Technical University of Applied Sciences Würzburg-Schweinfurt (THWS)
Faculty of Computer Science and Business Information Systems

Master Thesis

Step Detection

submitted to the Technical University of Applied Sciences Würzburg-Schweinfurt in the Faculty of Computer Science and Business Information Systems to complete a course of studies in Master of Artificial Intelligence

Muhammad Zakriya Shah Sarwar
K54738

Submitted on: 20.10.2023

Initial examiner: Prof. Dr. Magda Gregorova
Secondary examiner: Prof. Dr. Frank Deinzer

Abstract

This thesis investigates the domain of step detection using neural network models applied to accelerometer and gyroscope data. The study investigates two different approaches to step detection. The first approach employs Convolutional Neural Networks (CNNs) to extract spatial features from time-series data, enabling the precise localization of step events. Departing from previously used Long Short-Term Memory (LSTM) models, this alternative demonstrates higher accuracy and remarkably improves inference speed. The second approach adopts a YOLO-like architecture, aiming to predict steps in start-end pairs, thus minimizing post-processing requirements. While the CNN-based models exhibit superior accuracy compared to LSTM models, the YOLO-like model excels in real-time processing efficiency. Traditionally, the accuracy of the models was described by comparing the predicted number of steps to the original number of steps and the F1 score. This work also investigates how accurate the model is in predicting steps close to the actual labels by calculating mean absolute error.

Abstrakt

In dieser Arbeit wird der Bereich der Stufenerkennung mit Hilfe von neuronalen Netzmodellen untersucht, die auf Beschleunigungsmesser- und Gyroskopdaten angewendet werden. In der Studie werden zwei verschiedene Ansätze zur Stufenerkennung untersucht. Der erste Ansatz verwendet Convolutional Neural Networks (CNNs), um räumliche Merkmale aus Zeitreihendaten zu extrahieren, was eine präzise Lokalisierung von Schrittereignissen ermöglicht. Im Gegensatz zu den bisher verwendeten LSTM-Modellen (Long Short-Term Memory) weist diese Alternative eine höhere Genauigkeit auf und verbessert die Geschwindigkeit der Schlussfolgerungen erheblich. Der zweite Ansatz verwendet eine YOLO-ähnliche Architektur, die darauf abzielt, Schritte in Start-Ende-Paaren vorherzusagen und so die Nachbearbeitungsanforderungen zu minimieren. Während die CNN-basierten Modelle im Vergleich zu LSTM-Modellen eine höhere Richtig keit aufweisen, zeichnet sich das YOLO-ähnliche Modell durch eine effizientere Echtzeitverarbeitung aus. Traditionell wurde die Genauigkeit der Modelle durch den Vergleich der vorhergesagten Anzahl von Schritten mit der ursprünglichen Anzahl von Schritten und dem F1-Score beschrieben. In dieser Arbeit wird auch untersucht, wie genau das Modell Schritte vorhersagt, die nahe an den tatsächlichen Kennzeichnungen liegen, indem der mittlere absolute Fehler berechnet wird.

Acknowledgment

I would like to thank Prof. Dr. Magda Gregorová for her continuous input and feedback on the research and writing of this thesis work. I would also like to thank Markus Ebner and Steffen Kastner for making the dataset available.

Contents

1 Introduction **1**

2 Background **3**
2.1 Indoor Localization and Health 3
2.2 Machine Learning Basics 5
2.3 Artificial Neural Networks 9
2.4 Literature Review of step detection methods 16

3 Step Dataset **21**
3.1 Acquisition 21
3.2 Pre-processing 23
3.3 Steps Statistics 29

4 Models **31**
4.1 LSTM - A reference method 31
4.2 CNN based model 34
4.3 YOLO-Like 38
4.4 Hyperparameters 41

5 Experiments and Results **43**
5.1 Metrics 43
5.2 Results 45

6 Conclusion **53**
6.1 General Discussion 53
6.2 Future work 54
6.3 Personal Notes 55

Appendix **61**

Literature **67**

Declaration on oath **71**

Consent to plagiarism check **73**

1 Introduction

This thesis aims to develop a framework to detect when a user takes a step by analyzing the signal from the accelerometer and gyro-meter of a mobile device. This is now used more and more commonly to give users a summary of their daily activities and to locate them with high accuracy. In particular, this thesis work aims to apply neural networks to detect steps.

Mostly, steps are detected using heuristic methods such as peak detection in the accelerometer signal. Researchers manually look at the data and decide on the thresholds to be used in the deterministic algorithm. These methods are heavily constrained by the position of the device on the body and the type of walk. A mobile can be in the hands or in a pocket also, a person can be walking upstairs or downstairs. Similarly, a slight change in the walk pattern can also lead to the failure of the algorithm.

Traditional machine-learning algorithms have also been used for this task, like decision trees. This requires manual feature selection. These features can be mean, skewness, and peak heights. After careful selection of features to be used, they are extracted from the signal, and a machine learning algorithm is applied to determine the steps in the signal. However, these features can vary from person to person and be required to be adopted.

Some methods have been developed based on neural networks as well. They typically involve using a sliding window. The window is fed into the neural network model to obtain detected steps. Neural networks have the ability to be trained on a wide range of data and to learn and choose features that are best suited for the task. This thesis work tries to improve already built methods and suggest some different model architects.

The dataset used in this work was collected by wearing IMU sensors simultaneously on both feet, in the side and back pockets, and in handy. The participants were asked to walk at different speeds and in different scenarios, such as on stairs. The data was labeled using the feet sensors, which were synchronized with all other sensors.

The rest of the chapters and sections go as follows. In the background chapter, I will discuss why this task is useful and the basics of machine learning. Then, I will try to review the current methods. In the Step Dataset chapter, I will discuss how data was collected and processed before going into the networks for detection. In the chapter

Models, I will explain the neural networks I used in detail. The Experiments and Results chapter will discuss the accuracy of the proposed methods.

2 Background

2.1 Indoor Localization and Health

Detecting when a person takes a step during different types of walks is crucial in two main areas. One is accurately locating a person inside buildings, and the other is monitoring the health. These objectives can be achieved using only the built-in sensor found in mobile devices, known as the inertial measurement unit (IMU), without additional sensors.

The process of finding the exact location of a device within a small area, especially inside buildings, is called indoor localization. Global Positioning System launched by the United States in the 1970's gives the location of the user anywhere on earth. It has a typical accuracy within $4.9m$ [7] in open sky. However, this accuracy radically decreases when an obstacle between the user and the satellite exists. GPS completely fails to guide the mobile user where they are inside the building. For example, it is very difficult to locate which platform you are at a railway station. To achieve this accuracy, different methods have been developed using WiFi or Bluetooth. Distance to fixed-positioned WiFi access points are sometimes measured to estimate the location [24] [5].

One other method is called dead reckoning. In this method, the location of the user is estimated based on previously determined location or using the speed and time. In pedestrian dead reckoning, the signal from the IMU of a mobile device of a walking person is used. Detection of steps is the main component of this method. [16] uses this technique to localize the mobile inside different buildings.

Accurate detection of steps provides important insight into a person's activity and movement patterns. Many mobile applications provide useful summaries of users' daily activity by counting total steps. It is also useful for the progress monitoring of injury recovery. In the elderly, the caregivers can monitor their health by analyzing their walk and activity levels.

2.1.1 Sensor

An inertial measurement unit (IMU) is a device that measures proper acceleration, angular velocity, and body orientation using a combination of accelerometer and gyroscope. IMUs were historically used to navigate airplanes, satellites, space rockets, and missiles. But production of small-size digital IMUs has enabled their use in mobile phones, tablets, and smartwatches. This enables these gadgets to track orientation and movement. This is used to auto-rotate screens in gaming and in virtual reality. Many mobile health applications use IMUs to track the user's movements.

An accelerometer is a sensor that measures proper acceleration. Proper acceleration is acceleration relative to free fall. On earth's surface, everything experiences a constant acceleration g of $9.8m/s^{-2}$, and proper acceleration will be total acceleration minus g. A typical mechanical accelerometer has a spring attached to a mass. Acceleration on the sensor causes the mass to move some distance. The distance covered can be calculated by Hooke's Law as stated in equation 2.1. The sensor will measure the distance, and the distance can be converted to acceleration.

$$F = -Kx \tag{2.1}$$

Here K is the spring constant, and x is the distance covered by spring. Because the mass is known, F can be easily converted to acceleration.

A gyroscope is a device to measure or keep the orientation and angular velocity of the body. It is made of a disc and supporting outer frame as shown in the image 2.1 [34]. The disc can rotate freely in any orientation. When rotating, the disc will keep its axis of rotation even if the supporting frame changes its orientation. A sensor measures the difference in alignment of the disc's axis and the orientation of the outer frame.

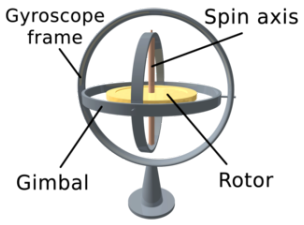


Figure 2.1: Gyroscope

Accelerometers and gyroscopes are also available as electric sensors. At the core of both sensors is a piezoelectric material. A piezoelectric material generates an electric charge in response to stress or vibration. The sensor will work as follows. The piezoelectric material is housed together with a seismic mass. When the sensor body is moved, the mass will resist the motion due to inertia and exert a force on the piezoelectric.

Piezoelectric, in return, produces an electric charge, which the sensor will measure. This reading can be converted to acceleration or angular velocity depending on the sensor type.

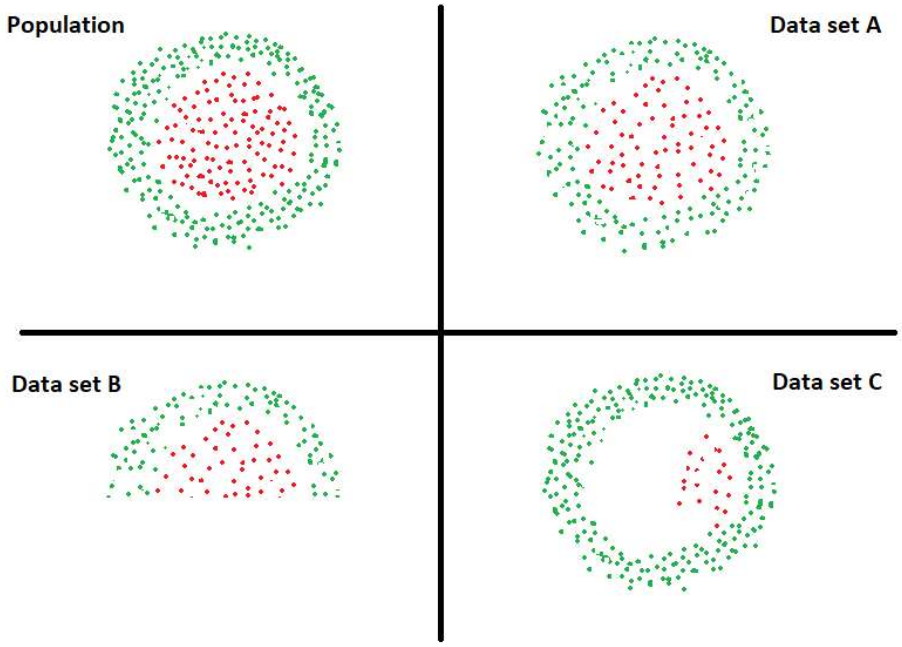
The electronic sensors measure the acceleration and angular velocity with a specific frequency. There are sensors with low, medium, and high-frequency ranges. Low-frequency sensors are typically in the range of 1 to 100 Hz. High-frequency sensors can measure readings up to the 10th of a millisecond. For our purpose, low-frequency IMUs are more suitable. The sensors used in this thesis work have a frequency of 80Hz. In other words, they measure values every 12.5 milliseconds.

2.2 Machine Learning Basics

Machine learning is the process of learning mathematical equations which can convert inputs to desired outputs. The learning term here implies that the equation is learned through a set of data called a training data set. A learned equation should be able to produce the desired output for the input that was not originally present in the training data set. The term machine learning was first coined by Arthur Samuel [31] in 1959. In his paper, he investigated two machine-learning procedures for the game of checkers. He wrote a program that could learn to play the game better than him.

Computers are usually programmed to do a specific task. Programs have some set of rules in them, which come from the understanding of the given task by the programmer. A simple example could be turning on the street lights when the light sensor's reading is above or below a threshold value. In machine learning, the task is to write a program, usually called a model, that will learn the threshold value. The learning will be done through the data set, which is probably crafted by humans in such a way that is suitable to the task. So one of the most crucial things for learning is the data and in large quantity.

The machine learning models are validated to check if and how will they perform on unseen inputs. This is achieved by dividing the data set. A small percentage of data is separated for the sole purpose of testing the model. If the data set truly represents the actual phenomena or system, then the train test split is usually an indicator that the model will perform adequately on unseen input. On the other hand, if the data set itself had some kind of bias or it was not generalizing the true population, then the model will never be able to generalize. Figure 2.2 data set A is a good example of how the training data should be sampled from the population. Data set B is entirely missing the lower half, so the model will most likely behave randomly to the inputs from that region. In data set C, the red class is underrepresented and is biased toward the right side.



-

Figure 2.2: Representative Dataset

Machine learning has three main types namely supervised, unsupervised, and reinforcement learning. In supervised learning the data set for training includes both the inputs and desired outputs, commonly known as labels. A mathematical model is constructed such that during training, input is fed to the model and its output is compared to the true label. For example in a task of classification of cats or dogs, the data set will have images of the two classes (cat and dog) and a label for each image whether it is a cat or a dog. A model will take an image and produce its prediction for the image. This prediction will be compared to the actual label, and based on this comparison the model will be improved as such that next time the prediction is correct. This comparison is called the measure of error or loss function. Some famous tasks in supervised learning are classification and regression

Unsupervised learning is the type in which there are no true labels to the inputs. The task is to find the structure in the data set. Clustering and dimensionality reduction are popular tasks in unsupervised learning. In clustering each input or data point is given an ID in such a way that points belonging to the same ID are more similar than points belonging to different IDs. In dimensionality reduction, the number of useful features of data points is reduced so that it still represents the original structure of data but occupies less computer memory.

In Reinforcement learning, we have an agent instead of a model. This agent has a

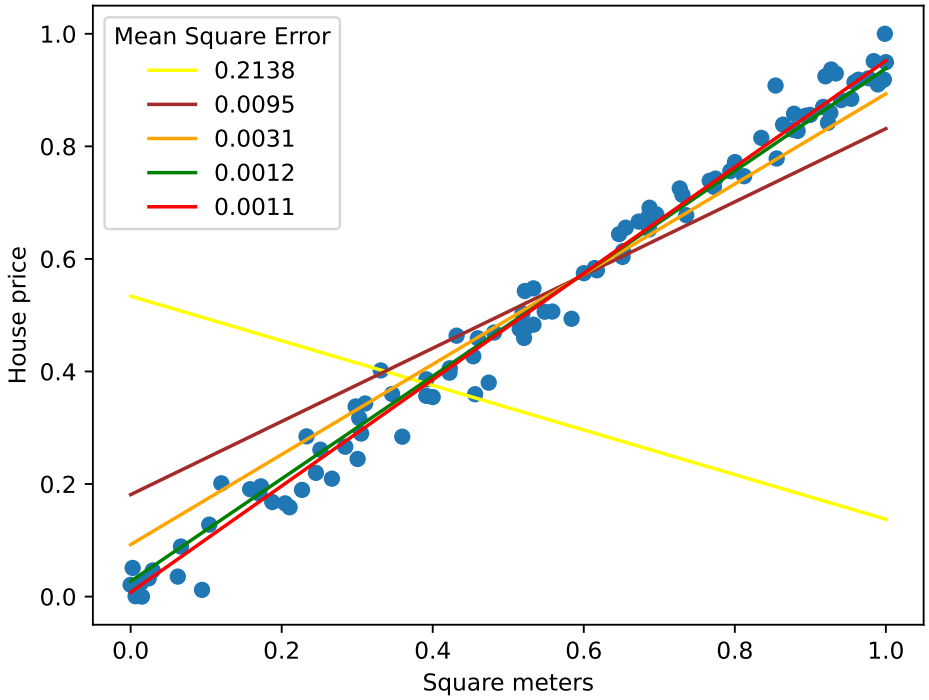


Figure 2.3: As the loss decreases, the prediction line better aligns itself to the spread of data. The yellow line is the result of randomly initialized parameters.

perceivable environment and can take action. The agent is rewarded based on those actions. After many iterations agent learns what actions to take and what actions are not suitable. The training is maintained such that the agent also explores the environment instead of just maximizing the reward quickly. This ensures that the agent can take better actions in an unseen environment.

Linear Regression: To give an example of supervised learning and explain more machine learning terms and methods, let us discuss simple linear regression with an example. Let's consider that we need to predict house prices based on the square meters of the house. In this case, price is a dependent variable, and area is an independent variable. Let's denote price as y and area as x. The plotted data is shown in Figure 2.3. Keep in mind that both square meters and prices are normalized to make them of equal scale. This is a common practice as housing prices can range to millions while square meters for houses range from 50 to $400m^2$.

In this case, x is our input, and y is our label. Linear regression, in this case, tries to fit a line with a minimum average distance from all input points. The line equation is in

equation 2.2. Where w is the slope of the line, b is the y-intercept of the line, and $\hat{y}$ is predicted house prices. The task of this learning problem is to learn the value of w and b.

$$\hat{y} = wx + b \tag{2.2}$$

We also need to check how well the line fits the data set. For this purpose, we will compare the distances between the actual price and the price predicted by the line. Then, we will find the average of all the distances. This will indicate if our line is close to true prices or not. We will use the square of the distances to avoid the cancellation of positive and negative distances. Squaring the distances will also have the impact of amplifying the differences. This loss function is called mean square error, and it can be written as in equation 2.3

$$L = \frac{1}{N}\sum_{i=0}^{N}(y - \hat{y})^2 \tag{2.3}$$

Where N is the total number of data points in our training data.

We start the learning process by randomly choosing the values for w and c. The next step is changing these values in such a way that loss is decreased. w and c of our equation are widely known as the parameters of our model, and the process of finding the values of these parameters which result in minimum loss is called optimization. Usually, the parameters are optimized in an iterative manner, slightly changing the values and decreasing loss in each iteration. One famous optimization technique is gradient descent [11], which updates the parameters using their old value and gradient of loss. The equation of parameter updating in gradient descent is as follows:

$$w_{new} = w_{old} - \alpha \nabla L(w_{old}) \tag{2.4}$$

Where α is the learning rate, which is usually a small value. And $\nabla L(w_{old})$ is the gradient of the loss function with respect to w_{old}, Which in this case will be the partial derivative of the loss function with respect to w_old

$$\nabla L(w_{old}) = \frac{2}{N}\sum_{i=0}^{N}(y - \hat{y})(-x_i) \tag{2.5}$$

Where x_i is the ith input to the model. We will have the same equation for the b, and w will be treated as constant in the partial derivative of the loss function with respect to b. The learning procedure will go like in the Algorithm 1.

After training the model, we will have an equation that best describes the data set in the given settings. The different lines that can be drawn through the data at different

Algorithm 1: Illustration of Gradient descent

1: Initialize w and c with random values
2: **while** Loss above threshold **do**
3: Find the loss value using equation 2.3
4: Update the parameters using equation 2.4.
5: **end while**

stages of learning are shown in figure 2.3. We can see that the red line has the least average distance. Other lines are drawn from early values of parameters during training. We can see that as the loss decreases, the line better aligns itself with the spread of data.

2.3 Artificial Neural Networks

Artificial neural networks are machine learning models loosely inspired by the neural connection in the human brain. Our brain is made up of cells called neurons that generate electrical signals. This signal is passed on to the target via synapses. Synapses are connections between the generator cell and the target, and they use chemicals to pass on the signals. Much like this, in deep learning, we have neurons and the connection between them. Neurons are, in this case, mathematical functions that transform inputs into outputs. Connection in artificial neural networks means that the output of one neuron is input to the other neuron.

Neural Networks were first proposed in 1943 by Warren McCulloch and Walter Pitts [28]. Yann LeCun et al. used backpropagation to train neural networks to recognize handwritten digits of zip codes [18]. Neural Networks were made popular when graphical processing units or GPUs were introduced. GPUs are able to do parallel computation, which reduces the time to run a process significantly. Huge neural networks can not be trained on a normal CPU in a reasonable time.

A building block of a neural network is a single neuron, essentially a mathematical function. A mathematical function is a set of relations defined between two variables. More formally, a function assigns from one set $\boldsymbol{X}$ to another set $\boldsymbol{Y}$ exactly one element from $\boldsymbol{Y}$ to each element in set $\boldsymbol{X}$. A function of a neuron can be $\hat{y} = wx + b$, same as in equation 2.2. w is referred to as the weight of the neuron, and b is called bias. Usually, the input and output are also shown as a neuron, as shown in Figure 2.4.

A network of neurons can be made by having multiple neurons in parallel and giving the outputs of previous neurons to the next neurons. Neurons in parallel are not connected with each other. One set of these parallel neurons is called a layer. The layers are

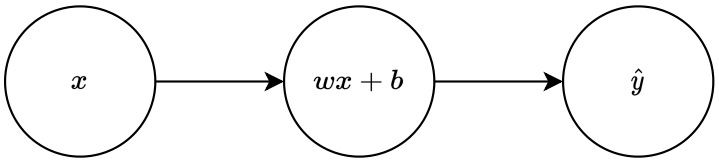


Figure 2.4: A single neuron with its function. Input x and output $\hat{y}$ are usually shown as same as a neuron.

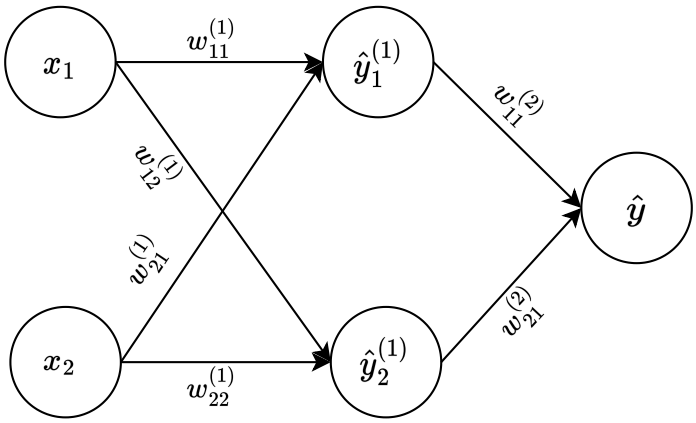


Figure 2.5: 2 layers neural network with input and output layer. w are weights with the superscript indicating the layer number, the first subscript indicates the number of neurons where the output comes from, and the third subscript indicates the receiver neuron's number. Bias b and outputs $\hat{y}$ have superscripts and subscripts to show the layer number and neuron number in that layer.

stacked, and the output of one layer is passed to the next layer. Most commonly, the output of a neuron from a layer becomes the input of every neuron in the second layer. This means neurons do not just take a single number as input but a vector or an array of values. They still produce a single-value output. This type of network is called a feed-forward network. We can change the function to accept a vector as follows:

$$\hat{y} = \mathbf{wx} + b \tag{2.6}$$

This function represents a weighted sum of the input with a bias. $\mathbf{wx}$ is dot product between vector of weights $\mathbf{w}$ and vector of inputs $\mathbf{x}$. Figure 2.5 shows a 2-layer network. 1st layer has two neurons, and the second layer has one neuron. It will produce one value output in real number space $\mathbb{R}$ and take input in space $\mathbb{R}^2$. Final equation of $\hat{y}$ in figure 2.5 can be written as:

$$\hat{y} = w_{11}^{(2)}(w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + b_1^{(1)}) + w_{21}^{(2)}(w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)}) + b_1^{(2)} \tag{2.7}$$

Weights w and biases b are the parameters of this model. These parameters are tuned to achieve the objective function. The equation 2.7 can also be written in matrix form

as follows.

$$\hat{y} = \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} \end{bmatrix} (\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}) + b_1^{(2)} \tag{2.8}$$

or

$$\hat{y} = \mathbf{w^{(2)T}}(\mathbf{W^{(1)}x} + \mathbf{b^{(1)}}) + b^{(2)} \tag{2.9}$$

Where $\mathbf{W^{(1)}}$ is the matrix of weights of the first layer, $\mathbf{x}$ is a vector of inputs, $\mathbf{b^{(1)}}$ is the vector of biases in the first layer, $\mathbf{w^{(2)}}$ is the vector of weights in the second layer and $b^{(2)}$ is value of bias in second layer.

Graphics processing units are made to do matrix operations. We can use this property of GPUs to compute the output of neural networks extremely fast. Neural Networks can have any number of layers. The layers between the first and last layers are called hidden layers. The networks which have a large number of layers are called deep neural networks. As the number of layers and neurons in each layer increases, the time to train the network also increases.

Backpropagation: Backpropagation is the process of updating the neural network weights starting from the last layer, i.e., the output layer, to the first layer. Backpropagation uses Leibniz's chain rule and gradient descent. Chain rule in calculus is the way to find the derivative of composite functions. A composite function is just a function of functions. For example, the neural network's output in figure 2.5 is a composite function. An example of a composite function is:

$$h(x) = f(g(x)) \tag{2.10}$$

and its derivative is:

$$h'(x) = f'(g(x))g'(x) \tag{2.11}$$

and in Leibniz's notation, the functions will be as follows:

$$y = g(x) \tag{2.12}$$

$$h = f(y) \tag{2.13}$$

and the derivative of h with respect to x is:

$$\frac{dh}{dx} = \frac{dh}{dy}\frac{dy}{dx} \tag{2.14}$$

Similarly, the whole neural network can be decomposed in smaller equations, like equation 2.10 is decomposed in equation 2.12 and 2.13. After decomposing the equations, we find the gradient of the loss with respect to each weight using the chain rule. We can use the same loss function in equation 2.3 for our two-layer network in figure 2.5. As we

calculate the loss for each single input, we can modify the equation to calculate squared error.

$$L = (y - \hat{y})^2 \tag{2.15}$$

Where y is the ground truth. The equation 2.9 can be decomposed into two equations.

$$\hat{y} = \mathbf{w^{(2)T}h} + b^{(2)} \quad \text{and} \quad \mathbf{h} = \mathbf{W^{(1)}x} + \mathbf{b^{(1)}}$$

We can update the weights and biases using equation 2.4. For that, we need the gradient of loss with respect to each weight and bias.

$$\frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y}) \quad \text{and} \quad \frac{\partial \hat{y}}{\partial h} = \mathbf{w^{(2)}}$$

$$\frac{\partial \hat{y}}{\partial \mathbf{w^{(2)}}} = \mathbf{h} \quad \text{and} \quad \frac{\partial \hat{y}}{\partial b^{(2)}} = 1$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{W^{(1)}}} = (\mathbf{x}, \mathbf{x}) \quad \text{and} \quad \frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \mathbf{W^{(1)}} \quad \text{and} \quad \frac{\partial \mathbf{h}}{\partial \mathbf{b^{(1)}}} = \mathbf{I_2}$$

The partial derivatives will be:

$$\frac{\partial L}{\partial \mathbf{w^{(2)}}} = \frac{\partial \hat{y}}{\partial \mathbf{w^{(2)}}} \frac{\partial L}{\partial \hat{y}} = -2\mathbf{h}(y - \hat{y}) \quad \text{and} \quad \frac{\partial L}{\partial b^{(2)}} = \frac{\partial \hat{y}}{\partial b^{(2)}} \frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y})$$

Similarly, for the parameters of the first layer, we will have

$$\frac{\partial L}{\partial \mathbf{W^{(1)}}} = \frac{\partial \mathbf{h}}{\partial \mathbf{W^{(1)}}} \frac{\partial \hat{y}}{\partial \mathbf{h}} \frac{\partial L}{\partial \hat{y}} = -2\mathbf{x} \otimes \mathbf{w^{(2)}}(y - \hat{y})$$

$$\frac{\partial L}{\partial \mathbf{W^{(1)}}} = \frac{\partial \mathbf{h}}{\partial \mathbf{b^{(1)}}} \frac{\partial \hat{y}}{\partial \mathbf{h}} \frac{\partial L}{\partial \hat{y}} = -2\mathbf{I_2}\mathbf{w^{(2)}}(y - \hat{y})$$

Using the gradients, we can update the parameters of the neural network. We can train the whole model using the same iterative algorithm as in 1.

One optimization technique is to calculate the loss for all input examples and then update parameters using that loss, this is called gradient descent. On the other hand, If parameters are updated for one input example or randomly picked multiple examples (i.e., in batches), then it is called stochastic gradient descent (SGD) [11].

The choice of learning rate α in equation 2.4 is important. Usually, a small value that is less than 1 is chosen. The learning rate decides how big the change should be in the values of parameters. A small learning rate gives a smooth transition to minimum loss

but can take more iterations. Conversely, a big learning rate will cause unstable learning and possibly miss the optimal values because of large jumps in the value of parameters. Adam optimizer [11] is another famous optimizer with an adaptive learning rate for each model parameter. The learning rate of parameters with large gradients is kept smaller, and of parameters with small gradients is kept relatively larger.

The neural network shown in figure 2.5 has only two layers, But a network can have many layers, and each layer can have a large or small number of neurons. This type of fully connected network is called a multi-layer perceptron. These networks or models are often called universal approximators as, in theory, they can approximate any continuous function given enough number of neurons and layers in them. If the problem at hand is nonlinear in nature, then adding activation layers to the neural network makes it easy for the model to converge.

Activation functions: Activation functions are nonlinear functions applied to the output of each neuron of a layer. Many activation functions are used for particular problems, but some are widely used. One of them is Rectified Linear Unit (ReLU) [9]. It turns any value to zero that is less than zero. Its function is:

$$ReLU(x) = max(0, x) \tag{2.16}$$

Another famous activation function is called the Sigmoid activation function [30]. Its output is always between 1 and 0 and follows an S-shaped curve. Its function is:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.17}$$

Softmax is another widely used activation function, especially for classification problems. It is applied to the last layer and gives the probabilities of each class. Its output and input is a vector. Its formula is:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.18}$$

Convolution Neural Networks

Convolution neural networks (CNN) are a special kind of neural network that learns useful features using convolution with some kernels. The values of the kernels are learned through the same process of backpropagation as discussed in the previous section. Kunihiko Fukushima first introduced them in 1980 [10]. He was inspired by the explanation of the working of visual cortices of cats by Hubel et al. [14] in 1962. Fukushima introduced the convolution layers and down-sampling layers in his model and showed that it

was able to recognize distorted images of digits. In 1989, Yann Le Cunn trained a CNN, which was able to classify handwritten zip codes. Their model had 2 convolution layers, each followed by a down-sampling layer.

Convolution is integral of two functions in which one function is inverted around the y-axis and shifted. Let's suppose two time-dependent functions, f and g. The convolution denoted by $*$ is then:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2.19}$$

Whereas τ is a dummy variable. The offset of t is added to g, and then it is slid along the τ-axis. At time t, the convolution value is the area under the overlapping region of two functions. The above equation is for the one-dimensional variable. For two-dimensional variables, the equation is:

$$(f * g)(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x', y')g(x - x', y - y')dx'dy' \tag{2.20}$$

And in discrete form:

$$(f * g)[i, j] = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} f[m, n]g[i - m, j - n] \tag{2.21}$$

While performing convolution, one of the functions is called kernel. For example, Figure 2.6 shows a 2D kernel in the middle. The kernel is of the same dimension as the input data.

In Figure 2.6, we can see that the resulting image only has edges in it. With different kernels, we can extract different features of the image. To have a machine learning model to do a task on the image or data, we will have to choose and design kernels, which, in return, give us useful features of the data. But in CNN, the network will learn a number of different kernels that are most useful to the given data set.

The convolution in 2.21 has a step of 1, i.e., the amount by which the convolutional kernel is moved across the input data when performing the convolution operation. It is also possible to have a step of more than 1, and this step number is called stride. The stride controls the spacing between the locations where the kernel is applied to the input. The stride value affects the size of the output feature map produced by the convolutional layer. Specifically, it determines how many positions the kernel "jumps" as it slides over the input. A larger stride leads to a spatially down-sampled output as the kernel skips over more elements in the input. We can also add zeros on all sides of the data to avoid this downsampling. This process is called zero-padding.

In CNN layers, convolution is not performed in an iterative manner. Instead, we have that many neurons required to produce output image simultaneously. The values of the

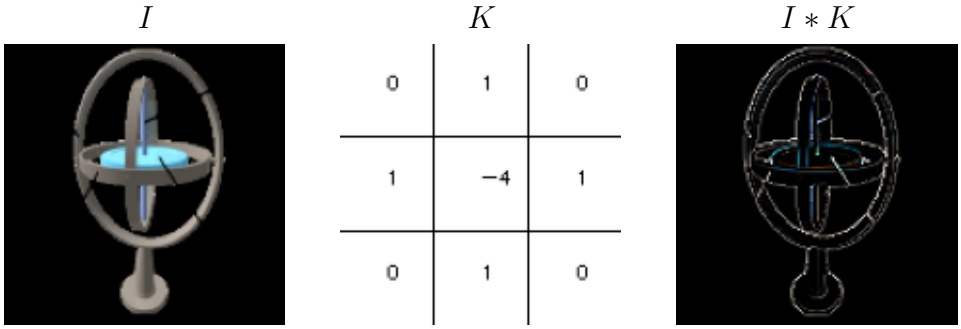


Figure 2.6: Convolution of an image of gyroscope at left with Laplacian 3×3 kernel in middle results in an image with edges only on the right side.

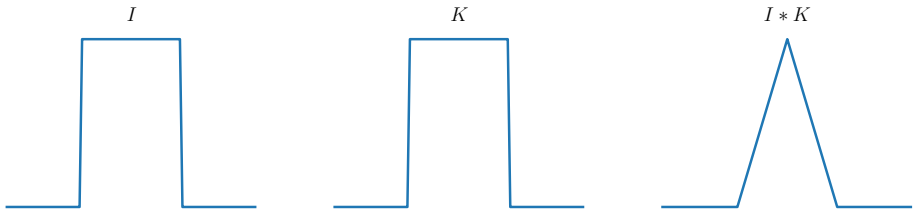


Figure 2.7: When two square waves are convoluted with each other, the result is a triangle-shaped wave. The signal k is inverted and then slid over the I. The resulting wave is the measure of the area under overlapping signals. As the inverted K moves over the I, the area will increase and reach its peak value when both overlap completely.

kernel are the same for all neurons. This is called weight sharing and is an important property of CNN. Weight sharing makes the CNNs translation invariant. That means a kernel will learn the same feature for every part of the image.

Instead of learning just one kernel in a single layer, CNNs learn a lot of kernels. All these kernels will be initialized randomly and will learn some different features. The number of kernels in a layer equals the number of output channels. It is important to note that if we have K number of kernels in a single layer, then the layer's output will be K convoluted images. For example, color images have 3 channels (red, green, blue). A layer with 8 kernels will produce an image with 8 channels. It is important to note that all 8 kernels in this case will be 3D kernels.

Lets assume we have image of size $m \times n$, and kernel of size k, stride s and number of padded zero rows and columns p then output image $i \times j$ will have size:

$$i = \frac{m + 2p - k}{s} + 1$$

$$j = \frac{n + 2p - k}{s} + 1$$

The number of neurons in the layer will be $kkij$. But trainable parameters will only be k^2 because of shared weights. And for K number of kernels, the output tensor will have a $k \times i \times j$ shape.

Another important technique employed in CNNs is downsampling. It is mostly applied to the output of a convolution layer. It reduces the spatial dimension of features and, by doing so, helps the network to focus on learning more abstract features in subsequent convolution layers. Also, by reducing the resolution of image networks, try to generalize the data set and avoid overfitting. The downsampling is achieved by dividing the image into grid-like regions and finding one value for each region. The down-sampling layer is commonly called the pooling layer and has many types. For example in Maximum pooling, the pooling layer will output the maximum value in a region of the image. In average pooling, the output will be the average of all values in a region of the image.

2.4 Literature Review of step detection methods

Research in monitoring and analyzing walks, also known as gait analysis, started in the 20th century. Early researchers used different kinds of mechanical and electrical apparatuses to record steps. In the 1960s and 1970s, accelerometers were also used in the research, and these sensors were large in size compared to modern sensors. Since the introduction of electronic watches and mobile phones, which had electronic accelerometers in them, more and more methods and algorithms have been introduced. We can

Figure 2.8: A maximum pooling layer returns the maximum value in the specified grid. A 6×6 tensor is reduced to just 2×2 tensor containing only maximum values in the corresponding color-coded 3×3 grid.

divide different methods into three categories: deterministic methods, also known as heuristics, methods that include machine learning models, and methods involving deep learning models. This section will discuss some methods and data sets used in a few research articles.

Heuristic Using a deterministic algorithm is the most proposed method in the literature. All the methods use the detection of peaks and choosing different threshold values in one way or another. Some methods use adaptive thresholds whose values are updated with changing signal statistics.

[19] uses step average and step deviation to filter out the peaks and valleys in the signal. They also used adaptive time thresholds to counter the varying pace of a walk. In their experimental setup, they had 30 people walk with different poses of smartphones on their bodies, such as calling, texting, or in their pockets. They also asked people to walk with different modes, such as running and free walking. In my opinion, Their data set has the most diverse walk and mobile phone positions.

[6] detects peaks in the opening angle of the human leg. They first represent sensor data in terms of the opening angle of the leg and then determine the peaks in those angles and predict the steps. In their case, sensors were only placed inside pockets. There were 18 people walking at different speeds on the running machine. This algorithm has a disadvantage because it has several different parameters that must be tuned for each user.

[13] tries to estimate step length first by smoothing the acceleration signal using the Fast Fourier transform and then using a set of predefined rules. The users had the mobile phone in their hands while they walked four different distances at different speeds. Their

algorithm predicts the total distance walked by the user.

Previously mentioned works [19][6] measure the accuracy of their methods by calculating the ratio of the total number of steps estimated to the actual number of steps. While this gives us an overall idea of how good the method is, we can not be sure if the step is detected at the exact time. [13] gives the accuracy by comparing the distance estimate of their method versus the actual distance. This also poses the same problem as mentioned before. Most of the deterministic methods focus on counting the total number of steps. They can estimate the timestamp of the step when the counter increases its value.

Machine Learning Methods that involve machine learning models require researchers to look into different features of the accelerometer and/or gyroscope signal. After picking the most useful features, a model is trained to predict steps. [21] uses a decision tree to count the total number of steps. After dividing the completer signal into smaller segments, they extracted features like average acceleration, location of maximum acceleration, and minimum value of acceleration and gyroscope in the segment. 4 people walked a different number of steps on the ground and stairs. The data was collected from smartwatches worn on the wrist.

[2] also used a decision tree (Light gradient boosting decision tree [17], which is a special kind of gradient boosting tree [8]) but looking at a lot more features. These include statistical features (mean, median, skewness), peak characteristics (peak height), and different frequency-based features. 8 people were asked to walk in different scenarios with sensors in their hands in different positions. In this article, they also looked for the accuracy of each step detected, i.e., they compared the timestamp of every detected step to its actual timestamp, and it only contributed to the accuracy if the difference was less than 0.6 seconds.

Even though deterministic and machine learning methods can achieve good results, they need to be tuned for different scenarios of walking or the position of sensors, especially for different people. Machine learning models usually need human-picked features to be trained on. They also might not work for the same reasons. On the other hand, neural networks are good at generalizing and learning the important features themselves. We can train a neural network to work for different people, different speeds of walk, and different positions of sensors on the human body by just giving the network a data set with all those scenarios.

Deep Learning With the rise in popularity of neural networks in many fields, a lot of researchers are now using them for step detection and step length estimation. [23] uses LSTM and CNN to predict if the segment of signal belongs to the right or left foot. And with the change of prediction from left to right or right to left, they predict the

instant of a step. To check the performance of their method, they calculate accuracy based on the ratio of the total number of steps predicted to the actual number of steps. The problem with this approach is that it only works if the user continuously walks and does not stand even for a short period of time.

[20] uses CNNs to count the number of steps. It counts the total steps in a segment of the signal, and the timestamp is estimated based on when the counter increases its value as the next segment of the signal is taken in. They recorded data from a total of 30 participants who were wearing sensors on the ankle, wrist, and hip. Also, the participants walked normally in a loop and did some activities to record irregular gaits. The performance was measured by calculating the accuracy of counted steps and also comparing the predicted and actual timestamps of the steps.

[33] uses LSTM to detect timestamps of the start and end of steps directly. They use a low-pass filter to remove high-frequency noise in data. After dividing the signal into smaller segments, they pass it through the LSTM network to predict, for each timestamp, if it is a start (and for the end separately) of a step or intermediate timestamp. As the network can miss a step at its start or end and can predict starts (or ends) very close to each other, they developed a comprehensive post-processing algorithm to filter out errors made by the model. This post-processing greatly increases their accuracy. In their experimental setup, users only hold the sensor in their hand.

I will use the method in [33] that uses LSTM to predict for each timestamp, but instead of LSTM, I will use CNN. I will also implement another method (section 4.3) of directly predicting indices of the start and end of steps. I will compare the results of my methods with [33] but without any of their post-processing. The data set I used is more comprehensive than [33] [20] as it includes sensors in hand, left, right, right, and back pockets. Users also walked up and down stairs along with regular walks and pauses.

3 Step Dataset

This chapter discusses the data set that was used to train the neural networks explained in the next chapter. First I will discuss how the data was collected and which sensor was used. Then, I will discuss the pre-processing of the data. The use of images and plots will aid the explanation.

3.1 Acquisition

Sensors Most mobile phones have a built-in inertial measurement unit or IMU. An IMU typically has one accelerometer and one gyroscope for each axis of a three-dimensional Cartesian plane. Combining the information from all these sensors, mobile applications can estimate the orientation and speed of the mobile phone. Some mobile phones use high-quality sensors that are more precise, and some have less precise sensors. Also, different mobile phones have different frequencies.

Mobile phones were not used to collect the data set as it is hard to synchronize different mobile phones placed on different body positions. Instead, external miniature IMU sensors made by Xsens were used. The name of the complete hardware is MTw Awinda [1] [26], and it consists of 6 wireless motion tracker dongles with 1 receiving dongle. These motion trackers have 5 sensors: a gyroscope, accelerometer, magnetometer, ther-

Figure 3.1: One of the six dongles in orange with the receiver dongle used to collect accelerometer and gyroscope data [1] © 2023 by Movella Inc.

Figure 3.2: Six wireless dongles attached to different body positions. The number 1 is in hand, 2 and 3 in right and left pocket, 4 in the back pocket, and 5 and 6 on ankles.

mometer, and barometer. A magnetometer measures the strength and direction of the surrounding magnetic field. All 6 dongles are synchronized, making it easy to record the readings from different body positions of a single person's walk simultaneously.

The wireless sensor dongle is of size $47mm \times 30mm \times 13mm$ and weighs only $16g$ [26]. This makes it easy to place the sensors on any body part using straps. For the collection of this data set, 1 sensor was kept in the right hand, 2 sensors were in the right and left pocket each, 1 sensor was in the back pocket, and 2 sensors were strapped to the ankle of the left and right foot. This setup can be seen in Figure 3.2.

Collection The complete data set consists of 32 different people walking on different paths for different amounts of duration. Participants were asked to walk naturally without any given directions. They had to walk on flat ground, up and downstairs. They were also asked to take pauses during the recording of the data by standing still. Different people walked on a number of different floor coverings like cement, wood, stone, and carpet. The sampling rate or frequency with which data is recorded is 80Hz.

The sensors on the feet were used to annotate the data from other sensors. These sensors are mounted on feet, so their magnitude change is highest and have clear peaks to indicate the foot touching the ground or having maximum acceleration. Indexes of steps were found using a heuristic method discussed in.

For each person, two CSV files are created. One has all the indexes for start and end in two columns. The other contains all the readings from all six sensors. Each sensor

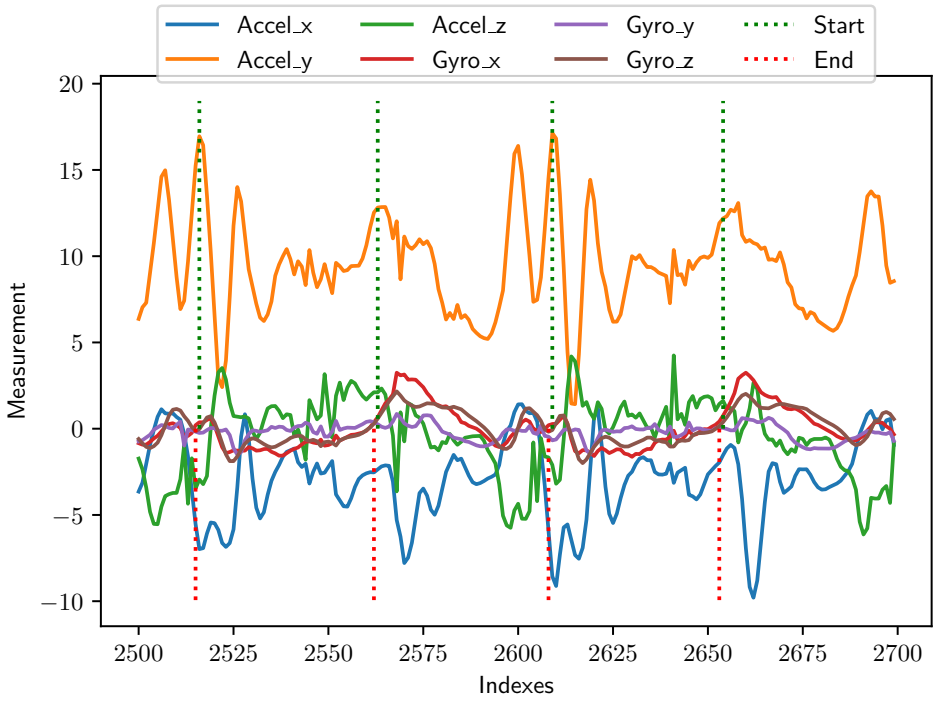


Figure 3.3: Plot of the left pocket sensor reading. Dotted green and red vertical lines indicate the start and end of steps. We can see that some x-axis are varying more in magnitude than others.

and axis has readings for acceleration, free acceleration, gyroscope, magnetometer, and rotation quaternion. For this topic, only acceleration and gyroscope data are used. Along with sensor values, we also have a column for timestamp and activity. The activity column indicates if a person is walking up or down stairs, walking or standing still (sometimes in the lift going down and up).

3.2 Pre-processing

Before any training, we need to process data to make it ready to be fed to models. This includes dividing the whole time series into smaller chunks called windows, low-pass filtering, and normalization.

Windows Unlike images, which are fixed-size matrices of pixel values, time series data has no fixed length similar to text. One data recording can be three minutes long, while the other can be just a few seconds long. But neural networks require fixed-size input that is predefined when they are created. To accommodate this, the whole time series

Data						Labels	
AccelX_3	AccelY_3	AccelZ_3	GyroX_3	GyroY_3	GyroZ_3	Start	End
0.677426	8.787494	-3.721742	0.206037	0.230636	1.145878	0.0	0.0
0.520967	6.932050	-2.928719	0.044333	0.407333	1.053624	0.0	0.0
-0.279782	7.415740	-0.943098	-0.471475	-0.983564	0.720878	0.0	0.0
-1.597696	9.674706	-4.350395	-0.417156	-0.046980	0.458047	0.0	0.0
-3.588921	12.408151	-1.714932	-0.245122	-0.069897	0.096609	0.0	0.0
-5.526516	15.234378	-3.466302	0.201103	-0.174418	-0.142280	0.0	**1.0**
-6.979936	16.951621	-2.946325	0.372922	-0.246831	-0.027834	**1.0**	0.0
-6.924718	16.437142	-3.214837	0.366545	-0.226820	0.304058	0.0	0.0
-6.078837	13.511440	-2.746450	0.508280	-0.081100	0.651148	0.0	0.0
-5.437353	10.255535	0.027262	0.612580	0.043835	0.725954	0.0	0.0

Table 3.1: A snippet of data with corresponding labels. Readings only from the sensor in the left pocket are shown here. 1 in the start column indicates that a step starts on this index or timestamp. 1 in the end column indicates that a step ends here.

is divided into smaller chunks, also known as windows, which are the same size as the input of neural networks.

The size of a window means how many timestamps are in one window. It depends on the specific problem and is usually experimented with different values to see which yields the best results. A smaller window will preserve more detail in the data, while a big window will give a smoother representation of data as it will cancel out small fluctuations. In other terms, smaller window sizes provide high temporal resolution to the network. A smaller window size should be chosen for events occurring in small intervals. For our problem, we should choose a window size that at least captures one step. The delay between when a step happens and when it should be reported should be considered for the upper limit of the window size.

Another value to be considered when creating windows is the step size. Step size means how many timestamps we move forward to create one window. For example, if the step size equals the window size, then windows do not have any overlapping data. Another example will be to have a step size of 1. This means we jump to the next value by 1 timestamp. In this case, the second window will have all data points of the first window except the initial data point plus one data point ahead.

A smaller step size like 1 helps capture more variations of time series. This makes the model more sensitive to the smaller variations. However, a smaller step size increases the computation cost significantly. A step size of 1 means having the same amount of windows as the total number of data points, which will always consume more computer

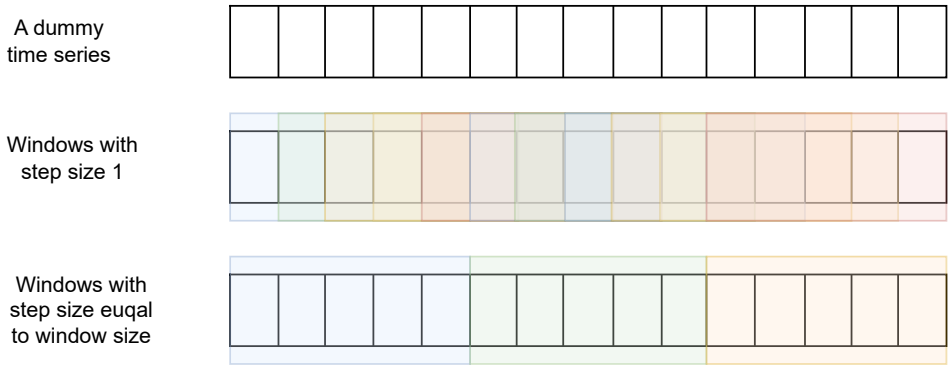


Figure 3.4: A dummy time series divided into smaller chunks with different step sizes. A step size smaller than the window size results in overlapping windows, and a large number of windows are created. On the other hand, step size that is equal to or greater than window size results in fewer windows, only 3 in this case.

memory. A large step size will result in faster learning as we will have fewer windows but probably less efficient.

The total number of windows created can be calculated with the following equation.

$$Total\_Windows = \lceil \frac{N - w}{s} \rceil + 1$$

w is window size, s is step size, and N is sequence length. We use the ceiling function to round the value to the nearest integer if the result of the division is a fraction.

Creating windows in this manner ensures that the time series is not altered. Each window will still have data points that come right after each other, and windows are never made by randomly picking data points from series. It is important to note that after windows have been created, they can be randomly shuffled along with their labels as they are now independent.

Normalization Changing values of different features to the same scale is called normalization. Usually, each data point in a data set has more than one feature. For example, in the step detection data set, we have 3 values for acceleration and 3 for gyroscope, 1 for each axis, making the number of features equal to 6. Now, each axis value can have a different range. It may happen that z-axis values range from -10 to $+10$ while x-axis values only change between -2 to $+2$. This will cause a problem as the z-axis might influence training more than the x-axis, even though the x-axis may have more information. The solution will be to bring both axes in the same range while keeping the structure.

There are various methods of normalization of a data set. The choice of a method depends on the type of machine learning model to be used and the kind of data set to be normalized. The two most used techniques of normalization are standardization and min-max scaling.

Standardization or z-score standardization is used when data does not have so many outliers. It makes the mean μ of data equal to 0 and standard deviation σ equals to 1 and can be computed as:

$$x' = \frac{x - \mu}{\sigma}$$

whereas

$$\mu = \frac{1}{N}\sum_{i=i}^{N} x_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \mu)^2}$$

Min-max scaling converts the values into a specific range, usually between 0 and 1. This method should be used when maximum and minimum values of data are known, and the data is uniformly distributed. For example, data generated from sensors will have a specific range given by the manufacturer. Data can be scaled by following the formula.

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.1}$$

x_{min} is the minimum value and x_{max} is maximum value in the data.

Filtering Filtering is removing some frequencies from the signal. In context to our data, it is useful to remove high-frequency noise, such as sensor noise or vibrations, while allowing lower frequencies as they are more meaningful, i.e., we are just interested in steps, but many other peaks in a shorter time span would be recorded because of other body movements.

Signals can also be represented in the frequency domain, i.e., amplitude vs frequency instead of amplitude vs time. To change into the frequency domain, we use the Fourier transform. Fourier Transform is a method of breaking mixed signals into sine and cosine waves of different frequencies, representing the frequencies present in the signal. Fourier transform can be applied to continuous time signals and discrete-time signals. The equation of the discrete-time Fourier transform is:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

where k is integer index representing frequencies from 0 to $N-1$, N is number of samples

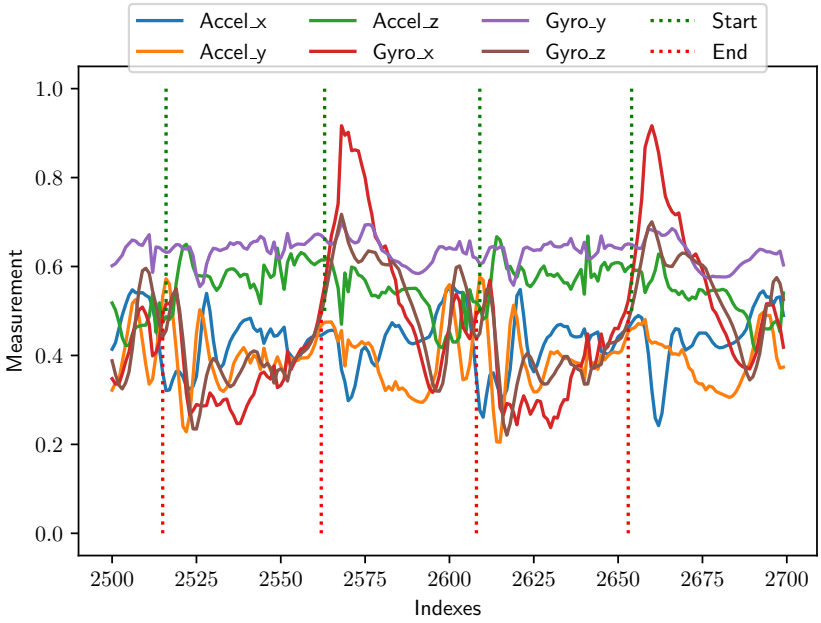


Figure 3.5: Same part of the data as in Figure 3.3 but normalized using min max scaling as in equation 3.1.

in time domain.

After Converting the signal into the frequency domain, we can create filters that can remove frequencies. A low pass filter will allow low frequency while blocking higher frequencies in the signal. A high pass filter will allow only higher frequencies. We also have bandwidth filters that allow frequencies in certain ranges. There are different filters available, which have their own benefits and problems. We will use third-order Butterworth [4] low pass filter. The cutoff frequency used is $3Hz$, same as in [33]. Figure 3.6 shows the smooth signal after filtering. Figure3.7 is the data after filtering and normalization.

Magnitude as Feature Calculating the Euclidean norm of three axes of each sensor and using it as an additional feature along with the axis data has been demonstrated to be helpful [33] [23]. Heuristic methods like [19] usually rely on the magnitude values to detect peaks. It can make it easy for neural networks by utilizing the extra feature instead of learning to calculate it independently. We can calculate magnitude using the equation 3.2.

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{3.2}$$

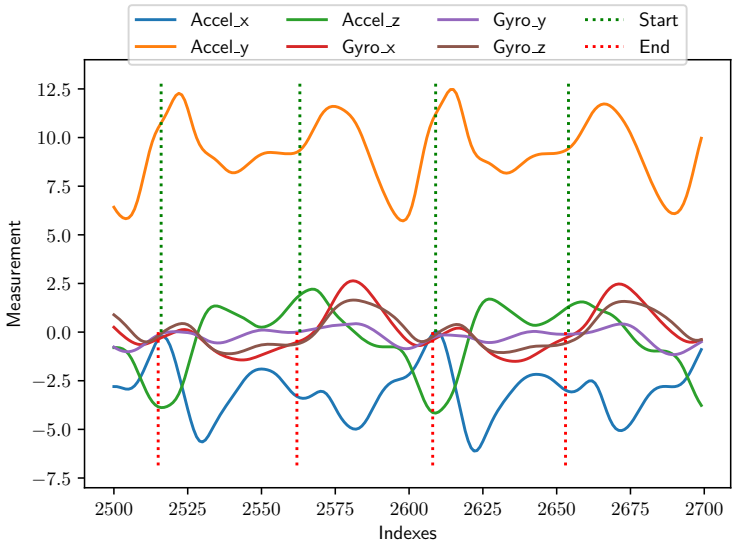


Figure 3.6: Same part of the data as in Figure 3.3 after filtered with 3rd order Butterworth filter with cut off frequency of 3Hz.

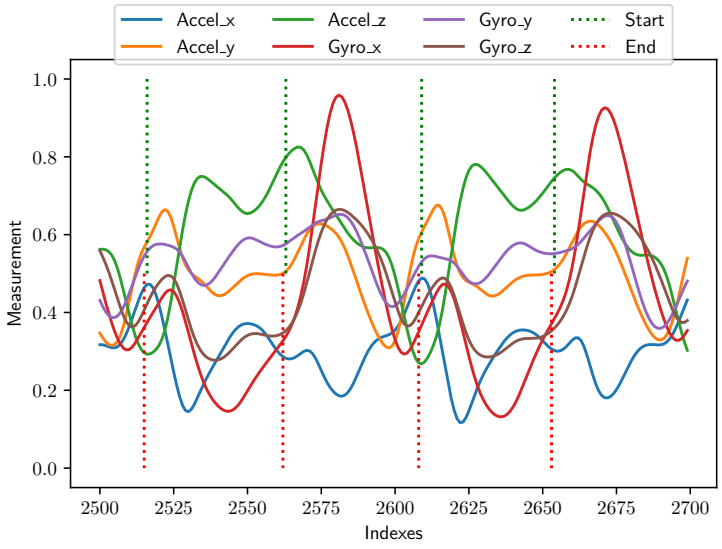


Figure 3.7: Same part of the data as in Figure 3.3 but filtered and normalized using min max scaling using equation 3.1.

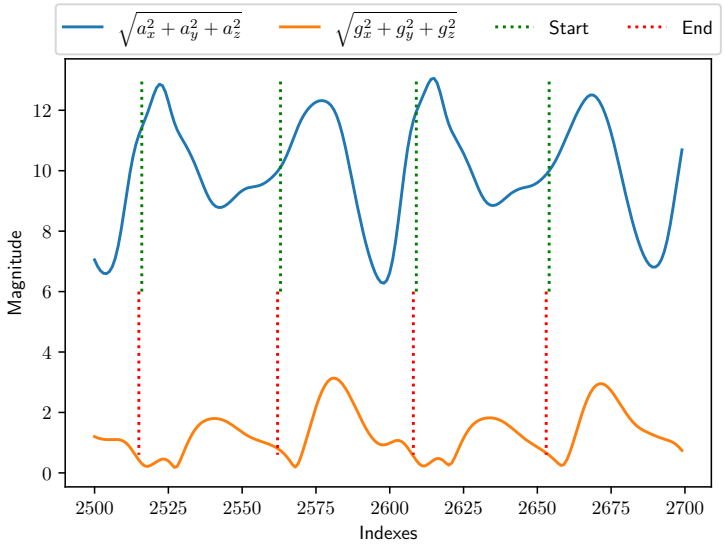


Figure 3.8: Euclidean norm of accelerometer and gyroscope after filtering with Butterworth low pass filter. The start and end of steps make more sense now, as a start can be seen before every peak.

3.3 Steps Statistics

Looking at the overall statistics of the steps of all participants can help in deciding window size and some threshold values required for post-processing techniques.

I use data from 12 participants, of which 3 are used for testing the models. Training data has 12856 steps, and testing data has 2117 steps. The mean step time is $\tilde{5}6.17$ seconds. The biggest step someone took was 1.32 seconds, and the smallest step was 0.25 seconds. These times are not about one gait cycle. A step is one movement made by one foot, and the gait cycle is 2 consecutive steps of both feet.

We can see from Figure 3.9 that most of the steps range from 20 to 70 samples. This means if the window size is 100, then we will be able to have one complete step in it. A small window size like 20 or 40 would mean the model mostly does not see a complete step. We can also assume that any predicted step smaller than 17 is not a step but rather a wrong prediction. Similarly, any predicted step larger than 110 is also not a step. These thresholds can help us refine the predictions of the models.

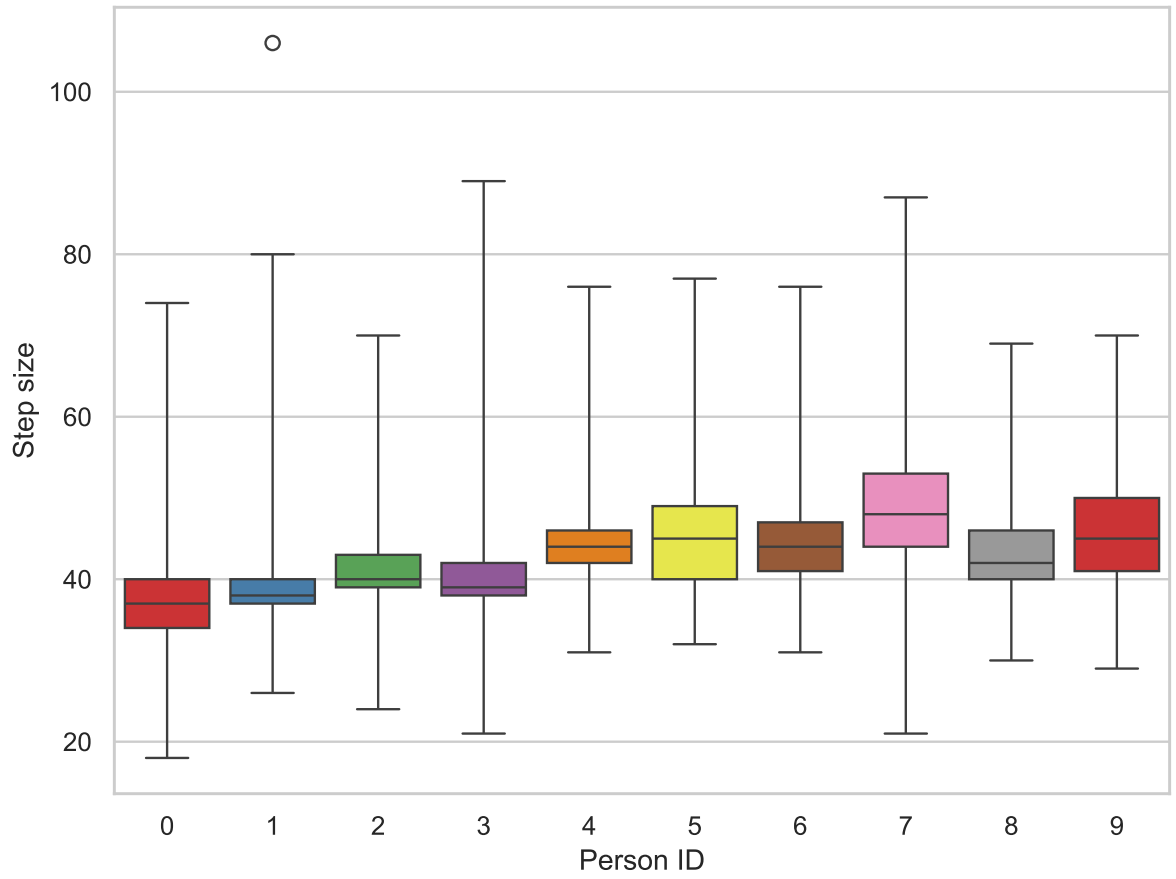


Figure 3.9: Boxplot of step sizes in samples of 10 different people. The middle line in the box is the median, and the box represents the central 50% of the data. The circles are outliers. The extends from the boxes represent the whiskers with a range of 15.

4 Models

This chapter describes the methods to predict steps within a window created in section 3.2. First, I discuss the method employed by [33], an LSTM network with comprehensive post-processing. Second, I discuss my proposed method of replacing LSTM with CNN in [33]. Lastly, I will propose a YOLO [29] like CNN architecture, which directly produces the indexes for the stop and end of a step.

4.1 LSTM - A reference method

Their paper "Deep-Learning-Based Step Detection and Step Length Estimation With a Handheld IMU" used LSTM neural networks with the pre-processing discussed in section 3.2. They also relaxed the labels to ease the training of the network. After producing the predictions for all the data, they use a post-processing method to purify the indexes of the start and end of a step.

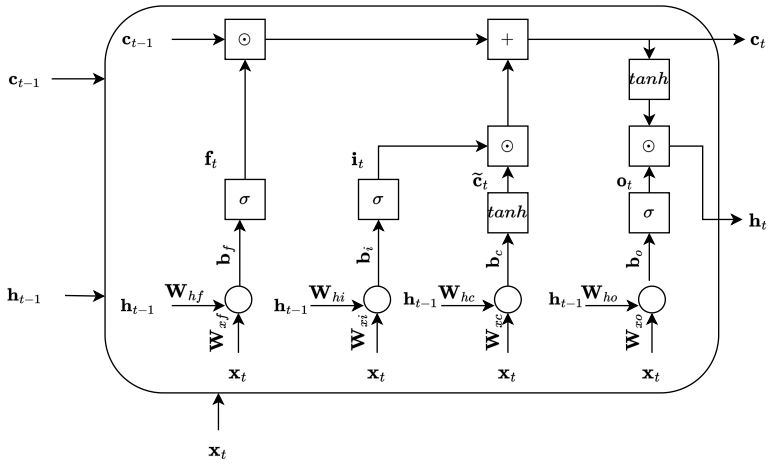


Figure 4.1: LSTM unit which has a forget gate $\mathbf{f_t}$, an input gate $\mathbf{i_t}$, and an output gate $\mathbf{o_t}$. $\mathbf{W}$s and $\mathbf{b}$s are the trainable parameters of this unit. $t-1$ indicates the previous output of this unit, and t means the current input and outputs. Parameters are shared across the time in LSTM.

An LSTM neural network consists of layers of LSTM units, which consist of a cell that remembers values over time and gates that control the flow of information in and out of the cell. Figure 4.1 illustrates an LSTM unit and its equation is:

$$\textbf{Input gate:} \quad \mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i)$$

$$\textbf{Forget gate:} \quad \mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f)$$

$$\textbf{Output gate:} \quad \mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o)$$

$$\textbf{Candidate memory:} \quad \widetilde{\mathbf{c}}_t = tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c)$$

$$\textbf{Memory cell:} \quad \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t_1} + \mathbf{i}_t \cdot \widetilde{\mathbf{c}}_t$$

$$\textbf{Hidden state:} \quad \mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$$

Where $\mathbf{x}_t \in \mathbb{R}^d$ is the current input, $\mathbf{h}_t$ and $\mathbf{h}_{t-1} \in (-1, 1)^h$ is the current and last hidden state, $\mathbf{b} \in \mathbb{R}^h$ is bias, $\mathbf{W}_x \in \mathbb{R}^{h \times d}$ and $\mathbf{W}_h \in \mathbb{R}^{h \times h}$ are weights. Here h is the number of hidden units, and d is the number of input features. $\odot$ represents the element-wise product, and σ is the Sigmoid function described in equation 2.17.

For each time step, the input (X_t) is fed into the cell, and its memory cell state (C_t) and hidden state (H_t) are updated. After the loss calculation, weights and biases are updated like other architects using some optimization technique such as gradient descent.

A layer of LSTM units can be formed by having multiple LSTM units in parallel, taking the same input but having different initial values of weights and biases. These units will learn different features in the input. A network can consist of one or more layers stacked together. The output of one layer (one or more C_t depending on the number of LSTM units in a layer) is concatenated and used as input for the next layer.

[33] uses 2 of these layers stacked together. The output of the second layer is passed through a fully connected layer followed by a sigmoid activation function (Equation 2.17). After each LSTM layer, a dropout layer ensures the model does not overfit. Figure 4.2 illustrates the network unfolded in time. They used two LSTM layers, each with a dropout layer and one fully connected layer, responsible for producing output sequences for both the start and end of the layer. The input to the network consists only of accelerometer data with an additional feature of magnitude measured for each timestamp.

Relaxed Labels The labels in the table 3.1 occur on exactly one timestamp, which is the exact time when a step started or ended. But in reality, it is also acceptable that the network predicts the starts and ends slightly before or after the actual instant. Also, having only one timestamp where the model needs to predict makes it hard for the network to train. For these reasons, it is suitable to relax the labels by some timestamps before and after, that is, adding ones around the true one. They used this in [33] as well

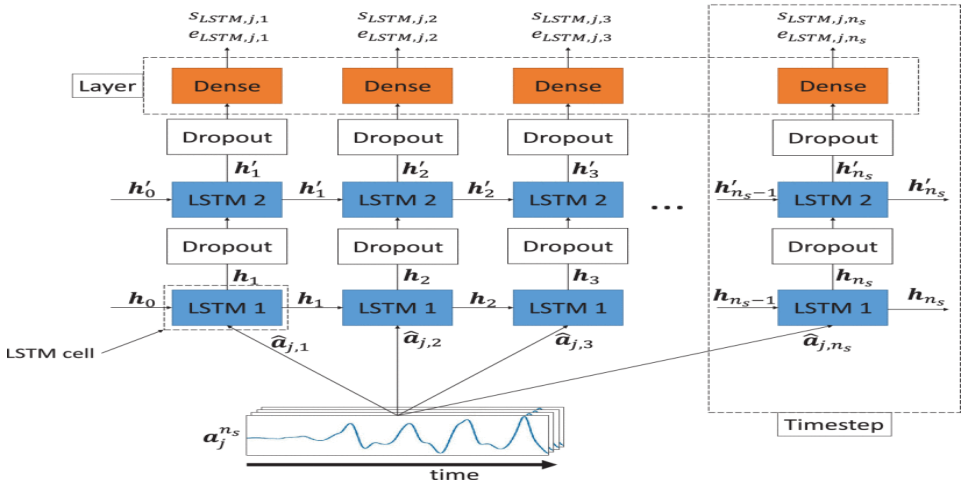


Figure 4.2: LSTM architecture used in [33] unfolded for few time steps. It has two LSTM layers with one fully connected layer at the end. The dense layer produces combined output for the start and end sequence. © 2023 IEEE.

Start	0.0	0.0	0.0	0.0	1.0	1.0	1.0	**1.0**	1.0	1.0	1.0
End	0.0	0.0	0.0	1.0	1.0	1.0	**1.0**	1.0	1.0	1.0	0.0

Table 4.1: An example of relaxing labels where 3 timestamps before and after the actual instant are changed to 1. These are the same labels as in 3.1. The true timestamp is shown in bold.

by adding ten 1s before and after the true timestamp of start and end.

Hyper-parameters They experimented with different numbers of hyper-parameters. They tested the model for different window sizes, of which the minimum size was 100. They also tried different numbers of LSTM units in the network. Also, the dropout value was also changed to find the optimal network. The optimal value for the window size is 200, 400 for the number of LSTM cells, and 0 or 0.2 for the dropout rate.

Post processing Because the network is trained to produce output for each timestamp in a window, its output can have, in the start and the end sequence, 1s, which are very close to each other but are not right next to each other. It is also possible that the network produces start and end sequences with a relatively large time in between, so large that the predicted step becomes more than a possible human step. To remove these problems from the output, [33] uses a heuristic algorithm with a number of different thresholds. For example, one procedure removes short bursts of 1 from the output. Another adds or deletes start and end indexes from the output that are not in pair. Only after this post-processing did they calculate the accuracy score of steps by comparing

where each step is occurring. This also has a threshold that decides if the predicted step is in close range to the actual step. If yes, it still contributes to the accuracy.

The loss function used by [33], and I for the CNN is cross entropy loss and will be discussed in the next section.

4.2 CNN based model

Convolution neural networks excel at recognizing patterns in the data. They are also translation invariance, which means they can detect those patterns regardless of their position in the data. For example, in the case of our accelerometer data, CNNs should be able to predict a step regardless of when it occurs in the input window. Another benefit of CNN's translation invariance property is that it can adapt easily to different human gaits and sensor body positions. RNNs, on the other hand, are not inherently translation invariant, even though they can be trained to capture spatial invariance. Therefore, I believe CNN can be more suitable for the step detection task, as we need our model to handle not only different styles of walks but also different persons and different body positions of the sensor.

A usual CNN model has some blocks of convolution followed by a fully connected layer with different activation functions. A block consists of a convolution layer, a pooling layer, and an activation layer. Some deep learning models [32] have consecutive convolution layers without any pooling layer to capture fine-grain details. I will test different models with different combinations of layers of different types with different values of hyperparameters.

The input and output of a model are decided before it can be trained. In this case, the input is a window of acceleration and gyroscope data. Since neural networks are not trained on just one example, multiple randomly chosen windows will be used for the input. I will experiment with different size windows denoted by W_s. The models' output will be the same length as windows for the steps' start and end. This means that we will have two sequences for the output, one for the start and one for the end, with a length equal to the window size.

I used Pytorch [25] framework to create, train, and test my models. The one-dimensional CNN layer requires the input tensor to be in the format of a batch, number of channels, and signal length (N, C_{in}, L). Batch size is the number of windows processed together in one forward or backward pass. The number of channels will be the number of features in our input. For example, if only accelerometer data with the x, y, and z-axis is used, then the number of channels is three. Using accelerometer and gyroscope data along with their magnitudes will result in 8 channels (3 axes for each and 2 for magnitudes).

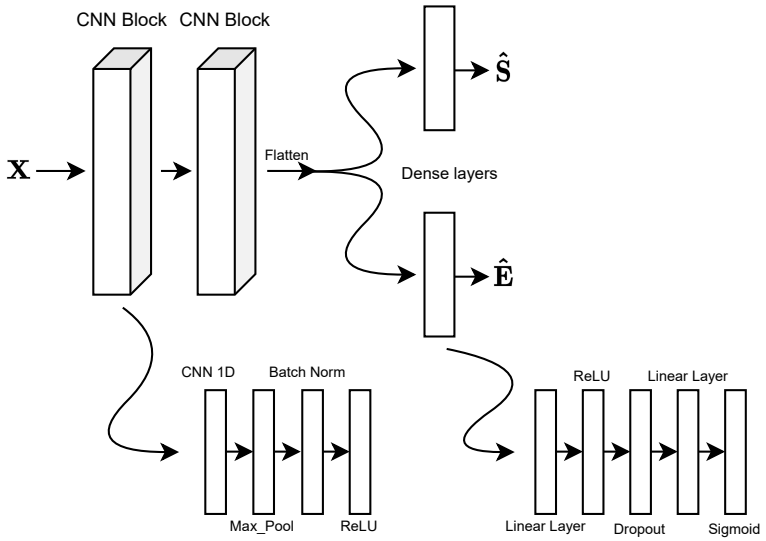


Figure 4.3: Proposed CNN model. It has two convolution blocks. Both dense layers receive exactly the same flattened output from the second convolution block.

Signal length L is the window size W_s.

Let's denote our input as $\mathbf{X} \in \mathbb{R}^{N \times C \times L}$, ground truth for start and end as $\mathbf{S} \in \{0,1\}^{N \times L}$ and $\mathbf{E} \in \{0,1\}^{N \times L}$ our neural network as a function $\mathbf{F}$, our desired output sequences for the start as $\hat{\mathbf{S}} \in \{0,1\}^{N \times L}$ and the end as $\hat{\mathbf{E}} \in \{0,1\}^{N \times L}$, then we can represent it as,

$$(\hat{\mathbf{S}}, \hat{\mathbf{E}}) = F(X)$$

Figure 4.3 depicts the CNN model I used. It has two convolution blocks followed by two separate linear blocks. The convolution block comprises a 1D convolution layer, a 1D max pooling layer, a batch Normalization layer, and a ReLU activation layer. The linear block comprises two fully connected layers, each followed by an activation layer. After the first layer, ReLU is used as an activation layer. The output from the last fully connected layer goes through the Sigmoid activation layer. The Sigmoid is used to convert the real numbers into a probability-like distribution, as the output of the Sigmoid is always between 0 and 1.

The input tensor is shaped (N, C_{in}, L). The first convolution layer has a kernel of size 5, with padding and stride equal to 1. It produces 16 output channels. So the output shape of the first layer is $(N, 16, L_1)$, where sequence length L_1 can be calculated as follow:

$$L_1 = \lfloor \frac{L + 2 \times Padding - Kernel\_size}{stride} + 1 \rfloor \tag{4.1}$$

The max pool layer has a kernel size of 5 with padding and stride equal to 1. The output shape of this layer will be $(N, 16, L_2)$, where L_2 can be calculated using the same

equation 4.1.

The batch normalization [15] layer does not change the shape as it is an element-wise operation. But it first calculates the mean and variance of each channel and then shifts them to make their mean 0 and variance 1. It then scales and shifts the channels by two learnable parameters for each channel. Batch normalization can help models learn faster by reducing the internal covariate shift.

The output shapes ($(N, 32, L_3)$ for the convolution layer with kernel size 5 and $(N, 32, L_4)$ for the max pool layer with kernel size 3.) of the second convolution block can be calculated similarly to the first block using equation 4.1. This output tensor is flattened across the batch dimension, giving us an output of shape (N, L_5), where L_5 is just a concatenation of 32 L_4.

The linear layers produce output equal to the number of neurons in them. ReLU activation layer does not change the shape as it is an element-wise operation as described in equation 2.16. The dropout layer [12] is used to avoid overfitting. It randomly drops the output of neurons with a given probability P_d. I used different values in experiments to find the optimal value.

Loss function binary cross entropy is one loss function that fits our problem. The reason is that we want each instance in $\hat{\mathbf{S}}$ to equal each instant in $\mathbf{S}$ (and the same for the end sequence). We can treat it as a classification problem for each instant of time, where we want to know whether the input at that instant belongs to the start(end) of the step or not. As both sequences only have values of either 0 or 1 on each time step, we treat it as a binary classification problem. Loss L_s for the start and L_e for the end is:

$$L_s = \frac{1}{NL} \sum_{n=1}^{N} \sum_{l=1}^{L} \mathbf{S}_{n,l} log(\hat{\mathbf{S}}_{n,l}) + (1 - \mathbf{S}_{n,l}) log((1 - \hat{\mathbf{S}}_{n,l})) \tag{4.2}$$

$$L_e = \frac{1}{NL} \sum_{n=1}^{N} \sum_{l=1}^{L} \mathbf{E}_{n,l} log(\hat{\mathbf{E}}_{n,l}) + (1 - \mathbf{E}_{n,l}) log((1 - \hat{\mathbf{E}}_{n,l})) \tag{4.3}$$

And total loss is:

$$L = \frac{L_s + L_e}{2}$$

As we can see in the above equations, if the predicted sequence is exactly the same as the desired sequence, the contribution to total loss is zero. Our objective now will be to minimize the loss for all training examples.

One problem in this loss function, discussed in [33] as well, is that it will make the model more biased towards predicting 0 in all the sequences. This is because our labels

have a massive imbalance between 0 and 1. This is because start/end instants are few compared to the intermediate instants, i.e., if the sampling rate is 80Hz and a person walks for 3 seconds and takes only 2 steps, then only two samples will have label 1 for each start, and the end and all other 238 samples out of 240 will have label 0.

To overcome this problem, I will use focal loss [22]. This loss balances the class difference by assigning a higher value weight to the less present class. We will assign a lower value weight for the class with many more samples. Equation 4.2 and 4.3 will change to:

$$L_s = \frac{1}{NL}\sum_{n=1}^{N}\sum_{l=1}^{L} p^s_{n,l}[\mathbf{S}_{n,l} log(\hat{\mathbf{S}}_{n,l}) + (1-\mathbf{S}_{n,l}) log((1-\hat{\mathbf{S}}_{n,l}))] \tag{4.4}$$

$$L_e = \frac{1}{NL}\sum_{n=1}^{N}\sum_{l=1}^{L} p^e_{n,l}[\mathbf{E}_{n,l} log(\hat{\mathbf{E}}_{n,l}) + (1-\mathbf{E}_{n,l}) log((1-\hat{\mathbf{E}}_{n,l}))] \tag{4.5}$$

Where $p_{n,l}$ is the weight for l-th instant in n-th batch. The weight will change to a higher or lower value depending on whether the ground truth is 1 or 0. We can calculate the values for the weights by finding the ratio of 1's and 0's in the complete data set. Using the similar formula in [33] weights values are:

$$p^s = \begin{cases} \frac{N^0_s + N^1_s}{2.N^1_s}, & \text{if} \quad S_{n,l} = 1 \\ \frac{N^0_s + N^1_s}{2.N^0_s}, & \text{if} \quad S_{n,l} = 0 \end{cases}$$

$$p^e = \begin{cases} \frac{N^0_e + N^1_e}{2.N^1_e}, & \text{if} \quad E_{n,l} = 1 \\ \frac{N^0_e + N^1_e}{2.N^0_e}, & \text{if} \quad E_{n,l} = 0 \end{cases}$$

Where N^0 is the number of total zeros, N^1 is the number of ones in each sequence, and N^0 is far greater than N^1 i.e. $N^0 >> N^1$.

The problem of class imbalance is also eased if the labels are relaxed, as discussed in the last section. By changing the labels of the surrounding start and end samples to 1, we increase the overall number of 1s in our labels. This will decrease the value of p^s and p^e when the ground label is 1. I will train the network with and without relaxed labels using the respective values of p^s, p^e.

Another problem with the above loss function is that it is not numerically stable, as $log(0)$ is undefined. Pytorch solves this problem by setting the output values of the log function equal to or greater than -100. Another solution is using the Sigmoid function inside the loss equation. This utilizes the log-sum-exp trick to give us more numerically stable calculations. For this purpose, I used binary cross entropy with logits loss (BCEwithLogitsLoss)of Pytorch. When using this loss, we do not need a Sigmoid layer at the end of the model.

4.3 YOLO-Like

You only look once, or YOLO [29] is a method that predicts the bounding boxes of multiple objects in an image in one forward pass. Before YOLO, the common strategy was to divide an image into smaller to big pieces and look for objects sequentially. YOLO works by creating smaller grids of the image and having 2 bounding box labels for all the grids. For each bounding box, they predict a confidence score, and for each grid, they predict ten conditional class probabilities. They get the box that fits the object most by multiplying each bounding box confidence score with the conditional class probability of the grid they occur in. This way, they get bounding boxes and the class labels of the objects in the image in one forward pass.

The problem with the current approach of producing a prediction for each time step of the window is that it has a lot of noise. It is possible that the network predicts the start of two steps consecutively without predicting an end in between. Or it can produce more than one start for the same steps. To overcome these problems [33] developed a heuristic method to clean the neural network's output. While this works, it will be a better approach to directly predict the start and end index within a window. But it is possible that a window has more than one step in it, so a YOLO-like approach is required.

The input to the network does not change and will have the same shape as the CNN method. The trick lies in modifying the labels for each window so that the model predicts indexes for the steps in it. One helpful property of our data set is that steps can not overlap, unlike in images where objects are sometimes partly in front or behind other objects. This makes our labels more simple to create and understand. We also have only two classes: There is a step or not. Therefore, we will not need to predict multiple class probabilities separately for the grids.

Figure 4.4 depicts the model's architecture. Unlike the CNN model described before, it has three separate heads of linear layers. Also, it has four convolution layers followed by a linear layer before the same output is passed onto the three heads. The output shape of convolution and max pooling layers can be found using the same equation as 4.1. The number of output channel for each block are 128, 64, 16, and 5 consecutively. The number of output neurons for each head is 10.

Labels I used windows for CNN models, where we have small fixed-size chunks of the whole signal. Our labels were also similar, with a window with 0 or 1 for each timestamp for start and end. Table 3.1 shows this kind of window. For this method, I will convert the labels to have the following corresponding values for each step: index of the middle of the step, half width of that step, and class label 1.

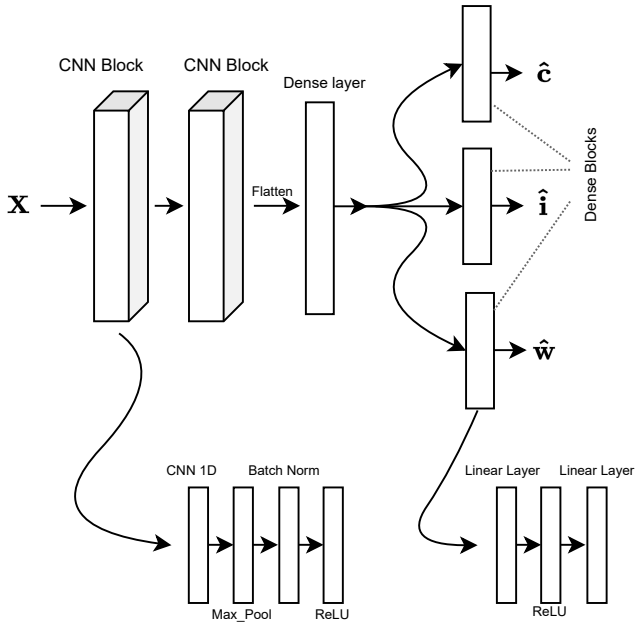


Figure 4.4: YOLO-like model with four convolution blocks and three separate heads of linear blocks. Each head is responsible for the confidence score $\hat{\mathbf{c}}$, index $\hat{\mathbf{i}}$, and width $\hat{\mathbf{w}}$ of the steps.

I divide the window into S grids. If the middle index of the step is in a grid, that grid is responsible for predicting that step. Each grid will have the possibility of predicting two steps. For each step, it will predict the middle index i of the step relative to the grid boundary, half-width w of the step, and the binary class c. If there is no step, all values, including the class, will be zero.

I used the window size of 200 for the data set. This window then has five grids, each 40 timestamps apart. For each grid, we have two pairs of (i,w,c). So the output shape is then $5 \times 2 \times 3$. The decision to allow predictions of two steps per grid was made after observing that some steps were very close in time, i.e., the distance between the two steps was less than 39.

It is possible that a step is not starting or ending in a window, but more than half of it is in that window. That does not affect the labels, as we are interested in the middle of the step. It also makes sense because if the significant portion of the step is known, the model should be able to make predictions without altogether seeing it.

We can use the following steps to get the start and end indexes from the model's output. Add the starting values of the grid to the predicted indexes $\hat{\mathbf{i}}$. Then, take the difference of the result and $\hat{}$ for the start, and add the result to $\hat{\mathbf{w}}$ for the end. Multiply the resulting values with the binary confidence score. To get the binary confidence score,

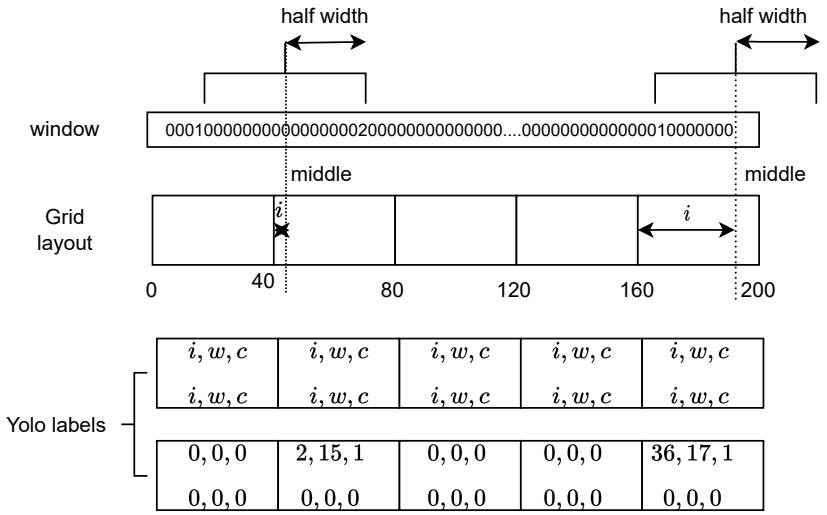


Figure 4.5: Illustration of labels for YOLO-like model using a dummy example. The middle index of the first step is close to the start of the first grid, so i has a value of 2. While the middle step is at the end, and i has a value of 36. These values are the difference between the start index of the grid and the middle step. Note that the second step does not end in this window. Nonetheless, its middle point is in this window, so the last grid of this window is responsible for it.

round the $\hat{\mathbf{c}}$ output vector of the model. This can be described as:

$$Start = (\hat{\mathbf{i}} + \mathbf{g} - \hat{\mathbf{w}})\hat{\mathbf{c}}$$

$$End = (\hat{\mathbf{i}} + \mathbf{g} + \hat{\mathbf{w}})\hat{\mathbf{c}}$$

where $\mathbf{g} = [0, 40, 80, 120, 160, 0, 40, 80, 120, 160], \hat{\mathbf{i}}, \hat{\mathbf{w}}, \hat{\mathbf{c}} \in \mathbb{R}^{10}$.

To get the indexes for all the data, I looped through all the windows with no overlap and added the window number to the above start and end indexes. This will give us indexes relative to the whole signal, not just the current window.

Loss function In this method, The purpose of the model is to produce real numbers, i.e., indexes and width. These problems usually have a distance loss, with which we try to reduce the distance or difference between two numbers. Therefore, I used mean square loss for the middle index, width, and class label. I am using mean square error for both the regression and classification part to have both losses to scale. if $\mathbf{i}, \mathbf{w}, \mathbf{c}$ are

actual labels of the data, and $\hat{\mathbf{i}}, \hat{\mathbf{w}}, \hat{\mathbf{c}}$ are the predictions of the model then loss is

$$L_i = \frac{1}{N}\frac{1}{10}\sum_{n=1}^{N}\sum_{j=1}^{10}(i - \hat{i})^2 +$$

$$L_w = \frac{1}{N}\frac{1}{10}\sum_{n=1}^{N}\sum_{j=1}^{10}(w - \hat{w})^2 +$$

$$L_c = \frac{1}{N}\frac{1}{10}\sum_{n=1}^{N}\sum_{j=1}^{10}(c - \hat{c})^2$$

$$L = L_i + L_w + L_c \tag{4.6}$$

Similar to the loss function this loss might also suffer the class imbalance problem. This can happen more in the second labels inside a grid (second row of labels in figure 4.5 which only have zeros.). However, I trained the Yolo-like models without the weights and still got good results. Future work can include refining this loss function to get even better results.

4.4 Hyperparameters

There are many hyperparameters that are important during the training and the extraction of labels. I will discuss some important hyperparameters briefly.

Learning rate Learning rate is the value that determines how much the weights are adjusted in each iteration. Stochastic gradient descent has a constant learning rate while Adam has an adaptive learning rate. I used Adam Optimizer for all the models, usually initializing it with 0.001.

Batch size Batch size determines how many examples are processed together in one forward pass. A smaller batch size takes more training time but usually yields better results. I mostly used 512 or 1024 for batch size to speed up the training.

post processing thresholds After getting the predictions from the model in raw form, we need to process them so that they can be compared with the original labels. This is very important in the case of CNN and LSTM models, as they produce relaxed labels like in 4.1. We need to get back to single index labels. For this purpose, I check for each

patch of these generated labels and if the patch width is above a threshold, I return the middle of the patch as the predicted index for the start (or end).

This threshold is 15 in the case of relaxed labels (labels are relaxed to 10 indexes on both sides of the original label). If the model was trained on original labels without relaxing them, it still produces a patch that is of arbitrary length around the original label. For this kind of model output, I used 4 as the threshold. So even if the model predicted a small patch for the step start (or end), it will be considered as a prediction.

Another threshold is the margin of prediction errors. In other terms, it tells us how much the model's predictions can be off from the exact time stamps we expected when it doesn't get them exactly right. The sample rate of the data is 80, and on average human step takes about half a second. That is around 40 samples. This threshold should be less than 40 samples (or the minimum value from data). I used 10 to 15 samples for this threshold. The error margin of 15 samples means that the model's prediction deviates maximum by 187.5 milliseconds. In the result section I will discuss the mean deviation value.

5 Experiments and Results

In this chapter, I will discuss the results of various experiments using the models discussed in the previous chapter. The experiments include different values of hyperparameters and types of inputs. Before going into the experiments, Let's look at the evaluation metrics I used for them.

5.1 Metrics

F1 score The F1 score is the harmonic mean of precision and recall. Precision is the measure of positive prediction being correct. The recall is the measure of how many true positives were identified as true by the model. A high precision score means that the model does not easily label a sample as positive. In contrast, a high recall score means the model will easily label a sample as positive, given even a small similarity.

The above scores depend on true positive, true negative, false positive, and false negative. True positive means the model predicts a positive sample as positive. True negative means the model predicts a negative sample as negative. False positive means the model wrongly predicts a negative sample as positive. False negative means the model wrongly predicts a positive sample as negative. In the case of our problem, label 1 to a sample is positive, and 0 is negative. If a model correctly predicts 1 to a sample whose ground truth is also 1, then it is a true positive, and so on. The matrix that consists of how many true positives and such we have is called the confusion matrix, depicted in table 5.1.

		Ground Truth	
		Positive	Negative
Predictions	Positive	**True Positives (TP)**	**False negatives (FN)**
	Negative	**False positives (FP)**	**True negatives (TN)**

Table 5.1: Confusion Matrix

The F1 score can be calculated as follows:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Where as Precision and Recall are calculated as such:

$$Precision = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FP}}$$

$$Recall = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FN}}$$

The F1 score in terms of true positives and negatives is then:

$$F1 = \frac{\mathbf{TP}}{\mathbf{TP} + \frac{1}{2}(\mathbf{FP} + \mathbf{FN})} \tag{5.1}$$

Time sensitivity F1 score gives us an overall idea about how well our model performs. Still, it will have a lower score even if the model predicted the start of a step one timestamp away. We need another metric that tells us how accurate our predictions are regarding time. We can calculate the average time differences between the actual and predicted timestamps. This is the same as mean absolute error. We also need to introduce a threshold after which the predicted timestamp is too early or too late and no longer contributes to the mean absolute error. To calculate this, we can use:

$$MAE = \frac{1}{\hat{T}} \sum |t - \hat{t}| \quad \text{for every} \quad |t - \hat{t}| \leq th \tag{5.2}$$

Where $\hat{T}$ is the total number of predicted steps within the threshold, t is the actual timestamp, and $\hat{t}$ is the predicted timestamp. th is the value after which we do not consider the predicted prediction accurate.

I will also create a histogram depicting how many predictions are far away from true predictions against how far away they are. A model for which most predictions are 0 timestamps far away can be considered the best.

Running count accuracy Another important and widely used accuracy metric is the ratio of the total number of predicted steps over the total number of actual steps. This number gives us a rough estimate of how many steps the model detected. This can be calculated as:

$$RCA = \frac{\hat{T}}{T} \tag{5.3}$$

where $\hat{T}$ is the total number of predicted steps within threshold th, and T is the total number of actual steps in the data.

5.2 Results

The data set includes simultaneous sensor readings from different body positions, such as hand, right, left, and back pockets. I will first discuss the results of the models using just hand-sensor readings. Then, I will discuss the models' performance for all sensors together. I also have a number of different pre-processing techniques for the input data, such as low-pass filtering, normalization, and magnitude as an input feature. Another important variable is the window size of the input. I will evaluate models with different combinations of sensor positions, window sizes, and pre-processing.

Hyper-parameters Some hyper-parameters must be fixed to decrease the total number of experiments. Therefore, all experiments' total number of epochs will be set at 30 unless mentioned otherwise. The batch size is set to 1024. The labels will be relaxed to 10 samples around the true label. The weights for loss in the case of regular and relaxed labels are the following:

$$p^s = \begin{cases} 28.103, & \text{if} \quad S_{n,l} = 1 \\ 0.509, & \text{if} \quad S_{n,l} = 0 \end{cases}$$

$$p^e = \begin{cases} 28.103, & \text{if} \quad E_{n,l} = 1 \\ 0.509, & \text{if} \quad E_{n,l} = 0 \end{cases}$$

In the case of relaxed labels

$$p^s = \begin{cases} 1.405, & \text{if} \quad S_{n,l} = 1 \\ 0.776, & \text{if} \quad S_{n,l} = 0 \end{cases}$$

$$p^e = \begin{cases} 1.405, & \text{if} \quad E_{n,l} = 1 \\ 0.776, & \text{if} \quad E_{n,l} = 0 \end{cases}$$

Test train split The data set I used consists of 12 people walking for different amounts of time. There are two ways of splitting the time series data. One is to take a small part from each person and use that to test the model. In this way, the model has already seen everyone's walk. It turned out during initial experiments that having this split would attain a higher accuracy score very quickly. The other way is to hold some people's data files away from training and use them for testing. This way, we can safely assume that

Model	Window size											
	100						200					
	F1 score		RCA		MAE		F1 score		RCA		MAE	
	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
LSTM	0.87	0.87	0.87	0.87	2.06	2.07	0.86	0.86	0.90	0.90	2.5	2.5
CNN	0.88	0.88	0.89	0.88	1.93	1.96	0.87	0.87	0.89	0.89	1.99	2.00
YOLO-like	-		-		-		0.92		0.93		4.25	

Table 5.2: Best scores of LSTM, CNN, and YOLO for handheld sensor. RCA means running count accuracy; the lower the number, the better the model is, and MAE is the mean absolute error. The input to these models is pre-processed slightly differently. Unlike YOLO-like, CNN and LSTM models predict the start and the end of the step separately.

the model is generalizing instead of memorizing the data. I took the latter approach and used three people's data to test all the models.

5.2.1 Hand held sensor

It is important to compare my methods' performance and the LSTM method used in [33]. Their data set was collected from three users, holding the sensor only in their hand in texting style. I will only use the readings from the sensor held in hand during the user's walk.

Table 5.2 shows the best accuracy scores of three different models. The number of epochs used is 20 for CNN and LSTM and 35 for the YOLO-like model. This is because the YOLO-like model has another loss function than the CNN and LSTM and takes more time to converge. Training the YOLO-like model even for more epochs can yield better results.

The LSTM model for window size 200 in table 5.2 is trained on data with no gyroscopic values in it, while the model for window size 100 has it. Both LSTM models' input is normalized and filtered, and magnitude is added as a feature. The CNN models for both window sizes are trained with gyroscopic data. The input is normalized, filtered, and magnitude added as a feature. The results suggest that CNN performs better than the LSTM model. The CNN model especially does well in predicting the steps close to the actual labels, with an average deviation of 1.96 samples (24.5 milliseconds).

YOLO-like model is trained to predict the middle of the step. So, I compared the true middles of steps to predicted middles. The model works well in predicting whether there is a step, but the predictions deviate more than CNN's predictions. The reason can be training the models for just 35 epochs. It was evident from the loss value in each epoch

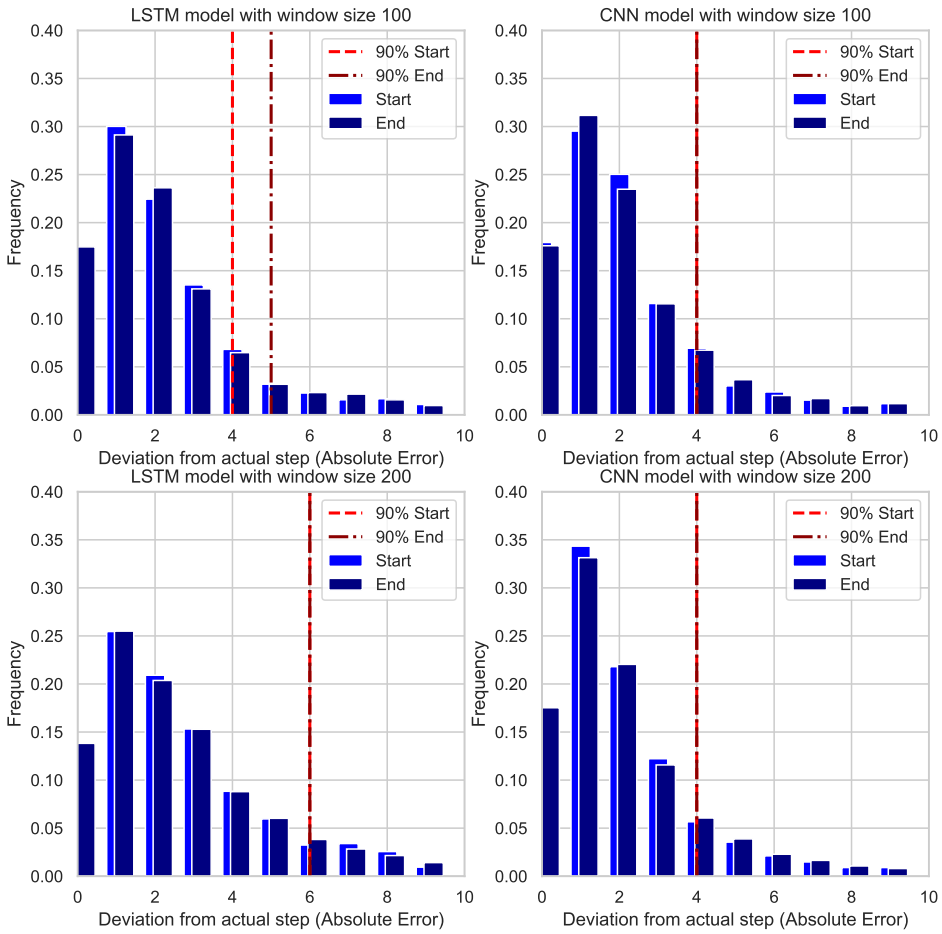


Figure 5.1: Mean absolute Error of CNN and LSTM models for handheld sensor. The vertical lines mark the point at which 90% of predicted steps exhibit deviations up to this level.

that the model can still be trained more to decrease loss further.

5.2.2 Combined sensors

The data set was collected from four sensors worn on four body positions in synchronization. This means we can combine the feature columns and use the same label again. This results in having four times the total number of steps. I trained the models on this data similarly to the handheld sensor models.

The table 5.3 shows the results of three different models for the combined data. The total number of steps in this case is now 8468. RCA indicates the ratio of the number of predicted steps to the total number of steps. CNN and LSTM were able to predict

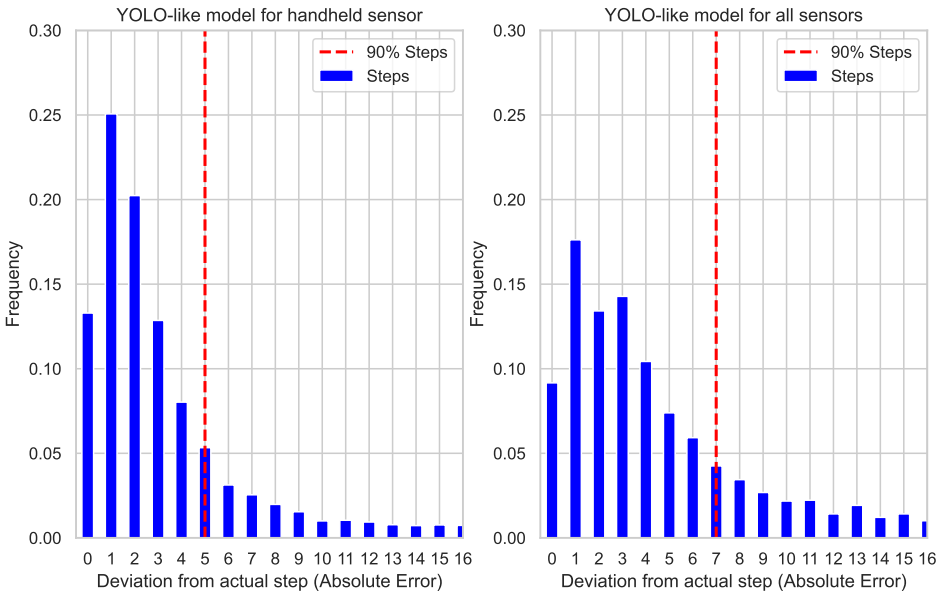


Figure 5.2: MAE of YOLO-like models for handheld sensor. The YOLO-like model performs worse than the CNN and LSTM models but predicts more steps in total, as shown in Table 5.2.

more than 93% of the steps, with CNN predicting better in terms of having the least average deviation of the predicted steps.

	Window size											
Model	100						200					
	F1 score		RCA		MAE		F1 score		RCA		MAE	
	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
LSTM	0.90	0.90	0.93	0.93	1.81	1.82	0.90	0.90	0.95	0.95	1.87	1.88
CNN	0.90	0.90	0.94	0.93	1.64	1.66	0.90	0.90	0.95	0.95	1.68	1.71
YOLO-like	-		-		-		0.93		0.89		3.05	

Table 5.3: Best scores of LSTM, CNN, and YOLO for all the sensor data combined. RCA means running count accuracy, and MAE is the mean absolute error. The input to these models is pre-processed slightly differently. CNN model is best in terms of correctly predicting the steps. CNN and LSTM are almost similar in the number of steps detected. YOLO seems to perform worst of all.

5.2.3 Effect of Different Preprocessings

I have a number of different ways to preprocess the input to models. This includes window size, filtering the sensor signals, adding magnitude as an extra feature to the

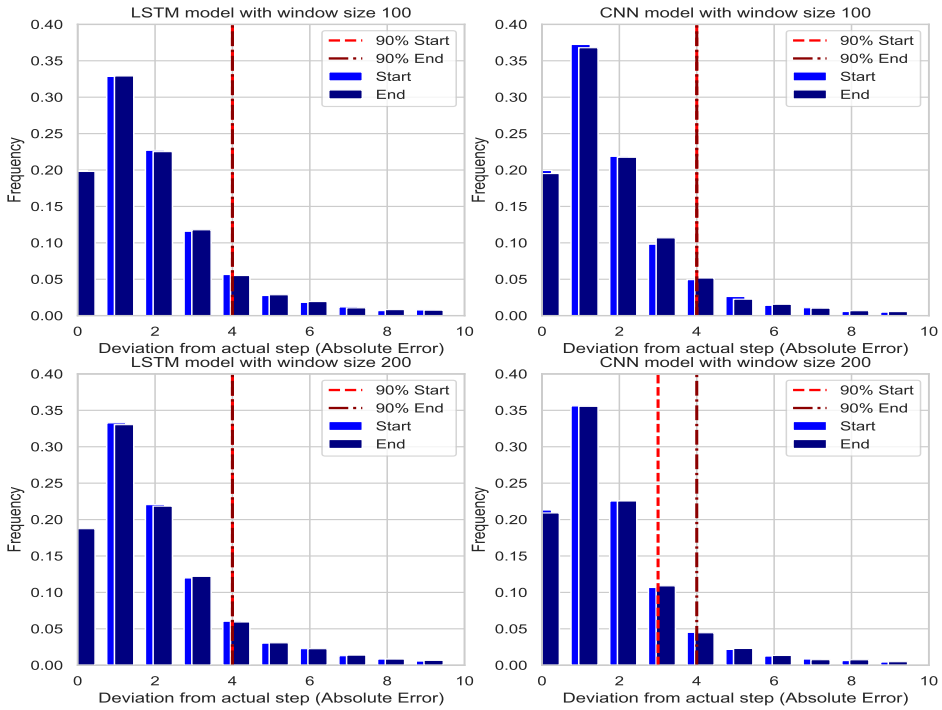


Figure 5.3: MAE of CNN and LSTM models for combined sensor data.

input, and having the gyroscope's data in the input. The window sizes I used are 100 and 200. Every other setting is a binary option. i.e., filtered data or unfiltered data. This gives a total of 16 combinations (2^4). I trained the CNN models with all these combinations to find out which setting affects the accuracy most.

After training the models, I saved the 4 test scores, namely mean absolute error and RCA for the start and the end. Then, I used random forest regressor [3] to calculate the feature importance. I used the Scikit-learn [27] implementation of this method. Random Forest is an ensemble of decision trees to make more accurate predictions and avoid overfitting.

For my purpose, I extracted the feature importance array for each setting of the input. The higher the value, the higher the importance of that setting. Figure 5.4 illustrates that having the gyroscope's data in the input makes the most impact on the performance of the model. This makes sense as gyroscope data provides additional information, whereas techniques like filtering and magnitude addition are primarily feature extraction methods from the same data. Nonetheless, filtering the signal or adding the magnitude does help the model to converge early.

The choice of window size does not make much difference in the model's performance. This might not be the case if a wide range of window sizes are tested, as I used only 2 different sizes. In general, it is good to have a small window size because if the model

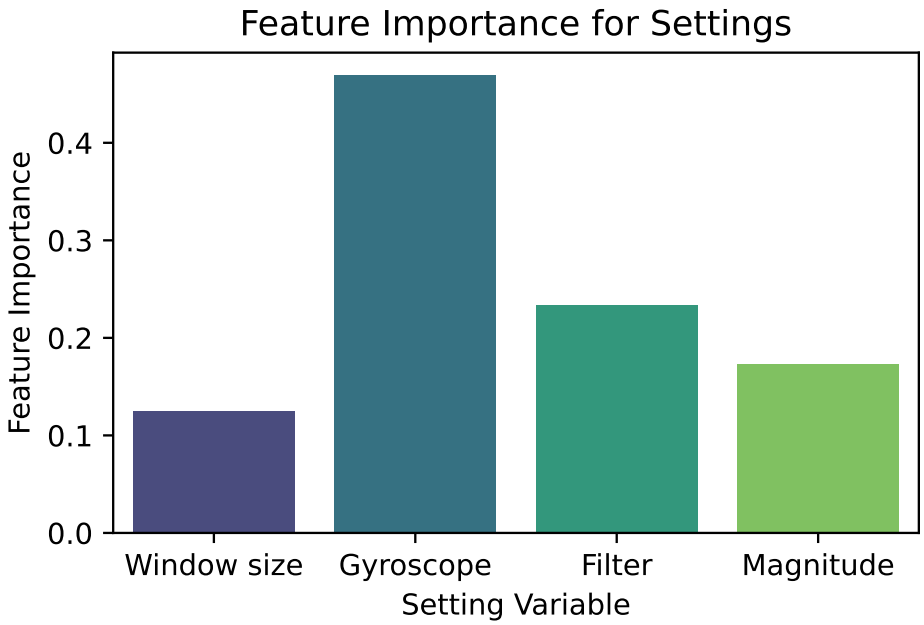


Figure 5.4: Importance of preprocessing settings. The higher the value, the higher the effect on the accuracies.

is implemented on hardware and requires a window size of, for example, 200, then there would be a delay of 2.5 seconds in predictions (given the sampling rate of 80).

5.2.4 Model Robustness

The above results were calculated by using the same test data. But what if the test data is different? How different will the accuracies be? For this purpose, k-fold cross-validation is used. In this technique, the dataset is divided into k subsets or folds. Then, the process iterates k times, taking one fold as the test set and k-1 as the training set. Then, all the results are reported, usually aggregated, mostly by calculating each score's mean and standard deviation.

I have 12 people in the data set, so I did 12-fold cross-validation. That means training 12 models, each time testing with a different person. Figure 5.5 illustrates this setting. The data set is divided into 12 subsets, and a different subset is used in each iteration. I trained CNN models with fixed preprocessing, which includes relaxing labels, filtering, adding magnitude, and using the gyroscope's data. The window size was 200.

In figure 5.6, we can see that most of the models score more than 80% in terms of RCA, with the exception of one scoring around 50%. RCA is the running count accuracy, and RCA of 1 means that all the steps were predicted within a certain range. MAE in the

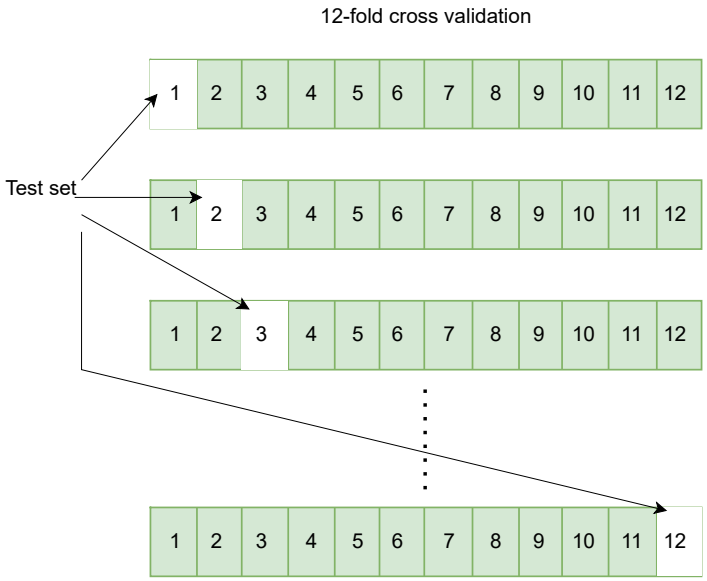


Figure 5.5: 12-fold cross validation. 12 is the number of different people in the dataset. In each iteration, a different model is trained with green data and tested with white data.

CNN	window size 200					
	F1 score		RCA		MAE	
	Start	End	Start	End	Start	End
Mean	0.89	0.89	0.91	0.92	2.20	2.17
STD	0.05	0.04	0.12	0.09	0.78	0.67
Max	0.94	0.94	0.99	0.99	4.07	3.46
Min	0.75	0.78	0.56	0.63	1.22	1.20

Table 5.4: Mean and standard deviation of scores of 12 CNN models, tested for one person each time. Max and Min mean the highest and lowest scores, but the lower the score for the MAE, the better it is.

figure is the mean absolute error, and a lower value means more accurate predictions. MAE is around 2.1 and 2.2 for the start and the end consecutively. The mean and standard deviation of all the scores for 12 people can be seen in the table 5.4.

Training and inference time A model's training includes loading and pre-processing the data and training the model itself. The training time of the model depends on how big the data is and how many parameters it has.

The LSTM model has around two million parameters. It also takes an incredible amount of time to train. To train 16 LSTM models, it took me almost three days. The inference time of the LSTM model is also large. On average, it took 0.3 to 0.5 seconds to get the

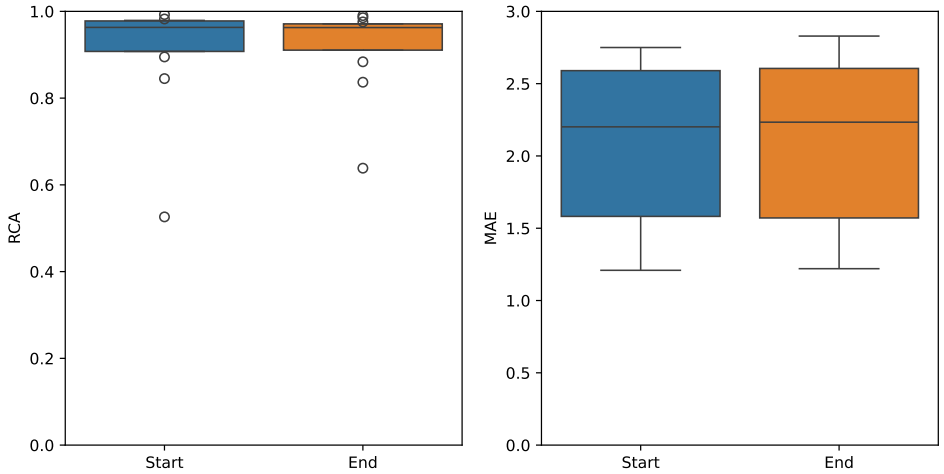


Figure 5.6: CNN robustness using 12-fold cross validation. RCA is running count accuracy, which is the ratio of the number of predicted steps to the number of actual steps. MAE is the mean absolute error.

predictions for one window.

On the other hand, the CNN model has just 0.8 million parameters. It is also extremely fast in terms of inference time, with an average of 0.003 seconds to process one window. The YOLO-like model has 3 million parameters but is also fast in inference time, averaging 0.005 seconds. Training of 16 models of CNN took around three hours, and the YOLO-like models took about 6 hours.

I trained my model on a cluster provided by the university. It is running the Ubuntu 20.04.6 operating system on an x86_64 system. The graphic card is an NVIDIA Tesla V100 PCIe 32GB. The inference time was calculated by running the model on the CPU, AMD Ryzen, with 2.10GHz.

6 Conclusion

6.1 General Discussion

This thesis work aimed to improve and develop neural network models to detect the steps in the accelerometer and gyroscope data recorded with different numbers of people and four different body positions of sensors. I proposed using a different model in the method used by [33] and another completely different approach inspired by YOLO [29] architecture.

The standard approach has been to count the number of steps in the entire signal. The thesis work in [20] used the CNN model to count the steps and integrated a method to detect the locality of the steps when the counter value increases. [33] introduced an approach of directly predicting where a step is using LSTM. They also have a comprehensive heuristic to clean the output of the LSTM model using different threshold values.

I proposed using the Convolution neural network instead of LSTM in [33] method. Even though the dataset is a time series, we are interested in some sort of peak detection, so we use CNN's ability to extract spatial features to detect these peaks. It is preferable to use CNN, as the models I propose have a lot fewer trainable parameters than LSTM. This, in turn, affects the training time and, more importantly, the inference time of the models. CNN models are, on average, 170 times faster than the LSTM models. This is important if the models are to be deployed on some hardware and are required to predict the steps as soon as they happen in real-time.

Another method I proposed is CNN-based YOLO-like architecture, which always predicts the steps in a start-end pair. The benefit of this model is that its output requires minimum post-processing. LSTM and simple CNN-based models predict the start and end independently, and it is possible that the model predicts the start of a step but skips its end. Even though the YOLO-like model has more parameters than LSTM and simple CNN models, its inference time is still far quicker than LSTM.

The accuracy of the CNN models is slightly higher than that of the LSTM models. In contrast, YOLO-like models perform slightly worse than CNN and LSTM models. It is

also important to state that [33] reported just the F1 score, and [20] discussed F1 and RCA, but both did not focus on reporting how close the predicted steps are to the actual steps. I also focused on discussing and achieving a better score of mean absolute error (between true and predicted steps).

Another key factor in performance comparison is the similarity and difference in the dataset. While [20] had sensors on different body positions (wrist, ankle, and hip), [33] had sensors only in hand. The dataset I used had sensors on different body positions (hand, side, and back pockets), and people were asked to walk in different scenarios. Also, I think it makes a difference in how the dataset is divided for training and testing, and if the split is not based on separating people for testing, then achieving a good result might be easy.

Overall, The proposed CNN-based method achieved better results in terms of accuracy and is faster than LSTM models. A YOLO-like model is also a viable solution as it requires minimum post-processing of the predictions and is as fast as CNN models.

6.2 Future work

Future work on this topic includes different paths. One obvious direction is exploring different architectures of the models. Both CNN and YOLO-like models had only two convolution layers. It can be beneficial to check if changing the number of layers or kernel sizes improves the accuracy of the models.

Another direction can be to explore different loss functions for CNN using the same techniques. I currently use binary cross-entropy loss, but other loss functions may be better suited for this problem. One such loss function is comparing the cumulative sum of the indexes of ones in the predicted and original label sequences.

As the CNN-based method also requires post-processing, another direction could be to come up with simple and efficient post-processing algorithms that cover the problem of missing the start(end) of a step, removing too big or too small steps, and various other problems.

In the case of YOLO-like models, It is worth experimenting with the model's architecture to find the optimal model for the accelerometer-based dataset. Also, it is worth exploring different logics of the YOLO-like technique. I am currently comparing the step's middle and half-width, but the step's start and width can also be used. Different loss functions can also be used for this model. I currently used mean square error even for the classification, but the more common loss, like binary cross-entropy, could be more appropriate.

Lastly, It is worth exploring the model's performance for different scenarios of walking. It is possible that the model suffers more, for example, when the user is walking up the stairs. Insight into the accuracy in this manner could also lead to better results overall.

6.3 Personal Notes

This thesis work can be divided into three parts: data pre-processing, model training, and calculation of accuracies. In comparison, the most difficult and time-consuming part was coming up with a robust data pre-processing script that could handle the diverse pre-processing settings. This included different window sizes, options to select if the magnitude is added to the input, filtering, gyroscope's data and their combinations, and different ways to split the dataset for training and testing.

The second challenge was correctly calculating the accuracies, especially the mean absolute error. While a simple F1 score is easy to calculate, it does not give enough insight into the model's performance. The easiest part for me was the training pipelines. As I have worked with Pytorch already, this task was straightforward.

During the writing of this thesis, I gained more clarity into the working of CNN and LSTM models. I also gained experience writing data-handling, pre-processing, and model training scripts, which polished my deep learning-related programming skills.

List of Figures

2.1 Gyroscope 4
2.2 Representative Dataset 6
2.3 Linear Regression 7
2.4 A single neuron 10
2.5 Neural network with 2 layers 10
2.6 2D Convolution 15
2.7 Convolution of two square waves 15
2.8 Maximum pooling 17

3.1 MTw Awinda Dongle 21
3.2 Body position of sensors 22
3.3 Plot of a snippet of data 23
3.4 Sliding Windows 25
3.5 Plot of snippet of normalized data 27
3.6 Plot of snippet of filtered data 28
3.7 Plot of snippet of filtered and normalized data 28
3.8 Plot of a snippet of filtered Norm of axis 29
3.9 Boxplot of steps 30

4.1 1 LSTM unit 31
4.2 LSTM model used in [33] 33
4.3 CNN Model 35
4.4 YOLO-like Model 39
4.5 Illustration of YOLO-like labels 40

5.1 MAE of CNN and LSTM models for handheld sensor 47
5.2 MAE of YOLO-like models for handheld sensor 48
5.3 MAE of LSTM and CNN models 49
5.4 Importance of preprocessing settings 50
5.5 12-fold cross validation 51
5.6 CNN robustness using 12-fold cross validation 52

List of Tables

3.1 A snippet of data set 24

4.1 Relaxed Labels 33

5.1 Confusion Matrix 43
5.2 Best scores of LSTM, CNN, and YOLO-like for handheld sensor 46
5.3 Best scores of LSTM, CNN, and YOLO-like for combined data 48
5.4 Aggregated results of 12-fold cross validation 51

1 Results of YOLO-like models 61
2 Part 1. Results of LSTM models 62
3 Part 2. Results of LSTM models 62
4 Part 1. Results of CNN models 63
5 Part 2. Results of CNN models 63
6 Part 1. Results of CNN models for most useful settings 64
7 Part 2. Results of CNN models for most useful settings 64
8 Part 1. 12 fold cross validation 65
9 Part 2. 12 fold cross validation 65

Appendix

	0	1	2	3	4	5	6	7
Window size	200	200	200	200	200	200	200	200
Gyro	✓	✓	✗	✗	✓	✓	✗	✗
Filter	✓	✓	✓	✓	✓	✓	✓	✓
magnitude	✓	✗	✓	✗	✓	✗	✓	✗
Just Hand	✓	✓	✓	v	✗	✗	✗	✗
F1	0.93	0.93	0.93	0.93	0.94	0.94	0.93	0.93
MAE	4.26	11.86	5.17	5.44	3.14	3.06	3.30	3.23
ratio steps	0.93	0.01	0.69	0.65	0.79	0.89	0.10	0.23
count steps	1975	21	1464	1383	6713	7576	860	1951
Total steps	2117	2117	2117	2117	8468	8468	8468	8468

Table 1: The results of the 8 YOLO-like models trained with different settings.

Model	0	1	2	3	4	5	6	7
Window size	100	100	200	200	100	100	200	200
Relaxed labels	✓	✗	✓	✗	✓	✗	✓	✗
Gyro	✓	✓	✓	✓	✓	✓	✓	✓
Filter	✓	✓	✓	✓	✓	✓	✓	✓
magnitude	✓	✓	✓	✓	✓	✓	✓	✓
Just Hand	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.88	0.86	0.90	0.84	0.91	0.90	0.91	0.89
F1 end	0.87	0.86	0.89	0.84	0.91	0.90	0.91	0.89
MAE start	2.06	2.31	2.11	1.89	1.81	3.39	1.74	2.58
Ratio start	0.87	0.74	0.92	0.89	0.94	0.01	0.95	0.03
Count start	1849	1567	1949	1881	7948	114	8029	249
MAE end	2.08	2.34	2.13	1.90	1.82	3.43	1.75	2.52
Ratio end	0.88	0.74	0.89	0.89	0.94	0.01	0.94	0.03
Count end	1854	1573	1876	1874	7948	124	7973	250
Total steps	2117	2117	2117	2117	8468	8468	8468	8468

Table 2: Part 1. The results of the first 8 of 16 LSTM models trained with different settings.

Model	8	9	10	11	12	13	14	15
Window size	100	100	200	200	100	100	200	200
Relaxed labels	✓	✗	✓	✗	✓	✗	✓	✗
Gyro	✗	✗	✗	✗	✗	✗	✗	✗
Filter	✓	✓	✓	✓	✓	✓	✓	✓
magnitude	✓	✓	✓	✓	✓	✓	✓	✓
Just hand	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.86	0.84	0.87	0.81	0.90	0.88	0.91	0.87
F1 end	0.86	0.83	0.87	0.81	0.90	0.88	0.90	0.87
MAE start	2.35	2.43	2.51	2.46	1.91	2.86	1.87	2.63
Ratio start	0.84	0.71	0.90	0.68	0.92	0.04	0.95	0.07
Count start	1788	1510	1913	1444	7802	331	8045	556
MAE end	2.36	2.44	2.50	2.48	1.92	2.93	1.89	2.64
Ratio end	0.84	0.70	0.90	0.71	0.92	0.04	0.95	0.07
Count end	1776	1481	1910	1502	7754	334	8071	553
Total steps	2117	2117	2117	2117	8468	8468	8468	8468

Table 3: Part 2. The results of the second 8 of 16 LSTM models trained with different settings.

	0	1	2	3	4	5	6	7
Window size	100	100	200	200	100	100	200	200
Relaxed labels	✓	✗	✓	✗	✓	✗	✓	✗
Gyro	✓	✓	✓	✓	✓	✓	✓	✓
Filter	✓	✓	✓	✓	✓	✓	✓	✓
Magnitude	✓	✓	✓	✓	✓	✓	✓	✓
Just Hand	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.88	0.84	0.88	0.86	0.91	0.84	0.91	0.84
F1 end	0.88	0.84	0.88	0.86	0.91	0.84	0.91	0.84
MAE start	1.93	2.06	2.00	1.92	1.64	1.81	1.68	1.80
Ratio start	0.89	0.86	0.89	0.74	0.95	0.05	0.96	0.09
Count start	1890	1817	1890	1565	8019	429	8091	758
MAE end	1.96	2.09	2.00	1.83	1.66	1.88	1.71	1.72
Ratio end	0.89	0.82	0.89	0.74	0.93	0.05	0.95	0.09
Count end	1883	1735	1887	1572	7901	400	8051	749
Total steps	2117	2117	2117	2117	8468	8468	8468	8468

Table 4: Part 1. The results of the first 8 of 16 CNN models trained with different settings.

	8	9	10	11	12	13	14	15
Window size	100	100	200	200	100	100	200	200
Relaxed labels	✓	✗	✓	✗	✓	✗	✓	✗
Gyro	✗	✗	✗	✗	✗	✗	✗	✗
Filter	✓	✓	✓	✓	✓	✓	✓	✓
Magnitude	✓	✓	✓	✓	✓	✓	✓	✓
Just Hand	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.83	0.80	0.87	0.82	0.90	0.83	0.90	0.83
F1 end	0.83	0.80	0.87	0.82	0.90	0.83	0.90	0.83
MAE start	2.26	2.32	2.13	2.07	1.82	1.85	1.82	1.88
Ratio start	0.82	0.70	0.88	0.62	0.94	0.09	0.94	0.08
Count start	1736	1487	1868	1313	7980	762	7993	664
MAE end	2.29	2.35	2.12	2.12	1.83	1.76	1.82	1.80
Ratio end	0.83	0.58	0.89	0.69	0.93	0.10	0.94	0.09
Count end	1753	1221	1883	1462	7873	815	7972	797
Total steps	2117	2117	2117	2117	8468	8468	8468	8468

Table 5: Part 2. The results of the second 8 of 16 CNN models trained with different settings.

	0	1	2	3	4	5	6	7
Window size	100	200	100	200	100	200	100	200
Gyro	✓	✓	✓	✓	✓	✓	✓	✓
Filter	✓	✓	✗	✗	✓	✓	✗	✗
Magnitude	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.88	0.88	0.88	0.89	0.87	0.87	0.89	0.88
F1 end	0.88	0.88	0.88	0.89	0.87	0.87	0.88	0.88
MAE start	1.91	2.00	1.89	1.79	2.11	2.08	1.87	1.92
Ratio start	0.91	0.90	0.88	0.91	0.88	0.85	0.88	0.88
Count start	1929	1910	1859	1924	1870	1808	1858	1861
MAE end	1.92	1.97	1.91	1.82	2.14	2.14	1.85	1.92
Ratio end	0.91	0.90	0.87	0.90	0.89	0.88	0.85	0.89
Count end	1924	1901	1852	1911	1875	1871	1800	1879
Total steps	2117	2117	2117	2117	2117	2117	2117	2117

Table 6: Part 1. The results of the first 8 of 16 CNN models trained with different settings. These values were used to find the feature importance in figure 5.4.

	8	9	10	11	12	13	14	15
Window size	100	200	100	200	100	200	100	200
Gyro	✗	✗	✗	✗	✗	✗	✗	✗
Filter	✓	✓	✗	✗	✓	✓	✗	✗
Magnitude	✓	✓	✓	✓	✗	✗	✗	✗
F1 start	0.85	0.86	0.87	0.88	0.85	0.86	0.85	0.87
F1 end	0.85	0.86	0.86	0.88	0.85	0.85	0.85	0.87
MAE start	2.14	2.25	2.11	2.03	2.32	2.10	2.22	2.03
Ratio start	0.79	0.85	0.84	0.90	0.82	0.83	0.79	0.86
Count start	1666	1806	1780	1911	1737	1754	1666	1814
MAE end	2.08	2.27	2.10	2.01	2.36	2.12	2.27	2.04
Ratio end	0.78	0.84	0.84	0.90	0.82	0.80	0.84	0.84
Count end	1657	1785	1774	1912	1734	1701	1771	1780
Total steps	2117	2117	2117	2117	2117	2117	2117	2117

Table 7: Part 2. The results of the last 8 of 16 CNN models trained with different settings. These values were used to find the feature importance in figure 5.4.

ID	10	11	4	2	3	6
F1 start	0.89	0.93	0.84	0.90	0.75	0.89
F1 end	0.89	0.93	0.84	0.90	0.78	0.89
MAE start	2.39	1.49	2.58	2.62	4.07	2.75
Ratio start	0.95	0.98	0.89	0.98	0.53	0.91
Count start	786	564	639	671	342	476
MAE end	2.40	1.54	2.59	2.65	3.47	2.83
Ratio end	0.94	0.98	0.88	0.99	0.64	0.92
Count end	779	562	631	673	415	480
Total steps	827	576	714	683	650	522

Table 8: Part 1 of results of 12 fold cross validation. The ID corresponds to the ID in figure 3.9.

ID	8	7	9	0	1	5
F1 start	0.93	0.92	0.95	0.85	0.91	0.94
F1 end	0.94	0.92	0.95	0.85	0.91	0.94
MAE start	1.21	2.10	1.32	2.30	2.02	1.61
Ratio start	0.99	0.96	0.97	0.84	0.97	0.98
Count start	1220	1285	1234	507	731	999
MAE end	1.22	2.21	1.32	2.25	1.99	1.58
Ratio end	0.99	0.96	0.97	0.84	0.97	0.97
count end	1217	1284	1234	502	729	991
Total steps	1229	1342	1274	600	752	1022

Table 9: Part 2 of results of 12 fold cross validation. The ID corresponds to the ID in figure 3.9.

Bibliography

[1] URL: https://www.movella.com/products/wearables/xsens-mtw-awinda.

[2] Nahime Al Abiad et al. "SMARTphone inertial sensors based STEP detection driven by human gait learning". In: *2021 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. 2021, pp. 1–8.

[3] Leo Breiman. "Random forests". In: *Machine learning* 45 (2001), pp. 5–32.

[4] Stephen Butterworth et al. "On the theory of filter amplifiers". In: *Wireless Engineer* 7.6 (1930), pp. 536–541.

[5] S. Chan and G. Sohn. "INDOOR LOCALIZATION USING WI-FI BASED FINGERPRINTING AND TRILATERATION TECHIQUES FOR LBS APPLICATIONS". In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XXXVIII-4/C26 (2012), pp. 1–5. URL: https://isprs-archives.copernicus.org/articles/XXXVIII-4-C26/1/2012/.

[6] Estefania Munoz Diaz and Ana Luz Mendiguchia Gonzalez. "Step detector and step length estimator for an inertial pocket navigation system". In: *2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. 2014, pp. 105–110.

[7] Frank van Diggelen and Per Enge. "The World's first GPS MOOC and Worldwide Laboratory using Smartphones". In: *Proceedings of the 28th International Technical Meeting of the Satellite Division of The Institute of Navigation*. 2015.

[8] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. URL: https://doi.org/10.1214/aos/1013203451.

[9] Kunihiko Fukushima. "Cognitron: A Self-Organizing Multilayered Neural Network". In: *Biol. Cybern.* 20.3–4 (Sept. 1975), pp. 121–136. URL: https://doi.org/10.1007/BF00342633.

[10] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. URL: https://doi.org/10.1007/BF00344251.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012.

[13] Ngoc-Huynh Ho, Phuc Huu Truong, and Gu-Min Jeong. "Step-Detection and Adaptive Step-Length Estimation for Pedestrian Dead-Reckoning at Various Walking Speeds Using a Smartphone". In: *Sensors* 16.9 (2016). URL: https://www.mdpi.com/1424-8220/16/9/1423.

[14] D. H. Hubel and T. N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *Physiol* 160.1 (Jan. 1962), pp. 106–154.2.

[15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015.

[16] Wonho Kang and Youngnam Han. "SmartPDR: Smartphone-Based Pedestrian Dead Reckoning for Indoor Localization". In: *IEEE Sensors Journal* 15.5 (2015), pp. 2906–2916.

[17] Guolin Ke et al. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017).

[18] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551.

[19] Hwan-hee Lee, Suji Choi, and Myeong-jin Lee. "Step Detection Robust against the Dynamics of Smartphones". In: *Sensors* 15.10 (2015), pp. 27230–27250. URL: https://www.mdpi.com/1424-8220/15/10/27230.

[20] Basil Lin. "Machine learning and pedometers: An integration-based convolutional neural network for step counting and detection". PhD thesis. Clemson University, 2020.

[21] Juanying LIN, L H Leanne CHAN, and Hong YAN. "A Decision Tree Based Pedometer and Its Implementation on the Android Platform". English. In: Third International Conference on Signal, Image Processing and Pattern Recognition (SIPP 2015) ; Conference date: 21-02-2015 Through 22-02-2015. Feb. 2015.

[22] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.

[23] Long Luu et al. "Accurate Step Count with Generalized and Personalized Deep Learning on Accelerometer Data". In: *Sensors* 22.11 (2022). URL: https://www.mdpi.com/1424-8220/22/11/3989.

[24] Eladio Martin et al. "Precise Indoor Localization Using Smart Phones". In: *Proceedings of the 18th ACM International Conference on Multimedia*. MM '10. Firenze, Italy: Association for Computing Machinery, 2010, pp. 787–790. URL: https://doi.org/10.1145/1873951.1874078.

[25] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019.

[26] Monique Paulich et al. “Xsens MTw Awinda: Miniature wireless inertial-magnetic motion tracker for highly accurate 3D kinematic applications”. In: *Xsens: Enschede, The Netherlands* (2018), pp. 1–9.

[27] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[28] Walter Pitts and Warren S. McCulloch. “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biophysics 5* (1943), pp. 115–113.

[29] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection.* 2016.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536. URL: `https://api.semanticscholar.org/CorpusID:205001834`.

[31] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[32] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2015.

[33] Stef Vandermeeren and Heidi Steendam. “Deep-Learning-Based Step Detection and Step Length Estimation With a Handheld IMU”. In: *IEEE Sensors Journal* 22.24 (2022), pp. 24205–24221.

[34] Lucas Vieira. URL: `https://upload.wikimedia.org/wikipedia/commons/e/e2/3D_Gyroscope.png`.

Declaration on oath

I hereby certify that I have written my master thesis independently and have not yet submitted it for examination purposes elsewhere. All sources and aids used are listed, literal and meaningful quotations have been marked as such.

Muhammad Zakriya Shah SarwarK54738, 20.10.2023

Consent to plagiarism check

I hereby agree that my submitted work may be sent to PlagScan (www.plagscan.com) in digital form for the purpose of checking for plagiarism and that it may be temporarily (max. 5 years) stored in the database maintained by PlagScan as well as personal data which are part of this work may be stored there.

Consent is voluntary. Without this consent, the plagiarism check cannot be prevented by removing all personal data and protecting the copyright requirements. Consent to the storage and use of personal data may be revoked at any time by notifying the faculty.

Muhammad Zakriya Shah SarwarK54738, 20.10.2023