

CIS 583, Midterm Project Documentation

Deep Audio Classifier – Capuchin Bird call Detector

Cover Page:

- **Team Members:** Harika Kanala, Pranitha Velusamy Sundararaj, Soham Saha
- **Section Number:** Team 11
- **Course:** CIS 583 – Deep Learning
- **Semester:** Winter 2025

Introduction

i. Abstract

This project uses an AI-driven system to automatically detect and detail the number of capuchin bird calls within audio recordings that were recorded in an ecologically rich rainforest. The study aims to show how AI can be effective in environments in which extraction of information could be difficult and time consuming. We go over the specific use case that has complexity and implications for a future where more datasets can be analyzed with computers and reduce the need for human effort in various fields. To better understand the data and be more efficient in the processing, we have integrated multiple signal preprocessing techniques along with the deep learning methodology. This ensures that the model is not only effective but also scalable for other applications in acoustic environments.

Our project uses Convolutional Neural Networks (CNNs), a class of deep learning models that are recognized for their pattern recognition and classifications capabilities with structured data. In this specific use case, we use spectrograms as the primary input. Spectrograms are data that represent changes in audio frequencies over time. The goal of this project is to create a CNN model that can be used to distinguish audio that contains calls from the capuchin bird as well as other background noises. The model was trained to be highly accurate and uses publicly available data. The model has also been tuned to operate in different situations which allows for further research and analysis.

This report details the project's progression through several critical stages, starting from preparation, pre-processing of the data to building the neural network architecture and

training the model. The work that is detailed in this report has the potential to be modified and used for a wide variety of audio classification tasks.

ii. Problem Statement or Objectives

The need for this project arose from limitations of traditional wildlife monitoring, which often struggles with scalability and accuracy. Traditional acoustic monitoring requires people and lots of time. This is not very effective or efficient and is prone to more human errors. It cannot be easily scaled and people cost a lot. Another issue is that skilled workers are not always available. They are limited by how much they can cover and there are loads of miscellaneous factors that could affect their progress. In addition to being prone to bias, humans can make mistakes because of their limitations in memory and experience. The goal of this project is to address all of these issues by creating an automated solution using AI.

The core objective of this project is to create a model that could automatically and accurately differentiate and enumerate distinct vocalizations of the capuchin bird against background noises in a rainforest. We need to not only detect but also differentiate the noises of the bird from other birds. This has the potential to cut down costs and increase scalability. If done correctly, there would be less human bias and more consistent results.

With regards to learning objectives, it has been the goal to analyze more about CNN's, resolve common issues with the model and develop techniques to reduce challenges of creating these kinds of environments and scaling them. Finally, we demonstrate a robust option to allow the use of enhanced and reliable data for conservation.

Background and AI Type

This project falls under **supervised learning** and uses **binary classification** techniques. In supervised learning, we use pre-labeled data that we train the model on. The most important part is that we want to be highly predictable and CNN's are very helpful for that. For this particular use case, we want to be able to determine how long the bird has been recorded and when to stop taking data for a single bird.

To ensure proper data analysis, the correct representation must be selected and there must be consideration for what data and what samples to analyze.

The other side of this project is binary classification. This technique is used to separate the positives from the negatives. Binary classifications enable the model to identify and reliably automate population monitoring to allow researchers to handle and analyze audio sample more accurately.

Tools and Technologies to be Used

1. **Programming Language:** Python
2. **Deep Learning Framework:** TensorFlow with Keras
3. **Audio Processing Libraries:** Librosa
4. **Data Download Library:** Kaggle
5. **Data Manipulation and Analysis:** NumPy, Pandas
6. **Visualization:** Matplotlib, Seaborn
7. **Development Environment:** Google Colab

The rationale for choosing these tools is as follows:

- **Python:** It is very simple, concise and easy to code with. It is also faster than most of the other available languages.
- **TensorFlow:** Tensorflow is an ideal choice for this project as it uses CNN's and machine learning. The documentation for Tensorflow is very thorough, easy to navigate and issues can be found and addressed very quickly. We can also integrate our GPU to speed up training with Tensorflow.
- **Librosa:** This library is used for a large portion of our code. It is used for handling and manipulation of audio files and allows to easy access to the data within those files.
- **Kagglehub:** We use open source network to easily download, organize and integrate our dataset within the colab.
- **Numpy & Pandas:** These are widely used libraries for conversion of data to be suitable for machine learning process.

- **Matplotlib & Seaborn:** Both of these libraries are used for plotting and visualizing out dataset for better understanding of the samples.
- **Google Colaboratory:** This is a flexible way to run and host our code. With everything running on the cloud, we do not need any local resources to train the model.

Dataset

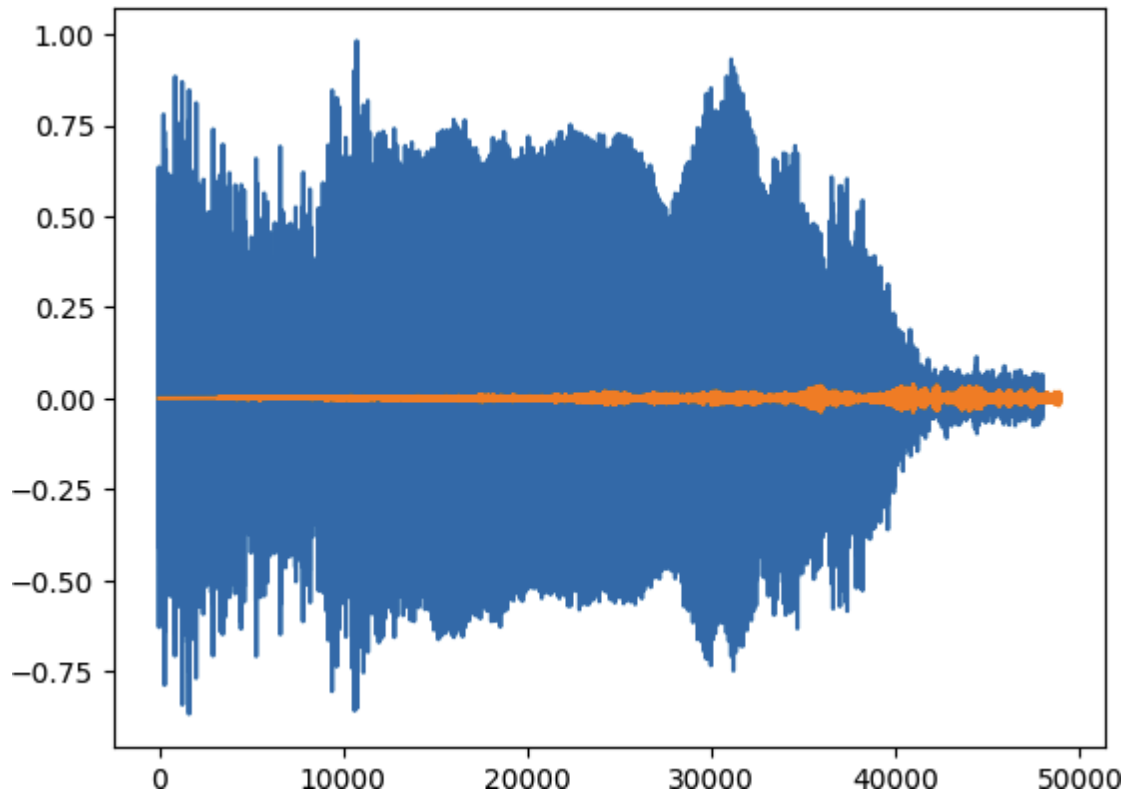
The data downloaded for this project comes directly from Kaggle. It was used due to its wide accessibility and open-source license policy. The quality of the data was well documented was provided a good base to build and test the model. The structure of the data has two parts – a training set of positives and negatives, and another recording for data extraction. The first is described below:

- **Parsed_Capuchinbird_Clips:** These clips contain isolated audio of only the capuchin bird. The sample was carefully selected to be very easy to measure.
- **Parsed_Not_Capuchinbird_Clips:** These clips contain different samples with varying sounds, to increase the depth and range of audio that can be listened.
- **Forest Recordings:** This is the actual data that the trained model is supposed to listen to. It contains a dataset of recordings made in the area.

i. Dataset Description

1. **Format:** To store and play the audio, we use Wav and MP3.
2. **Size:** The parsed data has several hundreds of samples of data.
3. **Acoustic Qualities:** It would have been preferable to have more data especially clean sets of isolated clips and other ambient sounds.

Example of a Capuchin Bird Clip and Non-Capuchin Bird Clip waveform:



X-axis (Horizontal Axis):

1. Represents **time** in samples.
2. The range is from 0 to 50000 samples.
 - Since the audio is sampled at 16 kHz (16,000 samples per second), the total duration of the audio is approximately:

$$\text{Duration} = \frac{50000 \text{ samples}}{16000 \text{ samples/second}} \approx 3.125 \text{ seconds}$$

Y-axis (Vertical Axis):

3. Represents the **amplitude** of the audio signal.
4. The amplitude ranges from -1.0 to 1.0 (normalized audio signal).
Positive values indicate positive pressure (compression), and negative values indicate negative pressure (rarefaction).

By comparing the two waveforms, you can visually identify differences in the structure and timing of the sounds.

The Capuchin call is likely more structured and periodic, while the non-Capuchin sound may be more random or noisy.

ii. Dataset Acquisition Challenges

The challenges faced in obtaining and preparing audio data for a machine learning task focused on classifying Capuchin bird calls versus non-Capuchin sounds are:

- The dataset may have significantly more samples of one class (e.g., non-Capuchin sounds) compared to the other (e.g., Capuchin calls), leading to biased model performance.
- Processing and analyzing large audio datasets require significant computational resources

Model Overview

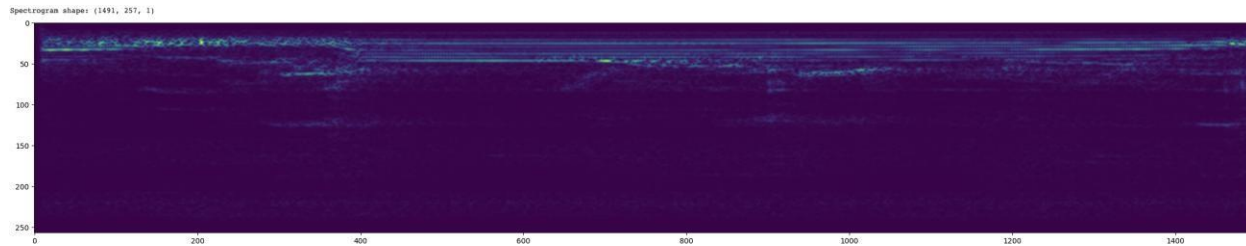
i. Data Loading and Preprocessing:

The project begins with downloading the dataset from Kaggle. The next step is to take the raw audio data and turn it into something that the machine learning model can understand and take as an input. This is the function **'load_wav_16k_mono'**. This function uses the librosa library to load the standard wav data and then perform operations to preprocess this. There are a number of samples and sized and in order to train well, we must standardize the data.

Next, we handle the structured data using Tensorflow so that the model can interpret and use for testing. The labels are either positive or negative. By carefully classifying the data, we allow the code to determine the effectiveness. We need to keep in mind that there will always be some conversion that is necessary to make a functional mode. This component also handles cleaning up the data with audio to keep the set consistent.

Then, we focus on preprocessing to enable the use of Spectrograms. We use the code to clean the data from noise and distortion, to ensure that the audio quality remains consistent and is not affected by noise from the raw data stream. The features that we extract are key for running and testing a successful audio model. Finally, we use some resizing and reshaping tools. After the signal processing, we need to standardize the

data so that it has the same dimensions and structure for the model. By calling these functions, the audio becomes clean and is ready for further work.



Spectrogram shape: (1491, 257, 1)

5. **X-axis (Horizontal Axis):**
 - Represents **time** in seconds or samples.
 - The range depends on the duration of the audio clip and the parameters used for the Short-Time Fourier Transform (STFT).
 - In your case, the audio is truncated or padded to 48,000 samples (3 seconds at 16 kHz).
6. **Y-axis (Vertical Axis):**
 - Represents **frequency** in Hertz (Hz).
 - The range depends on the sample rate and the STFT parameters.
 - For a sample rate of 16 kHz, the frequency range is typically from 0 Hz to 8 kHz (Nyquist frequency).
7. **Color Intensity (Z-axis):**
 - Represents the **magnitude** (amplitude) of the frequency components at each time-frequency bin.
 - Brighter colors (e.g., yellow) indicate higher magnitudes, while darker colors (e.g., blue) indicate lower magnitudes.
 - If the audio is a **Capuchin bird call**, the spectrogram might show:
 - Bright horizontal bands corresponding to the bird's vocal frequencies.
 - Periodic patterns reflecting the repetitive nature of the call.
 - If the audio is **background noise**, the spectrogram might show:
 - Random or scattered points with no clear patterns.
 - Lower overall intensity (darker colors).
8. If the audio is a **Capuchin bird call**, the spectrogram might show:
 - Bright horizontal bands corresponding to the bird's vocal frequencies.

- Periodic patterns reflecting the repetitive nature of the call.
9. If the audio is **background noise**, the spectrogram might show:
- Random or scattered points with no clear patterns.
 - Lower overall intensity (darker colors).

Challenges:

Issue 1:

Having trouble with TensorFlow I/O (tfio) not working properly, specifically getting a `NotImplementedError` related to the `libtensorflow_io.so` library. TensorFlow I/O often needs to match the exact version of TensorFlow. avoid using `tfio`, replace that part with a different resampling method, like using `librosa`, which is a Python library for audio processing, that might bypass the dependency on `tfio`. However, since TensorFlow operations need to be graph-compatible, using `librosa` directly in a TensorFlow pipeline might require wrapping it in a `tf.py_function` to integrate with the TensorFlow graph.

Issue 2:

The `input_shape` parameter in the first `Conv2D` layer is not matching the shape of the data being fed into the model.

Fix: The spectrogram generated by the `preprocess` function has a shape of `(time_steps, frequency_bins, 1)`. We need to ensure that:

- The `input_shape` in the model matches this shape.
- The data pipeline is correctly feeding the data into the model.

Issue 3:

The model has two `Conv2D` layers followed by a `Flatten` layer and two `Dense` layers. The `Flatten` layer is converting a 3D tensor into a 1D vector, which results in a very large number of parameters when connected to the `Dense` layers. For example, flattening a `(1487, 253, 16)` tensor gives $1487 \times 253 \times 16 =$ around 6 million elements. Then, connecting that to a `Dense` layer with 128 units would result in $6 \text{ million} \times 128 =$ around 770 million parameters. That's way too much.

So, to reduce the number of parameters, the user needs to modify the architecture. Maybe adding a pooling layer before flattening to reduce the spatial dimensions. Global Average Pooling or Max Pooling could help.

ii. Model Architecture and Training:

```
# 7.1 Load Tensorflow Dependencies
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, GlobalMaxPooling2D
from tensorflow.keras.metrics import Recall, Precision

# 7.2 Build Sequential Model, Compile, and View Summary
# CHANGED: Added regularization and dropout
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(1491,257,1),
                           kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GlobalMaxPooling2D(),
    tf.keras.layers.Dense(128, activation='relu',
                           kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
# CHANGED: Added gradient clipping and callbacks
optimizer = tf.keras.optimizers.Adam(
    learning_rate=0.001,
    clipnorm=1.0 # Gradient clipping
)

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_lr=1e-6
)

model.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=['accuracy',
             tf.keras.metrics.Precision(name='precision'),
             tf.keras.metrics.Recall(name='recall')]
)
```

We use a CNN model built using Tensorflow. We start with the convolutional layers. These are building blocks that extract the hierarchical features directly from the spectrogram of the audio data. By convolving filters across the spectrogram input, the layers can identify patterns within in. This initial processing will help the model to be more accurate in the final stages. Each convolution is paired with ReLU activation function. ReLU is essential of introducing non-linearity in the model. This allows the model to understand more complex relationships that would be impossible for a linear model. It also helps the model to recognize sounds that are as small as a few seconds.

The neural network takes as input a **mono audio spectrogram** represented as a 2D tensor with shape (1491, 257, 1), where 1491 likely corresponds to time steps (frames), 257 to frequency bins, and 1 channel for grayscale-like intensity values. The model uses two **Conv2D layers** (16 filters each, 3x3 kernels, ReLU activation) to extract spatial patterns (e.g., time-frequency features like call harmonics or temporal structures) from the spectrogram. These are followed by a **GlobalMaxPooling2D layer** to condense the most salient features across time and frequency into a 1D vector. A **Dense layer** (128 units, ReLU) then processes these features, and a final **Dense layer** with sigmoid activation outputs a probability score (0 to 1) indicating the presence of a Capuchin call in the input audio slice.

To get better efficiency and stability, the global max pooling is used. This allows for more rapid running time and less processing needed by the local hardware. With decrease in parameters, the risk of overfitting decreases as well. Next we have fully connected dense layers to help with data analysis and ensure that everything is correctly labelled and passed in.

In the final layer, we utilize the Sigmoid function to compress the output into one final figure to classify it. This has a large number of uses and will help determine how effective the current model is. In terms of loss, it takes it from a variety of different metrics to ensure the model is well trained and free from any overfitting or local minimums. The final model summary is shown below -

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 1489, 255, 16)	160
conv2d_1 (Conv2D)	(None, 1487, 253, 16)	2,320
global_max_pooling2d (GlobalMaxPooling2D)	(None, 16)	0
dense (Dense)	(None, 128)	2,176
dense_1 (Dense)	(None, 1)	129

Total params: 4,785 (18.69 KB)

Trainable params: 4,785 (18.69 KB)

Non-trainable params: 0 (0.00 B)

Now, let's talk about how we train the model, the loss function, the optimizer, and how we evaluate performance. Since this is a binary classification problem, we use binary cross-entropy. It penalizes incorrect predictions, making the model more confident in its classifications.

The optimizer we use is Adam. Adam combines the advantages of Momentum and RMSprop to provide a faster, more stable learning process. It adapts the learning rate dynamically to optimize convergence.

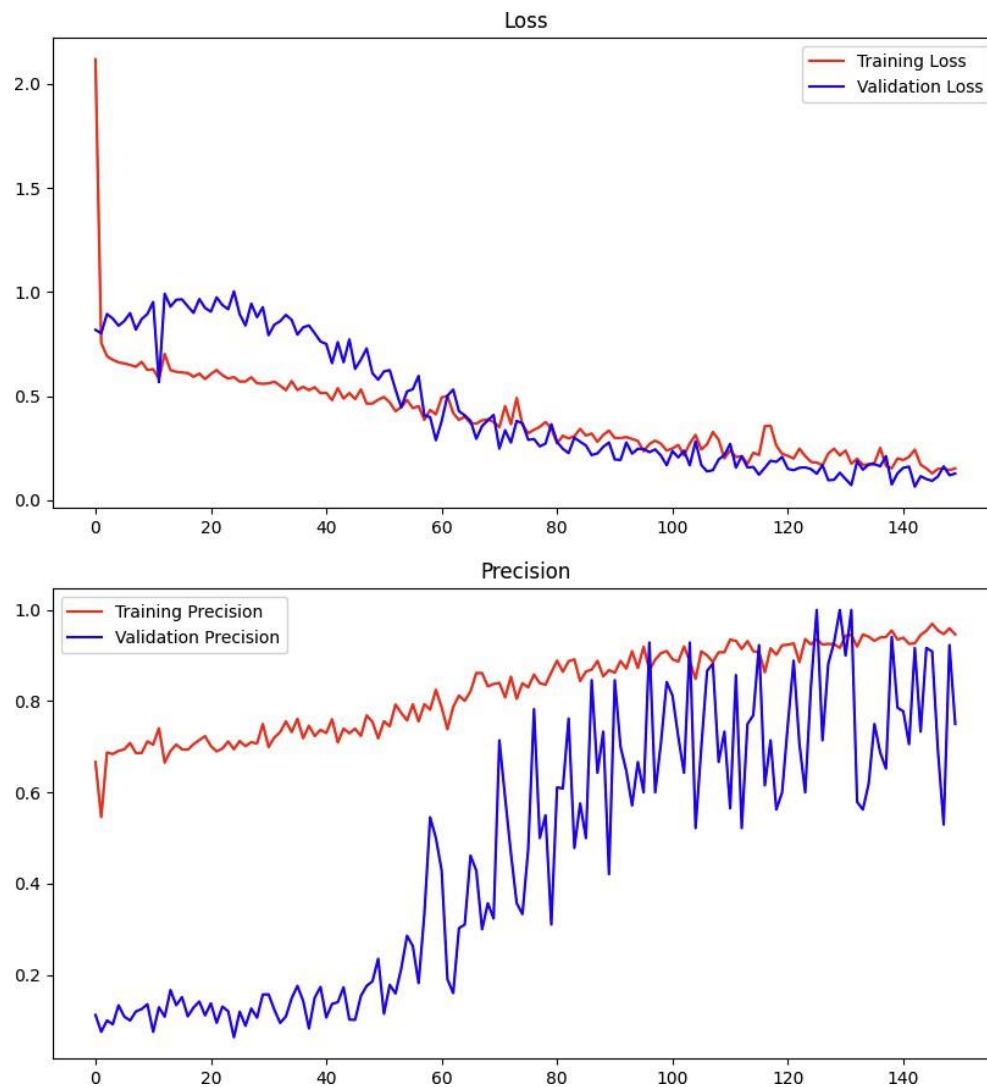
The evaluation metrics we use are -

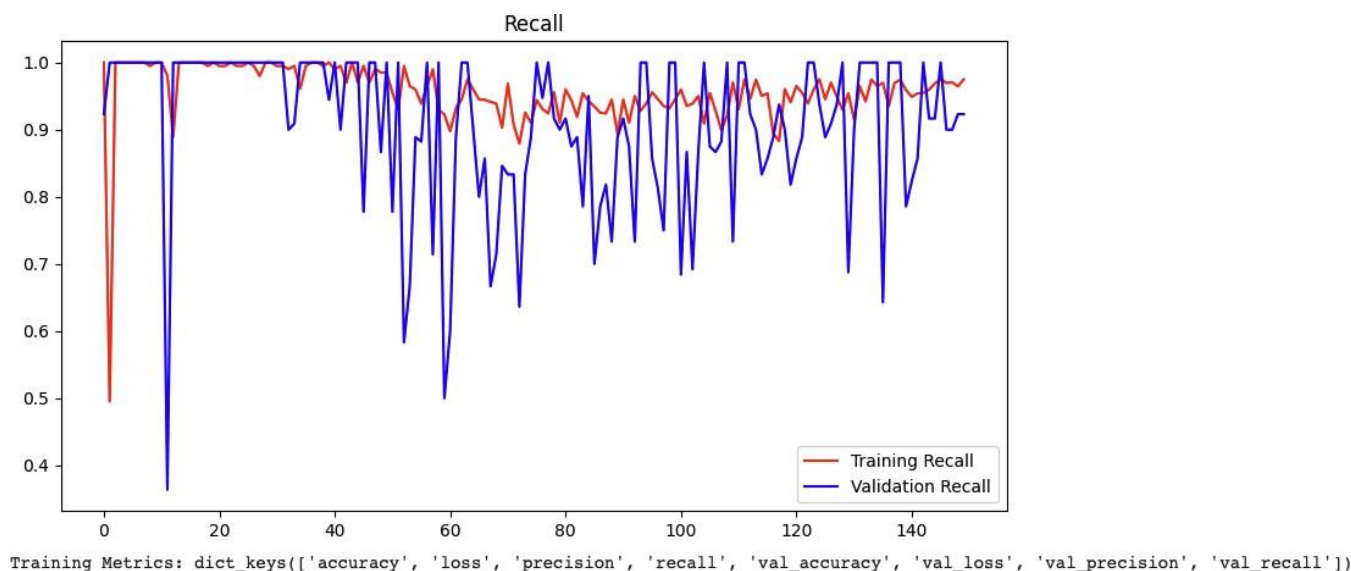
- i. Accuracy: Measures overall correctness.

- ii. Precision: Measures how many of the predicted positives were actually correct.
- iii. Recall: Measures how many actual positives were correctly predicted.

Precision and Recall are critical for imbalanced datasets, where misclassification can have a higher cost.

Observations from training the model:





First, looking at the early epochs (1-50), the validation accuracy starts really low, around 20%, and fluctuates a lot. The precision is also very low, sometimes even dropping below 0.1. That's a red flag. It seems like the model isn't generalizing well at all initially. The recall is consistently high, though, which makes me think the model is predicting almost everything as the positive class (Capuchin calls), leading to a high number of false positives and thus low precision.

Moving into the middle epochs (50-100), there's some improvement. Validation accuracy starts to climb, reaching around 50-70%. Precision is still low but better than before. However, the recall is still high, so the model is still biased towards predicting positives. The fluctuations in validation loss suggest instability, possibly due to a high learning rate or noisy data.

In the later epochs (100-150), things get more interesting. Validation accuracy jumps up to around 95-100%, and precision improves significantly, hitting values like 0.75 to 1.0. This indicates the model has started to learn meaningful patterns and is better at distinguishing between classes. The recall remains high, which is good, but there's a balance now between precision and recall. The validation loss stabilizes, showing the model is converging.

The model shows significant improvement over time, achieving high validation accuracy and balanced precision-recall by the end of training. However, initial instability and class imbalance issues suggest that further optimizations like data augmentation, class weighting, or learning rate adjustments could enhance performance. Overall, the model

is effective but could benefit from fine-tuning to achieve even better precision without sacrificing recall.

Post training results:

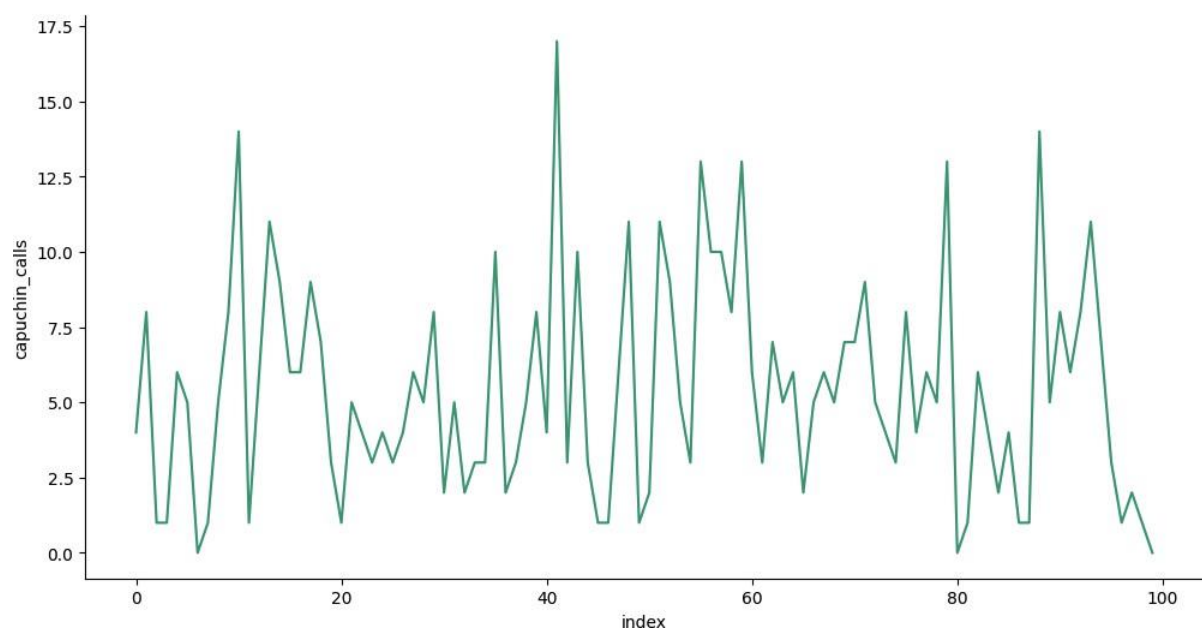
It begins by looping over all audio files in the 'Forest Recordings' directory. Each file is loaded using a function called `load_mp3_16k_mono`, which likely converts the audio to a 16kHz mono format. The audio is then sliced into segments of 48000 samples each using `timeseries_dataset_from_array`. This function creates batches of audio slices with a specified sequence length and stride, ensuring the data is in a suitable format for further processing.

Next, the code preprocesses each audio slice using a function called `preprocess_mp3`. The preprocessed slices are batched into groups of 64, and a machine learning model is used to predict whether each slice contains a capuchin call. These predictions are stored in a dictionary called `results`. The predictions, which are initially in the form of logits, are then converted into binary classes (0 or 1) based on a threshold of 0.5. If a prediction exceeds 0.5, it is classified as a capuchin call (1); otherwise, it is classified as not containing a call (0). These binary classifications are stored in another dictionary called `class_preds`.

The code then groups consecutive detections of capuchin calls and sums them up for each file. This is achieved using the `groupby` function from the `itertools` module and `reduce_sum` from TensorFlow. The results of this grouping are stored in a dictionary called `postprocessed`. Finally, the `postprocessed` dictionary is converted into a pandas DataFrame with columns 'recording' and 'capuchin_calls'. This DataFrame is displayed, providing a clear summary of the number of capuchin calls detected in each recording. This workflow automates the detection and counting of specific sounds in large sets of audio recordings, which is particularly useful in ecological and behavioral studies. By preprocessing the audio, making predictions, converting these predictions into binary classes, and grouping consecutive detections, the code efficiently summarizes the presence of capuchin calls in each recording, facilitating further analysis and interpretation.

	recording	capuchin_calls
0	recording_64.mp3	4
1	recording_47.mp3	8
2	recording_79.mp3	1
3	recording_50.mp3	1
4	recording_70.mp3	6
...
95	recording_19.mp3	3
96	recording_24.mp3	1
97	recording_58.mp3	2
98	recording_83.mp3	1
99	recording_66.mp3	0

100 rows x 2 columns



Changes/ Updates made for final term:

By looking at the results, we see that we get a training accuracy of around 96% while the validation precision is around 75%. This is a clear indication of overfitting. The model learns the training data too well but fails to generalize – likely due to lack of regularization, small batch size and no adaptive learning control. The lack of a stopping mechanism also means that we are probably training well past the optimal validation performance. We applied a few changes to the code to build a more robust model that doesn't just memorize but also learns meaningful patterns in the data.

i. Learning Rate Scheduling and Early Stopping

The current model uses an Adam optimizer with a default learning rate of 0.001. While Adam adapts learning rates internally per parameter, it doesn't adjust the global learning rate based on training progress. To address this, we introduce a learning rate scheduler from TensorFlow callbacks called ReduceLROnPlateau. When the validation loss stagnates for a few epochs, the learning rate will be reduced by a factor. This allows the optimizer to make fine updates as we get closer to the local minima.

We also use Early Stopping to monitor the validation loss and stop training if no improvement is seen for a number of epochs (set to 5). This avoids overfitting.

ii. Increasing Batch Size and Adding Regularization

We currently use a batch size of 8 which is quite small and can result in noisy gradient estimates, leading to unstable updates. Increasing the batch size to 32 or 64 can reduce variance and lead to smoother convergence. We start by adding L2 regularization to the kernel weights in the Conv2D layers, thereby penalizing larger weights and promote simple models. This is done by adding the regularization term to the loss function.


Next, we add a dropout function after key convolutional or dense layers to randomly deactivate neurons during training. This helps in reducing co-adaption and overfitting.

iii. Gradient Clipping

To further stabilize our training, especially if we increase model complexity or depth, we introduce gradient clipping in the optimizer. This prevents the gradients from becoming too large during backpropagation which would otherwise lead to exploding gradients and numerical instability. The clipnorm function ensures that the global norm of the gradient

vector is bounded, allowing for more stable and controlled updates.

The below images shows the updated model when compared to the previous one –

Aspect	Midterm (Baseline)	Final (Improved)
Class balancing	Not used	class_weight based on inverse class freq
Regularization	None	L2 regularization + dropout
Learning rate management	Static	ReduceLROnPlateau (adaptive decay)
Gradient clipping	None	Prevents exploding gradients
Batch size	8 (small, slow learning)	32 (better gradients, faster learning)
Shuffle buffer	100	1000 (more diverse batches)
Early stopping	None	Prevents overfitting
Data pipeline	Manually split	Automatically detects and reshuffles if needed
Result: Training Accuracy	90%+ from epoch 1 (very suspicious)	Gradual rise from 50% → 90% over 10–15 epochs
Result: Validation Precision	Near zero for long time	Starts at 0, but improves over epochs (class weighting) 
Result: Overfitting	Yes, train » val performance	Controlled and balanced
Model convergence behavior	Erratic, fluctuating recall/precision	Smooth, adaptive training

Conclusion

i. Project Impact

This project demonstrates the effectiveness of using an AI model to analyze and classify audio samples particularly for wildlife monitoring. By detecting capuchin calls in rainforest recordings, this project has contributed to conservation efforts by using an automated method that reduces human error and effort. Having the ability to analyze audio samples

means that researchers can quantify species, assess changes in environment and track population trends. The techniques used in the project such as spectrogram conversion, CNN's and binary classification have a broader impact in other fields such as speech recognition, acoustics etc.

iii. Future improvements

- Switch to a pre-trained model – By using a pre-trained model e.g., YAMNet whose weights are already computed, the training can be made faster and more accurate. These models are trained on large datasets and have learned general purpose features which makes it easier to update and modify.
- Generate synthetic data for minority class – Another method for tackling the problem of class imbalance is by generating synthetic/ artificial data for the minority class. By adding more examples for the class with the lesser number of data, the model gets rid of biases and can better distinguish/ classify.

Summary

i. Reflection on the Project

The process of converting audio data into spectrograms and applying CNN's is fascinating. It was super helpful to see how visualizing audio samples could have an impact on accuracy. One of the most challenging parts was the pre-processing, fine-tuning and applying the threshold techniques. Having to allocate proper memory due to the computational intensity of the training was also time consuming. However, I am proud of how the project has turned out and the impact it would have on the classification of capuchin calls.

ii. Project Overview

This project explores the application of deep learning models for audio classification by detecting and classifying capuchin calls in samples of rainforest recordings. Using

python and Tensorflow, we converted audio waveforms into spectrograms, visualized them using matplotlib and built a convolutional neural network model to classify sounds.

Team Name and Responsibilities of Each Team Member

1. **Team Name:** Team 11

2. **Responsibilities:**

- Soham Saha - Downloading the dataset, standardizing and preprocessing it for training.
- Pranitha Velusamy Sundararaj - Building model, compiling and training.
- Harika Kanala – Testing and Evaluating results.

Overall, the project work was much easier because we were a group of three, and we collaborated seamlessly. Each member contributed their strengths, shared responsibilities, and communicated effectively, which helped prevent any issues from arising. This teamwork made the process smoother and more efficient compared to working alone.