Kieran O’Gara

Peter Cosgrove

AI 686

Final Project

12/06/2024

Fall 2024

### Advanced Speech Command Recognition System

This paper reflects the design and implementation of a speech command recognition system that has the ability to interpret in real time on an edge device like Raspberry Pi. The basic fundamentals of the system utilizes a hybrid Convolution Neural Network (CNN) architecture to have the ability to process the dataset with over 100,000 audio files. This dataset was from the Google Speech Commands. The model attempted to remain efficient, while still robust in its ability to decipher commands even with background noise present. TensorFlow is used for the conversion and deployment of the architecture with Raspberry Pi.

As a whole, speech recognition has taken a major step forward in human’s daily lives over the last 15 years or so. It can be quite useful in many scenarios and tasks. Utilizing these methods in the context of driving creates a much safer environment for drivers, passengers and pedestrians while on or near the road. Speech recognition and command could also be applied to the Internet of Things (IoT) which includes smart homes or smart devices within homes, like a smart fridge or thermostat. Deploying speech recognition systems on an edge device does have certain challenges to consider. There are usually computational constraints to consider. Edge devices like Raspberry Pi have limited processing power and memory which makes using deeper learning models slightly more complicated. In the case of the IoT, consumer price point must also be considered. A normal fridge still keeps food cold, so the utility of a smart fridge may be overbearing to many consumers, and not worth the extra investment. This relates back to computational costs as well. Environment noise also has to be considered as a main challenge. Background noise can play a huge role in the ability to decipher the command speech properly. Also to be considered are the real time requirements necessary. Albright slightly related, new Google phones have the ability to real-time translate languages, even while disconnected from the internet or wifi. This means that it needs to be preloaded with the data and ability to translate. In our case, we must be linked back to our code with the Raspberry Pi. The Raspberry Pi is not nearly as capable as a Google phone in this regard.

To dive further into the dataset we used for this project, we must first comprehend the size. The Google Speech Commands Dataset contains over 100,000 audio samples. Each of these samples are roughly 1 second in length. The words used for this include “yes”, “no”, “left”, “right”, “on”, “off”, “go”, “stop”, “up” and “down”. Some audio samples had an ideal environment with no background noise, while others had more of a real-world condition with a noisy background. The feature engineering began with resampling each audio sample. The samples were initially 16 kHz but were then converted to Mel-frequency cepstral coefficients (MFCCs). The extraction of the MFCCs involved dividing the audio into frames that overlapped, calculating the power spectrum by applying a Fourier transformation, mapping the spectrum onto a Mel scale and finally computing the logarithm of the Mel spectrum. A discrete cosine transform was then applied and the MFCC features were reduced to 40 coefficients and scaled to zero mean for the purpose of normalization. Data augmentation was utilized to improve overall robustness in an effort to combat background noise. White Gaussian noise techniques were used to help simulate electronic or background interference. This was done by adding random noise to the background on samples. The pitch of the audio files was shifted to account for any real-world variable, like the tone of different speaker systems. The time of the audio files was stretched to be able to simulate slow or fast speech. There was also noise that was mixed in the background like office, street or crowd noises to help combat real-world scenarios. As a whole this helped broaden the range of our original dataset to anticipate future problems during a use case scenario.

The architecture was built on a lightweight CNN to be able to efficiently process the MFCCs. The input layer first accepted MFCC features with a shape of (40, 1) for each 1-second audio sample. THe single channel input allows for the simulation of a grayscale image processing for the computer vision aspect of the task. The first Convolutional layer had 16 filters with a kernel size of 3. It used the ReLu activation function and achieved the extraction of higher-level patterns like the transition between phonemes. Each Convolutional layer was followed by a max-pooling layer with a pool size of 2. This allows for the reduction of dimensionality while still retaining key features which optimize the overall computational efficiency. The fully connected layer had 64 neurons and a ReLU activation layer to be able to map extracted features in a higher-dimensional representation. This layer had a 0.2 dropout to reduce overfitting of the model. The output layer had 10 neurons, representing each command class. The softmax activation function was used to be able to convert feature representations into class probabilities. In order to train this, the Adam optimizer was used with a learning rate of 0.001 to be able to learn in an adaptive manner. The loss function allowed for a multi-class classification in a cross-entropy manner. The efficiency of the parameters made sure to keep in mind that the end goal edge device for the project has limited computational power and cannot be overwhelmed. In order to optimally achieve our goals with TensorFlow, we utilized the lite version to reduce size and computational complexity. We converted the 32 bit floating point weights to 8-bit integers to ensure our quantization technique was to par with everything else. This allowed us to use a dynamic range quantization technique to remain flexible in training. The model size was reduced to 2.3 MB without sacrificing a notable level of accuracy. In order to understand the real-time testing we had to first understand the latency. The average inference time was 20 ms. This means that the system was able to quickly respond to voice commands. The overall resource utilization was about 25-30% of the CPU. The memory footprint was about 50 MB during the run time.

The overall results of the training across 20 epochs with the Adam optimizer were successful. The training accuracy was 97.6% while the validation accuracy was 94.3%. The loss decreased consistently across epochs, meaning that the learning was effective over time. The noise robustness model had an accuracy of 87.2% when tested with noisy audio samples. Depending on the noise type, the accuracy varied. With white noise the accuracy was 85.6%, with crowd noise it was 89.3% and with street noise it was 84.7%. As a whole the results were satisfactory. The model was efficient and robust in this real-world, yet contained environment. The command vocabulary was limited, and the background noise was not extreme at any point in time. Future works to be considered could expand on this vocabulary or work to block out extreme noise situations. We could also look to use a more computationally powerful edge device to further expand real-world use cases.

References

Warden, P. (2018). "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." *arXiv preprint arXiv:1804.03209*.

TensorFlow. (2023). TensorFlow Lite Documentation. Retrieved from https://www.tensorflow.org/lite.

Librosa. (2023). Librosa Audio Processing Documentation. Retrieved from<https://librosa.org>.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.