

ECSE 444: Motion Classification on the STM32L4S5, using a Deep Neural Network

Eren Ozturk

Dept. of Electrical and Computer Eng.
McGill University
Montreal, Canada
eren.ozturk@mail.mcgill.ca

Mohammed Al Noman

Dept. of Electrical and Computer Eng.
McGill University
Montreal, Canada
mohammed.alnoman@mail.mcgill.ca

Rafid Saif

Dept. of Electrical and Computer Eng.
McGill University
Montreal, Canada
rafid.saif@mail.mcgill.ca

Karim Elgammal

Dept. of Electrical and Computer Eng.
McGill University
Montreal, Canada
karim.elgammal@mail.mcgill.ca

I. INTRODUCTION

For our final project, we developed a motion classification device that utilizes the accelerometer on the STM32L4S5 to detect various activities or motions such as; brushing teeth, drinking water, eating with utensils, and others. The device is our B-L4S5I-IOT01A development board, and it is attached to the wrist with rubber bands. A neural network runs in real-time on the board and classifies the motions. The network was built off-board using TensorFlow and trained with a public database of accelerometer data. [2] After the model was ready to classify, it was converted to a microcontroller-compatible form and transferred to the board.

To support the classification task, our project also includes features such as an I2C sensor (accelerometer) for data acquisition, CMSIS-DSP for sensor filtering and UART functions to display the time spent on each activity. We implemented two modes of operation for the program; first, an inference mode and second, a training mode. For the first mode, accelerometer data is used as input to our neural network to predict the activity that is currently being performed. For the second mode, training data can be saved to external flash with a label that is entered by the user and transmitted over UART to a separate machine for training.

The device and program designed has the potential of enabling various applications that require the use of motion tracking. These include health applications which track step counts, applications to aid activity monitoring for children or even people with various mental challenges, that require a log of their activities.

II. FEATURES AND METHODOLOGY

A. Neural Network

The neural network developed for classifying motions is inspired by V. Nunavath et al.'s "Deep Learning for Classifying Physical Activities from Accelerometer Data" [1]. More specifically, their paper compares multiple architectures used

to implement the network. The two notable implementations were deep nets (DNN) and recurrent nets (RNN), each trialed with different sliding window sizes. And so, to validate across our implementation, we commenced to assess the accuracy of each of the two implementations with varying values of the sliding window parameter. Our results can be found in table 1. The table also arranges the implementations based on difficulty of implementation; this is in order to create a realistic and feasible plan of action that will allow us to deliver the program in a timely manner. And so the easiest implementation lies in the top left of the table, while the more challenging implementations are represented within the bottom right section of the table.

Network Architecture/Sliding Window Sizes	DNN	RNN
3s, No Overlap	59.93%	70.39%
5s, 1s Stride	67.83%	85.6%
10s, 1s Stride	81.29%	96.52%

TABLE I: Different Neural Networks and Corresponding Accuracy Scores

While choosing the architecture we wanted to implement, we needed to take into account a fundamental difference between V. Nunavath et al.'s implementation and our project. In the paper, the data was collected with an accelerometer beforehand then, the classification was run offline on a computer. In our project, the classification is done on the microcontroller in real time.

This has some implications for the feasibility of some of the models described in the paper. For example, the paper's complex RNN model has 4 recursive cells and each cell has many cycles as there are points in a sliding window. However, TFLite and X-CUBE-AI toolchains require us to unroll recurrent layers for every cycle, resulting in a network too large for the board's memory. Therefore, RNN-based architectures are not suitable for our task.

Having made our choice in favor of deep nets instead of

recurrent nets, we then needed to choose the input windowing scheme. Sliding windows with strides presented some challenges. Since this is a real time signal processing task, the timing between data acquisition and neural network inference must be tightly controlled. Windows with overlaps would require us to keep FIFO buffers, which in turn would require us to dynamically allocate memory for a linked list or double ended queue. To meet project time constraints, we chose a simpler method of 3-second window with no overlap, resulting in a successful proof-of-concept.

Once we chose our architecture, we implemented it using Google Colab and trained the model. For training, we used the same data [2] used in the paper. Further information on the network architecture and the training process can be found on the Colab file in our submission.

Once training was complete, the model was converted from TensorFlow to TensorFlow Lite, which is optimized for low-powered end devices. The model was saved to a .tflite file, and it was transferred over to our board using the X-CUBE-AI toolchain provided by STMicroelectronics.

B. I2C & CMSIS DSP

The data collected by the LSM6DSL accelerometer, which was connected to the microprocessor through I2C, was initially in mg's. This was not in the same format as the training data, and therefore was not appropriately scaled for the neural network to use. To bring it to the same scaling as the training data, the collected data was quantized to 6-bit values ranging from 0 to 63. This conversion was achieved by mapping the data from the range of -1500mg to 1500mg to the range of 0 to 63. Any data collected outside this range was later clipped.

After the data was mapped, a moving average filter was ran using the CMSIS DSP library FIR function. The filter was a simple moving average with all the FIR coefficients being the same (0.1). After the data was filtered, everything was rounded. Data that was outside the range was clipped, and finally data was cast so they were ready to be used as input for the neural network.

Another challenge was to properly time the sensor data acquisition and to run it alongside the neural network inference. We did not use threading or FreeRTOS for this project, but we implemented a timer interrupt, one binary ("array_ready") and one counting ("array_idx") semaphore to ensure that the inference could run concurrently with the data acquisition for use in the next inference.

The sensor communication and filtering process is inside a timer ISR. When the interrupt triggers, the accelerometer data is acquired and processed as described above. Then, the ISR checks a global variable "array_idx" to know where in the array it last wrote the measurement data. If the array is not full yet, it writes the new x,y,z measurements, increments "array_idx" and returns. Else, it flags "array_ready". Now, the main loop knows that it can copy the complete array.

Upon receiving the semaphore "array_ready", the main loop pauses the sampling timer. It quickly makes a copy of the full sensor buffer array, and resets the "array_idx"

and "array_ready" semaphores to 0. Now, the interrupt has permission to start again from index 0 and rewrite the same buffer array. While the inference is running, it is interrupted at 32Hz by the sensor and there is no race condition since the main loop has a separate copy of the sensor buffer. While the inference is being run on the samples collected in the last 3 seconds, the samples for the current time are being collected at the same time.

C. QSPI Flash

The QSPI peripheral was used to interface with the external flash memory on the discovery board. This is used as persistent memory that the user can label in order to use as future training data and improve our neural network. Each block of memory contains the label that is acquired from user input through UART and the corresponding accelerometer data for the last 3 seconds. To keep track of the number of training samples currently in flash memory, the sample count itself is stored in the first block and loaded upon reset. This count can also be used to ensure that there is enough space left in memory.

```
void flashWrite(uint8_t* write_array, int write_array_size, uint8_t label)
{
    // For this demo, only use the start addresses of blocks. In an actual application,
    // this wasteful method would be replaced by a struct that buffers multiple sample
    // arrays and autosaves them at certain intervals.
    uint32_t current_addr = (num_samples+1) * 64 * 1024;
    // Erase the current block first
    if(BSP_QSPI_Erase_Block(current_addr) != QSPI_OK) {
        Error_Handler();
    }
    // Write the label byte to the current address
    if(BSP_QSPI_Write(&label, current_addr, 1) != QSPI_OK) {
        Error_Handler();
    }
    // Read back the label byte for verification
    uint8_t label_read;
    if(BSP_QSPI_Read(&label_read, current_addr, 1) != QSPI_OK) {
        Error_Handler();
    }
    // Increment the current write address by 1 to account for the label byte.
    current_addr += 1;
    // Write the data array to current write address
    if(BSP_QSPI_Write(write_array, current_addr, 288) != QSPI_OK) {
        Error_Handler();
    }
    // Read the data array from the current address and check for errors.
    uint8_t arr_cp[288];
    if(BSP_QSPI_Read(arr_cp, current_addr, 288) != QSPI_OK) {
        Error_Handler();
    }
    // Increment the current block number.
    ++current_block;
    // Erase the first block and check for errors.
    if(BSP_QSPI_Erase_Block(0x00) != QSPI_OK) {
        Error_Handler();
    }
    // Increment the number of samples.
    ++num_samples;
    // Write the updated number of samples to the beginning of the flash memory and check for errors.
    if(BSP_QSPI_Write(&num_samples, 0x00, sizeof(num_samples)) != QSPI_OK) {
        Error_Handler();
    }
}
```

Fig. 1: flashWrite() Function

In order to achieve this, we first set up the required pin configurations on CubeMX, which meant enabling QSPI on pins PE10-15. Afterwards, we use the board support package files to interface with the external flash memory. We then initialize the external flash memory chip using the BSP_QSPI_Init() function, and read the sample count already stored on the board BSP_QSPI_Read() function. When the user wants to save training data, they enter the label through UART and the data is saved to flash using the method found in figure 1, flashWrite(). This method saves the label and then 96 samples in each axis, or 3 seconds of data. We then increment the sample count and update this sample count in the first block of memory.

Finally, in order to actually utilize the saved training data, we must be able to transmit the data onto a more powerful computer in order to train the neural network. As a basic

interface, we read the data from memory, iterating through the number of samples stored in the first block of memory. We then transmit this to the user through UART, as described in the next section.

D. UART

UART was used as the main user interface for the classifier. There were two modes of operation for the classifier, sample and inference, that could be switched between using the pushbutton.

```
void flashRead()
{
    // Loop through each saved sample
    for (int i=1; i<num_samples; ++i)
    {
        // Calculate the address of the current sample
        uint32_t addr = i * 64 * 1024;
        // Read the label of the current sample
        uint8_t current_label;
        if (BSP_QSPI_Read(&current_label, addr, sizeof(current_label)) != QSPI_OK) Error_Handler();
        // Increment the address to read the sample array
        addr += sizeof(current_label);
        // Read the sample array
        uint8_t arr[288];
        if (BSP_QSPI_Read(arr, addr, sizeof(arr)) != QSPI_OK) Error_Handler();
        // Print the label of the current sample
        char task_desc[40] = "";
        sprintf(task_desc, "%s", output_labels[current_label-1]);
        HAL_UART_Transmit(&huart1, task_desc, sizeof(task_desc), HAL_MAX_DELAY);
        // Print each line of the sample array
        for (int j=0; j<288/3; ++j)
        {
            char current_line[20] = "";
            sprintf(current_line, "%d, %d, %d", arr[j*3], arr[j*3+1], arr[j*3+2]);
            HAL_UART_Transmit(&huart1, current_line, sizeof(current_line), HAL_MAX_DELAY);
        }
        // Print a newline character to separate samples
        char newline[2] = "\n";
        HAL_UART_Transmit(&huart1, newline, sizeof(newline), HAL_MAX_DELAY);
    }
}
```

Fig. 2: flashRead() Function

In inference mode, the classifier predicted what task was being performed. The function flashRead() seen in figure 2, was implemented to take in the raw classification result. The function then extracted the task being performed as well as the time that the current task has been going on for and outputted it on the terminal. An example output can be found in figure 3.



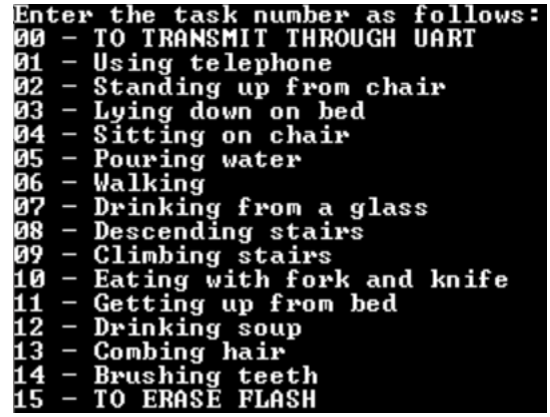
Pouring water for 15 seconds

Fig. 3: UART output in inference mode

In sample mode, the classifier ran for one interval. It then asked the user through UART what task was being performed. The user can then select a number from 0 to 15, with 1 to 14 being reserved for labels of the different tasks. Selecting one of the tasks prompts the device to save the array of inputs as well as the label corresponding to the task chosen to flash memory. The user could also select the number 0 which outputs the data from the device onto the UART, which can then be saved directly to a CSV file. This data could be used for further training of the neural network, allowing us to constantly retrain the network with new data. Finally, selecting 15 erases the contents of the flash. A summary of these indexes and prompts can be found in figure 4, and an example of the data transmitted to the flash memory can be found in figure 5.

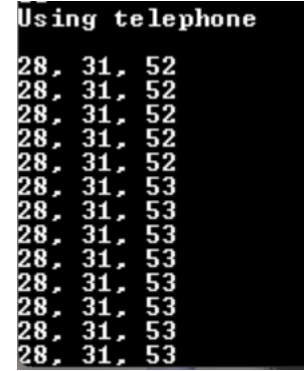
III. CONCLUSION

This project allowed us to explore the capabilities of the subject microcontroller. We were able to implement a device that reads data and performs classification on its own using



```
Enter the task number as follows:
00 - TO TRANSMIT THROUGH UART
01 - Using telephone
02 - Standing up from chair
03 - Lying down on bed
04 - Sitting on chair
05 - Pouring water
06 - Walking
07 - Drinking from a glass
08 - Descending stairs
09 - Climbing stairs
10 - Eating with fork and knife
11 - Getting up from bed
12 - Drinking soup
13 - Combing hair
14 - Brushing teeth
15 - TO ERASE FLASH
```

Fig. 4: UART menu



```
Using telephone
28, 31, 52
28, 31, 52
28, 31, 52
28, 31, 52
28, 31, 52
28, 31, 53
28, 31, 53
28, 31, 53
28, 31, 53
28, 31, 53
28, 31, 53
28, 31, 53
28, 31, 53
```

Fig. 5: Printing saved data

microprocessor features such as QSPI Flash, I2C, and UART. We were also able to develop an on-board neural network. We have also accumulated experience and codebase that will certainly be useful for future endeavors.

Al Noman worked on the preprocessing and mapping of accelerometer data onto a range that the neural network accepted. Saif wrote code for a CMSIS DSP moving average filter to filter the input, as well as the UART interface. Ozturk planned the design of the neural network and later trained it with the support of Elgammal. Ozturk then transferred it onto the board. Saif and Ozturk worked on implementing QSPI Flash. Saif and Ozturk drafted part of this report including discussions on the I2C feature, the Neural Network and QSPI Flash features. The rest of the report was written by Al Noman and Elgammal whom also formatted the report in \LaTeX .

REFERENCES

- [1] V. Nunavath et al., "Deep Learning for Classifying Physical Activities from Accelerometer Data," *Sensors*, vol. 21, no. 16, p. 5564, Aug. 2021, doi: <https://doi.org/10.3390/s21165564>.
- [2] "UCI Machine Learning Repository: Dataset for ADL Recognition with Wrist-worn Accelerometer Data Set," *archive.ics.uci.edu*, <https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer> (accessed Mar. 26, 2023).
- [3] Project Proposal Document myCourses.