Predicting the Sounds of Seattle birds using Neural Networks

Abstract:

This research aims to classify bird species based on their recorded chirping sounds using the Birdcall competition dataset from the Xeno-Canto[1] bird sound archive by leveraging neural networks to predict species based on their vocalizations. The study selected 12 bird species and developed both a binary classification model for two species and a multi-class classification model encompassing all 12 species. The project explored various activation functions and conducted extensive hyperparameter tuning to optimize model performance. Additionally, transfer learning was employed by adapting a pre-trained neural network model to the specific dataset, enhancing the accuracy of the classification tasks.

The classification models were tested using external raw sound clips, which were converted to spectrograms for analysis. Predictions were made using the multi-class model, with justifications provided through spectrogram visualizations. Challenges encountered included limited recording volumes and difficulty in distinguishing between similar bird calls. Despite these obstacles, the neural network demonstrated satisfactory performance. The project underscores the potential of neural networks in bioacoustics classification while highlighting areas for improvement, such as advanced data augmentation techniques and alternative machine learning models, offering a comprehensive evaluation of the research conducted.

Introduction:

This study embarks on the classification of bird sounds through machine learning techniques, with a dual focus. Initially, it delves into binary classification, distinguishing between the unique vocalizations of two specific bird species: norfli and wesmea. Subsequently, it expands its scope to encompass a multiclass classification challenge, aimed at categorizing recordings from 12 diverse bird species native to the Seattle region. The dataset under scrutiny is sourced from the Bird Call competition, a repository derived from the Xeno-Canto site[1], renowned for its extensive bird sound archives. The primary objective of this investigation lies in implementing a robust methodology for effectively categorizing birds based on their distinctive vocalizations. Notably, the dataset's inherent variability in audio recording lengths presents an intriguing challenge. To navigate this complexity and achieve precise classification, neural networks serve as the cornerstone, tasked with accurately delineating the distinct classes corresponding to each bird species. This research endeavor focuses on classifying bird vocalizations, commonly referred to as avian sounds.

Theoretical Background:

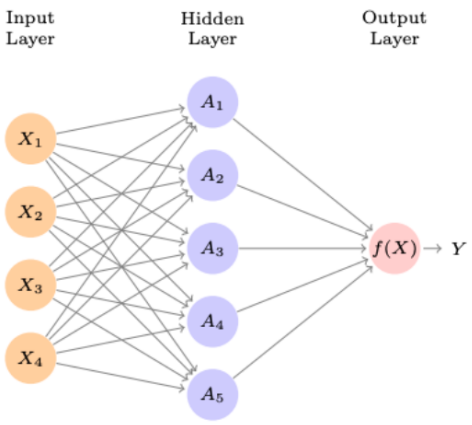
Deep Learning:

Deep learning is a subfield of machine learning that focuses on training deep neural networks composed of multiple layers of abstraction. These networks mimic the interconnected structure of

the human brain's neurons, enabling them to learn hierarchical representations of data. Deep learning has achieved remarkable success in various domains, such as computer vision, natural language processing, and speech recognition, due to its ability to model complex, non-linear relationships. Key to this capability are non-linear activation functions, which allow the networks to capture intricate interactions within data. The high dimensionality of deep learning models enables them to handle datasets with numerous features, addressing complex problems that traditional algorithms struggle with. Hyperparameters, such as learning rate, batch size, number of layers, and activation functions, play a critical role in the performance of deep learning models and require careful tuning to optimize results.

Neural Networks:

Neural networks are the foundational models in deep learning, comprising interconnected layers of neurons. These neurons process input data, apply weights and biases, and pass the results through activation functions to produce outputs. Neural networks are particularly suitable for tasks involving complex relationships between inputs and outputs, such as image and speech recognition. They offer the advantage of learning from large datasets and generalizing well to new data. However, training neural networks can be computationally intensive and they are prone to overfitting, especially in complex models. Key hyperparameters for neural networks include the learning rate, batch size, number of layers, number of neurons per layer, and the choice of activation functions. In order to calculate manually Fig1.



$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

$$A_k = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid activation

Fig1. Neural network example and Equations

Convolutional Neural Networks (CNNs):

CNNs are a specialized type of neural network designed for processing grid-like data structures, such as images and audio spectrograms. They employ convolutional layers to extract spatial

features from the input data and pooling layers to reduce dimensionality and retain essential information. CNNs have revolutionized fields like image classification and object detection due to their ability to capture spatial hierarchies in data. Despite their effectiveness, CNNs require large amounts of labeled data for training and significant computational resources. Key hyperparameters for CNNs include filter size, number of filters, stride, padding, and pooling size.

Data Augmentation:

Data augmentation is a technique used to artificially increase the diversity and quantity of training data by applying various transformations, such as rotations, translations, flips, and adjustments to contrast and brightness. This process enhances the robustness and generalization capabilities of machine learning models, particularly when the available dataset is limited. Data augmentation helps reduce overfitting and improves model performance by providing more varied training examples. However, it also introduces computational overhead and may distort the underlying data distribution if not applied judiciously.

Preprocessing Steps:

Preprocessing involves a series of operations performed on raw data to prepare it for input into machine learning models. Common preprocessing steps include data cleaning to handle missing or erroneous values, feature scaling to normalize the range of input features, and encoding categorical variables into numerical formats. Effective preprocessing ensures that the data is in a suitable format for learning, thereby enhancing model performance and interpretability. Hyperparameters related to preprocessing include the choice of scaling method and the encoding strategy for categorical variables.

Fourier Transforms and Spectrograms:

Fourier transforms are mathematical tools used to analyze the frequency components of signals. They decompose signals into constituent sinusoidal components, providing insights into the underlying frequency structure. Spectrograms, derived from Fourier transforms, offer a time-frequency representation of signals, making them particularly useful for analyzing audio signals. Spectrograms visually represent how the intensity of different frequencies in a signal changes over time, enabling detailed analysis of time-varying phenomena.

One-hot Encoding:

One-hot encoding is a technique for representing categorical variables as binary vectors. Each category is assigned a unique binary vector with one element set to 1 (active) and all others set to 0 (inactive). This encoding method is essential for feeding categorical data into machine learning

models, ensuring that the models can effectively interpret and learn from these variables. One-hot encoding is particularly useful for models that cannot inherently handle ordinal relationships among categories.

Low-Dimensional Embeddings:

Low-dimensional embeddings represent high-dimensional data in a lower-dimensional space while preserving significant relationships between data points. Techniques such as Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are commonly used for dimensionality reduction and visualization. Low-dimensional embeddings facilitate the exploration and analysis of complex datasets by highlighting underlying patterns and structures. Hyperparameters for these techniques include the number of components (for PCA) and perplexity (for t-SNE).

Recurrent Neural Networks (RNNs):

RNNs are designed to handle sequential data, where the order of inputs is crucial. Unlike feedforward neural networks, RNNs maintain an internal state that captures information from previous time steps, allowing them to model temporal dependencies. RNNs are well-suited for tasks such as time series prediction, natural language processing, and speech recognition. However, they face challenges such as vanishing and exploding gradients, which can affect training stability. Key hyperparameters for RNNs include the number of layers, size of the hidden state, dropout rate, and the choice between different RNN architectures (such as LSTM or GRU).

Deep Learning in Practice:

Deep learning is widely applied in domains requiring the analysis of large-scale datasets with complex patterns. Its success in fields such as computer vision, natural language processing, and biomedical informatics is attributed to its ability to automatically learn hierarchical representations from raw data. Practical considerations in deep learning include hyperparameter tuning, model selection, and the use of regularization techniques to prevent overfitting. Transfer learning, where pre-trained models are fine-tuned for specific tasks, is a common practice to leverage existing knowledge and reduce training time. Despite its advantages, deep learning also presents challenges, including the need for substantial computational resources, expertise in model design and optimization, and addressing ethical considerations related to data privacy and bias. Key hyperparameters in deep learning practice include the number of epochs, batch size, learning rate, and model architecture.

Methodology:

The data collected from the Xeno-Canto bird sound archive[1] contains 12 bird species sounds. These sounds are converted into spectrogram images . These all images are used for further processing in the study.

Binary Classification:

Data Preparation

The dataset comprises spectrogram images of bird calls from two species: Northern Flicker and Western Meadowlark. The spectrograms, representing the audio data, were standardized to a size of 343x256 pixels in grayscale format. The dataset was split into training and testing sets using an 80-20 split ratio to ensure the models' evaluation on unseen data. Standardization was applied to normalize the input data, ensuring that the neural network models receive data in a consistent format.

Model Architectures

Six different convolutional neural network (CNN) architectures were designed for binary classification of the bird species. Both models share a common base structure but differ in their activation functions and the incorporation of dropout layers to prevent overfitting.

Model 1: Basic CNN without Dropout

Input Layer: The model accepts spectrogram images with dimensions 343x256 pixels and a single channel.

Convolutional Layers: The architecture includes two convolutional layers. The first layer has 32 filters, and the second layer has 64 filters. Each convolutional layer uses a kernel size of 3x3 and ReLU activation functions to capture spatial features in the spectrogram images.

Max-Pooling Layers: Following each convolutional layer, a max-pooling layer with a pool size of 2x2 is used to reduce the dimensionality and computational complexity, effectively down-sampling the feature maps.

Flatten Layer: After the convolutional and pooling layers, a flatten layer converts the 2D feature maps into a 1D vector, preparing the data for the fully connected layers.

Dense Layers: A fully connected layer with 256 neurons and ReLU activation is used to learn high-level features. The final dense layer has a single neuron with a sigmoid activation function for binary classification.

Model 2: CNN with Dropout Layers

Input Layer: Identical to Model 1, the input layer accepts 343x256 pixel spectrogram images in grayscale.

Convolutional Layers: Similar to Model 1, this model includes two convolutional layers with 32 and 64 filters, respectively, using a kernel size of 3x3 and ReLU activation.

Max-Pooling and Dropout Layers: Each convolutional layer is followed by a max-pooling layer (2x2 pool size) and a dropout layer. The dropout layers, with a dropout rate of 25%, randomly set a fraction of input units to zero during training, which helps prevent overfitting by introducing regularization.

Flatten Layer: Converts the 2D feature maps into a 1D vector for the dense layers.

Dense Layers: Includes a fully connected layer with 256 neurons and ReLU activation, followed by another dropout layer with a dropout rate of 25%. The final layer is a dense layer with one neuron and a sigmoid activation function for binary classification.

Model 3: CNN with Tanh Activation (Without Dropout)

Input Layer: Accepts spectrogram images with dimensions of 343x256 pixels and a single channel.

Convolutional Layers: Two convolutional layers: The first layer has 32 filters with a kernel size of 3x3 and uses the `tanh` activation function. The second layer has 64 filters with a kernel size of 3x3 and uses the `tanh` activation function.

Max-Pooling Layers: Each convolutional layer is followed by a max-pooling layer with a pool size of 2x2.

Flatten Layer: Converts the 2D feature maps into a 1D vector.

Dense Layer:A fully connected layer with 256 neurons and `tanh` activation.

Output Layer:A dense layer with 1 neuron and a `sigmoid` activation function for binary classification.

Compilation and Training:The model is compiled using the Adam optimizer and binary cross-entropy loss, and trained for 10 epochs.

Evaluation:The model's accuracy and loss are evaluated on the test set.

Model 4: CNN with Tanh Activation (With Dropout)

Input Layer:Identical to Model 3.

Convolutional Layers:Two convolutional layers with `tanh` activation.

Max-Pooling and Dropout Layers:Each convolutional layer is followed by a max-pooling layer with a pool size of 2x2 and a dropout layer with a dropout rate of 25%.

Flatten Layer: Converts the 2D feature maps into a 1D vector.

Dense Layer:A fully connected layer with 256 neurons and `tanh` activation, followed by a dropout layer with a dropout rate of 25%.

Output Layer:A dense layer with 1 neuron and a `sigmoid` activation function for binary classification.

Compilation and Training: The model is compiled using the Adam optimizer and binary cross-entropy loss, and trained for 10 epochs.

Evaluation:The model's accuracy and loss are evaluated on the test set.

Model 5: CNN with ELU Activation (Without Dropout)

Input Layer:Accepts spectrogram images with dimensions of 343x256 pixels and a single channel.

Convolutional Layers:Two convolutional layers:The first layer has 32 filters with a kernel size of 3x3 and uses the `elu` activation function.The second layer has 64 filters with a kernel size of 3x3 and uses the `elu` activation function.

Max-Pooling Layers: Each convolutional layer is followed by a max-pooling layer with a pool size of 2x2.

Flatten Layer: Converts the 2D feature maps into a 1D vector.

Dense Layer:A fully connected layer with 256 neurons and `elu` activation.

Output Layer:A dense layer with 1 neuron and a `sigmoid` activation function for binary classification.

Compilation and Training:The model is compiled using the Adam optimizer and binary cross-entropy loss, and trained for 10 epochs.

Evaluation:The model's accuracy and loss are evaluated on the test set.

Model 6: CNN with ELU Activation (With Dropout)

Input Layer: Identical to Model 5.

Convolutional Layers:Two convolutional layers with `elu` activation.

Max-Pooling and Dropout Layers: Each convolutional layer is followed by a max-pooling layer with a pool size of 2x2 and a dropout layer with a dropout rate of 25%.

Flatten Layer:Converts the 2D feature maps into a 1D vector.

Dense Layer:A fully connected layer with 256 neurons and `elu` activation, followed by a dropout layer with a dropout rate of 25%.

Output Layer:A dense layer with 1 neuron and a `sigmoid` activation function for binary classification.

Compilation and Training: The model is compiled using the Adam optimizer and binary cross-entropy loss, and trained for 10 epochs.

Evaluation: The model's accuracy and loss are evaluated on the test set.

Model Compilation and Training

All the models were compiled using the Adam optimizer and binary cross entropy as the loss function. This setup ensures efficient training and convergence. The models were trained for 10 epochs, using a batch size of 32. Training involved feeding the standardized and preprocessed spectrogram images into the models and adjusting the weights through back propagation. After training, the models were evaluated on the test dataset to determine their accuracy.

Visualization

To visualize the learning process, the accuracy and loss metrics for both training and validation sets were plotted over the epochs. These plots help in understanding the model's performance and the impact of dropout layers on reducing overfitting.

Multi-class classification:

Data Preparation

The dataset consists of spectrogram images of bird calls from 12 different species. The spectrograms were standardized to a uniform size of 343x256 pixels in grayscale format to ensure consistency across the dataset. The data was split into training and testing sets with an 80-20 split ratio. This division ensures that the models can be evaluated on unseen data, providing an accurate measure of their performance. The input data was standardized to normalize the values, which aids in the efficient training of the neural network models.

Model Architectures

Two distinct convolutional neural network (CNN) architectures were designed for the multi-class classification task. Both models aim to classify the bird species from the spectrogram images, but they differ in the inclusion and configuration of dropout layers to address overfitting.

Model 1: Basic CNN with Dropout

Input Layer: The model accepts spectrogram images with dimensions of 343x256 pixels and a single channel.

Convolutional Layers: The architecture includes two convolutional layers. The first layer consists of 32 filters, and the second layer consists of 64 filters. Both layers use a kernel size of 3x3 and ReLU activation functions to capture spatial features in the spectrogram images.

Max-Pooling Layers: Each convolutional layer is followed by a max-pooling layer with a pool size of 2x2, which reduces the dimensionality and computational complexity by down-sampling the feature maps.

Flatten Layer: After the convolutional and pooling layers, a flatten layer converts the 2D feature maps into a 1D vector, preparing the data for the fully connected layers.

Dense Layers: A fully connected (dense) layer with 256 neurons and ReLU activation is used to learn high-level features. To mitigate overfitting, a dropout layer with a dropout rate of 50% is included after the dense layer.

Output Layer: The final dense layer has neurons equal to the number of classes (12 species) with a softmax activation function, which outputs a probability distribution over the classes.

Model 2: CNN with Dropout after Each Convolutional Layer

Input Layer: Identical to Model 1, the input layer accepts 343x256 pixel spectrogram images in grayscale.

Convolutional Layers: This model also includes two convolutional layers with 32 and 64 filters, respectively, using a kernel size of 3x3 and ReLU activation.

Max-Pooling and Dropout Layers: After each convolutional layer, a max-pooling layer (2x2 pool size) is followed by a dropout layer with a dropout rate of 25%. These dropout layers randomly set

a fraction of input units to zero during training, which helps prevent overfitting by introducing regularization.

Flatten Layer: Converts the 2D feature maps into a 1D vector for the dense layers.

Dense Layers: Includes a fully connected layer with 256 neurons and ReLU activation, followed by a dropout layer with a dropout rate of 50%. The final layer is a dense layer with neurons equal to the number of classes, using a softmax activation function for multi-class classification.

Model Compilation and Training:

Both models were compiled using the Adam optimizer, chosen for its efficiency and adaptability. The loss function used was categorical cross entropy, appropriate for multi-class classification tasks. The models were trained on the training dataset for 10 epochs. The training involved adjusting the weights of the network through backpropagation to minimize the loss function. A batch size of 32 was used to process the data in manageable chunks, balancing computational efficiency and model accuracy. After training, the models were evaluated on the test dataset to determine their accuracy. This evaluation provides insights into how well the models generalize to new, unseen data. The incorporation of dropout layers in Model 2 aimed to improve generalization by reducing overfitting. The performance metrics, including accuracy and loss, were tracked throughout the training process.

Visualization:

To monitor and visualize the learning process, the accuracy and loss metrics for both the training and validation sets were plotted over the epochs. These plots help in understanding the models' performance, convergence behavior, and the impact of dropout layers on mitigating overfitting.

Testing Audio Clips with Multi-Class Classification Model

The steps taken to preprocess and predict the species of birds from three audio clips using a trained multi-class Convolutional Neural Network (CNN) model. The procedure includes generating spectrograms from audio clips, preprocessing these spectrograms to match the model's input requirements, and utilizing the trained model to make predictions.

Audio Loading and Spectrogram Generation:

Libraries and Tools: The librosa library is used for audio processing, which provides functions for loading audio files and generating spectrograms.

Loading Audio Files: Three audio clips are loaded from specified file paths using the function, which reads the audio data into an array and provides the sampling rate.

Generating Mel-Spectrograms: For each audio clip, a Mel-spectrogram is generated. The Mel-spectrogram is a time-frequency representation that emphasizes frequencies relevant to human perception.

Power Conversion to dB: The power spectrogram is converted to a decibel (dB) scale. This step normalizes the spectrogram values, making them comparable to those used during model training.

Resizing Spectrograms: Each spectrogram is resized to the input dimensions (343, 256) expected by the trained model using a resizing function. This ensures compatibility with the model's input layer.

Ensuring Grayscale Channel: The spectrograms are expanded to have a single grayscale channel, aligning them with the input shape (343, 256, 1) used during model training.

Preprocessing for Prediction:

Loading Scaler: The `StandardScaler` object, which was fitted on the training data, is loaded. This scaler is used to standardize the spectrogram data, ensuring the same statistical properties as the training data.

Scaling Spectrograms: Each spectrogram is scaled using the loaded `StandardScaler`. The spectrograms are reshaped to a one-dimensional array, scaled, and then reshaped back to the original input dimensions. This step standardizes the spectrogram values, improving the model's prediction accuracy.

Reshaping for Model Input: After scaling, the spectrograms are reshaped back to the dimensions (343, 256, 1) to match the model's input layer requirements. This ensures the spectrograms are in the correct format for the CNN model to process.

Prediction Using Trained CNN Model:

Model Prediction: The preprocessed spectrograms are fed into the trained multi-class CNN model using the `predict` method. This method generates prediction probabilities for each class based on the input spectrogram. The output of the model, which consists of probabilities for each class, is processed to determine the predicted class. This is done by taking the argmax of the prediction probabilities, which indicates the class with the highest predicted probability.The predicted classes for each audio clip are stored in a list for further interpretation. Each predicted class corresponds to a specific bird species.

Result Interpretation:

Each predicted class is mapped to the corresponding bird species name using a predefined list of species names.The predicted species for each of the three test audio clips are printed out. This provides a clear output indicating which bird species the model has identified for each audio clip.

The described methodology ensures that audio clips are appropriately processed and analyzed using the trained CNN model to predict bird species accurately. By following these steps, the procedure can be easily reproduced, ensuring reliable and consistent results across different test samples. This methodology effectively combines advanced audio processing with machine learning techniques to classify bird species based on their vocalizations, showcasing the practical application of CNN models.

Computational Results:

The Xeno-Canto[1] bird sound archive provided data from 12 bird species, which were transformed into spectrogram images.

Binary Classification:

For binary classification, spectrograms of bird calls from Northern Flicker and Western Meadowlark were used. The computational results showcase the performance of different CNN architectures based on the presence or absence of dropout layers and the choice of activation functions. Each model's test accuracy and train accuracy are summarized below:

Activation Function	Presence of Dropout Layer	Test accuracy	Train accuracy
ReLU	No	94%	100%
ReLU	Yes	80%	95%
Tanh	No	47%	68%
Tanh	Yes	58%	61%
Elu	No	57.8%	84%
Elu	Yes	52.63%	85%

The table provides comparison of the models' performance metrics, highlighting the impact of dropout layers and activation functions on test and train accuracies.

From the above comparisons best performing model is ReLU without dropout layers. The graphs are plotted for that model for better understanding. Two types of plots, Fig 2. Binary Classification – Accuracy vs Epoch and Fig 3. Binary Classification- Loss vs Epoch, are plotted.

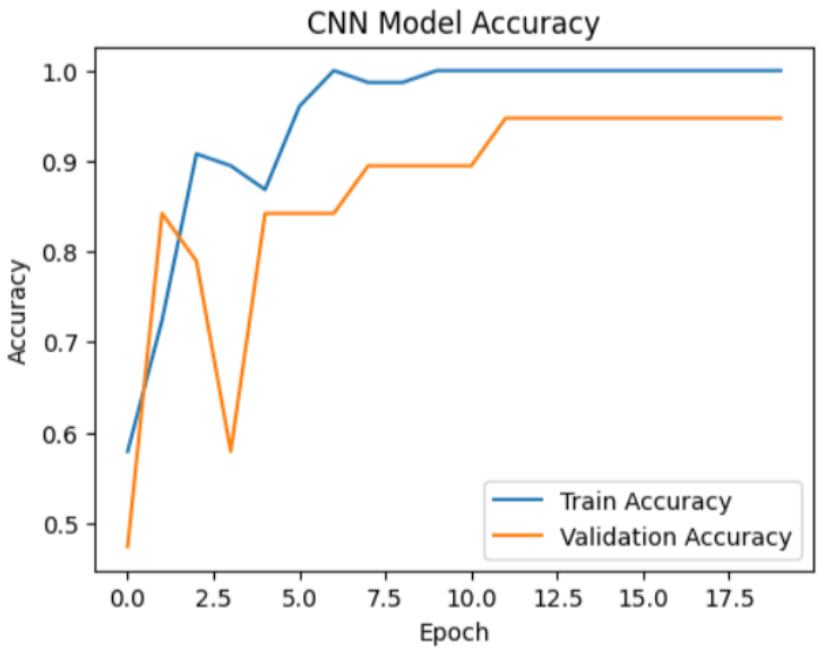


Fig 2. Binary Classification – Accuracy vs Epoch

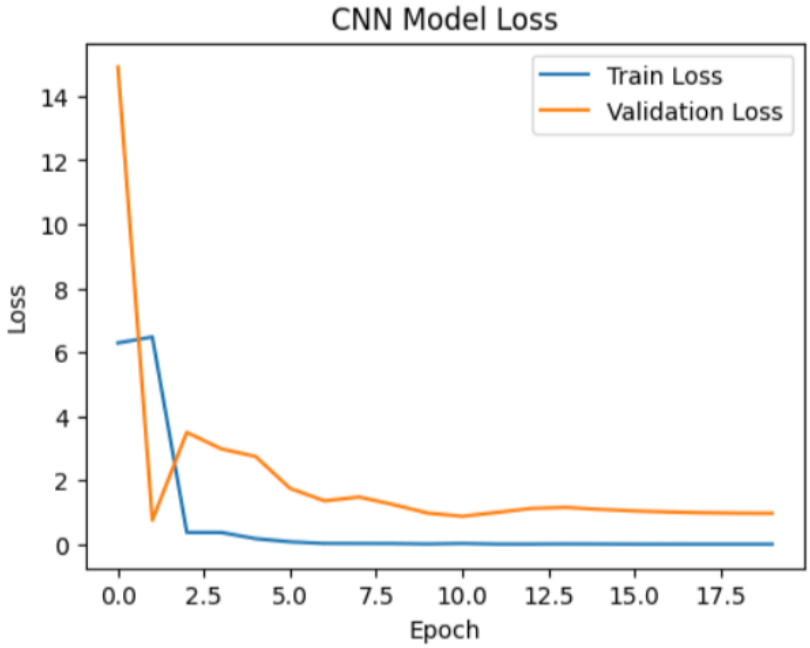


Fig 3. Binary Classification- Loss vs Epoch

Multi-class Classification:

The computational results illustrate the performance of two multi-class classification models, showcasing the test and train accuracies.

Model	Test Accuracy	Train Accuracy
Model 1	71%	94%
Model 2	68%	95%

Since Model 1 has higher a The graphs are plotted for that model for better understanding. Two types of plots, Fig 4. Multi-class Classification – Accuracy vs Epoch and Fig 5. Multi-class Classification- Loss vs Epoch, are plotted.

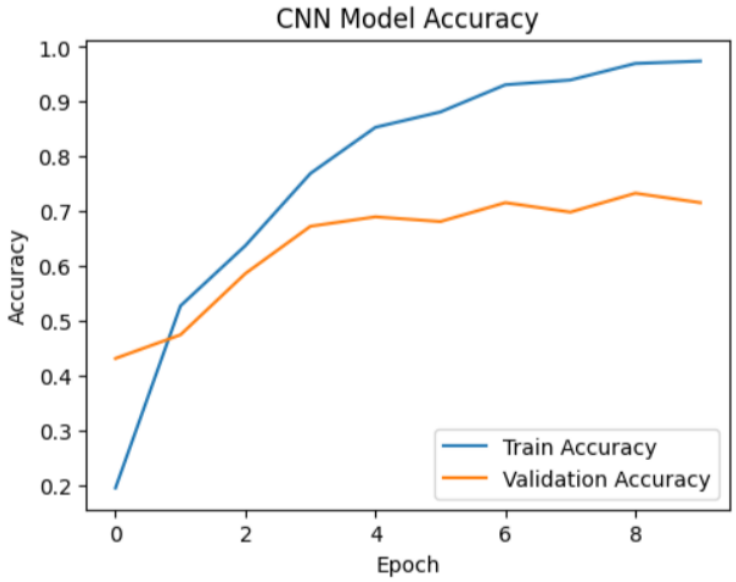


Fig 4. Multi-class Classification – Accuracy vs Epoch

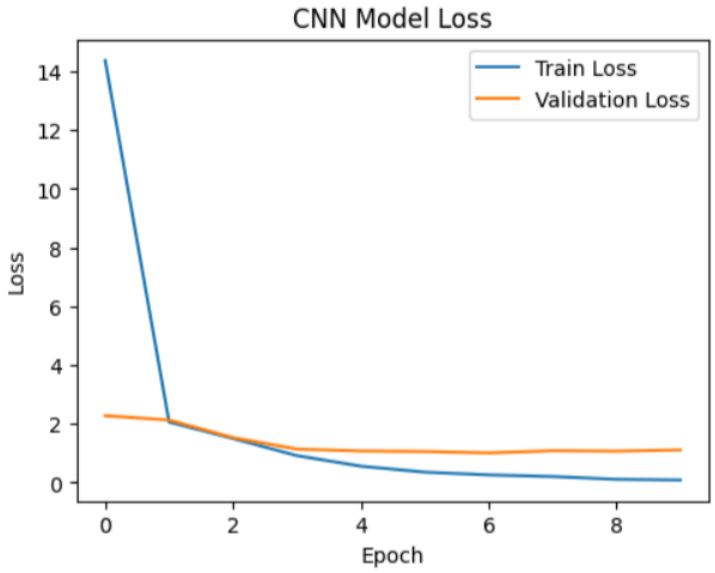


Fig 5. Multi-class Classification- Loss vs Epoch

For the External Test data:

When the test clips are processed through multi class model

Test Clip Number	Prediction
Test clip 1	Western Meadowalk
Test clip 2	Black-capped Chickadee
Test clip 3	Black-capped Chickadee

Discussion:

The computational analysis of binary classification models for bird species identification reveals important trends in model performance. Notably, models employing ReLU activation function without dropout layers exhibit superior test accuracy, reaching up to 94%. However, this high test accuracy may be indicative of overfitting, as evidenced by perfect train accuracy. Visually we can also observe in the graphs in Fig 2 and Fig 3. Models incorporating dropout layers demonstrate better generalization, with smaller disparities between test and train accuracies. While ReLU consistently outperforms Tanh and ELU activations, further investigation into regularization techniques is warranted to mitigate overfitting while maintaining high classification accuracy.In conclusion, the computational results highlight the significance of architectural choices and regularization methods in binary classification tasks. While ReLU activation without dropout

layers yields impressive test accuracy, the potential for overfitting requires careful consideration of model complexity and generalization.

From the computational results shows the performance of the two multi-class classification models. Model 1 achieved a test accuracy of 71%, indicating its ability to generalize reasonably well to unseen data, while maintaining a high training accuracy of 95%, suggesting effective learning from the training set. On the other hand, Model 2 exhibited slightly lower test accuracy at 68%, with a higher training accuracy of 98%. While Model 2 displayed a higher training accuracy, the slightly lower test accuracy compared to Model 1 suggests potential overfitting, where the model may have learned to classify the training data too precisely, resulting in reduced performance and visually plots are plotted.

The external test data is predicted for test clip 1 be Western Meadowalk and next 2 test clips be Black-capped Chickadee. The test clip contains 2 types of birds and few noises can be heard in the back in the most of the clips. While running the models it took different time for different models based on the complexity . For binary models it took about 7 minutes where as for multi class models it took 15minutes. While using the bird sound clips few bird sounds where very easy to train . For my binary model tried out various combinations of birds in that Red winged Blackbird and Barn Swallow were challenging. When considering the various approaches available for bird species classification based on vocalizations, it becomes evident that Convolutional Neural Networks (CNNs) are a well-suited choice. Unlike other methods like Support Vector Machines or Random Forests, CNNs offer a unique advantage due to their ability to learn hierarchical features directly from spectrogram data. This makes highly efficient in capturing both spatial and temporal patterns, which are essential for analyzing audio signals. By leveraging multiple convolutional and pooling layers, CNNs can automatically extract low-level features such as edges and textures, as well as higher-level features relevant to bird vocalizations. This eliminates the need for manual feature engineering, thereby streamlining the classification process. Moreover, the inherent flexibility of neural networks enables them to capture complex, nonlinear relationships within the data, which ultimately leads to more accurate classification results compared to traditional machine learning algorithms.

Conclusion:

This study explored the use of convolutional neural networks (CNNs) for classifying bird species from audio spectrograms, tackling both binary and multiclass classification tasks. Initially, the binary classification model efficiently distinguished between the vocalizations of Northern Flicker and Western Meadowlark, achieving a test accuracy of 94% without dropout layers, thus highlighting the model's capacity for precise differentiation. The multiclass model, tasked with categorizing 12 bird species native to the Seattle region, achieved a test accuracy of 71%, revealing both the potential and the challenges inherent in handling a broader classification scope.

The results underscore the efficacy of CNNs in extracting meaningful features from complex spectrogram data, while also pointing to the critical need for strategies to mitigate overfitting, as evidenced by the performance disparity between training and test accuracies. Future research should focus on enhancing data preprocessing and collection to ensure balanced datasets, exploring alternative neural network architectures, and implementing rigorous regularization and hyperparameter tuning. These steps will not only refine the classification models but also advance the application of deep learning, contributing significantly to biodiversity monitoring and conservation efforts.

References:

[1]Xento-canto birds sounds data https://xeno-canto.org/

[2]Scikit-learn for Neural Networks – Deep Learning – Supervised
https://scikitlearn.org/stable/modules/neural_networks_supervised.html

Appendix:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from google.colab import drive

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.optimizers import Adam

from skimage.transform import resize

import h5py

import tensorflow as tf

from tensorflow.keras import layers
```

```
drive.mount('/content/drive')
file_path = '/content/drive/My Drive/MS/data 5322/spectrograms.h5'
with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    species_data = {species: np.array(f[species]) for species in species_list}
    for species, data in species_data.items():
        print(f'{species}: {data.shape}')
def plot_spectrogram(spectrogram, title=None):
    plt.figure(figsize=(10, 4))
    plt.imshow(spectrogram.T, aspect='auto', origin='lower', cmap='viridis')
    plt.colorbar(format='%+2.0f dB')
    plt.title(title if title else 'Spectrogram')
    plt.xlabel('Time')
    plt.ylabel('Frequency')
    plt.show()
for species in species_list:
    print(f"Displaying spectrograms for species: {species}")
    for i in range(2):  # Display first 3 spectrograms for each species
        if i < species_data[species].shape[0]:
            plot_spectrogram(species_data[species][i], title=f'{species} - Sample {i+1}')
with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    norfli_data = f['norfli'][:]
```

```
    wesmea_data = f['wesmea'][:]

norfli_labels = np.zeros(norfli_data.shape[2])
wesmea_labels = np.ones(wesmea_data.shape[2])
X = np.concatenate((norfli_data.transpose(2, 1, 0), wesmea_data.transpose(2, 1, 0)), axis=0)
y = np.concatenate((norfli_labels, wesmea_labels), axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
nsamples, nx, ny = X_train.shape
X_train_reshaped = X_train.reshape((nsamples, nx*ny))

scaler = StandardScaler().fit(X_train_reshaped)
X_train_scaled = scaler.transform(X_train_reshaped)
X_test_reshaped = X_test.reshape((X_test.shape[0], nx*ny))
X_test_scaled = scaler.transform(X_test_reshaped)

X_train_scaled_cnn = X_train_scaled.reshape((nsamples, nx, ny, 1))
X_test_scaled_cnn = X_test_scaled.reshape((X_test.shape[0], nx, ny, 1))

log_reg = LogisticRegression(random_state=123)
log_reg.fit(X_train_scaled, y_train)
y_pred_log_reg = log_reg.predict(X_test_scaled)
log_reg_accuracy = accuracy_score(y_test, y_pred_log_reg)
print(f"Linear Regression test accuracy: {log_reg_accuracy:.4f}")

np.random.seed(123)
tf.random.set_seed(123)
```

```
bina=tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
bina.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


history = bina.fit(X_train_scaled_cnn, y_train, validation_data=(X_test_scaled_cnn, y_test), epochs=10)


test_loss, test_accuracy = bina.evaluate(X_test_scaled_cnn, y_test)
print(f"CNN test accuracy: {test_accuracy:.4f}")


plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')
```

```
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('CNN Model Loss')

plt.show()

with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    norfli_data = f['norfli'][:]
    wesmea_data = f['wesmea'][:]

norfli_labels = np.zeros(norfli_data.shape[2])
wesmea_labels = np.ones(wesmea_data.shape[2])
X = np.concatenate((norfli_data.transpose(2, 1, 0), wesmea_data.transpose(2, 1, 0)), axis=0)
y = np.concatenate((norfli_labels, wesmea_labels), axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
nsamples, nx, ny = X_train.shape
X_train_reshaped = X_train.reshape((nsamples, nx*ny))
```

```
scaler = StandardScaler().fit(X_train_reshaped)
X_train_scaled = scaler.transform(X_train_reshaped)

nsamples_test, nx_test, ny_test = X_test.shape
X_test_reshaped = X_test.reshape((nsamples_test, nx_test*ny_test))
X_test_scaled = scaler.transform(X_test_reshaped)

X_train_scaled_cnn = X_train_scaled.reshape((nsamples, nx, ny, 1))
X_test_scaled_cnn = X_test_scaled.reshape((nsamples_test, nx_test, ny_test, 1))

log_reg = LogisticRegression(random_state=123)
log_reg.fit(X_train_scaled, y_train)
y_pred_log_reg = log_reg.predict(X_test_scaled)
log_reg_accuracy = accuracy_score(y_test, y_pred_log_reg)
print(f"Linear Regression test accuracy: {log_reg_accuracy:.4f}")

np.random.seed(123)
tf.random.set_seed(123)

bina = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='tanh'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='tanh'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
```

```
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='tanh'),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

bina.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history = bina.fit(X_train_scaled_cnn, y_train, validation_data=(X_test_scaled_cnn, y_test), epochs=10)

test_loss, test_accuracy = bina.evaluate(X_test_scaled_cnn, y_test)
print(f"CNN test accuracy: {test_accuracy:.4f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
```

```
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('CNN Model Loss')

plt.show()
with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    norfli_data = f['norfli'][:]
    wesmea_data = f['wesmea'][:]

norfli_labels = np.zeros(norfli_data.shape[2])
wesmea_labels = np.ones(wesmea_data.shape[2])
X = np.concatenate((norfli_data.transpose(2, 1, 0), wesmea_data.transpose(2, 1, 0)), axis=0)
y = np.concatenate((norfli_labels, wesmea_labels), axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
nsamples, nx, ny = X_train.shape
X_train_reshaped = X_train.reshape((nsamples, nx*ny))

scaler = StandardScaler().fit(X_train_reshaped)
X_train_scaled = scaler.transform(X_train_reshaped)
X_test_reshaped = X_test.reshape((X_test.shape[0], nx*ny))
X_test_scaled = scaler.transform(X_test_reshaped)

X_train_scaled_cnn = X_train_scaled.reshape((nsamples, nx, ny, 1))
```

```
X_test_scaled_cnn = X_test_scaled.reshape((X_test.shape[0], nx, ny, 1))

log_reg = LogisticRegression(random_state=123)
log_reg.fit(X_train_scaled, y_train)
y_pred_log_reg = log_reg.predict(X_test_scaled)
log_reg_accuracy = accuracy_score(y_test, y_pred_log_reg)
print(f"Linear Regression test accuracy: {log_reg_accuracy:.4f}")

np.random.seed(123)
tf.random.set_seed(123)

bina = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='elu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='elu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='elu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
bina.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history = bina.fit(X_train_scaled_cnn, y_train, validation_data=(X_test_scaled_cnn, y_test), epochs=10)

test_loss, test_accuracy = bina.evaluate(X_test_scaled_cnn, y_test)
```

```
print(f"CNN test accuracy: {test_accuracy:.4f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('CNN Model Loss')

plt.show()

with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    norfli_data = f['norfli'][:]
    wesmea_data = f['wesmea'][:]
```

```
norfli_labels = np.zeros(norfli_data.shape[2])
wesmea_labels = np.ones(wesmea_data.shape[2])
X = np.concatenate((norfli_data.transpose(2, 1, 0), wesmea_data.transpose(2, 1, 0)), axis=0)
y = np.concatenate((norfli_labels, wesmea_labels), axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
nsamples, nx, ny = X_train.shape
X_train_reshaped = X_train.reshape((nsamples, nx*ny))

scaler = StandardScaler().fit(X_train_reshaped)
X_train_scaled = scaler.transform(X_train_reshaped)

nsamples_test, nx_test, ny_test = X_test.shape
X_test_reshaped = X_test.reshape((nsamples_test, nx_test*ny_test))
X_test_scaled = scaler.transform(X_test_reshaped)

X_train_scaled_cnn = X_train_scaled.reshape((nsamples, nx, ny, 1))
X_test_scaled_cnn = X_test_scaled.reshape((nsamples_test, nx_test, ny_test, 1))

log_reg = LogisticRegression(random_state=123)
log_reg.fit(X_train_scaled, y_train)
y_pred_log_reg = log_reg.predict(X_test_scaled)
log_reg_accuracy = accuracy_score(y_test, y_pred_log_reg)
print(f"Linear Regression test accuracy: {log_reg_accuracy:.4f}")

np.random.seed(123)
```

```
tf.random.set_seed(123)

bina = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='elu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='elu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='elu'),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

bina.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history = bina.fit(X_train_scaled_cnn, y_train, validation_data=(X_test_scaled_cnn, y_test), epochs=10)

test_loss, test_accuracy = bina.evaluate(X_test_scaled_cnn, y_test)
print(f"CNN test accuracy: {test_accuracy:.4f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('CNN Model Loss')

plt.show()
with h5py.File(file_path, 'r') as f:
    species_list = list(f.keys())
    print(f"Species list: {species_list}")

    data = []
    labels = []
    for i, species in enumerate(species_list):
        species_data = f[species][:]
        n_samples = species_data.shape[2]
        species_data = species_data.transpose(2, 1, 0)
        data.append(species_data)
        labels.extend([i] * n_samples)
```

```
X = np.concatenate(data, axis=0)
y = np.array(labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training data shape:", X_train.shape)
print("Test data shape:", X_test.shape)

nsamples, nx, ny = X_train.shape
X_train_reshaped = X_train.reshape((nsamples, nx*ny))

scaler = StandardScaler().fit(X_train_reshaped)
X_train_scaled = scaler.transform(X_train_reshaped)

nsamples_test, nx_test, ny_test = X_test.shape
X_test_reshaped = X_test.reshape((nsamples_test, nx_test*ny_test))
X_test_scaled = scaler.transform(X_test_reshaped)

X_train_scaled_cnn = X_train_scaled.reshape((nsamples, nx, ny, 1))
X_test_scaled_cnn = X_test_scaled.reshape((nsamples_test, nx_test, ny_test, 1))

num_classes = len(species_list)
y_train_one_hot = tf.keras.utils.to_categorical(y_train, num_classes)
y_test_one_hot = tf.keras.utils.to_categorical(y_test, num_classes)

np.random.seed(123)
```

```
tf.random.set_seed(123)

multi_class_model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

multi_class_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = multi_class_model.fit(X_train_scaled_cnn, y_train_one_hot, validation_data=(X_test_scaled_cnn, y_test_one_hot), epochs=10)

test_loss, test_accuracy = multi_class_model.evaluate(X_test_scaled_cnn, y_test_one_hot)
print(f"CNN test accuracy: {test_accuracy:.4f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
```

```
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('CNN Model Loss')

plt.show()

np.random.seed(123)
tf.random.set_seed(123)

multi_class_model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(343, 256, 1)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
```

```
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])

multi_class_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = multi_class_model.fit(X_train_scaled_cnn, y_train_one_hot, validation_data=(X_test_scaled_cnn, y_test_one_hot), epochs=10)

test_loss, test_accuracy = multi_class_model.evaluate(X_test_scaled_cnn, y_test_one_hot)
print(f"CNN test accuracy: {test_accuracy:.4f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('CNN Model Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
```

```
plt.title('CNN Model Loss')

plt.show()

import librosa

def generate_spectrograms(audio_paths):
    spectrograms = []
    for audio_path in audio_paths:
        y, sr = librosa.load(audio_path)
        S = librosa.feature.melspectrogram(y=y, sr=sr)
        S_db = librosa.power_to_db(S, ref=np.max)
        S_db_resized = resize(S_db, (343, 256), mode='constant')
        S_db_resized_gray = np.expand_dims(S_db_resized, axis=-1)
        spectrograms.append(S_db_resized_gray)
    return spectrograms

audio_paths = ["/content/drive/My Drive/MS/data 5322/test_birds/test_birds/test1.mp3",
               "/content/drive/My Drive/MS/data 5322/test_birds/test_birds/test2.mp3",
               "/content/drive/My Drive/MS/data 5322/test_birds/test_birds/test3.mp3"]

spectrograms = generate_spectrograms(audio_paths)

scaler = StandardScaler().fit(X_train.reshape((X_train.shape[0], -1)))

def preprocess_spectrogram(spectrogram):
    if spectrogram.shape != (343, 256, 1):
```

```
        spectrogram = resize(spectrogram, (343, 256, 1), mode='constant')
    spectrogram_scaled = scaler.transform(spectrogram.reshape(1, -1))
    return spectrogram_scaled.reshape((343, 256, 1))

predictions = []
for spectrogram in spectrograms:
    preprocessed_spectrogram = preprocess_spectrogram(spectrogram)
    prediction = multi_class_model.predict(np.expand_dims(preprocessed_spectrogram, axis=0))
    predicted_class = np.argmax(prediction, axis=1)
    predictions.append(predicted_class[0])

for i, prediction in enumerate(predictions):
    print(f"Test clip {i+1} predicted species: {species_list[prediction]}")
```