

Adaptive Controls for Hardware Fault-Correction in Embedded Systems

Aakash Mishra
Stanford University
Stanford, California, USA
aamishra@stanford.edu

Dea Dressel
Stanford University
Stanford, California, USA
ddressel@stanford.edu

Abstract

Embedded system fault-detection is a multi-modal problem that can difficult to solve. There are several issues in this problem space including the limitations of real-time fault detection and correction on remote-deployed embedded systems. One problem is that standard methods of assessing a bug on a remote-deployed system are likely subject to temporal and physical constraints. Adding to this problem is the issue with trust-ability of powered hardware components. It can be difficult to discern whether sensor input is reliable or assert whether or not the behavior of a motor is nominal. Overall, the options for a deployed system can be limited when it does encounter a hardware related problem. In our work, we aim to address this issue by creating a framework for adaptive controls for hardware fault-correction using machine programming.

We propose an autonomous method for addressing some of the issues with self-correcting systems with three novel innovations. We first present a new intermediate representation for firmware code in an embedded system called a Hardware Highlight Parse Tree (HHPT). Secondly, we train and evaluate a GNN-based detector that can detect the presence of anomalous hardware behavior. Lastly, we demonstrate the efficacy of a Reinforcement Learning based system that can adaptively control it's hardware to respond to possible hardware anomalies by combining the RL-agent model with the GNN detector. We show that it is possible to create a robust and adaptive system that can self-diagnose and correct hardware issues that may otherwise be difficult to recover from. We open-source our work and code here: <https://github.com/aakamishra/embedded-repair-mp>.

Keywords: Embedded Systems, Machine Programming, Hardware Fault Detection, Hardware Fault Correction, Reinforcement Learning

1 Introduction

Hardware fault analysis for embedded systems can be a difficult problem to tackle with standard methods for debugging. Embedded systems facilitate the integration and coordination between mechanical devices and sensors in robotic applications, which leaves them prone to facing non-traditional software and hardware fault points [9]. Additionally, a distributed system with several untrustworthy components

presents a unique problem for deterministic algorithms of program analysis [11]. Moreover, a fast and reliable method for detecting and correcting hardware faults would have to overcome many challenges.

1.1 Problem 1: Embedded Systems Fault-Detection is a Multi-Modal Problem

One of the challenges with debugging a robotic system is having to consider the multiple modalities of input [16]. Ideally, an embedded-system analysis tool would maintain an understanding of how the software and hardware work together. However, as embedded systems increase in complexity, the relationship between the software and hardware can become more convoluted. For example, an embedded system could run software that drives the behavior of several powered hardware components at one time. The interconnectedness of the software and hardware makes fault tracing difficult; erroneous behavior may be a result of the combined execution/interaction of hardware and software such that it may not be attributable to either one when viewed in an isolated way [7]. Therefore, a data structure that can represent how the software interacts with the hardware of an embedded system is needed.

1.2 Problem 2: Real-Time Fault Detection and Correction is Limited in Remote-Deployed Embedded Systems

Non-deterministic hardware errors in embedded systems often surface during deployment and are affected by environmental conditions and inputs [2]. Effectively solving these failures requires successful detection and isolation of the hardware fault and an appropriate recovery technique. To ensure the stable operation of a system, these hardware faults must be addressed in a fast manner. Doing so, however, is made difficult by temporal and spatial limitations of deployed embedded systems [3]. It may take a long time to detect and isolate the hardware error and some recovery techniques require physically replacing the faulty hardware in the system. Moreover, manual recovery strategies that require physically handling the system may be impossible, which is the case for remote-deployed embedded systems like satellites and space robotic systems. Therefore, a real-time fault detection method that can continuously monitor

the health of several hardware components would be advantageous. The fault could then be quickly detected, and the observed hardware condition data could be helpful in identifying an appropriate recovery strategy.

1.3 Problem 3: Powered Hardware Components Are Not Trust-able

Most embedded systems have a computer control system that measures, monitors, and executes all of the control functions in order to accomplish some pre-defined functionality. Reinforcement Learning (RL) agents have demonstrated success as end-to-end computer controllers that can learn a strategy for solving optimal control problems [6, 8, 12, 15]. The robustness of these RL agents, however, is constrained by the assumption that the powered hardware of the system is trustworthy [13]. The RL agents can perform well when all hardware components are functioning normally, but typically fail if a hardware component breaks. Ideally, an RL agent would be trained to handle various hardware faults such that the system can continue to function in some optimal way.

1.4 Our System

To address these problems, we present a RL control system that observes detected hardware faults in an embedded system using a Graph neural network and an novel intermediate representation. Our approach utilizes the structure of the code provided in the firmware for an embedded system to help the system understand the relationship between hardware and software components. We aim to create a system that aligns with the adaptive pillar of the Machine Programming paradigm presented in Gottschlich et al. [4]. This paper makes the following technical contributions:

- We present the *Hardware Highlighted Parse Tree (HHPT)*, a program representation designed to capture software-hardware relationships.
- We present a Graph Convolutional Neural Network Hardware Fault-Detector framework and show its efficacy across three robot configurations.
- We compare the performance of a RL agent acting with (i) nominal hardware conditions, (ii) undetected hardware faults, and (iii) detected hardware faults and show that the RL agent can better optimize its performance when given the GNN detection output in its observation space.

2 Background

Hardware fault detection, isolation, and recovery (FDIR) is a sub-field of control engineering which involves monitoring a system, identifying when a fault has occurred, and pinpointing the fault type and its location [1]. In order to identify when a fault has occurred, the FDIR mechanism must first have some understanding of the expected operational

functionality of the system. Building this understanding is typically achieved by means of monitoring internal signals and available sensors during nominal operation.

Ensuring the stable operation of a complex system requires fast FDIR, motivating recent research in this area [1, 10, 14, 19]. Shan et al. [14] presented CS-DKELM, a general diagnosis method which realizes the functional integration of signal compressed sampling, feature adaptive extraction and fast fault diagnosis. Compressed Sensing (CS) utilizes an approximate optimal sampling scheme to preserve all the information contained in the original signal while effectively reducing the dimension of the signal. As a result, the storage and calculation costs required for feature extraction and fault classification by the Deep Extreme Learning Machine (DKELM) are greatly reduced. DKELM is a multi-layer neural network which performs greedy layer-by-layer unsupervised learning. With an average fault classification accuracy of 99% and average run-time of 0.17s, the CS-DKELM method can be directly applied to industrial embedded platforms as an effective algorithm for mechanical equipment health monitoring. While this work effectively detects and isolates hardware faults in an embedded system, it relies on manual recovery techniques (e.g., replacing faulty hardware).

Reinforcement Learning (RL) is a sub-field of machine learning that has gained traction in solving control, or sequential decision making, problems [17]. A RL agent interacts with an environment to learn a control strategy based on the feedback it receives for its actions taken. Under the context of RL, any action leads to two consequences: receiving an immediate reward and arriving at a new state. A control agent should consider the delayed future rewards corresponding to the new state before selecting an action, instead of simply selecting the action with the highest reward. This ability to optimize the trade-off between short-term and long-term benefits makes RL a strong option for solving control problems.

Various works have explored leveraging RL to optimize controllers for embedded systems [6, 8, 12]. Koch et al. [8] integrated RL in the embedded system development process to automatically train and deploy agents in transient automobile driving conditions. Sallab et al. [12] proposed a pipelined framework for end-end training of a Deep Neural Network for autonomous driving. Greatwood and Richards [6] developed a method for enabling a quadrotor micro air vehicle to navigate unknown environments using RL and model predictive control. To our knowledge, no prior works in this space have investigated training a RL agent with hardware fault observations to improve system performance when hardware faults are present.

3 Technical Design

Aiming to address the current limitations of existing hardware fault detection, isolation, and recovery methods, we

propose a hardware fault detector model that feeds into a reinforcement learning based control system to optimize embedded system performance in the face of hardware failures. This system has two main advantages: (1) a real-time hardware fault detection method and (2) an adaptive control system. It leverages a Hardware Highlighted Parse Tree (HHPT) which captures the relationship between powered hardware components and the firmware.

3.1 Simulated Environment, Robot Setup and Objective

We adopt the following setup from the [openai-gym-line-follower](#) repository designed by *Nejc Planinšek*.

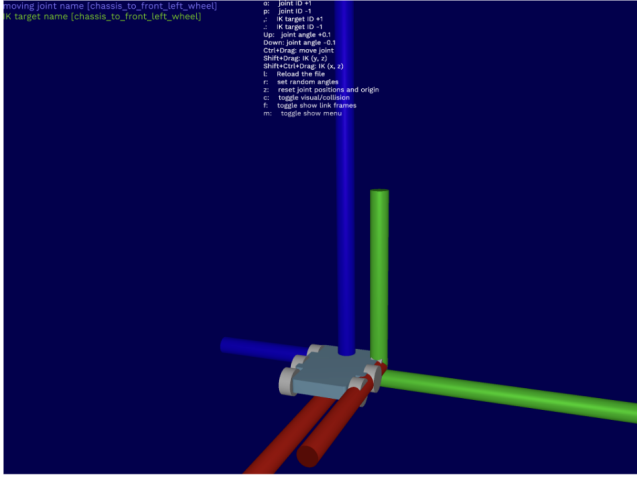


Figure 1. Example URDF File for a 6-wheel robot being visualized through the URDF-Viz tool.

Physics Environment. In order to rapidly generate large amounts of simulated data for sensors on an embedded system, we opt to use a physics engine and an iterable reinforcement learning environment. We utilize PyBullet to simulate the Martian terrain and create a line following robot with an objective of following a charted track. We define the multiple physics settings in PyBullet, including static friction for the wheel material, gravity, inertial properties, air resistance, air pressure, and visibility.

Robot Configuration. We create URDF configuration files using the URDF-viz library to set modeling and physical constraints on the simulated robot agent. These files also contain the coloration and physics settings for the robot representation. This includes settings such as moment of inertia, mass distribution, weight, friction, axle mounts, and angular momentum. We create three main configurations for the robot: the first configuration has six DC motors, six encoders, and one accelerometer; the second configuration has four motors, four encoders and one accelerometer; and the third configuration has two motors, two encoders and one accelerometer.

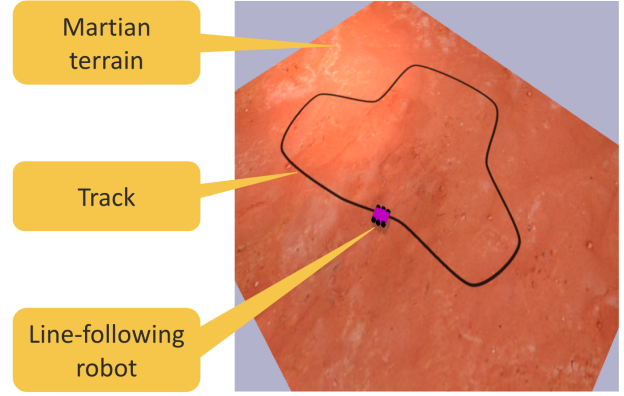


Figure 2. Simulated physics environment with the Martian terrain, track, and line-following robot.

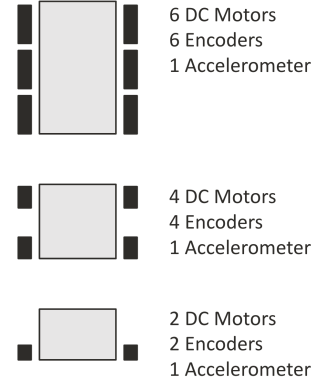


Figure 3. Visualization of the three robot configurations.

RL Environment Setup. We use OpenAI Gym to setup an episodic environment where we can record data and deploy our models. The API allows us to render an interface with which we can iteratively use the step function to call on our RL agent model training code and data collection APU as well.

Objective. We create a situation in the simulator where the robot has to follow the assigned track while avoiding small obstacles like rocks and other debris on the terrain. The goal of the robot is to keep moving around in a circle while staying on the track for as long as possible.

3.2 Hardware Highlighted Parse Tree

In this section, we first introduce the definition of a Hardware Highlighted Parse Tree (HHPT) by defining its constituent parts. We then describe how we convert a HHPT into an adjacency matrix representation and extract the node type vector.

DEFINITION 1 (PARSE TREE). A *parse tree* is a hierarchical representation of a program's syntactic structure according to

pooling and V_k = the set of k selected nodes. The two most common methods are mean-pooling and max-pooling:

$$h_{pool} = \frac{1}{k} \sum_{v \in V_k} h_v \quad (\text{mean pooling}) \quad (4)$$

$$h_{pool} = \max_{v \in V_k} h_v \quad (\text{maximum pooling}) \quad (5)$$

For our model, we use **maximum pooling** (Equation 5) with a **learned function of the node features** (Equation 2) for the Top-K selection.

The dimensions of the detector’s output is dependent on the number of Hardware Input Nodes and Hardware Output Nodes in the HHPT. If there are i Hardware Input Nodes and j Hardware Output Nodes, then the output of our GNN would be a vector of $i + j$ size. The vector contains the logit values that predict the likelihood that the hardware which the node represents is faulty.

3.3.2 Data Collection. The data collection process begins with the creation of realistic hardware failure scenarios. Two primary types of scenarios are considered. (1) **Encoder Input Node Failures.** These scenarios simulate malfunctioning encoders, which provide corrupted data to the robot’s controller logic. This is achieved by injecting Gaussian noise into the encoder readings with a specific probability, effectively mimicking a faulty sensor. (2) **Motor Output Node Failures.** These scenarios emulate situations where motors fail to respond to the controller’s commands. This is implemented by randomly resetting the motor output to zero during operation.

To generate data for training the GNN detector, a pre-trained RL agent operates the robot within each simulated failure scenario. This RL agent has been trained in environments with fully functional hardware, ensuring that its behavior deviates significantly when encountering unexpected hardware faults. Each failure scenario is executed for a large number of episodes (i.e., 10,000) with each episode lasting a specific time duration (i.e., 400 time steps). Each episode is assigned a binary encoded label, where each bit corresponds to a specific hardware node. The label value is 1 if the corresponding node is deemed faulty and 0 otherwise. The data for each episode is further processed by dividing the time steps into sequences and constructing an HHPT. Each Hardware Input Node or Hardware Output Node in the HHPT contains 10 feature values corresponding to the recorded values at each time step within the episode snippet.

By simulating various failure scenarios, the collected data encompasses a wide range of hardware faults, enabling the GNN detector to learn and generalize effectively. Utilizing a pre-trained RL agent ensures that the collected data reflects situations where the robot encounters unexpected hardware behavior, making the anomalies more pronounced and easier to identify. We aim to capture enough information per

each 10-timestep snippet to have the model determine some probabilistic judgment on the faulty hardware component.

3.4 Training Setup

We create a train-eval-test split of 60-20-20 where we control for the ratio of perfect examples where there are no hardware failures by ensuring the label ratio is 10% of each set. We train the model on 360,000 HHPTs and their corresponding labels ($10,000 * 400 / 10 * 0.8$) for 100 epochs using Adam as our optimizer. We set an initial learning rate of 0.01, $\beta_1 = 0.9$ and $\beta_2 = 0.99$. We train the model with Binary Cross entropy loss with logits in order to treat each label as a probability series. After 100 epochs, we find that the model converges to a final average loss of 0.617 on the evaluation set.

3.5 RL Framework

3.5.1 Actor-Critic Setup. In order to teach the line following robot to complete the objective task, we use deterministic deep policy gradient learning (DDPG). We create a sequential based actor model and a sequential based neural network for our critic model such that the actor model attempts to assign a set of actions for the robot to take. The critic model attempts to estimate the Q-value of the actions that the actor model has assigned for a given step using the information from the past steps as well as the observation space.

3.5.2 Action Space. In our RL framework, the action space consists of assigning voltages scaled on values from 0 to 1 for each motor on the robot. Those values are then passed into the DC motor API of our physics simulator which then calculates the corresponding torque applied by the motor given the voltage value from the controller.

3.5.3 Observation Space. The observation space for the RL-agent in the scenario with nominal hardware functionality consists of sensor values from each of the encoders which report back the speed of each wheel and the values from the accelerometer which informs the controller about the orientation and acceleration of the robot in the x, y and z planes. Additionally, the robot is given information through a camera sensor that constructs a Bezier curve to determine the approximate spline path it needs to take in its current environment. These values are all synthesized and concatenated together to be passed to the RL controller as input at each step. In the scenario where we train the RL-agent to use the GNN to get detection information about potential hardware failure, we concatenate the GNN output of $i + j$ logit values to the observation space and feed it into the RL-agent.

3.5.4 Training Setup. We use the Ornstein-Uhlenbeck (OU) process which adds noise to the actions chosen by the RL-agent to avoid falling into a local optima due to a solely deterministic policy. The equation for this process is

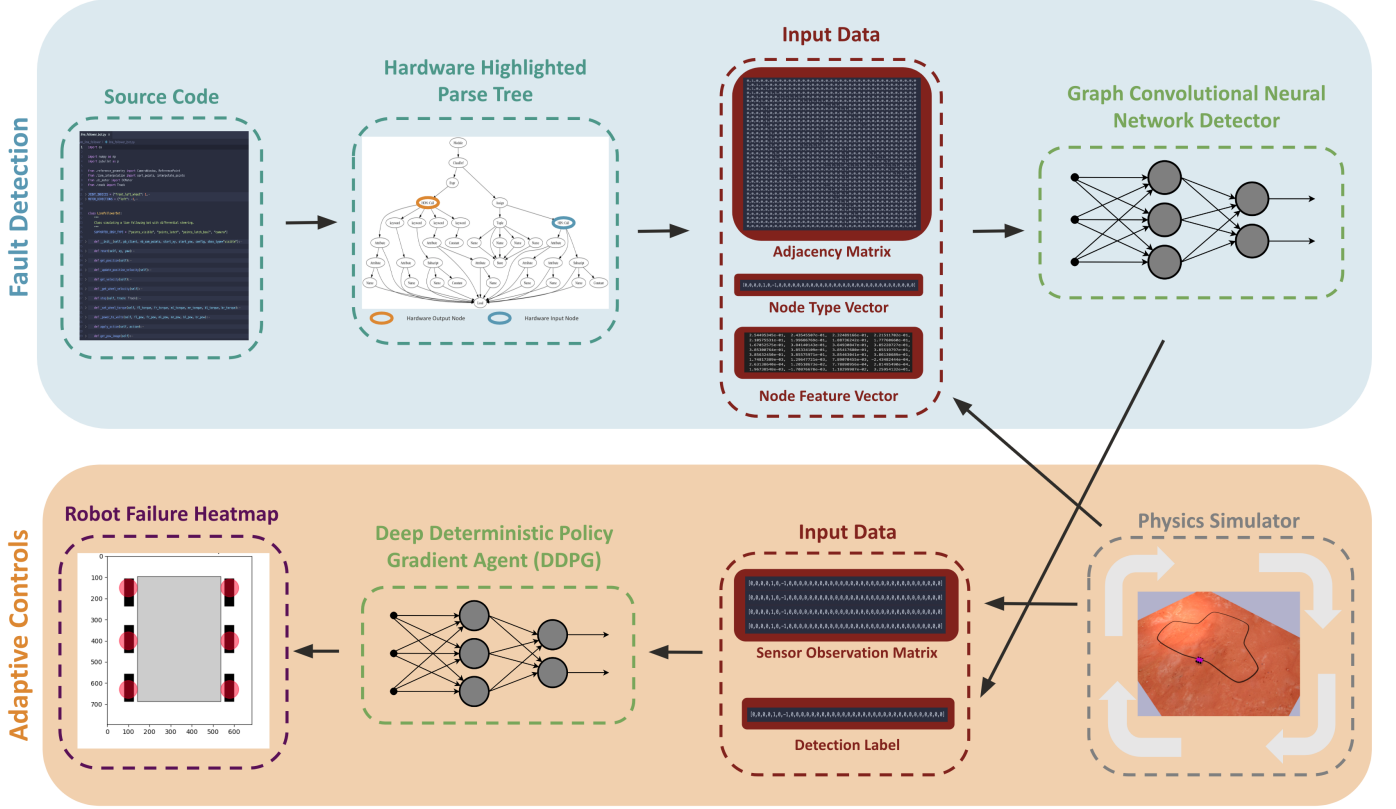


Figure 5. System design.

$$dX_t = \theta(\mu - X_t)dt + \sigma dW_t \quad (6)$$

where X_t is the state of the process at time t . θ is the rate of mean reversion which controls the rate at which the process returns to its mean value μ . σ represents the diffusion coefficient which controls the amount of noise. Lastly, dW_t is a Wiener process increment. Note that noise encourages the agent to explore a more diverse set of actions. For our OU process, we set $\theta = 0.5$, $\mu = 0.5$, and $\sigma = 0.5$.

For our DDPG agent, we use Keras-RL to setup the objective function and set $\gamma = 0.99$, the learning rate $\eta = 1e-3$, and the number of warm-up steps to be 80 for both the actor and the critic model. We train each scenario's RL-agent with 100,000 steps and a window size of 5 (the number of past observations the model can use as input).

4 Experimental Results

4.1 System Overview

The completed embedded self-repair system takes the following steps per cycle of simulation:

At the initial setup of the system, the program generates the structure of the HHPT given the input of the firmware file and identifies the Hardware Output Nodes and Hardware

Input Nodes for which to track the values of over time. As the simulation engine starts an episode, logged values are reported to the HHPT API which are then added as node feature values in sets of 10 steps. At the 11th step, the HHPT, with each node feature list filled with 10 values, is passed to the GConv (GNN) detector model. The GConv (GNN) detector model takes the values of the HHPT and outputs the detection labels, a probability score array that determines the logit score for each hardware node failing.

The detection labels are then passed to the RL agent which takes the robot's sensor observation array and the GNN detection labels as input and outputs an action set. The action set determines the voltage level distributed to each motor in order to help the robot accomplish its task in following the line track set around the simulation environment.

If some hardware component is broken, then the ideal sequence of events would be as follows. First, the values of that hardware node are logged along with the rest of the nodes and sensor outputs. Ideally, some discernible difference would be present in the final format of the HHPT. The GNN model would then output detection labels with a high probability of failure assigned to the node that has actually failed. Finally, the RL controller would take that into account

and assign the motion of the motors to behave in a way that would compensate for the faulty hardware component.

We illustrate a comprehensive visual overview of this system and algorithm in figure 5.

Algorithm 1 Self-Repairing Embedded System Algorithm

Require: H : Firmware file, $GConv$: GNN detector model, $RLAgent$: Reinforcement learning agent, S : Simulation engine.

Ensure: V : Voltage applied to each motor.

```

1: Initialization:
2: Generate HHPT structure from  $H$ 
3: Identify nodes for tracking
4: Initialize node feature lists for tracking values
5: while True do
6:   Start episode in  $S$ 
7:   for  $i = 1$  to 10 do
8:     Get logged values from  $S$ 
9:     Add logged values to correct node feature array
10:  end for
11:  Pass HHPT with filled feature lists to  $GConv$ 
12:  Receive probability scores for each node from  $GConv$ 
13:  Get sensor readings from  $S$ 
14:  Pass sensor readings w/ logit scores to  $RLAgent$ 
15:  Receive action set from  $RLAgent$ 
16:  Apply action set to motors in  $S$ 
17:  Monitor system behavior
18:  if Hardware failure detected then
19:    Record data related to failure
20:    Update  $RLAgent$  with failure information
21:    Adapt motor control strategy based on failure
22:  end if
23: end while

```

4.2 Testing

We test our system for three different robot configurations in three different scenarios:

(*Scenario 1*) The first scenario we test has every component working in their ideal state (i.e., there are no hardware faults in the system). We train the RL-agent in this scenario without the GNN detection as input and measure the best possible performance the ideal RL-agent can achieve given no environmental perturbations.

(*Scenario 2*) The second scenario we test has hardware failures that occur with some probability and can occur at any given time step. We allow the RL controller to train on these cases without GNN detection information, same as in *Scenario 1*. The goal of this second scenario is to establish the baseline for what the RL-agent can do alone without some sort of adaptive fault detection mechanism.

(*Scenario 3*) The third scenario has random hardware faults that can occur at any given step while the RL agent observes

the GNN detector output. This allows the agent to adapt to the new fault-detection information given the specified indices for which hardware node is likely broken. We then test this system to see if we can improve the *Scenario 2* baseline and close the gap with the *Scenario 1* topline. We test the entire simulated system setup on a commodity Macbook Pro with an Apple Unit (AU) CPU that contains 14-cores with 10 performance cores and 4 efficiency cores. In order to make sure our simulation is replicable, the random seed set for each scenario is "109".

4.3 Evaluation Metrics

To evaluate each scenario with each robot configuration, we chose to compare the maximum value of reward the robot can achieve in its main objective of following the assigned line path on the simulated surface by hitting each of the checkpoints. We define this rewards function as follows:

$$\begin{aligned}
 nb_{checkpoints} &= 500 \\
 e_t &= \text{Distance to closest point on track at step } t \\
 max_{error} &= \text{Maximum allowed track error} \\
 e_t &= \frac{e_t}{max_{error}} \\
 reward &= \frac{1000}{nb_{checkpoints}} \\
 R_t &= \begin{cases} reward * (1 - e_t)^2 & \text{if reached check at } t \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

Note that if the maximum amount of track error is reached, then we restart the episode with the assumption that the robot would not have recovered from that point onward.

4.4 Results

We test out each of our robots with each scenario. For *Scenario 1*, we observe that the topline value of the rewards function is around 800 for each of the robot configurations. We note that the six-motor robot has the highest perfect max reward; this is likely due to its extended manoeuvrability capabilities provided by having an more sets of wheels.

For *Scenario 2*, we introduce random hardware failures and observe that the RL-agent has a hard time learning from and adapting to them. We notice a steep decline in the value of the reward function score. The decrease is 48% for the 6-wheeler, 55% for the four wheeler and 63% for the two-wheeler. Note that the value of the rewards function seems to also decrease with the smaller sized robots. The reasoning for this stems from the larger robot having greater hardware redundancy than the smaller robots. For example, in a given episode, if the 2-wheeler robot lost both of its' motors, it would also lose the ability to drive forward. However, this is not necessarily the case for the 4-wheeler and least of all of

Robot Configuration	Perfect Max Reward	Reward w/ Hardware Faults	Reward w/ Hardware Faults + GNN Detector
	830.5	432.5	670.3
	810.3	378.2	523.3
	798.3	299.4	360.1

Figure 6. Experimental results.

the 6-wheeler, which has the greatest number of hardware components and therefore greater redundancy.

Using *Scenario 2* as our baseline for the reward value we need to improve upon, we then record the values for *Scenario 3* where the GNN information is fed back into the RL controller. Here we see a marked improvement over the previous hardware fault baseline. We see that the addition of the GNN input to the RL agent controller brings 55% gain for the 6-wheeler, a 38% gain for the 4-wheeler and a 20% gain for the 2-wheeler. We note that the gain for the 2-wheeler is still low because even if the RL-agent was "aware" of a hardware anomaly, it would have very little it could actually change in the action space to accommodate for it due to physical limitations. In contrast, if the middle right wheel was broken on the the 6-wheeler robot, then theoretically the RL agent could divert power intended for the middle right wheel to the front right and back right motors. We present these findings in figure 6.

5 Conclusion

We aimed to present a prototype system that addressed some key issues that affect remote-deployed embedded systems in the real-world. Namely, we wanted to create a system that could autonomously self-diagnose and then self-correct hardware problems. We chose to adopt a scenario that involved a simulated tank-drive based robot that had the task of following an assigned path. We then simulated several types of hardware failures that could occur in real-life, such as motor and sensor failures. We were able to use a novel IR, the Hardware Highlighted Parse Tree, to represent the software and hardware dependencies in the form of a graph based on the the firmware code. This graph was then populated with time steps values of the respective hardware node from each step of the simulator. Using the GNN trained

on parsed HHPTs and an RL agent that could optimize the action space of the robot given the sensor values and GNN hardware-fault detection output in the observation space, we show that it is possible to create a controller that can dynamically respond to hardware failure and optimize for it post-detection. This type of method can be very helpful for robots that cannot be analyzed or debugged due to temporal and spatial limitations. Examples of systems like this include rovers, satellites, deep-sea robots and many more. We hope that our work provides a proof-of-concept framework for future research along this avenue and encourages further development of autonomous systems that can self-diagnose and self-correct.

6 Future Work

This work presents a novel self-repairing embedded system framework with promising results. However, several aspects require future exploration to validate its effectiveness in real-world applications. Firstly, the current implementation utilizes a simulated environment and system. While this provides a controlled platform for initial development, real-world deployment necessitates consideration of resource limitations on edge devices. Memory and computational constraints may restrict the implementation of complex models on actual embedded systems. Therefore, future work should investigate efficient model architectures optimized for resource-constrained environments.

Secondly, the investigated scenario involved a limited and non-diverse environment. Expanding the testing domain to include other embedded systems like flying drones or underwater submersibles would be valuable to assess the generalizability of the proposed approach. Additionally, exploring a

wider variety of HHPTs from different hardware configurations could provide deeper insights into the robustness and adaptability of the system.

Furthermore, to validate the intermediate representation and the benefits of the GNN, ablation studies comparing its performance with simpler models like feed-forward neural networks or random forest regressors are key. We could input the raw sensor data to simpler models and compare their system’s resulting reward with the GNN-based approach. Such a comparison would allow us to assess whether the graph structure and representation offer inherent advantages to the system. By addressing these limitations and conducting further research, the proposed self-repairing embedded system can be validated for real-world applications.

References

- [1] Johnson Space Center. [n. d.]. Fault-Detection, Fault-Isolation and Recovery (FDIR) Techniques. ([n. d.]).
- [2] Siddharth Choudhuri and Tony Givargis. 2009. FlashBox: a system for logging non-deterministic events in deployed embedded systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 1676–1682.
- [3] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. 2006. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems* 2006 (2006), 1–19.
- [4] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2018).
- [5] Daniele Grattarola, Daniele Zamboni, Filippo Maria Bianchi, and Cesare Alippi. 2022. Understanding pooling in graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2022).
- [6] Colin Greatwood and Arthur G Richards. 2019. Reinforcement learning and model predictive control for robust embedded quadrotor guidance and control. *Autonomous Robots* 43 (2019), 1681–1693.
- [7] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, Andreas Herkersdorf, Rüdiger Kapitza, Daniel Lohmann, Peter Marwedel, Marco Platzner, Wolfgang Rosenstiel, Ulf Schlichtmann, Olaf Spinczyk, Mehdi Tahoori, Jürgen Teich, Norbert Wehn, and Hans-Joachim Wunderlich. 2011. Design and Architectures for Dependable Embedded Systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (Taipei, Taiwan) (CODES+ISSS '11)*. Association for Computing Machinery, New York, NY, USA, 69–78. <https://doi.org/10.1145/2039370.2039384>
- [8] Lucas Koch, Mario Picerno, Kevin Badalian, Sung-Yong Lee, and Jakob Andert. 2023. Automated function development for emission control with deep reinforcement learning. *Engineering Applications of Artificial Intelligence* 117 (2023), 105477. <https://doi.org/10.1016/j.engappai.2022.105477>
- [9] Roy A Maxion and Kymie MC Tan. 2002. Anomaly detection in embedded systems. *IEEE Trans. Comput.* 51, 2 (2002), 108–120.
- [10] Aleix Megías Homar. 2014. AOCS Fault Detection, Isolation and Recovery: A Model-Based Dynamic Verification and Validation Approach.
- [11] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. 2004. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 3 (2004), 461–491.
- [12] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. 2017. Deep reinforcement learning framework for autonomous driving. *arXiv preprint arXiv:1704.02532* (2017).
- [13] Guto Leoni Santos, Patricia Takako Endo, Theo Lynn, Djamel Sadok, and Judith Kelner. 2022. A reinforcement learning-based approach for availability-aware service function chain placement in large-scale networks. *Future Generation Computer Systems* 136 (2022), 93–109. <https://doi.org/10.1016/j.future.2022.05.021>
- [14] Nanliang Shan, Xinghua Xu, Xianqiang Bao, and Shaohua Qiu. 2022. Fast Fault Diagnosis in Industrial Embedded Systems Based on Compressed Sensing and Deep Kernel Extreme Learning Machines. *Sensors* 22, 11 (2022), 3997.
- [15] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
- [16] Bart Vermeulen. 2008. Functional Debug Techniques for Embedded Systems. *IEEE Design Test of Computers* 25, 3 (2008), 208–215. <https://doi.org/10.1109/MDT.2008.66>
- [17] Zhe Wang and Tianzhen Hong. 2020. Reinforcement learning for building controls: The opportunities and challenges. *Applied Energy* 269 (2020), 115036. <https://doi.org/10.1016/j.apenergy.2020.115036>
- [18] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (2019), 1–23.
- [19] Tianyi Zhang, Minjun Seo, Bryan Donyanavard, Nikil Dutt, and Fadi Kurdahi. 2020. Predicting Failures in Embedded Systems Using Long Short-Term Inference. *IEEE Embedded Systems Letters* 13, 3 (2020), 85–89.