SELF-ADAPTATIONS IN SDN-BASED IOT NETWORKS

CHARLES OREDOLA

Master of Science (Technology) Thesis

Supervisor: Adnan Ashraf

Software Engineering Laboratory

Department of Information Technologies

Åbo Akademi University

May 2023

ABSTRACT

In the digital age, frightening patterns in digital threats are emerging. It is impossible to ignore threats to IoT networks. Threats can take on any of the typical forms, including Denial-of-Service (DoS), Distributed Denial-of-Service (DDoS), Virus assault, Man-in-the-middle attack (Mitm), Advanced Persistent Threats (APT), Password Assault, and more. It is crucial to eliminate all threats from IoT networks and devices.

Reinforcement learning to detect anomalies in an IoT network is seen to be the greatest option for correcting risks in a network, hence fixing the afflicted nodes, according to this thesis, "Self-Adaptation of SDN-based IoT Networks." (Markov) MDP policies and MAPE-K loop properties in Self-aware systems are the bases of the design in this thesis. The network system exhibited self-adaptability features, which makes it self-correcting and self-healing.

The objective of this research is to propose a means to secure the devices in an IoT network by protecting them from any form of threats and ensuring that the devices function normally. Even at the advent of abnormal functioning of any node in the network, the system should be able to correct itself.

A Software Defined Network (SDN) architecture is proposed for the design in a later section, which explains the kind of SDN that should be in place for the intrusion detection system. Further into the thesis, we dived deep into the general overview of deep reinforcement learning. Then comes the implementation, which talks about the kind of reinforcement learning policy used in the work and how the result was derived. The other section discusses the result and discussion, where the result in this work was compared with the result of the traditional machine learning algorithm.

ACKNOWLEDGEMENT

I want to express my profound gratitude to my supervisor, Adnan Ashraf (Ph.D.) for his support, encouragement, and advice throughout my research work. I also want to sincerely appreciate my family for their understanding and timely prayers for the successful completion of this work.

Charles Oredola

Turku, May 31, 2023

CONTENTS

Abstract i

Acknowledgements ii

Contents iii

List of Figures vi

List of Tables vii

List of Equations viii

1. ***Introduction*** 1
   - 1.1. *IoT in General* 1
   - 1.2. *Self-Adaptation Properties of an IoT Network* 2
   - 1.3. *SDN-Enabled IoT Architectures* 2
   - 1.4. *SDN Nature and Security Framework for IoT* 3
2. ***Background And Related Works*** 4
   - 2.1. *Background* 4
   - 2.2. *Related Works* 5
3. ***Proposed Architecture*** 7
   - 3.1. *The Software-defined Network* 7
     - 3.1.1. *SDN-based Ad-Hoc Architecture* 7
     - 3.1.2. *SDN-based IoT Architecture* 10
     - 3.1.3. *Distributed SDN Security Solution* 11
     - 3.1.4. *DRL-based Intrusion Detection System* 13
     - 3.1.5. *Cyber Security Attacking Scenarios* 15
4. ***General Overview of Deep Reinforcement Learning (DRL)*** 18

4.1. RL and DL (The Two Essential Building Blocks of DRL) 18
4.1.1. Fundamental DRL Algorithm 20
4.1.2. Advanced DRL Algorithm 31
5. Implementation 34
5.1. IoT Network Architecture 35
5.2. Markov Decision Process 37
5.3. Model-Free Solution Technique 37
5.3.1. Training with PPO Algorithm 39
5.3.2. Policy Iteration Method 40
5.3.3. Sampling Method and Reward Function 43
6. Result and Discussion 47
6.1. Learning Rate and The Reward Function Curve 48
7. Conclusion and Further work 52
7.1. Exploring Capabilities of Model-based DRL Methods 52
7.2. Training DRL in Adversarial Cyber Environments 53
Bibliography 55

List of Figures

1. *Figure 1 – Equal Interaction SDN Networks* 8
2. *Figure 2 – A Node in Ad-Hoc Plane* 9
3. *Figure 3 – Extended SDN Domain* 10
4. *Figure 4 – Distributed Ad-Hoc Control Plane* 10
5. *Figure 5 – Grid of Security in SDN Domain* 13
6. *Figure 6 – General Methods of DRL* 19
7. *Figure 7 – Basic DRL Algorithms Organization* 20
8. *Figure 8 – DQN and DDQN with Target Networks* 22
9. *Figure 9 – Monte Carlo Policy Gradient and Actor Critics* 28
10. *Figure 10 – MAPE-K Loop; Self-Adaptation Loop* 34
11. *Figure 11 – General Feature of IoT Gateway* 36
12. *Figure 12 – IoT Network from GettyImages* 36
13. *Figure 13 – Sample of Agent Training using the PPO Model* 42
14. *Figure 14 – Policy Evaluation – Improvement Cycle* 43
15. *Figure 15 – Train Clipping Curve* 49
16. *Figure 16 – Episode Reward Mean* 51
17. *Figure 17 – Train Learning Rate* 51

List of Tables

1. *Table 1 – TON Dataset Features* 47
2. *Table 2 – TON Dataset Features Selected for Training* 47
3. *Table 3 – Performance of the ML-based solution on three different scenarios of threats test* 48

List of Equations

1. *Equation 1 – Monte Carlo Policy Evaluation* 38
2. *Equation 2 – Original Code Snippet* 38
3. *Equation 3 – Policy Iteration Algorithm* 45
4. *Equation 4 – Value Iteration Algorithm* 46
5. *Equation 5 – The Episodic Part of Value Iteration* 46
6. *Equation 6 – ML-Algorithm* 50

1. INTRODUCTION

1.1. IoT in General

The Internet of Things is the hub for connecting both real-world and virtual smart devices. IoT devices are small, low power, and have a limited processing capacity. The gadgets collect information about the surroundings and transmit it to the gateways for further processing. IoT systems can be developed using traditional technologies to provide a variety of services, but they are standalone systems. IoT expands existing technologies, such as radio frequency identification and wireless sensor networks, which have limited compatibility, to fulfill application-specific needs. IoT solutions enable direct or indirect communication between smart devices. Different proprietary protocols have been established since the inception of IoT, some of which are not compatible with other devices. There are certain established protocols among them. Some of these protocols have been standardized to allow for cross-device use. TCP, UDP, IEEE 802.15.4, 6LoWPAN, and other protocols are examples of interoperable or standardized protocols.

IoT sensors and gadgets respond to dynamic events and can adjust courses to accommodate environmental changes. The Internet of Things sensors and devices can receive instructions for installing, updating, or doing other management-related operations from a distance. These nodes receive prompt updates or orders from a remote management server to start up or shut down. The nodes' capacity for self-configuration reduces the need for manual changes made by humans.

For administrative purposes, identifying each device in an IoT network is easy, because each one has a distinct identity. Either IP addresses, URIs, or RFID tags are used to provide these distinct identities. An IoT network's sensors or devices may be linked to the Internet or another form of communication, and the data produced may be pre-processed locally or in the cloud. To make the best decisions, the knowledge would be taken from the data (Suryateja, n.d.).

1.2. Self-adaptation Properties of IoT Network

Self-adaptation is the ability of a system or network to continuously learn from and adjust to changing operational conditions (Lalanda et al., 2013). This is relevant when the system malfunctions and reconfigures itself, i.e., when it self-manages by defending, improving,

rearranging, or mending itself. A typical self-adaptation system uses a MAPE-K adaptation control loop, which consists of four main steps: monitoring the system and its environment, analyzing the data gathered from the system and its environment to determine whether adaptation is necessary, planning the adaptation, and carrying out the adaptation by implementing on the system.

1.3. IoT Architectures with SDN Support

SDN and IoT are separate technologies. In contrast to SDN, which is linked to network routing and serves as an orchestrator for network-level management, IoTs mostly comprise sensing devices that attribute distinct communication networks. IoT is a tiered architecture with numerous technologies at each level, whereas SDN separates the control plane from the data plane and offers vendor freedom. Consequently, IoT can benefit from the SDN control plane because SDN promises to meet new service expectations for the traditional network.

IoT architecture consists of three basic layers:

1. The perception layer: This is made up of tangible items and sensors.
2. Network layer: This sends data from physical devices to the network's gateway and edge.
3. Application layer: This handles user-requested applications and services.

There are three fundamental SDN architecture levels:

1. Data plane: The data plane comprises dumb forwarding devices, such as switches and routers, which only forward data when instructed to do so by the controller.
2. Control plane/controller: The controller functions as the network's brain and oversees its overall operation. The application layer abstracts the needs of the client, and the controller receives this information via northbound APIs, such as the Restful API. The controller oversees the entire network and has a thorough understanding of it.
3. Application layer: The controller runs above all other apps and programs.

1.4. IoT Security Framework and SDN Nature

In a wider network, IoT devices are susceptible to security risks. The authentication of IoT devices on the controller is the main emphasis of the security framework, so when a wireless object connects to the controller, it immediately blocks all ports and begins authenticating the device. The controller starts pushing flow only to an authenticated user. Several network controllers act as security guards and information concerning user authentication is shared between parties. If a guard controller fails, another border controller is chosen to serve as the security controller (Tayyaba et al., 2017).

The foundation of IoT network management is Software-Defined Networking (SDN). It makes the underpinning network programmable. Comparing traditional computer networks to SDN-based systems and networks, traditional networks' monitoring solutions call for ad-hoc software installation/configuration and low-level tools. SDN has offered a set of straightforward and reusable primitives for the gathering of network variables at various granularity levels, making them useful for a variety of management tasks. Network operators can dynamically designate the flows to monitor based on different packet fields (e.g., source and/or destination IP addresses) and count the number of bytes or packets for these flows using SDN flow-based switches (e.g., Open-Flow enabled devices) (Tangari et al., 2018).

2. BACKGROUND AND RELATED WORKS

2.1. BACKGROUND

The main ideas that are important to comprehend this work are explained in this chapter. Additionally, it describes the technology under inquiry and earlier research on the subject.

DDoS attacks employing hacked IoT devices are the most common kind of attack on IoT networks. The primary objective of this research work is to identify anomalies in the IoT network layer. To mitigate security risks, IoT networks must be able to self-protect and self-adapt. Due to their complexity and magnitude, self-adapting large-scale systems using machine learning are actively explored, and they are the subject of extensive research because they encapsulate the underlying network architecture. Solutions proposed by Narayanankutty, (2021) are useful for network control in complex systems due to their inherent properties of manageability, dynamism, cost-effectiveness, and adaptability by abstracting the underlying network architecture.

This work highlights self-adaptation in SDN-based IoT (Internet of Things) networks using reinforcement learning, which refers to a technique that allows an IoT network to adapt and optimize its behavior in response to changes in its environment or operating conditions. In this approach, the network uses a form of artificial intelligence known as reinforcement learning to learn from experience and make decisions that maximize its performance. This work reflects the use of the new module in reinforcement learning, the stable-baselines3 module. Here, the recreation of a quite simple environment, which is assumed to be a collection of nodes and edges in a network, with a state-space assumed to be anomalies-free was considered. Anomalies are introduced to the environment for model testing to ascertain the agent's level of capability. The agent explores the environment to detect introduced anomalies, and the reconfiguration process begins. This accounts for self-adaptation in IoT networks.

In a software-defined networking (SDN) architecture, a centralized controller that can dynamically adjust network resources and routing paths in response to changing network conditions controls the network. This architecture is well suited to IoT networks, which typically involve many devices with varying resource requirements and connectivity.

2.2. Related works

Research on the Software Defined Networks (SDN) on the Internet of Things is constantly gaining momentum. Software Defined Network (SDN) currently enables automatic network monitoring in network systems, in contrast to the conventional Computer Network, where low-level tools configuration and software installations are carried out. This section will examine some research projects related to self-adapting IoT networks.

By automatically and intelligently analyzing network flows, Sarica and Angin (2020) proposed an automated injection intrusion detection and mitigation approach that offers security in SDN-based networks. Next, mitigation measures are implemented in line with the intrusion detection component's determination. Li et al. (2022) improved an SDN's programmable controller with the well-known MAPE-K self-adaptation loop using genetic programming (GP).

Their strategy:

1. Periodically checks the SDN for congestion to create a model of the SDN that will be utilized to resolve congestion.
2. Applying GP allows the SDN data-forwarding algorithm's logic to evolve to reduce transmission time and changes to current data-transmission channels while also resolving congestion.
3. The logic of the SDN data-forwarding algorithm is updated to reduce future congestion and existing transmission channels are modified to address the current congestion.

The three categories of physical autonomous systems that Lei et al. (2020) concentrated on reviewing were autonomous robotics, intelligent cars, and the smart grid. Additionally, they discussed the uses of DRL for generic autonomous physical systems in IoT edge/fog/cloud computing systems and IoT communication networks. Alfonso et al. (2021) claim that without the need for earlier configuration by human operators, machine-learning systems may automatically distinguish between normal and abnormal patterns and notify a client or third party when something deviates from the observed standards. Learning algorithms for IoT systems can also aid in preventing disruptive events that affect system availability and QoS, for example, to predict when a hardware component might fail to take action and avoid system downtime.

Predictive maintenance of physical devices is one of the tasks that can be efficiently predicted by algorithm learning.

3 PROPOSED ARCHITECTURE

3.1 THE SOFTWARE-DEFINED NETWORK

A new paradigm for fostering innovation in networking research and development is Software Defined Networking (SDN). The network intelligence and state are logically centralized, and the control and data planes are disconnected. Through a secured OpenFlow channel, a new device called a controller connects to the switch and controls it using the OpenFlow protocol. Both proactively using pre-established rules and reactively in reaction to packets, the controller can add, amend, and delete flow entries. Additionally, SDN enables the deployment of dynamic security policies, granular traffic filtering, and quick response to security risks. Inside the controller, the SDN architecture offers a programmable interface. SDN makes it possible for network control activities to:

- utilize one or more server platforms with improved performance,
- make use of open operating systems and hardware that are vendor agnostic,
- exchange data via established protocols with other operating systems or control platforms.

By pushing security policies to those devices, SDN design expands the security perimeter to include network access endpoint devices (access switches, wireless access points, etc.). After connecting to the OpenFlow switches, the SDN controller creates a global network view based on the data received via the OpenFlow protocol. The SDN controller might additionally:

- use the Link Layer Discovery Protocol (LLDP) for network discovery,
- compile data on network traffic using a specific field in the flow rules that the controller had previously placed (Flauzac et al., 2015).

3.1.1 SDN-based Ad-Hoc Architecture

The default comportment of a controller is the same interaction. It has complete access to the switch and is subject to the same regulations as all other controllers (Flauzac et al., 2015).

A. SDN Architecture for Ad-Hoc Network

An SDN-based architecture for Ad-Hoc Networks is provided in this section. The Ad-Hoc network node is regarded as a combination of the following as in *Fig. 2*:

•legacy interfaces: the physical layer.

•programmable layer: a virtual switch that is compatible with SDN and an SDN controller.

•the OS layer consists of the operating system and its applications.

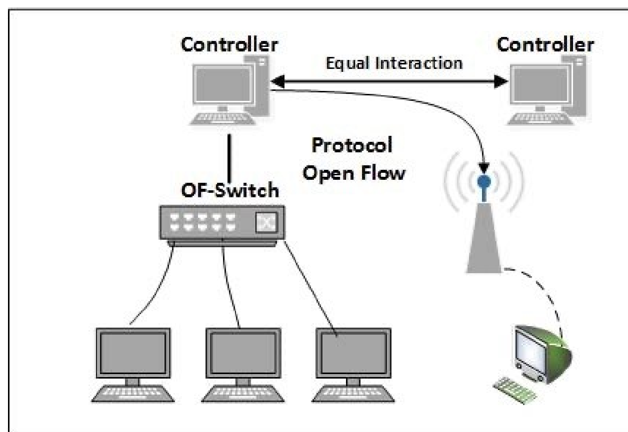


Figure 1: Equal Interaction SDN Networks

All legacy interfaces are linked to a virtual switch in this paradigm, which is managed by an SDN controller built within the node. Since each node's controllers interact with each other equally, there is no need to be concerned about the nodes' liability for the misbehaving users connecting through them. Ad-Hoc users will communicate with other nodes using an embedded switch that is SDN-compatible. The SDN controller can simultaneously improve connection and security across nodes through equal interaction. This new SDN-based Ad-Hoc network architecture's compatibility with SDN legacy networks is one of its benefits. The Ad-Hoc network may be connected to the legacy network to create an Extended SDN domain (Fig. 3), the node in the Ad-Hoc network has an embedded SDN-compatible switch and an SDN controller. Furthermore, all rules will

be synchronized because all controllers in the extended SDN domain interact equally. Ad-Hoc users must communicate with other nodes (Network gateways) that are linked to the SDN domain in this arrangement.

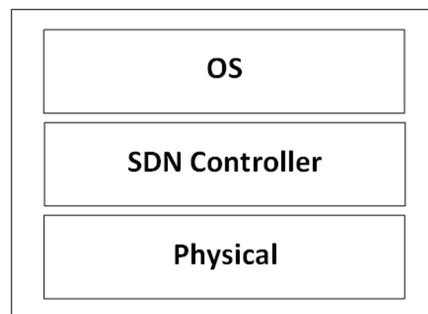


Figure 2: A Node in Ad-Hoc Plane

B. DISTRIBUTED AD-HOC CONTROL PLANE

The SDN control plane must oversee the evolution of each SDN virtual switch on each Ad-Hoc device because each Ad-Hoc node has its own SDN controller. Messages can be exchanged to synchronize all the rules whenever a new Ad-Hoc device joins or leaves the network. A distributed SDN design with several controllers is preferred to ensure scalability and fault tolerance. To ensure that, as shown in Figure 4, new controllers would be dynamically added to the Ad-Hoc network area, allowing specific nodes to conduct control operations. The new controllers will use the same global network view. Their capabilities and SDN control domain, however, will be constrained to a modest Ad-Hoc region. Since they are deployed on the users' side, those controllers will also be responsible for monitoring how the software switches behave.

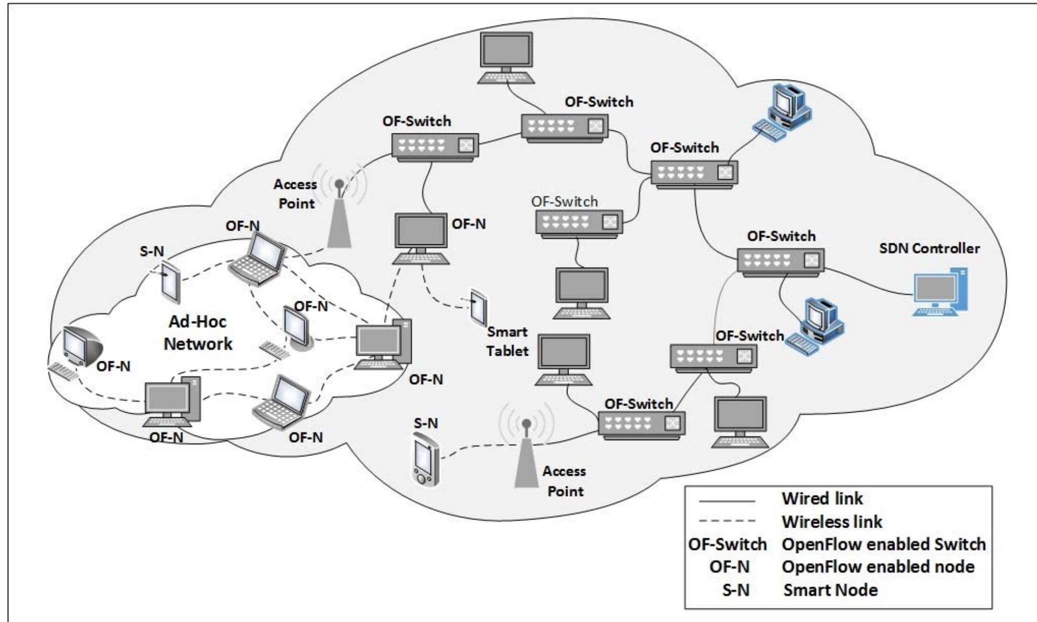


Figure 3: Extended SDN Domain

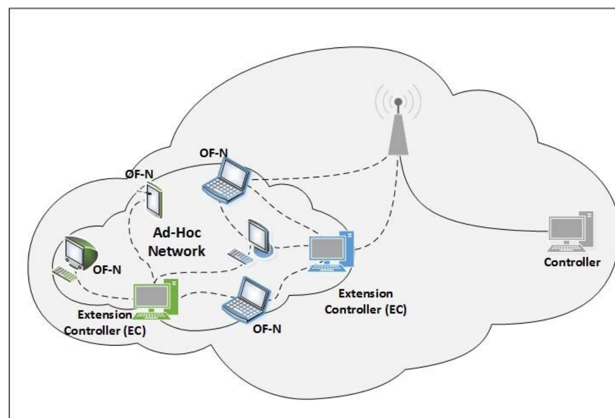


Figure 4: Distributed Ad-Hoc Control Plane

3.1.2. SDN-BASED IOT ARCHITECTURE

High levels of scalability, high volumes of data, and mobility are not supported by the standard network protocols and equipment.

A. SDN DOMAIN

As suggested in the previous section, each device in an IoT or sensor network cannot have an embedded SDN-compliant switch and an SDN controller. However, it can

be expected that each low-resource device can be connected to a neighboring node that has SDN capabilities. There are two different types of nodes in a domain in a heterogeneous network like the one in Figure 3. If a node has sufficient resources, it is called an OpenFlow node; otherwise, it is called a sensor or a smart object. The SDN controller for each domain manages all traffic within that domain.

B. SDN INTERCONNECTION DOMAIN

It is anticipated that each SDN domain in the proposed architecture has either one SDN controller or several SDN controllers. Each SDN domain in the proposed architecture has either one SDN controller or several SDN controllers. Only these controllers manage the devices within their domain. An enterprise data center is represented by a domain.

Many heterogeneous connections between SDN domains are necessary for an SDN-based architecture for the Internet of Things. A new form of controller, known as a root controller, also known as a border controller, must exist in each domain to enable the distribution of routing functions and security rules on each border controller to achieve such large-scale connectivity. Additionally, these controllers oversee the connection to other SDN border controllers and exchange data with them.

This design was developed with equitable interaction amongst controllers while utilizing the current security features. Each SDN domain has its management philosophy and set of security regulations. Due to the variety of security policies for the connected SDN domains, there may be issues that need to be resolved.

The Grid of Security idea put forth by Flauzac et al. (2015) is taken into consideration. A middleware for the decentralized enforcement of network security is called the Grid of Security.

3.1.3. DISTRIBUTED SDN SECURITY SOLUTION

In addition to network management, the controller effectively monitors and secures the network from both internal and external assaults.

By adding firewalls, IPS, and IDS modules to the SDN controller or by configuring security policies in OpenFlow switches, network security utilizing the SDN architecture is enabled. The development of the next generation of Internet architecture necessitates better security standards, including the need to authenticate network devices, users, and objects communicating to users via both wired and wireless technologies. Furthermore, it is necessary to monitor both user and object behavior to establish trust boundaries and apply accounting techniques and software verification. However, the security requirements of the next-generation Internet architecture are not met by the current security systems, which do not offer these security levels.

A straightforward solution offered by SDN architecture is for a controller to oversee security inside a single SDN domain. With the resources available on each control platform, this solution can potentially be expanded to accommodate numerous controllers. A secure model for the IoT network can be created by connecting all SDN domains via border controllers, which further extends the distributed control architecture.

Due to the lack of a network backbone, traditional Ad-Hoc architecture does not offer network access control or global traffic monitoring. These security restrictions are removed by the architecture suggested here, which also makes it possible to install dynamic security rules and configure networks.

According to Flauzac et al. (2015), this design attempts to achieve the maximum synchronization of SDN Controllers within a security perimeter, enabling fine-grained control over network access and ongoing monitoring of network endpoints.

The SDN controllers start by authenticating the network devices to secure network access and network resources. The SDN controllers start by authenticating the network devices to secure network access and resources. The controller restricts switch ports that are linked to users once the switch and controller establish a safe OpenFlow connection. After that, the controller authorizes only user authentication traffic after user authentication. Once the user has been validated, the controller will push the necessary flow entries to the software or hardware access switch depending on the user's level of authorization. The authentication method is expanded to include IoT network devices. Each device must connect to an OpenFlow-enabled node that is linked to a single controller inside its domain.

The grid of the security network is designed to expand the single SDN domain concept to several domains (Fig. 5), with each domain's controller exchanging security rules. On the periphery of the expanded SDN Domain, there are SDN controllers that act as security guards to protect the network Safety connections between domains, they could be provisioned and only added to SDN Controllers, and only recognized traffic could be accepted. The controllers are familiar with the rules in their domain, but they are unaware of the rules in other domains. Therefore, the flow needs to be routed to the Security Controller, also known as the Border Controller, whenever a node wants to communicate with another node in a different domain. According to Flauzac et al. (2015), the security controller queries each nearby controller to determine whether it is aware of the information's destination.

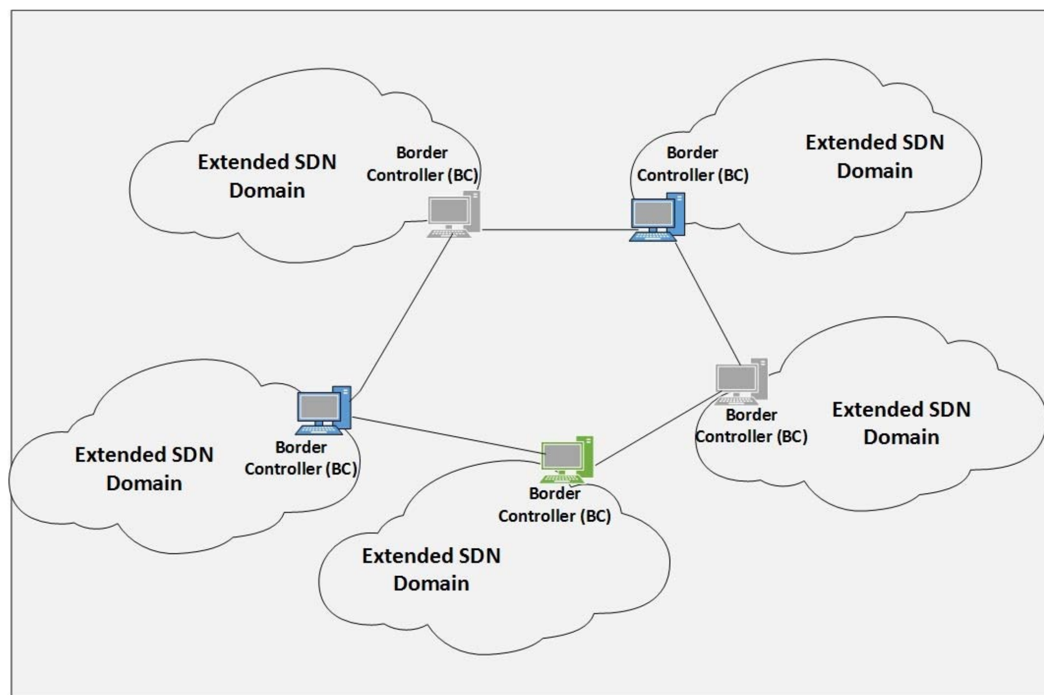


Figure 5: Grid of Security in SDN Domain

3.1.4 DRL-BASED INTRUSION DETECTION SYSTEMS

Security professionals typically need to observe and analyze audit data, such as application traces, network traffic flow, and user command data, to distinguish between

normal and abnormal behavior to discover intrusions. However, as the network capacity is increased, the volume of audit data quickly increases. Because of this, manual detection is challenging or even impossible. A software or hardware platform known as an intrusion detection system (IDS) is installed on host computers or network equipment to monitor audit data to identify and notify the administrator of any suspicious or malicious activity. A system for intrusion detection and prevention might be able to take the necessary steps right away to lessen the effects of malicious activity.

IDS are grouped into host-based and network-based groups based on several sorts of audit data. To identify unusual behaviors, host-based IDS often monitors and examines the host computer's log files or settings. A sniffer is used by network-based IDS to gather transmitting packets in the network and examines the traffic data for intrusion detection. Host-based systems normally lack cross-platform support and deploying them requires knowledge of the host operating systems and their configurations. Unlike host-based systems, network-based systems are more portable and attempt to monitor traffic over specified network segments. They are also independent of the operating systems. Thus, network-based systems are simpler to implement and have greater monitoring capabilities than host-based systems. Because it must inspect each packet that traverses its network segment, a network-based IDS could struggle to manage high-traffic and fast networks.

Regardless of the IDS types, there are two popular detection techniques, signature-based and anomaly-based detection. Signature detection entails the database of known attack patterns and the comparison of potential attack traits to those in the database. Anomaly detection monitors the system's typical behaviors and notifies the administrator if any occur, such as an unexpected rise in traffic rate, or the quantity of IP packets sent and received per second. Building adaptive IDSs has made extensive use of machine learning techniques, such as unsupervised clustering and supervised classification approaches. However, these techniques — such as neural networks, k-nearest neighbors, support vector machines (SVM), random forests, and more recently deep learning — typically rely on predefined properties of already-existing cyberattacks, making them ineffective at spotting brand-new or abnormal attacks. Unsupervised or supervised procedures produce ineffective solutions due to slow responses to dynamic

incursions. In this regard, numerous IDS applications have successfully used RL algorithms (Nguyen & Reddi, 2021).

The application of DRL techniques in host-based and network-based IDS is covered in the following subsections.

a. ***Host-based IDS***: Because they can only handle temporally isolated labeled or unlabeled data, adaptive intrusion detection models display limited performance as the volume of audit data and complexity of intrusion behaviors rise. Complicated invasions are made up of temporal sequences of dynamic actions. Thus, the intrusion detection problem is transformed into a Markov chain state value prediction task.
b. ***IDS on a network***: The disadvantages of anomaly-based and signature-based detection techniques are as follows: Traditional machine learning algorithms used for anomaly detection have a high false alarm rate, because they may classify routine user behavior as unusual. Since signature detection relies on a library of patterns from well-known attacks, it is unable to identify novel attack types. The suggested IDS is built using a combination of association rule learning, log correlation, and RL algorithms. When RL chooses to log files that include (or do not contain) anomalies or any indications of an attack, the system receives a reward (or a punishment). This process enables the system to pick log files that are more suitable while looking for signs of an attack. Dealing with the distributed denial-of-service (DDoS) threat, which is a DoS attack but has a dispersed character, occurs with high traffic volume, and compromises many hosts, is one of the most challenging tasks facing the modern Internet.

3.1.5 CYBER SECURITY ATTACKING SCENARIOS

Traditional cyber security techniques such as firewalls, anti-virus software, and intrusion detection typically fall behind dynamic threats and are passive and unilateral in nature. Because there are many different cyber components in cyberspace, reliable cyber security must consider how these components interact. Particularly, the choices made by

other components are influenced by the security policy applied to a component. As a result, when the system is huge, the decision space grows significantly and contains many what-if situations. Because it can explore a wide range of scenarios to determine the optimum course of action for every actor, game theory is effective in resolving such complex issues. Both its actions and those of other players influence a player's utility or return on investment. In other words, the effectiveness of cyber-defending measures must consider the tactics used by the attacker and the actions of other network users. Game theory can simulate the competition and collaboration of rational decision-makers, simulating the actions of cyber security issues including attackers and defenders. Game theory has been able to mathematically define and evaluate the complicated behavior of numerous competitive systems thanks to this similarity. The next section presents various cyber-security issues in various attack scenarios, such as jamming, spoofing, malware, and attacks in hostile environments (Nguyen & Reddi, 2021).

a) ***Jamming Attacks***: A DoS attack is any incident that impairs or destroys a network's ability to perform its intended purpose. Jamming attacks are a specific form of DoS attacks. Researchers have been interested in jamming, a severe attack on networking, and have employed machine learning, particularly RL, to solve this issue. The recent advancement of deep learning has made it easier to apply DRL for jamming handling or mitigation.
b) ***Spoofing Attacks***: These are common in wire-free networks where the attacker poses as another node and uses a fictitious identity, such as media access control, to gain unauthorized access to the network. Man-in-the-middle or denial-of-service attacks could result from this illicit intrusion.
c) ***Malware Attacks***: One of the most difficult types of malware for mobile devices is zero-day attacks, which prey on previously undiscovered security flaws. Until these attacks are stopped or neutralized, hackers may have already had negative impacts on the computer programs, data, or networks. The traces or log data generated by the applications must be processed in real-time to prevent such assaults. Mobile devices frequently transfer certain malware detection duties to secure servers in the cloud for processing due to their limited computing power, battery life, and radio bandwidth. The security server can process

activities more quickly and accurately and can deliver a detection report back to mobile devices with less lag time if it has robust computational resources and an updated malware database. As a result, the offloading procedure has a significant impact on the effectiveness of cloud-based virus detection. For instance, radio network congestion could cause a significant detection delay if too many tasks are transferred to the cloud server.

d) ***Attacks in Adversarial Environments***: Traditional networks enable direct communication between client applications and servers, but because each network has its switch control, reconfiguring the network is a slow and ineffective process. This method's additional drawback is that it could be necessary to employ various servers and databases to acquire the needed information. A next-generation networking solution that can adaptably alter the network is software-defined networking. SDN can efficiently manage and optimize network resources, since the control is programmable and has a broad perspective of the network architecture. RL has been widely shown to be a reliable strategy for managing SDNs. The attacker, if it is aware of the network control algorithm in an adversarial environment, may falsify the defender's training process notwithstanding RL's effectiveness in SDN controlling.

4. GENERAL OVERVIEW OF DEEP REINFORCEMENT LEARNING (DRL)

4.1. RL and DL (The Two Essential Building Blocks of DRL)

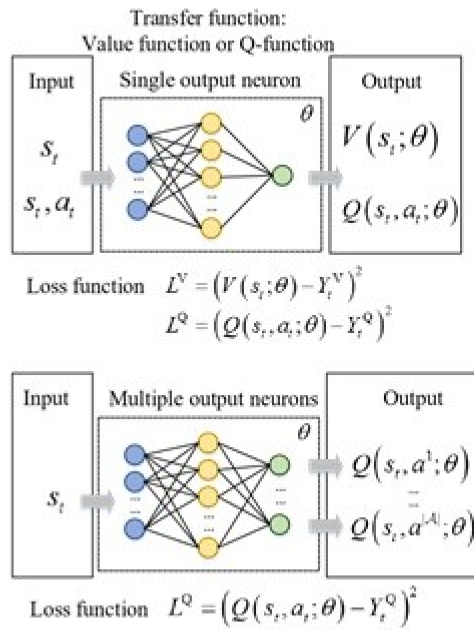
The value functions and Q-functions in RL typically require a substantial amount of memory. Since the state sets in most real-world issues are enormous and occasionally infinite, it is hard to store the value functions or Q-functions as tables. It is difficult to understand the environment's trial-and-error interaction due to the enormous computational complexity and storage needs. The application of DL allows various RL functions, such as value/Q-functions or policy functions, to be approximated with a more manageable set of parameters. The combination of RL and DL gives a powerful output – The DRL (Lei et al., 2020).

First, as shown in Fig. 2, fundamental DRL algorithms are divided into two major groups, value-based and policy gradient approaches, based on whether value/Q-functions or policy functions are approximated by NN. The policy gradient methodologies from three perspectives are as follows:

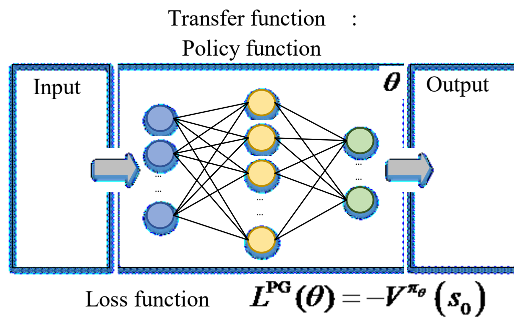
- introducing stochastic policy gradient (SPG) versus deterministic policy gradient (DPG) techniques based on the various characteristics of the approximated policy functions;
- as well as the Monte Carlo policy gradient against actor-critic approaches based on various methods of evaluating policies.
- the simple policy gradient versus natural policy gradient (NPG) methods are based on various learning or parameter updating strategies.

There are also two categories of advanced DRL algorithms, namely, the Partially Observable Markov Decision Process based DRL (POMDP-based DRL) and the Multi-Agent based DRL (MA-based DRL), which are expected to be very beneficial in resolving outstanding challenges, especially in self-adaptable IoT systems.

Figure 7 illustrates the basic DRL algorithms organization.



(a) General value-based methods for DRL



(b) General policy gradient methods for DRL

Figure 6: General methods of DRL

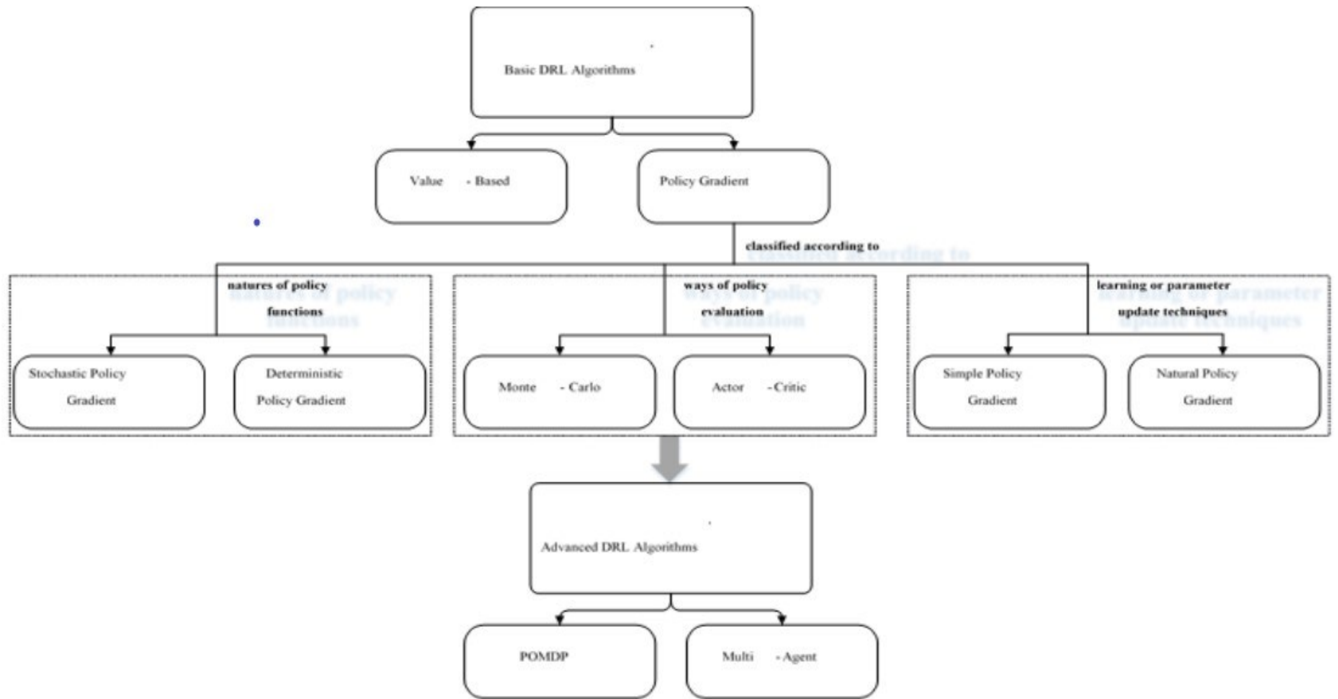


Figure 7: Basic DRL Algorithms Organization

4.1.1. Fundamental DRL Algorithm

1. ***Value-Based Methods***: In value-based methods for DRL, as shown in Fig. 6(a), the states $s_t \in$ S or state-action pairs $(s_t,a_t) \in$ S ×A are used as inputs to NNs, and Q-functions $Q^\pi(s_t,a_t)$ or value functions $V^\pi(s_t)$ are approximations of the input states provided by the NNs. For the input states or state-action pairs, a NN returns the approximations of the Q-functions or value functions. A single output neuron or several output neurons may exist, as shown in Fig. 6(a). The output for the first scenario can either be $V^\pi(s_t)$ or $Q^\pi(s_t,a_t)$ depending on the input s_t or (s_t,a_t). The Q-functions for state st combined with each action, i.e., $Q^\pi(s_t,a^1),\cdots,\ Q^\pi(s_t,a^{|A|})$, are the outputs for the latter case.

   Y_t^Q and Y_t^V are the goal values of the Q-functions and value functions, respectively, from which the loss functions are derived. When using value-based approaches, the regression loss can be used to assess how well the NN approximates Q-functions or value functions (Lei et al., 2020).

$$L^{Q} = \left(Q\left(s_t, a_t; \theta\right) - Y_t^{Q}\right)^2$$

Or

$$L^{V} = \left(V\left(s_t; \theta\right) - Y_t^{V}\right)^2$$

a. ***Deep Q-Networks***: This is based on the concept of NN fitting Q-functions. In Fig. 8(a), the DQN illustration is displayed. The NN in the DQN algorithm receives a state as input and outputs approximations of Q-functions for each action contained in the state.
In DQN, the method first initializes the network parameters at random with the value θ_0. The target Q-function Y_t^{DQN} is given under the Bellman equation as:

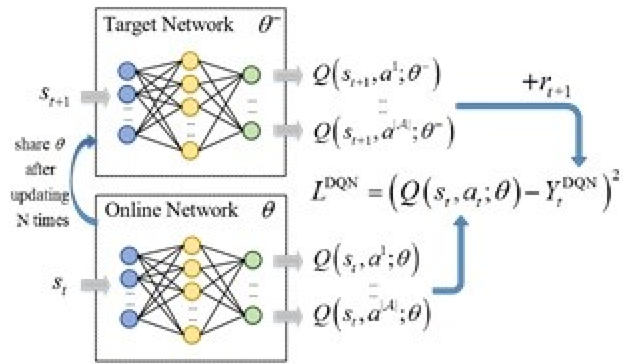
$$Y_t^{DQN} = r_{t+1} + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; \theta\right)$$

Where the variables' values at the tth or $(t + 1)^{th}$ iteration are indicated by the subscripts t or $t + 1$. By substituting YtDQN for YtQ, the loss function L^{DQN}, which can be derived from the first equation in the regression loss formula above, is minimized to update the parameters in DQN.

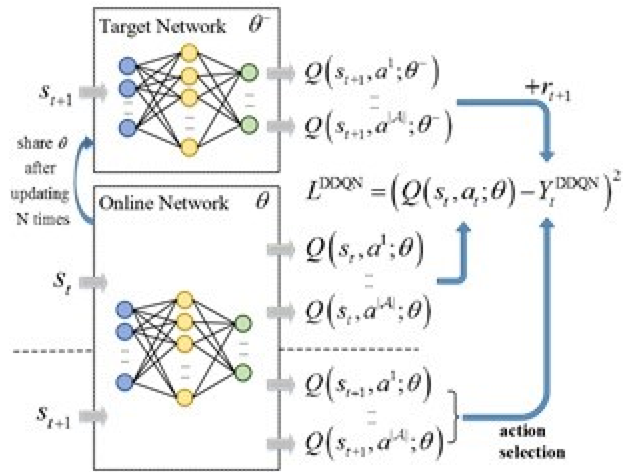
By applying stochastic gradient descent, the parameters are updated as;

$$\theta \leftarrow \theta + \alpha \left(Y_t - Q\left(s_t, a_t; \theta\right)\right) \nabla_\theta Q\left(s_t, a_t; \theta\right)$$

Where α is the learning rate.



(a) DQN



(b) DDQN

Figure 8: DQN and DDQN with Target Networks

Two key strategies —freezing target networks and experience replay— are used in DQN to address the limitations of DRL. The target networks, whose parameters θ_t^- are kept constant over time, are used to assess the Q-function of the subsequent state to increase the stability and controllability of the training process, such that instead of this Y_t^{DQN} in the given equation above, we have this below;

$$Y_t^{DQN} = r_{t+1} + \gamma \max_{a_{t+1}} Q\left(s_{t+1}, a_{t+1}; \theta^-\right)$$

After each cycle, the settings of the online network θ_t are changed. The online network communicates its parameters with the target network after a predetermined number of iterations. By doing this, the likelihood of divergence is decreased, and instability brought on by overly rapid propagation is avoided.

The experience of the agent at each time step is saved in a data set to perform experience replay. When updates are applied to this data set, correlations in the order of observations are eliminated and variations in the data distribution are smoothed. With this method, there is more opportunity to change the parameters in a wider way and the updates can cover a larger state-action space.

b. ***Double DQN***: This method likely overestimates an action's Q-function because it uses the Q-function assessed by target networks for both selection and evaluation. More actions will result in a higher estimate error. The double DQN (DDQN) method is used to address this issue, using two sets of parameters to obtain the goal value YtDDQN as illustrated in Fig. 8(b). The target Q-value in DDQN can be rewritten as.

$$Y_t^{DDQN} = r_{t+1} + \gamma Q(s_{t+1}, \underset{a_{t+1}}{\operatorname{argmax}} Q(s_{t+1}, a_{t+1}; \theta); \theta^-),$$

Where the parameters in the online network determine which action is chosen, and the parameters in the target network determine how well the present action is being evaluated. The performance of the DRL methods will therefore be improved because there will be less overestimation of the Q-Learning values and more stability. By changing Y_t^Q to Y_t^{DDQN} and updating the parameters accordingly, the loss function L^{DDQN} may be obtained from the first equation regression loss. While maintaining the majority of the DQN algorithm, the DDQN algorithm benefits from double Q-Learning. In addition to DQN and DDQN, there are alternative value-based approaches. Some of these methods, such as DDQN with Proportional Prioritization and DDQN with duel architecture, were developed based on DQN and DDQN with additional improvements.

(Value-based DRL methods: advantages and disadvantages) Although the value-based DRL approaches have significant shortcomings, the DQN and its upgraded versions have

been frequently used in the existing literature. This is mainly because of their relative simplicity and good performance. First, it is unable to resolve RL issues with expansive or continuous action spaces. Second, it is unable to resolve RL issues where the best course of action involves stochastic probabilities. Since value-based methods can only learn deterministic policies, most algorithms—including DQN—are off-policy.

2. ***Policy Gradient Methods***: When the environment is in state s, action a is chosen by a policy π. NNs can be used in policy gradient methods to directly approximate a policy as a function of the state, i.e., $\pi_\theta(s)$. As seen in Fig. 6(b), the states serve as the NNs' inputs, and policy π is roughly represented by the NNs' parameters θ of NNs as π_θ.
   To evaluate the performance of the existing policy, the objective function is described below:

$$J(\theta) = V^{\pi\theta}(s_0) = \mathrm{E}_{\tau s}0{\sim}_{\pi\theta}\,[G(\tau_{s0})], \forall s_0 \in \mathrm{S}$$

Where τ_{s0} denotes the sampling trajectory with an initial state s_0 and $V^{\pi\theta}(s_0)$ denotes the value function of policy π_θ. If we can determine the parameters θ for the policy π_θ so that the objective function $J(\theta)$ is maximized, then, we can resolve this issue. The basic tenet of policy gradient methods is to adjust the parameters in the direction of a higher expected reward. For this purpose, we can change the loss function of NN to be:

$$L^{\mathrm{PG}}(\theta) = -J(\theta) = -V^{\pi\theta}(s_0)$$

We must express the gradient of $J(\theta)$ with respect to the parameter as an expectation of stochastic estimates to update the parameters. The policy in RL can be divided into two types, namely the stochastic policy and the deterministic policy. As a result, the SPG method and DPG method are detailed below (Lei et al., 2020):

a. ***Comparing stochastic policy gradients to deterministic policy gradients***, DRL may be used to approximate a stochastic policy $\pi_\theta = \pi(a_t|s_t;\theta)$, which provides the

probability that an agent would conduct a particular action in an exact state when the policy specified by θ is followed. The weights and biases of a NN are frequently used as the policy parameters. A typical probability density function is the Softmax function for a DRL model with discrete state/action spaces. The policy is often described using a Gaussian distribution when the state and action spaces are continuous. The Gaussian distribution's standard deviation is specified by a set of parameters, and a NN is used to approximate the mean.
The policy gradient theorem is as follows:

$$\nabla \mathrm{E}_{\tau s 0 \sim \pi\theta}[G(\tau_{s0})] = \mathrm{E}_{\tau s t \sim \pi\theta}[G(\tau_{st}) \nabla_\theta \log \pi(a_t|s_t;\theta)]$$

Application of stochastic gradient decent results in the parameters being updated as:

$$\theta \leftarrow \theta + \alpha G(\tau_{st}) \nabla_\theta \log \pi(a_t|s_t;\theta)$$

Where the learning rate represents α. In this way, θ is changed to increase the probability that the trajectory $G(\tau_{st})$ will have a greater total reward.

From the perspective of NN, the loss function of the SPG algorithm is given below:

$$L^{\mathrm{SPG}}(\theta) = -G(\tau_{st}) \log \pi(a_t|s_t;\theta)$$

DPG represents the policy as a deterministic decision, i.e., $\pi_\theta = \pi(s_t;\theta)$, in contrast to SPG where the policy is described as a probability distribution over actions. We have:

$$\nabla \theta J(\theta) = \mathrm{E} s \sim \rho \pi \theta [\nabla \theta \pi(st;\theta) \nabla a Q \pi \theta(st,at;\varphi)|a=\pi(st;\theta)]$$

Where the policy improvement is divided into the gradient of the Q-function to actions and the gradient of the policy to the policy parameters. This is based on the

objective function provided in the 7th equation above and the DPG theorem. $\rho^{\pi\theta}$ Is the state distribution carried out according to policy π_{θ}. As a result, the parameters are changed as follows:

$$\theta \leftarrow \theta + \alpha h \nabla \theta \pi(st;\theta) \nabla a Q \pi \theta(st,at;\varphi)|a=\pi(st;\theta) i$$

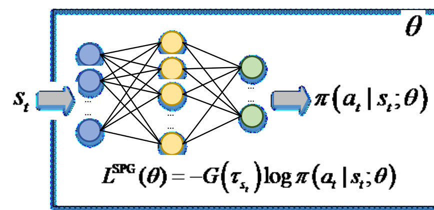
The gradient $\nabla_a Q^{\pi\theta}(s_t,a_t;\varphi)$can be changed to $\nabla_a Q^w(s_t,a_t;\varphi)$ by using a differentiable function $Q^w(s_t,a_t;\varphi)$ as an approximator of $Q^{\pi\theta}(s_t,a_t;\varphi)$. The approximator achieves $\nabla_a Q^w(s_t,a_t;\varphi)|_{a=\pi(st;\theta)}$ as $\nabla_\theta \pi(s_t;\theta)^{>} w$, which is consistent with the deterministic policy. From NN's vantage point, the DPG loss function is set to:

$$L\mathrm{DPG}(\theta) = -\pi(st;\theta) \nabla a Q \pi \theta(st,at;\varphi)|a=\pi(st;\theta)$$

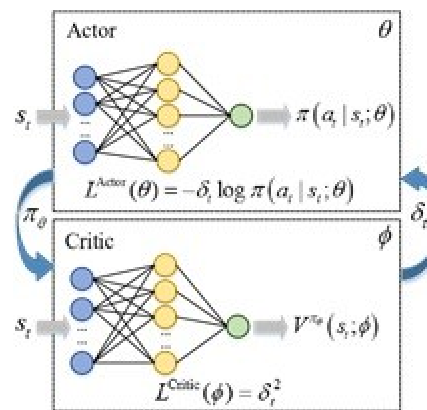
b. ***Comparing Monte Carlo Policy Gradient to Actor-Critic***, the values of $G(\tau_{st})$ and $\nabla_a Q^{\pi\theta}$ must be determined in the 11th and the 14th equation above respectively, to update the policy parameters θ in SPG and DPG. As shown in Figs. 9(a) and 9(b), respectively, this may be done for SPG by either using the actor-critic method or the Monte Carlo policy gradient method. This is typically accomplished with DPG using the actor-critic method, as shown in Fig. 9(c).
The Monte Carlo policy gradient method uses Monte Carlo simulation to attempt to quantify $G(\tau_{st})$. The REINFORCE algorithm is a typical Monte Carlo algorithm for SPG techniques. According to the Monte Carlo method, the current policy is initially sampled from an initial state s_0 by running the trajectory τ_{s0}. The total reward $G(\tau_{st})$ starting from time step t is then calculated for each time step t = 0, 1, …., T. This value is multiplied by the policy gradient $\nabla_\theta \log\pi(a_t|s_t;\theta)$ to update the parameters following the equation 10. The aforementioned process is performed several times, sampling a different trajectory in each run.
Moreover, a baseline function $b(s_t)$ that is independent of a_t is added to lessen the variance of the policy gradient. This leads to the introduction of the REINFORCE algorithm (Williams, n.d.) with baseline, whose loss function can be written as:

$$L^{\text{SPG BASE}}(\theta) = -(G(\tau_{st}) - b(s_t))\log\pi(a_t|s_t;\theta)$$

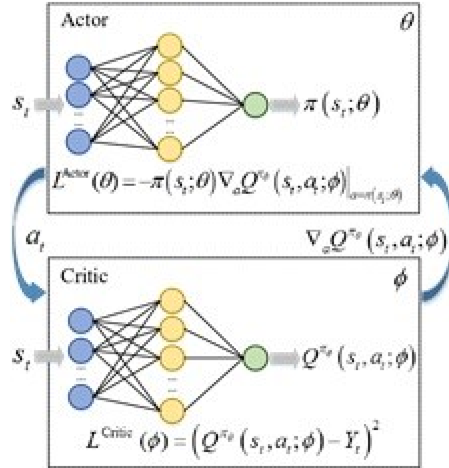
(Monte Carlo policy gradient DRL methods: advantages and disadvantages) The policy gradient approaches for DRL are a direct mapping from state to action as opposed to value-based DRL methods, which results in superior convergence qualities and improved efficiency in high-dimensional or continuous action spaces. Additionally, it can learn stochastic policies, which in some circumstances perform better than deterministic policies. However, the huge estimating variance makes Monte Carlo policy gradient approaches suffer. They are particularly sample-intensive since they are on-policy approaches that call for on-policy samples.



(a) Monte Carlo Policy Gradient



(b) Actor-critic methods for SPG.



(c) Actor-critic methods for DPG.

Figure 9: Monte Carlo Policy Gradient and Actor-Critics Method

Actor-critic approaches have received a lot of attention in DRL because they combine the benefits of value-based and Monte Carlo policy gradient methodologies. An actor-critic technique is often implemented by two NNs, i.e., the actor-network and the critic network, which share parameters, as shown in Figs. 9(b) and 9(c). The critic network is comparable to the NN of the value-based method, whereas the actor network is comparable to the NN of the policy gradient method. During the learning process, the critic modifies the value function parameters, i.e., φ, according to the actor given by the policy. In the meantime, the actor modifies the policy's parameters θ, or, according to the value functions, the critic has evaluated. Typically, two learning rates must be established respectively for the updates of φ and θ.
The critic network is utilized to determine the value of $G(\tau_{st})$ in equation 11 in the actor-critic technique for SPG. The value function $V^{\pi\theta}(s_t;\varphi)$; is specifically set as the baseline $b(s_t)$ in equation 15, and it is approximated by the critic network using the loss function shown in equation 2. When an agent chooses an action in a state s_t following the current policy π_θ as provided by the actor-network, they are rewarded with r_{t+1}, and the state then transits to s_{t+1}. The loss function for the critic network can be represented as equation 2 in a value-based manner as below:

$$L\text{Critic}(\varphi) = \delta t2$$

$$\text{where } \delta t = rt{+}1 + \gamma V\,\pi\theta\,(st{+}1;\varphi) - V\,\pi\theta(st;\varphi)$$

The parameters of the critic network are updated as described in equation 4 in DQN.

$$\varphi \leftarrow \varphi + \beta\delta_t\nabla_\varphi V^{\pi\theta}(s_t;\varphi)$$

Where the critic's rate of learning is β.

Note that, $G(\tau_{st})$ is a rough approximation of $Q^{\pi\theta}(s_t,a_t;\varphi) = r_{t+1} + \gamma V^{\pi\theta}(s_{t+1};\varphi)$. As a result, the value of $G(\tau_{st})$ - $b(s_t)$ in equation 15 can be substituted by δ_t in equation 17, which can be thought of as an estimate of the benefit of action a_t in state s_t given the value functions evaluated by the critic network. The actor network's loss function can be described in a manner similar to equation 15, i.e.,

$$L^{\text{Actor}}(\theta) = -\delta_t \log\pi\,(a_t|s_t;\theta$$

The parameters of the actor network are updated as described in equation 10 in the Policy Gradient method.

$$\theta \leftarrow \theta + \alpha\delta_t\nabla_\theta \log\pi(a_t|s_t;\theta)$$

where the actor's learning rate is α.

The critic can correctly approximate value functions through the actor-critic algorithm's updating procedures, and the actor can make better decisions to earn bigger rewards.

The asynchronous advantage actor-critic (A3C) algorithm and soft actor-critic (SAC) are two common actor-critic techniques for SPG. The former is primarily concerned with simultaneously training numerous actors who share global parameters. The latter includes off-policy updates, a tractable stochastic policy, and a soft Q-function. SAC performs well across a variety of continuous control tasks.

The Deep Deterministic Policy Gradient (DDPG) algorithm is a common actor-critic approach for DPG. The DDPG algorithm combines the concepts of DPG and DQN. It is a model-free off-policy actor-critic method. The target networks Q^0 and π^0 in the DDPG algorithm are specified by φ^0 and θ^0 respectively, in addition to the online critic network Q and the online actor network π with parameters φ *and* θ respectively. These four NNs' parameters must be changed throughout the learning process. The critic network is used to obtain the gradient $\nabla_a Q^{\pi\theta}(s_t, a_t; \varphi)$.
(Actor-Critical DRL Methods: Pros and Cons): The benefits of value-based and Monte Carlo policy gradient methods are combined in actor-critic methods. They may comply with the policy or not. They require far fewer learning samples and processing resources to choose an action than Monte Carlo approaches, especially when the action space is continuous. They can learn stochastic policies and handle RL problems with continuous actions, in contrast to value-based techniques. However, because of the recursive use of value estimate, it is vulnerable to instability. According to the description above, the policy gradient in the actor-critic approach is deterministic but biased, whereas the policy gradient in the Monte Carlo method is unbiased but has a high variation. Therefore, combining these two approaches would result in creating a successful strategy. It would address the issue of high sample complexity while combining the benefits of on-policy and off-policy approaches.

c. ***Comparing Simple Policy Gradient to Natural Policy Gradient:*** The policy gradient methods discussed above, update the NN parameters using a simple gradient of the loss function $\nabla_\theta L(\theta)$. To give a more effective solution, the NPG method updates the parameters in NN using the natural gradient $\nabla^N{}_\theta L(\theta)$, as opposed to a simple gradient. The loss function of NPG and SPG, whose general expression is given in equation 8, are identical. The variables are modified as:

$$\theta \leftarrow \theta + F_{\theta}^{-1} \nabla_{\theta} V^{\pi_{\theta}}(s)$$

Where,

$$F_{\theta} = E_{\pi\theta} h \nabla_{\theta} \log\pi (a_t|s_t;\theta)(\nabla_{\theta} \log\pi (a_t|s_t;\theta))^{Ti}$$

is utilized to calculate the update step size using the Fisher information matrix.
The NPG technique introduces a new step size definition that specifies the amount by which certain parameters should be modified, resulting in a more reliable and efficient update. However, the problem with NPG is that it is impossible to calculate the Fisher information matrix or store them effectively when sophisticated NN is used to estimate the policy if the number of parameters is big. The aforementioned issue is somewhat resolved by methods derived from NPG, such as Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO), which are frequently employed in practice for DRL. Additionally, some algorithms include NPG in actor-critic methods.

4.1.2. ADVANCED DRL ALGORITHM

A. ***POMDP-based DRL (A Partially Observable Markov Decision Process):***

Considering RL in a Markovian environment indicates that optimal control is always possible with only knowledge of the present state. The agent cannot accurately view the entire environment in many real-world issues, typically because of sensing and communication capabilities that are constrained. An agent operating in a partially observable environment can model the environment as a POMDP. To perform RL tasks in realistic contexts, it is necessary to deal with the noisy and partial state information produced by POMDP.

By including a limited number of observations and an associated observation model, POMDP can be considered an extension of MDP. Typically, a POMDP is described as six-tuple $< S, A, P, r, \Omega, O >$

Where the elements of the MDP state space S, action space A, transition probability P, and reward r have already been established

- Ω is the observation space, where $o \in \Omega$ is a possible observation.
- O $(o|s^0,a)$ is the conditional probability that taking an action a leading to a new state s^0 will result in an observation o.

Similar to MDP, an agent selects an action $a \in$ A under the policy $\pi(a|s)$, which causes the environment to change to a new state $s^0 \in$ S with probability $P(s^0|s,a)$, in exchange, the agent receives a reward $r(s,a)$. In contrast to MDP, the agent cannot directly see system states; instead, it receives an observation $o \in \Omega$ whose probability $\mathrm{O}(o|s^0,a)$. relies on the new state of the environment. The policy and Q-function are also changed to $\pi(a|o)$ and $Q(o,a)$ respectively.

The agent must use historical data to lessen ambiguity regarding the current state because it cannot directly view the underlying condition. The definition of the observation history at time step t is $h_t = \{(o_1,a_1),\cdots,(o_{t-1},a_{t-1}),(o_t,-)\}$ (Lei et al., 2020).

B. ***MA-DRL (Multi-Agent DRL):*** In the previous discussions, we focused primarily on single-agent DRL methods. In real-world applications, there are times when many agents must cooperate, such as during cooperative vehicle driving or manipulation in multi-robot systems. DRL approaches for Multiple-Agent (MA) systems are created in these circumstances.

An MA system has a high level of robustness and scalability and is composed of a collection of autonomous, interacting agents that share a common environment. The notion of a stochastic game is established to expand MDP into the MA context since the various agents in the system can interact with each other in cooperative or competitive situations. The definition of a stochastic game or MA-MDP with N agents is given by the tuple $< \mathrm{S}, \mathrm{A}_1, \cdots, \mathrm{A}_N, P, r_1, \cdots, r_N >$, where

- S is the discrete set of states,
- A_i, $i = 1, \cdots, N$ are the discrete sets of actions available to the agents, yielding the joint action set $\mathrm{A} = \mathrm{A}_1 \times \cdots \times \mathrm{A}_N$,
- $P : \mathrm{S} \times \mathrm{A} \times \mathrm{S} \rightarrow [0,1]$ is the state transition probability function,

- $r_i : S \times A \rightarrow R$, $i = 1, \cdots, N$ are the reward functions for the agents.

The joint action of all the agents $a = [a_1, \cdots, a_n]$, where $a \in A$ and $a_i \in A_i$, determine the state transitions in MA-MDP. All agents in the completely collaborative issues receive the same reward, or $r_1 = \cdots = r_N$. With $r_1 + \cdots + r_N = 0$, the agents in fully competitive issues receive opposing rewards. As a result, in the common case involving two agents, $r_1 = -r_2$. Mixed games are MA-MDP challenges that are neither very collaborative nor fully competitive.

Each agent in MA RL learns to enhance its policy by interacting with the outside world to gain rewards. The environment is typically complicated and dynamic for each agent, and the system may run into the action-space explosion problem. Since several agents are learning simultaneously, for a certain agent, the best course of action for one agent may also alter as other agents' policies change. This can disrupt the learning algorithm's convergence and lead to instability (Lei et al., 2020).

5. IMPLEMENTATION

In recent years, IoT security has been compromised. Security breaches could result in data loss and/or unethical manipulations of data, which could result in anomalies in the IoT network. To curb this, security measures are introduced to IoT networks, which make them self-reliant.

For simplicity's sake, the environment in this context, which is the IoT network, is simplified. Reinforcement learning is a type of machine learning that allows an agent (in this case, the network) to learn from experience by receiving feedback in the form of rewards or penalties. The agent tries to maximize its rewards by taking actions that are most likely to lead to positive outcomes. The reinforcement-learning agent learned to adjust network parameters such as bandwidth allocation, routing paths, or transmission power levels based on feedback from the network environment. Thus, it learned to prioritize traffic to a device experiencing connectivity issues or adjust the network topology to improve connectivity.

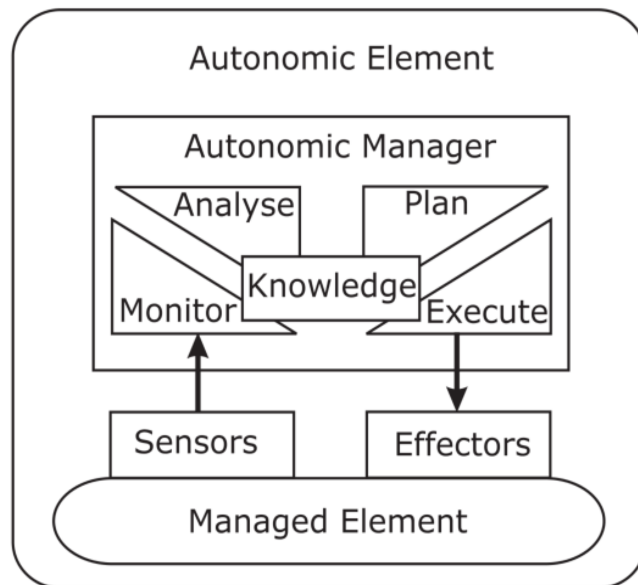


Figure 10: MAPE-K Loop; Self-Adaptation Control Feedback Loop (Lalanda et al., 2013)

Problem Statement**:* This paper focuses on the effectiveness of using dynamic programming (reinforcement learning***) to train an SDN-based IoT network to detect anomalies, correct them, and thereby, improving fault-tolerance properties in the IoT network.

We leveraged the concept of ***MAPE-K Loop*** in *Fig. 10* - the well-known self-adaptation control loop. There are four main attributes to the MAPE-K loop: these attributes help to monitor the system and its environment by analyzing the information collected from the system and its environment, they also decide and plan for the system's adaptation, thereby, executing the adaptation by applying it to the system.

The agent (in our case, the RL Agent), which is the autonomic manager in *Fig. 10*, implements autonomic loop(s): collects/decides/acts control loop, makes use of monitored data, and combines this with its internal knowledge of the system to plan and implement management tasks (Lalanda et al., 2013). The agent analyzes the IoT network environment, searches for anomalies, and tries to solve them by using the episodic settings in the Markov Decision Process discussed in this chapter. The type of reinforcement learning used here is MDPs where the probabilities or rewards are unknown. For this purpose, it is useful to define a further function, which corresponds to taking the action and then continuing optimally (or according to whatever policy one currently has):

$$Q(s, a) = \sum P_a(s, s')(R_a(s, s') + YV(s'))$$

While this function is also unknown, experience during learning is based on ***(s; a)*** pairs (together with the outcome ***s'***; that is, "I was in state ***s*** and I tried doing ***a*** and ***s'*** happened"). Thus, one has an array and uses experience to update it directly. This is known as Q-learning (*Markov Decision Process*, n.d.).

5.1. IoT Network Architecture

The IoT network architecture we considered for our solution was the general IoT network architecture. The general IoT architecture has the universal IoT features; the Sensing Domain (Perception layer: consisting of the sensors, gadgets, and other devices), the Network Domain (Network layer: connectivity between devices), and the Application Domain (Application layer: the layer the user interacts with); an example is shown in *Fig. 11*. These layers support IoT devices through data collection and processing. This architecture goes beyond the OSI model (The Open Systems Interconnection (OSI) model describes the seven layers that computer systems use to communicate over a network. It has seven layers – from the physical layer to the

application layer) to include the transformation of data into usable information using machine learning and artificial intelligence.

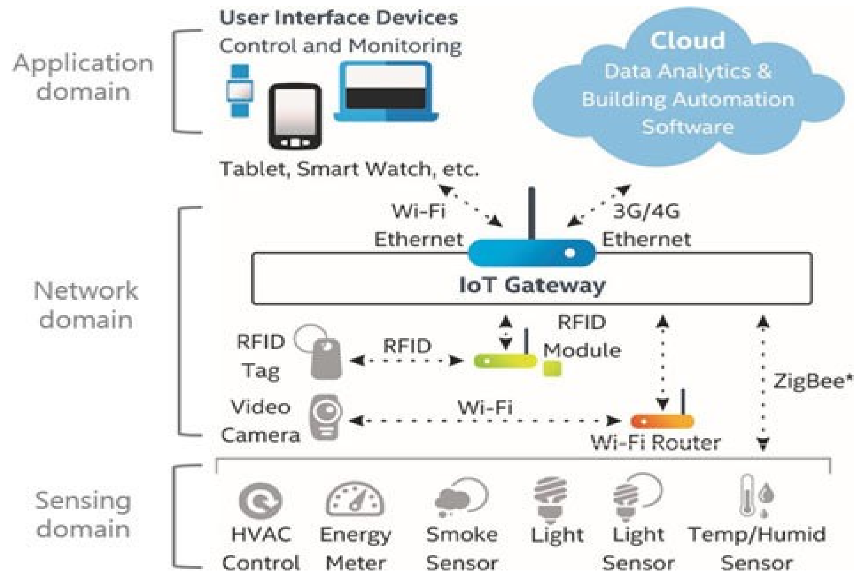


Figure 11: General Feature of IoT Gateway (Kang et al., 2017)

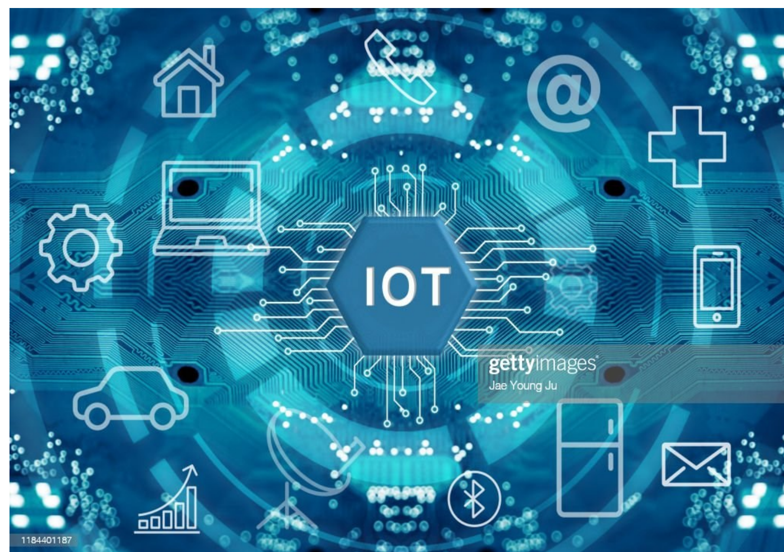


Figure 12: IoT Network from gettyimages

5.2. Markov Decision Process

The interactions between the agent and the environment in this context are a finite horizon with discrete state spaces. The Markov Decision Process (MDP) M = ***(S; A; P; r; ϒ; μ)***, is specified by:

• A state space **S**, which is finite in our case, is assumed a network of IoT devices.

• An action space ***A***, which is discrete.

• A transition function ***P: S X A*** ⟶ Δ***(S)***, where Δ***(S)*** is the space of probability distributions over ***S*** (i.e., the probability simplex). ***P(s' |s; a)*** is the probability of transitioning into state s_o upon taking action ***a*** in state ***s***.

We use ***P s; a*** to denote the vector ***P(|s; a)***.

• A reward function ***r : S X A*** ⟶[0; 1]. ***r(s; a)*** is the immediate reward associated with taking action ***a*** in state ***s***. More generally, the ***r(s; a)*** could be a random variable (where the distribution depends on ***s; a***). While we largely focus on the case where ***r(s; a)*** is deterministic, the extension to methods with stochastic rewards are often straightforward.

• A discount factor ***ϒ ϵ [0; 1)***, which defines a horizon for the problem.

• An initial state distribution ***μ ϵ*** Δ***(S)***, which specifies how the initial state s_o is generated. In many cases, we will assume that the initial state is fixed at s_o, i.e. ***μ*** is a distribution supported only on s_o (Agarwal et al., n.d.).

5.3. Model-free Solution Technique

The solution technique we employed in our work is a model-free policy-iteration technique that uses the standard MDP (M) = ***(S; A; P; r; ϒ; μ)***, where the state and action sets are finite and discrete. Furthermore, transition, reward, and value functions are assumed to store values for all states and actions separately. The core Dynamic Programming (DP) algorithm considered here is the Policy Iteration method, although, some value iteration was also done to calculate all possible rewards, and convergence at the optimal value was achieved. As a model-free learning algorithm, the agent learns directly from the environment. The approach employed for solving the environment is the Monte Carlo approach.

First-visit MC policy evaluation (returns $V \approx v_\pi$)

Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    Generate an episode using π
    For each state s appearing in the episode:
        $G \leftarrow$ return following the first occurrence of s
        Append G to $Returns(s)$
        $V(s) \leftarrow \text{average}(Returns(s))$

Equation 1: Monte Carlo Policy Evaluation

The Model-free Monte Carlo Approach is where the agent learns value functions directly from episodes of experience and only receives the reward at the end of an episode. The agent learns value functions from sample returns from episodes in the Markov Decision Process (MDP). Sample returns mean average rewards from episodes, where Episode = [(S1, A1, R1), (S2, A2, R2), (S3, A3, R3), and so on.. till the termination state]. An example is shown in the code snippet below:

```
# Calculating the reward for an episode of 10

episodes = 10
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = env.action_space.sample()
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
env.close()
```

Equation 2: Original Code Snippet

5.3.1. Training with PPO Algorithm

A stochastic policy is trained using PPO in an on-policy manner. This indicates that it samples activities during exploration following the most recent iteration of its stochastic policy. Both the training process and the initial conditions influence action selection's level of randomness. As the update rule encourages the policy to take advantage of rewards that it has already discovered, the policy normally becomes increasingly less random throughout the course of training. The policy could become stuck in local optima because of this.

An on-policy model that enables the agent to learn how to maximize its reward is the PPO policy model. Using the data that is currently available, improves a policy as much as it can without unintentionally causing performance to collapse. Similar to the TRPO, the PPO is a family of first-order algorithms that use a few heuristics to keep new policies near to the old. The PPO approaches are far easier to implement than the TRPO, which attempts to solve this problem via a complicated second-order algorithm.

When using a stochastic gradient ascent technique, policy gradient approaches compute an estimator of the policy gradient. The most popular gradient estimator has the following structure:

$$\hat{g} = \hat{E}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at time step t. Here, the expectation $\hat{E}_t[\ldots]$ indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization. Implementations that use automatic differentiation software work by constructing an objective function whose gradient is the policy gradient estimator; the estimator $\hat{g}$ is obtained by differentiating the objective:

$$L(\theta) = \hat{E}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

While using the same trajectory to perform multiple steps of optimization on this loss L may

seem appealing, doing so is not well-justified and frequently results in destructively large policy updates, according to empirical evidence (Schulman et al., 2017).

Our RL agent was trained and validated to detect anomalies in the network and immediately apply some fixes. The model was trained using the MlpPolicy policy model, which is one of the policy models in the Proximal Policy Optimization (PPO) algorithm of Stable Baselines3 that maps the action space to the state space in an environment. This work experiments on a Simplified Environment, i.e., an environment where simple state spaces of an IoT network are assumed to be free of anomalies. The agent's action spaces in this environment are discrete and the observation spaces are box. Some random anomalies were introduced into the environment state spaces for the agent to learn and act on, given some predefined actions in the agent's action spaces to solve the environment.

In the episodic settings, we had 10 iterations. The agent was able to map the action space with the state space to obtain the best action for the environment, which was at its initial state (The initial state was assumed to be anomaly-free). The agent was able to learn to carry out some pre-defined actions to correct the introduced anomalies, thereby, maximizing its reward.

The major drawback of this kind of policy is that as an on-policy model, the agent is only trained to make rigid tasks. It does not have the flexibility characteristic or feature to make future tasks possible (i.e., it is not flexible enough to handle other tasks that it is not trained for).

5.3.2. Policy Iteration Method

Policy iteration is a model-based reinforcement learning that iterates between the ***policy evaluation*** **phase** (which computes the value function of the current policy) and the ***policy improvement phase*** (which computes an improved policy by a maximization over the value function). This iteration is repeated until it converges to an optimal policy. Policy iteration starts with an arbitrarily initialized policy π_o. Then a sequence of iterations follows in which the current policy is evaluated, after which it is improved. The first step, the ***policy evaluation*** step, computes V^πk. The second step, the ***policy improvement*** step, computes πk+1 from πk using V^πk. For each state, the optimal action is determined. If for all states s, πk+1(s) = πk(s), the

policy is *stable,* then the policy iteration algorithm can stop. Policy iteration generates a sequence of alternating policies and value functions.

For finite MDPs, i.e., state and action spaces are finite policy iteration converges after a finite number of iterations. Each policy $\pi k+1$ is a strictly better policy than πk unless in case $\pi k = \pi*$, in which case the algorithm stops. Moreover, because for a finite MDP, the number of different policies is finite, policy iteration converges in finite time. In practice, it usually converges after a small number of iterations. Although policy iteration computes the optimal policy for a given MDP in finite time, it is relatively inefficient. In particular, the first step, the policy evaluation step, is computationally expensive. Value functions for all intermediate policies $\pi_o,\ldots,\pi k,\ldots,\pi*$ are computed, which involves multiple sweeps through the complete state space per iteration. A bound on the number of iterations is not known and depends on the MDP transition structure, but it often converges after a few iterations in practice (van Otterlo, n.d.).

The algorithm we critically followed in this context is similar to the Markov Decision Process of Policy Iteration Algorithm stated below in *Equation 3*. However, as stated earlier in this text, some value iteration policy with the algorithm as shown in *Equation 4* was also carried out for possible reward convergence, but we mainly studied the core iteration, which is the Policy Iteration method. The Value Iteration policy done here was episodic as considered in *Equation 5* below. In the episodic setting, the actual update rule approximates the expected return from the state s using the next reward Rk and the value function of the next state $v(s')$ via bootstrapping. Note that $v(s')$ can be decomposed as follows:

$$v(s')=\mathrm{E}[Rk+1+Rk+2+\ldots+RT|Sk+1=s'].$$

Therefore, $v(s')$ quantifies the part of the return aside from the next reward Rk, and the update rule indeed quantifies the expected return from s. $V(s')$ initialization affects the reward in a way that, if $v(s')$ is initialized too high or low, then the update rule will simultaneously be adjusting it to the optimal value $v*(s')$ while adjusting $v(s)$ closer to $v*(s)$. However, the bulk of the work was focused on policy iteration, and value iteration importance is insignificant to the work.

```python
log_path = os.path.join('Training', 'Logs')
model = PPO("MlpPolicy", env, verbose=1, tensorboard_log=log_path)
model.learn(total_timesteps=400000)
```

```
|    time_elapsed         | 7           |
|    total_timesteps      | 6144        |
| train/                  |             |
|    approx_kl            | 0.019601729 |
|    clip_fraction        | 0.274       |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.681      |
|    explained_variance   | 0.501       |
|    learning_rate        | 0.0003      |
|    loss                 | 0.0633      |
|    n_updates            | 20          |
|    policy_gradient_loss | -0.0231     |
|    value_loss           | 0.173       |
-----------------------------------------
----------------------------------------
| rollout/                |            |
|    ep_len_mean          | 60         |
|    ep_rew_mean          | 4.06       |
| time/                   |            |
|    fps                  | 706        |
|    iterations           | 4          |
|    time_elapsed         | 11         |
|    total_timesteps      | 8192       |
| train/                  |            |
|    approx_kl            | 0.01763113 |
|    clip_fraction        | 0.121      |
|    clip_range           | 0.2        |
|    entropy_loss         | -0.651     |
|    explained_variance   | 0.508      |
|    learning_rate        | 0.0003     |
|    loss                 | 0.117      |
```

Figure 13: Sample of Agent training using the PPO Policy Model

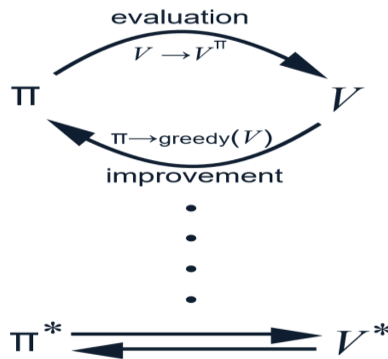


Figure 14: Policy Evaluation - Improvement Cycle

5.3.3. Sampling Method and Reward Function

In reinforcement learning, much emphasis is laid on finding the near-optimal policy or the near-optimal reward. One of the sampling methods used in this research work is the Episodic Sampling Method. In the episodic setting, for every episode, the learner acts for some finite number of steps, starting from a fixed starting state $s_o \sim \mu$, the learner observes the trajectory, and the state resets to $s_o \sim \mu$. This episodic model of feedback applies to both the finite-horizon MDPs (where each episode lasts for H-steps, and then the state is reset to $s_o \sim \mu$) and infinite horizon settings (Here, the agent can terminate the episode at will or the episode terminates at each step with probability 1 - ϒ. After termination, the state is reset to $s_o \sim \mu$).

The reward function is an important part of this episodic task, and it specifies implicitly the goal of learning. Here, we assigned states in which the agent has won a positive reward value and if the state conditions were not met, a negative reward value was assigned. The network is designed to detect anomalies and make some amendments by either reconfiguring the system files or replacing the faulty node automatically, without human intervention.

The reward function specifies rewards for being in a state or doing some action in a state. The state reward function is defined as $R : S \rightarrow R$, and it specifies the reward obtained in states. However, there are two other definitions of reward. One can define either $R : S \times A \rightarrow R$ or $R : S \times A \times S \rightarrow R$. The first one gives rewards for acting in a state, and the second gives rewards for transitions between states. All definitions are interchangeable, though the last one is convenient

in model-free algorithms because there we usually need both the starting state and the resulting state in backing up values. The reward function is an important part of the MDP that specifies implicitly the goal of learning. In the episodic task of this project, all states in which the agent has won a positive reward value were assigned a non-zero value, and all states in which the agent lost and received a negative reward were assigned a zero-value reward value. This means we either have a positive reward or a negative reward. The goal of the agent is to reach non-zero valued states, which means winning and getting the solution to the anomalies. Thus, the reward function is used to give direction in which way the system, i.e., the MDP, should be controlled. Often, the reward function assigns non-zero rewards to non-goal states as well, which can be interpreted as defining sub-goals for learning (van Otterlo, n.d.).

Algorithm 1: Policy Iteration Algorithm

```
Data: θ: a small number
Result: V: a value function s.t. V ≈ v_*, π: a deterministic policy
        s.t. π ≈ π_*
Function PolicyIteration is
    /* Initialization                                                    */
    Initialize V(s) arbitrarily;
    Randomly initialize policy π(s);
    /* Policy Evaluation                                                 */
    Δ ← 0;
    while Δ < θ do
        for each s ∈ S do
            v ← V(s);
            V(s) ← Σ_{s',r'} p(s', r|s, π(s))[r + γV(s')];
            Δ ← max(Δ, |v − V(s)|);
        end
    end
    /* Policy Improvement                                                */
    policy-stable ← true;
    for each s ∈ S do
        old-action ← π(s);
        π(s) ← arg max_a Σ_{s',r'} p(s', r|s, a)[r + γV(s')];
        if old-action != π(s) then
            policy-stable ← false;
        end
    end
    if policy − stable then
        return V ≈ v_* and π ≈ π_*;
    else
        go to Policy Evaluation;
    end
end
```

Equation 3: Policy Iteration Algorithm

Algorithm 2: Value Iteration Algorithm

```
Data: θ: a small number
Result: π: a deterministic policy s.t. π ≈ π_*
Function ValueIteration is
    /* Initialization                                          */
    Initialize V(s) arbitrarily, except V(terminal);
    V(terminal) ← 0;
    /* Loop until convergence                                  */
    Δ ← 0;
    while Δ < θ do
        for each s ∈ S do
            v ← V(s);
            V(s) ← max_a Σ_{s',r'} p(s',r|s,a)[r + γV(s')];
            Δ ← max(Δ, |v − V(s)|);
        end
    end
    /* Return optimal policy                                   */
    return π s.t. π(s) = arg max_a Σ_{s',r'} p(s',r|s,a)[r + γV(s')];
end
```

Equation 4: Value Iteration Algorithm

Initialize V arbitrarily, e.g.,$V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

Equation 5: The Episodic Part of Value Iteration Policy

6. Result and Discussion

Creating a simple IoT network environment, which has random anomalies introduced, a remarkable success was achieved by training the agent to guess what best actions to take in the environment. We evaluated the agent's performance in the environment using the PPO algorithm (MlpPolicy model) which is a policy gradient method that learns from online data.

We compared the performance of the Reinforcement Learning algorithm to the general Machine Learning-based solution (Anand et al., 2022), which used a publicly accessible dataset at the TON-IoT repository. This comparison clearly defines why RL-based solutions of anomaly detection and resolution in IoT networks should be considered to ML-based solutions.

Feature	Service profile	Feature	Service profile
Ts (timestamp)	Connection activity	Ssl_version	SSL activity
Src-ip	Connection activity	Ssl_cipher	SSL activity
Src-port	Connection activity	Ssl_resumed	SSL activity
Dst-ip	Connection activity	Ssl_established	SSL activity
Dst-port	Connection activity	Ssl_subject	SSL activity
Proto	Connection activity	Ssl_issuer	SSL activity
Service	Connection activity	Weird_name	Violation activity
Duration	Connection activity	Weird_addl	Violation activity
Src-bytes	Connection activity	Weird_notice	Violation activity
Dst-bytes	Connection activity	http_trans_depth	HTTP activity
Conn-state	Connection activity	http_method	HTTP activity
Missed-bytes	Connection activity	http_uri	HTTP activity
src_pkt	Statistical activity	http_version	HTTP activity
src_ip_bytes	Statistical activity	http_request_body_len	HTTP activity
dst_pkts	Statistical activity	http_response_body_len	HTTP activity
dst_ip_bytes	Statistical activity	http_status_code	HTTP activity
Dns-query	DNS activity	http_user_agent	HTTP activity
dns_qclass	DNS activity	http_orig_mime_types	HTTP activity
dns_qtype	DNS activity	http_resp_mime_types	HTTP activity
dns_rcode	DNS activity	label	Data labelling
dns_AA	DNS activity	type	Data labelling
dns_RD	DNS activity	dns_rejected	DNS activity
dns_RA	DNS activity		

Table 1: TON dataset features

Feature	Feature
Ts (timestamp)	Ssl_version
Conn-state	Ssl_cipher
Src-port	Dst-port
Dst-bytes	Src-bytes
Missed-bytes	http_uri
src_pkt	dns_RD
src_ip_bytes	http_request_body_len
dst_pkts	http_response_body_len
dst_ip_bytes	http_status_code
dns_RA	http_user_agent
dns_qclass	dns_rejected
dns_qtype	http_resp_mime_types
dns_rcode	proto
label	dns_AA
Ssl_established	

Table 2: TON dataset features selected for training

The ML-based solution classified some known and unknown threats with the use of some machine learning classifiers such as the KNN, the Decision Tree, the SVM, and others. The algorithm is shown in *Equation 6* below. The test solution was carried out on the different

distributions of the dataset features, as seen in Tables 1 & 2 above (Anand et al., 2022). Table 3 below shows how the basic ML models (Anand et al., 2022) underperformed in three different scenarios of known and unknown threats combinations, as compared to the RL model, which yielded impressive results despite using a generalized/ simple IoT networks environment and introducing random anomalies into it. The agent was trained to detect the random anomalies introduced into the environment and proffer the specified solution according to the required action state. The episodic function in our experiment shows the rewards of our agent per iteration and how it has reasonably learned to adopt some actions to prioritize its reward.

Dataset	T-1 (2 known:7 unknown), %	T-2 (4 known:5 unknown)	T-3 (6 known:3 unknown)
TON IoT-4	40.4	43.37	55.02
TON IoT-5	42.4	48.36	57.92
TON IoT-6	39.4	41.95	61.4

Table 3: Performance of the ML-based solution on three different scenarios of threats test

6.1. The Learning Rate and The Reward Function Curve

The monitor wrapper output shows that the episodic mean reward was pegged maximally at 60 irrespective of its variations from the training start point. The reward function is directly proportional to the actions taken; this explains why the variation occurred at the beginning of the training. The agent tried to maximize its reward by prioritizing its action, and at the starting point, little reward was awarded. This happens as the agent was only starting to learn to guess the correct actions to solve the riddle in the environment.

The reward curve per episode, which shows how the agent was accumulating its reward per episode in the environment, is shown in *Fig. 16* below. The Implementation here was with the OpenAI PPO and PPO2 (Proximal Policy Optimization). The PPO2 is simply an updated version of the PPO algorithm as released by OpenAI. PPO2 is an implementation with vectorized environments, **optimized for GPU and better supports parallel training**. There are several

other differences (some of whose advantages are automatic normalization and clipped value functions) but use the same mathematical foundations as with PPO.

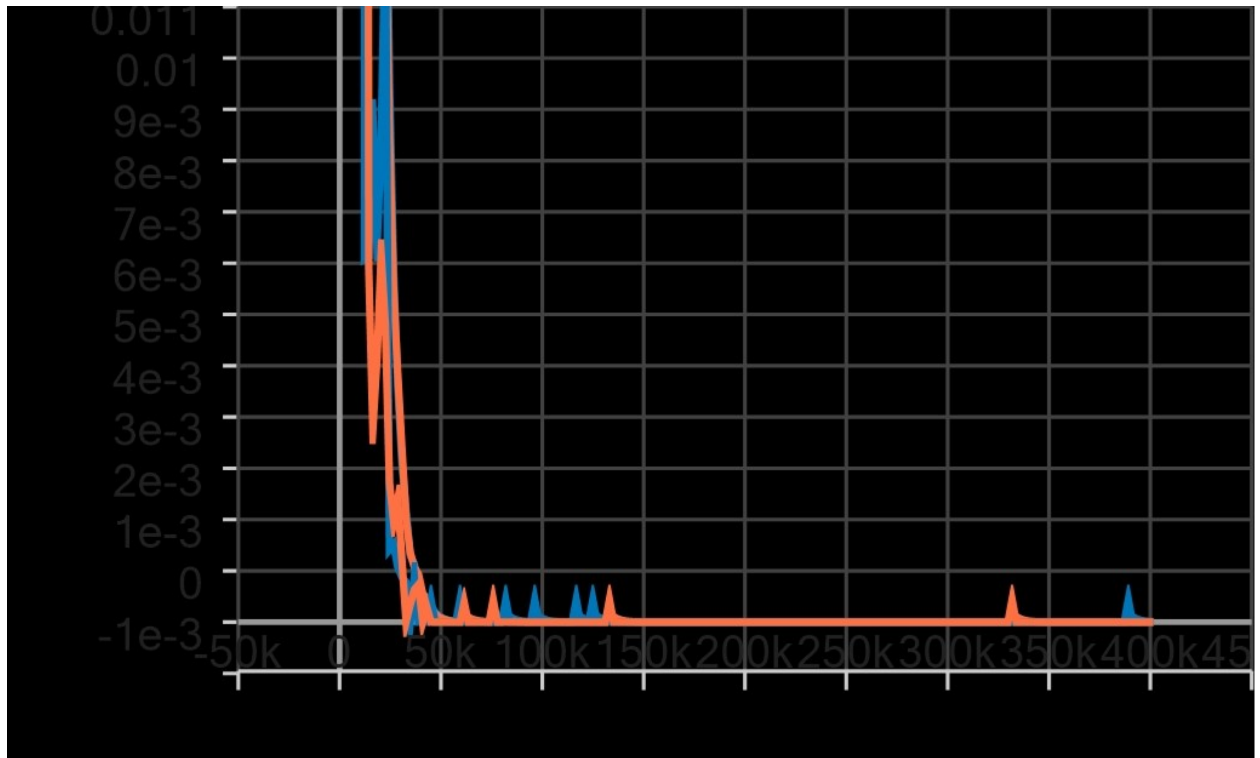


Figure 15: Train Clipping Curve

The learning curve as shown in *Fig. 17* below is a straight horizontal line that depicts the agent's smooth learning rate. The agent's rate of learning was PPO Clipping: A core feature of PPO is the **use of clipping in the policy and value function losses;** this is used to constrain the policy and value functions from drastically changing between iterations to stabilize the training process. As an on-policy model-free policy, the PPO model tries to strike a balance between the complexity of the TRPO and its performance. It tends to update the policy used and clipping is usually very small. It does that to closely monitor the difference between the updated policy and the old policy. Reusing generated samples is possible to some extent because PPO modifications are relatively minor, but it is important to sample terms correctly for updated action probabilities.

PPO executes K update steps; implementations frequently use early termination when divergence levels are exceeded.

Algorithm 1 Known and Unknown Threat Detection in Smart Home.

Input: Entities $SH_m, D_s, D_t, M_ds(f_s), M_dt(f_t)$.
Output: Successful Threat Detection using traffic analysis and Domain Adaptation.

1: **procedure** SALT: THE THREAT MODEL(SH_m)
2: System Initialization
3: Read the data
4: $D_s \leftarrow$ FetchSourceDomainData(SH_m)
5: $D_t \leftarrow$ FetchTargetDomainData(SH_m)
6: $PD_s \leftarrow$ CALL(PREPROCESSOR)(D_s)
7: $PD_t \leftarrow$ CALL(PREPROCESSOR)(D_t)
8: Calculate Marginal Probability Distribution of source and target data.
9: $M_ds(f_s) \leftarrow PD_s$
10: $M_dt(f_t) \leftarrow PD_t$
11: **if** ($M_ds(f_s) == M_dt(f_t)$ *and* $D_s == D_t$) **then**
12: Status $\leftarrow$ CALL(KNOWNTD)(D_s)
13: **else**
14: Status $\leftarrow$CALL(UNKNOWNTD)(D_s, D_t)
15: **end if**
16: **if** ($Status == 1$) **then**
17: $User \leftarrow Alert()$
18: print("Smart Home environment is under the threat")
19: **end if**
20: **end procedure**

Equation 6: ML-Algorithm

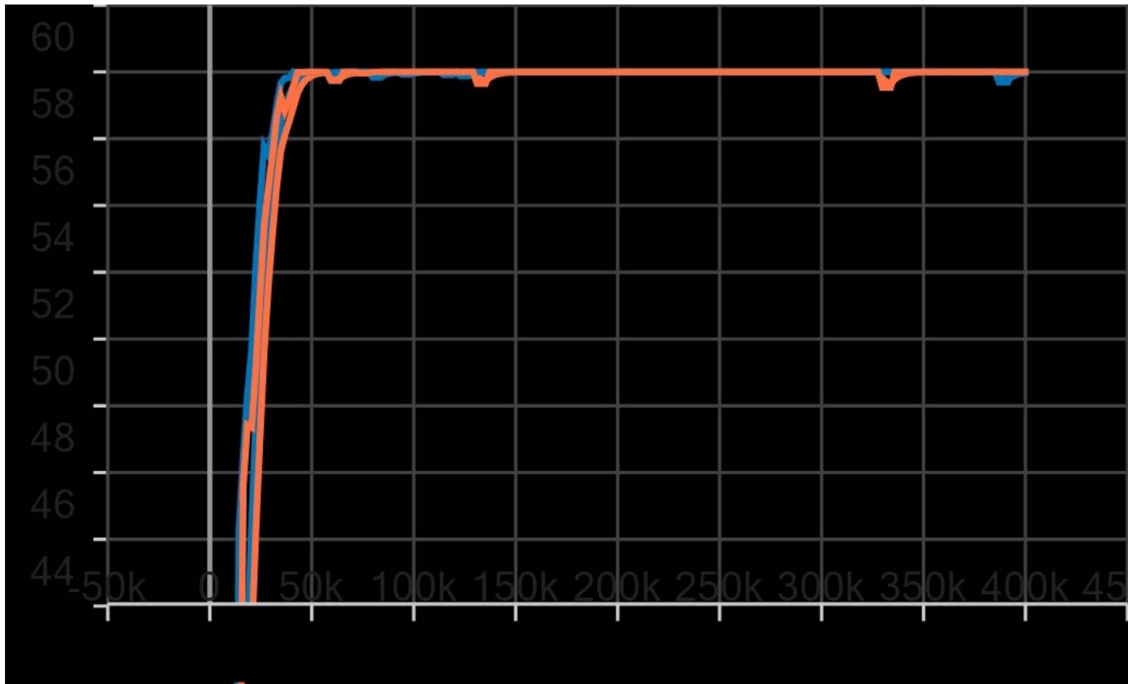


Figure 16: Episode Reward Mean

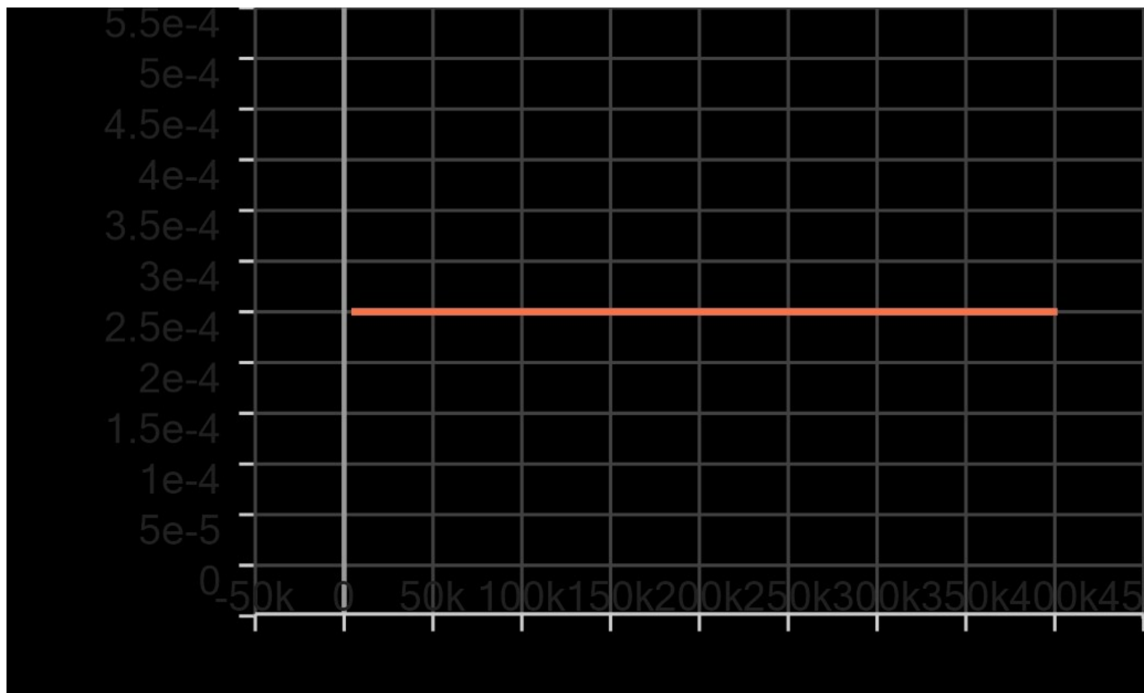


Figure 17: Train Learning Rate

7. Conclusion and Further Work

This research paper aims at developing a reinforcement learning model that could be used to detect and resolve anomalies in self-adaptable SDN-based IoT networks.

However, a simple and generalized environment was used in the research work. This simple technique was able to prove that the basic ML models could not resolve anomalies/ threats; they could only classify threats based on some predefined instructions/ algorithms. Reinforcement learning agents could make decisions as to what actions to take and what actions not to take to tackle the issue, which makes it more efficient in this kind of situation.

Reinforcement learning in self-adaptation systems has no limitations, as it is the best solution to knowledge-based, MAPE-K loop systems.

Further research work is needed in the area of specifying the kinds of attacks (which include but are not limited to, how the attacks are being tailored into the network). Further research work could be about finding what area of the network is under attack and the kind of attack(s) the affected node(s) suffer from. Another area of research could be about what measures to be considered to tackle the issue if any threat arises in the node(s) of an IoT network. Another area is specifying the kinds of actions to be taken by an agent to solve a dictionary of attacks in the IoT network environment and the agent's behaviors to the attacks in the environment. Many research areas need attention from researchers. Cyber threats are inevitably becoming sophisticated with an increase in technology use in our daily lives. In a sophisticated cyber-threat world, a counter-solution to the attacks must be devised. Some of the counter-solutions that could be researched are the following:

7.1. Exploring capabilities of model-based DRL methods.

Most DRL algorithms used for cyber defense so far are model-free methods, which are sample inefficient, as they require a large quantity of training data. These data are difficult to obtain in real cybersecurity practice. Researchers generally utilize simulators to validate their proposed approaches, but these simulators often do not

characterize the complexity and dynamics of real cyberspace of the IoT systems fully. Model-based DRL methods are more appropriate than model-free methods when training data are limitedly available because, with model-based DRL, it can be easy to collect data in a scalable way. Exploration of model-based DRL methods or the integration of model-based and model-free methods for cyber defense is, thus, an interesting future study. For example, function approximations can be used to learn a proxy model of the actual high-dimensional and possibly partial observable environment, which can be then employed to deploy planning algorithms, e.g., Monte-Carlo tree search techniques, to derive optimal actions. Alternatively, model-based and model-free combination approaches, such as model-free policy with planning capabilities or model-based look-ahead search can be used, as they aggregate advantages of both methods. However, current literature on applications of DRL to security in IoT networks often limits discretizing the action space, which restricts the full capability of the DRL solutions to real-world problems. Investigation of methods that can deal with continuous action spaces in cyber environments, e.g., policy gradient and actor-critic algorithms, is another encouraging research direction (Nguyen & Reddi, 2021).

7.2. Training DRL in adversarial cyber environments

AI can help defend against cyber attacks but can also facilitate dangerous attacks, i.e., *offensive AI*. Hackers can take advantage of AI to make attacks smarter and more sophisticated to bypass detection methods to penetrate computer systems or networks. For example, hackers may employ algorithms to observe the normal behaviors of users and employ the users' patterns to develop untraceable attack strategies. Machine learning-based systems can mimic humans to craft convincing fake messages that are utilized to conduct large-scale phishing attacks. Alternatively, attackers can poison the data pool used for training deep learning methods (i.e., *machine learning poisoning*) or attackers can manipulate the states or policies, and falsify part of the reward signals in RL to trick the agent into taking sub-optimal actions, resulting in the agent being compromised. These kinds of attacks are difficult to prevent, detect, and fight against, as

they are part of a battle between AI systems. *Adversarial machine learning*, especially supervised methods, has been used extensively in cyber security but very few studies have been found on using adversarial RL. Adversarial DRL or DRL algorithms trained in various adversarial cyber environments are worth comprehensive investigation, as they can be a solution to battle against the increasingly complex offensive AI systems (Nguyen & Reddi, 2021).

BIBLIOGRAPHY

Agarwal, A., Jiang, N., Kakade, S. M., & Sun, W. (n.d.). *Reinforcement Learning: Theory and Algorithms*.

Anand, P., Singh, Y., Singh, H., Alshehri, M. D., & Tanwar, S. (2022). SALT: Transfer learning-based threat model for attack detection in smart home. *Scientific Reports*, *12*(1), 12247. https://doi.org/10.1038/s41598-022-16261-9

Flauzac, O., Gonzalez, C., Hachani, A., & Nolot, F. (2015). SDN Based Architecture for IoT and Improvement of the Security. *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, 688–693. https://doi.org/10.1109/WAINA.2015.110

Https://en.wikipedia.org/wiki/Markov_decision_process. (n.d.). https://en.wikipedia.org/wiki/Markov_decision_process

Kang, B., Kim, D., & Choo, H. (2017). Internet of Everything: A Large-Scale Autonomic IoT Gateway. *IEEE Transactions on Multi-Scale Computing Systems*, *3*(3), 206–214. https://doi.org/10.1109/TMSCS.2017.2705683

Lalanda, P., McCann, J. A., & Diaconescu, A. (2013). *Autonomic Computing: Principles, Design and Implementation*. Springer London. https://doi.org/10.1007/978-1-4471-5007-7

Lei, L., Tan, Y., Zheng, K., Liu, S., Zhang, K., Xuemin, & Shen. (2020). *Deep Reinforcement Learning for Autonomous Internet of Things: Model, Applications and Challenges* (arXiv:1907.09059). arXiv. http://arxiv.org/abs/1907.09059

Li, J., Nejati, S., & Sabetzadeh, M. (2022). *Learning Self-adaptations for IoT Networks: A Genetic Programming Approach* (arXiv:2205.04352). arXiv. http://arxiv.org/abs/2205.04352

Narayanankutty, H. (2021). Self-Adapting Model-Based SDSec For IoT Networks Using Machine Learning. *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, 92–93. https://doi.org/10.1109/ICSA-C52384.2021.00023

Nguyen, T. T., & Reddi, V. J. (2021). Deep Reinforcement Learning for Cyber Security. *IEEE Transactions on Neural Networks and Learning Systems*, 1–17. https://doi.org/10.1109/TNNLS.2021.3121870

Sarica, A. K., & Angin, P. (2020). Explainable Security in SDN-Based IoT Networks. *Sensors*, *20*(24), 7326. https://doi.org/10.3390/s20247326

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms* (arXiv:1707.06347). arXiv. http://arxiv.org/abs/1707.06347

Tangari, G., Tuncer, D., Charalambides, M., Qi, Y., & Pavlou, G. (2018). Self-Adaptive Decentralized Monitoring in Software-Defined Networks. *IEEE Transactions on Network and Service Management*, *15*(4), 1277–1291. https://doi.org/10.1109/TNSM.2018.2874813

Tayyaba, S. K., Shah, M. A., Khan, O. A., & Ahmed, A. W. (2017). Software Defined Network (SDN) Based Internet of Things (IoT): A Road Ahead. *Proceedings of the International Conference on Future Networks and Distributed Systems*, 1–8. https://doi.org/10.1145/3102304.3102319

van Otterlo, M. (n.d.). *Markov Decision Processes: Concepts and Algorithms*.

Williams, R. J. (n.d.). *Simple statistical gradient-following algorithms for connectionist reinforcement learning*.