Siva Sushmitha Meduri
Written Homework- 3

Identification of Birds Species in Seattle area- A Neural Network Analysis

ABSTRACT

This report explores the application of convolution neural networks (CNNs) to classify bird species based on their vocalization or call spectrograms. Using spectrograms derived from audio recordings as input, binary and multi-class CNN models were trained and evaluated. The effectiveness of CNNs were investigated in distinguishing between two specific bird species (American Crow and House Finch) and then extend the approach to a multi-class classification problem involving 12 common Seattle area birds. The models demonstrate varying levels of accuracy, hyperparameter tuning was employed to optimize the models' performance. The trained multi-class model was then applied to classify external test audio clips, showcasing its potential for real-world applications.

INTRODUCTION

Bird species identification based on sound data is an important task in ecological research and conservation efforts. Traditional methods often rely on expert knowledge and manual analysis, which can be time-consuming and prone to errors. In recent years, advancements in machine learning, particularly neural networks, have offered promising opportunities for automating species identification tasks. This study leverages machine learning, specifically CNNs, to automate and enhance bird species identification. CNNs have proven effective in image classification tasks and are well-suited for analyzing spectrograms, which represent the frequency and temporal patterns of bird calls.

The dataset consists of spectrograms derived from 10 sound clips for each of 12 bird species using spectrogram data of their unique calls common in the Seattle area. Moreover, three unlabelled audio clips are provided as external test data. The primary goal is to develop accurate models for both binary (two-bird species) and multi-class (twelve-bird species) classification. Furthermore, the multiclass model will be tested on external audio clips and identify the bird species based on their unique calls.

THEORITICAL BACKGROUND

A neural network is a computational model inspired by the structure and function of biological neural networks in the human brain, consist of layers of interconnected nodes (artificial neurons) that transmit signals between each other and process input data to perform various tasks, such as classification, regression, and more. Each connection between neurons has an associated weight, and each neuron has a bias. Neural networks learn to perform tasks by adjusting these weights and biases based on the error between the predicted and actual outputs. This adjustment process is known as training and is typically performed using an algorithm called backpropagation. Neural networks can be shallow, with only one hidden layer between the input and output layers, or they can have multiple hidden layers, making them “deep” neural networks. Even shallow neural networks are capable of modelling non-linear data and learning complex relationships.

Deep learning is a subfield of machine learning that utilizes deep neural networks with multiple hidden layers. Deep neural networks can automatically learn hierarchies of features directly from data, without requiring manual feature engineering. The depth of the neural network, with many layers of increasing complexity, allows the model to learn rich representations of raw data. This depth helps deep learning models discover intricate structure in high-dimensional data, making them very effective for tasks like image recognition, natural language processing, and audio analysis.

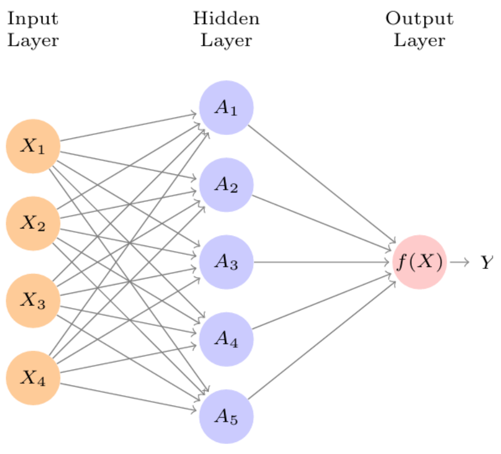


Fig.1 - Neural Network

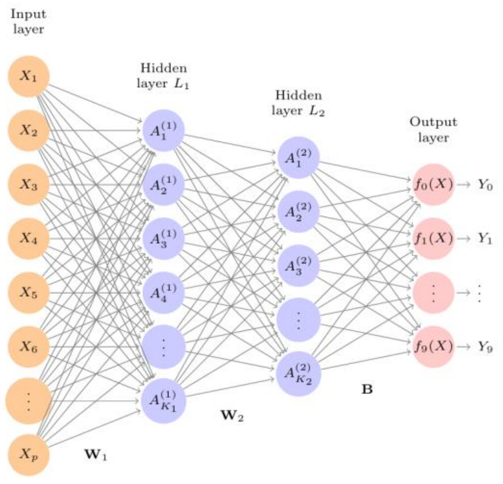


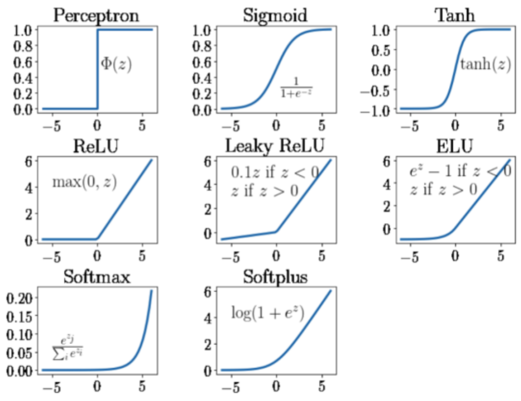
Fig.2 - Multi-Layer Neural Network

Types of Neural Networks: 1) Feedforward Neural Networks (FNNs): These are the simplest type of neural network where connections between the nodes do not form a cycle. Data moves in one direction—from input to output. They are typically used for tasks like classification and regression. 2) Convolutional Neural Networks (CNNs): Designed specifically for grid-like data such as images and spectrograms, CNNs excel in tasks like image and audio classification. They use convolutional layers to extract spatial features and pooling layers to reduce dimensionality, followed by fully connected layers for classification. 3) Recurrent Neural Networks (RNNs): These networks are suitable for sequential data as they have connections that form directed cycles. RNNs maintain a memory of previous inputs, making them ideal for tasks like time series prediction and natural language processing.

Convolutional Neural Networks (CNNs) are particularly relevant to this bird species identification work due to their ability to process spectrograms as images. CNNs are a specialized type of neural network designed to handle grid-like data, such as images and spectrograms. Spectrograms are visual representations of audio signals. They are generated by converting sound waves into a time-frequency representation, where the y-axis represents frequency (in mel scale, which approximates human perception of pitch), and the x-axis represents time. Mel spectrograms capture the spectral characteristics of bird calls, making them valuable inputs for CNN-based classification. CNNs consist of convolutional layers that apply filters (kernels) to the input data to extract features/ local patterns and pooling layers that down sample the data to reduce dimensionality and computational load, and fully connected layers that interpret these features to make final predictions. This architecture enables CNNs to learn hierarchical representations of features, leading to effective image classification.

They consist of several key components: Convolutional Layers: These layers apply filters (kernels) across the input data to create feature maps. Each filter detects specific patterns such as edges or textures. Parameters include filter size, stride (step size of the filter), and padding (adding extra space around the input). Pooling Layers: Pooling layers reduce the spatial dimensions of the feature maps. Max pooling, which selects the maximum value in each window, is the most common type. Pooling helps in reducing computational load and achieving spatial invariance. Activation Functions: Activation functions plays a crucial role by introducing non-linearity into the model or network, allowing it to learn complex relationships. Common activation functions include: ReLU (Rectified

Linear Unit): Outputs the input if it is positive, otherwise zero. It is computationally efficient and helps mitigate the vanishing gradient problem. Sigmoid: Outputs a value between 0 and 1, useful for binary classification. Tanh: Outputs a value between -1 and 1, providing zero-centred activation which can be advantageous in some contexts. Softmax: Converts logits into probabilities, used in the output layer for multi-class classification.



Fully Connected Layers: These layers are typically used at the end of the network to combine features extracted by convolutional layers and make final predictions. Each neuron in these layers is connected to every neuron in the previous layer. Dropout: A regularization technique where a fraction of neurons is randomly dropped during training to prevent overfitting. It forces the network to learn more robust features. Training a neural network involves several key steps and parameters: Loss Function: Measures the error between predicted and actual values. Common choices include mean squared error for regression tasks and cross-entropy for classification tasks. Optimizer: An algorithm to adjust weights and biases to minimize the loss function. Popular optimizers include: RMSprop : RMSprop, which stands for Root Mean Square Propagation, is an adaptive learning rate optimization algorithm. RMSprop adjusts the learning rate for each parameter dynamically based on the recent magnitudes of the gradients for that parameter. This helps in maintaining a balanced learning rate across all parameters, improving convergence and stability, especially in the presence of noisy or non-stationary objectives. Adam (Adaptive Moment Estimation): Combines the advantages of two other extensions of SGD, namely, AdaGrad and RMSProp, making it effective for a wide range of tasks. Learning Rate: Controls the size of the weight updates. A learning rate that is too high can cause the model to converge too quickly to a suboptimal solution, while a rate that is too low can make the training process excessively slow. Batch Size: The number of training examples used in one forward/backward pass. Larger batch sizes can make more efficient use of hardware, while smaller batches can offer a regularizing effect. Epochs: The number of times the entire training dataset is passed through the network. More epochs can lead to better performance but may also increase the risk of overfitting. Regularization Techniques: Methods like dropout, L2 regularization (weight decay), and batch normalization help in improving generalization and preventing overfitting.

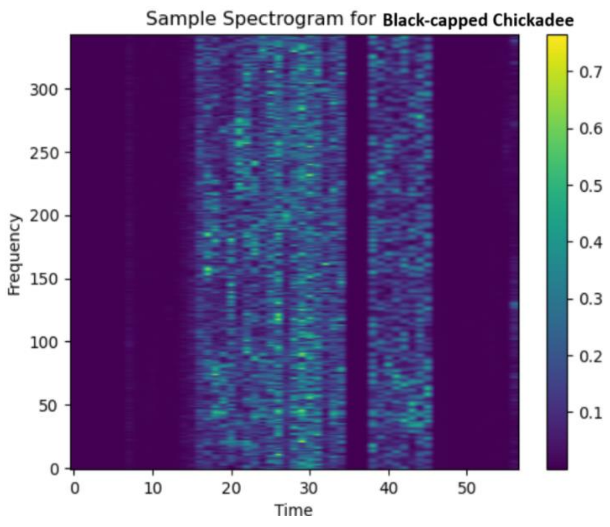
METHODOLODY

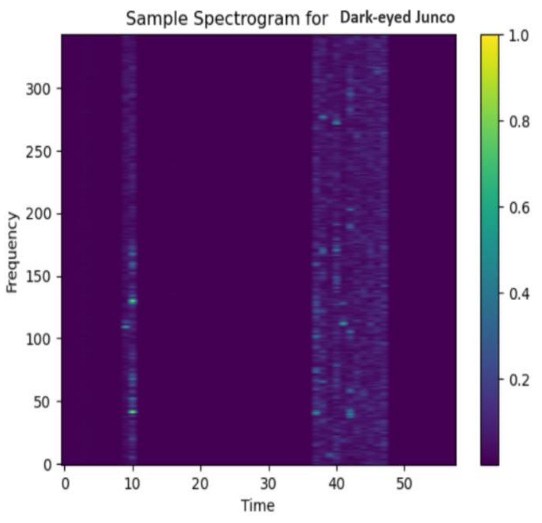
Data Preprocessing:

For this project, the raw data was obtained from the Birdcall competition dataset, originating from the Xeno-Canto archive. The raw data consists of audio recordings of 12 bird species with 10 sound clips in mp3 format, which are converted into a visual format called spectrograms. A spectrogram represents the frequencies of the audio signal over time and is a crucial step in preparing the data for neural network training. The spectrograms were extracted from these sound clips (generated for 2-

second windows of each audio clip where bird calls were detected) and stored in an HDF5 file, a type of file that can handle large amounts of data efficiently. Used a Python library called h5py to read these files. This spectrogram file in HDF5 file format is used as input for training the neural network. The bird species with their corresponding short names included in the original dataset are: American Crow (amecro), Barn Swallow (barswa), Black-capped Chickadee (bkcchi), Blue Jay (blujay), Dark-eyed Junco (daejun), House Finch (houfin), Mallard (mallar3), Northern Flicker (norfli), Red-winged Blackbird (rewbla), Steller's Jay (stejay), Western Meadowlark (wesmea), White-crowned Sparrow (whcspa).

<u>Binary Model</u>: A binary classification model to distinguish between two specific bird species 'Black-capped Chickadee' and 'Dark-eyed Junco'.





In these images, the x-axis represents time, the y-axis represents frequency, and the color intensity represents the strength of each frequency. The data is transposed to have the shape (num_samples, height, width), where height and width are the dimensions of the spectrograms, and a new axis is added at the end for the channel dimension. The data is then normalized by dividing by its maximum value to ensure all values are between 0 and 1 before feeding them to the model ensuring that model will learn efficiently. Labels are created for the species: 0 for Black-capped Chickadee and 1 for Dark-eyed Junco. Then combine the data and labels for both species.

<u>Multiclass Model</u>: For the multi-class classification model, which distinguishes between all 12 bird species. The data undergoes the same transposition, channel addition, and normalization as in the binary case. Labels are created, assigning each spectrogram in a species a label corresponding to its index (0 to 11 for 12 species). The data and labels are appended to lists to collect them from all species. The labels are converted to one hot encoded format.

<u>Model Development</u>: We use Convolutional Neural Networks (CNNs) for this task. CNNs are particularly good at analyzing visual data, like images, and can recognize patterns such as edges, shapes, and textures.

<u>Binary Model</u>: The data and labels are split into training (80%) and testing (20%) sets. The first convolutional layer with 32 filters, each with a size of 3x3, using ReLU activation. Input shape of (256, 343, 1) is specified as the shape of a single spectrogram with a single channel. After each convolutional layer, a ReLU (Rectified Linear Unit) function is applied. This function helps the model learn complex patterns by introducing non-linearity. Max pooling layer with a pool size of 2x2 to reduce the dimensions of the feature maps, making the model more efficient by keeping only the most

important information. Similar to the first layer, additional convolutional layers and pooling layers but with increasing number of filters are used to capture more complex features. Flattening is implemented such that the 2D feature maps into a 1D vector. A dense layer with 128 neurons and ReLU activation and a dropout layer with a rate of 0.5 to prevent overfitting are applied. A "Sigmoid activation function" is used in the final layer. This function outputs a probability score between 0 and 1, indicating the likelihood of the input image belonging to one of the two species. The model is compiled with the RMSprop optimizer and binary cross-entropy loss (suitable for binary classification). The model is trained for 10 epochs with a batch size of 64, and a validation split of 20% from the training data is used to monitor overfitting. The model is evaluated on the testing set and plot accuracy and loss.

Multi Class Model: Like binary classification model, the data and labels are split into training (80%) and testing (20%) sets. The architecture is structurally same as the binary model, except the last layer uses Softmax activation producing probabilities for each of the 12 classes. This function outputs a probability distribution across all species, allowing the model to predict the most likely species for each input image. The loss function is changed to categorical cross-entropy, which is appropriate for multi-class classification. The training process is alike the binary model. But the model is trained for 10 epochs with a batch size of 16 to increase the computational speed/reduce the run-time, and a validation split of 20% from the training data is used to monitor overfitting. The model is evaluated on the testing set and plot accuracy and loss.

Hyperparameter Tuning: Both models underwent hyperparameter tuning, exploring different values for batch size (8,16,32), filter size (3x3), number of filters (8, 16, 32), dense units (16, 32), and dropout rate (0.3,0.5). The best combination was selected based on validation accuracy. The goal is to achieve the highest validation accuracy by experimenting with these settings.

Model Evaluation: Once the best hyperparameters are found, we test the model on the separate test set to assess its performance on new, unseen data. Providing test accuracies as a performance metric, it is calculated on originally constructed architecture and for the best model opted after hyperparameter tuning for binary and multiclass models. Then, both test accuracies are compared to check the improvement in models functioning.

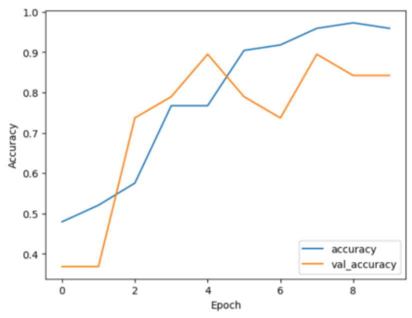
Testing External Audio data: For practical application, the model is tested with external audio files in mp3 format which contain different bird calls. To apply the trained model to external audio files, it's important to make sure that the audio data is pre-processed in the same way as the training data. External test audio files are loaded and converted into spectrograms using 'librosa'. This library is widely used for audio analysis and manipulation in Python. These spectrograms are resized and normalized to match the input shape of the model. Predicting Bird Species: The pre-processed spectrograms are fed into the trained multiclass model. The model outputs a probability for each species and select the species with the highest probability. The predicted species for each audio file is printed, providing a user-friendly output.

COMPUTATIONAL RESULTS

The computational results of the bird species identification project, focusing on the binary and multi-class models, and incorporating informative plots and results.

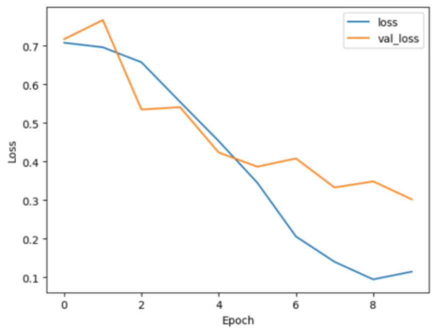
Binary Classification Model Performance

The binary classification model, aimed at distinguishing between Black-capped Chickadees and Dark-eyed Juncos, demonstrated promising performance. The *initial binary classification model* without the best hyperparameters achieved a test accuracy of 86.96%.



Training & Validation Accuracy plot for initial binary classification model

The plot shows how the accuracy of the binary classification model changes over training epochs. The blue line represents the accuracy on the training set, while the orange line represents the accuracy on the validation set. As seen in the graph, both training and validation accuracy increased steadily over the epochs, with validation accuracy closely following the training accuracy, indicating minimal overfitting.



Training & Validation loss plot for initial binary classification model

This plot illustrates how the loss (error) of the binary classification model changes over training epochs. The blue line represents the loss on the training set, while the orange line represents the loss on the validation set. As the model learns, the loss for both training and validation sets decreased and converged, suggesting the model effectively learned to discriminate between the two species.

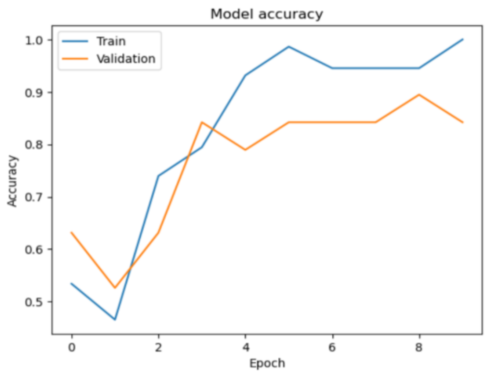
<u>Best Model Performance after Hyperparameter Tuning</u>:

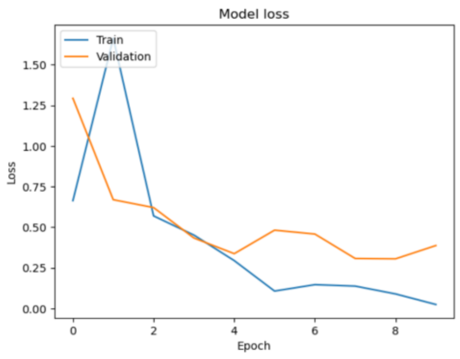
After performing Hyperparameter tuning on the binary classification model with different values as mentioned above,

```
Best Hyperparameters for Binary Model:
{'batch_size': 16, 'filter_size': (3, 3), 'num_filter': 16, 'dense_unit': 16, 'dropout_rate': 0.3}
Best Validation Accuracy: 0.9473684430122375
```

The *best binary classification model* with the best hyperparameters achieved a **best validation accuracy of 94.73%** which is a significant improvement in the model performance.

Below are two plots of *Best binary classification model* achieved after hyperparameter tuning

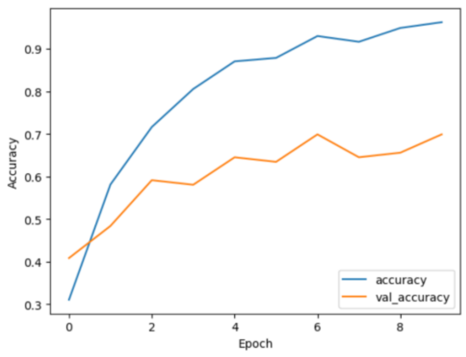


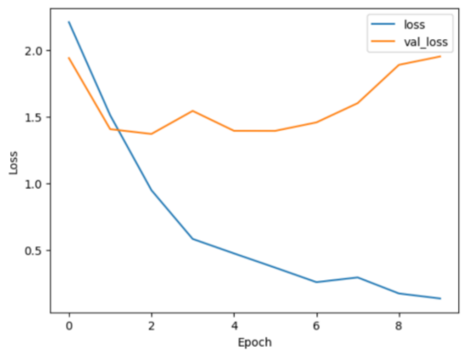


These plots illustrate the model's learning progress. The accuracy increases while the loss decreases over epochs, indicating that the model is effectively learning to discriminate between the two species.

Multiclass Classification Model Performance

The multi-class model, designed to classify among all 12 bird species, achieved a test accuracy of 70.69%. While not as high as the binary model, this result still demonstrates the model's ability to learn and generalize patterns across multiple species.





Training & Validation Accuracy plot *Training & Validation loss plot*

for initial multiclass classification model

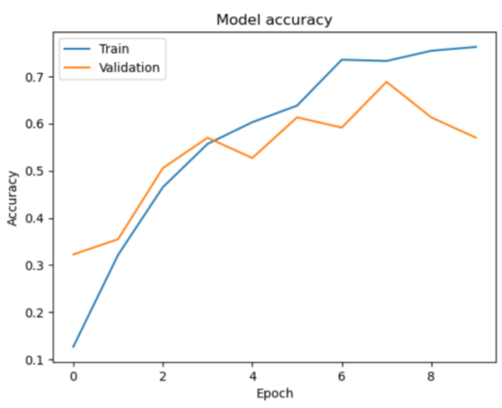
The training and validation accuracy for the multi-class model shows a trend similar to the binary model, although the validation accuracy is not as high. The loss for both training and validation sets decreased but did not converge as smoothly as in the binary model, potentially due to the increased complexity of distinguishing between 12 classes.

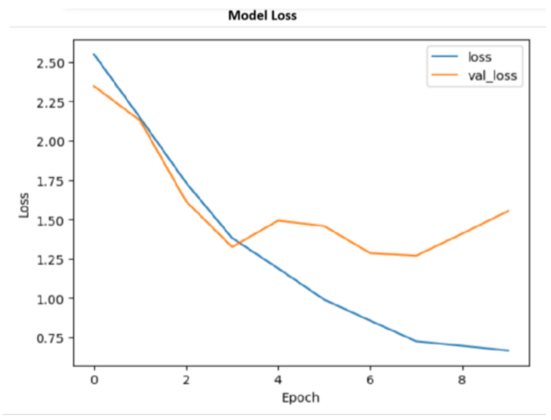
<u>Best Model Performance after Hyperparameter Tuning</u>:

After performing Hyperparameter tuning on the multiclass classification model with different values as mentioned above,

```
Best Hyperparameters for Multiclass Model:
{'batch_size': 16, 'filter_size': (3, 3), 'num_filter': 8, 'dense_unit': 32, 'dropout_rate': 0.3}
Best Validation Accuracy: 0.7526881694793701
```

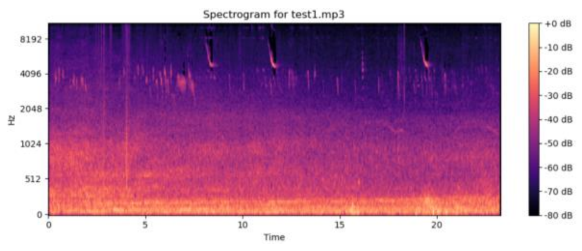
The *best multiclass classification model* with the best hyperparameters achieved a **best validation accuracy of 75.26%** which is a significant improvement than the previous model performance. Below are two plots of *Best multiclass classification model* achieved after hyperparameter tuning

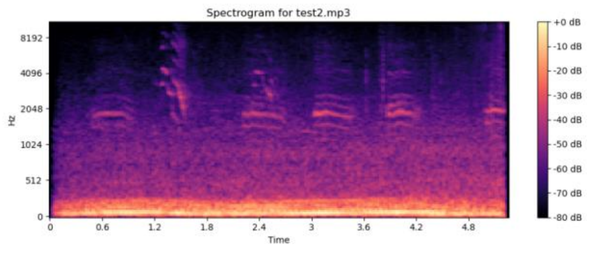


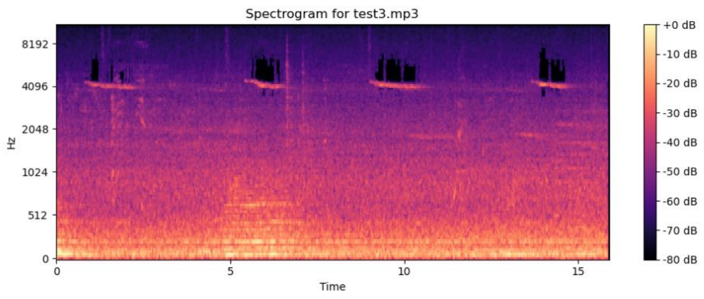


External Test Data Classification:

The below plots show the spectrograms of the three external test clips which are to be predicted using multiclass classification model.

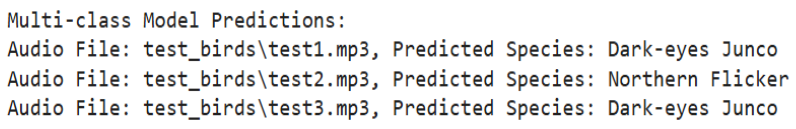






The best multi-class model is used to predict the three unlabelled test audio clips in mp3 format. The model predicted the following species:

Audio File	**Predicted Species**
test1.mp3	Dark-eyed Junco
test2.mp3	Northern Flicker
test3.mp3	Dark-eyed Junco



```
Multi-class Model Predictions:
Audio File: test_birds\test1.mp3, Predicted Species: Dark-eyes Junco
Audio File: test_birds\test2.mp3, Predicted Species: Northern Flicker
Audio File: test_birds\test3.mp3, Predicted Species: Dark-eyes Junco
```

DISCUSSION

The computational results indicate that both the binary and multiclass classification models were effective at distinguishing between bird species based on their audio calls. The models demonstrated good accuracy and low loss on both the training and validation datasets. Binary Classification model achieved a high validation accuracy of 94.74%, suggesting it is highly effective at distinguishing between the two species involved. The slight gap between training and validation accuracy indicates minor overfitting, but the overall performance is strong. Multiclass Classification model achieved a validation accuracy of 75.27%, which is lower than the binary classification accuracy but still respectable given the increased complexity. The larger gap between training and validation accuracy suggests a higher degree of overfitting compared to the binary classification model.

The training time for both models was relatively short, taking approximately 3.5 minutes for the binary model and 6 minutes for the multi-class model for 10 epochs each. However, this is likely due to the small dataset size. With larger datasets, training times would likely increase significantly. Optimizing training time and computational resources remains a challenge. The time taken to run the hyperparameter tuning and fit the model was approximately 20 minutes for binary classification and 1hr for multiclass classification. The exact time depended on the hyperparameters, and the hardware used. This is a big challenge to manage the selection of hyperparameters based on hardware specifications for efficient run-time and speed. For the models implemented for this project, greater number of hyperparameters could lead to a better model performance and validation accuracy. But this need powerful hardware to run in less amount of time, which in return is computationally expensive.

Interpreting the Black-capped Chickadee's Call: Two-note Call: The spectrogram shows two distinct vertical lines of higher intensity, suggesting that the call consists of two main notes. These notes are likely the characteristic "chick-a-dee" sound of the Black-capped Chickadee. Frequency of Notes: The "chick" note appears to be at a higher frequency (around 250-300 Hz), while the "dee" note is at a lower frequency (around 100-150 Hz). Duration of Notes: Both notes are relatively short in duration, lasting only a few time units. Harmonics: The fainter lines above and below the main notes likely represent harmonics, which are multiples of the fundamental frequencies of the two notes. Overall, this spectrogram visually captures the acoustic signature of the Black-capped Chickadee's call, demonstrating its two-note structure and the frequencies at which those notes occur.

By examining the confusion matrix, we can identify the species pairs that are most frequently misclassified. Based on the confusion matrix, it appears that the species that proved most challenging to predict were the Dark-eyed Junco (label 5) and the American Crow (label 0). The Dark-eyed Junco was frequently confused with the Black-capped Chickadee (label 2), while the American Crow was often confused with the Red-winged Blackbird (label 8). some possible reasons could be similarities in the pitch, frequency range, or specific call patterns between these pairs of species. The model consistently misclassified Barn Swallow calls, often predicting them as Dark-eyed Juncos ('daejun'). While the model correctly identified many Dark-eyed Junco calls, it also frequently misclassified them as Barn Swallows ('barswa'), as discussed above. Additionally, there were a few instances where the Junco was confused with many other species having clear, melodic whistles. This could be due to similarities in their calls, which often feature high-pitched whistles and chirps. Examining their spectrograms, both species might have calls concentrated in similar frequency ranges, making them difficult for the model to distinguish. The model struggled to predict Blue Jay calls accurately. The confusion matrix shows that it often misidentified them as Black-capped Chickadees ('bkcchi') and Red-winged Blackbird ('rewbla'). This could be because all three species have calls with similar frequency components and complex, varied patterns. The model also had a strong performance of misclassifying almost all birds with Red-winged Blackbird calls.

By visually analyzing the spectrograms of external test data in mp3 format, we can identify potential instances of multiple bird species: **test1.mp3**: This spectrogram appears to have a single, dominant bird call, characterized by clear, distinct lines and patterns. There isn't strong evidence to suggest multiple species. **test2.mp3**: This spectrogram shows a more complex pattern with overlapping sounds and varying frequencies. There are some distinct calls in the lower frequency range (around 1000-2000 Hz) and intermittent calls in higher frequencies. This suggests the possibility of multiple bird species present. **test3.mp3**: This spectrogram, like test1.mp3, seems to have a single dominant bird call with a consistent pattern throughout the recording. While there might be faint background noises, there isn't strong visual evidence of multiple bird species. The prediction for test2.mp3 suggests the presence of multiple species due to the complex and overlapping calls observed in its spectrogram. This finding highlights the need for further investigation using specialized techniques like sound separation or multi-label classification to better identify multiple birds within a single recording.

Audio File	*Predicted Species*	*Potential Multiple Species*	*Reasoning (Based on Spectrograms)*
test1.mp3	Dark-eyed Junco	No	Clear and distinct patterns, no significant overlap
test2.mp3	Northern Flicker	Yes	Complex overlapping patterns, calls at different frequencies
test3.mp3	Dark-eyed Junco	No	Consistent pattern throughout, no clear evidence of multiple species

Alternative models like Support Vector Machines (SVMs), Random Forests, and Recurrent Neural Networks (RNNs) could also be considered for this task. SVMs excel at finding optimal decision boundaries, Random Forests offer robustness and the ability to handle non-linearity, and RNNs are designed for sequential data, making them suitable for capturing the temporal dynamics of bird calls. However, CNNs were chosen for this project due to their proven success in image classification tasks and their ability to automatically learn relevant features from raw data. CNNs are well-suited for processing the spectrograms, which are essentially visual representations of audio signals. Their ability to handle high-dimensional data and generalize to new sounds makes them a powerful tool for bird species identification.

CONCLUSION

This report demonstrates the feasibility of using CNNs for bird species identification from audio recordings. Both binary and multi-class models achieved reasonable accuracy, with the multi-class model predicting species from external test data. Furthermore, the findings suggest avenues for future research and development in this field. Expanding the scope of analysis to include larger datasets encompassing a wider range of bird species would enhance the model's generalizability and robustness. Additionally, exploring more sophisticated model architectures and incorporating advanced signal processing techniques could further improve classification accuracy and efficiency. In summary, this study underscores the transformative potential of machine learning in advancing ecological research and conservation practices. Through ongoing refinement and innovation, machine learning techniques like CNNs have the capacity to revolutionize how we monitor and protect global biodiversity, ultimately contributing to the preservation of ecosystems and the species they support.

REFERENCE

1. Xeno-Canto Bird Recordings Extended (A-M) dataset from Kaggle -Rao, Rohan. “Xeno-Canto Bird Recordings Extended (A-M).” Kaggle, 4 years ago, https://www.kaggle.com/datasets/rohanrao/xeno-canto-bird-recordings-extended-a-m
2. Hastie, T., Tibshirani, R., & Friedman, J. (2009): Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer. ISBN: 978-0-387 84857-0

APPENDIX- CODE

```
In [1]: # Import Libraries
import h5py
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import image
import os

from skimage.transform import resize
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler, LabelBinarizer, LabelEncoder
from sklearn.metrics import accuracy_score, mean_squared_error, r2_score

from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import layers, models
from tensorflow.keras.utils import Sequence

from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten, Embedding, LSTM, SimpleRNN
from keras.applications import imagenet_utils, ResNet50
from keras.applications.imagenet_utils import preprocess_input
from keras.utils import to_categorical
from keras.optimizers import RMSprop, Adam
```

```
D:\Jupiter\Lib\site-packages\paramiko\transport.py:219: CryptographyDeprecationWarning: Blowfish has been deprecated
  "class": algorithms.Blowfish,
```

```
In [2]: file_path = 'spectrograms.h5'

with h5py.File(file_path, 'r') as f:

    bkcchi_shape = f['bkcchi'].shape
    print("Shape of 'Black-capped Chickadee':", bkcchi_shape)


    daejun_shape = f['daejun'].shape
    print("Shape of 'Dark-eyed Junco:", daejun_shape)
```

```
Shape of 'Black-capped Chickadee': (256, 343, 57)
Shape of 'Dark-eyed Junco: (256, 343, 58)
```

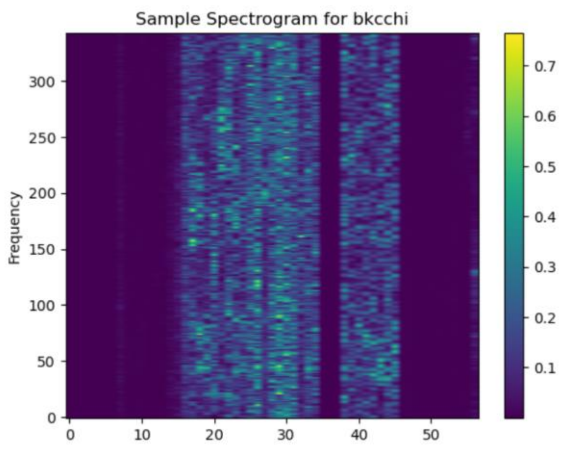
```
In [25]: # Load the spectrogram data

f = h5py.File(file_path, 'r')

# Inspect the keys in the dataset
species_keys = list(f.keys())
print(species_keys)
```

```
['amecro', 'barswa', 'bkcchi', 'blujay', 'daejun', 'houfin', 'mallar3', 'norfli', 'rewbla', 'stejay', 'wesmea', 'whcspa']
```

```
In [6]: # Select one species to inspect
selected_species_key = species_keys[2]
dset = f[selected_species_key]

# Check the shape of the dataset
print(dset.shape)
```

```
(256, 343, 57)
```

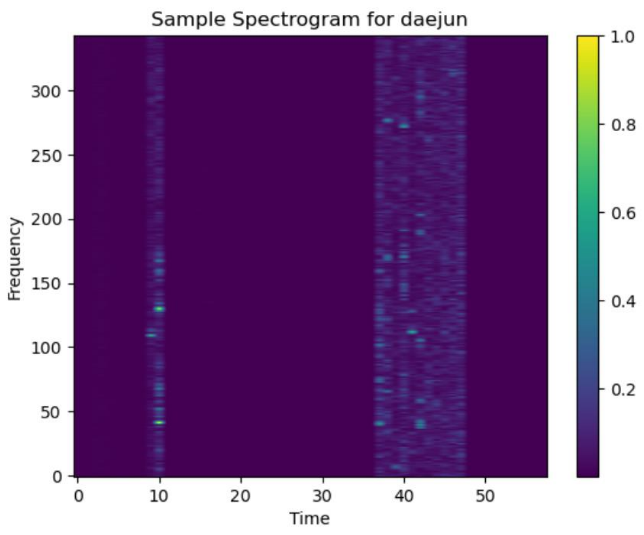
```
In [7]: # Visualize a sample spectrogram
sample_spectrogram = dset[2]
plt.imshow(sample_spectrogram, aspect='auto', origin='lower')
plt.title(f'Sample Spectrogram for {selected_species_key}')
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.colorbar()
plt.show()
```



In [8]:
```
# Select one species to inspect
selected_species_key = species_keys[4]
dset = f[selected_species_key]

# Check the shape of the dataset
print(dset.shape)
```

```
(256, 343, 58)
```

In [9]:
```
# Visualize a sample spectrogram
sample_spectrogram = dset[4]
plt.imshow(sample_spectrogram, aspect='auto', origin='lower')
plt.title(f'Sample Spectrogram for {selected_species_key}')
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.colorbar()
plt.show()
```



Binary Model

In [4]:
```
with h5py.File(file_path, 'r') as f:

    bkcchi_data = np.transpose(f['bkcchi'][:], (2, 0, 1))[..., np.newaxis]
    daejun_data = np.transpose(f['daejun'][:], (2, 0, 1))[..., np.newaxis]
    print("Reshaped 'Black-capped Chickadee':", bkcchi_data.shape)
    print("Reshaped 'Dark-eyed Junco':", daejun_data.shape)
```

```
Reshaped 'Black-capped Chickadee': (57, 256, 343, 1)
Reshaped 'Dark-eyed Junco': (58, 256, 343, 1)
```

In [5]:
```
# Normalize the data by the maximum value in the dataset
bkcchi_data_norm = bkcchi_data / np.max(bkcchi_data)
daejun_data_norm = daejun_data / np.max(daejun_data)
```

In [6]:
```
# Create labels for 'bkcchi' and 'daejun' species, Labels: 0 for 'bkcchi', 1 for 'daejun'
labels_bkcchi = np.zeros(bkcchi_data_norm.shape[0])
labels_daejun = np.ones(daejun_data_norm.shape[0])

# Concatenate data and labels
data = np.concatenate((bkcchi_data_norm, daejun_data_norm), axis=0)
labels = np.concatenate((labels_bkcchi, labels_daejun), axis=0)
```

In [7]:
```
# Perform train-test split
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)
```

In [14]:
```
# Build and Train Binary Classification Model

binary_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(256, 343, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

```
D:\Jupiter\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to
a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [15]:
```
binary_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

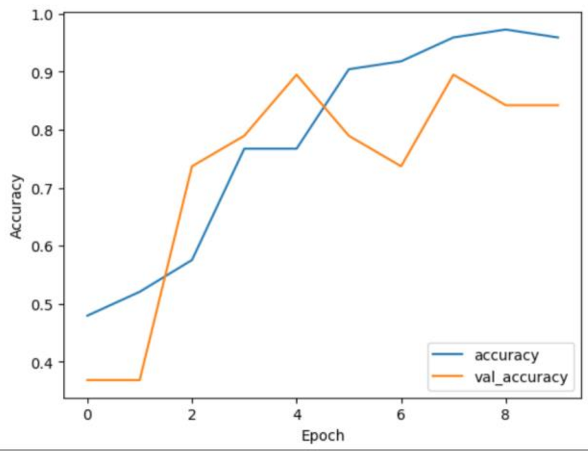
In [16]:
```
history = binary_model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

```
Epoch 1/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 30s 13s/step - accuracy: 0.4915 - loss: 0.7029 - val_accuracy: 0.3684 - val_loss: 0.7170
Epoch 2/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 21s 10s/step - accuracy: 0.5137 - loss: 0.6969 - val_accuracy: 0.3684 - val_loss: 0.7664
Epoch 3/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 19s 10s/step - accuracy: 0.5659 - loss: 0.6576 - val_accuracy: 0.7368 - val_loss: 0.5348
Epoch 4/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 23s 11s/step - accuracy: 0.7614 - loss: 0.5428 - val_accuracy: 0.7895 - val_loss: 0.5408
Epoch 5/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 17s 9s/step - accuracy: 0.7666 - loss: 0.4577 - val_accuracy: 0.8947 - val_loss: 0.4234
Epoch 6/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 16s 10s/step - accuracy: 0.9048 - loss: 0.3516 - val_accuracy: 0.7895 - val_loss: 0.3868
Epoch 7/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 22s 16s/step - accuracy: 0.9140 - loss: 0.2110 - val_accuracy: 0.7368 - val_loss: 0.4080
Epoch 8/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 19s 10s/step - accuracy: 0.9570 - loss: 0.1427 - val_accuracy: 0.8947 - val_loss: 0.3328
Epoch 9/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 20s 15s/step - accuracy: 0.9713 - loss: 0.0958 - val_accuracy: 0.8421 - val_loss: 0.3487
Epoch 10/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 22s 11s/step - accuracy: 0.9622 - loss: 0.1080 - val_accuracy: 0.8421 - val_loss: 0.3020
```

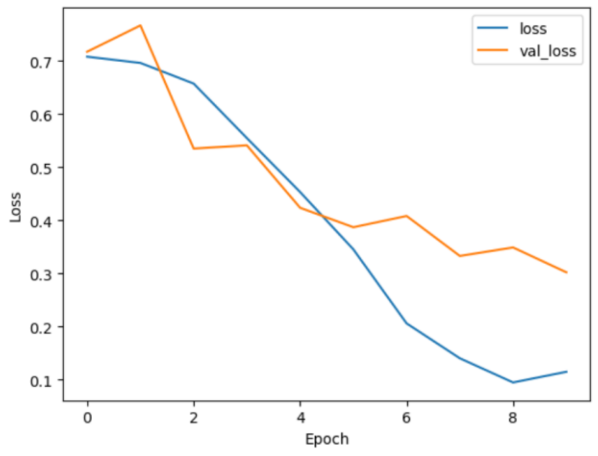
In [17]:
```
# Evaluating the binary model on the validation set.
score = binary_model.evaluate(X_test, y_test, verbose=0)
print("Test Accuracy: {:.2f}%".format(score[1]*100))
```

```
Test Accuracy: 86.96%
```

In [18]:
```
# Plot training & validation accuracy values

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['accuracy', 'val_accuracy'], loc='lower right')
plt.show();
```



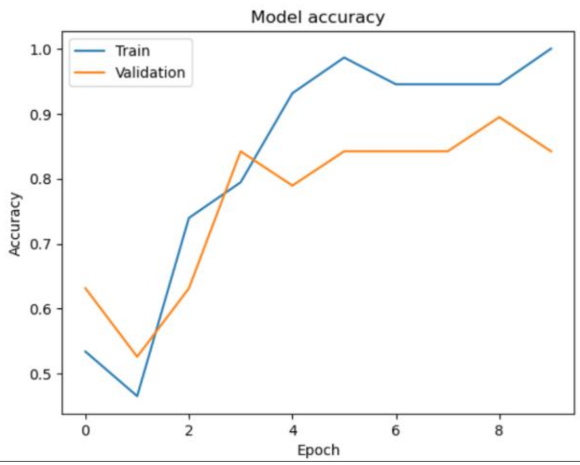
In [19]:
```
# Plot training & validation loss values

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.show();
```



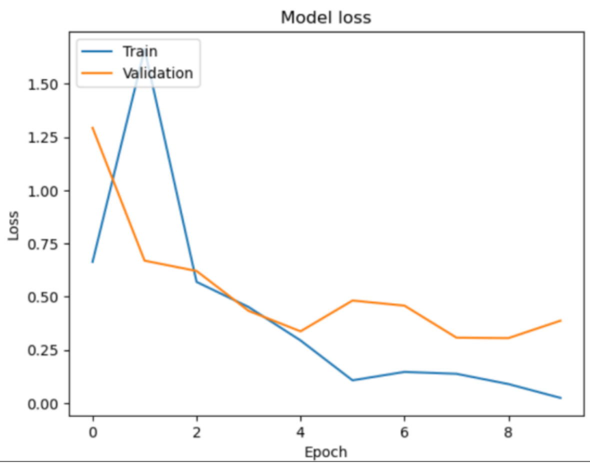
In [9]:
```
# Hyperparameter tuning binary

# Binary Model Hyperparameter Comparison
# Define a list of hyperparameters to tune
batch_sizes = [8, 16, 32]
filter_sizes = [(3, 3)]
num_filters = [8, 16, 32]
dense_units = [16, 32]
dropout_rates = [0.3, 0.5]

# Initialize variables to store the best hyperparameters and corresponding performance
best_accuracy_binary = 0
best_hyperparameters_binary = {}

# Iterate over hyperparameter combinations
for batch_size in batch_sizes:
    for filter_size in filter_sizes:
        for num_filter in num_filters:
            for dense_unit in dense_units:
                for dropout_rate in dropout_rates:
                    # Build the model
                    binary_model = Sequential([
                        Conv2D(num_filter, filter_size, activation='relu', input_shape=(256, 343, 1)),
                        MaxPooling2D((2, 2)),
                        Conv2D(num_filter*2, filter_size, activation='relu'),
                        MaxPooling2D((2, 2)),
                        Conv2D(num_filter*4, filter_size, activation='relu'),
                        Flatten(),
                        Dense(dense_unit, activation='relu'),
                        Dropout(dropout_rate),
                        Dense(1, activation='sigmoid')
                    ])

                    # Compile the model
                    binary_model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

                    # Train the model
                    history = binary_model.fit(X_train, y_train, epochs=10, batch_size=batch_size, validation_split=0.2, verbose=0)

                    # Evaluate the model on the validation set
                    accuracy = history.history['val_accuracy'][-1]

                    # Check if this combination of hyperparameters improved performance
                    if accuracy > best_accuracy_binary:
                        best_accuracy_binary = accuracy
                        best_hyperparameters_binary = {
                            'batch_size': batch_size,
                            'filter_size': filter_size,
                            'num_filter': num_filter,
                            'dense_unit': dense_unit,
                            'dropout_rate': dropout_rate
                        }

# Print the best hyperparameters and corresponding performance for the binary model
print("Best Hyperparameters for Binary Model:")
print(best_hyperparameters_binary)
print("Best Validation Accuracy:", best_accuracy_binary)
```

```
Best Hyperparameters for Binary Model:
{'batch_size': 16, 'filter_size': (3, 3), 'num_filter': 16, 'dense_unit': 16, 'dropout_rate': 0.3}
Best Validation Accuracy: 0.9473684430122375
```

In [10]:
```
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



In [11]:
```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Multiclass Model

In [13]:
```
# Load the data
file_path = 'spectrograms.h5'

with h5py.File(file_path, 'r') as f:
    num_classes = len(f.keys())

    spectrogram_data = []
    spectrogram_labels = []

    for idx, key in enumerate(f.keys()):
        data = f[key][:]
        data = np.transpose(data, (2, 0, 1))  # Transpose to match the desired shape
        data = np.expand_dims(data, axis=-1)  # Add a channel dimension
        data = data / np.max(data)  # Normalize the data

        labels = np.full((data.shape[0],), idx)  # Create labels

        spectrogram_data.append(data)
        spectrogram_labels.append(labels)

    spectrogram_data = np.concatenate(spectrogram_data, axis=0)
    spectrogram_labels = np.concatenate(spectrogram_labels, axis=0)
    spectrogram_labels = to_categorical(spectrogram_labels, num_classes=num_classes)

# Split data into training and testing sets
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(spectrogram_data, spectrogram_labels, test_size=0.2, random_state=42)
```

In [51]:
```
multiclass_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(256, 343, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

# Compile the model
multiclass_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

In [52]:
```
history = multiclass_model.fit(X_train_1, y_train_1, epochs=10, batch_size=16, validation_split=0.2)
```

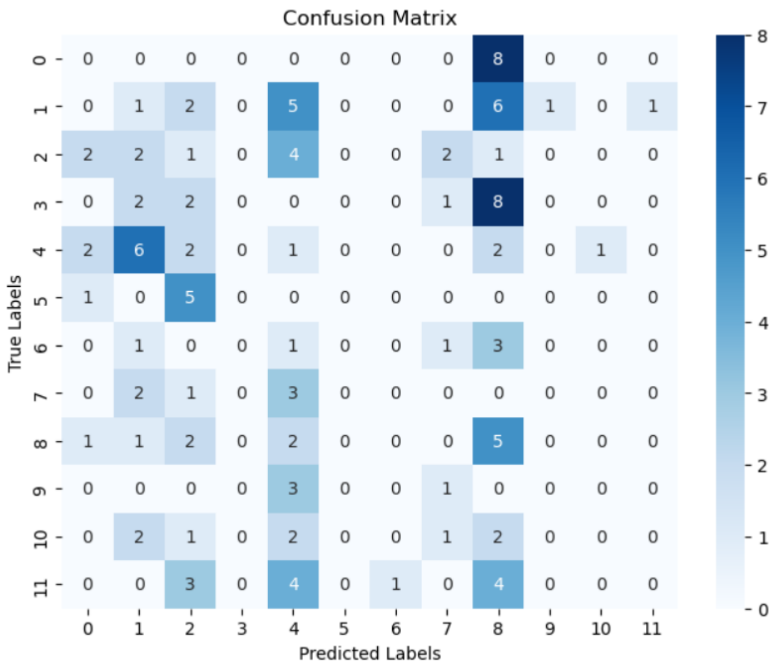
```
Epoch 1/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 71s 2s/step - accuracy: 0.2395 - loss: 2.3690 - val_accuracy: 0.4086 - val_loss: 1.9402
Epoch 2/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 31s 1s/step - accuracy: 0.5843 - loss: 1.5042 - val_accuracy: 0.4839 - val_loss: 1.4069
Epoch 3/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 40s 2s/step - accuracy: 0.7048 - loss: 0.9178 - val_accuracy: 0.5914 - val_loss: 1.3711
Epoch 4/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.8108 - loss: 0.5923 - val_accuracy: 0.5806 - val_loss: 1.5439
Epoch 5/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 33s 1s/step - accuracy: 0.8720 - loss: 0.4605 - val_accuracy: 0.6452 - val_loss: 1.3947
Epoch 6/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.8659 - loss: 0.4025 - val_accuracy: 0.6344 - val_loss: 1.3942
Epoch 7/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 33s 1s/step - accuracy: 0.9221 - loss: 0.2824 - val_accuracy: 0.6989 - val_loss: 1.4576
Epoch 8/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.9139 - loss: 0.2672 - val_accuracy: 0.6452 - val_loss: 1.6020
Epoch 9/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.9572 - loss: 0.1468 - val_accuracy: 0.6559 - val_loss: 1.8892
Epoch 10/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.9644 - loss: 0.1461 - val_accuracy: 0.6989 - val_loss: 1.9521
```

In [54]:
```python
# Evaluate the model on test data
score = multiclass_model.evaluate(X_test_1, y_test_1, verbose=0)
print("Test Accuracy: {:.2f}%".format(score[1] * 100))
```

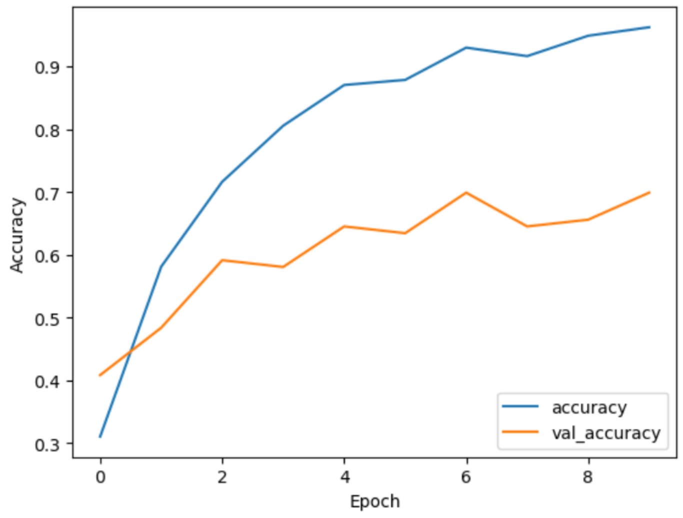
```
Test Accuracy: 70.69%
```

In [25]:
```python
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict classes for test data
y_pred = np.argmax(multiclass_model.predict(X_test_1), axis=1)

# Convert one-hot encoded labels back to categorical labels
y_true = np.argmax(y_test_1, axis=1)

# Generate confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=range(num_classes), yticklabels=range(num_classes))
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()


```

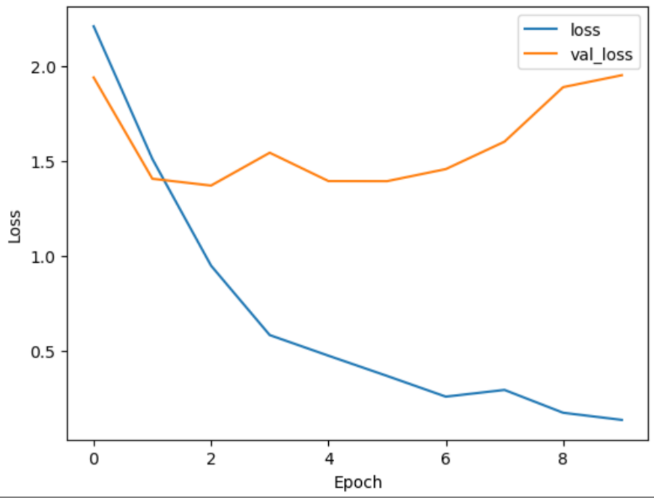
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 6s 432ms/step
```



In [55]:
```
# Plot training & validation accuracy values

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['accuracy', 'val_accuracy'], loc='lower right')
plt.show();
```



In [56]:
```
# Plot training & validation loss values

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.show();
```

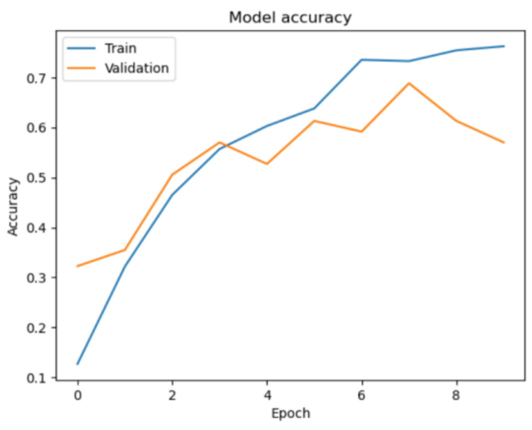


In [15]:
```python
# Multiclass Model Hyperparameter Comparison

# Define a list of hyperparameters to tune
batch_sizes_multi = [8, 16, 32]
filter_sizes_multi = [(3, 3)]
num_filters_multi = [8, 16, 32]
dense_units_multi = [16, 32]
dropout_rates_multi = [0.3, 0.5]

# Initialize variables to store the best hyperparameters and corresponding performance
best_accuracy_multiclass = 0
best_hyperparameters_multiclass = {}

# Iterate over hyperparameter combinations
for batch_size_multi in batch_sizes_multi:
    for filter_size_multi in filter_sizes_multi:
        for num_filter_multi in num_filters_multi:
            for dense_unit_multi in dense_units_multi:
                for dropout_rate_multi in dropout_rates_multi:
                    # Build the model
                    multiclass_model_2 = Sequential([
                        Conv2D(num_filter_multi, filter_size_multi, activation='relu', input_shape=(256, 343, 1)),
                        MaxPooling2D((2, 2)),
                        Conv2D(num_filter_multi*2, filter_size_multi, activation='relu'),
                        MaxPooling2D((2, 2)),
                        Conv2D(num_filter_multi*4, filter_size_multi, activation='relu'),
                        Flatten(),
                        Dense(dense_unit_multi, activation='relu'),
                        Dropout(dropout_rate_multi),
                        Dense(12, activation='softmax')
                    ])
```

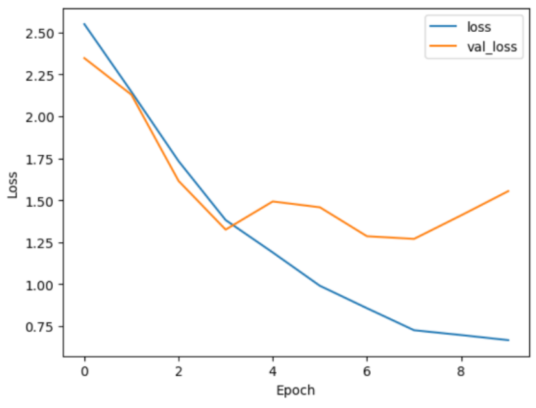
```python
                    # Compile the model
                    multiclass_model_2.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

                    # Train the model
                    history_multi = multiclass_model_2.fit(X_train_1, y_train_1, epochs=10, batch_size=batch_size_multi, validation_split=0.2, verbose=0)

                    # Evaluate the model on the validation set
                    accuracy_multi = history_multi.history['val_accuracy'][-1]

                    # Check if this combination of hyperparameters improved performance
                    if accuracy_multi > best_accuracy_multiclass:
                        best_accuracy_multiclass = accuracy_multi
                        best_hyperparameters_multiclass = {
                            'batch_size': batch_size_multi,
                            'filter_size': filter_size_multi,
                            'num_filter': num_filter_multi,
                            'dense_unit': dense_unit_multi,
                            'dropout_rate': dropout_rate_multi
                        }

# Print the best hyperparameters and corresponding performance for the multiclass model
print("Best Hyperparameters for Multiclass Model:")
print(best_hyperparameters_multiclass)
print("Best Validation Accuracy:", best_accuracy_multiclass)
```

```
Best Hyperparameters for Multiclass Model:
{'batch_size': 16, 'filter_size': (3, 3), 'num_filter': 8, 'dense_unit': 32, 'dropout_rate': 0.3}
Best Validation Accuracy: 0.7526881694793701
```

In [34]:
```python
# Plot training & validation accuracy values
plt.plot(history_multi.history['accuracy'])
plt.plot(history_multi.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



In [35]:
```
# Plot training & validation loss values

plt.plot(history_multi.history['loss'])
plt.plot(history_multi.history['val_loss'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.show();
```



External Test

In [1]:
```
# Dictionary that maps each species code to its full name.

species_full_names = {
    'amecro': 'American Crow',
    'barswa': 'Barn Swallow',
    'bkcchi': 'Black-capped Chickadee',
    'blujay': 'Blue Jay',
    'daejun': 'Dark-eyes Junco',
    'houfin': 'House Finch',
    'mallar3': 'Mallard',
    'norfli': 'Northern Flicker',
    'rewbla': 'Red-winged Blackbird',
    'stejay': "Steller's Jay",
    'wesmea': 'Western Meadowlark',
    'whcspa': 'White-crowned Sparrow'
}
```

In [2]:
```
import librosa
import soundfile as sf

# Define a function to preprocess audio data
def preprocess_audio(audio_path):
    # Load audio file
    y, sr = librosa.load(audio_path, sr=22050)

    # Extract spectrogram
    spectrogram = librosa.feature.melspectrogram(y=y, sr=sr)
    spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
    spectrogram = resize(spectrogram, (256, 343), anti_aliasing=True)

    # Reshape to match the input shape of the models
    spectrogram = spectrogram.reshape(1, 256, 343, 1)

    return spectrogram
```

In [3]:
```
# Function to predict bird species using the multi-class model
def predict_multiclass(audio_path):
    # Preprocess audio data
    spectrogram = preprocess_audio(audio_path)

    # Predict using the multi-class model
    prediction = multiclass_model_2.predict(spectrogram)

    # Convert prediction to bird species
    predicted_class = np.argmax(prediction)
    predicted_species = species_full_names[species_keys[predicted_class]]

    return predicted_species
```

In [4]:
```
# Define the paths to the external test audio files
import os

# Define the directory containing the external test audio files
audio_dir = 'test_birds'

# Define the paths to the external test audio files
audio_paths = [os.path.join(audio_dir, filename) for filename in os.listdir(audio_dir)]
```

In [30]:
```
# Predict bird species for each audio file using multiclass model

multiclass_predictions = [predict_multiclass(audio_path) for audio_path in audio_paths]
```

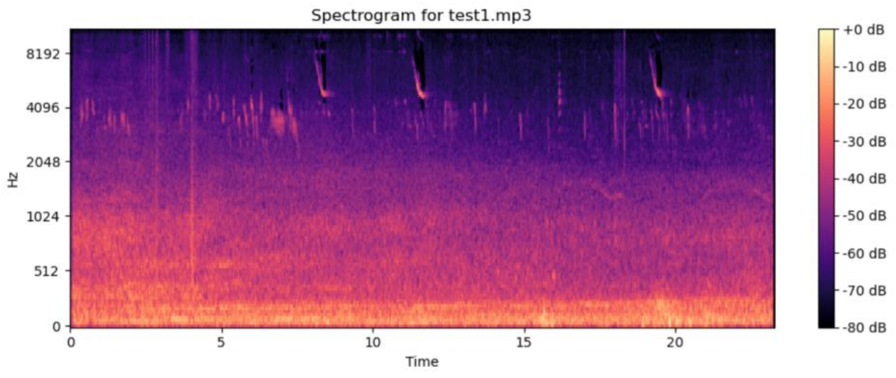
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 138ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 122ms/step
```

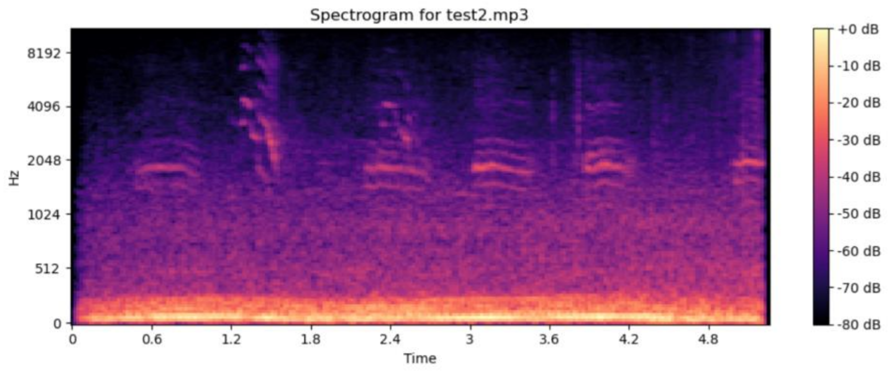
In [31]:
```python
# Print the predictions

print("\nMulti-class Model Predictions:")
for audio_path, prediction in zip(audio_paths, multiclass_predictions):
    print(f"Audio File: {audio_path}, Predicted Species: {prediction}")
```

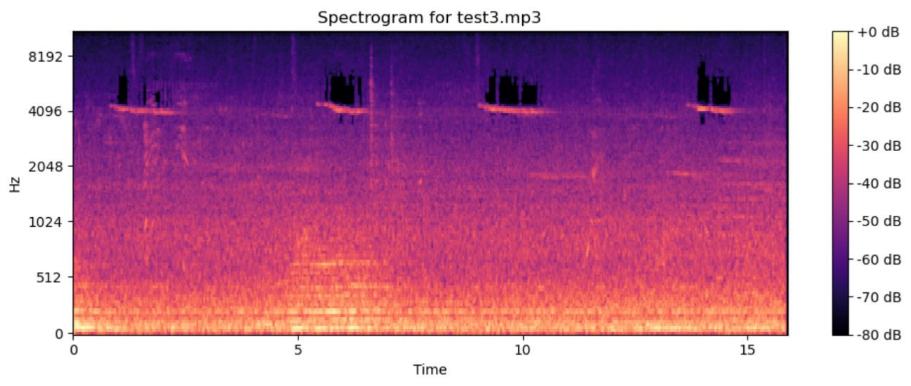
```
Multi-class Model Predictions:
Audio File: test_birds\test1.mp3, Predicted Species: Dark-eyes Junco
Audio File: test_birds\test2.mp3, Predicted Species: Northern Flicker
Audio File: test_birds\test3.mp3, Predicted Species: Dark-eyes Junco
```

In [7]:
```python
# Function to plot spectrogram of an audio file
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
import os
from skimage.transform import resize

def plot_spectrogram(audio_path):
    y, sr = librosa.load(audio_path, sr=22050)
    S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)
    S_dB = librosa.power_to_db(S, ref=np.max)

    plt.figure(figsize=(10, 4))
    librosa.display.specshow(S_dB, sr=sr, x_axis='time', y_axis='mel')
    plt.colorbar(format='%+2.0f dB')
    plt.title(f'Spectrogram for {os.path.basename(audio_path)}')
    plt.tight_layout()
    plt.show()

# Plot the spectrograms for the test audio files
for audio_path in audio_paths:
    plot_spectrogram(audio_path)
```

```
D:\Jupiter\Lib\site-packages\paramiko\transport.py:219: CryptographyDeprecationWarning: Blowfish has been deprecated
  "class": algorithms.Blowfish,
```


Spectrogram for test1.mp3
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


Spectrogram for test2.mp3
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


Spectrogram for test3.mp3
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB