Implementation of a Keyword Spotter using an Embedded Board for Efficient Wake Word Detection

By: David Nichols and John Smith

***Abstract*—This paper presents the implementation of a keyword spotter using an embedded board for efficient wake word detection. Wake word detection is an essential component of many speech recognition applications and involves detecting a specific phrase or word that triggers the system to start processing audio input. In this project, we explore the use of an embedded board, specifically the Nano 33 BLE Arduino Board, to develop a low-cost and efficient solution for wake word detection. We first train a deep learning model on a dataset of audio samples to recognize the target wake words; yes and computer. The model is then deployed on the NANO BLE, where audio input is then continuously monitored, and the wake words are detected in real-time. We evaluate the performance of our system using various metrics, including accuracy, speed, and resource utilization. Our results demonstrate that the proposed solution achieves high accuracy while utilizing minimal resources, making it suitable for deployment in resource-constrained environments. Overall, this paper presents a practical and effective approach to implementing wake word detection using an embedded board, with potential applications in smart home devices, voice assistants, and other speech recognition systems.**

INTRODUCTION

Voice assistants and speech recognition are becoming increasingly common in our every day lives. From Siri and Google on our phones, to the ever expanding world of smart devices connected in our homes, wake words are the foundation of these technologies. In this project, the team intend to train a model to recognize a wake or ”trigger” words: ”yes computer.”

In the following sections, this paper will cover the teams model architecture to include the topology of the teams neural network, input features and experimentation. Next, we will discuss our data and training including data collection and augmentation. Next the team will provide results, discussion of those results, to include the viability of this model in a deployed setting. We will then conclude with lessons learned.

MODEL ARCHITECTURE

When choosing the model architecture, two features were considered: parameters and runtime. The model, to be considered successful, needed to have an inference time of 150ms or less; closer to 100 if possible. Additionally, the number of parameters plays a crucial role in the accuracy and inference time. Too little and the model becomes widely inaccurate, but, too high, the models inference time makes it almost unusable. Therefore, choosing the correct model became somewhat of a balancing act.

First, the team tried to train using a CNN model. Convolutional Recurrent Neural Networks (CRNNs) have been widely used in speech models due to their ability to capture both temporal and spectral features of speech signals. CRNNs combine the advantages of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) by using CNNs to extract local spectral features and RNNs to model the temporal dynamics of speech signals. This allows the model to learn complex relationships between speech signals and their corresponding labels, leading to improved accuracy and robustness in speech recognition tasks. This failed due to the Convolutional Recurrent Neural Network(CRNN) having bidirectional layers, and therefore we could not get it to flash to the board.

Next, an LCNN, or Lightweight Convolutional Neural Network, are a type of neural network that is designed to be computationally efficient while still achieving high accuracy in classification tasks. In speech recognition, LCNNs can be used to extract spectral features from audio signals, which can be used to classify speech sounds and transcribe spoken words. Due to their efficiency and accuracy, LCNNs have become increasingly popular in speech recognition applications, particularly in edge computing environments where computational resources are limited.

The LCNN model did not work. The input shape of this model was 32x32x1, increasing the parameter size of the model significantly. This increase in model size made flashing to the board difficult due to the lack of memory contained on the NANO specifically. Additionally, using models that have so many parameters significantly reduces the usability of a real time application. Therefore, another model was sought out.

Next, the TinySpeech model was explored. This model showed promise, but in implimentation, the team could not get the allOpsResolver to work. After many failed attempts and no viable solutions found online, the

team reverted back to the original model provided with the project. This CNN was then slightly modified, adding three DepthwiseConv2D layers, BathNormalization, used an atypical learning rate, and altered the number of filters in each layer. Doing so provided the model shown in Figure 1.

```
Input shape: (24, 32, 1)
Model: "simple_cnn"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 22, 30, 20)        200

 batch_normalization (BatchN  (None, 22, 30, 20)       80
 ormalization)

 activation (Activation)     (None, 22, 30, 20)        0

 pool1 (MaxPooling2D)        (None, 11, 15, 20)        0

 dropout (Dropout)           (None, 11, 15, 20)        0

 depthwise_conv2d (Depthwise  (None, 11, 15, 20)       200
 Conv2D)

 conv2d_1 (Conv2D)           (None, 11, 15, 32)        672

 batch_normalization_1 (Batc  (None, 11, 15, 32)       128
 hNormalization)

 activation_1 (Activation)   (None, 11, 15, 32)        0

 dropout_1 (Dropout)         (None, 11, 15, 32)        0

 depthwise_conv2d_1 (Depthwi  (None, 11, 15, 32)       320
 seConv2D)

 conv2d_2 (Conv2D)           (None, 11, 15, 64)        2112

 batch_normalization_2 (Batc  (None, 11, 15, 64)       256
 hNormalization)

 activation_2 (Activation)   (None, 11, 15, 64)        0

 dropout_2 (Dropout)         (None, 11, 15, 64)        0

 depthwise_conv2d_2 (Depthwi  (None, 11, 15, 64)       640
 seConv2D)

 conv2d_3 (Conv2D)           (None, 11, 15, 64)        4160

 batch_normalization_3 (Batc  (None, 11, 15, 64)       256
 hNormalization)

 activation_3 (Activation)   (None, 11, 15, 64)        0

 dropout_3 (Dropout)         (None, 11, 15, 64)        0

 max_pooling2d (MaxPooling2D  (None, 2, 3, 64)         0
 )

 dropout_4 (Dropout)         (None, 2, 3, 64)          0

 flatten (Flatten)           (None, 384)               0

 dense (Dense)               (None, 128)               49280

 batch_normalization_4 (Batc  (None, 128)              512
 hNormalization)

 activation_4 (Activation)   (None, 128)               0

 dropout_5 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 128)               16512

 batch_normalization_5 (Batc  (None, 128)              512
 hNormalization)

 activation_5 (Activation)   (None, 128)               0

 dropout_6 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 4)                 516

=================================================================
Total params: 76,356
Trainable params: 75,484
Non-trainable params: 872
_________________________________________________________________
```

Fig. 1: Model Architecture

Data and Training

To collect our data, the team used a M1 Macbook Pro to produce the majority of the teams audio recordings.

To collect our data, the provided link to the ”get and augment audio” notebook was used. This notebook allowed the team to record and visualize the audio, replay the record and then save if the sample was good. Each team member recorded and saved seven sample of the keyword ”computer” for a total of 14 original samples. Next, custom noise was recorded and added to the custom audio at increasing amplitudes. Lastly, the audio was subjected to time shift and pitch shift, creating new samples with each augmentation. After all augmentations were complete, the teams 14 original audio samples numbered 4,044 total samples. This is shown in the below Figure 2. This number of samples, derived from the amount of augmentations performed was chosen to match the number of ”yes” samples from the Google dataset that will be used later.

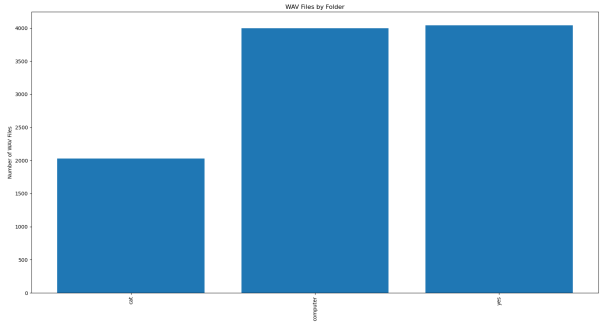

Fig. 2: Dataset

After all preprocessing and augmentation were complete, an example of the finished sample spectorgram is shown in Figure 3 bellow.

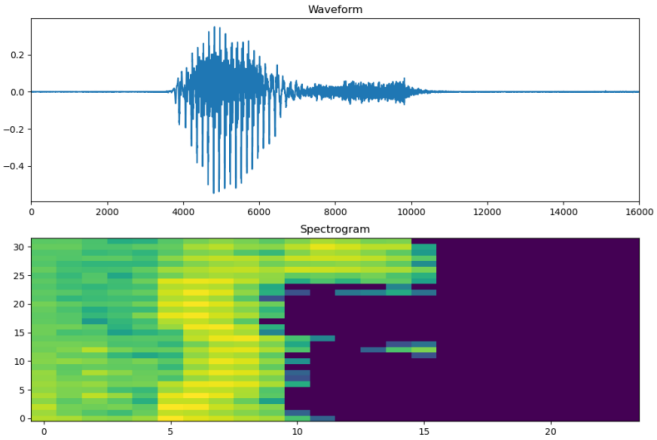

Fig. 3: Waveform and Spectrogram

From the above image, we can clear, distinct start and stop points of the word with minimual ambient noise. Samples with clear, defined boundries work better as we start to train.

RESULTS

After the model was trained, its accuracy was measured via jupyter notebook and found to be 92%. This was determined by comparing the predicted and true values and dividing by the length of the true values. The validation data showed an accuracy of 87%, while the test data had an accuracy of 90%. As seen in the below figure 4

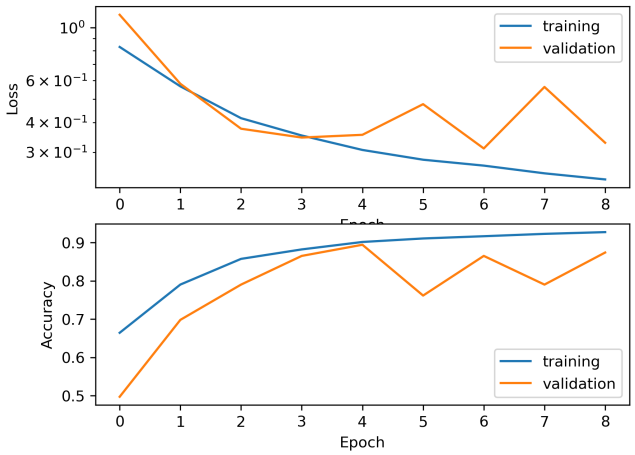

Fig. 4: Loss and Accuracy

In software, the model did very well predicting the target words, it did have issues predicting silence, specifically yes was categorized as silent fairly often. As seen in the below Figure 5.

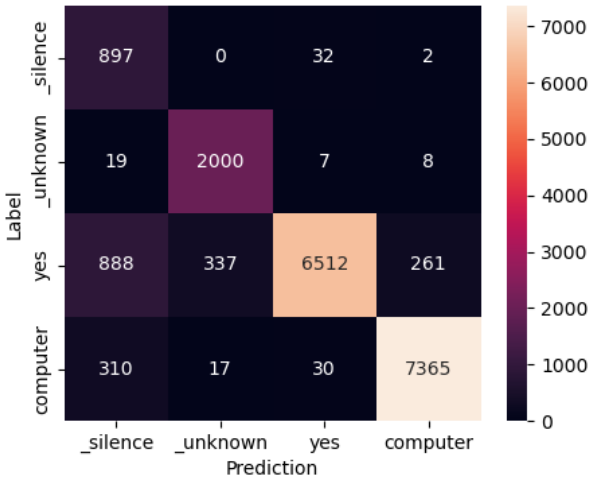

Fig. 5: Data set conf matrix

The model is able to respond to both target words, but it occasionally generates false positives, such as on the word ”commuter”. It also produces false alarms on the word ”computer” when there are loud noises. Furthermore, the model tends to fail more often on female or higher pitched voices, with a response rate of 1 out of 6 for ”yes” and 1 out of 10 for ”computer”.

During testing, the model was found to take 150ms to run and had a sample rate of 16khz. The test used a noise source with sound levels ranging from 70 to 80dB, which resulted in a false alarm rate of 180 per hour for the label ”computer”. However, the label ”yes” did not trigger any false alarms. If you’re interested, you can listen to the noise source used in the test at the following link: Microspeech Detection.

SUMMARY TABLE

Below is a table of our calculated measurements and metrics; shown in Table I.

TABLE I: Project II Results Summary

Metric	Result
Accuracy in TensorFlow: Training Data	92%
Accuracy in TensorFlow: Validation Data	87%
Accuracy in TensorFlow: Test Data	90%
FRR in TensorFlow for Speech Commands Dataset	19%
FRR in TensorFlow for Custom Word	5%
Number of Parameters	76,356
MACs	152,712
Input Tensor Shape	24x32x1
Sampling Rate	16kHz
False Alarm Rate	180 per hour

DISCUSSION

The model is generally usable, but it struggles to detect the word ”computer” unless the user speaks loudly. To improve performance, the model needs to be more responsive to quiet voices for this word and generate fewer false alarms when there are loud noises. Initially, only the label ”unknown” was detected before lowering the detection threshold to 100.

One possible improvement that was explored but not included in the final version of the program was adding explicit biases to the model. This involved adding a weight of 50 to the score for ”computer” and using different weights for ”yes” and ”silence” to improve detection. This helped the model pick up ”yes” and ”computer” more often and enabled it to detect ”silence” even if there was some noise in the room.

Overfitting was a challenging issue to overcome, and getting a working model onto the board took some effort. The team encountered setbacks during the development process, including an issue generating spectrograms that caused the

model to stop training a day before the original due date. As a result, the team had to delete everything and start over. Lack of diverse silent data also contributed to the model's difficulty predicting silence. In hindsight, the team would have recorded more audio samples of the target words and silence and selected a target word that was less likely to be confused with other words.

CONCLUSION

These results highlight some important lessons learned for the working speech detection models. First, while the overall accuracy of the model was high, it still had limitations in terms of generating false positives and responding to higher pitched voices. This suggests that there is a need for more diverse training data to improve the model's performance on these fronts. Additionally, the results showed that the model's performance was impacted by loud noises, indicating that the model may benefit from additional noise reduction techniques or more robust feature extraction methods. Finally, the results demonstrated the importance of thorough testing and evaluation, as the model's false alarm rate was found to be relatively high in certain situations. This highlights the need for ongoing monitoring and refinement of the model to ensure its reliability in real-world applications.

In conclusion, the team has successfully implemented a low-cost and efficient keyword spotter using an embedded board, specifically the Nano 33 BLE Arduino Board, for wake word detection. The team trained a CNN model on a dataset of audio samples to recognize the target wake words ”yes” and ”computer”. The model achieved high accuracy while utilizing minimal resources, making it suitable for deployment in resource-constrained environments. The team encountered challenges during the development process, including overfitting, lack of diverse silent data, and difficulty detecting the word ”computer” unless spoken loudly. However, the team was able to address some of these challenges through experimentation and exploration. Ultimately, the team has learned the importance of data augmentation and the challenges of deploying models in real-world applications.

APPENDIX A
GITHUB REPOSITORY

https://github.com/holleman-classes/project-2-team-1.git