opensoundscape

Release 0.7.0

May 26, 2022

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 (or 3.8) using the pip command `pip install opensoundscape==0.7.0`. 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 (or 3.8) conda environment for opensoundscape: `conda create --name opensoundscape pip python=3.7`
- Activate the environment: `conda activate opensoundscape`
- Install opensoundscape using pip: `pip install opensoundscape==0.7.0`
- Deactivate the environment when you're done using it: `conda deactivate`

1.2 Installation via `venv`

Download Python 3.7 (or 3.8) from this website.

Run the following commands in your bash terminal:

- Check that you have installed Python 3.7 (or 3.8)._: `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==0.7.0`
- 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 (or 3.8) conda environment for opensoundscape: `conda create --name opensoundscape pip python=3.7`
- Activate the environment: `conda activate opensoundscape`
- Install opensoundscape using pip: `pip install opensoundscape==0.7.0`

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==0.7.
↪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==3.8` 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      6      0 --:--:--  0:00:01 --:--:--    12
100 3750k  100 3750k    0     0  1929k      0  0:00:01  0:00:01 --:--:-- 21.7M
```

[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()` method 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]:

file	start_time	end_time
./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

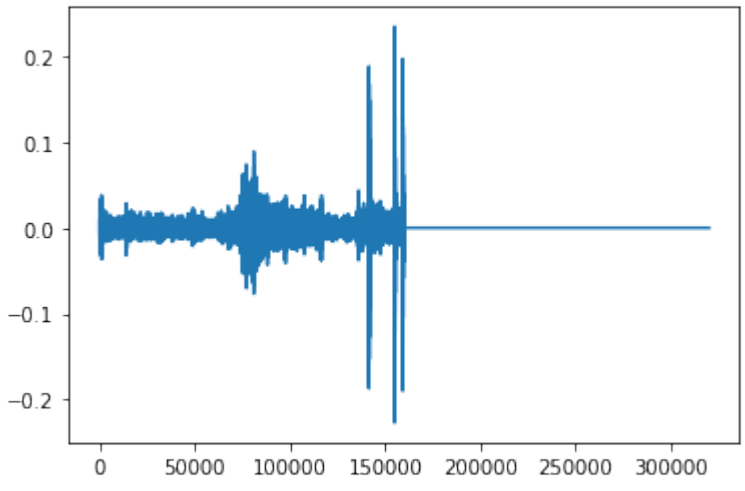
The `.extend()` method extends an audio file to a desired length by adding silence to the end.

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:
```

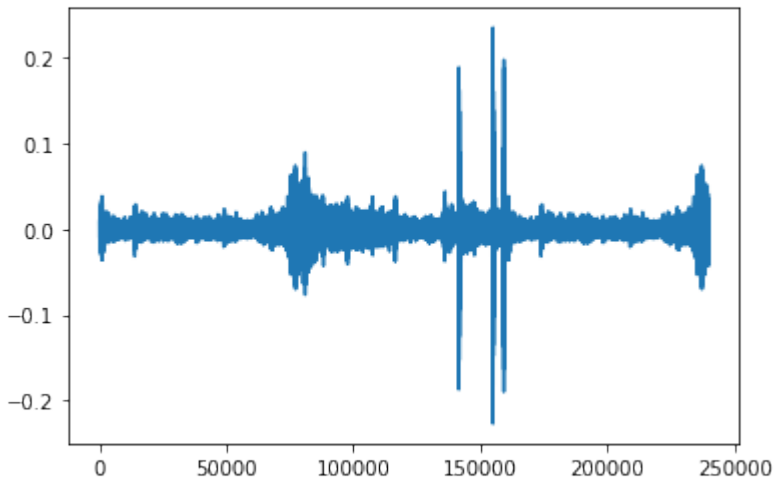


Looping example: create an audio object 1.5x as long, extending the end by looping

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

```
7.5
```

[22]:
```
[<matplotlib.lines.Line2D at 0x7fbb1eba8760>]
```

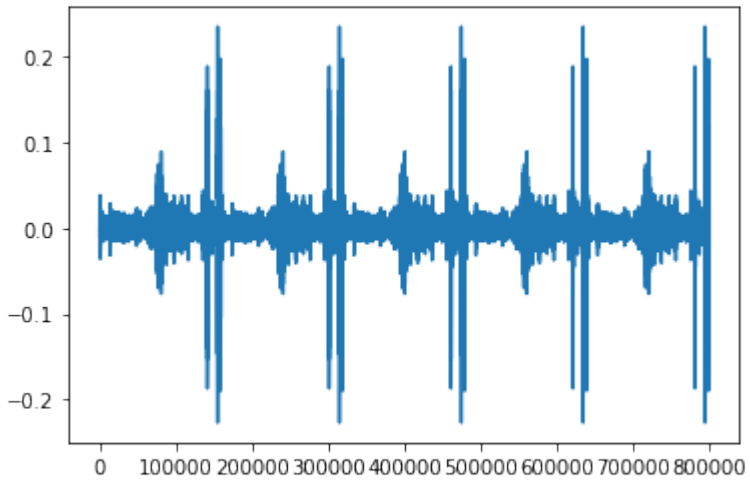


create an audio object that loops the original object 5 times and plot the samples

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

```
25.0
```

[23]:
```
[<matplotlib.lines.Line2D at 0x7fbb1df0a640>]
```



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
```

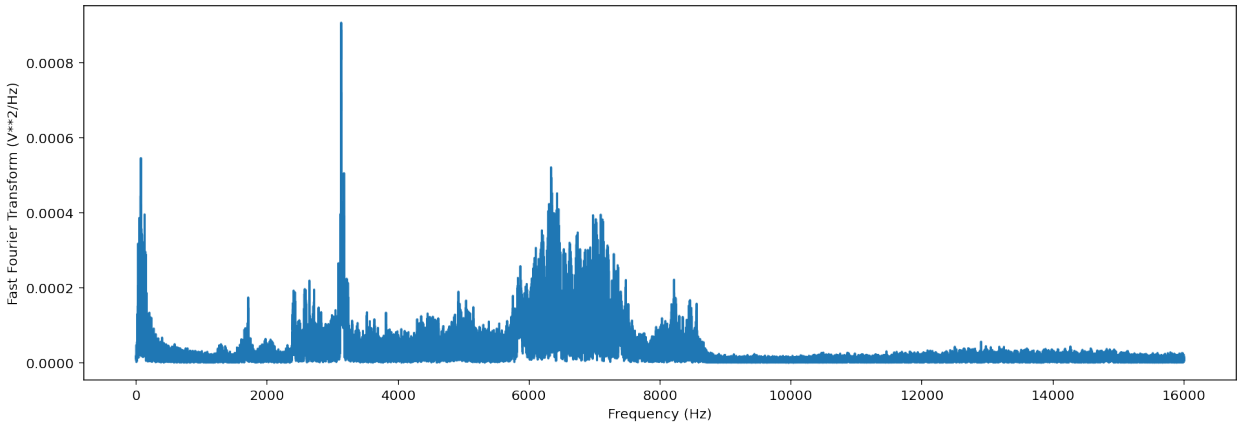
5.3.8 Generate a frequency spectrum

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

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/matplotlib_
↪inline/config.py:75: 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()
```

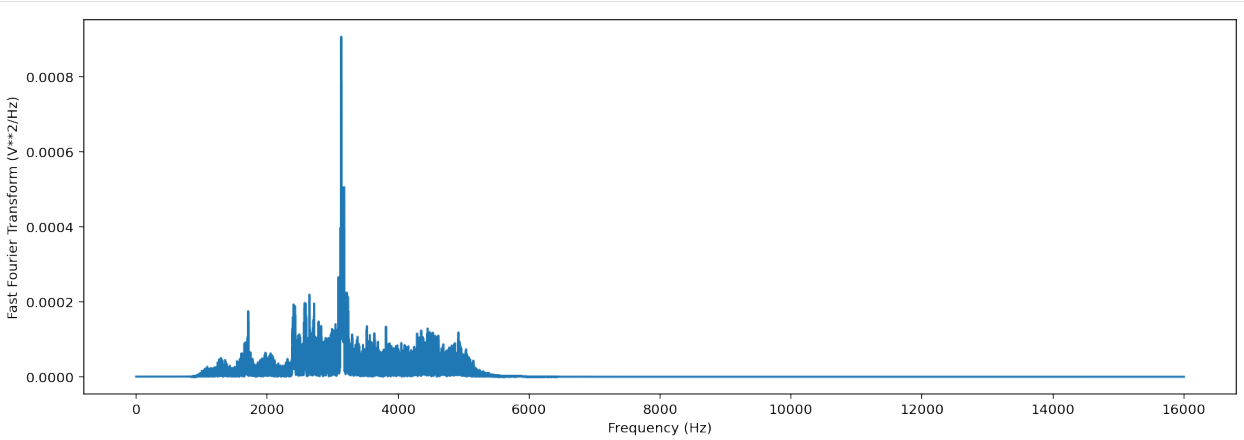
(continues on next page)

(continued from previous page)

```
# 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.

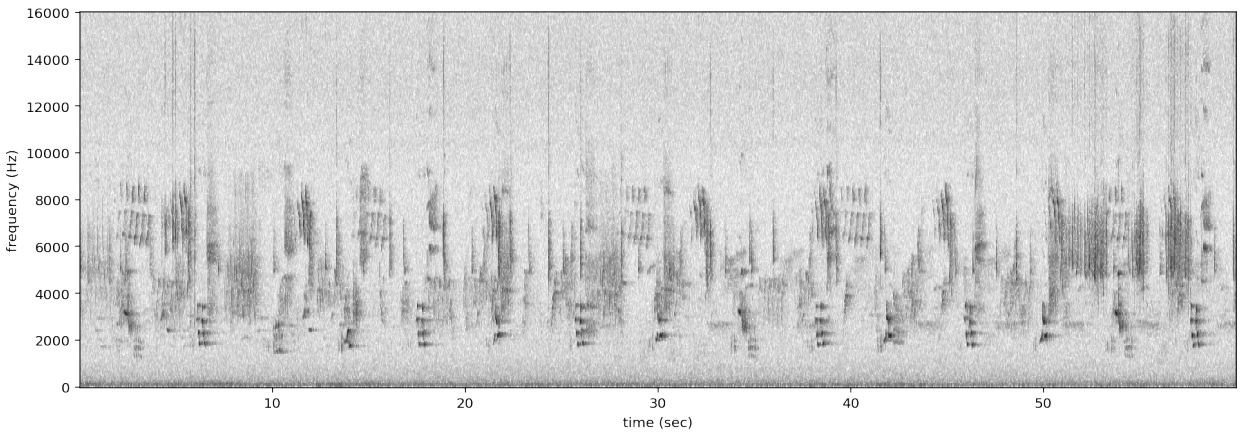
[28]:
```
spec = Spectrogram.from_audio(Audio.from_file(audio_filename))
print(f'the first few times: {spec.times[0:5]}')
print(f'the first few frequencies: {spec.frequencies[0:5]}')
```

```
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).

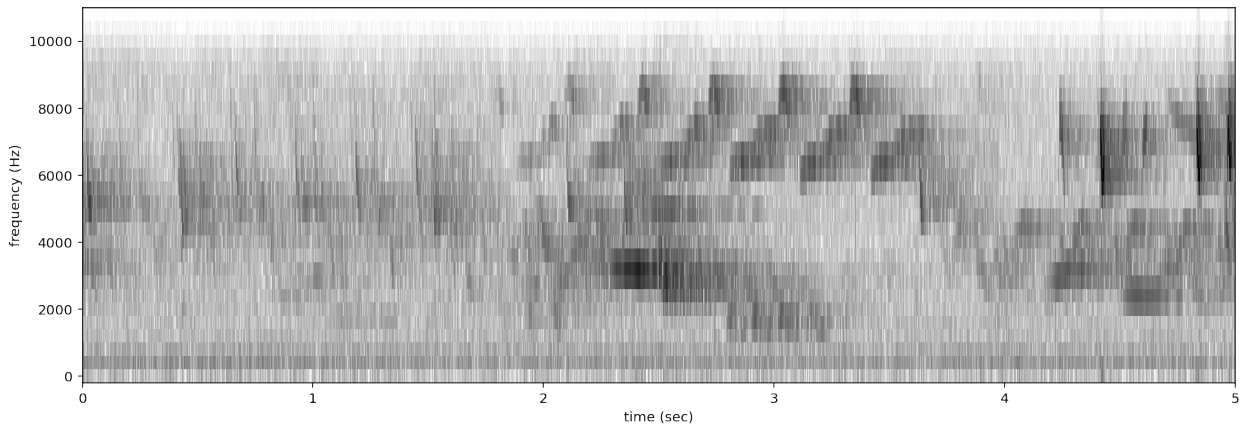
As an example, let's create two spectrograms, one with hight time resolution and another with high frequency resolution.

[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.

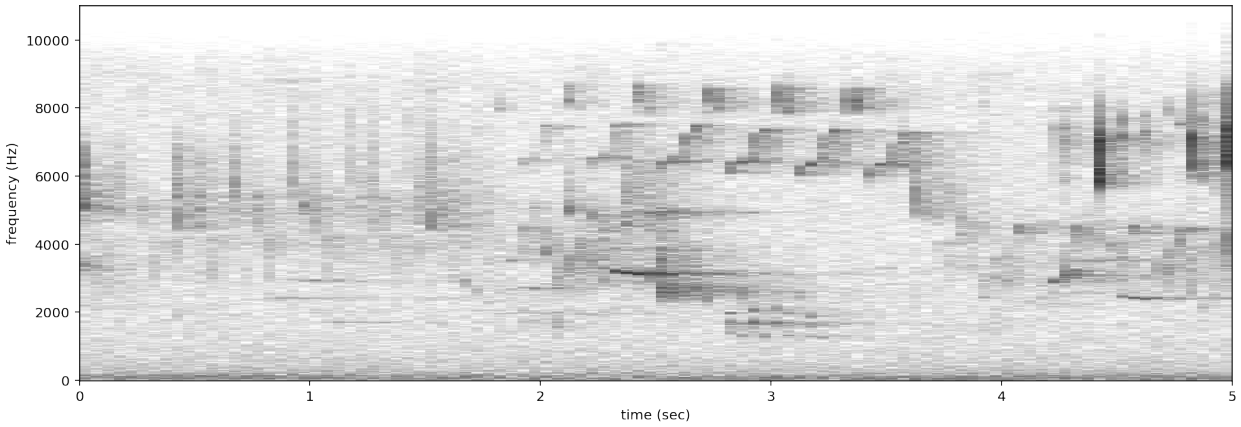
[31]:
```
spec = Spectrogram.from_audio(audio, window_samples=55, overlap_samples=0)
spec.plot()
```



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.

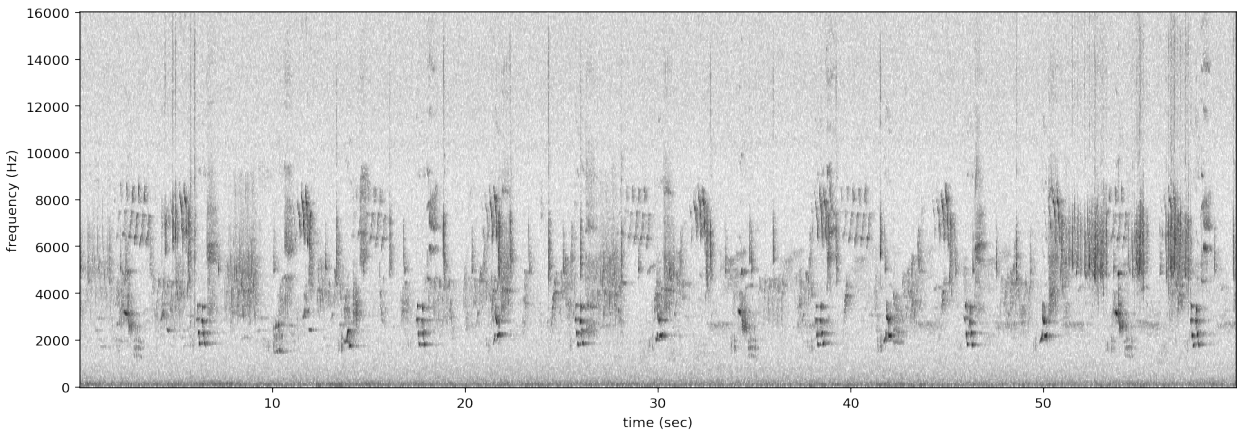
5.5 Spectrogram methods

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()
```

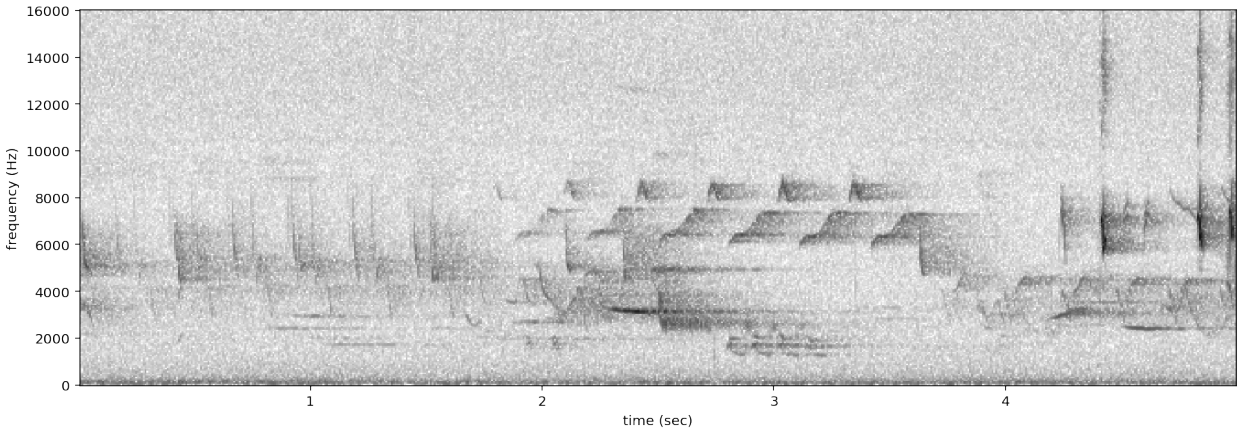


5.5.2 Load modified audio

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)
```

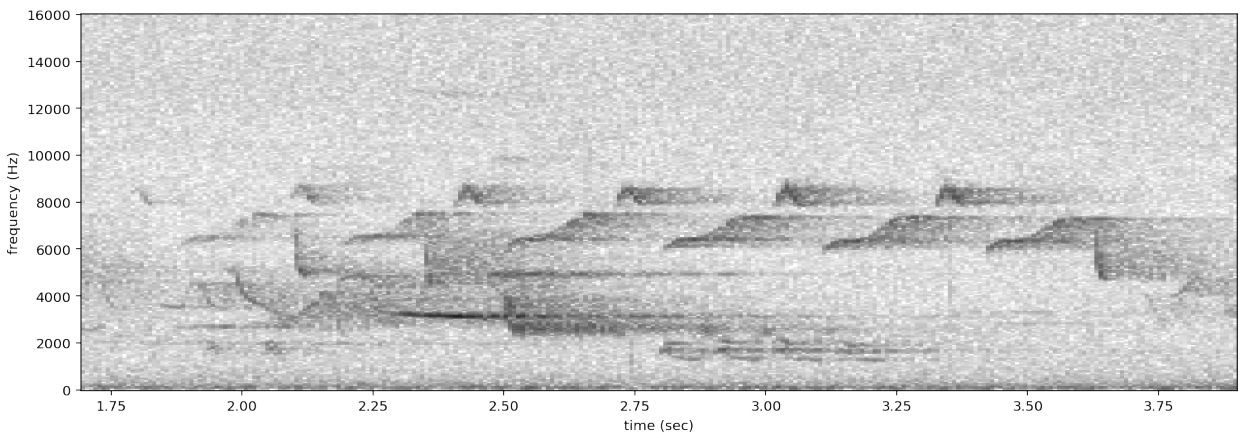
Delete the files created above.

[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

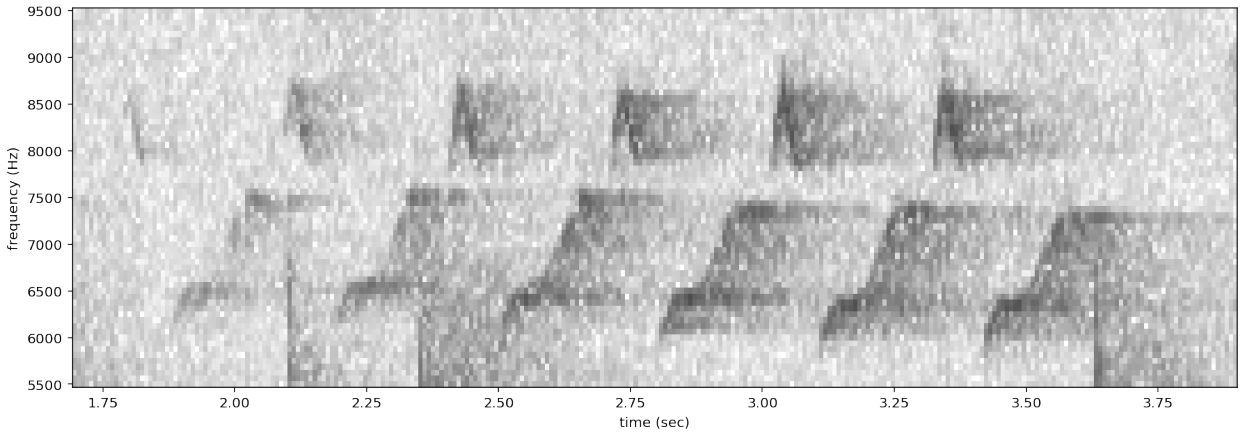
Spectrograms can be trimmed in frequency using `bandpass()`. This simply subsets the Spectrogram array rather than performing an audio-domain filter.

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()
```

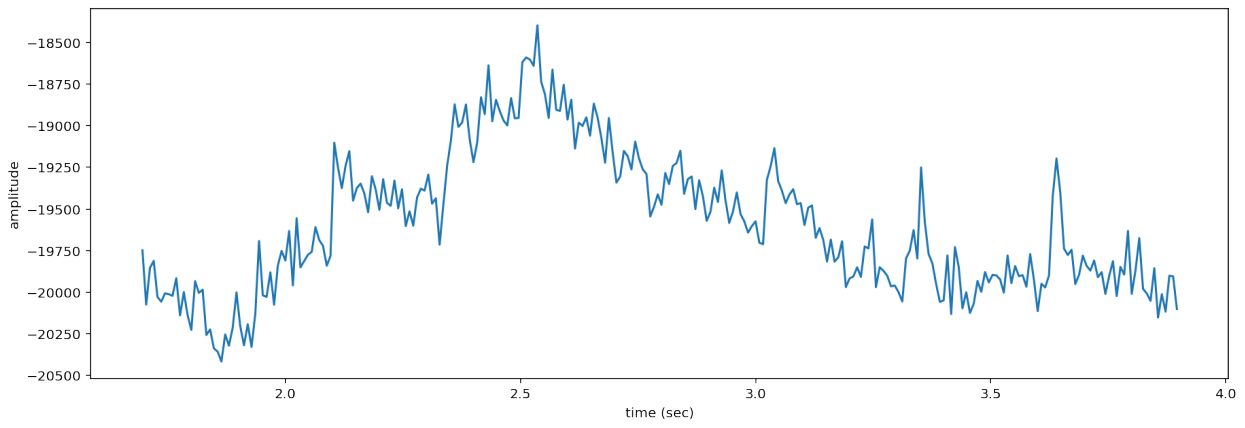


5.5.6 Sum the columns of a spectrogram

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

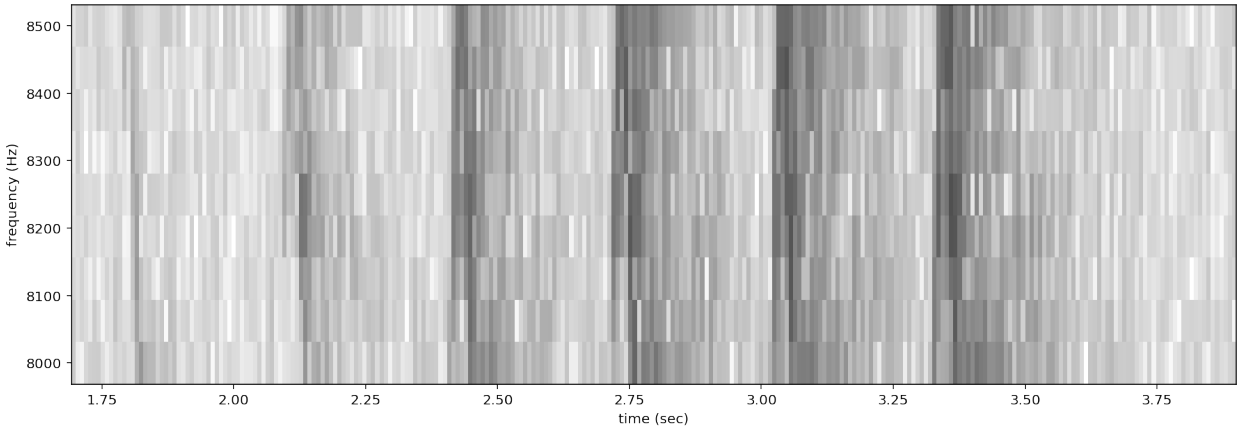
Note: the amplitude of the Spectrogram (and FFT) has units of power (V**2) over frequency (Hz) on a logarithmic scale

[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()
```



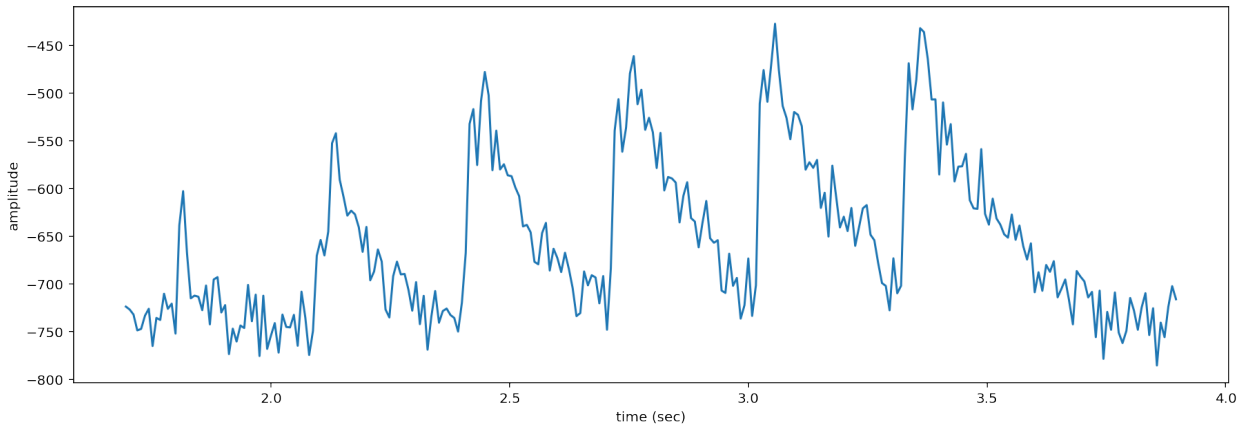
It is also possible to get the amplitude signal from a restricted range of frequencies, for instance, to look at the amplitude in the frequency range of a species of interest. For example, get the amplitude signal from the 8000 Hz to 8500 Hz range of the audio (displayed below):

[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
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      6      0 --:--:--  0:00:01 --:--:--     0
100 5432k  100 5432k    0     0  2393k      0  0:00:02  0:00:02 --:--:-- 11.0M
```

[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.

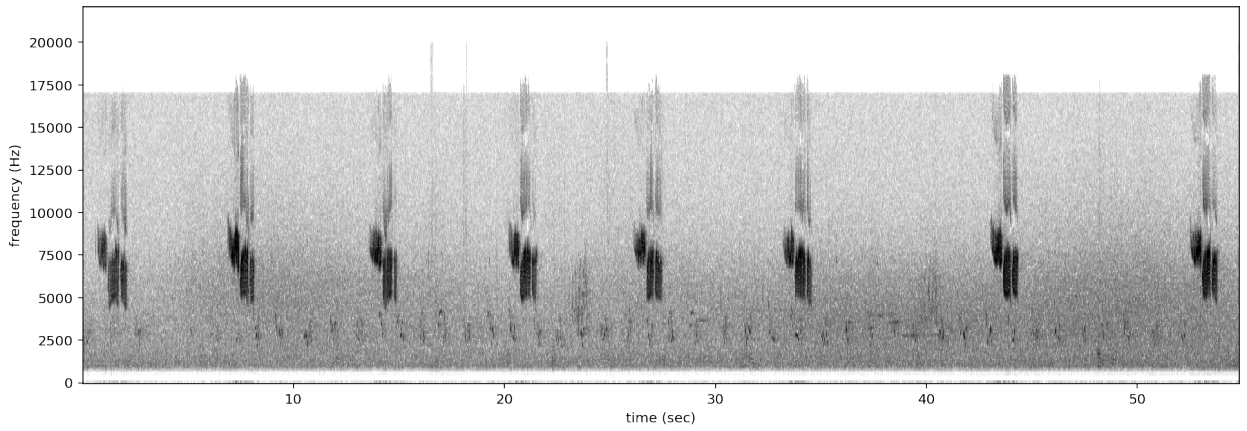
Note that we need to specify the name of the column containing annotations, since it can be named anything in Raven. The column will be renamed to "annotation".

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()
```



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')

#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
```

we could instead choose to load it with only the necessary columns, plus 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 ("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

Specify a list of classes to include in the subset

for example, we can subset to only annotations marked as '?'

[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 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 Splitting annotations along with audio

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).

We can accomplish this in three ways:

(1) Split the annotations (`.one_hot_labels_like()`) using the DataFrame returned by `Audio.split()` (this dataframe includes the start and end times of each clip)

(2) Create a dataframe of start and end times, and split the audio accordingly

(3) directly split the labels with `.one_hot_clip_labels()`, using splitting parameters that match Audio.split()

All three methods are demonstrated below.

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

After splitting audio with audio.split(), we'll use BoxedAnnotation's `one_hot_labels_like()` function to extract the labels for each audio clip. This function requires that we specify the minimum overlap of the label (in

seconds) with the clip for the clip to be labeled positive. We also specify the list of classes for one-hot labels (if we give classes=None, it will make a column for every unique label in the annotations).

[12]:
```
# load the Audio and Annotations
audio = Audio.from_file(audio_file)
annotations = BoxedAnnotations.from_raven_file(annotation_file,annotation_column=
↪'Species')

# 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, clip_overlap=0.0)

labels_df = annotations.one_hot_labels_like(clip_df,min_label_overlap=0.25,classes=[
↪'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()
```

[12]:
```
                     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 into labels (without audio splitting)

The function in the previous example, `one_hot_labels_like()`, splits the labels according to start and end times from a DataFrame. But how would we get that DataFrame if we aren't actually splitting Audio files?

We can create the dataframe with a helper function that takes the same splitting parameters as Audio.split(). Notice that we need to specify one additional parameter: the entire duration to be split (`full_duration`).

[13]:
```
# 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)

#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,classes=[
↪'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()
```

[13]:
```
                     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 3. Split annotations directly using splitting parameters

Though we recommend using one of the above methods, you can also split annotations by directly calling `one_hot_clip_labels()`. This method combines the two steps in the examples above (creating a clip df and splitting the annotations), and requires that you specify the parameters for both of those steps.

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

[14]:
```
labels_df = annotations.one_hot_clip_labels(
    full_duration=60,
    clip_duration=5,
    clip_overlap=0,
    classes=['GWWA_song'],
    min_label_overlap=0.25,
)
labels_df.head()
```

[14]:
```
                      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
```

6.6.1 Create audio clips and one-hot labels from many audio and annotation files

Let's get to the useful part - you have tons of audio files (with corresponding Raven files) and you need to create one-hot labels for 5 second clips on all of them. Can't we just give you the code you need to get this done?!

Sure :)

but be warned, matching up the correct raven and audio files might require some finagling

6.7 find all the Raven and audio files, and see if they match up one-to-one

caveat: you'll need to be careful about matching up the correct Raven files and audio files. In this example, we'll assume our Raven files have exactly the same name (ignoring the extensions like ".Table.1.selections.txt") as our audio files, *and* that these file names *are unique (!)* - that is, no two audio files have the same name.

[15]:
```
# 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 txt files are Raven files!)
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)
audio_files = glob(f"{audio_files_dir}/*.wav")+glob(f"{audio_files_dir}/*.WAV")+glob(f
↪"{audio_files_dir}/*.mp3")
```

(continues on next page)

(continued from previous page)

```
print(f"found {len(audio_files)} audio files")

# 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)]
```

```
found 3 annotation files
found 3 audio files

 audio files with duplicate names:
```

[15]:
```
Empty DataFrame
Columns: [audio_file]
Index: []
```

[16]:
```
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:
```

[16]:
```
                                                raven_file
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
13738  ./gwwa_audio_and_raven_annotations/GWWA_XC_Ann...
```

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

[17]:
```
#remove the second selection table for file 13738.wav
raven_df=raven_df[raven_df.raven_file.apply(lambda x: "selections2" not in x)]
```

[18]:
```
paired_df = audio_df.join(raven_df,how='outer')
```

check if any audio files don't have annotation files

[19]:
```
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
```

[19]:
```
                                              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

[20]:
```
#look at unmatched raven files
print(f"raven files without audio file: {len(paired_df[paired_df.audio_file.
↪apply(lambda x:x!=x)])}")
```

(continues on next page)

(continued from previous page)

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

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

[20]:
```
      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

[21]:
```
paired_df = paired_df.dropna()
```

[22]:
```
paired_df
```

[22]:
```
                                            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.

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.

Note: it 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.

[23]:
```
%%bash
mkdir -p ./temp_clips
```

[24]:
```
#choose settings for audio splitting
clip_duration = 3
clip_overlap = 0
final_clip = None
clip_dir = './temp_clips'

#choose settings for annotation splitting
classes = None#['GWWA_song','GWWA_dzit'] #list of all classes, or None
min_label_overlap = 0.1


#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():
```

(continues on next page)

(continued from previous page)

```
    #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 = annotations.one_hot_labels_like(clip_df,classes=classes,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
    labels = labels.reset_index(level=[1,2],drop=True)
    all_labels.append(labels)

    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)
```

[25]:
```
all_labels
```

[25]:

file	?	GWWA_song
./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

(continues on next page)

(continued from previous page)

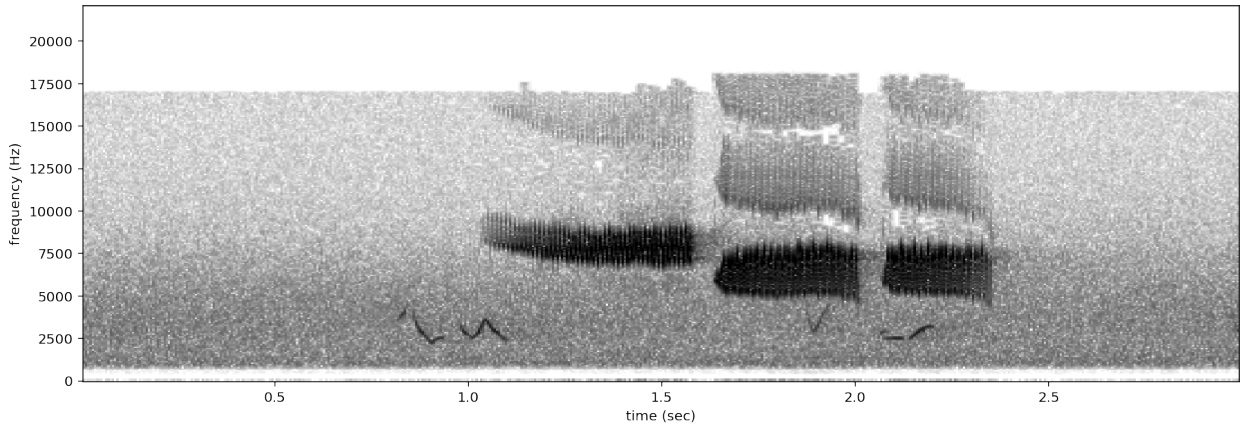
```
./temp_clips/13738_30.0s_33.0s.wav  0.0         0.0
./temp_clips/13738_33.0s_36.0s.wav  0.0         1.0
./temp_clips/13738_36.0s_39.0s.wav  0.0         0.0
./temp_clips/13738_39.0s_42.0s.wav  0.0         0.0
./temp_clips/13738_42.0s_45.0s.wav  0.0         1.0
./temp_clips/13738_45.0s_48.0s.wav  0.0         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
```

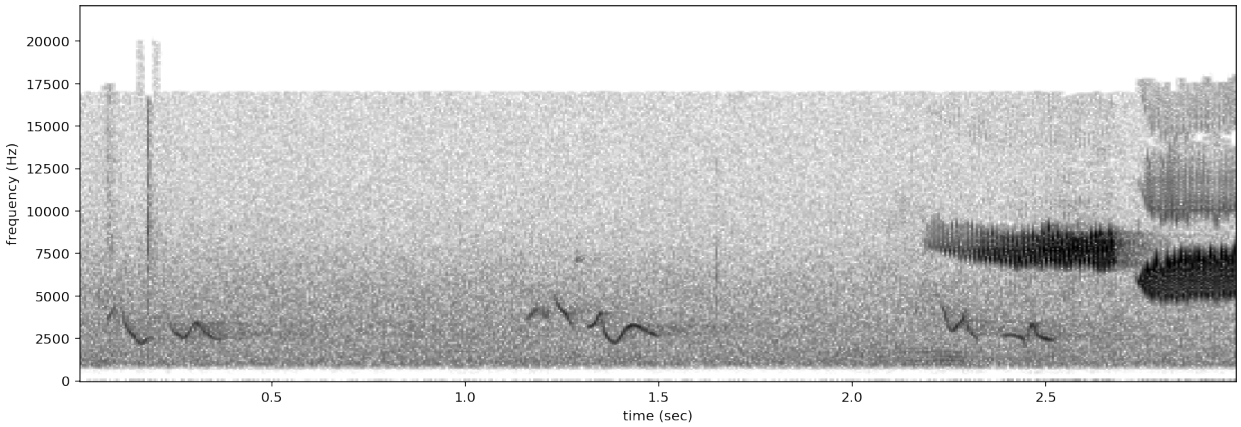
6.9 sanity check: look at spectrograms of clips labeled 0 and 1

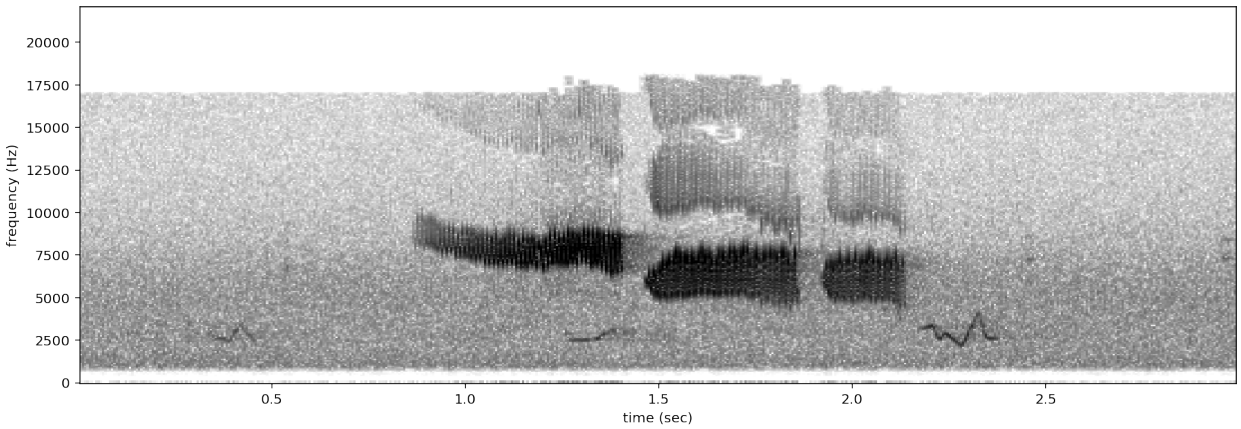
[26]:
```
# ignore the "?" annotations for this visualization
all_labels = all_labels[all_labels["?"]==0]
```

[27]:
```
# 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()
```

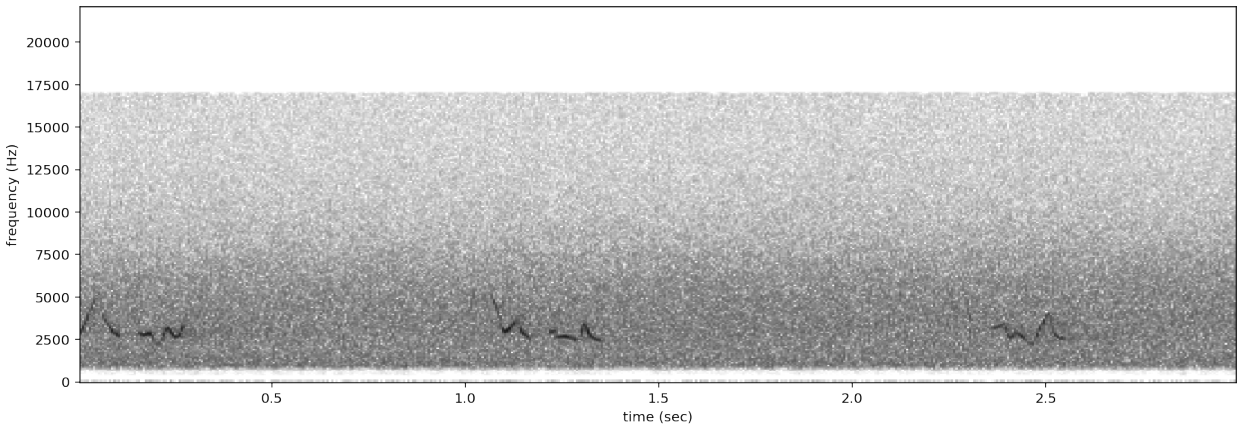
```
spectrograms of 3 random positive clips (label=1)
```

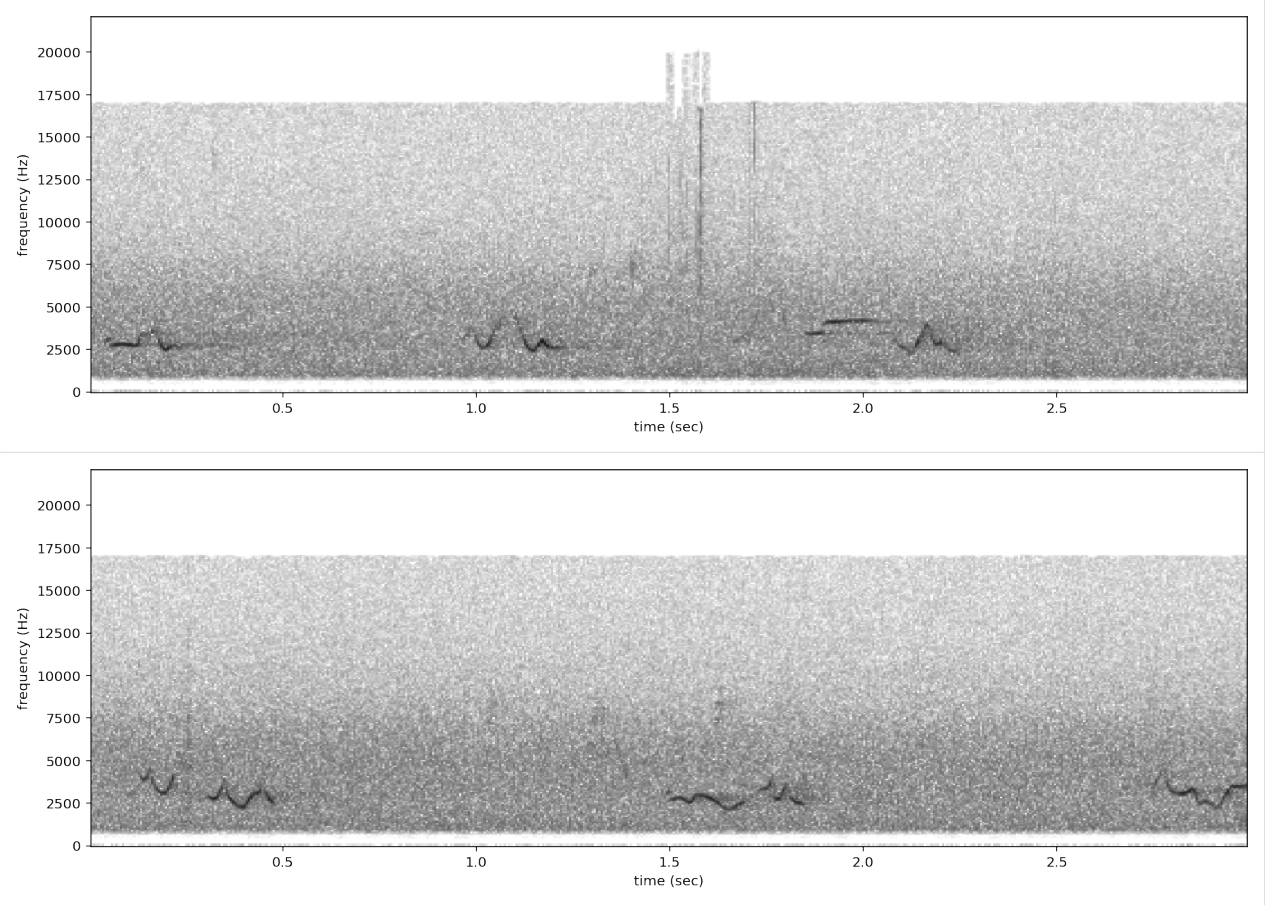







```
spectrogram of 3 random negative clips (label=0)
```





clean up: remove temp_clips directory

[28]:
```
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')
```

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      5      0 --:--:--  0:00:01 --:--:--     0
100 3750k  100 3750k    0     0   993k      0  0:00:03  0:00:03 --:--:-- 2265k
```

[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	-0.885565	-0.716252
	5.0	10.0	-0.927523	-0.182728
	10.0	15.0	-0.825782	-0.504427
	15.0	20.0	-1.131475	-0.679929
	20.0	25.0	-0.782559	-0.597284

7.3 Overlapping prediction clips

[9]:
```
scores, _, _ = model.predict(audio_files, overlap_fraction=0.5)
scores.head()
```

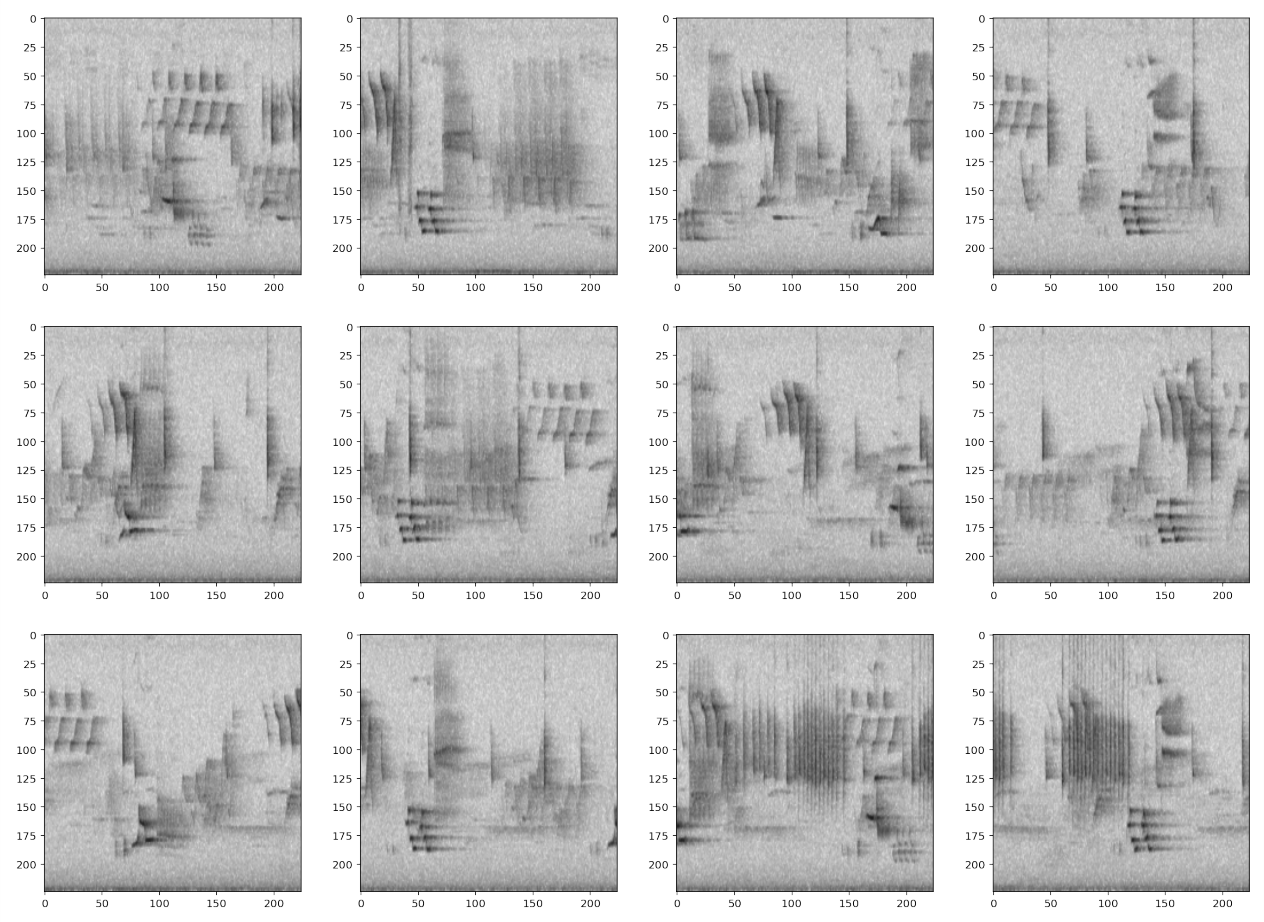
```
[]
```

[9]:

file	start_time	end_time	classA	classB
./1min_audio.wav	0.0	5.0	-0.885565	-0.716252
	2.5	7.5	-1.184270	-0.451556
	5.0	10.0	-0.927523	-0.182728
	7.5	12.5	-0.955600	-0.627236
	10.0	15.0	-0.825782	-0.504427

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)
```



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 use the `predict()` function for post-processing of predictions 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 use the `binary_preds` argument to generate 0/1 predictions for each sample and class. For presence/absence models, use the option `binary_preds='single_target'`. For multi-class models, think about whether each clip should be labeled with only one class (single target) or whether each clip could contain multiple classes (`binary_preds='multi_target'`)

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

```
[]
```

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      0.457773  0.542227
                 5.0        10.0     0.321957  0.678043
                 10.0       15.0     0.420345  0.579655
                 15.0       20.0     0.388993  0.611007
                 20.0       25.0     0.453813  0.546187
```

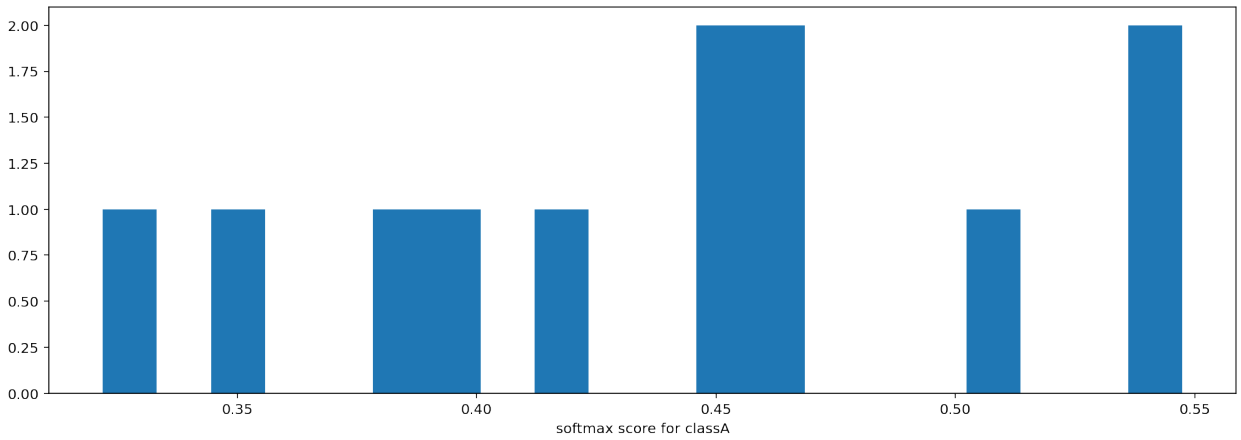
We also have an additional output, the binary 0/1 ("absent" vs "present") predictions generated by the model:

[13]:
```
binary_predictions.head()
```

[13]:
```
                                    classA  classB
file             start_time end_time
./1min_audio.wav 0.0        5.0        0.0     1.0
                 5.0        10.0       0.0     1.0
                 10.0       15.0       0.0     1.0
                 15.0       20.0       0.0     1.0
                 20.0       25.0       0.0     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/lglpty35omjhmq6cdz8cfudm43nn2t9f.
↪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      6      0 --:--:--  0:00:01 --:--:--     0
100 42.9M  100 42.9M    0     0  6247k      0  0:00:07  0:00:07 --:--:-- 10.4M
```

[15]:
```
CompletedProcess(args=['curl', 'https://pitt.box.com/shared/static/
↪lglpty35omjhmq6cdz8cfudm43nn2t9f.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)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

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

```
/Users/SML161/opensoundscape/opensoundscape/torch/models/cnn.py:1165: 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	2.994287
	5.0	10.0	3.646061
	10.0	15.0	3.004456
	15.0	20.0	3.158881
	20.0	25.0	2.755805

			acanthis-flammea-present
file	start_time	end_time	
./1min_audio.wav	0.0	5.0	-3.261499
	5.0	10.0	-3.892005

(continues on next page)

(continued from previous page)

```
10.0    15.0    -2.484303
15.0    20.0    -3.852344
20.0    25.0    -2.693598
```

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.x

If you saved a model with OpenSoundscape 0.6.x 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:

```
#OpenSoundscape version 0.6.x
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.

```
#newer OpenSoundscape version
model_dict = torch.load('/path/to/model_dict.pt')
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'])

#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.

7.6.3 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()
```

```
sys:1: ResourceWarning: unclosed socket <zmq.Socket(zmq.PUSH) at 0x7facfcfcc760>
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

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 --:--:--     0
100 9499k  100 9499k    0     0  4778k      0  0:00:01  0:00:01 --:--:-- 20.5M
```

[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 use the `categorical_to_one_hot` function from `opensoundscape.annotations` to create "one hot" labels - that is, a column for every class, with 1 for present or 0 for absent in each sample's row. In this case, our classes are simply `'negative'` for files without a woodcock and `'positive'` for files with a woodcock.

We'll need to put the paths to audio files as the index of the DataFrame.

Note that these classes are mutually exclusive, so we have a "single-target" problem, as opposed to a "multi-target" problem where multiple classes can simultaneously be present.

[6]:
```
from opensoundscape.annotations import categorical_to_one_hot
one_hot_labels, classes = categorical_to_one_hot(specky_table[['woodcock']].values)
labels = pd.DataFrame(index=specky_table['filename'],data=one_hot_labels,
↪columns=classes)
labels.head()
```

[6]:

filename	absent	present
./woodcock_labeled_data/d4c40b6066b489518f8da83...	0	1
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...	1	0
./woodcock_labeled_data/79678c979ebb880d5ed6d56...	0	1
./woodcock_labeled_data/49890077267b569e142440f...	0	1
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...	0	1

If we want to, we can always convert one_hot labels back to categorical labels:

[7]:
```
from opensoundscape.annotations import one_hot_to_categorical
categorical_labels = one_hot_to_categorical(one_hot_labels,classes)
categorical_labels[:3]
```

[7]:
```
[['present'], ['absent'], ['present']]
```

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.

[8]:
```
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 two-class, single-target model

This demonstrates using a two class, single-target model.

- The two classes in this case are "positive" and "negative"
- The model is "single target", meaning that each sample belongs to exactly one class, "positive" or "negative"

We usually use two-class, single-target models to predict the presence or absence of a single species. We often refer to this as a "binary" model, but be careful not to confuse this for thresholded "binary" output predictions (1 or 0).

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.

[9]:
```
# Create model object
classes = train_df.columns
model = CNN('resnet18',classes=classes,sample_duration=2.0,single_target=True)
```

8.2.2 Inspect training images

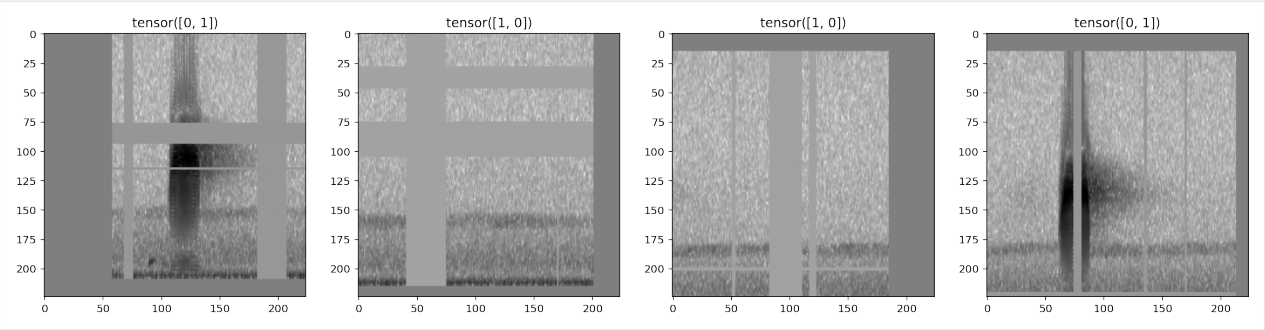
Before creating a machine learning algorithm, we strongly recommend making sure the images coming out of the preprocessor look like you expect them to. Here we generate images for a few samples.

[10]:
```
#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

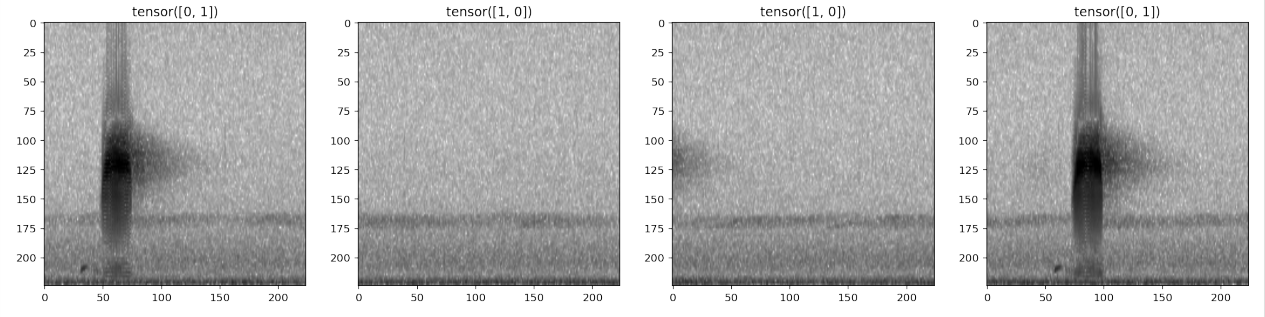
[11]:
```
#pick some random samples from the training set

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).
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:

[12]:
```
#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.

[13]:
```
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,
    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
```

(continues on next page)

(continued from previous page)

```
)


Training Epoch 0
Epoch: 0 [batch 0/3 (0.00%)]
        DistLoss: 0.811
Metrics:
Metrics:
        MAP: 0.508

Validation.
[]
Metrics:
        MAP: 0.500

Training Epoch 1
Epoch: 1 [batch 0/3 (0.00%)]
        DistLoss: 0.889
Metrics:
Metrics:
        MAP: 0.473

Validation.
[]
Metrics:
        MAP: 0.500

Training Epoch 2
Epoch: 2 [batch 0/3 (0.00%)]
        DistLoss: 0.375
Metrics:
Metrics:
        MAP: 0.671

Validation.
[]
Metrics:
        MAP: 0.500

Training Epoch 3
Epoch: 3 [batch 0/3 (0.00%)]
        DistLoss: 0.255
Metrics:
Metrics:
        MAP: 0.484

Validation.
[]
Metrics:
        MAP: 1.000

Training Epoch 4
Epoch: 4 [batch 0/3 (0.00%)]
        DistLoss: 0.833
Metrics:
Metrics:
        MAP: 0.539
```

(continues on next page)

(continued from previous page)

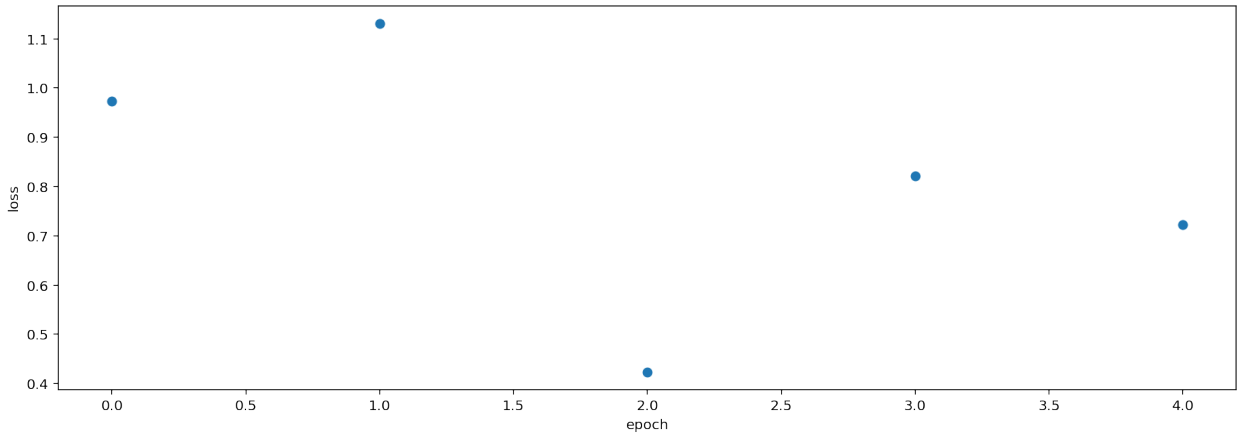
```
Validation.
[]
Metrics:
        MAP: 1.000

Best Model Appears at Epoch 3 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.

[14]:
```
plt.scatter(model.loss_hist.keys(),model.loss_hist.values())
plt.xlabel('epoch')
plt.ylabel('loss')
```

[14]:
```
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:

[15]:
```
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,
```

(continues on next page)

(continued from previous page)

```
    batch_size=8,
    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 three things:

- scores dataframe: numeric predictions from the model for each sample and class (by default these are raw outputs from the model)
- predictions dataframe: 0/1 predictions from the model for each sample and class (only generated if `binary_predictions` argument is supplied)
- unsafe_samples: list of any samples that failed to load properly

Let's predict on the two field recordings:

[16]:
```
from glob import glob
field_recordings = glob('./woodcock_labeled_data/field_data/*')
field_recordings
```

[16]:
```
['./woodcock_labeled_data/field_data/60s_field_data_sample_1.wav',
 './woodcock_labeled_data/field_data/60s_field_data_sample_2.wav']
```

[17]:
```
prediction_scores_df, prediction_binary_df, unsafe_samples = 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:

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

[18]:
```
                                                                    absent  \
file                                                start_time end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0        2.0      -1.584276
                                                    2.0        4.0      -1.434084
                                                    4.0        6.0      -1.290440
                                                    6.0        8.0      -1.509710
                                                    8.0        10.0     -1.339448

                                                                   present
file                                                start_time end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0        2.0       0.440277
                                                    2.0        4.0       0.147501
                                                    4.0        6.0       0.086156
                                                    6.0        8.0       0.476758
                                                    8.0        10.0      0.154228
```

Binary 0/1 predictions were not generated because the `binary_predictions` argument was not supplied

The `prediction_binary_df` dataframe is `None` - this is because we haven't specified an option for the `binary_preds` argument of predict. We can choose between `'single_target'` prediction (always predict the highest scoring class and no others) or `'multi_target'` (predict 1 for all classes exceeding a threshold).

[19]:
```
prediction_binary_df
```

If all of the samples were processed without errors, the unsafe_samples list will be empty

[20]:
```
unsafe_samples
```

[20]:
```
[]
```

8.3.2 Create presence/absence (0/1) predictions

Supplying the `binary_preds` argument returns a dataframe in which the scores are transformed from continuous numbers to either 0 or 1.

Note: Binary 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.

If you wish to output binary predictions, three options are available:

- `None`: default. do not create or return binary predictions
- `'single_target'`: predict that the highest-scoring class = 1, all others = 0
- `'multi_target'`: provide a `threshold`. Scores above threshold = 1, others = 0

For instance, using the option `'single_target'` chooses whichever of `'negative'` or `'positive'` is higher.

[21]:
```
scores,preds,labels = model.predict(field_recordings,binary_preds='single_target')
preds.head()
```

```
[]
```

[21]:
```
                                                                  absent  \
file                                                start_time end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0        2.0          0.0
```

(continues on next page)

(continued from previous page)

```
                                                    2.0         4.0          0.0
                                                    4.0         6.0          0.0
                                                    6.0         8.0          0.0
                                                    8.0         10.0         0.0

                                                                          present
file                                                start_time  end_time
./woodcock_labeled_data/field_data/60s_field_da... 0.0         2.0           1.0
                                                    2.0         4.0           1.0
                                                    4.0         6.0           1.0
                                                    6.0         8.0           1.0
                                                    8.0         10.0          1.0
```

The `'multi_target'` option allows you to select a threshold. If a score meets that threshold, the binary prediction is 1; otherwise, it is 0.

Each score will have a function applied to it that takes the score from the real numbers, (-inf, inf), to the range [0, 1] (specifically the logistic sigmoid, or expit function). Whether the score meets this threshold will be based off of the sigmoid, not the raw score.

[22]:
```
score_df, pred_df, label_df = model.predict(
    validation_df,
    binary_preds='multi_target',
    threshold=0.99,
)
pred_df.head()
```

```
[]
```

[22]:
```
                                                                    absent  \
file                                                start_time  end_time
./woodcock_labeled_data/882de25226ed989b31274ee... 0.0         2.0           0.0
./woodcock_labeled_data/92647ab903049a9ee4125ab... 0.0         2.0           0.0
./woodcock_labeled_data/75b2f63e032dbd6d1979004... 0.0         2.0           0.0
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c... 0.0         2.0           0.0
./woodcock_labeled_data/ad14ac7ffa729060712b442... 0.0         2.0           0.0

                                                                    present
file                                                start_time  end_time
./woodcock_labeled_data/882de25226ed989b31274ee... 0.0         2.0           0.0
./woodcock_labeled_data/92647ab903049a9ee4125ab... 0.0         2.0           1.0
./woodcock_labeled_data/75b2f63e032dbd6d1979004... 0.0         2.0           1.0
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c... 0.0         2.0           0.0
./woodcock_labeled_data/ad14ac7ffa729060712b442... 0.0         2.0           0.0
```

Note that it is possible both the negative and positive classes are predicted to be present. This is because the `'multi_target'` option assumes that the classes are not mutually exclusive. For a presence/absence model like the one above, the `'single_target'` option 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 they sum to 1

In this case, since we are choosing between two mutually exclusive classes, we want to use the `'softmax'` activation.

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).

[23]:
```
valid_scores, valid_preds, unsafe_samples = model.predict(
    validation_df,
    activation_layer='softmax',
    split_files_into_clips=False
)
```

```
[]
```

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

[24]:
```
valid_scores.columns = ['pred_negative','pred_positive']
validation_df.join(valid_scores).sample(5)
```

[24]:

filename	absent	present	pred_negative	pred_positive
./woodcock_labeled_data/4afa902e823095e03ba23eb...	0	1	0.009125	0.990875
./woodcock_labeled_data/882de25226ed989b31274ee...	0	1	0.096479	0.903521
./woodcock_labeled_data/ad14ac7ffa729060712b442...	1	0	0.679203	0.320797
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	0	1	0.000921	0.999079
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	0	1	0.074364	0.925636

we can see that our model hasn't really learned anything yet. It just predicts everything is a negative.

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.

[25]:
```
score_df, pred_df, label_df = model.predict(
    validation_df,
    batch_size=8,
    num_workers=0,
    binary_preds='multi_target',
    threshold = 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:

[26]:
```
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)
```

```
model.single_target: False
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

8.4.1 Train

Training looks the same as in two-class models.

[27]:
```
model.train(
    train_df,
    validation_df,
    save_path='./multilabel_train/',
    epochs=1,
    batch_size=16,
    save_interval=100,
    num_workers=0
)
```

```
Training Epoch 0
Epoch: 0 [batch 0/2 (0.00%)]
        DistLoss: 21.955
Metrics:
```

(continues on next page)

(continued from previous page)

```
Metrics:
        MAP: 0.502

Validation.
[]
Metrics:
        MAP: 0.500

Best Model Appears at Epoch 0 with Validation score 0.500.
```

8.4.2 Predict

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

- What `activation_layer` do you want?
- If outputting binary predictions for each sample and class, is my model single-target (`binary_preds='single_target'`) or multi-target (`binary_preds='multi_target'`)?

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

[28]:
```
train_preds,_,_ = model.predict(train_df,split_files_into_clips=False,)
train_preds.columns = ['pred_negative','pred_positive']
train_df.join(train_preds).tail()
```

```
[]
```

[28]:
```
                                                  absent  present  \
filename
./woodcock_labeled_data/0ab7732b506105717708ea9...       0        1
./woodcock_labeled_data/f87d427bef752f5accbd899...       0        1
./woodcock_labeled_data/24073ce519bf1d24107da8a...       0        1
./woodcock_labeled_data/cd0b8d8a89321046e96abee...       1        0
./woodcock_labeled_data/0fc107ec5e76bf7a98dd207...       1        0

                                                  pred_negative  \
filename
./woodcock_labeled_data/0ab7732b506105717708ea9...     -4.099388
./woodcock_labeled_data/f87d427bef752f5accbd899...     -4.016674
./woodcock_labeled_data/24073ce519bf1d24107da8a...     -5.151122
./woodcock_labeled_data/cd0b8d8a89321046e96abee...     -3.734205
./woodcock_labeled_data/0fc107ec5e76bf7a98dd207...     -3.676885

                                                  pred_positive
filename
./woodcock_labeled_data/0ab7732b506105717708ea9...      1.953143
./woodcock_labeled_data/f87d427bef752f5accbd899...      1.815702
./woodcock_labeled_data/24073ce519bf1d24107da8a...      2.740391
./woodcock_labeled_data/cd0b8d8a89321046e96abee...      1.688036
./woodcock_labeled_data/0fc107ec5e76bf7a98dd207...      1.800463
```

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 train-

ing/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)`

[29]:
```
model1 = CNN('resnet18',classes,2.0,single_target=True)
# 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')
```

```
Training Epoch 0
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

```
Epoch: 0 [batch 0/3 (0.00%)]
        DistLoss: 0.877
Metrics:
Metrics:
        MAP: 0.470

Validation.
[]
Metrics:
        MAP: 0.500

Training Epoch 1
Epoch: 1 [batch 0/3 (0.00%)]
        DistLoss: 0.594
Metrics:
Metrics:
        MAP: 0.522

Validation.
[]
Metrics:
```

(continues on next page)

(continued from previous page)

```
        MAP: 0.500

Training Epoch 2
Epoch: 2 [batch 0/3 (0.00%)]
        DistLoss: 0.441
Metrics:
Metrics:
        MAP: 0.671

Validation.
[]
Metrics:
        MAP: 0.500

Best Model Appears at Epoch 0 with Validation score 0.500.
```

Load

Re-load a saved model with the load_model function:

[30]:
```
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

[31]:
```
# load the saved model
model = load_model('./binary_train/best.model')

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

```
[]
```

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.

[32]:
```
# 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:

[33]:
```
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

First, import some packages.

[1]:
```
# 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
```

Set up plotting

[2]:
```
#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.

[3]:
```
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

9.1 Preparing audio data

9.1.1 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.

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

[4]:
```
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
```

(continues on next page)

(continued from previous page)

```
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 --:--:--     0
100 9499k  100 9499k    0     0  3958k      0  0:00:02  0:00:02 --:--:-- 13.8M
```

[4]:
```
CompletedProcess(args=['rm', 'woodcock_labeled_data.tar.gz'], returncode=0)
```

9.1.2 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:

[5]:
```
# 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()
```

[5]:

file	present	absent
./woodcock_labeled_data/d4c40b6066b489518f8da83...	1	0
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...	0	1
./woodcock_labeled_data/79678c979ebb880d5ed6d56...	1	0
./woodcock_labeled_data/49890077267b569e142440f...	1	0
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...	1	0

9.2 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 (tldr: files are split into clips if the preprocessor's `.clip_times_df` is a dataframe specifying desired clip sample paths and times). 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.

9.2.1 Initialize preprocessor

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

[6]:
```
pre = SpectrogramPreprocessor(sample_duration=2.0)
```

9.3 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`:

[7]:
```
dataset = AudioFileDataset(labels,pre)
```

9.3.1 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].

[8]:
```
dataset[0] #loads and preprocesses the sample at row 0 of dataset.df
```

[8]:
```
{'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],
```

(continues on next page)

(continued from previous page)

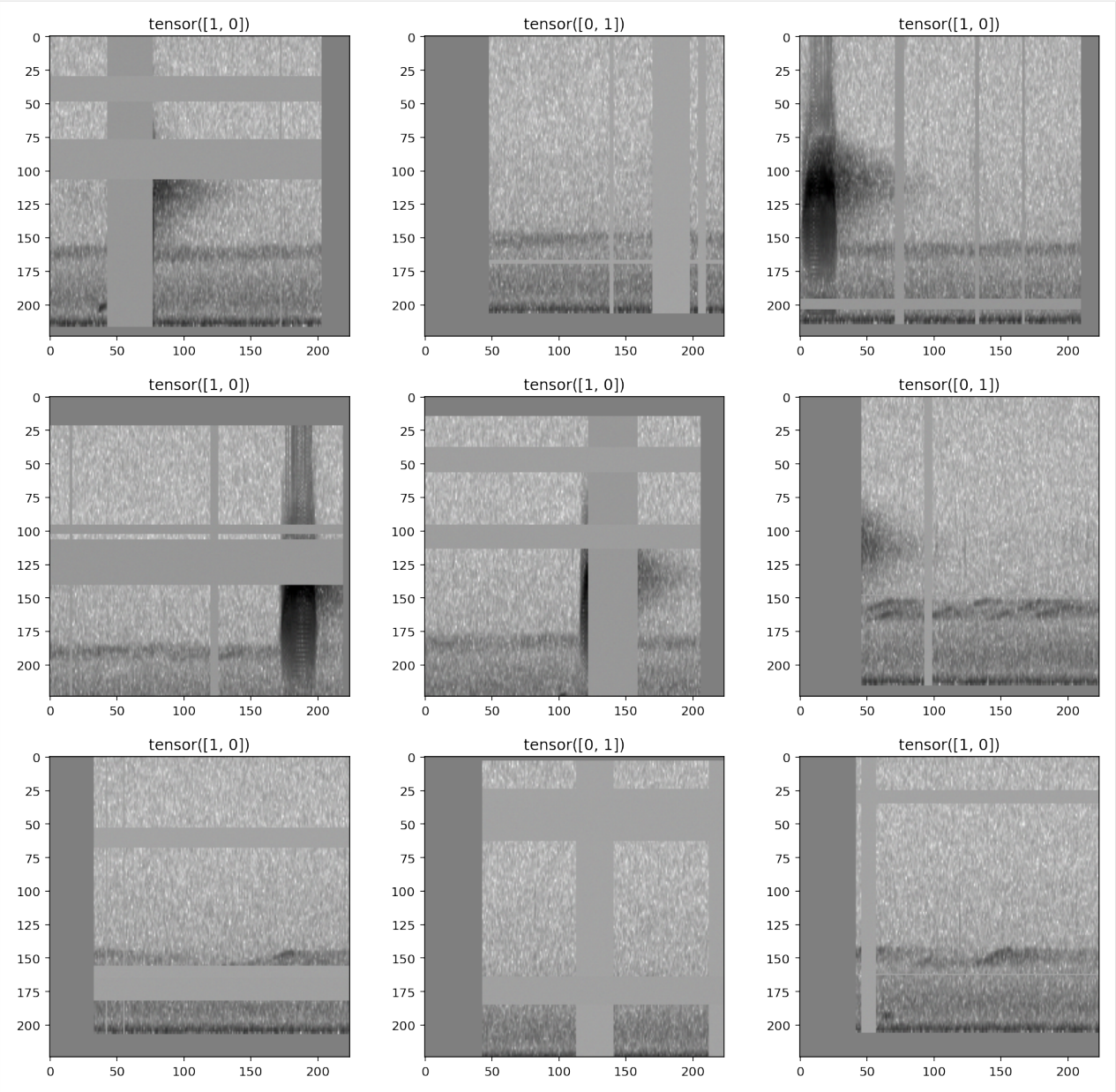
```
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

         [[ 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])}
```

Visualize multiple samples

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.

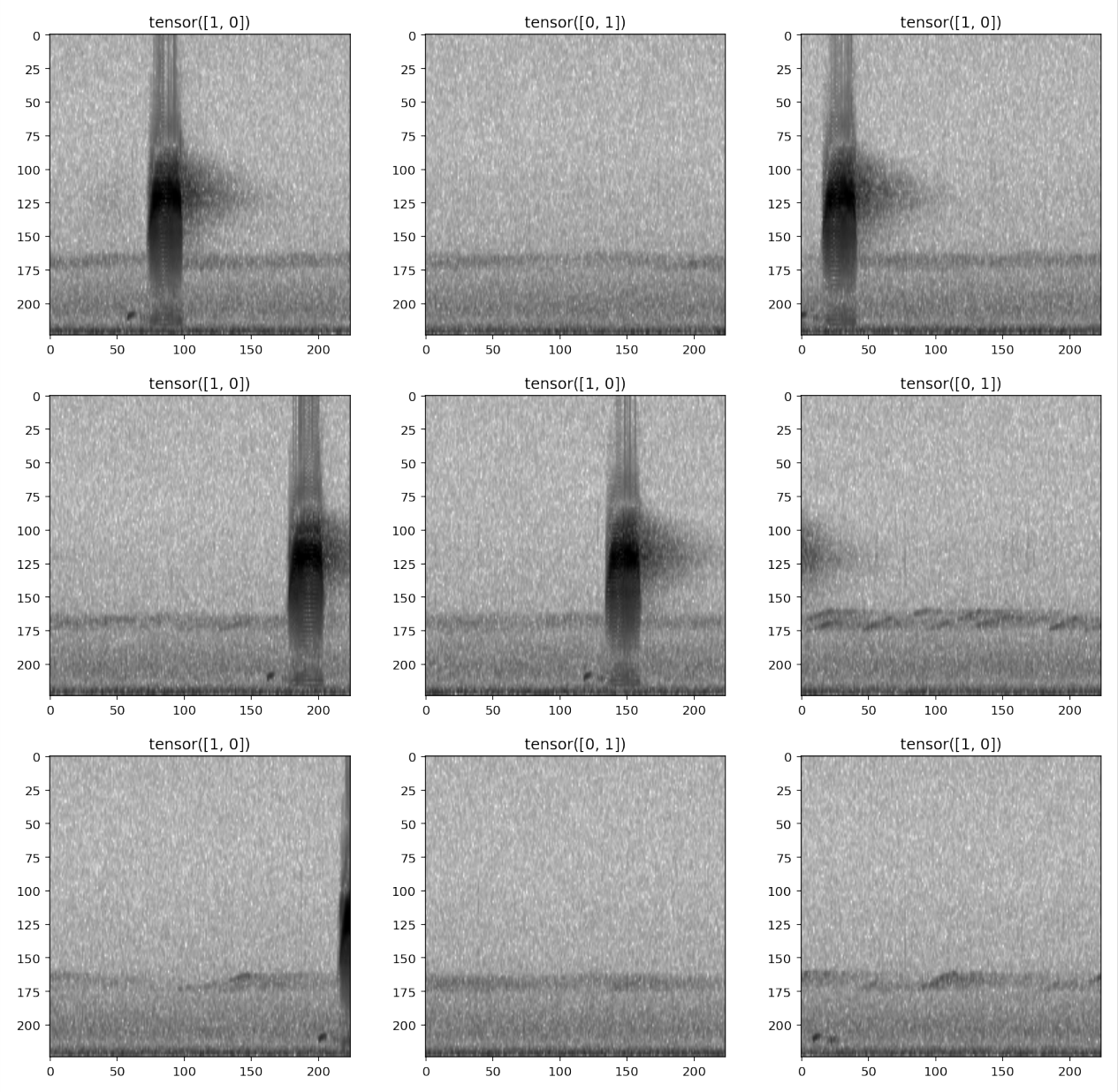
[9]:
```
from opensoundscape.preprocess.utils import show_tensor_grid

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)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
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

[10]:
```
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)
```



DataLoaders and batch sizes

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.

9.3.2 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)

[11]:
```
len(dataset)
```
[11]:
```
29
```

Select the first 10 samples (non-random)

[12]:
```
len(dataset.head(10))
```
[12]:
```
10
```

Randomly select an absolute number of samples

[13]:
```
len(dataset.sample(n=10))
```
[13]:
```
10
```

Randomly select a fraction of samples

[14]:
```
len(dataset.sample(frac=0.5))
```
[14]:
```
14
```

9.4 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)

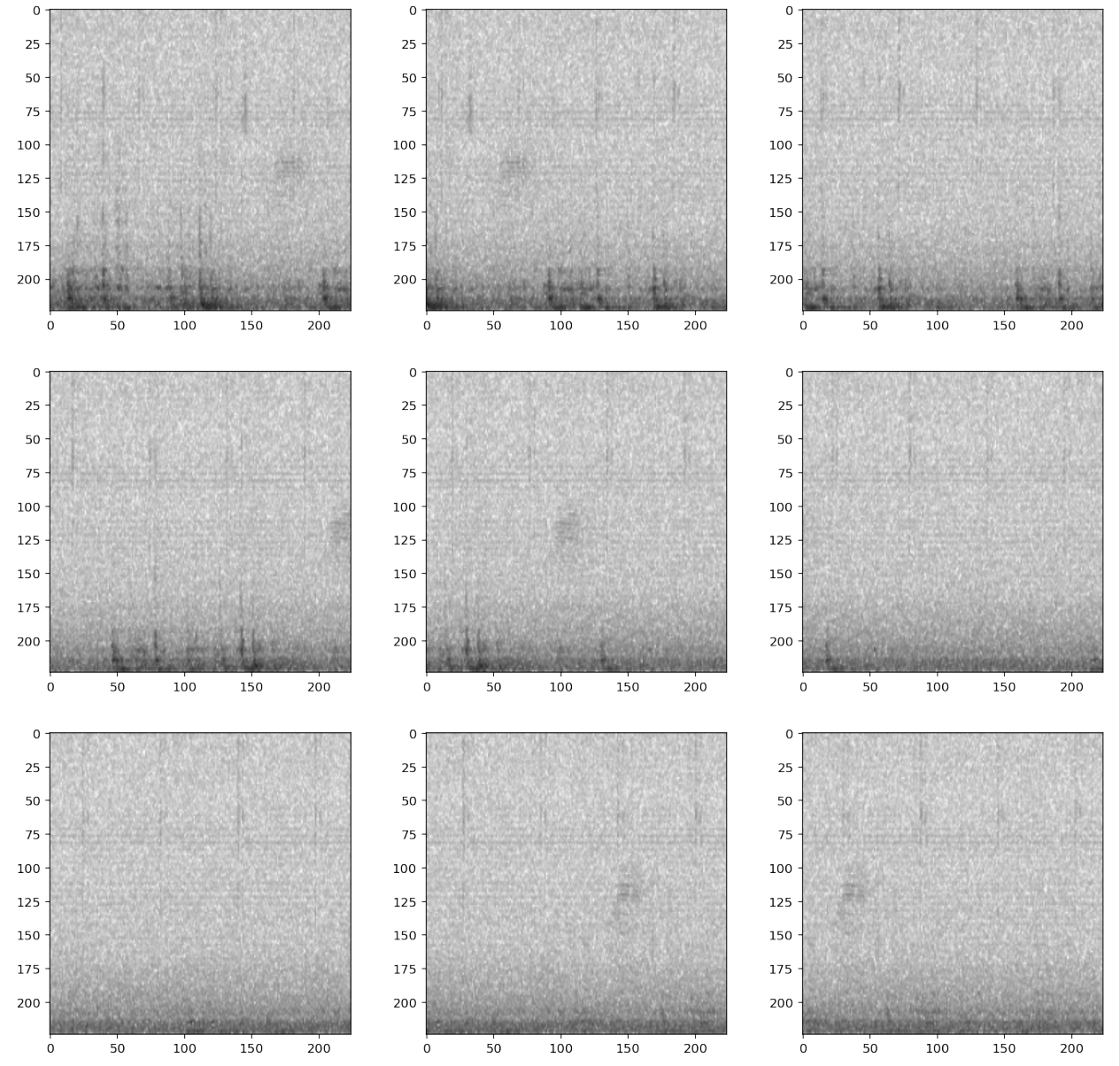
Here's an example of how to use `AudioSplittingDataset` to create several samples from a long audio file:

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

[15]:
```
prediction_df = pd.DataFrame(index=['./woodcock_labeled_data/field_data/60s_field_
↪data_sample_1.wav'])
```

[16]:
```
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)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```



9.5 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.

9.5.1 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.

[17]:
```
# inspect the current pipeline (ordered sequence of Actions to take)
preprocessor = SpectrogramPreprocessor(sample_duration=2)
preprocessor.pipeline
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

[17]:
```
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 0x7f9f6...
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 0x7f9...
random_affine        Augmentation Action calling <function torch_ra...
dtype: object
```

9.5.2 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

9.6 Modifying Actions

9.6.1 View default parameters for an Action

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

[18]:
```
#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
```

[18]:
```
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
dtype: object
```

9.6.2 Modify Action parameters

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

[19]:
```
preprocessor.pipeline.to_spec.set(dB_scale=False)
```

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

[20]:
```
preprocessor.pipeline.to_spec.params.window_samples = 512
preprocessor.pipeline.to_spec.params['overlap_fraction'] = 0.75

preprocessor.pipeline.to_spec.params
```

[20]:
```
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
dtype: object
```

9.6.3 Bypass actions

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

[21]:
```
preprocessor = SpectrogramPreprocessor(sample_duration=2.0)

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

(continues on next page)

(continued from previous page)

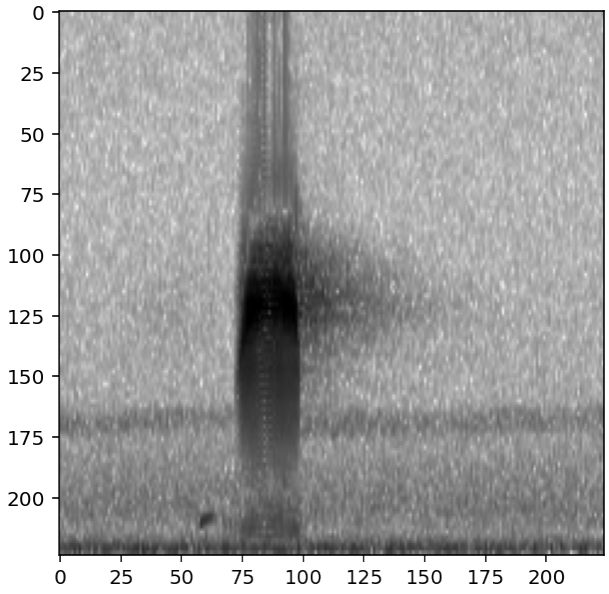
```
#printing the pipeline will show which actions are bypassed
preprocessor.pipeline
```

[21]:
```
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 0x7f9f6...
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 0x7f9...
random_affine         Augmentation Action calling <function torch_ra...
dtype: object
```

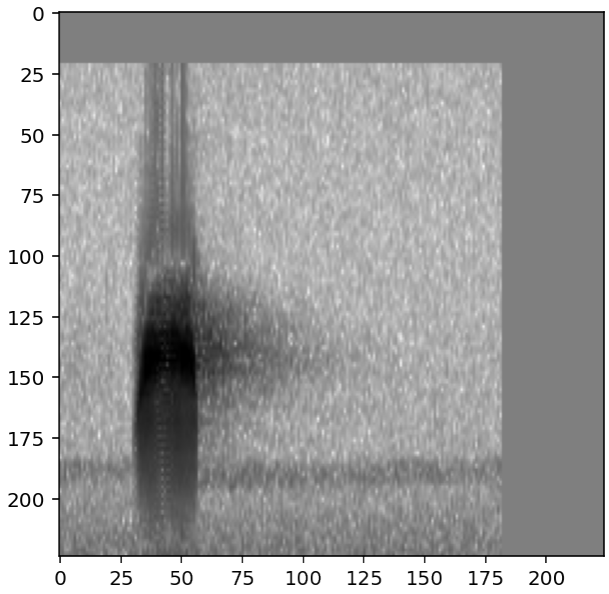
create a Dataset with this preprocessor and our label dataframe

[22]:
```
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
```




```
random affine on
```



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

[23]:
```
# The AudioLoader Action that is still on
preprocessor.pipeline.load_audio.bypass
```

[23]:
```
False
```

[24]:
```
# The frequency_mask Action that we turned off
preprocessor.pipeline.frequency_mask.bypass
```

[24]:
```
True
```

9.7 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

9.7.1 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:

[25]:
```
#initialize a preprocessor
preprocessor = SpectrogramPreprocessor(2.0)
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]
```

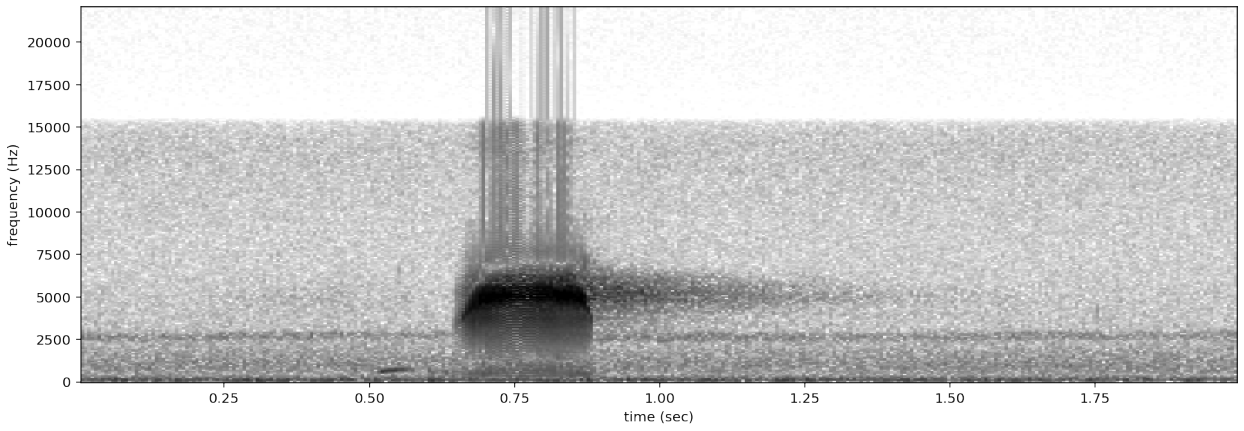
(continues on next page)

(continued from previous page)

```
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()
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

```
original pipeline:
Action calling <bound method Audio.from_file of <class 'opensoundscape.audio.Audio'>>
Augmentation Action calling <function trim_audio at 0x7f9f6462b8b0>
Action calling <function trim_audio at 0x7f9f6462b8b0>
Action calling <bound method Spectrogram.from_audio of <class 'opensoundscape.
↪spectrogram.Spectrogram'>>
Action calling <function Spectrogram.bandpass at 0x7f9f618fe430>
Action calling <function Spectrogram.to_image at 0x7f9f618fe700>
Augmentation Action calling <function time_mask at 0x7f9f6463f160>
Augmentation Action calling <function frequency_mask at 0x7f9f6463f1f0>
Augmentation Action calling <function tensor_add_noise at 0x7f9f6463f280>
Action calling <function scale_tensor at 0x7f9f6463f0d0>
Augmentation Action calling <function torch_random_affine at 0x7f9f6463ef70>

new pipeline:
Action calling <bound method Audio.from_file of <class 'opensoundscape.audio.Audio'>>
Augmentation Action calling <function trim_audio at 0x7f9f6462b8b0>
Action calling <function trim_audio at 0x7f9f6462b8b0>
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'>
```



9.8 Customizing preprocessing to achieve better machine learning outcomes

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.

9.8.1 Modify the sample rate

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

[26]:
```
pre = SpectrogramPreprocessor(sample_duration=2)

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

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

9.8.2 Modify spectrogram window length and overlap

(see Spectrogram.from_audio() for detailed documentation)

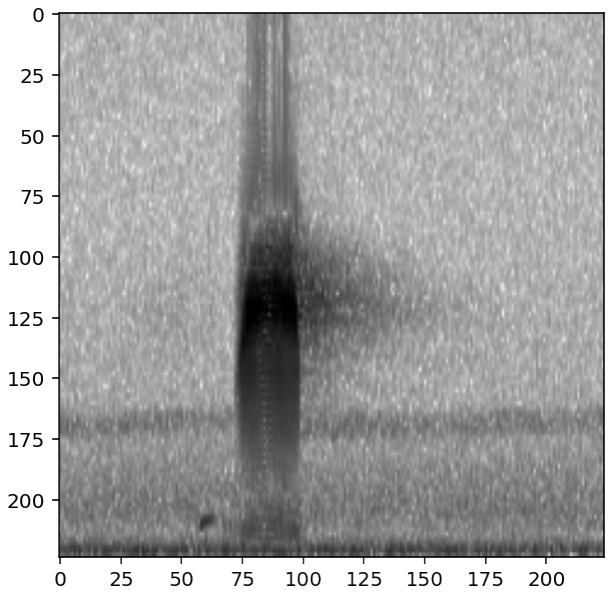
[27]:
```
dataset = AudioFileDataset(labels,SpectrogramPreprocessor(sample_duration=2))
dataset.bypass_augmentations=True

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

(continues on next page)

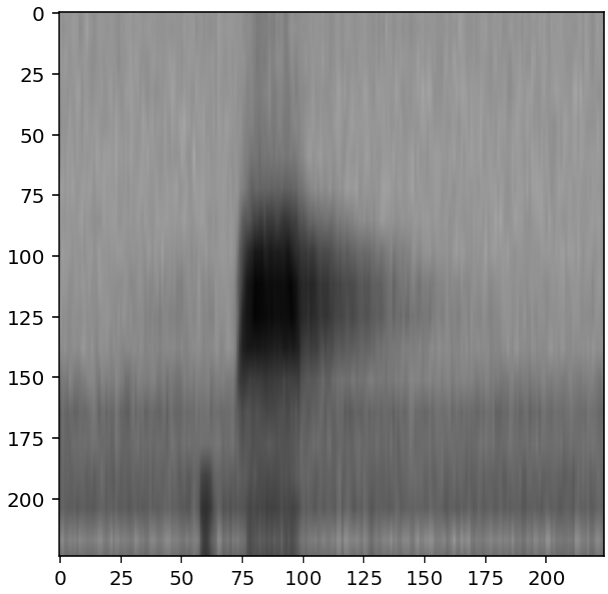
(continued from previous page)

```
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:
```




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



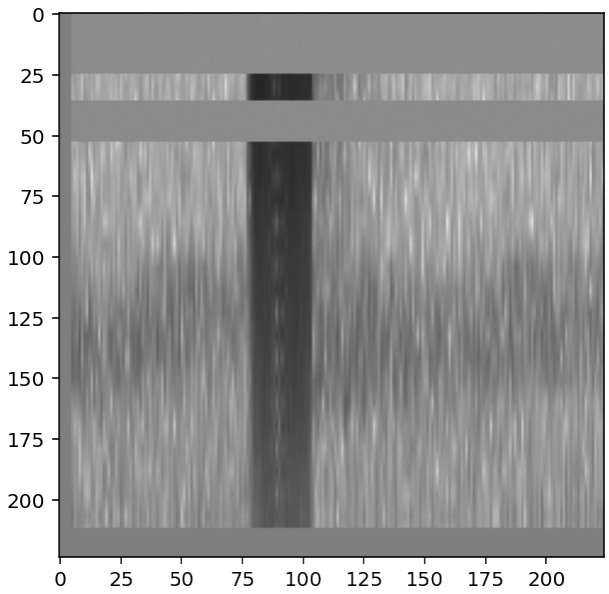
9.8.3 Bandpass spectrograms

Trim spectrograms to a specified frequency range:

[28]:
```
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)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```

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

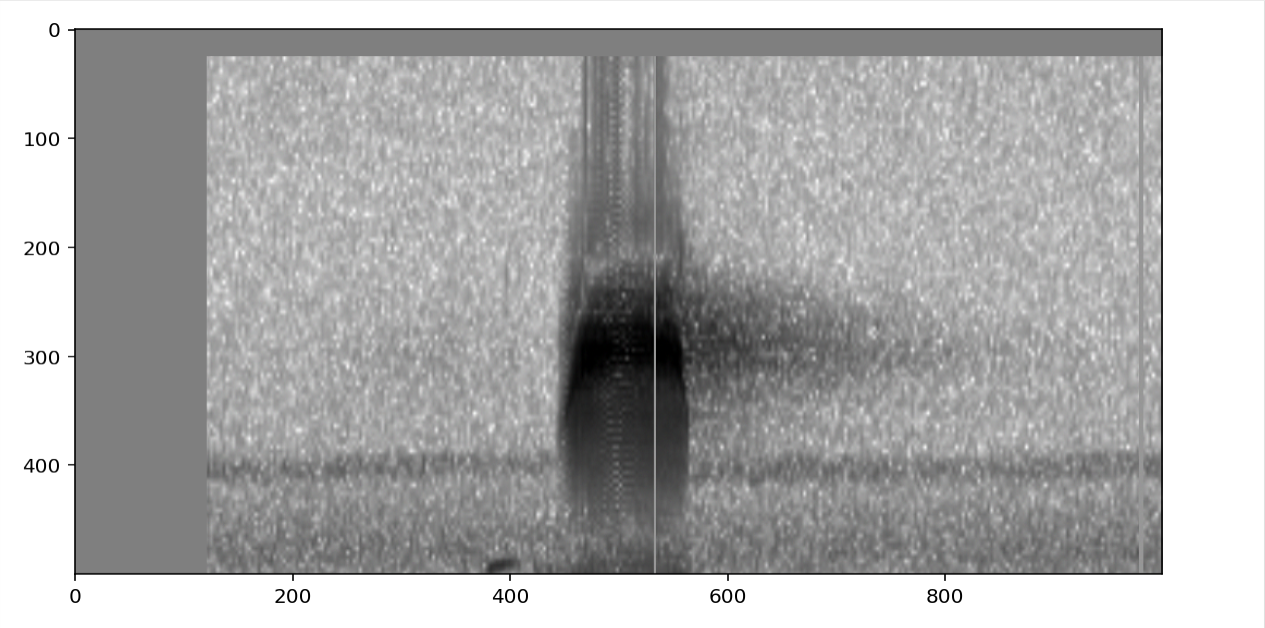


9.8.4 Change the output shape

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

[29]:
```
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)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or␣
↪[0..255] for integers).
```

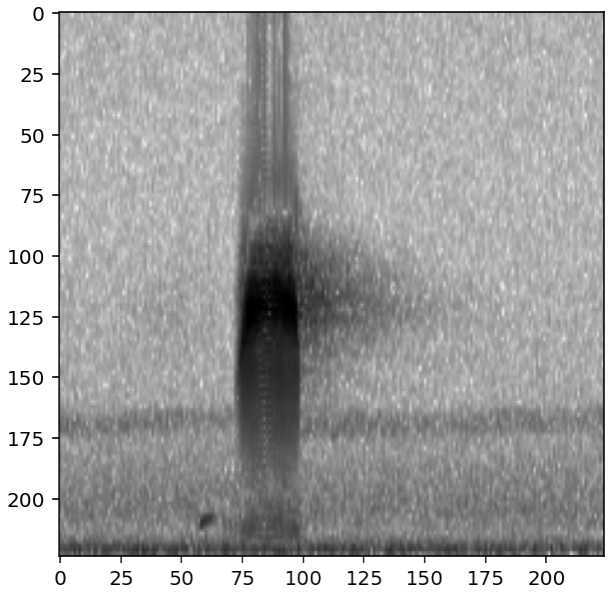


9.8.5 Turn all augmentation on or off

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.

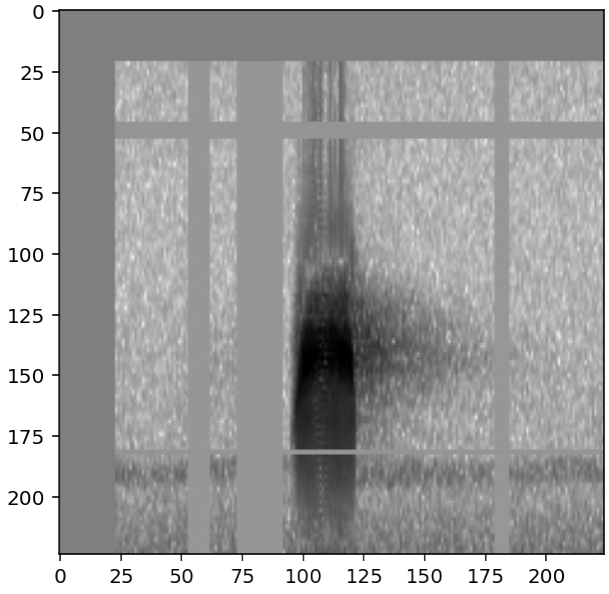
[30]:
```
dataset = AudioFileDataset(labels, SpectrogramPreprocessor(2.0))

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

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```



[31]:
```
dataset.bypass_augmentations = False
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```

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

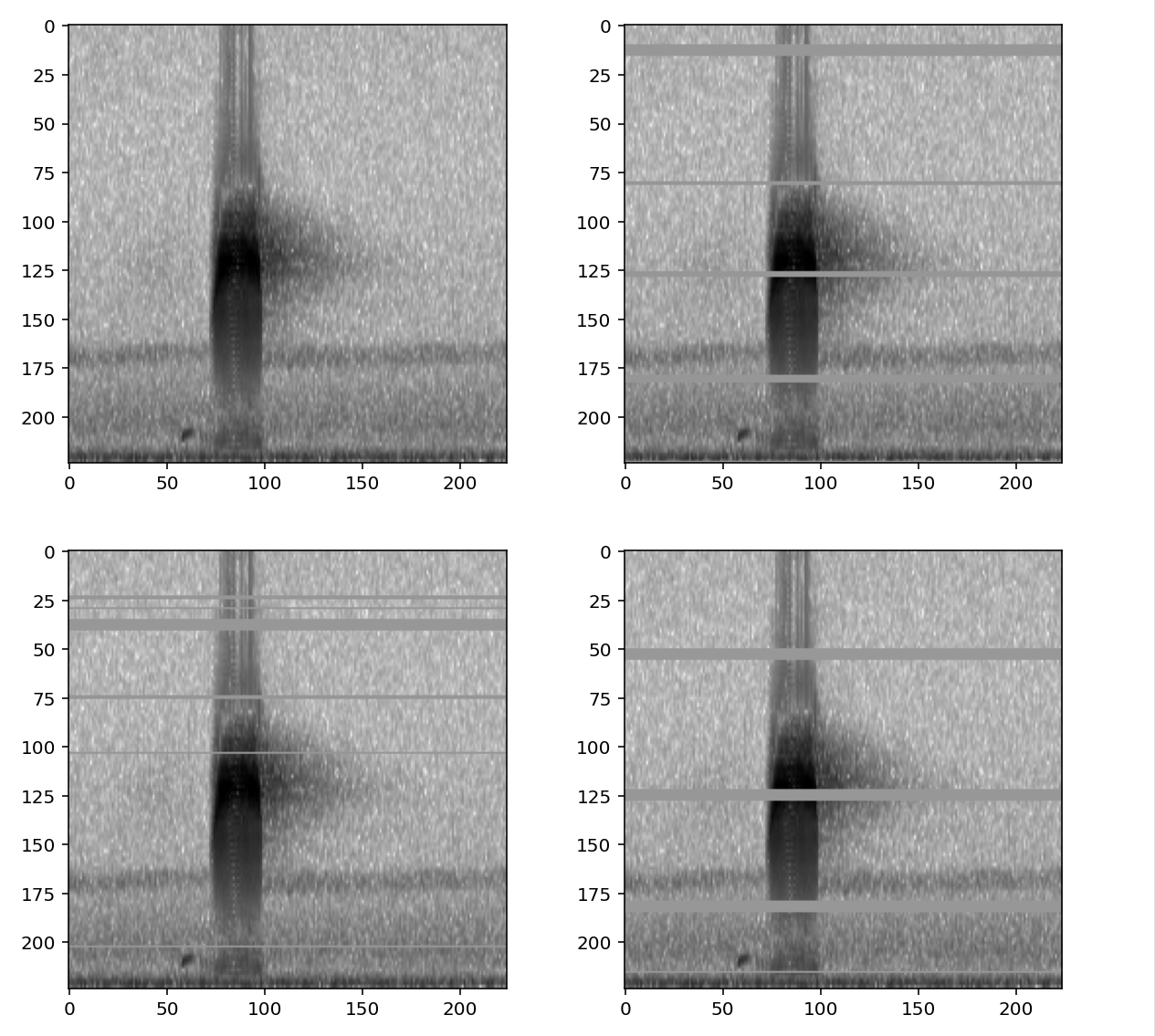


9.8.6 Modify augmentation parameters

`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.

[32]:
```
#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()
```

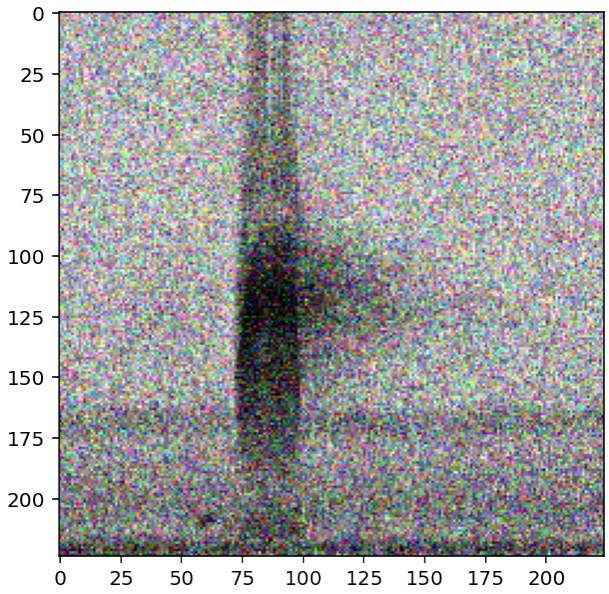
```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: DeprecationWarning: The default dtype for empty Series will be
↪'object' instead of 'float64' in a future version. Specify a dtype explicitly to␣
↪silence this warning.
  other = Series(other)
```



turn off frequency mask and turn on gaussian noise

[33]:
```
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)
```

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



9.8.7 remove an action by its name

[34]:
```
preprocessor.remove_action('add_noise')
preprocessor.pipeline
```

[34]:
```
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 0x7f9f6...
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 0x7f9...
random_affine         ## Bypassed ## Augmentation Action calling <fu...
dtype: object
```

9.8.8 add an action at a specific position

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

[35]:
```
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=...)
)
```

[36]:
```
preprocessor.pipeline
```

[36]:
```
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 0x7f9f6...
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 0x7f9...
random_affine         ## Bypassed ## Augmentation Action calling <fu...
dtype: object
```

it will complain if you use a non-unique index

[37]:
```
from opensoundscape.preprocess.actions import Action, tensor_add_noise
import pytest

with pytest.raises(AssertionError): #make sure it raises an error, and catch it
    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=...)
    )
```

9.8.9 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).

[38]:
```
#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)
```

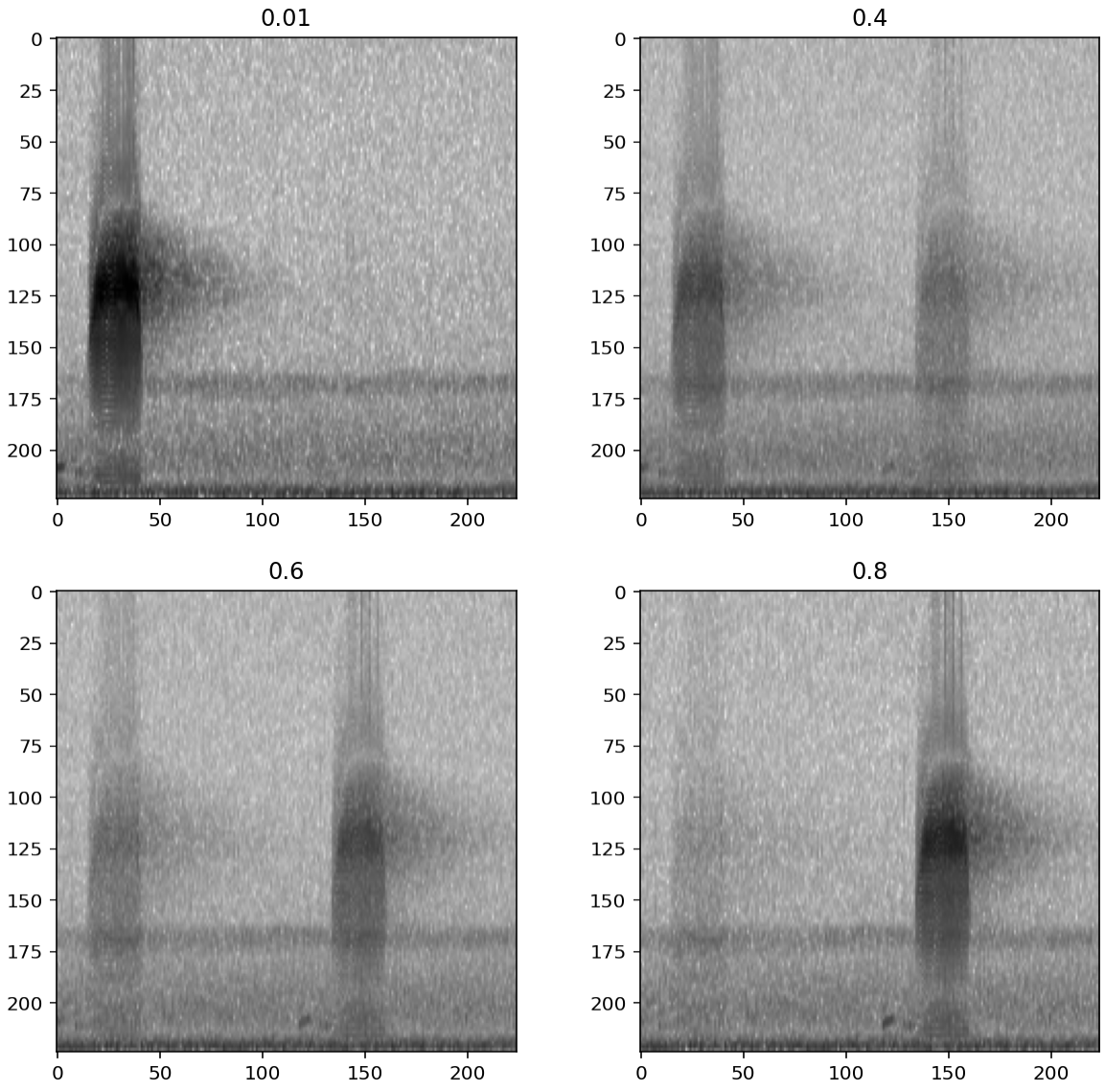
```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: DeprecationWarning: The default dtype for empty Series will be
↪'object' instead of 'float64' in a future version. Specify a dtype explicitly to␣
↪silence this warning.
  other = Series(other)
```

Modify `overlay_weight`

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.

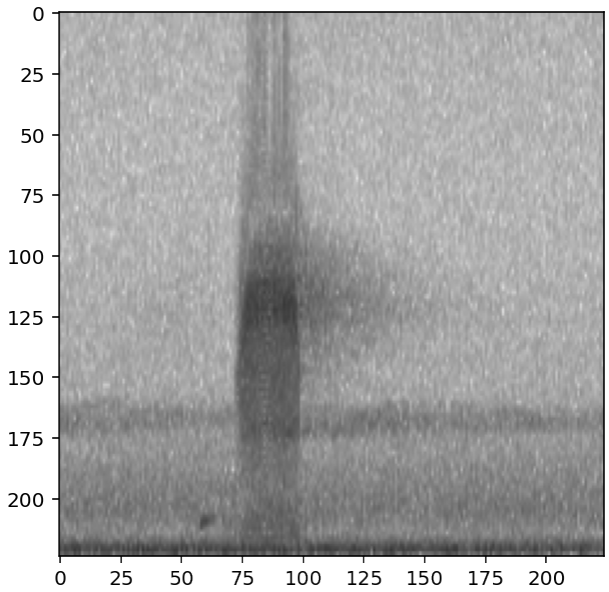
[39]:
```
preprocessor.pipeline.overlay.set(overlay_class='present')
tensors = []
overlay_weights = [0.01, 0.4, 0.6, 0.8]
for w in overlay_weights:
    preprocessor.pipeline.overlay.set(overlay_weight=w)
    dataset = AudioFileDataset(labels,preprocessor)
    np.random.seed(0) #get the same overlay every time
    tensors.append(dataset[2]['X'])
_ = show_tensor_grid(tensors, 2, labels=overlay_weights)
```



Overlay samples from a specific class

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

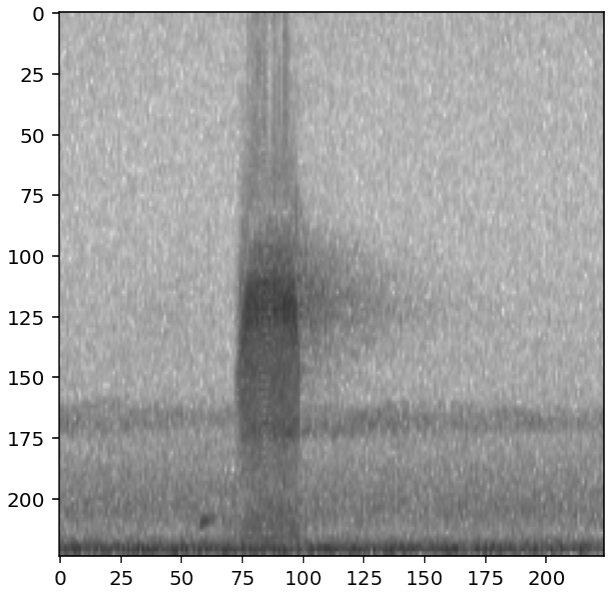
[40]:
```
dataset.preprocessor.pipeline.overlay.set(
    overlay_class='absent',
    overlay_weight=0.4
)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```



Overlaying samples from any class

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

[41]:
```
dataset.preprocessor.pipeline.overlay.set(overlay_class=None)
show_tensor(dataset[0]['X'],invert=True,transform_from_zero_centered=True)
```



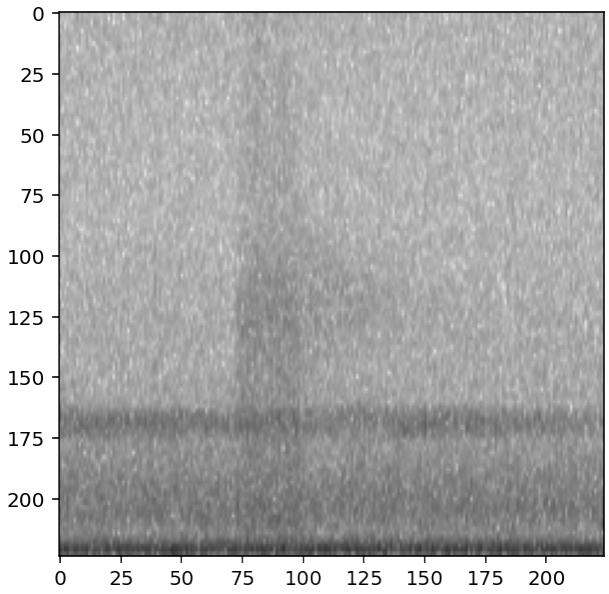
Overlaying samples from a "different" class

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%).

[42]:
```
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)
```



Updating labels

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`.

[43]:
```
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()}")
```

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

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'`.

9.9 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.

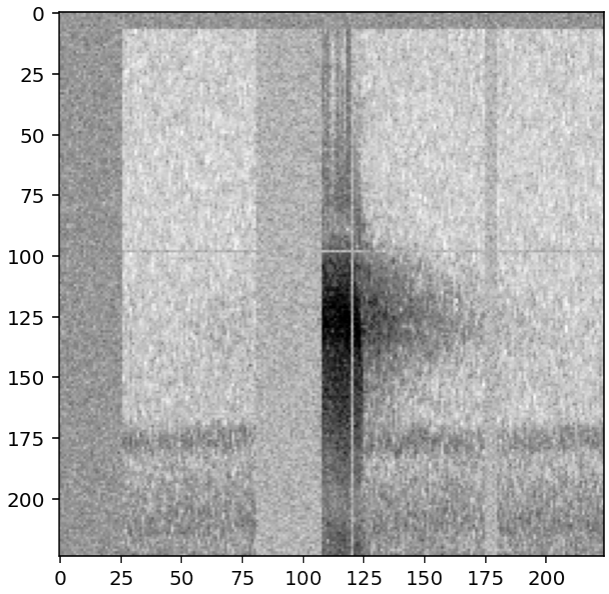
Your subclass might add a different set of Actions, define a different pipeline, or even override the `__getitem__` method of BasePreprocessor.

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

[44]:
```
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)
```

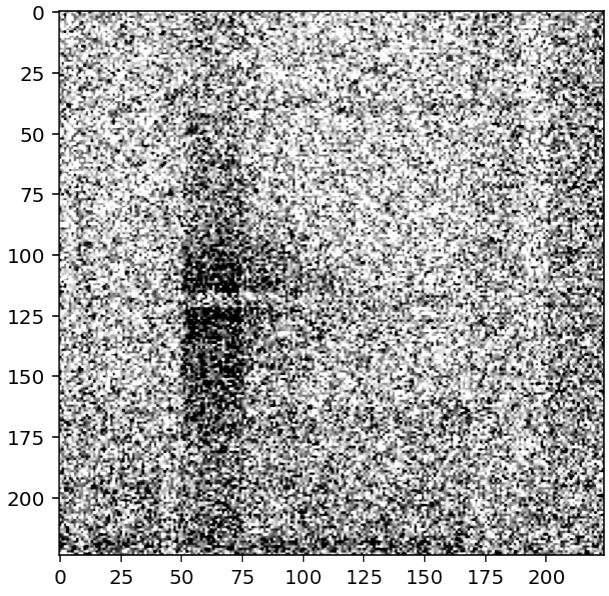
[45]:
```
dataset = AudioFileDataset(labels, MyPreprocessor(sample_duration=2.0, magic_
↪parameter=1))
show_tensor(dataset[0]['X'],invert=False)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```



[46]:
```
dataset = AudioFileDataset(labels, MyPreprocessor(sample_duration=2.0, magic_
↪parameter=4))
show_tensor(dataset[0]['X'],invert=False)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```



9.10 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.

9.10.1 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
```

[47]:
```
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)
```

Test it out:

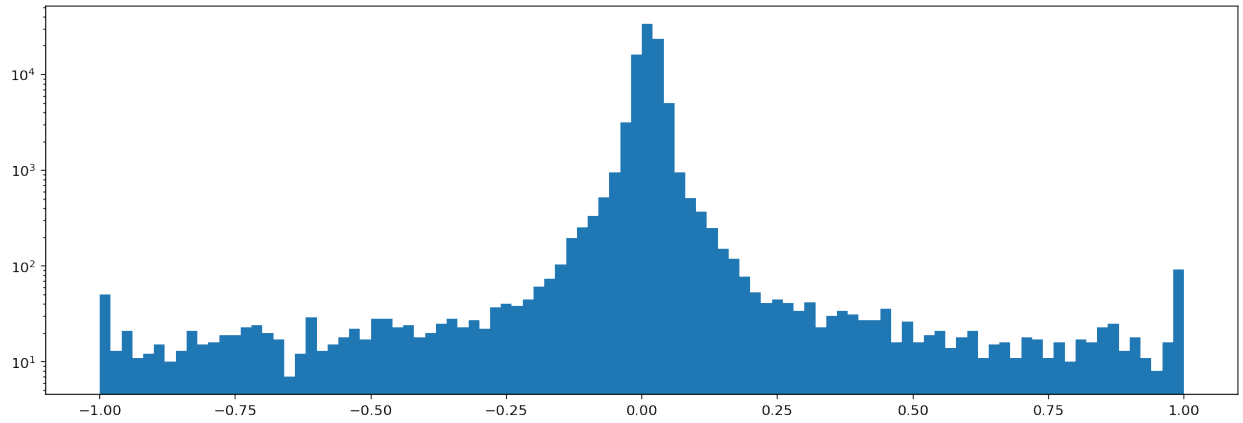
[48]:
```
from opensoundscape.audio import Audio

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()
```

(continues on next page)

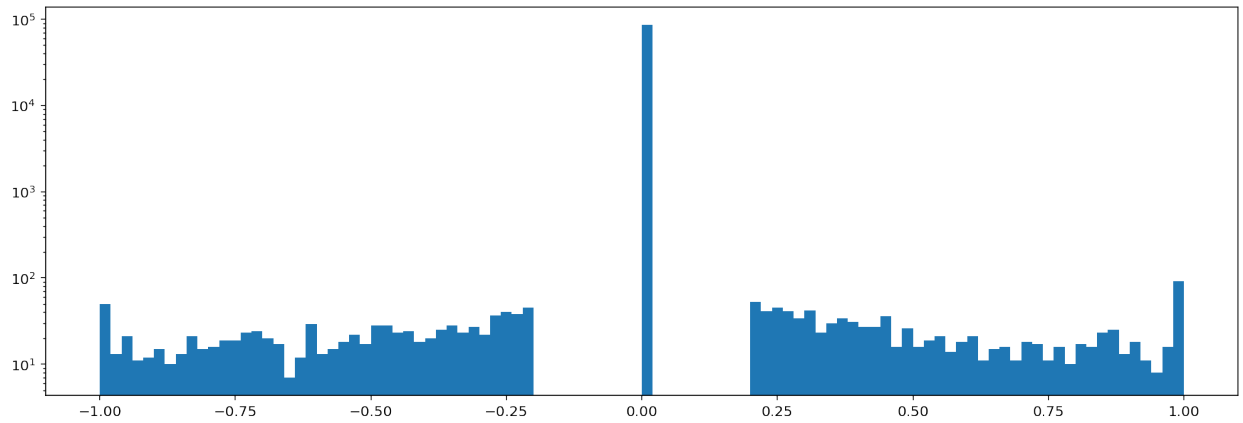
(continued from previous page)

```
plt.show()

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()
```

```
histogram of samples
```

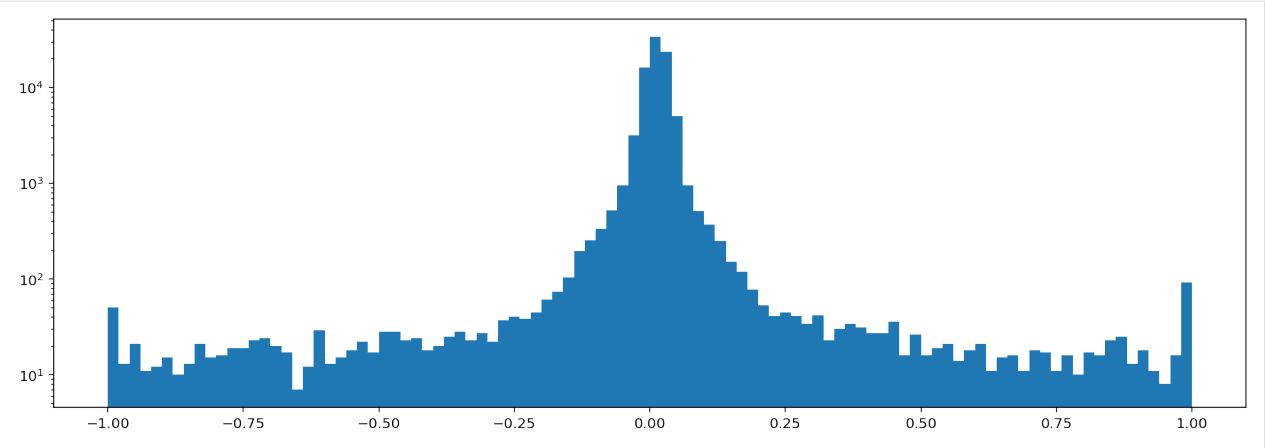



```
histogram of samples after audio gate
```




```
histogram of samples after audio gate, when labels[0]==0
```

[48]:
```
[]
```



Clean up files created during this tutorial:

[49]:
```
import shutil
shutil.rmtree('./woodcock_labeled_data')
```

CHAPTER 10

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'
```

10.1 Prepare audio data

10.1.1 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      5      0 --:--:--  0:00:01 --:--:--     9
100 9499k  100 9499k    0     0  4521k      0  0:00:02  0:00:02 --:--:-- 20.1M
```

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

10.1.2 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')
# 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]:

file	present	absent
./woodcock_labeled_data/d4c40b6066b489518f8da83...	1	0
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...	0	1
./woodcock_labeled_data/79678c979ebb880d5ed6d56...	1	0
./woodcock_labeled_data/49890077267b569e142440f...	1	0
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...	1	0

10.1.3 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)
```

10.2 Creating a model

We initialize a model object by specifying the architecture, a list of classes, and the duration of individual samples in seconds

[5]:
```
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)
```

10.2.1 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,single_target=True)
```

10.3 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}
```

10.3.1 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
```

10.3.2 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}}
```

10.3.3 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
```

10.3.4 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
```

10.4 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 (`use_pretrained=False` for random weights). For instance, lets create an alexnet architecture with random weights:

[14]:
```
my_arch = cnn_architectures.alexnet(num_classes=len(classes),use_pretrained=False)
```

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']
```

[16]:
```
model = cnn.CNN(architecture='resnet18',classes=classes, sample_duration=2.0)
```

10.4.1 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 `use_pretrained` argument to `False`, when creating an architecture:

[17]:
```
arch = cnn_architectures.alexnet(num_classes=10,use_pretrained=False)
```

10.4.2 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, use_
↪pretrained=False)
```

10.4.3 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,use_pretrained=False,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
preds, _, _ = model.predict(valid_df)
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/torchvision/
↪models/inception.py:81: 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('The default weight initialization of inception_v3 will be changed in␣
↪future releases of '


Training Epoch 0
Epoch: 0 [batch 0/6 (0.00%)]
        DistLoss: 0.888
Metrics:
Metrics:
        MAP: 0.511
```

(continues on next page)

(continued from previous page)

```
Validation.
[]
Metrics:
        MAP: 0.500

Best Model Appears at Epoch 0 with Validation score 0.500.
[]
```

10.4.4 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, use_pretrained=False)

# replace the alexnet architecture with the densenet architecture
model.network = new_arch
```

10.5 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'>
```

```
/Users/SML161/opt/miniconda3/envs/opso_dev/lib/python3.8/site-packages/pandas/core/
↪series.py:3247: 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.
  other = Series(other)
```

10.6 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.

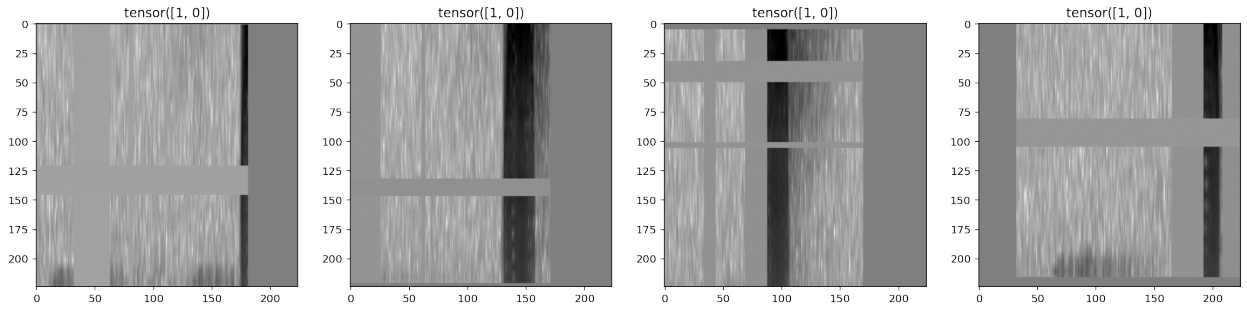
It's good practice to create the validation from the training dataset (after any modifications are made), so that they perform the same preprocessing. You may or may not want to use augmentation on the validation dataset.

10.6.1 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)
```

```
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).
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.

10.6.2 clean up

remove files

[24]:
```
import shutil
shutil.rmtree('./woodcock_labeled_data')

for p in Path('.').glob('*.model'):
    p.unlink()
```

CHAPTER 11

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

11.1 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)
```

11.2 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      6      0 --:--:--  0:00:01 --:--:--     0
100 11.6M  100 11.6M    0     0  5308k      0  0:00:02  0:00:02 --:--:-- 18.3M
```

[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

11.2.1 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)

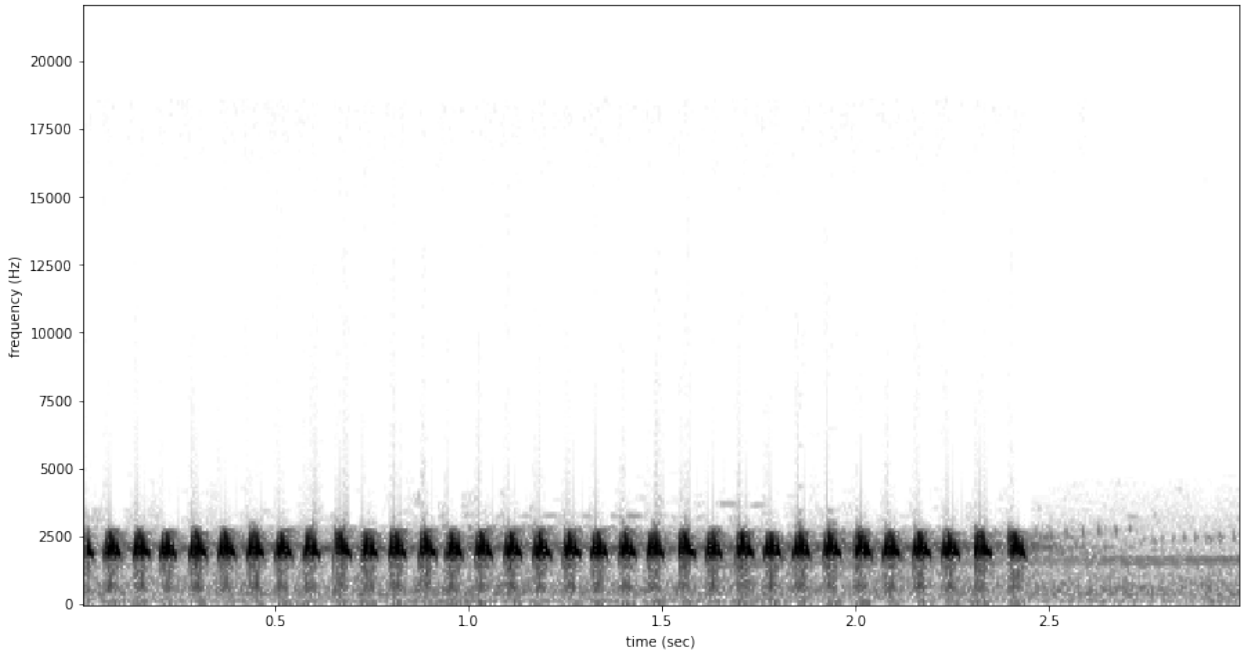
(continued from previous page)

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

11.2.2 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()
```



11.3 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]:
```
# 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
```

11.4 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.

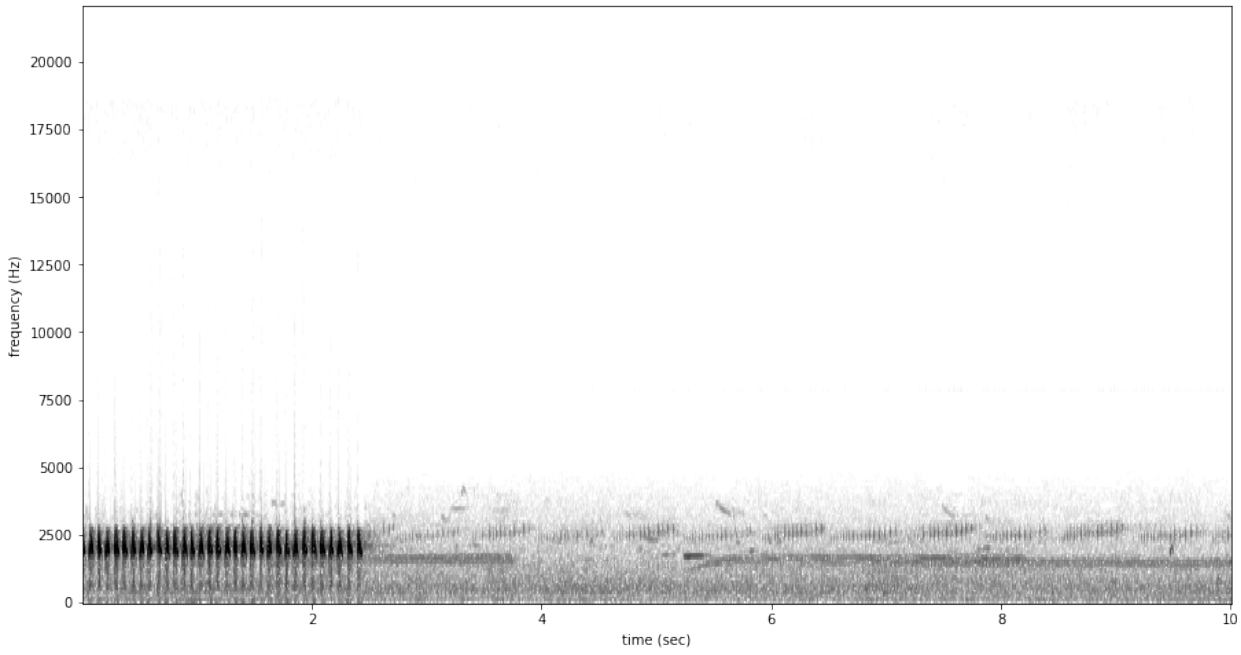
[6]:
```
#get the audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

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

(continues on next page)

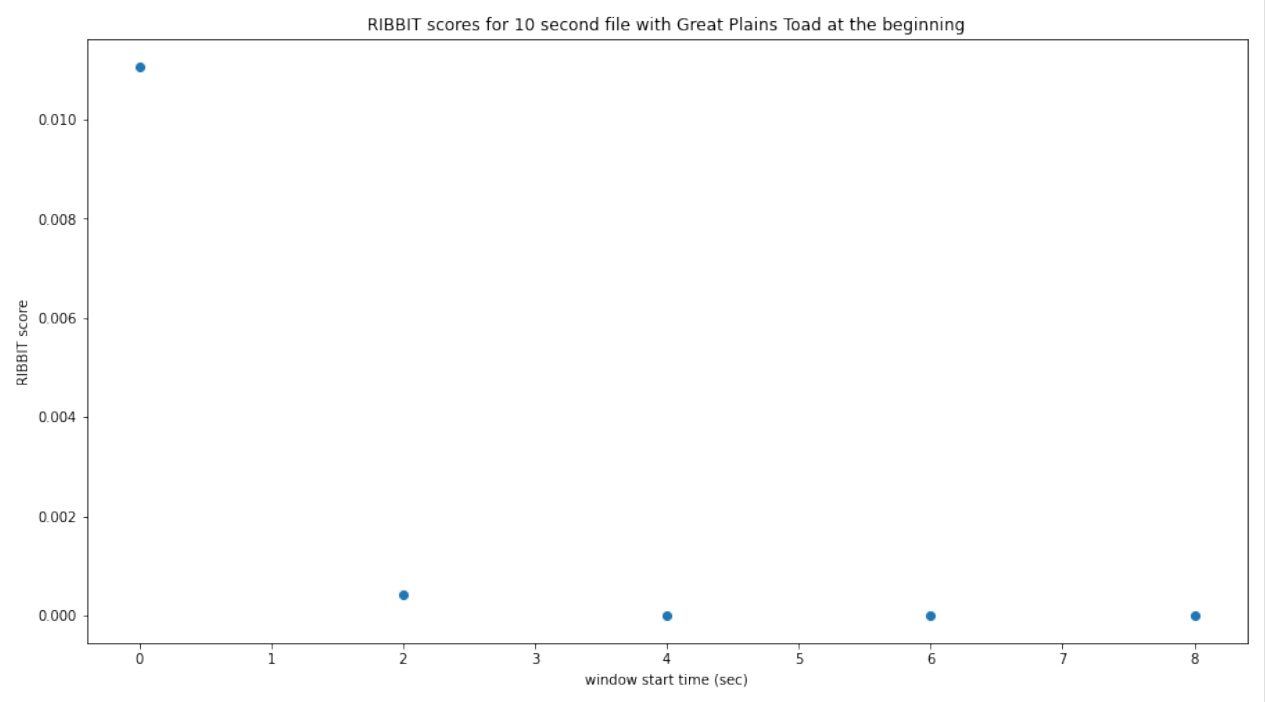
(continued from previous page)

```
#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

11.5 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)

11.6 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)
```

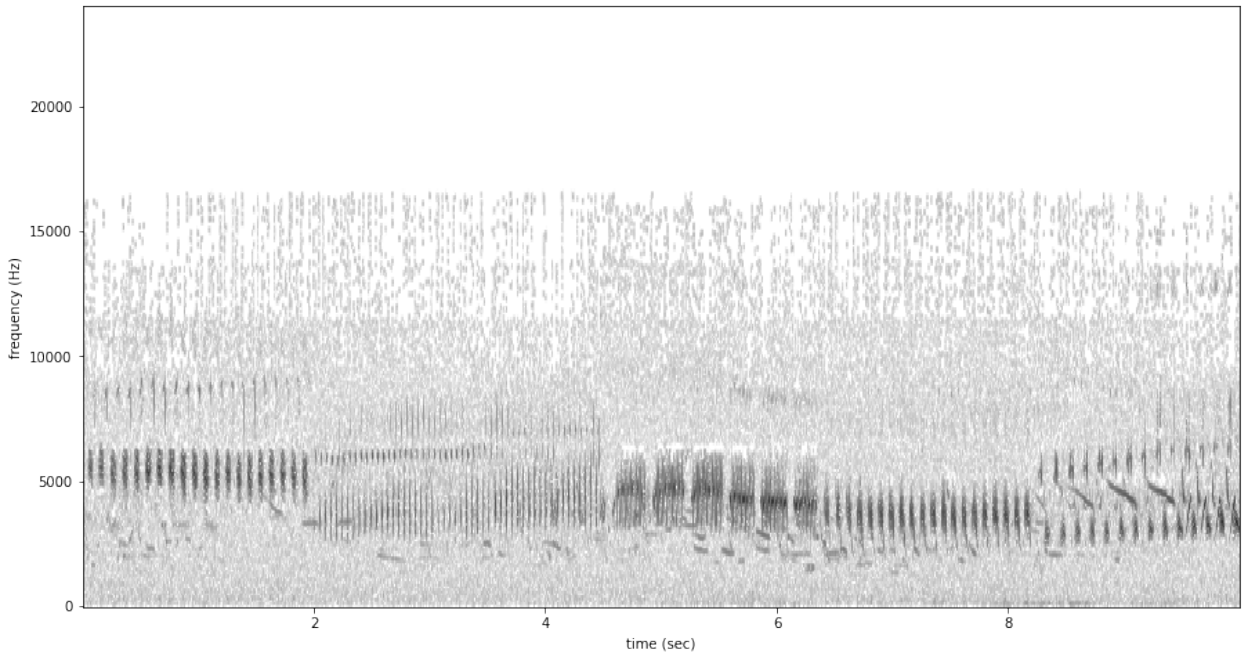
```
Files with scores for each species, sorted by 'bird_series' score:
```

[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()
```



11.6.1 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]]
```

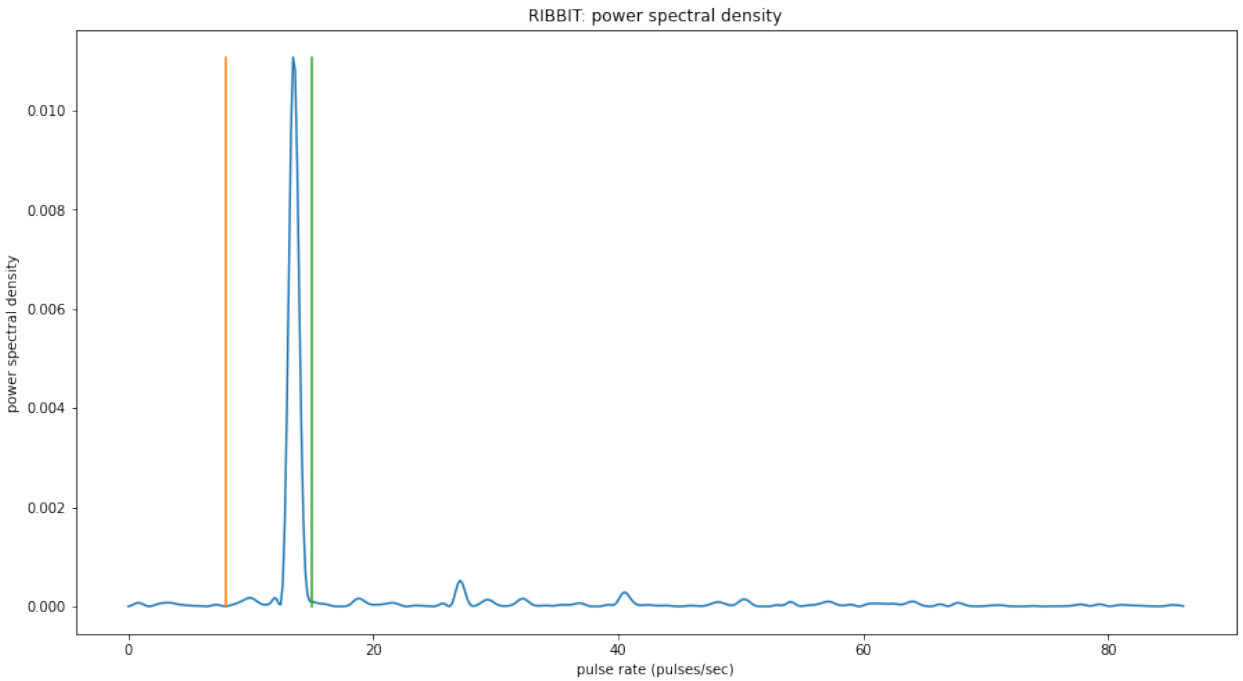
11.7 Detail view of RIBBIT method

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.

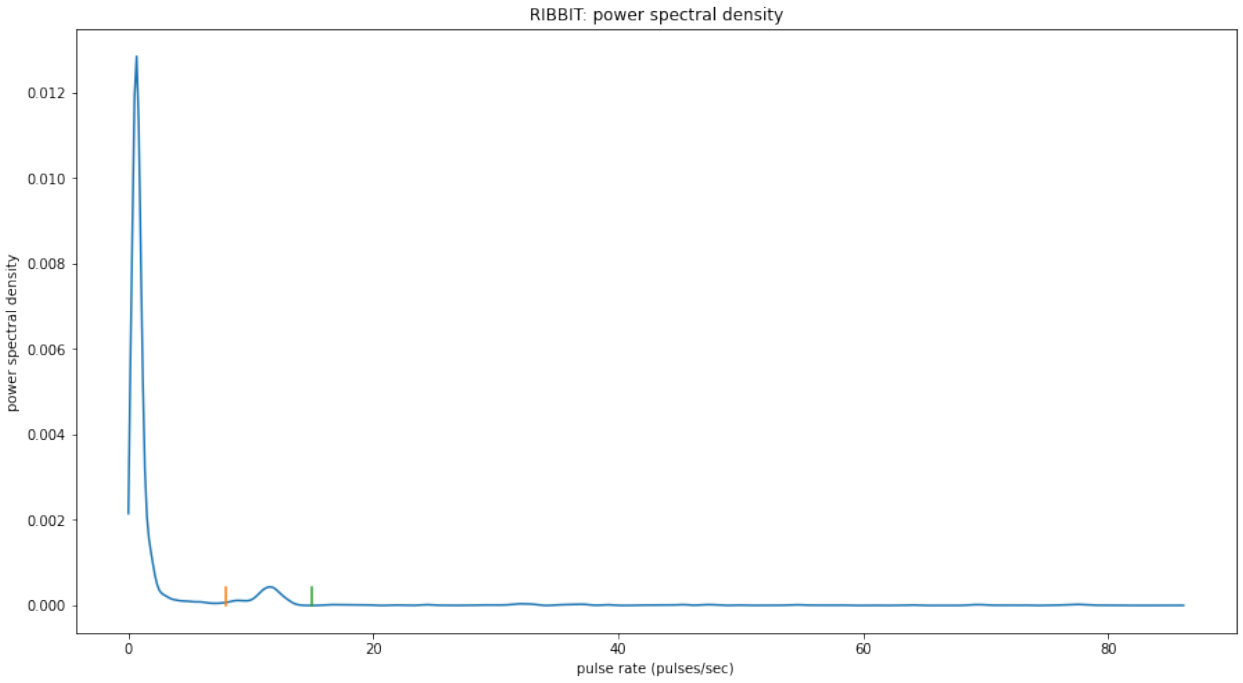
In the file `gpt0.wav`, the Great Plains Toad vocalizes for a couple seconds at the beginning, then stops. We expect to see a peak in the power spectral density at 15 pulses/sec in the first 2 second window, and maybe a bit in the second, but not later in the audio.

[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)
```

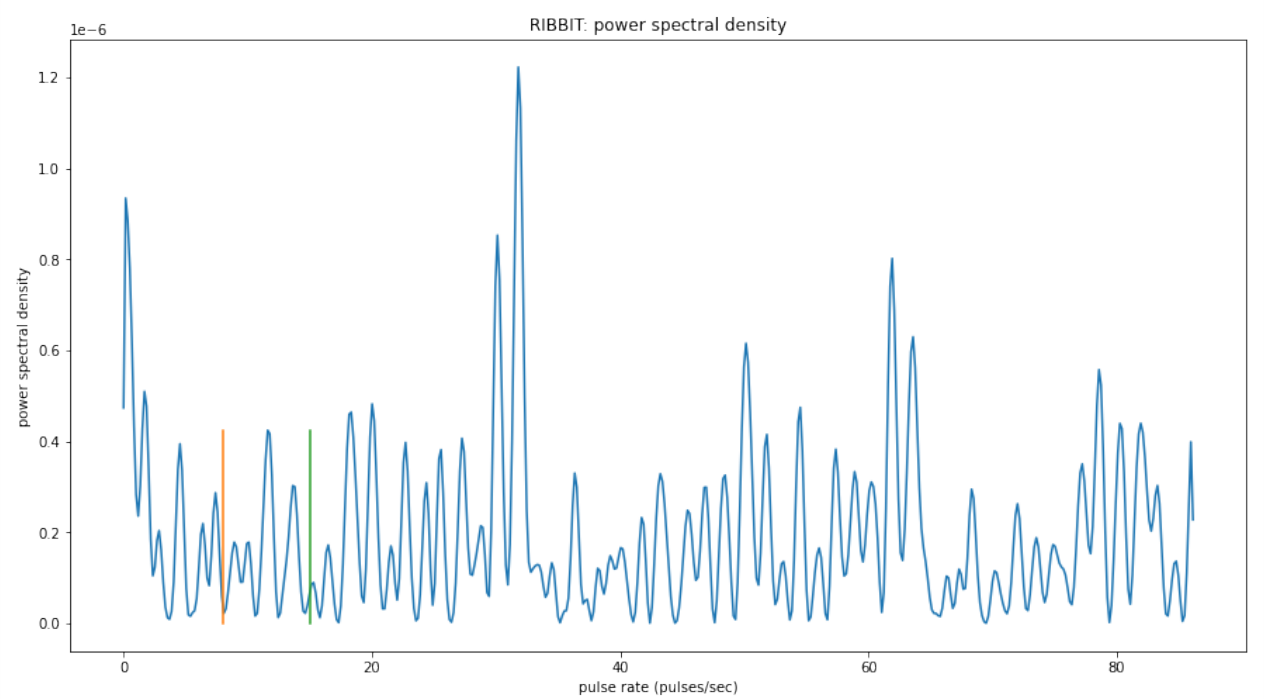
```
window: 0.0 to 2.0 sec
```



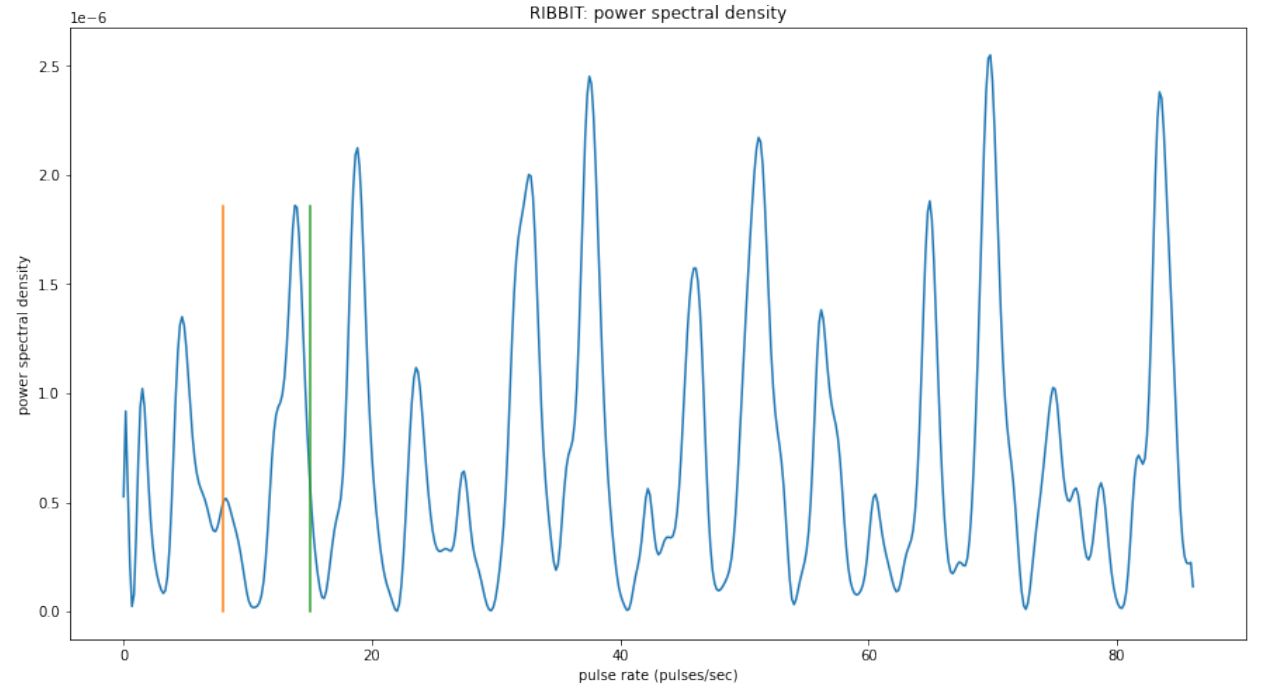

```
window: 2.0 to 4.0 sec
```




```
window: 4.0 to 6.0 sec
```




```
window: 6.0 to 8.0 sec
```



11.8 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 12

Audio and Spectrograms

12.1 Annotations

functions and classes for manipulating annotations of audio

includes BoxedAnnotations class and utilities to combine or "diff" annotations, etc.

class opensoundscape.annotations.**BoxedAnnotations**(*df*, *audio_file=None*)
container for "boxed" (frequency-time) annotations of audio

(for instance, annotations created in Raven software) includes functionality to load annotations from Raven txt files, output one-hot labels for specific clip lengths or clip start/end times, apply corrections/conversions to annotations, and more.

Contains some analogous functions to Audio and Spectrogram, such as trim() [limit time range] and bandpass() [limit frequency range]

bandpass(*low_f*, *high_f*, *edge_mode='trim'*)
Bandpass a set of annotations, analogous to Spectrogram.bandpass()

Out-of-place operation: does not modify itself, returns new object

Parameters

- **low_f** – low frequency (Hz) bound
- **high_f** – high frequench (Hz) bound
- **edge_mode** – what to do when boxes overlap with edges of trim region - 'trim': trim boxes to bounds - 'keep': allow boxes to extend beyond bounds - 'remove': completely remove boxes that extend beyond bounds

Returns a copy of the BoxedAnnotations object on the bandpassed region

convert_labels(*conversion_table*)
modify annotations according to a conversion table

Changes the values of 'annotation' column of dataframe. Any labels that do not have specified conversions are left unchanged.

Returns a new BoxedAnnotations object, does not modify itself (out-of-place operation). So use could look like: *my_annotations = my_annotations.convert_labels(table)*

Parameters **conversion_table** – current values -> new values. can be either - pd.DataFrame with 2 columns [current value, new values] or - dictionary {current values: new values}

Returns new BoxedAnnotations object with converted annotation labels

classmethod from_raven_file(*path*, *annotation_column*, *keep_extra_columns=True*, *audio_file=None*)

load annotations from Raven txt file

Parameters

- **path** – location of raven .txt file, str or pathlib.Path
- **annotation_column** – (str) column containing annotations
- **keep_extra_columns** – keep or discard extra Raven file columns (always keeps start_time, end_time, low_f, high_f, annotation audio_file). [default: True] - True: keep all - False: keep none - or iterable of specific columns to keep
- **audio_file** – optionally specify the name or path of a corresponding audio file.

Returns BoxedAnnotations object containing annotaitons from the Raven file

global_one_hot_labels(*classes*)

get a dictionary of one-hot labels for entire duration :param classes: iterable of class names to give 0/1 labels

Returns list of 0/1 labels for each class

one_hot_clip_labels(*full_duration*, *clip_duration*, *clip_overlap*, *classes*, *min_label_overlap*, *min_label_fraction=1*, *final_clip=None*)

Generate one-hot labels for clips of fixed duration

wraps helpers.generate_clip_times_df() with self.one_hot_labels_like() - Clips are created in the same way as Audio.split() - Labels are applied based on overlap, using self.one_hot_labels_like()

Parameters

- **full_duration** – The amount of time (seconds) to split into clips
- **clip_duration** (*float*) – The duration in seconds of the clips
- **clip_overlap** (*float*) – The overlap of the clips in seconds [default: 0]
- **classes** – list of classes for one-hot labels. If None, classes will be all unique values of self.df['annotation']
- **min_label_overlap** – minimum duration (seconds) of annotation within the time interval for it to count as a label. Note that any annotation of length less than this value will be discarded. We recommend a value of 0.25 for typical bird songs, or shorter values for very short-duration events such as chip calls or nocturnal flight calls.
- **min_label_fraction** – [default: None] if >= this fraction of an annotation overlaps with the time window, it counts as a label regardless of its duration. Note that *if either* of the two criterea (overlap and fraction) is met, the label is 1. if None (default), this criterion is not used (i.e., only min_label_overlap is used). A value of 0.5 for ths parameter would ensure that all annotations result in at least one clip being labeled 1 (if there are no gaps between clips).

- **final_clip** (*str*) – Behavior if final_clip is less than clip_duration seconds long. By default, discards remaining time if less than clip_duration seconds long [default: None]. Options:
  - None: Discard the remainder (do not make a clip)
  - "extend": Extend the final clip beyond full_duration to reach clip_duration length
  - "remainder": Use only remainder of full_duration (final clip will be shorter than clip_duration)
  - "full": Increase overlap with previous clip to yield a clip with clip_duration length

Returns dataframe with index ['start_time','end_time'] and columns=classes

one_hot_labels_like(*clip_df*, *classes*, *min_label_overlap*, *min_label_fraction=None*, *keep_index=False*)

create a dataframe of one-hot clip labels based on given starts/ends

Uses start and end clip times from clip_df to define a set of clips. Then extracts annotatations associated overlapping with each clip. Required overlap parameters are selected by user: annotation must satisfy the minimum time overlap OR minimum % overlap to be included (doesn't require both conditions to be met, only one)

clip_df can be created using opensoundscap.helpers.generate_clip_times_df

Parameters

- **clip_df** – dataframe with 'start_time' and 'end_time' columns specifying the temporal bounds of each clip
- **min_label_overlap** – minimum duration (seconds) of annotation within the time interval for it to count as a label. Note that any annotation of length less than this value will be discarded. We recommend a value of 0.25 for typical bird songs, or shorter values for very short-duration events such as chip calls or nocturnal flight calls.
- **min_label_fraction** – [default: None] if >= this fraction of an annotation overlaps with the time window, it counts as a label regardless of its duration. Note that *if either* of the two criterea (overlap and fraction) is met, the label is 1. if None (default), this criterion is not used (i.e., only min_label_overlap is used). A value of 0.5 for ths parameter would ensure that all annotations result in at least one clip being labeled 1 (if there are no gaps between clips).
- **classes** – list of classes for one-hot labels. If None, classes will be all unique values of self.df['annotation']
- **keep_index** – if True, keeps the index of clip_df as an index in the returned DataFrame. [default:False]

Returns DataFrame of one-hot labels (multi-index of (start_time, end_time), columns for each class, values 0=absent or 1=present)

subset(*classes*)

subset annotations to those from a list of classes

out-of-place operation (returns new filtered BoxedAnnotations object)

Parameters

- **classes** – list of classes to retain (all others are discarded)
- **the list can include np.nan or None if you want to keep them** (–) –

Returns new BoxedAnnotations object containing only annotations in *classes*

to_raven_file(*path*)
save annotations to a Raven-compatible tab-separated text file

Parameters **path** – path for saved test file (extension must be ".tsv") - can be str or pathlib.Path

Outcomes: creates a file containing the annotations in a format compatible with Raven Pro/Lite.

Note: Raven Lite does not support additional columns beyond a single annotation column. Additional columns will not be shown in the Raven Lite interface.

trim(*start_time*, *end_time*, *edge_mode='trim'*)
Trim a set of annotations, analogous to Audio/Spectrogram.trim()

Out-of-place operation: does not modify itself, returns new object

Parameters

- **start_time** – time (seconds) since beginning for left bound
- **end_time** – time (seconds) since beginning for right bound
- **edge_mode** – what to do when boxes overlap with edges of trim region - 'trim': trim boxes to bounds - 'keep': allow boxes to extend beyond bounds - 'remove': completely remove boxes that extend beyond bounds

Returns a copy of the BoxedAnnotations object on the trimmed region. - note that, like Audio.trim(), there is a new reference point for 0.0 seconds (located at start_time in the original object)

unique_labels()
get list of all unique (non-Falsy) labels

opensoundscape.annotations.**categorical_to_one_hot**(*labels*, *classes=None*)
transform multi-target categorical labels (list of lists) to one-hot array

Parameters

- **labels** – list of lists of categorical labels, eg [['white','red'],['green','white']] or [[0,1,2],[3]]
- **classes=None** – list of classes for one-hot labels. if None, taken to be the unique set of values in *labels*

Returns 2d array with 0 for absent and 1 for present classes: list of classes corresponding to columns in the array

Return type one_hot

opensoundscape.annotations.**combine**(*list_of_annotation_objects*)
combine annotations with user-specified preferences Not Implemented.

opensoundscape.annotations.**diff**(*base_annotations*, *comparison_annotations*)
look at differences between two BoxedAnnotations objects Not Implemented.

Compare different labels of the same boxes (Assumes that a second annotator used the same boxes as the first, but applied new labels to the boxes)

opensoundscape.annotations.**one_hot_labels_on_time_interval**(*df*, *classes*, *start_time*, *end_time*, *min_label_overlap*, *min_label_fraction=None*)
generate a dictionary of one-hot labels for given time-interval

Each class is labeled 1 if any annotation overlaps sufficiently with the time interval. Otherwise the class is labeled 0.

Parameters

- **df** – DataFrame with columns 'start_time', 'end_time' and 'annotation'
- **classes** – list of classes for one-hot labels. If None, classes will be all unique values of self.df['annotation']
- **start_time** – beginning of time interval (seconds)
- **end_time** – end of time interval (seconds)
- **min_label_overlap** – minimum duration (seconds) of annotation within the time interval for it to count as a label. Note that any annotation of length less than this value will be discarded. We recommend a value of 0.25 for typical bird songs, or shorter values for very short-duration events such as chip calls or nocturnal flight calls.
- **min_label_fraction** – [default: None] if >= this fraction of an annotation overlaps with the time window, it counts as a label regardless of its duration. Note that *if either* of the two criterea (overlap and fraction) is met, the label is 1. if None (default), the criterion is not used (only min_label_overlap is used). A value of 0.5 would ensure that all annotations result in at least one clip being labeled 1 (if no gaps between clips).

Returns label 0/1} for all classes

Return type dictionary of {class

opensoundscape.annotations.**one_hot_to_categorical**(*one_hot*, *classes*)

transform one_hot labels to multi-target categorical (list of lists)

Parameters

- **one_hot** – 2d array with 0 for absent and 1 for present
- **classes** – list of classes corresponding to columns in the array

Returns

list of lists of categorical labels for each sample, eg [['white','red'],['green','white']] or [[0,1,2],[3]]

Return type labels

12.2 Audio

audio.py: Utilities for loading and modifying Audio objects

Note: Out-of-place operations

Functions that modify Audio (and Spectrogram) objects are "out of place", meaning that they return a new Audio object instead of modifying the original object. 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)` `

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

Container for audio samples

Initialization requires sample array. To load audio file, use *Audio.from_file()*

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

Parameters

- **samples** (*np.array*) – The audio samples
- **sample_rate** (*integer*) – The sampling rate for the audio samples
- **resample_type** (*str*) – The resampling method to use [default: "kaiser_fast"]

Returns An initialized *Audio* object

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

Bandpass audio signal with a butterworth filter

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

Parameters

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

duration()

Return duration of Audio

Returns The duration of the Audio

Return type duration (float)

extend(*length*)

Extend audio file by adding silence to the end

Parameters **length** – the final duration in seconds of the extended audio object

Returns a new Audio object of the desired duration

classmethod from_bytesio(*bytesio*, *sample_rate=None*, *resample_type='kaiser_fast'*)

Read from bytesio object

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

Parameters

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

Returns An initialized Audio object

classmethod from_file(*path*, *sample_rate=None*, *resample_type='kaiser_fast'*, *metadata=True*, *offset=None*, *duration=None*, *start_timestamp=None*, *out_of_bounds_mode='warn'*)

Load audio from files

Deal with the various possible input types to load an audio file Also attempts to load metadata using tinytag.

Audio objects only support mono (one-channel) at this time. Files with multiple channels are mixed down to a single channel.

Optionally, load only a piece of a file using *offset* and *duration*. This will efficiently read sections of a .wav file regardless of where the desired clip is in the audio. For mp3 files, access time grows linearly with time since the beginning of the file.

This function relies on librosa.load(), which supports wav natively but requires ffmpeg for mp3 support.

Parameters

- **path** (*str, Path*) – path to an audio file
- **sample_rate** (*int, None*) – resample audio with value and resample_type, if None use source sample_rate (default: None)
- **resample_type** – method used to resample_type (default: kaiser_fast)
- **metadata** (*bool*) – if True, attempts to load metadata from the audio file. If an exception occurs, self.metadata will be *None*. Otherwise self.metadata is a dictionary. Note: will also attempt to parse AudioMoth metadata from the *comment* field, if the *artist* field includes *AudioMoth*. The parsing function for AudioMoth is likely to break when new firmware versions change the *comment* metadata field.
- **offset** – load audio starting at this time (seconds) after the start of the file. Defaults to 0 seconds. - cannot specify both *offset* and *start_timestamp*
- **duration** – load audio of this duration (seconds) starting at *offset*. If None, loads all the way to the end of the file.
- **start_timestamp** – load audio starting at this localized datetime.datetime timestamp - cannot specify both *offset* and *start_timestamp* - will only work if loading metadata results in localized datetime

  object for 'recording_start_time' key

  – will raise AudioOutOfBoundsError if requested time period

  is not full contained within the audio file Example of creating localized timestamp: ` `import pytz; from datetime import datetime; local_timestamp = datetime(2020,12,25,23,59,59) local_timezone = pytz.timezone('US/Eastern') timestamp = local_timezone.localize(local_timestamp)` `
- **out_of_bounds_mode** –
  - 'warn': generate a warning [default]
  - 'raise': raise an AudioOutOfBoundsError
  - 'ignore': return any available audio with no warning/error

Returns samples, sample_rate, resample_type, metadata (dict or None)

Return type Audio object with attributes

Note: default sample_rate=None means use file's sample rate, does not resample

loop(*length=None*, *n=None*)

Extend audio file by looping it

Parameters

- **length** – the final length in seconds of the looped file (cannot be used with n)[default: None]

- **n** – the number of occurences of the original audio sample (cannot be used with length) [default: None] For example, n=1 returns the original sample, and n=2 returns two concatenated copies of the original sample

Returns a new Audio object of the desired length or repetitions

resample(*sample_rate*, *resample_type=None*)

Resample Audio object

Parameters

- **sample_rate** (*scalar*) – the new sample rate
- **resample_type** (*str*) – resampling algorithm to use [default: None (uses self.resample_type of instance)]

Returns a new Audio object of the desired sample rate

save(*path*)

Save Audio to file

NOTE: currently, only saving to .wav format supported

Parameters **path** – destination for output

spectrum()

Create frequency spectrum from an Audio object using fft

Parameters **self** –

Returns fft, frequencies

split(*clip_duration*, *clip_overlap=0*, *final_clip=None*)

Split Audio into even-lengthed clips

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

Parameters

- **clip_duration** (*float*) – The duration in seconds of the clips
- **clip_overlap** (*float*) – The overlap of the clips in seconds [default: 0]
- **final_clip** (*str*) – Behavior if final_clip is less than clip_duration seconds long. By default, discards remaining audio if less than clip_duration seconds long [default: None]. Options:
  - None: Discard the remainder (do not make a clip)
  - "extend": Extend the final clip with silence to reach clip_duration length
  - "remainder": Use only remainder of Audio (final clip will be shorter than clip_duration)
  - "full": Increase overlap with previous clip to yield a clip with clip_duration length

Returns list of audio objects - dataframe w/columns for start_time and end_time of each clip

Return type

- audio_clips

split_and_save(*destination*, *prefix*, *clip_duration*, *clip_overlap=0*, *final_clip=None*, *dry_run=False*)

Split audio into clips and save them to a folder

Parameters

- **destination** – A folder to write clips to

- **prefix** – A name to prepend to the written clips
- **clip_duration** – The duration of each clip in seconds
- **clip_overlap** – The overlap of each clip in seconds [default: 0]
- **final_clip** (*str*) – Behavior if final_clip is less than clip_duration seconds long. [default: None] By default, ignores final clip entirely. Possible options (any other input will ignore the final clip entirely),
  - "remainder": Include the remainder of the Audio (clip will not have clip_duration length)
  - "full": Increase the overlap to yield a clip with clip_duration length
  - "extend": Similar to remainder but extend (repeat) the clip to reach clip_duration length
  - None: Discard the remainder
- **dry_run** (*bool*) – If True, skip writing audio and just return clip DataFrame [default: False]

Returns pandas.DataFrame containing paths and start and end times for each clip

time_to_sample(*time*)

Given a time, convert it to the corresponding sample

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

Returns The rounded sample

Return type sample

trim(*start_time*, *end_time*)

Trim Audio object in time

If start_time is less than zero, output starts from time 0 If end_time is beyond the end of the sample, trims to end of sample

Parameters

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

Returns a new Audio object containing samples from start_time to end_time

Warning: metadata is lost during this operation

exception opensoundscape.audio.**AudioOutOfBoundsError**

Custom exception indicating the user tried to load audio outside of the time period that exists in the audio object

exception opensoundscape.audio.**OpsoLoadAudioInputError**

Custom exception indicating we can't load input

opensoundscape.audio.**load_channels_as_audio**(*path*, *sample_rate=None*, *resample_type='kaiser_fast'*, *offset=0*, *duration=None*)

Load each channel of an audio file to a separate Audio object

Provides a way to access individual channels, since Audio.from_file mixes down to mono by default

Parameters **Audio.from_file()** (*see*) –

Returns list of Audio objects (one per channel)

12.3 AudioMoth

Utilities specifically for audio files recoreded by AudioMoths

opensoundscape.audiomoth.**audiomoth_start_time**(*file*, *filename_timezone='UTC'*, *to_utc=False*)

parse audiomoth file name into a time stamp

AudioMoths create their file name based on the time that recording starts. This function parses the name into a timestamp. Older AudioMoth firmwares used a hexidecimal unix time format, while newer firmwares use a human-readable naming convention. This function handles both conventions.

Parameters

- **file** – (str) path or file name from AudioMoth recording
- **filename_timezone** – (str) name of a pytz time zone (for options see pytz.all_timezones). This is the time zone that the AudioMoth uses to record its name, not the time zone local to the recording site. Usually, this is 'UTC' because the AudioMoth records file names in UTC.
- **to_utc** – if True, converts timestamps to UTC localized time stamp. Otherwise, will return timestamp localized to *timezone* argument [default: False]

Returns localized datetime object - if to_utc=True, datetime is always "localized" to UTC

opensoundscape.audiomoth.**parse_audiomoth_metadata**(*metadata*)

parse a dictionary of AudioMoth .wav file metadata

-parses the comment field -adds keys for gain_setting, battery_state, recording_start_time -if available (firmware >=1.4.0), addes temperature

Notes on comment field: - Starting with Firmware 1.4.0, the audiomoth logs Temperature to the

metadata (wav header) eg "and temperature was 11.2C."

- At some point the firmware shifted from writing "gain setting 2" to "medium gain setting". Should handle both modes.

Tested for AudioMoth firmware versions: 1.5.0

Parameters **metadata** – dictionary with audiomoth metadata

Returns metadata dictionary with added keys and values

12.4 Audio Tools

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

opensoundscape.audio_tools.**bandpass_filter**(*signal*, *low_f*, *high_f*, *sample_rate*, *order=9*)

perform a butterworth bandpass filter on a discrete time signal using scipy.signal's butter and solfiltfilt (phase-preserving version of sosfilt)

Parameters

- **signal** – discrete time signal (audio samples, list of float)
- **low_f** – -3db point (?) for highpass filter (Hz)
- **high_f** – -3db point (?) for highpass filter (Hz)

- **sample_rate** – samples per second (Hz)
- **order=9** – higher values -> steeper dropoff

Returns filtered time signal

opensoundscape.audio_tools.**butter_bandpass**(*low_f*, *high_f*, *sample_rate*, *order=9*)

generate coefficients for bandpass_filter()

Parameters

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

Returns set of coefficients used in sosfiltfilt()

opensoundscape.audio_tools.**clipping_detector**(*samples*, *threshold=0.6*)

count the number of samples above a threshold value

Parameters

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

Returns number of samples exceeding threshold

opensoundscape.audio_tools.**convolve_file**(*in_file*, *out_file*, *ir_file*, *input_gain=1.0*)

apply an impulse_response to a file using ffmpeg's afir convolution

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

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

Parameters

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

Returns os response of ffmpeg command

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

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

Parameters

- **files_to_mix** – list of audio file paths
- **destination** – path to save mixdown to

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

Returns ffmpeg command line output

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

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

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

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

Parameters

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

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

If threshold is None: returns net_energy over background noise

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

Calculate audio energy with a sliding window

Calculate the energy in an array of audio samples

Parameters

- **samples** (*np.ndarray*) – array of audio samples loaded using librosa.load
- **window_len_samples** – samples per window
- **overlap_len_samples** – number of samples shared between consecutive windows

Returns list of energy level (float) for each window

12.5 Spectrogram

spectrogram.py: Utilities for dealing with spectrograms

class opensoundscape.spectrogram.**MelSpectrogram**(*spectrogram*, *frequencies*, *times*, *decibel_limits*, *window_samples=None*, *overlap_samples=None*, *window_type=None*, *audio_sample_rate=None*)

Immutable mel-spectrogram container

A mel spectrogram is a spectrogram with pseudo-logarithmically spaced frequency bins (see literature) rather than linearly spaced bins.

See Spectrogram class an Librosa's melspectrogram for detailed documentation.

NOTE: Here we rely on scipy's spectrogram function (via Spectrogram) rather than on librosa's _spectrogram or melspectrogram, because the amplitude of librosa's spectrograms do not match expectations. We only use the mel frequency bank from Librosa.

classmethod from_audio(*audio*, *n_mels=64*, *window_samples=512*, *overlap_samples=256*, *decibel_limits=(-100*, *-20)*, *htk=False*, *norm='slaney'*, *window_type='hann'*, *dB_scale=True*)

Create a MelSpectrogram object from an Audio object

First creates a spectrogram and a mel-frequency filter bank, then computes the dot product of the filter bank with the spectrogram.

The kwargs for the mel frequency bank are documented at: - https://librosa.org/doc/latest/generated/librosa.feature.melspectrogram.html#librosa.feature.melspectrogram - https://librosa.org/doc/latest/generated/librosa.filters.mel.html?librosa.filters.mel

Parameters

- **n_mels** – Number of mel bands to generate [default: 128] Note: n_mels should be chosen for compatibility with the Spectrogram parameter *window_samples*. Choosing a value > ~ *window_samples/10* will result in zero-valued rows while small values blend rows from the original spectrogram.
- **window_type** – The windowing function to use [default: "hann"]
- **window_samples** – n samples per window [default: 512]
- **overlap_samples** – n samples shared by consecutive windows [default: 256]
- **htk** – use HTK mel-filter bank instead of Slaney, see Librosa docs [default: False]
- **norm='slanley'** – mel filter bank normalization, see Librosa docs
- **dB_scale=True** – If True, rescales values to decibels, x=10*log10(x) - if dB_scale is False, decibel_limits is ignored

Returns opensoundscape.spectrogram.MelSpectrogram object

plot(*inline=True*, *fname=None*, *show_colorbar=False*)

Plot the mel spectrogram with matplotlib.pyplot

We can't use pcolormesh because it will smash pixels to achieve a linear y-axis, rather than preserving the mel scale.

Parameters

- **inline=True** –

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

class opensoundscape.spectrogram.**Spectrogram**(*spectrogram*, *frequencies*, *times*, *decibel_limits*, *window_samples=None*, *overlap_samples=None*, *window_type=None*, *audio_sample_rate=None*)

Immutable spectrogram container

Can be initialized directly from spectrogram, frequency, and time values or created from an Audio object using the .from_audio() method.

frequencies
(list) discrete frequency bins generated by fft

times
(list) time from beginning of file to the center of each window

spectrogram
a 2d array containing 10*log10(fft) for each time window

decibel_limits
minimum and maximum decibel values in .spectrogram

window_samples
number of samples per window when spec was created [default: none]

overlap_samples
number of samples overlapped in consecutive windows when spec was created [default: none]

window_type
window fn used to make spectrogram, eg 'hann' [default: none]

audio_sample_rate
sample rate of audio from which spec was created [default: none]

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

by summing pixels in the vertical dimension

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

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

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

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

Parameters

- **min_f** – low frequency in Hz for bandpass
- **max_f** – high frequency in Hz for bandpass
- **out_of_bounds_ok** – (bool) if False, raises ValueError if min_f or max_f are not within the range of the original spectrogram's frequencies [default: True]

Returns bandpassed spectrogram object

duration()
calculate the ammount of time represented in the spectrogram

Note: time may be shorter than the duration of the audio from which the spectrogram was created, because the windows may align in a way such that some samples from the end of the original audio were discarded

classmethod from_audio(*audio*, *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*)
create a Spectrogram object from an Audio object

Parameters

- **window_type="hann"** – see scipy.signal.spectrogram docs for description of window parameter
- **window_samples** – number of audio samples per spectrogram window (pixel) - Defaults to 512 if window_samples and window_length_sec are None - Note: cannot specify both window_samples and window_length_sec
- **window_length_sec** – length of a single window in seconds - Note: cannot specify both window_samples and window_length_sec - Warning: specifying this parameter often results in less efficient

  spectrogram computation because window_samples will not be a power of 2.
- **overlap_samples** – number of samples shared by consecutive windows - Note: must not specify both overlap_samples and overlap_fraction
- **overlap_fraction** – fractional temporal overlap between consecutive windows - Defaults to 0.5 if overlap_samples and overlap_fraction are None - Note: cannot specify both overlap_samples and overlap_fraction
- **fft_size** – see scipy.signal.spectrogram's *nfft* parameter
- **decibel_limits** – limit the dB values to (min,max) (lower values set to min, higher values set to max)
- **dB_scale** – If True, rescales values to decibels, x=10*log10(x) - if dB_scale is False, decibel_limits is ignored

Returns opensoundscape.spectrogram.Spectrogram object

classmethod from_file()
create a Spectrogram object from a file

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

Returns opensoundscape.spectrogram.Spectrogram object

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

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

similar to Audacity's gain and range parameters

Parameters

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

Returns Spectrogram object with db range applied

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

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

Returns Spectrogram object with values rescaled to feature_range

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

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

Returns Spectrogram object with values rescaled to feature_range

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

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

Parameters

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

return: time-series array of net amplitude

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

Parameters

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

to_image (*shape=None*, *channels=1*, *colormap=None*, *invert=False*, *return_type='pil'*)
Create an image from spectrogram (array, tensor, or PIL.Image)

Linearly rescales values in the spectrogram from self.decibel_limits to [0,255] (PIL.Image) or [0,1] (array/tensor)

Default of self.decibel_limits on load is [-100, -20], so, e.g., -20 db is loudest -> black, -100 db is quietest -> white

Parameters

- **shape** – tuple of output dimensions as (height, width) - if None, retains original shape of self.spectrogram
- **channels** – eg 3 for rgb, 1 for greyscale - must be 3 to use colormap
- **colormap** – if None, greyscale spectrogram is generated Can be any matplotlib colormap name such as 'jet'
- **return_type** – type of returned object - 'pil': PIL.Image - 'np': numpy.ndarray - 'torch': torch.tensor

Returns

- **PIL.Image with c channels and shape w,h given by *shape*** and values in [0,255]
- np.ndarray with shape [c,h,w] and values in [0,1]

- or torch.tensor with shape [c,h,w] and values in [0,1]

Return type Image/array with type depending on *return_type*

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

Parameters

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

Returns spectrogram object from extracted time segment

window_length()
calculate length of a single fft window, in seconds:

window_start_times()
get start times of each window, rather than midpoint times

window_step()
calculate time difference (sec) between consecutive windows' centers

CHAPTER 13

Machine Learning

13.1 Convolutional Neural Networks

classes for pytorch machine learning models in opensoundscape

For tutorials, see notebooks on opensoundscape.org

class opensoundscape.torch.models.cnn.**CNN**(*architecture, classes, sample_duration, single_target=False, preprocessor_class=<class 'opensoundscape.preprocess.preprocessors.SpectrogramPreprocessor'>, sample_shape=[224, 224, 3]*)

Generic CNN Model with .train(), .predict(), and .save()

flexible architecture, optimizer, loss function, parameters

for tutorials and examples see opensoundscape.org

Parameters

- **architecture** – *EITHER* a pytorch model object (subclass of torch.nn.Module), for example one generated with the *cnn_architectures* module *OR* a string matching one of the architectures listed by cnn_architectures.list_architectures(), eg 'resnet18'. - If a string is provided, uses default parameters

  (including use_pretrained=True)

- **classes** – list of class names. Must match with training dataset classes if training.
- **single_target** –
  - True: model expects exactly one positive class per sample
  - False: samples can have any number of positive classes

  [default: False]

eval(*targets*, *scores*, *logging_offset=0*)

compute single-target or multi-target metrics from targets and scores

Override this function to use a different set of metrics. It should always return (1) a single score (float) used as an overall metric of model quality and (2) a dictionary of computed metrics

Parameters

- **targets** – 0/1 for each sample and each class
- **scores** – continuous values in 0/1 for each sample and class
- **logging_offset** – modify verbosity - for example, -1 will reduce the amount of printing/logging by 1 level

load_weights(*path*, *strict=True*)

load network weights state dict from a file

For instance, load weights saved with .save_weights() in-place operation

Parameters

- **path** – file path with saved weights
- **strict** – (bool) see torch.load()

predict(*samples*, *batch_size=1*, *num_workers=0*, *activation_layer=None*, *binary_preds=None*, *threshold=None*, *split_files_into_clips=True*, *overlap_fraction=0*, *final_clip=None*, *bypass_augmentations=True*, *unsafe_samples_log=None*)

Generate predictions on a dataset

Choose to return any combination of scores, labels, and single-target or multi-target binary predictions. Also choose activation layer for scores (softmax, sigmoid, softmax then logit, or None). Binary predictions are performed post-activation layer

Note: the order of returned dataframes is (scores, preds, labels)

Parameters

- **samples** – the files to generate predictions for. Can be: - a dataframe with index containing audio paths, OR - a list (or np.ndarray) of audio file paths
- **batch_size** – Number of files to load simultaneously [default: 1]
- **num_workers** – parallelization (ie cpus or cores), use 0 for current process [default: 0]
- **activation_layer** – Optionally apply an activation layer such as sigmoid or softmax to the raw outputs of the model. options: - None: no activation, return raw scores (ie logit, [-inf:inf]) - 'softmax': scores all classes sum to 1 - 'sigmoid': all scores in [0,1] but don't sum to 1 - 'softmax_and_logit': applies softmax first then logit [default: None]
- **binary_preds** – Optionally return binary (thresholded 0/1) predictions options: - 'single_target': max scoring class = 1, others = 0 - 'multi_target': scores above threshold = 1, others = 0 - None: do not create or return binary predictions [default: None] Note: if you choose multi-target, you must specify *threshold*
- **threshold** – prediction threshold(s) for post-activation layer scores. Only relevant when binary_preds == 'multi_target' If activation layer is sigmoid, choose value in [0,1] If activation layer is None or softmax_and_logit, in [-inf,inf]
- **overlap_fraction** – fraction of overlap between consecutive clips when predicting on clips of longer audio files. For instance, 0.5 gives 50% overlap between consecutive clips.
- **final_clip** – see *opensoundscape.helpers.generate_clip_times_df*
- **bypass_augmentations** – If False, Actions with is_augmentation==True are performed. Default True.

- **unsafe_samples_log** – if not None, samples that failed to preprocess will be listed in this text file.

Returns df of post-activation_layer scores predictions: df of 0/1 preds for each class unsafe_samples: list of samples that failed to preprocess

Return type scores

Note: if loading an audio file raises a PreprocessingError, the scores and predictions for that sample will be np.nan

Note: if no return type is selected for *binary_preds*, returns None instead of a DataFrame for *predictions*

save_weights(*path*)
save just the weights of the network

This allows the saved weights to be used more flexibly than model.save() which will pickle the entire object. The weights are saved in a pickled dictionary using torch.save(self.network.state_dict())

Parameters **path** – location to save weights file

train(*train_df*, *validation_df=None*, *epochs=1*, *batch_size=1*, *num_workers=0*, *save_path='.'*, *save_interval=1*, *log_interval=10*, *validation_interval=1*, *unsafe_samples_log='./unsafe_training_samples.log'*)
train the model on samples from train_dataset

If customized loss functions, networks, optimizers, or schedulers are desired, modify the respective attributes before calling .train().

Parameters

- **train_df** – a dataframe of files and labels for training the model
- **validation_df** – a dataframe of files and labels for evaluating the model [default: None means no validation is performed]
- **epochs** – number of epochs to train for (1 epoch constitutes 1 view of each training sample)
- **batch_size** – number of training files simultaneously passed through forward pass, loss function, and backpropagation
- **num_workers** – number of parallel CPU tasks for preprocessing Note: use 0 for single (root) process (not 1)
- **save_path** – location to save intermediate and best model objects [default="."", ie current location of script]
- **save_interval** – interval in epochs to save model object with weights [default:1] Note: the best model is always saved to best.model in addition to other saved epochs.
- **log_interval** – interval in epochs to evaluate model with validation dataset and print metrics to the log
- **validation_interval** – interval in epochs to test the model on the validation set Note that model will only update it's best score and save best.model file on epochs that it performs validation.
- **unsafe_samples_log** – file path: log all samples that failed in preprocessing (file written when training completes) - if None, does not write a file

class opensoundscape.torch.models.cnn.**InceptionV3**(*classes, sample_duration, single_target=False, preprocessor_class=<class 'opensoundscape.preprocess.preprocessors.SpectrogramPreprocessor'>, freeze_feature_extractor=False, use_pretrained=True, sample_shape=[299, 299, 3]*)

opensoundscape.torch.models.cnn.**load_model**(*path*, *device=None*)

load a saved model object

Parameters

- **path** – file path of saved model
- **device** – which device to load into, eg 'cuda:1'
- **[default** – None] will choose first gpu if available, otherwise cpu

Returns a model object with loaded weights

opensoundscape.torch.models.cnn.**load_outdated_model**(*path, architecture, sample_duration, model_class=<class 'opensoundscape.torch.models.cnn.CNN'>, device=None*)

load a CNN saved with a previous version of OpenSoundscape

This function enables you to load models saved with opso 0.4.x and 0.5.x. If your model was saved with .save() in a previous version of OpenSoundscape >=0.6.0, you must re-load the model using the original package version and save it's network's state dict, i.e., *torch.save(model.network.state_dict(),path)*, then load the state dict to a new model object with model.load_weights(). See the *Predict with pre-trained CNN* tutorial for details.

For models created with the same version of OpenSoundscape as the one you are using, simply use opensoundscape.torch.models.cnn.load_model().

Note: for future use of the loaded model, you can simply call *model.save(path)* after creating it, then reload it with *model = load_model(path)*. The saved model will be fully compatible with opensoundscape >=0.7.0.

Examples: ‘‘‘ #load a binary resnet18 model from opso 0.4.x, 0.5.x, or 0.6.0 from opensoundscape.torch.models.cnn import CNN model = load_outdated_model('old_model.tar',architecture='resnet18')

#load a resnet50 model of class CNN created with opso 0.5.0 from opensoundscape.torch.models.cnn import CNN model_050 = load_outdated_model('opso050_pytorch_model_r50.model',architecture='resnet50') ‘‘‘

Parameters

- **path** – path to model file, ie .model or .tar file
- **architecture** – see CNN docs (pass None if the class __init__ does not take architecture as an argument)
- **sample_duration** – length of samples in seconds
- **model_class** – class to construct. Normally CNN.
- **device** – optionally specify a device to map tensors onto, eg 'cpu', 'cuda:0', 'cuda:1'[default: None] - if None, will choose cuda:0 if cuda is available, otherwise chooses cpu

Returns a cnn model object with the weights loaded from the saved model

opensoundscape.torch.models.cnn.**separate_resnet_feat_clf**(*model*)
Separate feature/classifier training params for a ResNet model

Parameters **model** – an opso model object with a pytorch resnet architecture

Returs: model with modified .optimizer_params and ._init_optimizer() method

Effects: creates a new self.opt_net object that replaces the old one resets self.current_epoch to 0

opensoundscape.torch.models.cnn.**use_resample_loss**(*model*)
Modify a model to use ResampleLoss for multi-target training

ResampleLoss may perform better than BCE Loss for multitarget problems in some scenarios.

class opensoundscape.torch.models.utils.**BaseModule**
Base class for a pytorch model pipeline class.

All child classes should define load, save, etc

opensoundscape.torch.models.utils.**apply_activation_layer**(*x*, *activation_layer=None*)
applies an activation layer to a set of scores

Parameters

- **x** – input values
- **activation_layer** –
  - None [default]: return original values
  - 'softmax': apply softmax activation
  - 'sigmoid': apply sigmoid activation
  - 'softmax_and_logit': apply softmax then logit transform

Returns values with activation layer applied

opensoundscape.torch.models.utils.**cas_dataloader**(*dataset*, *batch_size*, *num_workers*)
Return a dataloader that uses the class aware sampler

Class aware sampler tries to balance the examples per class in each batch. It selects just a few classes to be present in each batch, then samples those classes for even representation in the batch.

Parameters

- **dataset** – a pytorch dataset type object
- **batch_size** – see DataLoader
- **num_workers** – see DataLoader

opensoundscape.torch.models.utils.**collate_lists_of_audio_clips**(*batch*)
Collate function for splitting + prediction of long audio files

Puts each data field into a tensor with outer dimension batch size

Additionally, concats the dfs from each audio file into one long df for the entire batch

opensoundscape.torch.models.utils.**get_batch**(*array*, *batch_size*, *batch_number*)
get a single slice of a larger array

using the batch size and batch index, from zero

Parameters

- **array** – iterable to split into batches

- **batch_size** – num elements per batch
- **batch_number** – index of batch

Returns one batch (subset of array)

Note: the final elements are returned as the last batch even if there are fewer than batch_size

Example

if array=[1,2,3,4,5,6,7] then:

- get_batch(array,3,0) returns [1,2,3]
- get_batch(array,3,3) returns [7]

opensoundscape.torch.models.utils.**tensor_binary_predictions**(*scores*, *mode*, *threshold=None*)

generate binary 0/1 predictions from continuous scores

Does not transform scores: compares scores directly to threshold.

Parameters

- **scores** – torch.Tensor of dim (batch_size, n_classes) with input scores [-inf:inf]
- **mode** – 'single_target', 'multi_target', or None (return empty tensor)
- **threshold** – minimum score to predict 1, if mode=='multi_target'. threshold
- **be a single value for all classes or a list of class-specific values.** (*can*) –

Returns torch.Tensor of 0/1 predictions in same shape as scores

Note: expects real-valued (unbounded) input scores, i.e. scores take values in [-inf, inf]. Sigmoid layer is applied before multi-target prediction, so the threshold should be in [0,1].

Module to initialize PyTorch CNN architectures with custom output shape

This module allows the use of several built-in CNN architectures from PyTorch. The architecture refers to the specific layers and layer input/output shapes (including convolution sizes and strides, etc) - such as the ResNet18 or Inception V3 architecture.

We provide wrappers which modify the output layer to the desired shape (to match the number of classes). The way to change the output layer shape depends on the architecture, which is why we need a wrapper for each one. This code is based on pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

To use these wrappers, for example, if your model has 10 output classes, write

my_arch=resnet18(10)

Then you can initialize a model object from *opensoundscape.torch.models.cnn* with your architecture:

model=PytorchModel(my_arch,classes)

or override an existing model's architecture:

model.network = my_arch

Note: the InceptionV3 architecture must be used differently than other architectures - the easiest way is to simply use the InceptionV3 class in opensoundscape.torch.models.cnn.

opensoundscape.torch.architectures.cnn_architectures.**alexnet**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for AlexNet architecture

input size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**densenet121**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for densenet121 architecture

input size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**freeze_params**(*model*)

remove gradients (aka freeze) all model parameters

opensoundscape.torch.architectures.cnn_architectures.**inception_v3**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for Inception v3 architecture

Input: 229x229

WARNING: expects (299,299) sized images and has auxiliary output. See InceptionV3 class in *opensoundscape.torch.models.cnn* for use.

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**modify_resnet**(*model*, *num_classes*, *num_channels*)

modify input and output shape of a resnet architecture

opensoundscape.torch.architectures.cnn_architectures.**resnet101**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for ResNet101 architecture

input_size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**resnet152**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for ResNet152 architecture

input_size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**resnet18**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for ResNet18 architecture

input_size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**resnet34**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for ResNet34 architecture

input_size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**resnet50**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for ResNet50 architecture

input_size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**squeezenet1_0**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for squeezenet architecture

input size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.
- **num_channels** – specify channels in input sample, eg [channels h,w] sample shape

opensoundscape.torch.architectures.cnn_architectures.**vgg11_bn**(*num_classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *num_channels=3*)

Wrapper for vgg11 architecture

input size = 224

Parameters

- **num_classes** – number of output nodes for the final layer
- **freeze_feature_extractor** – if False (default), entire network will have gradients and can train if True, feature block is frozen and only final layer is trained
- **use_pretrained** – if True, uses pre-trained ImageNet features from Pytorch's model zoo.

defines feature extractor and Architecture class for ResNet CNN

This implementation of the ResNet18 architecture allows for separate access to the feature extraction and classification blocks. This can be useful, for instance, to freeze the feature extractor and only train the classifier layer; or to specify different learning rates for the two blocks.

This implementation is used in the Resnet18Binary and Resnet18Multiclass classes of opensoundscape.torch.models.cnn.

class opensoundscape.torch.architectures.resnet.**ResNetArchitecture**(*num_cls*, *weights_init='ImageNet'*, *num_layers=18*, *init_classifier_weights=False*)

ResNet architecture with 18 or 50 layers

This implementation enables separate access to feature and classification blocks.

Parameters

- **num_cls** – number of classes (int)
- **weights_init** –
  - "ImageNet": load the pre-trained weights for ImageNet dataset
  - path: load weights from a path on your computer or a url
  - None: initialize with random weights
- **num_layers** – 18 for Resnet18 or 50 for Resnet50
- **init_classifier_weights** –
  - if True, load the weights of the classification layer as well as

  feature extraction layers - if False (default), only load the weights of the feature extraction layers

load(*init_path*, *init_classifier_weights=True*, *verbose=False*)

load state dict (weights) of the feature+classifier optionally load only feature weights not classifier weights

Parameters

- **init_path** –
  - url containing "http": download weights from web
  - path: load weights from local path
- **init_classifier_weights** –
  - if True, load the weights of the classification layer as well as

  feature extraction layers - if False (default), only load the weights of the feature extraction layers

- **verbose** – if True, print missing/unused keys [default: False]

class opensoundscape.torch.architectures.resnet.**ResNetFeature**(*block*, *layers*, *zero_init_residual=False*, *groups=1*, *width_per_group=64*, *replace_stride_with_dilation=None*, *norm_layer=None*)

class opensoundscape.torch.architectures.utils.**BaseArchitecture**

Base architecture for reference.

class opensoundscape.torch.architectures.utils.**CompositeArchitecture**(**args*, ***kwargs*)

Architecture with separate feature and classsifier blocks

13.2 Data Selection

opensoundscape.data_selection.**resample**(*df*, *n_samples_per_class*, *upsample=True*, *downsample=True*, *random_state=None*)

resample a one-hot encoded label df for a target n_samples_per_class

Parameters

- **df** – dataframe with one-hot encoded labels: columns are classes, index is sample name/path
- **n_samples_per_class** – target number of samples per class
- **upsample** – if True, duplicate samples for classes with <n samples to get to n samples
- **downsample** – if True, randomly sample classis with >n samples to get to n samples
- **random_state** – passed to np.random calls. If None, random state is not fixed.

Note: The algorithm assumes that the label df is single-label. If the label df is multi-label, some classes can end up over-represented.

Note 2: The resulting df will have samples ordered by class label, even if the input df had samples in a random order.

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

Given a input DataFrame of categorical labels, upsample to maximum value

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

The input df is NOT one-hot encoded in this case, but instead contains categorical labels in a specified label_columns

Parameters

- **input_df** – A DataFrame to upsample
- **label_column** – The column to draw unique labels from
- **random_state** – Set the random_state during sampling

Returns An upsampled DataFrame

Return type df

13.3 Grad Cam

GradCAM is a method of visualizing the activation of the network on parts of an image

Author: Kazuto Nakashima # URL: http://kazuto1011.github.io # Created: 2017-05-26

13.4 Loss Functions

loss function classes to use with opensoundscape models

class opensoundscape.torch.loss.**BCEWithLogitsLoss_hot**
use pytorch's nn.BCEWithLogitsLoss for one-hot labels by simply converting y from long to float

class opensoundscape.torch.loss.**CrossEntropyLoss_hot**
use pytorch's nn.CrossEntropyLoss for one-hot labels by converting labels from 1-hot to integer labels

throws a ValueError if labels are not one-hot

class opensoundscape.torch.loss.**ResampleLoss**(*class_freq*, *reduction='mean'*, *loss_weight=1.0*)

opensoundscape.torch.loss.**reduce_loss**(*loss*, *reduction*)
Reduce loss as specified.

Parameters

- **loss** (*Tensor*) – Elementwise loss tensor.
- **reduction** (*str*) – Options are "none", "mean" and "sum".

Returns Reduced loss tensor.

Return type Tensor

opensoundscape.torch.loss.**weight_reduce_loss**(*loss*, *weight=None*, *reduction='mean'*, *avg_factor=None*)
Apply element-wise weight and reduce loss.

Parameters

- **loss** (*Tensor*) – Element-wise loss.
- **weight** (*Tensor*) – Element-wise weights.
- **reduction** (*str*) – Same as built-in losses of PyTorch.
- **avg_factor** (*float*) – Avarage factor when computing the mean of losses.

Returns Processed loss values.

Return type Tensor

13.5 Safe Dataloading

Dataset wrapper to handle errors gracefully in Preprocessor classes

A SafeDataset handles errors in a potentially misleading way: If an error is raised while trying to load a sample, the SafeDataset will instead load a different sample. The indices of any samples that failed to load will be stored in ._unsafe_indices.

The behavior may be desireable for training a model, but could cause silent errors when predicting a model (replacing a bad file with a different file), and you should always be careful to check for ._unsafe_indices after using a SafeDataset.

based on an implementation by @msamogh in nonechucks (github.com/msamogh/nonechucks/)

class opensoundscape.torch.safe_dataset.**SafeDataset**(*dataset*, *unsafe_behavior*, *eager_eval=False*)

A wrapper for a Dataset that handles errors when loading samples

WARNING: When iterating, will skip the failed sample, but when using within a DataLoader, finds the next good sample and uses it for the current index (see __getitem__).

Note that this class does not subclass DataSet. Instead, it contains a *.dataset* attribute that is a DataSet (or a Preprocessor, which subclasses DataSet).

Parameters

- **dataset** – a torch Dataset instance or child such as a Preprocessor
- **eager_eval** – If True, checks if every file is able to be loaded during initialization (logs _safe_indices and _unsafe_indices)

Attributes: _safe_indices and _unsafe_indices can be accessed later to check which samples threw errors.

_build_index()

tries to load each sample, logs _safe_indices and _unsafe_indices

__getitem__(*index*)

If loading an index fails, keeps trying the next index until success

_safe_get_item()

Tries to load a sample, returns None if error occurs

is_index_built

Returns True if all indices of the original dataset have been classified into safe_samples_indices or _unsafe_samples_indices.

13.6 Sampling

classes for strategically sampling within a DataLoader

class opensoundscape.torch.sampling.**ClassAwareSampler**(*labels*, *num_samples_cls=1*)

In each batch of samples, pick a limited number of classes to include and give even representation to each class

class opensoundscape.torch.sampling.**ImbalancedDatasetSampler**(*dataset*, *indices=None*, *num_samples=None*, *callback_get_label=None*)

Samples elements randomly from a given list of indices for imbalanced dataset :param indices: a list of indices :type indices: list, optional :param num_samples: number of samples to draw :type num_samples: int, optional :param callback_get_label func: a callback-like function which takes two arguments - dataset and index

13.7 Performance Metrics

opensoundscape.metrics.**binary_predictions**(*scores*, *single_target=False*, *threshold=0.5*)

convert numeric scores to binary predictions

return 0/1 for an array of scores: samples (rows) x classes (columns)

Parameters

- **scores** – a 2-d list or np.array. row=sample, columns=classes
- **single_target** – if True, predict 1 for highest scoring class per sample, 0 for other classes. If False, predict 1 for all scores > threshold [default: False]
- **threshold** – Predict 1 for score > threshold. only used if single_target = False. [default: 0.5]

opensoundscape.metrics.**multi_target_metrics**(*targets*, *scores*, *class_names*, *threshold*)

generate various metrics for a set of scores and labels (targets)

Parameters

- **targets** – 0/1 lables in 2d array
- **scores** – continuous values in 2d array
- **class_names** – list of strings
- **threshold** – scores >= threshold result in prediction of 1, while scores < threshold result in prediction of 0

Returns dictionary of various overall and per-class metrics

Return type metrics_dict

opensoundscape.metrics.**single_target_metrics**(*targets*, *scores*, *class_names*)

generate various metrics for a set of scores and labels (targets)

Predicts 1 for the highest scoring class per sample and 0 for all other classes.

Parameters

- **targets** – 0/1 lables in 2d array
- **scores** – continuous values in 2d array
- **class_names** – list of strings

Returns dictionary of various overall and per-class metrics

Return type metrics_dict

CHAPTER 14

Preprocessing

14.1 Image Augmentation

Transforms and augmentations for PIL.Images

opensoundscape.preprocess.img_augment.**time_split**(*img*, *seed=None*)

Given a PIL.Image, split into left/right parts and swap

Randomly chooses the slicing location For example, if *h* chosen

abcdefghijklmnop ^

hijklmnop + abcdefg

Parameters **img** – A PIL.Image

Returns A PIL.Image

14.2 Preprocessing Actions

Actions for augmentation and preprocessing pipelines

This module contains Action classes which act as the elements in Preprocessor pipelines. Action classes have go(), on(), off(), and set() methods. They take a single sample of a specific type and return the transformed or augmented sample, which may or may not be the same type as the original.

See the preprocessor module and Preprocessing tutorial for details on how to use and create your own actions.

class opensoundscape.preprocess.actions.**Action**(*fn*, *is_augmentation=False*, *extra_args=[]*, ***kwargs*)

Action class for an arbitrary function

The function must take the sample as the first argument

Note that this allows two use cases: (A) regular function that takes an input object as first argument

eg. Audio.from_file(path,**kwargs)

(B) method of a class, which takes 'self' as the first argument, eg. Spectrogram.bandpass(self,**kwargs)

Other arguments are an arbitrary list of kwargs.

class opensoundscape.preprocess.actions.**AudioClipLoader**(***kwargs*)
Action to load clips from an audio file

Loads an audio file or part of a file to an Audio object. Will load entire audio file if _start_time and _end_time are None. see Audio.from_file() for documentation.

Parameters **Audio.from_file()** (*see*) –

class opensoundscape.preprocess.actions.**AudioTrim**(***kwargs*)
Action to trim/extend audio to desired length

Parameters **actions.trim_audio** (*see*) –

class opensoundscape.preprocess.actions.**BaseAction**
Parent class for all Actions (used in Preprocessor pipelines)

New actions should subclass this class.

Subclasses should set *self.requires_labels = True* if go() expects (X,y) instead of (X). y is a row of a dataframe (a pd.Series) with index (.name) = original file path, columns=class names, values=labels (0,1). X is the sample, and can be of various types (path, Audio, Spectrogram, Tensor, etc). See Overlay for an example of an Action that uses labels.

set(***kwargs*)
only allow keys that exist in self.params

class opensoundscape.preprocess.actions.**Overlay**(*is_augmentation=True*, ***kwargs*)
Action Class for augmentation that overlays samples on eachother

Required Args: overlay_df: dataframe of audio files (index) and labels to use for overlay update_labels (bool): if True, labels of sample are updated to include

labels of overlayed sample

See overlay() for other arguments and default values.

opensoundscape.preprocess.actions.**frequency_mask**(*tensor*, *max_masks=3*, *max_width=0.2*)
add random horizontal bars over Tensor

Parameters

- **tensor** – input Torch.tensor sample
- **max_masks** – max number of horizontal bars [default: 3]
- **max_width** – maximum size of horizontal bars as fraction of sample height

Returns augmented tensor

opensoundscape.preprocess.actions.**image_to_tensor**(*img*, *greyscale=False*)
Convert PIL image to RGB or greyscale Tensor (PIL.Image -> Tensor)

convert PIL.Image w/range [0,255] to torch Tensor w/range [0,1]

Parameters

- **img** – PIL.Image
- **greyscale** – if False, converts image to RGB (3 channels). If True, converts image to one channel.

opensoundscape.preprocess.actions.**overlay**(*x*, *_labels*, *_preprocessor*, *overlay_df*, *update_labels*, *overlay_class=None*, *overlay_prob=1*, *max_overlay_num=1*, *overlay_weight=0.5*)

iteratively overlay 2d samples on top of eachother

Overlays (blends) image-like samples from overlay_df on top of the sample with probability *overlay_prob* until stopping condition. If necessary, trims overlay audio to the length of the input audio.

Overlays can be used in a few general ways:

1. a separate df where any file can be overlayed (overlay_class=None)
2. **same df as training, where the overlay class is "different" ie,** does not contain overlapping labels with the original sample
3. **same df as training, where samples from a specific class are used** for overlays

Parameters

- **overlay_df** – a labels dataframe with audio files as the index and classes as columns
- **_labels** – labels of the original sample
- **_preprocessor** – the preprocessing pipeline
- **update_labels** – if True, add overlayed sample's labels to original sample
- **overlay_class** – how to choose files from overlay_df to overlay Options [default: "different"]: 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
- **overlay_prob** – the probability of applying each subsequent overlay
- **max_overlay_num** – the maximum number of samples to overlay on original - for example, if overlay_prob = 0.5 and max_overlay_num=2,

  1/2 of samples will recieve 1 overlay and 1/4 will recieve an additional second overlay
- **overlay_weight** – a float > 0 and < 1, or a list of 2 floats [min, max] between which the weight will be randomly chosen. e.g. [0.1,0.7] An overlay_weight <0.5 means more emphasis on original sample.

Returns overlayed sample, (possibly updated) labels

opensoundscape.preprocess.actions.**scale_tensor**(*tensor*, *input_mean=0.5*, *input_std=0.5*)

linear scaling of tensor values using torch.transforms.Normalize

(Tensor->Tensor)

WARNING: This does not perform per-image normalization. Instead, it takes as arguments a fixed u and s, ie for the entire dataset, and performs X=(X-input_mean)/input_std.

Parameters

- **input_mean** – mean of input sample pixels (average across dataset)
- **input_std** – standard deviation of input sample pixels (average across dataset)
- **are NOT the target mu and sd, but the original mu and sd of img** (*(these)* –

- **which the output will have mu=0, std=1)** (*for*) –

Returns modified tensor

opensoundscape.preprocess.actions.**tensor_add_noise**(*tensor*, *std=1*)
Add gaussian noise to sample (Tensor -> Tensor)

Parameters **std** – standard deviation for Gaussian noise [default: 1]

Note: be aware that scaling before/after this action will change the effect of a fixed stdev Gaussian noise

opensoundscape.preprocess.actions.**time_mask**(*tensor*, *max_masks=3*, *max_width=0.2*)
add random vertical bars over sample (Tensor -> Tensor)

Parameters

- **tensor** – input Torch.tensor sample
- **max_masks** – maximum number of vertical bars [default: 3]
- **max_width** – maximum size of bars as fraction of sample width

Returns augmented tensor

opensoundscape.preprocess.actions.**torch_color_jitter**(*tensor*, *brightness=0.3*, *contrast=0.3*, *saturation=0.3*, *hue=0*)
Wraps torchvision.transforms.ColorJitter

(Tensor -> Tensor) or (PIL Img -> PIL Img)

Parameters

- **tensor** – input sample
- **brightness=0.3** –
- **contrast=0.3** –
- **saturation=0.3** –
- **hue=0** –

Returns modified tensor

opensoundscape.preprocess.actions.**torch_random_affine**(*tensor*, *degrees=0*, *translate=(0.3*, *0.1)*, *fill=0*)
Wraps for torchvision.transforms.RandomAffine

(Tensor -> Tensor) or (PIL Img -> PIL Img)

Parameters

- **tensor** – torch.Tensor input saple
- **= 0** (*degrees*) –
- **=** (*translate*) –
- **= 0-255, duplicated across channels** (*fill*) –

Returns modified tensor

Note: If applying per-image normalization, we recommend applying RandomAffine after image normalization. In this case, an intermediate gray value is ~0. If normalization is applied after RandomAffine on a PIL image, use an intermediate fill color such as (122,122,122).

opensoundscape.preprocess.actions.**trim_audio**(*audio*, *_sample_duration*, *extend=True*, *random_trim=False*, *tol=1e-05*)

trim audio clips (Audio -> Audio)

Trims an audio file to desired length Allows audio to be trimmed from start or from a random time Optionally extends audio shorter than clip_length with silence

Parameters

- **audio** – Audio object
- **_sample_duration** – desired final length (sec) - if None, no trim is performed
- **extend** – if True, clips shorter than _sample_duration are extended with silence to required length
- **random_trim** – if True, chooses a random segment of length _sample_duration from the input audio. If False, the file is trimmed from 0 seconds to _sample_duration seconds.
- **tol** – tolerance for considering a clip to be of the correct length (sec)

Returns trimmed audio

14.3 Preprocessors

class opensoundscape.preprocess.preprocessors.**BasePreprocessor**(*sample_duration=None*)

Class for defining an ordered set of Actions and a way to run them

Custom Preprocessor classes should subclass this class or its children

Preprocessors have one job: to transform samples from some input (eg a file path) to some output (eg a torch.Tensor) using a specific procedure. The procedure consists of Actions ordered by the Preprocessor's index. Preprocessors have a forward() method which runs the set of Actions specified in the index.

Parameters

- **action_dict** – dictionary of name:Action actions to perform sequentially
- **sample_duration** – length of audio samples to generate (seconds)

forward(*sample*, *break_on_type=None*, *break_on_key=None*, *clip_times=None*, *bypass_augmentations=False*)

perform actions in self.pipeline on a sample (until a break point)

Actions with .bypass = True are skipped. Actions with .is_augmentation = True can be skipped by passing bypass_augmentations=True.

Parameters

- **sample** – either: - pd.Series with file path as index (.name) and labels - OR a file path as pathlib.Path or string
- **break_on_type** – if not None, the pipeline will be stopped when it reaches an Action of this class. The matching action is not performed.
- **break_on_key** – if not None, the pipeline will be stopped when it reaches an Action whose index equals this value. The matching action is not performed.
- **bypass_augmentations** – if True, actions with .is_augmentatino=True are skipped

Returns preprocessed sample, 'y':labels} if return_labels==True, otherwise {'X':preprocessed sample}

Return type {'X'

insert_action(*action_index*, *action*, *after_key=None*, *before_key=None*)
insert an action in specific specific position

This is an in-place operation

Inserts a new action before or after a specific key. If after_key and before_key are both None, action is appended to the end of the index.

Parameters

- **action_index** – string key for new action in index
- **action** – the action object, must be subclass of BaseAction
- **after_key** – insert the action immediately after this key in index
- **before_key** – insert the action immediately before this key in index Note: only one of (after_key, before_key) can be specified

remove_action(*action_index*)
alias for self.drop(...,inplace=True), removes an action

This is an in-place operation

Parameters **action_index** – index of action to remove

class opensoundscape.preprocess.preprocessors.**SpectrogramPreprocessor**(*sample_duration, overlay_df=None, out_shape=[224, 224, 3]*)
Child of BasePreprocessor that creates specrogram Tensors w/augmentation

loads audio, creates spectrogram, performs augmentations, returns tensor

by default, does not resample audio, but bandpasses to 0-11.025 kHz (to ensure all outputs have same scale in y-axis) can change with .load_audio.set(sample_rate=sr)

during prediction, will load clips from long audio files rather than entire audio files.

Parameters

- **sample_duration** – length in seconds of audio samples generated If not None, longer clips trimmed to this length. By default, shorter clips will be extended (modify random_trim_audio and trim_audio to change behavior).
- **overlay_df** – if not None, will include an overlay action drawing samples from this df
- **out_shape** – output shape of tensor h,w,channels [default: [224,224,3]]

exception opensoundscape.preprocess.utils.**PreprocessingError**
Custom exception indicating that a Preprocessor pipeline failed

opensoundscape.preprocess.utils.**get_args**(*func*)
get list of arguments and default values from a function

opensoundscape.preprocess.utils.**get_reqd_args**(*func*)
get list of required arguments and default values from a function

opensoundscape.preprocess.utils.**show_tensor**(*tensor*, *channel=None*, *transform_from_zero_centered=True*, *invert=True*)
helper function for displaying a sample as an image

Parameters

- **tensor** – torch.Tensor of shape [c,w,h] with values centered around zero
- **channel** – specify an integer to plot only one channel, otherwise will attempt to plot all channels
- **transform_from_zero_centered** – if True, transforms values from [-1,1] to [0,1]
- **invert** – if true, flips value range via x=1-x

opensoundscape.preprocess.utils.**show_tensor_grid**(*tensors*, *columns*, *channel=None*, *transform_from_zero_centered=True*, *invert=True*, *labels=None*)

create image of nxn tensors

Parameters

- **tensors** – list of samples
- **columns** – number of columns in grid
- **labels** – title of each subplot
- **other args, see show_tensor()** (*for*) –

14.4 Tensor Augmentation

Augmentations and transforms for torch.Tensors

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

opensoundscape.preprocess.tensor_augment.**freq_mask**(*spec*, *F=30*, *max_masks=3*, *replace_with_zero=False*)

draws horizontal bars over the image

F:maximum frequency-width of bars in pixels

max_masks: maximum number of bars to draw

replace_with_zero: if True, bars are 0s, otherwise, mean img value

opensoundscape.preprocess.tensor_augment.**time_mask**(*spec*, *T=40*, *max_masks=3*, *replace_with_zero=False*)

draws vertical bars over the image

T:maximum time-width of bars in pixels

max_masks: maximum number of bars to draw

replace_with_zero: if True, bars are 0s, otherwise, mean img value

opensoundscape.preprocess.tensor_augment.**time_warp**(*spec*, *W=5*)

apply time stretch and shearing to spectrogram

fills empty space on right side with horizontal bars

W controls amount of warping. Random with occasional large warp.

CHAPTER 15

Signal Processing

15.1 RIBBIT

Detect periodic vocalizations with RIBBIT

This module provides functionality to search audio for periodically fluctuating vocalizations.

opensoundscape.ribbit.**calculate_pulse_score**(*amplitude*, *amplitude_sample_rate*, *pulse_rate_range*, *plot=False*, *nfft=1024*)

Search for amplitude pulsing in an audio signal in a range of pulse repetition rates (PRR)

scores an audio amplitude signal by highest value of power spectral density in the PRR range

Parameters

- **amplitude** – a time series of the audio signal's amplitude (for instance a smoothed raw audio signal)
- **amplitude_sample_rate** – sample rate in Hz of amplitude signal, normally ~20-200 Hz
- **pulse_rate_range** – [min, max] values for amplitude modulation in Hz
- **plot=False** – if True, creates a plot visualizing the power spectral density
- **nfft=1024** – controls the resolution of the power spectral density (see scipy.signal.welch)

Returns pulse rate score for this audio segment (float)

opensoundscape.ribbit.**ribbit**(*spectrogram*, *signal_band*, *pulse_rate_range*, *clip_duration*, *clip_overlap=0*, *final_clip=None*, *noise_bands=None*, *plot=False*)

Run RIBBIT detector to search for periodic calls in audio

This tool searches for periodic energy fluctuations at specific repetition rates and frequencies.

Parameters

- **spectrogram** – opensoundscape.Spectrogram object of an audio file
- **signal_band** – [min, max] frequency range of the target species, in Hz

- **pulse_rate_range** – [min,max] pulses per second for the target species
- **clip_duration** – the length of audio (in seconds) to analyze at one time - each clip is analyzed independently and recieves a ribbit score
- **clip_overlap** (*float*) – overlap between consecutive clips (sec)
- **final_clip** (*str*) – behavior if final clip is less than clip_duration seconds long. By default, discards remaining audio if less than clip_duration seconds long [default: None]. Options: - None: Discard the remainder (do not make a clip) - "remainder": Use only remainder of Audio (final clip will be shorter than clip_duration) - "full": Increase overlap with previous clip to yield a clip with clip_duration length Note that the "extend" option is not supported for RIBBIT.
- **noise_bands** – list of frequency ranges to subtract from the signal_band For instance: [[min1,max1] , [min2,max2]] - if *None*, no noise bands are used - default: None
- **plot=False** – if True, plot the power spectral density for each clip

Returns DataFrame of index=('start_time','end_time'), columns=['score'], with a row for each clip.

Notes

__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. 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. For example, choosing *pulse_rate_range = [10, 20]* means that RIBBIT should look for pulses no slower than 10 pulses per second and no faster than 20 pulses per second.

Clip Duration: The *clip_duration* parameter tells RIBBIT how many seconds of audio to analyze at one time. Generally, you should choose a *clip_length* that is similar to the length of the target species vocalization, or a little bit longer. For very slowly pulsing vocalizations, choose a longer window so that at least 5 pulses can occur in one window (0.5 pulses per second -> 10 second window). Typical values for are 0.3 to 10 seconds. Also, *clip_overlap* can be used for overlap between sequential clips. This is more computationally expensive but will be more likely to center a target sound in the clip (with zero overlap, the target sound may be split up between adjacent clips).

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*).

__ALGORITHM__ This is the procedure RIBBIT follows: divide the audio into segments of length clip_duration for each clip:

> calculate time series of energy in signal band (signal_band) and subtract noise band energies (noise_bands) calculate power spectral density of the amplitude time series score the file based on the maximum value of power spectral density in the pulse rate range

15.2 Signal Processing

Signal processing tools for feature extraction and more

opensoundscape.signal.**cwt_peaks**(*audio*, *center_frequency*, *wavelet='morl'*, *peak_threshold=0.2*, *peak_separation=None*, *plot=False*)

compute a cwt, post-process, then extract peaks

Performs a continuous wavelet transform (cwt) on an audio signal at a single frequency. It then squares, smooths, and normalizes the signal. Finally, it detects peaks in the resulting signal and returns the times and magnitudes of detected peaks. It is used as a feature extractor for Ruffed Grouse drumming detection.

Parameters

- **audio** – an Audio object
- **center_frequency** – the target frequency to extract peaks from
- **wavelet** – (str) name of a pywt wavelet, eg 'morl' (see pywt docs)
- **peak_threshold** – minimum height of peaks - if None, no minimum peak height - see "height" argument to scipy.signal.find_peaks
- **peak_separation** – minimum time between detected peaks, in seconds - if None, no minimum distance - see "distance" argument to scipy.signal.find_peaks

Returns list of times (from beginning of signal) of each peak peak_levels: list of magnitudes of each detected peak

Return type peak_times

Note: consider downsampling audio to reduce computational cost. Audio must have sample rate of at least 2x target frequency.

---

opensoundscape.signal.**detect_peak_sequence_cwt**(*audio*, *sr=400*, *window_len=60*, *center_frequency=50*, *wavelet='morl'*, *peak_threshold=0.2*, *peak_separation=0.0375*, *dt_range=[0.05, 0.8]*, *dy_range=[-0.2, 0]*, *d2y_range=[-0.05, 0.15]*, *max_skip=3*, *duration_range=[1, 15]*, *points_range=[9, 100]*, *plot=False*)

Use a continuous wavelet transform to detect accellerating sequences

This function creates a continuous wavelet transform (cwt) feature and searches for accelerating sequences of peaks in the feature. It was developed to detect Ruffed Grouse drumming events in audio signals. Default parameters are tuned for Ruffed Grouse drumming detection.

Analysis is performed on analysis windows of fixed length without overlap. Detections from each analysis window across the audio file are aggregated.

Parameters

- **audio** – Audio object
- **sr=400** – resample audio to this sample rate (Hz)
- **window_len=60** – length of analysis window (sec)
- **center_frequency=50** – target audio frequency of cwt
- **wavelet='morl'** – (str) pywt wavelet name (see pywavelets docs)

- **peak_threshold=0.2** – height threhsold (0-1) for peaks in normalized signal
- **peak_separation=15/400** – min separation (sec) for peak finding
- **0.8]** (*dt_range=[0.05,*) – sequence detection point-to-point criterion 1 - Note: the upper limit is also used as sequence termination criterion 2
- **0]** (*dy_range=[-0.2,*) – sequence detection point-to-point criterion 2
- **0.15]** (*d2y_range=[-0.05,*) – sequence detection point-to-point criterion 3
- **max_skip=3** – sequence termination criterion 1: max sequential invalid points
- **15]** (*duration_range=[1,*) – sequence criterion 1: length (sec) of sequence
- **100]** (*points_range=[9,*) – sequence criterion 2: num points in sequence
- **plot=False** – if True, plot peaks and detected sequences with pyplot

Returns dataframe summarizing detected sequences

Note: for Ruffed Grouse drumming, which is very low pitched, audio is resampled to 400 Hz. This greatly increases the efficiency of the cwt, but will only detect frequencies up to 400/2=200Hz. Generally, choose a resample frequency as low as possible but >=2x the target frequency

Note: the cwt signal is normalized on each analysis window, so changing the analysis window size can change the detection results.

Note: if there is an incomplete window remaining at the end of the audio file, it is discarded (not analyzed).

opensoundscape.signal.**find_accel_sequences**(*t, dt_range=[0.05, 0.8], dy_range=[-0.2, 0], d2y_range=[-0.05, 0.15], max_skip=3, duration_range=[1, 15], points_range=[5, 100]*)

detect accelerating/decelerating sequences in time series

developed for deteting Ruffed Grouse drumming events in a series of peaks extracted from cwt signal

The algorithm computes the forward difference of t, y(t). It iterates through the [y(t), t] points searching for sequences of points that meet a set of conditions. It begins with an empty candidate sequence.

"Point-to-point criterea": Valid ranges for dt, dy, and d2y are checked for each subsequent point and are based on previous points in the candidate sequence. If they are met, the point is added to the candidate sequence.

"Continuation criterea": Conditions for max_skip and the upper bound of dt are used to determine when a sequence should be terminated.

- max_skip: max number of sequential invalid points before terminating
- dt<=dt_range[1]: if dt is long, sequence should be broken

"Sequence criterea": When a sequence is terminated, it is evaluated on conditions for duration_range and points_range. If it meets these conditions, it is saved as a detected sequence.

- duration_range: length of sequence in seconds from first to last point
- points_range: number of points included in sequence

When a sequence is terminated, the search continues with the next point and an empty sequence.

Parameters

- **t** – (list or np.array) times of all detected peaks (seconds)
- **dt_range=[0.05,0.8]** – valid values for t(i) - t(i-1)
- **dy_range=[-0.2,0]** – valid values for change in y (grouse: difference in time between consecutive beats should decrease)

- **d2y_range=[-.05,15]** – limit change in dy: should not show large decrease (sharp curve downward on y vs t plot)
- **max_skip=3** – max invalid points between valid points for a sequence (grouse: should not have many noisy points between beats)
- **duration_range=[1,15]** – total duration of sequence (sec)
- **points_range=[9,100]** – total number of points in sequence

Returns lists of t and y for each detected sequence

Return type sequences_t, sequences_y

opensoundscape.signal.**frequency2scale**(*frequency*, *wavelet*, *sr*)

determine appropriate wavelet scale for desired center frequency

Parameters

- **frequency** – desired center frequency of wavelet in Hz (1/seconds)
- **wavelet** – (str) name of pywt wavelet, eg 'morl' for Morlet
- **sr** – sample rate in Hz (1/seconds)

Returns (float) scale parameter for pywt.ctw() to extract desired frequency

Return type scale

Note: this function is not exactly an inverse of pywt.scale2frequency(), because that function returns frequency in sample-units (cycles/sample) rather than frequency in Hz (cycles/second). In other words, freuquency_hz = pywt.scale2frequency(w,scale)*sr.

opensoundscape.signal.**thresholded_event_durations**(*x*, *threshold*, *normalize=False*, *sr=1*)

Detect positions and durations of events over threshold in 1D signal

This function takes a 1D numeric vector and searches for segments that are continuously greater than a threshold value. The input signal can optionally be normalized, and if a sample rate is provided the start positions will be in the units of [sr]^-1 (ie if sr is Hz, start positions will be in seconds).

Parameters

- **x** – 1d input signal, a vector of numeric values
- **threshold** – minimum value of signal to be a detection
- **normalize** – if True, performs x=x/max(x)
- **sr** – sample rate of input signal

Returns start time of each detected event durations: duration (# samples/sr) of each detected event

Return type start_times

CHAPTER 16

Misc tools

16.1 Helpers

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

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

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

opensoundscape.helpers.**generate_clip_times_df**(*full_duration*, *clip_duration*, *clip_overlap=0*, *final_clip=None*)
generate start and end times for even-lengthed clips

The behavior for incomplete final clips at the end of the full_duration depends on the final_clip parameter.

This function only creates a dataframe with start and end times, it does not perform any actual trimming of audio or other objects.

Parameters

- **full_duration** – The amount of time (seconds) to split into clips
- **clip_duration** (*float*) – The duration in seconds of the clips
- **clip_overlap** (*float*) – The overlap of the clips in seconds [default: 0]
- **final_clip** (*str*) – Behavior if final_clip is less than clip_duration seconds long. By default, discards remaining time if less than clip_duration seconds long [default: None]. Options:
  - None: Discard the remainder (do not make a clip)
  - "extend": Extend the final clip beyond full_duration to reach clip_duration length
  - "remainder": Use only remainder of full_duration (final clip will be shorter than clip_duration)

– ”full”: Increase overlap with previous clip to yield a clip with clip_duration length

Returns DataFrame with columns for ‘start_time’, ‘end_time’, and ‘clip_duration’ of each clip (which may differ from *clip_duration* argument for final clip only)

Return type clip_df

Note: using “remainder” or “full” with clip_overlap>0 is not recommended. This combination may result in several duplications of the same final clip.

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

Example usage: ‘‘‘ # Get the UTC timestamp t = hex_to_time(‘5F16A04E’)

Convert it to a desired timezone my_timezone = pytz.timezone(“US/Mountain”) t = t.astimezone(my_timezone) ‘‘‘

Parameters **s** (*string*) – hexadecimal Unix epoch time string, e.g. ‘5F16A04E’

Returns datetime.datetime object representing the date and time in UTC

opensoundscape.helpers.**inrange**(*x*, *r*)
return true if x is in range [r[0],r1] (inclusive)

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

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

Parameters

- **x** – scalar, array, or nd-array of numeric type
- **width** – multiplier for random variable (stdev for ‘gaussian’ or r for ‘uniform’)
- **distribution** – ‘gaussian’ (default) or ‘uniform’ if ‘gaussian’: draw jitter from gaussian with mu = 0, std = width if ‘uniform’: draw jitter from uniform on [-width, width]

Returns x + random jitter

Return type jittered_x

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

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

Outputs: new_array: A translated array

opensoundscape.helpers.**make_clip_df**(*files*, *clip_duration*, *clip_overlap=0*, *final_clip=None*)
generate df of fixed-length clip times for a set of file_batch_size

Used internally to prepare a dataframe listing clips of longer audio files

This function creates a single dataframe with audio files as the index and columns: ‘start_time’, ‘end_time’. It will list clips of a fixed duration from the beginning to end of each audio file.

Parameters

- **files** – list of audio file paths
- **clip_duration** (*float*) – see generate_clip_times_df
- **clip_overlap** (*float*) – see generate_clip_times_df
- **final_clip** (*str*) – see generate_clip_times_df

Returns

dataframe with columns 'start_time', 'end_time' and file paths as index

unsafe_samples: list of file paths that did not exist or failed to produce a valid list of clips

Return type clip_df

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

opensoundscape.helpers.**overlap**(*r1*, *r2*)
"calculate the amount of overlap between two real-numbered ranges

opensoundscape.helpers.**overlap_fraction**(*r1*, *r2*)
"calculate the fraction of r1 (low, high range) that overlaps with r2

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

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

returns rescaled feature set and rescaling vector

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

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

16.2 Taxa

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

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

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

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

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

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

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

16.3 Localization

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

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

Parameters **temperature** – ambient temperature in Celsius

Returns the speed of sound in meters per second

opensoundscape.localization.**localize**(*receiver_positions*, *arrival_times*, *temperature=20.0*, *invert_alg='gps'*, *center=True*, *pseudo=True*)

Perform TDOA localization on a sound event

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

Parameters

- **receiver_positions** – a list of [x,y,z] positions for each receiver Positions should be in meters, e.g., the UTM coordinate system.
- **arrival_times** – a list of TDOA times (onset times) for each recorder The times should be in seconds.
- **temperature** – ambient temperature in Celsius
- **invert_alg** – what inversion algorithm to use (only 'gps' is implemented)
- **center** – whether to center recorders before computing localization result. Computes localization relative to centered plot, then translates solution back to original recorder locations. (For behavior of original Sound Finder, use True)
- **pseudo** – whether to use the pseudorange error (True) or sum of squares discrepancy (False) to pick the solution to return (For behavior of original Sound Finder, use False. However, in initial tests, pseudorange error appears to perform better.)

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

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

Compute Lorentz inner product of two vectors

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

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

Or, for 2-dimensional case as

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

Parameters

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

Returns value of Lorentz IP

Return type float

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

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

Parameters

- **source** – cartesian position [x,y] or [x,y,z] of sound source

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

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

CHAPTER 17

Index

CHAPTER 18

Modules

- modindex

Python Module Index

o

Index

Symbols

A

B

C

P

R

S

T

U

V

W

X