opensoundscape

Release 0.4.5

Oct 20, 2020

Getting Started

OpenSoundsoundscape is free and open source software for the analysis of bioacoustic recordings. Its main goals are to allow users to train their own custom species classification models using a variety of frameworks (including convolutional neural networks) and to use trained models to predict whether species are present in field recordings. OpSo can be installed and run on a single computer or in a cluster or cloud environment.

OpenSoundcape is developed and maintained by the Kitzes Lab at the University of Pittsburgh.

The Getting Started section below provide guidance on installing OpSo. The Tutorials pages below are written as Jupyter Notebooks that can also be downloaded from the project repository on GitHub.

CHAPTER 1

OpenSoundscape

OpenSoundscape is a utility library for analyzing bioacoustic data. It consists of command line scripts for tasks such as preprocessing audio data, training machine learning models to classify vocalizations, estimating the spatial location of sounds, identifying which species' sounds are present in acoustic data, and more.

These utilities can be strung together to create data analysis pipelines. OpenSoundscape is designed to be run on any scale of computer: laptop, desktop, or computing cluster.

OpenSoundscape is currently in active development. If you find a bug, please submit an issue. If you have another question about OpenSoundscape, please email Sam Lapp (`sam.lapp` at `pitt.edu`) or Tessa Rhinehart (`tessa.rhinehart` at `pitt.edu`).

For examples of some of the utilities offered, please see the "Tutorials" section of the documentation. Included are instructions on how to download and use a pretrained machine learning model from our publicly available set of models. We plan to add additional tutorials soon.

CHAPTER 2

Installation

OpenSoundscape can be installed either via pip (for users) or poetry (for developers contributing to the code). Either way, Python 3.7 or higher is required.

2.1 Installation via pip (most users)

2.1.1 Just give me the pip command!

Already familiar with installing python packages via pip? The pip command to install OpenSoundscape is `pip install opensoundscape==0.4.5`.

2.1.2 Detailed instructions

Python 3.7 is required to run OpenSoundscape. Download it from this website.

We recommend installing OpenSoundscape in a virtual environment to prevent dependency conflicts. Below are instructions for installation with Python's included virtual environment manager, `venv`, but feel free to use another virtual environment manager (e.g. `conda`, `virtualenvwrapper`) if desired.

Run the following commands in your bash terminal:

- Check that you have installed Python 3.7._: `python3 --version`
- Change directories to where you wish to store the environment: `cd [path for environments folder]`
  - Tip: You can use this folder to store virtual environments for other projects as well, so put it somewhere that makes sense for you, e.g. in your home directory.
- Make a directory for virtual environments and `cd` into it: `mkdir .venv && cd .venv`
- Create an environment called `opensoundscape` in the directory: `python3 -m venv opensoundscape`

- **For Windows computers:** activate/use the environment: `opensoundscape\Scripts\activate.bat`
- **For Mac computers:** activate/use the environment `source opensoundscape/bin/activate`
- Install OpenSoundscape in the environment: `pip install opensoundscape==0.4.5`
- Once you are done with OpenSoundscape, deactivate the environment: `deactivate`
- To use the environment again, you will have to refer to absolute path of the virtual environments folder. For instance, if I were on a Mac and created `.venv` inside a directory `/Users/MyFiles/Code` I would activate the virtual environment using: `source /Users/MyFiles/Code/.venv/opensoundscape/bin/activate`

For some of our functions, you will need a version of `ffmpeg >= 0.4.1`. On Mac machines, `ffmpeg` can be installed via `brew`.

2.2 Installation via poetry (contributors and advanced users)

Poetry installation allows direct use of the most recent version of the code. This workflow allows advanced users to use the newest features in OpenSoundscape, and allows developers/contributors to build and test their contributions.

To install via poetry, do the following:

- Download poetry
- Download virtualenvwrapper
- Link `poetry` and `virtualenvwrapper`:
  - Figure out where the virtualenvwrapper.sh file is: `which virtualenvwrapper.sh`
  - Add the following to your `~/.bashrc` and source it. .. code-block:

```
# virtualenvwrapper + poetry
export PATH=~/.local/bin:$PATH
export WORKON_HOME=~/Library/Caches/pypoetry/virtualenvs
source [insert path to virtualenvwrapper.sh, e.g. ~/.local/bin/
↪virtualenvwrapper_lazy.sh]
```

- **Users**: clone this github repository to your machine: `git clone https://github.com/kitzeslab/opensoundscape.git`
- **Contributors**: fork this github repository and clone the fork to your machine
- Ensure you are in the top-level directory of the clone
- Switch to the development branch of OpenSoundscape: `git checkout develop`
- Build the virtual environment for opensoundscape: `poetry install`
  - If poetry install outputs the following error, make sure to download Python 3.7: .. code-block:

```
Installing build dependencies: started
Installing build dependencies: finished with status 'done'
opensoundscape requires Python '>=3.7,<4.0' but the running Python is 3.6.10
```

    If you are using `conda`, install Python 3.7 using `conda install python==3.7`
  - If you are on a Mac and poetry install fails to install `numba`, contact one of the developers for help troubleshooting your issues.

- Activate the virtual environment with the name provided at install e.g.: `workon opensoundscape-dxMTH98s-py3.7` or `poetry shell`
- Check that OpenSoundscape runs: `opensoundscape -h`
- Run tests (from the top-level directory): `poetry run pytest`
- Go back to your system's Python when you are done: `deactivate`

2.2.1 Jupyter

To use OpenSoundscape within JupyterLab, you will have to make an `ipykernel` for the OpenSoundscape virtual environment.

- Activate poetry virtual environment, e.g.: `workon opensoundscape-dxMTH98s-py3.7`
  - Use `poetry env list` if you're not sure what the name of the environment is
- Create ipython kernel: `python -m ipykernel install --user --name=[name of poetry environment] --display-name=OpenSoundscape`
- Now when you make a new document on JupyterLab, you should see a Python kernel available called OpenSoundscape.
- Contributors: if you include Jupyter's `autoreload`, any changes you make to the source code installed via poetry will be reflected whenever you run the `%autoreload` line magic in a cell: .. code-block:

```
%load_ext autoreload
%autoreload
```

2.2.2 Contributing to code

Make contributions by editing the code in your fork. Create branches for features using `git checkout -b feature_branch_name` and push these changes to remote using `git push -u origin feature_branch_name`. To merge a feature branch into the development branch, use the GitHub web interface to create a merge request.

When contributions in your fork are complete, open a pull request using the GitHub web interface. Before opening a PR, do the following to ensure the code is consistent with the rest of the package:

- Run tests: `poetry run pytest`
- Format the code with `black` style (from the top level of the repo): `poetry run black .`
  - To automatically handle this, `poetry run pre-commit install`
- Additional libraries to be installed should be installed with `poetry add`, but in most cases contributors should not add libraries.

2.2.3 Contributing to documentation

Build the documentation using either poetry or sphinx-build

- With poetry: `poetry run build_docs`
- With sphinx-build: `sphinx-build doc doc/_build`

CHAPTER 3

Audio and spectrograms

This tutorial demonstrates how to use OpenSoundscape to open and modify audio files and spectrograms.

Audio files can be loaded into OpenSoundscape and modified using its `Audio` class. The class gives access to modifications such as trimming short clips from longer recordings, splitting a long clip into multiple segments, bandpassing recordings, and extending the length of recordings by looping them. Spectrograms can be created from `Audio` objects using the `Spectrogram` class. This class also allows useful features like measuring the amplitude signal of a recording, trimming a spectrogram in time and frequency, and converting the spectrogram to a saveable image.

To download the tutorial as a Jupyter Notebook, click the "Edit on GitHub" button at the top right of the tutorial. You will have to install OpenSoundscape to use the tutorial.

3.1 Quickstart

First, load the classes from OpenSoundscape.

[1]:
```
# import Audio and Spectrogram classes from OpenSoundscape
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
```

The following code loads an audio file, creates a 224px x 224px -sized spectrogram from it, then creates and saves an image of the spectrogram to the desired path. Each step is discussed in depth below.

[2]:
```
# Settings
original_audio_file = '../tests/1min.wav'
image_shape = (224,224)
spectrogram_save_path = './saved_spectrogram.png'

# Open as Audio
audio = Audio.from_file(original_audio_file)

# Convert into spectrogram
spectrogram = Spectrogram.from_audio(audio)
```

(continues on next page)

(continued from previous page)

```
# Convert into image
image = spectrogram.to_image(shape=image_shape)

# Save image
image.save(spectrogram_save_path)
```

The above function calls can be condensed to a single line:

[3]:
```
Spectrogram.from_audio(Audio.from_file(original_audio_file)).to_image(shape=image_
↪shape).save(spectrogram_save_path)
```

3.2 Audio loading

Load audio files using OpenSoundscape's `Audio` class.

OpenSoundscape uses a package called `librosa` to help load audio files. Librosa automatically supports `.wav` files, but to use `.mp3` files requires that librosa be installed with a package like `ffmpeg`. See Librosa's installation tips for more information.

An example audio file is stored in the `../tests/` directory of OpenSoundscape. Load that file:

[4]:
```
audio_path = '../tests/1min.wav'
audio_object = Audio.from_file(audio_path)
```

3.2.1 Audio properties

The properties of an `Audio` object include its samples (the actual audio data) and the sample rate (the number of audio samples taken per second, required to understand the samples). After an audio file has been loaded, these can be accessed using the `samples` and `sample_rate` properties, respectively.

[5]:
```
audio_object.samples
```

[5]:
```
array([-0.00888062, -0.00344849,  0.00378418, ..., -0.00048828,
        0.00253296,  0.00109863], dtype=float32)
```

[6]:
```
audio_object.sample_rate
```

[6]:
```
32000
```

3.2.2 Loading options

By default, an audio object is loaded with the same sample rate as the source recording. When loading from a file, the sampling rate can be changed or specified. This is useful when working with multiple files and ensuring that all files have a consistent sampling rate. Below, load the same audio file as above, but specify a sampling rate of 22050 Hz.

[7]:
```
audio_object_resample = Audio.from_file(audio_path, sample_rate=22050)
audio_object_resample.sample_rate
```

[7]:
```
22050
```

For other options when loading audio objects, see the `from_file() documentation <api.html#opensoundscape.audio.Audio.from_file>‘__.

3.3 Audio methods

The `Audio` class gives access to a variety of tools to change audio files, load them with special properties, or get information about them. The below examples demonstrate how to bandpass audio recordings, get their duration, extending their length, and trim them. These modifications do not change the original object or the original file itself; instead, they save or return new objects.

Another helpful tool enables the user to trim a series of consecutive clips from a longer audio file. This can be used to split up long files to ready them as inputs to machine learning algorithms. For an example of this, see the data preparation section of the prediction tutorial. For a description of the entire `Audio` object API, see the API documentation.

3.3.1 Bandpassing

Bandpass the audio file to limit its frequency range to 1000 Hz to 5000 Hz.

[8]:
```
bandpassed = audio_object.bandpass(low_f = 1000, high_f = 5000)
```

3.3.2 Duration

Get the current duration of the audio in `audio_object`.

[9]:
```
length = audio_object.duration()
print(length)
```

```
60.0
```

3.3.3 Extending

Using the duration gotten above, extend the recording to twice its original duration. Internally, this function loops the recording until it reaches the desired length.

[10]:
```
extended = audio_object.extend(length * 2)
print(extended.duration())
```

```
120.0
```

3.3.4 Trimming

Trim the extended recording to its original length again, but select the last 60 seconds instead of the first 60 seconds.

[11]:
```
trimmed = extended.trim(start_time = 60.0, end_time = 120.0)
```

The below logic shows that the samples of the original audio object are equal to the samples of the extended-then-trimmed audio object.

[12]:
```
from numpy.testing import assert_array_equal
assert_array_equal(trimmed.samples, audio_object.samples)
```

3.4 Spectrogram creation

3.4.1 Loading spectrograms

A `Spectrogram` object can be created from an audio object using the `from_audio()` method.

[13]:
```
audio_path = '../tests/1min.wav'
audio_object = Audio.from_file(audio_path)
spectrogram_object = Spectrogram.from_audio(audio_object)
```

Spectrograms can also be loaded from saved images using the `from_file()` method.

3.4.2 Spectrogram properties

To check the scale of a spectrogram, you can look at its `times` and `frequencies` properties. The `times` property is the list of times represented by each column of the spectrogram. The `frequencies` property is the list of frequencies represented by each row of the spectrogram. These are not the actual values of the spectrogram – just the scale of the spectrogram itself.

[14]:
```
spec = Spectrogram.from_audio(Audio.from_file('../tests/1min.wav'))
print(f'the first few times: {spec.times[0:5]}')
print(f'the first few frequencies: {spec.frequencies[0:5]}')
```

```
the first few times: [0.008 0.016 0.024 0.032 0.04 ]
the first few frequencies: [  0.   62.5 125.  187.5 250. ]
```

3.4.3 Loading options

Loading a spectrogram from an `Audio` object gives access to several options to customize the calculation of the spectrogram. For instance, use the following steps to create a spectrogram with a higher time-resolution.

First, load an audio file with high sample rate.

[15]:
```
audio = Audio.from_file('../tests/1min.wav', sample_rate=44100)
```

Next, create a spectrogram with 100-sample windows (100/44100 s of audio per window) and no overlap.

[16]:
```
spec = Spectrogram.from_audio(audio, window_samples=100, overlap_samples=0)
```

Note that while this increases the time-resolution of a spectrogram, it reduces the frequency-resolution of the spectrogram.

For other options when loading spectrogram objects from audio objects, see the `` `from_audio() documentation <api.html#opensoundscape.spectrogram.Spectrogram.from_audio>‘__.

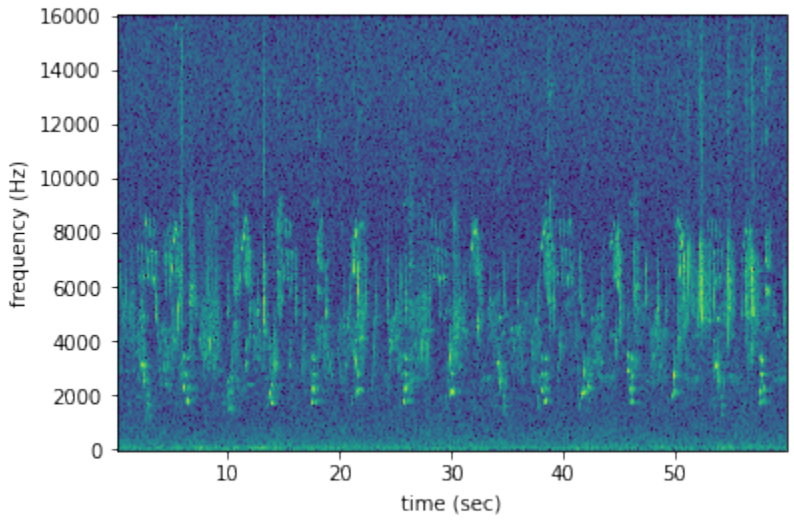
3.5 Spectrogram methods

The tools and features of the spectrogram class are demonstrated here, including plotting; how spectrograms can be generated from modified audio; saving a spectrogram as an image; customizing a spectrogram; trimming and bandpassing a spectrogram; and calculating the amplitude signal from a spectrogram.

3.5.1 Plotting

A `Spectrogram` object can be plotted using its `plot()` method.

[17]:
```
audio_path = '../tests/1min.wav'
audio_object = Audio.from_file(audio_path)
spectrogram_object = Spectrogram.from_audio(audio_object)
spectrogram_object.plot()
```

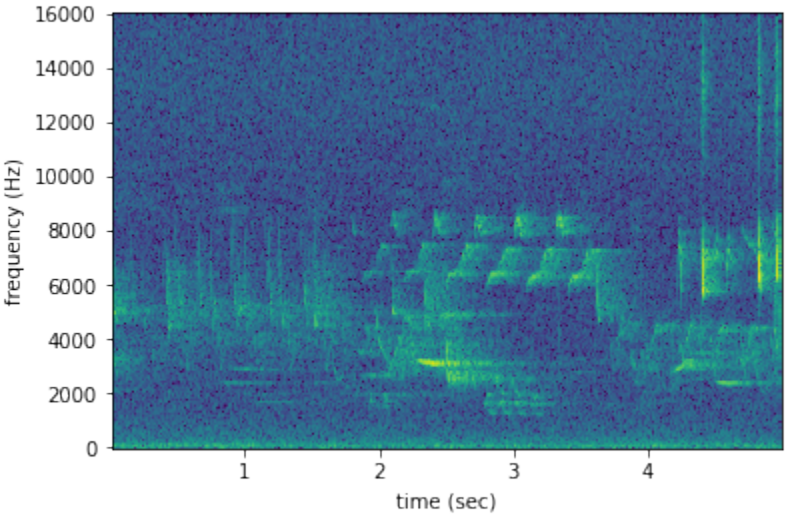


3.5.2 Loading modified audio

The `from_audio` method converts whatever audio is inside the audio object into a spectrogram. So, modified `Audio` objects can be turned into spectrograms as well.

For example, the code below demonstrates creating a spectrogram from a 5 second long trim of the audio object. Compare this plot to the plot above.

[18]:
```
# Trim the original audio
trimmed = audio_object.trim(0, 5)

# Create a spectrogram from the trimmed audio
spec = Spectrogram.from_audio(trimmed)

# Plot the spectrogram
spec.plot()
```



3.5.3 Saving a spectrogram

To save the created spectrogram, first convert it to an image. It will no longer be an OpenSoundscape `Spectrogram` object, but instead a Python Image Library (PIL) `Image` object.

[19]:
```
print("Type of `spectrogram_audio`, before conversion:", type(spectrogram_object))
spectrogram_image = spectrogram_object.to_image()
print("Type of `spectrogram_image`, after conversion:", type(spectrogram_image))
```

```
Type of `spectrogram_audio`, before conversion: <class 'opensoundscape.spectrogram.
↪Spectrogram'>
Type of `spectrogram_image`, after conversion: <class 'PIL.Image.Image'>
```

Save the PIL Image using its `save()` method, supplying the filename at which you want to save the image.

[20]:
```
image_path = './saved_spectrogram.png'
spectrogram_image.save(image_path)
```

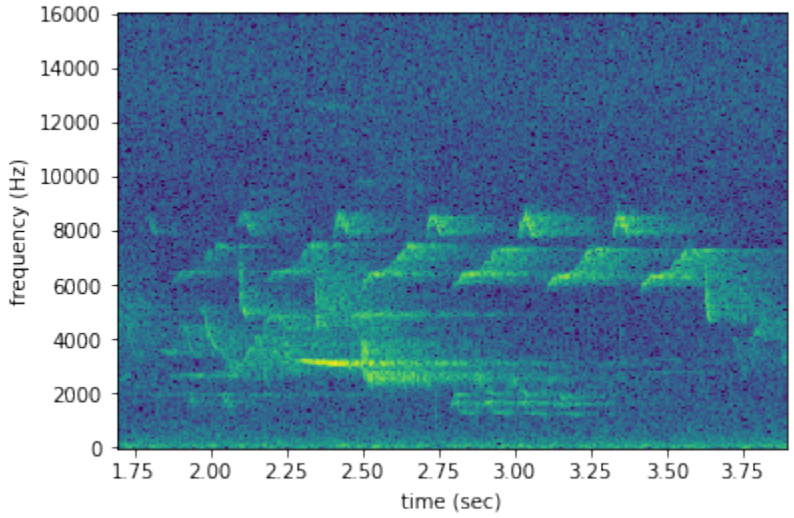
To save the spectrogram at a desired size, specify the image shape when converting the `Spectrogram` to a PIL `Image`.

[21]:
```
image_shape = (512,512)
image_path = './saved_spectrogram_large.png'
spectrogram_image = spectrogram_object.to_image(shape=image_shape)
spectrogram_image.save(image_path)
```

3.5.4 Trimming

Spectrograms can be trimmed in time using `trim()`. Trim the above spectrogram to zoom in on one vocalization.

[22]:
```
spec_trimmed = spec.trim(1.7, 3.9)
spec_trimmed.plot()
```



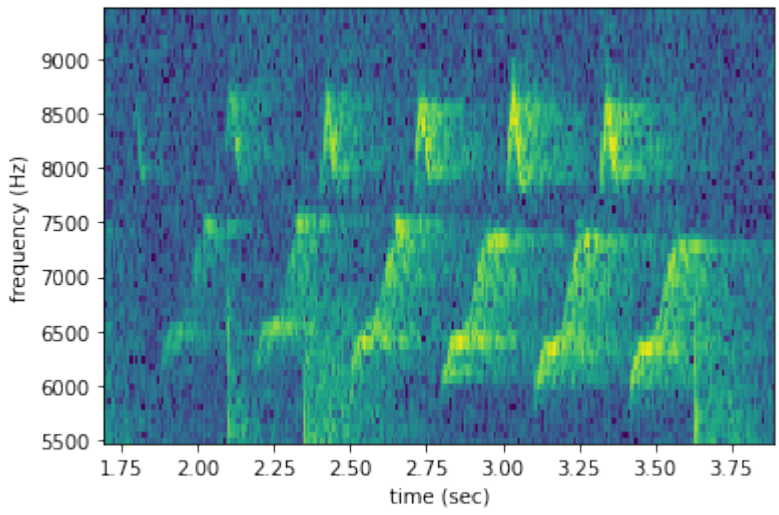
3.5.5 Bandpassing

Spectrograms can be trimmed in frequency using `bandpass()`. For instance, the vocalization zoomed in on above is the song of a Black-and-white Warbler (*Mniotilta varia*), one of the highest-frequency bird songs in our area. Set its approximate frequency range.

[23]:
```
baww_low_freq = 5500
baww_high_freq = 9500
```

Bandpass the above time-trimmed spectrogram in frequency as well to limit the spectrogram view to the vocalization of interest.

[24]:
```
spec_bandpassed = spec_trimmed.bandpass(baww_low_freq, baww_high_freq)
spec_bandpassed.plot()
```



3.5.6 Calculating amplitude signal

OpenSoundscape can calculate the amplitude of an audio file over time using the `Spectrogram` class. First, make a spectrogram from 5 seconds' worth of audio.

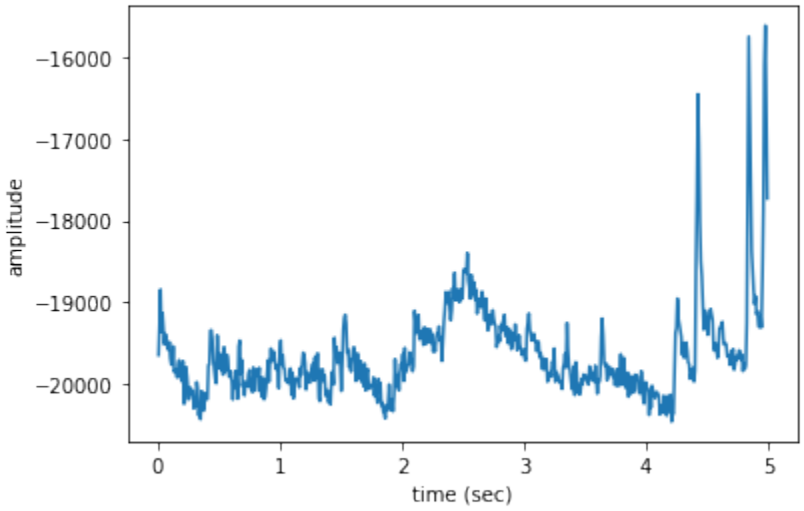
[25]:
```
spec = Spectrogram.from_audio(Audio.from_file('../tests/1min.wav').trim(0,5))
```

Next, use the `amplitude()` method to get the amplitude signal.

[26]:
```
high_freq_amplitude = spec.amplitude()
```

Plot this signal over time to visualize it.

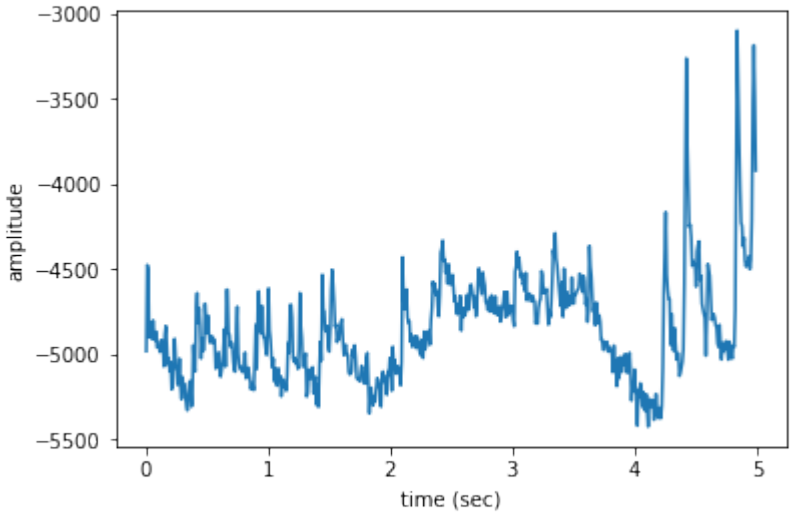
[27]:
```
from matplotlib import pyplot as plt
plt.plot(spec.times,high_freq_amplitude)
plt.xlabel('time (sec)')
plt.ylabel('amplitude')
plt.show()
```



It is also possible to get the amplitude signal from a restricted range of frequencies, e.g., to look at the amplitude in the frequency range of a species of interest.

Look again at the frequency range of the Black-and-white Warbler, discussed above.

[28]:
```
# Get amplitude signal
high_freq_amplitude = spec.amplitude(freq_range=[baww_low_freq, baww_high_freq])

# Plot signal
plt.plot(spec.times, high_freq_amplitude)
plt.xlabel('time (sec)')
plt.ylabel('amplitude')
plt.show()
```



The amplitude in the Black-and-white Warbler frequency range is on average lower in the first two seconds of the recording, gets higher when the warbler sings between 2-4s, and then drops off at 4s. At 4-5s into the recording, there are large spikes in this frequency range from high-frequency noise, the loud "chips" of another animal.

Amplitude signals like these can be used to identify periodic calls, like those by many species of frogs. A pulsing-call identification pipeline called RIBBIT is implemented in OpenSoundscape.

Amplitude signals may not be the most reliable method of identification for species like birds. In this case, it is possible to create a machine learning algorithm to identify calls based on their appearance on spectrograms. For more information, see the algorithm training tutorial. The developers of OpenSoundscape have trained machine learning models for over 500 common North American bird species; for examples of how to download demonstration models, see the prediction tutorial.

CHAPTER 4

Machine learning: training

Biologists are increasingly using acoustic recorders to study species of interest. Many bioacousticians want to determine the identity of the sounds they have recorded; a variety of manual and automated methods exist for this purpose. Automated methods can make it easier and faster to quickly predict which species or sounds are in one's recordings.

Using a process called machine learning, bioacousticians can create (or "train") algorithms that can predict the identities of species vocalizing in acoustic recordings. These algorithms, called classifiers, typically do not identify sounds using the recording alone. Instead, they use image recognition techniques to identify sounds in spectrograms created from short segments of audio.

This tutorial will guide you through the process of training a simple classifier for a single species. To download the tutorial as a Jupyter Notebook and run it on your own computer, click the "Edit on GitHub" button at the top right of the tutorial. You will have to install OpenSoundscape to use the tutorial.

First, use the following packages to create a machine learning classifier. First, from OpenSoundscape import the following three functions (`run_command`, `binary_train_valid_split`, and `train`) and three classes (`Audio`, `Spectrogram`, and `SingleTargetAudioDataset`).

[1]:
```
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
from opensoundscape.datasets import SingleTargetAudioDataset

from opensoundscape.helpers import run_command
from opensoundscape.data_selection import binary_train_valid_split
from opensoundscape.torch.train import train
```

Import the following machine learning-related modules. OpenSoundscape uses PyTorch to do machine learning.

[2]:
```
import torch
import torch.nn
import torch.optim
import torchvision.models
```

Lastly, use a few miscellaneous functions.

[3]:
```
# For interacting with paths on the filesystem
import os.path
from pathlib import Path

# For working with dataframes, arrays, and plotting
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# For handling output of the training function
import io
from contextlib import redirect_stdout
```

4.1 Prepare audio data

4.1.1 Download labeled audio files

Training a machine learning model requires some pre-labeled data. These data, in the form of audio recordings or spectrograms, are labeled with whether or not they contain the sound of the species of interest. These data can be obtained from online databases such as Xeno-Canto.org, or by labeling one's own ARU data using a program like Cornell's "Raven" sound analysis software.

The Kitzes Lab has created a small labeled dataset of short clips of American Woodcock vocalizations. You have two options for obtaining the folder of data, called `woodcock_labeled_data`:

1. Run the following cell to download this small dataset. These commands require you to have `curl` and `tar` installed on your computer, as they will download and unzip a compressed file in `.tar.gz` format.
2. Download a `.zip` version of the files by clicking here. You will have to unzip this folder and place the unzipped folder in the same folder that this notebook is in.

[4]:
```
commands = [
    "curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -
↪o ./woodcock_labeled_data.tar.gz",
    "tar -xzf woodcock_labeled_data.tar.gz", # Unzip the downloaded tar.gz file
    "rm woodcock_labeled_data.tar.gz" # Remove the file after its contents are␣
↪unzipped
]
for command in commands:
    run_command(command)
```

4.1.2 Inspect the data

The folder contains 2s long audio clips taken from an autonomous recording unit. It also contains a file `woodcock_labels.csv` which contains the names of each file and its corresponding label information, created using a program called Specky.

Look at the contents of `woodcock_labels.csv`. First, load them into a `pandas` DataFrame called `labels`. Use `labels.shape` to see how many audio files there are.

[5]:
```
labels = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))
labels.shape
```

[5]:
```
(29, 3)
```

The above call to `labels.shape` showed that there were 29 rows and 3 columns in the loaded dataframe. Look at the `head()` of this dataframe to see the first 5 rows of its contents.

[6]:
```
labels.head()
```

[6]:
```
                                filename woodcock sound_type
0  d4c40b6066b489518f8da83af1ee4984.wav  present       song
1  e84a4b60a4f2d049d73162ee99a7ead8.wav   absent         na
2  79678c979ebb880d5ed6d56f26ba69ff.wav  present       song
3  49890077267b569e142440fa39b3041c.wav  present       song
4  0c453a87185d8c7ce05c5c5ac5d525dc.wav  present       song
```

Before splitting this dataframe into training and validation sets, prepend the name of the folder in front of the filename. This allows our computer program to find these files on the filesystem during the training process.

[7]:
```
labels['filename'] = 'woodcock_labeled_data' + os.path.sep + labels['filename'].
↪astype(str)
labels.head()
```

[7]:
```
                                        filename woodcock sound_type
0  woodcock_labeled_data/d4c40b6066b489518f8da83a...  present       song
1  woodcock_labeled_data/e84a4b60a4f2d049d73162ee...   absent         na
2  woodcock_labeled_data/79678c979ebb880d5ed6d56f...  present       song
3  woodcock_labeled_data/49890077267b569e142440fa...  present       song
4  woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...  present       song
```

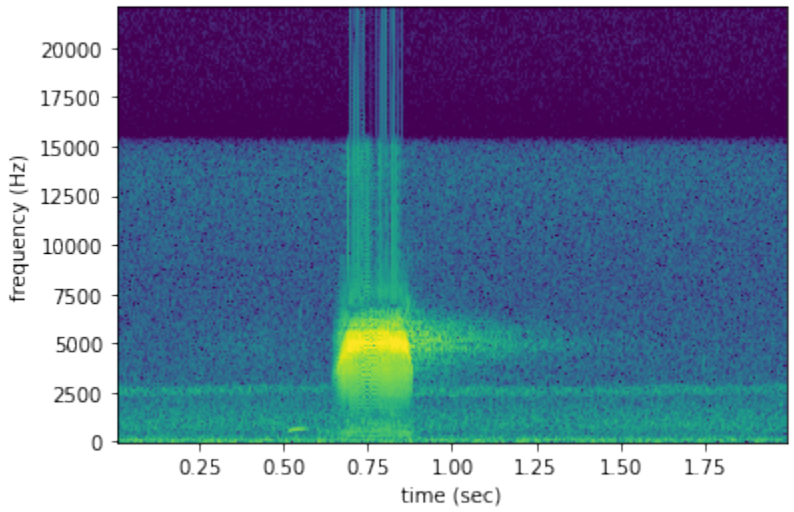
Now, use OpenSoundscape's `Spectrogram` and `Audio` classes to take a look at these files. For more information on the use of these classes, see the tutorial.

The first row in the `labels` dataframe contains a file with the following labels: the American Woodcock is present (`"woodcock" = "present"` and it makes a "song" in the recording (`"sound_type" = "song"`). Get the filename for this recording.

[8]:
```
filename0 = labels.iloc[0]['filename']
```

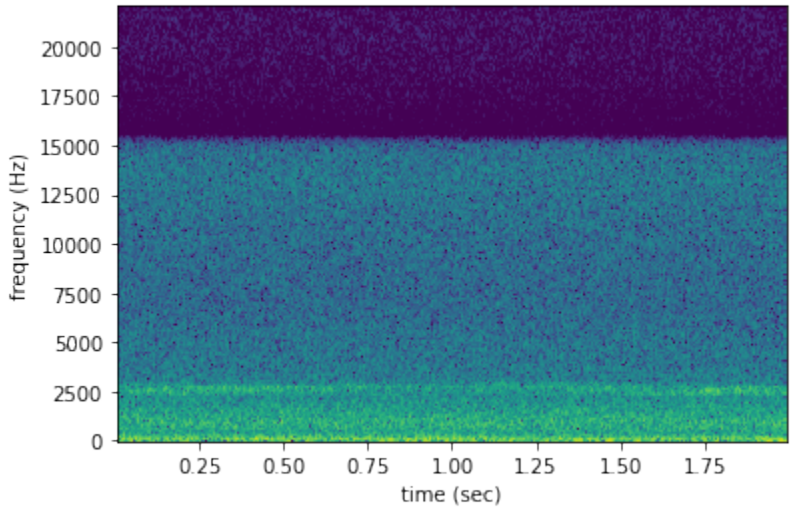
Create a spectrogram from this file. The high-contrast signal of an American Woodcock display sound ("peent") is visible about 0.6 seconds into the recording.

[9]:
```
spect = Spectrogram.from_audio(Audio.from_file(filename0))
spect.plot()
```



The second file, which is marked as not having a woodcock in it (`"woodcock" = "absent"`), has no such signal:

[10]:
```
filename1 = labels.iloc[1]['filename']
spect = Spectrogram.from_audio(Audio.from_file(filename1))
spect.plot()
```



Farther below in the dataset, there are recordings labeled to contain only the "call" of a woodcock. First, list these recordings:

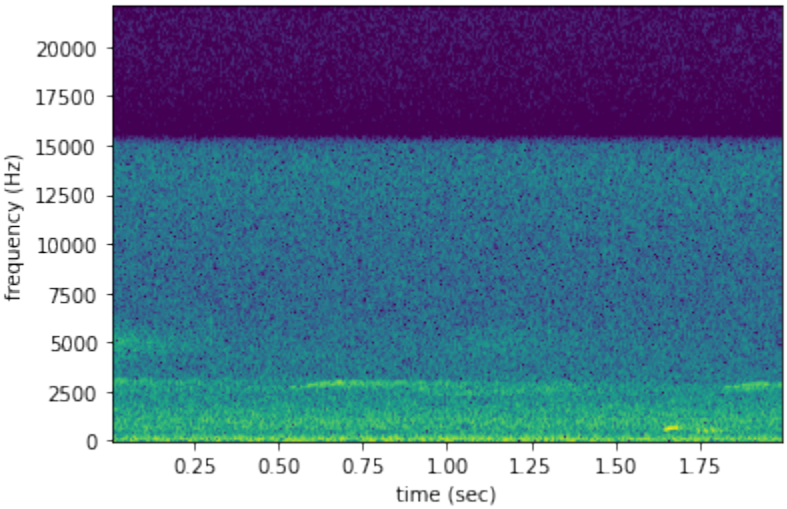
[11]:
```
labels[labels["sound_type"] == "call"]
```

[11]:

	filename	woodcock	sound_type
8	woodcock_labeled_data/f87d427bef752f5accbd8990...	present	call
22	woodcock_labeled_data/c057a4486b25cd638850fc07...	present	call

In reality, the "call" designation means that the woodcock only makes a short, soft, and low introductory sound, instead of the full "peent." Make a spectrogram of one of them to see the difference.

[12]:
```
filename22 = labels.iloc[22]['filename']
spect = Spectrogram.from_audio(Audio.from_file(filename22))
spect.plot()
```



The low sound of the introductory note appears around 1000 kHz at about 1.6 seconds into the recording. Compared to the spectrogram above containing the song, this introductory note is similar to the note that comes before the loud "peent." Although in some applications the user might want to identify this call, it is probably better to mark these as "absences." The following code creates a new column just to identify whether or not the peent is present:

[13]:
```
labels['woodcock_song'] = np.where(labels['sound_type']=='song', 'present', 'absent')
labels
```

[13]:

	filename	woodcock	sound_type	\
0	woodcock_labeled_data/d4c40b6066b489518f8da83a...	present	song	
1	woodcock_labeled_data/e84a4b60a4f2d049d73162ee...	absent	na	
2	woodcock_labeled_data/79678c979ebb880d5ed6d56f...	present	song	
3	woodcock_labeled_data/49890077267b569e142440fa...	present	song	
4	woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...	present	song	
5	woodcock_labeled_data/0fc107ec5e76bf7a98dd207a...	absent	na	
6	woodcock_labeled_data/50b6b7c7e843597e0dbc6986...	present	song	
7	woodcock_labeled_data/35ca80c22127c3c0ae032a08...	absent	na	
8	woodcock_labeled_data/f87d427bef752f5accbd8990...	present	call	
9	woodcock_labeled_data/0ab7732b506105717708ea95...	present	song	
10	woodcock_labeled_data/ad90eefb6196ca83f9cf43b6...	present	song	
11	woodcock_labeled_data/cd0b8d8a89321046e96abee2...	absent	na	
12	woodcock_labeled_data/24073ce519bf1d24107da8a9...	present	song	
13	woodcock_labeled_data/863095c237c52ec51cff7395...	present	song	
14	woodcock_labeled_data/882de25226ed989b31274eea...	present	song	
15	woodcock_labeled_data/6a83b011665c482c1f260d8e...	absent	na	
16	woodcock_labeled_data/45c4b1ed3d7d0acc27125579...	present	song	
17	woodcock_labeled_data/4bb7dbc13db479e8b5769dd9...	present	song	
18	woodcock_labeled_data/75b2f63e032dbd6d19790049...	present	song	
19	woodcock_labeled_data/4afa902e823095e03ba23ebc...	present	song	
20	woodcock_labeled_data/01c5d0c90bd4652f308fd9c7...	present	song	
21	woodcock_labeled_data/92647ab903049a9ee4125abd...	present	song	
22	woodcock_labeled_data/c057a4486b25cd638850fc07...	present	call	
23	woodcock_labeled_data/e9e7153d11de3ac8fc3f7164...	present	song	
24	woodcock_labeled_data/724d8e61b678a6a897b47ed6...	absent	na	
25	woodcock_labeled_data/ad14ac7ffa729060712b442e...	absent	na	
26	woodcock_labeled_data/0d043e9954d9d80ca2c3e860...	present	song	
27	woodcock_labeled_data/78654b6f687d7635f50fba35...	present	song	
28	woodcock_labeled_data/ec0bd96aee95f03b47628b9c...	present	song	

(continues on next page)

(continued from previous page)

```
   woodcock_song
0        present
1         absent
2        present
3        present
4        present
5         absent
6        present
7         absent
8         absent
9        present
10       present
11        absent
12       present
13       present
14       present
15        absent
16       present
17       present
18       present
19       present
20       present
21       present
22        absent
23       present
24        absent
25        absent
26       present
27       present
28       present
```

4.1.3 Create numeric labels

Although the labels are currently "present" and "absent," the library used for machine learning requires numeric labels, not string labels. So, use the following code to transform the "present" and "absent" labels into 0s and 1s. First, define a mapping from the string labels to the numeric labels:

[14]:
```
stringlabel_to_numbericlabel = {"absent":0, "present":1}
```

Next, create a new column of numeric labels:

[15]:
```
labels["numeric_labels"] = labels["woodcock_song"].apply(lambda x: stringlabel_to_
↪numbericlabel[x])
labels.head()
```

[15]:
```
                                            filename woodcock sound_type  \
0  woodcock_labeled_data/d4c40b6066b489518f8da83a...  present       song
1  woodcock_labeled_data/e84a4b60a4f2d049d73162ee...   absent         na
2  woodcock_labeled_data/79678c979ebb880d5ed6d56f...  present       song
3  woodcock_labeled_data/49890077267b569e142440fa...  present       song
4  woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...  present       song

  woodcock_song  numeric_labels
0       present               1
```

(continues on next page)

(continued from previous page)

```
1        absent                0
2       present                1
3       present                1
4       present                1
```

Now drop the unnecessary columns of this dataset, leaving just the `"filename"` and the `"numeric_labels"` columns required to train a machine learning algorithm.

[16]:
```
labels = labels[["filename", "numeric_labels"]]
labels.head()
```

[16]:
```
                                            filename  numeric_labels
0  woodcock_labeled_data/d4c40b6066b489518f8da83a...               1
1  woodcock_labeled_data/e84a4b60a4f2d049d73162ee...               0
2  woodcock_labeled_data/79678c979ebb880d5ed6d56f...               1
3  woodcock_labeled_data/49890077267b569e142440fa...               1
4  woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...               1
```

In order to make it easier for future users to interpret the model results, save a dictionary that associates each numeric label with an explanatory string variable. In this case, mark the 0-labeled recordings `"scolopax-minor-absent"` and the 1-labeled recordings `"scolopax-minor-present"`. That way, as long as the model is bundled with this metadata, it will be easy to see that the `1` prediction corresponds to American Woodcock (scientific name *Scolopax minor*).

[17]:
```
label_dict = {0:'scolopax-minor-absent', 1:'scolopax-minor-present'}
```

4.2 Create machine learning datasets

4.2.1 Training-validation split

Next, to use machine learning on these files, they must be separated into two datasets. The "training" dataset will be used to teach the machine learning algorithm. The "validation" dataset will be used to evaluate the algorithm's performance each epoch. The process of separating the data into multiple datasets is often known in machine learning as creating a "split."

Typically, machine learning practitioners use a separate validation set to check on the model's performance during and after training. While the training data are used to teach the model how to identify its focal species, the validation data are not used to teach the model. Instead, they are held out as a separate comparison. This allows us to check how well the model generalizes to data it has never seen before. A model that performs well on the training set, but very poorly on the validation set, is said to be *overfit*. Overfit models are great at identifying the original recordings they saw, but are often not useful for real applications.

First, look at the dataframe again.

[18]:
```
labels.head()
```

[18]:
```
                                            filename  numeric_labels
0  woodcock_labeled_data/d4c40b6066b489518f8da83a...               1
1  woodcock_labeled_data/e84a4b60a4f2d049d73162ee...               0
2  woodcock_labeled_data/79678c979ebb880d5ed6d56f...               1
3  woodcock_labeled_data/49890077267b569e142440fa...               1
4  woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...               1
```

It's often desirable to make a *stratified split*. This means that the percentage of samples in the original dataset that have each label, will be roughly equal to the percentage of each label in the training and validation datasets. So, for

instance, if half of the recordings in the original dataframe had the species present, in a stratified split, half of the recordings in the training dataframe and in the validation dataframe would have the species present.

Use a scikit-learn function to do this, specifying the `"numeric_labels"` column as the one to stratify over.

[19]:
```
train_df, valid_df = train_test_split(labels, train_size=0.8, stratify=labels[
↪'numeric_labels'])
```

Check that the dataframes are stratified correctly. Compare the fraction of positives in the original dataset with the fraction of positives in the training and validation subsets.

[20]:
```
num_samples = labels.shape[0]
num_present = sum(labels['numeric_labels'] == 1)
print(f"Fraction of original dataframe with woodcock present: {num_present/num_
↪samples:.2f}")
```

```
Fraction of original dataframe with woodcock present: 0.69
```

[21]:
```
num_train_samples = train_df.shape[0]
num_train_present = sum(train_df['numeric_labels'] == 1)
print(f"Fraction of train samples with woodcock present: {num_train_present/num_train_
↪samples:.2f}")
```

```
Fraction of train samples with woodcock present: 0.70
```

[22]:
```
num_valid_samples = valid_df.shape[0]
num_valid_present = sum(valid_df['numeric_labels'] == 1)
print(f"Fraction of train samples with woodcock present: {num_valid_present/num_valid_
↪samples:.2f}")
```

```
Fraction of train samples with woodcock present: 0.67
```

So, the fraction is very close, though not exact–owing to the difference in size of these two datasets. This is not unexpected.

4.2.2 Format as `SingleTargetAudioDatasets`

Turn these dataframes into "Datasets" using the `SingleTargetAudioDataset` class. Once they are set up in this class, they can be used by the training algorithm. Data augmentation could be applied in this step, but is not demonstrated here; for more information, see the relevant API documentation.

To use this class, specify the names of the relevant columns in the dataframes:

[23]:
```
train_dataset = SingleTargetAudioDataset(
    df=train_df, label_dict=label_dict, label_column='numeric_labels', filename_
↪column='filename')
valid_dataset = SingleTargetAudioDataset(
    df=valid_df, label_dict=label_dict, label_column='numeric_labels', filename_
↪column='filename')
```

4.3 Train the machine learning model

Next, set up the architecture of the machine learning model and train it.

4.3.1 Set up model architecture

The model architecture is a neural network. Neural networks are so-named for their loose similarity to neurons. Each **neuron** takes in a small amount of data, performs a transformation to the data, and passes it on with some weight to the next neuron. Neurons are usually organized in **layers**; each neuron in one layer can be connected to one or multiple neurons in the next layer. Complex structures can arise from this series of connections.

The neural network used here is a combination of a feature extractor and a classifier. The **feature extractor** is a convolutional neural network (CNN). CNNs are a special class of neural network commonly used for image classification. They are able to interpret pixels that are near each other to identify shapes or textures in images, like lines, dots, and edges. During the training process, the CNN learns which shapes and textures are important for distinguishing between different classes.

The specific CNN used here is `resnet18`, using the `pretrained=True` option. This means that the model loaded is a version that somebody has already trained on another image dataset called ImageNet, so it has a head start on understanding features commonly seen in images. Although spectrograms aren't the same type of images as the photographs used in ImageNet, using the pretrained model will allow the model to more quickly adapt to identifying spectrograms.

[24]:
```
model = torchvision.models.resnet18(pretrained = True)
```

Although we refer to the whole neural network as a classifier, the part of the neural network that actually does the species classification is its `fc`, or "fully connected," layers. This part of the neural network is called "fully connected" because it consists of several layers of neurons, where every neuron in each layer is connected to every other neuron in its adjacent layers.

These layers come after the CNN layers, which have already interpreted an image's features. The `fc` layers then use those interpretations to classify the image. The number of output features of the CNN, therefore, is the number of input features of the `fc` layers:

[25]:
```
model.fc.in_features
```

[25]:
```
512
```

Use a `Linear` classifier for the `fc`. To set up the `Linear` classifier, identify the input and output size for this classifier. As described above, the `fc` takes in the outputs of the feature extractor, so `in_features = model.fc.in_features`. The model identifies one species, so it has to be able to output a "present" or "absent" classification. Thus, `out_features=2`. A multi-species model would use `out_features=number_of_species`.

[26]:
```
model.fc = torch.nn.Linear(in_features = model.fc.in_features, out_features = 2)
```

4.3.2 Train the model

Next, create set up a directory in which to save results.

[27]:
```
results_path = Path('model_train_results')
if not results_path.exists(): results_path.mkdir()
```

The scikit-learn function may throw errors when calculating metrics; the following code will silence them.

[28]:
```
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

Finally, run the model training with the following parameters: * `save_dir`: the directory in which to save results (which is created if it doesn't exist) * `model`: the model set up in the previous cell * `train_dataset`: the training dataset created using `SingleTargetAudioDataset` * `optimizer`: the optimizer to use for training the algorithm * `loss_fn`: the loss function used to assess the algorithm's performance during training * `epochs`: the number of times the model will run through the training data * `log_every`: how frequently to save performance data and save intermediate machine learning weights (`log_every=1` will save every epoch)

The `train` function allows the user to control more parameters, but they are not demonstrated here. For more information, see the train API.

[29]:
```
train_outputs = io.StringIO()
with redirect_stdout(train_outputs):
    train(
        save_dir = results_path,
        model = model,
        train_dataset = train_dataset,
        valid_dataset = valid_dataset,
        optimizer = torch.optim.SGD(model.parameters(), lr=1e-3),
        loss_fn = torch.nn.CrossEntropyLoss(),
        epochs=10,
        log_every=1,
        print_logging=True,
    )
```

4.4 Evaluate model performance

When training is complete, it is important to check the training results to see how well the model identifies sounds. This model was only trained on a limited amount of data, so the model is expected to not be usable–it is for demonstration purposes only.

The outputs of the training function were saved to `train_outputs`. Check out the first 100 characters of this output.

[30]:
```
source_text = train_outputs.getvalue()
print(source_text[:100])
```

```
Epoch 0
  Training.
  Validating.
  Validation results:
    train_loss: 0.7144076221663019
    train
```

These functions help to parse the log text. They simply extract the resulting "metric" in each epoch. Metrics include accuracy, precision, recall, and f1 score.

[31]:
```
def extract_all_lines_containing(source_text, str_to_extract):
    """Case-sensitive search for lines containing str_to_extract"""
    finished = False
    lines = source_text.split('\n')
    extract_lines = [line for line in lines if str_to_extract in line]
    return extract_lines

def strip_log(log, sep='     '):
    return log.split(sep)[1]
```

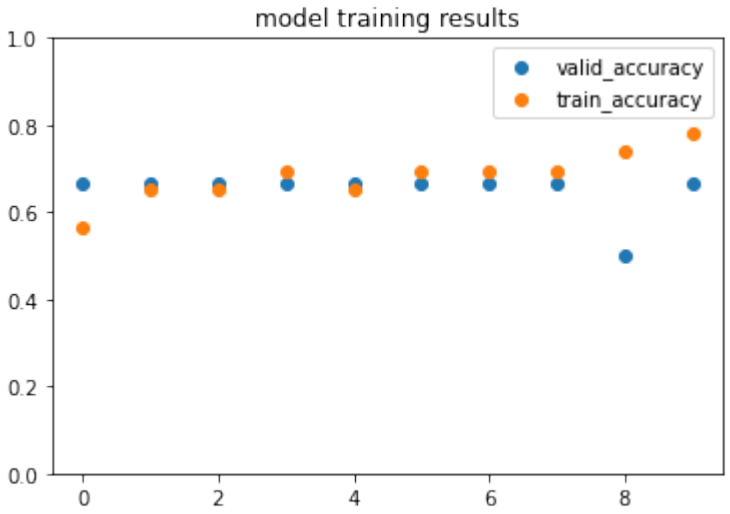
(continues on next page)

(continued from previous page)

```
def get_metric_from_log(source_text, metric):
    if 'precision' in metric or 'recall' in metric:
        return [float(strip_log(line, sep=': ').strip('[]').split()[1]) for line in␣
↪extract_all_lines_containing(source_text, metric)]
    return [float(strip_log(line, sep=': ')) for line in extract_all_lines_
↪containing(source_text, metric)]
```

Plot the validation accuracy each epoch. These results will look different every time the model is trained, as it is a stochastic process.

[32]:
```
metrics_to_plot = ['valid_accuracy', 'train_accuracy']
fig, ax = plt.subplots(1, 1)
for metric in metrics_to_plot:
    results = get_metric_from_log(source_text, metric)
    ax.scatter(range(len(results)), results)
ax.set_ylim(0, 1)
ax.set_title('model training results')
ax.legend(metrics_to_plot)
plt.show()
```



Lastly, this command "cleans up" by deleting all the downloaded files and results. Only run this if you are ready to remove the results of this analysis.

[33]:
```
import shutil
# Delete downloads
shutil.rmtree(Path("woodcock_labeled_data"))
# Delete results
shutil.rmtree(results_path)
```

CHAPTER 5

Machine learning: prediction

Machine learning-trained algorithms can predict whether bioacoustic recordings contain a sound of interest. For instance, an algorithm trained how to detect the sound of a Wood Thrush can be used to predict where Wood Thrushes vocalize in a set of autonomous recordings.

The Kitzes Lab, the developers of OpenSoundscape, pre-trained a series of baseline machine learning models that can be used to predict the presence of 485 species of common North American birds. These are "beta" models; if you are interested in using them for research, please contact us at the Kitzes Lab. Information about the training process is available at this README.

This tutorial downloads an example model and demonstrates how to use it to predict the identity of birds in recordings. To download the tutorial as a Jupyter Notebook, click the "Edit on GitHub" button at the top right of the tutorial. To run the Jupyter Notebook tutorial, follow these instructions to install OpenSoundscape and add the OpenSoundscape environment to your Jupyter kernels.

5.1 Import modules

Import the following modules to run a pre-trained machine learning learning classifier. First, from OpenSoundscape we will need two classes (`Audio` and `SingleTargetAudioDataset`) and three functions (`run_command`, `lowercase_annotations`, and `predict`).

[1]:
```
from opensoundscape.audio import Audio
from opensoundscape.datasets import SingleTargetAudioDataset
from opensoundscape.helpers import run_command
from opensoundscape.raven import lowercase_annotations
from opensoundscape.torch.predict import predict
```

Import the following machine learning-related modules. OpenSoundscape uses PyTorch to do machine learning.

[2]:
```
import torch
import torch.nn
import torchvision.models
```

(continues on next page)

(continued from previous page)

```
import torch.utils.data
import torchvision.transforms
```

Lastly, use a few miscellaneous functions.

[3]:
```
import yaml
import os.path
import pandas as pd
from pathlib import Path
from math import floor
import matplotlib.pyplot as plt
```

5.2 Download model

To use the model, it must be downloaded onto your computer and loaded with the same specifications it was created with.

Download the example model for Wood Thrush, *Hylocichla mustelina*. First, create a folder called `"prediction_example"` to store the model and its data in.

[4]:
```
folder_name = "prediction_example"
folder_path = Path(folder_name)
if not folder_path.exists(): folder_path.mkdir()
```

Next, download the model from the Box storage site using the following lines.

If you prefer, you can also download the model directly off of the shared folder (see introduction paragraphs). Make sure to move it into the `"prediction_example"` folder and ensure that it is named `"hylocichla-mustelina.tar"`. These instructions can be modified for any of the species included in the pre-trained set of models.

[5]:
```
def download_from_box(link, name):
    run_command(f"curl -L {link} -o ./{name}")
```

This link format enables direct download.

[6]:
```
link_to_model = "https://pitt.box.com/shared/static/0xl7aqjlhdrx83am7k4w0e72fsg08dey.
↪tar"
```

Now, use the function created above to download the model file.

[7]:
```
model_filename = folder_path.joinpath("hylocichla-mustelina.tar")
download_from_box(
    link=link_to_model,
    name=model_filename,
)
```

Make sure that the model was downloaded correctly.

[8]:
```
assert(model_filename.exists())
```

5.3 Load model

At its core, a machine learning model consists of two things: its architecture and its weights.

5.3.1 Create architecture

The architecture is the complex structure of the model, which in this case, is a convolutional neural network. Convolutional neural networks are a particular set of algorithms especially suited to extracting and interpreting features from images, such as combinations of lines, dots, and edges. In this case, we use a `resnet18` convolutional neural network. After feature extraction, the convolutional neural network's features are passed to a classifier. The classifier decides how to weight each feature in predicting the final class identity. The model was trained with a `Linear` classifier.

Create the architecture of the model. First, designate the model as a `resnet18` CNN.

[9]:
```
model = torchvision.models.resnet18(pretrained=False)
```

Then, add the `fc` layers. "FC" stands for "fully connected". To set up the proper architecture, we need to specify the correct number of input features, output features, and classifier type.

The number of input features to the FC is equal to the number of features extracted from the convolutional neural network and passed to the the FC layer: `model.fc.in_features`

[10]:
```
num_cnn_features = model.fc.in_features
```

The models were trained to predict two classes (species present and species absent), so the number of output features of the FC layer is 2.

[11]:
```
num_classes = 2
```

Finally, the classifier type is a `torch.nn.Linear` classifier.

[12]:
```
model.fc = torch.nn.Linear(
    in_features = num_cnn_features,
    out_features = num_classes)
```

5.3.2 Load weights and metadata

The weights of the model are distinguished from its architecture because, while the architecture is decided by humans, the weights of the architecture are learned during the machine learning process.

When downloading the machine learning model, you downloaded a compressed file that contains the weights and some metadata about the model. First, inspect what you downloaded using `torch.load` to extract the compressed `.tar` model file.

[13]:
```
model_and_metadata = torch.load(model_filename)
```

Inspect metadata

The variable `model_and_metadata` is a dictionary. The keys of the dictionary that we can use to access the model information are:

[14]:
```
model_and_metadata.keys()
```

[14]:
```
dict_keys(['train_loss', 'train_accuracy', 'train_precision', 'train_recall', 'train_
↪f1', 'train_confusion_matrix', 'valid_accuracy', 'valid_precision', 'valid_recall',
↪'valid_f1', 'valid_confusion_matrix', 'model_state_dict', 'optimizer_state_dict',
↪'labels_yaml', 'train_scores', 'train_targets', 'valid_scores', 'valid_targets'])
```

Some of the metadata included in the model is information about the model's performance during training. A full description of what each of these keys means is given in the download folder (see introduction).

For instance, the machine learning model is trained using a set of recordings where the species is known to be present, and a set where the species is known to be absent. These files are divided into two sets: a "training" set, which the model directly learns from, and a "validation" set, which the model does not learn from but we use to check the model's performance as it trains.

The model outputs a score for each file. We want the model's scores for the species-present files to be lower than those for the species-absent files. We can inspect the dictionary's `valid_targets` and `valid_scores` attributes, which respectively give a 1 or a 0 based on whether a training file included the species or did not; and a real number score for that file.

First, extract the validation score:

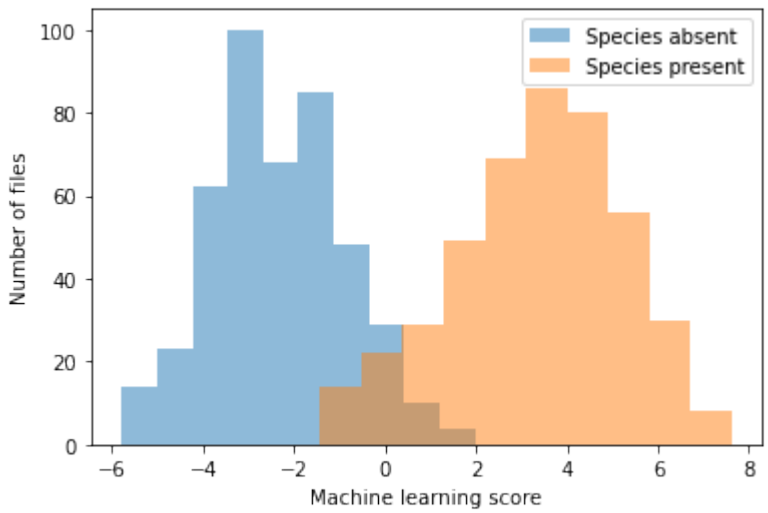
[15]:
```
validation_scores = pd.DataFrame(model_and_metadata['valid_scores'])
validation_true_targets = pd.Series(model_and_metadata['valid_targets'])
```

Then, separate the scores for the species-present files and the species-absent files:

[16]:
```
true_absent_scores = validation_scores[validation_true_targets == 0][1]
true_present_scores = validation_scores[validation_true_targets == 1][1]
```

Finally, plot a histogram of the scores for the two file types:

[17]:
```
plt.hist(true_absent_scores, alpha=0.5, label='Species absent')
plt.hist(true_present_scores, alpha=0.5, label='Species present')
plt.legend()
plt.xlabel('Machine learning score')
plt.ylabel('Number of files')
plt.show()
```



This model performs fairly well at differentiating the validation files, which are segments of Xeno-Canto recordings.

Note: this doesn't mean the model will perform similarly well on ARU recordings!

Load weights onto architecture

To use the model itself, access the dictionary's `'model_state_dict'` attribute:

[18]:
```
weights = model_and_metadata['model_state_dict']
```

Now, apply these weights to the model architecture created above.

[19]:
```
model.load_state_dict(weights)
```

[19]:
```
<All keys matched successfully>
```

5.4 Prepare prediction files

To actually use the model, we need to download and prepare a set of recordings. The model was trained to make predictions on spectrograms made from 5 second-long recordings, so we will have to split the recordings up and transform them into spectrograms.

As example data, we have provided a 1 minute-long soundscape which contains Wood Thrush vocalizations.

The following code downloads this audio file into the `"prediction_example"` folder created above. If you prefer, you can also download this file at this link. Make sure to move it into the `"prediction_example"` folder and ensure that it is named `"1min.wav"`.

[20]:
```
data_filename = folder_path.joinpath("1minexamplefile.wav")
download_from_box(
    link = "https://pitt.box.com/shared/static/z73eked7quh1t2pp93axzrrpq6wwydx0.wav",
    name = data_filename
)
```

The example soundscape must be split up into soundscapes of the same size as the ones the model was trained on. In this case, the soundscapes should be 5s long.

First, create a directory in which to save split files.

[21]:
```
split_directory = folder_path.joinpath("split_files")
if not split_directory.exists(): split_directory.mkdir()
```

Next, load the 1-minute long file as an `Audio` object.

[22]:
```
base_file = Audio.from_file(data_filename)
```

To split `base_file` into 5s long segments, use the `split` method of `opensoundscape.Audio`. This will return a list of dictionaries, containing a 5s long `Audio` object, and its start time and end time with respect to `base_file`. The argument `final_clip=None` only makes a difference if the source audio didn't have a length divisible by 5s, by removing the remainder clip so that we are only left with 5s long clips. For more information on the behavior of this argument, see the API Documentation.

[23]:
```
split_files = base_file.split(
    clip_duration=5,
    clip_overlap=0,
    final_clip=None,
)
```

See what the first element of the resulting array looks like:

[24]:
```
split_files[0]
```

[24]:
```
{'clip': <Audio(samples=(160000,), sample_rate=32000)>,
 'clip_duration': 5.0,
 'begin_time': 0,
 'end_time': 5}
```

Save each of these split files in the directory created earlier. Save each of them with a prefix and the start and end times of the split with respect to the original file. In this case, make the prefix equal to the filename of the base file, without the extension (its stem):

[25]:
```
split_prefix = data_filename.stem
split_prefix
```

[25]:
```
'1minexamplefile'
```

The following loop saves each of the clips in the array.

[26]:
```
filenames = []
for split_file_dict in split_files:
    # Extract the Audio object
    clip_audio = split_file_dict['clip']

    # The start and end time of this clip with respect to the original recording
    begin_time = split_file_dict['begin_time']
    end_time = split_file_dict['end_time']

    # Create a filename that starts with the filename of the original recording
    filename = split_directory.joinpath(f'{split_prefix}_{begin_time}s-{end_time}s.wav
↪')

    # Keep track of the filenames in an array
    filenames.append(filename)

    # Save the filenames
    clip_audio.save(path=filename)
```

5.5 Create a Dataset

Now that the data are split, we can create a "dataset" from them using OpenSoundscape's `SingleTargetAudioDataset`. This structure takes in a `DataFrame` of filenames. It can be accessed like a list of the same length as the DataFrame of filenames. When it is accessed, it takes the filename, loads the audio at the filename, and transforms that audio into a spectrogram in the correct format to use for our machine learning models.

This dataset, `SingleTargetAudioDataset`, is intended for models that predict the presence of a single target, e.g., models that predict whether a single species is present in a file, like the model we are using. It can be used in both training and prediction, and has many options for implementing image augmentation during training (see the API Documentation). Just use the default options for prediction.

To create a dataset, first format the list of 5s clip filenames into a pandas `DataFrame`. Name the column containing the filenames `'file_path'`.

[27]:
```
filename_column = 'file_path'
files_to_predict_on = pd.DataFrame(filenames, columns=[filename_column])
```

Additionally, the `SingleTargetAudioDataset` requires that we use a dictionary that associates numeric labels with the class names: 1 is for predicting a Wood Thrush's presence; 0 is for predicting a Wood Thrush's absence. This dictionary is packaged with the model under the key `'labels_yaml'`:

[28]:
```
label_dict = yaml.safe_load(model_and_metadata['labels_yaml'])
label_dict
```

[28]:
```
{0: 'hylocichla-mustelina-absent', 1: 'hylocichla-mustelina-present'}
```

Now create the `SingleTargetAudioDataset`.

[29]:
```
test_dataset = SingleTargetAudioDataset(
    df=files_to_predict_on,
    filename_column=filename_column,
    label_dict=label_dict,
)
```

The `test_dataset` is a list of dictionaries. Each element of the list contains a dictionary for one of the files to predict on.

[30]:
```
len(test_dataset)
```

[30]:
```
12
```

Each dictionary in `test_dataset` has one or two keys. In all cases, the dictionary has a key `'X'` which refers to the spectrogram. If a dataset is created with true labels, the dictionary also has a `'y'` key which links to the true label. Because it is unknown which of these files contain Wood Thrush songs, no true labels were given when creating the dataset.

The spectrogram itself is stored as a PyTorch tensor. For example, here is the tensor of the first spectrogram:

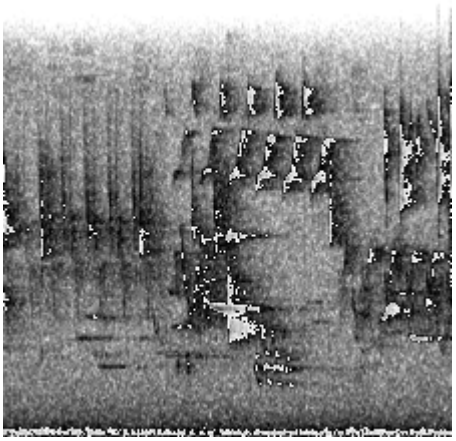
[31]:
```
first_tensor = test_dataset[0]['X']
first_tensor
```

[31]:
```
tensor([[[ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         ...,
         [-0.2314, -0.1451,  0.1373,  ..., -0.0902, -0.1373, -0.0902],
         [-0.1529, -0.2157,  0.1922,  ..., -0.1451, -0.1686, -0.0275],
         [ 0.2784,  0.0275,  0.3647,  ...,  0.0510,  0.1059,  0.3176]],

        [[ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         ...,
         [-0.2314, -0.1451,  0.1373,  ..., -0.0902, -0.1373, -0.0902],
         [-0.1529, -0.2157,  0.1922,  ..., -0.1451, -0.1686, -0.0275],
         [ 0.2784,  0.0275,  0.3647,  ...,  0.0510,  0.1059,  0.3176]],

        [[ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         ...,
         [-0.2314, -0.1451,  0.1373,  ..., -0.0902, -0.1373, -0.0902],
         [-0.1529, -0.2157,  0.1922,  ..., -0.1451, -0.1686, -0.0275],
         [ 0.2784,  0.0275,  0.3647,  ...,  0.0510,  0.1059,  0.3176]]])
```

To view this spectrogram, use PyTorch's `transforms.ToPILImage()` function. This function returns a transformer. Call the transformer on the first tensor to display the spectrogram as an image.

[32]:
```
transformer = torchvision.transforms.ToPILImage()
transformer(first_tensor)
```

[32]:



5.6 Use model on prediction files

Finally, the model can be used for prediction. Use OpenSoundscape's `predict` function to call the model on the test dataset. The `label_dict` created above is used to make the classes interpretable; otherwise, the classes would just be numbered.

[33]:
```
prediction_df = predict(model, test_dataset, label_dict=label_dict)
prediction_df
```

[33]:

	hylocichla-mustelina-absent	hylocichla-mustelina-present
prediction_example/split_files/1minexamplefile_...	-2.699836	2.345755
prediction_example/split_files/1minexamplefile_...	-3.972310	4.026467
prediction_example/split_files/1minexamplefile_...	-5.235929	5.631192
prediction_example/split_files/1minexamplefile_...	-2.984634	2.917254
prediction_example/split_files/1minexamplefile_...	-1.895094	2.359997
prediction_example/split_files/1minexamplefile_...	-3.669389	4.548790
prediction_example/split_files/1minexamplefile_...	-3.804515	4.256460
prediction_example/split_files/1minexamplefile_...	-4.488552	4.726258
prediction_example/split_files/1minexamplefile_...	-3.719126	3.582299
prediction_example/split_files/1minexamplefile_...	-4.844445	4.841421
prediction_example/split_files/1minexamplefile_...	-3.637188	3.066844
prediction_example/split_files/1minexamplefile_...	-3.639840	3.178823

Interpreting these scores is the challenging part of machine learning. One typical method is to empirically determine a score threshold below which the species is considered absent and present, and listen to a sample of recordings above and below the threshold to determine the false positive and false negative rate of the threshold.

Note that the classifier usually performs worse on autonomous recording unit data than it does on the validation set. In particular, the distributions of the species-present and species-absent values may have greater variance and may be centered on different values, closer to each other than in the validation set (see histograms above).

Finally, this command "cleans up" by deleting all the downloaded files and results. Only run this if you are ready to remove the results of this analysis.

[34]:
```
import shutil
shutil.rmtree(folder_path)
```

CHAPTER 6

RIBBIT Pulse Rate model demonstration

RIBBIT (Repeat-Interval Based Bioacoustic Identification Tool) is a tool for detecting vocalizations that have a repeating structure.

This tool is useful for detecting vocalizations of frogs, toads, and other animals that produce vocalizations with a periodic structure. In this notebook, we demonstrate how to select model parameters for the Great Plains Toad, then run the model on data to detect vocalizations.

RIBBIT was introduced in the 2020 poster "Automatic Detection of Pulsed Vocalizations"

This notebook demonstrates how to use the RIBBIT tool implemented in opensoundscape as `opensoundscape.ribbit.ribbit()`

For help instaling OpenSoundscape, see the documentation

6.1 import packages

[1]:
```
# suppress warnings
import warnings
warnings.simplefilter('ignore')

#import packages
import numpy as np
from glob import glob
import pandas as pd
from matplotlib import pyplot as plt

#local imports from opensoundscape
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
from opensoundscape.ribbit import ribbit

# create big visuals
plt.rcParams['figure.figsize']=[15,8]
```

6.2 download example audio

first, let's download some example audio to work with.

You can run the cell below, **OR** visit this link to downlaod the data (whichever you find easier):

https://pitt.box.com/shared/static/0xclmulc4gy0obewtzbzyfnsczwgr9we.zip

If you download using the link above, first un-zip the folder (double-click on mac or right-click -> extract all on Windows). Then, move the `great_plains_toad_dataset` folder to the same location on your computer as this notebook. Then you can skip this cell:

[2]:
```
from opensoundscape.helpers import run_command
#download files from box.com to the current directory
_ = run_command(f"curl -L https://pitt.box.com/shared/static/
↪9mrxib85y1jmf1ybbjvbr0tv171iekvy.gz -o ./great_plains_toad_dataset.tar.gz")# | tar -
↪xz -f")
_ = run_command(f"tar -xz -f great_plains_toad_dataset.tar.gz")

#this will print `0` if everything went correctly. If it prints 256 or another number,
↪ something is wrong (try downloading from the link above)
```

now, you should have a folder in the same location as this notebook called `great_plains_toad_dataset`

if you had trouble accessing the data, you can try using your own audio files - just put them in a folder called `great_plains_toad_dataset` in the same location as this notebook, and this notebook will load whatever is in that folder

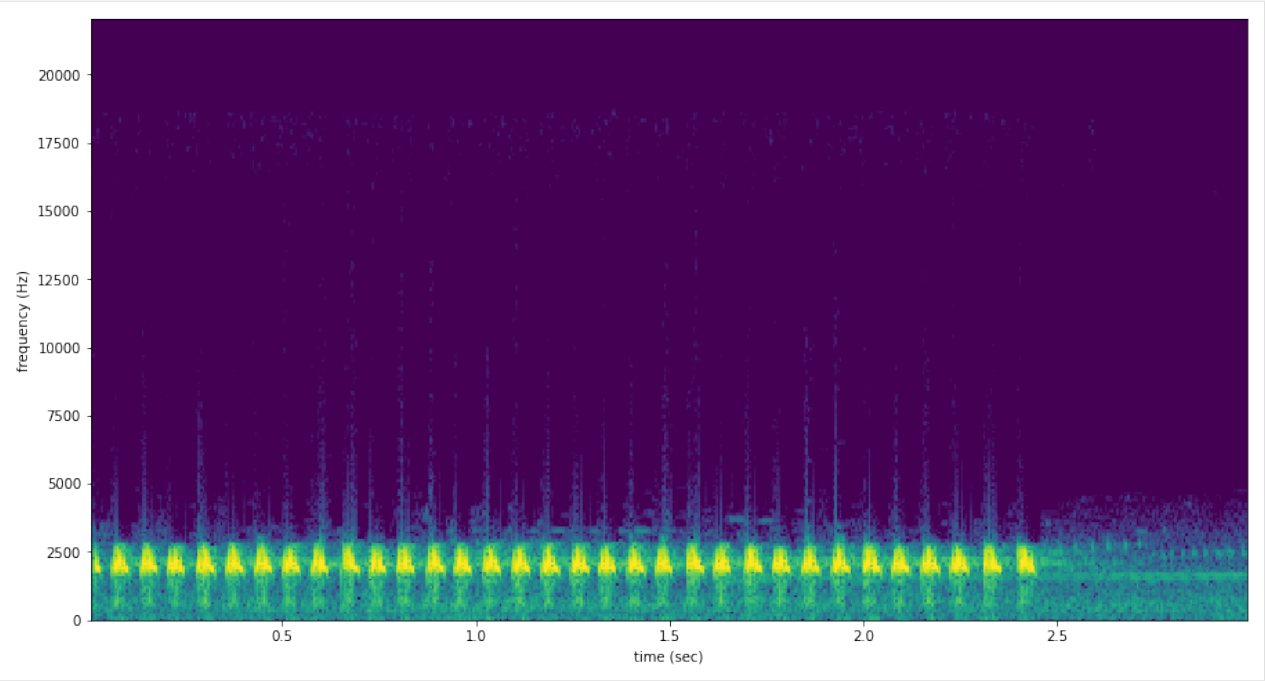
6.2.1 load an audio file and create a spectrogram

[3]:
```
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

#load the audio file into an OpenSoundscape Audio object
audio = Audio.from_file(audio_path)

#trim the audio to the time from 0-3 seconds for a closer look
audio = audio.trim(0,3)

#create a Spectrogram object
spectrogram = Spectrogram.from_audio(audio)
```

6.2.2 show the Great Plains Toad spectrogram as an image

a spectrogram is a visual representation of audio with frequency on the vertical axis, time on the horizontal axis, and intensity represented by the color of the pixels

[4]:
```
spectrogram.plot()
```



6.3 select model parameters

RIBBIT requires the user to select a set of parameters that describe the target vocalization. Here is some detailed advice on how to use these parameters.

Signal Band: The signal band is the frequency range where RIBBIT looks for the target species. Based on the spectrogram above, we can see that the Great Plains Toad vocalization has the strongest energy around 2000-2500 Hz, so we will specify `signal_band = [2000,2500]`. It is best to pick a narrow signal band if possible, so that the model focuses on a specific part of the spectrogram and has less potential to include erronious sounds.

Noise Bands: Optionally, users can specify other frequency ranges called noise bands. Sounds in the `noise_bands` are *subtracted* from the `signal_band`. Noise bands help the model filter out erronious sounds from the recordings, which could include confusion species, background noise, and popping/clicking of the microphone due to rain, wind, or digital errors. It's usually good to include one noise band for very low frequencies – this specifically eliminates popping and clicking from being registered as a vocalization. It's also good to specify noise bands that target confusion species. Another approach is to specify two narrow `noise_bands` that are directly above and below the `signal_band`.

Pulse Rate Range: This parameters specifies the minimum and maximum pulse rate (the number of pulses per second, also known as pulse repetition rate) RIBBIT should look for to find the focal species. Looking at the spectrogram above, we can see that the pulse rate of this Great Plains Toad vocalization is about 15 pulses per second. By looking at other vocalizations in different environmental conditions, we notice that the pulse rate can be as slow as 10 pulses per second or as fast as 20. So, we choose `pulse_rate_range = [10, 20]` meaning that RIBBIT should look for pulses no slower than 10 pulses per second and no faster than 20 pulses per second.

Window Length: This parameter tells the algorithm how many seconds of audio to analyze at one time. Generally, you should choose a `window_length` that is similar to the length of the target species vocalization, or a little bit longer. For very slowly pulsing vocalizations, choose a longer window so that at least 5 pulses can occur in one window (0.5 pulses per second -> 10 second window). Typical values for `window_length` are 1 to 10 seconds. Keep in mind that The Great Plains Toad has a vocalization that continues on for many seconds (or minutes!) so we chose a 2-second window which will include plenty of pulses.

Plot: We can choose to show the power spectrum of pulse repetition rate for each window by setting `plot=True`. The default is not to show these plots (`plot=False`).

[5]:
```
# minimum and maximum rate of pulsing (pulses per second) to search for
pulse_rate_range = [10,20]

# look for a vocalization in the range of 1000-2000 Hz
signal_band = [2000,2500]

# subtract the amplitude signal from these frequency ranges
noise_bands = [ [0,200], [10000,10100]]

#divides the signal into segments this many seconds long, analyzes each independently
window_length = 2 #(seconds)

#if True, it will show the power spectrum plot for each audio segment
show_plots = True
```

6.4 search for pulsing vocalizations with `ribbit()`

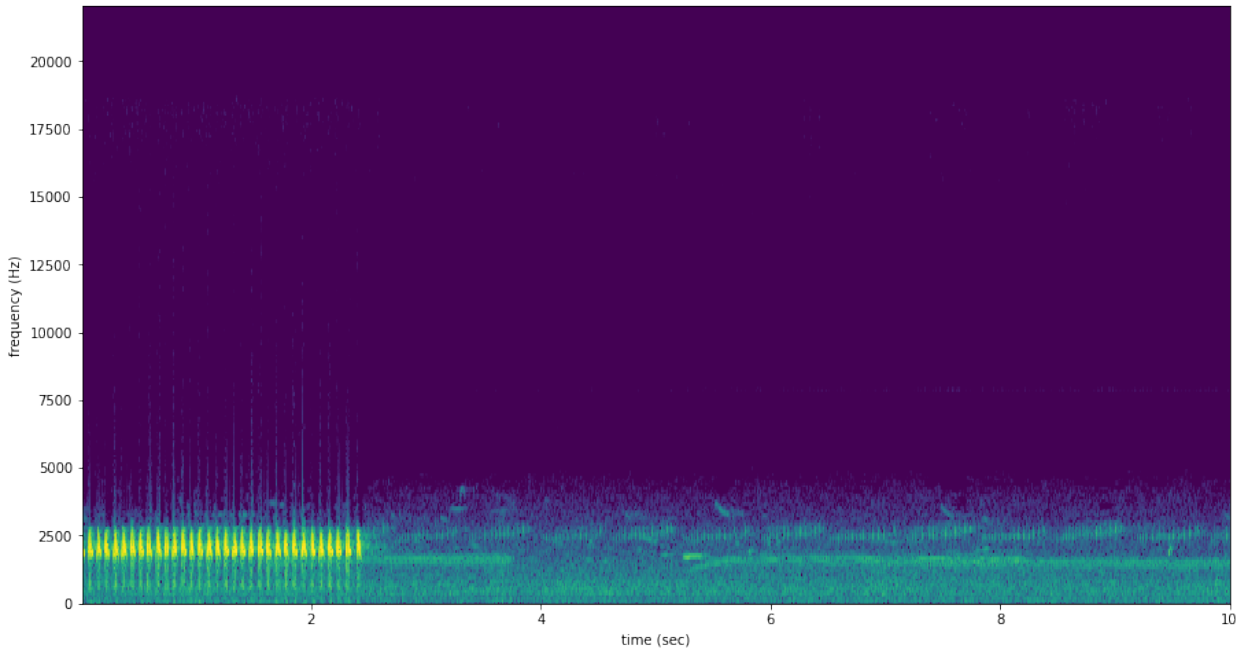
This function takes the parameters we chose above as arguments, performs the analysis, and returns two arrays: - **scores:** the pulse rate score for each window - **times:** the start time in seconds of each window

The scores output by the function may be very low or very high. They do not represent a "confidence" or "probability" from 0 to 1. Instead, the relative values of scores on a set of files should be considered: when RIBBIT detects the target species, the scores will be significantly higher than when the species is not detected.

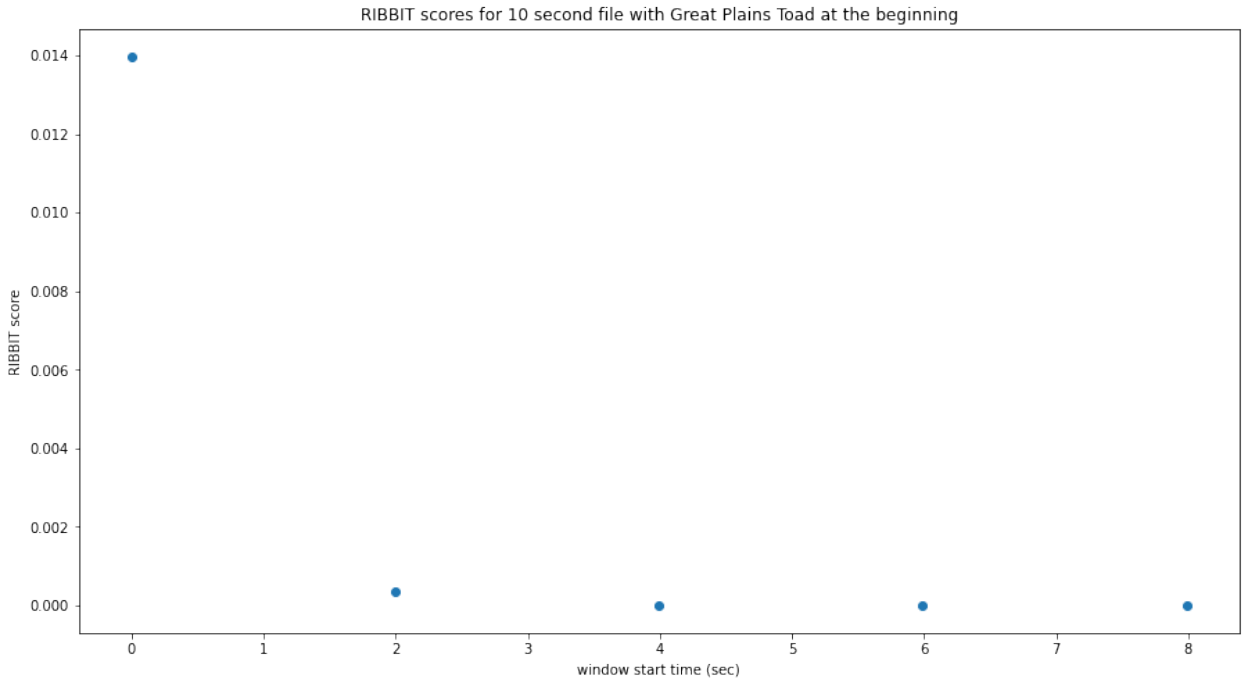
The file `gpt0.wav` has a Great Plains Toad vocalizing only at the beginning. Let's analyze the file with RIBBIT and look at the scores versus time.

[8]:
```
#get the audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

#make the spectrogram
spec = Spectrogram.from_audio(audio.from_file(audio_path))

#run RIBBIT
scores, times =  ribbit(
                    spec,
                    pulse_rate_range=pulse_rate_range,
                    signal_band=signal_band,
                    window_len=window_length,
                    noise_bands=noise_bands,
                    plot=False)


#show the spectrogram
print('spectrogram of 10 second file with Great Plains Toad at the beginning')
spec.plot()

# plot the score vs time of each window
plt.scatter(times,scores)
plt.xlabel('window start time (sec)')
plt.ylabel('RIBBIT score')
plt.title('RIBBIT scores for 10 second file with Great Plains Toad at the beginning')
```

```
spectrogram of 10 second file with Great Plains Toad at the beginning
```




```
[8]: Text(0.5, 1.0, 'RIBBIT scores for 10 second file with Great Plains Toad at the␣
↪beginning')
```



as we hoped, RIBBIT outputs a high score during the vocalization (the window from 0-2 seconds) and a low score when the frog is not vocalizing

6.5 analyzing a set of files

[12]:
```
# set up a dataframe for storing files' scores and labels
df = pd.DataFrame(index = glob('./great_plains_toad_dataset/*'),columns=['score',
↪'label'])

# label is 1 if the file contains a Great Plains Toad vocalization, and 0 if it does␣
↪not
df['label'] = [1 if 'gpt' in f else 0 for f in df.index]

# calculate RIBBIT scores
for path in df.index:

    #make the spectrogram
    spec = Spectrogram.from_audio(audio.from_file(path))

    #run RIBBIT
    scores, times =  ribbit(
                        spec,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        window_len=window_length,
                        noise_bands=noise_bands,
                        plot=False)

    # use the maximum RIBBIT score from any window as the score for this file
    # multiply the score by 10,000 to make it easier to read
    df.at[path,'score'] = max(scores) * 10000

print("Files sorted by score, from highest to lowest:")
df.sort_values(by='score',ascending=False)
```

```
Files sorted by score, from highest to lowest:
```

[12]:

	score	label
./great_plains_toad_dataset/gpt0.mp3	139.765	1
./great_plains_toad_dataset/gpt3.mp3	13.8338	1
./great_plains_toad_dataset/gpt2.mp3	8.25766	1
./great_plains_toad_dataset/gpt1.mp3	6.1136	1
./great_plains_toad_dataset/negative3.mp3	2.34044	0
./great_plains_toad_dataset/negative2.mp3	1.73015	0
./great_plains_toad_dataset/negative4.mp3	1.56953	0
./great_plains_toad_dataset/negative1.mp3	1.21802	0
./great_plains_toad_dataset/negative9.mp3	1.13301	0
./great_plains_toad_dataset/negative8.mp3	1.08165	0
./great_plains_toad_dataset/negative6.mp3	0.966176	0
./great_plains_toad_dataset/negative5.mp3	0.695368	0
./great_plains_toad_dataset/gpt4.mp3	0.634423	1
./great_plains_toad_dataset/pops2.mp3	0.51215	0
./great_plains_toad_dataset/water.mp3	0.510283	0
./great_plains_toad_dataset/pops1.mp3	0.493911	0
./great_plains_toad_dataset/negative7.mp3	0.0156971	0
./great_plains_toad_dataset/silent.mp3	0	0

So, how good is RIBBIT at finding the Great Plains Toad?

We can see that the scores for all of the files with Great Plains Toad (gpt) score above 6 except `gpt4.mp3` (which contains only a very quiet and distant vocalization). All files that do not contain the Great Plains Toad score less than 2.5. So, RIBBIT is doing a good job separating Great Plains Toads vocalizations from other sounds!

Notably, noisy files like `pops1.mp3` score low even though they have lots of periodic energy - our `noise_bands` sucessfully rejected these files. Without using `noise_bands`, files like these would receive very high scores. Also, some birds in "negatives" files that have periodic calls around the same pulsre rate as the Great Plains Toad received low scores. This is also a result of choosing a tight `signal_band` and strategic `noise_bands`. You can try adjusting or eliminating these bands to see their effect on the audio.

(HINT: elimintating the `noise_bands` will result in high scores for the "pops" files)

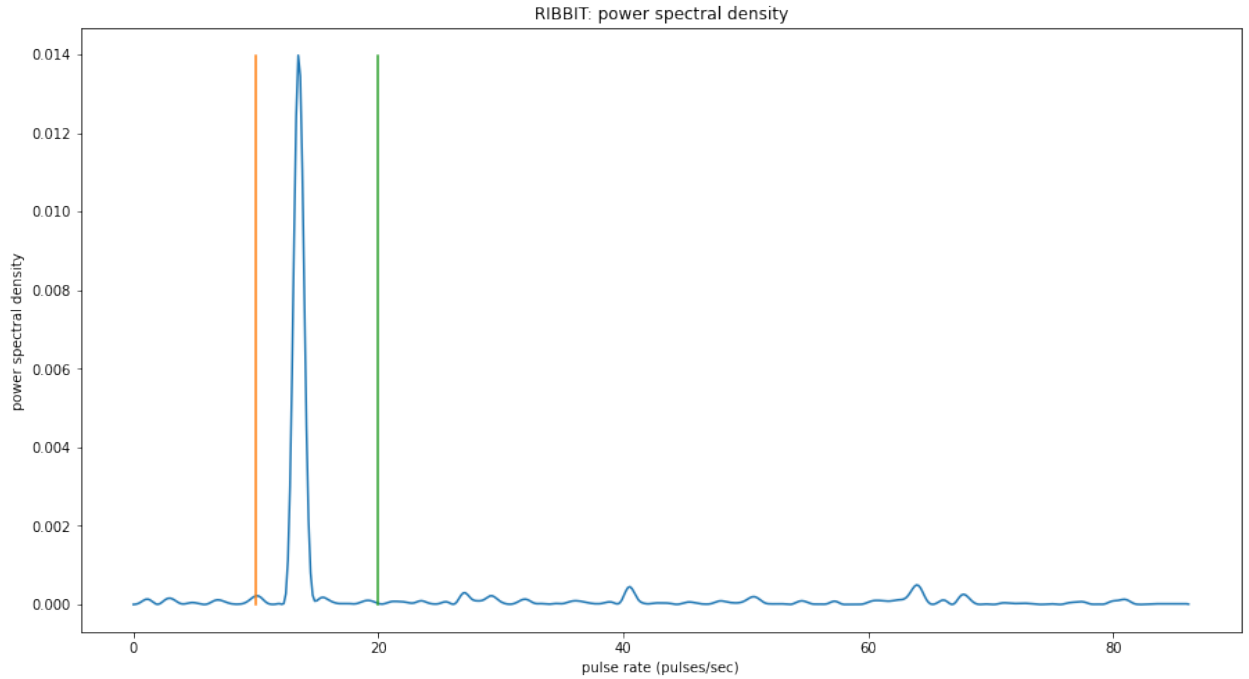
6.6 detail view

Now, lets look at one 10 second file and tell ribbit to plot the power spectral density for each window (`plot=True`). This way, we can see if peaks are emerging at the expected pulse rates. Since our `window_length` is 2 seconds, each of these plots represents 2 seconds of audio. The vertical lines on the power spectral density represent the lower and upper `pulse_rate_range` limits.

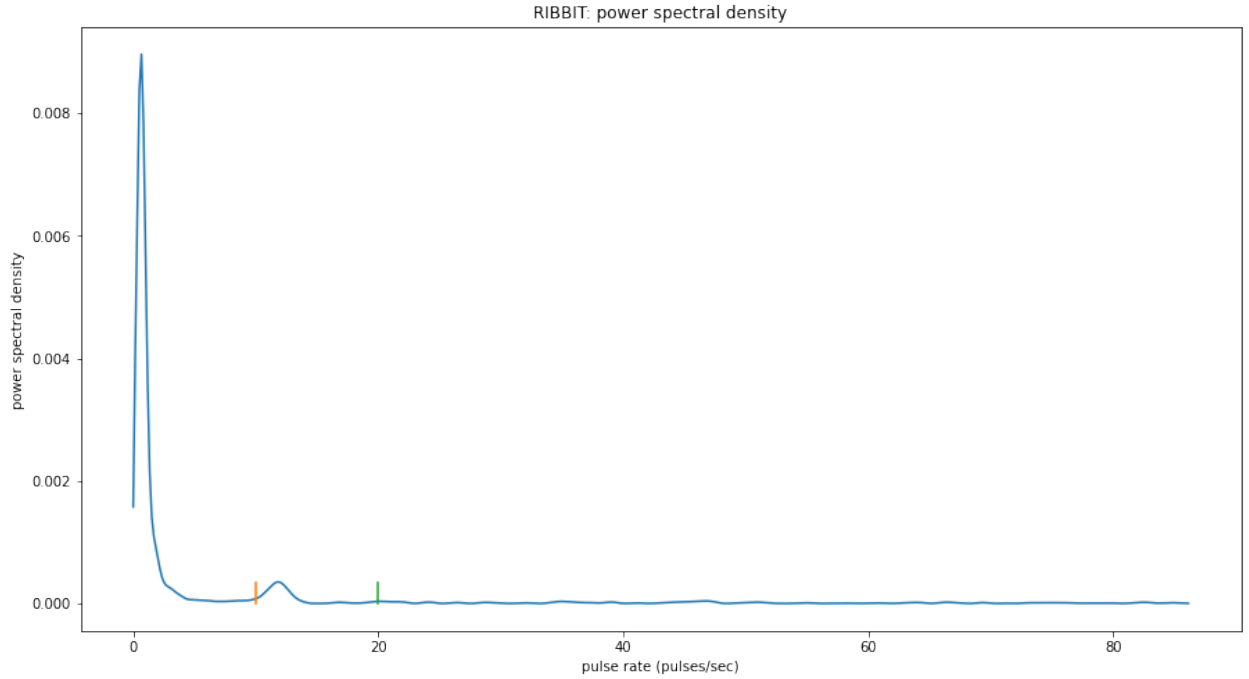
In the file `gpt0.mp3`, the Great Plains Toad vocalizes for a couple seconds at the beginning, then stops. We expect to see a peak in the power spectral density at 15 pulses/sec in the first 2 second window, and maybe a bit in the second, but not later in the audio.

[45]:
```
#create a spectrogram from the file, like above:
# 1. get audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]
# 2. make audio object and trim (this time 0-10 seconds)
audio = Audio.from_file(audio_path).trim(0,10)
# 3. make spectrogram
spectrogram = Spectrogram.from_audio(audio)

scores, times =  ribbit(
                        spectrogram,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        window_len=window_length,
                        noise_bands=noise_bands,
                        plot=show_plots)
```

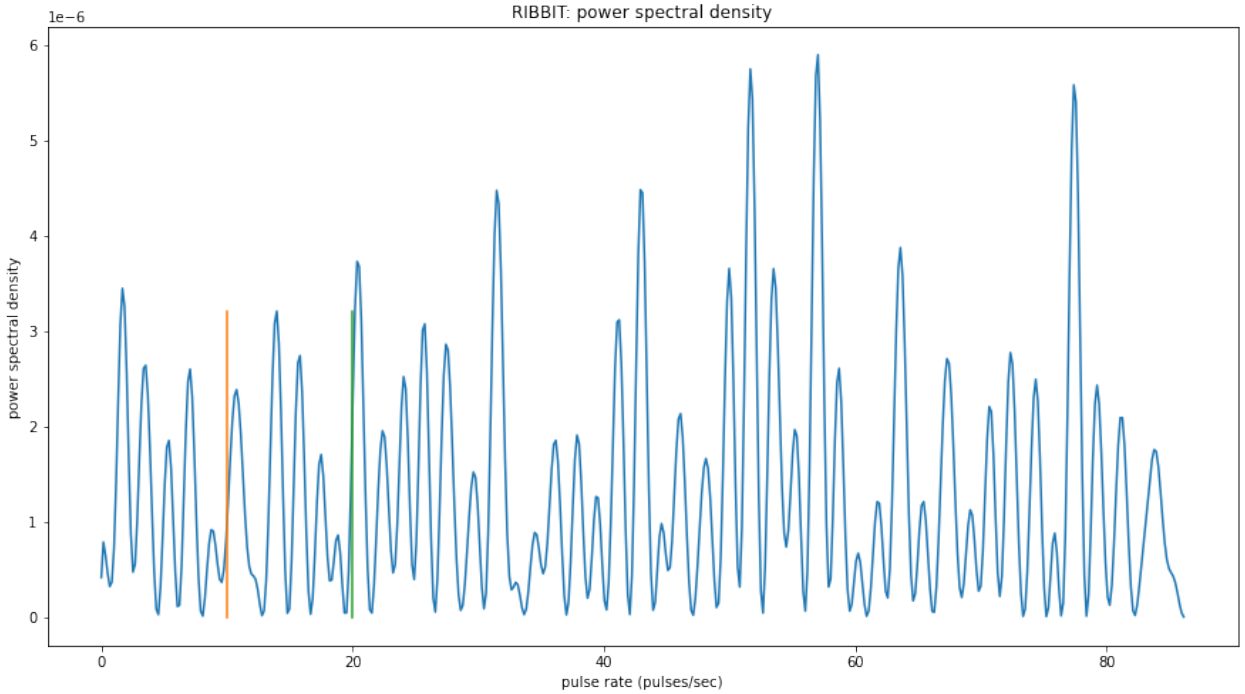
```
window: 0.0000 sec to 1.9969 sec
peak freq: 13.4583
```



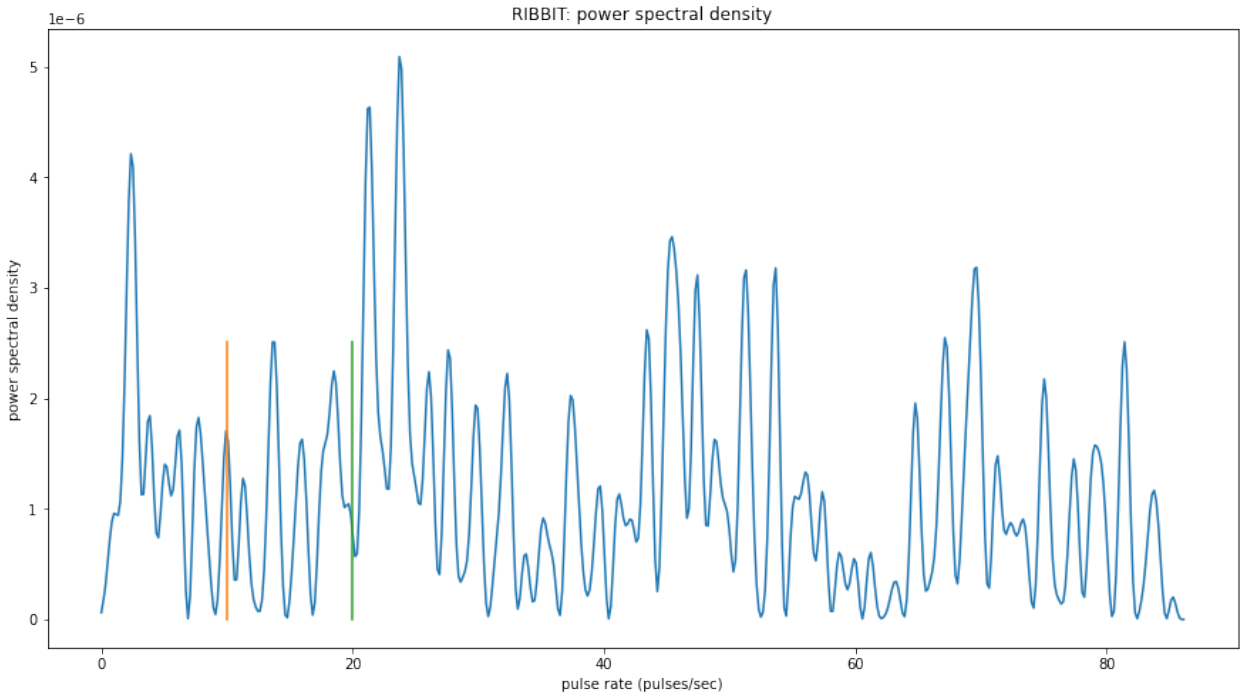

```
window: 1.9969 sec to 3.9938 sec
peak freq: 0.6729
```




```
window: 3.9938 sec to 5.9907 sec
peak freq: 57.0293
```




```
window: 5.9907 sec to 7.9877 sec
peak freq: 23.7202
```



6.7 Time to experiment for yourself

Now that you know the basics of how to use RIBBIT, you can try using it on your own data. We recommend spending some time looking at different recordings of your focal species before choosing parameters. Experiment with the noise bands and window length, and get in touch if you have questions!

Sam's email: sam . lapp [at] pitt.edu

this cell will delete the folder `great_plains_toad_dataset`. Only run it if you wish delete that folder and the example audio inside it.

[3]:
```
_ = run_command('rm -r ./great_plains_toad_dataset/')
_ = run_command('rm ./great_plains_toad_dataset.tar.gz')
```

[ ]:
```

```

CHAPTER 7

API Documentation

7.1 Audio

audio.py: Utilities for dealing with audio files

class opensoundscape.audio.**Audio**(*samples*, *sample_rate*, *resample_type='kaiser_fast'*, *max_duration=None*)

Container for audio samples

Initializing an *Audio* object directly requires the specification of the sample rate. Use *Audio.from_file* or *Audio.from_bytesio* with *sample_rate=None* to use a native sampling rate.

Parameters

- **samples** (*np.array*) – The audio samples
- **sample_rate** (*integer*) – The sampling rate for the audio samples
- **resample_type** (*str*) – The resampling method to use [default: "kaiser_fast"]
- **max_duration** (*None or integer*) – The maximum duration allowed for the audio file [default: None]

Returns An initialized *Audio* object

bandpass(*low_f*, *high_f*, *order*)

bandpass audio signal frequencies

uses a phase-preserving algorithm (scipy.signal's butter and solfiltfilt)

Parameters

- **low_f** – low frequency cutoff (-3 dB) in Hz of bandpass filter
- **high_f** – high frequency cutoff (-3 dB) in Hz of bandpass filter
- **order** – butterworth filter order (integer) ~= steepness of cutoff

duration()

Return duration of Audio

Output: duration (float): The duration of the Audio

extend(*length*)
Extend audio file by looping it

Parameters **length** – the final length in seconds of the extended file

Returns a new Audio object of the desired length

classmethod from_bytesio(*bytesio*, *sample_rate=None*, *max_duration=None*, *resample_type='kaiser_fast'*)
Read from bytesio object

Read an Audio object from a BytesIO object. This is primarily used for passing Audio over HTTP.

Parameters

- **bytesio** – Contents of WAV file as BytesIO
- **sample_rate** – The final sampling rate of Audio object [default: None]
- **max_duration** – The maximum duration of the audio file [default: None]
- **resample_type** – The librosa method to do resampling [default: "kaiser_fast"]

Returns An initialized Audio object

classmethod from_file(*path*, *sample_rate=None*, *resample_type='kaiser_fast'*, *max_duration=None*)
Load audio from files

Deal with the various possible input types to load an audio file and generate a spectrogram

Parameters

- **path** (*str, Path*) – path to an audio file
- **sample_rate** (*int, None*) – resample audio with value and resample_type, if None use source sample_rate (default: None)
- **resample_type** – method used to resample_type (default: kaiser_fast)
- **max_duration** – the maximum length of an input file, None is no maximum (default: None)

Returns attributes samples and sample_rate

Return type *Audio*

save(*path*)
save Audio to file

Parameters **path** – destination for output

spectrum()
create frequency spectrum from an Audio object using fft

Parameters **self** –

Returns fft, frequencies

split(*clip_duration=5*, *clip_overlap=1*, *final_clip=None*)
Split Audio into clips

The Audio object is split into clips of a specified duration and overlap

Parameters

- **clip_duration** – The duration in seconds of the clips

- **clip_overlap** – The overlap of the clips in seconds
- **final_clip** – Possible options (any other input will ignore the final clip entirely), - "remainder": Include the remainder of the Audio

  (clip will not have clip_duration length)

  - "full": Increase the overlap to yield a clip with clip_duration
  - "extend": Similar to remainder but extend the clip to clip_duration

Results: A list of dictionaries with keys: ["audio", "begin_time", "end_time"]

time_to_sample(*time*)
Given a time, convert it to the corresponding sample

Parameters **time** – The time to multiply with the sample_rate

Returns The rounded sample

Return type sample

trim(*start_time*, *end_time*)
trim Audio object in time

Parameters

- **start_time** – time in seconds for start of extracted clip
- **end_time** – time in seconds for end of extracted clip

Returns a new Audio object containing samples from start_time to end_time

exception opensoundscape.audio.**OpsoLoadAudioInputError**
Custom exception indicating we can't load input

exception opensoundscape.audio.**OpsoLoadAudioInputTooLong**
Custom exception indicating length of audio is too long

opensoundscape.audio.**split_and_save**(*audio*, *destination*, *prefix*, *clip_duration=5*, *clip_overlap=1*, *final_clip=None*, *dry_run=False*)
Split audio into clips and save them to a folder

Parameters

- **audio** – The input Audio to split
- **destination** – A folder to write clips to
- **prefix** – A name to prepend to the written clips
- **clip_duration** – The duration of each clip in seconds [default: 5]
- **clip_overlap** – The overlap of each clip in seconds [default: 1]
- **final_clip** – Possible options (any other input will ignore the final clip entirely) [default: None] - "remainder": Include the remainder of the Audio

  (clip will not have clip_duration length)

  - "full": Increase the overlap to yield a clip with clip_duration
  - "extend": Similar to remainder but extend the clip to clip_duration

- **dry_run** – If True, skip writing audio and just return clip DataFrame [default: False]

Returns pandas.DataFrame containing begin and end times for each clip from the source audio

7.2 Audio Tools

audio_tools.py: set of tools that filter or modify audio files or sample arrays (not Audio objects)

opensoundscape.audio_tools.**bandpass_filter**(*signal*, *low_f*, *high_f*, *sample_rate*, *order=9*)
perform a butterworth bandpass filter on a discrete time signal using scipy.signal's butter and solfiltfilt (phase-preserving version of sosfilt)

Parameters

- **signal** – discrete time signal (audio samples, list of float)
- **low_f** – -3db point (?) for highpass filter (Hz)
- **high_f** – -3db point (?) for highpass filter (Hz)
- **sample_rate** – samples per second (Hz)
- **order=9** – higher values -> steeper dropoff

Returns filtered time signal

opensoundscape.audio_tools.**butter_bandpass**(*low_f*, *high_f*, *sample_rate*, *order=9*)
generate coefficients for bandpass_filter()

Parameters

- **low_f** – low frequency of butterworth bandpass filter
- **high_f** – high frequency of butterworth bandpass filter
- **sample_rate** – audio sample rate
- **order=9** – order of butterworth filter

Returns set of coefficients used in sosfiltfilt()

opensoundscape.audio_tools.**clipping_detector**(*samples*, *threshold=0.6*)
count the number of samples above a threshold value

Parameters

- **samples** – a time series of float values
- **threshold=0.6** – minimum value of sample to count as clipping

Returns number of samples exceeding threshold

opensoundscape.audio_tools.**convolve_file**(*in_file*, *out_file*, *ir_file*, *input_gain=1.0*)
apply an impulse_response to a file using ffmpeg's afir convolution

ir_file is an audio file containing a short burst of noise recorded in a space whose acoustics are to be recreated

this makes the files 'sound as if' it were recorded in the location that the impulse response (ir_file) was recorded

Parameters

- **in_file** – path to an audio file to process
- **out_file** – path to save output to
- **ir_file** – path to impulse response file
- **input_gain=1.0** – ratio for in_file sound's amplitude in (0,1)

Returns os response of ffmpeg command

opensoundscape.audio_tools.**mixdown_with_delays**(*files_to_mix*, *destination*, *delays=None*, *levels=None*, *duration='first'*, *verbose=0*, *create_txt_file=False*)

use ffmpeg to mixdown a set of audio files, each starting at a specified time (padding beginnings with zeros)

Parameters

- **files_to_mix** – list of audio file paths
- **destination** – path to save mixdown to
- **delays=None** – list of delays (how many seconds of zero-padding to add at beginning of each file)
- **levels=None** – optionally provide a list of relative levels (amplitudes) for each input
- **duration='first'** – ffmpeg option for duration of output file: match duration of 'longest','shortest',or 'first' input file
- **verbose=0** – if >0, prints ffmpeg command and doesn't suppress ffmpeg output (command line output is returned from this function)
- **create_txt_file=False** – if True, also creates a second output file which lists all files that were included in the mixdown

Returns ffmpeg command line output

opensoundscape.audio_tools.**silence_filter**(*filename*, *smoothing_factor=10*, *window_len_samples=256*, *overlap_len_samples=128*, *threshold=None*)

Identify whether a file is silent (0) or not (1)

Load samples from an mp3 file and identify whether or not it is likely to be silent. Silence is determined by finding the energy in windowed regions of these samples, and normalizing the detected energy by the average energy level in the recording.

If any windowed region has energy above the threshold, returns a 0; else returns 1.

Parameters

- **filename** (*str*) – file to inspect
- **smoothing_factor** (*int*) – modifier to window_len_samples
- **window_len_samples** – number of samples per window segment
- **overlap_len_samples** – number of samples to overlap each window segment
- **threshold** – threshold value (experimentally determined)

Returns 0 if file contains no significant energy over bakcground 1 if file contains significant energy over bakcground

If threshold is None: returns net_energy over background noise

opensoundscape.audio_tools.**window_energy**(*samples*, *window_len_samples=256*, *overlap_len_samples=128*)

Calculate audio energy with a sliding window

Calculate the energy in an array of audio samples

Parameters

- **samples** (*np.ndarray*) – array of audio samples loaded using librosa.load

- **window_len_samples** – samples per window
- **overlap_len_samples** – number of samples shared between consecutive windows

Returns list of energy level (float) for each window

7.3 Commands

opensoundscape.commands.**run_command**(*cmd*)
Run a command returning output, error

Input: cmd: A string containing some command

Output: (stdout, stderr): A tuple of standard out and standard error

opensoundscape.commands.**run_command_return_code**(*cmd*)
Run a command returning the return code

Input: cmd: A string containing some command

Output: return_code: The return code of the function

7.4 Completions

7.5 Config

opensoundscape.config.**get_default_config**()
Get the default configuration file as a dictionary

Output: dict: A dictionary containing the default Opensoundscape configuration

opensoundscape.config.**validate**(*config*)
Validate a configuration string

Input: config: A string containing an Opensoundscape configuration

Output: dict: A dictionary of the validated Opensoundscape configuration

opensoundscape.config.**validate_file**(*fname*)
Validate a configuration file

Input: fname: A filename containing an Opensoundscape configuration

Output: dict: A dictionary of the validated Opensoundscape configuration

7.6 Console Checks

Utilities related to console checks on docopt args

7.7 Console

console.py: Entrypoint for opensoundscape

opensoundscape.console.**build_docs**()
Run sphinx-build for our project

opensoundscape.console.**entrypoint**()
The Opensoundscape entrypoint for console interaction

7.8 Data Selection

opensoundscape.data_selection.**add_binary_numeric_labels**(*input_df*, *label*, *input_column='Labels'*, *output_column='NumericLabels'*)
Add binary numeric labels to dataframe based on label

Given a dataframe and a label from input_column produce a new dataframe with an output_column and a label map

Parameters

- **input_df** – A dataframe
- **label** – The label to set to 1
- **input_column** – The column to read labels from
- **output_column** – The column to write numeric labels to

Output: output_df: A dataframe with an additional output_column label_map: A dictionary, keys are f"not_{label}" and f"{label}", values are 0 and 1

opensoundscape.data_selection.**add_numeric_labels**(*input_df*, *input_column='Labels'*, *output_column='NumericLabels'*)
Add numeric labels to dataframe

Given a dataframe with input_column produce a new dataframe with an output_column and a label map

Parameters

- **input_df** – A dataframe
- **input_column** – The column to read labels from
- **output_column** – The column to write numeric labels to

Output: output_df: A dataframe with an additional output_column label_map: A dictionary, keys are the unique labels and monotonically increasing values starting at 0

opensoundscape.data_selection.**expand_multi_labeled**(*input_df*)
Given a multi-labeled dataframe, generate a singly-labeled dataframe

Given a Dataframe with a "Labels" column that is multi-labeled (e.g. "hello|world") split the row into singly labeled rows.

Parameters **input_df** – A Dataframe with a multi-labeled "Labels" column (separated by "|")

Output: output_df: A Dataframe with singly-labeled "Labels" column

opensoundscape.data_selection.**train_valid_split**(*input_df*, *stratify_from_column='Labels'*, *train_size=0.8*, *random_state=101*)
Split a dataframe into train and validation dataframes

Given an input dataframe with a labels column split each unique label into a train size and 1 - train_size for training and validation sets. If stratify_from_column is *None* don't stratify.

Parameters

- **input_df** – A dataframe
- **stratify_from_column** – Name of the column that labels should come from [default: "Labels"] - given *None* will not attempt stratified sampling
- **train_size** – The decimal fraction to use for the training set [default: 0.8]
- **random_state** – The random state to use for train_test_split [default: 101]

Output: train_df: A Dataframe containing the training set valid_df: A Dataframe containing the validation set

opensoundscape.data_selection.**upsample**(*input_df*, *label_column='Labels'*, *random_state=None*)

Given a input DataFrame upsample to maximum value

Upsampling removes the class imbalance in your dataset. Rows for each label are repeated up to *max_count // rows*. Then, we randomly sample the rows to fill up to *max_count*.

Input: input_df: A DataFrame to upsample label_column: The column to draw unique labels from random_state: Set the random_state during sampling

Output: df: An upsampled DataFrame

7.9 Datasets

class opensoundscape.datasets.**SingleTargetAudioDataset**(*df*, *label_dict*, *filename_column='Destination'*, *from_audio=True*, *label_column=None*, *height=224*, *width=224*, *add_noise=False*, *save_dir=None*, *random_trim_length=None*, *extend_short_clips=False*, *max_overlay_num=0*, *overlay_prob=0.2*, *overlay_weight='random'*, *overlay_class=None*, *audio_sample_rate=22050*, *debug=None*)

Single Target Audio -> Image Dataset

Given a DataFrame with audio files in one of the columns, generate a Dataset of spectrogram images for basic machine learning tasks.

This class provides access to several types of augmentations that act on audio and images with the following arguments: - add_noise: for adding RandomAffine and ColorJitter noise to images - random_trim_length: for only using a short random clip extracted from the training data - max_overlay_num / overlay_prob / overlay_weight:

controlling the maximum number of additional spectrograms to overlay, the probability of overlaying an individual spectrogram, and the weight for the weighted sum of the spectrograms

Additional augmentations on tensors are available when calling *train()* from the module *opensoundscape.torch.train*.

Input: df: A DataFrame with a column containing audio files label_dict: a dictionary mapping numeric labels to class names,

- for example: {0:'American Robin',1:'Northern Cardinal'}
- pass *None* if you wish to retain numeric labels

filename_column: The column in the DataFrame which contains paths to data [default: Destination]

from_audio: Whether the raw dataset is audio [default: True] label_column: The column with numeric labels if present [default: None] height: Height for resulting Tensor [default: 224] width: Width for resulting Tensor [default: 224] add_noise: Apply RandomAffine and ColorJitter filters [default: False] save_dir: Save images to a directory [default: None] random_trim_length: Extract a clip of this many seconds of audio

starting at a random time. If None, the original clip will be used [default: None]

extend_short_clips: If a file to be overlaid or trimmed from is too short, extend it to the desired length by repeating it. [default: False]

max_overlay_num: The maximum number of additional images to overlay, each with probability overlay_prob [default: 0]

overlay_prob: Probability of an image from a different class being overlayed (combined as a weighted sum) on the training image. typical values: 0, 0.66 [default: 0.2]

overlay_weight: The weight given to the overlaid image during augmentation. When 'random', will randomly select a different weight between 0.2 and 0.5 for each overlay. When not 'random', should be a float between 0 and 1 [default: 'random']

overlay_class: The label of the class that overlays should be drawn from. Must be specified if max_overlay_num > 0. If 'different', draws overlays from any class that is not the same class as the audio. If set to a class label, draws overlays from that class. When creating a presence/absence classifier, set overlay_class equal to the absence class label [default: None]

audio_sample_rate: resample audio to this sample rate; specify None to use original audio sample rate [default: 22050]

debug: path to save img files, images are created from the tensor immediately before it is returned. When None, does not save images. [default: None]

Output:

Dictionary: { "X": (3, H, W) , "y": (1) if label_column != None }

image_from_audio(*audio*, *mode='RGB'*)
Create a PIL image from audio

Inputs: audio: audio object mode: PIL image mode, e.g. "L" or "RGB" [default: RGB]

overlay_random_image(*original_image*, *original_length*, *original_class*, *original_path*)
Overlay an image from another class

Select a random file from a different class. Trim if necessary to the same length as the given image. Overlay the images on top of each other with a weight

class opensoundscape.datasets.**SplitterDataset**(*wavs*, *annotations=False*, *label_corrections=None*, *overlap=1*, *duration=5*, *output_directory='segments'*, *include_last_segment=False*, *column_separator='t'*, *species_separator='|'*)

A PyTorch Dataset for splitting a WAV files

Inputs: wavs: A list of WAV files to split annotations: Should we search for corresponding annotations files? (default: False) label_corrections: Specify a correction labels CSV file w/ column headers "raw" and "corrected" (default: None) overlap: How much overlap should there be between samples (units: seconds, default: 1) duration: How long should each segment be? (units: seconds, default: 5) output_directory Where should segments be written? (default: segments/) include_last_segment: Do you want to include the last segment? (default: False) column_separator: What character should we use to separate columns (default: " ") species_separator: What character should we use to separate species (default: "|")

Effects:

- Segments will be written to the *output_directory*

Outputs:

output: A list of CSV rows (separated by *column_separator*) containing the source audio, segment begin time (seconds), segment end time (seconds), segment audio, and present classes separated by *species_separator* if annotations were requested

opensoundscape.datasets.**annotations_with_overlaps_with_clip**(*df*, *begin*, *end*)
Determine if any rows overlap with current segment

Inputs: df: A dataframe containing a Raven annotation file begin: The begin time of the current segment (unit: seconds) end: The end time of the current segment (unit: seconds)

Output: sub_df: A dataframe of annotations which overlap with the begin/end times

opensoundscape.datasets.**get_md5_digest**(*input_string*)
Generate MD5 sum for a string

Inputs: input_string: An input string

Outputs: output: A string containing the md5 hash of input string

7.10 Grad Cam

7.11 Helpers

opensoundscape.helpers.**binarize**(*x*, *threshold*)
return a list of 0, 1 by thresholding vector x

opensoundscape.helpers.**bound**(*x*, *bounds*)
restrict x to a range of bounds = [min, max]

opensoundscape.helpers.**file_name**(*path*)
get file name without extension from a path

opensoundscape.helpers.**hex_to_time**(*s*)
convert a hexidecimal, Unix time string to a datetime timestamp

opensoundscape.helpers.**isNan**(*x*)
check for nan by equating x to itself

opensoundscape.helpers.**jitter**(*x*, *width*, *distribution='gaussian'*)
Jitter (add random noise to) each value of x

Parameters

- **x** – scalar, array, or nd-array of numeric type

- **width** – multiplier for random variable (stdev for 'gaussian' or r for 'uniform')
- **distribution** – 'gaussian' (default) or 'uniform' if 'gaussian': draw jitter from gaussian with mu = 0, std = width if 'uniform': draw jitter from uniform on [-width, width]

Returns x + random jitter

Return type jittered_x

opensoundscape.helpers.**linear_scale**(*array*, *in_range=(0, 1)*, *out_range=(0, 255)*)
Translate from range in_range to out_range

Inputs: in_range: The starting range [default: (0, 1)] out_range: The output range [default: (0, 255)]

Outputs: new_array: A translated array

opensoundscape.helpers.**min_max_scale**(*array*, *feature_range=(0, 1)*)
rescale vaues in an a array linearly to feature_range

opensoundscape.helpers.**rescale_features**(*X*, *rescaling_vector=None*)
rescale all features by dividing by the max value for each feature

optionally provide the rescaling vector (1xlen(X) np.array), so that you can rescale a new dataset consistently with an old one

returns rescaled feature set and rescaling vector

opensoundscape.helpers.**run_command**(*cmd*)
run a bash command with Popen, return response

opensoundscape.helpers.**sigmoid**(*x*)
sigmoid function

7.12 Localization

opensoundscape.localization.**calc_speed_of_sound**(*temperature=20*)
Calculate speed of sound in meters per second

Calculate speed of sound for a given temperature in Celsius (Humidity has a negligible effect on speed of sound and so this functionality is not implemented)

Parameters **temperature** – ambient temperature in Celsius

Returns the speed of sound in meters per second

opensoundscape.localization.**localize**(*receiver_positions*, *arrival_times*, *temperature=20.0*, *invert_alg='gps'*, *center=True*, *pseudo=True*)
Perform TDOA localization on a sound event

Localize a sound event given relative arrival times at multiple receivers. This function implements a localization algorithm from the equations described in the class handout ("Global Positioning Systems"). Localization can be performed in a global coordinate system in meters (i.e., UTM), or relative to recorder positions in meters.

Parameters

- **receiver_positions** – a list of [x,y,z] positions for each receiver Positions should be in meters, e.g., the UTM coordinate system.
- **arrival_times** – a list of TDOA times (onset times) for each recorder The times should be in seconds.
- **temperature** – ambient temperature in Celsius

- **invert_alg** – what inversion algorithm to use
- **center** – whether to center recorders before computing localization result. Computes localization relative to centered plot, then translates solution back to original recorder locations. (For behavior of original Sound Finder, use True)
- **pseudo** – whether to use the pseudorange error (True) or sum of squares discrepancy (False) to pick the solution to return (For behavior of original Sound Finder, use False. However, in initial tests, pseudorange error appears to perform better.)

Returns The solution (x,y,z,b) with the lower sum of squares discrepancy b is the error in the pseudorange (distance to mics), b=c*delta_t (delta_t is time error)

opensoundscape.localization.**lorentz_ip**(*u*, *v=None*)

Compute Lorentz inner product of two vectors

For vectors *u* and *v*, the Lorentz inner product for 3-dimensional case is defined as

u[0]*v[0] + u[1]*v[1] + u[2]*v[2] - u[3]*v[3]

Or, for 2-dimensional case as

u[0]*v[0] + u[1]*v[1] - u[2]*v[2]

Args u: vector with shape either (3,) or (4,) v: vector with same shape as x1; if None (default), sets v = u

Returns float: value of Lorentz IP

opensoundscape.localization.**travel_time**(*source*, *receiver*, *speed_of_sound*)

Calculate time required for sound to travel from a souce to a receiver

Parameters

- **source** – cartesian position [x,y] or [x,y,z] of sound source
- **receiver** – cartesian position [x,y] or [x,y,z] of sound receiver
- **speed_of_sound** – speed of sound in m/s

Returns time in seconds for sound to travel from source to receiver

7.13 Metrics

class opensoundscape.metrics.**Metrics**(*classes*, *dataset_len*)

Basic Example

See opensoundscape.torch.train for an in-depth example

‘‘‘ dataset = Dataset(. . .) dataloader = DataLoader(dataset, . . .) classes = [0, 1, 2, 3, 4] # An example list of classes for epoch in epochs:

metrics = Metrics(classes, len(dataset)) for batch in dataloader:

X, y = batch[“X”], batch[“y”] targets = y.squeeze(0) # dim: (batch_size) . . . loss = . . . # dim: (0) predictions = . . . # dim: (batch_size) metrics.accumulate_batch_metrics(

loss.item(), targets.cpu(), predictions.cpu()

)

metrics_dictionary = metrics.compute_epoch_metrics()

‘‘‘

accumulate_batch_metrics(*loss*, *targets*, *predictions*)
For a batch, accumulate loss and confusion matrix

For validation pass 0 for loss.

Parameters

- **loss** – The loss for this batch
- **targets** – The correct y labels
- **predictions** – The predicted labels

compute_epoch_metrics()
Compute metrics from learning

Computes the loss and accuracy, precision, recall, and f1 scores from the confusion matrix and returns dictionary with metric name as keys and their corresponding values

Returns [loss, accuracy, precision, recall, f1, confusion_matrix]

Return type dictionary with keys

7.14 Pulse Finder

7.15 PyTorch Prediction

DEPRECATED: use opensoundscape.torch.predict instead

these functions are currently used only to support *localization.py* the module contains a pytorch prediction function (deprecated) and some additional functionality for using gradcam

opensoundscape.pytorch_prediction.**activation_region_limits**(*gcam*, *threshold=0.2*)
calculate bounds of a GradCam activation region

Parameters

- **gcam** – a 2-d array gradcam activation array generated by gradcam_region()
- **threshold=0.2** – minimum value of gradcam (0-1) to count as 'activated'

Returns [[min row, max_row], [min_col, max_col]] indices of gradcam elements exceeding threshold

opensoundscape.pytorch_prediction.**activation_region_to_box**(*activation_region*, *threshold=0.2*)
draw a rectangle of the activation box as a boolean array (useful for plotting a mask over a spectrogram)

Parameters

- **activation_region** – a 2-d gradcam activation array
- **threshold=0.2** – minimum value of activation to count as 'activated'

Returns mask 2-d array of 0, 1 where 1's form a solid box of activated region

opensoundscape.pytorch_prediction.**gradcam_region**(*model*, *img_paths*, *img_shape*, *predictions=None*, *save_gcams=True*, *box_threshold=0.2*)
Compute the GradCam activation region (the area of an image that was most important for classification in the CNN)

Parameters

- **model** – a pytorch model object
- **img_paths** – list of paths to image files
- **= None** (*predictions*) – [list of float] optionally, provide model predictions per file to avoid re-computing
- **= True** (*save_gcams*) – bool, if False only box regions around gcams are saved

Returns limits of the box surrounding the gcam activation region, as indices: [[min row, max row], [min col, max col]] gcams: (only returned if save_gcams == True) arrays with gcam activation values, shape = shape of image

Return type boxes

opensoundscape.pytorch_prediction.**in_box**(*x*, *y*, *box_lims*)
check if an x, y position falls within a set of limits

Parameters

- **x** – first index
- **y** – second index
- **box_lims** – [[x low,x high], [y low,y high]]

Returns: True if (x,y) is in box_lims, otherwise False

opensoundscape.pytorch_prediction.**predict**(*model*, *img_paths*, *img_shape*, *batch_size=1*, *num_workers=12*, *apply_softmax=True*)
get multi-class model predictions from a pytorch model for a set of images

Parameters

- **model** – a pytorch model object (not path to weights)
- **img_paths** – a list of paths to RGB png spectrograms
- **batch_size=1** – pytorch parallelization parameter
- **num_workers=12** – pytorch parallelization parameter
- **apply_softmax=True** – if True, performs a softmax on raw output of network

returns: df of predictions indexed by file

7.16 Raven

raven.py: Utilities for dealing with Raven files

opensoundscape.raven.**annotation_check**(*directory*)
Check Raven annotations files for a non-null class

Input: directory: The path which contains Raven annotations file

Output: None

opensoundscape.raven.**generate_class_corrections**(*directory*)
Generate a CSV to specify any class overrides

Input: directory: The path which contains Raven annotations files ending in *.selections.txt.lower

Output:

csv (string): A multiline string containing a CSV file with two columns *raw* and *corrected*

opensoundscape.raven.**lowercase_annotations**(*directory*)
Convert Raven annotation files to lowercase

Input: directory: The path which contains Raven annotations file

Output: None

opensoundscape.raven.**query_annotations**(*directory*, *cls*)
Given a directory of Raven annotations, query for a specific class

Input: directory: The path which contains Raven annotations file cls: The class which you would like to query for

Output: output (string): A multiline string containing annotation file and rows matching the query cls

7.17 Species Table

7.18 Spectrogram

spectrogram.py: Utilities for dealing with spectrograms

class opensoundscape.spectrogram.**Spectrogram**(*spectrogram*, *frequencies*, *times*)
Immutable spectrogram container

amplitude(*freq_range=None*)
create an amplitude vs time signal from spectrogram

by summing pixels in the vertical dimension

Args freq_range=None: sum Spectrogrm only in this range of [low, high] frequencies in Hz (if None, all frequencies are summed)

Returns a time-series array of the vertical sum of spectrogram value

bandpass(*min_f*, *max_f*)
extract a frequency band from a spectrogram

crops the 2-d array of the spectrograms to the desired frequency range

Parameters

- **min_f** – low frequency in Hz for bandpass
- **high_f** – high frequency in Hz for bandpass

Returns bandpassed spectrogram object

classmethod from_audio(*audio*, *window_type='hann'*, *window_samples=512*, *overlap_samples=256*, *decibel_limits=(-100*, *-20)*)
create a Spectrogram object from an Audio object

Parameters

- **window_type="hann"** – see scipy.signal.spectrogram docs for description of window parameter
- **window_samples=512** – number of audio samples per spectrogram window (pixel)
- **overlap_samples=256** – number of samples shared by consecutive windows
- **=** (*decibel_limits*) – limit the dB values to (min,max) (lower values set to min, higher values set to max)

Returns opensoundscape.spectrogram.Spectrogram object

classmethod from_file()
create a Spectrogram object from a file

Parameters **file** – path of image to load

Returns opensoundscape.spectrogram.Spectrogram object

limit_db_range(*min_db=-100*, *max_db=-20*)
Limit the decibel values of the spectrogram to range from min_db to max_db

values less than min_db are set to min_db values greater than max_db are set to max_db

similar to Audacity's gain and range parameters

Parameters

- **min_db** – values lower than this are set to this
- **max_db** – values higher than this are set to this

Returns Spectrogram object with db range applied

linear_scale(*feature_range=(0, 1)*)
Linearly rescale spectrogram values to a range of values using in_range as decibel_limits

Parameters **feature_range** – tuple of (low,high) values for output

Returns Spectrogram object with values rescaled to feature_range

min_max_scale(*feature_range=(0, 1)*)
Linearly rescale spectrogram values to a range of values using in_range as minimum and maximum

Parameters **feature_range** – tuple of (low,high) values for output

Returns Spectrogram object with values rescaled to feature_range

net_amplitude(*signal_band*, *reject_bands=None*)
create amplitude signal in signal_band and subtract amplitude from reject_bands

rescale the signal and reject bands by dividing by their bandwidths in Hz (amplitude of each reject_band is divided by the total bandwidth of all reject_bands. amplitude of signal_band is divided by badwidth of signal_band.)

Parameters

- **signal_band** – [low,high] frequency range in Hz (positive contribution)
- **band** (*reject*) – list of [low,high] frequency ranges in Hz (negative contribution)

return: time-series array of net amplitude

plot(*inline=True*, *fname=None*, *show_colorbar=False*)
Plot the spectrogram with matplotlib.pyplot

Parameters

- **inline=True** –
- **fname=None** – specify a string path to save the plot to (ending in .png/.pdf)
- **show_colorbar** – include image legend colorbar from pyplot

to_image(*shape=None, mode='RGB', spec_range=[-100, -20]*)
create a Pillow Image from spectrogram linearly rescales values from db_range (default [-100, -20]) to [255,0] (ie, -20 db is loudest -> black, -100 db is quietest -> white)

Parameters

- **destination** – a file path (string)
- **shape=None** – tuple of image dimensions, eg (224,224)
- **mode="RGB"** – RGB for 3-channel color or "L" for 1-channel grayscale
- **spec_range=[-100,-20]** – the lowest and highest possible values in the spectrogram

Returns Pillow Image object

trim(*start_time*, *end_time*)
extract a time segment from a spectrogram

Parameters

- **start_time** – in seconds
- **end_time** – in seconds

Returns spectrogram object from extracted time segment

7.19 Taxa

a set of utilites for converting between scientific and common names of bird species in different naming systems (xeno canto and bird net)

opensoundscape.taxa.**bn_common_to_sci**(*common*)
convert bird net common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

opensoundscape.taxa.**common_to_sci**(*common*)
convert bird net common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

opensoundscape.taxa.**get_species_list**()
list of scientific-names (lowercase-hyphenated) of species in the loaded species table

opensoundscape.taxa.**sci_to_bn_common**(*scientific*)
convert scientific name as lowercase-hyphenated to birdnet common name as lowercasenospaces

opensoundscape.taxa.**sci_to_xc_common**(*scientific*)
convert scientific name as lowercase-hyphenated to xeno-canto common name as lowercasenospaces

opensoundscape.taxa.**xc_common_to_sci**(*common*)
convert xeno-canto common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

7.20 Torch Spectrogram Augmentation

These functions were implemented for PyTorch in the following repository https://github.com/zcaceres/spec_augment
The original paper is available on https://arxiv.org/abs/1904.08779

7.21 Torch Training

opensoundscape.torch.train.**train**(*save_dir*, *model*, *train_dataset*, *valid_dataset*, *optimizer*, *loss_fn*, *epochs=25*, *batch_size=1*, *num_workers=0*, *log_every=5*, *tensor_augment=False*, *debug=False*, *print_logging=True*, *save_scores=False*)

Train a model

Input: save_dir: A directory to save intermediate results model: A binary torch model,

- e.g. torchvision.models.resnet18(pretrained=True)
- must override classes, e.g. model.fc = torch.nn.Linear(model.fc.in_features, 2)

train_dataset: The training Dataset, e.g. created by SingleTargetAudioDataset() valid_dataset: The validation Dataset, e.g. created by SingleTargetAudioDataset() optimizer: A torch optimizer, e.g. torch.optim.SGD(model.parameters(), lr=1e-3) loss_fn: A torch loss function, e.g. torch.nn.CrossEntropyLoss() epochs: The number of epochs [default: 25] batch_size: The size of the batches [default: 1] num_workers: The number of cores to use for batch preparation [default: 1] log_every: Log statistics when epoch % log_every == 0 [default: 5] tensor_augment: Whether or not to use the tensor augment procedures [default: False] debug: Whether or not to write intermediate images [default: False] print_logging: Whether to print training progress to stdout [default: True] save_scores: Whether to save the scores on the train/val set each epoch [default: False]

Side Effects: Write a file *epoch-{epoch}.tar* containing (rate of *log_every*): - Model state dictionary - Optimizer state dictionary - Labels in YAML format - Train: loss, accuracy, precision, recall, and f1 score - Validation: accuracy, precision, recall, and f1 score - train_dataset.label_dict Write a metadata file with parameter values to save_dir/metadata.txt

Output: None

Effects: model parameters are saved to

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

T

U

V

W

X