MLEE Project Report

COURSE CODE: EEE G513
COURSE TITLE: Machine Learning for Electronics Engineers

Submitted by

Lovenya Jain - 2021B4AA1732P

Param Gupta - 2021A3PS0236P

Submitted to

Dr. Meetha V Shenoy, Assistant Professor

Pilani Campus
Birla Institute of Technology & Science, Pilani
Rajasthan, India

Introduction

Keyword Spotting (KWS) is a crucial technology for enabling voice-activated systems and applications. It allows devices to recognize specific predefined words (such as “Hello”, “Stop”, “Play”) in real-time, making them responsive to user commands. This capability is often utilized in smart assistants (e.g., Amazon Alexa, Google Assistant) and other IoT devices, enabling them to perform actions upon hearing specific words or phrases.

However, for many edge devices like microcontrollers, which have limited computational resources (both in terms of memory and processing power), implementing such systems poses significant challenges. These devices often do not have the computational power of more robust devices like smartphones or desktops, so a well-optimized KWS model is necessary to ensure fast and efficient performance. Edge deployment in embedded systems like the **Arduino Nano 33 BLE Sense** requires a careful balance between model size, inference speed, and memory usage.

Challenges in Edge Deployment for KWS

- **Memory Constraints**: Microcontrollers like the Arduino Nano 33 BLE Sense have limited memory and storage. The Arduino Nano 33 BLE Sense, for example, has only 1MB of flash memory and 256KB of SRAM. This imposes strict limits on the size of the model and the complexity of the computation that can be done on-device.
- **Latency**: For real-time applications, especially in speech recognition, latency is a critical factor. For the system to be useful in real-time scenarios, the model must process incoming audio data and make predictions quickly—typically within tens of milliseconds. This makes it essential to optimize both the model and its deployment to minimize inference time.
- **Power Efficiency**: Edge devices often run on battery power, so it is crucial to design the model and inference process to consume as little energy as possible while still maintaining performance.
- **Data Privacy**: Deploying KWS models on edge devices also enhances privacy, as the audio data never leaves the device and is

processed locally, mitigating concerns about sending sensitive data to cloud servers.

Project Goal and Objective

The goal of this project is to develop a low-latency Keyword Spotting system suitable for deployment on the **Arduino Nano 33 BLE Sense**. The project focuses on optimizing the model architecture, preprocessing pipeline, and the inference process to meet the constraints of the device in terms of both memory and processing power. This involves using **lightweight neural network architectures** such as **depthwise separable convolutions** and **GRU-based recurrent layers**, which are well-suited for edge deployment due to their smaller size and lower computational requirements compared to traditional convolutional neural networks (CNNs).

1. Techniques applied to solve the problem-starting from data pre-processing, tuning parameters

Feature Extraction

Transforms speech signals into feature representations for **model training and inference.** We used MFCC techniques for our pre-processing pipeline.

Steps:

Pre-emphasis: High-pass filtering to balance high/low frequencies.
Frame Splitting and Windowing: Ensures smooth transitions and reduces spectral leakage.
FFT: Converts windowed signals to frequency domain.
Mel Filter Bank: Non-linear scaling for high resolution at lower frequencies.
DCT: Generates decorrelated coefficients for compact representation.

A. Pre - Emphasis

Pre-emphasis is a signal processing technique applied to speech signals to enhance **high-frequency components**.
It uses a **high-pass filter**, typically defined as:

$$y(t) = x(t) - ax(t-1), a = 0.9375$$

B. Splitting frames Adding windows

Although the speech signal is a non-smooth process, there exists a short-time smooth process characteristic; the Fourier transform of short-time frames, which is used to obtain a good approximation of the signal frequency profile by **connecting neighbouring frames.**
The algorithm segments the speech waveform using a fixed length and defines the frame length. Adjacent frames overlap, and the distance moved between frames is called the frame shift.
The text provides an example using a 32ms/frame frame length. This results in 256 normalized sampling points - a 20ms frame shift with 160 sampling points, and a 12ms overlap region with 96 sampling points. This

is done to avoid large changes between adjacent frames.

C. FFT Technique

After the signal passes through the Hamming window, its energy distribution over the spectrum is obtained by FFT, i.e., the specturm of each frame is obtained.

$$S_i(k) = \sum_{n=1}^{N} s_i(n) e^{\frac{-j2\pi kn}{N}} \quad 1 \leq k \leq N-1$$

D. MEL Filters Banks

The Mel filter is a simulation of the human auditory perception system, defining the Mel scale to simulate the non-linear perception of sound by the human ear.

1. Then we **convert frequencies to mel frequencies.**

   Converts from frequency to Mel scale:

   $$f_{mel} = 2595 \times log_{10}\left(1 + \frac{f}{700}\right)$$

2. **Apply Mel Filter Bank:**
   - **a.** Once the FFT spectrum is mapped to Mel frequencies, a Mel filter bank is applied to this spectrum. The Mel filter bank consists of **triangular filters** that **emphasize lower frequencies** and gradually taper off at higher frequencies, simulating how the human ear perceives sound.
   - **b.** The result is a Mel-spectrogram that **highlights speech-relevant features** while reducing **unnecessary high-frequency details.**

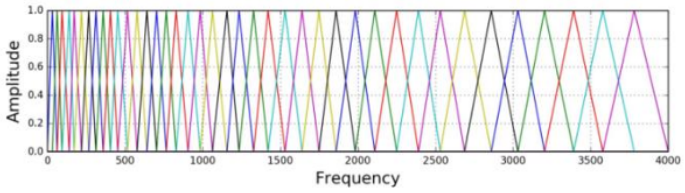
The calculations behind MEL Filters Banks

$$f(i) = \left\lfloor (nfft + 1) \times \frac{h(i)}{sample_{rate}} \right\rfloor \qquad (7)$$

The frequency response of a triangular filter is defined by modelling it by the following equation.

$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \le k < f(m) \\ 1 & k = f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)} & f(m) < k \le f(m+1) \\ 0 & k > f(m+1) \end{cases} \qquad (8)$$

- **f(i)** is the quantized frequency corresponding to the iii-th Mel filter.
- $\mathbf{n_{fft}}$ is the number of FFT bins (i.e., the number of frequency bins after performing the FFT).
- **h(i)** is the Mel frequency value for the i-th Mel filter.
- $\mathbf{sample_{rate}}$ is the sampling rate of the signal (e.g., 8kHz in the paper's case.).
- The floor function ⌊·⌋ is used to round down to the nearest integer, effectively mapping the continuous Mel frequency to a **discrete bin.**
- f(m+1), f(m-1) is the value of f at the next and previous frequency in relative to the peak of triangle which is at the center.



E. Discrete Cosine Transform

The human ear perceives signals as an approximation to Log, and the logarithm enables features to be insensitive to perturbations in the input signal, and the **logarithmic energy** of the output for each filter bank is:

$$s(m) = ln\left(\sum_{m=0}^{N-1} |S_i(k)|^2 H_m(k)\right) \quad 0 \le m \le M$$

The MFCC coefficients are obtained by applying the Discrete Cosine Transform (DCT).

$$C(n) = \sum_{m=0}^{N-1} s(m)\, cos\left(\frac{\pi n(m-0.5)}{M}\right) \quad n = 1,2,\cdots,L \quad (10)$$

Order L refers to the order of the MFCC coefficients, and usually **L takes the value of 12-16**.
For speech recognition, the common cepstrum coefficients **2-13 are retained**, and the rest are discarded, and the other coefficients are discarded because they indicate the rapid change of the coefficients of the filter bank and the details will not contribute to the speech recognition.

2. Model Architecture, model optimization

In this section, we will discuss the **model architecture** used for Keyword Spotting (KWS) on the Arduino Nano 33 BLE Sense and the **optimization techniques** applied to meet the device's constraints of memory and computational power. Specifically, we will cover the use of Depthwise Separable Convolutional Neural Networks (DS-CNN), Recurrent Convolutional Neural Networks (RCNN), and Model Distillation for improving the performance and efficiency of the model.

1. Model Architecture

The architecture for the KWS model is designed to be both lightweight and effective in recognizing keywords from audio data captured by the Arduino Nano 33 BLE Sense. The model architecture combines **Depthwise Separable Convolutions** (DS-CNN) and **Recurrent Neural Networks** (RNN), specifically **Bidirectional GRU** layers, to balance feature extraction and sequence modeling. Here's how the model is structured:

1.1 Depthwise Separable Convolutional Neural Networks (DS-CNN)

Depthwise Separable Convolutions (DS-CNN) are a highly efficient variation of traditional CNNs that reduce the number of parameters and computation required. A traditional convolution layer performs both spatial convolution and depthwise convolution, which can be computationally expensive. **DS-CNNs**, on the other hand, break down the convolution process into two separate operations:

1. **Depthwise Convolution**: Each input channel is convolved with its own filter. This significantly reduces the number of parameters compared to standard convolutions.
2. **Pointwise Convolution**: A 1×1 convolution is applied across all the input channels to combine the outputs from the depthwise convolution. This helps in mixing the feature maps from different channels while keeping the computational cost low.

Advantages of DS-CNN:

- **Fewer Parameters**: DS-CNN significantly reduces the number of parameters compared to regular CNNs, making the model lightweight and suitable for edge devices with limited memory.
- **Computational Efficiency**: Since it reduces the number of operations required per convolution, DS-CNNs are faster and more energy-efficient, which is crucial for real-time applications on embedded systems like the Arduino Nano 33 BLE Sense.

1.2 Recurrent Convolutional Neural Networks (RCNN)

Recurrent Convolutional Neural Networks (RCNN) combine the power of convolutional layers (for local feature extraction) with the ability of recurrent layers (such as GRU or LSTM) to capture sequential dependencies. In this architecture, convolutional layers act as feature extractors, and the recurrent layer is responsible for learning long-term temporal patterns across frames.

- **Convolutional Layers**: These layers extract local features from the input, such as the frequency patterns in the audio signal (in this case, the MFCC features).
- **Recurrent Layers (GRU)**: After extracting features through convolutions, the **Bidirectional GRU** layer processes these features in both forward and backward directions, which allows the model to understand context from both past and future frames. This is essential for capturing the temporal dependencies in speech data, as the context of a keyword can depend on both the preceding and following audio frames.

Advantages of RCNN:

- **Sequential Context**: RCNNs allow the model to understand the temporal dependencies in speech data, which is crucial for keyword spotting tasks where the context and order of words matter.
- **Flexibility**: The combination of convolutional and recurrent layers provides flexibility, enabling the model to learn both local and global patterns efficiently.

---

2. Model Optimization

Given the constraints of the **Arduino Nano 33 BLE Sense**, optimizing the model is crucial to ensure that it works effectively without exceeding the device's memory and processing limitations. The optimization techniques used in this project include **quantization**, and **model distillation**. Below, we will discuss how these techniques were applied to the KWS model.

2.1 Quantization

Quantization is a technique that reduces the precision of the model weights and activations to lower bit-widths (e.g., 8-bit integers instead of 32-bit floats). This significantly reduces the model size and speeds up inference, as operations on lower-precision data are faster and require less memory. TensorFlow Lite offers support for **Post-training Quantization**, which allows us to reduce the model size after training without requiring retraining.

- **Post-Training Quantization**: The KWS model was quantized to **INT8** precision for both weights and activations. This reduced the model size and enabled faster inference on the Arduino device, while maintaining the performance accuracy close to the original model.

2.2 Model Distillation

Model Distillation is a technique where a smaller model (the student model) is trained to replicate the behavior of a larger, more complex model (the teacher model). The smaller model is trained using the outputs (soft labels) of the teacher model instead of the original hard labels. This allows the smaller model to learn to approximate the performance of the larger model while using fewer parameters and requiring less computation.

In the context of this project:

- **Teacher Model**: A larger, more complex KWS model with higher accuracy but unsuitable for edge deployment due to its size and computation requirements.

- **Student Model**: A smaller, optimized version of the KWS model that was trained to approximate the teacher model's performance using **soft labels** produced by the teacher model. This model is lightweight enough to be deployed on the **Arduino Nano 33 BLE Sense**.

Advantages of Model Distillation:

- **Smaller Model Size**: The student model is smaller and more efficient, making it suitable for deployment on resource-constrained devices like the Arduino Nano 33 BLE Sense.
- **Preserved Performance**: Despite the reduced size, the student model retains much of the performance of the larger teacher model, making distillation an effective optimization technique.

3. Architecture and Training parameters

a. DS-CNN

Model Architecture Parameters

1. **Input Shape**:
    - `(99, 12, 1)` — Corresponds to the shape of the MFCC features (99 frames, 12 coefficients, 1 channel).
2. **Convolutional Layers**:
    - `Conv2D(64, (3, 3), activation="relu", strides=(1, 1))` — 64 filters, 3x3 kernel size, ReLU activation, stride of 1.
    - `DepthwiseConv2D((3, 3), activation="relu")` — Depthwise convolution for parameter-efficient learning.
    - `Conv2D(64, (1, 1), activation="relu")` — Pointwise convolution for feature refinement.
3. **Pooling and Dense Layers**:
    - `GlobalAveragePooling2D()` — Reduces the spatial dimensions to a single value per channel.
    - `Dense(len(class_names), activation="softmax")` — Final classification layer, with the number of outputs matching the number of classes.

Training Parameters

1. **Optimizer**:
    - `Adam` optimizer (default parameters: `learning_rate=0.001`).
2. **Loss Function**:

    - Sparse Categorical Crossentropy (sparse_categorical_crossentropy) — Used for multi-class classification with integer labels.
3. **Metrics**:
    - Accuracy — Monitors model performance during training and validation.
4. **Batch Size**:
    - 32
5. **Number of Epochs**:
    - epochs=100

b. R-CNN

Model Architecture Parameters

1. **Input Shape**:
    - [None, 12, 1] — Suggests input consists of a time dimension (None for variable length), 12 MFCC coefficients, and 1 channel.
2. **Convolutional Layers**:
    - Conv2D(64, (3, 3), activation="relu", padding="same") — 64 filters, 3x3 kernel, ReLU activation, same padding.
    - BatchNormalization — Normalizes activations for faster convergence.
    - MaxPooling2D(pool_size=(2, 2)) — Down-samples feature maps by a factor of 2.
    - Conv2D(128, (3, 3), activation="relu", padding="same") — Similar structure with 128 filters.
3. **Reshaping for Recurrent Layer**:
    - Reshape((-1, x.shape[-1] * x.shape[-2])) — Converts spatial data into sequences for the RNN.
4. **Recurrent Layer**:
    - Bidirectional(layers.GRU(128, return_sequences=False)) — A bidirectional GRU layer with 128 units. Captures sequential relationships in MFCC data.
5. **Dropout**:
    - Dropout(0.5) — Reduces overfitting by randomly disabling 50% of GRU outputs.
6. **Output Layer**:
    - Dense(num_classes, activation="softmax") — Predicts class probabilities, where num_classes = len(class_names).

Training Parameters

1. **Optimizer**:
    - Adam optimizer (default learning_rate=0.001).
2. **Loss Function**:
    - Sparse Categorical Crossentropy (sparse_categorical_crossentropy) — Suitable for multi-class classification with integer labels.
3. **Metrics**:
    - Accuracy — Tracks performance during training and validation.
4. **Batch Size**:
    - 32.
5. **Number of Epochs**:
    - epochs=10

c. Knowledge Distillation

Teacher Model

- **Architecture**:
  - Layers:
    - Conv2D with 128 filters and ReLU activation.
    - Batch Normalization after the second Conv2D layer.
    - MaxPooling2D with a pool size of (2, 2).
    - GlobalAveragePooling2D reduces spatial dimensions.
    - Dense output layer with softmax activation.
- **Training**:
  - **Loss**: SparseCategoricalCrossentropy.
  - **Optimizer**: Adam.
  - **Epochs**: 7.
  - **Batch Size**: 32

Student Model

- **Architecture**:
  - Compact architecture leveraging depthwise separable convolutions:
    - Conv2D: 64 filters, kernel size (3, 3).
    - DepthwiseConv2D: Efficient spatial filtering.
    - Conv2D with kernel size (1, 1) for feature combination.
    - GlobalAveragePooling2D for dimensionality reduction.
    - Dense output layer with softmax.

Knowledge Distillation

1. **Distillation Loss**:
   - Combines:
     - **Hard Loss**: Cross-entropy between true labels and student predictions.
     - **Soft Loss**: Kullback-Leibler divergence between teacher and student logits (softened using temperature).
   - Weighting controlled by:
     - alpha=0.7 (hard label emphasis).
     - temperature=5 (soft label smoothing).
2. **Training**:
   - Teacher logits are extracted from the second-to-last teacher layer (teacher_model.layers[-2].output).
   - **Learning Rate Scheduler**:
     - Exponential decay: 0.001 * 0.95^epoch.
   - **Callbacks**:
     - Early stopping with patience of 5 epochs on validation loss.

4. Performance Parameters

In evaluating the efficacy of different neural network architectures for keyword spotting on the Arduino Nano BLE 33 Sense, the critical performance parameter analyzed was: **inference latency**. These metrics are particularly crucial for edge deployment scenarios where both storage constraints and real-time performance requirements must be satisfied.

Inference Latency

Inference latency measurements capture the time required to process a single audio input and generate the corresponding keyword prediction. The following latency metrics were observed:

- DS-CNN: 95 milliseconds
- RCNN: 100 milliseconds
- Knowledge Distillation Model: 96 milliseconds

The latency measurements were conducted under consistent conditions, with the Arduino Nano BLE 33 Sense operating at its standard clock frequency of 64 MHz. Each model was tested with identical input samples to ensure fair comparison.

5. Performance Parameters

Accuracy Metrics

The evaluation of keyword spotting models implemented on the Arduino Nano BLE 33 Sense reveals varying levels of recognition accuracy across the three architectures. Each model was assessed using standard metrics including accuracy, precision, recall, and F1-score to provide a comprehensive understanding of their performance capabilities..

Model-wise Performance

DS-CNN (Depthwise Separable CNN)

```
poch 81: Training Accuracy = 0.8731, Validation Accuracy = 0.8569
poch 82: Training Accuracy = 0.8731, Validation Accuracy = 0.8576
poch 83: Training Accuracy = 0.8738, Validation Accuracy = 0.8577
poch 84: Training Accuracy = 0.8743, Validation Accuracy = 0.8576
poch 85: Training Accuracy = 0.8746, Validation Accuracy = 0.8588
poch 86: Training Accuracy = 0.8752, Validation Accuracy = 0.8590
poch 87: Training Accuracy = 0.8756, Validation Accuracy = 0.8586
poch 88: Training Accuracy = 0.8761, Validation Accuracy = 0.8591
poch 89: Training Accuracy = 0.8766, Validation Accuracy = 0.8595
poch 90: Training Accuracy = 0.8766, Validation Accuracy = 0.8601
poch 91: Training Accuracy = 0.8775, Validation Accuracy = 0.8600
poch 92: Training Accuracy = 0.8776, Validation Accuracy = 0.8604
poch 93: Training Accuracy = 0.8780, Validation Accuracy = 0.8608
poch 94: Training Accuracy = 0.8781, Validation Accuracy = 0.8619
poch 95: Training Accuracy = 0.8787, Validation Accuracy = 0.8623
poch 96: Training Accuracy = 0.8791, Validation Accuracy = 0.8619
poch 97: Training Accuracy = 0.8797, Validation Accuracy = 0.8625
poch 98: Training Accuracy = 0.8798, Validation Accuracy = 0.8621
poch 99: Training Accuracy = 0.8799, Validation Accuracy = 0.8614
poch 100: Training Accuracy = 0.8803, Validation Accuracy = 0.8620
```

RCNN (Recurrent CNN)

```
Epoch 1/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 117s 44ms/step - accuracy: 0.5653 - loss: 1.6050 - val_accuracy: 0.8364 - val_loss: 0.5460
Epoch 2/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 112s 42ms/step - accuracy: 0.8516 - loss: 0.5024 - val_accuracy: 0.8625 - val_loss: 0.4533
Epoch 3/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 112s 42ms/step - accuracy: 0.8825 - loss: 0.3973 - val_accuracy: 0.8848 - val_loss: 0.3881
Epoch 4/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 134s 51ms/step - accuracy: 0.8999 - loss: 0.3371 - val_accuracy: 0.8831 - val_loss: 0.4055
Epoch 5/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 130s 49ms/step - accuracy: 0.9096 - loss: 0.3012 - val_accuracy: 0.8914 - val_loss: 0.3782
Epoch 6/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 148s 56ms/step - accuracy: 0.9182 - loss: 0.2685 - val_accuracy: 0.8908 - val_loss: 0.3844
Epoch 7/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 164s 62ms/step - accuracy: 0.9277 - loss: 0.2386 - val_accuracy: 0.8996 - val_loss: 0.3517
Epoch 8/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 163s 61ms/step - accuracy: 0.9319 - loss: 0.2180 - val_accuracy: 0.9060 - val_loss: 0.3374
Epoch 9/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 162s 61ms/step - accuracy: 0.9383 - loss: 0.2000 - val_accuracy: 0.8933 - val_loss: 0.3848
Epoch 10/10
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 162s 61ms/step - accuracy: 0.9420 - loss: 0.1831 - val_accuracy: 0.8995 - val_loss: 0.3684
```

Classification Report

	precision	recall	f1-score
_background_noise_	0.00	0.00	0.00
backward	0.99	0.84	0.90
bed	0.89	0.76	0.82
bird	0.93	0.78	0.85
cat	0.92	0.83	0.87
dog	0.92	0.78	0.85
down	0.92	0.81	0.86
eight	0.97	0.89	0.93
five	0.85	0.93	0.89
follow	0.62	0.87	0.73
forward	0.86	0.69	0.77
four	0.77	0.92	0.84
go	0.91	0.80	0.85
happy	0.96	0.89	0.92
house	0.82	0.91	0.86
learn	0.89	0.54	0.67
left	0.96	0.89	0.92
marvin	0.91	0.93	0.92
nine	0.98	0.87	0.92
no	0.81	0.90	0.85
off	0.67	0.93	0.78
on	0.93	0.74	0.83
one	0.99	0.82	0.90
right	0.97	0.91	0.94
seven	0.89	0.93	0.91
sheila	0.89	0.96	0.92
six	0.85	0.96	0.90
stop	0.86	0.94	0.90
three	0.89	0.85	0.87
tree	0.76	0.85	0.80
two	0.87	0.92	0.89
up	0.91	0.71	0.80
visual	0.87	0.88	0.87
wow	0.86	0.90	0.88
yes	0.92	0.96	0.94
zero	0.76	0.98	0.85
accuracy			0.87
macro avg	0.85	0.84	0.84
weighted avg	0.88	0.87	0.87

Knowledge Distillation Model

```
Epoch 1/7
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 324s 122ms/step - accuracy: 0.5887 - loss: 1.4573 - val_accuracy: 0.7864 - val_loss: 0.8165
Epoch 2/7
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 415s 157ms/step - accuracy: 0.8577 - loss: 0.4860 - val_accuracy: 0.8025 - val_loss: 0.7263
Epoch 3/7
2646/2646 ━━━━━━━━━━━━━━━━━━━━ 223s 84ms/step - accuracy: 0.8907 - loss: 0.3717 - val_accuracy: 0.8211 - val_loss: 0.6693
```

Codes:

Param-GG/Embedded-AI-for-Low-Latency-Speech-Recognition

It includes a lot of unnecessary files as a part of experimenting.
I'll mention the relevant files here.
download_dataset.py
preprocess_data.py
ds_cnn_train.py
rcnn_train.py
knowledge_distillation.py
edge_device_deployment/edge_keyword_spotting/edge_rcnn/