Machine Learning Engineer Nanodegree

Speaker Classification Capstone Project

Wendell Luckow

05/04/17

I. Definition 2

Project Overview 2

Problem Statement 2

Metrics 2

II. Analysis 3

Data Exploration 3

Exploratory Visualization 3

Algorithms and Techniques 4

Benchmark 5

III. Methodology 6

Data Preprocessing 6

Crop the Files to Three Minutes 6

Search for and Save 200ms-Length Samples of Audio 6

Create Labels for the Data 7

Split the Data into Training/Testing/Validation Sets 7

Implementation 7

Attempt #1: K-Nearest Neighbors 7

Attempt #2: A Simple Neural Network 8

Attempt #3: A Convolutional Neural Network on Spectrograms 9

Attempt #4: A Convolutional Neural Network on RGB Images of STFT Data 10

Refinement 11

IV. Results 12

Model Evaluation and Validation 12

Justification 12

V. Conclusion 13

Free-Form Visualization 13

Reflection 14

Improvement 15

I. Definition

Project Overview

The goal of this project is to find an efficient and accurate way to classify who is speaking from a small set of possible speakers. For instance, suppose you listened to three minutes of ten different people speaking. If one of those ten people then said something new, would you be able to identify who spoke?

The applications for a project like this are enormous. For instance, on the popular video platform YouTube, there are now automated subtitles created by advanced algorithms and networks. However, the networks are unable to differentiate who is speaking based on the voices only. Another application for speaker classification exists in security, such as speaking a word or passphrase as a computer password.

Problem Statement

In this project, ten different audio files will be used containing ten different speakers reading ten different chapters or sections from some reading material. The problem to be solved is to identify which speaker is talking when introducing a network some new, previously unseen data.

To approach this problem, the following steps must be taken:

- Find ten clear sources of audio, read by ten different speakers. Include 5 male voices and 5 female voices.
- Preprocess the data so that it can be introduced to a neural network as an array.
- To train a network, create labels for the data as this is a supervised classification task.

Once the data is in a usable format, there are several options which could all be explored. Because the problem is so open-ended, several options will be tested in this project:

- K-nearest neighbors on the raw data: Using SciKit-Learn, input the raw audio files as arrays into a K-nearest neighbors algorithm.
- Feed forward neural network on the raw data: Using the same data arrays, input each data point into a neural network as its own feature similar to how each pixel can be used to classify MNIST images.
- Convolutional neural network on spectrograms: Using matplotlib, create a spectrogram for the various audio files and run the spectrograms through a convolutional neural network.
- Convolutional neural network on Short-time Fourier Transformed data, plotted as an image. This is similar to the above approach, using convolutional neural networks on spectrograms, however will be more precise, as the matplotlib spectrograms use predefined sections and averages of frequency magnitudes.

The goal of this project will be to achieve an accuracy of over 90%.

Metrics

To measure the performance of the models used in the project, total accuracy will be the primary metric. Because there are ten speakers, randomly guessing will achieve around 10% accuracy. This is a general multi-class classification problem with ten possible classes, so using the following equation for measuring the performance will be sufficient:

$$accuracy = \frac{total\ correct\ classifications}{total\ attempted\ classifications}$$

II. Analysis

Data Exploration

All data was downloaded from LibriVox. All data is in the public domain for free use. The data includes 10 different .mp3 files of 10 different speakers reading a chapter or section from a book or other reading material. The length of each audio clip ranges between 7 minutes and almost 30 minutes.

In order to input the data into Python, each audio file was converted to a .wav using the free software, VLC Media Player. Then, using the open-source library Librosa, data was input into Python. Audio data imported through Librosa is automatically normalized to contain values between 0 and 1, rather than true frequencies.

The .wav files that are imported into Python come in the form of numpy arrays. In audio files, it's important to know what the sampling rate is. For instance, all of the files used in this project had a sample rate of 48000. The data included in an audio file is the frequency for each sample. To calculate the length of an array, the following equation can be used:

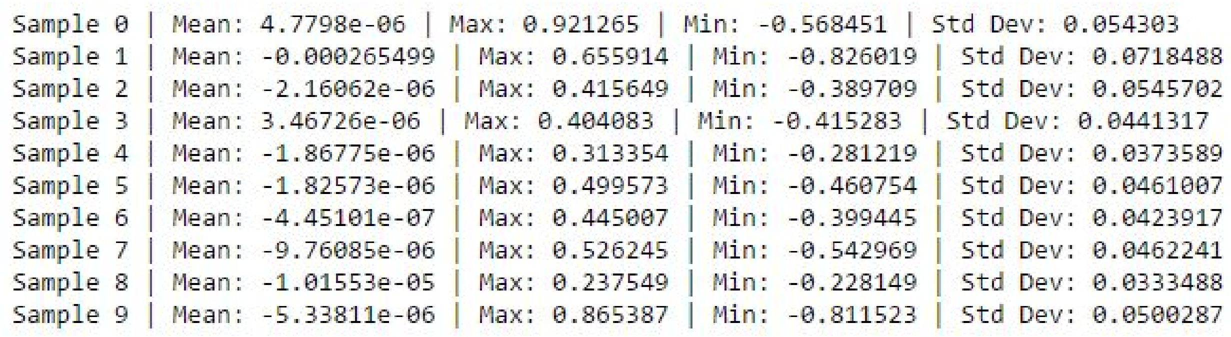
$$length\ of\ array\ =\ sample\ rate\ \cdot\ length\ of\ audio\ (seconds)$$

For example, one of the audio clips that was imported into Python has a duration of 28 minutes and 10 seconds, which is equivalent to 1690 seconds. The audio clip also has a sample rate of 48000 samples per second, which means that the imported array has a length of 81,120,000 samples. In order to make the data more consistent, the audio was cropped down to the first three minutes (180 seconds). Then the 10 three-minute samples were saved as their own dataset. In other words, only the first (48000 * 180) samples were used, and the rest were deleted using numpy.delete.

Along with these ten samples, another dataset was simultaneously created to act as the labels for the neural networks. For instance, if an audio clip was spoken by the first speaker, the following label was created: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]. If an audio clip was spoken by the second speaker, that label would be: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0], and so on.

Exploratory Visualization

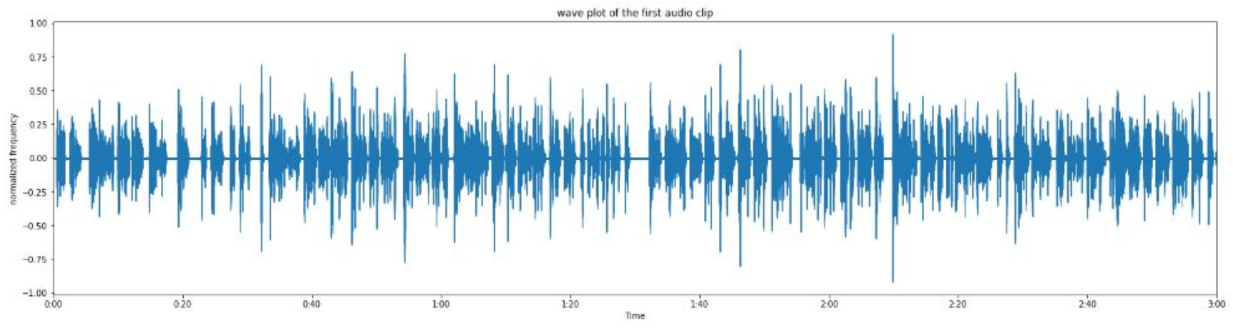
Once all of the audio clips were cut down to three minutes in length, some basic statistical information was calculated about the samples. For each sample, the mean frequency, minimum frequency, maximum frequency, and standard deviation was calculated.

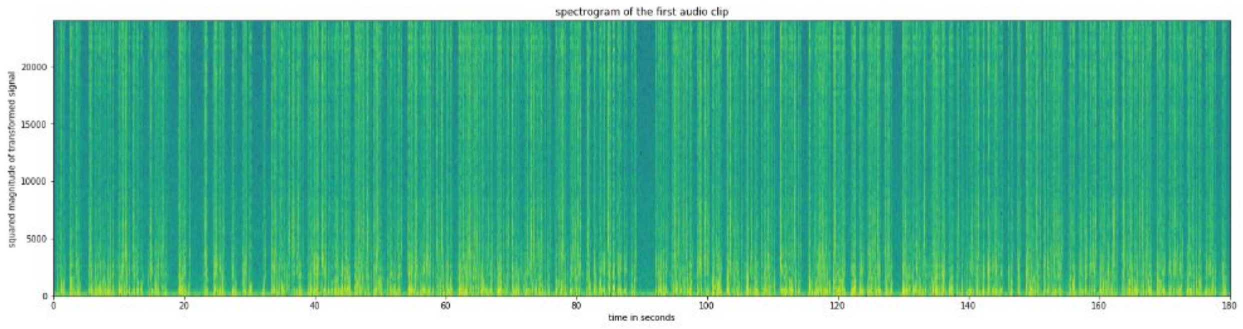


```
Sample 0 | Mean: 4.7798e-06 | Max: 0.921265 | Min: -0.568451 | Std Dev: 0.054303
Sample 1 | Mean: -0.000265499 | Max: 0.655914 | Min: -0.826019 | Std Dev: 0.0718488
Sample 2 | Mean: -2.16062e-06 | Max: 0.415649 | Min: -0.389709 | Std Dev: 0.0545702
Sample 3 | Mean: 3.46726e-06 | Max: 0.404083 | Min: -0.415283 | Std Dev: 0.0441317
Sample 4 | Mean: -1.86775e-06 | Max: 0.313354 | Min: -0.281219 | Std Dev: 0.0373589
Sample 5 | Mean: -1.82573e-06 | Max: 0.499573 | Min: -0.460754 | Std Dev: 0.0461007
Sample 6 | Mean: -4.45101e-07 | Max: 0.445007 | Min: -0.399445 | Std Dev: 0.0423917
Sample 7 | Mean: -9.76085e-06 | Max: 0.526245 | Min: -0.542969 | Std Dev: 0.0462241
Sample 8 | Mean: -1.01553e-05 | Max: 0.237549 | Min: -0.228149 | Std Dev: 0.0333488
Sample 9 | Mean: -5.33811e-06 | Max: 0.865387 | Min: -0.811523 | Std Dev: 0.0500287
```

However, it became evident very clearly that this information was not very helpful. Every mean value was very close to 0, due to the nature of sound waves. Frequencies are recorded in Hz (hertz) which is effectively cycles per second. Thus it makes sense that half of the frequencies would be above zero and half would be below zero, leaving the mean at zero.

To further illustrate how these frequencies look, every value was plotted using a waveplot. In other words, each value of the array is plotted. Oftentimes, more information can be extrapolated from audio files by looking at the magnitudes of frequencies within set-length duration windows. The Python library, matplotlib, includes a visualization tool for plotting the spectrograms. Below is the wave plot and the spectrogram for the first audio clip:





It is interesting to note that there exists a long pause near the very center of the clip. In the waveplot, this is seen as a flat line, whereas in the spectrogram, this is seen as an empty band. In the audio clip, this sounds like a long pause by the reader.

Algorithms and Techniques

Because all of the data exists in three minute clips, it will be important to extract shorter samples of the speaker clearly talking, so that long pauses and breaths will not be used as data to classify who is speaking. To first further preprocess the data, many short 200ms samples will need to be pulled from the data that contain an average frequency over some threshold to ensure that a voice can be heard during that short duration.

The first step will be to split the samples into training, testing, and validation sets. Using Scikit-Learn's train-test-split function, the data will be split using the following breakdowns: 60% of the data will be used for testing. 20% of the data will be used for cross-validation, and the remaining 20% will be used as a final testing set.

After all of the 200ms samples with high average frequencies are pulled, there are several possibilities for solving this classification task. One method is to run a supervised clustering algorithm to determine if patterns exist in the raw data which can differentiate between the different speakers. One common yet effective unsupervised algorithm is K-Nearest Neighbors (KNN).

Because there are so many features (significantly more features than samples in the sample space), it is likely that KNN will not be effective. After scoring the accuracy of KNN, the next step will be to create a neural network in TensorFlow to determine if a feed-forward neural network will be able to find the patterns in the data.

The next step will be to try creating a dataset by creating spectrograms for every single 200ms sample. Using that dataset, convolutional neural networks (also built in TensorFlow) will be able to look for the patterns on the spectrogram images.

Because the spectrogram images will be created using matplotlib, some control is lost in the various Short-time Fourier transformations (STFT). To create a more accurate dataset, the STFT data can be calculated using Librosa's stft function. The newly created transformed data will be complex and of the form a + bi, so there needs to exist a method to convert these arrays into something that can be plotted in matplotlib. To do this, a color wheel can be defined such that the complex numbers can be converted into Red, Green, and Blue values to create images with a depth of three. Finally, these newly created images (which are more precise than matplotlib's spectrogram plots) can be run through a new convolutional neural network to determine if a higher accuracy score can be achieved.

Benchmark

The problem of speaker identification is not new. There are entire textbooks written on the subject, and researchers finding stronger ways to identify speakers from data. In 2011, researchers at the University of Western Australia were able to obtain an accuracy of 94.5% by using Gaussian Mixture Models, a Universal Background Model, and Support Vector Machines (Togneri & Pullella, 2011). Because this projects' models are significantly less complex and limited, and the feature selection process will be automated rather than pulling specific utterances from the data, a realistic benchmark will be somewhere around 90%.

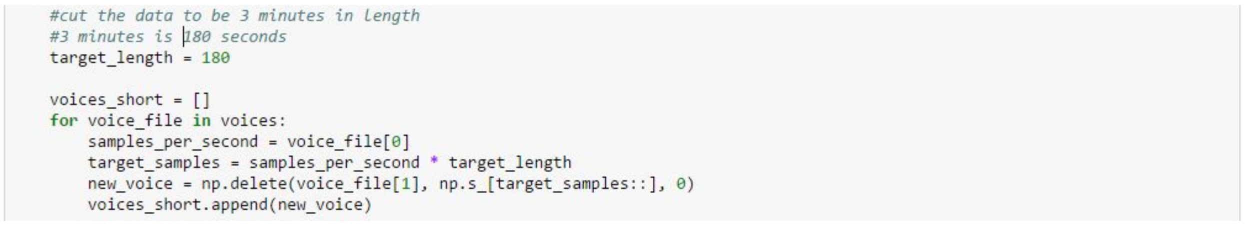
Source:
Togneri, R., Pullella, D. (2011). An Overview of Speaker Identification: Accuracy and Robustness Issues. *Circuits and Systems Magazine.* doi: 10.1109/MCAS.2011.941079

III. Methodology

Data Preprocessing

Crop the Files to Three Minutes

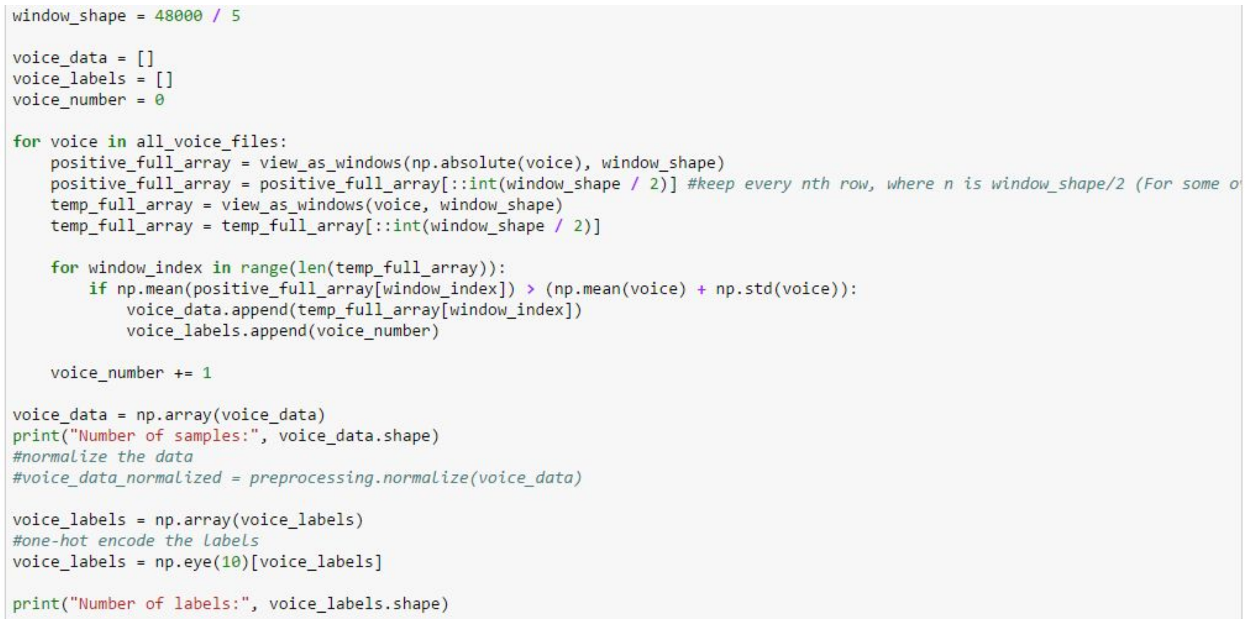
In order to crop each of the ten files into three minute sections, the input arrays were cut short using the numpy.delete method.



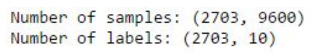
```
    #cut the data to be 3 minutes in length
    #3 minutes is 180 seconds
    target_length = 180

    voices_short = []
    for voice_file in voices:
        samples_per_second = voice_file[0]
        target_samples = samples_per_second * target_length
        new_voice = np.delete(voice_file[1], np.s_[target_samples::], 0)
        voices_short.append(new_voice)
```

Search for and Save 200ms-Length Samples of Audio

To determine the criteria for picking 200ms samples, the mean of each 200ms window was calculated, and if the mean was greater than the total sample mean + one standard deviation, it was included in the dataset. This ensures that only voice data was included and not pauses in between words.



```
window_shape = 48000 / 5

voice_data = []
voice_labels = []
voice_number = 0

for voice in all_voice_files:
    positive_full_array = view_as_windows(np.absolute(voice), window_shape)
    positive_full_array = positive_full_array[::int(window_shape / 2)] #keep every nth row, where n is window_shape/2 (For some o
    temp_full_array = view_as_windows(voice, window_shape)
    temp_full_array = temp_full_array[::int(window_shape / 2)]

    for window_index in range(len(temp_full_array)):
        if np.mean(positive_full_array[window_index]) > (np.mean(voice) + np.std(voice)):
            voice_data.append(temp_full_array[window_index])
            voice_labels.append(voice_number)

    voice_number += 1

voice_data = np.array(voice_data)
print("Number of samples:", voice_data.shape)
#normalize the data
#voice_data_normalized = preprocessing.normalize(voice_data)

voice_labels = np.array(voice_labels)
#one-hot encode the labels
voice_labels = np.eye(10)[voice_labels]

print("Number of labels:", voice_labels.shape)
```



```
Number of samples: (2703, 9600)
Number of labels: (2703, 10)
```

Create Labels for the Data

After creating labels for the data, the number of 200ms samples from each of the original ten voice samples was counted. Overall, roughly 250 samples were found for each voice. This means that an algorithm which randomly guesses would not achieve exactly 10% accuracy, as the likelihood for each sample is slightly unequal.

```
identity_matrix = np.identity(10)
number_of_samples = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for i in range(len(voice_labels)):
    for j in range(10):
        number_of_samples[j] += np.sum(np.all(np.equal(voice_labels[i], identity_matrix[j])))

for i in range(10):
    print("Number of samples from voice", i, ':', number_of_samples[i])
```

```
Number of samples from voice 0 : 249
Number of samples from voice 1 : 280
Number of samples from voice 2 : 265
Number of samples from voice 3 : 250
Number of samples from voice 4 : 307
Number of samples from voice 5 : 232
Number of samples from voice 6 : 352
Number of samples from voice 7 : 260
Number of samples from voice 8 : 294
Number of samples from voice 9 : 214
```

Split the Data into Training/Testing/Validation Sets

Data was split using SciKit-Learn's train_test_split method. The breakdown was 60% data for training, 20% data for validation, and 20% data for testing.

```
X_train, X_test, y_train, y_test = train_test_split(voice_data, voice_labels, test_size = 0.40, random_state = 7)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size = 0.5, random_state = 7)
```

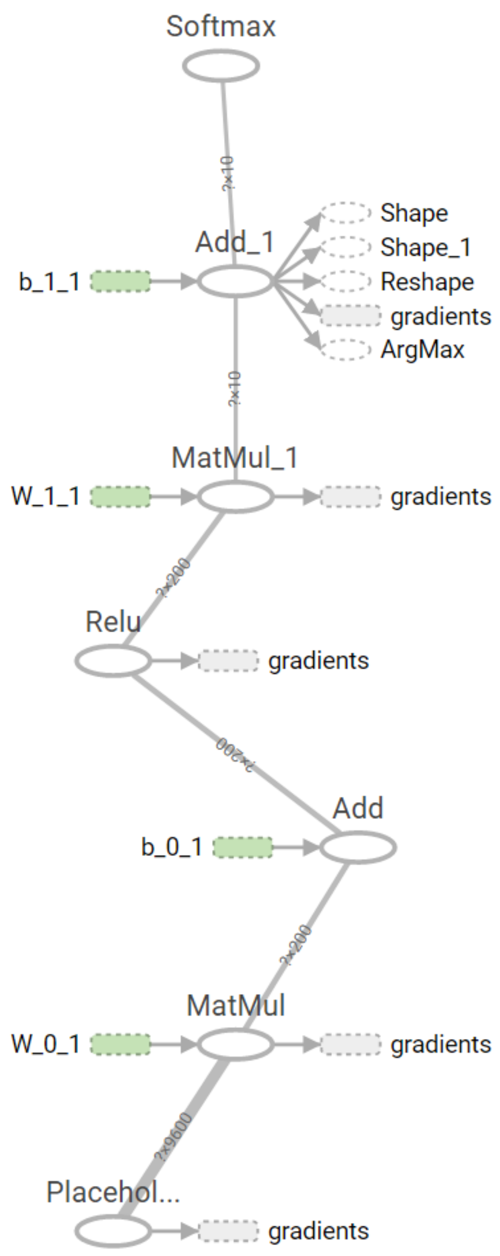
Implementation

Attempt #1: K-Nearest Neighbors

A quick K-Nearest Neighbors implementation using SciKit-Learn. The accuracy was calculated using SciKit-Learn's accuracy_score method.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

model = KNeighborsClassifier(n_neighbors = 10)
model.fit(X_train, y_train)
y_prediction = model.predict(X_val)

print("Validation accuracy:", accuracy_score(y_val, y_prediction))
```

Attempt #2: A Simple Neural Network



The first neural network used each value in an input array as one of the features, and had a very simple three-layer network architecture. The network was written in TensorFlow and was set up with the following specifications:

Input to Hidden Layer: Fully Connected with ReLU activation.

Hidden to Output Layer: Fully Connected with Softmax activation.

The input layer has 9600 nodes (48000 sample rate * (1/5 duration)).

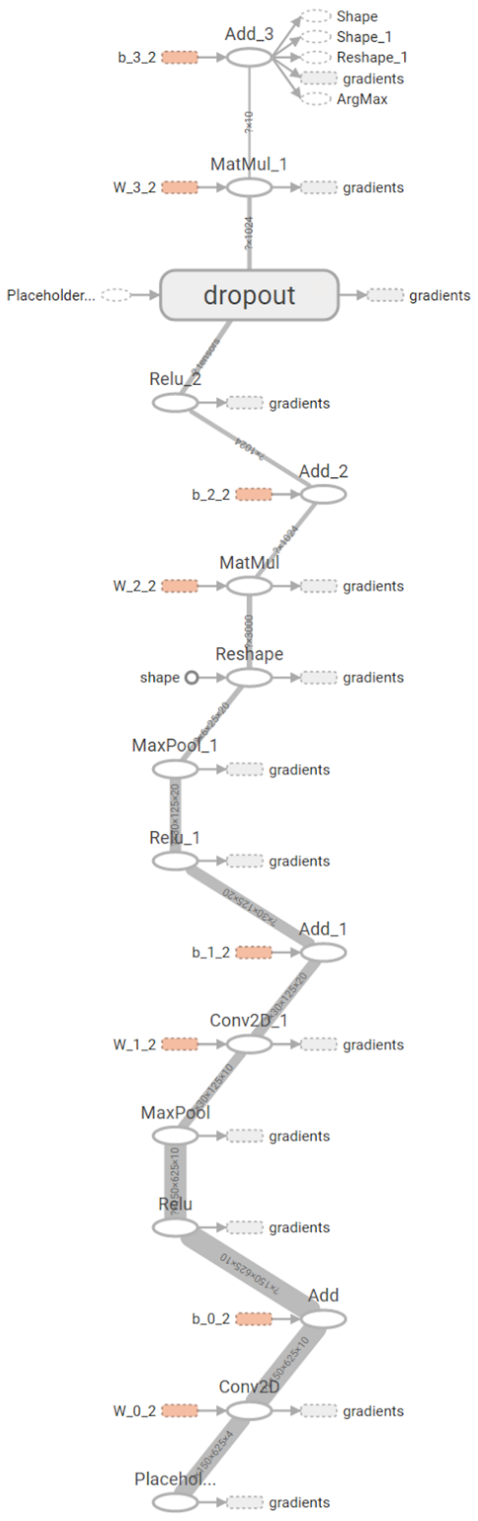
The hidden layer has 200 nodes.

The output layer has 10 nodes, corresponding to the ten different classes.

The graph to the left was made using TensorBoard and illustrates the network architecture. The weights were randomly initialized using a normal distribution, and the biases were initialized as zeroes.

Optimization was done using TensorFlow's implementation of Gradient Descent.

Attempt #3: A Convolutional Neural Network on Spectrograms



This network is slightly more complex than the last network. Again, this network was written using TensorFlow and had the following architecture:

Convolutional Layer with size 5x5x10 with ReLU activation and Max Pooling.

Another Convolutional Layer with size 5x5x20, again with ReLU activation and Max Pooling.

One Fully Connected Layer with 1024 nodes and ReLU activation.

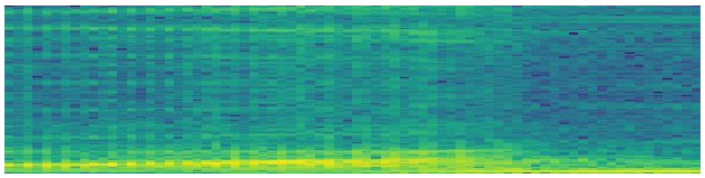
A dropout layer with 50% dropout.

Another Fully Connected Layer with 10 nodes and softmax activation, corresponding with the ten different classes.

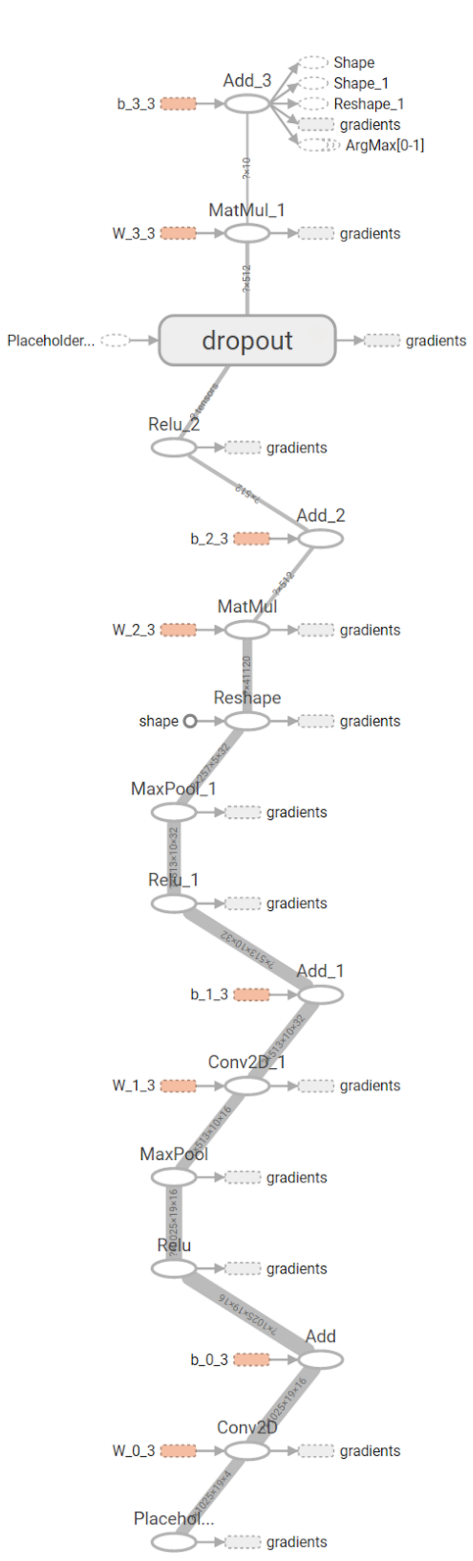
The inputs were the spectrograms created by matplotlib's specgram method. They had dimensions 150x625x4. An example of one of the spectrograms can be viewed below.

The graph to the left was made using TensorBoard and illustrates the network architecture. The weights were randomly initialized using a normal distribution with a standard deviation of 0.1, and the biases were initialized as 0.1.

Optimization was done using TensorFlow's implementation of the Adam Optimizer.



Attempt #4: A Convolutional Neural Network on RGB Images of STFT Data



This network was written using TensorFlow and had the following architecture:

Convolutional Layer with size 1x1x16 with ReLU activation and Max Pooling.

Another Convolutional Layer with size 4x4x32, again with ReLU activation and Max Pooling.

One Fully Connected Layer with 512 nodes and ReLU activation.

A dropout layer with 50% dropout.

Another Fully Connected Layer with 10 nodes and softmax activation, corresponding with the ten different classes.

Creating the images took several steps. First, the input arrays were transformed using Librosa's stft method, which transforms using a short-time Fourier transform (STFT). Because the resulting array contains complex numbers, the values are converted to HSL (hue, saturation, and lightness) values. From there, the HSL can be converted to RGB and saved as an image to run convolutions on. The images have dimensions 1025x19x4.

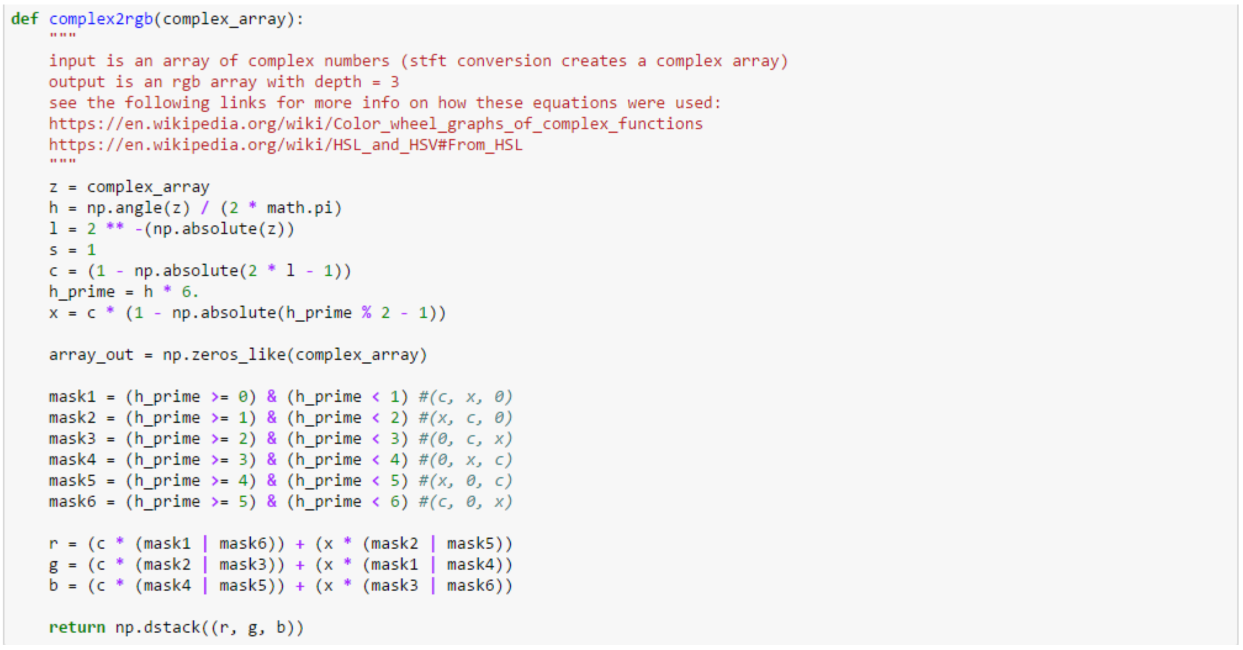
One primary advantage to creating images this way is that they can be converted back to the original audio format using inverse functions. There is no way to do this using matplotlib's spectrograms.

An example of one of the images can be viewed to the left.

The graph to the far left was made using TensorBoard and illustrates the network architecture. The weights were randomly initialized using a normal distribution with a standard deviation of 0.1, and the biases were initialized as 0.1.

Optimization was done using TensorFlow's implementation of the Adam Optimizer.

The code to convert from the STFT data to rgb images can be viewed below.



```python
def complex2rgb(complex_array):
    """
    input is an array of complex numbers (stft conversion creates a complex array)
    output is an rgb array with depth = 3
    see the following links for more info on how these equations were used:
    https://en.wikipedia.org/wiki/Color_wheel_graphs_of_complex_functions
    https://en.wikipedia.org/wiki/HSL_and_HSV#From_HSL
    """
    z = complex_array
    h = np.angle(z) / (2 * math.pi)
    l = 2 ** -(np.absolute(z))
    s = 1
    c = (1 - np.absolute(2 * l - 1))
    h_prime = h * 6.
    x = c * (1 - np.absolute(h_prime % 2 - 1))

    array_out = np.zeros_like(complex_array)

    mask1 = (h_prime >= 0) & (h_prime < 1) #(c, x, 0)
    mask2 = (h_prime >= 1) & (h_prime < 2) #(x, c, 0)
    mask3 = (h_prime >= 2) & (h_prime < 3) #(0, c, x)
    mask4 = (h_prime >= 3) & (h_prime < 4) #(0, x, c)
    mask5 = (h_prime >= 4) & (h_prime < 5) #(x, 0, c)
    mask6 = (h_prime >= 5) & (h_prime < 6) #(c, 0, x)

    r = (c * (mask1 | mask6)) + (x * (mask2 | mask5))
    g = (c * (mask2 | mask3)) + (x * (mask1 | mask4))
    b = (c * (mask4 | mask5)) + (x * (mask3 | mask6))

    return np.dstack((r, g, b))
```

The equations used to convert from the complex arrays to HSL were found on this wikipedia article: https://en.wikipedia.org/wiki/Color_wheel_graphs_of_complex_functions

The equations used to convert from the HSL data to RGB data were found on this wikipedia article: https://en.wikipedia.org/wiki/HSL_and_HSV#From_HSL

Refinement

For the first three attempts, not many small refinements were made, as it was predicted that the fourth attempt would be the strongest model, thus most time was spent refining that one. The following parameters were tweaked several times until the final model was created:

- Number of epochs
- Batch Size (the primary motivation for reducing this to 16 was due to running into memory errors)
- Dropout Probability (originally was at 80%, then 75%, then finally settled at 50% to reduce overfitting)
- The size of convolutions in the first two layers
    - After a lot of trial and error, eventually the fourth model utilized 1x1 convolutions in the first layer with a depth of 16, and 4x4 convolutions in the second layer with a depth of 32. Out of all the tested combinations, this provided high accuracy while also being manageable by the computer.
- Optimizer switched from Gradient Descent to Adam. The Adam optimizer yielded a much higher accuracy.

IV. Results

Model Evaluation and Validation

The first model, which used K-Nearest Neighbors only achieved a testing accuracy of 0.112 (~11%). Due to the distribution of the data, this does not appear to be much higher (if at all) than randomly guessing.

The second model, which used fully connected neural networks on the raw data achieved a much higher accuracy of 0.701 (~70%). While it is much higher, it is still not near the target goal of around 90%.

The third model was even stronger, which used convolutional neural networks on matplotlib-generated spectrograms. In fact, after fully training with 500 epochs, it achieved an accuracy of 0.885 (~89%), which was very close to the target goal of 90%.

The final model was the fourth model which used a convolutional neural network on RGB counterparts to a short-time Fourier Transformation of the raw data. **The final testing accuracy on the testing set was 0.939002 (~94%)** which is very respectable for a task this complex. In addition, because only 200ms of data was used for each sample, these results are very strong. Due to the short length of audio clip, some features could not be engineered, such as word use, syllables, timing of breaths, etc. Only pure audio data from speaking was used.

The testing set contained 541 previously unseen data samples, which shows that the model was able to generalize very well. In fact, it is clear that the model was not overfitting due to the similarities between the validation accuracy during training, and the final testing accuracy. In both cases, the accuracy was in the low 90's.

Justification

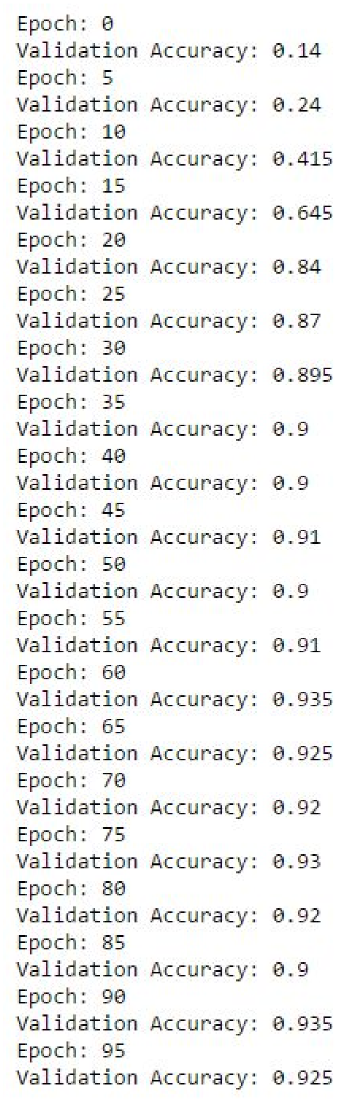
In the benchmark section of this report, it was explained that an accuracy of around 90% would be considered a strong model. There exist some very complex models which have obtained 94.5% accuracy, which shows that this convolutional model with 94% is also very strong. Thus, this model appears to be strong enough to have solved the initial problem.

The problem of speaker identification is not an easy problem to solve. There are classes on the subject and entire textbooks written on methods for solving this problem. 94% accuracy using 200ms of data is very impressive, as it is basically a model which can almost always correctly guess who is talking after only hearing less than a full word of audio.

V. Conclusion

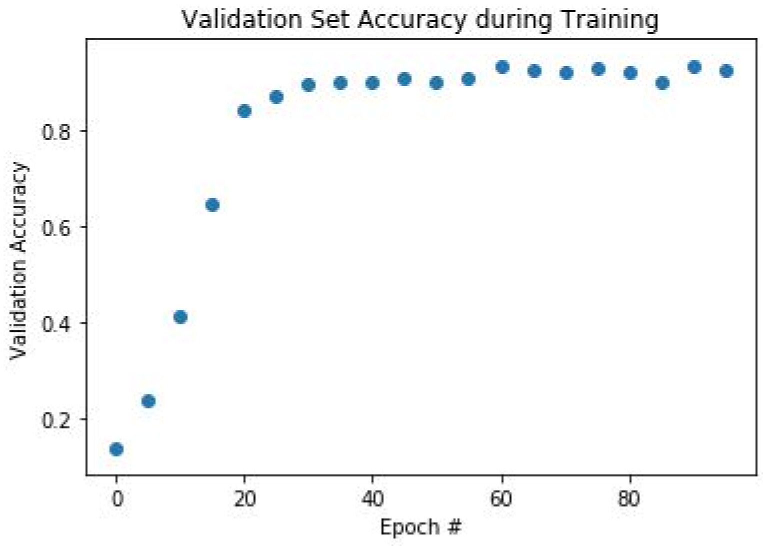
Free-Form Visualization

There are some important things to note when examining the final model used for this problem. One thing to note is how quickly the model is able to fully train. TensorFlow gives researchers the ability to easily utilize a computer's GPU to do the calculations (particularly on images) much faster. The training was done on an NVIDIA GTX 980 and only took 9 minutes and 41 seconds to fully train. Not only that, but validation accuracy hit ~90% after just 35 epochs, or after only around 3 minutes and 14 seconds.



```
Epoch: 0
Validation Accuracy: 0.14
Epoch: 5
Validation Accuracy: 0.24
Epoch: 10
Validation Accuracy: 0.415
Epoch: 15
Validation Accuracy: 0.645
Epoch: 20
Validation Accuracy: 0.84
Epoch: 25
Validation Accuracy: 0.87
Epoch: 30
Validation Accuracy: 0.895
Epoch: 35
Validation Accuracy: 0.9
Epoch: 40
Validation Accuracy: 0.9
Epoch: 45
Validation Accuracy: 0.91
Epoch: 50
Validation Accuracy: 0.9
Epoch: 55
Validation Accuracy: 0.91
Epoch: 60
Validation Accuracy: 0.935
Epoch: 65
Validation Accuracy: 0.925
Epoch: 70
Validation Accuracy: 0.92
Epoch: 75
Validation Accuracy: 0.93
Epoch: 80
Validation Accuracy: 0.92
Epoch: 85
Validation Accuracy: 0.9
Epoch: 90
Validation Accuracy: 0.935
Epoch: 95
Validation Accuracy: 0.925
```

Another important thing to note is how the model does not overtrain on the data. The entire dataset was broken into three different sets: A training set (60% of the data), a validation set (20% of the data), and a final testing set (20%) of the data. Nowhere is the training set being evaluated during training. While the model trains, the network prints out the current validation accuracy on 200 random samples from the validation set. Again, the validation data is kept completely separate from the training procedure. However, just having a validation set is not enough to ensure that the data is not being overfit to a model. If the hyperparameters are being adjusted such that the validation set achieves high accuracy, then there is no guarantee that the model is generalizable beyond that set. Thus introduces the need for a third set, the testing set. Below is a scatterplot showing the validation accuracy during training and how it compares to the final testing accuracy of 94%. Note that the validation accuracy is never much higher (overfitting) than the final testing accuracy, implying that the model is generalizable to new, unseen data.



One final interesting thing to note is how quickly the model performs when predicting on new data. When classifying 541 examples (the testing set), the model is able to predict and score all of the data in 0.835 seconds! That comes out to less than 0.002 seconds per 200ms sample.

Reflection

Below are the summarized steps of how a solution was found to the problem:

- Found 10 sources of high-quality audio of 10 different speakers.
- Cut the data down to 3 minutes.
- Searched for 200ms samples in those 3 minutes which contained a speaker’s voice clearly.
- Split the 200ms samples into training, validation, and testing sets.
- Created labels for the data.
- Models:
    - #1: Trained using SciKit-Learn’s K-Nearest Neighbors model.
    - #2: Created a simple three-layer neural network in TensorFlow.
    - #3: Convolutional Attempt 1
        - Used the 200ms audio samples to create spectrograms using matplotlib
        - Created a convolutional network in TensorFlow using the spectrograms as data
    - #4: Convolutional Attempt 2
        - Transform the 200ms audio data using short-time Fourier Transforms
        - Converted the new transformed data into HSL (hue, saturation, lightness) data
        - Converted the HSL data into RGB images
        - Created a convolutional network in Tensorflow using the RGB images as data.

One very interesting thing about this project is how quickly a solution was found. Originally, it was anticipated that there would need to exist significantly more feature engineering, such as:

- MFCC (Mel-frequency cepstral coefficient)
- Audio Spectrum Flatness
- Audio Spectrum Centroid
- Audio Spectrum Envelope
- Harmonicity
- Fundamental Frequency

However, the only data that was needed was the raw data, and the only calculations done on the data were a short-time Fourier transform (STFT), which was then turned into HSL data, which was finally turned into an RGB image. Another great thing about this approach is that by using inverse functions on the RGB images, one can get back to the original audio file. This is not possible through traditional spectrograms (recreating audio from spectrograms).

One difficulty emerged when looking to figure out how to turn the STFT data into images that can be used in convolutional networks. Once data is transformed using STFT processes, it exists as an array of complex numbers with a real portion and an imaginary portion, which is not plottable using matplotlib. Thus, it took a lot of research to find the solution to this difficulty, which was to first convert it to HSL data and then finally convert that into RGB data for matplotlib.

Improvement

The problem was solved using 3 minute clips of 10 different speakers. To create a stronger problem, there is some data that can be included to determine if such a high accuracy can still exist while maintaining the efficacy of the model. For instance, one could add more than 10 possible classes. It would be interesting to see how a model could do while classifying between 100 different speakers, or more!

Another way to improve a model would be to introduce messier data, with background sounds or extra noises which could conflict with the data. If such a model was to be created, modifications would need to occur when finding 200ms samples of data to train and test on. Recall, the method in which 200ms samples were found for this project was by looking at 200ms windows throughout a three minute clip and finding average frequencies that were higher than the overall (3 minute) average frequency plus a standard deviation.

If using the same data, there could exist some additional improvements as well. By examining the network architecture and modifying it by adding more convolutional layers or more depth, a higher accuracy could be achieved. The current accuracy suggests that there is certainly patterns in the data, so perhaps a stronger network architecture could pinpoint those patterns even more.

Another change to the current network and data could be to increase training time. However, as shown above, validation accuracy floated at around 90% after 3 minutes of training, so additional training time might not be very beneficial.