

# BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

## PROJECT REPORT

EEE-G513: Machine Learning for Electronics Engineers

Date of Submission – 4<sup>th</sup> December 2023

Students involved -

Priyanshi Varshney	2022H1240084P
Bharathi Shrinivasan T R	2022H1400182P
Avadh Harkishanka	2018HS230322P

---

**Objective:** Learn complete workflow of solving an engineering problem using ML technique. And implement a run-time inferencing in Embedded system.

**Project:** The project addresses the challenge of autonomous speech-based robot control, specifically focusing on speech classification. The goal is to enable a robotic subsystem to respond to voice commands such as "forwards" and "backwards" using machine learning techniques.

**Project database links:**

Complete Google Drive - [Link](#)

1. Collected dataset
2. Source code of training model, preparing tflite model, pre-processing test input for Inference in Arduino (Google colab .ipynb)
3. Arduino sketch for TinyML implementation in target board – Arduino Nano 33 BLE. (.ino)
4. Our model dump, tflite model dump (.keras)
5. Our model dump in bytes (c-array) – (.h)
6. Finally, a complete demo video

---

## *Table of Contents:*

---

- *Components*
  - *Data Preparation*
  - *Voice feature - Spectrogram. Preprocessing of dataset*
  - *Training of multiple ML models*
  - *Model Architectures*
  - *Quantize/ Prune to get tf lite model*
  - *Deploying and running the interface*
  - *Conclusion*
-

## Activities Carried:

### 1. Dataset Preparation:

- Collection of a dataset consisting of voice samples for commands like forward, reverse, etc.
- Voice features extracted through spectrogram analysis for model training.

### 2. Model Training:

- Utilization of various machine learning models, including Artificial Neural Networks (ANN), 2D Convolutional Neural Networks (CNN), and 1D CNN.
- Extensive training on the dataset to enable accurate speech command recognition.

### 3. Model Optimization:

- Quantization and pruning of the trained model for memory-efficient deployment.
- Conversion to TensorFlow Lite format for compatibility with embedded systems.

### 4. Arduino Nano 33 BLE Integration:

- Integration of the TensorFlow Lite model into Arduino Nano 33 BLE.
- Implementation of an interpreter on Arduino for real-time inferencing.

### 5. Live Speech Classification:

- Utilization of the deployed model on Arduino for live speech classification.
- Input tensor setup, interpreter invocation, and output verification on the Arduino Nano 33 BLE.

### 6. Quantization and Memory Management:

- Quantization of the model to optimize for size and memory constraints.
- Evaluation and optimization of memory usage to fit within Arduino Nano's limitations.

### 7. Alternative Deployment (Edge Impulse):

- Exploration of Edge Impulse as an alternative deployment platform.
- Training and testing the model on Edge Impulse with a provided dataset.

## **Data Preparation:**

The data preparation process involved recording, pre-processing, and labeling audio data, followed by the extraction of relevant features for training a speech classification model. The dataset was split for training and testing, and various models were experimented with to find the most suitable architecture for the given task. This comprehensive approach ensures that the model is well-prepared to accurately classify speech commands in real-world scenarios. Let's break down the process of data preparation:

### **1. Audio Recording:**

You used Audacity to record live speech for different commands, including "forward," "reverse," and an "unknown" category. The recordings were saved in 16-bit PCM format, ensuring compatibility with subsequent processing steps.

### **2. Recording Specifications:**

The recordings were made at a 16 kHz sampling rate in mono channel format. The audio data was recorded with 16-bit precision to match the requirements of the subsequent processing steps.

### **3. Pre-processing in Colab:**

In Google Colab, you performed pre-processing on the recorded audio to prepare the data for model input. The pre-processing involved converting the audio data into a suitable format for feature extraction.

### **4. Feature Extraction:**

You utilized Mel Frequency Cepstral Coefficients (MFCCs) as features for the speech classification model.

The feature extraction process involved transforming the raw audio data into a set of coefficients that represent the characteristics of the audio.

### **5. Labeling and Categorization:**

Each audio sample was labeled based on the corresponding command (e.g., "forward," "reverse," or "unknown"). The labeled data was categorized into different classes to train a supervised machine learning model.

### **6. Dataset Splitting:**

The dataset was split into training and testing sets to evaluate the model's performance accurately. The training set was used to train the machine learning models, while the testing set was reserved for evaluating the model's accuracy on unseen data.

### **7. Model Training:**

Different machine learning models were experimented with, including 2D convolutions, 1D convolutions, and various architectures. The models were trained using the pre-processed and labeled audio data.

### **8. Model Evaluation:**

The trained models were evaluated on a testing dataset to assess their accuracy and performance. Metrics such as confusion matrices and accuracy were used to analyze the model's effectiveness.

## Voice feature - Spectrogram. Preprocessing of dataset:

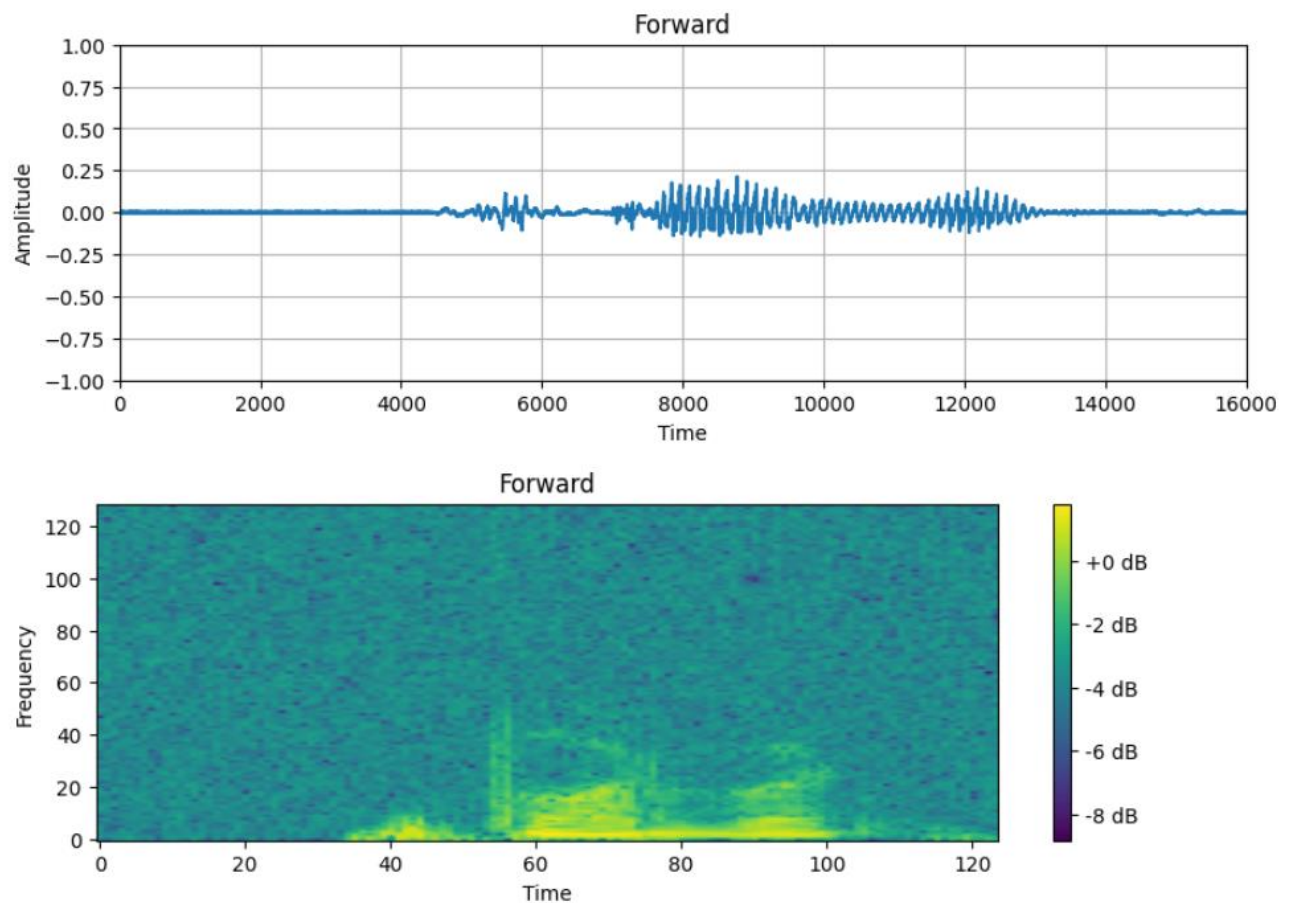
The dataset underwent preprocessing, including labeling, splitting, and feature extraction. The creation of spectrograms facilitated the extraction of MFCC features, which served as inputs for training machine learning models. This comprehensive preprocessing ensured that the dataset was well-structured for training and evaluating speech classification models.

### Spectrogram:

A spectrogram is a visual representation of the spectrum of frequencies of a signal as they vary with time. It provides a way to analyze how the frequencies of a signal change over time. The raw audio data collected during the recording phase was converted into a spectrogram as part of the feature extraction process.

### Processing Steps:

- The process of creating a spectrogram involves breaking the audio signal into short segments called frames.
- For each frame, a mathematical transformation (usually the Fast Fourier Transform or FFT) is applied to obtain the frequency components.
- The result is a 2D representation where one axis represents time, the other frequency, and the color intensity represents the magnitude of each frequency component.



### Mel Frequency Cepstral Coefficients (MFCCs):

MFCCs are often derived from the spectrogram. These coefficients capture the power spectrum of an audio signal. The MFCCs represent the short-term power spectrum of a sound signal, emphasizing the characteristics of human hearing.

### Preprocessing of Dataset:

1. **Labeling:** Each audio recording was labeled based on the command it represented, such as "forward," "reverse," or "unknown."
2. **Data Splitting:** The dataset was divided into training and testing sets to facilitate model evaluation. This ensures that the model's performance is assessed on unseen data.
3. **Feature Extraction (MFCCs):** The spectrogram was used to extract Mel Frequency Cepstral Coefficients (MFCCs) as features for each audio sample. MFCCs provide a compact representation of the spectral characteristics of the audio signals.
4. **Dataset Preparation:** The dataset was prepared by associating each audio sample with its corresponding MFCC features and the labeled command. This pairing of features and labels is crucial for supervised machine learning tasks.
5. **Model Training:** Different machine learning models were trained using the preprocessed dataset. These models were designed to classify speech commands based on the extracted MFCC features.
6. **Model Evaluation:** The trained models were evaluated using a separate testing dataset to measure their accuracy and performance. Metrics such as confusion matrices and accuracy were employed to assess the models' effectiveness in classifying speech commands.

## Training of multiple ML models:

This comprehensive explanation provides a high-level understanding of the entire training process for different machine learning models in our speech classification project. Let's delve into the detailed process of training multiple machine learning models (Artificial Neural Network - ANN, 2D Convolutional Neural Network - CNN, and 1D CNN) for our speech classification project.

### Model Architectures:

#### Artificial Neural Network (ANN):

- Input Layer: Flatten the 2D spectrogram into a 1D vector.
- Hidden Layers: Incorporate dense layers for learning hierarchical features. Use activation functions like ReLU to introduce non-linearity. Apply dropout to prevent overfitting.
- Output Layer: A softmax layer to produce class probabilities. Choose the class with the highest probability as the final prediction.

#### 2D Convolutional Neural Network (CNN):

- Input Layer: Accept 2D spectrogram images directly.
- Convolutional Layers: Utilize convolutional layers with filters to capture spatial patterns. Apply activation functions like ReLU.
- Pooling Layers: Downsample the spatial dimensions to reduce computational complexity.
- Flatten Layer: Transform the 2D output into a 1D vector.
- Dense Layers: Implement dense layers for classification.
- Output Layer: A softmax layer for multi-class classification.

#### 1D Convolutional Neural Network (CNN):

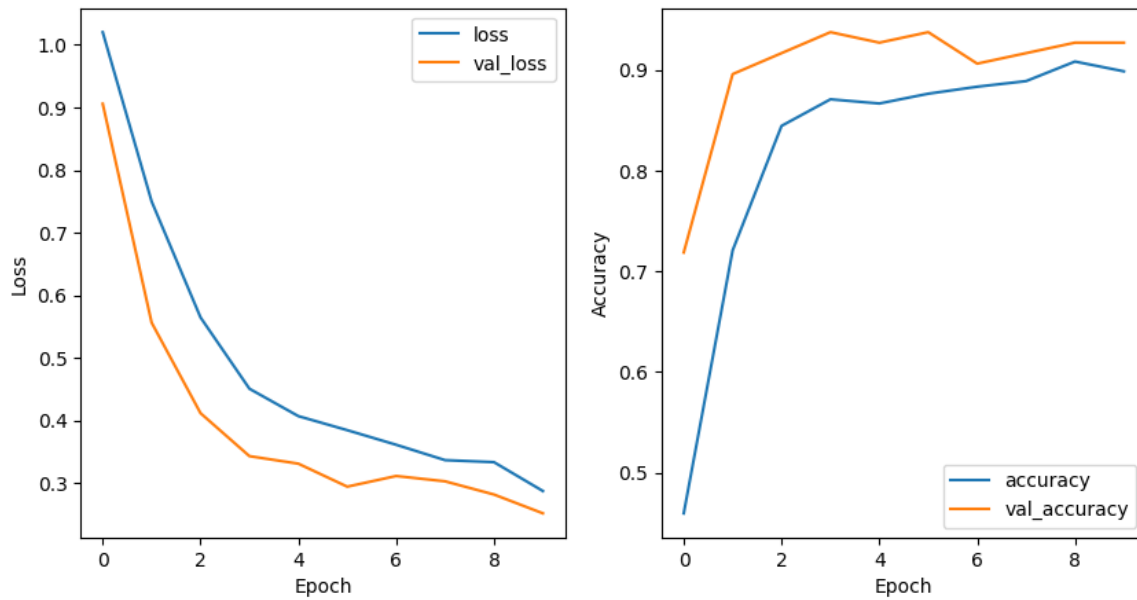
- Input Layer: Accept 1D spectrogram vectors.
- Convolutional Layers: Use 1D convolutional layers with kernels for extracting temporal features. Apply activation functions.
- Flatten Layer: Convert the 1D output into a flat vector.
- Dense Layers: Include dense layers for higher-level feature learning.
- Output Layer: A softmax layer for classification.

### Training Process:

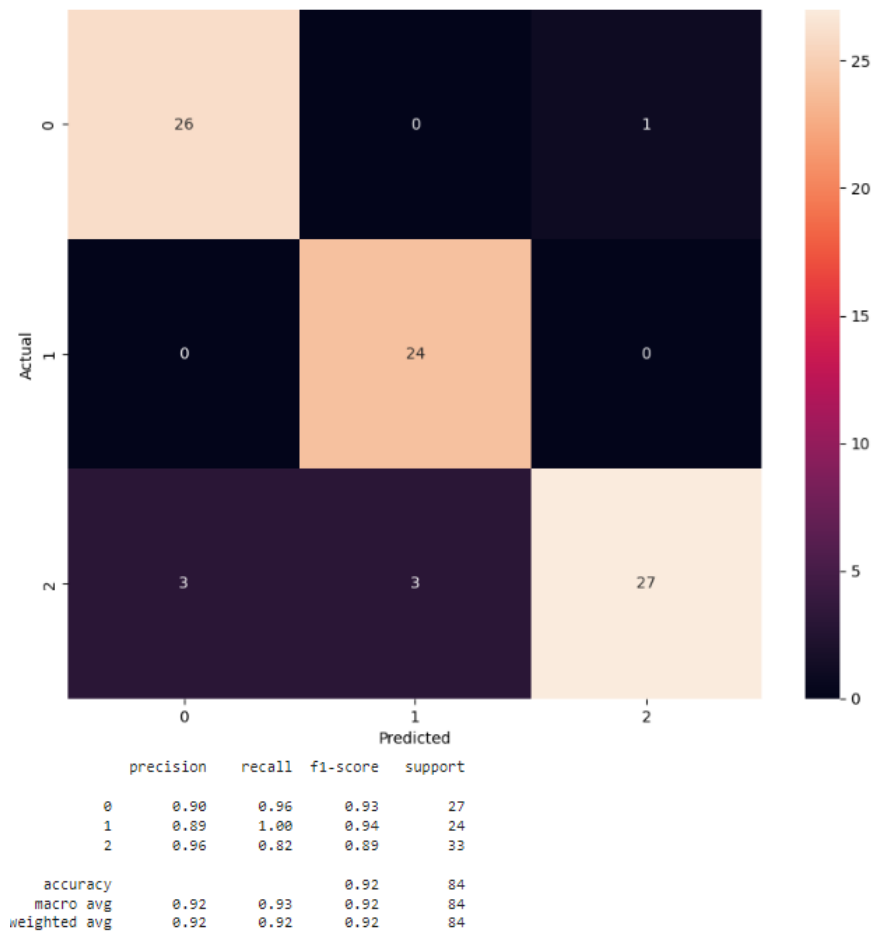
- Loss Function: Utilize categorical cross-entropy loss for multi-class classification.
- Optimizer: Choose an optimizer (e.g., Adam) for updating weights during training.
- Metrics: Monitor metrics such as accuracy, precision, recall, and F1-score during training.
- Epochs: Train each model for a certain number of epochs.
- Validate performance on a separate validation set.
- Batch Size: Specify the number of samples processed before updating the model's weights.

History of model-1

6/6 [-----] - 0s 19ms/step



Complete training data is in Google colab sheet.





**Model Evaluation:**

- **Performance Metrics:** Evaluate models on the test set using various metrics. Analyze accuracy, confusion matrix, precision, recall, and F1-score.
- **Comparison:** Compare the performance of different models to identify the most effective one.

**Model Selection Criteria:** Choose the model that best balances performance and computational efficiency.

- **Hyperparameter Tuning:** Fine-tune hyperparameters for the selected model to enhance performance.
- **Deployment Considerations:**
- **Model Size:** Consider the memory constraints of the deployment platform. Optimize model size if required.
- **Inference Speed:** Assess the speed of model inference, crucial for real-time applications.
- **Compatibility:** Ensure compatibility with the target deployment platform (e.g., Arduino Nano).

## Quantize/ Prune to get tflite model:

The steps involved in preparing the model for quantization and pruning, converting it to TensorFlow Lite format, and deploying it on Arduino Nano. Emphasize the balance between model size, computational efficiency, and accuracy, crucial for successful deployment on resource-constrained platforms.

### Preparing the Model for Quantization:

Model Background: The pre-trained model used in the speech classification project is based on a 2D convolutional neural network (CNN) with 16 layers.

#### Quantization Steps:

- **Weight Quantization:** Convert model weights to 8-bit integers using the TensorFlow Lite Converter, optimizing for size.
- **Activation Quantization:** Quantize intermediate activations during inference.
- **TensorFlow Lite Converter:** Employ the TensorFlow Lite Converter to quantize the pre-trained 2D CNN model. Specify quantization details during conversion for both weights and activations.

### Preparing the Model for Pruning:

- **Model Pruning Techniques:** Identify and prune less significant weights in the 2D CNN model to reduce its size. Use the TensorFlow Model Optimization Toolkit, applying pruning algorithms.

### TensorFlow Model Optimization Toolkit:

Utilize the toolkit to apply weight pruning to the pre-trained model. Fine-tune the pruned model to recover any accuracy loss.

### Conversion to TensorFlow Lite (TFLite) Model:

- **TFLite Model Conversion:** Convert the quantized and pruned model to the TFLite format using the TensorFlow Lite Converter. Specify the target platform, such as tflite.Microcontroller.
- **Model Size and Inference Performance:** Assess the reduced size of the TFLite model, crucial for deploying on resource-constrained devices. Verify the real-time inference performance of the TFLite model.

### Deployment on Arduino Nano:

Integration with Arduino TensorFlow Lite Library: Leverage the Arduino TensorFlow Lite library to seamlessly integrate the TFLite model with Arduino Nano. Implement an interpreter within the Arduino code.

Inference on Arduino Nano: Embed the TFLite model in the Arduino code for real-time speech classification. Develop code to handle input data, invoke the interpreter, and process the output.

Model Validation on Arduino: Validate the accuracy of the deployed TFLite model on Arduino Nano using a test set. Ensure that the model meets the memory constraints of the microcontroller.

## Deploying and running the interface:

Steps taken to deploy the TensorFlow Lite model on Arduino Nano 33 BLE, emphasizing live speech classification and real-time inferencing in an embedded system.

### Deploying in Arduino Nano 33 BLE:

#### 1. Integration with TensorFlow Lite:

- Utilize the Arduino TensorFlow Lite library to seamlessly integrate the TensorFlow Lite model into Arduino Nano 33 BLE.
- Incorporate the necessary libraries and dependencies in the Arduino code.

#### 2. Interpreter Setup:

- Implement an interpreter within the Arduino code to facilitate real-time inferencing.
- Configure the interpreter to handle input data and process the model output.

#### 3. Inference on Arduino Nano:

- Develop Arduino code to perform live speech classification using the deployed TensorFlow Lite model.
- Set up the input tensor arena and invoke the interpreter to obtain real-time predictions.

#### 4. Model Validation on Arduino:

- Validate the accuracy and performance of the deployed TFLite model on Arduino Nano using a test set.
- Ensure that the model meets the memory constraints of the Arduino Nano 33 BLE.

### Running Inferencing in Embedded System (Arduino Nano 33 BLE):

#### 1. Live Voice Recording:

- Record live audio using the Arduino Nano 33 BLE's microphone.

#### 2. Pre-processing on Arduino:

- Implement pre-processing steps on Arduino to extract voice features from the live audio input.
- Prepare the input tensor to feed into the TensorFlow Lite model.

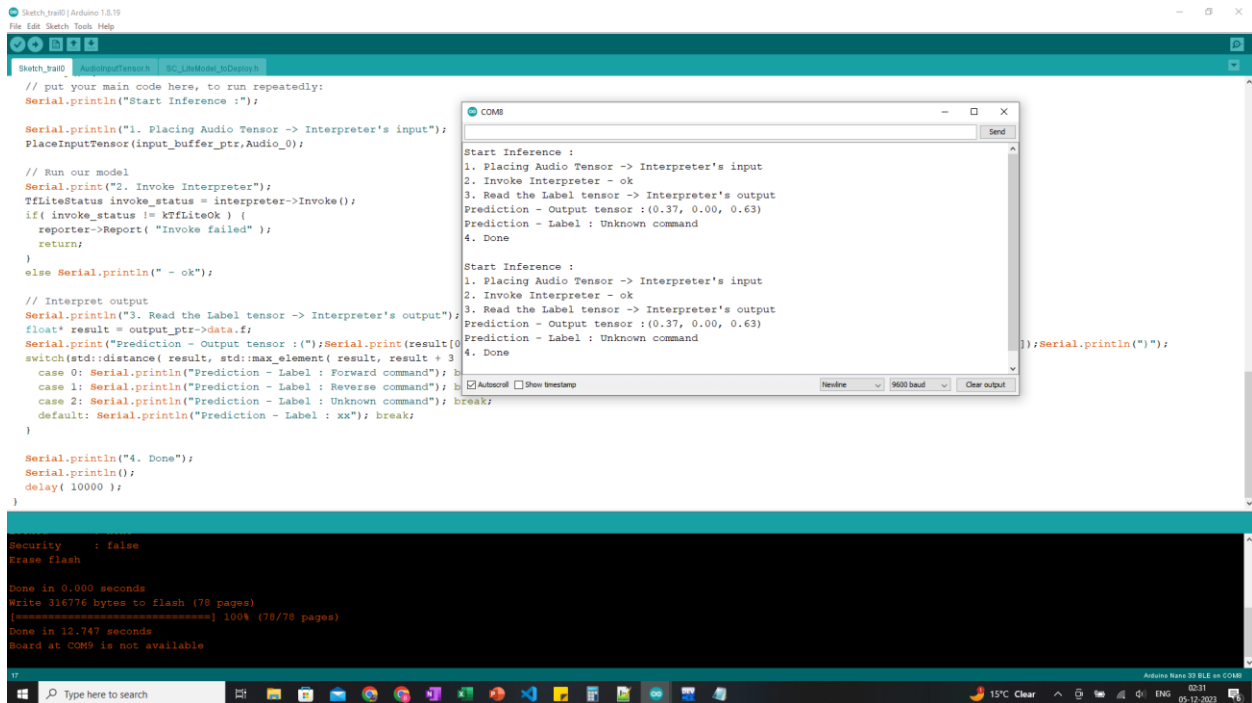
#### 3. Model Inference:

- Use the deployed TensorFlow Lite model to perform real-time inferencing on the Arduino Nano 33 BLE.
- Interpret the model output to classify the speech into predefined commands (forward, reverse, unknown, etc.).

#### 4. Output Verification:

- Check the output of the inference against expected results.
- Ensure that the Arduino Nano 33 BLE can handle the inferencing process efficiently.

Inference in Arduino -



```
Sketch_vault | Arduino Nano 33 BLE | File Edit Sketch Tools Help
// put your main code here, to run repeatedly:
Serial.println("Start Inference :");

Serial.println("1. Placing Audio Tensor -> Interpreter's input");
PlaceInputTensor(input_buffer_ptr, Audio_0);

// Run our model
Serial.println("2. Invoke Interpreter");
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  reporter->Report("Invoke failed");
  return;
} else Serial.println(" - ok");

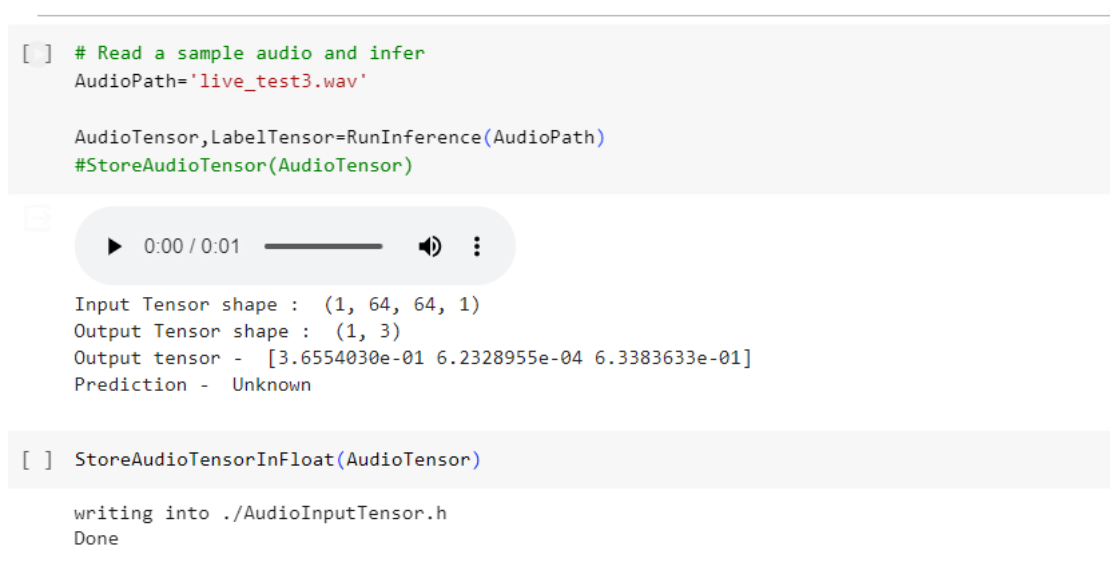
// Interpret output
Serial.println("3. Read the Label tensor -> Interpreter's output");
float* result = output_ptr->data.f;
Serial.print("Prediction - Output tensor : ("); Serial.print(result[0]);
switch(std::distance(result, std::max_element(result, result + 3))
case 0: Serial.println("Prediction - Label : Forward command"); break;
case 1: Serial.println("Prediction - Label : Reverse command"); break;
case 2: Serial.println("Prediction - Label : Unknown command"); break;
default: Serial.println("Prediction - Label : xx"); break;
}

Serial.println("4. Done");
Serial.println();
delay(10000);
}

Security : false
Erase flash

Done in 0.000 seconds
Write 316776 bytes to flash (78 pages)
[=====] 100% (78/78 pages)
Done in 12.747 seconds
Board at COM9 is not available
Arduino Nano 33 BLE on COM9
```

Inferencing in Google colab -



```
[ ] # Read a sample audio and infer
AudioPath='live_test3.wav'

AudioTensor,LabelTensor=RunInference(AudioPath)
#StoreAudioTensor(AudioTensor)

▶ 0:00 / 0:01

Input Tensor shape : (1, 64, 64, 1)
Output Tensor shape : (1, 3)
Output tensor - [3.6554030e-01 6.2328955e-04 6.3383633e-01]
Prediction - Unknown

[ ] StoreAudioTensorInFloat(AudioTensor)

writing into ./AudioInputTensor.h
Done
```

Both the output tensors are matching and the label is derived from the argmax correctly.

## Quantization and Memory Considerations:

### 1. Quantization for Embedded Deployment:

- Employ quantization techniques to optimize the model for deployment on resource-constrained devices.
- Utilize the TensorFlow Lite Converter to convert the model with quantization parameters suitable for Arduino Nano.

### 2. Memory Management:

- Evaluate the memory usage of the TensorFlow Lite model on Arduino Nano 33 BLE.
- Optimize the model to fit within the available memory constraints of the microcontroller.

## Edge Impulse Deployment (Optional):

### 1. Alternative Deployment Platform:

- Explore alternative deployment platforms like Edge Impulse for embedded systems.
- Assess the possibility of deploying the speech classification model through Edge Impulse.

### 2. Model Training and Testing:

- Utilize Edge Impulse for model training and testing with your provided dataset.
- Investigate the performance and accuracy of the model trained through Edge Impulse.

### 3. Binary Generation:

- Generate a binary from Edge Impulse suitable for deployment on Arduino Nano 33 BLE.
- Evaluate the ease of integration and deployment using Edge Impulse.

The screenshot displays the Edge Impulse web interface. The left sidebar contains navigation links: Dashboard, Devices, Data acquisition, Impulse design, EON Tuner, Retrain model, Live classification, Model testing, Performance calibration, Versioning, and Deployment. The main content area shows the 'SELECTED DEPLOYMENT' as 'Arduino Nano 33 BLE Sense'. Below this, 'MODEL OPTIMIZATIONS' are listed, including 'Enable EON™ Compiler'. Two tables compare 'Quantized (int8)' and 'Unoptimized (float32)' models. The 'Quantized' table shows a total latency of 299 ms and 12.0K RAM usage. The 'Unoptimized' table shows a total latency of 401 ms and 13.0K RAM usage. A 'Run model testing' button is present. On the right, a 'Build output' window shows a warning about incompatible architectures and a 'building model...' progress bar at the bottom.

	LATENCY	MPCC	CLASSIFIER	TOTAL
Quantized (int8)	254 ms	5 ms	299 ms	299 ms
RAM	12.0K	2.0K	12.0K	12.0K
FLASH	-	31.0K	-	-
ACCURACY	-	-	-	-

	LATENCY	MPCC	CLASSIFIER	TOTAL
Unoptimized (float32)	254 ms	107 ms	401 ms	401 ms
RAM	13.0K	3.0K	13.0K	13.0K
FLASH	-	27.0K	-	-
ACCURACY	-	-	-	88.96%

## Conclusion:

The project involved training various machine learning models, such as Artificial Neural Networks (ANN), 2D Convolutional Neural Networks (CNN), and 1D CNN, on a specific dataset for speech command recognition. The models were carefully optimized to achieve high accuracy during the training phase.

To make these models suitable for deployment on resource-constrained embedded systems like Arduino Nano 33 BLE, optimization techniques like quantization and pruning were applied. TensorFlow Lite format was used to ensure compatibility with the targeted embedded system.

The successful deployment of these models onto Arduino Nano 33 BLE showcased the feasibility of running machine learning models on edge devices. Real-time inferencing was implemented on Arduino for live speech classification, allowing users to give voice commands in real-time.

The project also explored Edge Impulse as an alternative deployment platform, demonstrating adaptability to different environments. Rigorous testing was conducted to validate the accuracy and robustness of the deployed model in recognizing various voice commands.

Quantization was employed to optimize the model for size and memory constraints, considering the limited resources of Arduino Nano 33 BLE. The project also considered resource constraints, with a focus on continuous optimization for improved efficiency.

In conclusion, the project achieved success in implementing a real-time voice command recognition system on an embedded system, laying the groundwork for future improvements and applications in edge computing and IoT devices. The versatility and adaptability of the solution were highlighted through successful integration with Arduino Nano 33 BLE and exploration of alternative deployment platforms.