Creating a Basic Speech Recognition System

Phillip Spratling

Introduction

Speech recognition technology has come a long way in the last couple decades. Until recently, it was predominantly performed using methods such as hidden Markov models, but now deep learning neural networks have proven to be much more effective. Because of the improvement in performance, speech recognition in devices is begging to show up everywhere. Previously used mostly in military applications and telephone call processing, you can now find sophisticated speech recognition being used in smartphones, cars, televisions, and even refrigerators.

Despite the recent surge in popularity in speech recognition technology, most independent makers and entrepreneurs have had difficulty building a simple speech detector. Neural networks require a significant amount of data to be effective, and gathering (not to mention processing and cleaning) voice clips of hundreds or thousands of people saying multiple phrases multiple times is not exactly easy. Thankfully, TensorFlow recently released the Speech Commands Datasets, which include 65,000 one-second long utterances of 30 short words, by thousands of different people.

In this project, I will be using this dataset to build a speech recognition algorithm that understands 10 of these words and will understand whether a sound clip is silent as well. It will also know to classify sounds that aren't one of these 11 categories as "unknown". These words were chosen to be used as a speech interface, so they are mostly commands such as "yes," "no," "stop," "go," etc. The goal is to build an system that could be used in a simple speech recognition application, or for the foundation of a more sophisticated application.

To build the algorithm, I will process the sound clips as spectrograms and into Mel-frequency cepstral coefficients for use as input into my neural networks. These can be

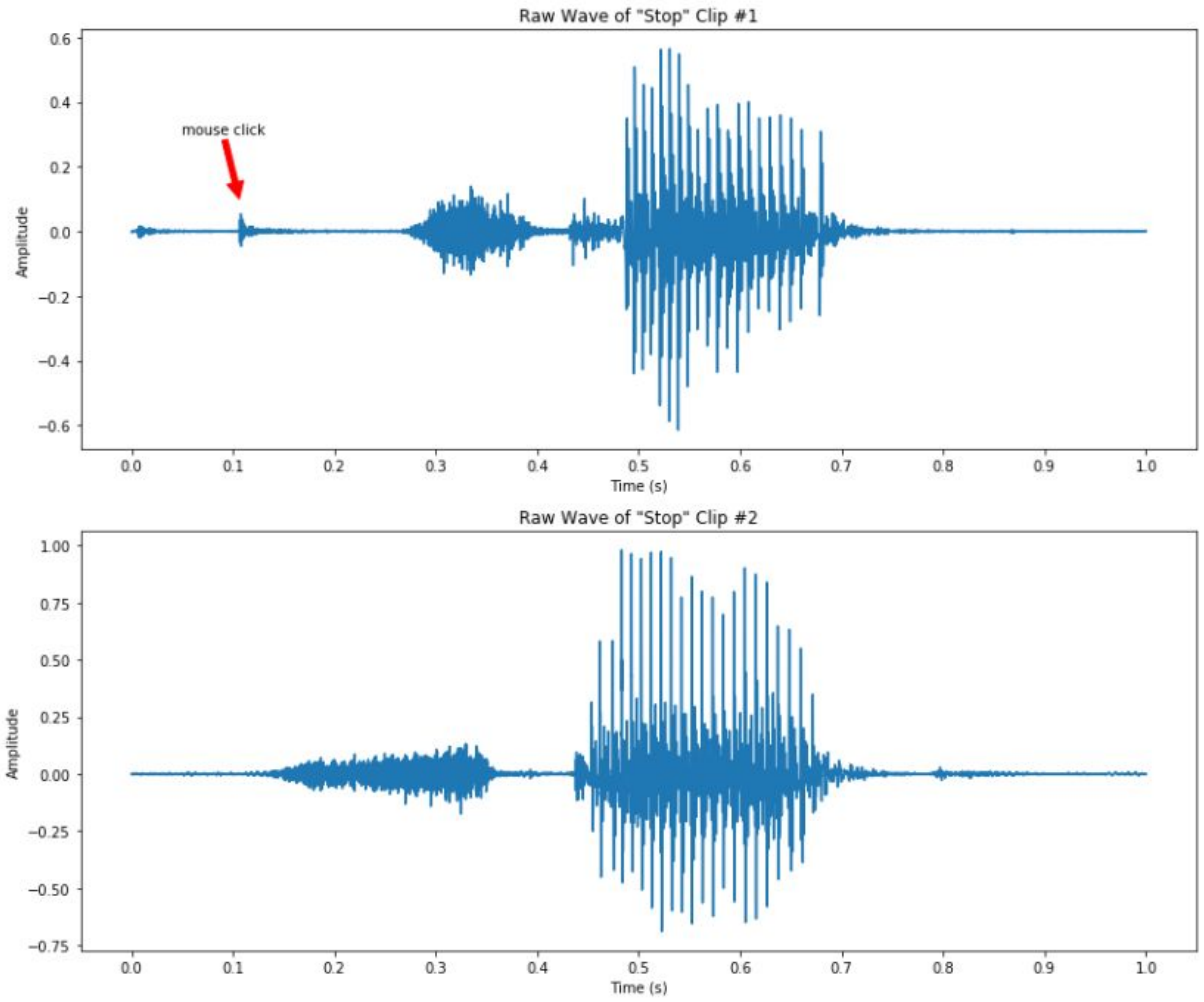
treated as images, and thus will be used as inputs for a convolutional neural network. I will use TensorFlow with Keras to build the neural networks.

To improve model performance and better simulate speech in noisy environments, I will also create new samples from the original voice clips mixed with some background noise. This will add to the size of the input data, and will hopefully produce better accuracy as a result.

Data Cleaning and EDA

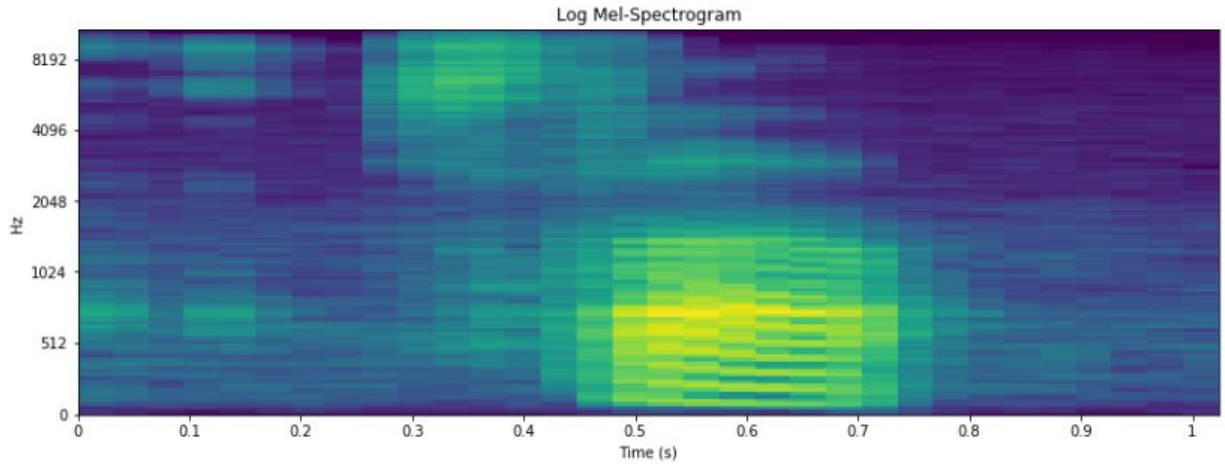
The raw speech data was gathered from the TensorFlow website. The data consists of 35 words, each with around around 500 samples. Each sample is approximately 1 second long and was sampled at a sample rate of 16,000. The data was originally in WAV files, so I needed to do some preprocessing before they could be fed into the end models. Additionally, since the clips are not exactly the same length, I had to pad or delete the ends of all the clips so that they are exactly 1 second long. Once the samples were ready and separated into training, validation, and testing sets, I could begin to process the audio data.

There are a few ways speech data can be processed. The first and the simplest is by extracting the WAV files into the basic wave form frequencies. One sample is then a one-dimensional vector with each data point indicating an amplitude at a certain point in time. This method is visualized below for two sample for the word “stop.”



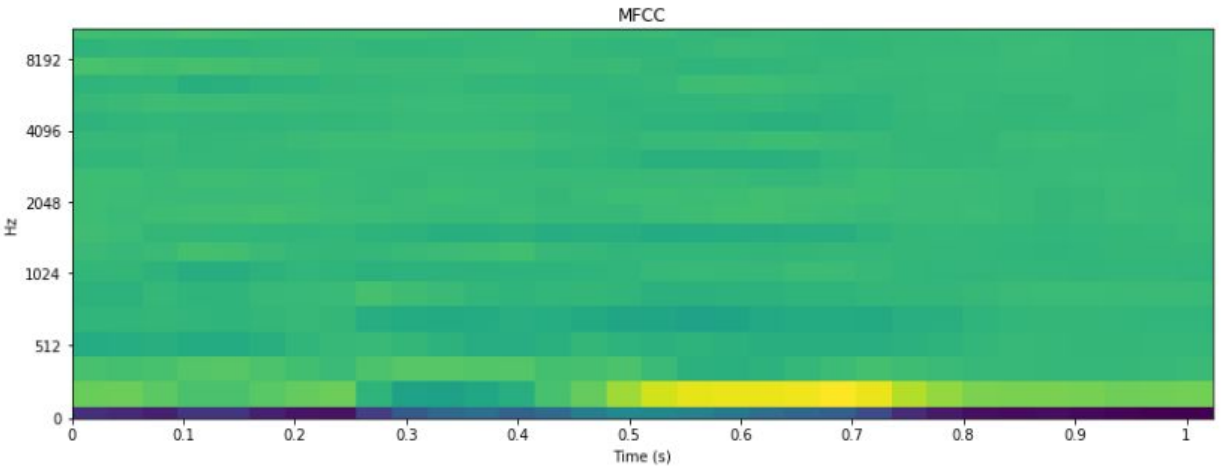
It is important to note that the amplitude is unitless when extracted by librosa - the tool used to process the audio samples. This method is the simplest, but will not produce as good results as other more sophisticated methods.

The second method is to extract the samples into their log-spectrograms. This adds a dimension of complexity to the raw wave forms - it is a 2 dimensional array of amplitudes at different frequencies at different points of time. This method is more complex, but will produce much better results than the raw wave data. This method is visualized below for the first sample of the word “stop”:



The more yellow colors represent a louder noise at that frequency at that point in time. Already we can see the benefit of the added dimension - this spectrogram shows that the “s” in “stop” is in the higher frequencies while the rest of the word is in the lower frequencies. This added information would be lost in the raw wave form data, so with this method we should get better results.

The final method of data extraction used is the Mel Frequency Cepstral Coefficient (MFCC). MFCCs are a feature widely used in speech recognition, and are designed to extract information about linguistic content from audio while ignoring background noise. These are used in most speech recognition systems today, and so we will use these for our final model. One sample of the same word “stop” using this technique is visualized below:



Although we appear to lose some information, these features will probably work the best when it comes to classification with neural networks, while also using less data than the raw spectrograms.

In addition to the speech samples, there were 6 samples of background noises provided to use as “silence” data. These were around a minute long instead of the second long speech samples. To use these as data for the neural networks, I extracted at random 1 second clips from each of these samples, and then multiplied the raw wave data by a random number between 0 and 1 to simulate different noise levels. This was done until I had an equal number of silence samples as the other words. These were then converted to the MFCC form like the speech samples to use in the neural networks.

Preliminary Models

After performing the data extraction and cleaning, the next step was to build a neural network to feed it into. I started by creating a simple single layer perceptron model that would only classify the ten command words with silence as a proof of concept. I also added a batch normalization layer at the start so that the optimizer wouldn’t get stuck as easily. This classified words with around 56% accuracy.

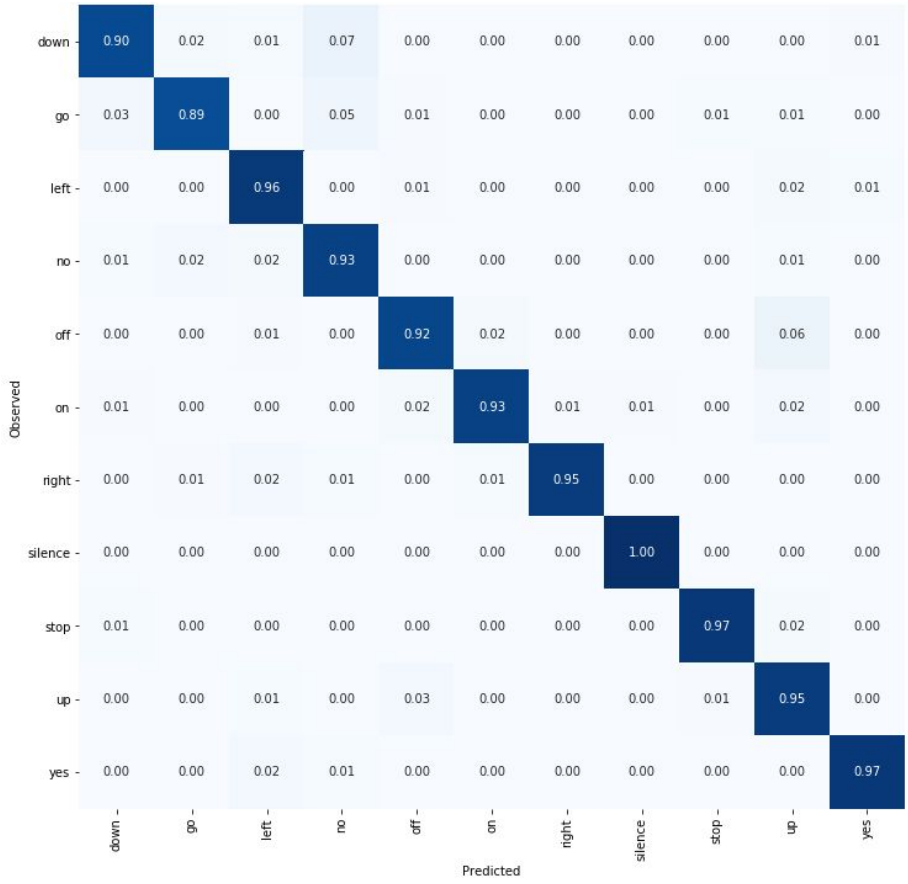
After showing that this project is viable, the next step was to raise the model by a layer of complexity by creating a feedforward neural network. It followed a very similar architecture to the single layer perceptron, with 3 hidden layers added. Already with only these 3 hidden layers, the accuracy was raised to around 81% after training for only 10 epochs.

Finally, I created a convolutional neural network. Convolutional neural networks tend to work best for image processing, and since we are treating our speech data as images, using

one gave the best results yet. This time, the architecture was a little more sophisticated. It was as follows:

```
model = Sequential()
model.add(InputLayer(input_shape=(X_train_img[0].shape)))
model.add(BatchNormalization())
model.add(Conv2D(128, kernel_size=(2, 2), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(256, kernel_size=(2, 2), activation='relu'))
model.add(Conv2D(512, kernel_size=(2, 2), activation='relu'))
model.add(Dropout(0.2))
model.add(GlobalMaxPooling2D())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(11, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(X_train_img, y_train_hot, batch_size=128, epochs=30, verbose=1,
          validation_data=(X_val_img, y_val_hot), callbacks=[EarlyStopping(patience=3)])
```

I used many of the defaults in terms of optimizers and activation functions, but now the network is much larger and we've added a couple dropout layers as well. This model gave us an accuracy of about 94%! Let's look at the confusion matrix:

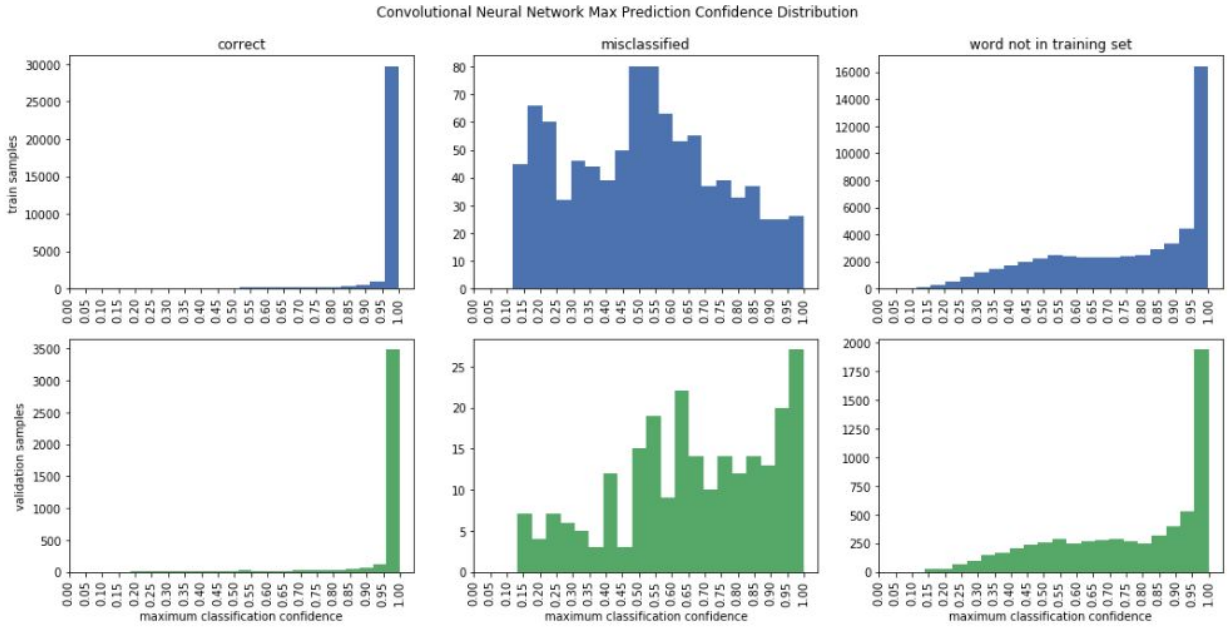


This time it performs very well across the board. It still gets a couple of words confused (e.g. off and up), but overall performs very well.

Adding Unknown Detection

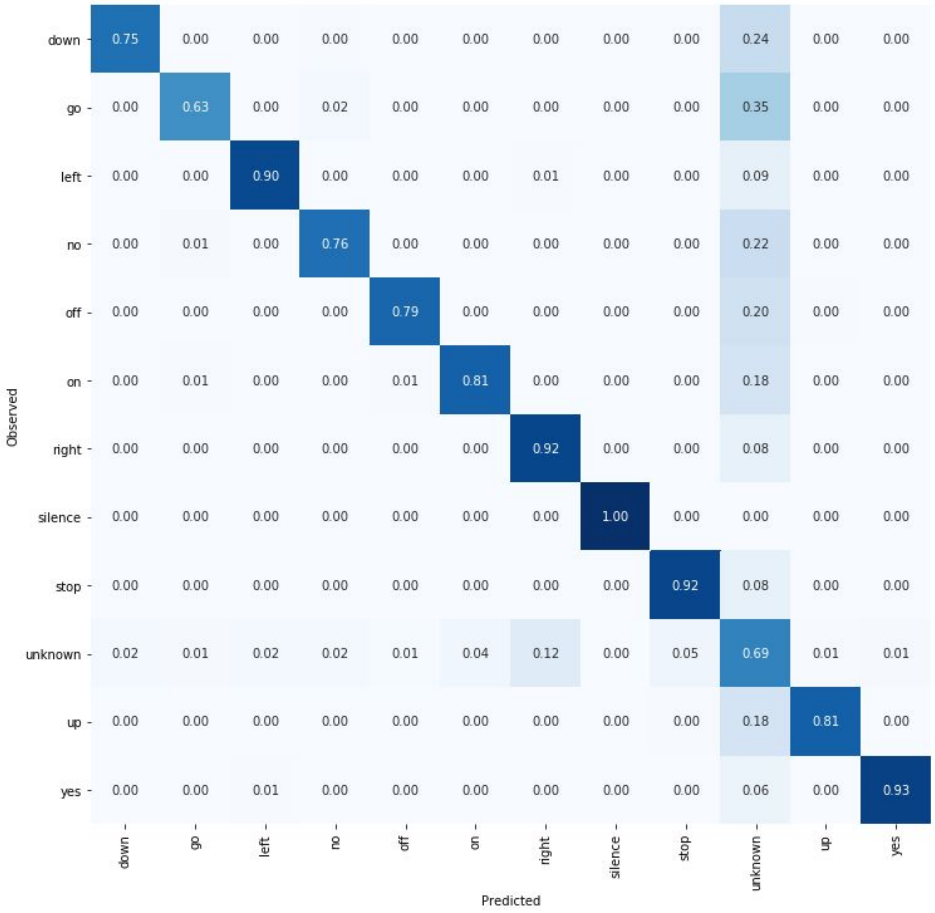
One last feature the models have been missing so far is to classify words not in the command set as "unknown." One way to accomplish this is to train it on the 25 words not in the command words set, and label them as unknown. This would work with the data we're given, but we want it to be able to predict any given word or noise it doesn't understand as unknown. I performed this by training on the command words and validating and testing on the entire sample set, classifying any words under a certain confidence threshold as "unknown." I then can tune this confidence threshold to maximize accuracy, or maybe minimize false positive rate depending on which metric is considered more important.

I started by testing on the entire dataset and seeing what distribution of confidences our model is giving for words it classifies correctly, those in the command words set it misclassifies, and unknown words.



The convolutional neural network tends to be very confident in its predictions for correctly classified words. For words it knows that it misclassifies, the model can be anywhere from not confident at all to very confident. For words outside the command words set, the model tends to predict with very high confidence most of the time, but can also predict with low and medium confidence fairly frequently as well. Again, this could be due to the words sounding familiar to ones it does know (e.g. "nine" and "no").

Based on the histograms above, I decided to set the threshold for predicting as unknown right around .95. This will eliminate most of the misclassified words it should know (the kind of error we want to avoid the most) without removing too many of the correctly classified words. Testing again, we get an overall accuracy of ~75% and the following confusion matrix:



Although we only have an accuracy of around 75%, the accuracy for most individual word is considerably higher than that. This is because of the large class disparity - over half of the total words are 'unknown'. These words are bringing down the overall accuracy. Still, we have very few other false positives. For command words, the model tends to predict them correctly or as unknown when it's not confident enough. This is exactly how we want it to behave.

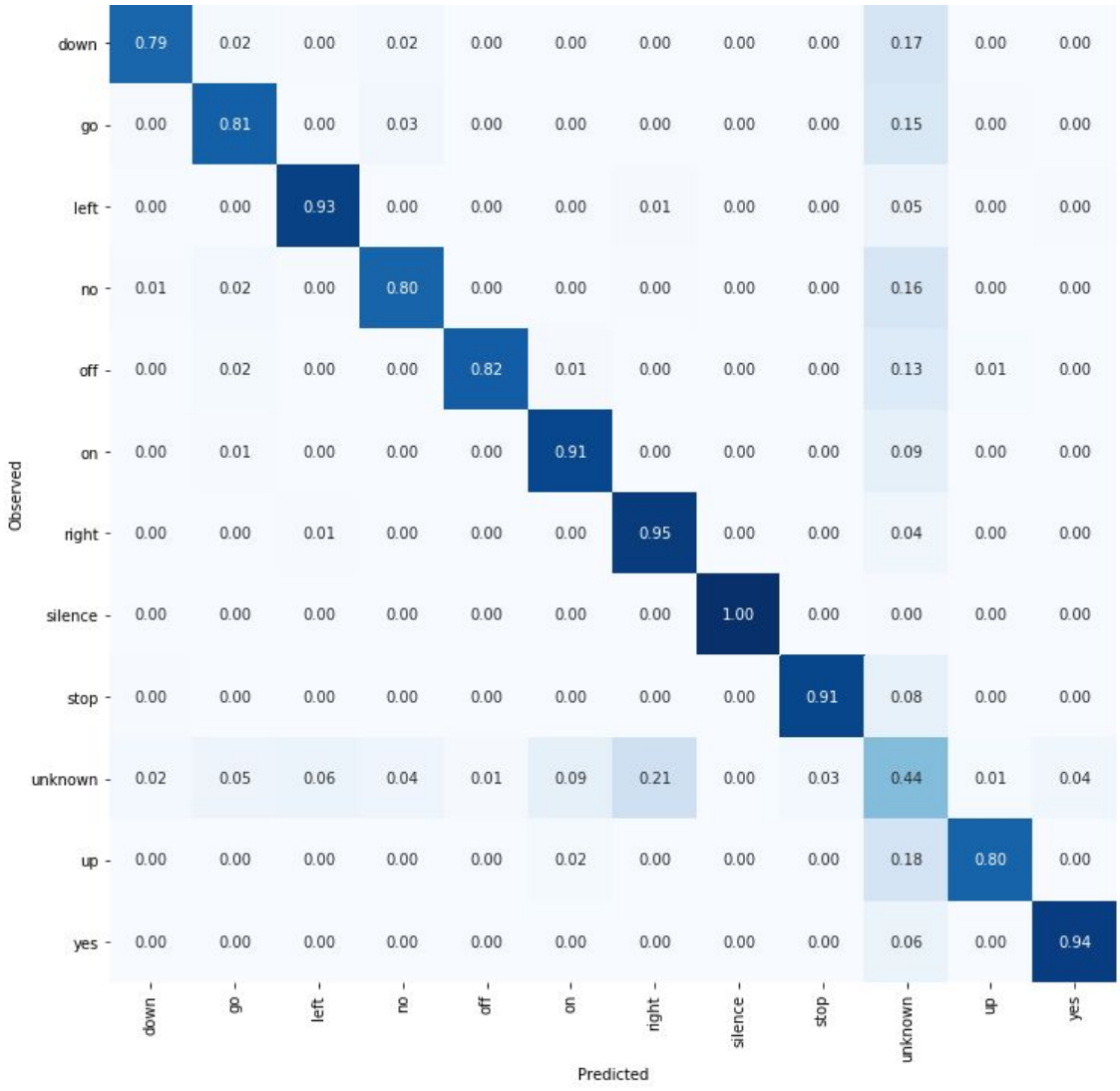
This model works well enough that we could stop here. Still, there are a few things left to do to optimize performance.

Fine Tuning Model Performance

Now that we have the foundation for a successful model, we can focus on fighting for each additional percent in accuracy. The first step was to add augmented data for the words the model tends to struggle with in particular. I performed this by randomly sampling clips of these words, adding various levels of background noise to the data, and adding these to the training set. This added some additional training data as well as some bias towards these words.

Finally, I performed a grid search over the best hyperparameters to use. To save training time, I performed a 3-fold cross validation over only a small subset (~20%) of the total training data, and trained each combination for 20 epochs. I began by grid searching over the optimizer, learning rate, and batch size simultaneously. For the optimizers, I included SGD, RMSprop, Adagrad, Adadelta, and Adam. I also searched over learning rates of 0.001, 0.01, 0.1, 0.2, and 0.3, with batch sizes of 16, 32, 64, 128, and 256. I ended up getting the best results with the RMSprop optimizer with a learning rate of 0.001 and a batch size of 32. Using these hyperparameters, I then searched over various dropout percentages - a scale from 0 to 1 in increasing 0.1 intervals for each dropout layer in the network.

After obtaining the best hyperparameters from the grid search, I used them to train on the entire training set, including the new augmented data. I obtained an accuracy of 62% with the following confusion matrix:



Overall, the accuracy actually went down slightly after the grid search. This is likely due to only searching over a random subset of the training data. To find the true optimal hyperparameters, I will likely have to search over the entire data set. Still, with the default values we have accomplished the initial goal of creating a very successful simple speech recognition system.

Conclusion and Next Steps

In this project, I explored the TensorFlow Speech Commands Datasets and used them to create a simple speech recognition system. I found that a convolutional neural network approach worked best, and was able to classify words with a 80-90% accuracy with very few false positives.

Some steps to be taken to improve these results is to implement a grid search over the entire training set, add more augmented training data, and to implement a transfer learning approach from other speech recognition systems. Another idea is to implement the model in ta program that could classify real-time streaming audio. Still, this model would be accurate enough to implement as a simple speech recognition system for basic use cases, or could be used as the foundation for more sophisticated needs.