

DroneDetect V2 - Spectrogram Feature Extraction

Overview

This notebook implements Spectrogram (2D time-frequency) feature extraction for RF-based drone classification using CNN-based transfer learning.

Methodology

We follow a **two-stage pipeline**: (1) Spectrogram Generation → (2) CNN Training (VGG16/ResNet50)

The spectrogram extraction process:

1. Loads raw I/Q samples from .dat files (60 MHz sampling rate)
2. **Normalizes per-file** (Z-score) before segmentation
3. Segments continuous signals into fixed-duration windows (20ms)
4. Computes **STFT** (Short-Time Fourier Transform) with nperseg=1024
5. Converts to **dB scale** and applies **Viridis RGB colormap**
6. Resizes to **224x224x3** for VGG16/ResNet50 compatibility
7. Saves spectrogram images with encoded labels

Why Spectrogram Features?

Spectrograms: Time-frequency representations capture temporal dynamics (e.g., frequency hopping, burst patterns) that static PSD misses.
Ideal for convolutional architectures pre-trained on ImageNet.

Advantages:

- Captures temporal evolution of spectrum

- Leverages pretrained CNNs (transfer learning)
- Visual representation aids debugging
- Effective for complex patterns

Parameter Selection

Aligned with reference implementation:

- **FFT size (nperseg): 1024** - Frequency resolution: ~58.6 kHz/bin
- **Segment duration: 20ms**
- **Output size: 224x224x3** - Matches VGG16/ResNet50 input
- **Colormap: Viridis RGB** - Perceptually uniform, 3-channel for transfer learning
- **Normalization: Per-file Z-score** before segmentation

Downstream Usage

Spectrogram features are consumed by:

- `04_training_cnn_COLAB.ipynb` - VGG16/ResNet50 transfer learning

Reference Alignment

Parameters verified against REFERENTIEL_DRONEDECTECT_RFCLASSIFICATION.md Section 2.2-2.3.

```
In [1]: import sys
sys.path.insert(0, '../src')

import gc
import psutil
import os
import numpy as np
from tqdm import tqdm
from typing import Any, List
```

```

from sklearn.preprocessing import LabelEncoder

from dronedetect import config, data_loader, preprocessing, features

# Memory monitoring utility

def get_memory_mb():
    return psutil.Process(os.getpid()).memory_info().rss / 1024 / 1024

# Create output directory
config.FEATURES_DIR.mkdir(parents=True, exist_ok=True)

print(f"Initial memory: {get_memory_mb():.0f} MB")

```

Initial memory: 424 MB

Specifications

All preprocessing parameters defined here for reproducibility.

```

In [2]: # =====
# PREPROCESSING SPECIFICATIONS
# =====
# All parameters controlling feature extraction defined here for reproducibility.

# --- Signal Segmentation ---
SEGMENT_DURATION_MS = 20          # Window Length (config.DEFAULT_SEGMENT_MS)
                                    # RFClassification results: 10ms→20ms→50ms improves 76.9%→83.6%→89.4%

# --- Feature Extraction Parameters ---
N_FFT = 1024                      # FFT size - RFClassification/run_dronedetect_feat.py
                                    # Frequency resolution = 60MHz/1024 ≈ 58.6 kHz/bin

SPECTROGRAM_SIZE = (224, 224)      # Output image dimensions (config.DEFAULT_SPEC_SIZE)
                                    # Matches VGG16/ResNet input for transfer learning

IQ_DOWNSAMPLE_TARGET = 10000        # Downsampled IQ samples (config.DEFAULT_IQ_DOWNSAMPLE)
                                    # Original: ~1.2M samples/segment → 10k for memory efficiency

```

```

# --- Batch Processing ---
BATCH_SIZE = 10                      # Files per batch (adjust based on available RAM)

# --- Dataset Info ---
SAMPLING_RATE_MHZ = 60                 # DroneDetect V2 sampling frequency
EXPECTED_SEGMENTS_PER_FILE = 100        # Approx (2s recording / 20ms window)

print("Specifications loaded:")
print(f" Segment: {SEGMENT_DURATION_MS}ms | FFT: {N_FFT} | Batch: {BATCH_SIZE} files")

```

Specifications loaded:
Segment: 20ms | FFT: 1024 | Batch: 10 files

Preprocessing Architecture

Modular pipeline system for flexible feature extraction.

```

In [3]: # =====
# PREPROCESSING ARCHITECTURE
# =====

class PreprocessingStep:
    """Base class for preprocessing steps."""

    def process(self, segment: np.ndarray) -> Any:
        """Process a signal segment.

        Args:
            segment: Input signal segment

        Returns:
            Processed output (type depends on step)
        """
        raise NotImplementedError

    def __repr__(self):
        return self.__class__.__name__

```

```
class PSDStep(PreprocessingStep):
    """Power Spectral Density via Welch method with per-sample normalization."""

    def __init__(self, nfft=1024):
        self.nfft = nfft

    def process(self, segment):
        _, psd = features.compute_psd(segment, nfft=self.nfft)
        # Per-sample normalization (reference: REFERENTIEL Section 1.3.1, Line 220)
        # Division by max with zero-protection
        psd_max = np.max(psd)
        if psd_max < 1e-15: # Essentially zero power
            return np.zeros_like(psd)
        return psd / psd_max

    def __repr__(self):
        return f"PSDStep(nfft={self.nfft})"

class SpectrogramStep(PreprocessingStep):
    """Spectrogram via STFT + resize."""

    def __init__(self, target_size=(224, 224)):
        self.target_size = target_size

    def process(self, segment):
        return features.compute_spectrogram(segment, target_size=self.target_size)

    def __repr__(self):
        return f"SpectrogramStep(size={self.target_size})"

class DownsampleIQStep(PreprocessingStep):
    """Downsample IQ via linear interpolation."""

    def __init__(self, target_samples=10000):
        self.target_samples = target_samples

    def process(self, segment):
```

```
        return preprocessing.downsample_iq(segment, target_samples=self.target_samples)

    def __repr__(self):
        return f"DownsampleIQStep(n={self.target_samples})"

class FeaturePipeline:
    """Pipeline orchestrating multiple preprocessing steps."""

    def __init__(self, name: str, steps: List[PreprocessingStep]):
        """Initialize pipeline.

        Args:
            name: Pipeline identifier (used for output filename)
            steps: List of preprocessing steps to apply in order
        """
        self.name = name
        self.steps = steps

    def process_segment(self, segment: np.ndarray) -> np.ndarray:
        """Apply all steps sequentially.

        Args:
            segment: Input signal segment

        Returns:
            Final processed features
        """
        data = segment
        for step in self.steps:
            data = step.process(data)
        return data

    def get_output_filename(self) -> str:
        """Get output filename for this pipeline."""
        return f"{self.name}_features.npz"

    def __repr__(self):
        steps_str = ' → '.join([str(s) for s in self.steps])
        return f"Pipeline({self.name}): {steps_str}"
```

```
print("Preprocessing architecture loaded")
```

```
Preprocessing architecture loaded
```

Pipeline Configuration

Define preprocessing pipelines for different feature types.

```
In [4]: # =====
# PIPELINE CONFIGURATION
# =====

# Single pipeline for Spectrogram features
pipeline = FeaturePipeline(
    name='spectrogram',
    steps=[SpectrogramStep(target_size=SPECTROGRAM_SIZE)])
)

print(f"Configured pipeline: {pipeline}")
```

```
Configured pipeline: Pipeline(spectrogram): SpectrogramStep(size=(224, 224))
```

1. Scan Dataset

```
In [5]: df = data_loader.get_dataset_metadata(config.DATA_DIR)
print(f"Total files: {len(df)}")
```

```
Total files: 195
```

```
In [6]: df.describe(include="all")
```

Out[6]:

| | drone_code | drone_folder | wifi | bluetooth | interference | state | index | file_path | in |
|---------------|------------|--------------|------|-----------|--------------|-------|------------|---|-----|
| count | 195 | 195 | 195 | 195 | 195 | 195 | 195.000000 | | 195 |
| unique | 7 | 7 | 2 | 2 | 2 | 3 | NaN | | 195 |
| top | AIR | AIR | True | True | BOTH | ON | NaN | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| freq | 30 | 30 | 100 | 100 | 100 | 70 | NaN | | 1 |
| mean | NaN | NaN | NaN | NaN | NaN | NaN | 2.000000 | | NaN |
| std | NaN | NaN | NaN | NaN | NaN | NaN | 1.417854 | | NaN |
| min | NaN | NaN | NaN | NaN | NaN | NaN | 0.000000 | | NaN |
| 25% | NaN | NaN | NaN | NaN | NaN | NaN | 1.000000 | | NaN |
| 50% | NaN | NaN | NaN | NaN | NaN | NaN | 2.000000 | | NaN |
| 75% | NaN | NaN | NaN | NaN | NaN | NaN | 3.000000 | | NaN |
| max | NaN | NaN | NaN | NaN | NaN | NaN | 4.000000 | | NaN |



In [7]: `df.head()`

Out[7]:

| | drone_code | drone_folder | wifi | bluetooth | interference | state | index | file_path | interference_1 |
|----------|------------|--------------|------|-----------|--------------|-------|-------|---|----------------|
| 0 | AIR | AIR | True | True | BOTH | FY | 0 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 1 | AIR | AIR | True | True | BOTH | FY | 1 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 2 | AIR | AIR | True | True | BOTH | FY | 2 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 3 | AIR | AIR | True | True | BOTH | FY | 3 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 4 | AIR | AIR | True | True | BOTH | FY | 4 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |



```
In [8]: df.tail()
```

| | drone_code | drone_folder | wifi | bluetooth | interference | state | index | file_path | interference_folder |
|-----|------------|--------------|-------|-----------|--------------|-------|-------|---|---------------------|
| 190 | PHA | PHA | False | False | CLEAN | ON | 0 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 191 | PHA | PHA | False | False | CLEAN | ON | 1 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 192 | PHA | PHA | False | False | CLEAN | ON | 2 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 193 | PHA | PHA | False | False | CLEAN | ON | 3 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |
| 194 | PHA | PHA | False | False | CLEAN | ON | 4 | /home/sambot/win_downloads/DATASETS/drones/Dro... | |



```
In [9]: df.info(verbose=True, show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   drone_code      195 non-null    object 
 1   drone_folder    195 non-null    object 
 2   wifi            195 non-null    bool   
 3   bluetooth       195 non-null    bool   
 4   interference    195 non-null    object 
 5   state           195 non-null    object 
 6   index           195 non-null    int64  
 7   file_path       195 non-null    object 
 8   interference_folder 195 non-null  object 
dtypes: bool(2), int64(1), object(6)
memory usage: 11.2+ KB
```

2. Extract Features (Batch Processing)

Process files in batches to avoid memory saturation. Features are written progressively to disk.

```
In [10]: # =====
# BATCH PROCESSING - Extract Features
# =====

# Initialize Label encoders
from sklearn.preprocessing import LabelEncoder

drone_encoder = LabelEncoder()
interference_encoder = LabelEncoder()
state_encoder = LabelEncoder()

drone_encoder.fit(df['drone_code'].unique())
interference_encoder.fit(df['interference'].unique())
state_encoder.fit(df['state'].unique())

print(f"Label encoders initialized")
print(f" Drones: {drone_encoder.classes_}")
print(f" Interference: {interference_encoder.classes_}")
print(f" States: {state_encoder.classes_}")

# Batch processing
batch_files = []
batch_idx = 0

for batch_start in tqdm(range(0, len(df), BATCH_SIZE), desc="Processing batches"):
    batch_df = df.iloc[batch_start:batch_start + BATCH_SIZE]

    # Storage for current batch
    batch_features = []
    labels_batch = []
    file_ids_batch = []

    print(f"\nBatch {batch_idx}: files {batch_start}-{batch_start + len(batch_df)}")

    for idx, row in batch_df.iterrows():
        try:
            # Load IQ data
            iq_data = data_loader.load_raw_iq(row['file_path'])

            # CRITICAL: Z-score normalization for PSD/Spectrogram

```

```
    iq_normalized = preprocessing.normalize(iq_data)

    # Segment into 20ms windows
    segments = preprocessing.segment_signal(iq_normalized, segment_ms=SEGMENT_DURATION_MS)

    # Process each segment through pipeline
    for seg in segments:
        seg = seg.copy() # Break view to allow memory release

        # Extract features
        feature_output = pipeline.process_segment(seg)
        batch_features.append(feature_output)

        # Encode labels
        labels_batch.append({
            'drone': drone_encoder.transform([row['drone_code']])[0],
            'interference': interference_encoder.transform([row['interference']])[0],
            'state': state_encoder.transform([row['state']])[0]
        })
        file_ids_batch.append(idx)

    del seg

    # Free memory
    del iq_data, iq_normalized, segments
    gc.collect()

except Exception as e:
    print(f"  Error processing {row['file_path']}: {e}")
    continue

# Save batch to disk
batch_file = config.FEATURES_DIR / f'batch_{batch_idx:03d}.npz'
np.savez_compressed(
    batch_file,
    spectrogram=np.array(batch_features),
    labels=np.array(labels_batch, dtype=object),
    file_ids=np.array(file_ids_batch, dtype=np.int32)
)
batch_files.append(batch_file)
```

```
    print(f"  Saved {len(batch_features)} samples to {batch_file.name}")
    print(f"  Memory: {get_memory_mb():.0f} MB")

    # Clean batch data
    del batch_features, labels_batch, file_ids_batch
    gc.collect()

    batch_idx += 1

print(f"\nBatch processing complete: {len(batch_files)} batches saved")
print(f"Memory: {get_memory_mb():.0f} MB")
```

Label encoders initialized

Drones: ['AIR' 'DIS' 'INS' 'MA1' 'MAV' 'MIN' 'PHA']

Interference: ['BOTH' 'CLEAN']

States: ['FY' 'HO' 'ON']

Processing batches: 0% | 0/20 [00:00<?, ?it/s]

Batch 0: files 0-10

Processing batches: 5% | 1/20 [01:26<27:28, 86.77s/it]

Saved 1000 samples to batch_000.npz

Memory: 1045 MB

Batch 1: files 10-20

Processing batches: 10% | 2/20 [02:51<25:36, 85.35s/it]

Saved 1000 samples to batch_001.npz

Memory: 1050 MB

Batch 2: files 20-30

Processing batches: 15% | 3/20 [04:13<23:48, 84.05s/it]

Saved 991 samples to batch_002.npz

Memory: 1045 MB

Batch 3: files 30-40

Processing batches: 20% | 4/20 [05:37<22:20, 83.80s/it]

Saved 1000 samples to batch_003.npz

Memory: 1050 MB

Batch 4: files 40-50

Processing batches: 25% | 5/20 [06:54<20:20, 81.37s/it]

Saved 1000 samples to batch_004.npz

Memory: 1050 MB

Batch 5: files 50-60

Processing batches: 30% |██████| 6/20 [08:21<19:28, 83.44s/it]

Saved 1000 samples to batch_005.npz

Memory: 1050 MB

Batch 6: files 60-70

Processing batches: 35% |██████| 7/20 [09:46<18:12, 84.07s/it]

Saved 1000 samples to batch_006.npz

Memory: 1050 MB

Batch 7: files 70-80

Processing batches: 40% |██████| 8/20 [11:12<16:53, 84.47s/it]

Saved 1000 samples to batch_007.npz

Memory: 1050 MB

Batch 8: files 80-90

Processing batches: 45% |██████| 9/20 [12:38<15:35, 85.01s/it]

Saved 1000 samples to batch_008.npz

Memory: 1050 MB

Batch 9: files 90-100

Processing batches: 50% |██████| 10/20 [14:02<14:07, 84.77s/it]

Saved 1000 samples to batch_009.npz

Memory: 1050 MB

Batch 10: files 100-110

Processing batches: 55% |██████| 11/20 [15:29<12:48, 85.38s/it]

Saved 1000 samples to batch_010.npz

Memory: 1050 MB

Batch 11: files 110-120

Processing batches: 60% |██████| 12/20 [17:02<11:41, 87.64s/it]

Saved 1000 samples to batch_011.npz

Memory: 1050 MB

Batch 12: files 120-130

Processing batches: 65% |██████| 13/20 [18:33<10:20, 88.63s/it]

Saved 987 samples to batch_012.npz

Memory: 1043 MB

Batch 13: files 130-140

Processing batches: 70% |██████| 14/20 [19:58<08:45, 87.63s/it]

Saved 1000 samples to batch_013.npz

Memory: 1050 MB

Batch 14: files 140-150

Processing batches: 75% |██████| 15/20 [21:21<07:11, 86.37s/it]

Saved 1000 samples to batch_014.npz

Memory: 1050 MB

Batch 15: files 150-160

Processing batches: 80% |██████| 16/20 [22:46<05:43, 85.78s/it]

Saved 1000 samples to batch_015.npz

Memory: 1050 MB

Batch 16: files 160-170

Processing batches: 85% |██████| 17/20 [24:07<04:13, 84.40s/it]

Saved 1000 samples to batch_016.npz

Memory: 1050 MB

Batch 17: files 170-180

Processing batches: 90% |██████| 18/20 [25:30<02:48, 84.07s/it]

Saved 1000 samples to batch_017.npz

Memory: 1050 MB

Batch 18: files 180-190

Processing batches: 95% |██████| 19/20 [26:52<01:23, 83.26s/it]

Saved 1000 samples to batch_018.npz

Memory: 1050 MB

Batch 19: files 190-195

Processing batches: 100% |██████| 20/20 [27:32<00:00, 82.64s/it]

```
Saved 500 samples to batch_019.npz
```

```
Memory: 763 MB
```

```
Batch processing complete: 20 batches saved
```

```
Memory: 763 MB
```

```
In [ ]:
```

3. Merge Batches into Final Files

Combine all batch files into single feature files.

```
In [11]:
```

```
# =====
# OPTIMIZED BATCH MERGING (Disk-backed, ZERO RAM accumulation)
# =====
print("Merging batches (disk-backed memmap)...")

MERGE_CHUNK_SIZE = 5 # Process N batches at a time

def merge_pipeline_chunked(pipeline_name, batch_files, chunk_size=5):
    """Merge batches for a single pipeline using disk-backed memory mapping.

    This approach prevents memory saturation by:
    1. Writing directly to disk via memmap (NO RAM accumulation)
    2. Processing one pipeline at a time
    3. Pre-allocating final array size
    4. Keeping memmap for direct npz save (no RAM load)

    Args:
        pipeline_name: Name of pipeline ('psd', 'spectrogram', 'iq')
        batch_files: List of batch file paths
        chunk_size: Number of batches to process simultaneously

    Returns:
        dict with memmap + metadata (labels stay in RAM - small)
    """
    print(f"\n[{pipeline_name}] Processing {len(batch_files)} batches (disk-backed)...")
```

```

# STEP 1: Calculate total size by scanning batches
print(f"[{pipeline_name}] Scanning batches to determine size...")
total_samples = 0
feature_shape = None

for batch_file in batch_files:
    data = np.load(batch_file, allow_pickle=True)
    batch_features = data[pipeline_name]
    total_samples += len(batch_features)
    if feature_shape is None:
        feature_shape = batch_features.shape[1:] # (1024,) or (224,224,3)
    del data

print(f"[{pipeline_name}] Total samples: {total_samples}, feature shape: {feature_shape}")

# STEP 2: Create memmap file (writes directly to disk)
memmap_file = config.FEATURES_DIR / f'{pipeline_name}_features_X.mmap'
final_shape = (total_samples,) + feature_shape

print(f"[{pipeline_name}] Creating memmap: {final_shape} ({np.prod(final_shape)*4/1e9:.2f} GB)")
X_memmap = np.memmap(memmap_file, dtype='float32', mode='w+', shape=final_shape)

# Storage for labels (small, can stay in RAM)
all_drone_labels = []
all_interference_labels = []
all_state_labels = []
all_file_ids = []

# STEP 3: Fill memmap progressively
write_offset = 0

for chunk_start in tqdm(range(0, len(batch_files), chunk_size), desc=f"{pipeline_name} chunks"):
    chunk_batch_files = batch_files[chunk_start:chunk_start + chunk_size]

    for batch_file in chunk_batch_files:
        data = np.load(batch_file, allow_pickle=True)

        # Write features directly to disk
        batch_features = data[pipeline_name]
        n_samples = len(batch_features)

```

```

        X_memmap[write_offset:write_offset + n_samples] = batch_features
        write_offset += n_samples

        # Collect Labels (small)
        labels = data['labels']
        all_drone_labels.extend([label['drone'] for label in labels])
        all_interference_labels.extend([label['interference'] for label in labels])
        all_state_labels.extend([label['state'] for label in labels])
        all_file_ids.append(data['file_ids'])

    del data, batch_features

    # Flush to disk periodically
    X_memmap.flush()
    gc.collect()

    print(f" Written {write_offset}/{total_samples} samples, RAM: {get_memory_mb():.0f} MB")

# STEP 4: Final flush (keep memmap, do NOT Load into RAM)
X_memmap.flush()

final_file_ids = np.concatenate(all_file_ids, axis=0)

return {
    'X_memmap': X_memmap, # Return memmap directly
    'memmap_file': memmap_file,
    'memmap_shape': final_shape,
    'y_drone': np.array(all_drone_labels, dtype=np.int32),
    'y_interference': np.array(all_interference_labels, dtype=np.int32),
    'y_state': np.array(all_state_labels, dtype=np.int32),
    'file_ids': final_file_ids
}

# Process each pipeline separately (reduces peak memory ~3x)
final_data = {}
# Process single pipeline
print(f"\n{'='*60}")
final_data = merge_pipeline_chunked(
    "spectrogram",
    batch_files,

```

```

        chunk_size=MERGE_CHUNK_SIZE
    )
print(f"[spectrogram] Shape: {final_data['memmap_shape']}")
print(f"Memory after spectrogram: {get_memory_mb():.0f} MB")

print("\n" + "="*60)
print("Final shapes (memmap-backed):")
# Process single pipeline
print(f" {pipeline.name}: {final_data['memmap_shape']}")
print(f" File IDs: {final_data['file_ids'].shape} (unique: {len(np.unique(final_data['file_ids']))})")
print(f"Final memory: {get_memory_mb():.0f} MB")

```

Merging batches (disk-backed memmap)...

=====

```

[spectrogram] Processing 20 batches (disk-backed)...
[spectrogram] Scanning batches to determine size...
[spectrogram] Total samples: 19478, feature shape: (224, 224, 3)
[spectrogram] Creating memmap: (19478, 224, 224, 3) (11.73 GB)

spectrogram chunks: 25%|██████| 1/4 [00:14<00:44, 14.80s/it]
Written 4991/19478 samples, RAM: 3630 MB

spectrogram chunks: 50%|██████| 2/4 [00:26<00:25, 13.00s/it]
Written 9991/19478 samples, RAM: 6502 MB

spectrogram chunks: 75%|██████| 3/4 [00:37<00:11, 11.90s/it]
Written 14978/19478 samples, RAM: 9365 MB

spectrogram chunks: 100%|██████| 4/4 [00:44<00:00, 11.07s/it]
Written 19478/19478 samples, RAM: 11950 MB

[spectrogram] Shape: (19478, 224, 224, 3)
Memory after spectrogram: 11950 MB

```

=====

```

Final shapes (memmap-backed):
spectrogram: (19478, 224, 224, 3)
File IDs: (19478,) (unique: 195)
Final memory: 11950 MB

```

4. Save Final Features to Disk

```
In [12]: # Save features for each pipeline (directly from memmap, no RAM Load)
print("Saving features to npz (streaming from memmap)...")

# Save single pipeline
output_file = config.FEATURES_DIR / "spectrogram_features.npz"

# Save directly from memmap (npz will read from disk as needed)
np.savez_compressed(
    output_file,
    X=final_data['X_memmap'], # Memmap array
    y_drone=final_data['y_drone'],
    y_interference=final_data['y_interference'],
    y_state=final_data['y_state'],
    file_ids=final_data['file_ids'],
    drone_classes=drone_encoder.classes_,
    interference_classes=interference_encoder.classes_,
    state_classes=state_encoder.classes_,
    # Metadata for frequency conversion (baseband to absolute RF)
    fs=config.FS,
    center_freq=config.CENTER_FREQ,
    bandwidth=config.BANDWIDTH
)
print(f"Saved spectrogram features to {output_file}")

# Clean up memmap
del final_data['X_memmap']
final_data['memmap_file'].unlink()

# Free remaining memory
del final_data
gc.collect()

print(f"\nAll features saved successfully! Final RAM: {get_memory_mb():.0f} MB")
```

```
Saving features to npz (streaming from memmap)...
Saved spectrogram features to data/features/spectrogram_features.npz
```

```
All features saved successfully! Final RAM: 765 MB
```

5. Verify Saved Files

In [13]:

```
# Load and verify
psd_data = np.load(config.FEATURES_DIR / 'spectrogram_features.npz', allow_pickle=True)
print("Spectrogram features loaded:")
print(f" X shape: {psd_data['X'].shape}")
print(f" y_drone shape: {psd_data['y_drone'].shape}")
print(f" Drone classes: {psd_data['drone_classes']}")
print(f" Interference classes: {psd_data['interference_classes']}")
print(f" State classes: {psd_data['state_classes'])}
```

```
Spectrogram features loaded:
 X shape: (19478, 224, 224, 3)
 y_drone shape: (19478,)
 Drone classes: ['AIR' 'DIS' 'INS' 'MA1' 'MAV' 'MIN' 'PHA']
 Interference classes: ['BOTH' 'CLEAN']
 State classes: ['FY' 'HO' 'ON']
```

Appendix: Stratified Train/Test Split with File-Level Grouping

Problem: Temporal Autocorrelation and Data Leakage

Each `.dat` file in DroneDetect V2 contains approximately 2 seconds of continuous RF signal sampled at 60 MHz. During preprocessing, this signal is segmented into ~100 overlapping windows of 20ms each.

Critical observation: Consecutive segments from the same recording exhibit strong temporal autocorrelation. The RF characteristics (carrier frequency drift, hardware imperfections, environmental noise) remain largely constant within a single acquisition.

If segments from the same source file appear in both training and test sets, the model may learn to recognize recording-specific artifacts rather than generalizable drone RF signatures. This constitutes **data leakage** and leads to overly optimistic performance estimates that fail to generalize to unseen recordings.

Solution: StratifiedGroupKFold

We implement a file-grouped stratified split using `sklearn.model_selection.StratifiedGroupKFold`:

1. **Grouping constraint:** All segments from a given `.dat` file are assigned to the same split (train OR test, never both)
2. **Stratification:** Splits maintain the drone class distribution to ensure balanced representation
3. **Validation:** An assertion verifies zero file overlap between splits

This approach ensures that test set performance reflects the model's ability to generalize to entirely new recordings, providing a realistic estimate of real-world deployment accuracy.

```
In [14]: from sklearn.model_selection import StratifiedGroupKFold

def get_stratified_file_split(X, y, file_ids, test_size=0.2, random_state=42):
    """
    Split data at FILE level to prevent data leakage.

    Segments from the same .dat file (~100 segments) will never appear
    in both train and test sets.

    Parameters
    -----
    X : array-like
        Features (n_samples, ...)
    y : array-like
        Labels for stratification (n_samples,)
    file_ids : array-like
        Source file ID for each sample (n_samples,)
    test_size : float
        Approximate test set proportion (actual may vary due to file grouping)
    random_state : int
        Random seed for reproducibility

    Returns
    -----
    train_idx, test_idx : arrays
        Indices for train/test split
    """
    n_splits = int(1 / test_size) # e.g., test_size=0.2 -> 5 splits -> 1 fold = 20%

    sgkf = StratifiedGroupKFold(n_splits=n_splits, shuffle=True, random_state=random_state)
```

```
# Take first fold as train/test split
train_idx, test_idx = next(sgkf.split(X, y, groups=file_ids))

# Verify no file leakage
train_files = set(file_ids[train_idx])
test_files = set(file_ids[test_idx])
assert len(train_files & test_files) == 0, "Data leakage detected: files in both splits"

return train_idx, test_idx

# Usage example (run after loading features):
# psd_data = np.load(config.FEATURES_DIR / 'spectrogram_features.npz')
# X, y, file_ids = psd_data['X'], psd_data['y_drone'], psd_data['file_ids']
# train_idx, test_idx = get_stratified_file_split(X, y, file_ids)
# X_train, X_test = X[train_idx], X[test_idx]
# y_train, y_test = y[train_idx], y[test_idx]

print("Split function defined: get_stratified_file_split(X, y, file_ids, test_size=0.2)")
```

```
Split function defined: get_stratified_file_split(X, y, file_ids, test_size=0.2)
```