Convolutional Neural Networks for the Classification of Pseudorca crassidens Passive Acoustic Data

University of St Andrews

Francis Joshua Arrabaca

Supervisors: Dr Chrissy Fell and Dr Hannah Worthington

Master of Science in Data-Intensive Analysis

School of Mathematics and Statistics

University of St Andrews

August 2024

Table of Contents

List of Figures

List of Tables

Code Listings

Declaration of Authorship

I, Francis Joshua Arrabaca, hereby declare that this dissertation has been done for the degree in Msc Data-Intensive Analysis at the University of St Andrews for the academic term 2023–2024, and has not been submitted for any other degree or professional qualification.

I confirm that this dissertation has been composed by myself and was done entirely on my own, save for programming code references which have been duly acknowledged, and published work duly attributed.

Acknowledgements

My deepest gratitude to my supervisors Dr Chrissy Fell and Dr Hannah Worthington of the School of Mathematics and Statistics for their extensive guidance, support and feedback. Many thanks also to the Dolphin Acoustics Vertically Integrated Project, their project head Dr Julie Oswald, and especially to Emily McCloskey for her insights in Hawaiian False Killer Whales, audio data collection and curation process.

To my wife, Mariae, and my family back home in the Philippines who have patiently supported and encouraged me from afar, I express my heartfelt love and appreciation.

Finally, to the Chevening Scholarship Program and the University of St Andrews for the opportunity to study at one of the top universities in the world, as well as to my friends, mentors and colleagues who have supported me throughout this journey, my sincerest gratitude.

Date: 13 August 2024

Abstract

This study involves programming experiments using Convolutional Neural Networks (CNNs) on passive acoustic data from Hawaiian False Killer Whales, and would be of interest to entry-level statisticians and computer scientists looking to study CNNs, and apply them in the field of biacoustics.

Bioacoustic classification involves manual and statistical methods of identifying animal groups using audio data. This is useful when visual or geospatial classification methods may be lacking or unfeasible. In some cases, only audio data is provided to researchers, but differences in sound may not be easily perceptible to humans, so machine learning methods are required.

This study uses bioacoustic classification to answer the question: is it possible to identify between two genetically distinct groups of False Killer Whales (*Insular* and *Pelagic*) using audio recordings alone? In answering this question, the audio data was first converted into spectrogram image data to be used in CNNs. Transfer Learning was then applied which allowed the use of pre-trained CNNs, eliminating the need to create neural networks from scratch.

The result shows that using rudimentary CNN transfer learning, it is possible to distinguish between *Insular* and *Pelagic* False Killer Whales with an accuracy of 93.89%.

Keywords— Bioacoustics - Convolutional Neural Network - Transfer Learning

1 | Introduction

This study discusses the use of Convolutional Neural Networks (CNNs) on marine mammal acoustic data, and has been written for audiences in post-graduate statistics and computer science programs. Readers are assumed to be familiar with data analysis and introductory machine learning, but no assumption is made for specific knowledge in marine bioacoustics or CNNs.

As such, statistical and machine learning concepts such as classifier models and confusion matrices are mentioned but not explained, while other concepts such as False Killer Whales, marine audio surveys, spectrograms, CNNs and Transfer Learning are discussed more thoroughly.

1.1 Pseudorca crassidens

Figure 1.1: An illustration of an adult *Pseudorca crassidens* (False Killer Whale) by Dewynter (2019) for the CARI'MAM project.

Pseudorca crassidens or False Killer Whales are large oceanic dolphins that inhabit warm waters around the world, mostly in the open ocean (*Pelagic*), with some groups observed close to shore (*Insular*)(Baird (2009)). These animals grow to 5–6 metres in length, and are black or dark grey in colour as shown in Figure 1.1.

False Killer Whales tend to group in pods of 20 to 100 individuals (Ridgway & Harrison (1999)), and may associate with other marine mammals such as bottlenose dolphins. Their sounds include whistles around 5KHz lasting 0.5 seconds and echolocation clicks around 40KHz of 0.03 seconds (DOSITS (2021)).

In the wild, False Killer Whales are of interest as there have been unfortunate interactions with humans in the commercial and recreational fishing industries. These animals feed on the same fish species targeted by commercial fisheries (such as tunas) and have been found stealing fish from these fishing operations (Baird (2009)). Considered a nuisance, False Killer Whales have been killed directly or as bycatch by these fishing operators (*Pseudorca crassidens: Baird, R.W.: The IUCN Red List of Threatened Species 2018: e.T18596A145357488* (2018)).

1.2 HICEAS 2017 and False Killer Whales around the Hawaiian Islands

The Hawaiian Islands Cetacean and Ecosystem Assessment Survey (HICEAS) 2017 was a large-scale survey of marine mammals and seabirds around the waters of the Hawaiian islands (Yano et al. (2018)), including the extensive waters of the Papahānaumokuākea Marine National Monument (PMNM) to the northwest of Hawaii. The survey range can be seen in Figure 1.2. The 2017 survey took place from July to December, with similar surveys conducted in 2002, 2010, and most recently in 2023 (NOAA Fisheries (2024)). The survey aimed to collect data in order to estimate the abundance and distribution of the animals around the Hawaiian Islands.

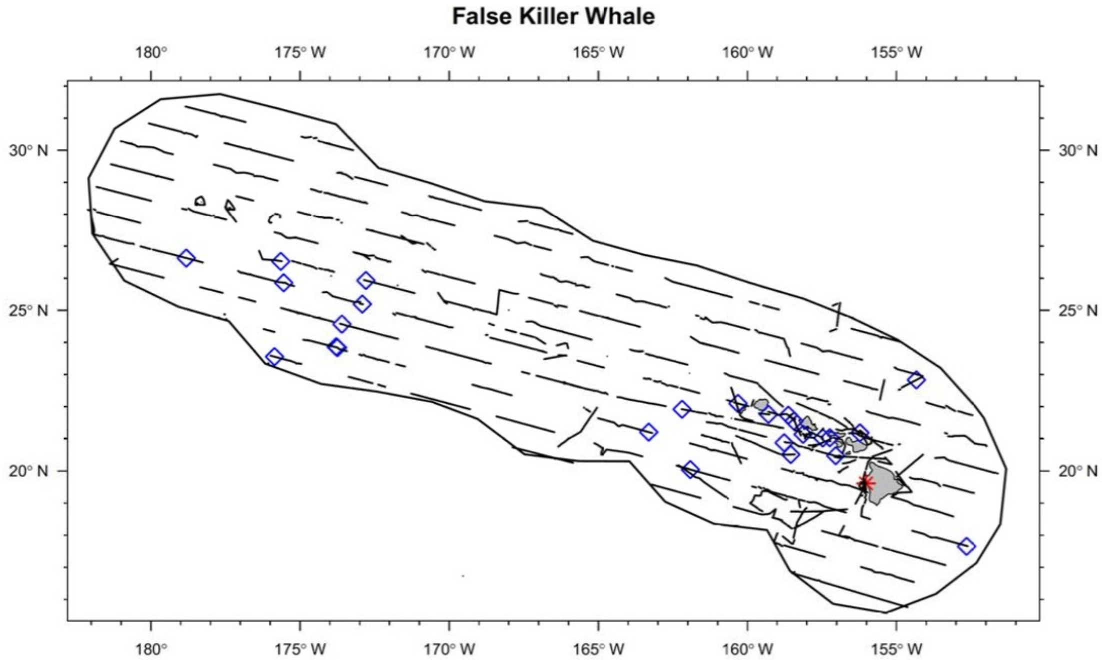


Figure 1.2: A map by Yano et al. (2018) showing the survey area and False Killer Whale sightings. The survey includes the waters immediately surrounding the Hawaiian Islands in the lower right, but also extends further out to the PMNM area to the upper left. The black diagonal lines show the visual effort which coincide with the transects of the ship journeys. The red asterisks show visual detection only of False Killer Whales, while the blue diamonds indicate both acoustic and visual detections.

The survey involved the two research ships *Oscar Elton Sette* and *Reuben Lasker* independently travelling along transects (equidistant, fixed and straight paths) in the survey area for four weeks at a time, with 46 scientists assigned between the two ships. The survey effort involved both visual and audio assessments of *cetaceans* (whales and dolphins) and seabirds. Aside from the visual and audio survey, biopsy samples were collected and satellite tags deployed on cetaceans.

During the survey, three groups of False Killer Whales were observed: *Insular*, which inhabit the waters between and near the main Hawaiian islands; *Northwestern*, which live mostly in the PMNM area; and *Pelagic*, which live in the open ocean waters near the main Hawaiian islands and the PMNM (Carretta et al. (2022)). While all three groups have been found to be demographically distinct through genetic, telemetry, and photo-identification studies, their geographic ranges overlap each other around the Kauai and Niihau islands, which are the Northwestern-most main Hawaiian islands.

This study focuses on *Insular* and *Pelagic* groups only. As of 2015, the abundance estimate for the Hawaiian *Insular* stock was 149 animals, while the *Pelagic* stock was 928 animals (Carretta et al. (2022)). In other words, there are substantially fewer *Insular* animals than *Pelagic*, and out of 1,077 animal counts, it is expected that 13.8% would be *Insular*, while 86.2% would be *Pelagic*.

1.3 Audio Data Collection and Curation

The original audio data for this study was taken from HICEAS 2017 was created by recording audio using towed hydrophone arrays while the ships were running along transects during the survey (Yano et al. (2018)). These files were then curated by the University of St Andrews - Dolphin Acoustics Vertically Integrated Project (VIP) team (McCloskey (2024)).

Only audio files that had been positively identified as having *Pseudorca* sounds with no other animal species present were included. The audio files were then split into 1-second clips for the VIP students to trace the whistles using PAMGuard and ROCCA, which are software used for passive acoustic data processing and species identification.

The dataset provided comprised 3 main folders: *Pelagic* with 2,842 audio files; *Insular* with 439 files, and *Unknown* with 4,637 files. This last category contained audio files that came from either group, but could not be positively confirmed. No audio files were provided for the *Northwestern* group.

Comparing the two known groups, audio data from the *Pelagic* group comprised 86.7% of all the tagged data, with *Insular* only accounting for 13.4% of the dataset. This imbalance is to be expected since *Pelagic* False Killer Whales comprise 86.2% of animal counts between the two groups, as discussed in Section 1.2.

Subfolders in each group were descriptively named after the ship or leg journey (*Sette* or *Lasker*), and the audio file were named after the date and time of the recording.

The audio files contained *Pseudorca* whistles and clicks, but also captured ambient audio such as boat engine noise and ocean wave sounds. The files varied from one channel to six audio channels, and ranged from around 0.9Mb to 3.0Mb in size (with a few files larger than 7.0Mb), totalling around 16 gigabytes.

For this study, only the identified groups *Pelagic* and *Insular* were used during model training, while the *Unknown* set was used for the prediction simulation.

1.4 Audio Visualisation Using Spectrograms

Due to the large size of the audio files, it was necessary to convert the data from audio to some form of sound visualisation to make the data more manageable.

Sound is perceived when a mechanical action moves through physical matter as vibrations (Bradley & Stern (2008)), and can be described using several terms: *Wavelength* describes the distance of each *wave* (or vibration) from each other; *Frequency* (f) is the number of repetitions (or vibrations) that occur per period (usually one second) and is described with *Hertz* (Hz); and *Amplitude* describes the magnitude or strength of the vibrations.

High amplitude does not necessarily mean high frequency in the audio, but refers rather to the strength of the individual waves. A high frequency file and low frequency file could therefore have the same amplitude, as shown in Figure 1.3.

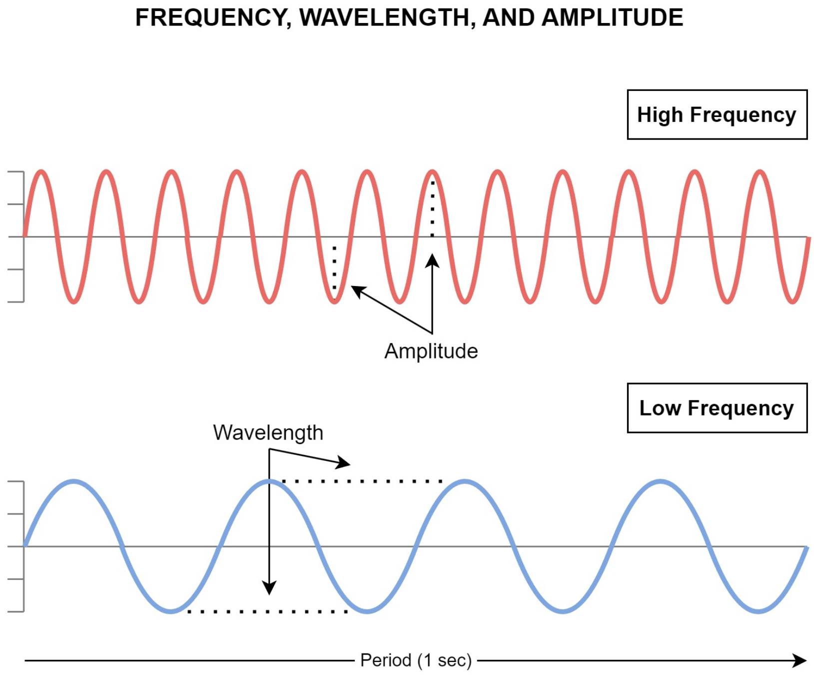
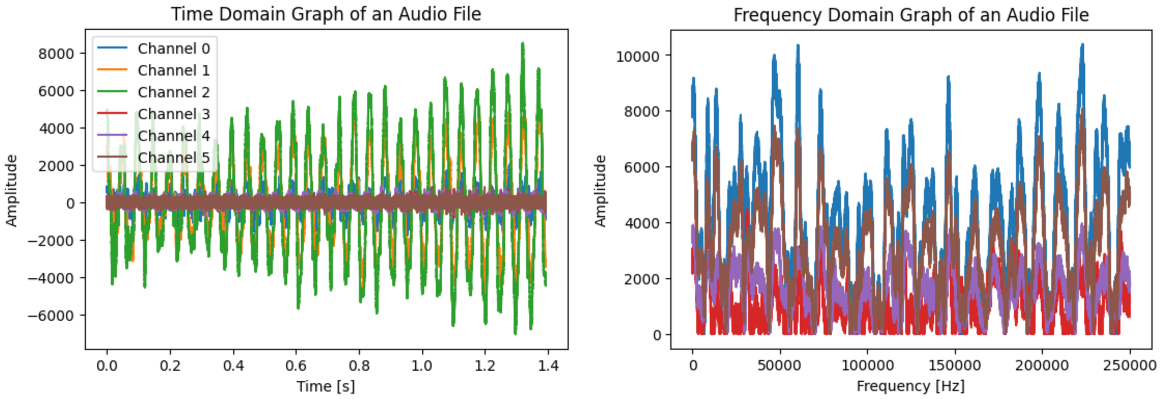


Figure 1.3: A simple diagram showing the relationship between frequency, wavelength and amplitude.

There are several ways to visualise sound; a straightforward way would be to plot the combined amplitude of its frequencies over time as a *Time domain graph* (Abbot (2024)) as in Figure 1.4 (a). Although this shows the strength of the sound over time, it leaves out the intricate details of the sound's frequencies.

Another way would be to show the frequencies in a *Frequency domain graph* as in Figure 1.4 (b). This graph shows the sound's frequencies for a certain period, and it is possible to have several different frequencies on one graph. However, this graph is not able to show the time period associated with the sound.

A different form of visualisation is therefore needed to show all three dimensions of frequency, amplitude, and time.



(a) Time Domain Graph (b) Frequency Domain Graph

Figure 1.4: An audio file from the *Pelagic* class represented in a time domain *(left)* and the same file as a frequency domain *(right)*. The left graph shows the audio amplitude of the signal across time, while the right graph shows the amplitude for the different audio frequencies. The different colours represent the audio channels in the file.

A Spectrogram is a combination of a time domain graph and a frequency domain graph in that it shows frequencies present and their strength over the course of time. Spectrograms are created using a process involving *the Short-Time Fourier Transform* (SFFT). This process takes in the amplitudes from the time domain and cuts the data into equal segments across the time period. It then computes the Discrete Fourier Transform per segment to show the magnitude of the different frequencies for that segment, and applies a *window* function to smooth the edges of each segment in the entire spectrogram (Oppenheim et al. (1998)).

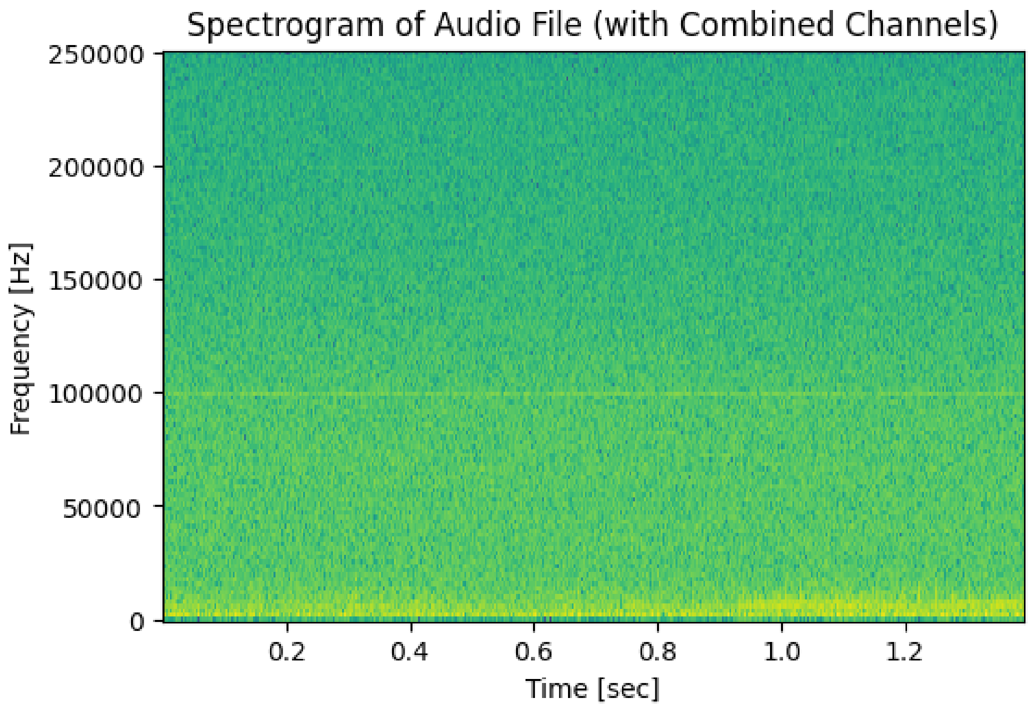


Figure 1.5: The audio file in Figure 1.4 now shown in a spectrogram. Unlike the previous plots, this plot shows the audio frequencies, amplitude and time in one graphical representation. While False Killer Whale whistles should be seen at around 5,000Hz and clicks at around 40,000Hz, this spectrogram shows high amplitude at 10,000Hz throughout the period, as well as around 200Hz starting around 1.0 sec. This might be indicative of other sounds such as engine and wave noises picked up by the hydrophone recorders.

A spectrogram can then be read as a series of frequencies on the y-axis with the amplitude for the different frequencies represented by a colour scale, shown over time on the x-axis.

1.5 Motivation for Study

The HICEAS survey is part of NOAA's mandate to monitor the population and habitats of marine mammals around Hawaii (NOAA Fisheries (2024)). These surveys are extremely time-intensive (up to five months continually at sea) and labour-intensive (at least 40 science officers are required excluding the ships' crew) (Yano et al. (2018)).

Accurate identification of the False Killer Whales around the Hawaiian waters is important to properly monitor the state of the groups related to the HICEAS mandate. This information in turn would be of interest to the United States Navy which conducts naval training in the area, as well as the US Bureau of Ocean Energy Management which assesses future sites for wind-warm development (Yano et al. (2018)).

If the False Killer Whale groups can be classified using audio files only, then labour-intensive recording methods could substantially be reduced, and instead other more cost-effective monitoring tools could be used such as bottom-mounted autonomous recorders, which can be deployed for years at a time (Sousa-Lima (2013)), and autonomous gliders, which are extremely energy-efficient and require fewer deployment and recovery intervals (National Oceanography Centre (2024)). Therefore, future data collection could be less time-consuming, less expensive, and less labour-intensive.

2 | Literature Review

2.1 Machine Learning for Acoustic Data

Prior to machine learning techniques, a variety of manual and computer-aided methods were used for identifying and classifying acoustic data. These include the unaided listening and identifying of audio recordings (Gerhard (2003)) and inspecting spectrograms or other visualisations (Digby et al. (2013)). However, these methods may introduce some human bias by the observer, or may not be feasible for very large datasets (Honig & Schackwitz (2023)). These shortcomings are addressed with machine learning.

Machine learning for audio data typically start with some form of feature extraction. This involves transforming audio data into some vector data which still represent the audio, but in a smaller format (Giannakopoulos & Pikrakis (2014)). Data is usually converted into Spectrograms, Linear predictive coding (LPCs), or Mel-Feature Cepstral Coefficients (MFCCs); LPCs use a linear function on the audio signal (Bradbury (2000)) while MFCCs make use of the Fourier Transform, but with log transform on the mel (or melody-based) scale (Bäckström et al. (2022)). Both LPCs and MFCCs are most used for applications involving human speech because they describe features approximating the human voice, but they also have wider applications in bioacoustics and acoustics in general.

Several studies have successfully made use of MFCCs and different statistical methods for general audio classification. Ma et al. (2006) showed that a Hidden Markov Model (HMM) trained on MFCCs was able to accurately classify ambient office and street sounds with an accuracy of 93% to 96%. A Gaussian Mixture Model (GMM) trained on MFCCs worked well on music classification (also known as Music Information Retrieval) but plateaus for complex variations. Aucouturier & Pachet (2004) showed the R-precision (i.e., the precision measure for information retrieval) struggled to get past 65%.

In bioacoustics, Cheng et al. (2012) used support vector machines (SVMs) on LPCs and MFCCs, and achieved up to 100% accuracy on bird songs from three different species. An HMM used by Clemins et al. (2005) for the identification of African elephant calls showed overall accuracy of 79.7% (but still achieved 90.9% for specific animal vocalisations of rumbles). Fagerlund (2007) used a multi-class classifier using two different datasets of at least six bird species; the classifier involved a decision tree with binary SVM classifiers at each node using MFCCs, and achieved accuracy scores from 91% to 98% depending on the species.

In a review of audio classification techniques used from 2000-2021, Mutanu et al. (2022) showed that the most popular pre-processing techniques include Short-Time Fourier Transformations, MFCCs, and LPCs. Machine learning ensemble algorithms used here resulted in accuracy scores of up to 87%, while Convolutional Neural Networks (CNNs) similarly resulted in accuracy scores of up to 88% (CNNs in bioacoustics are discussed in detail in section 2.3).

While earlier statistical learning techniques have shown good results in bioacoustic classification, later studies show more interest in Deep Learning and Neural Network techniques.

2.2 Convolutional Neural Networks

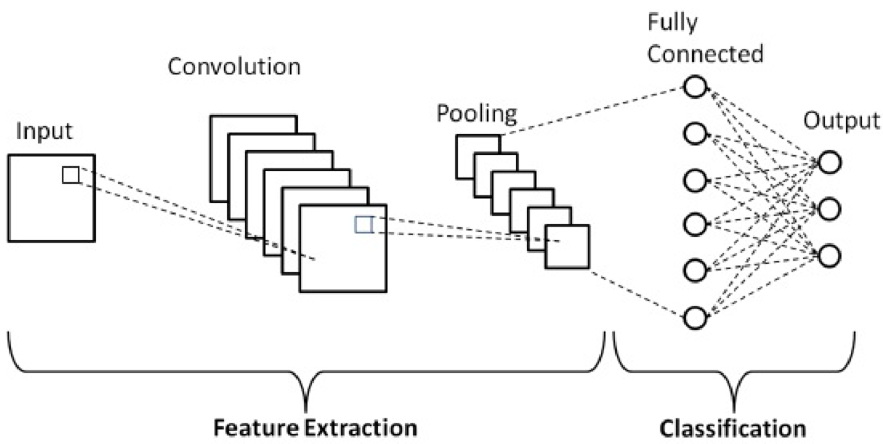


Figure 2.1: A diagram of a simple Convolutional Neural Network by Phung & Rhee (2019). The *Convolution* layer extracts features, while the *Pooling* layer downsamples the data.

Convolutional Neural Networks (CNNs) are a type of neural network that are highly suited for image data as they are able to extract features while reducing the dimensionality of the inputs (O'Shea & Nash (2015)). While CNNs share some properties with other neural networks such as having interconnected node structures, activation functions, and learning through backpropagation, they also have unique layers called *Convolutional* layers and *Pooling* layers (Géron (2023)). Figure 2.1 show these layers in relation to the rest of the neural network.

The *Convolutional* layer extracts important attributes from the input image before passing it on to the next layers. The image input can be though of as a matrix of pixel data, and the convolutional filter (also known as a kernel) transforms every pixel in all rows and columns of the matrix.

$$I_K(u,v) = \sum_{h=-m/2}^{m/2} \sum_{h=-m/2}^{m/2} K(h,k)I(u+h,v+k)$$

The formula above shows the convolution linear expression described by Fusiello (2024), where I_K is the output of a convolution kernel K on an image input I, with (u, v) denoting the pixel position of the input and output image, and (h, k) denoting the row, column position in the kernel, and m denoting the height or width of the kernel. Figure 2.2 shows how this is applied in a matrix.

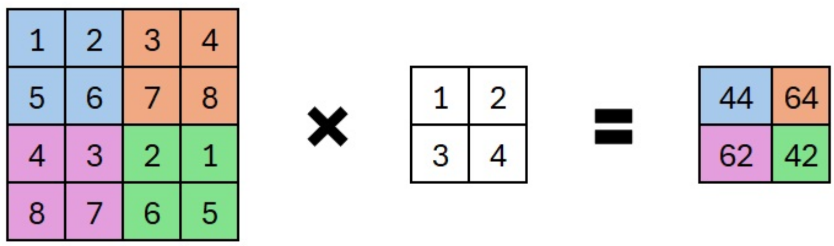


Figure 2.2: An application of the convolution linear expression showing a convolution kernel of size 2 x 2 applied on a matrix of size 4 x 4 with a step size of 2-cells both horizontally and vertically. The kernel K is applied to each colour quadrant of the image I, and outputs one cell value in I_K. So $I_K(1, 1) = (1*1)+(2*2)+(5*3)+(6*4)$ which is 44, and the process is applied to the rest of the input matrix.

One issue with applying the convolution kernel in the above example is the output matrix size is smaller than the input due to how the kernel moves through the image. The outer cells are not able to contribute as much as the inner cells, so information is lost in the outer perimeter. *Padding* addresses this problem by adding rows and columns to the edges, thereby increasing the size of the input matrix (Zhang et al. (2023)). Figure 2.3 shows how padding can output the same sized matrix as the input.

The *Pooling* layer downsamples the image matrix which helps to reduce the dimensions of the data while still keeping the important features, effectively outputting a "summary" of the data. Figure 2.4 shows an example of the pooling layer.

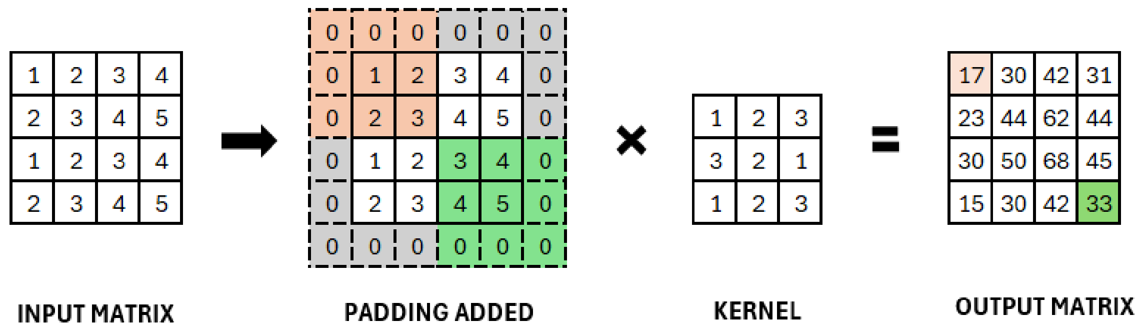


Figure 2.3: A matrix of size 4 x 4 is padded with extra cells around the boundaries. The recommended size of the horizontal and vertical padding p_h and p_w with respect to the kernel size k are $p_h = k_h - 1$ and $p_w = k_w - 1$. Here, p_h is $3 - 1 = 2$, which is the same for p_w. The padded cells *(in grey)* are distributed around the perimeter and are filled with 0 for all cells (also known as *zero padding*). Here, the step size is 1 cell at a time, so the output matrix will have the same size as the input matrix. Similar to Figure 2.2, $I_K(1,1)$ can be computed as $(0*1)+(0*2)+(0*3)+(0*3)+(1*2)+(2*1)+(1*0)+(2*2)+(3*3) = 17$, and is similar for the rest of the cells for the output image I_K.

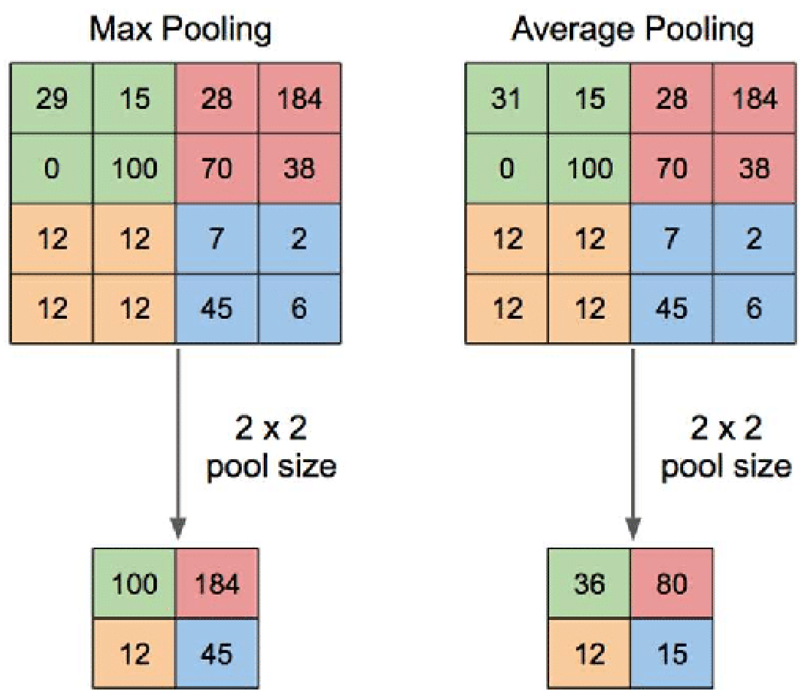


Figure 2.4: Illustrative examples of different pooling layers by Yani et al. (2019), showing how the pooling operation downsamples through the feature map. As their names suggest, Max Pooling *(left)* outputs the maximum value from each patch, while Average Pooling *(right)* outputs the average value for each patch.

One other important concept in convolutional layers is the *Stride*. This dictates how far the kernel steps through the image matrix. In Figure 2.3, the stride value was 1, and with the added padding, the matrix size is maintained. In the earlier example Figure 2.2 however, the stride value is 2, and outputs a smaller sized matrix. The stride can therefore allow the convolutional layer to extract features and downsample at the same time. Most CNN architectures have alternating convolutional and pooling layers, but Springenberg et al. (2015) showed that it is possible to replace the pooling layers with an increased stride value without loss in the CNN's accuracy.

CNNs will have different architectures for their convolutional and pooling layers, but their basic application is the same; they take inputs and then extract and downsample the inputs before passing them on downstream. Figure 2.5 shows a simplified analogy of the convolution and pooling layers' application.

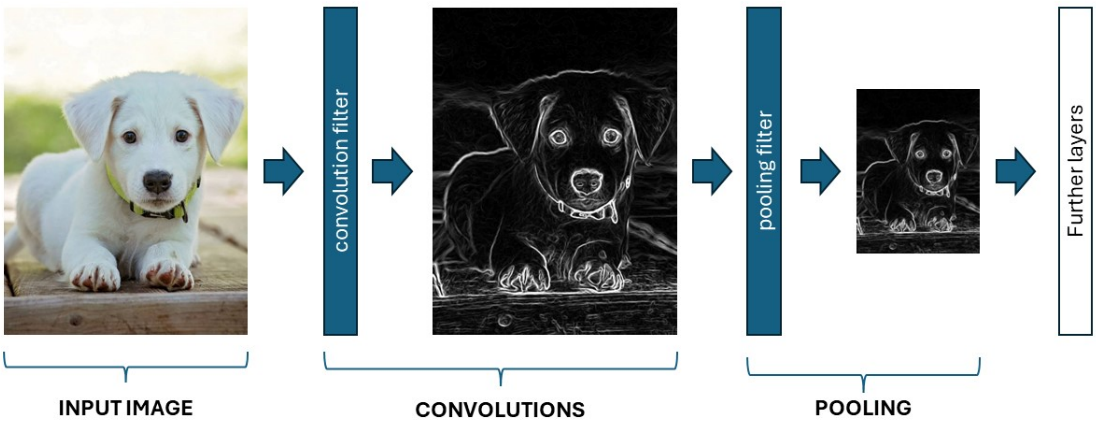


Figure 2.5: An illustrative example of how a convolutional layer and pooling layer process an image of a puppy from Pixabay (2016). The convolutional filter identifies important features such as the puppy's eyes, nose and paws, and then downsamples the input while still keeping the important features.

While the earliest neural networks were created to address image-based problems such as Lecun et al. (1998)'s LeNet-5 which was used to classify handwritten numbers, some models were applied to the field of bioacoustics. The model by Parsons (2001) was able to identify bat species through echolocation recordings converted into power spectra data. Parson's model was a neural network created using MatLab's NeuralNetwork Toolbox, and had an accuracy rate of 99% for one species.

These models were purposefully built from scratch, but the current practice makes use of pretrained CNNs applied to new and unseen data.

2.3 Transfer Learning

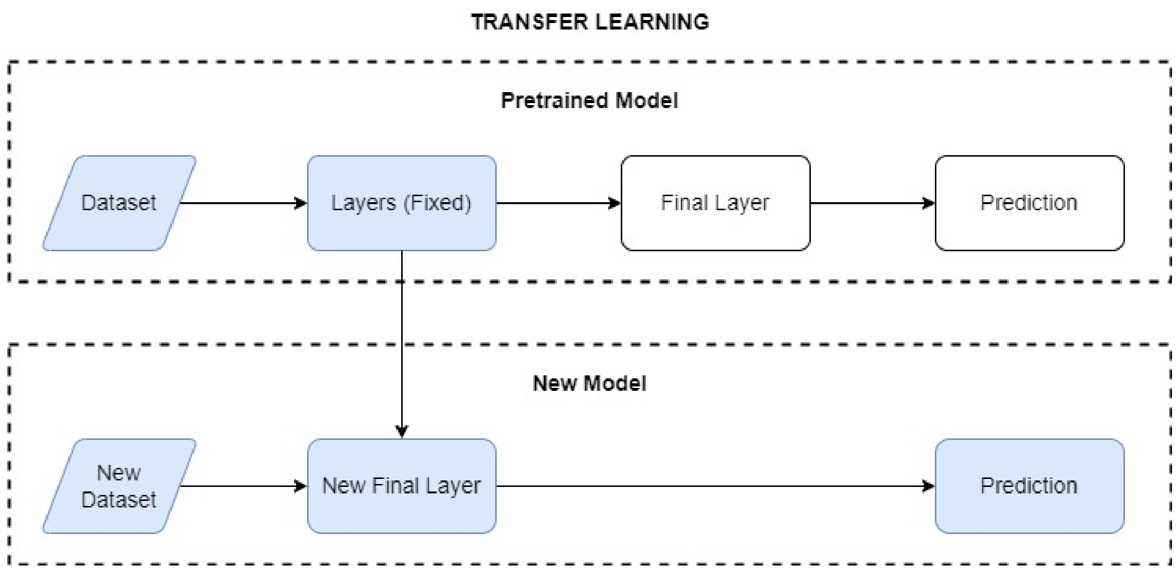


Figure 2.6: A simplified diagram of transfer learning showing how a pretrained model can be used on a different dataset for a new prediction.

Transfer Learning refers to the practice of using a model pretrained on a domain set and using the model as a feature extractor on a different target set (Chilamkurthy (2024)) or by fine-tuning the layers of the model through backpropagation (Li & Adeli (2024)). This process makes use of the "weights" learned by the model on the original dataset, but allows the model to learn on a new dataset, leading to a different prediction as illustrated in Figure 2.6.

Transfer learning also allows the use of smaller datasets which is immensely practical as CNNs typically use large datasets. In this way, researchers do not need to create a model from scratch nor build a large dataset, but are still able to achieve strong results (Razavian et al. (2014)).

Many models are available for transfer learning and have performed well on different datasets than the ones they were originally trained on. GoogLeNet (also known as *Inception*) is a CNN which uses multiple convolution kernels (instead of just one kernel) between layers for increased network depth and width (Szegedy et al. (2014)). Whereas traditional CNNs have connections only to the next layer, DenseNet by Huang et al. (2018) has connections to every other layer, for deeper training. In contrast, ResNet by He et al. (2015) has "shortcut connections" which skip one or more layers while still having sequentially connected layers. EfficientNet by Tan & Le (2020) builds on Resnet, but with additional scaling.

While CNNs and Transfer Learning are usually applied in computer vision use cases, these have also been effectively used in acoustics. The work by Palanisamy et al. (2020) on music genre classification using Densenet showed a result of 92.89% accuracy while Song et al. (2017) used Recurrent Neural Networks (RNNs) achieved 95.8% (RNNs are a related type of neural network that works well on sequential data (Keren & Schuller (2016)). Both studies used the GTZAN music dataset (Sturm (2014)) which contain 1,000 audio files of 30-second clips representing 10 music genres.

In the field of bioacoustic classification, strong results have also been found also using animal sounds. In a study by Zhong et al. (2020) on tropical birds and amphibians audio using ResNet50 originally trained on the ILSVRC-2015 dataset, also known as *ImageNet* (Russakovsky et al. (2015)), the researchers achieved an amazing 99.5% Area Under the Curve by adding a custom loss function and pseudo-labeling. A study by Baptista & Antunes (2021) which also used ResNet and a similar bird and frog dataset scored 91% accuracy. A series of experiments by Ghani et al. (2023) included AudioMAE, a transformer-type model originally trained on AudioSet, a database of generic sounds extracted from Youtube (Gemmeke et al. (2017)). The model was used on Black-tailed Godwit bird calls and achieved an AUC of over 90.0%. However, it should also be noted that in the same series, the models originally trained on bird and mammal sounds achieved far superior results.

For marine bioacoustics specifically, transfer learning has also been applied with similar success rates. Nur Korkmaz et al. (2023) used VGG16 (Simonyan & Zisserman (2015)) to detect the presence of dolphin whistles with an accuracy of 92.3%. In another study, Williams et al. (2024) created called ReefSet, a large annotated dataset on coral reef sounds. They used BirdNet (Kahl et al. (2021)) on ReefSet and obtained 90.8% AUC. Similar to Ghani et al. (2023)'s experiments, they also tried YAMNet (based on MobileNet (Howard et al. (2017)) and VGGish (based on VGG) originally trained on general audio, but these resulted in much poorer scores. Lu et al. (2021) used AlexNet Krizhevsky et al. (2017) on the Watkins Marine Mammal Sound Database (Sayigh et al. (2017)) and achieved 97.4% accuracy on classification tasks. These studies and their corresponding scores are summarised in Table 2.1.

Author and year	**Base model**	**Target dataset**	**Score**
Williams et al. (2024)	BirdNET	ReefSet	90.8% AUC
Ghani et al. (2023)	AudioMAE	Godwit Calls	>90.0% AUC
Nur Korkmaz et al. (2023)	VGG16	Dolphin whistles in Israel	92.3% accuracy
Baptista & Antunes (2021)	ResNet	Puerto Rican birds & frogs	91.0% accuracy
Lu et al. (2021)	AlexNet	Watkins Marine DB	97.4% accuracy
Zhong et al. (2020)	ResNet	Puerto Rican birds & frogs	99.5% AUC

Table 2.1: A selected list of recent studies in CNN Transfer Learning focusing on bioacoustics. While the target datasets focused on bioacoustics (bird calls, dolphin whistles, etc.), the studies all achieved superior results, even with base models that were originally trained on image data or generic audio data.

In all these studies, audio data was first converted into spectrogram or mel-spectrogram images before being applied to the CNNs, however they achieved transfer learning in different ways. Ghani et al. (2023), Williams et al. (2024), and Lu et al. (2021) removed or replaced some of their CNN's layers. Zhong et al. (2020) and Nur Korkmaz et al. (2023) used a custom loss function or optimiser. Baptista & Antunes (2021) experimented with different window sizes in their Mel spectrograms.

Despite the overall good results, the studies by Ghani et al. (2023) and Williams et al. (2024) also highlight that original training done using bioacoustic data (rather than general acoustic sounds or images) still lead to higher scores.

Based on these studies, transfer learning can therefore be achieved by slightly modifying the CNN architecture, or by hyperparameter tuning, or by data augmentation at the source, and respectable scores can still be obtained with CNN models trained on general datasets.

2.4 CNN Software Frameworks

Due to the complexity of these models' architectures, software frameworks have been made available for easier research and development in CNNs. Rather than creating these models from scratch, these frameworks allow the use of design patterns (i.e. repeatable "blueprints" for writing code) and CNN models ready for transfer training.

In Deep Learning applications, tensors are used which are a multi-dimesional type of data object (Kolecki (2002)) that allows for faster training on a GPU (Pytorch (2024)). This data structure is used for encoding and decoding model outputs by software frameworks such as TensorFlow and PyTorch.

TensorFlow was developed by Google's internal research team (Abadi et al. (2016)) and released in 2015 as an open-source Deep Learning library, with deep interactions with Google Cloud's platform. PyTorch was developed by Meta AI also as an open-source Deep Learning library but with a more "Pythonic programming style" (Paszke et al. (2019)).

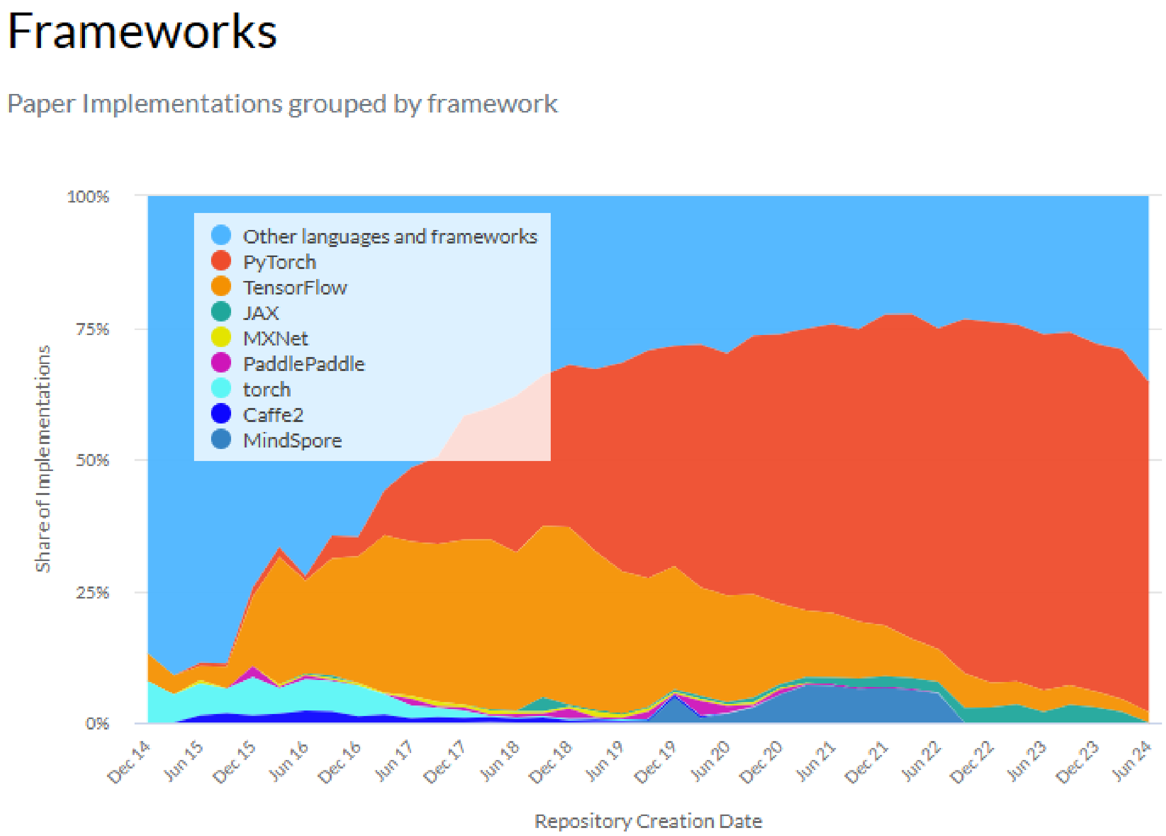


Figure 2.7: A screenshot from *Papers With Code: Trends* (2024) showing the framework usage from June 2014 to June 2024. Here, TensorFlow led the trend up until 2019 when PyTorch took a significant lead.

The graph in Figure 2.7 shows that as of June 2024, *Papers With Code: Trends* (2024) recorded that 63% of published papers' repositories used PyTorch, while only 2% used TensorFlow (and 35% used other languages). PyTorch's popularity along with its "pythonesque" syntax make it ideal for this study's experimentation in CNNs.

3 | Methodology

The experiments for this study comprised five main sections; first, the wav files needed to be converted to spectrogram images; second, transfer learning with the image dataset was done using pretrained CNNs; third, the disproportioned dataset was addressed; fourth, the model hyperparameters were manually tuned; and fifth, the best model was used to make predictions on the *Unknown* data. Figure 3.1 shows the flow of these experiments.

This study used the Python programming language for data processing, model training and analysis. Training was done using a remote NVIDIA A30 Tensor Core GPU, configured to 4 cores and 16GB of memory, using a Python virtual environment.

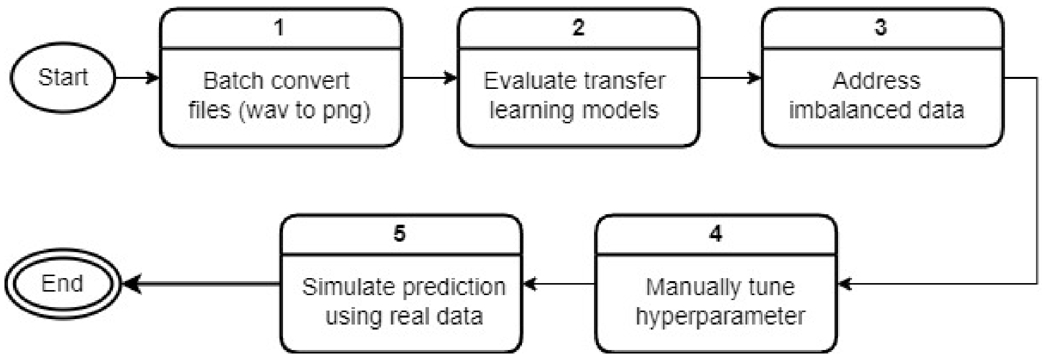


Figure 3.1: Outline of the processes done for this study.

3.1 Converting Audio WAVs to Spectrogram PNGs

As previous studies have shown that CNNs work well on image data converted from audio, this study will make use of spectrogram images from the original audio dataset.

A batch function was created that walked through all folders and subfolders in the dataset and checked only folders for *Insular* and *Pelagic* parent folders, ignoring the *Unknown* folder. The function also looked for *wav* files only, ignoring other file types.

For each *wav* file found, the raw audio data was extracted and "flattened" to only one channel. The audio sample data and sample rate were then converted to a *NumPy* (Harris et al. (2020)) array. Using these samples and sample rate, the raw frequencies, times, and spectrogram data were then extracted.

Finally, spectrogram plots were created using the frequencies, times and spectrogram data. The plot size was set to 12 x 8 inches for easier visual inspection, and were created without any axes and labels as these are not important to the CNN model. The created plots were then saved to their respective *Insular* and *Pelagic* folders.

3.2 Transfer Learning Using Pretrained Models

Due to the small dataset size of around 2,800 audio files, transfer learning was used with pretrained CNNs as fixed feature extractors (i.e., the weights in the network are fixed except for the final fully connected layer only). Transfer learning was done using PyTorch and *Torchvision* (2024) as their "pythonesque" interface flowed well with the rest of the Python scripts, and the PyTorch code by Chilamkurthy (2024) was readily available for use. Code Listing 3.1 shows the relevant snippet where the pretrained model can be specified and the last layer only can be updated.

```python
# Loading the pretrained model
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# Freezing the model's other layers
for param in model_conv.parameters():
    param.requires_grad = False

# Changing the number of features to 2 (for Insular and Pelagic)
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

# Optimising the parameters of the final layer "fc" layer only
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001,
    momentum=0.9)
```

Code Listing 3.1: Code snippets adapted from the official PyTorch documentation by Chilamkurthy (2024) showing how to use a CNN as fixed feature extractor in PyTorch. This example shows ResNet, but the code is similar for the other CNNs in this study.

The pretrained models Densenet, Resnet, Googlenet and EfficientNet were used as base models as these were used in similar studies that showed good results, and they share similar APIs making the code easier to configure between the pretrained models. All the models were loaded from *torchvision.models*, and *IMAGENET1K_ V1* was used for the weights.

The number of features in the final layer was changed to "2" for the target classes (*Pelagic* and *Insular*), and only this fully connected layer was optimised while the rest of the layers were fixed.

The optimiser used was the same as the default provided in the Pytorch documentation (SGD, learning rate set to 0.001, and momentum set to 0.9), and the image inputs transformed to 224x224 pixels (the minimum required for all models except for EfficientNet which required 384x384), converted to tensor, then normalised with mean values of [0.485, 0.456, 0.406], and standard deviation values of [0.229, 0.224, 0.225].

The converted images were then loaded by batch using Torchvision's Datasets and Dataloader, randomly split into train, validation and test sets of 60%, 20%, 20% respectively (with the test set only used on the best model during prediction simulation). However, although these experiments used the *random_ split* method, the *torch.manual_ seed* was set to "220029955" to ensure reproducibility across all experiments.

The models were trained in 24 epochs using a batch size of 64, and the loss and accuracy were computed for each training epoch, with the highest scoring epoch model saved. The accuracy scores for all models were then compared to determine the best pretrained model to be used for the rest of the study.

Although accuracy might not be ideal for imbalanced data, it was used here to compare with related studies' accuracy scores as discussed in Section 2.3. To provide a better gauge of performance, the precision and recall metrics were also included as these take into account how the model predicts for the underrepresented class.

3.3 Addressing Imbalanced Data

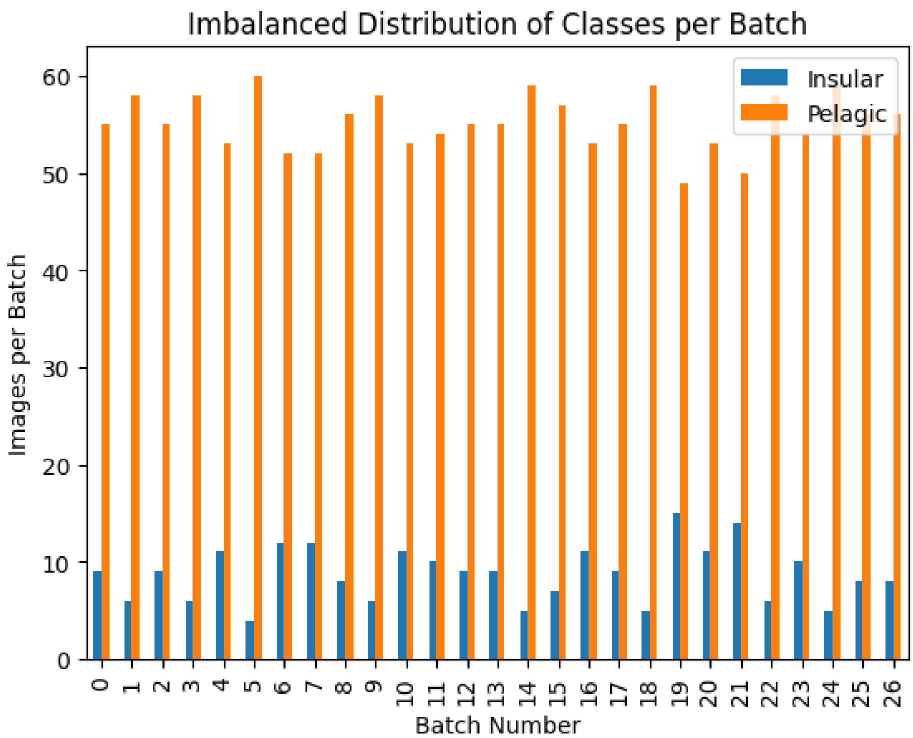


Figure 3.2: Imbalanced data per batch using PyTorch's Dataloader shows the *Pelagic* has overwhelmingly more selected instances than *Insular*.

The dataset was severely imbalanced with audio from the *Pelagic* group comprising 86.7% of all data. This imbalance led to the Dataloader continuously undersampling the *Insular* class for all batches as shown in Figure 3.2. Predicting without addressing this imbalance would lead to a poorly performing predictor (Yadav & Bhole (2020)), or as Géron (2023) calls it, a *weak learner* (i.e., a classifier that performs as well as random guessing). The CNN could score an accuracy of close to 86.7% which looks good without context, but would be the same result as the model guessing all instances as the majority class *Pelagic*.

An alternative here is to use straightforward undersampling of the majority *Pelagic* class to match the number of *Insular* instances. However, this would lead to severe loss of information as there would only be 439 data points for each class.

To address this, the *Weighted Random Sampler* from *torch.utils.data* (2023) was used which assigns a weight to each item in the dataset for the Dataloader sampler, allowing the Dataloader to undersample the *Pelagic* class, and oversample (sample with replacement) the *Insular* class. While oversampling may lead to overfitting due to the synthetic instances used (Alkhawaldeh et al. (2023)), it is still a better alternative as using the imbalanced data would lead to the model learning only the majority class, and using straightforward undersampling would lead to loss of informationn.

In addition to the accuracy, precision and recall metrics, Precision-Recall Curves and Precision-Recall Threshold plots were also used to evaluate the effectiveness of the weighted sampler.

3.4 Hyperparameter Tuning and Final Testing

Hyperparameter tuning allows for changing the model values that affects the training process, leading to a better performing model. Instead of external software libraries, hyperparameter tuning for this study was done manually by looping through different values for the *learning rate* and *momentum*.

First, a python script looped through the following learning rates with the momentum fixed at 0.90: [0.0005, 0.001, 0.005, 0.01 0.5, 0.1]. Accuracy scores were compared to determine the best learning rate. Then, with the best learning rate, another script looped through the following momentum values: [0.90, 0.95, 0.97, 0.99].

The accuracy scores are then compared and the hyperparameters with the best scores are selected as the best model.

As a final test, the best model was compared against the unseen test set. A confusion matrix was also created to show how the model performs for the minority and majority classes.

3.5 Simulating Predictions with Real Data

```
output:  tensor([[ 0.3622, -0.5187]])
index: 0
class name: Insular
probabilities: tensor([[0.7070, 0.2930]])

There's a 70.7 % chance that this audio file belongs to the Insular
    group.
```

Code Listing 3.2: Sample output when predicting for each test audio file.

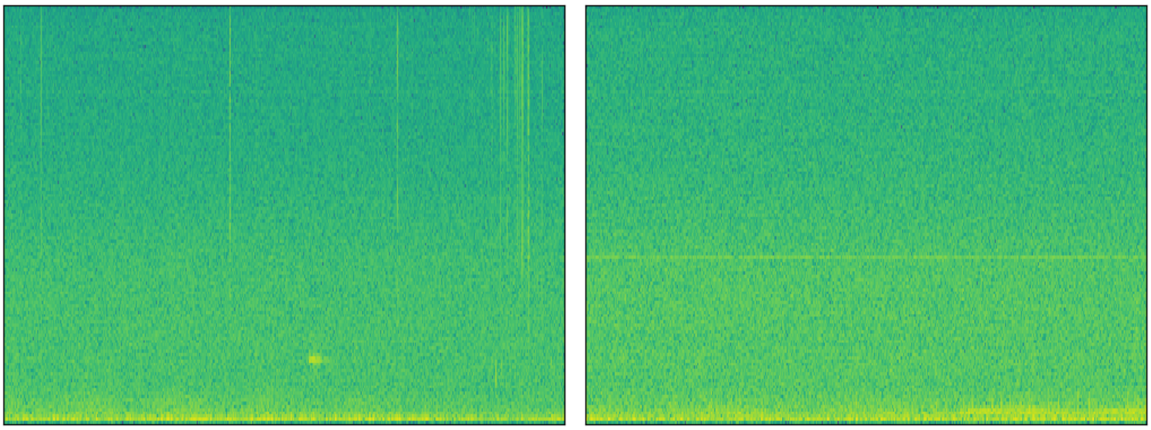
To prototype the best model in a realistic setting, a simple Python script was created which takes in a wav file, creates a spectrogram plot, transforms the spectrogram into a tensor, and inputs that tensor into the model.

The model then outputs the predicted class along with the probability for that class.

10 audio files were then selected from the *Unknown* dataset through convenience sampling (i.e., the samples were selected randomly by the researcher).

4 | Results

A total of 2,842 spectrogram plot images were created from the audio dataset without any labels or axes marks so only spectrogram image data was fed into the CNN, as shown in Figure 4.1. Each spectrogram file was around 300kb in size, and with dimensions of 950 x 636 pixels (12 x 8 inches) each, although these were resized later during the transfer learning to 224 x 224 or 384 x 384 depending on the pretrained CNN.



(a) Insular class (b) Pelagic class

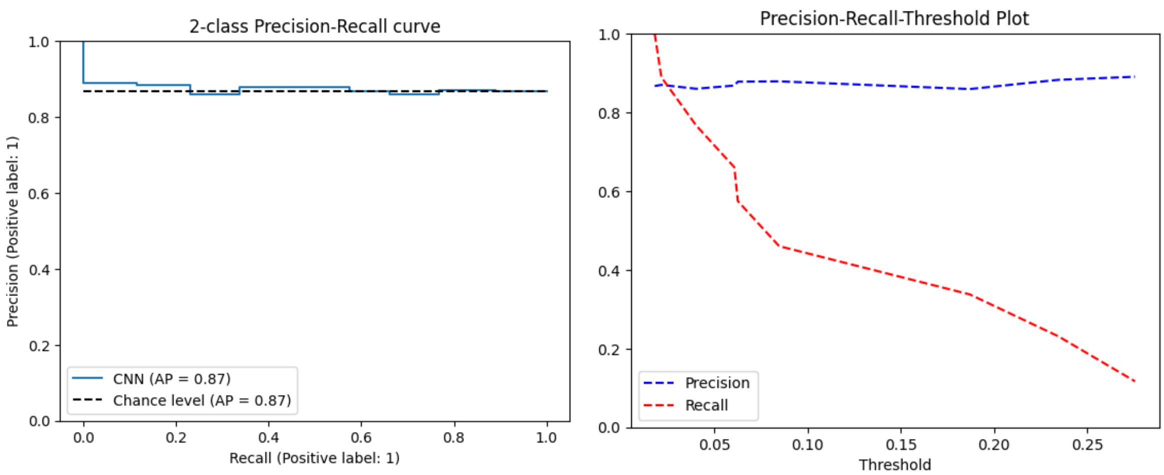
Figure 4.1: Spectrograms from the first wav files in each class; *Insular* on the left and *Pelagic* on the right. The vertical streaks in the *Insular* example are not found in the *Pelagic* class. In contrast, the horizontal line in the middle of the *Pelagic* example can't be seen in the other class. However, these may be attributed to other ambient ocean sounds.

The different pretrained CNN models were loaded and transfer learning applied on the spectrogram imageset, with ResNet showing the best accuracy score of 0.9238. DenseNet followed with an accuracy of 0.9149, but GoogLeNet and EfficientNet performed poorly with accuracies below 0.9, and both had recall values of 1.0. A summary of their results can be seen in Table 4.1.

Base Model	Accuracy	Precision	Recall
ResNet	0.9238	0.9273	0.9918
DenseNet	0.9149	0.9122	0.9980
GoogLeNet	0.8759	0.87479	1.0000
EfficientNet	0.8670	0.8670	1.0000

Table 4.1: Validation results from transfer training using different pre-trained models, sorted by best accuracy score descending.

These accuracy scores resulted from using the imbalanced data, and even ResNet showed poor results for its Precision-Recall tradeoff as seen in figure 4.2. The precision-recall curve should show the line curving toward the 1.0 precision and 1.0 recall (in the upper right), and the precision-recall threshold should show the precision and recall lines meeting at around 0.5. However, neither was the case for these graphs, indicating a critical need to address the imbalance.



(a) Precision-Recall Curve Plot

(b) Precision-Recall Threshold Plot

Figure 4.2: The precision-recall curve *(left)* show how the model learns towards a precision of 0.87, which is about the same as the majority class ratio of 87.6%. The threshold plot *(right)* also show the effect of the severely imbalanced data, with the threshold at close to 0.0 where recall is close to 1.0. These plots show the effect of the imbalanced majority class in the dataset.

The ResNet transfer learning model was run again with weighted data as shown in Figure 4.3. With the *Weighted Data Sampler* applied, the ResNet model was run with the balanced Datasampler, and the resulting model yielded a lower accuracy score of 0.9078 compared to the previous imbalanced dataset as seen in Table 4.2.

Model	**Accuracy**	**Precision**	**Recall**
ResNet (Imbalanced)	0.9238	0.9273	0.9918
ResNet (Weighted)	0.9078	0.9700	0.9223

Table 4.2: Validation accuracy results comparing the ResNet model *before* and *after* balancing the dataset. The accuracy dipped slightly after applying Pytorch's *Weighted Random Sampler*.

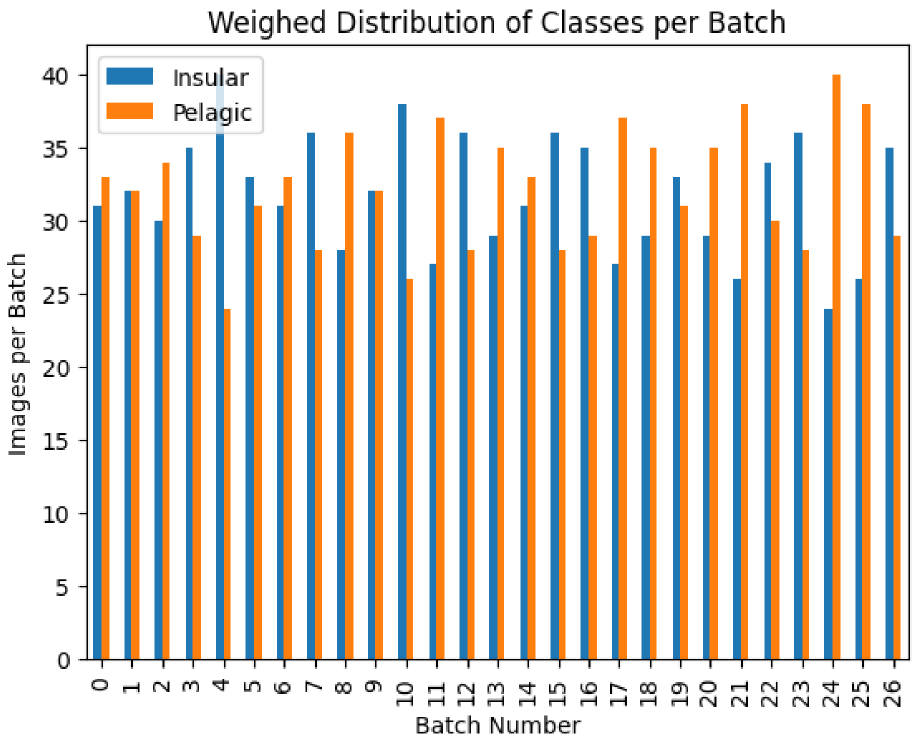
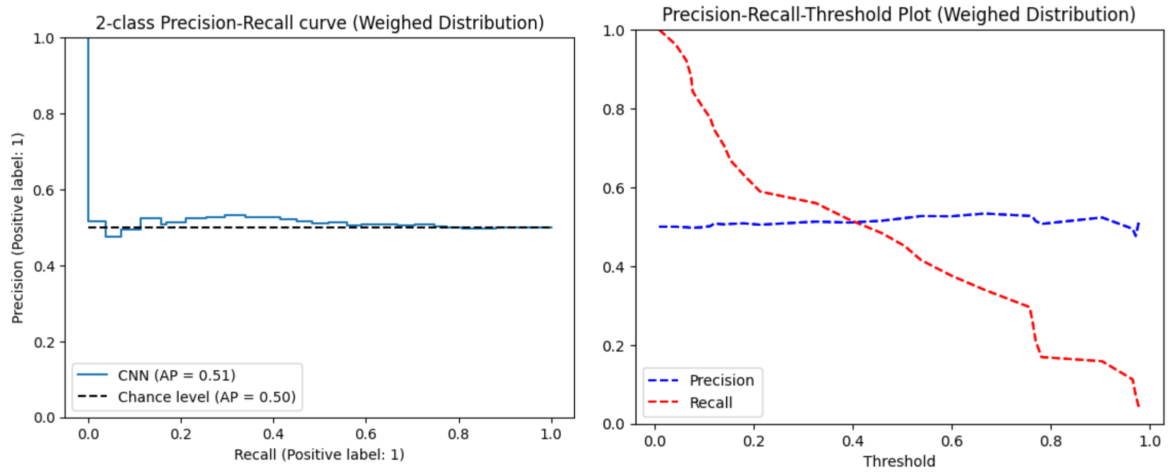


Figure 4.3: Using Pytorch's *Weighted Data Sampler*, the Dataloader was able to undersample *Pelagic* instances and oversample *Insular* instances which produced a more balanced dataset across all batches.



(a) Precision-Recall Curve

(b) Precision-Recall Threshold

Figure 4.4: ResNet Precision-Recall plots after applying *Weighted Data Sampler*. The precision and recall threshold *(right)* came closer to a more balanced threshold of around 0.4, however the precision-recall curve *(left)* while showing a balance chance level of 0.5 still showed the model line "hugging" the chance line, indicating there were still deficiencies with the model.

Despite the resulting lower accuracy score and poor precision-recall curve, the precision increased significantly to 0.9700 after using the weighted dataset, so hyperparameter tuning was performed on the newer model, with the learning rate tuned first followed by the momentum. The results in Table 4.3 showed the best accuracy scores were a learning rate of 0.01, and momentum values of 0.95 and 0.97.

Loss curve graphs were also created for these, and it was expected that they would show smooth curves for training and validation. However, Figure 4.5 shows erratic movement for both hyperparameters, and was most noticeable for the momentum training.

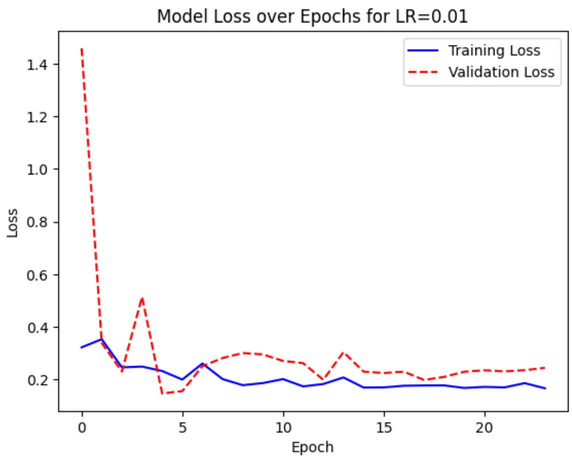
Learning Rate (LR)

LR Value	Accuracy
0.0005	0.9131
0.001	0.9291
0.005	0.9450
0.01	0.9486
0.05	0.9433
0.1	0.9220

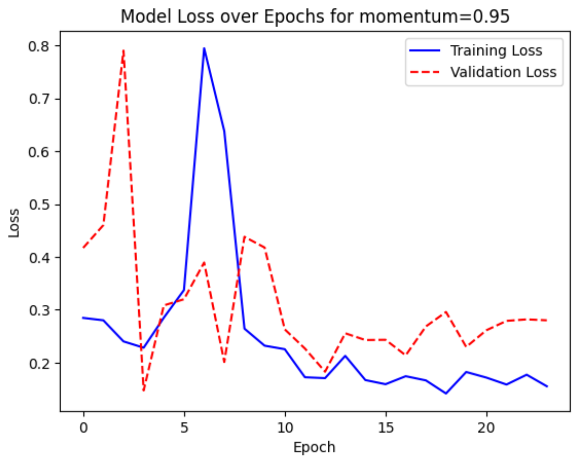
Momentum

Momentum Value	Accuracy
0.90	0.9415
0.95	0.9486
0.97	0.9486
0.99	0.9468

Table 4.3: Accuracy results for manual hyperparameter tuning for Learning Rate *(left)* and Momentum *(right)*. For Learning Rate, the momentum was fixed at 0.90, and subsequent Momentum tuning with the best LR of 0.01. The highlighted rows show the best scores.



(a) Model Loss for LR=0.01



(b) Model Loss for momentum=0.95

Figure 4.5: Model Losses over Epochs for best learning rate *(left)* and momentum *(right)* over 24 epochs. Despite having the best scores for hyperparameter tuning, the graphs show erratic behaviour.

The final test with the unseen set was done using the ResNet model with the balanced dataset, a learning rate of 0.01 and a momentum of 0.95, resulting in a final accuracy score of 0.9309 as seen in Table 4.4.

(a) Confusion Matrix

		Predicted 1	Predicted 0
Actual	**1**	48	26
	0	13	477

(b) Metrics Test Set

Metric	**Score**
Accuracy	0.9309
Precision	0.9483
Recall	0.9735

Table 4.4: The confusion matrix *(left)* showed the model prediction performance, where the True Positives represent the *Insular* class, while True Negatives are the *Pelagic* class. The model performed a decent job as it correctly identified almost two-thirds of the minority class *Insular*.

Putting these results in the context of the flow of the study, Table 4.5 shows relatively stable accuracy scores from the starting base model's validation accuracy of 0.9238 up to the final model's test accuracy of 0.9389.

Table Ref.	**Description**	**Accuracy**	**Precision**	**Recall**
4.1	ResNet with transfer learning	0.9238	0.9273	0.9918
4.2	Above model + balanced Dataloader	0.9078	0.9700	0.9223
4.3	Above model + best hyperparameters	0.9486	0.9617	0.9755
4.4	Above model on unseen test set	0.9309	0.9483	0.9735

Table 4.5: Final accuracy results comparing the best base model, the base model with weighted dataset, the weighted dataset model with the best hyperparameters, and the same model with the unseen data. Note that all model scores were tested using the validation set, except the final model which used the test set.

The final model was used on a sample of 10 files in the *Unknown* audio dataset, and was able to predict their respective classes as seen in Table 4.6.

Source Folder	**Predicted Class**	**Probability**
Lasker_AC_67	Insular	0.5603
Lasker_AC_67	Pelagic	0.9424
Lasker_AC_150	Pelagic	0.8420
Lasker_AC_276	Pelagic	0.9082
Lasker_AC_276	Pelagic	0.9986
Lasker_AC_276	Pelagic	0.9422
Sette_AC_44	Pelagic	0.9770
Sette_AC_91	Pelagic	0.9982
Sette_AC_256	Insular	0.9937
Sette_AC_256	Insular	0.9497

Table 4.6: Predictions using a sample of the *Unknown* audio files. Most of the predictions were given with high probabilities of >0.90.

5 | Discussion

During the initial CNN transfer learning iterations, the model was expected to perform with an accuracy of at least 0.8670 (which is the percentage of *Pelagic* audio in the dataset). ResNet was the best model, and its accuracy of 0.9238 meant that the CNN performed better than a weak learner, and was comparable to the related studies in CNN Transfer Learning (Section 2.3) where one model had an accuracy of 91.0%.

In contrast, GoogLeNet and EfficientNet both had accuracy scores very close to the ratio of 0.87, and had recall of 1.0, suggesting that these models did not perform any better than guessing. Table 5.1 shows how poorly the worst model, EfficientNet, compared with the best model, Resnet. While ResNet had the best results and was selected as the base model for the succeeding experiments, it was interesting to note that the poorly performing EfficientNet was actually built on top of ResNet, and its "compound scaling method" architecture did not work for this use case.

		Predicted	
		1	**0**
Actual	**1**	0	75
	0	0	489

(a)
Accuracy: 0.8670
Recall: 1.0000

		Predicted	
		1	**0**
Actual	**1**	37	38
	0	4	485

(b)
Accuracy: 0.9256
Recall: 0.9918

Table 5.1: Confusion matrices for *(a)***EfficientNet** on the left and *(b)***ResNet** on the right. Aside from the EfficientNet's poorer accuracy, it also had a Recall of 1.0 and its confusion matrix shows why this is the case; it predicted all cases as *Pelagic* (True Negatives), while none for *Insular* (True Positives).

The results with the weighted dataset were also interesting since the accuracy decreased after using the *Weighted Random Sampler*, despite the dataset being more balanced. However, the decreased accuracy was not indicative of overall model performance. Table 5.2 below shows how "sacrificing" the majority class *Pelagic* (here, the True Negatives) can lead to more correct cases of the minority class *Insular*(the True Positives), despite the lower overall accuracy.

The increase in precision to 0.9700 also showed that, although the accuracy was lower, overall the model predicted more correct cases for each class. Therefore, the weighted dataset was still used in the next phase of the study.

		Predicted 1	Predicted 0
Actual	**1**	37	38
	0	4	485

(a)
Accuracy: 0.9256
Precision: 0.9273

		Predicted 1	Predicted 0
Actual	**1**	61	14
	0	38	451

(b)
Accuracy: 0.9078
Precision: 0.9700

Table 5.2: The confusion matrices of the ResNet model *(a)***before** and *(b)***after** applying the *Weighted Random Sampler (WRS)*, showing how a model with lower accuracy might be more acceptable. In the left matrix, the model has a higher accuracy of 0.9256 with more True Negatives *(Pelagic)*, but with fewer True Positives *(Insular)*, leading to a lower precision of 0.9273. In the right matrix, the model has a slightly lower accuracy of 0.9078 due to fewer True Negatives*(Pelagic)*, but more True Positives *(Insular)*, leading to a much higher precision of 0.9700.

The predictions on the *Unknown* dataset shows the feasibility of this model as a prototype binary classifier. Most of the *Unknowns* were found to be Pelagic, although this could due to the weighted data being oversampled for *Insular*, and undersampled for *Pelagic* (i.e., the model might still have learned better with the varied *Pelagic* data).

Based on the above predictions on the *Unknown* set, an assumption could be made that files in the same folder are mostly of the same class, but based on the audio survey methods, one folder may contain recordings from different dates and times, so different groups may still be found in the same folder.

While the final scores for accuracy, precision and recall were overall good, attention should also be given to the precision-recall graphs in and the loss curves in show deficiencies in the model; Figure 4.4 shows the model line still hugging the chance line, and Figure 4.5 shows fluctuations in both training and validation losses. These problems could have been caused by the repetitive synthetic data or some gradient issues, and could have been addressed with regularisation (James et al. (2023)), optimiser selection (Géron (2023)), or additional hyperparameter tuning.

Despite these shortcomings, this study shows that CNN Transfer Learning can be quite capable of bioacoustics binary classification, and the results suggest that it is possible to distinguish between the *Insular* and *Pelagic* groups of Hawaiian False Killer Whales based on audio data alone.

However, the difference in audio might not be solely attributed to the animal sounds, as other factors might have influenced the CNN classifier. The classifier may have mistakenly considered engine and ambient noise as important features, and sea sounds may have varied due to the effect of water temperature, salinity and recording depth on the recording process (Bradley & Stern (2008)).

On the other hand, if it were to be assumed that the difference could be attributed to the animal sounds alone, then the final test scores can still be improved as similar studies have resulted in much higher metrics (as Zhong et al. (2020) showed an AUC of 99.5%). Therefore, there is still much work that can be done in distinguishing between *Pelagic* and *Insular* False Killer Whale sounds.

Additional methods such as regularisation or additional hyperparameter tuning could have been applied to the CNNs to improve the final models'scores. Based on similar studies, strong results could also be obtained from simpler machine learning methods such as support vector machines which have resulted in 100% accuracy (Cheng et al. (2012)). Alternatively, since the current dataset is small, it could also be possible to attain high accuracy scores by pre-training the CNN on larger bioacoustic datasets such as *ReefSet* rather than on the general dataset *ImageNet* as done by Ghani et al. (2023) and Williams et al. (2024).

Finally, other CNN methods could also be explored with audio and signal processing libraries such as PyTorch's *TorchAudio* (Hwang et al. (2023)) which do not require spectrograms or other image inputs, but instead trains directly on audio data.

6 | Conclusion

This study aimed to determine whether it is possible to classify between *Pelagic* and *Insular* groups of Hawaiian Killer Whales based on their audio recordings. The results show that this classification task is possible using simple CNN transfer learning, with an accuracy of 93.89%.

The significant class imbalance in the dataset (with 87.6% belonging to the *Pelagic* class alone) affected the initial transfer learning accuracy scores, but this was addressed using Pytorch libraries. The model's learning rate and momentum were also tuned to produce slightly better accuracy scores.

The final model was able to predict on an *Unknown* dataset with some confidence, showing that CNN transfer learning can be used effectively for bioacoustic classification.

While this study can be extended with the use of further machine learning and CNN techniques, the current results demonstrate the effectiveness of the CNN "Black Box" with transfer learning; a CNN trained on the *ImageNet* dataset can work remarkably well on *Pseudorcas* audio-to-image dataset by training the last fully connected layer, and achieve strong accuracy, precision and recall scores.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. & Zheng, X. (2016), 'Tensorflow: A system for large-scale machine learning'.
URL: *https://arxiv.org/abs/1605.08695*

Abbot, D. (2024), *Understanding Sound*, Pressbooks.
URL: *https://pressbooks.pub/sound/*

Alkhawaldeh, I. M., Albalkhi, I. & Naswhan, A. J. (2023), 'Challenges and limitations of synthetic minority oversampling techniques in machine learning', *World Journal of Methodology* **13**(5), 373âĂŞ378.
URL: *http://dx.doi.org/10.5662/wjm.v13.i5.373*

Aucouturier, J.-j. & Pachet, F. (2004), 'Improving timbre similarity: How high's the sky?', *J. Negat. Results Speech Audio Sci* **1**.

Bäckström, T., RÃďsÃďnen, O., Zewoudie, A., Zarazaga, P. P., Koivusalo, L., Das, S., Mellado, E. G., Mansali, M. B., Ramos, D., Kadiri, S. & Alku, P. (2022), *Introduction to Speech Processing*, 2 edn.
URL: *https://speechprocessingbook.aalto.fi*

Baird, R. W. (2009), F - false killer whale: Pseudorca crassidens, *in* W. F. Perrin, B. WÃijrsig & J. G. M. Thewissen, eds, 'Encyclopedia of Marine Mammals (Second Edition)', Academic Press, pp. 405–406.
URL: *https://www.sciencedirect.com/science/article/pii/B9780123735539000973*

Baptista, P. B. & Antunes, C. (2021), 'Bioacoustic classification framework using transfer learning', *Model Decision Artificial Intelligence* **35**.

Bradbury, J. (2000), 'Linear predictive coding', *Mc G. Hill* .

Bradley, D. L. & Stern, R. (2008), 'Underwater sound and the marine mammal acoustic

environment: A guide to fundamental principles'.
URL: *https://www.mmc.gov/wp-content/uploads/sound_bklet.pdf*

Carretta, J. V., Oleson, E. M., Oleson, E. M., Forney, K. A., Muto, M. M., Weller, D. W., Lang, A. R., Baker, J., Hanson, B., Orr, A. J., Barlow, J., Moore, J. E. & Brownell, R. L. J. (2022), 'U.s. pacific marine mammal stock assessments: 2021'.
URL: *https://repository.library.noaa.gov/view/noaa/44406*

Cheng, J., Xie, B., Lin, C. & Ji, L. (2012), 'A comparative study in birds: call-type-independent species and individual recognition using four machine-learning methods and two acoustic features', *Bioacoustics* **21**(2), 157–171.
URL: *http://www.tandfonline.com/doi/abs/10.1080/09524622.2012.669664*

Chilamkurthy, S. (2024), 'Transfer learning for computer vision tutorial - pytorch tutorials 2.3.0+cu121 documentation'.
URL: *https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html*

Clemins, P. J., Johnson, M. T., Leong, K. M. & Savage, A. (2005), 'Automatic classification and speaker identification of African elephant (*Loxodonta africana*) vocalizations', *The Journal of the Acoustical Society of America* **117**(2), 956–963.
URL: *https://pubs.aip.org/jasa/article/117/2/956/541588/Automatic-classification-and-speaker*

Dewynter, M. (2019), 'Natural science illustrations of marine mammals'.
URL: *https://sanctuaire-agoa.fr/editorial/natural-science-illustrations-marine-mammals*

Digby, A., Towsey, M., Bell, B. D. & Teal, P. D. (2013), 'A practical comparison of manual and autonomous methods for acoustic monitoring', *Methods in Ecology and Evolution* **4**(7), 675-683.
URL: *http://dx.doi.org/10.1111/2041-210X.12060*

DOSITS (2021), 'False killer whale sounds'.
URL: *https://dosits.org/galleries/audio-gallery/marine-mammals/toothed-whales/false-killer-whale/*

Fagerlund, S. (2007), 'Bird species recognition using support vector machines', *EURASIP Journal on Advances in Signal Processing* **2007**(1), 038637.

Fusiello, A. (2024), *Computer vision: Three-dimensional reconstruction techniques*, Springer Nature, Cham, Switzerland.

Gemmeke, J. F., Ellis, D. P. W., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., Plakal, M. & Ritter, M. (2017), Audio set: An ontology and human-labeled dataset for audio events, *in* 'Proc. IEEE ICASSP 2017', New Orleans, LA.

Gerhard, D. (2003), 'Audio signal classification: History and current techniques', *Semantic Scholar* .
URL: *https://api.semanticscholar.org/CorpusID:7652753*

Géron, A. (2023), *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*, Data science / machine learning, third edition edn, O'Reilly, Beijing Boston Farnham Sebastopol Tokyo.

Ghani, B., Denton, T., Kahl, S. & Klinck, H. (2023), 'Global birdsong embeddings enable superior transfer learning for bioacoustic classification', *Scientific Reports* **13**(1).
URL: *http://dx.doi.org/10.1038/s41598-023-49989-z*

Giannakopoulos, T. & Pikrakis, A. (2014), *Introduction to Audio Analysis*, Elsevier.
URL: *https://linkinghub.elsevier.com/retrieve/pii/C20120035247*

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. & Oliphant, T. E. (2020), 'Array programming with NumPy', *Nature* **585**, 357-362.

He, K., Zhang, X., Ren, S. & Sun, J. (2015), 'Deep residual learning for image recog-

nition', *CoRR* **abs/1512.03385**.
URL: *http://arxiv.org/abs/1512.03385*

Honig, M. & Schackwitz, W. (2023), 'Manual versus semiautomated bioacoustic analysis methods of multiple vocalizations in tricolored blackbird colonies', *Journal of Fish and Wildlife Management* **14**(1), 225-238.
URL: *http://dx.doi.org/10.3996/JFWM-22-065*

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. & Adam, H. (2017), 'Mobilenets: Efficient convolutional neural networks for mobile vision applications'.
URL: *https://arxiv.org/abs/1704.04861*

Huang, G., Liu, Z., van der Maaten, L. & Weinberger, K. Q. (2018), 'Densely Connected Convolutional Networks'. arXiv:1608.06993 [cs].
URL: *http://arxiv.org/abs/1608.06993*

Hwang, J., Hira, M., Chen, C., Zhang, X., Ni, Z., Sun, G., Ma, P., Huang, R., Pratap, V., Zhang, Y., Kumar, A., Yu, C.-Y., Zhu, C., Liu, C., Kahn, J., Ravanelli, M., Sun, P., Watanabe, S., Shi, Y., Tao, Y., Scheibler, R., Cornell, S., Kim, S. & Petridis, S. (2023), 'Torchaudio 2.1: Advancing speech recognition, self-supervised learning, and audio processing components for pytorch'.

James, G., Witten, D., Hastie, T., Tibshirani, R. & Taylor, J. (2023), *An Introduction to Statistical Learning with Applications in Python*, Springer Texts in Statistics, Springer, Cham.
URL: *https://link.springer.com/book/10.1007/978-3-031-38747-0*

Kahl, S., Wood, C. M., Eibl, M. & Klinck, H. (2021), 'Birdnet: A deep learning solution for avian diversity monitoring', *Ecological Informatics* **61**, 101236.
URL: *https://www.sciencedirect.com/science/article/pii/S1574954121000273*

Keren, G. & Schuller, B. (2016), 'Convolutional rnn: An enhanced model for extracting features from sequential data', pp. 3412-3419.

Kolecki, J. C. (2002), 'An introduction to tensors for students of physics and engineering'.

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2017), 'Imagenet classification with deep convolutional neural networks', *Commun. ACM* **60**(6), 84âĂŞ90.
URL: *https://doi.org/10.1145/3065386*

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.
URL: *http://ieeexplore.ieee.org/document/726791/*

Li, F.-F. & Adeli, E. (2024), 'CS231n: Deep Learning for Computer Vision'.
URL: *https://cs231n.stanford.edu/*

Lu, T., Han, B. & Yu, F. (2021), 'Detection and classification of marine mammal sounds using alexnet with transfer learning', *Ecological Informatics* **62**, 101277.
URL: *https://www.sciencedirect.com/science/article/pii/S1574954121000686*

Ma, L., Milner, B. & Smith, D. (2006), 'Acoustic environment classification', *ACM Transactions on Speech and Language Processing* **3**(2), 1–22.
URL: *https://dl.acm.org/doi/10.1145/1149290.1149292*

McCloskey, E. (2024), 'Interview with the school of biology on false killer whales audio'.

Mutanu, L., Gohil, J., Gupta, K., Wagio, P. & Kotonya, G. (2022), 'A Review of Automated Bioacoustics and General Acoustics Classification Research', *Sensors* **22**(21), 8361.
URL: *https://www.mdpi.com/1424-8220/22/21/8361*

National Oceanography Centre (2024), 'Gliders'.

NOAA Fisheries (2024), 'Mission on the high seas: Hawaiian islands cetacean and ecosystem assessment survey'.
URL: *https://www.fisheries.noaa.gov/feature-story/mission-high-seas-hawaiian-islands-cetacean-and-ecosystem-assessment-survey*

Nur Korkmaz, B., Diamant, R., Danino, G. & Testolin, A. (2023), 'Automated

detection of dolphin whistles with convolutional networks and transfer learning', *Frontiers in Artificial Intelligence* **6**.
URL: *https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2023.1099022*

Oppenheim, A. V., Schafer, R. W. & Buck, J. R. (1998), *Discrete-time signal processing*, 2 edn, Pearson, Upper Saddle River, NJ.

O'Shea, K. & Nash, R. (2015), 'An introduction to convolutional neural networks'.
URL: *https://arxiv.org/abs/1511.08458*

Palanisamy, K., Singhania, D. & Yao, A. (2020), 'Rethinking CNN models for audio classification'.
URL: *http://arxiv.org/abs/2007.11154*

Papers With Code: Trends (2024).
URL: *https://paperswithcode.com/trends*

Parsons, S. (2001), 'Identification of New Zealand bats (*Chalinolobus tuberculatus* and *Mystacina tuberculata*) in flight from analysis of echolocation calls by artificial neural networks', *Journal of Zoology* **253**(4), 447–456.
URL: *https://zslpublications.onlinelibrary.wiley.com/doi/10.1017/S0952836901000413*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., KÃűpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), 'Pytorch: An imperative style, high-performance deep learning library'.
URL: *https://arxiv.org/abs/1912.01703*

Phung, V. H. & Rhee, E. J. (2019), 'A High-Accuracy Model Average Ensemble of Convolutional Neural Networks for Classification of Cloud Image Patches on Small Datasets', *Applied Sciences* **9**(21), 4500.
URL: *https://www.mdpi.com/2076-3417/9/21/4500*

Pixabay (2016), 'Puppy, dog, pet image. free for use.'.
URL: *https://pixabay.com/photos/puppy-dog-pet-collar-dog-collar-1903313/*

Pseudorca crassidens: Baird, R.W.: The IUCN Red List of Threatened Species 2018: e.T18596A145357488 (2018).
URL: *http://dx.doi.org/10.2305/IUCN.UK.2018-2.RLTS.T18596A145357488.en*

Pytorch (2024), 'Tensors'.
URL: *https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html*

Razavian, A. S., Azizpour, H., Sullivan, J. & Carlsson, S. (2014), 'CNN Features off-the-shelf: an Astounding Baseline for Recognition'. arXiv:1403.6382 [cs].
URL: *http://arxiv.org/abs/1403.6382*

Ridgway, S. H. & Harrison, R. J. (1999), *Handbook of marine mammals*, Academic press.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C. & Fei-Fei, L. (2015), 'Imagenet large scale visual recognition challenge'.
URL: *https://arxiv.org/abs/1409.0575*

Sayigh, L., Daher, M. A., Allen, J., Gordon, H., Joyce, K., Stuhlmann, C. & Tyack, P. (2017), 'The Watkins Marine Mammal Sound Database: An online, freely accessible resource', *Proceedings of Meetings on Acoustics* **27**(1), 040013.
URL: *https://doi.org/10.1121/2.0000358*

Simonyan, K. & Zisserman, A. (2015), 'Very Deep Convolutional Networks for Large-Scale Image Recognition'. arXiv:1409.1556 [cs].
URL: *http://arxiv.org/abs/1409.1556*

Song, G., Wang, Z., Han, F. & Ding, S. (2017), Transfer Learning for Music Genre Classification, *in* Z. Shi, B. Goertzel & J. Feng, eds, 'Intelligence Science I', Vol. 510, Springer International Publishing, Cham, pp. 183–190. Series Title: IFIP Advances in Information and Communication Technology.
URL: *https://link.springer.com/10.1007/978-3-319-68121-4_19*

Sousa-Lima, R. (2013), 'Errata: Aquatic mammals, 39(1), 2013, pp. 23-53 a review and inventory of fixed autonomous recorders for passive acoustic monitoring of marine

mammals', *Aquatic Mammals* **39**(2), 205-210.
URL: *http://dx.doi.org/10.1578/AM.39.2.2013.205*

Springenberg, J. T., Dosovitskiy, A., Brox, T. & Riedmiller, M. (2015), 'Striving for simplicity: The all convolutional net'.
URL: *https://arxiv.org/abs/1412.6806*

Sturm, B. L. (2014), 'The GTZAN dataset: Its contents, its faults, their effects on evaluation, and its future use', *Journal of New Music Research* **43**(2), 147–172. arXiv:1306.1461 [cs].
URL: *http://arxiv.org/abs/1306.1461*

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. & Rabinovich, A. (2014), 'Going Deeper with Convolutions'. arXiv:1409.4842 [cs].
URL: *http://arxiv.org/abs/1409.4842*

Tan, M. & Le, Q. V. (2020), 'EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks'. arXiv:1905.11946 [cs, stat].
URL: *http://arxiv.org/abs/1905.11946*

torch.utils.data (2023).
URL: *https://pytorch.org/docs/stable/data.html*

Torchvision (2024).
URL: *https://pytorch.org/vision/stable/index.html*

Williams, B., van MerriÃńnboer, B., Dumoulin, V., Hamer, J., Triantafillou, E., Fleishman, A. B., McKown, M., Munger, J. E., Rice, A. N., Lillis, A., White, C. E., Hobbs, C. A. D., Razak, T. B., Jones, K. E. & Denton, T. (2024), 'Leveraging tropical reef, bird and unrelated sounds for superior transfer learning in marine bioacoustics'.
URL: *https://arxiv.org/abs/2404.16436*

Yadav, S. & Bhole, G. P. (2020), Handling imbalanced dataset classification in machine learning, *in* '2020 IEEE Pune Section International Conference (PuneCon)', pp. 38–43.

Yani, M., Irawan, S. & Setianingsih, C. (2019), 'Application of transfer learning using convolutional neural network method for early detection of terry's nail', *Journal of Physics: Conference Series* **1201**, 012052.

Yano, K. M., Oleson, E. M. E. M., Keating, J. L., Ballance, L. T. L. T., Hill, M. C. M. C., Bradford, A. L., Allen, A. N., Joyce, T. W., Moore, J. E. & Henry, A. E. (2018), 'Cetacean and seabird data collected during the hawaiian islands cetacean and ecosystem assessment survey (HICEAS), july - december 2017'. Publisher: United States. National Marine Fisheries Service Pacific Islands Fisheries Science Center (U.S.) BOEM Alaska Environmental Studies Program.
URL: *https://repository.library.noaa.gov/view/noaa/18660*

Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2023), *Dive into Deep Learning*, Cambridge University Press. https://D2L.ai.

Zhong, M., LeBien, J., Campos-Cerqueira, M., Dodhia, R., Lavista Ferres, J., Velev, J. P. & Aide, T. M. (2020), 'Multispecies bioacoustic classification using transfer learning of deep convolutional neural networks with pseudo-labeling', *Applied Acoustics* **166**, 107375.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0003682X20304795*

A | Python Code

This code is for my MSc Data-Intensive Analysis dissertation entitled *"Convolutional Neural Networks for the Classification of Pseudorca crassidens Passive Acoustic Data"*. The version for this dissertation can be found in https://github.com/josh-arrabaca/standrewsdissertation/releases/tag/Submission3.

To run these, please ensure the code is in the top folder, and the related dataset is in the subfolder "data", and that the audio files are in the appropriate sub-subfolders "Insular" and "Pelagic".

To make predictions, you can run at the command line: "python c_make_prediction.py filename.wav".

The flow of experiments is as follows:

1. For converting the audio files to spectrograms:
   - a_convert_wav_to_png.py
2. For applying Transfer Learning to different pretrainined CNNs. These mostly use the same code, except a different model was loaded for each experiment:
   - b_CNN_model_DenseNet.py
   - b_CNN_model_googlenet.py
   - b_CNN_model_efficientnet.py
   - b_CNN_model_ResNet.py
3. For applying the *Weighed Random Sampler* to the imbalanced dataset. Much of the code is similar to the above, except for adding the sampler library, and plotting the new distibution per batch:
   - b_CNN_model_ResNet_wrs.py

4. For hyperparameter tuning. Tuning was done first with the learning rate, then the momentum value. The code is also very similar to the above, except for the for-loop code added to train and test each of the hyperparameter values:
    - b_CNN_model_ResNet_wrs_lr.py
    - b_CNN_model_ResNet_wrs_momentum.py
5. For testing the best model with the unseen test set. As with the above, most of the code is the same, except for the metrics:
    - b_CNN_model_ResNet_best.py
6. For making predictions using the saved best model:
    - c_make_prediction.py

Code Listing A.1: code/a_convert_wav_to_png.py

```
# Import Libraries
import os.path
import numpy as np
from pydub import AudioSegment
from scipy import signal
import matplotlib.pyplot as plt

# Define the function that will convert the files
def batch_wav_to_png (folder):
    """ Goes through all subfolders in a chosen folder, and converts wav files to
    spectrogram png."""
    print("Finding files and converting wav to png...\n")

    # Create folders for the images
    if not os.path.exists(os.path.join("data", "images", "Insular")):
        os.makedirs(os.path.join("data", "images", "Insular"))
    if not os.path.exists(os.path.join("data", "images", "Pelagic")):
        os.makedirs(os.path.join("data", "images", "Pelagic"))

    # Initialise counters
    wavfile_counter = 0
    otherfile_counter = 0
    folder_counter = 0

    # Walkthrough all subfolders
    for root, dirs, files in os.walk(folder):

        for file in files:

            # Process files if they are wav
            if file.endswith((".wav")) and any(dolphin_type in str(root) for
    dolphin_type in ["Pelagic", "Insular"]):
                fileloc =  os.path.join(root, file)

                # Extract raw audio data from the wav, and "flatten" channels to only
     one channel
                sound = AudioSegment.from_wav(fileloc)
                sound = sound.set_channels(1)

                # Convert the result to ndarry, and find the sample_rate
                samples = sound.get_array_of_samples()
                samples = np.array(samples).astype(np.int16)
                sample_rate = sound.frame_rate

```

```
                # Convert to spectrogram data
                frequencies, times, spectrogram = signal.spectrogram(samples, sample_rate)

                # Plot the result without any axes or labels
                # Part of this code was provided by Dr Chrissy Fell, School of Mathematics and Statistics, University of St Andrews
                plt.pcolormesh(times, frequencies, np.log(spectrogram))
                plt.tick_params(axis='both', which='both', bottom=False, left=False, top=False,
                                labelbottom=False, labelleft=False)
                plt.rcParams["figure.figsize"] = (12,8)
                if "Insular" in str(root):
                    my_file = file + ".png"
                    plt.savefig(os.path.join(folder, "images", "Insular", my_file))
                elif "Pelagic" in str(root):
                    my_file = file + ".png"
                    plt.savefig(os.path.join(folder, "images", "Pelagic", my_file))
                plt.close()

                #Increment the counters
                wavfile_counter += 1
                if wavfile_counter % 50 == 0:
                    print (f"Processing {wavfile_counter}th wave file...")

            else:
                otherfile_counter += 1

        folder_counter += 1

    print (f"Processing done!\nConverted {wavfile_counter} .wav files to .png in {folder_counter} folders.\nThe files are located in '{folder}\images' folder.\n\nIgnored {otherfile_counter} other files.")


# Name the folder with wav files here
folder = "data"

# Run the function
batch_wav_to_png (folder)
```

Code Listing A.2: code/b_CNN_model_DenseNet.py

```
# Adapted from the original Pytorch tutorial by Sasank Chilamkurthy

# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
from sklearn import metrics


cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")


# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Densenet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required
    normalisation for Densenet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=
    data_transforms)

# Define the classes (Insular and Pelagic)
classes = dataset.classes

```

```
# Split the data into train, val and test sets
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split
     as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size
    }\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# below code is taken from
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):

    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0
```

```python

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # zero the parameter gradients
                    optimizer.zero_grad()

                    # forward
                    # track history if only in train
                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        # backward + optimize only if in training phase
                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    # statistics
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                    scheduler.step()

                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() / dataset_sizes[phase]

                print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

```

```
                # deep copy the model
                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f
    }s\n\n')
        print(f'The best val accuracy score is: {best_acc:4f}\n\n')

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# Load Densenet
model_conv = torchvision.models.densenet121(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.classifier.in_features
model_conv.classifier = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.classifier.parameters(), lr=0.001, momentum
    =0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Now train the model, and view the loss and accuracy scores
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=num_epochs)


# This part creates the classes and scores from the validation data for the metrics
```

```
y_score = []
true_classes = []
predicted_classes = []

for inputs, labels in val_dataloader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0] # Get probabilities of the positive class ('Insular') only
            y_score.append(top_proba)# predicted_classes

# Let's find the precision, recall and confusion matrix as well
print(f"Precision: {round(metrics.precision_score (true_classes, predicted_classes),5)}")
print(f"Recall: {round(metrics.recall_score(true_classes, predicted_classes),5)}")
print("Confusion matrix:\n", metrics.confusion_matrix(true_classes, predicted_classes))
```

Code Listing A.3: code/b_CNN_model_efficientnet.py

```
# Adapted from the original Pytorch tutorial by Sasank Chilamkurthy

# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
from sklearn import metrics

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")

# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([384,384]), # Minimum size needed for Efficientnet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required
     normalisation for Efficientnet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=
    data_transforms)

# Define the classes (Insular and Pelagic)
classes = dataset.classes

# Split the data into train, val and test sets
train_size = int(0.6 * len(dataset))
```

```
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split
    as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size
    }\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# below code is taken from
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):

    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
```

```
        print(f'Epoch {epoch}/{num_epochs - 1}')
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
```

```
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s\n\n')
        print(f'The best val accuracy score is: {best_acc:4f}\n\n')

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# Loading efficientnet
model_conv = torchvision.models.efficientnet_v2_s(weights='IMAGENET1K_V1')


# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.classifier[1].in_features
model_conv.classifier[1] = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.classifier[1].parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Now train the model, and view the loss and accuracy scores
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=num_epochs)

# This part creates the classes and scores from the validation data for the metrics
y_score = []
true_classes = []
```

```
predicted_classes = []

for inputs, labels in val_dataloader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each
    output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0] # Get probabilities of the positive
    class ('Insular') only
            y_score.append(top_proba)# predicted_classes

# Let's find the precision, recall and confusion matrix as well
print(f"Precision: {round(metrics.precision_score (true_classes, predicted_classes)
    ,5)}")
print(f"Recall: {round(metrics.recall_score(true_classes, predicted_classes),5)}")
print("Confusion matrix:\n", metrics.confusion_matrix(true_classes, predicted_classes
    ))
```

Code Listing A.4: code/b_CNN_model_ResNet.py

```
# Adapted from the original Pytorch tutorial by Sasank Chilamkurthy

# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
from sklearn import metrics
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import precision_recall_curve

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")


# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Resnet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required normalisation for Resnet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=data_transforms)

# Define the classes (Insular and Pelagic)
```

```
classes = dataset.classes

# Split the data into train, val and test sets
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size}\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# below code is taken from
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):

    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

```

```
        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # zero the parameter gradients
                    optimizer.zero_grad()

                    # forward
                    # track history if only in train
                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        # backward + optimize only if in training phase
                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    # statistics
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                    scheduler.step()

                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() / dataset_sizes[phase]

```

```
                print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

                # deep copy the model
                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s\n\n')
        print(f'The best val accuracy score is: {best_acc:4f}\n\n')

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# Loading ResNet
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)


# Now train the model, and view the loss and accuracy scores
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=num_epochs)

```

```python
# This part creates the classes and scores from the validation data for the plots
y_score = []
true_classes = []
predicted_classes = []

for inputs, labels in val_dataloader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0] # Get probabilities of the positive class ('Insular') only
            y_score.append(top_proba)# predicted_classes

# Let's also check the Precision-Recall Curve, and the Threshold Plots
# These plots are saved locally as image files
display = PrecisionRecallDisplay.from_predictions(true_classes,
                                                  y_score,
                                                  name="CNN",
                                                  plot_chance_level=True)
_ = display.ax_.set_title("2-class Precision-Recall curve")
plt.ylim([0,1])
plt.savefig("precision-recall-curve.png", bbox_inches='tight')
plt.close()

precision, recall, thresholds = precision_recall_curve(true_classes, y_score)
plt.plot(thresholds, precision[:-1], 'b--', label='Precision')
plt.plot(thresholds, recall[:-1], 'r--', label='Recall')
plt.xlabel('Threshold')
plt.legend(loc='lower left')
plt.ylim([0,1])
plt.title("Precision-Recall-Threshold Plot")
plt.savefig("precision-recall-threshold.png", bbox_inches='tight')
```

```
# Let's find the precision, recall and confusion matrix as well
print(f"Precision: {round(metrics.precision_score (true_classes, predicted_classes),5)}")
print(f"Recall: {round(metrics.recall_score(true_classes, predicted_classes),5)}")
print("Confusion matrix:\n", metrics.confusion_matrix(true_classes, predicted_classes))
```

Code Listing A.5: code/b_CNN_model_ResNet_wrs.py

```
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import precision_recall_curve
from torch.utils.data import DataLoader
import pandas as pd
from torch.utils.data import WeightedRandomSampler

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")


# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Densenet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required
    normalisation for Densenet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=
    data_transforms)

# Define the classes (Insular and Pelagic)
```

```
classes = dataset.classes

# Split the data into train, val and test sets
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size}\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
# train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)


# Define a function to visualise the class distributions per batch
def plot_classes(dataloader):
        batch_num_counts = []
        cur_batch = 0
        class_0_counts = []
        class_1_counts = []
        for i in range(len(train_dataloader)):
                my_inputs, my_labels = next(iter(train_dataloader))
                this_class_0_count = 0
                this_class_1_count = 0
                for label in my_labels:
                        if label.cpu().numpy() == 0:
                                this_class_0_count+=1
                        elif label.cpu().numpy() == 1:
                                this_class_1_count+=1
```

```
                class_0_counts.append(this_class_0_count)
                class_1_counts.append(this_class_1_count)
                batch_num_counts.append(cur_batch)
                cur_batch += 1

        df = pd.DataFrame({'Insular': class_0_counts,
                           'Pelagic': class_1_counts})

        df.plot(kind="bar",
                title="Weighed Distribution of Classes per Batch",
                xlabel="Batch Number",
                ylabel="Images per Batch")

        plt.savefig("weighed_distribution_pre_optim.png", bbox_inches='tight')
        plt.close()
        print(df)

# # Visualise the distribution
# plot_classes(train_dataloader, "Imbalanced_Distribution_of_Classes_per_Batch")

# Define function to create a balanced sampler
# The following code was adapted from Vivek Maskara
# https://www.maskaravivek.com/post/pytorch-weighted-random-sampler/
def balanced_sampler(full_dataset, train_dataset):
    # Find number of samples per class
    y_train_indices = train_dataset.indices
    y_train = [full_dataset.targets[i] for i in y_train_indices]
    class_sample_count = np.array([len(np.where(y_train == t)[0]) for t in np.unique(
    y_train)])

    # Find weights per class
    weight = 1. / class_sample_count
    samples_weight = np.array([weight[t] for t in y_train])
    samples_weight = torch.from_numpy(samples_weight)

    # Define sampler
    sampler = WeightedRandomSampler(samples_weight.type('torch.DoubleTensor'), len(
    samples_weight))

    return sampler

# Create a balanced sampler
sampler = balanced_sampler(dataset, train_dataset)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler)

# Visualise the balanced dataset per batch
```

```
plot_classes(train_dataloader)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# Adapted from the original Pytorch tutorial by Sasank Chilamkurthy
train_losses = []
val_losses = []
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):

    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # zero the parameter gradients
                    optimizer.zero_grad()

                    # forward
```

```
                    # track history if only in train
                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)

                        # backward + optimize only if in training phase
                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    # statistics
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                    scheduler.step()

                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() / dataset_sizes[phase]

                if phase == 'train':
                     train_losses.append(epoch_loss)
                else:
                     val_losses.append(epoch_loss)
                print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')



                # deep copy the model
                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s\n\n')
        print(f'The best val accuracy score is: {best_acc:4f}\n\n')

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# We load ResNet since this had the best base accuracy compared to the other CNNs
```

```
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Now we train the model
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=num_epochs)


# This part creates the classes and scores from the training data for the plots
plt.plot(train_losses, 'b', label='Training Loss')
plt.plot(val_losses, 'r--', label='Validation Loss')
plt.legend(loc='upper right')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title("Model Loss over Epochs")
plt.savefig("losses.png", bbox_inches='tight')
plt.close()


y_score = []
true_classes = []
predicted_classes = []

for inputs, labels in train_dataloader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
```

```
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0] # Get probabilities of the positive class ('Insular') only
            y_score.append(top_proba)# predicted_classes

# Let's also check the Precision-Recall Curve, and the Threshold Plots after using the weighed sampler
# These plots are saved locally as image files
display = PrecisionRecallDisplay.from_predictions(true_classes,
                                                  y_score,
                                                  name="CNN",
                                                  plot_chance_level=True)
_ = display.ax_.set_title("2-class Precision-Recall curve (Weighed Distribution)")
plt.ylim([0,1])
plt.savefig("precision-recall-curve-weighed.png", bbox_inches='tight')
plt.close()

precision, recall, thresholds = precision_recall_curve(true_classes, y_score)
plt.plot(thresholds, precision[:-1], 'b--', label='Precision')
plt.plot(thresholds, recall[:-1], 'r--', label='Recall')
plt.xlabel('Threshold')
plt.legend(loc='lower left')
plt.ylim([0,1])
plt.title("Precision-Recall-Threshold Plot (Weighed Distribution)")
plt.savefig("precision-recall-threshold-weighed.png", bbox_inches='tight')
plt.close()
```

Code Listing A.6: code/b_CNN_model_ResNet_wrs_lr.py

```
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
import pandas as pd
from torch.utils.data import WeightedRandomSampler

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")


# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Densenet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required normalisation for Densenet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=data_transforms)

# Define the classes (Insular and Pelagic)
classes = dataset.classes

# Split the data into train, val and test sets
```

```
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size}\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)


# Define function to create a balanced sampler
# https://www.maskaravivek.com/post/pytorch-weighted-random-sampler/
def balanced_sampler(full_dataset, train_dataset):
    # Find number of samples per class
    y_train_indices = train_dataset.indices
    y_train = [full_dataset.targets[i] for i in y_train_indices]
    class_sample_count = np.array([len(np.where(y_train == t)[0]) for t in np.unique(y_train)])

    # Find weights per class
    weight = 1. / class_sample_count
    samples_weight = np.array([weight[t] for t in y_train])
    samples_weight = torch.from_numpy(samples_weight)

    # Define sampler
    sampler = WeightedRandomSampler(samples_weight.type('torch.DoubleTensor'), len(samples_weight))

```

```
    return sampler

# Create a balanced sampler
sampler = balanced_sampler(dataset, train_dataset)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# Originally taken from the Pytorch tutorial by Sasank Chilamkurthy
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    train_losses = []
    val_losses = []
    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # zero the parameter gradients
```

```
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            if phase == 'train':
                 train_losses.append(epoch_loss)
            else:
                 val_losses.append(epoch_loss)
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                torch.save(model.state_dict(), best_model_params_path)

        print()

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s\n\n')
    print(f'The best val accuracy score is: {best_acc:4f}\n\n')

    # Plot the losses
    plt.plot(train_losses, 'b', label='Training Loss')
    plt.plot(val_losses, 'r--', label='Validation Loss')
    plt.legend(loc='upper right')
    plt.xlabel('Epoch')
```

```
        plt.ylabel('Loss')
        plt.title("Model Loss over Epochs for LR=" + str(lr))
        plt.savefig("losses_"+str(lr)+".png", bbox_inches='tight')
        plt.close()

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# We load Resnet since this had the best base accuracy
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Here we try some different learning rates
learning_rates = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1]

for learning_rate in learning_rates:
    lr = learning_rate
    optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=lr, momentum=0.9)

    # Decay LR by a factor of 0.1 every 7 epochs
    exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

    # Now train the model, and see the loss and accuracy scores for the different
    learning rates
    model_conv = train_model(model_conv, criterion, optimizer_conv,
                            exp_lr_scheduler, num_epochs=num_epochs)
```

Code Listing A.7: code/b_CNN_model_ResNet_wrs_momentum.py

```
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import DataLoader
import pandas as pd
from torch.utils.data import WeightedRandomSampler
from sklearn import metrics

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")

# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Densenet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required
    normalisation for Densenet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=
    data_transforms)

# Define the classes (Insular and Pelagic)
classes = dataset.classes

# Split the data into train, val and test sets
```

```
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split
     as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size
    }\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# Define function to create a balanced sampler
# https://www.maskaravivek.com/post/pytorch-weighted-random-sampler/
def balanced_sampler(full_dataset, train_dataset):
    # Find number of samples per class
    y_train_indices = train_dataset.indices
    y_train = [full_dataset.targets[i] for i in y_train_indices]
    class_sample_count = np.array([len(np.where(y_train == t)[0]) for t in np.unique(
    y_train)])

    # Find weights per class
    weight = 1. / class_sample_count
    samples_weight = np.array([weight[t] for t in y_train])
    samples_weight = torch.from_numpy(samples_weight)

    # Define sampler
    sampler = WeightedRandomSampler(samples_weight.type('torch.DoubleTensor'), len(
    samples_weight))

    return sampler
```

```

# Create a balanced sampler
sampler = balanced_sampler(dataset, train_dataset)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# Originally taken from the Pytorch tutorial by Sasank Chilamkurthy
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    train_losses = []
    val_losses = []
    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # zero the parameter gradients
                    optimizer.zero_grad()
```

```python

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            if phase == 'train':
                train_losses.append(epoch_loss)
            else:
                val_losses.append(epoch_loss)
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                torch.save(model.state_dict(), best_model_params_path)

        print()

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s\n\n')
    print(f'The best val accuracy score is: {best_acc:4f}\n\n')

    # Plot the losses
    plt.plot(train_losses, 'b', label='Training Loss')
    plt.plot(val_losses, 'r--', label='Validation Loss')
    plt.legend(loc='upper right')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
```

```
        plt.title("Model Loss over Epochs for momentum=" + str(momentum))
        plt.savefig("losses_"+str(momentum)+".png", bbox_inches='tight')
        plt.close()

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# Load the model
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Here we try different momentum values with the lr set to the best lr
momentum_values = [0.90, 0.95, 0.97, 0.99]

for value in momentum_values:
    momentum = value
    optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.01, momentum=momentum)

    # Decay LR by a factor of 0.1 every 7 epochs
    exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

    # Now train the model and see the loss and accuracy scores
    model_conv = train_model(model_conv, criterion, optimizer_conv,
                             exp_lr_scheduler, num_epochs=num_epochs)

# This part creates the classes and scores from the validation data for the metrics
y_score = []
true_classes = []
predicted_classes = []

for inputs, labels in val_dataloader:
    inputs = inputs.to(device)
```

```
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each
    output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0] # Get probabilities of the positive
    class ('Insular') only
            y_score.append(top_proba)# predicted_classes

# Let's find the precision, recall and confusion matrix as well
print(f"Precision: {round(metrics.precision_score (true_classes, predicted_classes)
    ,5)}")
print(f"Recall: {round(metrics.recall_score(true_classes, predicted_classes),5)}")
print("Confusion matrix:\n", metrics.confusion_matrix(true_classes, predicted_classes
    ))
```

Code Listing A.8: code/b_CNN_model_ResNet_best.py

```
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory
from sklearn import metrics
from torch.utils.data import DataLoader
import pandas as pd
from torch.utils.data import WeightedRandomSampler

cudnn.benchmark = True
plt.ion()   # interactive mode

# Set 'random' seed
torch.manual_seed(220029955)

# Welcome message
print("Welcome! We will train the last layer of a pre-trained CNN model.\n")

# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Densenet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required normalisation for Densenet
    ])

# Get the dataset from the images created from the wav files
dataset = datasets.ImageFolder(os.path.join("data", "images"), transform=data_transforms)

# Define the classes (Insular and Pelagic)
classes = dataset.classes

# Split the data into train, val and test sets
```

```
train_size = int(0.6 * len(dataset))
val_size = int((len(dataset) - train_size) / 2)
test_size = val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, val_size, test_size])
print(f"The dataset consists of {train_size + val_size + test_size} datapoints, split
     as follows:")
print(f"Train set: {train_size} \nValidation set: {val_size} \nTest size: {test_size
    }\n")

# Define the device to be used for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Define the batch size and number of epochs based on the device
if str(device) == "cuda:0":
    batch_size = 64
    num_epochs = 24
else:
    batch_size = 20
    num_epochs = 3

# Dataloaders
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# Define function to create a balanced sampler
# https://www.maskaravivek.com/post/pytorch-weighted-random-sampler/
def balanced_sampler(full_dataset, train_dataset):
    # Find number of samples per class
    y_train_indices = train_dataset.indices
    y_train = [full_dataset.targets[i] for i in y_train_indices]
    class_sample_count = np.array([len(np.where(y_train == t)[0]) for t in np.unique(
    y_train)])

    # Find weights per class
    weight = 1. / class_sample_count
    samples_weight = np.array([weight[t] for t in y_train])
    samples_weight = torch.from_numpy(samples_weight)

    # Define sampler
    sampler = WeightedRandomSampler(samples_weight.type('torch.DoubleTensor'), len(
    samples_weight))

    return sampler

```

```
# Create a balanced sampler
sampler = balanced_sampler(dataset, train_dataset)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler)

# # Visualise the balanced dataset per batch
# plot_classes(train_dataloader)

dataloaders = {"train": train_dataloader,
               "val": val_dataloader}

dataset_sizes = {"train": len (train_dataset),
                 "val": len(test_dataset)}


# Originally taken from the {ytorch tutorial by Sasank Chilamkurthy
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    train_losses = []
    val_losses = []
    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, 'best_model_params.pt')

        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0

        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
                else:
                    model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

```

```
                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            if phase == 'train':
                 train_losses.append(epoch_loss)
            else:
                 val_losses.append(epoch_loss)
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')



            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                torch.save(model.state_dict(), best_model_params_path)

        print()

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f
}s\n\n')
    print(f'The best val accuracy score is: {best_acc:4f}\n\n')

    # Plot the losses
    plt.plot(train_losses, 'b', label='Training Loss')
```

```
        plt.plot(val_losses, 'r--', label='Validation Loss')
        plt.legend(loc='upper right')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title("Model Loss over Epochs for LR=" + str(lr))
        plt.savefig("losses_"+str(lr)+".png", bbox_inches='tight')
        plt.close()

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    return model


# Load the model
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# This part does the training on the final layer only
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Set the learning rate and momentum to the best ones found during hyperparameter tuning
lr = 0.01
momentum = 0.95
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=lr, momentum=momentum)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Now train the model!
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=num_epochs)

# This part saves entire model
torch.save(model_conv, "best-model-resnet.pth")

# This part tests the model
y_score = []
```

```
true_classes = []
predicted_classes = []

for inputs, labels in test_dataloader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # This part 'flattens' the tensor into a list
    labels = labels.cpu().numpy().tolist()
    true_classes.extend(labels)

    with torch.no_grad():
        model_conv.eval()
        output = model_conv(inputs)

        for each_output in output:
            predicted_class = each_output.cpu().data.numpy().argmax() # Numpify each output
            predicted_classes.append(predicted_class) # Conactenate

            p = torch.nn.functional.softmax(output, dim=1) # Get probabilities
            top_proba = p.cpu().numpy()[0][0]
            y_score.append(top_proba)# predicted_classes


# Now let's check the metrics
print("\nNow we'll test using the unseen test set. The following are the results:")
print(f"Accuracy: {round(metrics.accuracy_score (true_classes, predicted_classes),5)}")
print(f"Precision: {round(metrics.precision_score (true_classes, predicted_classes),5)}")
print(f"Recall: {round(metrics.recall_score(true_classes, predicted_classes),5)}")
print(f"F1: {round(metrics.f1_score(true_classes, predicted_classes),5)}")
print("Confusion matrix:\n", metrics.confusion_matrix(true_classes, predicted_classes))
```

Code Listing A.9: code/c_make_prediction.py

```
# Import Libraries
import torch
import os.path
from scipy import signal
from pydub import AudioSegment
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import sys
from torchvision import transforms
import warnings
warnings.filterwarnings("ignore")

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'The device being used is: {device}\n')

# Load the best model
if str(device) == "cuda:0":
    model = torch.load('best-model-resnet.pth')
else:
    model = torch.load('best-model-resnet.pth', map_location=torch.device('cpu'))
model.eval()

# Define function for converting single wav to png file
def new_audio_to_png (wavfile):
    sound = AudioSegment.from_wav(wavfile)
    sound = sound.set_channels(1)

    # Convert the result to ndarry, and find the sample_rate
    samples = sound.get_array_of_samples()
    samples = np.array(samples).astype(np.int16)
    sample_rate = sound.frame_rate

    # Convert to spectrogram data
    frequencies, times, spectrogram = signal.spectrogram(samples, sample_rate)

    # Plot the result without any axes or labels
    plt.pcolormesh(times, frequencies, np.log(spectrogram))
    plt.tick_params(axis='both', which='both', bottom=False, left=False, top=False,
                    labelbottom=False, labelleft=False)
    plt.rcParams["figure.figsize"] = (12,8)
    plt.savefig("spectrogram_to_be_predicted.png", bbox_inches='tight')
    plt.close()

```

```
file_name = sys.argv[1]
new_audio_to_png(os.path.join(file_name))

# Define the transforms needed
data_transforms = transforms.Compose([
        transforms.Resize([224,224]), # Minimum size needed for Resnet
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # Required normalisation for Resnet
    ])

# Define classes
classes = ['Insular', 'Pelagic']

# Convert the image to a tensor
img = Image.open("spectrogram_to_be_predicted.png").convert('RGB')
new_tensor = data_transforms(img)
new_tensor = new_tensor.unsqueeze(0)

# Move the data to the GPU if using gpu
if str(device) == "cuda:0":
    new_tensor = new_tensor.to(device)

# Get prediction
with torch.no_grad():
      output = model(new_tensor)
      index = output.data.cpu().numpy().argmax()
      class_name = classes[index]

# Get probabilities of the prediction
p = torch.nn.functional.softmax(output, dim=1)
top_proba = p.cpu().numpy()[0][index]

# Print the results
print("output: ", output)
print("index:", index)
print("class name:", class_name)
print("probabilities:", p)
print(f"\nThere's a {round((top_proba*100),2)} % chance that this audio file belongs to the {class_name} group.")
```