TNO innovation for life

Reinforcement Learning for Intelligent Diagnostics

TNO Public ⟩ TNO 2024 S12781
11 September 2024

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2024 S12781 – 11 September 2024

Reinforcement Learning for Intelligent Diagnostics

Author(s)	Noah Farr
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	24 (excl. front and back cover)
Number of appendices	0

Contents

Table of Contents

No table of contents entries found.

1 Introduction

Efficient and accurate diagnostics is crucial for minimizing downtime of complex cyber-physical systems. However, locating the root cause of failures in such systems is challenging due to their scale, complexity, and the potential for fault propagation across subsystems. This research proposes a novel methodology that integrates elements of the SD2Act diagnostic approach[1] with reinforcement learning (RL) techniques to optimize and guide the diagnostic process for large-scale systems.

RL is a subfield of machine learning where an autonomous agent learns through interaction with its environment, receiving rewards or penalties for the actions it takes with the ultimate goal of maximizing its cumulative reward over time. RL has shown great success in sequential decision making problems ranging from robotics and autonomous vehicles to game playing and recommendation systems. By modeling the diagnosis task as a sequential decision process and applying RL algorithms, diagnostic agents can learn efficient troubleshooting strategies that balance the value of information gained against the cost of performing different diagnostic actions.

The SD2Act approach leveraged in this work provides a systematic methodology to transform system design information, such as functional hierarchies and component dependencies, into a unified diagnostic network. This network takes the form of a graph representation capturing the key components, functions, and relationships that govern fault propagation in the system. The diagnostic network serves as the basis for automatically generating a probabilistic reasoning model, such as a Bayesian network, which the RL agent uses to iteratively infer the most probable fault hypotheses and recommend optimal next steps for a service engineer to efficiently isolate the root cause.

Integrating RL with the SD2Act framework enables learning customized diagnostic policies that minimize the expected total cost of diagnosing a specific system. By training the RL agent on historical maintenance data or simulations of the target system, it can discover optimal decision strategies that account for component reliabilities, failure modes, diagnostic action costs, and other system-specific factors. This approach takes a step towards intelligent, adaptive diagnostic assistants that can improve their troubleshooting efficiency over time by learning from experience.

The remainder of this report details the proposed methodology, covering the core steps of diagnostic network creation, reasoning model generation, and RL-based diagnostic policy optimization. Implementation details and results of initial proof-of-concept experiments are presented to demonstrate the feasibility and potential of this approach for enhancing complex system diagnostics. The long-term vision is to develop practical tools that enable faster, more economical diagnosis and maintenance of critical cyber-physical systems.

[1] TNO 2025 R10010 *Guided diagnosis of functional failures in cyberphysical systems*

2 Problem Definition

In today's increasingly sophisticated and interconnected high-tech systems, such as advanced manufacturing equipment, aerospace systems, and data centre infrastructures, diagnosing and resolving issues efficiently is a critical challenge. The complexity of these systems, often comprising hundreds or thousands of interrelated hardware and software components, makes pinpointing the root cause of a malfunction a difficult task.

When a system experiences a failure or performance degradation, there are typically numerous potential causes, ranging from faulty hardware components to software bugs, misconfigurations, or environmental factors. Exhaustively testing every component and subsystem can be extremely time-consuming and costly, leading to longer downtimes, reduced productivity, and customer dissatisfaction.

2.1 Problem in the field

The key problem lies in determining the optimal sequence of diagnostic tests to identify the root cause as quickly and efficiently (and therefore, cheaply) as possible, given the current system state and the results of diagnostic tests performed so far. Field service engineers and technicians often rely on their experience, intuition, and trial-and-error to guide their troubleshooting efforts. However, the growing complexity of modern systems makes it increasingly difficult for human experts to keep pace.

2.2 Diagnostic Problem

In the methodology proposed in the SD2Act project, a simple weighted sum approach is used to select the next test, considering both the cost and information gain associated with each available test. The information gain represents the expected reduction in uncertainty about the root cause based on the test outcome, while the cost reflects the time, resources, and potential risks involved in conducting the test. A fixed weight is assigned to balance the trade-off between cost and information gain, guiding the diagnoser to choose tests that provide the most valuable information at the lowest cost.

2.3 Reinforcement Learning Problem

This challenge can be formulated as a reinforcement learning (RL) problem, where an RL agent learns a policy to select the best next diagnostic test based on the current state of the system and the outcomes of previous tests. The agent's goal is to minimize the total time and cost required to identify the root cause and resolve the issue.

Developing an effective AI-assisted diagnostic solution involves several key challenges:

1. **State Representation**: Defining a suitable representation of the system state that captures the relevant information for decision-making, such as the current operational status of components, error codes, performance metrics, and previous test results.

2. **Action Space**: Defining the set of available diagnostic tests and actions that the agent can choose from, considering factors such as test coverage, execution time, cost, and potential risks.

3. **Reward Function**: Designing an appropriate reward function that incentivizes the agent to identify the root cause quickly and efficiently, balancing the trade-offs between test costs, time, and accuracy.

The goal of this project is to develop and train an RL-based diagnostic agent that can demonstrate the potential for AI-assisted root cause analysis in complex high-tech systems. By learning to intelligently select and sequence diagnostic tests based on the current system state and previous test results, the agent aims to reduce the time and cost associated with troubleshooting, ultimately leading to improved system availability, productivity, and customer satisfaction. The successful implementation of such an agent could pave the way for more advanced AI-driven diagnostic and maintenance solutions in various industrial domains.

3 Background

3.1 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on how agents can learn to make sequential decisions through interaction with an environment. Unlike supervised learning, where an algorithm learns from labelled examples, or unsupervised learning, where it finds patterns in unlabelled data, RL is centered around the concept of an agent learning from its own experiences and the consequences of its actions.

The core idea of RL is inspired by behavioural psychology, particularly the way humans and animals learn through trial and error. In an RL framework, an agent is placed in an environment where it must learn to perform a task or achieve a goal. The agent is not explicitly told which actions to take, but instead must discover which actions yield the most reward by trying them out.

The RL process can be broken down into several key components:

1. Agent: The learner or decision-maker that interacts with the environment.
2. Environment: The world in which the agent operates and learns.
3. State: A representation of the current situation in the environment.
4. Action: A decision made by the agent that changes the state of the environment.
5. Reward: A feedback signal that indicates the desirability of the state resulting from an action.
6. Policy: The strategy that the agent employs to determine its actions.

The learning process in RL is cyclical:

1. The agent observes the current state of the environment.
2. Based on this state, the agent chooses an action according to its policy.
3. The environment transitions to a new state as a result of this action.
4. The agent receives a reward (or penalty) based on the outcome of its action.
5. The agent uses this experience to update its policy, aiming to make better decisions in the future.

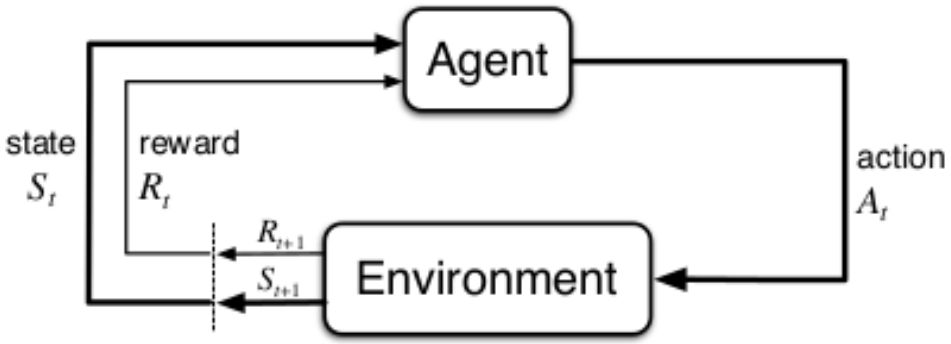

Figure 1: Visualization of the interaction steps between the agent and the environment.

6. This cycle continues, with the agent progressively improving its policy to maximize its cumulative reward over time. The ultimate goal of RL is for the agent to learn an optimal policy that consistently chooses actions that lead to the highest long-term reward. The learning is stopped when the user considers than further training will not meaningfully improve the cumulative reward.

3.2 Markov Decision Process

Reinforcement learning problems are often formalized using the framework of Markov Decision Processes (MDPs).
An MDP provides a rigorous mathematical foundation for modelling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. It is an effective way of representing and solving RL problems because it captures the essential elements of the task while making simplifying assumptions that make the problem tractable, allowing for the development of algorithms with provable convergence properties and performance guarantees. MDPs are the basis of many classic and modern RL algorithms, including value iteration, policy iteration, Q-learning, and their deep learning-based counterparts.

An MDP is defined by a tuple (S, A, P, R, γ), where each component has a specific meaning and role:

1. S - State space: This is the set of all possible situations or configurations in which the agent might find itself. In a chess game, for example, the state would be the current configuration of pieces on the board. In a more complex scenario like autonomous driving, the state might include the car's position, speed, direction, and information about nearby vehicles and obstacles.
2. A - Action space: This is the set of all possible actions the agent can take. Actions cause transitions between states and can be discrete (like moving a chess piece) or continuous (like adjusting the steering angle of a car).
3. P - Transition probability function: This function, often denoted as $P(s'|s, a)$, defines the dynamics of the environment. It gives the probability of transitioning to state s' when action a is taken in state s. This probabilistic nature allows MDPs to model environments with inherent uncertainty or stochasticity.
4. R - Reward function: Denoted as $R(s, a)$, this function defines the immediate reward (or penalty) the agent receives for taking action a in state s. The reward function is crucial as it encodes the goal of the task and guides the learning process. Designing an appropriate reward function is often one of the most challenging aspects of applying RL to real-world problems.
5. γ - Discount factor: This parameter, ranging from 0 to 1, determines the importance of future rewards. A discount factor close to 0 makes the agent "myopic", focusing almost entirely on immediate rewards, while a factor close to 1 makes it "far-sighted," valuing future rewards almost as much as immediate ones. The discount factor helps manage the trade-off between short-term and long-term rewards and ensures that the cumulative reward remains finite in tasks with no clear endpoint.

The Markov property, which gives MDPs their name, is a crucial assumption in this framework. It states that the future depends only on the current state and action, not on the history of previous states and actions. While this might seem limiting, many complex

problems can be modelled effectively within this framework by carefully designing the state representation to include relevant historical information.

3.3 Agent

3.3.1 Policy

In the context of Reinforcement Learning, a policy is the core component that defines an agent's behaviour. It is essentially the strategy or set of rules that the agent follows to make decisions in any given state. More formally, a policy is a function that maps states to actions, determining which action the agent should take when it finds itself in a particular state.

There are two main types of policies in RL:

1. Deterministic policy: In this case, the policy is a function π(s) that maps each state s to a specific action a. For any given state, a deterministic policy will always choose the same action. This type of policy is straightforward and can be optimal in fully observable environments with deterministic dynamics. However, it may struggle in stochastic environments or situations where exploration is crucial.
2. Stochastic policy: A stochastic policy, denoted as $\pi(a|s)$, defines a probability distribution over actions for each state. Instead of always selecting the same action for a given state, a stochastic policy assigns probabilities to each possible action. The action is then chosen by sampling from this distribution. Stochastic policies are more flexible and can be beneficial in partially observable environments by helping to manage uncertainty, competitive scenarios, or situations where exploration is important by using randomness to encourage exploration of the state space.

The choice between deterministic and stochastic policies often depends on the nature of the problem and the characteristics of the environment.

Regardless of the type, the ultimate goal in RL is to find an optimal policy π* that maximizes the expected cumulative reward over time. This is typically achieved through an iterative process of policy evaluation and policy improvement:

1. Policy evaluation: This step assesses the value of the current policy by estimating the expected cumulative reward from each state when following that policy.
2. Policy improvement: Based on the evaluation, the policy is updated to increase the probability of taking actions that lead to higher expected rewards.

These steps are repeated until the policy converges to an optimal or near-optimal solution. Various RL algorithms implement this process in different ways:

- Value-based methods (like Q-learning) implicitly represent the policy by learning a value function and selecting actions that maximize this value.
- Policy-based methods directly optimize the policy, often using gradient-based approaches to adjust the parameters of a parameterized policy.
- Actor-critic methods combine both approaches, using a value function (the critic) to guide updates to an explicit policy representation (the actor).

In practical applications, policies are often represented using function approximators such as neural networks, especially when dealing with large or continuous state spaces. These neural network policies can be trained using techniques like policy gradient methods or evolutionary strategies.

For diagnostic testing applications, the policy guides the selection of diagnostic tests based on the current knowledge state. The goal is to efficiently diagnose the system while minimizing costs, such as the number of tests performed or the time taken to reach a diagnosis. The policy in this context might be represented as a decision tree, a set of rules, or a neural network that takes the current state (e.g., observed symptoms, test results) as input and outputs the next most informative test to perform.

3.3.2 Value Function

Value functions are fundamental concepts in reinforcement learning that the expected return associated with being in a specific state or performing a particular action within a state. They provide a way to evaluate the long-term desirability of states and actions, taking into account not just immediate rewards but also the potential for future rewards. There are two main types of value functions:

1. State-value function V(s):
   The state-value function V(s) represents the expected cumulative reward an agent can obtain starting from state s and following a particular policy π thereafter. Mathematically, it's defined as:

$$V_\pi(s) = E_\pi[\sum \gamma^t R_t | S_0 = s]$$

where

- E_π denotes the expected value when following policy π
- Σ represents the sum over time steps t = 0 to infinity
- γ is the discount factor ($0 \leq \gamma \leq 1$)
- R_t is the reward received t time steps into the future
- S_0 is the current state

The state-value function allows the agent to assess the long-term value of being in different states, which is crucial for making informed decisions.

2. Action-value function $Q(s, a)$:
   Also known as the Q-function, the action-value function $Q(s, a)$ represents the expected cumulative reward an agent can obtain by taking action a in state s and then following policy π thereafter. It's defined as:

$$Q_\pi(s, a) = E_\pi[\sum \gamma^t R_t | S_0 = s, A_0 = a]$$

   The Q-function provides more detailed information than the V-function, as it quantifies the value of each action in a given state. This makes it particularly useful for action selection and policy improvement.

Learning value functions is a key challenge in RL. Understanding and effectively estimating value functions is crucial for developing efficient RL algorithms and solving complex decision-making problems. In simple environments with small, discrete state spaces, value functions can be represented as tables and learned through direct experience. However, in more complex environments with large or continuous state spaces, function approximation techniques (often involving neural networks) are used to estimate value functions.

3.3.3 Deep Q-Networks

Introduced by DeepMind in 2013 and further refined in subsequent years, the DQN algorithm has become a cornerstone of modern RL, capable of learning effective policies in complex environments with high-dimensional state spaces.

The core idea behind DQN is to use a deep neural network to approximate the action-value function $Q(s, a)$. This approach offers several advantages:

1. Handling high-dimensional state spaces: Traditional tabular Q-learning becomes intractable in environments with large or continuous state spaces. Neural networks can effectively generalize across similar states, allowing DQN to handle complex, high-dimensional input data such as images.
2. Feature learning: The neural network learns to extract relevant features from raw input data, eliminating the need for manual feature engineering.
3. Scalability: a DQN can be applied to a wide range of problems without significant modifications to the core algorithm.

The DQN algorithm incorporates several key innovations to stabilize learning and improve performance:

1. Experience replay: DQN stores the agent's experiences (state transitions, actions, rewards) in a replay buffer. During training, random batches of experiences are sampled from this buffer. This technique breaks the correlation between consecutive samples and allows the network to learn from a more diverse and representative set of experiences.
2. Target network: DQN uses two neural networks; a main network for selecting actions and updating Q-values, and a separate target network for generating target Q-values. The target network is periodically updated with the weights of the main network. This helps stabilize learning by reducing the moving target problem that arises from trying to optimize two co-varying parameters, the Q-values and actions, at the same time.
3. ε-greedy exploration: To balance exploration and exploitation, DQN typically uses an ε-greedy policy. With probability ε, the agent chooses a random action, and with probability $1 - \varepsilon$, it chooses the action with the highest predicted Q-value. The value of ε is often annealed over time, gradually shifting from exploration to exploitation.

The training process for DQN involves the following steps:

1. Initialize the main Q-network and target Q-network with random weights.
2. For each episode:
    a. Reset the environment and get the initial state.
    b. For each time step:
        i. Choose an action using the ε-greedy policy based on the main Q-network.
        ii. Execute the action and observe the reward and next state.
        iii. Store the experience (s, a, r, s') in the replay buffer.
        iv. Sample a random batch of experiences from the replay buffer.
        v. Compute the target Q-values using the target network.
        vi. Update the main Q-network by minimizing the loss between predicted and target Q-values.

c. Periodically update the target network weights with the main network weights.

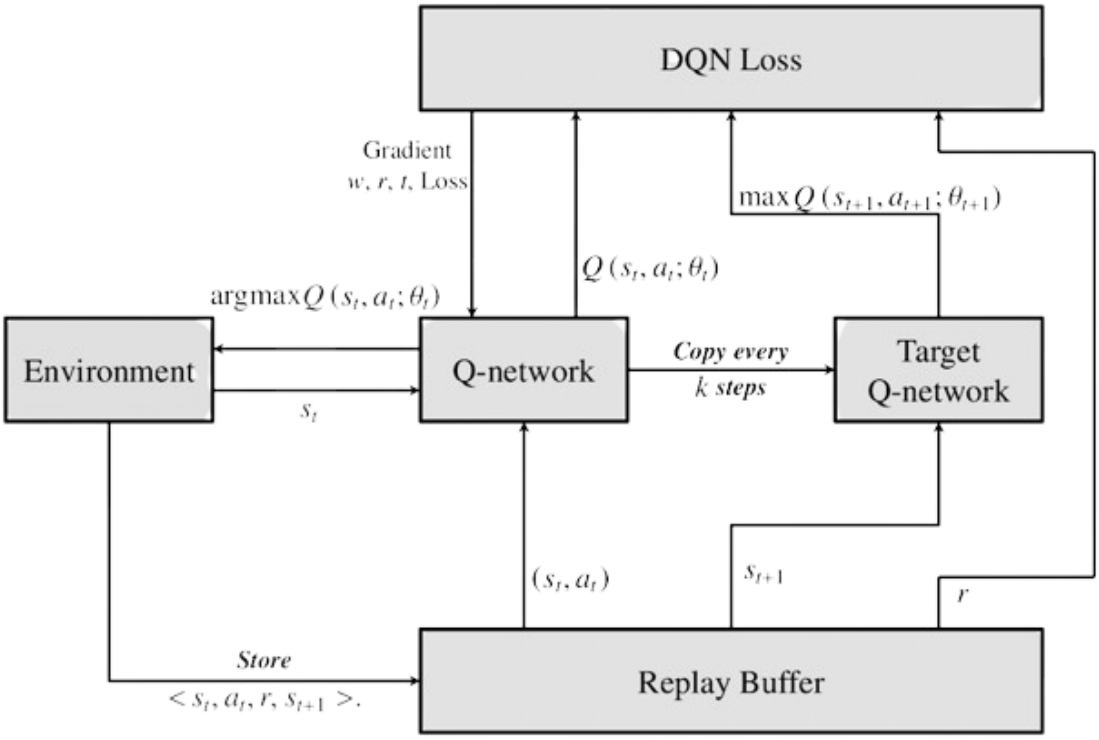

Figure 2: Visualization of the DQN algorithm. Observe how this is a more complex view of the basic diagram depicted in figure 1 where the internal structure of the agent is fully detailed.

Several extensions and improvements to the original DQN algorithm have been proposed. We used the following extensions:

1. Double DQN: Addresses the overestimation bias in Q-learning by using the main network for action selection and the target network for value estimation.
2. Prioritized experience replay: Assigns priorities to experiences in the replay buffer based on their TD error (i.e. the absolute difference between what the agent is estimating and what the true value is), allowing the agent to learn more effectively from important or rare experiences.

Despite its successes, DQN has some limitations. It is primarily designed for discrete action spaces and may struggle in environments with continuous actions. Additionally, as a value-based method, it can be less sample-efficient than some policy-based or actor-critic methods. Nonetheless, DQN remains a powerful and widely-used algorithm in the RL toolkit, serving as a foundation for many advanced deep RL techniques.

4 Methodology

The methodology employed in this research combines elements from the SD2Act project's diagnostic approach with reinforcement learning techniques to optimize the diagnostic process. The core steps are as follows:

1. *Diagnostic Network Creation*: System design information is transformed into a diagnostic network, which is a graph representation with clearly defined types of nodes (e.g., hardware, functions, observables) and relations between nodes. This transformation can be done automatically if the design is documented in a structured manner, such as in a model-based systems engineering (MBSE) approach, or manually using tools like the Excel template presented in the SD2Act report (pending publication).
2. *Reasoning Model Generation*: A reasoning model is generated from the diagnostic network. Both Bayesian Networks and Tensor Networks are evaluated as potential reasoning formalisms. The reasoning model takes system observations and human input to iteratively suggest the next diagnostic test for a service engineer, guiding them efficiently to identify the fault.
3. *Diagnostic Test Recommendation*: The reasoning model incorporates both the expected information gain and the cost of performing diagnostic tests when recommending the next action. A loss function is defined to balance these two factors, aiming to minimize the total diagnostic costs by reducing the number of steps and the cost per action. The balancing of information gain and cost is a key challenge.
4. *Reinforcement Learning for Optimization*: To optimize the balance between information gain and cost in the diagnostic process, reinforcement learning techniques are employed. The diagnostic process is modelled as a Markov Decision Process (MDP), where the state represents the current knowledge about the system's state, actions correspond to diagnostic tests, and rewards are based on the reduction in diagnostic uncertainty and associated costs.
5. *Policy Learning*: Using the defined MDP, a reinforcement learning algorithm, such as DQN, is used to learn an optimal policy for selecting diagnostic actions. The policy aims to maximize the expected cumulative reward, effectively minimizing the total diagnostic costs over multiple episodes.

By combining the structured diagnostic modelling approach from SD2Act with reinforcement learning for optimization, this methodology aims to create an intelligent and adaptive diagnostic system that can effectively guide service engineers while minimizing overall diagnostic costs. The integration of reinforcement learning allows the system to learn from experience and continuously improve its decision-making capabilities.

One very important observation we made was that the agent learns a lot better when there are no root causes that are not diagnosable. Its critical to ensure that for every root cause once all tests are executed the correct root cause will be uniquely identified. If there are root causes that are not uniquely identifiable one should remove those during the training of the agent to ensure better performance. Alternatively, one should develop finer stopping criteria that can identify when a diagnostic procedure has concluded.

4.1 Environment

4.1.1 Observation Space

The observation space of an RL environment describes what information is given to the agent i.e. what to use as input to the neural network that's learning the policy. The observation space should contain all information that's necessary to learn a good policy.

Test Results	Costs	Information Gains	OK	NOK	Healthy	Broken
result_0	cost_0	info_gain_0	ok_0	nok_0	healthy_0	broken_0
result_1	cost_1	info_gain_1	ok_1	nok_1	healthy_1	broken_1
result_2	cost_2	info_gain_2	ok_2	nok_2	healthy_2	broken_2
result_3	cost_3	info_gain_3	ok_3	nok_3		
result_4	cost_4	info_gain_4	ok_4	nok_4		

Figure 3: Representation of the observation space. Each column represents one part of the observation space of a diagnostic environment

4.1.1.1 Result of executed diagnostic tests

The state includes the outcomes of all diagnostic tests that have been performed up to the current point in the diagnostic process. This information helps track the progress and narrow down the potential causes of the fault based on the available evidence.

4.1.1.2 Cost of each diagnostic test

The cost associated with each diagnostic test is incorporated into the state representation. This cost can be monetary, time-based, or a combination of factors, depending on the specific diagnostic scenario. Including the cost in the state allows the reinforcement learning algorithm to consider it when making decisions.

4.1.1.3 Information gain of each diagnostic test

The expected information gain of each diagnostic test is included in the state. Information gain represents the reduction in uncertainty about the system's fault state that a particular test can provide. By considering the information gain, the algorithm can prioritize tests that are most likely to yield valuable insights.

4.1.1.4 Success/Failure probability of each diagnostic test

The state includes in columns 4 and 5 the conditional probability of each diagnostic test succeeding or failing based on the currently available evidence. These probabilities are derived from the reasoning model and are updated as new evidence is obtained. Including these probabilities in the state allows the algorithm to make informed decisions based on the likelihood of test outcomes.

4.1.1.5 Healthy/Broken probability of each hardware node

The state also incorporates the probability of each hardware node being healthy or broken, given the current evidence. These probabilities are obtained from the reasoning model and reflect the current belief about the health status of each component. Including these probabilities in the state enables the algorithm to focus on the most likely faulty components.

4.1.2 Action Space

The action space of an RL environment contains the actions available to the agent i.e. the output of the neural network that's then used to take a step in the environment.
The action space of our MDP consists of one entry for each test that can be executed. Each action corresponds to performing a specific diagnostic test. The algorithm selects an action based on the current state and the learned policy, aiming to maximize the expected cumulative reward.

action	diagnostic_test_0	diagnostic_test_1	diagnostic_test_2	diagnostic_test_3	diagnostic_test_4
	0	1	2	3	4

Figure 4: In a discrete action space, each action is assigned a number. The action recommended by the agent is the one corresponding to the number output by the agent's neural network.

4.1.2.1 Action Masking

Our action space includes all existing diagnostic tests, but it does not make sense to include tests that already have been executed. To improve learning speed we can mask the action space to include only the diagnostic tests that have not yet been executed.

4.1.3 Termination and Truncation

An episode terminates once one hardware node has a probability of being broken greater than 85% or once all tests are executed. The 85% threshold is chosen to balance the confidence in identifying the faulty component with the potential cost of additional tests. If all tests are executed without reaching the threshold, the episode ends, and the component with the highest probability of being broken is considered the most likely cause of the fault. By choosing the termination condition in this way we incentivize the agent to choose actions that have a higher information gain as that leads to faster termination.

4.1.4 Reward Function

The reward in our MDP is defined as the negative cost of the executed test. By assigning a negative reward, the algorithm is incentivized to minimize the total cost of the diagnostic process. The goal is to find the optimal policy that maximizes the expected cumulative reward, effectively minimizing the overall diagnostic costs.

4.2 Diagnostic System Generator

To evaluate the RL agent on systems of arbitrary size, we developed a generator that generates random diagnostic systems of desired size. This generator allows us to create synthetic diagnostic systems with varying complexity and characteristics, enabling a

thorough evaluation of the agent's performance across a wide range of scenarios. The generated models follow the methodology outlined in the MBDLyb diagnostics library[2].

Generating the network topology: Expander graphs.

The generator begins by creating an expander graph with n nodes, where n is the desired size of the diagnostic system. Each node in this graph represents a function within the system. Expander graphs are chosen for their connectivity properties, because they ensure good connectivity (every node is just a few hops away from any other node) while keeping the graph almost tree-like. Mathematically, expander graphs have high vertex expansion, meaning that every subset of nodes has a large number of neighbours while having bounded degree, leading to good mixing properties and fast information propagation while maintaining sparsity. This property is crucial for modelling realistic diagnostic systems where clusters are interconnected but functions are locally influenced by just a few other nodes, largely within their cluster and following a clear causal structure.

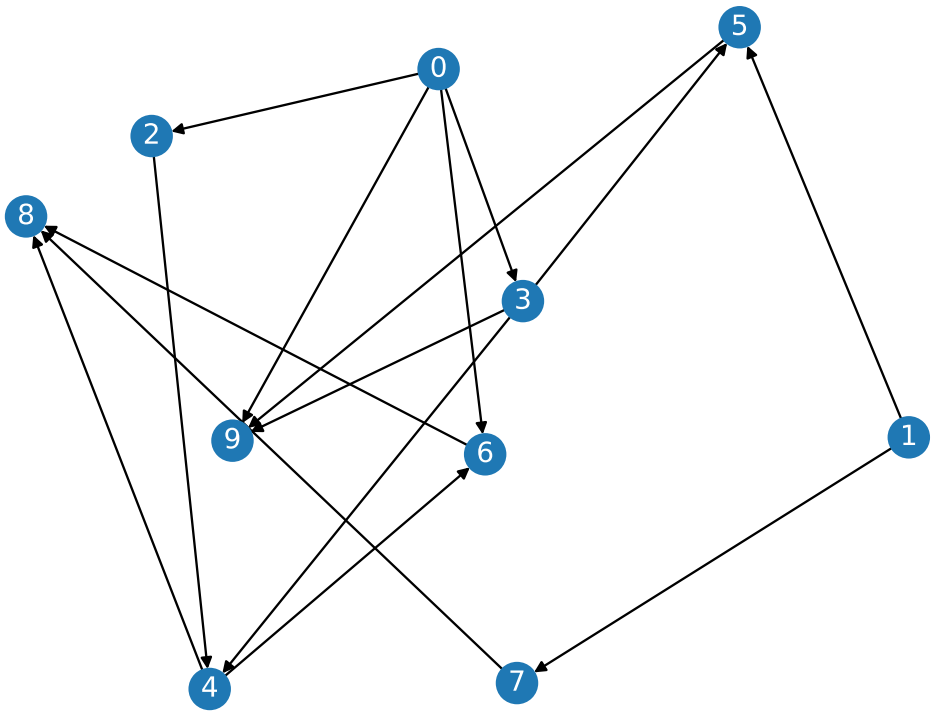

Figure 5: Example topology of the function nodes in the generated model.

4.2.1.1.1 *Enhancing the graph*

Once the expander graph is generated, we proceed to add hardware nodes to each function node. These hardware nodes represent the physical components associated with each function in the diagnostic system.

[2] [2] TNO 2025 R10010 *Guided diagnosis of functional failures in cyberphysical systems*

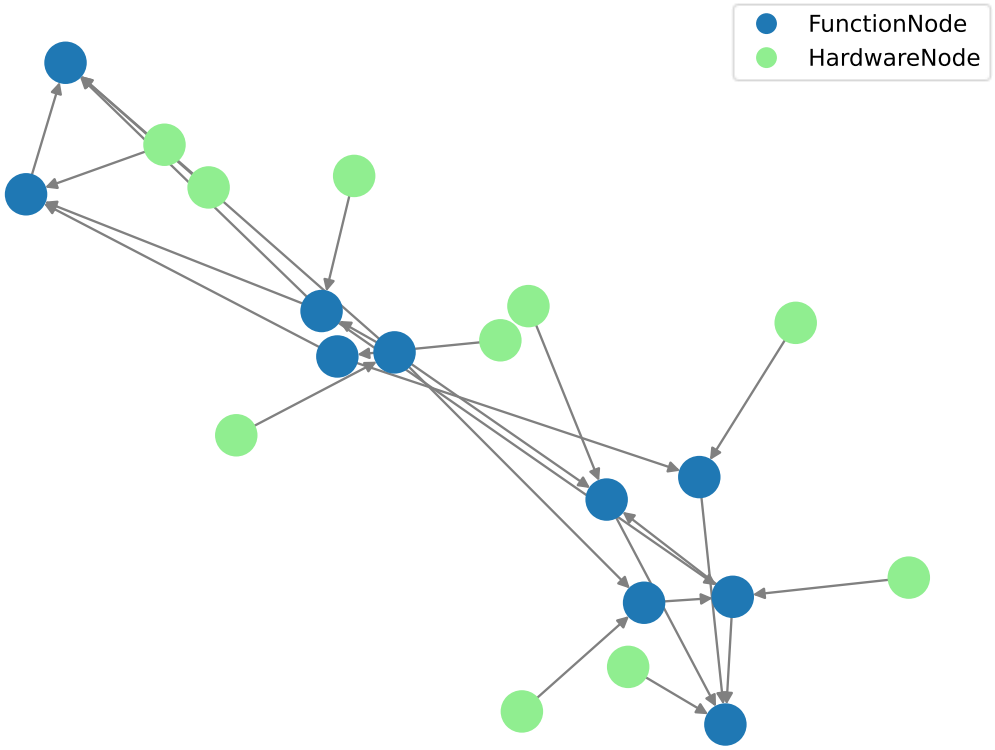

Figure 6: The model from figure 5 expanded with hardware and function nodes. Each function is assumed to be realized by a unique hardware piece.

For each function node, we then randomly assign either a diagnostic test or a direct observable. With an 80% probability, a diagnostic test is added to the function node. Diagnostic tests are procedures or methods used to assess the health or functionality of the associated hardware component. They provide valuable information for troubleshooting and identifying potential issues within the system. It is possible for two or more function nodes to be tested by a single diagnostic test.
In the remaining 20% of cases, a direct observable is assigned to the hardware node instead of a diagnostic test. Direct observables are measurable or observable characteristics of the system that can be directly monitored or inspected without the need for explicit testing. These observables serve as indicators of the component's state or performance.

4.2.1.1.2

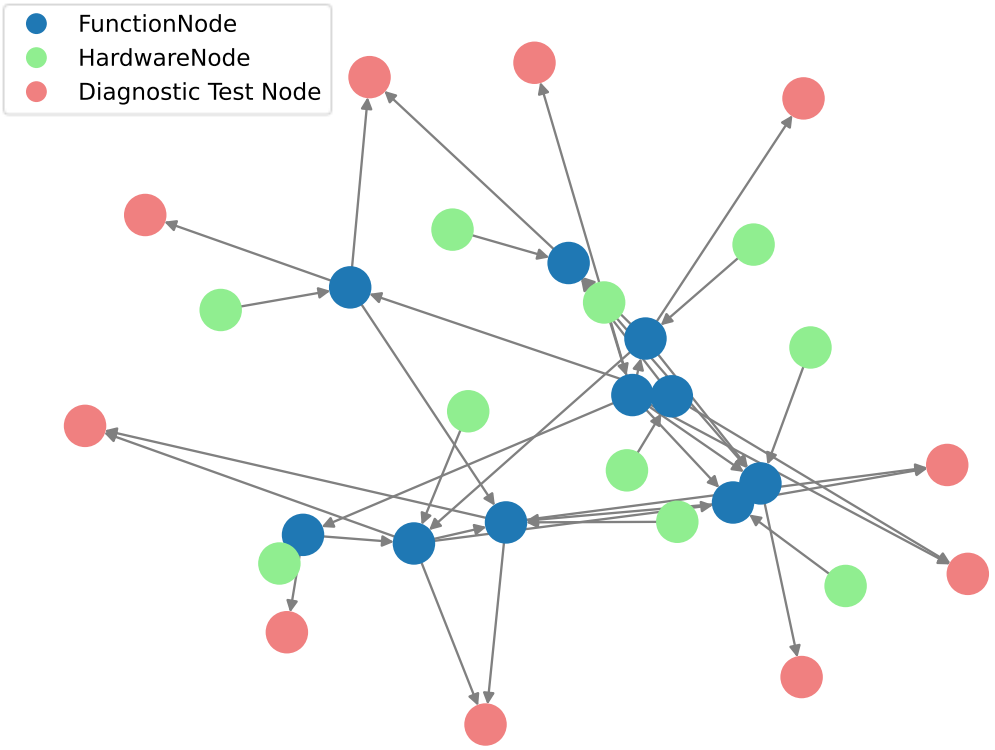

Figure 7: Fully realized diagnostic model. Observe how each hardware node has associated a unique function node, but the same is not true for diagnostic tests.

4.2.1.1.3 Generation of costs

In the last step, we add costs to the diagnostic test nodes. We use a Log-normal distribution to generate these costs, with a mean of 3 and a variance of 0.5. This distribution is particularly useful because it gives us a good spread of values without too many extremes, while still allowing for some higher-cost outliers. Other distributions, namely the Normal, Zipf and Gamma distributions, were tried but yielded either too flat or too peaked histograms. The advantage of the Log-normal distribution is that the orders of magnitude are distributed normally, leading to exponentially decaying probabilities for values that are several orders of magnitude bigger or smaller than the mode, but relatively flat probabilities for values that are just one or even two orders of magnitude above or below the mode. Then we assign the costs to the diagnostic tests based on the centrality of the respective observable node in the graph. This ensures that more central tests in the graph have higher costs than those positioned farther from the center.

4.2.1.1.4 Export to MBDIyb

After generating the diagnostic system structure, we perform a series of transformations to export the system as an Excel file. This process involves organizing the system's data, including the functions, hardware nodes, diagnostic tests, and direct observables, into a structured format that can be stored and accessed using MBDIyb.
The exported Excel model serves as a comprehensive representation of the generated diagnostic system. It captures the hierarchical relationships between functions, hardware

components, and their associated diagnostic tests or observables. This Excel file can then be loaded using MBDlyb.

			Functions							Observables	
			CDRadioPlayer		PowerSystem	CDReader	RadioReceiver	AudioSystem	Direct	Direct	DiagnosticTest
Cluster	**Type**	**Name**	PlayCD	PlayRadio	ProvidePower	ReadCD	ReceiveRadio	GenerateAudio	No_CD	No_Radio_Signal	ListenAudio
PowerSystem	Function	ProvidePower									
PowerSystem	Hardware	PowerSystem									
CDReader	Function	ReadCD									
CDReader	Hardware	CDReader									
RadioReceiver	Function	ReceiveRadio									
RadioReceiver	Hardware	RadioReceiver									
AudioSystem	Function	GenerateAudio									
AudioSystem	Hardware	AudioSystem									
Cost		Time									5

Figure 8: Example of an excel sheet that can be parsed with MBDlyb

The diagnostic system generator provides a flexible and scalable approach to creating diverse test cases for the RL agent. By generating systems of varying sizes and complexity, we can assess the agent's ability to handle a wide range of diagnostic scenarios. This comprehensive evaluation helps ensure the robustness and generalization capabilities of the agent, making it more suitable for real-world applications.

5 Results

To evaluate our methodology we trained the RL agent on multiple systems with varying number of nodes and edges.

5.1 CD Radio Player

To begin our experiment, we initially trained the agent using the CDRadioPlayer diagnostic model, which is designed to simulate fault diagnosis in complex hardware systems. The CDRadioPlayer model consists of 19 distinct hardware nodes, each representing a different component or subsystem of a CD radio player. Additionally, the model includes 12 diagnostic tests that can be performed to detect faults in the system. However, out of the 19 hardware nodes, 11 were found to be non-uniquely diagnosable, meaning that the faults in these nodes could not be distinguished from other hardware nodes using the available diagnostic tests. This is an issue of the model, regardless of diagnostic strategy. As a result, these non-uniquely diagnosable nodes were excluded from the training process to streamline the model and ensure that the agent focused on more actionable data.

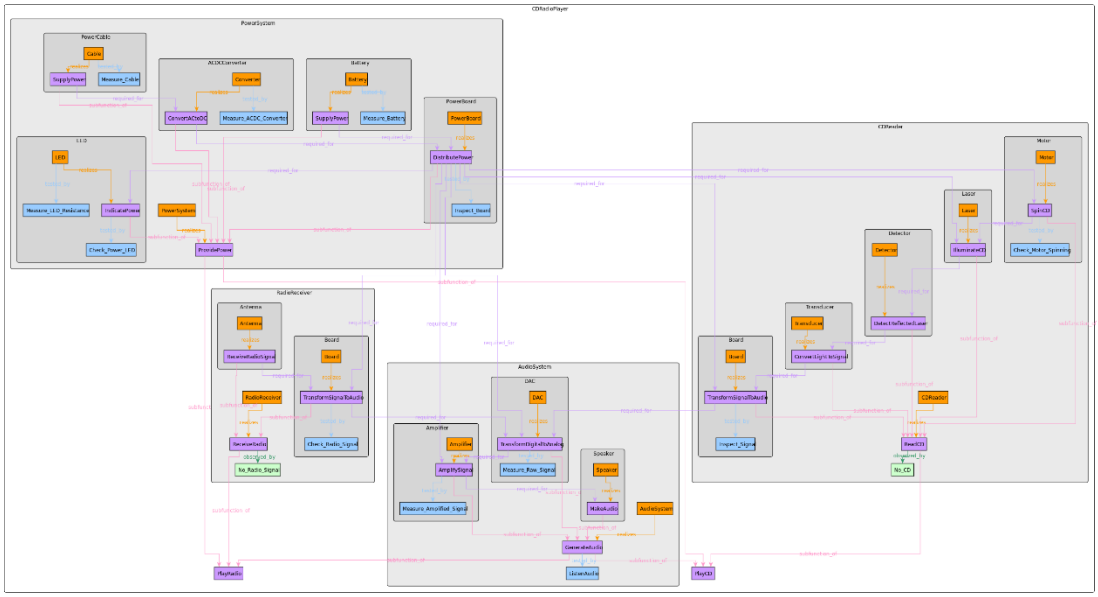

Figure 9: Diagnostic network corresponding to the CDRadioPlayer system. A detailed discussion of this system can be found in the MBDIyb report.

During the training phase, the agent was tasked with minimizing the diagnostic cost, s. In this particular case, this is equivalent to the number of tests since all tests are given a cost of 1. After the training process was complete, the agent demonstrated its ability to outperform the baseline approach on the CDRadioPlayer model, although not by a large amount. Specifically, the agent achieved a mean diagnostic cost of -17.625. For comparison, the baseline approach, which uses the best fixed weighting scheme (with weight 0.95 on entropy and 0.05 on cost for test selection),s resulted in a higher total diagnostic cost of -

18.75. This indicates that the agent was more efficient in diagnosing faults, utilizing fewer tests and reducing overall diagnostic costs in comparison to the baseline.

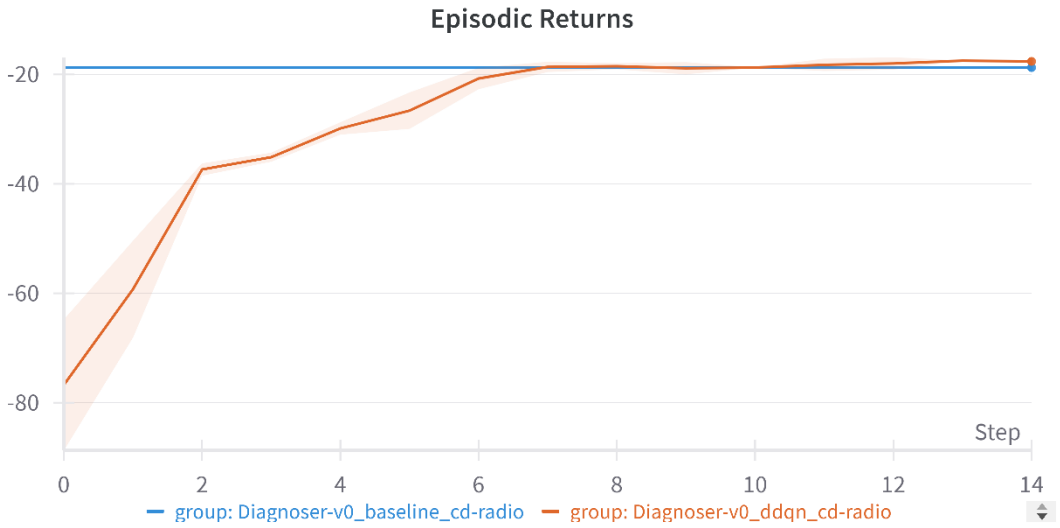

Figure 10: Orange line indicates negative cost of diagnosis averaged over root causes as a function of the number of training steps of the RL agent (in thousands). Blue line indicates the (negative) cost of the best fixed policy based on balancing entropy and cost.

5.2 Diagnostic System Generator

Following the initial training on the CDRadioPlayer model, we extended the complexity of the task by training the agent on a more challenging, randomly generated diagnostic model. This new model, created using the diagnostic system generator, consisted of 30 hardware nodes and 30 diagnostic tests, significantly increasing the difficulty due to the larger number of components and possible fault scenarios. The increased number of hardware nodes and diagnostic tests introduced more variability and complexity, making it harder for the agent to find optimal diagnostic strategies.

As a result of this added complexity, the agent was unable to surpass the performance of the baseline during training with every initialization seed. However, despite not always outperforming the baseline, the agent demonstrated consistent improvement throughout the training process and came close to matching the baseline's level of performance. Specifically, the agent achieved a *mean* diagnostic score of -54 across five different random seeds, while the baseline, which utilized a fixed strategy for test selection, achieved an average score of approximately -50.
Notably, in one of the five seeds, the agent managed to outperform the baseline, achieving a score of around -47, which demonstrates the agent's potential to surpass the baseline. While the average result didn't exceed the baseline, the agent's ability to approach and, in one case, surpass the baseline highlights its potential to handle **more complex diagnostic scenarios with further training and optimization.**

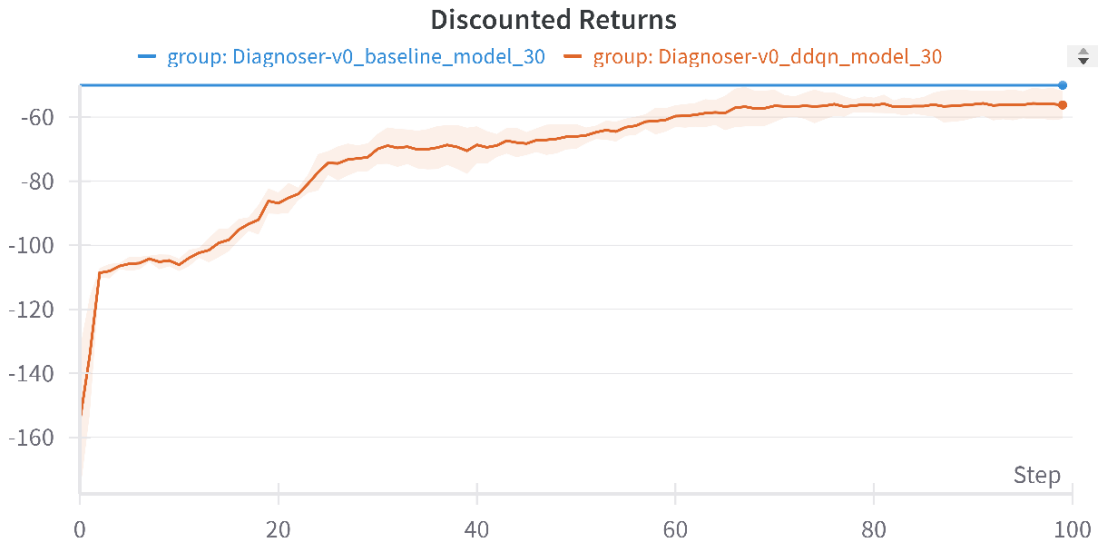

Figure 11: Mean negative cost of diagnosis of the five RL agents trained. One of the five agents succeeded in beating the baseline. Steps in thousands, blue line indicates the cost of the best fixed policy based on balancing entropy and cost.

6 Conclusion

This report provided a first investigation into the application of reinforcement learning for intelligent diagnostics, particularly within the domain of system diagnostics. The results here detailed indicate that integration of reinforcement learning techniques can enable the development of more efficient and adaptive models that have the potential to improve diagnostic accuracy over time. Throughout this research, various methodologies were explored, and the results showed promising potential for real-world applications, particularly in systems where dynamic and complex decision-making is required.

One roadblock we encountered was dealing with the slow environment. Due to having to calculate the conditional entropy of the hardware nodes at every timestep, we only manage to get around 10 steps per second on small models and even fewer on larger models. This means that we did not have time to run an exhaustive hyperparameter search for optimal hyperparameters. It is likely that there exists a hyperparameter configuration that manages to beat the baseline more consistently even on larger models.

We learned that making all root causes uniquely identifiable is crucial to the agent being able to learn good policies. When having multiple undiagnosable root causes, the agent seems unable to learn a sophisticated policy. That can be attributed to our termination condition only ending an episode once one of the hardware nodes has a >85% probability of being broken. For some nodes this may not be possible and thus for some root causes it does not matter which actions you take at what step. One way to solve this would be to have an adaptive broken limit that changes with the root cause to make every root cause identifiable. Alternatively one can exclude undiagnosable root causes from training.

Regarding future work, there are several areas that can be explored. One possibility is to experiment with more advanced reinforcement learning algorithms specifically designed for discrete action spaces, such as proximal policy optimization (PPO), DreamerV3, MPO or soft actor critic (SAC) for discrete action spaces. These algorithms may provide better convergence and performance, especially in scenarios with complex state-action mappings. We already provided an implementation of PPO in the project codebase.

A promising attempt at speeding up the environment could be experimentation with environment vectorization. We tried the native gymnasium vectorization and it did not provide any notable speedup, but there are libraries that implement a much more sophisticated vectorization like PufferLib. These libraries could offer a noticeable speedup. One drawback of using PufferLib is that the only implemented algorithm they have is PPO.

Additionally, there was a case with DQN where the agent overestimates the Q-values of initial states. That means the learned Q-function is not very accurate. This may be improved by testing different variants of DQN or experimenting with the hyperparameters.

Another key area for future exploration is applying the reinforcement learning model to diagnostic problems where certain actions do not yield new information but impact the cost distribution of the diagnostic process. Addressing these scenarios will help in developing

models that are not only accurate but also cost-effective, balancing the trade-off between action efficiency and the quality of information gathered.

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl

TNO innovation for life