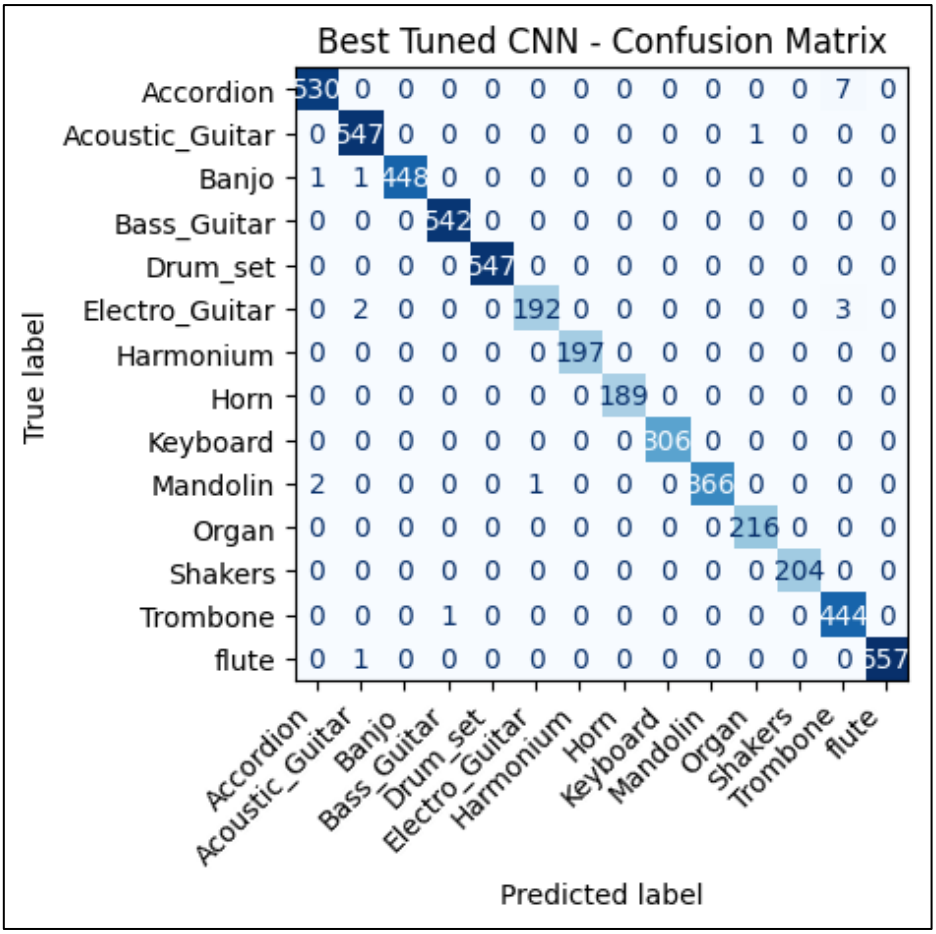

Figure 81: Best Tuned CNN Confusion Matrix

Figure 81 shows that the model classifies almost all instruments correctly, with predictions concentrated on the diagonal. Most classes, such as drum set, keyboard, shakers, organ, and harmonium, already achieve perfect recognition with zero misclassifications. Even more challenging instruments defined previously, like accordion, banjo, and trombone, show only less errors, and the previously weaker electric guitar now reaches strong performance with only 5 mislabeling.

Classification report:

	precision	recall	f1-score	support
Accordion	0.994	0.987	0.991	537
Acoustic_Guitar	0.993	0.998	0.995	548
Banjo	1.000	0.996	0.998	450
Bass_Guitar	0.998	1.000	0.999	542
Drum_set	1.000	1.000	1.000	547
Electro_Guitar	0.995	0.975	0.985	197
Harmonium	1.000	1.000	1.000	197
Horn	1.000	1.000	1.000	189
Keyboard	1.000	1.000	1.000	306
Mandolin	1.000	0.992	0.996	369
Organ	0.995	1.000	0.998	216
Shakers	1.000	1.000	1.000	204
Trombone	0.978	0.998	0.988	445
flute	1.000	0.998	0.999	558
accuracy			0.996	5305
macro avg	0.997	0.996	0.996	5305
weighted avg	0.996	0.996	0.996	5305

Figure 82: Best Tuned CNN Classification Report

The best-tuned CNN achieves 99.6% overall accuracy with both macro-F1 and weighted-F1, also at 0.996. It shows very strong performance across classes. Most instruments are essentially perfect with an F1 of 1.000, like drum set, harmonium, horn, keyboard, shakers, flute, bass guitar, banjo, and organ. While the remaining classes are only a fraction below perfect, as already seen in the confusion matrix, the electro guitar has an F1 of 0.985 and a recall of 0.975, and the trombone has 0.988 F1 and 0.978 precision. The near-equality of macro and weighted averages also shows that high accuracy is not driven only by frequent classes. The rare classes also perform well. Compared to the enhanced CNN about 95.2% accuracy, this tuned configuration sets a very strong benchmark for the dataset.

4.1.4 Summary

To conclude, the CNN experiments progressed from a simple baseline to an enhanced, tuned model that can show clear improvements at each stage. The baseline CNN provided a starting point with about 88.5% accuracy. It demonstrates that even a shallow structure can learn basic timbre features but struggles with some instruments. The enhanced CNN introduced batch normalization, dropout, a deeper filter progression, and the Adam optimizer, which lifted performance significantly to around 95.2% accuracy. It reduces errors in difficult classes while

generalizing better. Finally, through grid search hyperparameter tuning, the CNN reached 99.6% accuracy, with strong precision, recall, and F1 across all 14 instrument classes. This progression shows that of deeper design choices and tuning successfully make the CNN as a highly capable model for musical instrument classification.

4.2 RNN-BiGRU

4.2.1 Baseline RNN-BiGRU

```
from tensorflow.keras import Sequential, layers, optimizers

# Input: mel spectrograms shaped (num_mel_bins, time, 1)
input_shape = (num_mel_bins, None, 1)

base_model_rnn = Sequential(name="base_model_rnn")
base_model_rnn.add(layers.Input(shape=input_shape))

# Flatten frequency/channel so RNN sees sequences of mel bins
# From (batch, mel_bins, time, 1) -> (batch, time, mel_bins)
base_model_rnn.add(layers.Permute((2, 1, 3)))
base_model_rnn.add(layers.Lambda(lambda x: tf.squeeze(x, axis=-1), name="squeeze_chan"))

# Block 1: BiGRU
base_model_rnn.add(layers.Bidirectional(layers.GRU(8, return_sequences=True), name="bigru_8"))

# Block 2: BiGRU
base_model_rnn.add(layers.Bidirectional(layers.GRU(8), name="bigru2_8"))

# Head
base_model_rnn.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

# Use same optimizer/loss as CNN baseline
base_model_rnn.compile(
    optimizer=optimizers.SGD(learning_rate=0.01),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Figure 83: Build Baseline RNN-BiGRU Model

After completing the CNN studies, the second proposed model focuses on temporal modeling with an RNN together with BiGRU. Whereas the CNN learns local time-frequency patterns from spectrogram images, the RNN treats each clip as a sequence of frames and learns how the sound evolves across time, such as attacks, decays, tremolo, strums, and rhythmic pulses. The input remains the log-Mel spectrogram with shape *(mels, time, 1)*, but two simple reshapes are needed to make it ready, which are *Permute((2,1,3))* that swaps axes so time becomes the leading dimension, and squeeze removes the singleton channel, becoming *(time, mels)*. This is to match the recurrent layers' input requirement, that each time step is a 64-dimensional Mel vector summarizing the spectrum at that moment.

Two stacked BiGRU blocks form the backbone of this neural network. Bidirectionality runs one GRU forward and another backward over the sequence, which can let the model use past and future context within the 3-second clip. This will be ideal for offline classification where the whole clip is available. GRUs are chosen over LSTMs here because they use fewer gates and parameters. It can also train faster theoretically, and often matches LSTM accuracy on short to medium sequences. The first BiGRU has *return_sequences=True*, which passes a time series of hidden states to the second BiGRU, so a longer-range structure can be built hierarchically. The second BiGRU outputs a single summary vector, which will become its final state after looking at both ways. A final softmax Dense layer maps that summary to the 14 instrument classes.

The parameters set for this baseline model are also intentionally small, with only 8 units per GRU direction to mirror the CNN baseline pattern, which starts small to see data sensitivity and overfitting risk before adding capacity. No convolution or pooling is used in this baseline so that the contribution of pure temporal modeling is isolated. The optimizer and loss match the CNN baseline by using SGD with a learning rate of 0.01, sparse cross entropy, and accuracy, and also try to provide an apples-to-apples comparison of learning dynamics between the two baselines. Overall, this design checks whether BiGRU performs well or not of the way sounds change over time, which adds something different from the CNN. While CNNs are good at spotting shapes and patterns in the spectrogram, the BiGRU looks at how those patterns move and evolve, like the rise and fall of notes or rhythms.

Model: "base_model_rnn"

Layer (type)	Output Shape	Param #
permute_1 (Permute)	(None, None, 64, 1)	0
squeeze_chan (Lambda)	(None, None, 64)	0
bigru_8 (Bidirectional)	(None, None, 16)	3,552
bigru2_8 (Bidirectional)	(None, 16)	1,248
Classifier (Dense)	(None, 14)	238

Total params: 5,038 (19.68 KB)

Trainable params: 5,038 (19.68 KB)

Non-trainable params: 0 (0.00 B)

Figure 84: Baseline RNN-BiGRU Model Summary

The model summary shows that the baseline BiGRU network is very lightweight, with only 5,038 trainable parameters, and will be fast to train. The first two layers are Permute and Lambda, which reshape the input so the GRU layers can process the time sequence correctly without adding parameters. The two BiGRU layers together form the main part of the model, using about 4,800 parameters to capture how the Mel features change over time. Finally, a Dense classifier with 14 output nodes maps the learned temporal patterns to the instrument classes.

Compared to the CNN baseline, which only has 790 trainable parameters, the baseline BiGRU model is larger with 5,038 parameters. This increase maybe comes from the recurrent layers because each GRU unit includes multiple internal gates and weights to manage temporal dependencies in the sequence. While the CNN baseline focuses on learning short, local patterns from the spectrogram as small image patches, the BiGRU baseline adds more complexity to capture how sound features evolve over time.

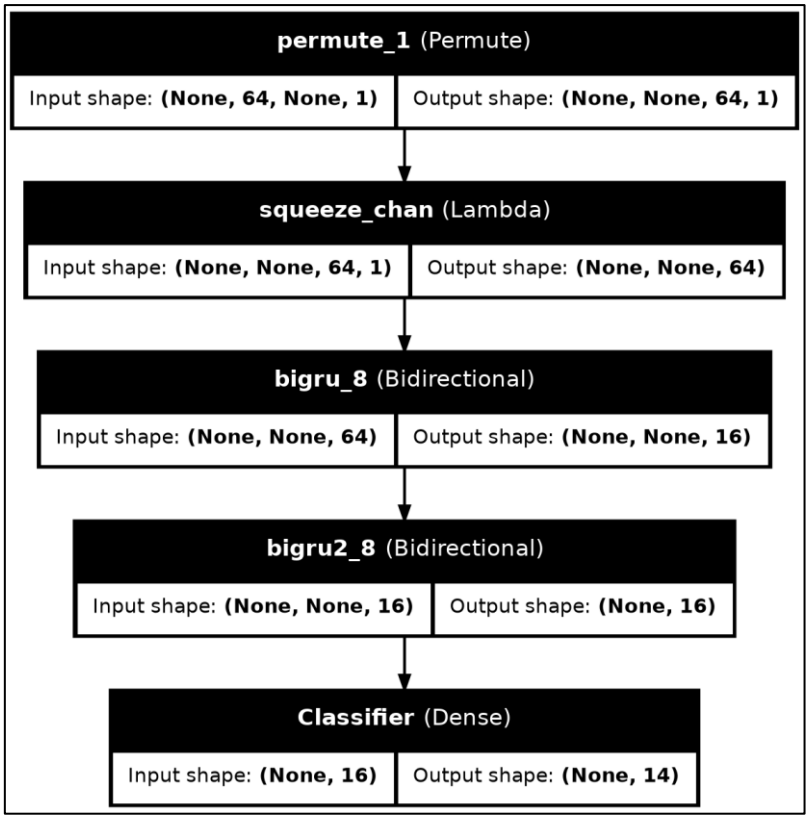

Figure 85: Baseline RNN-BiGRU Model Architecture

Figure 85 shows the baseline BiGRU model flow, where the spectrogram is first rearranged into time sequences, then passed through two stacked bidirectional GRU layers to capture temporal context, and finally classified by a dense layer into 14 instrument classes, as what already explained in the baseline model design.

```
from tensorflow.keras import callbacks

# Log epoch-by-epoch metrics to a CSV
log = callbacks.CSVLogger("../output/training_log/base_model_rnn.csv", append=False)

# Train baseline RNN on log-Mel inputs
history = base_model_rnn.fit(
    train_ds,
    validation_data=val_ds,
    epochs=30,
    callbacks=log,
    verbose=1
)
```

Figure 86: Train Baseline RNN-BiGRU

Then, the training process of the baseline model starts on the log-Mel inputs for 30 epochs. Same as the CNN baseline, a *CSVLogger* saves epoch-by-epoch metrics, which are loss and accuracy for training and validation, to a file. The call to *fit(...)* will start iterating over *train_ds*, updating weights each batch, and evaluating on *val_ds* at the end of every epoch. The progress is printed to the console via *verbose=1*.

```
Epoch 1/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 55s 69ms/step - accuracy: 0.4838 - loss: 1.7648 - val_accuracy: 0.6017 - val_loss: 1.3750
Epoch 2/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.6744 - loss: 1.1722 - val_accuracy: 0.7374 - val_loss: 1.0142
Epoch 3/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 55s 71ms/step - accuracy: 0.7649 - loss: 0.8918 - val_accuracy: 0.7908 - val_loss: 0.7953
Epoch 4/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 54s 70ms/step - accuracy: 0.8079 - loss: 0.7213 - val_accuracy: 0.8215 - val_loss: 0.6613
Epoch 5/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 56s 73ms/step - accuracy: 0.8300 - loss: 0.6143 - val_accuracy: 0.8320 - val_loss: 0.5940
```

Figure 87: Baseline RNN-BiGRU First 5 Epoch

```
Epoch 26/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 57s 74ms/step - accuracy: 0.9367 - loss: 0.2257 - val_accuracy: 0.9274 - val_loss: 0.2465
Epoch 27/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 60s 78ms/step - accuracy: 0.9403 - loss: 0.2104 - val_accuracy: 0.9024 - val_loss: 0.3351
Epoch 28/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 56s 72ms/step - accuracy: 0.9455 - loss: 0.1960 - val_accuracy: 0.9344 - val_loss: 0.2245
Epoch 29/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 61s 78ms/step - accuracy: 0.9476 - loss: 0.1858 - val_accuracy: 0.9367 - val_loss: 0.2170
Epoch 30/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 55s 71ms/step - accuracy: 0.9481 - loss: 0.1833 - val_accuracy: 0.9376 - val_loss: 0.2146
```

Figure 88: Baseline RNN-BiGRU Last 5 Epoch

The baseline RNN-BiGRU training log starts with low accuracy at around 48% in the first epoch, but quickly improves and reaches over 80% by the fifth epoch. By the final few epochs, training accuracy already stabilizes around 95%, with validation accuracy around 93 to 94%. The model can generalize well without severe overfitting. The decrease in training and validation loss across epochs also shows that the model effectively captures sequential relationships in the spectrogram inputs, as a baseline model, to be honest, this performance is good enough already.

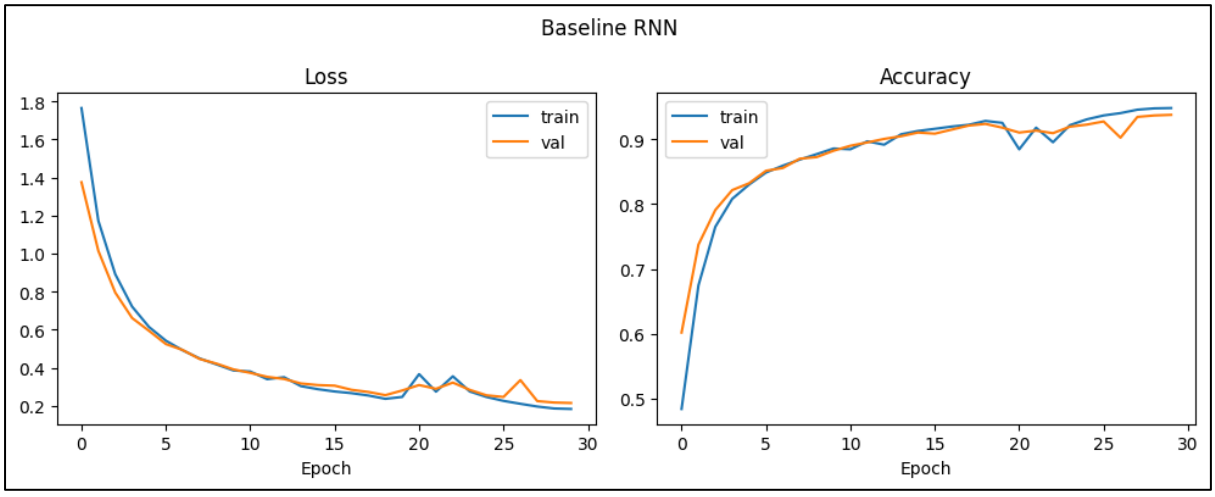

Figure 89: Baseline RNN-BiGRU Training Curves

The training curves for the baseline RNN-BiGRU show that both training and validation loss drop consistently across epochs, while accuracy rises until 95%, as shown in the training log already. The training and validation lines track closely without large gaps, which means the

model is learning effectively without severe overfitting (although validation accuracy is lower than training, but still within 5%). The accuracy curve rises in the early epochs and then stabilizes.

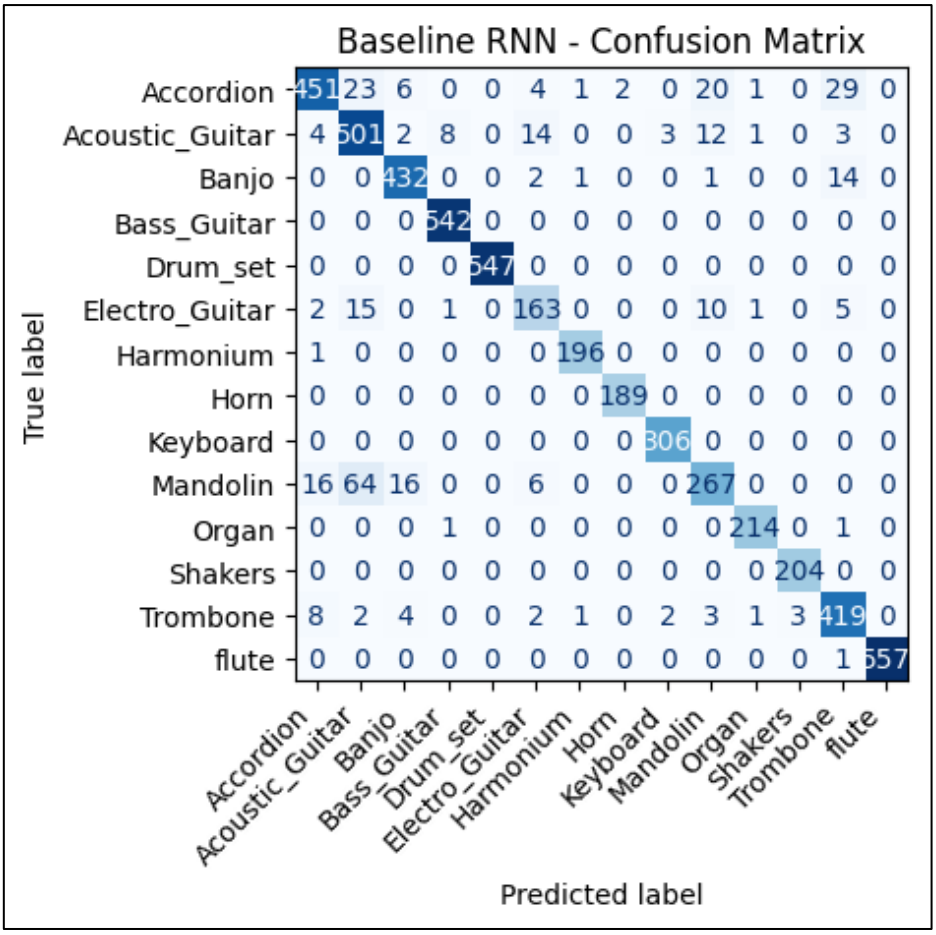

Figure 90: Baseline RNN-BiGRU Confusion Matrix

Most instruments like the drum set, keyboard, flute, and harmonium have very strong diagonal values. They are correctly classified most of the time. However, there are some misclassifications in instruments like accordion (67 wrong classifications), mandolin (105), trombone (88), and electro guitar (71), where predictions spread into other classes. Through observation, the accordion often gets confused with the mandolin, and electro guitar has overlaps with the acoustic Guitar.

Classification report:

	precision	recall	f1-score	support
Accordion	0.936	0.840	0.885	537
Acoustic_Guitar	0.828	0.914	0.869	548
Banjo	0.939	0.960	0.949	450
Bass_Guitar	0.982	1.000	0.991	542
Drum_set	1.000	1.000	1.000	547
Electro_Guitar	0.853	0.827	0.840	197
Harmonium	0.985	0.995	0.990	197
Horn	0.990	1.000	0.995	189
Keyboard	0.984	1.000	0.992	306
Mandolin	0.853	0.724	0.783	369
Organ	0.982	0.991	0.986	216
Shakers	0.986	1.000	0.993	204
Trombone	0.888	0.942	0.914	445
flute	1.000	0.998	0.999	558
accuracy			0.940	5305
macro avg	0.943	0.942	0.942	5305
weighted avg	0.940	0.940	0.939	5305

Figure 91: Baseline RNN-BiGRU Classification Report

The baseline CNN has 88.5% overall accuracy, with a macro-F1 of 0.893 and weighted-F1 of 0.888. The performance is good but uneven across classes. Some instruments are reliable, like the drum set with 1.00 F1, the flute with 0.996 F1, the keyboard with 0.992 F1, the harmonium with 0.987 F1, the bass guitar with 0.973 F1, and the shakers with 0.993 F1. The weak spots are the electro guitar again with a low recall of 0.640, mandolin with the lowest F1 of 0.623, accordion with 0.772 F1, and trombone with 0.864 F1. Overall, precision is high for most classes, but recall is not so good, meaning missed positives rather than many false alarms.

4.2.2 Enhanced RNN-BiGRU

```
import tensorflow as tf
from tensorflow.keras import Sequential, layers, optimizers

input_shape = (num_mel_bins, None, 1)

enhanced_model_rnn = Sequential(name="enhanced_model_rnn")
enhanced_model_rnn.add(layers.Input(shape=input_shape))

# Adapt CNN-style tensors to RNN sequence format
enhanced_model_rnn.add(layers.Permute((2, 1, 3), name="permute_time_first"))     # (time, mel, 1)
enhanced_model_rnn.add(layers.Lambda(lambda x: tf.squeeze(x, axis=-1), name="squeeze_chan"))  # (time, mel)

# Block 1: BiGRU with dropout, return full sequence
enhanced_model_rnn.add(layers.Bidirectional(
    layers.GRU(16, dropout=0.3, return_sequences=True),
    name=f"bigru_16"
))

# Block 2: BiGRU with dropout, last output only
enhanced_model_rnn.add(layers.Bidirectional(
    layers.GRU(16, dropout=0.3, return_sequences=False),
    name=f"bigru2_16"
))

# Small dense head
enhanced_model_rnn.add(layers.Dense(32, activation="relu", name="head_dense_32"))
enhanced_model_rnn.add(layers.Dropout(0.3, name="head_dropout"))

# Classifier
enhanced_model_rnn.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

# Optimizer
enhanced_model_rnn.compile(
    optimizer=optimizers.Adam(learning_rate=3e-4),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
```

Figure 92: Build Enhanced RNN-BiGRU Model

To enhance the BiGRU model, the sequence learner should be made deeper and better regularized. The input format is kept consistent with the earlier pipeline. The spectrogram tensor is first rearranged, so the time axis comes first using *Permute((2,1,3))*, then the singleton channel is squeezed away with a clean *(time, mels)* sequence for recurrent layers. Because every clip is exactly 3 seconds and has a consistent number of frames, so masking layer is not added.

The core of the network is the same as baseline, which is a two-block BiGRU stack with 16 units per direction in each block. The first BiGRU uses *return_sequences=True* to pass the full sequence of hidden states forward to let the next layer build longer-range patterns on top of shorter ones. The second BiGRU returns only its final representation. 0.3 dropout in each GRU

can provide inline regularization so the model does not overfit to specific transients or note shapes.

Above the recurrent stack is a small dense head *(Dense(32, relu) with Dropout(0.3))*. This adds a non-linear mixing stage that can separate instrument signatures after temporal modeling without making the model heavy. The final *Dense(num_classes, softmax)* layer produces the 14-class probabilities. Training uses Adam with a 0.0003 learning rate with *sparse_categorical_crossentropy*, which typically converges faster and more stably than plain SGD for sequence models.

In short, compared to the baseline BiGRU that only has 8 units with no head and uses SGD, this version doubles the capacity, adds dropout at the right places, has a compact head, and uses Adam. Theoretically, this model can present better performance than the baseline.

```
enhanced_model_rnn.summary()
✓ 0.1s
```

Model: "enhanced_model_rnn"

Layer (type)	Output Shape	Param #
permute_time_first (Permute)	(None, None, 64, 1)	0
squeeze_chan (Lambda)	(None, None, 64)	0
bigru_16 (Bidirectional)	(None, None, 32)	7,872
bigru2_16 (Bidirectional)	(None, 32)	4,800
head_dense_32 (Dense)	(None, 32)	1,056
head_dropout (Dropout)	(None, 32)	0
Classifier (Dense)	(None, 14)	462

Total params: 14,190 (55.43 KB)

Trainable params: 14,190 (55.43 KB)

Non-trainable params: 0 (0.00 B)

Figure 93: Enhanced RNN-BiGRU Model Summary

The model summary shows the enhanced BiGRU is still compact with 14,190 trainable parameters, but larger than the baseline RNN-BiGRU, that only about 5,000. The first two

layers rearrange the input, two BiGRUs is the sequence core, together they account for most of the parameters with respectively 7,872 and 4,800. A small dense head has 1,056 params and adds a learnable mixing stage before the 14-way classifier with 462 params. Output shapes confirm the flow from time-wise sequences to a compact vector and finally to class probabilities.

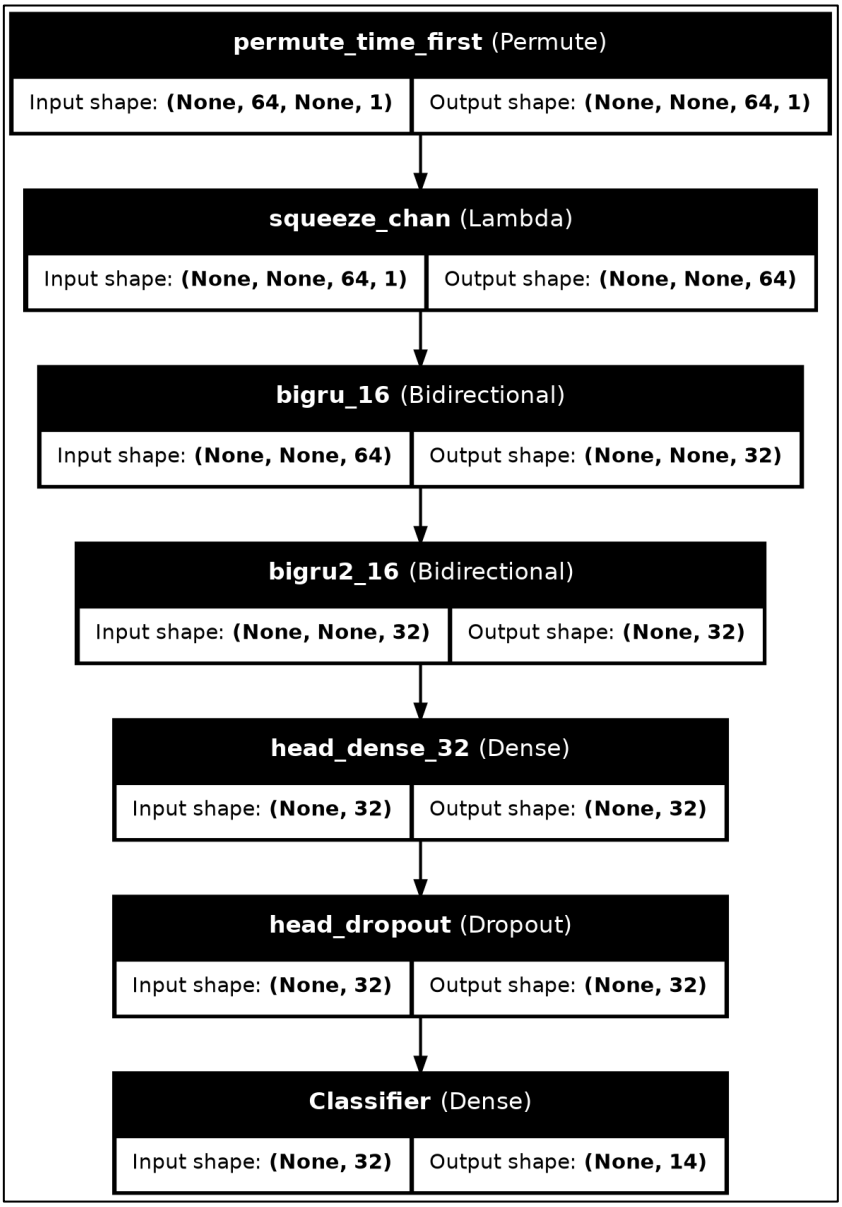

Figure 94: Enhanced RNN-BiGRU Model Architecture

Figure 94 shows the flow of the enhanced BiGRU, just for display purposes and showing how the model combines sequential learning with regularization for stronger classification.

```python
from tensorflow.keras import callbacks

# Stop when validation accuracy stops improving
es = callbacks.EarlyStopping(
    monitor="val_accuracy",
    mode="max",
    patience=6,
    restore_best_weights=True,
    verbose=1
)

# Halve the learning rate if validation loss plateaus
rlr = callbacks.ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.5,
    patience=3,
    verbose=1
)

# Log epoch metrics to CSV
log = callbacks.CSVLogger("../output/training_log/enhanced_model_rnn.csv", append=False)

# Train the enhanced RNN
history = enhanced_model_rnn.fit(
    train_ds,
    validation_data=val_ds,
    epochs=30,
    callbacks=[es, rlr, log],
    verbose=1
)
```

Figure 95: Train Enhanced RNN-BiGRU

The training method for the enhanced BiGRU is the same as the enhanced CNN. Three callbacks manage learning and logging while *fit(...)* runs for up to 30 epochs. *EarlyStopping* see validation accuracy and stops training if it doesn't improve for 6 epochs, then restores the best weights so evaluation uses the strongest checkpoint. *ReduceLROnPlateau* monitors validation loss and reduces the learning rate whenever progress stalls for 3 epochs to help the Adam optimizer fine-tune instead of bouncing around a shallow minimum. Lastly, *CSVLogger* records training and validation loss and accuracy each epoch to a CSV.

```
Epoch 1/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.3801 - loss: 1.8858 - val_accuracy: 0.6271 - val_loss: 1.1564 - learning_rate: 3.0000e-04
Epoch 2/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 57s 74ms/step - accuracy: 0.5550 - loss: 1.2851 - val_accuracy: 0.7184 - val_loss: 0.8227 - learning_rate: 3.0000e-04
Epoch 3/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 63s 81ms/step - accuracy: 0.6378 - loss: 1.0379 - val_accuracy: 0.7640 - val_loss: 0.6705 - learning_rate: 3.0000e-04
Epoch 4/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 71s 92ms/step - accuracy: 0.6937 - loss: 0.8733 - val_accuracy: 0.7868 - val_loss: 0.5765 - learning_rate: 3.0000e-04
Epoch 5/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 69s 90ms/step - accuracy: 0.7270 - loss: 0.7863 - val_accuracy: 0.8196 - val_loss: 0.5035 - learning_rate: 3.0000e-04
```

Figure 96: Enhanced RNN-BiGRU First 5 Epoch

```
Epoch 26/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 66s 86ms/step - accuracy: 0.8828 - loss: 0.3610 - val_accuracy: 0.9384 - val_loss: 0.1943 - learning_rate: 3.0000e-04
Epoch 27/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 73s 95ms/step - accuracy: 0.8835 - loss: 0.3548 - val_accuracy: 0.9393 - val_loss: 0.1921 - learning_rate: 3.0000e-04
Epoch 28/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 70s 90ms/step - accuracy: 0.8859 - loss: 0.3541 - val_accuracy: 0.9401 - val_loss: 0.1850 - learning_rate: 3.0000e-04
Epoch 29/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 80s 104ms/step - accuracy: 0.8885 - loss: 0.3435 - val_accuracy: 0.9399 - val_loss: 0.1823 - learning_rate: 3.0000e-04
Epoch 30/30
774/774 ━━━━━━━━━━━━━━━━━━━━ 79s 102ms/step - accuracy: 0.8917 - loss: 0.3337 - val_accuracy: 0.9425 - val_loss: 0.1842 - learning_rate: 3.0000e-04
Restoring model weights from the end of the best epoch: 30.
```

Figure 97: Enhanced RNN-BiGRU Last 5 Epoch

In the first five epochs of the enhanced RNN-BiGRU, the training accuracy also rises quickly, just like the previous model, from about 38% to 73%, while validation accuracy also improves from around 63% to 82%. The losses steadily decrease on both training and validation, which shows the model is learning effectively from the start. By the final five epochs, the model stabilizes with training accuracy around 89% and validation accuracy at about 94%. Validation loss remains low. This means that the model has strong generalization and no major overfitting.

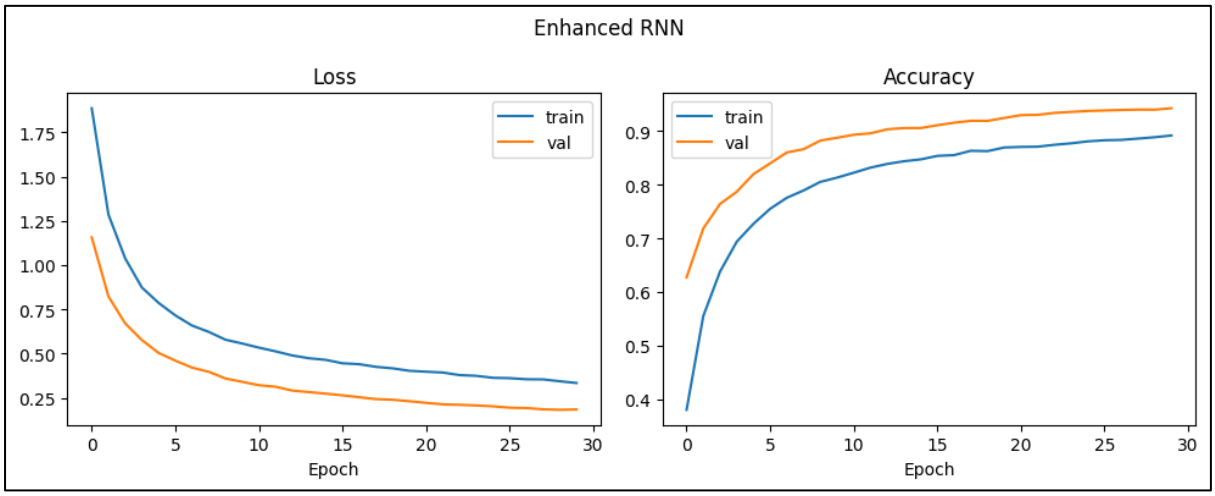

Figure 98: Enhanced RNN-BiGRU Training Curves

The training curves of the enhanced RNN-BiGRU show the improvement across the 30 epochs. The loss graph has a consistent decrease for both training and validation, with validation loss dropping faster and settling lower than training loss. This means that the model generalizes well without overfitting. The accuracy graph shows validation accuracy rising in the early epochs and stabilizing above 94% as shown in the training log, while training accuracy rises more gradually and levels off around 90%. The validation performance outperforms training, becoming a sign of strong generalization on unseen data.

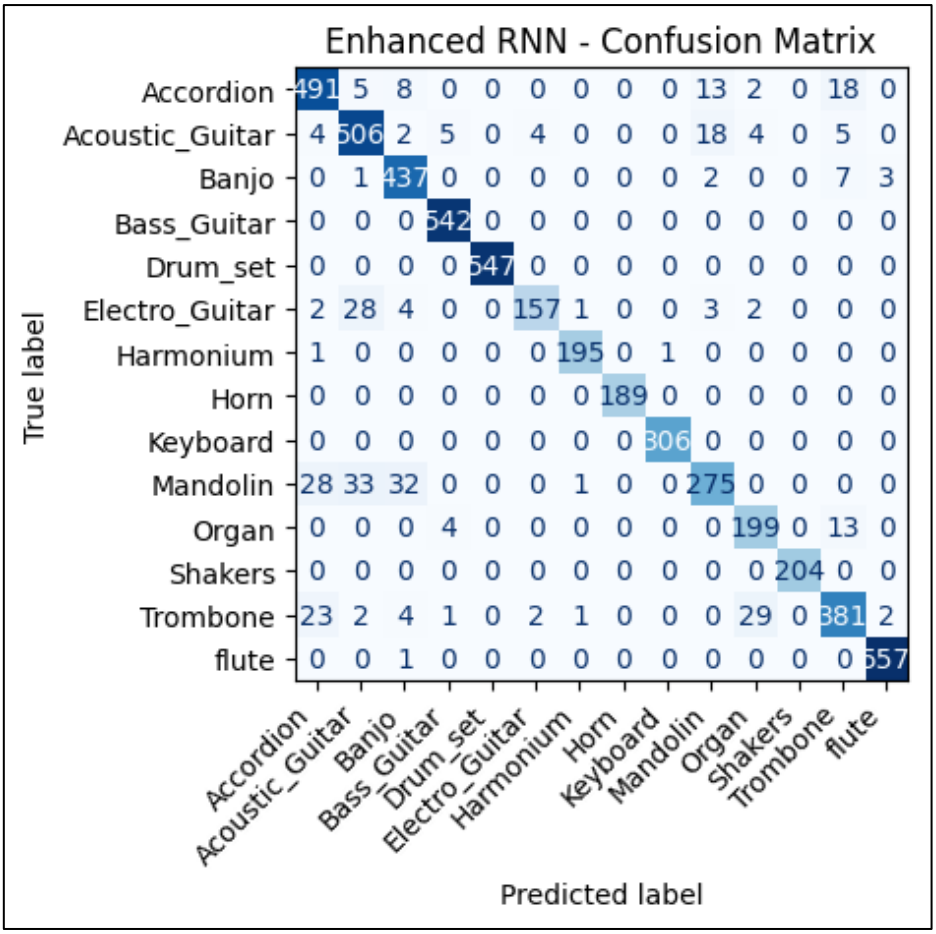

Figure 99: Enhanced RNN-BiGRU Confusion Matrix

The enhanced RNN-BiGRU shows good performance, but it cannot be said to be either better or weaker than the baseline. Some classes, like accordion and banjo, are classified slightly better, with fewer misclassifications compared to the baseline (451 to 491 and 432 to 437, respectively). Mandolin also gains some stability, though it still shows confusion with similar string instruments. Trombone, which is already classified strongly in the baseline model, performs worse here, from 419 correct to only 381, with more errors against the organ and the accordion. Stable classes such as drum set, bass guitar, harmonium, horn, keyboard, shakers, and flute remain consistently strong across both models.

Classification report:

	precision	recall	f1-score	support
Accordion	0.894	0.914	0.904	537
Acoustic_Guitar	0.880	0.923	0.901	548
Banjo	0.895	0.971	0.932	450
Bass_Guitar	0.982	1.000	0.991	542
Drum_set	1.000	1.000	1.000	547
Electro_Guitar	0.963	0.797	0.872	197
Harmonium	0.985	0.990	0.987	197
Horn	1.000	1.000	1.000	189
Keyboard	0.997	1.000	0.998	306
Mandolin	0.884	0.745	0.809	369
Organ	0.843	0.921	0.881	216
Shakers	1.000	1.000	1.000	204
Trombone	0.899	0.856	0.877	445
flute	0.991	0.998	0.995	558
accuracy			0.940	5305
macro avg	0.944	0.937	0.939	5305
weighted avg	0.940	0.940	0.939	5305

Figure 100: Enhanced RNN-BiGRU Classification Report

The enhanced RNN-BiGRU and the baseline RNN have almost the same overall result, with both about 94% accuracy and macro-F1 of 0.939 and 0.942, respectively, but they trade wins by class. There are clear gains for accordion with F1 0.904 compared to 0.885, acoustic guitar with 0.901 from 0.869, mandolin with 0.809 from 0.783, and keyboard with 0.998 from 0.992. Electro guitar also improves in F1 to 0.872 from 0.840, driven by higher precision, though its recall dips to 0.797 from 0.827. Several classes remain effectively unchanged and strong, including bass guitar, drum set, horn, shakers, and Flute. The model is weaker than the baseline for banjo from 0.949 to 0.932, Trombone from 0.914 to 0.877, and Organ from 0.986 to 0.881. This situation shows that the temporal emphasis of the enhanced RNN helps some timbres but can hurt others where the baseline already captured stable cues. Overall, the enhanced RNN-BiGRU performance is not guaranteed to be drastically better than the baseline. In contrast, it is similar to the baseline.

4.2.3 Hyperparameter Tuning RNN-BiGRU

```
import os, json, itertools, gc, hashlib, time, numpy as np, tensorflow as tf
from tensorflow.keras import Sequential, layers, optimizers, regularizers, callbacks, backend as K
from sklearn.metrics import classification_report, confusion_matrix

# Reproducibility for fix seeds for TF and NumPy
def set_seed(seed=42):
    tf.random.set_seed(seed)
    np.random.seed(seed)

set_seed(42)

# Directory for all RNN grid runs, logs and checkpoints
RUNS_DIR = "../output/model/rnn_grid_runs"
os.makedirs(RUNS_DIR, exist_ok=True)

# Reduce TensorFlow console verbosity
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
```

Figure 101: Setup for RNN-BiGRU Hyperparameter Tuning

After training the baseline and enhanced RNN-BiGRU models, but still getting the same result, the process moves to hyperparameter tuning to test different model configurations and find the best-performing setup. The tuning workflow for RNN-BiGRU follows the same structure as the CNN tuning done earlier. It begins by fixing random seeds for both TensorFlow and NumPy so results are reproducible, then sets up a dedicated directory to store all RNN grid search runs, logs, and checkpoint files. This ensures that every trial is tracked and results can be compared later. Finally, the TensorFlow log level is reduced to keep the output clean and focused only on the important progress messages.

```python
import tensorflow as tf
from tensorflow.keras import Sequential, layers, optimizers

def build_model(input_shape, num_classes, p):
    u1, u2 = p["gru_units"]
    dp = float(p["dropout"])

    model = Sequential(name="base_model_rnn_tuned")
    model.add(layers.Input(shape=input_shape))                  # (mel, time, 1)

    model.add(layers.Permute((2, 1, 3), name="permute"))      # -> (time, mel, 1)
    model.add(layers.Lambda(lambda x: tf.squeeze(x, axis=-1), name="squeeze"))  # -> (time, mel)

    # Block 1: BiGRU with dropout, return full sequence for stacking
    model.add(layers.Bidirectional(
        layers.GRU(u1, dropout=dp, return_sequences=True),
        name=f"bigru_{u1}"
    ))

    # Block 2: BiGRU with dropout, last output only for classification head
    model.add(layers.Bidirectional(
        layers.GRU(u2, dropout=dp, return_sequences=False),
        name=f"bigru2_{u2}"
    ))

    # Optional small dense head to increase capacity
    if p["use_dense_head"]:
        model.add(layers.Dense(32, activation="relu", name="head_dense_32"))

    # Classifier
    model.add(layers.Dense(num_classes, activation="softmax", name="Classifier"))

    # Optimizer configured per trial
    opt = optimizers.Adam(learning_rate=p["learning_rate"])
    model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model
```

Figure 102: RNN-BiGRU Model Factory

The model factory function for tuning the BiGRU network is then created. Instead of hard-coding sizes, it takes a parameter *dict p* and builds a BiGRU classifier accordingly. The first two lines pull the two GRU widths *(u1, u2)* and a shared dropout rate *(dp)*. The input log-Mel tensor *(mel, time, 1)* is permuted to *(time, mel)*, which is a must so the RNN reads a sequence of Mel frames. Then, two BiGRU blocks are stacked, the first returns the full sequence, while the second emits only the final summary vector for classification. An optional tiny dense head *(use_dense_head)* can be toggled. The last layer is a softmax over the instrument classes, and each trial uses Adam with a tunable learning rate. This design is a compromise because earlier results showed the baseline and enhanced RNNs are close in accuracy, so the search focuses on just the influential knobs, like GRU units, dropout, optional head, and learning rate, rather than exploding model complexity.

```
# 3 * 2 * 1 * 2 * 2 = 24 trials
param_grid = {
    "gru_units": [(32, 16), (32, 32), (64, 32)],  # units for (GRU1, GRU2)
    "dropout": [0.0, 0.3],                        # per-GRU dropout
    "use_dense_head": [False, True],              # optional small dense before classifier
    "learning_rate": [3e-4, 1e-3],                # Adam LR options
    "epochs": [30],                               # fixed budget per trial
    "batch_size": [32],                           # stable memory usage
}
```

Figure 103: RNN-BiGRU Hyperparameter Grid

Since baseline and enhanced RNNs already reach high accuracy, and RNN training is slower due to more parameters and sequential ops, a smaller grid is used compared to CNN without exploding runtime, and if needed, a second pass can also refine around the best region.

- *gru_units: (32,16)* can keep the second layer lighter, which encourages the first layer to capture most structure, while *(32,32)* balances capacity across layers as a common sweet spot for short clips, and *(64,32)* tests a higher-capacity first layer.
- *dropout: 0.0* measures the unregularized ceiling and ensures dropout is not masking useful signals, while *0.3* gives a strong but typical level of regularization for sequence models to see if it will improve generalization.
- *use_dense_head: False* keeps the model lean, which relies entirely on the BiGRU summary, just like baseline, while *True* adds a light mixing layer that can separate borderline classes like in enhanced RNN.
- *learning_rate: 3e-4* is a conservative and stable choice, while *1e-3* accelerates early learning
- *epochs:* Epoch is fixed to ensure a fair budget across trials. With early stopping scheduling, 30 epochs are enough in this classification model to separate strong from weak configs without wasting compute.
- *batch_size:* It is also fixed because it holds memory use and gradient noise constant.

Overall, the baseline and enhanced models already perform well, and RNN trials are slower than CNN trials. The targeted grid with 24 trials should be able to find the best configuration.

```
def dict_product(d):
    # Cartesian product over parameter lists to sequence of dicts
    keys = list(d.keys())
    for values in itertools.product(*[d[k] for k in keys]):
        yield dict(zip(keys, values))

def short_id_from_params(p):
    # Short, stable ID from params (good for folder names)
    key_str = json.dumps(p, sort_keys=True)
    return hashlib.md5(key_str.encode()).hexdigest()[:8]
```

Figure 104: Generate Parameter Sets and Unique Trial IDs

Then, the functions defined work the same as in CNN tuning, with *dict_product* generating every parameter combination from the grid, and *short_id_from_params* creates a short and unique ID for each trial to label its log and checkpoint files.

```python
def train_one_trial(p, input_shape, num_classes, train_ds, val_ds):
    run_id = short_id_from_params(p)
    run_dir = os.path.join(RUNS_DIR, run_id)
    os.makedirs(run_dir, exist_ok=True)

    # File paths per trial
    csv_path = os.path.join(run_dir, "history.csv")
    ckpt_path = os.path.join(run_dir, "best.weights.h5")

    # Callbacks: CSV log, best checkpoint (val_accuracy), early stopping
    cbs = [
        callbacks.CSVLogger(csv_path, append=False),
        callbacks.ModelCheckpoint(
            ckpt_path,
            save_weights_only=True,
            monitor="val_accuracy",
            mode="max",
            save_best_only=True,
            verbose=0
        ),
        callbacks.EarlyStopping(
            monitor="val_accuracy",
            mode="max",
            patience=6,
            restore_best_weights=True,
            verbose=0
        ),
    ]

    # Build and train the model for this param set
    model = build_model(input_shape, num_classes, p)
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=p["epochs"],
        batch_size=p["batch_size"],
        verbose=0,
        callbacks=cbs
    )

    # Ensure best weights are loaded before evaluation
    if os.path.exists(ckpt_path):
        model.load_weights(ckpt_path)

    # Validation metrics for selection
    val_loss, val_acc = model.evaluate(val_ds, verbose=0)

    # Persist a compact summary for later aggregation
    with open(os.path.join(run_dir, "summary.json"), "w") as f:
        json.dump({"params": p, "val_loss": float(val_loss), "val_accuracy": float(val_acc)}, f, indent=2)

    print(f"[{run_id}] val_acc={val_acc:.4f} | params={p}")
    return run_id, val_acc, val_loss, run_dir, model
```

Figure 105: Train, Evaluate, and Log One RNN-BiGRU Trial Function

The function for running one RNN-BiGRU tuning trial is also initiated, following the same logic as the CNN tuning process as well. It builds the model with the given parameters, trains it using callbacks for early stopping, best-weight checkpointing, and CSV logging, then reloads the best weights to evaluate validation accuracy and loss. The results and parameters are saved.

```
def count_trials(pg):
    # Multiply lengths of all value lists to get Cartesian product size
    total = 1
    for k, vals in pg.items():
        total *= len(vals)
    return total

TOTAL_TRIALS = count_trials(param_grid)
print(f"Total Trials: {TOTAL_TRIALS}")
```

```
Total Trials: 24
```

Figure 106: Count Total Trials

Figure 106 confirms that the RNN-BiGRU tuning grid will generate a total of 24 trials for evaluation.

```
import sys, json, os

# Path to CSV for trial summaries
LIVE_CSV = os.path.join(RUNS_DIR, "grid_summary_live.csv")

# Initialize the CSV with a header
if not os.path.exists(LIVE_CSV):
    with open(LIVE_CSV, "w") as f:
        f.write("idx,run_id,val_accuracy,val_loss,params_json\n")
        f.flush(); os.fsync(f.fileno())
```

Figure 107: Prepare Live Summary CSV for Grid Runs

Next, a live summary CSV file is created to record each RNN tuning trial's ID, validation accuracy, loss, and parameters, to ensure all results are logged and easy to track during the grid search.

```
# Input shape for log-Mel tensors and containers for results
input_shape = (num_mel_bins, None, 1)
results = []
best = {"val_acc": -1, "run_id": None, "run_dir": None, "params": None}
start_all = time.time()

# Iterate over every parameter combination
for i, p in enumerate(dict_product(param_grid), start=1):
    K.clear_session(); gc.collect(); set_seed(42)  # reset graph and seeds for fair trials
    t0 = time.time()
    model = None
    try:
        # Train a single trial and collect metrics
        run_id, val_acc, val_loss, run_dir, model = train_one_trial(
            p, input_shape, num_classes, train_ds, val_ds
        )
        results.append({"run_id": run_id, "val_accuracy": float(val_acc), "val_loss": float(val_loss), "params": p})

        # Track best model so far and persist weights immediately
        if val_acc > best["val_acc"]:
            best.update({"val_acc": val_acc, "run_id": run_id, "run_dir": run_dir, "params": p})
            best_weights_path = os.path.join(RUNS_DIR, "best_overall.weights.h5")
            model.save_weights(best_weights_path)

        # Concise and single-line progress output
        p_show = {
            "gru": p["gru_units"],
            "drop": p["dropout"],
            "dense_head": p["use_dense_head"],
            "lr": p["learning_rate"],
            "bs": p["batch_size"],
            "ep": p["epochs"],
        }
        elapsed = time.time() - t0
        line = f"[{i}/{TOTAL_TRIALS}] {run_id} val_acc={val_acc:.4f} val_loss={val_loss:.4f} | {p_show} | {elapsed:.1f}s"
        print(line); sys.stdout.flush()  # force print in notebook

        # Append to progress.log (text) with fsync
        with open(os.path.join(RUNS_DIR, "progress.log"), "a") as f:
            f.write(line + "\n"); f.flush(); os.fsync(f.fileno())

        # Append to live CSV with fsync (params JSON with commas replaced)
        with open(LIVE_CSV, "a") as f:
            params_json = json.dumps(p, sort_keys=True).replace(",", ";")
            f.write(f"{i},{run_id},{val_acc:.6f},{val_loss:.6f},{params_json}\n")
            f.flush(); os.fsync(f.fileno())

    finally:
        # Clean up model and free graph memory between trials
        if model is not None:
            del model
        K.clear_session(); gc.collect()

# Total wall-clock time for the entire grid
total_time = time.time() - start_all
```

Figure 108: Run RNN-BiGRU Grid Search

The RNN-BiGRU tuning starts by looping through all 24 parameter combinations in the grid, training each configuration one by one, which has a similar flow to CNN hyperparameter tuning. Each model is trained and evaluated on validation data, with its accuracy, loss, and parameters recorded. Whenever a model achieves a higher validation accuracy than previous ones, its weight is saved as the current best model. Throughout the process, the progress is printed and written to a live CSV. After each trial, the total runtime is recorded once all trials finish. This process ensures every RNN configuration is tested reliably and that the best-performing one is safely stored.

```
[73705c0e] val_acc=0.9947 | params={'gru_units': (32, 16), 'dropout': 0.0, 'use_dense_head': False, 'learning_rate': 0.0003, 'epochs': 30, 'batch_size': 32}
[1/24] 73705c0e val_acc=0.9947 val_loss=0.0248 | {'gru': (32, 16), 'drop': 0.0, 'dense_head': False, 'lr': 0.0003, 'bs': 32, 'ep': 30} | 1976.8s
[57e4c8cf] val_acc=0.9925 | params={'gru_units': (32, 16), 'dropout': 0.0, 'use_dense_head': False, 'learning_rate': 0.001, 'epochs': 30, 'batch_size': 32}
[2/24] 57e4c8cf val_acc=0.9925 val_loss=0.0273 | {'gru': (32, 16), 'drop': 0.0, 'dense_head': False, 'lr': 0.001, 'bs': 32, 'ep': 30} | 1175.2s
[1fa5da5b] val_acc=0.9949 | params={'gru_units': (32, 16), 'dropout': 0.0, 'use_dense_head': True, 'learning_rate': 0.0003, 'epochs': 30, 'batch_size': 32}
[3/24] 1fa5da5b val_acc=0.9949 val_loss=0.0206 | {'gru': (32, 16), 'drop': 0.0, 'dense_head': True, 'lr': 0.0003, 'bs': 32, 'ep': 30} | 1657.6s
[c09fb670] val_acc=0.9926 | params={'gru_units': (32, 16), 'dropout': 0.0, 'use_dense_head': True, 'learning_rate': 0.001, 'epochs': 30, 'batch_size': 32}
[4/24] c09fb670 val_acc=0.9926 val_loss=0.0275 | {'gru': (32, 16), 'drop': 0.0, 'dense_head': True, 'lr': 0.001, 'bs': 32, 'ep': 30} | 1283.3s
[a7eed3b7] val_acc=0.9793 | params={'gru_units': (32, 16), 'dropout': 0.3, 'use_dense_head': False, 'learning_rate': 0.0003, 'epochs': 30, 'batch_size': 32}
[5/24] a7eed3b7 val_acc=0.9793 val_loss=0.0696 | {'gru': (32, 16), 'drop': 0.3, 'dense_head': False, 'lr': 0.0003, 'bs': 32, 'ep': 30} | 2244.5s
```

Figure 109: RNN-BiGRU First 5 Tuning Trials

```
[fd7f77f6] val_acc=0.9951 | params={'gru_units': (64, 32), 'dropout': 0.0, 'use_dense_head': True, 'learning_rate': 0.001, 'epochs': 30, 'batch_size': 32}
[20/24] fd7f77f6 val_acc=0.9951 val_loss=0.0239 | {'gru': (64, 32), 'drop': 0.0, 'dense_head': True, 'lr': 0.001, 'bs': 32, 'ep': 30} | 1053.4s
[def358b6] val_acc=0.9887 | params={'gru_units': (64, 32), 'dropout': 0.3, 'use_dense_head': False, 'learning_rate': 0.0003, 'epochs': 30, 'batch_size': 32}
[21/24] def358b6 val_acc=0.9887 val_loss=0.0410 | {'gru': (64, 32), 'drop': 0.3, 'dense_head': False, 'lr': 0.0003, 'bs': 32, 'ep': 30} | 1742.9s
[9b9aa482] val_acc=0.9885 | params={'gru_units': (64, 32), 'dropout': 0.3, 'use_dense_head': False, 'learning_rate': 0.001, 'epochs': 30, 'batch_size': 32}
[22/24] 9b9aa482 val_acc=0.9885 val_loss=0.0412 | {'gru': (64, 32), 'drop': 0.3, 'dense_head': False, 'lr': 0.001, 'bs': 32, 'ep': 30} | 1174.3s
[a4a18802] val_acc=0.9862 | params={'gru_units': (64, 32), 'dropout': 0.3, 'use_dense_head': True, 'learning_rate': 0.0003, 'epochs': 30, 'batch_size': 32}
[23/24] a4a18802 val_acc=0.9862 val_loss=0.0508 | {'gru': (64, 32), 'drop': 0.3, 'dense_head': True, 'lr': 0.0003, 'bs': 32, 'ep': 30} | 1596.2s
[cba05186] val_acc=0.9885 | params={'gru_units': (64, 32), 'dropout': 0.3, 'use_dense_head': True, 'learning_rate': 0.001, 'epochs': 30, 'batch_size': 32}
[24/24] cba05186 val_acc=0.9885 val_loss=0.0401 | {'gru': (64, 32), 'drop': 0.3, 'dense_head': True, 'lr': 0.001, 'bs': 32, 'ep': 30} | 1487.0s
```

Figure 110: RNN-BiGRU Last 5 Tuning Trials

The hyperparameter tuning process for the RNN-BiGRU model runs successfully as designed. All 24 trials are executed without error, and the printed logs show that each configuration is trained, validated, and logged properly, with validation accuracy and loss also reported.

```python
import pandas as pd

# Read results CSV and sort descending by validation accuracy
df_results = pd.read_csv(LIVE_CSV)
df_results = df_results.sort_values("val_accuracy", ascending=False)

# Display options for a concise leaderboard
pd.set_option("display.max_rows", 12)
pd.set_option("display.max_colwidth", 60)

print("Top 10 Trials by Val Accuracy:")
display(df_results.head(10)[["idx","run_id","val_accuracy","val_loss"]].round(4))

print(f"Trials: {len(df_results)} | Elapsed: {total_time/60:.1f} min")
```

```
Top 10 Trials by Val Accuracy:
```

	idx	run_id	val_accuracy	val_loss
16	17	24c344e8	0.9959	0.0185
18	19	5cdb1571	0.9957	0.0154
19	20	fd7f77f6	0.9951	0.0239
2	3	1fa5da5b	0.9949	0.0206
10	11	c055b7cc	0.9949	0.0193
0	1	73705c0e	0.9947	0.0248
17	18	6d0e489d	0.9945	0.0261
8	9	8d4b5437	0.9936	0.0245
9	10	9d2e366f	0.9936	0.0264
11	12	09bc0f7e	0.9928	0.0274

```
Trials: 24 | Elapsed: 584.2 min
```

Figure 111: RNN-BiGRU Hyperparameter Tuning Summary

The best model is on the 17th trial, with run ID 24c344e8, achieved a validation accuracy of 99.59% and a very low validation loss of 0.0185. Several other trials also achieved similarly high accuracy above 99%, which shows that the model can perform well across various configurations. Overall, the tuning process is effective, taking about 584 minutes to complete. Also, this proves that the RNN architecture is capable of this classification task.

```
best = df_results.iloc[0]
params_str = str(best["params_json"]).replace(";", ",")
best_params = json.loads(params_str)

print("\nBest hyperparameters:")
print(json.dumps(best_params, indent=2, sort_keys=True))
```

```
Best hyperparameters:
{
  "batch_size": 32,
  "dropout": 0.0,
  "epochs": 30,
  "gru_units": [
    64,
    32
  ],
  "learning_rate": 0.0003,
  "use_dense_head": false
}
```

Figure 112: Print RNN-BiGRU Best Hyperparameter

The best hyperparameter combination shows that the optimal RNN-BiGRU model performs best with 64 and 32 GRU units for its two layers, no need for dropout, as well as no dense head before the final classifier. It also uses a learning rate of 0.0003, batch size of 32, and runs for 30 epochs. This setup means that the model benefits from keeping all neurons active, like what is done in the baseline model. The higher GRU units give it enough capacity to capture complex time-based features. Overall, this configuration achieves the best balance between accuracy and generalization for the dataset.

```python
# Reload best model, plot curves and evaluate
import os, json, pandas as pd, numpy as np
import tensorflow as tf
from tensorflow.keras import backend as K

# Recover best params/run_id even if the session restarted
def _recover_best(RUNS_DIR, best_dict=None):
    # Read leaderboard CSV (prefer final, fallback to live)
    grid_csv = os.path.join(RUNS_DIR, "grid_summary.csv")
    if not os.path.exists(grid_csv):
        grid_csv = os.path.join(RUNS_DIR, "grid_summary_live.csv")

    df = pd.read_csv(grid_csv)
    top = df.sort_values("val_accuracy", ascending=False).iloc[0]
    run_id = top["run_id"]

    # Params stored in the trial's summary.json
    with open(os.path.join(RUNS_DIR, run_id, "summary.json"), "r") as f:
        params = json.load(f)["params"]
    return run_id, params

best_run_id, best_params = _recover_best(RUNS_DIR, best)

# Build and load weights for the best model
best_model = build_model(input_shape, num_classes, best_params)
best_weights = os.path.join(RUNS_DIR, "best_overall.weights.h5")
best_model.load_weights(best_weights)
```

Figure 113: Reload Best Tuned RNN-BiGRU and Restore Weights

Next, the top-performing RNN-BiGRU trial is reloaded and its weights are restored for evaluation. It will read the tuning leaderboard CSV, then pick the row with the highest validation accuracy, and grab that run's parameters from its *summary.json*. With those parameters, it rebuilds the exact model architecture, then loads the saved best_*overall.weights.h5* checkpoint, so the model is ready to plot training curves and test on the testing dataset using the best learned weights.

```
best_model.summary()
```

```
Model: "base_model_rnn_tuned"
```

Layer (type)	Output Shape	Param #
permute (Permute)	(None, None, 64, 1)	0
squeeze (Lambda)	(None, None, 64)	0
bigru_64 (Bidirectional)	(None, None, 128)	49,920
bigru2_32 (Bidirectional)	(None, 64)	31,104
Classifier (Dense)	(None, 14)	910

```
Total params: 81,934 (320.05 KB)

Trainable params: 81,934 (320.05 KB)

Non-trainable params: 0 (0.00 B)
```

Figure 114: Best Tuned RNN-BiGRU Model Summary

The best-tuned RNN-BiGRU model is a simple and effective architecture that is almost the same as the baseline RNN-BiGRU, but with optimized parameters that enhance learning stability and accuracy. Same, the model starts by permuting the input shape so that the time dimension comes first to allow the GRU layers to read the mel-spectrograms as temporal sequences. The squeeze layer then removes the redundant channel dimension to simplify the input.

The first BiGRU layer uses 64 units, and because it is bidirectional, it learns both forward and backward temporal patterns, with 128 outputs per timestep. This helps the network capture both past and future context within the sound sequence. The second BiGRU layer with 32 units refines these features and outputs a 64-dimensional feature vector that summarizes the whole sequence. The final dense classifier layer with 14 outputs predicts the instrument class probabilities using softmax activation.

The best-tuned model has 81,934 trainable parameters, which is the largest model ever compared to the previous models. Overall, this design is similar to the baseline RNN-BiGRU structure. However, the tuned configuration, like the combination of 64 and 32 GRU units, gives a good balance between model complexity and training efficiency.

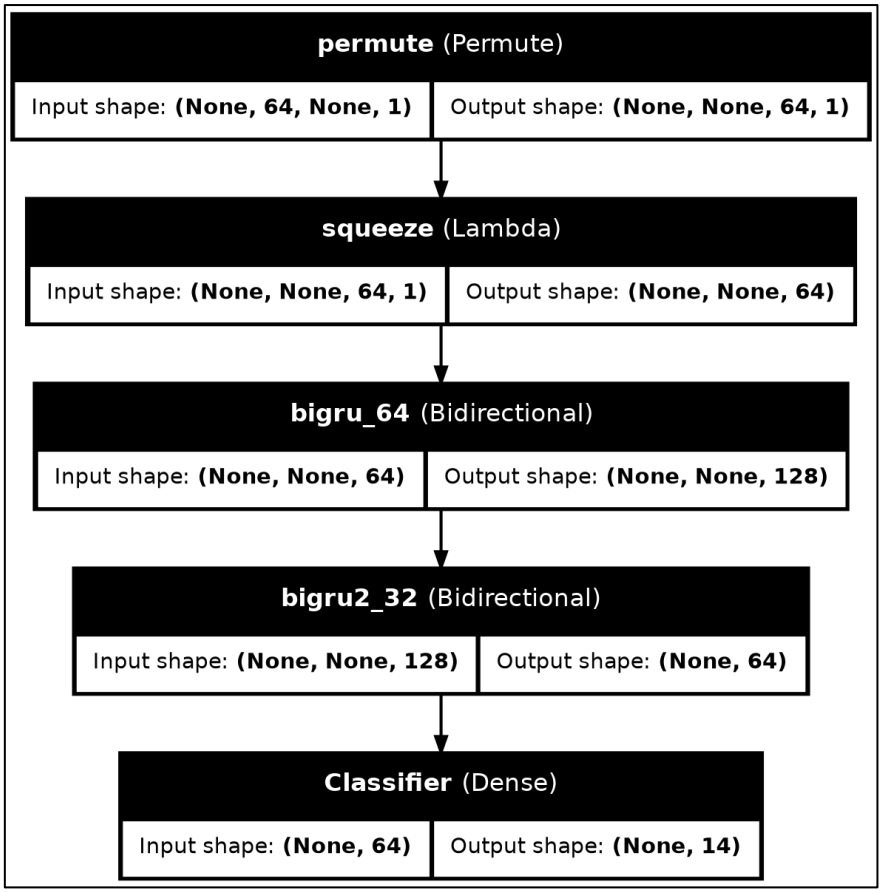

Figure 115: Best Tuned RNN-BiGRU Model Architecture

Figure 115 shows the architecture of the best-tuned RNN-BiGRU model.

```
# Recreate a History-like object from the best trial's history.csv for plotting
hist_path = os.path.join(RUNS_DIR, best_run_id, "history.csv")
if os.path.exists(hist_path):
    hist_df = pd.read_csv(hist_path)
    # Keras History-like shim
    class _SimpleHistory:
        pass
    history_for_plot = _SimpleHistory()
    # Ensure lists (drop epoch column if present)
    hist_cols = [c for c in hist_df.columns if c != "epoch"]
    history_for_plot.history = {c: hist_df[c].tolist() for c in hist_cols}

# Plot training curves
plot_training_curves(
    history_for_plot,
    metrics=("loss", "accuracy"),
    suptitle=f"Best Tuned RNN (run on id: {best_run_id})"
)
```

Figure 116: Recreate History for Best Tuned RNN-BiGRU

After that, the best RNN trial's training log is reloaded so it can be plotted with the same helper used earlier. The *plot_training_curves* function will visualize how the tuned RNN is trained over epochs, just like what is done in the CNN tuning step.

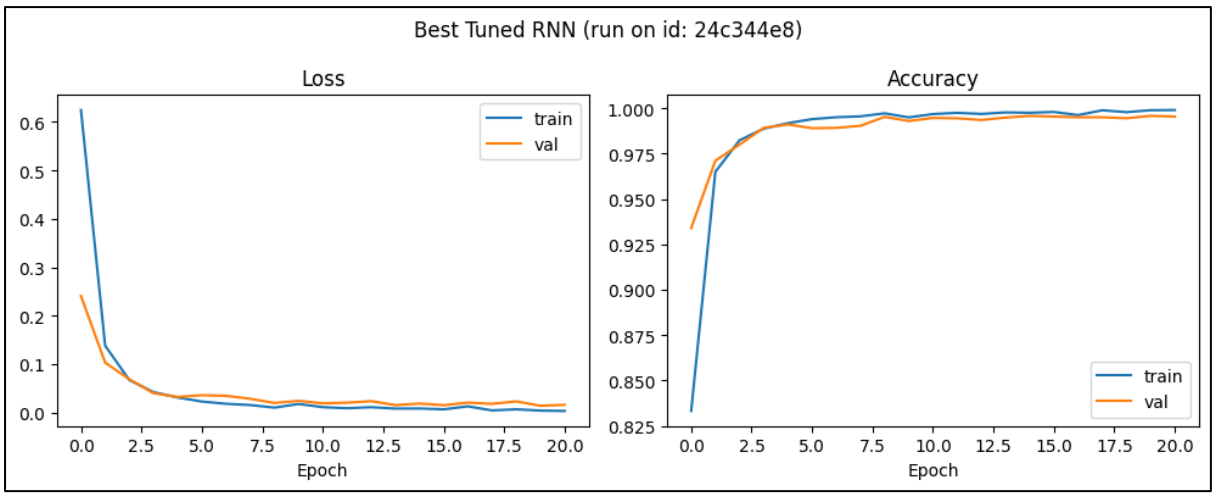

Figure 117: Best Tuned RNN-BiGRU Training Curves

Figure 117 shows the training and validation performance of the best-tuned RNN-BiGRU model. The loss curve drops sharply within the first few epochs and stays near zero, while both training and validation accuracy rise above 98% and stabilize close to 100%. The close overlap between the two lines means the model has no overfitting.

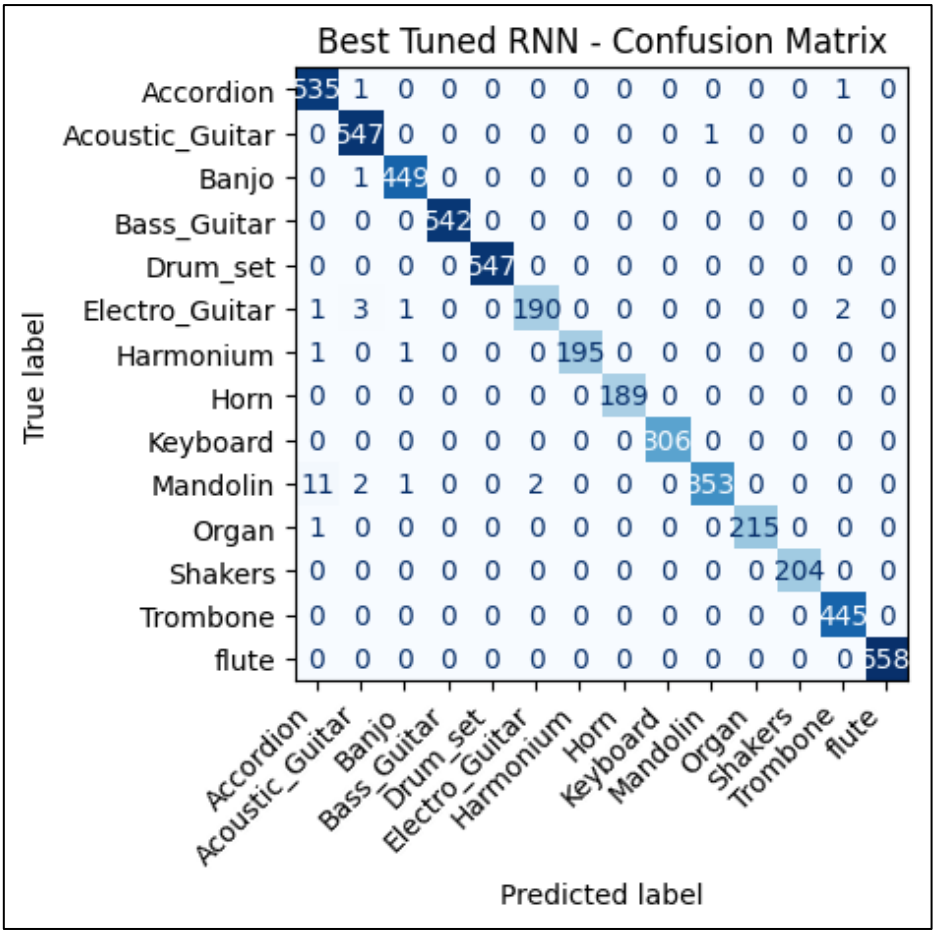

Figure 118: Best Tuned RNN-BiGRU Confusion Matrix

The best-tuned RNN-BiGRU model performs very well across all instrument classes, with most predictions correctly classified along the diagonal line. Compared to both the baseline and enhanced RNN models, this tuned version has a significant improvement in overall accuracy and consistency. Nearly all instruments, such as accordion, acoustic guitar, drum set, bass guitar, and flute, are classified with no or few errors only. The only very minor weakness is with the mandolin class, where 16 samples are misclassified, but overall, the model achieves good recognition across the test dataset.

Classification report:

	precision	recall	f1-score	support
Accordion	0.974	0.996	0.985	537
Acoustic_Guitar	0.987	0.998	0.993	548
Banjo	0.993	0.998	0.996	450
Bass_Guitar	1.000	1.000	1.000	542
Drum_set	1.000	1.000	1.000	547
Electro_Guitar	0.990	0.964	0.977	197
Harmonium	1.000	0.990	0.995	197
Horn	1.000	1.000	1.000	189
Keyboard	1.000	1.000	1.000	306
Mandolin	0.997	0.957	0.976	369
Organ	1.000	0.995	0.998	216
Shakers	1.000	1.000	1.000	204
Trombone	0.993	1.000	0.997	445
flute	1.000	1.000	1.000	558
accuracy			0.994	5305
macro avg	0.995	0.993	0.994	5305
weighted avg	0.994	0.994	0.994	5305

Figure 119: Best Tuned RNN-BiGRU Classification Report

The classification report shows that the best-tuned RNN-BiGRU has an overall accuracy of 99.4% with macro-F1 of 0.994 and weighted-F1 of 0.994, which shows both strong balance and reliability across classes. Most instruments are essentially perfect, including bass guitar, drum set, horn, keyboard, shakers, and flute, with a 1.00 F1 score. Difficult classes identified before also perform well, like accordion reaches 0.985 F1, banjo 0.996 F1, trombone 0.997 F1, mandolin 0.976 F1, and electro guitar improves to 0.977 F1 with 0.964 recall. Overall, the tuned RNN generalizes well and better than the earlier RNN variants.

4.2.4 Summary

In summary, the RNN-BiGRU modelling shows the progression of an increase in performance and stability across three stages. The baseline model already achieved a high accuracy of around 94%. The enhanced RNN added larger hidden units, dropout, and the Adam optimizer, though its performance is similar to the baseline. Finally, the tuned RNN-BiGRU achieved a big improvement, reaching 99.4% accuracy after optimizing key parameters such as GRU units, dropout, and learning rate. It has near-perfect classification across most instruments. Overall, this tuning phase improves the model's generalization and allows the RNN-BiGRU to perform better than earlier versions by a large margin.

5 Model Comparison

The best model of both proposed algorithms is chosen for comparison.

Aspect	Best Tuned CNN	Best Tuned RNN-BiGRU
Input representation	Log-Mel spectrograms, shape (64, T, 1)	Log-Mel spectrograms, shape (64, T, 1), but permuted to (T, 64)
Key configuration	Conv filters (24, 48, 96), Dense 128, dropout in head	GRU units (64, 32), no dense head
Optimizer / LR	Adam, 3e-4	Adam, 3e-4
Total parameters	67,118	81,934
Trainable / non-trainable	66,782 / 336	81,934 / 0
Regularization	BatchNorm + Dropout (head)	None
Test accuracy	0.996	0.994
Macro avg (P / R / F1)	0.997 / 0.996 / 0.996	0.995 / 0.993 / 0.994
Weighted avg (P / R / F1)	0.996 / 0.996 / 0.996	0.994 / 0.994 / 0.994

Table 2: Comparison Table for CNN and RNN-BiGRU

As a result, both the best-tuned CNN and RNN-BiGRU models show strong results, with only small differences in accuracy. The CNN model achieved slightly higher overall performance at 99.6% accuracy, compared to 99.4% for the RNN-BiGRU. It is also lighter, with about 18% fewer parameters, and includes batch normalization and dropout. The RNN-BiGRU is slightly heavier, also performs well, which shows that sequential GRU layers can also effectively capture time-based patterns in the audio. Overall, both models are reliable.

Class	CNN (P/R/F1)	RNN-BiGRU (P/R/F1)	F1 Difference (CNN – RNN-BiGRU)
Accordion	0.994 / 0.987 / 0.991	0.974 / 0.996 / 0.985	+0.006
Acoustic Guitar	0.993 / 0.998 / 0.995	0.987 / 0.998 / 0.993	+0.002
Banjo	1.000 / 0.996 / 0.998	0.993 / 0.998 / 0.996	+0.002
Bass Guitar	0.998 / 1.000 / 0.999	1.000 / 1.000 / 1.000	-0.001
Drum Set	1.000 / 1.000 / 1.000	1.000 / 1.000 / 1.000	0
Electro Guitar	0.995 / 0.975 / 0.985	0.990 / 0.964 / 0.977	+0.008
Harmonium	1.000 / 1.000 / 1.000	1.000 / 0.990 / 0.995	+0.005
Horn	1.000 / 1.000 / 1.000	1.000 / 1.000 / 1.000	0
Keyboard	1.000 / 1.000 / 1.000	1.000 / 1.000 / 1.000	0
Mandolin	1.000 / 0.992 / 0.996	0.997 / 0.957 / 0.976	+0.020
Organ	0.995 / 1.000 / 0.998	1.000 / 0.995 / 0.998	0
Shakers	1.000 / 1.000 / 1.000	1.000 / 1.000 / 1.000	0
Trombone	0.978 / 0.998 / 0.988	0.993 / 1.000 / 0.997	-0.009
Flute	1.000 / 0.998 / 0.999	1.000 / 1.000 / 1.000	-0.001

Table 3: Comparison by Class Performance

Both the CNN and RNN-BiGRU perform well across all instruments, with F1-scores consistently above 0.97. The CNN performs slightly better in most classes, such as accordion, acoustic guitar, electro guitar, and especially mandolin, which shows the largest difference of 0.020. The RNN-BiGRU also has a better side, very slightly better than the CNN for trombone, bass guitar, and flute, where temporal cues might be an easier pattern extraction. Several instruments, like drum set, keyboard, horn, and shakers, achieve perfect scores on both models. Overall, both models are highly reliable, and the CNN is slightly stronger across the wider range of instrument types.

Author	Dataset Used	Model Type	Features / Input	Accuracy / F1
Sauyang (this study)	Kaggle Musical Instrument Sounds for Classification (14 classes)	CNN	Log-Mel Spectrogram	Accuracy: 99.6% Macro F1: 0.996
Sauyang (this study)	Same as above	BiGRU	Log-Mel Spectrogram sequence	Accuracy: 99.4% Macro F1: 0.994
Chen et al. (2025)	Same as above, but 28 classes	Custom CNN with handcrafted rhythm-motif features	Music-inspired handcrafted features + CNN	Accuracy: 98.76%
Giri and Radhitya (2024)	PPMI dataset (12 instruments)	CNN	MFCC features	Accuracy: 71.8% F1: 0.68
Su (2023)	Philharmonia (7 instruments)	CNN, RNN, SVM	Spectral + temporal features	Accuracy CNN: 96.82% RNN: 94.83%
Ashraf et al. (2023)	GTZAN (10 genres)	CNN + BiGRU Hybrid	Mel-spectrogram	Accuracy: 89.3% F1: 0.88

Table 4: Comparison to Previous Research

Compared to past research, both proposed models achieve good performance on the same Kaggle dataset, and are better than the earlier CNN-based works that peaked near 98%. The CNN shows very good generalization in music instrument recognition, while the BiGRU also captures sequential features effectively. However, if there is further research, this model should be trained on full classes and can be used to try on the other dataset, such as PPMI and Philharmonia, to see whether it can still get this good result or not.

6 Conclusion

In conclusion, this study builds two deep learning models, CNN and RNN-BiGRU, for musical instrument sound classification using the Kaggle Musical Instrument Sounds dataset. Both models perform well after hyperparameter tuning, and CNN achieves slightly higher accuracy at 99.6% compared to the RNN-BiGRU with 99.4%. CNN shows strong performance in learning spatial features from spectrograms, while the RNN-BiGRU can also learn temporal sound patterns well. For future improvement, the system can consider extending it to include all 28 instrument classes from the full dataset because this study only focuses on 14 classes due to computational resource limitations.

References

Ashraf, M., Abid, F., Din, I. U., Rasheed, J., Yesiltepe, M., Yeo, S. F., & Ersoy, M. T. (2023). A hybrid CNN and RNN variant model for music classification. *Applied Sciences*, *13*(3), 1476. https://doi.org/10.3390/app13031476

Ba, T. C., Le, T. D. T., & Van, L. T. (2025). Music genre classification using deep neural networks and data augmentation. *Entertainment Computing*, 100929. https://doi.org/10.1016/j.entcom.2025.100929

Bidirectional Gated Recurrent unit. (2024, August 5). Simple Science. https://scisimple.com/en/keywords/bidirectional-gated-recurrent-unit--kkg7mdd

Borovčak, K., & Babac, M. B. (2025). Instrument Classification in Musical Audio Signals using Deep Learning. *Croatian Regional Development Journal*, *6*(1), 84–99. https://doi.org/10.2478/crdj-2025-0006

Campos, J. (2025, April 6). *Unlocking the Power of Audio FFT: A Deep Dive into Audio Signal Processing*. beEmbedded. https://beembedded.com/audio-signal-processing-with-fft/

Chen, M., Tang, D., Xiang, Y., Shi, L., Tuncer, T., Ozyurt, F., & Dogan, S. (2025). Instrument sound classification using a music-based feature extraction model inspired by Mozart's Turkish March pattern. *Alexandria Engineering Journal*, *118*, 354–370. https://doi.org/10.1016/j.aej.2025.01.059

Choice of HOP size | Spectral Audio Signal Processing. (n.d.). https://www.dsprelated.com/freebooks/sasp/Choice_Hop_Size.html

Dorfler, M., Bammer, R., & Grill, T. (2017). Inside the spectrogram: Convolutional neural networks in audio processing. *International Conference on Sampling Theory and Applications*, 152–155. https://doi.org/10.1109/sampta.2017.8024472

Giri, G. a. V. M., & Radhitya, M. L. (2024). Musical Instrument Classification using Audio Features and Convolutional Neural Network. *Journal of Applied Informatics and Computing*, *8*(1), 226–234. https://doi.org/10.30871/jaic.v8i1.8058

Han, Y., Kim, J., & Lee, K. (2016). Deep convolutional neural networks for predominant instrument recognition in polyphonic music. *IEEE/ACM Transactions on Audio Speech and Language Processing*, *25*(1), 208–221. https://doi.org/10.1109/taslp.2016.2632307

Keary, T. (2025, May 21). *Recurrent Neural Network (RNN)*. Techopedia. https://www.techopedia.com/definition/recurrent-neural-network-rnn

Kumar, A., Gaur, N., Chakravarty, S., Alsharif, M. H., Uthansakul, P., & Uthansakul, M. (2023). Analysis of spectrum sensing using deep learning algorithms: CNNs and RNNs. *Ain Shams Engineering Journal*, *15*(3), 102505. https://doi.org/10.1016/j.asej.2023.102505

Kumar, A., & Yadav, J. (2024). Machine learning for audio processing: from feature extraction to model selection. In *Elsevier eBooks* (pp. 97–123). https://doi.org/10.1016/b978-0-443-22158-3.00005-3

Nandi, P. (2025, January 22). *Recurrent neural nets for audio classification*. Towards Data Science. https://towardsdatascience.com/recurrent-neural-nets-for-audio-classification-81cb62327990/

Niu, N. (2022). Music emotion Recognition model using gated recurrent unit networks and Multi-Feature extraction. *Mobile Information Systems*, *2022*, 1–11. https://doi.org/10.1155/2022/5732687

Park, T., & Lee, T. (2015). Musical instrument sound classification with deep convolutional neural network using feature fusion approach. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1512.07370

Sharma, K. (2025, April 1). *Gated Recurrent Unit (GRU) in deep learning explained*. Pickl.AI. https://www.pickl.ai/blog/gated-recurrent-unit-in-deep-learning/

Su, Y. (2023). Instrument classification using different machine learning and deep learning methods. *Highlights in Science Engineering and Technology*, *34*, 136–142. https://doi.org/10.54097/hset.v34i.5435

Vahap, A. (2024). Music instrument sounds for classification [Dataset]. In *Kaggle*. https://www.kaggle.com/datasets/abdulvahap/music-instrunment-sounds-for-classification

What is BiGRU explained? (2025, June 6). Ora. https://ora.it.com/sequence-processing-model/what-is-bigru-explained/

Wieclaw, M. (2025, May 14). *What is a CNN model in deep learning and how does it work?* AI Questions. https://aiquestions.co.uk/what-is-a-cnn-model-in-deep-learning-and-how-does-it-work/#What_is_a_CNN_Model