HUMAN ACTIVITY RECOGNITION USING WEARABLE INERTIA SENSOR DATA AND MACHINE LEARNING

by

Xiaoyu Yu

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

School of Engineering Technology

West Lafayette, Indiana

August 2019

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF COMMITTEE APPROVAL

Dr. Suranjan Panigrahi, Chair

School of Electrical Engineering Technology

Dr. Frederick Berry

School of Mechanical Engineering Technology

Professor Robert J. Herrick

School of Electrical Engineering Technology

Approved by:

Dr. Duane D. Dunlap

Head of the Graduate Program

ACKNOWLEDGEMENTS

I would like to express my thank Dr. Suranjan Panigrahi, my advisor, for his constant support and guidance. I acknowledge my thesis committee members, Professor Robert J. Herrick and Dr. Frederick Berry, for their insightful comments and guidance. Also, I want thank lab partners, Ke Xu and Ridhi Deo for their helps and advises with my work. I want to thank the staffs in School of Engineering Technology and the staffs from IRB office at Purdue, for their direct and indirect helps to my work. Finally, I want to thank my parents for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HAR:	Human Activity Recognition
ADL:	Activity of Daily Livings
SVM:	Support Vector Machine
ANN:	Artificial Neural Network
CNN:	Convolution Neural Network
MTF:	Markov Transition Fields
RNN:	Recurrent Neural Network
LSTM:	Long-Short Term Memory
PCA:	Principle Component Analysis
ReLU:	Rectified Linear Units

GLOSSARY

Training data: The data that are used to train classification models.

Validation data: The data that are used to test the performance of the classification model during the training process. Validation data are used to fine tune parameters in the classification model and the data should not be part of the training data.

Testing data: The data that are used to test the performance of the trained classification model (after training process). Testing data are used to evaluate the final performance of the classification model. Testing data should not be part of training data or validation data. No fine tune should be made based on the result of testing data.

Overfitting: The classification model performs consistently better on training data than on validation data and testing data.

ABSTRACT

Author: Yu, Xiaoyu. MS
Institution: Purdue University
Degree Received: August 2019
Title: Human Activity Recognition using Wearable Inertia Sensor Data and Machine Learning
Committee Chair: Dr. Suranjan Panigrahi

Falling in indoor home setting can be dangerous for elderly population (in USA and globally), causing hospitalization, long term reduced mobility, disability or even death. Prevention of fall by monitoring different human activities or identifying the aftermath of fall has greater significance for elderly population. This is possible due to the availability and emergence of miniaturized sensors with advanced electronics and data analytics tools. This thesis aims at developing machine learning models to classify fall activities and non-fall activities. In this thesis, two types of neural networks with different parameters were tested for their capability in dealing with such tasks. A publicly available dataset was used to conduct the experiments. The two types of neural network models, convolution and recurrent neural network, were developed and evaluated. Convolution neural network achieved an accuracy of over 95% for classifying fall and non-fall activities. Recurrent neural network provided an accuracy of over 97% accuracy in predicting fall, non-fall and a third category activity (defined in this study as "pre/postcondition"). Both neural network models show high potential for being used in fall prevention and management activity. Moreover, two theoretical designs of fall detection systems were proposed in this thesis based on the developed convolution and recurrent neural networks.

CHAPTER 1. INTRODUCTION

1.1 Scope

As the number of elderly population is rapidly growing, the need of palliative care is increasing. As a part of that, human activities are critical to be monitored to help patients and their supporters to provide proper services at the right time. This process is called human activity recognition (HAR) and is widely used in healthcare in various applications including tracking elderly people's daily activities (Chen, Nugent, & Wang, 2012), estimating energy expenditure for obesity prevention (Sazonov, Fulk, Hill, Schutz, & Browning, 2011), monitoring abnormal conditions for cardiac patients (Katoch & Augustyniak, 2012), and also fall detection and intervention for elderly people (Nghiem, Auvinet, & Meunier, 2012). This research focused on developing system to predict undesirable event (i.e., fall down) and to characterize different activities of daily living (ADLs).

The system mainly consisted of two components: data collection and data analysis. For data collection, wearable inertia sensors were used. We used publicly available dataset with appropriate IRB approval process. The dataset consisted of data collected from variety of wearable inertia sensors. Based on the dataset, an intelligent activity recognition model was developed. And based on further analysis of the classification model, a frame work of a complete HAR system was designed and validated.

1.2 Significance

With increasing number of elderly people, the health status of these people has become a big concern around the world. Especially for those elderly people who have trouble in taking care of themselves or even reporting their health problem, a system that can monitor their activities in real time while not violating their privacy too much is needed.

If the system framework proposed in this research can be successfully implemented, it should be able to help elderly people avoid harmful events (i.e., fall down) from happening in a lot of cases, so that the chance of these elderly people getting injured in daily activities can be greatly reduced, and thus improve their life quality.

1.3 Objectives

1. Develop a computational model for human activity recognition.
2. Analyze the computation model to better understand the important features for falling activities. Based on this understanding, validate and further improve the performance of the model on fall detection.
3. Based on the result of the computation model, develop a theoretical design for a fall detection system.

CHAPTER 2. REVIEW OF LITERATURE

Elderly people's health status has been a bigger and bigger concern nowadays. Lots of undesirable activities like falling down can be big threats to their health. In 2014, falls are the first leading cause of death and injuries for elder people (over 65 years old) in United States (Bergen, 2016). To reduce the possibility of elderly people getting injured from undesirable harmful activities, effective HAR systems are needed. This literature review will provide an overview of the popular techniques that have been applied in HAR.

2.1 HAR Technologies overview

There are mainly three different sensing technologies for HAR right now, including RGB camera, depth sensors, and wearable sensors (Ann & Theng, 2014). RGB camera method is to develop a vision system to capture human movement and use feature extraction and supervised learning to recognize human activities; depth sensor method is to use infrared sensor or camera to measure the distance between the sensor and human body parts and use these distances to recognize human activities; Wearable sensor method uses sensors like accelerometer and gyroscopes which can be mounted on human bodies to collect data and then do HAR. In spite of the lack of accuracy compared to depth sensors and RGB camera, wearable sensors have been the most popular choice for HAR in recent years because of its flexibility, low cost and low power consumption (Ann & Theng, 2014).

2.2 Different designs of wearable inertia sensor system

Although it is widely agreed that wearable inertia sensor is most suitable choice for data collection, the number of sensors and the location where the sensors should be mounted are still in debate. As for the number of sensors, it is no doubt that more sensors can give higher accuracy. In the work done by Bao and Intille (2004), five bi-axial accelerometers were used for data collection to study 20 different activities, and the five sensors were mounted on left arm, right wrist, left thigh, and hip (waist). Similar configuration was used in quite a few later researches. In the work done by Ozdemir and Barshan (2014), six 9-axis inertia sensors were used for data collection to study 16 daily activities and 20 different fall actions. The 9-axis

sensor can measure 3-axis acceleration, 3-axis angular velocity and 3-axis magnetism, and the six sensors were mounted on head, chest, waist, right thigh, right wrist and right ankle. Yet it is not hard to imagine that it is very inconvenient for the user to wear sensors all over his/her body.

Although each individual sensor is not big in size, a sensor system with five or six of them is still not a user-friendly design. As a result, researches started to focus on using less sensors for data collection. In the work done by Reiss and Stricker (2012), three inertia sensors are used to study 18 different activities. The sensors are mounted on chest, wrist on dominant arm, and dominant ankle. In the work done by T. Zhang, Wang, Liu, and Hou (2006), the possibility of using just one sensor for data collection was explored. One tri-axial accelerometer was used in the experiment to recognize five categories of activities including two categories of fall activities. The sensor is mounted on the back of a cellphone, and the cellphone is put inside the pocked of clothes or hanged on the neck. In the work done by M. Zhang and Sawchuk (2012), one inertia sensor was used to study nine different activities. The sensor contains a tri-axis accelerometer and a tri-axis gyroscope, and the sensor is mounted on the waist of subjects. Because more and more cellphone models in recent years have embedded inertia sensors, and cellphone is the kind of device that most people would carry around all the time no matter what, researchers start to explore the possibility of using the sensor embedded in cellphones to collect data for HAR. In the work done by Lee and Choi (2012), built-in sensors of Google nexus phone were used to detect fall action while the volunteers were holding the phone in four different position. In the work done by Anguita, Ghio, Oneto, Llanas Parra, and Reyes Ortiz (2013), the accelerometer embedded in Samsung Galaxy S2 smart phone was used to study six different ADLs. Besides cellphones, smart devices like smart watches have also been tested for HAR task (Tian, Xu, Tao, & Wang, 2017).

Yet although using the embedded sensor in smart phones or other smart devices for data collection is very convenient for the users, Stisen et al. (2015) argued that the heterogeneity of the embedded sensors in the smart devices can cause big problems for the sensor system. The heterogeneities can be caused by sensor biases, sampling rate heterogeneity and sampling rate instability (which can be very serious when the CPU is multitasking). The effect of heterogeneities can significantly lower the recognition accuracy. Yet it seems that the problems caused by the heterogeneities of sensors can be resolved by up-sampling or down-sampling the data (Stisen et al., 2015).

2.3 Methodologies for data analysis

In the past researches, various techniques have been applied for analyzing the data collected from wearable inertia sensors. Most of the techniques can be categorized into supervised learning including traditional machine learning techniques, and artificial neural network. Yet because it is very difficult to collect large amount of data of human activity, some researches also tried utilizing unsupervised learning for HAR.

2.3.1 Unsupervised learning

For unsupervised learning, the acceleration data are normally divided into fixed-length time window, and different features can be extracted for each of window. Then cluster methods like k-mean cluster, or other matrix based cluster methods like Ward-Linkage can be used to cluster data samples into different activities based on the distances between the feature vectors of each time window (Mejia-Ricart, Helling, & Olmsted, 2017; Wang, Lu, Wang, Liu, & Zhou, 2017).

2.3.2 Supervised learning using traditional machine learning

For supervised learning, machine learning is a widely used technique in HAR. The work done by Bao and Intille (2004) is one of the earliest works that use machine learning for HAR. In this work, the data are collected at a sampling frequency of 76.25Hz, and the data were divided into fixed-length sampling window with each window consisting of 512 samples. That means each window has 6.7 seconds of data. Between each adjacent window, there are 256 samples overlapping with each other. Then features like mean acceleration, energy, and some frequency domain features were extracted for each sampling window. These feature values of each window were put into one vector which means each sampling window has one feature vectors. These vectors were divided into training set and testing set and were used as input to machine learning algorithms. The machine learning algorithms tested in this work included Decision Table, Instance-Based Learning, C4.5 decision tree and Naive Bayes. Among the four algorithm, C4.5 decision tree achieved the highest accuracy which is 84.26% (Bao & Intille, 2004). Although the accuracy of this work is not great, it sets up a frame work for lots of latter works.

Up to today, machine learning has been a dominant methodology for data analysis in HAR. Most of the work that use machine learning follow a similar logic as is presented by Bao

and Intille (2004): divide the acceleration data into fixed-length window, extract features from each window to form feature vectors, input the feature vectors into machine learning model for training and validations. The windows length is mostly larger than four seconds, and the machine learning models include the models that are mentioned by Bao and Intille (2004), which are Decision table, Instance-Based Learning (including k-nearest-neighbor and Radio-based-network), decision table and Naive Bayes, and also some other algorithms like Support Vector Machine (SVM). In fact SVM has been one of the most popular choice in HAR. The work done by Sun, Zhang, Li, Guo, and Li (2010) used SVM for classification to recognize human activity when the acceleration sensor is mounted in different orientation. The work done by Y. Chen, Guo, and Wang (2016) used an improved SVM algorithm called Fuzzy Least Square Support Vector Machine while using Ensemble Empirical Mode Decomposition for feature extraction and Sparse Multinomial Logistic Regression for feature selection. In the work done by Anguita et al. (2013), a more energy efficient, hardware-friendly SVM algorithm was developed so that it can be easily implemented on a cell phone. In the work done by De Leonardis et al. (2018). A comparison was made on the performance of real time HAR between the popular machine learning algorithms that are mentioned earlier including K-nearest neighbor, Decision Tree, SVM, and Naive Bayes. Also, a shallow Feed Forward Neural Network was tested. Data were collected from fifteen subjects on nine different daily activities, and 38 different features were extracted from 5-second sampling window as input to the machine learning or neural network algorithm that are mentioned above. It turns out that K-nearest Neighbor has the best performance and achieves 97% accuracy; Feed Forward Neural Network, Naive Bayes and SVM all get a similar accuracy which is around 96%; and Decision tree has the worst performance in this experiment, whose accuracy was 91%, which is still acceptable. Yet although the result of this comparison shows that K-Nearest-Neighbor has the best accuracy, the performances of the five algorithms are actually very close and the result can be different if the test was done on a different dataset.

There are some other algorithms that deploy very big improvements based on the framework proposed by Bao and Intille (2004). Like in the work done by T. Zhang et al. (2006). In this work, a data analysis model that can predict fall activities in real time was developed. In this work, if the total acceleration stays near gravity for more than one second, it is considered to be a sign of motionless activity, and 192 data points (1.5 seconds) before the point where the

acceleration gets near gravity will be further analysis using one class SVM to divide the data into two categories which are doubtful fall event and activity of daily, and then Kernel Fisher discriminant algorithm and was used to further separate the data which are categorized as doubtful fall event for more precise classification, and k-nearest-neighbor was used to control the fall alarm rate. In the work done by M. Zhang and Sawchuk (2012), an algorithm called Bag-of-Feature was introduced. The method is inspired by natural language processing, as acceleration data on time domain can be divided into small time window just like an English word can be divided into multiple letters. While a word is a combination of different letters, an activity is a combination of different type of time window. The data collected in this work were divided into fix-length time windows, whose sizes are under two seconds, which are much smaller than the time window proposed by Bao and Intille (2004). Then 10 different features were extracted for each window and K-mean Clustering and Gaussian Mixture Model were used to categorize the windows into different ”letters.” Then vectors were created using these letters and were used to do activity classification using SVM

2.3.3 Artificial Neural Network

In recent years, with the availability of big data and computing power become stronger and stronger, artificial neural network and deep learning has become more and more popular for classification problems such as disease diagnostic (Hosseini-Asl, Keynton, & El-Baz, 2016). As a result, researchers have started to explore the possibility of using such a tool for HAR.

2.3.3.1 Introduction to Artificial Neural Network and its terminologies

Artificial Neural Network (ANN) is a man created computing system that simulates the structure of a biological computing system like human brain. An ANN consists of an input layer, an output layer and multiple hidden layers in between. Each layer consists of multiple nodes which represent neurons in human brain. Each nodes is a number with a weight and sometimes a bias assigned to it, and the nodes on one layer are connected the nodes on adjacent layers by linear or nonlinear transformation. This linear or nonlinear transformation is called activation function.

Data in ANN can flow in two direction, forward and backward. When the data flow forward, the process is called feed forward phase. Data flow into input layer, through hidden

layer, and finally reaches output layer and gives an output. Feed forward phase happens when the ANN is actually implemented in application, being tested or validated, or during training when error needs to be calculated; when the data flow backward, the process is called backward propagation. It means when an error is calculated through a feed forward phase, the error will be fed back into the ANN and update all the weights and biases in the network using certain algorithms to minimize the error and optimize the performance of the ANN. There are multiple algorithms for this optimization process, and gradient based learning like Stochastic Gradient Descent, is the most popular and successful algorithm (Lecun, Buttou, Bengio, & Haffner, 1998) When training the neural network, in order to save time, input data are normally divided groups, and instead of training one data sample at a time, all the data in one group are trained together. Each group is called a mini-batch, and the number of data in each mini-batch are called batch size. The total number of mini-batch that are trained during the training process is called iteration, and the number of times that the whole dataset is trained during the training process is called epoch.

While ANN has been widely used in lots of different applications, people did not start using ANN for HAR until recent years, and not a lot of researches have been done on this topic yet. And among these few researches, Convolution Neural Network (CNN) is a pretty popular structure to classify human activities. Also, some of the researches chose to use Recurrent Neural Network (RNN) because of its capability to deal with data sequence.

2.3.3.2 Introduction to Convolution Neural Network

CNN has become one of the most popular classification algorithms in recent years. A CNN means that the network use convolution layer for feature extraction. For each convolution layer, the nodes are convolved with a certain a certain number of filters. The filter number is predefined based on the complexity of the problem. The filters normally have the same size, and consists of random weights initially, and these weights will be updated through learning process. The output of the convolution is an array or matrix that have the same size with the convolution layer, and will be fed into the next layer as input. CNN was initially developed to deal with "variability of 2D shapes" (LeCun et al., 1998, p. 1). It is often used in applications like image processing where building an accurate pattern recognition system is very challenging. One of the popular CNN structure developed in early year is called LeNet5 (LeCun et al., 1998), which was

originally designed to recognize handwritten character. In recent years, deeper CNN model like Alex-net has been developed to deal with more complicated image classification problems and larger dataset (Krizhevsky, Sutskever, & Hinton, 2012).

2.3.3.3 Related work on using CNN for HAR

Although HAR based on wearable inertia sensors has always been considered to be a 1D signal processing problem, Jiang and Yin (2015) proposed to transform the 1D acceleration signal into a 2D image-like signal so that CNN can be applied to do the feature extraction and classification. The signal collected in this study is a 6-axis inertia data including 3-axis acceleration and 3-axis angular velocity. The multiple channels of 1D signals were mapped into a single-channel 2D signal in a way that every signal channel has a chance to be adjacent with every other channels.

The 2D single-channel 2D signal was used as input like an image to a CNN. The structure of the CNN is similar to LeNet5 (LeCun et al., 1998), which includes two convolution layer groups, one fully-connected layer and one soft-max layer. Each convolution layer group includes a convolution layer and a subsampling layer. The algorithm was tested on three different datasets, and the result showed that if the CNN structure were used along with a 2D Discrete Fourier Transform, it could give a minimum accuracy of over 97%, which is over 1% higher than SVM while the computation cost is only one third of SVM (Jiang & Yin, 2015).

Similarly, in the work done by Fakhrulddin, Fei, and Li (2017), 2D CNN was used to process the data collected from two 3-axis accelerometer. The 1D time series data were mapped into 2D image-like data using the Markov transition fields (MTF) algorithm proposed by Z. Wang and Oates (2015). The 2D signal was then put into LeNet (LeCun et al., 1998) based CNN structure for classification.

CNN is normally considered as a powerful tool to handle 2D signal processing problem. But because convolution is an often-used operation in 1D digital signal processing for feature extraction, people also use 1D CNN for 1D signal processing.

In the work done by Li, Zhang, Zhang, and Wei (2017), CNN was used to analyze electrocardiogram signal and diagnose cardiovascular disease. An accuracy of 97.5% was achieved in this study, which proves the feasibility of using CNN for 1D signal processing.

The work done by Lee, Yoon, and Cho (2017) used 1D CNN based method for HAR. The data were collected from a single embedded accelerometer on a Google Nexus Phone at 1Hz sampling frequency, which means the quality of the data can be considered very poor. The structure of the network contains one convolution layer with 128 filters for feature extraction, one max-pooling layer for feature selection, and a dropout layer with a rate of 0.5 to accelerate training process while increasing the generalizing ability of the network. The accuracy of the model based on CNN was compared with the accuracy based on random forest, and the result of CNN (91.32% and 92.71%) outperformed the result of random forest (85.72% and 89.10%) (Lee et al., 2017). Considering the poor quality of the data collected in this study, the result of this study proves the feasibility of CNN in HAR application.

In the work done by Hammerla, Halloran, and Ploetz (2016), a comparison was made between three types of ANN on their performance on HAR. The three types of ANN includes deep-forward neural network, CNN, and recurrent neural network (RNN). In this work, these three ANN were tested on three different datasets, including PAMAP2 dataset (Reiss & Stricker, 2012a), Opportunity dataset (Roggen et al., 2010), and Daphnet Gait dataset (Bachlin et al., 2010). The overall result showed that CNN and RNN have very similar performance, with RNN performing better on Daphet Gait dataset and Opportunity dataset, and CNN performing better on PAMAP2 dataset. Both of these two ANN structure have better performance than Deep-forward Neural Network on all the three datasets. Also, the result indicates that for CNN, if the raw data are fed directly into the network without any preprocessing, shallow structure has better performance than deep structure. But the research group did not do any research on what effect would adding preprocessing to the algorithm bring to the result yet. This is one of the few researches that used RNN for HAR, and the result shows that RNN have a very big potential on dealing with HAR dataset.

2.3.3.4 Introduction to Recurrent Neural Network

Recurrent neural network (RNN) has become more and more popular recently as a Neural Network model to deal with application that involves sequential data, like natural language processing (Karpathy, Johnson, & Fei-Fei, 2015). Take Natural Language processing as an example, the RNN takes each words in a sentence as one input. For each input, the RNN will produce the an output, and also a hidden state which serves like a memory to remember the

data in the early stage of the data sequence; then the RNN will use the hidden state and the new input to produce the next output and a new hidden state, and this process will continue until the end of the data sequence. Because of the existence of the hidden state, RNN has a very strong capability to deal with data on time domain. As a result, RNN is one of the most popular algorithm for natural language processing nowadays (Karpathy et al., 2015). Also for this reason, RNN should be able to handle HAR dataset very well.

In addition to the work done by Hammerla et al. (2016), Steven Eyobu and Han (2018) also used RNN for HAR. In this work, data were collected from a 6-axis accelerometer. 100-feature vectors and 200-feature vectors were generated from 3 second slicing windows. These feature vectors were used as input to RNN.

2.4 Conclusion for methodology

In the field of HAR, wearable inertia sensors are now the most popular choice for data collection as compared to other sensors like RGB cameras or depth sensors (infrared sensors) because of its flexibility (Ann & Theng, 2014). Yet given that wearable inertia sensor is the best choice of sensor, the number of sensors that should be used and where the sensor should be mounted is still in debate.

As wearable inertia sensor is used for data collection, multichannel 1D data on time domain will be used as input to categorize human activities. Most researches in the past use traditional machine learning technique for data analysis, and follow the framework that was proposed by Bao and Intille (2004). While SVM seems to be the most popular choice, other machine learning techniques like decision tree, K-nearest-neighbor, Naive Bayes, random forest have also been tested, and they all have pretty similar performance (Hammerla et al., 2016).

In recent years, with artificial neural network becoming more and more popular, researchers start to explore the possibility of using ANN for HAR. Quite a lot of works have been done on using feed-forward network, and it has similar performance with traditional machine learning techniques (De Leonardis et al., 2018). Recently, some researchers start to explore the possibility of using CNN and RNN for HAR, and they seem to better performance than Feed-Forward Network and other traditional machine learning techniques (Jiang & Yin, 2015; Lee et al., 2017).

2.5 Description of a publicly available dataset

As in this thesis, data will not be collected from volunteers, publicly available datasets was used for data analysis. The data that used are collected by Ozdemir and Barshan (2014). These data were collected at Erciyes University Clinical Research and Technology Center and is publicly available on UCI Machine Learning Repository (Dua & Graff, 2019). The data were collected from 10 male volunteers and seven female volunteers. Six MTx sensor units were placed on each volunteers, with one on the neck, one on the chest, one on the waist, one on the wrist of dominant arm, one the dominant thigh and one on the dominant ankle. Each MTx sensor unit is able to collect 3-axis acceleration data, 3-axis angular velocity data, 3-axis magnetometer and atmosphere pressure, and the sampling frequency of the sensor is 25Hz. The test collected data on 20 different fall activities and 16 activities of daily living. Below are description of the 36 activities. These description are copied directly from the dataset information obtained from Ozdemir and Barshan.

The 20 fall activities included:

"*1. Front-lying: from vertical falling forward to the floor.*

2. Front-protecting-lying: from vertical falling forward to the floor with arm protection

3. Front-knees: from vertical falling down on the knees

4. Front-knees-lying: from vertical falling down on the knees and then lying on the floor.

5. Front-quick-recovery: from vertical falling on the floor and quick recovery

6. Front-slow-recovery: from vertical falling on the floor and slow recovery.

7. Front-right: from vertical falling down on the floor, ending in right lateral position.

8. Front-left: from vertical falling down on the floor, ending in left lateral position.

9. Back-sitting: from vertical falling on the floor, ending sitting

10. Back-lying: from vertical falling on the floor, ending lying

11. Back-right: from vertical falling on the floor, ending lying in right lateral position.

12. Back-left: from vertical falling on the floor, ending in lying in left lateral position.

13. Right-sideway: from vertical falling on the floor, ending lying.

14. Right-recovery: from vertical falling on the floor with subsequent recovery.

15. Left-sideway: from vertical falling on the floor, ending lying.

16. Left-recovery: from vertical falling on the floor with subsequent recovery.

17. Rolling-out-bed: from lying, rolling out of bed and going on the floor

18. Podium: from vertical standing on a podium going on the floor

19. Syncope: from standing falling on the floor following a vertical trajectory

20. Syncope-wall: from standing falling down slowly slipping on a wall

The sixteen Activities of Daily Lives include:

1. Walking-fw: walking forward.

2. Walking-bw: walking backward.

3. Jogging: running.

4. Squatting-down: squatting, then standing up.

5. Bending: bending about 90 degrees.

6. Bending-pick-up: bending to pick up an object on the floor.

7. Limp: walking with a limp.

8. Stumble: stumbling with recovery.

9. Trip-over: bending while walking and then continue walking.

10. Coughing-sneezing: coughing or sneezing

11. Sit-chair: from vertical, to sitting with a certain acceleration on to a chair (hard surface)

12. Sit-sofa: from vertical, to sitting with a certain acceleration on to a sofa (soft surface)

13. Sit-air: from vertical, to sitting in the air exploiting the muscles of legs

14. Sit-bed: from vertical, to sitting with a certain acceleration on to a bed (soft surface)

15. Lying-bed: from vertical lying on the bed

16. Rising-bed: from lying to sitting"

(Ã–zdemir, A.T.; Barshan, B. â€œDetecting Falls with Wearable Sensors Using Machine Learning Techniques.â€• , Sensors 2014, 14, 10691-10708.)

For each volunteer, each activity was repeated for five times. For each trial, there are data on time domain with 25 data points in every seconds. The structure of the data can be seen in Figure 2.1:

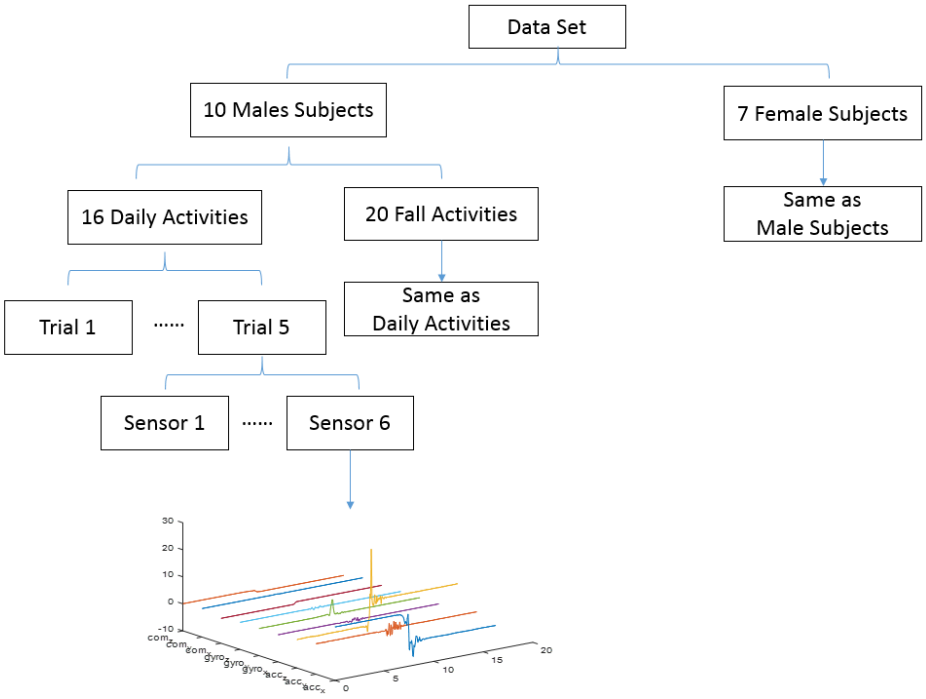


Figure 2.1. Structure of Simulated Falls and Daily Living Activities dataset (Ozdemir & Barshan, 2014)

Based on this dataset, several studies have been conducted on doing fall detection using different machine learning algorithms including k-nearest neighbor, Bayesian decision making, Support Vector Machine, Least Square method, Dynamic Time Warping, ANN (Ozdemir, 2016; Ozdemir & Barshan, 2014), K-Nearest Neighbor, Random Forest and Random Committee (Ntanasis, Pippa, Ozdemir, Barshan, & Megalooikonomou, 2016). The input is the 9-axis inertia data from one or multiple different sensors. For each trial of data, a peak was determined based on total acceleration. Then 2 seconds of data before and after the peak point, which is 4 seconds of data in total, was used for further analysis. Seven different types of features including minimum, maximum, variance, skewness, kurtosis, autocorrelation sequence and Discrete Fourier Transform peaks were extracted from the 4 seconds of data, and these features were put into one 1404 ×1 feature vector. Principal component analysis (PCA) was used to reduce the size of the feature vector to 30 × 1, and the new feature vector was input into the machine learning model for classification. The output of the model is binary class pair, being fall-activity or non-fall activity. The performance of the model was measured by accuracy, sensitivity and specificity. The result shows that:

When only one sensor is used, data from thigh sensor give the highest accuracy, being 99.48% when SVM classifier is used; data from waist sensor have a similar performance, which can give an accuracy of 99.28% when Random Forest is used Committee (Ntanasis et al., 2016). When multiple sensors are used, the accuracy would increase when the number of sensors are increased. The increase rate is relatively low, especially for those sensors that already have a good performance on their own (Ntanasis et al., 2016). Among all the different machine learning classifier, K-nearest neighbor has the best performance, with an overall accuracy of 99.21% when only one sensor is used, and ANN has the worst performance, with an accuracy of 94.92% (Ozdemir, 2016). Different from the works that have been done on this dataset, this thesis will use different algorithms for feature extraction and classification, including CNN and RNN. Also this thesis will try to develop a model that has the capability of doing real time fall prediction.

CHAPTER 3. RESEARCH METHODOLOGY

3.1 Data description

To build a computation model for human activity recognition, inertia data of different human activities are needed. Considering the limitation on time and resource, the data were not collected from human subjects. Instead, a public dataset was used during this process. (Ozdemir & Barshan, 2014)

The dataset that was mainly used to build the computation model is the Simulated Falls and Daily Activities dataset (Ozdemir & Barshan, 2014), which is introduced in Chapter 2.2.1.

The data in this dataset (Ozdemir & Barshan, 2014) were collected from 17 human subjects. The 17 human subjects include 10 males and seven females. They were all randomly selected healthy subjects. Each volunteer was asked to perform 20 fall activities and 16 activities, and each activity was performed for five to six trials. The data were collected from MTw sensor unit produced by Xsens (2009). Six MTw sensor units were mounted on six different body locations for each volunteers for data collection. The six body locations include: head, chest, wrist, waist, thigh, and ankle. Data collected for each location were recorded in a data file. (Ozdemir & Barshan, 2014). Figure 3.1 is an illustration of the structure of the dataset.

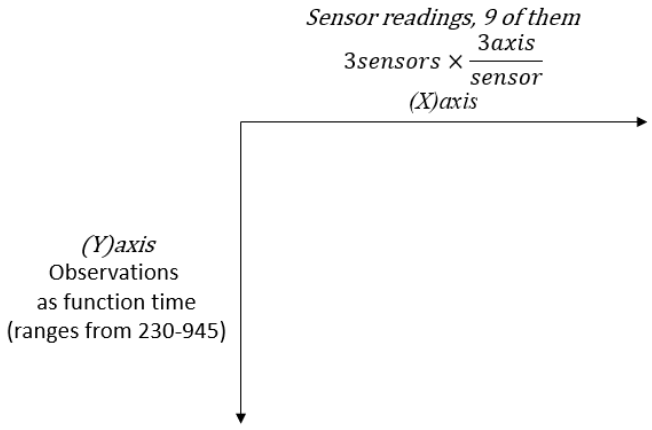


Figure 3.1. Structure of “Falls and Daily Living Activities” dataset

The experimental design of the data as reported by prior research (Ozdemir & Barshan, 2014), is as follow:

$$\frac{sensor\ readings}{sensor} \times \frac{3axis}{sensor} \times \frac{3sensors}{location} \times \frac{6location}{trial} \times \frac{5-6trials}{activity} \times \frac{36activities}{volunteer} \times 17volunteer$$

Sensor readings were collected at 25 Hz, which means the sensor unit collected 25 data samples every second (one data sample is collected every 0.04 second). Figure 3.2 is an example of the "Simulated Falls and Daily Activities" dataset. This example is the data collected from one location of one trial of one volunteer performing one activity. This example data file contains 616 data samples, but different data files may contain different number of data samples. The data are 2D data. One dimension is the nine readings, and the second dimension is time (data samples). This 2D data in each data file are collected from a single location. When data from multiple locations are used, the data become 3D data, with different locations becoming the third dimension.

$3sensors \times \frac{3axis}{sensor} = 9\,readings$

$Time(observation)$

	sensors1			sensors2			sensors3		
	ACC_X	ACC_Y	ACC_Z	GYR_X	GYR_Y	GYR_Z	MAG_X	MAG_Y	MAG_Z
1	9.6024	0.9186	-1.8341	0.0046	-0.0133	0.0036	-0.5876	-0.7644	0.6880
2	9.6168	0.8329	-1.7921	0.0055	-0.0114	0.0103	-0.5874	-0.7666	0.6912
3	9.6008	0.8713	-1.8173	-0.0013	-0.0089	0.0044	-0.5874	-0.7668	0.6855
4	9.6064	0.8636	-1.8177	0.0076	-0.0108	0.0053	-0.5874	-0.7666	0.6882
5	9.5978	0.9155	-1.7807	0.0082	-0.0141	0.0210	-0.5847	-0.7668	0.6880
6	9.6036	0.8397	-1.7651	0.0105	-0.0113	0.0225	-0.5874	-0.7666	0.6912
7	9.5856	0.8667	-1.7853	0.0196	-0.0124	0.0189	-0.5847	-0.7644	0.6851
8	9.6097	0.8595	-1.8426	0.0226	-0.0127	0.0167	-0.5903	-0.7644	0.6909
9	9.5963	0.8850	-1.8204	0.0109	-0.0101	0.0227	-0.5876	-0.7644	0.6880
⋮									
612	9.591889	0.183249	-2.16146	-0.00024	0.00248	0.010061	-0.67065	-0.48853	0.955322
613	9.573364	0.17395	-2.14996	0.00267	0.000811	0.013542	-0.67334	-0.48853	0.958252
614	9.579611	0.142574	-2.17564	0.001717	0.001907	0.012732	-0.67578	-0.48853	0.952881
615	9.570313	0.167847	-2.16064	0.006485	4.77E-05	0.010872	-0.67065	-0.48853	0.958252
616	9.582567	0.156999	-2.16527	-0.00072	0.001907	0.004911	-0.6731	-0.48853	0.952637

Figure 3.2. Example data from the "Simulated Falls and Daily Activities" dataset: one volunteer, one activity, one trial, and one location.

The dataset provides the possibility for comparing different sensors at different locations. A total of 36 activities (20 fall activities and 16 daily activities) were collected in this dataset, which means that the model developed from this dataset has a better capability to deal with the complicated situation in a real-world scenario.

3.2 Initial inspection on dataset

Figures 3.3-3.8 show some of the examples of the data plotted on time domain. In Figure 3.3 for example, acceleration data on x, y and z axis on right-sideway and left-sideway activities are plotted. (Detail description of activities can be found in Chapter 2.5) It can be seen that data of these two activities have similar patterns on x axis and z axis; but on y axis, the direction of peak (located in 10sec-15sec) is opposite, which means the two activities can be distinguished using the data on y axis. Figure 3.8 is a comparison of another pair of activities. In this figure, x,

y, z acceleration of lying-bed and rising-bed activities are plotted. (Detail description of activities can be found in chapter 2.5) In this activity pair, acceleration on y axis have similar patterns, but acceleration on x and z axis have different starting acceleration and ending acceleration.

Figures 3.3-3.5 are plots of two fall activities; Figures 3.6-3.8 are plots of two non-fall activities. Comparing the plots of fall activities (in Figures 3.3-3.5) and non-fall activities (in Figures 3.6-3.8), it can be seen that both plots of the two falling activities have one noticeable peak, while the plots for the two non-fall activities do not have such characteristic. This means that with these given example activities, fall activities and non-fall activities can be distinguished by peak.

Between different activities, very noticeable differences are observed. They can be used to distinguish different activities. Plots in Figures 3.3 and 3.4 and Figure 3.6 and 3.7 show the same activity performed by different human subjects. Plots from Figures 3.3 through 3.8 show the same activity performed by the same human subjects on different trials. Take the plots from Figures 3.6 and 3.7 for example. These two figures show the comparison of lying-bed activity and rising-bed activity conducted by two different human subjects. The accelerations on accel-x and accel-z have very similar patterns, but on accel-y, there is observable difference between the starting acceleration for rising-bed activity, and between the ending acceleration for lying-bed activity.

From these comparisons, it can be seen that between different activities, data have different patterns. But sometimes the patterns are significant, sometimes they are not. Between the same activities, the data usually have similar patterns, but there can still be exceptions like the example shown in the previous paragraph.

There are a total of 36 different activities in the dataset, and much more unique characteristics are needed to distinguish them from each other. Some of those characteristics can be visualized just like the examples mentioned above, while some of characteristics can only be extracted from certain mathematical operations; some of the characteristics can be found in the x, y, z acceleration data, while some of them may lie in data from gyroscope and compass. To deal with the complexity of the problem, I proposed using machine learning was chosen to do feature extraction, selection and classification. Two types of models including CNN and RNN were tested and compared in this thesis.

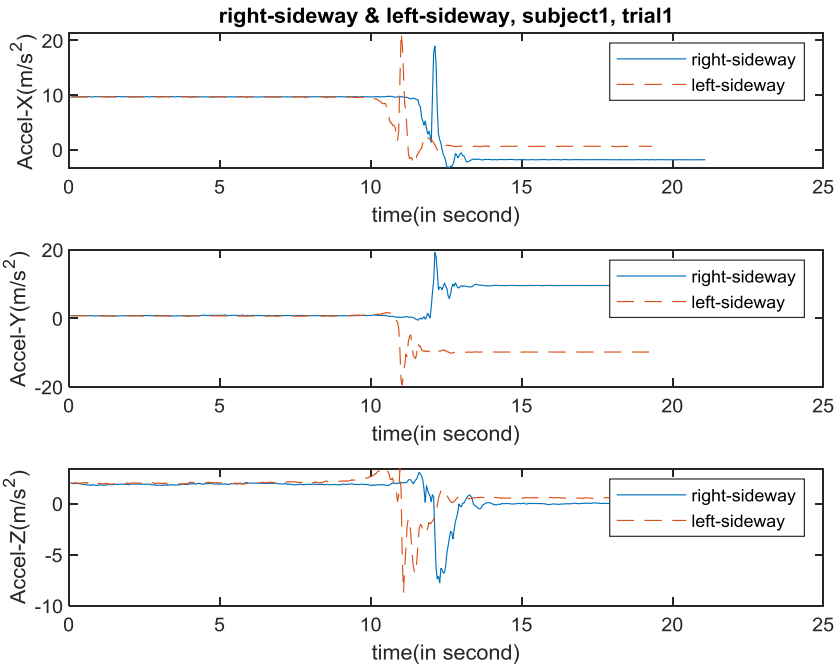


Figure 3.3. Data sample of falling right-sideway and falling left-sideway (fall activities), subject1, trial1

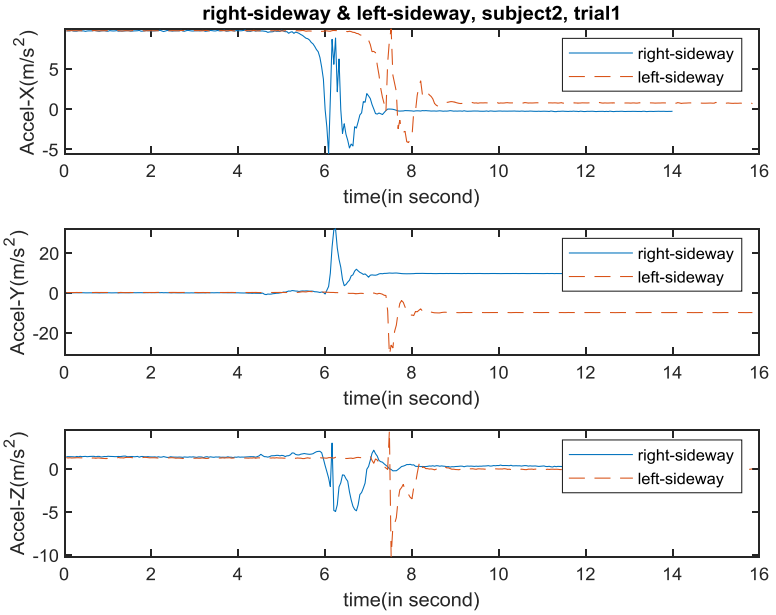


Figure 3.4. Data sample of falling right-sideway and falling left-sideway (fall activities), subject2, trial1

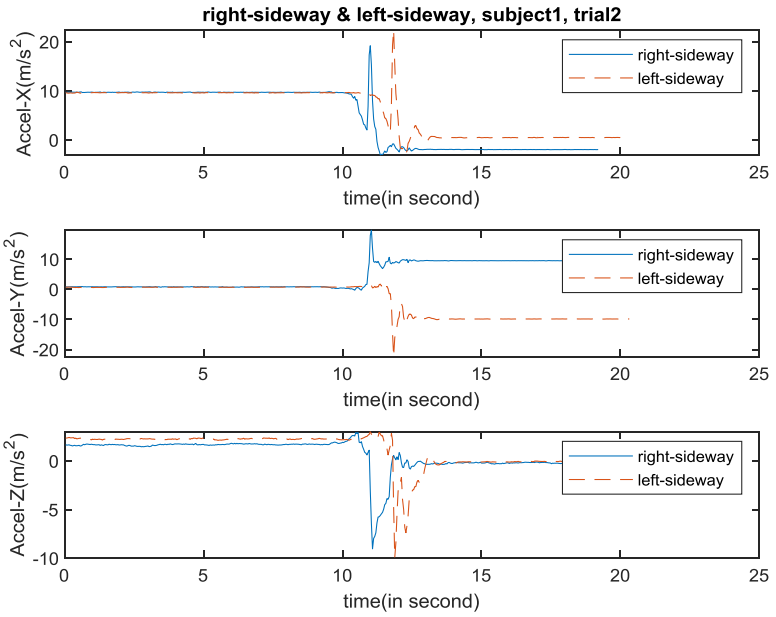


Figure 3.5. Data sample of falling right-sideway and falling left-sideway (fall activities), subject1, trial2

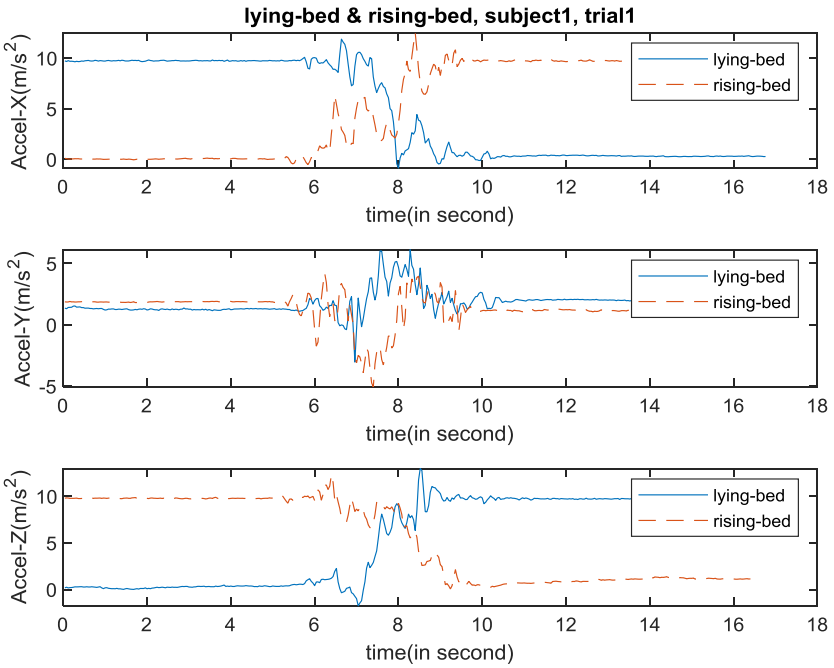


Figure 3.6. Data sample of lying-bed and rising-bed (non-fall activities), subject1, trial1

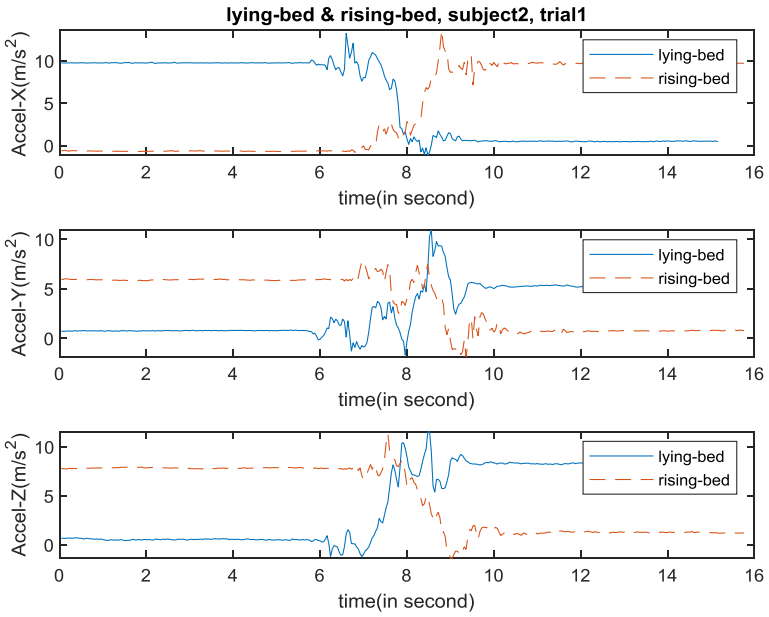


Figure 3.7. Data sample of lying-bed and rising-bed (non-fall activities), subject2, trial1

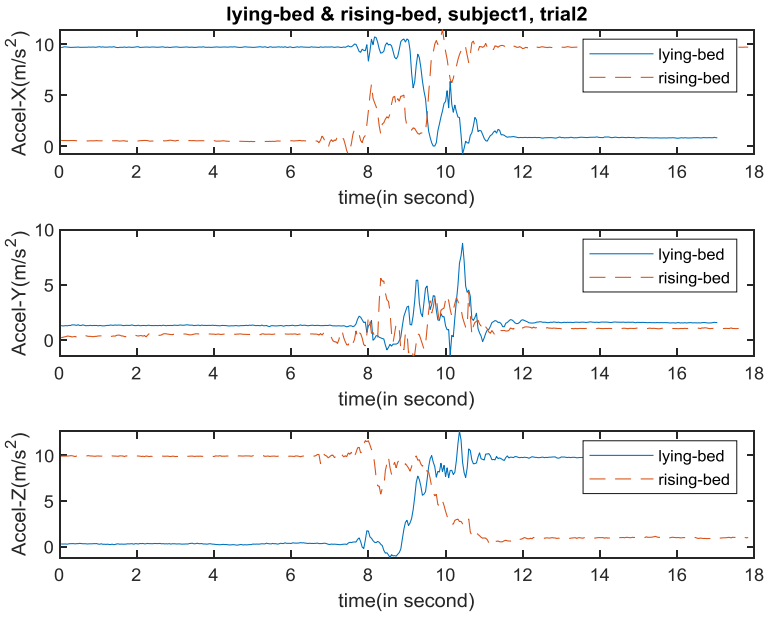


Figure 3.8. Data sample of lying-bed and rising-bed (non-fall activities), subject1, trial2

3.3 Prediction model development

3.3.1 Data preprocessing

Looking at the data plotted in Figures 3.3-3.8, it can be observed that there is a long flat data sequence at the beginning and at the end of every trial of data. This similarity is shared among the rest of the dataset.

The dataset (Ozdemir & Barshan, 2014) had two types of data files. One group represents fall activities, and the other group represents non-fall activities. Each data file involve other activities (i.e. standing still, sitting, lying down) before fall or non-fall activities, we call it precondition. Also, in each data file, the fall or non-fall activities were proceeded by other activities (i.e. standing still, sitting, lying down). We call these activities postcondition.

In this study, one of the objective was to develop a model to clarify if the data belongs to fall or non-fall category. To achieve that goal, it was considered necessary to separate a given data file into precondition, activity and postcondition regions.

The basic logic is to use variance and first order derivative to mark the actual activity. The reason is that ideally, when the human subject is standing still, lying or sitting, the variance should be close to zero; when the human subject is doing activities, the variance should be a considerably large number. This difference in variance can be used to differentiate actual activity and pre/postcondition. Yet in reality, considering the environment noise, the variance of the flat sequences can become quite significant. The algorithm shown below is developed to remove the flat sequences (This algorithm is applied to the data in each data file individually, and the algorithm is developed based on waist sensor, and acceleration data):

1. Calculate the total acceleration for every data points. Total acceleration is defined as:

$$Acc_T = \sqrt{{Acc_X}^2 + {Acc_Y}^2 + {Acc_Z}^2} \qquad (1)$$

Acc_T is total acceleration, Acc_X is x-axis acceleration, Acc_T is y-axis acceleration, and Acc_Z is z-axis acceleration. Figure 3.9 is an example of total acceleration.

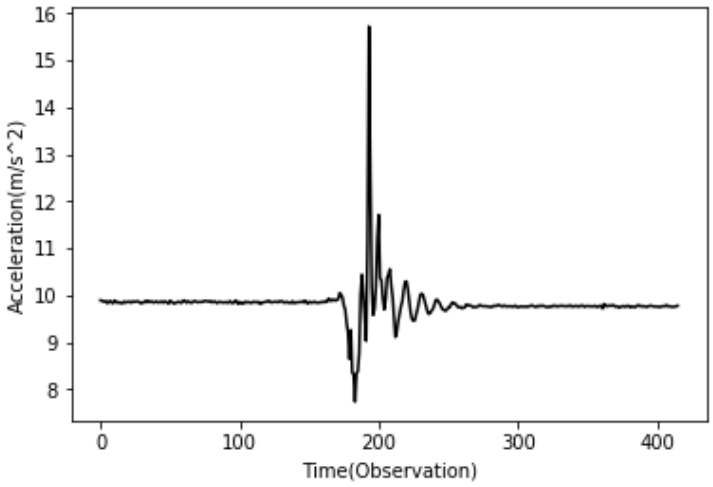


Figure 3.9. Sample acceleration data. x-aixs is time(number of data sample), and y-axis is acceleration(m/s2)

2. The given dataset was divided into multiple windows such that each window contains 10 observations. Based on the number observation in each data file, the number of windows may vary. Figure 3.10 shows an example data file with 400 observations and it is divided into 40 windows.

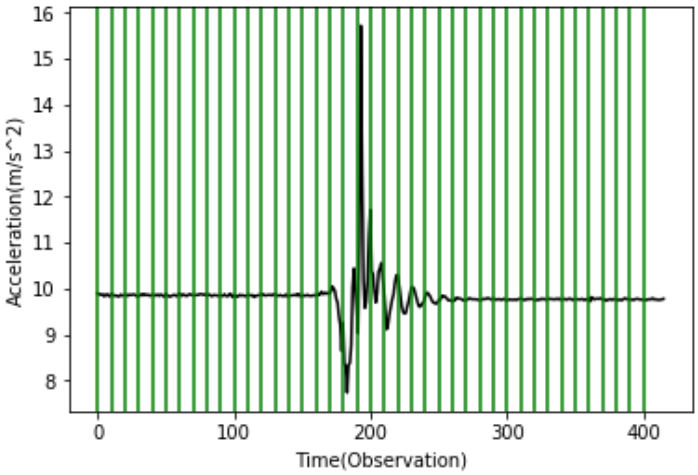


Figure 3.10.Sample data divided in to winodws of 10

3. Calculate the variance of acceleration for each windows to create a variance vector. Run the variance vector through a (1, 3) average filter, which means that the output for x_i is the average of x_i, x_{i-1} and x_{i+1}. The average filter will not be applied to the

first data point (x_1) or the last data point (x_n) due to the edge effect of average filter. x_1 is replaced by the average of x_1 and x_2, and similarly, x_n is replaced by the average of x_n and x_{n-1}. Figure 3.11 is a plot of the variance sequence.

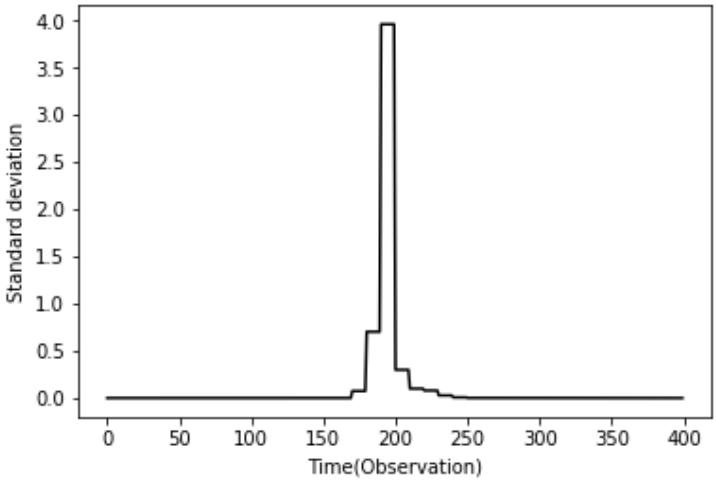


Figure 3.11. Variance sequence

4. Find out the window with the maximum variance. This window will be refereed as "activity window" in this work. In Figure 3.12, the window marked with red line is the activity window.

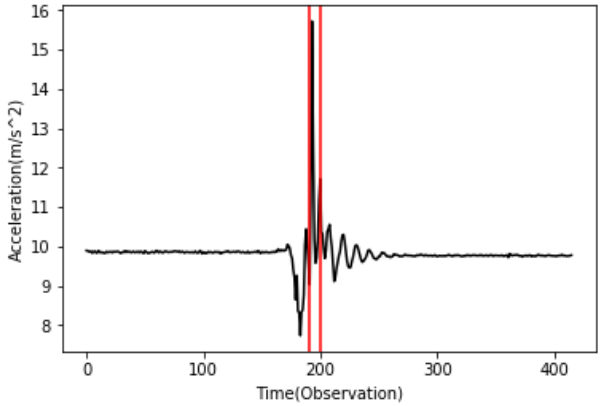


Figure 3.12. Activity window

5. Find out the windows with the minimum variance before and after the activity window. The standard deviation of this two windows are considered to be the system noise before and after the actual activity. (See step4 in appendix B4) The window

before the activity window will be refereed as “min_start window” (given by variable min_index_start in the program, appendix B4), and the window after the activity window will be refereed as “min_end window” (given by variable min_index_end in the program, appendix B4). The mean of the two window will be used as the base acceleration in later steps. In Figure 3.13, the window marked with yellow lines is the min_start window (Same as min_index_start); the window marked with orange line is the min_end window (Same as min_index_end). Then, calculate the average value of min_start window (given by variable mean_start in the program, appendix B4), and calculate the average value for min_end window (given by variable mean_end in the program, appendix B4).

6.

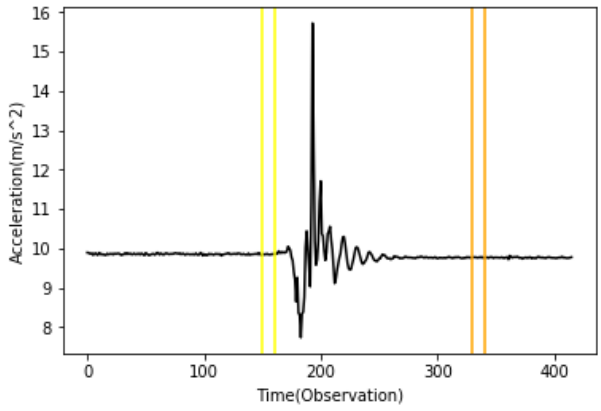


Figure 3.13 min_start window and min_end window

7. System noise is the standard deviation of the “min_start” window and “min_end” window (the yellow window, and orange window). In some cases (in some data files), the system noise is too low that even very small movement can be considered as actual activity. If the system noises that are lower than 0.01, they are set to 0.01. 0.01 is an empirical value which is drawn from experiments and proved to be the most effective for this work.
8. Scan from the “min_start window” to activity window (left to center), and from “min_end window” to activity center (right to center). For each new point, take this point and the next four points, then calculate the differences between these 5 points and mean_start/mean_end. A sequence of five difference values can be obtained

(given by variable diff_list_start/diff_list_end in the program, appendix B4). ***Diff1*** = minimum of the sequence. ***Diff2*** = mean of the sequence. (Diff1 refers to both Diff1_start and Diff1_end in appendix B4, and Diff2 refers to both Diff2_start and Diff2_end in appendix B4)

9. ***Diff1*** and ***Diff2*** are used to be compared with the scaled system noise. The scaled system noise is defined using the following equations:

$$\textbf{\textit{Diff1}}: sys\_noise/(4 - 3 * dist/len) \quad (2)$$

$$\textbf{\textit{Diff2}}: sys\_noise/(10 - 9 * dist/len) \quad (3)$$

In the two equations, $dist$ refers to the distance between the current data point and the "min_start window"/"min_end window," and len refers to the distance between min_start window/min_end windows to activity window. These two equations are heuristic equations. They are proved to be able to efficiently differentiate pre/postcondition data and actual data throughout preliminary exam.

10. When both ***Diff1*** and ***Diff2*** are smaller than the scaled system noise, the data point is considered to be part of the pre/postcondition. If either ***Diff1*** or ***Diff2*** is larger than the scaled system noise, the data point is considered to be part of the activity. Based on the result from experiments, the following two scaled system noise are proved to be most effective for this work.

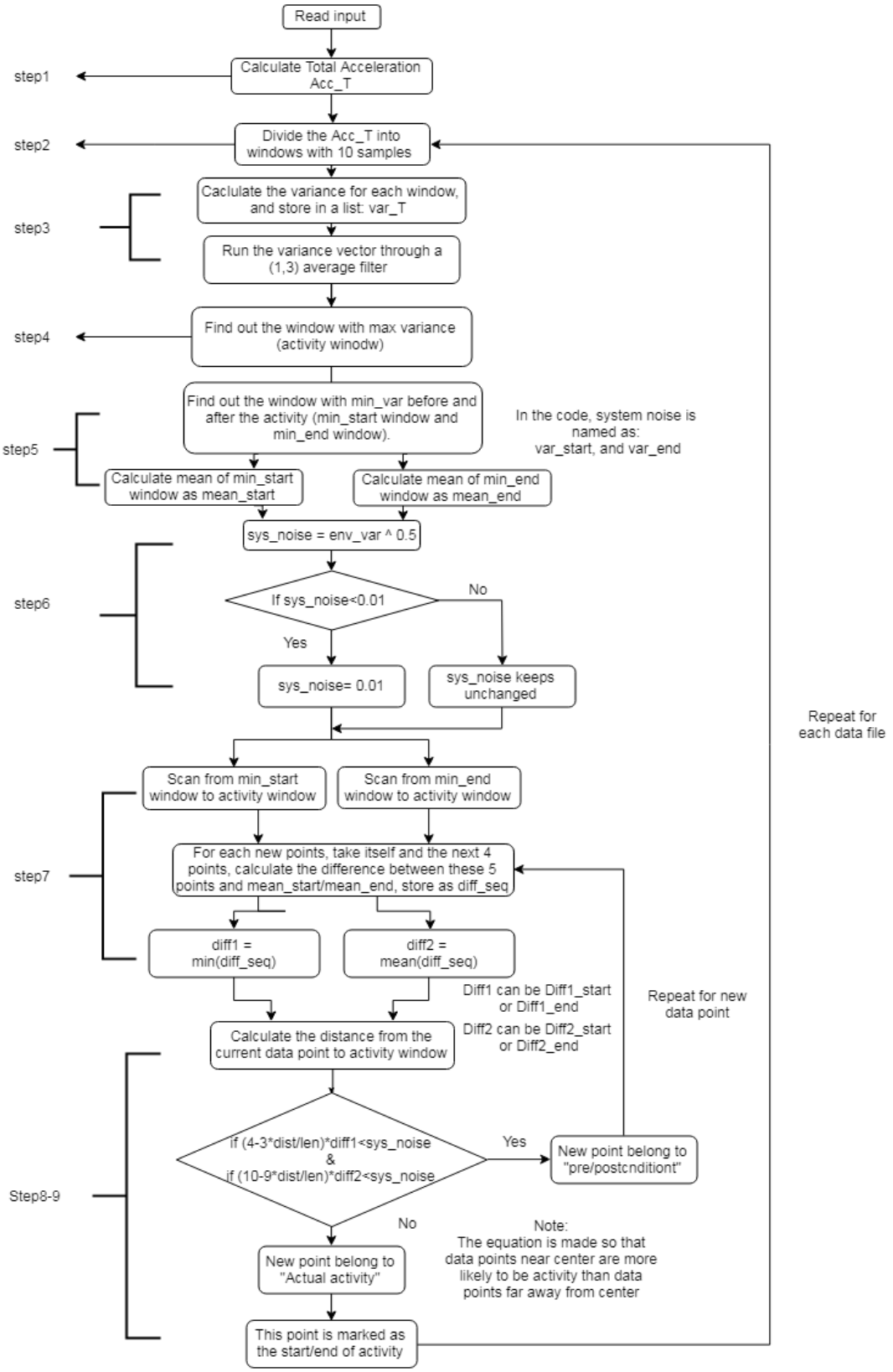


Figure 3.14. Flow chart of preprocessing algorithm

The source code of preprocessing is provided in Appendix B4.

The output of the preprocessing algorithm are two indices per data file. The indices mark the starting observation and ending observation of the actual activity.

Before applying the algorithm to the dataset, five activities were removed from the dataset because: for these five activities, the variance of the system noise and the variance for the actual activity are so close to each other that the algorithm mentioned above could not effectively distinguish one from another. The five removed activities include: (a) rolling out of bed (fall activity), (b) from standing to sitting in the air (daily activity), (c) coughing (daily activity), (d) squatting down (daily activity) and (e) bending over 90 degrees (daily activity).

After removing the five activities, 31 activities (12 daily activities and 19 fall activities) with 2842 data files are left in the dataset. When applying the algorithm to these 2842 data files, all but nine data files were relabeled properly. These nine data files were manually relabeled. (Program for manually relabeling can be found in appendix B5)

3.3.2 Convolution Neural Network

For this study, a 2D CNN was used. Convolution is the most commonly used mathematical operation to extract features from digital signals like the inertia data (Nielsen, 2015). For this study, nine different features (Ozdemir & Barshan, 2014), including 3-axis acceleration, 3-axis angular velocity and 3-axis magnetism were used. As mentioned in Chapter 3.1, the data collected by each sensor are 2D data, which is why 2D CNN is used in this study.

3.3.2.1 How 2D convolution work

2D convolution can be defined using the equation below (Proakis, 2001):

$$\sum_{\tau_u=-\infty}^{\tau_u=+\infty} \sum_{\tau_v=-\infty}^{\tau_v=+\infty} f(\tau_u, \tau_v) h(x - \tau_u, y - \tau_v) \qquad (4)$$

For this equation, *h()* is the data that need to be processed, and *f()* is filter/kernel that is used to process the data. The output is a 2D data array.

The convolution operation used in convolution neural network is not exactly the same as defined in the equation above, but it is very similar. Figures 3.15-3.17 show how convolution

layer works for single channel input. In these figures, filter of size (3, 3) is used; the data to be processed have a size of (10, 10).

The filter slides through the data from left to right, top to bottom. In Figure 3.15, the filter is on its first step. The output value can be calculated using the following equation:

$$output = (\sum_{n=1}^{9} f_n \times h_n) + b \qquad (5)$$

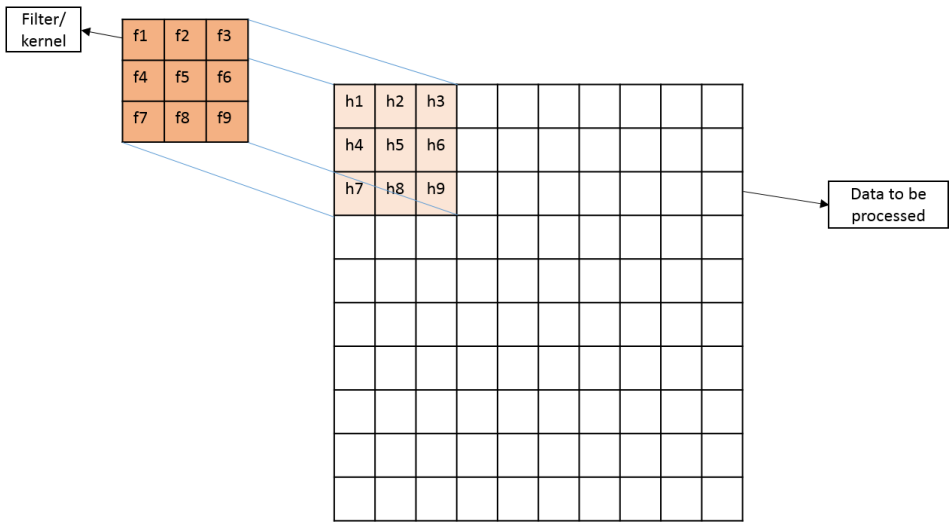


Figure 3.15. Convolution layer 1

In this equation, f_n are parameters in the filter, h_n are the input data, and b is the bias term. These parameters ($f_n$and bias) are initially randomly generated, and will be optimized through the training process.

The output is stored in at the center pixel. Then the filter moves one step to the right, as is shown in Figure 3.16.

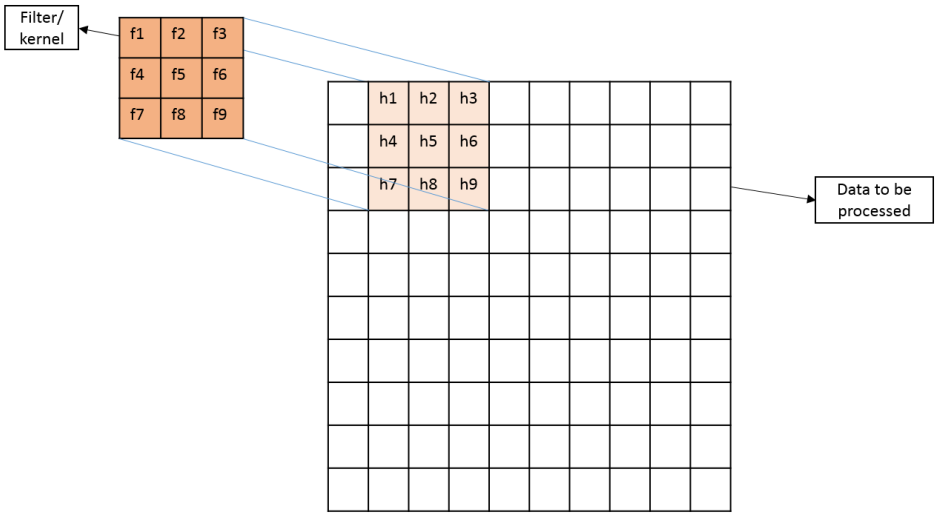


Figure 3.16. Convolution layer 2

The output is calculated using equation (5), and the output is stored in the center pixel (h_5). The filter start moving from left to right, top to bottom, until it reaches the right bottom corner of the data, as shown in Figure 3.17. The number of steps that the filter moves every time is called stride. In the example shown in Figures 3.15-3.17, the filter moves one step every time, which means stride is equal to 1.

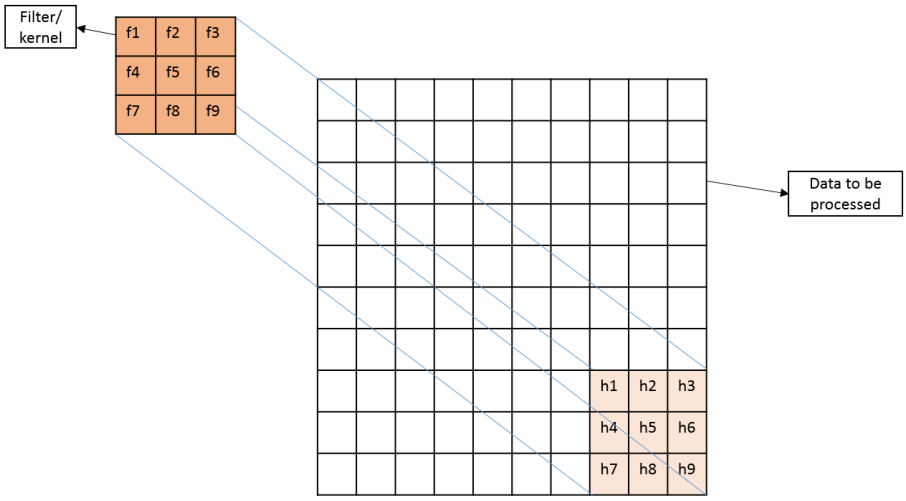


Figure 3.17. Convolution layer 3

At each step, an output is calculated using equation (2). These output forms a new feature map of size (9, 9).

Normal convolution procedure will reduce the size of the data by 1 when stride is equal to 1. In the example shown in Figure 3.17, the size of the input data is (10, 10), and the size of the output is (9, 9). In most of the CNN structure, the output size and the input size for a convolution layer should be the same. To achieve that, padding mechanism is used. As shown in Figure 3.18, one column of zeros and one row of zeros are added to surround the original data, so that the input size is increased from (10, 10) to (11, 11), and the size of the output becomes (10, 10), which is the same as the size of the original input data.

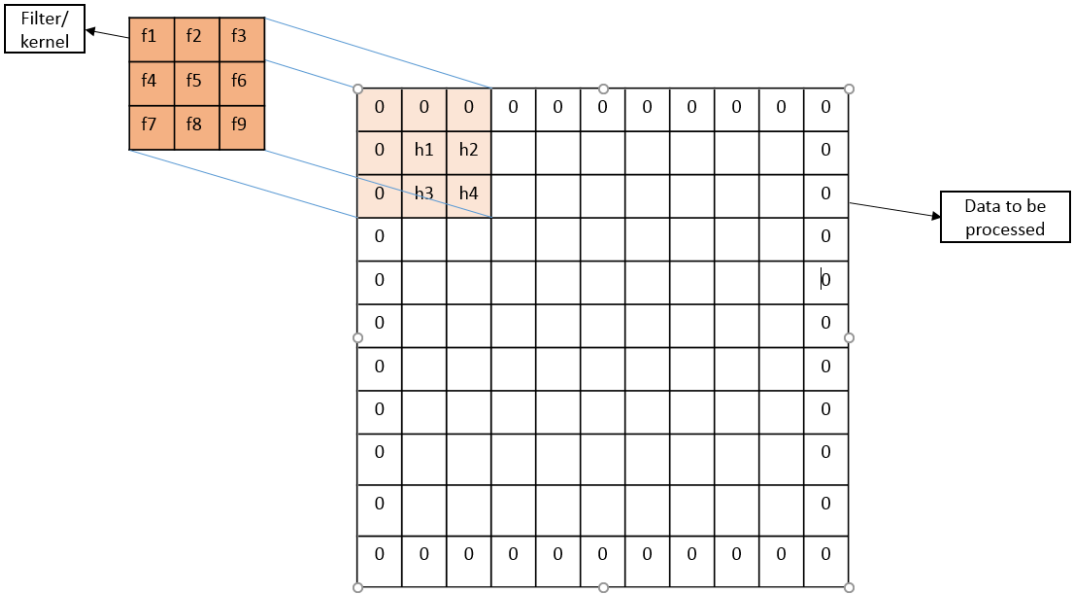


Figure 3.18. Convolution layer, with zero-padding

Take (3, 3) filter for example. There are nine parameters in the filter. These parameters, as well as the bias term, are consistent when the filter slides from the beginning to the end of the input data. For each filter of size (3, 3), there are 9 + 1 = 10 parameters that need to be optimized through the training process.

Normally, one filter is not sufficient to extract enough features from the input data. So multiple filters are used in a single convolution layer. Each filter has independent trainable parameters and bias. When each filter slides through the input data, a 2D feature map is produced as output. The number of feature maps is the same as the number of filters, and this number will be refereed as channel in this work. For example, when eight (3, 3) filters are used in a single convolution layer with zero-padding, and the input data is (10, 10), there will be eight channels

of 2D feature maps with a size of (10, 10) produced as output. Because each convolution layer has more filters, the layer has the capability to extract more features from the input data, but also the layer requires more computational power.

3.3.2.2 Data preparation for 2D CNN

As was mentioned at the end of Chapter 3.3.1, the output of the preprocessing algorithm are two indices that mark the start and end of the actual activity. Figure 19 is an example 3-axis acceleration data from the Simulated Falls and Daily Activities dataset.

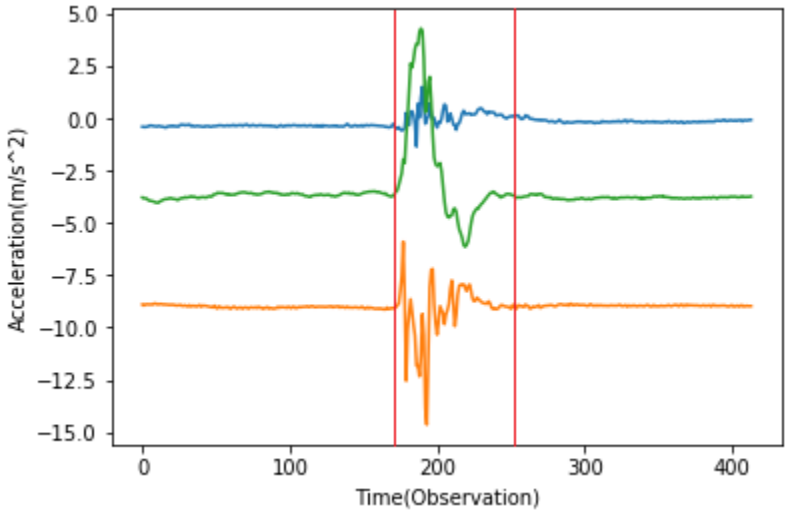


Figure 3.19. Sample data from the Simulated Falls and Daily Activities dataset.

The output for this example data from the preprocessing algorithm is [170, 250]. That means in this data file, observation 1-169 and observation 251-414(end) belong to pre-\condition and postcondition; observation 170-250 belong to actual activity. The actual activity observations are divided into fix-length windows. Two different window size are tested in this work, including 10 and 20. Take the data in Figure 3.19 for example, 80 samples belong to actual activity. When window size is 10, these 80 observations are divided into eight windows, with each window containing 10 observations; when window size is 20, these 80 observations are divided into four windows, with each window containing 20 observations. These fix-length windows will be used as input to CNN.

The CNN model was built using Tensorflow (version 1.10.0) (Abadi et al., 2015) with Keras back-end (Chollet, 2015). The code was written in python language (version 3.6.5) using

Anaconda platform (Version 5.2.0) (Anaconda Software Distribution, 2016), Spyder IDE (version 3.2.8). The program was run on PC.

The Keras' 2D CNN structure requires the input to be a 4D tensor with shape: [batch, rows, columns, channels] (Chollet, 2015). For the data used in this work, "batch" refers to fix-length windows; "rows" refers to observations within each window; "columns" refers to the nine readings; "channels" refers to different locations. In this work, there are 52,444 batches in training set, 16,532 batches in testing set and 3,895 batches in validation set. Number of row is 10 and 20 for windows size of 10 and 20 respectively. Number of columns is always nine. Number of channels depend on how many locations are used in the experiment. In this work different configurations of locations were tested including:

1. Waist alone (one location)
2. Head alone (one location)
3. Chest alone (one location)
4. Wrist alone (one location)
5. Thigh alone (one location)
6. Ankle alone (one location)
7. Waist + thigh (two locations)
8. Waist + thigh + wrist (three locations)
9. Waist + thigh + wrist + chest (four locations)
10. Waist + thigh + wrist + chest + head (five locations)
11. All six locations

Figure 3.20 is an example input data. This example contains one batch (window) of data, when window size is 10, and number of channels is two. This example data have a shape of [1, 10, 9, 2].

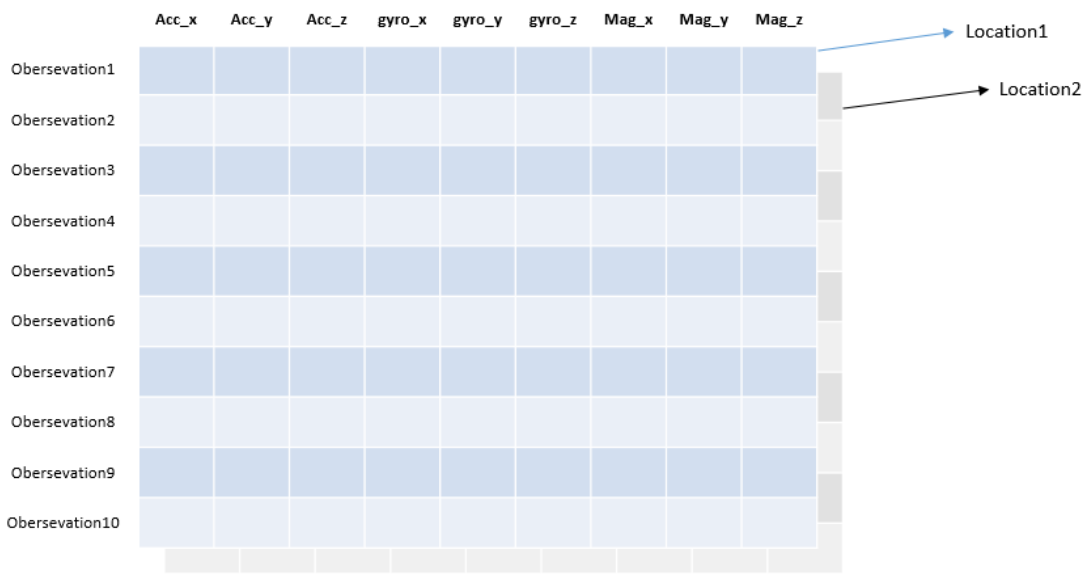


Figure 3.20. Example of input data structure for Keras 2D CNN
(one volunteer, one activity, one trial, one window, two channels (locations)

The output of the model is the activity group (fall or non-fall). The output is a "one-hot vector" with two values in it. Both values are binary numbers. One of the value refers to fall activity and the other one refers to ADLs. When the activity belongs to fall activity, the "one-hot vector" is [1, 0]; when the activity belongs to ADLs, the "one-hot vector" is [0, 1].

3.3.2.3 Structure of 2D CNN

The CNN structure used in this thesis is developed based on LeNet (LeCun, Bottou, Bengio, & Haffner, 1998) with some simple modification.

LeNet was a CNN structure proposed by LeCun et al. (1998) to recognize hand written numbers. In LeNet, there are two convolution groups and three fully connected layers. Each Convolution groups consists of one convolution layer and one subsampling layer.

Compared to simple CNN models which only has one or multiple convolution layers connected in series followed by a couple of fully-connected layers, LeNet has the capability of doing more complicated feature extraction and better generalization. Meanwhile, LeNet is a fairly simple structure as compared to deeper CNN like VGG16 (Simonyan & Zisserman, 2014). Because the dataset used in this work is fairly simple as compared to large images that very complicated feature space, LeNet should have the proper complexity to deal with this dataset.

The structure of the CNN used in this work is shown in Figure 3.21.

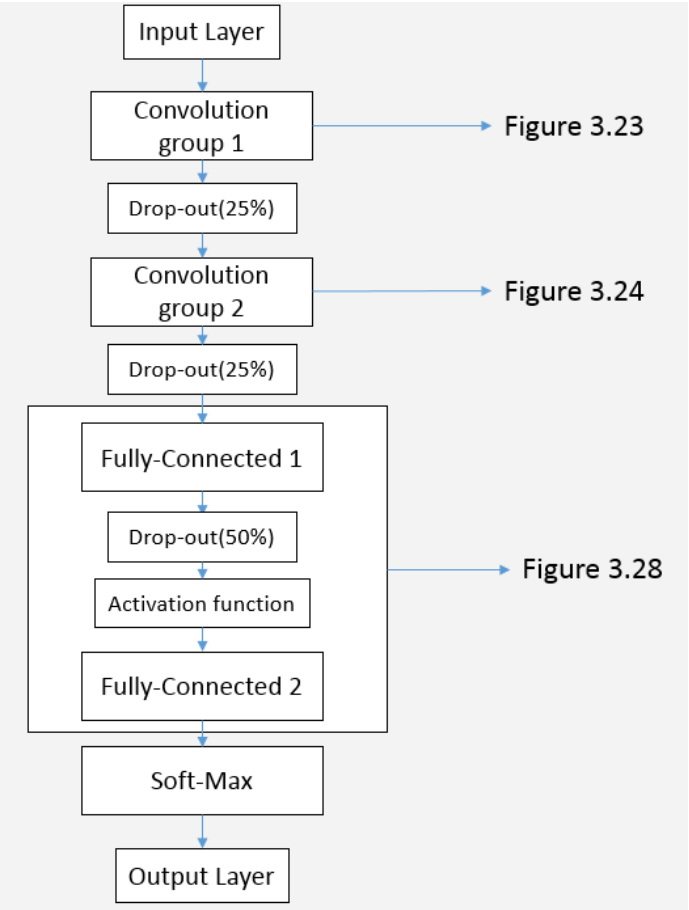


Figure 3.21. CNN structure used in this work

This model is a sequential model that mainly consists of two convolution groups and two fully connected layers. Convolution layer has been described in Chapter 3.3.2.1. Fully connected layer is another type of layer in neural network. In fully connected layers, each hidden nodes is connected every input. Figure 3.22 is an example of a fully connected layer. This example layer has two hidden nodes, and the input size to the layer is three.

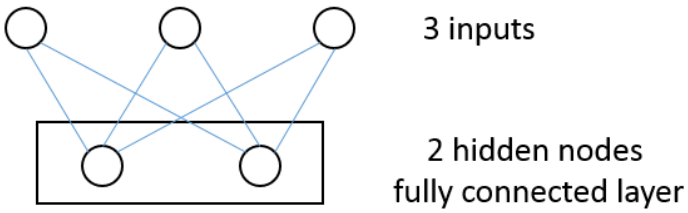


Figure 3.22. Example fully connected layer (two hidden nodes, input size of 3)

Between the first convolution group and the second convolution group, there is a drop-out layer with 25% dropout rate (Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov, 2014). Dropout is a concept developed to deal with overfitting effect and increase neural network's capability on generalization rate (Srivastava et al., 2014).

During the training process of neural network, it happens a lot that some parameters have dominant influence on the output as compared to the other parameters. In this case, the neural network can achieve good result as long as the dominant parameters are close to the ground truth. If one or some of these dominant parameters do not have as much influence to new data as to the training data, very serious overfitting effect will occur. As a result, dropout is implemented to force the neural network to ignore a certain percentage of parameters at each training epoch to avoid certain parameters overly influencing the output of the neural network. The percentage of the ignored parameters is called dropout rate (Srivastava et al., 2014). Between the two fully connected layers, there is drop-out layer with 50% drop-out rate, and also, there is an activation function. The activation functions used in CNN is called rectified linear unit (ReLU). It is defined by the equation (6), and a plot of ReLU function is shown in Figure 3.22 (Nair & Hinton, 2010).

$$y = \max(0, x) \qquad (6)$$

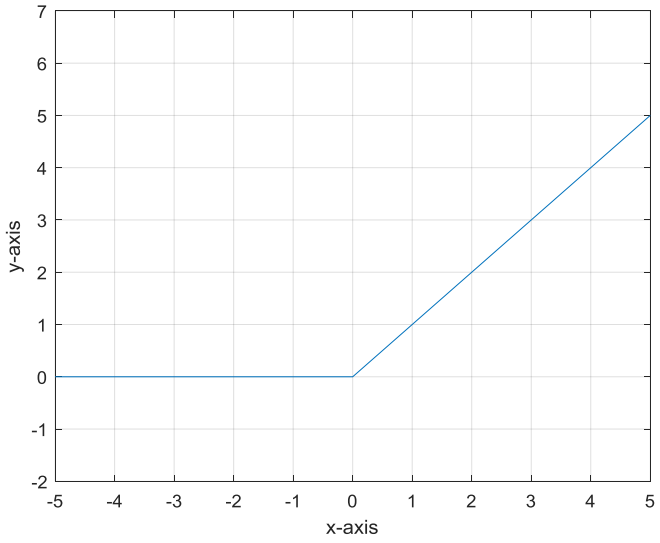


Figure 3.23. Plot of ReLU function

After the two fully connected layers, there is a soft max layer to convert the output of fully connected layer into probability distribution, all the output from soft max layer are floating numbers between 0 and 1, and the sum of these numbers is 1. Soft max can be defined using the following equation:

$$f(s_i) = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad (7)$$

In this equation, C stands for number of classes.

Finally, output layer used “argmax” function to convert the probability distribution from soft max layer to prediction labels. In this work, there are only two different classes for CNN (fall and non-fall), so the output is 0 for non-fall and 1 for fall.

The structure of the two convolution groups are shown in Figures 3.24 and 3.25.

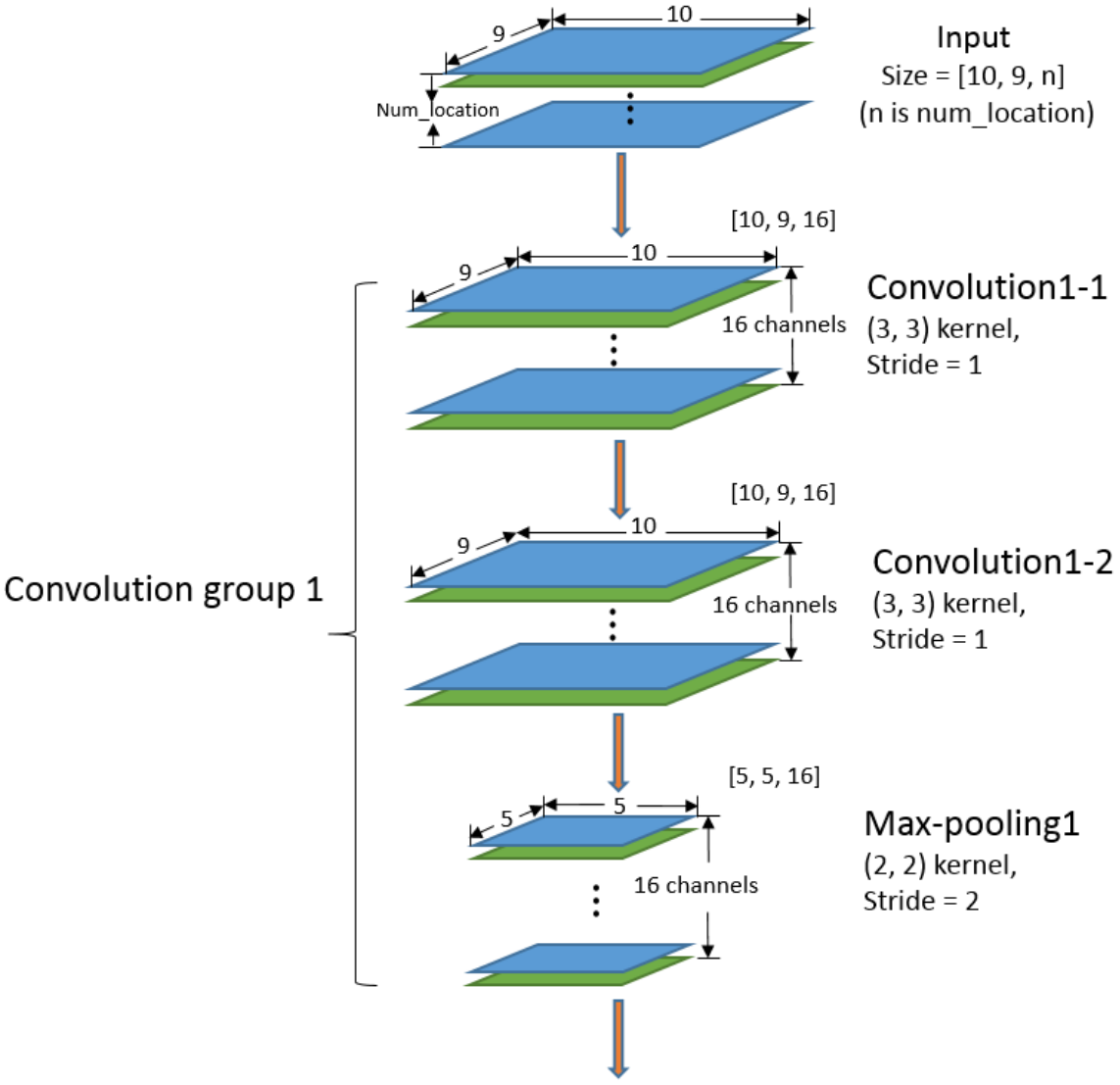


Figure 3.24. Structure of first convolution group for 16-filter model, when window_size = 10

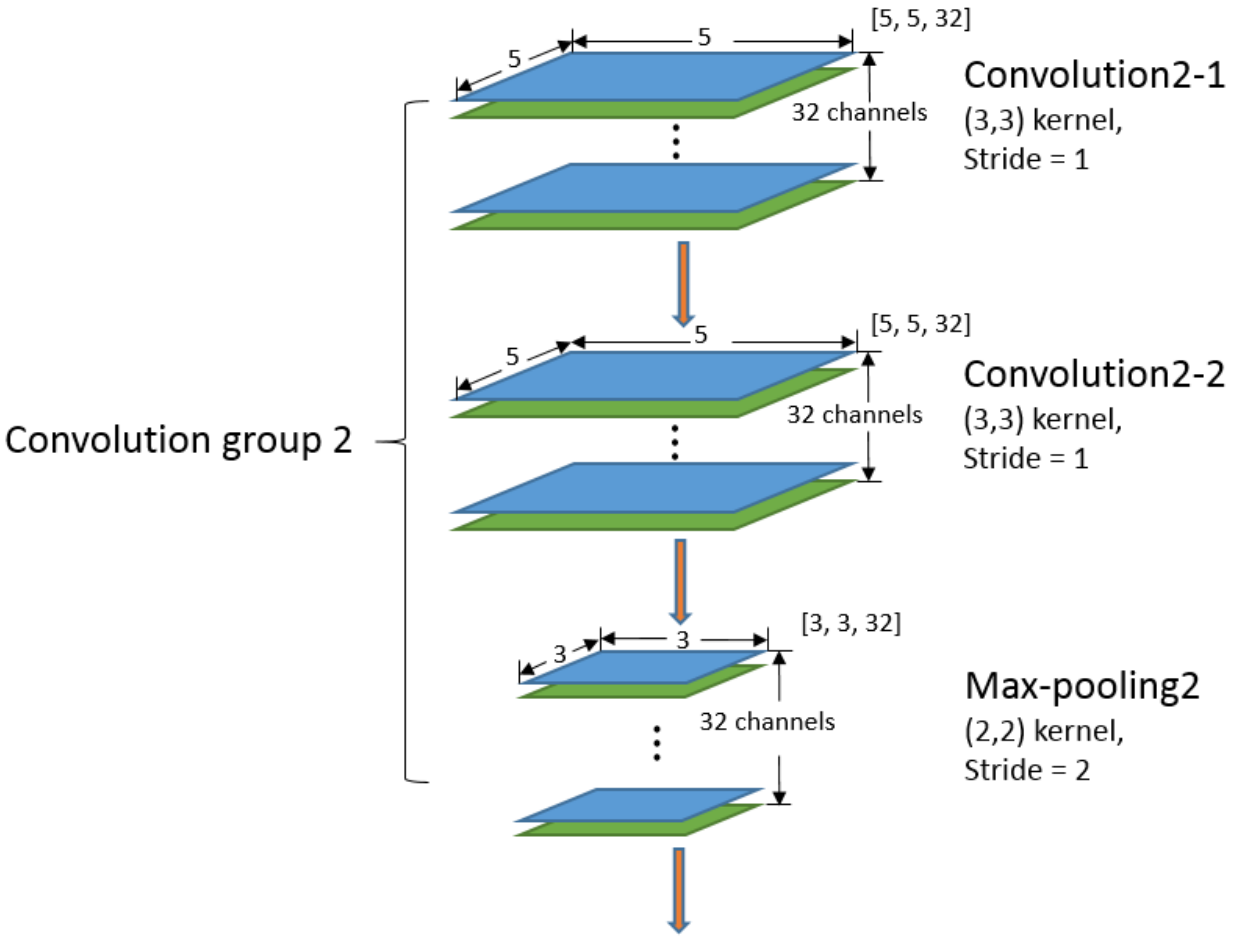


Figure 3.25. Structure of second convolution group for 16-filter model, when window_size = 10

Each convolution group consists of two convolution layers and one max-pooling layer. Max-pooling layer is used to reduce the size of feature maps (Nielsen, 2015). Take filter size = (2, 2), stride = two max-pooling layer for example, Figures 3.23-3.25 show such max-pooling layer applied to an 8 x 8 feature map:

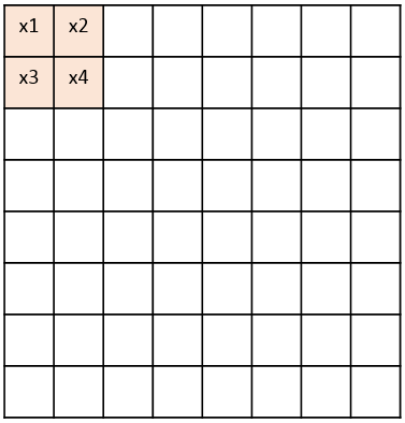


Figure 3.26. Max-pooling-layer 1

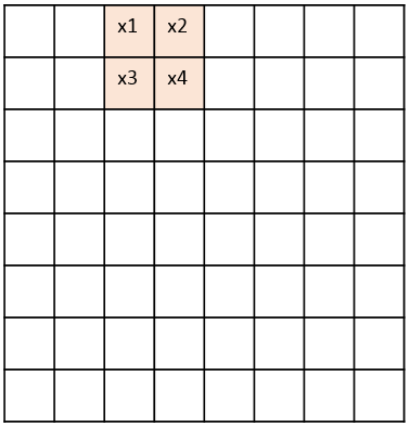


Figure 3.27. Max-pooling-layer 2

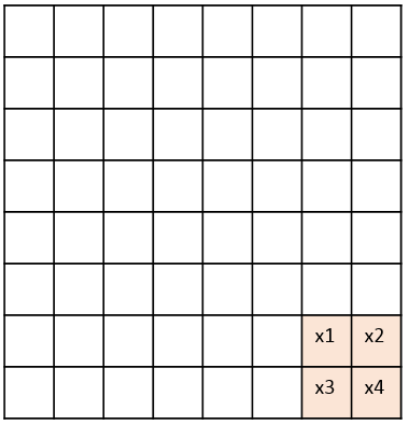


Figure 3.28. Max-pooling-layer 3

Max-pooling layer is essentially a maximum filter. At the first step, as is shown in Figure 3.26, the max filter is at the top left corner of the feature map. The max-pooling layer finds out

the maximum value of x1, x2, x3 and x4, and use that max value as output of the first step. At the second step, as is shown in Figure 3.26, the max filter moves two steps to the right because stride equals 2, and again, max-pooling layer finds out the maximum value of x1, x2, x3 and x4. The max-pooling layer continues repeating this process until the max filter reaches the right bottom corner of the feature map as is shown in Figure 3.28. For the 8*8 feature map as is shown in Figures 3.26-3.28, this process is repeated for 4*4 = 16 times, and the 16 output value forms a 4*4 new feature map. This new feature map is the output of max-pooling layer. In the example above, there are even number of rows and columns. In the case that there are odd number rows or columns, a column/row of $-\infty$ will be padded to the original data so that the number of rows and columns are even number (Nielsen, 2015). The max-pooling layer used in this work, uses filter of size (2, 2), and stride of 2.

The convolution layers implemented in this work use (3 3) kernel with stride equals to 1. Models with different number of filters were tested in this work, including 8-filter model, 16-filter model, 32-filter model and 64-filter model. Eight-filter model means that eight filters are used in the first convolution layer in the model. Similarly, for 16-filter model, 16 filters are used in the first convolution layer; for 32-filter model, 32 filters are used in the first convolution layer; for 64-filter model, 64 filters are used in the first convolution layer. Figures 3.24 and 3.25 show the structure of the two convolution groups for a 16-filter model. In Figure 3.23, the two convolution layers in the first convolution group have 16 filters of size (3, 3). Because the max-pooling layer in the first convolution group reduces the size of feature map from (9 10) to (5 5), the convolution layers in the second convolution group need to have more filters to compensate for the information loss caused by small feature map. So for 16-filter model, 16*2 = 32 filters were used for the convolution layers in the second convolution group. The max-pooling layer in the second convolution group again reduce the size of the feature map from (5 5) to (3 3).

The structure of the fully connected layers are shown in Figure 3.29. The first fully connected layer needs to have the number of hidden nodes that roughly matches the number of features extracted from the convolution layers. Normally, the number of hidden nodes is 4 times the number of filters in the last convolution layer. So for a 16-filter model, as is shown in Figure 3.29, the first fully connected layer contains 16*2*4 = 32*4 = 128 hidden nodes. The second fully connected layer needs to have the number of hidden nodes that equals to the number of

classes in the classification problem. In this work, the model is only designed to distinguish fall and non-fall activities. So there are two hidden nodes in the second fully connected layer.

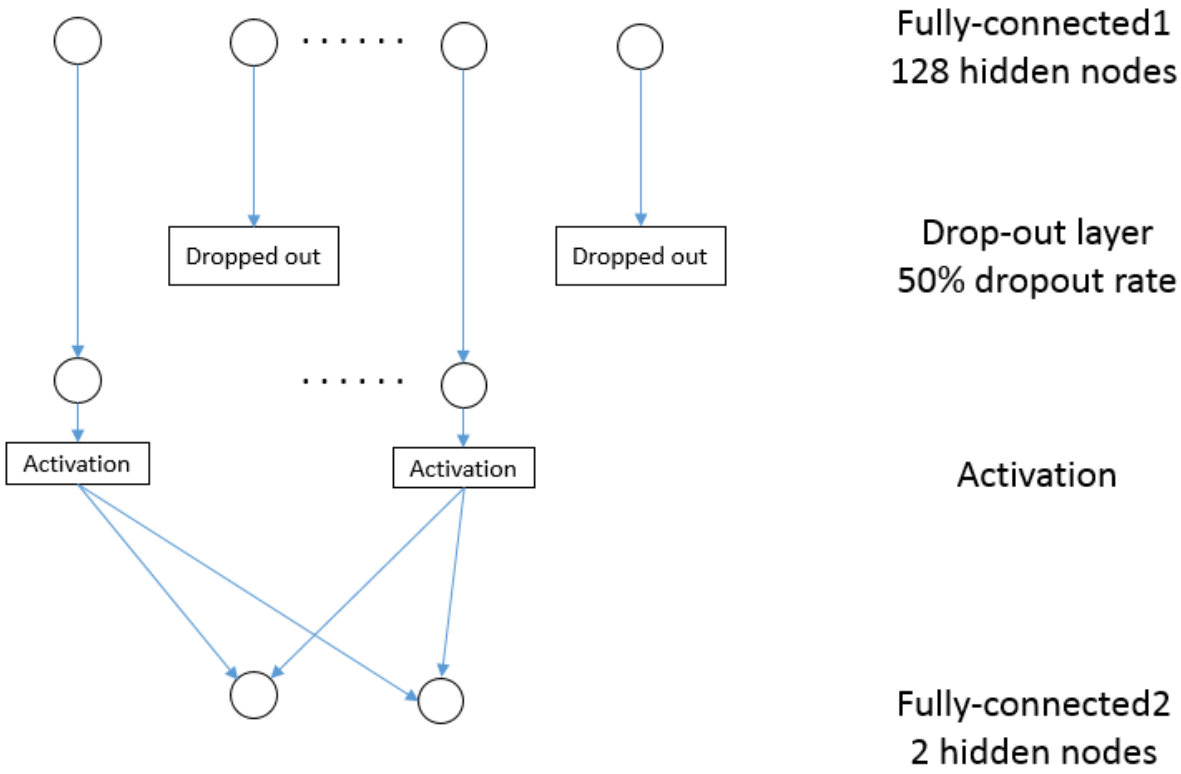


Figure 3.29. Structure of fully connected layers for 16-filter model

3.3.3 Recurrent Neural Network

The RNN structure used in this work is called “long short term memory (LSTM).”

3.3.3.1 How long short term memory work (LSTM)

LSTM is a type structure developed based on vanilla RNN. In vanilla RNN, the parameters of the neural network are multiplied by themselves repeatedly during backpropagation in training process. This will cause exploding gradient in the case when the parameter is larger than 1, and vanishing gradient in the case when the parameter is smaller than 1. Long short term memory/LSTM is developed to resolve the vanishing/exploding gradient problem. (Hochreiter & Schmidhuber, 1997)

LSTM can be expressed mathematically using the following equations (Hochreiter & Schmidhuber, 1997):

$$\begin{cases} i_t = \sigma_i(W_{ih}h_{t-1} + W_{ix}x_t + b_i) \\ f_t = \sigma_f(W_{fh}h_{t-1} + W_{fx}x_t + b_f) \\ o_t = \sigma_o(W_{oh}h_{t-1} + W_{ox}x_t + b_o) \\ g_t = \sigma_g(W_{gh}h_{t-1} + W_{gx}x_t + b_g) \end{cases} \quad (8)$$

$$c_t = f_t c_{t-1} + i_t g_t \quad (9)$$

$$h_t = o_t \sigma_c(c_t) \quad (10)$$

$$y_t = \sigma_y(W_y h_t + b_y) \quad (11)$$

In LSTM, there are two hidden structures. One is called hidden states, which is represented as h_t in equations (8)-(11). Both vanilla RNN and LSTM have this hidden structure to store information from previous data samples. The second structure is called hidden cell, and is represented as "c" in equation (8) – (11). Hidden cell is unique structure that LSTM has. This structure is created so that the recurrent network has a more straight forward gradient flow during training process, so that neither exploding gradient nor vanishing gradient would happen. Also, in LSTM, there are four hidden gates, which are represented using i, f, o and g in equation (8) - (11). This four gates decide:

1. Whether data from new input should be stored in hidden cell.
2. Whether the information stored in current hidden structure should be carried onto the next hidden structure.
3. Whether the information stored in the hidden cell should be used for the output.

Equation (5) is used to calculate the four gates. σ represents activation function. Among these four activation functions, σ_g is tanh, and the other three are sigmoid. Figure 3.30 is a plot of tanh function; Sigmoid function is defined in the equation (12), and Figure 3.31 is a plot of a sigmoid function:

$$y(x) = \frac{1}{1+e^{-x}} \quad (12)$$

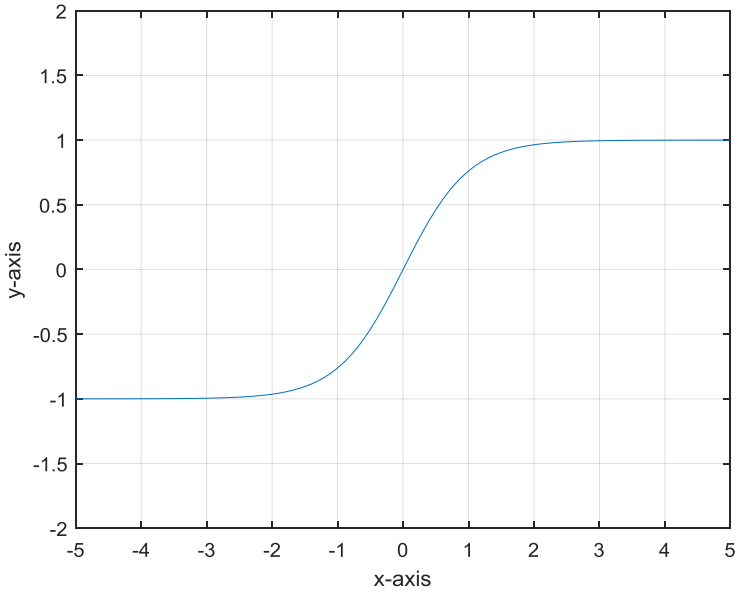


Figure 3.30. Figure for “tanh” function

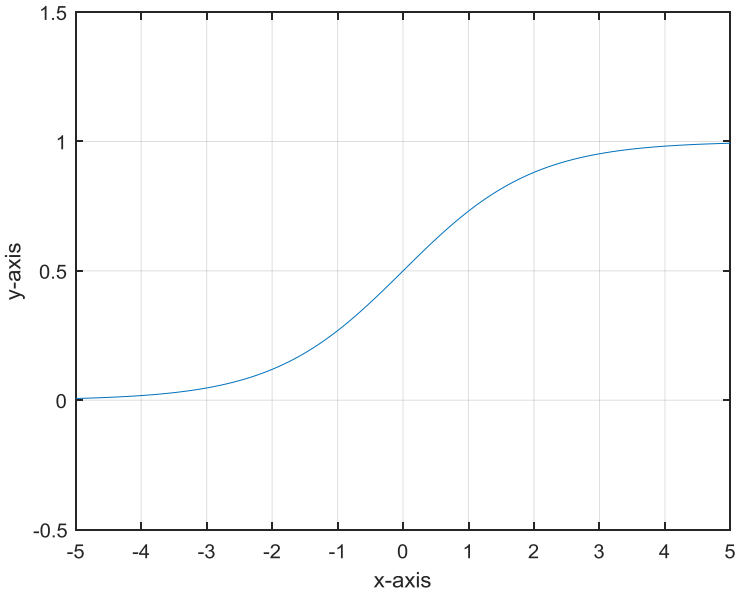


Figure 3.31. Plot of sigmoid function

This 4-gate structure gives LSTM a better capability to select useful information as compared to simple RNN. Figure 3.32 is a demonstration of the LSTM structure at one recurrent step.

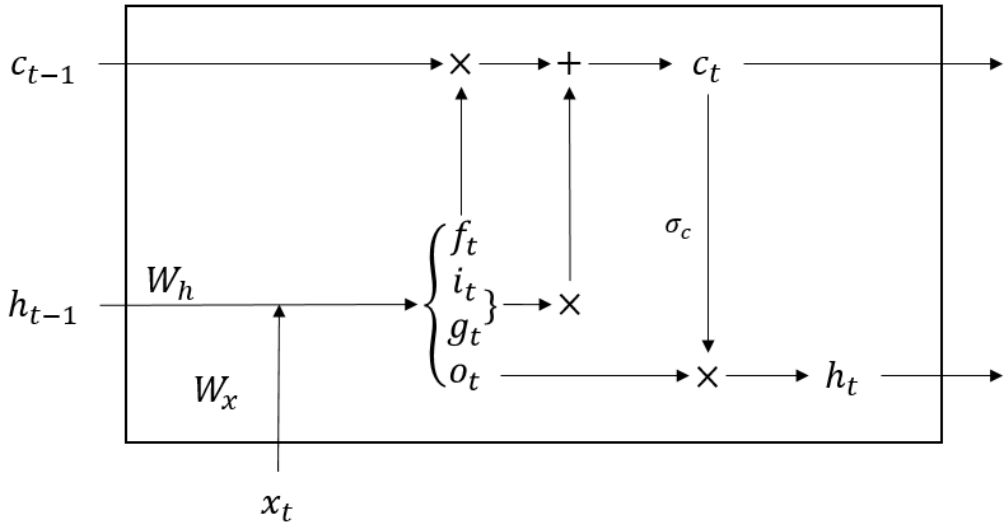


Figure 3.32. LSTM at one recurrent step

3.3.3.2 Data preparation

Like CNN, RNN was built using Tensorflow with Keras backend. The code was written in python language using anaconda platform, spyder IDE.

The input format for RNN is different from the one for CNN. For RNN, each data sample should be a sequence of data. For the dataset used in this work, one data sequence is one trial of activity performed by one volunteer. Each data sequence should consist of a sequence of fixed length slicing window, and each window is a 2D data for single sensor input, and multichannel 2D data for multi-sensor input.

In RNN, each input data sample needs to have the same size. But the lengths of each data file are different. Before inputting the data into RNN, some cropping were used to make the length of each data file consistent. In this work, all the data file were truncated to 150 data samples using the following algorithm:

1. Read in the dataset, and read in the relabel index. The relabel index is generated by the data preprocessing algorithm which is mentioned in Chapter 3.3.1.
2. If index_end < 150: The first 150 data samples in the data file are kept.

3. Else if (index_end – index_start) < 150: data from (index_end – 150) to index_end are kept
4. Else: data from index_start to (index_start + 150).
5. The 150 data samples within each data file were divided into 15 windows so that each window has 10 data samples.
6.

Figure 3.33 shows the flow chart of the data preparation algorithm:

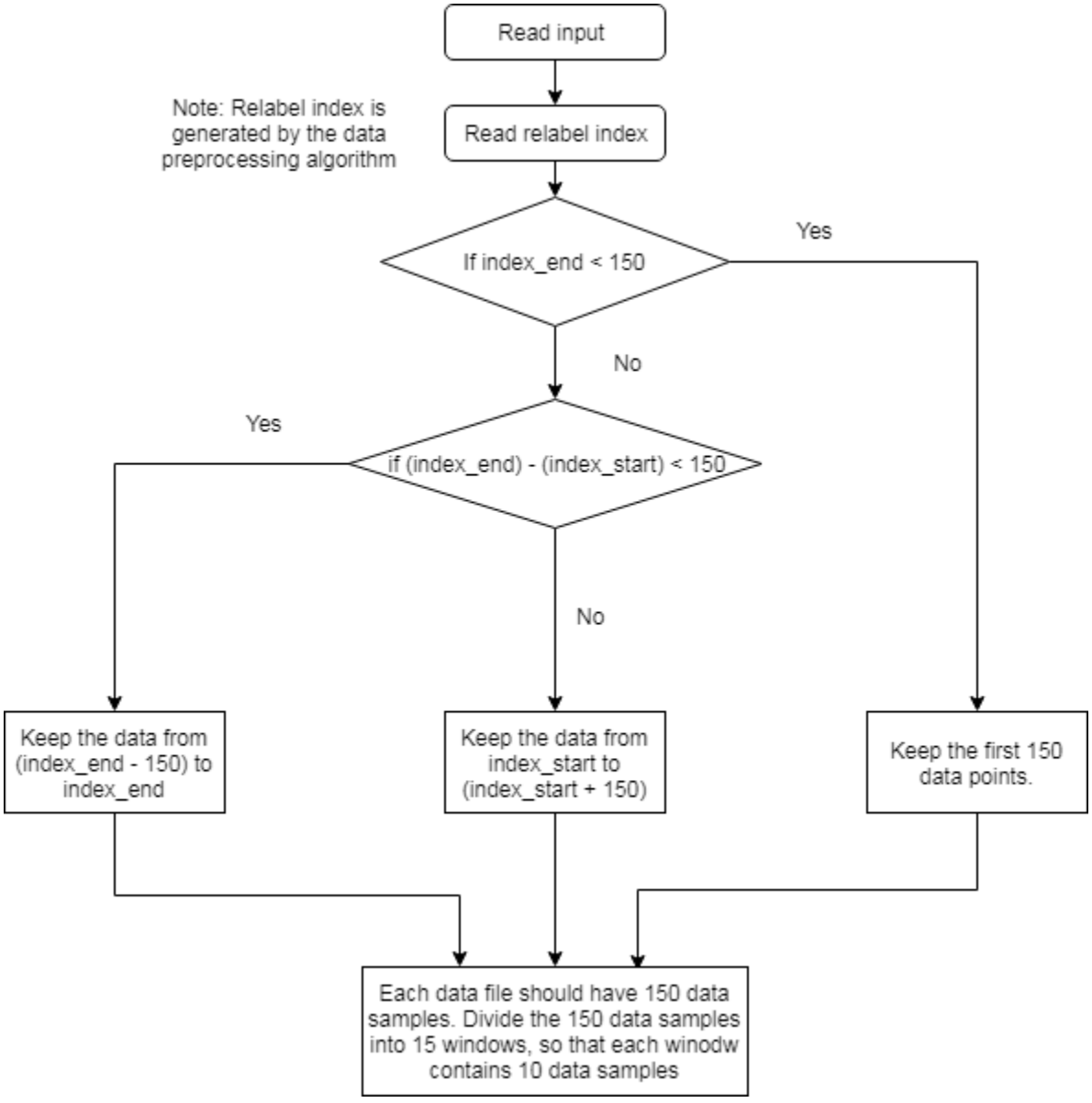


Figure 3.33. RNN data preparation algorithm

Part of the data that are kept belong to the “pre/postcondition” which was mentioned in earlier chapter. The pre/postcondition include activities like standing still, sitting and lying down. These activities were considered as the third category other than fall activity and non-fall activity. These activities will be referred as “pre/postcondition” later on. As a result, output of the RNN are one-hot vectors that have three values. Each one-hot vector is a prediction for one data window. [1,0,0] refers to “pre/postcondition”, [0,1,0] refers to “fall-activity,” and [0,0,1] refers to “non-fall activities.” The reason that 150 is chosen to be the length of each data file is that when each data file is cropped to 150, the number of data in each of the three classes are balanced.

3.3.3.3 LSTM model with convolution layers

One of the RNN models tested in this work used LSTM combined with convolution layers

The convolution structure used in this model is the same with the one used in 2D CNN which is mentioned in 3.3.2.1. It consists of two convolution layers. Each convolution layers has two convolution layers to extract features and one max-pooling layers to reduce the size of feature space. For each convolution groups, there is a dropout layer with 50% drop out rate to reduce the effect of overfitting. Because the size of the feature map in the second convolution group is half of the one in the first convolution group due to the max-pooling layer, the convolution layers in the second convolution group have twice as many filters as the convolution layers in the first convolution group.

The 2D output of the two convolution groups is flattened into a 1D vector and used as input to a single LSTM layer. Within the LSTM layer, there is a 50% drop off rate for the recurrent parameters. And for each RNN layers, different number of hidden nodes were tested.

After the RNN layers, there is a fully connected layer with three hidden nodes, and a soft-max layer to convert possibility distribution into one-hot vector.

The structure of this RNN model is shown in Figure 3.34. This figure shows the model with one layer of simple RNN. When multiple RNN layers are used, the additional RNN layers are connected in serious between the first RNN layer and fully connected layer.

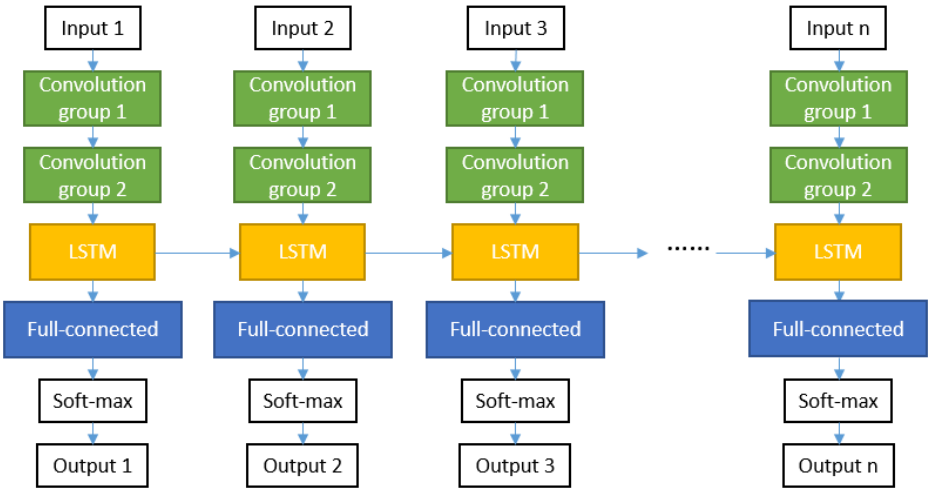


Figure 3.34. Structure of LSTM with Convolution layers

3.3.3.4 LSTM with convolution transformation

The other RNN structure used in this work is LSTM with convolution transformation. In this structure, only one LSTM layer and one fully-connected layer are used. The input data is fed directly into LSTM layer. After the LSTM layer, there is one fully-connected layer and one soft-max layer. Yet different from normal LSTM layer, the LSTM layer in this structure uses convolution transformation instead of linear transformation for input and recurrent transformation. The structure of this model is shown in Figure 3.35.

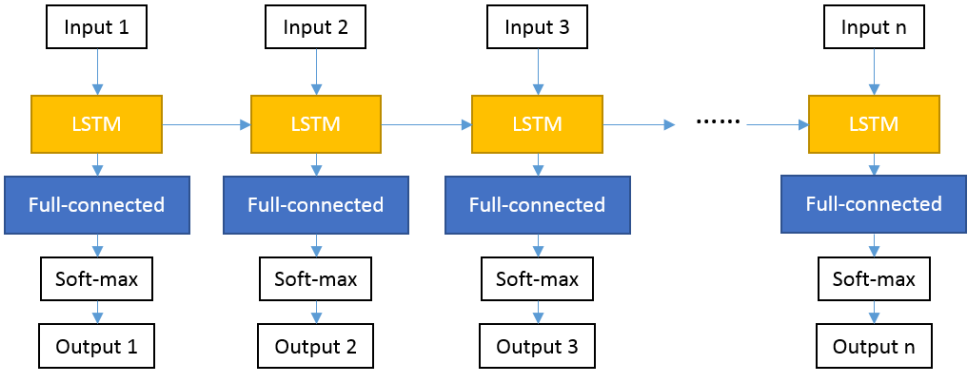


Figure 3.35. Structure of LSTM model with convolution transformation

3.3.4 Neural Network Training, Validation and Testing

Neural network models in this work use the following training setup:

1. Number of epochs: 100
2. Batch size = 30
3. Optimizer: Stochastic Gradient Descent
4. Loss function: Categorical cross-entropy
5. Learning rate used in this work is different for CNN and RNN. For CNN, the learning rate is 0.005 with no decay; for RNN, the learning rate is 0.002 with no decay.

The loss function used in this work is Categorical cross-entropy. It is a type of loss function is normally used in classification problems when only one result is correct. The loss function is defined as:

$$CE = \sum_i^C t_i \log(f(s_i)) \qquad (13)$$

In equation (13), CE stands for cross-entropy loss, C stands for number of classes, t_i is ground truth value and s_i is the score from the model. $f(s_i)$ is soft max function, and was defined early in equation (7). The optimization algorithm used in this work is Stochastic Gradient Descent. Gradient Descent is one of the most commonly used algorithm for optimizing mathematical models. This algorithm calculates the loss at every training step, then calculate the gradients of every parameter within the neural network with respect to the loss, and then update the parameters based on the gradients and loss. As compared to normal gradient descent, Stochastic Gradient Descent uses a small batch of data to calculate the loss instead of the whole dataset. As a result, when dealing with large dataset, Stochastic Gradient Descent has a faster computation speed than normal Gradient Descent. (Bottou, 2010)

As mentioned in Chapter 3.3.1, the dataset was divided into training set, validation set and testing set. The training set has data from 12 volunteers; testing set has data from four volunteers; validation set has data from one volunteer. For CNN model, the training set contains 52444 data samples (23969 fall samples, 28475 non-fall samples); testing set has data from four volunteers, containing 16532 data samples (7120 fall samples, 9412 non-fall samples); validation set has data from one volunteer, containing 3895 (1812 fall samples, 2070 non-fall samples). For RNN, the training set contains 2056 data samples/sequences, 30840 windows (9568 non-fall

windows, 11610 fall windows, and 9662 pre/postcondition windows); the testing set contains 632 data samples/sequences, 9480 data windows (3046 non-fall windows, 3482 fall windows and 2952 non-fall windows); the validation set contains 154 data samples/sequences, 2310 data windows (731 non-fall windows, 893 fall windows, and 686 pre/postcondition windows).

Initially, all the models are trained using the training set, and a training accuracy and loss are generated in real time; meanwhile, for each training epoch, the model is tested using validation set to generate a validation accuracy and loss. Both training accuracy/loss and validation accuracy/loss are generated during the training process, and these four values are plotted on training curves. After training for 100 epochs, the trained model is tested using the test set, and generate a test accuracy, a test loss, and a test result which contains a table of predictions. Based on the test result, sensitivity and specificity can be calculated using the following equations:

$$Sensitivity = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + number\ of\ false\ negatives} \quad (14)$$

$$Specificity = \frac{number\ of\ true\ negatives}{number\ of\ true\ negatives + number\ of\ false\ positives} \quad (15)$$

Test accuracy, sensitivity and specificity are used to make initial comparison between models.

In the experiments, two CNN models were found to have close performance based on these three parameters, and these two models were further compared using 10-fold cross validation.

In 10-fold cross validation process, the training set is divided evenly into 10 groups, and each group is called one fold. At each step, one fold is used for validation, and the other nine folds are used for training, and a validation accuracy can be obtained. This step is repeated for 10 times so that each fold is used for validation one time. For 10-fold cross validation, 10 validation accuracy can be obtained. The average of the 10 validation accuracies is used to compare the performance of different models.

CHAPTER 4. RESULTS

4.1 Convolution Neural Network

This chapter includes the results for all the experiment, and also the analysis and conclusion based on these results. All the accuracy on testing set, along with the sensitivity and specificity are recorded in Tables 4.1-4.10; all the training accuracy/loss and validation accuracy/loss are plotted and shown in Figures 4.1-4.10.

4.1.1 Comparison between different sensor setups

4.1.1.1 Comparison between different number of sensors

Theoretically, data from more locations contain more adequate and complete information. As a result, the model should have a better performance when more locations are used. Yet from real world application prospective, more locations means less convenience for the patients.

In this work, data from one location were tested initially. The locations that was used is waist because waist is normally considered to be the body part that has the least redundant movements. The result can be seen in Tables 4.1 and 4.2. The accuracy of the CNN ranges from 94.02% (8-filter model, window size is 10), to 96.65% (64-filter model, window size is 20).

Table 4.1 Experiment with data of 10 window size, from waist sensor

a. 8-filter model

8filter_10window_1location		
	TRUE	FALSE
Fall	6849	271
non_fall	8695	717
Accuracy	Sensitivity	Specificity
94.02%	96.19%	92.38%

b. 16-filter model

16filter_10window_1location		
	TRUE	FALSE
Fall	6901	219
non_fall	8856	556
Accuracy	Sensitivity	Specificity
95.31%	96.92%	94.09%

c. 32-filter model

32filter_10window_1location		
	TRUE	FALSE
Fall	6899	221
non_fall	9014	398
Accuracy	Sensitivity	Specificity
96.25%	96.89%	95.77%

d. 64-filter model

64filter_10window_1location		
	TRUE	FALSE
Fall	6886	234
non_fall	8967	445
Accuracy	Sensitivity	Specificity
95.89%	96.71%	95.27%

Table 4.2 Experiment with data of 20 window size, from waist sensor

a. 8-filter model

8filter_20window_1location		
	TRUE	FALSE
Fall	3236	134
non_fall	4346	234
Accuracy	Sensitivity	Specificity
95.37%	96.02%	94.89%

b. 16-filter model

16filter_20window_1location		
	TRUE	FALSE
Fall	3226	144
non_fall	4398	182
Accuracy	Sensitivity	Specificity
95.90%	95.3%	96.03%

c. 32-filter model

32filter_20window_1location		
	TRUE	FALSE
Fall	3221	149
non_fall	4437	143
Accuracy	Sensitivity	Specificity
96.33%	95.58%	96.88%

d. 64-filter model

64filter_20window_1location		
	TRUE	FALSE
Fall	3292	78
non_fall	4392	188
Accuracy	Sensitivity	Specificity
96.65%	97.69%	95.90%

Then two locations input were tested. In addition to waist, data from thigh was used, because thigh contains additional information from the movement on legs, which can be useful for activity recognition. The result can be seen in Tables 4.3 and 4.4.

Table 4.3 Experiment with data of 10 window size, from waist sensor & thigh sensor

a. 8-filter model

8filter_10window_2location		
	TRUE	FALSE
Fall	6510	610
non_fall	8635	777
Accuracy	Sensitivity	Specificity
91.61%	91.43%	91.75%

b. 16-filter model

16filter_10window_2location		
	TRUE	FALSE
Fall	6672	448
non_fall	8961	451
Accuracy	Sensitivity	Specificity
94.56%	93.71%	95.21%

c. 32-filter model

32filter_10window_2location		
	TRUE	FALSE
Fall	5845	1275
non_fall	9193	219
Accuracy	Sensitivity	Specificity
90.96%	82.09%	97.67%

d. 64-filter model

64filter_10window_2sensor		
	TRUE	FALSE
Fall	6742	378
non_fall	8981	431
Accuracy	Sensitivity	Specificity
95.11%	94.69%	95.42%

Table 4.4 Experiment with data of 20 window size, from waist sensor & thigh sensor

a. 8-filter model

8filter_20window_2location		
	TRUE	FALSE
Fall	3221	149
non_fall	4282	298
Accuracy	Sensitivity	Specificity
94.38%	95.58%	93.49%

b. 16-filter model

16filter_20window_2location		
	TRUE	FALSE
Fall	3255	115
non_fall	4286	294
Accuracy	Sensitivity	Specificity
94.86%	96.59%	93.58%

c. 32-filter model

32filter_20window_2location		
	TRUE	FALSE
Fall	3219	151
non_fall	4405	175
Accuracy	Sensitivity	Specificity
95.90%	95.52%	96.18%

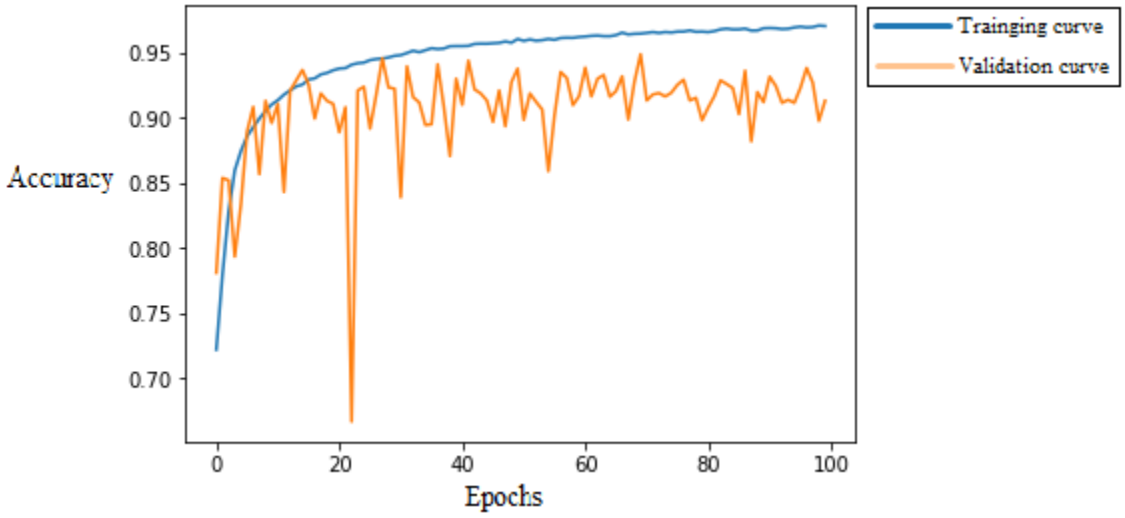
d. 64-filter model

64filter_20window_2location		
	TRUE	FALSE
Fall	3214	156
non_fall	4416	164
Accuracy	Sensitivity	Specificity
95.97%	95.37%	96.42%

It is expected 2-location input can provide better performance to the model as compared to 1-location input. But when comparing Tables 4.3 and 4.4 to Tables 4.1 and 4.2, it can be seen that models with 2-location input have worse performances. In most cases, the accuracy for 2-location input is about 1% lower than that of 1-location input. For the 8-filter model with window size of 10, and the 32-filters with window size of 10, the accuracies for 2-location input are approximately 2.5% lower those for 1-location input.

This decrease on performance is most possibily caused by overfitting. The reason can be seen from the training and validation plot in Figures 4.1 and 4.2.

a) Accuracy plot



b) Loss plot

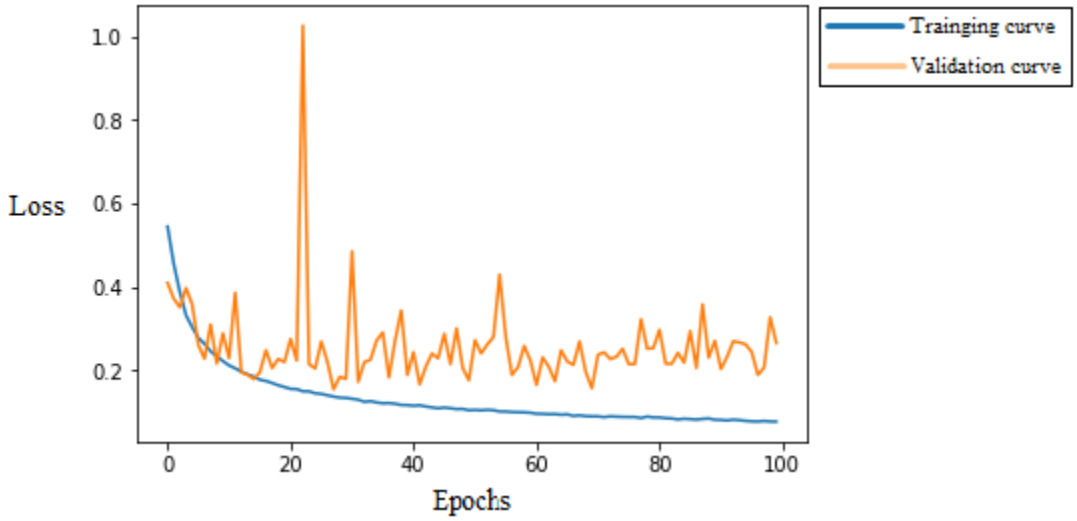
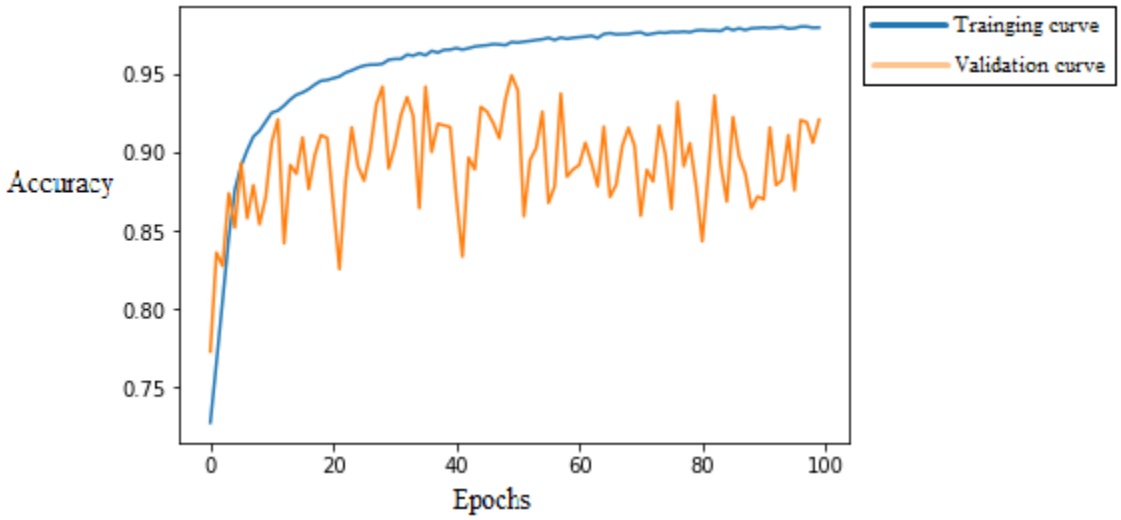


Figure 4.1. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist only

a) Accuracy plot



b) Loss plot

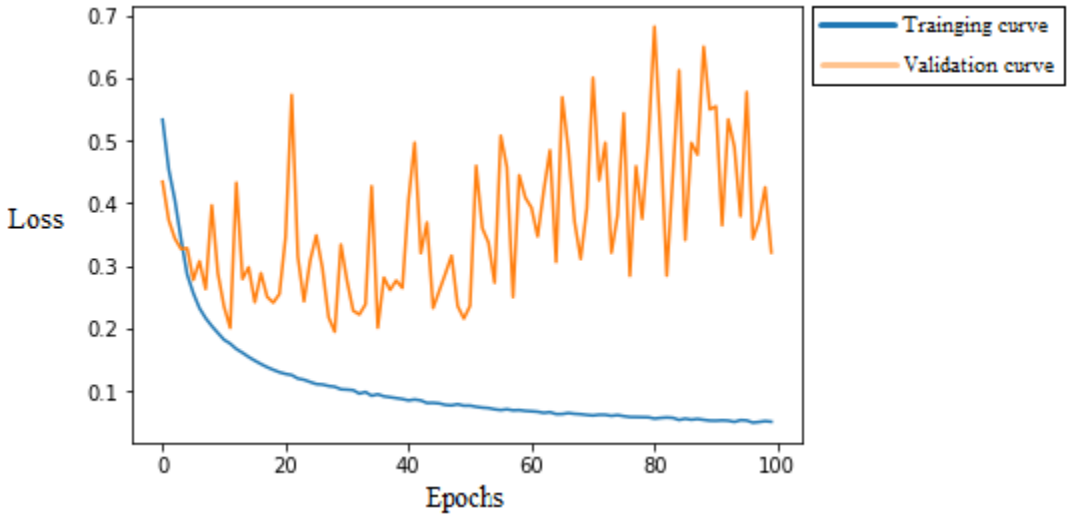


Figure 4.2. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist and thigh

Figure 4.1 shows the training and validation curve for 16-filter model with window size of 10, waist location only.` Figure 4.2 shows the training and validation curve for16-filter model with window size of 10, waist and thigh locations. In Figure 4.1, at around the 30th epochs, validation performance stops increasing, but there is no obvious decreasing trend either. That means for the 1-location model, from the 30th epoch, overfitting starts to appear, but it is so minor that it does not affect the performance of the model a lot. In Figure 4.2 however, at around 15th epoch, validation performance stops increasing, and a decreasing trend can be observed

from the figure. It can be concluded that overfitting in 2-location model happened a lot earlier than it happened in 1-location model. The overfitting is big enough to affect the performance of the model.

To further prove that using input data from more locations cannot help increase the performance of the model, 3-location, 4-location, 5-location and 6-location input were tested. Locations used for 3-location input were waist, thigh and wrist. Locations used for 4-location input were waist, thigh, wrist and chest. Locations used for 5-location input were waist, thigh, wrist, chest, and head. The models used for this test are 16-filter model with window size of 10, and 32-filter model with window size of 10, because these two models have the best performance for single sensor input, which will be discussed later on. The result of the test is recorded in Tables 4.5 and 4.6.

Table 4.5 Comparison between different numbers of sensor locations. Model used is 16-filter model with window size of 10 input.

a. 1 location

16filter_10window_1location		
	TRUE	FALSE
Fall	6901	219
non_fall	8856	556
Accuracy	Sensitivity	Specificity
95.31%	96.92%	94.10%

b. 2 location

16filter_10window_2location		
	TRUE	FALSE
Fall	6672	448
non_fall	8961	451
Accuracy	Sensitivity	Specificity
94.56%	93.71%	95.21%

c. 3 location

16filter_10window_3location		
	TRUE	FALSE
Fall	8845	567
non_fall	6688	432
Accuracy	Sensitivity	Specificity
93.96%	93.98%	93.93%

d. 4 location

16filter_10window_4location		
	TRUE	FALSE
Fall	8877	535
non_fall	6669	451
Accuracy	Sensitivity	Specificity
94.04%	94.32%	93.67%

e. 5 location

16filter_10window_5location		
	TRUE	FALSE
Fall	5961	1159
non_fall	8801	611
Accuracy	Sensitivity	Specificity
89.29%	83.72%	93.51%

Table 4.5 continued

f. 6 location

16filter_10window_6location		
	TRUE	FALSE
Fall	5143	1977
non_fall	9075	337
Accuracy	Sensitivity	Specificity
86.00%	72.23%	96.42%

Table 4.6 Comparison between different numbers of sensor locations. Model used is 32-filter model with window size of 10 input.

a. 1 location

32filter_10window_1location		
	TRUE	FALSE
Fall	6899	221
non_fall	9014	398
Accuracy	Sensitivity	Specificity
96.26%	96.90%	95.77%

b. 2 location

32filter_10window_2location		
	TRUE	FALSE
Fall	6672	448
non_fall	8961	451
Accuracy	Sensitivity	Specificity
94.56%	93.71%	95.21%

c. 3 location

32filter_10window_3location		
	TRUE	FALSE
Fall	8798	614
non_fall	6813	307
Accuracy	Sensitivity	Specificity
94.43%	93.48%	95.69%

Table 4.6 continued

d. 4 location

32filter_10window_4location		
	TRUE	FALSE
Fall	8761	651
non_fall	6603	517
Accuracy	Sensitivity	Specificity
92.93%	93.08%	92.74%

e. 5 location

32filter_10window_5location		
	TRUE	FALSE
Fall	5719	1401
non_fall	8853	559
Accuracy	Sensitivity	Specificity
88.14%	80.32%	94.06%

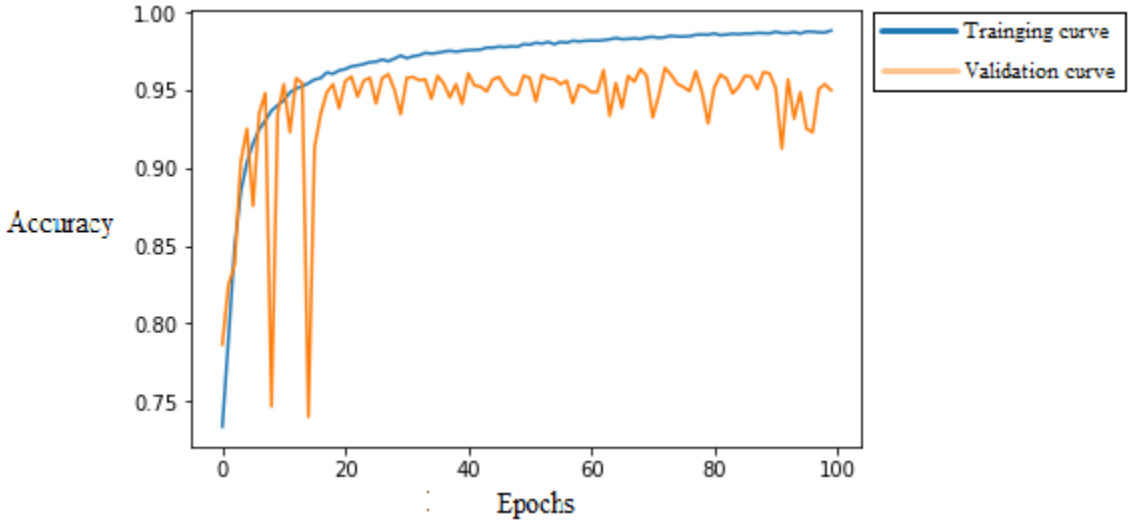
f. 6 location

32filter_10window_6location		
	TRUE	FALSE
Fall	6103	1017
non_fall	8829	483
Accuracy	Sensitivity	Specificity
90.87%	85.72%	94.81%

As shown in Tables 4.5 and 4.6, models with data from 1 location input provided accuracies above 95%. Models with 2-location, 3-location, and 4-location input had similar performances, with accuracies between 92.9% and 94.6%. 5-location and 6-location on the other hand, had the worst performance, with an accuracy around or even lower than 90%.

Figure 4.3-4.6 show the training and validation curves when three to six locations were used.

a) Accuracy plot



b) Loss plot

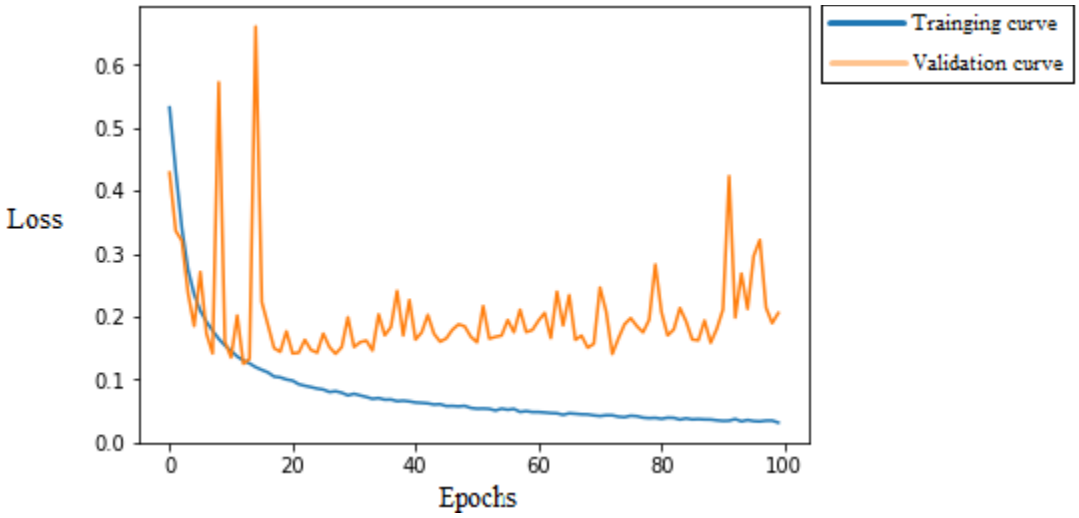
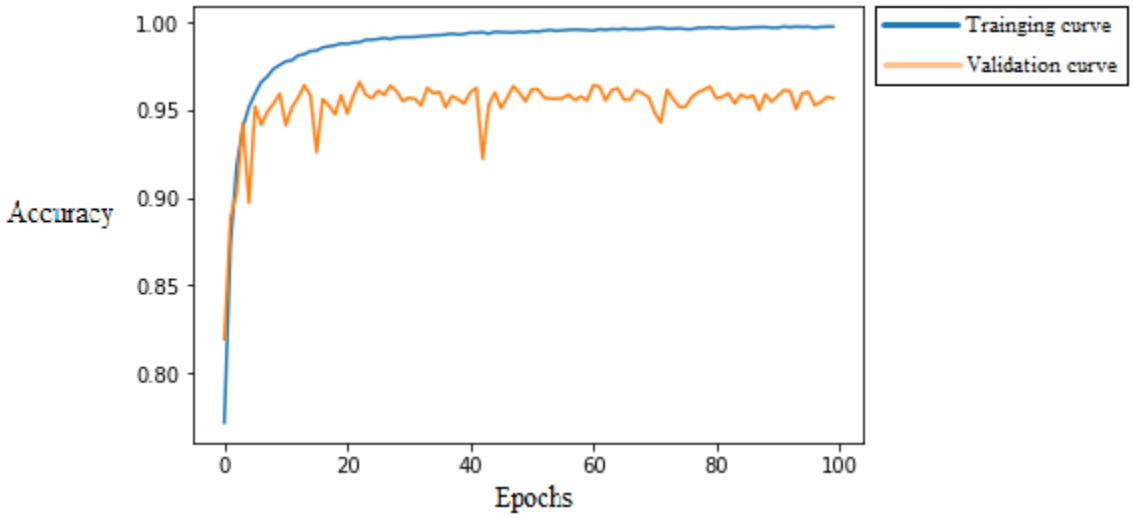


Figure 4.3. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist, thigh & wrist

a) Accuracy plot



b) Loss plot

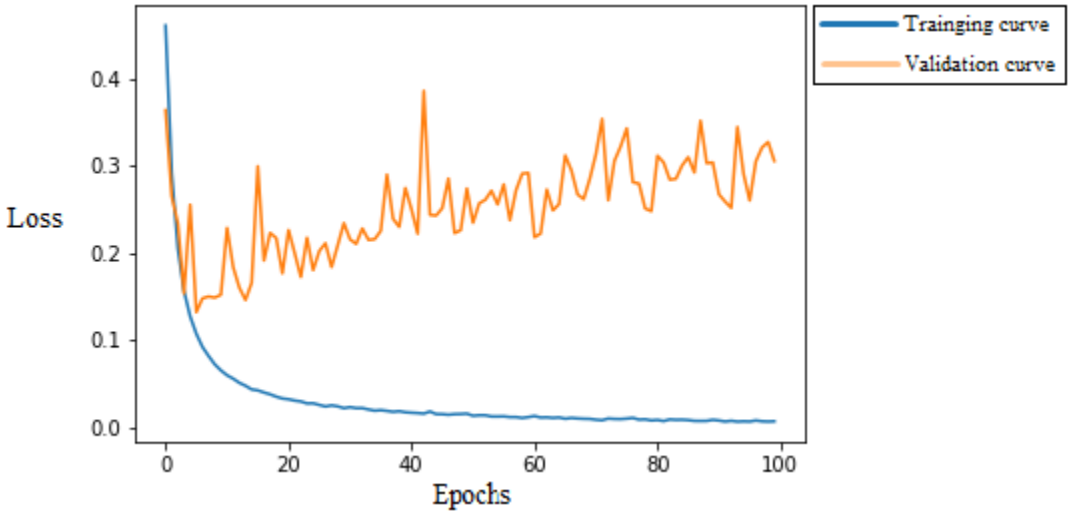
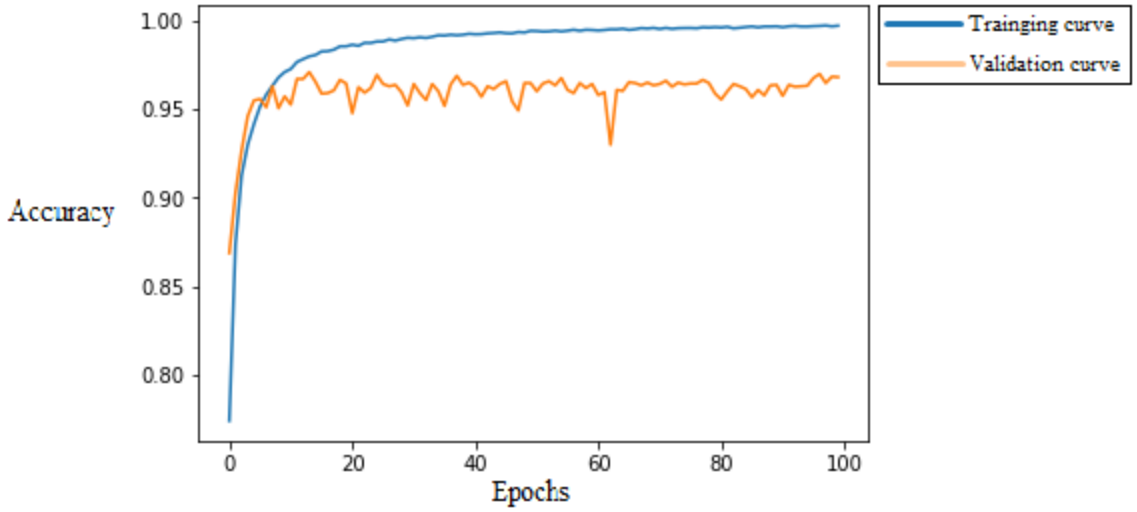


Figure 4.4. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist, thigh, wrist & chest

a) Accuracy plot



b) Loss plot

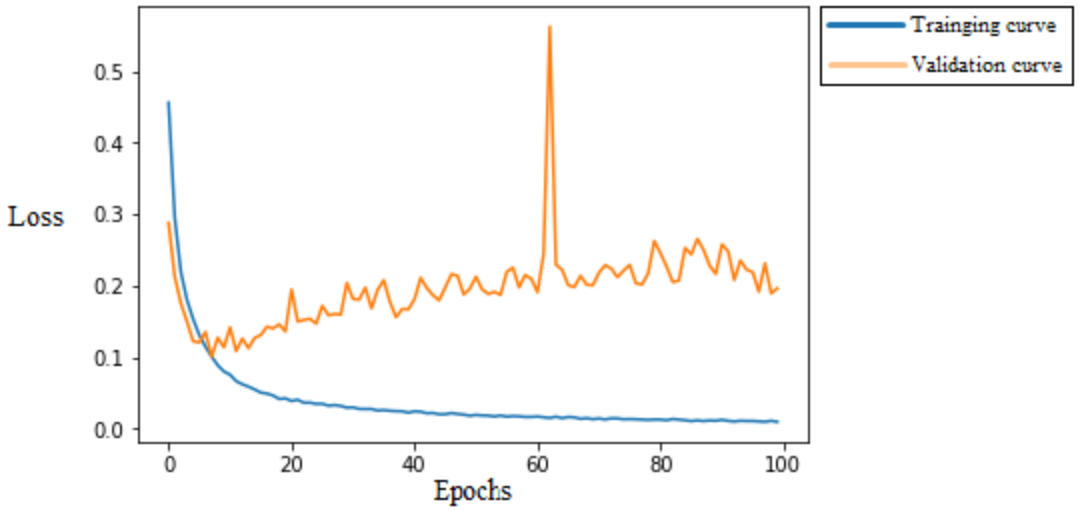
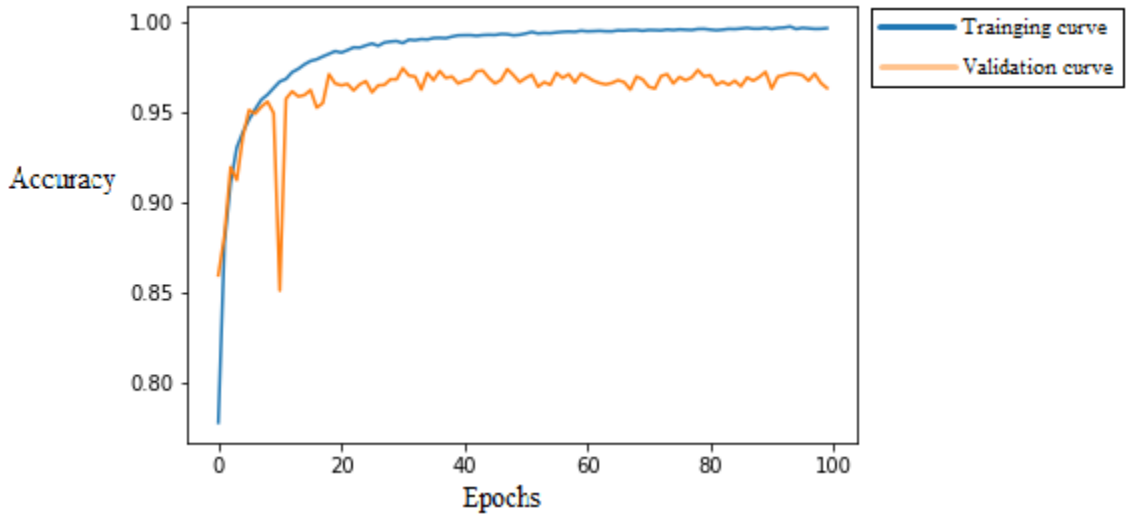


Figure 4.5. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist, thigh, wrist, chest, and head

a) Accuracy plot



b) Loss plot

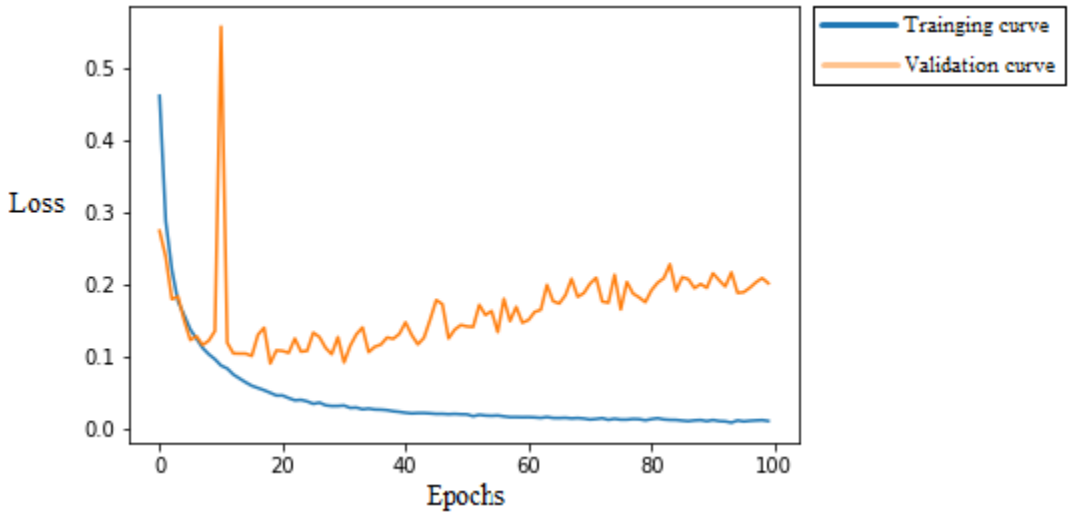


Figure 4.6. Accuracy and loss plot during training process, when filter = 16, window_size = 10, all six locations

When comparing Figures 4.3-4.6 to Figures 4.1 and 4.2, it can be seen that when more locations were used, the accuracy plot and loss plot converged faster, but the gaps between the training curve and validation curve became bigger. It can be concluded that when more locations of input were used, the overfitting effect became more serious, and the accuracy of the model became worse.

Considering the possibility that 2D convolution network cannot fully comprehend 3D data, 3D convolution neural network was tested on 2-location input and 3-location input. The

performances of both model are shown in Table 4.7. The training and validation curves for 2-location input is shown in Figure 4.7.

As can be seen in Table 4.7, 3D model has an accuracy of 95.42%, which is 0.57% higher than the 2D model, but still lower than single location model; also, according to the loss plot shown in Figure 4.7, big gap between training curve and validation curve can be observed. The 3D model cannot effectively solve the problem caused by overfitting.

As a result, with the CNN structure used in this work, adding data from more locations does not help increase the performance of the model. The possible reason for that is, for the given CNN structure, most of the additional information added by sensors other than waist sensor are not useful information that can help the model do better recognition. Thus, CNN tried to fit these "useless" information and leads to overfitting, which cause the performance of the model to decrease.

Table 4.7 Comparison between 2D CNN and 3D CNN for 2-location input.

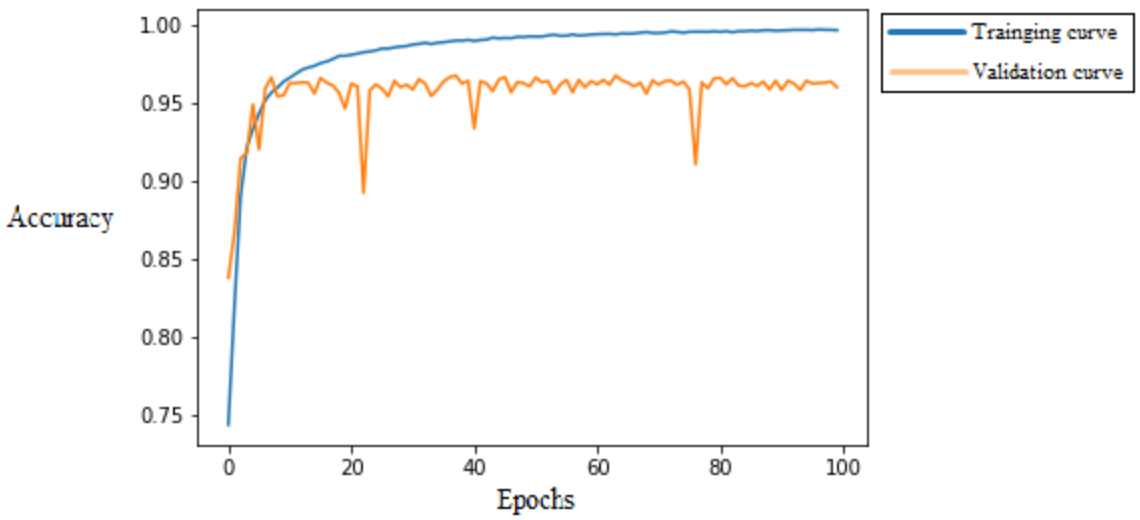
a. 16-filter, 20 window size, 2D model

16filter_20window_2sensor		
	TRUE	FALSE
Fall	3255	115
non_fall	4286	294
Accuracy	Sensitivity	Specificity
94.86%	96.59%	93.58%

b. 16-filter, 20 window size, 3D model

16filter_20window_2sensor_3d		
	TRUE	FALSE
Fall	3199	171
non_fall	4387	193
Accuracy	Sensitivity	Specificity
95.42%	94.93%	95.79%

a) Accuracy plot



b) Loss plot

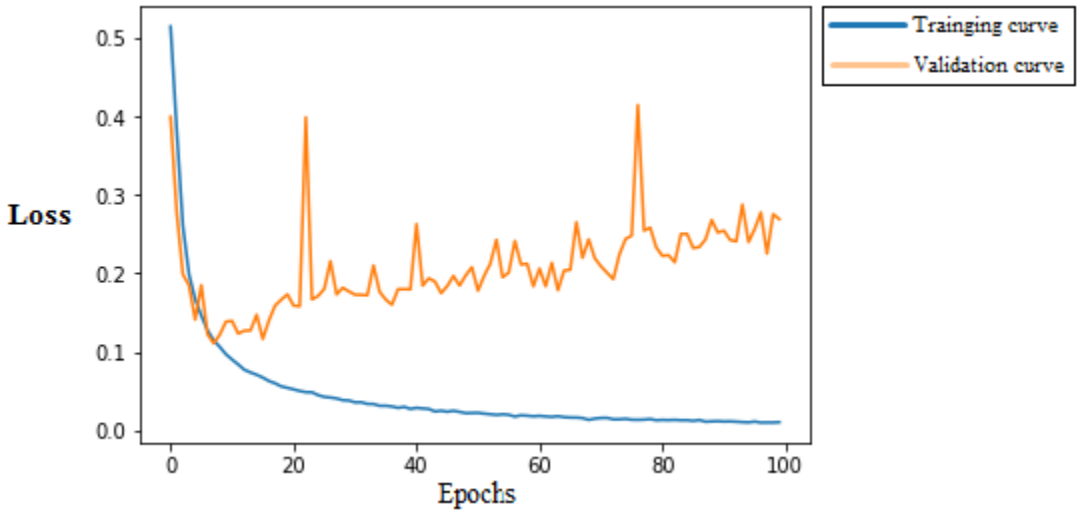


Figure 4.7. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist and thigh, using 3D convolution

4.1.1.2 Comparison between different locations for single location input

From the discussion in 4.1.1.1, single location input data was favored to be the best choice. It still required to know which location is the best among the six locations (ankle, chest, head, waist, wrist, and thigh).

Each of the six locations was used individually as the input to a 16-filter, 10-window-size CNN. The result is shown in Table 4.8.

Table 4.8 Comparison between different locations when single location input is used.

a. Waist

16filter_10window_waist		
	TRUE	FALSE
Fall	6901	219
non_fall	8856	556
Accuracy	Sensitivity	Specificity
95.31%	96.92%	94.09%

b. Ankle

16filter_10window_ankle		
	TRUE	FALSE
Fall	6457	663
non_fall	8288	1124
Accuracy	Sensitivity	Specificity
89.19%	90.69%	88.06%

c. Chest

16filter_10window_chest		
	TRUE	FALSE
Fall	6640	480
non_fall	8612	800
Accuracy	Sensitivity	Specificity
92.26%	93.26%	91.50%

d. Head

16filter_10window_head		
	TRUE	FALSE
Fall	5275	1845
non_fall	8464	948
Accuracy	Sensitivity	Specificity
83.11%	74.09%	89.93%

e. Thigh

16filter_10window_thigh		
	TRUE	FALSE
Fall	6795	325
non_fall	8582	830
Accuracy	Sensitivity	Specificity
93.01%	95.44%	91.18%

Table 4.8 continued

f. Wrist

16filter_10window_wrist		
	TRUE	FALSE
Fall	6600	520
non_fall	8355	1057
Accuracy	Sensitivity	Specificity
90.46%	92.70%	88.77%

As can be seen in Table 4.8 (f), the model achieved 95.3% accuracy when waist sensor was used. This is the highest accuracy among the six locations. So it can be concluded that mounting the sensor on waist is the best choice.

4.1.2 Comparison between different window sizes

Size of the window reflect the amount of information on time domain that is contained within each data sample. When the window size is bigger, it should be easier for the model to identify the type of activity.

Meanwhile, bigger window size means the model requires more data on time domain to make a decision. That indicates that the model has slower reflection speed when predicting fall events.

In this work, window size of 10 (0.4s/window) and window size of 20 (0.8s/window) were tested. The result are shown in Tables 4.1 and 4.2.

Because window size of 20 can already yield an accuracy up to almost 96.7%, larger window size were not tested in this work. Also, the CNN structure used in this work utilize (3,3) kernel to extract features, and max-pooling layers were used to reduce the size of the data by half, so window size smaller than 10 were not tested in this work either.

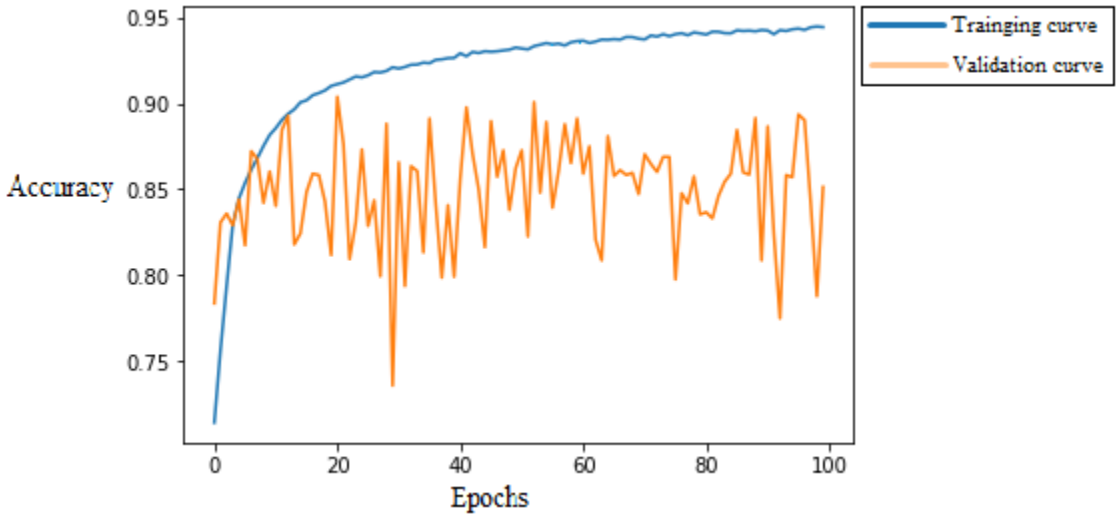
The highest accuracy for 10 window size input obtained by 32-filter model was 96.26% (Table 4.1(c)). The highest accuracy for 20 window size input obtained by 64-filter model, was 96.65% (Table 4.2(d)). Both models showed comparable accuracies. However, 10 window size is preferable because it provides a response time of 0.4 second as compared to the 0.8 second for 20 window size.

4.1.3 Comparison between different number of filters

For a given CNN structure, the number of filters used in the convolution layers represent the complexity of the model. More complex model has higher capability to extract features and do classification to high dimension data, but it also has higher chance of overfitting and requires larger computation power.

In this work, 8-filter model, 16-filter model, 32-filter model and 64-filter model were tested to find out the most suitable model for the given problem. As can be seen in Table 4.1, 8-filter models have the lowest accuracy among the four types of models, more than 1% less than the second worst model (16-filter model). The reason can be found in Figure 4.8 (a). This figure shows the accuracy plot for 8-filter model with window size of 10. In this figure, the training accuracy can barely reach 95% at the end of the training process. Compared to the other three models, this training accuracy is very low, which means that 8-filter model is too simple to extract enough features and do proper classification.

a) Accuracy plot



b) Loss plot

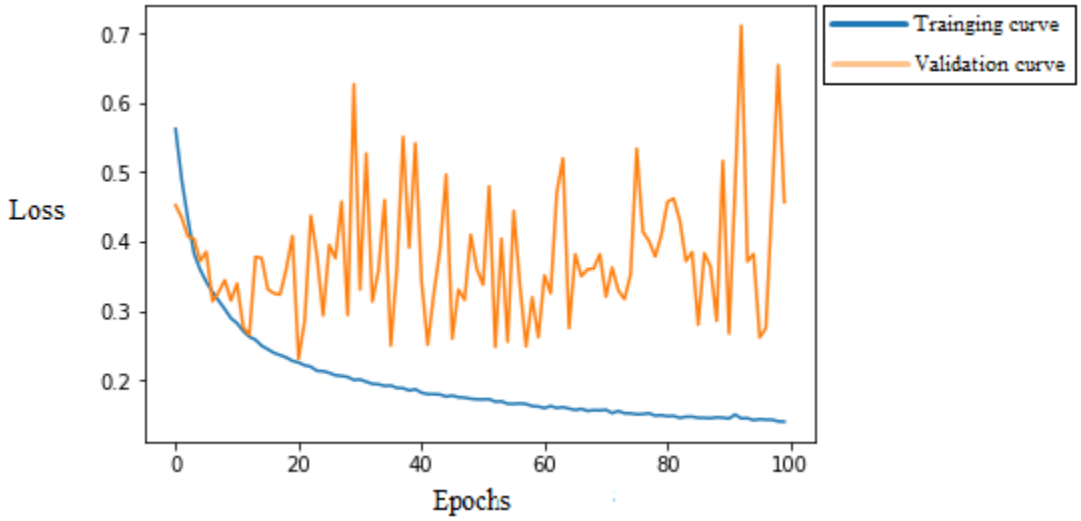
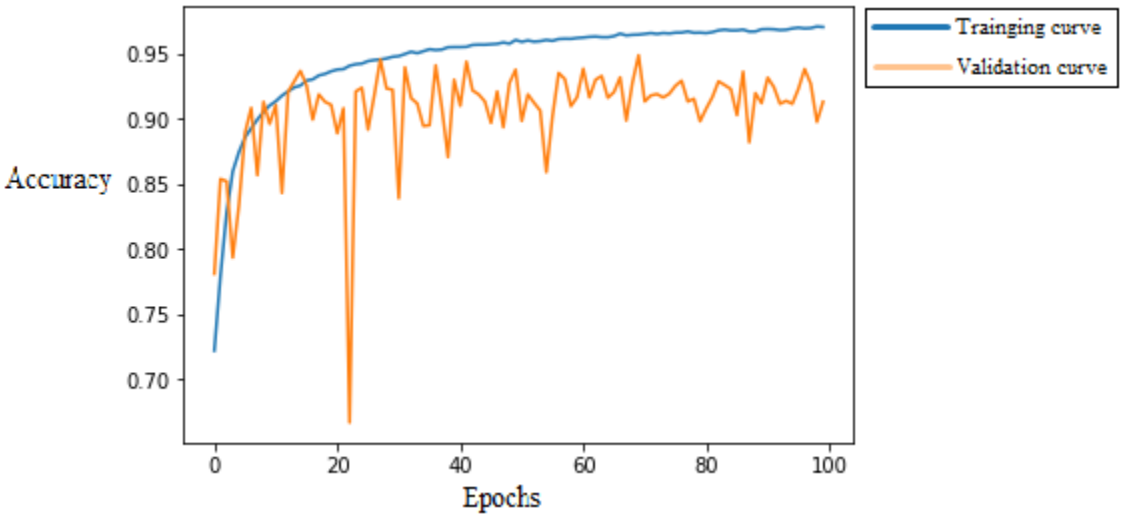


Figure 4.8. Accuracy and loss plot during training process, when filter = 8, window_size = 10, waist sensor only.

Among the other three models, 64-fitler model has the highest computational complexity, but the accuracy of 64-filter model is even lower than 32-filter model, only 0.5% higher than 16-filter model. The reason can be seen in Figure 4.7. There is a big gap between training curves and validation curves in both accuracy plot and loss plot, which means that there is a serious overfitting effect for 64-filter model. This overfitting effect is most possibly caused by the overly complexed structure of 64-filter model, which means that the complexity of 64-filter model is excessive for this given dataset.

For the 16-filter and 32-filter model, the training and validation curves in Figures 4.8 and 4.9 show that none of them have serious overfitting. The 32-filter model have better performance than 16-filter model, but the increase is only about 0.7%, which can be caused by random factors. Further comparison between 16-filter model and 32-filter model was made using 10-fold cross validation to eliminate the possible effect of random factors. The result is shown in Table 4.9.

a) Accuracy plot



b) Loss plot

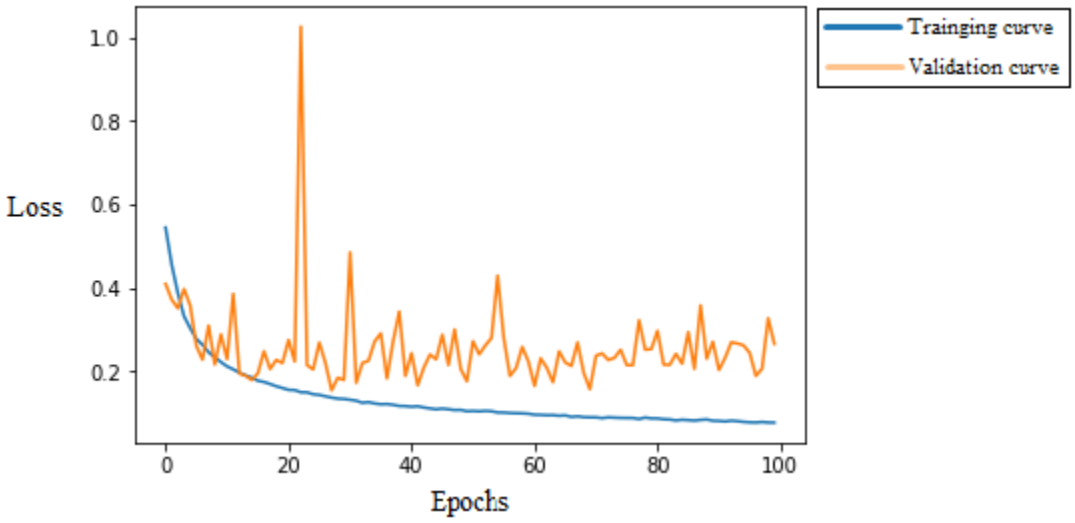
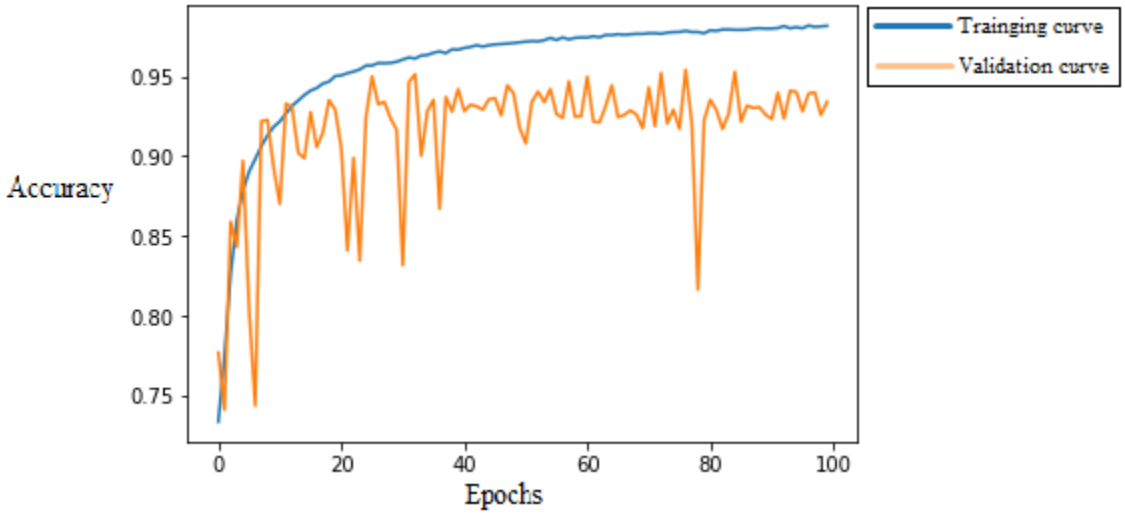


Figure 4.9. Accuracy and loss plot during training process, when filter = 16, window_size = 10, waist sensor only. (Same as figure 4.1)

a) Accuracy plot



b) Loss plot

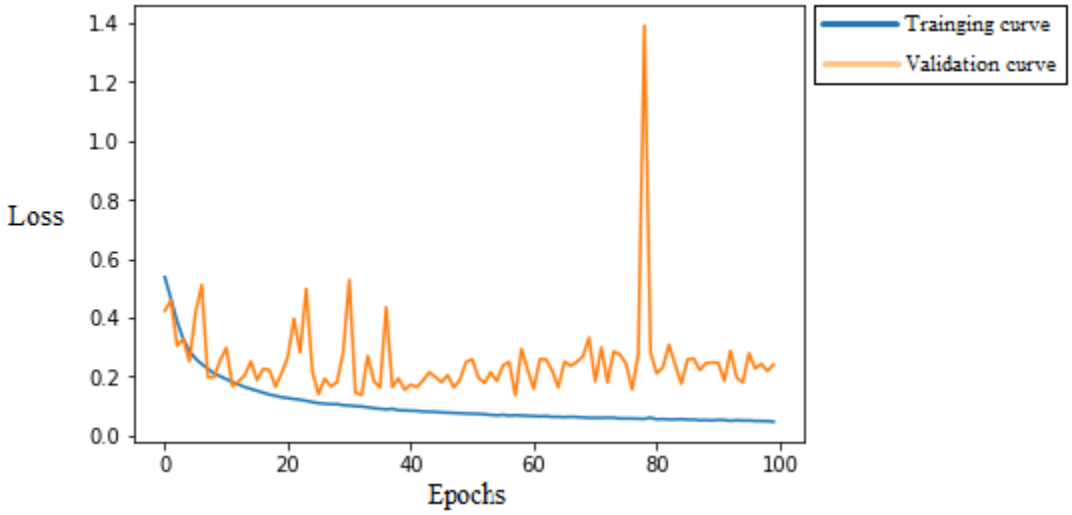


Figure 4.10. Accuracy and loss plot during training process, when filter = 32, window_size = 10, waist sensor only.

Table 4.9 Comparison between 16-filter model and 32-filter model using 10-fold cross validation

a. 16-filter model

10-fold cross validation (win_size=10, 16 filter, learning rate = 0.05)		
	Loss	Accuracy
1	0.07627	96.99%
2	0.06422	97.43%
3	0.06783	97.33%
4	0.06582	97.31%
5	0.0715	97.24%
6	0.06884	97.16%
7	0.07172	97.29%
8	0.09037	96.55%
9	0.06567	97.35%
10	0.06291	97.43%
Average accuracy	97.21%	
Epoch	100	

32-fitler model

10-fold cross validation (win_size=10, 32 filter, learning rate = 0.05)		
	Loss	Accuracy
1	0.0511	98.17%
2	0.05037	98.13%
3	0.06173	97.52%
4	0.04516	98.38%
5	0.04471	98.34%
6	0.05884	97.73%
7	0.05224	98.13%
8	0.05354	97.88%
9	0.04637	98.21%
10	0.04182	98.42%
Average accuracy	98.09%	
Epoch	100	

As can be seen in Table 4.9, 32-filter model has an average accuracy of 98.09%, about 0.8% higher than the 16-filter model. Yet considering the fact that 32-filter model has a higher computational complexity than 16-filter model, 16-filter model is a better choice for this given dataset.

4.1.4 Test on fall prediction capability

The result shown in Table 4.1 shows the performance of the models on recognizing fall and non-fall activity based on data starting from the beginning of the activity to the end of the activity. Yet it is possible that the models perform better on data in the middle and at the end of an activity, but perform not as well on data at the beginning of the data. In that case, the models do not have the capability of predicting fall activity before the fall happens.

In order to test the models' performance on predicting fall, the first 1second of data (4 windows, window size = 10) were extracted from each data file. These data represent the start of activities. For fall activities, most these data represent the data before the fall events happen. These data were used as input to 16-filter model. The result is shown in Table 4.10.

Table 4.10 Test result on performance of fall prediction

16filter_10window_1sensor		
	TRUE	FALSE
Fall	929	79
non_fall	1490	30
Accuracy	Sensitivity	Specificity
95.69%	92.16%	98.03%

As can be seen in Table 4.10, there is no obvious drop on accuracy as compared to the accuracy in Table 4.1(b), which means that the model is able to predict falling event before it happens.

4.2 Recurrent Neural Network

LSTM with convolution layers model and LSTM with convolution transformation model were tested in this work. In this work, waist location only is the only sensor configuration tested for RNN. The results are shown in tables 4.11 and 4.12. Training and validation curves are shown in figures 4.11 and 4.12. (LSTM with convolution layers model will be referred as LSTM1, and LSTM with convolution transformation will be referred as LSTM2 in later discussion)

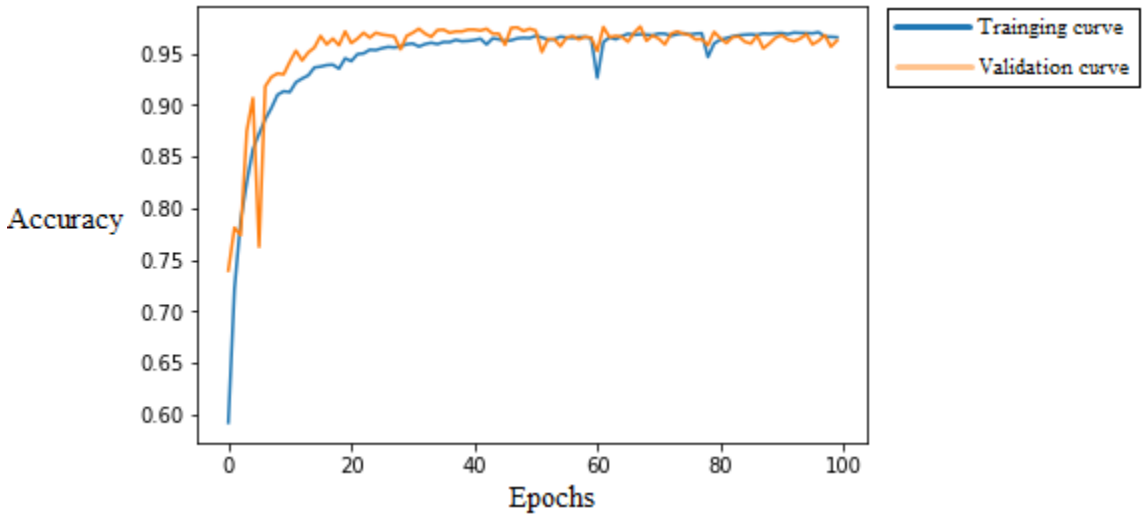
Table 4.11 Test result on LSTM with Convolution layer model (LSTM1)

LSTM with convolution layers(LSTM1 model)				
		Predicted Label		
		Fall	Pre/postcondition	Non-fall
True Label	Fall	3377	100	5
	Pre/postcondition	83	2854	15
	Non-fall	24	32	2990
	In-class accuracy	96.93%	95.58%	99.34%
	Overall accuracy	97.24%		

Table 4.12 Test result on LSTM with convolution transformation (LSTM2)

LSTM with convolution transformation (LSTM2 model)				
		Predicted Label		
		Fall	Pre/postcondition	Non-fall
True Label	Fall	3226	221	35
	Pre/postcondition	35	2869	48
	Non-fall	74	56	2916
	In-class accuracy	96.73%	91.20%	97.23%
	Overall accuracy	95.02%		

a) Accuracy plot



b) Loss plot

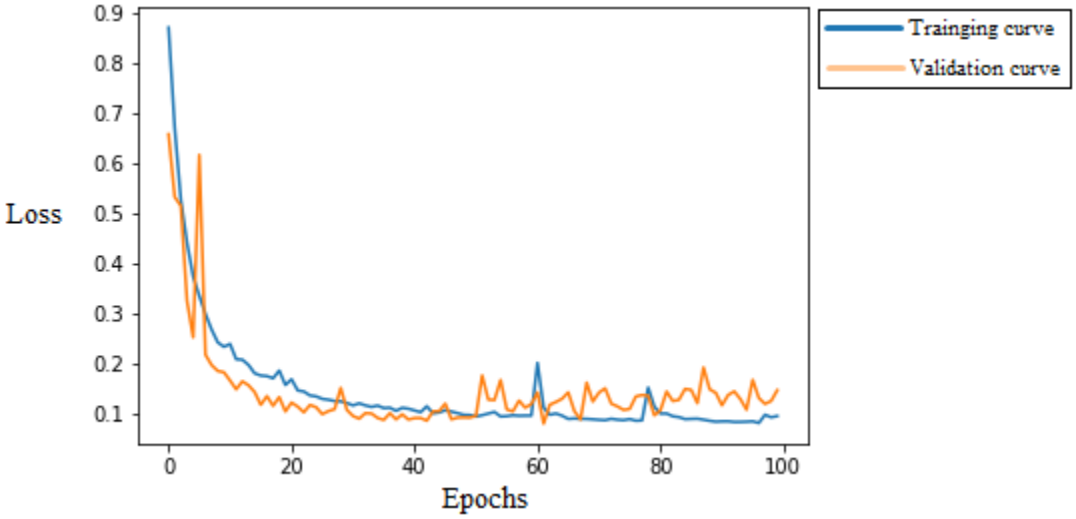
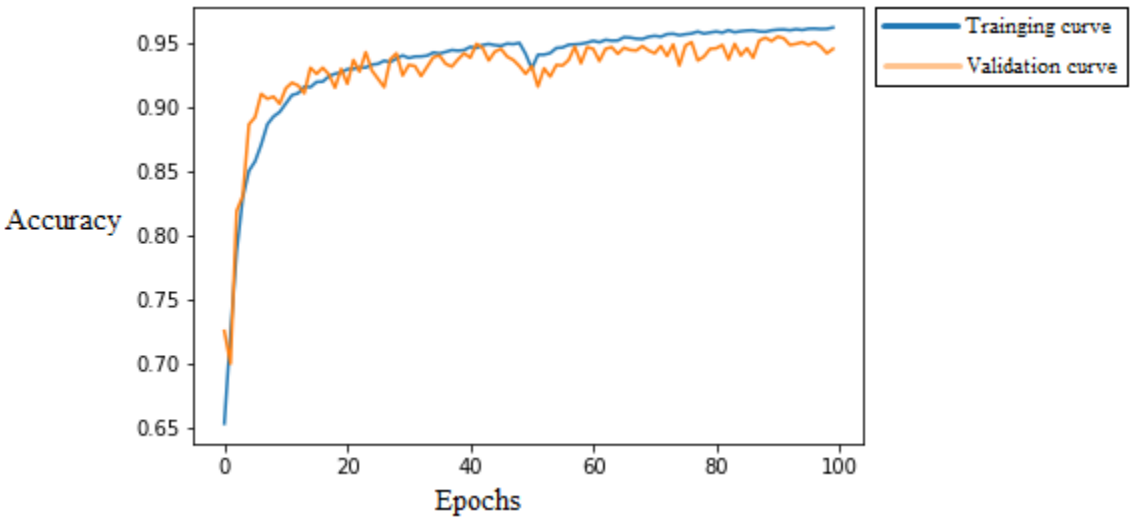


Figure 4.11 Accuracy and loss plot during training process for LSTM with convolution layers (LSTM1)

a) Accuracy plot



b) Loss plot

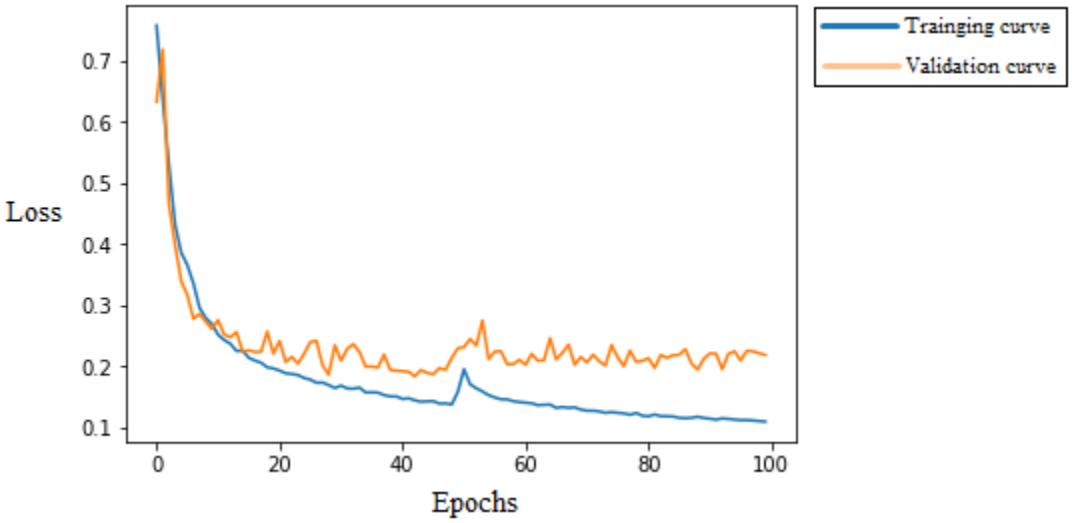


Figure 4.12 Accuracy and loss plot during training process for LSTM with convolution transformation (LSTM2)

As shown in table 4.11 and 4.12, LSTM1 model had an accuracy of 97.24%; LSTM2 model had an accuracy of 95.02%. LSTM1 provided very similar performance to that by LSTM2 model (LSTM1 is 2% higher). The difference of performance between the models can be more clearly observed from training and validation curves in figure 4.11 and 4.12. Training curves of both models converged to an accuracy of approximately 96% (figure 4.11a and 4.12a) and a loss of 0.1 (figure 4.11b and 4.12b). Comparison of training and validation curves of LSTM1 and LSTM2 model (figure 4.11 and 4.12) shows a noticeable gap between the training and validation

curves at later stage of epochs (>20) for LSTM2 model as compared to that in LSTM1 model. This indicates that LTM1 model has much less overfitting effect as compared to LSTM2.

It was observed from in-class accuracies (in table 4.11 and 4.12) that for both models, non-fall class has the highest accuracy (99.34% for LSTM1 model and 97.23% for LSTM2 model), and pre/postcondition class has the lowest accuracy (95.58% for LSTM1 model and 91.2% for LSTM2 model). The low accuracy for pre/postcondition class is largely caused by misclassifying pre/postcondition into fall activity. In LSTM1 model, 100 misclassification cases out of 132 were found in "fall activity" class. In LSTM2 model, 221 misclassification cases out of 277 were found in "fall activity" class. This means the transition point between precondition and fall activity is difficult for RNN to identify. Also, this result might be caused by any error in identifying the boundary between precondition and actual activity of the preprocessing algorithm. However, the RNN in this work tends to classify pre-condition into fall activity. Thus in real world application, the RNN would detect fall activities before it happens.

Compared to CNN model, the accuracy for LSTM1 model is very comparable to the highest accuracy of CNN (approximately 0.7% higher). Although this difference is very small, LSTM1 model achieved its accuracy using window size of 10 as input, while the highest accuracy of CNN was provided when input window size was 20. The result follows an expected pattern. Because for CNN, when the input window size was smaller, the time domain information contained in each window decreased. As CNN doesn't have the capability to find correlations between each window, the performance for CNN is lower when the window size is smaller. However, RNN has the capability to "remember" the information from the previous window because of the hidden unit structure. As a result, the performance of RNN doesn't get affected much by the decrease of window size. For conclusion, RNN (as compared to CNN) can provide high classification accuracy while increasing the speed of response of the system.

4.3 Discussion

Based on the result from the experiments, it can be concluded that 16-filter CNN model, when input is from waist only, and when input window size is 10, is the best choice for this dataset. This model set up has an accuracy of 95.31% on testing set. With the same sensor configuration and input setup, LSTM with convolution layers model achieves an accuracy of 97.24%.

In the work done by Ozdemir & Barshan (2014), over 99% accuracy was achieved by using machine learning algorithms including support vector machine, k-nearest-neighbor and etc. In their work, a time index A_T was found for each trial of activity. This index is the peak of total acceleration of waist location (total acceleration was defined in Chapter 3.2). Data of 2 seconds before and after A_T (25Hz*2second*2+1 = 101 data samples) were used as input, and all the other data samples were ignored. This input window was applied to all the nine sensor readings and all six locations. For each input window, there are six 101*9 arrays. The following features were extracted for each input window: minimum, maximum, mean, variance, skewness, kurtosis, first 11 values of autocorrelation sequence, and first five peaks of discrete Fourier transform. There are 234 feature values extracted from each location, which means 234*6 = 1404 feature values were extracted from each trial of activity. These 1040 feature values were put into principle component analysis, and 30 new values were generated. Each trial of activity has a feature vector that contains 30 values in it, and 2520 activity trials in total (1400 falls, 1120 non-falls) were used their work. These 2520 feature vectors were used as input to machine learning model. The accuracy was obtained by 10-fold cross validation.

When comparing the models developed in this work to the models developed by Ozdemir and Barshan, it can be seen that Ozdemir and Barshan's mode has much higher accuracy. Yet based on their method of creating input data, it can be safely concluded that their model does not have the capability to predict fall activities.

Yet for both CNN and RNN models developed in this work, they can use small data windows (0.4s) as input, and make classification at an interval of 0.2 seconds for CNN/0.4 second for RNN. So with additional modification, both RNN and CNN models developed in this work have the potential to predict falling events.

4.4 Theoretical design of a complete fall-prediction system

The systems proposed in this work should be able to read values from inertia sensors, use neural network to process the data and recognize when the patient is about to fall, and then give alarm to the patient. Two designs are proposed in this work: one is based on cell phone and the other one is based on microprocessor.

4.4.1 Design based on microprocessor

Sensors used for data collection in this system would include: an accelerometer, a gyroscope, and a compass. According to the conclusion drawn from the previous part, the system has the best performance when the sensors are only mounted on the waist. Each sensor has three axis. Nine different readings can be collected from the sensors at each time step. These nine readings will be referred as one data sample later on. The data will be stored as floating number which occupies 32 bits (4 bytes). The data should be collected at a frequency of 25Hz. A data sample will be collected every 0.04s (40ms).

A microprocessor will be used to collect and process data. Data collection and data processing may be done in parallel within the microprocessor depending on the capability of the processor. According to the conclusion from the previous part, both CNN and RNN take data with window size of 10 as input. For CNN, each adjacent window has 50% overlap, which means that in each input window, there are five new data samples and five old data samples. These 10 data samples (one window) will be fed into CNN directly without any preprocessing. The system will process one set of input every 0.04*(10*50%) = 0.2 second. So the processor that run CNN should be able to finish one forward phase within 0.2 second. Only 10 data samples will be stored in microprocessor's memory. For each new data samples that comes in, the oldest data sample will be erased from memory. The processor that run CNN should be able to finish one forward phase within 0.2 second. For RNN, there is no overlapping between each adjacent window. So, an input window can be formed for every 10 new data samples. These 10 data samples (one window) can be fed into RNN directly without any preprocessing. The system will process one set of input every 0.04*10=0.4 second. So the processor that run RNN should be able to finish one forward phase within 0.4 second. Like CNN, 10 data samples will be stored in the memory. For each new data samples that comes in, the oldest data sample will be erased from the memory.

The output for both neural network is a binary output. The number 1 indicates that the patient is about to fall down, and 0 indicates that the patient is doing daily activities (pre/postcondition in RNN will be categorized into daily activity). When the output is 1, the microprocessor will activate the actuator through GPIO. The choice of actuator will be decided in future work. The microprocessor will use Ubuntu as operating system. The system will be stored in a micro-SD card. Both python IDE and Tensorflow package will be installed in the

Ubuntu system. The implementation of neural network will use Tensorflow package with Keras backend. Sensors, the microprocessor and the actuator will be powered by batteries. The design of the system is shown in Figure 4.13.

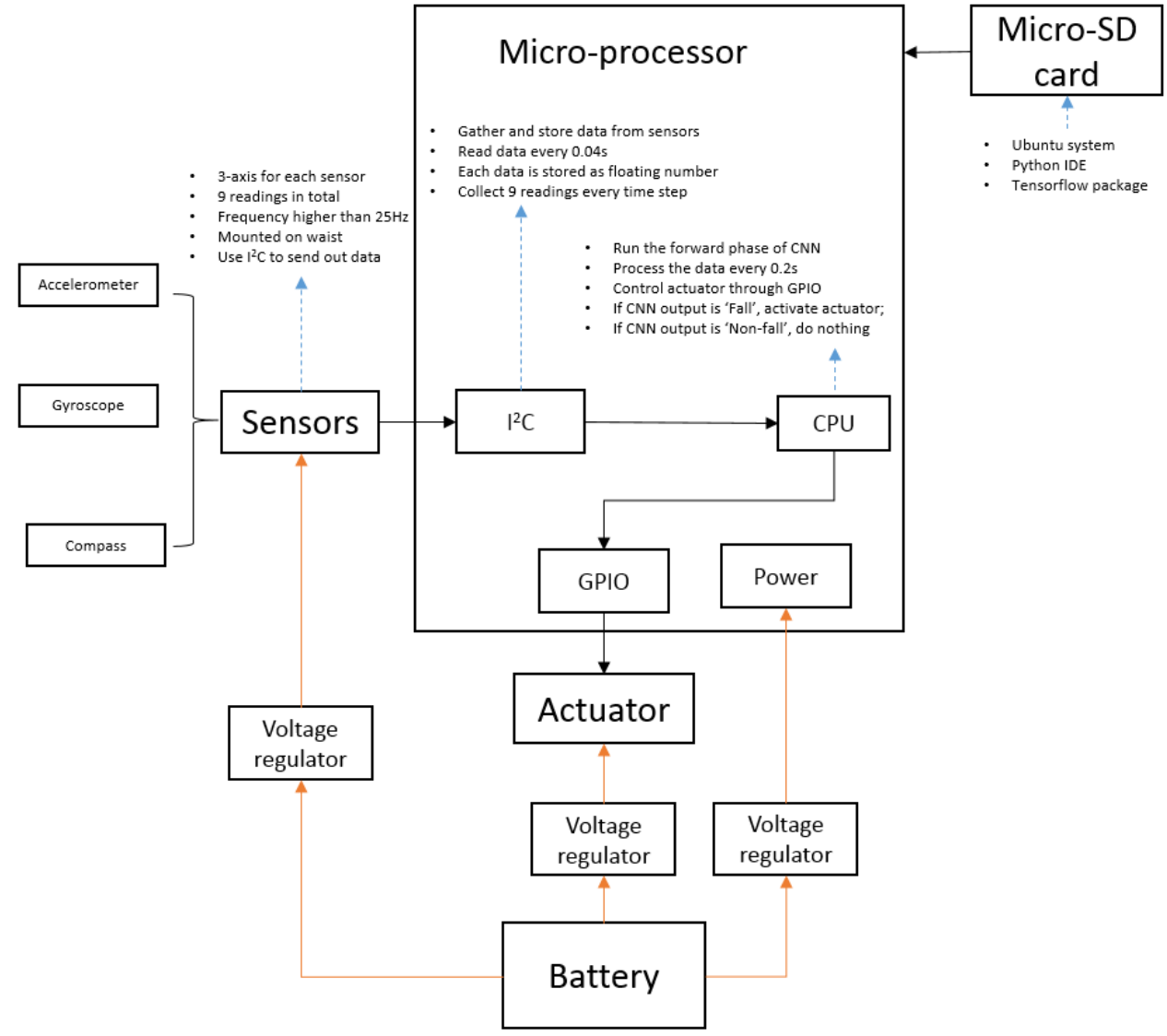


Figure 4.13 Design of system based on microprocessor

4.4.2 Design based on smart phone

The overall design for the system based on smart phone is very similar to the one based on microprocessor. Nowadays, most of the smart phones have in-built accelerometer (3-aixs),

gyroscope (3-axis), and compass (3-axis). However, for implementing model for any smart phones, the following factors needs to be considered:

1. The resolution and data collection rate of the smart phone in-build sensor needs to be the same or higher than the sensor mentioned in this study at all situation.
2. Chosen smart phone needs to have 3-axis accelerometer, 3-axis gyroscope and 3-axis compass.
3. The processor in the smart phone should be able to finish 1 forward phase of CNN within 0.2 second, or 1 forward phase of RNN within 0.4 second.

So the system will collect data from in-built accelerometer, gyroscope, and compass. Because each sensor has three axis, nine readings will be collected at each time step. The smart phone-embedded CPU/GPU will be used to process the data. The neural network model will be converted to cell phone app using Tensorflow Lite package. This app should read data from smart-phone in-built sensors, process the data using CNN/RNN model, and control an actuator to give alarm to the patient. The choice of actuator will be decided in future works. Figure 4.12 is a simple diagram of an Android cell phone-based pre-fall detection system.

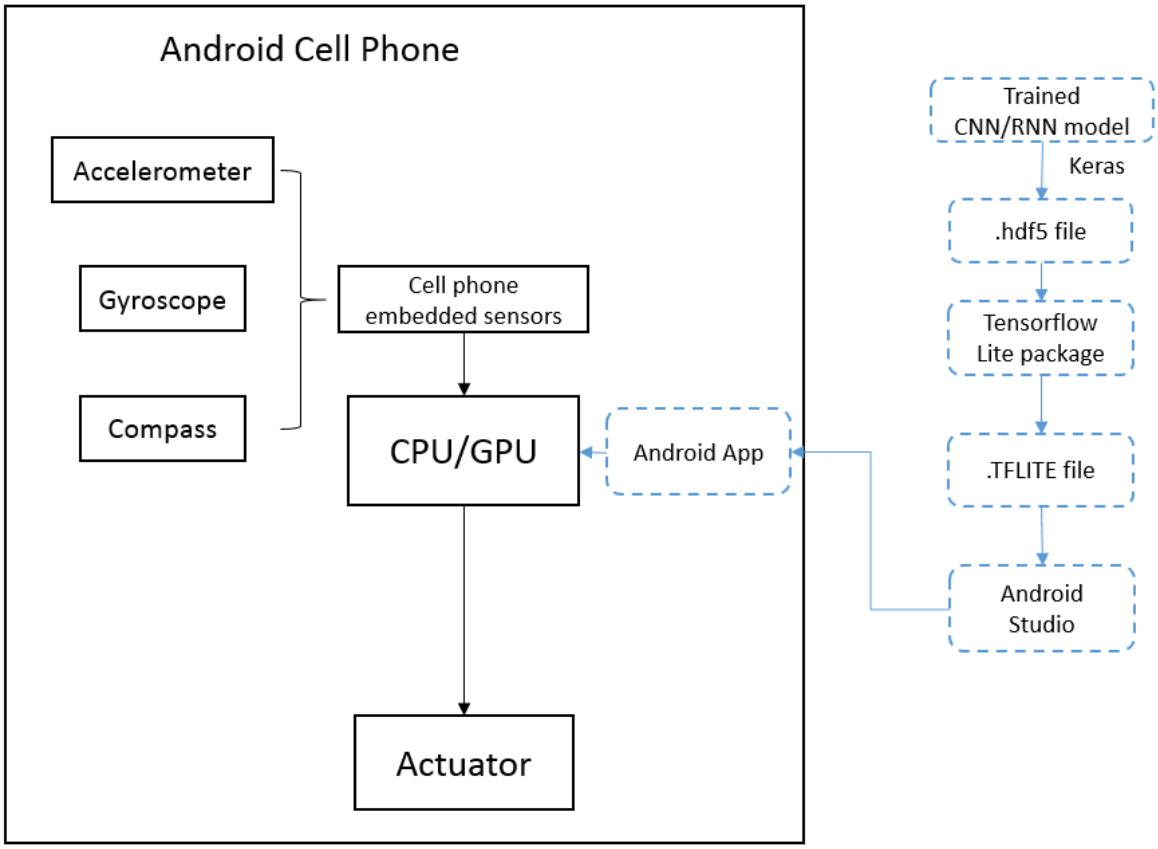


Figure 4.14 Design of system based on Android cell-phone

CHAPTER 5. SUMMARY, CONCLUSIONS, AND RECOMENDATIONS

5.1 Conclusion

In this work, a CNN model and a RNN mode were developed. According to the results of the experiments, both neural networks provided good performances regarding the "Simulated Falls and Daily Living Activities" dataset. All the CNN setups that were tested in this work could achieve an accuracy of over 85%. 16-filter model, with the input from waist location only, and with the window size of 10, proved to be the best setup for this given dataset. This model has an accuracy of 95.31% on testing set, and 97.21% based on 10-cross validation. Given that the data collection rate of the sensor is 25Hz, this mode can take 0.4 second input data window, and make prediction every 0.2 second. The input to the model is the data directly obtained from the sensor, and these data do not required any preprocessing. Two RNN models were tested in this work. The LSTM with convolution layers model (LSTM1 model) achieved an accuracy of 97.24% on testing set, using input from waist location and the window size of being 10. Given the data collection rate of the sensor is 25Hz, the model can take 0.4 second input data window and make prediction every 0.4 second. Based on the conclusion from the experiment result, two fall-prediction system framework designs were proposed.

5.2 Limitation

1. The data available did not have clear label for duration and transition of human activities. Therefore, certain assumptions were made during the development of preprocessing algorithm for labeling the activities.
2. The preprocessing algorithm used in this work used several heuristic equations. For other dataset, these equations might need modification.
3. This model is only tested for the dataset mentioned. The performance may vary on other dataset.
4. In this work, only one CNN structure (described in chapter 3.3.2.3) and two RNN structures (described in chapter 3.3.3.3 and 3.3.3.4) were tested because of the

limitation of computational power. There might be other neural network structures that might be more efficient for this given problem.

5.3 Future work

The following are recommended for future work:

1. The preprocessing algorithm could be further fine-tuned.
2. The developed models needs to be further validated on different datasets.
3. Other types of neural networks might be tested on this dataset.
4. Additional work can be made to prepare the neural network models for real world fall prediction application.
5. Additional investigations is needed to further evaluate RNN model on different parameters of this and other datasets.

APPENDIX A: FIGURES

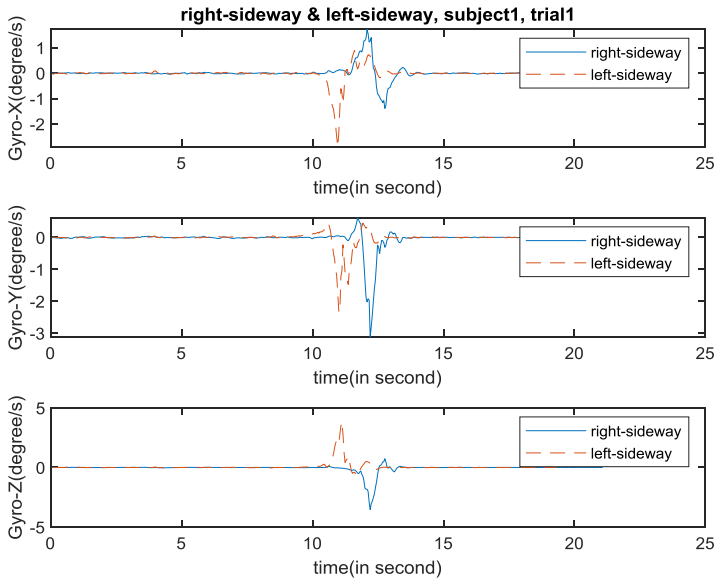


Figure A.1. Data sample of falling right-sideway and falling left-sideway, subject1, trial1, gyroscope. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

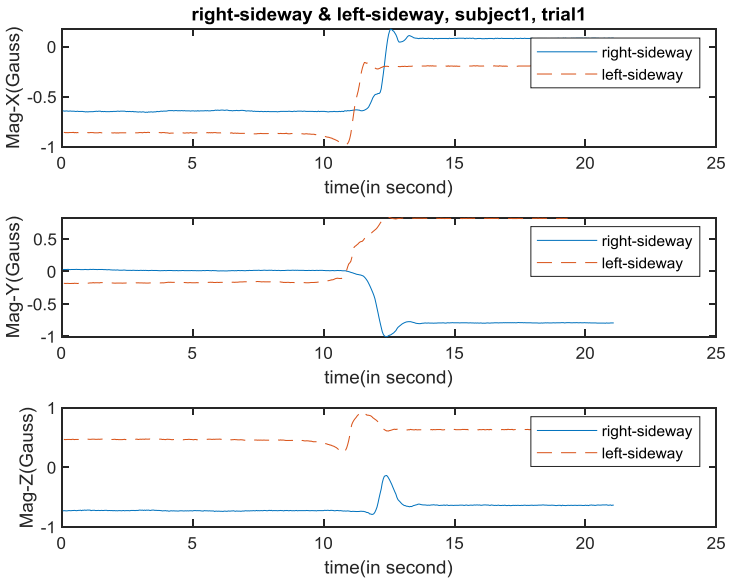


Figure A.2. Data sample of falling right-sideway and falling left-sideway, subject1, trial1, compass. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

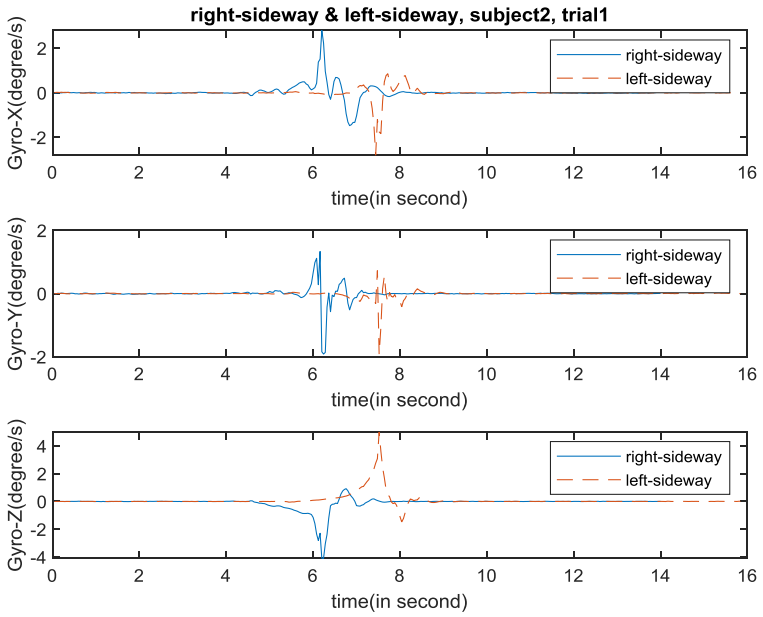


Figure A.3. Data sample of falling right-sideway and falling left-sideway, subject2, trial1, gyroscope. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

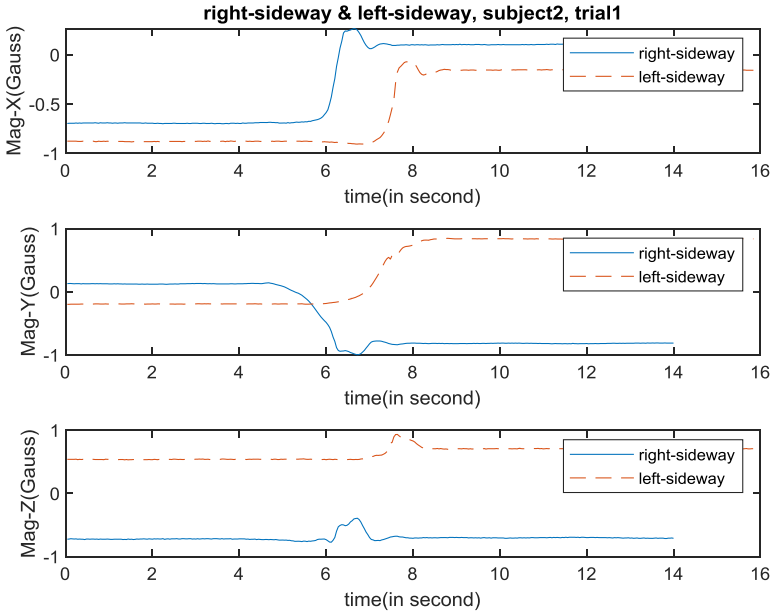


Figure A.4. Data sample of falling right-sideway and falling left-sideway, subject2, trial1, compass. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

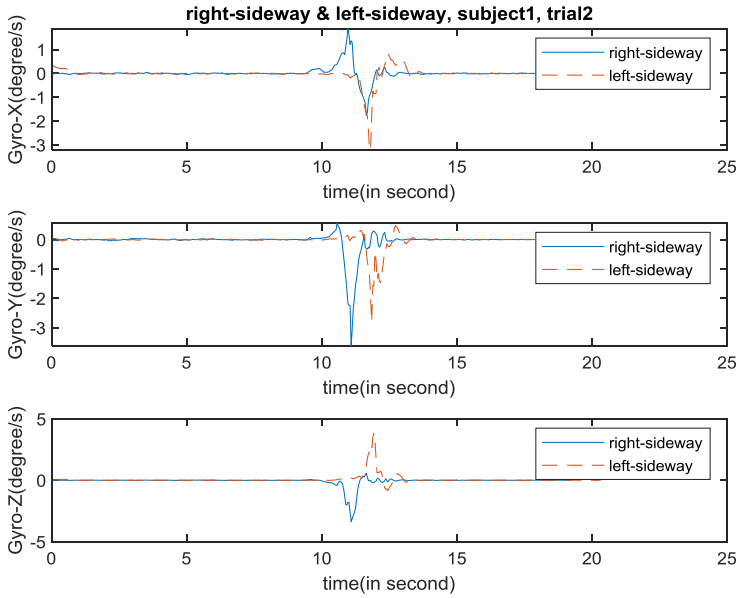


Figure A.5. Data sample of falling right-sideway and falling left-sideway, subject1, trial2, gyroscope. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

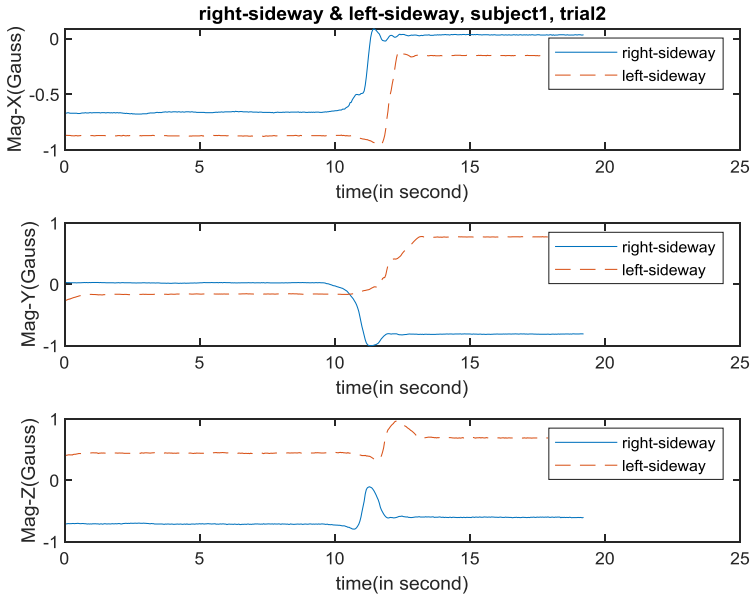


Figure A.6. Data sample of falling right-sideway and falling left-sideway, subject1, trial2, compass. (Fall activity 13 and fall activity 15. Descriptions are in page 22)

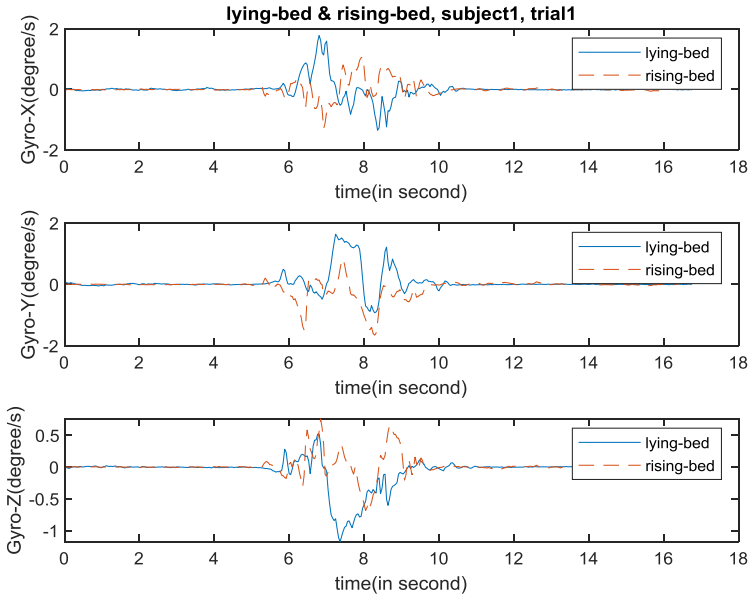


Figure A.7. Data sample of lying onto bed and rising from bed, subject1, trial1, gyroscope. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

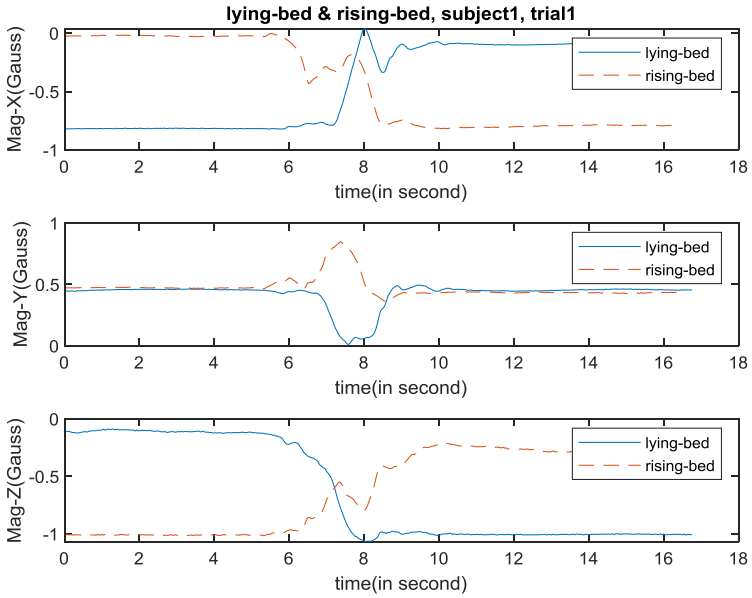


Figure A.8. Data sample of lying onto bed and rising from bed, subject1, trial1, compass. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

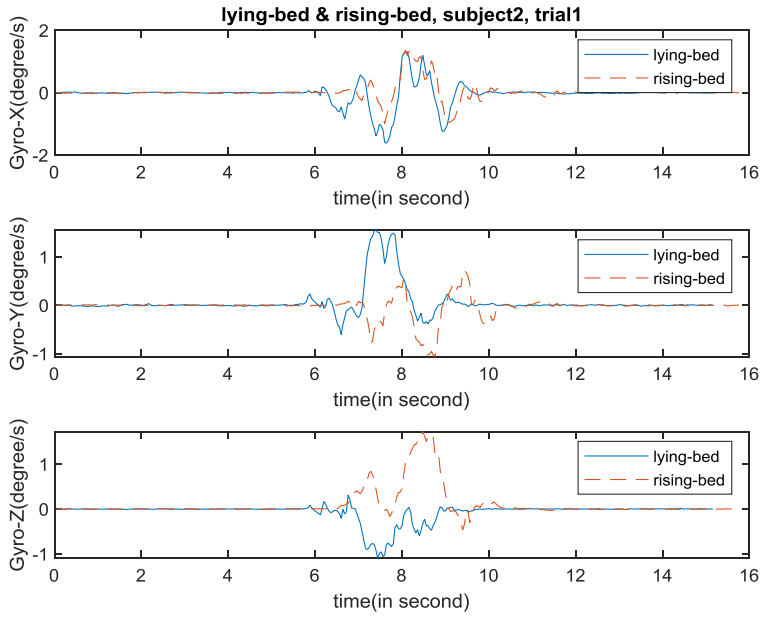


Figure A.9. Data sample of lying onto bed and rising from bed, subject2, trial1, gyroscope. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

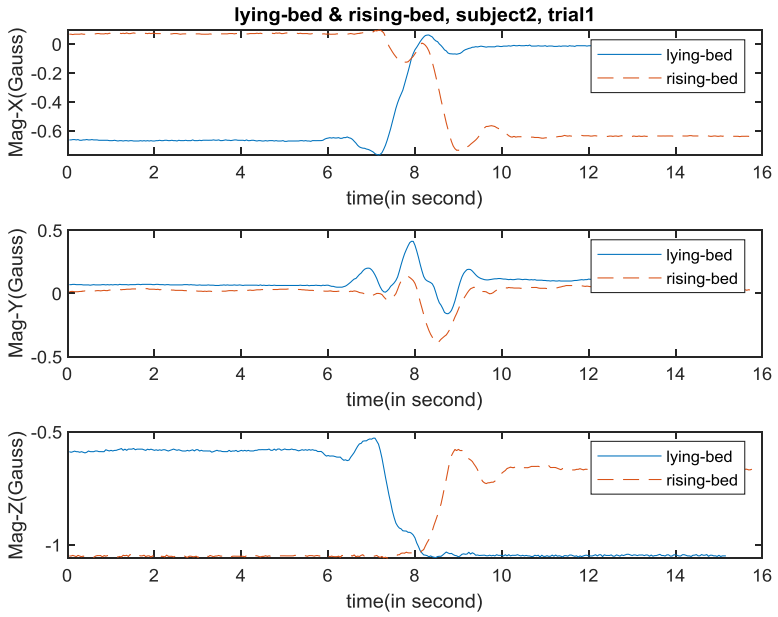


Figure A.10. Data sample of lying onto bed and rising from bed, subject2, trial1, compass. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

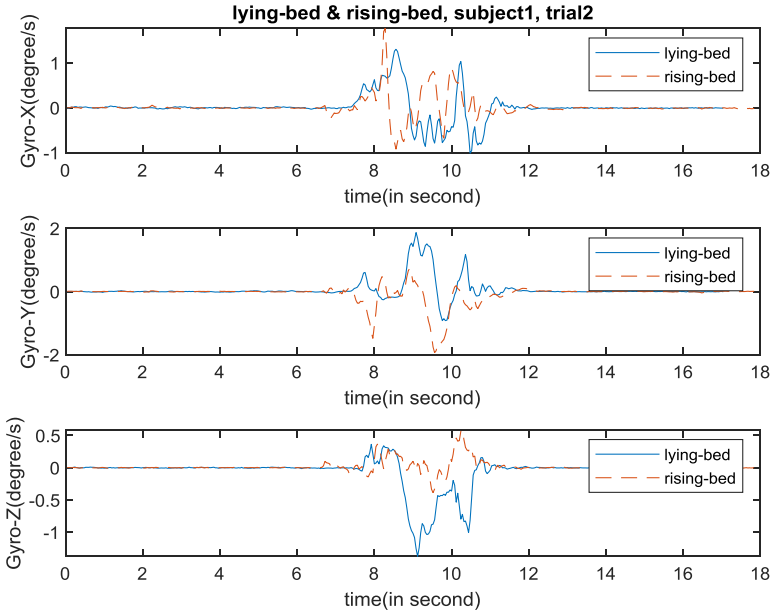


Figure A.11. Data sample of lying onto bed and rising from bed, subject1, trial2, gyroscope. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

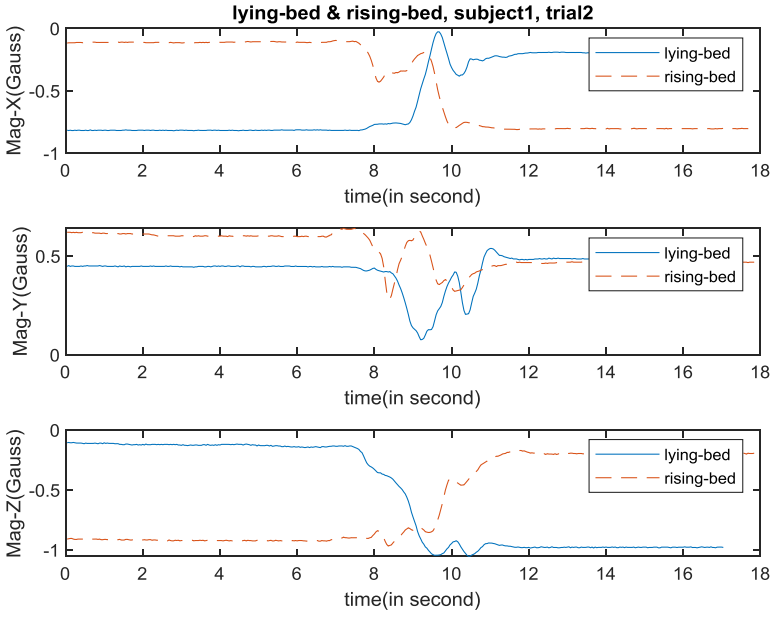


Figure A.12. Data sample of lying onto bed and rising from bed, subject1, trial2, compass. (Non-fall activity 15 and fall activity 16. Descriptions are in page 23)

APPENDIX B: SOURCE CODE

1. Program that read data input

```
"""
@author: yu492
File name: data_read
Input: Simulated falls and Daily living activities dataset
Output: two npy files that stores the data set and labels
Description: This program is to read data from text files,
and export the data into python work space(.npy file)
"""
import numpy as np
from tempfile import TemporaryFile
data_load = np.load('U:\Thesis\CNN_py\\data_test.npy')

import os,sys

data_x = []#Variable to store data
data_y = []#Variable to store labels
dir_root = 'U:\\Thesis\\Dataset\\Tests'
#Directory of root folder
files_sub = os.listdir(dir_root)
#List all folders under root directory(human subject folders)
for sub in range(len(files_sub)):
#Scan through all human subject folders
    dir_sub = dir_root+'\\'+files_sub[sub]+'\\'+'Testler Export'
            #Create directory for each human subjects
    files_act = os.listdir(dir_sub)
            #List all files under human subjects' folder(activity folders)
    print('reading data from subject', sub)
    for act in range(len(files_act)):
    #Scan through all activity folders
        dir_act = dir_sub + '\\' + files_act[act]
        #Create directory for activities
        files_trial = os.listdir(dir_act)
        #List all fiels under activity folders(trial folders)
        for trial in range(len(files_trial)):
        #Scan through all trial folders
            dir_file = dir_act + '\\' + files_trial[trial]
            #Create directory or trials
            files_data = os.listdir(dir_file)
            #List all files under trial folders(data files)
            for sensor in range(len(files_data)):
            #Scan through all data files within each trial folders
                dir_data = dir_file + '\\' + files_data[sensor]
                #Create directory for each data file
                data_file = []
                #Variable to store data from each data file
                with open(dir_data, 'r') as f:
                #Open text file
                    j = 0
                    #Number of lines
                    for line in f:
                        j = j+1
                        if j > 5:
                        #Get rid of the first 5 lines in each file(intro)
                            line = line.split('\t')
                            #split data based on "space"
```

```
                    data_line = []
                    #Data for one observation
                    for i in range(len(line)-1):
                    #Get rid of '\n' at the end of the line
                        if line[i] == '':
                            #Check problematic data
                            break
                        data_line.append(float(line[i]))
                    if len(data_line) == 23:
                    #Get rid of missing data
                    #(If there is no missing data, each observation has 23 characters)
                        data_file.append(data_line[9:18])
                        #Extract acc, gyro and compass(3*3)
            data_x.append(data_file)
            data_y.append([act, int(files_act[act][0:3]), dir_data])#Store the activity number as GT
np.save('U:\Thesis\CNN_py\\data_x',data_x)#Save data as data_x.npy
np.save('U:\Thesis\CNN_py\\data_y',data_y)#Save label as data_y.npy
```

2. Program that extract waist data

```
"""
Created on Tue Jan 29 14:30:27 2019

@author: Xiaoyu Yu
File name: Waist_extract
Input: Complete dataset
Output: Data from waist location
Description: This program is to extract data from waist sensor from the whole data set.
Also, the program excludes 5 activities from the data set.
The 5 activities include:
1. rolling out of bed
2. squating down
3. bending
4. coughing
5. sit-air
"""

import numpy as np
import matplotlib.pyplot as plt
data_x = np.load('data_x.npy') #Read in complete data set
data_y = np.load('data_y.npy') #Read in Ground Truth
data_waist_x = [] #Data to store waist data
data_waist_y = [] #Data to store waist label
count = 0

for i in range(len(data_x)): #Scan through the dataset
    a = data_y[i][2] #Directory of data file
    lenth = len(a)      #Number of data files
    sensor = int(a[lenth-7:lenth-4]) #Sensor index
    act = data_y[i][1] #Activity index
    if sensor == 535 and int(act) != 917 and int(act)!=813 and int(act)!= 810 and int(act)!= 804 and int(act)!= 805:
            #If waist sensor, if activity is not rolling out of bed, squating down, bending coughing sit-air
        data_waist_x.append(data_x[i])#Store data
        data_waist_y.append(data_y[i])#Store label


data_waist_x = np.delete(data_waist_x, 243,0) #Delete outlier
data_waist_y = np.delete(data_waist_y, 243,0) #Delete outlier
np.save('U:\Thesis\CNN_py\\data_waist_x',data_waist_x)
#Save waist data as data_waist_x.npy
np.save('U:\Thesis\CNN_py\\data_waist_y',data_waist_y)
#save waist labels as data_waist_y.npy
```

3. Program that calculate total acceleration

```
"""
Created on Tue Feb  5 15:49:47 2019
#Input: Waist data, waist label
#Output:Total acceleration
@author: Xiaoyu Yu
File name: Acc_waist_Total
This program is to calculate total acceleration for waist data
"""

import numpy as np
import matplotlib.pyplot as plt

data_x = np.load('data_waist_x.npy') #Load waist data
data_y = np.load('data_waist_y.npy') #Load waist label
Acc_T = [] #Variable to store total acceleration

for j in range(len(data_x)): #Scan through the data set
    Acc_file = [] #Variable to store total acceleration in each file
    data_test = data_x[j] #Variable to store data in each data file

    for i in range(len(data_test)): #Scan through the data file
        Acc_file.append((data_test[i][0]**2+data_test[i][1]**2+data_test[i][2]**2)**0.5)
                        #Calcualte total acceleration at each observation
    Acc_T.append(Acc_file)#Append the new total acceleration to the end of Acc_T
np.save('Acc_waist_T', Acc_T) #Save data
```

4. Program that preprocess raw data

```
"""
Created on Wed Jan  9 15:34:32 2019
Input: Data labels, total acceleration
Output: 2 indices for each data file
@author: Xiaoyu Yu
File name: data_preprocess
Description: This program is to do preprocessing before the data is fed into CNN/RNN
The preprocessing aims to get rid of the no-movement data for each data file.
This program can find out the boundary index for no-movement data and actual activity data.
The index values are recorded.
"""
import numpy as np
import matplotlib.pyplot as plt

win_size = 10 #Each window has 10 observations

data_y = np.load('U:\Thesis\CNN_py\\data_waist_y.npy') #Load data_waist_y.npy
Acc_T = np.load('U:\Thesis\CNN_py\\Acc_waist_T.npy')   #Load Acc_waist_T.npy
index_relabel = []                 #Used to store new label

var_start_T = []
var_end_T = []

'''
Scan through the whole dataset
Each data file is processed individually
Acc_T(total acceleration) is used to do pre-processing
Angular velocity(Gyro) and Magnitism(Mag) are not used in pre-processing
Acceleration x, y, z are transformed into total acceleration(in Acc_waist_Total.py)
'''
for j in range(len(Acc_T)):
    varT = []                    #Empty string to store variance for Total Acceleration
    meanT = []                     #Empty string to store mean for Total Acceleration
    AccT = Acc_T[j]                 #Total acceleration in 1 data file

    lenth = len(AccT)               #Lenth of the current data file

    '''Step2 & 3'''
    num_win = int(lenth/win_size)     #Calcualte the number of windows for each data file
    for i in range(num_win):
        varT.append(np.var(AccT[i*win_size:(i+1)*win_size]))    #Calculate variance string for Total Acc
        meanT.append(np.mean(AccT[i*win_size:(i+1)*win_size]))  #Calculate mean string for Total Acc

    #Let variance string run through average filter with size of (3,1)
    varT_avg = []
    for i in range(len(varT)):
        #Take care of special cases for the first and the last data point
        if i == 0:
            varT_avg.append((varT[i] + varT[i+1])/2)
        elif i == len(varT)-1:
            varT_avg.append((varT[i] + varT[i-1])/2)
        #Calcualte the average of itself and its two adjacent points
        else:
            varT_avg.append((varT[i-1] + varT[i] + varT[i+1])/3)
                        #Size of varT_avg is length/10

    '''Step4 '''
    max_index = varT_avg.index(max(varT_avg))
    #Find out the max variance in VarT_avg, related to fall/non-fall activity
```

```
'''Step5'''
min_index_start = varT.index(min(varT[0:max_index]))
#Find out the min variance in varT before the actual activity, related to precondition
min_index_end = varT.index(min(varT[max_index:len(varT)]))
#Find out the min variance in varT after the actual activity, related to postcondition

mean_start = meanT[min_index_start] #Mean of the min_start window (before the activity)
mean_end = meanT[min_index_end]     #Mean of the min_end window (after the acitivity)
sys_var_start = varT[min_index_start]   #Var of the min_start (before the acivity)
sys_var_end = varT[min_index_end]       #Var of the min_end window (after the activity)

'''Step 6'''
sys_noise_start = max(sys_var_start**0.5,0.01) #sys_nosie before activity
sys_noise_end = max(sys_var_end**0.5,0.01)     #sys_nosie after activity


'''
Consider the windows before min_index_start as no-movement
Consider the windows after min_index_end as no-movement
Scan from the min_start window to activity window, find the breaking point for actual activity and precondition
Scan from the min_end window to activity window, find the breaking point for actual activity and postcondition

'''

'''step7(before activity)'''
#Find out the break point between the actual activity and no-movement before the actual activity
count = 0
for i in range(win_size*(min_index_start+1), win_size*(max_index)):
    count = count + 1
    len_start = win_size*(max_index)-win_size*(min_index_start+1)
   #Lenth of the data between sys_var_start and var_max
    diff_list_start = []                                #List of break point before activity
    for k in range(5):                                  #Calcualte 1st order derivative, diff_list
        diff_list_start.append(abs(mean_start - AccT[i+k]))
    diff1_start = min(diff_list_start)                      #Min of diff_list
    diff2_start = np.mean(diff_list_start)                  #Mean of diff_list

    '''step8-10(before activity)'''
    #Compare diff1. diff2 to scaled sys_noise.
    #scaler is determined by the distance between current point and activity window
    #max_scaler for diff1 is 4, and max_scaler for diff2 is 10.
   #These values are empirical values and proves to be most effective
    if (diff1_start < (4-3*count/len_start)*sys_noise_start) and (diff2_start < (10-9*count/len_start)*sys_noise_start):
        #Condition not met
        mean_start = np.mean(AccT[0:i])       #update base acceleration
        sys_var_start = np.var(AccT[0:i])     #update system noise
        sys_noise_start = max(sys_var_start**0.5,0.01)
                    #Condition met, mark as start of actual activity
    else:
        index_start = i
        break

'''step7(before activity)'''
#Find out the break point between the actual activity and no-movement after the actual activity
count = 0
for i in range(len(AccT)-min_index_end*win_size, len(AccT)-max_index*win_size):
    count = count + 1
    len_end = len(AccT)-max_index*win_size - (len(AccT)-min_index_end*win_size)
   #Lenth of the data between sys_var_end and var_max
    diff_list_end = []                                      #List of break point before activity
    for k in range(5):                                      #Calcualte 1st order derivative, diff_list
        new_data = AccT[len(AccT)-i-k]
```

```
        diff_list_end.append(abs(mean_end - new_data))
    diff1_end = min(diff_list_end)                                  #Min of diff_list
    diff2_end = np.mean(diff_list_end)                              #Mean of diff_list

    '''step8-10(before activity)'''
    #Compare diff1. diff2 to scaled sys_noise.
    #scaler is determined by the distance between current point and activity window
    #max_scaler for diff1 is 4, and max_scaler for diff2 is 10.
    #These values are empirical values and proves to be most effective
    if (diff1_end < (4-3*count/len_end)*sys_noise_end) and (diff2_end < (10-9*count/len_end)*sys_noise_end):
    #Condition not met
        mean_end = np.mean(AccT[len(AccT)-i:len(AccT)])      #update base acceleration
        sys_var_end = np.var(AccT[len(AccT)-i:len(AccT)])    #update system noise
        sys_noise_end = max(sys_var_end**0.5,0.01)
        #Condition met, mark as end of actual activity
    else:
        index_end = len(AccT)-i
        break



var_start_T.append(varT[min_index_start])
var_end_T.append(varT[min_index_end])

index_relabel.append([index_start, index_end])
#np.save('index_relabel', index_relabel)
```

5. Program that manually modify outliers

```
"""
Created on Thu Mar 21 11:05:05 2019

@author: Xiaoyu Yu
File Name: outlier_mod
Input: Relabeling indices
Output: Modified relabeling indices
Description: This code is to make modification to outlier data samples.
The data_preprocessing can take care of most of the data samples except for 9.
This program is to manually modify the 9 outliers.
"""

import numpy as np
import matplotlib.pyplot as plt

index_relabel = np.load('index_relabel.npy') #Load relabeling index
Acc_T = np.load('Acc_waist_T.npy') #Load total acceleration(obtained from Acc_waist_total)

'''
The indices below are found by manually inspecting
the data file
'''
index_relabel[1205] = [160,355]#Manually define relabeling indices
index_relabel[1373] = [55,300] #Manually define relabeling indices
index_relabel[1586] = [155,448]#Manually define relabeling indices
index_relabel[1598] = [192,270]#Manually define relabeling indices
index_relabel[1652] = [105,600]#Manually define relabeling indices
index_relabel[1653] = [135,635]#Manually define relabeling indices
index_relabel[2315] = [145,390]#Manually define relabeling indices
index_relabel[2411] = [175,273]#Manually define relabeling indices
index_relabel[2660] = [215,680]#Manually define relabeling indices

np.save('index_relabel', index_relabel)#Save modified relabeling indices
```

6. Program that do data preparation for CNN model

```
"""
Created on Tue Feb 19 13:57:32 2019
Input: Waist data, waist labels
Output: Trainng data/labels, testing data/labels, validation data/labels
@author: Xiaoyu Yu
File name: Create_window_waist
Description: This program is to divide data into small windows,
and format the data in the way that it can be read by Keras CNN structure.
Also the program divide data into test set, validation set and training set.
"""

import numpy as np
import matplotlib.pyplot as plt

index_relabel = np.load("index_relabel.npy") #Load relabeled indices
data_x = np.load('data_waist_x.npy') #Load waist data
data_y = np.load('data_waist_y.npy') #Load waist labels
len_data = []  #length of data

overlap = 5   #number of overlab between adjacent winodw
win_size = 10 #window size
input_x = []
input_y = []
data_info = []
lenth = len(index_relabel) #Number of data files

train_x = [] #Training data
train_y = [] #Training labels
train_info = [] #Training data info
test_x = [] #Testing data
test_y = [] #Testing labels
test_info = [] #Testing data info
vali_x = [] #Validation data
vali_y = [] #Validation labels
vali_info = [] #Validation data info

'''
Round the indices to integer times of 10.
Because the windows size to be test are 10, 20 with 50% overlap
'''
for i in range(lenth):
    index_start = index_relabel[i][0] #Index of starting boudary
    index_end = index_relabel[i][1]   #Index of ending boundary
    index_start = int(index_start/10)*10 #Round down start index
    index_end = int(index_end/10+1)*10   #Round up end index


for i in range(lenth):
    len_data_mod = index_end - index_start #Number of data between two indices
    win_num = int(len_data_mod/overlap-1) #Calculate number of winodws
    data_relabel = data_x[i][index_start:index_end]
                #Extract actual activity data from each data file

    '''
    The following for loop is to label data with fall or non-fall
    and divide data into small windows
    '''
    for j in range(win_num): #Scan through dataset
        act = int(data_y[i][1])     #Activity index for one data file
        if act < 900:               #If fall activities
```

```
            input_x.append(data_relabel[j*overlap:j*overlap+win_size])
            #Divide data into windows
            input_y.append(0)
            #Label data as "0"
        elif act > 900:             #If non-fall activities
            input_x.append(data_relabel[j*overlap:j*overlap+win_size])
            #Divide data into windows
            input_y.append(1)
            #Label data as "0"
        data_info.append(data_y[i][1:3])

'''
The following for loop is to divide data into:
training, testing, and validation sets
'''
for i in range(len(input_x)):#Scan through the dataset
    sub = int(data_info[i][1][24:27]) #Extract human subject info
    if sub == 101 or sub == 102 or sub == 203 or sub == 204:
                #If the data is performed by these human subjects
        test_x.append(input_x[i])#Store the data into testing set
        test_y.append(input_y[i])#Store labels
        test_info.append(data_info[i])#Store other data infos

    elif sub == 103:
                #If the data is performed by this human subjects
        vali_x.append(input_x[i])#Store the data into validation set
        vali_y.append(input_y[i])#Store labels
        vali_info.append(data_info[i])#Store other data infos
    else:
                #The rest of the data are training data
        train_x.append(input_x[i])#Store the dat into training set
        train_y.append(input_y[i])#Store labels
        train_info.append(data_info[i])#Store other data infos

np.save('U:\Thesis\CNN_py\\train_x_waist', train_x)
#Store train data as train_x_waist.npy
np.save('U:\Thesis\CNN_py\\test_x_waist', test_x)
#Store test data as test_x_waist.npy
np.save('U:\Thesis\CNN_py\\train_y_waist', train_y)
#Store train label as train_y_waist.npy
np.save('U:\Thesis\CNN_py\\test_y_waist', test_y)
#Store train label as test_y_waist.npy
np.save('U:\Thesis\CNN_py\\train_info_waist', train_info)
#Store train label as train_info_waist.npy
np.save('U:\Thesis\CNN_py\\test_info_waist', test_info)
#Store train label as test_info_waist.npy
np.save('U:\Thesis\CNN_py\\vali_x_waist', vali_x)
#Store train label as vali_x_waist.npy
np.save('U:\Thesis\CNN_py\\vali_y_waist', vali_y)
#Store train label as vali_y_waist.npy
np.save('U:\Thesis\CNN_py\\vali_info_waist', vali_info)
#Store train label as vali_info_waist.npy
```

7. Program that build, train and save CNN model

```
"""
Created on Wed Feb 27 15:25:00 2019
Input: Training data, validation data, training labels, validation labels, testing data, testing labels
Output: Trained CNN models, testing accuracy
@author: Xiaoyu Yu
File Name: CNN_test
Description: This program is to train the CNN once, save the models, and test on test set.
"""
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, Conv1D, MaxPooling1D, Activation,
Dropout
from tensorflow.keras import optimizers
from tensorflow.keras.utils import plot_model
from tensorflow.keras.callbacks import ModelCheckpoint
from sklearn.model_selection import StratifiedKFold
import matplotlib.pyplot as plt


'''
Load data
'''
train_x = np.load('train_x.npy') #Load training data
train_y = np.load('train_y.npy') #Load training label
train_info = np.load('train_info.npy') #Load training info

test_x = np.load('test_x.npy') #Load testing data
test_y = np.load('test_y.npy') #Load testing label
test_info = np.load('test_info.npy') #Load testing info

vali_x = np.load('vali_x.npy') #Load validation data
vali_y = np.load('vali_y.npy') #Load validation label
vali_info = np.load('vali_info.npy') #Load validation info


train_x = train_x[:,:,:,np.newaxis]#Add a dimension for channel
train_y = tf.keras.utils.to_categorical(train_y, 2)#Convert into 1-hot vector
test_x = test_x[:,:,:,np.newaxis]#Add a dimension for channel
test_y = tf.keras.utils.to_categorical(test_y, 2)#Convert into 1-hot vector
vali_x = vali_x[:,:,:,np.newaxis]#Add a dimension for channel
vali_y = tf.keras.utils.to_categorical(vali_y, 2)#Convert into 1-hot vector

'''
Start of model
'''
model = tf.keras.Sequential() #Create sequential model

'''
Covolution group1
'''
model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv1',padding = 'same'))
#Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv2',padding = 'same'))
#Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
model.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
#Add max-pooling layer, pooling size (2,2), stride = 1, output size =input size
model.add(Dropout(0.25))
#Add dropout, dropout rate = 0.25
```

```
        '''
        Convolution group 2
        '''
        model.add(Conv2D(64, kernel_size = (3,3), activation = 'relu', name = 'Conv3',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
        model.add(Conv2D(64, kernel_size = (3,3), activation = 'relu', name = 'Conv4',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
        model.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
        #Add max-pooling layer, pooling size (2,2), stride = 1, output size =input size
        model.add(Dropout(0.25))
        #Add dropout, dropout rate = 0.25

        '''
        Fully connected layers
        '''
        model.add(Flatten())
        #Convert multi-dimension data into 1-D
        model.add(Dense(128))
        #Add Fully connected layer, 128 hidden nodes
        model.add(Activation('relu'))
        #Add Relu activation function
        model.add(Dropout(0.5))
        #Add dropout layer, dropout rate = 0.5
        model.add(Dense(2))
        #Add Fully connected layer, 2 hidden nodes
        model.add(Activation('softmax'))
        #Add softmax layer
        '''
        End of model
        '''
        sgd = optimizers.SGD(lr=0.005)
        #Define schocastic gradient descent as optimizer, learning rate = 0.05
        model.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])
        #Compile model, optimizer is SGD, loss function is categorical crossentropy, evaluation method: accuracy

        filepath="U:\Thesis\\CNN_py\\CNN model\\weights-improvement-{epoch:02d}.hdf5"
        #Directory to save model
        checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose = 0, save_best_only = True, save_weights_only =
True, mode='auto')
        #Save model when accuracy is best, save weight only
        callback_list = [checkpoint]
        acc_history = model.fit(train_x, train_y, epochs = 5, shuffle = 1, batch_size = 30,
callbacks=callback_list,validation_data = (vali_x, vali_y))
        model.save_weights("U:\Thesis\\CNN_py\\CNN model\\16filter_10window_withact\\Final_model.hdf5")
        #Save final model

        plt.plot(acc_history.history['acc']) #Plot training acc
        plt.plot(acc_history.history['val_acc'])#Plot validation acc
        plt.show() #Show figure

        plt.plot(acc_history.history['loss']) #Plot training loss
        plt.plot(acc_history.history['val_loss']) #Plot validation loss
        plt.show() #Show figure
        score = model.evaluate(test_x, test_y)#Calculate model performance
```

8. Program that build CNN model, and train with 10-fold-cross-validation

```
"""
Created on Tue Feb 19 15:57:57 2019
Input: Training set and trainign labels
Output: A table with the accuracies for 10-fold cross validation
@author: Xiaoyu Yu
File name: CNN_crossval
Description: This program is to train the CNN, and use 10-fold cross validation on the model.
"""
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, Conv1D, MaxPooling1D, Activation,
Dropout
from tensorflow.keras import optimizers
from tensorflow.keras.utils import plot_model
from tensorflow.keras.callbacks import ModelCheckpoint
from sklearn.model_selection import StratifiedKFold


seed = 7 #RNG seed for generating validation set
train_x = np.load('train_x.npy') #Load training data
train_y = np.load('train_y.npy') #Load training label
train_info = np.load('train_info.npy') #Load training info

train_x = train_x[:,:,:,np.newaxis]
#Create new dimension for training data. This step is not needed when multiple locations are used



kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
#Generate indices for 10 validation sets

score = [] #Variable to store accuracy for each fold

for train, test in kfold.split(train_x, train_y):
        '''
        Start of model
        '''
  model = tf.keras.Sequential()
        '''
        Covnolution group1
        '''
  model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv1',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
  model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv2',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
  model.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
        #Add max-pooling layer, pooling size (2,2), stride = 1, output size =input size
  model.add(Dropout(0.25))
        #Add dropout, dropout rate = 0.25

        '''
        Convolution group2
        '''
  model.add(Conv2D(64, kernel_size = (3,3), activation = 'relu', name = 'Conv3',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
  model.add(Conv2D(64, kernel_size = (3,3), activation = 'relu', name = 'Conv4',padding = 'same'))
        #Add convolution layer, 32 filter, kernel size (3,3), relu activation function, output size =input size
  model.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
```

```
                #Add max-pooling layer, pooling size (2,2), stride = 1, output size =input size
            model.add(Dropout(0.25))
                #Add dropout, dropout rate = 0.25

                '''
                Fully connected layers
                '''
            model.add(Flatten())
                #Convert multi-dimension data into 1D
            model.add(Dense(128))
                #Add Fully connected layer, 128 hidden nodes
            model.add(Activation('relu'))
                #Add Relu activation function
            model.add(Dropout(0.5))
                #Add dropout layer, dropout rate = 0.5
            model.add(Dense(2))
                #Add Fully connected layer, 2 hidden nodes
            model.add(Activation('softmax'))
                #Add softmax layer
                '''
                End of mdoel
                '''
            sgd = optimizers.SGD(lr=0.005)
                #Define schocastic gradient descent as optimizer, learning rate = 0.05
            model.compile(optimizer = sgd, loss = 'categorical_crossentropy', metrics = ['accuracy'])
                #Compile model, optimizer is SGD, loss function is categorical crossentropy, evaluation method: accuracy
            filepath="weights-improvement-{epoch:02d}.hdf5"
                #Directory to save model
            checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose = 0, save_best_only = True,
save_weights_only = True, mode='auto')
                #Save model when accuracy is best, save weight only
            callback_list = [checkpoint]
            train_y_train=train_y[train]
                #Training label
            train_y_train = tf.keras.utils.to_categorical(train_y_train)
                #Convert training label to 1-hot vector
            train_y_val = train_y[test]
                #Validation label
            train_y_val = tf.keras.utils.to_categorical(train_y_val)
                #Convert validation label to 1-hot vector
            model.fit(train_x[train], train_y_train, epochs = 100, shuffle = 1, batch_size = 30, callbacks=callback_list)
            #Train model, 100 epochs, 30 batch size, save model
            score.append(model.evaluate(train_x[test], train_y_val))
                #Calculate validation accuracy
```

9. Program that do data preparation for RNN model

```
"""
Created on Wed Mar 27 15:51:22 2019
Input: waist data, relabeling indices
Output: training set, validation set, testing set for RNN
@author: Xiaoyu Yu
File name: create_window_waist_RNN
Description: This program is to format the data in the way that
it can be fed into Keras RNN.
Also this program divide the data into training set, validation set and testing set.
"""
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

index_relabel = np.load("U:\Thesis\\CNN_py\\index_relabel.npy")
#Load relabeling indices
data_x = np.load('U:\Thesis\\CNN_py\\data_waist_x.npy')
#Load waist data
data_y = np.load('U:\Thesis\\CNN_py\\data_waist_y.npy')
#Load waist labels



file_len = 150 #150 data in each sequence (All data files are longer than 150)
overlap = 5    #overlap between adjacent windows
win_size = 10  #window size
input_x = []#Variable to store 150 data samples
input_y = []#Variable to store labels for each data file
train_x = []#Traning data
train_y = []#Training label
train_info = []#Training data info
test_x = []#Testing data
test_y = []#Testing label
test_info = []#Testing data info
vali_x = []#Validation data
vali_y = []#Validation label
vali_info = []#validation data info
count = 0

for i in range(len(data_x)):
    data_info = data_y[i][1:3]
    index_start = index_relabel[i][0]#activity start index
    index_end = index_relabel[i][1]  #activity end index
    len_data_mod = index_end - index_start #length of activity
    act = int(data_y[i][1])

    #fall = 1 indicate fall activity;fall = -1 indicate non-fall activity
    if act > 900:
        fall = 1
    else:
        fall = -1

    '''
    The following code is to crop each data files
    so that the length of each data file is 150
    '''
    input_y = np.zeros(len(data_x[i]))
    #Create all zero vector, length equal to the length of the data file
    #Zero represent precondition and postcondition
    input_y[index_start:index_end] = fall
```

```
    #Change the label between two indices to 1(fall activity) or 0(non-fall activity)
    if index_end < file_len:  #[start, end of activity] is shorter than 150
        input_x= data_x[i][0:file_len] #Keep data [0, 150]
        input_y = input_y[0:file_len]  #Keep labels [0, 150]
    elif len_data_mod < file_len: #else if activity length is shorter than 150
        input_x = data_x[i][index_end-file_len:index_end]
        #Keep data[activity end -150, activity end]
        input_y = input_y[index_end-file_len:index_end]
                        #Keep label[activity end -150, activity end]
    else: #else if activity length is longer than 150
        input_x = data_x[i][index_start:index_start+file_len]
        #Keep the first 150 data in activity
        input_y = input_y[index_start:index_start+file_len]
                        #Keep the first 150 labels in activity

    num_window = int(file_len/win_size)#Calculate number of windows
    input_x_win = []#Variable to store data in each data file
    input_y_win = []#Variable to store labels in each data file
    for j in range(num_window):
        input_x_win.append(input_x[win_size*j:win_size*(j+1)])
        #Data of 1 data file consists of 15 windows, each window has 10 observations
        input_y_win.append(fall*(sum(input_y[win_size*j:win_size*(j+1)])!=0))
        #Label of 1 data file consists of 15 windows, each window has 10 labels

    sub = int(data_info[1][24:27])#Human subject information

    if sub == 101 or sub == 102 or sub == 203 or sub == 204:
    #If data is from these four human subjects, data belogns to testing set
        test_x.append(input_x_win)#Store testing data
        test_y.append(input_y_win)#Store testing labels
        test_info.append(data_info)#Store other testing infos
    elif sub == 103:
            #If data is from this human subject, data belongs to testing set
        vali_x.append(input_x_win)#Store validation set
        vali_y.append(input_y_win)#Store validation labels
        vali_info.append(data_info)#Store other validation infos
    else:
            #Data from all other human subject belongs to training set
        train_x.append(input_x_win)#Store training data
        train_y.append(input_y_win)#Store training labels
        train_info.append(data_info)#Store other training infos

train_max = max(max(max(train_x)))#Find out max value for training data
test_max = max(max(max(test_x)))#Find out max value for testing data
vali_max = max(max(max(vali_x)))#Find out max value for validation data

train_x = np.asarray(train_x)#Convert training data type from list to numpy array
train_x=train_x/train_max#Scale training data to [-1,1]
train_x = train_x[:,:,:,:,np.newaxis]#Add an additional dimension to training data
train_y_oh = np.zeros((len(train_y), num_window,3))#Create variable to store training labels

test_x = np.asarray(test_x)#Convert testing data type from list to numpy array
train_x=train_x/train_max#Scale training data to [-1,1]
test_x = test_x[:,:,:,:,np.newaxis]#Add an additional dimension to testing data
test_y_oh = np.zeros((len(test_y), num_window,3))#Create variable to store testing labels

vali_x = np.asarray(vali_x)#Convert validation data type from list to numpy array
train_x=train_x/train_max#Scale training data to [-1,1]
vali_x = vali_x[:,:,:,:,np.newaxis]#Add an additional dimension to validation data
vali_y_oh = np.zeros((len(vali_y), num_window,3))#Create variable to store valdation labels

'''
```

```
Convert labels to one-hot vectors
'''
for i in range(len(train_x)):#Scan through training data
    for j in range(num_window):#Scan through each data file
        x = train_y[i][j]#Activity label
        train_y_oh[i][j] = tf.keras.utils.to_categorical(x, 3)
        #Convert categorical data into one-hot vector

for i in range(len(test_x)):#Scan through testing data
    for j in range(num_window):  #Scan through each data file
        y = test_y[i][j]#Activity label
        test_y_oh[i][j] = tf.keras.utils.to_categorical(y, 3)
        #Convert categorical data into one-hot vector

for i in range(len(vali_x)):#Scan through validation data
    for j in range(num_window): #Scan through each data file
        z = vali_y[i][j]#Activity label
        vali_y_oh[i][j] = tf.keras.utils.to_categorical(z, 3)
        #Convert categorical data into one-hot vector

np.save('U:\Thesis\RNN_py\\train_x.npy', train_x)#Save training data
np.save('U:\Thesis\RNN_py\\train_y.npy', train_y_oh)#Save training labels
np.save('U:\Thesis\RNN_py\\train_info', train_info)#Save training infos
np.save('U:\Thesis\RNN_py\\test_x.npy', test_x)#Save testing data
np.save('U:\Thesis\RNN_py\\test_y.npy', test_y_oh)#Save testing labels
np.save('U:\Thesis\RNN_py\\test_info', test_info)#Save testing infos
np.save('U:\Thesis\RNN_py\\vali_x.npy', vali_x)#Save validation data
np.save('U:\Thesis\RNN_py\\vali_y.npy', vali_y_oh)#Save validation labels
np.save('U:\Thesis\RNN_py\\vali_info', vali_info)#Save validation infos
```

10. Program that build, train and save RNN model

```
"""
Created on Tue Apr 30 15:48:19 2019
Input: training set, validation set, testing set
Output: trained RNN model, testing result, training/validation curve
@author: Xiaoyu Yu
File name: RNN_test
Description: This program is to train the RNN model, save the model and test with test set.
"""
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten, TimeDistributed, Dropout, Conv2D, MaxPooling2D,
GRU, LSTM, ConvLSTM2D
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import optimizers
import numpy as np

train_x = np.load('U:\Thesis\RNN_py\\train_x.npy')
train_y = np.load('U:\Thesis\RNN_py\\train_y.npy')
train_info = np.load('U:\Thesis\RNN_py\\train_info.npy')

test_x = np.load('U:\Thesis\RNN_py\\test_x.npy')
test_y = np.load('U:\Thesis\RNN_py\\test_y.npy')
test_info = np.load('U:\Thesis\RNN_py\\test_info.npy')

vali_x = np.load('U:\Thesis\RNN_py\\vali_x.npy')
vali_y = np.load('U:\Thesis\RNN_py\\vali_y.npy')
vali_info = np.load('U:\Thesis\RNN_py\\vali_info.npy')

#Start of neural network structure
'''
Convolution layers
'''
#Structure for convolution layers
#This part is used in LSTM with convolution layers
#This part is not used in LSTM with convolution transformation
CNN = Sequential()
CNN.add(Conv2D(16, kernel_size = (3,3), activation = 'relu', name = 'Conv1_1',padding = 'same'))
CNN.add(Conv2D(16, kernel_size = (3,3), activation = 'relu', name = 'Conv1_2',padding = 'same'))
CNN.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
CNN.add(Dropout(0.5))
CNN.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv2_1',padding = 'same'))
CNN.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', name = 'Conv2_2',padding = 'same'))
CNN.add(MaxPooling2D(pool_size = (2,2), strides = None, padding = 'same', data_format = None))
CNN.add(Dropout(0.5))
CNN.add(Flatten())
#End of convolution layers

#Structure RNN model
model = Sequential()#Build a sequential model

'''
Model that use LSTM with convolution transformation
This part is used for LSTM with convolution transformation
This part is not used for LSTM with convolution layers
'''
#model.add(ConvLSTM2D(16, kernel_size = [3,3], padding = 'same', data_format = 'channels_last',  return_sequences
= True, recurrent_dropout = 0.5, name = 'lstm1'))
#model.add(ConvLSTM2D(16, kernel_size = [3,3], padding = 'same', data_format = 'channels_last',  return_sequences
= True, name = 'lstm2'))
```

```
        '''
        Model that use LSTM along with CNN layers
        This part is used for LSTM with convolution layers
        This part is not used for LSTM with convolution transformation
        '''
        model.add(TimeDistributed(CNN))#Add CNN part into the model
        model.add(LSTM(32, return_sequences = True, recurrent_dropout = 0.5, name = 'RNN1'))
        #Add a LSTM layer
        model.add(TimeDistributed(Flatten()))#Convert multi-dimension data into 1-D
        model.add(TimeDistributed(Dense(3, name = 'FC1')))#Fully connected layer
        model.add(Activation('softmax', name = 'softmax'))#Soft-max Layer
        #End of neural network struture

        sgd = optimizers.SGD(lr=0.002)
        #Define optimizer, use stochastic gradient descent as optimizer, 0.02 as learning rate


        model.compile(loss = 'categorical_crossentropy',
                      optimizer = sgd,
                       metrics = ['accuracy'])
        #Compile data, use categorical crossentropy as loss, sgd as optimizer, accuracy as evaluation parameter


        filepath="U:\Thesis\\RNN_py\\RNN model\\Simple RNN with Conv1\\32-nodes-RNN-1-layer-{epoch:02d}-
{val_acc:.2f}.hdf5"
        #Directory to save models
        checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose = 0, save_best_only = True, save_weights_only =
True, mode='auto')
        #Save best model based on validation accuracy, save weights only
        callback_list = [checkpoint]
        acc_history = model.fit(train_x, train_y, epochs = 100, shuffle = 1, batch_size = 1,
callbacks=callback_list,validation_data = (vali_x, vali_y))
        #Train model for 100 epochs, shuffle for every training epochs, batch size is 1, save training history
        model.save_weights("U:\Thesis\\RNN_py\\RNN model\\Simple RNN with Conv1\\Final_model.hdf5")
        #Save final model

        #Plot training/validation acc&loss
        plt.plot(acc_history.history['acc'])#Plot training accuracy
        plt.plot(acc_history.history['val_acc'])#Plot validation accuracy
        plt.show()

        plt.plot(acc_history.history['loss'])#Plot validation accuracy
        plt.plot(acc_history.history['val_loss'])#Plot validation loss
        plt.show()

        score = model.evaluate(test_x, test_y)
        #Run trained model on testing set
```

APPENDIX C: IRB CERTIFICATE

Email from Purdue IRB:

Dear Professor Panigrahi,
Thank you for your submission. We have reviewed the above-referenced project and determined that it does not meet the definition of human subjects research as defined by 45 CFR 46. This determination is the Purdue
HRPP assessment of regulations related only to human subjects research protections. This determination does not constitute approval from any other Purdue campus department or outside agency. The Principal Investigator
and all researchers are required to affirm that the research meets all applicable local, state, and federal laws that may apply.

You are required to retain a copy of this letter for your records. We appreciate your commitment towards ensuring the ethical conduct of human subjects research and wish you luck with your project.

Best,

Angelina

Angelina Riggs
Project Assistant
Purdue University
Human Research Protection Program
Young Hall – Rm. 1032
irb@purdue.edu
Phone: 494-7090
For more information, visit our website at www.irb.purdue.edu

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., . . . Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. Retrieved from https://www.tensorflow.org/

Anaconda Software Distribution. Computer software. Vers. 2-2.4.0. Anaconda, Nov. 2016. Web. <https://anaconda.com>.

Anguita, D., Ghio, A., Oneto, L., Llanas Parra, F. X., & Reyes Ortiz, J. L. (2013). Energy efficient smartphone-based activity recognition using fixed-point arithmetic. *Journal of Universal Computer Science*, *19*, 1295–1314.

Ann, O. C., & Theng, L. B. (2014, Nov). Human activity recognition: A review. In *2014 IEEE International Conference on Control System, Computing and Engineering* (pp. 389-393). doi:10.1109/ICCSCE.2014.7072750

Bächlin, M., Plotnik, M., Roggen, D., Giladi, N., Hausdorff, J. M., & Tröster, G. (2010). A wearable system to assist walking of Parkinson s disease patients. *Methods of Information in Medicine*, *49*, 88–95.

Bao, L., & Intille, S. S. (2004). Activity recognition from user-annotated acceleration data. In *International Conference on Pervasive Computing* (pp. 1–17).

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010* (pp. 177-186). Physica-Verlag HD.

Bergen, G. (2016). Falls and fall injuries among adults aged≥ 65 years—United States, 2014. *MMWR. Morbidity and mortality weekly report*, *65*.

Chen, L., Nugent, C. D., & Wang, H. (2012, June). A knowledge-driven approach to activity recognition in smart homes. *IEEE Transactions on Knowledge and Data Engineering*, *24*, 961-974. doi:10.1109/TKDE.2011.51

Chen, Y., Guo, M., & Wang, Z. (2016). An improved algorithm for human activity recognition using wearable sensors. In *Advanced Computational Intelligence 2016 Eighth International Conference on* (pp. 248–252).

Chollet, F. (2015). Keras. Web. <https://keras.io>.

De Leonardis, G., Rosati, S., Balestra, G., Agostini, V., Panero, E., Gastaldi, L., & Knaflitz, M. (2018). Human activity recognition by wearable sensors: Comparison of different classifiers for real-time applications. In *2018 IEEE International Symposium on Medical Measurements and Applications* (pp. 1–6).

Ding, G., Tian, J., Wu, J., Zhao, Q., & Xie, L. (2018). Energy efficient human activity recognition using wearable sensors. In *Wireless communications and networking conference workshops (wcncw), 2018 ieee* (pp. 379–383).

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

Hammerla, N. Y., Halloran, S., & Ploetz, T. (2016). Deep, convolutional, and recurrent models for human activity recognition using wearables. *arXiv preprint arXiv:1604.08880* .

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation, 9*, 1735-1780.

Fakhrulddin, A. H., Fei, X., & Li, H. (2017). Convolutional neural networks (cnn) based human fall detection on body sensor networks (bsn) sensor data. *In Systems and informatics (icsai), 2017 4th international conference on* (pp.1461–1465).

Hammerla, N. Y., Halloran, S., & Ploetz, T. (2016). Deep, convolutional, and recurrent models for human activity recognition using wearables. *arXivpreprint arXiv:1604.08880*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*, 1735–1780.

Hosseini-Asl, E., Keynton, R., & El-Baz, A. (2016, Sept). Alzheimer's disease diagnostics by adaptation of 3d convolutional network. In *2016 ieee international conference on image processing (icip)* (p. 126-130). doi:10.1109/ICIP.2016.7532332

Jiang, W., & Yin, Z. (2015). Human activity recognition using wearable sensors by deep convolutional neural networks. In *Proceedings of the 23rd ACM International Conference on Multimedia* (pp. 1307–1310).

Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078.*

Katoch, E., & Augustyniak, P. (2012, September). Human activity surveillance based on wearable body sensor network. In *2012 computing in cardiology* (pp. 325-328).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (pp. 1097–1105).

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*, 2278–2324.

Lee, S.-M., Yoon, S. M., & Cho, H. (2017, Feb). Human activity recognition from accelerometer data using convolutional neural network. In *2017 IEEE International Conference on Big Data and Smart Computing* (pp. 131-134). doi:10.1109/BIGCOMP.2017.7881728

Li, D., Zhang, J., Zhang, Q., & Wei, X. (2017). Classification of ecg signals based on 1d convolution neural network. In *e-health networking, applications and services (healthcom), 2017 ieee 19th international conference on* (pp. 1–6).

Mejia-Ricart, L. F., Helling, P., & Olmsted, A. (2017). Evaluate action primitives for human activity recognition using unsupervised learning approach. In *2017 12th International Conference for Internet Technology and Secured Transactions* (pp. 186–188).

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 807-814).

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). San Francisco, CA: Determination Press.

Nghiem, A. T., Auvinet, E., & Meunier, J. (2012, July). Head detection using kinect camera and its application to fall detection. In *2012 11th international conference on information science, signal processing and their applications (isspa)* (pp. 164-169). doi:10.1109/ISSPA.2012.6310538

Özdemir, A. T. (2016). An analysis on sensor locations of the human body for wearable fall detection devices: Principles and practice. *Sensors, 16*, 1161.

Özdemir, A. T., & Barshan, B. (2014). Detecting falls with wearable sensors using machine learning techniques. *Sensors, 14*, 10691–10708.

Pierleoni, P., Belli, A., Palma, L., Pellegrini, M., Pernini, L., & Valenti, S. (2015, August). A high reliability wearable device for elderly fall detection. *IEEE Sensors Journal, 15*, 4544-4553. doi:10.1109/JSEN.2015.2423562

Proakis, J. G. (2001). *Digital signal processing: principles algorithms and applications*. City, India: Pearson Education.

Reiss, A., & Stricker, D. (2012a). Introducing a new benchmarked dataset for activity monitoring. *In Wearable Computers (iswc), 2012 16th international symposium on* (pp. 108-109).

Reiss, A., & Stricker, D. (2012b, June). Introducing a new benchmarked dataset for activity monitoring. *In 2012 16th international symposium on wearable computers* (pp. 108-109). doi:10.1109/ISWC.2012.13

Roggen, D., Calatroni, A., Rossi, M., Holleczek, T., Förster, K., Tröster, G., … others (2010). Collecting complex activity datasets in highly rich networked sensor environments. *In Networked sensing systems (inss), 2010 Seventh International Conference* on (pp. 233–240).

Sazonov, E. S., Fulk, G., Hill, J., Schutz, Y., & Browning, R. (2011, April). Monitoring of posture allocations and activities by a shoe-based wearable sensor. *IEEE Transactions on Biomedical Engineering, 58,* 983-990. doi:10.1109/TBME.2010.2046738

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15,* 1929-1958.

Steven Eyobu, O., & Han, D. (2018). Feature representation and data augmentation for human activity classification based on wearable IMU sensor data using a deep LSTM neural network. *Sensors,18,* 2892.

Stisen, A., Blunck, H., Bhattacharya, S., Prentow, T. S., Kjærgaard, M. B., Dey, A., . . . Jensen, M. M. (2015). Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM conference on embedded networked sensor systems* (pp. 127–140). doi:10.1145/2809695.2809718

Sun, L., Zhang, D., Li, B., Guo, B., & Li, S. (2010). Activity recognition on an accelerometer embedded mobile phone with varying positions and orientations. In *International Conference on Ubiquitous Intelligence and Computing* (pp. 548–562).

Tian, D., Xu, X., Tao, Y., & Wang, X. (2017, July). An improved activity recognition method based on smart watch data. In 2017 IEEE International Conference on Computational Science and Engineering and IEEE International Conference on Embedded and Ubiquitous Computing (Vol. 1, p. 756-759). doi:10.1109/CSE-EUC.2017.148

Vo, Q. V., Lee, G., & Choi, D. (2012). Fall detection based on movement and smart phone technology. In Computing and communication technologies, research, innovation, and vision for the future, 2012 ieee RIVF International Conference on (pp. 1–4).

Wang, X., Lu, Y., Wang, D., Liu, L., & Zhou, H. (2017). Using jaccard distance measure for unsupervised activity recognition with smartphone accelerometers. In *Asia-pacific web (apweb) and web-age information management (waim) joint conference on web and big data* (pp. 74–83).

Xsens. (2009). MTx sensor: MTi and MTx User Manual and Technical Documentation. Xsens Technologies B.V..

Zhang, M., & Sawchuk, A. A. (2012). Motion primitive-based human activity recognition using a bag-of-features approach. In *Proceedings of the 2nd ACM sighit international health informatics symposium* (pp. 631–640). doi:10.1145/2110363.2110433

Zhang, T., Wang, J., Liu, P., & Hou, J. (2006). Fall detection by embedding an accelerometer in cellphone and using kfd algorithm. *International Journal of Computer Science and Network Security*, *6*, 277–284.

Zhao, S., Li, W., Niu, W., Gravina, R., & Fortino, G. (2018, March). Recognition of human fall events based on single tri-axial gyroscope. In *2018 IEEE 15th international conference on networking, sensing and control* (p. 1-6). doi:10.1109/ICNSC.2018.8361365