

Edge AI for Sensor Networks

Project Report

Wake up call for smartwatch

Capteurs/Actionneurs & IA embarquée

Authors

EWA KUPCZYK

EIT Digital DSc MSc 1 student

`ewa.kupczyk@etu.univ-cotedazur.fr`

SIVADINESH PONRAJAN

EIT Digital AuS MSc 1 student

`sivadinesh.ponrajan@etu.univ-cotedazur.fr`

Université Cote d'Azur

Submitted: May 1, 2023

Contents

1	Introduction	1
2	Selection of the Project	2
2.1	Aim of Project	2
3	Recording the data	3
4	Training a neural networks model	4
4.1	Windowing	5
4.2	Learning Rate hyperparameter	7
5	Real-time prediction on the board	8
6	Inference	12
6.1	Full precision / Floating-point representation	12
6.1.1	Python Model vs C converted model	12
6.1.2	Onboard Evaluation	13
6.2	Quantization / Fixed-point representation	13
6.2.1	Python Model vs C converted model	14
6.2.2	Onboard Evaluation	14
6.3	Inference Analysis	15
6.3.1	Performance	15
6.3.2	Time Taken & Latency	15
6.3.3	ROM Consumption	16
6.3.4	Power Consumption	16
7	Conclusion	16

1 Introduction

The main objective of this lab is to explore the integration of Artificial Intelligence and embedded technology. With the rise of edge computing, it has become increasingly important to develop efficient algorithms that can run on resource-constrained devices. The integration of Artificial Intelligence (AI) in edge devices has opened up a whole new world of possibilities, from real-time data processing to intelligent decision-making. The final project of this lab aims to deploy a neural network that can run on the RFThings-AI Dev Kit board, a microcontroller, using data collected from sensors such as the accelerometer.

The project follows a series of steps, starting with the collection of the accelerometer sensor's data from the RFThings-AI Dev Kit board, and annotation of the collected data using the microAI GUI software, followed by preprocessing and training a convolutional neural network (CNN) using the collected data with the well known Tensorflow library. The purpose of the CNN is to process and analyze the accelerometer data, allowing us to successfully make predictions of a specific pattern. One of the key aspects of this project is converting the trained model into C code using microAI framework. This step enables the implementation of real-time prediction on the same board, showcasing the practical applications of AI in IoT. Finally, the model is used to infer results in real-time.

The project offers a hands-on experience with the complete pipeline of developing AI models for edge devices, including data collection, preprocessing, model training, and deployment. Through this project, we have gained insights into the challenges and opportunities associated with developing AI models that can run on resource-constrained devices. By the end of the project, we have a solid understanding of how to design and deploy neural networks on microcontrollers, opening up new possibilities for the future of edge computing.

2 Selection of the Project

The main goal of the project is to develop a system that can collect data, use neural networks to train them, and then **deploy the trained model on an edge device to predict a specific pattern**. To achieve this goal, we were given two options to proceed with the project.

- The first option was to use the **UCA Board - RFThings-DKA IoT board**, which comes with an accelerometer sensor. The accelerometer on the board can be used to collect data related to movement, vibration, and other physical activity.
- The second option was to use a **Nucleo board with a microphone shield** to work with an audio-based project. Microphones can be used to capture sound and convert it into electrical signals that can be processed by a computer or other device.

2.1 Aim of Project

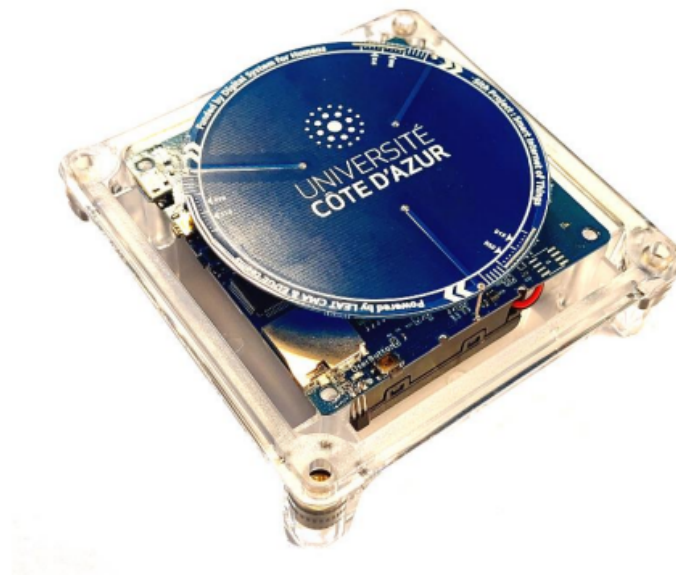


Figure 1: UCA Board - RFThings DKA IoT Board

After considering both options, we decided to use the UCA board and its accelerometer for our project. In our case, we wanted to detect the hand gesture corresponding to a time-looking movement, and our project is "**Wake up call for smartwatch**". In the battery-saving mode of a smartwatch, the display is typically turned off to conserve power. This means that the user needs to manually turn on the display by pressing the home or crown button. The aim of our project is to create an automated system that predicts a specific pattern in the accelerometer's data of the UCA board and uses this prediction to turn on the OLED display of a smartwatch.

3 Recording the data

The UCA board used in our project has an STM32-based Microcontroller that uses an ARM Cortex M4 architecture and runs at a frequency of 80 MHz. With 1 MB Flash and 128 KB SRAM, this board provided us with enough resources to work with. To begin, we started with the hardware environment. We recorded the data offline, but online data collection was also possible. The Arduino program is meant to be just a bridge between the accelerometer and the main computer.

In the `setup()` function of the Arduino code:

- We initialize the accelerometer using the I2C communication protocol (`icm.begin(Wire, 0);`)
- Once the code is uploaded, and when it is ready, "READY" will appear in the serial console stating that the recording of the data started.

Then we can start the acquisition. There is the variable `sample_i` that iterates until reaches `ACCEL_SAMPLES = DURATION * SAMPLE_RATE`. We set the duration at 5 minutes.

- During the data recording, the blue LED starts to blink.
- The duration of the recording is configured with `DURATION` constant (by default 5 minutes).
- During the data recording process, we split the first 2 minutes 30 seconds for the positive activity i.e. The time-looking movement and the rest of the 2 minutes 30 seconds for a negative activity like random activity.
- All the data are stored in the `accel_buf` array, which we will then read.
- Once the capture is finished and while waiting for the host the LED stops blinking and stays on.

Once the data is recorded successfully, we started the microAI GUI, and the collected data from the board will send all the serially printed values to the microAI GUI and plots it seamlessly.

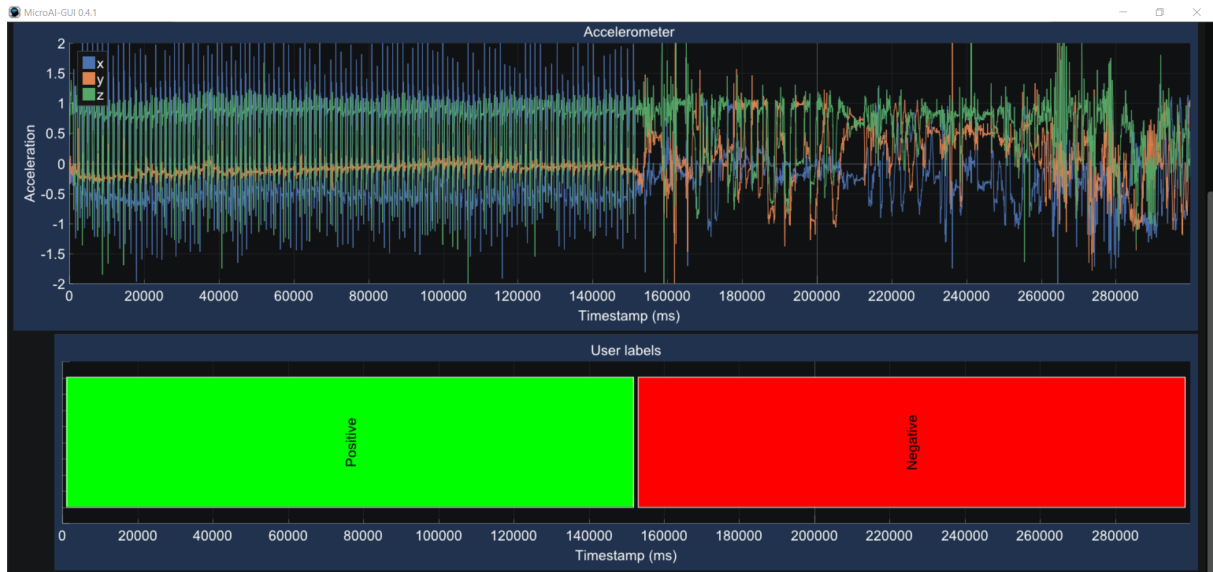


Figure 2: The MicroAI GUI after labelling the data

Once the data is plotted, we manually create the positive and negative labels for the recorded data. Since we also classified the first half of our time recording the positive actions and the rest for the negative actions, it is easy to label accordingly without any confusion. Once the data is labeled, it can be exported as a CSV file and saved locally. The next step is to train the data using neural networks.

4 Training a neural networks model

In order to train our neural network, we needed a dataset that could be used to teach our model how to recognize patterns in accelerometer data which we did with our custom dataset that was recorded and annotated earlier.

```

1 with open('dataSplit/UCA-Data.csv') as f:
2     next(f) # Skip header
3     for l in f:
4         d = l.split(';')
5         x_full.append([float(d[1]), float(d[2]), float(d[3])]) # Store
3-axis accelerometer adata
6         y_full.append(1 if 'Positive' in d[4] else 0) # Store positive
labels

```

Listing 1: Reading Data using Pandas

As we can see in the above code snippet, we converted the Y data into numerical form. If the existing data is Positive, it will be replaced by 1 else it will be 0.

4.1 Windowing

After that, in our notebook, we used a technique called **Windowing**.

Windowing

```
In [3]: SIZE = 32
        CLASSES = 2
        windowcount = np.ceil(x_full.shape[0]/SIZE).astype(int)
        x_full = np.resize(x_full, (windowcount, SIZE, x_full.shape[-1]))
        y_full = np.resize(y_full, (windowcount, SIZE))
        y_full = np.array([np.argmax(np.bincount(w)) for w in y_full]) # Select label with highest number of occurrence for each window
        y_full = to_categorical(y_full, num_classes=CLASSES) # Convert back to one-hot encoding
```

Figure 3: Windowing the data

- Windowing refers to the process of dividing a long sequence of data into shorter segments, called windows or frames, before feeding them into a neural network for analysis or prediction. This approach is often used in time series analysis.
- The main reason for using windowing is that long sequences of data can be computationally expensive to process and may require a lot of memory. By dividing the sequence into shorter windows, we can reduce the amount of data that needs to be processed at any given time and make the task more manageable.
- In addition, windowing can help capture local patterns or features in the data that might be missed if the entire sequence is analyzed at once. By breaking the data into shorter windows, we can focus on specific segments of the sequence and extract more meaningful information.
- The size of the window or frame is typically a hyperparameter that needs to be optimized for each specific task. A larger window size may capture more global patterns in the data, but may also be more computationally expensive and prone to overfitting. A smaller window size may capture more local patterns but may miss some of the broader trends in the data.
- Overall, windowing is a useful technique in deep learning for handling long sequences of data and extracting meaningful information from them.

Then, we shuffled the samples of windows to ensure that the model learns to recognize patterns in a more generalized way. This helps to prevent overfitting, which occurs when the model learns to recognize patterns that are unique to the training data but not to new data. It is important to note that the shuffling should happen with respect to the windows and not for individual data. Further, the data should be split for testing and training. We have made 80% of the data for the training phase and the remaining 20% for the testing phase.

We then used the dataset to train a convolutional neural network (CNN) model. CNNs are a type of neural network that is particularly effective at recognizing patterns in images or sequences of data, such as accelerometer data. We trained our model until we achieved a high level of accuracy in the validation set.



```
model = Sequential()
model.add(Input(shape=(SIZE, 3)))
model.add(Conv1D(filters=10, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=6))
model.add(Flatten())
model.add(Dense(units=6))
model.add(Dense(units=CLASSES))
model.add(Activation('softmax'))
opt = tf.keras.optimizers.Adam(learning_rate=9e-4)
model.summary()
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['categorical_accuracy'])
```

Figure 4: CNN Architecture

In the above code, various parameters were changed and we observed the impact on the model's accuracy. Some of the factors that influence much in the results are the number of layers, the type of layers, and the learning rate. We use ReLu activation function for the network to be able to learn more complex data, by doing this, we will introduce non-linearity to the neural network. What is more the 'softmax' activation function is used for our neural network to be able to output probability distribution over the classes, that sums to 1 and helps to conduct better predictions about the class of input. The learning rate is a crucial hyperparameter in neural network training, and choosing an appropriate value can significantly impact the performance of the model.

Model: "sequential"		
Layer (type)	Output Shape	Param #
=====		
conv1d (Conv1D)	(None, 30, 10)	100
max_pooling1d (MaxPooling1D)	(None, 5, 10)	0
flatten (Flatten)	(None, 50)	0
dense (Dense)	(None, 6)	306
dense_1 (Dense)	(None, 2)	14
activation (Activation)	(None, 2)	0
=====		
Total params: 420		
Trainable params: 420		
Non-trainable params: 0		

Figure 5: Model Summary

4.2 Learning Rate hyperparameter

- The learning rate is a hyperparameter in neural networks that determines the step size at each iteration while moving toward a minimum loss function during training. It plays a crucial role in the training process as it affects the speed and accuracy of convergence.
- If the learning rate is too high, the model may fail to converge, overshooting the minima and diverging, leading to poor accuracy. On the other hand, if the learning rate is too low, the model may converge slowly, requiring a longer time for training, and may get stuck in a local minimum instead of reaching the global minimum, leading to suboptimal accuracy.
- In general, it is recommended to start with a moderate learning rate and then adjust it based on the performance of the model during training.
- A well-tuned learning rate can lead to faster convergence and better accuracy, while an inappropriate value can cause the model to diverge or converge slowly. Therefore, it is essential to carefully choose the learning rate and monitor its impact on the model's training process.

Problems Faced:

Initially, we had more dense layers in the neural network architecture so that the size of the model became too big. It couldn't accommodate in the memory of the UCA Board so we had to restructure the neural network architecture and get an increased accuracy. The learning curve of the training and the validation is plotted as a graph and is as follows. Finally, the trained model is also saved in ".h5" format.

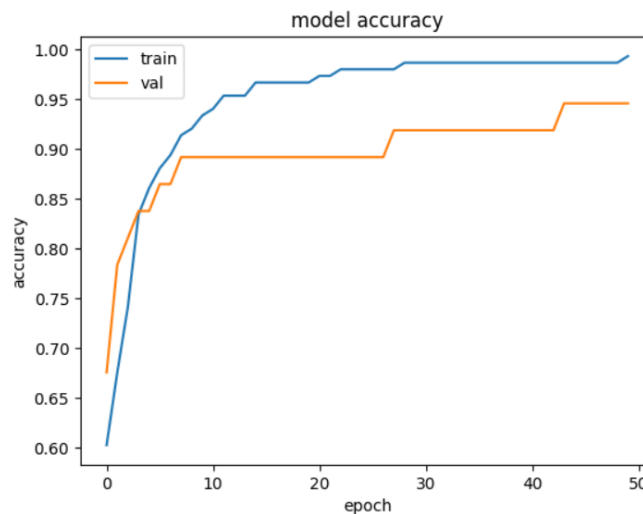


Figure 6: Accuracy of the training and validation

5 Real-time prediction on the board

In order to deploy a neural network model to a microcontroller for real-time prediction, we need to convert the model into C code that can be executed on the microcontroller's hardware. The **kerascnn2c** library from the **microAI framework** can be used to accomplish this task. However, we have to choose between two options for doing the computations: floating-point or fixed-point representation.

In this project, we will be implementing both floating-point and fixed-point representations of the neural network model and comparing their performance in terms of accuracy and resource consumption.

By default, the fixed point conversion was already provided in the jupyter notebook, so we proceeded with that and converted it into a ".h" header file which is compatible with the C Code to import them. Then this C code is moved to the inference Arduino folder and then the program is flashed. Once the code is uploaded, we can see the real-time detection in the serial monitor.

```
1 #include <ICM_20948.h>
2 extern "C" {
3 #include "project_model_fixed.h"
4 }
5
6 #define SAMPLE_RATE 20
7
8 // ICM
9 ICM_20948_I2C icm;
10
11 static long long timer = 0;
12
13 static float accel_buf[MODEL_INPUT_SAMPLES][MODEL_INPUT_CHANNELS];
14 static size_t sample_i = 0;
15
16 void setup() {
17     // 10-14-21: Mandatory on new board revision otherwise I2C does not
18     // work
19     pinMode(SD_ON_OFF, OUTPUT);
20     digitalWrite(SD_ON_OFF, HIGH);
21
22     // Initialize pin for blinking LED
23     pinMode(PIN_LED, OUTPUT);
24
25     // Initialize serial port
26     Serial.begin(115200);
```

```

26
27 // Wait for the initialization
28 while (!Serial && millis() < 5000);
29
30 // Initialize I2C used by IMU
31 Wire.begin();
32
33 // Initialize IMU
34 icm.begin(Wire, 0);
35
36 // Set sample rate to ~20Hz
37 icm.setSampleRate((ICM_20948_Internal_Acc | ICM_20948_Internal_Gyr), {
38     56, 55}); // a = 56 -> 20.09Hz, g = 55 -> 20Hz
39
40 // Notify readiness
41 Serial.println("READY");
42
43 timer = millis();
44 }
45
46 void loop() {
47     static char msg[128];
48
49     if (sample_i < MODEL_INPUT_SAMPLES) {
50         // Try to respect sampling rate
51         if (millis() > timer + (1000 / SAMPLE_RATE)) {
52             timer = millis();
53
54             if (icm.dataReady()) {
55                 // Read accelerometer data
56                 icm.getAGMT(); // The values are only updated when you call '
57                 getAGMT'
58             }
59
60             // Blink LED for activity indicator
61             //digitalWrite(PIN_LED, 1 - digitalRead(PIN_LED));
62
63             // Read accelerometer data
64             accel_buf[sample_i][0] = icm.accX() / 1000.0f;
65             accel_buf[sample_i][1] = icm.accY() / 1000.0f;
66             accel_buf[sample_i][2] = icm.accZ() / 1000.0f;
67             // Format message with accelerometer data
68             snprintf(msg, sizeof(msg), "0,%f,%f,%f\r\n", accel_buf[sample_i][0],
69                 accel_buf[sample_i][1], accel_buf[sample_i][2]);
70
71             // Send message over serial port
72             //Serial.print(msg);

```

```

70     sample_i++;
71 }
72 } else {
73     //delay(3000); digitalWrite(PIN_LED, LOW);
74     // Buffer is full, perform inference
75     static unsigned int inference_count = 0;
76     static number_t inputs[MODEL_INPUT_CHANNELS][MODEL_INPUT_SAMPLES];
77     static number_t outputs[MODEL_OUTPUT_SAMPLES];
78     long long t_start = millis();
79     // Convert inputs from floating-point then to fixed-point
80     for (size_t i = 0; i < MODEL_INPUT_SAMPLES; i++) {
81         for (size_t j = 0; j < MODEL_INPUT_CHANNELS; j++) {
82             inputs[j][i] = clamp_to_number_t((long_number_t)(accel_buf[i][j]
83 * (1<<FIXED_POINT)));
84         }
85     }
86     digitalWrite(PIN_LED, HIGH);
87     // Run inference
88     cnn(inputs, outputs);
89     digitalWrite(PIN_LED, LOW);
90     // Get output class
91     unsigned int label = 0;
92     float max_val = outputs[0];
93     for (unsigned int i = 0; i < MODEL_OUTPUT_SAMPLES; i++) {
94         Serial.print("Output n.");
95         Serial.print(i);
96         Serial.print(": ");
97         Serial.println(outputs[i]);
98         if (max_val < outputs[i]) {
99             max_val = outputs[i];
100             label=i;
101             Serial.print("Wake up call detected for the smartwatch!\n");
102         }
103     }
104     inference_count++;
105     long long t_end = millis();
106     snprintf(msg, sizeof(msg), "Result: %d,%lld", label, t_end - t_start
107 );
108     Serial.println(msg);
109     digitalWrite(PIN_LED, label);
110     sample_i = 0;
111 }

```

Listing 2: Arduino Script for the Inference

```
COM7
Output n.0: -57
Output n.1: 146
Wake up call detected for the smartwatch!
Result: 1,1
```

Figure 7: Output in Serial monitor when the pattern is detected

When we run the inference script and if the specific pattern of the accelerometer's data which is trained for positive movement is detected, it prints in the serial monitor and the blue LED in the board turns ON. For the rest of the actions, the LED stays OFF.



Figure 8: LED is ON indicating the pattern is detected

Thus we have successfully trained and deployed the neural network model in the embedded device and could successfully implement the real-time prediction of a specific pattern.

6 Inference

Let us discuss about the results that we achieved over different methods and infer from them.

6.1 Full precision / Floating-point representation

Floating-point representation is a method of representing numbers that allows for a wide range of values with high precision. This representation requires a larger amount of memory and computation power, which may not be feasible for microcontrollers with limited resources.

```

1 res = kerasnnc2c.Converter(output_path=Path('polyhar_output_float'),
2                             number_type='float',
3                             ).convert_model(copy.deepcopy(model))
4
5 with open('project_model_float.h', 'w') as f:
6     f.write(res)

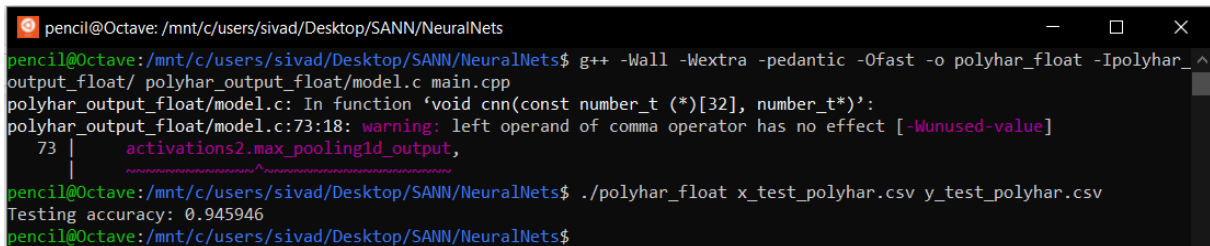
```

Listing 3: Generate C code for the trained model with floating-point representation

With the above snippet of code from the microAI framework using the kerasnnc2c library, we could successfully generate C code for the trained model with floating-point representation.

6.1.1 Python Model vs C converted model

When trained with Python on the desktop, we achieved an accuracy of around **95%** with the test data. To check whether the model that is converted into C code could also reproduce something similar, we compiled the converted C model and run it against the splitted test data CSV file as follows. We almost achieved the same accuracy as on the desktop model i.e. **94.59%**



```

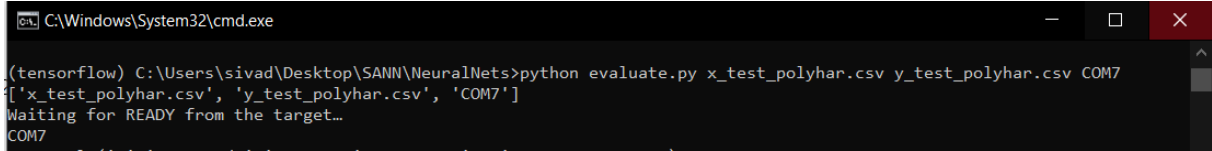
pencil@Octave: /mnt/c/users/sivad/Desktop/SANN/NeuralNets
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$ g++ -Wall -Wextra -pedantic -Ofast -o polyhar_float -Ipolyhar_
output_float/ polyhar_output_float/model.c main.cpp
polyhar_output_float/model.c: In function 'void cnn(const number_t (*)[32], number_t*)':
polyhar_output_float/model.c:73:18: warning: left operand of comma operator has no effect [-Wunused-value]
   73 |         activations2.max_pooling1d_output,
      |         ~~~~~^~~~~~
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$ ./polyhar_float x_test_polyhar.csv y_test_polyhar.csv
Testing accuracy: 0.945946
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$

```

Figure 9: Compile the Floating-point C code for x86 and evaluate

6.1.2 Onboard Evaluation

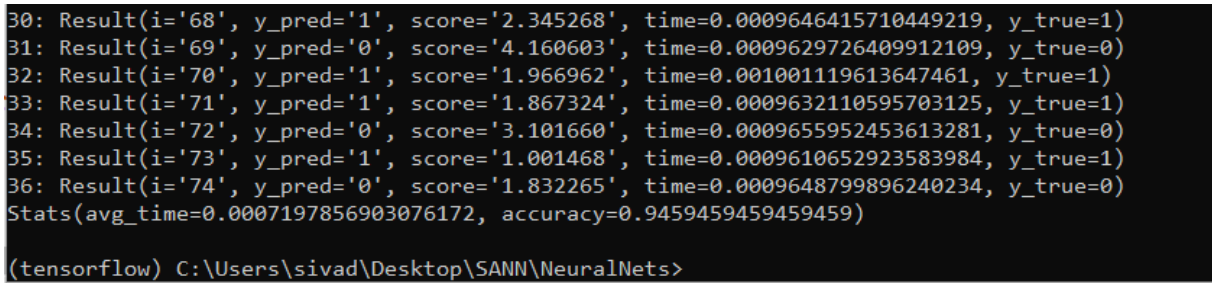
We are already provided with the script `evaluate.py` and the **S4Lab6** Arduino file. Once we add and modify the model in the Arduino script and upload it, we call the Python script of `evaluate.py` where we pass the argument of the test file names that we have splitted earlier and the Port name/ number where we have connected our UCA board to the desktop.



```
C:\Windows\System32\cmd.exe
(tensorflow) C:\Users\sivad\Desktop\SANN\NeuralNets>python evaluate.py x_test_polyhar.csv y_test_polyhar.csv COM7
['x_test_polyhar.csv', 'y_test_polyhar.csv', 'COM7']
Waiting for READY from the target...
COM7
```

Figure 10: Python evaluation script to evaluate onboard

Once the evaluation is made, we could see the results in the same terminal.



```
30: Result(i='68', y_pred='1', score='2.345268', time=0.0009646415710449219, y_true=1)
31: Result(i='69', y_pred='0', score='4.160603', time=0.0009629726409912109, y_true=0)
32: Result(i='70', y_pred='1', score='1.966962', time=0.001001119613647461, y_true=1)
33: Result(i='71', y_pred='1', score='1.867324', time=0.0009632110595703125, y_true=1)
34: Result(i='72', y_pred='0', score='3.101660', time=0.0009655952453613281, y_true=0)
35: Result(i='73', y_pred='1', score='1.001468', time=0.0009610652923583984, y_true=1)
36: Result(i='74', y_pred='0', score='1.832265', time=0.0009648799896240234, y_true=0)
Stats(avg_time=0.0007197856903076172, accuracy=0.9459459459459459)
(tensorflow) C:\Users\sivad\Desktop\SANN\NeuralNets>
```

Figure 11: Onboard evaluation of the generated model

As you could see in the above image, only 36 predictions were made because the collected series of data for the 5 minutes were grouped using the windowing technique. Since the test data we collected is very less and based on the size we provided, there were only 36 windows. And finally, we could see that we have achieved almost the same accuracy as that of the model trained on the desktop.

6.2 Quantization / Fixed-point representation

Fixed-point representation is a more efficient way of representing numbers in microcontrollers. In this representation, numbers are represented as integers, and a scaling factor is applied to convert them to their actual value. This allows for faster and more memory-efficient calculations as compared to floating-point representation.


```

1 res = kerasnnc2c.Converter(output_path=Path('polyhar_output_fixed'),
2     fixed_point=9, # Number of bits for the fractional part, Q7.9
3     format
4     number_type='int16_t', # Data type for weights/activations (16
5     bits quantization)
6     long_number_type='int32_t', # Data type for intermediate results
7     number_min=-(2**15), # Minimum value for 16-bit signed integers
8     number_max=(2**15)-1 # Maximum value for 16-bit signed integers
9     ).convert_model(copy.deepcopy(model))
10 with open('project_model_fixed.h', 'w') as f:
11     f.write(res)

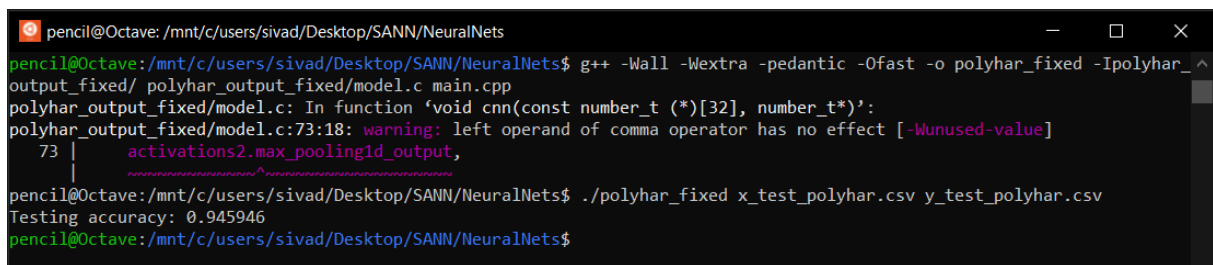
```

Listing 4: Generate C code for the trained model with 16-bit fixed-point representation

With the above snippet of code from the microAI framework using the kerasnnc2c library, we could successfully generate C code for the trained model with 16-bit fixed-point representation.

6.2.1 Python Model vs C converted model

As we know that when we trained with Python on the desktop, we achieved an accuracy of around **95%** with the test data. To check whether the model that is converted into C code could also reproduce something similar, we compiled the converted C model and run it against the splitted test data CSV file as follows. We almost achieved the same accuracy as on the desktop model and also similar to the one that we got with the floating point representation i.e. **94.59%**



```

pencil@Octave: /mnt/c/users/sivad/Desktop/SANN/NeuralNets
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$ g++ -Wall -Wextra -pedantic -Ofast -o polyhar_fixed -Ipolyhar_
output_fixed/ polyhar_output_fixed/model.c main.cpp
polyhar_output_fixed/model.c: In function 'void cnn(const number_t (*)[32], number_t*)':
polyhar_output_fixed/model.c:73:18: warning: left operand of comma operator has no effect [-Wunused-value]
   73 |     activations2.max_pooling1d_output,
      |     ~~~~~^~~~~~
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$ ./polyhar_fixed x_test_polyhar.csv y_test_polyhar.csv
Testing accuracy: 0.945946
pencil@Octave:/mnt/c/users/sivad/Desktop/SANN/NeuralNets$

```

Figure 12: Compile the 16-bit fixed-point C code for x86 and evaluate

6.2.2 Onboard Evaluation

We are already provided with the script **evaluate.py** and the **S4Lab6** Arduino file. Once we add and modify the model in the Arduino script and upload it, we call the Python script of **evaluate.py** where we pass the argument of the test file names that we have splitted earlier and the Port name/ number where we have connected our UCA board to the desktop. Once the evaluation is made, we could see the results in the same terminal.


```

31: Result(i='402', y_pred='0', score='2258.000000', time=0.0010008811950683594, y_true=0)
32: Result(i='403', y_pred='1', score='705.000000', time=0.0008647441864013672, y_true=1)
33: Result(i='404', y_pred='1', score='363.000000', time=0.0008723735809326172, y_true=1)
34: Result(i='405', y_pred='0', score='435.000000', time=0.0009191036224365234, y_true=0)
35: Result(i='406', y_pred='1', score='620.000000', time=0.0, y_true=1)
36: Result(i='407', y_pred='0', score='559.000000', time=0.0, y_true=0)
Stats(avg_time=0.00033943717544143265, accuracy=0.9459459459459459)
(tensorflow) C:\Users\sivad\Desktop\SANN\NeuralNets>

```

Figure 13: Onboard evaluation of the generated model with fixed-point representation

As you could see in the above image, we could see that we have achieved almost the same accuracy as that of the model trained on the desktop.

6.3 Inference Analysis

Metrics	Floating Point	Fixed Point
Performance	High	Low
Time Taken to Predict	Longer (719.79 μ s)	Shorter (337.44 μ s)
Latency	High	Low
Power Consumption	High	Low
RAM Consumption	87.923 (Comparatively higher)	85.027 (Comparatively lower)
ROM Consumption	68232/245760 bytes (High)	67128/245760 bytes (Low)

Table 1: Comparison of Floating Point and Fixed Point Representations

6.3.1 Performance

As we know that the performance (In our case it is the prediction), should be high in the floating point when compared to the fixed point representation. But in our case, we almost got the same accuracy of **94.59%** for both the full precision and the quantized models since the windowed dataset is very small in size.

6.3.2 Time Taken & Latency

When we calculated the average time taken using the evaluate.py script, we found that the actual time taken for the floating point representation model is way higher than the fixed point representation and it matches our expected results. In our project, we got the average time taken for the floating point representation as **719.79 μ s**. and the fixed point representation as **337.44 μ s**. The time taken is almost doubled which is because of

the complexity of computations in the full precision method. Latency was also expected to be similar.

6.3.3 ROM Consumption

When the Arduino script is uploaded for inference, the ROM consumption is printed at the terminal indicating the bytes along with the percentage. In our case, with respect to the percentage, both the full precision and the quantized occupied 27% approximately. But with the bytes occupied, still we can conclude that the floating point (68232 bytes) occupies more ROM than the fixed point (67128 bytes) as expected.

6.3.4 Power Consumption

In our project, we couldn't get a chance to work wirelessly with the batteries to calculate the power consumption accurately. We used it by serially powering it from the computer which also helped a lot for serially debugging. But the expected results are, the power consumption was supposed to be higher in full precision (floating point representation) than the quantized method (fixed point representation). This is because of the high computations for better performance made by the full precision method. When we calculated, it is found that the power consumption on active mode is **65 mW** and the average power consumption per inference over the period of time T is **0.01543 mW**.

7 Conclusion

In conclusion, this project provided an excellent opportunity to explore the potential of integrating Artificial Intelligence and embedded technology. The successful deployment of a neural network on a microcontroller showcases the practical applications of AI in IoT, offering real-time data processing and intelligent decision-making capabilities. The project highlights the importance of developing efficient algorithms that can run on resource-constrained devices, which is critical for the advancement of edge computing. Through the project, we have gained valuable insights into the complete pipeline of developing AI models for edge devices, including data collection, preprocessing, model training, and deployment. Overall, this project represents a significant milestone in the development of AI models for resource-constrained devices, opening up new possibilities for the future of edge computing.