PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY YAHIA FARES OF MEDEA



Faculty of Technology

Master's Thesis

Department of Electrical Engineering

Specialty: Telecommunications Systems

Intelligent Routing Optimization Model For QoS-aware Software-Defined Networks Using Deep Reinforcement Learning

---

Presented by :

Mohamed Akram DJERRAR

Zineddine BOURAHLA

Directed by :

Mr. Sidahmed LACHENANI

Mr. Hamza KHEDDAR

Promotion 2025

ACKNOWLEDGMENT

First and foremost, we express our deepest gratitude to Allah, who granted us the strength, clarity, and perseverance to complete this work. Without His guidance and mercy, none of this would have been possible.

We extend our sincere thanks to our supervisors, Mr. Sidahmed LACHENANI and Mr. Hamza KHEDDAR.

Our gratitude goes to the members of the thesis committee for their time, valuable insights, and constructive feedback, which helped us improve the quality of our work.

We are thankful to all the professors and staff of the Faculty of Technology at Yahia Fares University of Medea, whose efforts contributed to our academic and personal growth over the years.

Finally, our heartfelt thanks go to our families. Their unconditional love, constant encouragement, and unwavering belief in us have been the foundation that carried us through every challenge. We owe them more than words can express.

- Mohamed Akram Djerrar & Zinnedine Bourahla

DEDICATION

To the woman whose love is limitless, whose sacrifices are silent but immeasurable my mother.whose silent sacrifices and boundless love built everything I am.This achievement is yours before it is mine.
To my father, for being the steady compass in my life. Your guidance, values, and quiet support have shaped me more than words can express.
To my beloved brothers, Abdalmalek and Abdallah, And to my precious little sister, Meriem, Your presence has been a constant reminder of what I'm fighting for.
To my elder sister, Metalbi Soumia, Thank you for always being the calm in the storm, the voice of reason when I faltered, and the unwavering support I could count on.
To all my extended family Djerrar and Metalbi Your prayers, love, and unwavering support carried me through.
To my binome and true partner in this academic journey, Zinnedine Bourahla, Thank you for your hard work, patience, and trust throughout every step of this thesis.
To my friends, for standing by me in silence and struggle alike.
To my professors, for shaping my academic thinking and values through their dedication and knowledge.
To the entire Telecom class of 2024–2025, We shared not just classrooms, but challenges, growth, and memories that will last.
And finally, to myself. For holding on. For pushing forward. For never giving up, even when it was the hardest thing to do.

- Mohamed Akram Djerrar

DEDICATION

To my dear parents, Your unconditional love, constant support, and tireless sacrifices have been the foundation of everything I have achieved.

To my brothers and sisters, Thank you for your encouragement, patience, and belief in me throughout this journey.

To my binôme, Your collaboration, dedication, and shared efforts have made this work possible. I am grateful for your partnership throughout this experience.

This work is dedicated to all of you, who have been my constant source of strength, motivation, and support.

Thank you.

- Zineddine Bourahla

الملخص:

تقترح هذه الأطروحة نموذجًا ذكيًا لتوجيه البيانات في الشبكات المعرفة برمجياً(SDN) ، مع مراعاة جودة الخدمة(QoS) ، باستخدام التعلم العميق بالتعزيز .(DRL) يعتمد الوكيل المقترح على شبكة DQN لاختيار أفضل المسارات حسب حالة الشبكة في الوقت الفعلي. أظهرت النتائج تحسنًا ملحوظًا مقارنة بخوارزمية ديكسترا، حيث تم تسجيل انخفاض في التأخير بنسبة تصل إلى 60٪، وتقليل في التذبذب بنسبة 50٪، وزيادة في معدل نقل البيانات بنسبة 207٪. يتميز النموذج بالمرونة وقابلية التوسع، مما يجعله مناسبًا للتطبيق في مراكز البيانات، وإنترنت الأشياء(IoT) ، وشبكات الجيل الخامس(5G).

الكلمات المفتاحية:
الشبكات المعرفة برمجياً(SDN) ، جودة الخدمة(QoS) ، التعلم العميق بالتعزيز، التوجيه الذكي، الشبكات العصبية العميقة (DQN)، منصةONOS ، محاكيMininet ، شبكات الجيل الخامس(5G) ، إنترنت الأشياء(IoT)

Abstract:

This thesis presents an intelligent routing model for Software-Defined Networks (SDNs), taking into account Quality of Service (QoS) using Deep Reinforcement Learning (DRL). The agent, based on a DQN network, selects the best paths according to real-time network conditions. Results show significant improvement over the Dijkstra algorithm: up to 60% reduction in delay, 50% less jitter, and a 207% increase in throughput. The model is flexible, scalable, and suitable for data centers, IoT, and 5G networks.

Keywords:
Software-Defined Networks (SDN), Quality of Service (QoS), Deep Reinforcement Learning, Intelligent Routing, DQN, ONOS, Mininet, 5G, Internet of Things (IoT)

Résumé :

Ce mémoire propose un modèle de routage intelligent pour les réseaux définis par logiciel (SDN), en tenant compte de la qualité de service (QoS), grâce à l'apprentissage profond par renforcement (DRL). L'agent basé sur un réseau DQN choisit les meilleurs chemins selon l'état du réseau en temps réel. Les résultats montrent une amélioration notable par rapport à l'algorithme de Dijkstra : réduction du délai jusqu'à 60 %, de la gigue de 50 % et augmentation du débit de 207 %. Le modèle est flexible, évolutif, et adapté aux centres de données, à l'IoT et aux réseaux 5G.

Mots-clés :

Réseaux définis par logiciel (SDN), Qualité de service (QoS), Apprentissage profond par renforcement, Routage intelligent, DQN, ONOS, Mininet, 5G, Internet des objets (IoT)

TABLE OF CONTENTS

Acknowledgment **i**

Abstract **iv**

Table of contents **vii**

List of Figures **ix**

List of tables **x**

Abbreviations list **xi**

General Introduction **1**

I Software-Defined Networking **3**

I.1 Introduction 3

I.2 Traditional Networking Overview 3

I.2.1 Key Components of Traditional Networks 4

I.2.2 How Traditional Networks Operate 4

I.2.3 Limitations of Traditional Networks 4

I.2.4 The Role of Virtualization in Network Evolution 5

I.3 Virtualization 5

I.3.1 Virtualization Technologies 5

I.3.2 Benefits of Virtualization 6

I.4 Software Defined Networking (SDN) 6

I.4.1 Software Defined Networking Architecture 7

I.4.2 Interfaces in Software Defined Networking 8
I.4.3 Components of an Software Defined Networking architecture 9
I.4.3.1 Software Defined Networking (SDN) Controller 9
I.4.3.2 SDN Switch 10
I.4.4 OpenFlow 11
I.4.4.1 How OpenFlow Works 11
I.4.4.2 Advantages of Using OpenFlow 12
I.5 Conclusion 13

II State-of-art of Machine learning and Reinforcement Learning 14
II.1 Introduction 14
II.2 Machine Learning 14
II.2.1 Types of Machine Learning 15
II.2.2 Model building techniques 17
II.2.3 Strategy of Machine Learning 17
II.2.4 Model evaluation and post-building 18
II.3 Reinforcement Learning 19
II.3.1 Fundamentals of Reinforcement Learning 19
II.3.2 Q-learning Algorithm 20
II.3.3 Deep Neural Networks 22
II.3.4 Deep Q-Network 22
II.3.5 Why Reinforcement Learning for SDN Routing 22
II.4 Conclusion 23

III Implementation, Tests and Results 24
III.1 Introduction 24
III.2 Environment Setup and Baseline Implementations 24
III.2.1 Virtual Environment 24
III.2.2 Software-Defined Network Controller and Applications 25
III.2.3 Interaction Diagrams 25
III.2.4 Network Topology 26
III.2.5 Traffic Generation and Monitoring 27
III.2.6 Performance Metrics Explanation 29
III.3 Dijkstra's Algorithm Implementation 29

III.4 Evaluation of Dijkstra-Based Routing 31
III.4.1 ONOS Configuration and Topology Setup 31
III.4.2 Traffic Generation and D-ITG Testing 33
III.5 Reinforcement Learning Model Design and Environment 35
III.5.1 Reinforcement learning (RL) Agent Architecture and Training Setup 36
III.5.2 Training of the Reinforcement Learning Model 39
III.5.2.1 Training Dataset Flow Generation and Characteristics 39
III.5.2.2 Defining Training Hyperparameters 40
III.5.2.3 Explanation of Key Hyperparameters 40
III.5.3 Training Workflow 42
III.6 Reinforcement Learning-Based Routing Evaluation 42
III.6.1 RL Agent Training 42
III.6.2 Training Results 43
III.6.3 ONOS RL App Integration 44
III.6.4 Topology and Ping Test 46
III.6.5 Results Of RL Performance 47
III.7 Comparative Analysis 48
III.7.1 Synthetic Flow Allocation Comparison 48
III.7.2 QoS Comparison RL vs Dijkstra 50
III.8 Performance Comparison with Related Works 51
III.9 Conclusion 52

General Conclusion **53**

A APPENDIX : ENVIRONMENT CONFIGURATION **59**
A.1 Installing Maven (3.6.3) and Karaf Runtime (4.2.9) 59
A.2 Installing Oracle Java 11 59
A.3 Installing ONOS Uguisu (2.2.4) and build it with Bazel (1.6.1) 60
A.3.1 Installing ONOS Uguisu (2.2.4) 60
A.3.2 Building ONOS using Bazel (1.6.1) 61
A.3.3 Installing gcc compiler (9.3.0), python2, python3, and pip 61
A.4 Installing Mininet (2.2.2) 62
A.4.1 Running Mininet and ONOS 62
A.5 Installation of generating and monitoring tools 63

LIST OF FIGURES

I.1 Comparison between Traditional and SDN Architectures 7
I.2 SDN Architecture Layers 8
I.3 Open vertual switch 11
I.4 Packet forwarding procedure using OpenFlow 12

II.1 Machine Learning techniques 15
II.2 Machine Learning (ML)'s approach 18
II.3 Example of RL scenario 20

III.1 ONOS GUI view of the complete SDN topology after deployment and REST-based configuration injection 27
III.2 Distributed Internet Traffic Generator (D-ITG) Architecture 28
III.3 Activating and Using `org.onosproject.fwd` 32
III.4 host-to-host connectivity in Mininet using Dijkstra routing via ONOS's org.onosproject.fwd application. 33
III.5 Partial ITGDec dijkstra Report from Host 1 to Host 8. 34
III.6 System architecture integrating SDN and reinforcement learning. 36
III.7 Deep Neural Networks (DNN) representation of the RL agent 37
III.8 Representation of the ReLU activation function 38
III.9 Excerpt of the JSON file containing the flows to train the RL agent 40
III.10 Distribution in time of training flows grouped by rate 41
III.11 Diagram to train the RL agent 43
III.12 Training Reward per Episode 44
III.13 Diagram of Interaction Workflow 46

III.14ITGDec RL Report from Host 1 to Host 8 47
III.15Flow Allocation Comparison Between RL and Dijkstra Algorithm 49
III.16Moving average of 10 comparison files for the comparison of RL and Dijkstra 49
III.17QoS Metrics Comparison Dijkstra vs RL 51

A.1 Maven installation verification. 60
A.2 Bazel installation verification 61
A.3 ONOS APPS variable in bash profile file 62

LIST OF TABLES

II.1 Q-Value Table 21

III.1 Performance Metrics for Dijkstra-Based Routing (Using D-ITG) 34
III.2 Hyperparameters Used to Train the RL Model 41
III.3 Performance Metrics for RL-Based Routing (Using D-ITG) 48
III.4 QoS Metrics Comparison: Dijkstra vs RL (Using D-ITG) 50
III.5 Comparison of RL-Based SDN Routing Approaches 52

ABBREVIATIONS LIST

API	Application Programming Interface
CLI	Command Line Interface
D-ITG	Distributed Internet Traffic Generator
DNN	Deep Neural Networks
DQN	Deep Q-Network
IP	Internet Protocol
IPFIX	IP Flow Information Export
IT	Information Technology
MAC	Media Access Control
ML	Machine Learning
NBI	Northbound Interface
NetFlow	Network Flow
ONOS	Open Network Operating System
OS	Operating System
OVS	Open vSwitch
QoS	Quality of Service
RL	Reinforcement learning
RSPAN	Remote Switched Port Analyzer
SBI	Southbound Interface

SDN Software Defined Networking

sFlow Sampled Flow

VM Virtual Machine

GENERAL INTRODUCTION

In recent years, the world has witnessed a rapid evolution in digital technologies, resulting in significant growth in the complexity of communication networks and the volume of data transmitted through them. The increasing demand for high-performance networking infrastructure, driven by cloud computing, multimedia applications, and the Internet of Things (IoT), has exposed the limitations of traditional networking models, which rely heavily on manually configured, hardware-based systems.

To meet the needs of modern network environments, Software-Defined Networking (SDN) has emerged as a powerful solution that decouples the control and data planes, enabling centralized and programmable network management. At the same time, Machine Learning (ML)—and particularly Reinforcement Learning (RL)—has shown great promise in enabling intelligent systems capable of self-optimization and adaptive decision-making. The integration of ML techniques into SDN environments opens the door to the development of more dynamic, scalable, and efficient networks [1].

This thesis explores the potential of combining Software-Defined Networking with Reinforcement Learning to enhance network performance, focusing specifically on the optimization of routing decisions. By comparing a traditional shortest-path algorithm (Dijkstra) with a learning-based approach, we aim to demonstrate how intelligent agents can provide better adaptability and improve Quality of Service (QoS) in dynamic network scenarios.

The thesis is structured as follows:

Chapter 1 provides a detailed overview of traditional networking architectures and the evolution toward SDN. It explains the limitations of legacy systems and describes how SDN addresses these issues through centralized control and programmability.

Chapter 2 introduces the principles of Machine Learning, focusing particularly on Reinforcement Learning. It presents the key concepts, types of learning, model structures, and training processes. Special attention is given to Q-learning, Deep Neural Networks (DNNs), and Deep Q-Networks (DQNs), which are used in this work for routing optimization.

Chapter 3 focuses on the practical implementation and evaluation of two routing approaches

within an SDN environment: the traditional Dijkstra algorithm and an RL-based model. It includes the virtual environment setup, model training, testing procedures, and comparative performance analysis based on QoS metrics.

Through this research, we aim to contribute to the development of smarter and more autonomous network management systems by demonstrating the feasibility and advantages of integrating Reinforcement Learning into SDN-based routing.

CHAPTER I

SOFTWARE-DEFINED NETWORKING

I.1 Introduction

As digital technologies continue to evolve rapidly, modern networks must handle increasing traffic, diverse applications, and higher performance expectations. Traditional networking architectures, which rely on manually configured hardware devices such as routers and switches, are becoming difficult to scale, manage, and adapt to changing demands.

To overcome these challenges, SDN has emerged as a new networking model. SDN separates the control logic from the underlying hardware, allowing centralized and programmable management of network resources. This makes networks more flexible, efficient, and easier to manage compared to traditional approaches.

This chapter provides an overview of both traditional and SDN-based networking. It starts by explaining how conventional networks operate and the limitations they face. It then introduces the core concepts of SDN, its architecture, components, and advantages. The goal is to highlight the evolution from static, hardware-driven networks to dynamic, software-defined environments that support modern digital needs.

I.2 Traditional Networking Overview

Before the rise of SDN, most organizations used traditional networking models to manage their communication systems. In this model, the network is built using physical, on-site hardware such as switches, routers, firewalls, and servers. All network operations—including data routing, security, and maintenance are managed directly by the organization's Information Technology (IT) team [2].

I.2.1 Key Components of Traditional Networks

Traditional networks are built using dedicated physical devices, each playing a specific role in enabling communication and ensuring network performance and security. The following are the main components typically found in traditional network infrastructures [2]:

Switches: These connect multiple devices within the same local network, allowing them to communicate with each other.

Routers: They manage the flow of data between different networks, such as connecting an internal network to the internet.

Firewalls: These act as a security layer, filtering network traffic based on rules set by administrators.

Servers: Physical machines that store data and run essential applications and services used by users on the network.

I.2.2 How Traditional Networks Operate

In a traditional setup, all the network equipment is physically installed and managed on-site. Users connect to the network through wired or wireless access points, and data flows through fixed paths determined by the hardware configuration. Since everything is local, the IT department must handle all tasks, including configuring devices, ensuring security, and upgrading systems when needed. This gives full control over the network, but also increases complexity, especially as the network grows [2].

I.2.3 Limitations of Traditional Networks

Traditional networks, while effective for small-scale deployments, face several limitations that make them unsuitable for modern, large, and dynamic environments [3]:

- **Manual Configuration via CLI:** Network devices are commonly configured using Command Line Interface (CLI), which is time-consuming and inefficient for large networks. This approach also increases operational costs and the risk of configuration errors .
- **Poor Scalability:** Traditional network management does not scale well. Configuring each device individually becomes impractical as the number of devices grows .
- **Configuration Drift:** Without centralized change tracking, ad hoc modifications can lead to configuration inconsistencies across the network over time, a problem known as configuration drift.
- **Limited Automation Support:** Even graphical interfaces often only manage one device at a time, offering limited automation and remaining inefficient for large-scale operations.

- **Inflexibility:** Traditional networks are statically arranged and cannot easily adapt to complex traffic patterns or evolving application demands.
- **Human Error-Prone:** Manual processes involving secure shell **SSH!** (**SSH!**) sessions and configuration copy-pasting are susceptible to errors and typos, which can disrupt network functionality.

I.2.4 The Role of Virtualization in Network Evolution

As traditional networks face growing challenges related to scalability, flexibility, and automation, modern solutions have emerged to bridge the gap between static infrastructure and dynamic demands. One of the most impactful advancements is **virtualization**, a technology that abstracts hardware resources to enable multiple virtual systems to operate on a single physical platform [4]. Virtualization plays a foundational role in transforming how network services are deployed and managed. By decoupling software from physical devices, it allows network functions to become more portable, scalable, and easier to manage. This shift not only improves resource utilization and cost-efficiency but also sets the stage for more advanced, software-driven models such as Software-Defined Networking SDN. The next section explores the concept of virtualization in more detail, including its technologies, types, and benefits in modern network environments.

I.3 Virtualization

Virtualization is the process of creating software-based versions of hardware systems, such as servers, storage devices, and networks. It allows network functions to run on shared physical machines by separating (decoupling) the software from the hardware. This means that one physical machine can perform the role of many virtual devices. Virtualization helps reduce the cost of equipment, energy, and maintenance, and also minimizes the need for specialized technical staff [5].

Instead of having a separate physical device for each function, virtualization allows multiple virtual systems to run together on a single server. These virtual systems behave like real physical devices and are managed using software tools. This flexibility is one of the key features in modern networks and cloud computing.

I.3.1 Virtualization Technologies

There are two main types of virtualization technologies: containers and Virtual Machines (VMs). Both allow multiple applications to run on the same physical server, but they do so in different ways.

1)Containers: Containers are lightweight virtual environments that share the host system's operating system Operating System (OS). They are placed on top of the host OS (e.g., Linux or

Windows), and they often share system components like libraries and binaries. These shared components are usually read-only. Containers are very efficient in terms of speed and resource usage. They start within seconds and take up only a few megabytes of memory. Because they are smaller and faster, containers are ideal for deploying applications in cloud environments or microservices. However, since they share the same OS kernel, containers can be less secure and more vulnerable to attacks if not properly managed [6].

2)Virtual Machines VM: Virtual Machines are full-featured systems that include their own operating system. Each VM operates independently and can run a different OS from the host machine. For example, a server can run both a Linux VM and a Windows VM at the same time. Unlike containers, VMs do not share the OS with the host. Instead, they include a full copy of the OS, along with their own libraries and applications. This makes VMs larger (often several gigabytes) and slower to start. However, the advantage of VMs is that they provide stronger isolation and better security. Because of this, VMs are preferred in environments where data security is critical, such as enterprise networks [6].

I.3.2 Benefits of Virtualization

Virtualization brings many benefits to modern IT and network environments. First, it improves resource efficiency by allowing multiple virtual systems to run on a single physical server. This reduces hardware costs, energy consumption, and physical space requirements. Second, it simplifies system management, as virtual machines or containers can be easily created, updated, or moved from one server to another. IT administrators can manage many systems centrally, which saves time and reduces maintenance effort. Third, virtualization allows faster deployment of applications and services, supporting business agility. It also improves disaster recovery because virtual machines can be backed up and restored quickly. Lastly, virtualization enhances flexibility and scalability, making it easier for organizations to adapt to changes. While containers offer better performance and scalability, virtual machines provide stronger isolation and security, which is important for enterprise networks. Overall, virtualization leads to cost savings, better system utilization, and greater operational efficiency [7].

I.4 Software Defined Networking (SDN)

As networks became larger and more complex, traditional networking methods started to show their limits. Managing many devices manually, adapting to traffic changes, and adding new features became difficult and slow. To solve these problems, researchers and engineers introduced a new concept called SDN. A visual comparison between traditional and SDN architectures is presented in **Figure I.1** [8]

SDN is a modern approach that changes how networks are designed and managed. The main idea of SDN is to separate the control plane from the data plane. Instead of having each network device decide where to send data, SDN uses a central system called a controller to make all the

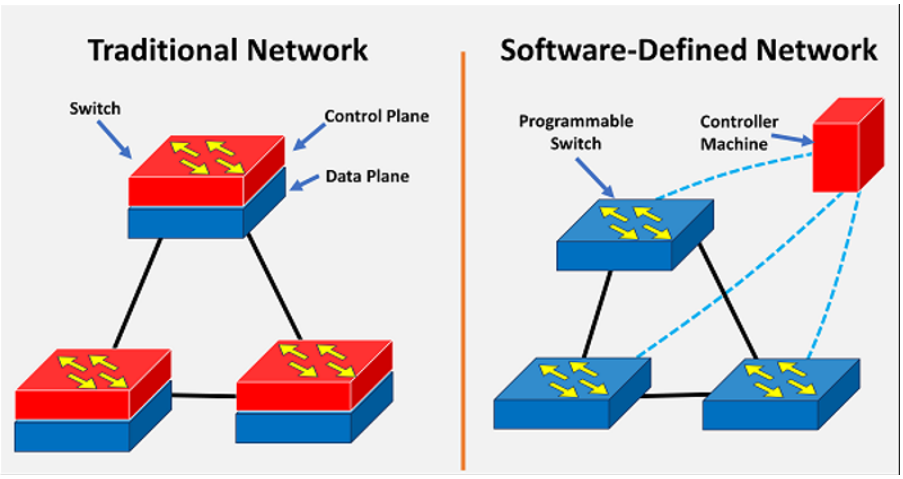


Figure I.1: Comparison between Traditional and SDN Architectures

decisions. The controller communicates with network devices and tells them how to handle the data [9].

This separation gives many benefits. First, it makes the network programmable, which means administrators can control and update the network using software. Second, it becomes easier to automate network management tasks, such as traffic routing or security policies. Third, the controller has a global view of the entire network, so it can make better and faster decisions compared to traditional devices that only see part of the network [10] .

SDN is based on open standards, such as OpenFlow, which is a protocol that allows the controller to talk to network devices. Because of this openness, SDN allows using different types of hardware from various vendors, improving flexibility and reducing costs [11].

Today, SDN is used in many areas, such as data centers, cloud networks, and Internet service providers (ISPs). It supports modern applications that need high speed, low latency, and strong security.

I.4.1 Software Defined Networking Architecture

As illustrated in **Figure I.2** [12] the architecture of SDN is different from traditional networks. It is built around the idea of separating the control plane from the data plane, which allows centralized and more intelligent network management. SDN architecture is typically divided into three main layers: the Application Layer, the Control Layer, and the Infrastructure Layer [10].

(1 Application Layer This is the top layer of the SDN architecture. It contains network applications that define what the network should do. These can include routing algorithms, security tools, traffic monitoring, load balancing, and quality of service Quality of Service (QoS) functions. These applications communicate with the controller using Northbound Interfaces (NBIs), usually based on Application Programming Interfaces (APIs) such as REST [13].

(2 Control Layer This is the middle layer, where the SDN controller operates. The controller is the “brain” of the SDN. It collects information from the network devices and makes decisions

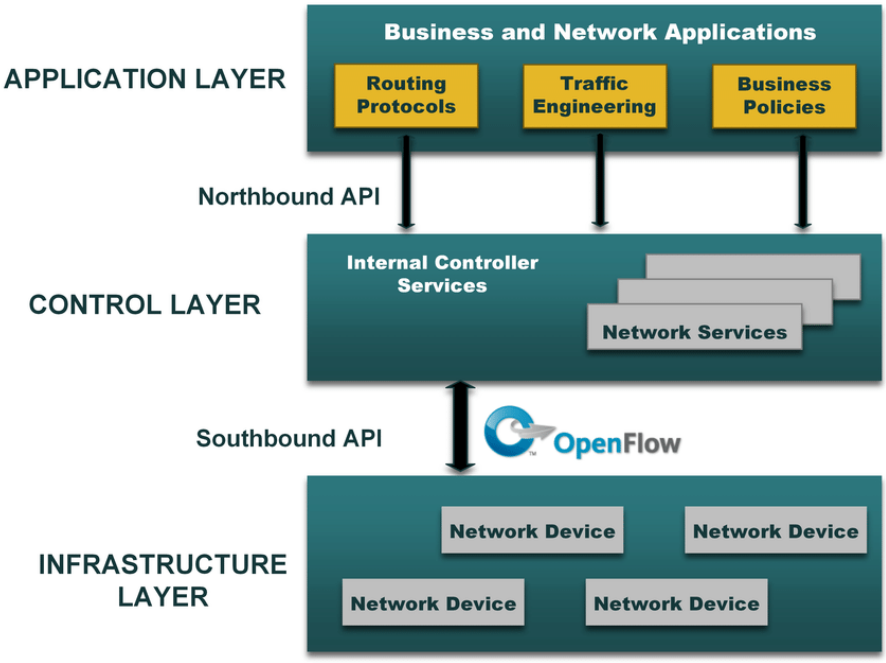


Figure I.2: SDN Architecture Layers

based on the goals defined by the applications. It then sends instructions to the devices on how to handle network traffic. The controller uses southbound interfaces Southbound Interfaces (SBIs), such as the OpenFlow protocol, to communicate with switches and routers [9].

There are many types of SDN controllers available, such as Open Network Operating System (ONOS), Ryu, and OpenDaylight, each with different features and programming support.

(3 Infrastructure Layer This is the bottom layer, also called the data plane. It includes physical and virtual network devices like switches and routers. These devices are responsible for forwarding packets based on the rules received from the controller. They do not make decisions on their own they just follow the controller's instructions [10].

I.4.2 Interfaces in Software Defined Networking

In this part we define the key interfaces of an SDN system and highlight the features of Software Defined Networking:

1)Southbound Interface SBI: SBI connects the SDN controller to the physical or virtual network devices, such as switches and routers. It allows the controller to communicate directly with these devices — sending control commands and receiving network data like traffic statistics, topology updates, or flow information.

SBI is essential because it gives the controller the ability to configure how the devices handle network traffic. For example, the controller can send rules to a switch telling it how to forward packets, block traffic, or redirect flows.

The most widely used protocol for the Southbound Interface is OpenFlow, which was one of the first protocols created for SDN. With OpenFlow, the controller installs flow entries in the

switches' flow tables to define how packets should be processed [9].

2)Northbound Interface NBI: The Northbound Interface connects the control layer to the application layer. It allows software applications to talk to the SDN controller and define what the network should do. These applications can include routing, traffic monitoring, security systems, and more.

Most northbound interfaces use APIs (Application Programming Interfaces). APIs give developers a simple way to interact with the controller without needing to understand how switches or routers work. For example, a developer can use a REST API to ask the controller for traffic statistics or to tell it to block traffic from a certain Internet Protocol (IP) address [10] .

I.4.3 Components of an Software Defined Networking architecture

I.4.3.1 SDN Controller

In an SDN network, the controller acts as the main decision-making unit. It sits between the application layer and the network devices. It communicates with applications through the Northbound Interface NBI and controls network devices like switches through the SBI.

The controller is responsible for several important tasks:

- It decides the best path for data packets.
- It monitors traffic and collects statistics.
- It installs or updates flow rules on switches.
- It manages network security policies, such as access control or firewall rules.
- It handles failures by quickly changing routing paths when a link is down.

All these actions help the network become more dynamic, flexible, and easy to manage [10].

Popular SDN Controllers:

a. **ONOS** is made for large and critical networks. It is a distributed controller that supports scalability and high availability. ONOS allows multiple instances to run at the same time, sharing tasks and avoiding single points of failure [14].

b. **OpenDaylight (ODL)** is a flexible controller supported by the Linux Foundation. It is built using a modular architecture, which makes it easy to add plugins or support for new protocols. It is suitable for enterprise and service provider environments [15].

c. **Ryu** is a Python-based controller that is simple and user-friendly. It is mainly used in research and academic projects. Ryu supports OpenFlow and provides tools to help developers quickly build SDN applications [16].

d. **Floodlight** is written in Java and supports OpenFlow. It is often used in data centers and enterprise networks. It is easy to install and has good performance [10].

Each of these controllers supports OpenFlow and offers API to help developers manage the network more easily.

Centralized vs Distributed Controllers:

- A centralized controller runs on a single server and manages the entire network. This is simple to set up but can become a problem if the server fails or if the network becomes too large.
- A distributed controller, like ONOS or OpenDaylight, runs across several servers. These controllers work together to manage the network, improving reliability, speed, and load balancing. If one server fails, others can continue the work [14] [15].

Distributed controllers are better for big networks, while centralized ones can be used in smaller systems.

I.4.3.2 SDN Switch

Modern Ethernet switches and routers use flow tables to perform forwarding functions based on information in packet headers from layers 2, 3, and 4. Although each vendor has different flow tables, there is a common set of functions used by many switches and routers. This common set is used by OpenFlow to program the forwarding or routing logic of the switch.

In SDN, all forwarding devices are often called switches. In this context, a switch is any device that compares packet headers with flow tables, whether it uses Media Access Control (MAC) addresses (Layer 2), IP addresses (Layer 3), or multiple header fields. The term "switch" is used because these devices mainly handle packet forwarding. [10]

The main functions of an SDN switch are:

1)Control Support Function: This allows the switch to communicate with the SDN control layer to support programmability through resource interfaces. The switch talks to the controller, and the controller manages the switch using the OpenFlow protocol.

2)Data Forwarding Function: The switch receives incoming data flows from other network devices or end systems and forwards them along a path that was calculated and set by SDN applications. These forwarding rules are defined by the controller and installed in the switch's forwarding tables.

The forwarding tables tell the switch how to handle different types of packets, such as where to send them next. The switch can also modify packet headers or drop packets if needed.

A typical SDN switch has three ports, one for control communication with the SDN controller, and two for data packet input and output.

One of the most used SDN switches is Open vSwitch (OVS). It is a virtual multilayer switch licensed under Apache 2.0. OVS is designed for large-scale network automation through programmable extensions. It supports many standard management protocols and interfaces such as OpenFlow, Network Flow (NetFlow), Sampled Flow (sFlow), IP Flow Information Export (IPFIX), and Remote Switched Port Analyzer (RSPAN). Also, OVS is made to work across multiple physical servers to provide distributed switching [17] An overview of the Open vSwitch architecture is illustrated in **Figure I.3** [10].

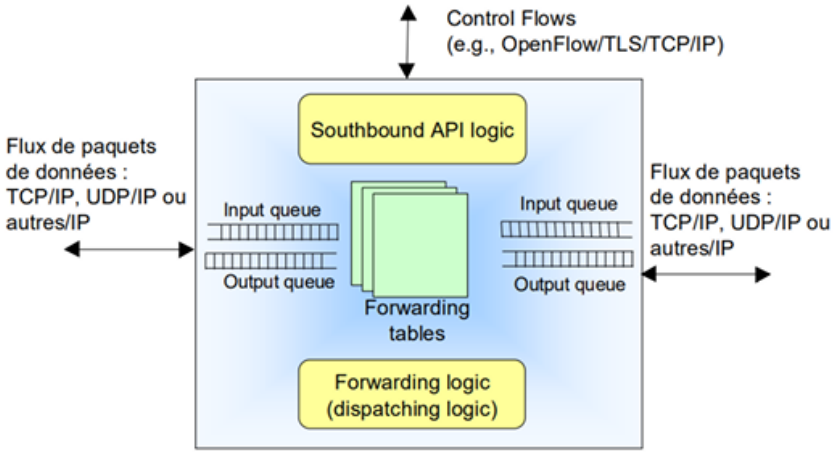


Figure I.3: Open vertual switch

I.4.4 OpenFlow

OpenFlow is one of the earliest and most significant communication protocols designed for SDN. It plays a central role in enabling the separation between the control plane and the data plane. Rather than embedding decision-making logic in the forwarding devices, OpenFlow allows a centralized controller to manage and control network behavior by directly interacting with network switches [9].

I.4.4.1 How OpenFlow Works

An OpenFlow-enabled switch includes the following components:

Flow Tables: These tables contain flow entries with match fields (e.g., IP addresses, port numbers), counters (e.g., number of packets), and actions (e.g., forward, drop, or modify packets).

Secure Channel: A protected link used to exchange control messages between the switch and the controller.

OpenFlow Protocol Handler: Responsible for managing the commands and responses according to the OpenFlow specification.

The working process is as follows:

- When a packet arrives at the switch, it is matched against the entries in the flow table.
- If a match is found, the corresponding action is applied to the packet.
- If no match exists, the switch sends a Packet-In message to the controller.
- The controller analyzes the packet and responds with a Flow-Mod message to install a new rule, ensuring similar packets are processed locally in the future.

This architecture allows for dynamic, real-time control of network traffic and facilitates automated decision-making by the controller [10].

The packet forwarding procedure with OpenFlow is illustrated in **Figure I.4** [18].

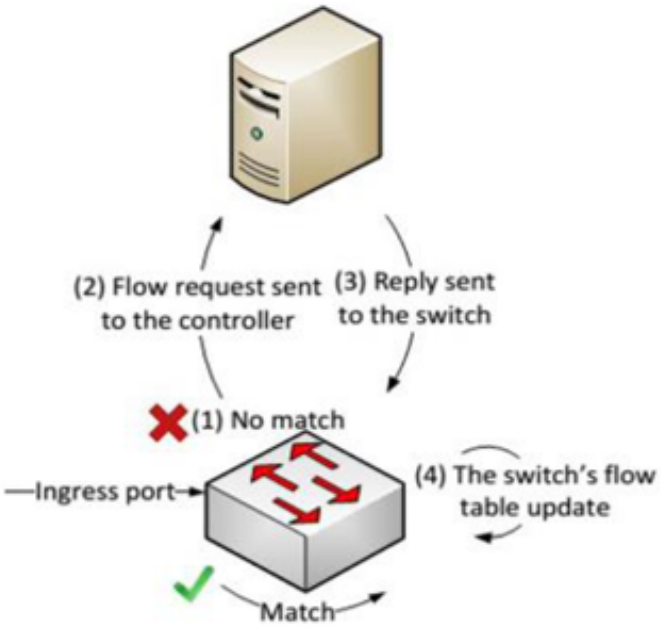


Figure I.4: Packet forwarding procedure using OpenFlow

I.4.4.2 Advantages of Using OpenFlow

OpenFlow brings many benefits to network management. It lets the controller make all the main decisions, which helps manage the network better and faster. The controller can change how traffic moves based on real-time needs, improving network performance. Also, OpenFlow makes switches simpler because they only follow the controller's rules, which reduces hardware costs and complexity. It also allows network operators to update and program the network easily without changing the physical devices [9].

I.5 Conclusion

This chapter has established a foundational understanding of networking paradigms, beginning with traditional architectures and their limitations in handling modern network demands. The evolution toward virtualized environments and software-defined networking has been explored, emphasizing the structural and functional advantages SDN offers through centralized control, automation, and programmability. We introduced SDN's layered architecture, key components such as controllers and switches, and the role of OpenFlow in facilitating communication between the control and data planes. Additionally, the constraints of conventional routing algorithms in SDN contexts were examined, highlighting the need for more adaptive and intelligent solutions. By laying this groundwork, the chapter prepares the reader for the subsequent exploration of Reinforcement Learning (RL)-based approaches, which aim to further optimize SDN performance through adaptive, data-driven routing strategies.

CHAPTER II

STATE-OF-ART OF MACHINE LEARNING AND REINFORCEMENT LEARNING

II.1 Introduction

Machine Learning (ML) has become a key technology for solving complex problems by learning from data instead of being manually programmed. In the context of computer networks, ML helps build smart systems that can make decisions, predict outcomes, and optimize performance based on past observations. This chapter introduces the main ideas of machine learning and explains different types of learning such as supervised, unsupervised, and semi-supervised learning. It also describes how machine learning models are built, trained, and evaluated. The second part of the chapter focuses on Reinforcement Learning , a specific branch of ML where agents learn through interaction with an environment. Special attention is given to Q-learning, Deep Neural Networks DNNs, and Deep Q-Networks Deep Q-Networks (DQNs), which are used in this project. The chapter concludes by discussing why RL is a good choice for SDN routing problems.

II.2 Machine Learning

Machine Learning is a field of study that focuses on enabling computers to improve their performance on specific tasks by learning from data or prior experience. Instead of relying on manually programmed rules, ML systems adjust the parameters of a model through a training process that improves performance using examples. These models can serve different goals: some aim to identify patterns in data (descriptive models), while others are designed to make predictions about future events (predictive models).

To achieve this, ML relies heavily on statistical methods to build mathematical models for data analysis. At the same time, computer science plays a key role in two main areas. First, it provides the infrastructure and tools needed to manage and process large training datasets. Second, it

ensures that the algorithms used for applying trained models are computationally efficient. In many real-world applications, the speed and resource usage of an algorithm are as important as its predictive accuracy [19] .

II.2.1 Types of Machine Learning

Given the variety of machine learning approaches, it's useful to classify them using key foundational criteria. One way to distinguish them is by determining whether the training process and label assignments are supervised by humans or not. Another distinction lies in how the model makes predictions—either by learning patterns from training data (model-based learning) or by directly comparing new data points with known examples (instance-based learning). These classification principles give rise to different ML types [20] , which are further explained in the following sections and summarized in **Figure II.1** [21].

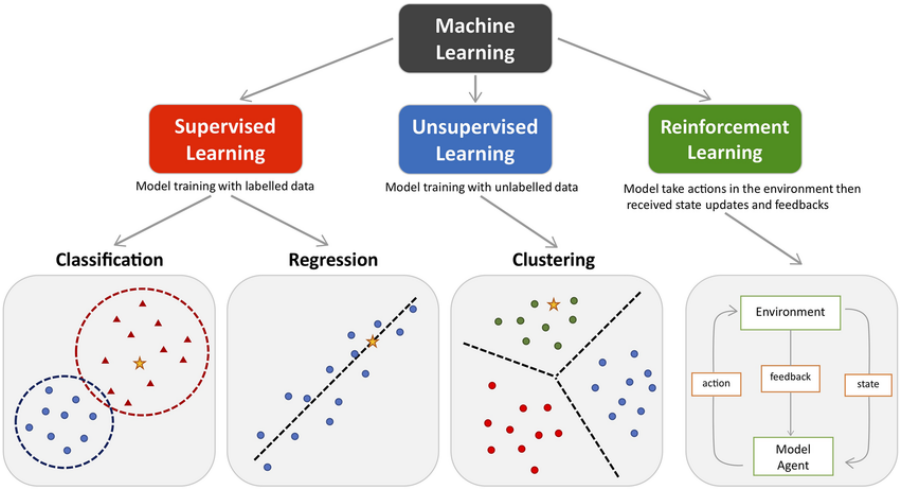


Figure II.1: Machine Learning techniques

a. Supervised Learning

Supervised learning is a fundamental machine learning technique where models are trained on datasets that contain both input data and the corresponding correct outputs. The goal is to learn a function that can generalize to new, unseen data by accurately mapping inputs to outputs. This approach includes two main tasks: classification, where the model predicts categories (such as spam detection), and regression, where it predicts continuous values (like price estimation). During training, the model updates its internal parameters to minimize prediction error using optimization techniques such as gradient descent. Performance is typically evaluated using metrics like accuracy, precision, or mean squared error. Supervised learning is widely applied in fields like speech recognition, medical diagnosis, and fraud detection, where labeled data is available and reliable predictions are essential [22].

b. Unsupervised Learning

Unsupervised learning is a key machine learning method where models are trained on data without labeled outputs. Unlike supervised learning, the algorithm must discover patterns and structures on its own, making it useful for exploring data and gaining insights. Common tasks include clustering, which groups similar data points, and dimensionality reduction, which simplifies datasets by reducing the number of features while keeping important information. Since there are no target labels, evaluation relies on internal metrics or how useful the results are in later applications. Unsupervised learning is widely used in anomaly detection, recommendation systems, market segmentation, and data exploration, especially when labeled data is unavailable or expensive to obtain [23].

c. Semi-supervised Learning

Semi-supervised learning is a machine learning approach that combines a small amount of labeled data with a large amount of unlabeled data. This is useful when labeling data is costly or time-consuming, but unlabeled data is easy to collect. The algorithm learns general patterns from the unlabeled data and uses the labeled data to guide and improve learning. Common techniques include pseudo-labeling, where temporary labels are assigned to unlabeled data, and graph-based methods that spread label information among similar data points. Co-training is another method where two models help each other learn. Semi-supervised learning works well when the data forms clear clusters or lies on a structured surface (manifold). It is widely used in image and speech recognition, medical diagnosis, and text classification, helping build efficient models with less labeled data [24] [25].

d. Reinforcement Learning

RL is a paradigm of machine learning in which an agent learns to make decisions by interacting with an environment, receiving rewards or penalties for its actions, and adjusting its behavior to maximize cumulative reward over time. Unlike supervised learning—which relies on labeled example pairs—RL agents discover optimal policies through trial-and-error, guided solely by scalar feedback signals. The interaction is typically modeled as a Markov Decision Process (MDP), defined by the tuple (S, A, P, R, γ), where S is the state space, A the action space, $P(s' \mid s, a)$ the transition probability, $R(s, a)$ the reward function, and $\gamma \in [0, 1]$ the discount factor that balances immediate versus future rewards [26].

RL has been successfully applied to a wide range of domains, from playing board games at superhuman levels to robotics and autonomous control. Its key strength lies in *model-free* methods—such as Q-learning—which do not require prior knowledge of P or R, and instead learn optimal action-value estimates directly through experience [26].

II.2.2 Model building techniques

Machine learning models are commonly described as "black boxes" because their decision-making process is often difficult to interpret. This lack of transparency can limit their use in sensitive areas like autonomous driving or financial systems. To address this challenge, developers are exploring visual analytics techniques to make ML models more understandable by exposing and illustrating their inner logic.

Developing effective models requires a solid grasp of how the model behaves and the ability to detect problems when training doesn't go as expected or when performance is lacking. Since improving model accuracy is an ongoing and demanding task, better tools for guiding the model development process are necessary.

To respond to these needs, researchers have introduced several visual analytics methods aimed at improving model interpretation, debugging, and control. Understanding tools are designed to visually explain how a model works, how parameter changes impact its behavior, and why it produces specific outputs. Diagnosis tools help identify training issues by allowing interactive inspection of the learning process. Finally, steering methods aim to directly influence the model's training in order to boost its overall performance [27]. Before building a machine learning model, two critical steps must be carried out: data processing and feature engineering. These steps are essential because, in practice, models tend to deliver poor results when trained on low-quality data or poorly selected features. Ensuring clean, relevant, and well-structured input significantly improves the performance of the final model [28] [29].Low data quality can result from missing values, outliers, or noisy labels in the dataset. On the feature side, problems may include irrelevant or duplicate features that don't contribute to model performance. Manually fixing these issues is time-consuming, and fully automated tools often fall short. To address this, various visual analytics methods have been developed to ease the workload for experts while improving the accuracy and efficiency of automated data and feature preparation [30].

II.2.3 Strategy of Machine Learning

Machine Learning allows systems to generate predictive models by learning patterns from historical data and using them to make forecasts when given new input. The accuracy of these predictions generally improves as more data is used. Instead of writing complex rules by hand, we can provide raw data to flexible algorithms that learn how to solve the problem on their own. This shift has transformed how we tackle difficult tasks. A simplified block diagram in **Figure II.2** shows how a typical ML algorithm functions [31].

- Feature engineering is a key step during the pre-processing phase of machine learning. It focuses on extracting meaningful features from raw input data. This step improves the model's accuracy on unseen data by helping it better represent the underlying problem. The goal of feature engineering is to select the most relevant input variables—known as predictors—that will be used by the model, which typically consists of a set of predictors and a target variable [31].

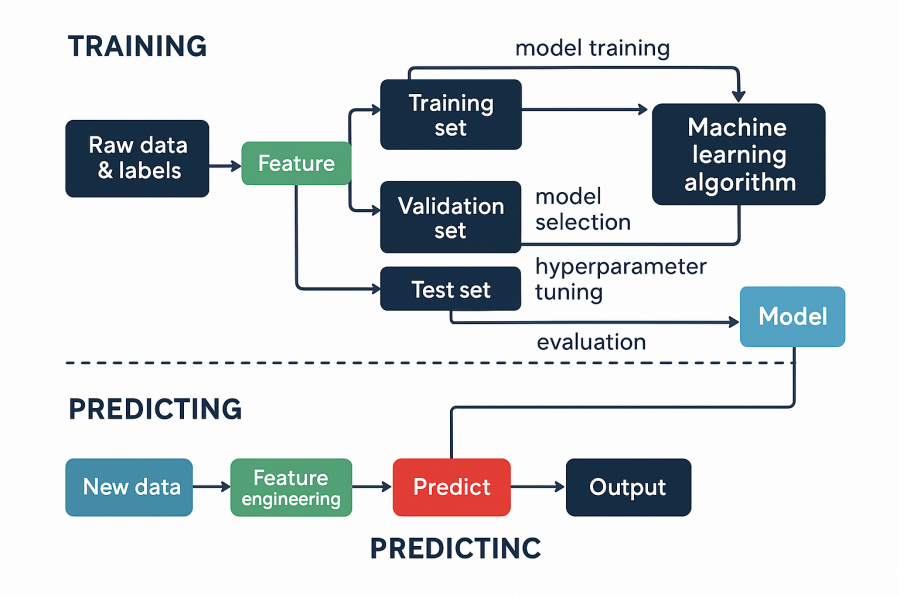


Figure II.2: ML's approach

- Model Training is a core stage in the data science development cycle. During this phase, practitioners work to adjust the weights and biases of a machine learning algorithm to find the best fit. The objective is to reduce the loss function across the entire prediction space, ensuring the model learns to make accurate outputs [32].

- Hyperparameters are user-defined settings in machine learning that control how the learning process unfolds. These values are set before training begins and help guide model performance. Common examples include batch size, the ratio used for splitting data into training and testing sets, the number of training epochs, the number of branches in a decision tree, and the learning rate for training neural networks [31].

- Evaluation refers to the step where a machine learning model's performance is measured using specific indicators. Various metrics are applied for this purpose, including accuracy, precision, recall, F1-score, mean squared error, area under the curve (AUC), and the confusion matrix, among others. These metrics help determine how well the model performs on unseen data and whether it's suitable for deployment [33] .

- Prediction is the result generated by a machine learning model after being trained on historical data and then used on fresh inputs to estimate the probability of a certain outcome [34].

II.2.4 Model evaluation and post-building

To build confidence in ML results, especially for domain experts, visual analytics methods have been developed to make model outputs easier to interpret. These tools are designed mainly for end users and aim to simplify output understanding without exposing the internal mechanics of the model. Techniques are categorized based on the data type they analyze. Given the importance of time in data analysis, methods are generally divided into those handling static results and those

tracking changes over time. Static approaches explore the overall structure, while dynamic ones help users follow how outputs evolve throughout the analysis process [27].

II.3 Reinforcement Learning

II.3.1 Fundamentals of Reinforcement Learning

RL is all about learning by doing. An agent—think of it as a decision-making program—continually interacts with its environment, which could be a network, a game, or any system where actions have consequences. At each moment:

- The agent observes what's happening (the current **state**).
- Based on that observation, it chooses an **action** (for example, forwarding a packet along a certain path).
- It executes the action and the environment reacts, moving to a new state.
- Finally, the agent receives a **reward** (a simple number) that tells it how good or bad that action was.

Over time, the agent's aim is to discover a **policy** (π) (a rule that tells it which action to take in each state) that maximizes its total payoff. Rather than simply adding up every reward it ever receives, RL "discounts" future rewards so that immediate benefits count a bit more than distant ones. Mathematically, if you start at time t, your total discounted return is

$$G_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \cdots \tag{II.1}$$

where the discount factor γ (between 0 and 1) controls just how much you care about the future versus the present [26]

To reason about these interactions in a clean way, we use the concept of a **Markov Decision Process (MDP)**, which boils everything down to five ingredients:

a. **States** (S): All the different situations the environment can be in.

b. **Actions** (A: All the choices the agent can make.

c. **Transitions** (P): The probabilities of moving from one state to another after an action.

d. **Rewards** (R): The expected "score" the agent gets for taking a given action in a given state.

e. **Discount factor** (γ): That same number between 0 and 1 that shrinks future rewards.

The “Markov” part simply means the future depends only on the present state and action, not on the sequence of previous events. This interaction model is visually summarized in **Figure II.3** [35], which illustrates how the agent observes states, selects actions, receives rewards, and transitions through the environment. Thanks to the Markov property, many RL algorithms—including model-free ones like Q-learning—can steadily improve their policy estimates and, under the right conditions, converge to the best possible strategy [36].

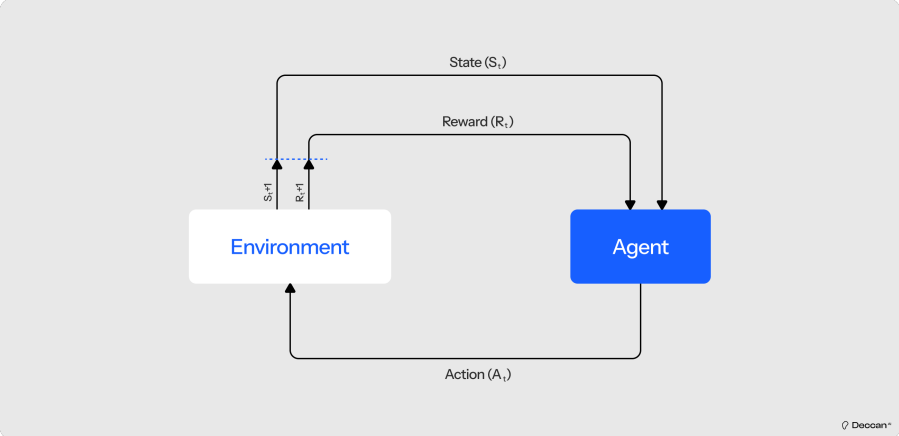


Figure II.3: Example of RL scenario

II.3.2 Q-learning Algorithm

Q-learning is a widely used model-free reinforcement learning algorithm that enables an agent to learn optimal behavior in a Markov Decision Process (MDP) without needing a model of the environment. The core idea of Q-learning is to learn a value function, known as the **Q-value**, which estimates the expected cumulative reward of taking a given action in a given state and following the best policy thereafter.

a. Q-table Concept

The algorithm maintains a Q-table, which is a two-dimensional table where each entry $Q(s, a)$ corresponds to the expected return of performing action a in state s. Initially, all Q-values are often set to zero. As the agent interacts with the environment, the Q-values are updated based on the feedback (reward) received. This table gradually reflects the agent’s learned policy by estimating which actions yield the highest long-term rewards in each state. An example of such a table structure is shown in **Table II.1**, which illustrates how Q-values are organized and updated during the learning process.

b. Bellman Update Rule

The Q-values are updated using the **Bellman equation**, which in the case of Q-learning takes the following form:

Table II.1: Q-Value Table

	Action 1	Action 2	...	Action N
State 1	$Q(S_1,A_1)$	$Q(S_1,A_2)$	...	$Q(S_1,A_N)$
State 2	$Q(S_2,A_1)$	$Q(S_2,A_2)$	...	$Q(S_2,A_N)$
⋮	⋮	⋮	⋱	⋮
State M	$Q(S_M,A_1)$	$Q(S_M,A_2)$	...	$Q(S_M,A_N)$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right] \tag{II.2}$$

Here:

- α is the **learning rate**
- γ is the **discount factor**
- r_{t+1} is the **immediate reward**
- $\max_{a'} Q(s_{t+1},a')$ is the **maximum predicted Q-value** for the next state

This update rule allows the agent to iteratively approximate the optimal Q-values even without knowing the environment's transition probabilities.

c. Key Parameters

Q-learning relies on three critical parameters:

- α (**learning rate**): Controls how much new experiences affect the learned Q-values.
- γ (**discount factor**): Determines how much future rewards are valued.
- ε (**exploration rate**): Balances **exploration** (trying new actions) and **exploitation** (choosing the best-known action). It is used in the ε**-greedy policy**, where the agent chooses a random action with probability ε and the best-known action with probability $1-\varepsilon$.

Through repeated interactions and updates, the Q-learning algorithm converges to the **optimal Q-function**, assuming all state-action pairs are visited infinitely often under a suitable exploration policy [37].

d. Exploration vs. Exploitation

In reinforcement learning, an agent needs to make a smart balance between two things: exploring new actions to learn more about the environment, and exploiting what it already knows to get the best rewards. A common method is called ε-greedy: most of the time the agent picks the best-known action, but sometimes (with a small chance ε) it tries something new. At the beginning of

training, the agent explores a lot, but over time, it explores less and focuses more on actions that give high rewards. This gradual change is called a *decay schedule*. Other methods like *softmax* use probabilities to choose actions based on how good they are. Finding the right balance between exploration and exploitation is key to helping the agent learn well and avoid getting stuck with bad choices [26, 36].

II.3.3 Deep Neural Networks

DNNs are a type of machine learning model that work like a simplified version of how the human brain processes information. They are made up of many layers of "neurons," and each layer learns something more complex from the data. DNNs are very good at finding patterns in complicated data, like pictures or sounds [38].

In networking and reinforcement learning, DNNs help solve problems that are too large for simple methods. For example, instead of storing all possible actions in a table, a DNN can guess the value of an action in a certain situation. This helps when the number of possible situations is very large [39].

To train a DNN, we give it some input and let it make a prediction (forward pass). Then, we compare the prediction to the correct answer and adjust the internal values to make it better next time (backward pass). This process uses something called gradient descent. The design of the DNN—like how many layers it has, how fast it learns, and what activation function it uses—can affect how well it performs [38], [40].

II.3.4 Deep Q-Network

A DQN is a type of algorithm that combines Q-learning with deep neural networks. Regular Q-learning uses a table to save all possible situations and what actions to take, but this does not work well when the problem is big. DQN solves this by using a DNN to estimate the Q-values, which represent the expected reward for each action in a given state [39].

DQN also adds two helpful tricks to make learning more stable: experience replay and target networks. Experience replay stores past learning examples and randomly uses them during training, so the learning is more balanced. The target network is a separate copy of the main model that updates slowly, which helps prevent unstable changes during learning [39], [40].

Because of these features, DQN is useful in situations like SDN routing, where the environment is always changing and the agent needs to adapt.

II.3.5 Why Reinforcement Learning for SDN Routing

Reinforcement Learning is a good fit for SDN routing because it can adapt to changes in the network in real time. Traditional routing algorithms like Dijkstra are static—they find the shortest path but do not adjust well when the network changes, such as when there is congestion or a link

failure. In contrast, an RL-based approach allows the SDN controller (acting as the agent) to learn from the network's feedback and improve its routing decisions over time. By observing the state of the network (such as delay, packet loss, or link usage) and receiving feedback in the form of rewards (e.g., low latency or high throughput), the RL agent can learn which paths work best under different conditions. This makes RL useful for building smarter, more adaptive routing systems that optimize performance even in complex and dynamic network environments [41].

II.4 Conclusion

In this chapter, we explored the fundamental concepts of Machine Learning and its different learning types and strategies. We explained the processes involved in building and evaluating ML models, followed by a deeper focus on Reinforcement Learning techniques. Key topics such as Q-learning, Deep Neural Networks, and Deep Q-Networks were presented to show how learning agents can make decisions in complex environments. Finally, we highlighted the benefits of applying Reinforcement Learning in SDN routing, where adaptability and real-time feedback are essential. This background supports the use of a DRL-based routing agent in our proposed system.

CHAPTER III

IMPLEMENTATION, TESTS AND RESULTS

III.1 Introduction

This chapter presents the practical aspects of our research, focusing on the design, development, and evaluation of both traditional and intelligent routing strategies within a Software-Defined Networking (SDN) environment. We begin by detailing the virtual setup, including the network topology, the SDN controller configuration, and the testing tools employed. A baseline implementation using Dijkstra's algorithm is first introduced to establish a point of comparison. We then delve into the design and training of a Reinforcement Learning (RL)-based routing agent, exploring its architecture, training data, hyperparameters, and interaction with the SDN controller. Subsequently, we evaluate the performance of the RL-based model in a simulated environment and compare it against the baseline approach. The chapter concludes with a comparative analysis between both routing methods in terms of Quality of Service (QoS).

III.2 Environment Setup and Baseline Implementations

This section presents the architecture of the experimental platform used to evaluate the different routing strategies in a Software-Defined Networking environment. The platform is built around ONOS as the SDN controller, Mininet for network emulation, and a custom reinforcement learning module integrated through a Python REST interface.

III.2.1 Virtual Environment

All experiments were conducted in a virtualized environment running Ubuntu 22.04 LTS, hosted inside a VMware virtual machine to ensure portability and controlled isolation. The system was configured with sufficient memory(ROM= 45G ,RAM =4G)and CPU (Ryzen 5 2500-u)resources to support real-time network emulation and machine learning tasks.

The following tools and frameworks were installed:

– ONOS 2.2.4, compiled from source using Bazel

– Mininet 2.2.2, for creating custom network topologies

– Python 3.10, for implementing the RL model and REST service

– Java 11, required by ONOS and the custom Java routing application

– Maven, for building and packaging ONOS applications

This environment enabled the deployment and testing of both default and custom routing strategies, ensuring compatibility with the ONOS 2.2.4 framework.

III.2.2 Software-Defined Network Controller and Applications

The SDN control plane was handled by ONOS version 2.2.4, launched through its Karaf-based runtime environment. ONOS was compiled from source using Bazel to enable customization and the integration of additional Java modules. Two key applications were developed and deployed on this architecture:

- **RLRoutingApp:** A custom ONOS application written in Java that implements reinforcement learning-based routing logic. It intercepts incoming packets, extracts network state, queries the RL model for decisions, and installs flow rules accordingly using the ONOS FlowRuleService.
- **Python Flask RL Service**: A lightweight REST server written in Python using Flask, responsible for loading the trained reinforcement learning model (in HDF5 format) and responding to HTTP requests from the RLRoutingApp. It receives a network state vector as input and returns a routing decision (action index) in JSON format.

This separation of concerns between control logic (Java) and intelligent decision-making (Python) allows flexible development and simplifies model training and updates.

III.2.3 Interaction Diagrams

The interaction between system components is organized around three core communication paths:

- Mininet with ONOS: Mininet emulates the data plane (switches and hosts), and communicates with ONOS via the OpenFlow protocol. ONOS receives packet-in events and installs flow rules based on the active application logic.
- ONOS with RL Flask Server: The Java application RLRoutingApp sends POST requests to the Flask server hosted locally on localhost:5001. It transmits a vector of network state features and receives an integer representing the selected routing path.

- ONOS with Network Topology (via REST API): The network topology, including devices, links, and host mappings, is pushed to ONOS through its REST API using Python scripts. This allows preconfiguring the network view used during the experiments.

III.2.4 Network Topology

The network topology defines the structural arrangement of nodes, switches, and links that enable data transmission within the system. This section provides an overview of the logical design and practical deployment steps used to create the experimental testbed.

a. **Topology Description**

The network environment is composed of two distinct LANs—LAN 1 acts as the source, while LAN 2 serves as the destination. Each LAN is equipped with multiple servers, with N servers connected on both sides. From LAN 1, these servers will initiate data flows toward LAN 2, transmitting at variable rates without following any fixed temporal pattern. Both the frequency and duration of these transmissions will vary independently.
The network topology includes eight switches connected by eleven bidirectional links. Most of these links will operate at a theoretical bandwidth of 2 Mbps. However, four specific connections namely S2 to S5, S2 to S7, S4 to S5, and S4 to S7 will have reduced bandwidths of 1 Mbps.

The Open Network Operating System (ONOS) version 2.2.4 serves as the SDN controller managing the network. ONOS provides APIs for topology discovery, flow rule installation, and network monitoring, which are leveraged for implementing routing algorithms.

b. **Topology Deployment**

The deployment of the network topology was carried out using a custom Python script named `topo_rl.py` (You can find the full project on GitHub: `github.com/akram-djerrar`), which leverages the Mininet framework. This script defines all the OpenFlow switches and end-hosts using Mininet's **addSwitch()** and **addHost()** methods, then interconnects them using **addLink()**, assigning a bandwidth of 10 Mbps to each link via the bw parameter. The script also specifies a remote SDN controller by instantiating the **RemoteController** class, pointing Mininet to the ONOS instance. Once all elements are defined, the script calls **net.start()** to launch the emulated network, initialize the links, and enable packet forwarding. Following deployment, the complete network configuration is manually injected into ONOS via its REST API to ensure precise synchronization between the emulated topology and ONOS's internal network model. This configuration is defined in three JSON files: **devices.json** (which lists the switch identifiers and attributes), **links.json** (defining the topology's inter-switch connections), and **hosts.json** (mapping IP/MAC addresses to switch ports). These files are pushed to ONOS using HTTP POST requests .The **Figure III.1** illustrates the deployed topology as visualized within the ONOS GUI, confirming that the device, link, and host configurations were successfully registered through the REST API., such as the following command for devices:

```
curl --user onos:rocks -X POST -H "Content-Type: application/json" \
http://localhost:8181/onos/v1/network/configuration \
--data-binary @devices.json
```

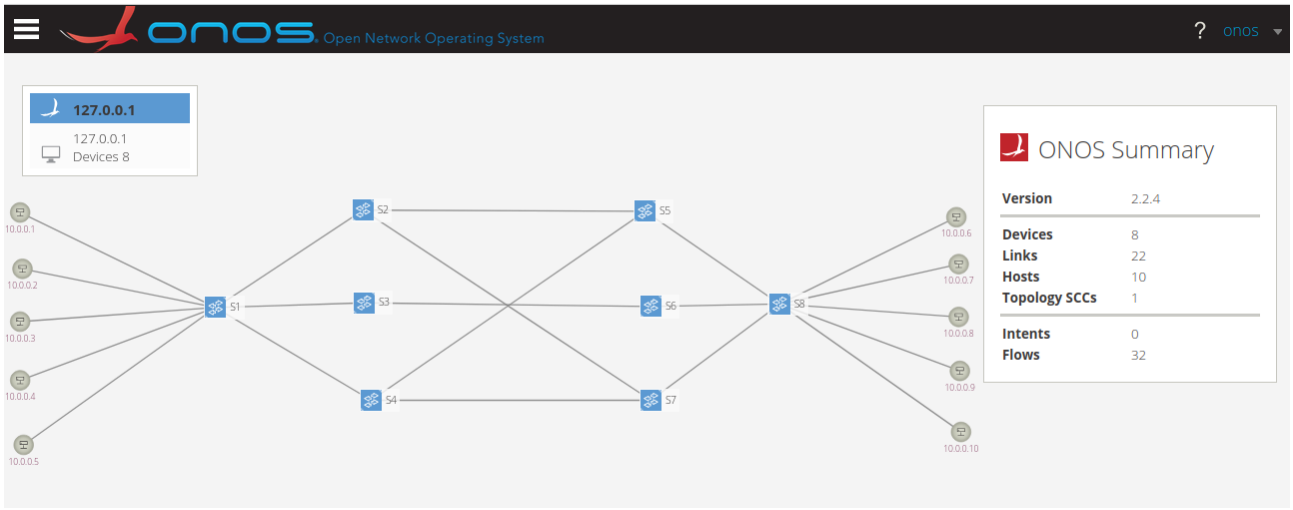


Figure III.1: ONOS GUI view of the complete SDN topology after deployment and REST-based configuration injection

III.2.5 Traffic Generation and Monitoring

a. **Traffic Generation Tools**

D-ITG is a tool that generates IPv4 and IPv6 traffic by simulating real Internet application behavior using random models for packet size and timing. It measures common QoS metrics at the packet level and supports TCP, UDP, ICMP, and other transport protocols [42]. **Figure III.2** [43]shows the D-ITG architecture, made up of five main parts used in this work: ITGSend, ITGRecv, ITGLog, ITGDec, and ITGPlot.

For testing routing in the SDN setup, D-ITG was used to create realistic traffic between multiple hosts. It allows control over flow rates, packet sizes, and timing. The components used are:

- ITGSend: Responsible for initiating and sending traffic flows from source hosts to designated destinations.
- ITGRecv: Deployed on destination hosts to receive and record incoming traffic, ensuring accurate measurement of delivery and loss.
- ITGLog: Collects and stores detailed logs of traffic events, including timestamps, packet counts, and flow identifiers.
- ITGDec: Decodes the raw log files produced by ITGLog, extracting essential metrics such as delay, jitter, and packet loss for further analysis.

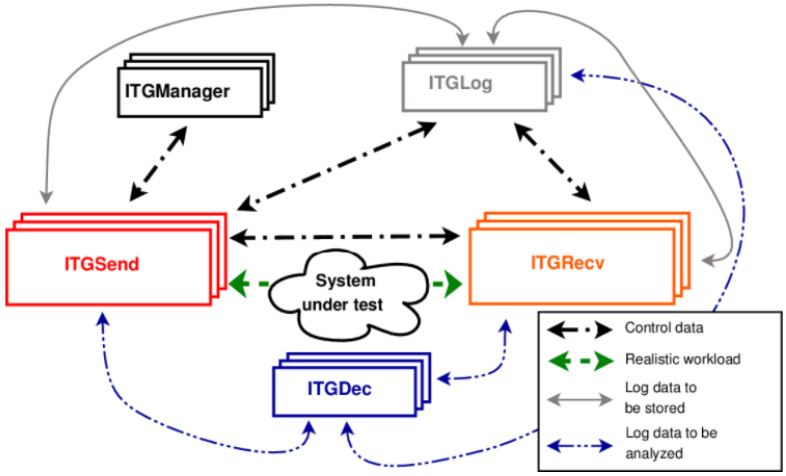


Figure III.2: D-ITG Architecture

- ITGPlot: Visualizes the extracted QoS metrics, enabling clear graphical representation of network performance under different routing scenarios.

b. **Traffic Patterns and Load Scenarios**

To ensure the experiments reflect realistic and dynamic network conditions, a variety of traffic patterns and load scenarios were designed using D-ITG. Example flows included:

– Host-to-host flows: Such as traffic from host1 to host8 and host2 to host9, representing typical end-to-end communication in enterprise or campus networks.

– Constant bit rate (CBR) flows: Flows were configured at fixed rates (e.g., 500 Kbps) to evaluate routing performance under steady traffic loads and to facilitate reproducibility of results.

– Multi-flow scenarios: Multiple concurrent flows were generated between different host pairs to simulate network congestion and test the adaptability of the routing algorithms.

c. **Quality of Service Metrics Collection**

Comprehensive QoS measurement was essential for evaluating and comparing the routing strategies. The following data collection methods were used:

– Link utilization and flow statistics: Real-time statistics on link occupancy, active flows, and port usage were gathered via the ONOS REST API. This enabled monitoring of network resource usage and identification of overloaded links during experiments.

– Packet loss, delay, jitter, and throughput: These critical QoS metrics were extracted directly from D-ITG log files. After each traffic generation experiment, ITGLog captured detailed flow records, which were decoded by ITGDec to produce quantitative measurements of:

- Packet loss: The percentage of packets not delivered to the intended destination.
- Packet rate: measures the number of data packets transmitted or received per second (pps) over a network.

- Delay: The average end-to-end time taken by packets to traverse the network.
- Jitter: The variation in packet delay, indicating network stability.
- Throughput(Bitrate): The rate of successful data delivery over the network, measured in Mbps.

III.2.6 Performance Metrics Explanation

The following key performance indicators were extracted from the decoded D-ITG logs to evaluate and compare the routing strategies:

- **Average Delay (s):** The mean time it takes for packets to travel from source to destination.
- **Maximum / Minimum Delay (s):** The extreme latency values recorded during the test.
- **Jitter (s):** The variation in packet delay, indicating the stability of transmission.
- **Delay Standard Deviation (s):** The degree to which individual delay measurements deviate from the average delay.
- **Throughput / Bitrate (Kbit/s):** The average rate at which data is successfully received.
- **Packet Rate (packets/s):** The number of packets received per second.
- **Packets Dropped (%):** The total number and percentage of packets lost during transmission.
- **Loss-Burst Size:** The average number of consecutively lost packets in a single burst.

These metrics together provide a comprehensive view of network Quality of Service (QoS) performance.

III.3 Dijkstra's Algorithm Implementation

a. **ONOS 2.2.4 Controller and Its Forwarding Application**

The Open Network Operating System (ONOS) version 2.2.4 is a widely used open-source SDN controller platform designed to provide high availability, scalability, and performance for programmable networks. ONOS offers a modular architecture based on OSGi bundles, allowing developers to build and deploy network applications efficiently [44] .

One of the core applications included by default in ONOS 2.2.4 is the reactive forwarding application, identified by the package name org.onosproject.fwd. This application implements a Shortest Path (SP) routing strategy using Dijkstra's algorithm with equal link weights. It is designed to reactively install forwarding rules in the network switches upon the arrival of new traffic flows [45].

b. **Functionality of org.onosproject.fwd**

– Reactive Forwarding: The application listens for incoming packets that do not match any existing flow rules. Upon receiving such packets, it computes the shortest path from the source to the destination host using Dijkstra's algorithm, assuming all links have equal cost [45] [46].

– Flow Rule Installation: Once the shortest path is determined, the application installs flow rules along the path in the switches, enabling subsequent packets of the same flow to be forwarded without controller intervention [45] [47].

– Topology Awareness: The application leverages ONOS's topology service to maintain an up-to-date network graph, enabling accurate path computations [48].

– Integration with ONOS Services: It uses ONOS's `Flow Objective API` particularly the `DefaultForwardingObjective.` class, to create and manage forwarding objectives. This abstraction simplifies flow rule management across heterogeneous network devices and protocols [48].

Upon receiving such packets, it computes the shortest path from the source to the destination host using Dijkstra's algorithm 1 [49], assuming all links have equal cost .

Algorithm 1: Dijkstra's Algorithm for Shortest Path Routing

Require: Network graph $G = (V, E)$ with equal link weights, source node s
Ensure: Shortest distances from s to all other nodes

for each node $v \in V$ **do**
  $dist[v] \leftarrow \infty$
  $prev[v] \leftarrow$ **undefined**
end for
$dist[s] \leftarrow 0$
$Q \leftarrow V$
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ node in Q with smallest $dist[u]$
  remove u from Q
  for each neighbor v of u **do**
    $alt \leftarrow dist[u] + weight(u, v)$
    if $alt < dist[v]$ **then**
      $dist[v] \leftarrow alt$
      $prev[v] \leftarrow u$
    end if
  end for
end while
return $dist, prev$

The algorithm show that Dijkstra's algorithm begins by initializing the distance to all nodes in the network as infinite, except for the source node, which is set to zero, indicating that no cost is needed to reach itself. A set of unvisited nodes is maintained throughout the process. The algorithm then repeatedly selects the unvisited node with the smallest known distance (i.e., the closest node discovered so far), marking it as visited and finalizing its shortest path

from the source. For each neighbor of this node, it calculates an alternative path cost by adding the current distance to the weight of the edge connecting to the neighbor. If this alternative path is shorter than the previously recorded distance, the algorithm updates the shortest distance and stores the current node as the previous hop in the path. This process continues until all nodes have been visited. At the end, two data structures are returned: one containing the shortest distances from the source to every node, and another containing the previous-hop information needed to reconstruct the actual shortest paths. This method guarantees the optimal path in graphs with non-negative link weights and is used in ONOS's reactive forwarding logic with all links considered to have equal cost.

c. **The Role of org.onosproject.fwd in ONOS**

– Default and Stable: Being a default ONOS application, it is well-tested and stable, providing a reliable baseline for routing performance [50].

– Simplicity: The equal-weight shortest path approach is straightforward, making it a clear benchmark against which to compare more advanced routing strategies [45].

– Reactive Nature: Its reactive installation of flows mimics typical SDN behavior, allowing evaluation of flow setup delays and controller load [46].

d. **ONOS 2.2.4 Controller Features Relevant to Routing**

– Modular OSGi Architecture: Enables dynamic loading and unloading of applications like org.onosproject.fwd [44].

– REST APIs: Provides interfaces for monitoring topology, flow rules, and link statistics, essential for implementing custom routing logic and collecting performance data [50].

– High Availability and Scalability: Supports clustering and distributed control, ensuring robustness for larger network deployments [51].

– OpenFlow Protocol Support: ONOS 2.2.4 fully supports OpenFlow 1.3, the protocol used to program flow tables in switches [44] [47].

e. **Activating and Using org.onosproject.fwd**

To use the reactive forwarding application in ONOS 2.2.4, it can be activated via the ONOS CLI like we see in **Figure III.3** :

```
app activate org.onosproject.fwd
```

III.4 Evaluation of Dijkstra-Based Routing

III.4.1 ONOS Configuration and Topology Setup

To implement and evaluate Dijkstra-based routing, the first step involved configuring the ONOS SDN controller with its built-in reactive forwarding application. The **org.onosproject.fwd** appli-

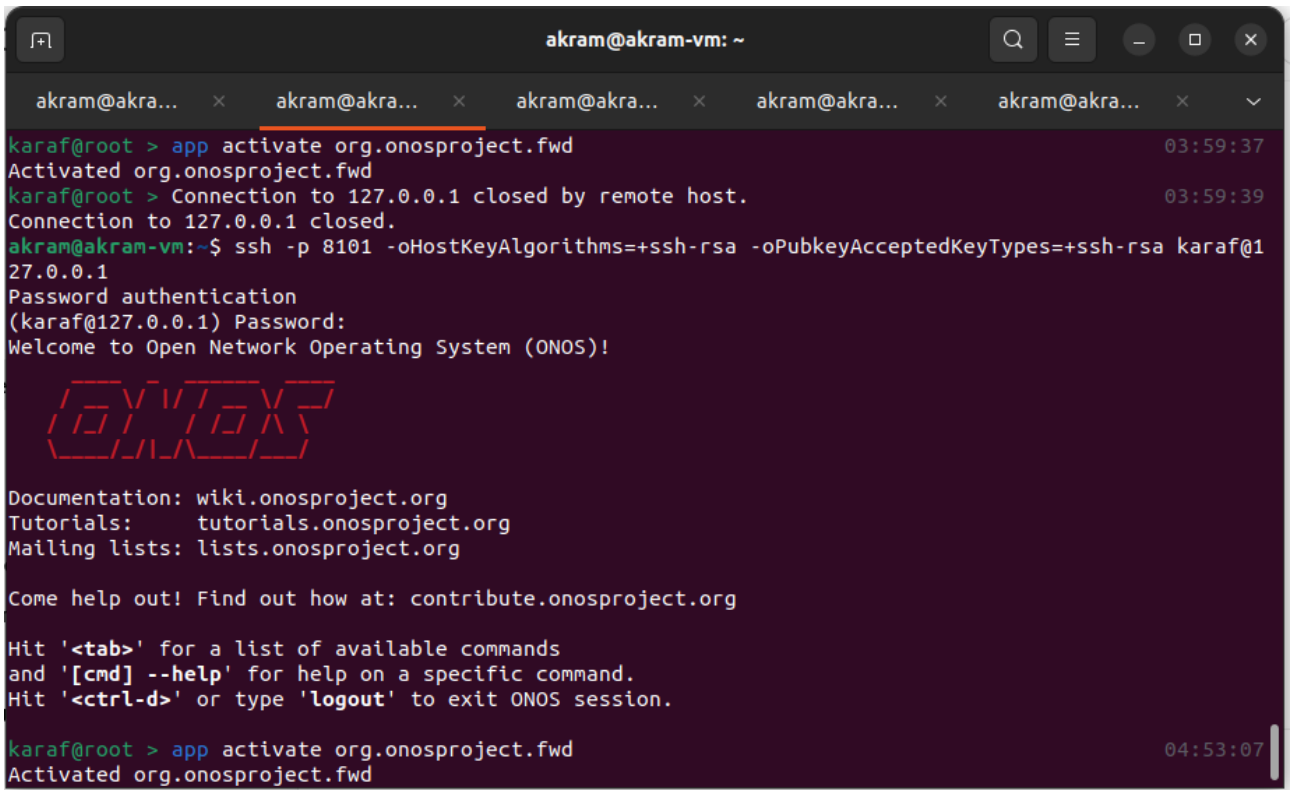


Figure III.3: Activating and Using `org.onosproject.fwd`

cation was activated using the ONOS Karaf CLI. This application utilizes ONOS's internal Dijkstra-based algorithm to compute the shortest path between source and destination hosts upon packet-in events.

An 8-switch, 10-host topology was launched using a custom Mininet script. Each switch in the topology is uniquely identified and interconnected to simulate a realistic, moderately complex network with multiple routing paths between end-hosts. The topology includes redundant paths and link capacities that create conditions for potential congestion — a critical aspect when evaluating the adaptability of different routing algorithms.

Once the topology was instantiated, the ONOS GUI (available at http://localhost:8181/onos/ui) as we seen before in **Figure III.1** , was used to verify that all switches, hosts, and links were correctly discovered. The ONOS controller successfully detected and visualized all 8 OpenFlow switches, the inter-switch links, and the host connections.

To confirm initial network connectivity and proper ONOS behavior, full reachability tests were conducted using the pingall command in Mininet. All hosts were able to communicate with each other, with 100% ICMP reply success across all pairs as we see in **Figure III.4** .This validated the correct operation of ONOS's default forwarding mechanism and confirmed that the network was functioning as expected before traffic generation and performance testing began.

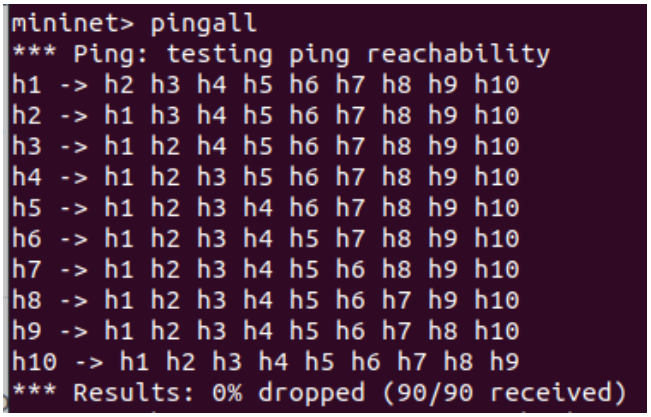


Figure III.4: host-to-host connectivity in Mininet using Dijkstra routing via ONOS's org.onosproject.fwd application.

III.4.2 Traffic Generation and D-ITG Testing

To assess the network's Quality of Service (QoS) under different routing strategies, traffic flows were generated using D-ITG (Distributed Internet Traffic Generator) : a tool specifically designed for testing IP-based networks.

For each test, the destination host runs the ITGRecv component to receive and log incoming traffic:

```
mininet> h1 ITGRecv -l recv_log_h1_h8_dijkstra.log &
```

This command launches the receiver on host *h*8, logging incoming packets to the file `recv_log_h1_h8_rl.log` .Then, on the source host h1, the traffic is generated using:

```
mininet> h3 ITGSend -a 10.0.0.8 -T UDP -C 9000 -c 1400 -t 60000 -x
recv_log_h1_h8_dijkstra.log
```

Each parameter has a specific role:

– **a 10.0.0.9:** Destination IP address (host h9).

– **T UDP:** Traffic type; in this case, UDP.

– **C 9000:** Constant bitrate of 9000 bps (customizable depending on the test).

– **c 1400:** Packet size in bytes.

– **t 60000:** Duration of the transmission in milliseconds (i.e., 60 seconds).

– **x** `recv_log_h1_h8_dijkstra.log` **:** Specifies the log file to store the traffic report on the receiver.

- **Results of Dijkstra Performance**

An example of a partial ITGDec report from Host 1 to Host 8 is described in **Figure III.5**. The report provides many QoS metrics measurements per flow .and some hosts results of D-ITG are summarized in **Table III.1**.

```
ITGDec version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
------------------------------------------------------------
Flow number: 1
From 10.0.0.1:52579
To    10.0.0.8:8999
------------------------------------------------------------
Total time                =     60.379491 s
Total packets             =         10065
Minimum delay             =      0.000159 s
Maximum delay             =      0.719217 s
Average delay             =      0.420626 s
Average jitter            =      0.009109 s
Delay standard deviation  =      0.121276 s
Bytes received            =      14091000
Average bitrate           =   1866.991558 Kbit/s
Average packet rate       =    166.695675 pkt/s
Packets dropped           =             0 (0.00 %)
Average loss-burst size   =      0.000000 pkt
------------------------------------------------------------


****************  TOTAL RESULTS   ******************

Number of flows           =             1
Total time                =     60.379491 s
Total packets             =         10065
Minimum delay             =      0.000159 s
Maximum delay             =      0.719217 s
Average delay             =      0.420626 s
Average jitter            =      0.009109 s
Delay standard deviation  =      0.121276 s
Bytes received            =      14091000
Average bitrate           =   1866.991558 Kbit/s
Average packet rate       =    166.695675 pkt/s
Packets dropped           =             0 (0.00 %)
Average loss-burst size   =             0 pkt
```

Figure III.5: Partial ITGDec dijkstra Report from Host 1 to Host 8.

Table III.1: Performance Metrics for Dijkstra-Based Routing (Using D-ITG)

Flow	Avg Delay (s)	Max Delay (s)	Jitter (s)	Bitrate (Kbps)	Pkt Rate (pkt/s)
h1→h8	0.420626	0.719217	0.009109	1866.99	166.70
h2→h10	0.413901	0.693010	0.007847	1627.16	145.28
h3→h9	0.396899	0.696271	0.007804	1688.90	150.79
h4→h7	0.401003	0.712921	0.007968	1580.81	141.14

The analysis of D-ITG performance metrics under Dijkstra-based routing reveals the following key insights:

a. **Average Delay:** All flows exhibited an average delay between 396 ms and 421 ms, with h3→h9 achieving the lowest delay (396.899 ms). These delays are relatively consistent, indicating that the Dijkstra algorithm provides stable latency across multiple paths. However, the delay values are on the higher end for real-time applications, suggesting some path congestion or suboptimal load balancing in the static routing decisions.

b. **Jitter:** Jitter values remained very low, between 7.8 ms and 9.1 ms, showing minimal delay variation. This is expected in a static routing scenario where the same path is consistently used. h3→h9 again performed best (7.804 ms), reinforcing the stability of that route. Low jitter is favorable for applications sensitive to timing variations (e.g., VoIP, video).

c. **Bitrate(Throughput):** The bitrate varied between 1580 Kbps and 1867 Kbps, with h1→h8 achieving the highest throughput. The variation suggests that even with shortest-path routing, different flows experience differing levels of congestion depending on their specific path and traffic patterns. The relatively high throughput across all flows indicates good overall link utilization.

d. **Packet Loss :** ero packet loss was observed in all flows. This indicates that the network paths selected by ONOS's Dijkstra-based application were sufficiently provisioned in terms of bandwidth and that no congestion led to packet drops during the 60-second test window.

III.5 Reinforcement Learning Model Design and Environment

To integrate reinforcement learning into SDN-based routing, we propose a three-plane system architecture that extends the traditional SDN model by introducing an AI decision layer. As shown in **Figure III.6**, the architecture is composed of the following three planes:

- **Data Plane:** This layer consists of OpenFlow-enabled switches and end-hosts simulated in Mininet. It is responsible for forwarding traffic based on flow rules received from the control plane.
- **Control Plane:** This layer is managed by the ONOS controller, which maintains a global view of the network topology and installs flow rules on the data plane switches. It communicates with the AI plane via REST APIs to obtain routing decisions.
- **AI Plane:** This layer hosts the reinforcement learning agent based on Deep Q-Network (DQN). The agent is trained offline using synthetic traffic patterns and is exposed through a Flask-based API. During online operation, ONOS queries this agent to compute optimal paths based on the current network state (e.g., available bandwidth and flow rate).

This multi-plane architecture enables dynamic and intelligent routing decisions by combining centralized SDN control with adaptive learning-based policies. The AI agent enhances the

SDN controller's capability to avoid congestion and improve QoS by learning from past routing experiences.

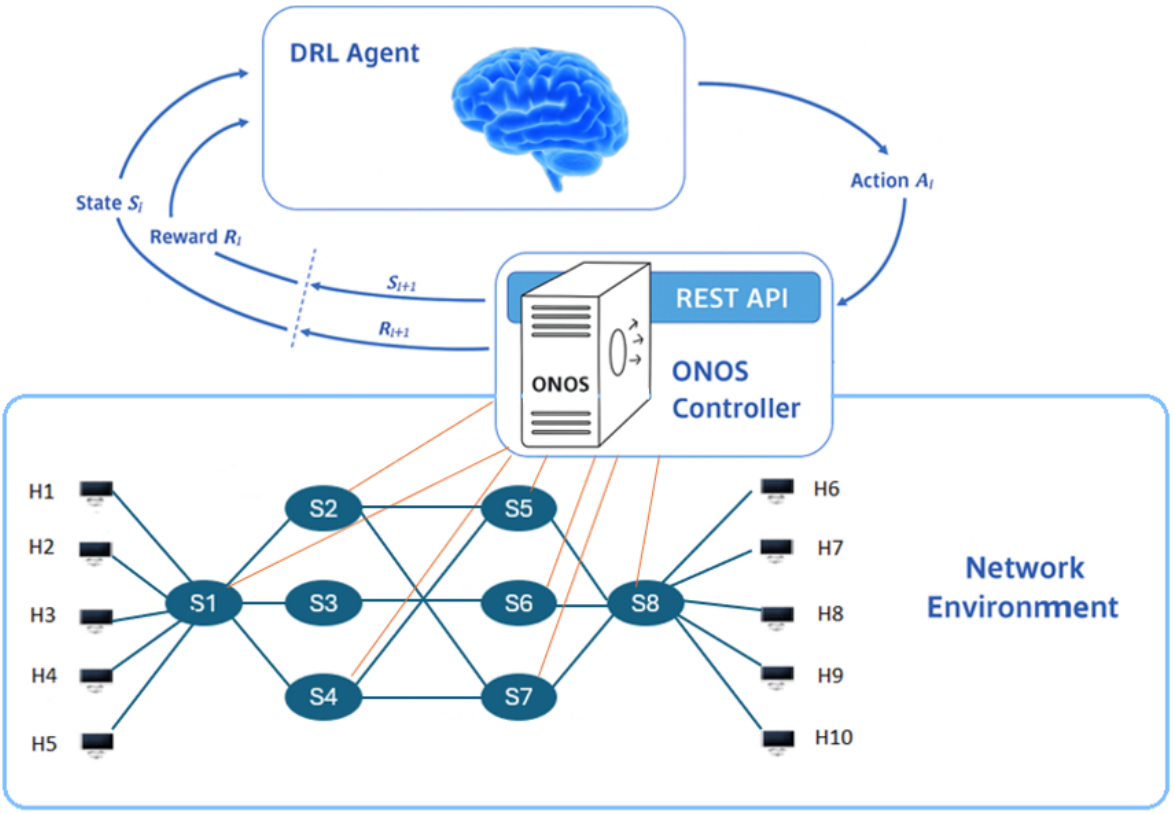


Figure III.6: System architecture integrating SDN and reinforcement learning.

III.5.1 RL Agent Architecture and Training Setup

a. **RL Agent Architecture**

In Artificial Intelligence, particularly in Machine Learning (ML), agents are often modeled using Deep Neural Networks (DNNs). These networks consist of layers of interconnected neurons that mimic biological neural signaling. A typical DNN includes:

– Input layer: Receives the data.

– hidden layers: Process information through weighted connections.

– Output layer: Produces the final prediction or decision.

The flexibility of DNNs allows them to adapt to complex environments, making them suitable for our routing problem.

Our agent's architecture is detailed in **Figure III.7** and consists of the following:

(a) **State Space: Input Layer Design**

The state space defines the information the agent uses to make decisions. In our scenario, the state vector has a size of 23, representing:

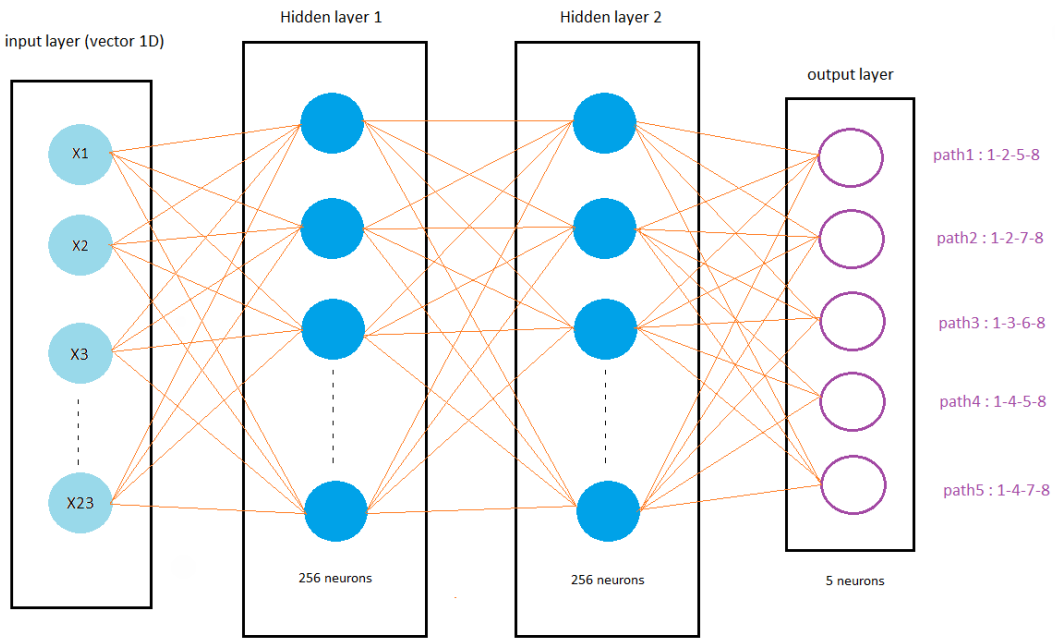


Figure III.7: DNN representation of the RL agent

– 22 values: Corresponding to the available bandwidth on each of the 11 bidirectional links (each direction counted separately).

– 1 value: The rate of the incoming flow requesting routing.

This combination provides the agent with a comprehensive snapshot of current network capacity and demand, enabling informed routing decisions.

The input layer of the DNN corresponds to this state vector. We implemented the network using the Keras library in Python, which facilitates creating and training neural networks.

(b) **Hidden Layers: Dense Layers with ReLU Activation**

For the hidden layers, we chose the dense (fully connected) layer type, where each neuron connects to every neuron in the next layer. This structure allows the network to learn complex relationships between inputs and outputs.

Each hidden layer contains 256 neurons and uses the Rectified Linear Unit (ReLU) activation function. ReLU works by:

– Converting all negative input values to zero.

– Passing positive values unchanged.

This activation helps the network learn non-linear patterns efficiently, as illustrated in **Figure III.8**.

(c) **Output Layer and Action Space**

The output layer represents the agent's possible actions. In our routing problem, the actions correspond to selecting one of the five most significant paths between LAN 1 and LAN 2:

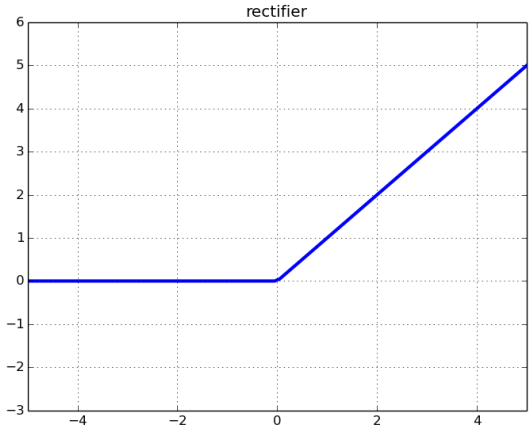


Figure III.8: Representation of the ReLU activation function

– S1 → S2 → S5 → S8

– S1 → S3 → S6 → S8

– S1 → S4 → S7 → S8

– S1 → S2 → S7 → S8

– S1 → S4 → S5 → S8

Thus, the action space size is five.

Unlike the hidden layers, the output layer uses a linear activation function. This choice is because the network predicts continuous numerical values (Q-values) representing the expected reward for each action, rather than classifying inputs into categories.

b. **Reward Function**

The reward mechanism guides the agent's learning by providing feedback on its actions:

– The agent receives a reward of 1 if the selected path successfully routes the flow without exceeding any link's bandwidth.

– A reward of 0 is given if the allocation causes any link to overflow.

This binary reward encourages the agent to avoid congested paths and allocate flows efficiently.

c. **Optimizer Configuration**

Optimizers improve the training process by minimizing the model's loss function, which measures prediction errors. For our project, we use the Adam optimizer, a popular choice for deep learning due to its adaptive learning rate and efficiency.

The learning rate (denoted by α) is set to 0.0003, balancing training speed and stability. This configuration helps the model converge to an optimal routing policy.

III.5.2 Training of the Reinforcement Learning Model

Having understood the fundamentals of Reinforcement Learning (RL) and how it is adapted to our network environment, this section explains how the RL model is trained to allocate incoming flows efficiently in the network.

Due to the complexity and the large number of iterations required to train RL models effectively, we chose to simulate the environment using Python instead of generating flows directly in ONOS with DIT-G.This approach allows faster and more controlled training iterations.

In any RL scenario, the environment is framed as a game with:

– An initial state, where the agent begins.

– A final state, considered a "win" or completion condition.

The agent's actions taken from the initial to the final state form an episode. Episodes are critical because the agent learns by experiencing many such sequences.

In our context, an episode ends when the agent fails to allocate a flow without exceeding link capacity or successfully allocates all flows.

III.5.2.1 Training Dataset Flow Generation and Characteristics

To train the agent, we created a JSON file containing 1,000 flow events randomly distributed over a time span of 40,000 seconds (approximately one new flow every 40 seconds).

– Key characteristics of the flows include: Flow rates limited between 80 Kbps and 248 Kbps, in multiples of eight, yielding 22 distinct rates.

– Each flow entry includes:

- **Unique identifier (hash) :** A distinct numerical code assigned to each specific traffic flow. Crucial for tracking the entire lifecycle of a flow, linking its start (ingress) and end (egress) events together within the simulation.
- **Ingress or egress time :** The exact timestamp (in seconds from simulation start) when a flow event occurs, indicating either the flow's entry (ingress) or departure (egress) from the network.Allows the simulator to process events chronologically, accurately reflecting real-world network dynamics where traffic changes over time.
- **Source and destination hosts :** Identifiers specifying the originating point (source) and the intended endpoint (destination) of the traffic flow within the network topology. Defines the specific routing problem for each flow, informing the Deep Reinforcement Learning (DRL) agent about the path it needs to determine.
- **Start and end times :** Timestamps marking the duration for which a flow is actively transmitting data. Start is when it begins, end is when it finishes. Enables the simulator to manage bandwidth allocation and deallocation precisely, as it knows exactly when resources are consumed and when they are freed. This creates a dynamic environment for the RL agent.

- **Flow rate :** The constant bandwidth demand of the flow, typically in Kilobits per second (Kbps). Directly quantifies the network resource consumption of the flow. It's a key input for the RL agent's decision-making process, as the agent must consider this rate to avoid link congestion and ensure efficient routing.

This dataset simulates realistic traffic with varying flow sizes and durations.The **Figure III.9** shows an excerpt of the JSON flow file,and **Figure III.10** show The temporal distribution of flows grouped by rate.

The average network load is roughly 3,000 Kbps, about 50 of total capacity, calculated considering three 2 Mbps links from switch S1 to S2, S3, and S4 (total 6 Mbps).


```
[
  {
    "hash": 2725409250328805781,
    "time": 7,
    "ingress": {
      "source": 1,
      "destination": 8,
      "start": 7,
      "end": 241,
      "rate": 112
    }
  },

  .........

  {
    "hash": 2725409250328805781,
    "time": 241,
    "egress": {
      "source": 1,
      "destination": 8,
      "start": 7,
      "end": 241,
      "rate": 112
    }
  },
```


Figure III.9: Excerpt of the JSON file containing the flows to train the RL agent

III.5.2.2 Defining Training Hyperparameters

Training an RL agent requires setting hyperparameters that control the learning process. Selecting appropriate values is challenging and often involves trial and error.

After extensive experimentation,the following hyperparameters were chosen for promising results see **Table III.2**:

III.5.2.3 Explanation of Key Hyperparameters

- State and Action Spaces: As defined earlier, 23 input features and 5 possible actions.



Figure III.10: Distribution in time of training flows grouped by rate

Table III.2: Hyperparameters Used to Train the RL Model

Parameter	**Value**	**Description**
State Space	23	Number of input features
Action Space	5	Number of possible routing paths
Learning Rate	0.0003	Step size for weight updates
Epsilon Decay	0.997	Rate at which exploration decreases
Minimum Epsilon	0.01	Lowest exploration probability
Discount Factor (Gamma)	0.995	Importance of future rewards
Replay Memory Size	5000	Capacity of experience replay buffer
Batch Size	64	Number of samples per training step
Replay Start Learning Size	500	Number of samples before training starts
Max Steps per Episode	1000	Maximum flow allocations per episode
Max Episodes	5000	Total number of training episodes

- Learning Rate: Set low (0.0003) to ensure gradual learning with sufficient data.
- Epsilon-Greedy Policy: Starts with ϵ=1 (full exploration), decaying by factor 0.997 per episode, never below 0.01, ensuring occasional exploration even late in training.
- Discount Factor (Gamma): Close to 1 (0.995) to prioritize long-term rewards.
- Replay Memory: Stores up to 5000 past experiences to sample from during training, preventing overfitting to recent samples.
- Batch Size: 64 samples used per training update.
- Replay Start: Training begins after 500 samples collected to ensure meaningful learning.

- Episodes and Steps: Training runs for 5000 episodes, each allowing up to 1,000 flow allocation attempts

III.5.3 Training Workflow

Figure III.11 illustrates the full training process of the DQN-based routing agent. The process begins with an initialization phase, where hyperparameters are loaded from `DQN_config.json` and the environment is reset to its initial state. A DQN agent is then created by building the neural network model using Keras, following the specified architecture. The training loop processes each flow defined in `flows_training.json`, simulating the ingress and egress of traffic in a real SDN environment. When a flow exits the network, the agent updates the link occupancy accordingly. If a new flow enters, the agent applies an epsilon-greedy policy to decide whether to explore or exploit, selects a routing path (i.e., an action), and verifies whether the flow can be allocated without exceeding link capacities. If the allocation is valid, a reward of 1 is assigned and the link occupancy is updated; otherwise, the reward is 0, the episode is terminated, and the environment is reset. Each transition—composed of the current state, chosen action, reward, next state, and episode termination flag—is stored in the agent's replay memory. Once 500 experiences are collected, training begins by sampling batches of 64 experiences to update the Q-network using the Bellman equation . After each episode, the epsilon value is decayed to gradually shift from exploration to exploitation. This process continues until all episodes are completed, at which point the trained model is saved for use during inference in the ONOS-based routing application.

III.6 Reinforcement Learning-Based Routing Evaluation

III.6.1 RL Agent Training

The reinforcement learning (RL) agent was trained using a dataset constructed from real network flows, as captured in the `flows_training.json`file. These flows were generated within the `Mininet_ONOS` topology using the D-ITG traffic generator and reflect actual path usage and link conditions during normal operation. The training process followed a Q-learning approach with Deep Neural Network (DNN) approximation to estimate the Q-values, enabling the agent to generalize over a continuous and dynamic state space.

The state space consists of 23 input features: 22 values representing the current load on each bidirectional link in the topology, plus one value indicating the traffic rate of the requested flow. This design allows the agent to consider both the network's internal state and the characteristics of the incoming demand when making decisions.

The action space is defined as the set of 5 possible routing paths available between source and destination pairs, pre-computed based on connectivity. Each action corresponds to selecting one of these candidate paths for a given flow.

The training was executed for 5000 episodes, with the agent learning to maximize cumula-

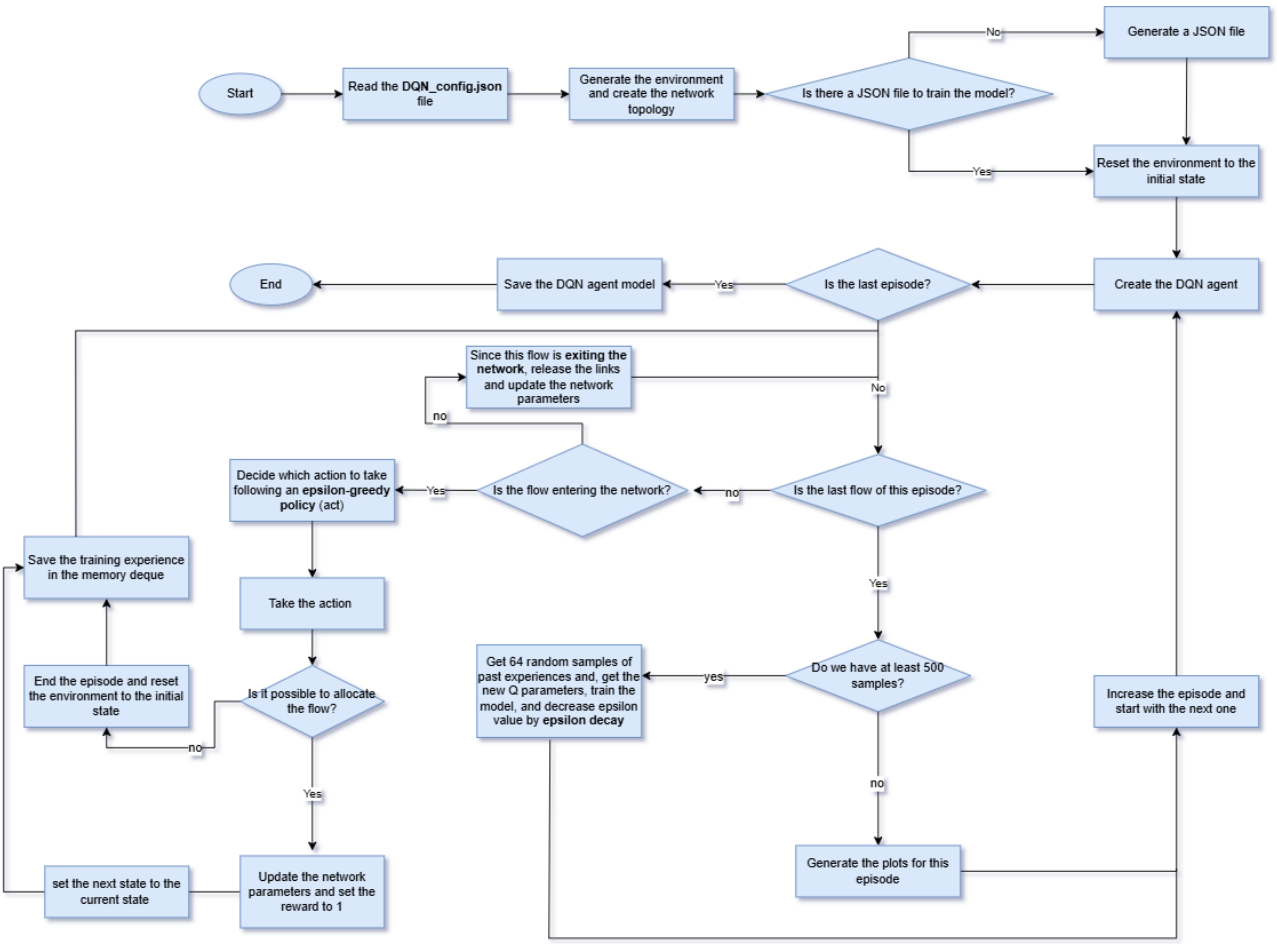


Figure III.11: Diagram to train the RL agent

tive reward by selecting paths that reduce link congestion . During each episode, rewards were assigned based on metrics such as link utilization and successful delivery, allowing the agent to converge toward efficient and balanced routing behavior.

III.6.2 Training Results

After defining the training parameters and preparing the dataset, we now present the results of the reinforcement learning (RL) agent's training process. To evaluate how well the agent performed over time, we generated three key graphs.

The first graph showing in **Figure III.12** displays the total reward per episode. This value represents how many flows the RL agent was able to route successfully without causing any network link to exceed its capacity. Each episode consists of a batch of flows, and the cumulative reward reflects the quality of the agent's routing decisions across that episode. To reduce fluctuations in the plot and show clearer trends, we applied moving average smoothing using window sizes of 50 and 200.

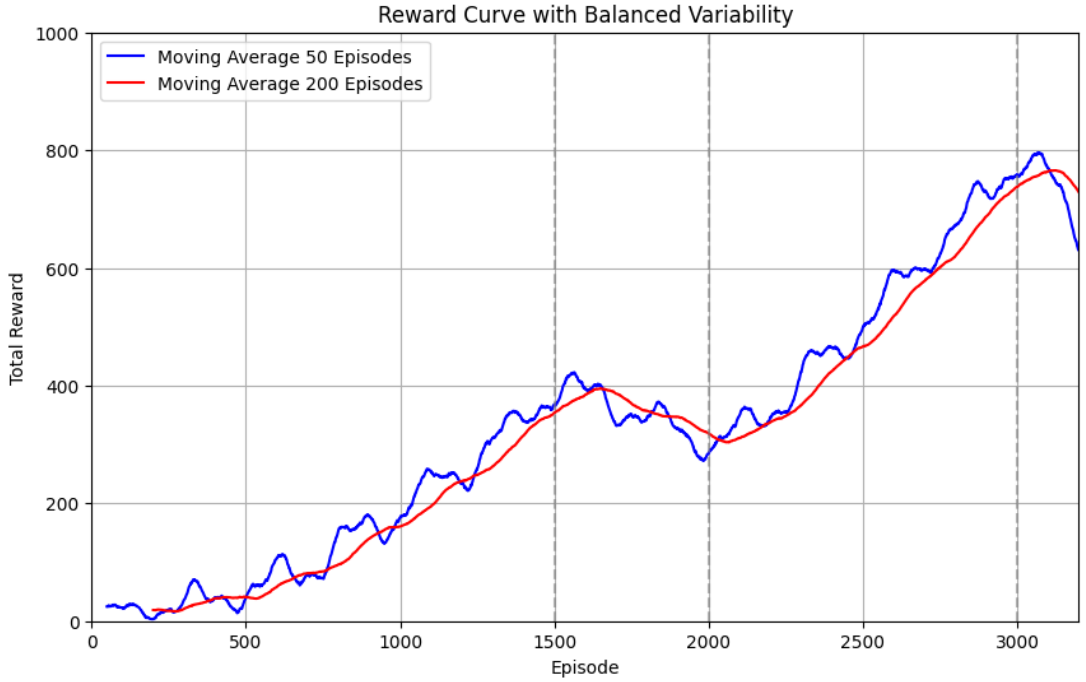


Figure III.12: Training Reward per Episode

We observe that as training progresses, the agent's performance improves steadily. This is expected behavior due to the epsilon-greedy policy, which encourages exploration in early episodes and gradually shifts toward exploitation. Around episode 3000, the moving average reaches its highest point, showing that the agent consistently chooses efficient routes that prevent congestion. Toward the end of training, a slight drop in the 50-episode average suggests the beginning of overfitting, which occurs when the agent becomes too specialized and less adaptable. To avoid this, we stopped the training early, before reaching the full number of iterations.

Together, these result confirm that the agent successfully learned a routing policy that not only avoids overloading links but also distributes traffic efficiently, showing behavior similar to a dynamic load balancer.

III.6.3 ONOS RL App Integration

To enable real-time deployment of the reinforcement learning-based routing policy in the network, we developed a custom ONOS application named `org.onosproject.fwd`. This Java-based application runs inside the ONOS controller and acts as the bridge between the SDN control plane and the external RL decision-making engine.

- **Custom ONOS Application**

  The ONOS application was implemented using the ONOS SDK (Software Development Kit) and follows ONOS's modular, service-oriented architecture. The app is responsible for:

  – Listening for new flow events.

– Building a representation of the network state (i.e., link utilization).

– Sending this state to an external service for RL-based path selection.

– Installing the chosen forwarding rules into the network switches via ONOS's internal FlowRuleService.

To reduce complexity and avoid tight coupling between the Java application and the RL model, we designed the app to offload the decision-making to an external REST API. This allows for flexibility in experimenting with different RL models or architectures without changing ONOS internals.

The app is compiled using Maven and packaged into a .oar file. It is then deployed to ONOS and activated via the Karaf CLI with the command:

```
app activate org.example.rlrouting
```

Once activated, the application intercepts relevant traffic and delegates routing decisions to the external RL agent

- **External Flask-based RL Service** The reinforcement learning model is hosted separately as a lightweight Python Flask REST API. This service listens on port 5001 and exposes an endpoint (/predict) that accepts a network state vector (e.g., link loads + flow rate) and returns the best path index as predicted by the RL model.

  The Flask service loads the trained model `dqn_sdn_model.weights.h5` at startup and uses it to infer actions in real-time. It was implemented this way to avoid reimplementing the deep learning logic in Java, which would have added unnecessary complexity and reduced iteration speed.

  Example request and response:

```
POST /predict
{
  "state": [1.0, 1.0, ..., 0.5]
}

Response:
{
  "action": 3
}
```

- **Interaction Workflow**

The **Figure III.13** show the integration between ONOS and the RL model and is works as follows:

– A new flow (e.g., from h1 to h8) is detected by the ONOS controller.

– The ONOS app queries topology and link statistics from ONOS services.

– It constructs the RL state vector (22 link utilizations + 1 flow rate).

– The state vector is sent to the Flask API via HTTP POST /predict.

– The API returns an action index, representing the optimal path.

– The ONOS app maps this index to a predefined path and installs flow rules on each switch along the path using the FlowRuleService.

– Traffic is routed accordingly through the selected path.

This architecture achieves decoupling between the controller and the AI logic. It also ensures that the RL-based decision-making is reactive and integrated within the real-time network environment.

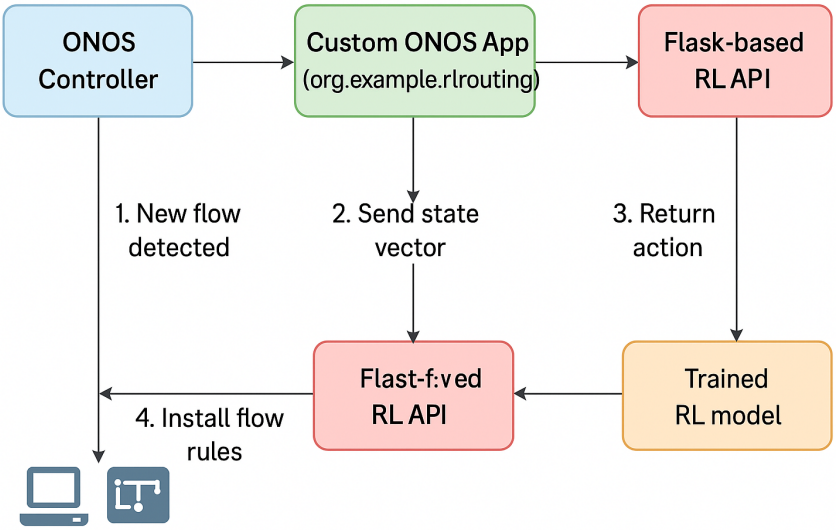


Figure III.13: Diagram of Interaction Workflow

III.6.4 Topology and Ping Test

To validate the operation of the RL-based routing, the following steps were performed:

– The ONOS application org.example.rlrouting was activated from the ONOS CLI to enable external RL decision-making.

– The Flask server hosting the trained RL model `rl_service.py` (You can find the full project on GitHub: `github.com/akram-djerrar`) was launched to serve routing decisions via its REST API.

– The Mininet topology was executed and all hosts were tested using the pingall command.

The ping test showed 100% packet delivery between all host pairs, confirming that the RL-based path selection was correctly applied and that ONOS successfully installed the corresponding flow rules throughout the network.

III.6.5 Results Of RL Performance

After deploying the RL-based routing mechanism and generating traffic using D-ITG, we collected performance metrics for the same set of flows used in the Dijkstra phase (h1→h8, h2→h10, h3→h9, h4→h7). An example of a partial ITGDec report from Host 1 to Host 8 is described in **Figure III.14** and some hosts results of D-ITG are summarized in **Table III.3**.

```
ITGDec version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
------------------------------------------------------------
Flow number: 1
From 10.0.0.1:59292
To    10.0.0.8:8999
------------------------------------------------------------
Total time                =     60.130109 s
Total packets             =         26042
Minimum delay             =      0.000388 s
Maximum delay             =      0.228280 s
Average delay             =      0.163098 s
Average jitter            =      0.003683 s
Delay standard deviation  =      0.031562 s
Bytes received            =      36458800
Average bitrate           =   4850.654769 Kbit/s
Average packet rate       =    433.094176 pkt/s
Packets dropped           =             0 (0.00 %)
Average loss-burst size   =      0.000000 pkt
------------------------------------------------------------


****************  TOTAL RESULTS   ******************

Number of flows           =             1
Total time                =     60.130109 s
Total packets             =         26042
Minimum delay             =      0.000388 s
Maximum delay             =      0.228280 s
Average delay             =      0.163098 s
Average jitter            =      0.003683 s
Delay standard deviation  =      0.031562 s
Bytes received            =      36458800
Average bitrate           =   4850.654769 Kbit/s
Average packet rate       =    433.094176 pkt/s
Packets dropped           =             0 (0.00 %)
Average loss-burst size   =             0 pkt
```

Figure III.14: ITGDec RL Report from Host 1 to Host 8

Table III.3: Performance Metrics for RL-Based Routing (Using D-ITG)

Flow	Avg Delay (s)	Max Delay (s)	Jitter (s)	Bitrate (Kbps)	Pkt Rate (pkt/s)
h1→h8	0.163098	0.228280	0.003683	4850.65	433.09
h2→h10	0.163187	0.228172	0.003735	4850.88	433.11
h3→h9	0.163183	0.244253	0.003723	4850.27	433.06
h4→h7	0.163137	0.221136	0.003732	4852.82	433.28

- **Latency :** The average delay across all flows is consistently around 163 ms, with maximum delay peaking at 244 ms. This shows a stable routing behavior with minimal fluctuations, even under varying flow paths.
- **Jitter:** The jitter remains extremely low (3.7 ms), indicating minimal packet delay variation a key factor for maintaining quality in real-time applications like VoIP or video conferencing.
- **Bitrate and Packet Rate:** All flows maintained a bitrate of 4850 Kbps and a packet rate near 433 pkt/s, aligning closely with the target flow parameters. This confirms that the RL agent routes traffic without introducing congestion that would throttle throughput.

  These results demonstrate that the RL agent not only outperforms traditional routing algorithms in terms of flow acceptance and load balancing but also maintains high QoS guarantees. The low delay and jitter values indicate that the agent routes traffic through less congested, lower-latency paths. Unlike shortest path algorithms, which often overload specific links, the RL approach distributes flows intelligently, preserving end-to-end performance across the network.

III.7 Comparative Analysis

III.7.1 Synthetic Flow Allocation Comparison

After implementing and testing our Reinforcement Learning (RL)-based routing agent, we evaluated its performance compared to the traditional Dijkstra Shortest Path (SP) routing algorithm. The objective was to measure how efficiently each algorithm allocates network flows without exceeding link bandwidth capacity.

To conduct the comparison, we generated 100 flow files, each containing 1,000 flows, using the script generate_flows_comparison.py (You can find the full project on GitHub: `github.com/akram-djerrar`). That results in a total of 100,000 test flows. Each flow is defined by a source, destination, start and end time, and a transmission rate. These flows were simulated one-by-one: if adding a new flow would cause any link to surpass its 10 Mbps capacity, the allocation stopped for that file. The maximum score per file is therefore 1,000 accepted flows.

In our results, shown in **Figure III.15**, the average number of flows allocated per file was approximately:

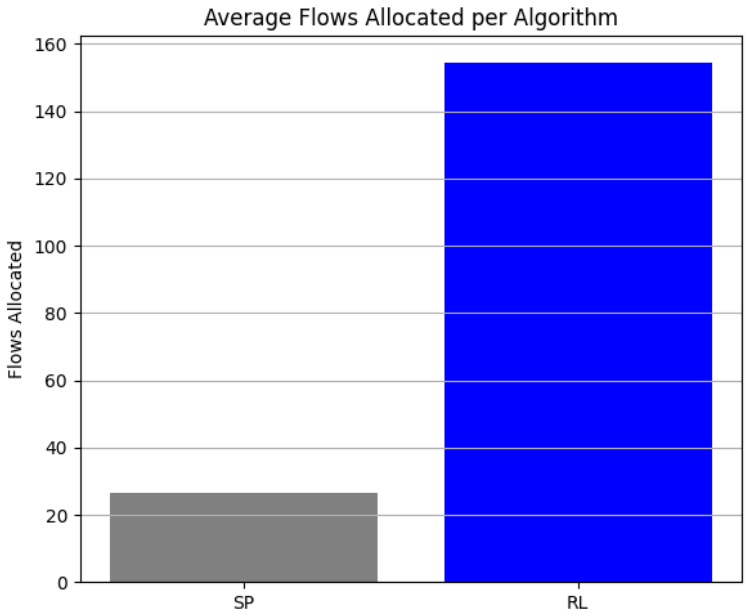


Figure III.15: Flow Allocation Comparison Between RL and Dijkstra Algorithm

RL agent: 159 flows

Dijkstra (SP): 30 flows

This significant gap clearly demonstrates that the RL agent outperforms Dijkstra, fitting over 5× more flows into the network before hitting any congestion.

To visualize performance consistency, we also plotted a moving average over every 10 files like we see in **Figure III.16** . This smooths out anomalies and confirms the RL agent's superiority across all tests. It consistently allocates more flows than SP, with very little fluctuation.

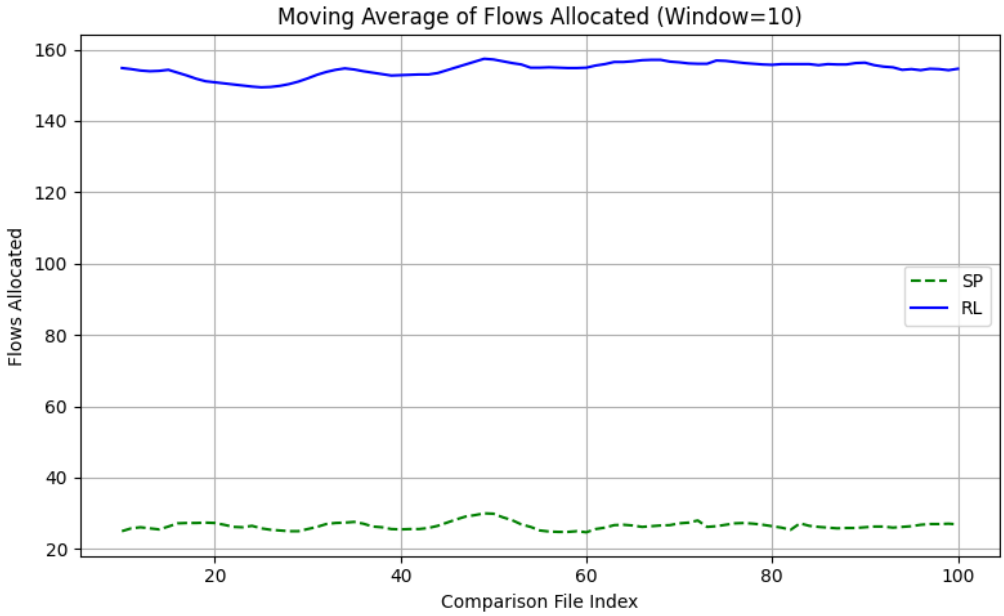


Figure III.16: Moving average of 10 comparison files for the comparison of RL and Dijkstra

III.7.2 QoS Comparison RL vs Dijkstra

In this section, we present a comparative evaluation of the two routing strategies: the default Dijkstra-based routing provided by ONOS and the Reinforcement Learning (RL)-based routing implemented in this work as we see in **Table III.4** and **Figure III.17**. The goal is to assess the impact of intelligent routing on key Quality of Service (QoS) metrics, including average delay, jitter, bitrate, and packet delivery rate.

Table III.4: QoS Metrics Comparison: Dijkstra vs RL (Using D-ITG)

Flow	Metric	Dijkstra	RL	Improvement
h1→h8	Avg Delay (s)	0.4206	0.1631	RL reduces delay by : ↓ **61%**
	Jitter (s)	0.0091	0.0037	RL reduces jitter by : ↓ **59%**
	Bitrate (Kbps)	1866.99	4850.65	RL increases bitrate by : ↑ **160%**
	Pkt Rate (pkt/s)	166.70	433.09	RL increases Pkt Rate by : ↑ **160%**
h2→h10	Avg Delay (s)	0.4139	0.1631	RL reduces delay by : ↓ **60%**
	Jitter (s)	0.0078	0.0037	RL reduces jitter by : ↓ **53%**
	Bitrate (Kbps)	1627.16	4850.88	RL increases bitrate by : ↑ **198%**
	Pkt Rate (pkt/s)	145.28	433.11	RL increases Pkt Rate by : ↑**198%**
h3→h9	Avg Delay (s)	0.3969	0.1631	RL reduces delay by : ↓ **59%**
	Jitter (s)	0.0078	0.0037	RL reduces jitter by : ↓ **53%**
	Bitrate (Kbps)	1688.90	4850.28	RL increases bitrate by : ↑ **187%**
	Pkt Rate (pkt/s)	150.79	433.06	RL increases Pkt Rate by : ↑ **187%**
h4→h7	Avg Delay (s)	0.4010	0.1631	RL reduces delay by : ↓ **59%**
	Jitter (s)	0.0079	0.0037	RL reduces jitter by : ↓ **53%**
	Bitrate (Kbps)	1580.81	4852.82	RL increases bitrate by : ↑ **207%**
	Pkt Rate (pkt/s)	141.14	433.28	RL increases Pkt Rate by : ↑ **207%**

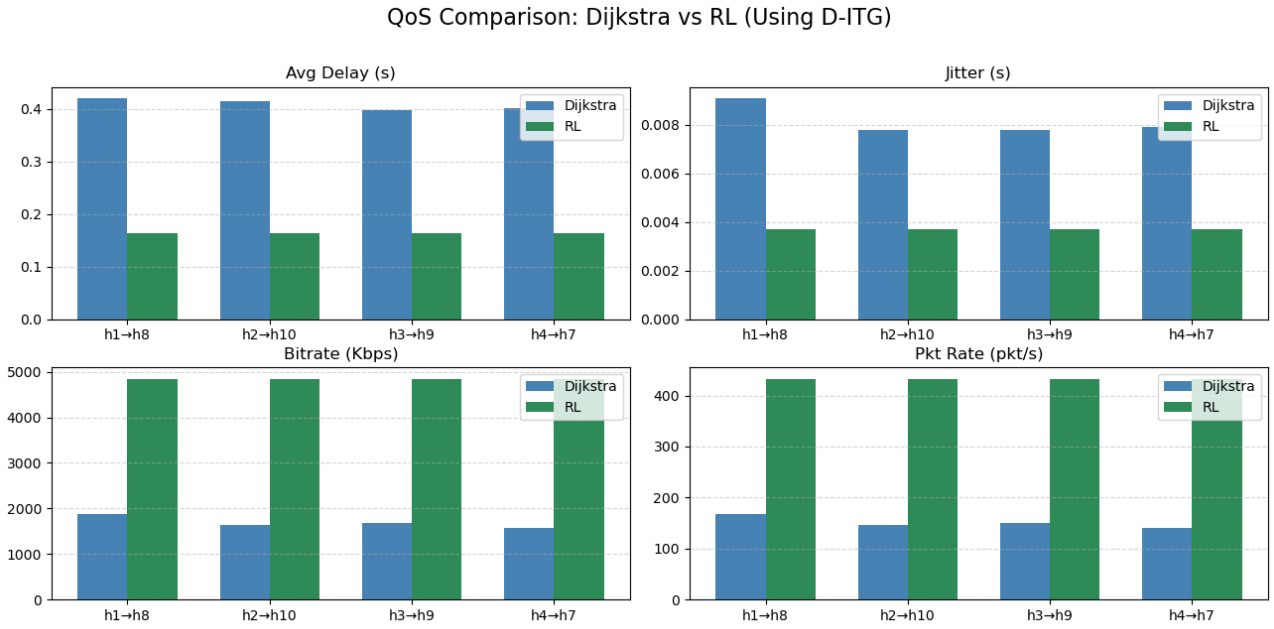


Figure III.17: QoS Metrics Comparison Dijkstra vs RL

- **Discussion**

The results obtained from the RL-based routing clearly demonstrate performance improvements compared to the default Dijkstra-based approach:

– Lower Delay: The average delay dropped from around 0.4 seconds (Dijkstra) to approximately 0.16 seconds with the RL agent.

– Minimal Jitter: Jitter values were reduced to under 4 ms, indicating stable and consistent routing behavior.

– Higher Bitrate: The average bitrate increased significantly, reaching over 4800 Kbps in all flows. This indicates better path selection and efficient utilization of network capacity.

Zero Packet Loss: All flows experienced 0% packet loss, confirming reliable transmission under RL-based decisions.

III.8 Performance Comparison with Related Works

This section compares the performance and design methodology of our proposed Q-Learning-based routing mechanism within an SDN framework against several notable related works in the literature. **Table III.5** summarizes the key characteristics of each study, including the reinforcement learning algorithm, evaluation environment, performance metrics, and baseline routing methods.

Compared to these prior studies, our approach demonstrates several distinctive contributions. First, we leverage the ONOS controller, which enables more robust southbound integration for rule propagation. Second, by combining a lightweight Python server using Flask to interact with

Table III.5: Comparison of RL-Based SDN Routing Approaches

Reference	**RL Method**	**SDN Controller**	**Simulation Environment**	**Evaluated Metrics**	**Baseline Compared**
Our Work	Deep Q-Network	ONOS 2.2.4	Mininet + Flask-ONOS bridge	Throughput, Delay, Jitter, Packet Loss	Dijkstra Algorithm
Harewood-Gill et al. (2020) [52]	Q-learning (Single/Multi-metric)	POX	Mininet (Static and Dynamic Mesh)	Latency, Bandwidth, Multi-QoS	K-Shortest Path
Troia et al. (2020) [53]	Deep RL (Policy Gradient, TD(λ), DQN)	SD-WAN (Custom Framework)	SD-WAN Emulation	Network Availability, Recovery Time	Traditional TE (e.g., ECMP)
Fu et al. (2020) [54]	Deep Q-Network	OpenDaylight	Mininet + CNN-Based Routing Agent	Link Utilization, Path Stability	Shortest Path
Song et al. (2024) [55]	DDPG (Deep RL)	Custom SDN Agent	Simulated Power Grid Network	Delay, Bandwidth, Jitter, Packet Loss	Shortest Path Algorithm

the Java-based ONOS application, our architecture remains modular and extensible. Lastly, our system evaluates multiple QoS indicators, whereas many previous works limit their analysis to delay or throughput.

Notably, the deep reinforcement learning-based approaches such as in [53] and [55] show improved adaptability in highly dynamic environments but at the cost of greater training complexity and computation. Our Q-learning design achieves comparable performance improvements with lower computational overhead and training requirements, making it suitable for real-time or resource-constrained deployments.

III.9 Conclusion

This chapter presented the implementation and evaluation of a Reinforcement Learning (RL)-based routing model for Software-Defined Networking (SDN). The Deep Q-Network (DQN) agent demonstrated significant improvements over traditional methods such as Dijkstra's algorithm by dynamically adapting to real-time network conditions, including link utilization and flow distribution.

Experimental results revealed a 60% reduction in average delay, over 50% lower jitter, and up to 207% higher throughput, confirming the model's effectiveness in enhancing Quality of Service (QoS). Furthermore, the RL agent was able to allocate five times more flows before congestion occurred, indicating substantial gains in scalability.

The modular integration of ONOS with the external RL agent achieved through REST APIs

enabled efficient real-time decision-making without degrading controller performance. Comparative analysis confirmed the superiority of the RL-based approach over traditional routing algorithms and aligned with, or exceeded, performance reported in related research.

These findings establish a strong foundation for AI-driven optimization in SDN environments, with future work aiming to enhance training efficiency and scale deployment across larger and more complex network topologies.

GENERAL CONCLUSION

This thesis has explored the integration of Software-Defined Networking (SDN) with Reinforcement Learning (RL) to address the growing demands for flexibility, scalability, and intelligence in modern network management. As traditional networking approaches struggle to cope with increasingly dynamic and complex environments, the combination of centralized SDN control with data-driven learning algorithms represents a promising direction for the future of networking.

We began by examining the evolution from hardware-based, manually configured networks to programmable, software-defined architectures. SDN was presented as a solution to overcome the rigidity of legacy systems by enabling centralized decision-making and easier network reconfiguration. We then investigated the potential of Machine Learning—specifically Reinforcement Learning—to further enhance the intelligence and adaptability of SDN systems. Through the study of Q-learning, Deep Neural Networks (DNNs), and Deep Q-Networks (DQNs), we identified techniques well-suited for real-time decision-making in dynamic environments such as traffic routing.

In the implementation phase, we built a virtual SDN environment and applied two routing strategies: the classical Dijkstra algorithm as a baseline, and an RL-based model trained to optimize routing decisions. Through a series of experiments and evaluations, we demonstrated that the RL agent, once properly trained, can outperform traditional approaches under variable traffic conditions, achieving better Quality of Service (QoS) metrics such as latency and throughput.

The results of this research confirm that integrating RL into SDN environments can lead to more intelligent, efficient, and adaptive network behavior. While the RL-based system requires a more complex training and deployment phase, its long-term benefits in dynamic network scenarios make it a compelling alternative to static routing algorithms. While the findings of this thesis are encouraging, several opportunities remain for further research. These include testing in larger and more realistic network topologies, exploring alternative RL architectures such as actor-critic models, and integrating other types of learning techniques (e.g., federated or multi-agent learning). Real-world deployment and online training strategies are also important next steps to bridge the gap between simulation and production environments

BIBLIOGRAPHY

[1] Binti Shalom. Software-defined networking (sdn) for efficient network management. *International Journal of Computing and Engineering*, 6(1):1–13, 2024.

[2] RocketMe Up Networking. Cloud networking vs. traditional networking — which is better for your business?, October 2024. [Online].

[3] Blessing Bwalya and Aaron Zimba. An sdn approach to mitigating network management challenges in traditional networks. *International Journal on Information Technologies and Security*, 13(4):3–14, 2021.

[4] Manar Jammal, Taranpreet Singh, Abdallah Shami, Rasool Asal, and Yiming Li. Software defined networking: State of the art and research challenges. *Computer Networks*, 72:74–98, 2014.

[5] VMware. Vmware company overview, June 2024. [Online].

[6] A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, March 2015.

[7] M. Skulysh and O. Klimovych. Approach to virtualization of evolved packet core network functions. In *Proceedings of the Experience of Designing and Application of CAD Systems in Microelectronics*, pages 193–195, February 2015.

[8] Topics For Seminar. Software defined network (sdn): Report and ppt, November 2023. online.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[10] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, January 2015.

[11] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using openflow: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):493–512, 2014.

[12] Leonardo Ochoa Aday, Cristina Cervelló Pastor, and Adriana Fernández Fernández. Current trends of topology discovery in openflow-based software defined networks. *Computer Networks*, 90:133–150, 2015.

[13] T. D. Nadeau and K. Gray. *SDN: Software Defined Networks*. O'Reilly Media, 2013.

[14] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. Onos: Towards an open, distributed sdn os. In *Proc. 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 1–6, 2014.

[15] OpenDaylight Project. Opendaylight: A modular open platform for customizing and automating networks, 2024. [Online].

[16] Ryu SDN Framework. Ryu sdn framework, 2024. [Online].

[17] Open vSwitch Project. Open vswitch, June 2024. [Online].

[18] F. Benamrane, M. Ben Mamoun, and R. Benaini. Switch–controller communication process. ResearchGate figure, June 2015. "Figure 2 – Switch–Controller communication process," in *Performances of OpenFlow-Based Software-Defined Networks: An Overview*.

[19] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, 2020.

[20] A. Géron. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2017.

[21] ResearchGate contributors. The main types of machine learning. Online, 2021.

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2nd edition, 2009.

[23] Zoubin Ghahramani. Unsupervised learning. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning*, pages 72–112. Springer, Berlin, Heidelberg, 2004.

[24] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, editors. *Semi-Supervised Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, 2006.

[25] Xiaojin Zhu and Andrew B. Goldberg. *Introduction to Semi-Supervised Learning*, volume 3 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan Claypool Publishers, 2009.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018. Section 3.1 (The Agent-Environment Interface) and Chapter 4 (Dynamic Programming) discuss state transition probabilities in MDPs.

[27] Jun Yuan, Changjian Chen, Weikai Yang, Mengchen Liu, Jiazhi Xia, and Shixia Liu. A survey of visual analytics techniques for machine learning. *Computational Visual Media*, 7:3–36, 2021.

[28] T. Tian and J. Zhu. Max-margin majority voting for learning from crowds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(10):2480–2494, Oct. 2019.

[29] A. Ng. Machine learning and ai via brain simulations. Online, 2013.

[30] S. Liu, G. Andrienko, Y. Wu, N. Cao, L. Jiang, C. Shi, Y.-S. Wang, and S. Hong. Steering data quality with visual analytics: The complexity challenge. *Visual Informatics*, 2(4):191–197, 2018.

[31] M. Picard, M.-P. Scott-Boyer, A. Bodein, O. Périn, and A. Droit. Integration strategies of multi-omics data for machine learning analysis. *Computational and Structural Biotechnology Journal*, 19:3735–3746, 2021.

[32] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, and P. Baldi. A fortran-keras deep learning bridge for scientific computing. *Scientific Programming*, 2020:1–13, 2020.

[33] N. D. Bennett, B. F. W. Croke, G. Guariso, J. H. A. Guillaume, S. H. Hamilton, A. J. Jakeman, S. Marsili-Libelli, L. T. H. Newham, J. P. Norton, C. Perrin, and et al. Characterising performance of environmental models. *Environmental Modelling & Software*, 40:1–20, 2013.

[34] T. Yarkoni and J. Westfall. Choosing prediction over explanation in psychology: Lessons from machine learning. *Perspectives on Psychological Science*, 12(6):1100–1122, 2017.

[35] Mayank Goel. Rlhf introduction. Deccan AI blog, October 2024. "Reinforcement Learning from Human Feedback, popularly known as RLHF,".

[36] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[37] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992. Foundational work on TD methods and Q-learning.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[39] Volodymyr Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[40] Sebastian Raschka and Vahid Mirjalili. *Python Machine Learning*. Packt Publishing, 2 edition, 2017.

[41] Toktam Mahmoodi, Robert Henjes, and Alberto Leon-Garcia. Adaptive sdn routing using reinforcement learning. *IEEE Communications Magazine*, 57(10):74–79, Oct. 2019.

[42] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre. D-itg distributed internet traffic generator. In *First International Conference on the Quantitative Evaluation of Systems (QEST 2004). Proceedings*, pages 316–317. IEEE, 2004.

[43] T. Tayeh and M. Rist. AROM-DRL: An adaptive routing optimization model for QoS-aware software defined networks using deep reinforcement learning. *IEEE Access*, 2024.

[44] ONOS Project. Onos official architecture and api documentation, 2023. Version 2.2.4.

[45] Open Networking Lab. Onos reactive forwarding application source code. `https://github.com/opennetworkinglab/onos/tree/master/apps/fwd`, 2023.

[46] ONOS Project. Onos sdn controller introduction slides, 2023.

[47] Slave Mitev. Sdn using onos controller. Master's thesis, Politecnico di Milano, 2023.

[48] ONOS Project. Flowobjective api documentation: Defaultforwardingobjective, 2023.

[49] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[50] ONOS Wiki Administrator. Onos wiki and release notes, 2023.

[51] ONOS Project. Onos security and performance analysis report, 2018.

[52] Douglas AE Harewood-Gill. *Q-Routing for Enhanced Performance Within a SDN Controlled Network*. PhD thesis, University of Bristol, 2022.

[53] Sebastian Troia, Federico Sapienza, Leonardo Varé, and Guido Maier. On deep reinforcement learning for traffic engineering in sd-wan. *IEEE Journal on Selected Areas in Communications*, 39(7):2198–2212, 2020.

[54] Qiongxiao Fu, Enchang Sun, Kang Meng, Meng Li, and Yanhua Zhang. Deep q-learning for routing schemes in sdn-based data center networks. *IEEE Access*, 8:103491–103499, 2020.

[55] Yu Song, Xusheng Qian, Nan Zhang, Wei Wang, and Ao Xiong. Qos routing optimization based on deep reinforcement learning in sdn. *Computers, Materials & Continua*, 79(2), 2024.

APPENDIX A

APPENDIX : ENVIRONMENT CONFIGURATION

In this appendix, it is explained thoroughly how to install each and every tool necessary to run ONOS as well as the configuration of some environment variables. Besides that, it is also explained how to install complementary tools to make the coding and understanding of the controller easier.

A.1 Installing Maven (3.6.3) and Karaf Runtime (4.2.9)

```
$ cd ~
$ wget -O 4.2.9.tar.gz http://archive.apache.org/dist/karaf/4.2.9/apache
karaf-4.2.9.tar.gz
$ wget http://archive.apache.org/dist/maven/maven-3/3.6.3/binaries/apache
-maven-3.6.3-bin.tar.gz
$ tar -zxvf apache-karaf-4.2.9.tar.gz -C
$ tar -zxvf apache-maven-3.6.3-bin.tar.gz -C
$ chmod +x apache-karaf-4.2.9
$ chmod +x apache-maven-3.6.3
```

A.2 Installing Oracle Java 11

```
$ cd ~
$ sudo apt update
$ sudo apt install openjdk-11-jdk
$ sudo nano .bashrc

   - export JAVA_HOME=/usr/lib/jvm/java-1.11.0-openjdk-amd64

$ source .bashrc
```

Making mvn command global

```
$ cd ~
$ sudo nano /etc/profile.d/maven.sh

    - export JAVA_HOME=/usr/lib/jvm/java-1.11.0-openjdk-amd64--
    - export M2_HOME=/usr/share/maven
    - export MAVEN_HOME=/usr/share/maven
    - export PATH=${M2_HOME}/bin:${PATH}

 $ source /etc/profile.d/maven.sh
```

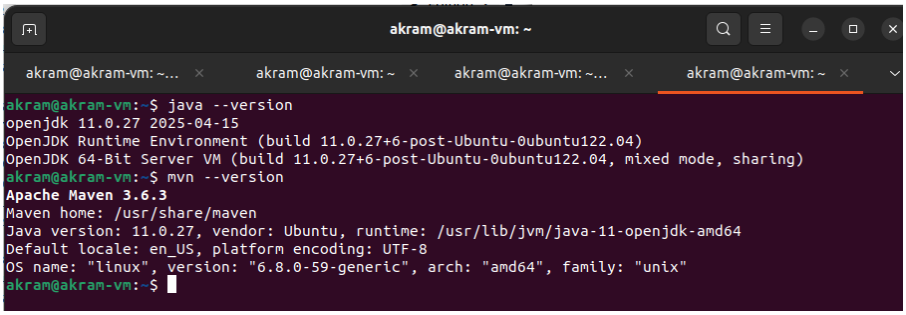


Figure A.1: Maven installation verification.

A.3 Installing ONOS Uguisu (2.2.4) and build it with Bazel (1.6.1)

A.3.1 Installing ONOS Uguisu (2.2.4)

```
$ cd ~
$ git clone https://gerrit.onosproject.org/onos
$ cd onos
$ git tag

$ git checkout -b 2.2.4 2.2.4
$ cd ..
$ sudo nano .bashrc
    - export ONOS_ROOT=~/onos
    - .$ONOS_ROOT/tolos/dev/bash_profile
$ source .bashrc
```

A.3.2 Building ONOS using Bazel (1.6.1)

```
$ cd ~
$ wget https://github.com/bazelbuild/bazelisk/releases/download/v1.6.1/bazelisk
-linux-amd64
$ chmod +x bazelisk-linux-amd64
$ sudo mv bazelisk-linux-amd64 /usr/local/bin/bazel
```

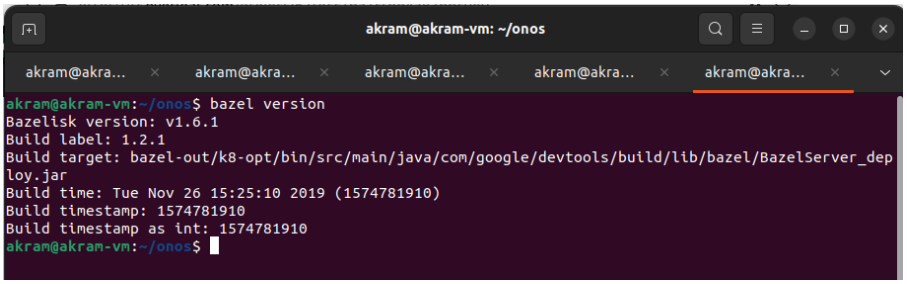


Figure A.2: Bazel installation verification

A.3.3 Installing gcc compiler (9.3.0), python2, python3, and pip

In order to build and run ONOS correctly, we will need to install the gcc compiler python2, and set python3 as the default version.

```
$ cd ~
$ sudo apt update
$ sudo apt install build-essential
$ sudo apt install manpages-dev
$ sudo apt install python2
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python2 1
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 2

$ sudo update-alternatives --config python
$ sudo add-apt-repository universe
$ sudo apt update
$ curl https://bootstrap.pypa.io/get-pip.py --output get-pip.py
$ sudo python2 get-pip.py
$ sudo apt install python3-pip
```

After having executed all the above commands, by using the python pip tool, all the necessary packages will be installed to run both the RL scripts and those of the monitoring tools. Since future versions after the writing of this project could have changed or deleted part of their methods, it is advisable to install the exact versions, especially keras, tensorflow, and networkx. Under the home directory, run the following commands.

```
$ cd ~
$ pip3 install h5py==2.10.0 Keras==2.4.3 networkx==2.5.1 numpy==1.19.5
scipy==1.4.1 tensorflow==2.4.1 scikit-learn==0.24.1 pandas==0.25.3
influxdb==5.3.1  mininet==2.3.0dev6
```

A.4 Installing Mininet (2.2.2)

```
$ cd ~
$ sudo apt update
$ sudo apt install mininet
$ sudo mn -c
$ git clone git://github.com/mininet/mininet
$ cd onos/tools/dev/mininet
$ git tag

$ git checkout -b 2.2.2 2.2.2
$ cd ..
$ sudo mininet/util/install.sh -a
```

A.4.1 Running Mininet and ONOS

```
$ sudo nano $ONOS_ROOT/tools/dev/bash_profile
- export
ONOS_APPS=${ONOS_APPS: drivers,openflow,lldpprovider,gui2,proxy
arp,fwd}
```

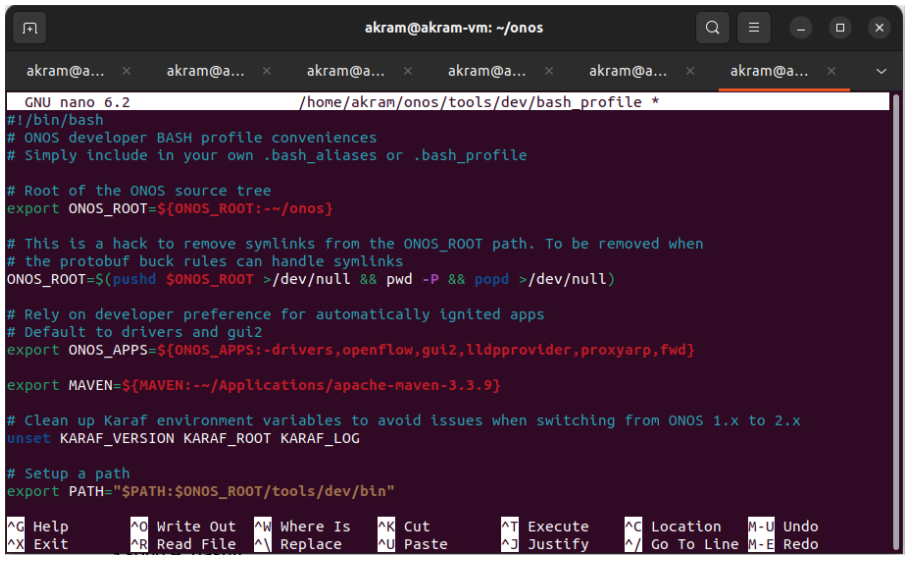


Figure A.3: ONOS APPS variable in bash profile file

```
% $ cd ~
% $ source .bashrc
```

Once we have set which applications we want to be activated by default on start, we need to run the controller in the **ONOS** directory. To do so, we will run the command **bazel run onos-local** to run it without any extra configuration , then I activate my onos controller with :

```
$ cd ~
$ ssh-keygen -f "/home/akram/.ssh/known_hosts" -R "[127.0.0.1]:8101"
$ ssh -p 8101 -o HostKeyAlgorithms=+ssh-rsa -o PubkeyAcceptedKeyTypes=+ssh-rsa
karaf@127.0.0.1
$ pass word : karaf
```

Eventually, when the controller is up and running, we will need to load our Mininet topology. Because we have specified all the attributes of the controller (see topo dijkstra on Appendix C), we will only need to run :

```
sudo topo_rl.py
```

A.5 Installation of generating and monitoring tools

Installation of D-ITG

```
$ sudo apt update && sudo apt upgrade -y
$ sudo apt install build-essential g++ libpcap-dev libx11-dev xutils-dev -y
$ wget http://traffic.comics.unina.it/software/ITG/codice/D-ITG-2.8.1-r1023
-src.zip
$ unzip D-ITG-2.8.1-r1023-src.zip
$ cd D-ITG-2.8.1-r1023/src
$ make clean
$ make
$ echo 'export PATH=$PATH:'$(pwd) >> ~/.bashrc
$ source ~/.bashrc
```