Simple Speech Command Recognition Using Neural Networks

Tyler Sagardoy
June 2018

This report details my project to build an algorithm that can understand a small library of 10 simple commands while being able to ignore, or distinguish from, unknown commands. Using Mel-Frequency Cepstral Coefficients to regularize and convey communicated information, I'll train a 1-dimensional convolutional neural network using RMSprop to minimize the categorical cross-entropy log loss to process and interpret the information to make a prediction as to the intended command.

I begin by discussing trends in command recognition, followed by a review of my project objective, a description of the data used for model training, and an outline of my project. I proceed to discuss steps taken during data preprocessing, including a brief overview of Mel-Frequency Cepstral Coefficients and how they are applied, before discussing the architectures of the 3 networks to be trained. Each network will be trained against hyperparameters in dropout rates, learning rates, and leaky rectified linear unit activation kernel alphas resulting in 81 separately trained models. From these models, I select the one that scores highest on accuracy when applied against an exclusive set of testing data and build a simple speech recognition algorithm from it.

Simple Speech Command Recognition, Rise & Market Trends

Within the past few years, voice command and activation has become a popular trend in user experience and design. Voice command allows for technologies to meet specific consumer demands, such as providing a user interface when hands or vision is occupied – a reason 61% of users state why they use voice command – or when one is in the car or on the go – the primary setting for voice usage in 55 percent of instances (Young, 2016). According to Google CEO Sundar Pichai, 20% of all queries made in 2016 on Google apps and Android devices were voice searches (Sterling, 2016).

The future looks bright for voice command. Comscore estimates that 50 percent of searches will be conducted by voice by 2020 (Young, 2016). While that figure may be highly optimistic, a study of 39 leading SEO experts identified voice usage as one of the top 3 important trends in search (Alameda Internet Marketing, 2016). While these estimates speak to the Internet search industry exclusively, the enthusiasm for voice activation in this industry portends demand for adoption in other industries.

As independent makers and entrepreneurs develop their products to feature this new and exciting interface method, or learn about it for the first time, many find that their needs for voice commands are minimal – only limited to a handful of basic operational commands. Many robust voice recognition algorithms are extraneous, costly, and unnecessary for many projects. Therefore, an opportunity exists to develop an open-source speech recognition algorithm intended for smaller developers and entry-level enthusiasts.

Project Purpose, Objectives & Objectives

The purpose of this report is to provide an algorithm that understands a small selection of simple audio commands. The selection of commands the algorithm should understand should be “Yes,” “No”, “Up”, “Down”, “Left”, “Right”, “On”, “Off”, “Stop” and “Go”.

I selected this project originally to compete in the TensorFlow Speech Recognition Challenge hosted by Kaggle and sponsored by Google; however, I used the data, guidance and competition objectives to outline my own

goals and I narrowed the scope of my project to accomplish my primary objective - training a simple 1-D convolutional neural network to accurately predict with at least 80% accuracy the library of ten words. For words and sounds that the algorithm isn't explicitly trained on, the algorithm will be trained to classify in a catch-all 'other' category.

Some of the principles I incorporated into my project include:

- Use the standard 16-bit [16000,1] integer tensor format for audio PCM-decoded at 16000 Mbps as an algorithm input.
- Output will be a [1, 11] tensor representing the prediction of the algorithm. Values will be between 0 and 1.
- Regularize decoded audio into tensors of Mel-Frequency Cepstral Coefficients (MFCCs).
- Objectively work to minimize the number of layers in the architecture to maintain some simplicity within the model and to minimize runtime.
- Use convolutional layers with a wide convolution window and moderate stride to identify speech patterns within inputs.
- Use pooling layers to reduce dimensionality between convolutional layers.
- Hyperparameters that I changed with each model were the dropout rate, the learning rate, and the alpha of the rectified linear unit kernels.

Dataset & Inputs

The algorithm will be trained and tested using the Speech Commands Dataset released by Google on August 3, 2017. The data contains 64,727 one-second audio clips of 30 short words. The audio files were crowdsourced by Google with the goal of collecting single-word commands (rather than words as said and used in conversation). A group of 20 core words were recorded with most speakers saying them 5 times. An additional group 10 words were recorded to help distinguish unrecognized words; most speakers recorded these words once. The core words consist of “Yes,” “No”, “Up”, “Down”, “Left”, “Right”, “On”, “Off”, “Stop”, “Go” and the numbers zero through nine. Auxiliary words consist of "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree" and "Wow".

This algorithm will take as inputs:

- A file of PCM-encoded data to be decoded into a 16-bit [16000, 1] integer tensor.
- Output will be a [1, 11] tensor representing the prediction of the algorithm. Values will be between 0 and 1.

Comparative Benchmark

With 10 command labels and an additional ‘non-command’ label included in the library, a naïve algorithm that outputs only one label would be correct 1 out of 11 times, or 9.1% of the time. For purposes of assessing whether a CNN has any predictive power, I will use this low 9.1% threshold. However, for practical and usability purposes, I hope to train a CNN that has at least 80% accuracy – a threshold that would have put me within the top quartile of competitors, indicating that my model performed better than 75.0% of all other models.

Evaluation Metrics

Models will be evaluated using Multiclass Accuracy, defined as the ratio of correct classifications over total classifications. The logic behind this evaluation is rather straight-forward: out of the number of testing cases, how many did the algorithm respond correctly?

Project Outline

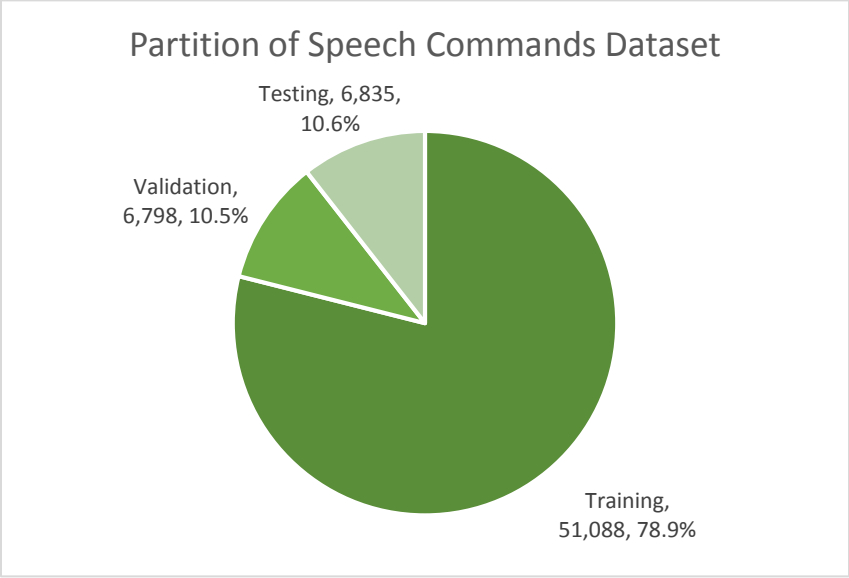
The procedural outline is as follows:

1. Download the Speech Commands Dataset from Google and Kaggle.
2. Decode the PCM-encoded audio files at 16000 Mbps into standardized 16-bit integer tensor of shape [16000, 1].
3. Preprocess data into tensors of Mel-Frequency Cepstral Coefficients (MFCCs)
4. Divide dataset into training, testing, and validation sets with inputs and one-hot encoded targets.
5. Define the architectures for the networks.
6. Compile and train models, retaining only the models with the best classification accuracy on the validation set
7. Use the test set to calculate Multiclass Accuracy and obtain at least 80.0% on one of the models trained.

1. Download Speech Commands Dataset from Google and Kaggle

As described by Kaggle, the Speech Commands Dataset is a set of 64,721 1-second .wav audio files, each consisting of 1-of-30 single spoken English words. These words are from a small set of commands and are spoken by a variety of different speakers. The audio files are organized into folders based on the word they contain, and this data set is designed to help train simple machine learning models. 6 additional audio files for different types of common noise were also included, though not employed in this project. The dataset for this project was downloaded at https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data. Data was released by Google on August 3, 2017.

A table describing the composition of the dataset, as well as the partitioned subsets, can be found on the next page. In general, each category of audio file makes up between 2.7% and 3.7% of the total dataset (or between 1,713 and 2,376 files each). The specific commands that the algorithm will learn comprise 36.6% of all files (or 23,682 files). With respect to how aggregate data is partitioned through the set, trainable command files are slightly underrepresented in the training set, while slightly overrepresented in the validation and testing sets. With regards to the size of the partition datasets, 78.9% of the data (51,088 files) will be used for training, 10.5% (6,798 files) for validation, and the remaining 10.6% (6,835 files) for testing.



DESCRIPTION OF SPEECH COMMANDS DATASET, 1-SECOND AUDIO FILES								
	All Files		Training Files		Validation Files		Testing Files	
Word	**#**	**%**	**#**	**%**	**#**	**%**	**#**	**%**
down	2,359	3.6%	1,842	3.6%	264	3.9%	253	3.7%
go	2,372	3.7%	1,861	3.6%	260	3.8%	251	3.7%
left	2,353	3.6%	1,839	3.6%	247	3.6%	267	3.9%
no	2,375	3.7%	1,853	3.6%	270	4.0%	252	3.7%
off	2,357	3.6%	1,839	3.6%	256	3.8%	262	3.8%
on	2,367	3.7%	1,864	3.6%	257	3.8%	246	3.6%
right	2,367	3.7%	1,852	3.6%	256	3.8%	259	3.8%
stop	2,380	3.7%	1,885	3.7%	246	3.6%	249	3.6%
up	2,375	3.7%	1,843	3.6%	260	3.8%	272	4.0%
yes	2,377	3.7%	1,860	3.6%	261	3.8%	256	3.7%
COMMAND FILES	**23,682**	**36.6%**	**18,538**	**36.3%**	**2,577**	**37.9%**	**2,567**	**37.6%**
bed	1,713	2.6%	1,340	2.6%	197	2.9%	176	2.6%
bird	1,731	2.7%	1,411	2.8%	162	2.4%	158	2.3%
cat	1,733	2.7%	1,399	2.7%	168	2.5%	166	2.4%
dog	1,746	2.7%	1,396	2.7%	170	2.5%	180	2.6%
eight	2,352	3.6%	1,852	3.6%	243	3.6%	257	3.8%
five	2,357	3.6%	1,844	3.6%	242	3.6%	271	4.0%
four	2,372	3.7%	1,839	3.6%	280	4.1%	253	3.7%
happy	1,750	2.7%	1,373	2.7%	189	2.8%	180	2.6%
house	1,742	2.7%	1,427	2.8%	173	2.5%	150	2.2%
marvin	1,746	2.7%	1,424	2.8%	160	2.4%	162	2.4%
nine	2,364	3.7%	1,875	3.7%	230	3.4%	259	3.8%
one	2,370	3.7%	1,892	3.7%	230	3.4%	248	3.6%
seven	2,377	3.7%	1,875	3.7%	263	3.9%	239	3.5%
sheila	1,734	2.7%	1,372	2.7%	176	2.6%	186	2.7%
six	2,369	3.7%	1,863	3.6%	262	3.9%	244	3.6%
three	2,356	3.6%	1,841	3.6%	248	3.6%	267	3.9%
tree	1,733	2.7%	1,374	2.7%	166	2.4%	193	2.8%
two	2,373	3.7%	1,873	3.7%	236	3.5%	264	3.9%
wow	1,745	2.7%	1,414	2.8%	166	2.4%	165	2.4%
zero	2,376	3.7%	1,866	3.7%	260	3.8%	250	3.7%
NON-COMMAND FILES	**41,039**	**63.4%**	**32,550**	**63.7%**	**4,221**	**62.1%**	**4,268**	**62.4%**
TOTAL FILES	**64,721**	**100.0%**	**51,088**	**100.0%**	**6,798**	**100.0%**	**6,835**	**100.0%**

2. Decode Audio Files

2a. Download and Import Necessary Python Libraries

Before I begin decoding the Speech Command Dataset audio files, I need to download the necessary libraries, modules and utilities into my programming environment, and import them into my iPython kernel. The libraries (and versions) I will be using are:

- SciPy (v0.19.1) - Python's scientific computing library; used in this project to directly decode PCM-encoded audio files at a rate of 16000 Mbps and to perform Discrete Cosine Transforms during MFCC calculations
- Numpy (v1.13.3) - library useful for handling and processing arrays of values and objects; used in this project to manage data and for performing Discrete Fourier Transforms during MFCC calculations

- Keras (v2.1.6) with TensorFlow (v1.8.0) backengine - Keras provides a high-level programming interface for deep learning and neural network development; Keras runs on top of TensorFlow, a library and engine for dataflow programming; used in this project to develop and train the neural networks and for one-hot encoding all target labels
- Glob - standard Python file searching library; used in this project to extrapolate filenames and paths for all audio files
- OS - standard Python operating system interface; used in this project to combine strings and values into valid file pathnames
- Math - standard Python mathematical functions; used in this project to calculate frequency bins in MFCC calculations

```
import os
from scipy.io import wavfile
from scipy.fftpack import dct
import numpy as np
import glob
import keras
from keras.utils import np_utils
import math
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.models import Sequential, load_model
from keras.layers import Dense, Activation, Conv1D, ZeroPadding1D, Dropout,
        AveragePooling1D
```

2b. Generate Lists of Training, Validation & Testing Filenames

The dataset downloaded from Kaggle contained two presorted lists of audio filepaths for validation and training lists. For purposes of transparency and ease, I made a third list of training files by first creating a list of all audio filepaths, then removing any filepaths that were found in either the validation or training lists. The list was saved with the other two lists. The training set contains 51,088 files; the validation set 6,798 files; the training set 6,835 files.

```
# set directory path
dir_path = os.path.join(os.path.dirname(os.path.abspath('__file__')), 'audio/audio')

# import pre-selected list of testing and validation audio filenames
testing_path = os.path.join(dir_path,'testing_list.txt')
testing_list = np.genfromtxt(testing_path, dtype=None, encoding=None)
print("Retrieved testing_list from "+ testing_path+ ". Size is "
    + str(testing_list.shape)+".")
validation_path = os.path.join(dir_path,'validation_list.txt')
validation_list = np.genfromtxt(validation_path, dtype=None, encoding=None)
print("Retrieved testing_list from "+ validation_path + ". Size is "
    + str(validation_list.shape)+".")

# generate exclusive list of training audio filenames
audio_path = os.path.join(dir_path, 'audio')
all_files_list = np.array(glob.glob(os.path.join(audio_path, '*', '*.wav')))
np.savetxt('audio/all_files_list.txt',all_files_list, encoding=None, fmt='%.100s')
audio_path_length = len(audio_path)+1
```

```
# create training_list by removing paths if in testing_list or validation_list
training_list = np.array([])
for index, value in np.ndenumerate(all_files_list):
    value = value[audio_path_length:]
    all_files_list[index] = value
    if np.isin([value],testing_list)==False and np.isin([value],
        validation_list)==False:
        training_list = np.append(training_list, all_files_list[index])
print(str(training_list.shape) + " is the shape of the training_list.")
np.savetxt('audio/training_list.txt',training_list, encoding=None, fmt='%.100s')
```

3. Calculating Mel-Frequency Cepstral Coefficients (MFCCs)

This project owes immense gratitude to Practical Cryptography for their excellent tutorial on Mel Frequency Cepstral Coefficients (MFCCs) (Practical Cryptography, 2018). MFCCs are a widely used tool in Speech Recognition and the logic behind them is fairly simple: human beings communicate by producing sounds through manipulation of their vocal tract (e.g. shape of tongue, position of tongue near teeth, shape of lips, etc.). That is, sound carries information about what is physically being communicated. In short durations, this information is manifested across ranges in the audio power spectrum. MFCCs describe the shape of this information.

To calculate MFCCs for each audio file, I first split the audio files into 2.5ms frames of 400 samples each. For this analysis, I added a stride of 200 samples - thus resulting in 79 frames total. For each frame, I performed a Discrete Fourier Transform to convert the signal from the time domain to the frequency domain. Since DFTs return complex numbers signifying both phase and amplitude, and because I'm only interested in the amplitudes of frequencies (and therefore real numbers), I calculate the periodogram estimate of the power spectrum by squaring each frequency power and dividing the result by the number of samples in the frame size. Only the first half of frequencies are kept; the rest are dropped.

Signals are now converted from frequency to Mels, which relates perceived pitch of a sound to its actual measured frequency. We aggregate these signals by frequency into overlapping bins of triangular filters to measure broader definitions of pitch. These bins span the reach of the human vocal spectrum (300hz to 8000hz). In typical MFCC calculations, between 20 and 40 filters are used in the filterbank. For this project, I use 26. The log of the summation of the energies in each filterbank is taken. We are left with a waveform described by 26 log filterbank energies. This waveform is decomposed using a Discrete Cosine Transform; the results are the MFCCs, describing the shape of the wavelength the power of pitches within the frame. In this analysis, only coefficients 2 - 13 are kept. In practice, typically the first half of coefficients are kept.

Once all frames are compiled, we are left with an array of 79 overlapping frames, each with 12 MFCCs.

```
# given filename, mfcc_conversion returns mel frequency cepstral coefficients array
# mfcc_conversion returns array of (79,12) representing 79 audio frames described by
# 12 coefficients

def mfcc_conversion(audio_path, filename, sample_rate = 16000, frame_size = 400,
    stride_size = 200, nfft = 512):
    # decode audio
    decoded_audio = audio_decoder(str(os.path.join(audio_path, filename)))
    audio = decoded_audio.reshape((16000,1))

    first_index = 0
    mfcc_coefficients = np.empty((0,12))
    # apply the following for each signal frame
    while first_index <= audio.shape[0]-frame_size:
        last_index = first_index+frame_size
        frame = audio[first_index:last_index,:]

        # calculate discrete Fourier transform
        frame = np.fft.fft(frame, n = nfft, axis=0)

# calculate the periodogram estimate of the power spectrum; drop last half of values
        power_spectrum = np.absolute(np.square(frame))/frame_size
        power_spectrum = power_spectrum[0:int(nfft/2),:].astype(float)
        power_spectrum = power_spectrum.reshape((power_spectrum.shape[0],))
        # print(power_spectrum)

        # apply the mel filterbank to power spectra, sum the energy in each filter
            # frequencies on which to define mel filterbanks
        mel_freqs = np.array([300, 383.4, 473.8, 571.7, 677.8, 792.7, 917.3, 1052.2,
                              1198.3, 1356.7, 1528.3, 1714.2, 1915.6, 2133.7, 2370.1,
                              2626.3, 2903.7, 3204.4, 3530.1, 3882.9, 4265.2,4679.4,
                              5128.2, 5614.4, 6141.1, 6711.8, 7330.1, 8000])
                              .astype(float)
        vfunc = np.vectorize(bin_index)
        mel_bins = vfunc(mel_freqs, nfft=nfft, sample_rate=16000)
        # print(mel_bins)

           # calculate filterbank
        mel_filterbank = np.empty((26,256))
        for i in range(0,mel_filterbank.shape[0]):
            for j in range(0,mel_filterbank.shape[1]):
                mel_bin_min = mel_bins[i]
                mel_bin_mid = mel_bins[i+1]
                mel_bin_max = mel_bins[i+2]
                if j >= mel_bin_min and j < mel_bin_mid:
                    filter = float(j - mel_bin_min) / float(mel_bin_mid-mel_bin_min)
                elif j >= mel_bin_mid and j < mel_bin_max:
                    filter = float(mel_bin_max- j) / float(mel_bin_max-mel_bin_mid)
                else:
                    filter = float(0)
                mel_filterbank[i,j] = filter
```

CONTINUED ON NEXT PAGE...

```
            # apply filterbank to power spectra and calculate log bank energies
        logbankenergies = np.zeros((mel_filterbank.shape[0]))
        for i in range(0, mel_filterbank.shape[0]):
            mel_filters = mel_filterbank[i,:]
            bankenergy = np.dot(power_spectrum,mel_filters)+1
            logbankenergies[i] = np.log(bankenergy)


        # take the discrete cosine transform of the log filterbank energies
        log_dct = dct(logbankenergies)

        # saving DCT coefficients 2-13; discard rest
        log_dct = np.transpose(log_dct[1:13])
        i = first_index/stride_size
        mfcc_coefficients = np.vstack((mfcc_coefficients, log_dct))

        # set up next frame
        first_index = first_index + stride_size

    return mfcc_coefficients

# calculates bin index given frequency and sample_rate
def bin_index(frequency, nfft=512, sample_rate=16000):
    bin = math.floor((nfft+1)*frequency/sample_rate)
    return bin

# decodes audio given a file name
def audio_decoder(filename):
    rate, data = wavfile.read(filename)
    data = np.array(data)
    if data.shape[0]<=16000:
        difference = 16000 - data.shape[0]
        data = np.append(data,np.zeros((difference,)))
        data = np.transpose(data.reshape((16000,1)))
        data = data.astype(int)
    return data
```

4a. Calculating training, validation, and testing MFCC datasets

Each list of audio files is decoded into MFCC data arrays and saved to disk.

```
# initialize MFCC data arrays in memory
x_training = np.empty((0, 79, 12), float)
x_validation = np.empty((0, 79, 12), float)
x_testing = np.empty((0, 79, 12), float)

audio_path = str(os.path.join('audio','audio'))

# decode training_list
for i in range(0, training_list.shape[0]):
    mfccs = mfcc_conversion(training_list[i])
    mfccs = np.expand_dims(mfccs, axis=0)
    x_training = np.vstack((x_training, mfccs))
np.save('data/x_training', x_training)
print("Training list audio decode complete.")

# decode validation_list
for i in range(0, validation_list.shape[0]):
    mfccs = mfcc_conversion(validation_list[i])
    mfccs = np.expand_dims(mfccs, axis=0)
    x_validation = np.vstack((x_validation, mfccs))
np.save('data/x_validation', x_validation)
print("Validation list audio decode complete.")

# decode testing_list
for i in range(0, testing_list.shape[0]):
    mfccs = mfcc_conversion(testing_list[i])
    mfccs = np.expand_dims(mfccs, axis=0)
    x_testing = np.vstack((x_testing, mfccs))
np.save('data/x_testing', x_testing)
print("Testing list audio decode complete.")
```

4b. Categorizing Targets

The models will train on targets reflecting 1 of 11 types of classifications - the 10 words and an 'other' catchall category for unknown sounds. To get these target labels, I look at the parent folder of the audio files and classify as one of 11 numbers. I then use Keras to turn my integer labels into one-hot encoded vectors of size 11 and add a dimension at axis 1 to allow the model to process. I save the results to disk.

```
# audio targets are determined by the name of their parent folder
def audio_categorizer(x_arr):
    y_arr=np.array([])
    for index, value in np.ndenumerate(x_arr):
        cat_name = os.path.dirname(value)
        if cat_name == "yes":
            y_arr = np.append(y_arr,1)
        elif cat_name == "no":
            y_arr = np.append(y_arr,2)
        elif cat_name == "up":
            y_arr = np.append(y_arr,3)
        elif cat_name == "down":
            y_arr = np.append(y_arr,4)
        elif cat_name == "left":
            y_arr = np.append(y_arr,5)
        elif cat_name == "right":
            y_arr = np.append(y_arr,6)
        elif cat_name == "on":
            y_arr = np.append(y_arr,7)
        elif cat_name == "off":
            y_arr = np.append(y_arr,8)
        elif cat_name == "stop":
            y_arr = np.append(y_arr,9)
        elif cat_name == "go":
            y_arr = np.append(y_arr,10)
        else:
            y_arr = np.append(y_arr,0)
    return y_arr

# categorize training labels
y_training = audio_categorizer(training_list)
y_validation = audio_categorizer(validation_list)
y_testing = audio_categorizer(testing_list)

# applying a one-hot encoding scheme
y_training = np_utils.to_categorical(y_training, 11)
y_validation = np_utils.to_categorical(y_validation, 11)
y_testing = np_utils.to_categorical(y_testing, 11)

# add dimension for network processing
y_training = np.expand_dims(y_training, axis=1)
y_validation = np.expand_dims(y_validation, axis=1)
y_testing = np.expand_dims(y_testing, axis=1)

print('The shape of training targets is '+str(y_training.shape))
print('The shape of validation targets is '+str(y_validation.shape))
print('The shape of testing targets is '+str(y_testing.shape))

# save MFCC datasets for future use
np.save('data/y_training', y_training)
np.save('data/y_validation', y_validation)
np.save('data/y_testing', y_testing)
```

5. Model Architecture

For model training and evaluation, I developed three different model architectures representing three different types of reasoning. In developing these architectures, I sought to balance these (often-competing) principles:

- Objectively work to minimize the number of layers in the architecture to maintain some simplicity within the model and to minimize runtime
- Use convolutional layers with a wide-enough convolution window and stride to identify patterns
- Use pooling layers to reduce dimensionality between convolutional layers

Each of the network uses 1-dimensional convolutional layers that slide along the time axis, convoluting MFCCs and reducing the size of the first dimension of the audio data. The idea behind each of these networks is to recognize patterns among MFCCs across time. Differentiating features among these three models were the size of the convolution windows, the overlapping stride of the windows, the number of filters, and the number of layers. The three networks I built are:

- Large Windows/Few Layers/Many Filters - this model consists of 1 padding layer, 2 convolutional layers with window size of 10 and stride of 5 each, an average pooling layer, and a dense output layer. The first and second convolutional layers have 50 and 100 filters, respectively. Dropout layers have been added after each convolutional layer.
- Small Windows/Fewer Filters - this model consists of 5 sets of convolutional layers with window size of 3 and strides of 2 followed by a dense output layer. Each convolutional layer has 20 filters. Dropout layers have been added after each convolutional layer.
- Moderate Windows/Increasing Filters - this model consists of a set of 1 padding and 2 convolutional layers with window size 4 and stride 2, followed by another set of 1 padding and 3 convolutional layers with small window sizes. The number of filters increase with each additional layer until the final dense output layer. Dropout layers have been added after each convolutional layer.

Hyperparameters that I changed with each model were:

- the dropout rate (0%,25%,50%),
- the learning rate (0.01, 0.05, 0.10), and
- the alpha of the rectified linear unit kernels (0,-0.1, -0.5).

Therefore, the total number of models evaluated was 81.

```
# Model_1 is the Large Windows/Many Filters network
def model_1_architecture(dropout_rate=0):
    model = Sequential()
    model.add(ZeroPadding1D(input_shape=(79, 12),padding=(0,1)))
    model.add(Conv1D(50, kernel_size=10, strides=5, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(100, kernel_size=10, strides=5, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(AveragePooling1D(pool_size=2))
    model.add(Dense(11, activation='softmax'))
    return model

# Model_2 is the Small Windows/Fewer Filters network
def model_2_architecture(dropout_rate=0):
    model = Sequential()
    model.add(Conv1D(20, kernel_size=3, strides=2, activation='relu', input_shape=(7
9, 12)))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(20, kernel_size=3, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(20, kernel_size=3, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(20, kernel_size=3, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(20, kernel_size=3, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(11, activation='softmax'))
    return model

# Model_3 is the Moderate Windows/Increasing Filters network
def model_3_architecture(dropout_rate=0):
    model = Sequential()
    model.add(ZeroPadding1D(input_shape=(79, 12),padding=(0,3)))
    model.add(Conv1D(10, kernel_size=4, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(20, kernel_size=4, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(ZeroPadding1D(padding=(0,1)))
    model.add(Conv1D(40, kernel_size=4, strides=2, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(60, kernel_size=3, strides=3, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Conv1D(80, kernel_size=3, strides=1, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(11, activation='softmax'))
    return model
```

6. Model Training

For network training, I trained the model on random batches of 1000 files per epoch over 60 epochs to ensure that most, if not all, of my training dataset of 51,088 is used. I used the RMSprop optimization algorithm to minimize categorical cross-entropy log loss and validated my models comparing both log loss from training and validation. Only the models that minimize log loss are saved to disk. I also enabled early stopping to be triggered after 10 epochs of no improvement in log loss.

Hyperparameters - dropbout rates, learning rates, and the leaky rectified linear unit alpha - were toggled for each of the 3 models resulting in 81 trained models.

```
def model_1_train(x_training, y_training,x_validation,y_validation, dropout_rate=0.0,
  learning_rate=0.01, relu_alpha=0.0, batch_size=1000, epochs=60):
    model = model_1_architecture(dropout_rate)
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
      metrics=['accuracy'])
    dir_path = os.path.join(os.path.dirname(os.path.abspath('__file__')),'models')
    file_name = 'model1_dr'+str(dropout_rate)+'_lr'+str(learning_rate)+'_ra'
      + str(relu_alpha)+'.hdf5'
    checkpoint_file = os.path.join(dir_path, file_name)
    checkpointer = ModelCheckpoint(filepath=checkpoint_file, verbose=1,
       save_best_only=True)
    early_stopper = EarlyStopping(patience=10, verbose=1)
    hist = model.fit(x_training, y_training, batch_size=batch_size, epochs=epochs,
       validation_data=(x_validation, y_validation), callbacks=[checkpointer,
       early_stopper], verbose=2, shuffle=True)

def model_2_train(x_training, y_training,x_validation,y_validation,dropout_rate=0.0,
  learning_rate=0.01, relu_alpha=0.0, batch_size=1000, epochs=60):
    model = model_2_architecture(dropout_rate)
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
      metrics=['accuracy'])
    dir_path = os.path.join(os.path.dirname(os.path.abspath('__file__')),'models')
    file_name = 'model2_dr'+str(dropout_rate)+'_lr'+str(learning_rate)+'_ra'
      +str(relu_alpha)+'.hdf5'
    checkpoint_file = os.path.join(dir_path, file_name)
    checkpointer = ModelCheckpoint(filepath=checkpoint_file, verbose=1,
      save_best_only=True)
    early_stopper = EarlyStopping(patience=10, verbose=1)
    hist = model.fit(x_training, y_training, batch_size=batch_size, epochs=epochs,
       validation_data=(x_validation, y_validation), callbacks=[checkpointer,
       early_stopper], verbose=2, shuffle=True)

def model_3_train(x_training, y_training,x_validation,y_validation,dropout_rate=0.0,
  learning_rate=0.01, relu_alpha=0.0, batch_size=1000, epochs=60):
    model = model_3_architecture(dropout_rate)
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
        metrics=['accuracy'])
    dir_path = os.path.join(os.path.dirname(os.path.abspath('__file__')),'models')
    file_name = 'model3_dr'+str(dropout_rate)+'_lr'+str(learning_rate)+'_ra'
      +str(relu_alpha)+'.hdf5'
    checkpoint_file = os.path.join(dir_path, file_name)
    checkpointer = ModelCheckpoint(filepath=checkpoint_file, verbose=1,
       save_best_only=True)
    early_stopper = EarlyStopping(patience=10, verbose=1)
    hist = model.fit(x_training, y_training, batch_size=batch_size, epochs=epochs,
       validation_data=(x_validation, y_validation), callbacks=[checkpointer,
       early_stopper], verbose=2, shuffle=True)
```

Below is my test data sorted by model architecture, then by dropout rate, by learning rate, and by ReLU alpha.

TESTING RESULTS - ACCURACY SCORES					
Dropout Rate	**Learning Rate**	**ReLU Alpha**	**Model 1**	**Model 2**	**Model 3**
0.00	0.01	0.00	84.61%	82.27%	83.48%
		0.10	83.44%	80.01%	83.61%
		0.50	84.11%	82.11%	83.61%
	0.05	0.00	83.04%	82.11%	82.08%
		0.10	84.96%	81.95%	82.79%
		0.50	84.67%	81.49%	82.38%
	0.10	0.00	84.64%	82.21%	82.77%
		0.10	82.93%	82.44%	82.15%
		0.50	84.23%	81.07%	83.44%
0.25	0.01	0.00	87.94%	71.31%	83.38%
		0.10	87.21%	71.02%	83.19%
		0.50	87.96%	72.28%	83.44%
	0.05	0.00	**88.21%**	72.32%	82.84%
		0.10	87.27%	74.48%	83.79%
		0.50	87.56%	73.12%	83.50%
	0.10	0.00	87.70%	72.11%	82.47%
		0.10	**88.21%**	70.04%	82.15%
		0.50	86.01%	74.92%	82.91%
0.50	0.01	0.00	84.73%	62.44%	64.89%
		0.10	84.80%	62.44%	66.99%
		0.50	84.71%	62.44%	66.19%
	0.05	0.00	85.41%	62.44%	65.57%
		0.10	84.51%	62.44%	67.17%
		0.50	85.30%	62.44%	65.31%
	0.10	0.00	85.21%	62.44%	65.75%
		0.10	84.65%	62.44%	65.85%
		0.50	84.77%	62.44%	66.47%

Based on the test data above, two models did equally best in categorizing the test data and beat my 80.0% accuracy objective with 82.2%:

1. Model 1 architecture with a dropout rate of 0.25, learning rate of 0.05, and ReLU Alpha of 0.0
2. Model 1 architecture with a dropout rate of 0.25, learning rate of 0.1, and ReLU Alpha of 0.1

Upholding the concept of the simpler the better, I select the first model since it has a more conservative learning rate and its kernel is a simpler non-leaky ReLU.

A few observations from my test results:

- Model architecture 1 was vastly superior to the other two models with all scores above the average of the testing data. This suggests to me that a greater convolution window, stride, and number of filters may render direct improvement in classification accuracy in the model
- A dropout rate of 0.25 was better than both 0 and 0.5 which isn't surprising considering that a rate of 0 leads to overfitting of test data and a substantial dropout rate will impede training as backpropagated calculations will fail to persist. This suggests an appropriate range of dropout rate should be narrower and skewed toward lower values greater than 0.
- In general, a learning rate of 0.05 tested better than rates at 0 and 0.1, suggesting that models tested in a narrower range including 0.05 may train and test better.

Final Algorithm

Below is the final algorithm, using both the mfcc_conversion utility to convert audio files into MFCC inputs, and the selected neural network to understand the speech command. Argmax is used to select the index with the highest classification probability.

```
def algo(audio_path, filename):
    print("My prediction for "+filename+" is...")
    model_path = os.path.join(os.path.dirname(os.path.abspath('__file__')),'models',
'model1_dr0.25_lr0.1_ra0.hdf5')
    model = load_model(model_path)
    mfccs = mfcc_conversion(audio_path, filename).reshape((1,79,12))
    prediction = np.argmax(model.predict(mfccs))
    if prediction == 1:
        prediction = "yes"
    elif prediction == 2:
        prediction = "no"
    elif prediction == 3:
        prediction = "up"
    elif prediction == 4:
        prediction = "down"
    elif prediction == 5:
        prediction = "left"
    elif prediction == 6:
        prediction = "right"
    elif prediction == 7:
        prediction = "on"
    elif prediction == 8:
        prediction = "off"
    elif prediction == 9:
        prediction = "stop"
    elif prediction == 10:
        prediction = "go"
    else:
        prediction = "UNKNOWN"
    print(prediction)
```

I demonstrate the algorithm's ability on some sample files of my own voice. Ultimately, 28 out of 30 commands (93.3% precision) were correctly interpreted with the 2 failures being 2 of the trained commands (8 of 10 commands were correctly understood resulting in 80.0% precision). From those two failures, I reviewed the audio files and made a couple observations. Regarding the left.wav failure, I spoke with an awkward inflection which probably skewed the MFCCs. Regarding up.wav, I spoke very quickly and abruptly toward the end of the audio file (not unlike a tic or grunt) which suggests to me that the algorithm didn't have enough information to make an accurate classification. Perhaps if I spoke just a little slower, the algorithm would've understood correctly.

RESULTS FROM TESTING ALGORITHM WITH MY OWN VOICE				
Filename	**Correct Target**	**Predicted**	**Result**	**Notes**
bed.wav	unknown	unknown	**Success**	Some data not understood by SciPy.
bird.wav	unknown	unknown	**Success**	
cat.wav	unknown	unknown	**Success**	
dog.wav	unknown	unknown	**Success**	
down.wav	down	down	**Success**	
eight.wav	unknown	unknown	**Success**	
five.wav	unknown	unknown	**Success**	
four.wav	unknown	unknown	**Success**	
go.wav	go	go	**Success**	
house.wav	unknown	unknown	**Success**	
happy.wav	unknown	unknown	**Success**	
left.wav	left	unknown	**Fail**	Spoken with awkward inflection.
marvin.wav	unknown	unknown	**Success**	
nine.wav	unknown	unknown	**Success**	
no.wav	no	no	**Success**	
off.wav	off	off	**Success**	
on.wav	on	on	**Success**	
one.wav	unknown	unknown	**Success**	
right.wav	right	right	**Success**	
seven.wav	unknown	unknown	**Success**	
sheila.wav	unknown	unknown	**Success**	
six.wav	unknown	unknown	**Success**	
stop.wav	stop	stop	**Success**	
three.wav	unknown	unknown	**Success**	
tree.wav	unknown	unknown	**Success**	
two.wav	unknown	unknown	**Success**	
up.wav	up	unknown	**Fail**	Spoken very quickly and abruptly.
wow.wav	unknown	unknown	**Success**	
yes.wav	yes	yes	**Success**	
zero.wav	unknown	unknown	**Success**	

Summary, Conclusion & Final Thoughts

To summarize, the purpose of this project was to build an algorithm that could understand a small library of 10 simple commands while being able to ignore or not interpret unknown commands. Using Mel-Frequency Cepstral Coefficients as the medium with which to regularize and convey communicated information, I trained a 1-dimensional convolutional neural network, using RMSprop to minimize the categorical cross-entropy log loss, to process and interpret the information to make a prediction as to the intended command.

3 different network architectures were designed and trained. Hyperparameters - dropout rates, learning rates, and the leaky rectified linear unit alpha - were toggled for each of the 3 architectures resulting in 81 trained models. All 81 models were tested and the algorithm that scored highest - 88.2% - was selected. This score was higher than the 80.0% I originally sought to achieve.

In conclusion, I want to leave with a few ideas I would like to incorporate into this project - or a similar project - in a future date:

- Use networks, or long-short term memory networks, to process the MFCCs. While I used convolutional neural networks to achieve my goal, perhaps other network architectures would be more applicable such as RNNs and LSTMs which are widely used in signal processing and speech recognition. Perhaps I'll repeat my procedures here with such architectures to see if I can improve upon my current accuracy.
- Augment and distort signals for testing. While I originally intended to add some random digital distortion to my input data to account for more realistic use cases and to aid against overfitting the models to only the noiseless cases, I ultimately decided this wasn't necessary for my purposes here. However, should I decide to incorporate distinctions between the commands, silence, and noise, then I can script the necessary code to do so.
- Reduce the sample rate of inputs to see if I can maintain accuracy with less information. While I trained the model with audio files decoded at 16000Mbps, perhaps I can achieve the same accuracy - or perhaps improve upon - with audio sampled at a lower bit rate - maybe 8000Mbps, for example. If successful, this would allow my algorithm to be used with electronics or networks with a more limited capacity.

Works Cited

Alameda Internet Marketing. (2016, 07 06). *39 Experts Share Their Top 3 SEO Trends for 2017 and Beyond*. Retrieved from Alameda Internet Marketing: https://alamedaim.com/seo-trends/

Practical Cryptography. (2018, January 15). *Mel Frequency Cepstral Coefficient (MFCC) tutorial*. Retrieved from Practical Cryptography: http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/

Sterling, G. (2016, 05 18). *Google says 20 percent of mobile queries are voice searches*. Retrieved from Search Engine Land: https://searchengineland.com/google-reveals-20-percent-queries-voice-queries-249917

Young, W. (2016, 06 20). *The voice search explosion and how it will change local search*. Retrieved from Search Engine Land: https://searchengineland.com/voice-search-explosion-will-change-local-search-251776