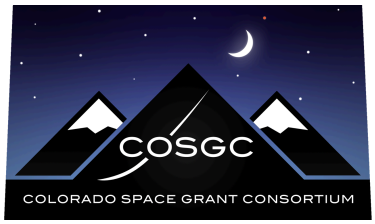




# Benchmarking Machine Learning Algorithms for Autonomous Rover Navigation Efficiency on a Resource-Constrained Platform

Authored By

**Jasper Bayless, Madeline Kovanda, Keven Martinez, Zoey Woodworth**

*Front Range Community College*

*2190 Miller Dr, Longmont, CO 80501*

Primary Contact

jasperbayless@gmail.com

# Abstract

This research explores the performance, computational efficiency, and energy consumption of three machine learning models – TensorFlow Lite (TFLite), PyTorch, and Q-Learning – when deployed on a Raspberry Pi-controlled rover navigating a simple indoor obstacle course. The study aims to determine the optimal machine learning approach for real-time decision-making under hardware and power constraints. Performance is measured based on average course completion time, onboard computational usage, and energy consumption. The findings will inform the selection of machine learning models for embedded robotics applications where efficiency and autonomy are critical.

---

## Brief Understanding of Machine Learning Frameworks

Some of the most popular deep learning frameworks include TensorFlow and Pytorch (Nguyen et al., 2019). These are the two deep learning frameworks that this research is testing and comparing but this research is also, uniquely, comparing deep learning to a tabular reinforcement learning framework, Q-Learning. These frameworks were chosen for their ease of implementation, huge popularity, and Python integrations and libraries (Alavizadeh & Jang-Jaccard, 2022; Nguyen et al., 2019).

### TensorFlow

TensorFlow is a software library created by Google Brain Team. It is designed for large scale training and inference. It can be used on small single CPU systems like the quad-core CPU used in this experimentation, as well as more powerful GPUs for larger scale systems. TensorFlow is by far the most popular deep learning tool, and it is fast evolving thanks to its support from the vast resources of Google (Nguyen et al., 2019). In this project study TensorFlow is used by manually driving the rover over and around standardized obstacles while recording the sensory data and motor control the whole time. Then a neural network is created which finds patterns between sensory data and motor control to later make real time navigation decisions.

### Pytorch

Pytorch is a Python library for deep learning. It is developed by Facebook's AI research group. Pytorch uses CPU and GPU tensor and neural network backends as independent libraries based on a C99 API (Alavizadeh & Jang-Jaccard, 2022). Pytorch is implemented the same as TensorFlow was by taking the same training data and building a neural network using the data to make decisions on what to do.

### Q-Learning

Unlike TensorFlow and Pytorch, Q-Learning is not a deep learning framework. Instead it is a reinforcement learning framework. Reinforcement learning frameworks work by having a set of states, this could be the sensor data and having a set of possible actions which would be the motor values and having a value for punishment or rewards. The way that the reinforcement learning works is that it makes decisions based on the robot's current state and is rewarded or punished based on the decision it makes, allowing you to build a table of weights based on the states, actions, and rewards provided in training (Habib 2019; Nguyen et al., 2019).

---

# Introduction

Autonomous robotic navigation is an essential element of sending robots to other planets and/or challenging environments for exploration and research. A robot's ability to consistently and effectively navigate an unpredictable and challenging environment can be mission critical. Going to an unknown planet like Mars requires needing to respond to unexpected obstacles and terrain. The ability to navigate through them with no issues is a challenge that has been tackled using different methods previously. Recent methods include using deep learning algorithms to create autonomous robotic navigation. Deep learning modules have grown increasingly popular recently with programs like Shogun, Keras, Torch (Nguyen et al., 2019). Autonomous robotic navigation relies heavily on machine learning models to make real-time decisions. However, deploying these models on resource-limited hardware, both computationally and electrically, presents challenges in computational load and power efficiency. This study takes into account how TensorFlow Lite, PyTorch, and Q-Learning perform under such constraints,

assessing their suitability for embedded systems like our demonstration rover. This research investigates the on-device electrical and computational demands of various machine learning models deployed on a robotic rover. This research explicitly focuses on the inference (decision making) phase, acknowledging that the training processes for these models may exhibit significantly different resource requirements. While the computational and energy costs associated with model training are recognized, they fall outside the scope of this study. The primary objective of this study is to characterize the resource utilization of deployed models within the constraints of a rover's limited onboard capabilities, thereby informing the design of energy-efficient extraterrestrial robotic systems. Our team could not find any research that compares a reinforcement learning model like Q-Learning to a deep learning model like TensorFlow or PyTorch. While there is extensive literature comparing deep learning methods against one another, there are noticeable differences in direct comparisons between reinforcement learning and deep learning models in the context of embedded autonomous navigation. This study aims to help address that difference and contribute further to research in this area of machine learning models. Our team expects TensorFlow to work the best all around for the purpose of traversing unknown land autonomously using deep learning frameworks due to its ease of use, mass adoption, consistent updates, and optimized lite version for edge computing (Spal & Kaur, 2018).

# Background

## Hardware Setup

To ensure consistent variables throughout the process this study consistently utilized the same hardware for each run. This ensures that no machine learning model has an edge over any of the others and all are tested equally so that our results adequately reflect the differences in the machine learning models.

### Processor

The processor chosen for each test is a Raspberry Pi 3B, 1GB RAM with a 4 core CPU. This processor was chosen for both its price and power consumption, since it is powerful enough to handle everything required for the testing for a relatively low price. The Raspberry Pi 3B is a very popular type of Single

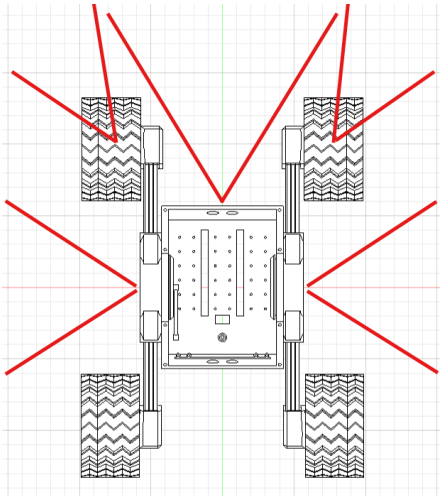
Board Computer (SBC) which are mini computers that consist of pc components such as cpu, memory, graphic components, and usb ports (Khan et al., 2022).

## Drive/Mobility

Each of the rover's independently driven four wheels has a 12V DC motor that is controlled by a L298N H-bridge motor driver. These motors were selected for their robust internal gearbox and impressive torque at a small size. Due to the current capabilities and fine control, H-bridge motor drivers were selected which can control up to 4 DC motors with directional speed and Pulse-Width-Modulation (PWM) control (Peerzada et al., 2021).While the motors output a good deal of torque, testing determined it to be insufficient for our design. In order to achieve the amount of torque necessary, our team designed and manufactured our own planetary hub for the wheels. We chose planetary gear reducers because they have significant advantages including; large power mass ratio, and strong bearing capability (Wang et al., 2022). The planetary gear hub greatly decreases the speed of the rover but increases torque to the wheels. Using high torque wheels instead of fast wheels affords the rover the torque necessary to successfully clear obstacles.

## Sensor Array

Three types of sensor were used; four HC-SR04 ultrasonic sensors, and one Arducam time-of-flight (ToF) camera for vision-based navigation. Additionally, Inertial Measurement Units (IMUs) were used for position sensing. The Arducam sensor is at the front of the rover to detect any obstacles that would be in front of it and the four ultrasonic sensors are split, two at the sides of the robot and two at the front wheels pointed outward shown in Figure 1.



(Figure 1)

An Arducam ToF camera was chosen because of its easy integration with the Raspberry Pi 3. ToF cameras work similar to lidar cameras. The camera emits a light at a specific frequency in the infrared spectrum that is polarized so that the sun's light doesn't interfere. Then the camera calculates the amount of time it takes that light to return to each pixel. This allows it to accurately measure the distance of its surroundings, in an effective and cost efficient way (Oner, 2014). Similarly the HC-SR04 ultrasonic sensor emits a sound wave of a specific frequency that bounces off of something in front of it and returns to the sensor. The sensor in turn can measure the time it took for the sound wave to come back, to compute a distance (Morgan, 2014). These sensors are useful because they can be used to see how far an obstacle is from the robot, which then can be used to avoid obstacles. Lastly, two MPU-6050 IMU sensors are used. This sensor is the combination of an accelerometer and gyroscope which can be used to measure how fast the rover is going in a x, y, and z axis, and how fast it's turning in a x, y, and z axis as well (Fedorov et al., 2015). For the sake of this research, it was determined that only the heading value would be needed as it would inform the rover how to return to forward.

## Data Recording and Processing

For the data gathering this demonstration rover has four HC-SR04 ultrasonic sensors connected to a microcontroller to collect the sensor data and turn it into distance data. The microcontroller selected is an Arduino Nano ESP32. The Arduino Nano ESP32 was chosen for its ability to collect data from the

environment where it is installed and to communicate the data to other devices such as the Raspberry Pi over serial communication (Oner, 2014). This provides the ability to collect the data from the HC-SR04 sensors and the MPU-6050 sensors and communicate them to the Raspberry Pi 3 throughout the process of data gathering and testing. Throughout testing the hardware was unchanged for consistency.

---

# Software Setup

The Raspberry Pi has a few Python scripts that take the data from the sensors connected to the Arduino. There is a Python script that allows the Raspberry Pi to connect to a gaming console controller. This is used when gathering data for training as direct inputs were sent to the motors to drive the rover through an obstacle course in similar fashion to a remote controlled toy. The various Python scripts publish data to a REmote DIctionary Server, or a REDIS[1] system, which is used to store various sensor and motor control data in RAM (Spal & Kaur 2018). A main program runs multiple scripts concurrently, simplifying debugging and maintaining consistency across platforms. It allows different inference models to be tested without altering the architecture of data flow, motor control, or logging, isolating the computational demands of the inference models.

## Scripts Used

To simplify debugging, isolate the inference scripts, and manage a clean flow of data, a different script was written for each of the rover's various subsystems.

- The Arduino has a simple script that receives the data from the sonar sensors and IMUs.
- A script was made that allows the Pi to interface with the Arduino and read the data it provides.
- A script for the data gathering and training script was written that takes input from the gaming console controller to drive the rover around and record data.
- A testing script was written to test and debug the motor controls.
- The Arducam ToF camera has its own script to easily ensure it was recording data.

[1] REDIS primarily functions as an extremely fast, in-memory key-value store. It is used for session management, real-time analytics, and even as a primary database for some use cases. Within the context of this study, REDIS is implemented to store data and share it between python scripts that can be called on by a different script for data management between scripts.

- A script was made to store all of the sensor and motor values into a Comma Separated Values (CSV) file for analysis and later training.

The script for data collection was structured so that if the Raspberry Pi was in recording mode, the data from all of the sensors, motors, camera, and timestamp would be saved to a CSV file. Google Colab[2] was used for both TensorFlow and PyTorch AI models. The camera data script for PyTorch and for TensorFlow is the same. First, the camera depth is downsampled from a 180x240 pixel image to 9x12 to simplify the data received. To process the camera data, first the script took the variable of maximum distance[3] and divided the depth values by it to use a zero to one range (normalization). The script looks at the raw depth data and the confidence score[4], which is how certain the depth measurement is. If the confidence score is less than the threshold of 30 percent, then the value is changed to zero (ArduCAM). The model relies heavily on the camera, so accurate camera values were required. The script uses a frame buffer to store the last three depth frames, which are averaged together to produce a smoother and more reliable input for the model.

## Machine Learning Models

### PyTorch

For PyTorch, a section to upload the CSV files and put the data into the labels (motors, heading, distance sensors, Arducam values, etc.) was created. The data was split into training data, and testing data. A rolling sequence was created for the data, which was important because this allowed the last ten rows of data to inform the model's next prediction, essentially a 'short-term-memory". This proved to be extremely useful as the model had better context to recognize the reason for turning or backing up. The neural network was separated into two branches for the camera data and the sensor data. These two branches were used because the camera data was collected from pixels and as such used convolutional

[2] Google Colab is a free, cloud-based platform that allows users to write and execute Python code in a Jupyter Notebook environment. It provides access to powerful computing resources, including GPUs and TPUs, making it well-suited for machine learning, data analysis, and collaborative research without requiring local setup.

[3] Defined as 850mm for this study. This threshold was set so that the robot wouldn't be influenced by any obstacles outside of its immediate vicinity such as a person walking in the distance.

[4] The confidence score is a per-pixel metric provided by the depth camera indicating the reliability of a given depth measurement. Scores range from 0 (unreliable) to 255 (highly reliable); in this project, we discard pixels with scores below 30 to mitigate noise.

```
depth = raw_depth.astype(np.float32)
confidence = raw_confidence.astype(np.uint8)
depth[confidence < 30] = 0  # Remove unreliable pixels
```

layers, whereas the sensor data was numerical and were trained with dense layers ("Build the Neural Network," 2025). Dense layers work well for numerical structured data, and convolutional layers work well to filter and find patterns with pixels ("Torch.Nn, 2024). A pool layer was used to lessen the computational load ("Build the Neural Network," 2025). Finally, the model was trained using the data collected in training by a loss function and an optimizer. This is where the model is tested on the test portion of the data and if it predicts the correct action the accuracy increases. In order to run the model on the Raspberry Pi, an inference Python script was developed that loaded the AI model. Within the inference script, the data from the cameras and the sensors is normalized before being processed by the inference model. This script also kept the last 10 rows of data in the history for the model to use. In testing it was noted that at times the model was wrong when predicting motor actions, so a process was implemented so that in order to make a decision (for the motor commands), the inference model must predict the same move three times before the command will go through to the motors. This made a noticeable difference as the model improved test course completions.

### TensorFlow

For TensorFlow, Google Colab was used as well. The CSVs uploaded would be combined in one dataframe using the Pandas library for processing and then labeled before making a numpy array ("Load a Pandas Dataframe," 2024). Just like with PyTorch, the last 10 rows of data were used for context. A sequence dataset was created which was then labelled with the motor action. Convoluted Neural Networks[5] (CNN) were used to reshape the data from the Arducam and then created a 3D tensor by stacking 10 frames. The CNN is then used to find patterns in the data and the motor controls. TensorFlow and PyTorch use test and training sets in the scripts, which is when a percentage of the data uploaded (from the CSV data) is used for training the AI model, and the rest of the data is used to test the AI model on what it learned. For this model, 80 percent of the data was used for training and the remaining 20

[5] A convolutional neural network (CNN) is a type of deep learning model particularly effective at processing grid-like data such as images. Instead of analyzing every pixel individually, CNNs use small filters (kernels) that slide over the input image to detect patterns like edges, textures, or shapes. These filters are learned during training and allow the model to focus on important visual features without needing to manually define them. In the context of our rover, the CNN helps extract useful obstacle patterns from the depth camera frames so that the model can learn how to respond to different terrain layouts efficiently.

percent for testing. Finally, the neural network was built, which was done with the Sequential model from TensorFlow (“Tf.Leras.Sequential,” 2024). This model has three hidden layers which help to recognize difficult patterns, followed by one output layer for the motor actions. The hidden layers consisted of dense layers to connect to the input layer, and a Rectified Linear Unit. The Rectified Linear Unit, or ReLU, is used to filter out negative values, so that only the positive values are output (“Tf.Keras.Activations.Relu,” 2024). The Colab also includes pruning, which is defined as the process of removing unnecessary weights in the script in order to optimize processing. This was achieved by using the TensorFlow Model Optimization Toolkit to bring half of the weights to zero by slowly increasing over 1000 steps at the beginning (“TensorFlow Model Optimization,” 2024). The model was trained with fifty epochs and then the AI model was converted to TensorFlow Lite. TensorFlow Lite was chosen over the full TensorFlow library due to the Raspberry Pi 3’s processor limitations. The inference script for the Raspberry Pi was similar to the PyTorch inference script. The model gets loaded into the script and the last 10 rows of data are kept for the model to use as context. The inference script also has the action stickiness, which is where the model must predict the same action two times in a row in order to successfully send a command to the motors.

### Q-Learning

The camera data script for Q-Learning was different from the script used for the Deep Learning frameworks. The difference is that the script for the deep learning models uses a rolling average and has a frame buffer for the last three frames. The Q-Learning script for the camera data has no average and no frame buffering, so each frame is processed immediately. Both scripts use a downsampling function to bring the 240x180 pixel image down to a manageable 12x9 depth map, then normalize those values (0-1). In order to use Q-Learning, the model must train by using a reward function and a Q table. A  PS4 controller was used for training Q-Learning, to return positive (reward) or negative (punishment) feedback and store that to a Q table. The Q table would at first randomly select moves (forward, left, right, and backwards) and after each move, the program would pause until it received feedback from the controller. The pi stores the state and the action it took, as well as the feedback received for the action it

took to a Q table. Over time, the Q table informs the moves that the rover makes and it is provided more and more positive feedback to the reinforcement learning program. Once it was rewarded and punished the program through multiple obstacle courses, it was ready to be used. The Q-Learning inference script had multiple thresholds for the camera, sensors, and heading data. The thresholds were for if the camera/sensors were near an obstacle or medium distance from an obstacle. The heading was separated into 8 bins in order to discretize[6] the data. Similar to the PyTorch and TensorFlow, the same action stickiness was made so that the model must predict the same action three times in a row before the rover would execute that action. The inference script for Q-Learning had to be discretized for the camera as well, so the camera was broken up into the left side, the middle, and the right side, and if the rover was near an obstacle, the value was 0, if it was medium distance, 1, and if it was far away the value was 2. Q-Learning needs a state, which is the situation it is in. The data consisted of the values from the camera, heading, and sensors converted to a tuple of numbers to store it in the Q table.

---

# Experimental Procedure

## Evaluation Metrics

The efficiency and efficacy of the different models are evaluated using three different key metrics. Real-world performance, onboard compute usage, and energy consumption.

- **Real-World Performance**: Measured by the average time each model takes to complete a standardized obstacle course. The rover is timed from start to finish across multiple runs using each learning framework.
- **Onboard Compute Usage.** CPU and RAM utilization are logged throughout each run using Python scripts on the Raspberry Pi, providing insight into computational demands during inference.

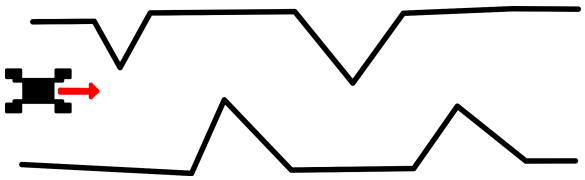
[6] The process of taking linear data and processing it into a binary dataset.

- **Energy Consumption.** Voltage and current sensors, connected via a secondary Arduino Nano, measure the Raspberry Pi's power draw in real time, revealing the electrical cost of running each model.

---

## Procedure

Each model was trained by hand using a gaming console controller wired to the Raspberry Pi via USB on simulated courses. An obstacle course was built to replicate what could be encountered in unknown locations. The obstacle courses had two walls on each side with more obstacles throughout the course to ensure that both the left and right side of the rover were trained fairly and with enough data. For the Pytorch and TensorFlow models, the training is identical. A gaming console controller connected to the Raspberry Pi was used to drive around the obstacle course and avoid obstacles. The script also collects all of the sensor data, and motor values and puts them all into a CSV file. This data is the training data that the team collects to use to train the TensorFlow and PyTorch deep learning. The data values gathered were used on both TensorFlow and PyTorch to ensure that they were both using the same training data. This data would then be loaded into their TensorFlow or PyTorch script respectively. The data is processed to make pattern connections from what the sensors see and what the motors are told to do. A neural network is then built that drives autonomously and avoids obstacles from the same or different courses that it was trained in. The training for Tensorflow and Pytorch is the same since they are both deep learning frameworks, however, Q-learning is not, so a different training method was used. To train Q-learning as similarly as possible to TensorFlow and Pytorch, the PS4 controller was used to reward or punish the rover through each decision it makes as it traverses the obstacle course letting it move a little to make a decision of either moving forward, back, turning left, or right and punishing it if it runs into an obstacle and rewarding it for avoiding an obstacle.



(Figure 2)

Figure 2 shows a scale diagram of the research training course. Note that there are two obstacles on the left, and two on the right. This ensures statistical reliability of research data by ensuring an equal amount of training to go left and to go right. The course is exactly 5 meters long, and this is the distance that the time-to-complete metrics was recorded on.

Both deep learning and reinforcement learning are widely applied in fields such as medicine and robotics, and their capabilities are rapidly expanding (Mahmud et al., 2018). Given their growing relevance, it is important to compare them as directly as possible. Deep learning excels with large datasets and complex pattern recognition, but may suffer from misclassification, particularly when overfitting occurs. Reinforcement learning, while effective across diverse tasks, struggles with large volumes of data (Mahmud et al., 2018). Because these approaches complement one another, many modern systems integrate them into deep reinforcement learning frameworks (Huang, 2020; Mahmud et al., 2018). In this study, the rover was deliberately trained and tested for each approach separately to compare their individual performance in traversing unknown terrain under hardware and power constraints.

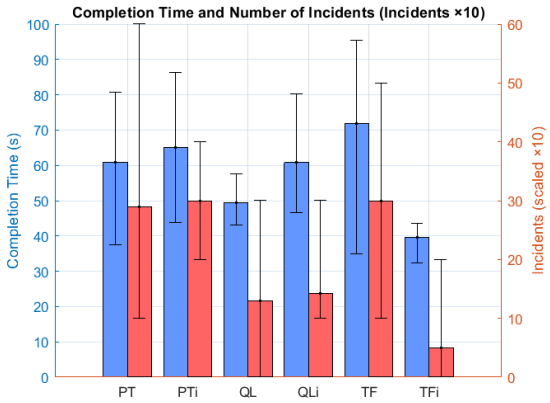
Each model was trained using identical input data where possible, and evaluated based on three metrics: real-world performance, onboard compute usage, and energy consumption. The rover was run through a standardized 5-meter obstacle course ten times per model. For each run, both course completion time and number of collisions were recorded by hand. To assess efficiency, onboard CPU and memory usage were recorded using Pi-based logging, and monitored power draw with external voltage and current sensors. Since processor load directly affects energy usage on the Raspberry Pi 3 (Bekaroo & Santokhee, n.d.), comparing these values allowed us to evaluate each model's tradeoffs between performance and

efficiency. A model that completed the course quickly and reliably while minimizing processor and energy demand was considered optimal.

---

# Results

The data that was gathered throughout the experiment was graphed using MatLab. This experiment was run twice for each learning model. The first run (TF and PT) of each deep learning model are done with a large outdoor dataset [40.3MB] recorded on various types of courses. TFi and PTi were trained on a smaller dataset [2.97MB] and specifically trained on an indoors course similar to the training course used for Q-Learning. This allows for the best direct comparison of PyTorch, TensorFlow, and Q-Learning frameworks, while also exploring the performance of full sized TensorFlow and PyTorch frameworks.




(Figure 3)

Figure 3 shows the average completion time for each of the learning models and the average incidents per learning model over the 5 meter research course.

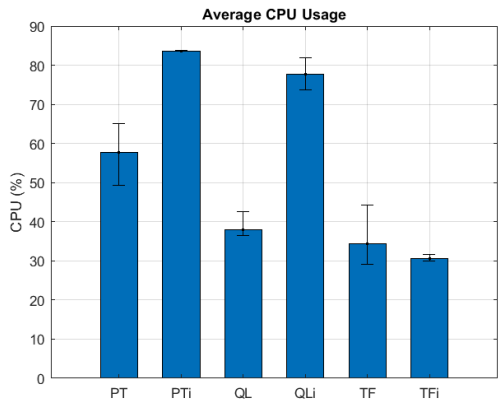
The experimental results in Figure 3 and the associated data provide insight into the relative performance of six machine learning approaches tested on a 5-meter indoor research course. Among all models, TFi (TensorFlow trained on a small, indoor-specific dataset) demonstrated the strongest real-world performance, achieving the fastest average completion time of 39.63 seconds and the lowest average number of incidents at 0.5. In contrast, the full-scale TensorFlow model (TF), trained on a larger and more

diverse outdoor dataset, performed the worst—averaging 71.9 seconds to complete the course with 3 incidents, suggesting that environmental mismatch and overfitting negatively impacted generalization[7]. The PyTorch models, PT and PTi, showed similar performance despite differing datasets. PT completed the course in 60.88 seconds with 2.9 incidents, while PTi, trained on indoor-specific data, finished in 65.10 seconds with 3 incidents. This parity implies that the PyTorch architecture may generalize more robustly than TensorFlow, or that the smaller dataset did not add significant value due to limited variation. Notably, both PyTorch models underperformed compared to TFi, reinforcing the advantage of environment-aligned training in constrained navigation tasks[8].

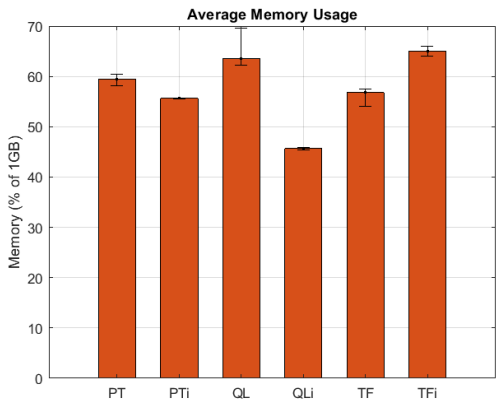
Q-Learning models (QL and QLi) performed consistently well, with QL averaging 49.39 seconds and 1.3 incidents, and QLi at 60.82 seconds and 1.43 incidents. Importantly, there is no actual difference in training data or architecture between QL and QLi; QLi was a repeated run introduced to validate an anomalously high power reading observed during the original QL test. Therefore, QL and QLi should be treated as equivalent in evaluating real-world performance and resource usage. The strong results from Q-Learning—despite being simpler in structure—highlight the effectiveness of training entirely within the target environment, ensuring high alignment between learned behaviors and deployment conditions.

[7] In our rover project, fit describes how well the model learns from training data, while generalization is its ability to perform in new environments. Overfitting happens when the model memorizes the training data too closely, reducing its adaptability and causing poor performance on unfamiliar terrain.

[8] Environment-aligned training ensures the model learns patterns relevant to the actual deployment setting. In constrained navigation tasks like our rover's, aligning training data with real-world terrain and obstacles improves decision-making and reduces failure when navigating unfamiliar paths.

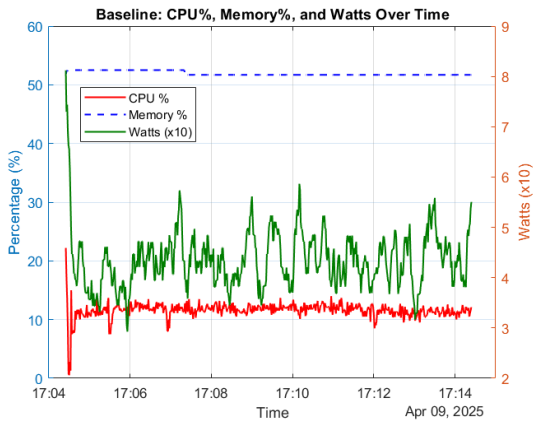



(Figure 4)



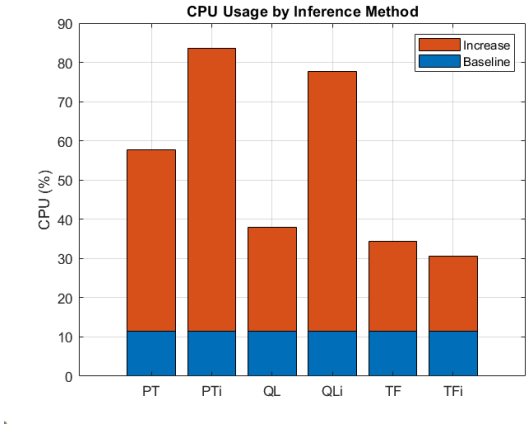

(Figure 5)

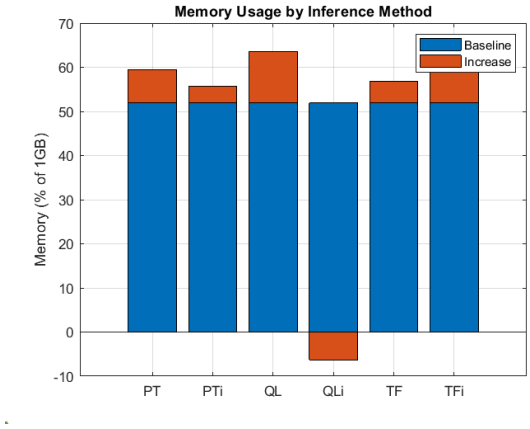
Figures 4 and 5 present the average CPU and memory usage for each model during inference on the Raspberry Pi 3. These metrics are critical for evaluating real-time performance and resource efficiency in embedded systems. The most resource-intensive model overall was PTi, which consumed an average of 83.66% CPU, followed by QLi at 77.66%. In contrast, the lowest CPU usage was observed in TFi at 30.54%, with TF and QL also performing relatively efficiently at 34.43% and 37.97% respectively. While QL and QLi were functionally identical, QLi exhibited significantly higher CPU usage, likely due to background system variability or differences in Redis/motor communication timing during that run. In terms of memory consumption (Figure 5), the most demanding model was TFi, averaging 65.05%, followed closely by QL at 63.57%. Conversely, QLi used the least memory, at only 45.64%, showing a stark contrast despite the models being equivalent. The full TensorFlow model (TF) used 56.76% memory, and the PyTorch models followed with PT at 59.48% and PTi at 55.62%. These results indicate that while curated models like TFi and QL performed best behaviorally, they did not always minimize computational demand. Therefore, TFi is the most efficient deep learning model in terms of CPU, but QL offers the best balance of performance and CPU efficiency, albeit with slightly higher memory usage. The variation observed between QL and QLi underscores the importance of repeated measurements when analyzing real-time system performance on embedded platforms.




(Figure 6)

Figure 6 shows the baseline CPU, Memory, and Power usage of the Raspberry Pi 3 as it was left untouched for 10 minutes to have a benchmark to compare the various inference models to, isolating the overhead that each model introduces.



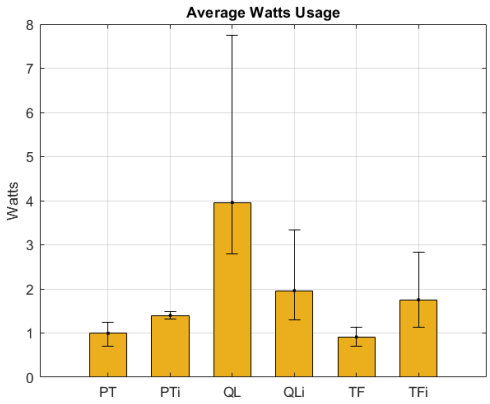




(Figure 7)

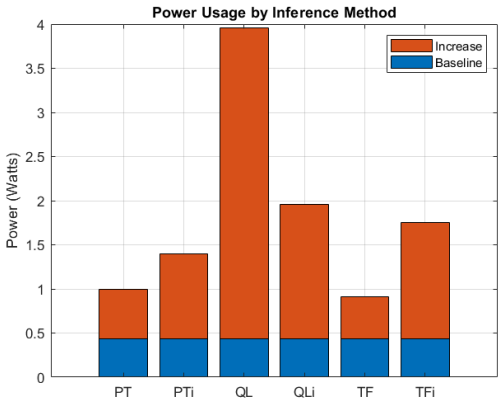
(Figure 8)

Figures 7 and 8 present CPU and memory usage as a percentage increase over the Raspberry Pi 3's baseline utilization, offering a clearer picture of how much load each inference method imposes relative to system idle. The baseline CPU usage during idle was approximately 11.56%, and baseline memory usage was around 51.93%. In terms of CPU overhead (Figure 7), PTi introduced the highest load, increasing CPU usage by +72.11%, followed by QLi at +66.11% and PT at +46.12%. By contrast, the TensorFlow models were significantly lighter, with TF adding only +22.87% and TFi just +18.98%, making TFi the most CPU-efficient model overall. Despite being behaviorally similar, QL and QLi showed large

variance—QL added only +26.41%, while QLi nearly tripled that. This discrepancy is not due to model differences but likely system-level factors during execution, and are outside the scope of this study (e.g., Redis timing, subprocess handling, or thermal throttling). Memory usage (Figure 8) shows a similar trend. TFi incurred the highest memory overhead at +13.12%, followed by QL with +11.64%. PT and PTi added +7.55% and +3.69%, respectively. TF added +4.83%, remaining relatively modest in overhead. Notably, QLi registered a memory delta of -6.29%, which, while counterintuitive, likely reflects momentary memory release or operating system variance during measurement. Overall, TFi provided the best balance of low CPU and memory impact, while PTi and QLi demanded the most computational resources. These delta-based metrics are crucial for real-world deployment on constrained hardware, revealing not just model performance but also system compatibility.




(Figure 9)

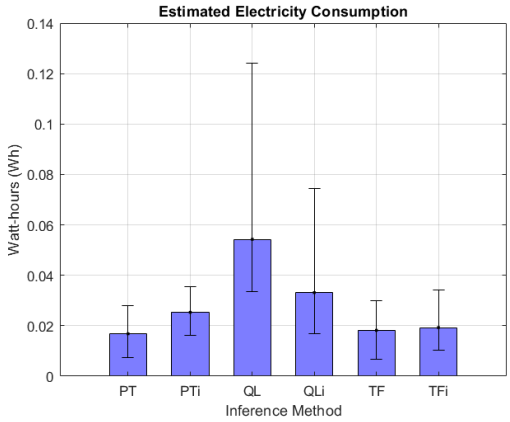



(Figure 10)

Figures 9 and 10 illustrate the average power usage of each inference method, both as raw wattage and as an increase above the Raspberry Pi 3's baseline draw of approximately 0.44 W. Power consumption provides an important dimension to model evaluation, particularly for mobile and embedded robotics applications where battery life and thermal constraints are critical. Among the deep learning models, PT and TF were the most power-efficient, with average draws of 0.99 W and 0.91 W, corresponding to increases of +0.56 W and +0.47 W, or roughly 128% and 109% above baseline, respectively. TFi and PTi, while efficient computationally, showed higher consumption, drawing 1.75 W and 1.40 W, respectively.

The most striking finding appears in the Q-Learning results. The initial QL run reported an average power consumption of 3.96 W, an increase of 3.52 W over baseline—an 808% rise, far exceeding any other method. This anomaly prompted a second trial, labeled QLi, which yielded a more reasonable average of 1.96 W, just +1.53 W above baseline (+350%). Given that both QL and QLi used identical models and training data, the drastic discrepancy likely stems from faulty sensor readings or interference during the first run. Our team suspects that either the ACS712 current sensor or the voltage monitoring system may have returned erroneous values—possibly due to loose wiring, inconsistent load calibration, or transient noise affecting analog-to-digital readings on the Arduino or Pi during data logging.

Another possible factor is mechanical resistance or excessive actuation caused by unaccounted motor friction or feedback control instability during the QL trial, potentially resulting in higher current draw. However, given that CPU and memory loads were relatively low during that run (+26.41% CPU and +11.64% memory over baseline), it is unlikely that the processor alone accounts for the observed spike. The subsequent QLi trial confirmed the expected range of values, reinforcing that the original measurement was an outlier. As such, QLi should be treated as the authoritative measure for Q-Learning power use, with the original QL run serving as a cautionary example of the importance of validating sensor-driven telemetry in performance benchmarking.




(Figure 11)

Figure 11 shows the average estimated amount of energy consumption per learning model course run.

When considering total energy consumed—a critical factor for battery-powered robotics—watt-hours (Wh) offer a more holistic view by combining both power draw and run duration. A model that consumes more watts may still use less total energy if it completes its task significantly faster.

The results show that TFi, despite a moderate power draw (~1.75 W), had one of the lowest energy consumptions at 0.0193 Wh, thanks to its fast average completion time of 39.6 seconds. Similarly, TF and PT also maintained low energy use at 0.0182 Wh and 0.0168 Wh, respectively. These models demonstrate that efficient execution combined with fast task completion yields favorable energy profiles.

In contrast, Q-Learning (QL) consumed the most energy at 0.0543 Wh, driven primarily by the anomalously high power draw in the first run. The repeated trial (QLi) dropped significantly to 0.0331 Wh, highlighting the importance of verifying telemetry. Still, QLi remained the highest among the reliable measurements, reflecting a tradeoff between adaptive learning capability and system efficiency. These results reinforce that TFi offers the best balance of low energy use, fast runtime, and strong real-world performance, making it ideal for resource-constrained deployments.

---

# Analysis

Although Q-Learning demonstrated commendable consistency in task completion and incident reduction, its relatively high power consumption, particularly in the more reliable QLi run, presents a significant limitation for power-constrained applications. At 0.0331 Wh, QLi consumed over 70% more energy than TFi, despite completing the course slower. This emphasizes a tradeoff: Q-Learning offers robust, environment-aligned performance, but lacks the computational and electrical efficiency critical for long-term or battery-powered deployment, especially in more dynamic or data-intensive environments.

PyTorch, on the other hand, underperformed across nearly every dimension. Both PT and PTi showed slower completion times and more collisions compared to TFi and QL, and demanded higher CPU usage (+46.1% and +72.1% above baseline, respectively). Even though raw power draw was modest, this came at the cost of increased runtime and poor course navigation, which inflated their overall energy usage

relative to their utility. These results affirm that PyTorch, at least in its unoptimized form, introduces excessive overhead for low-power CPUs like the Raspberry Pi 3B. The high CPU strain with minimal behavioral gains suggests PyTorch's dependency on GPU acceleration to reach optimal performance, making it ill-suited for embedded edge devices without dedicated hardware support.

TensorFlow – particularly the indoor-trained TFi variant – emerged as the most favorable model overall. It completed the course fastest (39.6s), incurred the fewest incidents (0.5 on average), and maintained the lowest CPU overhead (+18.98%). Despite its moderately elevated memory usage and wattage (1.75 W), its shorter run duration resulted in one of the lowest energy footprints at just 0.0193 Wh. This positions TFi as the most efficient and performant model for embedded systems, assuming the training data matches the deployment environment. TF (trained on a broader dataset) struggled with generalization indoors, reinforcing the critical importance of environment-specific training for real-world effectiveness. These results also validate the decision to include both full-scale and downsampled models, as TFi consistently outperformed its larger counterpart despite being smaller in size and faster to execute.

Overall, the data suggests that model fit, defined here as the alignment between training conditions and deployment context, is more impactful than model complexity or theoretical performance. TensorFlow Lite, when properly trained, balances low computational burden with strong behavioral outcomes. Q-Learning, though less efficient, remains a viable alternative where training can occur in-situ and electricity is less constrained. PyTorch, by contrast, was limited by its execution profile on CPU-bound systems.

In conclusion, while all models achieved baseline navigation, only TensorFlow Lite consistently delivered high accuracy, speed, and efficiency, making it the most viable candidate for embedded rover autonomy under real-world constraints.

---

# Conclusion and Future work

The inference model that an engineer should choose for their platform varies widely based on the data types, quantity of data, and use-case environment.

While outside the scope of the research, note that when sufficient computational resources are available, such as high-performance GPUs, PyTorch has been shown to perform comparably to TensorFlow in both training and inference tasks. In a study comparing framework performance across multiple model types, PyTorch achieved similar or better efficiency in 66% of inference cases and demonstrated competitive training times, particularly in NLP models (Georgiou et al., 2022).

If a consistently well performing model is needed to control a robot's navigation in a simple environment with a level surface such as a warehouse or similar environment, Q-Learning has the potential to be the strongest performer assuming energy is not a constraint.

If an inference model is needed that will perform consistently well in a complex environment with lots of sensory input, and energy is a significant constraint, TensorFlow Lite is the best performing, with the least CPU and Watt-hour draw. It is important, however, that the model is properly fit for the environment that the robot is deployed to or you risk overfitting and poor generalization of the model which spikes CPU and Watt usage.

The research was conducted within a controlled, indoor standardized environment. As computational capabilities continue to advance, we are optimistic about the potential to extend Q-Learning methodologies to more complex, outdoor settings.

---

# Acknowledgements

We would like to extend our sincere gratitude to all those who lent a hand in this process –

**Mrs. Stephanie Beck**, our faculty advisor, for being there through the whole process to ensure that our team always had the support that we needed to see this process through. Your kindness and thoroughness brought us farther than any of us expected.

**Greg Staples** for mechanical engineering feedback through the process of developing our research demonstration rover. Additionally for technical feedback and revisions on our research paper.

**Jacob Bayless** for the hours of revisions to our research paper.

**Mr. Ryan LaCount** for the electrical engineering knowledge that led us towards development of a custom PCB for our demonstration rover that would ensure consistent power to our processors.

**TinkerMill** and all of its members for generously providing a space for our team to work on and test our demonstration rover. When we had nowhere else to go, the TinkerMill makerspace was a hub of creativity and knowledge within which we could iterate and test our robot.

---

# Use of AI Tools

This research project made use of OpenAI's ChatGPT (GPT-4 and related models) to support various stages of development. ChatGPT was used to:

- Debug and assist in writing Python scripts for sensor integration, library integration, and data logging on the Raspberry Pi
- Rephrase and refine sections of the research paper for clarity and tone.
- Provide feedback and revisions on figures, analysis, and model explanations.

All technical decisions, experimental design, and final edits were made by the authors.

---

# References

Alavizadeh, H., & Jang-Jaccard, J. (2022). *Deep Q-Learning Based Reinforcement Learning Approach for Network Intrusion Detection*. Computers, 11(3), 41. https://doi.org/10.3 390/computers11030041.

ArduCAM. (n.d.). *Arducam ToF Camera SDK – for Raspberry Pi*. Arducam Documentation. https://docs.arducam.com/Raspberry-Pi-Camera/Tof-camera/Arducam-ToF-Camera-SDK/.

Bekaroo, G, & Santokhee, A. (n.d.) *Power Consumption of the Raspberry Pi: A Comparative Analysis*. repository.mdx.ac.uk/download/3dadcffb962fcf5ee1b904a44bbd7a5db2717ac0

73464d7912a822ae28137410/332622/Power%20Consumption%20of%20the%20Raspberry%20Pi_CameraReady.pdf.

*Build the Neural Network*. (2025). PyTorch. https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html.

Fedorov, D.S., Ivoylov, A.Y., Zhmud, & Trumin, V.G. (2015). *Using of Measuring System MPU6050 for the Determination of the Angular Velocities and Linear Accelerations.* http://www.jurnal.nips.ru/sites/default/files/A&SE-1-2015-11_0.pdf.

Georgiou, S., Kechagia, M., Sharma, T., Sarro, F., & Zou, Y. (2022). *Green AI: Do deep learning frameworks have different costs?* In Proceedings of the 44th International Conference on Software Engineering (ICSE '22), 1184–1196. Association for Computing Machinery. https://doi.org/10.1145/3510003.3510221 .

Habib, N. (2019). *Hands-On Q-Learning with Python: Practical Q-Learning with OpenAI Gym, Keras, and TensorFlow*. Packt Publishing. https://books.google.com/books?hl=en&lr=&id=xxiUDwAAQBAJ&oi=fnd&pg=PP1&dq=Tensorflow+vs+q+learning&ots=9msZLsuIoA&sig=RCNT5mEOQxnz9j-eCGU7vk21Fb4#v=onepage&q=tensorflow%20vs%20q%20learning&f=false.

Huang, Y. (2020). *Deep Q-Networks.* In: Dong, H., Ding, Z., Zhang, S. (eds) Deep Reinforcement Learning. Springer, Singapore. https://doi.org/10.1007/978-981-15-4095-0_4.

Khan, B. M., Fahad, M., Bilal, R., & Khan, A. H. (2022). *Performance Analysis of Raspberry Pi 3 IP PBX Based on Asterisk.* Electronics, 11(20), 3313. https://doi.org/10.3390/electronics 11203313.

*Load a Pandas Dataframe : Tensorflow Core.* (2024). TensorFlow. www.tensorflow.org/tutorials/load_data/pandas_dataframe.

Mahmud, M., Kaiser, M.S., Hussain, A., & Vassanelli, S. (2018). *Applications of Deep Learning and Reinforcement Learning to Biological Data.* 1-4. IEEE Trans. Neural Netw. Learn. Syst. https://arxiv.org/pdf/1711.03985.

Morgan, E.J. (2014). *HC-SR04 Ultrasonic Sensor.* https://www.bitmi.ro/domains/bitmi.ro/files/files/bitmi-datasheet-senzor-ultrasonic-hc-sr04-2702.pdf.

Nguyen, G., Dlugolinsky, S., Bobak, M., Tran, V., Garcia, A. L., Heredia, I., Malik, P., & Hluchy, L. (2019). *Machine Learning and Deep Learning Frameworks and Libraries for Large-Scale Data Mining: A Survery.* Artif Intell, Rev 52, pages 77-80. https://doi.org/10.1007/s10462-018-09679-z.

Oner, V. O. (2014). *Developing IoT Projects with ESP32*. Packt Publishing. https://books.google.com/books?id=e4k4EAAAQBAJ.

Peerzada, P., Larik, W. H., Mahar, A. A. (2021). *DC Motor Speed Control Through Arduino and L298N Motor Driver Using PID Controller.* International Journal of Electrical Engineering & Amp; Emerging Technology, vol 4, no.2, 21-24. http://www.ijeeet.com/index.php/ijeeet/ article/view/94.

Spal, G.K., & Kaur, J., (2018). *In-Memory Data Processing Using REDIS Database.* International Journal of Computer Applications, 180(Issue), 28, https://www.ijcaonline.org/archives/volume180/number25/spal-2018-ijca-916589.pdf.

*TensorFlow Model Optimization*. (2024). TensorFlow. https://www.tensorflow.org/model_optimiz ation

*Tf.Keras.Activations.Relu: Tensorflow V2.16.1.* (2024). TensorFlow. www.tensorflow.org/api_doc/python/tf/keras/activations/relu.

*Tf.Keras.Sequential: Tensorflow V2.16.1*. (2024). TensorFlow. www.tensorflow.org/api_docs/python/tf/keras/Sequential.

*Torch.Nn - PyTorch 2.6 Documentation*. (2024). PyTorch. pytorch.org/docs/stable/nn.html#convolution-layers.

Wang, S., Xu, X., Chen, X., & Renzheng, L. (2022). *Lightweight Design of the Planetary Gear Reducer of Reduction Drive Hub Motors.* IOP. https://dx.doi.org/10.1088/1742-6596/2338 /1/012059.