DroneDetect V2 - CNN Classification (Google Colab)

Train CNN classifiers on spectrogram features:

- VGG16 (frozen features + trainable FC)
- ResNet50 (frozen features + trainable FC)
- File-level stratified split to prevent data leakage
- Side-by-side performance comparison

1. Mount Google Drive

In [22]:
```python
from google.colab import drive

drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tru
e).
```

2. Imports

In [23]:
```python
!pip install -U kaleido==0.2.1
```

```
Requirement already satisfied: kaleido==0.2.1 in /usr/local/lib/python3.12/dist-packages (0.2.1)
```

In [24]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import torch
import torch.nn as nn
import torch.optim as optim
```

```python
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import StratifiedGroupKFold
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, f1_score
from tqdm import tqdm
import os
import gc
import psutil
import re
from pathlib import Path

import torchvision.models as tv_models

plt.style.use('seaborn-v0_8-darkgrid')
%matplotlib inline

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Figure saving configuration
NOTEBOOK_NAME = "training_cnn_COLAB"
FIGURES_DIR = Path("figures") / NOTEBOOK_NAME


def save_figure(fig) -> None:
    """Save plotly figure to PNG file using the figure's title as filename."""
    FIGURES_DIR.mkdir(parents=True, exist_ok=True)
    title = fig.layout.title.text if fig.layout.title.text else "untitled"
    filename = re.sub(r'[^\w\s-]', '', title).strip()
    filename = re.sub(r'[\s-]+', '_', filename)
    filepath = FIGURES_DIR / f"{filename}.png"
    try:
        fig.write_image(str(filepath), width=1200, height=800)
        print(f"Saved: {filepath}")
    except Exception as e:
        print(f"Warning: Could not save figure (kaleido required): {e}")
```

```
Using device: cuda
```

3. Configuration

In [25]:
```
CONFIG = {
    # Paths
    'features_path': 'drive/MyDrive/DroneDetect_V2/output/features/spectrogram_features.npz',
    'models_dir': 'drive/MyDrive/DroneDetect_V2/output/models/',
    'test_data_dir': 'drive/MyDrive/DroneDetect_V2/output/sample/test_data/',

    # Split parameters
    'test_size': 0.2,
    'random_state': 42,

    # Training parameters
    'batch_size': 128,
    'epochs': 10,
    'learning_rate': 0.01,

    # Device
    'device': device
}

print(f"Configuration: {CONFIG}")
```

```
Configuration: {'features_path': 'drive/MyDrive/DroneDetect_V2/output/features/spectrogram_features.npz', 'models_dir': 'drive/
MyDrive/DroneDetect_V2/output/models/', 'test_data_dir': 'drive/MyDrive/DroneDetect_V2/output/sample/test_data/', 'test_size':
0.2, 'random_state': 42, 'batch_size': 128, 'epochs': 10, 'learning_rate': 0.01, 'device': device(type='cuda')}
```

4. Model Definitions

In [26]:
```
class VGG16FC(nn.Module):
    """VGG16 with frozen features and trainable classifier.

    This model uses a pre-trained VGG16 backbone with weights frozen,
    replacing the classifier with a new fully connected layer for the
    specific number of classes.

    Attributes:
        features (nn.Sequential): The feature extractor part of VGG16.
        classifier (nn.Linear): The trainable classification layer.
    """
```

```python
    def __init__(self, num_classes: int):
        """Initializes VGG16FC.

        Args:
            num_classes (int): The number of output classes for classification.
        """
        super().__init__()
        # Removed from_array logic as inputs are strictly RGB

        vgg = tv_models.vgg16(weights='IMAGENET1K_V1')
        self.features = nn.Sequential(*list(vgg.children())[:-1])

        for param in self.features.parameters():
            param.requires_grad = False

        self.classifier = nn.Linear(25088, num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward pass of the model.

        Handles input permutation from NHWC to NCHW format if necessary.

        - NHWC: (Batch, Height, Width, Channels) - Common in NumPy/OpenCV.
        - NCHW: (Batch, Channels, Height, Width) - Standard for PyTorch.

        Args:
            x (torch.Tensor): Input tensor. Shape can be (N, H, W, C) or (N, C, H, W).

        Returns:
            torch.Tensor: Model logits.
        """
        # Handle RGB inputs: NHWC -> NCHW
        if x.dim() == 4 and x.shape[-1] == 3:
            x = x.permute(0, 3, 1, 2)

        x = self.features(x)
        # Use flatten(1) to handle non-contiguous tensors and flatten starting from batch dim
        x = x.flatten(1)
        return self.classifier(x)

    def reset_weights(self):
        """Resets the weights of the classifier layer."""
        self.classifier.reset_parameters()


class ResNet50FC(nn.Module):
    """ResNet50 with frozen features and trainable classifier.

    This model uses a pre-trained ResNet50 backbone with weights frozen.
    The final fully connected layer and adaptive pooling are removed and
    replaced with a new linear classifier.

    Attributes:
        features (nn.Sequential): The feature extractor part of ResNet50.
        classifier (nn.Linear): The trainable classification layer.
    """

    def __init__(self, num_classes: int):
        """Initializes ResNet50FC.

        Args:
            num_classes (int): The number of output classes for classification.
        """
        super().__init__()

        resnet = tv_models.resnet50(weights='IMAGENET1K_V1')
        # Remove FC and adaptive pooling
        self.features = nn.Sequential(*list(resnet.children())[:-2])

        for param in self.features.parameters():
            param.requires_grad = False

        # Output of ResNet50 before pooling: 2048 x 7 x 7 = 100352
        self.classifier = nn.Linear(100352, num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward pass of the model.

        Handles input permutation from NHWC to NCHW format if necessary.

        - NHWC: (Batch, Height, Width, Channels) - Common in NumPy/OpenCV.
        - NCHW: (Batch, Channels, Height, Width) - Standard for PyTorch.
        Args:
            x (torch.Tensor): Input tensor. Shape can be (N, H, W, C) or (N, C, H, W).

        Returns:
            torch.Tensor: Model logits.
        """
        if x.dim() == 4 and x.shape[-1] == 3:
            x = x.permute(0, 3, 1, 2)

        x = self.features(x)
        # Use flatten(1) to handle non-contiguous tensors and flatten starting from batch dim
        x = x.flatten(1)
        return self.classifier(x)

    def reset_weights(self):
        """Resets the weights of the classifier layer."""
        self.classifier.reset_parameters()

print("Model classes defined (VGG16FC, ResNet50FC)")
```

```
Model classes defined (VGG16FC, ResNet50FC)
```

4b. Memory-Efficient Dataset

Problem: Spectrogram NPZ file is 11.73 GB (19478 samples x 224x224x3 x 4 bytes). Using fancy indexing `X_train = X[train_idx]` on a memory-mapped array forces NumPy to load the entire subset into RAM, causing OOM on Colab.

Solution: Custom `RGBMemmapDataset` that stores indices and loads one sample at a time via `__getitem__`. The `.copy()` call breaks the memmap view and loads only the requested sample.

Memory savings: 11.73 GB upfront load → ~77 MB per batch (128 samples) + 2 GB PyTorch overhead. Total RAM usage: < 5 GB vs 11.73 GB.

Trade-off: Disk I/O overhead (~10-20s per epoch), but prevents OOM crashes.

In [27]:
```
class RGBMemmapDataset(Dataset):
    """
    Memory-efficient dataset for memmap RGB spectrograms.
```

```python
    Loads one sample at a time from disk instead of loading entire array into RAM.

    Memory optimization: Avoids fancy indexing on memmap which would load
    entire subset into RAM (11.73 GB). Instead, stores indices and loads
    samples individually via __getitem__.

    Memory usage: ~77 MB per batch (128 samples) vs 11.73 GB upfront.

    Parameters
    ----------
    memmap_array : np.memmap
        Memory-mapped array from np.load(..., mmap_mode='r')
    indices : np.ndarray
        Indices for this split (train or test)
    labels : np.ndarray
        Labels for samples
    """

    def __init__(self, memmap_array, indices, labels):
        self.memmap = memmap_array
        self.indices = indices
        self.labels = labels

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        # Map dataset index to original array index
        actual_idx = self.indices[idx]
        # CRITICAL: .copy() breaks memmap view and loads single sample
        rgb = self.memmap[actual_idx].copy()  # (224, 224, 3)

        # Convert to tensor
        x = torch.from_numpy(rgb).float()
        y_label = torch.tensor(self.labels[idx]).long()

        return x, y_label

print("Memory-efficient RGBMemmapDataset class defined")
```

```
Memory-efficient RGBMemmapDataset class defined
```

5. File-Level Stratified Split Function

```
In [28]: 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

print("Split function defined")
```

```
Split function defined
```

6. Streaming Dataset Definition

In [29]:
```
# Memory optimization: use memory mapping to avoid loading full array
data = np.load(CONFIG['features_path'], mmap_mode='r', allow_pickle=True)

X_memmap = data['X']  # Shape: (N, 224, 224, 3) - memory mapped, not loaded
y_drone = data['y_drone'][:]  # Load labels (small, ~76 KB)
y_interference = data['y_interference'][:]
y_state = data['y_state'][:]
file_ids = data['file_ids'][:]  # Load file IDs (small, ~76 KB)
drone_classes = data['drone_classes']
interference_classes = data['interference_classes']
state_classes = data['state_classes']

print(f"Spectrograms shape: {X_memmap.shape}")
print(f"Labels shape: {y_drone.shape}")
print(f"File IDs shape: {file_ids.shape} (unique files: {len(np.unique(file_ids))})")
print(f"Drone classes: {drone_classes}")
print(f"Interference classes: {interference_classes}")
print(f"State classes: {state_classes}")
print(f"Number of classes: {len(drone_classes)}")
```

```
Spectrograms shape: (19478, 224, 224, 3)
Labels shape: (19478,)
File IDs shape: (19478,) (unique files: 195)
Drone classes: ['AIR' 'DIS' 'INS' 'MA1' 'MAV' 'MIN' 'PHA']
Interference classes: ['BOTH' 'CLEAN']
State classes: ['FY' 'HO' 'ON']
Number of classes: 7
```

7. Train/Test Split

In [30]:
```python
print("Performing file-level stratified split...")
train_idx, test_idx = get_stratified_file_split(
    X_memmap, y_drone, file_ids,
    test_size=CONFIG['test_size'],
    random_state=CONFIG['random_state']
)

# Split labels only (X handled by RGBMemmapDataset)
y_train = y_drone[train_idx]
y_test = y_drone[test_idx]
y_interference_test = y_interference[test_idx]
y_state_test = y_state[test_idx]

print(f"Train set: {len(train_idx)} samples")
print(f"Test set: {len(test_idx)} samples")
print(f"Train class distribution: {np.bincount(y_train)}")
print(f"Test class distribution: {np.bincount(y_test)}")
print(f"\nMemory saved: Indices stored instead of 11.73 GB array copy")

# Save test data for reuse
test_data_dir = CONFIG['test_data_dir']
os.makedirs(test_data_dir, exist_ok=True)

# Load full test set into memory from memmap
print("\nLoading test set into memory for saving...")
X_test = X_memmap[test_idx].copy()

# Save full test data
test_data_path = os.path.join(test_data_dir, 'cnn_test_data.npz')
np.savez(
    test_data_path,
    X_test=X_test,
    y_test=y_test,
    y_interference_test=y_interference_test,
    y_state_test=y_state_test,
    test_idx=test_idx,
    file_ids_test=file_ids[test_idx],
    drone_classes=drone_classes,
    interference_classes=interference_classes,
    state_classes=state_classes
)
print(f"Full test data saved to {test_data_path}")

# Save separated files per Drone and Interference (Hierarchical)
print("\nGenerating separated test files (structure: spectrogram/INT/DRONE/)...")

for d_idx, drone_class in enumerate(drone_classes):
    for i_idx, int_class in enumerate(interference_classes):
        # Filter for specific drone and interference
        mask = (y_test == d_idx) & (y_interference_test == i_idx)

        if not np.any(mask):
            continue

        X_sub = X_test[mask]
        y_sub = y_test[mask]
        y_int_sub = y_interference_test[mask]

        # Define components for hierarchy and filename
        data_type = 'spectrogram'
        int_name = str(int_class)
        drone_name = str(drone_class)
        dims = "224x224x3"

        # Create directory structure: output/test_data/{INT}/
        save_dir = os.path.join(test_data_dir, int_name)
        os.makedirs(save_dir, exist_ok=True)

        # Construct filename: spectrogram_{INT}_{DRONE}_224x224x3.npz
        filename = f"{data_type}_{int_name}_{drone_name}_{dims}.npz"
        file_path = os.path.join(save_dir, filename)

        np.savez(
            file_path,
            X=X_sub,
            y=y_sub,
            y_interference=y_int_sub,
            drone_class=drone_class,
            interference_class=int_class
        )
        print(f"  Saved {filename} in {save_dir} ({len(X_sub)} samples)")
```

```
Performing file-level stratified split...
Train set: 15587 samples
Test set: 3891 samples
Train class distribution: [2400 1600 2387 2400 2300 2400 2100]
Test class distribution: [600 400 591 600 700 600 400]

Memory saved: Indices stored instead of 11.73 GB array copy

Loading test set into memory for saving...
Full test data saved to drive/MyDrive/DroneDetect_V2/output/sample/test_data/cnn_test_data.npz

Generating separated test files (structure: spectrogram/INT/DRONE/)...
  Saved spectrogram_BOTH_AIR_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (400 samples)
  Saved spectrogram_CLEAN_AIR_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (200 samples)
  Saved spectrogram_BOTH_DIS_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (100 samples)
  Saved spectrogram_CLEAN_DIS_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (300 samples)
  Saved spectrogram_BOTH_INS_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (291 samples)
  Saved spectrogram_CLEAN_INS_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (300 samples)
  Saved spectrogram_BOTH_MA1_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (300 samples)
  Saved spectrogram_CLEAN_MA1_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (300 samples)
  Saved spectrogram_BOTH_MAV_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (400 samples)
  Saved spectrogram_CLEAN_MAV_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (300 samples)
  Saved spectrogram_BOTH_MIN_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (300 samples)
  Saved spectrogram_CLEAN_MIN_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (300 samples)
  Saved spectrogram_BOTH_PHA_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/BOTH (200 samples)
  Saved spectrogram_CLEAN_PHA_224x224x3.npz in drive/MyDrive/DroneDetect_V2/output/sample/test_data/CLEAN (200 samples)
```

8. Create DataLoaders

In [31]:
```
train_dataset = RGBMemmapDataset(X_memmap, train_idx, y_train)
test_dataset = RGBMemmapDataset(X_memmap, test_idx, y_test)

train_loader = DataLoader(
    train_dataset,
    batch_size=CONFIG['batch_size'],
    shuffle=True,
    num_workers=2,
    pin_memory=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=CONFIG['batch_size'],
    shuffle=False,
    num_workers=2,
    pin_memory=True
)

print(f"Train batches: {len(train_loader)}")
print(f"Test batches: {len(test_loader)}")
print(f"Memory per batch: ~{CONFIG['batch_size'] * 224 * 224 * 3 * 4 / 1024**2:.1f} MB")
```

```
Train batches: 122
Test batches: 31
Memory per batch: ~73.5 MB
```

8b. Memory Profiling Checkpoint

In [32]:

```
process = psutil.Process()
mem_info = process.memory_info()
print(f"Memory usage: {mem_info.rss / 1024**3:.2f} GB")

# Force garbage collection
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
    print(f"GPU memory allocated: {torch.cuda.memory_allocated() / 1024**3:.2f} GB")
    print(f"GPU memory reserved: {torch.cuda.memory_reserved() / 1024**3:.2f} GB")
```

```
Memory usage: 15.50 GB
GPU memory allocated: 0.20 GB
GPU memory reserved: 0.22 GB
```

9. Training Function

In [33]:
```python
def train_model(model, train_loader, test_loader, epochs=10, lr=0.01, device='cuda'):
    """
    Train a PyTorch model with memory cleanup.

    Parameters
    ----------
    model : nn.Module
        Model to train
    train_loader : DataLoader
        Training data loader
    test_loader : DataLoader
        Test data loader
    epochs : int
        Number of epochs
    lr : float
        Learning rate
    device : str
        Device to use ('cuda' or 'cpu')

    Returns
    -------
    model : nn.Module
        Trained model
    history : dict
        Training history
    """
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    history = {
        'train_loss': [],
        'train_acc': [],
        'test_loss': [],
        'test_acc': []
    }

    for epoch in range(epochs):
        # Training phase
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_X, batch_y in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)

            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            train_total += batch_y.size(0)
            train_correct += (predicted == batch_y).sum().item()

        # Validation phase
        model.eval()
        test_loss = 0.0
        test_correct = 0
        test_total = 0

        with torch.no_grad():
            for batch_X, batch_y in test_loader:
                batch_X = batch_X.to(device)
                batch_y = batch_y.to(device)

                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                test_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                test_total += batch_y.size(0)
                test_correct += (predicted == batch_y).sum().item()

        # Record history
        history['train_loss'].append(train_loss / len(train_loader))
        history['train_acc'].append(100 * train_correct / train_total)
        history['test_loss'].append(test_loss / len(test_loader))
        history['test_acc'].append(100 * test_correct / test_total)

        print(f"Epoch {epoch+1}/{epochs}: "
              f"Train Loss: {history['train_loss'][-1]:.4f}, "
              f"Train Acc: {history['train_acc'][-1]:.2f}%, "
              f"Test Loss: {history['test_loss'][-1]:.4f}, "
              f"Test Acc: {history['test_acc'][-1]:.2f}%")

        # Memory cleanup after each epoch
        gc.collect()
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

    return model, history

print("Training function defined")
```

```
Training function defined
```

10. Train VGG16

In [34]:

```
num_classes = len(drone_classes)
vgg_model = VGG16FC(num_classes=num_classes)

print(f"Training VGG16 with {num_classes} classes...")
vgg_model, vgg_history = train_model(
    vgg_model,
    train_loader,
    test_loader,
    epochs=CONFIG['epochs'],
    lr=CONFIG['learning_rate'],
    device=CONFIG['device']
)
print("VGG16 training complete!")
```

```
Training VGG16 with 7 classes...
Epoch 1/10: 100%|██████████| 122/122 [00:08<00:00, 14.20it/s]
Epoch 1/10: Train Loss: 1.8804, Train Acc: 69.26%, Test Loss: 1.1507, Test Acc: 78.98%
Epoch 2/10: 100%|██████████| 122/122 [00:08<00:00, 14.17it/s]
Epoch 2/10: Train Loss: 0.6801, Train Acc: 82.84%, Test Loss: 1.0365, Test Acc: 80.83%
Epoch 3/10: 100%|██████████| 122/122 [00:08<00:00, 14.16it/s]
Epoch 3/10: Train Loss: 0.4813, Train Acc: 86.71%, Test Loss: 1.5858, Test Acc: 75.35%
Epoch 4/10: 100%|██████████| 122/122 [00:08<00:00, 14.13it/s]
Epoch 4/10: Train Loss: 0.3898, Train Acc: 88.99%, Test Loss: 1.3841, Test Acc: 80.98%
Epoch 5/10: 100%|██████████| 122/122 [00:08<00:00, 14.22it/s]
Epoch 5/10: Train Loss: 0.2737, Train Acc: 91.71%, Test Loss: 1.6196, Test Acc: 79.36%
Epoch 6/10: 100%|██████████| 122/122 [00:08<00:00, 14.23it/s]
Epoch 6/10: Train Loss: 0.2507, Train Acc: 92.40%, Test Loss: 1.9620, Test Acc: 79.21%
Epoch 7/10: 100%|██████████| 122/122 [00:08<00:00, 14.21it/s]
Epoch 7/10: Train Loss: 0.3509, Train Acc: 91.26%, Test Loss: 1.8645, Test Acc: 79.75%
Epoch 8/10: 100%|██████████| 122/122 [00:08<00:00, 14.22it/s]
Epoch 8/10: Train Loss: 0.1857, Train Acc: 94.48%, Test Loss: 2.0230, Test Acc: 76.87%
Epoch 9/10: 100%|██████████| 122/122 [00:08<00:00, 14.23it/s]
Epoch 9/10: Train Loss: 0.1481, Train Acc: 95.42%, Test Loss: 1.9568, Test Acc: 78.87%
Epoch 10/10: 100%|██████████| 122/122 [00:08<00:00, 14.20it/s]
Epoch 10/10: Train Loss: 0.1645, Train Acc: 95.11%, Test Loss: 2.0290, Test Acc: 78.51%
VGG16 training complete!
```

11. VGG16 Performance Metrics

In [35]:

```python
vgg_model.eval()
vgg_preds = []
vgg_true = []

with torch.no_grad():
    for batch_X, batch_y in test_loader:
        batch_X = batch_X.to(CONFIG['device'])
        outputs = vgg_model(batch_X)
        _, predicted = torch.max(outputs, 1)
        vgg_preds.extend(predicted.cpu().numpy())
        vgg_true.extend(batch_y.numpy())

vgg_preds = np.array(vgg_preds)
vgg_true = np.array(vgg_true)
```

```python
print("VGG16 Classification Report:")
print(classification_report(vgg_true, vgg_preds, target_names=drone_classes))
print(f"\nVGG16 Accuracy: {accuracy_score(vgg_true, vgg_preds):.4f}")
print(f"VGG16 F1 Score (macro): {f1_score(vgg_true, vgg_preds, average='macro'):.4f}")
```

```
VGG16 Classification Report:
              precision    recall  f1-score   support

         AIR       0.71      0.76      0.73       600
         DIS       0.97      0.86      0.91       400
         INS       0.87      0.75      0.80       591
         MA1       0.65      0.72      0.69       600
         MAV       0.67      0.56      0.61       700
         MIN       1.00      1.00      1.00       600
         PHA       0.73      0.97      0.83       400

    accuracy                           0.79      3891
   macro avg       0.80      0.80      0.80      3891
weighted avg       0.79      0.79      0.78      3891


VGG16 Accuracy: 0.7851
VGG16 F1 Score (macro): 0.7962
```

12. VGG16 Confusion Matrix

In [36]:

```python
cm_vgg = confusion_matrix(vgg_true, vgg_preds)

# Create confusion matrix heatmap with plotly
fig = go.Figure(data=go.Heatmap(
    z=cm_vgg,
    x=list(drone_classes),
    y=list(drone_classes),
    colorscale='Blues',
    text=cm_vgg,
    texttemplate='%{text}',
    textfont={'size': 12},
    hoverongaps=False
))
```

```python
fig.update_layout(
    title='VGG16 Confusion Matrix',
    xaxis_title='Predicted Label',
    yaxis_title='True Label',
    xaxis={'side': 'bottom'},
    yaxis={'autorange': 'reversed'},
    width=800,
    height=700
)
fig.show()
save_figure(fig)
```

```
Saved: figures/training_cnn_COLAB/VGG16_Confusion_Matrix.png
```

13. Train ResNet50

```
In [37]: num_classes = len(drone_classes)
         resnet_model = ResNet50FC(num_classes=num_classes)

         print(f"Training ResNet50 with {num_classes} classes...")
         resnet_model, resnet_history = train_model(
             resnet_model,
             train_loader,
             test_loader,
             epochs=CONFIG['epochs'],
             lr=CONFIG['learning_rate'],
             device=CONFIG['device']
         )
         print("ResNet50 training complete!")
```

```
Training ResNet50 with 7 classes...
Epoch 1/10: 100%|██████████| 122/122 [00:05<00:00, 20.55it/s]
Epoch 1/10: Train Loss: 70.6483, Train Acc: 59.32%, Test Loss: 13.0459, Test Acc: 74.15%
Epoch 2/10: 100%|██████████| 122/122 [00:05<00:00, 20.65it/s]
Epoch 2/10: Train Loss: 12.1760, Train Acc: 74.65%, Test Loss: 12.9827, Test Acc: 73.22%
Epoch 3/10: 100%|██████████| 122/122 [00:05<00:00, 20.63it/s]
Epoch 3/10: Train Loss: 10.6961, Train Acc: 79.26%, Test Loss: 21.0434, Test Acc: 70.26%
Epoch 4/10: 100%|██████████| 122/122 [00:05<00:00, 20.63it/s]
Epoch 4/10: Train Loss: 7.6036, Train Acc: 83.60%, Test Loss: 27.3043, Test Acc: 71.16%
Epoch 5/10: 100%|██████████| 122/122 [00:05<00:00, 20.56it/s]
Epoch 5/10: Train Loss: 8.6377, Train Acc: 83.79%, Test Loss: 19.4189, Test Acc: 73.81%
Epoch 6/10: 100%|██████████| 122/122 [00:05<00:00, 20.62it/s]
Epoch 6/10: Train Loss: 4.2715, Train Acc: 89.77%, Test Loss: 27.0581, Test Acc: 72.94%
Epoch 7/10: 100%|██████████| 122/122 [00:05<00:00, 20.64it/s]
Epoch 7/10: Train Loss: 5.4503, Train Acc: 88.74%, Test Loss: 25.0186, Test Acc: 72.73%
Epoch 8/10: 100%|██████████| 122/122 [00:05<00:00, 20.60it/s]
Epoch 8/10: Train Loss: 4.9373, Train Acc: 89.34%, Test Loss: 22.7478, Test Acc: 76.23%
Epoch 9/10: 100%|██████████| 122/122 [00:05<00:00, 20.64it/s]
Epoch 9/10: Train Loss: 3.3305, Train Acc: 92.24%, Test Loss: 30.9102, Test Acc: 69.73%
Epoch 10/10: 100%|██████████| 122/122 [00:05<00:00, 20.62it/s]
Epoch 10/10: Train Loss: 5.6662, Train Acc: 89.86%, Test Loss: 30.9221, Test Acc: 73.68%
ResNet50 training complete!
```

14. ResNet50 Performance Metrics

In [38]:

```
resnet_model.eval()
resnet_preds = []
resnet_true = []

with torch.no_grad():
    for batch_X, batch_y in test_loader:
        batch_X = batch_X.to(CONFIG['device'])
        outputs = resnet_model(batch_X)
        _, predicted = torch.max(outputs, 1)
        resnet_preds.extend(predicted.cpu().numpy())
        resnet_true.extend(batch_y.numpy())

resnet_preds = np.array(resnet_preds)
resnet_true = np.array(resnet_true)

print("ResNet50 Classification Report:")
print(classification_report(resnet_true, resnet_preds, target_names=drone_classes))
print(f"\nResNet50 Accuracy: {accuracy_score(resnet_true, resnet_preds):.4f}")
print(f"ResNet50 F1 Score (macro): {f1_score(resnet_true, resnet_preds, average='macro'):.4f}")
```

```
ResNet50 Classification Report:
              precision    recall  f1-score   support

         AIR       0.82      0.72      0.77       600
         DIS       0.73      0.94      0.82       400
         INS       0.70      0.80      0.75       591
         MA1       0.54      0.74      0.62       600
         MAV       0.64      0.40      0.49       700
         MIN       1.00      0.97      0.99       600
         PHA       0.87      0.69      0.77       400

    accuracy                           0.74      3891
   macro avg       0.76      0.75      0.74      3891
weighted avg       0.75      0.74      0.73      3891


ResNet50 Accuracy: 0.7368
ResNet50 F1 Score (macro): 0.7433
```

15. ResNet50 Confusion Matrix

In [39]:
```python
cm_resnet = confusion_matrix(resnet_true, resnet_preds)

# Create confusion matrix heatmap with plotly
fig = go.Figure(data=go.Heatmap(
    z=cm_resnet,
    x=list(drone_classes),
    y=list(drone_classes),
    colorscale='Greens',
    text=cm_resnet,
    texttemplate='%{text}',
    textfont={'size': 12},
    hoverongaps=False
))

fig.update_layout(
    title='ResNet50 Confusion Matrix',
    xaxis_title='Predicted Label',
    yaxis_title='True Label',
    xaxis={'side': 'bottom'},
    yaxis={'autorange': 'reversed'},
    width=800,
    height=700
)
fig.show()
save_figure(fig)
```

```
Saved: figures/training_cnn_COLAB/ResNet50_Confusion_Matrix.png
```

16. Side-by-Side Performance Comparison

In [40]:
```
# Training curves comparison
fig = make_subplots(rows=1, cols=2, subplot_titles=('Training History Comparison', 'Final Performance Comparison'))

# Training curves
epochs = list(range(1, len(vgg_history['train_acc']) + 1))

fig.add_trace(go.Scatter(x=epochs, y=vgg_history['train_acc'], mode='lines+markers', name='VGG16 Train', line=dict(color='blue
fig.add_trace(go.Scatter(x=epochs, y=vgg_history['test_acc'], mode='lines+markers', name='VGG16 Test', line=dict(color='blue',
fig.add_trace(go.Scatter(x=epochs, y=resnet_history['train_acc'], mode='lines+markers', name='ResNet50 Train', line=dict(color
fig.add_trace(go.Scatter(x=epochs, y=resnet_history['test_acc'], mode='lines+markers', name='ResNet50 Test', line=dict(color='

# Final metrics comparison
models = ['VGG16', 'ResNet50']
accuracies = [accuracy_score(vgg_true, vgg_preds), accuracy_score(resnet_true, resnet_preds)]
f1_scores = [f1_score(vgg_true, vgg_preds, average='macro'), f1_score(resnet_true, resnet_preds, average='macro')]

fig.add_trace(go.Bar(x=models, y=accuracies, name='Accuracy', marker_color='steelblue'), row=1, col=2)
fig.add_trace(go.Bar(x=models, y=f1_scores, name='F1 Score (macro)', marker_color='coral'), row=1, col=2)

fig.update_layout(
    title='CNN Training Comparison - VGG16 vs ResNet50',
    height=500,
    width=1200,
    barmode='group'
)
fig.update_xaxes(title_text='Epoch', row=1, col=1)
fig.update_yaxes(title_text='Accuracy (%)', row=1, col=1)
fig.update_xaxes(title_text='Model', row=1, col=2)
fig.update_yaxes(title_text='Score', row=1, col=2)

fig.show()
save_figure(fig)

print("\nFinal Results:")
print(f"VGG16    - Accuracy: {accuracies[0]:.4f}, F1: {f1_scores[0]:.4f}")
print(f"ResNet50 - Accuracy: {accuracies[1]:.4f}, F1: {f1_scores[1]:.4f}")
```

```
Saved: figures/training_cnn_COLAB/CNN_Training_Comparison_VGG16_vs_ResNet50.png

Final Results:
VGG16    - Accuracy: 0.7851, F1: 0.7962
ResNet50 - Accuracy: 0.7368, F1: 0.7433
```

17. Save Models

In [41]:
```
os.makedirs(CONFIG['models_dir'], exist_ok=True)

# Save VGG16
vgg_path = os.path.join(CONFIG['models_dir'], 'vgg16_cnn.pth')
torch.save({
    'model_state_dict': vgg_model.state_dict(),
    'history': vgg_history,
    'num_classes': num_classes,
    'drone_classes': drone_classes
}, vgg_path)
print(f"VGG16 model saved to {vgg_path}")

# Save ResNet50
resnet_path = os.path.join(CONFIG['models_dir'], 'resnet50_cnn.pth')
torch.save({
    'model_state_dict': resnet_model.state_dict(),
    'history': resnet_history,
    'num_classes': num_classes,
    'drone_classes': drone_classes
}, resnet_path)
print(f"ResNet50 model saved to {resnet_path}")
```

```
VGG16 model saved to drive/MyDrive/DroneDetect_V2/output/models/vgg16_cnn.pth
ResNet50 model saved to drive/MyDrive/DroneDetect_V2/output/models/resnet50_cnn.pth
```

18. Summary

In [42]:
```
print("=" * 60)
print("CNN TRAINING SUMMARY")
print("=" * 60)

print(f"\nDataset: {len(y_drone)} total | {len(y_train)} train | {len(y_test)} test | {num_classes} classes")
print(f"Training: {CONFIG['epochs']} epochs, batch_size={CONFIG['batch_size']}, lr={CONFIG['learning_rate']}, device={CONFIG['

print(f"\nVGG16: Accuracy={accuracy_score(vgg_true, vgg_preds):.4f}, F1={f1_score(vgg_true, vgg_preds, average='macro'):.4f}")
```

```python
print(f"ResNet50: Accuracy={accuracy_score(resnet_true, resnet_preds):.4f}, F1={f1_score(resnet_true, resnet_preds, average='m

print(f"\nModels saved to: {CONFIG['models_dir']}")
print("=" * 60)
```

```
============================================================
CNN TRAINING SUMMARY
============================================================

Dataset: 19478 total | 15587 train | 3891 test | 7 classes
Training: 10 epochs, batch_size=128, lr=0.01, device=cuda

VGG16: Accuracy=0.7851, F1=0.7962
ResNet50: Accuracy=0.7368, F1=0.7433

Models saved to: drive/MyDrive/DroneDetect_V2/output/models/
============================================================
```