Lying Posture Detection using IMU Sensor and Machine Learning

Tulasi Sainath Polisetty
ASUID: 1223964586

October 15, 2023

1 Requirements Document

1.1 Overview

Objective: The primary motivation for this project was to develop a portable, sophisticated system for real-time posture monitoring. This system would employ an IMU sensor unit embedded in an Arduino board, extracting signals from 3-axis accelerometer, gyroscope, and magnetometer sensors to simulate various postures. The ultimate ambition was to design a machine learning algorithm capable of distinguishing between different postures, and then implementing this algorithm on a microcontroller for real-time predictions.

Process: The endeavor began with data collection using the IMU sensors, mimicking various postures. This data was then processed, labeled, and segmented to train a machine learning model. The model, once validated and tested, was integrated into the Arduino microcontroller, enabling real-time posture detection.

Interaction with Existing Systems: While this was primarily a standalone system, it required communication with a base station, such as a laptop or smartphone, from which commands would be sent to the microcontroller to initiate real-time posture prediction.

1.2 Function Description

Prototype: The prototype system was equipped with IMU sensors attached to an Arduino board, coupled with a machine learning model integrated into the microcontroller. This enabled the prototype to execute real-time posture detection and interface with a base-station to display prediction results.

Performance: The system was designed to be efficient and accurate. Initial benchmarks were set by the raw LSTM model's accuracy scores. Despite the transition from LSTM to a simpler Dense Neural Network model due to microcontroller constraints, the performance criterion remained uncompromised.

Usability: Given the real-time monitoring objective, the system was designed for seamless user interaction. Though improvements were slated for the user interface, the primary focus remained on ensuring accurate posture detection.

1.3 Deliverables

- Fully integrated Arduino system capable of real-time posture detection.
- A comprehensive report detailing the project's objectives, methodologies, results, and challenges.
- Dataset curated for posture detection.

2 Experiment

2.1 Design & Environment

The experimental design was devised to capture various postures, including supine, prone, side (either right or left), sitting, and an 'unknown' posture category. Data collection exercises were conducted at a home workstation to ensure a controlled environment.

2.2 Data Collection & Labeling

Python scripts were utilized to label data segments appropriately. Sensor readings, which mimicked different postures, were stored in separate files for easier access and processing. A moving window, approximately 2 seconds in duration, was used to segment the data, which facilitated its subsequent use in model training.

3 Algorithm

3.1 Architectural Evolution

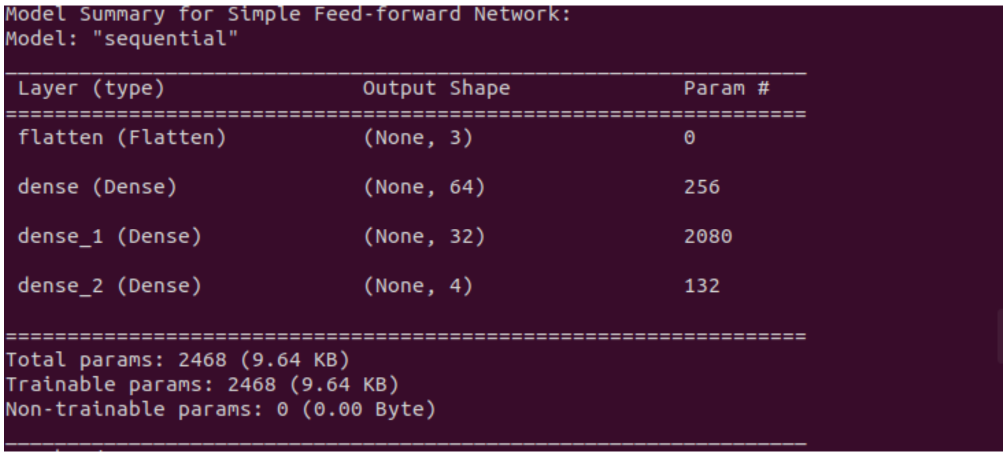
In the initial stages of the project, an LSTM (Long Short-Term Memory) model was considered. LSTMs, a subset of Recurrent Neural Networks (RNNs), excel at detecting patterns in sequential data because of their capacity to retain long-term dependencies. This feature made them an appealing choice for monitoring posture, a task fundamentally about identifying patterns in sequential sensor data.

However, difficulties arose when adapting the model for deployment on a microcontroller. TensorFlow Lite's interpreter struggled with certain custom operations inherent to the LSTM, prompting a reevaluation of the model's architecture.

Following this, I turned my attention to the Gated Recurrent Unit (GRU) model. Like LSTMs, GRUs are structured to recognize dependencies in sequences but boast a more streamlined architecture. I anticipated that this simplicity would facilitate the model's deployment on a resmyce-limited microcontroller.

Nevertheless, obstacles remained. Due to the restrictions of my Arduino-based system and the primary objective of real-time posture detection, a more efficient and direct model became essential. Consequently, the focus shifted to a Simple Dense Feed-forward Neural Network.

3.2 Simple Dense Feed-forward Neural Network



```
Model Summary for Simple Feed-forward Network:
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 3)                 0

 dense (Dense)               (None, 64)                256

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 4)                 132

=================================================================
Total params: 2468 (9.64 KB)
Trainable params: 2468 (9.64 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 1: Feed-Forward Model

The finalized architecture consisted of:

- **Flatten Layer:** This layer modified the incoming data to make it compatible for the layers that followed. As the data arrived in a 3-dimensional format (representing x, y, and z readings), the Flatten layer adjusted it into a single dimension.
- **Dense Layer (64 Neurons with ReLU activation):** The Rectified Linear Unit (ReLU) activation function was preferred due to its computational efficiency and its ability to introduce non-linearity into the model.
- **Dense Layer (32 Neurons with ReLU activation):** An additional layer designed to discern more complex patterns in the data.
- **Output Layer (4 Neurons with Softmax activation):** The primary role of this layer was to determine the probabilities for each of the fmy postures. The Softmax activation ensured the output values fell within the (0,1) range, and their total equaled 1, allowing them to be interpreted as probabilities.

Despite its seeming simplicity, this Feed-forward Network was meticulously tuned for the task, balancing both computational efficiency and accuracy.

3.2.1 Training Process

Data collected from the IMU sensor, representing various postures, forms the training dataset. This data undergoes pre-processing, including normalization, to ensure it's in a suitable format for training. Once the data is ready, it's fed into the model, which then adjusts its weights and biases based on the prediction error. This process is iteratively performed until the model's prediction error converges to a minimum value or until a pre-defined number of iterations is reached.

3.2.2 Training Parameters

Key training parameters include the learning rate, which dictates the step size during weight updates, and the batch size, which determines the number of data samples considered in each training iteration. Furthermore, the model's architecture, specifically the number of layers and neurons in each layer, also plays a critical role in the training process.

3.3 Deployment on Arduino

After training, the model was translated to the TensorFlow Lite format (.tflite) and then converted into a .cpp format appropriate for Arduino deployment.

On the Arduino, the main loop awaited instructions from the base station. When a specific sensor (accelerometer, gyroscope, or magnetometer) was selected, the related sensor data was collected, refined, and subsequently input into the deployed model for analysis. The outcome, namely the predicted posture, was then relayed back to the base station.

Nevertheless, certain challenges, specifically in the tensor loading phase, surfaced and are currently under active exploration.

3.4 Deployment Challenges

After the completion of the model's training, its adaptation to the Arduino microcontroller presented certain challenges. One primary issue was during the tensor loading phase. Specifically, bottlenecks related to data normalization processes emerged. This was integral as the model's training had involved Min-Max normalization where data values close to the minimum and maximum of the training dataset were adjusted to a specific range. However, during inference, the device's range of -4 to 4 for accelerometer values posed a significant mismatch with the training data range, which was roughly from about -0.7 to 0.9. Such disparities, if not appropriately addressed, could lead the model to misinterpret or inaccurately predict data, potentially affecting the model's efficacy in real-world scenarios.

3.5 Solution to Deployment Challenges

The critical issue faced during deployment was ensuring the data input into the model during inference was consistent with the data it was trained on. The model was trained on normalized data, and to maintain consistency, the data extracted from the IMU during real-time detection had to undergo the same normalization process.

The implemented solution involved calculating and setting constants from the training data statistics:

- Mean values for x, y, and z axis.
- Range values, derived from the summation of the maximum and minimum values for each axis.

A normalization function was then crafted to adjust the live sensor readings:

$$\text{normalized\_value} = \frac{\text{value} - \text{mean}}{\text{range}}$$

This normalization ensured that the values from the accelerometer, which inherently lie in the device's range of -4 to 4, were transformed to a range that the model was familiar with from its training phase. This alignment of input data format between training and inference phases was paramount to ensure the model's effective operation

3.5.1 Microcontroller and Base-Station Coordination

The coordination between the microcontroller and the base-station is established using serial communication. A Python script serves as the interface on the base-station, allowing the user to select the desired sensor for prediction, be it the accelerometer, gyroscope, or magnetometer. Based on the user's choice, a specific command is sent to the microcontroller via serial communication.

Upon receiving the command, the microcontroller collects the relevant sensor data, performs necessary pre-processing, feeds it to the pre-trained machine learning model, and then makes a prediction. The prediction result, representing the detected posture, is then transmitted back to the base-station, where it's mapped to a human-readable format and displayed to the user. This interactive cycle allows for on-demand posture detection, ensuring efficient and user-friendly operation.

The Python script, serving as the base-station interface, is designed for continuous user interaction. It repeatedly prompts the user for sensor selection, performs real-time posture prediction, and displays the result. Additionally, the script includes safeguards to handle potential communication errors, ensuring reliable and uninterrupted operation.

4 Results & Analysis

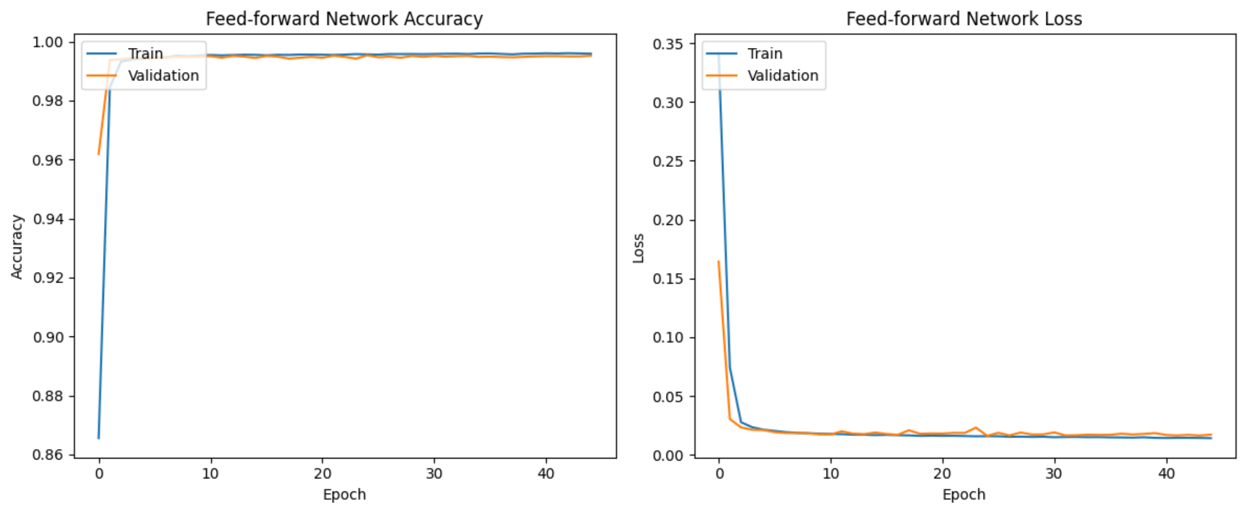
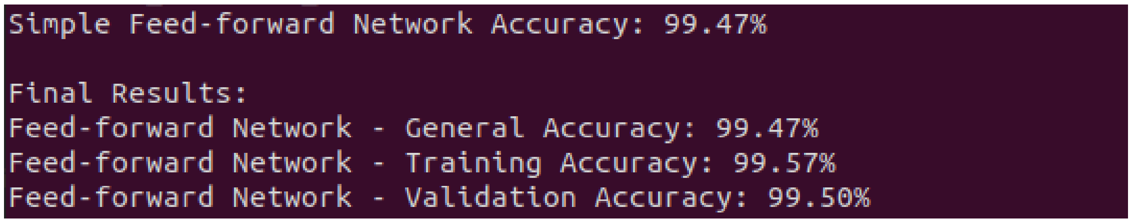


Figure 2: Accuracy and Loss plots for the feed-forward network.

The primary metric for model evaluation was accuracy. After an exhaustive training phase, the model was benchmarked against multiple datasets to validate its precision and reliability.

The results were promising; real-time predictions by the deployed system closely aligned with test dataset outcomes, reiterating the model's proficiency. It is noteworthy that the model managed to predict 3 out of the 4 classes with remarkable accuracy. This might be attributed to the distinct accelerometer characteristics inherent to these three postures. Still, the normalization intricacies during inference could potentially be the key to further refining the accuracy of the fifth class.



```
Simple Feed-forward Network Accuracy: 99.47%

Final Results:
Feed-forward Network - General Accuracy: 99.47%
Feed-forward Network - Training Accuracy: 99.57%
Feed-forward Network - Validation Accuracy: 99.50%
```

Figure 3: Accuracy of the feed-forward network.

4.0.1 Performance Metrics

The primary metric for evaluating the model's performance was accuracy, which signifies the percentage of correctly classified samples out of the total samples.

- **General Accuracy**: 99.47%
- **Training Accuracy**: 99.57%
- **Validation Accuracy**: 99.50%

4.0.2 Analysis

The high accuracy scores across the board are indicative of the model's robustness and its ability to generalize well to unseen data. The marginal difference between training and validation accuracy suggests that the model did not overfit to the training data, which is a positive sign of its applicability to real-world scenarios.

4.0.3 Real-time Prediction Performance

When deployed for real-time posture detection, the model showcased exceptional accuracy, particularly when leveraging data from the accelerometer. This performance was closely aligned with the results achieved on the test set. The consistent outcomes between real-time predictions and the test set results re-emphasize the model's adaptability and aptitude in practical scenarios. It's significant to note that during real-time evaluations, the model encountered various environmental and human dynamics. Despite these complexities, its performance, especially with the accelerometer, was outstanding.

The gyroscope, while being the second most reliable sensor, exhibited commendable results, although slightly inferior to the accelerometer. On the other hand, the magnetometer's predictions were the least precise among the three, suggesting its sensitivity to environmental disturbances or perhaps a need for more specific data preprocessing tailored to its characteristics.

4.0.4 Comparison

Drawing a parallel between the model's performance on the test dataset and its real-time predictions, one can discern a pattern of consistent proficiency. The high general accuracy of 99.47% underlines the model's reliable predictive capacity. This is especially true when the model processes accelerometer data, signifying its preeminence in capturing posture nuances. While the gyroscope's readings were commendably accurate, they didn't quite match the precision achieved by the accelerometer. The magnetometer, being the least accurate, might require further exploration or refinement to improve its predictive contribution in real-time scenarios.

5 Discussions

5.1 Summary of Results

The endeavor to detect posture in real-time using an IMU sensor and machine learning culminated successfully. The system's capability spanned from meticulous data collection and processing to deploying a robust machine learning model that could discern between different postures. However, the intricate landscape of real-world applications presented challenges that tested the system's adaptability and accuracy.

5.2 Challenges and Difficulties

A paramount challenge arose in engineering a posture detection system that was both orientation insensitive and sensor-agnostic. The ideal vision was a system adept at detecting postures, irrespective of sensor orientation or its brand/model. However, varied sensor calibrations and the inherent discrepancies in how different sensors register data introduced inconsistencies. These disparities could skew results, thereby impeding the system's universal accuracy in posture detection.

Additionally, translating the controlled successes of the model to a real-time framework was challenging. While the model demonstrated precision in controlled settings, the unpredictable nature of real-time scenarios, influenced by human dynamics and environmental variables, presented hurdles.

5.3 Looking Forward: Solutions and Improvements

Addressing orientation insensitivity could involve the infusion of advanced data augmentation techniques, or the incorporation of orientation correction algorithms during data preprocessing. Tackling the sensor-agnostic challenge might necessitate a more holistic approach: perhaps by sourcing data from an array of sensors and then training the model to be more accommodating of input variations.

Reflecting on the project, the most intricate segment was bridging the conceptual realm with the tangible, especially when venturing into the domains of the Arduino platform and real-time predictions. There is an evident need for refining the system, adapting it to effectively navigate the idiosyncrasies of live data versus controlled datasets.

In reflection, while the system's real-time predictions were good, they didn't meet the full expectations in terms of accuracy. Going forward, there's clearly a need for improvements, especially concerning the accuracy of all sensors. I plan to refine the model and consider newer machine learning techniques to enhance its real-time performance.