Trigger Word Detection

February 3, 2024

[66]:
```
### v2.1
```

[67]:
```
# Import the necessary packages

import numpy as np
from pydub import AudioSegment
import random
import sys
import io
import os
import glob
import IPython
from td_utils import *
%matplotlib inline
```

[68]:
```
# The "activate" directory contains positive examples of people saying the word␣
↪"activate".
# The "negatives" directory contains negative examples of people saying random␣
↪words other than "activate".
# There is one word per audio recording.
# The "backgrounds" directory contains 10 second clips of background noise in␣
↪different environments.

# Positive example

IPython.display.Audio("./raw_data/activates/1.wav")
```

[68]: <IPython.lib.display.Audio object>

[69]:
```
# Negative example

IPython.display.Audio("./raw_data/negatives/4.wav")
```

[69]: <IPython.lib.display.Audio object>

[70]:
```
# Background noise
```

```python
IPython.display.Audio("./raw_data/backgrounds/1.wav")
```

[70]: <IPython.lib.display.Audio object>

You will use these three types of recordings (positives/negatives/backgrounds) to create a labeled dataset.

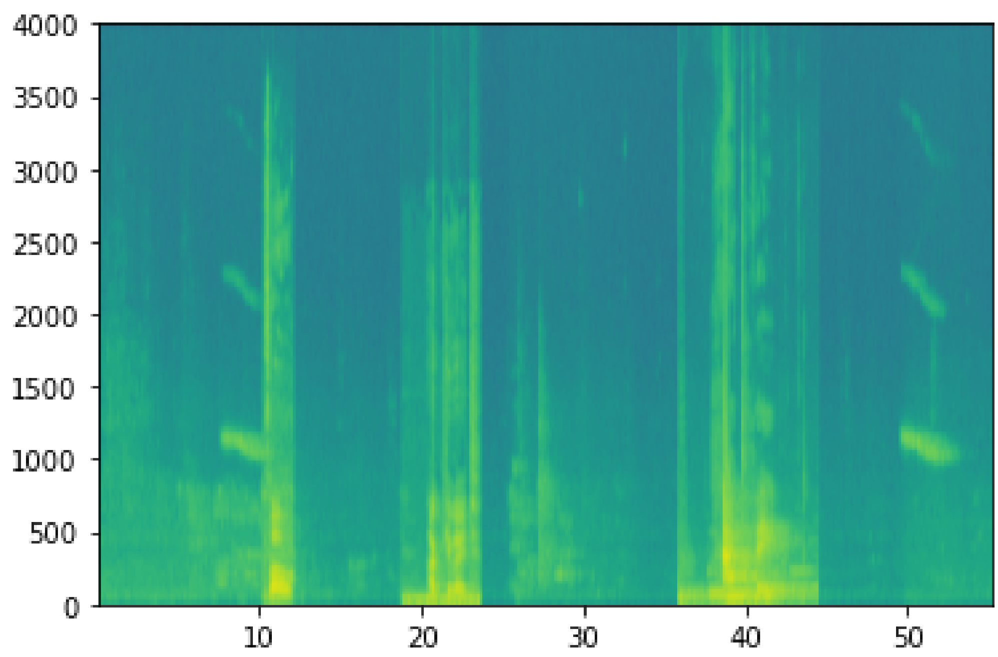
[71]:
```python
# What is an audio recording?

# A microphone records little variations in air pressure over time, and it is␣
↪these little variations in air pressure that
# your ear also perceives as sound.
# We can think of an audio recording as a long list of numbers measuring the␣
↪little air pressure changes detected by the
# microphone.

# We will use audio sampled at 44100 Hz (or 44100 Hertz).
# This means the microphone gives us 44,100 numbers per second.
# Thus, a 10 second audio clip is represented by 441,000 numbers (=  10×44,100)

# A sampling rate of 44,100 Hz means that the sound is sampled 44,100 times in␣
↪one second.
# Which means the sound is measured 44,100 times in one second and therefore we␣
↪have 44,100 measurements of the
# sound in 1 second.
# These 44,100 measurements in 1 second are air pressure variations.
# Different sounds are characterized by unique combinations of these air␣
↪pressure variations, which are determined by
# factors such as the frequency (pitch), amplitude (loudness), and timbre␣
↪(quality or color) of the sound.
# These air pressure variations are detected by our ears as sound.

# Difference between Frequnecy of sound and the sampling Frequency of sound

# 1) Frequency of Sound - Refers to the physical characteristic of the sound␣
↪wave itself, specifically, how many times the
# wave oscillates (vibrates) per second.
# 2) Sampling Frequency of Sound - Refers to how many times the air pressure␣
↪variations of the sound are measured in 1
# second by the microphone.

# Microphone's role

# The microphone's primary role is to convert air pressure variations into an␣
↪analog electrical signal.
# It measures the air pressure variations in the sound 44,100 times per second␣
↪and coverts these air pressure variations into
```

```
# an analog electrical signal. This analog signal reflects the changes in air␣
↪pressure over time.

# Analog-to-Digital Conversion

# The conversion of these air pressure variations (captured as an analog␣
↪electrical signal by the microphone) into
# numbers (digital data) is performed by an Analog-to-Digital Converter (ADC),␣
↪not the microphone itself.

# Storage using digital storage medium

# These numbers (digital audio data) are then stored in a digital storage␣
↪medium (not within the microphone).
# They can be processed, transmitted, or played back by digital audio devices.

# Playback using Digital-to-Analog Converter

# The Digital-to-Analog Converter converts the digital data back into an analog␣
↪electrical signal.
# The analog electrical signal is amplified using an amplifier.
# The amplified analog electrical signal is fed into a speaker or headphones␣
↪which causes the speaker cone or
# headphone drivers to move back and forth.
# This back and forth mechanical movement creates variations in air pressure,␣
↪similar to the original sound waves captured by
# the microphone. And these air pressure variations over time are detected as␣
↪sound by our ears.


IPython.display.Audio("audio_examples/example_train.wav")
```

[71]: <IPython.lib.display.Audio object>

[72]:
```
# Spectrogram

# It is quite difficult to figure out from this "raw" representation of audio␣
↪whether the word "activate" was said.
# In order to help our sequence model more easily learn to detect trigger␣
↪words, we will compute a spectrogram of the audio.
# The spectrogram tells us how much different frequencies are present in an␣
↪audio clip at any moment in time.

# A spectrogram is computed by sliding a window over the raw audio signal, and␣
↪calculating the most active frequencies in
# each window using a Fourier transform.
```

```
# At any particular time t, the audio may contain a large number of frequency␣
↪of sounds (different pitches).
# A spectrogram calculates the most active frequencies at a particular time t

x = graph_spectrogram("audio_examples/example_train.wav")

# The x-acxis denotes time and the y-axis denotes the frequency of sound.
# Green means a certain frequency is more active or more present in the audio␣
↪clip (louder).
# Blue squares denote less active frequencies.

# In this project, we will be working with 10 sec audio clips.
# And the number of timesteps of the spectrogram will be 5511.

# Note
# In this project, for a 10 sec audio clip, there are 5511 time steps of the␣
↪spectrogram and the total number of
# unique frequencies is 101. The numbers 5511 (number of time steps for a 10␣
↪sec audio clip) and
# 101 (number of unique frequencies of spectogram) depend on the software that␣
↪is converting the audio into the spectrogram.
```



[73]:
```
_, data = wavfile.read("audio_examples/example_train.wav")
print("Time steps in audio recording before spectrogram", data[:,0].shape)
```

```
print("Time steps in input after spectrogram", x.shape)

# Note
# In this project, for a 10 sec audio clip, there are 5511 time steps of the
↪spectrogram and the total number of
# unique frequencies is 101. The numbers 5511 (number of time steps for a 10
↪sec audio clip) and
# 101 (number of unique frequencies of spectogram) depend on the software that
↪is converting the audio into the spectrogram.
```

```
Time steps in audio recording before spectrogram (441000,)
Time steps in input after spectrogram (101, 5511)
```

[74]:
```
Tx = 5511 # The number of time steps input to the model from the spectrogram
n_freq = 101 # Number of frequencies input to the model at each time step of
↪the spectrogram

# Raw audio

# Raw audio (the one capture by microphone) divides 10 seconds into 441,000
↪units
# Spectrogram divides 10 seconds into 5,511 units (Tx = 5511)
# The output of our model will divide 10 seconds into 1,375 units ( =1375)
# For each of the 1375 time steps, the model predicts whether someone recently
↪finished saying the trigger word "activate".
```

[75]:
```
Ty = 1375 # The number of time steps in the output of our model
```

[76]:
```
# Synthesizing an audio clip
# Pick a random 10 second background audio clip
# Randomly insert 0-4 audio clips of "activate" into this 10 sec clip
# Randomly insert 0-2 audio clips of negative words into this 10 sec clip
# Because we had synthesized the word "activate" into the background clip, we
↪know exactly when in the 10 second clip the
# "activate" makes its appearance.
# This makes it easier to generate the labels     as well.

# Pydub converts raw audio files into lists of Pydub data structures and a 10
↪sec audio clip is represented using 10,000 steps.

# Load audio segments using pydub
activates, negatives, backgrounds = load_raw_audio('./raw_data/')

print("background len should be 10,000, since it is a 10 sec clip\n" +
↪str(len(backgrounds[0])),"\n")
print("activate[0] len may be around 1000, since an `activate` audio clip is
↪usually around 1 second (but varies a lot) \n" + str(len(activates[0])),"\n")
```

```
print("activate[1] len: different `activate` clips can have different␣
↪lengths\n" + str(len(activates[1])),"\n")
```

```
background len should be 10,000, since it is a 10 sec clip
10000

activate[0] len may be around 1000, since an `activate` audio clip is usually
around 1 second (but varies a lot)
916

activate[1] len: different `activate` clips can have different lengths
1579
```

[77]:
```
# Note

# When we insert or overlay an "activate" clip, we will also update labels for␣
↪

# Rather than updating the label of a single time step, we will update 50 steps␣
↪of the output to have target label 1 because
# of the following reasons :-

# 1) The exact moment when the word "activate" finishes can be spread across␣
↪several time steps in the spectrogram.
# Updating 50 consecutive steps helps cover the period right after the word is␣
↪said, accounting for the uncertainty in the
# exact time step the word finishes.

# 2) This approach helps in training the GRU model to recognize the end of the␣
↪trigger word more robustly.
# By marking a sequence of time steps as 1, the model is encouraged to learn␣
↪the temporal pattern associated with the end of
# "activate" and to respond to it over a span of steps, rather than a single,␣
↪potentially ambiguous point.

# 3) Speech data, and particularly trigger words within continuous speech, can␣
↪be sparse. The word "activate" might only
# occur a few times in a long audio clip. By marking 50 steps as 1 for each␣
↪occurrence, we increase the presence of the
# positive class (when "activate" is said) in the training data, helping to␣
↪balance the dataset and improve the model's
# ability to learn from both positive and negative examples.
```

[78]:
```
def get_random_time_segment(segment_ms):
    """
```

```
    Gets a random time segment of duration segment_ms in a 10,000 ms audio clip.

    Arguments:
    segment_ms -- the duration of the audio clip in ms ("ms" stands for␣
↪"milliseconds")

    Returns:
    segment_time -- a tuple of (segment_start, segment_end) in ms
    """

    segment_start = np.random.randint(low=0, high=10000-segment_ms)    # Make␣
↪sure segment doesn't run past the 10sec background
    # segment_start is a random integer between low and high
    segment_end = segment_start + segment_ms - 1

    return (segment_start, segment_end)
```

[79]:
```
def is_overlapping(segment_time, previous_segments):
    """
    Checks if the time of a segment overlaps with the times of existing␣
↪segments.

    Arguments:
    segment_time -- a tuple of (segment_start, segment_end) for the new segment
    previous_segments -- a list of tuples of (segment_start, segment_end) for␣
↪the existing segments

    Returns:
    True if the time segment overlaps with any of the existing segments, False␣
↪otherwise
    """

    segment_start, segment_end = segment_time

    # Initialize overlap as a "False" flag.
    overlap = False

    # loop over the previous_segments start and end times.
    # Compare start/end times and set the flag to True if there is an overlap
    for previous_start, previous_end in previous_segments:
        if ((segment_start >= previous_start and segment_start <= previous_end)
            or (segment_end >= previous_start and segment_end <= previous_end)
            or (segment_start < previous_start) and (segment_end >␣
↪previous_end)):
            overlap = True
            break
```

```
    return overlap
```

[80]:
```
### THIS CELL IS NOT EDITABLE

# UNIT TEST
def is_overlapping_test(target):
    assert target((670, 1430), []) == False, "Overlap with an empty list must␣
↪be False"
    assert target((500, 1000), [(100, 499), (1001, 1100)]) == False, "Almost␣
↪overlap, but still False"
    assert target((750, 900), [(100, 750), (1001, 1100)]) == True, "Must␣
↪overlap with the end of first segment"
    assert target((750, 1250), [(300, 600), (1250, 1500)]) == True, "Must␣
↪overlap with the beginning of second segment"
    assert target((750, 1250), [(300, 600), (600, 1500), (1600, 1800)]) ==␣
↪True, "Is contained in second segment"
    assert target((800, 1100), [(300, 600), (900, 1000), (1600, 1800)]) ==␣
↪True, "New segment contains the second segment"

    print("\033[92m All tests passed!")

is_overlapping_test(is_overlapping)
```

```
 All tests passed!
```

[81]:
```
overlap1 = is_overlapping((950, 1430), [(2000, 2550), (260, 949)])
overlap2 = is_overlapping((2305, 2950), [(824, 1532), (1900, 2305), (3424,␣
↪3656)])
print("Overlap 1 = ", overlap1)
print("Overlap 2 = ", overlap2)
```

```
Overlap 1 =  False
Overlap 2 =  True
```

[82]:
```
def insert_audio_clip(background, audio_clip, previous_segments):
    """
    Insert a new audio segment over the background noise at a random time step,␣
↪ensuring that the
    audio segment does not overlap with existing segments.

    Arguments:
    background -- a 10 second background audio recording.
    audio_clip -- the audio clip to be inserted/overlaid.
    previous_segments -- times where audio segments have already been placed

    Returns:
```

```
    new_background -- the updated background audio
    """

    # Get the duration of the audio clip in ms
    segment_ms = len(audio_clip)

    # Use one of the helper functions to pick a random time segment onto which␣
↪to insert
    # the new audio clip.
    segment_time = get_random_time_segment(segment_ms)

    # Check if the new segment_time overlaps with one of the previous_segments.␣
↪If so, keep
    # picking new segment_time at random until it doesn't overlap. To avoid an␣
↪endless loop
    # we retry 5 times
    retry = 5
    while is_overlapping(segment_time, previous_segments) and retry >= 0:
        segment_time = get_random_time_segment(segment_ms)
        retry = retry - 1
        #print(segment_time)
    # if last try is not overlaping, insert it to the background
    if not is_overlapping(segment_time, previous_segments):
        # Append the new segment_time to the list of previous_segments
        previous_segments.append(segment_time)
        # Superpose audio segment and background
        new_background = background.overlay(audio_clip, position =␣
↪segment_time[0])
    else:
        #print("Timeouted")
        new_background = background
        segment_time = (10000, 10000)

    return new_background, segment_time
```

[83]:

```
### THIS CELL IS NOT EDITABLE

# UNIT TEST
def insert_audio_clip_test(target):
    np.random.seed(5)
    audio_clip, segment_time = target(backgrounds[0], activates[0], [(0, 4400)])
    duration = segment_time[1] - segment_time[0]
    #print(f"xx: {segment_time}")
    assert segment_time[0] > 4400, "Error: The audio clip is overlapping with␣
↪the first segment"
    assert duration + 1 == len(activates[0]) , "The segment length must match␣
↪the audio clip length"
```

```
    assert audio_clip != backgrounds[0] , "The audio clip must be different␣
↪than the pure background"
    assert segment_time == (7286, 8201), f"Wrong segment. Expected: (7286,␣
↪8201) got:{segment_time}"

    # Not possible to insert clip into background
    audio_clip, segment_time = target(backgrounds[0], activates[0], [(0, 9999)])
    assert segment_time == (10000, 10000), "Segment must match the out by␣
↪max-retry mark"
    assert audio_clip == backgrounds[0], "output audio clip must be exactly the␣
↪same input background"

    print("\033[92m All tests passed!")

insert_audio_clip_test(insert_audio_clip)
```

```
All tests passed!
```

[84]:
```
np.random.seed(5)
audio_clip, segment_time = insert_audio_clip(backgrounds[0], activates[0],␣
↪[(3790, 4400)])
audio_clip.export("insert_test.wav", format="wav")
print("Segment Time: ", segment_time)
IPython.display.Audio("insert_test.wav")
```

```
Segment Time:  (2254, 3169)
```

[84]: <IPython.lib.display.Audio object>

[85]:
```
# In the above case, we notice that the 'positive' clip has been overlaid on␣
↪the 'background' clip in the
# segment (2254, 3169) ms
```

[86]:
```
def insert_ones(y, segment_end_ms):
    """
    Update the label vector y. The labels of the 50 output steps strictly after␣
↪the end of the segment
    should be set to 1. By strictly we mean that the label of segment_end_y␣
↪should be 0 while, the
    50 following labels should be ones.


    Arguments:
    y -- numpy array of shape (1, Ty), the labels of the training example
    segment_end_ms -- the end time of the segment in ms

    Returns:
```

```
    y -- updated labels
    """
    _, Ty = y.shape

    # duration of the background (in terms of spectrogram time-steps)
    segment_end_y = int(segment_end_ms * Ty / 10000.0)
    # The 'segment_end_ms' is the ending of the segment in milliseconds (total␣
↪time steps = 10,000)
    # But y contains a total of 1375 time steps
    # So we calculate the 'segment_end_y' given the 'segment_end_ms'

    if segment_end_y < Ty:
        # Add 1 to the correct index in the background label (y)
        for i in range(segment_end_y+1, segment_end_y+1 + 50): # We label 50␣
↪time steps after 'segment_end_y' as 1.
            if i < Ty: # This makes sure that i < Ty
                y[0, i] = 1
    return y
```

[87]:

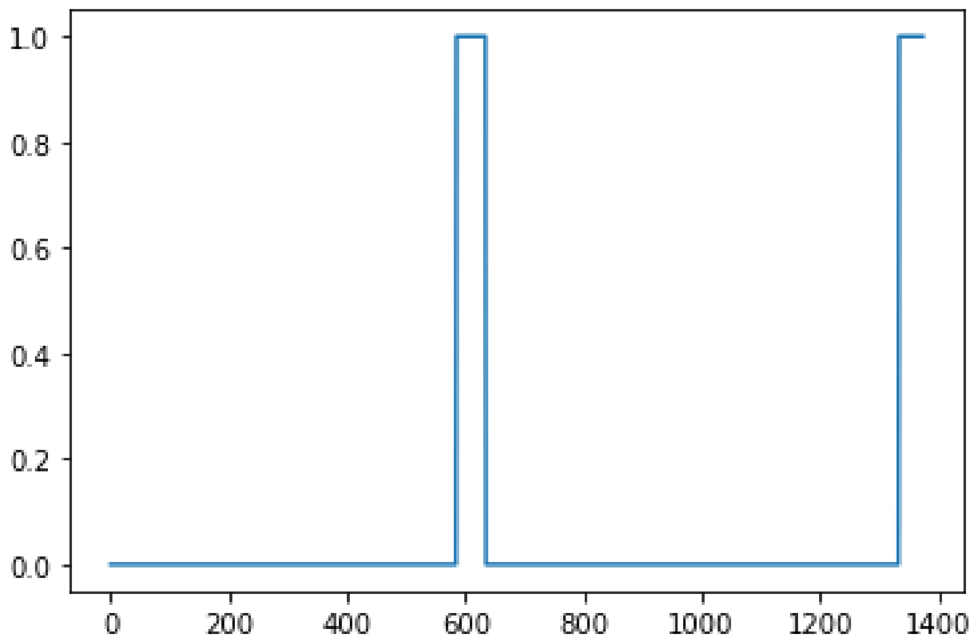
```
### THIS CELL IS NOT EDITABLE

# UNIT TEST
import random
def insert_ones_test(target):
    segment_end_y = random.randrange(0, Ty - 50)
    segment_end_ms = int(segment_end_y * 10000.4) / Ty;
    arr1 = target(np.zeros((1, Ty)), segment_end_ms)

    assert type(arr1) == np.ndarray, "Wrong type. Output must be a numpy array"
    assert arr1.shape == (1, Ty), "Wrong shape. It must match the input shape"
    assert np.sum(arr1) == 50, "It must insert exactly 50 ones"
    assert arr1[0][segment_end_y - 1] == 0, f"Array at {segment_end_y - 1} must␣
↪be 0"
    assert arr1[0][segment_end_y] == 0, f"Array at {segment_end_y} must be 0"
    assert arr1[0][segment_end_y + 1] == 1, f"Array at {segment_end_y + 1} must␣
↪be 1"
    assert arr1[0][segment_end_y + 50] == 1, f"Array at {segment_end_y + 50}␣
↪must be 1"
    assert arr1[0][segment_end_y + 51] == 0, f"Array at {segment_end_y + 51}␣
↪must be 0"

    arr1 = target(np.zeros((1, Ty)), 9632)
    assert np.sum(arr1) == 50, f"Expected sum of 50, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 9637)
    assert np.sum(arr1) == 49, f"Expected sum of 49, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 10008)
```

```
    assert np.sum(arr1) == 0, f"Expected sum of 0, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 10000)
    assert np.sum(arr1) == 0, f"Expected sum of 0, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 9996)
    assert np.sum(arr1) == 0, f"Expected sum of 0, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 9990)
    assert np.sum(arr1) == 1, f"Expected sum of 1, but got {np.sum(arr1)}"
    arr1 = target(np.zeros((1, Ty)), 9980)
    assert np.sum(arr1) == 2, f"Expected sum of 2, but got {np.sum(arr1)}"

    print("\033[92m All tests passed!")

insert_ones_test(insert_ones)
```

```
 All tests passed!
```

[88]:
```
arr1 = insert_ones(np.zeros((1, Ty)), 9700)
plt.plot(insert_ones(arr1, 4251)[0,:])
print("sanity checks:", arr1[0][1333], arr1[0][634], arr1[0][635])
```

```
sanity checks: 0.0 1.0 0.0
```



0.0.1 Creating one training example

[89]:
```
def create_training_example(background, activates, negatives, Ty):
    """
    Creates a training example with a given background, activates, and␣
↪negatives.

    Arguments:
    background -- a 10 second background audio recording
    activates -- a list of audio segments of the word "activate"
    negatives -- a list of audio segments of random words that are not␣
↪"activate"
    Ty -- The number of time steps in the output

    Returns:
    x -- the spectrogram of the training example
    y -- the label at each time step of the spectrogram
    """

    # Make background quieter
    background = background - 20
    # When the operation background = background - 20 is performed, it␣
↪decreases the volume of the background audio segment by
    # 20 decibels.

    # Initialize y (label vector) of zeros
    y = np.zeros((1,Ty))

    # Initialize segment times as empty list
    previous_segments = []

    # Select 0-4 random "activate" audio clips from the entire list of␣
↪"activates" recordings
    number_of_activates = np.random.randint(0, 5)
    random_indices = np.random.randint(len(activates), size=number_of_activates)
    # 'high' = len(activates). Default value for 'low' = 0. 'size' is the total␣
↪number of elements to be chosen randomly between
    # low and high
    random_activates = [activates[i] for i in random_indices]

    # Loop over randomly selected "activate" clips and insert in background
    for one_random_activate in random_activates:
        # Insert the audio clip on the background
        background, segment_time = insert_audio_clip(background,␣
↪one_random_activate, previous_segments)
        # Retrieve segment_start and segment_end from segment_time
```

```
        segment_start, segment_end = segment_time
        # Insert labels in "y" at segment_end
        y = insert_ones(y, segment_end)

    # It is possible that even after 5 retries, the 'audio clip' to be added to␣
↪'background' may be overlapping with the
    # 'previous_segments' in the 'background'. So the number of 'activate'␣
↪audio clips actually added to 'background' may not
    # be equal to the length of 'random_activates'

    # Select 0-2 random negatives audio recordings from the entire list of␣
↪"negatives" recordings
    number_of_negatives = np.random.randint(0, 3)
    random_indices = np.random.randint(len(negatives), size=number_of_negatives)
    random_negatives = [negatives[i] for i in random_indices]

    # Loop over randomly selected negative clips and insert in background
    for random_negative in random_negatives:
        # Insert the audio clip on the background
        background, _ = insert_audio_clip(background, random_negative,␣
↪previous_segments)

    # It is possible that even after 5 retries, the 'audio clip' to be added to␣
↪'background' may be overlapping with the
    # 'previous_segments' in the 'background'. So the number of 'negative'␣
↪audio clips actually added to 'background' may not
    # be equal to the length of 'random_activates'

    # Standardize the volume of the audio clip
    background = match_target_amplitude(background, -20.0)

    # Export new training example
    file_handle = background.export("train" + ".wav", format="wav")
    # The 'background' with superposition of positives and negatives is being␣
↪converted to audio format

    # Get and plot spectrogram of the new recording (background with␣
↪superposition of positive and negatives)
    x = graph_spectrogram("train.wav")

    return x, y
```

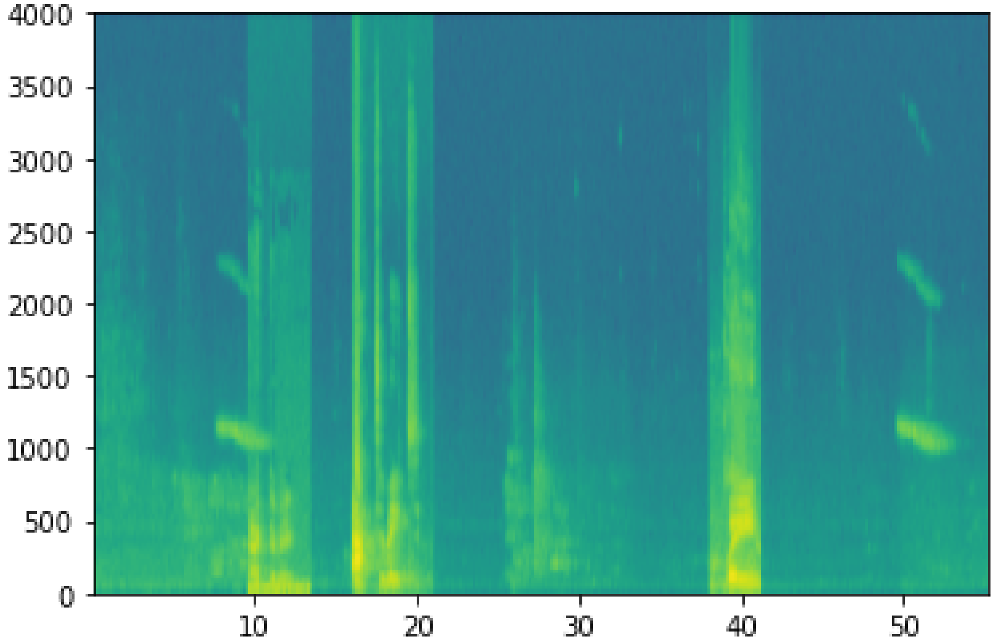
[90]:
```
# Role of spectogram

# Highlighting Dominant Frequencies: Speech and other foreground sounds often␣
↪have distinct frequency patterns that can
```

```
# stand out against the background noise in a spectrogram. By mapping these␣
↪frequencies over time, it becomes easier to
# identify when and where important audio events (like saying the word␣
↪"activate") occur.

# Background Noise Characteristics: Background noise often has a more uniform␣
↪distribution across frequencies or is
# concentrated in specific frequency bands that are different from those of␣
↪speech. In a well-processed spectrogram,
# these characteristics can help in distinguishing speech from noise.

# Filtering and Masking: Knowing the frequency content of background noise and␣
↪speech allows for the design of filters or
# the application of masking techniques to suppress unwanted noise, enhancing␣
↪the detection of target words or sounds.
```

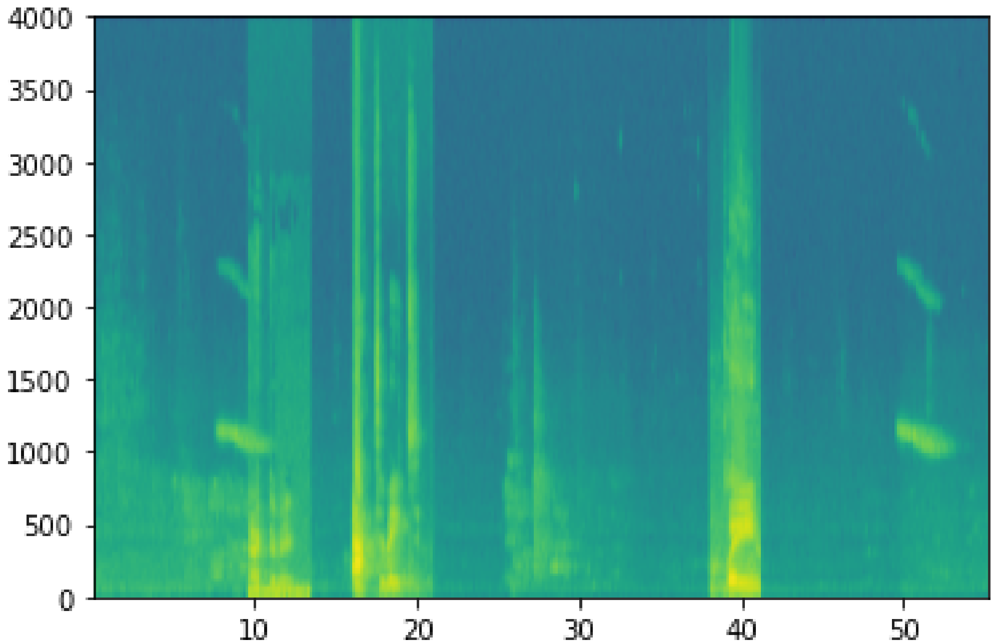
[91]:

```
### THIS CELL IS NOT EDITABLE

# UNIT TEST
def create_training_example_test(target):
    np.random.seed(18)
    x, y = target(backgrounds[0], activates, negatives, 1375)

    assert type(x) == np.ndarray, "Wrong type for x"
    assert type(y) == np.ndarray, "Wrong type for y"
    assert tuple(x.shape) == (101, 5511), "Wrong shape for x"
    assert tuple(y.shape) == (1, 1375), "Wrong shape for y"
    assert np.all(x > 0), "All x values must be higher than 0"
    assert np.all(y >= 0), "All y values must be higher or equal than 0"
    assert np.all(y <= 1), "All y values must be smaller or equal than 1"
    assert np.sum(y) >= 50, "It must contain at least one activate"
    assert np.sum(y) % 50 == 0, "Sum of activate marks must be a multiple of 50"
    assert np.isclose(np.linalg.norm(x), 39745552.52075), "Spectrogram is wrong.
↪ Check the parameters passed to the insert_audio_clip function"

    print("\033[92m All tests passed!")

create_training_example_test(create_training_example)
```

```
All tests passed!
```



[92]:
```
# Set the random seed
np.random.seed(18)
x, y = create_training_example(backgrounds[0], activates, negatives, Ty)
```



[93]:
```
print(x.shape)
print(y.shape)
```

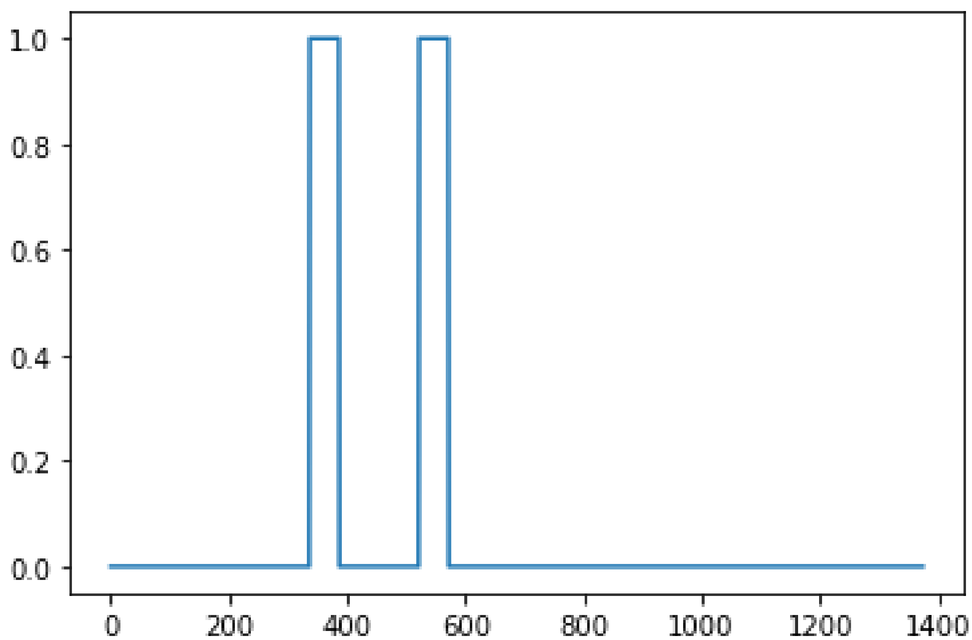
```
(101, 5511)
(1, 1375)
```

[94]:
```
# Listening to the training example we created

IPython.display.Audio("train.wav")
```

[94]:
```
<IPython.lib.display.Audio object>
```

[95]:
```
# Plotting the labels (y) of the generated training example

plt.plot(y[0])
```

[95]:
```
[<matplotlib.lines.Line2D at 0x7fc3130f2950>]
```

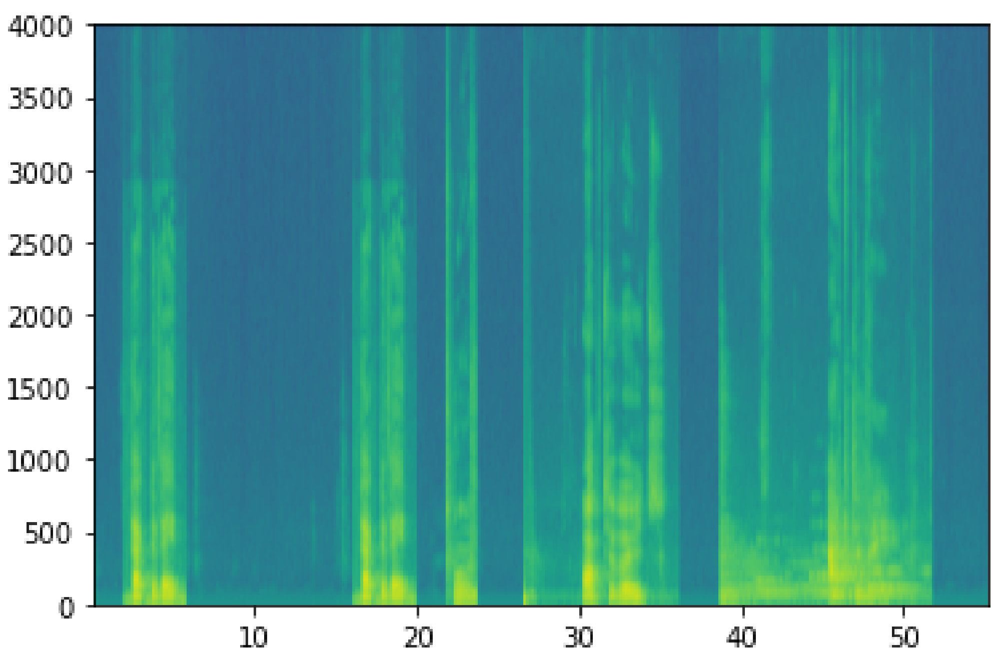


0.0.2 Creating Training set of 32 examples

[96]:
```
np.random.seed(4543)
nsamples = 32
X = []
Y = []
for i in range(0, nsamples):
```

```
    if i%10 == 0:
        print(i)
    x, y = create_training_example(backgrounds[i % 2], activates, negatives, Ty)
    X.append(x.swapaxes(0,1))
    Y.append(y.swapaxes(0,1))
X = np.array(X)
Y = np.array(Y)

# Note
# Only 2 different 'backgrounds' are being used to generate the training␣
↪examples.

# The spectrogram returns an array where the first axis (axis 0) represents␣
↪frequency values and the second axis (axis 1)
# represents the timestep. But when preparing data for training, the input to␣
↪the model is often expected to have a shape of
# (batch_size, time_steps, features). That's why we swap the axes of 'x'.

# Even if we don't swap the axes of y for each individual training example,
# the shape of Y after aggregating y from multiple training examples would␣
↪indeed be (m, Ty)
```

```
0
10
20
30
```



[97]:
```
# To test our model, we recorded a development set of 25 examples.
# While our training data is synthesized, we want to create a development set␣
↪using the same distribution as
# the real world data that our test set would be tested on.
# Thus, we recorded 25 10-second audio clips of people saying "activate" and␣
↪other random words, and labeled them by hand.
# This follows the principle that we should create the dev set to be as similar␣
↪as possible to the test set distribution

# This is why our dev set uses real audio rather than synthesized audio.

# Load preprocessed dev set examples
X_dev = np.load("./XY_dev/X_dev.npy")
Y_dev = np.load("./XY_dev/Y_dev.npy")
```

0.0.3 Building the Model

[98]:
```
# Importing the packages

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model, load_model, Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, Input, Masking,␣
↪TimeDistributed, LSTM, Conv1D
from tensorflow.keras.layers import GRU, Bidirectional, BatchNormalization,␣
↪Reshape
from tensorflow.keras.optimizers import Adam
```

[99]:
```
def modelf(input_shape):
    """
    Function creating the model's graph in Keras.

    Argument:
    input_shape -- shape of the model's input data (using Keras conventions)

    Returns:
    model -- Keras model instance
    """

    X_input = Input(shape = input_shape)

    # Add a Conv1D with 196 units, kernel size of 15 and stride of 4
    X = Conv1D(filters=196,kernel_size=15,strides=4)(X_input)
    # Batch normalization
    X = BatchNormalization()(X)
```

```
    # ReLu activation
    X = Activation("relu")(X)
    # dropout (use 0.8)
    X = Dropout(rate=0.8)(X)

    # First GRU Layer
    # GRU (use 128 units and return the sequences)
    X = GRU(units=128, return_sequences = True)(X)
    # In a standard LSTM, return_state=True will return the last hidden state␣
↪(a<Tx>) and the last cell state (c<Tx>), along
    # with the output sequence (if return_sequences=True) ([a<1>,a<2>,....
↪a<Tx>]) or the last
    # output (a<Tx>) (if return_sequences=False).
    # dropout (use 0.8)
    X = Dropout(rate=0.8)(X)
    # Batch normalization.
    X = BatchNormalization()(X)

    # Second GRU Layer
    # GRU (use 128 units and return the sequences)
    X = GRU(units=128, return_sequences = True)(X)
    # dropout (use 0.8)
    X = Dropout(rate=0.8)(X)
    # Batch normalization
    X = BatchNormalization()(X)
    # dropout (use 0.8)
    X = Dropout(rate=0.8)(X)

    # Time-distributed dense layer
    # TimeDistributed  with sigmoid activation
    # This creates a dense layer followed by a sigmoid, so that the parameters␣
↪used for the dense layer are the same for
    # every time step.
    X = TimeDistributed(Dense(1, activation = "sigmoid"))(X)
    # output 'X' is of shape (m, Tx=1375, 1)
    # The 'Dense' layer only transforms the last dimension.
    # When you pass a tensor of size (x,y,z) through the 'densor1', the␣
↪resultant is of size (x,y,10).

    model = Model(inputs = X_input, outputs = X)
    # X_input is of shape (m, Tx=5511, 101)
    # output 'X' is of shape (m, Tx=1375)

    return model
```

[100]:
```
# Note 1
# When each training example is 3-dimensional, then we use 2-d convolutions.␣
↪But here each training example of X is
# 2-dimensional and hence we use 1-d convolutions.

# We can think of the input (X) to the 1-D convolutions as each layer is 1 x␣
↪5511 and there are 101 such layers.

# The filter size is 15. The number of channels is not mentioned but is equal␣
↪to 101. The number of filters = 196.

# The output after 1-D convolutions would be each layer is 1 x 1375 and there␣
↪are 196 such layers.

# Computationally, the 1-D conv layer also helps speed up the model because now␣
↪the GRU can process only 1375 timesteps rather
# than 5511 timesteps.

# Even though there is an increase in the number of layers from 101 to 196, but␣
↪still the model is sped up because
# computational cost of RNNs like GRUs is more sensitive to sequence length␣
↪than to feature dimensionality.
# GRU layers have to maintain and update hidden states over time, which␣
↪involves matrix multiplications that are more
# computationally intensive as the sequence gets longer. Therefore, processing␣
↪shorter sequences, even with a higher
# feature dimension, can be more efficient.

# The GRU block shown in the model diagram has weights (W_c, W_u, W_r). The␣
↪input fed to the GRU block weights has to have some
# consistency in distribution across mini-batches. Hence we are applying batch␣
↪normalization before passing it into the
# GRU block.

# Note 2 - How batch normalization works with sequential data?

# For a particular timestep and a minibatch of m examples, the input is of␣
↪shape (m, 101).
# Batch normalization makes sure that first we calculate the mean and variance␣
↪of the first feature across all
# training examples and normalize the first feature and then multiply it with␣
↪gamma and beta so that the first feature has
# a mean of mu_1 and variance of sigma_squared_1. Then the mean and variance of␣
↪second feature across training examples is
# calculated and the second feature is normalized and then multiplied with␣
↪different gamma and beta so that the
```

```
# second feature has different mean of mu_2 and variance of sigma_squared_2.

# Note 3

# For a particular timestep, the gamma and beta vary with the features. The␣
↪gamma and beta do not change with the timestep.
# For example, the gamma and beta associated with the first feature of first␣
↪timestep is same as the gamma and beta associated
# with the first feature of second timestep.
# This is because the GRU block weights (W_c, W_u, W_r) are the same across␣
↪timesteps. The input fed to these weights need to
# have some consistency in distribution across mini-batches. Since the weights␣
↪are the same across the time steps and each of
# these same weights would expect the same input distribution, therefore the␣
↪gamma and beta do not change with the timestep.
# If the gamma and beta varies across timesteps, then the input distribution to␣
↪each of the GRU block weights would vary and
# this would not result in efficient learning of the GRU block weights.

# Note 4

# Autoregressive feedback is when we feed the output of a particular timestep␣
↪as one of the input to predict the output of the
# next timestep.

# Even in training phase of trigger word detection task, we can use auto␣
↪regressive feedback. When 'act' is already said and
# if we feed that as information as one of the input while predicting the␣
↪output of next time step, then the model will
# predict better that 'ivate' is likely to follow in the next time step.

# But we haven't used auto regressive feedback in this project because␣
↪implementing autoregressive feedback can be
# computationally more intensive, as it requires running the model sequentially␣
↪for each timestep rather than in parallel for
# the whole sequence.
```

[101]:

```
### THIS CELL IS NOT EDITABLE

# UNIT TEST
from test_utils import *

def modelf_test(target):
    Tx = 5511
    n_freq = 101
    model = target(input_shape = (Tx, n_freq))
```

```
    expected_model = [['InputLayer', [(None, 5511, 101)], 0],
                      ['Conv1D', (None, 1375, 196), 297136, 'valid', 'linear',␣
 ↪(4,), (15,), 'GlorotUniform'],
                      ['BatchNormalization', (None, 1375, 196), 784],
                      ['Activation', (None, 1375, 196), 0],
                      ['Dropout', (None, 1375, 196), 0, 0.8],
                      ['GRU', (None, 1375, 128), 125184, True],
                      ['Dropout', (None, 1375, 128), 0, 0.8],
                      ['BatchNormalization', (None, 1375, 128), 512],
                      ['GRU', (None, 1375, 128), 99072, True],
                      ['Dropout', (None, 1375, 128), 0, 0.8],
                      ['BatchNormalization', (None, 1375, 128), 512],
                      ['Dropout', (None, 1375, 128), 0, 0.8],
                      ['TimeDistributed', (None, 1375, 1), 129, 'sigmoid']]
    comparator(summary(model), expected_model)


modelf_test(modelf)
```

```
All tests passed!
```

[102]:
```
model = modelf(input_shape = (Tx, n_freq))
```

[103]:
```
# Let's print the model summary to keep track of the shapes.
```

[104]:
```
model.summary()
```

```
Model: "functional_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 5511, 101)]       0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 1375, 196)         297136
_________________________________________________________________
batch_normalization_12 (Batc (None, 1375, 196)         784
_________________________________________________________________
activation_4 (Activation)    (None, 1375, 196)         0
_________________________________________________________________
dropout_16 (Dropout)         (None, 1375, 196)         0
_________________________________________________________________
gru_8 (GRU)                  (None, 1375, 128)         125184
_________________________________________________________________
dropout_17 (Dropout)         (None, 1375, 128)         0
_________________________________________________________________
batch_normalization_13 (Batc (None, 1375, 128)         512
_________________________________________________________________
```

```
gru_9 (GRU)                  (None, 1375, 128)         99072
_________________________________________________________________
dropout_18 (Dropout)         (None, 1375, 128)         0
_________________________________________________________________
batch_normalization_14 (Batc (None, 1375, 128)         512
_________________________________________________________________
dropout_19 (Dropout)         (None, 1375, 128)         0
_________________________________________________________________
time_distributed_4 (TimeDist (None, 1375, 1)           129
=================================================================
Total params: 523,329
Trainable params: 522,425
Non-trainable params: 904
_________________________________________________________________
```

[105]:
```
# Note

# The output of the network is of shape (None, 1375, 1) while the input is
↪(None, 5511, 101).
# The Conv1D has reduced the number of steps from 5511 to 1375.

# The output of the network is of shape (None, 1375, 1) while the input is
↪(None, 5511, 101).
# The Conv1D has reduced the number of steps from 5511 to 1375.
```

0.0.4 Loading the Model

[106]:
```
# Trigger word detection takes a long time to train
# To save time, we've already trained a model for about 3 hours on a GPU using
↪the architecture we built above, and a
# large training set of about 4000 examples.

# Loading the model

from tensorflow.keras.models import model_from_json

json_file = open('./models/model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
model = model_from_json(loaded_model_json)
model.load_weights('./models/model.h5')
```

[107]:
```
# We are fine-tuning a pretrained model. So it is important to freeze the
↪weights of all our batchnormalization layers.

model.layers[2].trainable = False
```

```
model.layers[7].trainable = False
model.layers[10].trainable = False
```

0.0.5 Compiling the Model

[108]:
```
opt = Adam(lr=1e-6, beta_1=0.9, beta_2=0.999)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=["accuracy"])
```

[109]:
```
model.fit(X, Y, batch_size = 16, epochs=1)
```

```
2/2 [==============================] - 4s 2s/step - loss: 0.1674 - accuracy:
0.9462
```

[109]:
```
<tensorflow.python.keras.callbacks.History at 0x7fc30c497fd0>
```

0.0.6 Testing the Model

[110]:
```
# Testing the model on dev set is done using model.evaluate()

loss, acc, = model.evaluate(X_dev, Y_dev)
print("Dev set accuracy = ", acc)
```

```
1/1 [==============================] - 0s 1ms/step - loss: 0.1886 - accuracy:
0.9239
Dev set accuracy =  0.9238981604576111
```

[111]:
```
# Note

# Accuracy isn't a great metric for this task since the labels are heavily␣
↪skewed to 0's, a neural network that
# just outputs 0's would get slightly over 90% accuracy.

# We could define more useful metrics such as F1 score or Precision/Recall.
```

0.0.7 Making the Predictions

[112]:
```
def detect_triggerword(filename):
    plt.subplot(2, 1, 1)

    # Correct the amplitude of the input file before prediction
    audio_clip = AudioSegment.from_wav(filename)
    audio_clip = match_target_amplitude(audio_clip, -20.0)
    file_handle = audio_clip.export("tmp.wav", format="wav")
```

```
    filename = "tmp.wav"

    x = graph_spectrogram(filename)
    # the spectrogram outputs (freqs, Tx) and we want (Tx, freqs) to input into␣
↪the model
    x  = x.swapaxes(0,1)
    x = np.expand_dims(x, axis=0)
    # The above line of code adds an extra dimension to the array x at the␣
↪specified axis position.
    # In this case, axis=0 adds a new dimension at the beginning of the array.
    # The shape of x becomes from (Tx, n_freq) to (1, Tx, n_freq)
    # Note

    # Model Input Shape: When we define a Keras model with a 1D convolutional␣
↪layer, we specify the input shape without the
    # batch size: input_shape=(Tx, freqs). Keras implicitly assumes that the␣
↪data will be provided with an additional
    # batch size dimension when we run the model.

    # Batch Size Dimension: The data we feed into the model for training or␣
↪prediction should include the batch size dimension.
    # This is true even if we are predicting just one example (batch_size = 1).␣
↪Keras does not automatically add the
    # batch size dimension; we must explicitly format our data to have this␣
↪dimension.

    predictions = model.predict(x)

    plt.subplot(2, 1, 2)
    plt.plot(predictions[0,:,0])
    plt.ylabel('probability')
    plt.show()
    return predictions
```

0.0.8 Inserting a Chime

[113]:
```
# Inserting a chime to acknowledge the 'activate' trigger

#    might be near 1 for many values in a row after "activate" is said, yet we␣
↪want to chime only once.
# So we will insert a chime sound at most once every 75 output steps.
# This will help prevent us from inserting two chimes for a single instance of␣
↪"activate".
# This plays a role similar to non-max suppression from computer vision.
```

```
# We chose the number '75' because the first instance t when the person␣
↪finishes saying 'activate', y<t> is labelled as 1.
# Thereafter the label of next 49 timesteps are labelled as 1.
# If we had chosen any number less than '50', then for 1 'activate', more than␣
↪1 chime may be produced.
# If we had chosen any number much greater than '50', then there may be a␣
↪training example where the person says
# 'activate' twice quickly one after the other. We would have missed the 2nd␣
↪'activate' and there may be only
# 1 chime (for the 1st 'activate') produced.

# When a person finishes saying 'activate', y<t> may not be exactly equal to 1.␣
↪We have to check if y<t> is greater than a
# particular threshold.
```

[114]:
```
chime_file = "audio_examples/chime.wav"
def chime_on_activate(filename, predictions, threshold):
    audio_clip = AudioSegment.from_wav(filename)
    chime = AudioSegment.from_wav(chime_file)
    Ty = predictions.shape[1]
    # Initialize the number of consecutive output steps to 0
    consecutive_timesteps = 0
    i = 0
    # Loop over the output steps in the y
    while i < Ty:
        # Increment consecutive output steps
        consecutive_timesteps += 1
        # If prediction is higher than the threshold for 20 consecutive output␣
↪steps have passed
        if consecutive_timesteps > 20:
            # Superpose audio and background using pydub
            audio_clip = audio_clip.overlay(chime, position = ((i / Ty) *␣
↪audio_clip.duration_seconds) * 1000)
            # Reset consecutive output steps to 0
            consecutive_timesteps = 0
            i = 75 * (i // 75 + 1)
            continue
        # if amplitude is smaller than the threshold reset the␣
↪consecutive_timesteps counter
        if predictions[0, i, 0] < threshold:
            consecutive_timesteps = 0
        i += 1

    audio_clip.export("chime_output.wav", format='wav')
```

```
# Note - Potential issue in above code

# What if at i = 129, the person just completed saying 'activate'. Then from i␣
↪= 129 to i = 178 (both inclusive), y<t> is
# above threshold. At i = 149, consecutive_timesteps = 21 and a chime is␣
↪overlaid and i is set to 150 (i = 75 * (i // 75 + 1))
# and consecutive_timesteps is set to 0. Again at i = 170,␣
↪consecutive_timesteps = 21 and another chime is overlaid.
# Therefore for a single 'activate', 2 chimes are overlaid

# The 'if' statement should be corrected to below line of code
# if consecutive_timesteps > 25:

# What if at i = 124, the person just completed saying 'activate'. Then from i␣
↪= 124 to i = 173 (both inclusive), y<t> is
# above threshold. At i = 149, consecutive_timesteps = 26 and a chime is␣
↪overlaid and i is set to 150 (i = 75 * (i // 75 + 1))
# and consecutive_timesteps is set to 0. Again at i = 175,␣
↪consecutive_timesteps = 0 (This is because at i = 174, y<t> is
# less than threshold and hence consecutive_timesteps is set to 0) and no chime␣
↪is overlaid.
# Therefore for a single 'activate', only 1 chime is overlaid
```

0.0.9 Testing on Dev Examples

[115]:
```
# Testing our model on 2 unseen audio clips from the development set
```

[116]:
```
# Audio clip 1

IPython.display.Audio("./raw_data/dev/1.wav")
```

[116]: <IPython.lib.display.Audio object>

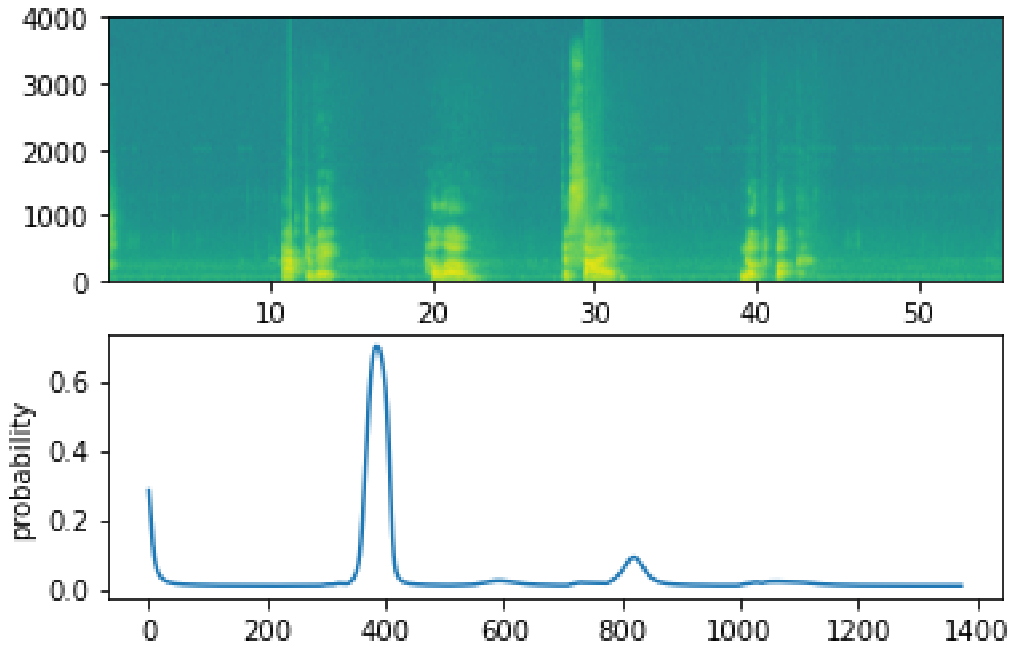
[117]:
```
# Audio clip 2

IPython.display.Audio("./raw_data/dev/2.wav")
```

[117]: <IPython.lib.display.Audio object>

[118]:
```
# Now lets run the model on these audio clips and see if it adds a chime after␣
↪"activate"
```

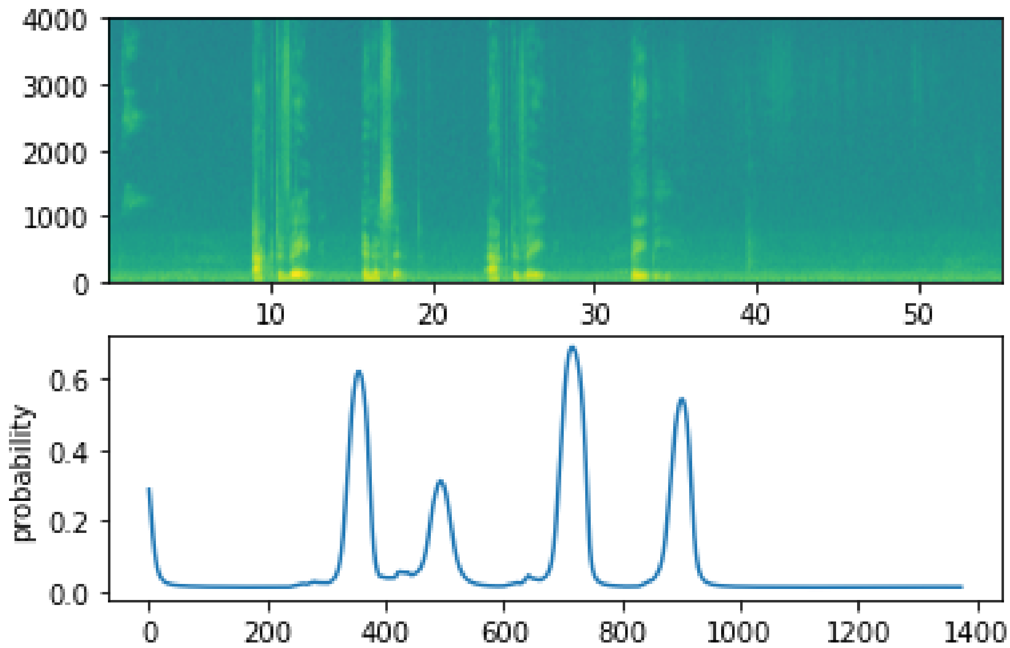
[119]:
```
# Audio clip 1 with Chime

filename = "./raw_data/dev/1.wav"
```

```
prediction = detect_triggerword(filename)
chime_on_activate(filename, prediction, 0.5)
IPython.display.Audio("./chime_output.wav")
```



[119]: <IPython.lib.display.Audio object>

[120]:
```
# Audio clip 2 with Chime

filename  = "./raw_data/dev/2.wav"
prediction = detect_triggerword(filename)
chime_on_activate(filename, prediction, 0.5)
IPython.display.Audio("./chime_output.wav")
```



[120]: <IPython.lib.display.Audio object>

[121]:
```
# Note

# Data synthesis is an effective way to create a large training set for speech␣
↪problems, specifically trigger word detection.
# Using a spectrogram and optionally a 1D conv layer is a common pre-processing␣
↪step prior to passing audio data to an RNN,
# GRU or LSTM.
# An end-to-end deep learning approach can be used to build a very effective␣
↪trigger word detection system.
```