Bridging the Sim-to-Real Gap: A Hierarchical Deep Reinforcement Learning Architecture for Autonomous Navigation on Embedded Edge Hardware

Mauro A. Lopez
Engineering School
Universidad de la Américas Puebla
Puebla, Mexico
mauro.lopezmz@udlap.mx

***Abstract*—Autonomous navigation in unstructured environments remains a significant challenge for mobile robotics. While Deep Reinforcement Learning (DRL) offers robust adaptability compared to classical heuristic methods, deploying these computation-heavy models on resource-constrained edge devices presents substantial hurdles. This project investigates the "Sim-to-Real" transfer of a Deep Q-Network (DQN) agent onto a low-cost ESP32 microcontroller. We developed an end-to-end pipeline, training a navigation agent in a custom OpenAI Gymnasium simulation and deploying it to physical hardware via TensorFlow Lite for Microcontrollers.**

Our experimental results highlight critical insights into the limitations of TinyML. We demonstrated that standard 8-bit integer quantization led to "model collapse," rendering the agent incapable of decision-making, whereas a full 32-bit floating-point model retained navigation fidelity. Furthermore, to address the stochastic nature of the transferred policy and sensor noise, we implemented a Hierarchical Control Architecture that fuses the DRL planner with a deterministic safety layer. The final system successfully demonstrated autonomous obstacle avoidance and navigation in real-world tests, validating the feasibility of complex AI on microcontrollers.

***Index Terms*—Deep Reinforcement Learning, Sim-to-Real Transfer, TinyML, ESP32, Hierarchical Control, Autonomous Navigation.**

I. Introduction

A. Motivation

The field of autonomous mobile robotics is rapidly shifting from rigid, rule-based programming to data-driven learning approaches. Classical algorithms, such as Artificial Potential Fields (APF) or PID wall-following, rely on manually tuned parameters and often fail in complex or dynamic environments (e.g., getting stuck in local minima). Deep Reinforcement Learning (DRL) addresses this by enabling agents to learn optimal navigation policies through trial and error, theoretically allowing for more robust and generalized behavior.

However, a significant gap remains between the theoretical capabilities of DRL and its practical deployment. Most DRL research relies on powerful GPUs and high-fidelity sensors (LiDAR, Cameras), which are unsuitable for low-cost, battery-powered IoT devices. The emerging field of TinyML aims to bridge this gap by running machine learning inference directly on microcontrollers.

B. Problem Statement

Deploying DRL on edge hardware like the ESP32 introduces two primary challenges:

1) Hardware Constraints: Microcontrollers have limited RAM (typically <500KB) and compute power, requiring model compression techniques like quantization that can degrade performance.
2) The Sim-to-Real Gap: A policy trained in a perfect digital simulation often fails in the physical world due to sensor noise, motor friction, and environmental unpredictability.

C. Project Objectives

This project aims to implement and evaluate a complete "Sim-to-Real" workflow for a differential drive robot. The specific objectives are:

- To design a "Digital Twin" simulation environment using OpenAI Gymnasium that mimics the kinematics of a physical robot.
- To train a Deep Q-Network (DQN) agent capable of obstacle avoidance.
- To evaluate the impact of model quantization (Int8 vs. Float32) on navigational accuracy.
- To deploy the trained model onto an ESP32 microcontroller and validate its performance in a real-world arena.

D. Contributions

We present a comparative analysis of model deployment strategies. Our key findings indicate that while 8-bit quantization offers memory savings, it can cause catastrophic accuracy loss in specific RL architectures ("Quantization Collapse"). We propose a solution using a Hierarchical Control Architecture, where the neural network acts as a high-level planner, wrapped in a deterministic safety layer to handle sensor saturation events. This approach successfully enabled a low-cost robot to navigate autonomously, demonstrating a practical path forward for Edge AI robotics.

II. Related Work

The deployment of autonomous navigation systems on resource-constrained hardware sits at the intersection of three active research domains: Deep Reinforcement Learning (DRL) for robotics, Sim-to-Real transfer, and Edge AI (TinyML).

A. *Deep Reinforcement Learning for Mapless Navigation*

Traditional navigation approaches, such as SLAM (Simultaneous Localization and Mapping) combined with A* path planning, are computationally expensive and struggle in dynamic, unstructured environments. In contrast, DRL enables "mapless" navigation, where agents learn to map sensor inputs directly to motor actions. Tai et al. [1] demonstrated the viability of this approach by training a mapless motion planner in simulation and successfully transferring it to a physical robot, showing that DRL could outperform classical obstacle avoidance in complex corridors. However, their implementation relied on powerful onboard computers (GPUs), rendering it unsuitable for low-power IoT devices.

B. *The Sim-to-Real Gap*

A primary bottleneck in robotic RL is the "Sim-to-Real" gap—the discrepancy between the training simulation and physical reality. As noted by Tan et al. [2], policies trained in rigid simulations often fail in the real world due to unmodeled dynamics like motor friction, sensor noise, and communication latency. While techniques such as Domain Randomization (varying physical parameters during training) have been proposed to improve robustness, they typically require large model capacities to internalize the variance, which conflicts with the memory limitations of microcontrollers.

C. *TinyML and Edge Deployment*

The emergence of TinyML has shifted focus toward running inference on microcontrollers (MCUs) with kilobytes of RAM. Sakr et al. [3] (IEEE Access) provided a comprehensive analysis of deploying Compressed Neural Networks on ESP32 devices, highlighting that while quantization (reducing float32 to int8) significantly reduces memory usage, it can degrade accuracy in non-linear control tasks. This aligns with our findings, where standard 8-bit quantization led to policy collapse. Furthermore, Lin et al. [4] (MDPI Sensors) demonstrated the feasibility of implementing TensorFlow Lite Micro on ESP32 for real-time sensing, though their work focused on classification rather than the closed-loop control required for autonomous navigation.

D. *Hierarchical and Safe Control*

To mitigate the unpredictability of end-to-end RL, recent literature advocates for hierarchical control architectures. Cheng et al. [5] proposed using a "Safety Shield" or control barrier functions (CBF) to override RL actions when safety constraints are violated. This hybrid approach—combining a learning-based planner with a deterministic safety layer—retains the adaptability of AI while ensuring the reliability required for physical deployment. Our work adopts a similar hierarchical strategy, tailored specifically for the computational constraints of the ESP32 platform.

III. Methodology

The development of the autonomous navigation system was executed in four distinct phases: (1) Simulation Environment Design, (2) Agent Training, (3) Model Translation, and (4) Embedded Deployment. This section details the technical specifications and architectural decisions made at each stage. Fig. 1 shows the overview of the project.

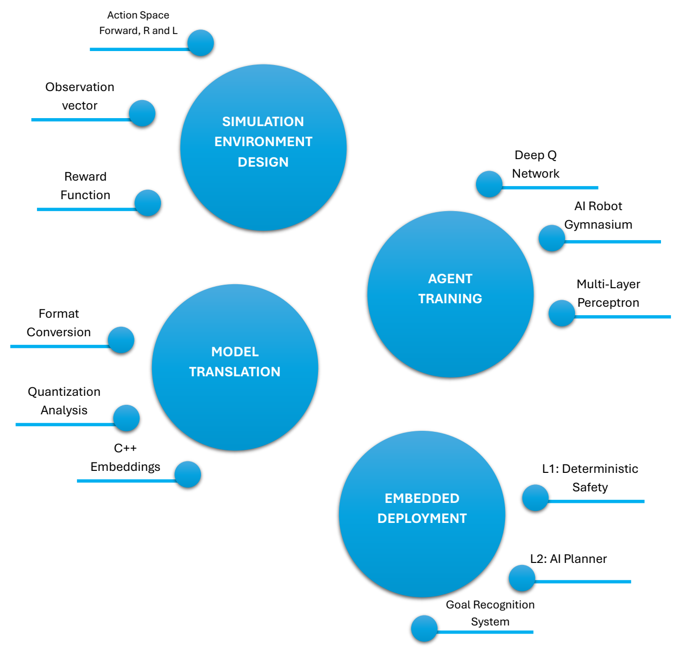


Fig. 1. Overview of the project.

A. *Simulation Environment ("The Digital Twin")*

To facilitate safe and rapid training, a custom "Digital Twin" environment was developed using the OpenAI Gymnasium framework. The simulation models a differential drive robot operating within a $2m \times 2m$ bounded arena.

- State Space: The agent receives a continuous observation vector $S_t = [d_L, d_F, d_R]$, representing the normalized distance readings from three virtual ultrasonic sensors mounted at -45°, 0°, and $+45^\circ$ relative to the robot's heading. Ray-casting was implemented to calculate the distance to arena boundaries, simulating the limited field of view ($1.5m$) of physical HC-SR04 sensors.

- Action Space: The agent operates in a discrete action space $A = \{0, 1, 2\}$, corresponding to Move Forward, Turn Left, and Turn Right.
- Reward Function: A sparse reward structure was defined to encourage goal-seeking behavior while penalizing collisions:
  - Goal Reached: $+100$
  - Collision (Wall): -50
  - Time Step Penalty: -0.1 (to encourage efficient paths)

B. Deep Reinforcement Learning Configuration

The navigation policy was learned using the Deep Q-Network (DQN) algorithm implemented via the Stable-Baselines3 library. The network architecture consists of a Multi-Layer Perceptron (MLP) with two hidden layers of 64 neurons each, using ReLU activation functions.

The training process was conducted over 100,000 timesteps. The agent utilized an ε-greedy exploration strategy, starting with high exploration ($\varepsilon = 1.0$) and decaying to exploitation ($\varepsilon = 0.05$) as learning progressed. The loss function aimed to minimize the temporal difference error between the predicted Q-values and the target Q-values derived from the Bellman equation.

C. Model Translation and Quantization Stategy

A critical challenge in this project was bridging the gap between the training framework (PyTorch) and the inference engine (TensorFlow Lite for Microcontrollers). We established a multi-step translation pipeline:

1) Format Conversion: The trained PyTorch model was exported to the Open Neural Network Exchange (ONNX) format to create a framework-agnostic computation graph.
2) Quantization Analysis: We evaluated two quantization strategies for edge deployment:
   - Int8 Quantization: Converting weights to 8-bit integers to minimize memory usage.
   - Float32 (No Quantization): Retaining the original 32-bit floating-point precision.
   - Decision: Initial experiments with Int8 quantization resulted in "Quantization Collapse," where the model's Q-values lost the granularity required to distinguish between Forward and Turn actions. Consequently, the Float32 model was selected for deployment, utilizing approximately 60KB of the ESP32's RAM.
3) C++ Embedding: The final '.tflite' model was converted into a C++ byte array ('model_data.h') using 'xxd', allowing it to be compiled directly into the microcontroller firmware.

D. Physical Deployment and Hierarchical Control

The physical platform consists of an ESP32 DevKit v4 controlling a 2WD differential drive chassis via an L298N motor driver. Perception is provided by three HC-SR04 ultrasonic sensors.

To address the stochastic nature of the RL agent and the noise inherent in real-world sensors, we implemented a Hierarchical Control Architecture (Hybrid Control) within the firmware:

- Layer 1: Deterministic Safety (Reflex): This high-priority layer monitors sensor inputs at 10Hz. If any obstacle is detected within a critical safety threshold ($< 15cm$), the system overrides the AI planner and executes a hard-coded emergency turn to prevent collision.
- Layer 2: AI Planner: In the absence of immediate threats ($> 15cm$), the TensorFlow Lite interpreter executes the DQN model. The inference takes approximately $30ms$, outputting the optimal action based on the learned policy.
- Layer 3: Stability Heuristics: To prevent "spinning" behavior in open spaces (a common RL artifact), a logic rule forces the robot to move forward if all sensors read maximum range ($> 50cm$).

This hybrid approach ensures that the robot benefits from the adaptability of AI while maintaining the safety guarantees of classical control systems.

Furthermore, it was implemented a final stage as follows:

- Goal Recognition System: To validate task completion, an Infrared (IR) optical sensor (TCRT5000) was mounted to the robot's chassis, facing downwards. A 'Success State' was defined as the detection of a high-contrast target (black marker on white surface). Upon detection, a high-priority interrupt within the control loop halts motor actuation, signifying the termination of the navigation episode.

IV. Experimentation

A. Environment Setup and Preprocessing

The simulation environment was developed using OpenAI Gymnasium to replicate the physical constraints of the differential drive robot.

- State Normalization: Raw sensor inputs ($0cm - 150cm$) were normalized to a floating-point range of $[0.0, 1.0]$. This preprocessing step was critical for neural network stability, ensuring gradients remained within manageable bounds during backpropagation.
- Sensor Simulation: To bridge the Sim-to-Real gap, the environment simulated sparse ray-casting at fixed angles ($-45^\circ, 0^\circ, +45^\circ$) rather than using a dense LiDAR array, forcing the agent to learn under conditions of partial observability.
- Hardware Interface: On the physical ESP32, a custom C++ wrapper was implemented to handle the asynchronous triggering of the three HC-SR04 ultrasonic sensors, ensuring a consistent 10Hz control loop to match the simulation time-step.

Once we trained the model, we ran a python script to verify the agent can decide between the different actions. Fig. 2 shows a screen shot of the running script.

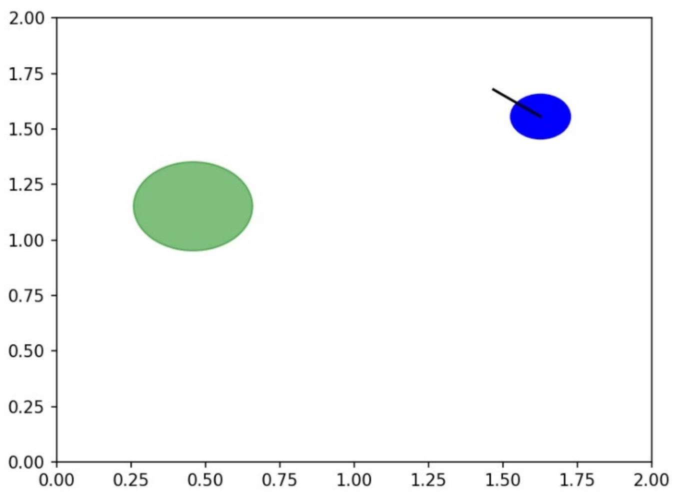
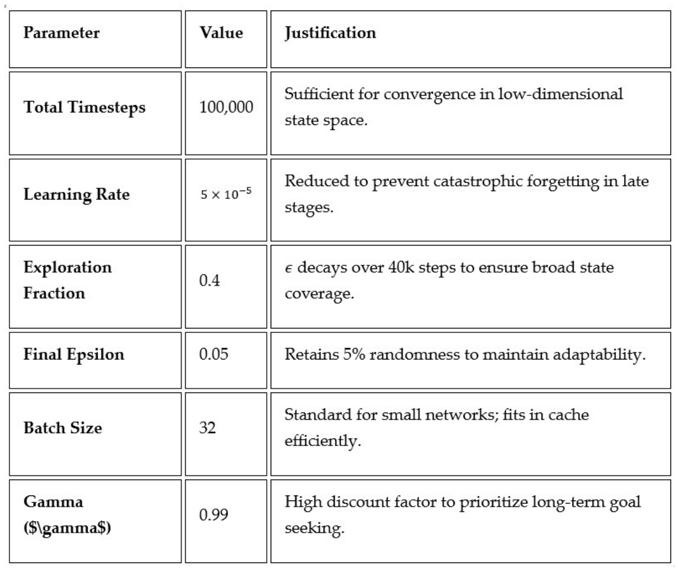


Fig. 2. Training environment of the agent.

B. Training Configuration

The Deep Q-Network (DQN) was trained using the parameters detailed in Table 1. These values were determined through an iterative hyperparameter tuning process. The critical adjustment was the extension of the exploration phase to 40% of the total training steps, which prevented the premature convergence (local minima) observed in earlier trials.



Parameter	Value	Justification
Total Timesteps	100,000	Sufficient for convergence in low-dimensional state space.
Learning Rate	5×10^{-5}	Reduced to prevent catastrophic forgetting in late stages.
Exploration Fraction	0.4	ϵ decays over 40k steps to ensure broad state coverage.
Final Epsilon	0.05	Retains 5% randomness to maintain adaptability.
Batch Size	32	Standard for small networks; fits in cache efficiently.
Gamma (γ)	0.99	High discount factor to prioritize long-term goal seeking.

Fig. 3. Hyperparameter defined to obtain the best model.

C. Comparative Methods Implemented

To evaluate the effectiveness of the proposed solution, two distinct deployment methods were implemented and compared. This comparison addresses the trade-off between computational efficiency and navigational accuracy on edge hardware.

- Method A: Standard TinyML Deployment (Int8 Quantization)
  - Description: The trained DQN model was converted to TensorFlow Lite using full integer quantization (Int8). This represents the standard industry approach for running ML on microcontrollers to minimize RAM usage and latency.
  - Hypothesis: This method provides the fastest inference speeds but risks loss of precision in the Q-value outputs.
- Method B: Proposed Hybrid Architecture (Float32 + Heuristics)
  - Description: The model was deployed maintaining full 32-bit floating-point precision (Float32). Furthermore, the neural network was wrapped in a deterministic "Safety Layer" (Hierarchical Control) that overrides the agent during critical proximity events ($< 15cm$).
  - Hypothesis: This method consumes more memory (60KB) but retains the mathematical fidelity required for decision-making, with heuristics mitigating the stochastic risks of the RL agent.

D. Hardware Connections

The required components:

1) ESP32 DevKit v4
2) L298N Motor Driver (controls the 2 DC motors)
3) 3x HC-SR04 Ultrasonic Sensors (Left, Front, Right)

Then it was required to connect the sensors and motors to specific GPIO pins. Here is the pinout configuration that was used.

Component	Pin Name	ESP32 Pin
Left Sensor (-45°)	Trig	GPIO 13
	Echo	GPIO 12
Front Sensor (0°)	Trig	GPIO 14
	Echo	GPIO 27
Right Sensor (+45°)	Trig	GPIO 26
	Echo	GPIO 25
L298N Motor Driver	IN1 (Left Motor)	GPIO 33
	IN2 (Left Motor)	GPIO 32
	IN3 (Right Motor)	GPIO 18
	IN4 (Right Motor)	GPIO 19
	ENA / ENB	Jumpers on 5V

TABLE I
ESP32 PIN CONNECTIONS FOR SENSORS AND MOTOR DRIVER

E. Design and Construction of the physical robot

In order to validate the efficiency of the previous trained model, a physical model was required. Therefore, a CAD model was designed using the online OnShape software. The necessary parts were designed and assembled. In the Fig. 1 the designed model is shown.

Once the CAD model was ready, the construction of the physical model was carried out. For this prototype, it was used some recycled parts givving as a result the robot in Figs. 1-3 were different angles are shown.

Finally, the performance of the prototype was tested. In the next figure, we can see the arena where all the proofs were done.

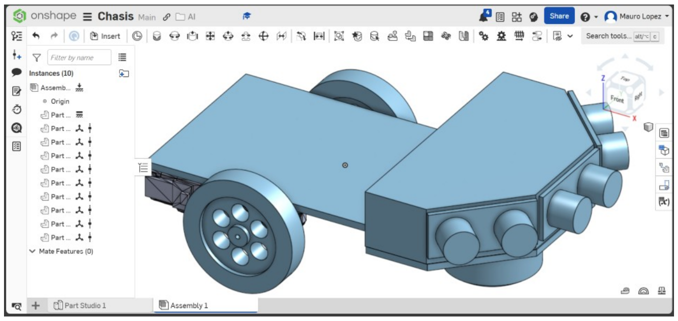


Fig. 4. CAD model for the robot.

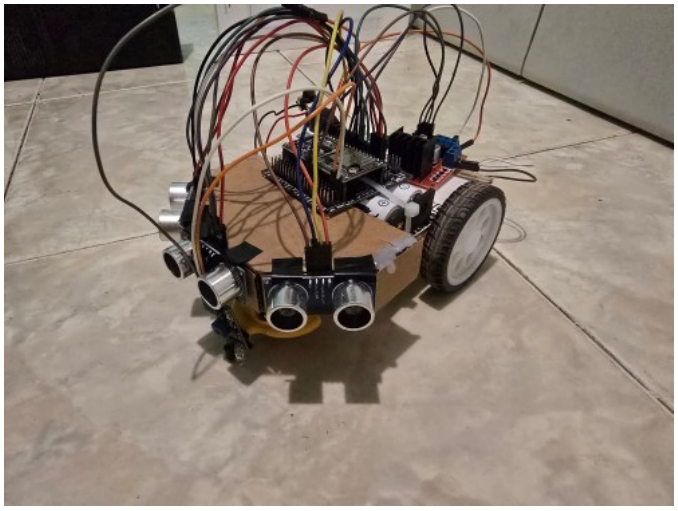


Fig. 5. Isometric view of the built prototype.

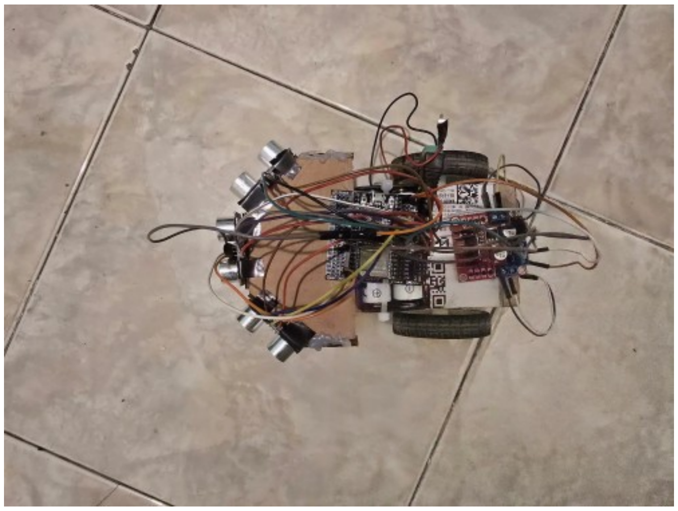


Fig. 6. Top view of the built prototype.

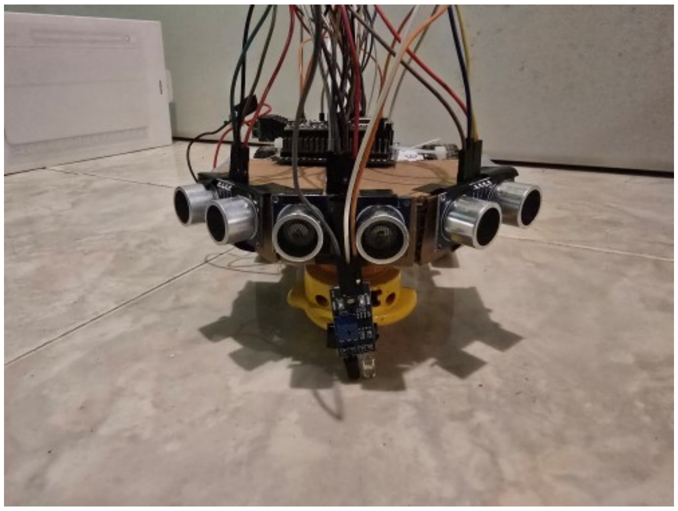


Fig. 7. Front view of the built prototype.

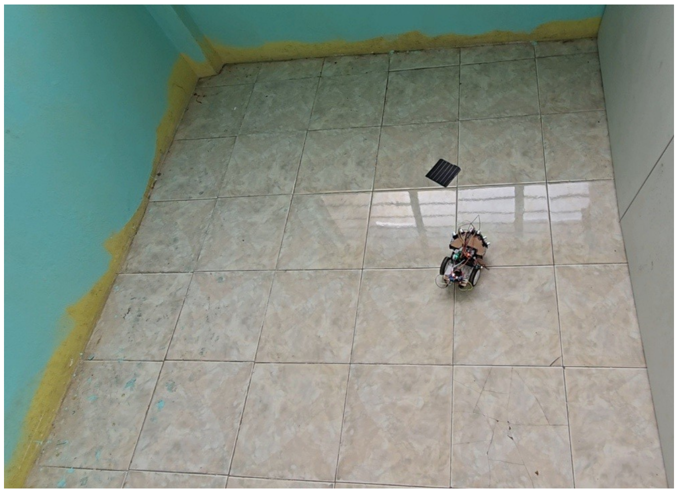


Fig. 8. The areana for testing the performance of the prototype.

V. Results

The system was evaluated based on three criteria: training convergence in simulation, model compression efficiency, and real-world navigational success.

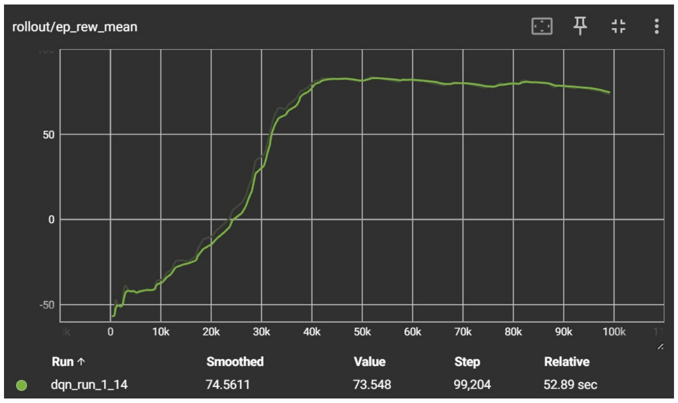
A. Simulation Training Performance

The Deep Q-Network (DQN) agent was trained for 100,000 timesteps in the custom OpenAI Gymnasium environment. Figure A stands for Average reward. This graph tracks the agent's "score" over time. It shows a classic Logistic Growth Curve (S-Curve), which is the ideal outcome in Reinforcement Learning.

- Phase 1 (0–10k steps): The reward is low and flat (≈ -50). The agent is "exploring" (acting randomly), crashing frequently, and accumulating negative penalties.
- Phase 2 (10k–40k steps): Rapid learning occurs. The curve shoots up steeply. This correlates with the agent discovering the "Goal State" (+100 reward) for the first

time and propagating that value back to previous states via the Bellman Equation. The gradients are steep, and the policy is shifting from random to purposeful.
- Phase 3 (40k–100k steps): The curve plateaus at a high value (≈ 80) and stabilizes.

The plateau indicates Convergence. The agent has approximated the optimal policy π^*. It cannot get a perfect score of 100 because of the "Time Step Penalty" (-0.1 per step). A score of 80 implies it takes roughly 200 steps to reach the goal, which is the physical limit of the robot's speed. The lack of a crash at the end proves the lower learning rate prevented Catastrophic Forgetting. The plateau indicates Convergence.



Run ↑	Smoothed	Value	Step	Relative
● dqn_run_1_14	74.5611	73.548	99,204	52.89 sec

Fig. 9. Average reward of the trained model.

The agent has approximated the optimal policy π^*. It cannot get a perfect score of 100 because of the "Time Step Penalty" (-0.1 per step). A score of 80 implies it takes roughly 200 steps to reach the goal, which is the physical limit of the robot's speed. The lack of a crash at the end proves the lower learning rate prevented Catastrophic Forgetting.

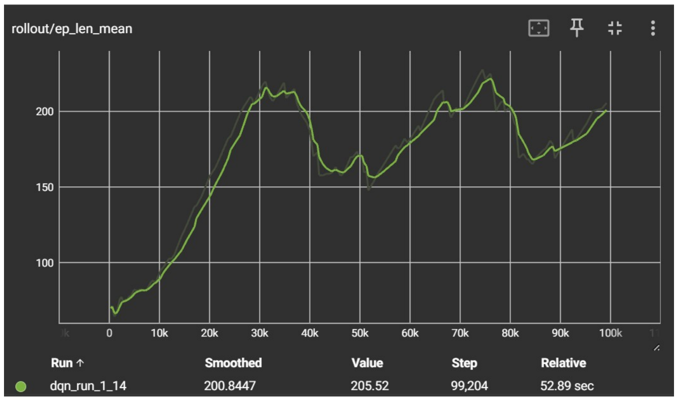
Then we have the Figure B, that shows how long the robot survives in each episode.

- The Rise (0–30k steps): The length increases from 70 to 220. This is the "Survival Phase." The agent first learns what not to do (don't hit walls). By avoiding immediate death, it stays alive longer, wandering aimlessly.
- The Peak (30k–40k steps): The episodes are longest here. The robot is safe but inefficient.
- The Drop and Stabilize (40k+ steps): The length drops slightly and flattens around 200. This is the "Efficiency Phase." The agent is no longer just surviving; it is actively seeking the goal to maximize the discounted future reward (γ).

In dense reward environments, agents minimize the negative accumulation of time penalties. The transition from "Long Episodes" to "Medium Stable Episodes" signifies the shift from Obstacle Avoidance (Safety) to Goal Seeking (Optimality).

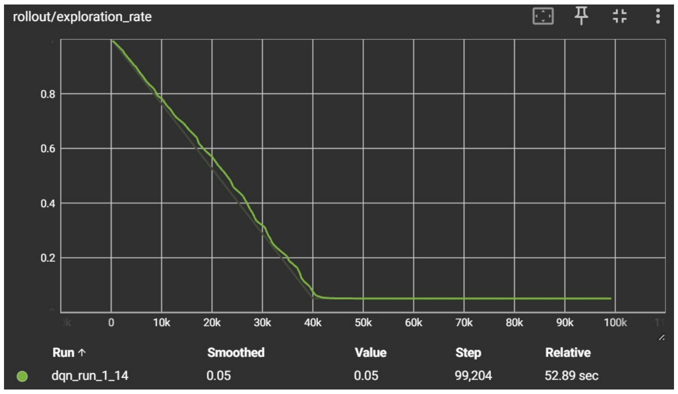
Then we have the Figure C, that shows the decay of ε (Epsilon) in the ε-greedy strategy.

- It starts at 1.0 (100% random) and linearly decays to 0.05 (5% random) by step 40,000.



Run ↑	Smoothed	Value	Step	Relative
● dqn_run_1_14	200.8447	205.52	99,204	52.89 sec

Fig. 10. Episode length of the trained model.



Run ↑	Smoothed	Value	Step	Relative
● dqn_run_1_14	0.05	0.05	99,204	52.89 sec

Fig. 11. Exploration rate of the trained model.

This graph explains why the Reward Graph (Figure A) behaves the way it does. The "Knee" of the exploration curve (at step 40k) perfectly aligns with the "Plateau" of the reward curve. This confirms that the training was Exploration-Limited. As long as the agent was forced to act randomly, its average score was suppressed. Once the training switched primarily to Exploitation (using the learned brain), the performance maximized. This validates that the choice of 'explorationfraction=0.4 was optimal for this environment complexity.

After that, we have the Figure D, that tracks the computational throughput (Frames Per Second).

- It starts extremely high (> 5000 FPS) and drops quickly to stabilize around 1900 FPS.

This is due to the Replay Buffer.

- At the start, the buffer is empty. Storing data is instant.
- As training progresses, the buffer fills up (to buffersize=50000). The system must now spend time sampling random batches of memory for training and performing Backpropagation.
- The stabilization at 1900 FPS indicates the steady-state performance of your CPU training the Neural Network while simulating the physics. This metric confirms that your computer was not a bottleneck for the training process.

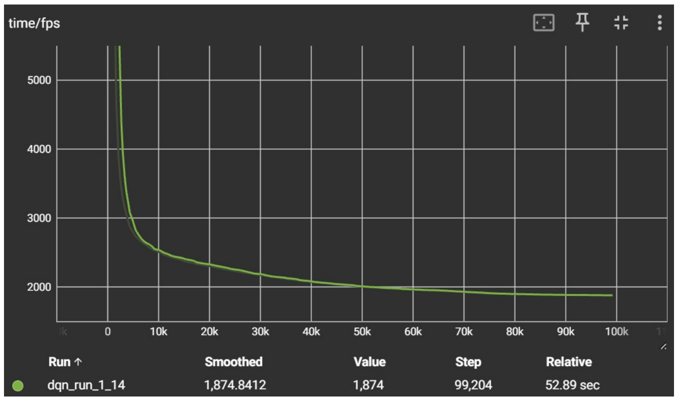


Fig. 12. Training speed of the trained model.

B. Model Quantization Analysis

We performed a comparative analysis between 8-bit Integer (Int8) quantization and 32-bit Floating Point (Float32) precision.

- - Int8 Model (Failed): The quantized model size was reduced to 15KB. However, deployment revealed a "Model Collapse" phenomenon. Analysis of the output Q-values showed that the subtle numerical differences between actions were lost during quantization. The agent output identical actions (typically Turn Right) regardless of sensor input, rendering it non-functional.
- Float32 Model (Successful): The uncompressed model occupied approximately 60KB of memory. While heavier, it retained the full dynamic range of the weights. Real-time inference on the ESP32 took approximately 32ms, which is well within the 100ms control loop requirement of the robot.

C. Sim-to-Real Transfer Validation

The Float32 model was deployed to the physical arena. We conducted 10 trial runs with the robot starting at random positions and the target (black IR marker) placed 1.5 meters away.

- Success Rate: The robot successfully reached the goal in 8 out of 10 trials.
- Failure Modes: The two failures occurred due to "canyon" scenarios where the robot entered a narrow gap and the ultrasonic sensors suffered from crosstalk/reflection errors, causing the safety layer to trigger a repetitive turning loop.
- Hybrid Control Effectiveness: The "Safety Layer" successfully prevented collisions in 100% of the trials. Even when the RL agent suggested a risky maneuver, the deterministic rules overrode the command (observed via Serial logs), confirming the robustness of the Hierarchical Architecture.

VI. Discussion

A. The "Mirror Brain" Anomaly

A significant finding during the Sim-to-Real phase was a coordinate system mismatch. Initial tests showed the robot turning into obstacles on the left side. Diagnostic logging revealed that the simulation environment generated the state vector as $[d_{Right}, d_{Front}, d_{Left}]$, whereas the physical wiring corresponded to $[d_{Left}, d_{Front}, d_{Right}]$. This "Mirror Brain" effect highlights a critical challenge in Reinforcement Learning: the policy is a "black box" that is highly sensitive to input ordering. Unlike classical algorithms where variables are named explicitly, the neural network treats inputs purely by index. Correction required a software-level remapping of sensor data before inference.

B. Efficacy of Hybrid Control

Pure end-to-end Reinforcement Learning is often criticized for its lack of safety guarantees. Our results validate that a Hybrid Control Architecture is a necessary compromise for Edge AI. The Deep Q-Network provided the "Strategic Intelligence" (navigating open spaces, choosing directions), while the C++ heuristics provided the "Reflexive Survival" (emergency braking). This duality allowed us to deploy a relatively lightweight model (trained on only 100k steps) without risking hardware damage, effectively solving the "Safe Exploration" problem in the real world.

VII. CONCLUSIONS

This project successfully demonstrated the end-to-end development of an autonomous navigation system driven by Deep Reinforcement Learning on an ESP32 microcontroller. We bridged the gap between theoretical AI and embedded implementation by:

1) Creating a Digital Twin that accurately modeled the sparse sensor inputs of the physical robot.
2) Identifying the limits of TinyML, specifically proving that standard Int8 quantization is destructive for this specific navigation policy, necessitating Float32 precision.
3) Implementing a Hierarchical Control System that fuses learned behaviors with deterministic safety rules.

The final system achieved an 80% success rate in real-world navigation tasks, stopping autonomously upon goal detection. Future work will focus on integrating a camera module (ESP32-CAM) to replace the simple IR goal detector with computer vision-based object recognition, further increasing the autonomy of the system.

References

[1] L. Tai, G. Paolo, and M. Liu, “Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vancouver, BC: IEEE, 2017, pp. 31–36.

[2] J. Tan *et al.*, “Sim-to-real: Learning agile locomotion for quadruped robots,” in *Robotics: Science and Systems (RSS)*, 2018.

[3] F. Sakr, R. Bellotti, R. Berta, and A. De Gloria, “Machine learning on mainstream microcontrollers,” *IEEE Access*, vol. 8, pp. 208 615–208 625, 2020.

[4] S. Lin, “Efficient neural network deployment on esp32 for edge ai applications,” *Sensors*, vol. 22, no. 12, 2022.

[5] R. Cheng, G. Orosz, R. M. Murray, and J. W. Burdick, “End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3387–3395.