opensoundscape

Release 0.5.0

Jun 22, 2021

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 using the pip command `pip install opensoundscape==0.5.0`. 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 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==0.5.0`
- Deactivate the environment when you're done using it: `conda deactivate`

1.2 Installation via `venv`

Download Python 3.7 from this website.

Run the following commands in your bash terminal:

- Check that you have installed Python 3.7._: `python3 --version`
- Change directories to where you wish to store the environment: `cd [path for environments folder]`
  - Tip: You can use this folder to store virtual environments for other projects as well, so put it somewhere that makes sense for you, e.g. in your home directory.
- Make a directory for virtual environments and `cd` into it: `mkdir .venv && cd .venv`
- Create an environment called `opensoundscape` in the directory: `python3 -m venv opensoundscape`
- Activate/use the environment: `source opensoundscape/bin/activate`
- Install OpenSoundscape in the environment: `pip install opensoundscape==0.5.0`
- 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 WSL1 (more information: https://docs.microsoft.com/en-us/windows/wsl/install-win10):

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /
↪all /norestart
```

- 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 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==0.5.0`

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

```
(opensoundscape_environment) username@computername:~$ pip install opensoundscape==0.5.
↪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 via 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

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

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

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

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

If you are using `conda`, install Python 3.7 using `conda install python==3.7`

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

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

3.2 Contribution workflow

3.2.1 Contributing to code

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

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

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

3.2.2 Contributing to documentation

Build the documentation using either poetry or sphinx-build

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

CHAPTER 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.

This tutorial uses an example audio file downloadable with the OpenSoundscape package. To use your own file for the following examples, replace the path in the line below with the absolute path to the file:

[1]:
```
audio_filename = "../../tests/audio/1min.wav"
```

NOTE: If this cell raises an error for you, make sure to replace `"../../tests/audio/1min.wav"` with a path to an audio file on your computer, in quotes.

5.1 Quick start

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

[2]:
```
# 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

[3]:
```
from pathlib import Path

# Settings
image_shape = (224, 224)
image_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 PIL Image
image = spectrogram.to_image(shape=image_shape)

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

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

[4]:
```
Spectrogram.from_audio(Audio.from_file(audio_filename)).to_image(shape=image_shape).
↪save(image_path)
```

Clean up by deleting the spectrogram saved above.

[5]:
```
image_path.unlink()
```

5.2 Audio loading

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

5.2.1 Load .wavs

Load the example audio from file:

[6]:
```
audio_object = Audio.from_file(audio_filename)
```

5.2.2 Load .mp3s

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 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.

[7]:
```
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
```

5.2.4 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.

[8]:
```
audio_object_resample = Audio.from_file(audio_filename, sample_rate=22050)
audio_object_resample.sample_rate
```

[8]:
```
22050
```

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 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**.

[9]:
```
audio_object.save('./my_audio.wav')
```

clean up: delete saved file

[10]:
```
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

[11]:
```
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).

[12]:
```
trimmed = audio_object.trim(0,5)
trimmed.duration()
```

[12]:
```
5.0
```

5.3.5 Split

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

The function returns a list of dictionaries, with each dictionary containing an audio clip and information about the clip.

[13]:
```
#split into 5-second clips with no overlap between adjascent clips
clips_and_info = audio_object.split(clip_duration=5,clip_overlap=0,final_clip=None)

#check the duration of the Audio object in the first returned element
clips_and_info[0]['clip'].duration()
```

[13]:
```
5.0
```

5.3.6 Extend and loop

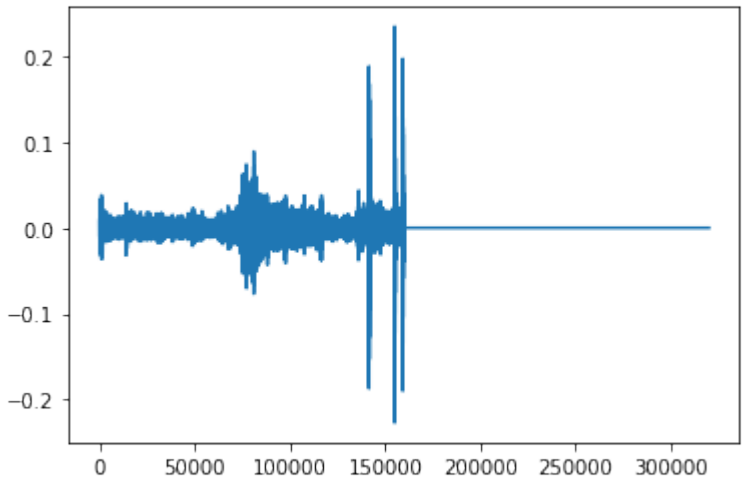
The `.extend()` method extends an audio file to a desired length by adding silence to the end.

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

[14]:
```
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(extended.duration())
plt.plot(extended.samples)
```

```
10.0
```

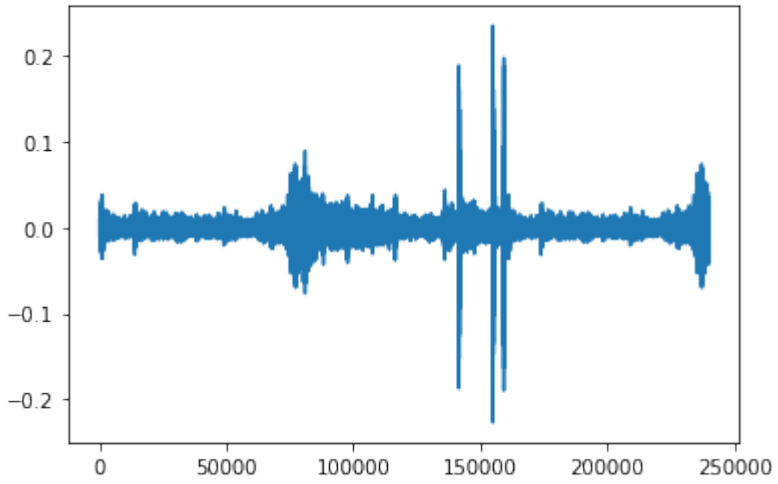
[14]:
```
[<matplotlib.lines.Line2D at 0x7fabff3ea6a0>]
```



[15]:
```
# create an audio object 1.5x as long, extending the end by looping
looped = trimmed.loop(trimmed.duration() * 1.5)
print(looped.duration())
plt.plot(looped.samples)
```

```
7.5
```

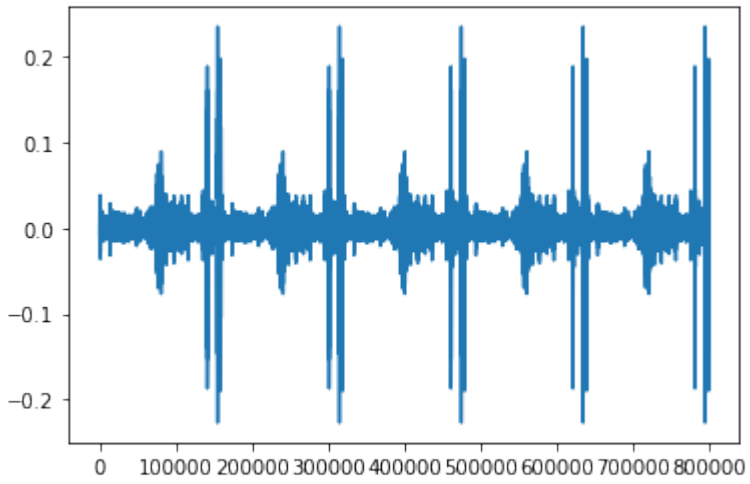
[15]:
```
[<matplotlib.lines.Line2D at 0x7fac00d6e5e0>]
```



[16]:
```
# create an audio object that loops the original object 3 times
looped = trimmed.loop(n=5)
print(looped.duration())
plt.plot(looped.samples)
```

```
25.0
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7fac02a2baf0>]
```



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)

[17]:
```
resampled = audio_object.resample(sample_rate=48000)
resampled.sample_rate
```

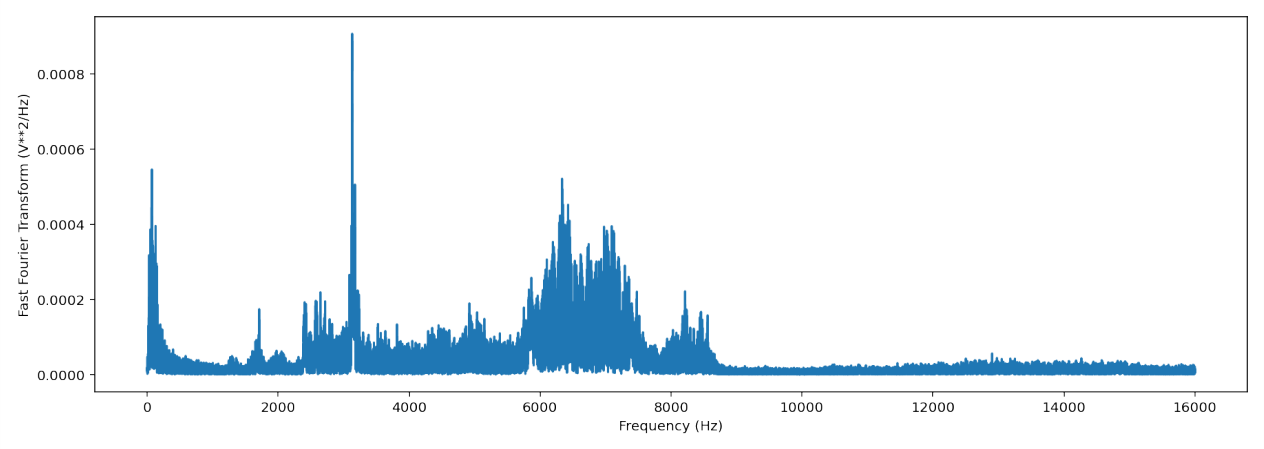
[17]:
```
48000
```

5.3.8 Generate a frequency spectrum

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

[18]:
```
# 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)')
```

[18]:
```
Text(0.5, 0, 'Frequency (Hz)')
```

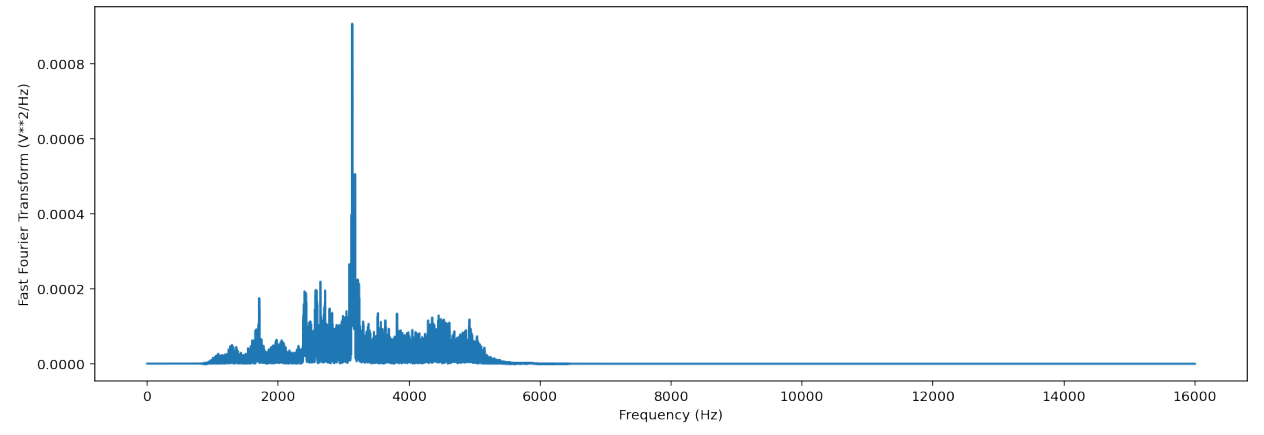


5.3.9 Bandpass

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.

[19]:
```
# apply a bandpass filter
bandpassed = trimmed.bandpass(low_f = 1000, high_f = 5000, order=9)

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

# plot
plt.plot(frequencies,fft_spectrum)
plt.ylabel('Fast Fourier Transform (V**2/Hz)')
plt.xlabel('Frequency (Hz)')
```

[19]:
```
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.

[20]:
```
audio_object = Audio.from_file(audio_filename)
spectrogram_object = Spectrogram.from_audio(audio_object)
```

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

5.4.2 Spectrogram properties

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

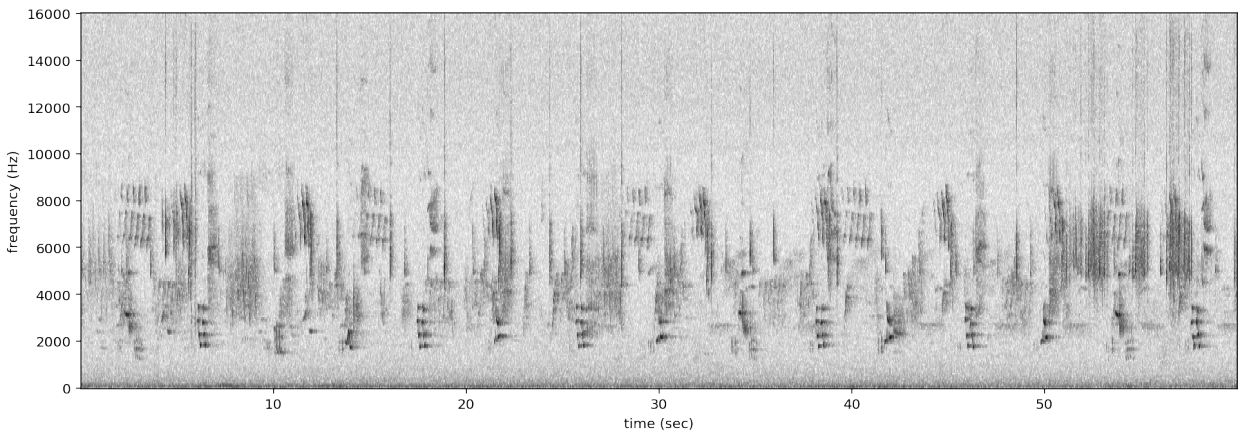
[21]:
```
spec = Spectrogram.from_audio(Audio.from_file(audio_filename))
print(f'the first few times: {spec.times[0:5]}')
print(f'the first few frequencies: {spec.frequencies[0:5]}')
```

```
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.

[22]:
```
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 consecutive 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 columns in the spectrogram per second) * Smaller value for `window_samples` –> higher time resolution (more rows in a single spectrogram column)

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.

Relationship

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).

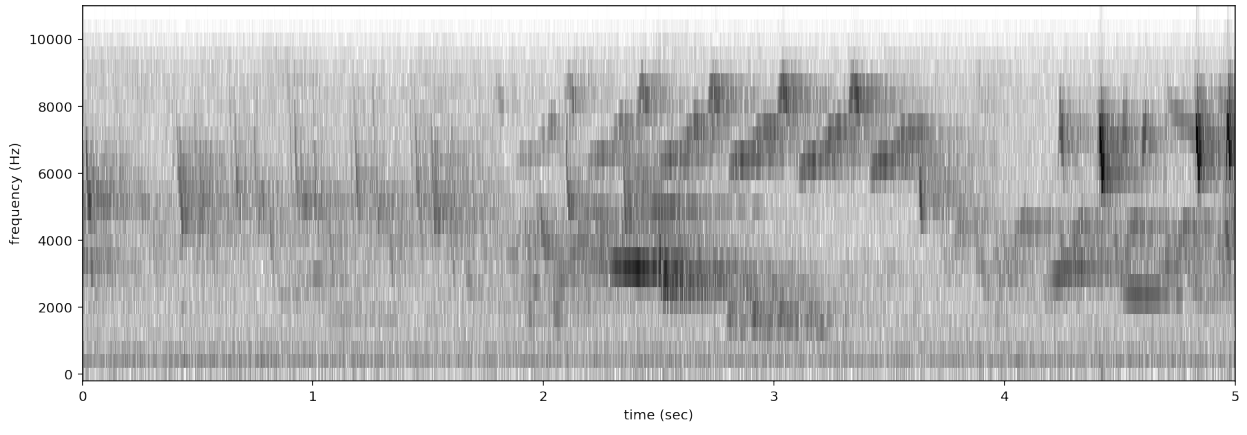
As an example, let's create two spectrograms, one with hight time resolution and another with high frequency resolution.

[23]:
```
# 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.

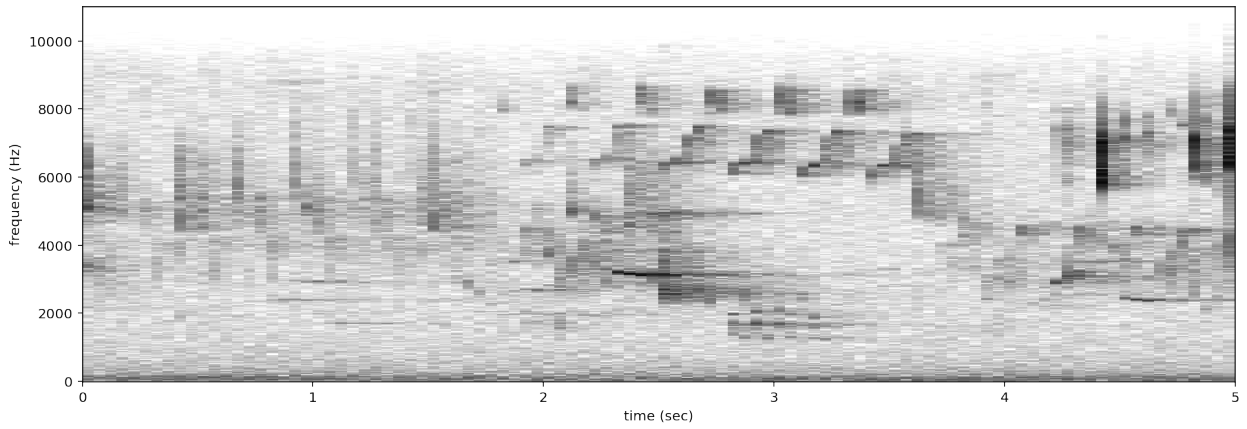
[24]:
```
spec = Spectrogram.from_audio(audio, window_samples=55, overlap_samples=0)
spec.plot()
```



Create a spectrogram with high time 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.

[25]:
```
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.

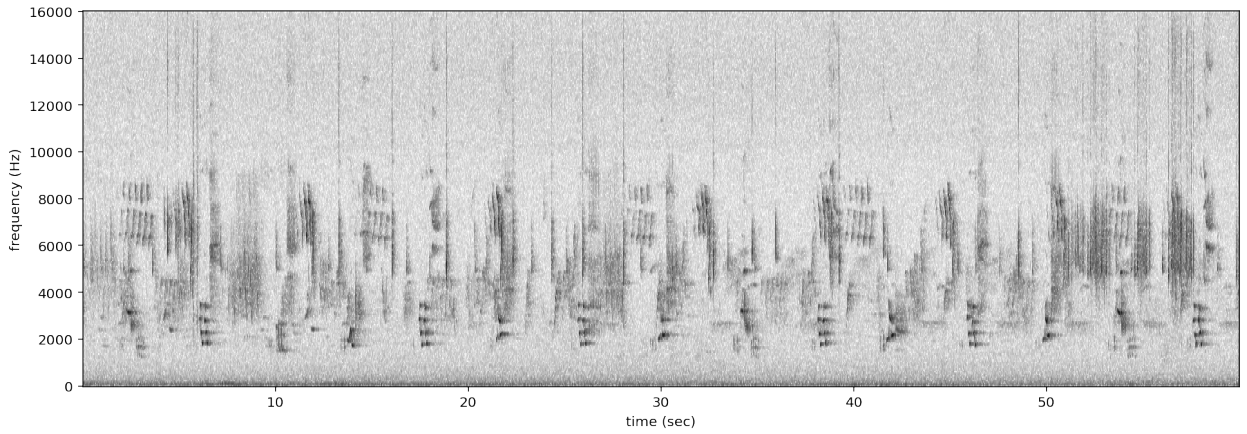
5.5 Spectrogram methods

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.

[26]:
```
audio_object = Audio.from_file(audio_filename)
spectrogram_object = Spectrogram.from_audio(audio_object)
spectrogram_object.plot()
```

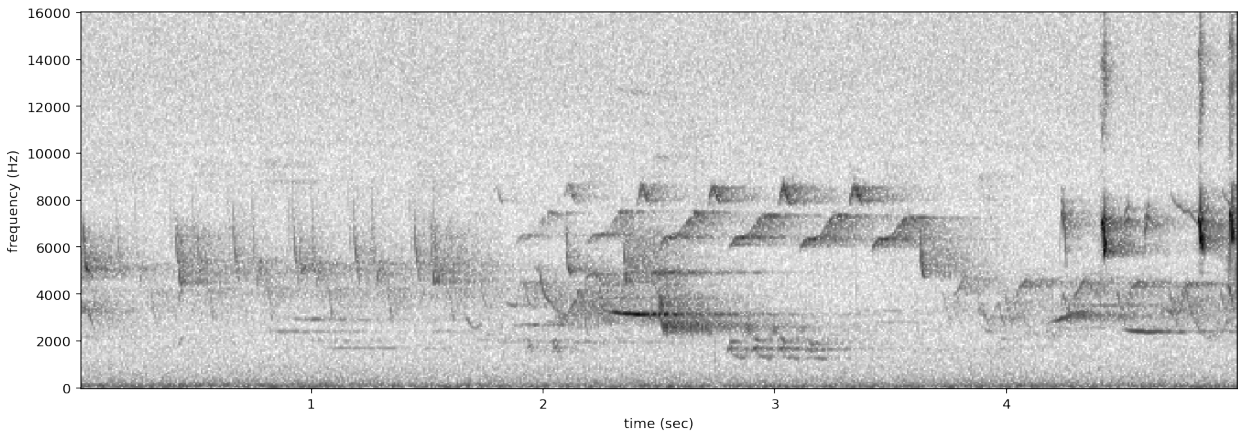


5.5.2 Load modified audio

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.

[27]:
```
# 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.

[28]:
```
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.

[29]:
```
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`.

[30]:
```
image_shape = (512,512)
large_image_path = Path('./saved_spectrogram_large.png')
```

(continues on next page)

(continued from previous page)

```
spectrogram_image = spectrogram_object.to_image(shape=image_shape)
spectrogram_image.save(large_image_path)
```

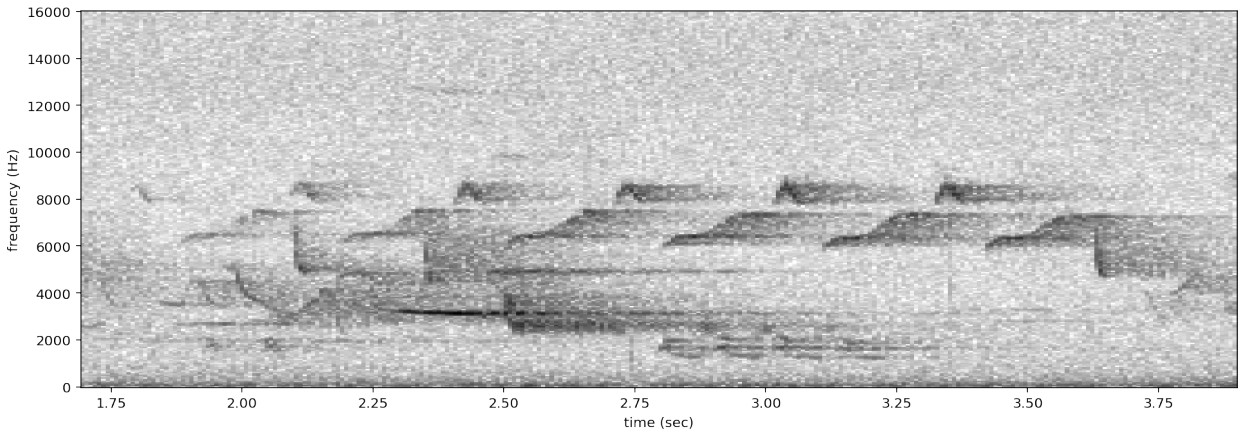
Delete the files created above.

[31]:
```
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.

[32]:
```
spec_trimmed = spec.trim(1.7, 3.9)
spec_trimmed.plot()
```



5.5.5 Bandpass

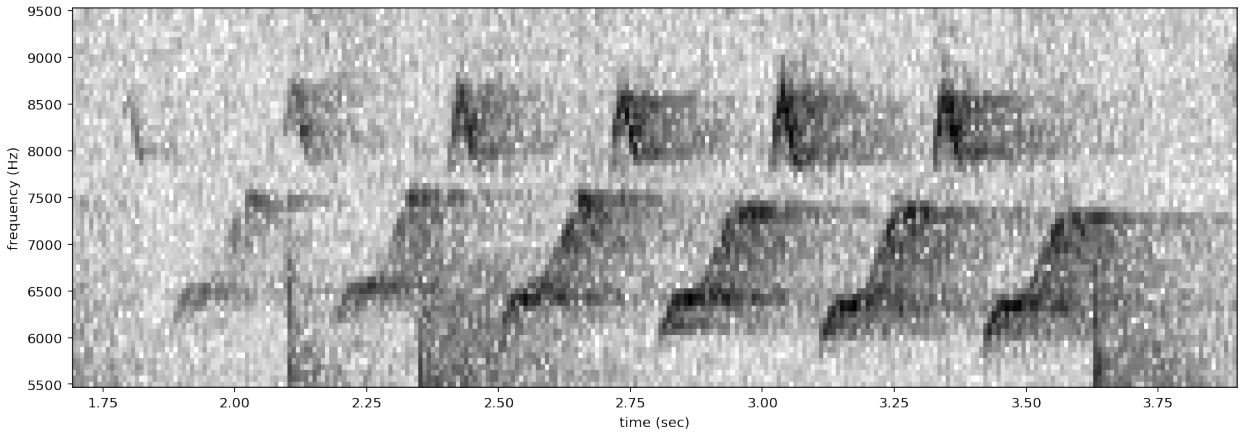
Spectrograms can be trimmed in frequency using `bandpass()`. This simply subsets the Spectrogram array rather than performing an audio-domain filter.

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.

[33]:
```
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.

[34]:
```
spec_bandpassed = spec_trimmed.bandpass(baww_low_freq, baww_high_freq)
spec_bandpassed.plot()
```

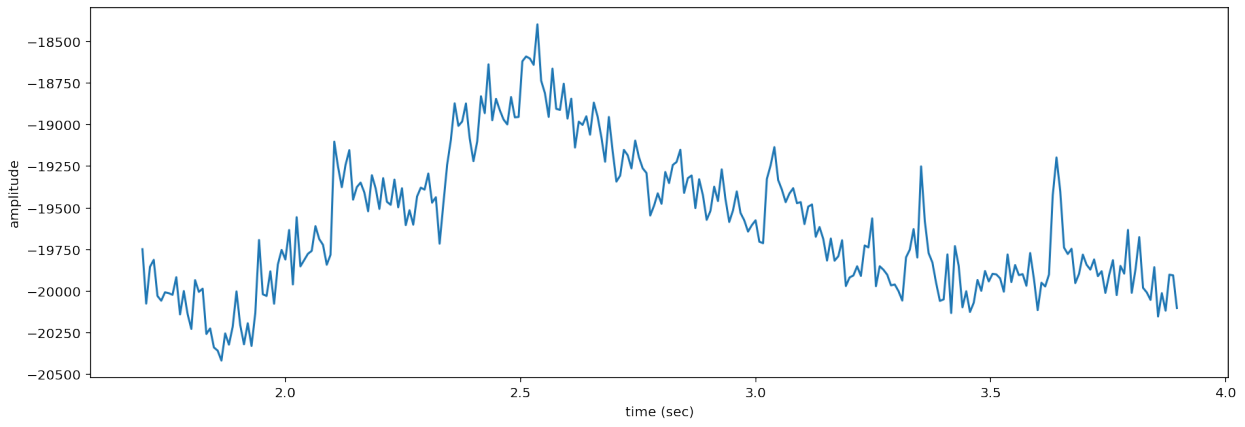


5.5.6 Calculate amplitude signal

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

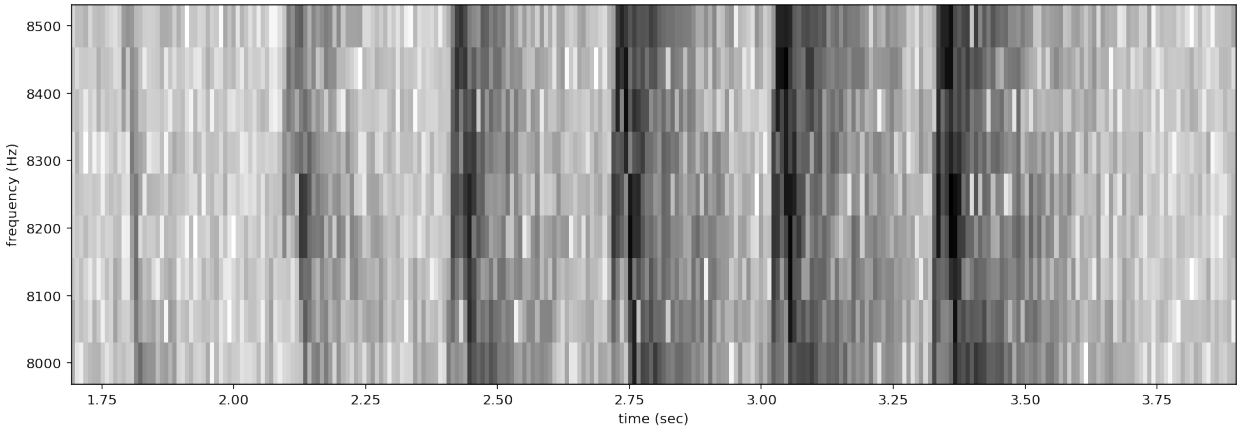
Note: the amplitude of the Spectrogram (and FFT) has units of power (V**2) over frequency (Hz)

[35]:
```
# 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()
```



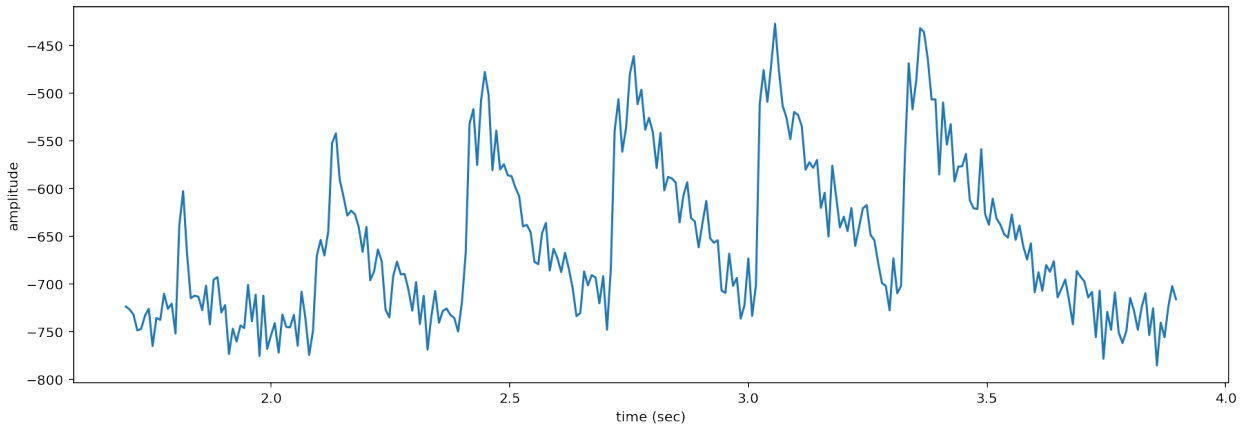
It is also possible to get the amplitude signal from a restricted range of frequencies, for instance, to look at the amplitude in the frequency range of a species of interest. For example, get the amplitude signal from the 8000 Hz to 8500 Hz range of the audio (displayed below):

[36]:
```
spec_bandpassed = spec_trimmed.bandpass(8000, 8500)
spec_bandpassed.plot()
```



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

[37]:
```
# 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 pretrained models" tutorial.

CHAPTER 6

Raven annotations

Raven Sound Analysis Software enables users to inspect spectrograms, draw time and frequency boxes around sounds of interest, and label these boxes with species identities. OpenSoundscape contains functionality to prepare and use these annotations for machine learning.

6.1 Download annotated data

We published an example Raven-annotated dataset here: https://doi.org/10.1002/ecy.3329

[1]:
```
from opensoundscape.commands import run_command
from pathlib import Path
```

Download the zipped data here:

[2]:
```
link = "https://esajournals.onlinelibrary.wiley.com/action/downloadSupplement?doi=10.
↪1002%2Fecy.3329&file=ecy3329-sup-0001-DataS1.zip"
name = 'powdermill_data.zip'
out = run_command(f"wget -O powdermill_data.zip {link}")
```

Unzip the files to a new directory, `powdermill_data/`

[3]:
```
out = run_command("unzip powdermill_data.zip -d powdermill_data")
```

Keep track of the files we have now so we can delete them later.

[4]:
```
files_to_delete = [Path("powdermill_data"), Path("powdermill_data.zip")]
```

6.2 Preprocess Raven data

The `opensoundscape.raven` module contains preprocessing functions for Raven data, including: * `annotation_check` - for all the selections files, make sure they all contain labels * `lowercase_annotations`

- lowercase all of the annotations * `generate_class_corrections` - create a CSV to see whether there are any weird names * Modify the CSV as needed. If you need to look up files you can use `query_annotations` * Can be used in `SplitterDataset` * `apply_class_corrections` - replace incorrect labels with correct labels * `query_annotations` - look for files that contain a particular species or a typo

[5]:
```
import pandas as pd
import opensoundscape.raven as raven
import opensoundscape.audio as audio
```

[6]:
```
raven_files_raw = Path("./powdermill_data/Annotation_Files/")
```

6.2.1 Check Raven files have labels

Check that all selections files contain labels under one column name. In this dataset the labels column is named `"species"`.

[7]:
```
raven.annotation_check(directory=raven_files_raw, col='species')
```

```
All rows in powdermill_data/Annotation_Files contain labels in column `species`
```

6.2.2 Create lowercase files

Convert all the text in the files to lowercase to standardize them. Save these to a new directory. They will be saved with the same filename but with ".lower" appended.

[8]:
```
raven_directory = Path('./powdermill_data/Annotation_Files_Standardized')
if not raven_directory.exists(): raven_directory.mkdir()
raven.lowercase_annotations(directory=raven_files_raw, out_dir=raven_directory)
```

Check that the outputs are saved as expected.

[9]:
```
list(raven_directory.glob("*.lower"))[:5]
```

[9]:
```
[PosixPath('powdermill_data/Annotation_Files_Standardized/Recording_1_Segment_22.
↪Table.1.selections.txt.lower'),
 PosixPath('powdermill_data/Annotation_Files_Standardized/Recording_4_Segment_15.
↪Table.1.selections.txt.lower'),
 PosixPath('powdermill_data/Annotation_Files_Standardized/Recording_4_Segment_24.
↪Table.1.selections.txt.lower'),
 PosixPath('powdermill_data/Annotation_Files_Standardized/Recording_1_Segment_13.
↪Table.1.selections.txt.lower'),
 PosixPath('powdermill_data/Annotation_Files_Standardized/Recording_1_Segment_06.
↪Table.1.selections.txt.lower')]
```

6.2.3 Generate class corrections

This function generates a table that can be modified by hand to correct labels with typos in them. It identifies the unique labels in the provided column (here `"species"`) in all of the lowercase files in the directory `raven_directory`.

For instance, the generated table could be something like the following:

```
raw,corrected
sparrow,sparrow
sparow,sparow
goose,goose
```

[10]:
```
print(raven.generate_class_corrections(directory=raven_directory, col='species'))
```

```
raw,corrected
amcr,amcr
amgo,amgo
amre,amre
amro,amro
baor,baor
baww,baww
bbwa,bbwa
bcch,bcch
bggn,bggn
bhco,bhco
bhvi,bhvi
blja,blja
brcr,brcr
btnw,btnw
bwwa,bwwa
cang,cang
carw,carw
cedw,cedw
cora,cora
coye,coye
cswa,cswa
dowo,dowo
eato,eato
eawp,eawp
hawo,hawo
heth,heth
howa,howa
kewa,kewa
lowa,lowa
nawa,nawa
noca,noca
nofl,nofl
oven,oven
piwo,piwo
rbgr,rbgr
rbwo,rbwo
rcki,rcki
revi,revi
rsha,rsha
rwbl,rwbl
scta,scta
swth,swth
tuti,tuti
veer,veer
wbnu,wbnu
witu,witu
woth,woth
ybcu,ybcu
```

The released dataset has no need for class corrections, but if it did, we could save the return text to a CSV and use the CSV to apply corrections to future dataframes.

6.2.4 Query annotations

This function can be used to print all annotations of a particular class, e.g. "amro" (American Robin)

[11]:
```
output = raven.query_annotations(directory=raven_directory, cls='amro', col='species',
↪ print_out=True)
```

```
================================================================================================
powdermill_data/Annotation_Files_Standardized/Recording_4_Segment_16.Table.1.
↪selections.txt.lower
================================================================================================

     selection           view  channel  begin time (s)  end time (s)  \
85          86  spectrogram 1        1       77.634876     82.129659
93          94  spectrogram 1        1       84.226733     86.313096
98          99  spectrogram 1        1       88.825438     91.272182
107        108  spectrogram 1        1       96.028977     97.552840
111        112  spectrogram 1        1       99.990354    100.914517
116        117  spectrogram 1        1      104.327755    108.656087
122        123  spectrogram 1        1      109.525937    112.021391
129        130  spectrogram 1        1      113.765766    117.386474
137        138  spectrogram 1        1      121.053454    121.383161
141        142  spectrogram 1        1      124.864220    129.139630
154        155  spectrogram 1        1      132.583749    135.017840
162        163  spectrogram 1        1      139.602300    142.087527
168        169  spectrogram 1        1      143.969913    146.785822
176        177  spectrogram 1        1      149.282840    151.873748
210        211  spectrogram 1        1      170.636021    174.123521
225        226  spectrogram 1        1      178.252401    181.670619
238        239  spectrogram 1        1      184.176135    188.110226
250        251  spectrogram 1        1      190.244089    192.858862
267        268  spectrogram 1        1      203.737856    204.958310
277        278  spectrogram 1        1      211.662233    216.270763

     low freq (hz)  high freq (hz) species
85          1539.7          3668.7    amro
93          1349.6          3630.6    amro
98          1539.7          4029.8    amro
107         1159.5          3573.6    amro
111         1539.7          3440.4    amro
116         1368.6          3041.4    amro
122         1577.7          3041.4    amro
129         1602.9          3831.4    amro
137         1993.9          2813.1    amro
141         1558.7          4200.9    amro
154         2186.0          3782.7    amro
162         1634.7          4200.9    amro
168         1748.8          3687.7    amro
176         1634.7          3744.7    amro
210         1444.7          4162.9    amro
225         1798.4          3831.4    amro
238         1653.7          3592.6    amro
250         1615.7          3687.7    amro
267         1563.1          4230.8    amro
```

(continues on next page)

(continued from previous page)

```
277          1646.5           4189.1     amro

================================================================================================
powdermill_data/Annotation_Files_Standardized/Recording_4_Segment_01.Table.1.
↪selections.txt.lower
================================================================================================

     selection           view  channel  begin time (s)  end time (s)  \
188        189  spectrogram 1        1      247.263069    249.107387
201        202  spectrogram 1        1      263.512160    264.851933

     low freq (hz)  high freq (hz) species
188         1249.2          2419.2    amro
201         1229.4          2558.0    amro
```

6.3 Split Raven annotations and audio files

The Raven module's `raven_audio_split_and_save` function enables splitting of both audio data and associated annotations. It requires that the annotation and audio filenames are unique, and that corresponding annotation and audiofilenames are named the same filenames as each other.

[12]:
```
audio_directory = Path('./powdermill_data/Recordings/')
destination = Path('./powdermill_data/Split_Recordings')
out = raven.raven_audio_split_and_save(

    # Where to look for Raven files
    raven_directory = raven_directory,

    # Where to look for audio files
    audio_directory = audio_directory,

    # The destination to save clips and the labels CSV to
    destination = destination,

    # The column name of the labels
    col = 'species',

    # Desired audio sample rate
    sample_rate = 22050,

    # Desired duration of clips
    clip_duration = 5,

    # Verbose (uncomment the next line to see progress--this cell takes a while to␣
↪run)
    #verbose=True,
)
```

```
Found 77 sets of matching audio files and selection tables out of 77 audio files and␣
↪77 selection tables
```

The results of the splitting are saved in the destination folder under the name `labels.csv`.

[13]:
```
labels = pd.read_csv(destination.joinpath("labels.csv"), index_col='filename')
labels.head()
```

[13]:

filename	amcr	amgo	amre	amro	baor	baww	bbwa	bcch	bggn	bhco	...	rsha	rwbl	scta	swth	tuti	veer	wbnu	witu	woth	ybcu
powdermill_data/Split_Recordings/Recording_4_Se...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
powdermill_data/Split_Recordings/Recording_4_Se...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
powdermill_data/Split_Recordings/Recording_4_Se...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
powdermill_data/Split_Recordings/Recording_4_Se...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
powdermill_data/Split_Recordings/Recording_4_Se...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

[5 rows x 48 columns]

The `raven_audio_split_and_save` function contains several options. Notable options are: * `clip_duration`: the length of the clips * `clip_overlap`: the overlap, in seconds, between clips * `final_clip`: what to do with the final clip if it is not exactly `clip_duration` in length (see API docs for more details) * `labeled_clips_only`: whether to only save labeled clips * `min_label_length`: minimum length, in seconds, of an annotation for a clip to be considered labeled. For instance, if an annotation only overlaps 0.1s

with a 5s clip, you might want to exclude it with `min_label_length=0.2`. * `species`: a subset of species to search for labels of (by default, finds all species labels in dataset) * `dry_run`: if `True`, produces print statements and returns dataframe of labels, but does not save files. * `verbose`: if `True`, prints more information, e.g. clip-by-clip progress.

For instance, let's extract labels for one species, American Redstart (AMRE) only saving clips that contain at least 0.5s of label for that species. The "verbose" flag causes the function to print progress splitting each clip.

[14]:
```
btnw_split_dir = Path('./powdermill_data/btnw_recordings')
out = raven.raven_audio_split_and_save(
    raven_directory = raven_directory,
    audio_directory = audio_directory,
    destination = btnw_split_dir,
    col = 'species',
    sample_rate = 22050,
    clip_duration = 5,
    clip_overlap = 0,
    verbose=True,
    species='amre',
    labeled_clips_only=True,
    min_label_len=1
)
```

```
Found 77 sets of matching audio files and selection tables out of 77 audio files and␣
↪77 selection tables
Making directory powdermill_data/btnw_recordings
1. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_13.mp3
2. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_33.mp3
3. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_26.mp3
4. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_19.mp3
5. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_11.mp3
6. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_13.mp3
7. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_29.mp3
8. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_01.mp3
9. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_15.mp3
10. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_20.mp3
11. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_12.mp3
12. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_36.mp3
13. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_25.mp3
14. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_26.mp3
15. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_14.mp3
16. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_10.mp3
17. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_11.mp3
18. Finished powdermill_data/Recordings/Recording_3/Recording_3_Segment_01.mp3
19. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_32.mp3
20. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_03.mp3
21. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_07.mp3
22. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_04.mp3
23. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_16.mp3
24. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_30.mp3
25. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_02.mp3
26. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_19.mp3
27. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_12.mp3
28. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_08.mp3
29. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_10.mp3
30. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_20.mp3
31. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_12.mp3
32. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_14.mp3
```

(continues on next page)

(continued from previous page)

```
33. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_16.mp3
34. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_25.mp3
35. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_09.mp3
36. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_17.mp3
37. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_07.mp3
38. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_02.mp3
39. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_02.mp3
40. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_08.mp3
41. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_09.mp3
42. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_05.mp3
43. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_08.mp3
44. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_05.mp3
45. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_18.mp3
46. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_14.mp3
47. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_09.mp3
48. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_23.mp3
49. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_06.mp3
50. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_34.mp3
51. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_10.mp3
52. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_27.mp3
53. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_06.mp3
54. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_31.mp3
55. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_04.mp3
56. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_24.mp3
57. Finished powdermill_data/Recordings/Recording_2/Recording_2_Segment_05.mp3
58. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_22.mp3
59. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_18.mp3
60. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_01.mp3
61. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_21.mp3
62. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_24.mp3
63. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_03.mp3
64. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_01.mp3
65. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_04.mp3
66. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_15.mp3
67. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_13.mp3
68. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_07.mp3
69. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_11.mp3
70. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_21.mp3
71. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_28.mp3
72. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_17.mp3
73. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_03.mp3
74. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_35.mp3
75. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_06.mp3
76. Finished powdermill_data/Recordings/Recording_4/Recording_4_Segment_23.mp3
77. Finished powdermill_data/Recordings/Recording_1/Recording_1_Segment_22.mp3
```

The labels CSV only has a column for the species of interest:

[15]:
```
btnw_labels = pd.read_csv(btnw_split_dir.joinpath("labels.csv"), index_col='filename')
btnw_labels.head()
```

[15]:

filename	amre
powdermill_data/btnw_recordings/Recording_2_Seg...	1.0
powdermill_data/btnw_recordings/Recording_2_Seg...	1.0
powdermill_data/btnw_recordings/Recording_2_Seg...	1.0
powdermill_data/btnw_recordings/Recording_2_Seg...	1.0

(continues on next page)

(continued from previous page)

```
powdermill_data/btnw_recordings/Recording_2_Seg...   1.0
```

The split files and associated labels csv can now be used to train machine learning models (see additional tutorials).

The command below cleans up after the tutorial is done – only run it if you want to delete all of the files.

[16]:
```
from shutil import rmtree
for file in files_to_delete:
    if file.is_dir():
        rmtree(file)
    else:
        file.unlink()
```

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 be found in the other tutorial notebooks on opensoundscape.org.

7.1 Load required packages

We will load several imports from OpenSoundscape. First, load the `AudiotoSpectrogramPreprocessor` class from the `preprocess.preprocessors` module. Preprocessor classes are used to load, transform, and augment audio samples for use in a machine learning model.

[1]:
```
from opensoundscape.preprocess.preprocessors import AudioToSpectrogramPreprocessor
```

Second, the `cnn` module provides classes for training and prediction with various structures of CNNs. For this example, load the `Resnet18Binary` class, used for models made with the Resnet18 architecture for predicting the presence or absence of a species (a "binary" classifier).

[2]:
```
# The cnn module provides classes for training/predicting with various types of CNNs
from opensoundscape.torch.models.cnn import Resnet18Binary
```

Third, the `run_command` function from the `helpers` module allows us to run command line commands from inside OpenSoundscape scripts.

[3]:
```
from opensoundscape.helpers import run_command
```

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

[4]:
```
# Other utilities and packages
import torch
from pathlib import Path
import numpy as np
import pandas as pd
from glob import glob
```

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

7.2 Prepare audio data for prediction

To run predictions on your audio data, you will need to have your audio already split up into the clip lengths that the model expects to receive. If your audio data are not already split, see the demonstration of the `Audio.split()` method in the `audio_and_spectrogram` notebook.

You can check the length of clips that the model to receives in the model's notes when you download it. This is often, but not always, 5.0 seconds.

7.2.1 Download 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 `curl` and `tar` installed on your computer, as they will download and unzip a compressed file in `.tar.gz` format.
2. OR 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.

[6]:
```
commands = [
    "curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -
↪o ./woodcock_labeled_data.tar.gz",
    "tar -xzf woodcock_labeled_data.tar.gz", # Unzip the downloaded tar.gz file
    "rm woodcock_labeled_data.tar.gz" # Remove the file after its contents are␣
↪unzipped
]
for command in commands:
    run_command(command)
```

7.2.2 Generate a Preprocessor object

In addition to having audio clips of the correct length, you will need to create a Preprocessor object that loads audio samples for the CNN.

First, generate a Pandas DataFrame with the index containing the paths to each file, as shown below.

[7]:
```
# collect a list of audio files
file_list = glob('./woodcock_labeled_data/*.wav')

# create a DataFrame with the audio files as the index
audio_file_df = pd.DataFrame(index=file_list)
```

Next, use that DataFrame to create a Preprocessor object suitable for your application. Use the argument `return_labels=False`, as our audio to predict on does not have labels.

If the model was trained with any special preprocesor settings, you should apply those settings here. For pretrained models created by the Kitzes Lab, see the model's notes from its download page for the exact code to use here.

[8]:
```
# create a Preprocessor object
# we use the option "return_labels=False" because our audio to predict on does not␣
↪have labels
from opensoundscape.preprocess.preprocessors import AudioToSpectrogramPreprocessor
prediction_dataset = AudioToSpectrogramPreprocessor(audio_file_df, return_
↪labels=False)
```

7.3 Models trained with OpenSoundscape v0.5.x

Check the model notes page for the appropriate model class to use and import the correct class from the `cnn` module.

[9]:
```
from opensoundscape.torch.models.cnn import Resnet18Binary
```

For the purpose of demonstration, let's generate a new Resnet18 model for binary prediction and save it to our local folder.

If you download a pre-trained model, you can skip this cell.

[10]:
```
model = Resnet18Binary(classes=['absent','present'])
model.save('./demo.model')
```

```
created PytorchModel model object with 2 classes
Saving to demo.model
```

Next, provide the model class's `from_checkpoint()` method with the path to your downloaded model.

[11]:
```
# load the model into the appropriate model class
model = Resnet18Binary.from_checkpoint('./demo.model')
```

```
created PytorchModel model object with 2 classes
loading weights from saved object
```

Generate predictions as follows. The `predict` method returns three arguments: scores, thresholded predictions, and labels. For unthresholded prediction on unlabeled data, only the first one is relevant, so discard the other returns are discarded using `scores, _, _`.

[12]:
```
# call model.predict() with the Preprocessor to generate predictions
scores, _, _ = model.predict(prediction_dataset)
```

```
(29, 2)
```

Look at the scores of the first 5 samples.

[13]:
```
#look at the scores of the first 5 samples
scores.head()
```

[13]:
```
                                                    absent   present
./woodcock_labeled_data/d4c40b6066b489518f8da83...  0.331394  0.815523
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...  0.440169  0.262892
./woodcock_labeled_data/79678c979ebb880d5ed6d56...  0.512454  0.545183
./woodcock_labeled_data/49890077267b569e142440f...  0.282791  0.827620
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...  0.194547  0.753871
```

7.3.1 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, let's generate scores using the following settings: * a softmax layer, to make the prediction scores for both classes sum to 1 * a logit layer, to map the prediction scores from [0, 1] to (-inf, +inf)

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'`)

[14]:
```
scores, binary_predictions, _ = model.predict(
    prediction_dataset,
    activation_layer='softmax_and_logit',
    binary_preds='single_target'
)
```

```
(29, 2)
```

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

[15]:
```
scores.head(2)
```

[15]:
```
                                                    absent   present
./woodcock_labeled_data/d4c40b6066b489518f8da83... -0.484129  0.484129
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...  0.177276 -0.177276
```

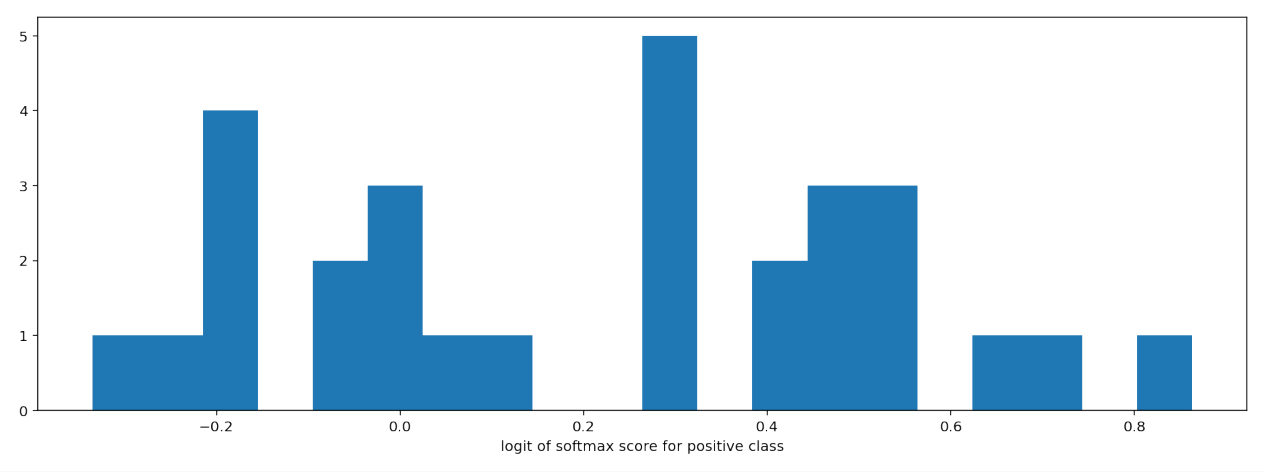
We also have an additional output, the binary 0/1 ("absent" vs "present") predictions generated by the model:

[16]:
```
binary_predictions.head(2)
```

[16]:
```
                                                  absent  present
./woodcock_labeled_data/d4c40b6066b489518f8da83...     0.0      1.0
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...     1.0      0.0
```

It is often helpful to look at a histogram of the scores for the positive class. We typically apply softmax and logit to predictions before plotting them as histograms.

[17]:
```
_ = plt.hist(scores['present'],bins=20)
_ = plt.xlabel('logit of softmax score for positive class')
```



7.4 Models trained with OpenSoundscape 0.4.x

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`

[18]:
```
%%bash
curl -L https://pitt.box.com/shared/static/lglpty35omjhmq6cdz8cfudm43nn2t9f.tar -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      7      0 --:--:--  0:00:01 --:--:--     0
100 42.9M  100 42.9M    0     0  4580k      0  0:00:09  0:00:09 --:--:-- 6184k
```

Next, load the weights from that model into an OpenSoundscape model object with the following code:

[19]:
```
from opensoundscape.torch.models.cnn import PytorchModel
from opensoundscape.torch.architectures.cnn_architectures import resnet18
import torch

# load the tar file into a dictionary
# (you could change this to the location of any .tar file on your computer)
opso_04_model_tar_path = "./opso_04_model_acanthis-flammea.tar"
opso_04_model_dict = torch.load(opso_04_model_tar_path)

# create a resnet18 binary model
# (all models created with Opensoundscape 0.4.x are 2-class resnet18 architectures)
architecture = resnet18(num_classes=2,use_pretrained=False)
model = PytorchModel(classes=['negative','positive'],architecture=architecture)

# load the model weights into our model object
# now, our model is equivalent to the trained model we downloaded
model.network.load_state_dict(opso_04_model_dict['model_state_dict'])
```

```
created PytorchModel model object with 2 classes
```

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

Now, we can use the model as normal to create predictions on audio. We'll use the same `prediction_dataset` from above.

Remember to choose the `activation_layer` you desire. In this example, we'll assume we just want to generate scores, not binary predictions. We'll apply a softmax layer, then the logit transform, to the scores using the `activation_layer="softmax_and_logit"` option. This will generate the type of scores that are useful for plotting score histograms, among other things.

[20]:
```
# generate predictions on our dataset
predition_scores_df,_,_ = model.predict(prediction_dataset, activation_layer='softmax_
↪and_logit')

predition_scores_df.head(3)
```

```
(29, 2)
```

[20]:
```
                                                  negative  positive
./woodcock_labeled_data/d4c40b6066b489518f8da83...  3.953401 -3.953400
./woodcock_labeled_data/e84a4b60a4f2d049d73162e...  3.650462 -3.650461
./woodcock_labeled_data/79678c979ebb880d5ed6d56...  2.165398 -2.165398
```

Remove the downloaded files to clean up.

[22]:
```
folder = Path('./woodcock_labeled_data')
[p.unlink() for p in folder.glob("*")]
folder.rmdir()
for p in Path('.').glob('*.model'):
    p.unlink()
for p in Path('.').glob('*.tar'):
    p.unlink()
```

CHAPTER 8

Basic 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 two-dimensional frequency vs. time representations 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
- making predictions using CNNs

Under the hood, OpenSoundscape uses Pytorch for machine learning tasks. By using OpenSoundscape's CNN classes such as `Resnet18Multiclass` in combination with preprocessor classes such as `CNNPreprocessor`, you can train and predict with PyTorch's powerful CNN architectures.

First, let's import some utilities.

[1]:
```
# Preprocessor classes are used to load, transform, and augment audio samples for use␣
↪in a machine learing model
from opensoundscape.preprocess.preprocessors import BasePreprocessor,␣
↪AudioToSpectrogramPreprocessor

# the cnn module provides classes for training/predicting with various types of CNNs
from opensoundscape.torch.models.cnn import Resnet18Multiclass, Resnet18Binary

#other utilities and packages
from opensoundscape.helpers import run_command
import torch
import pandas as pd
from pathlib import Path
import numpy as np
import pandas as pd
import random

#set up plotting
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

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

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

[2]:
```
torch.manual_seed(0)
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 `curl` and `tar` installed on your computer, as they will download and unzip a compressed file in `.tar.gz` format.
2. Download a `.zip` version of the files by clicking here. You will have to unzip this folder and place the unzipped folder in the same folder that this notebook is in.

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

[3]:
```
commands = [
    "curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -
↪o ./woodcock_labeled_data.tar.gz",
    "tar -xzf woodcock_labeled_data.tar.gz", # Unzip the downloaded tar.gz file
    "rm woodcock_labeled_data.tar.gz" # Remove the file after its contents are␣
↪unzipped
]
for command in commands:
    run_command(command)
```

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
labels = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))
labels.head()
```

[4]:

	filename	woodcock	sound_type
0	d4c40b6066b489518f8da83af1ee4984.wav	present	song
1	e84a4b60a4f2d049d73162ee99a7ead8.wav	absent	na
2	79678c979ebb880d5ed6d56f26ba69ff.wav	present	song

(continues on next page)

(continued from previous page)

```
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
labels.filename = ['./woodcock_labeled_data/'+f for f in labels.filename]
labels.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 manipulate the label dataframe to give "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.

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]:
```
#generate "one-hot" labels
labels['negative']=[0 if label=='present' else 1 for label in labels['woodcock']]
labels['positive']=[1 if label=='present' else 0 for label in labels['woodcock']]
```

Finally, format the dataframe in a consistent way: use `'filename'` as the "index" column, and keep only the one hot label columns. Preprocessor classes require this formatting (see below for more information).

[7]:
```
#use the file path as the index, and class names as the only columns
labels = labels.set_index('filename')[['negative','positive']]
labels.head()
```

[7]:

filename	negative	positive
./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

8.1.3 Split into training and validation sets

We simply use a utility from `sklearn` to randomly divide the labeled samples between 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 these samples once every "epoch" for several epochs–often 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,valid_df = train_test_split(labels,test_size=0.2,random_state=1)
```

8.1.4 Create preprocessors for training and validation

Preprocessors in OpenSoundscape can be used to process audio data, especially for training and prediction with convolutional neural networks.

To train a CNN, we use `CnnPreprocessor`, which loads audio files, creates spectrograms, performs various augmentations to the spectrograms, and returns a pytorch Tensor to be used in training or prediction. All of the steps in the preprocessing pipeline can be modified or skipped by modifying the preprocessor's `.actions`. For details on how to modify and customize a preprocessor, see the `preprocessing` notebook/tutorial.

Each Preprocessor must be initialized with a very specific dataframe with the following attributes: - 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.

The `train_df` and `valid_df` we created above meet these needs:

[9]:
```
train_df.head()
```

[9]:

filename	negative	positive
./woodcock_labeled_data/49890077267b569e142440f...	0	1
./woodcock_labeled_data/ad90eefb6196ca83f9cf43b...	0	1
./woodcock_labeled_data/e9e7153d11de3ac8fc3f716...	0	1
./woodcock_labeled_data/c057a4486b25cd638850fc0...	0	1
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...	0	1

Create a separate preprocessor for training and for validation. These data will be assessed separately each epoch, as described above.

[10]:
```
from opensoundscape.preprocess.preprocessors import CnnPreprocessor

train_dataset = CnnPreprocessor(train_df)

valid_dataset = CnnPreprocessor(valid_df)
```

8.1.5 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.

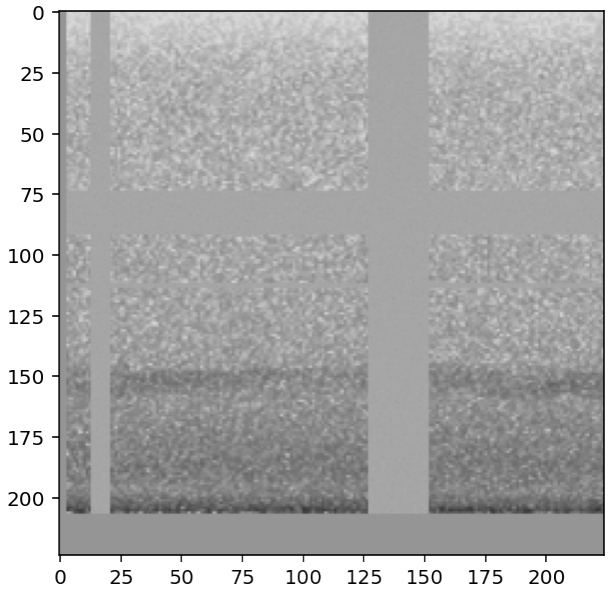
First, in order to view the images, we need a helper function that correctly displays the Tensor that comes out of the Preprocessor.

[11]:
```
# helper function for displaying a sample as an image
def show_tensor(sample):
    plt.imshow((sample['X'][0,:,:]/2+0.5)*-1,cmap='Greys',vmin=-1,vmax=0)
    plt.show()
```

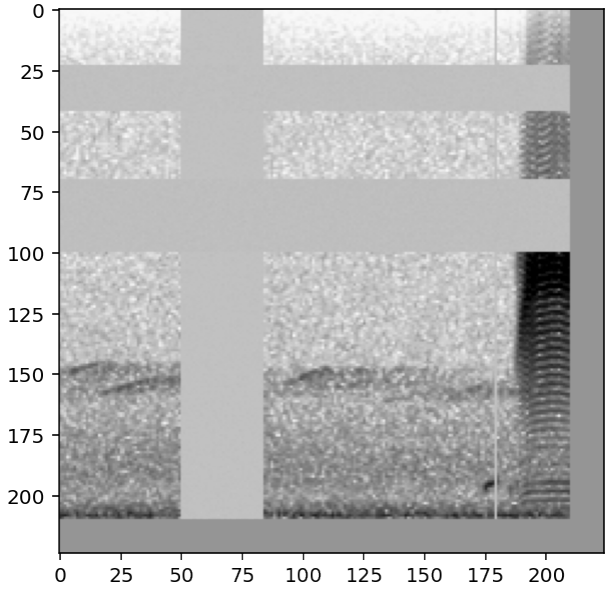
Now, load a handful of random samples, printing the labels and image for each:

[12]:
```
for i, d in enumerate(train_dataset.sample(n=4)):
    print(f"labels: {d['y']}")
    show_tensor(d)
```

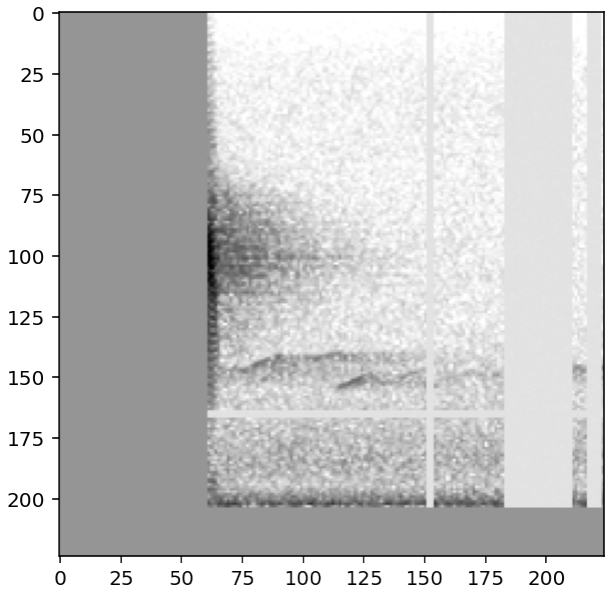
```
labels: tensor([1, 0])
```



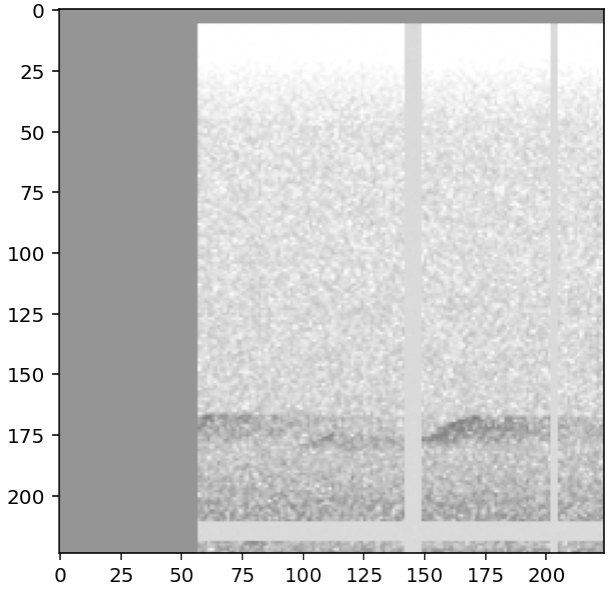

```
labels: tensor([0, 1])
```




```
labels: tensor([0, 1])
```



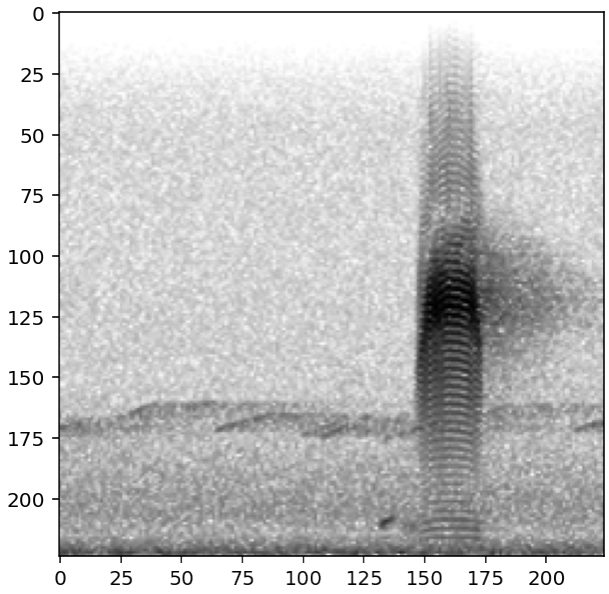

```
labels: tensor([0, 1])
```



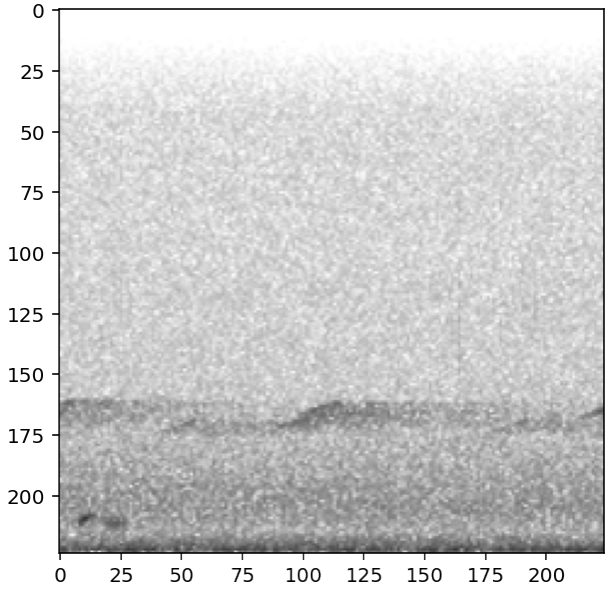
The `CnnPreprocessor` preprocessor allows you to turn all augmentation off or on as desired. Inspect the unaugmented images as well:

[13]:
```
train_dataset.augmentation_off()
for i, d in enumerate(train_dataset.sample(n=4)):
    print(f"labels: {d['y']}")
    show_tensor(d)
```

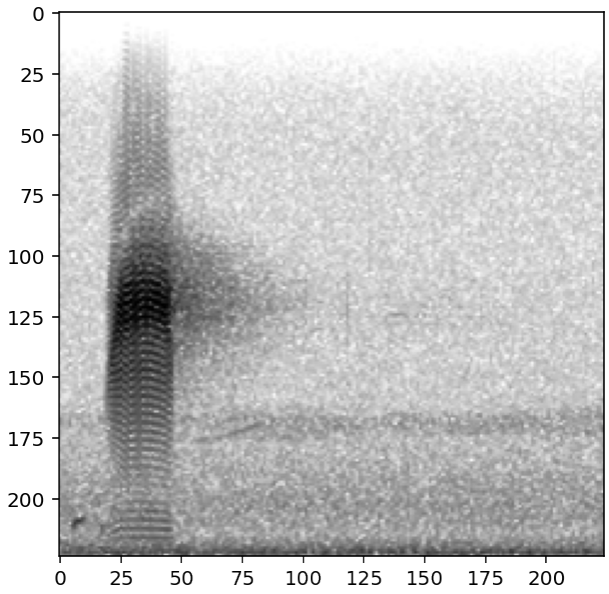
```
labels: tensor([0, 1])
```



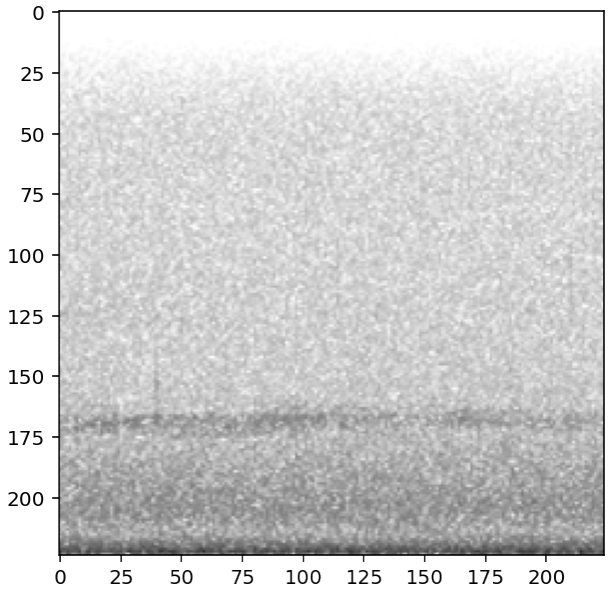

```
labels: tensor([0, 1])
```




```
labels: tensor([0, 1])
```




```
labels: tensor([1, 0])
```



8.2 Training

Now, we create a convolutional neural network model object, train it on the `train_dataset` with validation from `valid_dataset`, and use it for prediction.

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 "presence" and "absence." * The model is "single target," meaning that each sample belongs to exactly one class, "present" or "absent."

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 a thresholded model with binarized (1/0) outputs.

The model object should be initialized with a list of class names that matches the class names in the training dataset. We use the Resnet18 CNN architecture with binary classifier output, `Resnet18Binary`. For more details on other CNN architectures, see the "Custom CNNs" tutorial.

[14]:
```
# Create model object
classes = train_df.columns
model = Resnet18Binary(classes)
```

```
created PytorchModel model object with 2 classes
```

8.2.2 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 hundreds of epochs on hundreds of training files to create a useful model.

[15]:
```
model.train(
    train_dataset=train_dataset,
    valid_dataset=valid_dataset,
    save_path='./binary_train/',
    epochs=5,
    batch_size=8,
    save_interval=100,
    num_workers=0,
)
```

```
Epoch: 0 [batch 0/3 (0.00%)]
        Jacc: 0.125 Hamm: 0.750 DistLoss: 0.951

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Updating best model
Saving to binary_train/best.model
Epoch: 1 [batch 0/3 (0.00%)]
        Jacc: 0.312 Hamm: 0.375 DistLoss: 0.980

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Epoch: 2 [batch 0/3 (0.00%)]
        Jacc: 0.750 Hamm: 0.125 DistLoss: 0.435

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Epoch: 3 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.151
```

(continues on next page)

(continued from previous page)

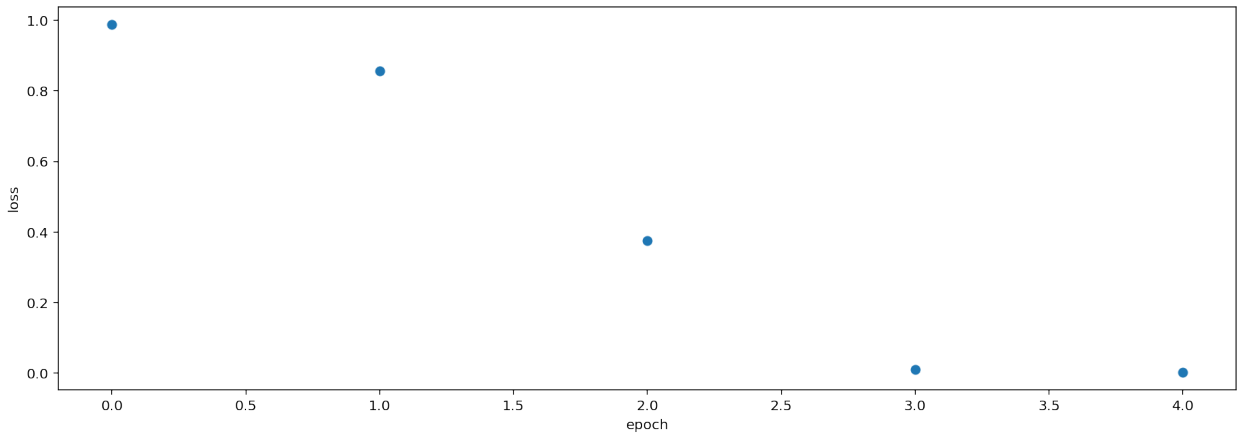
```
Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Epoch: 4 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.022

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Saving weights, metrics, and train/valid scores.
Saving to binary_train/epoch-4.model

Best Model Appears at Epoch 0 with F1 0.909.
```

8.2.3 Plot the loss history

We can plot the loss from each epoch to check that our loss is declining

[16]:
```
plt.scatter(model.loss_hist.keys(),model.loss_hist.values())
plt.xlabel('epoch')
plt.ylabel('loss')
```

[16]:
```
Text(0, 0.5, 'loss')
```



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.

8.3.1 Create preprocessor for prediction

Similar to training, prediction requires the use of a Preprocessor. The dataset can be an instance of any Preprocessor class, as long as it provides the correct tensor shape to the model. To load audio and create a spectrogram without applying any augmentation, we use the class `AudioToSpectrogramPreprocessor`.

In this instance, we'll reuse the validation dataset used above, but in a real application you would likely want to use model prediction on a separate dataset, e.g., an unlabeled dataset that you want to classify or a different labeled dataset for testing the model's performance.

[17]:
```
prediction_dataset = AudioToSpectrogramPreprocessor(valid_df)
```

8.3.2 Predict on the validation dataset

We simply call model's `.predict()` method on a Preprocessor instance.

This will return three dataframes: - scores : numeric predictions from the model for each sample and class - predictions: 0/1 predictions from the model for each sample and class (only generated if `binary_predictions` argument is supplied) - labels: Original labels from the dataset, if available

[18]:
```
valid_scores_df, valid_preds_df, valid_labels_df = model.predict(prediction_dataset)
valid_scores_df.head()
```

```
(6, 2)
```

[18]:

	negative	positive
./woodcock_labeled_data/882de25226ed989b31274ee...	-3.613683	4.224705
./woodcock_labeled_data/92647ab903049a9ee4125ab...	-3.267569	4.315749
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	-3.238488	4.818709
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	-3.355691	4.015820
./woodcock_labeled_data/ad14ac7ffa729060712b442...	0.057844	0.814907

[19]:
```
# None: not generated because the `binary_predictions` argument was not supplied
valid_preds_df
```

[20]:
```
valid_labels_df.head()
```

[20]:

	negative	positive
./woodcock_labeled_data/882de25226ed989b31274ee...	0	1
./woodcock_labeled_data/92647ab903049a9ee4125ab...	0	1
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	0	1
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	0	1
./woodcock_labeled_data/ad14ac7ffa729060712b442...	1	0

You may discover that `valid_preds` is currently `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).

8.3.3 Binarize predictions

Supplying the `binary_preds` argument returns a dataframe in which the scores are "binarized," i.e., transformed from real numbers to either 0 or 1.

Note: Typically it's not a good idea to treat binary predictions as scientific outputs unless your machine learning algorithm has been thoroughly tested and you understand its 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(prediction_dataset,binary_preds='single_target')
preds.head()
```

```
(6, 2)
```

[21]:

	negative	positive
./woodcock_labeled_data/882de25226ed989b31274ee...	0.0	1.0
./woodcock_labeled_data/92647ab903049a9ee4125ab...	0.0	1.0
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	0.0	1.0
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	0.0	1.0
./woodcock_labeled_data/ad14ac7ffa729060712b442...	0.0	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(
    prediction_dataset,
    binary_preds='multi_target',
    threshold=0.1,
)
pred_df.head()
```

```
(6, 2)
```

[22]:

	negative	positive
./woodcock_labeled_data/882de25226ed989b31274ee...	0.0	1.0
./woodcock_labeled_data/92647ab903049a9ee4125ab...	0.0	1.0
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	0.0	1.0
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	0.0	1.0
./woodcock_labeled_data/ad14ac7ffa729060712b442...	1.0	1.0

Note that in some of the above predictions, 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.

While thresholding can be used for presence/absence applications like the one demonstrated here, this problem will be common. Even a threshold of 0.5 or higher is not a guarantee that the results will be single-target: because the sigmoid function does not guarantee that the scores sum to 1, both classes could receive sigmoid'd scores higher than 0.5.

8.3.4 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).

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] 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 may want to use the `'softmax'` activation.

[23]:
```
valid_scores, valid_preds, valid_labels = model.predict(prediction_dataset,␣
↪activation_layer='softmax')
```

```
(6, 2)
```

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

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

[24]:

filename	negative	positive	pred_negative	pred_positive
./woodcock_labeled_data/75b2f63e032dbd6d1979004...	0	1	0.000317	0.999683
./woodcock_labeled_data/882de25226ed989b31274ee...	0	1	0.000394	0.999606
./woodcock_labeled_data/01c5d0c90bd4652f308fd9c...	0	1	0.000629	0.999371
./woodcock_labeled_data/ad14ac7ffa729060712b442...	1	0	0.319284	0.680716
./woodcock_labeled_data/92647ab903049a9ee4125ab...	0	1	0.000509	0.999491

8.3.5 Parallelizing prediction

Two parameters can be used to increase training 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
- `batch_size`: number of samples to predict on simultaneously

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

```
(6, 2)
```

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

Training and prediction with these models looks similar to that of two-class models, with a few extra considerations.

For example, make a `Resnet18Multiclass` model. This model is multi-target by default, which you can see by inspecting the `model.single_target` attribute:

[26]:
```
from opensoundscape.torch.models.cnn import Resnet18Multiclass
model = Resnet18Multiclass(['negative','positive'])
print("model.single_target:", model.single_target)
```

```
created PytorchModel model object with 2 classes
model.single_target: False
```

If you want a single-target model, uncomment and run the following line.

[27]:
```
#model.single_target = True
```

8.4.1 Train

Training looks the same as in two-class models. In practice, using larger batch sizes (64+) improves stability and generalizability of training.

[28]:
```
model.train(
    train_dataset=train_dataset,
    valid_dataset=valid_dataset,
    save_path='./multilabel_train/',
    epochs=1,
    batch_size=16,
    save_interval=100,
    num_workers=0
)
```

```
Epoch: 0 [batch 0/2 (0.00%)]
        Jacc: 0.513 Hamm: 0.438 DistLoss: 19.551

Validation.
(6, 2)
         Precision: 0.4166666666666667
         Recall: 0.5
         F1: 0.45454545454545453
Saving weights, metrics, and train/valid scores.
Saving to multilabel_train/epoch-0.model
Updating best model
Saving to multilabel_train/best.model

Best Model Appears at Epoch 0 with F1 0.455.
```

8.4.2 Predict

Prediction looks the same as demonstrated above, but make sure to think carefully: * What `activation_layer` do I 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.

[29]:
```
train_preds,_,_ = model.predict(train_dataset)
train_preds.columns = ['pred_negative','pred_positive']
train_dataset.df.join(train_preds).head()
```

```
(23, 2)
```

[29]:

filename	negative	positive	pred_negative	pred_positive
./woodcock_labeled_data/49890077267b569e142440f...	0	1	-5.401826	3.185288
./woodcock_labeled_data/ad90eefb6196ca83f9cf43b...	0	1	-5.691168	3.318331
./woodcock_labeled_data/e9e7153d11de3ac8fc3f716...	0	1	-6.223546	2.960380
./woodcock_labeled_data/c057a4486b25cd638850fc0...	0	1	-3.477443	1.881113
./woodcock_labeled_data/0c453a87185d8c7ce05c5c5...	0	1	-5.862498	3.441918

8.5 Save and load models

A machine learning model is, in essence, a complex mathematical equation comprised of two parts: * architecture: the particular structured connections between inputs and outputs of the model, not modified during training. Examples of architectures in OpenSoundscape include `Resnet18Binary`. * weights: the strength of these connections, tuned by training the model.

Training a model modifies the weights to attempt to better differentiate and classify samples. An analogy is linear regression: the "architecture" is a linear equation, and the "weights" are the slope and intercept identified to best fit the dataset.

Saving a model saves its weights.

Loading a model applies the saved weights to the architecture. We will need to know the architecture to load the model.

8.5.1 Save

OpenSoundscape saves models automatically during training: * The model saves weights to `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 F1 score on the validation dataset

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

[30]:
```
model1 = Resnet18Binary(classes)
# Save every 2 epochs
model1.train(
    train_dataset=train_dataset,
    valid_dataset=valid_dataset,
    epochs=6,
    batch_size=8,
    save_path='./binary_train/',
    save_interval=2,
    num_workers=0
)
model1.save('./binary_train/my_favorite.model')
```

```
created PytorchModel model object with 2 classes
Epoch: 0 [batch 0/3 (0.00%)]
        Jacc: 0.312 Hamm: 0.375 DistLoss: 0.586

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Updating best model
Saving to binary_train/best.model
Epoch: 1 [batch 0/3 (0.00%)]
        Jacc: 0.679 Hamm: 0.125 DistLoss: 0.257

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Saving weights, metrics, and train/valid scores.
Saving to binary_train/epoch-1.model
Epoch: 2 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.085

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Epoch: 3 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.017

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Saving weights, metrics, and train/valid scores.
Saving to binary_train/epoch-3.model
Epoch: 4 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.003

Validation.
(6, 2)
         Precision: 0.8333333333333334
```

(continues on next page)

(continued from previous page)

```
         Recall: 1.0
         F1: 0.9090909090909091
Epoch: 5 [batch 0/3 (0.00%)]
        Jacc: 1.000 Hamm: 0.000 DistLoss: 0.004

Validation.
(6, 2)
         Precision: 0.8333333333333334
         Recall: 1.0
         F1: 0.9090909090909091
Saving weights, metrics, and train/valid scores.
Saving to binary_train/epoch-5.model

Best Model Appears at Epoch 0 with F1 0.909.
Saving to binary_train/my_favorite.model
```

8.5.2 Load

Models can be loaded from a saved file by either of two methods: - creating an instance of the model class, then calling `.load()` - calling the model class's `.from_checkpoint()` method

In either case, you must know the architecture of the saved model (the OpenSoundscape class it was created with).

The two methods produce equivalent results:

[31]:
```
#create model object then load weights from checkpoint
model = Resnet18Binary(classes)
model.load('./binary_train/best.model')
```

```
created PytorchModel model object with 2 classes
loading weights from saved object
```

[32]:
```
#or, create object directly from checkpoint
model = Resnet18Binary.from_checkpoint('./binary_train/best.model')
```

```
created PytorchModel model object with 2 classes
loading weights from saved object
```

The model can now be used for prediction (`model.predict()`) or to continue training (`model.train()`).

8.6 Predict using saved model

Using a saved or downloaded model to run predictions on audio files is as simple as 1. Creating a model object with saved weights 2. Creating an instance of a preprocessor class for prediction, e.g. `AudioToSpectrogramPreprocessor()` 3. Running `model.predict()` on the preprocessor

[33]:
```
# load the saved model
model = Resnet18Binary.from_checkpoint('./binary_train/best.model')

# create a Preprocessor instance with the audio samples
prediction_dataset = AudioToSpectrogramPreprocessor(valid_df,return_labels=False)

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

```
created PytorchModel model object with 2 classes
loading weights from saved object
(6, 2)
```

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.

By default, `.load()` loads the optimizer parameters and learning rate parameters from the saved model, in addition to the network weights.

[34]:
```
# Create architecture
model = Resnet18Binary(classes)

# Load the model weights and training parameters
model.load('./binary_train/best.model')

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

```
created PytorchModel model object with 2 classes
loading weights from saved object

Best Model Appears at Epoch 0 with F1 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:

[35]:
```
dirs = ['./multilabel_train', './binary_train', './woodcock_labeled_data']
paths = [Path(f) for f in dirs]
for p in paths:
    [p.unlink() for p in p.glob("*")]
    p.rmdir()
```

CHAPTER 9

Custom preprocessing

Preprocessors in OpenSoundscape perform all of the preprocessing steps from loading a file from the disk up to providing a sample to the machine learning algorithm for training or prediction. These classes are used when (a) training a machine learning model in OpenSoundscape, or (b) making predictions with a machine learning model in OpenSoundscape.

If you are already familiar with PyTorch, you might notice that Preprocessors take the place of, and are children of, PyTorch's Dataset classes to provide each sample to PyTorch as a Tensor.

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 into "Pipelines" to preprocess data
- modifying the parameters of actions
- turning Actions on and off
- modifying the order and contents of pipelines
- use of the `AudioToSpectrogramPreprocessor` class, including examples of:
  - modifying audio and spectrogram parameters
  - changing the output image shape
  - changing the output type
- use of the `CnnPreprocessor` class, including examples of:
  - choosing between default "augmentation on" and "augmentation off" pipelines
  - modifying augmentation parameters
  - using the "overlay" augmentation
- writing custom preprocessors and actions

First, import the needed 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 BasePreprocessor,␣
↪AudioToSpectrogramPreprocessor, CnnPreprocessor

#other utilities and packages
from opensoundscape.helpers import run_command
import torch
import pandas as pd
from pathlib import Path
import numpy as np
import pandas as pd
import random
```

Set up plotting and some helper functions.

[2]:
```
#set up plotting
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize']=[15,5] #for large visuals
%config InlineBackend.figure_format = 'retina'

# helper function for displaying a sample as an image
def show_tensor(sample):
    plt.imshow((sample['X'][0,:,:]/2+0.5)*-1,cmap='Greys',vmin=-1,vmax=0)
    plt.show()
```

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)
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 `curl` and `tar` installed on your computer, as they will download and unzip a compressed file in `.tar.gz` format.
2. Download a `.zip` version of the files by clicking here. You will have to unzip this folder and place the unzipped folder in the same folder that this notebook is in.

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

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

(continues on next page)

(continued from previous page)

```
for command in commands:
    run_command(command)
```

9.1.2 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.

We manipulate the label dataframe to give "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. 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).

For more details on the steps below, see the basic CNN training and prediction tutorial.

[5]:
```
#load Specky output: a table of labeled audio files
labels = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))

#update the paths to the audio files
labels.filename = ['./woodcock_labeled_data/'+f for f in labels.filename]

#generate "one-hot" labels
labels['negative']=[0 if label=='present' else 1 for label in labels['woodcock']]
labels['positive']=[1 if label=='present' else 0 for label in labels['woodcock']]

#use the file path as the index, and class names as the only columns
classes = ['negative','positive']
labels = labels.set_index('filename')[classes]
labels.head()
```

[5]:

filename	negative	positive
./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

9.2 Intro to Preprocessors

Preprocessors prepare samples for use by machine learning algorithms by stringing together transformations called **Actions** into a **Pipeline**. The preprocessor sequentially applies to the sample each Action in the Pipeline. You can add, remove, and rearrange Actions from the pipeline and change the parameters of each Action.

The currently implemented Preprocessor classes and their Actions include:

- `CnnPreprocessor` - loads audio files, creates spectrograms, performs various augmentations, and returns a pytorch Tensor.
- `AudioToSpectrogramPreprocessor` - loads audio files, creates spectrograms, and returns a pytorch Tensor (no augmentation).

9.2.1 Initialize preprocessor

A Preprocessor must be initialized with 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 `CnnPreprocessor`:

[6]:
```
from opensoundscape.preprocess.preprocessors import CnnPreprocessor

preprocessor = CnnPreprocessor(labels)
```

9.2.2 Access sample from a Preprocessor

A sample is accessed in a preprocessor using indexing, like 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.

[7]:
```
preprocessor[0]
```

[7]:
```
{'X': tensor([[[0.0000, 0.0000, 0.0000,  ..., 0.4976, 0.4393, 0.4653],
          [0.0000, 0.0000, 0.0000,  ..., 0.4880, 0.4658, 0.4754],
          [0.0000, 0.0000, 0.0000,  ..., 0.4894, 0.4882, 0.4231],
          ...,
          [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,  ..., 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0000, 0.0000,  ..., 0.4694, 0.4241, 0.4743],
          [0.0000, 0.0000, 0.0000,  ..., 0.4994, 0.4457, 0.4963],
          [0.0000, 0.0000, 0.0000,  ..., 0.4734, 0.4847, 0.4085],
          ...,
          [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,  ..., 0.0000, 0.0000, 0.0000]],

         [[0.0000, 0.0000, 0.0000,  ..., 0.4864, 0.4233, 0.4630],
          [0.0000, 0.0000, 0.0000,  ..., 0.4857, 0.4357, 0.4965],
          [0.0000, 0.0000, 0.0000,  ..., 0.4995, 0.4914, 0.4010],
          ...,
          [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,  ..., 0.0000, 0.0000, 0.0000]]]),
 'y': tensor([0, 1])}
```

9.2.3 Subset samples from a Preprocessor

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

[8]:
```
len(preprocessor)
```

```
[8]: 29
```

Select the first 10 samples (non-random)

```
[9]: len(preprocessor.head(5))
[9]: 5
```

Randomly select an absolute number of samples

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

Randomly select a fraction of samples

```
[11]: len(preprocessor.sample(frac=0.5))
[11]: 14
```

9.3 Pipelines and actions

Each Preprocessor class has two attributes, `preprocessor.pipeline` and `preprocessor.actions`. Pipelines are comprised of Actions.

9.3.1 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 a list of Actions, where each Action is an instance of a class that sub-classes `opensoundscape.preprocess.BaseAction`.

Inspect the current pipeline.

```
[12]: # inspect the current pipeline (ordered sequence of Actions to take)
preprocessor.pipeline
```

```
[12]: [<opensoundscape.preprocess.actions.AudioLoader at 0x7fd98ef45e50>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd98ef454f0>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd98ef45970>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd99603ba30>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd99603b8e0>,
 <opensoundscape.preprocess.actions.BaseAction at 0x7fd99603b1f0>,
 <opensoundscape.preprocess.actions.TorchColorJitter at 0x7fd99603b220>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd99603b160>,
 <opensoundscape.preprocess.actions.TimeMask at 0x7fd99603b0a0>,
 <opensoundscape.preprocess.actions.FrequencyMask at 0x7fd99603b070>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd99603b280>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd98ef45c10>,
 <opensoundscape.preprocess.actions.TorchRandomAffine at 0x7fd99603b130>]
```

9.3.2 About actions

Preprocessors come with a set of predefined Actions that are available to the preprocessor. These are not necessarily all included in the preprocessing pipeline; these are just the transformations that are available to be strung together into a pipeline if desired.

- The Actions are stored in the `preprocessor.actions` attribute. Each Action is an instance of a class (described in more detail below).
- Each Action takes a sample (and its labels*), performs some transformation to them, and returns the sample (and its labels*). The code for this transformation is stored in the Action's `.go()` method.
- You can customize Actions using the `.on()` and `.off()` methods to turn the Action on or off, or by changing the action's parameters. Any customizable parameters for performing the Action are stored in a dictionary, `.params`. This dictionary can be modified using the Action's `.set()` method, e.g. `action.set(param=value, param2=value2, ...)`.
- You can view all the available Actions in a preprocessor using the `.list_actions()` method.

[13]:
```
# create a new instance of a CnnPreprocessor
preprocessor = AudioToSpectrogramPreprocessor(labels)

# print all Actions that have been added to the preprocessor
# (Note that this is not the pipeline, just a collection of available actions)
preprocessor.actions.list_actions()
```

[13]:
```
['load_audio',
 'trim_audio',
 'to_spec',
 'bandpass',
 'to_img',
 'to_tensor',
 'normalize']
```

Notice that the Actions in `preprocessor.actions.list_actions()` are not identical to the names listed in the pipeline, but are parallel. For example, in this case, `preprocessor.actions.to_spec` corresponds to an instance of `opensoundscape.preprocess.actions.AudioToSpectrogram`:

[14]:
```
preprocessor.actions.to_spec
```

[14]:
```
<opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd99698aa60>
```

That's because of the structure of actions:

- The `.actions` attribute is an instance of a class called `ActionContainer` (see below)
- The `ActionContainer` has an attribute for each possible action, e.g. `preprocessor.actions.to_spec`
- Each attribute is defined as an instance of an Action class, e.g. `AudioToSpectrogram`
- Each Action class is a child of a class called `BaseAction`; see the `actions` module for examples.

[15]:
```
preprocessor.actions?
```

```
Type:        ActionContainer
String form: <opensoundscape.preprocess.actions.ActionContainer object at␣
↪0x7fd99698a280>
File:        ~/Code/opensoundscape/opensoundscape/preprocess/actions.py
Docstring:
this is a container object which holds instances of Action child-classes
```

(continues on next page)

(continued from previous page)

```
the Actions it contains each have .go(), .on(), .off(), .set(), .get()

The actions are un-ordered and may not all be used. In preprocessor objects
such as AudioToSpectrogramPreprocessor, Actions from the action
container are listed in a pipeline(list), which defines their order of use.

To add actions to the container: action_container.loader = AudioLoader()
To set parameters of actions: action_container.loader.set(param=value,...)

Methods: list_actions()
```

9.4 Modifying Actions

9.4.1 View default parameters for an Action

The docstring for an individual action, such as `preprocessor.actions.to_spec`, gives information on what parameters can be changed and what the defaults are.

[16]:
```
preprocessor.actions.to_spec?
```

```
Type:        AudioToSpectrogram
String form: <opensoundscape.preprocess.actions.AudioToSpectrogram object at␣
↪0x7fd99698aa60>
File:        ~/Code/opensoundscape/opensoundscape/preprocess/actions.py
Docstring:
Action child class for Audio.from_file() (Audio -> Spectrogram)

see spectrogram.Spectrogram.from_audio for documentation

Args:
    window_type="hann":
        see scipy.signal.spectrogram docs for description of window parameter
    window_samples=512:
        number of audio samples per spectrogram window (pixel)
    overlap_samples=256:
        number of samples shared by consecutive windows
    decibel_limits = (-100,-20) :
    limit the dB values to (min,max)
    (lower values set to min, higher values set to max)
```

Any defaults that have been changed will be shown in the `.params` attribute of the action:

[17]:
```
preprocessor.actions.to_spec.params
```

[17]:
```
{}
```

9.4.2 Modify Action parameters

In general, Actions are modified using the `set()` method, e.g.:

[18]:
```
preprocessor.actions.to_spec.set(window_samples=256)
```

We can check that the values were actually changed by printing the action's params. This is not guaranteed to print the defaults, but will definitely print the parameters that have actively changed.

[19]:
```
print(preprocessor.actions.load_audio.params)
```

```
{'sample_rate': 22050}
```

9.4.3 Turn individual Actions on or off

Each Action has `.on()` and `.off()` methods which toggle a bypass of the Action in the pipeline. Note that the Actions will still remain in the same order in the pipeline, and can be turned back on again if desired.

[20]:
```
#initialize a preprocessor that includes augmentation
preprocessor = CnnPreprocessor(labels)
preprocessor.pipeline
```

[20]:
```
[<opensoundscape.preprocess.actions.AudioLoader at 0x7fd9969923a0>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd996992430>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd996992490>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd9969924f0>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd996992550>,
 <opensoundscape.preprocess.actions.BaseAction at 0x7fd996992670>,
 <opensoundscape.preprocess.actions.TorchColorJitter at 0x7fd9969926d0>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd9969925b0>,
 <opensoundscape.preprocess.actions.TimeMask at 0x7fd996992790>,
 <opensoundscape.preprocess.actions.FrequencyMask at 0x7fd9969927f0>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd996992850>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd996992640>,
 <opensoundscape.preprocess.actions.TorchRandomAffine at 0x7fd996992730>]
```

[21]:
```
#turn off augmentations other than noise
preprocessor.actions.color_jitter.off()
preprocessor.actions.add_noise.off()
preprocessor.actions.time_mask.off()
preprocessor.actions.frequency_mask.off()
preprocessor.pipeline
```

[21]:
```
[<opensoundscape.preprocess.actions.AudioLoader at 0x7fd9969923a0>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd996992430>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd996992490>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd9969924f0>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd996992550>,
 <opensoundscape.preprocess.actions.BaseAction at 0x7fd996992670>,
 <opensoundscape.preprocess.actions.TorchColorJitter at 0x7fd9969926d0>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd9969925b0>,
 <opensoundscape.preprocess.actions.TimeMask at 0x7fd996992790>,
 <opensoundscape.preprocess.actions.FrequencyMask at 0x7fd9969927f0>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd996992850>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd996992640>,
 <opensoundscape.preprocess.actions.TorchRandomAffine at 0x7fd996992730>]
```

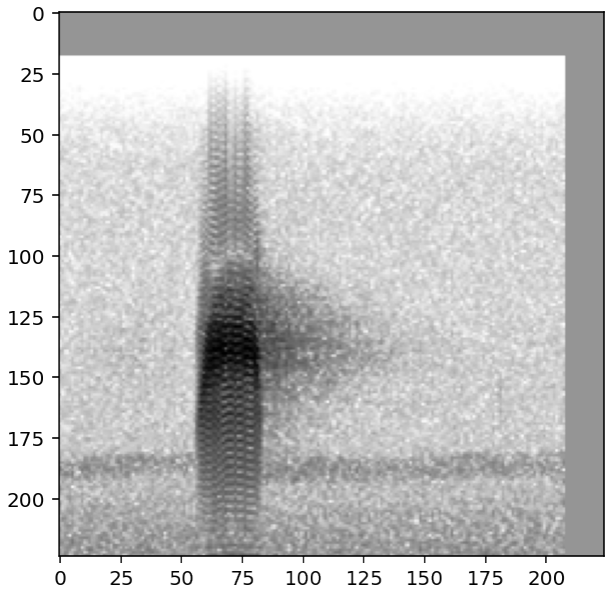
[22]:
```
print('random affine on')
show_tensor(preprocessor[0])
```

(continues on next page)

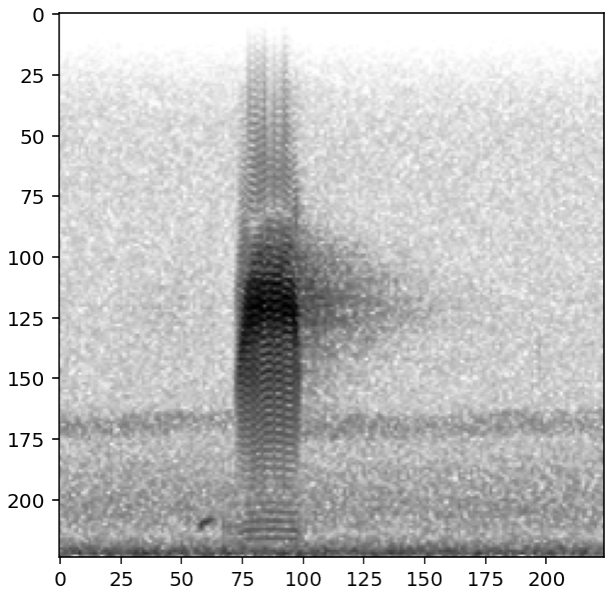
(continued from previous page)

```
print('random affine off')
preprocessor.actions.random_affine.off()
show_tensor(preprocessor[0])
```

```
random affine on
```




```
random affine off
```



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[0].bypass
```

[23]:
```
False
```

[24]:
```
# The TorchRandomAffine Action that we turned off
preprocessor.pipeline[-1].bypass
```

[24]:
```
True
```

9.5 Modifying the pipeline

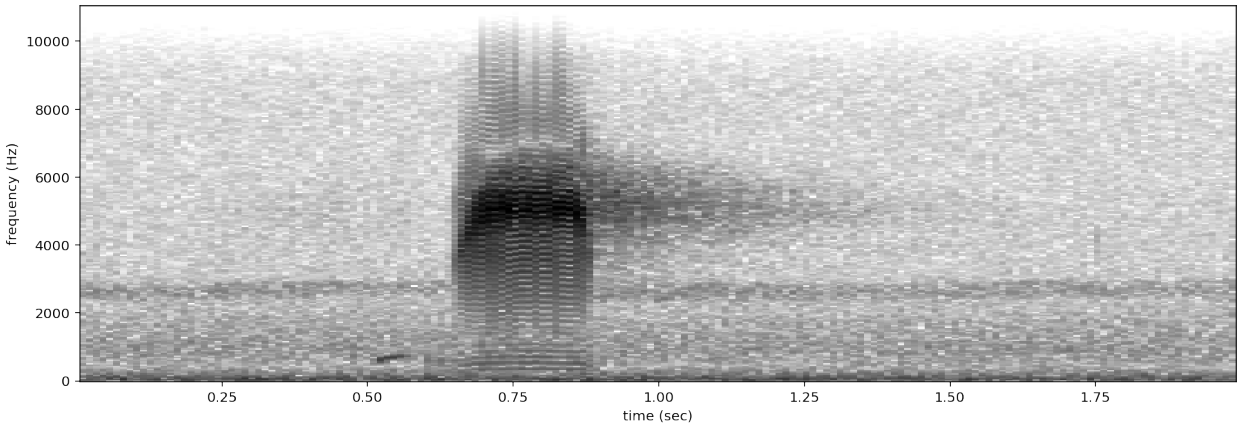
Sometimes, you may want to change the order or composition of the Preprocessor's pipeline. You can simply overwrite the `.pipeline` attribute, as long as the new pipeline is still a list of Action instances from the preprocessor's `.actions` ActionContainer.

9.5.1 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 = AudioToSpectrogramPreprocessor(labels)
print('original pipeline:')
[print(p) for p in preprocessor.pipeline]

#overwrite the pipeline with a slice of the original pipeline
print('\nnew pipeline:')
preprocessor.pipeline = preprocessor.pipeline[0:3]

[print(p) for p in preprocessor.pipeline]

print('\nwe now have a preprocessor that returns Spectrograms instead of Tensors:')
print(type(preprocessor[0]['X']))
preprocessor[0]['X'].plot()
```

```
original pipeline:
<opensoundscape.preprocess.actions.AudioLoader object at 0x7fd9947e1f70>
<opensoundscape.preprocess.actions.AudioTrimmer object at 0x7fd9947e1eb0>
<opensoundscape.preprocess.actions.AudioToSpectrogram object at 0x7fd9947e1b50>
<opensoundscape.preprocess.actions.SpectrogramBandpass object at 0x7fd9947e1b80>
<opensoundscape.preprocess.actions.SpecToImg object at 0x7fd9947e1e50>
<opensoundscape.preprocess.actions.ImgToTensor object at 0x7fd996b40fd0>
<opensoundscape.preprocess.actions.TensorNormalize object at 0x7fd996b2ed30>

new pipeline:
<opensoundscape.preprocess.actions.AudioLoader object at 0x7fd9947e1f70>
<opensoundscape.preprocess.actions.AudioTrimmer object at 0x7fd9947e1eb0>
<opensoundscape.preprocess.actions.AudioToSpectrogram object at 0x7fd9947e1b50>

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



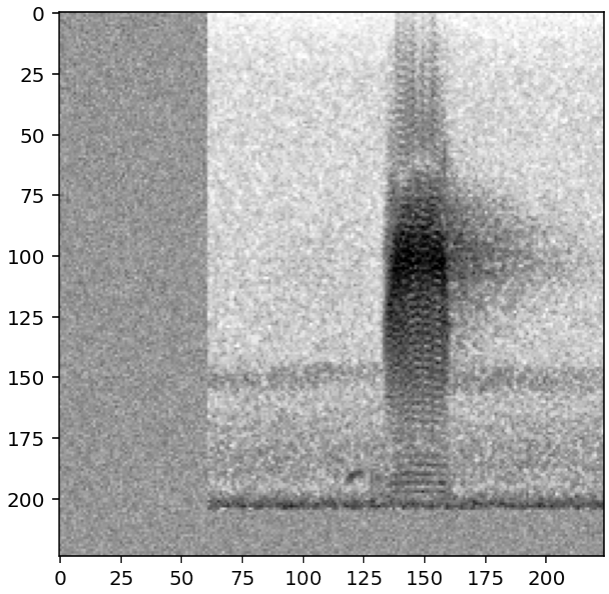
9.5.2 Example: custom augmentation pipeline

Here's an example where we add a new Action to the Action container, then overwrite the preprocessing pipeline with one that includes our new action.

Note that each Action requires a specific input Type and may return that same Type or a different Type. So you'll need to be careful about the order of your Actions in your pipeline

This custom pipeline will first performs a Gaussian noise augmentation, then a random affine, then our second noise augmentation (add_noise_2)

[26]:
```
#initialize a preprocessor
preprocessor = CnnPreprocessor(labels)

#add a new Action to the Action container
from opensoundscape.preprocess.actions import TensorAddNoise
preprocessor.actions.add_noise_2 = TensorAddNoise(std=0.1)

#overwrite the pipeline with a list of Actions from .actions
preprocessor.pipeline = [
    preprocessor.actions.load_audio,
    preprocessor.actions.trim_audio,
    preprocessor.actions.to_spec,
    preprocessor.actions.bandpass,
    preprocessor.actions.to_img,
    preprocessor.actions.to_tensor,
    preprocessor.actions.normalize,
    preprocessor.actions.add_noise,
    preprocessor.actions.random_affine,
    preprocessor.actions.add_noise_2
]

show_tensor(preprocessor[0])
```



9.5.3 Use an Action multiple times in a pipeline

If an Action is present multiple times in a pipeline (e.g. multiple overlays), changing the parameters of the Action at one point in the pipeline will change it at all points in the pipeline. For instance, create a pipeline with multiple "add noise" steps:

[27]:
```
#initialize a preprocessor that includes augmentation
preprocessor = CnnPreprocessor(labels)

# Insert another instance of the "add_noise" action into the pipeline
preprocessor.pipeline.insert(-2, preprocessor.actions.add_noise)
preprocessor.pipeline
```

[27]:
```
[<opensoundscape.preprocess.actions.AudioLoader at 0x7fd98e583130>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd996990040>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd996990f10>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd996990790>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd996990730>,
 <opensoundscape.preprocess.actions.BaseAction at 0x7fd996ab3eb0>,
 <opensoundscape.preprocess.actions.TorchColorJitter at 0x7fd98fdca8b0>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd996990ca0>,
 <opensoundscape.preprocess.actions.TimeMask at 0x7fd996ae0970>,
 <opensoundscape.preprocess.actions.FrequencyMask at 0x7fd996ae0910>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd996ae0430>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd996ae0430>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd996ae6880>,
 <opensoundscape.preprocess.actions.TorchRandomAffine at 0x7fd996ae0460>]
```

Note that changing the parameter of one of the `add_noise` steps changes the parameters for both of them.

[28]:
```
# Print the parameters of both of the TensorAddNoise Actions in the pipeline
print("Parameters of TensorAddNoise actions before changing:")
[print(f"Params of {p}:", p.params) for p in preprocessor.pipeline[-4:-2]]
```

(continues on next page)

(continued from previous page)

```
# Change the parameters of one of the add noise steps
preprocessor.pipeline[-4].set(std=0.01)

# The modification above is the same as:
#preprocessor.actions.add_noise.set(std=0.01)

# See that the parameters for both steps are changed
print("\nParameters of TensorAddNoise actions after changing:")
[print(f"Params of {p}:", p.params) for p in preprocessor.pipeline[-4:-2]];
```

```
Parameters of TensorAddNoise actions before changing:
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd996ae0430>:
↪ {'std': 0.005}
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd996ae0430>:
↪ {'std': 0.005}

Parameters of TensorAddNoise actions after changing:
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd996ae0430>:
↪ {'std': 0.01}
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd996ae0430>:
↪ {'std': 0.01}
```

To modify the parameters of Actions individually, add them as separate Actions in the pipeline by adding a new named action to the action container.

[29]:
```
from opensoundscape.preprocess.actions import TensorAddNoise

# Add a new possible action to the ActionContainer
preprocessor.actions.my_new_action = TensorAddNoise(std=0.005)

# Replace one of the old actions in the pipeline with the new one with different␣
↪parameters
preprocessor.pipeline[-3] = preprocessor.actions.my_new_action
```

Now notice that the two instances of the `TensorAddNoise` action can have different parameters.

[30]:
```
[print(f"Params of {p}:", p.params) for p in preprocessor.pipeline[-4:-2]];
```

```
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd996ae0430>:
↪ {'std': 0.01}
Params of <opensoundscape.preprocess.actions.TensorAddNoise object at 0x7fd98e598a60>:
↪ {'std': 0.005}
```

9.6 Customizing `AudioToSpectrogramPreprocessor`

Below are various examples of how to modify parameters of the Actions of the `AudioToSpectrogramPreprocessor` class, including the `AudioLoader`, `AudioToSpectrogram`, and `SpectrogramBandpass` actions.

9.6.1 Modify the sample rate

Re-sample all loaded audio to a specified rate during the load_audio action

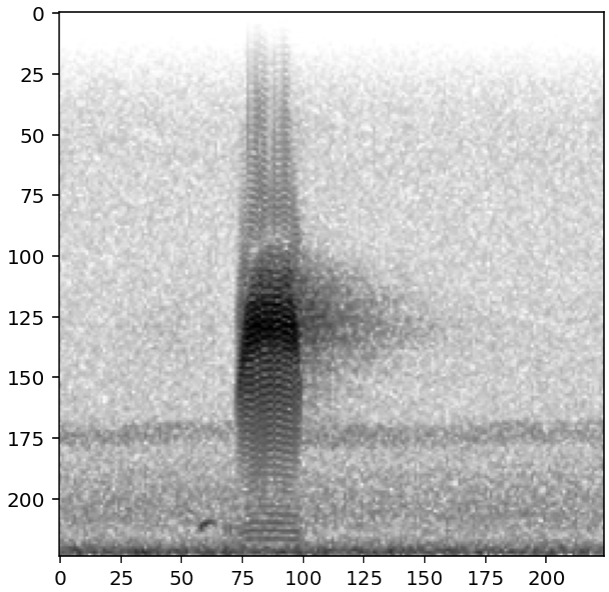
[31]:
```
preprocessor = AudioToSpectrogramPreprocessor(labels)

preprocessor.actions.load_audio.set(sample_rate=24000)
```

9.6.2 Modify spectrogram window length and overlap

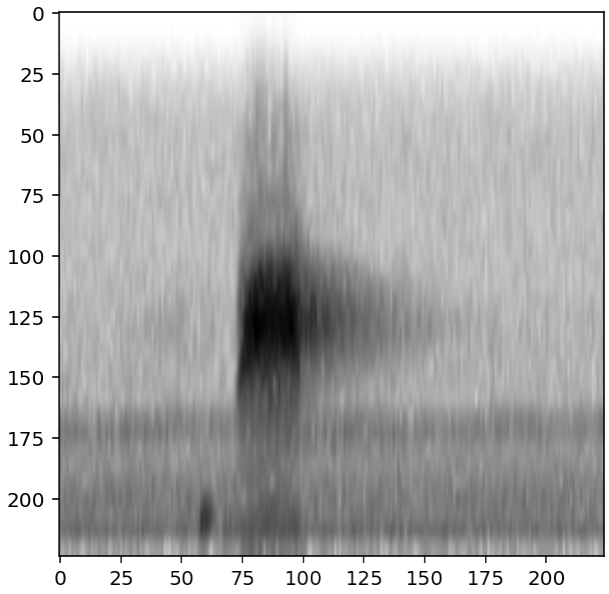
(see Spectrogram.from_audio() for detailed documentation)

[32]:
```
print('default parameters:')
show_tensor(preprocessor[0])

print('high time resolution, low frequency resolution:')
preprocessor.actions.to_spec.set(window_samples=64,overlap_samples=32)

show_tensor(preprocessor[0])
```

```
default parameters:
```




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

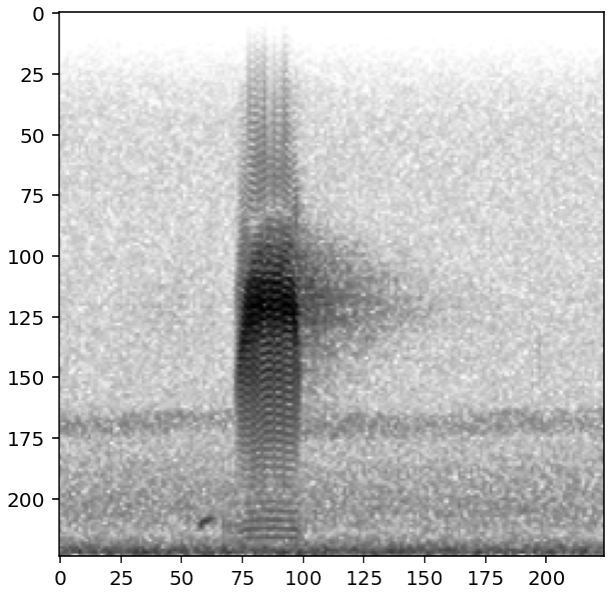


9.6.3 Bandpass spectrograms

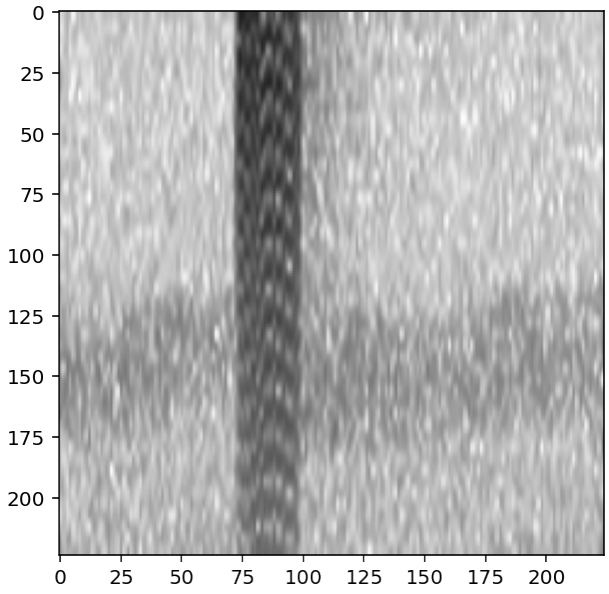
Trim spectrograms to a specified frequency range:

[33]:
```
preprocessor = AudioToSpectrogramPreprocessor(labels)

print('default parameters:')
show_tensor(preprocessor[0])

print('bandpassed to 2-4 kHz:')
preprocessor.actions.bandpass.set(min_f=2000,max_f=4000)
preprocessor.actions.bandpass.on()
show_tensor(preprocessor[0])
```

```
default parameters:
```



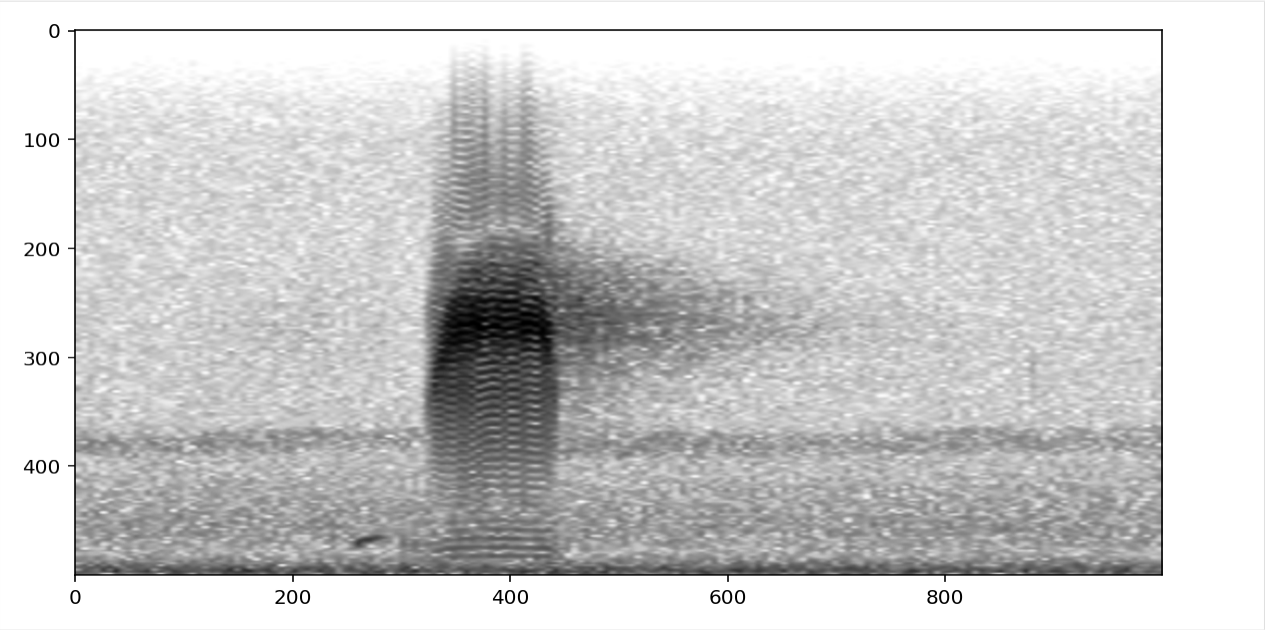

```
bandpassed to 2-4 kHz:
```



9.6.4 Change the output image

Change the shape of the output image:

[34]:
```
preprocessor = AudioToSpectrogramPreprocessor(labels)
preprocessor.actions.to_img.set(shape=[1000,500])
show_tensor(preprocessor[0])
```



9.7 Customizing `CnnPreprocessor`

The `CnnPreprocessor` class can be used to perform both audio and spectrogram transformation as well as augmentation for training with CNNs.

This section describes: * A special property of `CnnPreprocessor` which allows you to turn all augmentations on or off * Examples of modifying augmentation parameters for standard augmentations * Detailed descriptions of the useful "Overlay" augmentation

9.7.1 Turn all augmentation on or off

With `CnnPreprocessor`, we can easily choose between a pipeline that contains augmentations and a pipeline with no augmentations using the shortcuts `augmentation_off()` and `augmentation_on()` methods. Using these methods will overwrite any changes made to the pipeline, so apply them first before further customizing an instance of `CnnPreprocessor`.

[35]:
```
preprocessor = CnnPreprocessor(labels)
preprocessor.augmentation_off()
preprocessor.pipeline
```

[35]:
```
[<opensoundscape.preprocess.actions.AudioLoader at 0x7fd978c0af10>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd978c0afd0>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd978f9eb80>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd97974c760>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd97974c6a0>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd978ce20a0>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd978c217c0>]
```

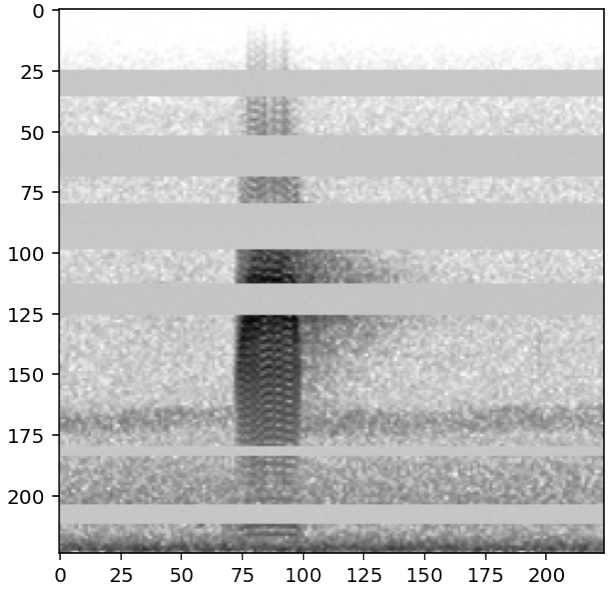
[36]:
```
preprocessor.augmentation_on()
preprocessor.pipeline
```

[36]:
```
[<opensoundscape.preprocess.actions.AudioLoader at 0x7fd978c0af10>,
 <opensoundscape.preprocess.actions.AudioTrimmer at 0x7fd978c0afd0>,
 <opensoundscape.preprocess.actions.AudioToSpectrogram at 0x7fd978f9eb80>,
 <opensoundscape.preprocess.actions.SpectrogramBandpass at 0x7fd97974c760>,
 <opensoundscape.preprocess.actions.SpecToImg at 0x7fd97974c6a0>,
 <opensoundscape.preprocess.actions.BaseAction at 0x7fd978c21670>,
 <opensoundscape.preprocess.actions.TorchColorJitter at 0x7fd978c217f0>,
 <opensoundscape.preprocess.actions.ImgToTensor at 0x7fd978ce20a0>,
 <opensoundscape.preprocess.actions.TimeMask at 0x7fd978c14d60>,
 <opensoundscape.preprocess.actions.FrequencyMask at 0x7fd978c14df0>,
 <opensoundscape.preprocess.actions.TensorAddNoise at 0x7fd978c14e20>,
 <opensoundscape.preprocess.actions.TensorNormalize at 0x7fd978c217c0>,
 <opensoundscape.preprocess.actions.TorchRandomAffine at 0x7fd978c14310>]
```

9.7.2 Modify augmentation parameters

`CnnPreprocessor` 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.

[37]:
```
#initialize a preprocessor
preprocessor = CnnPreprocessor(labels)

#turn off augmentations other than overlay
preprocessor.actions.color_jitter.off()
preprocessor.actions.random_affine.off()
preprocessor.actions.random_affine.off()
preprocessor.actions.time_mask.off()

# allow up to 20 horizontal masks, each spanning up to 0.1x the height of the image.
preprocessor.actions.frequency_mask.set(max_width = 0.1, max_masks=20)
show_tensor(preprocessor[0])
```

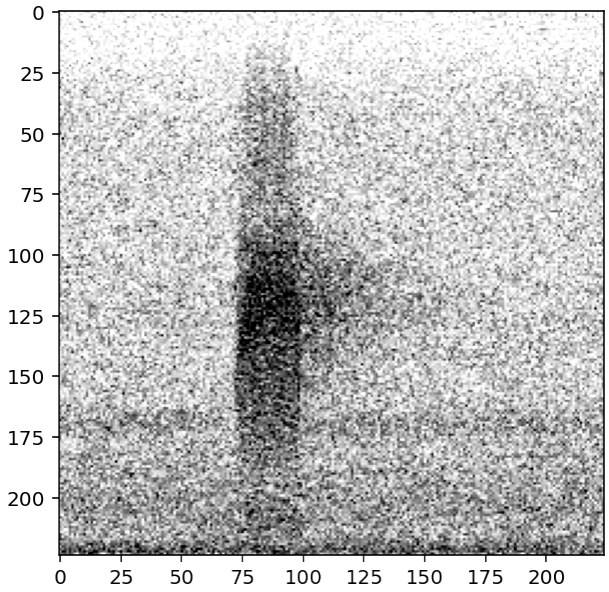


[38]:
```
#turn off frequency mask and turn on gaussian noise
preprocessor.actions.add_noise.on()
```

(continues on next page)

(continued from previous page)

```
preprocessor.actions.frequency_mask.off()

# increase the intensity of gaussian noise added to the image
preprocessor.actions.add_noise.set(std=0.2)
show_tensor(preprocessor[0])
```



9.7.3 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

Samples can be drawn from dataframes in a few general ways (each is demonstrated below):

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

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).

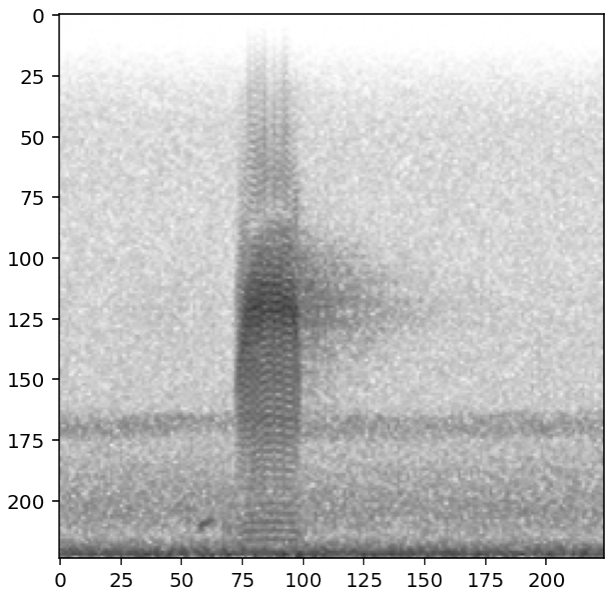
[39]:
```
#initialize a preprocessor and provide a dataframe with samples to use as overlays
preprocessor = CnnPreprocessor(labels, overlay_df=labels)

#turn off augmentations other than overlay
preprocessor.actions.color_jitter.off()
preprocessor.actions.random_affine.off()
preprocessor.actions.random_affine.off()
preprocessor.actions.time_mask.off()
preprocessor.actions.frequency_mask.off()
```

Modify `overlay_weight`

We'll first overlay a random sample with 30% of the final mix coming from the overlayed sample (70% coming from the original) by using `overlay_weight=0.3`.

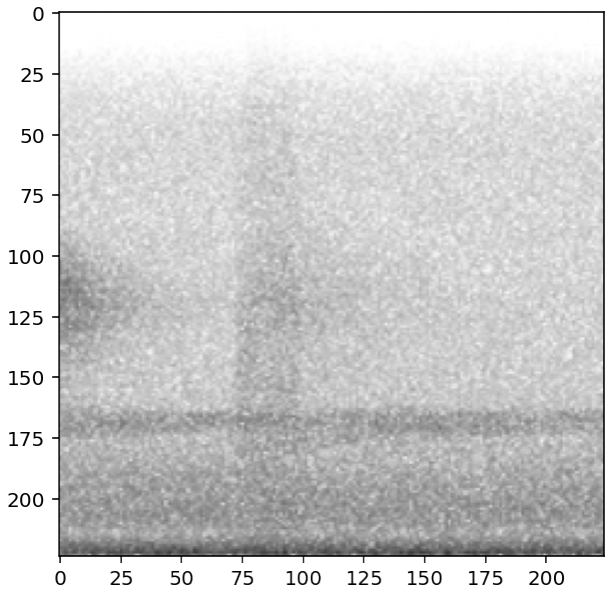
To demonstrate this, let's show what happens if we overlay samples from the "negative" class, resulting in the final sample having a higher or lower signal-to-noise ratio. By default, the overlay Action chooses a random file from the overlay dataframe. Instead, choose a sample from the class called `"negative"`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

[40]:
```
preprocessor.actions.overlay.set(
    overlay_class='negative',
    overlay_weight=0.3
)
show_tensor(preprocessor[0])
```



Now use `overlay_weight=0.8` to increase the contribution of the overlayed sample (80%) compared to the original sample (20%).

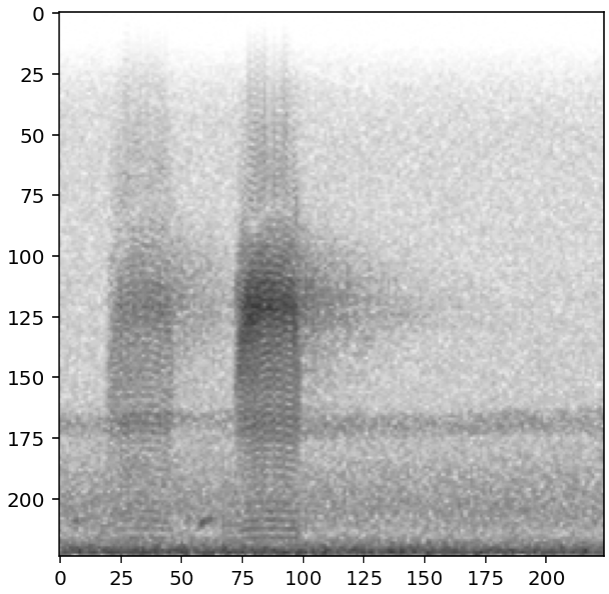
[41]:
```
preprocessor.actions.overlay.set(overlay_weight=0.8)
show_tensor(preprocessor[0])
```



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 "positive" class.

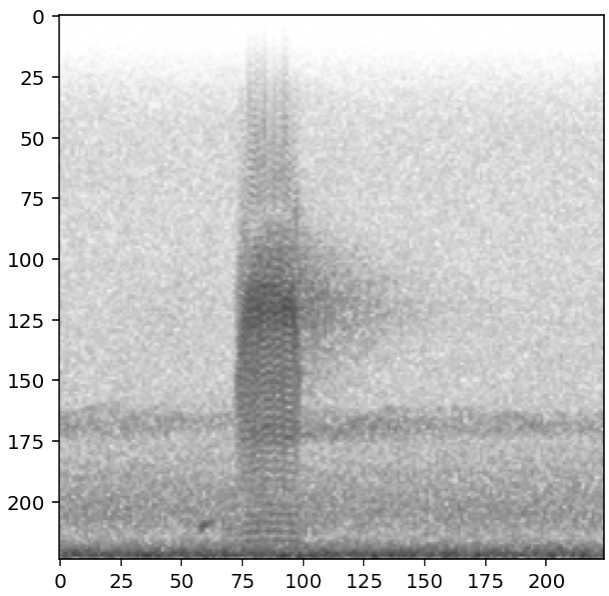
[42]:
```
preprocessor.actions.overlay.set(
    overlay_class='positive',
    overlay_weight=0.4
)
show_tensor(preprocessor[0])
```



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

[43]:
```
preprocessor.actions.overlay.set(overlay_class=None)
show_tensor(preprocessor[0])
```



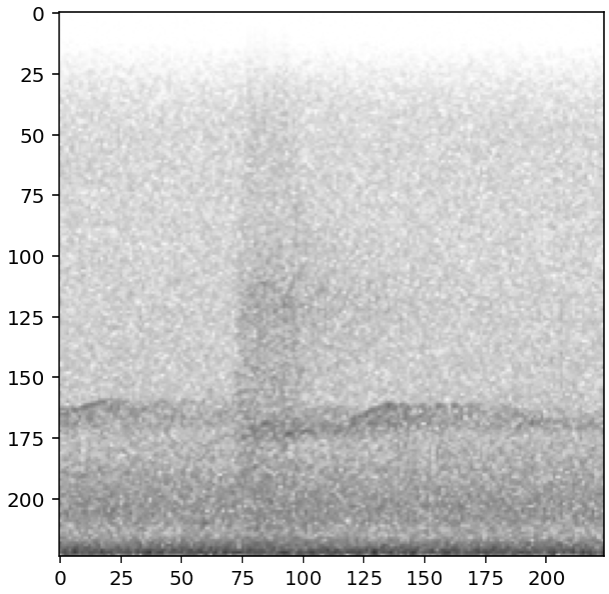
Overlaying samples from a "different" class

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%).

[44]:
```
preprocessor.actions.overlay.set(update_labels=False,overlay_class='different',
↪overlay_weight=0.8)
show_tensor(preprocessor[0])
```



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`.

[45]:
```
print('default: labels do not update')
preprocessor.actions.overlay.set(update_labels=False,overlay_class='different')
print(f"\t resulting labels: {preprocessor[0]['y'].numpy()}")

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

```
default: labels do not update
         resulting labels: [0 1]
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'`.

9.8 Creating a new Preprocessor class

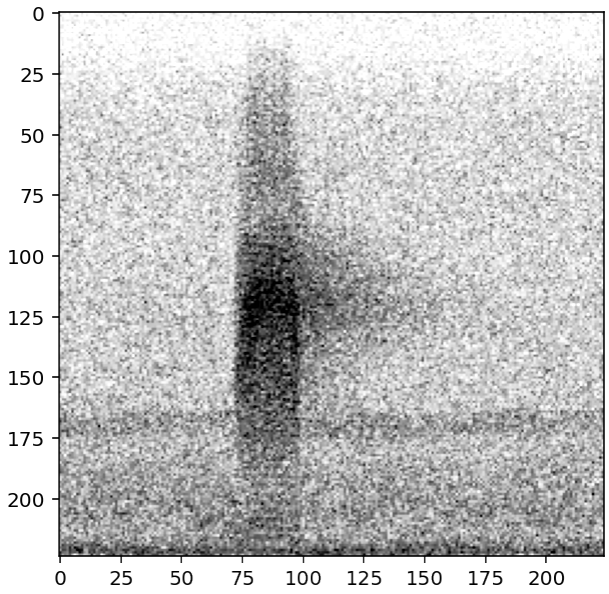
If you have a specific augmentation routine you want to perform, you may want to create your own Preprocessor class rather than modifying an existing one.

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

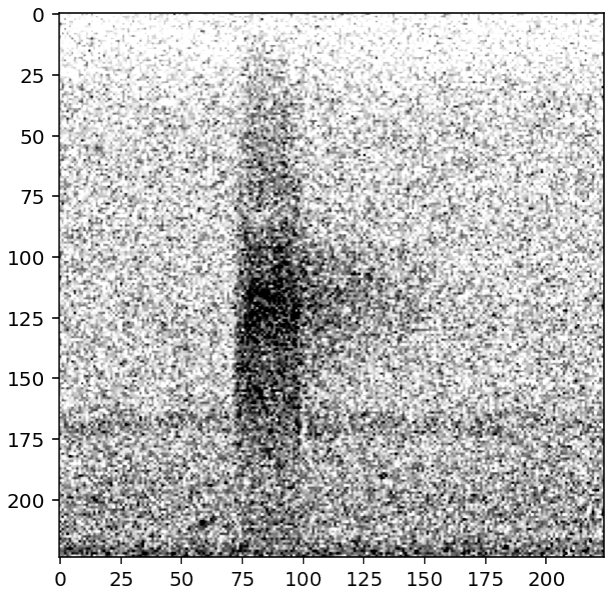
Here's an example of a customized preprocessor that subclasses `AudioToSpectrogramPreprocessor` and creates a pipeline that depends on the `magic_parameter` input.

[46]:
```
from opensoundscape.preprocess.actions import TensorAddNoise
class MyPreprocessor(AudioToSpectrogramPreprocessor):
    """Child of AudioToSpectrogramPreprocessor with weird augmentation routine"""

    def __init__(
        self,
        df,
        magic_parameter,
        audio_length=None,
        return_labels=True,
        out_shape=[224, 224],
    ):

        super(MyPreprocessor, self).__init__(
            df,
            audio_length=audio_length,
            out_shape=out_shape,
            return_labels=return_labels,
        )

        self.actions.add_noise = TensorAddNoise(std=0.1*magic_parameter)

        self.pipeline = [
            self.actions.load_audio,
            self.actions.trim_audio,
            self.actions.to_spec,
            self.actions.bandpass,
            self.actions.to_img,
            self.actions.to_tensor,
            self.actions.normalize,
        ] + [self.actions.add_noise for i in range(magic_parameter)]
```

[47]:
```
p = MyPreprocessor(labels, magic_parameter=2)
show_tensor(p[0])
```



[48]:
```
p = MyPreprocessor(labels, magic_parameter=3)
show_tensor(p[0])
```



9.9 Defining new Actions

You can define new Actions to include in your Preprocessor pipeline. They should subclass `opensoundscape.actions.BaseAction`.

You will need to define a `.go()` method for all actions. If you provide default parameter values, you will also need to define an `__init__()` method.

9.9.1 Without default parameters

If the Action does not need to have default arguments, it's trivial to create by defining a `go()` method.

[49]:
```
from opensoundscape.preprocess.actions import BaseAction
class SquareSamples(BaseAction):
    """Square values of every audio sample

    Audio in, Audio out
    """
    def go(self, audio):
        samples = np.array(audio.samples)**2
        return Audio(samples, audio.sample_rate)
```

Test it out:

[50]:
```
from opensoundscape.audio import Audio

square_action = SquareSamples(threshold=0.2)

audio = Audio.from_file('./woodcock_labeled_data/01c5d0c90bd4652f308fd9c73feb1bf5.wav
↪')
print(np.mean(audio.samples))
audio = square_action.go(audio)
print(np.mean(audio.samples))
```

```
0.012753859
0.008748752
```

9.9.2 With default parameters

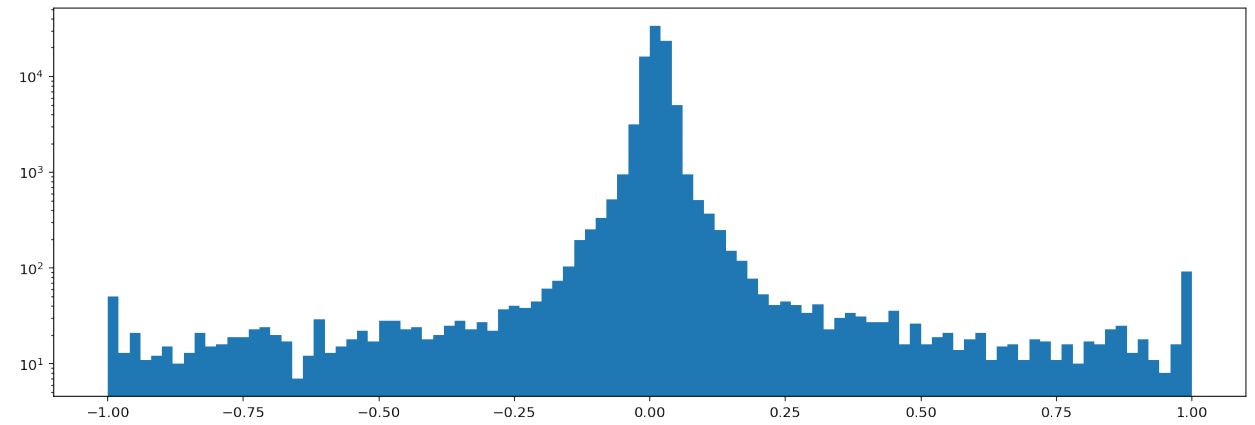
Here we overwrite the `__init__` method to provide a default parameter value. The Action below removes low-amplitude audio samples, acting somewhat as a "denoiser".

[51]:
```
class AudioGate(BaseAction):
    """Replace audio samples below a threshold with 0

    Audio in, Audio out

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

    def __init__(self, **kwargs):
        super(AudioGate, self).__init__(**kwargs)

        # default parameters
        self.params["threshold"] = 0.1

        # update/add any parameters passed to __init__
        self.params.update(kwargs)

    def go(self, audio):
        samples = np.array([0 if np.abs(s)<self.params["threshold"] else s for s in␣
↪audio.samples])
        return Audio(samples, audio.sample_rate)
```

Test it out:

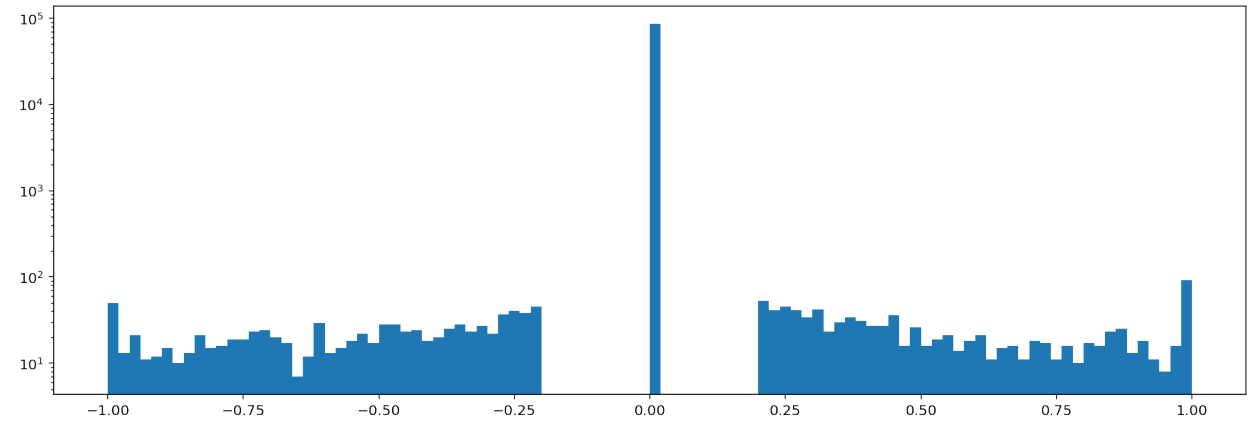
[52]:
```
gate_action = AudioGate(threshold=0.2)

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)
_ = plt.hist(audio_gated.samples,bins=100)
plt.semilogy()
```

```
histogram of samples
```




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

[52]:
```
[]
```



Clean up files created during this tutorial:

[54]:
```
folder = Path('./woodcock_labeled_data')
[p.unlink() for p in folder.glob("*")]
folder.rmdir()
```

CHAPTER 10

Custom CNN training

This notebook demonstrates how to use `opensoundscape.torch.cnn` classes to

- schedule the learning rate decay
- choose from various architectures
- use strategic sampling for imbalanced training data
- 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 "basic training and prediction with CNNs" tutorial (cnn.ipynb).

[1]:
```
from opensoundscape.preprocess.preprocessors import BasePreprocessor,␣
↪AudioToSpectrogramPreprocessor, CnnPreprocessor
from opensoundscape.torch.models.cnn import PytorchModel, Resnet18Multiclass,␣
↪Resnet18Binary, InceptionV3
from opensoundscape.helpers import run_command

import torch
import pandas as pd
from pathlib import Path
import numpy as np
import pandas as pd
import random

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

If you already have these files, you can skip or comment out this cell

[2]:
```
commands = [
    "curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -
↪o ./woodcock_labeled_data.tar.gz",
    "tar -xzf woodcock_labeled_data.tar.gz", # Unzip the downloaded tar.gz file
    "rm woodcock_labeled_data.tar.gz" # Remove the file after its contents are␣
↪unzipped
]
for command in commands:
    run_command(command)
```

10.1.2 Create one-hot encoded labels

See the "Basic training and prediction with CNNs" tutorial for more details.

The audio data includes 2s long audio clips taken from an autonomous recording unit and a CSV of labels. We manipulate the label dataframe to give "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. 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).

For more details on the steps below, see the "basic training and prediction with CNNs" tutorial.

[3]:
```
labels = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))
labels.filename = ['./woodcock_labeled_data/'+f for f in labels.filename]

labels['negative']=[0 if label=='present' else 1 for label in labels['woodcock']]
labels['positive']=[1 if label=='present' else 0 for label in labels['woodcock']]
classes=['negative','positive']
labels = labels.set_index('filename')[classes]
labels.head()
```

[3]:

filename	negative	positive
./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

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)
# for multi-class need at least a few images for each batch
len(train_df)
```

[4]:
```
23
```

10.1.4 Create Preprocessors

Preprocessors take the audio data specified by the dataframe created above and prepare it for use by Pytorch, e.g., creating spectrograms and performing augmentation. For more detail, see the "Basic training and prediction with CNNs" tutorial and the "Custom preprocessors" tutorial.

[5]:
```
from opensoundscape.preprocess.preprocessors import CnnPreprocessor

train_dataset = CnnPreprocessor(train_df, overlay_df=train_df)

valid_dataset = CnnPreprocessor(valid_df, overlay_df=valid_df, return_labels=True)
```

10.2 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.

[6]:
```
from opensoundscape.torch.models.cnn import Resnet18Binary
model = Resnet18Binary(classes)
model.optimizer_params
```

```
created PytorchModel model object with 2 classes
```

[6]:
```
{'feature': {'params': <generator object Module.parameters at 0x7f956d634970>,
  'lr': 0.001,
  'momentum': 0.9,
  'weight_decay': 0.0005},
 'classifier': {'params': <generator object Module.parameters at 0x7f956d634ac0>,
  'lr': 0.01,
  'momentum': 0.9,
  'weight_decay': 0.0005}}
```

10.2.1 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.

In `Resnet18Multiclass` and `Resnet18Binary`, 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 features and slower one for classifiers (though this might not be a good idea in practice):

[7]:
```
model = Resnet18Binary(classes)
model.optimizer_params['feature']['lr'] = 0.01
model.optimizer_params['classifier']['lr'] = 0.001
```

```
created PytorchModel model object with 2 classes
```

10.2.2 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.

[8]:
```
model.lr_cooling_factor = 0.1
model.lr_update_interval = 1
```

10.2.3 Regularization weight decay

The `Resnet18` classes 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.

[9]:
```
model.optimizer_params['feature']['weight_decay']=0.001
model.optimizer_params['classifier']['weight_decay']=0.001
```

10.2.4 Pretrained weights

In OpenSoundscape, most architectures implemented have the ability to use weights pretrained on the ImageNet image database turned on by default. It takes some time to download these weights the first time an instance of a model is created with pretrained weights.

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."

Currently, this feature cannot be turned off in the `Resnet18` classes. However, if you prefer, you can turn this off in many classes when creating a custom architecture (see "Network architectures" below) by changing the `use_pretrained` argument to `False`, e.g.:

[10]:
```
# See "InceptionV3 architecture" section below for more information
model = InceptionV3(classes, use_pretrained=False)
```

```
/Users/tessa/Library/Caches/pypoetry/virtualenvs/opensoundscape-dxMTH98s-py3.8/lib/
↪python3.8/site-packages/torchvision/models/inception.py:80: 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 '
```

```
created PytorchModel model object with 2 classes
```

10.2.5 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. To do so, set the `freeze_feature_extractor` argument to `True`. Below, we set the `use_pretrained` argument to `False` to avoid downloading the weights.

[11]:
```
# See "InceptionV3 architecture" section below for more information
model = InceptionV3(classes, freeze_feature_extractor=True, use_pretrained=False)
```

```
created PytorchModel model object with 2 classes
```

10.3 Network architecture

It is possible to use a different model architecture than ResNet18. The `opensoundscape.torch.models.cnn <https://github.com/kitzeslab/opensoundscape/blob/master/opensoundscape/torch/models/cnn.py>`__ module contains two types of classes for doing so: * Custom classes for both the ResNet18 architecture (`Resnet18Binary` and `Resnet18Multiclass`) and the InceptionV3 architecture (`InceptionV3` and `InceptionV3ResampleLoss`). * The `PytorchModel` class, which allows us to create a model with a different CNN architecture. The available architectures are listed in `opensoundscape.torch.architectures.cnn_architectures <https://github.com/kitzeslab/opensoundscape/blob/master/opensoundscape/torch/architectures/cnn_architectures.py>`__.

Below, we demonstrate the use of InceptionV3, how to create instances of other architectures, how to change the architecture on a model.

10.3.1 InceptionV3 architecture

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.

The InceptionV3 class in `cnn` handles the necessary modifications in training and prediction for you, but you'll need to make sure to pass images of the correct shape from your Preprocessor. Here's an example:

[12]:
```
from opensoundscape.torch.models.cnn import InceptionV3

#generate an Inception model
model = InceptionV3(classes=['negative','positive'],use_pretrained=False)

#create a copy of the training dataset
inception_dataset = train_dataset.sample(frac=1)

#modify the preprocessor to give 299x299 image shape
inception_dataset.actions.to_img.set(shape=[299,299])

#train and validate for 1 epoch
#note that Inception will complain if batch_size=1
model.train(inception_dataset,inception_dataset,epochs=1,batch_size=4)

#predict
preds, _, _ = model.predict(inception_dataset)
```

```
created PytorchModel model object with 2 classes
Epoch: 0 [batch 0/6 (0.00%)]
        Jacc: 0.500 Hamm: 0.500 DistLoss: 1.103

Validation.
(23, 2)
         Precision: 0.391304347826087
         Recall: 0.5
         F1: 0.4390243902439025
Saving weights, metrics, and train/valid scores.
Saving to epoch-0.model
Updating best model
Saving to best.model

Best Model Appears at Epoch 0 with F1 0.439.
(23, 2)
```

10.3.2 Pytorch stock architectures

The `opensoundscape.torch.architectures.cnn_architectures` module provides helper functions to generate various CNN architectures in Pytorch. These are well-known CNN architectures that Pytorch provides out of the box. The architectures provided include:

- Other ResNet types (resnet34, resnet50, resnet101, resnet152)
- AlexNet
- Vgg11
- Squeezenet
- Densenet121

Also implemented are ResNet18 and InceptionV3, but in most cases, you should use the pre-implemented classes for those instead of loading them into a `PytorchModel`.

Calling a function from this module, e.g. `alexnet()`, will return a CNN architecture that we can use to instantiate a `PytorchModel`.

Below and in the following examples, we set `use_pretrained=False` to avoid downloading all of the weights for these models.

[13]:
```
from opensoundscape.torch.architectures.cnn_architectures import alexnet
from opensoundscape.torch.models.cnn import PytorchModel

#initialize the AlexNet architecture
arch = alexnet(num_classes=2, use_pretrained=False)

#generate a model object with this architecture
model = PytorchModel(architecture=arch, classes=['negative','positive'])
```

```
created PytorchModel model object with 2 classes
```

10.3.3 Changing the architecture of an existing model

Even after initializing a model with an architecture, we can change it by replacing the model's `.network`:

[14]:
```
from opensoundscape.torch.architectures.cnn_architectures import densenet121

#initialize the AlexNet architecture
arch = densenet121(num_classes=2, use_pretrained=False)

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

10.3.4 Use a custom-built architecture

You can also build a custom architecture and initialize a PytorchModel model with it, or replace a model's `.network` with your custom architecture.

For example, we can use the `architectures.resnet` module to build the ResNet50 architecture (just for demonstration - we could also simply create this architecture using the `resnet50()` function in the `cnn_architectures` module).

[15]:
```
# import a module that builds ResNet architecture from scratch
from opensoundscape.torch.architectures.resnet import ResNetArchitecture

#initialize the ResNet50 architecture
net=ResNetArchitecture(
    num_cls=2,
    weights_init='ImageNet',
    num_layers=50,
)

#generate a regular resnet18 object
model = Resnet18Multiclass(classes=['negative','positive'])

#replace the model's network with the ResNet50 architecture
model.network = net

print('number of layers:')
print(model.network.num_layers)
```

```
created PytorchModel model object with 2 classes
number of layers:
50
```

10.4 Sampling for imbalanced training data

The imbalanced data sampler will help to ensure that a single batch contains only a few classes during training, and that the classes will recieve approximately equal representation within the batch. This is useful for *imbalanced* training data (when some classes have far fewer training samples than others).

[16]:
```
model = Resnet18Binary(classes)
model.sampler = 'imbalanced' #default is None

#...you can now train your model as normal
model.train(train_dataset, valid_dataset, epochs=0)

#once we run train(), we can see that the train_loader is using an␣
↪ImbalancedDatasetSampler
print('sampler:')
model.train_loader.sampler
```

```
created PytorchModel model object with 2 classes

Best Model Appears at Epoch 0 with F1 0.000.
sampler:
```

[16]:
```
<opensoundscape.torch.sampling.ImbalancedDatasetSampler at 0x7f953e04a760>
```

10.5 Training 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.

10.5.1 Bandpassed spectrograms

[17]:
```
model = Resnet18Binary(classes)

# turn on the bandpass action of the datasets
train_dataset.actions.bandpass.on()
valid_dataset.actions.bandpass.on()

# specify the min and max frequencies for the bandpass action
train_dataset.actions.bandpass.set(min_f=3000, max_f=5000)
valid_dataset.actions.bandpass.set(min_f=3000, max_f=5000)

# now we can train and validate on the bandpassed spectrograms
# don't forget that you'll need to apply the same bandpass actions to
# any datasets that you use for predicting on new audio files
model.train(train_dataset, valid_dataset, epochs=0)
```

```
created PytorchModel model object with 2 classes

Best Model Appears at Epoch 0 with F1 0.000.
```

10.5.2 clean up

remove files

[22]:
```
folder = Path('./woodcock_labeled_data')
[p.unlink() for p in folder.glob("*")]
folder.rmdir()
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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

11.2.1 Load an audio file and create a spectrogram

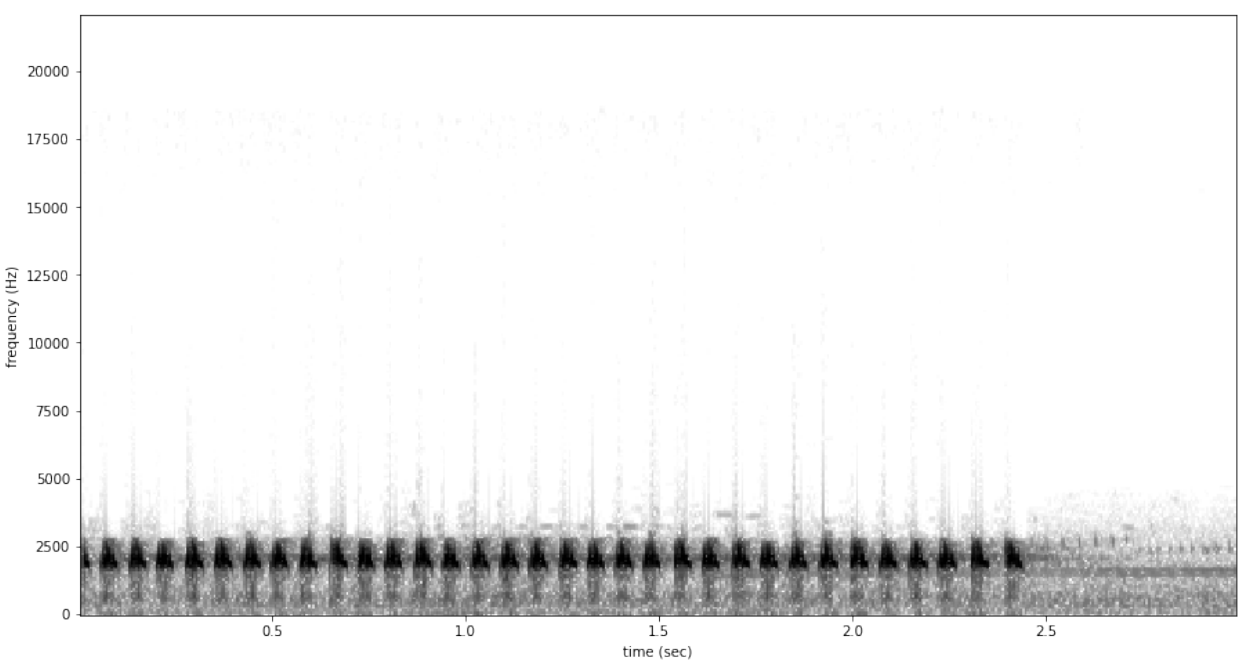
[3]:
```
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

#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)

#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()
```

```
/Users/tessa/opt/anaconda3/envs/opso_0.4.6/lib/python3.7/site-packages/ipykernel/
↪ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
↪cell` automatically in the future. Please pass the result to `transformed_cell`␣
↪argument and any exception that happen during thetransform in `preprocessing_exc_
↪tuple` in IPython 7.17 and above.
  and should_run_async(code)
```



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.

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

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

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

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

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

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

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

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.

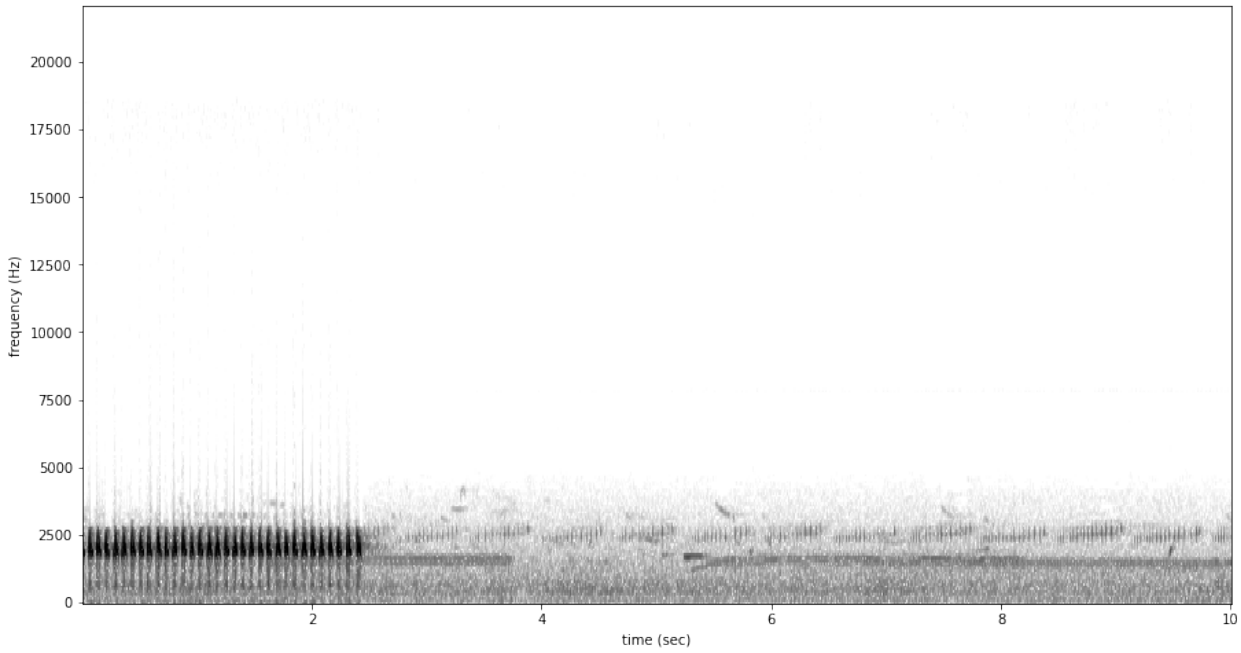
[6]:
```
#get the audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

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

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


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

(continues on next page)

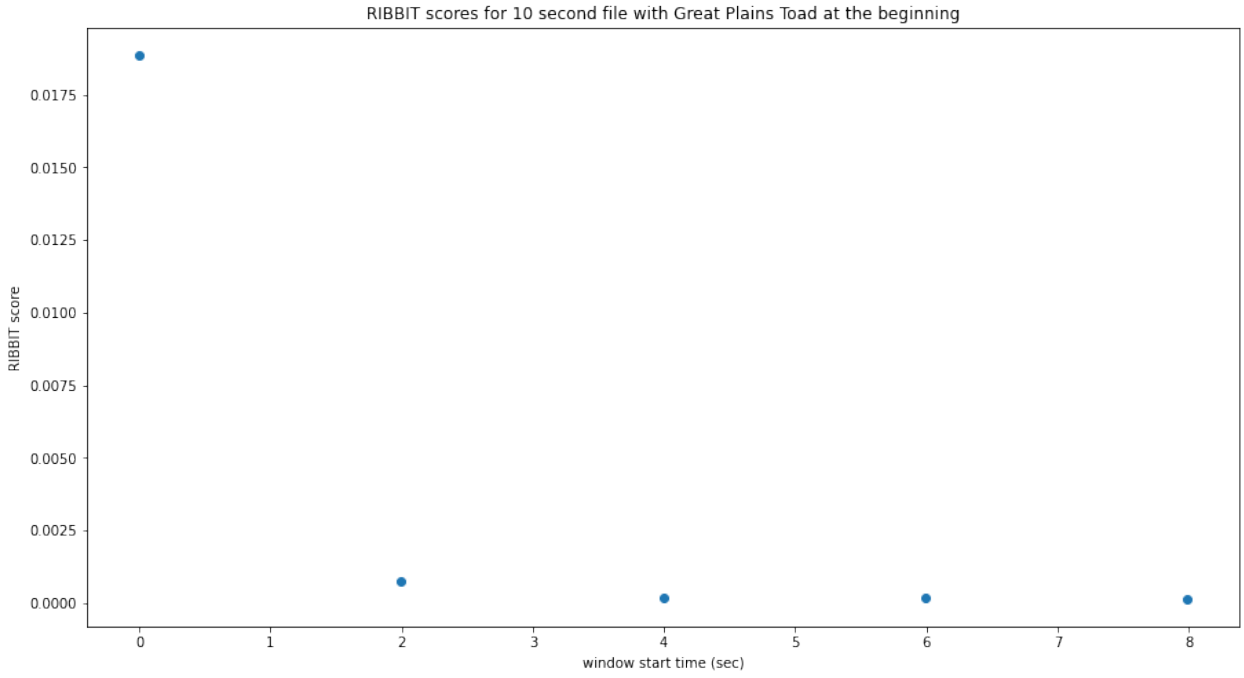
(continued from previous page)

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

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




```
[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
    scores, times =  ribbit(
                        spec,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        window_len=window_length,
                        noise_bands=noise_bands,
                        plot=False)

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

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

```
/Users/tessa/opt/anaconda3/envs/opso_0.4.6/lib/python3.7/site-packages/ipykernel/
↪ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
↪cell` automatically in the future. Please pass the result to `transformed_cell`␣
↪argument and any exception that happen during thetransform in `preprocessing_exc_
↪tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

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

[7]:

	score	label
./great_plains_toad_dataset/gpt0.mp3	188.681233	1
./great_plains_toad_dataset/gpt3.mp3	27.355522	1
./great_plains_toad_dataset/negative3.mp3	21.268281	0
./great_plains_toad_dataset/negative5.mp3	17.663214	0
./great_plains_toad_dataset/negative8.mp3	16.936452	0
./great_plains_toad_dataset/pops2.mp3	14.115037	0
./great_plains_toad_dataset/gpt4.mp3	13.923912	1
./great_plains_toad_dataset/gpt2.mp3	13.799077	1
./great_plains_toad_dataset/negative1.mp3	9.517518	0
./great_plains_toad_dataset/pops1.mp3	8.946919	0
./great_plains_toad_dataset/negative9.mp3	8.659933	0
./great_plains_toad_dataset/negative4.mp3	7.905783	0

(continues on next page)

(continued from previous page)

```
./great_plains_toad_dataset/negative7.mp3    7.726107      0
./great_plains_toad_dataset/gpt1.mp3         7.346534      1
./great_plains_toad_dataset/negative2.mp3    5.739785      0
./great_plains_toad_dataset/negative6.mp3     5.69147      0
./great_plains_toad_dataset/water.mp3        4.409431      0
./great_plains_toad_dataset/silent.mp3       0.866457      0
```

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

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

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

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

11.6 Detail view

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

[8]:
```
#create a spectrogram from the file, like above:
# 1. get audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]
# 2. make audio object and trim (this time 0-10 seconds)
audio = Audio.from_file(audio_path).trim(0,10)
# 3. make spectrogram
spectrogram = Spectrogram.from_audio(audio)

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

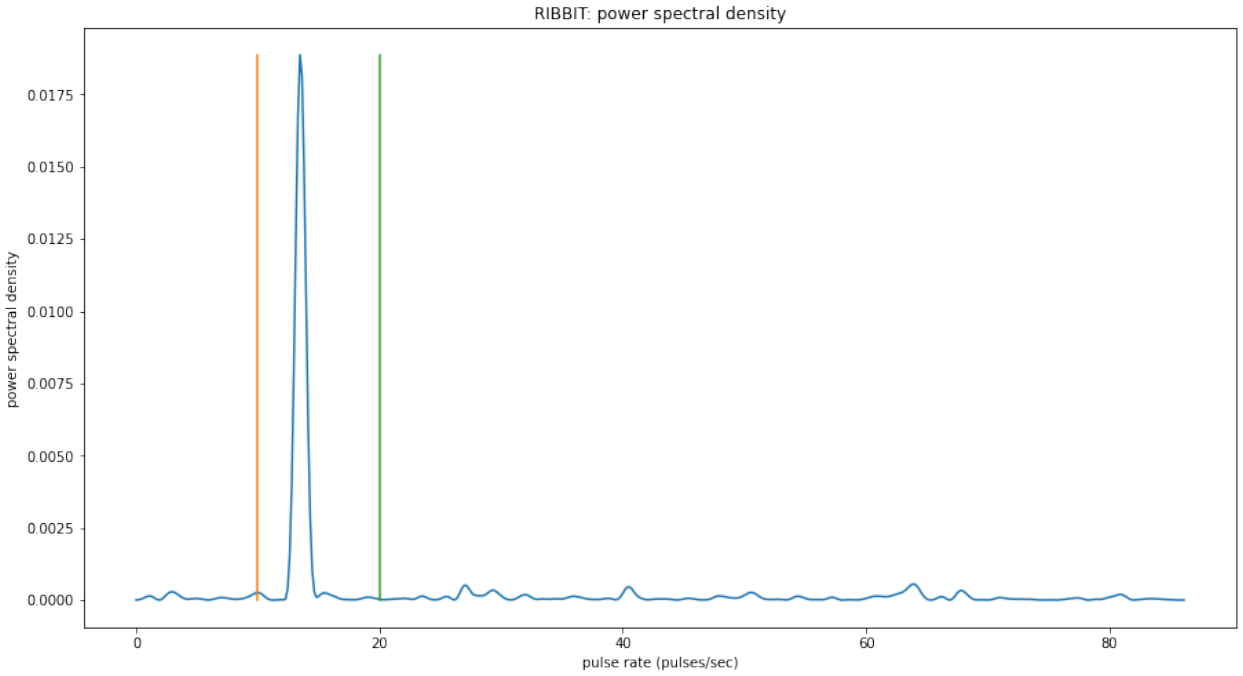
```
window: 0.0000 sec to 1.9969 sec
```

```
/Users/tessa/opt/anaconda3/envs/opso_0.4.6/lib/python3.7/site-packages/ipykernel/
↪ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
↪cell` automatically in the future. Please pass the result to `transformed_cell`␣
↪argument and any exception that happen during thetransform in `preprocessing_exc_
↪tuple` in IPython 7.17 and above.
```

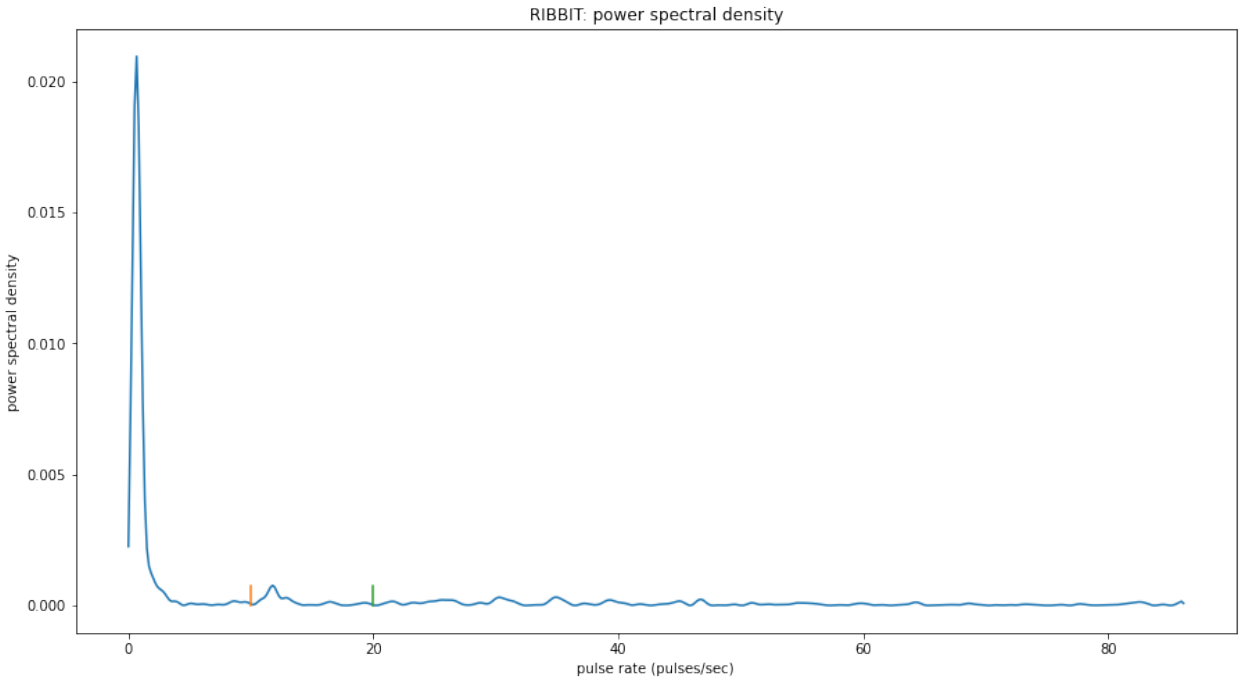
(continues on next page)

(continued from previous page)

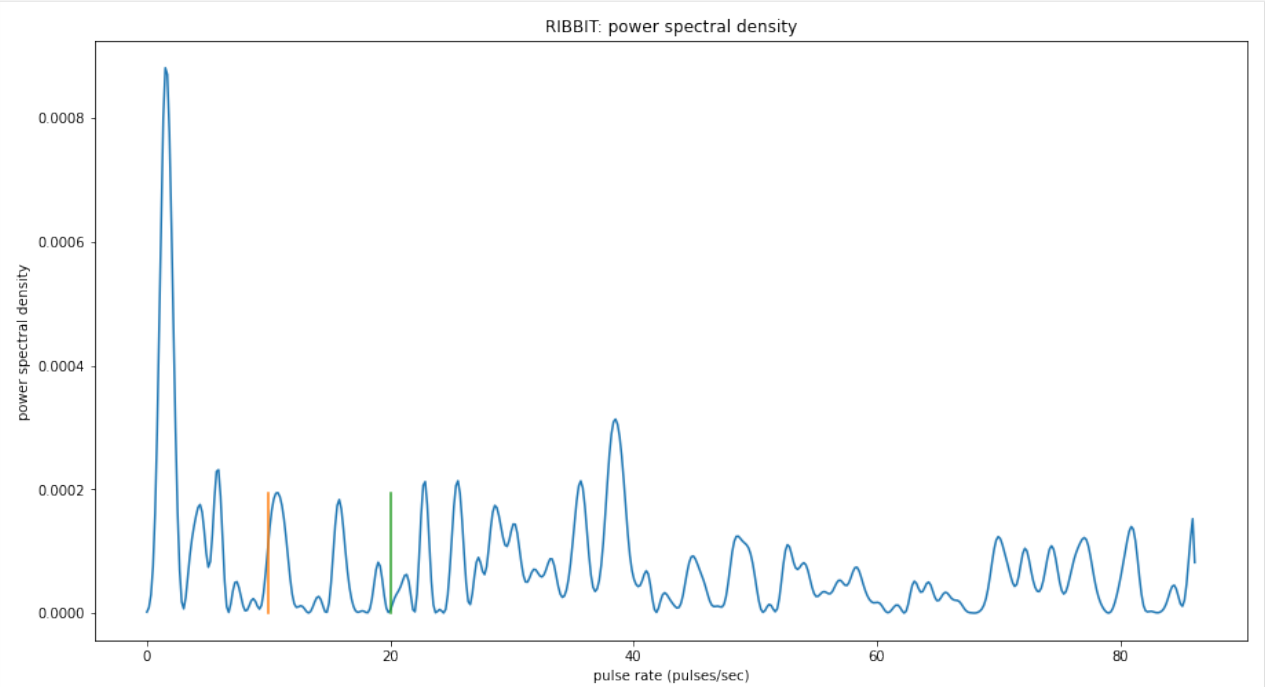
```
  and should_run_async(code)
```



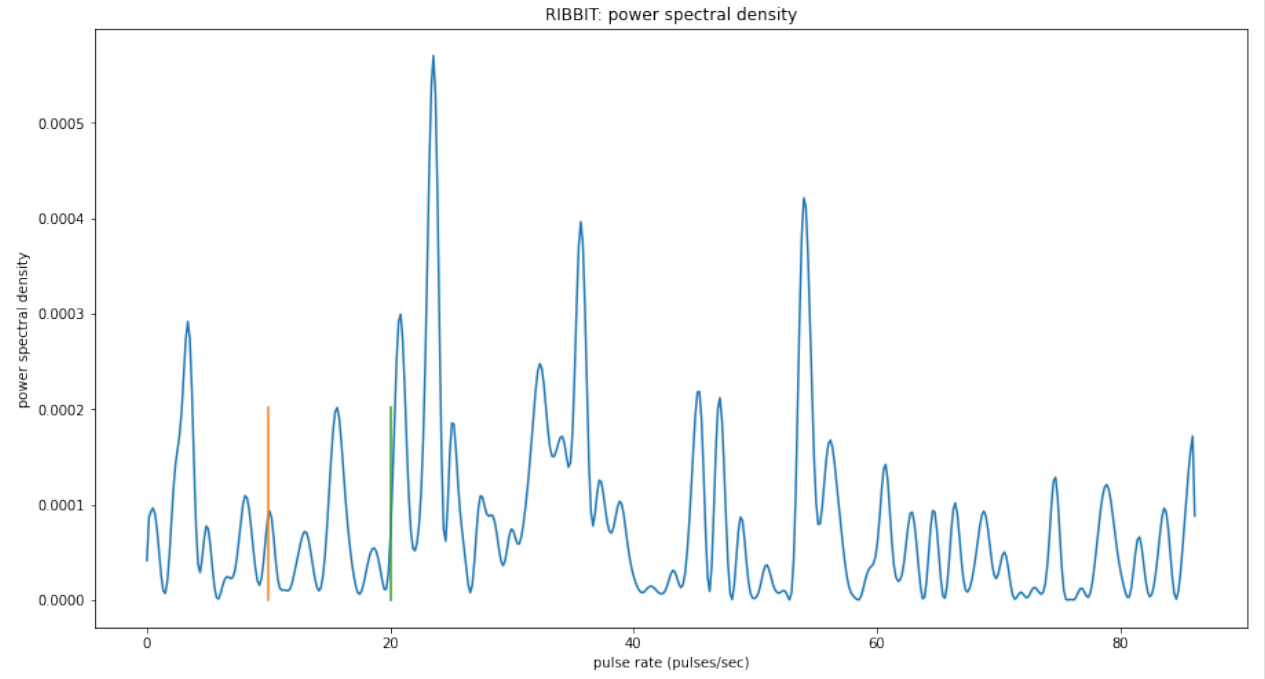

```
window: 1.9969 sec to 3.9938 sec
```




```
window: 3.9938 sec to 5.9907 sec
```




```
window: 5.9907 sec to 7.9877 sec
```



11.7 Time to experiment for yourself

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

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

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

[9]:
```
_ = run_command('rm -r ./great_plains_toad_dataset/')
_ = run_command('rm ./great_plains_toad_dataset.tar.gz')
```

```
/Users/tessa/opt/anaconda3/envs/opso_0.4.6/lib/python3.7/site-packages/ipykernel/
↪ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
↪cell` automatically in the future. Please pass the result to `transformed_cell`␣
↪argument and any exception that happen during thetransform in `preprocessing_exc_
↪tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

CHAPTER 12

Annotations

12.1 Raven

raven.py: Utilities for dealing with Raven files

opensoundscape.raven.**annotation_check**(*directory*, *col*)
Check that rows of Raven annotations files contain class labels

Parameters

- **directory** – The path which contains Raven annotations file(s)
- **col** – Name of column containing annotations

Returns None

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

Parameters

- **directory** – The path which contains lowercase Raven annotations file(s)
- **col** – Name of column containing annotations

Returns

A multiline string containing a CSV file with two columns *raw* and *corrected*

Return type csv (string)

opensoundscape.raven.**generate_split_labels_file**(*directory*, *col*, *split_len_s*, *total_len_s=None*, *species=None*, *out_csv=None*)
Generate binary labels for a directory of Raven annotations

Given a directory of lowercase Raven annotations, splits the annotations into segments that can be used as labels for machine learning programs that only take short segments.

Parameters

- **directory** – The path which contains lowercase Raven annotations file(s)
- **col** (*str*) – name of column in Raven file to look for annotations in
- **split_len_s** (*int*) – length of segments to break annotations into (e.g. for 5s: 5)
- **total_len_s** (*float*) – length of original files (e.g. for 5-minute file: 300). If not provided, estimates length individually for each file based on end time of last annotation [default: None]
- **species** (*str, list, or None*) – species or list of species annotations to look for [default: None]
- **out_csv** (*str) (optional*) – None]

Returns

split file of the format filename, start_seg, end_seg, species1, species2, ..., speciesN orig/fname1, 0, 5, 0, 1, ..., 1 orig/fname1, 5, 10, 0, 0, ..., 1 orig/fname2, 0, 5, 1, 1, ..., 1 ...

saves all_selections to out_csv if this is specified

Return type all_selections (pd.DataFrame)

opensoundscape.raven.**get_labels_in_dataset**(*selections_files*, *col*)

Get list of all labels in selections_files

Parameters

- **selections_files** (*list*) – list of Raven selections.txt files
- **col** (*str*) – the name of the column containing the labels

Returns a list of the unique values found in the label column of this dataset

opensoundscape.raven.**lowercase_annotations**(*directory*, *out_dir=None*)

Convert Raven annotation files to lowercase and save

Parameters

- **directory** – The path which contains Raven annotations file(s)
- **out_dir** – The path at which to save (default: save in *directory*, same location as annotations) [default: None]

Returns None

opensoundscape.raven.**query_annotations**(*directory*, *cls*, *col*, *print_out=False*)

Given a directory of Raven annotations, query for a specific class

Parameters

- **directory** – The path which contains lowercase Raven annotations file(s)
- **cls** – The class which you would like to query for
- **col** – Name of column containing annotations
- **print_out** –

  Format of output. If True, output contains delimiters. If False, returns output

  [default: False]

Returns A multiline string containing annotation file and rows matching the query cls

Return type output (string)

opensoundscape.raven.**raven_audio_split_and_save**(*raven_directory, audio_directory, destination, col, sample_rate, clip_duration, clip_overlap=0, final_clip=None, extensions=['wav', 'WAV', 'mp3'], csv_name='labels.csv', labeled_clips_only=False, min_label_len=0, species=None, dry_run=False, verbose=False*)

Split audio and annotations files simultaneously

Splits audio into short clips with the desired overlap. Saves these clips and a one-hot encoded labels CSV into the directory of choice. Labels for csv are selected based on all labels in clips.

Requires that audio and annotation filenames are unique, and that the "stem" of annotation filenames is the same as the corresponding stem of the audio filename (Raven saves files using this convention by default).

E.g. The following format is correct: audio_directory/audio_file_1.wav raven_directory/audio_file_1.Table.1.selections.txt

Parameters

- **raven_directory** (*str or pathlib.Path*) – The path which contains lowercase Raven annotations file(s)
- **audio_directory** (*str or pathlib.Path*) – The path which contains audio file(s) with names the same as annotation files
- **destination** (*str or pathlib.Path*) – The path at which to save the splits and the one-hot encoded labels file
- **col** (*str*) – The column containing species labels in the Raven files
- **sample_rate** (*int*) – Desired sample rate of split audio clips
- **clip_duration** (*float*) – Length of each clip
- **clip_overlap** (*float*) – Amount of overlap between subsequent clips [default: 0]
- **final_clip** (*str or None*) – 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),
  - "full": Increase the overlap with previous audio to yield a clip with clip_duration length
  - "remainder": Include the remainder of the Audio (clip will NOT have clip_duration length)
  - "extend": Similar to remainder but extend the clip with silence to reach clip_duration length
  - "loop": Similar to remainder but loop (repeat) the clip to reach clip_duration length
- **extensions** (*list*) – List of audio filename extensions to look for. [default: *['wav', 'WAV', 'mp3']*]
- **csv_name** (*str*) – Filename of the output csv, to be saved in the specified destination [default: 'labels.csv']
- **min_label_len** (*float*) – the minimum amount a label must overlap with the split to be considered a label. Useful for excluding short annotations or annotations that barely overlap the split. For example, if 1, the label will only be included if the annotation is at least 1s long and either starts at least 1s before the end of the split, or ends at least 1s after the start of the split. By default, any label is kept [default: 0]

- **labeled_clips_only** (*bool*) – Whether to only save clips that contain labels of the species of interest. [default: False]
- **species** (*str, list, or None*) – Species labels to get. If None, gets a list of labels from all selections files. [default: None]
- **dry_run** (*bool*) – If True, skip writing audio and just return clip DataFrame [default: False]
- **verbose** (*bool*) – If True, prints progress information [default:False]

Returns:

opensoundscape.raven.**split_single_annotation**(*raven_file*, *col*, *split_len_s*, *overlap_len_s=0*, *total_len_s=None*, *keep_final=False*, *species=None*, *min_label_len=0*)

Split a Raven selection table into short annotations

Aggregate one-hot annotations for even-lengthed time segments, drawing annotations from a specified column of a Raven selection table

Parameters

- **raven_file** (*str*) – path to Raven selections file
- **col** (*str*) – name of column in Raven file to look for annotations in
- **split_len_s** (*float*) – length of segments to break annotations into (e.g. for 5s: 5)
- **overlap_len_s** (*float*) – length of overlap between segments (e.g. for 2.5s: 2.5)
- **total_len_s** (*float*) – length of original file (e.g. for 5-minute file: 300) If not provided, estimates length based on end time of last annotation [default: None]
- **keep_final** (*string*) – whether to keep annotations from the final clip if the final clip is less than split_len_s long. If using "remainder", "full", "extend", or "loop" with split_and_save, make this True. Else, make it False. [default: False]
- **species** (*str, list, or None*) – species or list of species annotations to look for [default: None]
- **min_label_len** (*float*) – the minimum amount a label must overlap with the split to be considered a label. Useful for excluding short annotations or annotations that barely overlap the split. For example, if 1, the label will only be included if the annotation is at least 1s long and either starts at least 1s before the end of the split, or ends at least 1s after the start of the split. By default, any label is kept [default: 0]

Returns

columns 'seg_start', 'seg_end', and all species, each row containing 1/0 annotations for each species in a segment

Return type splits_df (pd.DataFrame)

opensoundscape.raven.**split_starts_ends**(*raven_file*, *col*, *starts*, *ends*, *species=None*, *min_label_len=0*)

Split Raven annotations using a list of start and end times

This function takes an array of start times and an array of end times, creating a one-hot encoded labels file by finding all Raven labels that fall within each start and end time pair.

This function is called by *split_single_annotation()*, which generates lists of start and end times. It is also called by *raven_audio_split_and_save()*, which gets the lists from metadata about audio files split by opensoundscape.audio.split_and_save.

Parameters

- **raven_file** (*pathlib.Path or str*) – path to selections.txt file
- **col** (*str*) – name of column containing annotations
- **starts** (*list*) – start times of clips
- **ends** (*list*) – end times of clips
- **species** (*str or list*) – species names for columns of one-hot encoded file [default: None]
- **min_label_len** (*float*) – the minimum amount a label must overlap with the split to be considered a label. Useful for excluding short annotations or annotations that barely overlap the split. For example, if 1, the label will only be included if the annotation is at least 1s long and either starts at least 1s before the end of the split, or ends at least 1s after the start of the split. By default, any label is kept [default: 0]

Returns columns: 'seg_start', 'seg_end', and all unique labels ('species') rows: one per segment, containing 1/0 annotations for each potential label

Return type splits_df (pd.DataFrame)

12.2 Species Table

12.3 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

CHAPTER 13

Audio

13.1 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'*, *max_duration=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"]
- **max_duration** (*None or integer*) – The maximum duration in seconds allowed for the audio file (longer files will raise an exception)[default: None] If None, no limit is enforced

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 length in seconds of the extended file

Returns a new Audio object of the desired length

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

Read from bytesio object

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

Parameters

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

Returns An initialized Audio object

classmethod from_file(*path*, *sample_rate=None*, *resample_type='kaiser_fast'*, *max_duration=None*)

Load audio from files

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

Parameters

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

Returns attributes samples and sample_rate

Return type *Audio*

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:
  - "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

Returns ["audio", "begin_time", "end_time"]

Return type A list of dictionaries with keys

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

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

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

opensoundscape.audio.**split_and_save**(*audio*, *destination*, *prefix*, *clip_duration*, *clip_overlap=0*, *final_clip=None*, *dry_run=False*)
Split audio into clips and save them to a folder

Parameters

- **audio** – The input Audio to split
- **destination** – A folder to write clips to
- **prefix** – A name to prepend to the written clips
- **clip_duration** – The duration of each clip in seconds
- **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 begin and end times for each clip from the source audio

13.2 Audio Tools

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

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

Parameters

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

Returns filtered time signal

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

Parameters

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

Returns set of coefficients used in sosfiltfilt()

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

Parameters

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

Returns number of samples exceeding threshold

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

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

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

Parameters

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

Returns os response of ffmpeg command

opensoundscape.audio_tools.**mixdown_with_delays**(*files_to_mix*, *destination*, *delays=None*, *levels=None*, *duration='first'*, *verbose=0*, *create_txt_file=False*)
use ffmpeg to mixdown a set of audio files, each starting at a specified time (padding beginnings with zeros)

Parameters

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

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

Returns ffmpeg command line output

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

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

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

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

Parameters

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

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

If threshold is None: returns net_energy over background noise

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

Calculate audio energy with a sliding window

Calculate the energy in an array of audio samples

Parameters

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

Returns list of energy level (float) for each window

CHAPTER 14

Localization

opensoundscape.localization.**calc_speed_of_sound**(*temperature=20*)

Calculate speed of sound in meters per second

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

Parameters **temperature** – ambient temperature in Celsius

Returns the speed of sound in meters per second

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

Perform TDOA localization on a sound event

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

Parameters

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

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

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

Compute Lorentz inner product of two vectors

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

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

Or, for 2-dimensional case as

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

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 15

Machine Learning

15.1 PyTorch CNNs

classes for pytorch machine learning models in opensoundscape

For tutorials, see notebooks on opensoundscape.org

class opensoundscape.torch.models.cnn.**CnnResampleLoss**(*architecture*, *classes*, *single_target=False*)

Subclass of PytorchModel with ResampleLoss.

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

Parameters

- **architecture** – a model architecture object, for example one generated with the torch.architectures.cnn_architectures module
- **classes** – list of class names. Must match with training dataset classes.
- **single_target** –
  - True: model expects exactly one positive class per sample
  - False: samples can have an number of positive classes

  [default: False]

class opensoundscape.torch.models.cnn.**InceptionV3**(*classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *single_target=False*)

train_epoch()

perform forward pass, loss, backpropagation for one epoch

need to override parent because Inception returns different outputs from the forward pass (final and auxiliary layers)

Returns: (targets, predictions, scores) on training files

class opensoundscape.torch.models.cnn.**InceptionV3ResampleLoss**(*classes*, *freeze_feature_extractor=False*, *use_pretrained=True*, *single_target=False*)

class opensoundscape.torch.models.cnn.**PytorchModel**(*architecture*, *classes*, *single_target=False*)

Generic Pytorch Model with .train() and .predict()

flexible architecture, optimizer, loss function, parameters

for tutorials and examples see opensoundscape.org

methods include train(), predict(), save(), and load()

Parameters

- **architecture** – a model architecture object, for example one generated with the torch.architectures.cnn_architectures module
- **classes** – list of class names. Must match with training dataset classes.
- **single_target** –
  - True: model expects exactly one positive class per sample
  - False: samples can have an number of positive classes

  [default: False]

load(*path*, *load_weights=True*, *load_classifier_weights=True*, *load_optimizer_state_dict=True*, *verbose=False*)

load model and optimizer state_dict from disk

the object should be saved with model.save() which uses torch.save with keys for 'model_state_dict' and 'optimizer_state_dict'

Parameters

- **path** – where the file is saved
- **load_weights** – if False, ignore network weights [default:True]
- **load_classifier_weights** – if False, ignore classifier layer weights Use False to only load feature weights, eg to re-use trained cnn's feature extractor for new class [default: True]
- **load_optimizer_state_dict** – if False, ignore saved parameters for optimizer's state [default: True]
- **verbose** – if True, print missing and unused keys for model weights

predict(*prediction_dataset*, *batch_size=1*, *num_workers=0*, *activation_layer=None*, *binary_preds=None*, *threshold=0.5*, *error_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).

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

Parameters

- **prediction_dataset** – a Preprocessor or DataSset object that returns tensors, such as AudioToSpectrogramPreprocessor (no augmentation) or CnnPreprocessor (w/augmentation) from opensoundscape.datasets
- **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]
- **threshold** – prediction threshold for sigmoid scores. Only relevant when binary_preds == 'multi_target'
- **error_log** – if not None, saves a list of files that raised errors to the specified file location [default: None]

Returns: 3 DataFrames (or Nones), w/index matching prediciton_dataset.df scores: post-activation_layer scores predictions: 0/1 preds for each class labels: labels from dataset (if available)

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 selected for labels/scores/preds, returns None instead of a DataFrame in the returned tuple

save(*path=None*, *save_weights=True*, *save_optimizer=True*, *extras={}*)
save model with weights (default location is self.save_path)

Parameters

- **path** – destination for saved model. if None, uses self.save_path
- **save_weights** – if False, only save metadata/metrics [default: True]
- **save_optimizer** – if False, don't save self.optim.state_dict()
- **extras** – arbitrary dictionary of things to save, eg valid-preds

train(*train_dataset*, *valid_dataset*, *epochs=1*, *batch_size=1*, *num_workers=0*, *save_path='.'*, *save_interval=1*, *log_interval=10*, *unsafe_sample_log='./unsafe_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_dataset** – a Preprocessor that loads sample (audio file + label) to Tensor in batches (see docs/tutorials for details)
- **valid_dataset** – a Preprocessor for evaluating performance
- **epochs** – number of epochs to train for [default=1] (1 epoch constitutes 1 view of each training sample)

- **batch_size** – number of training files to load/process before re-calculating the loss function and backpropagation
- **num_workers** – parallelization (ie, cores or cpus) 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
- **unsafe_sample_log** – file path: log all samples that failed in preprocessing (file written when training completes) - if None, does not write a file

train_epoch()

perform forward pass, loss, backpropagation for one epoch

Returns: (targets, predictions, scores) on training files

class opensoundscape.torch.models.cnn.**Resnet18Binary**(*classes*)

Subclass of PytorchModel with Resnet18 architecture

This subclass allows separate training parameters for the feature extractor and classifier

Parameters

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

  [default: False]

class opensoundscape.torch.models.cnn.**Resnet18Multiclass**(*classes*, *single_target=False*)

Multi-class model with resnet18 architecture and ResampleLoss.

Can be single or multi-target.

Parameters

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

  [default: False]

Notes - Allows separate parameters for feature & classifier blocks

via self.optimizer_params's keys: "feature" and "classifier" (by using hand-built architecture)

- Uses ResampleLoss which requires class counts as an input.

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.**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.**get_dataloader**(*dataset*, *batch_size=64*, *num_workers=1*, *shuffle=False*, *sampler=''*)
Create a DataLoader from a DataSet - chooses between normal pytorch DataLoader and ImbalancedDataset-Sampler. - Sampler: None -> default DataLoader; 'imbalanced'->ImbalancedDatasetSampler

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(classes,my_arch)

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*)
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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

opensoundscape.torch.architectures.cnn_architectures.**set_parameter_requires_grad**(*model*, *freeze_feature*

if necessary, remove gradients of all model parameters

if freeze_feature_extractor is True, we set requires_grad=False for all features in the feature extraction block. We would do this if we have a pre-trained CNN and only want to change the shape of the final layer, then train only that final classification layer without modifying the weights of the rest of the network.

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

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.

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

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.

15.2 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

15.3 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.

implemented by @msamogh in nonechucks (github.com/msamogh/nonechucks/)

class opensoundscape.torch.safe_dataset.**SafeDataset**(*dataset*, *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__).

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.

15.4 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

15.5 Data Selection

opensoundscape.data_selection.**upsample**(*input_df*, *label_column='Labels'*, *random_state=None*)
Given a input DataFrame upsample to maximum value

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

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

15.6 Performance Metrics

opensoundscape.metrics.**binary_metrics**(*targets, preds, class_names=[0, 1]*)
labels should be single-target

opensoundscape.metrics.**multiclass_metrics**(*targets*, *preds*, *class_names*)
provide a list or np.array of 0,1 targets and predictions

opensoundscape.metrics.**predict**(*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]

15.7 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

CHAPTER 16

Miscellaneous

16.1 Splitter Dataset

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

A PyTorch Dataset for splitting a WAV files

Segments will be written to the *output_directory*

Parameters

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

Returns

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

Return type output

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

Parameters

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

Returns A dataframe of annotations which overlap with the begin/end times

Return type sub_df

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

Parameters **input_string** – An input string

Returns A string containing the md5 hash of input string

Return type output

16.2 Commands

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

Parameters **cmd** – A string containing some command

Returns A tuple of standard out and standard error

Return type (stdout, stderr)

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

Parameters **cmd** – A string containing some command

Returns The return code of the function

Return type return_code

16.3 Helpers

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

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

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

opensoundscape.helpers.**hex_to_time**(*s*)
convert a hexidecimal, Unix time string to a datetime timestamp 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.**isNan**(*x*)
check for nan by equating x to itself

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

Parameters

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

Returns x + random jitter

Return type jittered_x

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

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

Outputs: new_array: A translated array

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

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

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

returns rescaled feature set and rescaling vector

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

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

CHAPTER 17

Preprocessing

17.1 Preprocessing

class opensoundscape.preprocess.preprocessors.**AudioLoadingPreprocessor**(*df*, *return_labels=True*, *audio_length=None*)

creates Audio objects from file paths

Parameters

- **df** – dataframe of samples. df must have audio paths in the index. If df has labels, the class names should be the columns, and the values of each row should be 0 or 1. If data does not have labels, df will have no columns
- **return_labels** – if True, __getitem__ returns {"X":batch_tensors,"y":labels} if False, __getitem__ returns {"X":batch_tensors} [default: True]
- **audio_length** – length in seconds of audio to return - None: do not trim the original audio - seconds (float): trim longer audio to this length. Shorter audio input will raise a ValueError.

class opensoundscape.preprocess.preprocessors.**AudioToSpectrogramPreprocessor**(*df*, *audio_length=None*, *out_shape=[224, 224]*, *return_labels=True*)

loads audio paths, creates spectrogram, returns tensor

by default, resamples audio to sr=22050 can change with .actions.load_audio.set(sample_rate=sr)

Parameters

- **df** – dataframe of samples. df must have audio paths in the index. If df has labels, the class names should be the columns, and the values of each row should be 0 or 1. If data does not have labels, df will have no columns
- **audio_length** – length in seconds of audio clips [default: None] If provided, longer clips trimmed to this length. By default, shorter clips will not be extended (modify actions.AudioTrimmer to change behavior).
- **out_shape** – output shape of tensor in pixels [default: [224,224]]
- **return_labels** – if True, the __getitem__ method will return {X:sample,y:labels} If False, the __getitem__ method will return {X:sample} If df has no labels (no columns), use return_labels=False [default: True]

class opensoundscape.preprocess.preprocessors.**BasePreprocessor**(*df*, *return_labels=True*)

Base class for Preprocessing pipelines (use in place of torch Dataset)

Custom Preprocessor classes should subclass this class or its children

Parameters

- **df** – dataframe of samples. df must have audio paths in the index. If df has labels, the class names should be the columns, and the values of each row should be 0 or 1. If data does not have labels, df will have no columns
- **return_labels** – if True, the __getitem__ method will return {X:sample,y:labels} If False, the __getitem__ method will return {X:sample} If df has no labels (no columns), use return_labels=False [default: True]

Raises PreprocessingError if exception is raised during __getitem__

class_counts_cal()
count number of each label

head(*n=5*)
out-of-place copy of first n samples

performs df.head(n) on self.df

Parameters

- **n** – number of first samples to return, see pandas.DataFrame.head()
- **[default** – 5]

Returns a new dataset object

sample(***kwargs*)
out-of-place random sample

creates copy of object with n rows randomly sampled from dataframe

Args: see pandas.DataFrame.sample()

Returns a new dataset object

class opensoundscape.preprocess.preprocessors.**CnnPreprocessor**(*df, audio_length=None, return_labels=True, debug=None, overlay_df=None, out_shape=[224, 224]*)

Child of AudioToSpectrogramPreprocessor with full augmentation pipeline

loads audio, creates spectrogram, performs augmentations, returns tensor

by default, resamples audio to sr=22050 can change with .actions.load_audio.set(sample_rate=sr)

Parameters

- **df** – dataframe of samples. df must have audio paths in the index. If df has labels, the class names should be the columns, and the values of each row should be 0 or 1. If data does not have labels, df will have no columns
- **audio_length** – length in seconds of audio clips [default: None] If provided, longer clips trimmed to this length. By default, shorter clips will not be extended (modify actions.AudioTrimmer to change behavior).
- **out_shape** – output shape of tensor in pixels [default: [224,224]]
- **return_labels** – if True, the __getitem__ method will return {X:sample,y:labels} If False, the __getitem__ method will return {X:sample} If df has no labels (no columns), use return_labels=False [default: True]
- **debug** – If a path is provided, generated samples (after all augmentation) will be saved to the path as an image. This is useful for checking that the sample provided to the model matches expectations. [default: None]

augmentation_off()
use pipeline that skips all augmentations

augmentation_on()
use pipeline containing all actions including augmentations

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

17.2 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.**ActionContainer**
this is a container object which holds instances of Action child-classes

the Actions it contains each have .go(), .on(), .off(), .set(), .get()

The actions are un-ordered and may not all be used. In preprocessor objects such as AudioToSpectrogramPreprocessor, Actions from the action container are listed in a pipeline(list), which defines their order of use.

To add actions to the container: action_container.loader = AudioLoader() To set parameters of actions: action_container.loader.set(param=value,...)

Methods: list_actions()

class opensoundscape.preprocess.actions.**AudioLoader**(***kwargs*)
Action child class for Audio.from_file() (path -> Audio)

Loads an audio file, see Audio.from_file() for documentation.

Parameters

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

Note: default sample_rate=None means use file's sample rate, don't resample

class opensoundscape.preprocess.actions.**AudioToSpectrogram**(***kwargs*)
Action child class for Audio.from_file() (Audio -> Spectrogram)

see spectrogram.Spectrogram.from_audio for documentation

Parameters

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

class opensoundscape.preprocess.actions.**AudioTrimmer**(***kwargs*)
Action child class for trimming audio (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_length** – desired final length (sec); if None, no trim is performed
- **extend** – if True, clips shorter than audio_length are extended with silence to required length
- **random_trim** – if True, a random segment of length audio_length is chosen from the input audio. If False, the file is trimmed from 0 seconds to audio_length seconds.

class opensoundscape.preprocess.actions.**BaseAction**(***kwargs*)
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 ImgOverlay for an example of an Action that uses labels.

class opensoundscape.preprocess.actions.**FrequencyMask**(***kwargs*)

add random horizontal bars over image

Parameters

- **max_masks** – max number of horizontal bars [default: 3]
- **max_width** – maximum size of horizontal bars as fraction of image height

go(*x*)

torch Tensor in, torch Tensor out

class opensoundscape.preprocess.actions.**ImgOverlay**(*overlay_df*, *audio_length*, *loader_pipeline*, *update_labels*, ***kwargs*)

iteratively overlay images on top of eachother

Overlays images 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 the images on top of each other with a weight.

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
- **audio_length** – length in seconds of original audio sample
- **loader_pipeline** – the preprocessing pipeline to load audio -> spec
- **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 images will recieve 1 overlay and 1/4 will recieve an additional second overlay
- **overlay_weight** – can be a float between 0-1 or range of floats (chooses randomly from within range) such as [0.1,0.7]. An overlay_weight <0.5 means more emphasis on original image.

go(*x*, *x_labels*)
Overlay images from overlay_df

class opensoundscape.preprocess.actions.**ImgToTensor**(***kwargs*)
Convert PIL image to RGB Tensor (PIL.Image -> Tensor)

convert PIL.Image w/range [0,255] to torch Tensor w/range [0,1] converts image to RGB (3 channels)

class opensoundscape.preprocess.actions.**ImgToTensorGrayscale**(***kwargs*)
Convert PIL image to greyscale Tensor (PIL.Image -> Tensor)

convert PIL.Image w/range [0,255] to torch Tensor w/range [0,1] converts image to grayscale (1 channel)

class opensoundscape.preprocess.actions.**SaveTensorToDisk**(*save_path*, ***kwargs*)
save a torch Tensor to disk (Tensor -> Tensor)

Requires x_labels because the index of the label-row (.name) gives the original file name for this sample.

Uses torchvision.utils.save_image. Creates save_path dir if it doesn't exist

Parameters **save_path** – a directory where tensor will be saved

go(*x*, *x_labels*)
we require x_labels because the .name gives origin file name

class opensoundscape.preprocess.actions.**SpecToImg**(***kwargs*)
Action class to transform Spectrogram to PIL image

(Spectrogram -> PIL.Image)

Parameters

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

class opensoundscape.preprocess.actions.**SpectrogramBandpass**(***kwargs*)
Action class for Spectrogram.bandpass() (Spectrogram -> Spectrogram)

see opensoundscape.spectrogram.Spectrogram.bandpass() for documentation

To bandpass the spectrogram from 1kHz to 5Khz: action = SpectrogramBandpass(1000,5000)

Parameters

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

class opensoundscape.preprocess.actions.**TensorAddNoise**(***kwargs*)
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

class opensoundscape.preprocess.actions.**TensorAugment**(***kwargs*)
combination of 3 augmentations with hard-coded parameters

time warp, time mask, and frequency mask

use (bool) time_warp, time_mask, freq_mask to turn each on/off

go(*x*)
torch Tensor in, torch Tensor out

class opensoundscape.preprocess.actions.**TensorNormalize**(***kwargs*)
torchvision.transforms.Normalize (WARNING: FIXED shift and scale)

(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-u)/s.

Params: mean=0.5 std=0.5

class opensoundscape.preprocess.actions.**TimeMask**(***kwargs*)
add random vertical bars over image (Tensor -> Tensor)

Parameters

- **max_masks** – maximum number of bars [default: 3]
- **max_width** – maximum width of horizontal bars as fraction of image width
- **[default** – 0.2]

class opensoundscape.preprocess.actions.**TimeWarp**(***kwargs*)
Time warp is an experimental augmentation that creates a tilted image.

Parameters **warp_amount** – use higher values for more skew and offset (experimental)

class opensoundscape.preprocess.actions.**TorchColorJitter**(***kwargs*)
Action class for torchvision.transforms.ColorJitter

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

Parameters

- **brightness=0.3** –
- **contrast=0.3** –
- **saturation=0.3** –
- **hue=0** –

class opensoundscape.preprocess.actions.**TorchRandomAffine**(***kwargs*)
Action class for torchvision.transforms.RandomAffine

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

Parameters

- **= 0** (*degrees*) –
- **=** (*fill*) –
- **=** –

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).

17.3 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

17.4 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 18

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.**pulse_finder_species_set**(*spec*, *species_df*, *window_len='from_df'*, *plot=False*)

perform windowed pulse finding (ribbit) on one file for each species in a set

Parameters

- **spec** – opensoundscape.Spectrogram object
- **species_df** – a dataframe describing species by their pulsed calls. columns: species | pulse_rate_low (Hz)| pulse_rate_high (Hz) | low_f (Hz)| high_f (Hz)| reject_low (Hz)| reject_high (Hz) |

  window_length (sec) (optional) | reject_low2 (opt) | reject_high2 |

- **window_len** – length of analysis window, in seconds. Or 'from_df' (default): read from dataframe. or 'dynamic': adjust window size based on pulse_rate

Returns the same dataframe with a "score" (max score) column and "time_of_score" column

opensoundscape.ribbit.**ribbit**(*spectrogram*, *signal_band*, *pulse_rate_range*, *window_len*, *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
- **windo_len** – the length of audio (in seconds) to analyze at one time - one RIBBIT score is produced for each window
- **noise_bands** – list of frequency bands to subtract from the desired 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 window

Returns pulse score (float) for each time window array of time: start time of each window

Return type array of pulse_score

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.

Window Length: This parameter tells RIBBIT how many seconds of audio to analyze at one time. Generally, you should choose a *window_length* that is similar to the length of the target species vocalization, or a little bit longer. For very slowly pulsing vocalizations, choose a longer window so that at least 5 pulses can occur in one window (0.5 pulses per second -> 10 second window). Typical values for *window_length* are 1 to 10 seconds.

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 window_len 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

opensoundscape.ribbit.**summarize_top_scores**(*audio_files*, *list_of_result_dfs*, *scale_factor=1.0*)

find the highest score for each file and each species, and put them in a dataframe

Note: this function expects that the first column of the results_df contains species names

Parameters

- **audio_files** – a list of file paths
- **list_of_result_dfs** – a list of pandas DataFrames generated by ribbit_species_set()
- **scale_factor=1.0** – optionally multiply all output values by a constant value

Returns a dataframe summarizing the highest score for each species in each file

CHAPTER 19

Spectrogram

19.1 Mel Spectrogram

melspectrogram.py: Utilities for dealing with mel spectrograms

WARNING: This module has not been thoroughly tested for compatibility with modules and tools in OpenSoundscape.

class opensoundscape.melspectrogram.**MelSpectrogram**(*S*, *sample_rate*, *hop_length*, *fmin*, *fmax*)

Immutable spectrogram container

WARNING: This class has not been thoroughly tested for compatibility with modules and tools in OpenSoundscape.

classmethod from_audio(*audio*, *n_fft=1024*, *n_mels=128*, *window='flattop'*, *win_length=256*, *hop_length=32*, *htk=True*, *fmin=None*, *fmax=None*)

Create a MelSpectrogram object from an Audio object

The kwargs are cherry-picked from:

- 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_fft** – Length of the FFT window [default: 1024]
- **n_mels** – Number of mel bands to generate [default: 128]
- **window** – The windowing function to use [default: "flattop"]
- **win_length** – Each frame of audio is windowed by *window*. The window will be of length *win_length* and then padded with zeros to match *n_fft* [default: 256]
- **hop_length** – Number of samples between successive frames [default: 32]

- **htk** – use HTK formula instead of Slaney [default: True]
- **fmin** – lowest frequency (in Hz) [default: None]
- **fmax** – highest frequency (in Hz). If None, use *fmax = sr / 2.0* [default: None]

Returns opensoundscape.melspectrogram.MelSpectrogram object

to_image(*shape=None*, *mode='RGB'*, *s_range=(0, 20)*)
Generate PIL Image from MelSpectrogram

Given a range of values for S (e.g. default is minimum 0, maximum 20) generate a PIL image in 3-channel (RGB) or single channel (L) mode. A user can optionally resize the image.

Parameters

- **shape** – Resize to shape (h, w) [default: None]
- **mode** – Mode to write out "RGB" or "L" [default: "RGB"]
- **s_range** – The input range of S [default: (0, 20)]

Returns PIL.Image

to_pcen(*gain=0.8*, *bias=10.0*, *power=0.25*, *time_constant=0.06*)
Create PCEN from MelSpectrogram

Argument descriptions come from https://librosa.org/doc/latest/generated/librosa.pcen.html?highlight=pcen#librosa-pcen

Parameters

- **gain** – The gain factor. Typical values should be slightly less than 1 [default: 0.8]
- **bias** – The bias point of the nonlinear compression [default: 10.0]
- **power** – The compression exponent. Typical values should be between 0 and 0.5. Smaller values of power result in stronger compression. At the limit power=0, polynomial compression becomes logarithmic [default: 0.25]
- **time_constant** – The time constant for IIR filtering, measured in seconds [default: 0.06]

Returns The per-channel energy normalized version of MelSpectrogram.S

19.2 Spectrogram

spectrogram.py: Utilities for dealing with spectrograms

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 genereated 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*)
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

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=512*, *overlap_samples=256*, *decibel_limits=(-100*, *-20)*)
create a Spectrogram object from an Audio object

Parameters

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

Returns opensoundscape.spectrogram.Spectrogram object

classmethod from_file()
create a Spectrogram object from a file

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

Returns opensoundscape.spectrogram.Spectrogram object

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

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

similar to Audacity's gain and range parameters

Parameters

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

Returns Spectrogram object with db range applied

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

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

Returns Spectrogram object with values rescaled to feature_range

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

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

Returns Spectrogram object with values rescaled to feature_range

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

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

Parameters

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

return: time-series array of net amplitude

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

Parameters

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

to_image(*shape=None*, *mode='RGB'*)
Create a Pillow Image from spectrogram

Linearly rescales values in the spectrogram from self.decibel_limits to [255,0]

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

Parameters

- **destination** – a file path (string)
- **shape=None** – tuple of image dimensions, eg (224,224)
- **mode="RGB"** – RGB for 3-channel color or "L" for 1-channel grayscale

Returns Pillow Image object

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

Parameters

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

Returns spectrogram object from extracted time segment

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 20

Index

Python Module Index

o

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

U

V

W

X