TU Delft

Use Reinforcement Learning to Generate Testing Commands for Onboard Software of Small Satellites

Zhuoheng Li

Use Reinforcement Learning to Generate Testing Commands for Onboard Software of Small Satellites

By

Zhuoheng Li

to obtain the degree of Master of Science
at the Delft University of Technology.
to be defended publicly on Wednesday November 30, 2022 at 14:00.

Student Number:
5003636

Supervisor:
Dr. Alessandra Menicucci

Thesis committee:
Dr. Jian Guo, Space System Engineering (SSE)
Dr. Alessandra Menicucci, Space System Engineering (SSE)
Dr. Annibale Panichella, Software Engineering Research Group (SERG)

The frontpage figure “testing a spacecraft” is generated by DALL-E, an AI artist of OpenAI.

TU Delft

Acknowledgement

I would like to thank the development team of the Delfi-PQ satellite. My supervisor **Alessandra Menicucci** gave me a great flexibility to explore. She also advised me to try stress testing with RL, which turned out to be a wonderful idea. **Stefano Speretta** is a great teacher and developer who wants to do something cool. We usually discussed interesting ideas until midnights. At first, I thought collecting near real-time code coverage from MSP432 controllers was difficult because COTS tools could not do that. Stefano disagreed, and now he is correct: we can write our own tool to achieve that. **Mehmet Şevket Uludag**, **Casper Broekhuizen,** and previous developers also provided good software and hardware platforms for this research. Besides, I thank my friends **Johan Monster** and **Tom Hendrix**, who developed the onboard computer code with me.

Many other people provided help to this research. I sought many useful suggestions from **Jian Guo** and his PhD student **Ruipeng Liu**. **Annibale Panichella** from the EEMCS faculty evaluated the basic research idea at the beginning. **Chengyu Jiang** from Exponential Deep Space and the Tsinghua university helped me to contact other researchers in this field for advice. I also learnt a lot from the experience in the National Space Science Centre of Chinese Academy of Sciences, where I developed scheduling algorithms for a CNSA-ESA space telescope, the Einstein Probe.

I thank my parents, **Jian Wang** and **Daoqing Li**, and other family members for supporting me in this long journey. Particularly, thank my girlfriend **Xiaodan Yin** for maintaining a long-distance relationship for such a long time.

I truly appreciate all my long-term friends for maintaining connections with me. I am very sorry that this page is too short to list all your names. However, without your encouragement, I can hardly move forward. It is my honour to have you as my friends. There are also many nice people in Delft, like my roommates and the shopkeepers at Eastern Snack house.

At the end, I want to thank you, the **readers**, for spending time to read this document. I hope this work is a little bit useful for you and promise that the results are real. The code is open source[1][2][3]. If you have any question, please drop me a message

[1] https://github.com/StarCycle/TestCommandGeneration
[2] https://github.com/StarCycle/CodeCoverage
[3] https://github.com/StarCycle/GraphExtract
Note: the emoji “Thanks!” by Teang, 2021 (http://m.weibo.cn/u/5564930684). In the public domain.

Abstract

Programmers usually write test cases to test onboard software. However, this procedure is time-consuming and needs sufficient prior knowledge. As a result, small satellite developers may not be able to test the software thoroughly.

A promising direction to solve this problem is reinforcement learning (RL) based testing. It searches testing commands to maximise the return, which represents the testing goal. Testers need not specify prior knowledge besides the reward function and hyperparameters. Reinforcement learning has matured in software testing scenarios, such as GUI testing. However, migration from such scenarios to onboard software testing is still challenging because of different environments.

This work is the first research to apply reinforcement learning in real onboard software testing and one of few studies that perform RL-based testing on embedded software without a GUI. In this work, the RL agent observes current code coverage and the interaction history, selects a pre-defined command, or organises a command from pre-defined parameters to maximise cumulative reward. The reward function can be code coverage (coverage testing) or estimated CPU load (stress testing). Three RL algorithms, including the tabular Q-Learning, Double Duelling Deep Q Network (D3QN), and Proximal Policy Optimization (PPO), are compared with a random testing baseline and a genetic algorithm baseline in the experiments.

This study also performs regression testing with a trained RL agent, i.e., to test a version of onboard software that it has never seen before. To do that, the agent processes graph input with code coverage information. The graph is extracted from the onboard software source code via static code analysis. The work tries two graph neural network architectures (GGNN and GAT) with several graph pooling mechanisms to process the graph input.

Apart from the test command generation algorithms, some middleware is also implemented, including a command/response parser, a state identification module, a branch coverage collection tool, and a tool to extract the graph representation and node features. During onboard software testing, the onboard computer (OBC) or the electrical group support equipment (EGSE) can be the master of the bus. The command generation algorithms can run on a lab PC or a cloud server.

The research reveals the advantages and drawbacks of using reinforcement learning to test onboard software. On the one hand, RL-based testing performs well in non-deterministic environments (e.g., stress testing) and regression testing. On the other hand, more straightforward methods like random testing and the genetic algorithm are more useful in deterministic environments.

This document also introduces relative background knowledge. It leaves many recommendations for future work, such as improving sampling efficiency, generalization, and learning a model for fault detection in satellite operation.

Acronyms

A2C	Asynchronous Advantage Actor-Critic, an RL algorithm
A3C	Synchronous Advantage Actor-Critic, an RL algorithm
ACER	Actor Critic with Experience Replay, an RL algorithm
ACKTR	Actor Critic using Kronecker-Factored Trust Region
ADCS	The Attitude Determination and Control Subsystem
ADB	Antenna Deployment Board
AI	Artificial Intelligence
API	Application Programming Interface
APP	Application program
CFG	Control Flow Graph
COMMS	Communication System
COTS	Commercial Off the Shelf
CPU	Processor
CRC	Cyclic Redundancy Check
CSV	Comma-separated values, a file format
CubeSat	A form factor of small satellites. The size of 1U is 10*10*10cm
COVID	Coronavirus. I hope young readers to be unfamiliar with it.
DDPG	Deep Deterministic Policy Gradient, an RL algorithm
Delfi-PQ	The first PocketQube Satellite of Delft University of Technology
DQN	Deep Q Network, an RL algorithm
D3QN	Double Dueling DQN, a variant of the DQN algorithm
EGSE	Electrical Ground Support Equipment
EPS	Electrical Power System
ESA	European space agency
FDIR	Fault Detection, Isolation, and Recovery
FLASH	A type of non-volatile computer memory
FRAM	Ferroelectric Random Access Memory
GAT	Graph Attention Network, a type of GNN
GGNN	Gated Graph Neural Network, a kind of GNN
GNN	Graph Neural Network
GPS	Global Positioning System
GUI	Graphic User Interface
I2C	Inter-IC-bus
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
JSON	JavaScript Object Notation
LOBE-P	Low frequency radio payload
MBSE	Model-based system engineering
MC/DC	Modified condition/decision coverage
MD5	Message-digest algorithm, a cryptographic protocol
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron
MPPT	Max Power Point Tracker

NASA	US space agency
NOS3	NASA Operational Simulation for Small Satellites
OBC	On Board Computer
PC	Personal Computer
PocketQube	A form factor of small satellites. The size of 1P is 5*5*5cm
PPO	Proximal Policy Optimization, an RL algorithm
PQ	Shorter abbreviation of Delfi-PQ
RAM	Random-Access Memory
RL	Reinforcement Learning
RX	Receiver
SAC	Soft Actor-Critic, an RL algorithm
SatNOGS	An open-source global network of satellite ground stations
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SUT	System Under Test
SWD	Serial Wire Debug interface
TD3	Twin Delayed DDPG, a variant of the DDPG algorithm
TRPO	Trust Region Policy Optimization
TX	Transceiver
UCB	The Upper Confidence Bound algorithm
XML	Extensible Markup Language

Nomenclature

A	A set of actions
A^{π}	Advantage function under the policy π
a	An action
c	A constant
c_{loop}	Number of clock cycles per loop in the scheduler
c_i	Coverage status of node i
f	Frequency of the CPU
G	Return (discounted cumulative reward) of an episode
g	Environmental Model
$h^{(l)}$	Output from layer l of a neural network
L	Total number of hidden layers in a neural network
$m^{(l)}$	Node embedding after l message passes in a GNN
n_{loop}	Number of loops recorded during the sampled time inteval
N_s	The number of times that state s has been visited
N_{target}	Frequency to update the target network (D3QN algorithm)
N_{τ}	Number of transitions in the replay buffer (D3QN algorithm)
o	A vector extracted from node embeddings of a graph
P	State transition probability function
Q	Action-Value function
R	Reward function
r	A reward
S	A set of states
s	A state
$t_{interval}$	Length of the sampled interval in the stress testing
T	Length of an episode
V	State-Value function
V_i	Feature vector of node i, $V_i = [\mu_i, c_i]$
w	Weights of a function approximator, e.g., neural network
y	A vector that contains previous actions
α	Learning rate in Q-Learning and its variants
$\alpha_{v,u}$	Self-attention factor of GAT, where v is the query node and u is the key node
ε	The probability to select a random action in Q-Learning
δ	TD residual
γ	Discount factor in the return
λ	A hyperparameter in the GAE method to estimate advantage function
τ	A transition $(s_t, a_t, r_{t+1}, s_{t+1})$
$\mathcal{N}$	Neighbors of a node in a graph
ξ	A hyperparameter to control policy update magnitude in PPO
μ_i	Feature vector of node i generated by Word2Vec
π	A policy

General Superscripts

$\blacksquare'$	A variable that has been updated
$\blacksquare^{-}$	A variable that has a lag in update
$\widehat{\blacksquare}$	Estimation of a variable
$\blacksquare^{\pi}$	A function conditional on the policy π

General Subscripts

$\blacksquare_{t}$	The t-th time step
$\blacksquare_{w}$	Function approximator with weights w
$\blacksquare_{start}$	Variable at the beginning
$\blacksquare_{end}$	Variable at the end

Contents

1 Introduction

"Quality is free, but only to those who are willing to pay heavily for it."
—Tom DeMarco and Timothy Lister

Sufficient testing is important for the reliability of small satellites. However, this procedure can be time-consuming and needs enough prior knowledge about the system design. This research tries reinforcement learning to solve this problem. Before introducing the whole idea, we start from the story of our recently launched satellite, the Delfi-PQ.

1.1. A Story of the Delfi-PQ Satellite

Delfi-PQ (Figure 1-1) is a 3P PocketQube and third student satellite made in TU Delft (Radu, Uludag, Speretta, Bouwmeester, Gill, & Foteinakis, 2018). **PocketQube** is a new form factor of tiny satellites. The size of 1P PocketQube is around 5×5×5cm, smaller than the 10×10×10cm of 1U CubeSat. Since they are smaller than CubeSats, PocketQubes have stricter constraints on mass, size, power, and communication budget. However, for some applications like education, technology demonstration, and gravity / magnetic / radiation multi-point measurement, PocketQubes are cheap and competitive (Bouwmeester et al., 2020).

The goal of Delfi-PQ is to demonstrate a reliable satellite bus. If Delfi-PQ is successful and affordable, TU Delft will update the satellite bus and launch a PocketQube periodically with different payloads. At the same time, there will always be a PocketQube in the laboratory for education. In that case, Delfi-PQ can strongly support space-related education and research in TU Delft.

The team made all subsystems, onboard software, and Delfi-PQ electric ground support equipment in-house. The project[4] is open source, and part of the telemetry is available on the SatNOGS[5] website. As a forerunner, Delfi-PQ also helps to set a standard for PocketQube satellites (Bouwmeester, van der Linden, Povalac, & Gill, 2018).

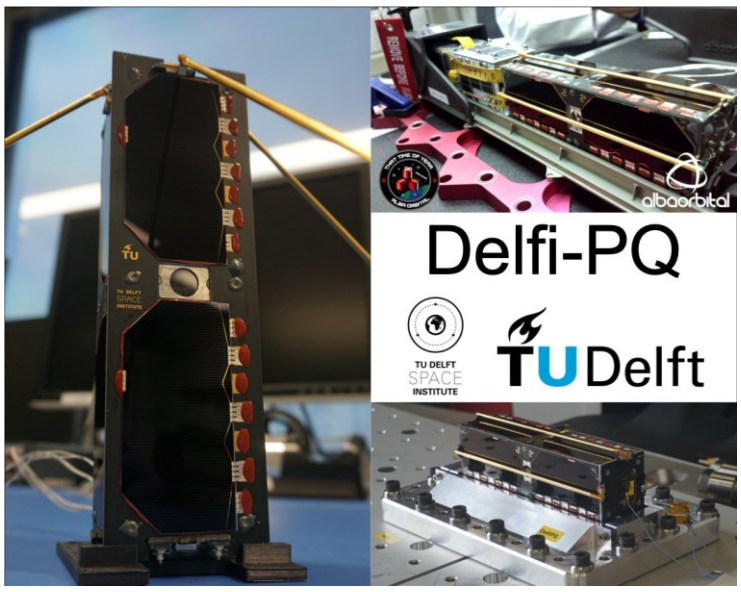


Figure 1-1: **Delfi-PQ satellite**

[4] Delfi-PQ repository: https://github.com/DelfiSpace
[5] SatNOGS is an open-source global network of satellite ground station. Their dashboard: https://dashboard.satnogs.org/

Because of the COVID-19 pandemic, the development and testing schedule was tight. When we transported Delfi-PQ to the launch site in October 2020, the flight software for ADCS (Attitude Determination and Control) and payload operation was incomplete. The GPS board just arrived in the Netherlands and could not work correctly. Furthermore, the engineers mainly conducted some basic tests, e.g., vibration and thermal vacuum tests. For each subsystem, we only specify several test cases to test its main functions.

Although we tested the satellite in a rush, the launch was late. The original plan was to launch it with SpaceX's Transporter-1 rideshare mission in January 2021. However, because of some political issues of Momentus[6], Delfi-PQ (along with some other satellites) was kicked off the deck. The satellite was stored in the United States for a year and finally launched with the Transporter-3 mission in January 2022 (Figure 1-2).

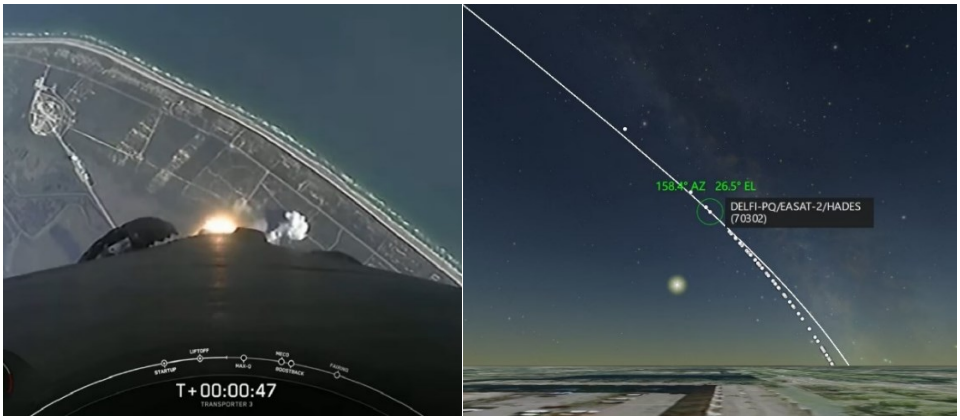


Figure 1-2: **The launch of the Delfi-PQ satellite**

Fortunately, Delfi-PQ survived. Ten hours after the launch, an amateur radio user of the SatNOGS network recorded the first signal from the tiny satellite. We also established the link between the satellite and the Delft ground station the following day. The satellite is still alive and sending telemetry (Figure 1-3).

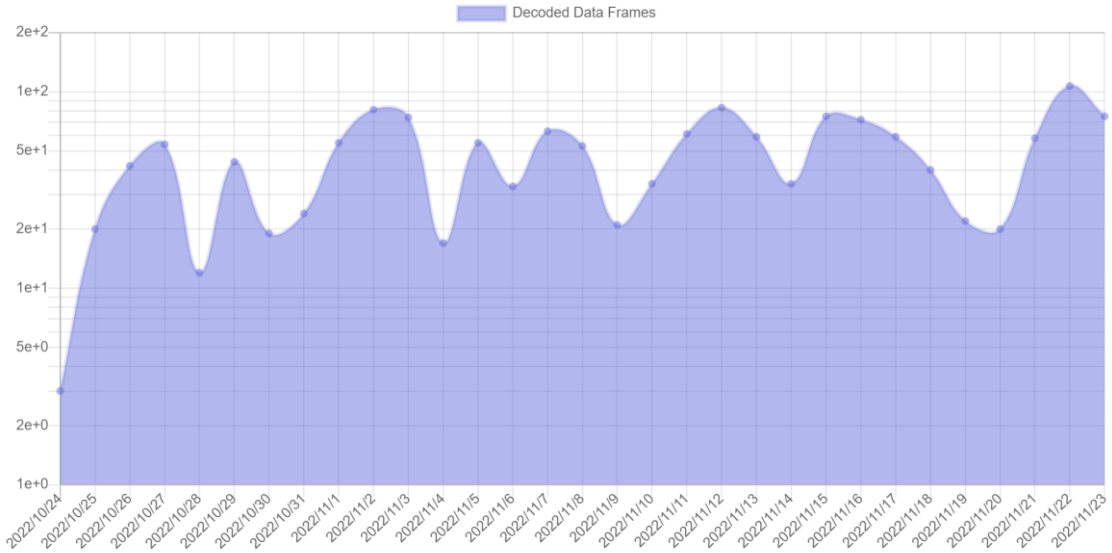


Figure 1-3: **Telemetry from Delfi-PQ received by SatNOGS in the past 30 days**[7]

[6] https://spacenews.com/faa-rejects-payload-review-for-momentus/
[7] From "Data Frames Decoded - 30 Days" ca. 2022. (https://db.satnogs.org/satellite/CEIC-4073-2863-5971-9670#data). In the public domain.

The team identified some problems during the operation of Delfi-PQ. The most significant is that the Electrical Power System (EPS) cannot support continuous downlink or software updates. After Delfi-PQ sends 1 or 2 messages to the ground station quickly, the power bus voltage will drop under a threshold, and all subsystems except the EPS will lose electricity. There are two reasons for this problem[8]:

- The resistance of the battery protection circuit under low temperatures is high, which was not expected before the launch. As a result, the batteries cannot maintain the power bus voltage above the threshold.
- Theoretically, the solar panels can also maintain the power bus voltage. However, the max power point tracker (MPPT) responds too slowly to the load change on the bus, i.e., it cannot shift the max power point before the voltage drops under the threshold.

Compared with other PocketQubes, Delfi-PQ is a "lucky guy." As far as we know, there were 14 PocketQube satellites launched with the Transporter-3 mission, but only four survived in space. Langer and Bouwmeester (2016) fitted the data of 178 launched CubeSat and reported a failure rate of 40% within the first six months, as shown in Figure 1-4. Jacklin (2019) also investigated 550 small satellite missions from 2000 to 2016, where the mass of surveyed satellites ranged from 0.5kg to several hundreds of kilograms. The result shows that 24.2% were total mission failures, and another 11% were partial ones.

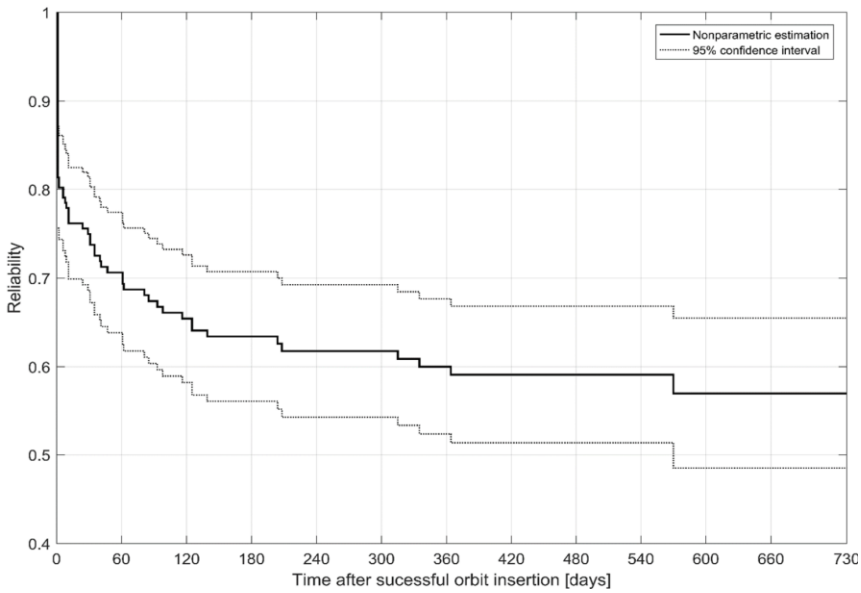


Figure 1-4: **Estimation of CubeSat Reliability Based on 178 Launched Missions (Langer and Bouwmeester, 2016)**

What can we learn from this story?

1.2. Lessons Learned: Keep It Simple and Test It Thoroughly

Commercial off-the-shelf (COTS) components are popular in small satellites. However, in the harsh space environment, they often do not work as described in their datasheets. As mentioned before, Delfi-PQ is suffering from such a problem. If the battery protection circuit does not give us a

[8] This problem was fixed by a successful software update in August 2022. In summer, the tiny satellite had a higher temperature when it passed the Delft ground station, which raised the battery's output voltage. And then, luckily, the battery maintained the bus voltage above the threshold during the initialization of the software update process.

“surprise,” then the satellite operation will be much easier. Unfortunately, the user guide of the battery does not even mention the performance in low temperatures.

Intuitively, engineers should add redundancy to the system since the components are unreliable. However, for tiny satellites like PocketQubes and CubeSat, there is not too much space to add redundant components. At the same time, redundancy adds complexity to the system. Developers need to achieve a careful balance between complexity and reliability.

On this issue, we have some experience with the previous two satellites made in TU Delft. The Delfi-C3 CubeSat, the first student satellite of TU Delft, has a simple design: passive ADCS, no battery, no SD card, and the solar panels directly drive the power bus. Moreover, Delfi-C3 offers limited redundancy by separated controllers to switch the power of subsystems and a redundant radio (Bouwmeester, Aalbers, & Ubbels, 2008). As a result, Delfi-C3 was launched in 2008 and is still alive.

The Delfi-n3xt CubeSat, the successor of Delfi-C3, puts more emphasis on redundancy. It has a redundant OBC, a redundant radio, redundant chains in the EPS, and a simplified backup of the ADCS. However, such redundancy is complex and time-consuming to implement (Bouwmeester, Menicucci, & Gill, 2022). In the end, Delfi-n3xt stopped transmission after only three months of operation[9].

Based on experience from previous student satellites, Delfi-PQ only keeps minimal redundancy, such as separated MPPTs for solar panels and a redundant IMU. The tiny satellite has many single points of failure, including the COMMS, the EPS, and the OBC. The PocketQube also has no more space for redundant hardware.

According to the in-orbit situation of Delfi-PQ, our approach is successful but imperfect. A problem is the lack of testing. Since we cannot avoid single points of failure, we should test them heavily. If Delfi-PQ receives sufficient environmental testing, we can likely identify the anomaly of the battery protection circuit.

Therefore, the take-home message of this section is to keep the tiny satellite simple and test it thoroughly. By Monte Carlo simulation of a failure model, (Bouwmeester et al., 2022) also found that improving testing is better than adding subsystem redundancy for CubeSat reliability.

We plan to take a test-driven approach when developing other PocketQubes at TU Delft. Unfortunately, traditional manual testing needs significant human labour and prior knowledge, so we only wrote several test cases for each subsystem of Delfi-PQ. Conventional manual testing limits our testing coverage.

A question comes to our mind: Can we automate the testing procedure?

1.3. Use Reinforcement Learning to Test Onboard Software Thoroughly

Satellites require many tests at different levels. Physical tests are usually expensive and cannot be performed for many times. For example, a vibration test of Delfi-PQ at a third-party organization takes around €10000. A thermal vacuum test has roughly the same cost. As a result, it is difficult to automate such physical tests in a master thesis with limited budget.

By contrast, onboard software testing is much cheaper and suitable for a thesis. Note that developers can also test some hardware in onboard software testing and simulate some external

[9] After 7 years of silence, Delfi-n3xt restarted to transmit signals in 2022 for a few months and then stopped beeping again.

signals (hardware-in-the-loop). This type of testing can also identify many potential problems with a satellite.

Therefore, this work focuses on testing onboard software automatically with limited prior knowledge. Delfi-PQ is a use case in the study.

Traditionally, the following methods can generate test cases:

- **Model-based testing**. Humans specify a model (e.g., a Finite State Machine) of the tested software and then use a tool to traverse the model. The tool records the traversal paths as test cases. However, a model written by experts with prior knowledge of the tested software is necessary.
- **Symbolic execution** substitutes all program variables with symbolic values, simulates the execution of tested software step by step with constraint solving, and looks for all executable paths. These paths are test cases. This method can only test small-scale software, such as unit testing, because of the difficulty of applying constraint solving to the whole software.
- **Random testing**, i.e., choosing testing command randomly. However, some behaviours of the tested software require particular command sequences. Generating such sequences is difficult for random testing, which does not consider any causal relation among commands.
- **Search-based testing** transfers the test case generation problem to an optimisation problem. It usually uses evolutionary algorithms (e.g., the genetic algorithm) to maximise a human-defined objective function. Solutions of the highest objective functions are test cases.

Among these methods, search-based testing is the most promising for Delfi-PQ onboard software. Random testing is too simple compared to search-based testing, and its performance cannot improve further. Applying model-based testing to the Delfi-PQ software is challenging because no model exists. Symbolic execution is also not suitable for the whole complex software.

Many papers use the genetic algorithm in search-based testing, but it has some shortages:

- It only uses the objective function of a whole test case to guide the search and discards all information generated at each test case step. Such lack of information may make the genetic algorithm challenging to reach a good solution. The situation worsens as the test case's length grows because the search space size will grow exponentially.
- Moreover, the output of the genetic algorithm is a fixed solution, e.g., a fixed command sequence. The command sequence may not work well in a non-deterministic environment.
- Another problem is that the genetic algorithm spends a long time generating test cases for a specific software version. It cannot learn the pattern of a series of versions. In regression testing, the tester may need to run the time-consuming algorithm repeatedly.

A novel approach to search-based testing is **reinforcement learning** (RL), which treats software testing as a sequential decision-making problem, or a **Markov Decision Process** (MDP). Figure 1-5 shows an intuitive example of reinforcement learning. It generates testing commands according to the observed system state and learns the system behaviour automatically. Thus, RL can utilise information generated at each test case step.

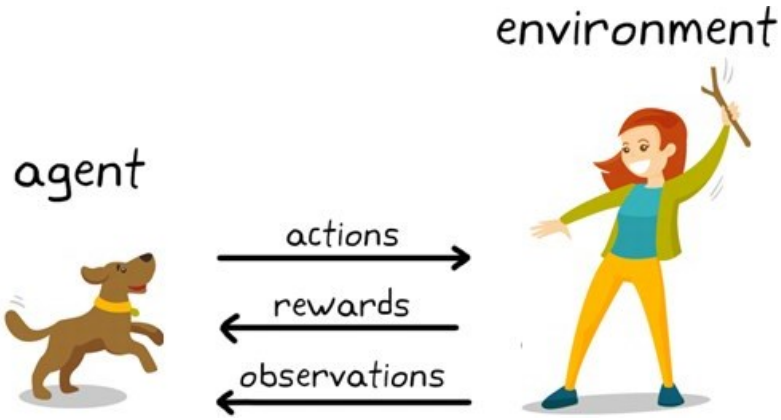


Figure 1-utilisetest case step in Dog Training[10]

RL has shown promising performance on some complex decision-making problems which the genetic algorithm cannot solve. A famous example is the AlphaGo (Silver et al., 2016) of DeepMind. AlphaGo beat the best human players in the Go game, which was never achieved before because of the enormous search space.

Therefore, RL-based software testing has attracted more attention recently, especially in GUI testing. Bytedance, the mother company of TikTok, uses an RL agent Fastbot as the primary stability and compatibility testing tool for more than 20 applications, locating more than 100 crashes daily (Cai, Zhang, & Yang, 2020). Electronic Arts, who made Need for Speed and Call of Duty, also uses RL to test first-person shooter games (Bergdahl, Gordillo, Tollmar, & Gisslén, 2020). Some commercial RL-based testing tools are already available, such as Test.AI for GUI testing of mobile apps (Test.ai, 2021) and Diffblue cover for unit testing of Java (Lodge, 2021).

However, no one has migrated RL-based testing to the space industry or onboard software testing. The migration is challenging because of different environments and software behaviours.

This research is the first work to apply reinforcement learning in integration tests of satellite onboard software:

- Unlike previous research that relies on GUI information, it utilises near real-time code coverage information from the software under test to compute states and rewards.
- A tool is written to retrieve the code coverage data and can be easily modified to adapt other embedded software.
- The research considers two types of testing goals, maximising code coverage and maximising the CPU load (stress testing).
- Experiments are carried out in two environments: a toy problem and the COMMS onboard software.
- It tries three reinforcement learning algorithms, including the Q-Learning, the Double Duelling Q Network (D3QN), and the Proximal Policy Optimization (PPO) algorithm to learn. Each algorithm has several configurations.
- Different state and action representations are tried. The RL testing agent can send human-specified commands or organise command parameters.
- Different neural network architectures (multi-Layer Perceptron MLP, Gated Graph Neural Network GGNN, and Graph Attention Network GAT) are tested. We also analyse several design details in the neural network architecture, such as the selection of the graph pooling layer.

[10] From “Three Things to Know About Reinforcement Learning,” by By Emmanouil Tzorakoleftherakis, 2019. (https://www.kdnuggets.com/2019/10/mathworks-reinforcement-learning.html). In the public domain.

- In the end, the performance of RL algorithms is compared with two baselines: random command generation and the genetic algorithm.

1.4. Structure

In the rest of this thesis, chapter 2 will introduce the related works, including basic concepts of software testing, a brief overview of embedded software testing and onboard software testing. It also discusses traditional test case generation methods and RL-based testing in detail. The end of this chapter lists research questions and assumptions.

Chapter 3 will analyse the onboard software of Delfi-PQ and formulate it as a Markov Decision Process (MDP). It also compares the Delfi-PQ flight software with other onboard software. At the same time, this chapter introduces the hardware and software tools used in this research, such as

- The tool to retrieve near real-time code coverage information.
- The means to extract graphs from the source code of onboard software
- The telemetry/telecommand parser
- The means to transfer messages between an online training server and the satellite hardware.

Chapter 4 describes the reinforcement learning algorithms used in the work, including the Q-Learning algorithm, the Double Duelling Deep Q Network (D3QN), and the Proximal Policy Optimization (PPO) algorithm. Note that each algorithm may have several configurations with different neural network architectures or state/action representations. The chapter also describes the random testing baseline and the genetic algorithm baseline.

Chapters 5, 6, 7, and 8 describe the designs of the experiments and the results. There are four types of experiments: a “filling grid” toy problem, stress testing, coverage testing, and regression testing.

Chapter 9 gives conclusions of this research, threats to validity, and recommendations for future works.

2 State of the Art

This section briefly reviews related works and provides definitions of some concepts. More specifically,

- Section 2.1 introduces some basic concepts in software testing. It also reviews some trends in embedded software testing and onboard software testing.
- Section 2.2 introduces and compares traditional test case generation techniques.
- Section 2.3 briefly explains the Markov process and reinforcement algorithms. It also reviews progress in RL-based software testing, particularly the formulations of states, actions, and rewards.
- Section 2.4 gives the assumptions and research questions in this work.
- Section 2.5 is a summary of this chapter.

2.1. A Bite on Software Testing

Traditionally, software testing has the following steps:

- Step 1: Analyse the requirement document.
- Step 2: Make a test plan, including testing methods and environment settings.
- Step 3: Set the testing environment.
- Step 4: Write test cases, execute them, and detect faults.
- Step 5: Record the faults and let software engineers debug.

Some concepts and definitions are listed in the following subsection.

2.1.1 Basic Concepts and Definitions

In the field of software testing,

- A **test case** is a pre-written sequence of **testing commands** which will be executed during a test. A **test suite** is a set of test cases.
- A **test oracle** is a set of conditions to determine whether the system under test behaves correctly or not.
- **Test coverage** is a metric that shows the amount of testing performed. Different coverage types include code coverage (e.g., line coverage, branch coverage, MC/DC coverage), state coverage, and requirement coverage. Developers usually pursue high test coverage.
- **Functional testing** verifies whether the **system under test** (SUT) behaves as expected. **Non-functional testing** examines non-functional parameters of the SUT, such as performance and security.
- **Black box testing** compares software output with expected values and sometimes reduces the number of test cases by pairwise testing, equivalence partitioning, and boundary analysis. **White box testing** drives test cases from the source code of SUT (manually or automatically).

Software testing has different levels. As shown by Figure 2-1 (Nakkasem, 2020), there are usually **unit testing**, **integration testing**, **system testing** and **acceptance testing**. For higher-level tests, the SUT becomes more complex, and it is usually more difficult for testers to access internal information of the SUT.

Before coding, we may also need to **test requirements** and system design, which technologies like model checking can do. After the acceptance test, if the software is updated and has some new capabilities, it needs **regression tests** to ensure the original capabilities are not affected. We can reuse test cases in regression tests.

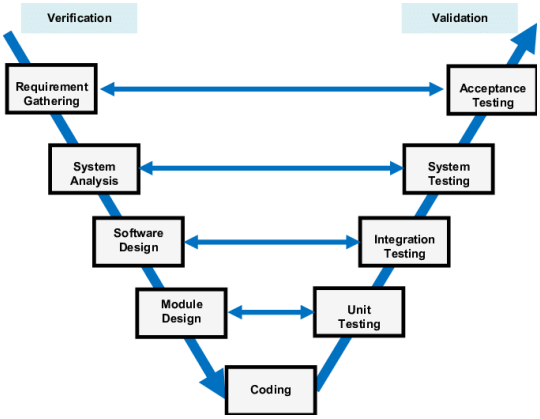


Figure 2-1: V Model of Software Development (Nakkasem, 2020)

2.1.2 Embedded Software Testing

Onboard software is embedded software that runs on a **target board** and does not have a direct user interface. A target board usually communicates with other target boards or a **host computer** with a direct user interface and a software development kit. The target board may also sense external signals like temperature, acceleration, and light intensity.

The following things make embedded software testing different from, and sometimes more complex than, conventional software testing (e.g., PC, web or mobile applications):

- The target board only has limited computing resources.
- It is more challenging to get information from the target board when the microcontrollers do not support debug capabilities like tracing. It raises a need for sophisticated instrumentation and probing when testing embedded systems.
- External signals like temperature will affect a test, and sometimes testers must provide such signals during the test.
- Embedded software is usually developed in parallel with hardware. There may be only a few new hardware available for software testing.
- Embedded software is closely integrated with hardware. A fault may come from hardware rather than software.

Such challenges have led to the wide adoption of **simulation-based testing** in the embedded software industry. Some simulators can simulate all or part of embedded hardware, so engineers do not always perform tests on target boards. Depending on which part is under simulation, this approach can be called **X-in-the-loop**, e.g., hardware-in-the-loop (HiL), software-in-the-loop (Sil), model-in-the-loop (MiL), processor-in-the-loop (PiL). Examples of such simulators include Qemu, Tina, and some simulation capabilities in embedded software IDEs. There are also simulators used as mocks of sensors or subsystems.

However, configuring such simulators may take much effort, especially when configuring different peripherals. Likewise, any simulator cannot 100% mimic real hardware or environment. Many embedded software tests are still performed on real hardware (Garousi, Felderer, Karapıçak, & Yılmaz, 2018), as shown in Figure 2-2.

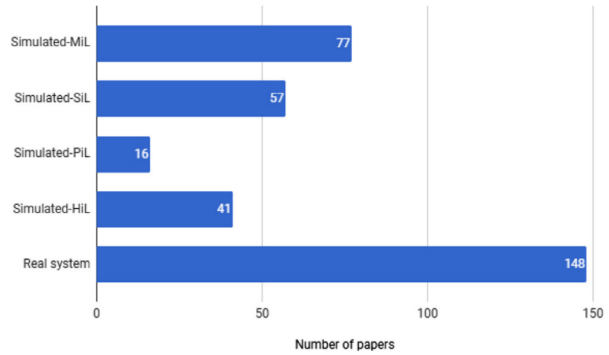


Figure 2-2: Papers in terms of using simulated or real SUTs (Garousi et al., 2018)

2.1.3 Testing Onboard Software of Satellites

Like other embedded software, simulators are popular in onboard software testing, especially for large spacecraft. They can simulate hardware, environment, or dynamic models of satellites. Theoretically, it is possible to build a **virtual satellite**. An example is the NASA Operational Simulation for Small Satellites (NOS3) used in the STF-1 CubeSat mission, which includes (Geletko et al., 2019), as shown in Figure 2-3:

- NASA Operational Simulator (NOS), which simulates hardware busses.
- core Flight System (cFS), an open-source flight software used by NASA since 1992.
- Custom hardware simulators, including a processor simulator.
- COSMOS is an open-source ground station software that sends telecommands to the system.
- OIPP, a planning tool that acknowledges the ground station when the satellite will be in view/sunshine.
- 42, an open-source simulator for spacecraft attitude and orbital dynamics.
- Vagrant helps to set up a virtual machine to run the applications in the NOS3 suite.

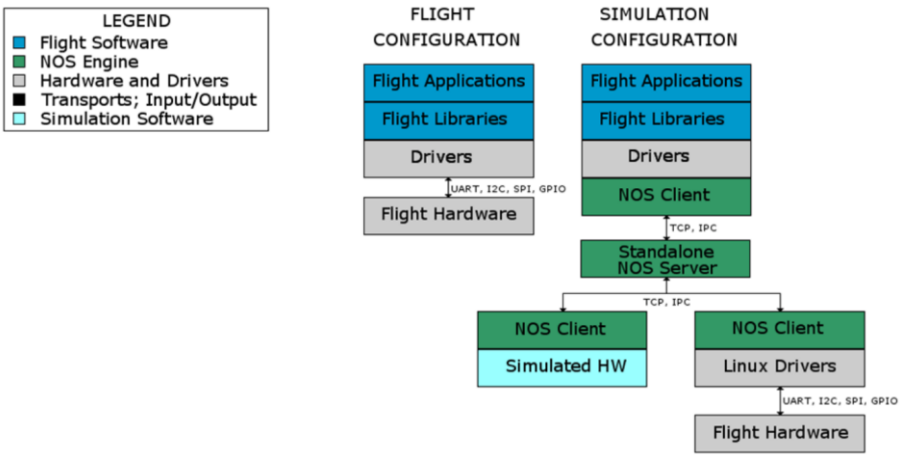


Figure 2-3: Architecture of NOS3 (Geletko et al., 2019)

While NASA can invest heavily in a “virtual satellite”, this approach is not feasible for many other developers. For example, the Delfi-PQ uses the Texas Instrument MSP432 microcontrollers, which do not have an available simulator. It is also tricky to create simulators for peripherals on the boards. Some developers only use limited simulation in onboard software testing as a compromise. The OpenSatKit only includes the core Flight System, COSMOS, and 42, which form a minimal simulated environment (McComas, 2021).

As a product of model-based system engineering (MBSE), **automated code generation** is getting prevalent in onboard software development. In this approach, developers define software architectures with graphic models, which include software behaviour with sequence diagrams and message passing among program modules. After that, low-level embedded code will be generated and tested automatically (Jacklin, 2015). An example of this approach is F Prime (Bocchino, Canham, Watney, Reder, & Levison, 2018), a famous open-source onboard software architecture of NASA Jet Propulsion Laboratory, as shown in Figure 2-4. F Prime generates boiler-plate code of components and ports from XML or SysML specifications. It can also automatically generate test classes for unit testing and provide a Python API for integration testing.

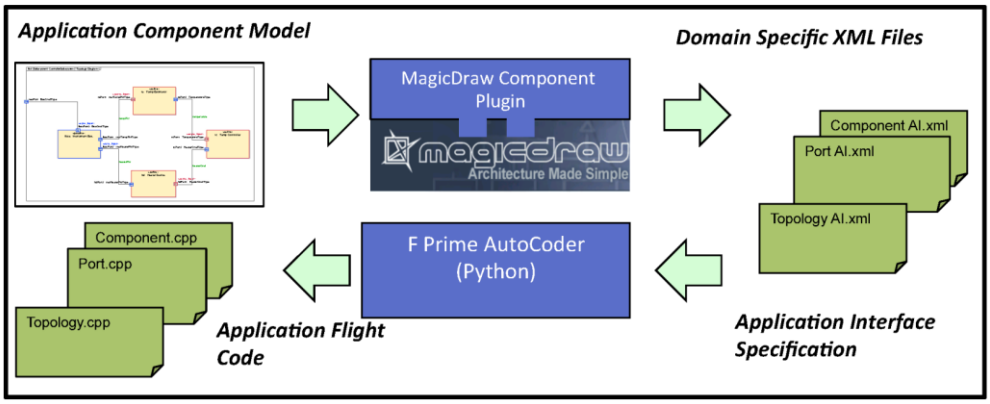


Figure 2-4: F Prime Generates Application Code from Models (Bocchino et al., 2018)

Automated code generation makes onboard software development less prone to errors and automated test generation possible. However, some developers argue that it is less flexible than hand-written code. The learning curve of such tools may also be higher (Jacklin, 2015).

A helpful testing method for high-safety onboard software is **model checking**. As shown by Figure 2-5 (Chen, & Wu, 2010), developers must represent the flight software in a finite state machine. Then a model checker will search every possible path to prove that the software satisfies some properties expressed in temporal logic. Instead of looking for a bug, model checking tends to prove that the software satisfies the requirements.

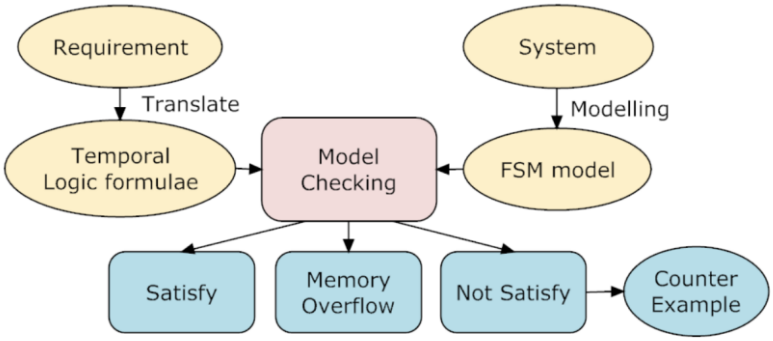


Figure 2-5: The Flow Chart of Model Checking (Chen, & Wu, 2010)

There are some challenges of model checking:

- It is challenging to represent onboard software as a finite state machine, especially when there are continuous parameters.
- The number of states may be too large to search.
- Not all requirements can be written in temporal logic, a precise mathematical specification.

Because of these challenges, model checking is helpful for some safety-critical software instead of the whole onboard software. As a use case, ESA has used its COMPASS toolset to perform model checking on a satellite's FDIR (Fault Detection, Isolation, and Recovery) software. There are thousands of requirements for the checked software, but only 106 can be described in temporal logic. The team chose 26 requirements and constructed a model of 4000 lines of code and around 50 million states. In the end, the engineers only successfully verified 16 requirements, and others took too much computing time to find a result (Esteve, Katoen, Nguyen, Postma, & Yushtein, 2012).

It is worth mentioning that onboard software testing may follow some **standards**, including:

- NASA-STD-8739.8A, the NASA Software Assurance and Software Safety Standard requirements.
- ECSS-Q-80, Software product assurance. ECSS means the European Cooperation for Space Standardization.
- ECSS-E-40, Space Software Engineering, which evolved from ISO 12207, Software life cycle processes.
- QJ 3027A-2016 5.7.22, Software Testing Standard for Spacecrafts in China.

For more information about onboard software testing, (Jacklin, 2015) is a comprehensive survey. Figure 2-6 also shows the popularity of some open-source onboard software architectures, which usually include testing tools.

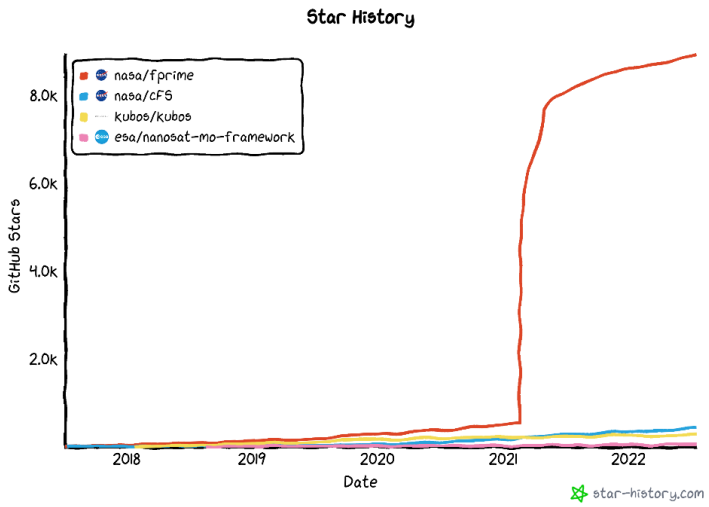


Figure 2-6: GitHub Star History of Popular Onboard Software Frameworks

While onboard software testing has its focuses and toolsets, its nature is not different from general software testing. Therefore, the next section will introduce automated test case generation from the point of view of general software testing.

2.2. Automated Test Case/Command Generation Techniques

Before introducing traditional test case generation techniques, it is helpful to understand goals of test cases.

2.2.1 Goals of Test Case/Command Generation

Test case generation means looking for test cases, i.e., sequences of testing commands before a test starts. During the test, these fixed test cases will be executed. On the other hand, **test command generation** means selecting testing commands during a test, according to the current state of SUT. The techniques introduced in this section are mainly for test case generation, while search-based testing with reinforcement learning can generate commands during a test.

There are many types of testing goals, but most of the methods in this section only support several of them:

- Model-based methods are suitable for the goals represented by a model, e.g., state coverage, transition coverage and requirement coverage.
- Symbolic execution is usually used to maximize code coverage.
- Random testing does not adapt to specific testing goals, but some researchers use code coverage to measure its performance.
- Search-based testing supports the testing goals that can be quantitatively represented as objective functions (Harman, Jia, & Zhang, 2015).

This section uses code coverage to compare different test case generation techniques. It is the only testing goal supported by all techniques mentioned here.

2.2.2 Model-Based Test Case Generation

In **model-based test case generation**, humans specify a model (e.g., a Finite State Machine) of the tested software and then use a tool to traverse the model. The test cases are the traverse paths (Shirole & Kumar, 2013). On the other hand, testers can also use mutated models to generate wrong test cases, which verify the fault handling mechanisms of SUT (Belli, Budnik, Hollmann, Tuglular, & Wong, 2016).

If there is a model available before testing (e.g., F Prime), this approach is convenient. It can find every path in the model. However, writing them can be time-consuming if a model is unavailable. Some testing tools can learn a model of SUT from execution traces, and we will discuss them in section 2.3.

The model-based testing process is summarized in Figure 2-7 by (Garousi, Felderer, Karapıçak, & Yılmaz, 2018).

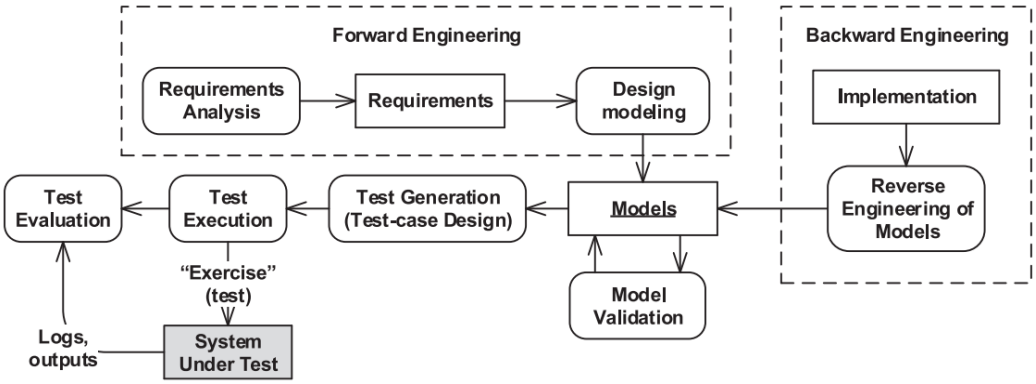


Figure 2-7: Process of Model-Based Testing (Garousi et al., 2018)

2.2.3 Symbolic Execution

Symbolic execution analyses source code to generate test data that can achieve high code coverage. In the process of code analysis, it uses symbolic variables to simulate the execution of the software. At any point during symbolic execution, it maintains current symbolic variables, a path constraint on the symbolic variables, and a program counter. This path is feasible only when software inputs can satisfy the path constraint. This way, symbolic execution can find all feasible paths, their path constraints and test inputs. Figure 2-8 shows an example of symbolic execution (Anand et al., 2013).

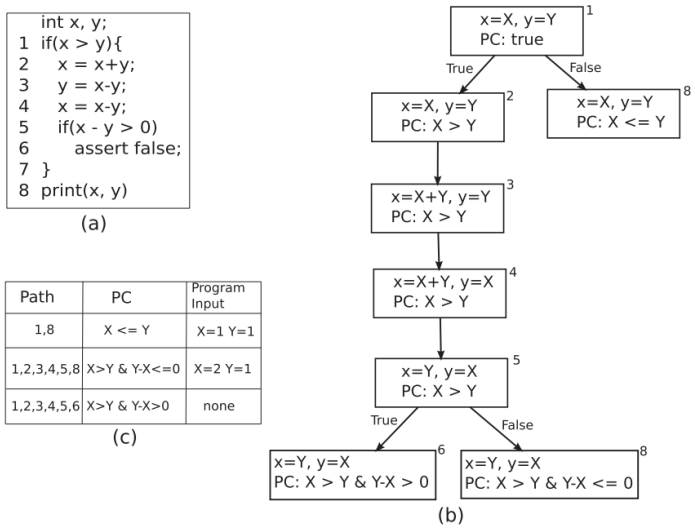


Fig. 1. (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

Figure 2-8: An Example of Symbolic Execution (Anand et al., 2013)

Although King proposed symbolic execution in 1975, the method only became feasible in the 21st century because of more powerful constraint solvers and computers. It still has some fundamental problems:

- Path explosion: Most real-world software has many paths, and many of these paths are infeasible. It takes too much time to execute all the paths symbolically.
- Path divergence: Most real-world software uses multiple programming languages, and parts of them may be available only in binary form. Users need to provide models for the problematic parts.
- Complex constraints: some path constraints include non-linear operations like multiplication and division and mathematical functions like *sin* and *log*, which available constraint solvers cannot solve.

For example, NASA used its Symbolic Java Pathfinder to perform symbolic execution on a Java model of an ascent abort handling software (Păsăreanu et al., 2008). To deal with path divergence and complex constraints, it used concrete executions of SUT to gather information for symbolic execution. The Pathfinder generated 200 test cases to cover all aborts and flight rules within 2 minutes. However, the Java model under test only contained ~600 lines of code and was not actual flight software.

2.2.4 Random Testing

Random testing means choosing testing commands randomly. The idea behind random testing is to let test cases spread evenly across the input domain. On the other hand, it is also the

disadvantage of random testing: sometimes, test cases should not be evenly spread in the input domain. For example, random testing only covered a few flight rules and no aborts of the Java model (Păsăreanu et al., 2008).

A famous random testing tool is the Monkey, provided by the Android SDK (Patel, Srinivasan, Rahaman & Neamtiu, 2018). It can quickly generate and execute test cases because it does not contain complex logic.

2.2.5 Search-Based Testing

Unlike random testing, **search-based testing** searches for test cases that maximise a pre-defined objective function. In other words, it transfers the test case generation problem to an optimisation problem. Many search algorithms are available (Utting, Pretschner, & Legeard, 2012), including metaheuristic search, simulated annealing, and evolutionary algorithms (the genetic algorithm).

Much of the literature on search-based testing focuses on the genetic algorithm (Harman, 2011). However, it has some disadvantages:

- It only uses the objective function to guide the search, which may be challenging to reach a good solution. The situation worsens as the test case's length grows because the search space size will grow exponentially.
- It generates a fixed test case unsuitable for non-deterministic software under test.
- If the software under test is modified, the genetic algorithm must be rerun to generate new test cases, which can be time-consuming.

An example of a search-based testing tool is the Sapienz (Mao, Harman & Jia, 2016). UCL first developed it as a research program, but Facebook massively deployed it after 17 months (Mao, 2018). Sapienz uses a multi-objective genetic algorithm to generate test cases with maximal objective functions. If the source code of the Android app is available, Sapienz measures statement coverage as the objective function during the app's execution. When the source code is unavailable, Sapienz measures method coverage or activity coverage instead. Figure 2-9 shows the workflow of Sapienz.

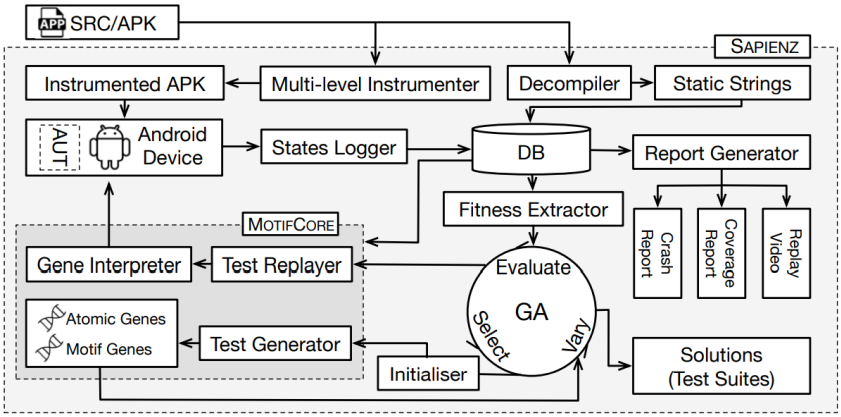


Figure 2-9: Sapienz Workflow (Mao, Harman & Jia, 2016)

2.2.6 Comparison

As a summary, Table 2-1 compares these techniques for test case generation.

It is also interesting to look at some statistics. Garousi, Felderer, Karapıçak, and Yılmaz (2018) reviewed 312 papers about embedded software testing. Among them, 150 papers are model-based, followed by 24 papers about search-based testing, 23 papers about random testing, and a few papers about symbolic execution. The survey shows that model-based testing is the most heavily researched for embedded software testing, and other techniques are less popular.

Why do we see this trend? Garousi et al. (2018) thought these papers on model-based test case generation are for requirement-based testing, i.e., using functional requirement coverage as the testing goal. For most software under test, it is necessary to perform tests against functional requirements. Encoding such requirements into a state transition model is straightforward (e.g., do X in state Y), so model-based test case generation is suitable for the general software development process. However, it has some disadvantages. For example, model-based testing needs sufficient prior knowledge to specify a model, and we do not have such a model for Delfi-PQ.

Table 2-1: Comparison of Methods for Test Case Generation[11]

Method	Prior Knowledge Needed	Code Coverage	Testing Level	Adapt to Other Testing Goals
Model-based	Usually need manual specification of the model (states and transitions)	Cover every transition of the model, instead of branches of source code	Usually in high level tests (e.g., system testing) because such models from requirements exist	Yes, if the goals can be represented in a model
Symbolic Execution	Automatically extract information from the source code	Can cover every path if constraint solving is feasible	Usually in low level tests (e.g., unit testing) because of limited constraint solving capability	No, it's mainly for code coverage
Random Testing	Not needed	Cannot cover every path if the SUT is complex	All levels	Cannot adapt to a specific goal
Search-based	Need specifications of the objective function	Better than random testing, the limited by the capability of the search algorithm	All levels	Yes, if the goals can be represented in an objective function

Search-based testing is less prevalent in the survey, but it is a promising option for testing goals that can be represented as an objective function. Therefore, search-based testing is an ideal tool to verify non-functional requirements. Table 2-1 shows that search-based testing does not have a "bad" property. Though the genetic algorithm may have some shortages (section 2.2.5), this approach with other search algorithms may perform better.

2.3. Use Reinforcement Learning to Generate Testing Commands

Reinforcement learning can be seen as a search-based testing method. Most reinforcement learning (RL) algorithms are developed for MDP. Therefore, before applying reinforcement learning to command generation, it is necessary to construct the problem as a Markov Decision Process (MDP).

2.3.1 Software Testing as a Markov Decision Process

An MDP is a 4-tuple (S, A, P, R), where (Howard, 1960):

[11] Meaning of colors in the table: green (helpful), yellow (acceptable), red (bad).

- S is a set of states.
- A is a set of actions.
- $P(s'|s,a)$ is the probability that action a in state s will lead to state s'.
- $R(s'|s,a)$ is the reward received after state transition.

An MDP needs to satisfy the **Markov property**, which means the state transition probability is only related to the current state and action instead of the entire history of the agent's interaction with the environment. Some MDP also assumes the current state is **fully observable**. These assumptions simplify the problem a lot.

We need to represent states, actions, and rewards for constructing software testing as an MDP. Figure 2-10 shows these elements and t is the time step.

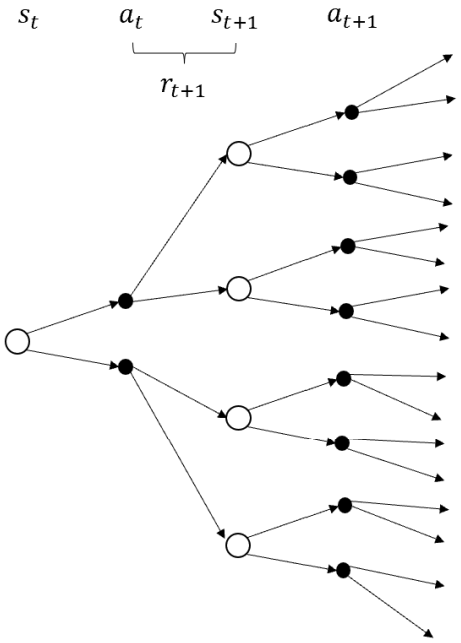


Figure 2-10: A Markov Decision Problem at Step *t* and *t+1*

There are three ways to represent states in RL-based testing: graphical user interface (GUI) information, parameters in the program, or code coverage information. Here are some examples:

- Adamo, Khan, Koppula, and Bryce (2018) extracted states from the GUI of Android applications. It used the Appium and UIAutomator tools to retrieve XML representations of the app's GUI, including widgets and the types of actions (e.g., click, long press) enabled on them. A state includes all actions available on all widgets.n
- Bergdahl et al. (2020) used parameters of the games under test to construct state vectors. A state vector contains the player's position relative to the goal, velocity, rotation, and jump cool-down time. Such practice is common for game AI.
- Dai, Li, Wang, Singh, Huang, and Kohli (2019) represented a state as a graph. A node in a graph is a program branch. It contained a node feature vector V and a coverage mask $C \rightarrow \{0,1\}$, indicating whether the node had been covered. Edges showed relations among branches. The structure of such a graph can be static or changing during testing (if new program branches are found).

RL-based testing can use many types of actions. For example, the actions space can be discrete, i.e., it contains a limited number of actions like clicks and long presses (Adamo et al., 2018). An

action can also be a vector with discrete values (multi-Discrete action space) or a vector with continuous values (continuous action space). For example, Dai et al. (2019) used a sequence of characters or a 2D array of characters as an action of the RL agent.

Rewards of RL-based testing are usually new state coverage, code coverage, risk, or load of the SUT. Testers can also use other metrics to calculate rewards. RL-based GUI testing usually maximizes state coverage. One problem with this approach is determining if a new state has been reached. For example, there may be other news on the same home page of a GUI, as shown in Figure 2-11. Traditional methods may think that the home page contains many states, but Pan, Huang, Wang, Zhang, and Li (2020) used a curiosity module to recognize the page as a single state.

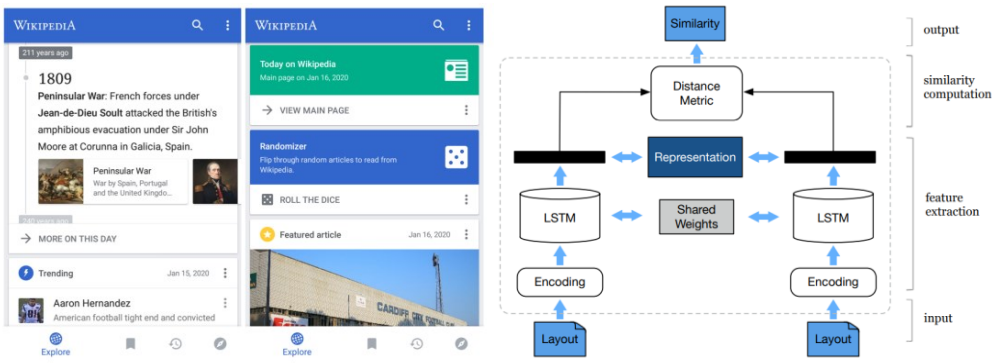


Figure 2-11: A Curiosity Module in RL-Based GUI Testing (Pan et al., 2020)

Few works, like (Dai et al., 2019), used code coverage to compute rewards, i.e., receive a reward when covering new program branches. By contrast, many papers used final code coverage as the metric to evaluate their RL agents, such as (Vuong & Takada, 2018), (Adamo et al., 2018), and (Pan et al., 2021). The reason behind such a phenomenon is the difficulty of getting "real-time" code coverage of software under test, i.e., measuring the code coverage growth caused by taking action. It is technically feasible, but most code coverage tools only generate a code coverage report after the software stops running.

Several papers included the risk or performance of the SUT in rewards. Reichstaller, Eberhardinger, Knapp, Reif, and Gehlen (2016) used a behaviour model of the SUT to evaluate the risk of given failure situations. The risk was then used as a reward for the reinforcement learning algorithm. Moreover, Ahmad, Ashraf, Truscan, and Porres (2019) calculated rewards with elapsed execution time.

Defining states, actions, and rewards do not mean the problem is a standard Markov process. For example, (Adamo et al., 2018) formulated the reward as:

$$R = \frac{1}{N_s + c} \tag{2-1}$$

where N_s is the number of times that state s has been visited, and c is a constant. Reward (2-1) decreases as N_s increase, so the rewards of the same transitions change with the interaction history. In this case, the problem is an **online MDP** (Even-Dar, Kakade, & Mansour, 2009), an extension to the standard MDP. Although the Q-Learning algorithm is not designed for online MDPs, it reached good code coverage in (Adamo et al., 2018).

2.3.2 Brief Introduction to Reinforcement Learning Algorithms

This section cannot introduce reinforcement learning algorithms in detail due to length constraints. For further information and detailed derivation of the formulas, please refer to the literature study document in the appendix. However, it is helpful to understand some basic concepts in reinforcement learning (Sutton & Barto, 2018):

A **return** G_t is a cumulated reward from time step t to **horizon** T:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \tag{2-2}$$

where $\gamma \in (0,1)$ is the **discount factor**.

A **policy function** $\pi(a|s) = P(a|s)$ determines the probability that the agent will select action a in state s. Under a policy π, an **action-value function** $Q^\pi(s,a)$ shows the expected return of state s if action a is selected (Q^π is time-independent):

$$Q^\pi(s,a) = \mathbb{E}[G_t | s_t = s, a_t = a] \tag{2-3}$$

Likewise, under a policy π, a **state-value function** $V^\pi(s)$ shows the expected return of state s (time-independent):

$$V^\pi(s) = \mathbb{E}[G_t | s_t = s] = \sum_{a \in A} \pi(a|s) Q^\pi(s,a) \tag{2-4}$$

The task of reinforcement learning is to approximate these functions to maximize the return. The approximators can be tables or neural networks. Using tables as approximators suits problems with small, discrete states and action space. On the other hand, if there are many (or continuous) states and actions, it is more suitable to use approximators like neural networks.

In the early days, reinforcement learning algorithms could be value-based or policy-based, depending on which function they approximated. **Value-based RL** algorithms only learn the action-value function $Q^\pi(s,a)$. After training, they will select the action with maximal $Q^\pi(s,a)$. Typical tabular value-based algorithms are SARSA (Rummery & Niranjan, 1994) and Q-Learning (Watkins & Dayan, 1992), which use a Q table to record $Q^\pi(s,a)$. Deep Q Network DQN (Mnih et al., 2015) and its variants use a neural network to approximate $Q^\pi(s,a)$. On the other hand, **policy-based RL** algorithms only learn the policy function $\pi(a|s)$, such as REINFORCE (Williams, 1992).

Both approaches have some disadvantages. For example, value-based algorithms are not good at problems with ample action space, and policy-based algorithms use basic Monte Carlo sampling with high variance and long sampling time. As a compromise, many later algorithms learn both the policy function, approximated by an **actor network** and the state-value function, approximated by a **critic network**. Examples of such algorithms are PPO (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017), A3C/A2C (Mnih et al., 2016), DDPG (Lillicrap et al., 2015), TD3 (Fujimoto, Hoof, & Meger, 2018), and SAC (Haarnoja, Zhou, Abbeel, & Levine, 2018).

Some RL algorithms also learn an **environmental model** $g(s_t, a_t) = s_{t+1}, r_t$, which predicts future states and rewards. A model can be approximated by a neural network or represented by a finite state machine. Such algorithms are called **model-based RL** and may improve the sampling efficiency. Several examples are the world model(Ha & Schmidhuber, 2018), MuZero (Schrittwieser et al., 2020), and EfficientZero (Ye, Liu, Kurutach, Abbeel, & Gao, 2021).

It is important to note that **learning a finite state machine** has been an old topic in computer science since the 1960s (Mohri, Rostamizadeh & Talwalkar, 2018). Whether it can be classified as

reinforcement learning remains to be a question. A state machine can be learnt passively or actively:

- Passive learning simply records states and transitions and can be easily integrated with RL algorithms like Q-Learning. An example is (Zheng et al., 2021) in web testing.
- Active learning chooses actions to infer the structure of state machines. An active learning algorithm called L* (Angluin, 1987) has been widely used in domains from network protocol inference to functional confirmation testing of circuits. However, L* requires a frequent reset to the initial state. L* only learns a model (state machine) and does not approximate a policy function or an action-value function.

2.3.3 Reinforcement Learning Algorithms in Software Testing

Table 2-2 summarizes some papers and projects about RL-based software testing. There are many other similar works in this field.

Table 2-2 Part of Papers and Projects on RL-based Software Testing

Paper	Field	RL Algorithm	Learn a Model of SUT	Testing Goal	Maturity
(Sant, Souter, & Greenwald, 2005)	Web Application Testing	Passive learning of state machine	Yes	Simply transfer user log to a model	Research at university
(Veanes, Roy, & Campbell, 2006)	Test programs with .NET languages	Like Q-Learning	No, it already has a (implicit) model written by developers	Action coverage (cover all actions in all states)	Research at Microsoft
(Bauersfeld & Vos, 2012)	GUI Testing	Q-Learning	No	Action coverage	Research at university
(Groce et al., 2012)	API Testing	SARSA	No	State coverage	Research at university
(Choi, Necula, & Sen, 2013)	Android GUI Testing	Active learning of state machine	Yes	State coverage	Research at university
(Reichstaller, Eberhardinger, Knapp, Reif, & Gehlen, 2016)	Interoperability testing	Q-Learning	No, it already has a model to calculate risk	Risk	Research at university
(Spieker, Gotlieb, Marijan, & Mossige, 2017)	Test Case Prioritization (Not test command generation, but has similarities)	Like DQN	No	Failure number of test cases	Research at company
(Su et al., 2017)	Android GUI Testing	Like Q-Learning (learn a model and then mutate it to get test cases)	Yes	State coverage (learn the model) Code coverage (mutate the model)	Research at university
(Adamo, Khan, Koppula, & Bryce, 2018)	Android GUI Testing	Q-Learning	No	Action coverage	Research at university
(Groz, Simao, Bremond, & Oriat, 2018)	Embedded System Testing (C++ microcontroller)	Active learning of state machine	Yes, up to 1000 states	Find all states	Research at university
(Vuong, & Takada, 2018)	Android GUI Testing	Q-Learning	No	Action coverage	Research at university

(Ahmad, Ashraf, Truscan, & Porres, 2019)	Stress Testing of Server	DDQN (a variant of DQN)	No	System response time	Research at university
(Dai, Li, Wang, Singh, Huang, & Kohli, 2019)	Test GUI and programs in domain specific languages	A2C with Gated Graph Neural Network (GGNN)	No	Code coverage	Research at Google DeepMind
(Zheng et al., 2019)	Game Testing	A2C+Evolutionary Algorithm	No	Game score and state coverage	Research at Netease and apply in real world
(Bergdahl, Gordillo, Tollmar, & Gisslén, 2020)	Game Testing	PPO	No	Game score and state coverage	Research at Electronic Arts
(Cai, Zhang, & Yang, 2020)	Android/IOS GUI Testing	UCB /Monte Carlo Tree Search /Q-Learning	Yes	Action coverage	Massively deployed in Bytedance
(Harries, 2020)	Windows GUI Testing	DQN with Graph Attention Network (GAT)	No	Multiple goals can be calculated	Research at Microsoft
(Pan, Huang, Wang, Zhang, & Li, 2020)	Android GUI Testing	Q-Learning with a curiosity module	No	Action coverage	Research at university
(Bagherzadeh, Kahani, & Briand, 2021)	Test Case Prioritization (Not test command generation, but has similarities)	Compare DQN, DDPG, A2C, ACER, ACKTR, TD3, SAC, PPO1/2, TRPO. Also uses (Spieker et al., 2017) as a baseline	No	Failure number of test cases, execution time of test cases, deviation from optimal solution of datasets	Research at university
(Lodge, 2021)	Java Unit Testing	The design details of the Diffblue Cover tool are unknown. They claim to use RL that can work on a developer laptop with 8GB memory and 2 Intel CPU core.			Commercial RL testing tool
(Moghadam et al., 2021)	Stress Testing	Q-Learning/DQN	No	Maximize response time and error rate	Research at institute
(Schwartz, & Kurniawati, 2021)	Penetration Testing	Q-Learning/DQN	No	Minimize exploit cost and find more sensitive machines	Research at university
(Tran et al., 2021)	Penetration Testing	Hierarchical DDQN (a variant of DQN)	No	Attack every host and finally find the flag	Research at university
(Test.AI, 2021)	GUI Testing	The details of the tool are unknown, but they claim to use RL which looks like Q-Learning based GUI testing mentioned above. The tool uses image recognition techniques to detect labels and icons of GUI.			Commercial RL testing tool
(Zheng et al., 2021)	Web Testing	Q-Learning with a curiosity module	Yes	Action coverage	Research at university
(Romdhana, Merlo, Ceccato, & Tonella, 2022)	Android GUI Testing	Compare Q-learning, DDPG, SAC, TD3	No, but it has a state machine written by developers	Action coverage and number of crashes	Research at university

In Table 2-2, the most popular approach to RL-based testing is using the Q-Learning algorithm to perform GUI testing. There are several reasons:

- As mentioned in section 2.3.1, it's easy to extract state information from GUI. It can be easily achieved by image recognition or tools like Automator.
- For a GUI application, the number of states and actions is usually limited if you properly filter equivalent states and actions. For example, (Choi, Necula, & Sen, 2013) tested 10 Android applications and could only find up to ~200 states in an APP. In this situation, it's suitable to use simple RL algorithms with tables as approximators.

RL algorithms with neural network approximators, or Deep Reinforcement Learning (DRL), came to the software testing field around 2017. (Spieker et al., 2017) was a remarkable success because it just used a very simple neural network (with one hidden layer) and achieved good performance on its dataset. (Bagherzadeh et al., 2021) Compared multiple DRL algorithms and found trust region methods like PPO, TRPO and ACER performed well in their settings. (Dai et al., 2019) and (Harries, 2020) were the first works to use graphs to represent states and use a graph neural network to process the states.

It's interesting that these papers only learn finite state machine models of the system under test (SUT) instead of neural network models. Two survey papers (Durelli et al., 2019) and (Omri & Sinz, 2021) showed that some researchers did model the SUT with neural networks, but no one combined it with reinforcement learning. This may be a research gap.

Several RL-based testing tools have been deployed since 2020. However, their design details are mostly unknown to the academic community. An exemption is the Fastbot of Bytedance (Cai et al., 2020), which uses Q-Learning and image recognition to perform GUI tests.

2.4. Formulate the Research

This work aims to find a way to test onboard software automatically with limited prior knowledge. Previous sections show random testing, and the genetic algorithm are the traditional ways to do that, but both have some disadvantages. Reinforcement learning, as a new approach in software testing, may solve these problems.

Based on the discussion above, this section formally formulates the research with assumptions, research questions, and opposite opinions.

2.4.1 Assumptions

Assumption-1 The onboard software testing process can be seen as an MDP, i.e., it satisfies the Markov property, and all state variables are observable.

The performance of the RL algorithm can verify this assumption.

Assumption-2 Performance of software testing algorithms can be measured by metrics like code coverage (in correctness testing) and CPU load (in stress testing).

This assumption is difficult to verify. There is no direct relation between test coverage and the number of identified bugs. However, most of the papers in this field used this assumption.

Assumption-3 **Prior knowledge** in software testing is human-defined problem-specific knowledge, such as:

- How to encode/decode commands and telemetry
- Rules are used to identify the current state from the telemetry or the interaction history.
- What action should be taken in the current state.
- Whether the current state contains an anomaly.
- A model is used to predict the system's future state under test.
- Design of the objective function or the reward function.

On the other hand, problem-independent knowledge, like the algorithms in the testing tools, is not thought of as prior knowledge in this study. Hyperparameters in the algorithms can be tuned by grid search or other methods, so they are not prior knowledge.

This assumption is intuitively valid.

Assumption-4 The amount of prior knowledge cannot be directly measured. However, we can list types of prior knowledge in a testing method and compare them based on experience.

Assumption-5 Other onboard software, especially software of CubeSats and PocketQubes, share similarities with Delfi-PQ flight software. Therefore, the conclusions and recommendations of this study can be partly generalized.

This assumption will be further discussed in chapter 3.

2.4.2 Research Questions

The main research question is

Can a reinforcement learning-based testing tool generate testing commands for small satellites with limited prior knowledge?

We divide it into the following sub-questions:

RQ-1 What is the goal of testing command generation?

As mentioned above, the testing goal can be maximizing state/action coverage, code coverage, system response time, or the number of failures. Since reinforcement learning uses a reward function to guide the search, we can measure the testing goal quantitatively.

RQ-2 What type of prior knowledge needs to be encoded?

Assumption-3 lists several types of prior knowledge.

RQ-3 Which RL algorithm is suitable for testing command generation?

It includes the selection of the RL algorithm and the implementation details, including the toolchain, representations of states and actions, and the neural network architecture.

RQ-4 What kind of testing environment should be used?

It includes the hardware and software settings of the test, such as communication between the host computer and the tested board, testing multiple boards simultaneously, and whether a simulator is needed. Note that we must ensure the testing commands will not damage the satellite.

RQ-5 Can we use a trained RL agent to test other software versions?

Testers may want to reuse an RL agent in regression testing. This study will measure the performance of such reuse and compare it with training an agent from scratch (cold start).

2.5. Brief Summary of the Chapter

- Compared with conventional software testing, embedded software testing has a series of challenges because of limited hardware capability and communication. Therefore, **simulation-based testing** is widely adopted. However, it is difficult to 100% mimic the behaviour of actual hardware.
- Onboard software testing is a type of embedded software testing. Some engineers use simulation or a **virtual satellite** to test the onboard software. Unfortunately, this approach is usually infeasible for small satellite programs with a limited budget. Another trend in onboard software development is **model-based automated code generation**, which makes the software safer and model-based testing easier. Moreover, **model-checking** tries to prove the SUT has the correct behaviour instead of testing it. Model checking requires a model of the SUT.
- Traditional automated testing approaches are **model-based testing**, **symbolic execution**, **random testing,** and **search-based testing**. It is convenient to use model-based testing to verify whether the SUT satisfies functional requirements. On the other hand, model-based testing heavily relies on prior knowledge specified in the models. At the same time, search-based testing is a powerful tool for verifying non-functional requirements and usually requires less prior knowledge.
- The most popular search-based testing algorithm is the **genetic algorithm**, but it has some disadvantages. It relies on the objective function to guide the search and discards other information from the execution. Furthermore, the time-consuming algorithm may need to be run repeatedly in regression testing.
- **Reinforcement learning-based testing** may compensate for the shortcomings of the genetic algorithm. It utilizes both the state and reward information during the testing. Moreover, it may learn a pattern of similar SUTs and not need to run again in regression testing. RL-based testing has been researched for about 15 years. In the recent 2~3 years, it has been deployed in production scenarios.
- No one has used reinforcement learning in onboard software testing yet. Few researchers use RL to test embedded software without a GUI. There are still some fundamental challenges in this field.

3 Testing Environment

This section introduces the testing environment used in the study. More specifically,

- Section 3.1 gives an overview of the functions of subsystems of Delfi-PQ, especially the hardware configuration.
- Section 3.2 introduces the onboard software of Delfi-PQ. It also compares the Delfi-PQ onboard software with open-source onboard software, e.g., NASA's core Flight System.
- Section 3.3 introduces the hardware and software tools used in testing.
- Section 3.4 explains how to extract information from the source code of onboard software, such as collecting code coverage, generating a graph representation of the program, and using the representation as input to a neural network.
- Section 3.5 is a summary of this chapter.

3.1. Overview of Delfi-PQ Subsystems

The first Delfi-PQ has 7 subsystems:

- On-Board Computer (OBC)
- Communication System (COMMS)
- Antenna Deployment Board (ADB)
- Electrical Power System (EPS)
- Attitude Determination and Control System (ADCS)
- A low frequency radio payload (LOBE-P)
- A redundant on-board computer.

The stack of these subsystems is shown in Figure 3-1.

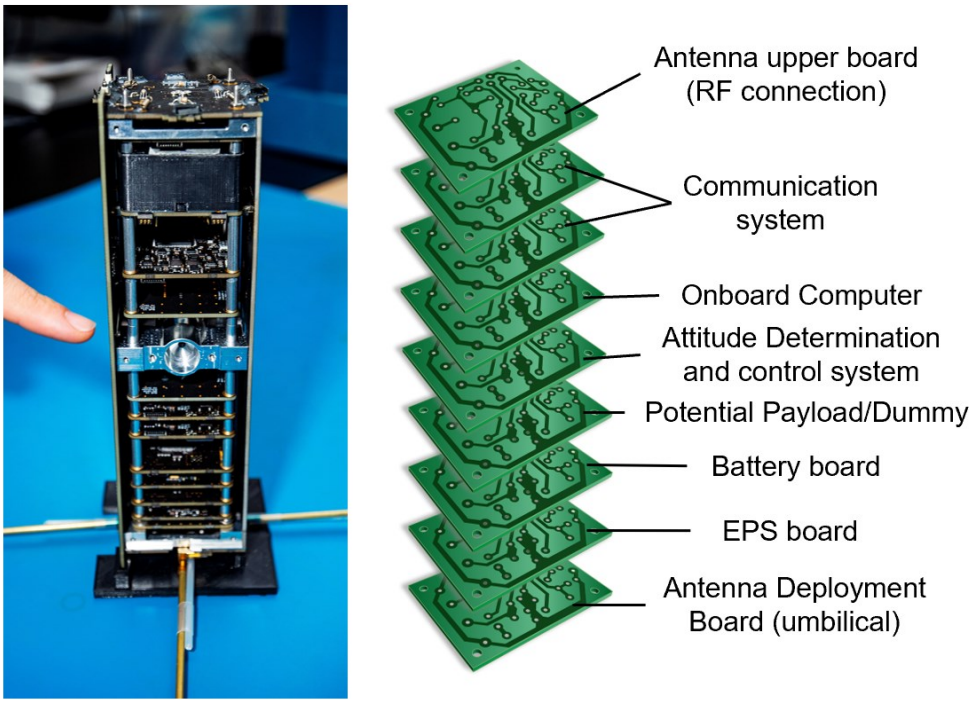


Figure 3-1 Stack of Delfi-PQ Subsystems

Each subsystem has a Texas Instrument MSP432P4111 microcontroller, which controls how the subsystem works. All these microcontrollers are connected to an RS-485 bus with a speed of 115.2kbps. The OBC is the master of the bus except during testing. Only the OBC can actively send frames over the bus, and other subsystems only reply passively. The bus only allows half-duplex communication; each frame has up to 253 bytes of payload. Each microcontroller also has SWD pins, which can be connected to a PC via a JLINK connector.

The 48MHz microcontroller has 2MB Flash and 256KB SRAM. There is also a 512KB FRAM for each microcontroller. Information in a FRAM will not be lost after a reset. However, only the OBC has a 2GB SD card to store telemetry.

Every microcontroller should kick an external watchdog on the board at least once every 2.5 seconds. Otherwise, the board will be reset. At the same time, a microcontroller should kick an internal watchdog at least once every 178 seconds. Otherwise, the controller will be reset. These are basic measures to deal with space radiation.

The OBC controls how the subsystems work. It has a state machine which covers the fundamental operations of the satellite. Figure 3-2 shows that the state machine has five modes: initial mode, antenna deployment mode, safe mode, ADCS mode, and normal operation mode. In the normal operations mode, the OBC will periodically request telemetry from every subsystem, save the telemetry in its SD card, and send it to the ground via COMMS. OBC will also periodically request ground commands from the COMMS. If the command is for OBC itself, it will deal with it and reply to the ground station. If the command is for another subsystem, OBC will forward the command to that subsystem, wait for the reply, and send a reply to the ground.

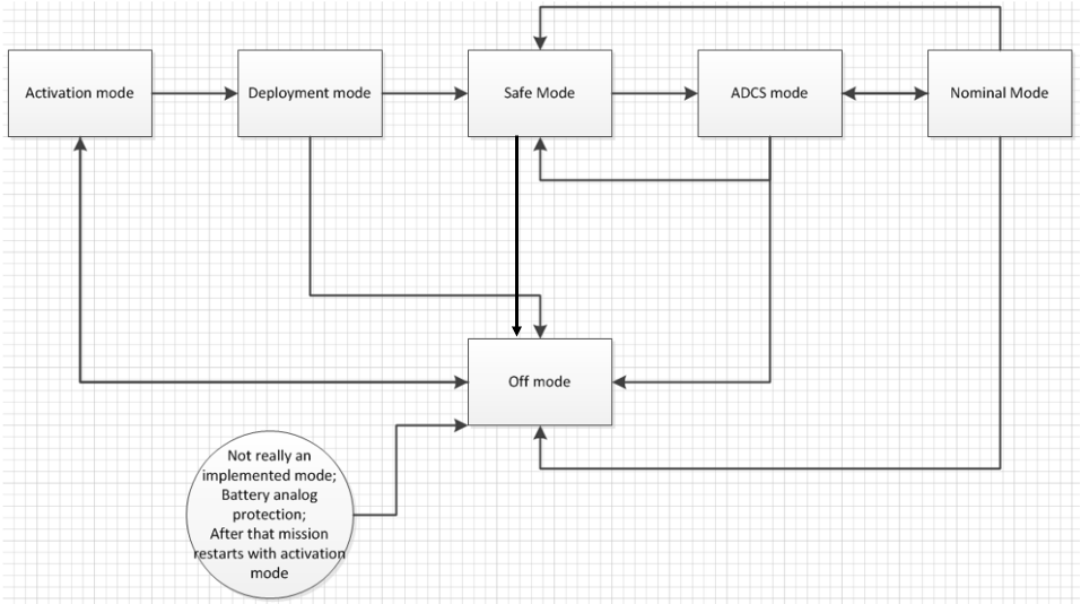


Figure 3-2: The State Machine in OBC

The EPS consists of the battery board (1500mAh, two batteries of 3.7V), the main EPS board and solar panels. EPS manages four power lines, and each line has some subsystems on it. According to commands from the OBC, EPS can enable, disable, reset, or power cycle a power line. EPS is designed to be constantly running after the deployer releases the Delfi-PQ. Therefore, a reset of the EPS leads to resetting the whole satellite. If the battery voltage is below 3.6V, the OBC should commend the EPS for disabling the power lines of unnecessary subsystems.

The COMMS receives and decodes the signal from the ground station. It automatically puts the ground commands into the RX (receiver) queue. If the OBC requests ground commands from the

queue, COMMS will take a ground command from the queue and send the command to the OBC. Furthermore, if the OBC needs to send a message to the ground station, it will command COMMS to put the message into the TX (transceiver) queue. COMMS will automatically send all messages in the TX queue to the ground. The RX/TX queue can store up to ~200 messages. The communication is full-duplex with a nominal speed of 1200 bps or a higher speed of 9600 bps at 2W power consumption. In an emergent case, if the COMMS receives a special command from the ground, it will raise a special line to reset the EPS, which will reset the whole satellite.

Another critical subsystem is the ADCS. It has an integrated sensor chip, including a gyroscope, an accelerometer, and a geomagnetic sensor. It also has three house-made coils as magnetometers to control the rotational speed of the satellite. If the rotation speed exceeds 5 deg/second, the OBC will command the ADCS to slow down the rotation.

ADB is used to deploy the antennas after the satellite is released from the deployer. The payload is another radio which will generate scientific data. The redundant onboard computer board only has an MSP432 and some essential components.

Table 3-1 summarizes part of the hardware of Delfi-PQ subsystems.

Table 3-1 Part of Hardware of Delfi-PQ Subsystems

Shared by every subsystem	Texas Instrument MSP432P4111 microcontroller	48MHz, 2MB Flash, 256KB SRAM, 65mW Internal watchdog period:178s
	Cypress CY15B104QN Ferroelectric RAM (FRAM)	512KB, controlled by on-board SPI bus
	Texas Instrument TMP100 temperature sensor	Controlled by on-board I2C bus
	Texas Instrument INA226 current and voltage sensor	Controlled by on-board I2C bus
	Texas Instrument TPS3813 external watchdog	Period: 2.5s
	STBB1-A DC-DC Converter	
	Analog Devices LTC4368 surge protector	
	Protection trip	
	Current limiting resistor	
	RS-485 inter-board bus	Speed: 115.2kbps Payload size of a frame: 256 bytes
	SWD pins for debug	
OBC specific hardware	SD card	2GB
EPS specific hardware	AW 16340 ICR123 750mAh battery * 2	1500mAh in total, 3.7V Integrated with protection circuits and gas gauges
	4 Solar panels	Each panel has an MPPT, a temperature sensor, and a voltage/current sensor Orbital average power: 1W
	4 Unregulated power lines	With monitoring + latch-up protection
COMMS specific hardware	Main radio board with SX1278 LoRa Module	Full control of the radio Multiple protocols Full duplex Data rate: 1200bps~9600bps
	RF front end with a power amplifier and a low-noise amplifier	Max power consumption: 2W
	Special line to reset EPS	
	UHF/VHF antennae	
ADCS specific hardware	Integrated sensor chip Bosch BMX-055	Include a gyroscope, an accelerometer, and a geomagnetic sensor

	Magnetorquer	With 3 house-made coils
Other hardware	A low frequency radio payload (LOBE-P)	
	A laser reflector	
	A board with MSP432 for students	It's the GPS board in the original plan, but the GPS module didn't work before integration
	Antenna Deployment Board (ADB)	

3.2. Onboard Software of Delfi-PQ

The onboard software has three parts:

- **Drivers**, i.e., driver functions of peripherals.
- **DelfiPQcore**, a lightweight operating system with some helper functions.
- **asks and services**. All subsystems share some tasks and services. On the other hand, some tasks and services are written for a specific subsystem.

The onboard software is edited, compiled, and loaded to the microcontroller by the Texas Instrument Code Composer Studio IDE. We use the old TI ARM C/C++ compiler (TI v20.2.0LTS), though TI has a new compiler based on Clang.

We will discuss the general workflow of the onboard software in section 3.2.1, the basic concepts of DelfiPQcore in section 3.2.2, and tasks and services in section 3.2.3. This section is based on internal reports by Stefano Speretta and Casper Broekhuizen.

3.2.1 Workflow of Onboard Software

The general workflow of the programs can be described as a sequence of initialization steps, after which the program will go into a continuous task loop, as shown in Figure 3-3. In this loop, it does the following:

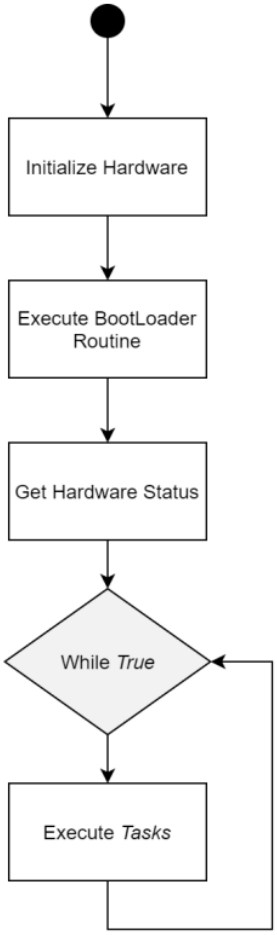


Figure 3-3: Workflow of Onboard Software of Delfi-PQ by Casper Broekhuizen

Step 1
Initialize Hardware

This step should initialize the critical components of the subsystem, including general hardware in the MCU and specific hardware for the subsystem.

Step 2
Execute Bootloader

One of the software's core features is the possibility of loading different software versions from the flash memory of the MCU.

Step 3
Get Hardware Status

Critical hardware status indicators should be collected and stored. These critical status indicators include the reset status (the reason for the last reboot) and possible clock faults.

Step 4
Execute Task

After the operating system has completed its boot steps, it starts a continuous task execution routine, which can be considered a simple non-pre-emptive, non-prioritized, linear scheduler (round-robin).

3.2.2 Important Concepts of DelfiPQCore

In this section, we introduce some basic concepts and functions of the DelfiPQCore, a straightforward operating system made in house.

Task

After the operating system starts, tasks can be executed. Any processing/data collection or other action executed by the device is a task.

A task consists of an initializer function, a function executed once during the initialization of the scheduler (called task manager), and a user function that executes every iteration of the task.

Every task has an execution flag. Raising this flag will tell the task manager that this task is ready for execution. If the execution flag is not raised, the task will not be executed and will be skipped by the task manager. The execution flag can be raised either externally by another task or using any interrupt routine. This action will henceforth be called "notifying a task".

PeriodicTask

Some tasks require periodic execution, and there might not be any clear external trigger available to notify such tasks (such as a telemetry collection task). Such a task is a periodic task, which includes another parameter which contains the required amount of 'counts' for the task to be notified (1 'count' is approximately 0.1 seconds). An external object, the task notifier, will notify the period tasks assigned to it in an interrupt routine.

Service

The most common source of notifying a task is from an external trigger over the satellite bus. The satellite bus driver will receive bytes over the bus using a hardware interrupt routine. If a complete frame is received, a command handler task will be notified, and copy the received frame into its buffer.

Then, the scheduler will execute the command handler task since its execution flag is raised. The command handler will read the data frame and 'poll' so-called services registered.

When a service detects that the received frame is for itself, it will process the received frame, set a response frame, and tell the command handler that the service has processed the received frame. The command handler will stop polling other services and reply over the bus.

A user should create a service for every functionality required over the satellite bus.

PQ9Frame

Though the command handler handles any frame it receives, the services used are frame-specific. The frame (PQ9Frame) is built in the following way:

Table 3-2 PQ9Frame Definition

Byte number	Description
0	Destination Address
1	Payload Size
2	Source Address
3 … N	Payload
N + 1	CRC1
N + 2	CRC2

Whereas the frame Payload is described as follows:

Table 3-3 PQ9Frame Payload Definition

Byte number	Description
0	Service Number
1	Action: - 0: ERROR - 1: REQUEST - 2: REPLY
2 … N	Service Payload

PQ9Frame has CRC verification.

Software Update

As mentioned earlier, one of the core functionalities of the DelfiPQcore is to execute a different software version from the FLASH. The bootloader handles this functionality. This bootloader requires an external memory (FRAM) that holds non-volatile information regarding which memory slot needs to be executed, whether the last execution was successful, and the number of reboots. If this information tells the bootloader that the target slot is broken or has issues (or if the external FRAM is unavailable), it will fall back on the default slot (Slot 0).

The device has three slots available, Slot 0, the default slot protected in the FLASH and cannot be reprogrammed, and Slot 1 & Slot 2, which can be reprogrammed. SoftwareUpdateService allows a binary file transfer of a new software version over the bus to reprogram the FLASH. Thus, a module can be reprogrammed externally and even in orbit. Note that the FRAM needs to be present for this functionality to work.

3.2.3 Tasks and Services in Each Subsystem

Table 3-4 summarizes the tasks and services of Delfi-PQ subsystems.

Table 3-4 Tasks and Services of Delfi-PQ subsystems

	Task	Service
DelfiPQcore (For all subsystems)	Timer task (a periodic task which collects telemetry of the subsystem every second)	Ping service (Reply to the Ping command)
	Command handler task (Process received frames and replies)	Reset service (Reset or power cycle the controller)
		FRAM service (Read, write, or erase the FRAM)
		Housekeeping service (Send the telemetry collected by the timer task as a response)
		Software update service (Handle software update commands, and new binary software is the payload of some software update commands)
Only for ADB	Burn task	Burn service (Burn the wire that locks the antenna, so the antenna is deployed)
Only for ADCS	None	Coil service (Set states of the magnetorquers)
Only for COMMS	CommRadio task	Radio service (A set of functions to interact with COMMS)
Only for EPS	None	Power bus handler (Set states of the power lines)
Only for LOBE-P	lobepRadio task (Like the CommRadio task, but work in lower frequency)	lobep service (Like the radio service, but working on lower frequency)
Only for OBC	State Machine task (A periodic task that runs the simple state machine in section 3.2.1)	State machine service (Get / set the current state; enable beacon or reset the state machine)
	File system task (Raised by the telemetry request service to retrieve telemetry from the SD card asynchronously, or raised by the state machine to store telemetry in the SD card)	Telemetry request service (Request telemetry file from the SD card or format the SD card)
		Bootloader override service (Command the microcontroller to jump to a specific slot. It should be a service shared by multiple subsystems but is only an OBC service at this moment.)

3.2.4 Safety Measurements in the Onboard Software

Delfi-PQ does not have complex fault-handling mechanisms. It only has some fundamental safety measurements, including:

- If there is no response from a subsystem, OBC will resend the command.
- The microcontroller kicks the external watchdog when receiving a PQ9Frame. The board will be reset if no frame is received, or the controller does not kick the watchdog during the 178 seconds. Therefore, if OBC does not send commands to EPS during the period, EPS will be reset, leading to a complete satellite reset.
- The microcontroller kicks the internal watchdog in the main loop. If the controller is stuck and does not kick the watchdog during the 2.5 seconds, the controller will be reset.
- With a special command from the ground station, COMMS can directly reset EPS via a special line.

- CRC checking of the PQ9Frame.
- MD5 checking of binary source code.
- Multiple software images in the flash of a microcontroller.
- Important program variables have backups in the FRAM.
- We carefully design the state machine of OBC, which includes sensor check and a degraded safe mode.

For more information about radiation hardening and safety measurements on CubeSat/PocketQube platforms, the readers can refer to (Yuen & Sima, 2019).

3.2.5 Compare with Other Onboard Software

The Delfi-PQ software has a similar architecture to the onboard software of the bigger satellites. For example, the NASA core Flight System (cFS) has three layers: user applications, the core Flight Executive (cFE), and the platform abstraction layer (McComas, 2021). The Delfi-PQ has three similar layers: user applications (subsystem-specific tasks & services), the DelfiPQcore, and the hardware abstraction layer (drivers).

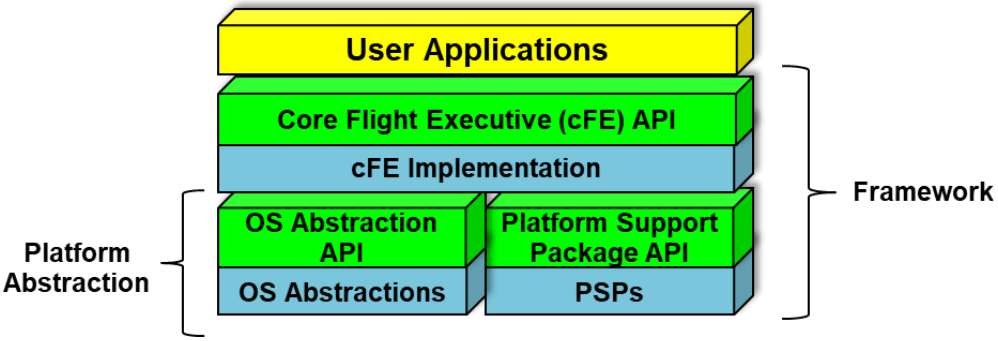


Figure 3-4 Architecture of NASA Core Flight System (McComas, 2021)

The DelfiPQcore offers some functionalities of a real-time operating system and the cFE. It provides a bootloader, task creation and scheduling, bus command handling, and software update capability. Although operating systems like FreeRTOS can support more, such as dynamic memory allocating and queues for inter-task communication, the current functionalities of DelfiPQcore are enough to use in a PocketQube mission.

Table 3-5 compares the tasks of Delfi-PQ with the open-source set of cFS applications (Timmons, 2020). The Delfi-PQ tasks and services offer some basic functionalities of the cFS applications, like command uplink and telemetry downlink. However, it does not support fault handling, memory integrity checking, or processor address sampling. The Delfi-PQ software is also incompatible with the standards of the CCSDS committee, which can be very time-consuming. cFS has many mission-specific applications, such as data processing, attitude control, navigation, and instrument calibration, which are much more complex than current Delfi-PQ applications.

Table 3-5 Compare cFS Open-Source Applications with Delfi-PQ Applications

Application in cFS	Explanation	Similar Functionality in Delfi-PQ
Health and Safety App	Kick watchdog, monitor applications and events, take table-defined actions	Kick watchdogs, but do not monitor tasks and take recovery actions
Housekeeping App	Collects and re-packages telemetry from other applications	Housekeeping service
Data Storage App	Record housekeeping, engineering. and science data onboard for downlink	OBC state machine does it
File Manager App	Interfaces to the ground for managing files	No

Limit Checker App	Compare the telemetry with thresholds and take table-defined actions	OBC state machine checks some telemetry parameters
Memory Dwell App	Sample data at any process address	No
Scheduler App	Schedule onboard applications	DelfiPQcore scheduler can do it
Stored Command App	Executes preloaded command sequences at predetermined time	No. Delfi-PQ execute all ground commands immediately
Software Bus Network	Passes Software Bus messages over various “plug-in” network protocols	No. Delfi-PQ has some dirty ways for inter-task communication, such as call back functions and global viriables
Checksum APP	Performs data integrity checking of memory, tables, and files	Delfi-PQ will check MD5 of binary code during software update, but it won’t check that in normal operation
Memory Manager App	Provides the ability to load and dump memory	No
CFDP App	Transfers/receives file data to/from the ground according to CCSDS CFDF protocol	The telemetry service can transfer telemetry data from the SD card to the ground station. It doesn’t follow the CFDF protocol
Command Ingest Lab	Accepts CCSDS telecommand packets over a UDP/IP port	OBC state machine and the radio service accept ground commands. It doesn’t follow the CCSDS standard
Telemetry Output Lab	Sends CCSDS telemetry packets over a UDP/IP port	OBC state machine and the radio service send telemetry to the ground. It doesn’t follow the CCSDS standard.

We can also look at onboard software developed by other universities. California Polytechnic State University (Cal Poly) was one of the universities that proposed the CubeSat standard. Their first-generation onboard software was developed from scratch and had similar functionalities to Delfi-PQ. However, this software has no hierarchy, and all source code is put in a single file. Their second-generation software was based on Linux (Manyak, 2011). Compared with the Delfi-PQ software, it only adds a system manager (like cFS Health and Safety App) and inter-process communication API (based on Linux UDP/IP tool).

Table 3-6 compares the lines of code of several open-source onboard software repositories (measured by cloc), which is a metric to measure their complexity. Note that this metric is inaccurate because some source codes may be duplicated.

Table 3-6. Lines of Code of Some Open-Source Onboard Software Repositories

Onboard Software Repository	Number of Lines of Code	Languages
Delfi-PQ Flight Software[12]	20107	C, C++
NASA Core Flight System[13]	100132	C, C++, Python, Perl
NASA JPL F Prime[14]	82915	C, C++, Python
KubOS[15]	57968	Rust, C, Python
ESA Nanosat-MO-Framework[16]	543396	Java
Cal Poly libproc[17] for CubeSat	25649	C, C++
FossaSat-1 Pocosatellite[18]	1534	C++
EASAT-2 PocketQube[19]	1816	C
MelbourneSpaceProgram[20]	21668	C, C++

[12] https://github.com/DelfiSpace/FlightSoftwareWorkspace
[13] https://github.com/nasa/cFS
[14] https://github.com/nasa/fprime
[15] https://github.com/kubos/kubos
[16] https://github.com/esa/nanosat-mo-framework
[17] https://github.com/PolySat/libproc
[18] https://github.com/FOSSASystems/FOSSASAT-1
[19] https://github.com/AMSAT-EA/easat-2
[20] https://github.com/MelbourneSpaceProgram/msp_flight_software_public

As a summary of this section, the Delfi-PQ onboard software is simpler than professional onboard software made by space agencies. However, it has a similar complexity to other educational CubeSats/PocketQubes and is even more complex than others.

Let us look at them using a hierarchical perspective. The DelfiPQcore provides similar capabilities to the professional onboard software, though it does not have a formal inter-task communication function or complex fault-handling mechanism. On the other hand, in the application layer, Delfi-PQ software is much simpler than professional onboard software.

As an educational program, Delfi-PQ does not strictly meet relevant standards. It makes the implementation simpler.

3.3. Test Set-Up

In this section, the hardware and software set-up used in this research will be discussed.

3.3.1 Hardware Set-Up

The subsystem boards can be placed on the Electrical Ground Support Equipment (EGSE) board during testing. The EGSE board transfers messages between the RS485 bus (of Delfi-PQ) and the USB stream. Moreover, the lab computer is also connected to the SWD pins of the microcontrollers on the subsystem boards. The SWD connection can be used to modify the onboard software code.

This study does not use an RF checkbox to communicate with the subsystems wirelessly. However, as shown in Figure 3-5, TX and RX antennae are connected to the COMMS board. If the transceiver and receiver in COMMS are set to the same data rate and frequency, the COMMS can “hear” the signal sent by itself and check the wireless communication channel.

The experiments can be conducted remotely. The testing command generation tool can run on the lab computer with a Windows 10 environment. The lab computer has an Intel E5-1620 CPU (released in 2014) with 8GB RAM. The computing power of the lab PC is low, but the communication speed between the lab PC and the EGSE board is quick.

On the other hand, the command generation tool can also run on cloud servers. Some experiments use a cloud server of Alibaba Cloud in Frankfurt. The server has an Intel Xeon 8163 CPU (released in 2017) with 31GB RAM and an Nvidia T4 GPU with 16GB memory (released in 2018). A ngrok server and the lab PC will transfer messages between the EGSE board and the cloud server. The computing power of the cloud server is relatively low, but the communication speed is slightly slower (shown in Figure 3-13).

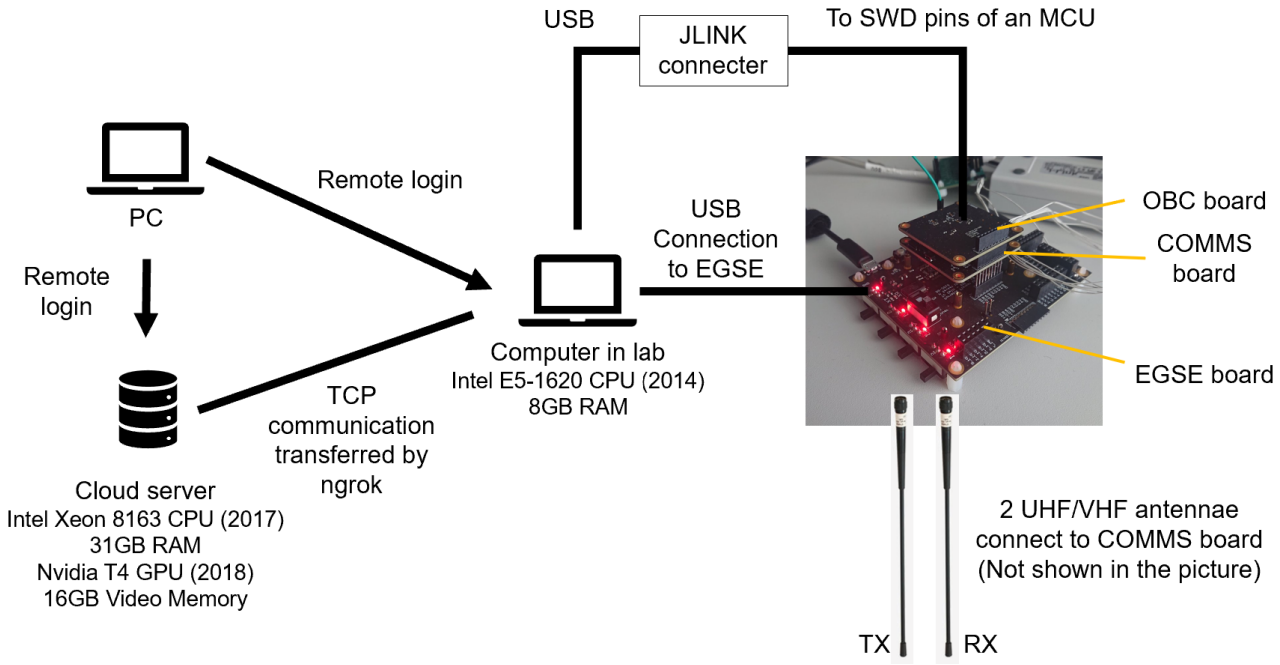


Figure 3-5: Hardware Set-up for the Testing

There are two ways for the lab computer to put a command over the bus:

- **Single subsystem testing**. The subsystem under test is put on the EGSE. The lab computer sends commands to the subsystem and gets responses via the EGSE. When testing the OBC, the OBC software is set to a “passive” mode, i.e., it waits for external commands rather than actively sends commands over the bus.
- **Multiple subsystems testing**. Several subsystems are put on the EGSE, and the OBC is the master of the bus. The testing tool mimics the behaviours of the COMMS subsystem, so it also has an RX and TX queue. The tool puts the “fake ground command” in the RX queue to send a command to a specific subsystem. The OBC will periodically request commands in the RX queue and transfer them to destinations according to the heads of the frames. After that, the OBC will collect replies to these frames and put them in the TX queue, which will be read by the testing tool.

Limited by time and resources, this work only tests the COMMS and OBC in the “single subsystem testing” scenario.

3.3.2 Software Set-Up

The basic software set-up is shown in Figure 3-6. The decision-making, IO processing, and code coverage collection components need to be implemented in this research.

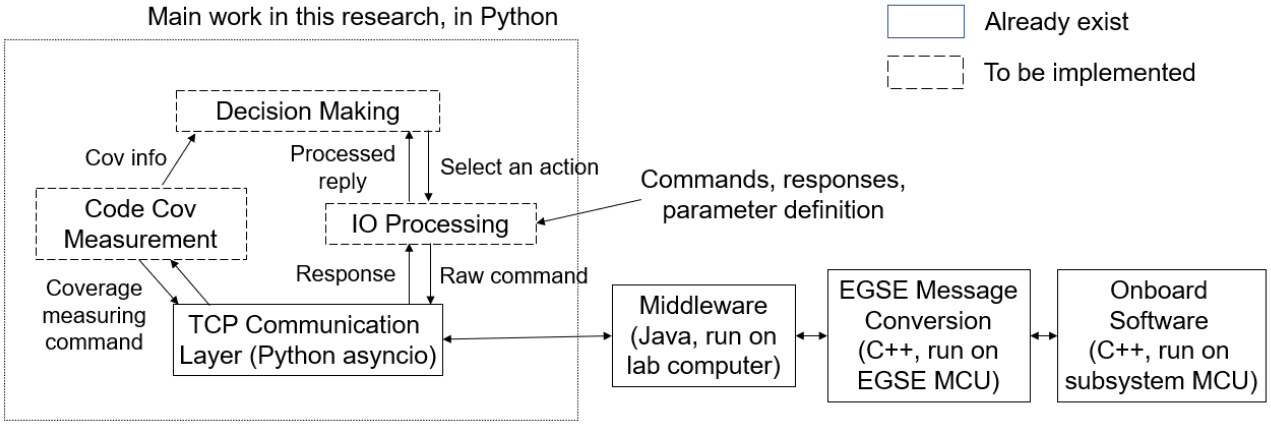


Figure 3-6: Software Set-up for the Testing

The Delfi-PQ team specifies the telemetry of Delfi-PQ in an XML file according to the XTCE standard (CCSDS 660). We can also use an open-source XTCETOOLS[21] to visualize the XML file. However, the XTCE standard does not include telecommand definitions. It is also challenging to use the XTCETOOLS to edit the XML file. Therefore, SUT parameters, telecommands, and responses are specified in CSV files in this study, as shown in Figures 3-7, Figure 3-8, and Figure 3-9.

Space System Name	Name	Type	Unit	Size	Format	Min Value	Max Value	Select From	Enum
/Delfi-PQ	AllowResume	ENUMERATED		8	unsigned			[0,1]	{"0":"No Resume", "1":"Allow Resume"}
/Delfi-PQ/COMMS	AmplifierCurrent	FLOAT32	A	16	twosComplement	0	1.5	Reply Only	
/Delfi-PQ/COMMS	AmplifierINAStatus	ENUMERATED		1	unsigned			Reply Only	{"0":"Fault", "1": "Normal"}
/Delfi-PQ/COMMS	AmplifierTemperature	FLOAT32	°C	16	twosComplement	-60	130	Reply Only	
/Delfi-PQ/COMMS	AmplifierTMPStatus	ENUMERATED		1	unsigned			Reply Only	{"0":"Fault", "1": "Normal"}
/Delfi-PQ/COMMS	AmplifierVoltage	FLOAT32	V	16	unsigned	0	6	Reply Only	

Figure 3-7: An Example of SUT Parameter Definition

Space System Name	Service	Command Name	Field Type	Parameter	Select From	Size
/Delfi-PQ/COMMS	Radio	SetTXBitRate	Command			
			Argument	Service	[25]	8
			Argument	Request	[1]	8
			Argument	SingleByte	[14]	8
			Argument	TXBitRate	[1200,9600]	16

Figure 3-8: An Example of Telecommand Definition

Space System Name	Service	Command Name	Field Type	Parameter Name
/Delfi-PQ/COMMS	Radio	GetRX_RSSI	Reply	
			Argument	Service
			Argument	Request
			Argument	RadioReply
			Argument	ReceiverRSSI

Figure 3-9: An Example of Response Definition

Based on these definitions, an **IO processing** module is implemented in this work. It includes a parser and a state identifier, as shown in Figure 3-6. The parser maintains a list of all telecommands. The decision-making module can select a command (e.g., the 12nd command) and send it over the bus. According to the previous command and the raw response from the bus, the parser can translate the response into a dictionary of interpretable parameters. After that, a state identifier will update the current system state, according to human-defined rules.

[21] https://gitlab.com/dovereem/xtcetools

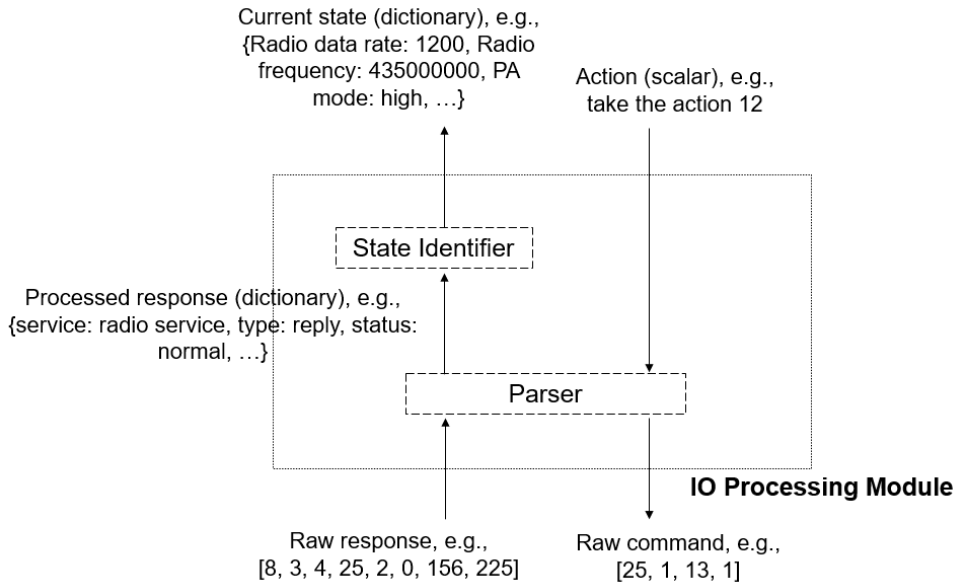


Figure 3-10: The IO Processing Module
(Some algorithms in this work only use part of this module or even do not use it)

The IO processing module contains much prior knowledge, including:

- The definitions of parameters, telecommands and responses used by the parser.
- The human-defined rules used by the state identifier.

Too much prior knowledge is undesired in this work. To reduce the use of prior knowledge, some algorithms in this study only use part of the IO processing module or even do not use it at all. The following section will discuss how to extract information from the source code of the onboard software, which does not need human-defined prior knowledge.

3.4. Extract Information from Source Code of Onboard Software

This section discusses collecting code coverage and generating a graph representation of the source code. As mentioned in chapter 2, code coverage can represent a current state or evaluate the amount of testing. On the other hand, code coverage can be added to the graph representation and become input to a graph neural network.

3.4.1 Code Coverage Collection

There are several types of code coverage metrics (Pani, 2014), such as:

- **Line coverage**. 100% line coverage means covering every line of the source code.
- **Branch coverage**. 100% branch coverage means covering every branch of the source code.
- **Condition coverage** (or **decision coverage**). 100% condition coverage means every condition has been True and False at least once.
- **Modified condition/decision coverage** (MC/DC coverage). 100% MC/DC coverage means every condition has been True & False for at least one, and its value determines the result independently.

Among these metrics, measuring line coverage takes more memory footprint because it needs to record the execution status of every line of code. MC/DC coverage is also complex to measure. By

contrast, branch coverage is relatively easy to measure since it just needs to record whether branched are executed.

Some safety-critical software requires all these metrics to reach 100%. For example, in the ECSS-E-ST-40C standard (2009), suppliers of space system software with the criticality "A" must achieve 100% statement coverage, decision coverage and MC/DC coverage, as shown in Table 3-7.

Table 3-7. Code Coverage Requirement in ECSS-E-ST-40C (2009)

b. The supplier shall verify that the following code coverage is achieved

Code coverage versus criticality category	A	B	C	D
Source code statement coverage	100%	100%	AM	AM
Source code decision coverage	100%	100%	AM	AM
Source code modified condition and decision coverage	100%	AM	AM	AM
NOTE: "AM" means that the value is agreed with the customer and measured as per ECSS-Q-ST-80 clause 6.3.5.2.				

Code coverage collection can be intrusive or non-intrusive (Pani, 2014). **Intrusive measurement** needs **instrumentation**, i.e., adding additional program code that does not change the behaviours of SUT. Such additional code can be added to the source, assembly, or binary code. By contrast, **non-intrusive measurement** utilizes the tracing capability of processors and usually needs special hardware. Intrusive measurement usually has more influence on the performance of SUT.

Several commercial off-the-shelf coverage measurement tools were tried for Delfi-PQ:

- Theoretically, the old TI ARM C/C++ compiler supports branch coverage collection. It will instrument the source code during compilation if an option in the Code Composer Studio is selected. When executing a particular command in the source code, a code coverage file will be sent from the microcontroller to the host PC. The compiler must be rerun to transfer the code coverage file to a readable report.

  However, this coverage collection functionality does not work for Delfi-PQ flight software. Even if it can work, it needs to manually rerun the compiler to get a coverage report, which is not convenient. The new TI Clang compiler does better in coverage collection, but we do not want to migrate to a new compiler.

- Segger J-Trace Pro hardware supports real-time non-intrusive code coverage collection. However, the MSP432 P series microcontrollers do not support tracing, which is necessary for J-Trace.

- Suppliers of other coverage measurement tools were also consulted. However, their prices are too high. In 2021, a Tessy license cost €8000, an LDRAcover cost €12000, and a VectorCAST cost €15000. Such high prices are not affordable for a thesis.

To solve the problem, a simple Python tool called pq9cov[22] has been implemented. It's inspired by a straightforward coverage collection tool GeCov[23]. Figure 3-11 briefly shows how pq9cov works.

[22] https://github.com/StarCycle/CodeCoverage
[23] https://github.com/EDI-Systems/G2T01_GeCov

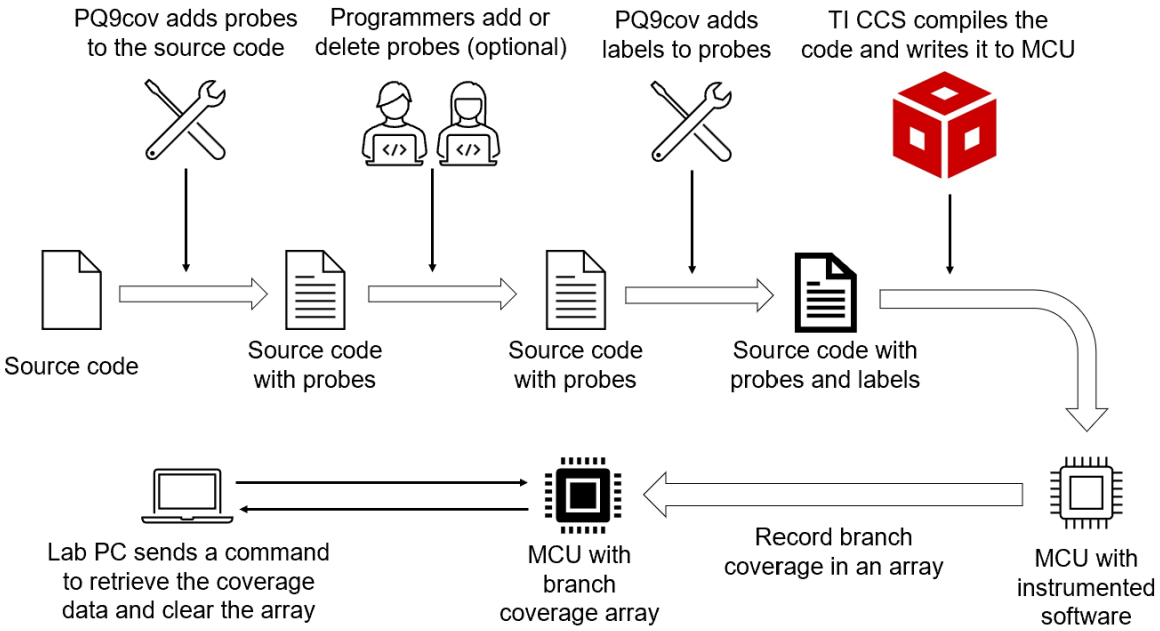


Figure 3-11: How PQ9cov Works

Testers first use pq9cov to add probes at the entry point of every branch. A probe is a function called CodeCount(). CodeCount is a global function to set a bit of the code coverage array to 1. For example, CodeCount(66) will set the 66th bit of the array to 1. Note that this step has no input parameter in the CodeCount().

PQ9cov automatically identifies entry points of branches. Traditional coverage tools achieve this by static code analysis, i.e., building abstract syntax trees of the source code. This approach is relatively complex. Hence, pq9cov use carefully designed regular expressions to locate conditional statements such as “if”, “for”, and “while”.

Nevertheless, regular expressions are still inaccurate and may ignore some edge cases. Testers can delete or add CodeCount() in the source code to compensate for this shortcoming. This step is optional. For Delfi-PQ flight software, such manual modification is usually not needed.

Then, testers need to use pq9cov to add labels to these CodeCount(). It will transfer CodeCount() to CodeCount(n), where n is the label of the entry point. After that, the instrumented source code will be compiled and run in the target MCU. During execution, the CodeCount(n) function calls will record branch coverage in an array. Testers can retrieve the array from the MCU with a special command or other ways.

Based on the code coverage array and the instrumented source code, pq9cov can generate a code coverage report in a CSV file and visualize the coverage result in the source code. Testers can also use pq9cov to automatically remove all probes in the source code.

Figure 3-12 shows the instrumentation process in detail.

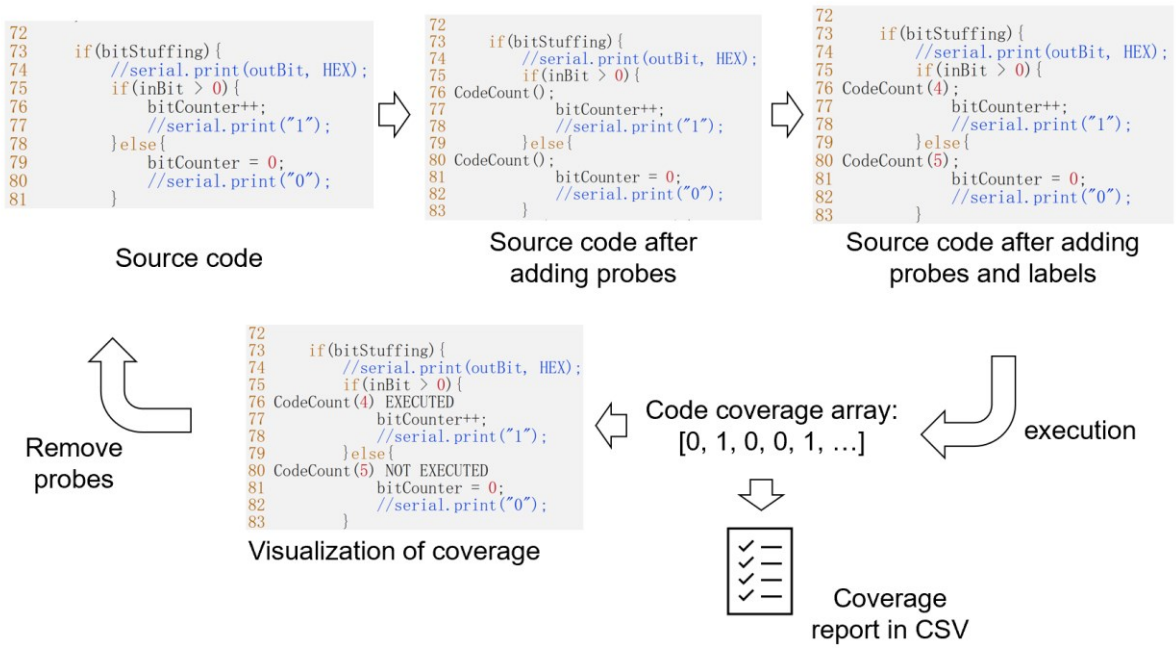


Figure 3-12: Details of Instrumentation in PQ9cov

PQ9cov only takes a small memory footprint. Each probe result is stored in 1 bit of the array. For onboard software of the COMMS and the OBC, there are about 716 probes (take 90 bytes) and 1329 probes (take 167 bytes), respectively. It is easy to store the code coverage array in an MSP432 MCU with 2MB SRAM and retrieve it by a PQ9Frame with 253 bytes of payload size.

Transmission of the array takes more time than the expected response. This is because the size of a typical payload is smaller than the size of the coverage array. Figure 3-13 shows the response time of a typical command and a coverage collection command. Nevertheless, the transmission time is still acceptable.

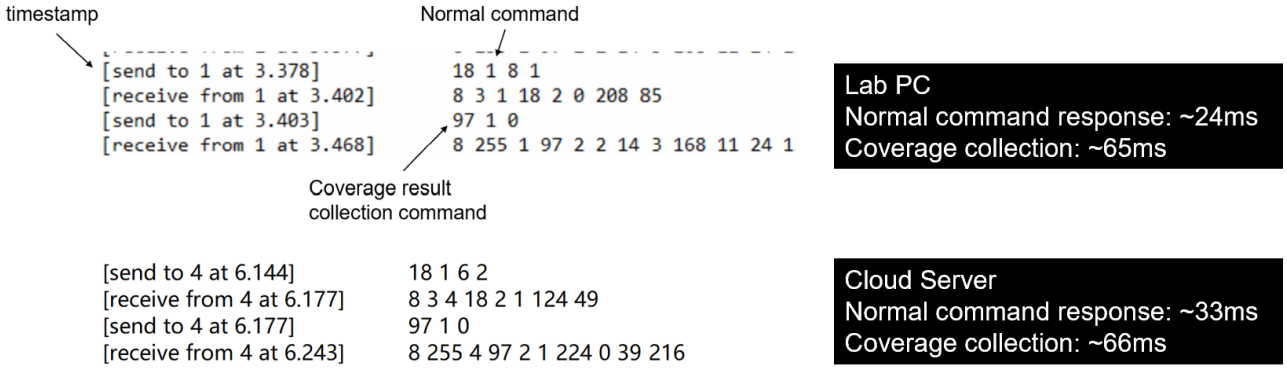


Figure 3-13: Response Time of a Normal Command and a Coverage Collection Command

Although the memory footprint and the transmission time are acceptable, this approach is still intrusive. The instrumented program needs to call CodeCount() at the start of every branch, which makes the program run slower. Such influence is difficult to measure, but we did observe it. For example, the instrumented COMMS software missed some commands from the lab PC, which was rare for non-instrumented software.

PQ9cov supports instrumentation of C/C++ source code but can adapt to other languages quickly. It only has ~150 lines of code to achieve all functions mentioned above. Table 3-8 summarizes the Pros and Cons of the coverage tool.

Table 3-8: Pros and Cons of PQ9cov

Pros	Cons
Adapt to any MCU with slight modification	Only support branch coverage
Testers can modify the probes before labelling	Inaccurate instrumentation
Simple (~150 lines) and easy to understand. You can modify the regular expressions to adapt to other languages	Only support C/C++ now
Low memory footprint and collection time	Calling CodeCount() makes the program slower. Such influence is difficult to measure.
Coverage visualization in source code & coverage report	

3.4.2 Several Ways to Feed Code Coverage into Neural Networks

After we collect code coverage, we need to feed it into a neural network. There are several ways to achieve this: use a plain coverage vector as input, use the source code with coverage result as input, or combine the graph representation of the program with coverage result as input. Figure 3-14 briefly explains these three ideas.

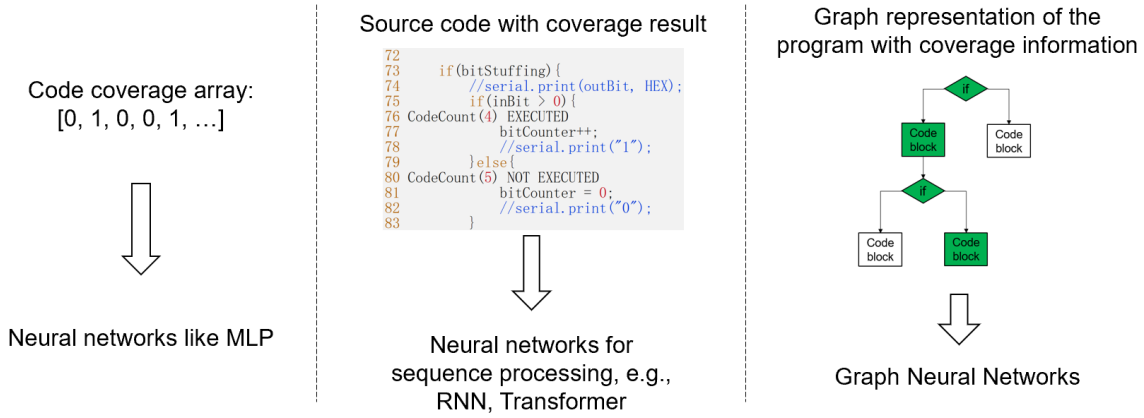


Figure 3-14: 3 Ways to Use Code Coverage as Neural Network Input

Using a plain vector as input is the most straightforward idea, but it also has a primary challenge. As explained by Figure 3-15, if programmers modify the source code of SUT and instrument it again, they may find that the number and order of the probes are changed. That is to say, the length of the coverage vector, and the corresponding branch of every element in the vector, are both changed after source code modification. As a result, the original neural network becomes useless.

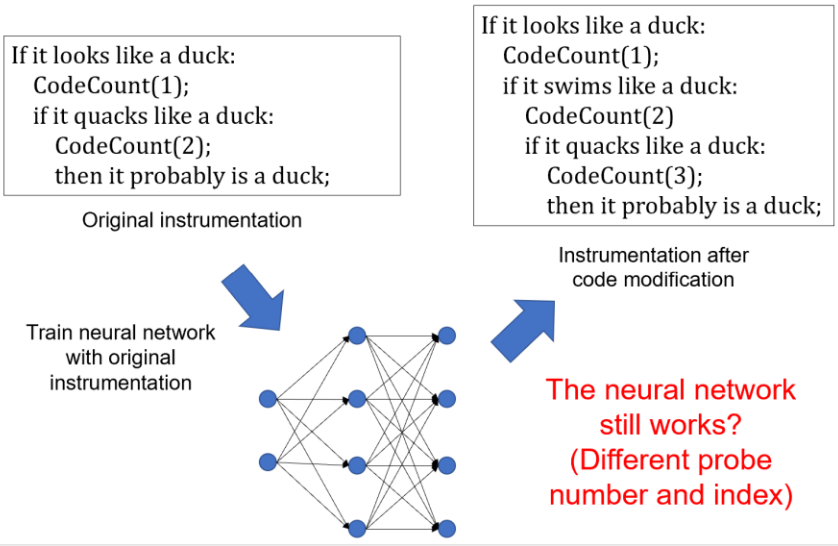


Figure 3-15: A Challenge of Using Coverage Vector as Input to a Neural Network

To make a trained neural network reusable after source code modification, we can put coverage results in the source code, as shown in Figure 3-15. Then the neural network directly uses the source code files as input. Nevertheless, such input can be very long and challenging to process.

The third idea is to use a graph to represent the source code of SUT and then combine the coverage information with the graph. Theoretically, a trained graph neural network can be reused for similar programs with similar graph representations. (Dai et al., 2019) took this approach in RL-based testing. However, they only tested some toy programs in domain-specific languages and did not explain how to build the graph.

This study will try the first approach (plain vector) and the third approach (graph) to generate input for neural networks. In the following sections, the readers can see how to generate a graph representation of a C/C++ program and combine it with code coverage information.

3.4.3 Extract Graph Representations of Programs from Execution Traces

We hope to generate a directed graph of the program under test. Some nodes of the graph represent probes, i.e., CodeCount(i). Furthermore, the graph's directed edges show the nodes' relations. Each node should have a feature vector and a coverage mask, indicating which nodes have been covered.

The straightforward idea is to generate the graph during the program's execution. Apart from writing the coverage array, the CodeCount(n) function call can also record the transition from the previous probe to the current probe. In the end, testers can build a graph with these nodes and transitions. Figure 3-16 explains this idea in detail: the instrumented program maintains an array to record the transition from the previous CodeCount() to the current CodeCount(). This array is sampled periodically, and new transitions will be added to the graph.

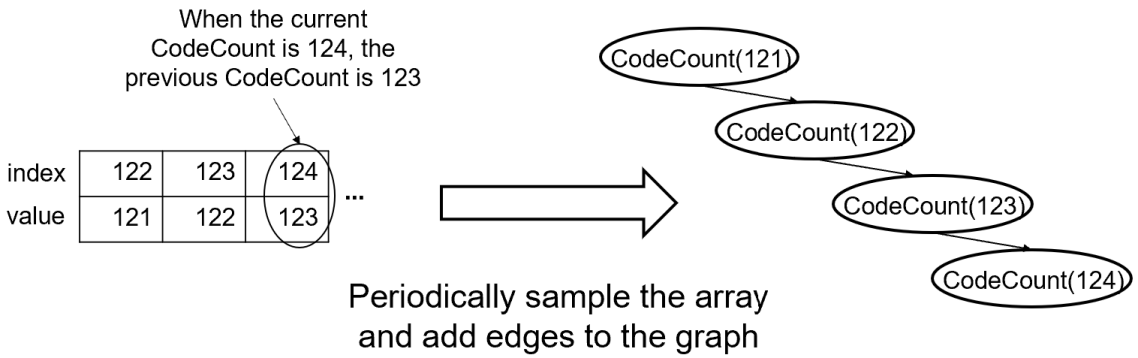


Figure 3-16: Generate the Graph Representation during Program Execution

This approach has been implemented. Figure 3-17 shows the generated graph for the COMMS software. In that version of COMMS software, there are 716 CodeCount probes, but only 299 are executed. Hence, there are only 299 nodes in the graph.

Moreover, this method's communication load and memory footprint are more severe. The transition array is larger than the coverage array because every element is a LONG variable with 4 bytes. Assuming there are 2024 probes in the source code, the size of the coverage array will be 2024/8=253 bytes, while the transition array will have 2024*4=8096 bytes. In the end, we must use multiple frames to retrieve the transition array to the lab PC.

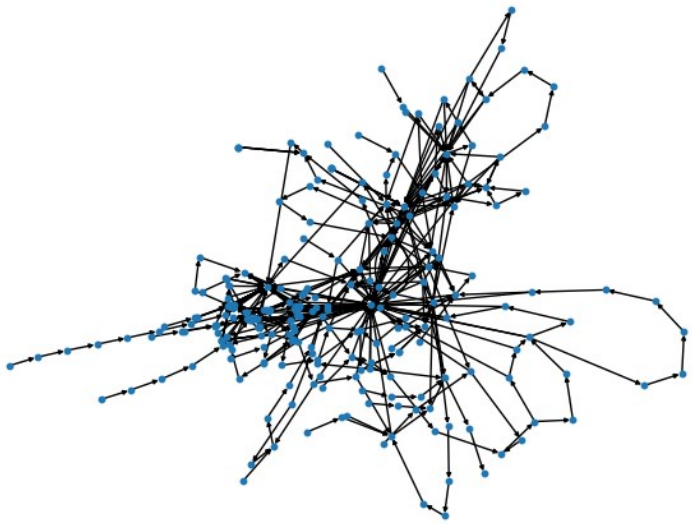


Figure 3-17: Graph of the COMMS Software Extracted from Execution Traces

In addition to performance effects and incompleteness of the graph, this method cannot generate feature vectors of nodes, which are needed by graph neural networks. Consequently, another graph generation approach needs to be taken.

3.4.4 Extract Graph Representations of Programs by Static Code Analysis

Another way to generate the graph is static code analysis. For example, there are tools to generate control flow graphs from source code. A control flow graph (CFG) records possible paths in a program, as shown on the right side of Figure 29. Theoretically, if we get a control flow graph of the SUT, we can combine it with code coverage information.

Like the story in section 3.4.1, several off-the-shelf tools were tried:

- TI Code Composer Studio can extract calling trees of the source code. However, the trees include too many low-level functions in the libraries used by Delfi-PQ software. Function names in the trees are also modified. It is challenging to combine these trees into a graph.
- Other tools, like CodeViz, rely on output from compilers like GCC and Clang. Unfortunately, they do not support the old TI ARM compiler we use.

An open-source code parser called Joern[24] is selected. Joern supports languages like C/C++, Python, and Java. Moreover, Joern can even parse the code with errors. These properties mean Joern can quickly adapt to different programs.

After parsing the source code, Joern can generate a CFG for every function in the program. However, it cannot generate a single CFG for the whole program. In other words, Joern cannot find the link between a function call and the called function. For example, if there is a Reset() call in the program, we cannot directly find where the definition of Reset() is.

This study implements a tool to generate a graph representation based on the output from Joern. We first use Joern to generate CFGs for functions in the programs with the following command:

```
cpg.method.isExternal(false).nameNot(".*<.*>.*").map(node => (node.id,
node.methodReturn.id, node.name, node.filename, node.lineNumber.l,
node.dotCfg.l)).toJsonPretty |> "methods.txt"
```

The command will generate the following information for every function in the source code:

- ID of the “start” node of the method, given by Joern
- ID of the “return” node of the method, given by Joern
- Method name
- Name of the file which contains the method
- Line number of the method
- Control flow graph of the method in JSON format

Based on this information, the tool traverses control flow graphs of all methods and connects method calls to method “start” nodes with the same method names, as shown in Figure 3-18. This step also connects separated control flow graphs into a complete control flow graph. Although such connections do not represent actual control flows, similar programs will still have similar graph structures, which makes the neural network reusable (section 3.4.2).

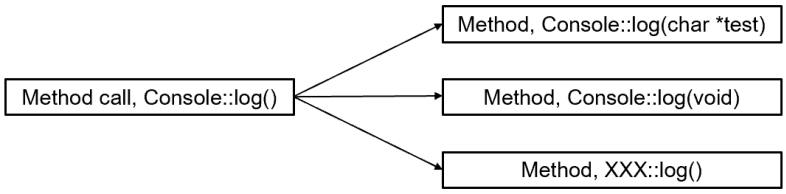


Figure 3-18: Connecting Method Call to Method

There are only three types of nodes in the complete control flow graph: “start” nodes of methods, “return” nodes of methods and CodeCount nodes. Figure 3-19 is the extracted graph structure from the COMMS software, which contains 1517 nodes and 3925 edges. When deleting all

[24] https://joern.io/

"start"/"return" nodes and directly connecting the CodeCount nodes, the number of edges will grow to a ~100,000 level.

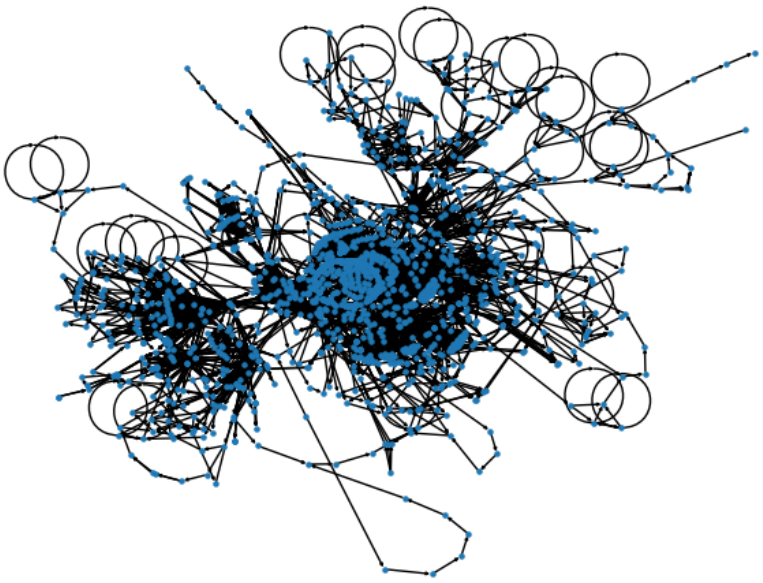


Figure 3-19: Graph of the COMMS Software by Static Analysis

The tool also generates feature vectors for nodes in the complete control flow graph. It first trains a Word2Vec model (Mikolov, Chen, Corrado, & Dean, 2013) with all source code files of the Delfi-PQ software. After that, the tool gives the following feature vector for node i:

$$\mu_i = (typeID_i, word2vec(methodName_i), word2vec(fileName_i)) \quad (3\text{-}1)$$

The $typeID_i$ is 0 for a method "start" node, 0.5 for a method "return" node, or 1 for a CodeCount node. The length of a feature vector μ_i is 127, plus a coverage mask $c_i \in \{0,1\}$. The tool uses the Gensim[25] library to train the Word2Vec model.

This method is better than the approach in section 3.4.3 since it does not affect the performance of SUT. It can also find all CodeCount probes and generate corresponding feature vectors.

3.5. Brief Summary of the Chapter

- The Delfi-PQ satellite consists of several subsystems. They have some standard hardware and unique components. Each subsystem has a microcontroller, and its onboard software runs on the microcontroller. These controllers are connected to an RS-485 bus, and the onboard computer (OBC) is the master of the bus.
- Onboard software of each subsystem shares the same DelfiPQcore, which acts as a lightweight operating system and relative middleware. The software also has applications depending on its functions.
- We compared the Delfi-PQ flight software with other onboard software. We found that the software of other PocketQubes and educational CubeSats have a similar (or lower) complexity as Delfi-PQ. The DelfiPQcore provides most of the mission-independent functions of a famous onboard software architecture, NASA core Flight System (cFS), although it does not follow

[25] https://radimrehurek.com/gensim/

some standards. However, the apps of Delfi-PQ are much simpler than some mission-specific applications developed for cFS.

- The testing command generation tool can run on a lab computer or a cloud server. The command will be put on the RS-485 bus by the Electrical Ground Support Equipment (EGSE) board. However, the bus only has one master, i.e., the EGSE or the OBC. To avoid potential bus contention, we can set the OBC to a “passive” mode that only receives commands from the bus and replies. Another approach is to let the EGSE mimic the COMMS subsystem. The OBC will poll the “fake COMMS system” to retrieve the “fake ground commands”. This research uses the first method.
- An IO processing module is implemented in this work. The IO processing module consists of a parser and a state identifier. The module can list all subsystem commands and extract symbolic state variables from the response. However, the module needs a significant amount of prior knowledge specified by the testers. Thus, some command generation algorithms in this work only use part of the module or even do not use it.
- A **code coverage measurement** tool is implemented in this work. The tool can instrument C/C++ source code automatically. A code coverage collection command is defined in the Delfi-PQ flight software. Moreover, the tool can analyse the response of the coverage collection command and generate a coverage report.
- We can directly use the code coverage vector as input to the neural network. However, the corresponding branch of each element in the vector may change after source code modification. It makes the trained RL algorithm unable to test a new software version (regression testing).
- To make the algorithm useful in regression testing, we can represent the code coverage in a graph (e.g., control flow graph). In the research, we use Joern to generate control flow graphs for every method in the source code of the onboard software. After that, we use an algorithm to connect these graphs and form a **graph representation of the full software**.
- In the graph, each node has a feature vector (3-1). The vector is embedded by a Word2Vec model from the method and file names. The edges do not have feature vectors.

4 Algorithm Designs

This section introduces the algorithms used in this study. Representative reinforcement learning algorithms in Table 2-2 are implemented, including:

- The tabular Q-Learning algorithm.
- The Deep Q Network (DQN) algorithm. We choose the Dueling Double Deep Q Network (D3QN). It is a value-based algorithm.
- The Proximal Policy Optimization (PPO) algorithm is an actor-critic algorithm.

The D3QN and PPO implementations also have different configurations. For example, they may have different types of neural networks, including:

- The MLP network, i.e., fully-connected layers.
- The Gated Graph Neural Network (GGNN).
- The Graph Attention Layers (GAT).
- Graph pooling layer (optional).

On the other hand, two baselines are used to compare with the RL-based testing. The first baseline is random testing, and the second is search-based testing with the genetic algorithm.

4.1. Q-Learning

Q-Learning is one of the simplest reinforcement learning algorithms, but it is widely used in software testing, especially GUI testing (Table 2-2). As a reference, this work tries to use Q-Learning to test the onboard software.

4.1.1 Brief Introduction

As mentioned in section 2.3.2, the Q-Learning algorithm uses a Q table to record estimations $\hat{Q}^{\pi}(s,a)$ of the action-value function $Q^{\pi}(s,a)$. To that end, states and actions in the Q-Learning algorithm should be discrete scalars. Figure 4-1 is an example of the Q table.

Figure 4-1: A Q Table to Store $\hat{Q}^{\pi}(s,a)$

The Q-Learning algorithm has an **ε-greedy behavior policy** and a **greedy target policy**:

- During training, the agent selects the action a with the highest $\hat{Q}(s,a)$ with probability 1 – ε, or it selects a random action with probability ε. ε is a small value (e.g., 0.1) and usually decreases over time.
- When performing the task, the agent simply selects the action a_i with the highest $\hat{Q}(s,a)$.

In every time step $t+1$, the agent updates the Q table by:

$$\hat{Q}^{\pi}(s_t, a_t) \leftarrow \hat{Q}^{\pi}(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \cdot max\hat{Q}^{\pi}(s_{t+1}, a) - \hat{Q}^{\pi}(s_t, a_t)\right] \quad (4\text{-}1)$$

Where α is the learning rate, γ is the discount factor, and r_{t+1} is the reward at step $t+1$.

4.1.2 States and Actions

We need to summarize the observation to a discrete scalar with the help of the IO processing module in section 3.3.2. After sending a command and getting a response, the state identifier generates a dictionary of the parameters of SUT. If the dictionary is never seen, it will be assigned a scalar index and a new column in the Q table. Otherwise, the Q learning agent will find the original index and column assigned to the dictionary.

At the same time, the parser lists all commands according to the CSV input files. The agent chooses the following command during training according to the ε-greedy behaviour policy.

Although Q-Learning is simple, it has many disadvantages. As discussed in section 3.3.2, the IO processing module needs much prior knowledge. Moreover, the Q-Learning algorithm can only handle scalar input/output instead of high-dimensional input/output like code coverage vectors or command vectors. It limits the usage of the algorithm.

4.2. Deep Q Network

The Deep Q Network (DQN) improves the basic Q-Learning algorithm. The idea is to use a neural network to replace the Q table. It is easy to develop this idea, but the algorithm with a Q network is not stable for a long time. Ultimately, this problem was solved with some tricks (Mnih et al., 2015).

4.2.1 Brief Introduction

To make the training more stable, the basic DQN algorithm has 2 neural networks: a **value network** with weights w to calculate $\hat{Q}^{\pi}(s_t, a_t)$, and a **target network** with weights w^- to compute $\hat{Q}^{\pi}(s_{t+1}, a)$. The value network and the target network share the same parameters at the beginning. In the training process, the optimizer only updates the weights in the value network. At every N_{target} steps, the target network will be set equal to the value network. In other words, w^- has a lag in weight updates compared with w.

At every time step t, DQN selects action a_t according to s_t and the ε-greedy behaviour policy. After that, it saves the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in a **replay buffer**. When update the value network, the algorithm will randomly sample some transitions in the replay buffer and minimize the following loss function:

$$Loss = -\frac{1}{N_\tau}\sum_{\tau}\left[r_{t+1} + \gamma \cdot max\hat{Q}_{w^-}^{\pi}(s_{t+1}, a) - \hat{Q}_{w}^{\pi}(s_t, a_t)\right]^2 \quad (4\text{-}2)$$

Where N_τ is the number of transitions τ sampled from the replay buffer. $\hat{Q}_{w^-}^{\pi}(s_{t+1}, a)$ is the output of the target network, whose inputs are s_{t+1} and a. $\hat{Q}_{w}^{\pi}(s_t, a_t)$ is the output of the value network, whose inputs are s_t and a_t. These two networks share the same architecture and number of neurons, but their weights are different.

The basic DQN algorithm tends to overestimate $Q^{\pi}(s,a)$. To alleviate this problem, Van Hasselt, Guez and Silver (2015) proposed the **Double DQN** algorithm. Its loss function is expressed as:

$$Loss = -\frac{1}{N_\tau}\sum_{\tau}\left[r_{t+1} + \gamma \cdot \hat{Q}_{w^-}^{\pi}\left(s_{t+1}, argmax\hat{Q}_{w}^{\pi}(s_{t+1},a)\right) - \hat{Q}_{w}^{\pi}(s_t,a_t)\right]^2 \quad (4\text{-}3)$$

In the second term, we select the action with the maximal $\hat{Q}^{\pi}(s_{t+1},a)$ using the value network and recalculate its $\hat{Q}^{\pi}(s_{t+1},a)$ using the target network. Since the 2 network has different weights, it is unlikely that they overestimate the same action. However, double DQN becomes more vulnerable to noise.

Another improvement to the DQN algorithm is the **Dueling DQN** (Wang et al., 2016). It estimates $Q^{\pi}(s,a)$ by 2 sub-networks: one estimates the state-value function $V^{\pi}(s)$, and the other estimate the **advantage function** $A^{\pi}(s,a)$. Estimations from the sub-networks can be combined:

$$\hat{Q}^{\pi}(s_t,a_t) = \hat{V}^{\pi}(s_t) + \hat{A}^{\pi}(s_t,a_t) \quad (4\text{-}4)$$

In this case, the loss function is still calculated by the value network and the target network, and each of them has 2 sub-networks. In other words, the dueling DQN only changes the architecture of the neural networks, which makes the estimation more accurate.

This study tries the Double Dueling DQN (D3QN) algorithm, i.e., combining the tricks mentioned above with the basic DQN algorithm. A forward propagation process of the D3QN algorithm is shown in Figure 4-2.

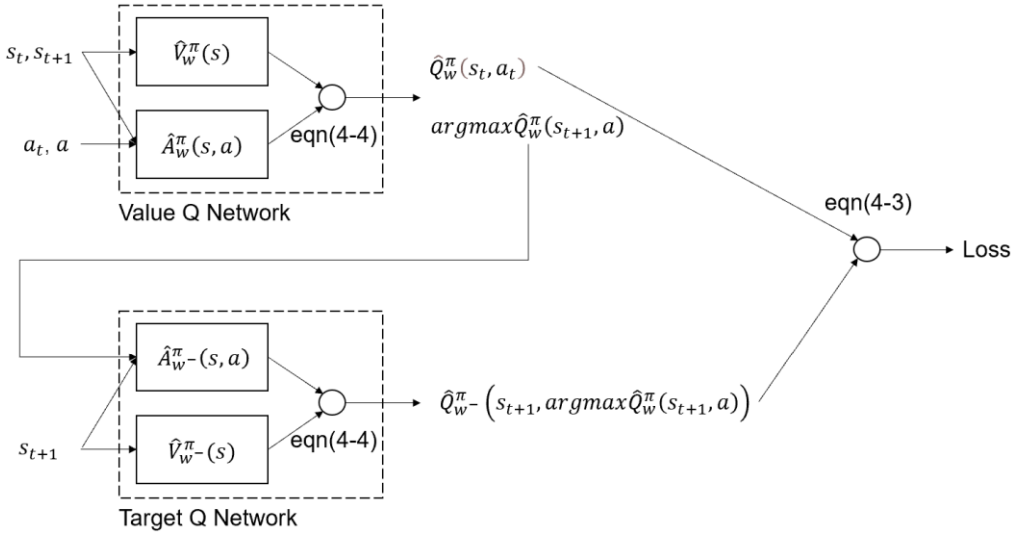


Figure 4-2: A Forward Propagation of the D3QN Algorithm

There are some hyperparameters in the D3QN algorithm, as shown in Table 4-1.

Table 4-1: Hyperparameters of D3QN

Hyperparameter	Explanation
γ	Discount factor of rewards
Buffer size	How many transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ the replay buffer can store
Mini-batch size	Transitions are grouped into mini-batches and then used to update the neural networks
Learning rate	Learning rate of the neural network optimizer
Learning starts	From which time steps the optimizer starts to update the value network

Training frequency	How many steps the agent interacts with the environment before updating neural networks
Total time steps	How many steps the agent interacts with the environment in the whole training process
ε_{start}	The initial ε at the beginning of exploration
ε_{end}	The ε at the end of exploration
Exploration fraction	A ratio to control the length of exploration. For example, if the training process runs 200000 steps in total and the exploration fraction is 0.5, then the exploration starts from step 1 to 100000. During the exploration, ε decreases from ε_{start} to ε_{end} linearly.
Epoch length	How many steps the agent interacts with the environment before resetting the environment
N_{target}	The frequency to set the target network to be equal to the value network
Neural Network Design	Network structure, neuron number, activation function, etc.
Network Initialization	Initialization method of each neural network layer

According to the state/action representation and neural network architecture, there are several configurations of D3QN algorithms in this study. The following subsections explain these configurations in detail.

4.2.2 D3QN with State Vectors and Discrete Actions (D3QN-Discrete-MLP)

In the most basic configuration, the parser outputs a list of all available commands (section 3.3.2). Each action is a scalar index of a pre-defined command in the list. It is called **Discrete** action space in reinforcement learning research.

At the same time, it uses a plain vector as state input and selects a discrete scalar as the following action. The state vector is a concatenation of 2 vectors:

- The code coverage vector (section 3.4.2). The length of this vector is the number of probes in the source code. Each element indicates whether the probe has been triggered.
- A vector that contains actions in previous k steps. The length of this vector is the number of actions listed by the parser. Each element indicates how often the action has been taken in previous k steps.

The neural network architecture to calculate $\hat{V}^{\pi}(s)$ is expressed by:

$$\begin{aligned} h^{(0)} &= s \\ h^{(l+1)} &= ReLU\left(MLP\left(h^{(l)}\right)\right) \\ \hat{V}^{\pi}(s) &= MLP(h^{(L)}) \end{aligned} \tag{4-5}$$

Here $h^{(l)}$ is the output of layer l, and L is the number of hidden layers. $MLP(\cdot)$ is a fully connected layer and $ReLU(\cdot)$ is an activation function. Similarly, the network to calculate $\hat{A}^{\pi}(s,a)$ is

$$\begin{aligned} h^{(0)} &= s \\ h^{(l+1)} &= ReLU\left(MLP\left(h^{(l)}\right)\right) \\ \hat{A}^{\pi}(s) &= MLP(h^{(L)}) \end{aligned} \tag{4-6}$$

Here $[\cdot]$ means concatenation of vectors. $\hat{A}^{\pi}(s)$ is a vector which can be expressed by

$$\hat{A}^{\pi}(s) = [\hat{A}^{\pi}(s, a_1), \hat{A}^{\pi}(s, a_2), \hat{A}^{\pi}(s, a_3), \dots] \tag{4-7}$$

Then, $\hat{A}^{\pi}(s, a)$ and $\hat{V}^{\pi}(s)$ are combined by (4-4) to calculate $\hat{Q}^{\pi}(s, a)$. The subsequent computation is the same as the previous section.

This configuration is represented by D3QN-Discrete-MLP in this study.

4.2.3 D3QN with State Graphs and Discrete Actions (D3QN-Discrete-GGNN)

This configuration use graphs as input states. The graphs are constructed by static code analysis in section 3.4.4, and the node features include code coverage information. Furthermore, we use the **Gated Graph Neural Network** GGNN (Li, Tarlow, Brockschmidt, & Zemel, 2015) to process the graphs. Unlike the Graph Convolutional Network GCN (Kipf & Welling, 2016) and the GraphSAGE, the GGNN can handle models with more than 20 layers and is helpful for complex network representation, e.g., program control flow (Hamilton, Ying, & Leskovec, 2017).

Let $h_v^{(l+1)}$ to be the hidden embedding vector of node v in the layer $l + 1$ of a GGNN, it can be expressed by:

$$\begin{aligned} h_v^{(0)} &= [\mu_v, c_v, c_v, c_v, \dots] \\ m_v^{(l+1)} &= Aggregate\left(\left\{MLP\left(h_u^{(l)}\right)\right\}_{u \in \mathcal{N}(v)}\right) \\ h_v^{(l+1)} &= GRU(h_v^{(l)}, m_v^{(l+1)}) \end{aligned} \tag{4-8}$$

Here, μ_v is the feature vector of node v acquired by equation (3-1), and $c_v \in \{0,1\}$ is the coverage mask to indicate whether the node has been covered. Note that the length of $h_v^{(0)}$, called **number of channels**, can be equal to or larger than the length of $[\mu_v, c_v]$. If the number of channels is larger, it will be padded with c_v.

$\mathcal{N}(v)$ is all neighbors of the node v on the graph, including node v itself. The aggregation method can be $\{sum, mean, max\}$. According to our experience, sum has the best performance in the problem. Moreover, GRU means a Gated Recurrent Unit (Chung, Gulcehre, Cho, & Bengio, 2014), a type of recurrent neural network layer.

The output from the GGNN is embedding vectors of all nodes in the graph. Since there are many nodes, the sum of the lengths of these vectors can be very large. It's not feasible to directly concatenate these vectors and use a fully connected layer to process $[h_1^{(L)}, h_2^{(L)}, h_3^{(L)}, \dots]$. Otherwise, the fully connected layer will contain too many weights and need significant time to train.

Thus, we need to find a way to extract information from node embeddings $h_v^{(l+1)}$. There are 2 options to solve the problem. The first option is using a **graph pooling layer**. For example, the simplest pooling methods are sum, $mean$, and max of all node embeddings. (Dai et al., 2019) used a node selection pooling method in their RL-based testing. Zhou, Liu, Siow, Du, and Liu (2019) used a 1D convolution layer to aggregate information in node embeddings. The Pytorch Geometric library (Fey & Lenssen, 2019) also provides several graph pooling layers.

However, in our own tests, the following **attentive pooling layer** (Li et al., 2015) has the best performance, i.e.,

$$o = ReLU\left(\sum_v \left(Sigmoid\left(MLP_1\left(h_v^{(L)}, \mu_v\right)\right) \odot ReLU\left(MLP_2\left(h_v^{(L)}, \mu_v\right)\right)\right)\right) \quad (4\text{-}9)$$

Here, o is the information extracted from the node embeddings, and L is the final layer of the GGNN. MLP_1 and MLP_2 are the MLP that take the concatenation of $h_v^{(L)}$ and μ_v as input and output real-valued vectors. $\odot$ mean dot product between two vectors.

The second option is to use an MLP to reduce node embeddings to scalars and then organize these scalars into a graph feature vector, i.e.,

$$o = [ReLU\left(MLP\left(h_1^{(L)}\right)\right), ReLU\left(MLP\left(h_2^{(L)}\right)\right), ReLU\left(MLP\left(h_3^{(L)}\right)\right), \dots] \quad (4\text{-}10)$$

In this study, we call this approach as **feature compression**.

As mentioned in section 3.4.2, we hope the same graph neural network can process similar graphs with different node number. However, in the feature compression approach, the length of the graph feature vector o depends on the node number. Therefore, the GNN with feature compression cannot process graphs with different node numbers. This study compares the performance of attentive pooling and feature compression to illustrate the influence of the graph pooling layer.

After that, we construct the state vector s based on the output o and the history vector y:

$$s = [o, y] \quad (4\text{-}11)$$

At the end, s will be substituted into equation (4-5) and (4-6). The subsequent computation is the same as the D3QN-Discrete-MLP.

As this configuration is more complex, it's important to mention the initialization methods of the parameters. The weights of the MLP network in equation (4-8) are initialized to a uniform distribution. The GRU in (4-8) and the MLP in (4-9) take the default initialization. The weights of the MLP in equation (4-10) takes a normal distribution with a standard deviation of $\sqrt{2}$.

4.2.4 D3QN with State Graphs and Discrete Actions (D3QN-Discrete-GAT)

This configuration is like the D3QN-Discrete-GGNN but uses the **Graph Attention Network** GAT (VELIČKOVIĆ and Petar, 2017) to process the graphs. Each layer of the GAT updates node embeddings by:

$$h_v^{(l+1)} = \alpha_{v,v} h_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \alpha_{v,u} h_u^{(l)} \quad (4\text{-}12)$$

Where $\alpha_{v,u}$ represents the self-attention with v being the index of the query node and u being the index of the key node (Harries, 2020). The subsequent processing is the same as the D3QN-Discrete-GGNN configuration. The GAT layers take the default initialization in Pytorch Geometric.

4.3. Proximal Policy Optimization

This study also selects the Proximal Policy Optimization (PPO) algorithm to implement. PPO is a popular deep reinforcement learning in recent years and the baseline RL algorithm of OpenAI. Many works in Table 2-2 used PPO or similar algorithms like A2C and A3C.

4.3.1 Brief Introduction

A PPO agent has 2 neural networks: the actor network and the critic network:

- The inputs of the actor network are state s and action a. The output of the network is $\pi_w(a|s)$, i.e., the probability to take the action a under state s with policy parameters w.
- The input of the critic network is state s. The output of the critic network is $V(s)$, which estimates the state-value function of state s.

Every N step, the PPO agent interacts with the environment and stores observations, actions, and rewards in the memory. The length of the memory is N. After that, the agent minimizes the following loss function by updating parameters of neural networks for several times:

$$Loss_{total} = Loss_{pg} - c_{ent}Loss_{ent} + c_{vf}Loss_{vf} \tag{4-13}$$

The $Loss_{total}$ contains 3 items: the policy gradient loss $Loss_{pg}$, the action entropy loss $Loss_{ent}$, and the value loss $Loss_{vf}$. There coefficients are 1, c_{ent}, and c_{vf}, respectively.

The **policy gradient loss** $Loss_{pg}$ is:

$$Loss_{pg} = -\frac{1}{N_\tau}\sum_{(s_t,a_t)} min\left\{\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}\hat{A}^\pi(s_t,a_t), clip\left[\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}, 1-\xi, 1+\xi\right]\hat{A}^\pi(s_t,a_t)\right\} \tag{4-14}$$

This loss item uses **importance sampling**, which makes it look complex. In the expression, $A^\pi(s_t,a_t)$ is the **advantage function** and shows how good action a_t is, comparing with other actions under state s_t. Note that the parameters of the actor network will change after updates. w' are the parameters when the actions were taken and w are the current parameters. ξ is a hyperparameter to control the magnitude of policy update and usually has a small value (like 0.2). N_τ is the number of transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ stored, depending on the environmental steps (Table 4-2).

Let δ_t^V to be the TD residual:

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4-15}$$

Where r_t is the reward of step t, $V(s_t)$ and $V(s_{t+1})$ are the values of s_t and s_{t+1} estimated by the critic network. Based on this definition, the advantage function $A^\pi(s_t,a_t)$ is computed by the GAE method (Schulman, Moritz, Levine, Jordan, & Abbeel, 2015):

$$\hat{A}^\pi(s_t,a_t) = \sum_{l=0}^{\infty}(\gamma\lambda)^l\,\delta_{t+l}^V \tag{4-16}$$

The **action entropy loss** $Loss_{ent}$ is:

$$Loss_{ent} = -\frac{1}{N_\tau}\sum_{s_t}\sum_{a}\pi_w(a|s_t)\cdot\log\left(\pi_w(a|s_t)\right) \tag{4-17}$$

If this term is higher, the policy π_w will be more random. Note that $c_{ent}Loss_{ent}$ is subtracted from equation (4-13). That's to say, (4-13) encourages the policy π_w to explore the environment. By tuning the coefficient c_{ent}, we make a balance between exploration and exploitation.

The **value loss** $Loss_{vf}$ can be calculated with:

$$Loss_{vf} = -\frac{1}{N_\tau}\sum_{s_t}\frac{1}{2}(V(s_t) - \widehat{G}_t)^2 \tag{4-18}$$

In (4-18), the $\widehat{G}_t$ is estimated by:

$$\widehat{G}_t = \hat{A}^\pi(s_t, a_t) + V(s_t) \tag{4-19}$$

We can use the **Kullback-Leibler divergence** indicates the magnitude of policy update, or in other words, how different the updated policy π_w is from the old policy $\pi_{w'}$. After updating the neural network, the agent will estimate the Kullback-Leibler divergence using the following expression (Schulman, 2020):

$$\widehat{KL}(\pi_w, \pi_{w'}) = \frac{1}{N_\tau}\sum_{(s_t,a_t)}\left[\left(\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)} - 1\right) - log\left(\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}\right)\right] \tag{4-20}$$

If the divergence is too small, it means the policy update is too slow and the training may take a long time. On the other hand, if the policy update is too quick, some assumptions of importance sampling may be invalid, and the training process will become unstable.

There are 2 ways to control the magnitude of policy update. One approach is carefully selecting ξ in equation (4-14) to limit $Loss_{pg}$. However, sometimes this approach does not work well. Another approach is setting a limitation KL_{target}. If the estimated $\widehat{KL}(\pi_w, \pi_{w'}) > KL_{target}$, the agent will reduce the learning rate of the neural network optimizer. This **learning rate annealing** is not common in standard PPO implementation but makes the training process more stable.

Although PPO is simple and robust, it's still sensitive to hyperparameters. Table 4-2 lists some important hyperparameters of this algorithm:

Table 4-2: Hyperparameters of PPO

Hyperparameter	Explanation
KL_{target}	If $\widehat{KL}(\pi_w, \pi_{w'}) > KL_{target}$, the algorithm will reduce the learning rate
Mini-batch size	Observations, actions, and rewards are grouped into mini-batches, and then used to update the neural networks
Initial learning rate	Initial learning rate of the neural network optimizer
γ	Discount factor of rewards
c_{ent}	Coefficient of $Loss_{ent}$ in the total loss
c_{vf}	Coefficient of $Loss_{vf}$ in the total loss
Total time steps	How many steps the agent interacts with the environment in the whole training process
Epoch length	How many steps the agent interacts with the environment before resetting the environment in an epoch
Environment steps	How many steps the agent interacts with the environment before updating neural networks

Update steps	How many times the optimizer updates the neural networks after interaction
λ	A coefficient of GAE in (4-15)
ξ	A coefficient of importance sampling in (4-13)
Maximal norm of gradient	The gradient larger than this limit will be clipped
Neural Network Design	Network structure, neuron number, activation function, etc.
Network Initialization	Initialization method of each neural network layer
std_{critic}^{output}	Initial standard deviation of the output layer of the critic network
std_{actor}^{output}	Initial standard deviation of the output layer of the actor network

Limited by pages, this section does not explain why the algorithm has adopted this design. The readers can refer to (Schulman et al., 2017) and (Huang, Julien, Antonin, Anssi, & Wang, 2022).

According to the state/action representation and neural network architecture, there are several configurations of PPO algorithms in this study. The following subsections explain these configurations in detail.

4.3.2 PPO with State Vectors and Discrete Actions (PPO-Discrete-MLP)

The input and output of this configuration are like D3QN-Discrete-MLP. The actor network to compute $\pi_w(a|s)$ can be expressed by:

$$\begin{aligned} h^{(0)} &= s \\ h^{(l+1)} &= ReLU\left(MLP\left(h^{(l)}\right)\right) \\ \pi_w(a|s) &= MLP(h^{(L)}) \end{aligned} \tag{4-20}$$

As mentioned above, $\pi_w(a|s)$ is a vector whose length is the number of actions provided by the parser. Each element of the vector indicates the probability to take an action under the policy π_w. One of the actions will be sampled according to the probability distribution.

The critic network to calculate $V(s)$ is:

$$\begin{aligned} h^{(0)} &= s \\ h^{(l+1)} &= ReLU\left(MLP\left(h^{(l)}\right)\right) \\ V(s) &= MLP(h^{(L)}) \end{aligned} \tag{4-21}$$

Here, $V(s)$ is a scalar. The actor network and the critic network are independent.

4.3.3 PPO with State Vectors and Action Vectors (PPO-MultiDiscrete-MLP)

Unlike previous configurations, this configuration sends command vectors like [25, 1, 13, 1] to the SUT. It's usually called the **MultiDiscrete** action space.

In this configuration, the parser lists all possible values of all parameters in a single list. Let the length of the command vector to be L_v and the number of parameter values to be n_p. The actor network will generate a probability distribution vector with the length of $L_v \times n_p$. According to the probability distribution, the agent samples L_v values, which are concatenated to a command vector.

The state vector only contains code coverage information without historical actions. In fact, integrating interaction history with the state vector is possible, but there is not enough time to try this function.

This configuration uses less prior knowledge about encoding of commands. On the other hand, it may send invalid or even dangerous commands to the SUT.

4.3.4 PPO with State Graphs and Discrete Actions (PPO-Discrete-GGNN)

This configuration is like the D3QN-Discrete-GGNN configuration. It firstly generates the state vector s with the Gated Graph Neural Network (GGNN), and then substitutes s into (4-20) and (4-21). In other words, the actor network and the critic network share the same GGNN in this configuration.

4.3.5 PPO with State Graphs and Discrete Actions (PPO-Discrete-GAT)

This configuration uses the Graph Attention Network, instead of a GGNN, to process graphs. The subsequent process is the same as PPO-Discrete-GGNN.

4.3.6 PPO with State Graphs and Action Vectors (PPO-MultiDiscrete-GGNN)

This configuration uses the Gated Graph Neural Network (GGNN). Its actor network generates a probability distribution vector with the length of $L_v \times n_p$, from which a command vector with L_v values is sampled.

4.4. Baselines

We use the following two baselines in the study.

4.4.1 Random Testing

In this research, the random testing baseline means selecting a random command from the command list, like PPO-Discrete-MLP.

The random testing baseline uses the same episode length as the reinforcement algorithms.

4.4.2 Testing with the Genetic Algorithm

This research uses the standard genetic algorithm (Holland, 1992) as another baseline. The genetic algorithm has four steps in an iteration:

- Calculate the objective function of every solution.
- Select good solutions.
- Perform the “crossover” operation on the good solutions, i.e., randomly swap some segments in the solution vectors.
- Perform the “mutation” operation on the good solutions, i.e., randomly change some elements in the solution vectors.

The genetic algorithm generates two types of solutions:

- A sequence of scalars in which each scalar represents a command in the command list.
- A sequence of command vectors, like the PPO-MultiDiscrete-MLP configuration.

The sequence length is equal to the episode length of the reinforcement learning algorithms. Furthermore, the genetic algorithm uses the sum of rewards as the objective function of the sequence.

4.5. Implementation Details

The Q-Learning algorithm in this research is written from scratch. The D3QN and PPO algorithms are modified from CleanRL (Huang, Dossa, Ye, & Braga, 2021). CleanRL is a repository including single-file implementations of deep reinforcement learning algorithms. These algorithms use Pytorch to perform low-level operations of neural networks like backpropagation.

The graph neural networks in the study are implemented with the Pytorch Geometric library (Fey & Lenssen, 2019), which is built on Pytorch. The library includes the implementation of GAT layers. However, we must implement the GGNN layers with the low-level APIs of the library. Our GGNN implementation takes from Longa and Pellegrini's tutorial (2022).

The implementation in this study is lightweight. Table 4-3 shows the number of lines of code in each file.

Table 4-3 Number of Lines of Code in Our Implementation

Configuration	File Name	Number of Lines of Code	Note
Q-Learning	QLearningAgent.py	40	Store and process the Q table
	Main.py	85	Main loop
	StateIdentification.py	92	Can only identify several states
D3QN-Discrete-MLP	D3qn.py	170	Include the D3qn algorithm and neural network in a single file
D3QN-Discrete-GGNN	D3qn.py	153	Include the D3qn algorithm
	GNN_Agent.py	100	The Gated Graph Neural Network
D3QN-Discrete-GAT	D3qn.py	153	Include the D3qn algorithm
	GNN_Agent.py	100	The Graph Attention Network
PPO-Discrete-MLP	PPO.py	200	Include the PPO algorithm and neural network in a single file
PPO-MultiDiscrete-MLP	PPO.py	203	Include the PPO algorithm and neural network in a single file
PPO-Discrete-GGNN	PPO.py	166	Include the PPO algorithm
	GNN_Agent.py	103	The Gated Graph Neural Network
PPO-Discrete-GGNN	PPO.py	164	Include the PPO algorithm
	GNN_Agent.py	87	The Graph Attention Network
PPO-MultiDiscrete-GGNN	PPO.py	166	Include the PPO algorithm
	GNN_Agent.py	124	The Graph Attention Network
Random Testing Baseline	Piece of code in MyEnv.py	10	Randomly select commands and execute them
Genetic Algorithm Baseline	GA.py	89	The genetic algorithm
	PQ9Client.py	81	Handle low-level communication

Shared functions	MyEnv.py	~100	Provide high-level APIs for the Delfi-PQ testing environment
	MyEnv_toy.py	~30	Provide high-level APIs for the toy problem
	Parser.py	210	Parse raw response and generate command list (Many algorithms only use the command list)
	PQ9cov.py	153	Instrument source code and generate coverage report
	GraphExtract.py	143	Extract a graph from the source code
	MyWord2Vec.py	20	Generate feature vectors for nodes

Moreover, the implementation can run in the CPU or GPU mode of Pytorch. However, as mentioned in section 3.1, the lab computer is relatively outdated and cannot support the GPU mode of Pytorch. Running deep reinforcement learning algorithms in the CPU mode may take much longer.

4.6. Brief Summary of the Chapter

- Three RL algorithms are implemented in this work, i.e., the Q-Learning algorithm, the D3QN algorithm, and the PPO algorithm.
- The **Q-Learning** algorithm is one of the most basic RL algorithms. It receives a scalar observation, updates a Q table, and gives a scalar action. However, it cannot adapt to other types of inputs or outputs. If the state space or action space is large, we may not have enough memory to store the Q table.
- The **Deep Q Network** (DQN) is an improvement over the original Q-learning algorithm. It uses a neural network to fit the action-value function $Q(s,a)$. It also adopts some tricks to improve the stability of the algorithm. The **Double Duelling Deep Q Network** (D3QN) algorithm is a good and simple variant of DQN.
- The **Proximal Policy Optimization** (PPO) has a policy network to give probability $\pi_w(a|s)$ of each action under current state s. It also has a value network to fit the state-value function $V(s)$. Compared with the standard PPO implementation, we add learning rate annealing to make the training process more stable.
- We implement three types of neural network structures for both the D3QN and the PPO algorithms, including the **MLP** network (fully connected layers), the **Gated Graph Neural Network** (GGNN), and the **Graph Attention Network** (GAT).
- For the graph neural networks, a challenge is how to extract a graph embedding vector from the node feature vectors. We can use a **graph pooling layer** or the **feature compression** technique. We compare different types of graph pooling layers in a supervised task and find the **attentive sum pooling layer** has the best performance.
- Unlike other configurations, the PPO-MultiDiscrete-MLP configuration can generate a command vector. Thus, it does not have to use human-specified commands.
- Random testing and the genetic algorithm are implemented as the baselines of this study.
- The RL algorithms in this study are light-weight. Each algorithm contains up to two files and less than 300 lines of code. They only use Pytorch and Pytorch Geometric, instead of some complex reinforcement learning libraries (like RLlib) and distributed computing middleware (like Ray).

5 Filling Grid Testing

It is difficult to debug if a deep reinforcement learning algorithm goes wrong. This study implements a toy problem called “grid filling” to make the debugging effortless. The RL algorithms are first tested on the toy problem. If everything looks well, the algorithms will run in the environment with the Delfi-PQ hardware. Furthermore, we can compare the training curves in the toy problem and the curves in the actual environment. Such comparison helps us to understand the training process better.

In this chapter,

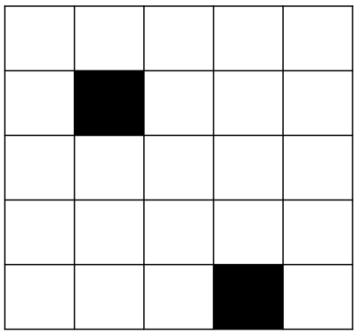
- Section 5.1 introduces the design of the “grid filling” problem.
- Section 5.2~5.8 shows the results of different configurations, including D3QN-Discrete-MLP, D3QN-Discrete-GGNN, D3QN-Discrete-GAT, PPO-Discrete-MLP, PPO-MultiDiscrete-MLP, PPO-Discrete-GGNN, PPO-Discrete-GAT.
- Section 5.9 compares the performance of these configurations and mentions some interesting findings.

5.1. About the Experiment

There are grids in the problem, and the RL algorithms should fill all grids as soon as possible. To the end, a state is represented by a vector of length or a graph with nodes. The state indicates which grids have been filled. An action can be a scalar (i.e., fill a specific grid) or a vector (i.e., fill a group of grids). The RL agent will receive a reward of 1 if it fills a blank grid or a reward of -1 if it fills a filled grid. Figure 5-1 explains the toy problem in detail.

If a deep reinforcement learning algorithm goes wrong, debugging is not easy. This study implements a toy problem called “grid filling” to make the debugging effortless. The RL algorithms are first tested on the toy problem. If everything looks well, the algorithms will be run in the actual environment with the Delfi-PQ hardware. Furthermore, we can compare the training curves in the toy problem and the curves in the actual environment. Such comparison helps us to understand the training process better.

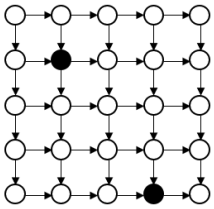
There are N grids in the problem, and the RL algorithms should fill all grids as soon as possible. To the end, a state is represented by a vector of length N or a graph with nodes N. The state indicates which grids have been filled. An action can be a scalar (i.e., fill a specific grid) or a vector (i.e., fill a group of grids). The RL agent will receive a reward of 1 if it fills a blank grid or a reward of -1 if it fills a filled grid. Figure 5-1 explains the toy problem in detail.



N grids (2 grids have been filled)

State:

[0,0,…1,…,1,0] or



Action: fill grid k, k∈[1,N]
Reward: 1 if grid k is blank, -1 if grid k has been filled

Figure 5-1: The "Filling Grid" Toy Problem

We can change the difficulty of the toy problem by modifying N, which is beneficial for debugging RL algorithms. If N increases, the RL algorithms usually need more time to learn. On the other hand, the toy problem become easier with a small N.

We only test the D3QN and PPO algorithms on the toy problem. It's difficult to apply the Q-Learning algorithm on this problem, since N grids have 2^N possible states. A Q table cannot deal with too many states.

In this experiment, we set $N = 36$. We will compare the cumulative rewards of different algorithms.

5.2. Results of D3QN-Discrete-MLP

In this experiment, an action is a scalar, and the episode length is 36. Table 5-1 shows the hyperparameters of the algorithm in this experiment.

Table 5-1: Hyperparameters of D3QN-Discrete-MLP in Filling Grid Test

Hyperparameter	Value
γ	0.9
Buffer size	20000
Mini-batch size	128
Learning rate	2.5e-4
Learning starts	128
Training frequency	10
Total time steps	20000
ε_{start}	1
ε_{end}	0
Exploration fraction	0.5
Epoch length	36
N_{target}	500
Neural Network Design	Value network: Input->MLP->ReLU->MLP->ReLU->MLP->value Advantage network: Input->MLP->ReLU->MLP->ReLU->MLP->advantage Every hidden layer has 512 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$.

Figure 5-2 shows the performance of this configuration. The cumulative reward and estimated Q value grow stably and quickly reach the maximum, i.e., fill all 36 grids.

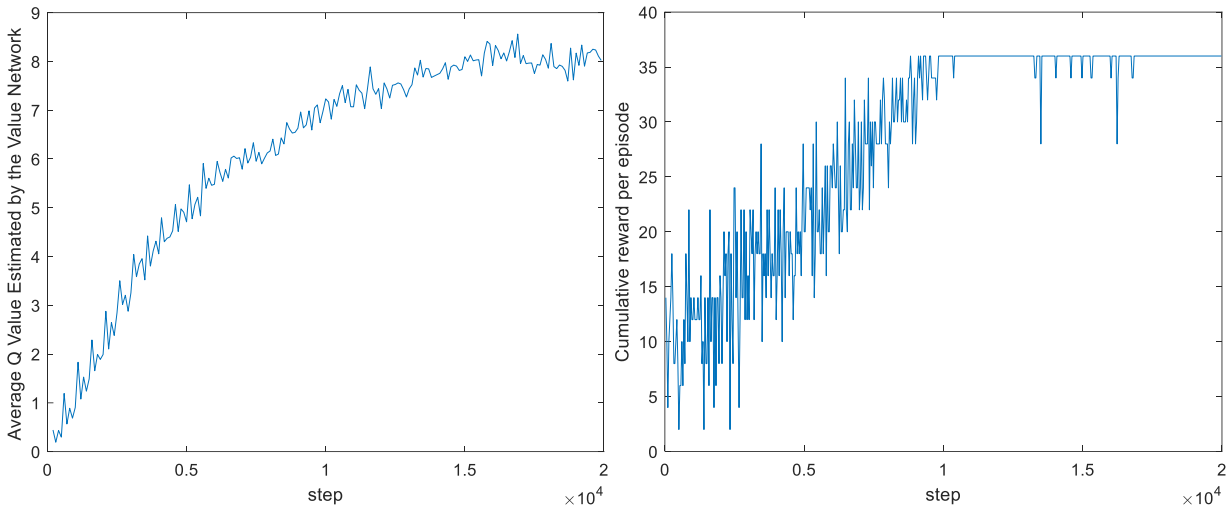


Figure 5-2: Results of D3QN-Discrete-MLP in the Toy Problem

5.3. Results of D3QN-Discrete-GGNN

The section shows the results of the D3QN algorithm with the Gated Graph Neural Network (GGNN). As mentioned in section 4.2.3, one problem is extracting information from node features, which GGNN has processed.

There are two design options to solve the problem. The first option is using an attentive sum pooling layer. The second option is to use an MLP to reduce node feature vectors to scalars and then organize these scalars into a graph feature vector (feature compression). We compare the performance of these two approaches in this section.

Table 5-2 shows the hyperparameters of the D3QN-Discrete-GGNN algorithm in this experiment:

Table 5-2: Hyperparameters of D3QN-Discrete-GGNN in Filling Grid Test

Hyperparameter	Value
γ	0.9
Buffer size	100000
Mini-batch size	128
Learning rate	2.5e-4 (feature compression) / 5e-4 (attentive pooling)
Learning starts	128
Training frequency	10
Total time steps	200000
ε_{start}	1
ε_{end}	0.05
Exploration fraction	0.8 (feature compression) / 0.9 (sum pooling)
Epoch length	36
N_{target}	500
Neural Network Design	Shared network: Input->GGNN(5 layers)->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Compute value: Graph embedding->MLP->value

	Compute advantage: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Initial std of the MLP for value or advantage computation is 0.01. Other layers are initialized in the same way as D3QN-Discrete-MLP

Figure 5-3 shows the performance of the attentive pooling approach. It performs similarly to the D3QN-Discrete-MLP configuration (Figure 5-2). The cumulative reward curve converges around 35.

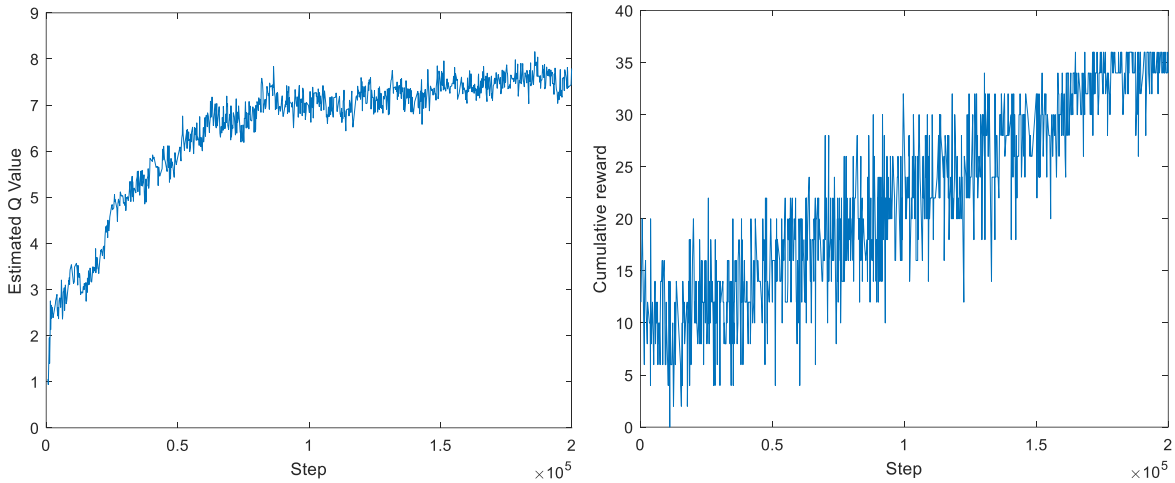


Figure 5-3: Results of D3QN-Discrete-GGNN with Attentive Pooling

By contrast, Figure 5-4 shows the results of the feature compression approach. Its Q value curve converges much more quickly. Note that in this test $\varepsilon_{end} = 0.05$, and the cumulative reward curve sometimes drops below 0. The average cumulative reward at the end of training is also around 35.

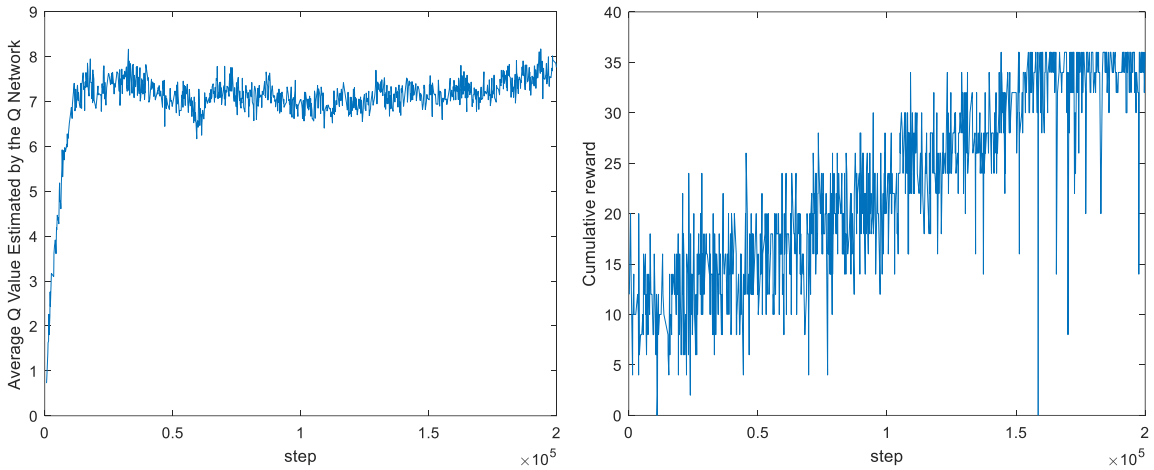


Figure 5-4: Results of D3QN-Discrete-GGNN with Feature Compression

5.4. Results of D3QN-Discrete-GAT

This configuration is like the D3QN-Discrete-GGNN but uses the Graph Attention Network (GAT) to process the graph. Table 5-3 shows the hyperparameters of this configuration in the experiment. The configuration only uses 2 GAT layers, and more GAT layers do not improve the performance in this test.

Table 5-3: Hyperparameters of D3QN-Discrete-GAT in Filling Grid Test

Hyperparameter	Value
γ	0.9
Buffer size	100000
Mini-batch size	128
Learning rate	2.5e-4 (feature compression) / 5e-4 (attentive pooling)
Learning starts	128
Training frequency	10
Total time steps	200000
ε_{start}	1
ε_{end}	0.05
Exploration fraction	0.8 (feature compression) / 0.9 (sum pooling)
Epoch length	36
N_{target}	500
Neural Network Design	Shared network: Input->GAT->ReLU->GAT->ReLU->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Compute value: Graph embedding->MLP->value Compute advantage: Graph embedding->MLP->advantage
Network Initialization	GAT uses default initialization. Initial std of output layers is 1. Other networks are initialized in the same way as D3QN-Discrete-MLP

Figures 5-5 and Figure 5-6 show the result with attentive pooling and feature compression, respectively. The Q value estimated by feature compression converges more quickly than the Q value estimated by attentive pooling. Unfortunately, both configurations have a lower cumulative reward (around 32) at the end of training.

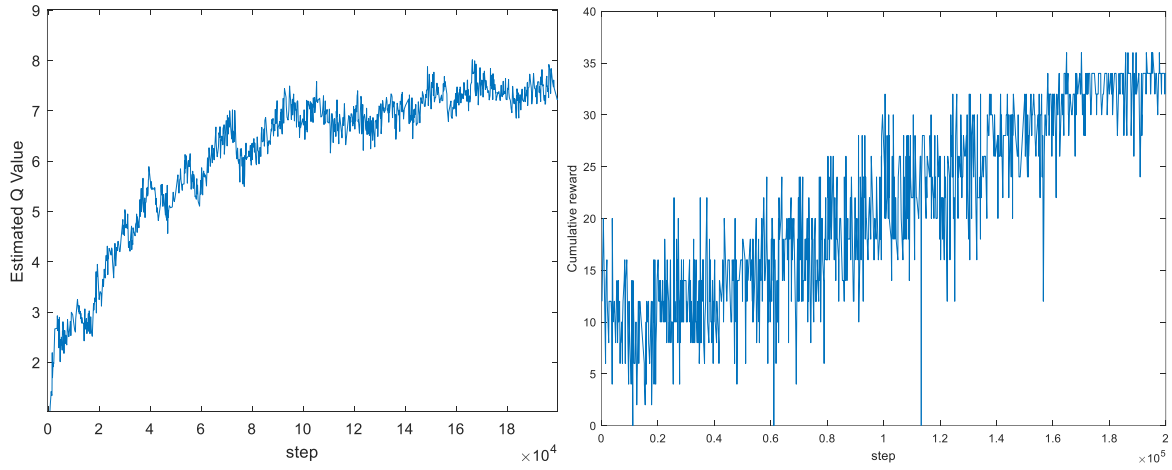


Figure 5-5: Results of D3QN-Discrete-GAT with Attentive Pooling

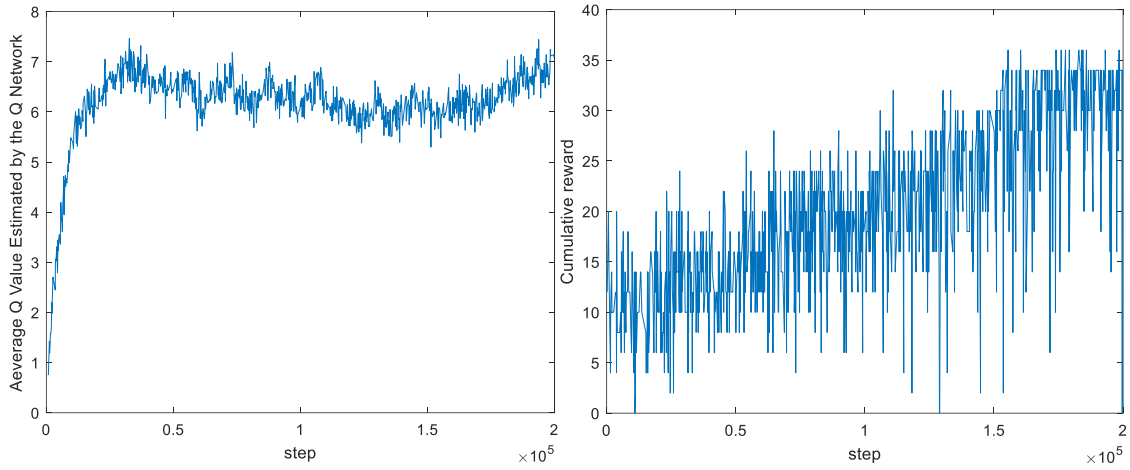


Figure 5-6: Results of D3QN-Discrete-GAT with Feature Compression

5.5. Results of PPO-Discrete-MLP

Now we consider the PPO algorithm. Table 5-4 shows the hyperparameters of the algorithm in this experiment.

Table 5-4: Hyperparameters of PPO-Discrete-MLP in Filling Grid Test

Hyperparameter	Value
KL_{target}	0.05
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	200000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2/0.1/0.05/0.02
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 512 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01

As mentioned in section 4.3.1, the policy update of PPO should not be too quick, otherwise the training process will become unstable. There are 2 ways to control the Kullback-Leibler divergence: policy gradient clipping with ξ or learning rate annealing with KL_{target}.

We first check the performance of the first approach. Figure 5-7 shows the estimated Kullback-Leibler divergence without learning rate annealing. Note that we use a sliding average filter to make this result more readable. The divergence starts from a small value at the beginning and then increase. Reducing the clipping ratio ξ does limit the KL divergence when $\xi > 0.1$. However, such effect is not obvious when $\xi < 0.1$. At the same time, the KL divergence is usually under 0.2 when $\xi < 0.1$.

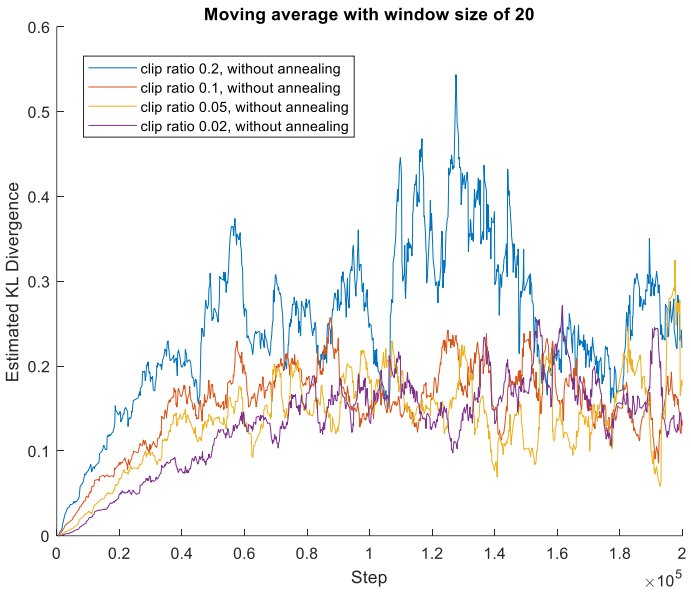
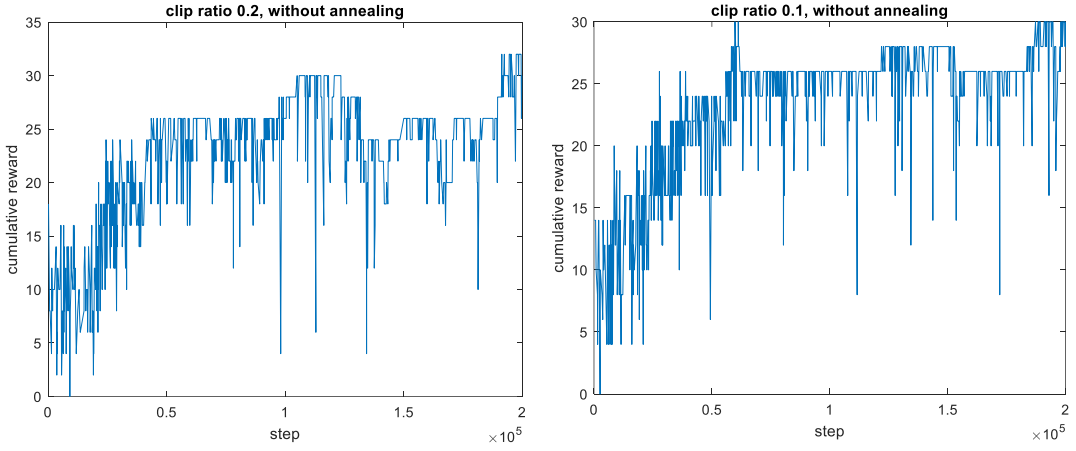


Figure 5-7: KL Divergence of PPO-Discrete-MLP without Learning Rate Annealing

Figure 5-8 shows the cumulative rewards per episode. Although the KL divergence is limited by the clipping with ξ, the algorithm without learning rate annealing is still unstable: its cumulative reward increases and then suddenly drops. Another problem is that the cumulative rewards do not reach the up limit of the toy problem, i.e., 36.



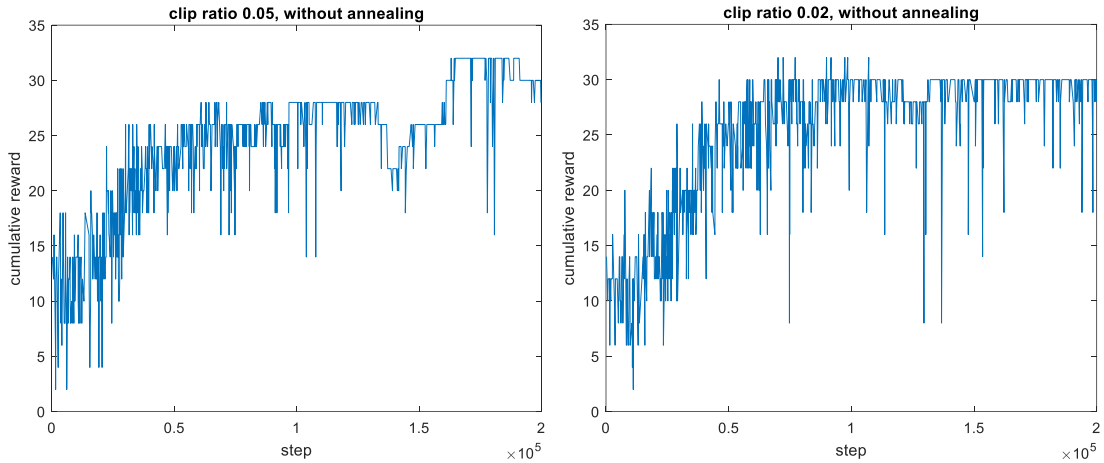


Figure 5-8: Cumulative Reward (per episode) without Learning Rate Annealing

Another metric is the entropy of the action probability distribution under the policy. The entropy of a probability distribution is computed by:

$$entropy = -\sum_{a} \pi_w(a|s_t) \cdot \log\left(\pi_w(a|s_t)\right) \tag{5-1}$$

In this experiment, the number of actions is 36. Assume the policy is fully random, $\pi_w(a|s_t)$ should be $1/36$, and the entropy is about 3.58. Figure 5-9 shows the average entropy of episodes without learning rate annealing, which starts from 3.58 and then decreases to 0.5. It means the policy becomes more deterministic.

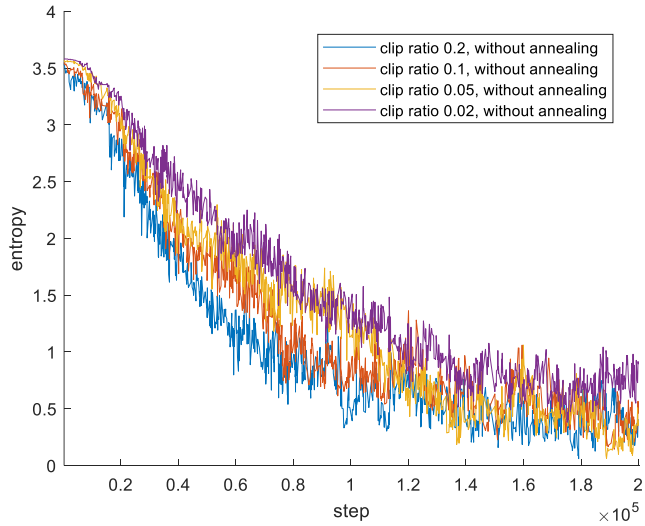


Figure 5-9 Average Entropy of PPO-Discrete-MLP without Learning Rate Annealing

Figure 5-10 shows the algorithm's performance with learning rate annealing. In the experiment, the clip ratio ξ is set to 0.2 and the KL_{target} is set to 0.05. Unlike Figure 5-7, the results are not filtered.

In Figure 5-10, the divergence with learning rate annealing is always below 0.14 and usually around 0.025 (half of KL_{target}). The learning rate starts from 5e-4 and decreases to 5e-5. As a result, the cumulative reward per episode grows stably and does not crash like in Figure 5-8. The annealing mechanism controls the KL divergence and makes the training process more stable.

Surprisingly, the final entropy in Figure 5-10 (around 2.2) is higher than in Figure 5-9 (around 0.5). If we reduce the learning rate to 1e-5 and do not adopt annealing, the final entropy is also around 2.2 (the result is not shown here). That is to say, the policy learned with a small learning rate or the learning rate annealing mechanism is more random than the policy with a constant high learning rate. However, the cumulative reward obtained by the policy is equal to or even slightly higher than the more deterministic policy.

One possible reason is that if the policy is too deterministic, a minor update to the weights of the actor-network can cause a significant change in the action probability distribution $\pi_w(a|s_t)$. It will lead to a high KL divergence and an unstable training process. To avoid that, the annealing mechanism reduces the learning rate before $\pi_w(a|s_t)$ becomes too sharp. A constant small learning rate may also have the same effect, but we should run the algorithm several times to find a reasonable learning rate.

To avoid repeated tuning of the learning rate during the study, we adopt the learning rate annealing mechanism in the following experiments.

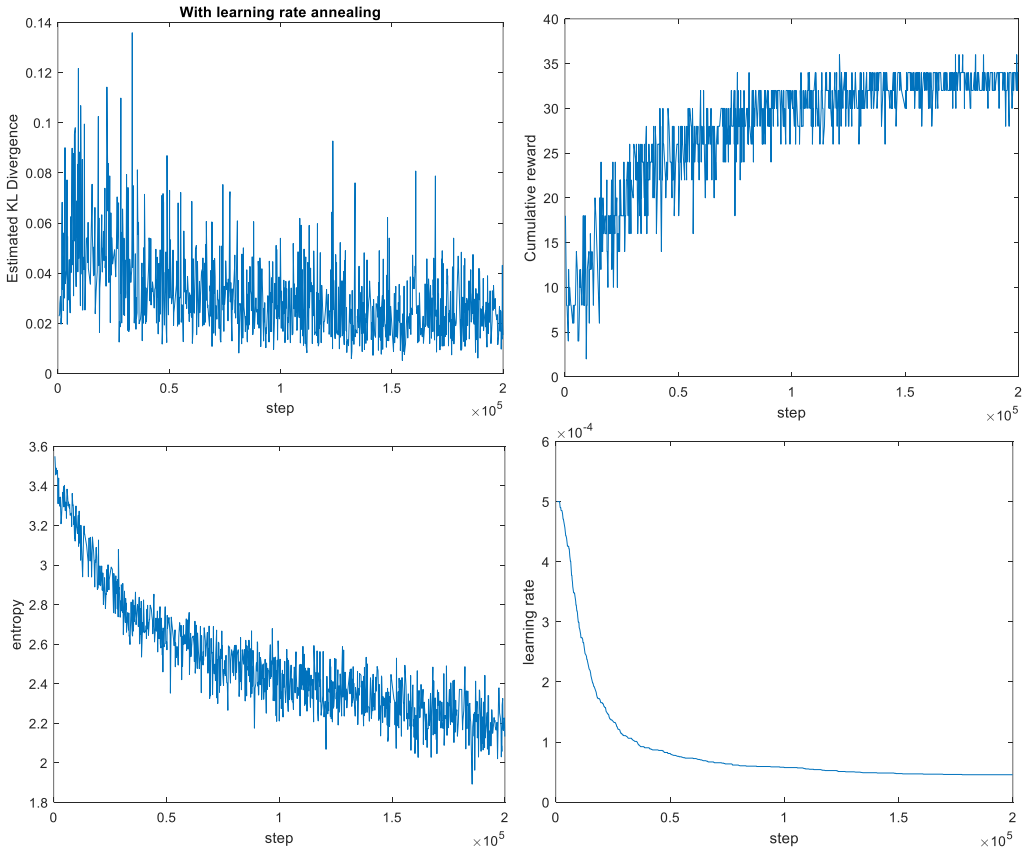


Figure 5-10: Results of PPO-Discrete-MLP without Learning Rate Annealing

5.6. Results of PPO-MultiDiscrete-MLP

In this experiment, an action is a vector of length 4, and the episode length is 9. The PPO-MultiDiscrete-MLP algorithm uses the following hyperparameters:

Table 5-5: Hyperparameters of PPO-MultiDiscrete-MLP in Filling Grid Test

Hyperparameter	Value
KL_{target}	0.05
Mini-batch size	32

Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	200000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 512 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01

The results are shown in Figure 5-11. In the “filling grid” environment, the algorithm has the similar performance as the PPO-Discrete-MLP algorithm. The entropy is higher because the action vector is longer, equivalent to having more moves to choose from.

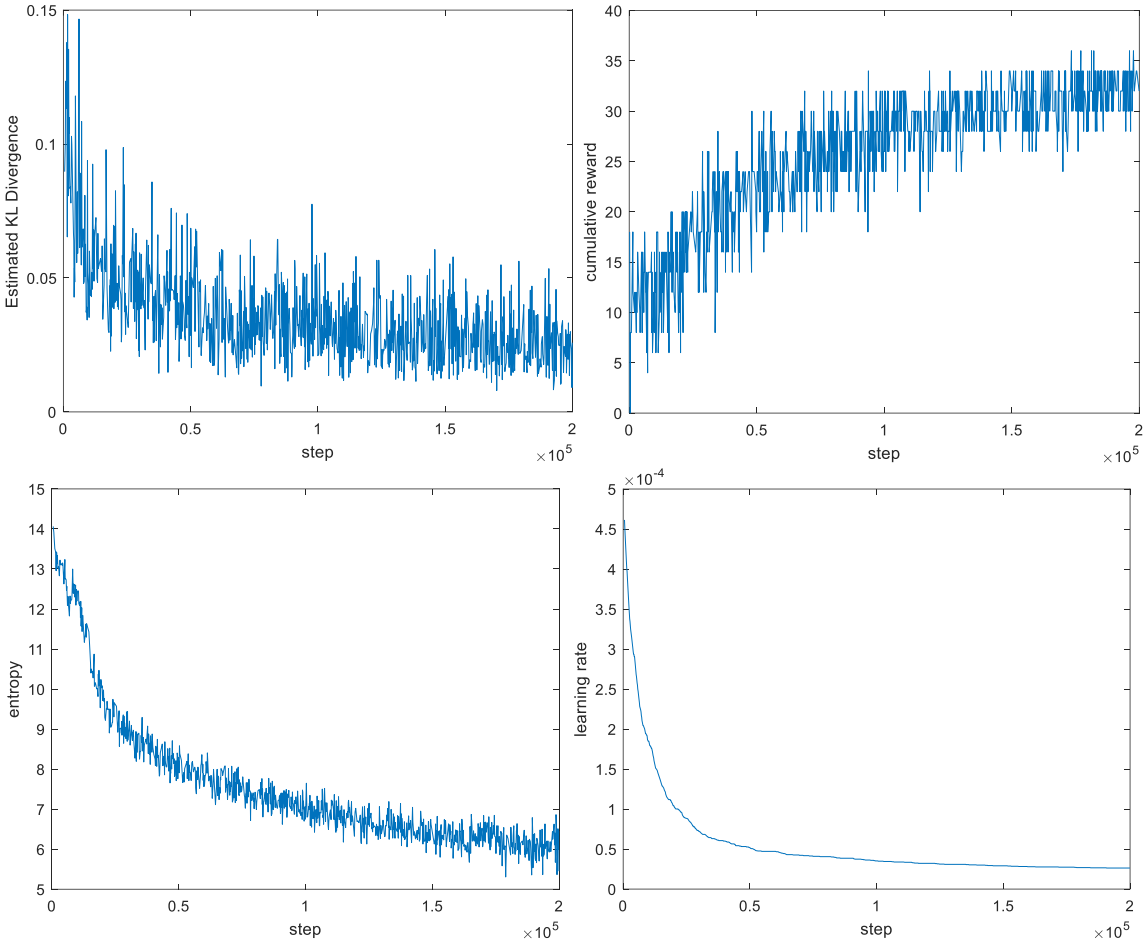


Figure 5-11: Results of PPO-MultiDiscrete-MLP on the Toy Problem

5.7. Results of PPO-Discrete-GGNN

The section shows the results of the PPO algorithm with the Gated Graph Neural Network (GGNN). Table 5-6 shows the hyperparameters of the PPO-Discrete-GGNN algorithm in this experiment:

Table 5-6: Hyperparameters of PPO-Discrete-GGNN in Filling Grid Test

Hyperparameter	Value
KL_{target}	0.05
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	200000 (Feature Compression) / 1M (Attentive Pooling)
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Shared network:

	Input->GGNN(3 layers)->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Critic: Graph embedding->MLP->$\hat{A}^{\pi}(s,a)$ Actor: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Other networks are initialized in the same way as PPO-Discrete-MLP
std_{critic}^{output}	1
std_{actor}^{output}	0.01

Figure 5-12 shows the performance of the attentive pooling approach. Like the D3QN-Discrete-GGNN configuration (Figure 5-3), the cumulative reward (per episode) increases slowly and takes about 700k steps to converge. When testing the real Delfi-PQ hardware, this approach may need significant time.

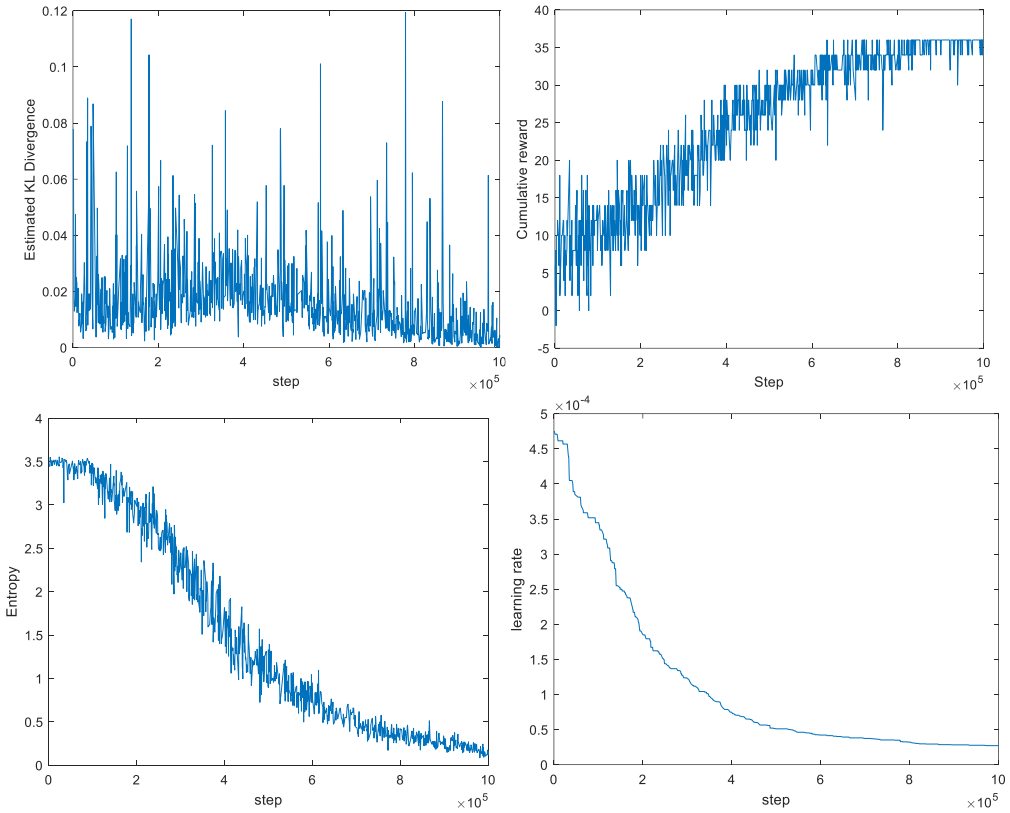


Figure 5-12: Results of PPO-Discrete-GGNN with Attentive Pooling on the Toy Problem

By contrast, Figure 5-13 shows the results of the feature compression approach. It converges much more quickly. In both cases, the entropy also drops to about 0.5, which means the agent learns a relatively deterministic policy.

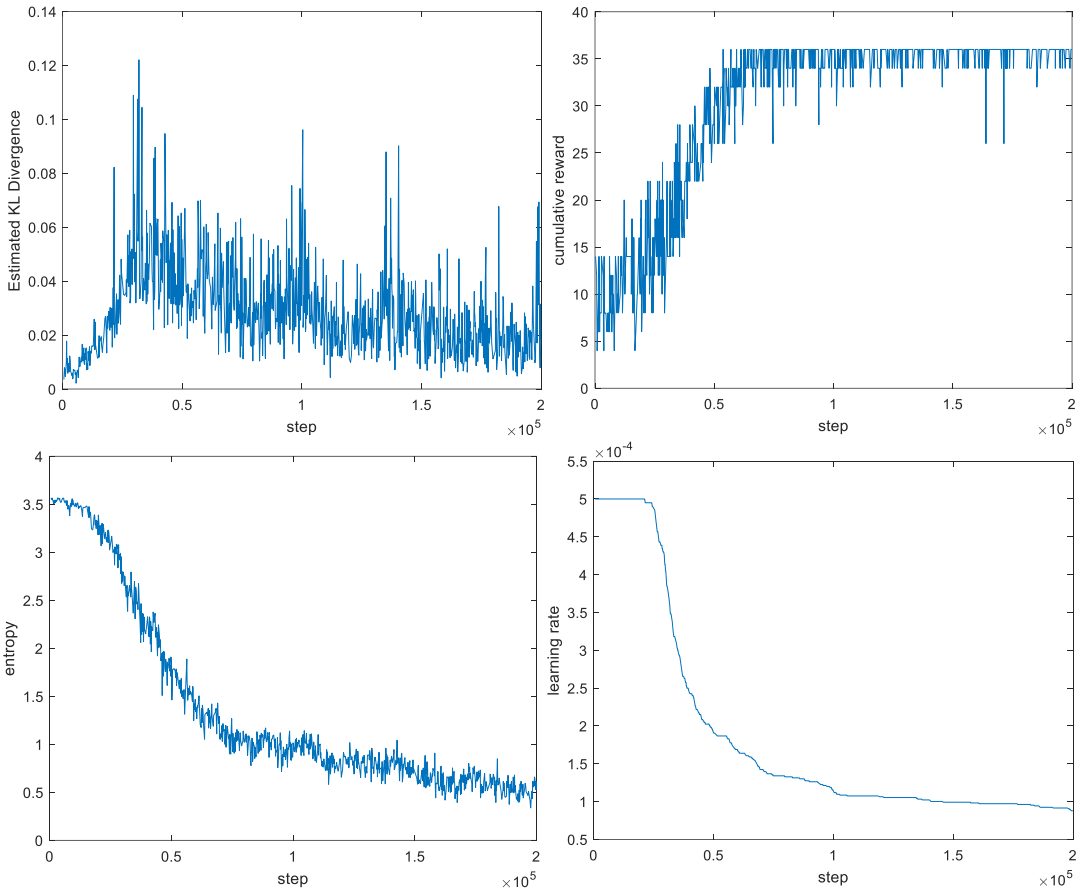


Figure 5-13: Results of PPO-Discrete-GGNN with Feature Compression on the Toy Problem

5.8. Results of PPO-Discrete-GAT

This configuration is like the PPO-Discrete-GGNN but uses the Graph Attention Network (GAT) to process the graph. Table 5-7 shows the hyperparameters of this configuration in the experiment. The configuration only uses 3 GAT layers, and more GAT layers do not improve the performance in this test.

Table 5-7: Hyperparameters of PPO-Discrete-GAT in Filling Grid Test

Hyperparameter	Value
KL_{target}	0.05
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	200000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Shared network:

	Input->GAT->ReLU->GAT->ReLU->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Critic: Graph embedding->MLP->$\hat{A}^{\pi}(s,a)$ Actor: Graph embedding->MLP->advantage
Network Initialization	GAT adopts the default initialization. Other networks are initialized in the same way as PPO-Discrete-MLP
std_{critic}^{output}	1
std_{actor}^{output}	0.01

Figure 5-14 shows the results of the configuration with attentive pooling. The cumulative reward does not reach the up limit (36 per episode) after 1 million steps. Using attentive pooling in this configuration may not be a good idea.

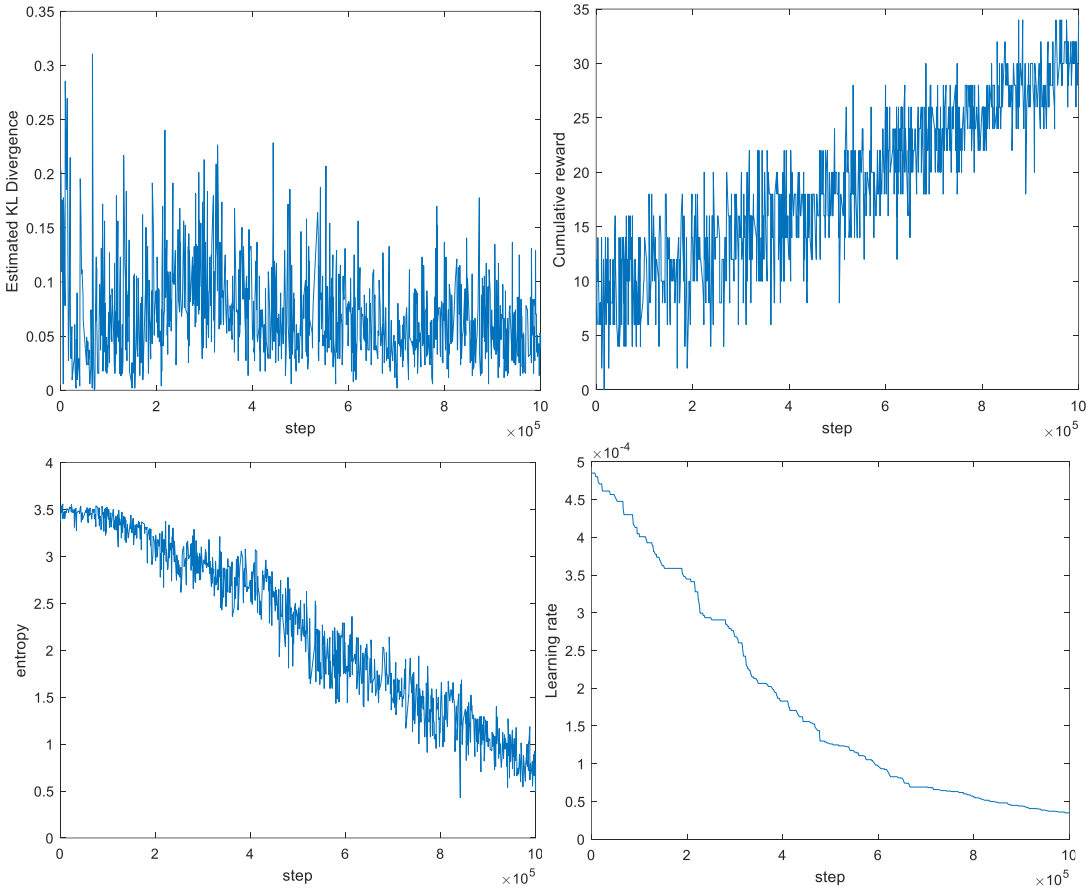


Figure 5-14: Results of PPO-Discrete-GAT with Attentive Pooling on the Toy Problem

Figure 5-15 shows the performance of feature compression. Compared with PPO-Discrete-GGNN, this configuration needs more time steps to converge. Moreover, when the cumulative curve reaches the up limit (36 per episode), the curve begins to oscillate violently and sometimes drops under 0. Such instability is not desired by software testing.

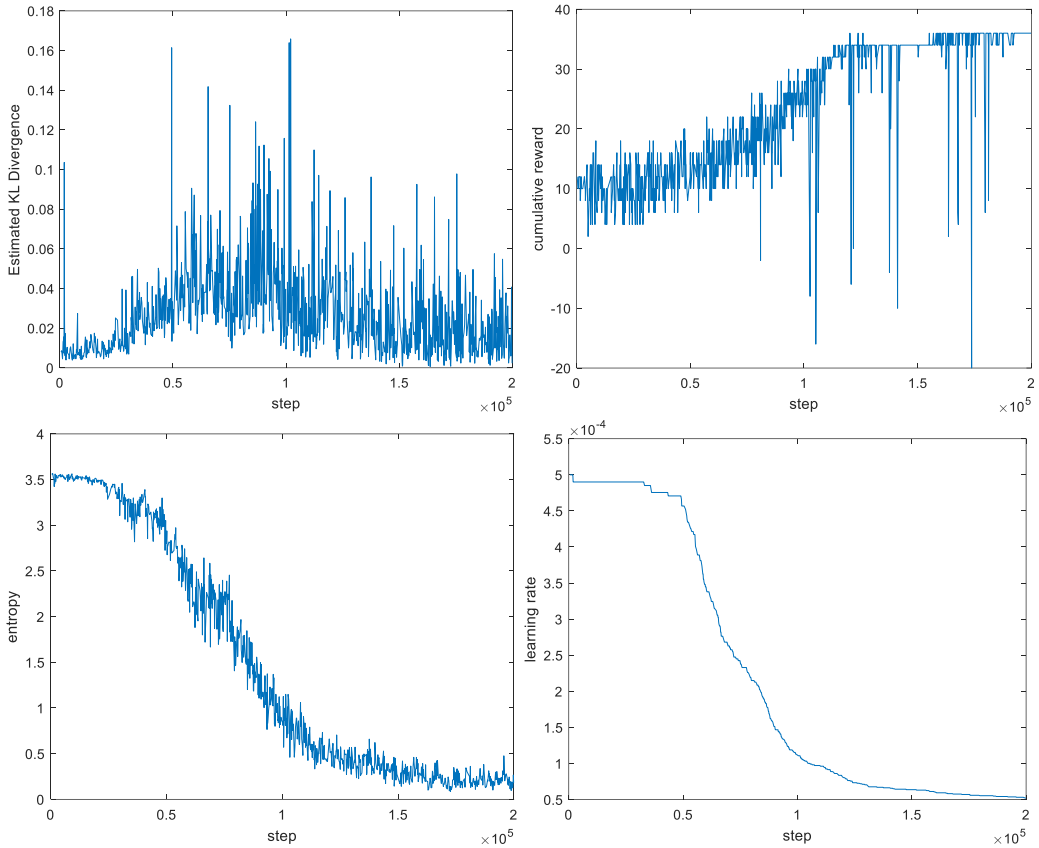


Figure 5-15: Results of PPO-Discrete-GAT with Feature Compression on the Toy Problem

5.9. Discussions

Table 5-8 summarizes all experiments in this chapter. Note that there are 36 grids in the toy problem.

Table 5-8: Brief Summary of Results in Filling Grid Test

Configuration	Number of Steps Needed by Convergence of Cumulative Reward	Fill All Grids?	Note
D3QN-Discrete-MLP	10k	Yes (36)	
D3QN-Discrete-GGNN (Attentive Pooling)	170k	~35 grids	Its Q value curve converges more slowly than the feature compression approach.
D3QN-Discrete-GGNN (Feature Compression)	155k	~35 grids	The cumulative reward curve fluctuates. Sometimes it drops below 0.
D3QN-Discrete-GAT (Attentive Pooling)	160k	~32 grids	Its Q value curve converges more slowly than the feature compression approach.
D3QN-Discrete-GGNN (Feature Compression)	150k	~32 grids	The cumulative reward curve fluctuates. Sometimes it drops below 0.
PPO-Discrete-MLP (Without Annealing)	200k	~30 grids	KL divergence ranges between 0.1 and 0.5. The cumulative reward curve fluctuates or even crashes. The entropy drops from 3.5 to 0.5.
PPO-Discrete-MLP (With Annealing)	150k	~33 grids	KL divergence around 0.025. The cumulative reward curve grows

			stably. The entropy drops from 3.5 to 2.2.
PPO-MultiDiscrete-MLP (With Annealing)	150k	~33 grids	The cumulative reward curve grows stably. The entropy drops from 14 to 6 (different action space).
PPO-Discrete-GGNN (With Annealing, Attentive Pooling)	700k	~35 grids	The cumulative reward curve grows stably. The entropy drops from 3.5 to 0.2.
PPO-Discrete-GGNN (With Annealing, Feature Compression)	60k	~35 grids	The cumulative reward curve grows stably. The entropy drops from 3.5 to 0.5.
PPO-Discrete-GAT (With Annealing, Attentive Pooling)	Not yet after 1M steps	~30 grids	The cumulative reward curve grows stably. The entropy drops from 3.5 to 0.7.
PPO-Discrete-GAT (With Annealing, Feature Compression)	120k	Yes (36)	The cumulative reward curve fluctuates. Sometimes it drops under 0. The entropy drops from 3.5 to 0.2.

The D3QN-Discrete-MLP configuration converges most quickly in the toy problem. It only needs 10k steps to reach the up limit, while the PPO-Discrete-MLP configuration needs about 150k steps. Other configurations usually need 60k~200k steps to converge, while the PPO configurations with attentive pooling take much more training steps.

On the other hand, the D3QN algorithm cannot directly generate a command vector as PPO-MultiDiscrete-MLP does. In other words, it cannot adapt to the MultiDiscrete action space. Assume the length of the command vector is L_v, and each element of the vector can be selected from N values. The D3QN algorithm needs computer N^{L_v} state-action values in every step, which is computationally expensive. By contrast, the actor-network in the PPO-MultiDiscrete-MLP only needs to generate a matrix with a size of $N \times L_v$. It is much easier.

From our experience, the D3QN algorithm is not very sensitive to hyperparameters in the filling grid problem. An important consideration is to control the rate at which ε falls by tuning ε_{start}, ε_{end}, total time steps, and exploration fraction. If ε falls too quickly, the exploration will stops before the agent learns a good policy. If ε falls too slowly, the training process will take too much time.

The learning rate annealing mechanism can make PPO more stable, but it may also increase the final entropy of the action probability distribution. When the action probability distribution is too "sharp" (low entropy), the KL divergence of a single policy update can be huge, triggering the learning rate annealing mechanism. However, it is unclear why high-entropy and low-entropy policies have similar performance in the toy problem.

From a neural network architecture point of view, the simplest MLP (with only two hidden layers) is the fastest option. However, it only accepts a plain vector as input. The length of the vector and the meaning of each element may change after programmers modify the code (section 3.4.2). We shall use a graph neural network to process graph input to solve the problem.

The Gated Graph Neural Network (GGNN) performs similarly to the Graph Attention Network (GAT) in both the D3QN and PPO algorithms. However, the GGNN can have more layers (5 layers in our case), while the GAT usually has 2~3 layers. In our experiment, if the GAT has more layers, its performance will drop significantly.

There are two options to extract information from the node features processed by the graph neural network: graph pooling or feature compression. Table 5-8 shows that feature compression leads to the cumulative reward curve oscillation for both the D3QN and PPO algorithms. At the same time,

the neuron number of its graph MLP layer depends on the node number of the graph. This may also cause problems when the SUT is modified and the number of probes changes.

Another approach to extract information is attentive pooling. It does not have the oscillation phenomenon of the feature compression approach. Furthermore, the neuron number does not depend on the node number of the graph. However, it usually needs more time steps to be trained.

We will use these configurations in the following sections to test actual Delfi-PQ software.

6 Stress Testing

This section introduces the stress testing and its results:

- Section 6.1 explains the design of the experiment.
- Section 6.2~6.9 shows the results of different configurations, including the random baseline, the genetic baseline, Q-Learning, D3QN-Discrete-MLP., PPO-Discrete-MLP, PPO-MultiDiscrete-MLP, PPO-Discrete-GGNN, PPO-MultiDiscrete-GGNN.
- Section 6.10 introduces some bugs of the onboard software identified during the stress testing.
- Section 6.11 compares results of these configuration and gives a brief summary.

6.1. About the Experiment

We try to maximize the CPU load of the COMMS subsystem of Delfi-PQ in this experiment. More specifically, we estimate number of clock cycles required by each loop in the scheduler:

$$c_{loop} = f t_{interval} / n_{loop} \tag{6-1}$$

Where f is the clock frequency of the CPU. For MSP432, $f = 48000000$. $t_{interval}$ is the time interval being sampled, and n_{loop} is the number of loops executed by the scheduler during the time interval. We set the reward of each step as $c_{loop}/1000$.

Figure 6-1 shows the time interval used in the experiment. The time interval is not precisely the interval that recording n_{loop}. However, since the length of time to send 2 “retrieve loop count” commands should be the same, the length of the 2 intervals should also be the same.

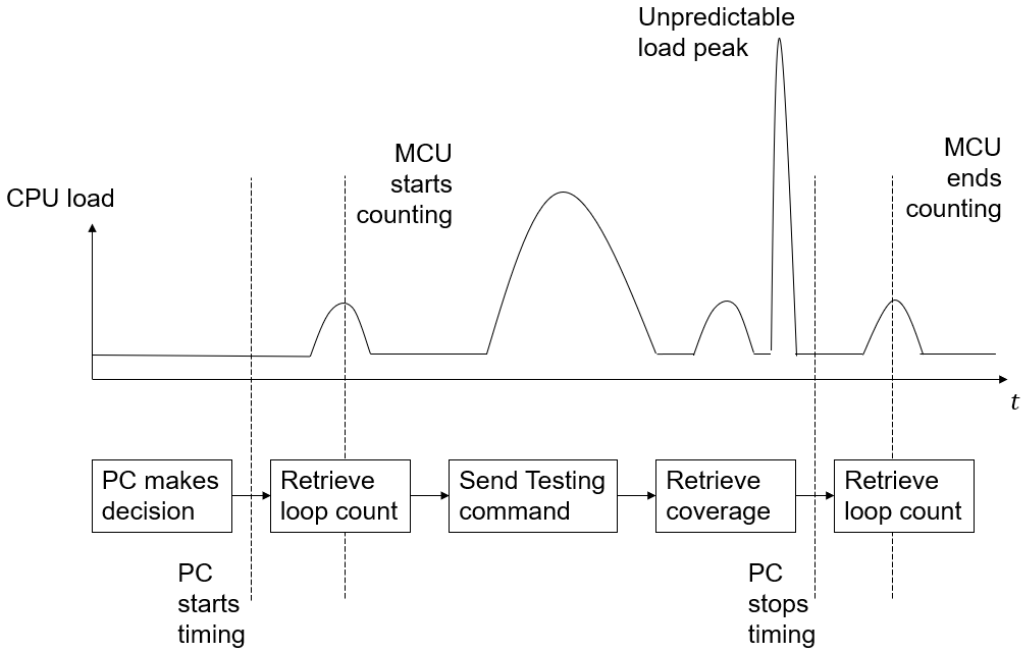


Figure 6-1: A Time Step in the Stress Testing

The processor is not always busy during the time interval. As shown by Figure 6-1, it usually gets busy for a while after receiving a command. On the other hand, we can only calculate the average c_{loop} during the time interval. Assume the communication speed decreases, $t_{interval}$ to send the 3 commands will get longer, but the CPU load triggered by the 3 commands will roughly remain the same. As a result, c_{loop} will be smaller. To avoid the influence of communication speed on the conclusions, we only compare the c_{loop} computed by the same machine, i.e., the lab PC or the server.

In Figure 6-1, one may notice that we always retrieve code coverage after sending a testing command. The coverage information is useless for some algorithms, e.g., the genetic algorithm. Why do we do this? Firstly, if the agent retrieves the loop count just after sending a testing command, it measures instantaneous CPU load with a very high variance. In practice, we want a high average CPU load rather than a high instantaneous load. Secondly, instead of delaying 50ms before checking the loop count, collecting code coverage is useful (for RL) and has the same time-delay effect.

Figure 6-2 shows an example of a time step in the experiment of PPO-MultiDiscrete-GGNN (section 6.9).

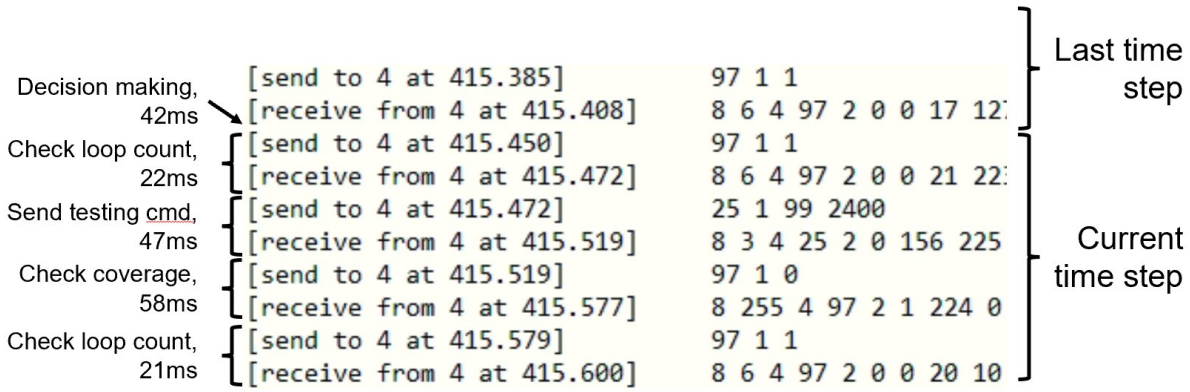


Figure 6-2: Example of a Time Step in Stress Testing (from Lab PC)

We also find unpredictable load peaks in the experiment, which means the stress testing environment is a non-deterministic environment for an RL agent. For example, if testers send the same testing command to the COMMS board 1000 times, they can record different CPU loads. Figure 6-3 shows the phenomenon.

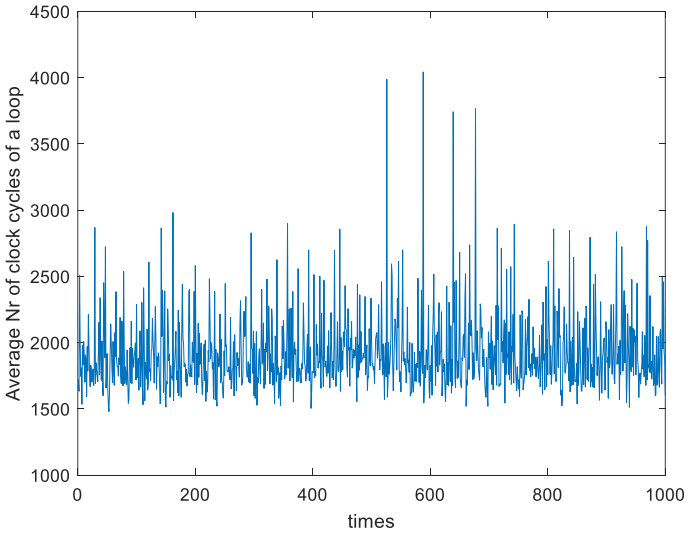


Figure 6-3: CPU Load of Send the Same Command for 1000 Times (from Lab PC)

When we perform the test, there are 630 probes in the source code and 58 commands specified by the tester.

6.2. Random Baseline

In the random baseline, the agent randomly selects a command from the command list and send it. Figure 6-4 shows the results of the baseline.

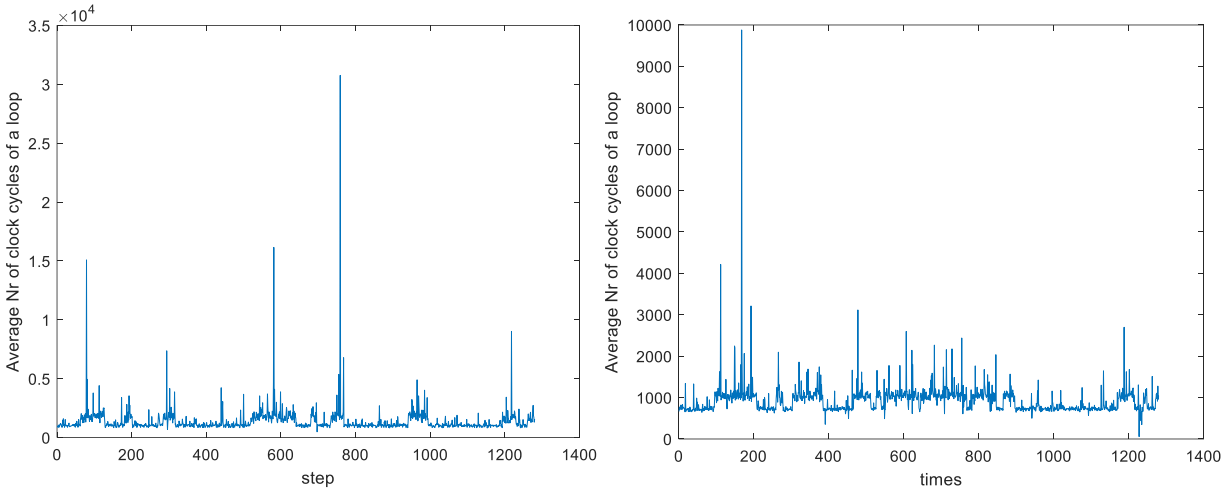


Figure 6-4: Results of Random Baseline in Stress Testing of COMMS (Left: Commands from Lab PC; Right: from the Cloud Server)

The CPU load peaks may be triggered by specific commands or unpredictable events. For the commands send by the lab PC, $c_{loop} \approx 900$. For the commands send from the cloud server, $c_{loop} \approx 700$. Obviously, the performance of the baseline will not improve during a test.

6.3. Genetic Algorithm Baseline

The genetic algorithm maintains a group of solutions called chromosomes and modifies them to get a higher cumulative reward. Table 6-1 shows the hyperparameters of the algorithm.

Table 6-1: Hyperparameters of the Genetic Algorithm in Stress Testing of COMMS

Total iteration number	100
Length of a solution	128 (Discrete) / 512 (MultiDiscrete)
Population	20
Crossover probability	0.4
Mutation probability	0.01

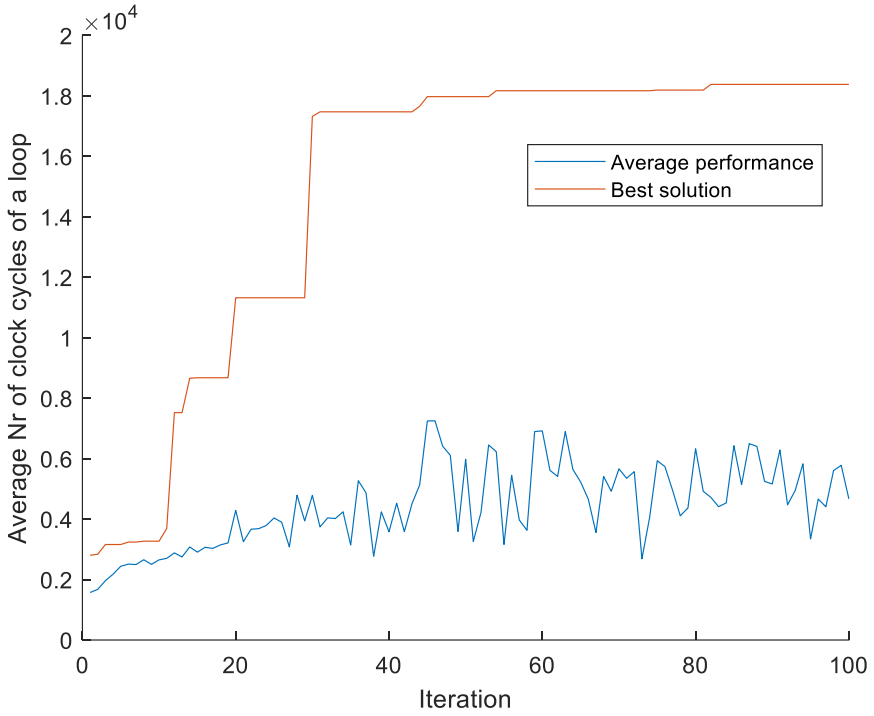


Figure 6-5: Results of the Genetic Algorithm (Discrete, from Lab PC)

In Figure 6-5, the genetic algorithm selects pre-defined commands from the parser, i.e., each element in the chromosome is an index of a human-defined command. As mentioned before, such action space is discrete. The testing commands come from the lab PC, and it takes 10.3 hours to complete the test. The curve converges after 40 iterations and 4.1 hours.

The average CPU load of the solutions in an iteration increases from 1600 to around 4500. The best command sequence found by the algorithm reaches a CPU load of 18360. However, when we repeat the command sequence ten times, we get the following results:

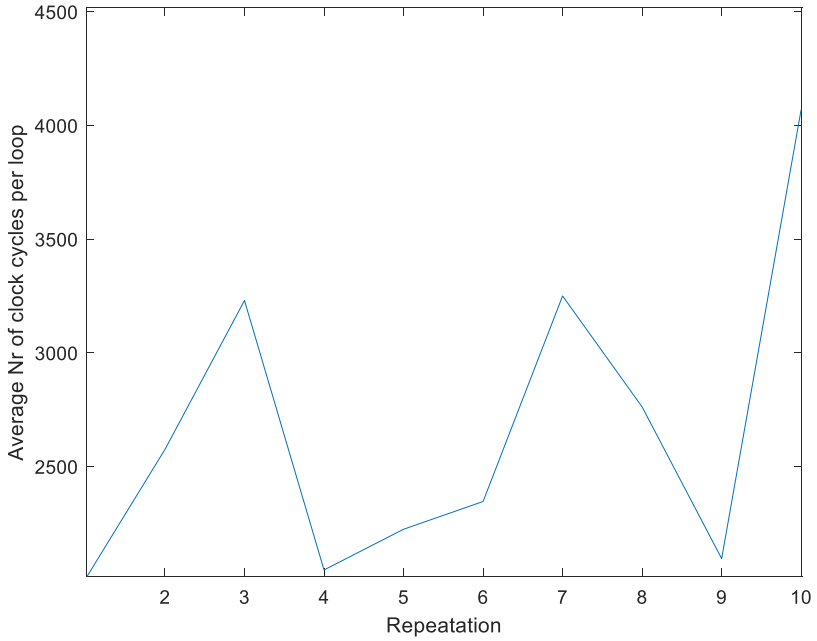


Figure 6-6: Repeat the "Best Command Sequence (Discrete)" for 10 Times

As shown by Figure 6-6, the performance of the best command sequence is not repeatable. It is because the stress testing is non-deterministic.

In contrast, Figure 6-7 shows the algorithm's performance in the MultiDiscrete action space. In such action space, the length of a chromosome is 512. Every four elements in the chromosome form a command vector, in which the first element is the service number, and the following three elements are the command payload.

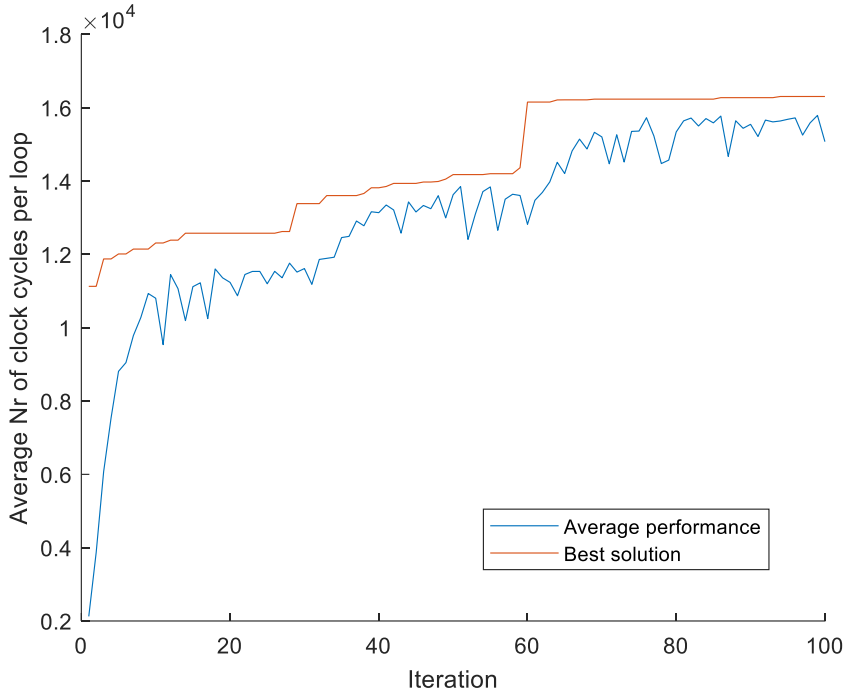


Figure 6-7: Results of the Genetic Algorithm (MultiDiscrete, from Lab PC)

The testing commands come from the lab PC, and it takes 12.9 hours to complete the test. The curve converges after 70 iterations and 9 hours. The average CPU load of an iteration increases from 2000 to about 15000, which is much higher than the CPU load in the discrete action space. It means the human-defined commands are not good at triggering a high CPU load.

We ran the best solution ten times and got the following result:

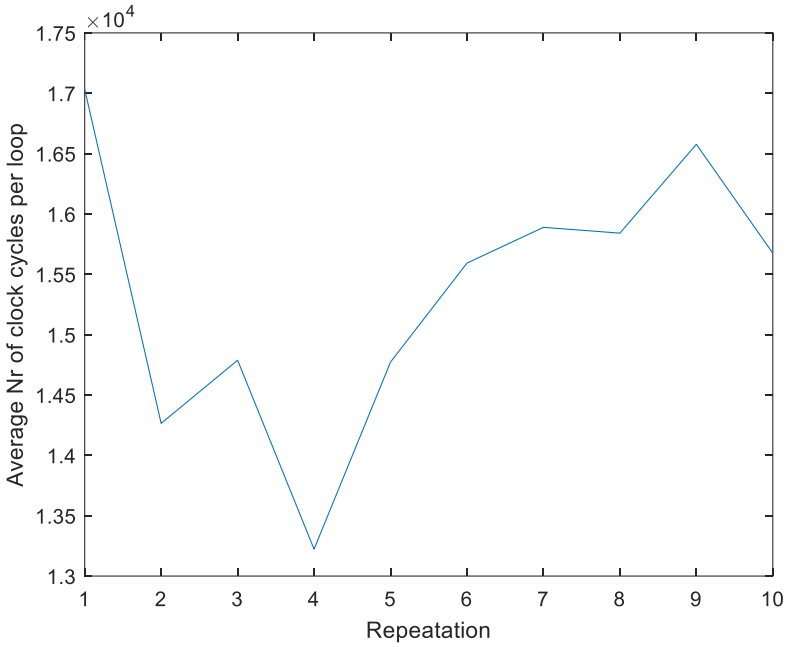


Figure 6-8: Repeat the "Best Command Sequence (MultiDiscrete)" for 10 Times

The performance of the best command sequence is still not repeatable.

6.4. Results of Q-Learning

Table 6-2 shows the hyperparameters in the tabular Q-Learning algorithm in this test, and the results are shown in Figure 6-9.

Table 6-2: Hyperparameters of the Q-Learning in Stress Testing for COMMS

Hyperparameter	Value
γ	0.9
Learning rate	0.01
Total time steps	65000
Episode length	128
ε	0.9

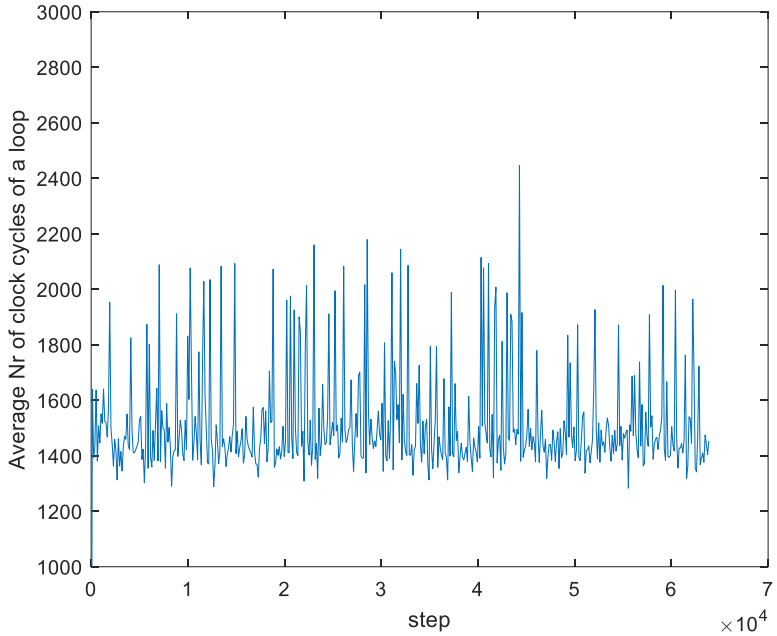


Figure 6-9: Performance of Q-Learning in the Stress Test of COMMS (from Lab PC)

The Q-Learning algorithm found 21 states during the test using the hyperparameters and state identification rules. However, the CPU load does not increase in the training process. One possible reason is that the state identification rules are not effective enough.

6.5. Results of D3QN-Discrete-MLP

Now we perform the stress testing with the D3QN-Discrete-MLP configuration. The hyperparameters are shown in Table 6-3, and the results are shown in Figure 6-10.

According to our experience in the "filling grid" experiment, the D3QN algorithm usually converges in fewer time steps when the training frequency is 1. However, frequently updating the neural network requires more computing time. Thus, we run the algorithm on the cloud server to reduce the computing time.

Table 6-3: Hyperparameters of D3QN-Discrete-MLP in the Stress Testing for COMMS

Hyperparameter	Value
γ	0.9
Buffer size	50000
Mini-batch size	128
Learning rate	5e-4/1e-4
Learning starts	128
Training frequency	1
Total time steps	50000
ε_{start}	1
ε_{end}	0.01
Exploration fraction	0.8
Epoch length	128
N_{target}	500
Neural Network Design	Value network: Input->MLP->ReLU->MLP->ReLU->MLP->value Advantage network: Input->MLP->ReLU->MLP->ReLU->MLP->advantage

	Every hidden layer has 2048 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$.

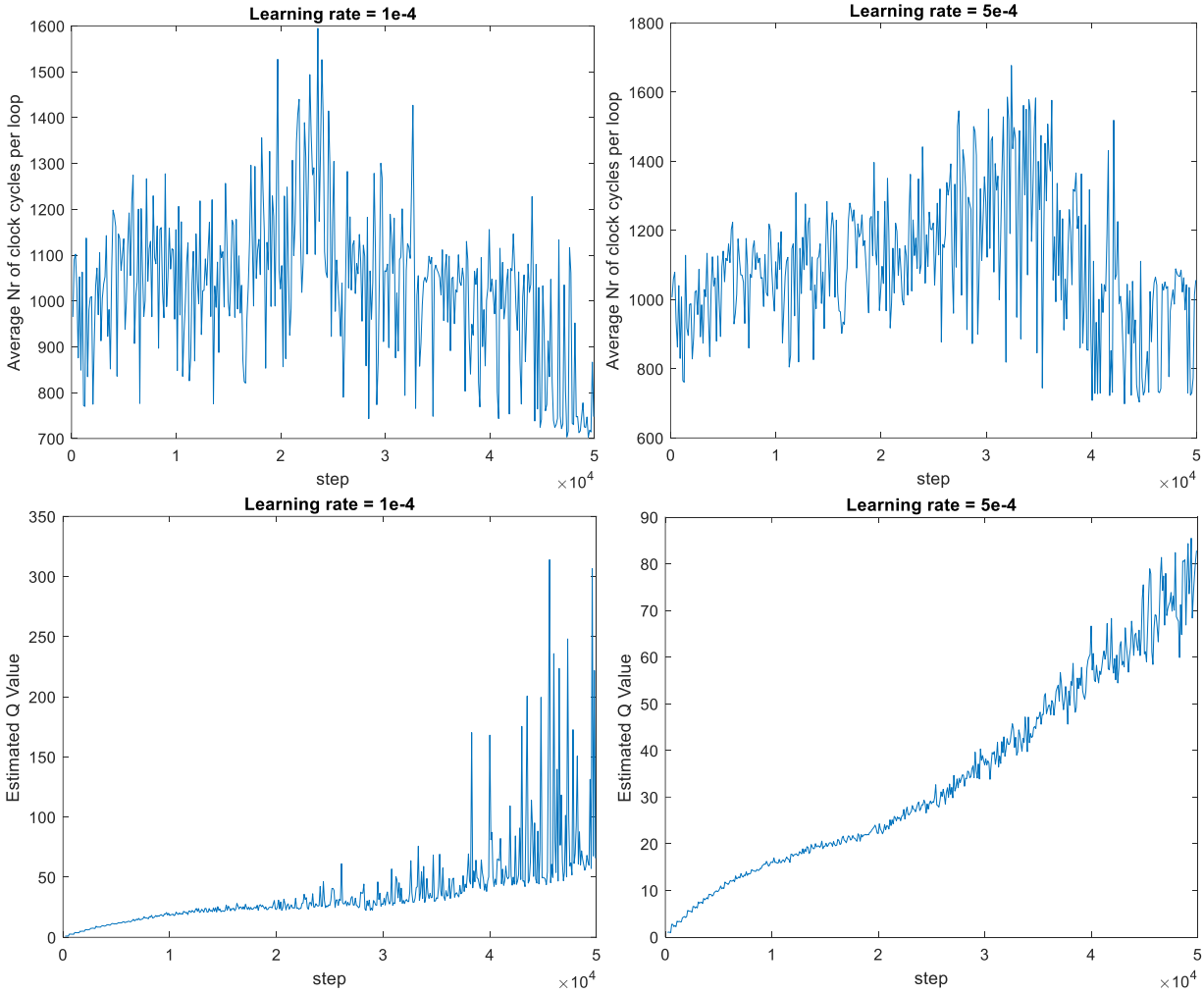


Figure 6-10: Performance of D3QN-Discrete-MLP in the Stress Test of COMMS (from Server)

The CPU load in Figure 6-10 first increases and then decreases. At the same time, the estimated Q value diverges at the end of the training process. Reducing the learning rate makes the situation even worse.

There are several possible reasons for the phenomenon. Firstly, the neural network may be overfitting. To avoid over-fitting, we can add an L2 regulation term to the loss function (4-3) or add dropout layers to the neural network. Secondly, such instability may be an inherent property of value-based RL algorithms because minor updates to $\hat{Q}^{\pi}(s,a)$ may significantly change the policy and the data distribution.

The D3QN-Discrete-MLP is the most basic configuration of the D3QN algorithm. Since it does not work well in stress testing, we will not try other configurations based on the D3QN algorithm. To some extent, this is also a result of the limited time available for the work.

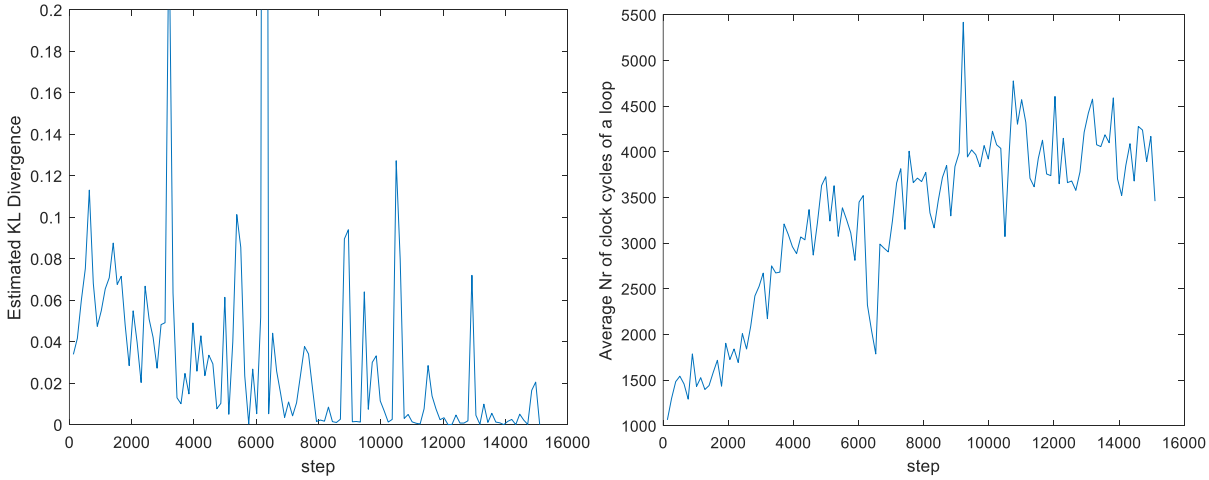
6.6. Results of PPO-Discrete-MLP

We try the PPO-Discrete-MLP configuration in this section. Table 6-4 lists the hyperparameters used, and Figure 6-11 shows the results. In the experiment, we find that the configuration can converge quickly without frequent network updates, so we run the algorithm on the lab PC.

The D3QN-Discrete-MLP is the most basic configuration of the D3QN algorithm. Since it does not work well in stress testing, we will not try other configurations based on the D3QN algorithm. To some extent, this is also a result of the limited time available for the work.

Table 6-4: Hyperparameters of PPO-Discrete-MLP in the Stress Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.01
c_{vf}	0.5
Total time steps	15000
Epoch length	128
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 2048 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01



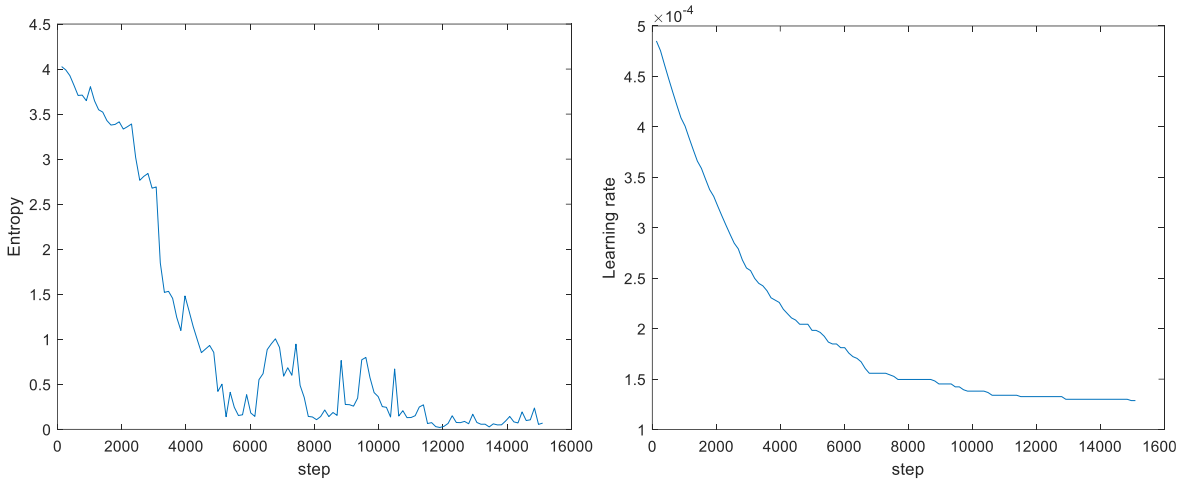


Figure 6-11: Performance of PPO-Discrete-MLP in the Stress Test of COMMS (from Lab PC)

The performance of this configuration is prospective. It only takes 52 minutes to get the result on the lab PC, and the CPU load maintains a level of around 4000 after 30 minutes. By contrast, the genetic algorithm in discrete action space performs similarly after 30 iterations, i.e., about 3 hours. At the same time, the genetic algorithm can only generate several good command sequences. It has non-deterministic performance under a non-deterministic environment. In contrast, the PPO algorithm generates a policy which has relatively stable performance in a non-deterministic environment.

Since the PPO-Discrete-MLP configuration works well, we keep trying other configurations based on the PPO algorithm.

6.7. Results of PPO-MultiDiscrete-MLP

We try the PPO-MultiDiscrete-MLP configuration in this section. Table 6-5 shows the hyperparameters. Figures 6-12 and 6-13 show the results.

Table 6-5: Hyperparameters of PPO-MultiDiscrete-MLP in the Stress Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	50000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 2048 neurons.

Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01
Command Vector Length	4/11

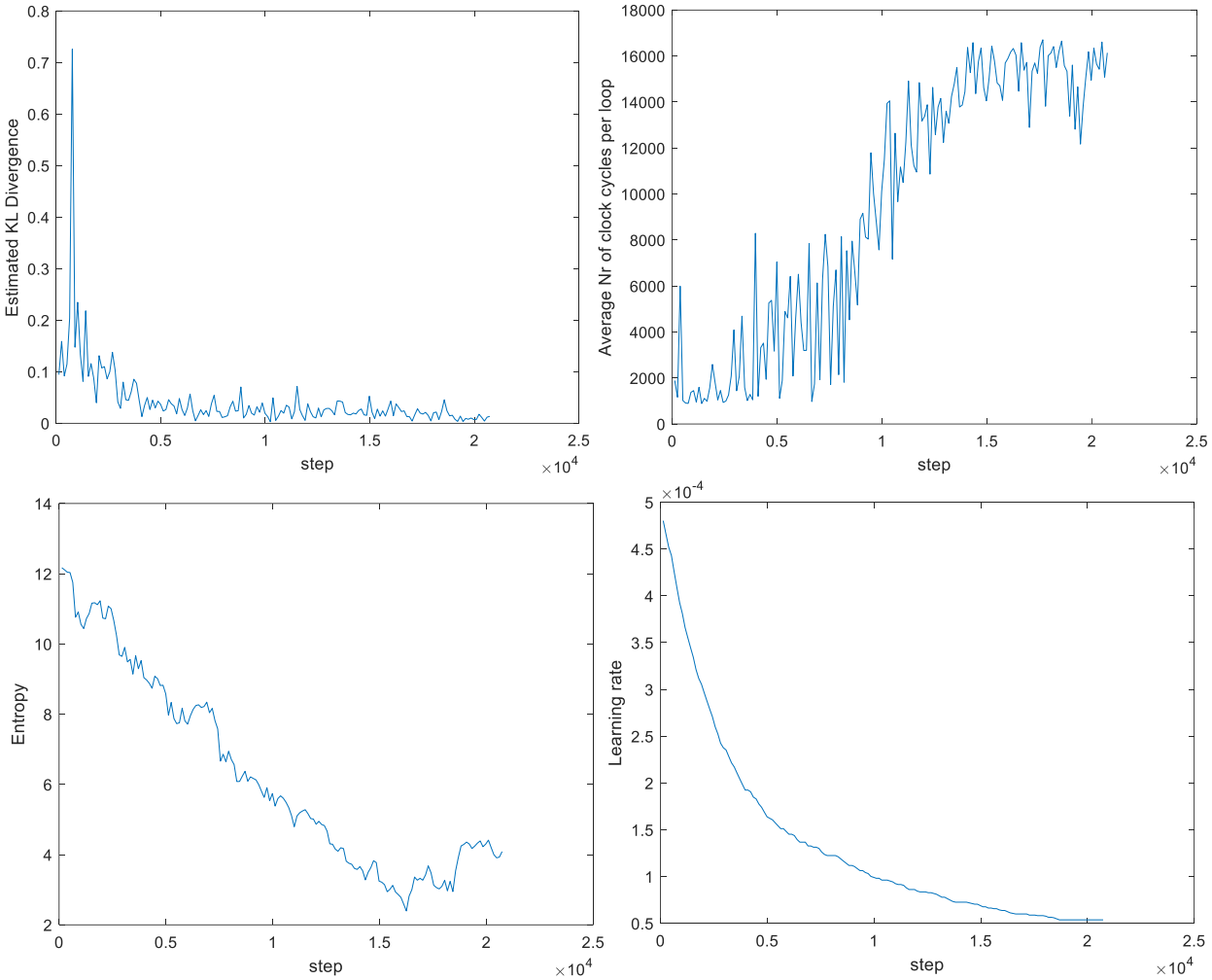


Figure 6-12: Performance of PPO-MultiDiscrete-MLP (Command Vector Length=4, from Lab PC)

In Figure 6-12, the configuration achieves a similar CPU load as the genetic algorithm in MultiDiscrete action space. However, the genetic algorithm takes 12.9 hours, while the PPO-MultiDiscrete-MLP configuration only takes 59 minutes on the lab PC. If we only consider the time needed for the curve to converge, the genetic algorithm converges after 9 hours, and PPO-MultiDiscrete-MLP converges after 44 minutes.

After convergence, the trained policy can achieve a c_{loop} between 14000 and 16000. By contrast, the best chromosome from the genetic algorithm can achieve a c_{loop} between 13000 and 17000. The policy from the PPO algorithm has relatively more stable performance.

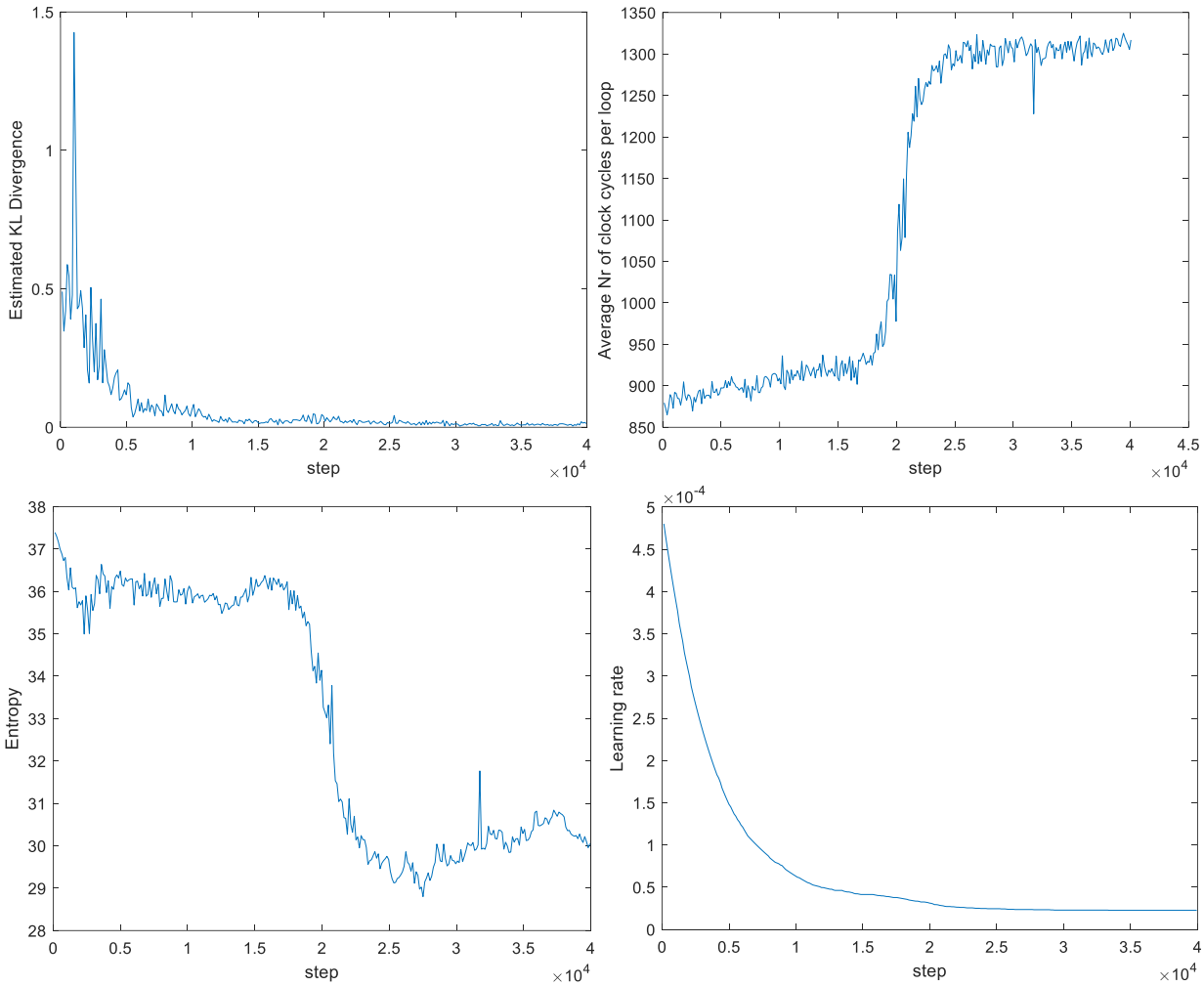


Figure 6-13: Performance of PPO-MultiDiscrete-MLP (Command Vector Length=11, from Lab PC)

Figure 6-13 shows the same configuration results, but the command vector length is set to 11. The lab PC takes 2 hours and 6 minutes to get the results. Although the CPU load does increase and the entropy does drop, its CPU load is much lower than in Figure 6-12. Why does this happen?

A possible reason is that it takes many attempts for the agent to get a valid command. In another experiment, after sending 200000 testing commands to the COMMS board, we only find valid command vectors with 4~5 parameters in the interaction record. If a valid command has more parameters, the probability of this command being attempted becomes very small. In other words, the action space for the PPO algorithm becomes too large to try.

6.8. Results of PPO-Discrete-GGNN

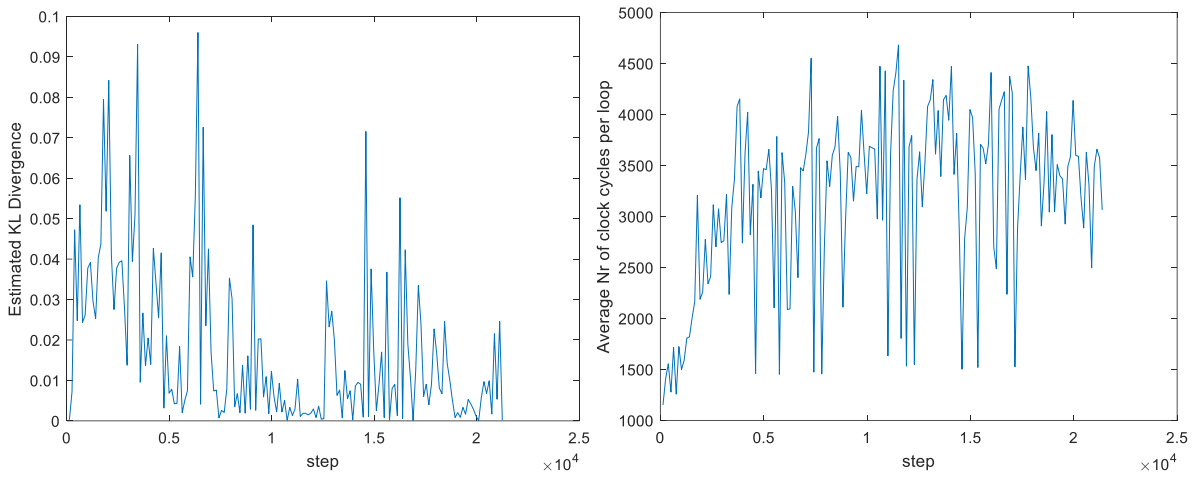
This section tries the PPO-Discrete-GGNN configuration. Note that the PPO-Discrete-GAT configuration performs worst in the filling grid test, so we do not try it in a real environment.

Table 6-6: Hyperparameters of PPO-Discrete-GGNN in the Stress Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9

c_{ent}	0.001
c_{vf}	0.5
Total time steps	25000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Shared network: Input->GGNN(3 layers)->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Critic: Graph embedding->MLP->$\hat{A}^{\pi}(s,a)$ Actor: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Other networks are initialized in the same way as PPO-Discrete-MLP
std_{critic}^{output}	1
std_{actor}^{output}	0.01

Figure 6-14 shows the results with feature compression. It takes 2 hours and 47 minutes on the lab PC to get the result. The average $c_{loop} \approx 3500$, which is lower than the c_{loop} of PPO-Discrete-MLP (Figure 5-23). The performance is also more unstable: sometimes c_{loop} drops under 1500.



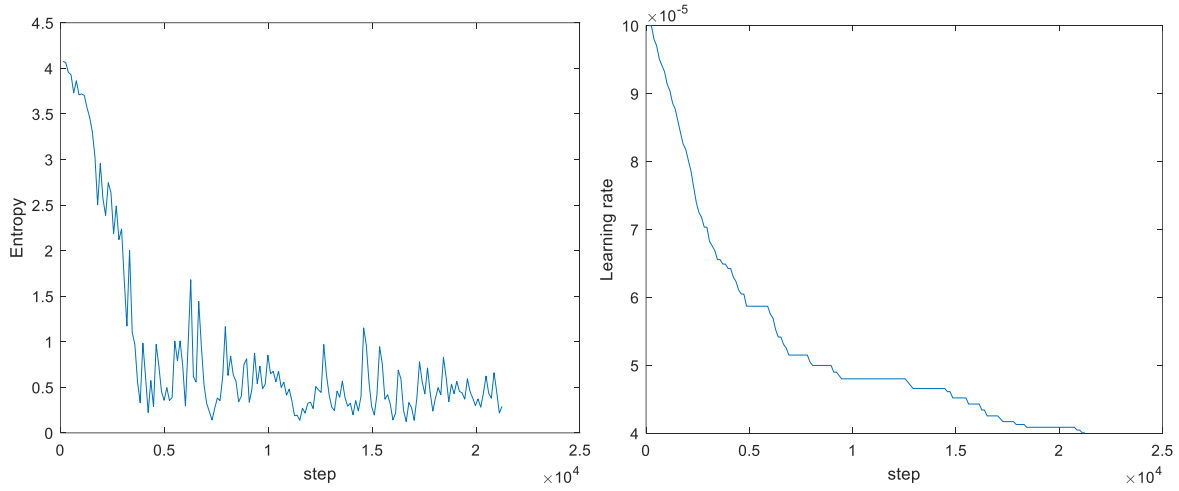


Figure 6-14: Performance of PPO-Discrete-GGNN with Feature Compression (from Lab PC)

By contrast, Figure 6-15 includes the results with attentive pooling. It takes 1 hour and 51 minutes on the lab PC to get the results. The configuration converges after about 3000 steps (26 minutes). Its average $c_{loop} \approx 4500$, which is higher than the performance of PPO-Discrete-MLP configuration. It also shows that attentive pooling performs better than feature compression in stress testing.

However, compared with the basic PPO-Discrete-MLP configuration, the CPU load curve in Figure 6-15 is still more unstable. Sometimes c_{loop} drops under 2000. This phenomenon may be caused by the Gated Graph Neural Network.

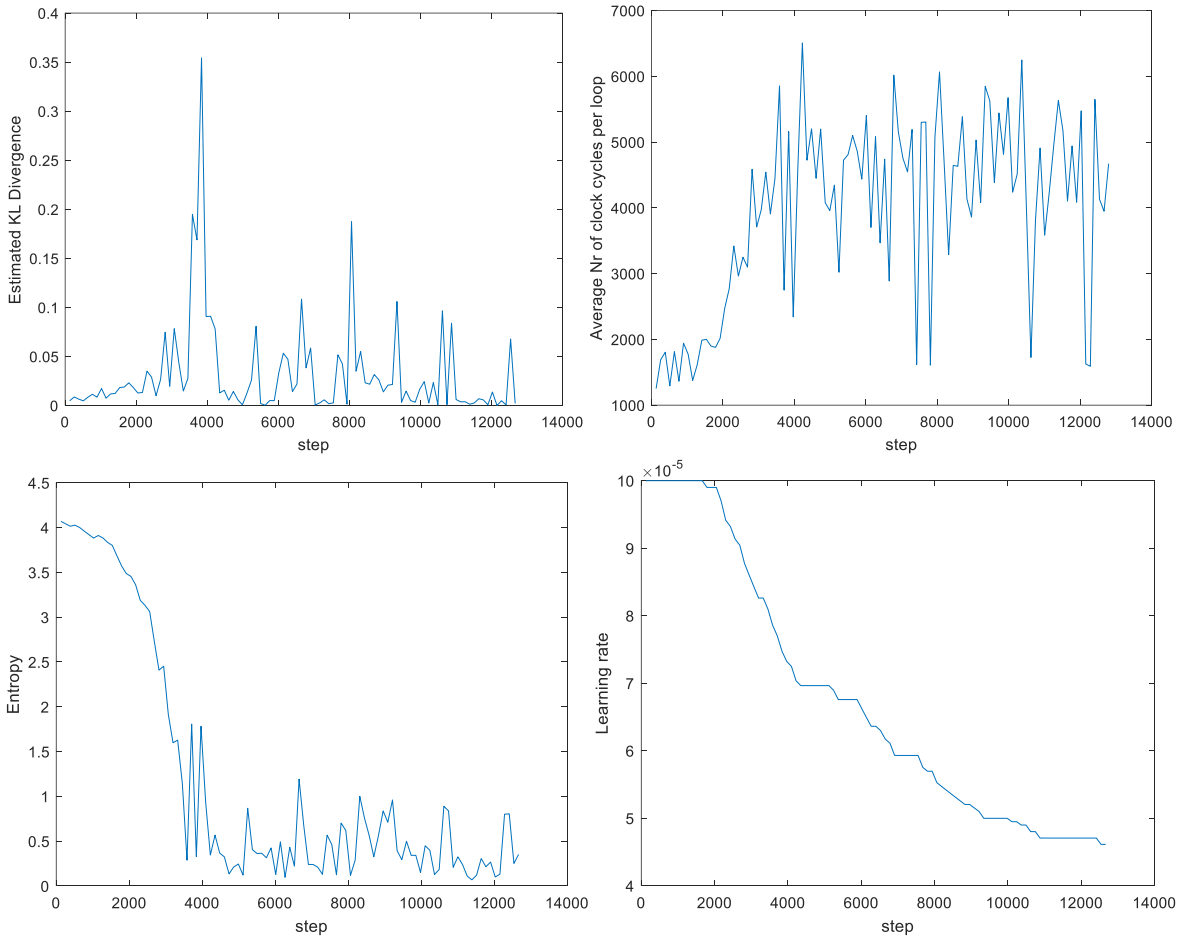


Figure 6-15: Performance of PPO-Discrete-GGNN with Attentive Pooling (from Lab PC)

6.9. Results of PPO-MultiDiscrete-GGNN

Based on experience of previous experiments, we also implement the PPO-MultiDiscrete-GGNN configuration. Note that this configuration has not been tested in the filling grid environment. It uses attentive pooling to extract the graph feature. Table 6-7 shows the hyperparameters.

Table 6-7: Hyperparameters of PPO-MultiDiscrete-GGNN in the Stress Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.001
c_{vf}	0.5
Total time steps	25000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Shared network: Input->GGNN(3 layers)->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Critic: Graph embedding->MLP->$\hat{A}^{\pi}(s,a)$ Actor: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Other networks are initialized in the same way as PPO-Discrete-MLP
std_{critic}^{output}	1
std_{actor}^{output}	0.01
Command Vector Length	4

Figure 6-16 shows the performance of this configuration. It takes 3 hours and 9 minutes on the lab PC to get the results. The c_{loop} curve converges after 13000 steps and 123 minutes. In fact, this is the best performance that we have ever seen. It achieves a c_{loop} around 20000. Moreover, after 17000 steps, the c_{loop} curve only ranges from 19000 and 23000.

In the first 10000 steps, the c_{loop} curve does not significantly increase, and the entropy decreases slowly. In fact, you can find similar pattern at the beginning of training process in Figures 5-12 and 6-15. Perhaps the attentive pooling layer was adapting to the problem in the initial training phase. Another possible reason is that the initial learning rate may be too high, and training becomes more effective when the learning rate decreases.

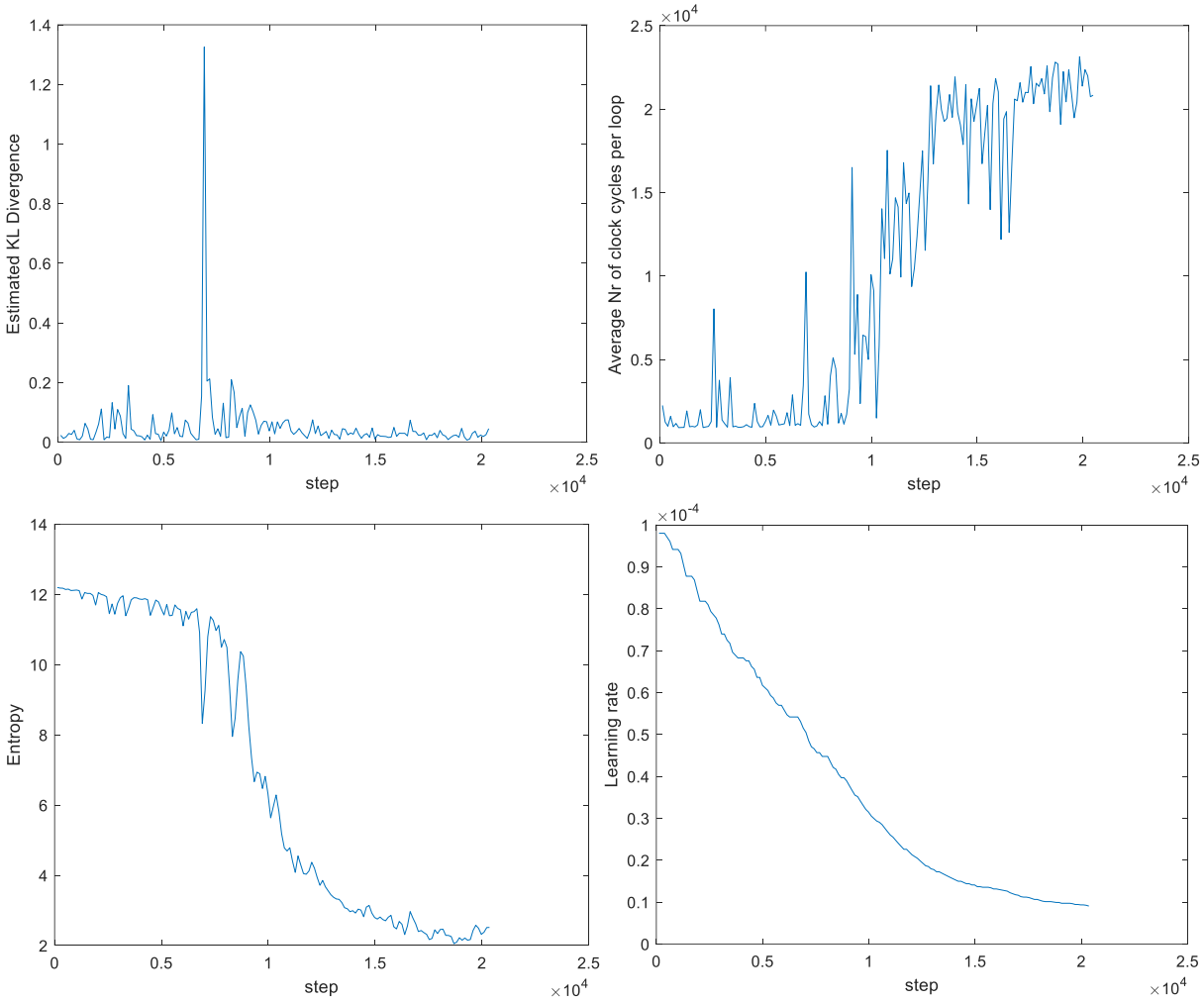


Figure 6-16: Performance of PPO-MultiDiscrete-GGNN (Command Vector Length=4, from Lab PC)

6.10. Bugs Identified in Stress Testing

Although there is no testing oracle in the experiment, we still find an anomaly during the stress testing. When running the algorithms in the MultiDiscrete action space, sometimes the COMMS board gets stuck and cannot reply. This phenomenon may disappear later or continue.

Limited by time, we have not found the root of the anomaly. However, it is possible to find anomalies in a phenomenon like a late response, no response, and unexpected response.

6.11. Discussions

Table 6-8 summarizes the results in stress testing.

Table 6-8: Brief Summary of Results in Stress Testing

Config.	Running Time	Average CPU Load at the end	Range of CPU Load at the end	Note
Random	-	900 (lab PC), 700 (cloud server)	There are spikes (up to 30k) in the c_{loop} curve	Its average CPU load does not increase, so we do not record the running time.

Genetic (Discrete)	4.1 hour to roughly converge, 10.1 hour in total	4500 (lab PC)	The “best” solution reaches 18360. Average c_{loop} ranges from 2500 to 6000.	Performance of the “best” solution is not repeatable. It only achieves c_{loop} between 2000 and 4000.
Genetic (MultiDiscrete)	9 hours to roughly converge, 12.9 hour in total	15000 (lab PC)	The “best” solution reaches 16000. Average c_{loop} ranges from 14500 to 15500.	Performance of the “best” solution is not repeatable. It only achieves c_{loop} between 13200 and 17000.
Q-Learning	-	1400 (lab PC)	There are spikes (up to 2400) in the c_{loop} curve	Its average CPU load does not increase, so we do not record the running time.
D3QN-Discrete-MLP	-	900 (cloud server)	700~1100, but not converge	c_{loop} increases and then decreases. The Q value curve diverges at the end.
PPO-Discrete-MLP	30 minutes to roughly converge, 52 minutes in total	4000 (lab PC)	3500~4500	
PPO-MultiDiscrete-MLP (Cmd Vector Length = 4)	44 minutes to roughly converge, 59 minutes in total	15000 (lab PC)	14000~16000	
PPO-MultiDiscrete-MLP (Cmd Vector length = 11)	79 minutes to roughly converge, 126 minutes in total	1300 (lab PC)	1280~1320	
PPO-Discrete-GGNN (Feature Compression)	33 minutes to roughly converge, 167 minutes in total	3500 (lab PC)	1500~4500	
PPO-Discrete-GGNN (Attentive Pooling)	26 minutes to roughly converge, 111 minutes in total	4500 (lab PC)	2000~6500	
PPO-MultiDiscrete-GGNN (Attentive Pooling, Cmd Vector Length = 4)	123 minutes to roughly converge, 189 minutes in total	20000 (lab PC)	19000~23000	Best performance in stress testing

Because of unpredictable load peaks, stress testing is a non-deterministic environment. Testers will get different average CPU loads if they send a fixed command sequence several times. Therefore, we shall evaluate how high a CPU load an algorithm can achieve and whether it has stable performance.

Random testing and the Q learning algorithm fail in both respects. Their CPU load does not increase during the test and remains unstable. We guess the Q-learning algorithm's human-defined state identification rules are ineffective. On the other hand, it also takes more human labour to design good state identification rules.

We try the genetic algorithm in the discrete and multi-discrete action spaces, i.e., selecting human-defined commands or organizing command vectors. The genetic algorithm does work in both cases

and triggers a higher CPU load in the multi-discrete action space. Nevertheless, we cannot get the same performance when we run the "best" solutions several times. The c_{loop} of the "best solution" in discrete action space ranges between 2000 and 4000. The c_{loop} of the "best solution" in multi-discrete action space ranges between 13000 and 17000.

Unlike the filling grid test results, the DQN algorithm does not perform well in stress testing. Its c_{loop} curve firstly increases and then decreases, while its entropy diverges at the end of the training process. Reducing the learning rate does not improve the situation but worsens it. Adding L2 regularization to the loss function or dropout layers to the neural network may be helpful, but we have not tried these options.

The PPO algorithm is the winner of the stress testing. In the discrete action space, the PPO-Discrete-GGNN configuration with attentive pooling has an average $c_{loop} \approx 4500$, but the curve sometimes drops under 2000. By contrast, the PPO-Discrete-MLP configuration has a lower $c_{loop} \approx 4500$, but its curve in Figure 6-10 looks more stable than the curve in Figure 6-15.

In the MultiDiscrete action space, PPO with GGNN achieves a higher CPU load but more robust oscillation. The PPO-MultiDiscrete-MLP configuration (command vector length is 4) has an average $c_{loop} \approx 15000$, and the value ranges between 14000 and 16000 most of the time after convergence. In contrast, the PPO-MultiDiscrete-GGNN configuration with the same command vector length has a lower $c_{loop} \approx 20000$, and the value ranges between 19000 and 23000.

Thus, the gated graph neural network (GGNN) can process graph input and improve the algorithm's performance. As a price, the performance of the algorithm may be more unstable. Moreover, attentive pooling works better than feature compression in GGNN, but it also makes the agent learn slowly during the initial training phase.

The PPO and genetic algorithm trigger higher CPU load with lower variance in the MultiDiscrete action space. This may be because the human-defined commands are unsuitable for stress testing. However, if the length of the command vector becomes larger (e.g., 11), the search space will be too large for the algorithm. In this case, the PPO algorithm will converge at a low c_{loop}, as shown in Figure 6-13.

All experiments in the chapter, except the D3QN-Discrete-MLP configuration, run on the lab PC. Among these experiments, the genetic algorithm takes the longest, and the PPO algorithm is much quicker. To some extent, this is not in line with our impression that "reinforcement learning takes longer training time". For example, the genetic algorithm in the multi-discrete action space converges after 9 hours. The PPO-MultiDiscrete-MLP configuration achieves similar performance in 44 minutes, and the PPO-MultiDiscrete-configuration converges at a higher CPU load after 123 minutes.

The most crucial finding in this section is the excellent performance of the PPO-MultiDiscrete-GGNN configuration. Suppose the command vector is not too long and attentive pooling is used. In that case, the configuration can achieve the highest CPU load within a limited time and maintain performance in a non-deterministic environment. Since the configuration organizes the commands and does not need to parse the reply, it also needs much less prior knowledge. These are precisely the properties we want to achieve in the research.

7 Coverage Testing

This section introduces the coverage testing and its results:

- Section 7.1 explains the design of the experiment.
- Section 7.2~7.9 shows the results of different configurations, including the random baseline, the genetic baseline, Q-Learning, D3QN-Discrete-MLP, D3QN-Discrete-GGNN, PPO-Discrete-MLP, PPO-MultiDiscrete-MLP, PPO-Discrete-GGNN.
- Section 7.10 introduces some bugs of the onboard software identified during the coverage testing.
- Section 7.11 compares results of these configuration and gives a brief summary.

7.1. About the Experiment

We try to maximize the code coverage of the COMMS flight software of Delfi-PQ in this experiment. When performing the test, there are 630 probes in the source code and 58 commands specified by the tester.

When the agent executes a probe that is never triggered before in an episode, the agent will receive a positive reward of +1. On the other hand, if the agent does not trigger any new probes after sending a command, it will receive a negative reward of -1. Section 4.2.2 shows the formulations of states and actions.

To collect the code coverage (status of probes), the agent will send a code coverage collection command after a testing command. In the previous chapter, the genetic algorithm sends a coverage collection command after every testing command. This is because we want to be the same for all algorithms running on the same machine. By contrast, in coverage testing, the genetic algorithm only needs to collect the code coverage at the end of an episode, which reduces the testing time.

Like the previous chapter, we send the same command 1000 times to verify whether the environment is deterministic:

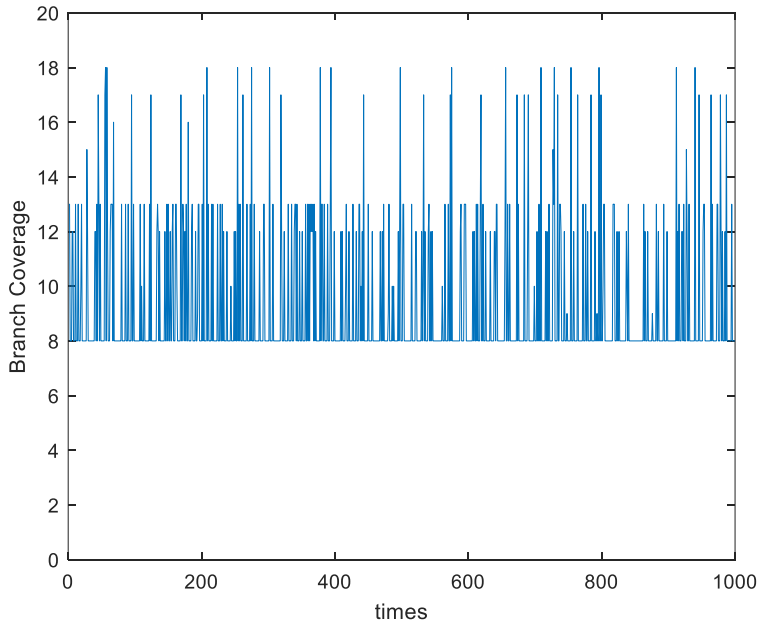


Figure 7-1: Branch Coverage of Send the Same Command for 1000 Times (from Lab PC)

Figure 7-1 shows that the coverage testing is also non-deterministic. In section 7.3, we will evaluate such uncertainty's impact by repeating the genetic algorithm's optimal command sequence.

7.2. Random Baseline

In the random baseline, the agent randomly selects a command from the command list and sends it. Table 7-1 shows the average branch coverage achieved by the baseline in 10 runs.

Table 7-1 Average Branch Coverage of Random Baseline in 10 Runs

Episode Length	Average Coverage	Max Coverage	Min Coverage	Time for 10 runs
128	224.6/630	236/630	210/630	395s
256	240/630	263/630	225/630	792s
512	241.8/630	267/630	229/630	1566s
1024	251/630	267/630	225/630	3150s

The coverage results of the random testing are also “random”. If the episode length is 1024, it may get a branch coverage from 225/630 to 267/630. A simple random policy does not always get high coverage.

At the same time, increasing the episode length can improve the average coverage. If testers can run random testing for more than 30 minutes, they will probably get the maximum branch coverage of the COMMS software because the COMMS software is relatively simple.

7.3. Genetic Algorithm Baseline

Table 7-2 and Figure 7-2 show the hyperparameters and coverage curve of the genetic algorithm, respectively. In this section, we only run the genetic algorithm in the discrete action space, i.e., selecting human-defined commands. Each element in the chromosome is an index of a pre-defined command in a list.

Table 7-2: Hyperparameters of the Genetic Algorithm in Coverage Testing for COMMS

Total iteration number	100
Length of a solution	128
Population	10
Crossover probability	0.4
Mutation probability	0.01

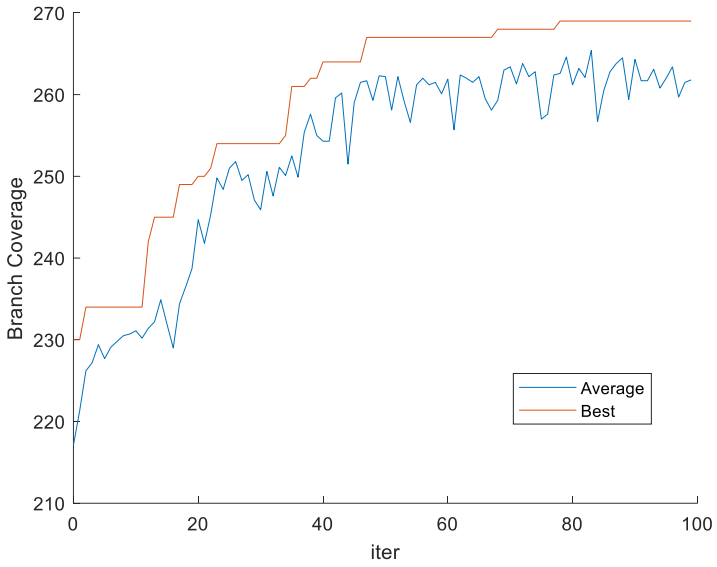


Figure 7-2: Coverage Curve of the Genetic Algorithm (from Lab PC)

In the end, the genetic algorithm reaches a branch coverage of 269/630, slightly higher than the best coverage in the random baseline. It takes 72.6 minutes for the algorithm to complete 100 iterations on the lab PC. The best solution is found after 80 iterations and 57 minutes. To speed up the algorithm, we only send the coverage collection command after all commands in a chromosome have been sent.

Like section 6.3, now we repeat the best chromosome ten times and get the following result:

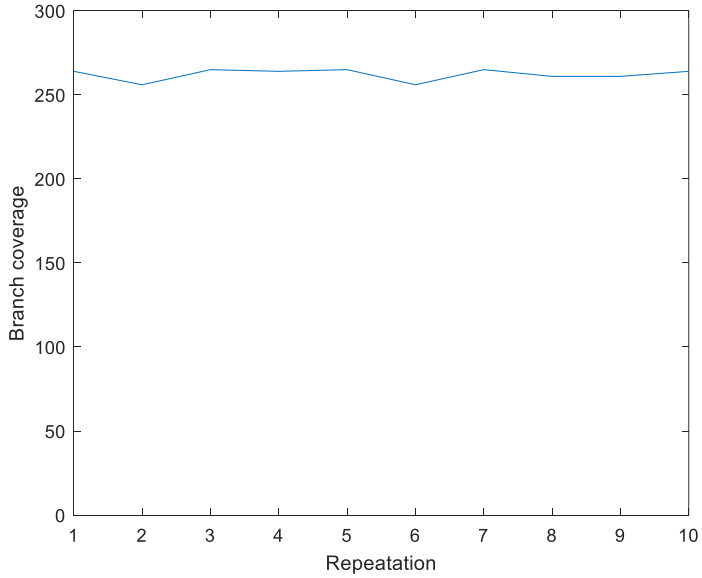


Figure 7-3: Repeat the "Best Command Sequence (Discrete)" for 10 Times

We can compare it with Figure 6-6 in the stress testing. Although the branch coverages differ, their variance is much smaller than the CPU load in Figure 6-6. On the other hand, the branch coverage triggered by a single command varies between 8 and 18 in Figure 7-1, but the coverage triggered by a command sequence only varies between 256 and 266 in Figure 7-3. That is to say, though the coverage of a single command is uncertain, the coverage of a command sequence is roughly the same. Since we use the final coverage of an episode as the performance metric, coverage testing seems to be more "deterministic" than stress testing.

7.4. Results of Q-Learning

Table 7-3 shows the hyperparameters in the tabular Q-Learning algorithm in this test, and the results are shown in Figure 7-4.

Table 7-3: Hyperparameters of the Q-Learning in Coverage Testing for COMMS

Hyperparameter	Value
γ	0.9
Learning rate	0.01
Total time steps	45000
Episode length	128
ε	0.9

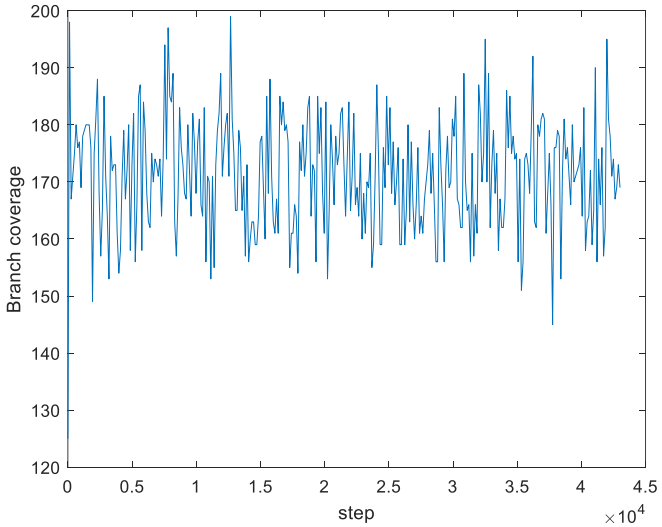


Figure 7-4: Coverage Curve of the Q-Learning Algorithm (from Lab PC)

Given the hyperparameters and state identification rules, the Q-Learning algorithm found 22 states during the test. However, the code coverage does not increase in the training process. The code coverage of Q-Learning (~170/630) is even worse than random testing, which has an average branch coverage of 224.6/630 with the same episode length.

Like stress testing, a possible reason is that the human-specified state identification rules significantly affect the performance of the Q-Learning algorithm.

7.5. Results of D3QN-Discrete-MLP

Now we test the COMMS software with the D3QN-Discrete-MLP configuration. Its hyperparameters are shown in Table 7-4.

Table 7-4: Hyperparameters of D3QN-Discrete-MLP in the Coverage Testing for COMMS

Hyperparameter	Value
γ	0.9
Buffer size	200000
Mini-batch size	128
Learning rate	1e-4
Learning starts	128
Training frequency	1
Total time steps	200000
ε_{start}	1

ε_{end}	0.01
Exploration fraction	0.6
Epoch length	128
N_{target}	500
Neural Network Design	Value network: Input->MLP->ReLU->MLP->ReLU->MLP->value Advantage network: Input->MLP->ReLU->MLP->ReLU->MLP->advantage Every hidden layer has 2048 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$.

Figure 7-5 shows the results of the configuration. The lab PC takes 9 hours and 51 minutes to get the results—the average Q value estimated by the network increases and decreases to a level. The average branch coverage increases from 225/630 to 260/630. However, the final branch coverage is not very stable with $\varepsilon_{end} = 0.01$. Sometimes it even drops under 220/630. We have seen such instability in section 6.5, which may be an inherent property of value-based RL algorithms.

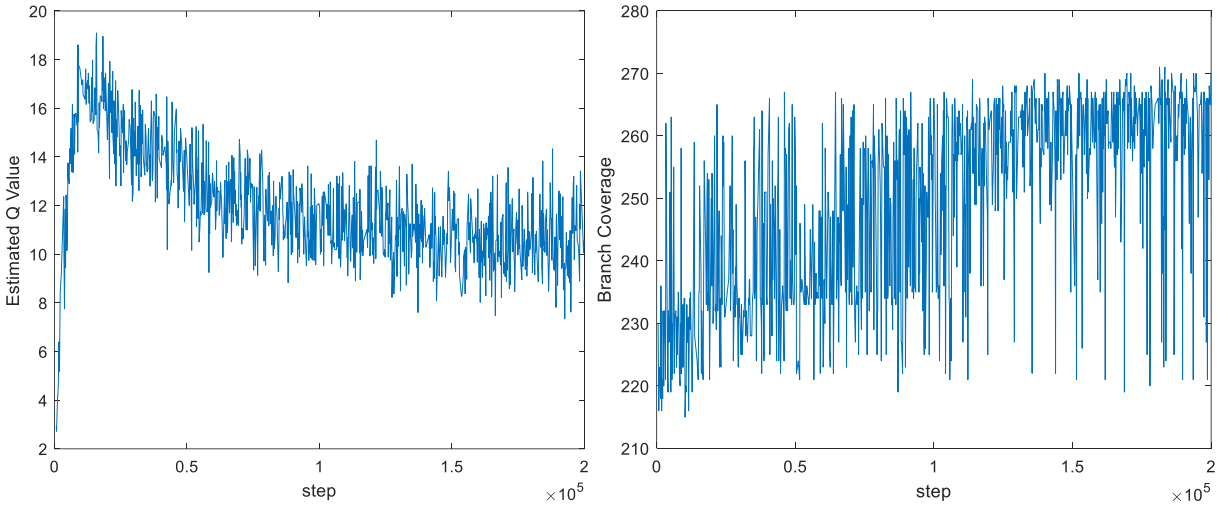


Figure 7-5: Results of the D3QN-Discrete-MLP in the Coverage Testing for COMMS (from Lab PC)

7.6. Results of D3QN-Discrete-GGNN

This section uses the D3QN-Discrete-GGNN configuration instead. We only use the feature compression approach to get a graph representation from node features. It is because we performed this experiment early in the research when we had not found attentive pooling as an effective way to extract graph embedding.

On the other hand, we also had not set up the cloud server then. To reduce the training time on the lab PC, we reduced the length of the node feature vectors from 127 to 7. Word2Vec generated these short node feature vectors from the source code of COMMS software.

The hyperparameters of this configuration are in Table 7-5, and the results are in Figure 7-6.

Table 7-5: Hyperparameters of D3QN-Discrete-GGNN in the Coverage Testing for COMMS

Hyperparameter	Value
γ	0.9

Buffer size	500000
Mini-batch size	128
Learning rate	2.5e-4
Learning starts	128
Training frequency	10
Total time steps	500000
ε_{start}	1
ε_{end}	0.01
Exploration fraction	0.9
Epoch length	36
N_{target}	500
Neural Network Design	Shared network: Input->GGNN(3 layers)->MLP->ReLU->Concatenation of node vectors->ReLU->Graph embedding Compute value: Graph embedding->MLP->value Compute advantage: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Initial std of output layers is 0.01. Other networks are initialized in the same way as D3QN-Discrete-MLP

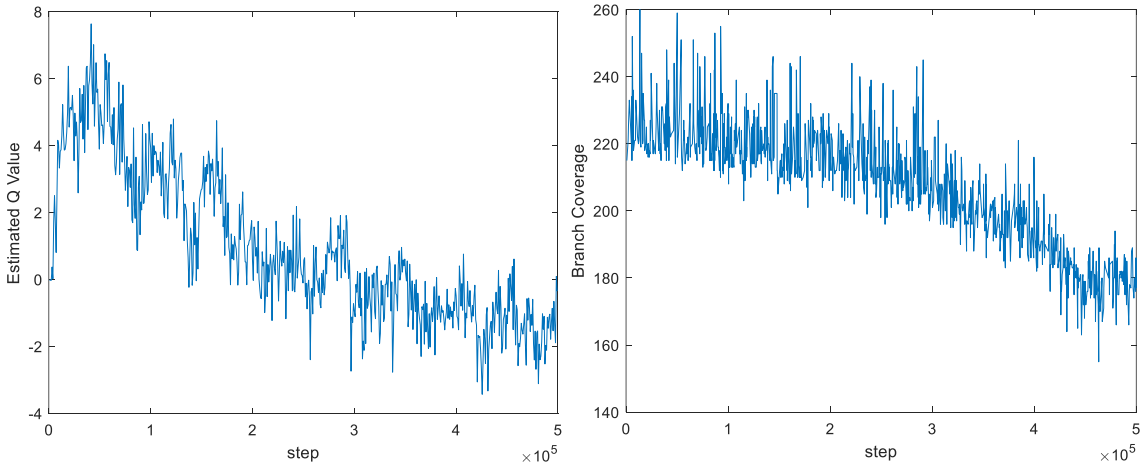


Figure 7-6: Results of the D3QN-Discrete-GGNN in the Coverage Testing for COMMS (from Lab PC)

It takes 26 hours and 1 minute to get the curves in Figure 7-6. Unfortunately, the branch coverage per episode does not increase even after we tried several sets of hyperparameters. It even drops during the training process. It may be caused by the configuration itself, unsuitable hyperparameters, or the short node feature vectors.

7.7. Results of PPO-Discrete-MLP

In this section, we use the PPO algorithm instead. Table 7-6 shows the hyperparameters of the PPO-Discrete-MLP configuration in this test. Figure 7-7 shows the performance of the configuration.

Table 7-6: Hyperparameters of PPO-Discrete-MLP in the Coverage Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32

Initial learning rate	1e-4
γ	0.9
c_{ent}	0.01
c_{vf}	0.5
Total time steps	200000
Epoch length	128
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 2048 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01

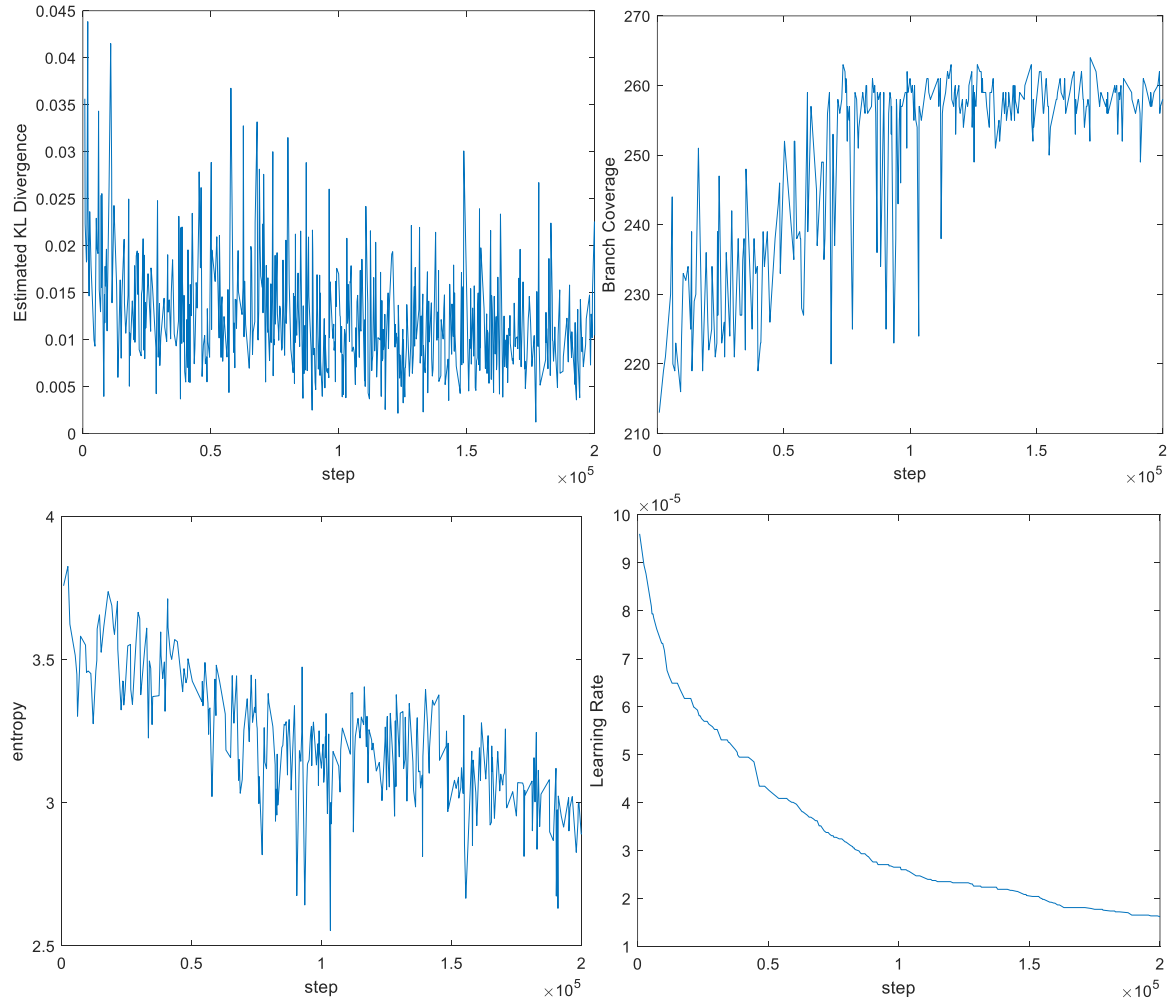


Figure 7-7: Results of the PPO-Discrete-MLP in the Coverage Testing for COMMS (from Lab PC)

It takes 6 hours and 12 minutes to get the curves in Figure 7-7. The branch coverage curve converges after roughly 100000 steps and 186 minutes. Like the D3QN-Discrete-MLP configuration, the branch coverage of this configuration also increases from 220/630 to around 260/630. The final code coverage is more stable than the curve of the D3QN algorithm. On the other hand, the KL divergence is under control, and the entropy drops from 3.75 to 2.9. These results show similar patterns to the results in the filling grid test.

7.8. Results of PPO-MultiDiscrete-MLP

Unlike previous configurations in this section, PPO-MultiDiscrete-MLP does not use the 58 commands specified by the testers. It organizes the commands by itself (section 4.3.3). The length of a command vector is 4, in which the first element is the service number, and the following three elements are the command payload. Table 7-7 and Figure 7-8 show the hyperparameter and performance of the configuration, respectively.

Table 7-7: Hyperparameters of PPO-MultiDiscrete-MLP in the Coverage Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02
Mini-batch size	32
Initial learning rate	5e-4
γ	0.9
c_{ent}	0.01
c_{vf}	0.5
Total time steps	200000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Critic network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{V}^{\pi}(s)$ Actor network: Input->MLP->ReLU->MLP->ReLU->MLP->$\hat{A}^{\pi}(s,a)$ Every hidden layer has 2048 neurons.
Network Initialization	Orthogonal initialization of weights. Initial biases are set to 0. Initial std of hidden layers is $\sqrt{2}$
std_{critic}^{output}	1
std_{actor}^{output}	0.01
Command Vector Length	4

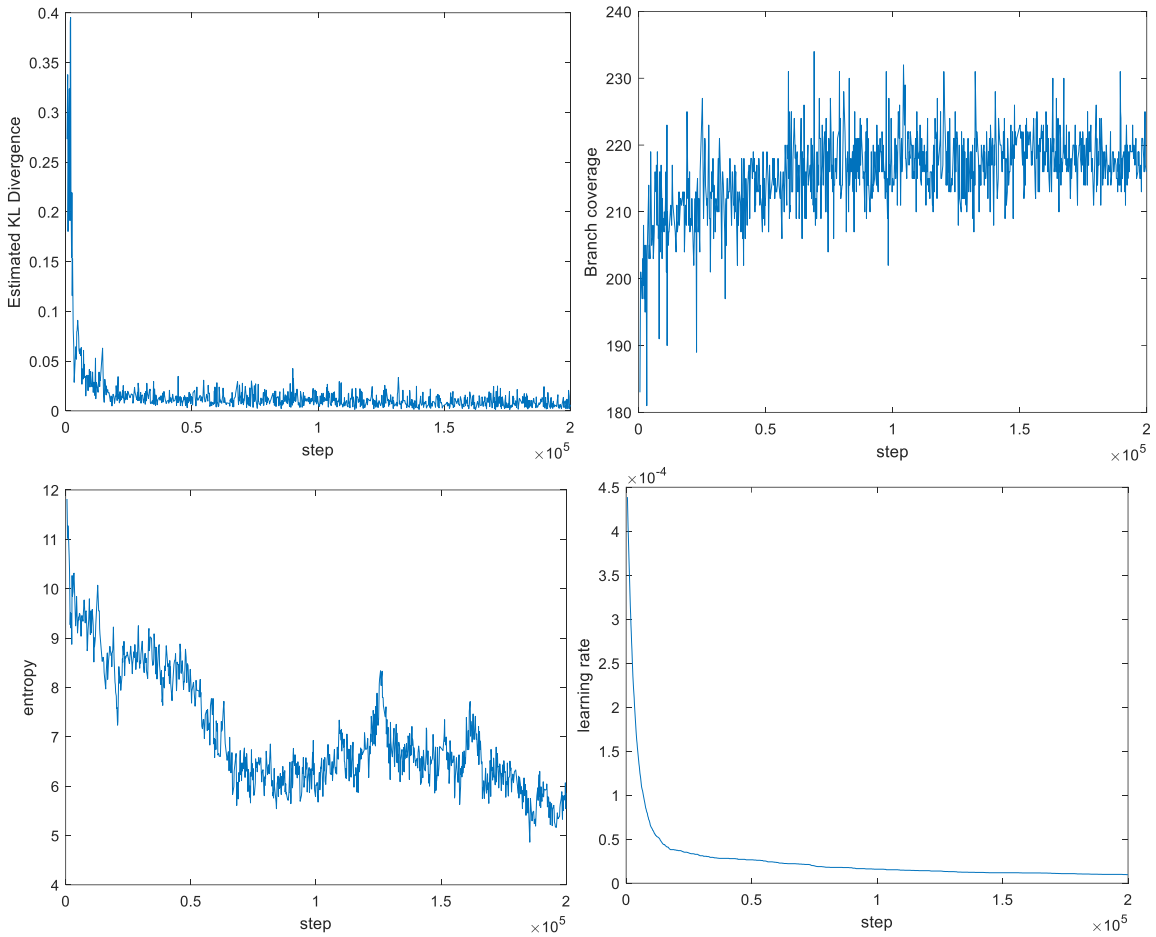


Figure 7-8: Results of the PPO-MultiDiscrete-MLP in the Coverage Testing for COMMS (from Lab PC)

It takes 7 hours and 12 minutes to get the results in Figure 7-8. Although the branch coverage does increase and the entropy does drop, its coverage is lower than the PPO-Discrete-MLP configuration.

The performance here is very different from the performance of the same configuration in stress testing. As mentioned in section 6.7, the PPO algorithm in the MultiDiscrete action space is not good at looking for valid long command vectors. Therefore, we set the command vector length as 4. Short command vectors trigger high CPU load but do not trigger good code coverage in our experiments.

Another problem is that the agent may try some “dangerous” commands to damage the board. Stress testing is acceptable because we want to “destroy” the system. However, it may not be suitable for other testing scenarios.

7.9. Results of PPO-Discrete-GGNN

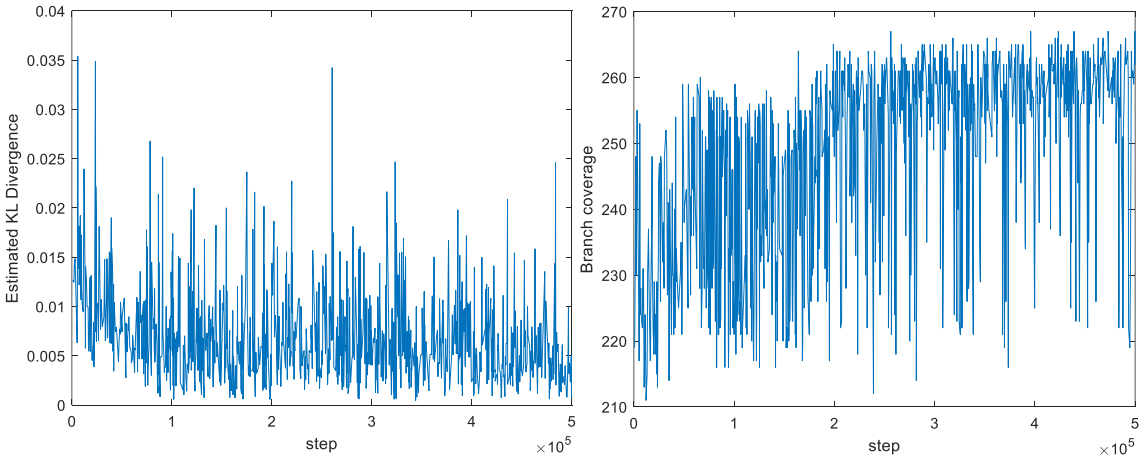
We try both the feature compression and attentive pooling approach for this configuration. Table 7-8 shows the hyperparameters. In the feature compression configuration, the length of node feature vectors is only seven because the experiment was performed earlier (just like in section 7.6). By contrast, in the attentive pooling configuration, the node feature vector length is 127.

Table 7-8: Hyperparameters of PPO-Discrete-GGNN in the Coverage Testing for COMMS

Hyperparameter	Value
KL_{target}	0.02 (feature compression) / 0.05 (attentive pooling)

Mini-batch size	32
Initial learning rate	1e-4
γ	0.9
c_{ent}	0.01 (feature compression) / 0.001 (attentive pooling)
c_{vf}	0.5
Total time steps	500000
Epoch length	36
Environment steps	128
Update steps	4
λ	0.95
ξ	0.2
Maximal norm of gradient	0.5
Neural Network Design	Shared network: Input->GGNN(3 layers)->MLP->ReLU->Attentive pooling/concatenation of node vectors->ReLU->Graph embedding Critic: Graph embedding->MLP->$\hat{A}^{\pi}(s,a)$ Actor: Graph embedding->MLP->advantage
Network Initialization	GGNN initialization is mentioned in section 4.2.4. Other networks are initialized in the same way as PPO-Discrete-MLP
std_{critic}^{output}	1
std_{actor}^{output}	0.01

Figure 7-9 shows results of the feature compression approach. It takes 21 hours and 7 minutes on the lab PC to get the results. The average branch coverage does converge around 255, but the curve fluctuates strongly.



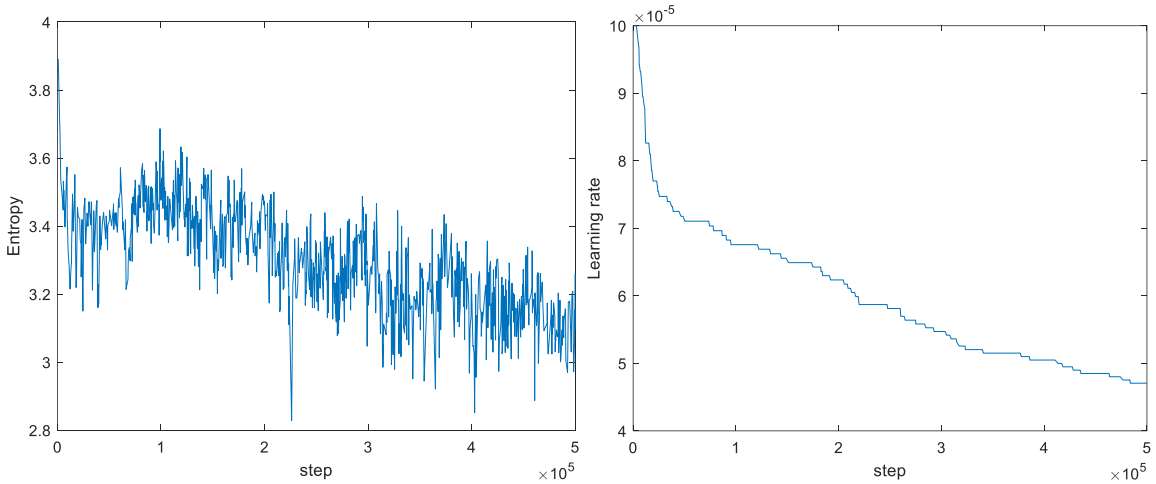


Figure 7-9: Results of the PPO-Discrete-GGNN with Feature Compression (from Lab PC)

Figure 7-10 shows results of the attentive pooling approach. It takes 18 hours and 21 minutes on the cloud server to get the results. Compared with the feature compression method, its average coverage is only 245/630. The curve also fluctuates strongly.

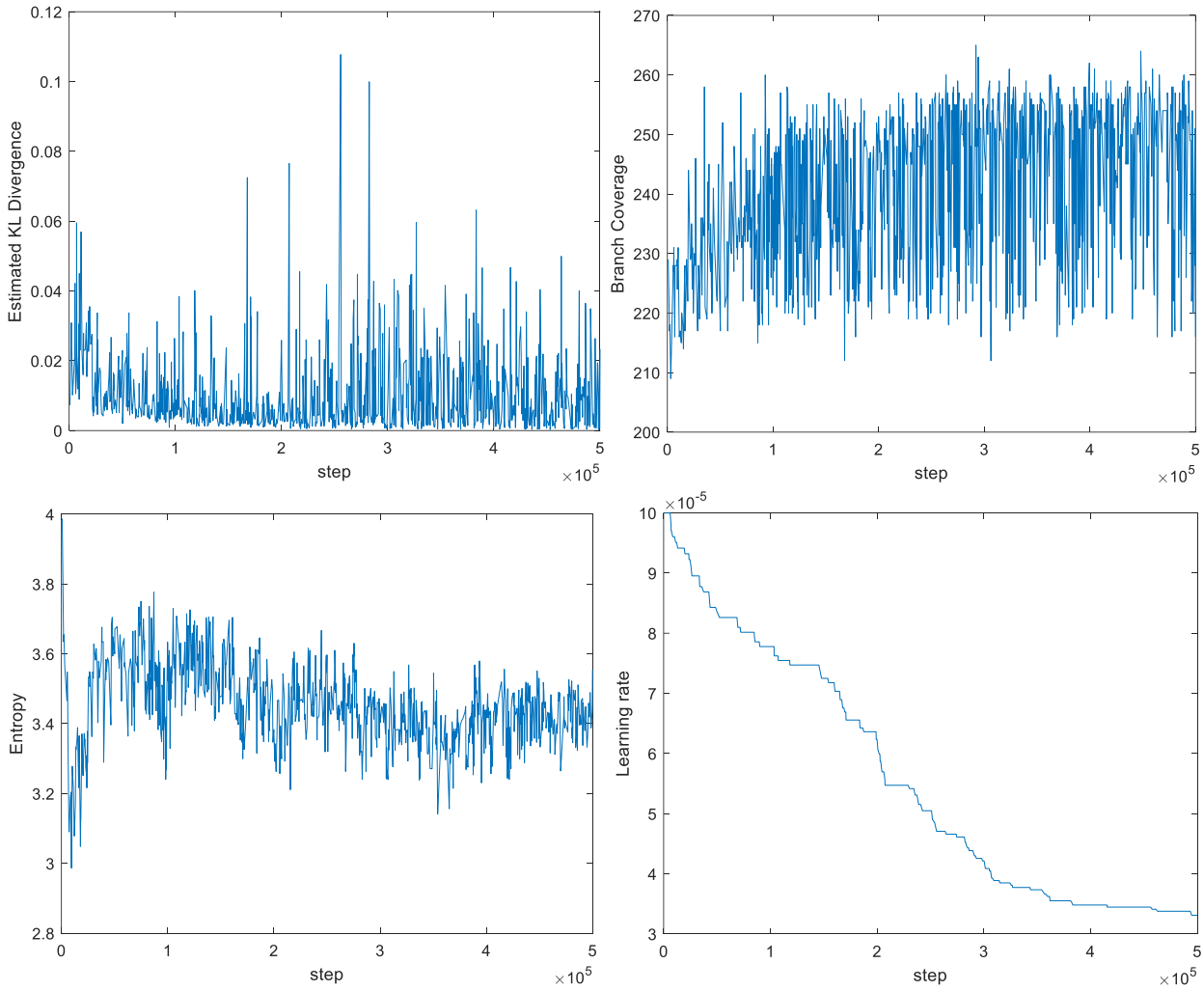


Figure 7-10: Results of the PPO-Discrete-GGNN with Attentive Pooling (from Server)

7.10. Bugs Found in the Coverage Testing

We found several bugs in the COMMS flight software during the coverage testing:

- The microcontroller sometimes gets stuck in an indefinite loop in a low-level library provided by Texas Instruments. It is because we do not set the UART module correctly.
- The “get metadata” command sometimes does not get a proper response because of a mistake in the software update service.
- The “erase slot” command does not have the correct response because of another mistake in the software update service.

The first bug was identified when no response was received. The following two bugs were identified because the parser could not parse the replies.

Finding more bugs in this chapter does not mean that coverage testing is more powerful than stress testing. We started coverage testing earlier, so it was more likely to detect some anomalies.

7.11. Discussions

Table 7-9 summarizes the results in coverage testing. Note that the total number of probes is 630.

Table 7-9: Brief Summary of Results in Coverage Testing

Config.	Running Time	Average Branch Coverage at the End	Range of Branch Coverage at the end	Note
Random	26 min per run (when episode length is 512)	251 (10 runs)	267 (highest coverage in the 10 runs)	
Genetic (Discrete)	57 minutes to roughly converge, 73 minutes in total	260	The “best” solution reaches 269. Average coverage ranges from 255 to 265.	Repeating the “best” solution gets coverage from 256 to 266.
Q-Learning	-	170	145~198	Its average coverage does not increase, so we do not record the running time.
D3QN-Discrete-MLP	384 minutes to roughly converge, 591 minutes in total	260	220~270	The coverage curve strongly oscillates.
D3QN-Discrete-GGNN	1561 minutes in total. Not converge.	180	-	The Q value curve first increases and then decreases.
PPO-Discrete-MLP	186 minutes to roughly converge, 372 minutes in total	260	255~265	
PPO-MultiDiscrete-MLP (Cmd Vector Length = 4)	206 minutes to roughly converge, 412 minutes in total	220	210~230	
PPO-Discrete-GGNN	633 minutes to roughly	255	220~265	The coverage curve strongly oscillates.

(Feature Compression)	converge, 1267 minutes in total			
PPO-Discrete-GGNN (Attentive Pooling)	660 minutes to roughly converge, 1101 minutes in total	245	210~265	The coverage curve strongly oscillates.

The coverage testing environment is also non-deterministic. If testers send the same command sequence several times, they will not always get the same coverage result. However, compared with the stress testing environment, it has less uncertainty because the coverage results have smaller variance than the average CPU load.

Random testing seems to be the best option to maximize code coverage of the COMMS onboard software. It is straightforward. Given enough time (e.g., 26 minutes), random testing may reach a good branch coverage of 267/630.

The genetic algorithm here performs much better than in the previous chapter. It finds a good solution that can reach branch coverage between 256/630 and 266/630 within 1 hour. There are two reasons. Firstly, as mentioned above, coverage testing has less uncertainty, making coverage of the best chromosome more repeatable. Secondly, we only collect code coverage after sending the command sequence, significantly reducing communication time[26].

The tabular Q-Learning algorithm still does not work. If the state identification rules are ineffective, the algorithm will not work well in coverage and stress testing.

The D3QN-Discrete-MLP configuration converges in coverage testing. Its average branch coverage increases from 225/630 to 260/630. However, the coverage of the configuration is unstable. It usually drops under 220/630. On the other hand, the D3QN-Discrete-GGNN configuration does not converge in this chapter. Its Q value curve first increases and then crashes.

The PPO-Discrete-MLP configuration converges in 186 minutes and achieves a stable coverage of around 260/630. Such performance is acceptable but takes more time than the genetic algorithm. At the same time, PPO-MultiDiscrete-MLP does not perform well in coverage testing. When the command vector length is 4, the configuration converges around 220/630. Short commands cannot trigger high code coverage in the COMMS onboard software.

We try PPO-Discrete-GGNN with both feature compression and attentive pooling. However, the coverage curve of both methods strongly fluctuates.

[26] There are some considerations for collecting code coverage frequently in stress testing. See section 6.1.

8 Regression Testing

This section introduces the regression testing and its results:

- Section 8.1 explains the design of the experiment.
- Section 8.2 shows the results of the genetic algorithm baseline.
- Section 8.3 shows the results of the PPO-MultiDiscrete-GGNN configuration.
- Section 8.4 compare the PPO-MultiDiscrete-GGNN configuration with the baseline.

8.1. About the Experiment

We perform regression testing in this section. That is to say, we train an RL agent with a version of onboard software and then use the agent to test another version of onboard software. More specifically, the PPO-MultiDiscrete-GGNN agent trained in section 6.9 performs stress testing on another version of COMMS software. As mentioned in section 3.4.2, the configuration can process graph input with different node numbers. Thus, it may work in regression testing when the number of probes (CodeCount) changes.

Table 8-1 shows the two versions used in the experiment. Note that the SUT under test has 21.7% fewer probes in the source code.

Table 8-1: 2 Versions of Onboard Software Used in Regression Testing

SUT to Train the Agent		**SUT to be Tested by the Agent**		**Differences**
Name/Hash	**Edit Time**	**Name / Hash**	**Edit Time**	
COMMS/32f35c5	2020-5-11	COMMS/dc3a952	2020-1-30	1523 insertions, 2659 deletions
DelfiPQcore/9e15f6c	2020-7-23	DelfiPQcore/7155318	2020-1-30	982 insertions, 2379 deletions
DSPI/43d195c	2020-9-7	DSPI/8757a37	2019-12-20	12 insertions, 67 deletions
DWire/8acd7b3	2020-7-7	DWire/bf514fb	2020-1-1	214 insertions, 10 deletions
INA226/a6ce237	2020-9-11	INA226/e6ef01c	2019-11-29	54 insertions, 72 deletions
MB85RS/2711fac	2020-9-8	MB85RS/1b6e934	2019-12-20	16 insertions, 73 deletions
PQ9Bus/680f461	2020-8-15	PQ9Bus/53de707	2020-1-15	59 insertions, 122 deletions
SX1276/6ea873e	2020-10-2	SX1276/36d9e38	2019-12-11	68 insertions, 98 deletions
TMP100/cf62dae	2020-9-12	TMP100/0b47229	2019-11-24	61 insertions, 69 deletions
Total Number of Probes: 630		**Total Number of Probes: 493**		**21.7%**

Figure 8-1 is the graph representation extracted from the version under test. It has a different structure compared with Figure 3-19.

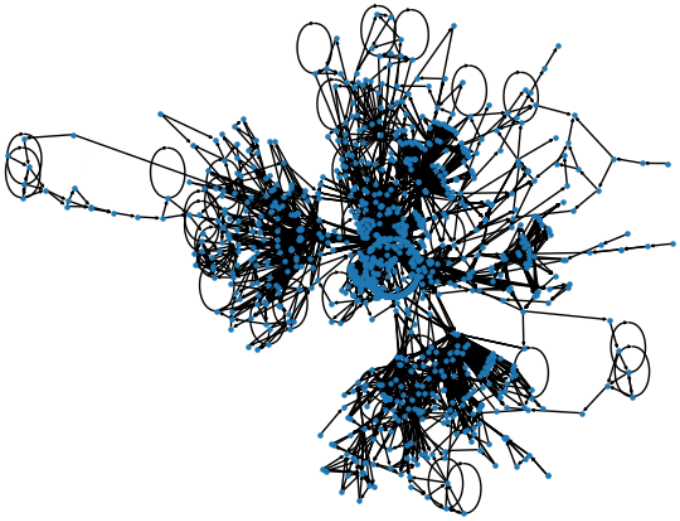


Figure 8-1: Graph Extracted from the COMMS Software under the Regression Test

The genetic algorithm in the MultiDiscrete action space is the baseline of this chapter. We set the best chromosome in section 6.3 as one of the initial chromosomes of the algorithm, so the algorithm has "experience" from the previous test.

8.2. Genetic Algorithm with Best Solution from Previous Test

This section uses the same hyperparameters as section 6.3. Figure 8-2 shows the CPU load triggered by the genetic algorithm.

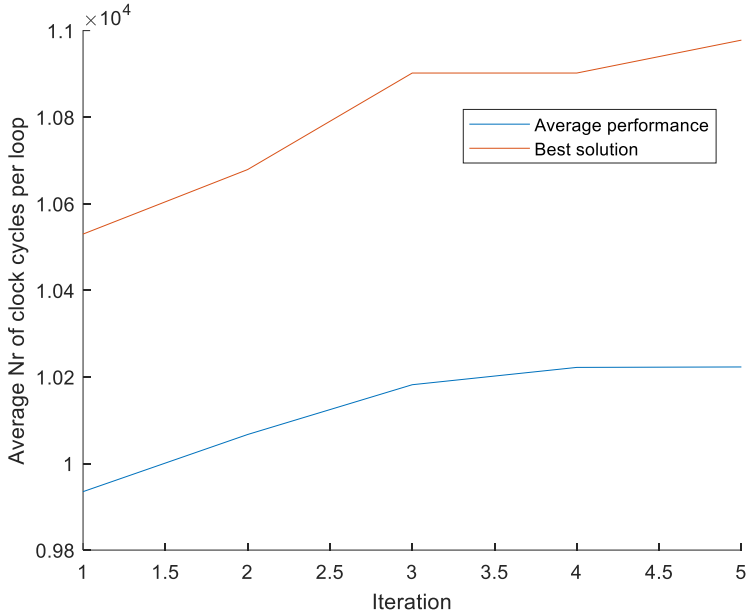


Figure 8-2: Results of the Genetic Algorithm (MultiDiscrete, from Lab PC)

We only run the algorithm for 5 iterations, which take 44 minutes on the lab PC. The average c_{loop} at the 5th iteration is 10223, and the best solution reaches 10978.

8.3. Results of PPO-MultiDiscrete-GGNN

This section uses the same hyperparameters as section 6.9. Attentive pooling is used in the GGNN network. We compare 2 cases here:

- In the first case, we train a new agent from scratch.
- In the second case, we use the agent trained in section 6.9.

Figure 8-3 shows the results. For both cases, it takes about 45 minutes to get the results on the lab PC. The trained agent maintains a $c_{loop} \approx 15000$ from the beginning, while c_{loop} of the new agent increases from 8000 and finally reaches 15000 after 2000 steps.

The trained agent also keeps a low entropy around 2.5, which is close to the final entropy in section 6.9. Although the new agent achieves similar c_{loop} after 5000 steps, it has a much higher entropy around 7. The reason of such phenomenon is not clear yet.

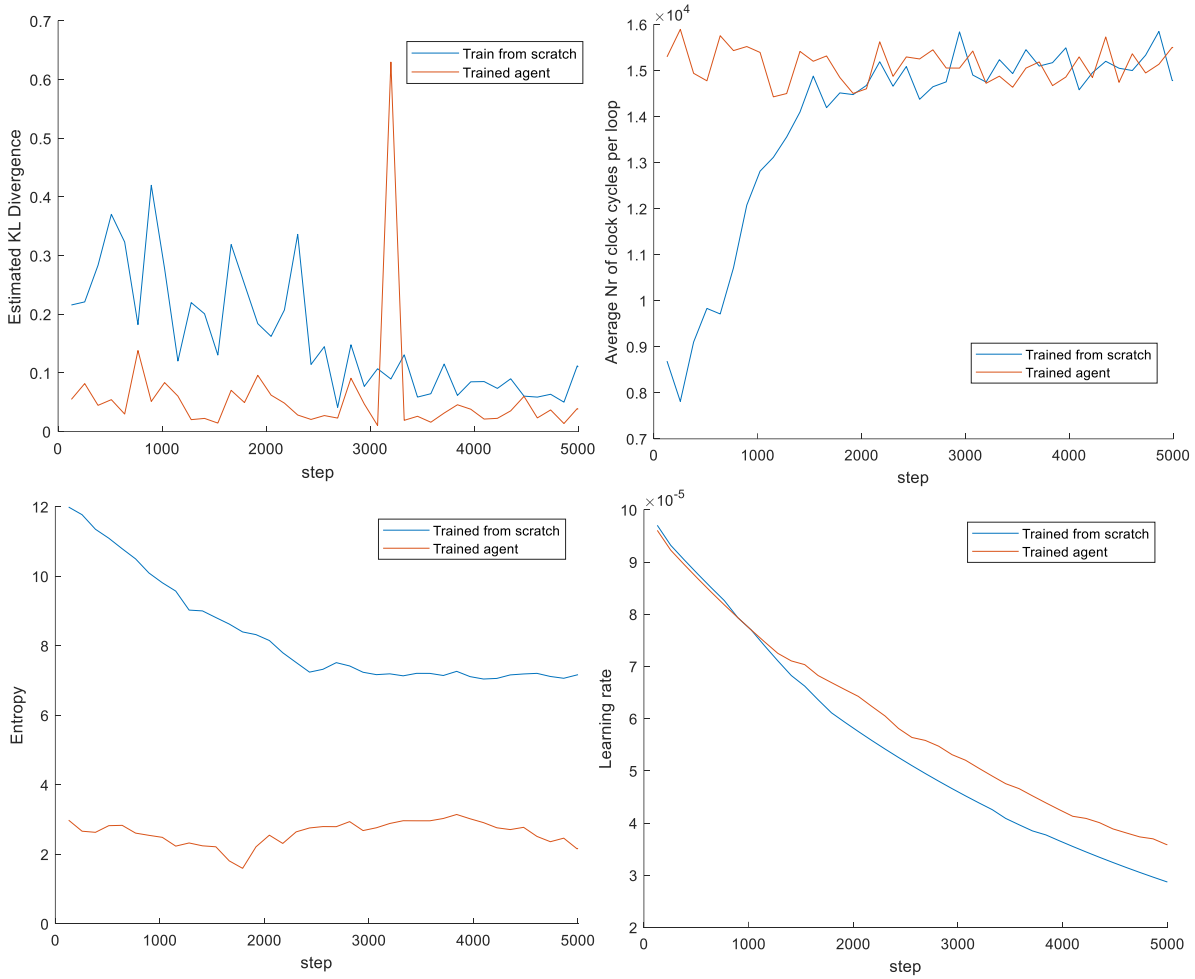


Figure 8-3: Results of the PPO-MultiDiscrete-GGNN (attentive pooling, from Lab PC)

8.4. Discussions

The results of regression testing are promising. As shown in Figure 8-3, an RL agent can learn some common knowledge from a version of SUT and then perform well on another version of SUT.

Note that the two versions have many differences. As shown in section 8.1, the SUT under test has 21.7% fewer probes.

The agent also outperforms the genetic algorithm. Compared with reusing the best solution, reusing a trained policy has better effects on stress testing.

Using RL in regression testing seems to be a promising direction for research. When two versions of the software under test become too different, the generalization of RL can be a problem. The trained agent may be overfitted to the previous version and cannot adapt to the next version of SUT. Some techniques to solve the generalization problem include multi-task learning, meta-learning, and causal reasoning.

9 Conclusions

This work starts with the testing problem of the Delfi-PQ satellite. It first surveys software testing, onboard software design, and sequential decision-making algorithms (e.g., reinforcement learning). A testing environment, a code coverage collection tool, and a graph extraction tool are built. We also implement reinforcement learning algorithms with different neural network structures. We compare them with two baselines in 4 types of testing scenarios (filling grid, stress, coverage, and regression).

This chapter summarizes the study. It includes four parts:

- Answers to the research questions listed in section 2.4.2.
- Threats to validity. Assumptions in section 2.4.1 will be considered.
- Contributions to the academic field.
- Recommendations for future research.

9.1. Answers to the Research Questions

9.1.1 Testing Goals

This subsection gives answers to the following research question in section 2.4.2:

RQ-1 What's the goal of testing command generation?

Chapter 2 lists several possible testing goals, including:

- State/action coverage[27]
- Code coverage
- CPU load
- Number of failures.

Before maximising state coverage, testers must define what a state is. This step may be simple for the SUTs with graphic user interfaces. Nevertheless, for the Delfi-PQ onboard software, states are defined by some human-defined "state identification rules" (section 3.3.2). We use the rules to extract state information from commands and responses. Writing these rules needs prior knowledge and significant human labour.

We try to apply these rules in the Q-Learning algorithm. Unfortunately, the Q-Learning algorithm fails to converge in stress testing (section 6.4) and coverage testing (section 7.4). One possible reason is that these rules are not effective enough.

Maximising code coverage is an achievable goal. For some embedded systems, retrieving code coverage from target microcontrollers is challenging. Commercial tools exist, but they can be expensive and not work in some scenarios.

To tackle the challenge, a branch coverage collection tool is implemented in this research (section 3.4.1). The tool can instrument the source code automatically, collect branch coverage with a particular command, and generate easy-to-read coverage reports. It is simple (153 lines in Python)

[27] Our state identification rules include information like "whether a command has been sent". Therefore, state coverage is a superset of action coverage in this work.

and easy to adapt to different programming languages/target microcontrollers. It also has a small memory footprint (1 bit per probe) & short collection time (~60ms). To some extent, the tool is the cornerstone of this study.

The coverage testing environment is non-deterministic, but the uncertainty is smaller than the stress testing. That is to say, one will get different coverage results when sending the same command sequence several times, but the variance of the results is relatively low (sections 7.1 & 7.3).

In chapter 7, the D3QN and the PPO algorithms can learn to trigger higher code coverage. However, this does not pay off because the two baselines perform better. The random baseline, i.e., randomly selecting a command to send, can reach a good code coverage on the COMMS onboard software if enough time (e.g., 26 minutes) is given. On the other hand, the genetic algorithm baseline can also find a good command sequence in less than 1 hour. The output command sequence can usually achieve good coverage.

We also try to maximise CPU load in this study. The CPU load metric c_{loop} means the "average number of clock cycles required by each loop in the scheduler". To estimate c_{loop}, we record the loop count on the microcontroller and time it on the computer. A drawback of this approach is that c_{loop} will be different when the communication time between the computer and the microcontroller changes (section 6.1).

The stress testing environment is non-deterministic because of unpredictable load peaks (section 6.1). For example, when we repeat the best command sequence from the genetic algorithm, the average c_{loop} ranges from 2000 to 4000 in the discrete action space and 13000 to 17000 in the multi-discrete action space (section 6.3). Such uncertainty makes the genetic algorithm less useful in stress testing. On the other hand, random testing cannot improve CPU load.

The deep reinforcement learning configurations based on the PPO algorithm have the best performance in stress testing. For example, the PPO-MultiDiscrete-GGNN configuration reaches $c_{loop} \approx 20000$ in 123 minutes (section 6.9), while the genetic algorithm reaches $c_{loop} \approx 15000$ 9 hours (section 6.3). On the other hand, the D3QN algorithm does not work well in stress testing.

We do not directly maximise the number of failures/anomalies. We only find three bugs and one anomaly of the COMMS software during the experiments (sections 6.10 & 7.10). They cannot provide too much information to guide the algorithms.

9.1.2 Prior Knowledge

This subsection gives answers to the following research question in section 2.4.2:

RQ-2 What type of prior knowledge needs to be encoded?

Chapter 2 lists several types of prior knowledge, including:

1. How to encode/decode commands and telemetry
2. Rules to identify the current state from the telemetry or the interaction history.
3. What action should be taken in the current state.
4. Whether the current state contains an anomaly.
5. A model used to predict the future state of the system under test.
6. Design of the objective function or the reward function.

Table 9-1 summarizes the usage of prior knowledge in this study.

Table 9-1: Usage of Prior Knowledge in Baselines and RL Algorithms

Algorithm	Action Space	Prior Knowl. 1	Prior Knowl. 2	Prior Knowl. 3	Prior Knowl. 4	Prior Knowl. 5	Prior Knowl. 6
Random[28]	Discrete	Only need to encode commands	No	No	No	No	No
Genetic	Discrete	Only need to encode commands	No	No	No	No	Yes
Genetic	MultiDiscrete	No	No	No	No	No	Yes
Q-Learning	Discrete	Yes	Yes	No	Partly. Sometimes the parser cannot parse the reply if an anomaly exists.	No	Yes
D3QN	Discrete	Only need to encode commands	No	No	No	No	Yes
PPO	Discrete	Only need to encode commands	No	No	No	No	Yes
PPO	MultiDiscrete	No	No	No	No	No	Yes

The PPO algorithms in multi-discrete action space use the fewest types of prior knowledge but have the best performance. Compared with sending human-defined commands (discrete action space), it is more desirable to let the algorithms select the parameters in command vectors. Nevertheless, this approach also has a drawback. If the command vectors are too long, the search space will become too large for the PPO algorithm (section 6.7).

On the other hand, although the PPO algorithm is good at triggering high CPU load, it is still weak in detecting an anomaly. Only when the target board crashes, reboots or does not reply for a while testers will notice that there is an anomaly. To detect anomalies effectively, we may need to learn a model of the SUT or add related prior knowledge.

Table 4-3 in section 4.5 gives an overview of the source code files of these algorithms, which also shows prior knowledge usage.

9.1.3 Algorithm Designs

This subsection gives answers to the following research question in section 2.4.2:

RQ-3 Which RL algorithm is suitable for testing command generation?

We answer the complex question from several aspects, including state representation, action space, reinforcement learning algorithms, neural network design, implementation, and debugging.

State Representation

Four types of state representations are available in onboard software testing.

The first representation is a discrete scalar, i.e., we use state identification rules to summarise the current command, response, and the previous interaction as a discrete scalar. The Q-Learning

[28] Random testing can run in the MultiDiscrete action space, but we have not tested this scenario.

algorithm needs such representation. Sections 3.1.2 and 4.1.2 discuss the details. However, as one can imagine, much information is lost with this type of extraction. As a result, the tabular Q-Learning algorithm does not work in the study.

The second state representation is a plain vector, a concatenation of 2 vectors (section 4.2.2): a branch coverage vector and an interaction history vector[29]. This representation is simple, but regression testing is a fundamental challenge. The length of the coverage vector and the corresponding branch of every element in the vector will change after the SUT's source code modification. As a result, the trained neural network becomes useless in regression testing (section 3.4.2).

We also try to represent the coverage information in a graph to solve this problem. We first build a directed graph representation of the tested software. Each node of the graph represents a start of a branch or the start/end of a method. It has a feature vector generated by a Word2Vec model from the file name, the method name, and the node type. Moreover, each edge of the graph indicates a possible transition between nodes during the execution of the tested program. To some extent, the graph looks like a control flow graph.

We compare two methods to generate the graph: analyzing the execution trace of SUT (section 3.4.3) or performing static analysis on the source code (section 3.4.4). The second method is more suitable than the first because it generates a complete graph and node feature vectors. The static analysis is performed with the help of an open-source tool Joern. Joern only produces control flow graphs of every method in the source code. We need to build a graph of the whole program based on Joern's output.

When running RL algorithms with graph input, we concatenate node feature vectors with coverage masks $c_i \in \{0,1\}$ (section 3.4.4). After that, a graph neural network will generate a graph embedding vector, which will be concatenated with an interaction history vector. In the end, the vector is fed into the following layers (section 4.2.3).

The fourth type of state representation is source code files with coverage results (section 3.4.2). Nevertheless, such input can be very long and challenging to process. Thus, we do not try it in the study.

Action Space

We consider two types of action space: Discrete and MultiDiscrete (sections 4.2.2 & 4.3.3). In the Discrete action space, each action is a scalar index of a pre-defined command in a list. On the other hand, a MultiDiscrete action is a command vector. Each vector element is an index of a pre-defined parameter in a list.

There is more flexibility in the MultiDiscrete action space. For example, the genetic and PPO algorithms trigger higher CPU load in the MultiDiscrete action space. They probably organize better commands than the human-defined ones.

MultiDiscrete action space also has an unexpected benefit. The PPO algorithm has a more stable performance in such action space. In stress testing, the variance of c_{loop} the PPO-MultiDiscrete-MLP configuration is much smaller than the PPO-Discrete-MLP (sections 6.6 & 6.7).

However, MultiDiscrete action space is only helpful when the length of command vectors is limited. If a valid command contains too many parameters, the probability of this command being attempted

[29] We do not include interaction history information when testing PPO algorithms in the MultiDiscrete action space. Theoretically it is possible, but we just do not have enough time to do that.

becomes very small (section 6.7). In other words, the MultiDiscrete action space becomes too large for the algorithm to search.

At the same time, if short commands cannot achieve the testing goal, MultiDiscrete action space becomes a lousy option. For example, the PPO-MultiDiscrete-MLP configuration does not trigger good branch coverage in coverage testing (section 7.8).

Testers may have concerns about the safety of the MultiDiscrete action space. In other words, the RL agent may generate a dangerous command to damage the satellite hardware. We do not worry too much about safety in our experiments. It is because the commands sent to the COMMS board will not be damaged. Other developers can only use MultiDiscrete action space in stress testing, where testers try to "destroy" the system.

Note that value-based RL algorithms like D3QN do not directly support MultiDiscrete action space. They must estimate the action-value function $Q(s, a)$ for every combination of the command vector (section 5.9). This process is costly.

Reinforcement Learning Algorithms

We try three reinforcement learning algorithms in the research: tabular Q-Learning, Double Duelling Deep Q Network (D3QN), and the Proximal Policy Optimization (PPO) algorithm.

As mentioned above, the tabular Q-Learning algorithm does not work well in stress and coverage testing (sections 6.4 & 7.4). A reason may be that there are no effective state identification rules.

The D3QN algorithm converges faster than the PPO algorithm in the "filling grid" toy problem (chapter 5). However, converging and maintaining stable performance is difficult when testing the existing onboard software. For example, the D3QN algorithm does not converge in the stress testing (section 6.6). The D3QN-Discrete-MLP configuration converges in the coverage testing, but the variance of branch coverage is significant (section 7.5). The D3QN-Discrete-GGNN configuration fails to converge in coverage testing (section 7.6).

Some reasons may explain this phenomenon. Firstly, value-based algorithms have inherent instability. Minor updates to $\hat{Q}^{\pi}(s, a)$ may significantly change the policy and therefore change the data distribution. Secondly, there is not enough time to tune the hyperparameters of the D3QN algorithm because we implemented the algorithm in the last few months of the research.

The PPO algorithm works well in the filling grid environment, stress testing, and coverage testing. The PPO configurations with 2~3 layers of MLP are very robust. On the other hand, the PPO-MultiDiscrete-GGNN configuration has the best performance in stress testing (section 6.9).

Note that our PPO implementation is different from the standard one. To make the training process more stable, we add a learning rate annealing mechanism to control the magnitude of policy updates (section 4.3.1). Section 5.5 compares the performance of PPO with and without the annealing mechanism. It shows that the mechanism can control the Kullback-Leibler divergence (a metric of policy update) under a threshold. It also prevents the cumulative reward curve from crashing.

Sometimes the annealing mechanism raises the entropy of a trained policy, but the policy still performs well (section 5.5). The exact reason for this phenomenon is not apparent yet.

Neural Network Design

We try three types of neural networks in the study: the Multi-Layer Perceptron network (MLP, i.e., the most straightforward fully-connected layers), the Gated Graph Neural Network (GGNN), and the

Graph Attention Network (GAT). For the graph neural networks, we also try different methods to aggregate node feature vectors into a graph embedding vector.

The RL configurations with only 2~3 MLP layers are usually robust and quick to converge. For example, in the filling grid problem, the D3QN-Discrete-MLP configuration can converge in 10k steps, while D3QN configurations using other neural networks need at least 150k steps (sections 5.2, 5.3, & 5.4).

MLP can only use plain vectors as input. However, to reuse the trained neural network in regression testing, we hope that it can process graph inputs. Ultimately, we try graph neural networks, including GGNN and GAT. GGNN has better performance than GAT in the filling grid problem. PPO-Discrete-GGNN can fill all grids after 700k steps, while PPO-Discrete-GAT can still not fill all grids after 1M steps (sections 5.7 & 5.8). One possible reason is that GGNN can have up to 20 layers, while GAT usually has fewer.

Compared with the configurations with MLP, the configurations with GGNN trigger higher CPU load in stress testing (section 6.11). However, they also cover fewer branches in coverage testing (section 7.11). There is still no clue about this phenomenon.

However, one thing is sure. In stress and coverage testing, GGNN will give a more considerable variance in the results. The CPU load or coverage curves with GGNN usually oscillate. The PPO-MultiDiscrete-GGNN configuration is a lucky exception: the PPO algorithms in the MultiDiscrete action space have relatively stable performance.

The graph neural network updates the node feature vectors. After that, we must aggregate the feature vectors into a graph embedding. We try different aggregation methods in a supervised learning task (not shown in the document), including:

- sum, $mean$, and max of all node features.
- A node selection pooling method in (Dai et al., 2019).
- Using a 1D convolution layer to aggregate node features (Zhou et al., 2019).
- Several graph pooling layers provided by the Pytorch Geometric library (Fey & Lenssen, 2019).
- Attentive pooling layer in (Li et al., 2015).
- Feature compression, i.e., using an MLP to reduce node embeddings to scalars and then concatenate these scalars into a graph feature vector

In the supervised learning task, attentive pooling and feature compression perform best. Therefore, we try both approaches in Chapters 5, 6, and 7. According to our experience, they each perform better in specific scenarios. However, attentive pooling usually takes more steps to train. In the filling grid problem, PPO-Discrete-GGNN with feature compression only needs 60k steps to converge, while the attentive pooling approach needs 700k (section 5.7).

Implementation and Debugging

The RL implementations in this study are lightweight. For example, the most complex PPO-MultiDiscrete-GGNN configuration only needs 290 lines of code. For more details, Table 4-3 gives an overview of the source code files of our algorithms.

We only use Pytorch and Pytorch Geometric to build the model without the help of reinforcement learning libraries like RLlib. Our implementation takes reference from CleanRL, a repository including many single-file implementations of reinforcement learning algorithms.

A trick to debug the RL algorithm starts with simple scenarios and the most straightforward configurations. We started with the filling grid toy problem (section 5.1) in the study and simple

configurations like PPO-Discrete-MLP and D3QN-Discrete-MLP. Initially, both configurations could not fill the 36 grids, so we had to reduce the grid number to 4 and debug them.

9.1.4 Testing Environment

This subsection gives answers to the following research question in section 2.4.2:

RQ-4 What kind of testing environment should be used?

Section 3.3 describes the hardware and software setup in this study. The setup has the following features:

- The test command generation tool can run on the lab PC or a remote server.
- Both the EGSE and the OBC can be the master of the bus.
- The radio on the COMMS board can send commands to itself to simulate the wireless channels.

Testers need to make a trade-off between communication delay and computing power. The lab PC has a short communication delay with the SUT and less computing power. By contrast, the remote server has a longer communication delay and more computing power. Figure 3-13 shows the time delays of the remote server and the lab PC.

Several challenges still exist:

- The training process spends most of the time on communication rather than updating the neural networks.
- It cannot simulate the interaction between the satellite and the space environment.
- The RL agent may generate unsafe commands.

We will discuss these challenges later.

9.1.5 Reuse a Trained Agent

This subsection gives answers to the following research question in section 2.4.2:

RQ-5 Can we use a trained RL agent to test other software versions?

Yes. In section 8.3, an agent trained in section 6.9 tests another version of COMMS software. The agent maintains a high CPU load from the beginning. Another agent, trained from scratch, only achieves a similar performance after 2000 steps (~18 minutes).

Chapter 8 also shows that reusing a trained agent is better than using the genetic algorithm's best solution.

There are some conditions for regression testing:

- As mentioned in section 3.4.4, the neural network architecture should be independent of the source code's number of probes. The Gated Graph Neural Network (GGNN) with attentive pooling is used in section 8.3.
- When the two versions of the software under test are too different, generalization of RL can be a problem. Some techniques to solve the generalization problem include multi-task learning, meta-learning, and causal reasoning.

9.1.6 A Brief Answer to Main Research Question

Now we can answer the main research question:

Can a reinforcement learning-based testing tool generate testing commands for small satellites with limited prior knowledge?

Yes. The PPO-MultiDiscrete-GGNN configuration in stress testing is an example. It is beneficial to do so under some conditions:

- Testers are performing stress testing, which is a non-deterministic environment. Compared with other options, RL is better in such an environment. Unsafe or invalid commands may also not be a problem in stress testing.
- They use the PPO algorithm. Other policy-based algorithms may perform similarly, but we did not try them. The D3QN algorithm, as a representative of value-based RL, does not perform well on Delfi-PQ software.
- If they use the MultiDiscrete action space, a command vector should not be too long. PPO also has some benefits in the Discrete action space.
- If they do not want regression testing, the PPO algorithm with 2~3 layers of MLP is a simple and robust option. It can converge quickly without too much computing power.
- If they want to use the RL agent in regression testing, GGNN with attentive pooling is a good neural network architecture. However, it usually takes a longer time to train.

There may be more scenarios and RL testing agent designs waiting to be discovered!

9.2. Threats to Validity

There are some threats to the validity of the conclusions. They are listed here:

Threat-1 In this study, there is insufficient time to tune the hyperparameters or find the optimal neural network architecture.

Explanation This threat cannot change the study's main conclusion, i.e., RL is more suitable for stress testing.

Indeed, the lack of tuning may make us underestimate the performance of some RL configurations. For example, the D3QN algorithm may be able to perform better than we think. However, the performance of random testing cannot grow further. A fixed command sequence from the genetic algorithm still has uncertain performance in a non-deterministic environment like stress testing. As a result, RL-based testing is still a better option for stress testing.

Threat-2 $t_{interval}$ does not include the lab PC or server decision-making time. When testing the software, random and genetic testing spend less time on decision-making. Given the same period, they may be able to send more commands than RL-based algorithms and trigger a higher average CPU load.

Explanation The computing power of the CPU strongly affects the decision-making time. To exclude the effect of CPU power in the comparison, we excluded the decision-making time $t_{interval}$.

Although different algorithms have various decision-making times, communication is usually shorter. In Figure 6-1, decision-making time only takes 42ms in a time step. By contrast, 148ms is spent on communication. Thus, even if random testing takes a

short time to select a command, it still cannot send too many commands in a given period because of the long communication time.

A more powerful PC will further reduce the effect of decision-making time.

Threat-3 For random and genetic testing, it is unnecessary to collect code coverage after sending a testing command.

Explanation Section 6.1 mentions the reason. Measuring instantaneous CPU load with very high variance is not helpful.

Threat-4 Instrumentation of the source code and code coverage collection may change the behaviors of the SUT. It is also difficult to predict the consequences of such influence.

Explanation It is true. We have observed such influence in section 3.4.1. The instrumented program needs to call CodeCount() at the start of every branch, which makes the program run slower.

However, the MSP432P series microcontrollers do not support non-intrusive coverage collection. For some other controllers, like the MSP432E series, which supports tracing, we can use a Segger J-Trace Pro to collect coverage without instrumentation.

Threat-5 Assumption-2 in section 2.4.1 assumes that the performance of software testing algorithms can be measured by metrics like code coverage and CPU load. However, these metrics do not directly relate to the number of anomalies found.

If RL-based testing can reach high coverage/CPU load but still needs human-defined rules to detect anomalies, then this approach does not reduce the usage of prior knowledge.

Explanation At this stage, the RL-based testing detects anomalies from some phenomena like crashes, reboots, no response, and late responses. In practice, it may still need human-defined rules in fault detection.

However, fault detection rules are only one type of prior knowledge. As shown by Figure 9-1, RL still does not need some types of prior knowledge in manual testing, e.g., what action should be taken under the current state.

Threat-6 As mentioned by assumption-4 in section 2.4.1, we cannot directly measure the amount of prior knowledge. RL uses fewer types of prior knowledge, but testers may still spend significant time on the prior knowledge.

Explanation In this study, RL-based testing only uses one type of prior knowledge, i.e., the design of the reward function. However, as shown in sections 6.1 and 7.1, our reward design is effortless and straightforward. Designing these rewards does not take too much time.

Other types of prior knowledge are shared by RL-based testing and traditional methods. The time to define this knowledge should be the same.

Threat-7 The assumption-5 may not be valid. In other words, the conclusions from the COMMS onboard software cannot be generalized to other onboard software.

Explanation Limited by time and resource, only the COMMS onboard software of the Delfi-PQ satellite is tested in the study. However, as mentioned in section 3.2.5, the software has a similar complexity to other educational CubeSat/PocketQube software. The conclusions here should be able to apply to these systems.

The onboard software of larger satellites usually has more complex applications. The use of RL for testing such software is still a gap.

9.3. Contributions to the Academic Field

This study has the following contributions:

- As far as we know, it is the first research that applies RL-based testing on onboard software and one of few studies that use RL to test embedded software without GUI.
- Unlike most RL-based testing research that relies on GUI information, it utilizes near real-time code coverage information from the software under test to compute states and rewards. The idea is developed by (Dai et al., 2019).
- To the end, an open-source tool is written to retrieve the code coverage data and can be easily modified to adapt other embedded software. Another tool to generate a complete graph from the source code is also implemented.
- It is large-scale research that covers many aspects:
    - 2 types of testing goals: maximizing code coverage and maximizing the CPU load.
    - 2 environments: a toy problem and the COMMS onboard software.
    - 3 reinforcement learning algorithms, including the Q-Learning, the Double Duelling Q Network (D3QN), and the Proximal Policy Optimization (PPO) algorithm to learn. Each algorithm has several configurations.
    - 2 state representations: direct vector input or graph input.
    - 2 action space: Discrete and MultiDiscrete.
    - Run the RL testing agent on a local PC or a remote server.
    - Different neural network architectures, including multi-Layer Perceptron MLP, Gated Graph Neural Network GGNN, and Graph Attention Network GAT.
    - Different graph aggregation methods, such as feature compression and attentive pooling.
    - Perform both regular testing and regression testing.
    - The performance of RL-based testing is compared with two baselines: random command generation and the genetic algorithm.
- It reveals that RL-based testing has advantages in highly non-deterministic environments like stress testing under some conditions.
- It reveals that RL-based testing with a graph neural network has advantages in regression testing under some conditions.

9.4. Recommendations for Future Research

Based on experience of this study, this section gives the following recommendations for future research.

- **Improve sampling efficiency**. Figure 6-1 shows that most training time is spent communicating with the target board. To make matters worse, there is only one target board, so parallel testing is not feasible. Training a more complex and powerful agent in this situation may take much longer.

It is also difficult to train the model in a simulated environment. Building a simulator for the microcontroller and the peripherals is more time-consuming than writing test cases if the manufacturer does not provide such a simulator.

A possible option is to use model-based reinforcement learning, which learns a model to predict rewards and future states. It may need much fewer samples to train. For example, EfficientZero (Ye, Liu, Kurutach, Abbeel, & Gao, 2021) mastered the Atari game in a shorter time than humans.

- **Apply the learned model in other fields**. If a model is learned during the software test, engineers may apply it in other fields, e.g., fault detection in the daily operation of the satellite. For example, if the real telemetry from the satellite is different from the model's prediction, there may be an anomaly.

  Traditionally, the prediction model is built by engineers, which can be time-consuming. A large amount of telemetry data can also train the model. However, there is not much telemetry when the satellite is just launched. Therefore, training a model during software testing may be a novel and helpful approach.

- **Simulate sensors and actuators in the test**. This study does not simulate sensors and actuators, but such simulation is necessary in many cases, e.g., testing the ADCS onboard software. Note that such environment simulators can be shared by different satellite programs, even if these programs use different hardware. Some environment simulators, like 42 (Geletko et al., 2019), have been used in onboard software testing.

- **Use reinforcement learning in regression testing**. Section 8.3 shows promising results in regression testing. However, as far as we know, few studies use RL in regression testing yet, and it may be an interesting gap.

  One problem can be generalization capability, from which many RL algorithms suffer. It means the trained agent is overfitted to a specific environment and cannot adapt to another one. Luckily, we do not meet this challenge in chapter 8.

Bibliography

Adamo, D., Khan, M. K., Koppula, S., & Bryce, R. (2018, November). Reinforcement learning for android gui testing. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (pp. 2-8).

Ahmad, T., Ashraf, A., Truscan, D., & Porres, I. (2019, August). Exploratory performance testing using reinforcement learning. In 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 156-163). IEEE.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... & Zhu, H. (2013). An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8), 1978-2001.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. Information and computation, 75(2), 87-106.

Bagherzadeh, M., Kahani, N., & Briand, L. (2021). Reinforcement learning for test case prioritization. IEEE Transactions on Software Engineering.

Bauersfeld, S., & Vos, T. (2012, September). A reinforcement learning approach to automated gui robustness testing. In Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012) (pp. 7-12).

Belli, F., Budnik, C. J., Hollmann, A., Tuglular, T., & Wong, W. E. (2016). Model-based mutation testing—approach and case studies. Science of Computer Programming, 120, 25-48.

Bergdahl, J., Gordillo, C., Tollmar, K., & Gisslén, L. (2020, August). Augmenting automated game testing with deep reinforcement learning. In 2020 IEEE Conference on Games (CoG) (pp. 600-603). IEEE.

Bocchino, R., Canham, T., Watney, G., Reder, L., & Levison, J. (2018). F Prime: an open-source framework for small-scale flight software systems.

Bouwmeester, J., Aalbers, G. T., & Ubbels, W. J. (2008, August). Preliminary mission results and project evaluation of the delfi-c3 nano-satellite. In 4S Symposium Small Satellites Systems and Services (Vol. 660, p. 25).

Bouwmeester, J., van der Linden, S. P., Povalac, A., & Gill, E. K. A. (2018). Towards an innovative electrical interface standard for PocketQubes and CubeSats. Advances in Space Research, 62(12), 3423-3437.

Bouwmeester, J., Radu, S., Uludag, M. S., Chronas, N., Speretta, S., Menicucci, A., & Gill, E. K. A. (2020). Utility and constraints of PocketQubes. CEAS Space Journal, 12(4), 573-586.

Bouwmeester, J., Menicucci, A., & Gill, E. K. (2022). Improving CubeSat reliability: Subsystem redundancy or improved testing. Reliability Engineering & System Safety, 220, 108288.

Cai, T., Zhang, Z., & Yang, P. (2020, October). Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd. In Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test (pp. 93-96).

Chen, J., & Wu, J. (2010, August). GMC: A performance model checker for concurrent systems. In 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE) (Vol. 3, pp. V3-6). IEEE.

Choi, W., Necula, G., & Sen, K. (2013). Guided gui testing of android apps with minimal restart and approximate learning. Acm Sigplan Notices, 48(10), 623-640.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

Dai, H., Li, Y., Wang, C., Singh, R., Huang, P. S., & Kohli, P. (2019). Learning transferable graph exploration. Advances in Neural Information Processing Systems, 32.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability, 68(3), 1189-1212.

ECSS-E-ST-40C – Software. (2009, March). ESA-ESTEC Requirements & Standards Division.

Esteve, M. A., Katoen, J. P., Nguyen, V. Y., Postma, B., & Yushtein, Y. (2012, June). Formal correctness, safety, dependability, and performance analysis of a satellite. In 2012 34th International conference on software engineering (ICSE) (pp. 1022-1031). IEEE.

Even-Dar, E., Kakade, S. M., & Mansour, Y. (2009). Online Markov decision processes. Mathematics of Operations Research, 34(3), 726-736.

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. arXiv preprint arXiv:1903.02428.

Fujimoto, S., Hoof, H., & Meger, D. (2018, July). Addressing function approximation error in actor-critic methods. In International Conference on Machine Learning (pp. 1587-1596). PMLR.

Garousi, V., Felderer, M., Karapıçak, Ç. M., & Yılmaz, U. (2018). Testing embedded software: A survey of the literature. Information and Software Technology, 104, 14-45.

Geletko, D. M., Grubb, M. D., Lucas, J. P., Morris, J. R., Spolaor, M., Suder, M. D., ... & Zemerick, S. A. (2019). Nasa operational simulator for small satellites (nos3): the stf-1 cubesat case study. arXiv preprint arXiv:1901.07583.

Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., & Lopez, C. (2012, November). Lightweight automated testing with adaptation-based programming. In 2012 IEEE 23rd International Symposium on Software Reliability Engineering (pp. 161-170). IEEE.

Groz, R., Simao, A., Bremond, N., & Oriat, C. (2018, May). Revisiting AI and testing methods to infer FSM models of black-box systems. In 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST) (pp. 16-19). IEEE.

Ha, D., & Schmidhuber, J. (2018). Recurrent world models facilitate policy evolution. Advances in neural information processing systems, 31.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, July). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning (pp. 1861-1870). PMLR.

Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. Advances in neural information processing systems, 30.

Harries, L., Clarke, R. S., Chapman, T., Nallamalli, S. V., Ozgur, L., Jain, S., ... & Ciosek, K. (2020). Drift: Deep reinforcement learning for functional software testing. arXiv preprint arXiv:2007.08220.

Harman, M. (2011). Software engineering meets evolutionary computation. Computer, 44(10), 31-39.

Harman, M., Jia, Y., & Zhang, Y. (2015, April). Achievements, open problems and challenges for search based software testing. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST) (pp. 1-12). IEEE.

Holland, J. H. (1992). Genetic algorithms. Scientific american, 267(1), 66-73.

Howard, R. A. (1960). Dynamic programming and markov processes.

Huang, S., Dossa, R. F. J., Ye, C., & Braga, J. (2021). CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *arXiv preprint arXiv:2111.08819*.

Huang, Shengyi, Rousslan Fernand Julien, Dossa, Antonin, Raffin, Anssi, Kanervisto, & Wang, Weixun. (2022, March 22). The 37 Implementation Details of Proximal Policy Optimization. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. Retrieved 31 August 2022, from undefined.

Jacklin, S. A. (2015). Survey of verification and validation techniques for small satellite software development (No. ARC-E-DAA-TN23631).

Jacklin, S. A. (2019). Small-satellite mission failure rates (No. NASA/TM-2018-220034).

King, J. C. (1975). A new approach to program testing. ACM Sigplan Notices, 10(6), 228-233.

Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

Langer, M., & Bouwmeester, J. (2016). Reliability of CubeSats-statistical data, developers' beliefs and the way forward.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015). Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Lodge, M. (2021, February 1). Reinforcement learning in diffblue cover. Diffblue. Retrieved August 1, 2022, from https://www.diffblue.com/blog/ai/testing/reinforcement-learning-in-diffblue-cover/

Longa, A, & Pellegrini, G. (2022, May 11). Tutorial 9: Recurrent GNNs. https://github.com/AntonioLonga/PytorchGeometricTutorial. Retrieved 13 September 2022, from Github.

Manyak, G. D. (2011). Fault tolerant and flexible cubesat software architecture.

Mao, K., Harman, M., & Jia, Y. (2016, July). Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 94-105).

Mao Ke. (2018). Sapienz: Intelligent Automated Software Testing at Scale. https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

McComas, D. (2021). OpenSatKit-A Flight Software System Educational Platform.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). PMLR.

Moghadam, M. H., Hamidi, G., Borg, M., Saadatmand, M., Bohlin, M., Lisper, B., & Potena, P. (2021, June). Performance testing using a smart reinforcement learning-driven test agent. In 2021 IEEE Congress on Evolutionary Computation (CEC) (pp. 2385-2394). IEEE.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). Foundations of machine learning. MIT press.

Nakkasem, T. (2020, April 16). V-model. Medium. Retrieved July 17, 2022, from https://medium.com/software-engineering-kmitl/v-model-3a71622b3d82

Omri, S., & Sinz, C. (2021). Machine Learning Techniques for Software Quality Assurance: A Survey. arXiv preprint arXiv:2104.14056.

Pan, M., Huang, A., Wang, G., Zhang, T., & Li, X. (2020, July). Reinforcement learning based curiosity-driven testing of android applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 153-164).

Pani, P. (2014). Measuring code coverage on an embedded target with highly limited resources (Doctoral dissertation, Master's Thesis. Graz University of Technology).

Păsăreanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., & Pape, M. (2008, July). Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In Proceedings of the 2008 international symposium on Software testing and analysis (pp. 15-26).

Patel, P., Srinivasan, G., Rahaman, S., & Neamtiu, I. (2018, May). On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In Proceedings of the 13th International Workshop on Automation of Software Test (pp. 34-37).

Radu, S., Uludag, M. S., Speretta, S., Bouwmeester, J., Gill, E., & Foteinakis, N. C. (2018). Delfi-PQ: The first pocketqube of Delft University of Technology. In 69th International Astronautical Congress, Bremen, Germany, IAC.

Reichstaller, A., Eberhardinger, B., Knapp, A., Reif, W., & Gehlen, M. (2016, October). Risk-based interoperability testing using reinforcement learning. In IFIP International Conference on Testing Software and Systems (pp. 52-69). Springer, Cham.

Romdhana, A., Merlo, A., Ceccato, M., & Tonella, P. (2022). Deep reinforcement learning for black-box testing of android apps. ACM Transactions on Software Engineering and Methodology.

Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems (Vol. 37, p. 20). Cambridge, England: University of Cambridge, Department of Engineering.

Sant, J., Souter, A., & Greenwald, L. (2005, May). An exploration of statistical models for automated test case generation. In Proceedings of the third international workshop on Dynamic analysis (pp. 1-7).

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., ... & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839), 604-609.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Schulman, John. (2020, March 7). Approximating KL Divergence. Retrieved 31 August 2022, from undefined.

Schwartz, J., & Kurniawati, H. (2019). Autonomous penetration testing using reinforcement learning. arXiv preprint arXiv:1905.05965.

Shirole, M., & Kumar, R. (2013). UML behavioral model-based test case generation: a survey. ACM SIGSOFT Software Engineering Notes, 38(4), 1-13.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484-489.

Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017, July). Reinforcement learning for automatic test case prioritization and selection in continuous integration. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 12-22).

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., ... & Su, Z. (2017, August). Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 245-256).

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

Test.ai. (2021). All Products. https://test.ai/ai-deep-dive

Timmons, E. J. (2020, April). Core Flight System (cFS) Training (No. 20205000691). NASA Technical Report.

Tran, K., Akella, A., Standen, M., Kim, J., Bowman, D., Richer, T., & Lin, C. T. (2021). Deep hierarchical reinforcement agents for automated penetration testing. arXiv preprint arXiv:2109.06449.

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model‐based testing approaches. Software testing, verification and reliability, 22(5), 297-312.

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

Veanes, M., Roy, P., & Campbell, C. (2006). Online testing with reinforcement learning. In Formal Approaches to Software Testing and Runtime Verification (pp. 240-253). Springer, Berlin, Heidelberg.

VELIČKOVIĆ, Petar, et al. Graph attention networks. arXiv preprint arXiv:1710.10903, 2017.

Vuong, T. A. T., & Takada, S. (2018, November). A reinforcement learning based approach to automated testing of android applications. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (pp. 31-37).

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3), 229-256.

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering atari games with limited data. Advances in Neural Information Processing Systems, 34, 25476-25488.

Yuen, B., & Sima, M. (2019). Low-Cost Radiation Hardened Software and Hardware Implementation for CubeSats. arXiv preprint arXiv:1902.04117.

Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., ... & Fan, C. (2019, November). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 772-784). IEEE.

Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., & Liu, Y. (2021, May). Automatic web testing using curiosity-driven reinforcement learning. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 423-435). IEEE.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems, 32..0