[AT60005] Embedded Machine Learning Assignment 2

-By Ayan Chakraborty 18EC10075

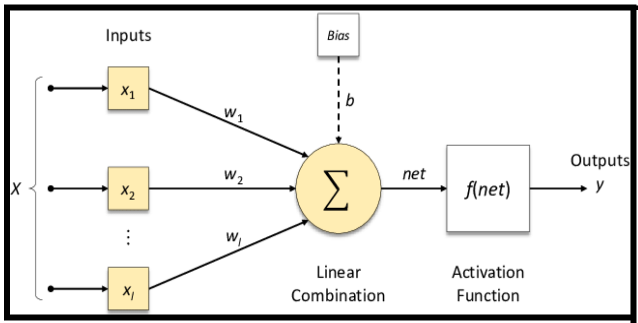
Problem Statement 1 Objectives:

1) Train a Neural Network model that can take an input value x, and predict its sine, y
2) Run the trained Neural Network on an embedded hardware device

Building and Training the Neural Network Model:

Brief theory of Neural Networks

Neural Networks are modern Deep Learning Algorithms, mainly used for supervised learning. A Perceptron is the most basic unit of a Neural Network, which does a MAC operation on an input vector, and applies an activation function to the sum, to generate the final output. Neural Networks are made up of multiple layers, each layer containing multiple perceptrons. Each layer takes it's input from the previous layer. Hence, we have an input layer which takes the actual input values, then we have hidden layers, which take their input values from their respective previous layer, and finally an output layer, which gives us our desired output. The layer weights are learned by the neural network during training, using the back propagation algorithm. They are diagrammatically shown below.



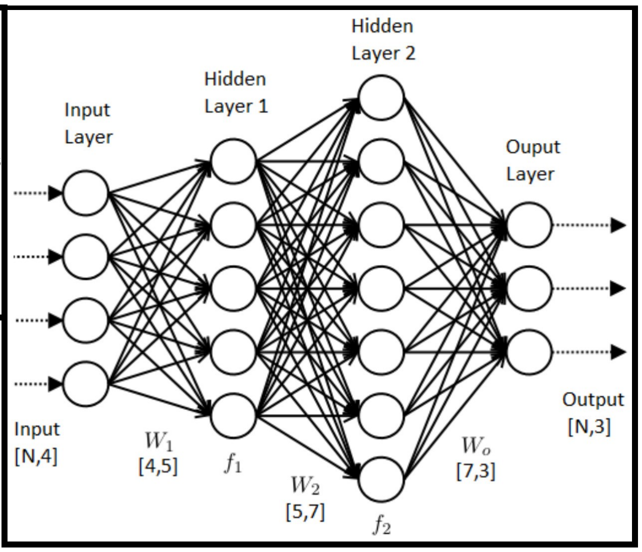


Fig. (1). Diagram showing a simple perceptron and a neural network with 1 input layer, 2 hidden layers, and 1 output layer

Generation of dataset

Our neural network needs to learn to approximate the sine function. Hence, we need a training dataset of numbers and their corresponding sine values. We generate this training dataset ourselves using python code. We generate 1,000 values that represent random points along a sine wave, and shuffle them in a random order. Sine wave is periodic with a period of 2π, hence it is sufficient to generate points in the range [0, 2π]. We then calculate the sine for each of these values. This will be our dataset. Randomizing the data order is important because the training process used in deep learning depends on data being fed to it in a truly random order. The code is shown below:

```
num_samples = 1000
random_seed = 1337
np.random.seed(random_seed)
tf.random.set_seed(random_seed)
x_values = np.random.uniform(low = 0, high = 2*np.pi, size = num_samples)
np.random.shuffle(x_values)
y_values = np.sin(x_values)
```

Currently, the data points are 100% accurate. In real life, we may not get such accurate data, hence, we add some noise sampled from a normal distribution to the sine values. This is much more reflective of a real-world situation, in which data is generally quite messy. The final generated dataset is shown below after the code snippet in Fig.(2).

```
y_values += 0.1*np.random.randn(*y_values.shape)
```

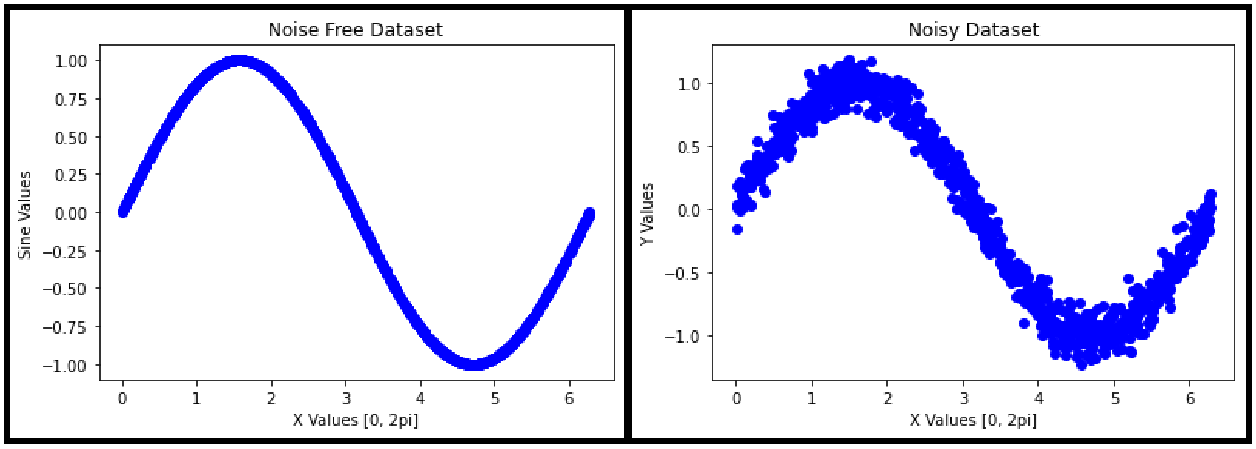


Fig. (2). Plot showing the initial noise free dataset, and the noisy dataset

Partitioning the dataset

We have a dataset of 1000 numbers and their corresponding values. We need some portion of this dataset to train the model, some portion to validate the model during training, and finally use the remaining data to test the performance of the model after training. The sets are called training dataset, validation dataset and test dataset respectively. We have partitioned our dataset in a ratio of 60:20:20.

```
train_split = int(0.6*num_samples)
validation_split = int(0.2*num_samples + train_split)
x_train, x_validate, x_test = np.split(x_values, [train_split,
validation_split])
y_train, y_validate, y_test = np.split(y_values, [train_split,
validation_split])
```

The partitioned data is plotted in Fig. (3). below, using different colors for each partition:

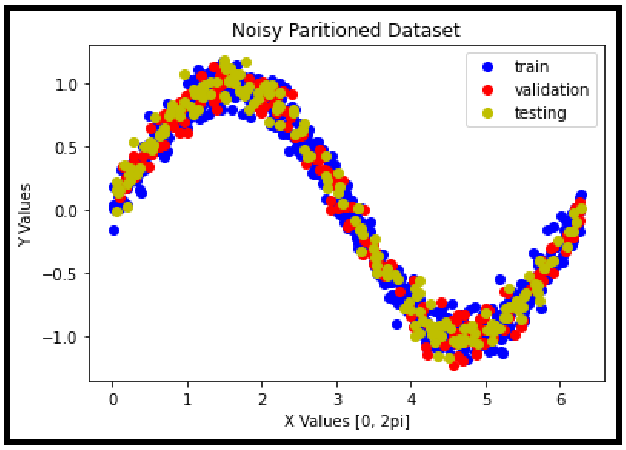


Fig. (3). Plot showing the final partitioned dataset, each of them resembling a sine wave

Creating and training a Neural Network Model

We have used the Tensorflow and Keras Deep Learning Libraries to build our models. The input layer will have a single neuron, as we have a single input value. The output layer will also have a single neuron, because we need a single numeric output. The output layer will not need an activation function, because this is a regression problem. We can vary the number of hidden layers, the number of units in each layer, and the activation function used to build different models. We have built and tested 5 different models to observe their variations. These models are described below:

# Hidden layers	# Neurons in each layer	Activation function used
0	NA	NA
1	16	ReLU
1	16	TanH
1	32	ReLU
2	16	ReLU

We use Root Mean Squared Propagation, or RMSProp, an extension of gradient descent for each parameter. This results in taking smaller steps, when we are closer to the target, improving the convergence of the network. We use Mean Square Error as our loss function, because this is a regression problem, and we want our predictions to exactly match our desired output. The number of iterations for which the neural network will train is called epochs, and the number of data points fed to the neural network in each epoch is called the batch size. We have used 1000 epochs with a batch size of 16 for all models.

The code to build a sample 2 layer network is given below. Model is initialized with an initial sequential layer for the input layer. Subsequent layers are added to the model to get as many layers as we want, each layer having its own specific number of neurons, and activation functions. Dense layers imply that the output of 1 neuron in a particular layer is connected to the input of all neurons in the next layers. The first layer must have it's input layer specified as (1,), because we have a scalar input, and that is required by syntax to build any model. We finally compile the model, using our optimizer and loss metrics.

```
new_model = tf.keras.Sequential()
new_model.add(layers.Dense(16, activation = "relu", input_shape = (1,)))
new_model.add(layers.Dense(16, activation = "relu"))
new_model.add(layers.Dense(1))
new_model.compile(optimizer = "rmsprop", loss = "mse", metrics = ["mse"])
```

Results of training the above 5 models on our dataset

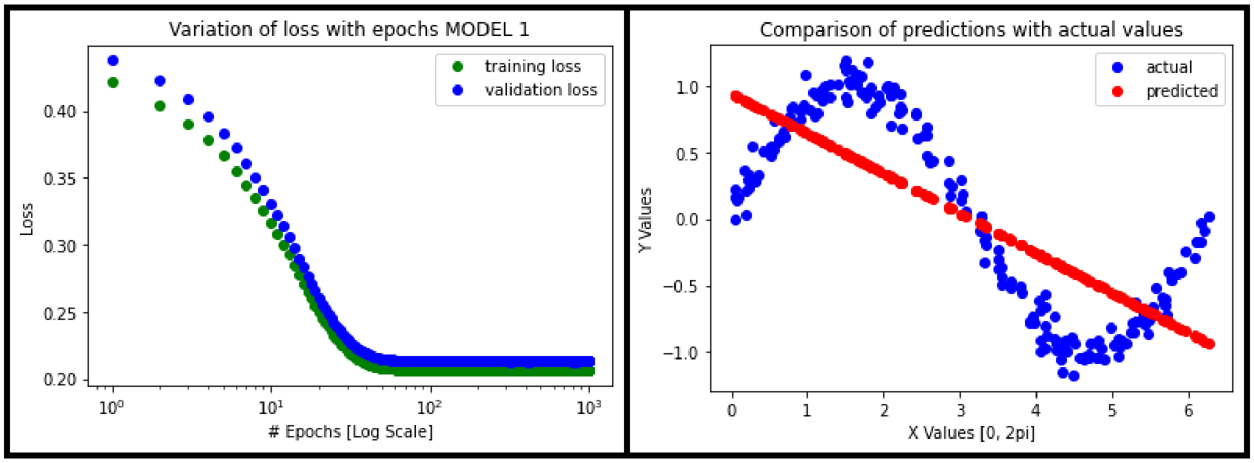


Fig. (4). Variation of Training Loss and performance evaluation on test data for MODEL 1

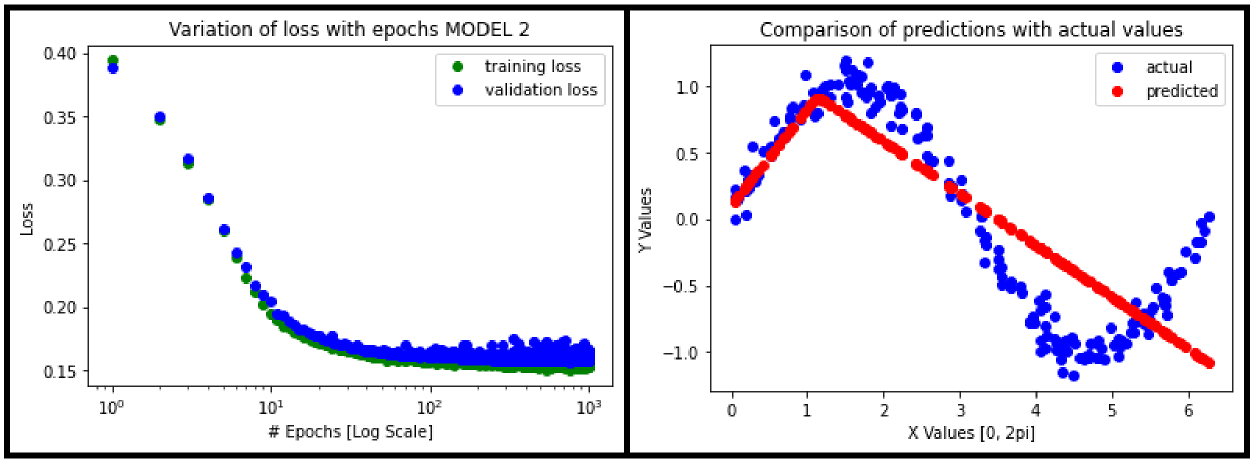


Fig. (5). Variation of Training Loss and performance evaluation on test data for MODEL 2

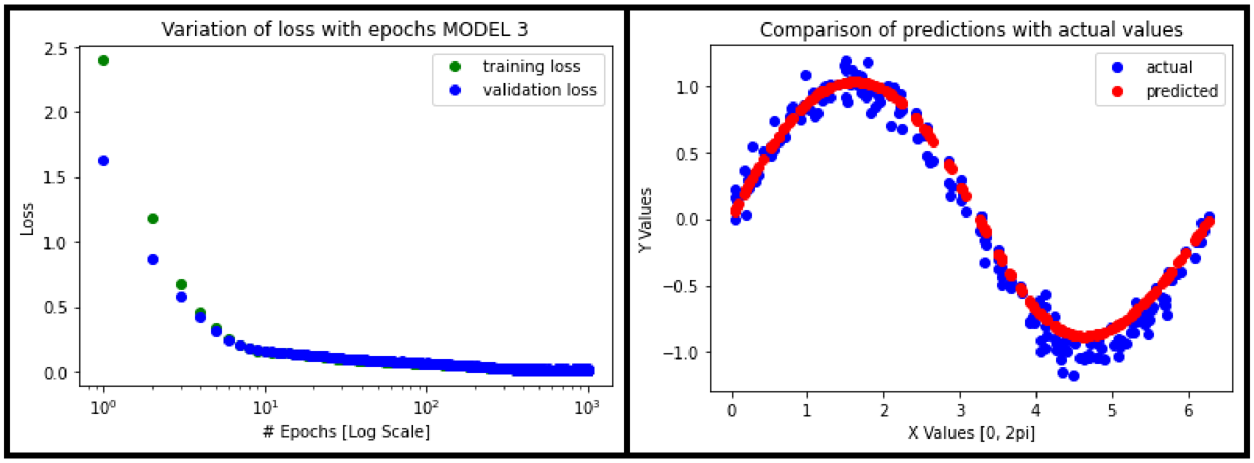


Fig. (6). Variation of Training Loss and Performance evaluation on test data for MODEL 3

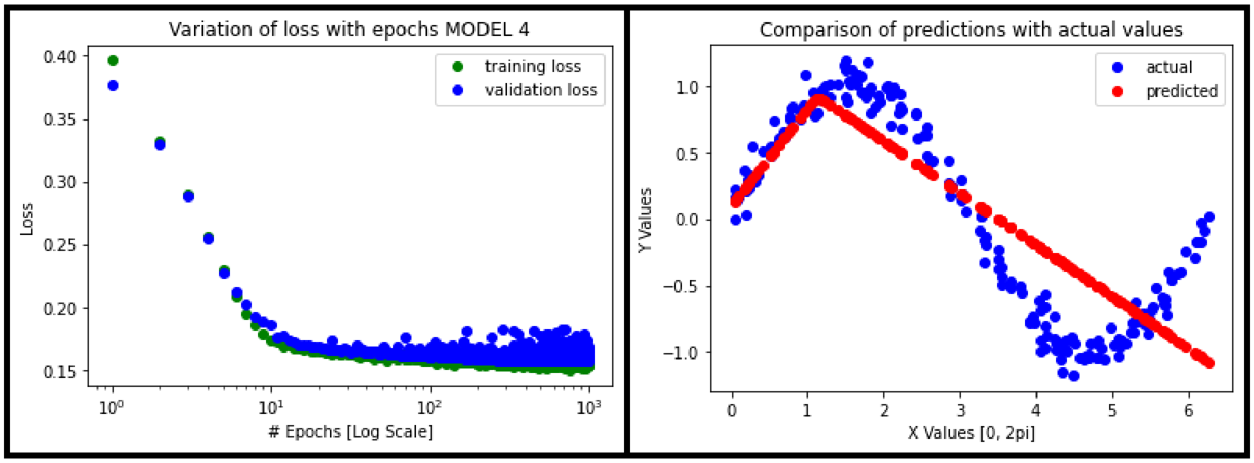


Fig. (7). Variation of Training Loss and performance evaluation on test data for MODEL 4

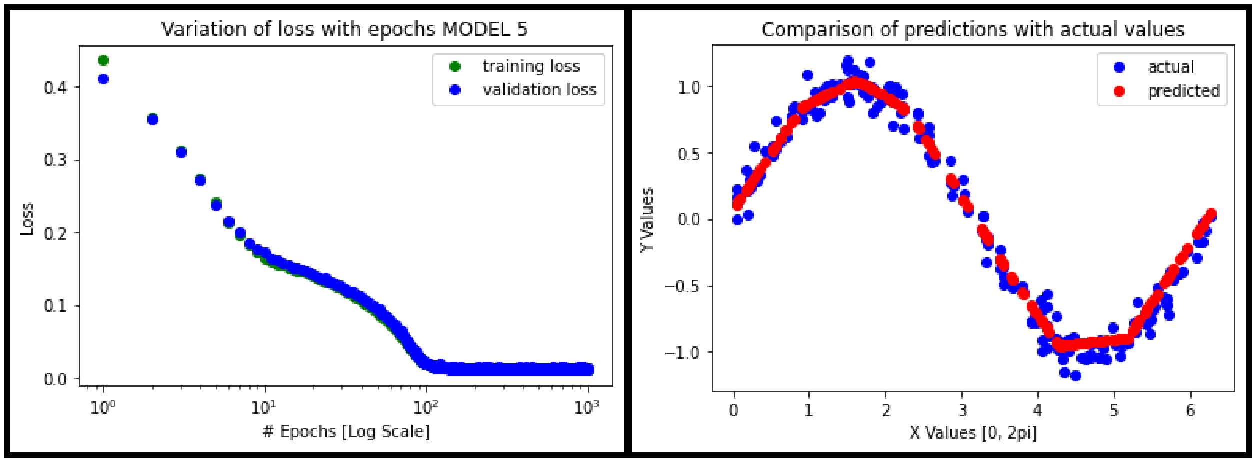


Fig. (8). Variation of Training Loss and performance evaluation on test data for MODEL 5

We can clearly see Model 3 and Model 5 perform best. The results are analysed next.

Analysis of Results and Conclusions

1) Model 1 has no hidden layer, hence, it is a simple linear regression model. There are no activation functions to provide non linearity, hence, the resulting model is a linear function, and does not approximate the nonlinear sine function well at all.

2) Next, Model 2 has 1 hidden layer with 16 neurons, using the ReLU activation function. This model has learned a piecewise linear model, that still does not approximate the sine function well, but performs better than linear regression, and has a lower training loss than Model 1.

3) However if we change the activation function to the TanH function, keeping everything else the same, which results in Model 3, then the resulting model approximates the sine function very well, and achieves close to 0 training error. Potential reasons for this can be due to the fact that the TanH function is also a trigonometric function, hence, it models the non linearity required by sine function much better than the TanH function, which is just piecewise linear.

4) Next, we observe the effects of increasing the number of neurons in Model 2, to see if it leads to any improvement, which results in Model 4. Model 4 performs similarly to Model 2, with no observable improvement. This leads to the conclusion that 1 hidden layer is insufficient to model the sine function, if we use the ReLU activation function.

5) The final Model 5 has 1 additional hidden layer of 16, using the ReLU activation function, and we observe that it performs much better than a single layer model, with a much lower training loss as well. The extra hidden layer allows modeling more complex nonlinear functions, which is the cause of the improved performance.

However, Model 3 still has a smoother and a better approximation of the sine function compared to Model 5. Model 3 also needs lesser space than Model 5 as it has fewer weights. So, Model 3 is the best choice to run on the embedded device.

Converting the model to an appropriate format

We cannot directly put the trained model on an embedded device. We have to convert it into a special space efficient file format, because space is limited on those devices. We use the TensorFlow Lite Converter. This converts TensorFlow models into the correct format. A simple conversion without any conversions is achieved as follows:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model_3)
tflite_model = converter.convert()
open("sine_model.tflite", "wb").write(tflite_model)
```

The TensorFlow Lite Converter can apply more optimizations to the trained model to reduce model size and make it run faster on small devices. One of these optimizations is quantization. By default, the weights and biases in a model are stored as 32-bit floating-point numbers so that high-precision calculations can occur during training. Quantization allows us to reduce the precision of these numbers so that they fit into 8-bit integers—a four times reduction in size. Even better, because it's easier for a CPU to perform math with integers than with floats, a quantized model will run faster. The process is very similar to the above. Additionally, we just need a representative dataset, so that the converter can figure out which conversions out of the default set to apply.

```
converter = tf.lite.TFLiteConverter.from_keras_model(new_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
def representative_dataset_generator():
  for value in x_test:
      yield [np.array(value, dtype=np.float32, ndmin=2)]
converter.representative_dataset = representative_dataset_generator
tflite_model = converter.convert()
open("sine_model_quantized.tflite", "wb").write(tflite_model)
```

We run inference using the TensorFlow Lite Interpreter. We have to first instantiate an interpreter object, allocate memory for the tensors, set the input data, invoke the model, and finally read the output value. The process is shown below:

```
sine_model_quantized = tf.lite.Interpreter('sine_model_quantized.tflite')
sine_model_quantized.allocate_tensors()
sine_model_quantized_input_index =
sine_model_quantized.get_input_details()[0]["index"]
sine_model_quantized_output_index =
sine_model_quantized.get_output_details()[0]["index"]
x_value_tensor = tf.convert_to_tensor([[x]], dtype=np.float32)
sine_model_quantized.set_tensor(sine_model_quantized_input_index,
x_value_tensor)
sine_model_quantized.invoke()
y = sine_model_quantized.get_tensor(sine_model_quantized_output_index)[0])
```

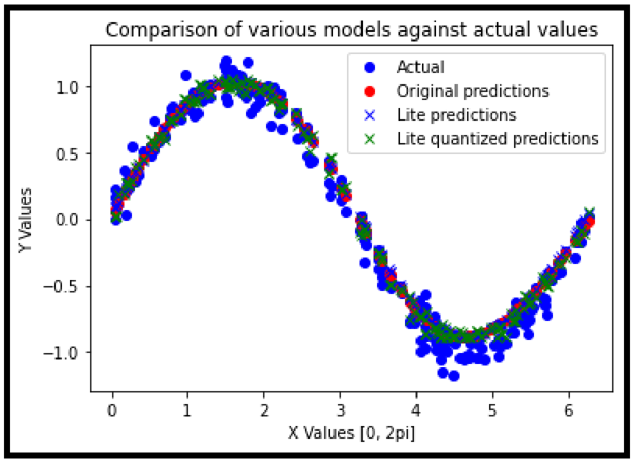


Fig. (9). Performance of converted models on test dataset using the above approach.

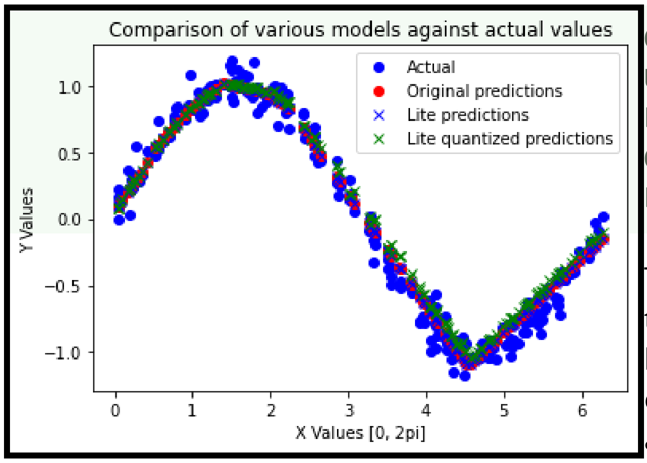
There is no observable drop in performance. Both models perform as well as the original model, so we can go ahead and use our optimised converted model.

We also measure the reduction in sizes of the two models compared to the original tensorflow model. The results are shown in the next section.

```
Original Model size is 4096 bytes
Unoptimised converted model is 1508 bytes
Optimised converted model is 1992 bytes
```

Both converted models are significantly smaller (=0.25x) than the actual model. But surprisingly, the optimized model is slightly bigger than the unoptimised model. This may be due to additional overheads involved in optimizing the TanH activation function. Regardless of size, the optimized version of model 5 will run faster on the embedded device compared to simply using the unoptimized version of model 3.

To see if there actually is a reduction in size for the optimized converted model, if a different activation function is used, we compare the sizes and performance of optimized and unoptimized converted models for the next best performing Model 5. The results are shown below:



```
Original Model size is 4096 bytes
Unoptimised converted model is 2912
bytes
Optimised converted model is 2712
bytes
```

The optimized converted model 5 is smaller than the unoptimised converted model 5, but both of them are still larger than the optimised converted model 3. Hence, we go ahead and use optimized converted model 3 for our application. Although in both cases, the reduction in size may not be so useful, because the original model itself is not too big, but for larger models used for more complex applications such as Object Detection and Recognition, which have millions of weights, the reduction in size becomes much more significant and useful.

Saving the converted model to a C file

So far, we've been using TensorFlow Lite's Python API. This means that we've been able to use the Interpreter constructor to load our model files from disk. However, most microcontrollers don't have a filesystem, and even if they did, the extra code required to load a model from disk would be wasteful given our limited space. Instead, as an elegant solution, we provide the model in a C source file that can be included in our binary and loaded directly into memory. This is done using the xxd command through the terminal.

```
!apt-get -qq install xxd
!xxd -i sine_model_quantized.tflite > sine_model_quantized.cc
!cat sine_model_quantized.cc
```

Running the model on an embedded device

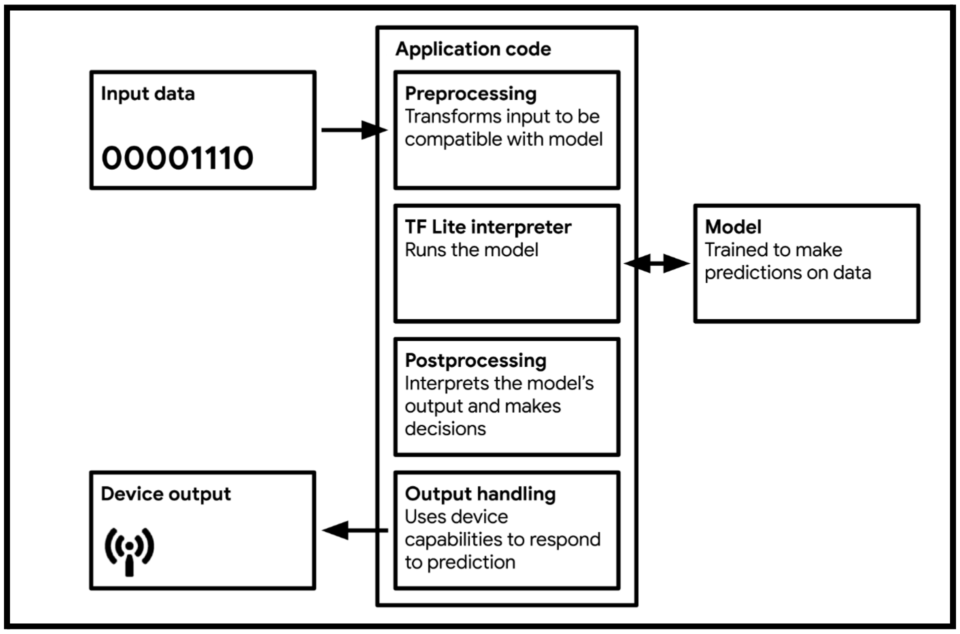

Application code
Input data
00001110
Preprocessing
Transforms input to be compatible with model
TF Lite interpreter
Runs the model
Model
Trained to make predictions on data
Postprocessing
Interprets the model's output and makes decisions
Device output
Output handling
Uses device capabilities to respond to prediction

Fig. (10). The inference pipeline of a Basic TinyML Application Architecture

We use the pre-written hello_world_test.cc file to explain the above architecture. We have slightly modified the test to run inference using our trained model on some manually chosen values as well as any random values from the terminal, to get a working example. It will also help us know if there are any problems with our model, or our methodology. Once we have verified that the tests are working correctly on our host machine, we can run them on our embedded device. The following steps are required to run inference:

1) **Include dependencies:** Include all the necessary header files needed

```
#include <math.h>
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include
"tensorflow/lite/micro/examples/hello_world/hello_world_model_data.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/testing/micro_test.h"
#include "tensorflow/lite/schema/schema_generated.h"
```

2) **Setup the test:** Name our test, and use macros to wrap the rest of our code in the necessary apparatus for it to be executed by the TensorFlow Lite for Microcontrollers testing framework.

```
TF_LITE_MICRO_TESTS_BEGIN
TF_LITE_MICRO_TEST(LoadModelAndPerformInference) {
```

3) **Setup Data Logging:** The MicroErrorReporter class defined in micro_error_reporter.h. It provides a mechanism for logging debug information during inference.

```
tflite::MicroErrorReporter micro_error_reporter;
```

4) **Mapping our model:** We map our model to a usable data structure, that we can call from our program, and use it's values. We take our model data array (defined in the file sine_model_data.h) and pass it into a method named GetModel(). This method returns a Model pointer, which is assigned to a variable named model. It is a struct variable.

```
const tflite::Model* model = ::tflite::GetModel(g_hello_world_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
      TF_LITE_REPORT_ERROR(&micro_error_reporter,
                            "Model provided is schema version %d not equal "
                            "to supported version %d.\n",
                            model->version(), TFLITE_SCHEMA_VERSION);
  }
```

5) **Creating an AllOpsResolver:** We create an instance of AllOpsResolver. This class, defined in all_ops_resolver.h, is what allows the TensorFlow Lite for Microcontrollers interpreter to access mathematical operations needed to run inputs through the model.

```
  tflite::AllOpsResolver resolver;
```

6) **Defining a Tensor Area:** We allocate an area of working memory that our model will need while it runs. This area of memory will be used to store the model's inputs, outputs, and intermediate tensors. It is known as the Tensor Arena.

```
  constexpr int kTensorArenaSize = 2000;
  uint8_t tensor_arena[kTensorArenaSize];
```

7) **Creating an interpreter:** We declare a MicroInterpreter named interpreter. This class is the heart of TensorFlow Lite for Microcontrollers: a piece of code that will execute our model on the data we provide.

```
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
kTensorArenaSize, &micro_error_reporter);
TF_LITE_MICRO_EXPECT_EQ(interpreter.AllocateTensors(), kTfLiteOk);
```

8) **Inspecting our model and the Input Tensor:** We write our input data to the model's input tensor, and grab the pointer to the input tensor. We verify our model dimensions and input tensor shapes are correct using assertions, and that the input tensor pointer is not NULL. We take the quantization parameters from the model and quantize the input from a floating point number to an integer. Later the quantized output will be converted back to a floating point number.

```
TfLiteTensor* input = interpreter.input(0);

  // Make sure the input has the properties we expect
  TF_LITE_MICRO_EXPECT_NE(nullptr, input);
  // The property "dims" tells us the tensor's shape. It has one element
for
  // each dimension. Our input is a 2D tensor containing 1 element, so
"dims"
  // should have size 2.
  TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);
  // The value of each element gives the length of the corresponding
tensor.
  // We should expect two single element tensors (one is contained within
the
  // other).
  TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
  TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
  // The input is an 8 bit integer value
  TF_LITE_MICRO_EXPECT_EQ(kTfLiteInt8, input->type);

  // Get the input quantization parameters
  float input_scale = input->params.scale;
  int input_zero_point = input->params.zero_point;

  // Quantize the input from floating-point to integer
  int8_t x_quantized = x / input_scale + input_zero_point;
```

9) **Running inference on input:** To run inference, we add a value to our input tensor and then instruct the interpreter to invoke the model. The model consists of a graph of mathematical operations which the interpreter executes to transform the input data into an output. This output is stored in the model's output tensors.

```
  input->data.int8[0] = x_quantized;
  // Run the model and check that it succeeds
  TfLiteStatus invoke_status = interpreter.Invoke();
  TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);
```

10) **Reading the output:** The model's output is accessed through a TfLiteTensor, and getting a pointer to it. The output is, like the input, a integer scalar value nestled inside a 2D tensor, that we need to convert back to a floating point number, using the quantization parameters of the model. We check to confirm the predicted value is approximately equal to actual value, using the TF_LITE_MICRO_EXPECT_NEAR assertion.

```
  // Obtain a pointer to the output tensor and make sure it has the
  // properties we expect. It should be the same as the input tensor.
  TfLiteTensor* output = interpreter.output(0);
  TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);
  TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[0]);
  TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[1]);
  TF_LITE_MICRO_EXPECT_EQ(kTfLiteInt8, output->type);

  // Get the output quantization parameters
  float output_scale = output->params.scale;
  int output_zero_point = output->params.zero_point;

  // Obtain the quantized output from model's output tensor
  int8_t y_pred_quantized = output->data.int8[0];
  // Dequantize the output from integer to floating-point
  float y_pred = (y_pred_quantized - output_zero_point) * output_scale;

  // Check if the output is within a small range of the expected output
  float epsilon = 0.05f;
  TF_LITE_MICRO_EXPECT_NEAR(y_true, y_pred, epsilon);
```

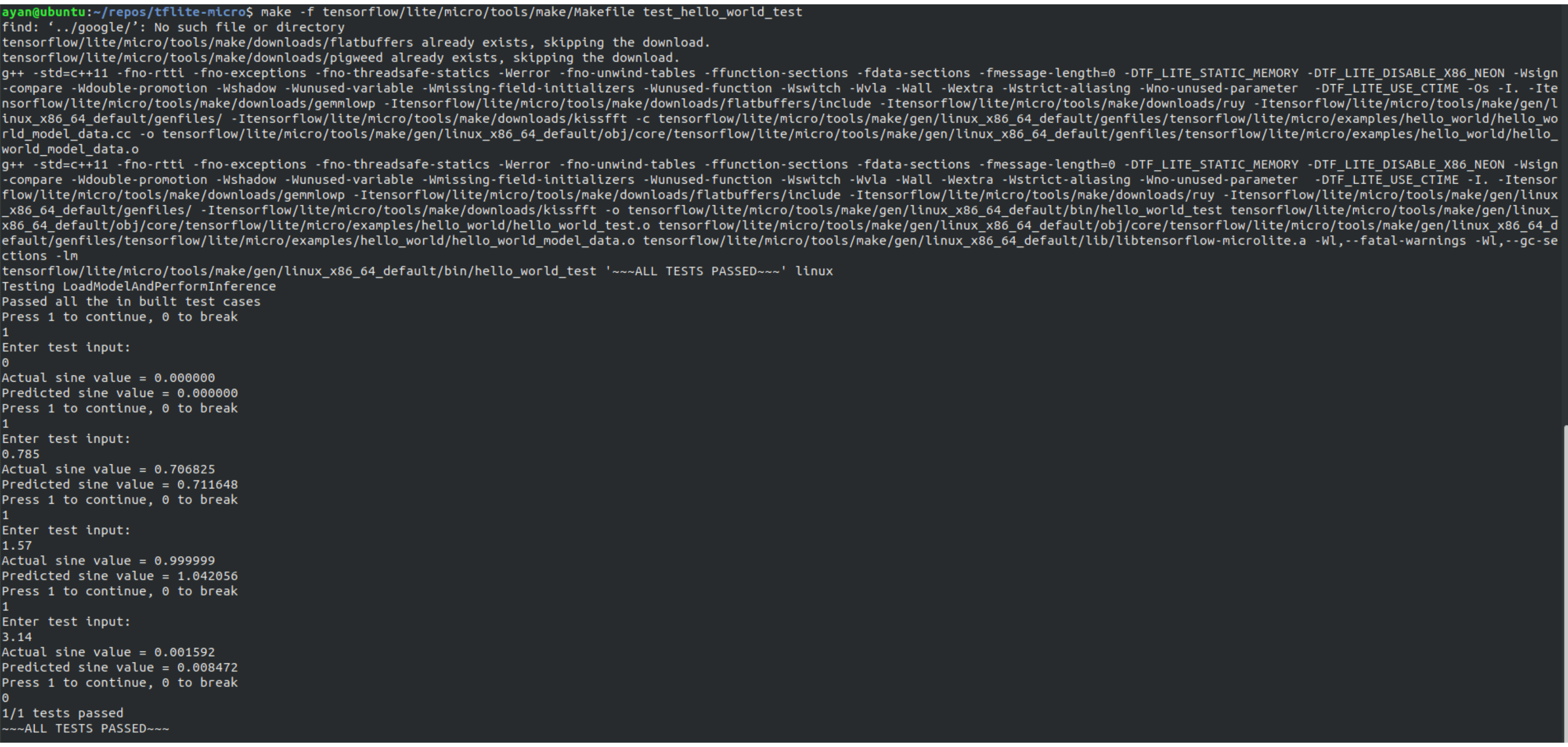
11) **Custom code:** After running inference on some hard coded input values, the custom extra code accepts a value from the terminal and runs the same inference process as above for as many times as the user wants, and displays the calculated result.

```
int flag;
  while(1){
      printf("Press 1 to continue, 0 to break\n");
      if(scanf("%d", &flag) == 1){
      if (flag == 0)
      break;
      printf("Enter test input:\n");
      if(scanf("%f", &x) == 1){
      y_true = sin(x);
      input->data.int8[0] = x / input_scale + input_zero_point;
      interpreter.Invoke();
      y_pred = (output->data.int8[0] - output_zero_point) * output_scale;
      printf("Actual sine value = %d.%.6d\n", (int)y_true,
(int)((y_true-(int)y_true)*1000000));
      printf("Predicted sine value = %d.%.6d\n", (int)y_pred,
(int)((y_pred-(int)y_pred)*1000000));
      }
      }}
```

Running the test script

We build the application using the given Makefile using the make command from the terminal as shown below, which results in the running of the test script.

```
make -f tensorflow/lite/micro/tools/make/Makefile test_hello_world_test
```



```
ayan@ubuntu:~/repos/tflite-micro$ make -f tensorflow/lite/micro/tools/make/Makefile test_hello_world_test
find: ‘../google/’: No such file or directory
tensorflow/lite/micro/tools/make/downloads/flatbuffers already exists, skipping the download.
tensorflow/lite/micro/tools/make/downloads/pigweed already exists, skipping the download.
g++ -std=c++11 -fno-rtti -fno-exceptions -fno-threadsafe-statics -Werror -fno-unwind-tables -ffunction-sections -fdata-sections -fmessage-length=0 -DTF_LITE_STATIC_MEMORY -DTF_LITE_DISABLE_X86_NEON -Wsign
-compare -Wdouble-promotion -Wshadow -Wunused-variable -Wmissing-field-initializers -Wunused-function -Wswitch -Wvla -Wall -Wextra -Wstrict-aliasing -Wno-unused-parameter  -DTF_LITE_USE_CTIME -Os -I. -Ite
nsorflow/lite/micro/tools/make/downloads/gemmlowp -Itensorflow/lite/micro/tools/make/downloads/flatbuffers/include -Itensorflow/lite/micro/tools/make/downloads/ruy -Itensorflow/lite/micro/tools/make/gen/l
inux_x86_64_default/genfiles/ -Itensorflow/lite/micro/tools/make/downloads/kissfft -c tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/genfiles/tensorflow/lite/micro/examples/hello_world/hello_wo
rld_model_data.cc -o tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/obj/core/tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/genfiles/tensorflow/lite/micro/examples/hello_world/hello_
world_model_data.o
g++ -std=c++11 -fno-rtti -fno-exceptions -fno-threadsafe-statics -Werror -fno-unwind-tables -ffunction-sections -fdata-sections -fmessage-length=0 -DTF_LITE_STATIC_MEMORY -DTF_LITE_DISABLE_X86_NEON -Wsign
-compare -Wdouble-promotion -Wshadow -Wunused-variable -Wmissing-field-initializers -Wunused-function -Wswitch -Wvla -Wall -Wextra -Wstrict-aliasing -Wno-unused-parameter  -DTF_LITE_USE_CTIME -I. -Itensor
flow/lite/micro/tools/make/downloads/gemmlowp -Itensorflow/lite/micro/tools/make/downloads/flatbuffers/include -Itensorflow/lite/micro/tools/make/downloads/ruy -Itensorflow/lite/micro/tools/make/gen/linux
_x86_64_default/genfiles/ -Itensorflow/lite/micro/tools/make/downloads/kissfft -o tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/bin/hello_world_test tensorflow/lite/micro/tools/make/gen/linux_
x86_64_default/obj/core/tensorflow/lite/micro/examples/hello_world/hello_world_test.o tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/obj/core/tensorflow/lite/micro/tools/make/gen/linux_x86_64_d
efault/genfiles/tensorflow/lite/micro/examples/hello_world/hello_world_model_data.o tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/lib/libtensorflow-microlite.a -Wl,--fatal-warnings -Wl,--gc-se
ctions -lm
tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/bin/hello_world_test '~~~ALL TESTS PASSED~~~' linux
Testing LoadModelAndPerformInference
Passed all the in built test cases
Press 1 to continue, 0 to break
1
Enter test input:
0
Actual sine value = 0.000000
Predicted sine value = 0.000000
Press 1 to continue, 0 to break
1
Enter test input:
0.785
Actual sine value = 0.706825
Predicted sine value = 0.711648
Press 1 to continue, 0 to break
1
Enter test input:
1.57
Actual sine value = 0.999999
Predicted sine value = 1.042056
Press 1 to continue, 0 to break
1
Enter test input:
3.14
Actual sine value = 0.001592
Predicted sine value = 0.008472
Press 1 to continue, 0 to break
0
1/1 tests passed
~~~ALL TESTS PASSED~~~
```

It is tested on 0, pi/4, pi/2, and pi as shown above, and its predicted result is very close to the actual result as shown. All tests have passed without any errors.

Deploying the application to an embedded device

The application is already included an example inside the repository and works very similarly as the above test script. The only difference is that there are multiple files performing different functionalities of the above code.

The main.cc file is the starting point of the code, and calls a setup function, and then indefinitely keeps on calling the loop function, a very characteristic way of programming microcontrollers, as shown below:

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
// This is the default main used on systems that have the standard C entry
```

```
point.
int main(int argc, char* argv[]) {
  setup();
  while (true) {
      loop();
  }
}
```

The setup() and loop() functions are written inside the main_functions.cc file. The setup(function basically does the same initializations as described earlier to set up the inference process. The loop(), keeps on generating an input value each iteration, and performs the inference process described earlier. It passes it's predicted value to the HandleOutput() function, which takes the predicted and does something with it on the microcontroller.

We use it for the arduino, hence inside the hello_world example of the arduino library, it is defined in the arduino_output_handler.cpp file. It initializes the LED pin as an output pin, the first time it is called using the pinMode() function available in the Arduino library. Next time onwards, the default behaviour is glowing a LED with brightness proportional to the predicted output. Value of -1 corresponds to 0 brightness, and value of 1 corresponds to maximum brightness. Our desired behaviour is slightly different. We want the brightness to correspond to the absolute value of the prediction. analogWrite() is used which uses PWM to approximate analog signals. Hence, we slightly modify the arduino_output_handler.cpp file as follows:

Original Code:

```
int brightness = (int)(127.5f * (y_value + 1));
// Set the brightness of the LED.
analogWrite(led, brightness);
```

Modified Code:

```
int brightness = (int)(255 * abs(y_value));
// Set the brightness of the LED.
analogWrite(led, brightness);
```

Conclusion

We have looked at the entire end to end process of generating a dataset, training a neural network model, converting it for use in an embedded scenario, testing it on a dev machine and then finally running the application on an embedded board.

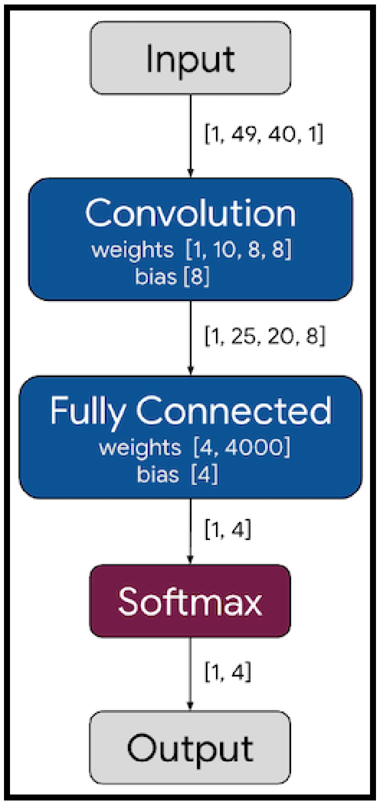
Problem Statement 2 Objectives:

1) Build and train a model to recognize the words “on” and “off” and is also capable of distinguishing between unknown words and silence or background noise.
2) Use the speech detection model to provide always-on wake-word detection using a tiny microcontroller.

Building and Training the Neural Network Model:

The entire process is similar to the previous part. The only addition needed here is the preprocessing of the speech data to convert it into a spectrogram, which contains the time variation of frequency through the entire speech signal. This is computed using the short time fast fourier transform. The spectrogram is a much more feature rich representation compared to the original raw speech data, hence, it is used as the input to the model. It is 2D in contrast to 1D input used in the previous part, the two axes being time and frequency.

Since this is a complex task, we are not creating a model of our own. We are using the predefined model architecture defined in the official github repository of Tensorflow: https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech/train . However, the default model can recognise only “yes” or “no”. Our required keywords are “on” and “off”. The datasets and everything are available, we just need to specify that these are the words we want the model to recognise, and retrain the model.



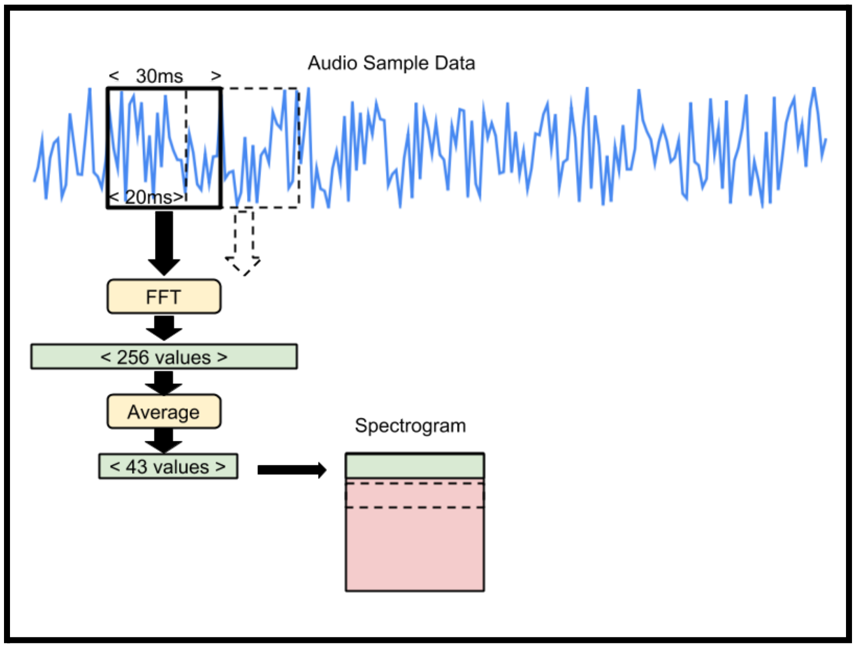


Fig. (11). Diagram showing the CNN Model architecture and an example of the spectrogram generation process.

Model Parameters

```
Training these words: on,off
Training steps in each stage: 12000,3000
Learning rate in each stage: 0.001,0.0001
Total number of training steps: 15000
```

We run 12,000 training loops in total, with a learning rate of 0.001 for the first 8,000, and 0.0001 for the final 3,000. A window stride of 20 is used to generate the spectrograms. The model takes about 3 hours to train on a GPU, and reaches a final accuracy of 88.5% on the test dataset as shown below (output of the training code snippet):

```
WARNING:tensorflow:Final test accuracy = 88.5% (N=1198)
W1025 15:50:43.162516 140074903672704 train.py:318] Final test accuracy =
88.5% (N=1198)
```

The training is done using the tensorflow/tensorflow/examples/speech_commands/train.py file. Checkpoints are saved under train/, which we “freeze” and convert into the final model using tensorflow/tensorflow/examples/speech_commands/freeze.py. The model is saved under saved_model/

Conversion of model into TensorFlow Lite Model

The basic process of conversion is similar to the previous part. The unoptimised converted model is generated by using the TFLiteConverter as shown below:

```
float_converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL)
float_tflite_model = float_converter.convert()
float_tflite_model_size = open(FLOAT_MODEL_TFLITE,
"wb").write(float_tflite_model)
```

Generating the optimized quantized converted model is a bit more complicated. Our aim is to use 8-bit integer inputs, and 8-bit integer outputs. The process of generating the representative dataset is a bit more complex but everything else is the same. We need a model_settings object and an audio_processor object. The model settings object prepares a dictionary of important model and data hyperparameters so that they can be saved and reused. The Audio Processor provides important functions related to audio preprocessing.

```
SAMPLE_RATE = 16000
CLIP_DURATION_MS = 1000
WINDOW_SIZE_MS = 30.0
FEATURE_BIN_COUNT = 40
BACKGROUND_FREQUENCY = 0.8
BACKGROUND_VOLUME_RANGE = 0.1
```

```
TIME_SHIFT_MS = 100.0
DATA_URL =
'https://storage.googleapis.com/download.tensorflow.org/data/speech_command
s_v0.02.tar.gz'
VALIDATION_PERCENTAGE = 10
TESTING_PERCENTAGE = 10
model_settings = models.prepare_model_settings(
      len(input_data.prepare_words_list(WANTED_WORDS.split(','))),
      SAMPLE_RATE, CLIP_DURATION_MS, WINDOW_SIZE_MS,
      WINDOW_STRIDE, FEATURE_BIN_COUNT, PREPROCESS)
audio_processor = input_data.AudioProcessor(
      DATA_URL, DATASET_DIR,
      SILENT_PERCENTAGE, UNKNOWN_PERCENTAGE,
      WANTED_WORDS.split(','), VALIDATION_PERCENTAGE,
      TESTING_PERCENTAGE, model_settings, LOGS_DIR)

converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL)
  converter.optimizations = [tf.lite.Optimize.DEFAULT]
  converter.inference_input_type = tf.int8
  converter.inference_output_type = tf.int8
  def representative_dataset_gen():
      for i in range(100):
      data, _ = audio_processor.get_data(1, i*1, model_settings,
                                          BACKGROUND_FREQUENCY,
                                          BACKGROUND_VOLUME_RANGE,
                                          TIME_SHIFT_MS,
                                          'testing',
                                          sess)
      flattened_data = np.array(data.flatten(),
dtype=np.float32).reshape(1, 1960)
      yield [flattened_data]
  converter.representative_dataset = representative_dataset_gen
  tflite_model = converter.convert()
  tflite_model_size = open(MODEL_TFLITE, "wb").write(tflite_model)
```

We then do a performance and size comparison between the optimised and unoptimized converted models

	Test Data Accuracy (1198 samples)	**Model Size (bytes)**
Unoptimised Model	88.564%	68164
Optimised Model	88.397%	18840

We observe that there is only a 0.2% drop in accuracy, while the model size is reduced by 3.62 times. This is a huge benefit. Apart from size, the optimised model will run faster as well. The 0.2% drop in accuracy is negligible compared to these benefits. It is finally converted into a C file using the xxd tool as shown below:

```
!apt-get update && apt-get -qq install xxd
# Convert to a C source file
!xxd -i {MODEL_TFLITE} > {MODEL_TFLITE_MICRO}
# Update variable names
REPLACE_TEXT = MODEL_TFLITE.replace('/', '_').replace('.', '_')
!sed -i 's/'{REPLACE_TEXT}'/g_model/g' {MODEL_TFLITE_MICRO}
!cat {MODEL_TFLITE_MICRO}
```

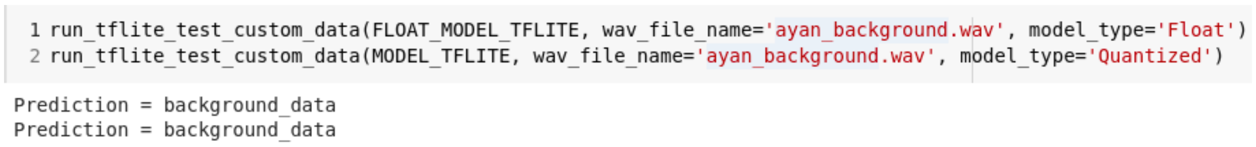
Testing model performance on personal voice data

One additional helper function is written to convert the prediction numeric value into a label, and the inference function is modified slightly to take wav files as input, instead of using the wav files from the dataset. The labels 0 and 1 are hardcoded to silence and background noise respectively inside the model, and the remaining labels are assigned in the order they were specified. Since, we had specified “on” first, “on” has the label 2.

```
def get_label_from_pred(value):
  if value == 0:
      return "silence"
  elif value == 1:
      return "background_data"
  elif value == 2:
      return "on"
  elif value == 3:
      return "off"
```

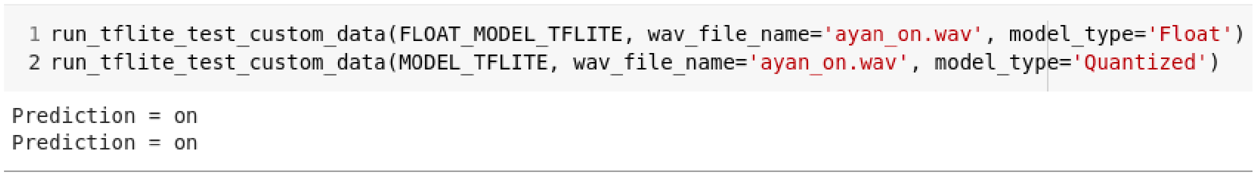
```
def run_tflite_test_custom_data(tflite_model_path, wav_file_name = None,
model_type="Float"):
  # Load test data
  np.random.seed(0) # set random seed for reproducible test results.
  with tf.compat.v1.Session() as sess:
      test_data = audio_processor.get_features_for_wav(wav_file_name,
model_settings, sess)
  test_data = np.reshape(test_data, [1, 1960])
  test_data = np.expand_dims(test_data, axis=1).astype(np.float32)

  # Initialize the interpreter
  interpreter = tf.lite.Interpreter(tflite_model_path)
  interpreter.allocate_tensors()

  input_details = interpreter.get_input_details()[0]
  output_details = interpreter.get_output_details()[0]

  # For quantized models, manually quantize the input data from float to
integer
  if model_type == "Quantized":
      input_scale, input_zero_point = input_details["quantization"]
      test_data = test_data / input_scale + input_zero_point
      test_data = test_data.astype(input_details["dtype"])

  correct_predictions = 0
  interpreter.set_tensor(input_details["index"], test_data[0])
  interpreter.invoke()
  output = interpreter.get_tensor(output_details["index"])[0]
  top_prediction = output.argmax()

  print("Prediction = {}".format(get_label_from_pred(top_prediction)))
```

The following results were observed for each class of audio:



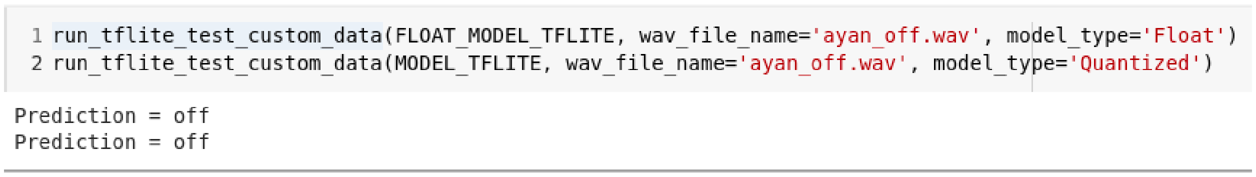
```
run_tflite_test_custom_data(FLOAT_MODEL_TFLITE, wav_file_name='ayan_background.wav', model_type='Float')
run_tflite_test_custom_data(MODEL_TFLITE, wav_file_name='ayan_background.wav', model_type='Quantized')
```

```
Prediction = background_data
Prediction = background_data
```



```
run_tflite_test_custom_data(FLOAT_MODEL_TFLITE, wav_file_name='ayan_on.wav', model_type='Float')
run_tflite_test_custom_data(MODEL_TFLITE, wav_file_name='ayan_on.wav', model_type='Quantized')
```

```
Prediction = on
Prediction = on
```



```
run_tflite_test_custom_data(FLOAT_MODEL_TFLITE, wav_file_name='ayan_off.wav', model_type='Float')
run_tflite_test_custom_data(MODEL_TFLITE, wav_file_name='ayan_off.wav', model_type='Quantized')
```

```
Prediction = off
Prediction = off
```

Both converted models gave the same correct prediction for each class of input waveform. It was not possible to get a fully silent voice clip, so that’s why it's not included in the results. We have confirmed that both models are working fine on indian accents as well. The process of building and training the model, and converting it into an appropriate format is done. Next, we need to deploy this as an application on a real embedded board.

Running the model on an embedded device

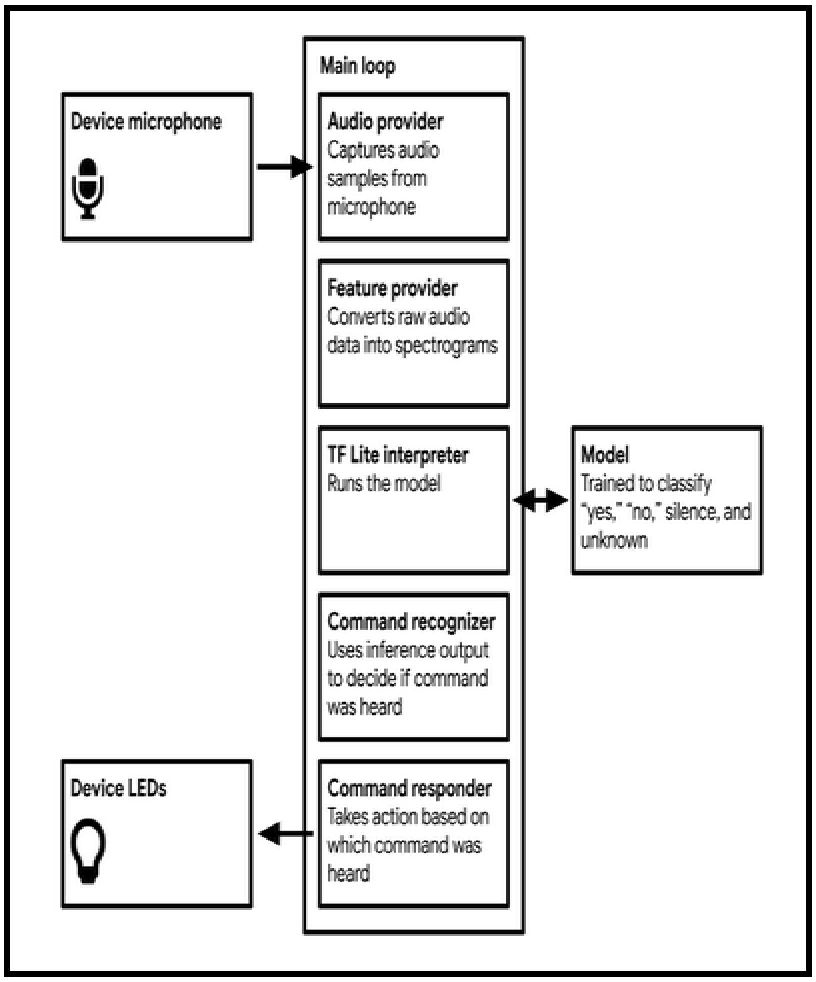

Main loop
Device microphone
Audio provider
Captures audio samples from microphone
Feature provider
Converts raw audio data into spectrograms
TF Lite interpreter
Runs the model
Model
Trained to classify "yes," "no," silence, and unknown
Command recognizer
Uses inference output to decide if command was heard
Device LEDs
Command responder
Takes action based on which command was heard

Fig. (11). The inference pipeline of the speech detection and classification model

It is similar to the pipeline in part 1, but there are some additional components here like the audio provider and the feature provider. They are explained briefly below:

1) **Main loop:** Like the "hello world" example, our application runs in a continuous loop. All of the subsequent processes are contained within it, and they execute continually, as fast as the microcontroller can run them, which is multiple times per second.

2) **Audio provider:** The audio provider captures raw audio data from the microphone. Because the methods for capturing audio vary from device to device, this component can be overridden and customized.

3) **Feature provider:** The feature provider converts raw audio data into the spectrogram format that our model requires. It does so on a rolling basis as part of the main loop, providing the interpreter with a sequence of overlapping one-second windows.

4) **TF Lite interpreter:** The interpreter runs the TensorFlow Lite model, transforming the

input spectrogram into a set of probabilities.

5) **Model:** The model is included as a data array and run by the interpreter. The array is located in tiny_conv_micro_features_model_data.cc.

6) **Command recognizer:** Because inference is run multiple times per second, the RecognizeCommands class aggregates the results and determines whether, on average, a known word was heard.

7) **Command responder:** If a command was heard, the command responder uses the device's output capabilities to let the user know. Depending on the device, this could mean flashing an LED or showing data on an LCD display. It can be overridden for different device types.

Running the Tests

Since, this is a very complex application with lots of components, a single test will not suffice like the previous example. The following tests are available, each testing a different component of the pipeline:

1) **micro_speech_test.cc:** Shows how to run inference on spectrogram data and interpret the results.

2) **audio_provider_test.cc:** Shows how to use the audio provider.

3) **feature_provider_mock_test.cc:** Shows how to use the feature provider, using a mock (fake) implementation of the audio provider to pass in fake data

4) **recognize_commands_test.cc:** Shows how to interpret the model's output to decide whether a command was found.

5) **command_responder_test.cc:** Shows how to call the command responder to trigger an output

As a short demonstration of the inference process, we go through the methodology of the micro_speech_test.cc file. This is similar to the inference process of part 1 as well, so we don't explain the similar code again but just briefly describe the internal structure:

1) **Define necessary operations:** A given model probably doesn't use all of the dozens of available operations. When deployed to a device, these unnecessary operations will take up valuable memory, so it's best if we include only those we need.

2) **Setup error logging and load our model:** After our model is loaded, we declare a MicroMutableOpResolver and use its method AddBuiltin() to add the ops we listed earlier.

3) **Setup the interpreter and the working memory:** In our "hello world" application we

allocated only 2 * 1,024 bytes for the tensor_arena, given that the model was so small. Our speech model is a lot bigger, and it deals with more complex input and output, so it needs more space (10*1,024). This is primarily determined by trial and error. Because we're dealing with a spectrogram as our input, the input tensor has more dimensions—four, in total. The first dimension is just a wrapper containing a single element. The second and third represent the "rows" and "columns" of our spectrogram, which happens to have 49 rows and 40 columns. The fourth, innermost dimension of the input tensor, which has size 1, holds each individual "pixel" of the spectrogram.

4) **Allocate input tensor:** We grab a sample spectrogram for a "yes," stored in the constant g_yes_micro_f2e59fea_nohash_1_data. The constant is defined in the file micro_features/yes_micro_features_data.cc, which was included by this test. The spectrogram exists as a 1D array, and we just iterate through it to copy it into the input tensor.

5) **Run inference:** We invoke the interpreter, and run inference on the assigned input. We do sanity checks on the dimensions of the output tensor.

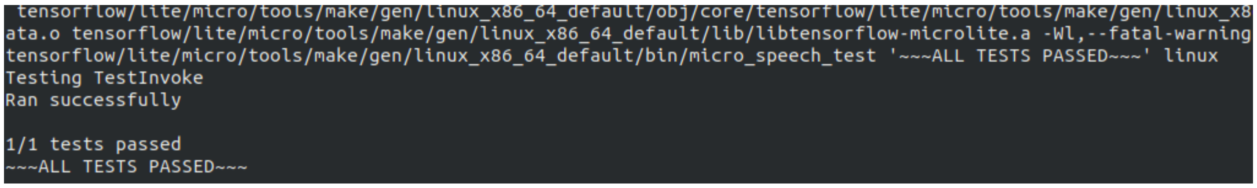
6) **Check if the output is correct or not:** After inference is done, we confirm that our expected label achieved the highest score, by making sure that it's greater than all the other scores.

The example application contains an audio provider only compatible with macOS. We can use the make command to build the application, and then run the generated executable as follows, but we will not get any output, because it is not compatible with my laptop's linux system. If it were a Mac, it would keep on detecting words and display it's confidence scores in real time.

```
make -f tensorflow/lite/micro/tools/make/Makefile micro_speech
tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/bin/micro_speech
```

However, we can still run some sanity checks to check if the individual components are working correctly or not by running the micro_speech_test described above:

```
make -f tensorflow/lite/micro/tools/make/Makefile test_micro_speech_test
```



```
 tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/obj/core/tensorflow/lite/micro/tools/make/gen/linux_x8
ata.o tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/lib/libtensorflow-microlite.a -Wl,--fatal-warning
tensorflow/lite/micro/tools/make/gen/linux_x86_64_default/bin/micro_speech_test '~~~ALL TESTS PASSED~~~' linux
Testing TestInvoke
Ran successfully

1/1 tests passed
~~~ALL TESTS PASSED~~~
```

The default application is designed to work with "yes" and "no" word detections. Our aim is to use it as a wakework detector, and for that reason, we have retrained the model to recognise "on" and "off". We need to make some changes in the application as well to achieve our desired behaviour.

Using the model in our project

To use our new model, we need to do three things:

1) In micro_features/tiny_conv_micro_features_model_data.cc, replace the original model data with our new model.
2) Update the label names in micro_features/micro_model_settings.cc with our new "on" and "off" labels.
3) Update the device-specific command_responder.cc to take the actions we want for the new labels.

Replacing the model is easy, as we have already trained our model and converted into an appropriate format. To change the labels, we need to modify the file where the labels are stored: micro_features/micro_model_setting.cc. We can simply replace "yes" and "no" with "on" and "off" respectively, because as shown earlier they correspond to prediction = 2 and 3 respectively. The changes are shown below:

Original Code:

```
const char* kCategoryLabels[kCategoryCount] = {
      "silence",
      "unknown",
      "yes",
      "no",
};
```

Modified Code:

```
const char* kCategoryLabels[kCategoryCount] = {
      "silence",
      "unknown",
      "on",
      "off",
};
```

The remaining step is to update the output behaviour of the code that uses these labels. We use it for the arduino, hence inside the micro_speech example of the arduino library, it is defined in the arduino_command_responder.cpp file. The default behaviour of the code is to light up the Green LED on detecting the word "yes", and light up the Red LED on detecting the word "no", and light up the Blue LED on detecting an unknown word. If a command was uttered, but has been silent for > 3 secs, then all the LEDs are switched off. Our desired behaviour is similar. We wish to turn the built-in LED of the microcontroller on by saying "on" and off by saying "off". We have already updated the labels earlier. We cannot do this just by comparing the 1st alphabet, since it is "o" for both of them, hence we compare both the 1st and 2nd alphabet. We turn on/off the Blue LED, and do nothing if an unknown voice is detected. We also comment out the automatic switching off part after 3 secs, because we wish to turn it off by saying “off”. The modifications are shown as follows:

Original Code:

```
if (is_new_command) {
      TF_LITE_REPORT_ERROR(error_reporter, "Heard %s (%d) @%dms",
found_command, score, current_time);
      if (found_command[0] == 'y') {
      last_command_time = current_time;
      digitalWrite(LEDG, LOW);  // Green for yes
      }
      if (found_command[0] == 'n') {
      last_command_time = current_time;
      digitalWrite(LEDR, LOW);  // Red for no
      }
      if (found_command[0] == 'u') {
      last_command_time = current_time;
      digitalWrite(LEDB, LOW);  // Blue for unknown
      }
  }
  if (last_command_time != 0) {
      if (last_command_time < (current_time - 3000)) {
      last_command_time = 0;
      digitalWrite(LED_BUILTIN, LOW);
      digitalWrite(LEDR, HIGH);
      digitalWrite(LEDG, HIGH);
      digitalWrite(LEDB, HIGH);
      }
      return;
  }
```

Modified Code:

```
if (is_new_command) {
      TF_LITE_REPORT_ERROR(error_reporter, "Heard %s (%d) @%dms",
found_command, score, current_time);
      if (found_command[0] == 'o' && found_command[1] == 'n') {
      digitalWrite(LEDB, LOW);  // Blue ON for ON
      }
      if (found_command[0] == 'o' && found_command[1] == 'f') {
      digitalWrite(LEDB, HIGH);  // BLUE OFF FOR OFF
      }
  }
```

Conclusion

We have looked at the entire end to end process of training a complex neural network model, converting it for use in an embedded scenario, testing it on a dev machine and then finally running the application on an embedded board. We have also looked at necessary changes needed to be made for our own application, the effects of optimizations and quantization on the size and performance of a model.

Uploaded deliverables

Please find the attached zip folder, with the following files containing the required deliverables:

```
18EC10075_EML_Assignment_2_Submission
├── micro_speech
│    ├── arduino_command_responder.cpp
│    ├── micro_model_setting.cc
│    ├── models
│    │    ├── float_model.tflite
│    │    ├── model.cc
│    │    ├── model.tflite
│    │    └── saved_model
│    │         ├── saved_model.pb
│    │         └── variables
│    │               ├── variables.data-00000-of-00001
│    │               └── variables.index
│    └── train
│      ├── checkpoint
│      ├── tiny_conv.ckpt-11000.data-00000-of-00001
│      ├── tiny_conv.ckpt-11000.index
│      ├── tiny_conv.ckpt-11000.meta
│      ├── tiny_conv.ckpt-12000.data-00000-of-00001
│      ├── tiny_conv.ckpt-12000.index
│      ├── tiny_conv.ckpt-12000.meta
│      ├── tiny_conv.ckpt-13000.data-00000-of-00001
│      ├── tiny_conv.ckpt-13000.index
│      ├── tiny_conv.ckpt-13000.meta
│      ├── tiny_conv.ckpt-14000.data-00000-of-00001
│      ├── tiny_conv.ckpt-14000.index
│      ├── tiny_conv.ckpt-14000.meta
│      ├── tiny_conv.ckpt-15000.data-00000-of-00001
│      ├── tiny_conv.ckpt-15000.index
│      ├── tiny_conv.ckpt-15000.meta
│       ├── tiny_conv_labels.txt
│       └── tiny_conv.pbtxt
└── sine_value_prediction
        ├── arduino_output_handler.cpp
        ├── orig_model_3
        │     ├── assets
        │     ├── keras_metadata.pb
        │     ├── saved_model.pb
        │     └── variables
        │        ├── variables.data-00000-of-00001
        │        └── variables.index
        ├── orig_model_5
        │     ├── assets
        │     ├── keras_metadata.pb
        │     ├── saved_model.pb
        │     └── variables
        │        ├── variables.data-00000-of-00001
        │        └── variables.index
        ├── sine_model_3.tflite
        ├── sine_model_5.tflite
        ├── sine_model_quantized_3.cc
        ├── sine_model_quantized_3.tflite
        ├── sine_model_quantized_5.cc
        └── sine_model_quantized_5.tflite
```

Important files for Sine Value Prediction:

1) arduino_output_handler.cpp : Modified output behaviour
2) orig_model_3: Saved model directory for model 3
3) orig_model_5: Saved model directory for model 5
4) sine_model_quantized_3.cc: Cpp file containing converted optimised model 3
5) sine_model_quantized_5.cc: Cpp file containing converted optimised model 5

Important files for micro speech detection:

1) arduino_commnd_responder.cpp: Modified output behaviour
2) micro_model_setting.cc: Modified labels of new model
3) models/ : Directory containing original and converted models
4) train/ : Directory containing the training checkpoints

Link to Google Colab Notebook used for part 1: Sine Wave Prediction

Link to Google Colab Notebook used for part 2: Speech Detection