Lying Posture Detection using Embedded Machine Learning.

System Design:

- Our lying posture detection project is driven by the potential to make a substantial impact in two critical domains: the assessment of sleep quality and the prevention of pressure ulcers. By effectively identifying a person's sleeping posture, we can acquire valuable information about their sleep patterns, ultimately contributing to the enhancement of their sleep quality. Moreover, the recognition of postures like supine and prone positions can play a crucial role in averting pressure ulcers, which can lead to severe health complications.
- To accomplish this task, we make use of IMU sensors. Specifically, we utilize the accelerometer to gauge acceleration along the x, y, and z axes. Using these acceleration data, we compute the angle 'theta' relative to the gravitational force. The magnitude of the z-vector, when combined with 'theta,' plays a crucial role in determining the individual's lying posture.

Our observations have revealed key insights:

➢ Acceleration Range:

- Typically, acceleration values along the x, y, and z axes range between -1 and 1, indicating the sensor's responsiveness to orientation changes.

➢ Data Sampling Rate:

- To prevent data redundancy and account for the sensor's gradual rotation while being held, we have configured the data sampling rate to capture one measurement every 200 microseconds.

Our approach to posture detection involves a 6-layer feed-forward neural network with a neuron distribution of (256-128-64-32-5) across the layers with three input channels. This architecture is designed with the intention of producing probabilities for each of the five posture cases as the output.

Deliverables:

In summary, our project's significance lies in its potential to enhance sleep quality assessment, prevent pressure ulcers, and respect privacy concerns. By leveraging IMU sensors and machine learning techniques, we aim to accurately recognize and classify various lying postures, ultimately contributing to improved health and well-being.

Experiment:

- We initiated our data collection process by initializing the IMU sensors and gathering acceleration data. To acquire this data, we manually rotated the sensor gently. These experiments took place in an indoor setting, deliberately chosen for its minimal external disturbances due to the absence of airflow.

- It's important to note that throughout the experiments, we maintained a USB connection with the Arduino board. This connection introduced slight disturbances in the sensor readings, which can be considered as noise. We had to account for and mitigate this noise factor in our data preprocessing and modeling steps.
- For assembling data related to the 'unknown class,' we strategically relied on the transitional periods that occurred between the known classes. These transitional phases provided valuable data points to train our model in recognizing scenarios where the posture did not belong to any of the predefined classes.
- An essential insight that significantly contributed to our classification process was our understanding of how the IMU unit references the gravity vector. This understanding enabled us to achieve a high level of accuracy and confidence in classifying all five posture cases. We utilized Theta_y (Theta about the y-axis), Theta_x (Theta about the x-axis), and the acceleration values along the respective axes (ax, ay, az) for this purpose.
- Specifically, we calculated Theta_x as the tangent of the ratio between ax and az, and Theta_y as the tangent of the ratio between ay and az.
- Addressing the challenge of achieving orientation-independent posture classification, we decided to treat the upright and downright positions as equivalent, essentially considering them as the same posture. In our approach, we categorized the downright position as part of the 'unknown' class, simplifying our classification task. This strategic decision led us to amass a substantial dataset comprising 1000 sensor readings, each associated with corresponding labels and acceleration values.

<u>**Algorithm:**</u>

- We chose to use a feed-forward neural network, which operates by transmitting information unidirectionally, moving from the input layers to the output layers, without involving any feedback loops or cycles.

➢ Architecture:

1. Layer Configuration:

- Within our network architecture, we've employed a total of five layers. These encompass an initial input layer, followed by three concealed layers, each comprising 128, 64, 32 and 16 neurons, respectively. The final layer, which serves as the output layer, consists of 5 neurons. This specific layer structure was chosen due to the nature of our task, which is a classification problem involving five distinct classes.

2. Activation Function Exploration:

- ReLU (Rectified Linear Unit).

➢ **Training Process:**

- Loss Function: Our choice of loss function was the Negative Log Likelihood (NLL) loss function. It served to calculate the negative log likelihood of the actual target labels, given the predicted log probabilities. This was employed alongside the LogSoftmax activation function in the output layer, facilitating the computation of the negative log likelihood for the true class labels based on the predicted probabilities.

- Optimization Algorithm: The optimization algorithm utilized in our approach was Adam. We set the learning rate at 0.0007 for efficient optimization.
- Dropout: To mitigate the risk of overfitting, we introduced dropout with a probability of 0.3. Dropout played a crucial role by randomly deactivating a fraction of the input units during each forward pass.
- Batch Size: Our training process was carried out with a batch size of 512.
- Number of Epochs: The model underwent training for a total of 2000 epochs.

Deployment on Arduino:

- Model Conversion: After training and testing the model in a Python environment, we converted it into a TensorFlow Lite model using the TensorFlow Converter.
- Hex Dump Creation: We further prepared the model for Arduino deployment by generating a hex dump of the .tflite model using the "xxd" command in Linux. This step is crucial for embedding the model within the Arduino program.
- Header File Integration: The resulting .CPP file, containing the hex dump, was included as a header file in the Arduino base station program.

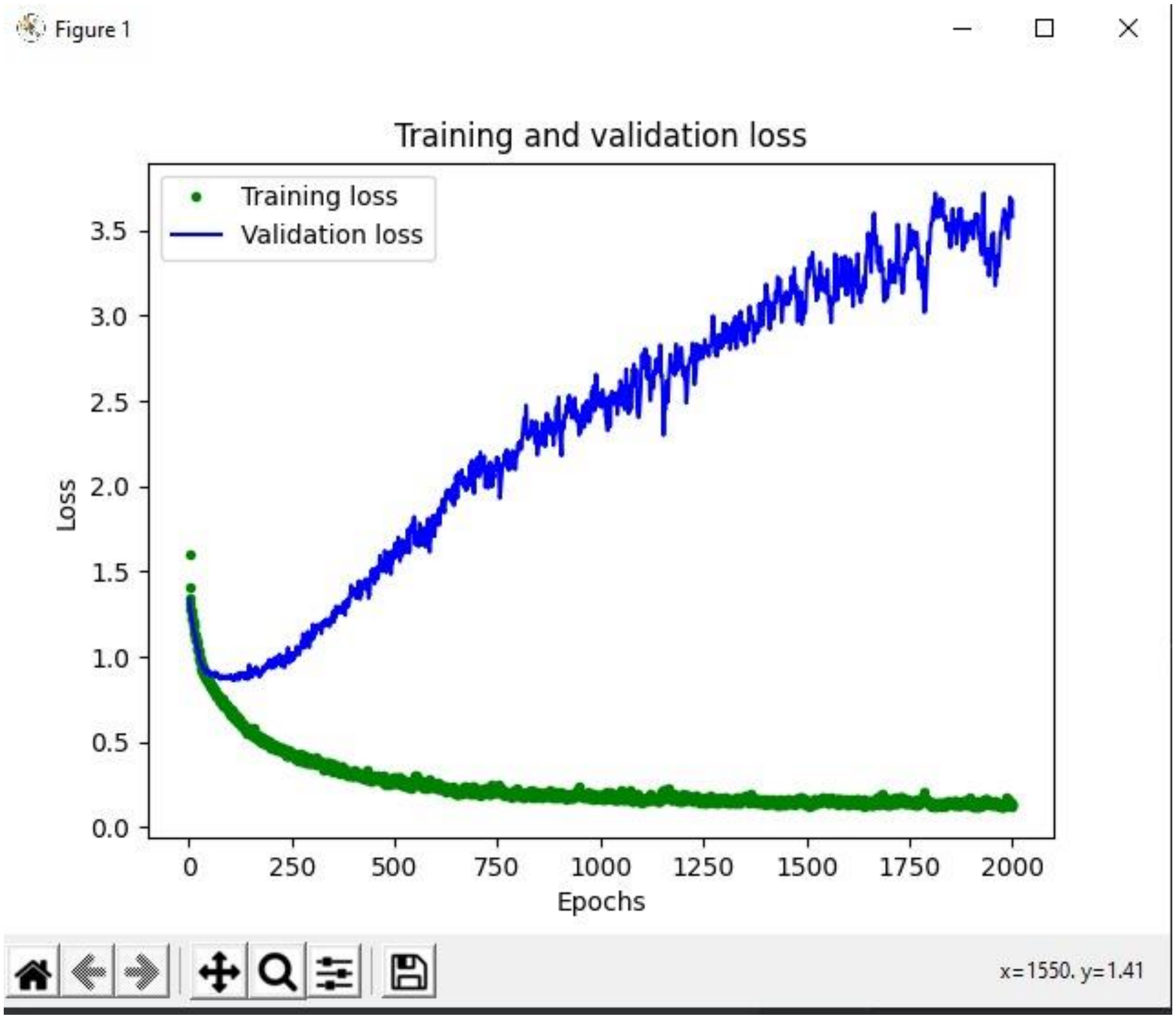
Base Station Implementation:

- Library Usage: For implementing the base station, we utilized the TensorFlow Lite and Arduino_LSM9DS1 libraries. These libraries provide essential functionalities for integrating the model and working with sensors.
- Memory Allocation: We allocated 20,480 KB of memory for our Tensor Arena, a crucial step for ensuring that there is sufficient memory to load and run the TensorFlow Lite model.
- Model Initialization: An instance of our TensorFlow Lite model was created using the "tflite::MicroInterpreter" class. This is where the model is loaded into memory, and we set the stage for inference.
- Sensor Switching: To switch between the different sensors, we used the Serial.read() function. We defined specific symbols for sensor selection: 'a' for accelerometer, 'g' for gyroscope, and 'm' for magnetometer. This approach allows the base station to communicate with and gather data from the selected sensor.
- Task Coordination: We employed a switch-case construct and an IMU_state variable, which can take on values 0, 1, 2, or 999. This variable plays a crucial role in coordinating actions between the base station, the selected sensor, and any connected laptop or device. It ensures that the base station carries out the appropriate tasks based on the sensor chosen.
- Idle State: As specified in our project description, the base station remains idle until a specific sensor is selected using Serial.read(), with an initial IMU_state value of 999. This design ensures that the base station awaits user input for sensor selection before taking action.

Results:

- We achieved a successful implementation of our model by utilizing Python in conjunction with the Tensorflow library. To ensure the thorough evaluation and validation of our system, we adhered to a conventional data partitioning strategy, allocating 50% for training, 20% for validation, and another 30% for testing. This deliberate division of the dataset into three separate subsets enabled us to conduct a rigorous evaluation of our posture detection model's performance and make well-informed judgments regarding its effectiveness.

- The graphical and representation of the training and validation results are given below:



Test results:

1. ReLu Test Accuracy: 72.6%.

Discussion:

- The project attained a moderate degree of success, marked by our ability to reasonably detect various postures, and effectively tackle the development challenges we encountered.
- The most formidable obstacle we faced during the project revolved around designing and fine-tuning the machine learning models. The task of determining the appropriate model architecture and hyperparameters necessitated a rigorous and iterative experimental approach. In this phase of the project, we relied on our intuition to select model parameters, though we recognize that adopting a more systematic and data-driven method could yield superior results in the future.
- Another notable challenge was associated with the data collection process. During data gathering, our sensor was connected via USB and executed smooth rotations at a slow pace, lacking abrupt changes in movement. To conduct more authentic and representative experiments, it is essential that we transition to a portable setup, enabling the sensor to be worn by an actual individual who can mimic a variety of postures. This adjustment will ensure that the collected data is more diverse and closely resembles real-world scenarios.
- In summary, while we achieved a reasonable level of success in our posture detection project, there is ample room for improvement in our model design, data collection methodology, and experimental setup. By addressing these challenges and refining our approach, we aim to enhance the accuracy and practicality of our posture detection system in future iterations.
