PUNCH!

Kim Hyeong-Seok

August 1, 2025

Abstract

PUNCH! is a mobile application designed to facilitate combat-style exercise using only a smartphone, enabling users to engage in physical activity even in confined spaces. This paper details the development and testing of PUNCH!, focusing on the utilization of smartphone sensors to track movement and classify punch types. The application employs advanced techniques such as orientation estimation, noise reduction, and deep learning models to achieve accurate motion detection and classification. The results demonstrate the feasibility of using smartphones for effective combat-style exercise, addressing common barriers to physical activity in modern lifestyles.

1 Introduction

Modern individuals often struggle to maintain a healthy lifestyle due to a lack of physical activity, chronic stress, and the demands of a busy daily routine. In particular, time and space constraints, financial burdens, and psychological barriers associated with exercise contribute to the growing difficulty of engaging in regular physical activity. To address these challenges, we developed PUNCH!, a mobile application that enables users to easily engage in combat-style exercise using only a smartphone, even within confined spaces.

The development and testing of this application were conducted using a MacBook Pro (M1, 2021) and an iPhone 14 Pro, which will hereafter be referred to as the smartphone. All data collection and code execution were performed exclusively on these two devices, and the application has not been tested on platforms other than iOS.

2 Methods

2.1 Utilization of Smartphone Sensors

2.1.1 Measurement of Acceleration via Calibration and Differencing

To isolate acceleration caused solely by device movement, a calibration process was performed by maintaining the smartphone in a standard orientation—perpendicular to the ground with the screen facing the user—for one second. The accelerometer readings obtained during this calibration were used as a reference, and subsequent acceleration measurements were compared against this baseline to estimate changes due exclusively to motion. This approach assumes gravitational acceleration as a constant bias, allowing for a simplified implementation that focuses on relative changes in acceleration.

However, gravitational force continuously affects accelerometer readings, and variations in device tilt or orientation can significantly alter the measured values. As such, ignoring the influence of gravity imposes a limitation on the accuracy of this method. The approach lacks the precision necessary to isolate pure translational acceleration. Consequently, we decided to incorporate a mathematically and physically grounded method for tracking the device's orientation to account for gravitational effects more accurately.

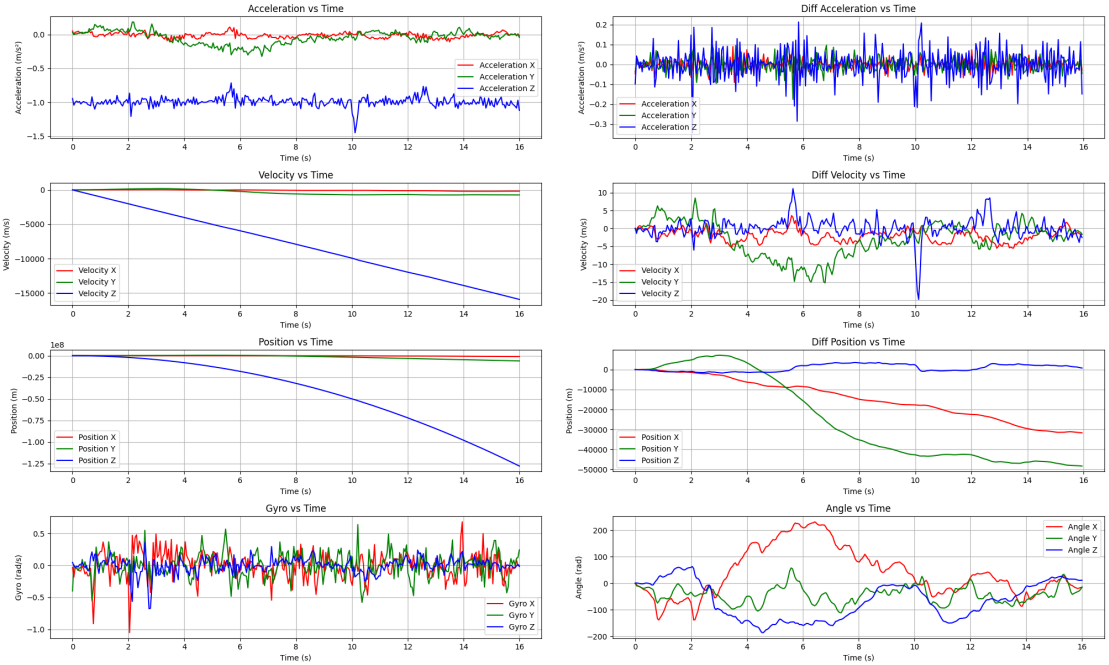


Figure 1: Charts comparing raw data and differentiated data

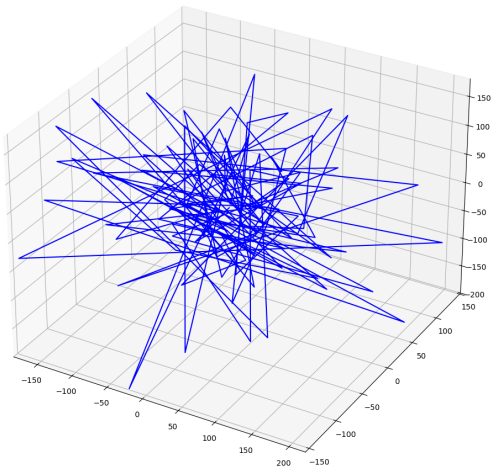


Figure 2: An image of 3-dimension path estimated with differentiated data

2.1.2 Orientation Estimation Using the Accelerometer

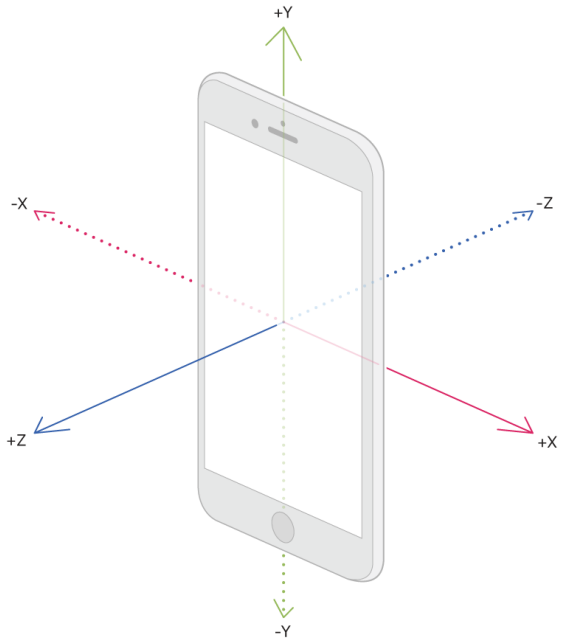


Figure 3: Smartphone accelerometer axis orientation

The coordinate system of the smartphone's accelerometer follows the right-hand rule, as illustrated in Figure 3. The output values of the sensor are normalized by the gravitational acceleration $9.8\,\mathrm{m/s^2}$, and thus each axis returns a value in units of g (an equivalent to gravitational force).

When the smartphone is held vertically such that the xz-plane is parallel to the ground and the screen is facing the user, the expected accelerometer reading $\vec{g}$ due to gravity is:

$$\vec{g} = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^\top$$

This vector $\vec{g}$ serves as the reference vector, representing the direction of gravity in the device's coordinate system under ideal upright conditions.

Let $\begin{bmatrix} x & y & z \end{bmatrix}^\top$ be the first accelerometer reading obtained after calibration. By comparing this measurement to the reference vector, we can estimate the initial orientation of the device. Before doing so, the accelerometer vector must be normalized to ensure consistency in magnitude.

The orientation is estimated using a sequence of rotation matrices about the x-, y-, and z-axes:

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}$$

$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix}$$

$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Assuming that the orientation angles θ_x, θ_y, and θ_z represent rotations about the respective axes, the rotated reference vector becomes:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_y(\theta_y) \cdot R_x(\theta_x) \cdot R_z(\theta_z) \cdot \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} = R_y(\theta_y) \cdot R_x(\theta_x) \cdot \begin{bmatrix} \sin\theta_z \\ -\cos\theta_z \\ 0 \end{bmatrix} = R_y(\theta_y) \cdot \begin{bmatrix} \sin\theta_z \\ -\cos\theta_x\cos\theta_z \\ -\sin\theta_x\cos\theta_z \end{bmatrix}$$

Multiplying out the rotations yields:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\theta_y \sin\theta_z - \sin\theta_x \sin\theta_y \cos\theta_z \\ -\cos\theta_x \cos\theta_z \\ -\sin\theta_y \sin\theta_z - \sin\theta_x \cos\theta_y \cos\theta_z \end{bmatrix}$$

Since the accelerometer cannot determine rotation about the y-axis (yaw), we assume $\theta_y = 0$, simplifying the expression to:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sin\theta_z \\ -\cos\theta_x \cos\theta_z \\ -\sin\theta_x \cos\theta_z \end{bmatrix}$$

From this simplified form, we can compute the orientation angles as follows. The x-component yields:

$$\theta_z = \arcsin(x)$$

To isolate θ_x, we divide the third component by the second:

$$\frac{z}{y} = \frac{-\sin\theta_x \cos\theta_z}{-\cos\theta_x \cos\theta_z} = \tan\theta_x \quad \Rightarrow \quad \theta_x = \arctan\left(\frac{z}{y}\right)$$

Let the initial orientation of the device be denoted by $\begin{bmatrix} \psi_0 & \theta_0 & \phi_0 \end{bmatrix}^\top$, corresponding to rotations around the x-, y-, and z-axes, respectively. Using the above method, we can estimate the initial orientation as:

$$\begin{bmatrix} \psi_0 \\ \theta_0 \\ \phi_0 \end{bmatrix} = \begin{bmatrix} \arctan\left(\frac{z}{y}\right) \\ 0 \\ \arcsin(x) \end{bmatrix}$$

This method allows for a simple and effective estimation of the device's initial pitch and roll, based on gravitational acceleration and assuming no initial yaw rotation.

```python
def numerical_integration_with_range(self, integrand, start, end):
    result = [0.0]
    for idx, i in enumerate(range(start + 1, end + 1)):
        result.append((integrand[i] + integrand[i - 1]) * (self.time[i] -
            self.time[i - 1]) / 2 + result[idx - 1])
    return result

def initialize_orientation(self):
    _initial_vector = np.array([self.acceleration_x[0],
        self.acceleration_y[0], self.acceleration_z[0]])
    _initial_vector = _initial_vector / np.linalg.norm(_initial_vector)

    self.initial_rotation_x = np.arctan2(_initial_vector[2],
        _initial_vector[1])
    self.initial_rotation_y = 0.0
    self.initial_rotation_z = np.arcsin(_initial_vector[0])

    print(f"Initial Vector: {_initial_vector}")
    print(f"Initial Rotation: {np.degrees(self.initial_rotation_x)},
        {np.degrees(self.initial_rotation_y)},
        {np.degrees(self.initial_rotation_z)}")

    self.rotation_x = [p + self.initial_rotation_x for p in
        self.numerical_integration(self.gyro_x, self.time)]
    self.rotation_y = [r + self.initial_rotation_y for r in
        self.numerical_integration(self.gyro_y, self.time)]
    self.rotation_z = [y + self.initial_rotation_z for y in
        self.numerical_integration(self.gyro_z, self.time)]
```

Listing 1: Orientation estimation using accelerometer data

2.1.3 Valid Range for Accelerometer-Based Orientation Estimation

When the smartphone is tilted around the z-axis, the x-axis acceleration value increases, while the y and z acceleration values approach zero. In this case, the term $z \div y$ in the previous orientation estimation formula diverges, leading to instability in the computation of θ_x.

To address this issue, an alternative derivation using a product of the y and z components is proposed. Starting from the simplified rotation matrix:

$$x = \sin\theta_z$$

$$y \cdot z = \cos\theta_x \cos\theta_z \sin\theta_x \cos\theta_z = \sin\theta_x \cos\theta_x \cos^2\theta_z = \sin\theta_x \cos\theta_x (1 - \sin^2\theta_z) = \frac{1}{2}\sin(2\theta_x)(1 - x^2)$$

From this, we can isolate θ_x as:

$$\theta_x = \frac{1}{2}\arcsin\left(\frac{y \cdot z}{1 - x^2}\right)$$

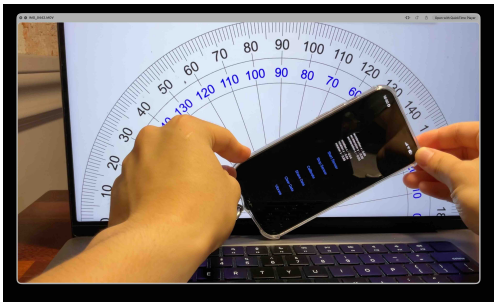
However, this formulation also fails when the x-axis acceleration value approaches 1 or -1, i.e., when $\theta_z \to \pm\frac{\pi}{2}$, which corresponds to tilting the smartphone to 90 degrees around the z-axis. In this situation, the denominator $1 - x^2$ approaches zero, causing the expression to diverge and leading to instability in the calculation of θ_x.

Empirical tests showed that when the tilt angle exceeds approximately 80°, the accelerometer data becomes increasingly noisy and unreliable, as illustrated in Figure 4. In such cases, the estimated θ_x value becomes extremely large and physically implausible.

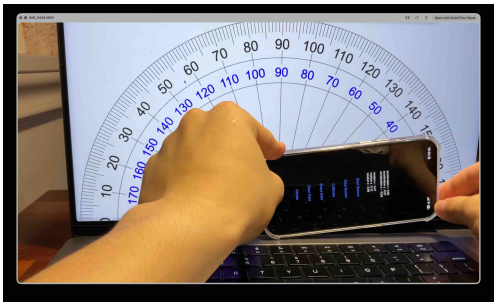
To ensure reliable orientation estimation, we restrict the accelerometer-based calculation of orientation (Section 2.1.2) to cases where the cosine similarity between the reference vector $\vec{g}$ and the normalized acceleration vector $\mathbf{a}$ is greater than 0.5, which corresponds to a maximum tilt angle of approximately 60° from the ideal upright orientation:

$$\cos\theta = \frac{\vec{g} \cdot \vec{a}}{\|\vec{g}\|\|\vec{a}\|} = \frac{-a_y}{\|\vec{a}\|} > 0.5$$

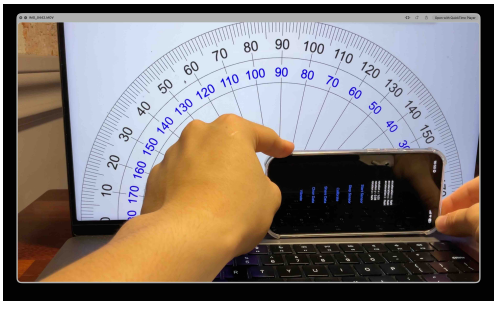
If the initial orientation does not satisfy this condition, the user is prompted to hold the device in an upright position to proceed with calibration.



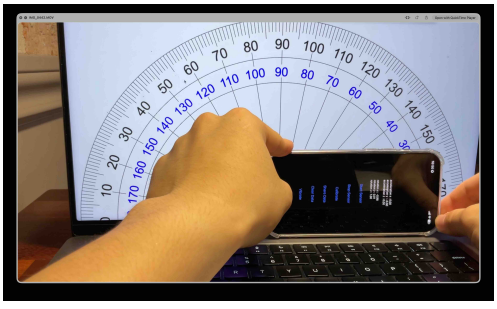
(a) An image of appropriately estimated orientation



(b) Inapposite θ_x caused by y, z getting small



(c) Error caused by zero θ_x



(d) Inapposite θ_x caused by y to zero

Figure 4: A set of images to find the limit of appropriate estimation of orientation

2.1.4 Orientation Tracking Using the Gyroscope

To continuously track changes in the smartphone's orientation during movement, we numerically integrated the gyroscope data over time. The trapezoidal rule was employed for numerical integration to accumulate angular velocity into angular displacement.

Let t denote the current time, and δt the time step between measurements. The gyroscope readings collected up to time t are represented as:

$$\left[\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}, \begin{bmatrix} x_{\delta t} \\ y_{\delta t} \\ z_{\delta t} \end{bmatrix}, \cdots, \begin{bmatrix} x_{t-\delta t} \\ y_{t-\delta t} \\ z_{t-\delta t} \end{bmatrix}, \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \right]$$

The orientation of the smartphone at each corresponding time step is given by:

$$\left[\begin{bmatrix} \psi_0 \\ \theta_0 \\ \phi_0 \end{bmatrix}, \begin{bmatrix} \psi_{\delta t} \\ \theta_{\delta t} \\ \phi_{\delta t} \end{bmatrix}, \cdots, \begin{bmatrix} \psi_{t-\delta t} \\ \theta_{t-\delta t} \\ \phi_{t-\delta t} \end{bmatrix}, \begin{bmatrix} \psi_t \\ \theta_t \\ \phi_t \end{bmatrix} \right]$$

Here, the initial gyroscope reading $\begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^\top$ is assumed to be zero, and the initial orientation $\begin{bmatrix} \psi_0 & \theta_0 & \phi_0 \end{bmatrix}^\top$ is taken from the accelerometer-based estimation described in Section 2.1.2.

To compute the current orientation $\begin{bmatrix} \psi_t & \theta_t & \phi_t \end{bmatrix}^\top$, we use the trapezoidal integration method:

$$\psi_t = \frac{1}{2}(x_t + x_{t-\delta t}) \cdot \delta t + \psi_{t-\delta t}$$

$$\theta_t = \frac{1}{2}(y_t + y_{t-\delta t}) \cdot \delta t + \theta_{t-\delta t}$$

$$\phi_t = \frac{1}{2}(z_t + z_{t-\delta t}) \cdot \delta t + \phi_{t-\delta t}$$

Through this integration, the angular velocity data from the gyroscope can be accumulated to track the smartphone's orientation over time.

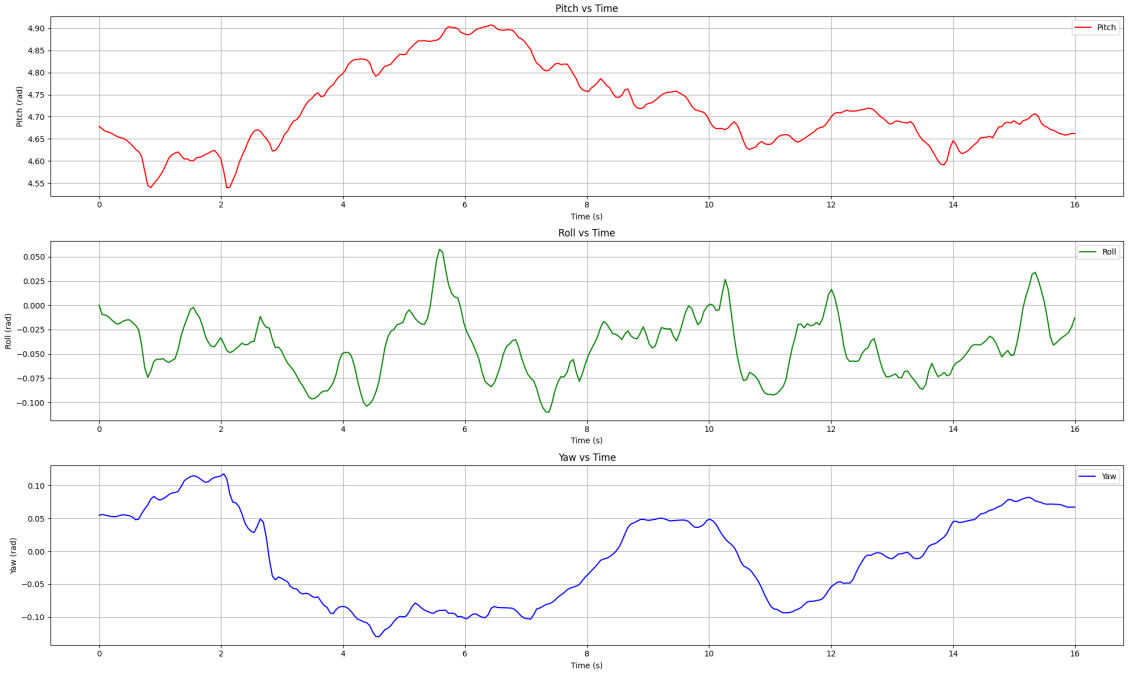


Figure 5: Orientation tracking using trapezoidal integration of gyroscope data

2.1.5 Removal of the Gravity Component

The smartphone's accelerometer inherently includes the effect of gravity. To isolate the linear acceleration resulting purely from the device's movement, the gravitational component must be removed. This is accomplished by reconstructing the gravity vector using the orientation estimated in the previous steps.

This process is essentially the inverse of the approach described in Section 2.1.2. When the device is held vertically with its screen facing the user, the expected acceleration due to gravity is $\vec{g}$. By rotating this reference vector according to the current orientation of the device, we can recover the instantaneous gravity vector in the device's coordinate frame.

Let the current orientation of the device be represented by the Euler angles:

$$\begin{bmatrix} \psi_t \\ \theta_t \\ \phi_t \end{bmatrix}$$

and let the reconstructed gravity vector at time t be denoted as 3-dimensional vector $\vec{g}_t$. Then, the gravity vector is given by:

$$\vec{g}_t = \begin{bmatrix} -\sin\phi_t \\ -\cos\psi_t \cos\phi_t \\ \sin\psi_t \cos\phi_t \end{bmatrix}$$

Finally, the linear acceleration $\vec{A}_t$ caused by the actual movement of the device can be extracted by subtracting the gravity vector from the raw accelerometer reading $\vec{a}_t$:

$$\vec{A}_t = \vec{a}_t - \vec{g}_t$$

This process enables accurate detection of motion-induced acceleration ignoring the gravitational effect in real time.

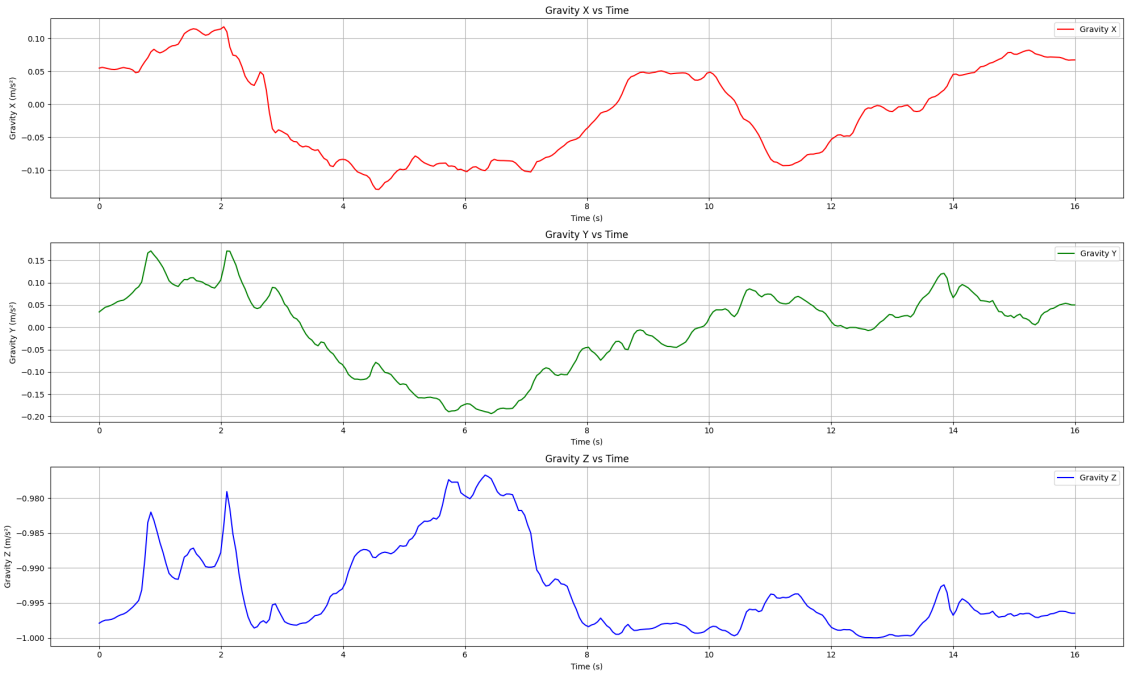


Figure 6: Estimated gravity component arranged by time

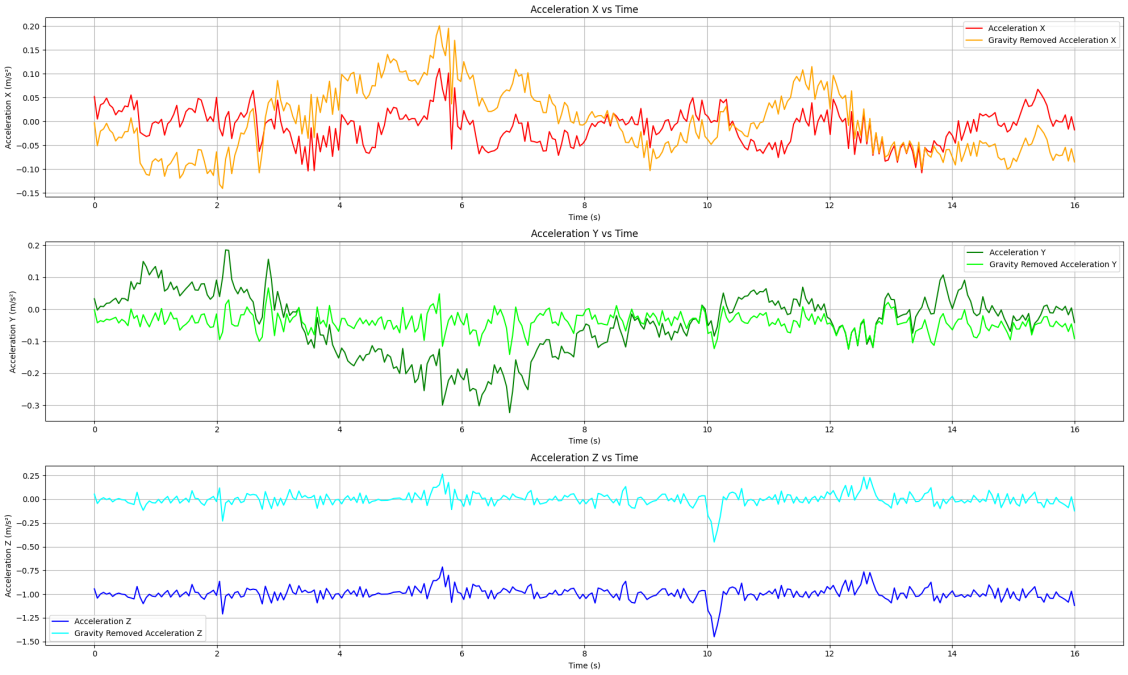


Figure 7: Comparing between raw data and gravity component removed data

2.1.6 Auto Calibration of Orientation

Since gyroscopes accumulate error over time, the orientation should be periodically recalibrated. To achieve this, if the magnitude of the acceleration vector is below a certain threshold(which is set to 1.1) and the device is able to calibrate its orientation(Section 2.1.3), the orientation is recalibrated through the accelerometer data(Section 2.1.2).

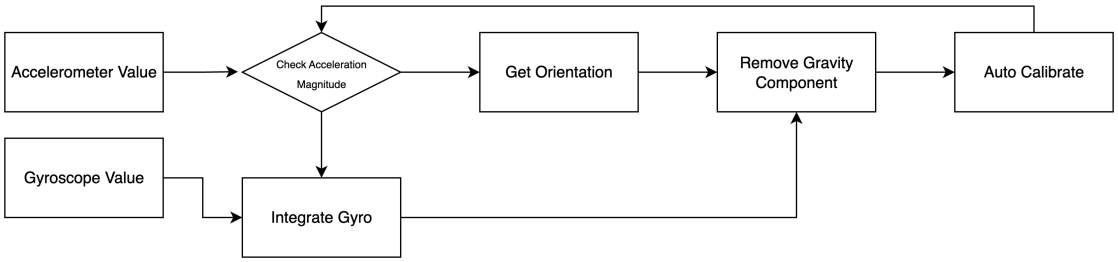


Figure 8: Flow chart for the auto calibration process

2.1.7 Noise Removal

When estimating position from accelerometer data, numerical integration over time often leads to the accumulation of error, resulting in a gradual degradation of accuracy. Figure 9 illustrates this phenomenon.

In this experiment, the smartphone was placed flat with its z-axis perpendicular to the ground. A simple up-and-down motion was performed by slowly lifting and lowering the device. Theoretically, this movement should result in a sinusoidal pattern in the z-axis acceleration, ideally resembling a function of the form $\sin x,\ x \in [0,\ 2\pi]$.

However, the actual measurements deviated significantly from this expected pattern. As shown in the graph, the z-axis acceleration (blue line) did not follow the anticipated sine wave. Surprisingly, the x-axis—unrelated to the vertical motion—displayed unexpected fluctuations, and the y-axis exhibited a divergence trend over time.

These anomalies highlight the impact of sensor noise and the sensitivity of numerical integration to even minor inaccuracies. Small errors in acceleration accumulate rapidly during integration,

leading to significant drift in the estimated position that does not correspond to the actual motion.

Therefore, to achieve reliable position tracking, the incorporation of advanced noise reduction techniques is essential.

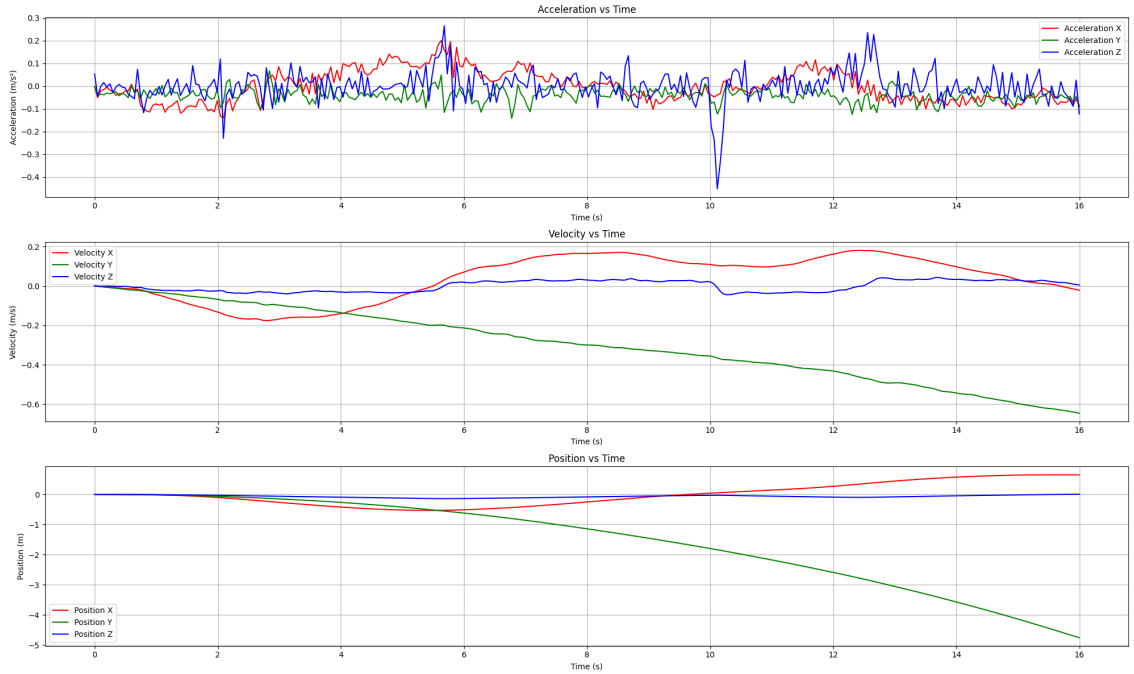


Figure 9: Aceleration, speed, position data before noise removal

2.1.8 Noise Removal - Using Threshold

To reduce noise in the accelerometer data, a simple threshold-based filtering technique was initially employed. This method assumes that if the difference between the current and previous sensor readings is below a predefined threshold, the change is not significant and thus the previous value is retained instead of updating with the new measurement.

To determine an appropriate threshold, we analyzed the distribution of accelerometer values when the smartphone was stationary, as shown in Figure 10. In this state, the x and y axes—which are not affected by gravity—should ideally report values close to zero. However, histogram analysis revealed that values commonly fell within the ranges of 0.14 to 0.15 and -0.16 to -0.15, indicating the presence of noise rather than actual motion. Based on this observation, a threshold of 0.16 was selected. If the change in acceleration between consecutive time steps was less than this threshold, the previous value was maintained.

Despite its simplicity, this method showed limitations in practice. Experimental results indicated that the filter failed to capture small but meaningful movements and led to accumulated error over time. In other words, the threshold-based approach lacked the sensitivity and precision required for accurate motion tracking in this study.

Consequently, we concluded that a more sophisticated filtering technique is necessary. We plan to implement the Kalman Filter, a widely used and robust method for dynamic state estimation, to improve noise reduction while maintaining responsiveness to actual motion.

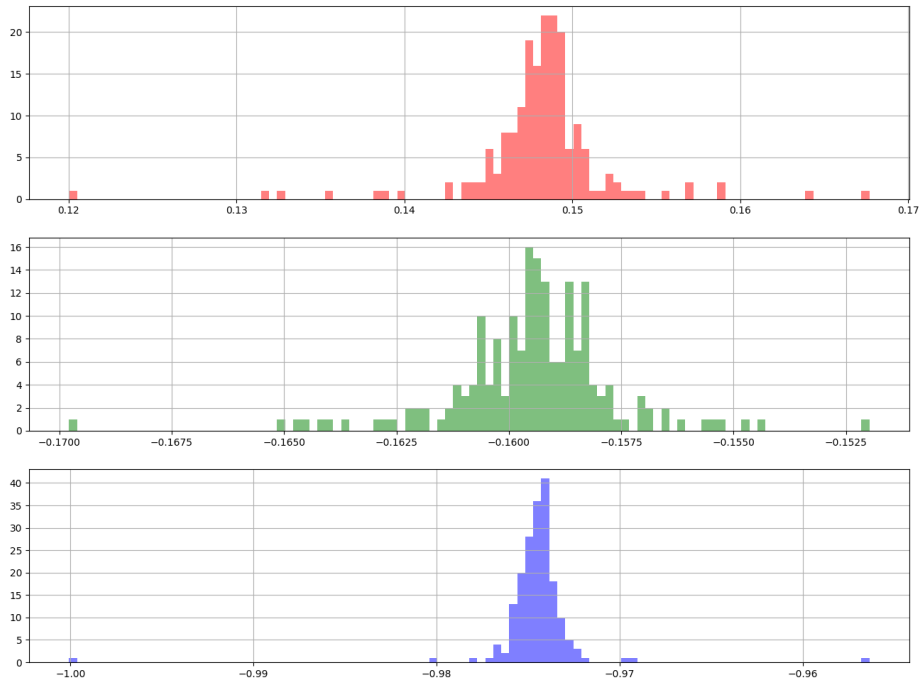


Figure 10: Distribution of accelerometer values when the smartphone is stationary(each graph shows x, y, and z axis in order)

2.1.9 Noise Removal - Using Kalman Filter

To overcome the limitations of the simple threshold-based noise removal method discussed earlier, this study applied a more sophisticated filtering technique—the Kalman Filter. The Kalman Filter is widely recognized for its effectiveness in noise suppression and state estimation in time-series data, and is commonly employed in processing various types of sensor data.

In this experiment, the Kalman Filter was applied to the accelerometer measurements to attenuate sensor noise and achieve more stable position estimation. As expected, the filtered output exhibited smoother and more stable transitions compared to the previous method, indicating improved noise reduction performance.

However, in terms of positional accuracy, the Kalman Filter did not yield a significantly meaningful improvement. While it enhanced the visual smoothness of the signal, it was insufficient to correct the inherent drift and integration errors associated with accelerometer-based position estimation.

Therefore, in the next stage of this study, we plan to apply Fast Fourier Transform (FFT) techniques to analyze and suppress noise in the frequency domain, aiming for further improvement in signal fidelity and positional accuracy.

```
acceleration_x = acceleration_x - gravity_x
acceleration_y = acceleration_y - gravity_y
acceleration_z = acceleration_z - gravity_z

kf = KalmanFilter(
    transition_matrices=[1],
    observation_matrices=[1],
    initial_state_mean=0,
    initial_state_covariance=1,
    observation_covariance=1,
    transition_covariance=0.01,
)
```

```
13
14 state_means_x, state_covariances_x = kf.filter(acceleration_x)
15 state_means_y, state_covariances_y = kf.filter(acceleration_y)
16 state_means_z, state_covariances_z = kf.filter(acceleration_z)
```

Listing 2: Kalman Filter for noise reduction

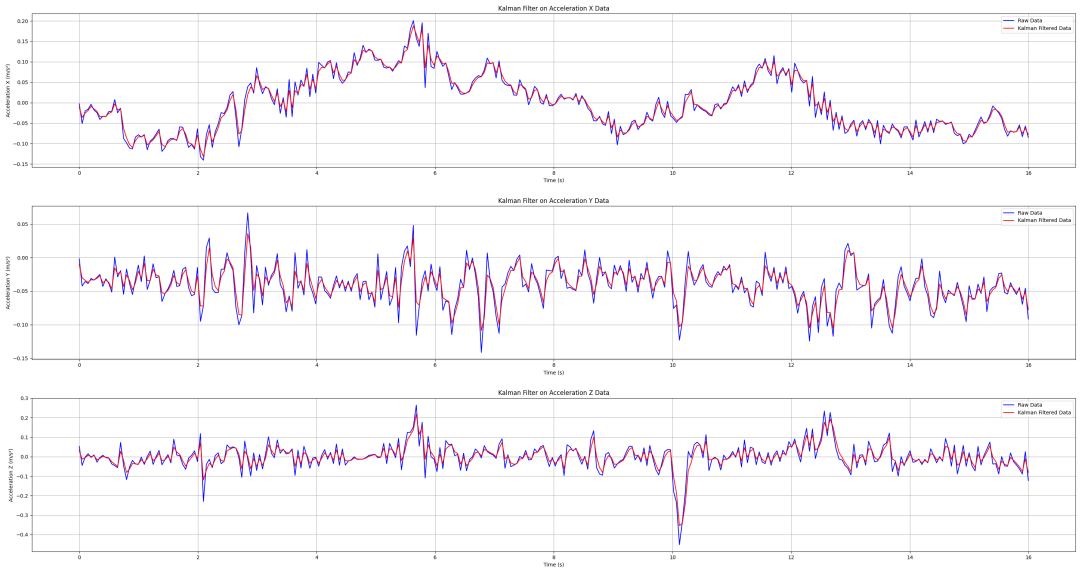


Figure 11: Comparison between raw accelerometer data and noise reduced accelerometer data using Kalman Filter

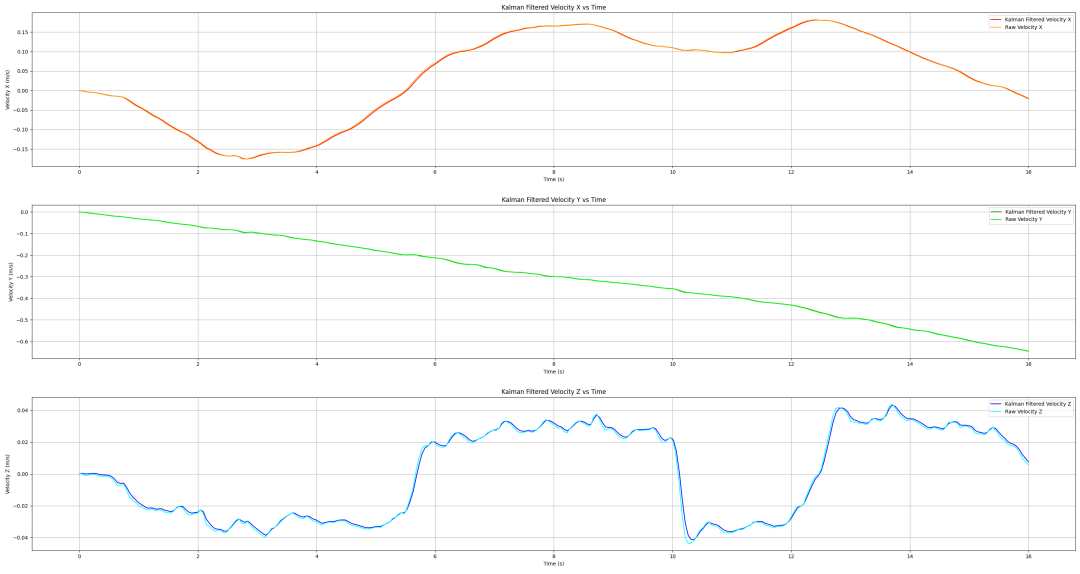


Figure 12: Comparison between raw speed data and noise reduced speed data using Kalman Filter

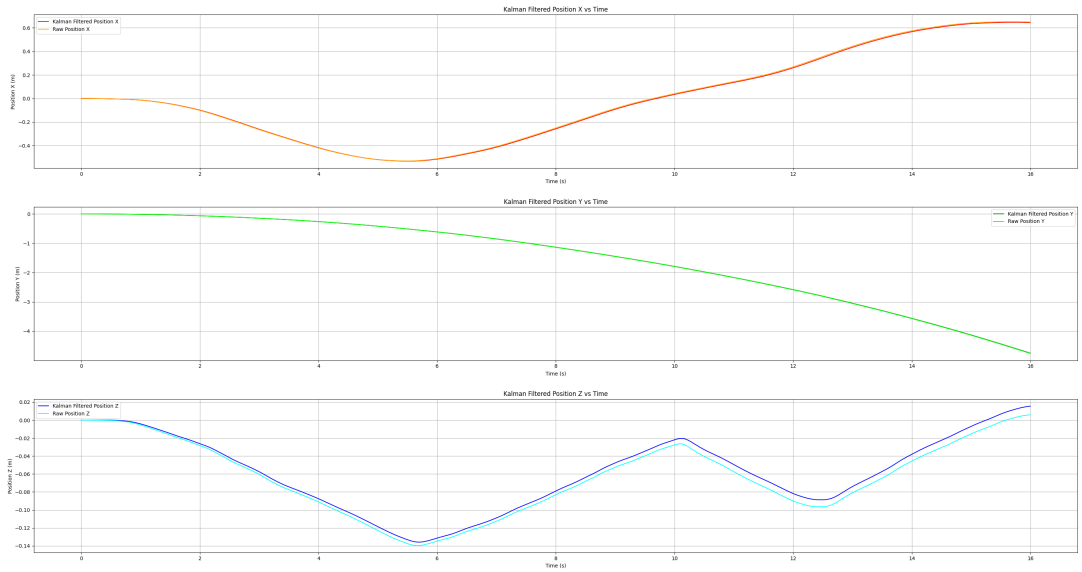


Figure 13: Comparison between raw position data and noise reduced position data using Kalman Filter

2.1.10 Noise Removal - Using FFT(Fast Fourier Transform)

Given the lack of significant improvement in position estimation accuracy after applying the Kalman Filter, this study further explored frequency-based noise reduction techniques using the Fast Fourier Transform (FFT). FFT is a powerful method that converts discrete acceleration signals from the time domain into the frequency domain, allowing for the identification and removal of high-frequency components or unwanted noise.

In this experiment, FFT was applied to the time-series accelerometer data to obtain its frequency spectrum. Frequency components with amplitudes below a threshold of 0.1 were considered insignificant and were eliminated. Subsequently, the Inverse FFT (IFFT) was used to reconstruct the denoised signal in the time domain.

Despite the theoretical effectiveness of this approach, the experimental results showed no substantial improvement in position estimation accuracy. The reconstructed signals appeared cleaner, but the overall trajectory estimation remained inaccurate. Thus, the FFT-based filtering approach, like previous methods, did not yield a practically meaningful enhancement in performance.

```
n = len(fft_time)

fft_x = np.fft.fft(fft_acceleration_x)
fft_x_single = np.abs(fft_x[:n//2])
fft_x_single[1:] = 2 * fft_x_single[1:]

fft_y = np.fft.fft(fft_acceleration_y)
fft_y_single = np.abs(fft_y[:n//2])
fft_y_single[1:] = 2 * fft_y_single[1:]

fft_z = np.fft.fft(fft_acceleration_z)
fft_z_single = np.abs(fft_z[:n//2])
fft_z_single[1:] = 2 * fft_z_single[1:]

frequencies = np.fft.fftfreq(len(fft_time), d=(fft_time[1] - fft_time[0]))
```

Listing 3: Fast Fourier Transform for noise reduction

Figure 14: Comparision between raw accelerometer data and noise reduced accelerometer data using FFT

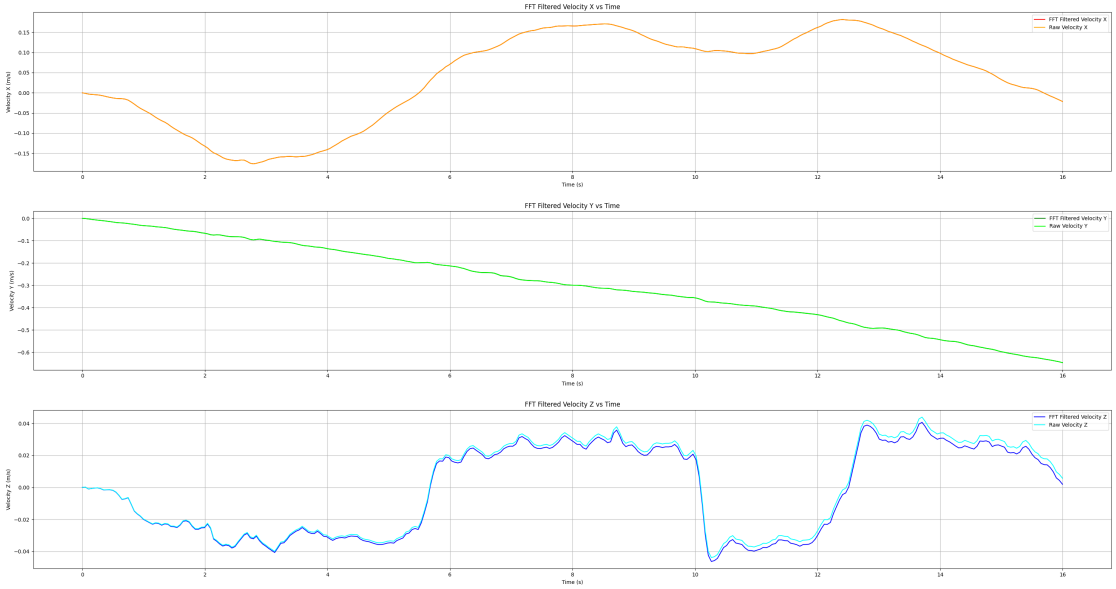


Figure 15: Comparision between raw speed data and noise reduced speed data using FFT

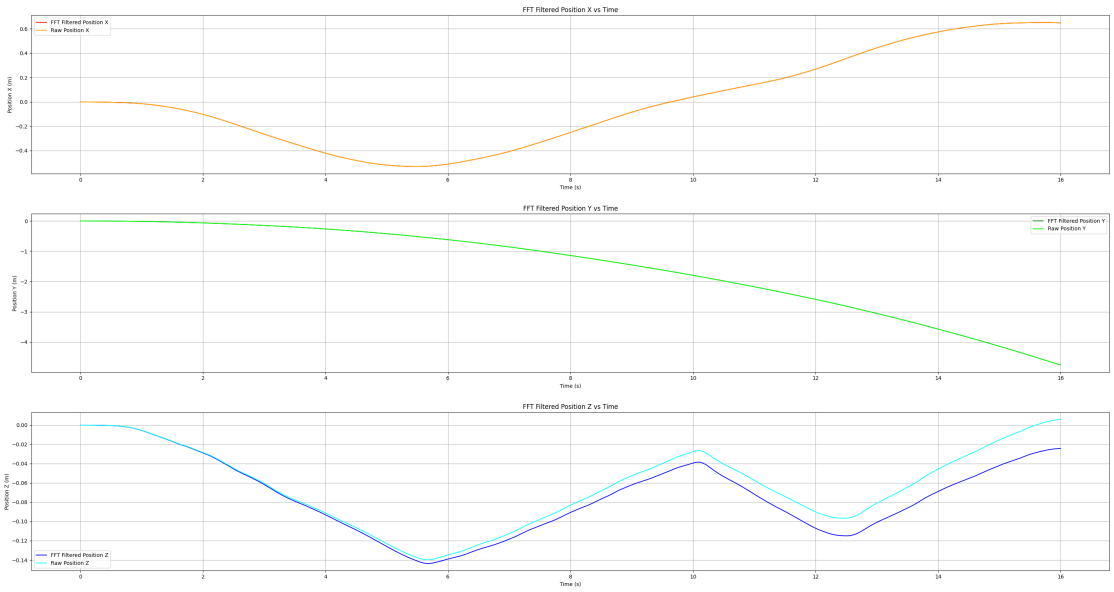


Figure 16: Comparision between raw position data and noise reduced position data using FFT

2.1.11 Limit of Accelerometer

Theoretically, integrating acceleration yields velocity, and a second integration provides position. However, in real-world environments, this process suffers from significant cumulative errors due to the inherent noise present in accelerometer measurements. As a result, accurate position estimation using raw sensor data becomes extremely challenging.

To address this issue, several preprocessing techniques were applied, including Kalman filtering, Fast Fourier Transformation (FFT), and its inverse. While these methods aimed to reduce noise and improve signal quality, they provided only marginal improvements in accuracy when considering the limited computational resources available on mobile devices. In practice, the increased algorithmic complexity outweighed the performance gains, making these approaches less suitable for real-time mobile applications.

In response, this study proposes a novel approach: instead of relying on traditional physics-based integration methods, we remove only the gravitational component from the acceleration data and employ a deep learning model to learn and predict the trajectory of the smartphone. By leveraging data-driven learning rather than error-prone numerical integration, this method aims to alleviate the issue of accumulated drift and achieve more practical and reliable position estimation in mobile environments.

2.2 Planning Punch Prediction Model

While multiple model architectures were explored—including pure Convolutional Neural Networks (CNN) and hybrid CNN-RNN structures—the final implementation adopts a lightweight Gated Recurrent Unit (GRU) model. This decision is primarily motivated by two factors.

First, the punch motion exhibits relatively simple and distinct temporal patterns that do not require deep spatial feature extraction. Second, considering the operational constraints of mobile environments, the model must perform inference up to 20 times per second. In this context, the computational efficiency of GRU makes it better suited for real-time execution than heavier CNN or LSTM-based models.

Although CNNs and CNN-RNN hybrids (e.g., Conv2D+LSTM or Conv2D+GRU) were evaluated, their larger parameter sizes and higher computational cost presented challenges for on-device deployment. The GRU architecture, by contrast, achieves a good balance between accuracy and speed, especially for time-sensitive applications such as punch classification.

To facilitate faster convergence and enhance model robustness, a `tanh` activation function is applied to the input layer. This serves as a normalization step, scaling the accelerometer data to fall within a bounded range, which is particularly beneficial when the input magnitude varies

significantly across different users or punching styles.

Final GRU-Based Architecture

- **Input:** 10×3 sequence of accelerometer vectors
- **Preprocessing:** `tanh` activation applied to raw input for normalization
- **Recurrent Layer:** Single GRU layer with 8–16 hidden units
- **Fully Connected Layer:** 4 output neurons (for punch classes: straight, hook, upper cut, body)
- **SoftMax:** Converts outputs to class probabilities
- **Threshold-Based Trigger:** Inference is executed only when the magnitude of acceleration exceeds a predefined threshold

This design ensures that the model maintains real-time performance with minimal energy usage, making it suitable for continuous punch classification on mobile devices.

2.3 Preparing Punch Dataset

2.3.1 Data Construction

To develop an effective deep learning-based punch recognition model, we constructed time-series input data derived from smartphone sensor readings and created corresponding labeled datasets.

The raw sensor data were collected using the built-in Inertial Measurement Unit (IMU), which includes a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. However, the magnetometer data were excluded from the model due to high noise levels and their limited contribution to motion recognition. Similarly, gyroscope data—although useful for orientation estimation—were also excluded from the input features to reduce model complexity. In contrast, since the x and z axis orientation datas are stable and correct in comparison to the y axis, these datas can provide a certain pattern for training.

Based on this setup, the input features $\mathbf{X}$ can be whether processed 3-dimension acceleration data or include additional 2 dimensions for x, z, axis orientation data over a fixed time window T.

The target label $\mathbf{y}$ corresponds to the punch class annotated at each timestamp. The class labels are defined as follows: `0`: No punch detected, `1`: Straight, `2`: Hook, `3`: Uppercut, `4`: Body punch.

For training, class labels are initially provided as strings (e.g., ‘"Straight"‘, ‘"None"‘, etc.), and then encoded either as one-hot vectors for use with the `categorical_crossentropy` loss function, or as integer indices for `sparse_categorical_crossentropy`, depending on the model's configuration.

```
# Example shape: (T, 1, 3)
X = [
    [ acc_x_1, acc_y_1, acc_z_1, (orient_x_1, orient_z_1) ],
    [ acc_x_2, acc_y_2, acc_z_2, (orient_x_2, orient_z_2) ],
    ...
    [ acc_x_T, acc_y_T, acc_z_T, (orient_x_T, orient_z_T) ]
] # orientation is optional

y = [
  "None",
  "Straight",
  "None",
  ...
  "None",
]
```

Listing 4: Example input-output data format

This labeling scheme ensures that the model learns not only to recognize punch gestures, but also to distinguish them from idle or transitional motion patterns. The model is trained to output a class prediction at each time step, enabling real-time recognition of punch events as they occur.

2.3.2 Data Augmentation Using Sliding Window

To increase the size of the training dataset and improve model generalizability, we adopted a sliding window algorithm to generate overlapping time-series samples. This method is particularly effective for time-dependent sensor data, as it allows the model to learn from slightly shifted but semantically similar motion sequences.

Given a raw data stream $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_n]$, where each $\mathbf{d}_i$ is a 3×3 matrix representing three consecutive accelerometer readings across the x, y, and z axes, we construct input sequences $\mathbf{X}$ of fixed length T using a sliding window of size T and stride 1. The label $\mathbf{y}$ for each sequence is assigned based on the punch class at the last timestep of the window.

The following Python snippet illustrates the implementation of this process:

```
WINDOW = 20  # sequence length T

X_list = []
y_list = []

for i in range(len(data) - WINDOW):
    window = data[i:i+WINDOW]
    label = label_list[i + WINDOW - 1]  # label at the last timestep
    X_list.append(window)
    y_list.append(label)
```

Listing 5: Sliding window algorithm for data augmentation

This approach enables a significant increase in the number of training samples, while preserving the temporal structure of the original data. Additionally, by assigning the label based on the final frame in each window, the model is encouraged to predict punch classes based on their full temporal context, rather than instantaneous changes.

2.3.3 Data Accumulation

To accumulate the punch dataset, a website was developed to allow users to record their punch data using a smartphone. The datas are stored into CloudFlare Storage via CloudFlare Workers.

The columns of the accumulated datasets are as follows:

- **Index**: The index of the data entry.
- **Time**: The timestamp of the data collection.
- **Raw Acceleration X, Y, Z**: The raw acceleration values recorded by the smartphone's accelerometer.
- **Acceleration X, Y, Z**: Gravity-removed acceleration values calculated from the raw data.
- **Gyro X, Y, Z**: The gyroscope readings for the x, y, and z axes.
- **Orientation X, Y, Z**: The estimated orientation angles of the smartphone in radians.
- **Punch Type**: The type of punch performed, such as "None", "Straight", "Hook", "Uppercut", or "Body".

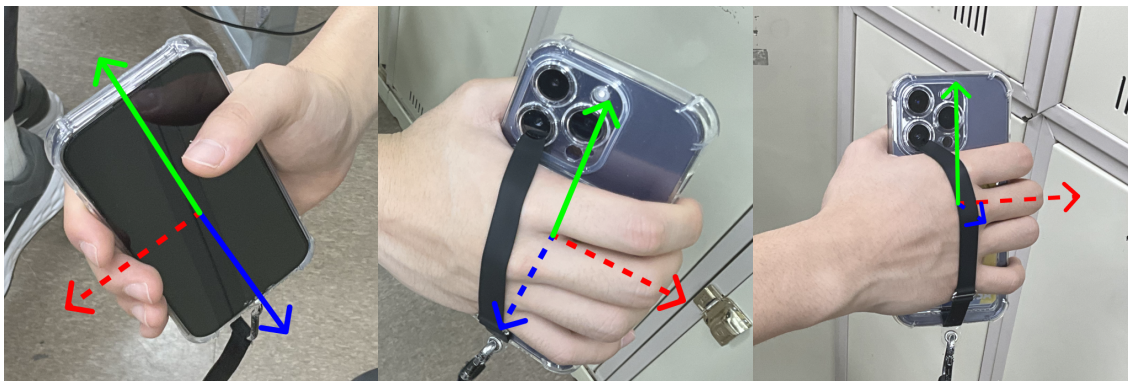


Figure 17: A set of images showing the smartphone being held (refer to Figure 3 for orientation, the dotted lines are negative axes)

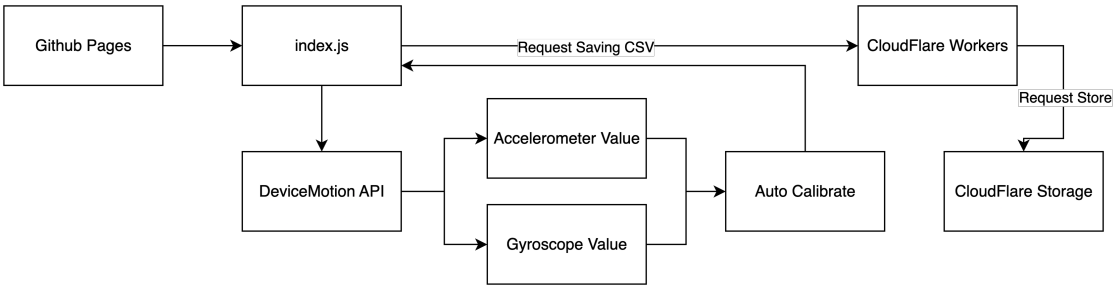
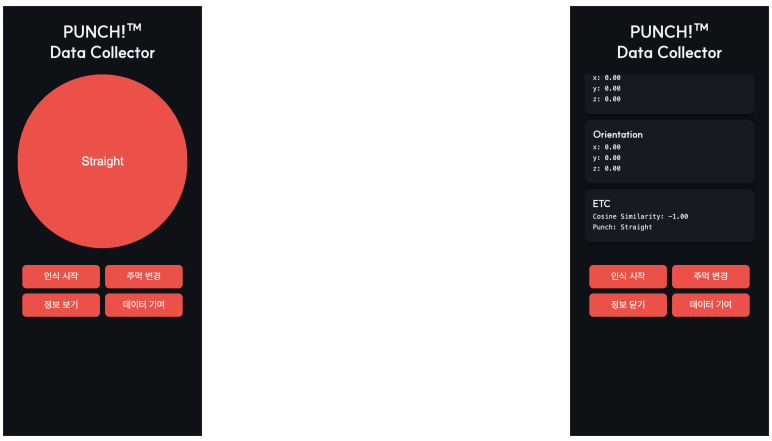


Figure 18: Punch data collection website flow chart



(a) A screen for recording punch data

(b) A screen for viewing recorded punch data

Figure 19: Screenshots of the data collection website

2.4 Analyzing Punch Data

As the graph is shown in Figure 20 and Figure 21, a Straight punch and a Hook punch seems to have a relation between acceleration. Especially the local minima of the X-axis acceleration usually matches the peak of the punch. Besides, the value of Y-axis orientation had difference with the real situation. This seems to be occured by accumulated error of the gyroscope.

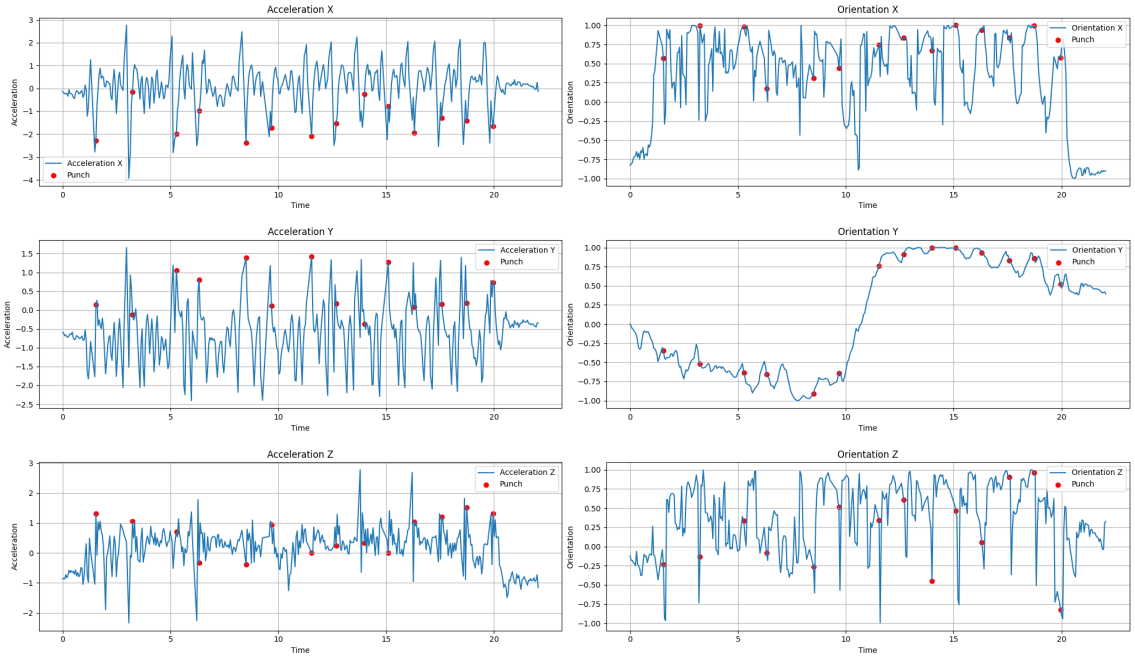


Figure 20: Punch data collected from a user performing a straight punch, red dots indicate the peak of the punch

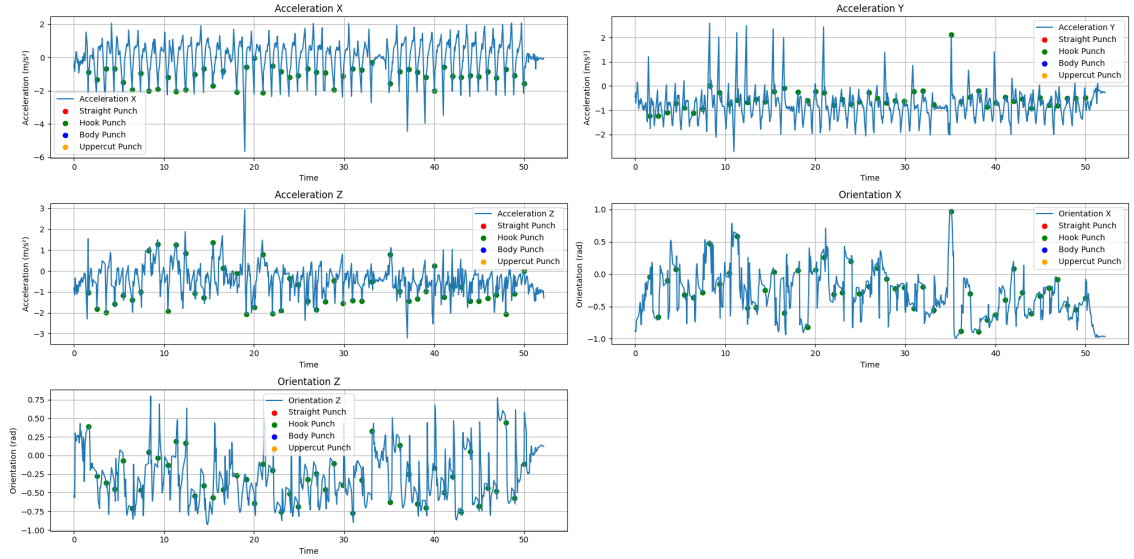


Figure 21: Punch data collected from a user performing a hook punch, green dots indicate the peak of the punch (the y axis orientation is removed due to accumulated error)

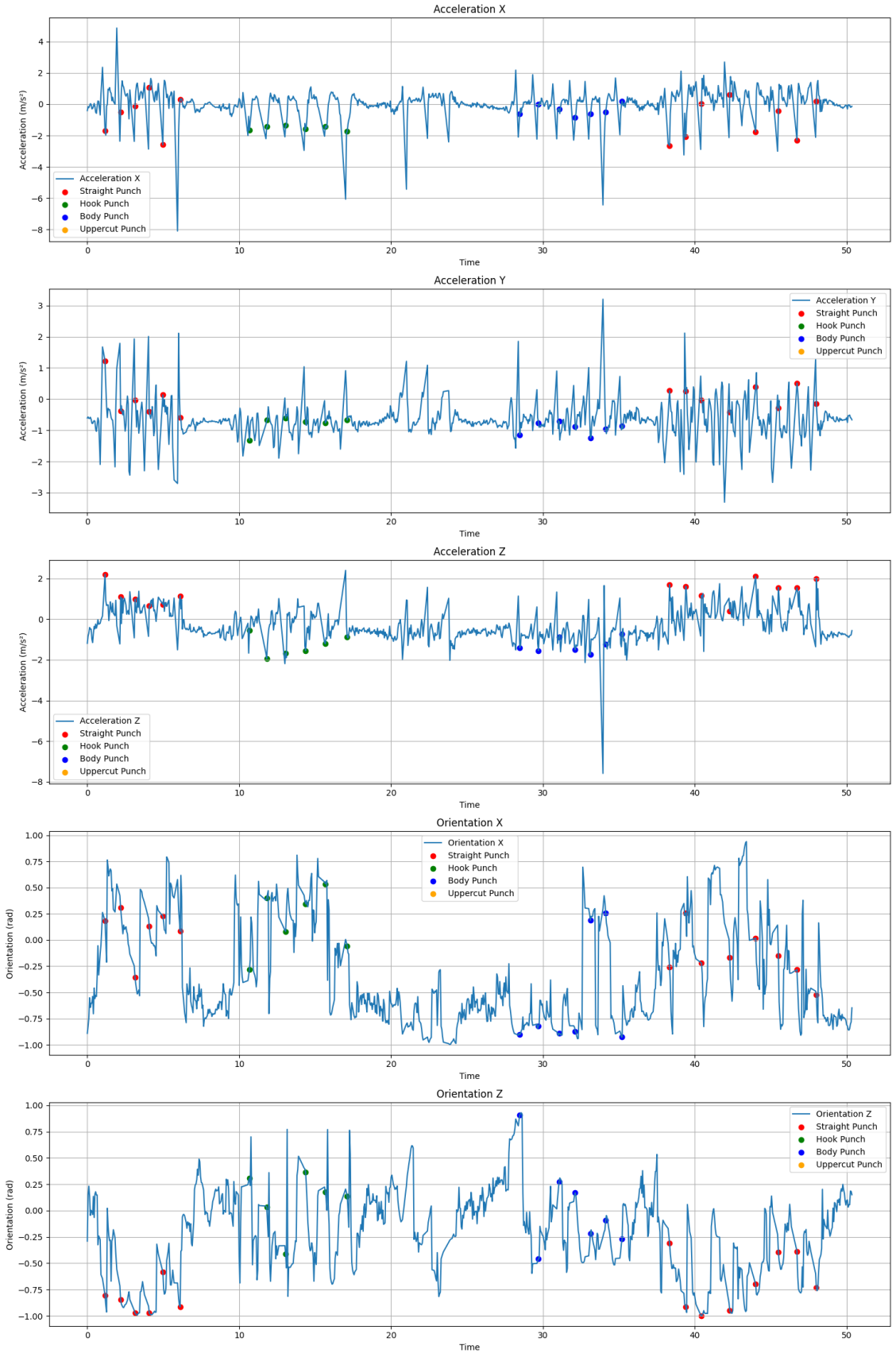


Figure 22: Punch data collected from a user performing various punches(straight, body, hook, uppercut), each colored dot indicates the punch

2.5 Building Punch Predict Model

Various GRU-based models were built to predict the punch type based on the input data. The model architecture is designed to handle time-series data, with the input layer accepting sequences of accelerometer readings and the output layer predicting the punch class.

Table 1 summarizes the different GRU model configurations used in this study. Each model is indexed by its input layer size, hidden layer size, number of GRU layers, and window size. The input layer size corresponds to the number of features in the input data (e.g., 3 for acceleration data, 5 for acceleration data with orientation). The hidden layer size determines the number of units in the GRU layer. The window size indicates the length of the input sequence used for training. Since a single punch can be thrown in a short time, the window size is set to 20, which is equivalent to $1 (= 0.05 \times 20)$ second. Data used for training is either preprocessed acceleration (shown as Acceleration in rest of the paper) data or raw acceleration data with orientation.

Input Layer Size	Hidden Layer Size	GRU Layer Count	Window Size	Data
3	5	1	20	Acceleration
5	10	2		Raw Acceleration
		4		
		8		

Table 1: GRU model indexing

`GRU_3I_5H_1L_5W_A`: GRU model with 3 input features, 5 hidden units, and 1 layer, trained with acceleration data accumulated over a 5 window size

0.2% of the total data was used for validation, and early stopping was applied with the factor related to the validation loss. Since train data and test(validation) data are seperated, the model was trained with the training data and evaluated with the validation data.

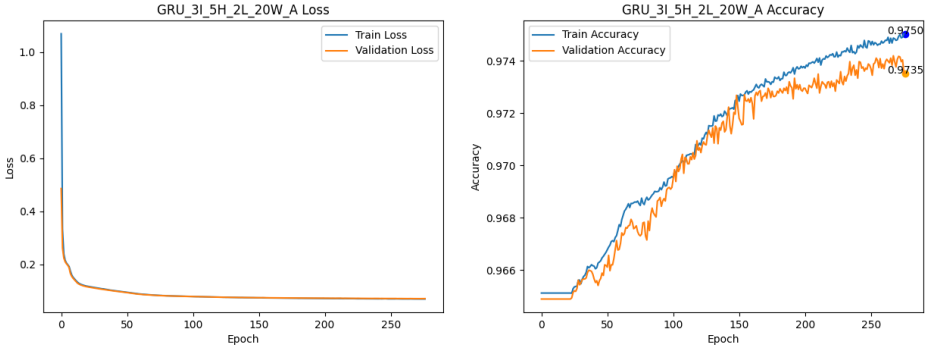
In the first try, since it can build a model with fewer code, the model was built by TensorFlow and imported to various platforms by TensorFlow.js. However, in order to specify options(e.g. adding a hyperbolic tangent layer right behind the input layer for data normalization), the model was built again by PyTorch. Both codes are available in GitHub.

2.6 Evaluating Various Punch Predict Model

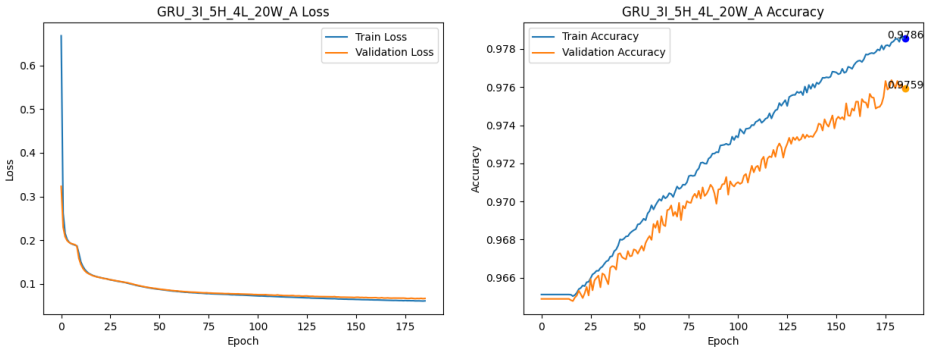
Model Name	Validation Accuracy
`GRU_3I_5H_1L_20W_A`	96.67%
`GRU_3I_5H_2L_20W_A`	97.35%
`GRU_3I_5H_4L_20W_A`	97.59%
`GRU_3I_5H_8L_20W_A`	96.84%
`GRU_3I_10H_1L_20W_A`	97.86%
`GRU_3I_10H_2L_20W_A`	98.80%
`GRU_3I_10H_4L_20W_A`	98.88%
`GRU_3I_10H_8L_20W_A`	98.12%
`GRU_3I_10H_1L_20W_R`	97.59%
`GRU_3I_10H_2L_20W_R`	98.78%
`GRU_3I_10H_4L_20W_R`	98.78%
`GRU_3I_10H_8L_20W_R`	98.56%
`GRU_5I_10H_1L_20W_A`	97.88%
`GRU_5I_10H_2L_20W_A`	98.82%
`GRU_5I_10H_4L_20W_A`	99.15%
`GRU_5I_10H_8L_20W_A`	98.04%

Table 2: Validation accuracy of various GRU models trained with acceleration data

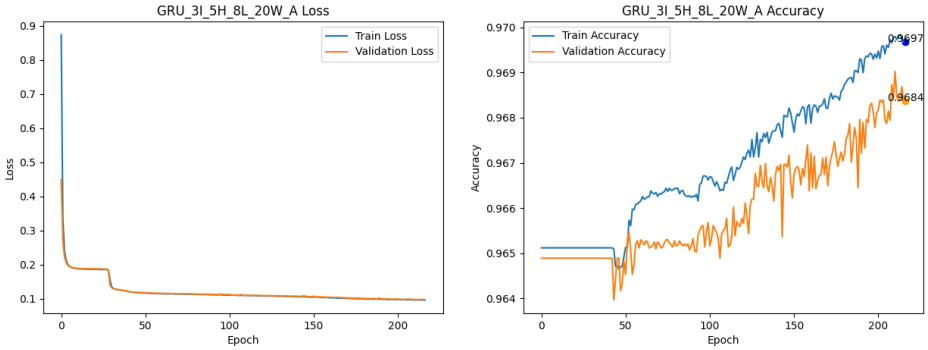
(a) GRU model with 3 input features, 5 hidden units, and 1 layer, trained with 20 window size



(b) GRU model with 3 input features, 5 hidden units, and 2 layers, trained with 20 window size

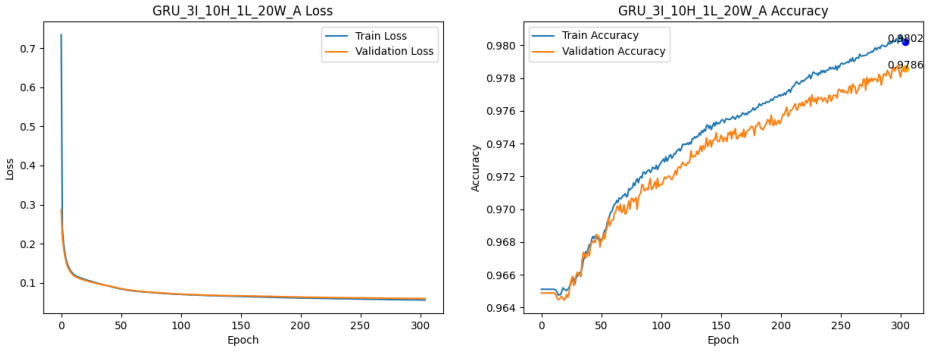


(c) GRU model with 3 input features, 5 hidden units, and 4 layers, trained with 20 window size

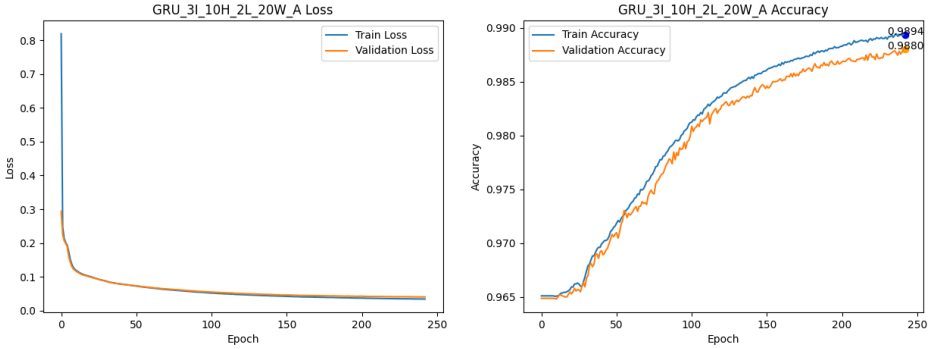


(d) GRU model with 3 input features, 5 hidden units, and 8 layers, trained with 20 window size

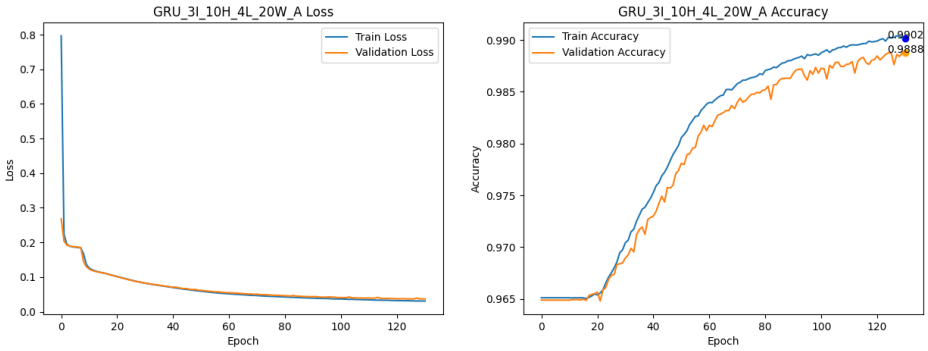
Figure 23: Various GRU models with different input features, hidden units, and layers, trained with acceleration data



(a) GRU model with 3 input features, 10 hidden units, and 1 layer, trained with 20 window size



(b) GRU model with 3 input features, 10 hidden units, and 2 layers, trained with 20 window size

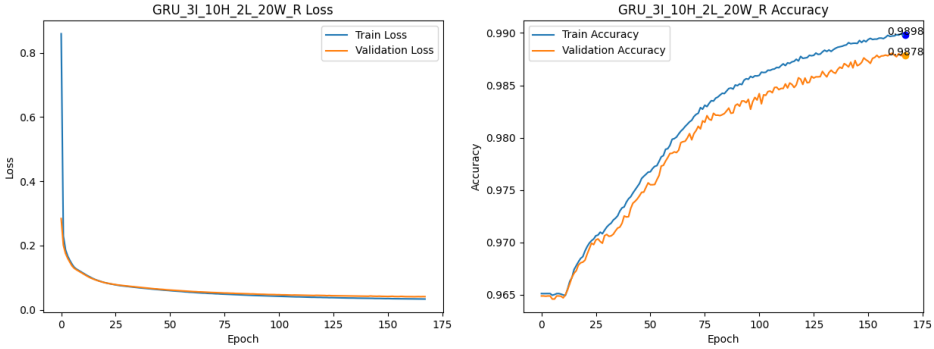


(c) GRU model with 3 input features, 10 hidden units, and 4 layers, trained with 20 window size

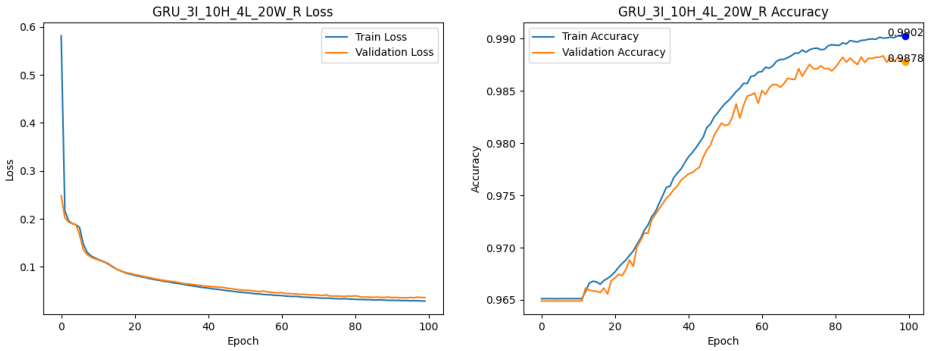
(d) GRU model with 3 input features, 10 hidden units, and 8 layers, trained with 20 window size

Figure 24: Various GRU models with different input features, hidden units, and layers, trained with acceleration data

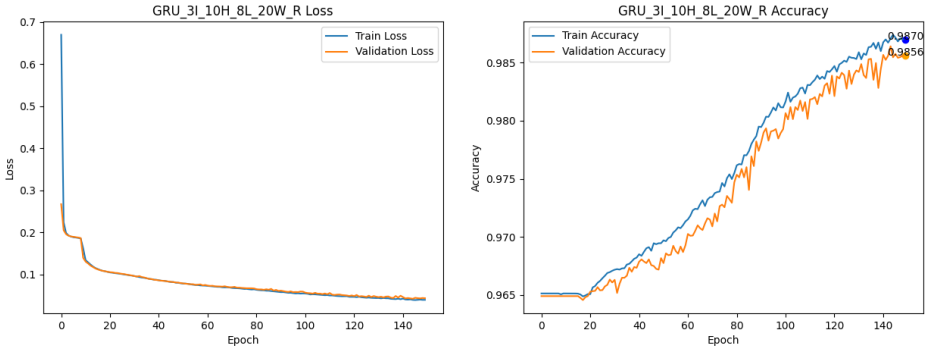
(a) GRU model with 3 input features, 10 hidden units, and 1 layer, trained with 20 window size



(b) GRU model with 3 input features, 10 hidden units, and 2 layers, trained with 20 window size



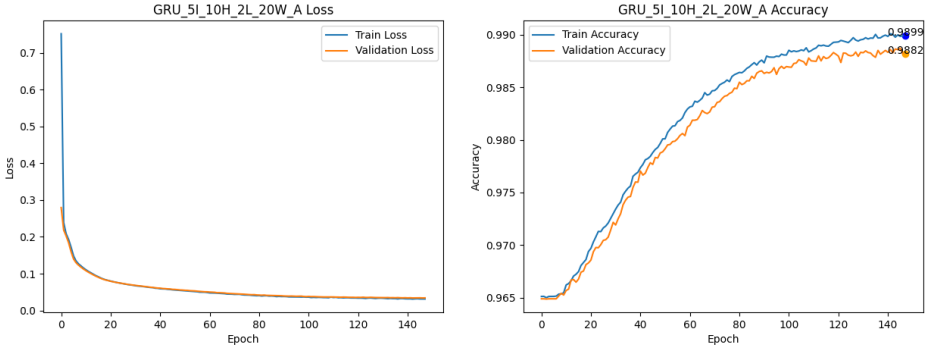
(c) GRU model with 3 input features, 10 hidden units, and 4 layers, trained with 20 window size



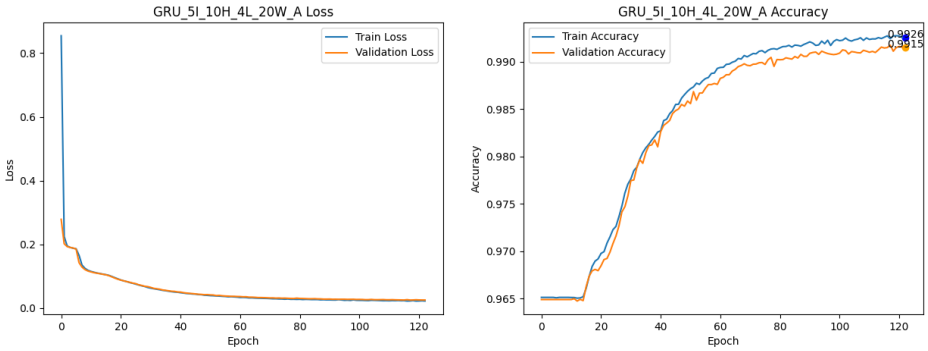
(d) GRU model with 3 input features, 10 hidden units, and 8 layers, trained with 20 window size

Figure 25: Various GRU models with different input features, hidden units, and layers, trained with raw acceleration data

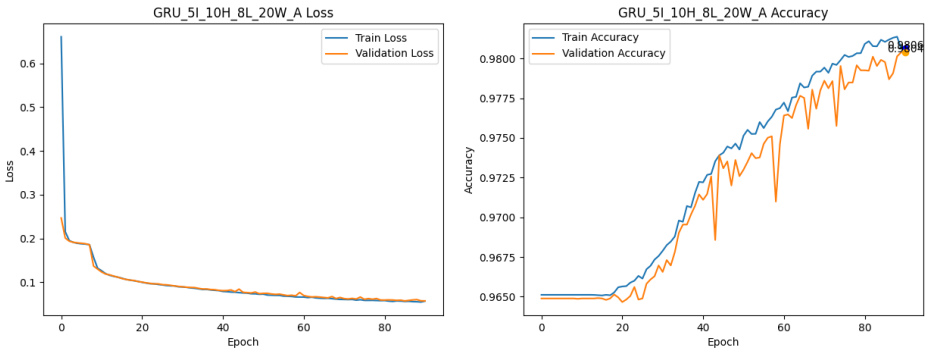
(a) GRU model with 5 input features, 10 hidden units, and 1 layer, trained with 20 window size



(b) GRU model with 5 input features, 10 hidden units, and 2 layers, trained with 20 window size



(c) GRU model with 5 input features, 10 hidden units, and 4 layers, trained with 20 window size



(d) GRU model with 5 input features, 10 hidden units, and 8 layers, trained with 20 window size

Figure 26: Various GRU models with different input features, hidden units, and layers, trained with acceleration(x, y, z) and orientation(only x, z) data

- **Impact of Orientation Data**: Regarding situations represented in Figure 24, and Figure 26, the model with 3 input features achieved an average validation accuracy of 98.415%, while the model with 5 input features achieved an average validation accuracy of 98.4725%. This suggests that the additional orientation data (x, z axes) does not have a significant impact on providing useful information for punch classification.
- **Impact of Hidden Layer Size**: Regarding situations represented in Figure 23, Figure 24, models with 10 hidden units showed an average validation accuracy of 98.415%, which is slightly higher than models with 5 hidden units resulting in an average validation accuracy of 97.1125%. This indicates that increasing the number of hidden units can lead to better validation accuracy than models with 5 hidden units.
- **Impact of Number of Layers**: Regarding situations represented in Figure 23, Figure 24, Figure 25 and Figure 26, models with 4 layers achieved the highest average validation accuracy of 98.6% compared to models with 1 layer (97.5%), 2 layers (98.4375%) and 8 layers (97.89%). This indicates that increasing the number of layers can lead to better validation accuracy, but the performance gain is marginal beyond 4 layers.
- **Impact of Preprocessed Acceleration Data**: Regarding situations represented in Figure 24, Figure 25, models using preprocessed acceleration data showed an average validation accuracy of 98.415%, while models using raw acceleration data achieved an average validation accuracy of 98.4275%. This suggests that removing the gravity component from the accelerometer data does not have a significant impact on improving the model's performance.

2.7 Applying Model

The trained model was deployed on a website and implemented in a mobile application. The website allows users to test the model with validation data, while the application provides real-time punch classification with the smartphone's accelerometer and gyroscope data.

3 Results

While removing the gravity component from the accelerometer data did not lead to a significant improvement in model performance, the application of a deep learning model for punch classification yielded promising outcomes. The best-performing model achieved a validation accuracy of 99.15% on the punch dataset, demonstrating its ability to effectively distinguish between different types of punches based solely on smartphone accelerometer data.

These results highlight the potential of leveraging smartphone sensors for real-time motion recognition. In particular, this approach offers a practical solution for punch classification in mobile applications, contributing to more engaging and active user experiences—potentially addressing issues associated with sedentary lifestyles.

To further enhance model performance and generalizability, future work may explore the following directions:

- **Data Augmentation**: Expanding the dataset by collecting punch data from a more diverse range of users and environments, thereby improving the model's robustness across different conditions.
- **Time Interval Adjustment**: Modifying the sensor sampling interval to better capture the dynamics of phone movement, which may enable classification based on the phone's positional changes during a punch.

All the sources of this project are available in GitHub.