DGMD S-14 Wearable Device

Project: Head Position Identifier

Wei Wen & Zhengyuan Liu

1. Introduction

1.1 System Mission

One of the modern-day issues with white collar workers and at a growing proportion of occurrence is neck pain and neck spasm. Neck pain and neck spasm is defined as involuntary contraction of the muscles of your neck. The neck muscles get tightened, froze up, often limiting the motion range of your neck. My teammate and I are both suffering from the symptom. Neck pain has an annual prevalence rate exceeding 30% among adults in the US; nearly 50% of individuals will continue to experience some degree of chronic neck pain or frequent occurrences. Among adults, 20% to 70% will experience neck pain that interferes with their daily activities during their lifetime.

The root cause of neck pain and neck spasm is poor posture or stress. Sometimes when we go to bed at night and we wake up with a neck spasm, and the spasm can last for days causing tremendous pain. If left untreated, chronic neck spasm can often lead to spinal abnormalities such as spinal displacement and minor dislocation of the vertebra.

The best exercise for chronic neck spasm is regular exercise. Targeted rehabilitation of neck area muscle is the only way to regain neck functionality. IoT devices that focus on neck pain is trying to address the pain by providing massages and heated treatments, but the key is developing neck area muscle strong enough to support and sustain daily activities.

In short, the device we have in mind is a wearable headband that knows our head and neck position. Our team is trying to accomplish the first part of the rehabilitation machine, which is identifying neck and head position first. Logically when the device knows where our head is, it can determine the motion next and achieve the rehabilitation purpose.

1.2 System Requirements

List of Required Equipment and Materials
1) 1x STMicroelectronics SensorTile kit.
2) 1x STMicroelectronics Nucleo Board.
3) 1x Personal Computer
4) 1x USB 2.0 A-Male to Micro-B Cable (micro USB cable).
5) 1x USB 2.0 A-Male to Mini-B Cable (mini USB cable).
6) Network access to the Internet.

1.3 Recommended Background

Zhengyuan and Wei Wen took 2 introductory Java course and CS50 here in Harvard Extension School prior to taking this class. We know some coding but does not have systematic training in Python as well as data analysis. Machine learning is a completely new concept that we often heard but never learned how to implement. Diving into the class it is our understanding that the class will walk you through how to implement a hardware system that has embedded machine learning functionality.

2. System Design

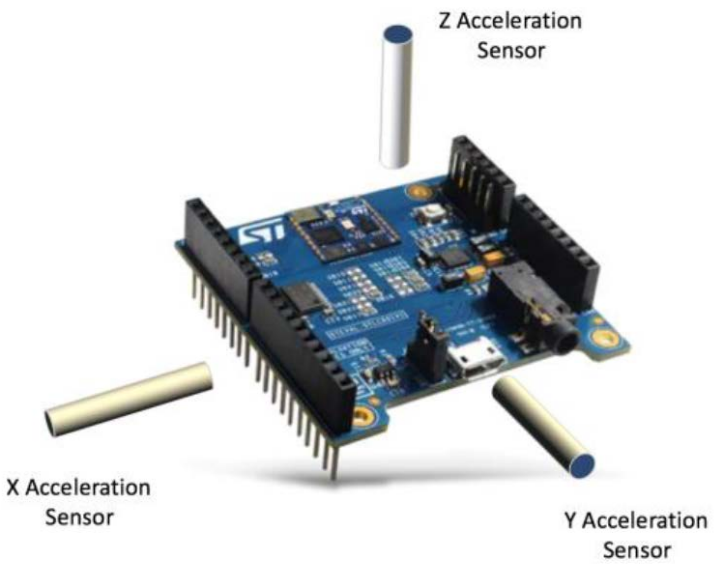


Figure 1: SensorTile axis and orientation, STMicroelectronics Tutorial 11

The project is primarily based on STMicroelectronics SensorTile Tutorial: IoT Machine Learning Motion Classification. Project initial source code is downloaded from: https://drive.google.com/open?id=1IWkwmeVwvyZyM03Y37d-2B6hD2ejfpgM

After successfully downloading the project it is then imported into System Workbench for further customization.

We are designing the system to recognize the orientation of SensorTile when the SensorTile is attached to our head, and recognize turning action. The four actions that we hope to recognize are:

1) Turn Left – Orientation 1
2) Turn Right – Orientation 2
3) Turn Up – Orientation 3
4) Turn Down – Orientation 4

Our original design was to implement the motion recognition by a state machine with 4 states and different threshold values. The advantage of a state machine is that it's easy to implement. Once acceleration on different axis reached certain threshold, a state is recognized.

However, the disadvantage is also apparent: threshold value on different axis might be different for users. By testing it manually we found out that it can be either too sensitive or too insensitive to reach a certain state. At one instance our neck is twisted to an inhumane degree to the left and still the system is recognizing as turning up.

Because individuals have different motion range and head motion patterns, it is best to implement Machine Learning Algorithms here to train the system to implement motion detection to allow some degree of customization. At each instance, users will follow the instructions and poses the “correct”

However, Machine Learning requires complex codes and can be unstable. For example, during one of our iteration we found out after training that even we leave the SensorTile stationary on the table, the recognized output can be different.

Details of the implementation and related code will be in section 4: implementation. Please note that we tried to understand and provide explanation for the related source code from the tutorial and added some degrees of customization.

3. Sensor Data Analysis and Feature Extraction

The default position of the system will be SensorTile facing the sky and lying on the table.

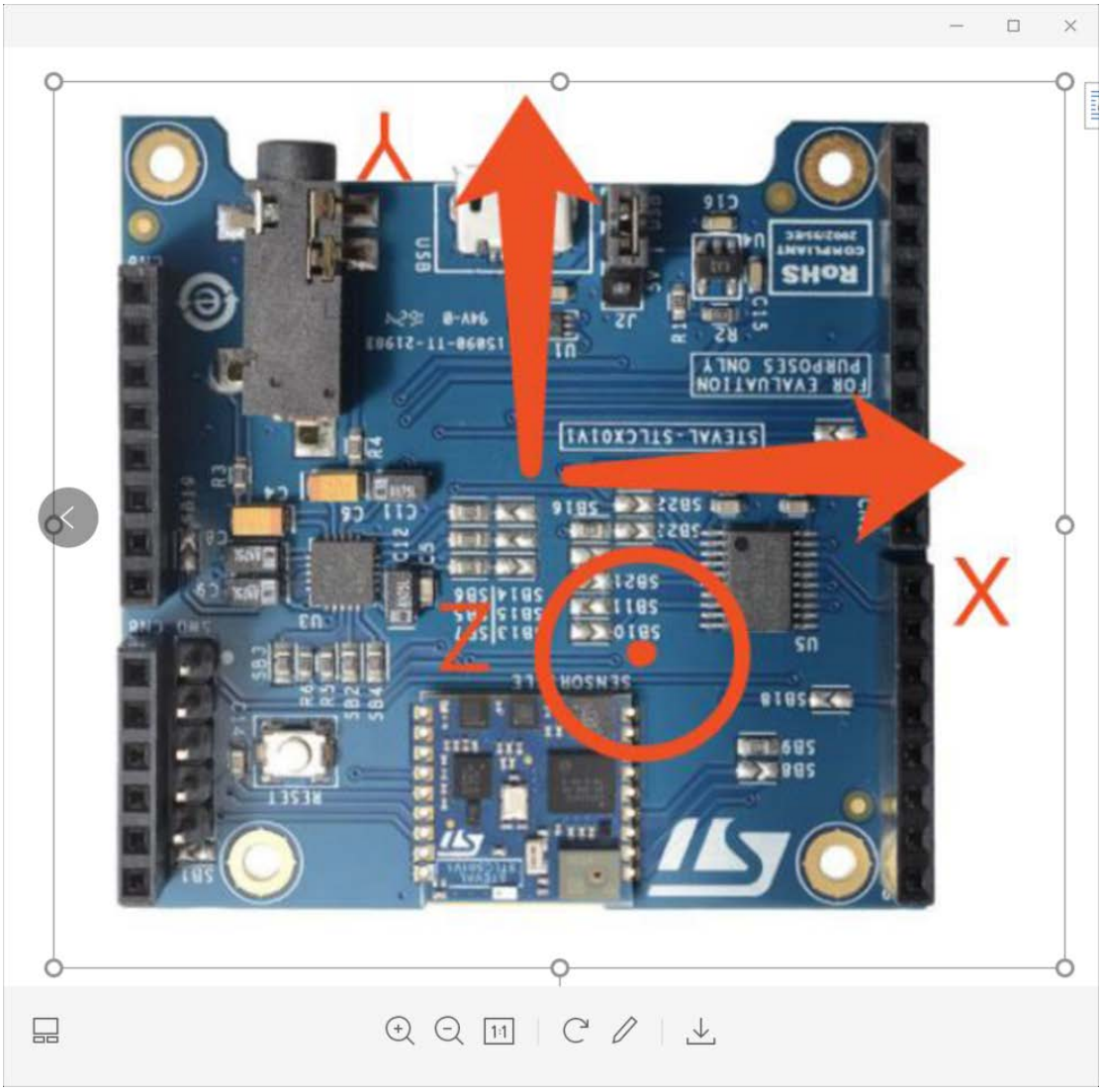


Figure2: SensorTile default position.

We expect each position to follow the following metrics

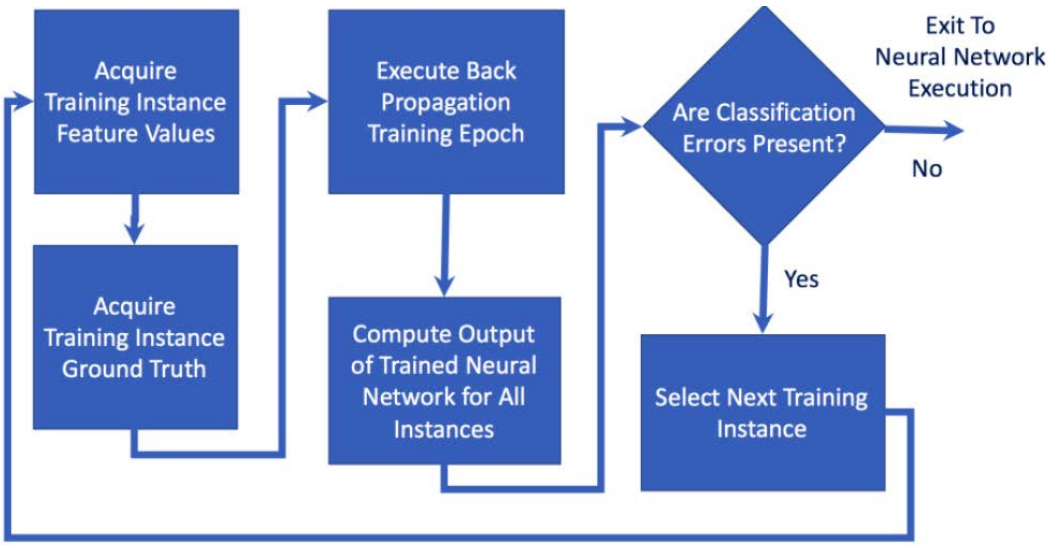
Position	X	Y	Z
Left	positive	Positive	positive
Right	Positive	Negative	positive
Up	Positive	Positive	Positive
Down	Positive	Positive	Negative

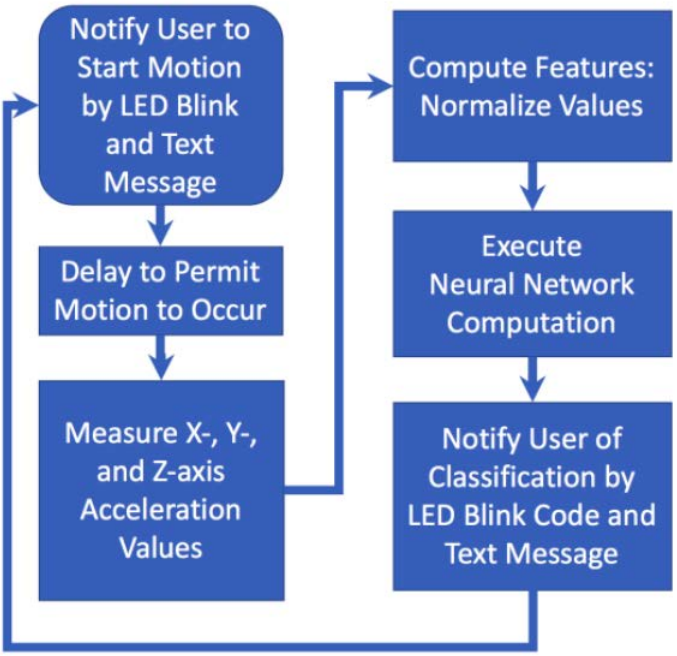
4. Implementation

System is activated when double tapped. After activation, user is prompted to go through training session and acquire necessary training data ("ground truth"). Please note that during the data acquisition we have to come back to the default state after logging each movement data.

4.1 System Execution Outputs

System flowchart: training + testing





Source: STMicroelectronics tutorial 11

See figure below for implementation details:

Step 1: Initial start of the System

EmbeddedML Single State Orientation Classification

Double Tap to start training

Step 2: log data: follow the instruction to move to different orientations, left, right, up and down

```
Training Start in 2 seconds ..
Move to Start Position - Wait for LED On
Turn Left when the LED is On
Start Motion to new Orientation and hold while LED On
 Motion Complete
Accel 2115180
 Softmax Input          211     51      80
 Softmax Output         91      22      34
```

Softmax output is the actual value times 100 to avoid confusion and show normalization.

Step 3: training

```
Training Epochs: 0

State 0   Max 140  Mean 136          Z-score 98          Outputs  140   131   138   136   Cla
State 0   Max 0    Mean 0    Z-score 0 Outputs  -121    -115     -119  -118  Classification Error
State 0   Max 0    Mean 0    Z-score 0 Outputs  136     130      134   132   Classification Error
State 0   Max 0    Mean 0    Z-score 0 Outputs  -104    -80      -88   -82   Classification Error
Error State: 1


Training Epochs: 20

State 0   Max 112  Mean 99   Z-score 62          Outputs  112   67    109   108   Classificatior
State 1   Max 35   Mean 0    Z-score 125         Outputs  -33   35    -10   5
State 0   Max 0    Mean 0    Z-score 0 Outputs   119      107   117   113   Classification Error
State 3   Max 59   Mean 28   Z-score 142         Outputs  9     25    20    59
Error State: 1


Training Epochs: 40

State 0   Max 106  Mean 69   Z-score 52          Outputs  106   -37   104   103   Classificatior
State 1   Max 73   Mean 22   Z-score 140         Outputs  -9    73    6     20
```

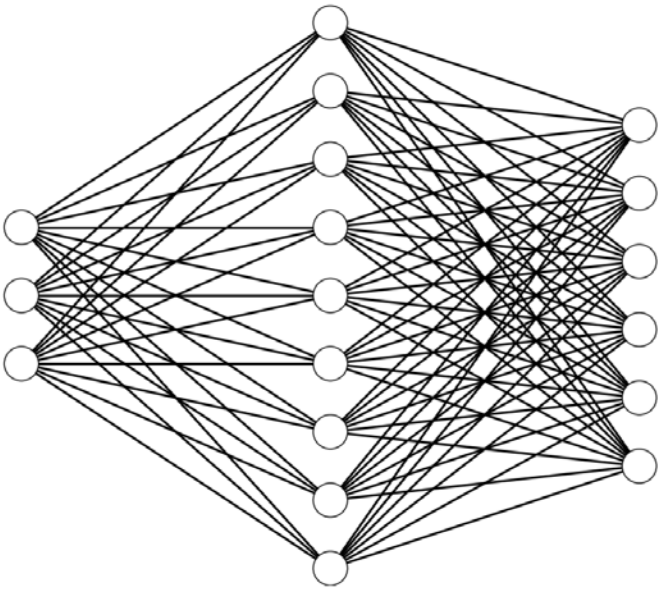
Step 4: testing

```
Move to Start Position - Wait for LED On
Start Motion to new Orientation and hold while LED On
 Motion Complete
 Softmax Input:    110     -289    -1507
 Softmax Output:   7       -18     -97

Look Down

Move to Start Position - Wait for LED On
Start Motion to new Orientation and hold while LED On
 Motion Complete
 Softmax Input:    216     16      20
 Softmax Output:   99      7       9

Turn Left

Move to Start Position - Wait for LED On
Start Motion to new Orientation and hold while LED On
 Motion Complete
 Softmax Input:    -175    8       -55
 Softmax Output:   -95     4       -29
```

After the inputs are recorded, the system will train each input for 2000 epochs.

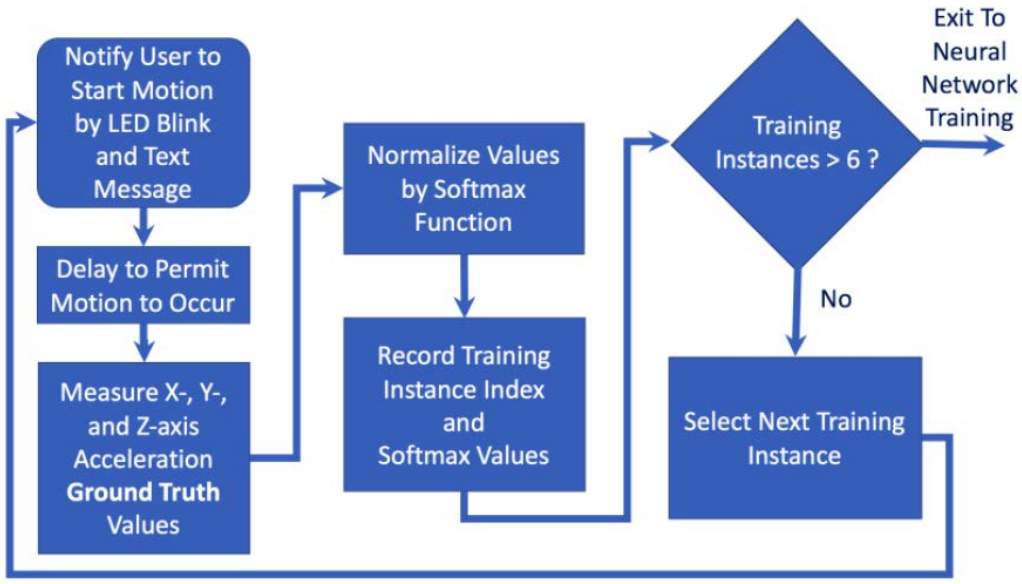
4.2 Code Implementation:

We used SensorTile embeddedML.c Embedded Machine Learning Library by Charles Zaloom to implement the training sessions. We understand that the neural network will contain 3 inputs, one hidden layer with 9 neurons and 6 neurons in the Output Layer.



Source: STMicroelectronics Tutorial 11

Overall Training flowchart:



Source: STMicroelectronics Tutorial 11

4.2.1 Data Acquisition：

Three features are required to be supplied to the three input neurons. Here we used the following input values as input neurons:

```
  /*
   * Compute feature values including only X Y Z axes and
   *
   */

  *ttt_1 = ttt[0] - ttt_initial[0];
  *ttt_2 = ttt[1] - ttt_initial[1];
  *ttt_3 = ttt[2] - ttt_initial[2];
  *ttt_mag_scale = (int)(accel_mag);

  return;
```

We can see that these features are the differences between the X Y and Z axis accelerations, respectively, between a final acceleration value when motion is complete and an initial value. These values are extracted and included in **`Feature_Extraction_State_0`** function.

When extracting acceleration differences between initial and final state, we used **`getAccel()` function.**

```
void getAccel(void *handle, int *xyz) {
	uint8_t id;
	SensorAxes_t acceleration;
	uint8_t status;

	BSP_ACCELERO_Get_Instance(handle, &id);

	BSP_ACCELERO_IsInitialized(handle, &status);

	if (status == 1) {
		if (BSP_ACCELERO_Get_Axes(handle, &acceleration) ==
COMPONENT_ERROR) {
			acceleration.AXIS_X = 0;
			acceleration.AXIS_Y = 0;
			acceleration.AXIS_Z = 0;
		}

		xyz[0] = (int) acceleration.AXIS_X;
		xyz[1] = (int) acceleration.AXIS_Y;
		xyz[2] = (int) acceleration.AXIS_Z;
	}
}
```

We can see acceleration on different axis is stored as xyz[] dataset. However, we need to perform additional smoothing to generate training data.

4.2.2 Data Cleaning:

EmbeddedML.c only takes in input value of -1.0 to 1.0. Hence we need to cleanup and normalize the input value.

Normalization occurs in **`motion_softmax()`** function in the following sequence:

1) Store getAccel value in XYZ[] array

```
XYZ[0] = (float) ttt_1;
XYZ[1] = (float) ttt_2;
XYZ[2] = (float) ttt_3;
```

2) Call motion_softmax function to cleanup and normalize x, y, z acceleration difference data ttt_1, ttt_2 and ttt_3

```
motion_softmax(net->topology[0], XYZ, xyz);
```

```
void motion_softmax(int size, float *x, float *y) {
      float norm;

      norm = sqrt((x[0] * x[0]) + (x[1] * x[1]) + (x[2] * x[2]));
      y[0] = x[0] / norm;
      y[1] = x[1] / norm;
      y[2] = x[2] / norm;

}
```

Result is stored back into xyz[] array. xyz[] array now contains value of only -1.0 to 1.0.

Training data stored for training purpose:

```
training_dataset[i][k][0] = xyz[0];
training_dataset[i][k][1] = xyz[1];
training_dataset[i][k][2] = xyz[2];
```

in this instance, i is the corresponding positional cases (in this case we have 4 position case, up, down, left and right). K represents number of training data cycles. We set number of training cycle equals 1.

Training data is now ready for ANN training.

4.2.3 ANN Training using embededML.c

The training capability is provided by embededML.c. Although we used out of the box training set up, we modified the code to make sure that we can customize our training to certain extent.

Here we called `train_ann()` function and provided training_dataset. Note that the training dataset is stored by different position instances (i.e left, right etc.)

```
            switch (j) {

            case 0:
                train_ann(net, training_dataset[j][k], _Motion_1);
                break;
            case 1:
                train_ann(net, training_dataset[j][k], _Motion_2);
                break;
            case 2:
                train_ann(net, training_dataset[j][k], _Motion_3);
                break;
            case 3:
                train_ann(net, training_dataset[j][k], _Motion_4);
                break;

            default:
                break;
            }
            i++;
            HAL_Delay(5);
        }
```

```
void train_ann(ANN *net, float *input, float *output){
    float delta[net->topology[1]];
    BP_ANN(net, input, output, net->weights, net->dedw, net->bias, delta,
net->n_layers-1);
}
```

With `train_ann()` function, output array is stored `_Motion_1 to _Motion 4.` `training_dataset` corresponding to each position instance is feed into the function.

The ML training model is back propagation. "In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually." Basically backpropagation means we train the neurons to have specific weights in order to arrive at the final "ground truth" value with the given input value. Once the training is complete, the weights will be modified. We can see in the BP_ANN() function that weights are computed and modified several times per instance.

```
//-----ANN-----
void BP_ANN(ANN *net, float *input, float *output, float *weights, float *velocity, float *bias, float *delta, int depth)
    unsigned int i,j;
    unsigned int DIM[2] = {net->topology[net->n_layers - depth], net->topology[net->n_layers - depth - 1]};

    if(depth == 1){
        for(i = 0; i < DIM[0]; i++){
            net->output[i] = 0.0;
            for(j = 0; j < DIM[1]; j++){
                net->output[i] += weights[(DIM[1]*i)+j]*input[j];
            }
            net->output[i] = net->output[i] + bias[i];
            delta[i] = (output[i]-net->output_activation_function(net->output[i])) * net->output_activation_derivative(ne
            net->output[i] = net->output_activation_function(net->output[i]);
            bias[i] = bias[i] + delta[i]*net->beta;
        }

        float dEdW[DIM[0]*DIM[1]];
        for(i = 0; i < DIM[0]; i++){
            for(j = 0; j < DIM[1]; j++){
                dEdW[(DIM[1]*i)+j] = delta[i]*input[j];
            }
        }
        for(i = 0; i < DIM[0]*DIM[1]; i++){
            velocity[i] = dEdW[i]*net->eta - velocity[i]*net->alpha;
            weights[i] = weights[i] + velocity[i];
        }
        return;
    }
```

Several key metrics used in BP_ANN function is listed herein:

```
//---EMBEDDED ANN---
float weights[81] = { 0.680700, 0.324900, 0.607300, 0.365800, 0.693000,
                      0.527200, 0.754400, 0.287800, 0.592300, 0.570900,
0.644000,
                      0.416500, 0.249200, 0.704200, 0.598700, 0.250300,
0.632700,
                      0.372900, 0.684000, 0.661200, 0.230300, 0.516900,
0.770900,
                      0.315700, 0.756000, 0.293300, 0.509900, 0.627800,
0.781600,
                      0.733500, 0.509700, 0.382600, 0.551200, 0.326700,
0.781000,
                      0.563300, 0.297900, 0.714900, 0.257900, 0.682100,
0.596700,
                      0.467200, 0.339300, 0.533600, 0.548500, 0.374500,
0.722800,
                      0.209100, 0.619400, 0.635700, 0.300100, 0.715300,
0.670800,
                      0.794400, 0.766800, 0.349000, 0.412400, 0.619600,
0.353000,
                      0.690300, 0.772200, 0.666600, 0.254900, 0.402400,
0.780100,
                      0.285300, 0.697700, 0.540800, 0.222800, 0.693300,
0.229800,
                      0.698100, 0.463500, 0.201300, 0.786500, 0.581400,
0.706300,
                      0.653600, 0.542500, 0.766900, 0.411500};
       float dedw[81];
       float bias[15];
       unsigned int network_topology[3] = { 3, 9, 4 };
       float output[4];

       ANN net;
       net.weights = weights;
       net.dedw = dedw;
       net.bias = bias;
       net.topology = network_topology;
       net.n_layers = 3;
```

```
net.n_weights = 81;
net.n_bias = 15;
net.output = output;
```

These metrics are all stored in net data structure.

There are 81 present weights but we did not used the entirety of 81 weights. 81 weights correspond to 3 input, 9 neurons, and 6 output. However, we only have 4 outputs in our case, so the correct weight should be 63 (27 + 36). In order to preserve the integrity of the model, and because the weight not used is not effecting the accuracy of the ANN model, we didn't change the metric.

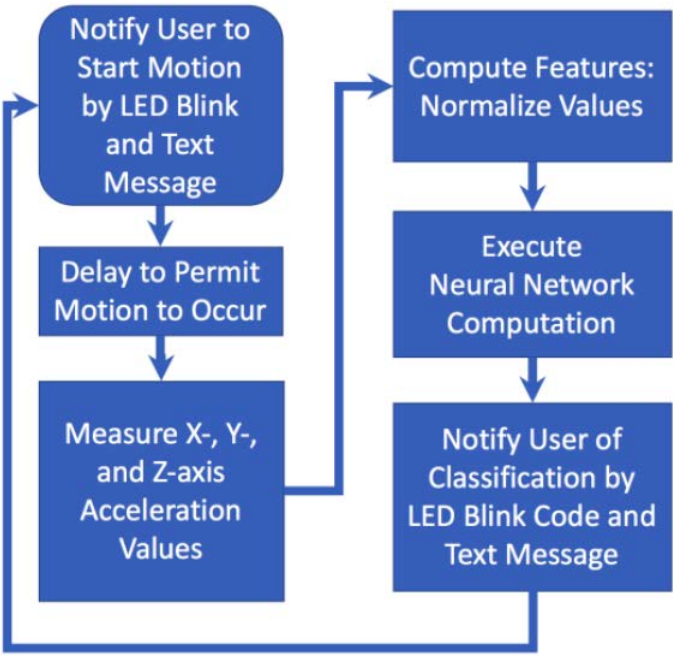
We set ouput to 4 outputs and network_topology to `{ 3, 9, 4 }`.

Neural Network model is ready to use for testing purposes. Weights are balance towards classifying the correct motion based on training data collected. Note that the training epochs are set to maximum of 2000 instances. If the model is ready before 2000, the training will stop.

4.2.4 Testing

The model is now ready for testing, meaning that given 3 unknown inputs, the model will go through 9 neurons to correctly classify the action to the 4 correct orientations of the system.

Testing flowchart:



Source: STMicroelectronics tutorial 11.

Similar to the training process, it first requires the system to capture accelerometer

data using Accel_Sensor_Handler() function. Feature will be extracted and normalized with the following code using `Feature_Extraction_State_0() and motion_softmax()` function.

```
Feature_Extraction_State_0(handle, &ttt_1, &ttt_2, &ttt_3,
                           &ttt_mag_scale);
XYZ[0] = (float) ttt_1;
XYZ[1] = (float) ttt_2;
XYZ[2] = (float) ttt_3;

motion_softmax(net->topology[0], XYZ, xyz);
```

Normalized data is then feed into `run_ann()` function.

```
run_ann(net, xyz);
```

```
void run_ann(ANN *net, float *input){
    FP_ANN(net, input, net->n_layers-1, net->weights);
}

void FP_ANN(ANN *net, float *input, unsigned int depth, float *weights){
    unsigned int DIM[2] = {net->topology[net->n_layers - depth],
net->topology[net->n_layers - depth - 1]};
    unsigned int i,k;

    if(depth == 1){
        for(i = 0; i < DIM[0]; i++){
            net->output[i] = 0.0;
            for(k = 0; k < DIM[1]; k++){
                net->output[i] += weights[(DIM[1]*i)+k]*input[k];
            }
            net->output[i] = net->output_activation_function(net->output[i]
+ net->bias[i]);
        }
        return;
    }
    else{
        float a[DIM[0]];
        for(i = 0; i < DIM[0]; i++){
            a[i] = 0.0;
            for(k = 0; k < DIM[1]; k++){
                a[i] += weights[(DIM[1]*i)+k]*input[k];
            }
            a[i] = net->hidden_activation_function(a[i] + net->bias[i]);
            //if(depth == 2) printf("%f,", a[i]);
        }
        //if(depth == 2) printf("0\n");
        FP_ANN(net, a, depth-1, &weights[DIM[0]*DIM[1]]);
    }
    return;
}
```

Computed value is returned stored in variable “loc”.

```
or (i = 0; i < net->topology[net->n_layers - 1]; i++) {
                    if (net->output[i] > point && net->output[i] > 0.1) {
                           point = net->output[i];
```

```
                        loc = i;
                }
        }

switch (loc) {
        case 0:
                sprintf(msg1, "\n\rTurn Left");
                CDC_Fill_Buffer((uint8_t *) msg1, strlen(msg1));
                break;
        case 1:
                sprintf(msg1, "\n\rTurn Right");
                CDC_Fill_Buffer((uint8_t *) msg1, strlen(msg1));
                break;
        case 2:
                sprintf(msg1, "\n\rLook Up");
                CDC_Fill_Buffer((uint8_t *) msg1, strlen(msg1));
                break;
        case 3:
                sprintf(msg1, "\n\rLook Down");
                CDC_Fill_Buffer((uint8_t *) msg1, strlen(msg1));
                break;
```

The testing is now complete and classification of head position should be accurate.

Disclaimer: if it's not accurate, it's due to the limitation of embeddedML.c.

4. Demonstration and Final Comment

Please refer https://youtu.be/AfoU-YRGdCs for demonstration of the system.

The system is successful in predicting the correct head position. However, we still need additional system level development. Our foreseeable next step if time is allowed would be to test more corner cases when head position is not turning but tilting slightly or if the head is limited in motion range because that would be more realistic in nature with neck spasm.

In addition, it would be reasonable to see in the future that we add audio output to the system and give out voice comment to consist of a complete neck and head exercise. The system can give out command and identify if the action is performed by the user.

One final comment is that we are impressed by the SensorTile system and its capability of performing machine learning algorithm and training on a small equipment. Prior to this course my understanding was that you need a powerful GPU to perform any machine learning. However, this course taught us that machine learning is not that complicated. We will definitely continue to explore machine learning and edge computing in the future.

We are a two-person team with equal contribution to the project. Zhengyuan and I held weekly online meetings where we research the project together. We had similar coding background and Zhengyuan has better coding skills. He implemented majority of the customization and I was responsible in researching and

understanding the codes. We believe this course is above our capability because we did not have coding project and c language experience. This is also the first hardware course for both of us and we feel the level of difficulty is sometimes too hard. However we were able to overcome the difficulties and implemented our first machine learning project.