KATNN: KAT WALK C ALTERNATIVE MOTION CAPTURE ALGORITHM

KATNN: MOTION CAPTURE AND MACHINE LEARNING TO PREDICT REALISTIC CHARACTER TRAJECTORY IN A VIRTUAL GAME FOR THE KAT WALK C

BY

KENNETH MATIRA, B.Sc.

A Report

submitted to the Computing & Software

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Masters of Engineering

Masters of Engineering (2023) McMaster University

(Computing & Software) Hamilton, Ontario, Canada

TITLE:	KATNN: Motion Capture and Machine Learning to Predict Realistic Character Trajectory in a Virtual Game for the KAT Walk C
AUTHOR:	Kenneth Matira B.Sc (Actuarial & Financial Mathematics Co-Op), McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Ryan Leduc
NUMBER OF PAGES:	xx, 81

Lay Abstract

In the world of Virtual Reality (VR), motion sickness, nausea, and disorientation remains a big concern for many users. The KAT Walk C is an omni-directional treadmill, which aims to convert human motion to virtual movement. This is intended to reduce the aforementioned concerns. However, the original KAT C algorithm of human locomotion has its limitations, where motion is frequently converted incorrectly. In this report, we will introduce an alternative input mechanism for the KAT Walk C, KATNN, which focuses on two primary objectives: allowing the user to move in multiple directions, and having the ability to register slower type motions. KATNN was created by the construction of modular neural networks. We will discuss steps to create the models, investigate current issues and potential solutions involving calibration and disorientation. Readers may optionally view the results by watching the following video: https://youtu.be/SbUXoQ0-G9Q.

Abstract

In the world of virtual reality (VR), motion sickness, nausea, and disorientation remains a big concern for many users. This issue is rooted in the idea of your virtual character moving around in a virtual environment while your body remains stationary, resulting in sensory conflict within your body, LaViola (2000). The KAT Walk C is an omni-directional treadmill, which strives to convert human motion to virtual motion, in hopes of mitigating these conflicting sensory inputs. However the original KAT C algorithm of human locomotion has its limitations: it may not always register all user movements effectively, and in some cases, might not register them at all. This results in discrepancies between user actions and the virtual character.

In this report, we will introduce KATNN, an alternative input mechanism for the KAT Walk C. KATNN focuses on two primary objectives: allowing the user to move in additional axis (having the ability to move left and right on top of moving forward), and having the ability to register slower type movements. While it may sound easy, one of the biggest obstacles of this project was working with very sparse sensor data. KATNN was created by constructing modular neural networks and in this report, we will discuss the process of creating those models. We will discuss the development of a VR Unity game, methods of data collection/processing, and discuss common issues with using the KAT Walk C in the form of calibration and

disorientation. We also propose solutions on how this can be solved through our research sandbox game. Readers may optionally view the results by viewing the following video: https://youtu.be/SbUXoQ0-G9Q.

Acknowledgements

I would like to thank my research supervisor, Dr. Ryan Leduc, for giving me the opportunity to work on this project, as well as guiding and supporting my research throughout my academic time. Regular updates, insightful feedback, and sharing ideas during regular meetings have immensely improved the direction of my work. I also would like to thank him for providing me the resources necessary in order to carry out my research project which included providing the KAT Walk C treadmill, KAT Walk C shoes, and a Meta Quest 2.

Many thanks to Dr. Anwar Mirza of the W Booth School of Engineering Practice and Technology department, for teaching the Deep Learning course that I took as well as providing insight during a one-on-one meeting about my project. Thanks for providing insightful feedback on the issues of my initial approach, and for providing an alternative approach which was ultimately the approach I used for my algorithm.

Special thanks to the KAT VR support and development team for being cooperative and providing the Software Development Kit for me to work on for my project. Ultimately without it, this project would have not been possible as we would not be able to access the necessary sensor data.

Lastly, I would like to mention McMaster alumni, Jan Wolos, who volunteered their time to test out my implementation, assisted with data recording sessions, as

well as provided ideas and feedback on how this project could potentially be improved.

Contents

List of Figures

List of Tables

List of Algorithms

Notation, Definitions, and Abbreviations

Notation

$A << B$ A is significantly less than B. Significantly will be defined as being smaller by at least a factor of 10.

Definitions

Data Augmentation

Refers to the technique of creating new data in order to enlargen the training dataset by manipulating the current data.

Data Processing

Refers to the process of running a script in order to extract and classify information.

Interpolation Refers to the technique of estimating an object based on its closest keypoints, and taking the weighted average of those closest keypoints.

Iteration Similarly to loops, iteration is defined as the number of times the script ran. In the context of sensor recording, iteration number refers to the number of times the script ran since the start of data recording session.

SADSR Sum of Absolute Delta Sensor Readings. A technique to calculate the total absolute change of each sensors within a given time interval.

Sparse Data Refers to a dataset which contains missing or very limited information.

TNSAT Total Number of Sequences Above a Threshold. A technique to calculate the number of cycles by identifying a sequence that only appears once in a cycle in the form of a peak.

TNSBT Total Number of Sequences Below a Threshold. A technique to calculate the number of cycles by identifying a sequence that only appears once in a cycle in the form of a trough.

Virtual Character

Refers to the first person character in the virtual environment that imitates the user.

Abbreviations

AVERAGE Average/Medium Pace (Motion Speed Class)

BPM Beats per Minute

BSTEPS	Backward Steps (Motion Class)
CSV	Comma Separated Values (File Type)
FAST	Fast Pace (Motion Speed Class)
FPS	Frames Per Second
KATCAlgo	KAT C Algorithm (Original Algorithm)
KATNN	KAT Neural Network (Our Algorithm)
LAR	Large Steps (Motion Type Class)
LPitch	Left Foot Pitch Rotation Data
LRoll	Left Foot Roll Rotation Data
LSIDESTEPS	Left Sidesteps (Motion Class)
RPitch	Right Foot Pitch Rotation Data
RRoll	Right Foot Roll Rotation Data
RSIDESTEPS	Right Sidesteps (Motion Class)
SADSR	Sum of Absolute Delta Sensor Readings
SDK	Software Development Kit
SLOW	Slow Pace (Motion Speed Class)
SML	Small Steps (Motion Type Class)
STEPS	Walking/Steps (Motion Class)

TNSAT	Total Number of Sequences Above a Threshold
TNSBT	Total Number of Sequences Below a Threshold
VR	Virtual Reality
XLSX	Microsoft Excel Spreadsheet (File Type)

Declaration of Academic Achievement

I, Kenneth Matira, declare this independent project to be my own work. I am the sole author of this document. No part of this work has been published or submitted for publication or for a higher degree at another institution.

To the best of my knowledge, the content of this document does not infringe on anyone's copyright.

My research supervisor, Dr. Ryan Leduc, other contributing members such as Dr. Anwar Mirza, KAT VR SDK, and Jan Wolos, have provided guidance and support at various stages of this project. I completed all of the research work.

Chapter 1

Introduction

Virtual Reality (VR) has rapidly evolved over the years through its advancement in realism, making users feel more immersed in the virtual world. However, one common issue of VR usage is the feeling of motion sickness, nausea, disorientation, and unease. This is caused by movements in the virtual world not aligning to the user's real world actions, resulting in conflicting sensory, LaViola (2000). KAT Walk C is an omni-directional treadmill which aims to reduce that issue by integrating human locomotion into virtual environments. However, the original KAT C algorithm of human locomotion has its limitations: it may not always register all user movements effectively, or in some cases, not register at all. This results in discrepancies between the user's actions and the character's motion in the virtual game. In this report, our objective is to introduce a new algorithm, KATNN, which will better depict the users velocity by focusing on many of the current implementation's limitations.

This algorithm, KATNN, will be significant for two main reasons. Relative to the limitations of the original KAT C implementation, our algorithm will tackle the possibility of moving in multiple directions (left and right as well as forward), and

effectively register slower-motions such as "sneaky steps". While this may not sound challenging, one of the biggest obstacles with this project was working with the very sparse data that the KAT devices provided. Additionally, this report will investigate other issues, including ineffective calibration which leads to movement drifting to the left or right, and solutions on how this could be solved.

This report encompasses a variety of chapters. The upcoming chapters will cover the following subjects:

- **Chapter 2:** This report will first introduce important topics and required information in order to understand the rest of the report.
- **Chapter 3:** The focus of the report shifts to the realm of our VR research game. In this chapter, we will discuss the game's development and evolution, as well as dive into potential enhancements that could elevate its realism.
- **Chapter 4:** This chapter focuses on the methodologies of data collection, data processing processing techniques, and the technique of creating synthetic data.
- **Chapter 5:** Our algorithm, KATNN, takes center stage in this chapter. We will introduce the different layers of our algorithm and the different motion classes that each layer could predict.
- **Chapter 6:** A thorough examination of our algorithm's results will unfold in this chapter. We will analyze its performance, gauge its effectiveness, as well as point out its limitations.
- **Chapter 7:** This chapter focuses on the related work that has been conducted externally to this project. Additionally, it covers work that was scrapped and

not utilized in the final implementation of KATNN due to its poor performance. While those implementations may have failed, possible tweaking could yield much better results.

- **Chapter 8:** This chapter will provide concluding thoughts regarding the contributions made in this report. Additionally, it will explore possible avenues for improvement and potential future directions.

- **Appendix A:** This appendix will provide a deeper explanation on the data augmentation methods used in this project while providing a more understandable example.

- **Appendix B:** This appendix will contain additional, more technical information.

- **Appendix C:** This will contain the instructions on how to install the code on your device.

- **Appendix D:** Finally, this will contain additional information on the KAT SDK and problems with the version of the SDK that was used for this project.

Chapter 2

Overview/Background

This section serves as a foundational explanation into key concepts and details for comprehending the subsequent content of this report. Some of these topics will include defining rotations and translations in a 3-dimensional space, explaining KAT Walk C and the data provided from the SDK, differences between HTTP Requests and WebSocket, and a high level overview on how the data will flow.

2.1 Rotation and Translation

In the eyes of motion capture, three distinct types of rotations are recognized: pitch, yaw, and roll rotations. A visual representation of these rotations is provided in Figure 2.1. To comprehend these rotations from the perspective of foot movements (as this project exclusively relies on foot and hip sensor data), pitch entails tilting the foot up and down. Roll involves twisting the ankle left and right; this is frequently observed in various sports and instances of ankle rolling. The final rotation, yaw, is observed from the point of the hip (since yaw rotation data is not collected). Yaw

can be visualized as the twisting of the hips to the left or right.

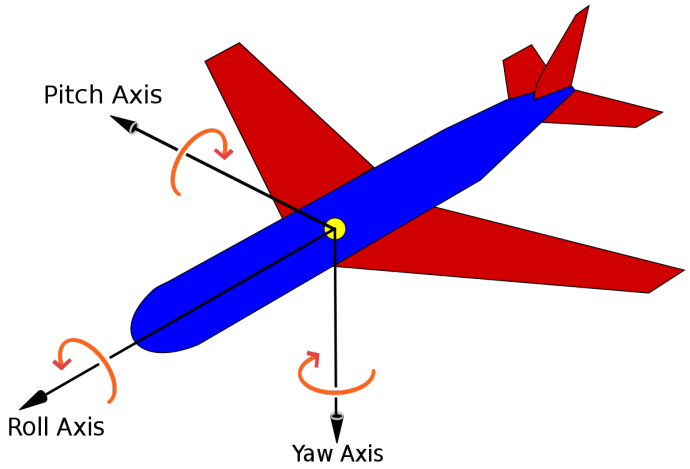

Figure 2.1: This image was taken from the Aircraft principal axes Wikipedia page to visualize the different axes of rotations, Wikipedia (2007). While this is about aircrafts, this is very related to foot rotations which we will study.

Translations involve the relocation of an object within a specific plane or environment. Object movement can be measured by determining its positioning along each axis. However, an alternate method of achieving object translation is by defining a trajectory/velocity to the object. With respect to movement, moving forward and back is defined as moving along the Z-axis, moving from left to right is defined as moving along the X-axis, and for formality, moving up and down is defined as moving along the Y-axis (this project will not look at the Y-axis). There are two types of axes to understand: the global axis which is the axis of the virtual environment (this will never change) and the local axis, which is the the axis relative to the direction that the object is facing. This project will exclusively talk about the local axis, as the goal of this project is based on moving the user relative from its perspective.

2.2 KAT Walk C Devices Introduction

The KAT Walk C includes the treadmill which the user steps on to perform their motion. This can be seen in the left image of Figure 2.2. Contrary to belief, the surface of the treadmill actually has no sensors on it to determine the location of the foot. However, the treadmill does have a sensor on the back plate which collects the yaw rotation (the top of the left image where the straps are attached to). On the right of Figure 2.2 are the KAT Walk C shoes that are used with the treadmill. These have a sensor on each of the shoe which capture the following data: left foot pitch rotation (LPitch), left foot roll rotation (LRoll), right foot pitch rotation (RPitch), and right foot roll rotation (RRoll).

Figure 2.2: This figure shows the KAT Walk C Treadmill (Left) along with the accompanied KAT Walk Shoes (Right).

The surface of the KAT Walk C is also quite unique. It can be described as a bowl, where at the centre it is very flat, but as you move further away from the centre

and closer to the edge of the KAT surface, you will experience more of an incline, resulting in a different foot rotation and different sensor readings, this can be seen in Figure 2.3.

Figure 2.3: This figure depicts the incline of the KAT Walk C surface when the shoe is near the edge of the surface.

The primary challenge within this project revolves around the limited foot positional awareness. Our task entails working with highly sparse sensor data encompassing only 4 foot rotations and 1 body yaw rotation. Despite these constraints, we are aiming to leverage this information to accurately anticipate the user's trajectory.

2.3 Python Websocket Server

WebSockets serve as a communication protocol facilitating the connection between a client and a server. In contrast to other commonly used protocols like HTTPS, WebSockets are favored for their persistent connection and bidirectional nature. Given that our game generates numerous requests per minute, opting for an HTTP approach would involve a substantial amount of overhead data due to repeated requests. In contrast, WebSockets require just one initial communication for setup. Once established, the client can generate multiple requests without the need to transmit that redundant overhead data. Additionally, the bidirectional characteristic of WebSockets aligns seamlessly with our requirements, as it enables the client to send data to the server and receive responses in return, making it an ideal choice for this project.

In this project, a Python WebSocket server was created which will be used to communicate with the research game. This server will run continuously until manually halted by the user (or an error has occurred). The server awaits the establishment of a client connection. Upon connection, it anticipates a message from the client, structured as a JSON array with dimensions $(windowSize, 4)$. Upon receipt of this message, the server undertakes a processing procedure in the form of machine learning and a prediction, the details of which will be expounded upon in Chapter 5 when the KATNN algorithm is formally introduced.

Upon completion, the server furnishes a response in the form of a JSON object featuring three attributes: $xVelocity, yVelocity, zVelocity$. This response is subsequently relayed back to the client, where it is utilized to modify the trajectory of the virtual character. The underlying server logic is seen in Algorithm 1.

Algorithm 1 Python WebSocket Server. This algorithm provides a high level overview of how the WebSocket API will operate.

```
while Server Is Running do
    while Client is Connected do
        ClientMessage ← WebSocket.GetMessage()          ▷ Get client message
        ResponseMessage ← ProcessRequest(ClientMessage)
        ResponseJson ← ResponseMessage.ToJSON()
        webSocket.Send(ResponseJson)                    ▷ Send JSON response to client
    end while
end while
```

2.4 Workflow

With the communication protocol now established, the first assumption made consists of successfully establishing a connection between the research game and the server. Once the assumption is fulfilled, the model will begin in the context of the research game. In this game, the primary objective is to gather essential data. Once enough data is collected, the next step involves transmitting this data to the server through the initiated WebSocket connection.

Upon receiving the data, the server undertakes all required processing tasks. Subsequently, once the server's processing is finalized, it sends its output back to the game. This output will be in the form of a 3 dimensional-vector representing the velocity/trajectory of the character.

Once the game has received that calculated response from the server, the game will update the character's trajectory and restart the cycle of collecting data and feeding the newly fetched data back to the server to then be processed again. This cycle will continue unless the connection is aborted or the user does not want to continue to use the KATNN algorithm.

This workflow structure is visually represented in Figure 2.4.

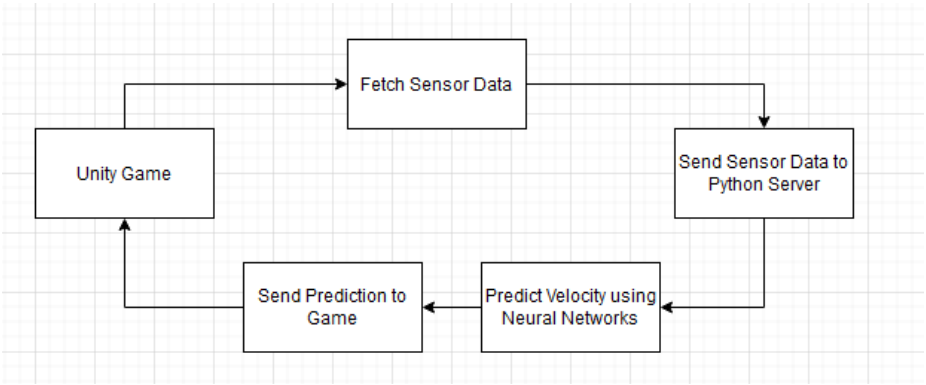

Figure 2.4: This figure depicts the communication workflow between Client (game) and Server (Python).

Chapter 3

Unity SDK and Research Game

After obtaining the KAT SDK software, an additional key piece of software that was provided was a demonstration game that contained the necessary logic to use the KAT SDK and KAT input system in a Virtual Reality environment (KATVR (2020)). However, during the testing of this game significant issues came to light. One of the key issue was a feeling of disorientation. This was caused by the textures used in the demo game. There was additionally a lack of objects present while moving away from the centre of the demo game, causing confusion or loss of direction due to the absence of recognizable cues to gauge movement or change in position.

3.1 Map Adjustments

To work around these issues, two notable adjustments were made to the game map: (1) addition of walls so that the user can't move away from the intended sandbox area. Adding walls additionally provided the user a visual estimate for their action, using the walls as a point of reference; (2) added a grid like texture pack ("Gridbox

Prototype Materials Unity Package") (Ciathyza (2021)) which makes it much easier to gauge the user moving around the sandbox area. This texture/material was used on all of the objects in the games This includes the floor, the walls, and the objects (cubes, capsules) that the user can collide with. Different colours were used on different objects which added another layer of recognizable cues when users moved around. These changes can be seen Figure 3.1, where the left image represents the demo game provided by KATVR and the right image represents the updated game for this research project.

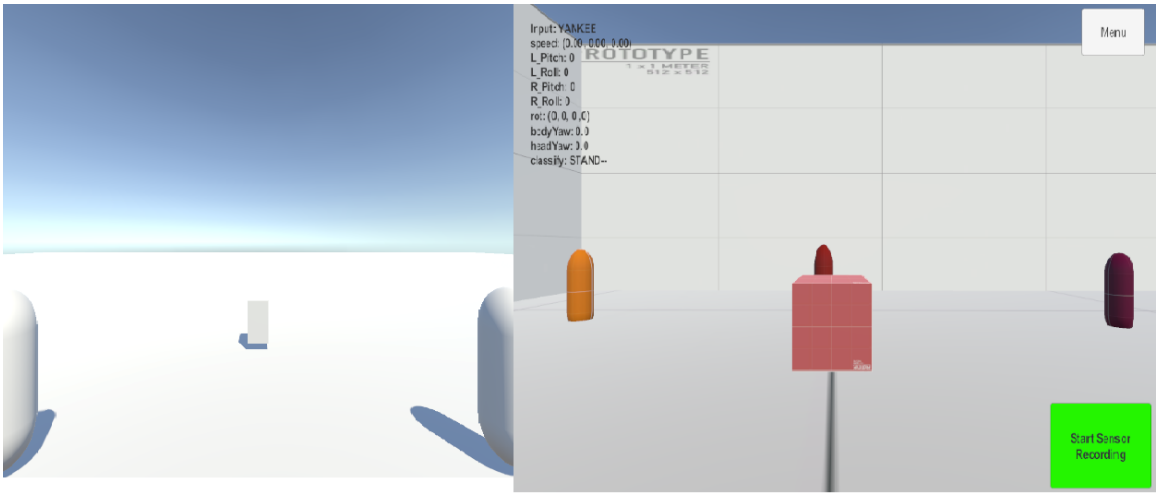

Figure 3.1: Demo game provided by SDK (Left Image) vs. Updated game after changes (Right Image)

3.2 FixedUpdate: Consistent Polling Across Computers

In the Unity game scripts, the FixedUpdate() function is used over the Update() function. The rationale for this choice was that it provided a constant polling rate

between different computers. FixedUpdate() is defined to run at a constant interval, whereas Update() is defined to run based on how many FPS (frames per second) the application is using. FixedUpdate() is defined to run at around every 0.02 seconds or 50 times a second (UnityDocumentation (2021)). The consistent frequency of using FixedUpdate() is key when using different computers as these scripts will now fetch sensor data at the same rate which is critical for accurate calculations (velocity, orientation of the user, etc.).

3.3 HUD for Sensor Data

At the upper left corner of the screen depicted in Figure 3.2 we will depict a collection of labels and text. These elements represent real-time sensor data and character information. Displaying this information in the heads up display (HUD) serves the purpose of providing convenient access to data when reviewing past motions, primarily for training and analysis purposes. Some of the HUD values displayed include the relevant sensor readings which were used for research, the speed of the virtual character, the input system that's being used (the original system or our new one), as well as the predicted output class from the KATNN algorithm.

3.4 Sensor Recording

On the game interface, there exists a button labeled "Start Sensor Recording." This button is designed for users to capture sensor data throughout a recording session. Once the recording concludes, the accumulated data is extracted and saved as a CSV file on the user's system. Some of the exported data includes the time of the recorded

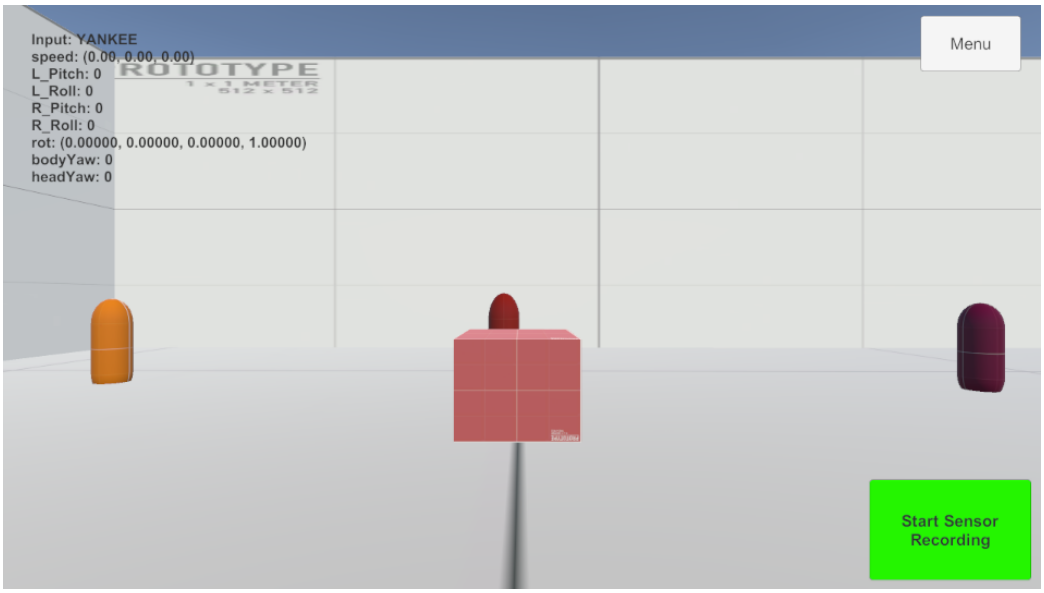

Figure 3.2: The UI of the game screen. Note, when the user is using the VR headset, they do not see the buttons or the sensor data HUD.

data, the iteration of the data (which is defined as the number of times the script ran since the start of the recording session), and the LPitch, LRoll, RPitch, and RRoll rotations.

This was achieved by initializing all lists/arrays which will be exported. Started when a user requests to start recording sensor data, the data will be added to their corresponding array for each loop/iteration. Upon a user's request to conclude recording, a function is triggered to export the collected data into a CSV file, with each array being mapped to a distinct column in the CSV. Once the export is complete, then all arrays are cleared to prepare for another recording in the event that a user would like to start another recording session. An overarching view of this logic is encapsulated in Algorithm 2.

Algorithm 2 Sensor Recording Button Logic. In this algorithm *isSensorbuttonPressed* is controlled by whether or not the sensor recording button is pressed. When first pressed, we collect data and save to array. When pressed again, we save that data and export that data to a CSV.

Require: $isSensorbuttonPressed \in \{true, false\}$
Require: $LPitchReading \in \mathbb{Z}$ ▷ $\mathbb{Z}$ represent all integers
Require: $LRollReading \in \mathbb{Z}$
Require: $RPitchReading \in \mathbb{Z}$
Require: $RRollReading \in \mathbb{Z}$
  $totalRecording \leftarrow 0$ ▷ Define the variables needed
  $DateTimeArray \leftarrow []$
  $IteratonArray \leftarrow []$
  $LPitchArray \leftarrow []$
  $LRollArray \leftarrow []$
  $RPitchArray \leftarrow []$
  $RRollArray \leftarrow []$
  while Game is Running **do**
    if $isSensorbuttonPressed$ is $true$ **then**
      $DateTimeArray.Append(GetDate())$ ▷ Save the sensor data to arrays
      $IterationArray.Append(totalRecording)$
      $LPitchArray.Append(LPitchReading)$
      $LRollArray.Append(LRollReading)$
      $RPitchArray.Append(RPitchReading)$
      $RRollArray.Append(RRollReading)$
      $totalRecording \leftarrow totalRecording + 1$
    else if $isSensorbuttonPressed$ is $false$ & $totalRecording > 0$ **then**
      $exportArraysToCSV$ ▷ Export all arrays mentioned above to CSV
      $totalRecording \leftarrow 0$ ▷ Reset all Recorded data
      $DateTimeArray \leftarrow []$
      $IteratonArray \leftarrow []$
      $LPitchArray \leftarrow []$
      $LRollArray \leftarrow []$
      $RPitchArray \leftarrow []$
      $RRollArray \leftarrow []$
    end if
  end while

3.5 Selecting Input System

The game also consist of the option to toggle between two distinct input systems: the original KAT C input system (KATCAlgo) and the new KATNN algorithm. The user can see what the current input system that has been selected based on the name of the button, shown in Fig 3.3. Here, the original KATC input system is named KATCAlgo or Yankee, while the name KATNN or Tango represents the KATNN algorithm.

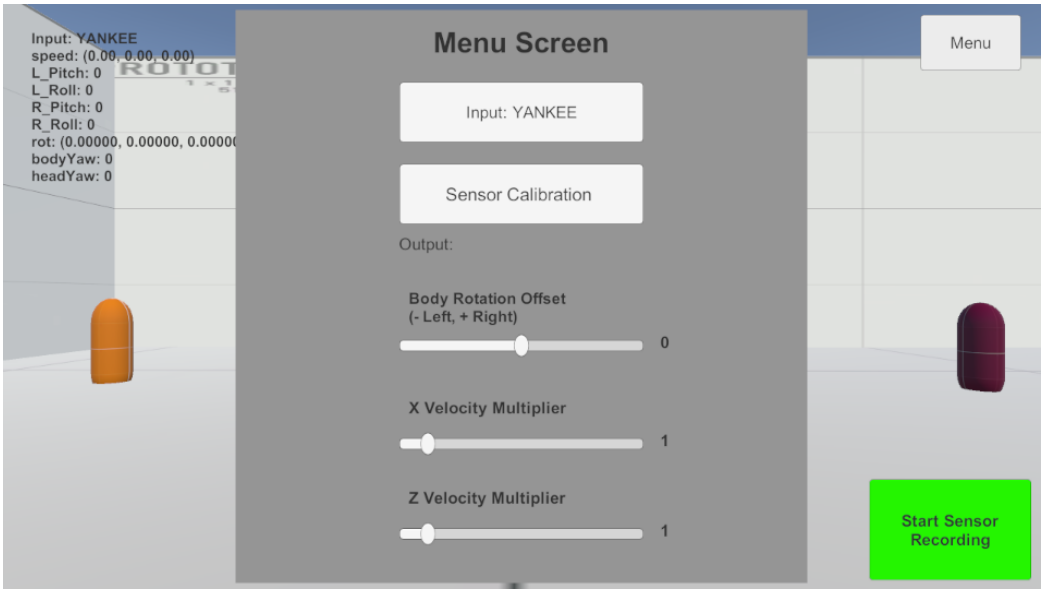

Figure 3.3: The UI of the Menu screen. This menu contains a variety of functionalities including toggling input system, sensor calibration, and tweaking user speeds.

3.5.1 Original KAT C Input System (KATCAlgo)

In the event that the input system is using the the original KAT C input system, all that needs to be done is to call a function provided by the KAT SDK which will

return the 3 dimensional vector, (x, y, z), and use that returned vector as the velocity for the virtual character.

3.5.2 KATNN Algorithm

If the input system is the KATNN algorithm, the collection of a window of the latest sensor data is saved in an array. Once enough sensor data has been collected and saved, this array then sends the latest 26 sensor readings to the Python server via the WebSocket connection. This server will then process that data in order to make a prediction of the characters trajectory. Once a response is made, the game uses that response value to update the velocity of the virtual character. The following logic to how this was implemented can be seen in Algorithm 3.

3.5.3 Debugging

The original implementation of Algorithm 3 did not contain the variable readyMessageWebsocket, and what was discovered was that the Python server was being overwhelmed by incoming requests, resulting in a steadily increase in backlog of pending requests. This occurred because the Unity game was generating requests at a rate that exceeded the processing capacity of the Python server. To work around this issue, the variable $readyMessageWebsocket$ was included and initiated to $True$. This variable is key as it can become an additional condition in order to make a new request.

Once a new request is made, the state of that variable will be updated to $False$, to avoid making more requests. When the server has processed the request, it will then communicate to the game by running the procedure $WebSocketResponse$. In this procedure, adjustments to the speed of the virtual character will be implemented,

Algorithm 3 Input System Logic. In this algorithm *isInputbuttonPressed* is controlled by the input button being pressed. This is done to change input system.

Require: $isSensorbuttonPressed \in \{true, false\}$
Require: $inputSystem \in \{0, 1\}$
Require: $windowSize \in \mathbb{Z}^+$ ▷ $\mathbb{Z}^+$ represent all positive integers
Require: $LPitch \in \mathbb{Z}$ ▷ $\mathbb{Z}$ represent all integers
Require: $LRoll \in \mathbb{Z}$
Require: $RPitch \in \mathbb{Z}$
Require: $RRoll \in \mathbb{Z}$
  $recentSensorDataArray \leftarrow Queue()$ ▷ Array that will track latest sensor data
  $webSocket.connect()$ ▷ Initialize Websocket connection
  $readyMessageWebsocket \leftarrow True$
  while Game is Running **do**
    $recentSensorDataArray.Queue(LPitch, LRoll, RPitch, RRoll)$ ▷ Add data
    if $recentSensorDataArray > windowSize$ **then**
      $recentSensorDataArray.DeQueue()$ ▷ Remove oldest sensor data
    end if
    if $inputSystem = 0$ **then**
      $KATSDK.GetSpeed()$ ▷ Get KAT SDK speed and use that speed
    else if $inputSystem = 1$ & $readyMessageWebsocket = True$ **then**
      $webSocket.Send(recentSensorDataArray)$ ▷ Send data to Websocket
      $readyMessageWebsocket \leftarrow False$
    end if
  end while
  procedure WebSocketResponse($responseData$)
    $xVelocity = responseData.x$
    $yVelocity = responseData.y$
    $zVelocity = responseData.z$
    Update the character velocity according
    $readyMessageWebsocket \leftarrow True$
  end procedure

alongside the state change of $readyMessageWebsocket$ being set back to $True$, knowing that the Python server has completed its prior tasks.

3.6 Speed/Velocity Multiplier

Additionally, users can adjust their movement speed in the game to cater to personal preferences or specific game dynamics. Taller individuals may prefer higher speeds, and open-world users might also opt for speeds to explore expansive terrains with ease.

To accommodate for this, two additional sliders were added to the menu, "X Velocity Multiplier" and "Z Velocity Multiplier". The rationale behind having separate sliders for the Z-velocity and the X-velocity lies in their potential preference. This approach acknowledges that users might be content with their current speed along one axis while desiring an adjustment along the other. By providing distinct sliders, users gain finer control over each axis, enabling them to customize their experience precisely as they prefer.

These two sliders can be seen in Figure 3.3. Each of those sliders has the ability to go from 0.1 to 10, with both having a default value of 1. The current value of each of those sliders are taken and used as a constant to multiply the original velocity. This can be seen in the following equation:

$$adjustedZVelocity = originalZVelocity * zMultiplier \tag{3.6.1}$$

$$adjustedXVelocity = originalXVelocity * xMultiplier \tag{3.6.2}$$

3.7 Calibration and Offset

This part of the research game will investigate the causes of calibration issues, which resulted in unexpected direction of motion or drifting movements. To study this, the characters yaw rotation will be examined. $characterYaw$ determines the orientation of the virtual character. $bodyRotation$ is the yaw rotation sensor value taken from the KAT SDK (aka. the hip rotation), and $headsetYaw$ is the sensor value taken from the VR Headset. Additionally, $correctionYaw$ will be a calculated value based on the calibration logic, and $offsetYaw$ will be a value taken based on the slider value, which has a default value of 0.

$$characterYaw = bodyRotationYaw + correctionYaw + offsetYaw \tag{3.7.1}$$

3.7.1 Calibration Button (Sensor Calibration)

The purpose of calibration is to address situations where the user's movements deviate unexpectedly. This arises from the inherent functioning of sensors, where the initial rotation reference is established at device startup. Moreover, the headset's rotation value operates independently of the KAT Walk C body rotation value. Consequently, users may find themselves facing one direction while their character moves independently in another, leading to a disorienting experience. Calibration seeks to align these parameters and ensure a congruent user-character interaction, mitigating disorientation issues.

To tackle this issue, a variable called $correctionYaw$ is introduced. During the calibration process, the user is prompted to face forward, aligning the body sensor

with the headset sensor. This alignment calculates the disparity between the two sensor readings, which is then incorporated into the character rotation, $characterYaw$. The calibration procedure's logic is detailed in Algorithm 4.

Algorithm 4 Calibration Button Logic. In this algorithm $isCalibrationButtonPressed$ is controlled by whether the user clicked the calibration Button.

Require: $isCalibrationButtonPressed \in \{true, false\}$
Require: $headsetYaw \in \mathbb{R}$ ▷ $\mathbb{R}$ represents all real numbers
Require: $bodyRotationYaw \in \mathbb{R}$
  $correctionYaw \leftarrow 0$
  while Game Is Running **do**
    if $isCalibrationButtonPressed$ is $true$ **then**
      $isCalibrationButtonPressed \leftarrow false$
      $correctionYaw \leftarrow headsetYaw - bodyRotationYaw$
    end if
  end while

This logic is called in the game when the user presses the "Sensor Calibration Button" as well as when the game is first launched. To obtain the best calibration results upon launch, the user is requested to look straight forward.

3.7.2 Offset Slider (Body Rotation Slider)

While the calibration methodology solved the major issue of independence and movements in unexpected directions, another problem that arose was drifting. Upon investigation, it was revealed that the cause of this problem was related to how users secured themselves to the KAT Walk C. The design of the multiple straps occasionally led to misalignment, causing the body rotation on the KAT C to point slightly left or right relative to the actual hip rotation.

To address this challenge, a new variable named $offsetYaw$ is introduced. Users

have control over this variable through a slider accessible in the menu screen, as illustrated in Figure 3.3. This slider encompasses a range from -30 to 30 degrees, with a default value set at 0 degrees. When users encounter drifting to the left, they can increment the body rotation offset slider, while drifting to the right can be mitigated by decreasing the slider.

The combination of both the "Sensor Calibration" button as well as the "Body Rotation Offset" slider allows for a much more natural user experience.

Chapter 4

Datasets and Training

The following section provides an in-depth exploration of the project's data management, data collection, data processing, and data augmentation.

Altogether with data collection and other techniques used, the final dataset contained 108 minutes of labelled sparse motion capture data. This data can be broken down into the following:

- 57 minutes of STEPS data.
- 24 minutes of RSIDESTEPS data (Right Sidesteps).
- 24 minutes of LSIDESTEPS data (Left Sidesteps).
- 3 minutes of STANDING data.

4.1 Data Collection

The process of data collection involved employing a range of tools. In chapter 3, we discussed the research game being equipped with the capability to export sensor

data to a CSV format. Additionally, video recording tools like Open Broadcaster Software (OBS), Software (2017), was used to capture the game interface, allowing for the examination of the iteration value at designated instances of the recording. Moreover, a webcam feed was utilized to visually monitor the user's executed motion. Figure 4.1 depicts one of the video recordings which show the iteration number along with the webcam feed. Altogether, all techniques and software used were necessary in order to collect and render the data usable for training purposes.

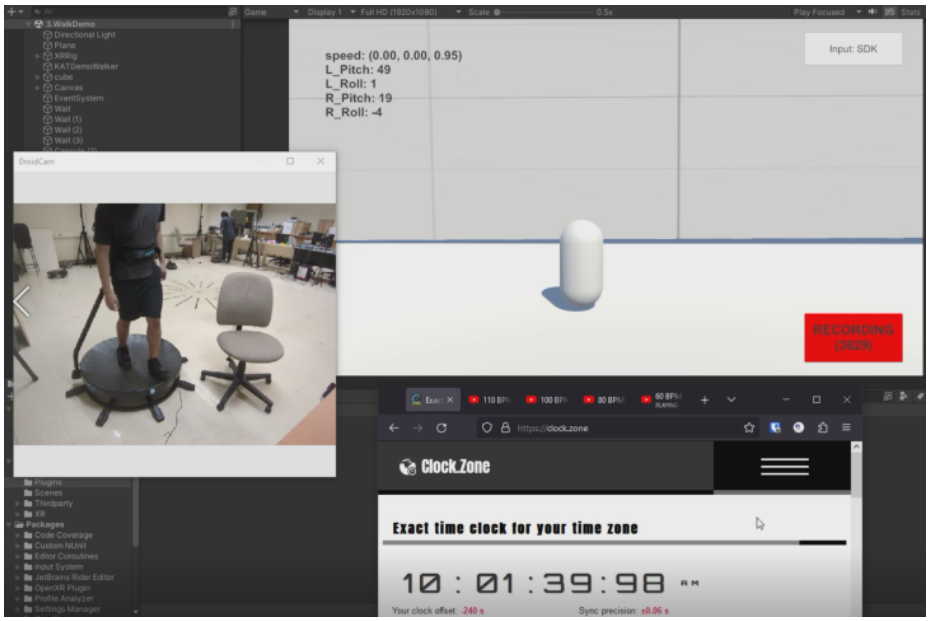

Figure 4.1: This figure represents an screenshot from one of the video recordings of a sensor recording session. Notice the webcam feed as well as the iteration number.

During the initial attempt at data collection, a repetitive motion was performed for approximately 60 seconds. However, a significant challenge arose when attempting to label the data, particularly defining the motion's speed. Due to the lack of any predefined parameters to objectively determine speed, the speed had to be labeled subjectively, relying heavily on intuitive judgment.

To address this issue, additional constraints were introduced during recording sessions. Firstly, a BPM (beats per minute) sound was played during the recording session to ensure that the executed motion closely matched the frequency of the BPM sound. Secondly, tape was placed on the treadmill to provide a visual target to the user for motion execution, enhancing consistency. Figure 4.2 illustrates the tape markers used for data collection. These two constraints significantly facilitated data processing and the objective labeling of the collected data.

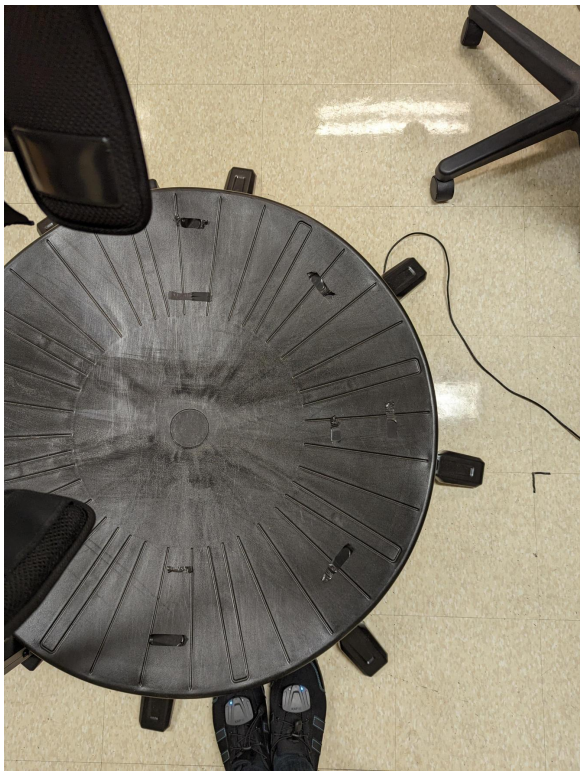

Figure 4.2: This figure shows the tape markers which represents a visual target for the foot during motion execution.

4.2 Data Processing

So far, the current state of the collected data is unsuitable for analysis. The data contains segments that must be discarded and lacks any form of labeling. Figure 4.3 offers an example of a processed file. This section will delve into the logic applied to transform raw data into a refined, processed format.

	A	B	C	D	E	F	G	H	N	O	P	Q	R	S	T
1	Timestamp	Iteration	L_Pitch	L_Roll	R_Pitch	R_Roll	X_Vel	Z_Vel	L_Pitch_Delta	L_Roll_Delta	R_Pitch_Delta	R_Roll_Delta	Class_Motion	Class_MotionType	Class_MotionSpeed
2	26-06-2023 10:01:15.119	2385	19	2	17	-3	0	0	0	0	0	0	STAND	SML	SLOW
3	26-06-2023 10:01:15.136	2386	19	2	17	-3	0	0	0	0	0	0	STAND	SML	SLOW
4	26-06-2023 10:01:15.159	2387	18	2	17	-3	0	0	-1	0	0	0	STAND	SML	SLOW
5	26-06-2023 10:01:15.177	2388	18	2	17	-3	0	0	0	0	0	0	STAND	SML	SLOW
6	26-06-2023 10:01:15.197	2389	18	1	17	-3	0	0.91	0	-1	0	0	STEPS	LAR	AVERAGE
7	26-06-2023 10:01:15.217	2390	18	2	17	-3	0	0.91	0	1	0	0	STEPS	LAR	AVERAGE
8	26-06-2023 10:01:15.236	2391	17	2	17	-3	0	0.91	-1	0	0	0	STEPS	LAR	AVERAGE
9	26-06-2023 10:01:15.256	2392	17	3	17	-3	0	0.91	0	1	0	0	STEPS	LAR	AVERAGE
10	26-06-2023 10:01:15.275	2393	19	3	17	-3	0	0.91	2	0	0	0	STEPS	LAR	AVERAGE
11	26-06-2023 10:01:15.295	2394	25	3	17	-2	0	0.91	6	0	0	1	STEPS	LAR	AVERAGE
12	26-06-2023 10:01:15.317	2395	33	3	17	-2	0	0.91	8	0	0	0	STEPS	LAR	AVERAGE
13	26-06-2023 10:01:15.337	2396	28	4	17	-2	0	0.91	-5	1	0	0	STEPS	LAR	AVERAGE
14	26-06-2023 10:01:15.354	2397	26	5	17	-2	0	0.91	-2	1	0	0	STEPS	LAR	AVERAGE
15	26-06-2023 10:01:15.375	2398	29	5	17	-2	0	0.91	3	0	0	0	STEPS	LAR	AVERAGE
16	26-06-2023 10:01:15.397	2399	24	5	17	-2	0	0.91	-5	0	0	0	STEPS	LAR	AVERAGE
17	26-06-2023 10:01:15.415	2400	22	4	17	-2	0	0.91	-2	-1	0	0	STEPS	LAR	AVERAGE
18	26-06-2023 10:01:15.435	2401	19	2	17	-2	0	0.91	-3	-2	0	0	STEPS	LAR	AVERAGE
19	26-06-2023 10:01:15.454	2402	18	1	17	-2	0	0.91	-1	-1	0	0	STEPS	LAR	AVERAGE
20	26-06-2023 10:01:15.476	2403	15	0	17	-2	0	0.91	-3	-1	0	0	STEPS	LAR	AVERAGE
21	26-06-2023 10:01:15.498	2404	14	0	17	-2	0	0.91	-1	0	0	0	STEPS	LAR	AVERAGE
22	26-06-2023 10:01:15.517	2405	12	1	17	-2	0	0.91	-2	1	0	0	STEPS	LAR	AVERAGE
23	26-06-2023 10:01:15.535	2406	12	2	17	-2	0	0.91	0	1	0	0	STEPS	LAR	AVERAGE
24	26-06-2023 10:01:15.555	2407	12	3	17	-2	0	0.91	0	1	0	0	STEPS	LAR	AVERAGE
25	26-06-2023 10:01:15.578	2408	14	6	17	-2	0	0.91	2	3	0	0	STEPS	LAR	AVERAGE
26	26-06-2023 10:01:15.599	2409	17	6	17	-2	0	0.91	3	0	0	0	STEPS	LAR	AVERAGE
27	26-06-2023 10:01:15.614	2410	19	6	17	-1	0	0.91	2	0	0	1	STEPS	LAR	AVERAGE
28	26-06-2023 10:01:15.636	2411	22	6	17	-1	0	0.91	3	0	0	0	STEPS	LAR	AVERAGE
29	26-06-2023 10:01:15.655	2412	28	5	17	-1	0	0.91	6	-1	0	0	STEPS	LAR	AVERAGE

Figure 4.3: This figure represents the output of the processed data.

4.2.1 Data Trimming

Data trimming refers to removing the rows in the data which do not reflect the motion execution. The rationale behind data trimming was that the recording session was initiated before getting on the treadmill. Data was also recorded when the user was getting off the KAT Walk C treadmill. This was caused by the user initiating the recording session at the desktop, a few steps away from the treadmill. Recall from data collection, for each datafile collected, there exist an accompanied recording which has a video feed of the user motion as well as the iteration number of what is

happening at that exact instance, (see Figure 4.1 for an example). Also recall that the datafile CSV also has the iteration value for each row, an example can be seen in Figure 4.4.

	A	B	C	D	E	F	G	H	I
1	Timestamp	Iteration	L_Pitch	L_Roll	R_Pitch	R_Roll	X_Vel	Z_Vel	Notes1
2	26-06-2023 10:00:27.417	0	20	5	18	-6			CUTOFF
3	26-06-2023 10:00:27.434	1	20	5	18	-6			CUTOFF
4	26-06-2023 10:00:27.456	2	20	5	18	-6			CUTOFF
5	26-06-2023 10:00:27.476	3	20	5	18	-6			CUTOFF
6	26-06-2023 10:00:27.494	4	20	5	18	-6			CUTOFF
7	26-06-2023 10:00:27.514	5	20	5	18	-6			CUTOFF
8	26-06-2023 10:00:27.535	6	20	5	18	-6			CUTOFF
9	26-06-2023 10:00:27.555	7	20	5	18	-6			CUTOFF
10	26-06-2023 10:00:27.574	8	20	5	18	-6			CUTOFF
11	26-06-2023 10:00:27.596	9	20	5	18	-6			CUTOFF
12	26-06-2023 10:00:27.616	10	20	5	18	-6			CUTOFF
13	26-06-2023 10:00:27.634	11	20	5	18	-6			CUTOFF
14	26-06-2023 10:00:27.659	12	20	5	18	-6			CUTOFF
15	26-06-2023 10:00:27.676	13	20	5	18	-6			CUTOFF
16	26-06-2023 10:00:27.694	14	20	5	18	-6			CUTOFF
17	26-06-2023 10:00:27.715	15	20	5	18	-6			CUTOFF
18	26-06-2023 10:00:27.734	16	20	5	18	-6			CUTOFF
19	26-06-2023 10:00:27.756	17	20	5	18	-6			CUTOFF
20	26-06-2023 10:00:27.780	18	20	5	18	-6			CUTOFF
21	26-06-2023 10:00:27.794	19	20	5	18	-6			CUTOFF
22	26-06-2023 10:00:27.815	20	20	5	18	-6			CUTOFF

Figure 4.4: This figure represents the RAW data output exported from the research game. Lots of processing needs to be done in order for the data to be used such as calculations and trimming.

Altogether, this becomes useful since the webcam feed can be used to identify when the actual motion was started. Upon identification, the video can be paused to determine the exact iteration value. Consequently, all rows preceding this iteration value can be safely eliminated from the processed data. Similarly, a comparable methodology can be employed to detect when the user concludes the motion, signifying that any rows with iteration values beyond this identified point should be excluded from the processed dataset. Figure 4.4 illustrates this process, with the column labeled $NOTES1$ indicating rows to be removed during datafile processing, marked with the value $CUTOFF$.

4.2.2 Data Labelling

The second essential aspect was data labeling. For a given sensor data reading or a window of sensor data readings, a systematic method was required to assign unique class labels to the data. The first step was to identify the motion the user was executing. This was extracted from the file name. The second step was to classify the type of motion being performed by the user. This involves distinguishing whether they are taking small steps or large steps. Fortunately, this was also extracted from the file name. The final feature needed was the motion speed, is the user travelling slow, average, or fast. This was obtained through a calculation which used the BPM (Beats per Minute) from the filename. In this context, class boundaries are introduced as slow, average, or fast, with their definitions based on specific BPM thresholds. For instance, if an interval is established from $V_{Slow,A}$ to $V_{Slow,B}$, any BPM falling within this range is categorized into the "SLOW" class. Similarly, this categorization process is extended to the "AVERAGE" and "FAST" classes based on their respective BPM intervals. The logic for motion speed classification can be seen in Algorithm 5.

Algorithm 5 Motion Speed Logic. This algorithm shows the logic on how the motion speed is calculated.

Require: $BPM > 0$ ▷ BPM value.
Require: $upperBoundSlow \geq 0$ ▷ Upper Bound Slow BPM
Require: $upperBoundAvg > upperBoundSlow$ ▷ Upper Bound Avg BPM
  if $BPM \leq upperBoundSlow$ **then return** $SLOW$
  else if $BPM \leq upperBoundAvg$ **then return** $AVERAGE$
  else if $BPM > upperBoundAvg$ **then return** $FAST$
  end if

4.3 Data Augmentation

Data augmentation refers to the ability to increase the training dataset by manipulating current data in order to create new synthetic data. This tactic is used for this project to increase the training dataset without needing to record new sensor readings. It was also used to create data for motions at speeds that were difficult to perform. The two types of data augmentation used in this project will be called "Double Speed" and "Half-Speed".

4.3.1 Double Speed

Doubling the speed involves manipulating the data to replicate a user performing the original motion at twice the speed. Since the sensor's polling rate remains constant, we can achieve this motion by skipping every second data point from our original dataset.

Lets assume we have a dataset X which represents our original motion. Since each data recording is formed by executing the same motion over a period of time, our dataset contains cycles. Lets assume our data has cycle length k. This means that for the first k elements in X (i.e $x_1, x_2, ..., x_k$) form a cycle. Now, if we select a subset of X by picking only the data points with even indices (i.e $x_2, x_4, .., x_k$), these values form a new thinned cycle (i.e data points are further away from each other) within this new dataset. This new cycle will have a length that is roughly half of our original cycle.

In essence, if the user simulates the same motion at double the speed, the research game would capture every other sensor data value from the original motion (i.e. the even indices). As a result of this manipulation, the new dataset will contain cycles

where the cycle length is half of the original length. This reduction in cycle length mirrors the user executing the motion at twice the original speed.

This double speed data can be seen in Figure 4.5, where the augmented data has very short cycles lengths (orange graph) for a given number of iterations, relative to the original data (blue graph). The logic for double speed can also be seen in Algorithm 6. A technical explanation on the concept of doubling speed is discussed in Appendix A.1.

Algorithm 6 Double Speed Data Augmentation. This algorithm shows the logic on how data augmentation was performed for double speed.

Require: $filePath$ ▷ File path to data file.
Require: $data \leftarrow ReadData(filePath)$
Require: $data.rows > 0$ ▷ Need the data to have 1 or more rows of data.
  $i \leftarrow 0$
  $evenRows \leftarrow []$
  $oddRows \leftarrow []$
  $doubleSpeedData \leftarrow []$
  for $i <$ *data.rows* **do** ▷ Loop through each data row
    $i \leftarrow i + 1$
    if i is even **then**
      $evenRows.Append(data[i])$
    else if i is odd **then**
      $oddRows.Append(data[i])$
    end if
  end for
  $doubleSpeedData \leftarrow Concatenate(evenRows, oddRows)$
    return *doubleSpeedData*

4.3.2 Half-Speed

Manipulation of the original dataset is required to achieve a new dataset that represent the motion at half the original speed. This manipulation involves creating synthetic data points between each pair of original data points. These synthetic points are

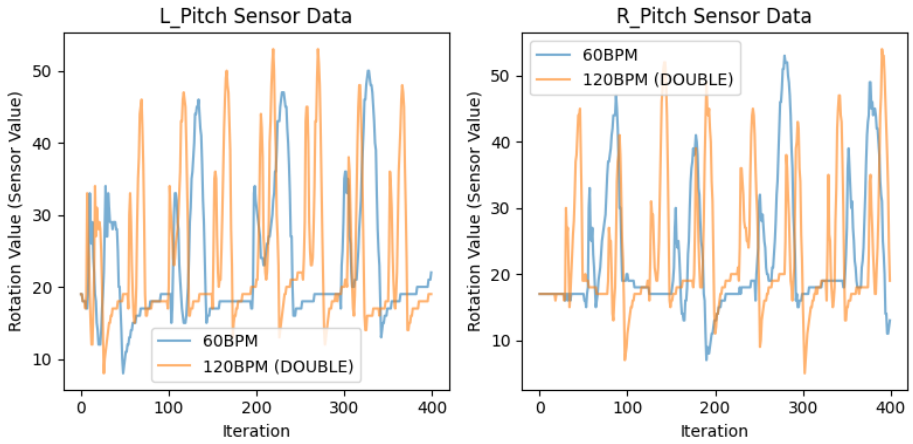

Figure 4.5: This figure represents the original data at 60BPM (BLUE) and the augmented data at 120BPM (ORANGE). Notice the augmented data have shorter cycles.

determined by taking the average of the two adjacent original data values, as expressed by the formula: $x_{i+0.5} = \frac{x_i + x_{i+1}}{2}$.

By adding these synthetic points to the original data effectively double the number of data points in the dataset, all while keeping the same number of executed motions. Consequently, the new dataset will contain cycles that are approximately twice as long as those in the original data, which results in the new dataset to simulate the execution of the motion at half the original speed.

Half-speed data can be seen in Figure 4.6, where the augmented data has very long cycles lengths (orange graph), relative to the original data (blue graph). The logic for half-speed can also be seen in Algorithm 7. A technical explanation on the concept of half-speed is discussed in Appendix A.2.

Algorithm 7 Half-Speed Data Augmentation. This algorithm shows the logic on how data augmentation was performed for half-speed.

Require: $filePath$ ▷ File path to data file.
Require: $data \leftarrow ReadData(filePath)$
Require: $data.rows > 0$ ▷ Need the data to have 1 or more rows of data.
  $i \leftarrow 0$
  $halfSpeedData \leftarrow []$
  for $i < data.rows - 1$ **do** ▷ Loop through each data row (except last)
    $i \leftarrow i + 1$
    $halfSpeedData.Append(data[i])$ ▷ Add real data
    $average \leftarrow Calculateaverage(data[i], data[i+1])$ ▷ Calc avg. between current and next data
    $halfSpeedData.Append(average)$ ▷ Add synthetic data
  end for
    return $halfSpeedData$

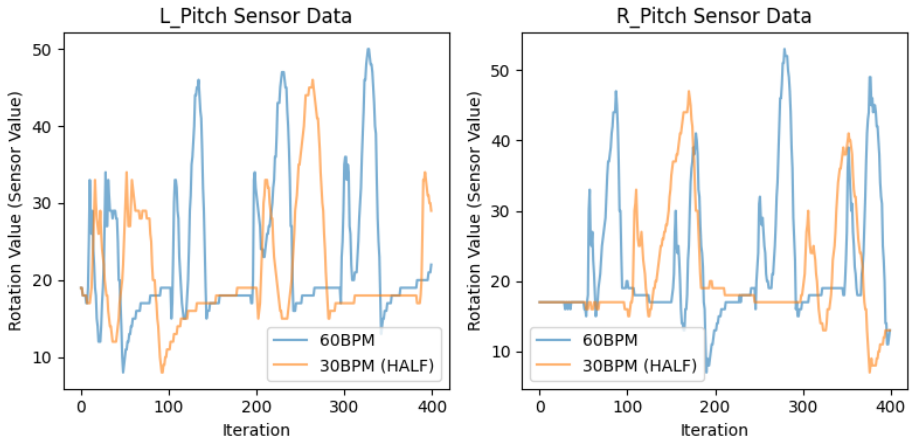

Figure 4.6: This figure represents the original data at 60BPM (BLUE) and the augmented data at 30BPM (ORANGE). Notice the augmented data have longer cycles.

Chapter 5

The KATNN Algorithm

One of the concerns of the original KAT C algorithm was its limitations, where some motions would not register user movements effectively, and in some cases, not recognized at all. Some of these motions include executing slower motions such as "sneaky steps" not effectively working. Another was taking left or right sidesteps not being recognized at all. This section formally introduce the new algorithm, KATNN, which predicts the trajectory of the character based on a window of the latest sensor data, while aiming to solve the aforementioned issues. KATNN was designed using machine learning and neural networks. The primary goal of the models is to leverage all available data to optimize the accuracy of its predictions.

KATNN will have up to 3 layers of classification. The initial layer, referred to as the motion layer, focuses on the predicted motion: is the user walking forward, taking sidesteps to the left, taking sidesteps to the right, or standing still? The second layer, referred to as the motion type layer, focuses on the step size: is the user taking small or large steps? The final layer, referred to as the motion speed layer, will emphasize on the speed of the motion: is the user moving slow, average, or fast?

One pivotal decision in formulating the algorithm involved the adoption of modular neural networks, with each neural network dedicated to addressing a specific problem within a particular segment of the algorithm. A few reasons for this approach include: (1) working with modular neural network made debugging and error correction much easier, (2) different scenarios and different layers required different inputs (e.g. executing sidesteps required different inputs compared to walking forward).

The three layers and the different neural networks used can be seen in Figure 5.1. Oval nodes in the figure represent a singular neural network, and the rectangle boxes represent all the different classification outputs for that neural network. Each layer will be explained in detail in the following sections.

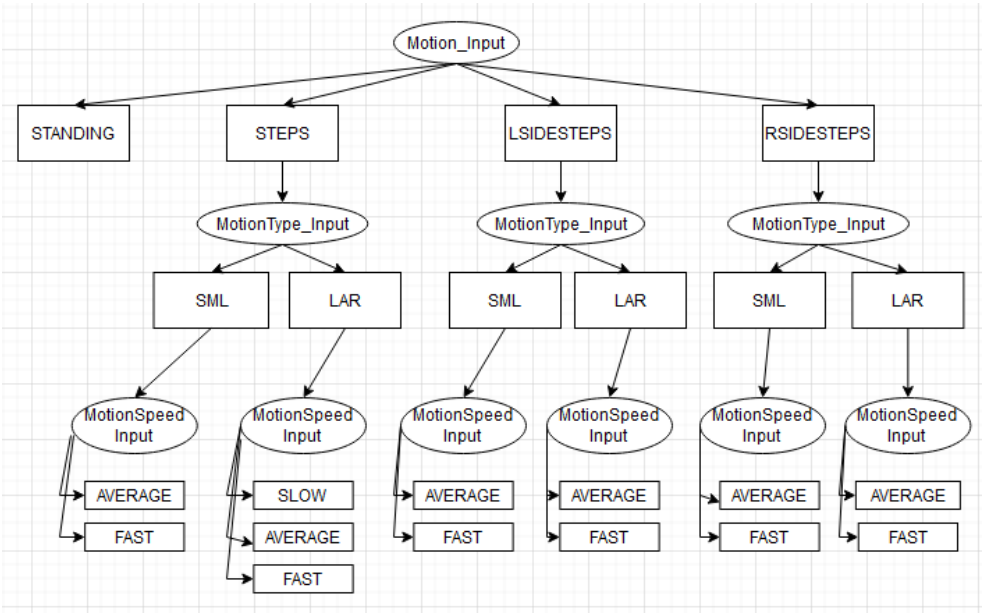

Figure 5.1: This figure represents the logic that the KATNN will follow, where the first layer is at the top and works its way down until it reaches a tail node.

5.1 Sum of Absolute Delta Sensor Readings (SADSR)

The concept of Sum of Absolute Delta Sensor Readings (SADSR) plays a pivotal role in the algorithm and is frequently referenced in the subsequent chapters. This function, when applied to a window of sensor data, calculates the total change in sensor readings. Notably, it treats the direction of sensor changes as irrelevant, meaning that changes of $+15$ or -15 are considered equivalent to 15.

Consider an example to calculate SADSR using the following sequential sensor readings: $12, 30, 20, 5$. The steps involved are as follows:

1. Calculate the change in sensor reading (delta) from the sequence: $\{18, -10, -15\}$. To do this, we take the current sensor reading and subtract it to the previous sensor reading ($delta_i = x_i - x_{i-1}$ where x_i is sensor reading at time i).

2. Take the absolute value of each element in step 1: $\{18, 10, 15\}$

3. Calculate the sum of all values from step 2 : $\{43\}$

Therefore, given those 4 sensor readings, the sensor has acquired a total absolute change of 43. In this context, it's important to note that there are four different sensor readings. Ideally, the algorithm would produce four SADSR values, with each value corresponding to a specific sensor, expressed as $SADSR_{LPitch}$, $SADSR_{LRoll}$, $SADSR_{RPitch}$, $SADSR_{RRoll}$.

What does this signify? It's important to recognize that SADSR represents the summation of deltas, effectively quantifying the extent of foot rotation within a given time interval. To illustrate, when the user is executing sidesteps to the left, it would likely incur substantial roll rotation in the left foot. Conversely, while moving forward,

pitch rotation would be more prominent. These variations are primarily influenced by both the manner in which the motion is executed and the curvature of the KAT Walk C surface.

5.2 Total Number of Sequences Above/Below a Threshold (TNSAT/TNSBT)

The Total Number of Sequences Above a Threshold (TNSAT) and the Total Number of Sequences Below a Threshold (TNSBT) are two closely related algorithms employed for estimating the number of cycles within a specified data window. This estimation is achieved by identifying a sequence of data points that occurs exactly once per cycle, typically in the form of peaks and troughs. When the data sequence corresponds to peaks, we apply the TNSAT algorithm. Conversely, when the data sequence corresponds to troughs, we apply the TNSBT algorithm. The logic used in both TNSAT and TNSBT is found in Algorithm 8.

5.3 Layer 1: Motion

Layer 1, referred to as "Motion," pertains to the specific movement the user is executing. In contrast to the original KAT C algorithm, which limited users to moving forward or remaining stationary, KATNN offers the capability to perform left sidesteps (class: LSIDESTEPS) for leftward movement and right sidesteps (class: RSIDESTEPS) for rightward movement. This layer holds a large importance within the algorithm, as it dictates the direction of the user's movement. Therefore, the

Algorithm 8 Total Number of Sequences Below/Above a Threshold (TNSBT/TNSAT). This algorithm shows the logic on how to calculate the number of sequences below/above a predefined threshold.

Require: $algorithmType \in \{TNSBT, TNSAT\}$ ▷ Determine which algorithm
Require: $sensorDataArray$ ▷ Array of Sensor Data.
Require: $sensorDataArray.length > 0$
Require: $thresholdValue$ ▷ Define a threshold value.
  $i \leftarrow 0$
  $sequenceCount \leftarrow 0$
  $aboveThreshold \leftarrow False$
  for $i < sensorDataArray.rows$ **do** ▷ Loop through each sensor value
    if $algorithmType = TNSAT$ **then** ▷ START OF TNSAT LOGIC
      if $sensorDataArray[i] \geq thresholdValue$ **then**
        if $aboveThreshold = False$ **then** ▷ If first value above threshold
          $sequenceCount \leftarrow sequenceCount + 1$ ▷ Add to count
          $aboveThreshold \leftarrow True$
        end if
      else if $sensorDataArray[i] < thresholdValue$ **then** ▷ Check below cutoff
        $aboveThreshold \leftarrow False$
      end if
    else if $algorithmType = TNSBT$ **then** ▷ START OF TNSBT LOGIC
      if $sensorDataArray[i] \leq thresholdValue$ **then**
        if $belowThreshold = False$ **then** ▷ If first value above threshold
          $sequenceCount \leftarrow sequenceCount + 1$ ▷ Add to count
          $belowThreshold \leftarrow True$
        end if
      else if $sensorDataArray[i] > thresholdValue$ **then** ▷ Check above cutoff
        $belowThreshold \leftarrow False$
      end if
    end if
  end for
    return $sequenceCount$

priority of this algorithm is to maintain a very high level of prediction accuracy for Layer 1. This emphasis is driven by the recognition that inaccuracies in this layer's predictions could result in a misalignment between the user's movement direction and the character's movement direction, potentially leading to discomfort.

For this neural network, the considered inputs encompassed the $SADSR$ values from all four sensors. These $SADSR$ values were chosen because they provide information about which foot is in motion (by detecting changes in rotations) and the direction of that foot's movement (by distinguishing between changes in roll and changes in pitch).

During the training process, the network was supplied with $SADSR$ data from each sensor, and the target classifications included STANDING, STEPS, LSIDESTEPS, and RSIDESTEPS. This approach yielded positive results, with the network effectively discerning boundaries for each of these classes. While the exact boundaries may not be precisely determined, there is a general understanding of how the network distinguishes between the classes:

- If all 4 $SADSR$ are 0, return "STANDING"
- Else If $SADSR_{LRoll}$ is very high, return "LSIDESTEPS"
- Else If $SADSR_{RRoll}$ is very high, return "RSIDESTEPS"
- Else If $SADSR_{LPitch}$ or $SADSR_{RPitch}$ is very high, return "STEPS"

Table 5.1 summarizes all of the Motion Neural Networks used in Layer 1. Here, input size denotes the data input shape required by the model.

Neural Name	**Parent Class**	**Input Size**	**Input Desc**	**Output Classes**
Motion1	None	1×4	$SADSR_{LPitch}$, $SADSR_{LRoll}$, $SADSR_{RPitch}$, $SADSR_{RRoll}$	STANDING, STEPS, LSIDESTEPS, RSIDESTEPS

Table 5.1: All Motion Neural Networks Used.

Furthermore given the significance of this layer, careful consideration was given to the reduction of the window size. Decreasing the window size would result in lower latency when detecting changes in motion. However, it was imperative to strike a balance, ensuring that the window size was not reduced excessively, as this could potentially lead to a decrease in prediction accuracy.

5.3.1 Tweaking Standing Sensitivity

It's worth noting that in the previous explanation of the neural network, it was mentioned that if all four $SADSR$ values are 0, the network would predict "STANDING." This is due to the training data provided to the neural network typically exhibited minimal to no variation during the sensor recording phase, as standing still generally results in stable sensor readings, which logically corresponds to a prediction of "STANDING."

However, it was later discovered that even when standing still, sensor values could change. One particular scenario is when there is oscillation. If the sensor readings fluctuate closely between two values while standing still, this oscillation can lead to an increase in $SADSR$ value, potentially causing the network to predict movement.

To address this issue, two distinct logic components were introduced to enhance prediction accuracy. The first logic involves setting a minimum $SADSR$ threshold. If the calculated $SADSR$ value falls below this predefined threshold, the algorithm overrides the prediction and assigns "STANDING" as the motion state. This threshold essentially represents the minimum level of motion required to trigger a movement prediction.

The second logic component, known as the "minimum difference," evaluates the

difference between the maximum and minimum sensor readings, expressed as $diff = maxSensorReading - minSensorReading$. If this calculated difference is smaller than the defined minimum threshold, the prediction is again overwritten as "STANDING."

Together, these two logic components played a significant role in reducing the occurrence of false movement predictions generated by the initial network when standing, ensuring more accurate motion detection.

5.4 Layer 2: Motion Type

Layer 2, known as "Motion Type," is designed to determine the scale of the motion being executed, i.e. whether it falls into the category of a small step (class: SML) or a large step (class: LAR). Originally, the plan included three distinct classes, with the third class intended to represent medium steps (class: MED). However, upon closer examination of the data, it became evident that there was no clear distinction between medium and large steps. As a result, these two classes were merged into a single class, labeled as "large steps" (LAR).

To differentiate between each motion type, it's crucial to consider the motion predicted in Layer 1 as different motions require distinct inputs. It is also important to note that the KAT surface is inclined, meaning that as you move further from the center, the foot experiences significant incline, resulting in variations in sensor readings. This can be examined by looking at the data collected. For instance, when the motion is "STEPS," as depicted in Figure 5.2, a discernible pattern emerges where pitch values peak higher for "LAR" (large) steps compared to "SML" (small) steps. Conversely, for "LSIDESTEPS" or "RSIDESTEPS," as illustrated in Figure

5.3, there's a noticeable distinction in the maximum and minimum roll values between large and small sidesteps. Similarly, there's a significant difference in the maximum pitch value when performing large sidesteps versus small" sidesteps. These variations in sensor values help classify the motion type accurately.

Table 5.2 summarizes all of the Motion Type Neural Networks used in Layer 2. Here, input size denotes the data input shape required by the model.

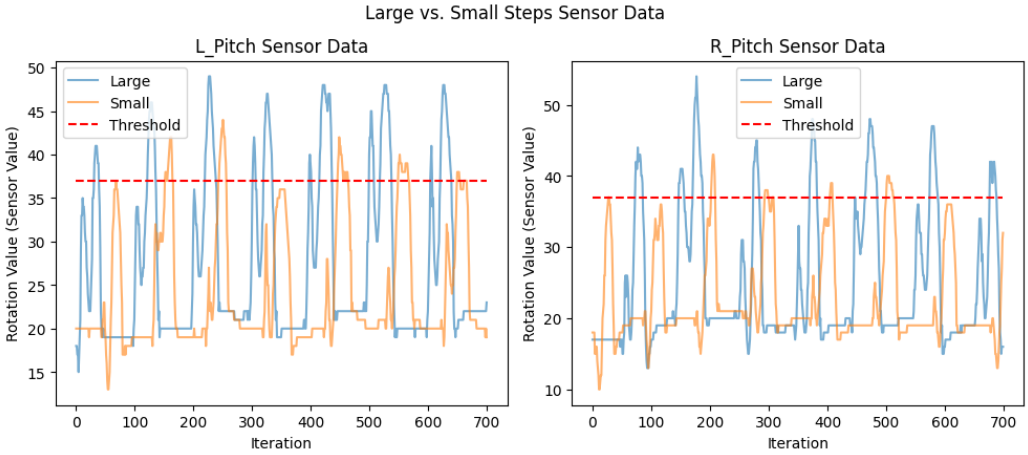

Figure 5.2: This figure represents the sensor readings between large steps (BLUE) vs. small steps (ORANGE). Notice that large steps yield larger maximum pitch values.

5.5 Layer 3: Motion Speed

The final layer within the algorithm, motion speed, serves to determine the speed of the ongoing motion. The classes in this layer are categorized as average (class: AVERAGE) or fast (class: FAST). However an additional class, slow (class: SLOW), is introduced when taking large steps, as this is the most common motion. This additional class provides a broader spectrum of speed options for the user, increasing

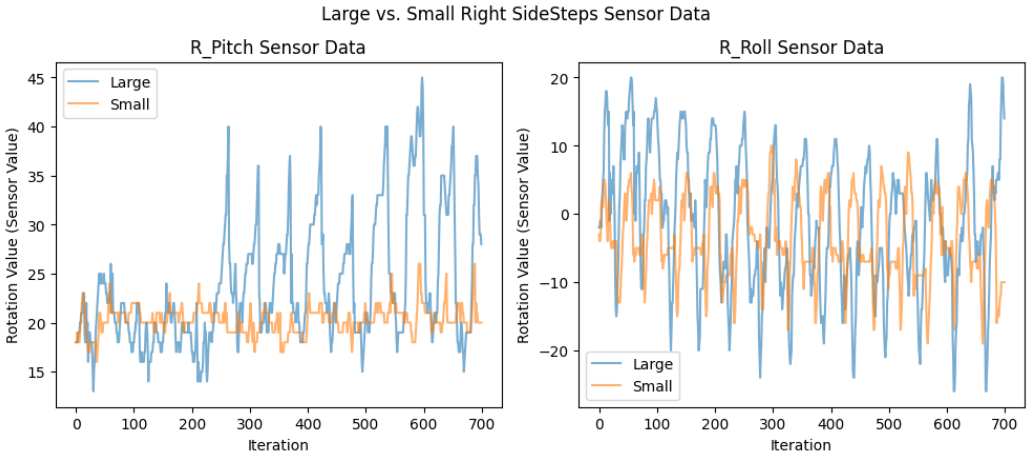

Figure 5.3: This figure represents the sensor readings between large right sidesteps (BLUE) vs. small right sidesteps (ORANGE). Notice that large right sidesteps yield larger peaks and minimums in Roll Rotations.

their immersion.

The initial discriminant for predicting each class relies on utilizing the relevant $SADSR$ value. The underlying concept is that the faster an individual performs a motion, the higher the user $SADSR$ value tends to be. This correlation arises because rapid motion execution within a constant window length results in more rotations, thereby increasing the $SADSR$ value. A visual representation of this relationship can be observed in Figure 5.4, showing a tendency towards a higher $SADSR$ values for faster classification classes.

The second differentiating factor is using TNSAT or TNSBT values. Counting such sequences can estimate the number of cycles, which reflects the speed of motion execution. For instance, in the left image of Figure 5.2, the blue graph displays peaks occurring around 40 before decreasing again. This information allows the threshold to be set at 37 and count the total sequences that surpass 37 to calculate the TNSAT. Similarly, in Figure 5.3, troughs in the left roll can be observed at around -15 to

Neural Name	**Parent Class**	**Input Size**	**Input Desc.**	**Output Classes**
MotionType1	STEPS	1×1	max(max(LPitch), max(RPitch))	SML, LAR
MotionType2	LSIDESTEPS	1×3	$min(LRoll)$, $max(LRoll)$, $max(LPitch)$	SML, LAR
MotionType3	RSIDESTEPS	1×3	$min(RRoll)$, $max(RRoll)$, $max(RPitch)$	SML, LAR

Table 5.2: All Motion Type Neural Networks Used.

-20 before increasing. This means the threshold can be set around -10 and count all sequences that dip below -10 to calculate TNSBT. These calculations would yield an estimate on the number of cycles.

The underlying principle is that a higher number of sequences corresponds to a higher frequency, indicating faster motion execution and predicting a faster motion speed. However, a notable challenge associated with this approach pertains to the consistency of the executed motion. As illustrated in Figure 5.2, the selection of an appropriate threshold pitch value involves a delicate balance. Setting a threshold too high may result in certain sequences not being included in the count, while a threshold set too low may lead to an excessive number of sequences being included.

Table 5.3 summarizes all of the Motion Speed Neural Networks used in Layer 3. Here, input size denotes the data input shape required by the model.

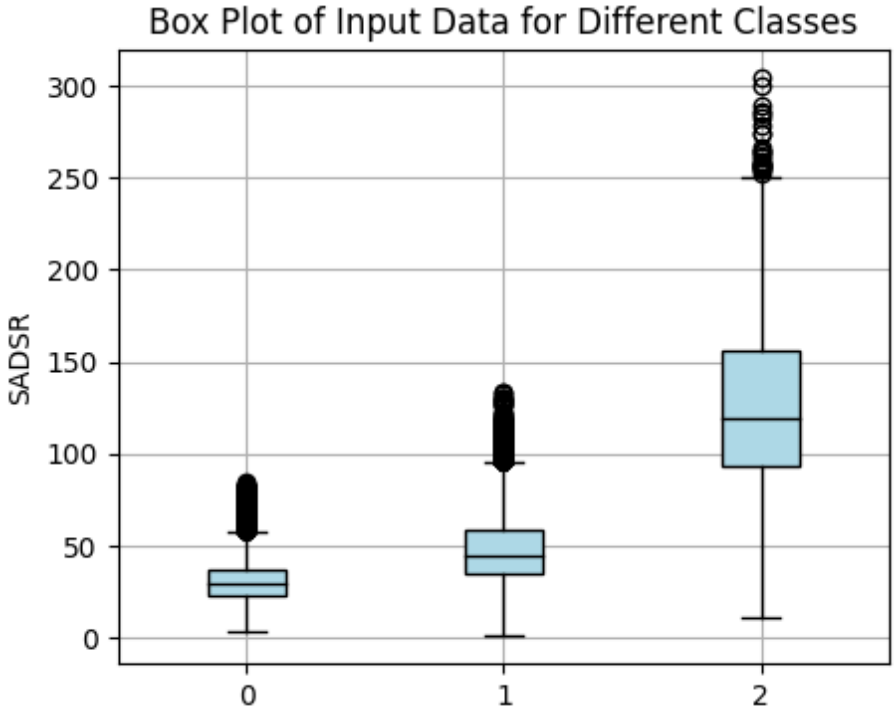

Figure 5.4: This figure represents the different motion speed classes (0 = SLOW, 1 = AVERAGE, 2 = FAST) and the $SADSR$ values (y-axis). Notice that on average the $SADSR$ values are higher for faster classifications.

5.6 Post Prediction Logic

Following the completion of predictions, they are presented as a string in the format "A-B-C," where A represents the motion prediction, B denotes the motion type prediction, and C signifies the predicted motion speed. This string, "A-B-C," is then employed to access the velocity configuration, as shown in Figure 5.5.

After retrieving the velocity settings from the configuration file, the prediction process for the user's velocity and trajectory based on the provided sensor data is concluded. The final step involves sending this predicted velocity and trajectory back to the client, which in this case, was the research game.

Neural Name	**Parent Class**	**Input Size**	**Input Desc**	**Output Classes**
MotionSpeed1	STEPS-SML	1×2	$SADSR_{LPitch}$+ $SADSR_{RPitch}$, $TNSAT(LPitch)$+ $TNSAT(RPitch)$	AVERAGE, FAST
MotionSpeed2	STEPS-LAR	1×2	$SADSR_{LPitch}$+ $SADSR_{RPitch}$, $TNSAT(LPitch)$+ $TNSAT(RPitch)$	AVERAGE, FAST, SLOW
MotionSpeed3	LSIDESTEPS-SML	1×2	$SADSR_{LRoll}$, $TNSBT(LRoll)$	AVERAGE, FAST
MotionSpeed4	LSIDESTEPS-LAR	1×2	$SADSR_{LRoll}$, $TNSBT(LRoll)$	AVERAGE, FAST
MotionSpeed5	RSIDESTEPS-SML	1×2	$SADSR_{RRoll}$, $TNSBT(RRoll)$	AVERAGE, FAST
MotionSpeed6	RIDESTEPS-LAR	1×2	$SADSR_{RRoll}$, $TNSBT(RRoll)$	AVERAGE, FAST

Table 5.3: All Motion Speed Neural Networks Used.

5.7 Optional: Unused Additional Logic

In this section, optional information is presented regarding additional logic that was integrated into request processing but was subsequently disabled or left unused. The rationale behind these decisions is explained below.

5.7.1 Dynamic Velocity

An issue that was taken into consideration pertained to two different motions that were predicted to be the same motion speed class, but slightly different execution speed. To address this, the concept of dynamic velocity was contemplated but eventually deactivated, as discussed below.

Dynamic velocity, in this context, refers to the idea of adjusting one's velocity

Figure 5.5: This figure represents the velocity configuration file used in this project.

dynamically according to a specific input parameter. In this scenario, the input parameter would be the "SADSR" value. The premise behind this concept is that higher a "SADSR" value would indicate faster user execution of the motion.

This would mean the new velocity would be in the form of the following equation, where k represents the sensitivity constant (higher k yields higher additional velocity).

$$DynamicVelocity = PredictedVelocity + k * SADSR \tag{5.7.1}$$

The challenge with this implementation stemmed from the fact that the "SADSR" value varied throughout the execution of a single motion. For instance, consider the act of taking a step: the "SADSR" may be higher during the initial phase of a step, which involves lifting the foot, pushing it forward, and placing it down. However, during the later phase, where the foot is retracted back to the center, a smaller "SADSR"

may be recorded. These differences in "SADSR" values resulted in different dynamic velocities, even though a consistent motion speed was expected. This variation in velocity caused discomfort during testing. As a result, the utilization of dynamic velocities was disabled during the final testing phase, although the logic remains in the code, with a variable in the configuration set to "False" to deactivate it.

Chapter 6

Results & Limitations

Now with the algorithm and the game being defined, the performance of KATNN is the next focus. This section delves into the training results of the Neural Network, the feedback gathered from testing it within the virtual environment, an evaluation of KATNN's performance on an additional user, and an exploration of the algorithm's limitations.

6.1 Neural Network Results & In-Game Testing

The outcomes of the trained models are presented in Table 6.1. Accuracy columns will denote the proportion of data that the model predicted correctly. Additionally, loss columns will denote the measure of how far the model's predictions are from the actual values.

Notably, the performance of the Layer 1 model stands out, achieving a training accuracy of 96.1%. Initially, a higher accuracy of around 97% was attained using the original window size of 26 data rows (or half a second of data). However, the

window size for Layer 1 was systematically reduced until a noticeable decline in accuracy was observed. Given that Layer 1 plays a pivotal role in determining the user's movement direction, maintaining high accuracy was imperative for the algorithm's success. Aside from the first layer, both Layer 2 and Layer 3 also demonstrated commendable performance, with the least accurate model achieving a training accuracy of 81.6% using default window size of 26. While it might have been possible to enhance prediction accuracy by using a larger window size, the trade-off was increased latency.

Neural Name	**Parent Class**	**Training Loss (%)**	**Training Accuracy (%)**	**Validation Loss (%)**	**Validation Accuracy (%)**
Layer 1: Motion					
Motion1	Motion	10.3%	96.1%	10.21%	96.2%
Layer 2: MotionType					
MotionType1	STEPS	37.1%	82.3%	37.0%	96.2%
MotionType2	LSIDESTEPS	28.5%	87.3%	26.8%	87.8%
MotionType3	RSIDESTEPS	34.4%	84.7%	33.6%	85.2%
Layer 3: MotionSpeed					
MotionSpeed1	SML-STEPS	21.9%	90.6%	21.8%	90.8%
MotionSpeed2	LAR-STEPS	37.6%	83.9%	37.5%	84.2%
MotionSpeed3	SML-LSIDESTEPS	18.5%	92.1%	18.7%	91.9%
MotionSpeed4	LAR-LSIDESTEPS	20.7%	90.7%	20.8%	90.6%
MotionSpeed5	SML-RSIDESTEPS	40.9%	81.6%	40.6%	82.1%
MotionSpeed6	LAR-RSIDESTEPS	22.9%	89.3%	23.0%	89.3%

Table 6.1: The Results of Our Neural Network Models.

During testing, a notable improvement was apparent through the increased base speed for movement, as the original KAT C algorithm felt slow while navigating the

virtual environment. Another positive aspect observed during testing was maneuvering around objects. In the game, there are obstacles such as capsules and cubes that the user can collide with. With the original KAT C implementation, when a collision occurred, the only option was to rotate 90 degrees, move forward, rotate back 90 degrees, and continue forward. This felt very counterintuitive. However with KATNN, users could simply sidestep to avoid the obstacle and then continue moving forward, resulting in a much more natural and user-friendly experience.

Additionally, KATNN excelled in recognizing slower movements, such as the "sneaky step" motion. In this motion, the expected output sequence is movement, standing, and then movement again. With the original KAT C algorithm, it often registered as movement, standing, and standing, causing the last part of the motion cycle to be ineffective. In contrast, KATNN accurately detected movement, standing, and movement, enhancing the overall user experience.

6.2 External User Testing & Analysis

When involving another individual in the use of the KAT Walk C, a few steps were taken before their participation. First, the input mechanism names in the research game were altered. The KAT C algorithm was renamed "YANKEE," while the KATNN name was changed to "TANGO." This adjustment was made to ensure that the user remained unaware of which implementation was developed by the researcher, thus eliminating potential bias. Subsequently, the user performed a standing motion, and data collection of their standing data readings was conducted. The results of this data collection can be observed in Table 6.2. Notably, the other user exhibited higher standing readings for all rotations, with the exception of right roll rotation.

	My Data (Standing)	**Other Data (Standing)**	**Deltas (+/-)**
L_Pitch	18	21	3
L_Roll	3	5	2
R_Pitch	17	23	6
R_Roll	-2	-9	-7

Table 6.2: My Data vs. External User Standing Data Comparison.

Additional data collection was conducted, instructing the user to perform large steps and large sidesteps. Figures 6.1 and 6.2 illustrate the variations in sensor readings between the my data and the external user's motions.

Upon comparing the large step data, one notable difference is that the peak pitch values in the other user's motion do not reach the same peaks as my execution of motion. This discrepancy could potentially pose an issue in Layer 2, where KATNN may predict a small step instead of a large one. A similar issue arises when examining the sidestep data. In the left roll (LRoll) graph, it is evident that the minimum peak value is significantly higher in the other user's motion compared to my motion. Again, this could lead to KATNN erroneously predicting a small step. Nevertheless, Layer 1 performed well, accurately predicting the user's intended direction, and allowing the user to move in additional directions, which provided a better experience for them.

6.3 Assessing Limitation

KATNN, while offering numerous advantages, also exhibits certain limitations that warrant discussion. The primary limitation pertains to variations in the execution of the same motion from different individuals may lead to incorrect predictions. As previously discussed, Layer 2, Motion Type, determines whether a motion is a large

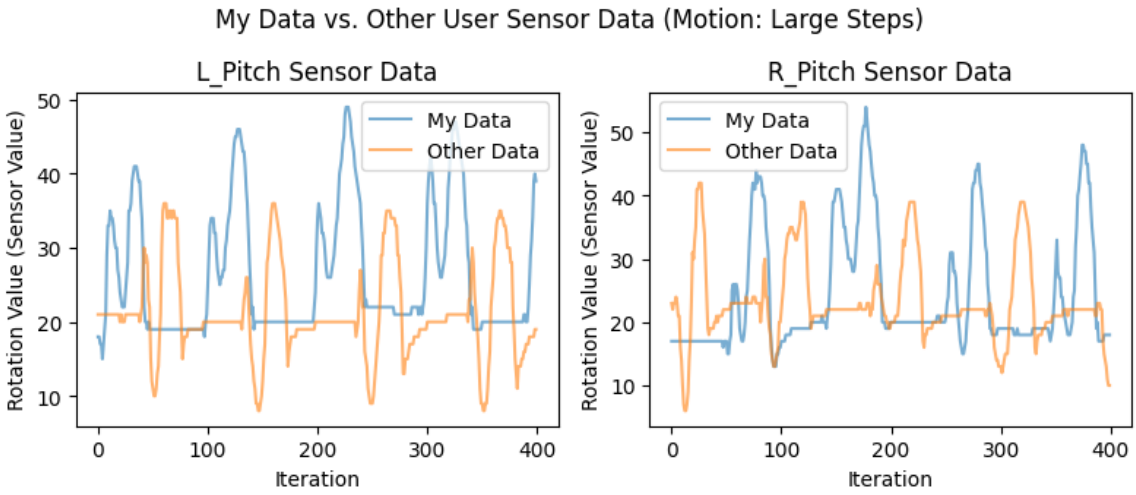

Figure 6.1: This figure represents the sensor data readings between my data (BLUE) and other person data (ORANGE) when performing large left sidesteps.

step or a small step by analyzing the peak sensor values. In practice, individuals performing what appears to be a small step could execute it differently by lifting their foot more, resulting in a higher peak sensor value. This deviation from the model's training data could lead to a false prediction of a large step, illustrating the model's limitations when applied to different users.

A second noteworthy constraint is latency. It's important to remember that the algorithm relies on a window of sensor data, which includes the most recent sensor reading along with the preceding ones. If a user transitions to a different motion, the algorithm may not detect this change until the previous sensor data aligns with the new motion, leading to a delay and lag in recognizing the new motion. In the worst-case scenario, this latency can be as long as 0.52 seconds.

The final noteworthy constraint of KATNN is the issue of rotating your body resulting in wrongful classification. When a user rotates their body on the KAT

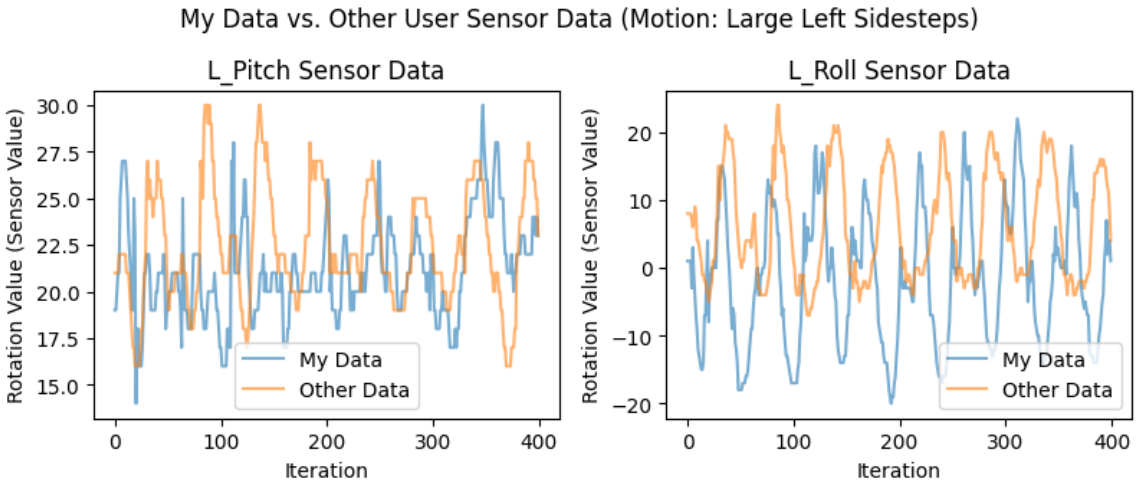

Figure 6.2: This figure represents the sensor data between sensor readings between my data (BLUE) and other person data (ORANGE) when performing large left sidesteps.

treadmill, this will result in taking small steps to the side to readjust the orientation of their feet. These small steps will yield wrongful classifications of sidesteps from KATNN leading to a discrepancy between the user's motion and the virtual character's motion.

Chapter 7

Alternative Attempts and Related Work

In this section, relevant research conducted in the context of this project and related to the KAT Walk C will be discussed. Some of this research was integrated into the project, while other aspects proved to be less effective than initially expected, resulting in their eventual archiving. Additionally, research conducted in connection with the KAT Walk C will be explored.

It's worth noting that the optional logic discussed in chapter 6 remains visible and present in the codebase. In contrast, the topics covered in this chapter have either been archived or studied in other projects.

7.1 LSTM Neural Network Approach

LSTM (Long short-term memory) Neural Networks belong to a class of recurrent neural networks designed for effectively handling time-dependent data. They find applications in various domains, such as predicting stock prices and forecasting weather conditions, due to their ability to capture long-range patterns within sequences. In this project, the data in question comprises sequences of sensor rotation data, where each row corresponds to a specific moment in time, creating a time-series problem. The concept here was to use the model to predict velocity. To achieve this, both the current velocity and the sensor data were inputted into the model to forecast future velocity.

Figure 7.1 showcases the training results, which demonstrate a promising 100% training accuracy. However, upon closer examination of Figure 7.2, discrepancies become evident. The blue graph illustrates the actual velocity, which varies over time, while the predicted velocity from the LSTM network, depicted in orange, remains constant. This discrepancy raises concerns about the model's predictive capabilities.

```
Epoch 1/20
7178/7178 [==============================] - 48s 7ms/step - loss: 3.3740e-04 - acc: 1.0000 - val_loss: 1.3114e-06 - val_acc: 1.0000
Epoch 2/20
7178/7178 [==============================] - 49s 7ms/step - loss: 2.6110e-05 - acc: 1.0000 - val_loss: 2.1315e-07 - val_acc: 1.0000
Epoch 3/20
7178/7178 [==============================] - 47s 7ms/step - loss: 2.7550e-07 - acc: 1.0000 - val_loss: 3.8979e-09 - val_acc: 1.0000
Epoch 4/20
7178/7178 [==============================] - 47s 7ms/step - loss: 7.6651e-09 - acc: 1.0000 - val_loss: 1.0301e-09 - val_acc: 1.0000
Epoch 5/20
7178/7178 [==============================] - 46s 6ms/step - loss: 1.0312e-09 - acc: 1.0000 - val_loss: 1.3160e-09 - val_acc: 1.0000
Epoch 6/20
7178/7178 [==============================] - 46s 6ms/step - loss: 4.3741e-10 - acc: 1.0000 - val_loss: 1.0442e-09 - val_acc: 1.0000
```

Figure 7.1: This figure represents the output during LSTM training, notice that the training is 100% accurate during the first round of training.

The puzzling disparity between the promising 100% training accuracy and the terrible predictions is in need of an explanation. To comprehend this, let's revisit

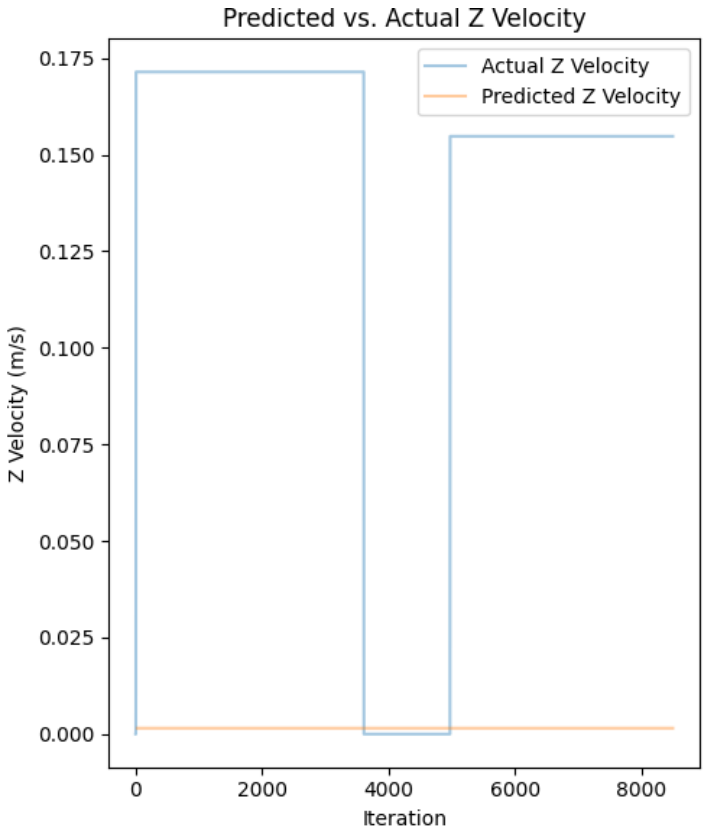

Figure 7.2: This figure represents the actual velocity of the user (BLUE) vs. the predicted velocity using the LSTM model (ORANGE).

the data collection and labeling section. The dataset comprises multiple files, with each file representing a single motion. Consequently, within a given file, the velocity remains constant since it corresponds to a consistent motion. During training, it seems that the Neural Network learned to extract the initial velocity from the input and, rather naively, forecasted future velocity as identical to the current velocity. While this strategy yielded flawless results in the training phase, it ultimately proves to be a woeful prediction approach in real-world scenarios.

Though LSTM networks still hold potential with different strategies and approaches, they were abandoned in the context of this implementation and were

archived.

7.2 Issues of Introducing Backsteps

A suggestion arose regarding the implementation of a backstep feature within the algorithm, considering its existing capabilities of moving forward, left, and right. The initial step involved executing the backstep motion while simultaneously collecting sensor data, which proceeded without complications. The subsequent challenge lay in data analysis: how could this new motion be seamlessly integrated into the model?

Upon analyzing the data, it became apparent that distinguishing backsteps from forward steps was not straightforward using the $SADSR$ values in Layer 1, as both types yielded similar high $SADSR(LPitch)$ and $SADSR(RPitch)$ values. A potential solution emerged in Layer 2, specifically within the motion type classification. A new class labeled $BSTEPS$ could be introduced to represent backsteps. Figure 7.3 visually highlights the distinctive characteristic of backsteps, particularly in the left graph, where backsteps exhibit significantly lower peaks. Thus, the idea emerged to incorporate logic that detects backsteps based on the magnitude of this peak, allowing for accurate recognition of this unique motion.

While technically feasible, there are significant concerns regarding the potential negative impact on accuracy and the severity of incorrect predictions when adding backsteps ($BSTEPS$) to the algorithm. This issue becomes evident in the right graph of Figure 7.3, where performing steps with the right foot does not exhibit the same distinct low peak as seen with the left foot. This lack of clear differentiation between normal and back steps would likely result in a decrease in accuracy.

Another critical concern pertains to the severity of incorrect predictions. In previous scenarios, when Layer 2 made an incorrect classification, it often had minimal consequences; for instance, predicting a small step instead of a large one simply led to slightly different user movements. However, with the introduction of backsteps, the stakes are higher. A wrongful classification might lead a user to perform a backstep but be incorrectly classified as a normal step, significantly impacting the user's immersion and experience.

Although there may still be potential for incorporating backsteps into the algorithm using a completely different approach, this idea was abandoned and archived for this implementation.

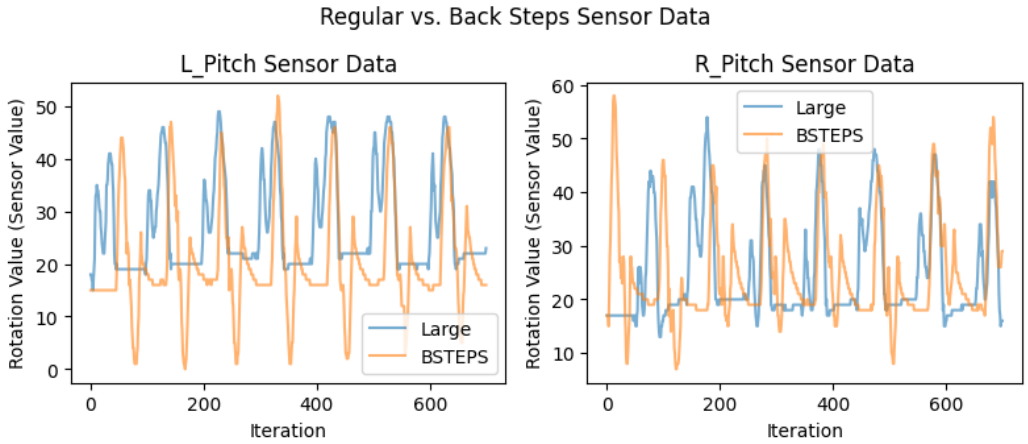

Figure 7.3: This figure represents the pitch sensor readings when executing a regular step vs. back steps.

7.3 Estimating Position of Foot on KAT Walk C

A project was undertaken in a course that centered around estimating the position of the foot on the KAT Walk C treadmill using the concept of interpolation. This

algorithm relies on estimating a position based on the three closest key-points and computing a weighted average (Matira (2023a)). The project's concept involved analyzing sensor readings while taking into account the curvature and incline of the KAT Walk C surface, which resulted in varying sensor readings across different areas of the surface. However, a significant constraint of this project was that accurate predictions could only be made if the user kept their feet on the surface, meaning they had to slide their feet instead of lifting them to walk naturally. Due to this limitation, the project was not further pursued after the course was completed, as the focus of the algorithm shifted toward accommodating more natural movements.

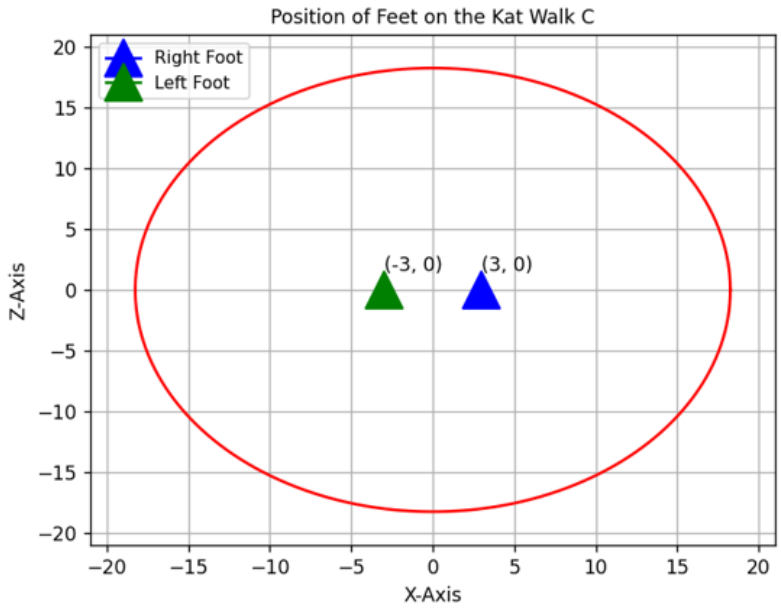

Figure 7.4: This figure represents a sample diagram that would have the ability to track the position of the foot on the KAT Walk C surface in real time based on interpolation.

Chapter 8

Conclusion & Future Work

8.1 Conclusion

This report has introduced an alternative input mechanism for the KAT Walk C, known as KATNN, which has demonstrated significant success in areas where the original KAT C implementation faced challenges. KATNN excels in registering movements in various directions, including the capability to accurately capture sidesteps. Moreover, it exhibits enhanced performance in capturing slower, subtle motions, such as "sneaky steps".

However, it is crucial to acknowledge the constraints of this algorithm. Like many machine learning solutions, its performance is contingent on the quality of its training data. This implies that the algorithm excels in motions on which it was trained extensively but may not perform as well for movements it has not encountered. Another limitation pertains to latency, as the algorithm relies on a window of sensor data. Consequently, users transitioning between motions may not see this reflected in the virtual game until as much as 0.52 seconds after initiating the new motion in

the worst-case scenario.

Throughout this project, numerous challenges were encountered, with two major hurdles standing out. These included the initial difficulty of extracting sensor data from the SDK and the complexities arising from handling sparse sensor data. Additionally, the performance of the neural networks did not meet expectations. However, through in-depth problem analysis and investigation, alternative solutions were devised and ultimately proved effective.

In summary, this project began with the daunting task of working with sparse sensor data that initially presented challenges. However, through continuous brainstorming, the development of innovative strategies, a significant learning process, rigorous data collection, and thorough data analysis, valuable insights were gradually extracted from these limited sensor readings. This experience underscores the boundless potential of this approach and should serve as a foundation for future students, researchers, and professionals to build upon, with the goal of further enhancing the VR experience for users.

8.2 Future Work

While the algorithm achieved several capabilities, such as executing sidesteps for lateral movement and accurately recognizing slower motions such as "sneaky steps", it also exhibits clear limitations that leave room for improvement.

8.2.1 Implementation of Previous/Related Work

Several potential directions for future development have emerged, some of which were hinted at in the earlier discussion of related work. In the context of LSTM models, there's room for exploring alternative methods that weren't initially considered. Questions arise about the sufficiency of existing data and whether additional data collection could enhance model performance. Additionally, the possibility of introducing noise to the data to prevent constant velocity predictions throughout a file warrants investigation.

Regarding the concept of backsteps, it was contemplated as a late addition to the project, requiring a complete overhaul of the algorithm. However, the absence of distinct traits during data analysis presented a significant challenge. Exploring this avenue further would necessitate additional research and a potentially different approach.

8.2.2 Foot on Surface Algorithm

An intriguing avenue of exploration lies in the development of an algorithm capable of discerning whether a foot is in contact with the KAT Walk C surface. To contextualize this idea, recall the constraints of the foot position estimation project, as detailed in Matira (2023a). Lifting one's foot during movement was prohibited due to the potential for inaccurate position estimations, leading to erroneous trajectory predictions.

The innovation here would involve creating an algorithm capable of determining the foot's contact status with the surface. Such an algorithm could signal the position estimation system to perform calculations exclusively when the foot is in contact with

the surface. This advancement would revolutionize the user experience, allowing for natural walking (including foot lifting) while simultaneously providing accurate foot position estimations and velocity data. In essence, it would enable trajectory predictions without the need of machine learning. It's plausible that, with this idea, such algorithms could outperform the KATNN algorithm.

8.2.3 KATNN Improvements

One suggestion to improve the KATNN experience is minimizing the discrepancy when the user is turning their body. One way to do this is to feed another attribute of sensors to the KATNN algorithm, the body yaw rotation. Feeding this rotation value will allow to calculate the difference between the maximum and minimum rotation. We can then use logic in our code to say that if the difference is bigger then some threshold value, to overwrite any prediction to a standing classification. This will allow users to turn without experiencing movement in the virtual environment.

Additionally, potential future enhancements could involve investigating the development of a more universally applicable model. This endeavor might necessitate the inclusion of user-specific parameters, such as individual user height, to fine-tune the network and optimize velocity predictions. The question arises: could the implementation of a calibration tool be feasible? This tool would involve instructing a user to perform a series of specific motions for a brief period, collecting relevant data, and dynamically generating a model tailored to their unique characteristics.

8.2.4 Calibration Improvements

Regarding the persistent issues of drifting and calibration, an alternative and user-friendly solution beyond the methods discussed in this report is proposed. This solution involves the introduction of an additional virtual environment, referred to as the "loading environment". Within this environment, users are transported to a distinct virtual world where they are presented with a prominent arrow. Their objective is to align their headset and hips with this designated arrow's orientation. Once users achieve the proper alignment, they can proceed to walk forward. The system then calculates their path, monitors any deviations from the intended trajectory, and calculates the necessary rotational adjustments to maintain users on the correct path. This approach offers a more intuitive and precise calibration method, effectively eliminating the reliance on trial-and-error adjustments via sliders in the game menu.

Bibliography

Ciathyza. 2021. Gridbox Prototype Materials. https://assetstore.unity.com/packages/2d/textures-materials/gridbox-prototype-materials-129127 Last accessed May 31, 2023.

KATVR. 2020. KAT Walk C Resources. https://www.kat-vr.com/pages/support-detail?id=6e02bc5de38c4a3ba2edd9d8f7096ba0&name=KAT%20Walk%20C Last accessed May 1st, 2023.

Joseph J. LaViola. 2000. A Discussion of Cybersickness in Virtual Environments. *SIGCHI Bull.* 32, 1 (jan 2000), 47–56. https://doi.org/10.1145/333329.333344

Kenneth Matira. 2023a. Repository: kat-walk-c-estimating-position. https://github.com/JKen0/kat-walk-c-estimating-position Last accessed April 10th, 2023.

Kenneth Matira. 2023b. Repository: vr-kat-project-game-research. https://github.com/JKen0/vr-kat-project-unity/ Last accessed August 30th, 2023.

Kenneth Matira. 2023c. Repository: vr-kat-project-python-research. https://github.com/JKen0/vr-kat-project-python/ Last accessed August 30th, 2023.

Open Broadcaster Software. 2017. Welcome to OBS Studio's documentation! https://docs.obsproject.com/ Last accessed June 3rd, 2023.

UnityDocumentation. 2021. Unity User Manual 2021.3 (LTS). https://docs.unity3d.com/2021.3/Documentation/Manual/index.html Last accessed May 31, 2023.

Wikipedia. 2007. Aircraft principal axes. https://en.wikipedia.org/wiki/Aircraft_principal_axes Last accessed August 28, 2023.

Appendix A

Data Augmentation: Technical Explanation

To illustrate this approach, we will investigate a single cycle of a sine wave, divided into 20 evenly spaced points. The corresponding sine function will be defined as $f(t) = \sin\left(\frac{2\pi t}{19}\right)$, where t represents iteration value, where $0 \leq t \leq 19$, and the output $f(t)$ will depict a normalized sensor reading at time t. The sequence of values $f(0)$, $f(1)$, $f(2)$, ..., $f(19)$ generates a single cycle of the sine graph, as illustrated in Figure A.1.

A.1 Double Speed Logic

The concept underlying double-speed logic involves the observation that doubling the speed of an exact motion results in sensor readings exhibiting a nearly twofold rate of change at each constant interval. This is achieved by reorganizing the initial data sequence as follows: first, gather all the even indexed values together (i.e. f(0), f(2),

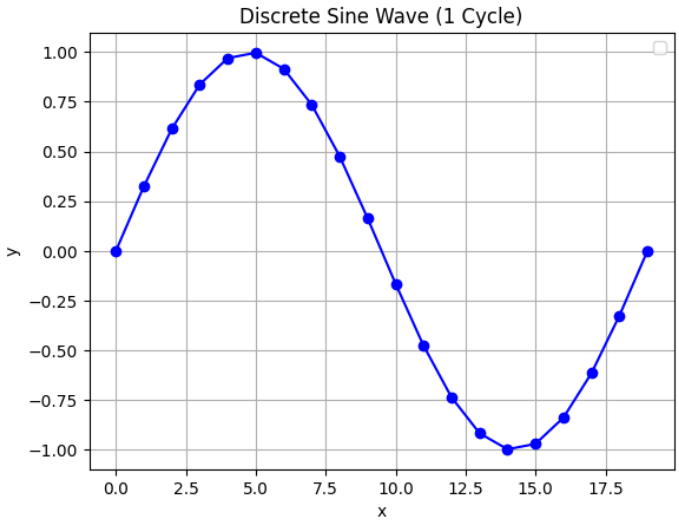

Figure A.1: This figure illustrates a single cycle of a sine wave divided into 20 discrete points.

... f(18)), which will be called S_{even}. Similarly, gather all the odd indexed values together (i.e. f(1), f(2), ..., f(19)), which will be called S_{odd}. Finally, concatenate all values from S_{even} followed by all values from S_{odd}. This results in the rearranged sequence $f(0)$, $f(2)$, ..., $f(18)$, $f(1)$, $f(3)$, ..., $f(19)$, which can be seen in Figure A.2. Notably, this rearrangement shortens the cycle length by a half, indicating that the user requires only half the time to execute the motion, implying that the user must be moving at double the speed.

However, this methodology rests on certain assumptions. First, it assumes that the underlying data exhibits cyclical behavior; without this fundamental assumption, the concatenation of even and odd data loses its meaningful interpretation. Since the same motion was executed repeatedly during data recordings, this came inherently with the data collected. Second, the effectiveness of this approach depends on cycle lengths that are not excessively small. For instance, if the original cycle had a length of 4, implementing this method would result in a new cycle with a length of 2, which

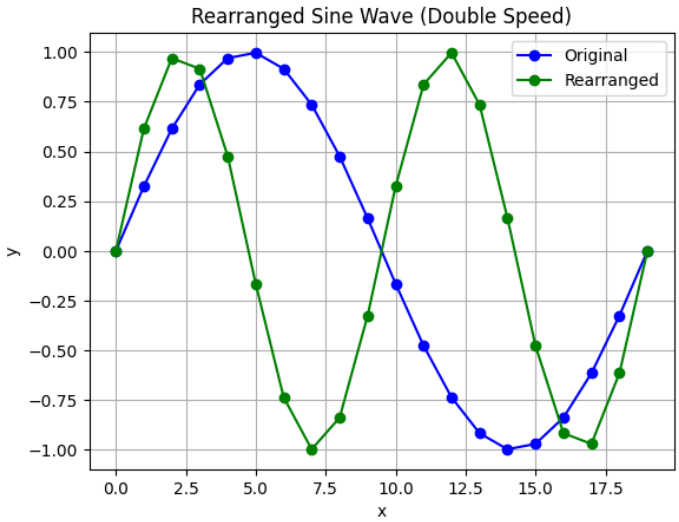

Figure A.2: This figure illustrates the rearrangement of the original sine graph, grouping all odd values together followed by all even points together. This arrangement simulates two sparser cycles.

may not accurately represent the true underlying cycle. This was not an issue with the data collected as cycle lengths ranged from 25 to 70.

A.2 Half-Speed Logic

The concept of half-speed logic centers on the creation of synthetic data to extend the length of the original cycle. This concept is grounded in the assumption of a consistent time interval between sensor readings, where a slower execution of motion leads to a halving of the rate of change in sensor readings due to the reduced motion speed. Synthetic data takes the form of an average between the current sensor reading and the subsequent reading, expressed by the formula $s(t) = \frac{f(t)+f(t+1)}{2}$. In the resulting sequence, each original data point, apart from the initial sensor reading, is paired with a synthetic value. This pairing results in the following sequence: $f(0)$, $s(0)$,

$f(1)$, $s(1)$, ..., $s(38)$, $f(39)$. This new sequence can be seen in Figure A.3. Observe that when $x = 20$, the new sequence has only completed half of a cycle. This indicates that the cycle length has effectively doubled, implying that the motion is executing at half the speed of the original motion.

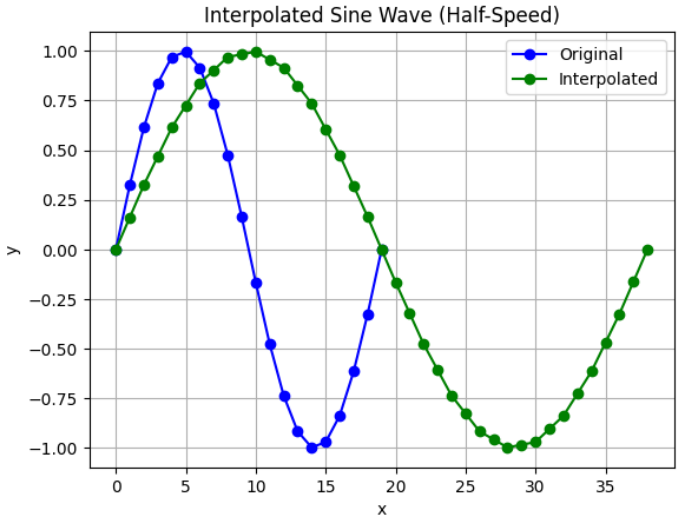

Figure A.3: This figure depicts the addition of synthetic data points between each original point to create a single cycle. This inclusion of points nearly doubles the cycle's length compared to the original.

Appendix B

Additional Information

- Knowledge regarding Deep Learning and Neural Networks was acquired from taking SEP 740.
- Knowledge regarding computer animations and exposure to sparse sensor techniques and motion capture techniques was acquired from taking CAS 737.
- Github link to the Python Server Code:

  https://github.com/JKen0/vr-kat-project-python/, Matira (2023c)
- Github link for the compiled game:

  https://github.com/JKen0/vr-kat-project-unity/, Matira (2023b)
- A link to the Estimating Position of Foot algorithm can be found here:

  https://github.com/JKen0/kat-walk-c-estimating-position, Matira (2023a)
- You can take a look each of the repositories and look at the commit history to see how the code evolved over time, and commits that were made that added or removed functionalities over time.

- The window size in Python is defined as 25 but in our Unity game it is defined as 26. The reason for this is that Unity will provide 26 rows of raw data, but Python will need 25 rows of delta data, so that 26 rows of raw data gets converted to 25 in Python.

- Layer 2 and Layer 3 Neural Networks use a window size of 25. This means it bases its predictions on sensor readings from up to 0.52 seconds ago.

- Layer 1 Neural Network used to use a window size of 25, but was reduced to 15 to reduce the delay in the event of a change in motion. This reduces the delay from 0.52 to 0.32 seconds.

- The reason why using KATNN may result in a delay in the first prediction is it needs 26 rows of sensor data in order to make the first prediction. Which can cause a latency of up to 0.52 seconds.

- In the python code, in the folder "4-PROCESSED-DATA" folder you may notice the folders "TEST2" and "TRAIN2." These folders hold the data for training/validation, where originally, "TRAIN2" was for training and "TEST2" was for validation. However, model performance improved a lot when we combined both datasets and used a function to split the combined dataset.

- 80% of our dataset was used for training, while the remaining 20% was used for validation.

- Two types of loss functions were used for our models, binary cross-entropy for models with only 2 prediction classes, and sparse categorical cross-entropy for

models with more then 2 prediction classes. Additional Information about the loss functions can be found here: https://keras.io/api/losses/probabilistic_losses/

- Directory "1-RAW-VIDEO" contains a few video recordings of the sensor recording sessions.

Appendix C

Code Installation and Information

C.1 Setting up Python Server

1. First Install Python version 3.6
2. Once installed install the following Python packages:
    - pip install pandas
    - pip install numpy
    - pip install scikit-learn
    - pip install tensorflow
    - pip install asyncio
    - pip install websockets
    - pip install openpyxl
3. Open the project in Visual Studio Code

4. Go to "./python-servers/webSocketServer_Sync.py" and open the script.

5. When the script is open, on the top right of Visual Studio Code, there exists a play button. Click it.

6. Our WebSocket server should now be running locally on port 3003!

C.2 Setting up Compiled Research Game

1. First, install Visual Studio 2022 and the following addons:

   - .NET desktop development (NECESSARY)
   - Game development with Unity (NECESSARY)

2. Secondly, make sure the Python WebSocket server is running on port 3003. If not refer to previous section.

3. When server is running, locate "./game-files/" and run test-kat-project application. This will launch the game. Game should be loaded on your desktop.

4. On your game application, on the top right you should see a button called MENU, this will open the menu which will show you a button called "INPUT: SDK", click the button so it says "INPUT: KATNN". This will change the input system from the SDK implementation to our Neural Network implementation. Then close the menu.

5. Turn on Meta Quest 2, and connect Meta Quest 2 to your desktop via Airlink/CableLink.

6. When connected, launch to the desktop and on the taskbar should be a unity icon which will bring you to launch the game on your Meta Quest.

7. Once done, you should see the game environment in the lens of your Meta Quest 2 using the KATNN input system.

C.3 Code Information

- Python Code Folder Structure Explanation:
  - **config folder**: This folder contains parameters and functions used throughout the project, this is so that the parameter will only need to be changed here rather than EACH file.
  - **NeuralNetwork folder**: this folder contains all of the pre-processed logic/data, the training logic for our neural networks, and the trained neural networks.
  - **processed-training-data folder**: This folder contains all of the training data in the form of raw data (data extracted from the unity game) and processed data (altered data after running a python script).
  - **python-servers folder**: This contains all python servers.
  - **graphs folder**: this folder contains all of the graphs used for the report utilizing matplotlib.
- All of the logic regarding KATNN can be found in ./python-servers/webSocketServer_Sync.py. Every other file and folder is used for training and data collection, and data analysis.

- Regarding the unity game, all logic of everything explained and modified by me can be found in KATXRWalker.cs

Appendix D

KAT SDK and Issues

The KAT SDK was obtained after contacting their support team and showing interest in creating a VR game which utilizes the KAT Walk C. Their SDK provides two versions, one for Unity game development, and the other one for Unreal Engine. Oddly enough, the Unreal Engine SDK does not have support to extract extra info data (like foot sensor rotation, and body raw), but the unity SDK does. For that reason, the project went the direction of creating a VR game in Unity. In the Unity SDK, you have the ability to extract extra information from many of their devices including KAT Walk C2, KAT Walk Loco S, and KAT Walk Mini S. While the logic is in place in the SDK, I can't confirm whether it actually works or not.

D.1 Issues

Despite being provided the SDK, one massive issue was the foot rotation sensor data was not being pulled correctly. I know this is true because I could see on KAT Gateway (software that comes with the KAT Walk C) the sensor data was changing but when

calling the function in the SDK, all sensors values just returned as 0. After weeks of communication with their support and development email, they provided little to no insight, as they mentioned that "developing a game with KAT WALK C does not require the sensor data." While this is generally true, it was not true for this research project. After weeks of attempts to resolve the issue, no progress was made, until one day, out of desperation, I decided to randomly make modifications to the Extra Info Class structure. Figure D.1 represents the original code from KAT SDK and Figure D.2 represents code modified in order to get the sensor data to return correctly. For whatever reason, modifying the data structure by removing some attributes made some sensor data display properly, and modifying the data structure again by moving other attributes made the other sensor data display properly. Because of that, we utilize 2 different classes which represent the left foot sensor data and the right foot sensor data.

Another common issue was the body rotation Yaw value would not work on some devices. When I connected the KAT Walk C to my laptop, foot sensor rotations would work, but the body yaw rotation would not register and always remain 0. This occurred for both the SDK and the KAT Gateway software. However when connecting the KAT Walk C to the lab desktop computer, the foot sensor rotations, and the body yaw rotation would work, for both the SDK and the KAT Gateway. I tried downloading different versions of KAT Gateway on the laptop but none ended up working. Because of this, I made sure to not make any changes to KAT Gateway on the lab computer since all testing will be done on the lab computer.

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
4 references
public struct extraInfo
{
    0 references
    public extraBaseSDKInfo baseSDKinfo;
    0 references
    public uint L_Status;
    0 references
    public float L_Pitch;
    0 references
    public float L_Roll;
    0 references
    public uint R_Status;
    0 references
    public float R_Pitch;
    0 references
    public float R_Roll;
    0 references
    public uint Hall_Status;
};
```

Figure D.1: This figure represents the original code to the Extra Info Class. Using this class results in the sensor data not being pulled.

```
// MY EDITS - USE THIS DATA STRUCTURE TO GET LEFT ROLL AND LEFT PITCH
[StructLayout(LayoutKind.Sequential, Pack = 1)]
4 references
public struct extraInfo1
{
    0 references
    public extraBaseSDKInfo baseSDKinfo;
    0 references
    public float L_Pitch;
    0 references
    public float L_Roll;
    0 references
    public float R_Pitch;
    0 references
    public float R_Roll;
};

// MY EDITS - USE THIS DATA STRUCTURE TO GET RIGHT ROLL AND RIGHT PITCH
[StructLayout(LayoutKind.Sequential, Pack = 1)]
4 references
public struct extraInfo2
{
    0 references
    public extraBaseSDKInfo baseSDKinfo;
    0 references
    public uint L_Status;
    0 references
    public float L_Pitch;
    0 references
    public float L_Roll;
    0 references
    public float R_Pitch;
    0 references
    public float R_Roll;
```

Figure D.2: This figure represents the modified code to the Extra Info Class in order to correctly fetch the correct sensor data.