Bird Audio Classification with EfficientNet-B7 and SpecAugment

Overview

This project implements a deep learning-based audio classification model using **EfficientNet-B7** for bird sound classification. The approach leverages **transfer learning** and **SpecAugment** for improved feature extraction and robustness.

The workflow consists of:

- **Preprocessing Audio Data**: Converting bird sound recordings into **Mel spectrograms**.
- **Data Augmentation**: Applying **SpecAugment** (time and frequency masking).
- **Model Architecture**: Utilizing a **pretrained EfficientNet-B7** with a modified classifier layer.
- **Training and Evaluation**: Implementing **mixed-precision training (AMP)** and **OneCycleLR** for optimized learning rate scheduling.
- **Checkpointing and Resumption**: Saving the best model weights and allowing for resuming training.

Data Preprocessing

1. **Audio Loading & Resampling**: Audio files are loaded and resampled to a fixed sample rate of **32 kHz**.
2. **Spectrogram Generation**: Mel spectrograms are computed using **torchaudio**.
3. **SpecAugment**: Random **time and frequency masking** is applied to improve generalization.
4. **Resizing**: Spectrograms are resized to **224x224 pixels** and converted to **3-channel images**.

Model Architecture

- Uses **EfficientNet-B7** pretrained on ImageNet.
- The final classifier layer is replaced with a **fully connected layer** matching the number of bird species.
- **Cross-entropy loss with label smoothing** is applied for better regularization.

Training Pipeline

- Uses **AdamW optimizer** with **OneCycleLR** learning rate scheduling.
- **Automatic Mixed Precision (AMP)** for efficient training.
- **Checkpoints**:
  - Loads pretrained weights on first run.
  - Resumes training from the best saved checkpoint if available.
  - Saves the best-performing model (`best_model.pth`).

Steps to Run the Model

1. **Install Dependencies:** Ensure that `torch`, `torchaudio`, `torchvision`, and other required libraries are installed.
2. **Set Configuration**: Update `config` parameters, including `data_path`, `learning_rate`, and `epochs`.
3. **Run Training**: Execute the script to preprocess audio, train the model, and evaluate performance.

Outputs

- **Best Model Weights**: Saved as `best_model.pth`.
- **Training Metrics**: Loss and accuracy stored in `training_metrics.pkl`.
- **Evaluation Results:** Displays test accuracy on unseen bird audio samples.

Key Features

▯ **Transfer Learning**: EfficientNet-B7 for high-performance classification.
▯ **Data Augmentation**: SpecAugment to improve robustness.
▯ **AMP for Speedup**: Faster and memory-efficient training.
▯ **Checkpointing**: Saves and resumes best model performance.
▯ **OneCycleLR**: Dynamic learning rate scheduling for improved convergence.

Research Papers & Resources

Below are the key research papers and resources that inspired and contributed to this project:

▯ **PANNs: Large-Scale Pretrained Audio Neural Networks for Audio Pattern Recognition**
A comprehensive study on large-scale pretrained audio neural networks, demonstrating their effectiveness in **audio pattern recognition** tasks.

▯ **SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition**
Introduces **SpecAugment**, a powerful data augmentation technique that enhances **robustness in speech recognition models** by applying time and frequency masking.

```python
# Load necessary library
import os
import glob
import random
import pickle
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

```
import torchaudio
from torchvision.models import efficientnet_b7,
EfficientNet_B7_Weights
import torch.nn.functional as F

from tqdm import tqdm

# ------------------------------
# 1. Reproducibility Utilities
# ------------------------------
def set_seed(seed):
    """Set seed for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
```

Configuration

The `config` dictionary stores key hyperparameters and settings for the project. Each parameter plays a crucial role in controlling how the model is trained, tested, and executed. Below is a description of each configuration parameter:

Parameter	Description	Example Value
`seed`	Random seed to ensure reproducibility. Helps produce consistent results across multiple runs.	`42`
`data_path`	Path to the dataset. In this case, it's pointing to bioacoustics data in `.mp3` format.	`/kaggle/input/bioacoustics-data/osa bird recordings/**/*.mp3`
`test_size`	Fraction of the dataset used for testing.	`0.1` (10%)
`val_size`	Fraction of the dataset used for validation.	`0.1` (10%)

Parameter	Description	Example Value
`sample_rate`	Sampling rate for audio files in Hertz (Hz).	`32000`
`duration`	Duration of the audio to use per file (in seconds).	`60`
`num_workers`	Number of worker threads for data loading operations. Improves data loading performance.	`4`
`num_epochs`	Number of training epochs (how many complete passes through the dataset).	`1`
`learning_rate`	The base learning rate for the optimizer. Controls how much the model adjusts in each step.	`1e-3`
`weight_decay`	Regularization parameter to avoid overfitting. Applies L2 penalty on model weights.	`1e-4`
`max_lr`	Maximum learning rate.	`1e-3`
`device`	Specifies whether the code should use a GPU (`cuda`) or CPU for computations.	`"cuda:0"` or `"cpu"`
`initial_checkpoint_path`	Path to the model checkpoint file to initialize training.	`/kaggle/working/best_model.pth`
`save_checkpoint_dir`	Path to a saved model checkpoint to resume training from.	`/kaggle/working/best_model.pth`

```
# ------------------------------
# 2. Configuration
# ------------------------------
config = {
    "seed": 42,

    # Load multiple folder directories
    "data_path": [
        "/kaggle/input/bird-recording/osa bird
recordings/train_data/**/*.mp3",
    ],

    "test_size": 0.1,
    "val_size": 0.1,
    "sample_rate": 32000,
    "duration": 60,          # seconds of audio to use per file
    "num_workers": 4,
    "num_epochs": 1,
    "learning_rate": 1e-3,
    "weight_decay": 1e-4,
    "max_lr": 1e-3,
    "device": "cuda:0" if torch.cuda.is_available() else "cpu",
    "initial_checkpoint_path": "/kaggle/input/bioacoustics-model-
weight/best_model.pth",
    "save_checkpoint_dir": "/kaggle/working/best_model.pth"
}
```

Data Collection and Splitting

This section contains functions to **load audio files** from one or more directories and **split the dataset** into training, validation, and test sets.

`load_audio_files(path_patterns)`

This function scans one or more provided file path patterns, extracts file paths, and assigns labels based on their parent directory. It accepts either a single glob pattern (as a string) or a list of glob patterns.

Parameters:

- `path_patterns` *(str or list of str)*: A file path pattern (e.g., `"/path/to/audio/**/*.mp3"`) or a list of such patterns to search for audio files recursively.

Returns:

- `df` *(pandas.DataFrame)*: A DataFrame containing:
  - `'filepath'`: Full path of the audio file.
  - `'label'`: The category or label of the audio file, inferred from its parent directory.

`split_data(df, test_size, val_size, random_state=42)`

This function splits the dataset into **training, validation, and test sets**, ensuring that the splits are stratified by label.

Parameters:

- `df` *(pandas.DataFrame)*: The dataset containing `'filepath'` and `'label'`.
- `test_size` *(float)*: Proportion of the dataset to allocate for testing.
- `val_size` *(float)*: Proportion of the remaining dataset to allocate for validation.
- `random_state` *(int, default=42)*: Seed value for reproducibility.

Returns:

- `train` *(pandas.DataFrame)*: Training set.
- `val` *(pandas.DataFrame)*: Validation set.
- `test` *(pandas.DataFrame)*: Test set.

This ensures that:

- The test set is `test_size` fraction of the full dataset.
- The validation set is `val_size` fraction of the remaining data after the test split.
- Data is stratified, meaning the label distribution remains consistent across all splits.

```
# -----------------------------
# 3. Data Collection and Splitting
# -----------------------------
def load_audio_files(path_patterns):
    """
    Returns a DataFrame with columns: 'filepath' and 'label'.
    Assumes that the parent directory of each file is its label.
    Accepts a single glob pattern (str) or a list of glob patterns.
    """
    if isinstance(path_patterns, str):
        path_patterns = [path_patterns]

    data = []
    for pattern in path_patterns:
        file_paths = glob.glob(pattern, recursive=True)
        for fp in file_paths:
            label = os.path.basename(os.path.dirname(fp))
            data.append({'filepath': fp, 'label': label})
    return pd.DataFrame(data)

def split_data(df, test_size, val_size, random_state=42):
    """
    Split the dataframe into train, validation, and test sets.
    Stratify based on the label.
    """
    train_val, test = train_test_split(df, test_size=test_size,
                                       stratify=df['label'],
random_state=random_state)
    train, val = train_test_split(train_val, test_size=val_size,
                                  stratify=train_val['label'],
random_state=random_state)
    return train, val, test
```

Dataset Definition with Audio Augmentation

This section defines the **`BirdAudioDataset`** class, a custom PyTorch dataset designed to process bird audio recordings, apply transformations, and generate spectrogram images suitable for deep learning models.

`BirdAudioDataset`

This class loads audio files, converts them into **Mel spectrograms**, applies **data augmentation** (if enabled), and normalizes the data.

Initialization (`__init__` method)

The dataset is initialized with key parameters:

Parameters:

- `df` *(pandas.DataFrame)*: A dataset containing columns `'filepath'` (audio file path) and `'label_id'` (numeric label).
- `sample_rate` *(int)*: The target sampling rate (Hz) to which all audio files will be resampled.
- `duration` *(int)*: The number of seconds of audio to use per file.
- `augment` *(bool, default=False)*: Whether to apply **SpecAugment**-style data augmentation.

Preprocessing Steps:

- The number of samples per file is calculated as `sample_rate * duration`.
- **Mel Spectrogram Transformation**:
  - Converts audio into a **Mel Spectrogram** with 128 Mel frequency bins.
  - Converts amplitude values to **decibels (dB)** using `AmplitudeToDB()`.
- **SpecAugment (if enabled)**:
  - `FrequencyMasking`: Masks random frequency bands to improve generalization.
  - `TimeMasking`: Masks random time intervals to simulate real-world distortions.

Dataset Length (`__len__` method)

Returns the number of samples in the dataset:

```python
def __len__(self):
    return len(self.df)
```

Loading & Processing Audio (`__getitem__` method)

This method:

1. **Loads the audio file** using `torchaudio.load(filepath)`.
2. **Converts stereo to mono** (if applicable).
3. **Resamples** to the target `sample_rate` (if needed).

4. **Trims/Pads** the waveform to the required duration.
5. **Generates a Mel spectrogram** and converts it to dB scale.
6. **Applies SpecAugment transformations** (if enabled).
7. **Normalizes and resizes** the spectrogram to **224×224** pixels.
8. **Converts the spectrogram into a 3-channel image** for compatibility with deep learning models.
9. **Returns** the processed **spectrogram image** and the **label**.

Returns:

- `image` *(Tensor, shape* `[3, 224, 224]`*)*: A spectrogram image with 3 color channels.
- `label` *(int)*: The corresponding label for the audio file.

```
# -----------------------------
# 4. Dataset Definition with Audio Augmentation
# -----------------------------
class BirdAudioDataset(Dataset):
    def __init__(self, df, sample_rate, duration, augment=False):
        """
        df: DataFrame with columns 'filepath' and 'label_id'
        sample_rate: target sample rate (Hz)
        duration: duration (in seconds) to use from each audio file
        augment: whether to apply SpecAugment style augmentation
        """
        self.df = df.reset_index(drop=True)
        self.sample_rate = sample_rate
        self.duration = duration
        self.num_samples = sample_rate * duration
        self.augment = augment

        # Create MelSpectrogram transform.
        self.mel_transform = torchaudio.transforms.MelSpectrogram(
            sample_rate=sample_rate, n_fft=1024, hop_length=512,
n_mels=128
        )
        self.amplitude_to_db = torchaudio.transforms.AmplitudeToDB()

        # SpecAugment transforms (if augment=True)
        if self.augment:
            self.freq_mask =
torchaudio.transforms.FrequencyMasking(freq_mask_param=15)
            self.time_mask =
torchaudio.transforms.TimeMasking(time_mask_param=30)

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        filepath = row['filepath']
        # Load and process audio.
        waveform, sr = torchaudio.load(filepath)
        if waveform.shape[0] > 1:
            waveform = waveform.mean(dim=0, keepdim=True)
        if sr != self.sample_rate:
            resampler = torchaudio.transforms.Resample(sr,
self.sample_rate)
            waveform = resampler(waveform)
        if waveform.shape[1] < self.num_samples:
            padding = self.num_samples - waveform.shape[1]
            waveform = F.pad(waveform, (0, padding))
        else:
            waveform = waveform[:, :self.num_samples]

        mel_spec = self.mel_transform(waveform)
        mel_spec_db = self.amplitude_to_db(mel_spec)

        if self.augment:
            mel_spec_db = self.freq_mask(mel_spec_db)
            mel_spec_db = self.time_mask(mel_spec_db)

        # Normalize and resize.
        mel_spec_db = (mel_spec_db - mel_spec_db.mean()) /
(mel_spec_db.std() + 1e-9)
        mel_spec_db = F.interpolate(mel_spec_db.unsqueeze(0),
size=(224, 224),
                                    mode='bilinear',
align_corners=False).squeeze(0)
        image = mel_spec_db.repeat(3, 1, 1)  # Convert to 3 channels

        label = row['label_id']
        return image, label
```

Model Definition (EfficientNet-B7 with Fine-Tuning)

This section defines the **`BirdClassifier`** model, which fine-tunes **EfficientNet-B7** for bird sound classification. EfficientNet-B7 is a **state-of-the-art** convolutional neural network (CNN) known for its high performance on image classification tasks.

`BirdClassifier` Class

This class initializes a pre-trained **EfficientNet-B7** model and modifies the classifier layer to match the number of output classes.

Initialization (`__init__` method)

- Loads a pre-trained **EfficientNet-B7** model using `efficientnet_b7()`.
- Retrieves the default **pre-trained weights** (`EfficientNet_B7_Weights.DEFAULT`).
- Replaces the final **fully connected (FC) layer** with a new `nn.Linear` layer to match the number of output classes.

Parameters:

- `num_classes` *(int)*: The number of classes in the dataset.

Modifications:

- Extracts the number of input features from the **original classifier**.
- Replaces the final FC layer with a new linear layer of shape **(in_features, num_classes)**.

Forward Pass (`forward` method)

Defines the forward propagation of input **image tensors** through the model.

- **Input**: A batch of images (`x`) with shape `[batch_size, 3, 224, 224]`.
- **Output**: A tensor of shape `[batch_size, num_classes]`, containing class logits (before applying softmax).

```
def forward(self, x):
    return self.model(x)
```

Key Features of the Model

- **Uses EfficientNet-B7**, a **highly efficient** CNN with a strong accuracy-to-performance ratio.
- **Leverages pre-trained weights**, allowing for **transfer learning**—reducing the need for large datasets.
- **Replaces the classifier** to accommodate a new classification task (i.e., bird sound spectrograms).
- **Outputs logits** for classification, which can be converted to probabilities using `torch.nn.functional.softmax`.

```
# -----------------------------
# 5. Model Definition (EfficientNet-B7 with Fine-Tuning)
# -----------------------------
class BirdClassifier(nn.Module):
    def __init__(self, num_classes):
        super(BirdClassifier, self).__init__()
        # Use EfficientNet-B7 and its default weights.
        weights = EfficientNet_B7_Weights.DEFAULT
        self.model = efficientnet_b7(weights=weights)
        in_features = self.model.classifier[1].in_features
        self.model.classifier[1] = nn.Linear(in_features, num_classes)

    def forward(self, x):
        return self.model(x)
```

Training and Evaluation (with AMP and Checkpointing)

This section defines functions for **training, evaluating, and testing** the model using **Automatic Mixed Precision (AMP)** for efficient computation and **checkpointing** to save the best model.

`train_model` Function

This function trains the **EfficientNet-B7** model using **AdamW optimizer, OneCycle learning rate scheduling, and label smoothing** for better generalization.

Parameters:

- `model` *(nn.Module)*: The PyTorch model to train.
- `train_loader` *(DataLoader)*: DataLoader for the training dataset.
- `val_loader` *(DataLoader)*: DataLoader for the validation dataset.
- `device` *(str)*: `"cuda"` or `"cpu"`, based on availability.
- `num_epochs` *(int)*: Total number of training epochs.
- `max_lr` *(float)*: Maximum learning rate for **OneCycleLR** scheduler.

Training Process:

1. **Loss & Optimization Setup**
   - Uses **CrossEntropyLoss** with `label_smoothing=0.1` to improve generalization.
   - Uses **AdamW optimizer** for training.
   - Uses **OneCycleLR scheduler** for dynamic learning rate adjustments.
   - Uses **AMP GradScaler** for mixed precision training (reducing memory and increasing speed).
2. **Training Loop**
   - Iterates through the dataset, computes loss, and updates model parameters.
   - Uses **autocast** for mixed precision computations.
   - Tracks **training loss and accuracy**.
3. **Validation Loop**
   - Evaluates the model on the validation set without updating weights.
   - Tracks **validation loss and accuracy**.
4. **Model Checkpointing**
   - Saves the model if the validation accuracy improves.
5. **Returns:**
   - The **trained model**.
   - A dictionary containing **training and validation metrics** (loss and accuracy).

`test_model` Function

This function evaluates the trained model on a **test dataset**.

Parameters:

- `model` *(nn.Module)*: Trained model.
- `test_loader` *(DataLoader)*: DataLoader for the test dataset.
- `device` *(str)*: `"cuda"` or `"cpu"`, based on availability.

Testing Process:

1. Sets the model to **evaluation mode** (`model.eval()`).
2. Iterates through the test dataset, making predictions.
3. Computes the **test accuracy**.

Returns:

- Prints the **test accuracy**.

Key Features:

- **Automatic Mixed Precision (AMP)**: Uses `torch.amp.autocast()` for faster training with reduced memory consumption.
- **OneCycle Learning Rate Scheduler**: Adjusts learning rate dynamically for stable convergence.
- **Label Smoothing (0.1)**: Reduces overconfidence in predictions and improves generalization.
- **Checkpointing**: Saves the best model when validation accuracy improves.
- **Progress Tracking**: Uses `tqdm` for real-time monitoring of training progress.

```
# ------------------------------
# 6. Training and Evaluation (with AMP and Checkpointing)
# ------------------------------
def train_model(model, train_loader, val_loader, device, num_epochs,
max_lr):
    model.to(device)

    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
    optimizer = optim.AdamW(model.parameters(), lr=max_lr,
weight_decay=config["weight_decay"])
    total_steps = len(train_loader) * num_epochs
    scheduler = optim.lr_scheduler.OneCycleLR(optimizer,
max_lr=max_lr,
                                               total_steps=total_steps,
                                               pct_start=0.1,
anneal_strategy='cos',
                                               div_factor=25.0,
final_div_factor=1e4)

    # Initialize AMP GradScaler using the updated API.
    scaler = torch.amp.GradScaler()

    best_val_acc = 0.0
    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []

    for epoch in range(num_epochs):
        model.train()
        running_loss, running_corrects, total = 0.0, 0, 0

        train_pbar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{num_epochs} Training", leave=False)
        for inputs, labels in train_pbar:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()

            # Use the new autocast API with explicit device type.
            with torch.amp.autocast(device_type="cuda"):
                outputs = model(inputs)
                loss = criterion(outputs, labels)

            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scheduler.step()
            scaler.update()


            running_loss += loss.item() * inputs.size(0)
            _, preds = torch.max(outputs, 1)
            running_corrects += (preds == labels).sum().item()
            total += labels.size(0)
            train_pbar.set_postfix(loss=loss.item())

        epoch_loss = running_loss / total
        epoch_acc = running_corrects / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        # Validation phase.
        model.eval()
        val_running_loss, val_running_corrects, val_total = 0.0, 0, 0
        val_pbar = tqdm(val_loader, desc=f"Epoch
{epoch+1}/{num_epochs} Validation", leave=False)
        with torch.no_grad():
            for inputs, labels in val_pbar:
                inputs, labels = inputs.to(device), labels.to(device)
                with torch.amp.autocast(device_type="cuda"):
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                val_running_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_running_corrects += (preds == labels).sum().item()
                val_total += labels.size(0)
                val_pbar.set_postfix(loss=loss.item())

        epoch_val_loss = val_running_loss / val_total
        epoch_val_acc = val_running_corrects / val_total
        val_losses.append(epoch_val_loss)
        val_accuracies.append(epoch_val_acc)

        print(f"Epoch [{epoch+1}/{num_epochs}] | Train Loss:
{epoch_loss:.4f}, Train Acc: {epoch_acc:.4f} | "
              f"Val Loss: {epoch_val_loss:.4f}, Val Acc:
{epoch_val_acc:.4f}")

        # Checkpointing: Save model if validation accuracy improves.
        if epoch_val_acc > best_val_acc:
            best_val_acc = epoch_val_acc
            torch.save(model.state_dict(),
config["save_checkpoint_dir"])
            print(f"Best model updated (Val Acc: {best_val_acc:.4f}).
Checkpoint saved.")

    metrics = {
        'train_loss': train_losses,
        'train_acc': train_accuracies,
        'val_loss': val_losses,
        'val_acc': val_accuracies
    }
    return model, metrics

def test_model(model, test_loader, device):
    model.eval()
    correct, total = 0, 0
    for inputs, labels in tqdm(test_loader, desc="Testing"):
        inputs, labels = inputs.to(device), labels.to(device)
        with torch.amp.autocast(device_type="cuda"):
            outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)
    acc = correct / total
    print(f"Test Accuracy: {acc:.4f}")
```

Main Function: Data Loading, Training, and Saving Metrics

This section defines the **`main()`** function, which handles the **entire pipeline** from data loading, dataset preparation, model training, evaluation, and saving metrics.

Overview of `main()`

1. **Sets the random seed** for reproducibility.
2. **Loads and processes the dataset**, assigning unique labels to each bird species.
3. **Splits the data** into training, validation, and test sets.
4. **Creates DataLoaders** for efficient batch processing.
5. **Initializes the model** with EfficientNet-B7.
6. **Loads model checkpoints** (if available) to resume training.
7. **Trains the model** and evaluates it on the validation dataset.
8. **Tests the trained model** on the test dataset.
9. **Saves training metrics** to a `.pkl` file.

Step-by-Step Breakdown

1. Set Random Seed for Reproducibility

```
set_seed(config["seed"])
```

- Ensures consistent results across multiple runs.

2. Load and Process the Data

```
df = load_audio_files(config["data_path"])
labels_sorted = sorted(df['label'].unique())
label2id = {label: i for i, label in enumerate(labels_sorted)}
df['label_id'] = df['label'].map(label2id)
print(f"Found {len(df)} audio files across {len(labels_sorted)}
classes.")
```

- Loads audio file paths and assigns labels based on directory names.
- Maps each unique label to a numeric **label ID**.

3. Split Data into Train, Validation, and Test Sets

```
train_df, val_df, test_df = split_data(df, config["test_size"],
config["val_size"], random_state=config["seed"])
print(f"Train: {len(train_df)}, Val: {len(val_df)}, Test:
{len(test_df)}")
```

- Uses **stratified splitting** to maintain class balance.

4. Create Datasets and DataLoaders

```
train_dataset = BirdAudioDataset(train_df, config["sample_rate"],
config["duration"], augment=True)
val_dataset = BirdAudioDataset(val_df, config["sample_rate"],
config["duration"], augment=False)
test_dataset = BirdAudioDataset(test_df, config["sample_rate"],
config["duration"], augment=False)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,
num_workers=config["num_workers"])
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,
num_workers=config["num_workers"])
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False,
num_workers=config["num_workers"])
```

- **Augments training data** using SpecAugment.
- **Creates DataLoaders** for efficient mini-batch processing.

5. Initialize Model

```
num_classes = len(labels_sorted)
model = BirdClassifier(num_classes)
device = torch.device(config["device"])
```

- Creates an **EfficientNet-B7** model.
- Moves model to **CPU or GPU** based on availability.

6. Load Model Checkpoints (If Available)

```
if os.path.exists(config["initial_checkpoint_path"]):
    print(f"Existing resume checkpoint found at
{config['initial_checkpoint_path']}. Loading model weights to resume
training.")

model.load_state_dict(torch.load(config["initial_checkpoint_path"],
map_location=device, weights_only=True))
else:
    print("No checkpoint found. Starting training from scratch.")
```

- **Resumes training** from the latest checkpoint if available.
- **Loads pre-trained weights** if no resume checkpoint exists.
- **Starts fresh training** if no checkpoints are found.

7. Clear GPU Cache

```
torch.cuda.empty_cache()
```

- Frees unused GPU memory before training starts.

8. Train the Model

```
model, metrics = train_model(model, train_loader, val_loader, device,
                             num_epochs=config["num_epochs"],
max_lr=config["max_lr"])
```

- Trains the model using **OneCycleLR, label smoothing, and AMP**.
- Returns **training metrics** (loss & accuracy).

9. Test the Model

```
test_model(model, test_loader, device)
```

- Evaluates the model on the **test dataset**.

10. Save Training Metrics

```
with open("/kaggle/working/training_metrics.pkl", "wb") as f:
    pickle.dump(metrics, f)
print("Training metrics saved to training_metrics.pkl")
```

- Saves training and validation **loss & accuracy** for further analysis.

Key Features:

□ **End-to-End Workflow** – Handles data preparation, training, evaluation, and saving results.
□ **Model Checkpointing** – Prevents loss of progress by saving the best model.
□ **GPU Optimization** – Uses **AMP and CUDA** for faster and efficient training.
□ **Reproducibility** – Ensures consistent results using a **fixed random seed**.
□ **Efficient Data Processing** – Uses **multi-threaded DataLoaders** for faster data loading.

```
# -----------------------------
# 7. Main Function: Data Loading, Training, and Saving Metrics
# -----------------------------
def main():
    # Set seed for reproducibility.
    set_seed(config["seed"])

    # Load and prepare the data.
    df = load_audio_files(config["data_path"])
    labels_sorted = sorted(df['label'].unique())
    label2id = {label: i for i, label in enumerate(labels_sorted)}
    df['label_id'] = df['label'].map(label2id)
    print(f"Found {len(df)} audio files across {len(labels_sorted)}
classes.")

    train_df, val_df, test_df = split_data(df, config["test_size"],
config["val_size"], random_state=config["seed"])
    print(f"Train: {len(train_df)}, Val: {len(val_df)}, Test:
{len(test_df)}")
```

```
    train_dataset = BirdAudioDataset(train_df, config["sample_rate"],
config["duration"], augment=True)
    val_dataset = BirdAudioDataset(val_df, config["sample_rate"],
config["duration"], augment=False)
    test_dataset = BirdAudioDataset(test_df, config["sample_rate"],
config["duration"], augment=False)

    train_loader = DataLoader(train_dataset, batch_size=32,
shuffle=True, num_workers=config["num_workers"])
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,
num_workers=config["num_workers"])
    test_loader = DataLoader(test_dataset, batch_size=8,
shuffle=False, num_workers=config["num_workers"])

    num_classes = len(labels_sorted)
    model = BirdClassifier(num_classes)

    device = torch.device(config["device"])

    # ---------------
    # Check for and load checkpoints:
    # ---------------
    if os.path.exists(config["initial_checkpoint_path"]):
        print(f"Loading initial weights from
{config['initial_checkpoint_path']} for the first run.")
        checkpoint = torch.load(config["initial_checkpoint_path"],
map_location=device, weights_only=True)
        old_weight_count =
checkpoint["model.classifier.1.weight"].shape[0]
        num_new_classes = num_classes
        if num_new_classes > 0:
            print(f"Expanding classifier layer: adding
{num_new_classes} new classes.")
            current_fc_weight = model.model.classifier[1].weight.data
            current_fc_bias = model.model.classifier[1].bias.data
            # Copy weights and biases for the old classes from the
checkpoint.
            current_fc_weight[:old_weight_count, :] =
checkpoint["model.classifier.1.weight"]
            current_fc_bias[:old_weight_count] =
checkpoint["model.classifier.1.bias"]
            # Update the checkpoint with the modified classifier
weights.
            checkpoint["model.classifier.1.weight"] =
current_fc_weight
            checkpoint["model.classifier.1.bias"] = current_fc_bias
        model.load_state_dict(checkpoint, strict=False)
    else:
        print("No initial checkpoint found. Starting training from
scratch.")
        torch.cuda.empty_cache()

    # Train the model.
    model, metrics = train_model(model, train_loader, val_loader,
device,
                                 num_epochs=config["num_epochs"],
max_lr=config["max_lr"])

    # Test the model.
    test_model(model, test_loader, device)

    # Save training metrics.
    with open("/kaggle/working/training_metrics.pkl", "wb") as f:
        pickle.dump(metrics, f)
    print("Training metrics saved to training_metrics.pkl")


# Runs the full pipeline from data loading to model evaluation
if __name__ == '__main__':
    main()
```

```
Found 14221 audio files across 41 classes.
Train: 11518, Val: 1280, Test: 1423

Downloading:
"https://download.pytorch.org/models/efficientnet_b7_lukemelas-
c5b4e57e.pth" to
/root/.cache/torch/hub/checkpoints/efficientnet_b7_lukemelas-
c5b4e57e.pth
100%|██████████| 255M/255M [00:01<00:00, 183MB/s]

Loading initial weights from
/kaggle/input/bioacoustics-model-weight/best_model.pth for the first
run.
Expanding classifier layer: adding 41 new classes.



Epoch [1/1] | Train Loss: 0.9194, Train Acc: 0.9302 | Val Loss:
1.4588, Val Acc: 0.7688
Best model updated (Val Acc: 0.7688). Checkpoint saved.

Testing: 100%|██████████| 178/178 [01:57<00:00,  1.52it/s]

Test Accuracy: 0.7667
Training metrics saved to training_metrics.pkl
```

```python
# Open the pickle file in read-binary mode
with open("/kaggle/working/training_metrics.pkl", "rb") as f:
    data = pickle.load(f)

# Print or inspect the data
print(data)
```

```
{'train_loss': [0.9193847264281411], 'train_acc':
[0.9301962146205939], 'val_loss': [1.4587901130318641], 'val_acc':
[0.76875]}
```