Eslab Final Project Report

Fall Detection

電機四 陳元芃 B05901038
電機四 孫鍾恩 B05901071
電機四 曹峻豪 B05901164

1. Motivation

Falls in the elderly have always been a serious medical and social problem. To detect and predict falls, and further more roughly determine what movement type currently is, a hidden Markov model (HMM)-based method using tri-axial accelerations of human body is proposed.

2. Implementation

(a) Data Collection

We used BSP_B-L475E-IOT01 library to read acceleration from the built-in tri-axial accelerometer of STM32L475 MCU. In convenience, we calculated the magnitude of accelerations instead of using the 3-dimensional data straightforwardly. After attaching STM32 to the participant, we collected acceleration data of 4 different movement types (predict-to-fall, falling, running, walking), which are then fed into 4 different HMM models for training respectively. Detailedly, we collect 95 pieces of training data, 31 of predict-to-fall ("predict" hereafter), 31 of falling, 17 of running and 16 of walking, by asking our participant to do the instructed movement. Each piece of data is T=1.5 second long, and is sampled with sample rate f_s=100Hz. That is, each piece has 150 values.

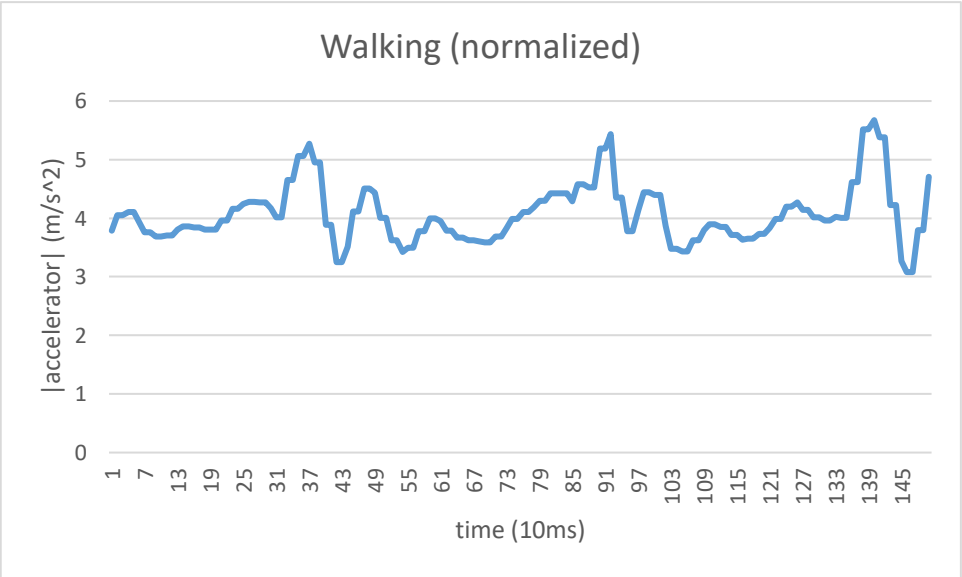


Fig 1. Example of walking acceleration data.

(b) Training of HMM Models

We use discrete HMM models by first quantizing the acceleration data into 8 levels. And the levels are designed as follows:

[0, 1.1)
[1.1, 2.2)
[2.2, 3.4)
[3.4, 4.7)
[4.7, 6.6)
[6.6, 8.7)
[8.7, 11)
[11, ∞)

Note that the data are normalized such that the default value when the participant is not moving at all is 4m/s^2.

After quantizing data, we can feed 4 different kinds of movement data into 4 corresponding HMM models, and then Baum-Welch algorithm is used for training. Detailedly, each model has 3 hidden states, and 8 possible observation values. After training, we got for each model a 3-by-3 state transition matrix, an 8-by-3 observation matrix, and a 3-dimensional initial probability array. The training process is done on a personal computer, and then the trained parameters are copied-and-pasted to the .cpp file in STM32 in order to save computational time.

```
double run_pi[3] = {5.23E-193,1,8.79E-132};
double run_a[3][3] = {{0.911397594,0.004520484,0.117142582},{0.016535984,0.928022794,0.105564851},{0.0
double run_b[8][3] = {
{0.376659084,2.89177E-36,7.32707E-22},
{0.623340916,0.0000000000177971,0.000132496},
{4.75775E-11,2.94127E-19,0.579184439},
{1.64936E-22,5.24812E-05,0.420680016},
{3.64058E-23,0.329060907,3.04952E-06},
{2.42714E-31,0.497695775,2.88081E-15},
{5.99981E-26,0.173190837,6.12921E-17},
{0,0,0}};
```

Fig 2. Model parameter declaration in main.cpp of STM32

(c) Viterbi Algorithm

With the parameters, we could run Viterbi algorithm on STM32 to determine which movement type has the highest probability at current time. Detailedly, we manually design a fifth HMM model corresponding to the movement “still” to represent that the participant is barely moving. Then for each T=1.5 second acceleration sequence, we can feed it to 5 models and run Viterbi algorithm to yield the probability of each movement, and then the movement with maximum probability is declared. It is noteworthy that since we haven’t found any existing package for Viterbi algorithm in c++, we write the code of Viterbi algorithm by

ourselves.

```
void viterbi(HMM* hmm, int seqlen, int* seq, int model_num, double* p) {
    //printf("calling viterbi\n");
    for(int i=0;i<model_num;i++){
        double viterbi[hmm[i].state_num][seqlen];
        for(int j=0;j<hmm[i].state_num;j++){
            viterbi[j][0]=hmm[i].initial[j]*hmm[i].observation[seq[0]][j];
        }
        for(int k=1;k<seqlen;k++){
            for(int l=0;l<hmm[i].state_num;l++){
                double max=0;
                for(int m=0;m<hmm[i].state_num;m++){
                    if(viterbi[m][k-1]*hmm[i].transition[m][l]>max){
                        max=viterbi[m][k-1]*hmm[i].transition[m][l];
                    }
                }
                viterbi[l][k]=max*hmm[i].observation[seq[k]][l];
            }
        }
        for(int a=0;a<hmm[i].state_num;a++){
            if(viterbi[a][seqlen-1]>p[i]){
                p[i]=viterbi[a][seqlen-1];
            }
        }
```

Fig 3. Viterbi algorithm realized by ourselves.

(d) Sliding Window

For a general acceleration sequence that is typically longer than 1.5 second, we use a sliding window with T=1.5 second and perform the abovementioned algorithm in the extracted subsequence to determine the movement type at current time, and then the sliding window moves forward for stride $T_s=f_s^{-1}$=10ms to repeat the algorithm again. The procedure keeps going on, and we can determine the movement type at every time instant.

In practice, since STM32 has finite memory, we cannot store all the acceleration value at all the time instants. Hence we create a buffer with buffer_size=450, which is slightly larger than window_size=150. There is an iterator such that the acceleration magnitude extracted by STM32 is stored to the corresponding buffer entry pointed by the iterator, and then the iterator is incremented by 1. If the iterator is near the end of the buffer, it assigns acceleration magnitude not only to the current entry but also to the entry near the beginning of the buffer. In this way when the iterator reaches the end, it returns back to the first possible position, and Viterbi algorithm can immediately be performed since the values has been copied

before. The following is an illustrated example with buffer_size=6 and window_size=3.

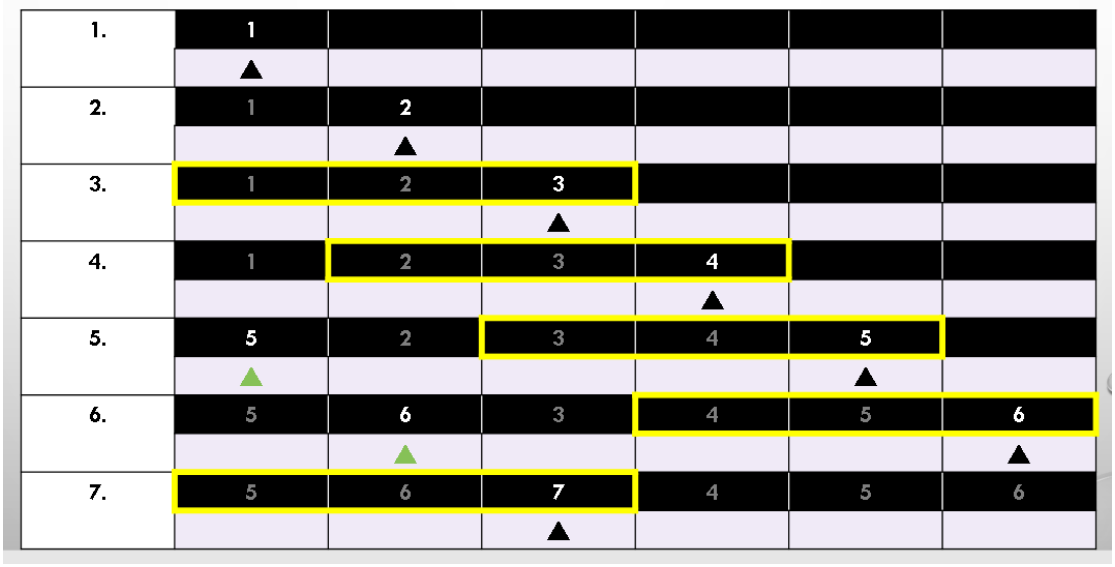


Fig 4. An example of realization of sliding window. When the iterator (black triangle) is near the end (5. 6. iteration), entries near the beginning (green triangle) is updated too. Then in iteration 7. Viterbi algorithm can be performed immediately.

The key point is that at each time instant there is only 1 or 2 entry update, keeping the runtime nearly the same, which is a crucial point in real-time programming.

(e) Denoising

Our previous implementation has several drawbacks, one of which is there are undesirable noise.

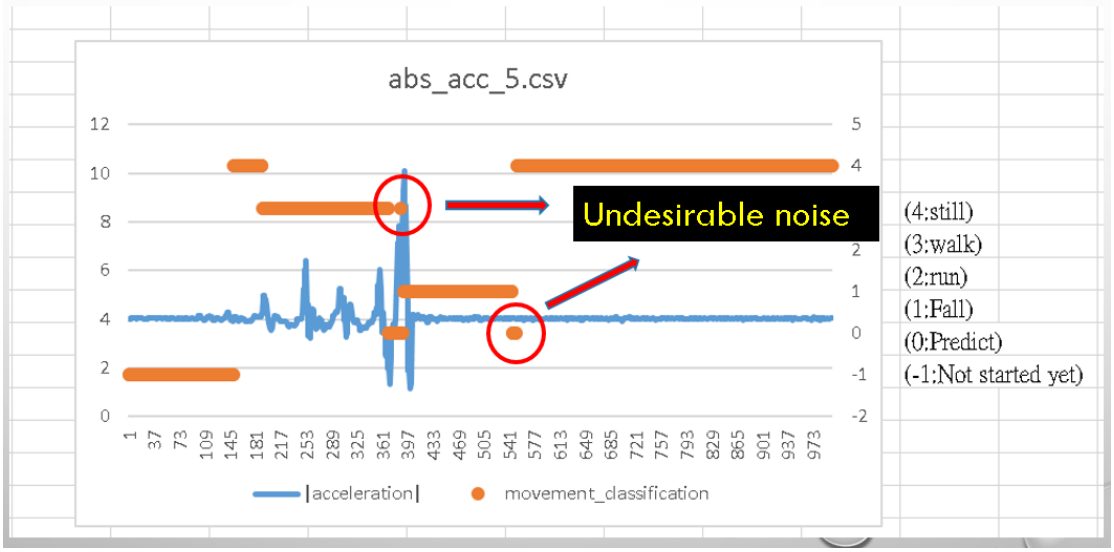


Fig 5. An example of previous result with undesirable noise.

It is reasonable that a movement will keep for an adequate amount of time, and hence a movement that lasts not long enough (0.2 second by our setting) can be viewed as noise and should be smoothened. In practice we maintain a state machine to ensure that current state would be changed if and only if the state outcomes of 20 values are all equal contiguously.

```
int len;
int output_size = 20;
output_buf.push_back(argmax);
if(output_buf.size()==output_size){
    int count=0;
    for(int i=0;i<output_size;i++){
        if(output_buf[i]==argmax){
            count++;
        }
    }
    if (count==output_size){
        if(current_state!=argmax){
            current_state = argmax;
            len = sprintf(acc_json,"%d",argmax);
            cout<<movement_arr[argmax].model_name<<endl;
            socket.send(acc_json,len);
        }
    }
    output_buf.pop_front();
}
```

Fig 6. Source code of the denoising part the and data sending part.

(f) Socket Server (Rpi side)

We also implemented a WiFi socket server-client system to send the result from STM32 to Rpi, which is an application of wireless communication taught in class. Wifi-ISM43362 library was added to STM32 to realize the utility. To prevent from constantly sending data which might increase the runtime a lot, STM32 sends movement only when the movement is changed. The received movement type will be printed on Rpi's terminal, and when falling is predicted/detected, the LED connected to Rpi will turn yellow/red to indicate the fall is predicted/detected, respectively. It is noteworthy that Rpi only deals with the received movement type, and nearly all the calculation are done on STM32, showing the potential calculation ability of STM32 that may be used for a variety of further application.

Consequently, find out the user's movement type. We also implemented websocket to send the result from STM32 to Rpi. Wifi-ISM43362 library was added to STM32 to realize the utility. When fall is detected, the LED connected to Rpi will turn red to indicate the fall is detected.

(g)

3. Result

The result is as we expected. It could predict and detect fall with good precision. The following is the final result on a piece of our pre-collected data, which satisfyingly detect the movement change from “still”, “walk”, “predict”, “fall”, to “still”.

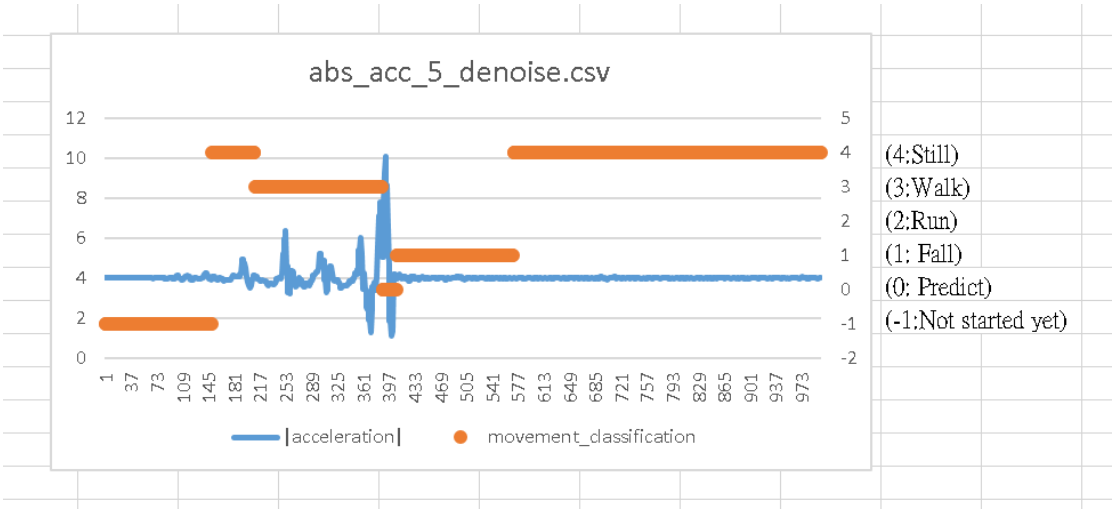
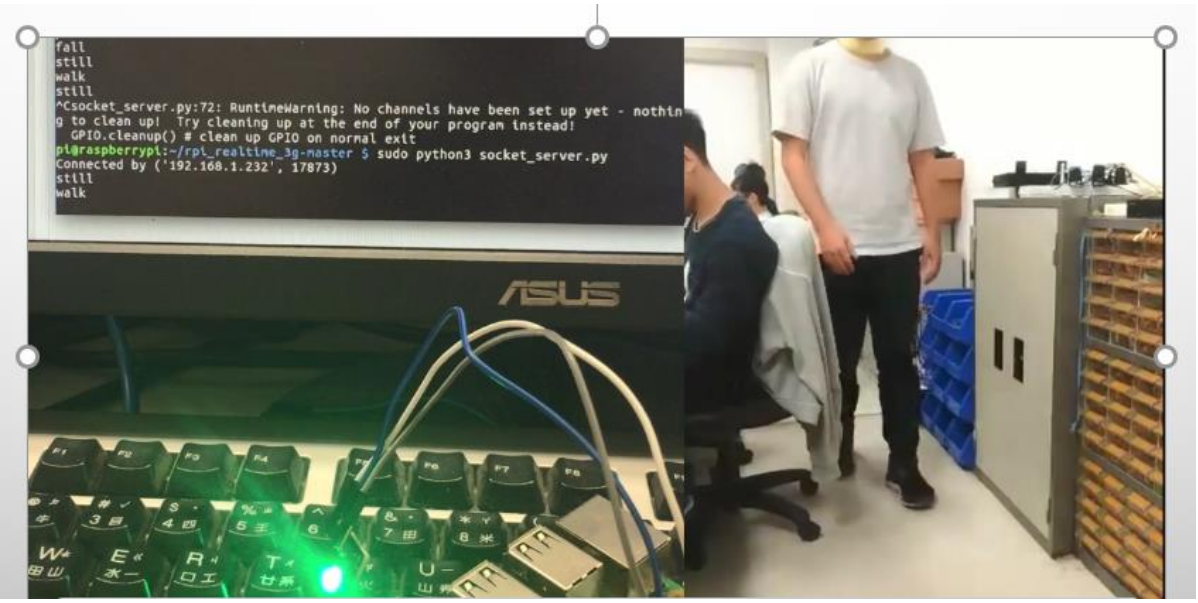


Fig 7. An example of final result of pre-collected data.

As for real-time demo, whenever our participant walks and then falls, our device can always determine the correct movement types “still”, “walk”, “predict” and then “fall”, within some inevitable yet acceptable Internet delay.



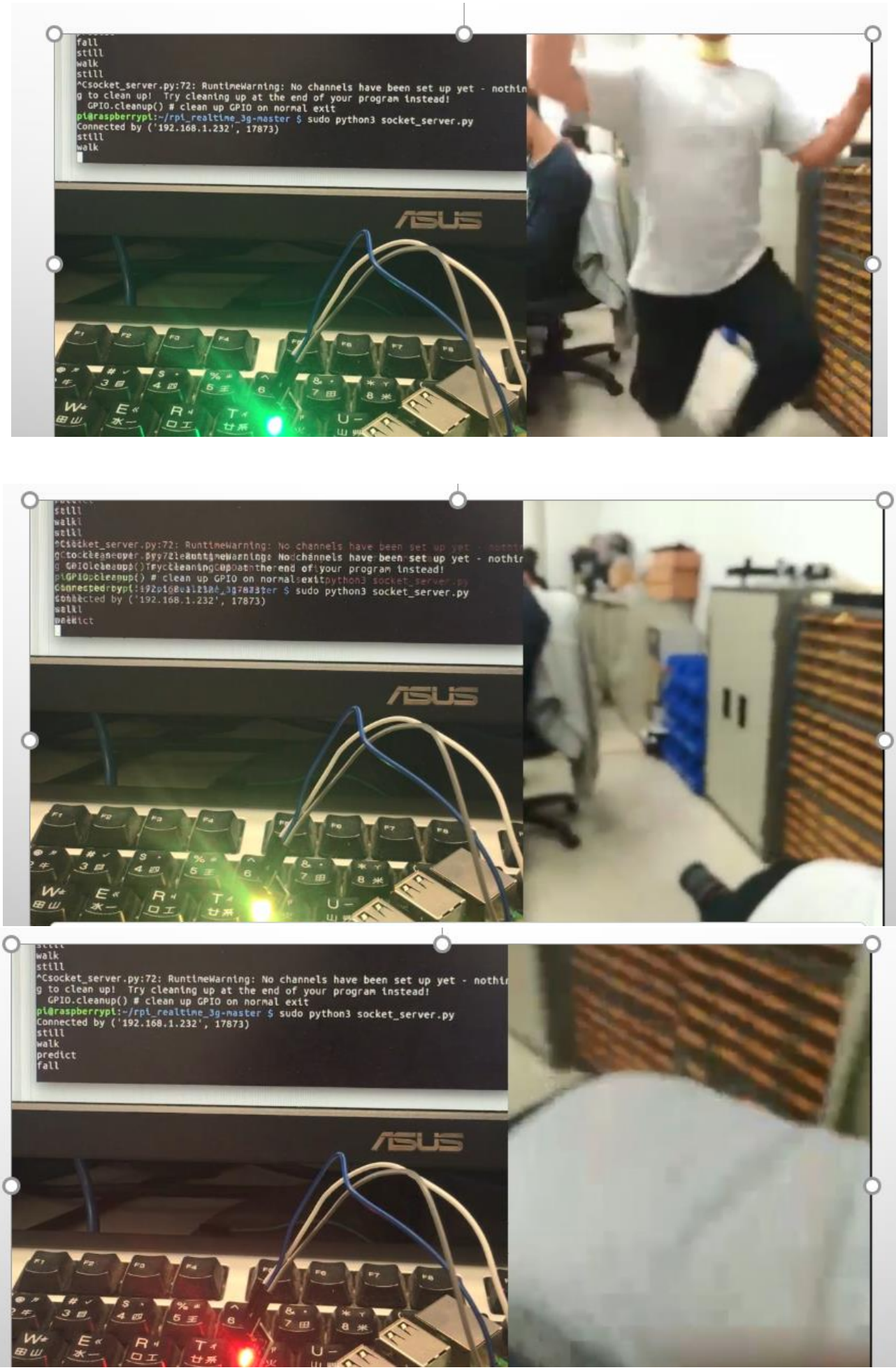


Fig 8. A real-time demo when the participant starts to walk and then fall. The participant's movement is shown in the right, and the simultaneous output of Rpi is shown in the left.

However, there is still room for improvement. We failed to accomplish is that when determining the running states, the output often will have false alarm before it. For example, while we start running, it will first output “predict” and “fall” for a while. We belief it's mainly because the data we collected while starting running is too similar to the running data, therefore, leading to false alarm.

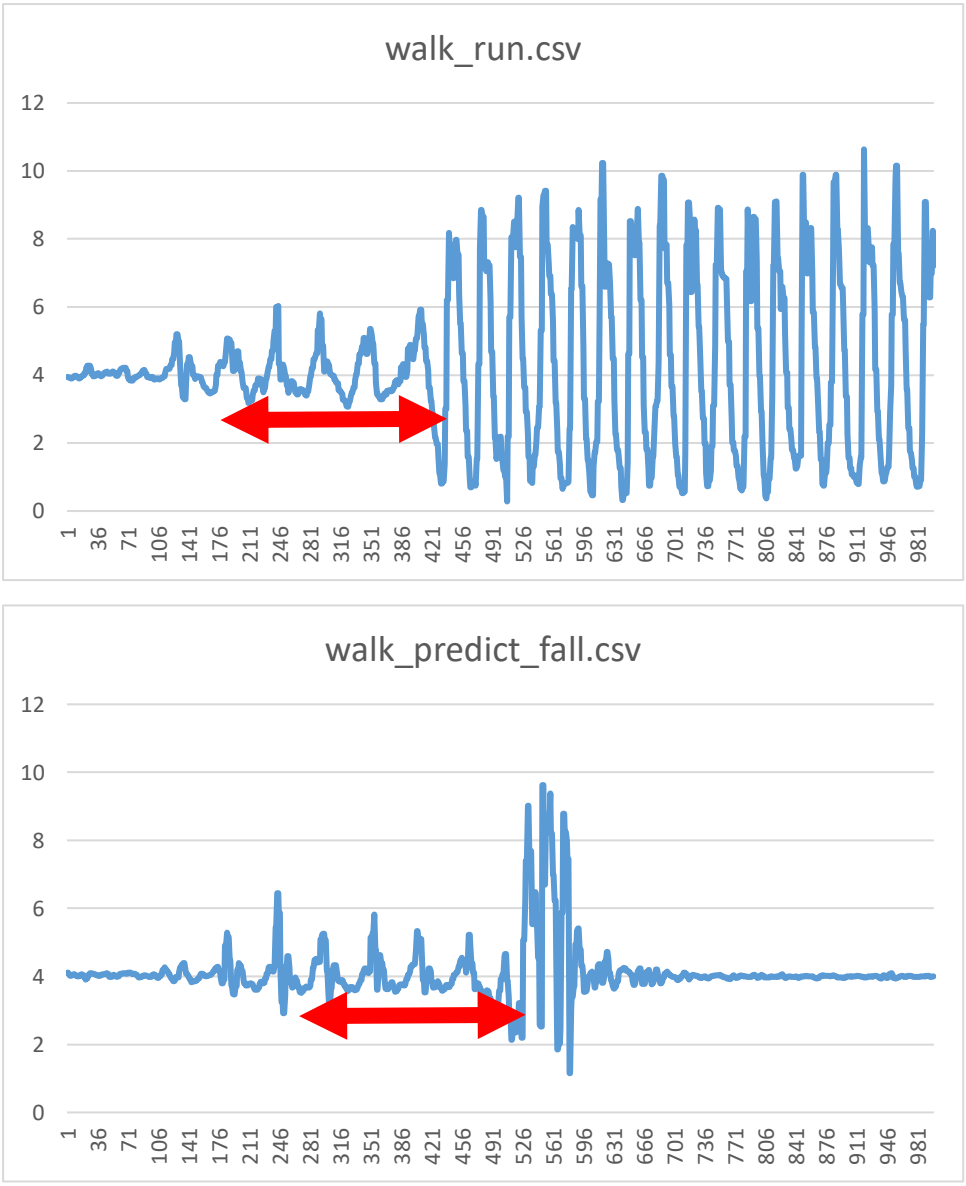


Fig 9. A piece of running data and a piece of data consisting of walking, predicting, and then falling. Note that the patterns in the two sliding window (red arrow) are quite similar.

4. Future Work

Since the movement detection is not entirely prefect now, the future work will be focused on pursuing better precision of the result. Implementing Kalman

filter or re-train the model with better dataset or with resized window size are efforts we can be working on. Furthermore, although we have made the prediction, we can only predict but couldn't do anything after that. Hence, developing a protection gear to take action after prediction is also a subject that we concern.

5. Reference

[1] L. Tong, Q. Song, Y. Ge and M. Liu, "HMM-Based Human Fall Detection and Prediction Method Using Tri-Axial Accelerometer," in IEEE Sensors Journal, vol. 13, no. 5, pp. 1849-1856, May 2013.

[2] Lim, Dongha & Park, Chulho & Kim, Nam & Kim, Sang-Hoon & Yu, Yun Seop. (2014). Fall-Detection Algorithm Using 3-Axis Acceleration: Combination with Simple Threshold and Hidden Markov Model. Journal of Applied Mathematics. 2014. 10.1155/2014/896030.