General Motion Classifier: Master Manual

Last Edited: Derek Xu, UCLA

Table of Contents:

(Version 1.0)

Introduction

General Motion Classifier is software that classifies **repetitive** motions in real-time over BLE communication with the **STMicorelectronics Sensortile**, using optional user-defined Keras + Scipy models.

The required libraries and packages are: **Keras (Python)**, **LiquidDSP (C)**, **gatttool**, **bluetoothctl**, and **expect**. The software also requires the **Sensortile** to be running code found in **UCLA STMicroelectronics Tutorial 8**. Please refer to the following link for this: http://iot.ee.ucla.edu/UCLA-STMicro/index.php/UCLA_-_STMicroelectronics.

The program allows the user to define their own machine learning and signal processing classification models in the Python (See *Custom Models*). But, by default, the current program comes with the **LSTM** and the **CNN+LSTM** machine learning classification models.

Note, models do **not** interact with raw sensor data. Rather, raw sensor values are first passed through a **low-pass filter** and optionally **bias-corrected** (See *Flags and Extensions*). Furthermore, *as of now*, the program only uses **x, y, and z acceleration** data from the sensor, discarding the other sensor values.

Note, the current program cannot run on the Beaglebone platform as the space required to store Keras is too large (despite Keras/Theano supporting 32bit ARM systems). Rather, it is meant for prototyping valid machine learning models before being ported onto embedded platforms. The current program also cannot run on Windows systems due to the use of POSIX-defined multi-processing, synchronization, and IPC.

Getting Started

Base Program

To run General Motion Classifier with the default CNN+LSTM model, first modify **bctl.txt** such that the first line is the **MAC address** of your **computer** and the second line is the **MAC address** of your **Sensortile**:

Next, run the following command: **`$ make program`**

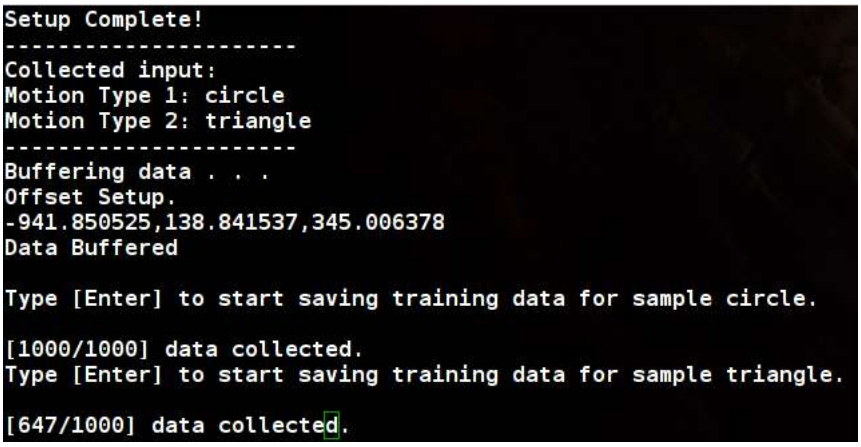
Turn on your **Sensortile** using software from **Tutorial 8 of the UCLA STMicroelectronics Tutorials**.

Run the following command: **`$ ./main`** `(or $ sudo ./main)`

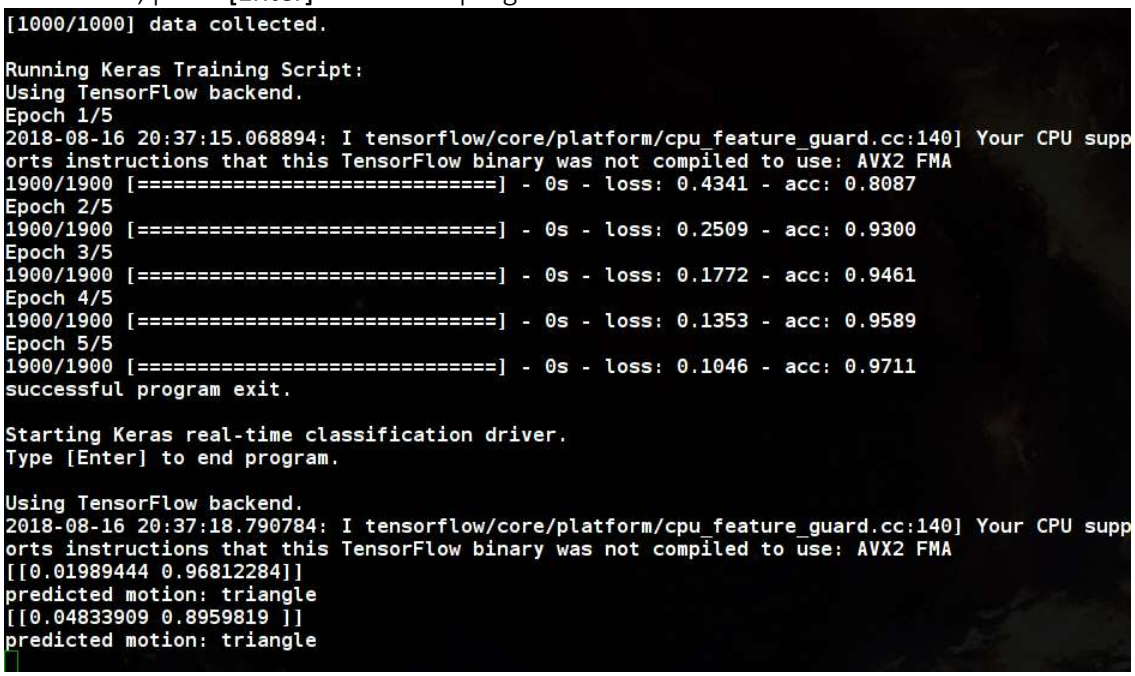
First, fill out the prompts for the number of motions and the names of each motion.

```
Please enter the number of classifications being made [2-8]: 2
Please enter the name of classifications [1/2]: circle
Please enter the name of classifications [2/2]: triangle
```

When prompted, press **[Enter]** and make each **repetitive motion** for approximately one minute or until '`[1000/1000] data collected.`' is reached. This stage is for the system to collect training data used to train the real-time classification system.



Finally, the system will train the LSTM (or user-defined) classifier and begin real-time classification. From here, press [Enter] to exit the program.



Flags and Extensions

The following describes the available flags:

```
./main [-r] [--bias=[status]]
```

The **-r** flag will skip the training step and perform real-time classification with an existing model.m5 file. Note, the user will still need to enter the names of the motions for the program to match the model predictions with human-readable mnemonics.

The **--bias=off** flag will turn off bias-correction in the DSP driver (See *Program Flow*).

The **--bias=save** flag will save the bias weights used by the DSP driver in a bias.config file.

The **--bias=load** flag will load saved bias weights from a bias.config file.

Custom Models

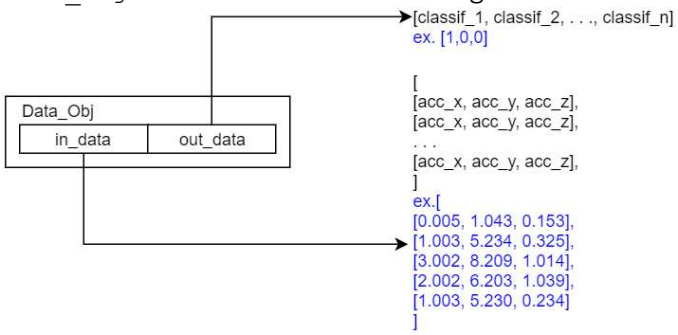
To make a custom model, simply alter the 2 functions in the **custom.py** file.

Function 1: data_func()

`data_func()` takes in a standard data format (defined below) and parses it into numpy matrices used for training by the user-defined Keras model.

Input: `[<Data_Obj>, <Data_Obj>, … , <Data_Obj>]` *i.e. list of* `Data_Obj`

`Data_Obj` is a class with the following structure:



Note: if the program is running real-time classification, the user will receive 'None' for `Data_Obj.out_data`*. For more on how the input data is collected, see Appendix A.*

Output: `(<X_data>, <Y_data>)` *i.e. tuple of numpy matrices*

`X_data` is the input data for the custom built Keras model.
`Y_data` is the predicted classification for said Keras model.

This data, usually numpy matrices, will be fed directly into the Keras model.

The following is sample code parsing input for a simple LSTM network:

```
def data_func(raw_data):
    # Default Data Manipulation (LSTM)
    keras_X = []
    keras_Y = []
    for data in raw_data:
        keras_X.append(np.asmatrix(data.in_data).transpose())
        if data.out_data != None:
            keras_Y.append(np.asarray(data.out_data))
    keras_X = np.asarray(keras_X)
    keras_Y = np.asarray(keras_Y)
    return (keras_X, keras_Y)
```

Function 2: training_func()

training_func() takes in X_data and Y_data to train a custom Keras model and produce a .m5 model for real-time classification.

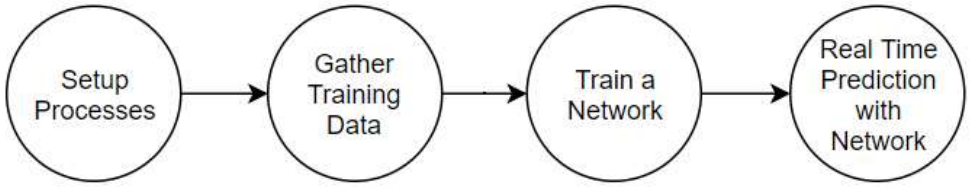
The following is example code for training a simple LSTM network:

```
def train_func(keras_training_X, keras_training_Y, training_size, classif_num,
queue_size, channels = 3):

    # basic LSTM Network
    model = Sequential()
    model.add(LSTM(100, input_shape=(channels, queue_size)))
    model.add(Dense(classif_num, activation = 'sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.fit(keras_training_X, keras_training_Y, epochs = 5, batch_size = 64)

    # save model
    model.save('model.h5')
```

For more on how the data is processed and where these functions are called, refer to *Program Flow*. Note, users can add their own DSP operations to the data, using libraries such as SciPy, during data_func() or training_func(), for more complex classification models.

Program Flow

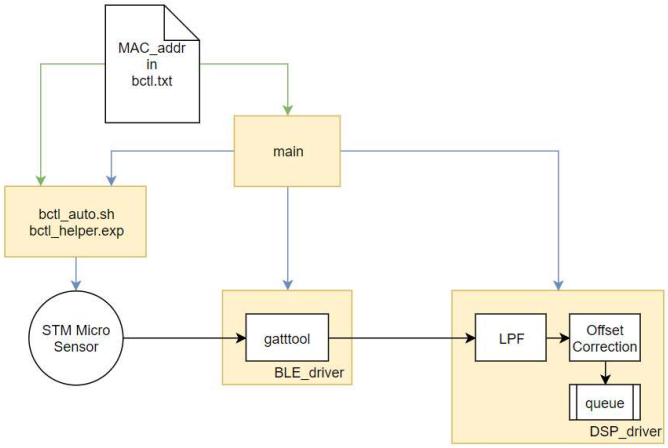
The overall program flow is as follows:



First, we spawn necessary processes, and collect metadata on classification types.
Then, we collect training data for our model.
Then, we train our model to produce a model.m5 file.
Finally, we perform real-time classification with the model.m5 file.

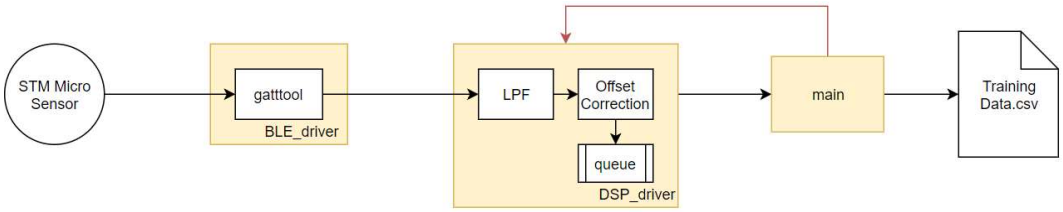
Each Step is defined, in more detail, in the following Diagram:

Step 1: Spawning Processes



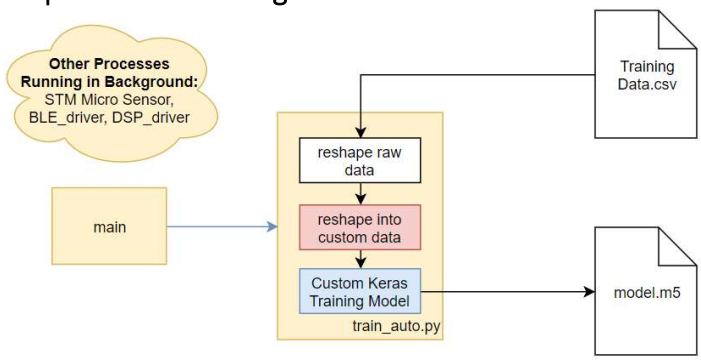
We spawn all the drivers required to collect and filter BLE data from the STMicro sensor, using MAC addresses set in `'bctl.txt'`.

Step 2: Collecting Training Data



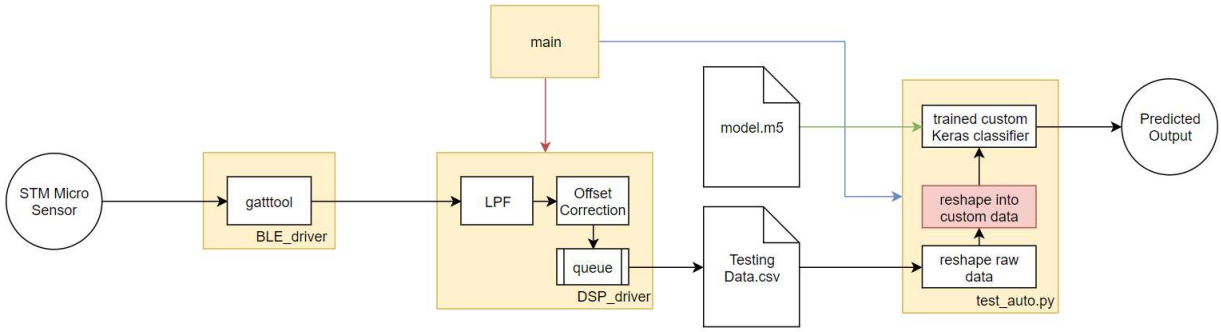
We collect training data and save it (and metadata) into a file named `'training_data.csv'`.

Step 3: Model Training



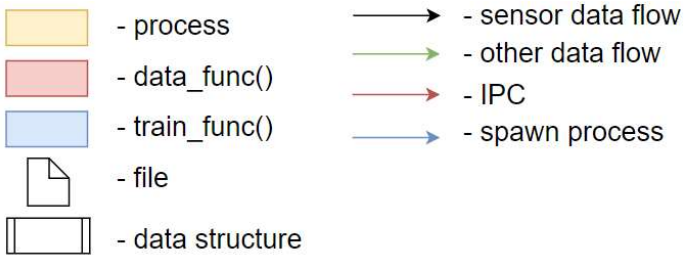
We call a python program to parse `'training_data.csv'` and train a Keras model on said training data.

Step 4: Real-Time Classification



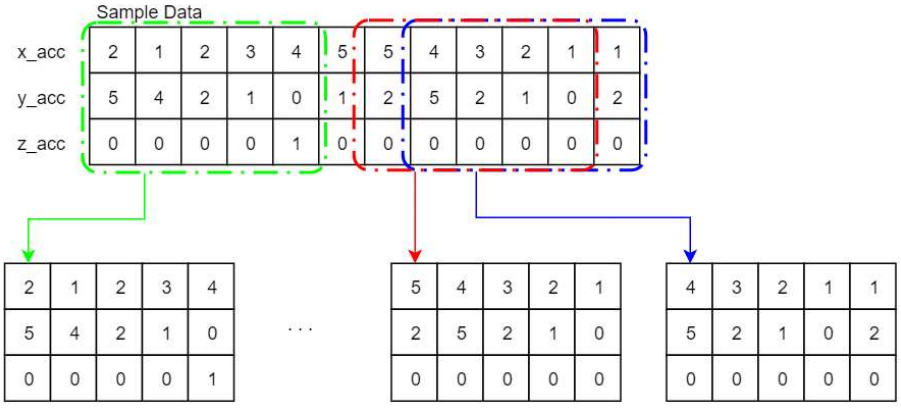
We run real-time classification with the trained model.

Legend:



Appendix A: Training Data Collection

We collect 1000 data points for each motion and divide them up to training samples as follows:



We form a window of size 50 and collect 1000-50+1=951 windowed samples for each classification.
We assign a one-hot array (ex. [0,0,1], [0,1,0], . . .) to each classification.

We store each windowed sample with the corresponding one-hot classification array in a `Data_Obj`, such that the windowed sample is `in_data` and the one-hot array is `out_data`.

We store each `Data_Obj` in a list, which we feed into `data_func()` to be transformed into the right format to be then fed into the machine learning model in `training_func()`.

NOTE: For real-time testing, we will be using the last 50 acceleration samples to classify each motion.

Appendix B: Real-Time Constraints

The input to the real-time testing will be using the last 50 acceleration samples to classify the motion. The program will run a new test for every 30 data samples collected from the Sensortile. The program can sample at a faster frequency depending on the type of model being used; however, the threshold seems to be 20 data samples per test. There are a couple constraining bottlenecks:

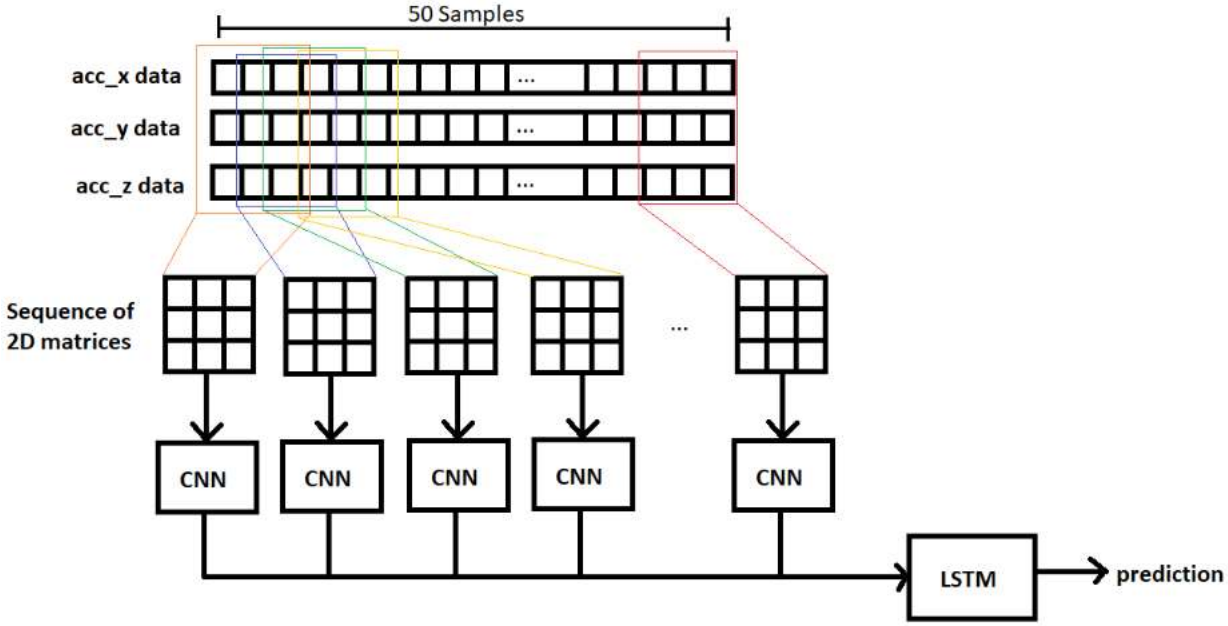
1. We are using heavy file I/O for testing: every iteration, the DSP driver C process saves the last 50 values in a file. The Python program then detects the file and operates on it. The Python program then deletes the file. File I/O in general is very slow
2. We are using a python program to run the classifications, which is slower than C.
3. Models such as CNN+LSTM is relatively complex.

Appendix C: Default LSTM Model

The default LSTM is a basic LSTM model. See code for more details.

Appendix D: Default CNN+LSTM Model

The default CNN+LSTM is shown in the following architecture:



The 50 samples are windowed at 3 samples per window. Each window is fed to a CNN layer. The CNN outputs are then fed into the LSTM to make the prediction.