TweetyNet: Eco-Acoustic Event Detection Pipeline

Introduction to Eco-acoustic Event Detection

It is important to understand the health of the planet's ecosystems in order to help conserve them. One way that has been employed to assess the health of ecosystems is audio event detection, which uses the sound generated by animals to detect their presence and indirectly gauge the health of an ecosystem and is the goal of the TweetyNet machine learning model. Audio event detection has benefited conservation efforts because it allows researchers to automate the task of detecting the presence of birds with the use of machine learning techniques. A brief example of audio event detection in everyday use is hands-free interfacing, which detects the presence of spoken voice commands in order to perform the desired action. Eco-acoustic event detection is audio event detection with the intended use of detecting sounds that are characteristic of an ecosystem and can be used to assess the health of ecosystems. One of the most useful indicators of an ecosystem's health is bird vocalizations, which are relatively easy to separate from the background noise of an environment. The usefulness of detecting bird presence through their vocalizations presents a challenge, as manual inspection and annotation of audio data is a painstaking process. Through the use of machine learning, we can anticipate a reduction in time spent manually annotating and identifying birds' vocalizations which will allow researchers to monitor and learn about ecosystems with greater accuracy and efficiency. Birds inhabit and share nearly every environmental niche and are more sensitive to ecological change than other kinds of animals and so are capable of representing the biodiversity of ecosystems disturbed by deforestation and climate change. By applying eco-acoustic event detection to birds, it can also be adapted to the detection of other so-called 'indicator' species. Beyond measuring ecological health, eco-acoustic detection can accumulate environmental information, refine our understanding, and reflect on the impact of human activity on our planet's health. All of these goals are possible applications of the TweetyNet model.

Methodology

Machine learning paired with signal processing methods allows for audio data to be processed and learned as image data via conversion to Mel-spectrograms which can be used to train TweetyNet, a convolutional and recurrent neural net hybrid model built for the purpose of event detection in audio data [1]. Initial research for this model's potential to be repurposed for eco-acoustic event detection presented itself as a challenge since there was no ready-to-use pipeline for replicating preliminary results. Many of the model's methods discussed in the paper such as spectrogram windowing and data ingestion for audio of various lengths were not available aside from the neural network architecture which was available in the original author's Github repository. The model has been replicated using the paper Pytorch and adapted to use both CPU and GPU processing. Audio data ingestion has also been generalized to intake various lengths of recordings from various species for the purpose of diversifying extracted features of bird vocalizations in the convolutional layers as well as temporal dependent features learned in the recurrent layers [1]. This replicate model has proven itself an excellent potential tool for remote sensing as an eco-acoustic event detection method. When compared to published results that accurately detect the presence of bird vocalizations trained on high-quality annotations from 3 species [1]. Previous iterations of the replicate model have averaged accuracy of .6 - .65 to now .88-.89, which is well within the range of accuracy discussed in the results of the published work [1].

Audio files are first converted to spectrograms and sliced into windows of a predetermined length in seconds using fast Fourier transformation and signal processing methods; the time bins in these windows are time bin frames that are later classified for either presence or absence of vocalization [1]. The windows resulting classified annotations should match with what is visually present in the windowed spectrogram image [1]. In the same iteration of this windowing function, corresponding annotator labels and unique species ID are sourced from a data frame of annotations which then get grouped with these windows and corresponding time bins in a dictionary. After being windowed to their new respective lengths, this object containing the information per window of predetermined length for all audio data ingested is then split into training validation and testing using a split of 70:10:20. If this data has been previously processed then these splits can be loaded due to a condition in the data load function that checks for its expected file path to determine if this preprocessing step is necessary. After splitting, a GPU device should be configured using the "torch.cuda.device" function. Training, validation, and test splits are then converted to float sensors, their corresponding labels are converted to long tensors and sent to the determined device using the ".to" function. These tensors are then passed into a CustomAudioDataset function which is a wrapper method for returning the corresponding windows, labels, time frames, and UIDS. If you wanted to display these values, i.e spectrogram of a window in the windows array, you would need to send them back to the CPU device then detach them from their computational graph in the GPU device this then allows NumPy conversion. Since these returned values are tensors and Librosa requires a NumPy array to display a spectrogram, they can be displayed. It's important I mention this seemingly irrelevant detail since the correct format of the data being processed has proven itself necessary to the GPUs success. Previous iterations of the model followed the assumption that

GPU could process non-tensor type data but gets rerouted to CPU without explicit notice and that the data would be sent to the GPU when passed into the model's training pipeline function these were pitfalls that kept the progress of the project stagnant for a short period of time. However, it was an important learning experience for a first-time deep neural network replication project.

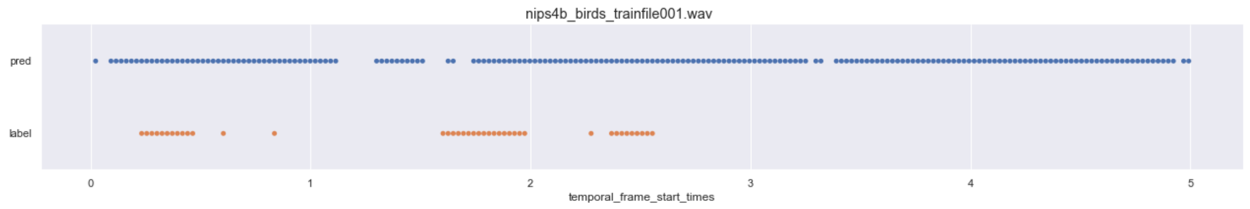
Now having the data in the correct format and ported to the desired hardware, designating hardware must also be done for the model. From here the model is ready for training, passed into the model's training pipeline will be the previously split train and validation dataset as well as parameters such as learning rate and batch size, and epochs which are configurable JSON variables in the configs folder. Both datasets will be partitioned in the "DataLoader" function according to the batch size; this is the number of windows trained on per epoch during the training and validation training cycles. A learning rate optimizer scheduler is also instantiated after passing the splits into the data loader. Using the function "OneCycleLR" takes as parameter input the models selected adam(adaptive) optimizer, learning rate, total epochs, number of batches per epoch, and a linear aneal-decay strategy for faster convergence when approximating global optimum. Both data loader objects(batched datasets), scheduler, and total epochs are passed into the training.

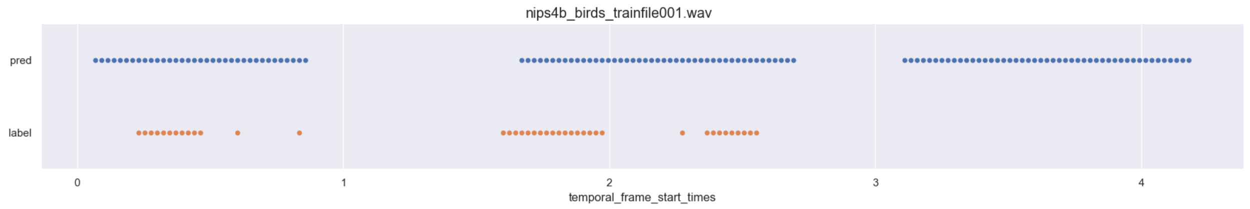
For the set number of epochs, the batched training and validation datasets are unpacked from their data loader for their float tensor representations of windowed spectrograms set to input and long tensor representations of annotation labels set to labels. The inputs are then passed into the model for feature extraction in the convolutional layer and sequentially learned via the recurrent layer. The cross-entropy loss is then computed between the output from passing this batched input into the model and the true batched labels. This loss is then back propagated to adjust parameter tensor values through the use of the adam optimizer; which updates selected model parameter tensors by using the gradients stored in the same parameter tensors when “loss.backward()” is called. Linear annealing-decay is applied from the scheduler mentioned previously to adjust the learning rate, this adjustment occurs per epoch. A similar process is shared with the validation dataset returning recorded best model weights at each epoch determined by best validation accuracy scores in those epochs.

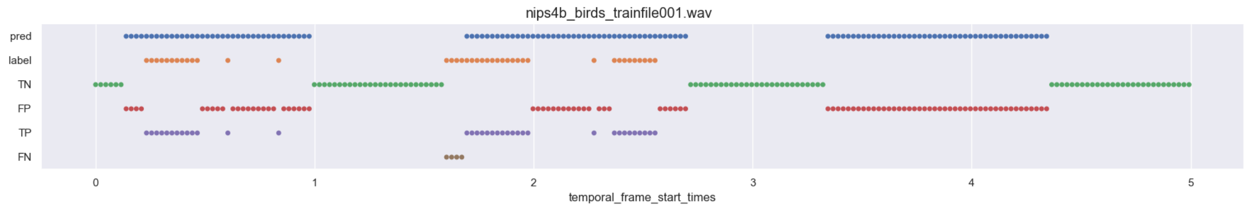
The model now trained can generate temporal classifications on the test dataset. In the testing cycle, the model takes into account the recorded best model weights that were generated from the training cycle and is loaded into the model through the PyTorch built-in “load_state_dict” function in the neural network module.py file. Similar to before, the test data is passed into a data loader for batch size partitioning. The test data loader, selected hop length, sampling rate, and window size are passed as inputs for the model testing function. The hop length and sampling rate are used to calculate the number of rows corresponding to the window's time bin frames which are then labeled either 1 being a classification of presence or 0, absence. These time bin frames in the window spectrogram match in length to the model’s Long Short Term Memory units in the recurrent layer, “The number of the LSTMs units is equivalent to the number of time bins in the windowed spectrogram”[1], this is intentional to learn the temporal dependencies through forward and backward passes in the recurring hidden cell states being updated with relevant features to reference in the future in the recurrent layer [1]. During training, in the convolutional layer features are originally extracted as temporal independent features [1]. During classification, these features are used as learned filters in the convolutional layer when the best model weights determined by the validation accuracy are loaded. From these fresh learned and referenced temporal dependent features, i.e. the vocalizations or environmental noise, an influence matrix is returned for these temporal features and their presence and absence classification [1]. This results in a vector of sequential time bins and their corresponding probability that a time bin frame t in the window is either a learned vocalization that can be found in the annotation’s presence labels or a learned feature of environmental noise in the annotation's absence labels [1]. The window's predictions are then concatenated together to the original spectrogram's length; this can then be displayed and be used for clustering analysis. Such proposed monitoring analysis methods include learning what bird vocalizations are common to an area. To potentially observe if their presence over time decreases due to known ecosystem disturbances, this data then can be used for data-driven conservation strategies and wildlife ecosystem protection enactments.

Results and Discussion

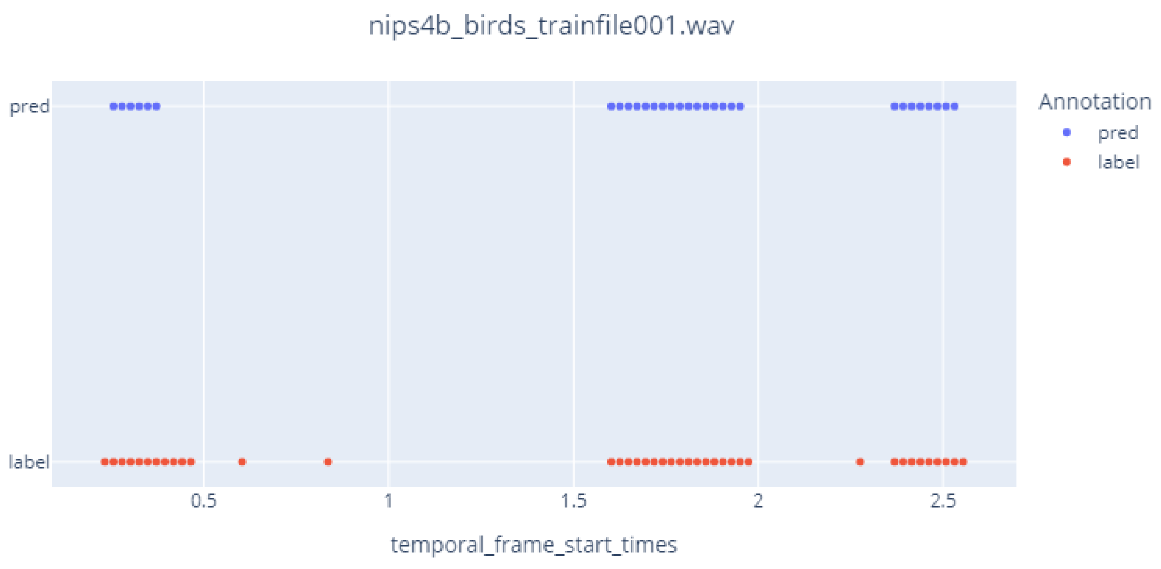
Preliminary results: spectrogram windowing and GPU training not implemented

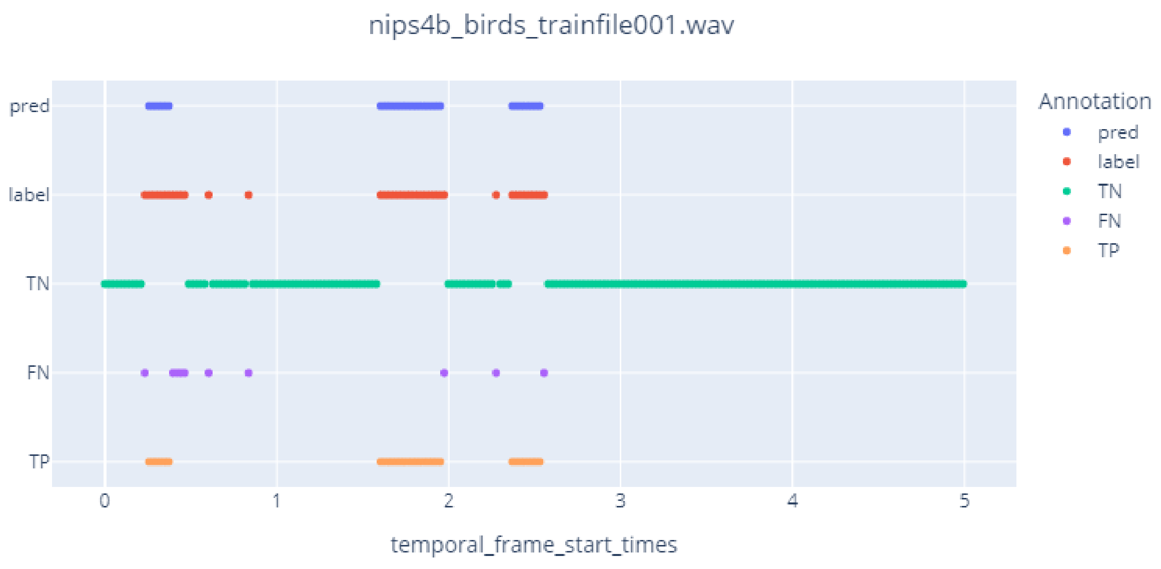




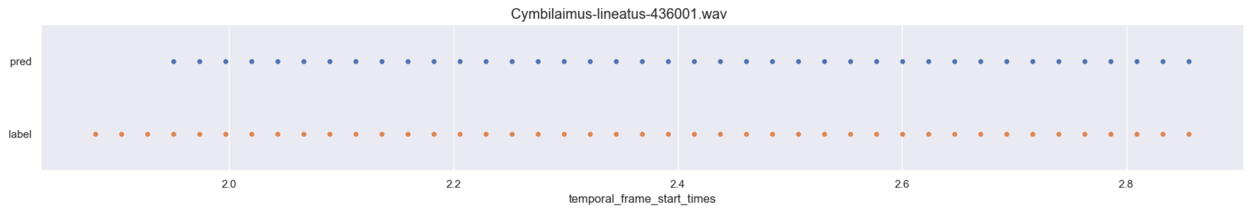


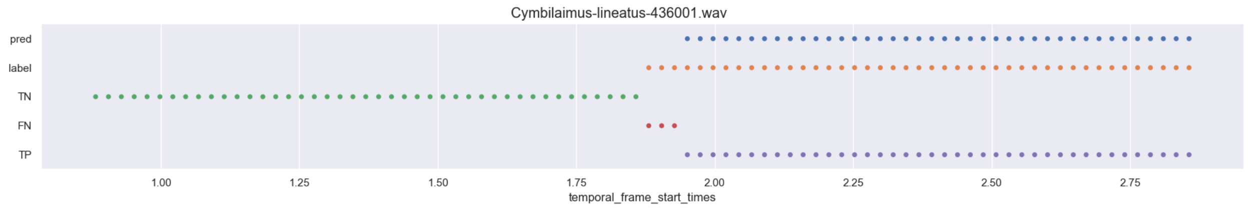
Final results: for the zoom function, interactive plots are available on the website.



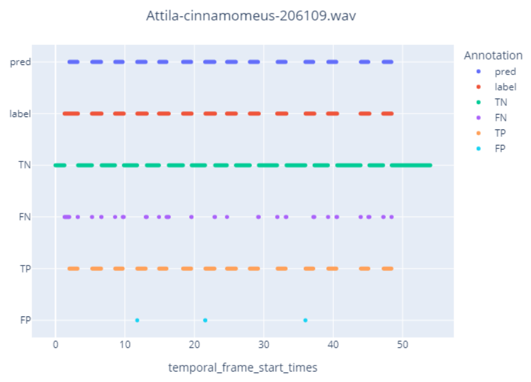


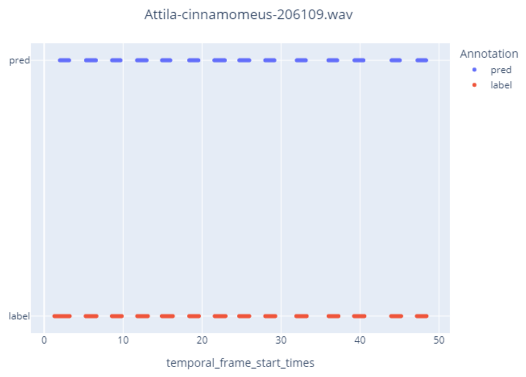
- Both nips files are predictions resulting from CPU, windowing implemented in the model and improved detail to training per file.



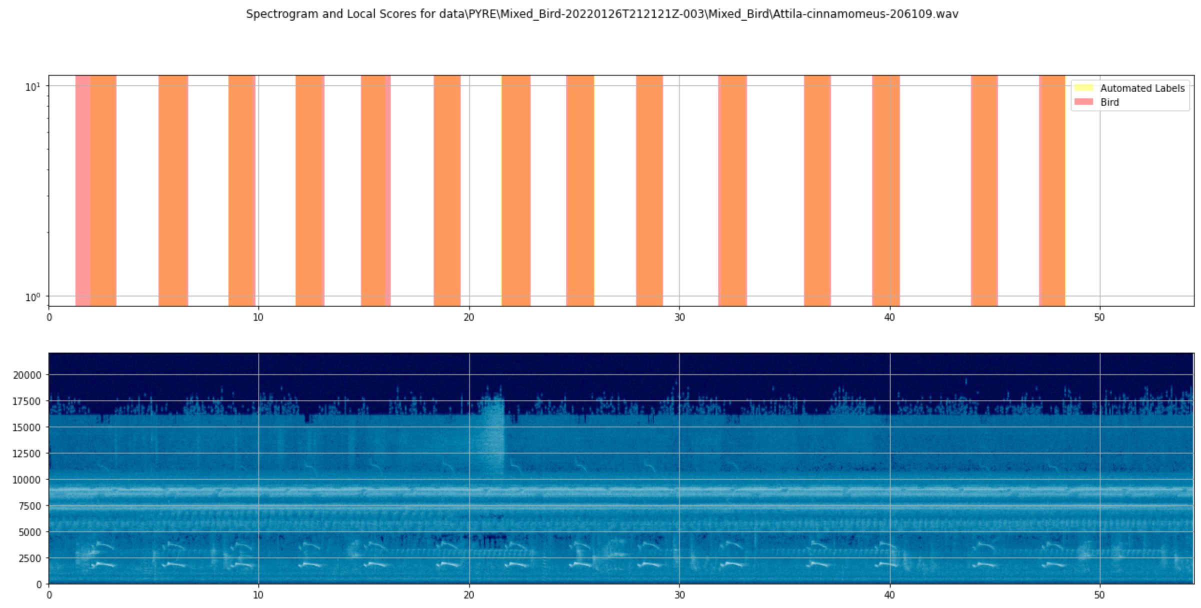


- CPU model resulting classifications when trained on Pyre Note data, before using the Plotly library for interactive visualizations.





- Pyrenote classification resulting from the GPU Model(interactive plot on the website as well)



- PyHa Visualization on the Pyrenote classification resulting from the GPU Model
- Red streaks were left out from the yellow overlap, suggesting that the model learned features of noise that were present in the annotation.

Neural Network Training Process Meta-Analysis:

Average training time for GPU Model 1 min 10 sec and CPU Model 12 min 33 sec. Due to this drastic time difference, an unpooled two-sample T-test was conducted on the training accuracy and loss after 100 Epochs to validate exact implementation between the two models' training process by investigating whether the means of both training samples (Acc/Loss) differ from one another. In such a test, the null hypothesis is that the means of both groups are the same. Each model trained a total of 15 times for 100 epochs with a batch size of 64 and a learning rate of .005 training on the same files on the same computer. The accuracy and loss were recorded for this analysis and come from their last epoch training cycle. A total of 15 accuracies and losses are logged from the CPU model(Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz) and GPU model(NVIDIA GeForce GTX 1650 with Max-Q design).

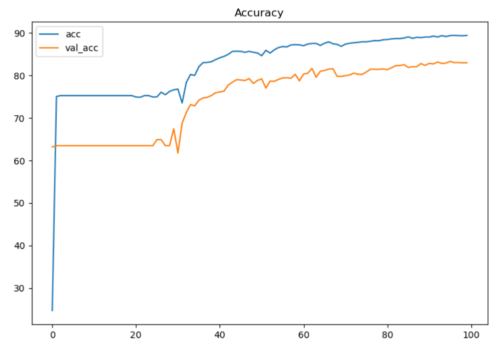
H_0 : We lack evidence to suggest that the training process is different between models

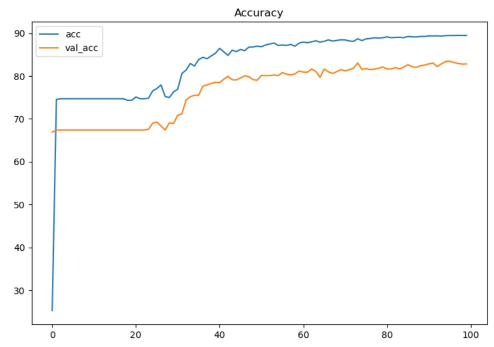
H_A : Evidence suggests that the training process is different between models

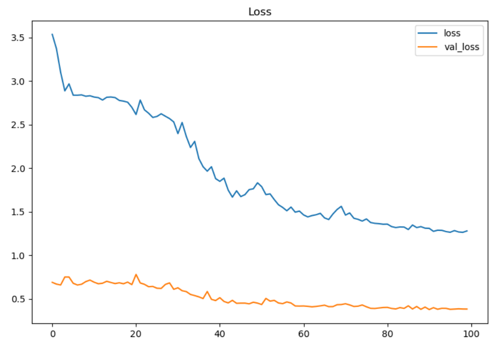
Significance Level: .05

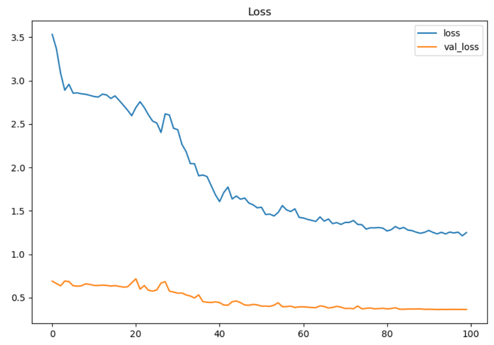
The resulting test for both models' training accuracy after 100 epochs yields a p-value of 0.1519. Using a 95% confidence level we can fail to reject the null hypothesis since the p-value is greater than the corresponding significance level of 5%.

The resulting test for both models' training loss after 100 epochs yields a p-value of 0.1877. Using a 95% confidence level we can also fail to reject the null hypothesis since the p-value is greater than the corresponding significance level of 5%

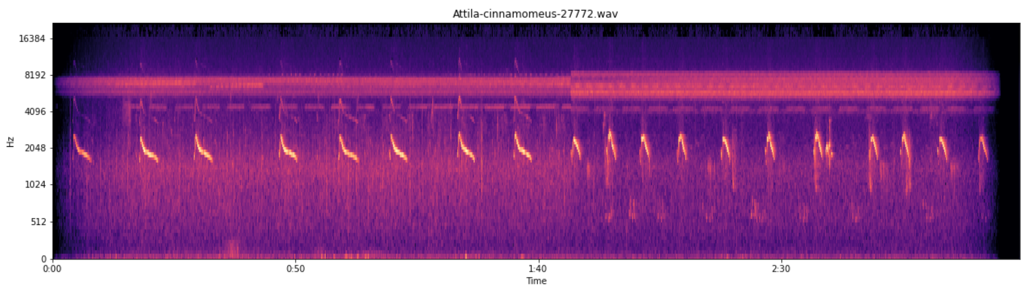


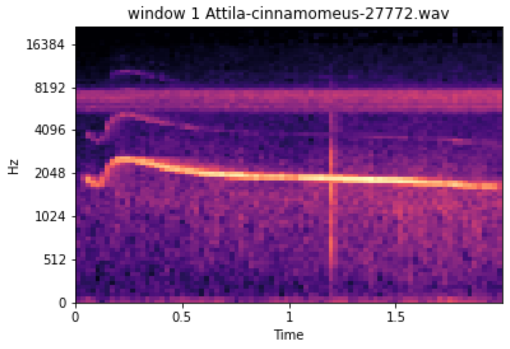


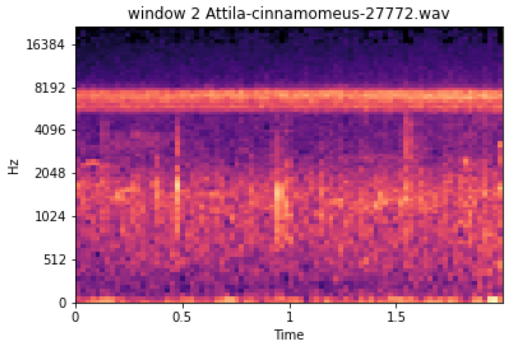


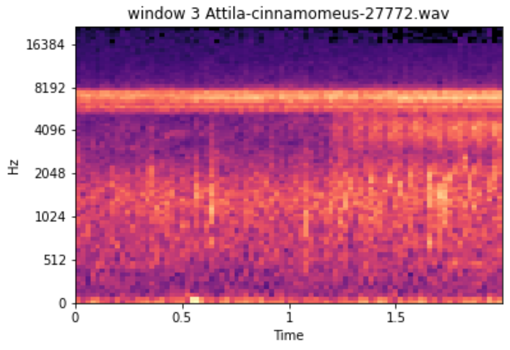


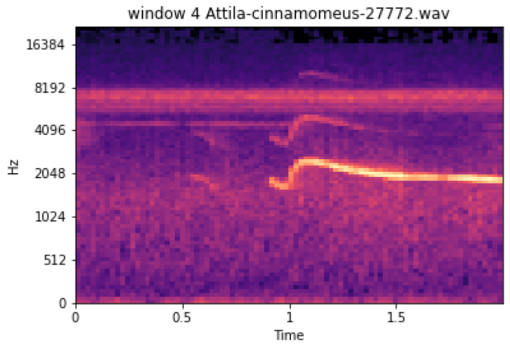
- Typical accuracy and loss graphs for a 100 epoch training cycle for both the GPU(left) and CPU(right) models are shown. Graph curvature demonstrates a close resemblance between the two models, suggesting the training process is the same in both models.

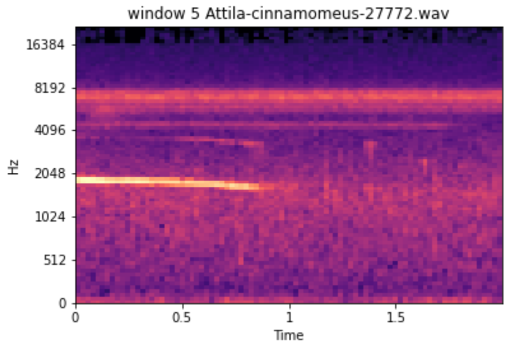

Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0:00
0:50
1:40
2:30
Time

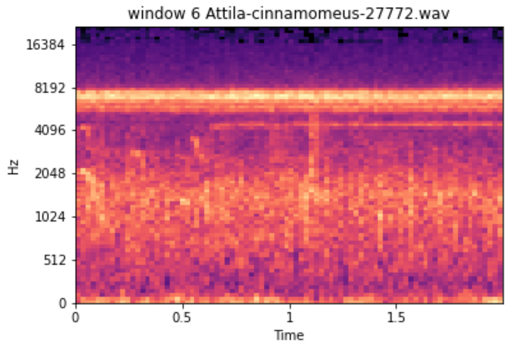

window 1 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time


window 2 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time


window 3 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time


window 4 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time


window 5 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time


window 6 Attila-cinnamomeus-27772.wav
16384
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
Time

Here we see a file of similar species. The one shown previously was used only for classification; this one here happened to be in the training split. But what can this inform us about the model? We can see here that this training file likely had detailed annotations, windows 2, 3, and 6 are perfect examples of either background noise or other species activity; it is likely they were not selected for containing a vocalization for the species of interest. This also involves the fact that annotators are tasked with annotating the species of interest during the annotation process. This file was in the train split for training; there are two files of this kind of species in the training set out of 221 files present in the training split. This observation portrays how well the recurrent layer is learning the extracted features sequentially and maintaining relevant features during the forward pass and backward pass learning through time via BPTT backpropagation through time. It can be confidently said that this model has potential for eco-acoustic event detection/remote sensing and where future development can further its potential for multi-class species temporal classifications.

Conclusion

Replicate model results show that the GPU does make an improvement in annotation quality and acceleration compared to the manual annotation process. Which was sought after by the original authors of this model. Eager to follow this model's paper to replicate results of our own, implementation of the unshared model PyTorch script has proven the process of building a deep learning model a slow and concentrated process. Although the original author's motivation to design such a hybrid CNN RNN model was unrelated to the domain of wildlife conservation, the replicate model demonstrates that such a robust model design can extend benefits across scientific research domains. This model holds potential for future research and development that can lead to data-driven conservation policies and monitoring strategies. If given another year to work on this project, proposed developments include adapting this model to classify and count the species present in a recording.

Works cited

1. Yarden Cohen, David Nicholson, Alexa Sanchioni, Emily K. Mallaber, Viktoriya Skidanova, Timothy J. Gardner Tweety Net: A neural network that enables high-throughput, automated annotation of birdsong bioRxiv 2020.08.28.272088; doi: https://doi.org/10.1101/2020.08.28.272088