opensoundscape

Release 0.8.0

Jan 12, 2023

Contents

OpenSoundscape is free and open source software for the analysis of bioacoustic recordings (GitHub). 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 Installation section below provides 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

Mac and Linux

OpenSoundscape can be installed on Mac and Linux machines with Python 3.7, 3.8, or 3.9 using the pip command `pip install opensoundscape==0.8.0`. We recommend installing OpenSoundscape in a virtual environment to prevent dependency conflicts.

Below are instructions for installation with two package managers:

- `conda`: Python and package management through Anaconda, a package manager popular among scientific programmers
- `venv`: Python's included virtual environment manager, `venv`

Feel free to use another virtual environment manager (e.g. `virtualenvwrapper`) if desired.

1.1 Installation via Anaconda

- Install Anaconda if you don't already have it.
  - Download the installer here, or
  - follow the installation instructions for your operating system.
- Create a Python (3.7, 3.8, or 3.9) conda environment for opensoundscape: `conda create --name opensoundscape pip python=3.9`
- Activate the environment: `conda activate opensoundscape`
- Install opensoundscape using pip: `pip install opensoundscape==0.8.0`
- Deactivate the environment when you're done using it: `conda deactivate`

1.2 Installation via `venv`

Download Python 3.7, 3.8, or 3.8 from this website.

Run the following commands in your bash terminal:

- Check that you have installed Python 3.7, 3.8, or 3.9._: `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`
- Activate/use the environment: `source opensoundscape/bin/activate`
- Install OpenSoundscape in the environment: `pip install opensoundscape==0.8.0`
- 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`.

CHAPTER 2

Windows

We recommend that Windows users install and use OpenSoundscape using Windows Subsystem for Linux, because some of the machine learning and audio processing packages required by OpenSoundscape do not install easily on Windows computers. Below we describe the typical installation method. This gives you access to a Linux operating system (we recommend Ubuntu 20.04) in which to use Python and install and use OpenSoundscape. Using Ubuntu 20.04 is as simple as opening a program on your computer.

2.1 Get Ubuntu shell

If you don't already use Windows Subsystem for Linux (WSL), activate it using the following:

- Search for the "Powershell" program on your computer
- Right click on "Powershell," then click "Run as administrator" and in the pop-up, allow it to run as administrator
- Install WSL2 (more information: https://docs.microsoft.com/en-us/windows/wsl/install-win10):

```
wsl --install
```

- Restart your computer

Once you have WSL, follow these steps to get an Ubuntu shell on your computer:

- Open Windows Store, search for "Ubuntu" and click "Ubuntu 20.04 LTS"
- Click "Get", wait for the program to download, then click "Launch"
- An Ubuntu shell will open. Wait for Ubuntu to install.
- Set username and password to something you will remember
- Run `sudo apt update` and type in the password you just set

2.2 Download Anaconda

We recommend installing OpenSoundscape in a package manager. We find that the easiest package manager for new users is "Anaconda," a program which includes Python and tools for managing Python packages. Below are instructions for downloading Anaconda in the Ubuntu environment.

- Open this page and scroll down to the "Anaconda Installers" section. Under the Linux section, right click on the link "64-Bit (x86) Installer" and click "Copy link"'
- Download the installer:
  - Open the Ubuntu terminal
  - Type in `wget` then paste the link you copied, e.g.: (the filename of your file may differ)

```
wget https://repo.anaconda.com/archive/Anaconda3-2020.07-Linux-x86_64.sh
```

- Execute the downloaded installer, e.g.: (the filename of your file may differ)

```
bash Anaconda3-2020.07-Linux-x86_64.sh
```

  - Press ENTER, read the installation requirements, press Q, then type "yes" and press enter to install
  - Wait for it to install
  - If your download hangs, press CTRL+C, `rm -rf ~/anaconda3` and try again
- Type "yes" to initialize `conda`
  - If you skipped this step, initialize your conda installation: run `source ~/anaconda3/bin/activate` and then after that command has run, `conda init`.
- Remove the downloaded file after installation, e.g. `rm Anaconda3-2020.07-Linux-x86_64.sh`
- Close and reopen terminal window to have access to the initialized Anaconda distribution

You can now manage packages with `conda`.

2.3 Install OpenSoundscape in virtual environment

- Create a Python (3.7, 3.8, or 3.9) conda environment for opensoundscape: `conda create --name opensoundscape pip python=3.9`
- Activate the environment: `conda activate opensoundscape`
- Install opensoundscape using pip: `pip install opensoundscape==0.8.0`

If you see an error that says "No matching distribution found...", your best bet is to use these commands to download then install the package:

```
cd
git clone https://github.com/kitzeslab/opensoundscape.git
cd opensoundscape/
pip install .
```

If you run into this error and you are on a Windows 10 machine:

```
(opensoundscape_environment) username@computername:~$ pip install opensoundscape==0.8.
↪0
WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None,␣
↪status=None)) after connection broken by 'NewConnectionError('<pip._vendor.urllib3.
↪connection.HTTPSConnection object at 0x7f7603c5da90>: Failed to establish a new␣
↪connection: [Errno -2] Name or service not known')': /simple/opensoundscape/
```

You may be able to solve it by going to System Settings, searching for "Proxy Settings," and beneath "Automatic proxy setup," turning "Automatically detect settings" OFF. Restart your terminal for changes to take effect. Then activate the environment and install OpenSoundscape using pip.

CHAPTER 3

Contributors

Contributors and advanced users can use this workflow to install OpenSoundscape using Poetry. 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.

3.1 Poetry installation

- Install poetry
- Create a new virtual environment for the OpenSoundscape installation. If you are using Anaconda, you can create a new environment with `conda create -n opso-dev python=3.9` where `opso-dev` is the name of the new virtual environment. Use `conda activate opso-dev` to enter the environment to work on OpenSoundscape and `conda deactivate opso-dev` to return to your base Python installation. If you are not using Anaconda, other packages such as `virtualenv` should work as well. Ensure that the Python version is compatible with the current version of OpenSoundscape.
- **Internal Contributors**: Clone this github repository to your machine: `git clone https://github.com/kitzeslab/opensoundscape.git`
- **External 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`
- Install OpenSoundscape using `poetry install`. This will install OpenSoundscape and its dependencies into the `opso-dev` virtual environment. By default it will install OpenSoundscape in develop mode, so that updated code in the respository can be imported without reinstallation.
  - If you are on a Mac and poetry install fails to install `numba`, contact one of the developers for help troubleshooting your issues.
- Install the `ffmpeg` dependency. On a Mac, `ffmpeg` can be installed using Homebrew.
- Run the test suite to ensure that everything installed properly. From the top-level directory, run the command `pytest`.

3.2 Contribution workflow

3.2.1 Contributing to code

Make contributions by editing the code in your repo. Create branches for features by starting with the `develop` branch and then running `git checkout -b feature_branch_name`. Once work is complete, push the new branch 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 or a pull request. Before opening a PR, do the following to ensure the code is consistent with the rest of the package:

- Run the test suite using `pytest`
- Format the code with `black` style (from the top level of the repo): `black .`

3.2.2 Contributing to documentation

Build the documentation using `sphinx-build docs docs/_build`

CHAPTER 4

Jupyter

To use OpenSoundscape in JupyterLab or in a Jupyter Notebook, you may either start Jupyter from within your OpenSoundscape virtual environment and use the "Python 3" kernel in your notebooks, or create a separate "OpenSoundscape" kernel using the instructions below

The following steps assume you have already used your operating system-specific installation instructions to create a virtual environement containing OpenSoundscape and its dependencies.

4.1 Use virtual environment

- Activate your virtual environment
- Start JupyterLab or Jupyter Notebook from inside the conda environment, e.g.: `jupyter lab`
- Copy and paste the JupyterLab link into your web browser

With this method, the default "Python 3" kernel will be able to import `opensoundscape` modules.

4.2 Create independent kernel

Use the following steps to create a kernel that appears in any notebook you open, not just notebooks opened from your virtual environment.

- Activate your virtual environment to have access to the `ipykernel` package
- Create ipython kernel with the following command, replacing `ENV_NAME` with the name of your OpenSoundscape virtual environment.

```
python -m ipykernel install --user --name=ENV_NAME --display-name=OpenSoundscape
```

- Now when you make a new notebook on JupyterLab, or change kernels on an existing notebook, you can choose to use the "OpenSoundscape" Python kernel

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:

```
%load_ext autoreload
%autoreload
```

CHAPTER 5

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. Using it requires that you install OpenSoundscape and follow the instructions for using it in Jupyter.

As an example, we will download a file from the Kitzes Lab box location using the code below, and use it throughout the tutorial. To use your own file for the following examples, change the string assigned to `audio_filename` to any audio file on your computer.

[1]:
```
import subprocess
subprocess.run(['curl',
                'https://pitt.box.com/shared/static/z73eked7quh1t2pp93axzrrpq6wwydx0.
↪wav',
                 '-L', '-o', '1min_audio.wav'])
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      5      0 --:--:--  0:00:01 --:--:--  7000
100 3750k  100 3750k    0     0  1408k      0  0:00:02  0:00:02 --:--:-- 7508k
```

[1]:
```
CompletedProcess(args=['curl', 'https://pitt.box.com/shared/static/
↪z73eked7quh1t2pp93axzrrpq6wwydx0.wav', '-L', '-o', '1min_audio.wav'], returncode=0)
```

[2]:
```
audio_filename = './1min_audio.wav'
```

5.1 Quick start

Import the `Audio` and `Spectrogram` classes from OpenSoundscape. (For more information about Python imports, review this article.)

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

These classes provide a variety of tools to load and manipulate audio and spectrograms. The code below demonstrates a basic pipeline:

- load an audio file
- generate a spectrogram with default parameters
- create a 224px X 224px-sized image of the spectrogram
- save the image to a file

```
[4]: from pathlib import Path

# Settings
image_shape = (224, 224) #(height, width) not (width, height)
image_save_path = Path('./saved_spectrogram.png')

# Load audio file as Audio object
audio = Audio.from_file(audio_filename)

# Create Spectrogram object from Audio object
spectrogram = Spectrogram.from_audio(audio)

# Convert Spectrogram object to Python Imaging Library (PIL) Image
image = spectrogram.to_image(shape=image_shape,invert=True)

# Save image to file
image.save(image_save_path)
```

The above function calls could even be condensed to a single line:

```
[5]: Spectrogram.from_audio(Audio.from_file(audio_filename)).to_image(shape=image_shape,
↪invert=True).save(image_save_path)
```

Clean up by deleting the spectrogram saved above.

```
[6]: image_save_path.unlink()
```

5.2 Audio loading

The `Audio` class in OpenSoundscape allows loading and manipulation of audio files.

5.2.1 Load .wav(s)

Load the example audio from file:

[7]:
```
audio_object = Audio.from_file(audio_filename)
```

5.2.2 Load .mp3(s)

OpenSoundscape uses a package called `librosa` to help load audio files. Librosa automatically supports `.wav` files, but loading `.mp3` files requires that you also install `ffmpeg` or an alternative. See librosa's installation tips for more information.

5.2.3 Load a segment of a file

We can directly load a section of a `.wav` file very quickly (even if the audio file is large) using the `offset` and `duration` parameters.

For example, let's load 1 second of audio from 2.0-3.0 seconds after the start of the file:

[8]:
```
audio_segment = Audio.from_file(audio_filename,offset=2.0,duration=1.0)
audio_segment.duration
```

[8]:
```
1.0
```

5.2.4 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 properties can be accessed using the `samples` and `sample_rate` attributes, respectively.

[9]:
```
print(f"How many samples does this audio object have? {len(audio_object.samples)}")
print(f"What is the sampling rate? {audio_object.sample_rate}")
```

```
How many samples does this audio object have? 1920000
What is the sampling rate? 32000
```

5.2.5 Resample audio during load

By default, an audio object is loaded with the same sample rate as the source recording.

The `sample_rate` parameter of `Audio.from_file` allows you to re-sample the file during the creation of the object. This is useful when working with multiple files to ensure that all files have a consistent sampling rate.

Let's load the same audio file as above, but specify a sampling rate of 22050 Hz.

[10]:
```
audio_object_resample = Audio.from_file(audio_filename, sample_rate=22050)
audio_object_resample.sample_rate
```

[10]:
```
22050
```

5.2.6 Load audio from a specific real-world time from AudioMoth recordings

OpenSoundscape parses metadata of files recorded on AudioMoth recorders, and can use the metadata to extract pieces of audio corresponding to specific real-world times. (Note that AudioMoth internal clocks can drift an estimated 10-60 seconds per month).

[11]:
```
from datetime import datetime; import pytz

start_time = pytz.timezone('UTC').localize(datetime(2020,4,4,10,25))
audio_length = 5 #seconds
path = '/path/to/audiomoth_recording.WAV' #an AudioMoth recording

#this line is commented because it will fail unless you specify a valid path in the␣
↪line above
# Audio.from_file(path, start_timestamp=start_time,duration=audio_length)
```

For other options when loading audio objects, see the Audio.from_file() documentation.

5.3 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. Various examples are shown below.

For a description of the entire `Audio` object API, see the API documentation.

5.3.1 NOTE: Out-of-place operations

Functions that modify `Audio` (and `Spectrogram`) objects are "out of place", meaning that they return a new, modified instance of `Audio` instead of modifying the original instance. This means that running a line

```
audio_object.resample(22050) # WRONG!
```

will **not** change the sample rate of `audio_object`! If your goal was to overwrite `audio_object` with the new, resampled audio, you would instead write

```
audio_object = audio_object.resample(22050)
```

5.3.2 Save audio to file

Opensoundscape currently supports saving Audio objects to `.wav` formats **only**. It does **not** currently support saving metadata (tags) along with wav files - only the samples and sample rate will be preserved in the file.

[12]:
```
audio_object.save('./my_audio.wav')
```

clean up: delete saved file

[13]:
```
from pathlib import Path
Path('./my_audio.wav').unlink()
```

5.3.3 Get duration

The `.duration` property returns the length of the audio in seconds

[14]:
```
length = audio_object.duration
print(length)
```

```
60.0
```

5.3.4 Trim

The `.trim()` method extracts audio from a specified time period in seconds (relative to the start of the audio object).

```
[15]: trimmed = audio_object.trim(0,5)
      trimmed.duration
```

```
[15]: 5.0
```

5.3.5 Split Audio into clips

The `.split()` method divides audio into even-lengthed clips, optionally with overlap between adjacent clips (default is no overlap). See the function's documentation for options on how to handle the last clip.

The function returns a list containing Audio objects for each clip and a DataFrame giving the start and end times of each clip with respect to the original file.

split

```
[16]: #split into 5-second clips with no overlap between adjacent clips
      clips, clip_df = audio_object.split(clip_duration=5,clip_overlap=0,final_clip=None)

      #check the duration of the Audio object in the first returned element
      print(f"duration of first clip: {clips[0].duration}")

      print(f"head of clip_df")
      clip_df.head(3)
```

```
duration of first clip: 5.0
head of clip_df
```

```
[16]:    start_time  end_time
      0         0.0       5.0
      1         5.0      10.0
      2        10.0      15.0
```

split with overlap

if we want overlap between consecutive clips

Note that a negative "overlap" value would leave *gaps* between consecutive clips.

```
[17]: _, clip_df = audio_object.split(clip_duration=5,clip_overlap=2.5,final_clip=None)
      print(f"head of clip_df")
      clip_df.head()
```

```
head of clip_df
```

```
[17]:    start_time  end_time
      0         0.0       5.0
      1         2.5       7.5
      2         5.0      10.0
```

(continues on next page)

(continued from previous page)

```
3          7.5       12.5
4         10.0       15.0
```

split and save

The `Audio.split_and_save()` method splits audio into clips and immediately saves them to files in a specified location. You provide it with a naming prefix, and it will add on a suffix indicating the start and end times of the clip (eg `_5.0-10.0s.wav`). It returns just a DataFrame with the paths and start/end times for each clip (it does not return Audio objects).

The splitting options are the same as `.split()`: clip_duration, clip_overlap, and final_clip

[18]:
```
#split into 5-second clips with no overlap between adjacent clips
Path('./temp_audio').mkdir(exist_ok=True)
clip_df = audio_object.split_and_save(
    destination='./temp_audio',
    prefix='audio_clip_',
    clip_duration=5,
    clip_overlap=0,
    final_clip=None
)

print(f"head of clip_df")
clip_df.head()
```

```
head of clip_df
```

[18]:
```
                                          start_time  end_time
file
./temp_audio/audio_clip__0.0s_5.0s.wav           0.0       5.0
./temp_audio/audio_clip__5.0s_10.0s.wav          5.0      10.0
./temp_audio/audio_clip__10.0s_15.0s.wav        10.0      15.0
./temp_audio/audio_clip__15.0s_20.0s.wav        15.0      20.0
./temp_audio/audio_clip__20.0s_25.0s.wav        20.0      25.0
```

The folder `temp_audio` should now contain 12 5-second clips created from the 60-second audio file.

clean up: delete temp folder of saved audio clips

[19]:
```
from shutil import rmtree
rmtree('./temp_audio')
```

split_and_save dry run

we can use the `dry_run=True` option to produce only the clip_df but not actually process the audio. this is useful as a quick test to see if the function is behaving as expected, before doing any (potentially slow) splitting on huge audio files.

Just for fun, we'll use an overlap of -5 in this example (5 second gap between each consecutive clip)

This function returns a DataFrame of clips, but does not actually process the audio files or write any new files.

[20]:
```
clip_df = audio_object.split_and_save(
    destination='./temp_audio',
    prefix='audio_clip_',
```

(continues on next page)

(continued from previous page)

```
    clip_duration=5,
    clip_overlap=-5,
    final_clip=None,
    dry_run=True,
)
clip_df
```

[20]:

```
                                           start_time  end_time
file
./temp_audio/audio_clip__0.0s_5.0s.wav           0.0       5.0
./temp_audio/audio_clip__10.0s_15.0s.wav        10.0      15.0
./temp_audio/audio_clip__20.0s_25.0s.wav        20.0      25.0
./temp_audio/audio_clip__30.0s_35.0s.wav        30.0      35.0
./temp_audio/audio_clip__40.0s_45.0s.wav        40.0      45.0
./temp_audio/audio_clip__50.0s_55.0s.wav        50.0      55.0
```

5.3.6 Extend and loop

The `.extend()` method extends an audio file to a desired length by adding silence to the end.

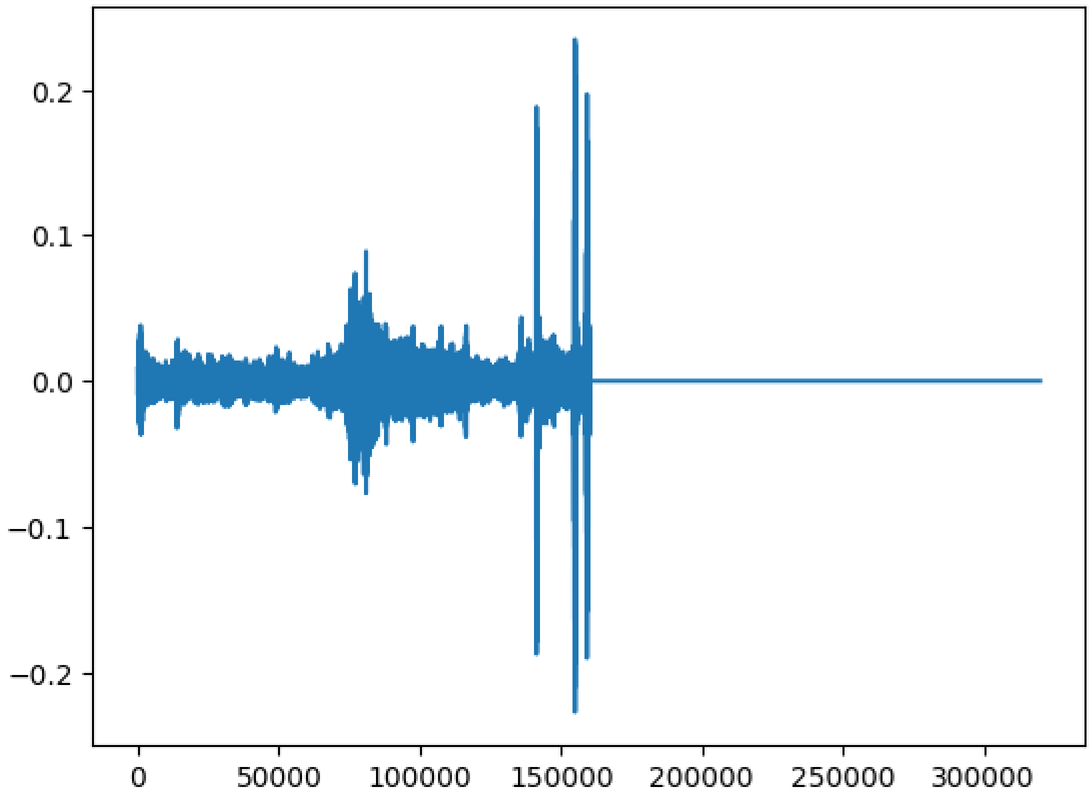
The `.loop()` method extends an audio file to a desired length (or number of repetitions) by looping the audio.

extend() example: create an Audio object twice as long as the original, extending with silence (0 valued samples)

[21]:

```
import matplotlib.pyplot as plt

# create an audio object twice as long, extending the end with silence (zero-values)
extended = trimmed.extend(trimmed.duration * 2)

print(f"duration of original clip: {trimmed.duration}")
print(f"duration of extended clip: {extended.duration}")
print(f"samples of extended clip:")
plt.plot(extended.samples)
plt.show()
```

```
duration of original clip: 5.0
duration of extended clip: 10.0
samples of extended clip:
```

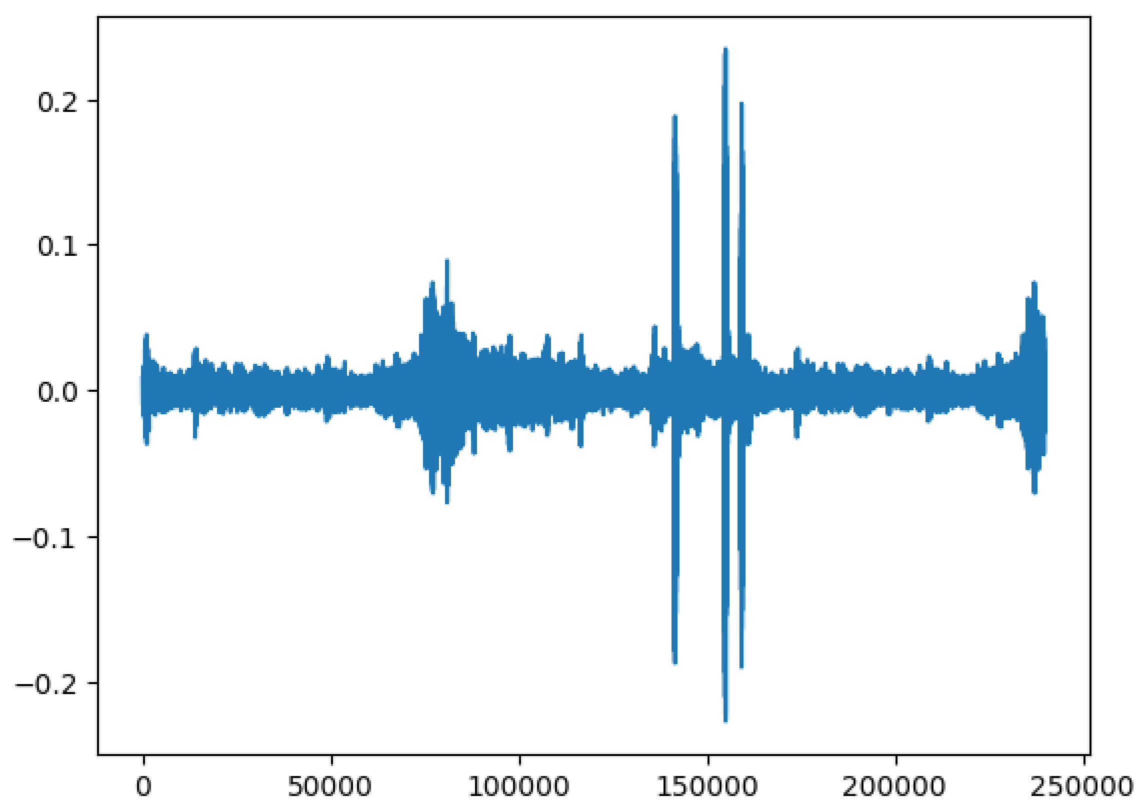


Looping example: create an audio object 1.5x as long, extending the end by looping

[22]:
```
looped = trimmed.loop(trimmed.duration * 1.5)
print(looped.duration)
plt.plot(looped.samples)
```

```
7.5
```

[22]:
```
[<matplotlib.lines.Line2D at 0x2915b5040>]
```

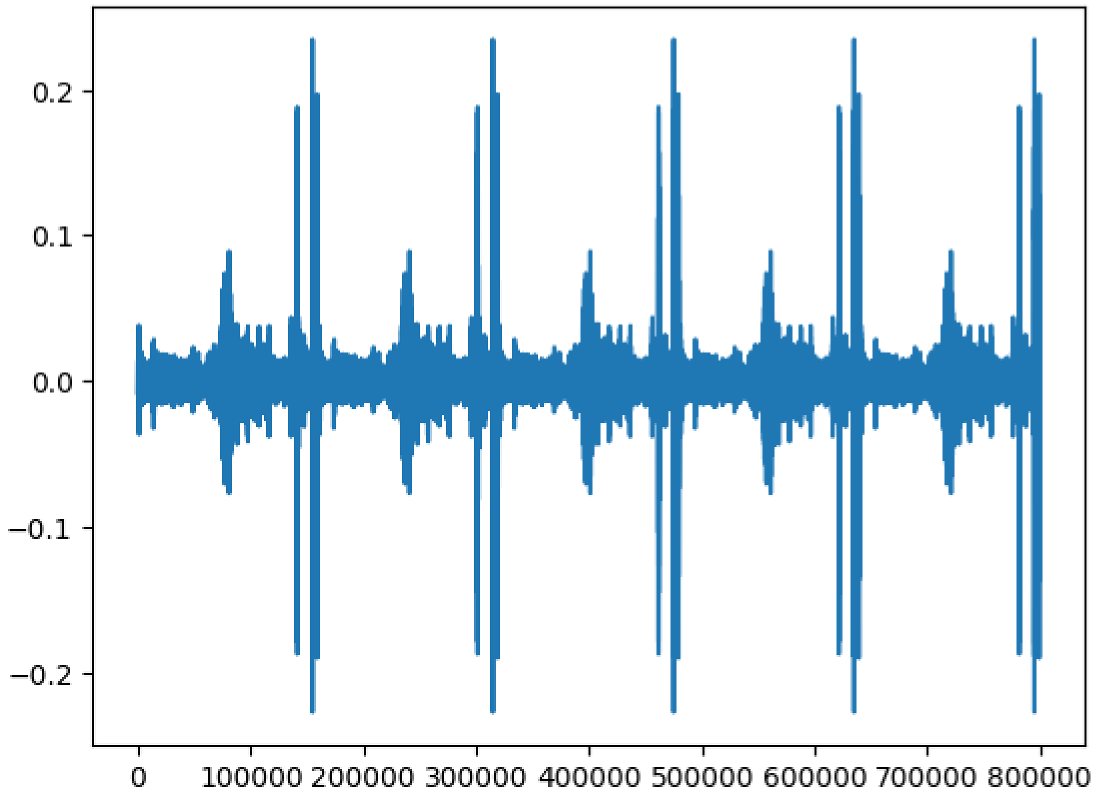


create an audio object that loops the original object 5 times and plot the samples

[23]:
```
looped = trimmed.loop(n=5)
print(looped.duration)
plt.plot(looped.samples)
```

```
25.0
```

[23]:
```
[<matplotlib.lines.Line2D at 0x291b3cf70>]
```



5.3.7 Resample

The `.resample()` method resamples the audio object to a new sampling rate (can be lower or higher than the original sampling rate)

[24]:
```
resampled = trimmed.resample(sample_rate=48000)
resampled.sample_rate
```

[24]:
```
48000
```

5.3.8 Generate a frequency spectrum

The `.spectrum()` method provides an easy way to compute a Fourier Transform on an audio object to measure its frequency composition.

[25]:
```
# calculate the fft
fft_spectrum, frequencies = trimmed.spectrum()

#plot settings
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for big visuals
%config InlineBackend.figure_format = 'retina'

# plot
plt.plot(frequencies,fft_spectrum)
```

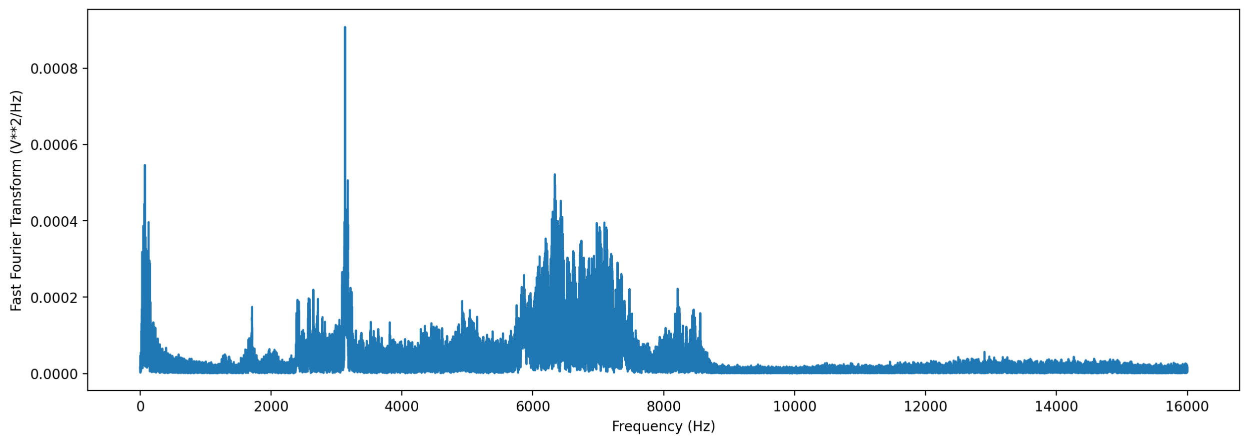
(continues on next page)

(continued from previous page)

```
plt.ylabel('Fast Fourier Transform (V**2/Hz)')
plt.xlabel('Frequency (Hz)')
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/matplotlib_inline/
↪config.py:68: DeprecationWarning: InlineBackend._figure_format_changed is␣
↪deprecated in traitlets 4.1: use @observe and @unobserve instead.
  def _figure_format_changed(self, name, old, new):
```

```
[25]: Text(0.5, 0, 'Frequency (Hz)')
```



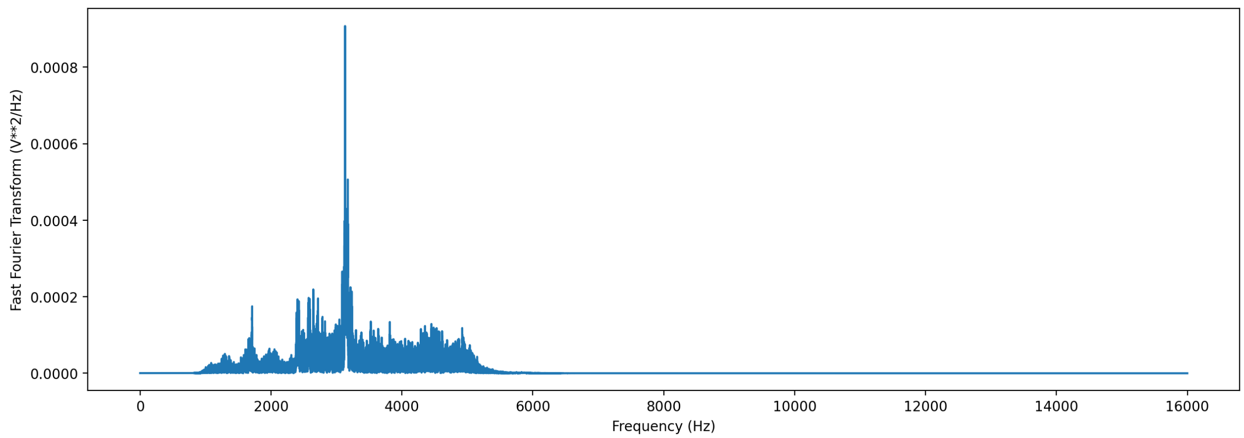
5.3.9 Bandpass

Bandpass the audio file to limit its frequency range to 1000 Hz to 5000 Hz. The bandpass operation uses a Butterworth filter with a user-provided order.

```
[26]: # apply a bandpass filter
bandpassed = trimmed.bandpass(low_f = 1000, high_f = 5000, order=9)

# calculate the bandpassed audio's spectrum
fft_spectrum, frequencies = bandpassed.spectrum()

# plot
print('spectrum after bandpassing the audio:')
plt.plot(frequencies,fft_spectrum)
plt.ylabel('Fast Fourier Transform (V**2/Hz)')
plt.xlabel('Frequency (Hz)')
```

```
spectrum after bandpassing the audio:
```

```
[26]: Text(0.5, 0, 'Frequency (Hz)')
```



5.4 Spectrogram creation

5.4.1 Load spectrogram

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

[27]:
```
audio_object = Audio.from_file(audio_filename)
spectrogram_object = Spectrogram.from_audio(audio_object)
```

5.4.2 Spectrogram properties

To check the time and frequency axes of a spectrogram, you can look at its `times` and `frequencies` attributes. The `times` attribute is the list of the spectrogram windows' centers' times in seconds relative to the beginning of the audio. The `frequencies` attribute is the list of frequencies represented by each row of the spectrogram. These are not the actual values of the spectrogram — just the values of the axes.

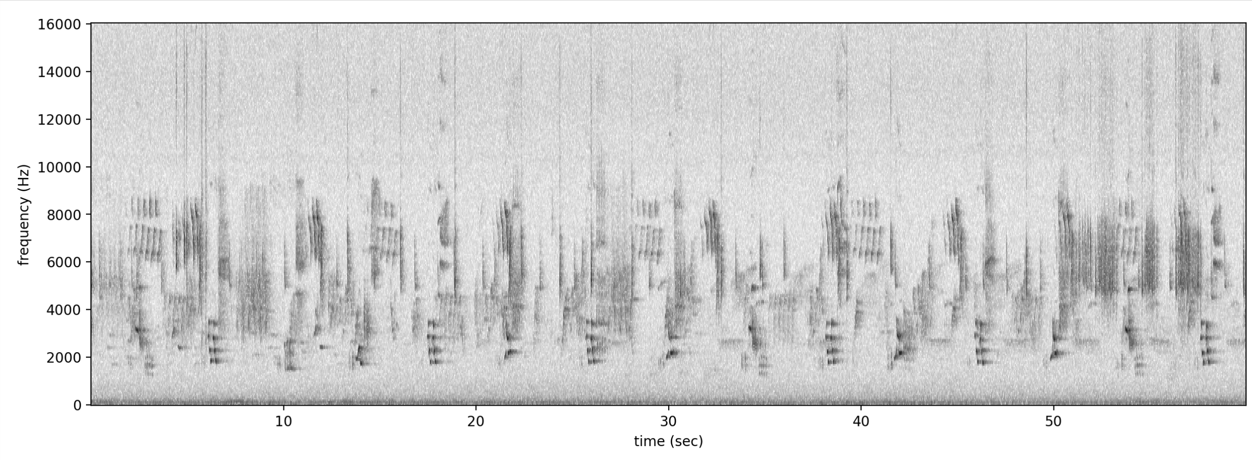
[28]:
```
spec = Spectrogram.from_audio(Audio.from_file(audio_filename))
print(f'the first few times: {spec.times[0:5]}')
print(f'the first few frequencies: {spec.frequencies[0:5]}')
```

```
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. ]
```

5.4.3 Plot spectrogram

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

[29]:
```
audio_object = Audio.from_file(audio_filename)
spectrogram_object = Spectrogram.from_audio(audio_object)
spectrogram_object.plot()
```



5.4.4 Spectrogram parameters

Spectrograms are created using "windows". A window is a subset of consecutive samples of the original audio that is analyzed to create one pixel in the horizontal direction (one "column") on the resulting spectrogram. The appearance of a spectrogram depends on two parameters that control the size and spacing of these windows:

Samples per window, `window_samples`

This parameter is the length (in audio samples) of each spectrogram window. Choosing the value for `window_samples` represents a trade-off between frequency resolution and time resolution:

- Larger value for `window_samples` –> higher frequency resolution (more rows in a single spectrogram column)
- Smaller value for `window_samples` –> higher time resolution (more columns in the spectrogram per second)

Overlap of consecutive windows, `overlap_samples`

`overlap_samples`: this is the number of audio samples that will be re-used (overlap) between two consecutive Specrogram windows. It must be less than `window_samples` and greater than or equal to zero. Zero means no overlap between windows, while a value of `window_samples`/2 would give 50% overlap between consecutive windows. Using higher overlap percentages can sometimes yield better time resolution in a spectrogram, but will take more computational time to generate.

Spectrogram parameter tradeoffs

When there is zero overlap between windows, the number of columns per second is equal to the size in Hz of each spectrogram row. Consider the relationship between time resolution (columns in the spectrogram per second) and frequency resolution (rows in a given frequency range) in the following example:

- Let `sample_rate=48000`, `window_samples=480`, and `overlap_samples=0`
- Each window ("spectrogram column") represents `480/48000 = 1/100 = 0.01` seconds of audio
- There will be `1/(length of window in seconds) = 1/0.01` = 100 columns in the spectrogram per second.

- Each pixel will span 100 Hz in the frequency dimension, i.e., the lowest pixel spans 0-100 Hz, the next lowest 100-200 Hz, then 200-300 Hz, etc.

If `window_samples=4800`, then the spectrogram would have better time resolution (each window represents only 4800/48000 = 0.001s of audio) but worse frequency resolution (each row of the spectrogram would represent 1000 Hz in the frequency range).

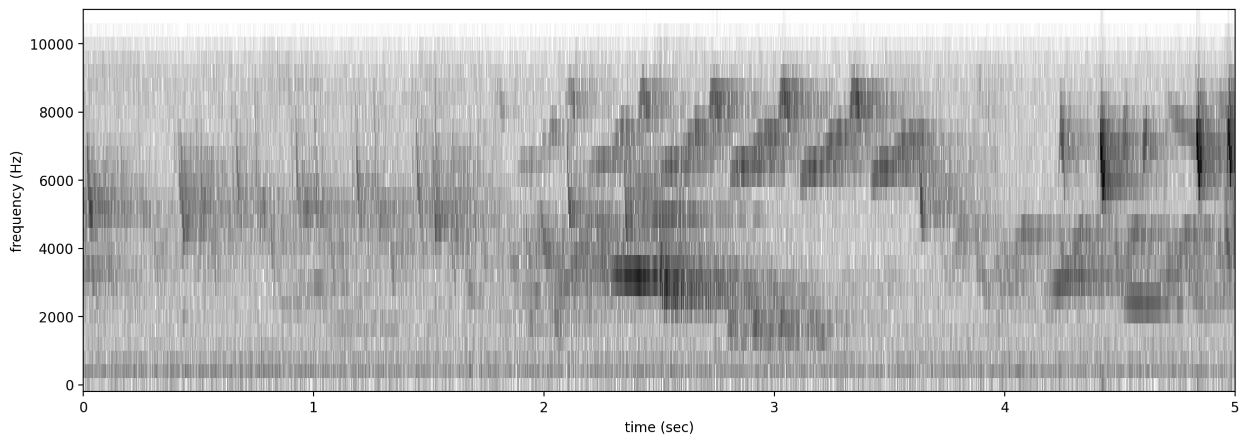
As an example, let's create two spectrograms, one with hight time resolution and another with high frequency resolution.

[30]:
```
# Load audio
audio = Audio.from_file(audio_filename, sample_rate=22000).trim(0,5)
```

Create a spectrogram with high time resolution

Using `window_samples=55` and `overlap_samples=0` gives 55/22000 = 0.0025 seconds of audio per window, or 1/0.0025 = 400 windows per second. Each spectrogram pixel spans 400 Hz.

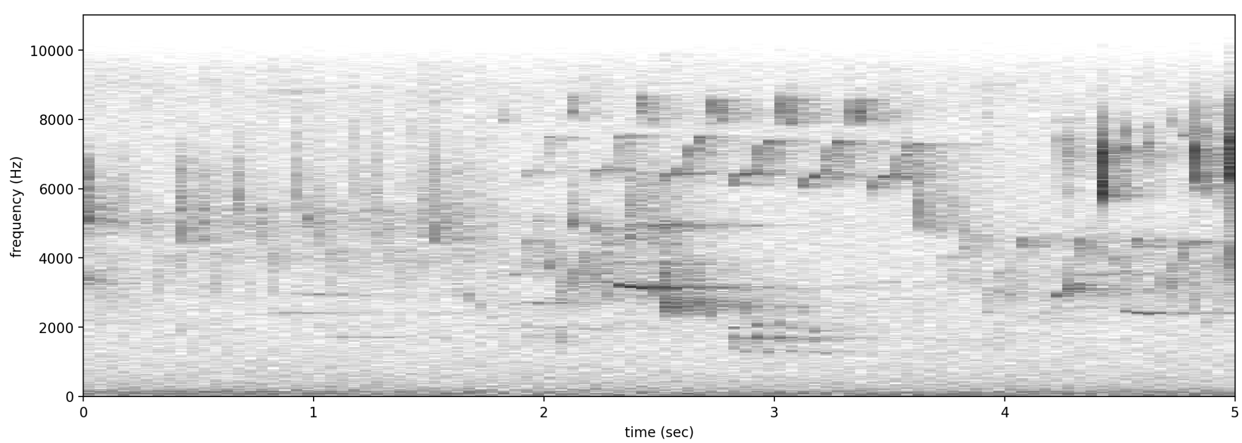
[31]:
```
spec = Spectrogram.from_audio(audio, window_samples=55, overlap_samples=0)
spec.plot()
```



Create a spectrogram with high frequency resolution

Using `window_samples=1100` and `overlap_samples=0` gives 1100/22000 = 0.05 seconds of audio per window, or 1/0.05 = 20 windows per second. Each spectrogram pixel spans 20 Hz.

[32]:
```
spec = Spectrogram.from_audio(audio, window_samples=1100, overlap_samples=0)
spec.plot()
```



For other options when loading spectrogram objects from audio objects, see the `from_audio()` documentation.

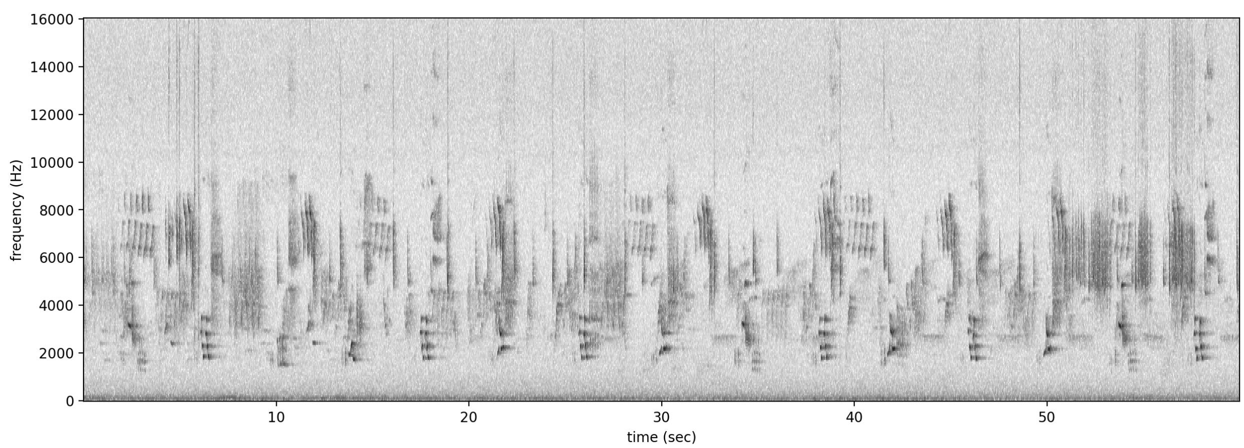
5.5 Spectrogram methods

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.

5.5.1 Plot

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

[33]:
```
audio_object = Audio.from_file(audio_filename)
spectrogram_object = Spectrogram.from_audio(audio_object)
spectrogram_object.plot()
```

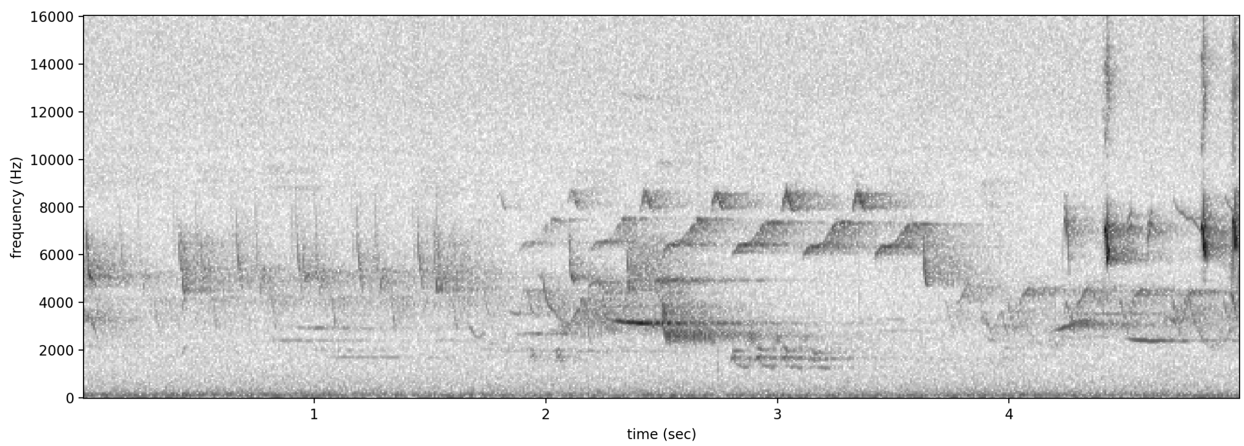


5.5.2 Load modified audio

Sometimes, you may wish to trim or modify an audio object before creating a spectrogram. In this case, you should first modify the Audio object, then call `Spectrogram.from_audio()`.

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.

[34]:
```
# 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()
```



5.5.3 Save spectrogram to file

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.

[35]:
```
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.

[36]:
```
image_path = 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`.

[37]:
```
image_shape = (512,512)
large_image_path = Path('./saved_spectrogram_large.png')
spectrogram_image = spectrogram_object.to_image(shape=image_shape)
spectrogram_image.save(large_image_path)
```

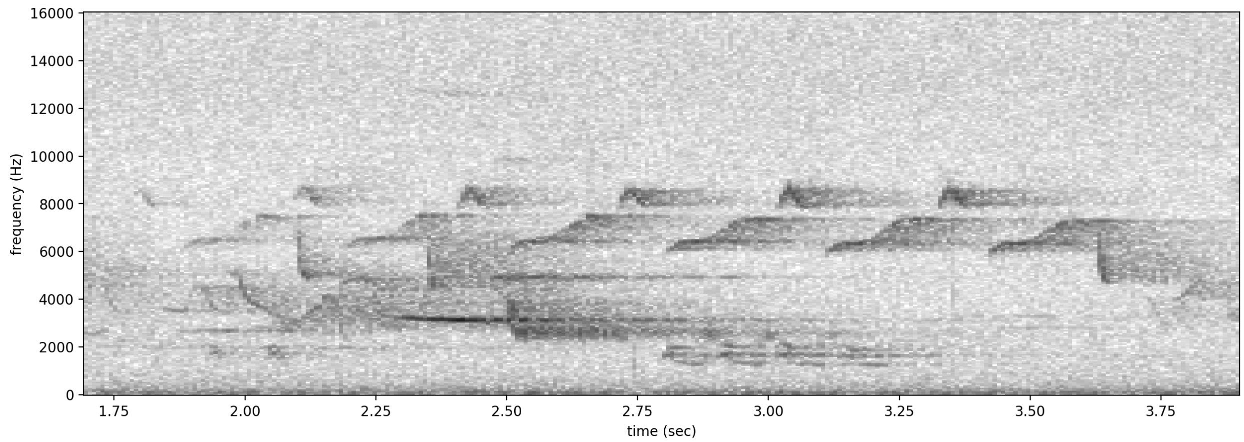
Delete the files created above.

[38]:
```
image_path.unlink()
large_image_path.unlink()
```

5.5.4 Trim

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

[39]:
```
spec_trimmed = spec.trim(1.7, 3.9)
spec_trimmed.plot()
```



5.5.5 Bandpass

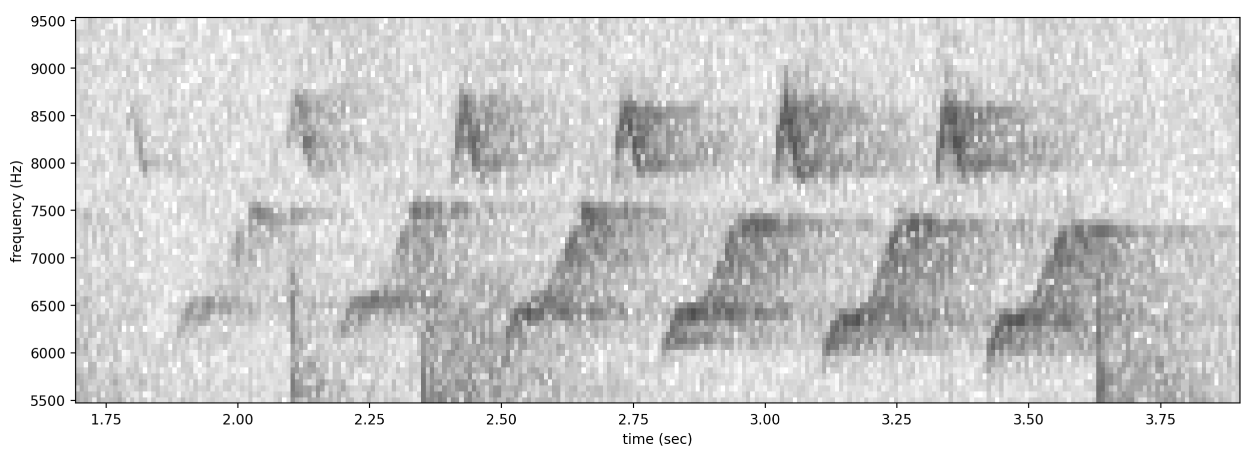
Spectrograms can be trimmed in frequency using `bandpass()`. This simply subsets the Spectrogram array rather than performing an audio-domain filter.

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.

[40]:
```
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.

[41]:
```
spec_bandpassed = spec_trimmed.bandpass(baww_low_freq, baww_high_freq)
spec_bandpassed.plot()
```

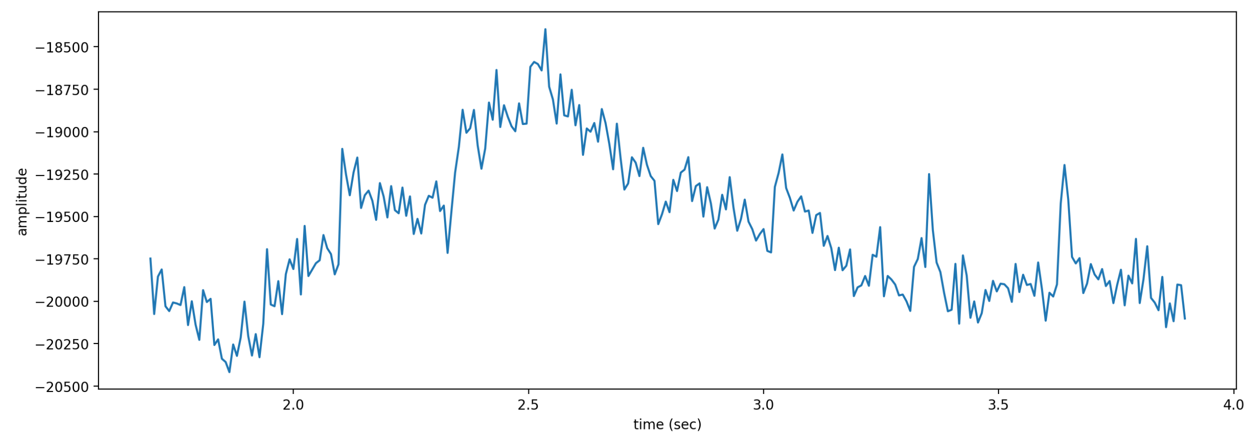


5.5.6 Sum the columns of a spectrogram

The `.amplitude()` method sums the columns of the spectrogram to create a one-dimensional amplitude versus time vector.

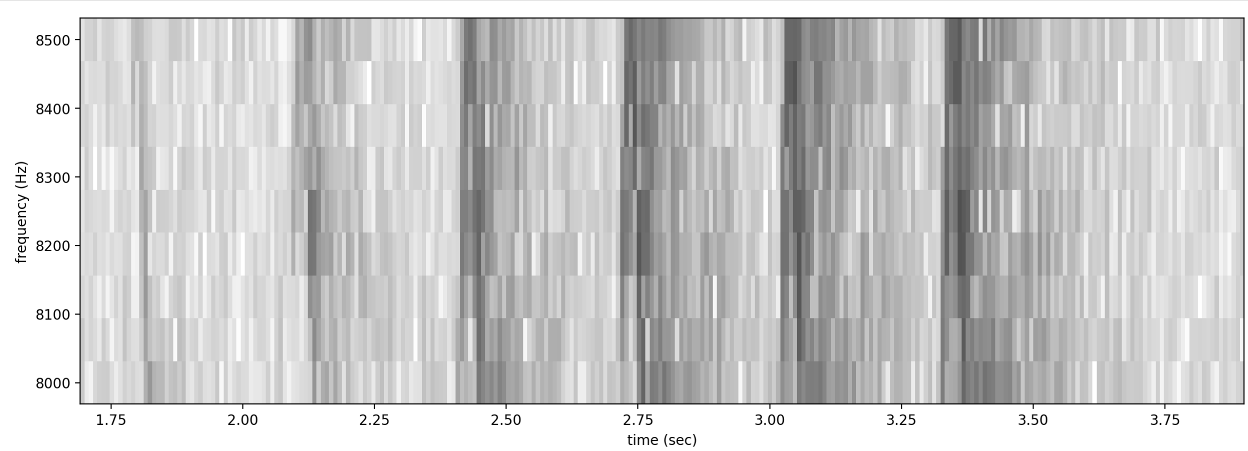
Note: the amplitude of the Spectrogram (and FFT) has units of power (V**2) over frequency (Hz) on a logarithmic scale

[42]:
```
# calculate amplitude signal
high_freq_amplitude = spec_trimmed.amplitude()

# plot
from matplotlib import pyplot as plt
plt.plot(spec_trimmed.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, for instance, to look at the amplitude in the frequency range of a species of interest. For example, get the amplitude signal from the 8000 Hz to 8500 Hz range of the audio (displayed below):

[43]:
```
spec_bandpassed = spec_trimmed.bandpass(8000, 8500)
spec_bandpassed.plot()
```



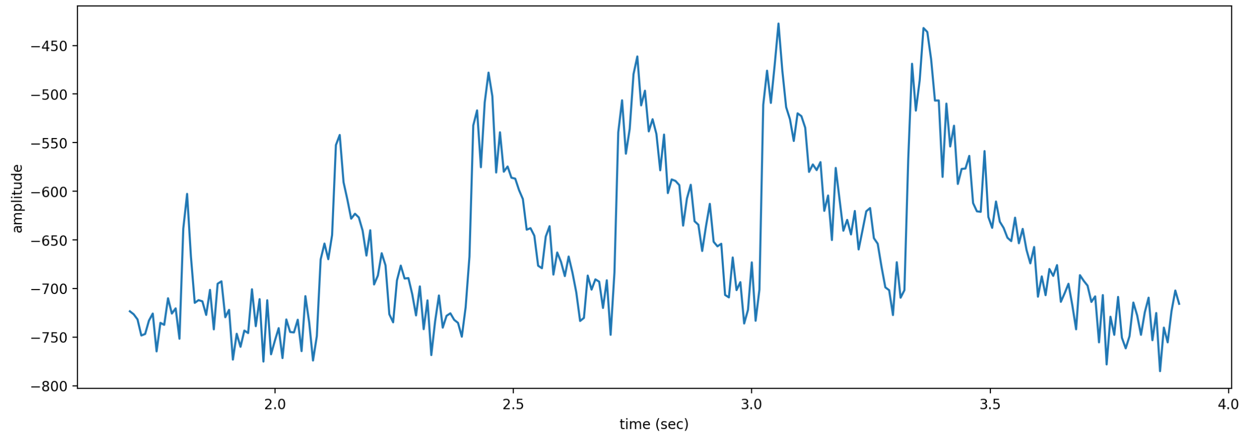
Get and plot the amplitude signal of only 8-8.5 kHz.

[44]:
```
# Get amplitude signal
high_freq_amplitude = spec_trimmed.amplitude(freq_range=[8000,8500])

# Get amplitude signal
```

(continues on next page)

(continued from previous page)

```
high_freq_amplitude = spec_trimmed.amplitude(freq_range=[8000,8500])

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



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.

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 with pre-trained CNNs tutorial.

5.5.7 clean up

```
[45]: #delete the file we downloaded for the tutorial
Path('1min_audio.wav').unlink()
```

CHAPTER 6

Manipulating audio annotations

This notebook demonstrates how to use the `annotations` module of OpenSoundscape to:

- load annotations from Raven files
- create a set of one-hot labels corresponding to fixed-length audio clips
- split a set of labeled audio files into clips and create labels dataframe for all clips

The audio recordings used in thise notebook were recorded by Andrew Spencer and are available under a Creative Commons License (CC BY-NC-ND 2.5) from xeno-canto.org. Annotations were performed in Raven Pro software by our team.

[1]:
```
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
from opensoundscape.annotations import BoxedAnnotations

import numpy as np
import pandas as pd
from glob import glob

from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for big visuals
%config InlineBackend.figure_format = 'retina'
```

6.1 Download example files

Run the code below to download a set of example audio and raven annotations files for this tutorial.

[2]:
```
import subprocess
subprocess.run(['curl','https://pitt.box.com/shared/static/
↪nzdzwwmyr3tkr6ig6sltw4b7jg3ptfe4.gz','-L', '-o','gwwa_audio_and_raven_annotations.
↪tar.gz']) # Download the data
subprocess.run(["tar","-xzf", "gwwa_audio_and_raven_annotations.tar.gz"]) # Unzip the␣
↪downloaded tar.gz file
```

(continues on next page)

(continued from previous page)

```
subprocess.run(["rm", "gwwa_audio_and_raven_annotations.tar.gz"]) # Remove the file␣
↪after its contents are unzipped
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      5      0 --:--:--  0:00:01 --:--:--     5
100 5432k  100 5432k    0     0  2431k      0  0:00:02  0:00:02 --:--:-- 6413k
```

[2]:
```
CompletedProcess(args=['rm', 'gwwa_audio_and_raven_annotations.tar.gz'], returncode=0)
```

6.1.1 Load a single Raven annotation table from a txt file

We can use the BoxedAnnotation class's `from_raven_file` method to load a Raven .txt file into OpenSoundscape. This table contains the frequency and time limits of rectangular "boxes" representing each annotation that was created in Raven.

Note that we need to specify the name of the column containing annotations, since it can be named anything in Raven. The column will be renamed to "annotation".

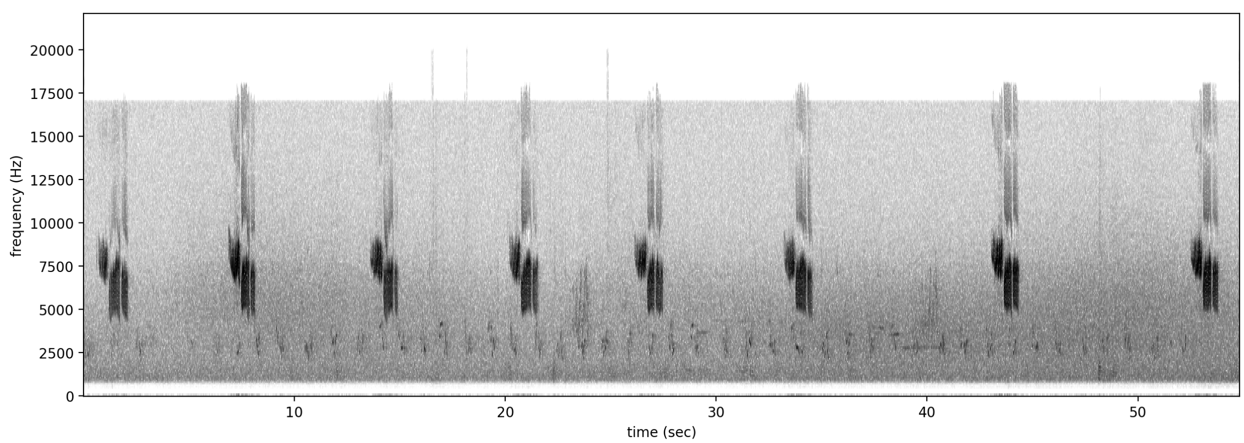
This table looks a lot like what you would see in the Raven interface.

[3]:
```
# specify an audio file and corresponding raven annotation file
audio_file = './gwwa_audio_and_raven_annotations/GWWA_XC/13738.wav'
annotation_file = './gwwa_audio_and_raven_annotations/GWWA_XC_AnnoTables/13738.Table.
↪1.selections.txt'
```

Let's look at a spectrogram of the audio file to see what we're working with.

[4]:
```
Spectrogram.from_audio(Audio.from_file(audio_file)).plot()
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/matplotlib_inline/
↪config.py:68: DeprecationWarning: InlineBackend._figure_format_changed is␣
↪deprecated in traitlets 4.1: use @observe and @unobserve instead.
  def _figure_format_changed(self, name, old, new):
```



Now, let's load the annotations from the Raven annotation file.

[5]:
```
#create an object from Raven file
annotations = BoxedAnnotations.from_raven_file(annotation_file,annotation_column=
↪'Species')
```

(continues on next page)

(continued from previous page)

```
#inspect the object's .df attribute, which contains the table of annotations
annotations.df.head()
```

```
[5]:    Selection           View  Channel  start_time   end_time   low_f   high_f  \
0           1  Spectrogram 1        1    0.459636   2.298182  4029.8  17006.4
1           2  Spectrogram 1        1    6.705283   8.246417  4156.6  17031.7
2           3  Spectrogram 1        1   13.464641  15.005775  3903.1  17082.4
3           4  Spectrogram 1        1   20.128208  21.601748  4055.2  16930.3
4           5  Spectrogram 1        1   26.047590  27.521131  4207.2  17057.1

  annotation  Notes
0  GWWA_song    NaN
1  GWWA_song    NaN
2          ?    NaN
3  GWWA_song    NaN
4  GWWA_song    NaN
```

Note: if you **do not have an annotation column**, e.g., if you are annotating the sounds of a single species, the function above doesn't work in the current version of OpenSoundscape. A workaround is to use `annotation_column='Channel'`, in which case, the "annotation" of your file will be equal to the channel number. If your recordings are single-channel, then the "class" of every annotation will just be the number 1.

We could instead choose to only load the necessary columns (`start_time`, `end_time`, `low_f`, `high_f`, and `annotation`) using the `keep_extra_columns=None`.

In this example, we use `keep_extra_columns=['Notes']` to keep only the Notes column.

```
[6]: annotations = BoxedAnnotations.from_raven_file(annotation_file,annotation_column=
↪'Species',keep_extra_columns=['Notes'])
annotations.df.head()
```

```
[6]:    start_time   end_time   low_f   high_f annotation  Notes
0    0.459636   2.298182  4029.8  17006.4  GWWA_song    NaN
1    6.705283   8.246417  4156.6  17031.7  GWWA_song    NaN
2   13.464641  15.005775  3903.1  17082.4          ?    NaN
3   20.128208  21.601748  4055.2  16930.3  GWWA_song    NaN
4   26.047590  27.521131  4207.2  17057.1  GWWA_song    NaN
```

6.1.2 Convert or correct annotations

We can provide a DataFrame (e.g., from a .csv file) or a dictionary to convert original values to new values.

Let's load up a little .csv file that specifies a set of conversions we'd like to make. The .csv file should have two columns, but it doesn't matter what they are called. If you create a table in Microsoft Excel, you can export it to a .csv file to use it as your conversion table.

```
[7]: conversion_table = pd.read_csv('./gwwa_audio_and_raven_annotations/conversion_table.
↪csv')
conversion_table
```

```
[7]:     original   new
0  gwwa_song  gwwa
```

Alternatively, we could simply write a Python dictionary for the conversion table. For instance:

[8]:
```
conversion_table = {
    "GWWA_song":"GWWA",
    "?":np.nan
}
```

Now, we can apply the conversions in the table to our annotations.

This will create a new BoxedAnnotations object rather than modifying the original object (an "out of place operation").

[9]:
```
annotations_corrected = annotations.convert_labels(conversion_table)
annotations_corrected.df
```

[9]:

	start_time	end_time	low_f	high_f	annotation	Notes
0	0.459636	2.298182	4029.8	17006.4	GWWA	NaN
1	6.705283	8.246417	4156.6	17031.7	GWWA	NaN
2	13.464641	15.005775	3903.1	17082.4	NaN	NaN
3	20.128208	21.601748	4055.2	16930.3	GWWA	NaN
4	26.047590	27.521131	4207.2	17057.1	GWWA	NaN
5	33.121946	34.663079	4207.2	17082.4	GWWA	NaN
6	42.967925	44.427946	4181.9	17057.1	GWWA	NaN
7	52.417508	53.891048	4232.6	16930.3	GWWA	NaN

6.2 View a subset of annotations

We can specify a list of classes to view annotations of.

For example, we can subset to only annotations marked as "?" - perhaps we're interested in looking at these annotations in Raven again to determine what class they really were.

[10]:
```
classes_to_keep = ['?']
annotations_only_unsure = annotations.subset(classes_to_keep)
annotations_only_unsure.df
```

[10]:

	start_time	end_time	low_f	high_f	annotation	Notes
2	13.464641	15.005775	3903.1	17082.4	?	NaN

6.3 Saving annotations to Raven-compatible file

We can always save our BoxedAnnotations object to a Raven-compatible .txt file, which can be opened in Raven along with an audio file just like the .txt files Raven creates itself. You must specify a path for the save file that ends with `.txt`.

[11]:
```
annotations_only_unsure.to_raven_file('./gwwa_audio_and_raven_annotations/13738_
↪unsure.txt')
```

6.3.1 Split an audio clip and its annotations

Often, we want to train or validate models on short audio segments (e.g., 5 seconds) rather than on long files (e.g., 2 hours).

To do this, we need two things:

- A saved set of short audio clips

- A table that associates each split audio clip with the annotations in the clip (e.g., which species, if any, are present in each clip). Usually these are in "one-hot encoding" form (see explanation below).

We can use OpenSoundscape to split annotations, and optionally, the audio clips associated with the annotations, in three ways:

Splitting both Audio and annotations:

1. Split the audio first using `Audio.split()`, then use the DataFrame of clip start/end times returned by this function to split the annotations using `BoxedAnnotations.one_hot_labels_like()`

Splitting annotations only:

2. Use `BoxedAnnotations.one_hot_clip_labels()` to split the annotations in one step
3. Create a DataFrame of clip start/end times similar to the one generated by `Audio.split()`, then use it to split the annotations using `BoxedAnnotations.one_hot_labels_like()`

All three methods are demonstrated below.

What is one-hot encoding?

The functions below demonstrate the creation of one-hot encoded labels.

This machine learning term, "one-hot encoding," refers to a way to format a table of labels in which: * Each row represents a single sample, like a single 5-second long clip * Each column represents a single possible class (e.g. one of multiple species) * A "0" in a row and column means that in that sample, the class is not present * A "1" is "hot," meaning that in that sample, the class *IS* present.

For example, let's say we had a 15-second audio clip that we were splitting into three 5s clips. Let's say we are training a classifier to identify coyotes and dogs, and we labeled the clip and found: * a coyote howled from 2.5 to 4 seconds into the clip (so, only the first clip contains it) * a dog barked from 4 seconds to 10 seconds into the clip (so, both the first and second clips contain it) * and there was silence for the last 5 seconds of the clip (so, the third clip has neither coyotes nor dogs in it).

The one-hot encoded labels file for this example would look like:

[12]:
```
pd.DataFrame({
    "start_time":[0, 5, 10],
    "end_time":[5, 10, 15],
    "COYOTE":[1, 0, 0],
    "DOG":[1, 1, 0]
})
```

[12]:

	start_time	end_time	COYOTE	DOG
0	0	5	1	1
1	5	10	0	1
2	10	15	0	0

6.4 1. Split Audio object, then split annotations to match

First, split an Audio object with `Audio.split()`, which returns two things:

1. A list of audio clip objects
2. A dataframe of start/end times

[13]:
```
# Load the Audio and Annotations
audio = Audio.from_file(audio_file)
annotations = BoxedAnnotations.from_raven_file(annotation_file, annotation_column=
↪'Species')
```

(continues on next page)

(continued from previous page)

```
# Split the audio into 5 second clips with no overlap (we use _ because we don't␣
↪really need to save the audio clip objects for this demo)
_, clip_df = audio.split(
    clip_duration=5.0, # How long each clip should be
    clip_overlap=0.0 # By how many seconds each subsequent clip should overlap
)
clip_df.head()
```

[13]:

```
   start_time  end_time
0         0.0       5.0
1         5.0      10.0
2        10.0      15.0
3        15.0      20.0
4        20.0      25.0
```

Different overlap and duration settings produce different results:

[14]:

```
# Split the audio into 5 second clips with no overlap (we use _ because we don't␣
↪really need to save the audio clip objects for this demo)
_, clip_df_short_with_overlaps = audio.split(
    clip_duration=3.0, # How long each clip should be
    clip_overlap=1.0 # By how many seconds each subsequent clip should overlap
)
clip_df_short_with_overlaps.head()
```

[14]:

```
   start_time  end_time
0         0.0       3.0
1         2.0       5.0
2         4.0       7.0
3         6.0       9.0
4         8.0      11.0
```

Next, extract annotations for each clip using `BoxedAnnotations.one_hot_labels_like()`.

This function requires that we specify the minimum overlap of the label (in seconds) with the clip for the clip to be labeled positive. It also requires that we either (1) specify the list of classes for one-hot labels or (2) specify `class_subset=None`, which will make a column for every unique label in the annotations. In this example, that would include a “?” class.

[15]:

```
# Split the annotations using the returned clip_df
labels_df = annotations.one_hot_labels_like(
    clip_df,
    min_label_overlap=0.25, # Minimum label overlap
    class_subset=['GWWA_song']
)

#the returned dataframe of one-hot labels (0/1 for each class and each clip) has rows␣
↪corresponding to each audio clip
labels_df.head()
```

[15]:

```
                     GWWA_song
start_time end_time
0.0        5.0             1.0
5.0        10.0            1.0
10.0       15.0            0.0
15.0       20.0            0.0
20.0       25.0            1.0
```

6.5 2. Split annotations directly using splitting parameters

If you prefer to split only annotations, you can do this using `one_hot_clip_labels()`.

This method combines the two steps in the example above (creating a clip dataframe and splitting the annotations), and requires that you specify the parameters for both of those steps.

Notice that we can't tell what the length of the entire audio file is from the annotation file alone, so we need to specify one additional parameter: the entire duration of the audio file to be split (`full_duration`).

Here's an example that produces equivalent results to the other examples:

[17]:
```
labels_df = annotations.one_hot_clip_labels(
    full_duration=60, # The duration of the entire audio file
    clip_duration=5,
    clip_overlap=0,
    class_subset=['GWWA_song'],
    min_label_overlap=0.25,
)
labels_df.head()
```

[17]:
```
                      GWWA_song
start_time end_time
0          5                1.0
5          10               1.0
10         15               0.0
15         20               0.0
20         25               1.0
```

6.6 3. Split annotations using your own clip DF

A more verbose option than #2: we can split annotations using a separately created DataFrame of start and end times.

This method could be useful if you wished to hand-create the DataFrame of clip start and end times to have more control over the start and end times you were interested in.

In this example, we will use a helper function to create the DataFrame, `generate_clip_times_df()`, which takes the same splitting parameters as `Audio.split()`.

[18]:
```
# Generate clip start/end time DataFrame
from opensoundscape.helpers import generate_clip_times_df
clip_df = generate_clip_times_df(full_duration=60, clip_duration=5.0, clip_overlap=0.
↪0)
clip_df.head()
```

[18]:
```
   start_time  end_time
0         0.0       5.0
1         5.0      10.0
2        10.0      15.0
3        15.0      20.0
4        20.0      25.0
```

[20]:
```
# We can use the clip_df to split the Annotations in the same way as before
labels_df = annotations.one_hot_labels_like(clip_df, min_label_overlap=0.25, class_
↪subset=['GWWA_song'])
```

(continues on next page)

(continued from previous page)

```
# The returned dataframe of one-hot labels (0/1 for each class and each clip) has␣
↪rows corresponding to each audio clip
labels_df.head()
```

[20]:

```
                      GWWA_song
start_time end_time
0.0        5.0              1.0
5.0        10.0             1.0
10.0       15.0             0.0
15.0       20.0             0.0
20.0       25.0             1.0
```

6.6.1 Split many audio clips and their annotations

The steps above described how to split a single audio clip and its annotations.

In practice, we have tons of audio files with their corresponding Raven files. We need to:

- Pair up all the audio files with their Raven annotation files
- Split and save short audio clips
- Split and save the annotations corresponding to the audio clips

Let's walk through the steps required to do this. But be warned, pairing Raven files and audio files might require more finagling than shown here.

6.7 Match up audio files and Raven annotations

The first step in the process is associating audio files with their corresponding Raven files. Perhaps not every audio file is annotated, and perhaps some audio files have been annotated multiple times. This code walks through some steps of sorting through these data to pair files.

Caveat: you'll need to be careful using the code below, depending on how your audio and Raven files are named and organized.

In this example, we'll assume that each audio file has the same name as its Raven annotation file (ignoring the extensions like ".Table.1.selections.txt"), which is the default naming convention when using Raven. We'll also start by assuming that the audio filenames *are unique (!)* - that is, no two audio files have the same name.

First, find all the Raven files and all the audio files.

[21]:

```
# Specify folder containing Raven annotations
raven_files_dir = "./gwwa_audio_and_raven_annotations/GWWA_XC_AnnoTables/"

# Find all .txt files
# We'll naively assume all files with the suffix ".txt" are Raven files!
# A better assumption could be to search for files with the suffix ".selections.txt"
raven_files = glob(f"{raven_files_dir}/*.txt")
print(f"found {len(raven_files)} annotation files")

# Specify folder containing audio files
audio_files_dir = "./gwwa_audio_and_raven_annotations/GWWA_XC/"

# Find all audio files (we'll assume they are .wav, .WAV, or .mp3)
```

(continues on next page)

(continued from previous page)

```
audio_files = glob(f"{audio_files_dir}/*.wav")+glob(f"{audio_files_dir}/*.WAV")+glob(f
↪"{audio_files_dir}/*.mp3")
print(f"found {len(audio_files)} audio files")
```

```
found 3 annotation files
found 3 audio files
```

Next, starting by assuming that audio files have unique names, use the audio filenames to pair up the annotation files. Then, double-check that our assumption is correct.

[22]:
```
# Pair up the Raven and audio files based on the audio file name
from pathlib import Path
audio_df = pd.DataFrame({'audio_file':audio_files})
audio_df.index = [Path(f).stem for f in audio_files]

# Check that there aren't duplicate audio file names
print('\n audio files with duplicate names:')
audio_df[audio_df.index.duplicated(keep=False)]
```

```

 audio files with duplicate names:
```

[22]:
```
Empty DataFrame
Columns: [audio_file]
Index: []
```

Seeing that no audio files have duplicate names, check to make sure the same is true for Raven files.

[23]:
```
raven_df = pd.DataFrame({'raven_file':raven_files})
raven_df.index = [Path(f).stem.split('.Table')[0] for f in raven_files]

#check that there aren't duplicate audio file names
print('\n raven files with duplicate names:')
raven_df[raven_df.index.duplicated(keep=False)]
```

```

 raven files with duplicate names:
```

[23]:
```
                                                raven_file
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
```

Since we found some duplicate Raven files, resolve this issue by deleting the extra Raven file, which in this case was named "selections2".

[24]:
```
#remove the second selection table for file 13738.wav
raven_df=raven_df[raven_df.raven_file.apply(lambda x: "selections2" not in x)]
```

Once we've resolved any issues with duplicate names, we can match up Raven and audio files.

[25]:
```
paired_df = audio_df.join(raven_df,how='outer')
```

Check if any audio files don't have Raven annotation files:

[26]:
```
print(f"audio files without raven file: {len(paired_df[paired_df.raven_file.
↪apply(lambda x:x!=x)])}")
paired_df[paired_df.raven_file.apply(lambda x:x!=x)]
```

```
audio files without raven file: 2
```

[26]:
```
                                              audio_file raven_file
135601  ./gwwa_audio_and_raven_annotations/GWWA_XC/135...        NaN
13742   ./gwwa_audio_and_raven_annotations/GWWA_XC/137...        NaN
```

Check if any Raven files don't have audio files:

[27]:
```
#look at unmatched raven files
print(f"raven files without audio file: {len(paired_df[paired_df.audio_file.
↪apply(lambda x:x!=x)])}")

paired_df[paired_df.audio_file.apply(lambda x:x!=x)]
```

```
raven files without audio file: 1
```

[27]:
```
      audio_file                                         raven_file
16989        NaN  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
```

In this example, let's discard any unpaired Raven or audio files.

[28]:
```
paired_df = paired_df.dropna()
```

[29]:
```
paired_df
```

[29]:
```
                                             audio_file  \
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC/137...

                                             raven_file
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
```

6.8 Split and save the audio and annotations

Now we have a set of paired up Raven and audio files.

Let's split each of the audio files and create the corresponding labels.

Note: this step will be confusing and annoying if your Raven files use different names for the annotation column. Ideally, all of your Raven files should have the same column name for the annotations.

First, make a directory to put the split audio files in.

[30]:
```
clip_dir = './temp_clips'
Path(clip_dir).mkdir(exist_ok=True)
```

Next, set up the settings for audio splitting: * The duration of the clips

- Whether subsequent clips should overlap each other (e.g., `clip_overlap=1` would mean that the second clip started 1s before the first clip ended)
- What to do with the final clip if it would be less than `clip_duration` size (see API documentation for full information about the options for this)
- And the directory in which to save audio files.

[31]:
```
# Choose settings for audio splitting
clip_duration = 3
clip_overlap = 0
```

(continues on next page)

(continued from previous page)

```
final_clip = None
clip_dir = './temp_clips'
```

Next, set up the settings for annotation splitting:

- Whether to use a subset of classes
- How many seconds a label should overlap a clip, at minimum, in order for that clip to be labeled

[32]:
```
# Choose settings for annotation splitting
class_subset = None #Equivalent to a list of all classes: ['GWWA_song', '?']
min_label_overlap = 0.1
```

We'll want to keep the names of the audio clips that we create using `Audio.split_and_save()` so that we can correspond them with one-hot clip labels.

[33]:
```
# Store the label dataframes from each audio file so that we can aggregate them later
# Note: if you have a huge number (millions) of annotations, this might get very␣
↪large.
# An alternative would be to save the individual dataframes to files, then␣
↪concatenate them later.
all_labels = []

cnt = 0

for i, row in paired_df.iterrows():
    # Load the audio into an Audio object
    audio = Audio.from_file(row['audio_file'])

    # In this example, only the first 60 seconds of audio is annotated
    # So trim the audio to 60 seconds max
    audio = audio.trim(0,60)

    # Split the audio and save the clips
    clip_df = audio.split_and_save(
        clip_dir,
        prefix=row.name,
        clip_duration=clip_duration,
        clip_overlap=clip_overlap,
        final_clip=final_clip,
        dry_run=False
    )

    # Load the annotation file into a BoxedAnnotation object
    annotations = BoxedAnnotations.from_raven_file(row['raven_file'],annotation_
↪column='Species')

    # Since we trimmed the audio, we'll also trim the annotations for consistency
    annotations = annotations.trim(0,60)

    # Split the annotations to match the audio
    # We choose to keep_index=True so that we retain the audio clip's path in the␣
↪final label dataframe
    labels_df = annotations.one_hot_labels_like(clip_df,class_subset=class_subset,min_
↪label_overlap=min_label_overlap,keep_index=True)

    # Since we have saved short audio clips, we can discard the start_time and end_
↪time indices
```

(continues on next page)

(continued from previous page)

```
    labels_df = labels_df.reset_index(level=[1,2],drop=True)
    all_labels.append(labels_df)

    cnt+=1
    if cnt>2:
        break

#make one big dataframe with all of the labels. We could use this for training, for␣
↪instance.
all_labels = pd.concat(all_labels)
all_labels.to_csv("temp_clips/clip_annotations.csv")
all_labels
```

[33]:

```
                                  ?  GWWA_song
file
./temp_clips/13738_0.0s_3.0s.wav    0.0        1.0
./temp_clips/13738_3.0s_6.0s.wav    0.0        0.0
./temp_clips/13738_6.0s_9.0s.wav    0.0        1.0
./temp_clips/13738_9.0s_12.0s.wav   0.0        0.0
./temp_clips/13738_12.0s_15.0s.wav  1.0        0.0
./temp_clips/13738_15.0s_18.0s.wav  0.0        0.0
./temp_clips/13738_18.0s_21.0s.wav  0.0        1.0
./temp_clips/13738_21.0s_24.0s.wav  0.0        1.0
./temp_clips/13738_24.0s_27.0s.wav  0.0        1.0
./temp_clips/13738_27.0s_30.0s.wav  0.0        1.0
./temp_clips/13738_30.0s_33.0s.wav  0.0        0.0
./temp_clips/13738_33.0s_36.0s.wav  0.0        1.0
./temp_clips/13738_36.0s_39.0s.wav  0.0        0.0
./temp_clips/13738_39.0s_42.0s.wav  0.0        0.0
./temp_clips/13738_42.0s_45.0s.wav  0.0        1.0
./temp_clips/13738_45.0s_48.0s.wav  0.0        0.0
./temp_clips/13738_48.0s_51.0s.wav  0.0        0.0
./temp_clips/13738_51.0s_54.0s.wav  0.0        1.0
```

6.9 Sanity check: look at spectrograms of clips labeled 0 and 1

[34]:

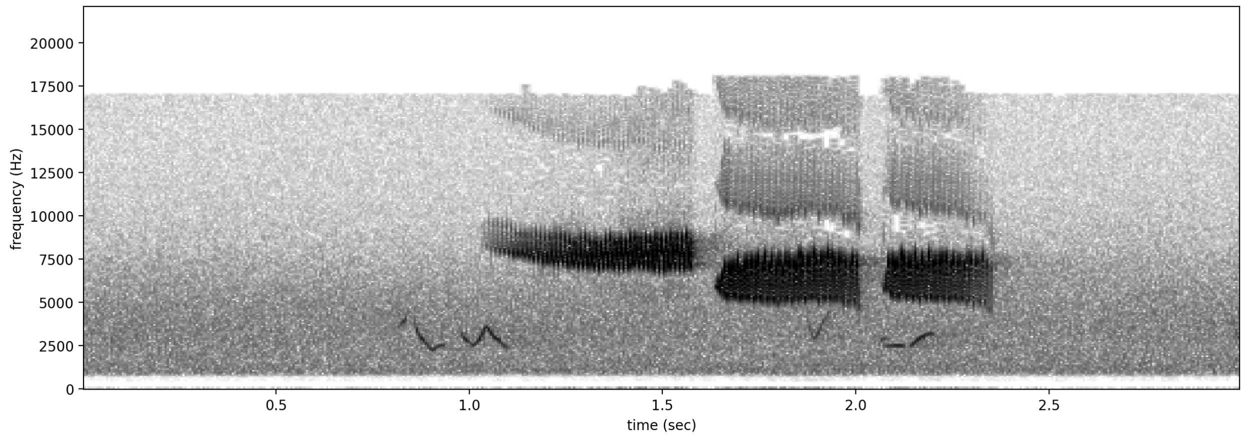
```
# ignore the "?" annotations for this visualization
all_labels = all_labels[all_labels["?"]==0]
```

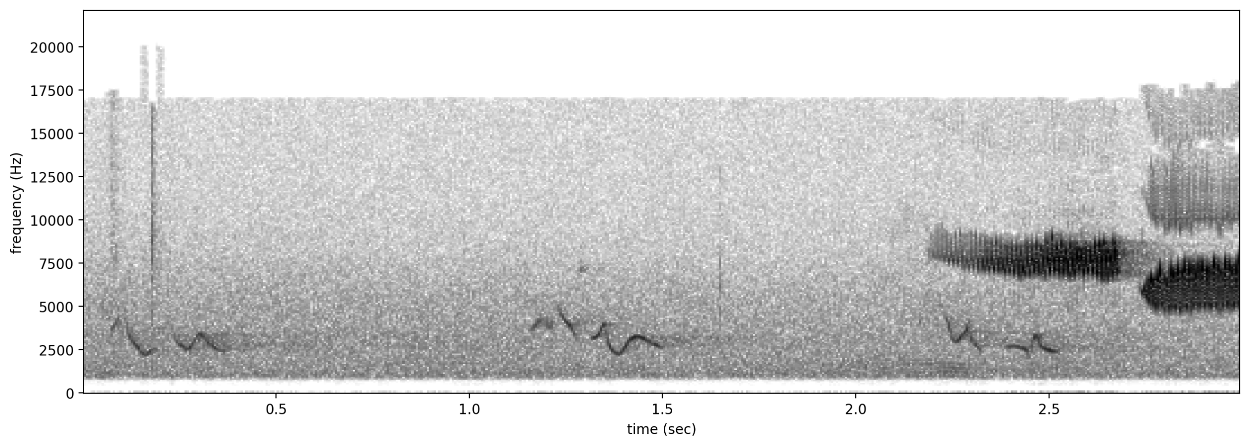
Note: replace the "GWWA_song" here with a class name from your own dataset.

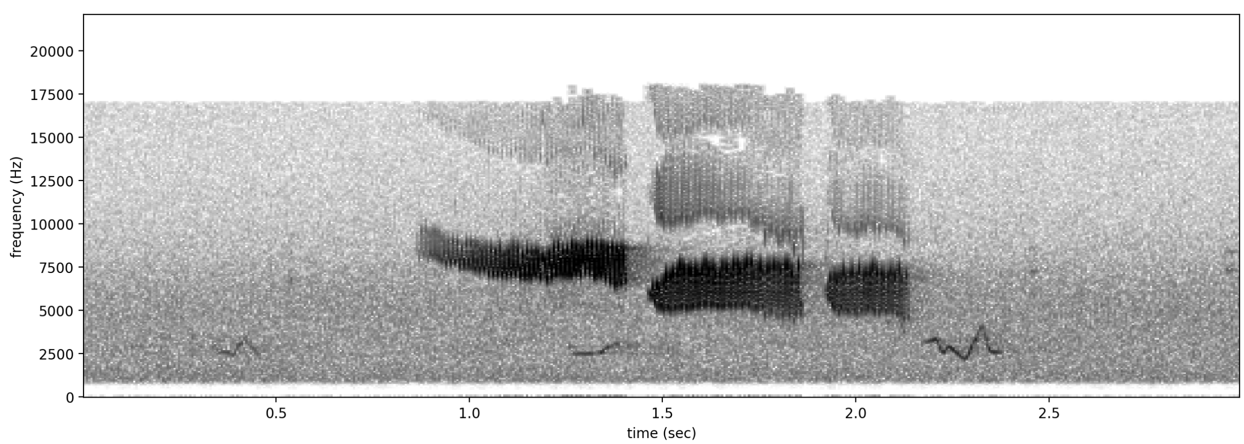
[35]:

```
# plot spectrograms for 3 random positive clips
positives = all_labels[all_labels['GWWA_song']==1].sample(3,random_state=0)
print("spectrograms of 3 random positive clips (label=1)")
for positive_clip in positives.index.values:
    Spectrogram.from_audio(Audio.from_file(positive_clip)).plot()

# plot spectrograms for 5 random negative clips
negatives = all_labels[all_labels['GWWA_song']==0].sample(3,random_state=0)
print("spectrogram of 3 random negative clips (label=0)")
for negative_clip in negatives.index.values:
    Spectrogram.from_audio(Audio.from_file(negative_clip)).plot()
```

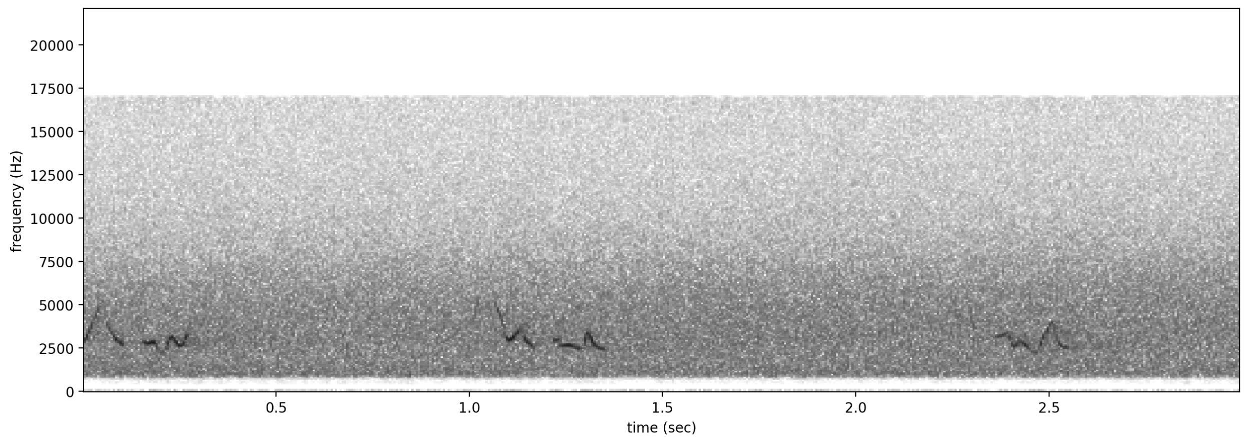
```
spectrograms of 3 random positive clips (label=1)
```

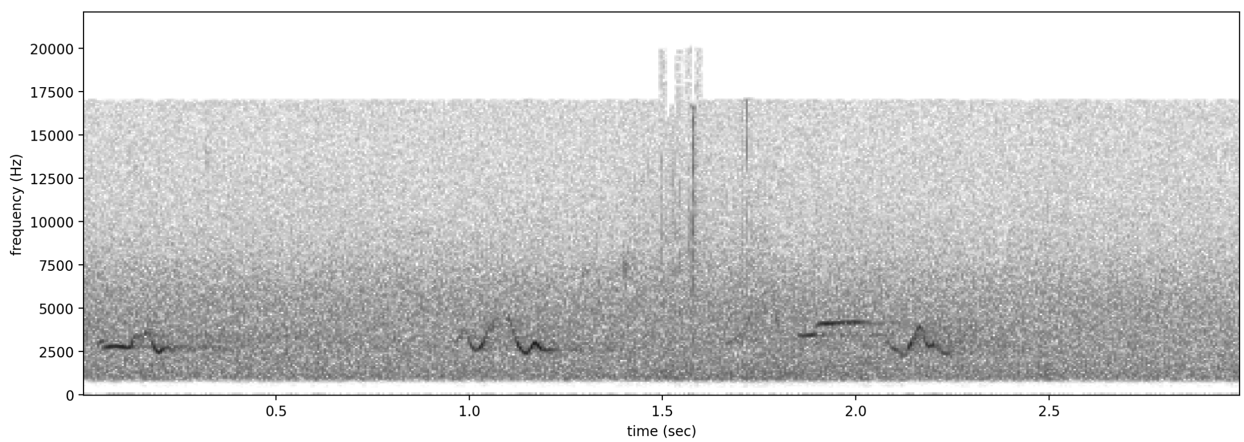


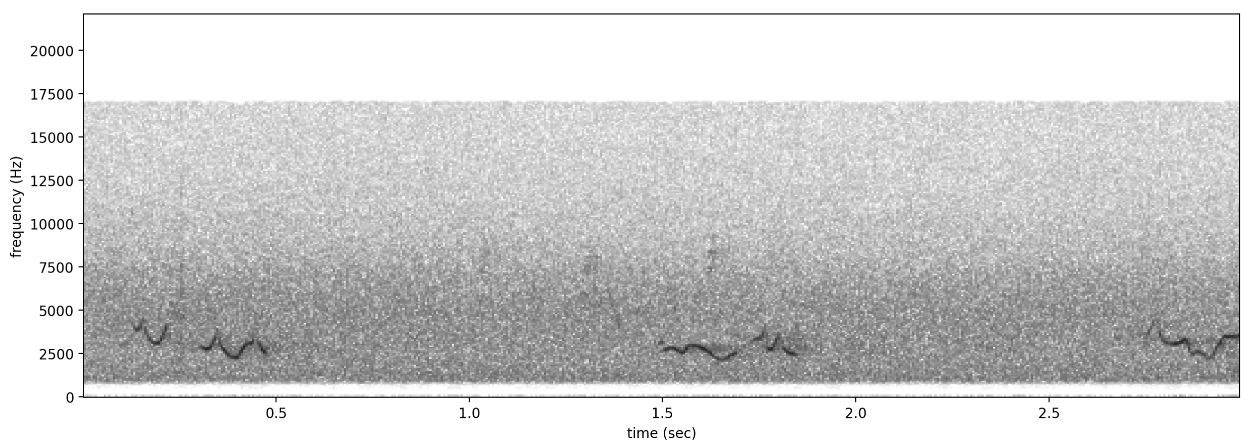





```
spectrogram of 3 random negative clips (label=0)
```







Clean up: remove the sounds that we downloaded for this tutorial as well as the `temp_clips` directory containing the split, saved clips.

[36]:
```
import shutil
shutil.rmtree('./gwwa_audio_and_raven_annotations')
shutil.rmtree('./temp_clips')
```

CHAPTER 7

Prediction with pre-trained CNNs

This notebook contains all the code you need to use a pre-trained OpenSoundscape convolutional neural network model (CNN) to make predictions on your own data. Before attempting this tutorial, install OpenSoundscape by following the instructions on the OpenSoundscape website, opensoundscape.org. More detailed tutorials about data preprocessing, training CNNs, and customizing prediction methods can also be found on this site.

7.1 Load required packages

The `cnn` module provides a function `load_model` to load saved opensoundscape models

[1]:
```
from opensoundscape.torch.models.cnn import load_model
import opensoundscape
```

load some additional packages and perform some setup for the Jupyter notebook.

[2]:
```
# Other utilities and packages
import torch
from pathlib import Path
import numpy as np
import pandas as pd
from glob import glob
import subprocess
```

[3]:
```
#set up plotting
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for large visuals
%config InlineBackend.figure_format = 'retina'
```

For this example, let's create an untrained model and save it. This 2-class model is not actually good at recognizing any particular species, but it's useful for illustrating how prediction works.

[4]:
```
from opensoundscape.torch.models.cnn import CNN
CNN('resnet18',['classA','classB'],5.0).save('./temp.model')
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and␣
↪may be removed in the future, please use 'weights' instead.
  warnings.warn(
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
↪'weights' are deprecated since 0.13 and may be removed in the future. The current␣
↪behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can␣
↪also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

7.1.1 Load a saved model

load the model object using the `load_model` function imported above

(if the model was created with an older version of opensoundscape, see instructions below)

[5]:
```
model = load_model('./temp.model')
```

7.1.2 Choose audio files for prediction

Create a list of audio files to predict on. They can be of any length. Consider using `glob` to find many files at once.

For this example, let's download a 1-minute audio clip from the Kitzes Lab box to use as an example.

[6]:
```
subprocess.run(['curl',
                'https://pitt.box.com/shared/static/z73eked7quh1t2pp93axzrrpq6wwydx0.
↪wav',
                 '-L', '-o', '1min_audio.wav'])
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      4      0 --:--:--  0:00:01 --:--:--  7000
100 3750k  100 3750k    0     0  1289k      0  0:00:02  0:00:02 --:--:-- 5677k
```

[6]:
```
CompletedProcess(args=['curl', 'https://pitt.box.com/shared/static/
↪z73eked7quh1t2pp93axzrrpq6wwydx0.wav', '-L', '-o', '1min_audio.wav'], returncode=0)
```

use glob to create a list of all files matching a pattern in a folder:

[7]:
```
from glob import glob
audio_files = glob('./*.wav') #match all .wav files in the current directory
audio_files
```

[7]:
```
['./1min_audio.wav']
```

7.2 generate predictions with the model

The model returns a dataframe with a MultiIndex of file, start_time, and end_time. There is one column for each class.

[8]:
```
scores = model.predict(audio_files)
scores.head()
```

[8]:

```
                                        classA    classB
file              start_time end_time
./1min_audio.wav 0.0         5.0      -0.290774 -0.155345
                 5.0         10.0     -0.154260 -0.143534
                 10.0        15.0     -0.043310 -0.486556
                 15.0        20.0     -0.162963 -0.302960
                 20.0        25.0     -0.265351 -0.279445
```

7.3 Overlapping prediction clips

[9]:

```
scores = model.predict(audio_files, overlap_fraction=0.5)
scores.head()
```

[9]:

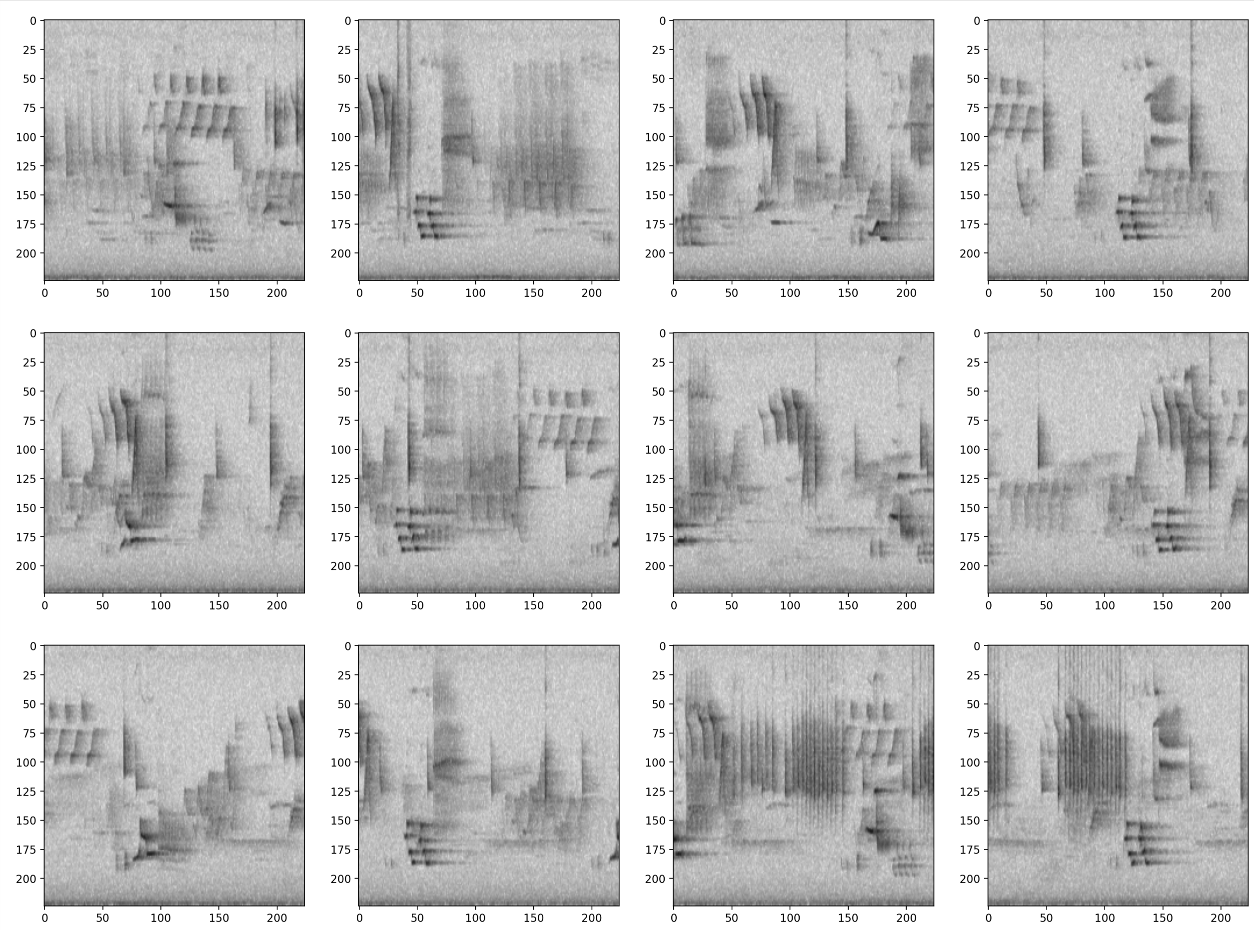
```
                                        classA    classB
file              start_time end_time
./1min_audio.wav 0.0         5.0      -0.290774 -0.155345
                 2.5         7.5      -0.075400 -0.448670
                 5.0         10.0     -0.154260 -0.143534
                 7.5         12.5     -0.216965 -0.149215
                 10.0        15.0     -0.043310 -0.486556
```

7.4 Inspect samples generated during prediction

[10]:

```
from opensoundscape.preprocess.utils import show_tensor_grid
from opensoundscape.torch.datasets import AudioSplittingDataset

#generate a dataset with the samples we wish to generate and the model's preprocessor
inspection_dataset = AudioSplittingDataset(audio_files, model.preprocessor)
inspection_dataset.bypass_augmentations = True

samples = [sample['X'] for sample in inspection_dataset]
_ = show_tensor_grid(samples,4)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/matplotlib_inline/
↪config.py:68: DeprecationWarning: InlineBackend._figure_format_changed is␣
↪deprecated in traitlets 4.1: use @observe and @unobserve instead.
  def _figure_format_changed(self, name, old, new):
```



7.5 Options for prediction

The code above returns the raw predictions of the model without any post-processing (such as a softmax layer or a sigmoid layer).

For details on how to post-processing prediction scores and to generate binary 0/1 predictions of class presence, see the "Basic training and prediction with CNNs" tutorial notebook. But, as a quick example here, let's add a softmax layer to make the prediction scores for both classes sum to 1.

We can also convert our continuous scores into True/False (or 1/0) predictions for the presence of each class in each sample. Think about whether each clip should be labeled with only one class (use `metrics.predict_single_target_labels`) or whether each clip could contain zero, one, or multiple classes (use `metrics.predict_multi_target_labels`)

[11]:
```
scores = model.predict(
    audio_files,
    activation_layer='softmax',
)
```

As before, the `scores` are continuous variables, but now have been softmaxed:

[12]:
```
scores.head()
```

[12]:
```
                                          classA    classB
file              start_time end_time
./1min_audio.wav  0.0        5.0        0.466194  0.533806
                  5.0        10.0       0.497319  0.502681
                  10.0       15.0       0.609032  0.390968
                  15.0       20.0       0.534942  0.465058
                  20.0       25.0       0.503524  0.496476
```

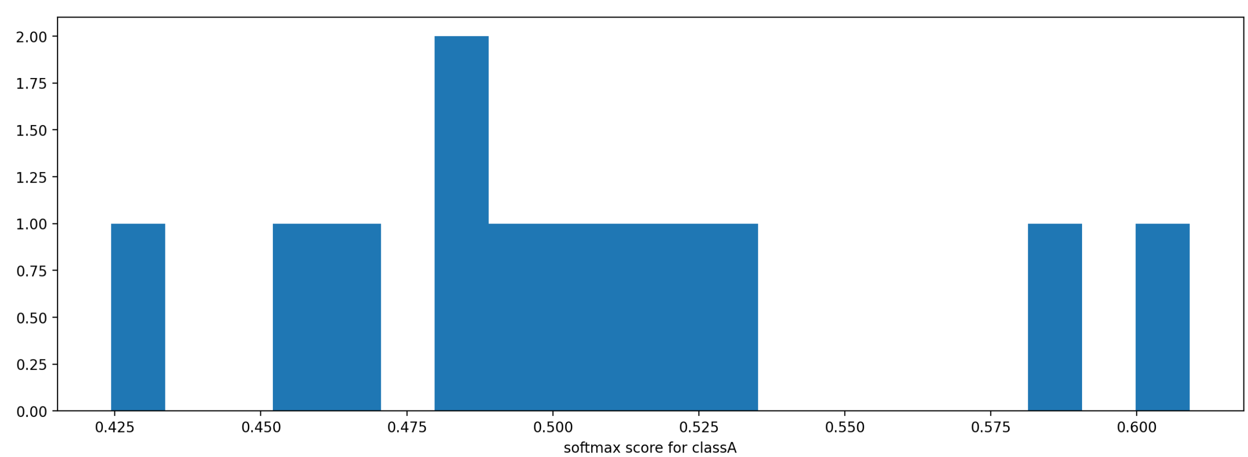
Now let's use the `predict_single_target_labels(scores)` function to label the highest scoring class 1 for each sample, and other classes 0.

[13]:
```
from opensoundscape.metrics import predict_single_target_labels
predicted_labels = predict_single_target_labels(scores)
predicted_labels.head()
```

[13]:
```
                                        classA  classB
file              start_time end_time
./1min_audio.wav  0.0        5.0             0       1
                  5.0        10.0            0       1
                  10.0       15.0            1       0
                  15.0       20.0            1       0
                  20.0       25.0            1       0
```

It is sometimes helpful to look at a histogram of the scores:

[14]:
```
_ = plt.hist(scores['classA'],bins=20)
_ = plt.xlabel('softmax score for classA')
```



7.6 Using models from older OpenSoundscape versions

7.6.1 Models from OpenSoundscape 0.4.x and 0.5.x

Models trained and saved with OpenSoundscape versions 0.4.x and 0.5.x need to be loaded in a different way, and require that you know the architecture of the saved model.

For example, one set of our publicly availably binary models for 500 species was created with an older version of OpenSoundscape. These models require a little bit of manipulation to load into OpenSoundscape 0.5.x and onward.

First, let's download one of these models (it's stored in a .tar format) and save it to the same directory as this notebook in a file called `opso_04_model_acanthis-flammea.tar`

[15]:
```
subprocess.run(['curl',
               'https://pitt.box.com/shared/static/1glpty35omjhmq6cdz8cfudm43nn2t9f.
↪tar',
                '-L', '-o', 'opso_04_model_acanthis-flammea.tar'])
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     8    0     8    0     0      4      0 --:--:--  0:00:01 --:--:--   571
100 42.9M  100 42.9M    0     0  6128k      0  0:00:07  0:00:07 --:--:-- 10.6M
```

[15]:
```
CompletedProcess(args=['curl', 'https://pitt.box.com/shared/static/
↪1glpty35omjhmq6cdz8cfudm43nn2t9f.tar', '-L', '-o', 'opso_04_model_acanthis-flammea.
↪tar'], returncode=0)
```

From the model notes page, we know that this is a single-target model with a resnet18 architecture trained on 5 second files. Let's load the model with `load_outdated_model`. We also need to make sure we use the same preprocessing settings as the original model. In this case, the original model used the same preprocessing settings as the default CNN.preprocessor.

[16]:
```
from opensoundscape.torch.models.cnn import load_outdated_model
```

[17]:
```
model = load_outdated_model('./opso_04_model_acanthis-flammea.tar','resnet18',5.0)

#invert values to match the convention of OpenSoundscape 0.7.x (lowest values = quiet,
↪ highest = loud)
model.preprocessor.pipeline.to_img.set(invert=True)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and␣
↪may be removed in the future, please use 'weights' instead.
  warnings.warn(
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
↪'weights' are deprecated since 0.13 and may be removed in the future. The current␣
↪behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can␣
↪also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
mismatched keys:
<All keys matched successfully>
```

```
/Users/SML161/opensoundscape/opensoundscape/torch/models/cnn.py:1363: UserWarning:␣
↪After loading a model, you still need to ensure that your preprocessing (model.
↪preprocessor) matches the settings used to createthe original model.
  warnings.warn(
```

Again, you may need to modify model.preprocessor to match the settings used to train the model.

The model is now fully compatible with OpenSoundscape, and can be used as above. For example:

[18]:
```
scores = model.predict(audio_files)
scores.head()
```

[18]:
```
                                           acanthis-flammea-absent  \
file             start_time end_time
./1min_audio.wav 0.0        5.0                           6.254239
                 5.0        10.0                          4.935342
                 10.0       15.0                          6.227312
```

(continues on next page)

(continued from previous page)

```
                 15.0       20.0                    5.256021
                 20.0       25.0                    4.836051

                                                acanthis-flammea-present
file             start_time end_time
./1min_audio.wav 0.0        5.0                    -5.859637
                 5.0        10.0                   -4.917323
                 10.0       15.0                   -5.752949
                 15.0       20.0                   -5.732774
                 20.0       25.0                   -5.272484
```

if we save the model using `model.save(path)`, we can re-load the full model object later using `load_model()` rather than repeating the procedure above.

7.6.2 Loading models from OpenSoundscape 0.6.0

If you saved a model with OpenSoundscape 0.6.0 and want to use it in 0.7.0 or above, you will need to re-load the model using the original OpenSoundscape version that it was created with and save the model's weights explicitly. Here's an example of code you could run *in an environment with opensoundscape version 0.6.0* to export a model for use in later OpenSoundscape versions:

```
# This code will only work in an environment with OpenSoundscape version 0.6.0
# Use it if you need to save a model created in OpenSoundscape v0.6.0 for use in␣
↪later opso versions

import torch
from opensoundscape.torch.models.cnn import Resnet18Binary #choose the class used to␣
↪create the model

model = Resnet18Binary(classes=['negative','positive']) #provide the list of classes␣
↪for the model
model.load('/path/to/saved.model')

dict_to_save = {
    'network_state_dict':model.network.state_dict(),
    'classes': model.classes,
}
torch.save(dict_to_save, '/path/to/model_dict.pt')
```

Then, you will be able to create a new model object in OpenSoundscape >=0.7.0 and load the weights from the state dict as demonstrated above. Make sure to specify the correct architecture and sample duration when you create the CNN object.

```
#run this code in an envrionment with a newer OpenSoundscape version >=0.7.0

import torch
from opensoundscape.torch.models.cnn import CNN

model_dict = torch.load('/path/to/model_dict.pt')
classes = model_dict["classes"]

#remove the 'feature' prefix on weights and replace the 'classifier' prefix with 'fc'
model_dict['network_state_dict'] = {
    k.replace('feature.','').replace('classifier.','fc.'):v
    for k, v in model_dict['network_state_dict'].items()
```

(continues on next page)

(continued from previous page)

```
}

architecture = 'resnet18' #match this with the original model!

sample_duration = 5.0 #match this with the original model!

model = CNN('resnet18',classes,sample_duration)
model.network.load_state_dict(model_dict['network_state_dict'])

#invert values to match the convention of OpenSoundscape 0.7.x
model.preprocessor.pipeline.to_img.set(invert=True)

#save the model object so that we can simply reload it with load_model() in the␣
↪future:
model.save('/path/to/saved_full_object.model')

# Next time, we can just load the full model object directly:
from opensoundscape.torch.models.cnn import load_model
model = load_model('/path/to/saved_full_object.model')
```

7.6.3 Loading models from OpenSoundscape 0.6.1 and 0.6.2

If you saved a model with OpenSoundscape 0.6.1 or 0.6.2 and want to use it in 0.7.0 or above, you will need to re-load the model using the original OpenSoundscape version that it was created with and save the model's weights explicitly. Here's an example of code you could run *in an environment with opensoundscape version 0.6.1 or 0.6.2* to export a model for use in later OpenSoundscape versions:

```
# This code will only work in an environment with OpenSoundscape version 0.6.1 or 0.6.
↪2
# Use it if you need to save a model created in OpenSoundscape v0.6.1 or 0.6.2 for␣
↪use in later opso
versions

import torch
from opensoundscape.torch.models.cnn import load_model
model = load_model('/path/to/saved.model')

dict_to_save = {
    'network_state_dict':model.network.state_dict(),
    'classes': model.classes,
    '
}
torch.save(dict_to_save, '/path/to/model_dict.pt')
```

Then, you will be able to create a new model object in OpenSoundscape 0.7.0 and load the weights from the state dict as demonstrated above. Make sure to specify the correct architecture and sample duration when you create the CNN object.

```
#run this code in an envrionment with a newer OpenSoundscape version >=0.7.0

import torch
from opensoundscape.torch.models.cnn import CNN

model_dict = torch.load('/path/to/model_dict.pt')
```

(continues on next page)

(continued from previous page)

```
classes = model_dict["classes"]

architecture = 'resnet18' #match this with the original model!

sample_duration = 5.0 #match this with the original model!

model = CNN('resnet18',classes,sample_duration)
model.network.load_state_dict(model_dict['network_state_dict'])

#invert values to match the convention of OpenSoundscape 0.7.x
model.preprocessor.pipeline.to_img.set(invert=True)

#save the model object so that we can simply reload it with load_model() in the␣
↪future:
model.save('/path/to/saved_full_object.model')

# Next time, we can just load the full model object directly:
from opensoundscape.torch.models.cnn import load_model
model = load_model('/path/to/saved_full_object.model')
```

OpenSoundscape model objects include helper functions .save_weights() and .load_weights() which allow you to save and load platform/class independent dictionaries for increased flexibility. The weights saved and loaded by these functions are simply a dictionary of keys and numeric values, so they don't depend on the existence of particular classes in the code base. We recommend saving both the full model object (`.save()`) and the raw weights (`.save_weights()`) for models you plan to use in the future.

7.6.4 Clean up: delete model objects

[19]:
```
from pathlib import Path
for p in Path('.').glob('*.model'):
    p.unlink()
for p in Path('.').glob('*.tar'):
    p.unlink()
Path('1min_audio.wav').unlink()
```

CHAPTER 8

Beginner friendly training and prediction with CNNs

Convolutional Neural Networks (CNNs) are a popular tool for developing automated machine learning classifiers on images or image-like samples. By converting audio into a two-dimensional frequency vs. time representation such as a spectrogram, we can generate image-like samples that can be used to train CNNs. This tutorial demonstrates the basic use of OpenSoundscape's `preprocessors` and `cnn` modules for training CNNs and making predictions using CNNs.

Under the hood, OpenSoundscape uses Pytorch for machine learning tasks. By using the class `opensoundscape.torch.models.cnn.CNN`, you can train and predict with PyTorch's powerful CNN architectures in just a few lines of code.

First, let's import some utilities.

[1]:
```
# the cnn module provides classes for training/predicting with various types of CNNs
from opensoundscape.torch.models.cnn import CNN

#other utilities and packages
import torch
import pandas as pd
from pathlib import Path
import numpy as np
import pandas as pd
import random
import subprocess

#set up plotting
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for large visuals
%config InlineBackend.figure_format = 'retina'
```

Set manual seeds for pytorch and python. These ensure the training results are reproducible. You probably don't want to do this when you actually train your model, but it's useful for debugging.

[2]:
```
torch.manual_seed(0)
random.seed(0)
np.random.seed(0)
```

8.1 Prepare audio data

8.1.1 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 `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.

Note: Once you have the data, you do not need to run this cell again.

[3]:
```
subprocess.run(['curl','https://pitt.box.com/shared/static/
↪79fi7d715dulcldsy6uogz02rsn5uesd.gz','-L', '-o','woodcock_labeled_data.tar.gz']) #␣
↪Download the data
subprocess.run(["tar","-xzf", "woodcock_labeled_data.tar.gz"]) # Unzip the downloaded␣
↪tar.gz file
subprocess.run(["rm", "woodcock_labeled_data.tar.gz"]) # Remove the file after its␣
↪contents are unzipped
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      6      0 --:--:--  0:00:01 --:--:--    12
100 9499k  100 9499k    0     0  3210k      0  0:00:02  0:00:02 --:--:-- 6464k
```

[3]:
```
CompletedProcess(args=['rm', 'woodcock_labeled_data.tar.gz'], returncode=0)
```

8.1.2 Generate one-hot encoded labels

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.

[4]:
```
#load Specky output: a table of labeled audio files
specky_table = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))
specky_table.head()
```

[4]:
```
                                 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
```

This table must provide an accurate path to the files of interest. For this self-contained tutorial, we can use relative paths (starting with a dot and referring to files in the same folder), but you may want to use absolute paths for your training.

[5]:
```
#update the paths to the audio files
specky_table.filename = ['./woodcock_labeled_data/'+f for f in specky_table.filename]
specky_table.head()
```

[5]:
```
                                             filename woodcock sound_type
0  ./woodcock_labeled_data/d4c40b6066b489518f8da8...  present       song
1  ./woodcock_labeled_data/e84a4b60a4f2d049d73162...   absent         na
2  ./woodcock_labeled_data/79678c979ebb880d5ed6d5...  present       song
3  ./woodcock_labeled_data/49890077267b569e142440...  present       song
4  ./woodcock_labeled_data/0c453a87185d8c7ce05c5c...  present       song
```

We then modify these labels, replacing `present` with 1 and `absent` with zero. Ones and zeros are the way that presences and absences are represented in a machine learning model.

[17]:
```
# create a new dataframe with the filenames from the previous table as the index
labels = pd.DataFrame(index=specky_table['filename'])

#convert 'present' to 1 and 'absent' to 0
labels['woodcock']=[1 if l=='present' else 0 for l in specky_table['woodcock']]

#look at the first rows
labels.head(3)
```

[17]:
```
                                                    woodcock
filename
./woodcock_labeled_data/d4c40b6066b489518f8da83...         1
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...         0
./woodcock_labeled_data/79678c979ebb880d5ed6d56...         1
```

8.1.3 Split into training and validation sets

We use a utility from `sklearn` to randomly divide the labeled samples into two sets. The first set, `train_df`, will be used to train the CNN, while the second set, `valid_df`, will be used to test how well the model can predict the classes of samples that it was not trained with.

During the training process, the CNN will go through all of the samples once every "epoch" for several (sometimes hundreds of) epochs. Each epoch usually consists of a "learning" step and a "validation" step. In the learning step, the CNN iterates through all of the training samples while the computer program is modifying the weights of the convolutional neural network. In the validation step, the program performs prediction on all of the validation samples and prints out metrics to assess how well the classifier generalizes to unseen data.

Note: using the `random_state` argument with a fixed number means that the "random" split will be exactly the same each time we run it. This is useful for reproducible results, but to get a different split each time you would not use the `random_state` argument.

[19]:
```
from sklearn.model_selection import train_test_split
train_df,validation_df = train_test_split(labels,test_size=0.2,random_state=1)
```

8.2 Create and train a model

Now, we create a convolutional neural network model object, train it on the `train_dataset` with validation from `validation_dataset`

8.2.1 Set up a one-class CNN

The purpose of this model is to predict the presence or absence of a single species, so it has one class "woodcock". Its also possible to train models to recognize multiple species - we call these "multi-class models" and each category of sounds it learns to recognize is a "class".

The model object should be initialized with a list of class names that matches the class names in the training dataset. Here we'll use the resnet18 architecture, a popular and powerful architecture that makes a good starting point. For more details on other CNN architectures, see the "Advanced CNN Training" tutorial.

[26]:
```
# Create model object
classes = train_df.columns #in this case, there's just one class: ["woodcock"]
model = CNN('resnet18',classes=classes,sample_duration=2.0)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and␣
↪may be removed in the future, please use 'weights' instead.
  warnings.warn(
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
↪'weights' are deprecated since 0.13 and may be removed in the future. The current␣
↪behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can␣
↪also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

CAVEAT: the default audio preprocessing in this class bandpasses spectrograms to 0-11025 Hz. If your audio has a sample rate of less than 22050 Hz, the preprocessing will raise an error because the spectrogram will not contain the expected frequencies. In this case you could change the parameters of the bandpass action, or simply disable the bandpass action:

```
model.preprocessor.pipeline.bandpass.bypass=True
```

8.2.2 Inspect training images

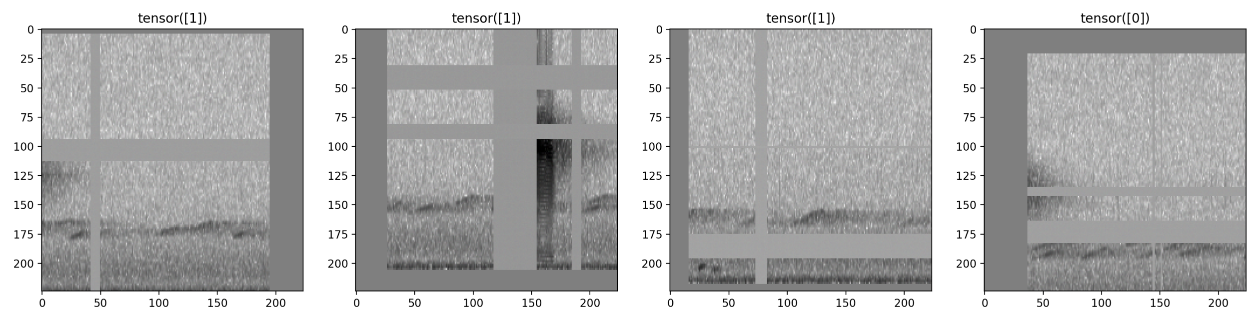
Before creating a machine learning algorithm, we strongly recommend making sure the images coming out of the preprocessor look like you expect them to. Here we generate images for a few samples.

[27]:
```
#helper functions to visualize processed samples
from opensoundscape.preprocess.utils import show_tensor_grid, show_tensor
from opensoundscape.torch.datasets import AudioFileDataset
```

Now, let's check what the samples generated by our model look like

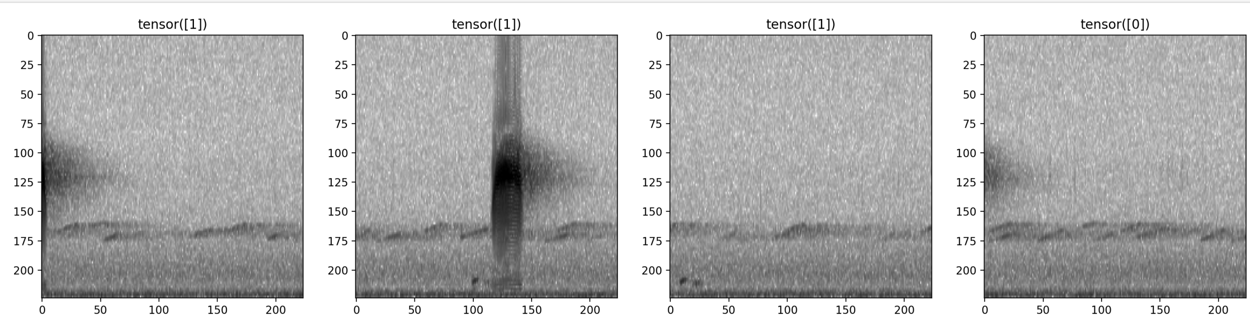
[28]:
```
#pick some random samples from the training set

sample_of_4 = train_df.sample(n=4)

#generate a dataset with the samples we wish to generate and the model's preprocessor
inspection_dataset = AudioFileDataset(sample_of_4, model.preprocessor)

#generate the samples using the dataset
samples = [sample['X'] for sample in inspection_dataset]
labels = [sample['y'] for sample in inspection_dataset]

#display the samples
_ = show_tensor_grid(samples,4,labels=labels)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
↪[0..255] for integers).
```



The dataset allows you to turn all augmentation off or on as desired. Inspect the unaugmented images as well:

[29]:
```
#turn augmentation off for the dataset
inspection_dataset.bypass_augmentations = True

#generate the samples without augmentation
samples = [sample['X'] for sample in inspection_dataset]
labels = [sample['y'] for sample in inspection_dataset]

#display the samples
_ = show_tensor_grid(samples,4,labels=labels)
```



8.2.3 Train the model

Depending on the speed of your computer, training the CNN may take a few minutes.

We'll only train for 5 epochs on this small dataset as a demonstration, but you'll probably need to train for tens (or hundreds) of epochs on hundreds (or thousands) of training files to create a useful model.

Batch size refers to the number of samples that are simultaneously processed by the model. In practice, using larger batch sizes (64+) improves stability and generalizability of training, particularly for architectures (such as ResNet) that contain a 'batch norm' layer. Here we use a small batch size to keep the computational requirements for this tutorial low.

[30]:
```
model.train(
    train_df=train_df,
    validation_df=validation_df,
    save_path='./binary_train/', #where to save the trained model
    epochs=5,
    batch_size=8,
```

(continues on next page)

(continued from previous page)

```
    save_interval=5, #save model every 5 epochs (the best model is always saved in␣
↪addition)
    num_workers=0, #specify 4 if you have 4 CPU processes, eg; 0 means only the root␣
↪process
)
```

```
Training Epoch 0
Epoch: 0 [batch 0/3, 0.00%]
        DistLoss: 0.763
Metrics:
Metrics:
        MAP: 0.921

Validation.
Metrics:
        MAP: 1.000

Training Epoch 1
Epoch: 1 [batch 0/3, 0.00%]
        DistLoss: 0.316
Metrics:
Metrics:
        MAP: 0.816

Validation.
Metrics:
        MAP: 1.000

Training Epoch 2
Epoch: 2 [batch 0/3, 0.00%]
        DistLoss: 0.335
Metrics:
Metrics:
        MAP: 0.899

Validation.
Metrics:
        MAP: 1.000

Training Epoch 3
Epoch: 3 [batch 0/3, 0.00%]
        DistLoss: 0.361
Metrics:
Metrics:
        MAP: 0.993

Validation.
Metrics:
        MAP: 1.000

Training Epoch 4
Epoch: 4 [batch 0/3, 0.00%]
        DistLoss: 0.460
Metrics:
Metrics:
        MAP: 0.909
```

(continues on next page)

(continued from previous page)

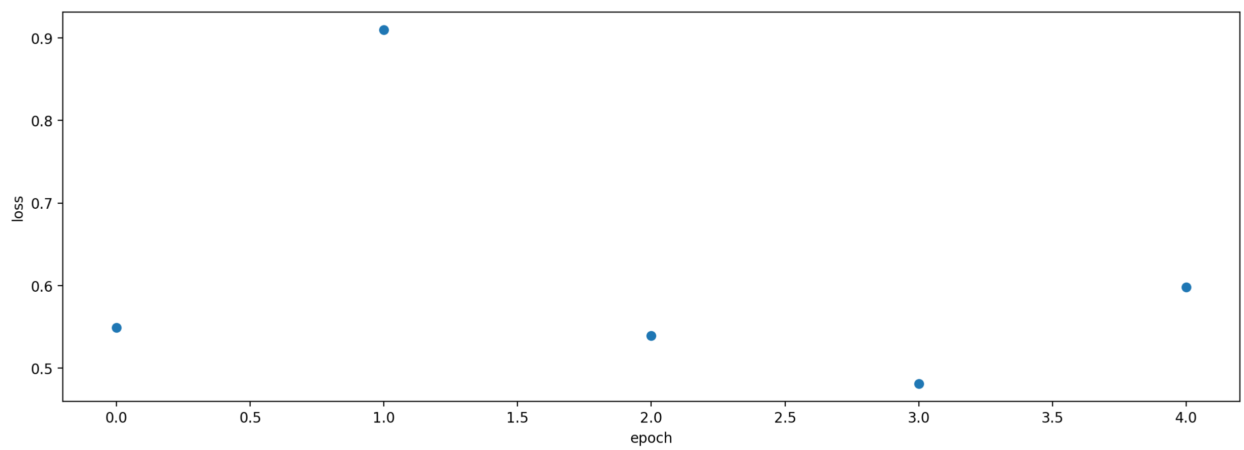
```
Validation.
Metrics:
        MAP: 0.967

Best Model Appears at Epoch 0 with Validation score 1.000.
```

8.2.4 Plot the loss history

We can plot the loss from each epoch to check that our loss is declining. Loss should decline as the model learns, but may have ups and downs along the way.

```
[31]: plt.scatter(model.loss_hist.keys(),model.loss_hist.values())
plt.xlabel('epoch')
plt.ylabel('loss')
```

```
[31]: Text(0, 0.5, 'loss')
```



8.2.5 Printing and Logging outputs

We can log the outputs of the training process to a file, and/or print them. We can independently modify how much content is logged/printed with the model's attributes `model.verbose` and `model.logging_level`. Content increases from level 0 (nothing) to 1 (standard), 2, 3, etc. For instance, let's train for an epoch with lots of logged content but no printed output:

```
[32]: model.logging_level = 3 #request lots of logged content
model.log_file = './binary_train/training_log.txt' #specify a file to log output to
Path(model.log_file).parent.mkdir(parents=True,exist_ok=True) #make the folder ./
↪binary_train

model.verbose = 0 #don't print anything to the screen during training
model.train(
    train_df=train_df,
    validation_df=validation_df,
    save_path='./binary_train/', #where to save the trained model
    epochs=1,
    batch_size=8,
```

(continues on next page)

(continued from previous page)

```
    save_interval=5, #save model every 5 epochs (the best model is always saved in␣
↪addition)
    num_workers=0, #specify 4 if you have 4 CPU processes, eg; 0 means only the root␣
↪process
)
```

8.3 Prediction

We haven't actually trained a useful model in 5 epochs, but we can use the trained model to demonstrate how prediction works and show several of the settings useful for prediction.

We will run prediction on two one-minute clips of field data recorded by an AudioMoth acoustic recorded. The two files are located in `woodcock_labeled_data/field_data`

8.3.1 Predict on the field data

To run prediction, also known as "inference", wich a CNN, we simply call model's `predict` method and pass it a list of file paths (or a dataframe with file paths in the index).

The predict function will internally split audio files into the appropriate length clips for prediction and generate prediction scores for each clip.

- By default, there is no overlap between these clips, but we can specify a fraction of overlap with consecutive clips with the `overlap_fraction` argument (eg, 0.5 for 50% overlap).
- Additionally, if we want to predict on audio files that are already trimmed to the same duration as the training files, we can specify `split_files_into_clips=False`.

Calling `.predict()` will return a dataframe with numeric (continuous score) predictions from the model for each sample and class (by default these are raw outputs from the model).

Let's predict on the two field recordings:

[33]:
```
from glob import glob
field_recordings = glob('./woodcock_labeled_data/field_data/*')
field_recordings
```

[33]:
```
['./woodcock_labeled_data/field_data/60s_field_data_sample_1.wav',
 './woodcock_labeled_data/field_data/60s_field_data_sample_2.wav']
```

[34]:
```
prediction_scores_df = model.predict(field_recordings)
```

The predict function generated a dataframe with rows for each 2-second segment of each 1-minute audio clip. Let's look at the first few rows:

[35]:
```
prediction_scores_df.head()
```

[35]:

file	start_time	end_time	woodcock
./woodcock_labeled_data/field_data/60s_field_da...	0.0	2.0	1.556411
	2.0	4.0	1.474465
	4.0	6.0	1.229040
	6.0	8.0	1.426526
	8.0	10.0	0.769797

8.3.2 Generate boolean predicted class labels (0/1) from the continuous scores

Note: Presence/absence predictions always have some error rates, sometimes large ones. It is not generally advisable to use these binary predictions as scientific observations without a thorough understanding of the model's false-positive and false-negative rates.

There are two different ways we might want to predict class labels which reflect the nature of the classes themselves:

single target means that out of a set of classes, one and only one should be chosen for each sample. For instance, if our classes were days of the week, any single should be labeled with one and only one day of the week. In opensoundscape, use the function `generate_single_target_labels()` to convert scores to predicted single target labels. For each sample, the class with the highest score will recieve a label of 1 and all other classes will recieve a label of 0.

multi-target means that a sample can have 0, 1, or more than 1 labels. For instance, if our classes were the types of flowers in a photo, any given photo might have none of the classes, one class, or multiple different classes at once. In opensoundscape, use the function `generate_multi_target_labels()` to convert scores to predicted multi target labels. For each sample and each class, the class will be labeled 1 if its score is higher than a user-specified threshold and 0 otherwise. You can choose to use a single threshold for all classes, or specify a different threshold for each class.

[36]:
```
from opensoundscape.metrics import predict_single_target_labels

score_df = model.predict(field_recordings)

pred_df = predict_single_target_labels(score_df)
pred_df.head()
```

[36]:
```
                                                                   woodcock
file                                               start_time end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0        2.0            1
                                                   2.0        4.0            1
                                                   4.0        6.0            1
                                                   6.0        8.0            1
                                                   8.0        10.0           1
```

The `predict_multi_target_labels` function allows you to select a threshold. If a score exceeds that threshold, the binary prediction is 1; otherwise, it is 0. You can also specify a list of thresholds, with one for each class

[40]:
```
from opensoundscape.metrics import predict_multi_target_labels

multi_target_pred_df = predict_multi_target_labels(score_df,threshold=0.99)
multi_target_pred_df.head()
```

[40]:
```
                                                                   woodcock
file                                               start_time end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0        2.0            1
                                                   2.0        4.0            1
                                                   4.0        6.0            1
                                                   6.0        8.0            1
                                                   8.0        10.0           0
```

Note that it is possible both the negative and positive classes are predicted to be present. This is because multi_target labeling assumes that the classes are not mutually exclusive. For a presence/absence model like the one above, single_target labeling is more appropriate.

8.3.3 Change the activation layer

We can modify the final activation layer to change the scores returned by the `predict()` function. Note that this does not impact the results of the binary predictions (described above), which are always calculated using a sigmoid transformation (for multi-target models) or softmax function (for single-target models).

Options include:

- `None`: default. Just the raw outputs of the network, which are in (-inf, inf)
- `'softmax'`: scores across all classes will sum to 1 for each sample
- `'softmax_and_logit'`: softmax the scores across all classes so they sum to 1, then apply the "logit" transformation to these scores, taking them from [0,1] back to (-inf,inf)
- `'sigmoid'`: transforms each score individually to [0, 1] without requiring that all scores sum to 1

In this case, since we are just looking at the output of one class, we can use the 'sigmoid' activation layer to put scores on the interval [0,1]

Let's generate binary 0/1 predictions on the validation set. Since these samples are the same length as the training files, we'll specify `split_files_into_clips=False` (we just want one prediction per file, we don't want to divide each file into shorter clips).

[41]:
```
valid_scores = model.predict(
    validation_df,
    activation_layer='sigmoid',
    split_files_into_clips=False
)
```

Compare the softmax scores to the true labels for this dataset, side-by-side:

[43]:
```
valid_scores.columns = ['pred_woodcock']
validation_df.join(valid_scores).sample(5)
```

[43]:

filename	woodcock	pred_woodcock
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	1	0.998974
./woodcock_labeled_data/4afa902e823095e03ba23eb...	1	0.999948
./woodcock_labeled_data/92647ab903049a9ee4125ab...	1	0.999713
./woodcock_labeled_data/882de25226ed989b31274ee...	1	0.997224
./woodcock_labeled_data/ad14ac7ffa729060712b442...	0	0.893294

We can directly compare our model's confidence that woodcock is present with the original labels

8.3.4 Parallelizing prediction

Two parameters can be used to increase prediction efficiency, depending on the computational resources available:

- `num_workers`: Pytorch's method of parallelizing across cores (CPUs) - choose 0 to predict on the root process, or >1 if you want to use more than 1 CPU process.
- `batch_size`: number of samples to predict on simultaneously. You can try increasing this by factors of two until you get a memory error, which means your batch size is too large for your system.

[45]:
```
score_df = model.predict(
    validation_df,
    batch_size=8,
    num_workers=0,
)
```

8.4 Multi-class models

A multi-class model can have any number of classes, and can be either

- multi-target: any number of classes can be positive for one sample
- single-target: exactly one class is positive for each sample

Models that are multi-target benefit from a modified loss function, and we have implemented a special class that is specifically designed for multi-target problems called `ResampleLoss`. We can use it as follows:

[82]:
```
from opensoundscape.torch.models.cnn import use_resample_loss
model = CNN('resnet18',classes,2.0,single_target=False)
use_resample_loss(model)
print("model.single_target:", model.single_target)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and␣
↪may be removed in the future, please use 'weights' instead.
  warnings.warn(
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
↪'weights' are deprecated since 0.13 and may be removed in the future. The current␣
↪behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can␣
↪also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
model.single_target: False
```

8.4.1 Train

Training looks the same as in one-class models.

[83]:
```
model.train(
    train_df,
    validation_df,
    save_path='./multilabel_train/',
    epochs=1,
    batch_size=64,
    save_interval=100,
    num_workers=0,
)
```

```

Training Epoch 0
Epoch: 0 [batch 0/1, 0.00%]
        DistLoss: nan
Metrics:
Metrics:
        MAP: nan

Validation.

Best Model Appears at Epoch 0 with Validation score 0.000.
```

```
/Users/SML161/opensoundscape/opensoundscape/torch/models/cnn.py:675: UserWarning:␣
↪Recieved empty list of predictions (or all nan)
  warnings.warn("Recieved empty list of predictions (or all nan)")
```

Note: since we used the same data as above, we just trained a 1 class model with "resample loss". You should not actually use resample loss for single class models!

8.4.2 Predict

Prediction looks the same as demonstrated above, but make sure to think carefully:

- What `activation_layer` do you want?
- If creating boolean (0/1 or True/False) predictions for each sample and class, is my model single-target (use `metrics.predict_single_target_labels`) or multi-target (use `metrics.predict_multi_target_labels`)?

For more detail on these choices, see the sections about activation layers and boolean predictions above.

8.5 Save and load models

Models can be easily saved to a file and loaded at a later time. If the model was saved with OpenSoundscape version >=0.6.1, the entire model object will be saved - including the class, cnn architecture, loss function, and training/validation datasets. Models saved with earlier versions of OpenSoundscape do not contain all of this information and may require that you know their class and architecture (see below).

8.5.1 Save and load a model

OpenSoundscape saves models automatically during training:

- The model saves a copy of itself `self.save_path` to `epoch-X.model` automatically during training every `save_interval` epochs
- The model keeps the file `best.model` updated with the weights that achieve the best score on the validation dataset. By default the model is evaluated using the mean average precision (MAP) score, but you can overwrite `model.eval()` if you want to use a different metric for the best model.

You can also save the model manually at any time with `model.save(path)`

[85]:
```
model1 = CNN('resnet18',classes,2.0,single_target=False)
# Save every 2 epochs
model1.train(
    train_df,
    validation_df,
    epochs=3,
    batch_size=8,
    save_path='./binary_train/',
    save_interval=2,
    num_workers=0
)
model1.save('./binary_train/my_favorite.model')
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and␣
↪may be removed in the future, please use 'weights' instead.
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
↪'weights' are deprecated since 0.13 and may be removed in the future. The current␣
↪behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can␣
↪also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```

Training Epoch 0
Epoch: 0 [batch 0/3, 0.00%]
        DistLoss: 0.693
Metrics:
Metrics:
        MAP: 0.645

Validation.
Metrics:
        MAP: 1.000

Training Epoch 1
Epoch: 1 [batch 0/3, 0.00%]
        DistLoss: 0.729
Metrics:
Metrics:
        MAP: 0.796

Validation.
Metrics:
        MAP: 1.000

Training Epoch 2
Epoch: 2 [batch 0/3, 0.00%]
        DistLoss: 0.331
Metrics:
Metrics:
        MAP: 0.830

Validation.
Metrics:
        MAP: 1.000

Best Model Appears at Epoch 0 with Validation score 1.000.
```

Load

Re-load a saved model with the load_model function:

[86]:
```
from opensoundscape.torch.models.cnn import load_model
model = load_model('./binary_train/best.model')
```

Note on saving models and version compatability

Loading a model in a different version of OpenSoundscape than the version that saved the model may not work. To use a model across different versions of OpenSoundscape, you should save the model.network's state dict using `model.save_weights(path)` as described in the "predicting with pre-trained models" tutorial. You can load weights from a saved state dict with `model.load_weights(path)`. We recommend saving both the full model object (`.save()`) and the raw weights (`.save_weights()`) for models you plan to use in the future.

Models saved with OpenSoundscape 0.4.x and 0.5.x can be loaded with `load_outdated_model` - but be sure to update the model.preprocessor after loading to match the settings used during training. See the tutorial "predicting with pre-trained models" for more details on loading models from earlier OpenSoundscape versions.

8.6 Predict using saved (or pre-trained) model

Using a saved or downloaded model to run predictions on audio files is as simple as

1. Loading a previously saved model
2. Generating a list of files for prediction
3. Running `model.predict()` on the preprocessor

[88]:
```
# load the saved model
model = load_model('./binary_train/best.model')

#predict on a dataset
scores = model.predict(field_recordings, activation_layer='sigmoid')
```

NOTE: See the tutorial "predicting with pre-trained models" for loading and using models from earlier OpenSoundscape versions

8.7 Continue training from saved model

Similar to predicting using a saved model, we can also continue to train a model after loading it from a saved file.

Note that `.load()` loads the entire model object, which includes optimizer parameters and learning rate parameters from the saved model, in addition to the network weights.

[89]:
```
# Create architecture
model = load_model('./binary_train/best.model')

# Continue training from the checkpoint where the model was saved
model.train(train_df,validation_df,save_path='.',epochs=0)
```

```

Best Model Appears at Epoch 0 with Validation score 0.000.
```

8.8 Next steps

You now have seen the basic usage of training CNNs with OpenSoundscape and generating predictions.

Additional tutorials you might be interested in are: * Custom preprocessing: how to change spectrogram parameters, modify augmentation routines, etc. * Custom training: how to modify and customize model training * Predict with pre-trained CNNs: details on how to predict with pre-trained CNNs. Much of this information was covered in the tutorial above, but this tutorial also includes information about using models made with previous versions of OpenSoundscape

Finally, clean up and remove files created during this tutorial:

[90]:
```python
import shutil
dirs = ['./multilabel_train', './binary_train', './woodcock_labeled_data']
for d in dirs:
    try:
        shutil.rmtree(d)
    except:
        pass
```

CHAPTER 9

Preprocessing audio samples with OpenSoundscape

`Preprocessors` in OpenSoundscape perform all of the preprocessing steps from loading a file from disk, up to providing a sample to the machine learning algorithm for training or prediction. They are designed to be flexible and customizable. These classes are used internally by classes such as `opensoundscape.torch.models.cnn.CNN` when (a) training a machine learning model in OpenSoundscape, or (b) making predictions with a machine learning model in OpenSoundscape.

`Datasets` are PyTorch's way of handling a list of inputs to preprocess. In OpenSoundscape, there are two built-in classes (AudioFileDataset and AudioSplittingDataset) which use a Preprocessor to generate samples from a list of file paths.

While the `CNN` class in OpenSoundscape contains a default Preprocessor, you may want to modify or create your own Preprocessor depending on the specific way you wish to generate samples. Preprocessors are designed to be flexible and modular, so that each step of the preprocessing pipeline can be modified or removed. This notebook demonstrates:

- preparation of audio data to be used by a preprocessor
- how "Actions" are strung together in a Preprocessor to define how samples are generated
- modifying the parameters of actions
- turning Actions on and off
- modifying the order and contents of a Preprocessor
- use of the `SpectrogramPreprocessor` class, including examples of:
  - modifying audio and spectrogram parameters
  - changing the output image shape
  - changing the output type
  - turning augmentation on and off
  - modifying augmentation parameters
  - using the "overlay" augmentation
- writing custom preprocessors and actions

it also uses the Dataset classes to demonstrate - how to load one sample per file path - how to load long audio files as a series of shorter clips

9.1 Modifying the preprocessor of the CNN class

When training a CNN model in OpenSoundscape, you will create an object of the CNN class. There are two ways to modify the preprocessing:

1) modify the model.preprocessor directly The model contains a preprocessor object that you can modify, for instance:

```
model.preprocessor.pipeline.bandpass.bypass = True
```

2) overwrite the preprocessor with a new one:

```
my_preprocessor = SpectrogramPreprocessor(....) #this tutorial will help you with␣
↪how to make this object
model.preprocessor = my_preprocessor
```

Note that if you want to create a preprocessor with overlay augmentation, it's easiest to use option 2 and initialize the preprocessor with an overlay_df.

Note on augmentations: - While training, the CNN class will use all actions in the preprocessor's pipeline. - When runing validation or prediction, by default, the CNN will bypass actions with action.is_augmentation==True.

First, import some packages.

[1]:
```
import warnings
```

[2]:
```
# Preprocessor classes are used to load, transform, and augment audio samples for use␣
↪in a machine learing model
from opensoundscape.preprocess.preprocessors import SpectrogramPreprocessor
from opensoundscape.torch.datasets import AudioFileDataset, AudioSplittingDataset


# helper function for displaying a sample as an image
from opensoundscape.preprocess.utils import show_tensor, show_tensor_grid

#other utilities and packages
import torch
import pandas as pd
from pathlib import Path
import numpy as np
import random
import subprocess
import IPython.display as ipd
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/scipy/__init__.py:
↪146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version␣
↪of SciPy (detected version 1.23.3
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Set up plotting

[3]:
```
#set up plotting
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.rcParams['figure.figsize']=[15,5] #for large visuals
%config InlineBackend.figure_format = 'retina'
```

Set manual seeds for pytorch and python. These ensure the training results are reproducible. You probably don't want to do this when you actually train your model, but it's useful for debugging.

```
[4]: torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

9.1.1 Preparing audio data

9.2 Download labeled audio files

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 `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.

Note: Once you have the data, you do not need to run this cell again.

```
[5]: subprocess.run(['curl','https://pitt.box.com/shared/static/
↪79fi7d715dulcldsy6uogz02rsn5uesd.gz','-L', '-o','woodcock_labeled_data.tar.gz']) #␣
↪Download the data
subprocess.run(["tar","-xzf", "woodcock_labeled_data.tar.gz"]) # Unzip the downloaded␣
↪tar.gz file
subprocess.run(["rm", "woodcock_labeled_data.tar.gz"]) # Remove the file after its␣
↪contents are unzipped
```

```
[5]: CompletedProcess(args=['rm', 'woodcock_labeled_data.tar.gz'], returncode=0)
```

9.3 Load dataframe of files and labels

We need a dataframe with file paths in the index, so we manipulate the included one_hot_labels.csv slightly:

```
[6]: # load one-hot labels dataframe
labels = pd.read_csv('./woodcock_labeled_data/one_hot_labels.csv').set_index('file')

# prepend the folder location to the file paths
labels.index = pd.Series(labels.index).apply(lambda f: './woodcock_labeled_data/'+f)

#inspect
labels.head()
```

```
[6]:                                                  present  absent
file
./woodcock_labeled_data/d4c40b6066b489518f8da83...        1       0
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...        0       1
./woodcock_labeled_data/79678c979ebb880d5ed6d56...        1       0
```

(continues on next page)

(continued from previous page)

```
./woodcock_labeled_data/49890077267b569e142440f...        1        0
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...        1        0
```

9.3.1 Intro to Preprocessors

Preprocessors prepare samples for use by machine learning algorithms by performing a sequential procedure on each sample, like a recipe. The procedure is defined by a **Pipeline** which contains a sequential set of steps called **Actions**. There are 3 important characteristics of Preprocessors and Actions:

- [1] A Preprocessor has a `pipeline` which defines a list of `Actions` to perform on each sample
- [2] `Actions` contain parameters that modify their behavior in the attribute `.params`. You can modify parameter values directly or use the action's .set() method to change parameter values.
- [3] Preprocessing can be performed with or without augmentation. The Preprocessor's `.bypass_augmentations` boolean variable will determine whether Actions in the pipeline with attribute `.is_augmentation==True` are performed or bypassed
- [3] SpecPreprocessor (the default Preprocessor class) loads audio in two distinct modes: (a) loading one sample per file, and (b) spliting files into clips, and creating a sample from each clip. You can see examples of each mode below. By default, OpenSoundscape's CNN class loads one sample per file during training and splits files into clips during prediction.

In this notebook, you will see how to edit, add, remove, and bypass Actions in the `pipeline` to modify the Preprocessing procedure.

The CNN class in OpenSoundscape has an internal Preprocessor object which it use to generate samples during training, validation, and prediction. We can modify or overwrite the cnn model's preprocessor object if we want to change how it generates samples.

The starting point for most preprocessors will be the `SpecPreprocessor` class, which loads audio files, creates spectrograms from the audio, performs various augmentations, and returns a pytorch Tensor.

9.4 Initialize preprocessor

We need to tell the preprocessor the duration (in seconds) of each sample it should create.

[7]:
```
pre = SpectrogramPreprocessor(sample_duration=2.0)
```

9.4.1 Initialize a Dataset

A Dataset pairs a set of samples (possibly including labels) with a Preprocessor

The Dataset draws samples from it's `.df` attribute which must be a very specific dataframe:

- the index of the dataframe provides paths to audio samples
- the columns are the class names
- the values are 0 (absent/False) or 1 (present/True) for each sample and each class.

For example, we've set up the labels dataframe with files as the index and classes as the columns, so we can use it to make an instance of `SpecPreprocessor`:

[8]:
```
dataset = AudioFileDataset(labels,pre)
```

9.5 Generate a sample from a Dataset

We can ask a dataset for a specific sample using its numeric index, like accessing an element of a list. Each sample is a dictionary with two keys: 'X', the Tensor of the sample, and 'y', the Tensor of labels of the sample. The shape of 'X' is [channels, height, width] and the shape of 'y' is [number of classes].

[9]:
```
dataset[0] #loads and preprocesses the sample at row 0 of dataset.df
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

[9]:
```
{'X': tensor([[[ 0.0000,  0.0000,  0.0000,  ..., -0.3139, -0.4861, -0.4208],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.3800, -0.3729, -0.4864],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.4506, -0.3056, -0.5758],
          ...,
          [ 0.0000,  0.0000,  0.0000,  ...,  0.4784,  0.4597,  0.3293],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

         [[ 0.0000,  0.0000,  0.0000,  ..., -0.3006, -0.4739, -0.4238],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.4015, -0.3679, -0.4939],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.4587, -0.3104, -0.5777],
          ...,
          [ 0.0000,  0.0000,  0.0000,  ...,  0.4656,  0.4625,  0.3283],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

         [[ 0.0000,  0.0000,  0.0000,  ..., -0.3303, -0.4976, -0.3977],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.3992, -0.3732, -0.4985],
          [ 0.0000,  0.0000,  0.0000,  ..., -0.4614, -0.2974, -0.5799],
          ...,
          [ 0.0000,  0.0000,  0.0000,  ...,  0.4731,  0.4623,  0.3363],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]]]),
 'y': tensor([1, 0])}
```

Using a helper function, we can easily visualze a set of samples on a grid. We *highly* recommend inspecting your preprocessed samples in this way before training or predicting with a machine learning model. By inspecting the samples, you can confirm that your labeled data is reasonable and that the preprocessing is representing your samples in a reasonable way.

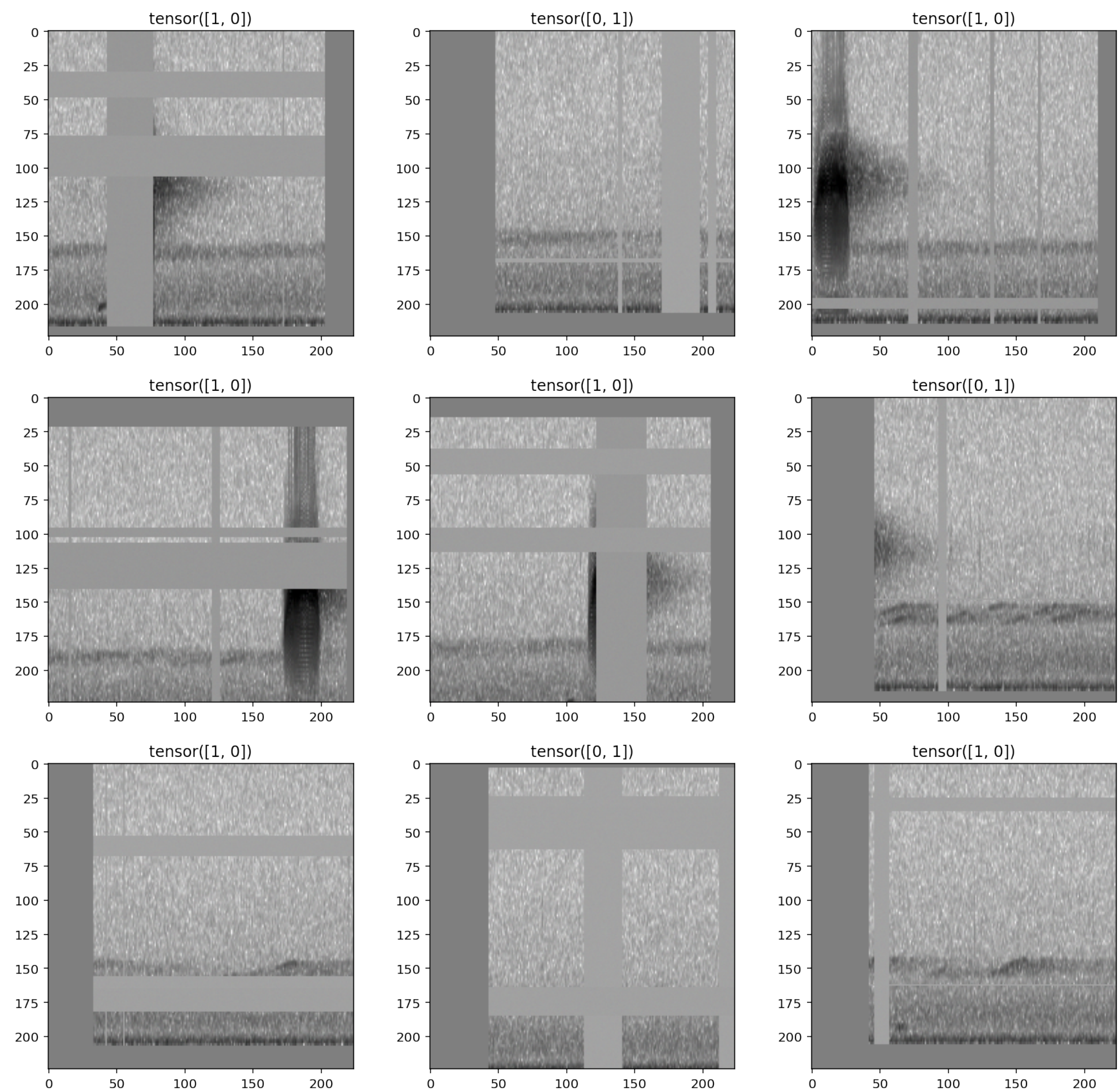
[10]:
```
from opensoundscape.preprocess.utils import show_tensor_grid

pre = SpectrogramPreprocessor(sample_duration=2.0)
dataset = AudioFileDataset(labels,pre)

tensors = [dataset[i]['X'] for i in range(9)]
sample_labels = [dataset[i]['y'] for i in range(9)]

_ = show_tensor_grid(tensors,3,labels=sample_labels)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

(continues on next page)

(continued from previous page)

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/pylab/
↪config.py:79: DeprecationWarning: InlineBackend._figure_format_changed is␣
↪deprecated in traitlets 4.1: use @observe and @unobserve instead.
  def _figure_format_changed(self, name, old, new):
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```



let’s repeat the exercise of inspecting preprocessed samples, this time without augmentation

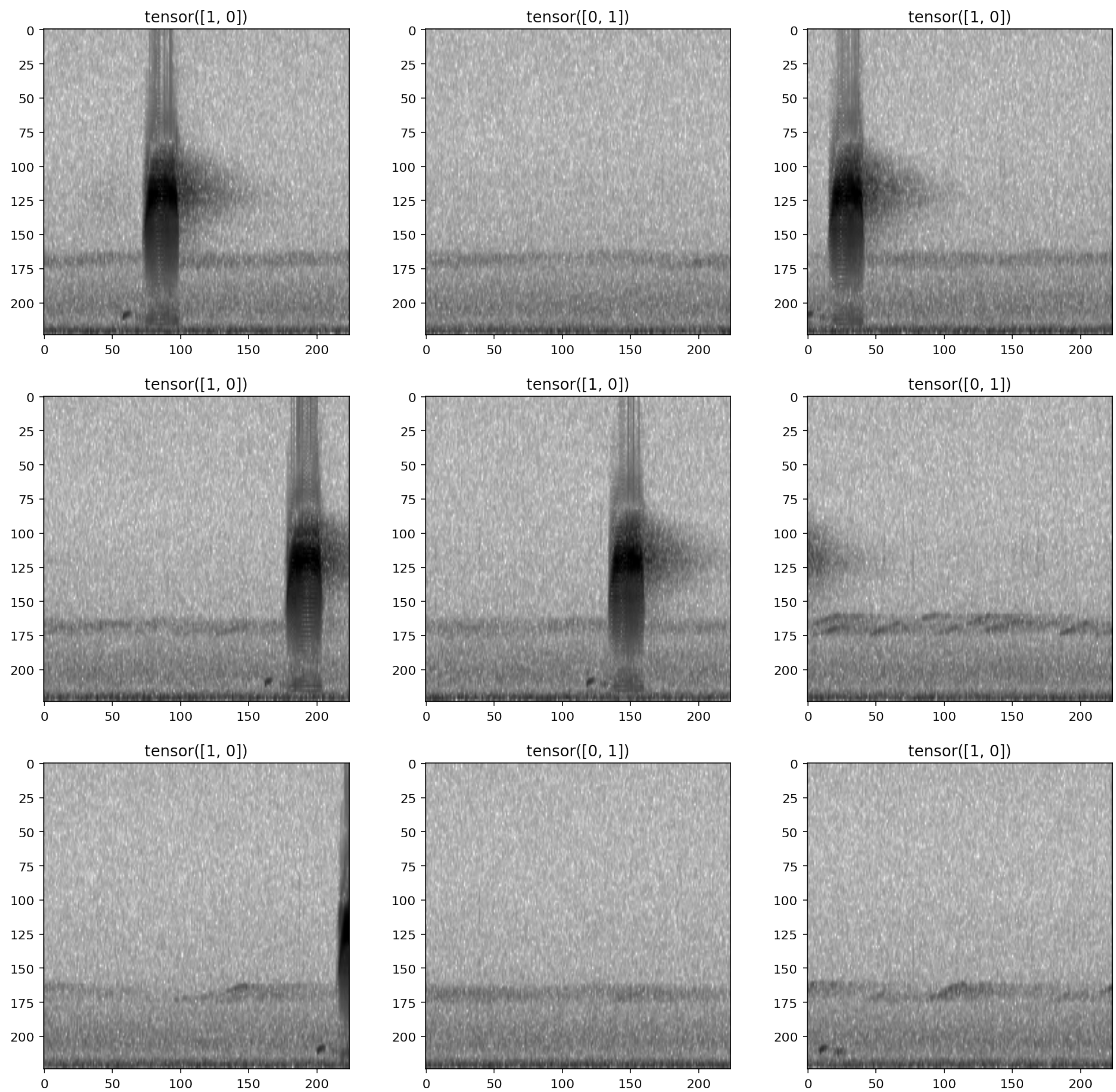
[11]:
```
dataset.bypass_augmentations = True

tensors = [dataset[i]['X'] for i in range(9)]
sample_labels = [dataset[i]['y'] for i in range(9)]

_ = show_tensor_grid(tensors,3,labels=sample_labels)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

(continues on next page)

(continued from previous page)

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



during machine learning tasks with Pytorch, a `DataLoader` is often used on top of a Dataset to "batch" samples - that is, to prepare multiple samples at once. A batch returned by a DataLoader will have an extra leading dimension for both 'X' and 'y'; for instance, a `batch_size` of 16 would produce 'X' withs shape [16, 3, 224, 224] for 3-channel 224x224 tensors and 'y' with shape [16, 5] if the labels contain 5 classes (columns). OpenSoundscape uses DataLoaders internally to create batches of samples during CNN training and prediction.

9.6 Subset samples from a Dataset

Preprocessors allow you to select a subset of samples using `sample()` and `head()` methods (like Pandas DataFrames).

(note that these methods subset files from the index, they do not subset individual clips from files)

```
[12]: len(dataset)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

```
[12]: 29
```

Select the first 10 samples (non-random)

```
[13]: len(dataset.head(10))
```

```
[13]: 10
```

Randomly select an absolute number of samples

```
[14]: len(dataset.sample(n=10))
```

```
[14]: 10
```

Randomly select a fraction of samples

```
[15]: len(dataset.sample(frac=0.5))
```

```
[15]: 14
```

9.6.1 Loading many fixed-duration samples from longer audio files

When preprocessing should result in many fixed-length samples per input file, instead of one sample per file, we use `AudioSplittingDataset` instead of `AudioFileDataset`. This dataset can be customized with parameters for:

- fractional overlap between consecutive samples
- how to handle remaining audio at the end of a file (if it is shorter than the desired sample duration)

The CNN.predict() function uses `AudioSplittingDataset` internally, so that the user can specify long audio file paths and get back predictions on fixed-length clips. (If one sample per file is desired, you can pass the argument `split_files_into_clips=False` to CNN.predict)

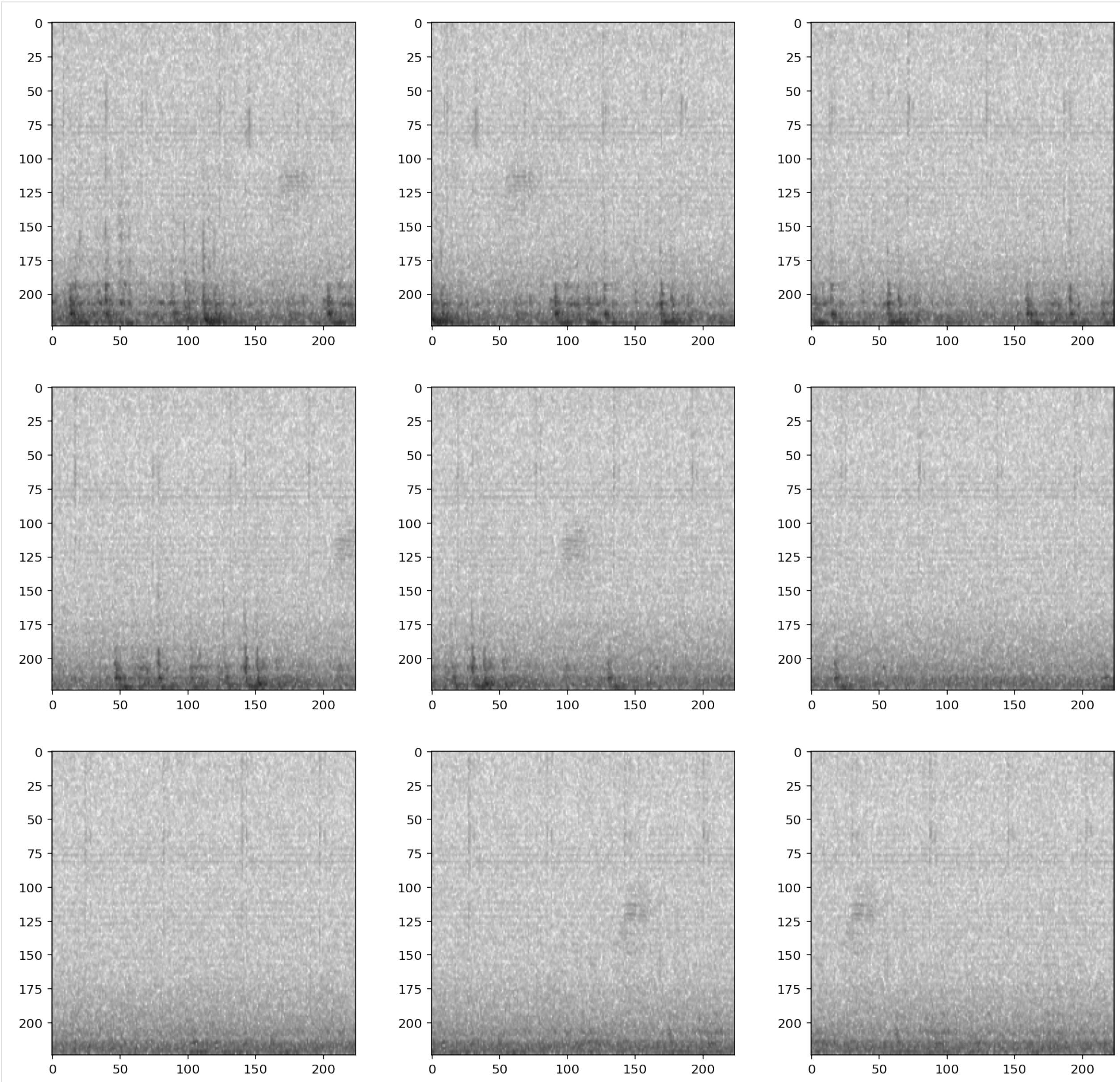
Here's an example of how to use `AudioSplittingDataset` to create several samples from a long audio file:

(Note that you never have to manually create AudioSplittingDataset or AudioFileDataset objects to train and predict with the CNN class, they are created internally.)

```
[16]: prediction_df = pd.DataFrame(index=['./woodcock_labeled_data/field_data/60s_field_
↪data_sample_1.wav'])
```

[17]:
```
pre = SpectrogramPreprocessor(sample_duration=2.0)
splitting_dataset = AudioSplittingDataset(prediction_df,pre,overlap_fraction=0.5)
splitting_dataset.bypass_augmentations = True

#get the first 9 samples and plot them
tensors = [splitting_dataset[i]['X'] for i in range(9)]

_ = show_tensor_grid(tensors,3)
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



9.6.2 Pipelines and actions

Each Preprocessor class has a `pipeline` which is an ordered set of operations that are performed on each sample, in the form of a pandas.Series object. Each element of the series is an object of class Action (or one of its subclasses) and represents a transformation on the sample.

9.7 About Pipelines

The preprocessor's Pipeline is the ordered list of Actions that the preprocessor performs on each sample.

- The Pipeline is stored in the `preprocessor.pipeline` attribute.

- You can modify the contents or order of Preprocessor Actions by overwriting the preprocessor's `.pipeline` attribute. When you modify this attribute, you must provide pd.Series with elements name:Action, where each Action is an instance of a class that sub-classes `opensoundscape.preprocess.BaseAction`.

Let's Inspect the current pipeline of our preprocessor.

[18]:
```
# inspect the current pipeline (ordered sequence of Actions to take)
preprocessor = SpectrogramPreprocessor(sample_duration=2)
preprocessor.pipeline
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

[18]:
```
load_audio           Action calling <bound method Audio.from_file o...
random_trim_audio    Augmentation Action calling <function trim_aud...
trim_audio           Action calling <function trim_audio at 0x7f0cf...
to_spec              Action calling <bound method Spectrogram.from_...
bandpass             Action calling <function Spectrogram.bandpass ...
to_img               Action calling <function Spectrogram.to_image ...
time_mask            Augmentation Action calling <function time_mas...
frequency_mask       Augmentation Action calling <function frequenc...
add_noise            Augmentation Action calling <function tensor_a...
rescale              Action calling <function scale_tensor at 0x7f0...
random_affine        Augmentation Action calling <function torch_ra...
dtype: object
```

9.8 About actions

Each element of the preprocessor's `pipeline` (a pd.Series) contains a name (string) and an action (Action)

- Each Action takes a sample (and its labels), performs some transformation to them, and returns the sample (and its labels).
- You can generate an Action based on a function like this : Action(fn=my_function, other parameters…). The function you pass (`my_function` in this case) must expect the sample as the first argument. It can then take additional parameters. For instance, if we define the function:

  ```
  def multiply(x,n):
      return x*n
  ```

  then we can create an action to multiply by 3 with `action=Action(fn=multiply,n=3)`
- Any customizable parameters for performing the Action are stored in a dictionary, `.params`. These parameters can be modified directly (e.g. `Action.params.param1=value1`) or using the Action's `.set()` method (e.g. `action.set(param=value, param2=value2, ...)`)
- You can bypass an action in a pipeline by changing `Action.bypass` to `True`
- You can declare whether an Action is an augmentation (should not be performed if bypass_augmentation=True) using its `.is_augmentation` boolean attribute

9.8.1 Modifying Actions

9.9 View default parameters for an Action

the `.params` attribute of an Action is a pandas Series containing parameters that can be modified

[19]:
```
#since the pipeline is a series, we can access elements like pipeline.to_spec as well␣
↪as pipeline['to_spec']
preprocessor.pipeline.to_spec.params
```

[19]:
```
window_type                   hann
window_samples                None
window_length_sec             None
overlap_samples               None
overlap_fraction              None
fft_size                      None
decibel_limits         (-100, -20)
dB_scale                      True
scaling                   spectrum
dtype: object
```

9.10 Modify Action parameters

we can modify parameters with the Action's .set() method:

[20]:
```
preprocessor.pipeline.to_spec.set(dB_scale=False)
```

or by accessing the parameter directly (params is a pandas Series)

[21]:
```
preprocessor.pipeline.to_spec.params.window_samples = 512
preprocessor.pipeline.to_spec.params['overlap_fraction'] = 0.75

preprocessor.pipeline.to_spec.params
```

[21]:
```
window_type                   hann
window_samples                 512
window_length_sec             None
overlap_samples               None
overlap_fraction              0.75
fft_size                      None
decibel_limits         (-100, -20)
dB_scale                     False
scaling                   spectrum
dtype: object
```

9.11 Bypass actions

Actions can be bypassed by changing the attribute `.bypass=True`. A bypassed action is never performed regardless of the `.perform_augmentations` attribute.

[22]:
```
preprocessor = SpectrogramPreprocessor(sample_duration=2.0)
```

(continues on next page)

(continued from previous page)

```
#turn off augmentations other than noise
preprocessor.pipeline.add_noise.bypass=True
preprocessor.pipeline.time_mask.bypass=True
preprocessor.pipeline.frequency_mask.bypass=True

#printing the pipeline will show which actions are bypassed
preprocessor.pipeline
```

[22]:

```
load_audio              Action calling <bound method Audio.from_file o...
random_trim_audio       Augmentation Action calling <function trim_aud...
trim_audio              Action calling <function trim_audio at 0x7f0cf...
to_spec                 Action calling <bound method Spectrogram.from_...
bandpass                Action calling <function Spectrogram.bandpass ...
to_img                  Action calling <function Spectrogram.to_image ...
time_mask               ## Bypassed ## Augmentation Action calling <fu...
frequency_mask          ## Bypassed ## Augmentation Action calling <fu...
add_noise               ## Bypassed ## Augmentation Action calling <fu...
rescale                 Action calling <function scale_tensor at 0x7f0...
random_affine           Augmentation Action calling <function torch_ra...
dtype: object
```

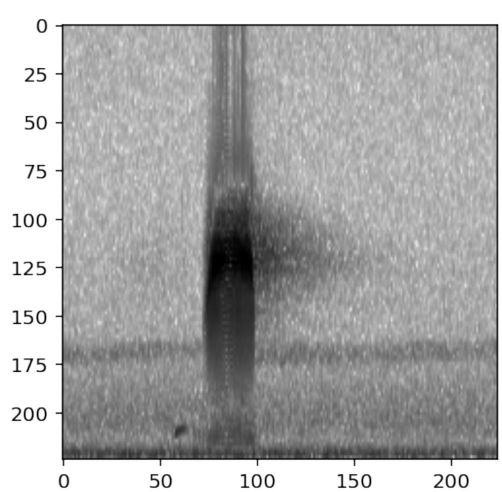
create a Dataset with this preprocessor and our label dataframe

[23]:

```
dataset = AudioFileDataset(labels,preprocessor)

print('random affine off')
preprocessor.pipeline.random_affine.bypass = True
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
plt.show()

print('random affine on')
preprocessor.pipeline.random_affine.bypass = False
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

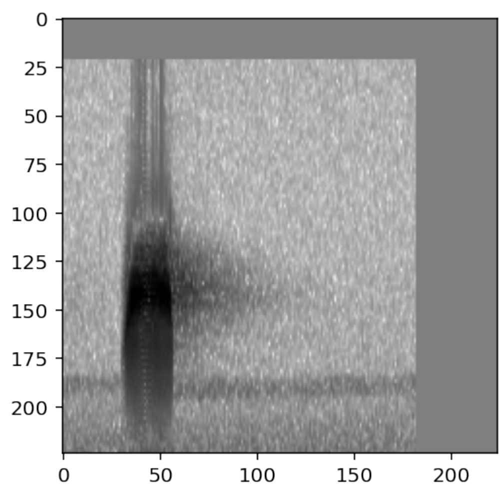
```
random affine off
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

```
random affine on
```



To view whether an individual Action in a pipeline is on or off, inspect its `bypass` attribute:

```
[24]: # The AudioLoader Action that is still on
      preprocessor.pipeline.load_audio.bypass
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

```
[24]: False
```

```
[25]: # The frequency_mask Action that we turned off
      preprocessor.pipeline.frequency_mask.bypass
```

```
[25]: True
```

9.11.1 Modifying the pipeline

Sometimes, you may want to change the order or composition of the Preprocessor's pipeline. You can simply overwrite the `.pipeline` attribute, as long as it is a pandas Series of names:Actions

9.12 Example: return Spectrogram instead of Tensor

Here's an example where we replace the pipeline with one that just loads audio and converts it to a Spectrogram, returning a Spectrogram instead of a Tensor:

```
[26]: #initialize a preprocessor
      preprocessor = SpectrogramPreprocessor(2.0)
```

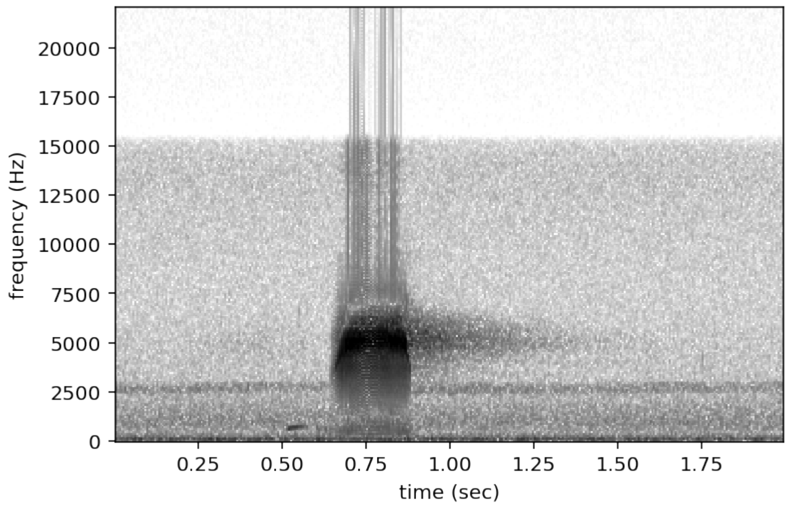
(continues on next page)

(continued from previous page)

```
print('original pipeline:')
[print(p) for p in pre.pipeline]

#overwrite the pipeline with a slice of the original pipeline
print('\nnew pipeline:')
preprocessor.pipeline = preprocessor.pipeline[0:4]
[print(p) for p in preprocessor.pipeline]

print('\nWe now have a preprocessor that returns Spectrograms instead of Tensors:')
dataset = AudioFileDataset(labels,preprocessor)
print(f"Type of returned sample: {type(dataset[0]['X'])}")
dataset[0]['X'].plot()
```

```
original pipeline:
Action calling <bound method Audio.from_file of <class 'opensoundscape.audio.Audio'>>
Augmentation Action calling <function trim_audio at 0x7f0cfbd90310>
Action calling <function trim_audio at 0x7f0cfbd90310>
Action calling <bound method Spectrogram.from_audio of <class 'opensoundscape.
↪spectrogram.Spectrogram'>>
Action calling <function Spectrogram.bandpass at 0x7f0cff890dc0>
Action calling <function Spectrogram.to_image at 0x7f0cff8930d0>
Augmentation Action calling <function time_mask at 0x7f0cfbd9c430>
Augmentation Action calling <function frequency_mask at 0x7f0cfbd9c4c0>
Augmentation Action calling <function tensor_add_noise at 0x7f0cfbd9c550>
Action calling <function scale_tensor at 0x7f0cfbd9c3a0>
Augmentation Action calling <function torch_random_affine at 0x7f0cfbd9c280>

new pipeline:
Action calling <bound method Audio.from_file of <class 'opensoundscape.audio.Audio'>>
Augmentation Action calling <function trim_audio at 0x7f0cfbd90310>
Action calling <function trim_audio at 0x7f0cfbd90310>
Action calling <bound method Spectrogram.from_audio of <class 'opensoundscape.
↪spectrogram.Spectrogram'>>

We now have a preprocessor that returns Spectrograms instead of Tensors:
Type of returned sample: <class 'opensoundscape.spectrogram.Spectrogram'>
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



9.12.1 Analyzing/debugging the pipeline

In order to debug the Preprocessor's pipeline you can utilize the `trace` argument to save and review the output of `action` step in the pipeline as part of the sample information returned by the preprocessor.

[27]:
```
# initialize a preprocessor
preprocessor = SpectrogramPreprocessor(2.0)
pre.pipeline
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

[27]:
```
load_audio            Action calling <bound method Audio.from_file o...
random_trim_audio     Augmentation Action calling <function trim_aud...
trim_audio            Action calling <function trim_audio at 0x7f0cf...
to_spec               Action calling <bound method Spectrogram.from_...
bandpass              Action calling <function Spectrogram.bandpass ...
to_img                Action calling <function Spectrogram.to_image ...
time_mask             Augmentation Action calling <function time_mas...
frequency_mask        Augmentation Action calling <function frequenc...
add_noise             Augmentation Action calling <function tensor_a...
rescale               Action calling <function scale_tensor at 0x7f0...
random_affine         Augmentation Action calling <function torch_ra...
dtype: object
```

[28]:
```
# pass a sample through the preprocessor's pipeline
x, sample_info = preprocessor.forward(labels.iloc[0], trace=True)
sample_info["_trace"]
```

```
/home/jatink/repos/opensoundscape/opensoundscape/preprocess/preprocessors.py:155:␣
↪DeprecationWarning: The default dtype for empty Series will be 'object' instead of
↪'float64' in a future version. Specify a dtype explicitly to silence this warning.
  "_trace": pd.Series(index=self.pipeline.index) if trace else None,
```

(continues on next page)

(continued from previous page)

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

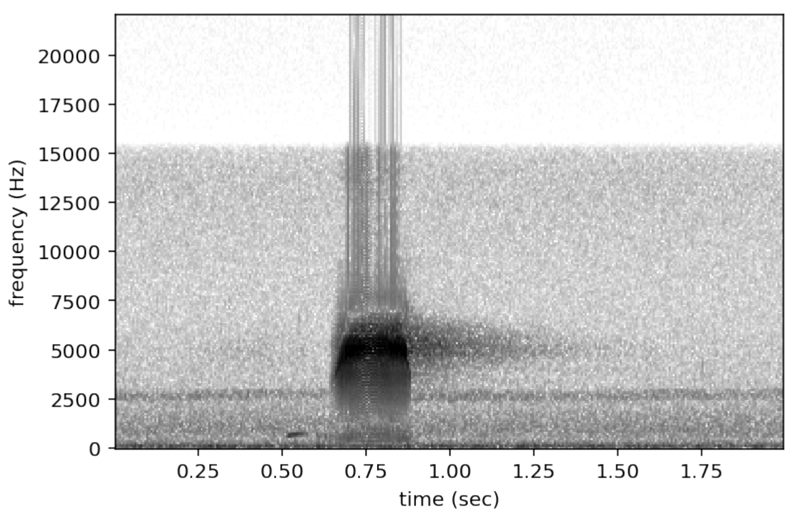
[28]:
```
load_audio                  <Audio(samples=(88200,), sample_rate=44100)>
random_trim_audio           <Audio(samples=(88200,), sample_rate=44100)>
trim_audio                  <Audio(samples=(88200,), sample_rate=44100)>
to_spec                <Spectrogram(spectrogram=(257, 343), frequenci...
bandpass               <Spectrogram(spectrogram=(129, 343), frequenci...
to_img                 [[[tensor(0.3032), tensor(0.3257), tensor(0.30...
time_mask              [[[tensor(0.3032), tensor(0.3257), tensor(0.30...
frequency_mask         [[[tensor(0.3032), tensor(0.3257), tensor(0.30...
add_noise              [[[tensor(0.3075), tensor(0.3274), tensor(0.30...
rescale                [[[tensor(-0.3850), tensor(-0.3453), tensor(-0...
random_affine          [[[tensor(0.), tensor(0.), tensor(0.), tensor(...
dtype: object
```

9.13 analyse the output at steps of interest

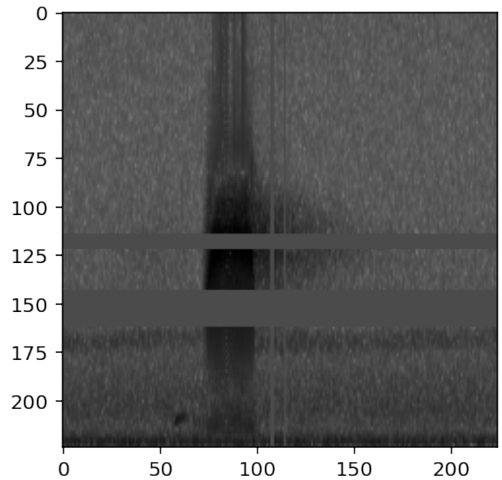
[29]:
```
# Initial audio
audio = sample_info["_trace"]["load_audio"]
ipd.display(ipd.Audio(audio.samples, rate=audio.sample_rate, autoplay=False),␣
↪clear=True)
```

```
<IPython.lib.display.Audio object>
```

[30]:
```
# Initial spectrogram
sample_info["_trace"]["to_spec"].plot()
```



[31]:
```
# After applyin frequency mask
show_tensor(sample_info["_trace"]["frequency_mask"],invert=True,transform_from_zero_
↪centered=True)
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200

CHAPTER 10

adding the preprocessor to a CNN

You can always overwrite the preprocessor of a CNN model object with a new one:

```
my_preprocessor = SpectrogramPreprocessor(....)
...
model.preprocessor = my_preprocessor
```

WARNING: Be careful! If your new preprocessor has a different sample duration (eg 3 seconds instead of 2) or shape (eg [100,100,3] instead of [224,224,1]), these new values will also take effect when using the CNN.

The right choice of preprocessing depends heavily on the characteristics of the sounds you wish to study. The best way to tune preprocessing parameters is to visually inspect samples created by your preprocessing procedure and tweak parameters to achieve visual clarity of the sounds of interest in your samples. We find these heuristics to be a good starting point:

- The duration of a sample should be approximately 2-5x the duration of the target sound. For instance, a very short nocturnal flight call lasting 0.1 seconds might be best visualized with a 0.3 second sample_duration. Meahwhile, a 10-second bout of ruffed grouse drumming might deserve a 20 second sample_duartion.
- The frequency range of a sample should be wider than the target sound, but not by more than 1 order of magnitude. For instance, sounds that are low-pitched will be more clearly visualized when bandpassing a spectrogram to the low frequencies. If you use a 0-10,000 Hz spectrogram for a 500 Hz target sound, your target sound will only occupy a small fraction of your sample.
- Spectrogram parameters should be matched to the temporal or spectral features of the target sound. Modify the Spectorgram's window_samples to achieve high enough time resolution (lower value of window_samples) or frequency resolution (higher value of window_samples) to see features of your target sound clearly on the resulting sample. For example, a rapid trill with a pulse repetition rate of 50 Hz will only be distinctive on a spectrogram if the Spectrogram windows are less than `1/(50*2) = 0.01` seconds in duration. On the other hand, visualizing a distinctive harmonic "ladder" structure of a nasal sound might require long spectrogram windows which will increase frequency resolution.

Augmentations are Actions that are only performed during training, not during prediction. These actions manipulate the sample in some randomized way, so that each time the same sample is provided to the model as training data, the actual values of the sample are different. This prevents over-training of a model on a training set and effectively increases the size of a training dataset. In general, you can expect that a basic set of augmentations (such as those included by default in the SpecPreprocessor and CNN classes) will be necessary to train a useful machine learning model. In particular, "overlay" augmentations which blend together multiple samples often increase the generalizability (transferability) of a model. You might choose to use audio from your target system (for instance, field recordings at your study site) to make the training data look more similar to the data that the model will be applied to.

Below are various examples of how to modify parameters of the Actions to achieve different preprocessing outcomes.

10.1 Modify the sample rate

Resample all loaded audio to a specified rate during the load_audio action

[32]:
```
pre = SpectrogramPreprocessor(sample_duration=2)

pre.pipeline.load_audio.set(sample_rate=24000)
```

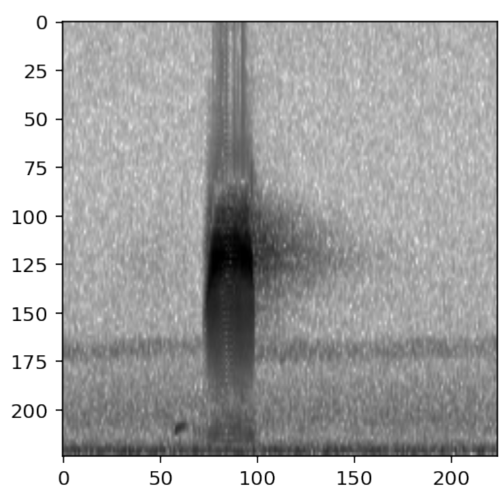
10.2 Modify spectrogram window length and overlap

(see Spectrogram.from_audio() for detailed documentation)

[33]:
```
dataset = AudioFileDataset(labels,SpectrogramPreprocessor(sample_duration=2))
dataset.bypass_augmentations=True

print('default parameters:')
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
plt.show()

print('high time resolution, low frequency resolution:')
dataset.preprocessor.pipeline.to_spec.set(window_samples=64)

show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

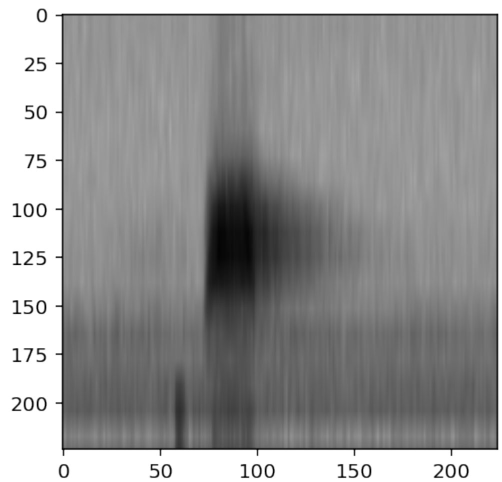
```
default parameters:
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



```
high time resolution, low frequency resolution:
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



10.3 Bandpass spectrograms

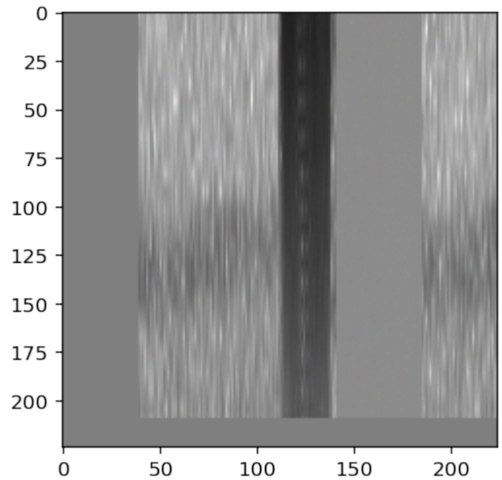
Trim spectrograms to a specified frequency range:

[34]:
```
dataset = AudioFileDataset(labels, SpectrogramPreprocessor(2.0))

print('default parameters:')
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)

print('bandpassed to 2-4 kHz:')
dataset.preprocessor.pipeline.bandpass.set(min_f=2000,max_f=4000)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
default parameters:
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

```
bandpassed to 2-4 kHz:
```

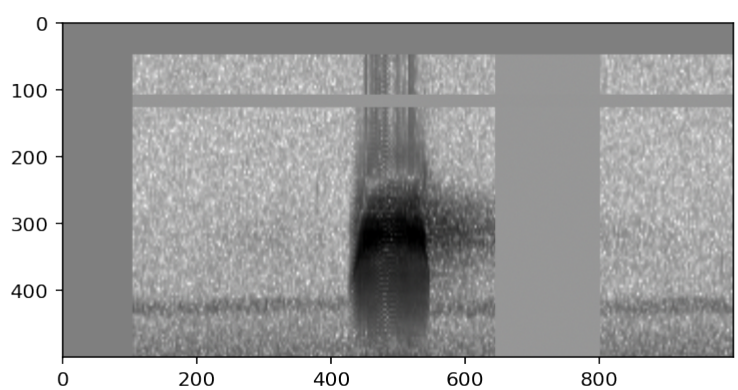


10.4 Change the output shape

Change the shape of the output sample - note that the shape argument expects (height, width), not (width, height)

[35]:
```
dataset = AudioFileDataset(labels, SpectrogramPreprocessor(2.0))

dataset.preprocessor.pipeline.to_img.set(shape=[500,1000])
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```

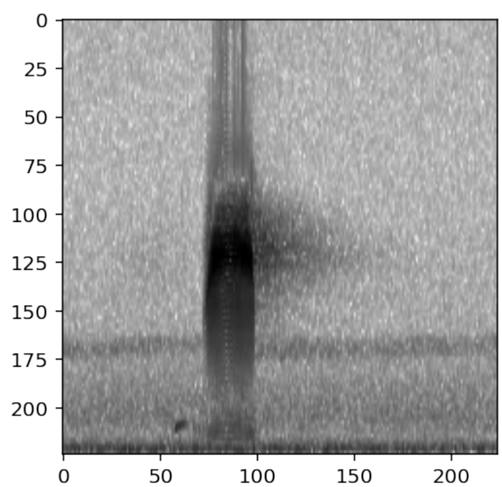


10.5 Turn all augmentation on or off

augmentation is controlled by the `preprocessor.bypass_augmentation` boolean (aka True/False) variable. By default, augmentations are performed. A CNN will internally manipulate this attribute to perform augmentations during training but not during validation or prediction.

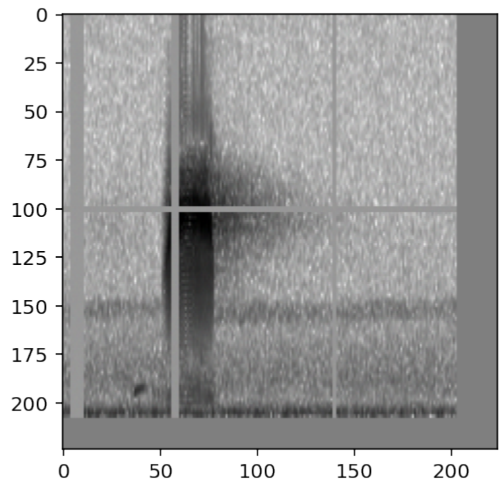
[36]:
```
dataset = AudioFileDataset(labels, SpectrogramPreprocessor(2.0))

dataset.bypass_augmentations = True
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



[37]:
```
dataset.bypass_augmentations = False
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```



10.6 Modify augmentation parameters

SpectrogramPreprocessor includes several augmentations with customizable parameters. Here we provide a couple of illustrative examples - see any action's documentation for details on how to use its parameters.

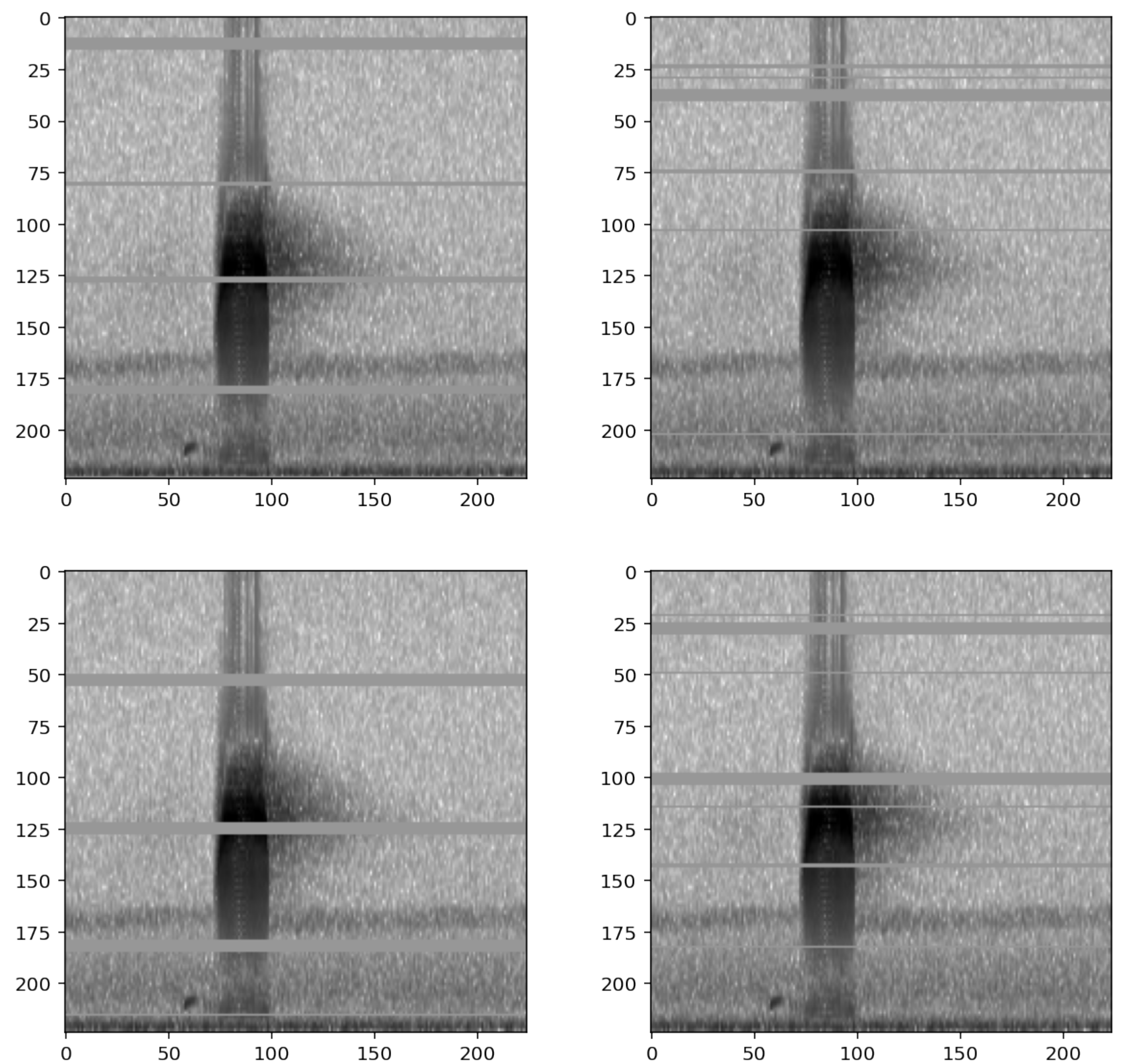
[38]:
```
#initialize a preprocessor
preprocessor = SpectrogramPreprocessor(2.0)

#turn off augmentations other than overlay
preprocessor.pipeline.random_affine.bypass=True
preprocessor.pipeline.time_mask.bypass=True
preprocessor.pipeline.add_noise.bypass=True

# allow up to 20 horizontal masks, each spanning up to 0.1x the height of the image.
preprocessor.pipeline.frequency_mask.set(max_width = 0.03, max_masks=20)

#preprocess the same sample 4 times
dataset = AudioFileDataset(labels,preprocessor)
tensors = [dataset[0]['X'] for i in range(4)]
fig = show_tensor_grid(tensors,2)
plt.show()
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

(continues on next page)

(continued from previous page)

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



turn off frequency mask and turn on gaussian noise

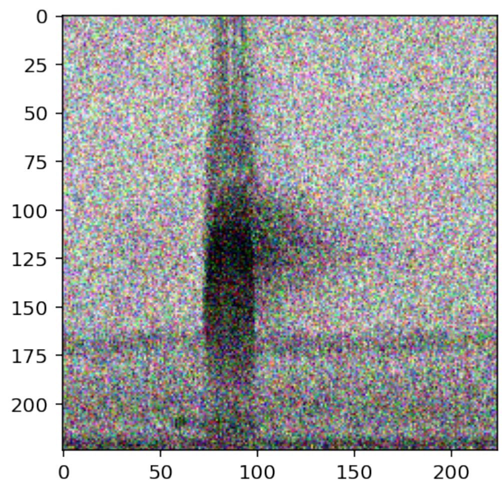
[39]:
```
dataset.preprocessor.pipeline.add_noise.bypass = False
dataset.preprocessor.pipeline.frequency_mask.bypass =True

# increase the intensity of gaussian noise added to the image
dataset.preprocessor.pipeline.add_noise.set(std=0.2)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
```

(continues on next page)

(continued from previous page)

```
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```



10.7 remove an action by its name

[40]:
```
preprocessor.remove_action('add_noise')
preprocessor.pipeline
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

[40]:
```
load_audio           Action calling <bound method Audio.from_file o...
random_trim_audio    Augmentation Action calling <function trim_aud...
trim_audio           Action calling <function trim_audio at 0x7f0cf...
to_spec              Action calling <bound method Spectrogram.from_...
bandpass             Action calling <function Spectrogram.bandpass ...
to_img               Action calling <function Spectrogram.to_image ...
time_mask            ## Bypassed ## Augmentation Action calling <fu...
frequency_mask       ## Bypassed ## Augmentation Action calling <fu...
rescale              Action calling <function scale_tensor at 0x7f0...
random_affine        ## Bypassed ## Augmentation Action calling <fu...
dtype: object
```

10.8 add an action at a specific position

specify the action in the pipeline you want to insert before or after

[41]:
```python
from opensoundscape.preprocess.actions import Action, tensor_add_noise

preprocessor.insert_action(
    action_index='add_noise_NEW', #give it a name
    action=Action(tensor_add_noise,std=0.01), #the action object
    after_key='to_img', #where to put it (can also use before_key=...)
)
```

[42]:
```python
preprocessor.pipeline
```

[42]:
```
load_audio           Action calling <bound method Audio.from_file o...
random_trim_audio    Augmentation Action calling <function trim_aud...
trim_audio           Action calling <function trim_audio at 0x7f0cf...
to_spec              Action calling <bound method Spectrogram.from_...
bandpass             Action calling <function Spectrogram.bandpass ...
to_img               Action calling <function Spectrogram.to_image ...
add_noise_NEW        Action calling <function tensor_add_noise at 0...
time_mask            ## Bypassed ## Augmentation Action calling <fu...
frequency_mask       ## Bypassed ## Augmentation Action calling <fu...
rescale              Action calling <function scale_tensor at 0x7f0...
random_affine        ## Bypassed ## Augmentation Action calling <fu...
dtype: object
```

it will complain if you use a non-unique index

[43]:
```python
from opensoundscape.preprocess.actions import Action, tensor_add_noise

try:
    preprocessor.insert_action(
        action_index='add_noise_NEW', #using the same name as a currentaction will␣
↪lead to an AssertionError
        action=Action(tensor_add_noise,std=0.01), #the action object
        after_key='to_img', #where to put it (can also use before_key=...)
    )
except AssertionError:
    print("raised Assertion Error, as expected")
```

```
raised Assertion Error, as expected
```

10.9 Overlay augmentation

Overlay is a powerful Action that allows additional samples to be overlayed or blended with the original sample.

The additional samples are chosen from the `overlay_df` that is provided to the preprocessor when it is initialized. The index of the `overlay_df` must be paths to audio files. The dataframe can be simply an index containing audio files with no other columns, or it can have the same columns as the sample dataframe for the preprocessor.

Samples for overlays are chosen based on their class labels, according to the parameter `overlay_class`:

- `None` - Randomly select any file from `overlay_df`
- `"different"` - Select a random file from `overlay_df` containing none of the classes this file contains
- specific class name - always choose files from this class

By default, the overlay Action does **not** change the labels of the sample it modifies. However, if you wish to add the labels from overlayed samples to the original sample's labels, you can set `update_labels=True` (see example below).

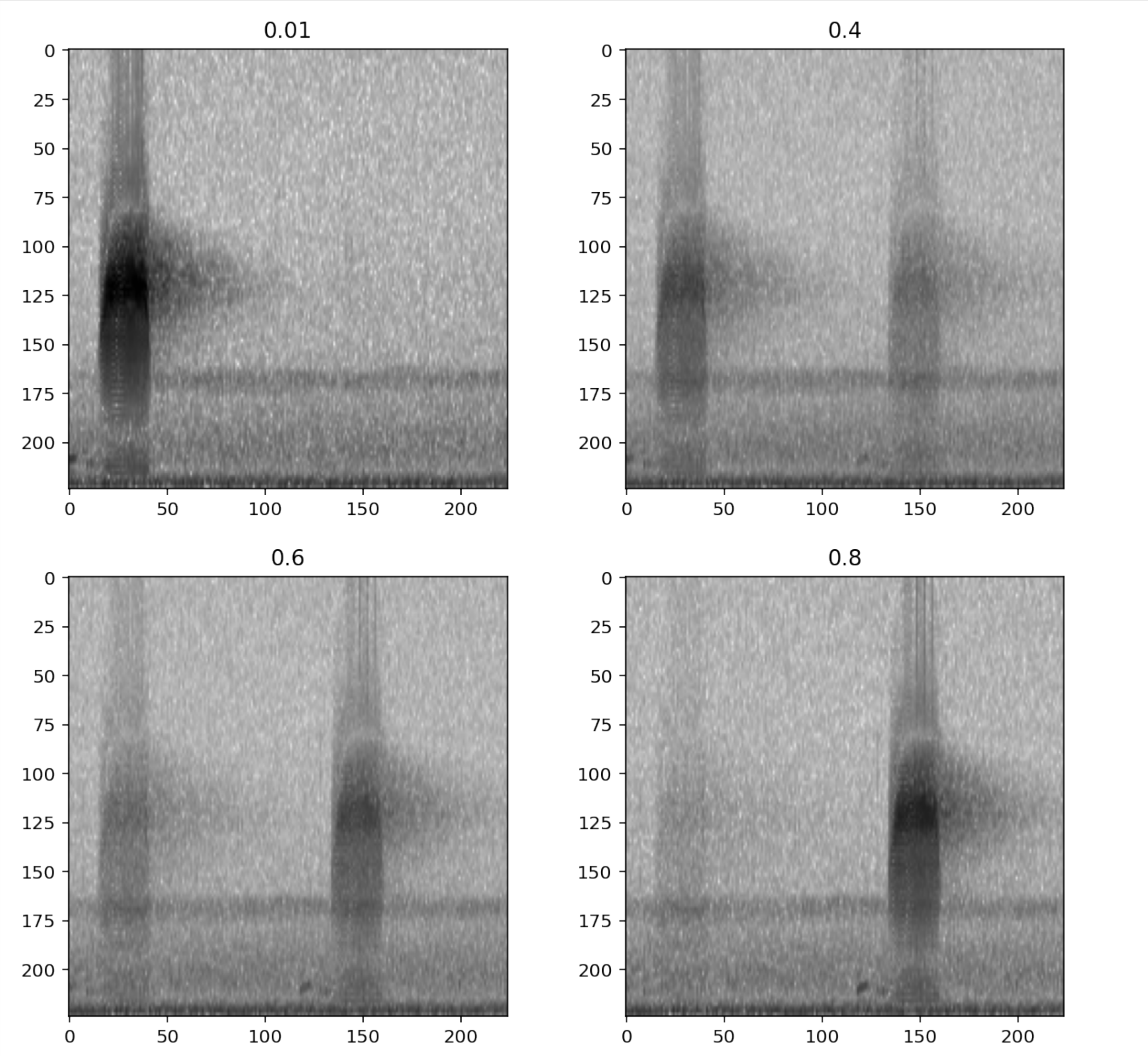
[44]:
```
#initialize a preprocessor and provide a dataframe with samples to use as overlays
preprocessor = SpectrogramPreprocessor(2.0, overlay_df=labels)

#remove augmentations other than overlay
for name in ['random_affine','time_mask','frequency_mask','add_noise']:
    preprocessor.remove_action(name)
```

Let's change `overlay_weight` to

To demonstrate this, let's show what happens if we overlay samples from the "negative" class, resulting in the final sample having a higher or lower signal-to-noise ratio. By default, the overlay Action chooses a random file from the overlay dataframe. Instead, choose a sample from the class called `"present"` using the `overlay_class` parameter.

[45]:
```
preprocessor.pipeline.overlay.set(overlay_class='present')
tensors = []
overlay_weights = [0.01, 0.4, 0.6, 0.8]
for w in overlay_weights:
    preprocessor.pipeline.overlay.set(overlay_weight=w)
    dataset = AudioFileDataset(labels,preprocessor)
    np.random.seed(0) #get the same overlay every time
    tensors.append(dataset[2]['X'])
_ = show_tensor_grid(tensors, 2, labels=overlay_weights)
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



As demonstrated above, you can choose a specific class to choose samples from. Here, instead, we choose samples from the "absent" class.

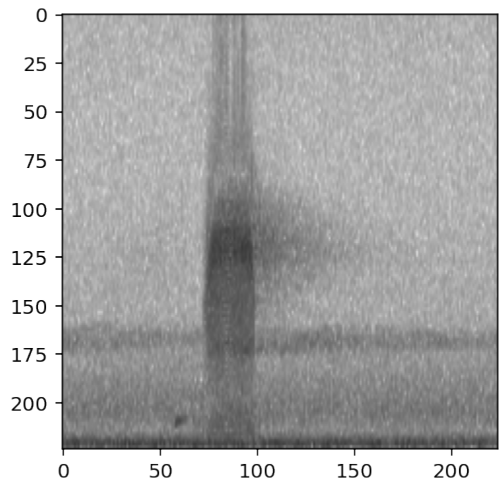
[46]:
```
dataset.preprocessor.pipeline.overlay.set(
    overlay_class='absent',
    overlay_weight=0.4
)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
```

(continues on next page)

(continued from previous page)

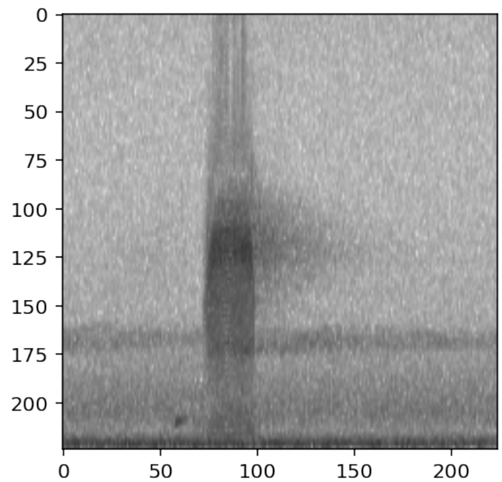
```
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



By default, or by specifying `overlay_class=None`, the overlay sample is chosen randomly from the overlay_df with no restrictions.

[47]:
```
dataset.preprocessor.pipeline.overlay.set(overlay_class=None)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



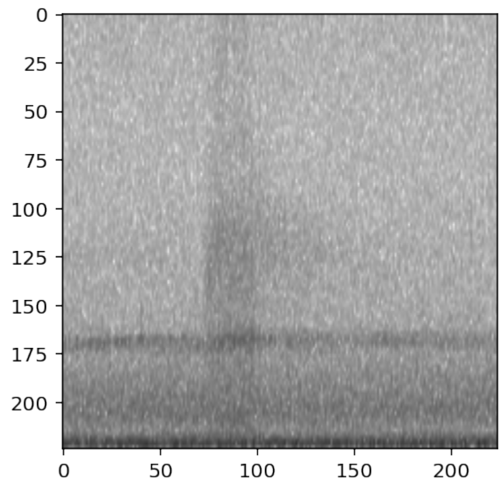
The `'different'` option for `overlay_class` chooses a sample to overlay that has non-overlapping labels with the original sample.

In the case of this example, this has the same effect as drawing samples from the `"negative"` class a demonstrated above. In multi-class examples, this would draw from any of the samples not labeled with the class(es) of the original sample.

We'll again use `overlay_weight=0.8` to exaggerate the importance of the overlayed sample (80%) compared to the original sample (20%).

[48]:
```
dataset.preprocessor.pipeline.overlay.set(update_labels=False,overlay_class='different
↪',overlay_weight=0.8)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



By default, the overlay Action does **not** change the labels of the sample it modifies.

For instance, if the overlayed sample has labels [1,0] and the original sample has labels [0,1], the default behavior will return a sample with labels [0,1] not [1,1].

If you wish to add the labels from overlayed samples to the original sample's labels, you can set `update_labels=True`.

[49]:
```
print('default: labels do not update')
dataset.preprocessor.pipeline.overlay.set(update_labels=False,overlay_class='different
↪')
print(f"\t resulting labels: {dataset[0]['y'].numpy()}")

print('Using update_labels=True')
dataset.preprocessor.pipeline.overlay.set(update_labels=True,overlay_class='different
↪')
print(f"\t resulting labels: {dataset[0]['y'].numpy()}")
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

```
default: labels do not update
         resulting labels: [1 0]
Using update_labels=True
         resulting labels: [1 1]
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

This example is a single-target problem: the two classes represent "woodcock absent" and "woodcock present." Because the labels are mutually exclusive, labels [1,1] do not make sense. So, for this single-target problem, we would **not** want to use `update_labels=True`, and it would probably make most sense to only overlay absent recordings, e.g., `overlay_class='absent'`.

10.9.1 Creating a new Preprocessor class

If you have a specific augmentation routine you want to perform, you may want to create your own Preprocessor class rather than modifying an existing one.

Your subclass might add a different set of Actions, define a different pipeline, or even override the `__getitem__` method of BasePreprocessor.

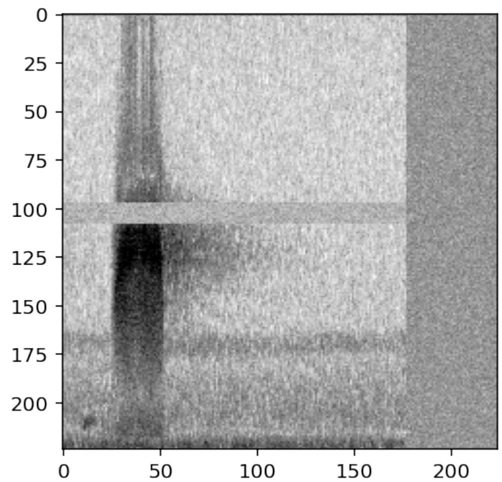
Here's an example of a customized preprocessor that subclasses `AudioToSpectrogramPreprocessor` and creates a pipeline that depends on the `magic_parameter` input.

[50]:
```
from opensoundscape.preprocess.actions import Action, tensor_add_noise
class MyPreprocessor(SpectrogramPreprocessor):
    """Child of AudioToSpectrogramPreprocessor with weird augmentation routine"""

    def __init__(
        self,
        magic_parameter,
        sample_duration,
        return_labels=True,
        out_shape=[224, 224,1],
    ):

        super(MyPreprocessor, self).__init__(
            sample_duration=sample_duration,
            out_shape=out_shape,
        )

        for i in range(magic_parameter):
            action = Action(tensor_add_noise, std=0.1*magic_parameter)
            self.insert_action(f'noise_{i}',action)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

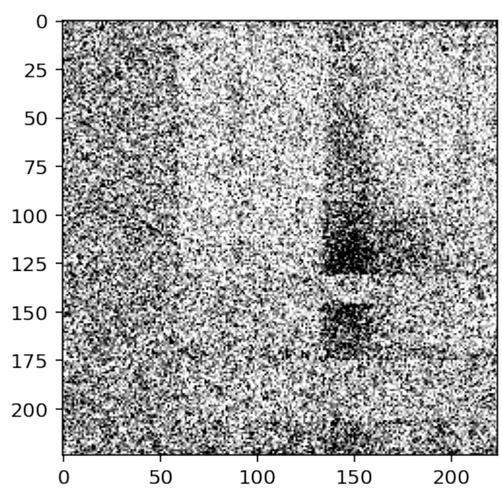
[51]:
```
dataset = AudioFileDataset(labels, MyPreprocessor(sample_duration=2.0, magic_
↪parameter=1))
show_tensor(dataset[0]['X'],invert=False)
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



```
[52]: dataset = AudioFileDataset(labels, MyPreprocessor(sample_duration=2.0, magic_
      ↪parameter=4))
      show_tensor(dataset[0]['X'],invert=False)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```



10.9.2 Defining new Actions

You can usually define a new action simply by passing a method to Action(). However, you can also write a subclass of Action for more advanced use cases - this is necessary if the action needs inputs other than the sample, such as labels.

10.10 using additional input in an Action

The following additional variables can be requested by an action, and will be passed from the pipeline when the action is run:

```
"_path": audio file path
"_labels": row of pd.DataFrame with 0/1 labels for each class (pd.Series)
"_start_time": start time of clip within longer audio file, if splitting long files␣
↪into clips during preprocessing
"_sample_duration": sample_duration of clip in seconds
"_pipeline": a copy of the preprocessor's pipeline itself
```

[53]:
```
from opensoundscape.preprocess.actions import Action

def my_action_fn(x, _labels,threshold=0.1):
    if _labels[0]==1:
        samples = np.array([0 if np.abs(s)<threshold else s for s in audio.samples])
        x = Audio(samples, audio.sample_rate)
    return x

class AudioGate(Action):
    """Replace audio samples below a threshold with 0, but only if label[0]==1

    Audio in, Audio out

    Args:
        threshold: sample values below this will become 0
    """

    def __init__(self, **kwargs):
        super(AudioGate, self).__init__(my_action_fn,extra_args=['_labels'],**kwargs)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

Test it out:

[54]:
```
from opensoundscape.audio import Audio

gate_action = AudioGate(threshold=0.2)

print('histogram of samples')
audio = Audio.from_file('./woodcock_labeled_data/01c5d0c90bd4652f308fd9c73feb1bf5.wav
↪')
_ = plt.hist(audio.samples,bins=100)
plt.semilogy()
plt.show()

print('histogram of samples after audio gate')
audio_gated = gate_action.go(audio,_labels={0:1,1:0})
_ = plt.hist(audio_gated.samples,bins=100)
plt.semilogy()
plt.show()
```

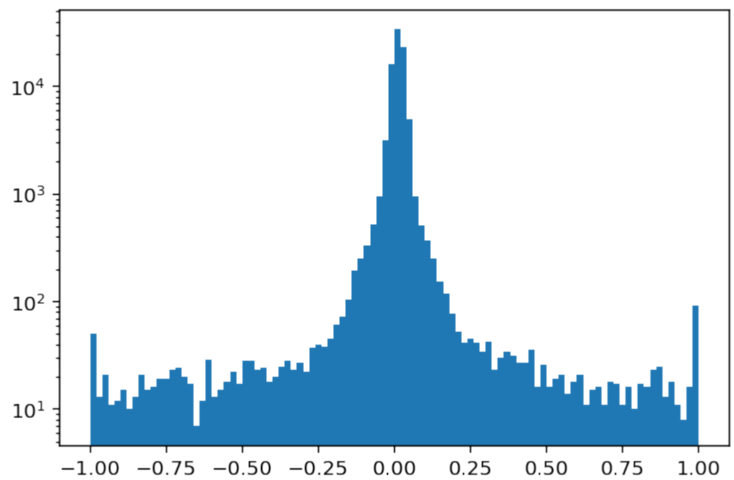
(continues on next page)

(continued from previous page)

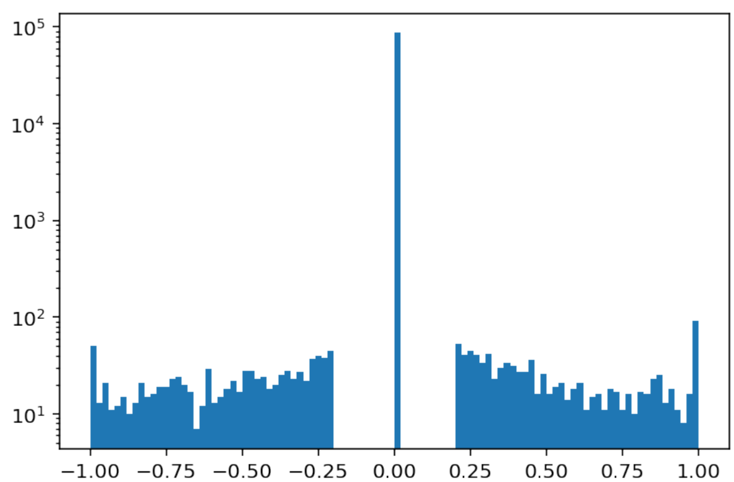
```
print('histogram of samples after audio gate, when labels[0]==0')
audio_gated = gate_action.go(audio,_labels={0:0,1:1})
_ = plt.hist(audio_gated.samples,bins=100)
plt.semilogy()
```

```
/home/jatink/repos/opensoundscape/opensoundscape/audio.py:1162: UserWarning: Failed␣
↪to load metadata: argument of type 'NoneType' is not iterable. Metadata will be None
  warnings.warn(f"Failed to load metadata: {exc}. Metadata will be None")
```

```
histogram of samples
```



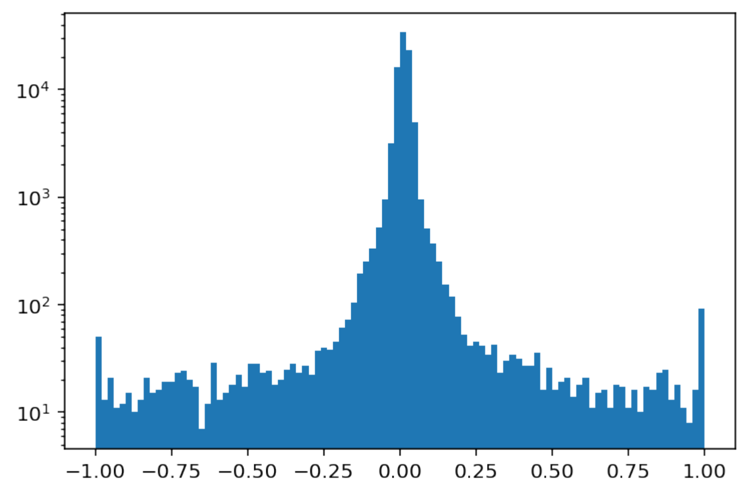

```
histogram of samples after audio gate
```




```
histogram of samples after audio gate, when labels[0]==0
```

[54]:

```
[]
```



10.10.1 Add custom Action to a preprocessor

For instance, if you want to use your custom Action while training a cnn, you can add it to the `cnn.preprocessor`'s pipeline.

In this example, we put the custom AudioGate action before the `to_spec` action.

[55]:
```
gate_action = AudioGate(threshold=0.2)
preprocessor.insert_action(
    action_index='custom_audio_gate', #give it a name
    action=gate_action,
    before_key='to_spec', #where to put it (can also use before_key=...)
)
```

```
/home/jatink/miniconda3/envs/opso-dev/lib/python3.8/site-packages/ipykernel/ipkernel.
↪py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

[56]:
```
preprocessor.pipeline
```

[56]:
```
load_audio            Action calling <bound method Audio.from_file o...
random_trim_audio     Augmentation Action calling <function trim_aud...
trim_audio            Action calling <function trim_audio at 0x7f0cf...
custom_audio_gate     Action calling <function my_action_fn at 0x7f0...
to_spec               Action calling <bound method Spectrogram.from_...
bandpass              Action calling <function Spectrogram.bandpass ...
to_img                Action calling <function Spectrogram.to_image ...
overlay               Augmentation Action calling <function overlay ...
rescale               Action calling <function scale_tensor at 0x7f0...
dtype: object
```

Clean up files created during this tutorial:

[57]:
```
import shutil
shutil.rmtree('./woodcock_labeled_data')
```

CHAPTER 11

Advanced CNN training

This notebook demonstrates how to use classes from `opensoundscape.torch.models.cnn` and architectures created using `opensoundscape.torch.architectures.cnn_architectures` to

- choose between single-target and multi-target model behavior
- modify learning rates, learning rate decay schedule, and regularization
- choose from various CNN architectures
- train a multi-target model with a special loss function
- use strategic sampling for imbalanced training data
- customize preprocessing: train on spectrograms with a bandpassed frequency range

Rather than demonstrating their effects on training (model training is slow!), most examples in this notebook either don't train the model or "train" it for 0 epochs for the purpose of demonstration.

For introductory demos (basic training, prediction, saving/loading models), see the "Beginner-friendly training and prediction with CNNs" tutorial (cnn.ipynb).

[1]:
```
from opensoundscape.preprocess import preprocessors
from opensoundscape.torch.models import cnn
from opensoundscape.torch.architectures import cnn_architectures

import torch
import pandas as pd
from pathlib import Path
import numpy as np
import random
import subprocess

from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for big visuals
%config InlineBackend.figure_format = 'retina'
```

11.1 Prepare audio data

11.1.1 Download labeled audio files

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 `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.

If you already have these files, you can skip or comment out this cell

[2]:
```
subprocess.run(['curl','https://pitt.box.com/shared/static/
↪79fi7d715dulcldsy6uogz02rsn5uesd.gz','-L', '-o','woodcock_labeled_data.tar.gz']) #␣
↪Download the data
subprocess.run(["tar","-xzf", "woodcock_labeled_data.tar.gz"]) # Unzip the downloaded␣
↪tar.gz file
subprocess.run(["rm", "woodcock_labeled_data.tar.gz"]) # Remove the file after its␣
↪contents are unzipped
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      4      0 --:--:--  0:00:01 --:--:--  7000
100 9499k  100 9499k    0     0  3513k      0  0:00:02  0:00:02 --:--:-- 13.1M
```

[2]:
```
CompletedProcess(args=['rm', 'woodcock_labeled_data.tar.gz'], returncode=0)
```

11.1.2 Load dataframe of files and one-hot labels

We need a dataframe with file paths in the index, so we manipulate the included one_hot_labels.csv slightly

See the "Basic training and prediction with CNNs" tutorial for more details.

[3]:
```
# load one-hot labels dataframe
labels = pd.read_csv('./woodcock_labeled_data/one_hot_labels.csv').set_index('file')[[
↪'present']]
# prepend the folder location to the file paths
labels.index = pd.Series(labels.index).apply(lambda f: './woodcock_labeled_data/'+f)
#create class list
classes = labels.columns
#inspect
labels.head()
```

[3]:
```
                                                     present
file
./woodcock_labeled_data/d4c40b6066b489518f8da83...        1
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...        0
./woodcock_labeled_data/79678c979ebb880d5ed6d56...        1
./woodcock_labeled_data/49890077267b569e142440f...        1
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...        1
```

11.1.3 Split into train and validation sets

Randomly split the data into training data and validation data.

[4]:
```
from sklearn.model_selection import train_test_split
train_df, valid_df = train_test_split(labels, test_size=0.2, random_state=0)
print(f"created train_df (len {len(train_df)}) and valid_df (len {len(valid_df)})")
```

```
created train_df (len 23) and valid_df (len 6)
```

11.2 Creating a model

We initialize a model object by specifying the architecture, a list of classes, and the duration of individual samples in seconds

[25]:
```
arch = cnn_architectures.resnet50(num_classes=len(classes))
model = cnn.CNN(arch,classes,sample_duration=2.0)
```

Alternatively, we can specify the name of an architecture as a string (see Cnn Architectures below for details or use `cnn_architectures.list_architectures()` for options)

[6]:
```
model = cnn.CNN('resnet18',classes,2.0)
```

11.2.1 Single-target versus multi-target

One important decision is whether your model is single-target (exactly one label per sample) or multi-target (any number of labels per sample, including 0). Single-target models have a softmax activation layer which forces the sum of all class scores to be 1.0. By default, models are created as multi-target, but you can set `single_target=True` either when creating the object or afterwards.

[7]:
```
#change the model to be single_target
model.single_target = True

#or specify single_target when you create the object
model = cnn.CNN(arch,classes,2.0)
```

11.3 Model training parameters

We can modify various parameters about model training, including:

- The learning rate
- The learning rate schedule
- Weight decay for regularization

Let's take a peek at the current parameters, stored in a dictionary.

[8]:
```
model.optimizer_params
```

[8]:
```
{'lr': 0.01, 'momentum': 0.9, 'weight_decay': 0.0005}
```

11.3.1 Learning rates

The learning rate determines how much the model's weights change every time it calculates the loss function.

Faster learning rates improve the speed of training and help the model leave local minima as it learns to classify, but if the learning rate is too fast, the model may not successfully fit the data or its fitting might be unstable.

Often after training a model for a while at a relatively high learning rate (think 0.01), we might want to "fine tune" the model by training for a few epochs with a lower learning rate. Let's set a low learning rate for fine tuning:

[9]:
```
model.optimizer_params['lr']=0.001
```

11.3.2 Separate learning rates for feature and classifier blocks

For ResNet architectures, we can modify the learning rates for the feature extration and classification blocks of the network separately. For example, we can specify a relatively fast learning rate for classifier and slower one for features, if we think the features from a pre-trained model are close to optimal but we have a different set of classes than the pre-trained model. We first use a helper function to separate the feature and classifier parameters, then specify parameters for each:

[10]:
```
from opensoundscape.torch.models.cnn import separate_resnet_feat_clf
```

[11]:
```
r18_model = cnn.CNN('resnet18',classes,2.0)
print(r18_model.optimizer_params)

separate_resnet_feat_clf(r18_model) #in place operation!

#now we can specify separate parameters for the 'feature' and 'classifier' portions␣
↪of the network
r18_model.optimizer_params['feature']['lr'] = 0.001
r18_model.optimizer_params['classifier']['lr'] = 0.01

r18_model.optimizer_params
```

```
{'lr': 0.01, 'momentum': 0.9, 'weight_decay': 0.0005}
```

[11]:
```
{'feature': {'lr': 0.001, 'momentum': 0.9, 'weight_decay': 0.0005},
 'classifier': {'lr': 0.01, 'momentum': 0.9, 'weight_decay': 0.0005}}
```

11.3.3 Learning rate schedule

It's often helpful to decrease the learning rate over the course of training. By reducing the amount that the model's weights are updated as time goes on, this causes the learning to gradually switch from coarsely searching across possible weights to fine-tuning the weights.

By default, the learning rates are multiplied by 0.7 (the learning rate "cooling factor") once every 10 epochs (the learning rate "update interval").

Let's modify that for a very fast training schedule, where we want to multiply the learning rates by 0.1 every epoch.

[12]:
```
model.lr_cooling_factor = 0.1
model.lr_update_interval = 1
```

11.3.4 Regularization weight decay

Pytorch optimizers perform L2 regularization, giving the optimizer an incentive for the model to have small weights rather than large weights. The goal of this regularization is to reduce overfitting to the training data by reducing the complexity of the model.

Depending on how much emphasis you want to place on the L2 regularization, you can change the weight decay parameter. By default, it is 0.0005. The higher the value for the "weight decay" parameter, the more the model training algorithm prioritizes smaller weights.

[13]:
```
model.optimizer_params['weight_decay']=0.001
```

11.4 Selecting CNN architectures

The `opensoundscape.torch.architectures.cnn_architectures <https://github.com/kitzeslab/opensoundscape/blob/master/opensoundscape/torch/architectures/cnn_architectures.py>`__ module provides functions to create several common CNN architectures. These architectures are built in to pytorch, but the OpenSoundscape module helps us out by reshaping the final layer to match the number of classes we have.

You could also create a custom architecture by subclassing an existing pytorch model or writing one from scratch (the minimum requirement is that it subclasses `torch.nn.Module` - it should at least have `.forward()` and `.backward()` methods.

In general, we can create any pytorch model architecture and pass it to the `architecture` argument when creating a model in opensoundscape. We can choose whether to use pre-trained (ImageNet) weights or start from scratch (`weights=None` for random weights). For instance, lets create an alexnet architecture with random weights:

[14]:
```
my_arch = cnn_architectures.alexnet(num_classes=len(classes),weights=None)
```

For convenience, we can also initialize a model object by providing the name of an architecture as a string, rather than the architecture object. For a list of valid architecture names, use `cnn_architectures.list_architectures()`. Note that these will use default architecture parameters, including using pre-trained ImageNet weights. If you don't want to use pre-trained weights, follow the method above of creating the architecture and passing it to the initialization of CNN.

[15]:
```
print(cnn_architectures.list_architectures())
```

```
['resnet18', 'resnet34', 'resnet50', 'resnet101', 'resnet152', 'alexnet', 'vgg11_bn',
↪'squeezenet1_0', 'densenet121', 'inception_v3', 'efficientnet_b0', 'efficientnet_b4
↪', 'efficientnet_widese_b0', 'efficientnet_widese_b4']
```

[16]:
```
model = cnn.CNN(architecture='resnet18',classes=classes, sample_duration=2.0)
```

11.4.1 Pretrained weights

In OpenSoundscape, by default, model architectures are initialized with weights pretrained on the ImageNet image database. It takes some time for pytorch to download these weights from an online repository the first time an instance of a particular architecture is created with pretrained weights - pytorch will do this automatically and only once.

Using pretrained weights often speeds up training significantly, as the representation learned from ImageNet is a good start at beginning to interpret spectrograms, even though they are not true "pictures."

If you prefer not to use pre-trained weights, or if you don't have an internet connection, you can specify `weights` argument to `None`, when creating an architecture:

[17]:
```
arch = cnn_architectures.alexnet(num_classes=10,weights=None)
```

11.4.2 Freezing the feature extractor

Convolutional Neural Networks can be thought of as having two parts: a **feature extractor** which learns how to represent/"see" the input data, and a **classifier** which takes those representations and transforms them into predictions about the class identity of each sample.

You can freeze the feature extractor if you only want to train the final classification layer of the network but not modify any other weights. This could be useful for applying pre-trained classifiers to new data, i.e. "transfer learning". To do so, set the `freeze_feature_extractor` argument to `True` when you create an architecture.

[18]:
```
# See "InceptionV3 architecture" section below for more information
arch = cnn_architectures.resnet50(num_classes=10, freeze_feature_extractor=True,␣
↪weights=None)
```

11.4.3 InceptionV3 class

The Inception architecture requires slightly different training and preprocessing from the ResNet architectures and the other architectures implemented in OpenSoundscape (see below), because:

1) the input image shape must be 299x299, and
2) Inception's forward pass gives output + auxiliary output instead of a single output

The InceptionV3 class in `cnn` handles the necessary modifications in training and prediction for you, so use that instead of CNN:

[19]:
```
from opensoundscape.torch.models.cnn import InceptionV3

#generate an Inception model
model = InceptionV3(classes=classes,weights=None,sample_duration=2)

#train and validate for 1 epoch
#note that Inception will complain if batch_size=1
model.train(train_df,valid_df,epochs=1,batch_size=4)

#predict
scores = model.predict(valid_df)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/torchvision/models/
↪inception.py:43: FutureWarning: The default weight initialization of inception_v3␣
↪will be changed in future releases of torchvision. If you wish to keep the old␣
↪behavior (which leads to long initialization times due to scipy/scipy#11299),␣
↪please set init_weights=True.
  warnings.warn(
```

```

Training Epoch 0
Epoch: 0 [batch 0/6, 0.00%]
        DistLoss: 1.131
Metrics:
Metrics:
        MAP: 0.797

Validation.
```

(continues on next page)

(continued from previous page)

```
Metrics:
        MAP: 1.000

Best Model Appears at Epoch 0 with Validation score 1.000.
```

11.4.4 Changing the architecture of an existing model (not recommended)

The architecture is stored in the model object's `.newtork` attribute. We can access parameters of the network or even replace it entirely.

Note that replacing the architecture will completely remove anything the model has "learned" since the learned weights are a part of the architecture.

[20]:
```
#initialize the AlexNet architecture
new_arch = cnn_architectures.densenet121(num_classes=2, weights=None)

# replace the alexnet architecture with the densenet architecture
model.network = new_arch
```

11.5 Multi-target training with ResampleLoss

Training multi-target models (a.k.a. multi-label: there can be any number of positive labels on each sample) is challenging and can benefit from using a modified loss function. OpenSoundscape provides a loss function designed for training multi-target models. We recommend using this loss function when training multi-target models. You can add it to a class with an in-place helper function:

[21]:
```
from opensoundscape.torch.models.cnn import use_resample_loss
```

[22]:
```
model = cnn.CNN('resnet18',classes,2.0)
use_resample_loss(model)
print(model.loss_cls)

#use as normal...
#model.train(...)
#model.predict(...)
```

```
<class 'opensoundscape.torch.loss.ResampleLoss'>
```

11.6 Training and predicting with custom preprocessors

The preprocessing tutorial gives in-depth descriptions of how to customize your preprocessing pipeline.

Here, we'll just give a quick example of tweaking the preprocessing pipeline: providing the CNN with a bandpassed spectrogram object instead of the full frequency range.

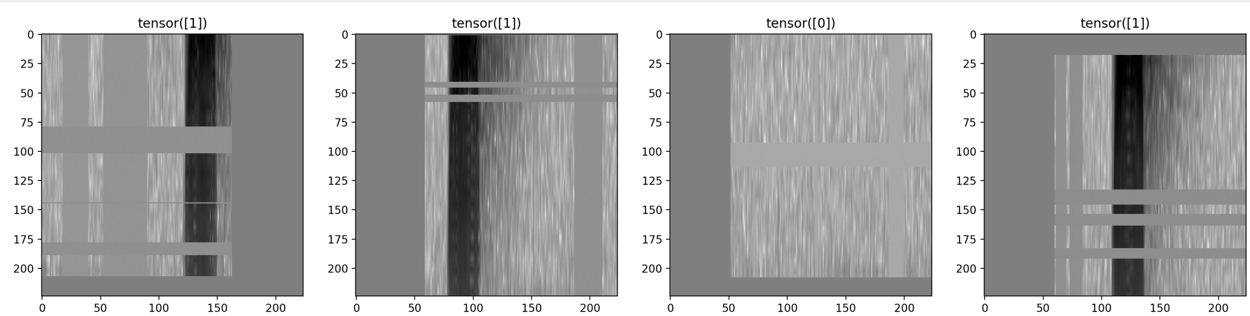
It's good practice to create the validation from the training dataset (after any modifications are made), so that they perform the same preprocessing. You may or may not want to use augmentation on the validation dataset.

11.6.1 Example: Training on bandpassed spectrograms

[23]:
```
model = cnn.CNN('resnet18', classes, 2.0)

# change the min and max frequencies for the spectrogram bandpass action
model.preprocessor.pipeline.bandpass.set(min_f=3000, max_f=5000)

# inspect a few preprocessed samples (see basic CNN training and prediction tutorial␣
↪for details)
from opensoundscape.preprocess.utils import show_tensor_grid
from opensoundscape.torch.datasets import AudioFileDataset
sample_of_4 = train_df.sample(n=4)
inspection_dataset = AudioFileDataset(sample_of_4, model.preprocessor)
samples = [sample['X'] for sample in inspection_dataset]
labels = [sample['y'] for sample in inspection_dataset]
_ = show_tensor_grid(samples,4,labels=labels)

# now we can train and validate on the bandpassed spectrograms
model.train(train_df, valid_df, epochs=0)
```

```
/Users/SML161/miniconda3/envs/opso_dev/lib/python3.9/site-packages/matplotlib_inline/
↪config.py:68: DeprecationWarning: InlineBackend._figure_format_changed is␣
↪deprecated in traitlets 4.1: use @observe and @unobserve instead.
  def _figure_format_changed(self, name, old, new):
/Users/SML161/opensoundscape/opensoundscape/preprocess/utils.py:87:␣
↪MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since␣
↪3.6 and will be removed two minor releases later; explicitly call ax.remove() as␣
↪needed.
  ax = plt.subplot(len(tensors) // columns + 1, columns, i + 1)
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```

```
Best Model Appears at Epoch 0 with Validation score 0.000.
```



If we predict using this model for prediction, it will use the same preprocessor settings, bandpassing the prediction samples in the same way as the training samples.

11.6.2 clean up

remove files

[24]:
```
import shutil
shutil.rmtree('./woodcock_labeled_data')

for p in Path('.').glob('*.model'):
    p.unlink()
```

CHAPTER 12

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.

This work is described in:

- 2021 paper, "Automated detection of frog calls and choruses by pulse repetition rate"
- 2020 poster, "Automatic Detection of Pulsed Vocalizations"

RIBBIT is also available as an R package.

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

For help instaling OpenSoundscape, see the documentation

12.1 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
import subprocess

#local imports from opensoundscape
```
(continues on next page)

(continued from previous page)

```
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
from opensoundscape.ribbit import ribbit

# create big visuals
plt.rcParams['figure.figsize']=[15,8]
pd.set_option('display.precision', 2)
```

12.2 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]:
```
#download files from box.com to the current directory
subprocess.run(['curl', 'https://pitt.box.com/shared/static/
↪211w13xunwfw0ucg9ft9fmjnaxl4uutt.gz','-L', '-o', 'great_plains_toad_dataset.tar.gz
↪']) # Download the data
subprocess.run(["tar","-xzf", "great_plains_toad_dataset.tar.gz"]) # Unzip the␣
↪downloaded tar.gz file
subprocess.run(["rm", "great_plains_toad_dataset.tar.gz"]) # Remove the file after␣
↪its contents are unzipped
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     8    0     8    0     0      5      0 --:--:--  0:00:01 --:--:--  8000
100 11.6M  100 11.6M    0     0  3724k      0  0:00:03  0:00:03 --:--:-- 9371k
```

[2]:
```
CompletedProcess(args=['rm', 'great_plains_toad_dataset.tar.gz'], returncode=0)
```

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

12.2.1 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)
```

(continues on next page)

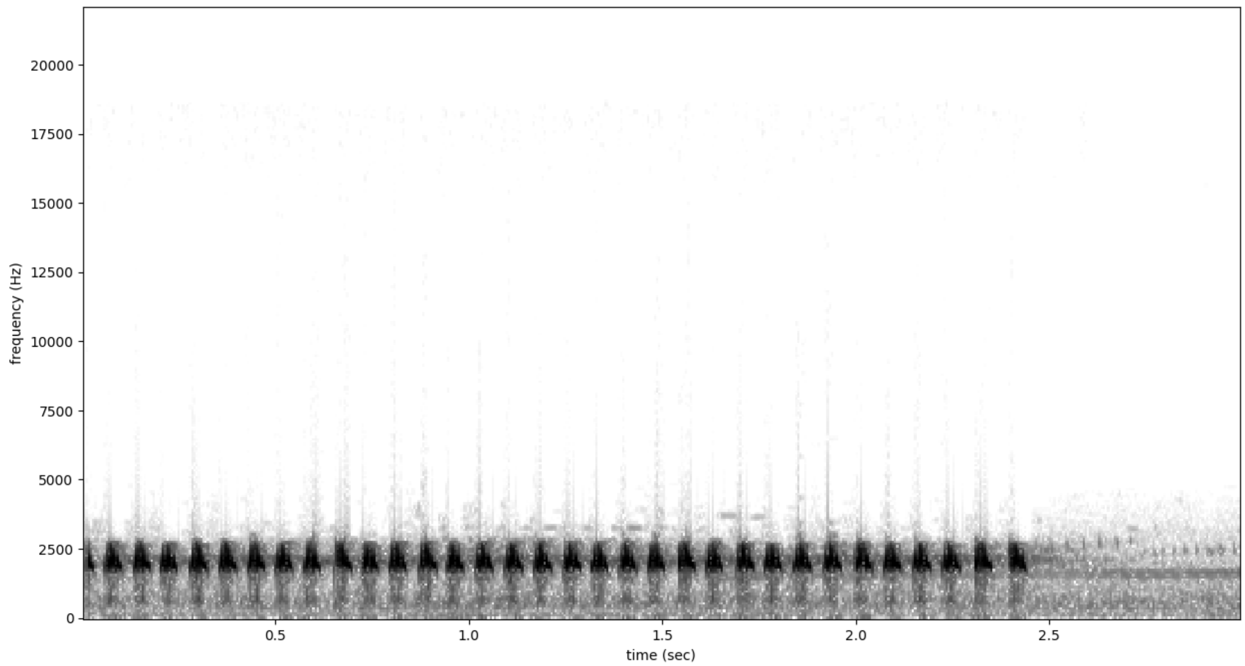
(continued from previous page)

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

12.2.2 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()
```



12.3 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.

Clip Duration: This parameter tells the algorithm how many seconds of audio to analyze at one time. Generally, you should choose a `clip_duration` that is ~2x longer than 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 `clip_duration` are 0.3 to 10 seconds. Here, because the The Great Plains Toad has a vocalization that continues on for many seconds (or minutes!), we chose a 2-second window which will include plenty of pulses.

- we can also set `clip_overlap` if we want overlapping clips. For instance, a `clip_duration` of 2 with `clip_overlap` of 1 results in 50% overlap of each consecutive clip. This can help avoid sounds being split up across two clips, and therefore not being detected.
- `final_clip` determines what should be done when there is less than `clip_duration` audio remaining at the end of an audio file. We'll just use `final_clip=None` to discard any remaining audio that doesn't make a complete clip.

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]:
```python
# minimum and maximum rate of pulsing (pulses per second) to search for
pulse_rate_range = [8,15]

# look for a vocalization in the range of 1000-2000 Hz
signal_band = [1800,2400]

# subtract the amplitude signal from these frequency ranges
noise_bands = [ [0,1000], [3000,3200] ]

#divides the signal into segments this many seconds long, analyzes each independently
clip_duration = 2 #seconds
clip_overlap = 0 #seconds

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

12.4 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.

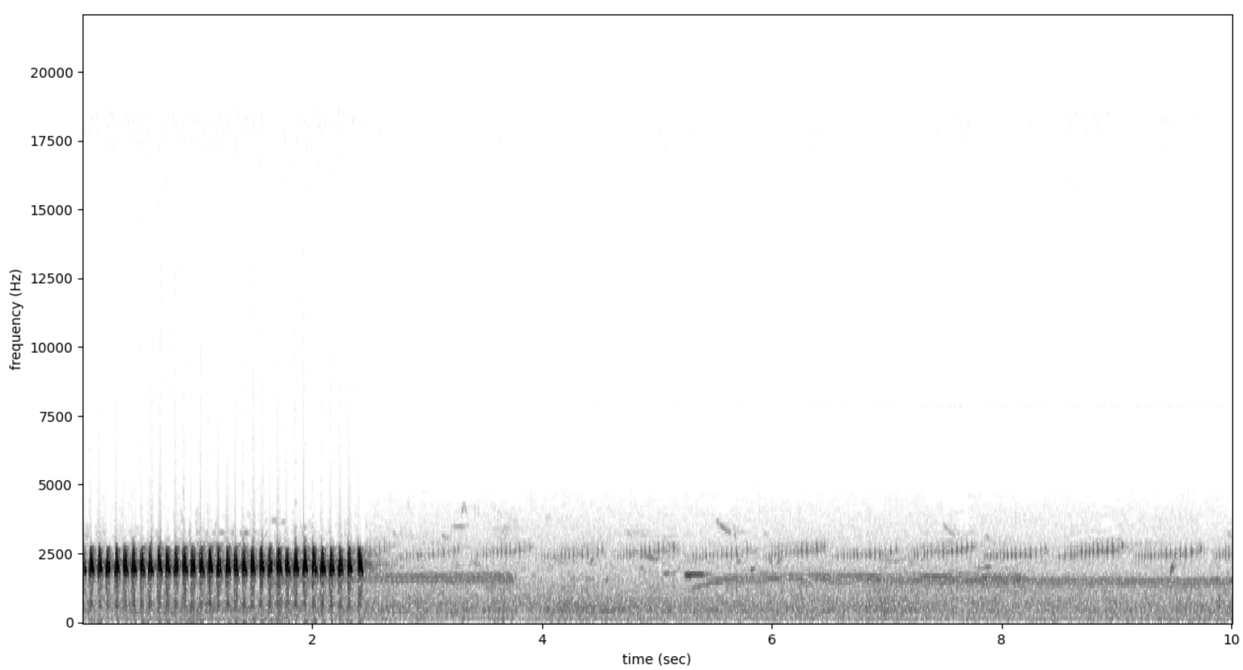
[6]:
```python
#get the audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

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

(continues on next page)

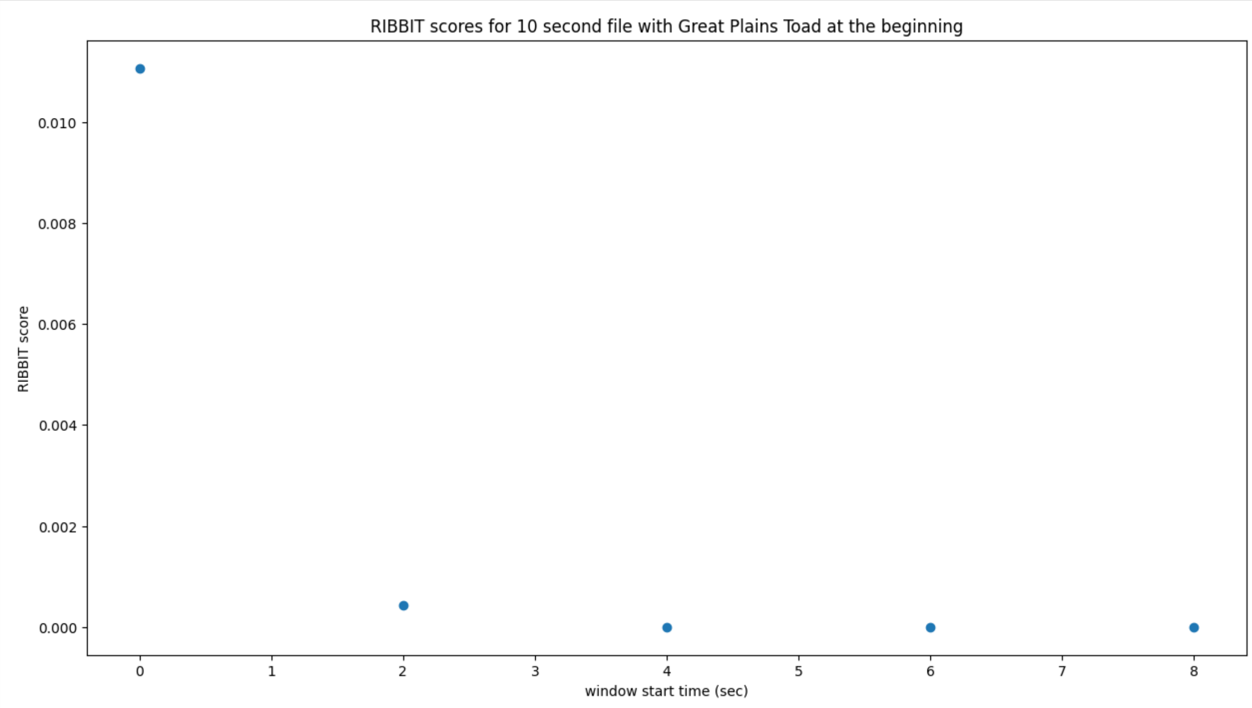
(continued from previous page)

```
#run RIBBIT
score_df =  ribbit(
                    spec,
                    pulse_rate_range=pulse_rate_range,
                    signal_band=signal_band,
                    clip_duration=clip_duration,
                    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(score_df['start_time'],score_df['score'])
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
```




```
[6]: 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

12.5 Analyzing a set of files

[7]:
```
# 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
    score_df =  ribbit(
                        spec,
                        pulse_rate_range=[8,20],
                        signal_band=[1900,2400],
                        clip_duration=clip_duration,
                        noise_bands=[[0,1500],[2500,3500]],
                        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
```

(continues on next page)

(continued from previous page)

```
    df.at[path,'score'] = max(score_df['score']) * 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:
```

[7]:

	score	label
./great_plains_toad_dataset/gpt0.wav	107.65	1
./great_plains_toad_dataset/gpt3.wav	29.31	1
./great_plains_toad_dataset/gpt2.wav	16.69	1
./great_plains_toad_dataset/gpt1.wav	10.13	1
./great_plains_toad_dataset/negative9.wav	3.04	0
./great_plains_toad_dataset/negative8.wav	0.89	0
./great_plains_toad_dataset/negative4.wav	0.76	0
./great_plains_toad_dataset/negative2.wav	0.65	0
./great_plains_toad_dataset/negative1.wav	0.3	0
./great_plains_toad_dataset/negative3.wav	0.3	0
./great_plains_toad_dataset/gpt4.wav	0.12	1
./great_plains_toad_dataset/negative6.wav	0.06	0
./great_plains_toad_dataset/pops2.wav	0.0	0
./great_plains_toad_dataset/pops1.wav	0.0	0
./great_plains_toad_dataset/negative5.wav	0.0	0
./great_plains_toad_dataset/silent.wav	0.0	0
./great_plains_toad_dataset/negative7.wav	0.0	0
./great_plains_toad_dataset/water.wav	0.0	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 10 except `gpt4.wav` (which contains only a very quiet and distant vocalization). All files that do not contain the Great Plains Toad score less than 3.5. So, RIBBIT is doing a good job separating Great Plains Toads vocalizations from other sounds!

Notably, noisy files like `pops1.wav` 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 pulse 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: eliminating the `noise_bands` will result in high scores for the "pops" files)

12.6 Run RIBBIT on multiple species simultaneously

If you want to search for multiple species, its best to combine the analysis into one function - that way you only have to load each audio file (and make it's spectrogram) one time, instead of once for each species. (If you have thousands of audio files, this might be a big time saver.)

This code gives a quick exmaple of how you could use a pre-made dataframe (could load it in from a spreadsheet, for instance) of parameters for a set of species to run RIBBIT on all of them.

Note that this example assumes you are using the same spectrogram settings for each species - this might not be the case in practice, if some species require high time-resolution spectrograms and others require high frequency-resolution spectrograms.

[8]:
```
#we'll create a dataframe here, but you could also load it from a spreadsheet
species_df = pd.DataFrame(columns=['pulse_rate_range','signal_band','clip_duration',
↪'noise_bands'])
```

(continues on next page)

(continued from previous page)

```
species_df.loc['great_plains_toad']={
    'pulse_rate_range':[8,20],
    'signal_band':[1900,2400],
    'clip_duration':2.0,
    'noise_bands':[[0,1500],[2500,3500]]
}

species_df.loc['bird_series']={
    'pulse_rate_range':[8,11],
    'signal_band':[5000,6500],
    'clip_duration':2.0,
    'noise_bands':[[0,4000]]
}


species_df
```

[8]:

```
                   pulse_rate_range   signal_band  clip_duration  \
great_plains_toad           [8, 20]  [1900, 2400]            2.0
bird_series                 [8, 11]  [5000, 6500]            2.0

                                 noise_bands
great_plains_toad  [[0, 1500], [2500, 3500]]
bird_series                      [[0, 4000]]
```

now let's analyze each audio file for each species.

We'll save the results in a table that has a column for each species.

[9]:

```
# set up a dataframe for storing files' scores and labels
df = pd.DataFrame(index = glob('./great_plains_toad_dataset/*'),columns=species_df.
↪index.values)

# calculate RIBBIT scores
for path in df.index:

    for species, species_params in species_df.iterrows():
    #use RIBBIT for each species in species_df
        #make the spectrogram
        spec = Spectrogram.from_audio(audio.from_file(path))

        #run RIBBIT
        score_df =  ribbit(
                            spec,
                            pulse_rate_range=species_params['pulse_rate_range'],
                            signal_band=species_params['signal_band'],
                            clip_duration=species_params['clip_duration'],
                            noise_bands=species_params['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,species] = max(score_df['score']) * 10000

print("Files with scores for each species, sorted by 'bird_series' score:")
df.sort_values(by='bird_series',ascending=False)
```

```
Files with scores for each species, sorted by 'bird_series' score:
```

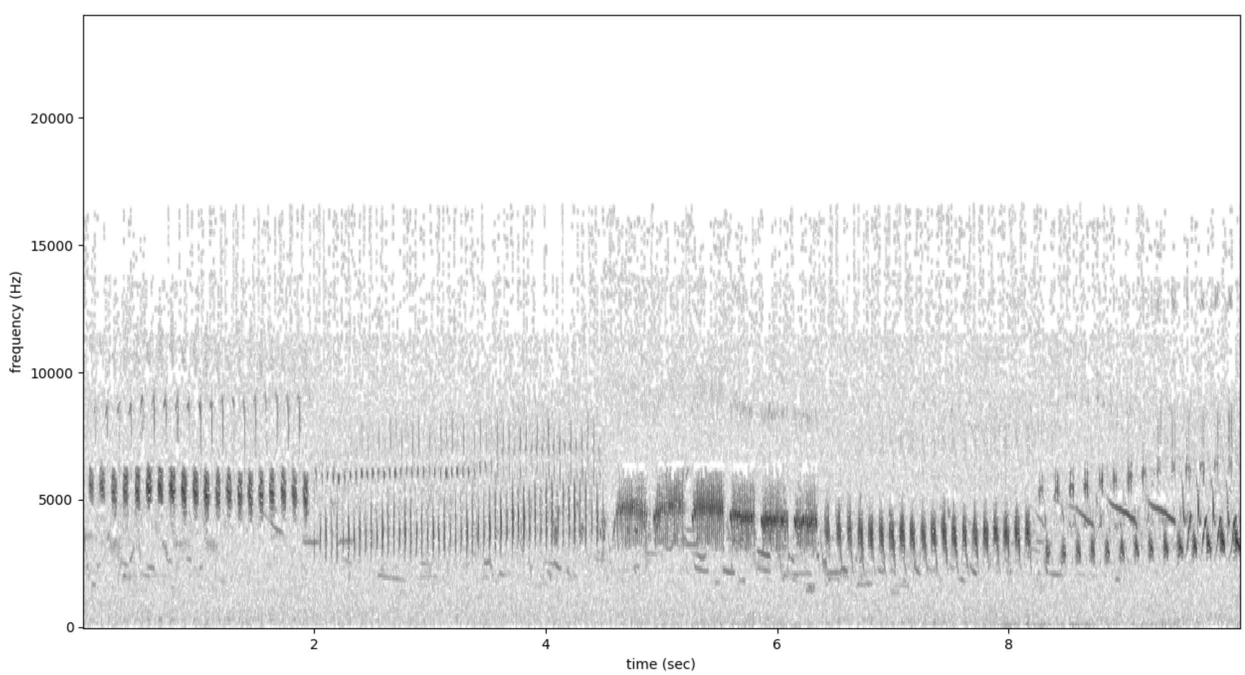
[9]:

```
                                        great_plains_toad bird_series
./great_plains_toad_dataset/negative5.wav             0.0        93.81
./great_plains_toad_dataset/negative1.wav             0.3        72.63
./great_plains_toad_dataset/negative3.wav             0.3         5.05
./great_plains_toad_dataset/negative7.wav             0.0         2.87
./great_plains_toad_dataset/negative9.wav            3.04         0.09
./great_plains_toad_dataset/negative2.wav            0.65         0.02
./great_plains_toad_dataset/negative6.wav            0.06         0.01
./great_plains_toad_dataset/pops2.wav                 0.0         0.01
./great_plains_toad_dataset/negative8.wav            0.89          0.0
./great_plains_toad_dataset/negative4.wav            0.76          0.0
./great_plains_toad_dataset/water.wav                 0.0          0.0
./great_plains_toad_dataset/pops1.wav                 0.0          0.0
./great_plains_toad_dataset/silent.wav                0.0          0.0
./great_plains_toad_dataset/gpt4.wav                 0.12          0.0
./great_plains_toad_dataset/gpt0.wav               107.65          0.0
./great_plains_toad_dataset/gpt1.wav                10.13          0.0
./great_plains_toad_dataset/gpt3.wav                29.31          0.0
./great_plains_toad_dataset/gpt2.wav                16.69          0.0
```

looking at the highest scoring file for 'bird_series', it has the trilled bird sound at 5-6.5 kHz

[10]:

```
Spectrogram.from_audio(audio.from_file('./great_plains_toad_dataset/negative5.wav')).
↪plot()
```



12.6.1 Warning

when loading a dataframe from a file, lists of numbers like [8,20] might be read in as strings ("[8,20]") rather than a list of numbers. Here's a handy little piece of code that will load the values in the desired format

[11]:
```
#let's say we have the species df saved as a csv file
species_df.index.name='species'
species_df.to_csv('species_df.csv')

#define the conversion parameters for each column
import ast
generic = lambda x: ast.literal_eval(x)
conv = {
    'pulse_rate_range':generic,
    'signal_band':generic,
    'noise_bands':generic
}
#tell pandas to use them when loading the csv
species_df=pd.read_csv('./species_df.csv',converters=conv).set_index('species')

#now the species_df has numeric values instead of strings
species_df
```

[11]:
```
                   pulse_rate_range   signal_band  clip_duration  \
species
great_plains_toad           [8, 20]  [1900, 2400]            2.0
bird_series                 [8, 11]  [5000, 6500]            2.0

                                  noise_bands
species
great_plains_toad  [[0, 1500], [2500, 3500]]
bird_series                       [[0, 4000]]
```

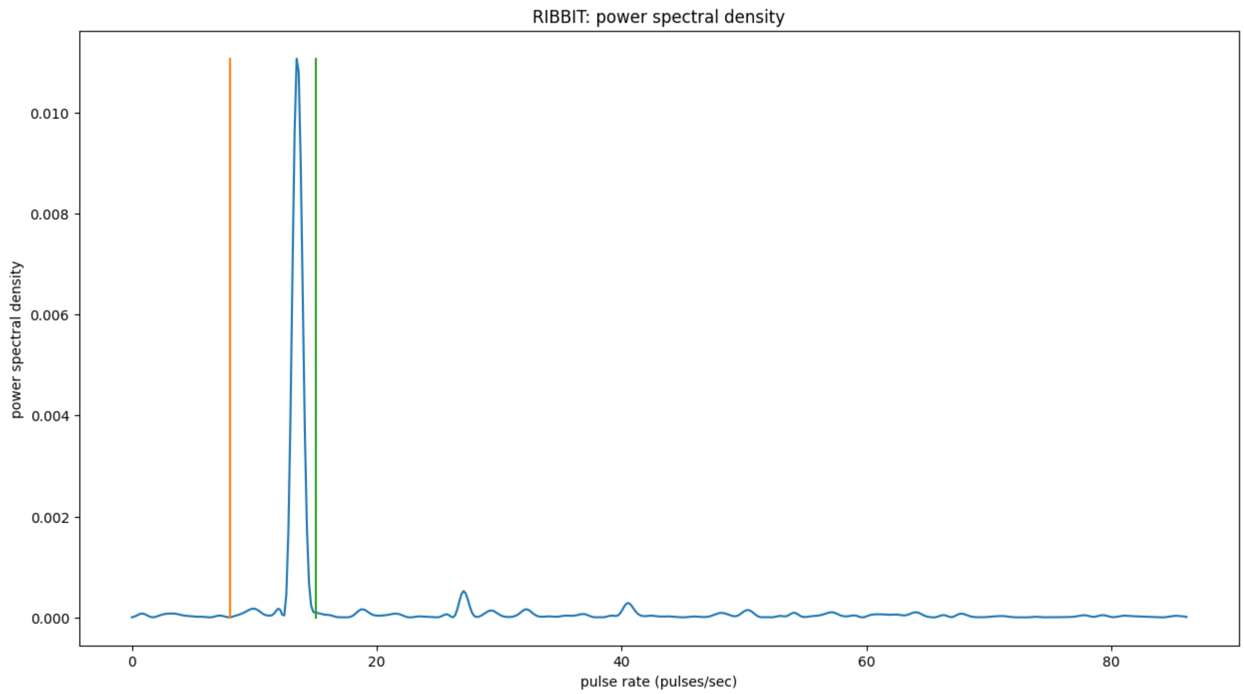
12.7 Detail view of RIBBIT method

Now, let's 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.

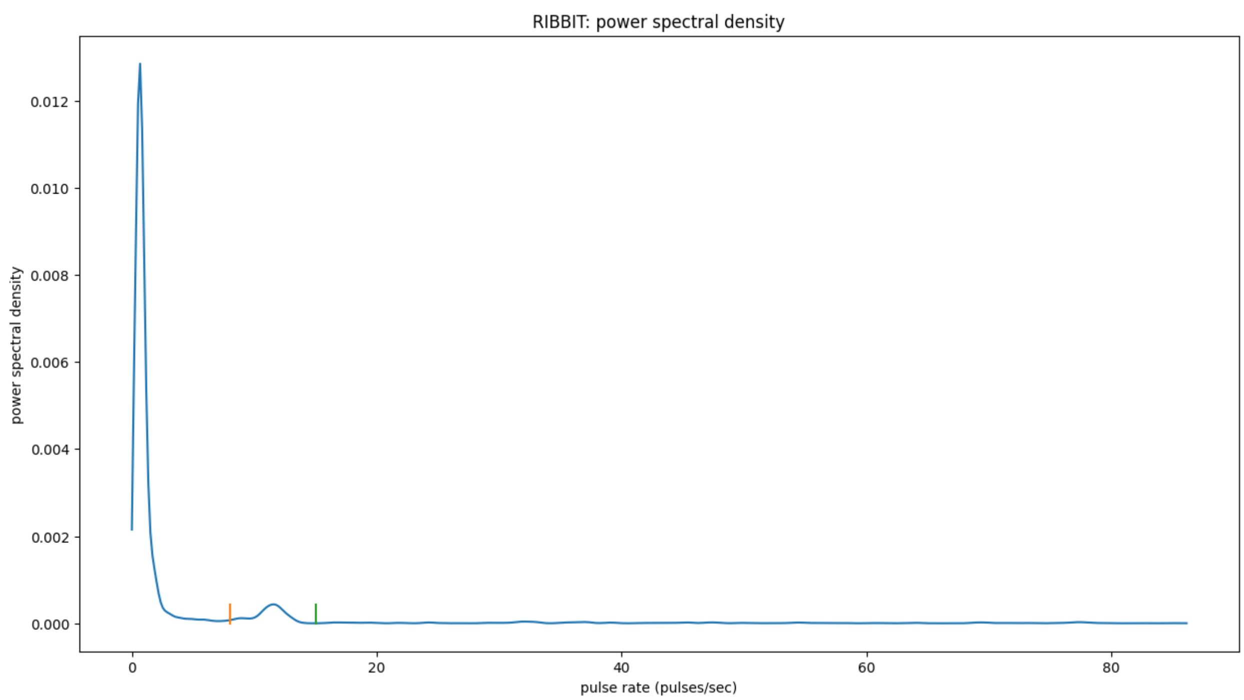
In the file `gpt0.wav`, the Great Plains Toad vocalizes for a couple seconds at the beginning, then stops. We expect to see a peak in the power spectral density at 15 pulses/sec in the first 2 second window, and maybe a bit in the second, but not later in the audio.

[12]:
```
#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)

clip_df =  ribbit(
                        spectrogram,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        clip_duration=clip_duration,
                        noise_bands=noise_bands,
                        plot=show_plots)
```

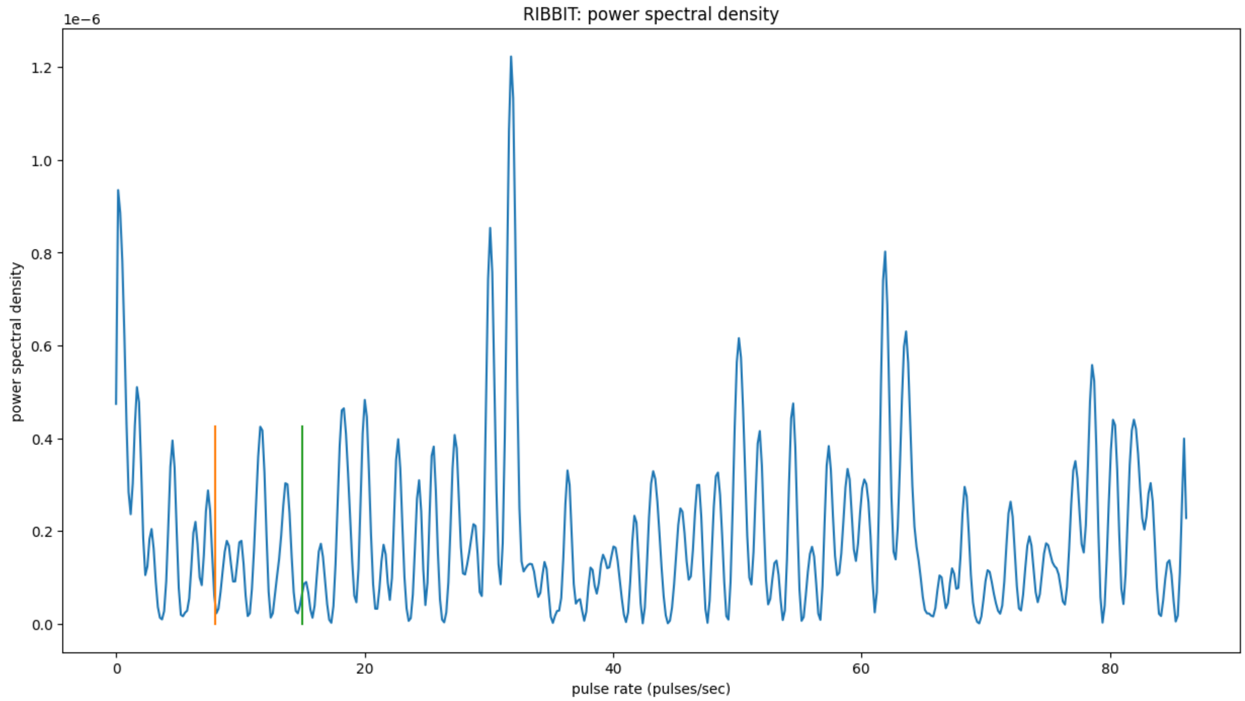
```
window: 0.0 to 2.0 sec
```



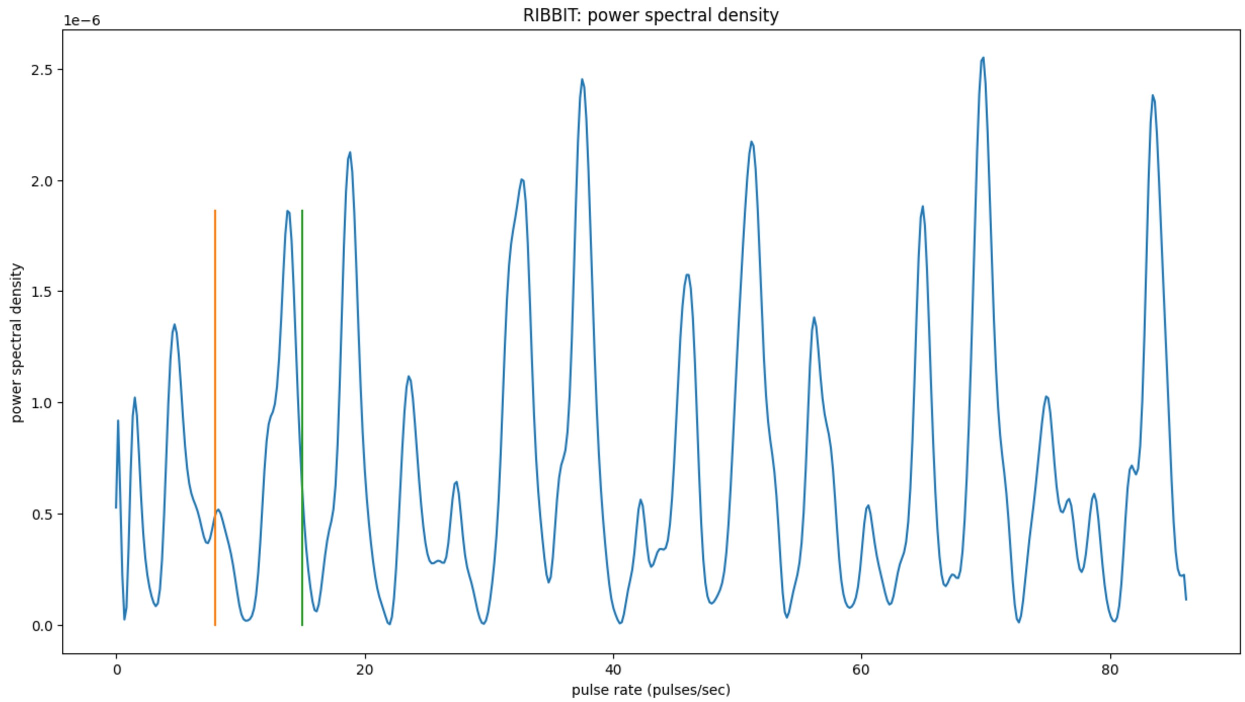

```
window: 2.0 to 4.0 sec
```




```
window: 4.0 to 6.0 sec
```




```
window: 6.0 to 8.0 sec
```



12.8 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.

[13]:
```
from pathlib import Path
import shutil
shutil.rmtree('./great_plains_toad_dataset/')
Path('./species_df.csv').unlink()
```

CHAPTER 13

Annotations

CHAPTER 14

Audio

CHAPTER 15

AudioMoth

CHAPTER 16

Audio Tools

CHAPTER 17

Spectrogram

CHAPTER 18

CNN

CHAPTER 19

torch.models.utils

CHAPTER 20

CNN Architectures

CHAPTER 21

torch.architectures.utils

CHAPTER 22

WandB (Weights and Biases)

CHAPTER 23

Data Selection

CHAPTER 24

Datasets

CHAPTER 25

GradCam

CHAPTER 26

Loss

CHAPTER 27

Safe Dataset

CHAPTER 28

Sampling

CHAPTER 29

Metrics

CHAPTER 30

Image Augmentation

CHAPTER 31

Actions

CHAPTER 32

Preprocessors

CHAPTER 33

preprocessors.utils

CHAPTER 34

Tensor Augment

CHAPTER 35

RIBBIT

CHAPTER 36

Signal Processing

CHAPTER 37

Taxa

CHAPTER 38

Localization

CHAPTER 39

helpers

CHAPTER 40

Index

CHAPTER 41

Modules

- modindex