RepMate – Your ML Powered Lifting Coach

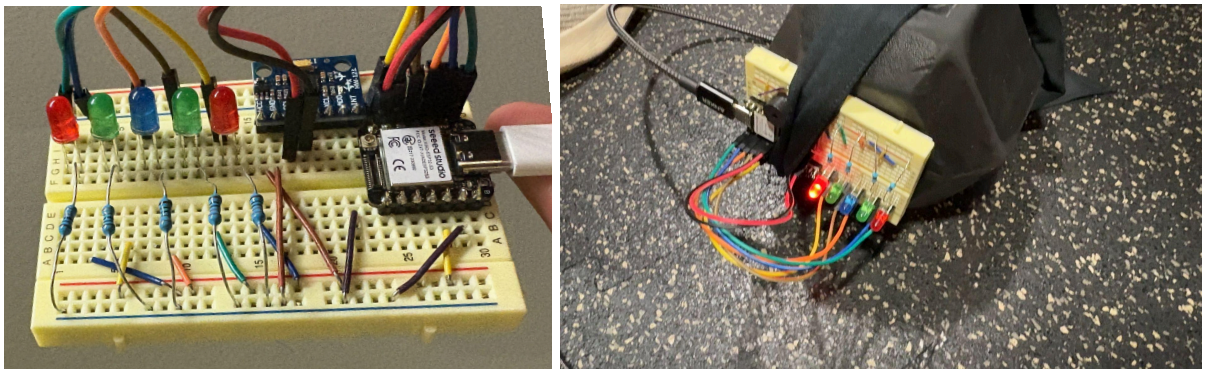
GitHub/RepMate | Demo Video

Rohan Panday - EE ‘26
rpanday@seas.upenn.edu

Vedansh Goenka - CIS ‘26
vedanshg@seas.upenn.edu

Introduction:

This project aims to develop an attachable smart device that connects to a smartphone via Bluetooth to monitor and evaluate lifts at the gym. The device will recognize proper and improper lifting techniques (such as shaking, incomplete motion, or tilting) and provide instant audio feedback to guide the user so they know how to correct their lifts mid-rep, exactly how a real coach would. The device will also log detailed information for the user to review post-workout to see if they consistently use a suboptimal form (and how to improve it).



Our final device

Motivation:

Lifting weights is essential to many people’s fitness journey, but ensuring proper form is crucial to avoid injury and achieve results, especially for novice lifters who may not know what “good form” looks like. Improper lifting techniques can lead to injuries or reduced gains. This project aims to provide real-time feedback to users on their lifts using an attachable IMU device that recognizes different types of mistakes. This device lets lifters receive instant feedback during a workout and track their form over time, creating a safer and more productive workout experience.

Dataset Details:

Our dataset is split into six classes: Proper Form, Lift Instability, Partial Motion, Off-Axis, Swinging Weight, and No Lift. We defined the classes as follows:

- Proper Form: perfect lifting form
- Lift Instability: the user is shaking the weight as it is being lifted – indicating over-lifting
- Partial Motion: the weight is not being lifted to full extension – indicating fatigue
- Off-Axis: the user is lifting the weight off the proper axis of the lift (for example, in a curl, the user is crossing their body with the weight) – this targets a different array of muscles
- Swinging Weight: the user is hunching or swinging their body to lift the weight – too heavy
- No Lift: the weight is not being actively lifted; the user may be readjusting the weight, walking around, or just holding the weight

The dataset was collected from 4 users on both of their left and right arms, and we had a total of 556 unique data points (pre-augmentation) split into the lift types as follows:

Dumbell Curls | 298	Bench Press | 127	Dumbell Flys | 131

We augmented our data by randomly applying one or two augmentations for a given sample. We scaled the magnitude uniformly (90% to 110%), added Gaussian noise (mean = 0, std dev = 0.03), randomly selected 5% of our data points (6000 possible per sample) to any value between the acceleration and gyroscope class max/min, and finally, rotated the data by 90 deg while preserving the direction of gravity – simulating the IMU attached to the other side of the dumbbell.

We multiplied the max lift type class value by the augmentation factor to determine the number of augmentations to apply. We removed the original data points for that specific class: (max class * augmentation factor)—original data points per class.

Our final dataset split was train (72.8% | aug factor = 2), validation (14% | aug factor = 1.25), test (13% | aug factor = 2)

Here's an example of the final dataset for one of our classes (Dumbell Curls)

Dumbell Curls						
# of Samples	Train		Validation		Test	
	Pre aug	Post aug	Pre aug	Post aug	Pre aug	Post aug
No Lift	7	14	12	15	39	78
Off Axis	6	14	9	15	32	78
Partial Motion	3	14	9	15	36	78
Swinging Weight	2	14	12	15	35	78
Lift Instabillity	4	14	11	15	33	78
Perfect Form	7	14	6	15	35	78

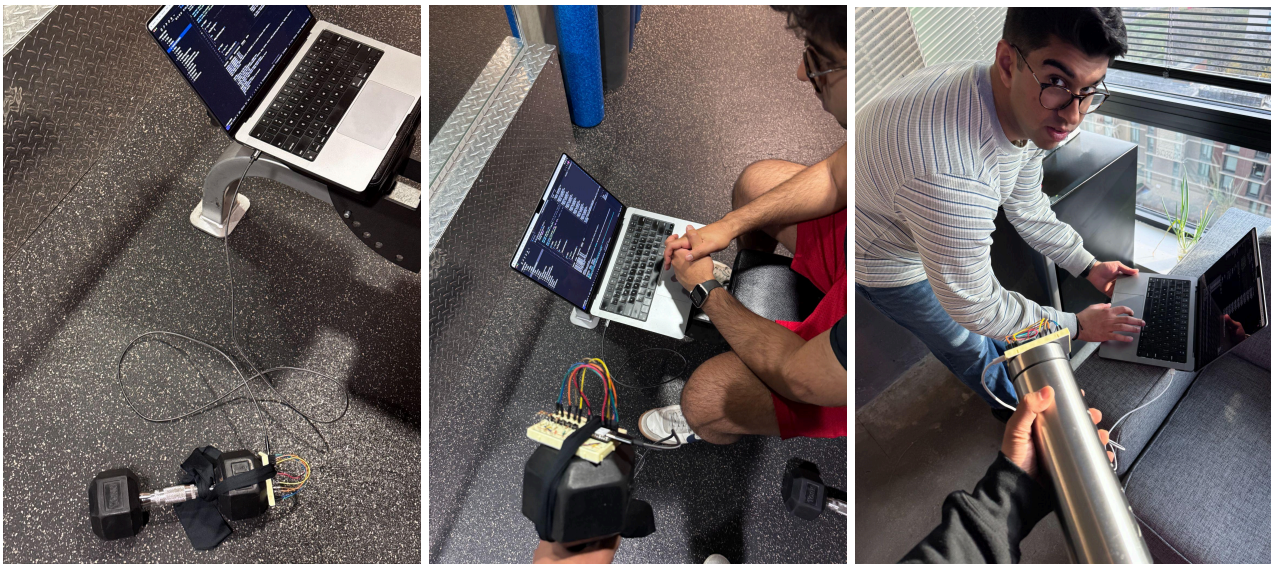
Data Collection:

Our data collection process involves taking a modified version of our device made of an XIAO, the MPU 6050 IMU connected with I2C, and six buttons (1 for each class). To record data, we simply pressed the button corresponding to the class, and the IMU recorded data and saved it for 5 seconds (sampling at 1 ms + 4 ms overhead). We then transferred the data to the laptop via serial every ~10 lifts and stored it in a file directory based on the class.

We created a custom JSON file that formats our data as needed (with lift type, classification, and time series data). A complete "formalized" schema is available in the GitHub repository as "ESE3600_input_format_schema.json."

```
{
 "lN": {"possible_values": ["dC", "bP", "dF"]},
 "lC": {"possible_values": ["p_f", "l_i", "p_m", "o_a", "s_w"]},
 "tSD": {
       "t": {},"aX": {},"aY": {},"aZ": {},"gX": {},"gY": {},"gZ": {}
}}
```

Data Collection Images



Data Cleaning and Processing (Model Training):

Given that we have the custom JSON file, the data is automatically labeled and formatted as we collect it. Since all data points are timestamped, we can delete any bad data points. We needed to do this a few times when we started collecting data, as the device would sometimes fall off or the person's form was not as expected.

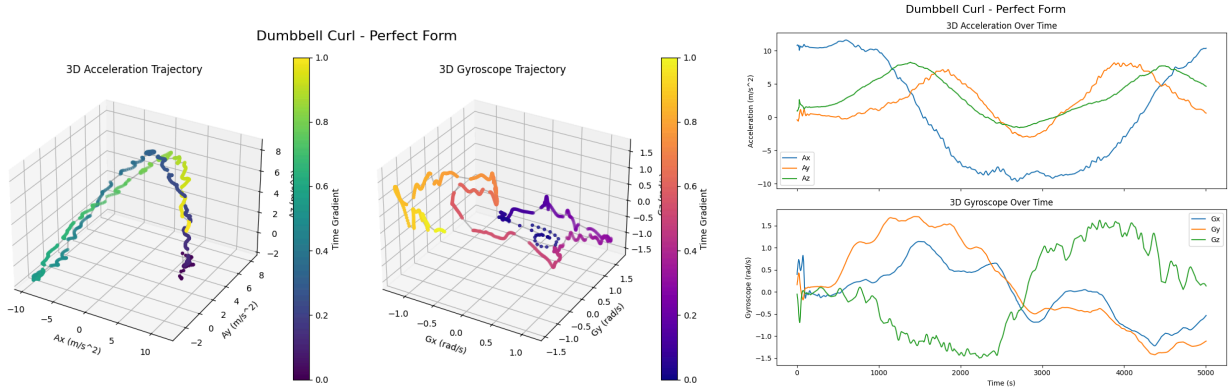
The pre-processing is as follows: We took the 5s of data sampled at 5ms (giving us between 960 and 1100 data points), we then either trimmed the head or tail (if > 1000) or duplicated the head and tail (if < 1000) to get exactly 1000 samples, finally, we downsampled (window avg 5:1) to get 200 data points; this helped to reduce the noise and also the input shape for the model size by 80%. We found the min and max value of the training set (offset by 10%) and normalized to [0,1].

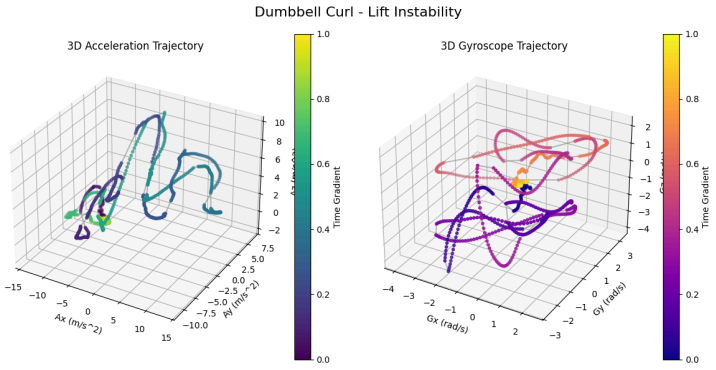
```
Normalization Parameters:
Acc Min: -25.09375, Acc Max: 30.8825, Gyro Min: -8.54875, Gyro Max: 7.995
```

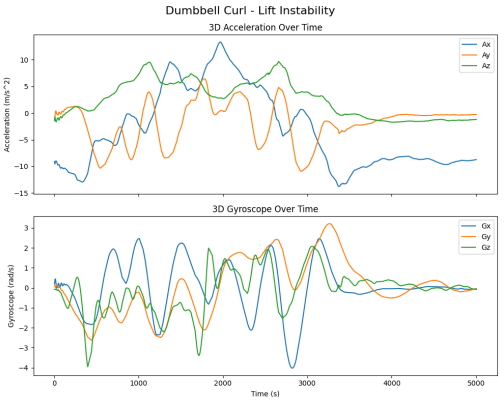
Data Visualization:

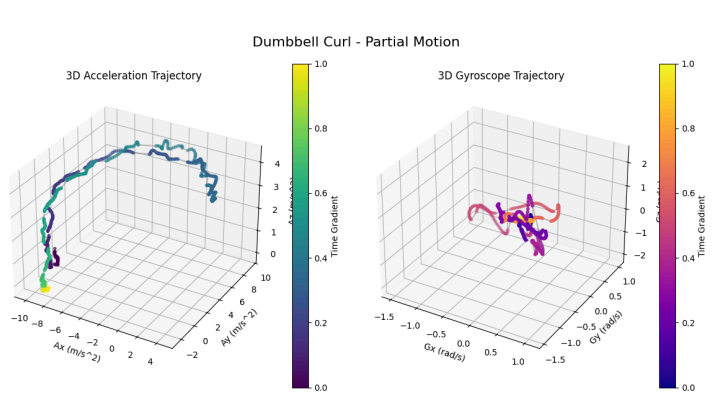
We thoroughly visualized the data at each step to help understand the data and build the preprocessing pipelines. We've attached images of different plots that we generated to accomplish this:

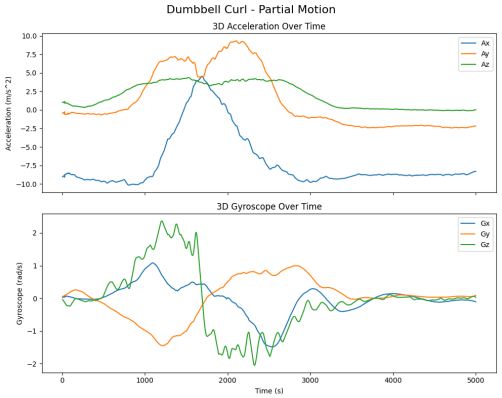
Initial Data Exploration Plots:



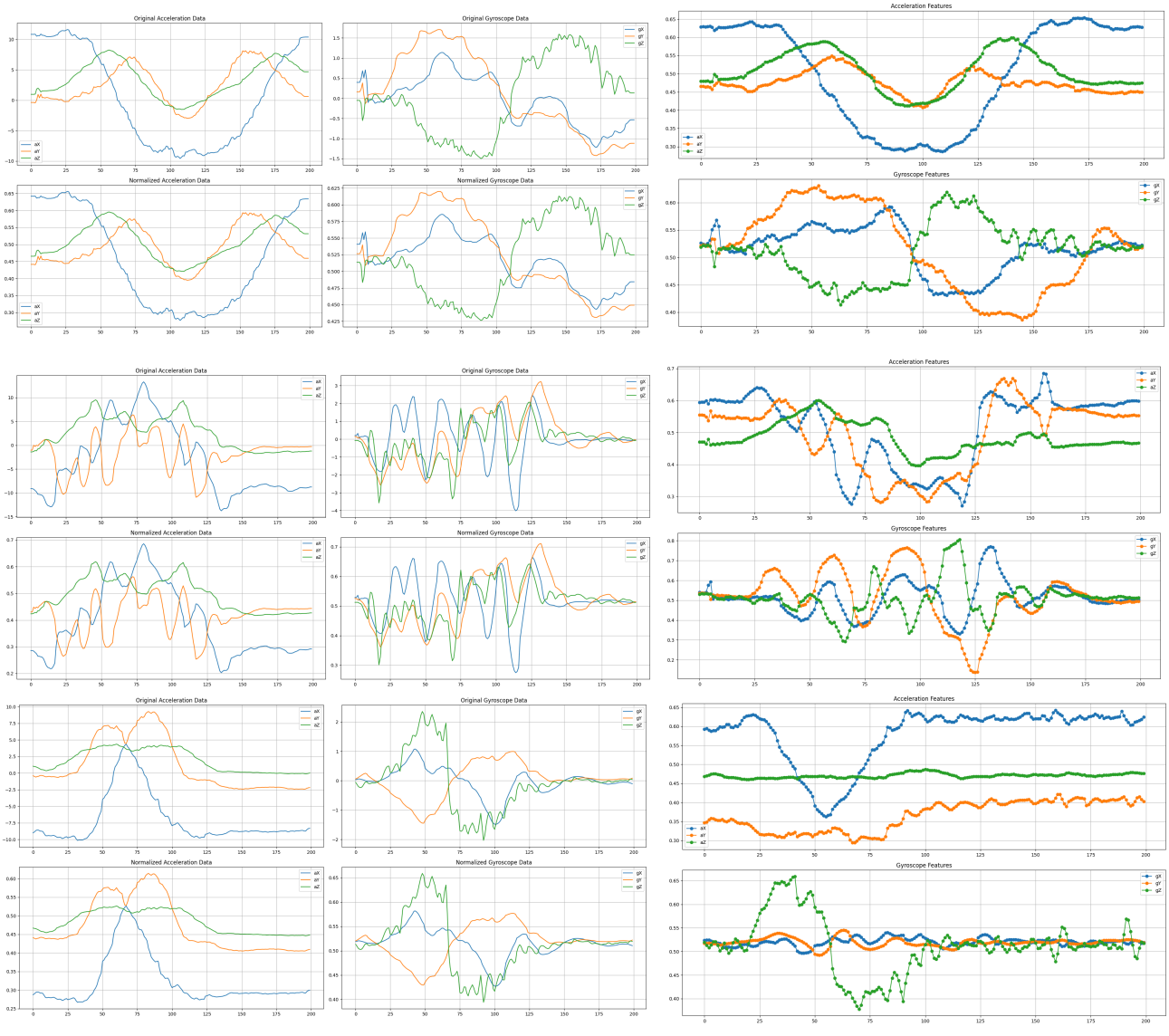








Normalization & Downsampling Plots

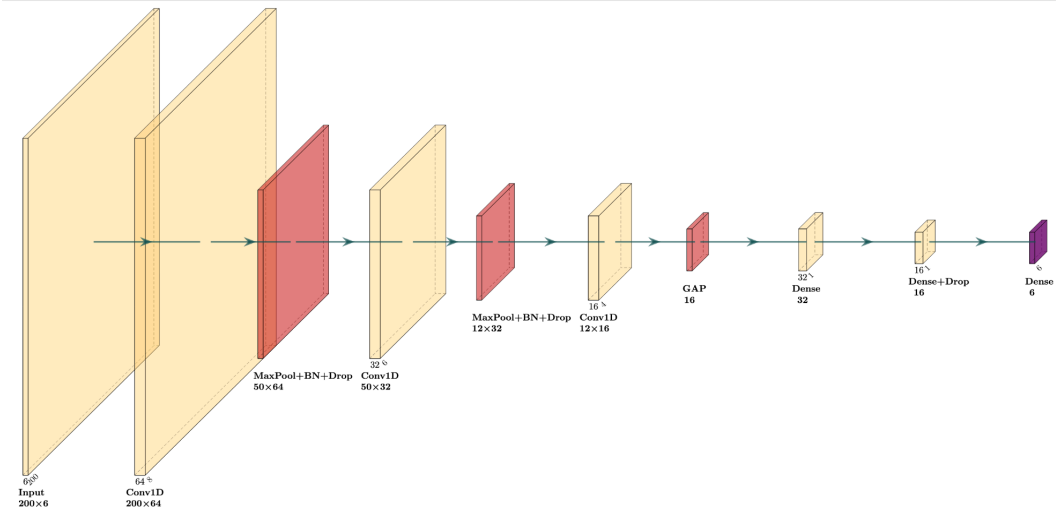


Model Choice and Design:

This weight lift classification model was built as a series of stacked feature extraction blocks, each composed of Conv1D layers that progressively reduce the number of filters (64 down to 32, then 16) before passing through a dense layer that similarly reduces dimensionality (32 to 16). The idea behind stacking these Conv1D layers is that the initial layers, with more filters, learn a broad range of complex patterns from the raw sensor data. In contrast, with fewer filters, subsequent layers `refine and distill these representations into more abstract and meaningful features.

Notably, we noticed that the acceleration data was being affected by the acceleration due to gravity, and this was a non-uniform effect that depended on the orientation of the device; by increasing the initial filter size, our hope is the model would “learn” to filter this out as inherently necessary. This also motivated us to add MaxPooling1D layers, which helps make feature detection invariant to scale and orientation changes. The ReLU adds non-linearity, while Batch Normalization accelerates the convergence of the model, the Dropout layers help to prevent overfitting.

Although using Conv1D layers prevents quantization due to lack of support within TFLite Micro, this trade-off results in a relatively small (~80 kB) and highly effective classifier. We experimented with alternative model architectures, only using supported operations within TFLite Micro, and found that our full-integer quantized model was ~60 kB while achieving similar accuracy. However, when performing inference at the edge, our post-quantized models performed significantly worse than our unquantized CNN model – likely due to its ability to extract robust temporal features crucial for distinguishing subtle differences in weight-lifting motions.



Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 200, 6)	0
conv1d (Conv1D)	(None, 200, 64)	3,136
re_lu (ReLU)	(None, 200, 64)	0
max_pooling1d (MaxPooling1D)	(None, 50, 64)	0
batch_normalization (BatchNormalization)	(None, 50, 64)	256
dropout (Dropout)	(None, 50, 64)	0
conv1d_1 (Conv1D)	(None, 50, 32)	12,320
re_lu_1 (ReLU)	(None, 50, 32)	0
max_pooling1d_1 (MaxPooling1D)	(None, 12, 32)	0
batch_normalization_1 (BatchNormalization)	(None, 12, 32)	128
dropout_1 (Dropout)	(None, 12, 32)	0
conv1d_2 (Conv1D)	(None, 12, 16)	2,064
re_lu_2 (ReLU)	(None, 12, 16)	0
global_average_pooling1d (GlobalAveragePooling1D)	(None, 16)	0
dense (Dense)	(None, 32)	544
re_lu_3 (ReLU)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
re_lu_4 (ReLU)	(None, 16)	0
dropout_2 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 6)	102

Total params: 19,078 (74.52 KB)

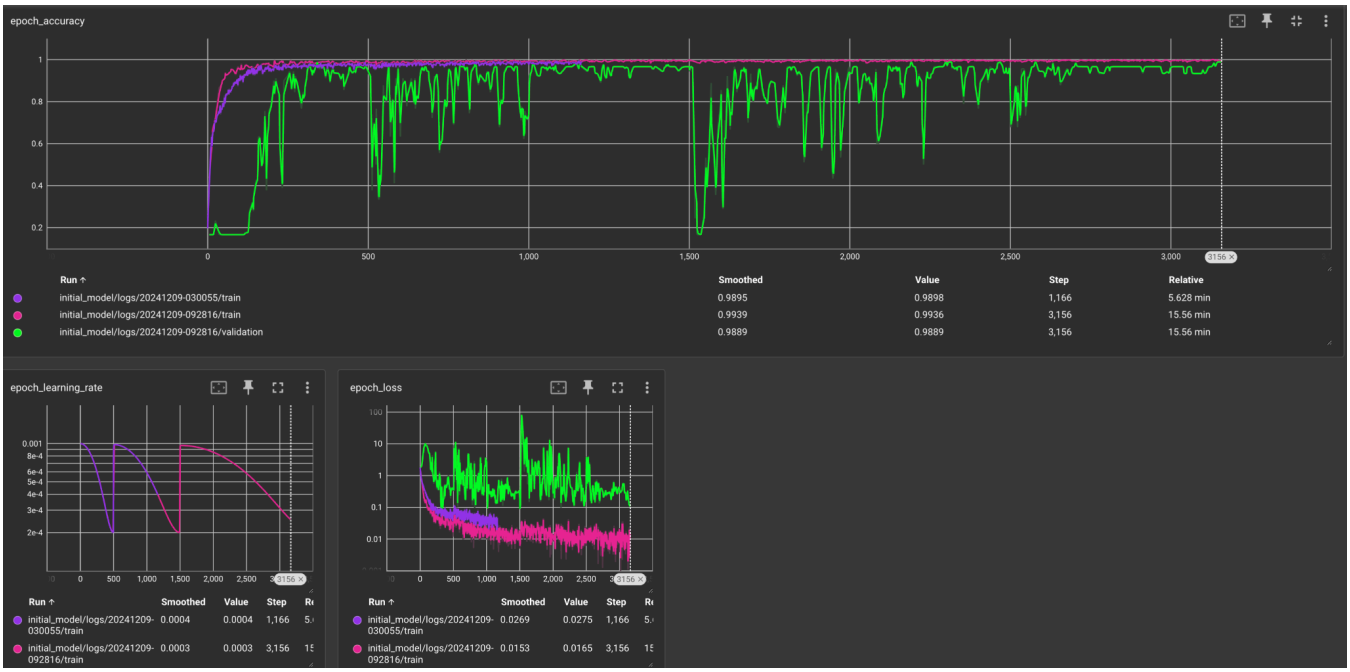
Trainable params: 18,886 (73.77 KB)

Non-trainable params: 192 (768.00 B)

Model Training:

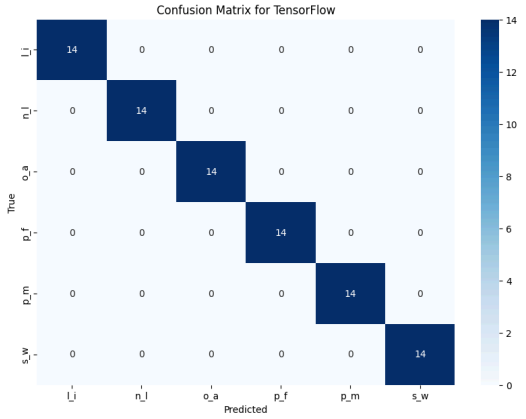
We selected CosineDecayRestarts for the learning rate schedule because it allows us to “restart” the learning rate if the desired accuracy isn’t reached within a decay cycle. The 50-epoch warmup prevents unstable early training. The first decay cycle of 500 epochs provides sufficient time for initial pattern recognition in motion sequences while doubling the restart cycle length (t_mul=2) allows for increasingly fine-tuned learning as training progresses. We maintain 98% of the peak learning rate on restarts (m_mul=0.98) to prevent too aggressive learning after each cycle, keeping a 20% minimum rate (alpha=0.2) to ensure continued learning capability in later epochs. AdamW was chosen over standard Adam to help prevent overfitting on specific motion patterns and make the model more generalizable. We set our max epochs to 5000.

We were training on TensorFlow Metal on M1 Macs, with each epoch taking roughly 100 ms (including validation eval) – making iteration easy. The early stopping callback, with a validation threshold of 0.97 and patience of 16 epochs, was determined as we would randomly hit our validation threshold goals, and our model would 1. have poor training set or 2. have poor edge performance. We were overfitting to the dataset. As the CosineDecayRestarts scheduler allows us to “restart” training, we found that integrating patience – or requiring the model to have at least n epochs > val threshold, would mean that our model had converged to a validation accuracy rather than temporarily spiking. We found that increasing this parameter increased the number of restarts – as the decay cycles got progressively longer – and it took 3157 epochs to finally reach our desired threshold. However, this model performed significantly better both through our test data and also while deployed on the edge.



Tensorboard Visualization of Training (3156 Epochs | 97% validation accuracy | 16 patience)

```
Maintained 0.97 accuracy for 16 epochs, stopping training!
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 74ms/step - accuracy: 0.9917 - loss: 0.0206 - val_accuracy: 0.9889 - val_loss: 0.1067
```



Model	Size	Reduction
TensorFlow	295,501 bytes	
TensorFlow Lite	85,320 bytes	(reduced by 210,181 bytes)

Model	Accuracy	Precision	Recall	F1 Score
TensorFlow	100.00%	100.00%	100.00%	100.00%
TFLite Float	100.00%	100.00%	100.00%	100.00%

Evaluation Results

We chose accuracy for evaluation metrics because it provided a simple, intuitive metric that directly measures the proportion of correctly classified instances out of all predictions. In a practical training scenario, users often care about how often the system is “right” without delving too deeply into the complexities of each type of error, so this metric makes sense. However, in working with the model more, we realized that if we were to work on this again, it might be helpful to consider other metrics.

Precision focuses specifically on how often the model’s positive predictions are correct. For example, if the model flags a specific lift as “off-axis,” precision would reveal what fraction of those flagged instances truly were off-axis. This would have been very useful as we noticed the model often favored one class over the others (in our final model, it was off-axis), and using recall as a metric would have helped us notice this and account for it in training.

One of the other issues we saw in the model was that even though we were hitting high validation accuracies, we needed better performance on-device. This led to an investigation with three outcomes.

First, we wanted to ensure that the model on the edge device would provide the expected output if given a well-formed input. In doing so, we would verify that our C++ implementation on the XIAO for inference was bug-free and mirrored the Python TFLite interpreter results. To do this, I had the Python notebook output the pre-processed test data in C++, then simply copied that over and ran inference on all those inputs. Once we verified that the results were identical, then we knew that we could attribute all future outputs to one of 3 categories:

1. The model needs to be improved.
2. The pre-processing on the edge deployment needs to be improved
3. More data is needed

On the first point, we realized that the model didn’t generalize the features well. With a simple validation accuracy callback, we would essentially capture an overfitted model when the validation accuracy happened to spike for an epoch. We utilized patience as a metric to mitigate this, requiring the model to remain steady at 5, 10, and then 16 epochs for our final model. At the same time, this significantly increased the training time; we saw a drastic improvement in the edge inference. Our takeaway is that these metrics, especially on smaller datasets, aren’t the end-all-be-all; you need to run edge inference while your full pipeline is working. Our metrics didn’t drastically improve between Patience of 1, 5, 10, and 16, but the testing we did (randomly selecting a class and doing a lift, five for each of the six classes) had a substantial improvement.

Next, we wanted to ensure that pre-processing on the edge matched the pre-processing we used to test the notebook. Checking correctness was trivial, simply recreating the pipeline in C and then squashing bugs until a few selected samples matched between the two. We initially attempted to implement a circular buffer approach for greater user flexibility so the users could lift at their own pace, but this introduced many complications. For reasons we cover later, we decided to remove it. By syncing our output to a timer, we better centered our data within the time series – like in the input data. This also had tremendous real-world gains.

As a result, we reverted to triggering data recording with a buzzer, which unfortunately forced users to time their lifting activities fixedly. While this was not ideal, we noticed that the inconvenience would be manageable because people tend to follow a steady lifting rhythm and naturally pause between reps.

Finally, from our testing with the device, we noticed the subtle differences between different classes, like partial motion or off-axis lifts, remained difficult for the model to differentiate reliably. Improving this requires more extensive and varied training data from more users and having “experts” judge the form. However, in its current state, the issues in differentiating subtle classes also lowered the user experience.

Deployment and Hardware Details:

We used the XIAO ESP32S3 board for this project, integrating it with our primary sensor, the MPU 6050 IMU. However, the breakout board connections for the XIAO and the I2C breakout were flaky, so we used a breadboard instead.

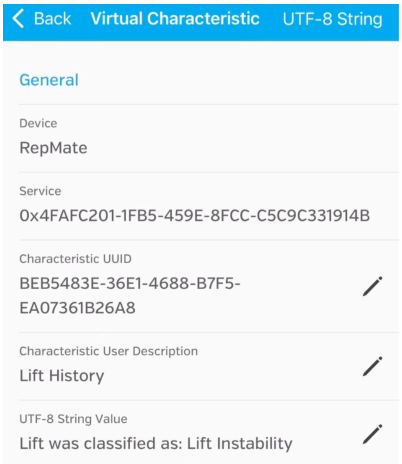
We added six buttons for data collection, each connected to a digital input pin on the microcontroller. These buttons also had a pull-down resistor, so the input would never float. This allowed us to collect data easily: We could push the button and collect the data for 5 seconds, and the microcontroller could format the data easily using the JSON format, as it would know what lift type we had just recorded (due to the button).

For the final prototype, we removed the buttons and added five lights, one for each lift type (excluding no lift and unknown, as we wouldn't need to alert the user for these types). When the model detects one of these lift types, the digital pin will go high, and the LED will light up. We also have a buzzer that sounds when a lift is detected so the user knows as they are lifting.

One of the more complicated aspects of the hardware to make function was using Bluetooth Low Energy (BLE) to send data to the user's phone (so that we wouldn't just read values out over serial). We used the LightBlue app, created a virtual BLE device as a server, and configured our XIAO as a client device.

Our code builds on the XIAO BLE client example code and provides a robust implementation of a BLE client that interacts with the BLE server on the LightBlue app. The workflow begins with the BLE setup function, which initializes the BLE stack and starts a scan to discover nearby devices advertising their presence. The MyAdvertisedDeviceCallbacks class handles discovered devices, checking for specific criteria like a matching service UUID or device name (we found device name to be more reliable in finding the correct BLE device as in rooms with a large number of devices, filtering our unnamed devices saved time in establishing a connection). When a target device is identified, the doConnect flag is set, triggering an attempt to establish a connection in the BLEloop function via connectToServer. This function handles connecting to the server, discovering the desired service and characteristic, and optionally enabling real-time notifications to receive updates.

The LightBlue app acts as the server by advertising the predefined service UUID and exposing characteristics the client can read, write, or subscribe to for notifications (in our case, we gave the client full access to every characteristic). This setup allows the BLE client to interact dynamically with the server's data, such as reading current values, sending updates, or receiving notifications of changes. This approach gives the essentials of working with BLE devices: discovering devices, identifying services, connecting to the server, and interacting with its characteristics.



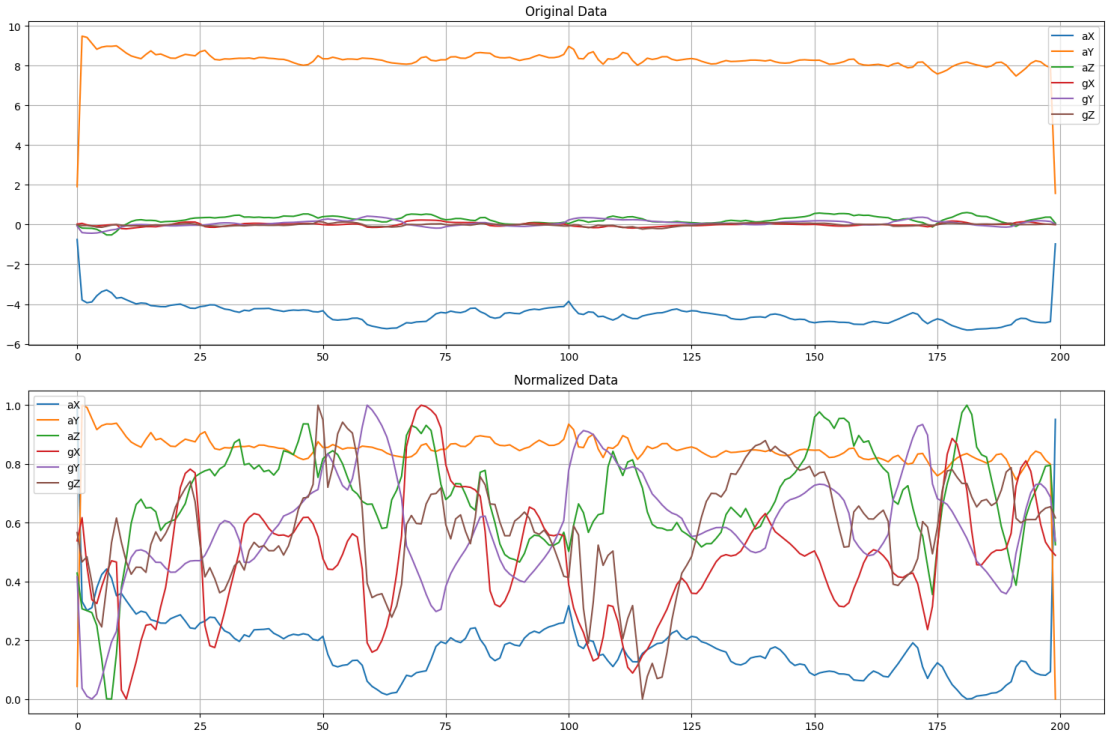
Output of inference to LightBlue app (using BLE)

Challenges:

Our project has overcome several significant challenges throughout its development, highlighting the complexity and learning opportunities inherent in embedded machine-learning systems.

One of the early technical obstacles was the flaky connection to the breakout board, particularly the I2C connection to the MPU sensor. This instability caused frequent data dropouts and unreliable sensor readings. After troubleshooting, we discovered that the issue stemmed from the wiring and pin connections, which were exacerbated by the setup on the breakout board. Transitioning to a breadboard resolved this issue by providing more secure connections, allowing for stable communication between the microcontroller and the sensor.

When we first ran the machine learning model on the embedded device, the output consistently classified movements as either swinging weight or no lift, failing to recognize the full range of expected activities. Closer data analysis revealed that our normalization process needed to be revised. Instead of using a fixed global min and max for normalization, we normalized the data to the minimum and maximum values within each lift. This approach amplified noise in the signal, leading to erratic predictions (as illustrated in the data below). The solution was to normalize the data using the min and max values from the training dataset, ensuring consistent scaling and improving the model's inference performance. This adjustment significantly enhanced the reliability of our predictions across various lifting scenarios.



An example of the detrimental effects of incorrect quantization (No Lift)

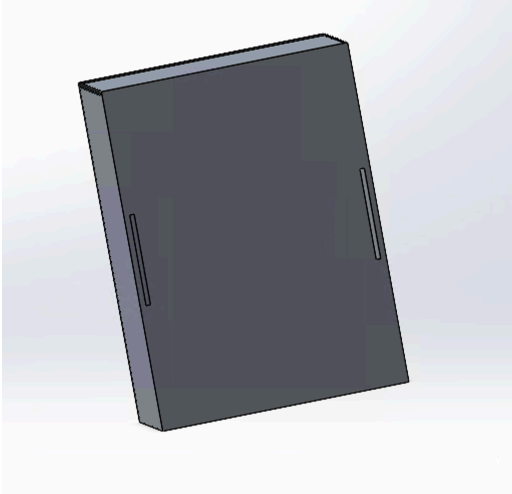
Another issue we ran into was implementing a circular buffer to allow us to stream data and perform inference continuously. There were a few key issues with this approach–both from a technical perspective and from a user experience perspective. First, the buffer added additional overhead to all of the operations. With inference taking ~300 ms, the buffer would have “data gaps,” which entirely killed the point of having a continuous data stream. From a user experience standpoint, it makes more sense to have the device to indicate when to perform a rep – collect data for that 5-second window – perform inference – and then delay for another ~1.7 seconds. We were able to both significantly simplify the pre-processing pipeline and also provide a strong cadence for the user – something that enhances the user experience.

Another significant hurdle arose when integrating Bluetooth Low Energy (BLE) functionality into the system. Our initial implementation configured the XIAO microcontroller as a BLE server, but this approach caused the device to crash due to excessive RAM usage. The server configuration consumed approximately 80% of the available RAM, leaving insufficient resources for the model and other critical functions. Reviewing the design, we realized that the phone, not the XIAO, should be the BLE server. By reconfiguring the system to use the phone as the server and the XIAO as the BLE client, the memory usage dropped to just 10-15%, resolving the crashes. This change optimized resource allocation and aligned with the proper BLE device roles, enabling proper communication between the microcontroller and the phone's virtual devices.

While initially setbacks, these challenges have provided valuable insights into hardware integration, data processing, and efficient resource management in embedded systems. Each solution has incrementally improved the system's stability, functionality, and performance, paving the way for a robust final implementation.

Future Work:

If we had the time, there are a few features we would want to explore. One key feature would be the physical integration of the device. We have already designed a 3D enclosure to house the components securely (see image below), and the next step is to 3D print this enclosure and test its suitability for real-world use. A robust and ergonomic housing will protect the electronics while ensuring the device is portable and convenient for users. This step will also allow us to refine the device's form factor, potentially making it more visually appealing.



Our designed enclosure

Functionality is another area where we see significant potential for improvement. Currently, the system is designed to accurately classify specific types of lifts, but we aim to expand its capabilities to handle a broader range of lift types. This would involve gathering additional training data for lifts such as deadlifts or overhead presses and refining the machine learning model to recognize these new movements. Such improvements would make the device more versatile and appealing to a wider audience of fitness enthusiasts with diverse workout routines.

Additionally, we considered redesigning the device to increase its compatibility with various types of weights and exercise equipment. For example, the current design works well with dumbbells, but adapting it for use with Smith machines or barbells would significantly broaden its application. This redesign might involve adjusting mounting options or incorporating sensors that handle different attachment points and orientations. Furthermore, we need to refine the model to handle this ambiguity related to orientation and mounting.

Another feature we want to implement is a speaker system that announces lift results audibly in real time. This addition would enhance the user experience by providing immediate feedback during workouts, allowing users to stay focused on their lifts without needing to check a phone or display. The audible feedback could include metrics such as lift type, weight, and count, making the device more interactive and accessible, especially in scenarios where visual displays are impractical.

Finally, if we wanted to bring this to market as a “product,” here is the path forward: Change this to a wearable form factor—it's easier for gym goers or people to wear it, either through custom hardware or converting it to an Apple Watch app. For the second option, we can integrate this into Apple Workouts, providing a new dimension to users and creating custom cadencing that is announced via your AirPods. People can log the weights and let the watch manage the reps. From there, one could make all types of metrics and visualizations.

Poster (Next Page):

RepMate: Your ML Powered Lifting Coach

Rohan Panday (EE '26), Vedansh Goenka (CIS '26)

Penn

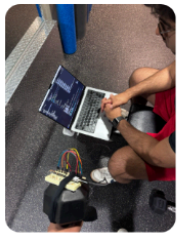
TinyML @Penn

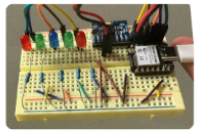
Problem

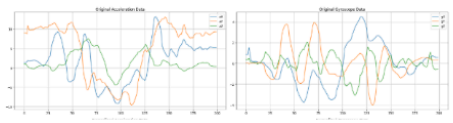
Lifting weights is essential to many people's fitness journey, but ensuring proper form is crucial to avoid injury and achieve results, especially for novice lifters who may not know what "good form" looks like. Improper lifting techniques can lead to injuries or reduced gains.

Goals

This project aims to develop an attachable smart device that connects to a smartphone via Bluetooth to monitor and evaluate lifts at the gym. The device will recognize proper and improper lifting techniques (such as shaking, incomplete motion, or tilting) and provide instant audio feedback to guide the user so they know how to correct their lifts mid-rep, exactly how a real coach would.







IMU Accelerometer Data

Data cleaning, processing, and normalization

Data goes to chosen model

Bench Press Model

Dumbbell Curl Model

Dumbbell Fly Model

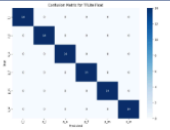
Inference

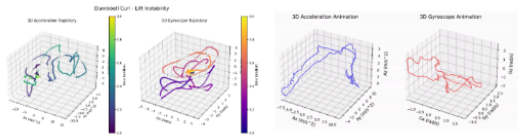
UTF-8 String Value

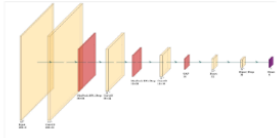
Lift was classified as: Lift Instability

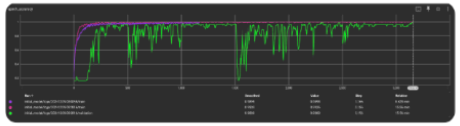
LED lit to indicate lift type

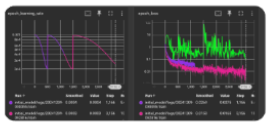
Transmit data to phone via BLE











Data Collection

- **3 lift types:** bench press, dumbbell curl, dumbbell flys
- **6 lift classes:** perfect form, no lift, swinging weight, off-axis, partial motion, lift instability
- **4 individuals**, both their **left** and **right** arms
- **100-250 data points** per lift type, split evenly between the classes
- Data augmented by adding **noise**, **drops**, **magnification**, and **axis inversion**
- Custom JSON to format and auto-label data as it is collected
- Data **normalized** to training data max/min

Model

Stacked blocks of:

- Conv1D (64 → 32 → 16) Dense (32 → 16)
- ReLU [1 - 5]
- MaxPooling1D [1, 3]
- Batch Normalization [1, 2]
- Dropout [1, 2, 5]

Used 5 total blocks, ending with a softmax

Avoided Quantization of the model

- TFLite Micro doesn't support Conv1D operations
- Small size model (~80 kB)

Training

Training Setup:

- Total epochs: 5000 (limit)
- Initial learning rate: 0.001
- Loss: Categorical Cross Entropy
- Metric: Val Accuracy (0.97 w/ 16 patience)

Learning Rate Scheduler:

- Cosine Decay with Warmup

Optimizer (Adam)

- Weight decay: 0.0005
- Beta1: 0.9
- Beta2: 0.999
- Epsilon: 1e-07

Checkpointing

- Saves model every new "max" value

Results + Demo

Metrics:

- Tensorflow (296 kB) → TFLite (85 kB) ↓ **71%**
- Tensorflow (100%) → TFLite (100%) ↓ **0%**

Future Improvements:

- Significantly increase data gathered
- Reduce overall footprint of device
- Signal Processing (Unscented Kalman Filter)

Demo Video: