Audio Wake Word detection on an ESP32 microcontroller

Adam Kocsis
BU
adamkam@bu.edu

Joseph Aoun
BU
joseph04@bu.edu

Abstract

This project aims to implement a audio wake word detection system on an ESP32 microcontroller. We will train and optimize a model using TensorFlow Lite Micro and deploy it using the Espressif IDF (IoT Development Framework). This approach will enable low-power, always-on voice activation for various IoT applications.

1 Background

Tiny Machine Learning (TinyML) enables the deployment of machine learning models on microcontrollers and edge devices with limited computational resources. By optimizing algorithms through techniques like quantization and pruning, TinyML facilitates real-time inference while maintaining low power consumption—crucial for Internet of Things (IoT) applications.

One key application of TinyML is wake word detection, which allows hands-free voice activation of devices. Traditional systems often rely on cloud processing, introducing latency and privacy concerns. This project focuses on implementing an audio wake word detection system on the ESP32 microcontroller, leveraging TensorFlow Lite Micro to create an efficient, locally processed solution. Our work aims to enhance user interaction while preserving privacy and minimizing energy usage.

2 Goals

Our goal is to create a system that can recognize a specific 1 second long words or phrases spoken by a user, triggering a response or action on a small, low-power device (ESP32 wrover). Imagine a smart home device that springs to life when you say "Hey Home," ready to assist you with various tasks. Although, certain wake words could be used to trigger more diverse hardware outputs such as displaying an image on an OLED screen or begin connection to Wifi. Additionally we would like compare different architectures and optimization methods.

3 Input / Output and User Interaction

Input:

- 1 second long audio from INMP441 Microphone with
- 40ms window frame
- 20ms window stride
- 16KHz sample rate
- 16-bit signed PCM data
- single channel (mono)

Output:
User defined action for up to 4 different wake words ("yes", "no", "silence", and "unknown"). In practice 2 wake word classes would be relegated to "noise" and "silence". Device enters an active listening state for further commands or only listens after some user input such as a button press.

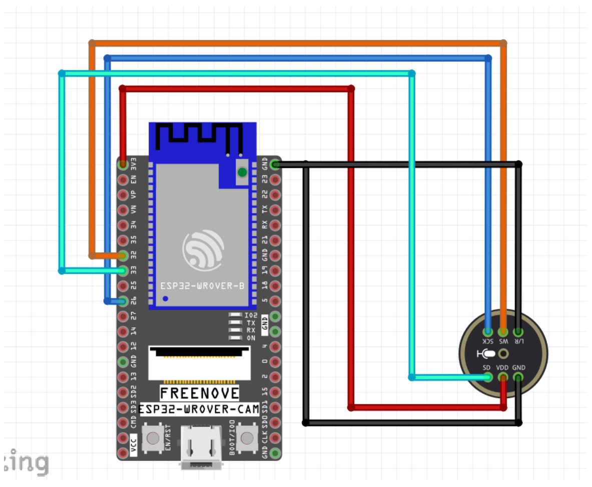

Figure 1: Basic Schematic

4 Project Impact

If successful, this project will be helpful to people in several ways:

- **Accessibility:** It will enable hands-free activation of devices, benefiting those with mobility impairments or when users' hands are occupied.
- **Energy Efficiency:** By using a low-power microcontroller, we can create always-on devices that consume minimal energy, reducing electricity costs and environmental impact.
- **Privacy:** Processing wake words locally on the device, rather than in the cloud, enhances user privacy by minimizing data transmission.
- **Cost-Effective Smart Devices:** The ESP32's affordability could lead to more accessible smart home and IoT devices for a broader range of consumers. This is why in practice devices such as Google home only cost around 15 dollars, since the device hardware itself is just a microphone and a microcontroller with WiFi capability.

5 Methodology

- **Data Collection:** Gather a dataset of wake word utterances and background noise, either from online sources or by creating our own with the necessarily data augmentation pipeline.
- **Model Design**: Develop a lightweight neural network architecture suitable for microcontrollers using tensorflow lite micro.
- **Training:** Use TensorFlow to train models on our dataset and compare different hyperparamters values with output metrics.
- **Optimization:** Employ TensorFlow Lite Micro to optimize the model for the ESP32's limited resources using pruning, quantization, compression, etc.
- **Deployment:** Utilize the Espressif IDF to deploy the optimized model onto the ESP32 using the Espressif wake word and neural network libraries.
- **Evaluation:** Assess the system's accuracy, power consumption, and latency.

6 Data

The data we used to train our models is the Tensorflow mini speech command dataset which itself is a subset of the speech command dataset. We extracted a subset of 4 classes each 1 second long, 16kHz single channel audio. The classes are "yes", "no", "silence" and "unknown". Each class has 1000 samples except "unknown" which has 4000 samples comprised of non-target words: "no", and "yes". Thus we end up with an initial dataset of size 9000. We can then run a subset of the training data excluding "silence" samples through an augmentation function that can apply time stretching, pitch shifting, volume adjustment, and noise addition. The function can generate anywhere between 1 to max-augmentation(5) amount of random augmentations with a user defined max number of augmentations for each type. For example, an augmentation can look like: pitch-shift, noise-addition, and pitch shifting. As previously mentioned a user can define the max number of times an augmentation can be used per augmentation cycle in an effort to avoid cases where we have a large number with the same augmentation. The augmented dataset is then appended to the original dataset currently resulting in a dataset size of 17,000. We then shuffle, batch, and split the dataset into train(80%), test(10%), validation(10%) datasets. Data augmentation helps our model by providing more diverse training data. It also helps reduce overfitting and improves its performance in real-world settings, since a voice will likely contain background noise and impurities.

7 Audio Preprocessing Brief overview

Our preprocessing pipeline is designed to convert raw, one-second audio clips into mel-spectrogram features suitable for training a wake word detection model. The steps ensure that each input is standardized, noise-augmented, and properly formatted before reaching the modeling stage. Here is a concise breakdown of our approach: Each input .wav file is read using TensorFlow's native audio API and normalized. The processed audio waveform is then segmented into 40 ms overlapping frames with a stride of 20 ms. After that we apply a window function (in our case the Hann function) to each frame, which tapers the edges and prevents distortions when analyzing its frequencies. We apply a real-valued FFT to each frame to obtain its power spectrum. Next, we map the linear frequency spec-

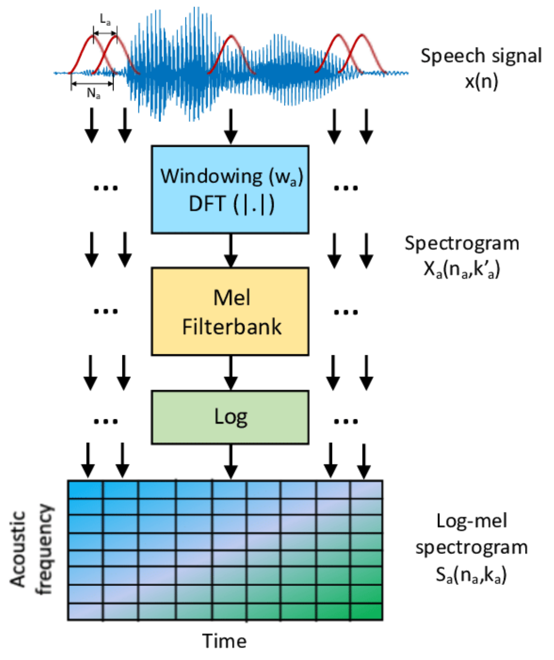

Figure 2: Audio Signal Pipeline

trum to a mel-scale using a mel-filter bank. This non-linear frequency scaling is more aligned with human auditory perception and helps the model learn from frequency patterns more effectively. We then apply a logarithmic transformation resulting in a 2D feature matrix (time frames × mel-frequency bins) that serves as a compact, informative representation of the original audio signal.

8 Overview of Model Optimizations

Quantization is the process of reducing the precision of a model's weights and activations—often from 32-bit floating-point to lower-bit integers like 8-bit—without significantly degrading model accuracy. By using fewer bits to represent numbers, quantization dramatically reduces the memory footprint and computational requirements, making models more suitable for microcontrollers and other resource-constrained devices.

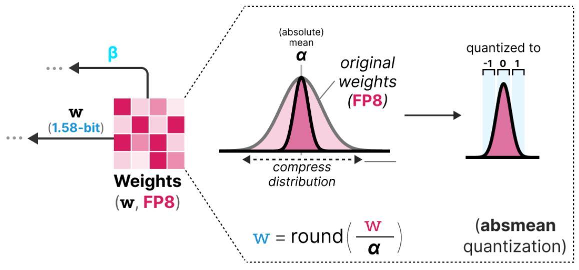

Figure 3: Visual Representation of Quantization

Post-Training Quantization: After a model is trained with standard 32-bit floating-point weights, we convert them to lower-precision representations (e.g., float16, int8). This step is done offline and can provide substantial size reductions with minimal accuracy loss.

Quantization-Aware Training: Instead of quantizing weights after training, the model is trained with simulated low-precision arithmetic in the forward pass. This allows the model to "learn" around the quantization constraints.

Pruning: Removes weights that contribute the least to the model's outputs, creating a sparser network. We used a combination of structured and unstructured pruning. Unlike unstructured pruning which removes individual connections, structured pruning removes complete structures like filters in a convolutional layer or neurons in a fully connected layer.

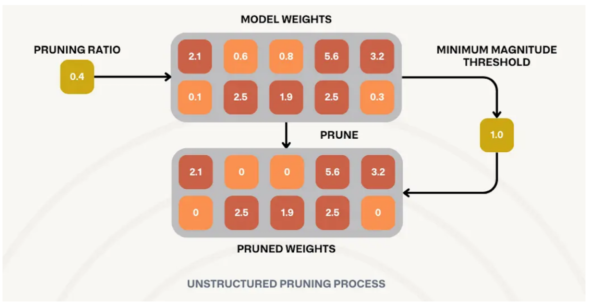

Figure 4: Visual Representation of Pruning

Weight Clustering: Similar weights within a neural network are grouped together into clusters, replacing each individual weight with the corresponding cluster centroid, effectively reducing the model size by significantly decreasing the number of unique weight values needed to store

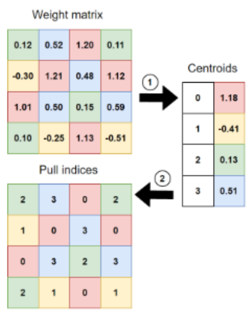

Figure 5: Visual Representation of Clustering

Collaborative Optimization:

While employing any one of the mentioned approaches on a neural network is quite effective and beneficial in isolation, the primary complication that unfortunately emerges is when attempting to sequence these strategies. Using a subsequent one frequently disrupts the outcomes of the earlier method, negating the collective benefits of implementing them all at once. To address this challenge, we adopted an optimization procedure known as Sparsity and cluster preserving quantization aware training, enabling us to integrate Post Training Quantization, pruning, and weight clustering. The result is a quantized deployment model with a reduced number of unique values as well as a significant number of sparse weights, depending on the target sparsity specified at training time. Beyond substantial model compression perks, specialized hardware can leverage these sparse, clustered variants to to achieve substantial reductions in inference latency.

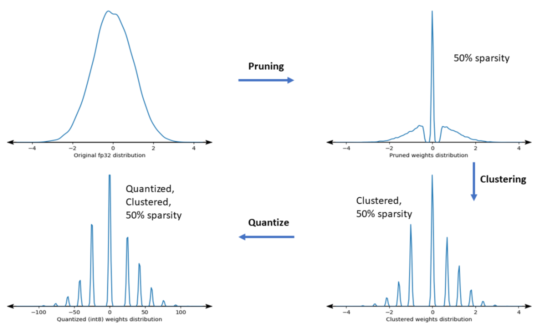

Figure 6: Visual Representation of PCQAT

The process begins by creating and training a baseline model to predict the correct output with good accuracy. Once trained, the model undergoes pruning, where weights with small magnitudes are removed, followed by fine-tuning to recover any accuracy loss. Subsequently, sparsity-preserving clustering is applied to reduce the number of unique weights while ensuring that the sparsity achieved during pruning is maintained. The clustered model is then fine-tuned once more to recover accuracy lost during clustering. Finally, the model undergoes Quantization Aware Training (QAT) and Post Training Quantization (PTQ) to simulate and optimize for low-precision computation, completing the pipeline.

9 Evaluation Metrics

Our evaluation metrics are the same as the regular gamut of metrics for neural network models such as accuracy, loss, F1 score with the additional important metric of model size. It should be noted that in most cases accuracy does not scale linearly with model size; for example, a model with size 50kB achieves 85% while another model of size 100kB may only achieve an accuracy of 87%.

10 Baseline

Our project is based on the tensorflow lite micro example where they themselves based their models on papers 1 and 2. They made several sample models for differing applications but their baseline for a low-resource model is a tiny convolutional model architecture of the form:

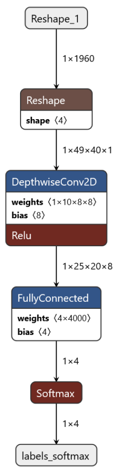

Figure 7: baseline model

After training 4 different networks with architectures based off of the previosuly mentioned papers we were able to achieve an un-quantized score of 83% with smallest model size being 36Kbs.

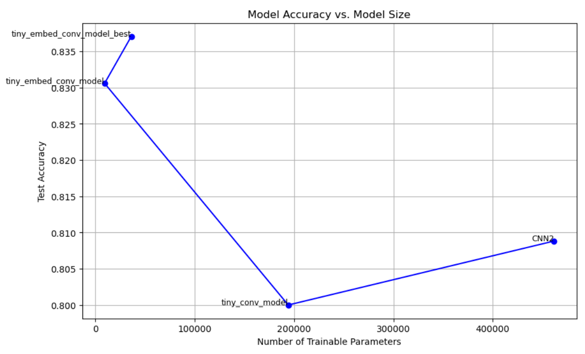

Figure 8: Base model accuracies vs size

11 Baseline Optimizations

The only model optimization that we first did was pre and post training quantization. For our post-training quantization we do float-16 , full integer quantization, dynamic range quantization, and int16 activations with int8 weights. For quantization aware training we quantize to int8 weights and uint8 activations similar to the full-weight post training quantization but now we don't just naively cut the weights down, instead use quantized numbers in the forward pass and only use full precision values when computing the gradient, allowing the model to learn the effects of being quantized thus performing better after being quantized.

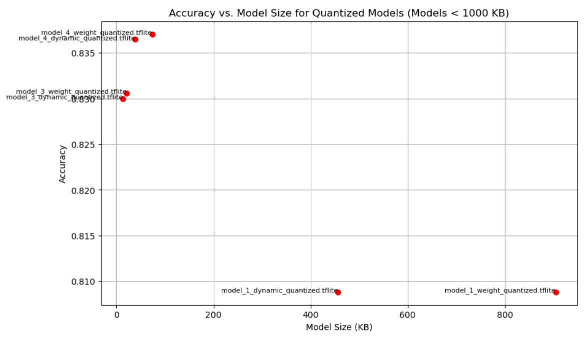

Figure 9: Quantized accuracy vs size

As seen above our quantized models do not suffer from any loss of accuracy from quantization and in the case of the baseline model it actually very slightly improved on the test set since it was slightly over-fitting during training. In the end we achieved an 83% accuracy with a smallest model size of 13.66KBs after applying dynamic range quantization and float16 weight quantization.

12 Improved results

We used AutoKeras to generate multiple model architectures with varying shapes and layers. The baseline models achieved a maximum accuracy of 97.3885% at a gzipped size of approximately 190 KB, and 97.292% accuracy at around 25 KB.

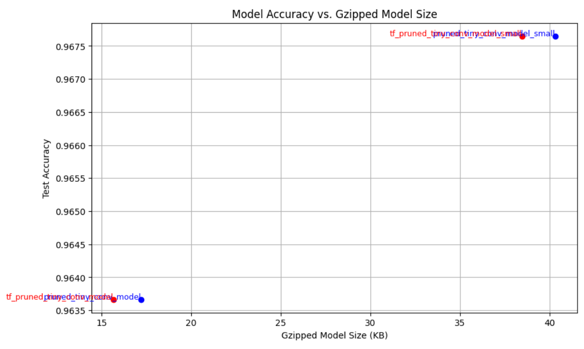

Figure 10: Pruned Models' accuracies and sizes

To optimize these models, we applied a series of compression techniques. First, pruning with a sparsity of 50% reduced the model sizes to 38 KB and 42 KB, while maintaining their respective accuracies.

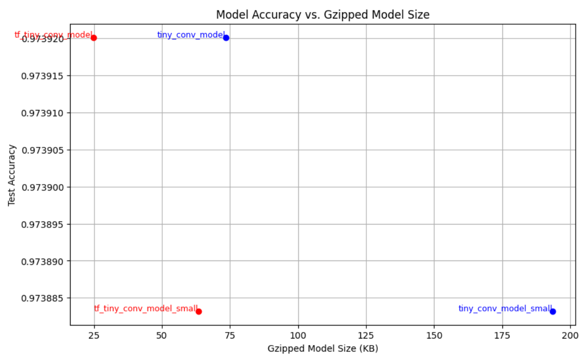

Figure 11: Baseline models accuracy and size

Next, we applied clustering with a cluster size of 10, further shrinking the models to 6 KB and 10 KB. The resulting accuracies for these clustered models were 97.290% and 97.250%, respectively.

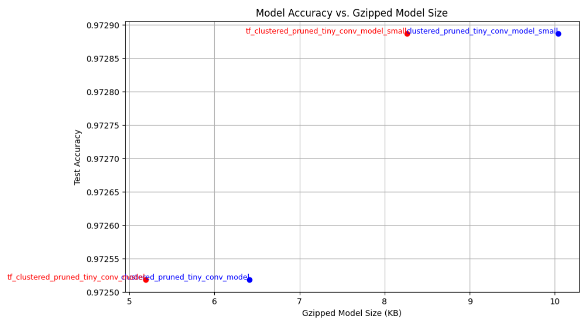

Figure 12: Pruned and Clustered models' accuracies and sizes

Lastly, we applied full integer post-training dynamic quantization, reducing the final model size to just 4 KB with an accuracy of 97%. This resulted in approximately 10x compression with minimal loss in accuracy. These results highlight that significantly fewer neurons are needed to achieve high performance, emphasizing the potential for efficient, lightweight model designs.

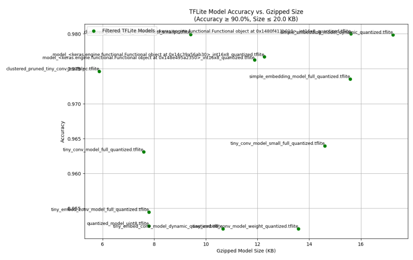

Figure 13: Quantized Models' accuracies and sizes

Through this pipeline, we demonstrated significant reductions in model size while preserving high accuracy, showcasing the potential of pruning and clustering for efficient model compression.

13 Conclusion

This project bridges the gap between sophisticated voice recognition technology and resource-constrained IoT devices. By enabling efficient wake word detection on the ESP32, we pave the way for more intelligent, responsive, and accessible smart devices that respect user privacy and energy conservation.

14 Resources

tensorflow micro speech github
espressif wakeNet github
espressif wakeNet
espressif wakeword github
espressif NN github
espressif tf-lite-micro github
MIT course on TinyML tensorflow micro speech github
Tensorflow Model Optimization
Weight Clustering
Pruning

Sparsity and cluster preserving quantization aware training

Limitations

- **Memory and Computational Limitations:** The inherent limited memory capacity of these micro-controllers poses a big challenge, especially when dealing with a bigger set of vocabulary. Further work on extending the system to recognize a multitude of words would require careful considerations of the tight memory and computational constraints.

- **On Device fine-tuning / training:** Our current model does not offer an on device fine-tuning, that can adapt to new user voices, or support a different set of vocabulary.

- **Larger Context window**: Further work to incorporate contextual clues such as previous interactions to enable more mode context-aware and intelligent word recognition.