



DIOGO MIGUEL GONÇALVES SIMÕES

Degree in Electrical and Computers Engineering Sciences

**INTELLIGENT ROUTING FOR
SOFTWARE-DEFINED MEDIA NETWORKS**

MASTER IN ELECTRICAL AND COMPUTERS ENGINEERING

NOVA University Lisbon
February, 2022



INTELLIGENT ROUTING FOR SOFTWARE-DEFINED MEDIA NETWORKS

DIOGO MIGUEL GONÇALVES SIMÕES

Degree in Electrical and Computers Engineering Sciences

Adviser: Pedro Miguel Figueiredo Amaral
Assistant Professor, NOVA School of Science and Technology

Co-adviser: Flávio Dinis Gonçalves Rosa Jacinto
Senior System Developer, Skyline Communications

Examination Committee:

Chair: Ph.D. João Paulo Branquinho Pimentão
Assistant Professor, NOVA School of Science and Technology

Adviser: Ph.D. Pedro Miguel Figueiredo Amaral
Assistant Professor, NOVA School of Science and Technology

Member: Ph.D. Rodolfo Alexandre Duarte Oliveira
Associate Professor, NOVA School of Science and Technology

Intelligent Routing for Software-Defined Media Networks

Copyright © Diogo Miguel Gonçalves Simões, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To Rita and my family.

ACKNOWLEDGEMENTS

In this section I would like to express my dearest thanks to the ones that helped me along this journey.

First of all, I seize the opportunity to thank my adviser, Dr. Pedro Amaral, for allowing me to develop this project and for all his assistance and availability during the extent of this dissertation. His expertise was a fundamental support to the work developed.

Furthermore, I would also like to praise my co-adviser, Flávio Jacinto, for all the time invested in me and this project. Not only his collaboration was key to this project's success, but he also revealed to be a great motivator and, ultimately, a friend.

Additionally, my sincere thank you to Skyline Communications for the opportunity provided and all the staff for the sympathy, care and availability revealed.

Many thanks are also due to NOVA School of Science and Technology, NOVA University of Lisbon, and to all the teaching staff sharing information and knowledge throughout my academic journey. This environment helped me grow both professionally and personally.

A very special thank you to my parents and sister, for all the support, patience and affection they showed, not only during this project and course, but during my whole life. They were the reason why I was able to get where I am today and become who I have become.

It is my privilege to thank my beautiful girlfriend, Rita Panóias, one of the most important parts of my life. She has offered me nothing but support, patience, companionship and love during every single minute we have spent together. I could not ask for more from her and can never thank her enough.

Last but not least, I need to thank my dear friends Miguel Azevedo and Ricardo Cruz, for being there during every step of the way and for making the path to get here easier. A thank you also to Alexandru Tabarcea, Álisson Dias and André Barradas for the ideas and moments shared in the past 5 years.

To all of you,

My most sincere thank you!

“Everything is theoretically impossible, until it is done.”
(Robert A. Heinlein)

ABSTRACT

The multimedia market is an industry with an ever-growing demand coupled with strict requirements. Be it in live streaming services or file content broadcast, multimedia providers need to deliver the best possible quality in order to meet their customer's requirements and gain or keep their trust. Multimedia traffic has a high impact on networks and, due to its nature, is sensitive to congestion or hardware failure. Thus, it is frequently that multimedia providers resort to third-party software to monitor quality parameters.

Skyline Communications' DataMiner[®] offers network monitoring, orchestrating and automation capabilities across a broad range of applications and environments. These features are enabled by the emergence of [Software-Defined Networking \(SDN\)](#) which provides a global view of networks and the ability to change network properties through software applications. This contrasts with traditional networks which are rigid, static and difficult to scale-up.

An application that greatly benefits from the global network view of [SDN](#) is routing optimization. Through routing optimization, a network can effectively deliver more traffic by efficiently balancing load across the different links and paths between end points of a service, reaching an increased performance in data transport.

This dissertation comes to light with the goal of optimizing DataMiner's routing mechanism by exploring the routing optimization possibilities enabled by its [SDN](#)-like architecture. Both link cost optimization-based and [Machine Learning \(ML\)](#) approaches are evaluated as possible solutions to Skyline's problem and several experiments were conducted to compare them and understand their impact on network performance while transporting multimedia streams.

Keywords: Multimedia providers, Software-Defined Networking, Routing optimization, Machine learning

RESUMO

O mercado audiovisual é uma indústria onde a procura está em constante crescimento, bem como a exigência. Tanto durante transmissões ao vivo como de conteúdo multimédia pré-gravado, os provedores de multimédia necessitam de garantir a melhor qualidade possível para corresponderem aos requisitos dos seus clientes e conquistarem ou manterem a sua confiança nos seus serviços. O tráfego multimédia tem um forte impacto nas redes que o transportam e, graças à sua natureza, é bastante sensível a congestão ou a falhas de equipamento. Por este motivo, é frequente os provedores de multimédia recorrerem a aplicações externas para monitorização de parâmetros de qualidade.

O DataMiner[®], desenvolvido pela Skyline Communications, oferece a capacidade de monitorizar e orquestrar redes de transporte de multimédia bem como de automatizar as suas funcionalidades num vasto conjunto de enquadramentos e ambientes. Tais funcionalidades são oferecidas pelo aparecimento de SDN que permite que se tenha uma visão global de uma rede e que se altere de forma flexível as suas definições através de aplicações. As características de redes deste tipo contrastam fortemente com as redes tradicionais marcadas pela sua rigidez, estaticidade e dificuldade de expansão.

Uma área que beneficia bastante com a visão global de redes oferecida pela tecnologia de SDN é a otimização do transporte de dados. Desta forma, uma rede consegue transportar mais dados de forma eficiente através do balanceamento da carga a que é submetida pelas diferentes ligações entre elementos e caminhos que conectam pontos de entrada e saída da mesma, atingindo altos níveis de desempenho.

A presente dissertação surge da intenção da Skyline de otimizar o seu algoritmo de encaminhamento através da exploração de métodos alternativos introduzidos pela tecnologia de SDN. Tanto métodos baseados em otimização do custo de ligações da rede como em aprendizagem automática são avaliados como possíveis soluções para o problema proposto e diversas simulações são conduzidas para as comparar e averiguar o seu impacto no desempenho de redes de transporte de dados multimédia.

Palavras-chave: Provedores de multimédia, *Software-defined networking*, Otimização do transporte de dados, Encaminhamento, Aprendizagem automática

CONTENTS

Contents	ix
List of Figures	xii
List of Tables	xiv
Glossary	xv
Acronyms	xvii
1 Introduction	1
1.1 Motivation	2
1.2 Objectives	2
1.3 Contributions	3
1.4 Outline	3
2 State of the art	5
2.1 Software-Defined Networking	5
2.1.1 Towards Software-Defined Networking	5
2.1.2 Software-Defined Networking: A definition	5
2.1.3 OpenFlow	7
2.1.4 Virtualization and SDN	8
2.1.5 DataMiner’s approach to SDNs	9
2.2 Software-Defined Media Networks	10
2.3 Traffic forecasting	11
2.3.1 Time Series Forecasting	12
2.3.2 Traffic forecasting with Neural Networks	16
2.4 Routing	17
2.4.1 Routing in SDN	18
2.4.2 Optimization-based dynamic routing	18
2.5 Machine Learning	21

CONTENTS

2.5.1	Deep Learning	22
2.5.2	Neural Networks	22
2.5.3	Reinforcement Learning	25
2.5.4	Deep Reinforcement Learning	27
3	System design	38
3.1	Service Resource Manager architecture	38
3.1.1	Concepts	39
3.1.2	Resource monitoring	40
3.1.3	Protocols & Templates	41
3.1.4	Automation	43
3.2	Ryu Controller	43
3.3	Mininet	44
3.4	Deep Q-Network enhancements	44
3.4.1	Double Deep Q-Networks	44
3.4.2	Deep Q-Networks with Prioritized Experience Replay	45
3.4.3	Dueling Deep Q-Networks	45
3.5	Solution design	45
3.5.1	Dynamic weight routing in SRM	46
3.5.2	Dynamic weight routing in the link cost modification simulation	46
3.6	Architecture of the routing optimization mechanism using DRL	46
4	Implementation	50
4.1	Dynamic weight routing in SRM	50
4.1.1	Using automation scripts	50
4.1.2	Using a protocol	53
4.2	Dynamic weight routing in the SDN simulation	55
4.2.1	Mininet network	56
4.2.2	Ryu controller	57
4.3	Deep Reinforcement Learning approach for path selection	58
4.3.1	DRL ecosystem	58
4.3.2	Worst-case scenario	60
4.3.3	Agents	61
5	Results	64
5.1	Evaluation metrics	64
5.2	Topologies	65
5.2.1	ARPANET	65
5.2.2	Real-world media network	66
5.3	Training settings	67
5.4	DRL agents' training	68
5.4.1	Agents' learning process in ARPANET	69

5.4.2	Agents' learning process in Network RW	73
5.5	Performance comparison between routing solutions	74
5.5.1	Link cost optimization techniques	74
5.5.2	DRL versus Dijkstra in ARPANET	74
5.5.3	DDQN's performance in unknown environment conditions	77
5.5.4	DRL versus Dijkstra in Network RW	77
6	Conclusions and future work	79
6.1	Final remarks	79
6.2	Future work	79
6.2.1	Request tailoring	80
6.2.2	Traffic forecasting	80
6.2.3	Complex neural networks	80
6.2.4	Reward function	80
6.2.5	Agent optimization	81
6.2.6	Problem of topology change	81
	Bibliography	82
	Annexes	
I	Annex 1 - Link cost optimization algorithms' performance	90
II	Annex 2 - DRL agents' performance	91

LIST OF FIGURES

2.1	SDN fundamental architecture (adapted from [7][12][13][14][15]).	6
2.2	OpenFlow switch’s architecture (adapted from [18][19]).	7
2.3	Simple Neural Network (NN) with 3 layers (adapted from [62]).	23
2.4	Rectified Linear Unit (ReLU) activation function (adapted from [64]).	24
2.5	Reinforcement Learning (RL) framework (adapted from [69]).	26
2.6	Actor-critic methods’ generalized behaviour (adapted from [69]).	30
2.7	Architecture of multimedia traffic control in SDN (adapted from [53]).	31
2.8	Interaction between system components. Red lines represent outputs and green lines inputs (adapted from [53]).	33
3.1	Transport-Cisco service definition (from DataMiner).	39
3.2	Overview of a DMA (adapted from [77]).	42
3.3	Execution of a group triggered by a timer (adapted from [77]).	43
3.4	Dynamic Dijkstra’s architecture.	46
4.1	JSON InputData example (from DataMiner).	51
4.2	Edge resource properties (from DataMiner).	52
4.3	Service Resource Manager (SRM) Routing Manager’s workflow (adapted from [77]).	53
4.4	SRM Routing Manager implementation.	54
4.5	Routing management table (from DataMiner).	55
4.6	Ryu controller and Mininet API integration.	56
4.7	Interaction between the environment, the environment engine and the topology file.	59
4.8	Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) architecture.	61
4.9	Dueling Deep Q-Network (Dueling DQN) architecture (adapted from [86]). In this figure “N” stands for “N_ACTIONS”.	62
5.1	Advanced Research Projects Agency Network (ARPANET) topology with blue entry nodes, red exit nodes and green switches (adapted from [88]).	65

5.2	Network RW topology with blue entry nodes, red exit nodes and green switches.	67
5.3	Comparison of training results between agents in environment setup 1. . .	70
5.4	Comparison of training results between agents in environment setup 2. . .	70
5.5	Comparison of training results between agents in environment setup 3. . .	71
5.6	Comparison of training results between agents in environment setup 4. . .	71
5.7	Comparison of training results between agents in environment setup 1 and Network RW.	73
5.8	Average bitrate of each agent in each setup and Dijkstra's.	75
5.9	Average Round-trip-time (RTT) of each agent in each setup and Dijkstra's.	76
5.10	Number of uncongested requests of each agent in each setup and Dijkstra's.	76

LIST OF TABLES

4.1	DQN and DDQN network’s layers.	61
4.2	Dueling DQN network layers.	62
5.1	Agent’s networks layers (DQN and DDQN).	66
5.2	Agent’s networks layers (Dueling DQN).	66
5.3	Agent’s networks layers (DQN and DDQN).	67
5.4	Agent’s networks layers (Dueling DQN).	68
5.5	Agents parameters in ARPANET.	69
5.6	Comparison between link cost optimization algorithms and Dijkstra (Minimum Hop Algorithm (MHA)).	74
5.7	Performance comparison between DDQN (setup 1) and Dijkstra.	77
5.8	Performance of the Deep Reinforcement Learning (DRL) agents compared to Dijkstra in Network RW.	78
I.1	Comparison of link cost optimization algorithms in 32 TCP flow requests.	90
II.1	Performance comparison between DRL agents in 32 TCP flow requests (setup 1, ARPANET).	91
II.2	Performance comparison between DRL agents in 32 TCP flow requests (setup 2, ARPANET).	92
II.3	Performance comparison between DRL agents in 32 TCP flow requests (setup 3, ARPANET).	93
II.4	Performance comparison between DRL agents in 32 TCP flow requests (setup 4, ARPANET).	94
II.5	Performance comparison between Dijkstra and DDQN in unseen scenario settings (setup 1, ARPANET).	94
II.6	Performance comparison between Dijkstra and the three agents (setup 1, Network RW).	95

GLOSSARY

API	An API, or Application Programming Interface, is a set of method definitions that facilitate the integration of existing software in the development of new applications. xii , xvi , 6 , 9 , 18 , 38 , 40 , 43 , 44 , 56
ARP	The Address Resolution Protocol is used to discover the addresses of network devices. 57
CLI	The Command Line Interface (CLI) establishes a connection to devices and sends instructions through text commands. 41
DMA	A Dataminer Agent hosts multiple interacting processes. xii , xv , 40 , 42 , 43 , 52
DMS	A Dataminer System includes one or multiple DMAs . 40 , 41 , 43
HTTP	The Hypertext Transfer Protocol is a request/response application layer protocol, with a client-server architecture, that handles distributed, collaborative, hypermedia information systems [1]. xvi , 40
IP	The Internet Protocol is the Internet's standard for addressing and routing data through transport protocols in the form of packets. 10 , 11 , 39 , 40 , 44
JSON	"JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format." [2]. Its biggest advantage is the readability it offers due to its structured representation in attribute-value pairs. xii , 51

LINQ	"Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language." [3]. 51
MAC address	A MAC address is the physical address (i.e. unique identifier) of a machine in a network. 57
NETCONF	The Network Configuration Protocol (NETCONF) defines a way to access and manipulate network devices' configuration data. With this protocol, the device is exposed by means of an API . NETCONF uses XML -encoded messages [4]. xvi, 9
QoE	Quality of Experience is a valuation of network performance from the user's perspective. It measures the customer's satisfaction with the service provided. 10, 31, 32, 33
QoS	Quality of Service is a valuation of network performance. It takes into consideration network performance metrics such as bit rate, packet loss, jitter, throughput, delay and others. 8, 10, 12, 17, 20, 34, 44, 46
RESTCONF	RESTCONF is a HTTP -based protocol that provides accessibility to configuration data using concepts defined for NETCONF [5]. 9
TCP	The Transmission Control Protocol (TCP) is a connection-oriented communication protocol between a server and a client that is reliable, has error control and keeps the order of the data sent. xiv, 65, 74, 90, 91, 92, 93, 94
TCP/IP	TCP/IP stands for Transmission Control Protocol/Internet Protocol and is a standard for computer communication on Internet and other networks. 12, 65
UI	A User Interface is how a user interacts with a product or service. 42, 53, 55
XML	XML (Extensible Markup Language) is a markup language for storing data. xvi, 41, 53

ACRONYMS

ANN	Artificial Neural Network 22, 23
AR	Autoregressive 13, 14
ARIMA	Autoregressive Integrated Moving Average 13, 14, 15, 16, 17
ARMA	Autoregressive Moving Average 14
ARPANET	Advanced Research Projects Agency Network xii, xiv, 35, 36, 65, 66, 69, 73, 74, 78, 81, 91, 92, 93, 94
CNN	Convolutional Neural Network 24, 25, 80
COTS	Commercial-off-the-shelf 10
DCF	DataMiner Connectivity Framework 39, 51, 52, 53, 54
DDPG	Deep Deterministic Policy Gradient 31, 32, 33, 34, 35
DDQN	Double Deep Q-Network xii, xiv, 44, 45, 58, 61, 64, 66, 67, 69, 72, 73, 75, 77, 94
DL	Deep Learning 2, 22, 23, 27, 28, 44, 64
DLC	Dynamic Link Cost 18, 19, 20
DLCMI	Dynamic Link Cost and Minimum Interference 18, 19, 20
DNN	Deep Neural Network 22, 24, 27, 28, 29, 31, 32, 66
DORA	Dynamic Online Routing Algorithm 20
DPG	Deterministic Policy Gradient 30
DPML	DataMiner Protocol Markup Language 41
DQN	Deep Q-Network xii, xiv, 28, 29, 30, 31, 44, 45, 48, 58, 61, 64, 66, 67, 69, 72, 73, 75, 81
DRL	Deep Reinforcement Learning xiv, 2, 3, 27, 28, 30, 31, 32, 33, 34, 35, 36, 38, 48, 55, 58, 60, 61, 64, 66, 67, 68, 74, 75, 77, 78, 79, 80, 81, 91, 92, 93, 94
DROM	DDPG Routing Optimization Mechanism 33, 34
DSP	Dynamic Shortest Path 19, 50, 74, 79

ACRONYMS

Dueling DDQN	Dueling Double Deep Q-Network 36
Dueling DQN	Dueling Deep Q-Network xii , xiv , 45 , 58 , 61 , 62 , 64 , 66 , 68 , 69 , 72 , 73 , 75
DWSP	Dynamic Widest Shortest Path 19
DWT	Discrete Wavelet Transform 17
ES	Exponential Smoothing 14
FD	Forwarding Device 6 , 7 , 20 , 32
FFNN	Feedforward Neural Network 16 , 17 , 23 , 24 , 25 , 36
FIFO	First-In First-Out 44
GNN	Graph Neural Network 16 , 17
GRU	Gated Recurrent Unit 17 , 25 , 36
GUID	Globally Unique Identifier 39 , 52
HD	High Definition 10
IDP	Infrastructure Discovery and Provisioning 18
ILIOA	Improved Least Interference Optimization Algorithm 20
KPI	Key Performance Indicator 17
LIOA	Least Interference Optimization Algorithm 19 , 50 , 74 , 79
LLDP	Link Layer Discovery Protocol 18
LSTM	Long Short-Term Memory 17 , 25
MA	Moving Average 14
MAC	Media Access Control 57
MAPE	Mean Absolute Percentage Error 12 , 15
MCR	Master Control Room 10 , 11
MDP	Markov Decision Process 26 , 32
MHA	Minimum Hop Algorithm xiv , 19 , 50 , 74
MIB	Management Information Base 40
MIRA	Minimum Interference Routing Algorithm 20
ML	Machine Learning vii , 2 , 3 , 18 , 21 , 22 , 27 , 28 , 61 , 81
MLP	Multilayer Perceptron 16
MOS	Mean Opinion Score 32 , 33
MSE	Mean Squared Error 24 , 63

NFV	Network Functions Virtualization 8, 9, 10
NMS	Network Management Station 40
NN	Neural Network xii, 2, 16, 17, 22, 23, 24, 25
NNE	Neural Network Ensemble 16
NSFNET	National Science Foundation Network 35, 36
OID	Object Identifier 40, 41
ONF	Open Network Foundation 5, 7
OSPF	Open Shortest Path First 20, 36
OTT	Over-the-top 10
PER	Prioritized Experience Replay 35, 45, 81
ReLU	Rectified Linear Unit xii, 23, 24, 63
RL	Reinforcement Learning xii, 2, 21, 25, 26, 27, 28, 31, 44
RNN	Recurrent Neural Network 16, 17, 25, 80
RTT	Round-trip-time xiii, 64, 65, 75, 76, 78
SD	Standard Definition 10
SDI	Serial Digital interface 10, 11, 38, 39, 54, 66
SDMN	Software-Defined Media Network 1, 3
SDN	Software-Defined Networking vii, viii, xii, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 17, 18, 19, 20, 21, 31, 32, 35, 38, 41, 43, 68, 79
SGD	Stochastic Policy Gradient 30
SLC	Static Link Cost 18
SNMP	Simple Network Management Protocol 9, 40, 41, 42, 47, 50, 51
SP	Shortest Path 19, 35
SPF	Shortest Path First 20, 35, 36
SRM	Service Resource Manager xii, 9, 10, 11, 18, 38, 39, 40, 43, 46, 52, 53, 54, 55, 56, 60, 80
SSE	Sum Squared Error 12
TD	Temporal Difference 28
TE	Traffic Engineering 2, 17, 24, 34, 35
TL	Transfer Learning 81
TSF	Time Series Forecasting 12, 16
VNF	Virtual Network Function 8, 9
WSP	Widest Shortest Path 19, 20

INTRODUCTION

Skyline Communications, the “global leading supplier of end-to-end multi-vendor network management and OSS solutions for the broadcast, satellite, cable, telco and mobile industry” [6], exposed a problem to be solved during the time span of this studentship program.

DataMiner uses a [Software-Defined Networking \(SDN\)](#)-like framework by implementing a centralized point of control, monitoring and orchestration of a customer’s network. Despite DataMiner’s capability to be applied to all sorts of systems, the multimedia industry is a big focus of this company, hence the designation in this work’s title of [Software-Defined Media Networks \(SDMNs\)](#).

In its current state, DataMiner relies on the Dijkstra algorithm to route multimedia streams between endpoints of a managed video streaming service. Before that service is deployed, the algorithm finds multiple routes for every target node in the available network topology and lets the user choose the path he wants his data to go through. Not only does the user make this choice without real knowledge of the network state, but also Dijkstra is a naive approach to routing, since its output only considers the cost of the links and not the network traffic distribution.

Therefore, to create a load-aware, quality-focused, dynamic routing solution capable of finding optimal routes for these services, an algorithm that considers the network state when computing its paths is required, for which constantly updated information about the traffic matrix is essential. This will promote a better utilization of the network resources, assuring that they are not being under or overused, which represents a loss of profitability or causes deterioration in the service quality, respectively. However, to achieve this idealized solution is a problem classified as NP-hard.

Moreover, during an important streaming event, like a sports match or a presidential election live coverage, it is usual to have network operators working around the clock, leading to exhaustion and error propensity. Thus, an important action taken by companies at times like these is to give rest to the operators in preparation of such large-scale events, leaving the offices nearly empty in its’ eyes. To allow this kind of management, a certain level of automation in processes is required, which opens the solution’s door to

technologies built with [Machine Learning \(ML\)](#).

1.1 Motivation

Computer networks are highly dynamic and, at times, unpredictable. The most common approach to perform [Traffic Engineering \(TE\)](#) in these networks is to create complex traffic models that try to comprehend their behaviour. However, as mentioned before, to optimally route traffic considering its properties and the networks' dynamism is a problem of intractable complexity. Additionally, these traffic models will always be static and unable to cope with networks' variability.

[ML](#) and related technologies like [Neural Networks \(NNs\)](#) and [Deep Learning \(DL\)](#) are also a hot topic in today's industry. A subcategory of [ML](#) that is being explored to answer resource allocation and [TE](#) problems is [Reinforcement Learning \(RL\)](#) or, specifically, [Deep Reinforcement Learning \(DRL\)](#). This is because these [DRL](#)-based solutions can understand the network's behaviour through experience and without the need to create a mathematical traffic model, which significantly reduces their complexity.

Moreover, DataMiner's [SDN](#) nature allows for flexible and programmable networks. Thus, leveraging a global view of the network and its state, decisions and changes can be made towards the improvement of the services' performance and applied in a single point of centralized control.

In this dissertation, we will attempt to combine DataMiner's [SDN](#)-like architecture with [DRL](#)'s applicability to complex routing problems by training the algorithm to make [TE](#) decisions that can be installed in the form of rules through [SDN](#)'s capabilities. Such framework will grant us with the possibility to understand if [DRL](#) can be used in a practical scenario, dissipating doubts that exist on its scalability when network size and/or traffic load increase.

1.2 Objectives

The goals to be accomplished in this dissertation are as listed:

1. In a first stage, it is expected to understand the fundamentals of routing in software-defined networks, evaluate Skyline's requirements for the new solution and investigate ways to transform the current routing mechanism into a dynamic algorithm.
2. Use heuristic routing optimization techniques, such as link cost modeling, to improve the current static routing solution by accounting for network load. This should help develop a better understanding of DataMiner and how to utilize Skyline's resources.
3. Develop a dynamic routing algorithm using [DRL](#) that can adjust routes based on the conditions and changes in the network's state.

4. Produce an analysis on how the developed solution compares to the current routing scheme.

1.3 Contributions

Regarding the actual dissertation's contributions, we list the following ones:

1. A comprehensive study on routing optimization techniques to support the developed work, both with and without the use of [ML](#), according to the system's architecture.
2. A benchmark of dynamic link cost routing optimization solutions and their comparison to Dijkstra.
3. A brand-new routing algorithm based on [DRL](#) that accounts for bandwidth reservation state in each of the network's links.
4. A detailed analysis of the developed algorithm exploring severe network traffic conditions, different traffic types and two distinct network topologies, as well as a comparison between its performance and the baseline solution's.

1.4 Outline

This report continues with the following structure.

Chapter 2 introduces the related work considered in this dissertation's development, starting with brief introductions to the base concepts and tools to be used and followed by a critical analysis of different approaches applied to solve similar problems in the current literature. Namely sections on:

1. [SDN](#)'s architecture and its relationship with OpenFlow and virtualization.
2. Multimedia content delivery with [SDMNs](#).
3. Different methods for load prediction in software-defined networks.
4. Routing optimization through link cost modeling techniques.
5. The application of [ML](#) to routing optimization with emphasis on [DRL](#)-powered solutions.

Chapter 3 details the framework where the developed solution must be implemented, the required technological tools used to develop and test that solution and its complete design.

Chapter 4 describes the implementation of the solution presented in Chapter 3.

Chapter 5 describes the performed simulations and presents the results comparing the proposed solution to the baseline and other intermediate alternatives.

Lastly, Chapter 6 presents the conclusions on the developed work and points towards ways to improve and extend it.

STATE OF THE ART

2.1 Software-Defined Networking

2.1.1 Towards Software-Defined Networking

A traditional network is embodied by different elements, from multiple providers, known to be vertically integrated, which means that the control (responsible for traffic routing decisions) and data (responsible for forwarding traffic at the control plane's command) planes are wrapped inside every device in the network [7]. This poses obstacles to networks' evolution since their architecture imposes the updating of every device when a new feature is added or an existing is removed.

The appearance of the virtualization paradigm and the concept of virtual machines allowed for a system (i.e. a physical host) to execute a variable number of client operating systems. This, allied with the advances in memory, storage, and computing power of computers, led to the appearance of datacenters capable of executing and controlling thousands of virtual machines [8]. Thus, what was once required to execute in a physical machine (e.g. running windows) was now being emulated in virtual environments, which allowed for operation centers to process these environments as files, pausing them and passing them through physical devices with ease.

This has steered towards a set of network specifications that traditional networks were not able to cope with. Virtual machines migrate in the physical datacenter network with great dynamism, while traditional networks are much more static. By enabling centralized control through software and increasing network scalability, [SDN](#) allowed for the needed dynamism to support the migration to virtual machines in datacenters.

2.1.2 Software-Defined Networking: A definition

In its initial presentation, [SDN](#), standardized and commercialized by the [Open Network Foundation \(ONF\)](#), was introduced as "A network in which the control plane is physically separate from the forwarding plane, and a single control plane controls several forwarding devices" [9].

In other words, **SDN** builds upon the idea of separating the control and data planes in devices, breaking the vertical integration that characterized them up until then. By making this change, it is possible to have a logically centralized unit, the *SDN controller*, managing the activity of many packet *Forwarding Devices (FDs)* through programming, while leveraging an abstract view of the network topology. Such separation allows for the complexity of protocols and routing functions to be dispatched to the controller, reducing the intelligence required from **FDs**, consequently lowering their cost, and ultimately facilitating the management and configuration of the network according to traffic flows [7][10][11].

2.1.2.1 Software-Defined Networking: Architecture

SDN is organized into three main layers, or planes, as illustrated in Figure 2.1.

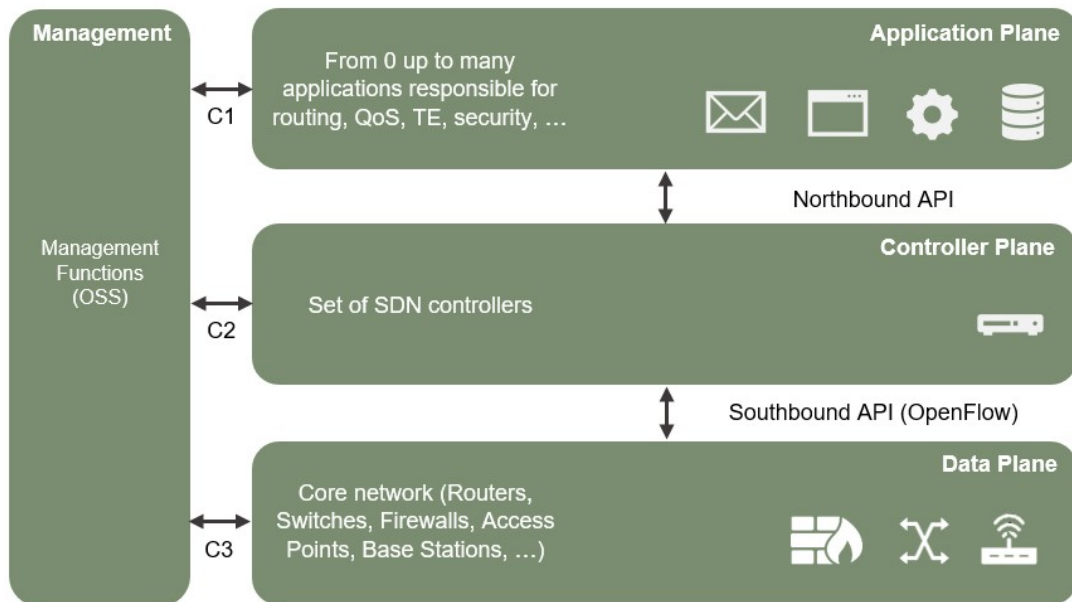


Figure 2.1: **SDN** fundamental architecture (adapted from [7][12][13][14][15]).

In the *data plane*, the capabilities of each physical device are represented virtually as resources at the **SDN** controller's disposal. Following a flow¹ based model, these devices, or *switches*, apply to flows the rules and policies sent by the *control plane* through the *southbound API*, using protocols such as OpenFlow [7][11][15][16].

In a scenario with multiple **SDN** controllers, each one has power over a range of data plane resources provided by one or more devices (i.e. its control domain). **SDN** controllers expose network resources to applications in the *application plane* through the *northbound API* and answer to services' requests that use them. The advantage of having many **SDN** controllers is the resilience they offer in guaranteeing adequate levels of performance,

¹Refers to the path of packets with equal source and destination between them.

scalability and robustness. The coordination of its control domain by an SDN controller is called *orchestration* [7][11][15].

The *management* or *administration* layer interfaces with each plane, performing resource allocation for clients and establishing a direct connection between the data and application planes [15][16].

2.1.3 OpenFlow

With the decoupling of functions that were once wrapped together came the need to create an interface that could maintain communication between the control and data planes. That interface is OpenFlow, which is a protocol also managed by ONF. Mainly, changes introduced by the OpenFlow protocol occur at the FD level, whose previous intelligence is now redirected to the SDN controller.

This renewed device, the *OpenFlow switch*, has two segments separated by an abstraction layer, a set of flow tables and a secure channel for communicating with the managing SDN controller [17].

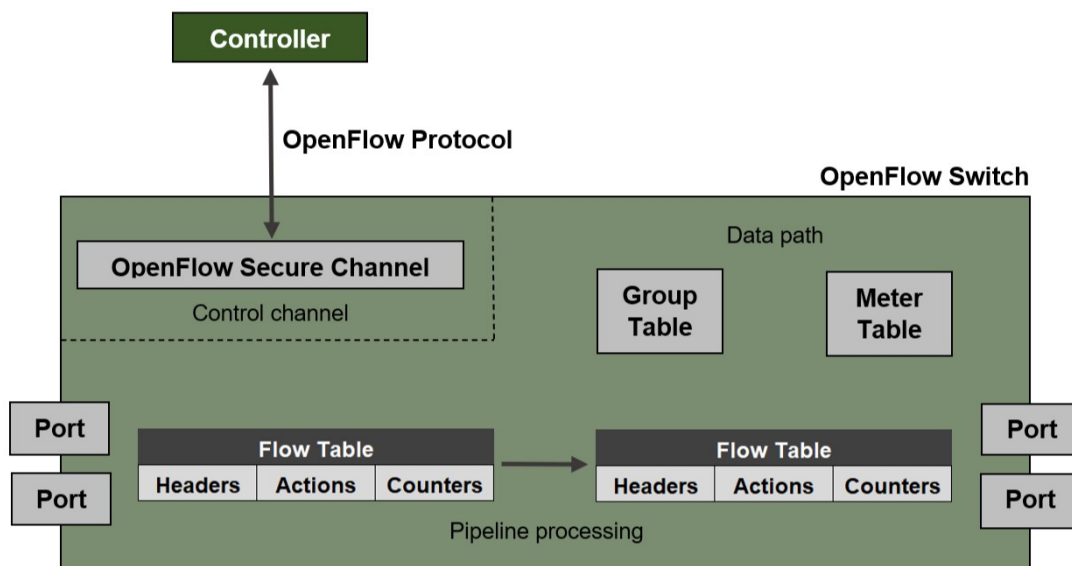


Figure 2.2: OpenFlow switch's architecture (adapted from [18][19]).

A flow table acts as a group of flows, or flow entries. It is through a flow entry that each packet crossing the network gets processed, by following the rules of the flow they belong to. To achieve this purpose, flow entries are described by a few different arguments [18][20]:

1. **Match fields:** Used to validate incoming packets through comparison with their headers, ingress port and metadata. When a packet enters a switch, its header fields are extracted and crosschecked with the match information of the flow entries inside the flow table. If the matching is successful, the received packet belongs to the flow in question and is processed according to its rules.

2. **Actions/Instructions:** Describe the instructions given to a packet that passes the flow table look-up phase.
3. **Counters:** Important for flow statistics purposes, recording packets received, packet byte count and flow duration.
4. **Priority:** Defines the order in which flow entries are organized inside flow tables.
5. **Timeouts:** Either maximum lifetime or idle time of a flow entry before expiring.
6. **Cookie:** Describes the filtering the controller demands for the flow entries.

To deal with the possibility of an unmatched packet, a table-miss entry with priority 0 must exist in every flow table specifying how to proceed in such situation. To respond to this event, the packet may be dropped, matched with a different flow table thanks to pipeline processing², or forwarded to the controller [18][20].

To enable the pipeline processing, a group table is required to provide additional forwarding methods, like broadcast or multipath. Furthermore, a meter table is used for QoS purposes [18][20].

2.1.4 Virtualization and SDN

A key partner in SDN's ascension is virtualization. Although these two concepts are commonly referred to interchangeably, it is wrong mixing them as they have different purposes and capabilities. Having said that, the combination of SDN and Network Functions Virtualization (NFV) is one of the key factors for the success of both technologies [21][22].

2.1.4.1 Network Functions Virtualization

NFV is based on replacing network functions, once provided by vendor-specific hardware, with software-based applications running on virtual machines. These functions are then assigned to the control of a *hypervisor* [22][23].

Generically, *Virtual Network Functions (VNFs)* are obtained through three main components [22]:

1. **Physical Server:** Corresponds to the hardware running the function.
2. **Hypervisor:** Has a monitoring and managing role. It is the software controlling function resources.
3. **Guest virtual machine:** Represents the final product which is a functional software version of a hardware equipment. Applications can then run on top of this virtual machine and execute the functionalities it provides.

²Introduced in OpenFlow specification 1.1.0 with the coexistence of multiple flow tables in a single switch. These tables are linked with each other through pipeline processing.

2.1.4.2 NFV and SDN integration

SDN and NFV both rely on network abstraction. SDN focuses on decoupling the control and data planes, separating network control functions from data forwarding functions, but maintaining the amount of equipment in the network. Contrarily, NFV targets the abstraction of each of these functions from the hardware they run on, allowing the network to expand without adding more devices [16].

Despite their differences, the collaboration of both technologies is greatly beneficial. SDN helps NFV by connecting VNFs through programmable interfaces which are managed by their orchestrator according to network needs. In regards to SDN, NFV allows for the implementation of network functions through software, enabling multiple network architecture possibilities and allowing for their configuration and behaviour to be altered programmatically [22][23].

2.1.5 DataMiner's approach to SDNs

Throughout this section, the most strict and pure vision of SDNs has been introduced. However, the transition from traditional to completely SDN-ready networks is a very complex process. For that reason, many systems nowadays opt for less radical approaches to SDN.

An example of these approaches is the *Control Plane/Broker SDN*, which translates to networks that maintain the existing control planes in the devices. However, such devices offer APIs that allow applications to communicate with and operate them [10]. In this case, a SDN controller, also called orchestration platform, mediates communication between applications and the network devices. This hybrid SDN model allows for network programmability through network protocols such as *Simple Network Management Protocol (SNMP)*, *NETCONF*, or *RESTCONF*, that enable the broker to retrieve information from the devices and manipulate their control planes (e.g. install forwarding rules) [10].

Skyline's DataMiner supports this and other approaches to SDN. In its specific architecture, the roles of both an SDN controller and an orchestration layer are combined to provide SDN orchestration and encapsulated in *Service Resource Manager (SRM)*, a DataMiner module that will be explained in more detail in Chapter 3. In this environment, the controller is responsible for keeping a copy of the network topology, its links to media devices and the bandwidth of each network link. With this information, it can calculate efficient paths and install, for instance, static multicast routes in the network's devices. This installation relies on the device's vendor and supported protocols. Then, the orchestration layer, also called *service orchestrator*, operates on top of the SDN controller and allows for resource scheduling, enabling the prediction and allocation of the network capacity in the future [24].

2.2 Software-Defined Media Networks

Before the rise of **SDN** and **NFV**, the only possible solution for the distribution of multimedia content was through best-effort internet [25].

Multimedia consumption has gone exponentially upwards in the last few years, with companies like YouTube, Netflix or HBO being great examples of this demand [26][27]. The dominance of network traffic belongs to video and its impact is getting heavier due to the improvement of device's screens, with 4 and 8K resolutions now being common, which calls for an higher bitrate when compared to **Standard Definition (SD)** or **High Definition (HD)** video [26].

Such high interest in these services, as well as their network load, has posed difficulties in their transportation, such as bandwidth fluctuations caused by network congestion [25]. In spite of the various solutions designed to improve the delivery of these contents, many face problems like network bottlenecks and congestion, related to the lack of a global network view [25], which affect the quality of service provided (**QoS**) and the customer's quality of experience (**QoE**). Both **QoE** and **QoS** are metrics used to assess the service quality from a customers' perspective or through network analysis, respectively. From the scope of multimedia providers, a network capable of supporting such quality requirements is expected.

Skyline's product, DataMiner, is comprised of a set of modules that together provide control and monitoring of networks in order to meet multimedia providers' demands. The **SRM** module's goal is to encapsulate automation and orchestration capabilities and apply them to multiple types of operating infrastructures such as:

1. Automated broadcasting of live or file contents.
2. Remote production of broadcast feed when the video source and the broadcast production are in different locations.
3. **Master Control Room (MCR)** automation.
4. **Over-the-top (OTT)** content delivery (i.e. through shared internet).
5. **SDN** network control for media **IP** streams (e.g. using Dijkstra's algorithm).

Currently, there is an ongoing migration towards **ALL-IP** broadcast infrastructure, which means that providers are adopting **IP**-based technologies for the distribution of cable services to their managed networks [28]. This is in pursuit of flexibility, agility and video quality, achieved with less cabling, equipment and bandwidth required for the media delivery. Such process is not just about replacing traditional **Serial Digital interface (SDI)**, but also any other cabling out of the **IP** scope and incorporating the use of **Commercial-off-the-shelf (COTS)**³ hardware that supports media of multiple formats and is easily managed and updated [29].

³Products that are sold and afterwards tailored to the specifications required from them.

Additionally, IP supports higher resolutions than SDI, handles video, audio and data separately, and allows for plug-and-play, which is a capability of systems that automatically fetch available sources and destinations in the network, simplifying the installation and expansion of IP networks [29].

In the way of ALL-IP are a series of challenges, such as the increased number of flows to monitor, the complexity involved in that task and the possible increase in delay variation, jitter or latency. Another issue concerning ALL-IP implementations is security. Since infrastructures are no longer physically static and enclosed, a stronger authentication mechanism is required to protect media contents [28].

ALL-IP is a great example of SRM's importance because it revolves around many different multi-vendor technologies with distinct characteristics, which calls for the existence of a flexible monitoring and orchestration platform. This control layer must consider the resources' capabilities, constraints, capacities and availability, in order to prevent over-subscription in blocking or non-blocking topologies [28]. These network properties refer to the limitations that it imposes in terms of bandwidth. In a blocking topology, the link bandwidth capacities might be a bottleneck, which requires an operator to account for and provision them. On the other hand, in non-blocking topologies, the link capacities are usually much greater than the bandwidth requirements, which allows the operator to perform given tasks without considering the links' utilization rates.

In a common MCR, there is a prior booking of resources and a guarantee of capacity and availability, which is manageable thanks to the blocking nature of the infrastructure. However, when it comes to live broadcasting, "most uncompressed video-over-IP implementations are based on a non-blocking infrastructure" [28], which means that broadcast controllers cannot manage such aspects of the resources, leading to underutilization and overprovisioning of the infrastructure [28]. Yet, at the same time, in these environments there is the need for fast switching, which is contradictory to the nature of blocking infrastructures.

Thus, the combination of SDN controllers to execute fast commands and an orchestration layer to coordinate the network resources assuring their bandwidth availability is crucial to bring blocking infrastructure's properties (i.e. bandwidth guarantees from resources) into a non-blocking environment where fast switching is required [28].

2.3 Traffic forecasting

Before diving deeper into this subject, we alert to the fact that this section's subject is not crucial to the document's understanding since it is not used in the solution proposed. However, it can be useful for a reader who intends to continue this dissertation's work.

Given this raising demand for multimedia services, it is important that multimedia providers try to predict and model traffic demands so that they can understand their

customers' viewing patterns and optimally provision network resources accordingly. Although currently this task is performed by experienced network operators [30], forecasting network traffic in TCP/IP networks can greatly reduce the gap between the administrators' prediction and the real network traffic levels. Such accuracy would, ultimately, allow for routing, resource allocation or admission control optimization, consequently improving QoS [30][31].

2.3.1 Time Series Forecasting

Time is a key factor in computer networks since it determines satisfaction, both the customer's, measured by delay experienced while consuming multimedia services, or the provider's, from an economic perspective. Information flows from these providers to their customers in packets sent at consecutive time steps, which gives these types of networks a temporal dependency [31]. As a result, Time Series Forecasting (TSF) is a popular mathematical process for computer networks' traffic forecasting, as it acts upon chronologically ordered variables [30].

A time series consists of a group of observations x_t performed sequentially in time and recorded at specific time steps t [32][33]. These can be further discriminated into discrete and continuous time series, distinguished by if either we can or cannot enumerate every time step under analysis. In the latter case, it is common to have continuous variables digitized into stamped time intervals, originating a discrete time series with minimal loss of information in the process if the chosen time intervals are sufficiently small [32]. Most applications for TSF are, thus, built from discrete time samples, either naturally discrete, transformed by sampling from a continuous variable as previously described, or through aggregation of values over a period of time [32]. An important concept is that observations in a time series are not independent. Because of the influence that observations have on each other, the order in which they are registered matters.

A TSF model expects past patterns to repeat in the future, with the time between an occurrence and its repetition being called *lead time* h . The accuracy of a time series predicting model can be measured by the Sum Squared Error (SSE) and Mean Absolute Percentage Error (MAPE) indicators, which are described by:

$$e_t = y_t - \hat{y}_{t,t-h}, \quad (2.1)$$

$$SSE_h = \sum_{i=P+1}^{P+N} e_i^2, \quad (2.2)$$

$$MAPE_h = \sum_{i=P+1}^{P+N} \frac{|e_i|}{y_i \times N} \times 100\%, \quad (2.3)$$

with e_t representing the prediction error at instant t , y_t the target value, $\hat{y}_{t,t-h}$ the value forecast, P the current time and N the number of predictions [30]. The MAPE metric is

broadly utilized because it assesses the relation between the result of the forecast and the real value observed.

Through the model's evaluation, its parameters can be adjusted, and results checked against actual future observations. This is called *out-of-sample* behaviour, as opposed to *in-sample* fit, in which the fitness of the model is measured by matching the data used for model estimation with itself [32].

From the analysis of time series, a few types of variations can be extrapolated. These describe the patterns of the data being monitored and can be listed as [32][33]:

1. **Seasonal variation:** Periodic patterns at specific segments of the year. An example is the search for vacation bookings in the summer.
2. **Trend:** Describes a steady growth or decrease of a value, like some stock market shares.
3. **Other cyclic variation:** Cyclic patterns that are verified with different frequency than a year.
4. **Irregular fluctuations:** Any other events, such as random observations or unaddressed tendencies from the previous bullet points.

Additionally, forecasting types can also be classified with respect to their time span, starting from real-time (i.e. a few minutes before) to long-term, verified when a forecast is issued months or even years in advance (e.g. investments) [30].

In the following subsections, the [Autoregressive Integrated Moving Average \(ARIMA\)](#) and Holt-Winters forecasting methods are presented, whose appliance to computer networks has been broadly explored.

2.3.1.1 ARIMA models

[ARIMA](#) represents a class of models that integrate different components, hence its name [34]. Such models emerged from the concept of future values being forecasted from white noise characteristics and past values [31]. [ARIMA](#) is mostly applied to short-term forecasting [34].

2.3.1.1.1 Autoregressive processes

[Autoregressive \(AR\)](#) models make use of a weighted linear sum of the last p observations, also called lagged observations, plus a random shock or white noise (i.e. a value from the distribution with constant variance and zero mean), with p corresponding to the order of the model. This way, the following elements in the time series are predicted [32][33]. Thus, a time series X_t can be represented by:

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \dots + \phi_p X_{t-p} + Z_t, \quad (2.4)$$

where X_n are previous values, ϕ_n the associated weights and Z_t the white noise.

2.3.1.1.2 Moving average processes

On the other hand, **Moving Average (MA)** processes use the dependency between an observation and the associated error by summing the last q weighted shocks to the series' average value:

$$X_t = \mu + \theta_1 Z_{t-1} + \dots + \theta_q Z_{t-q}, \quad (2.5)$$

where Z_t represents each past shock, θ_j their respective weights and μ the time series' average value [32][33].

2.3.1.1.3 Autoregressive integrated moving average processes

Ultimately, **Autoregressive Moving Average (ARMA)** methods combine the two formerly presented processes by using both a linear combination of the last p observation values and the last q forecasting errors to predict a future value. Hence its **ARMA**(p, q) terminology. However, most time series are non-stationary⁴ and so a differentiation is applied to convert them, turning **ARMA**(p, q) into **ARIMA**(p, d, q), where d stands for the number of differentials performed on the time series [32][33]. This model can be defined as:

$$\begin{aligned} \phi(B)(1 - B)^d X_t &= \theta(B)Z_t, \\ \phi(B) &= 1 + \phi_1 B + \dots + \phi_p B^p, \\ \theta(B) &= 1 + \theta_1 B + \dots + \theta_q B^q, \end{aligned} \quad (2.6)$$

where X_t denotes the time series, Z_t the random shock and $\phi(B)$ and $\theta(B)$ are polynomials from the **AR** and **MA** processes, respectively. Additionally, B stands for the backward shift operator that can, from an element of the time series, produce its predecessor.

2.3.1.2 Holt-Winters' method

Holt-Winters' method is based on the capture of trends and seasonal variations distinguishable from noise by averaging past values observed [30].

To better understand how Holt-Winters' method works, it is important to study **Exponential Smoothing (ES)**, the family of methods to which Holt-Winters belongs [33]. A method has such designation when it produces forecasts through weighted averages of past observations, considering an exponential decay in their importance as new ones are recorded [35].

Holt-Winters is a seasonal method that deploys the forecast equation and three smoothing expressions that correspond to the *level*, *trend* and *seasonal* components. Depending

⁴Stationary time series have constant properties independently of the observation time, as well as the parameters' mean and variance. This is a requirement for time series analysis [35].

on how stable the seasonal variations of the time series are, Holt-Winters' method provides two approaches: *additive* and *multiplicative*. The additive variant is used for nearly constant seasonal variations along the series' duration and is expressed in absolute scale to the time series [35]. At the end of each year, the seasonal components' sum should be close to zero. The Holt-Winters' additive method is described mathematically as:

$$\begin{aligned}
 \hat{y}_{t+h|t} &= l_t + hb_t + s_{t+h-m(k+1)}, \\
 l_t &= \alpha(y_t - s_{t-m}) + (1 - \alpha)(l_{t-1} + b_{t-1}), \\
 b_t &= \beta^*(l_t - l_{t-1}) + (1 - \beta^*)b_{t-1}, \\
 s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1 - \gamma)s_{t-m},
 \end{aligned} \tag{2.7}$$

where l_t , b_t and s_t stand for the level, trend and seasonal equations, while α , β^* and γ correspond to the smoothing parameters, respectively. The letter m is used to state the seasonality frequency (e.g. if monthly, $m = 12$) and $k = (h - 1)/m$ [35].

On the other hand, the multiplicative method thrives when applied to increasing seasonal variations. In this case, the seasonal component is depicted in percentages. At the time span of one year, the seasonal component will, approximately, amount to m [35]. This variant is described as:

$$\begin{aligned}
 \hat{y}_{t+h|t} &= l_t + hb_t + s_{t+h-m(k+1)}, \\
 l_t &= \alpha \frac{y_t}{s_{t-m}} + (1 - \alpha)(l_{t-1} + b_{t-1}), \\
 b_t &= \beta^*(l_t - l_{t-1}) + (1 - \beta^*)b_{t-1}, \\
 s_t &= \gamma \frac{y_t}{l_{t-1} - b_{t-1}} + (1 - \gamma)s_{t-m}.
 \end{aligned} \tag{2.8}$$

2.3.1.3 TSF techniques and their characteristics

Many studies on the performance of both [ARIMA](#) and Holt-Winters have been conducted ([36][37][38]). However, it cannot be stated that one of these models is consistently better than its peer, since both outperform each other in some cases, while in others it is the other way around.

[ARIMA](#) is a complex model but shows high reliability over a wider set of series than Holt-Winters. To achieve such high levels of accuracy, [ARIMA](#) makes assumptions upon the data it is presented with. This is a constraint because it makes the model's results vary from highly precise when the assumptions are correct, to inaccurate when the assumptions are wrong [36].

On the other hand, Holt-Winters is simple, computationally effective and of low resource cost, but tends to show lower levels of accuracy when measured by the [MAPE](#) metric [30][36]. In spite of this, Holt-Winters' results are still accurate, especially when

using more complex variations of the native method. It is also recommended that Holt-Winters' predictions do not exceed the seasonal cycle of the time series, so that a loss of accuracy can be avoided [37].

Both [ARIMA](#) and Holt-Winters are linear forecasting models, which means that they are best suited to evaluate datasets where each variable's behaviour, independently, can be represented by a straight line, which does not imply that the combination of many variables must also result in a constant slope. Furthermore, both methods require the data to be *stationary*, as mentioned earlier, and *self-similar* (i.e. at least approximately similar to the whole), independently of the segment selected for analysis [39].

An alternative approach to traffic forecasting with [TSF](#) that is not limited by these aspects is the use of [NNs](#). [NNs](#) can learn from both linear and nonlinear⁵ inputs and are noise tolerant, thus being able to establish predictions upon imperfect datasets [41].

2.3.2 Traffic forecasting with Neural Networks

Since this section is of optional reading and [NNs](#) are a relevant topic to this dissertation as a whole, their detailed explanation can be found in Section 2.5.2. However, to give a broad idea of the topic, [NNs](#) are able to detect patterns and extract features from the data they are fed with, making them a valuable option for traffic forecasting. Additionally, it is important to note that many variations of [NNs](#) exist, with different characteristics and purposes, such as [Feedforward Neural Networks \(FFNNs\)](#), [Recurrent Neural Networks \(RNNs\)](#) or [Graph Neural Networks \(GNNs\)](#). A few studies that made use of these types of networks are hereby presented.

In [30], the authors chose to focus their attention on previous studies that used [Multilayer Perceptron \(MLP\)](#)⁶ networks with one hidden layer containing N hidden nodes. With this approach, the utilization of [NNs](#) is approximated to [TSF](#) by a technique called *Time Lagged Feedforward Network*, where a sliding window concept is applied by building forecasts based on a predefined number of previous inputs. To improve the model's accuracy, since it is greatly influenced by the randomly generated initial weights, the authors resorted to the use of a [Neural Network Ensemble \(NNE\)](#), where multiple networks are trained concurrently, and the result is obtained from averaging all networks' outputs. After testing different setups, with distinct numbers of lagged steps, hidden nodes or lead times⁷, it could be stated that the [NNE](#) approach to forecasting resulted in more accurate outputs across almost every setup when compared to linear approaches (i.e. [ARIMA](#) and Holt-Winters).

Despite the positive results shown by the use of [MLP](#) networks, these face performance issues while modelling certain time-dependent dynamic systems. In contrast, [RNNs](#) are suitable for such systems and are explored in [31]. Besides exploiting the

⁵"Linearity or nonlinearity of a dynamic system is associated with the differential equation that defines the behaviour of that specific system"[40].

⁶These networks belong to the feed forward family.

⁷From online configurations to hours.

performance of these networks on their own, the authors also evaluate how a joint solution combining both [ARIMA](#) and [RNNs](#) performs. The goal is to separate the input data through a [Discrete Wavelet Transform \(DWT\)](#) into low and high frequency component batches, ultimately distinguishing nonlinear from linear data, submitted to [RNNs](#) and [ARIMA](#), respectively. The resulting forecasts are then summed, producing the final forecast. By analysing the results from both [ARIMA](#), [RNNs](#) and their combination, the [NNs](#)-based technique surpasses [ARIMA](#) in most cases, while their conjugation shows the best performance.

However, traditional [RNNs](#) still have challenges that can be surpassed by [Long Short-Term Memory \(LSTM\)](#) networks, which are a specific type of [RNNs](#). In [42] and [43], these networks are explored in a [SDN](#) context, which allows for easily collected link statistics. Both of these works conclude that [LSTMs](#) perform significantly better than [ARIMA](#) and Holt-Winters. Additionally, in [42], [LSTMs](#) were also found to be a better solution than [FFNNs](#) while using a similar sliding window feeding technique to the one exploited in [30].

Another kind of [RNNs](#) commonly applied to this field are [Gated Recurrent Units \(GRUs\)](#), which have a similar but simpler architecture when compared to [LSTMs](#). In [44], [GRUs](#) were evaluated against both traditional [RNNs](#) and [LSTMs](#) and showed oscillatory results in comparison.

Although these techniques produce good estimations when compared to the traditional linear methods, computer networks are typically represented as graphs, making the presented solutions not best suited for their modelling. For this reason, studies ([45][46]) present [GNNs](#) as an alternative, since they thrive in processing data represented in graphs. Both solutions are enclosed in a [SDN](#) environment and are capable of transforming network data (i.e. topology, routing list and a traffic matrix defined by the bandwidth between network nodes) into valuable [Key Performance Indicators \(KPIs\)](#). Through this methodology, the topological relations between adjacent nodes in the network can be considered, optimizing the considered [KPIs](#).

2.4 Routing

Once a load prediction model is built, it is easier to use [TE](#) concepts to soften the impact of congested periods and, consequently, optimize the network's performance. An example of these techniques is *routing optimization*, which can be accomplished, for instance, through the re-routing of traffic from congested links to freer ones (i.e. *load balancing* through packet forwarding rules), allowing the network to become more robust and maintain a higher [QoS](#).

Routing can be *static* or *dynamic* in terms of the link costs considered. Static routing does not take into account fluctuations in the network usability, since the networks' link costs are assigned and never altered. This causes an uneven traffic distribution that incites

link congestion and a resource misuse [47]. In dynamic routing, the implemented solution supports networks' variability by adjusting its rules according to updated network conditions, which are available due to periodic link costs and routing tables' updates. This dynamic property can be achieved through both linear optimization, where the goal is to optimize one or multiple network performance metrics (e.g. maximizing network utilization considering a set of constraints such as the underlying resources' capacities), and ML powered solutions.

2.4.1 Routing in SDN

SDN has had a great impact in this particular area by tackling previously existing difficulties. The concept of re-routing traffic is typically translated into a cost increment of the network's congested links, forcing the underlying routing algorithm to choose more vacant connections that ultimately lead to the same destination. The problem with this is that when the cost of a link is changed, it directly affects many traffic flows, hence introducing the possibility to negatively influence other segments of the network by the large-scale necessity to update routing information. Additionally, there was few information about traffic flows, which made their identification and the implementation of load balancing techniques harder [48].

SDN impacts typical routing by providing closed loop control, sending periodic network utilization information to different applications and allowing them to adapt their rules and, consequently, the network functioning [11]. Furthermore, the network stability is maintained since the SDN controller has the ability to apply rules to specifically targeted flows [48]. Hence, SDN offers a way to reduce problems such as slow convergence, complex implementations and lack of adaptability.

In Skyline's solution, traffic routes are calculated with Dijkstra, which uses a graph network view to find the shortest path considering the network's link costs between edges. Furthermore, these network topologies can be built manually (i.e. in SRM) or discovered using DataMiner's [Infrastructure Discovery and Provisioning \(IDP\)](#) application. IDP searches for devices through network protocols, which are selected according to the devices' manufacturer and type. In addition, when connected to a device through any API, IDP also retrieves connectivity information by reading [Link Layer Discovery Protocol \(LLDP\)](#) data. In this protocol, the different nodes in the network communicate with each other by advertising their identity and capabilities.

2.4.2 Optimization-based dynamic routing

Throughout this section, a few optimization-based dynamic routing implementations using different techniques and link cost equations will be explored.

A thorough analysis of existing routing solutions is offered by [49]. The authors start by dividing existing algorithms into three categories: [Static Link Cost \(SLC\)](#), [Dynamic Link Cost \(DLC\)](#) and [Dynamic Link Cost and Minimum Interference \(DLCMI\)](#) algorithms.

For static algorithms, the SDN controller determines paths using unchangeable values as link costs, such as the hop count in [Minimum Hop Algorithm \(MHA\)](#), the inverse of the link's bandwidth capacity in [Shortest Path \(SP\)](#), or the link's bandwidth capacity in [Widest Shortest Path \(WSP\)](#). These link cost equations can be represented as:

$$\begin{aligned}
 MHA : C_{(u,v)} &= 1, \\
 SP : C_{(u,v)} &= \frac{1}{BW_{(u,v)}}, \\
 WSP : C_{(u,v)} &= BW_{(u,v)},
 \end{aligned} \tag{2.9}$$

where $C_{(u,v)}$ is the cost of the link that connects nodes u and v and $BW_{(u,v)}$ is its total bandwidth capacity. In the first two cases, Dijkstra calculates the paths after the costs are assigned whereas the last algorithm resorts to a reverse version of Dijkstra where the link with higher cost is selected. Since the links' costs are always the same, the algorithm will constantly return the same path for a given edge pair. This offers some advantages by reducing path computation time, however, when the network load increases, the performance will deeply suffer due to the reasons stated in the last section.

Using the SDN controller's ability to collect network link's statistics in real time, such as bandwidth utilization, it is possible to define dynamic link costs and have them reflect the network's current state. Ultimately, the best path between two nodes in the network will change according to the network's state. This adaptability reduces congestion by promoting better load balancing and link utilization. In the presented study, two alternatives are introduced as [Dynamic Shortest Path \(DSP\)](#) and [Dynamic Widest Shortest Path \(DWSP\)](#). These algorithms are improved versions of the aforementioned static alternatives, which only differ from using the total link bandwidth capacity to the available bandwidth at the path's computation time:

$$\begin{aligned}
 DSP : C_{(u,v)} &= \frac{1}{RBW_{(u,v)}}, \\
 DWSP : C_{(u,v)} &= RBW_{(u,v)},
 \end{aligned} \tag{2.10}$$

where $RBW_{(u,v)}$ stands for the available bandwidth between the two nodes.

By adding metrics related to the quantity of flows carried by network links, the [DL-CMI](#) algorithms' category aims to optimize performance with emphasis on the delay and packet loss metrics, without sacrificing the bandwidth and throughput benefits of the [DLC](#) algorithms. The [Least Interference Optimization Algorithm \(LIOA\)](#) increments on [DSP](#)'s link cost equation by multiplying it by the number of flows being transported I and raising the fraction to a constant α :

$$LIOA : C_{(u,v)} = \left(\frac{I_{(u,v)}}{RBW_{(u,v)}} \right)^\alpha. \tag{2.11}$$

Therefore, the interference registered between source-destination pairs should decrease. The **Improved Least Interference Optimization Algorithm (ILIOA)** factors in link utilization in its link cost equation:

$$\begin{aligned}
 ILIOA : C_{(u,v)} &= (1 - U_{(u,v)}) \times \frac{I_{(u,v)}^\beta}{BW_{(u,v)}^\beta} + U_{(u,v)} \times \frac{I_{(u,v)}^\alpha}{RBW_{(u,v)}^\alpha}, \\
 U_{(u,v)} &= \frac{RBW_{(u,v)}}{BW_{(u,v)}} \times m,
 \end{aligned} \tag{2.12}$$

where $U_{(u,v)}$ is the link utilization, m is the number of links and β a constant.

The last **DLCMI** algorithm is the **Minimum Interference Routing Algorithm (MIRA)**, which has a very different framework. **MIRA** maximizes the minimum available bandwidth between every pair of nodes in the network by determining the crucial links for an edge pair and using the least crucial links to the remaining pairs while computing a flow's shortest path. However, **MIRA** requires previous knowledge of source-destination pairs. Additionally, it entails a lot of processing, making it ineffective, especially in large-scale networks. In this study's results, the authors concluded that the **DLC** and **DLCMI** algorithm families outperform static approaches, with the exception of **MIRA**, which showed high computation times.

In [50], the authors aim for an adaptable flow routing model in **SDN**-enabled networks. The first step taken was the definition of the routing metric (i.e. the way link costs are calculated) as a function of packet loss ratio, bandwidth, end-to-end delay, jitter and computational efficiency of the processing **FD**. According to the changes verified in these parameters, their weights in the cost metric are adjusted towards optimal network behaviour and **QoS**. Towards the comparison of their solution to traditional routing algorithms such as **Open Shortest Path First (OSPF)**⁸, the authors used Mininet and Iperf to generate networks and simulate traffic, respectively. The results obtained greatly surpassed traditional static weight approaches, especially in the packet loss, delay and jitter metrics.

Both [51] and [52] explore routing optimization approaches based on using links' available bandwidth as link costs and targeting network throughput maximization. The authors of [51] compare the static **Shortest Path First (SPF)** to **MIRA**, **WSP** and **Dynamic Online Routing Algorithm (DORA)**⁹. The study's results conclude that **DORA** performs the best, while **MIRA** is not adequate in the simulation conditions. Additionally, the results showed that choosing paths with the most available bandwidth improved performance in comparison to the traditional **SPF** algorithm. The contributions of [52] are the simulation of both unicast and multicast flows and the use of admission control when the cost of a path to get to the new node is above a predefined threshold.

⁸Which relies on Dijkstra.

⁹Similarly to the its alternatives, **DORA** aims to minimize network congestion by carefully attributing paths with given bandwidth constraints to flows, maintaining link utilization balance.

Traffic flows have fluctuating bit rates, which translates into traffic burstiness that negatively impacts these dynamic routing solutions. Traffic matrices, containing the flows' information, are also hard to model since the volume of traffic in a network may largely vary, making its prediction a complex procedure. Additionally, in multimedia services, it is common to have different users choose different services with different requirements, which forces the network to support flows with distinct characteristics [53]. Moreover, these solutions only consider the current network state, making greedy assumptions while failing to account for future network changes or needs. Thus, all of these dynamic routing solutions are based on simple ideas, however, given the complexity of computer networks, these optimization problems often become too difficult, increasing the computational load in the SDN controllers, or unsolvable, which leads to the exploration of other alternatives.

On the other hand, after training, ML algorithms can quickly calculate near-optimal routes without requiring an exact mathematical model of the network. Although different fields of ML can be explored to develop routing solutions, routing is based on deciding a path from a set of multiple possibilities, a decision that will then have an impact on an underlying environment that is fed back to influence the selection of new paths, a similar process to RL's framework [54].

2.5 Machine Learning

ML qualifies as a subcategory of artificial intelligence where an agent can gather knowledge from sample data, create a mathematical model describing the relationship between observed inputs and the respective results, and apply its experience in many different analysis techniques such as forecasting or classification [55].

The sample data that is fed into ML techniques is typically split into two, the *training set* and the *validation set*. Each element of these datasets is called an *example*¹⁰, which can be further discriminated into *labeled* or *unlabeled*, depending on whether or not the inputs, commonly referred to as *features*, are assigned to a *label* (i.e. the value we want to predict) [55][56].

A resulting model of the input data is produced considering the relationships between features and labels encountered in the training set in a process called *training*. After this step, the validation set is fed to the model in order to accurately assess its performance when submitted to new unseen data. In case this validation set is unlabeled, applying the trained model to it is called *inference* [56].

Given the input data, the resulting trained model can be called a *regressor* or a *classifier*. The first predicts continuous values (e.g. stock market shares), while the latter predicts discrete values (e.g. distinguishing images of dogs and cats) [56].

According to the characteristics of the training data, ML can be divided into [54][55][57]:

¹⁰A combination of feature(s) and, depending on the kind of example, a label.

1. **Supervised learning:** Built from labeled examples. From the training data associations between features and the corresponding label, the model learns how to solve tasks in a supervised manner.
2. **Unsupervised learning:** Unlike with supervised learning, in this case, the examples fed to the **ML** model are unlabeled, forcing it to find patterns or knowledge by organizing data in clusters according to their similarity and infer a way to predict the output of unseen values by associating them to the arranged groups.
3. **Semi-supervised learning:** As the name suggests, the model receives both labeled and unlabeled data. The present type of learning is common when the process of obtaining labels is somewhat "expensive", unlike features, which are abundant.
4. **Reinforcement learning:** Because it is one of the main topics of this project, this technique will be extensively described in a following subsection.

2.5.1 Deep Learning

DL is a subcategory of **ML** that is based on a set of algorithms, mostly organized in multiple cascaded processing layers performing linear and nonlinear transformations, that build models capable of representing high-level data abstractions by extracting features from data they are fed with [58][59].

DL, in spite of belonging to **ML**, stands out due to its characteristics. A **ML** model is fed real-world features as pre-processed data for it to be able to deal with them. Then, there is a feature extraction and processing layer, the hidden layer, where patterns are recognized and a final output layer that arranges data according to its goal of either classification, regression, or others. Contrarily, **DL** models usually have many hidden layers, which allows them to receive raw data as input, instead of pre-processed, and automatically find the most relevant features for its task [59].

Thus, **DL** is based on more complex networks and requires more data for its training process, but excuses a degree of human intervention that is required in typical **ML**. Despite their differences, both **ML** and **DL** rely on either **NNs** or **Deep Neural Networks (DNNs)**, if the **NNs** have a depth¹¹ of more than three.

2.5.2 Neural Networks

NNs emerged from the notion of replicating the biological neurons' learning process. A **NN**, or **Artificial Neural Network (ANN)** as it can also be called, is expressed as a dynamic and adaptive computational learning system that is instructed by using interconnected *nodes* distributed in *layers* of abstraction. Such architecture mimics a human brain's structure, the neurons' ability to work together to decipher human senses and how impulses are propagated [60][61].

¹¹The number of layers.

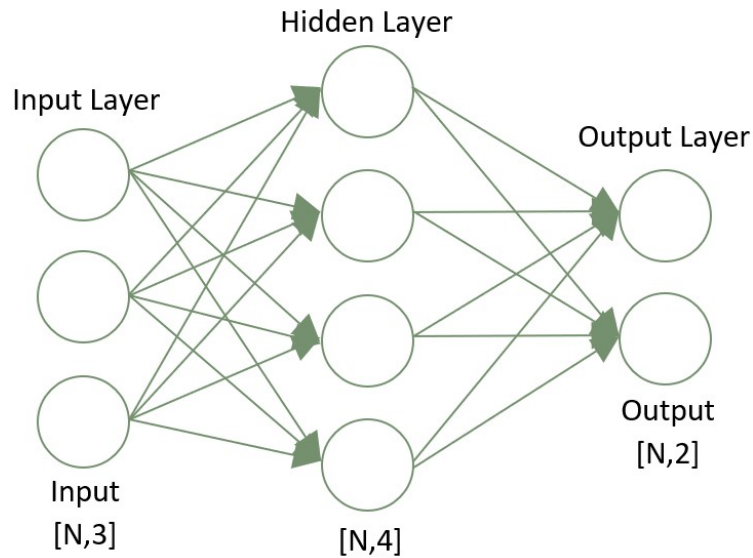


Figure 2.3: Simple NN with 3 layers (adapted from [62]).

In the case of the ANNs, the inputs are not human senses but, instead, datasets containing information retrieved from real-life situations which NNs translate into a desired and often different output [61]. Given this principle, NNs are being used in many situations where either image, speech or patterns recognition, data classification or forecasting is necessary, with applications in various fields like technology, finances, or medicine [61][63].

The different nodes in a NN's layer are connected by *links* and have associated *weights* and *thresholds*. Such weights are updated during the training of a NN for a specific goal, in order to make it able to perform a task with increasing accuracy. In a NN there are *input*, *hidden* and *output* layers. Data travels between these types of layers when a node's output is above the respective threshold (i.e. node's excitation) [60].

Thus, in a NN, each node is described by a linear regression model that combines inputs x_i , weights w_i , a threshold or *bias* and the output. An *activation function* $f(x)$ is defined and used between the NNs' layers to determine if whether or not data is passed forward in the network. This function represents the neuron's output by applying some kind of filter to the sum of the weighted inputs and the bias, as equation exemplifies [60].

$$\begin{cases} 1, & \text{if } \sum_{i=1}^m w_i x_i + bias \geq 0 \\ -1, & \text{if } \sum_{i=1}^m w_i x_i + bias < 0 \end{cases} \quad (2.13)$$

A popular activation function in DL models is **Rectified Linear Unit (ReLU)**, described by Figure 2.4.

Typically, a neuron's output is the input of one or many nodes in a following layer of the network and, to this general type of networks, the designation of **FFNNs** is given [60]. The weight a neuron assigns to each input determines how big of an influence it has on

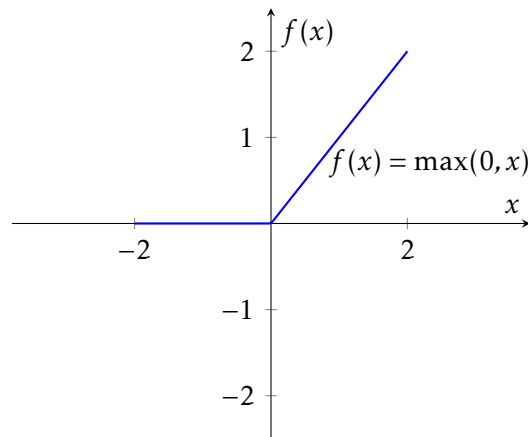


Figure 2.4: ReLU activation function (adapted from [64]).

the neurons' output. Therefore, NN's parameters (i.e. weights and biases) are regulated during the training of a NN model to maximize its accuracy by minimizing a given *loss function*, such as the **Mean Squared Error (MSE)**:

$$MSE = \frac{1}{2m} \sum_{i=1}^m (\hat{y} - y)^2, \quad (2.14)$$

where i is the sample's index, \hat{y} the predicted value, y the real value and m the total number of samples, through the *back-propagation* algorithm. Back-propagation deploys a nonlinear optimization process called *gradient-descent*, which differentiates the error function in relation to the network's weights and modifies them towards error minimization [60]. Its name comes from the direction of its influence, because once the outputs of the nodes are calculated, in a forward pass, then the derivative of the error for every parameter is calculated going backwards in the network. This is what a NN's training process looks like.

Several types of **DNNs** can be used in computer networks for diverse applications besides **FFNNs**. A few of them are introduced below and some were mentioned in Section 2.3.2 due to their use in traffic forecasting studies.

1. **Convolutional Neural Networks (CNNs)**: These networks are mainly used to process image data. Since computer networks' states are translated to traffic matrices, which can be viewed as one-dimensional images, this type of networks is popular for applications such as **TE** and traffic forecasting [41]. **CNNs'** layers use convolutional filters which consist of linear functions applied to the node's inputs in a sliding mechanism. Upon the sliding of this filter or weight matrix, the inner product between the input and the filter is calculated, allowing to train such filter for pattern recognition [65]. **CNNs** also have an important advantage over the more traditional fully connected **FFNNs** that is the generation of fewer parameters and

the consequent avoidance of overfitting¹².

2. **RNNs**: Both **FFNNs** and **CNNs** have difficulties while modelling certain time-dependant dynamic systems. In contrast, **RNNs** are able to persist information between steps thanks to the use of feedback loops that build an internal memory and help find correlations between events [31][66]. However, **RNNs** suffer from the vanishing gradient problem that is described by very small gradients incapable of updating the weights of the nodes and stopping the learning process of the **NN** or, at times, its explosion (i.e. large gradients).
3. **LSTMs**: These networks are a subtype of **RNNs** that excel at remembering information from distant past observations and can solve **RNNs**' complications. **LSTM** networks are built upon a chain of what are called *repeating modules*. A module comprises three gates: an *input gate* i_t , an *output gate* o_t and a *forget gate* f_t . An important concept of **LSTMs** is the *cell state* C_t , whose role is to propagate information, adding relevant or removing irrelevant pieces according to the gates' selection [67][68]. It represents the long-term memory of the model.
4. **GRUs**: **GRU** networks have a similar architecture to **LSTM** networks due to the existence of gated units controlling the flow of information in a neuron. However, **GRUs** do not use a memory unit and, instead, expose the hidden state without further control. Unlike **LSTMs**, **GRUs** only have two gates, a *reset gate* r , which merges the new input with existing memory, and an *update gate* z , responsible for defining the memory's importance in the new state. This reduction in the number of gates allows for fewer parameters.

2.5.3 Reinforcement Learning

RL comes from the approximation of the algorithm's learning process to the human behaviour, where knowledge is acquired by performing a certain action and observing its effect. Ultimately, the goal of a **RL** model is to understand which actions produce the best outcome, according to a predefined overall objective. This learning method is traditionally known as *trial-and-error*.

Formally, in a **RL** setup, the input data comes from a dynamic process, called *environment*, that is constantly generating relevant data for the model's objective. From this configuration, an *agent*, the **RL** algorithm, chooses an *action* a_t from the *action space* of the current *state* S_t , an environment snapshot at a discrete time step t [69]. In other words, given the state of the environment, the agent will choose what to do from a set of possible actions. Then, the action performed will impact the environment by changing its conditions, transforming it into a new state S_{t+1} and producing a *reward* r_t . The reward indicates how the action, taken at a given time step and environment state, influences

¹²Refers to a model that is too detailed and specific to its training set and does not perform as well when submitted to different validation sets. It basically means that the model memorized the dataset.

that is a strong motivation for the use of ML, specifically RL, over optimization-based techniques. On the other hand, *model-free* algorithms are usually easier to develop and train [70].

RL models can find the best course of actions that allow them to perform their tasks without prior knowledge of the environment. However, in its traditional form, these models have elevated convergence times and low convergence rates. Additionally, when faced with large-scale problems that involve vast state and action spaces, these challenges intensify [54][71]. To solve these issues, DRL has been developed.

2.5.4 Deep Reinforcement Learning

As the name suggests, DRL is the aggregation of RL with DL. This means that the agent entity is based on a DL algorithm, specifically, a DNN. Through DNNs, a DRL model adaptively adjusts its action selection policy based on the experiences it gradually gathers by observing the effect of past decisions on the underlying environment. The use of such learning agents improves the learning process of traditional RL, making these models faster, more versatile and scalable [71]. Once the DNNs are trained, they can receive a state and action space, estimate the long-term reward of each action, and choose the one that leads to the highest [54]. In the following subsections, value and policy functions will be presented, as well as popular categories of model-free DRL algorithms that rely on them.

2.5.4.1 Value and policy functions

A *policy* π is a function responsible for either mapping states to a probability distribution over the range of actions that the agent can choose from (i.e. probabilistic) or produce the action itself (i.e. deterministic) [69]. In the first case, the considered best actions have an higher associated probability, which leads to their more frequent selection by the agent. An important take on this idea is that a DRL model is based on *experience*, which means that the higher the number of different actions it chooses, the more knowledge it gathers and the better it can select an optimal action in the future. This points towards a common dilemma that is what strategy should the agent follow, either to choose the best action based on the knowledge it has gathered thus far (i.e. *exploitation*), or to explore other options to widen its experience (i.e. *exploration*). To balance these two concepts, DRL algorithms use an *epsilon-greedy* approach, which translates into a dynamic parameter that decreases during the training of an agent and determines when to choose an action based on the current agent's policy, or randomly.

Since the best actions are the ones that lead to the highest rewards, the policy that maximizes expected rewards is called the *optimal policy*. However, there are two ways for the agent to maximize expected rewards: by choosing the best actions in a given state or learning the most valuable states and take actions that lead to them, a concept that serves as introduction to value functions [69].

A *value function* $V_{\pi}(s)$ maps a state to its expected reward, which is the long-term rewards' average from being in a state or taking an action in that state, provided that a certain policy is consistently followed. A special type of value function is the *Q-function*, broadly applied to [DRL](#) and, specifically, to [Deep Q-Networks \(DQNs\)](#) [69].

2.5.4.2 Deep Q-Networks

A [DQN](#) is the combination of *Q-learning*, a [RL](#) algorithm based on the Q-function, with [DNNs](#). The Q-function is an *action-value* function $Q_{\pi}(s, a)$, to which the input is not only a state, but a state-action pair that returns the value of taking the input action considering the input state. These values are called *Q-values* and are stored in the *Q-table*. When Q-learning and [DL](#) are brought together, the Q-table takes the form of a [DNN](#) [69].

In Q-learning, to optimize the prediction function, the *expected return* is calculated by multiplying each reward by a reward coefficient that measures its importance, the *discount* factor, and by comparing the result to the actual accumulated rewards observed. Upon this comparison, the function is updated according to:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R(S_t, A_t) + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)], \quad (2.15)$$

where the updated Q-value is equal to the current Q-value plus the [Temporal Difference \(TD\)](#) that separates the predicted future Q-value, calculated by summing $R(S_t, A_t)$ and $\gamma \max_{a'} Q(S_{t+1}, a')$ (i.e. the reward and the maximum Q-value of all the possible actions), and its current value $Q(S_t, A_t)$. In this equation, γ is the discount factor and α is the learning rate¹³. In other words, the update of the Q-value equals it to the current predicted Q-value plus the amount of value expected in the future [69].

From the analysis of the Q-function update rule, it can be stated that Q-values obtained from the Q-function will not be constant, since they greatly depend on the rewards observed. This is due to rewards being commonly unstable, divergent (i.e. a given state-action pair might result in different outcomes on separate observations) or even sparse in some applications, which translates into constant function and policy updates. There are two main techniques that can be used to solve or minimize this problem [71]:

1. *Experience replay*: This mechanism allows for batch updating in online settings using a memory system. The algorithm initializes the *replay buffer* which will continuously store experiences (s_t, a_t, r_t, s_{t+1}) (i.e. transitions described by a state, an action taken in that state, the reward obtained and the resulting state). Then, at each Q-function update, a few experiences will be sampled from this buffer and form the mini-batch used as input to the Q-learning algorithm. From that point on, actions will be chosen according to a renewed policy, generating new experiences for the

¹³Used to train [ML](#) algorithms.

replay buffer. This means that, upon the next update, since the sampling of experiences is performed randomly, old and new experiences should get sampled together, stabilizing the algorithm. When the buffer is full, the oldest transitions are deleted.

2. *Target network*: A copy of the main DQN \hat{Q} , that has its parameters updated with a certain lag. This will help stabilizing the back-propagation training process since the target network will hold off on updating its parameters. Additionally, by using this network to calculate the target Q-value to train the main DQN, the effect of recent updates is decreased.

With the addition of these techniques, the DQN algorithm can be summarized by Algorithm 1 [72].

Algorithm 1 DQN with experience replay and a target network

- 1: Initialize the replay buffer **RB**.
- 2: Initialize the Q-network Q with weights θ .
- 3: Initialize the target Q-network \hat{Q} with weights θ' .
- 4:
- 5: **for** episode=1 to T **do**
- 6: Choose action a_t at random or $a_t = \operatorname{argmax} Q^*(s_t, a_t, \theta)$ (i.e. a function that takes an array, finds the largest value in it, and returns its index) considering the exploration parameter.
- 7: Get the resulting state s_{t+1} and reward r_t by performing step(a_t).
- 8: Store transition (s_t, a_t, r_t, s_{t+1}) in **RB**.
- 9: Sample a batch of random transitions from **RB** and use them to train the DNN through gradient-descent and the following update rule:

$$[r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta') - Q(s_t, a_t; \theta)]^2. \quad (2.16)$$

- 10: Make $\hat{Q} = Q$ every N steps.
 - 11: **end for**
-

It is also important to point out that Q-learning is an *off-policy* algorithm in which, in contrast to an *on-policy* algorithm, the policy choice does not affect the process of learning Q-values. Even if the actions are always chosen at random, with sufficient amounts of data (i.e. experiences), the algorithm will learn accurate Q-values. However, this methodology is inefficient and that is why the epsilon-greedy policy is commonly used. On the other hand, on-policy algorithms learn policies while using them to collect experiences, making their learning process dependent on the policy they are using [69].

2.5.4.3 Policy gradient methods

Policy gradient methods are on-policy algorithms that dismiss DQN's use of a policy to select actions besides predicting action values. Alternatively, these methods train a DNN to learn a policy itself.

In policy gradient methods, the resulting network intends to output a probability distribution over the set of available actions [69]. Thus, these methods aim to optimize a performance target, usually the expected cumulative reward, by repeatedly adjusting their policy.

A common policy gradient method is the **Stochastic Policy Gradient (SGD)**. It outputs a probability distribution over a set of actions, assigning higher probabilities to the ones that carry higher Q-values, allowing for a healthy balance between exploration and exploitation. On the other hand, there is another popular method called **Deterministic Policy Gradient (DPG)** that outputs only the action the agent must perform.

2.5.4.4 Actor-critic methods

Given that **DQN-based DRL** is only viable for control problems with low-dimensional and discrete action spaces, it cannot be used for continuous control situations where it is required to find the reward maximizing action at each training epoch [70].

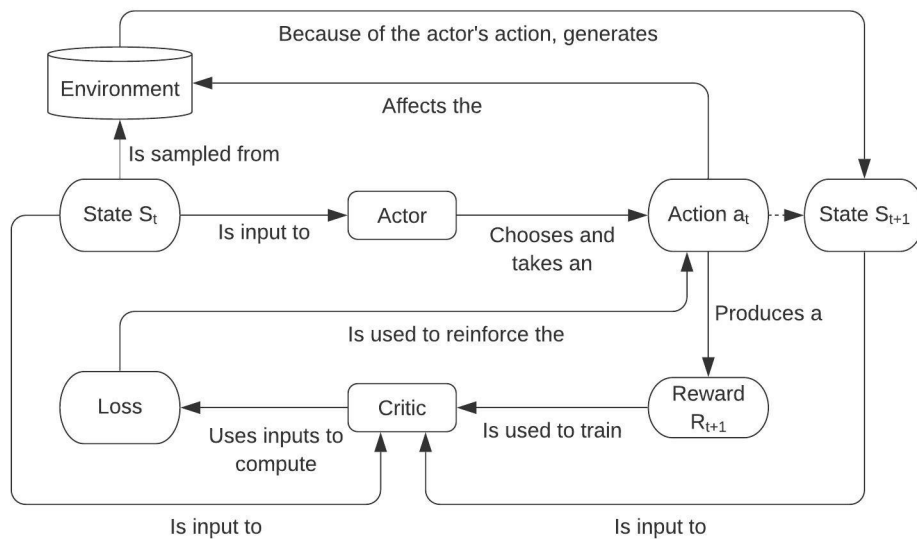


Figure 2.6: Actor-critic methods' generalized behaviour (adapted from [69]).

In *actor-critic* algorithms this problem is tackled by combining features from both value (i.e. Q-value) and policy functions to leverage from both methods' advantages. As the name suggests, actor-critic models are split into two parts, an actor network and a critic network. The actor corresponds to the policy function deciding an action to take, while the critic reports on the adequacy of the action taken by the actor in a given state. Through this combination, the value function can stabilize rewards which are then used to train the policy [69].

2.5.4.5 Deep Reinforcement Learning for Multimedia Traffic Transport

In this section, a few model-free approaches ([53][70][73][74][75]) to routing optimization of multimedia traffic in a SDN environment through DRL will be detailed and compared, in order to discuss the most important aspects in the state of the art of approaches similar to our proposal.

In [53], because it is directly influenced by multiple other individual metrics and thanks to its direct relation to the customer's satisfaction, QoE was considered the most relevant metric and the solution was developed towards its optimization (i.e. cumulative QoE maximization). In summary, this solution's contributions are: the capability to learn how to allocate a clear path with bandwidth guarantee for each flow, QoE mapping using a DNN for accurate reward measuring and a Deep Deterministic Policy Gradient (DDPG) implementation, an actor-critic method that thrives in solving continuous control¹⁴ problems over, for example, DQN, and improves the convergence time of traditional actor-critic implementations.

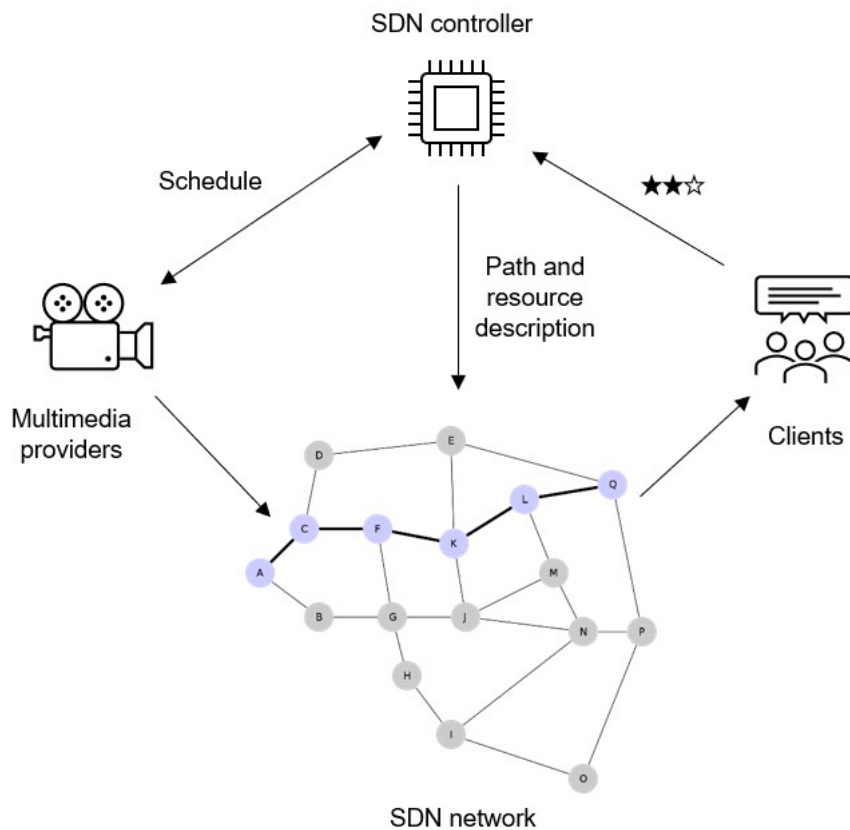


Figure 2.7: Architecture of multimedia traffic control in SDN (adapted from [53]).

Figure 2.7 exhibits the key components of multimedia traffic control in SDN as proposed in [53]. In this architecture, the SDN controller, given its global view, is responsible

¹⁴In RL, continuous control refers to the use of continuous values as the output action.

for managing the underlying network, collecting data, defining paths and allocating the required resources for them. The *SDN-enabled network* is a set of **FDs** at the controller's disposal. The *clients* consume the services delivered by the network and provide feedback of their experience. *Multimedia providers* are the source of the multimedia content.

Typically, clients request services which are delivered to them in the form of flows. The controller finds paths for each flow crossing the network leveraging its global network view for the paths' optimization. Then, it transmits synchronization information to the network (path and bandwidth) so that the **FDs** can build forwarding tables and process the upcoming information from multimedia providers. While consuming their service requests, clients can report their satisfaction to the controller, which is used to dynamically adapt resources.

The **DRL** application to this problem was made under the interpretation of a **MDP**, with its elements described as:

1. **Environment:** The combination of the **SDN-enabled network**, the clients and the multimedia providers.
2. **Agent:** Encapsulated in the **SDN** controller. It will interact with the environment, observe the effect of its decisions, and learn how to behave through experience, determining the optimal traffic control policy.
3. **State:** A snapshot of the environment, in this case, the state of the multimedia flows, containing bandwidth, delay, jitter and packet loss information (i.e. the traffic matrix).
4. **Action:** In this panorama, an action consists of the path reported by the controller and the bandwidth changes that the network is required to perform on other flows. Actions should respect network constraints and adjust to the clients' feedback.
5. **Reward:** Upon ordaining an action in a given network state, the agent receives from the environment, specifically, the clients, a reward. In this study, the reward is measured through **QoE**, which is evaluated using a **Mean Opinion Score (MOS)**. Since real time retrieval of this information is difficult due to the lack of constant interaction with the customers, the authors chose to use a multi-layer **DNN** in the flows' statistics - **MOS** translation.

To build the agent according to a **DDPG** method, the authors used a few previously covered techniques such as a replay buffer and target networks, for both the actor and critic, to improve the method's stability. In this particular case, the actor of the primary network receives as input a state and outputs an action, due to its deterministic property, hence exploring the current policy. On the other hand, the critic receives the state and the action choice of the actor and calculates the pair's value. This value is then fed into primary actor for training (i.e. gradient-descent) and used in the loss function with the

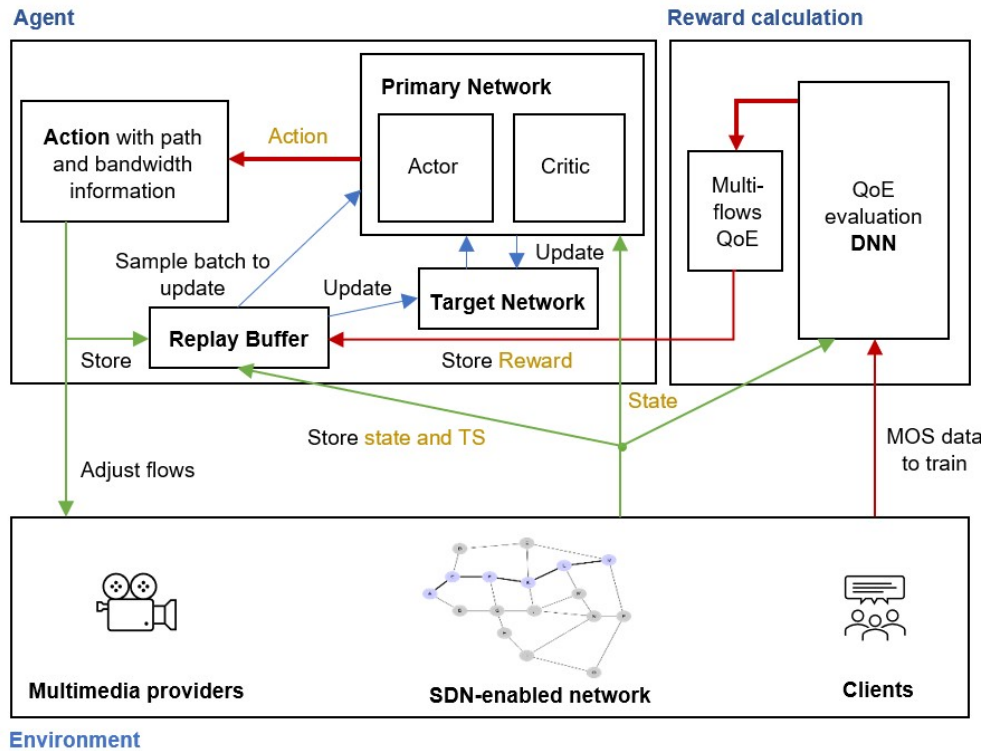


Figure 2.8: Interaction between system components. Red lines represent outputs and green lines inputs (adapted from [53]).

target Q-value calculated by the target critic network and the transition's reward, hence updating the critic network's weights.

The solution developed in this study has reached good results at maximizing the mean QoE when compared to other selected optimization-based approaches. These were obtained in multiple topologies of different dimensions and the best results were achieved when the network size increased, which supports the solution's scalability. In comparison to our problem, this work's motivation is a theoretical routing scenario where the objective is not only to select a path but also to assign to it a bandwidth value. Thus, this solution is not applicable to our scenario since, in our case, the bandwidth of flows is user-defined and cannot be modified. Furthermore, since this study details a simulated problem, the authors were able to consider the existence of clients and receive feedback from them, however not specifying how. On the other hand, such feedback is impossible to retrieve in our framework. The sparsity of that feedback was also considered, however, to overcome it, the study uses metrics that are also not accessible in our case to obtain a MOS score. In conclusion, although this work allows us to thoroughly understand how DRL can be applied to multimedia routing and to prove its efficiency and scalability, our solution must be different in some respects.

Yu *et al.* ([73]) also presented a DDPG-based solution called **DDPG Routing Optimization Mechanism (DROM)**. This solution's model is very similar to the one suggested

in [53]. However, it has the additional functionality of customizing the network performance parameters considered from only delay to a function of multiple metrics and changes the action entity to an array of link weights. In this work, a direct comparison to an implementation of a static weight Dijkstra’s algorithm-based solution is presented. Accordingly, **DROM** showed significantly lower delay when selecting the delay as the performance metric to optimize. In another test, throughput maximization was defined as the objective function. When compared to two selected throughput maximization solutions, either achieved by link cost optimization and congestion preventive scheduling or **QoS**-aware reward functions that implement dynamic routing, **DROM** outperformed both. Such results were obtained through OMNET++ and using multiple traffic load scenarios. Since **DROM** was able to both reduce delay and improve throughput in all traffic load configurations, this solution also confirms **DRL**’s scalability. Contrarily to our problem’s motivation, this work is motivated by a theoretical scenario. Moreover, this approach computes the link weights that Dijkstra is supposed to consider when calculating the shortest paths. To assign updated weights to routers’ interfaces in DataMiner is a computationally heavy process, thus our solution must be able to dismiss the use of Dijkstra. For these reasons, this solution is also not entirely applicable to our problem.

Xu *et al.* ([70]) proposed another alternative to **DDPG**-based solutions called **DRL-TE**, a **DRL** approach for **TE** problems. **DRL-TE**, as opposed to **DDPG**, promotes **TE**-aware exploration and an actor-critic-based algorithm to optimize the agent’s behaviour, reducing the measured end-to-end delay and increasing the network utility as well as maintaining or increasing the throughput. The authors also claim that **DDPG** is not an effective solution, since its framework does not specify how to explore the environment and because it uses a uniform experiences’ sampling, ignoring their importance. In this work, ns-3¹⁵ is used to simulate and prove that the solution proposed is more effective.

The **TE** problem on the basis of this work is postulated by the authors as a set of communication sessions, described by a source, a destination and a group of candidate paths to route traffic between those endpoints that need to be managed, with the objective of minimizing end-to-end delay and maximizing the network utility. For this purpose, the state and action spaces, as well as the reward function, are modelled as follows:

1. **State space:** A set of throughput and delay tuples from each communication session.
2. **Action space:** The set of split ratios for all sessions (i.e. the probability of routing traffic load across each of the candidate paths for all existing communication sessions).
3. **Reward function:** The total utility considering all communication sessions, calculated by the sum of each utility value measured in reference to the end-to-end delay and throughput.

¹⁵“ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use.” [76]

A relevant contribution introduced in [70] was the use of a baseline TE solution, like SPF or other link cost optimization algorithms such as those introduced in Section 2.4.1, for exploration while training the DRL agent. DDPG adds random noise to the action selected by the actor network, contrarily, DRL-TE decides the next action between a_{base} , proposed by the baseline TE algorithm and a , the output of the actor network, with probabilities ε , an adjustable parameter that balances exploration and exploitation (i.e. the epsilon-greedy approach), and $(1 - \varepsilon)$, respectively. The more epochs the agent is trained for, the lower ε gets, promoting the choice of optimal actions. Furthermore, DRL-TE uses Prioritized Experience Replay (PER)¹⁶ to improve performance.

DRL-TE was compared to other known technologies, such as SPF and DDPG, using the end-to-end delay, total network utility and throughput as performance metrics, and in two common network topologies, National Science Foundation Network (NSFNET) and Advanced Research Projects Agency Network (ARPANET), as well as a randomly generated topology. From the results, it can be understood that this approach reduces the verified end-to-end delay and improves the network utility while, at least, offering comparable throughput in multiple distinct traffic scenarios. In comparison to the baseline DDPG, DRL-TE showed significant improvements.

Once again, this solution applies to a theoretical problem. In this environment, the real-time end-to-end delay is obtainable and used in the state formulation. However, in our scenario, this metric is not considerable.

Sun *et al.* ([75]) introduced ScaleDRL, a DRL-based TE solution for large networks. So that the scalability of DRL algorithms for TE can be strengthened given the exponential dimension increase of state and action spaces as network's size grows, the authors use pinning control theory to select critical links (i.e. overloaded links). Then, the weights assigned to those links shall be adjusted by the DRL agent. This is an attempt to improve the convergence rate of the algorithm, since it will only be controlling a few selected links, thus reducing the action and state spaces. Once the weights are regulated, the network paths are generated by a weighted SP algorithm.

Two parts of this solution can be highlighted, the link selection algorithm and the DRL algorithm. Both reside inside the SDN controller that collects network states and adapts TE policies. Hence, this solution works online and offline. During the offline mode, the first aforementioned algorithm computes the critical links. Then, in online settings, the DRL algorithm adjusts those link's weights, thus optimizing the paths that are loaded onto the network switches.

The link selection algorithm signals critical links by measuring their centrality. This value represents how many times a link is utilized by the paths connecting endpoints of flows. Once the centrality of all the links is calculated, these values are ordered, and the middle k are marked as critical.

¹⁶A topic covered in Section 3.4.2.

As for the [DRL](#) algorithm, it is based on a typical actor-critic architecture. A particularity of this work's approach is the use of [GRUs](#) to model traffic considering its temporal characteristics, preceding a two-layer [FFNN](#) in both the actor and critic's structure. As for the algorithms' elements, the state consists of the normalized links' traffic distribution, the action is a description of critical links and the number of candidate paths for each flow and the reward is calculated by weighing the maximum link utilization ratio and the average end-to-end delay of all flows in the network.

The authors ran simulations mostly regarding the average end-to-end delay. ScaleDRL was compared to both [SPF](#) and DRL-TE and resulted in the lowest levels of delay amongst the three across variable network sizes. In spite of the computational load that this approach could save by applying the [DRL](#) algorithm to only a few selected links, it still uses Dijkstra as the final step of the solution. Additionally, this approach considers the average end-to-end latency, which is not at reach in our case.

Previous approaches all have in common the formulation of the problem in a continuous control structure, since they either use link weight arrays as their action ([\[73\]](#)[\[50\]](#)), a path bandwidth continuous value ([\[53\]](#)) or an array of path choice probabilities ([\[70\]](#)). On the other hand, Chen *et al.* ([\[74\]](#)) propose a solution, called RL-Routing, which uses a discrete action space and targets throughput maximization and communication delay minimization. RL-Routing is formulated as follows:

1. **State space:** An array of 10 elements, such as the link capacity rate, link throughput rate, link delay, binary link status (i.e. 1 if up, 0 otherwise), link trust level (i.e. based on packet loss), switch throughput rates, link-to-switch rate (i.e. maps a link's load percentage on a switch), the day of the week and the part of day (i.e. time intervals of 6 hours).
2. **Action space:** The agent can choose, for a given communication, from a set of pre-calculated k-shortest paths between each source-destination pair.
3. **Reward function:** A sum of the chosen action's delay and throughput rate.

To train this algorithm, the authors used a multi-agent (i.e. an agent per switch) [Dueling Double Deep Q-Network \(Dueling DDQN\)](#).

For the development and testing environment, a combination of a Ryu controller to manage the network, Mininet for network simulation and Iperf for traffic injection was deployed. For topologies, the authors selected [NSFNET](#) and [ARPANET](#). To assess how the developed solution compares to baseline solutions like [OSPF](#), the reward sum, a file transmission's completion time and utilization rate (i.e. the link bandwidth ratio used for the file transfer) were considered. In either of the simulated topologies, RL-Routing showed higher rewards and shorter transmission times while maximizing the utilization rate.

As mentioned before, unlike the theoretical approaches presented, our solution must be developed for DataMiner. This real-world nature of our problem increments the importance of optimizing the proposed solution's computational effectiveness. This study uses a complex state to characterize the environment, which contributes to the increase of its CPU usage. Additionally, as was the case of previous approaches, this study also has access to values that, in our scenario, cannot be considered. Thus, although many important aspects can be extracted from this solution, some adaptations must be done, such as the simplification of the state space and change of considered metrics.

SYSTEM DESIGN

As previously mentioned, this dissertation's aim is to bring innovation into the scope of Skyline Communications' DataMiner, specifically the **SRM** module, by optimizing its routing mechanism. Various possibilities on how to complete this goal have been introduced and were on the basis of this project's solution design. This chapter provides a brief overview of **SRM** (only the concepts related to this dissertation's work) to contextualize the environment where the developed mechanism is to be applied and how it can be implemented. It then describes the components of the deployed simulation's architecture, developed using the *Python Ryu SDN controller* connected to the *Mininet* network simulator via its Python **API**. Finally, the specific **DRL** algorithms used or considered when solving the proposed problem will be presented as well as the solution design.

3.1 Service Resource Manager architecture

DataMiner monitors and controls every device inside a customer's ecosystem by means of **APIs**. **SRM** takes this idea one step forward by allowing this management process to be scheduled for some time in the future. This is accomplished by dynamically combining devices' functions, instead of entire products, to respond to a given service request according to their capabilities and capacities.

SRM is divided into multiple building blocks that together can ensure that multimedia services' requirements are met and monitored throughout their duration. The management of these blocks is performed, on a lower level, by dedicated modules, which are then bundled up and coordinated in the *SRM Booking Manager* application.

As the name suggests, the **SRM** Booking Manager is where *bookings* are submitted. A booking is a scheduled resource reservation for a certain multimedia service. The booking's duration comprises three distinct time periods: *pre-roll*, *active* and *post-roll*. During the pre-roll, *resources*, such as **SDI** encoders, **SDI** decoders, switches, routers, or antennas, are configured for the specific type of booking requested. The required devices' functions for a booking are specified in the *service definition*, while the configurable parameters of those functions are stored in *profiles* [77].

In the case of this dissertation’s development, the target service definition includes: a **SDI encoder**, where the multimedia source is originated and translated to **IP**, a **transport network**, a nested service definition that routes data according to Dijkstra’s algorithm, and a **SDI decoder** connected to the exit edge node of the transport network.

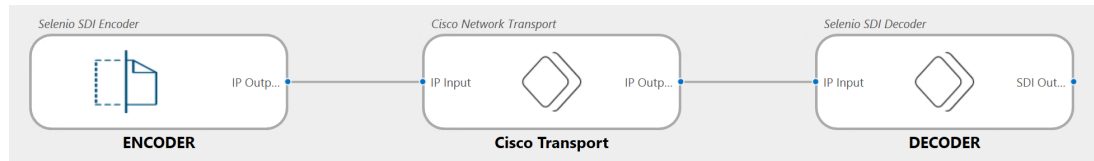


Figure 3.1: Transport-Cisco service definition (from DataMiner).

A booking reaches its end when the active time period has elapsed. At that moment, the reserved resources are released and deconfigured during the post-roll.

3.1.1 Concepts

In **SRM**, objects are identified by a unique string called **Globally Unique Identifier (GUID)**. These objects can be of multiple types and managed by different modules. The term *Resource* is used to refer to any object managed by the *ResourceManager* module, such as a physical device (e.g. Cisco Nexus router), or device components (e.g. a router’s Ethernet port). When these resources are representations of devices (i.e. when they have a specific function) they are called *FunctionResources*. Resources of the same category are arranged in a collection called *ResourcePool* [77].

In this dissertation’s use case, a relevant type of resource is a Cisco Nexus router’s interface. Physical interfaces are represented in DataMiner as **DataMiner Connectivity Framework (DCF) interfaces** that are linked through *connections*. The **DCF** manages DataMiner elements’ connectivity through these two main concepts. Virtual interfaces also exist and represent internal connections of an element. **DCF** also allows these interfaces to have *properties*. In the case of the Cisco Nexus routers, each port is split into receiver (Rx) and transceiver (Tx) interfaces described by status, concurrency, bitrate capacity, Id, connectivity to other router interfaces, or **cost**¹ properties.

During a multimedia flow transmission, resources are used to perform certain actions. A description of which resources are in use, how much of their capacity is occupied and when that use occurs is a *ReservationInstance*. This definition enables the Resource-Manager to control the resources’ availability at any point in time and provision their capacities accordingly. *ReservationInstances* create *Services* while they are active, called *ServiceReservationInstances*, that typically use a series of connected *FunctionResources* [77].

Services are instances of *ServiceDefinitions* and managed by the *ServiceManager*. A *ServiceDefinition* connects functions to provide a customer with a service description of

¹The cost of a link between two interfaces is equal to their cost sum. Then, those link costs, or weights, are used by the Dijkstra’s algorithm to find the shortest path to route a certain multimedia flow.

a given type. These functions correspond to *FunctionDefinitions* that, when instantiated, form *FunctionInstances*. The *FunctionDefinitions* inside a *ServiceDefinition* are connected by means of their interfaces' profiles [77].

Bundling all these terms into the SRM Booking Manager plane, a user places a booking that belongs to a chosen *ServiceDefinition*. Additionally, the user selects the resources required for that *ServiceDefinition* according to the *FunctionDefinitions* that it may include. This action creates a *ReservationInstance* that indicates the chosen resources' capacity usage and utilization time and duration. When that *ReservationInstance* becomes active a service is generated, starting the data transmission, typically in the form of a video feed, according to the *ServiceDefinition* that it instantiates. This service assures the allocation of the resources contained in the underlying *ReservationInstance*.

3.1.2 Resource monitoring

DataMiner interacts with the resources it monitors and/or orchestrates through multiple types of protocols such as HTTP, SNMP and Serial protocols or dedicated APIs. In this specific scenario, since this dissertation was proposed to fix a routing problem, the important values to access are the routers' metrics, which DataMiner retrieves using SNMP. Thus, this protocol in particular will be in focus.

SNMP is a broadly recognized and used application layer protocol for monitoring IP networks' devices, such as switches, routers, or servers, and managing them by configuring changes to their behaviour according to the information gathered [78].

The SNMP architectural model includes [77][78]:

1. *Managed nodes*: The network elements, which run a software component, the *Agent*, that provides accessibility to the device.
2. *Network Management Stations (NMSs)*: These SNMP entities, also known as *Managers*, support both unidirectional and bidirectional requests traded with the managed nodes. In the scope of DataMiner, it corresponds to the DataMiner Agents (*DMAs*). A DataMiner System (*DMS*) can contain from one to multiple *DMAs*.
3. *Management protocol*: Which transports information between SNMP entities (the network elements and the management stations).

Thus, *NMSs* communicate with the network elements using the SNMP protocol in order to inspect or alter their exposed objects by the *Management Information Base (MIB)*, functionalities that are provided by the SNMP agent. The *MIB* describes the information an SNMP Agent exposes in the form of objects or variables. This information is stored in an hierarchical structure that uses *Object Identifiers (OIDs)* to refer to each variable in the tree [78].

The most recent version of [SNMP](#) is *SNMPv3*, which built on the existent modular architecture of previous versions in order to define new capabilities, improving its performance, flexibility and security. DataMiner supports every [SNMP](#) version [77][78].

The system provided by Skyline to operate and manipulate during this dissertation was a local installation of DataMiner with interface's metrics' values fed by *SNMPWALKS* converted to [SNMP](#) simulations. A *SNMPWALK* is a [SNMP](#) application that queries devices for their status information. These requests specify the variables to be retrieved through their [OIDs](#). In this simulation, the returned values will vary while being confined to a predefined range. For that reason, these walks provide a given dynamic to the system. However, this simulation is not reactive to the changes that will be made (i.e. assigning a path to a booking will not make the bandwidth used by the routers in that path increase and, consequently, make the cost of the links that connect them follow the same trend). This issue was surpassed with the development of the external [SDN](#) environment.

The interfaces of each router have their metrics detailed in the form of a table, called *ifTable*. In some cases, when a metric's value is frequently changing and it can be beneficial to predict a future value, trending templates are used. DataMiner uses its *Trending* module for this purpose. Moreover, when a monitored device is queried and returns an abnormal value for a certain property, DataMiner can raise an *alarm* that informs the user [77].

After a path for a booking is selected, it needs to be sent to the network devices so that they know where to route the packets they receive. To accomplish this, the process depends on the specific vendor of the device in question. However, in most Cisco devices, which were the ones used in this dissertation, first communication with the device is established using [CLI](#), then [SNMP](#) configurations are declared. Lastly, it is expected that the device builds the routing tables according to the information provided.

3.1.3 Protocols & Templates

A protocol, or *connector*, is mostly developed using [DataMiner Protocol Markup Language \(DPML\)](#), Skyline's proprietary markup language, similar to [XML](#). A *DMS* stores protocol versions that can be used to create *elements* that follow the behaviour described in them [77].

An element is a virtual entity that can refer to both physical devices and software processes. Considering the example of a Cisco Nexus router, it is represented in DataMiner as an element. This type of element exposes many resources, which are the router's interfaces. The router element behaves according to its running protocol which, amongst other things, defines how DataMiner and the physical device communicate considering its supported connectivity [77].

A protocol with a [SNMP](#) connection, as illustrated in Figure 3.2 (i.e. elements A and B), describes the logic needed to poll a device through [SNMP](#) GET requests, change

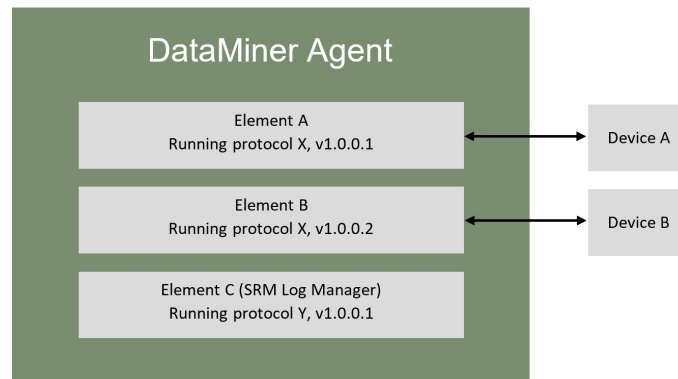


Figure 3.2: Overview of a DMA (adapted from [77]).

parameter values with [SNMP SET](#) messages and receive unplanned [SNMP traps](#)². These capabilities are granted by the *SLSNMPManager* process running on the DMA. Element C illustrates an example of an element that is not applied to a physical device but instead to a software process that coordinates a feature (e.g. logging) [77].

A protocol's logic is executed by *SLProtocol* processes. This logic is based on the interaction of a few components [77]:

1. **Parameters:** A protocol parameter can be used to represent either a data table, a table column, an internal logic value such as a counter, or a [UI](#) component (e.g. a button).
2. **Groups:** They allow the gathering of equal type items, such as parameters, triggers, or actions.
3. **Timers:** Responsible for defining the periodic groups' execution. At the timer's end (e.g. every 10 seconds), the group it controls is added to the group execution stack. The execution time of the group is then defined by the contents of that stack.
4. **Triggers:** Can, for instance, execute an action.
5. **Actions:** Define something to do, such as copy values between parameters or increment a parameter value. Can be activated by triggers or be part of a group.
6. **Quick actions:** Also called *QActions*, they are C# code blocks capable of implementing complex logic that is executed by the *SLScripting* process.

An example of these components' interaction can be reviewed in [Figure 3.3](#).

A protocol can have a multi-threaded behaviour, allowing for simple actions to execute on a parallel thread to the main protocol execution thread while it is busy performing a complex *QAction* [77].

²Alert messages sent from a devices' agent to a network management station.

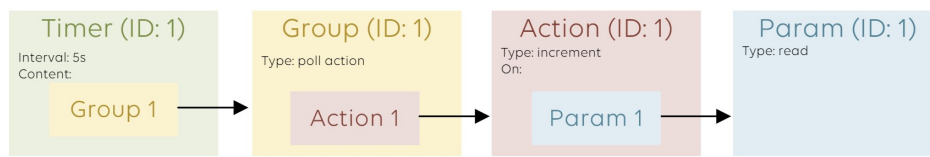


Figure 3.3: Execution of a group triggered by a timer (adapted from [77]).

3.1.4 Automation

In the *Automation* module, all automation scripts are saved. An automation script can either perform a simple change on a resource's property or be responsible for controlling complex sequences of instructions. The *SRM* Booking Manager triggers multiple automation scripts that simplify given processes [77].

These scripts can be developed using the built-in tools that create, for instance, if-else blocks. However, when meant to handle complex tasks, they are written in C# code. Automation scripts run using an underlying process called *SLAutomation*. This process performs method calls answered by a larger scale process (*SLNet*), which controls all communication between *DMAs* and a *DMS* and its clients. This process answers to many different other processes, as is the case of the protocols' *QActions* process (*SLScripting*). However, protocols use the *SLProtocol* object to call those methods and automation scripts use the *Engine* object [77].

3.2 Ryu Controller

Ryu is an open-source controller for *SDN* developed in Python. It supports multiple network devices' management protocols such as OpenFlow. It also provides a well-defined and documented *API* for the development of custom network control applications [79].

A Ryu application extends the `ryu.base.app_manager.RyuApp` Python class. Its behaviour is based on observing and generating *events*, generally triggered by OpenFlow messages. These events are associated to methods through `set_ev_cls` decorators that specify which method is the event handler. These decorators take two arguments, the event type and the dispatchers', such as the `MAIN_DISPATCHER`, when a switch is connected to the Ryu controller, the `DEAD_DISPATCHER` when disconnecting or the `CONFIG_DISPATCHER` when the OpenFlow version is agreed on. These dispatchers specify the negotiation phases in which the handler will be triggered [79].

Since the events represent OpenFlow messages received from connected switches, they are referenced through the `ofp_event` class. Each decorated method will have an event as input. From this, the OpenFlow message object `msg` can be accessed and from there the `datapath` object is available. A `datapath` instance describes the switch that sent the event that triggered the OpenFlow message. Besides identifying the source of the OpenFlow message, from the `datapath` object other attributes can be retrieved: the `ofproto`, which exports OpenFlow definitions and the `ofproto_parser`, a message encoder and decoder for

the agreed OpenFlow version. It is with this parser that flow arguments, such as the match, actions, or instructions, introduced in Section 2.1.3, are created. Then, these are usually combined in a `OFPPacketOut` message that is sent to the respective switch using the `send_msg` method and according to the datapath.

Multiple events can be received almost simultaneously by the Ryu controller, which makes it impossible, in some cases, for an event to be fully processed before a new one arrives. To solve this issue, Ryu uses a **First-In First-Out (FIFO)** queue system that maintains the order of the events received while they cannot be processed [79].

3.3 Mininet

Mininet is a virtual emulator of physical networks that creates devices such as switches, routers, or hosts, in a development environment for software-defined networks. These components behave almost always exactly like the real hardware devices. For that reason, it is possible to recreate a real network using Mininet, with switches connected through Ethernet interfaces with a given capacity and delay. To measure the communication performance between a pair of hosts, a tool such as *Iperf*³ can be used. Mininet also provides a Python API [81].

3.4 Deep Q-Network enhancements

In Section 2.5.4.2, **DQNs** were introduced as an algorithm capable of combining the best features of both **DL** and **RL** to optimize a Q-function in discrete action settings. While **DQNs** are perfectly capable of answering certain problem's requirements, there are situations where a **DQN** is not sufficient. Thus, a few improvements can be added to the traditional **DQN** algorithm.

3.4.1 Double Deep Q-Networks

A **DQN** estimates the Q-value of every state-action pair available. However, a single Q-network is known for its tendency to over-estimate the value of actions and, therefore, damage the algorithm's performance [72][82]. With a typical **DQN**, the Q-value is calculated using the sum of the reward and the maximum Q-value, which introduces a positive bias caused by using the maximum action value as an approximation of the maximum expected action value (equation 2.15). This occurs because the same samples are used to select and evaluate an action [72]. Thus, the over-estimation of some actions' value will neglect the possibility of other actions being more suitable in certain conditions [82].

Double Deep Q-Networks (DDQNs) implement *Double Deep Q-learning* by using two value functions, initialized as a copy of each other, to separately select and evaluate

³Iperf measures the maximum achievable bandwidth on **IP** networks, reporting on **QoS** parameters such as the bandwidth, loss, jitter or end-to-end round-trip-time [80].

action values. This approach leverages from the target network technique by using it as the second value function [73]. The difference between **DDQNs** and traditional **DQNs** that use target networks is the way the update function is built. In this case, it is not the target network selecting the best action to calculate the target Q-value, but rather the primary network. Thus, this action selected by the main network through *argmax* will then be used to estimate the value of the current Q-function using the target network's \hat{Q} weights θ' . This process is described by equation 3.1 [73].

In this format, the weights of the second Q-network are updated according to the weights stored in the target network \hat{Q} , while the target network's weights θ' are updated periodically according to the online weights θ as is done in traditional **DQNs**. This process is described by the equation 3.1 [72].

$$[r_j + \gamma \hat{Q}(s_{t+1}, \operatorname{argmax}_{a'} Q(s_{t+1}, a'; \theta); \theta') - Q(s_t, a_t; \theta)]^2 \quad (3.1)$$

3.4.2 Deep Q-Networks with Prioritized Experience Replay

Experience replay has been suggested as a performance improving factor to **DQNs**. However, samples are extracted at random from this data structure and not carefully selected. To take into consideration that some experience transitions hold higher value to the algorithm's learning process than others (e.g. transitions with high rewards), **PER** was developed. **PER** allows for a more frequent sampling of priority transitions, characterized by their recent addition to the replay buffer or low error verified in past transitions. This mechanism is only effective when the value of transitions can be accurately estimated.

3.4.3 Dueling Deep Q-Networks

Dueling Deep Q-Networks (Dueling DQNs) arise from the decomposition of the state-action value into two different functions, the state-value function $V(s)$ (i.e. the value of being in a given state) and action-value (i.e. the value of an action compared to its alternatives), or advantage, function $A(a)$ [72][82]. These functions correspond to different layers in the **DQN** that combined generate a singular Q-value described as:

$$Q(s, a; \alpha, \beta) = V(s; \beta) + \left(A(s, a; \alpha) - \frac{\sum_{a'} A(s, a'; \alpha)}{|A|} \right), \quad (3.2)$$

where α and β are the two individual function's parameters and $|A|$ is the total number of actions. This division of the Q-value into two functions makes it easier for the **DQN** to learn action-values.

3.5 Solution design

Before this dissertation, DataMiner used a static Dijkstra's algorithm to define the path that a scheduled booking's data must follow. Dijkstra runs on top of the Cisco Transport

block depicted in Figure 3.1, which includes a series of routers' interfaces capable of transporting data between edge nodes, and selects the k -shortest paths available for the data transfer and the customer's choosing. At this stage, the cost of each interface was assigned as a copy of its total bandwidth capacity. Besides assigning higher costs to higher capacity links, which is contradictory to the typical Dijkstra's approach, once an interface was created its cost would never change regardless of the network state (i.e. load or node failure). These were limitations that Skyline intended to overcome. The course of actions taken to find a solution to this problem can be divided into three phases, which are broadly detailed in the following subsections.

3.5.1 Dynamic weight routing in SRM

A preliminary solution to the problem proposed was the implementation of a dynamic Dijkstra, which can be structured as Figure 3.4 displays.

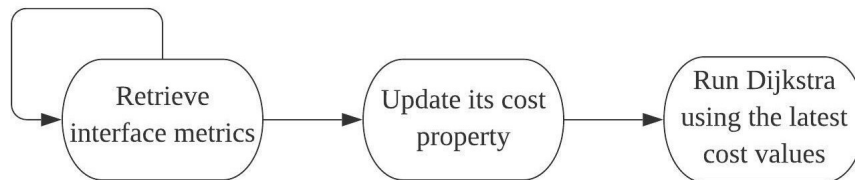


Figure 3.4: Dynamic Dijkstra's architecture.

This phase's goal was to promote the familiarization with Skyline's software, through the manipulation of network device's properties, and SRM's workflow logic, by transforming the existing static link cost solution into a dynamic alternative.

3.5.2 Dynamic weight routing in the link cost modification simulation

The purpose of this simulation was to replicate SRM's behaviour and extrapolate results of what is expected to be SRM's response to the link cost manipulation deployed in terms of QoS, in order to validate the proposed dynamic routing mechanism.

In this environment, the retrieval of interface metrics is performed by the controller in five seconds intervals in order to detect network state changes during the Iperf transmissions that simulate SRM's bookings. The controller then updates these costs, recalculates paths and when a new communication request arrives the best path is already pre-calculated. Thus, its framework is in every way like Figure 3.4 details.

3.6 Architecture of the routing optimization mechanism using DRL

The objective of this work can be summarized in the intent to create a dynamic algorithm capable of selecting the best path to assign to one or multiple upcoming multimedia

services (i.e. bookings). Given the nature of the transported content, there were a few requirements, imposed by Skyline or their software's architecture, that had to be followed. When a booking is scheduled, the customer expects to have the full requested capacity available at any instant of the service's duration. This makes it difficult to route upcoming bookings through interfaces that are already partially occupied by other active bookings by only considering their current available bandwidth, since there are bandwidth fluctuations during the content's transmission, which could incite the overbooking of interfaces that are just currently not being fully loaded. Therefore, and contrasting with the first solution implemented that used interface metrics retrieved from [SNMP](#)'s `ifTable`, in this case, the interfaces' available bandwidth is calculated considering the contracted bitrate of each service.

Thus, the designed model follows a heuristic approach to control the booked bandwidth in each interface and is trained to select paths with the lowest reserved bandwidth. This model is defined as $M = \langle S, A, R \rangle$, where:

1. S is the **state space**. A state s is sampled from the state space at time step t and is represented by a 4-dimensional tensor with dimensions $[N, N, k, 1]$, where N equals the number of hosts n , or edge nodes, in the network and k equals a pre-computed⁴ number of paths $p_{n_{src}, n_{dst}}$ that can be used on bookings that connect n_{src} and n_{dst} . The rightmost index of the tensor's dimensions corresponds to the selected network representation metric, in this case, the available bandwidth in the bottleneck link of each path (i.e the link with the smallest available bandwidth), depicted as $abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}})$.
2. A is the **action space**. It can be defined as $A = \{a_1, a_2, \dots, a_k\}$, where k is the number of pre-computed paths and $a = p_{n_{src}, n_{dst}}$.
3. $R(s, a)$ is the **reward function** that translates an action a in state s to a reward value that is calculated according to the available bandwidth of the paths in the resulting state. It is defined as:

$$R(s, a) = \begin{cases} +50, & \text{if } abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}}) \geq 75 \\ +30, & \text{if } abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}}) \geq 50 \\ 0, & \text{if } abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}}) \geq 25 \\ -10, & \text{if } abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}}) \geq 0 \\ -100, & \text{if } abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}}) < 0 \end{cases}, \forall (n_{src}, n_{dst}) \in N \text{ and } p_{n_{src}, n_{dst}} \in k. \quad (3.3)$$

For each combination (n_{src}, n_{dst}) and $p_{n_{src}, n_{dst}}$, the reward function will increment or decrement the total reward according to the respective $abw(n_{src}, n_{dst}, p_{n_{src}, n_{dst}})$. This function intends to severely penalize requests that are assigned to use paths

⁴Using Dijkstra's algorithm.

with links that will be overbooked, causing congestion. However, if there was only a penalty for the specific request that causes a link's capacity to be overbooked, the reward function would be neglecting the previous decisions that led to that network state. Thus, to account for this situation, whenever the links' utilization rates are inside predefined ranges that are considered acceptable, positive rewards are assigned.

Not only this problem formulation tackles the customer-side requirements of the problem (i.e. complying with the negotiated service requirements), but also significantly reduces the computational load of both the initial solution, since Dijkstra was executed for every service and according to this approach it will only be executed before the training of the [DRL](#) algorithm, and of the link cost optimization mechanisms presented that required interfaces to be periodically updated.

From the existing and introduced families of [DRL](#) algorithms, Deep Q-Learning was selected to achieve this dissertation's goal due to its simplicity, the extensive studies in which it has shown good results and, most importantly, the discrete nature of the action space designed.

To train the [DRL](#) algorithm, it must be submitted to multiple environment episodes. Each of these episodes consists of a set of requests that simulate a service booking and will generate distinct states and stimulate the agent's learning. To make the algorithm robust, the duration of an episode (i.e. the number of simulated requests), the communicating pairs and the bandwidth reserved for each session can be variable.

Algorithm 2 abstracts from the theoretical approach detailed in Algorithm 1 and provides a broad idea of how the training process occurs in our solution proposition. After the `replay_buffer` and the main `model` and `target_model` Q-networks are initialized with equal weights, as Algorithm 1 specifies, the training loop will run for `i` episodes, a number that can be dependent on a multiple number of factors, in this case, mostly the network's dimensions. After control variables are initialized to indicate the number of active communication sessions in the current episode (`requests`), when an episode must end (`done`) and the number of total episode requests (`max_requests`), a series of requests are started inside a `while` loop until the variable `done` makes it stop. Inside that loop, the action to perform is selected according to the `argmax`, which returns the highest value action considering the current state and model weights. Then, that action is applied to the environment through the method `make_reservation` that initializes a simulated reservation between two endpoints sampled from lists of possibilities, assigns it a predefined or randomized amount of bandwidth, and allocates it to the path computed. This method will return the updated state, which will account for the request that was just initialized. This state will then be evaluated according to equation 3.3 and produce a reward. Afterwards, the current transition is added to the replay buffer and the model is updated using a sample of transitions from the replay buffer and the update rule of the model's architecture (e.g. equation 2.16 if it is a basic [DQN](#)). Additionally, with a given update

Algorithm 2 Training environment

```

1: replay_buffer ← ReplayBuffer()
2: model ← Q_Network()
3: target_model ← model
4: for i episodes do
5:   requests ← 0
6:   done ← False
7:   max_requests ← N
8:   while not done do
9:     action ← argmax(model, state)  ▷ The action will be received from the model
    according to the current state.
10:    transformed_state ← make_reservation(action)  ▷ The communications will
    occur between edge pairs sampled from a list of possible hosts with a given request
    bandwidth.
11:    reward ← evaluate(transformed_state)
12:    requests ← requests + 1
13:    replay_buffer ← Add(state, action, reward, transformed_state)
14:    model ← update_model(sample(replay_buffer))
15:    if i == update frequency then
16:      target_model ← model
17:    end if
18:    if requests == max_requests then
19:      done ← True
20:    end if
21:  end while
22: end for

```

frequency, the target model's weights will match the main model's. Finally, the number of requests is incremented and if it has reached the total number of episode requests, the loop stops and a new episode, with new communication sessions, starts.

IMPLEMENTATION

In this chapter, the implementation of the solution elements presented in Chapter 3 is described. The public code of the implementation can be found at <https://github.com/diogomgsimoes/DRL-Network-Path-Selection-For-Multimedia-Traffic-in-SDNs>.

4.1 Dynamic weight routing in SRM

The first step consisted in the implementation of a dynamic weight path calculation protocol by using three distinct link cost equations mentioned in [49] (MHA, DSP and LIOA). Regardless of the equation, the implementation process was similar.

The selected link cost equations are based on up to three main parameters: the interface's total bandwidth capacity, its current utilization and the number of transported flows. While the first two values are gathered from the SNMP ifTable, the latter had to be introduced into the system.

With the cost metrics chosen, the remaining decision to make was when to update the interfaces' cost property. This decision implies major differences in implementation.

4.1.1 Using automation scripts

The first approach regarding the periodicity of the costs' update was to have it happen at the start of every new booking. Given this decision, the implementation was conducted using three different automation scripts:

1. "SRM_AddDcfInterfacesAsResources": For simplicity, this will be referred to as **Script A**. It is the core script in which the routers' interfaces are added as resources, specifically FunctionResources, and where their properties can be manipulated.
2. "SRM_NetworkPathSelection": **Script B** is responsible for calling auxiliary scripts that handle the path selection when a booking is being scheduled. It also increments the number of flows property.
3. "SRM_FlowsNumberDecrement": **Script C** is responsible for handling the flows number decrement when a booking ends.

Script A is ran manually and receives as input a serialized [JSON](#) string, which is then deserialized into a C# object of the `InputData` class. In this object, relevant parameters are specified, such as the capabilities and capacities of the resources, the functions they embody, the protocol they follow and to which resource pool they will belong. Concerning the model of the interface cost, two parameters are used to refer to `ifTable` column indexes, the `CostColumnID`, which holds an interface's total bandwidth capacity, and the `BandwidthUtilizationColumnId` that refers to an interface's current utilization level.

```
{
  "Capabilities": [],
  "Capacities": [
    {
      "CapacityValue": 55.55,
      "ColumnId": null,
      "ProfileParameterName": "Bitrate"
    }
  ],
  "CostColumnId": 2815,
  "BandwidthUtilizationColumnId": 2816,
  "Functions": [
    "IP Interface - Tx (Cisco Nexus)",
    "IP Interface - Rx (Cisco Nexus)"
  ],
  "ParameterGroupId": 1,
  "ParameterGroupName": "Interfaces",
  "ProtocolName": "CISCO Nexus",
  "ResourcePools": [
    "Cisco Network"
  ],
  "View": null
}
```

Figure 4.1: [JSON](#) `InputData` example (from `DataMiner`).

With the information received from the `InputData` object, script A instantiates variables such as the resource pool and functions and fetches the elements that implement the specified protocol. Then, for each physical interface of those elements, the values of the [SNMP](#) `ifTable` columns required to calculate the interface's cost are retrieved. All these variables are combined in the creation of [DCF](#) interfaces, which are then added as resources if they are not yet registered in the respective resource pool. In case they already are, using [LINQ](#)¹'s capabilities, the two resource objects are compared and the existing resource's dynamic attributes, such as the interface's cost, are updated according to the new resource. Previously, this script's approach was to replace all existing resources regardless of their properties, which would cause static user-defined configurations to be

¹A C# query syntax.

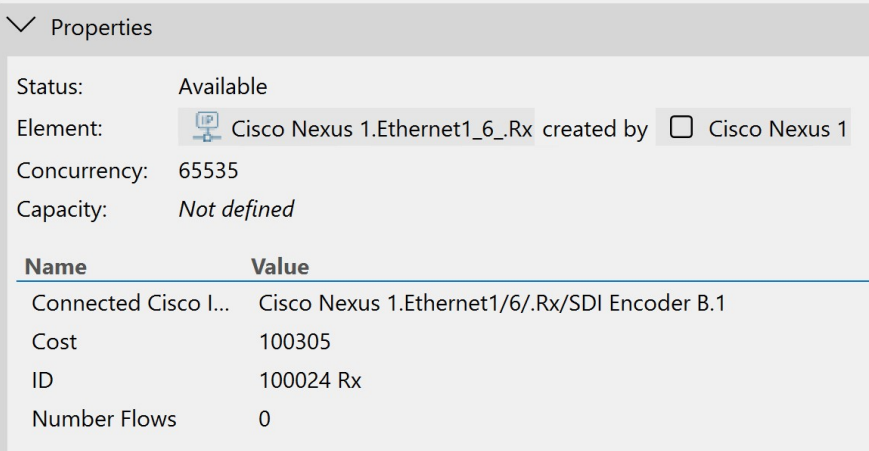
lost, such as the connectivity between [DCF](#) interfaces. With our approach, this problem has been surpassed.

The metrics used to calculate an interface's cost, specifically the current bandwidth utilization, can take advantage of [DataMiner's](#) trending feature. The `GetTrendData` method creates a `GetTrendDataMessage` request with the [DMA](#) Id of the element being processed, its element Id, the parameter to retrieve and the trending type, in this case, the average value registered between the also provided start and end times. Then, each cost equation that uses the available bandwidth metric considers its value as a relationship between the trend value of the last fifteen minutes, with a weight of 0.25, and the bandwidth utilization observed at the cost update time, with a weight of 0.75.

While most metrics used to calculate the interfaces costs are retrieved from a device's statistics, the number of flows needs to be handled through program logic. When a resource is added, its number of flows is 0, which will increment and decrement according to the interface's utilization.


Script B is responsible for the variable's increment. During a booking's scheduling in the [SRM](#) Booking Manager, this script is triggered, allowing the user to select the path he wants to route his traffic through. The system returns the k-shortest paths between the user-defined entry and exit nodes (i.e. the encoders and decoders) and the user selects the checkbox that corresponds to the preferred path. Then, by iterating through that path's connections, the interfaces of each node will have their number of flows incremented.

On the other hand, the process of decrementing this variable is very different, hence a new script had to be created. In the [SRM](#) Booking Manager it is possible to set automation scripts to run when a booking ends, which is the case of Script C. After fetching the ending reservation instance, if it is a [Transport-Cisco](#) booking, the resources that belong to its transport network, which were stored as a list of [GUIDs](#) in the reservation instance upon scheduling, have the property decremented.



Properties

Status: Available

Element:  Cisco Nexus 1.Ethernet1_6_.Rx created by Cisco Nexus 1

Concurrency: 65535

Capacity: *Not defined*

Name	Value
Connected Cisco I...	Cisco Nexus 1.Ethernet1/6/.Rx/SDI Encoder B.1
Cost	100305
ID	100024 Rx
Number Flows	0

Figure 4.2: Edge resource properties (from [DataMiner](#)).

4.1.2 Using a protocol

Script A takes the `ifTable`'s physical interfaces' data, creates `DCF` interfaces and not only configures their specific properties but also declares their connectivity to each other. Thus, its execution is a slow process. In the approach detailed in the last section, the intent was to execute it at booking run time, which caused its scheduling to become inefficient. For that reason, the script was ported to a protocol format. This way, there was the possibility to make the update of the resources happen with a given periodicity, making it possible to minimize the overhead of performing it while scheduling a booking.

A protocol begins with a `XML` template file where some metadata must be defined, including the protocol name, in this case, *SRM Routing Manager*.

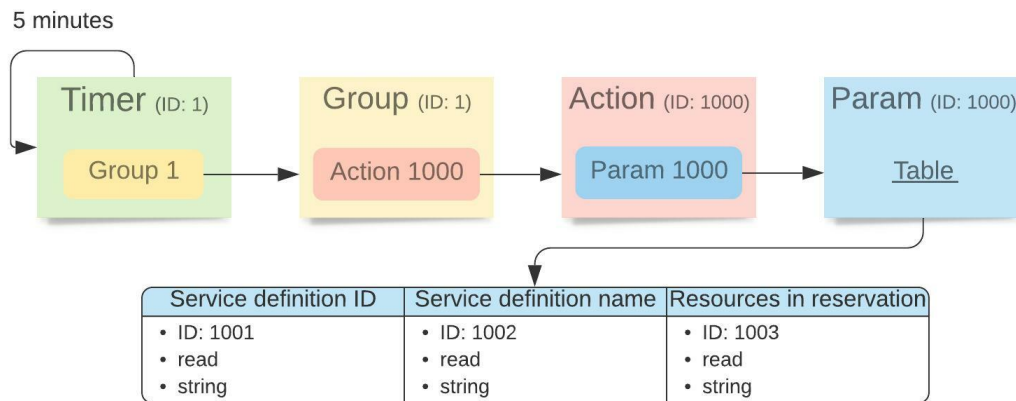


Figure 4.3: *SRM Routing Manager*'s workflow (adapted from [77]).

Besides the periodic link cost update, the protocol also includes the discovery of all edge nodes (i.e. interfaces of routers that are connected to encoders and decoders). Then, for each pair of edge nodes, the current shortest path connecting them is calculated and stored in an `UI` table parameter depicted in Figure 4.3. This way, when a booking is being scheduled between certain edges, the best path for that specific pair is always up-to-date and easily retrievable.

The *SRM Routing Manager* protocol is structured according to the model from Figure 4.3. The general flow of the protocol is controlled by a timer with a five-minute interval, at the end of which a group is added to the execution stack. When it comes to this group's turn of executing, it will poll an action and run it. Consequently, this action will run a `QAction` responsible for impacting the table parameter's columns, which are, according to the protocol's `XML` definition, parameters on their own.

To recall, before migrating into the protocol solution, the update of the interfaces' costs was managed by script A. We intended to transfer the logic implemented in that script to this protocol, however, while in automation scripts the object `Engine` is used to make `SLNet` calls, in protocols, it is replaced by the `SLProtocol` object. This implied a

complete code refactoring in order to find methods that deployed equivalent logic to the script's.

To calculate the paths, the methodology depicted in Figure 4.4 was implemented.

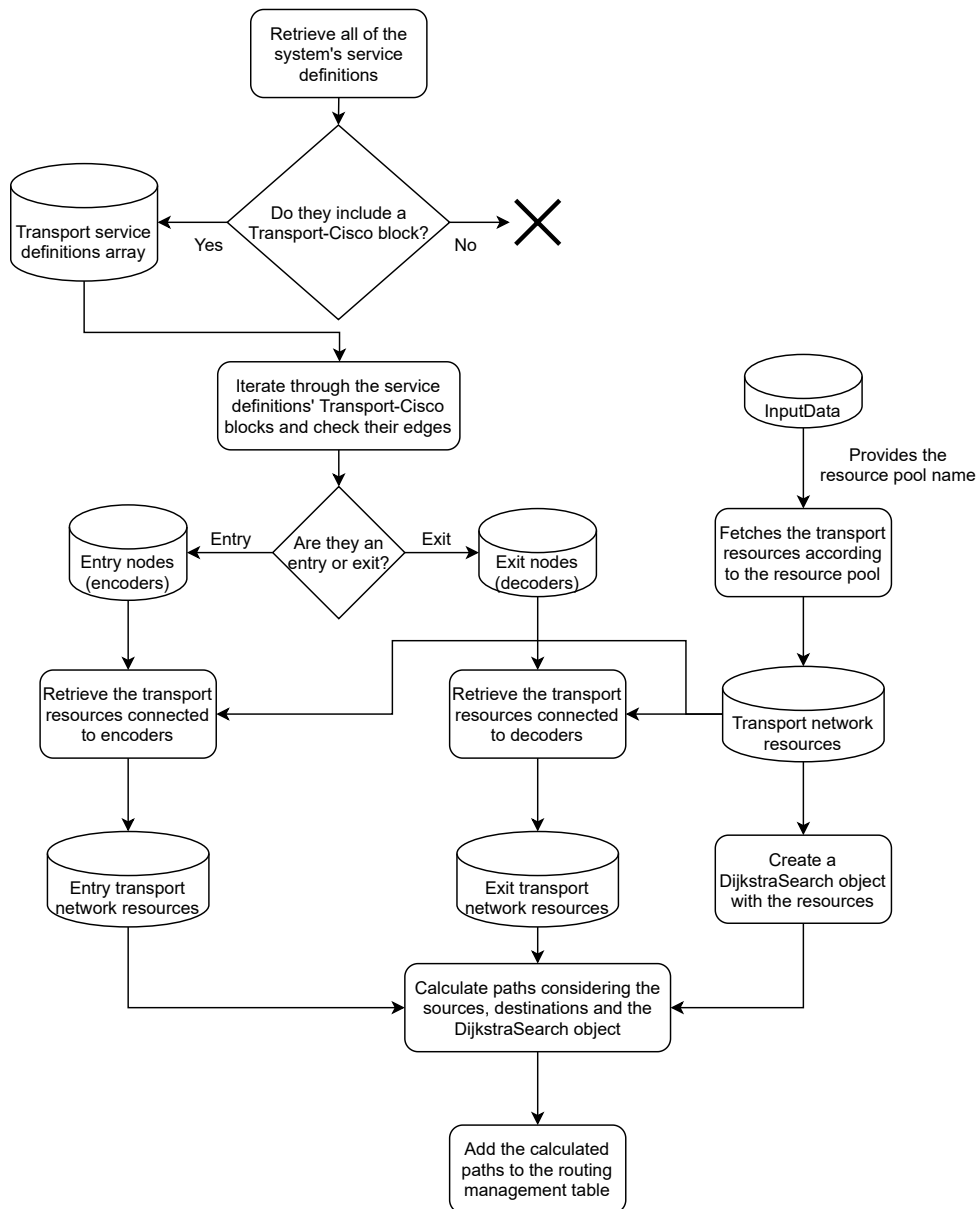


Figure 4.4: SRM Routing Manager implementation.

The process begins with the retrieval of every service definition registered in the system. Then, those service definitions are analysed and if they include a Transport-Cisco block inside of them, they are added to an array of transport service definitions. For each of those service definitions, the edges of their Transport-Cisco block are collected and separated according to whether they correspond to entry or exit nodes. If they are entry nodes, it means that they are SDI encoders, alternatively, the exit nodes are SDI decoders. Then, the Transport-Cisco network resources (i.e. DCF interfaces) are collected

based on their resource pool, which is indicated by the `InputData` object already presented, and stored in a list of resources. Using these resources and the entry and exit nodes discovered, two lists are created that hold the transport resources connected to encoders and the ones connected to decoders. This is possible due to a property called “Connected Cisco Interface” that is declared with the same value on both the encoders and decoders and the interfaces that they are connected to. The resources array is also used to create a `DijkstraSearch` object, which specifies the number of paths to calculate, the constraints of the search, the topology formed by the available resources, amongst other attributes. With the required inputs gathered, the Dijkstra algorithm can be executed and the list of path objects returned for each source/destination combination is saved. Using the `SLProtocol` object’s `FillArray` method, the `SRM` Routing Manager’s table is populated with the service definition Id (column “Service Definition ID” of the routing management table), its name (“Service Definition Name”) and the paths calculated (“Resources in Reservation”). These path objects include the source, destination and the actual path as attributes.

Service Definition ID	Service Definition Name	Resources in Reservation
61cf0fe8-aa6d-4acd-8a39-6284ed313627	Transport-Cisco_NEW	{{"Destination":"384V387","Path":[{"Capacity":0,"ConnectionID":5,"Cost":200607,

Figure 4.5: Routing management table (from DataMiner).

To finalize the protocol’s task, the path selection at booking scheduling and the number of flows increment, which were performed in script B, were moved to a different script. In its current state, when a customer is scheduling a booking, the path selection is dismissed, the best path for the specified endpoints automatically appears in the `UI`, and the interfaces in that path have their number of flows property incremented.

4.2 Dynamic weight routing in the SDN simulation

Recollecting, this simulation was created to test the link cost modifications implemented in `SRM` in an environment that is reactive to ongoing services (i.e. that reflects active services’ bandwidth utilization in the network links’ available bandwidth), which was not the case of `SRM` as explained in Section 3.1.2. Furthermore, this system will also be adapted to test the `DRL`-based solution and compare it to the dynamic weight routing approach.

To implement the desired behaviour, the developed system required four components: a Ryu controller script, a Mininet simulation script that loads a topology from a text file, a path computation script and a text file that works as a data transfer mechanism between the controller and simulation scripts.

The controller is constantly monitoring the network state (i.e. the available bandwidth and the number of flows occupying each link). The update of these metrics is performed every five seconds. After their values are changed, multiple link cost equations can be applied in the dynamic weight routing algorithm. These costs are then used in the path

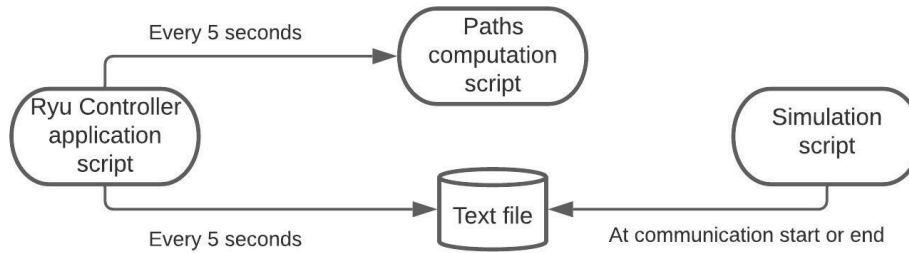


Figure 4.6: Ryu controller and Mininet [API](#) integration.

computation script, which builds a graph from the topology text file and the received link weights and calculates the current shortest path between each pair of hosts in the network using NetworkX². These paths are then stored in the controller. The transport requests are originated from the simulation script that, after the topology loading, will perform Iperf commands to inject traffic into the network and simulate video streams. Using threads and timers, the active communications (i.e. active Iperf sessions) are kept updated in a list data structure. This list will also be monitored in a thread that will update the data transfer text file every time changes occur. This will let the controller know which hosts are communicating and, consequently, allow it to not modify those paths during an Iperf, in resemblance to [SRM](#)'s bookings.

4.2.1 Mininet network

The network is built using Mininet's Python [API](#). First, a network object was created from the `Mininet` class, specifying the connection to a remote controller, the switch type as `OVSSwitch`³ and the link type as `TCLink`, which allows for the definition of each link's bandwidth capacity, delay and loss. Then, a text file describing each connection of the topology is read. Each line details a connection from either host to switch or between switches, as well as the bandwidth of the connecting link, while the delay was set to 1 millisecond and the loss to 0 on every link. While each encountered host and switch are added using the network's `addHost` and `addSwitch` methods, those same devices are connected through the `addLink` function. Then, the controller is added, and the network is started.

²A Python package used to create and manipulate networks. It offers a vast amount of built-in functionalities, such as the computation of shortest paths.

³"Open vSwitch is a production quality, multilayer virtual switch licensed under the open-source Apache 2.0 license." [83]

4.2.2 Ryu controller

For the Ryu controller, a proactive approach was followed. This translates to the pre-installation of flow rules in the switches before traffic arrives. Such approach allowed for a significant reduction of the network's setup time comparing to the [Media Access Control \(MAC\) learning](#)⁴ alternative, enabling the emulation of complex and large-scale topologies. Thus, at the controller's start of operation, the topology is retrieved from the same text file mentioned in the last topic and used to build the following dictionary data structures:

1. `adjacency`: Holds pairs of connected switches as keys and the port through which packets must leave the first switch to reach the second as value.
2. `host_to_switch_port`: A nested dictionary that maps a host's [MAC address](#) and a switch's Id to the switch port that connects them.
3. `costs`: Uses a pair of switches as the key and the cost of the link that connects them as the value.
4. `number_flows`: Maps pairs of switches to the number of flows occupying their connecting link.

Furthermore, a traffic monitoring system is assembled using two methods, `_monitor` and `_request_stats`. Every five seconds, they send an `OFPPortStatsRequest` message to each datapath registered in the controller. Switches are connected to the controller and added to a list of datapaths by a function decorated with the `OFPSwitchFeatures` event. This method adds and removes datapaths from the list as they connect or disconnect from the controller. When a switch is establishing a connection with the controller, activating the `OFPSwitchFeatures` event, a table-miss flow entry is added, whose importance was detailed in [Section 2.1.3](#).

The switches that receive the stats request from the monitoring system will return a message with a stats object. For each stat received whose port matches a port saved in the `adjacency` dictionary, the switch pair link's available bandwidth is calculated, as well as its cost and number of flows. Due to changes in the links' costs, the paths between host pairs must be updated. Once the correct paths are calculated, the existing flow rules are uninstalled from the switches and replaced with the most recent paths.

Since this controller is proactive, it does not receive packets from the datapaths. This means that, besides installing the required routing rules, it is also needed to pre-install [ARP](#) rules to both fill the hosts' [ARP](#) tables and let them know where to reach their destinations and prevent errors such as broadcast storms. This is done in the simulation script right after the topology is built.

⁴The name given to the learning of [MAC](#) addresses reachability through information in the arriving packets.

4.3 Deep Reinforcement Learning approach for path selection

A [DQN](#) on its own, as mentioned in Section 2.5.4.2, can be unstable. Therefore, in order to stabilize the learning process, the replay buffer and target network techniques were used. Furthermore, the enhanced [DDQN](#) and [Dueling DQN](#), algorithms presented in Section 3.4, were implemented and compared to the basic [DQN](#).

4.3.1 DRL ecosystem

To train these agents, three building blocks are required:

1. Environment (`RoutingEnv`): Manages the state, action and reward variables fed to the agents to train on.
2. Environment engine (`DRLEngine`): Provides the environment with the required logic to manage the state, action and reward variables.
3. Topology file: Holds the links (i.e. source, destination and link capacity) that characterize a topology.

The interaction between these parts is described in Figure 4.7.

In the green block, the `DRLEngine` python class is represented. Inside, the grey block encloses what belongs to the `init` method of this class. This method begins with the initialization of two data structures, the `communication_pairs` list, containing all the pairs of hosts that will be communicating during the training process, and the `state_helper` dictionary, which will be explained further along this section. With the input of the topology file, the `upload_topology` method creates two more dictionaries that represent the links' bandwidth capacities (`link_bw_capacity`) and their current available (i.e. unbooked) bandwidth (`current_link_bw`). On the other hand, the `build_graph` method also uses the topology file, but to create a `NetworkX` graph (`graph`). Lastly, using the `graph` object, the `calculate_paths` method will populate a dictionary (`paths`) using every pair of hosts combination as key and the `k`-shortest paths calculated with the `NetworkX` package as value. The blue block inside `DRLEngine` is responsible for interacting with the `RoutingEnv`.

On the right, the blue block `RoutingEnv` depicts the main methods of such class. Its `init` method initializes the engine, control variables and the [DRL](#) entities `observation_space`, `action_space` and `state`. The first two are defined according to two `OpenAI Gym`⁵ package's standard space types, the `Box`, which is described using the maximum and minimum values a state can have, and the `Discrete`, which is a set of integers $\{1, \dots, k\}$, where k is the number of possible paths between edges as described in Section 3.6. Additionally, the `state` is equal to the result of a `DRLEngine` method (`build_state`) that returns a `numpy`⁶ multidimensional array with dimensions $(N, N, k, 1)$, where k and N are the variables

⁵OpenAI Gym is a reinforcement learning toolkit for developing and testing algorithms.

⁶A Python package that offers various mathematical functions and data types.

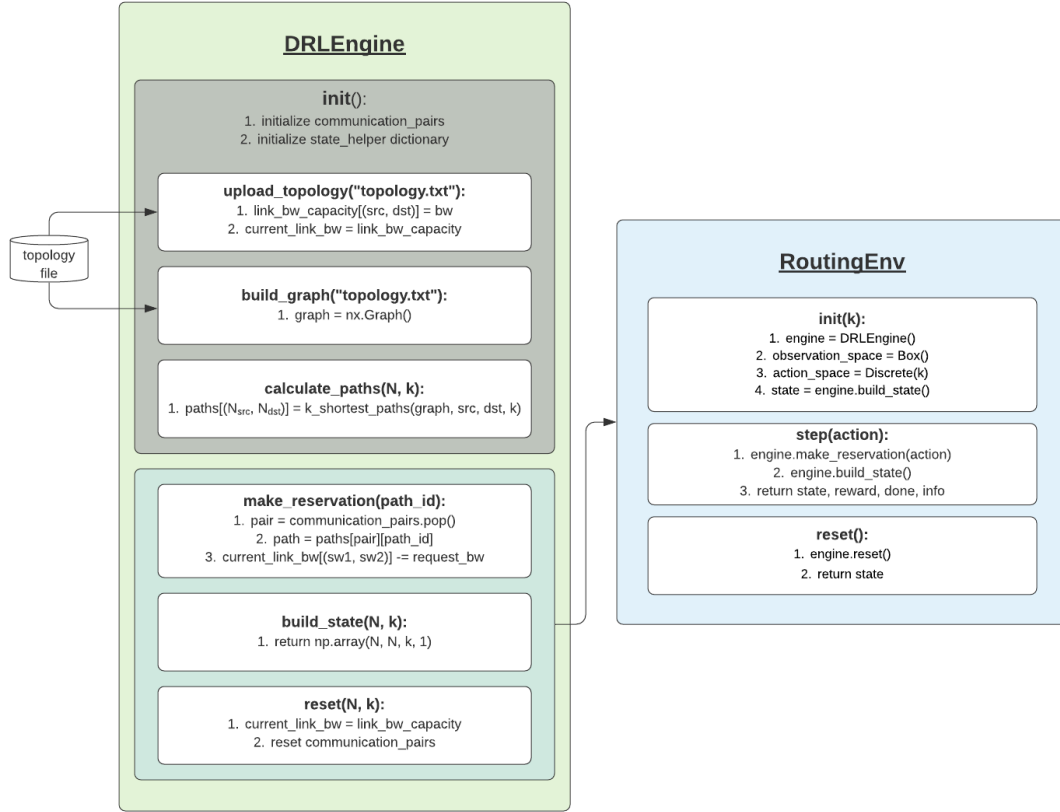


Figure 4.7: Interaction between the environment, the environment engine and the topology file.

introduced in Section 3.6, which is built considering the `paths` and `current_link_bw` data structures. Every time this method is called, the dictionary `state_helper` is updated with the total bandwidth capacity of the bottleneck link of a path, which is used in the state definition.

During the training phase, the environment's method `step` is called repeatedly. After receiving an action from the agent, the engine's `make_reservation` function is called to simulate a service reservation. To do so, it will sample a pair of hosts from the `communication_pairs` list and fetch the path that connects the nodes of the sampled pair and has the index of the received action. To manifest this reservation, the link bandwidth of each connection between switches in the selected path is decreased by the request bandwidth value. Afterwards, the `step` method requests an updated state to evaluate the effect of the placed reservation, using it to calculate the reward according to equation 3.3. To calculate the available bandwidth percentage required for the reward computation, the `step` method consults the engine's `state_helper` data structure and divides the bottleneck link's available bandwidth by its capacity. When the total number of requests matches the current episode request index, the `step` method ends with the return of the variables

state, reward, done and info⁷.

Between episodes, the method `reset` is invoked to clear the environment's control variables. This triggers the reset of the engine's control variables as well. Additionally, the `state` is returned.

This architecture is used to train the agents, however, to evaluate their performance after being trained, we resort to a variation of the Ryu controller ecosystem introduced in Section 4.2. The methodology used in that section's proposal promotes the recomputation of paths every five seconds, since the link weights are updated with that same frequency. As mentioned earlier, such approach is unbearable for DataMiner due to the computational cost of that operation. However, with the solution implementation described in this section, such update is not required. This is because Dijkstra is only executed once, at the start of the algorithm's training, to compute the path possibilities between hosts. Additionally, this computation is based on an unweighted Dijkstra, which only considers the number of hops to discover the shortest paths. Furthermore, the state of the network is kept updated in a centralized data structure that controls the bandwidth availability of the network devices. Thus, the agent learns which paths to choose based on the variations in this data structure. This approach also dismisses the need to update interfaces in the SRM environment. Additionally, to test the DRL agents developed, the environment's simulated requests are replaced with Iperf sessions.

4.3.2 Worst-case scenario

The definition of the communicating hosts plays a huge rule in the agents' training process efficiency and the performance of the resulting algorithm. To optimize these aspects, an algorithm to determine the worst-case traffic scenario was developed and used for both training and simulating the agents.

In this algorithm, the concept of centrality⁸ is explored, both for nodes and its abstraction for edges. Thus, the NetworkX package was used to fetch a dictionary that holds tuples (e.g. ("S10", "S15")) as keys and the edge's centrality ranking as value. From this data structure, the k paths calculated for each host pair are iterated by and a score is assigned to each of them. For each path, this score will be the sum of the centrality of its links divided by the maximum edge centrality value in the network. The resulting dictionary is then sorted by decreasing value and the host pairs that cause the most congestion can be extracted from the top item and onwards. This is useful to train robust algorithms and simulate high load situations where the routing solutions can be thoroughly tested.

⁷It is a gym environment standard to report something about the step or episode. However, it was not used in this implementation and always holds the value "{}".

⁸Centrality refers to a ranking that is assigned to nodes based on their position in the network. In this case, a node that is included in the paths that connect multiple host pairs has an higher centrality ranking than a node that participates in a path between only two hosts.

4.3.3 Agents

The implementations of the three chosen agents (**DQN**, **DDQN** and **Dueling DQN**) are based on those from [69], [84] and [85], respectively, and were done using PyTorch, a **ML** framework.

The agents interact with the **DRL** ecosystem presented in Section 4.3.1 through the `step` and `reset` methods. At each training step, the agent selects the action for the service being initiated, sends it to the **DRL** ecosystem and receives the updated state, the reward of the chosen action and a flag to indicate if the episode is over. If it is, it resets the **DRL** ecosystem. Internally, the **DRL** ecosystem manages the remaining data structures.

For **DQN** and **DDQN**, the network layers follow the structure summarized in Table 4.1 and illustrated in Figure 4.8.

Table 4.1: **DQN** and **DDQN** network's layers.

Type	Input Size	Output Size	Activation Function
Linear	STATE_SIZE	l1_out	ReLU
Linear	l2_in	l2_out	ReLU
Linear	l3_in	l3_out	ReLU
Linear	l4_in	N_ACTIONS	-

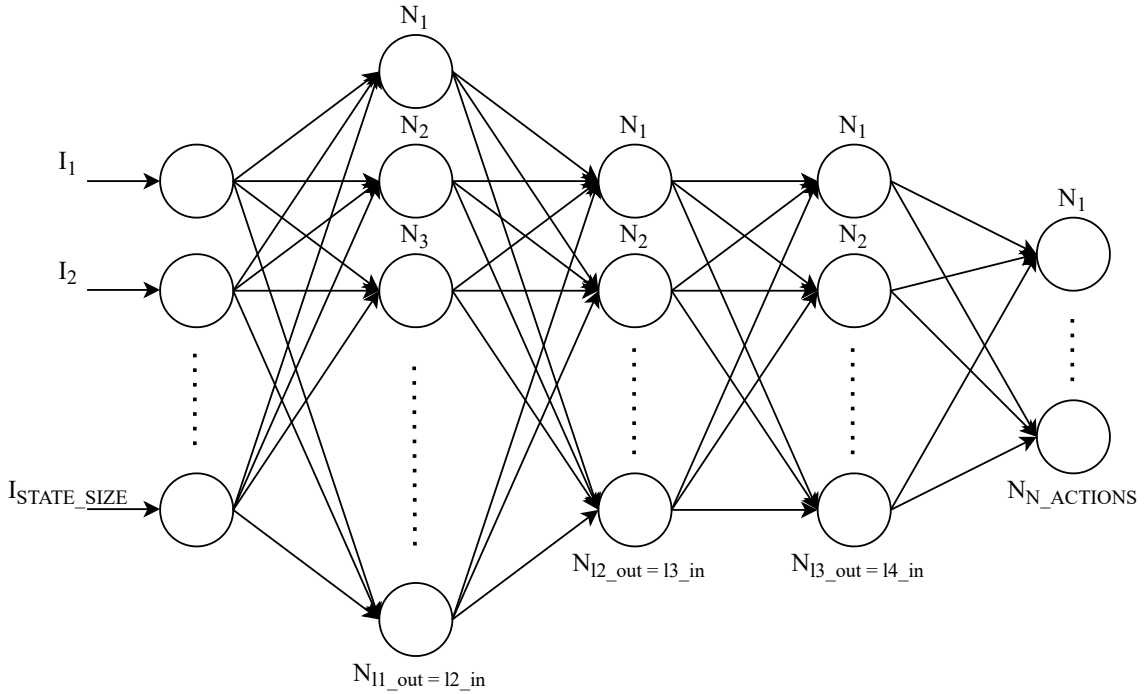


Figure 4.8: **DQN** and **DDQN** architecture.

Given its architecture, which separates the value and advantage functions, the **Dueling DQN**'s layers are organized differently, as Table 4.2 and Figure 4.9 demonstrate. In

Figure 4.9, the green blocks correspond to the value segment of the **Dueling DQN** and the red blocks to the advantage. After calculating the state’s value and the advantage of each action, the results are aggregated and the Q-value for each state-action pair is generated (in blue). These Q-values are calculated as the sum of the value function and the respective advantage subtracted by the average advantage of all actions.

Table 4.2: **Dueling DQN** network layers.

Type	Input Size	Output Size	Activation Function
Linear	STATE_SIZE	l1_out	ReLU
Linear	l2_in	l2_out	ReLU
Linear (Value)	l3a_in	l3a_out	ReLU
Linear (Value)	l4a_in	1	-
Linear (Advantage)	l3b_in	l3b_out	ReLU
Linear (Advantage)	l4b_in	N_ACTIONS	-

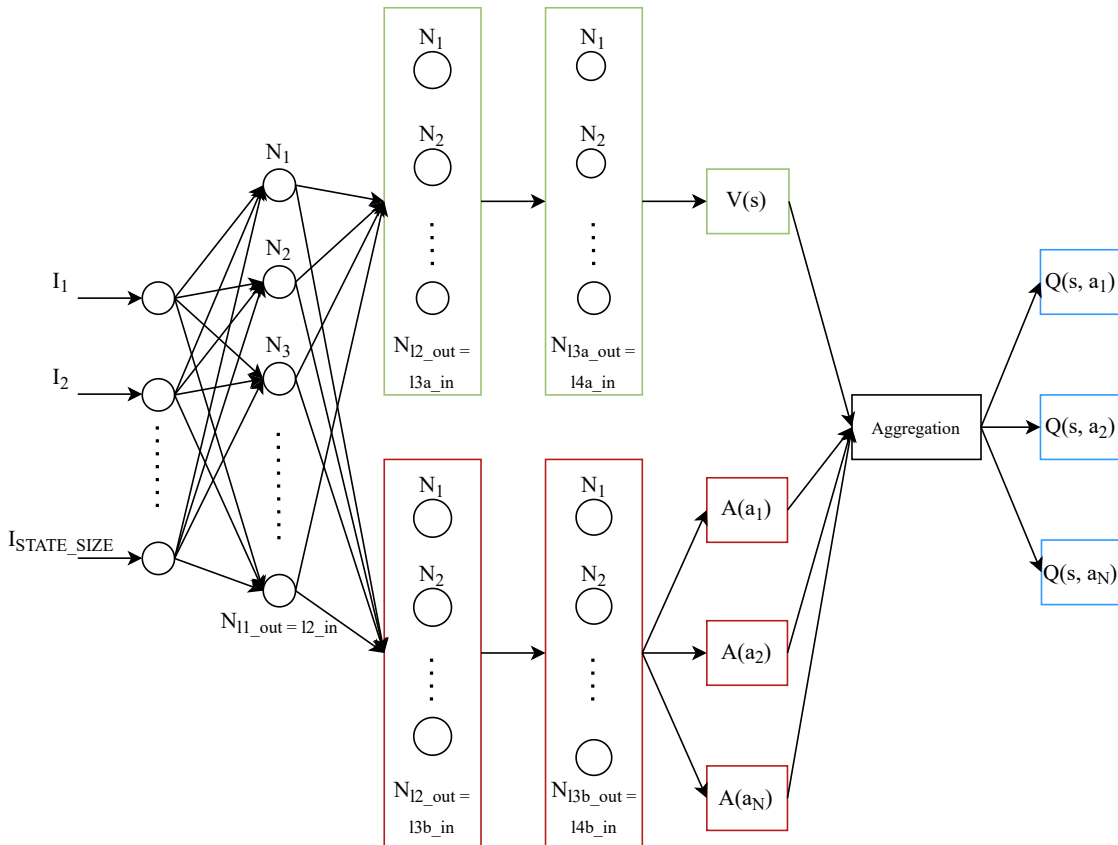


Figure 4.9: **Dueling DQN** architecture (adapted from [86]). In this figure “N” stands for “N_ACTIONS”.

In all cases, the “STATE_SIZE” is equal to the size of the flattened state *numpy* array and “N_ACTIONS” corresponds to the number of possible actions the agent can choose

from (i.e. the number of possible paths between each hosts pair). The sizes of the inner layers are dependent on the topology the algorithm is applied to. If the network size increases (i.e. additional edges and possible paths between them), the state flattened array will contain more values, which means that more nodes will be used in each layer. Additionally, the chosen activation function for every layer in the models is [ReLU](#).

The agents' performance depends on the training process they undergo. This process is reliant on multiple parameters that must be optimized for the environment they are intended to manage. Those parameters include:

1. α : the algorithm's learning rate.
2. Initial ϵ : the initial exploration rate.
3. Final ϵ : the final exploration rate.
4. γ : the reward's discount rate.
5. Replay buffer size: the number of observations stored.
6. Batch size: the number of observations sampled from the replay buffer that are used to update the agent's policy.
7. Update steps: the periodicity with which the target network is updated.
8. Epochs: the number of episodes that the agent needs to understand the environment.

The update of the networks' weights with the batch of observations sampled from the replay buffer is based on the [MSE](#) loss function. Furthermore, the optimizer selected to minimize the error of that loss function was *Adam*, an adaptive stochastic gradient-descent-based algorithm [87].

RESULTS

This chapter contains the results obtained during the simulations of the developed [DRL](#) routing solution, the dynamic link cost algorithms and the static weight Dijkstra baseline. The chapter is divided into sections, which will cover the development and simulation environment used in this work, the metrics selected to evaluate the algorithms' performance, the used topologies and the deployed [DRL](#) environments, and the obtained results.

The three [DRL](#) agents developed ([DQN](#), [DDQN](#) and [Dueling DQN](#)) were trained on a machine with the following specifications:

1. Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz processor,
2. 16 GB RAM,
3. NVIDIA GeForce GTX 1660 SUPER graphics card,
4. 64-bit operating system.

Once trained, the agents were tested in the Ryu controller, Mininet and Iperf environment against the remaining routing approaches, which do not require a training phase. This simulation system is implemented in an Ubuntu VMWare virtual machine with 4 GB dedicated RAM and two dedicated processors from the main system. [DL](#) algorithms greatly benefit from strong processors and dedicated graphics cards to speed up their training phase. This information is useful to understand that, despite the training duration in the presented system being feasible in a real-world situation, using better hardware could still greatly reduce it.

5.1 Evaluation metrics

To compare and evaluate the different solutions, three metrics have been selected:

1. *Flow's average bitrate*, retrieved through Iperf reports.
2. *Flow's average [Round-trip-time \(RTT\)](#)*, recorded by Iperf reports.

3. *Number of uncongested flows.* This value is calculated by counting the number of flows that show, when rounded, an average bitrate equal to the amount requested. Abstracting from the simulation environment, this measure represents the number of bookings that meet the requested requirements.

These metrics have a strong influence on each other. When a network becomes congested, the bitrate of a flow decreases and the average [RTT](#) increases. Moreover, that same bitrate drop makes the number of uncongested flows also decrease. This happens because an increasing number of flows to transport in the same links causes an increase in packet loss. Since traffic is [TCP](#), this will stop the sender's packet transmission until the lost packets are re-transmitted. Network congestion is also reflected by high [RTT](#) levels that are caused by packet queuing in the routers that anticipate congested links.

5.2 Topologies

In the following subsections, the two topologies used in this study are presented.

5.2.1 ARPANET

The first network is [ARPANET](#), a pioneer topology for packet-switching networks and one of the first to implement the [TCP/IP](#) protocol stack.

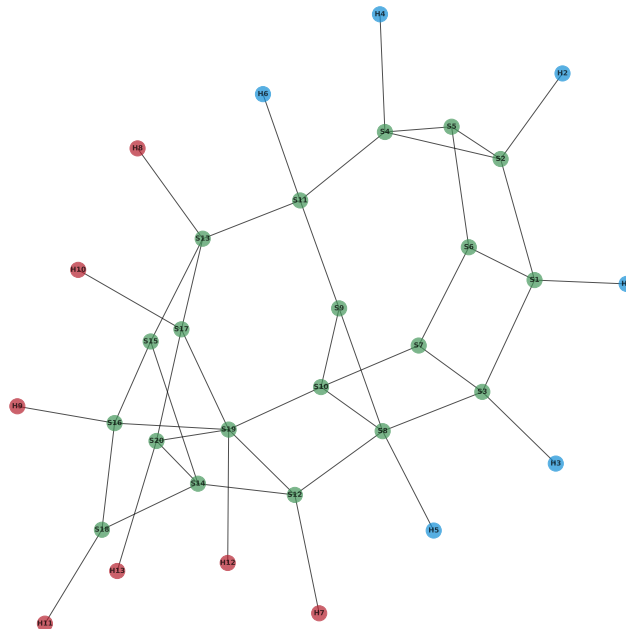


Figure 5.1: [ARPANET](#) topology with blue entry nodes, red exit nodes and green switches (adapted from [88]).

To load the topology from Figure 5.1 to Mininet, a text file containing the different links was written. Each line of this file is a space separated string, such as “S1 S4 100”, which means that switch 1 is connected to switch 4 through a link with a capacity of 100 Mb. Entry and exit nodes were added to the network in the form of hosts and connected to selected switches.

Thus, this network contains 13 hosts, 20 switches and 45 links. Given the network dimensions, the DRL environment state dimension is $[N, N, k, 1]$, with $N = 13$ and $k = 5$. The number of paths between each host was chosen through a manual analysis of the network. A state described by a *numpy* array with such dimensions, when flattened, is converted to a 1-dimensional *numpy* array with $[13 \times 13 \times 5 \times 1] = [845]$ size. Tailoring the dimensions of the DRL agents’ DNNs to this size, the resulting sets of layers according to the structures introduced in Sections 4.1 and 4.2 are presented in Tables 5.1 and 5.2.

Table 5.1: Agent’s networks layers (DQN and DDQN).

Type	Input Size	Output Size	Activation Function
Linear	845	1500	ReLU
Linear	1500	700	ReLU
Linear	700	200	ReLU
Linear	200	5	-

Table 5.2: Agent’s networks layers (Dueling DQN).

Type	Input Size	Output Size	Activation Function
Linear	845	1500	ReLU
Linear	1500	700	ReLU
Linear (Value)	700	200	ReLU
Linear (Value)	200	1	-
Linear (Advantage)	700	200	ReLU
Linear (Advantage)	200	5	-

5.2.2 Real-world media network

Since the work in this dissertation was developed in partnership with Skyline Communications, access was granted to a media network topology that belongs to a real service provider. For the sake of simplicity and to maintain its confidentiality, this network will be called **Network RW** in the remainder of this document. Its structure is depicted in Figure 5.2.

This topology is uploaded to Mininet in the same format as ARPANET is. In this case, the hosts represent network edges, which are frequently connected to SDI decoders or encoders. Such topology poses as a scale up from ARPANET, containing 10 hosts, 98 switches and 131 links.

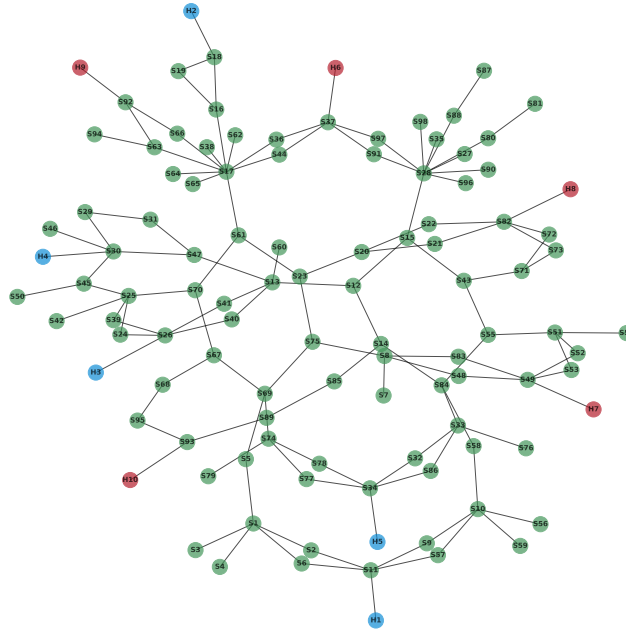


Figure 5.2: Network RW topology with blue entry nodes, red exit nodes and green switches.

Furthermore, the [DRL](#) environment state dimension is $[N, N, k, 1]$, with $N = 10$ and $k = 10$. Such state, when flattened, is converted to a 1-dimensional *numpy* array with $[10 \times 10 \times 10 \times 1] = [1000]$ size. The resulting sets of layers are depicted in [Tables 5.3](#) and [5.4](#).

Table 5.3: Agent’s networks layers ([DQN](#) and [DDQN](#)).

Type	Input Size	Output Size	Activation Function
Linear	1000	1800	ReLU
Linear	1800	900	ReLU
Linear	900	300	ReLU
Linear	300	10	-

5.3 Training settings

To optimize the [DRL](#) algorithms for this application framework, four distinct environment setups were designed, picturing different customer profiles.

Setup 1 uses fixed settings per training episode, which recreates a scenario where the network use is patterned. Therefore, in this setup, it was defined that each episode will have 32 concurrent requests and that each will require a bandwidth of 15 Mb. These

Table 5.4: Agent’s networks layers ([Dueling DQN](#)).

Type	Input Size	Output Size	Activation Function
Linear	1000	1800	ReLU
Linear	1800	900	ReLU
Linear (Value)	900	300	ReLU
Linear (Value)	300	1	-
Linear (Advantage)	900	300	ReLU
Linear (Advantage)	300	10	-

requests are selected from the worst-case scenario computations. Thus, not only the number of requests but also the sources and destinations of those requests remain the same across the whole training process. To recall, these requests are emulated strictly by a decrease of available bandwidth along their assigned paths, since the training process occurs without the integration of the [SDN](#) controller and Mininet system, as well as Iperf simulated services.

In order to make the agents more robust and prepare them for irregular network uses, some randomness was added to the training settings. **Setup 2** does that by varying the number of requests per episode to a random number between 1 and the number of requests per episode used in the first setup. All other settings remained unchanged.

Setup 3 simulates a tailored network use where some variation may still occur by turning this completely random approach to the number of requests per episode into a smoother variation that uses a normal distribution with a central value of 24 requests. Moreover, the bandwidth used by each communication session is selected randomly from a set of four possibilities: 5, 10, 15 and 18 Mb.

In the first three scenarios, the order of the hosts communication requests varies, however, the requests always have the same endpoints. **Setup 4** maintains the normal distribution to determine the number of requests per episode and the bandwidth randomization from scenario 3, and introduces host pairs selection for requests by sampling from two lists of possible sources and destinations.

In real scenarios, using the [DRL](#) algorithms in DataMiner, either an historical record of bookings will be used or the client will provide information (i.e. expected number of concurrent bookings, average bitrate per booking, network topology, number of desired paths per host pair, amongst others) to allow for that same parameter tailoring in the training environment setup.

5.4 DRL agents’ training

The agents’ performance depends on the training process they are submitted to. This process is reliant on multiple parameters that must be optimized for each topology the algorithm is applied to.

5.4.1 Agents' learning process in ARPANET

In [ARPANET](#), the training parameters, for all the agent implementations, were configured as follows:

Table 5.5: Agents parameters in [ARPANET](#).

Parameter	Value
α	0.001
Initial ϵ	0.5
Final ϵ	0.01
γ	0.99
Replay buffer size	50000
Batch size	256
Update steps	16

Using this set of parameters, the agents were trained in the four presented setups. Given the increasing complexity of the different setups, the remaining parameter, which is the number of epochs, will vary according to the setup configurations. Since setup 1 uses fixed settings, the agents are able to understand the environment faster. However, in the three remaining agents, more epochs pass before the agents stabilize their rewards. This increase is more pronounced in setup 2. To summarize, the four setups use 4000, 7000, 5000 and 5000 epochs, respectively. These values were defined by experimentation. The evolution of the reward along those epochs will be here presented in the form of plots, which will have the rewards as a function of epochs. However, the first setup will use a reward sum for each epoch (i.e. the sum of an epoch's total rewards), while the remaining setups use the average value of rewards in each epoch. This is due to the fact that the variation in the number of requests per epoch greatly affects the sum of its request rewards and, thus, impacts the ability to see the agents' learning tendencies. By using the reward average, this problem is diminished. These training sessions took, on average, from 3 to 5 hours, with [DQN](#) and [Dueling DQN](#) being on the lower end of the spectrum and [DDQN](#) on the upper. In order to smoothen the plots, averages of batches of epochs were used. Thus, the real horizontal axis value of each plot is found by multiplying it by the size of the batch, indicated in the vertical axis.

From the four graphs listed between Figures 5.3 and 5.6, it is possible to conclude that the [DDQN](#) agent has the best performance all around, while the [Dueling DQN](#) does not perform significantly or consistently better than the [DQN](#), despite its theoretical enhancements presented in Section 3.4.

In the first setup, the three agents can learn and stabilize their return values, which happens earlier for both [DDQN](#) and [Dueling DQN](#), when compared to [DQN](#). However, while they all begin their training with similar return values (i.e. the variation is mostly due to randomly assigned initial weights), the [DDQN](#) stabilizes the episodes' returns at nearly -425, a significant improvement over [DQN](#) and [Dueling DQN](#), both stabilizing

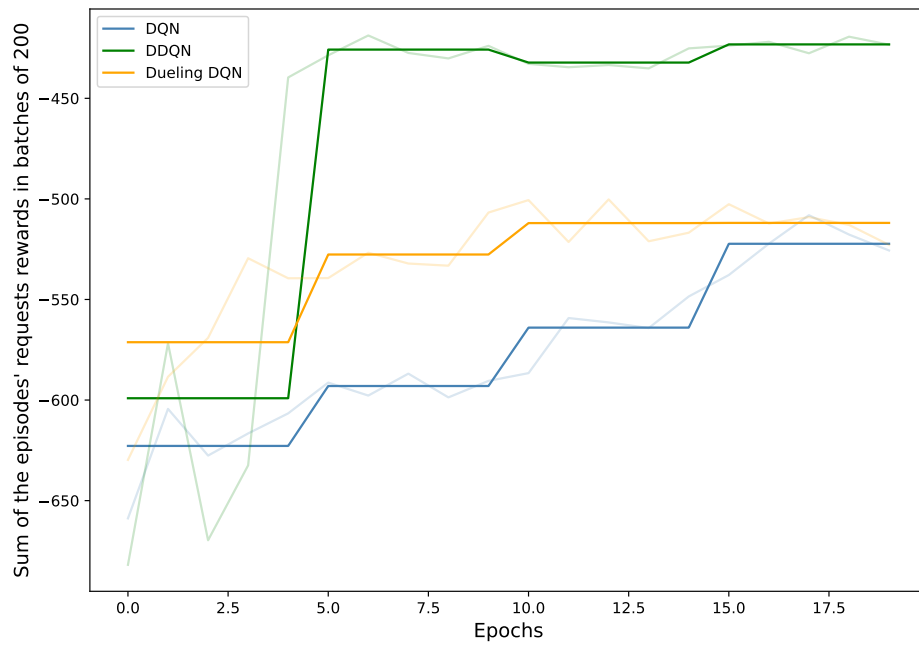


Figure 5.3: Comparison of training results between agents in environment setup 1.

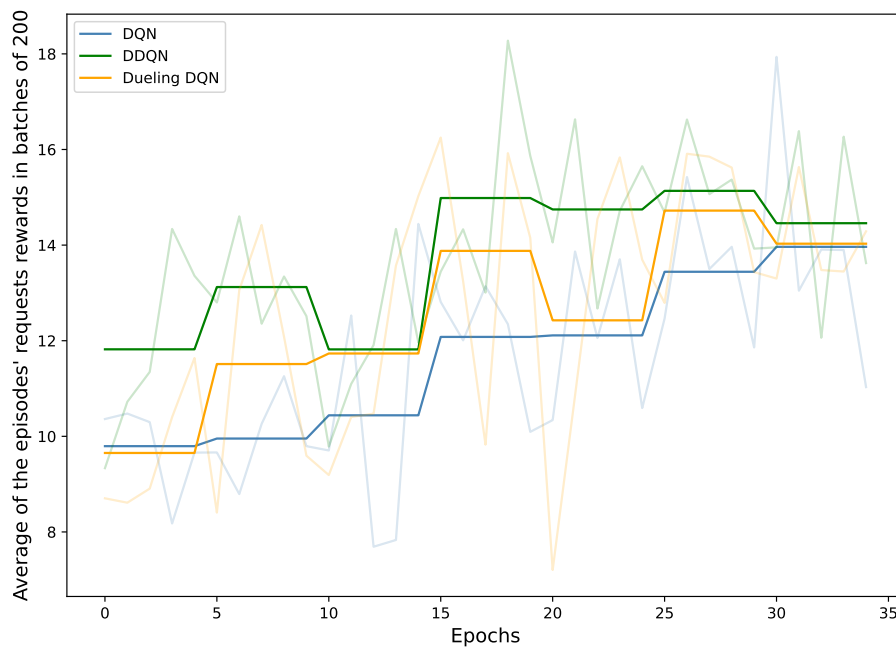


Figure 5.4: Comparison of training results between agents in environment setup 2.

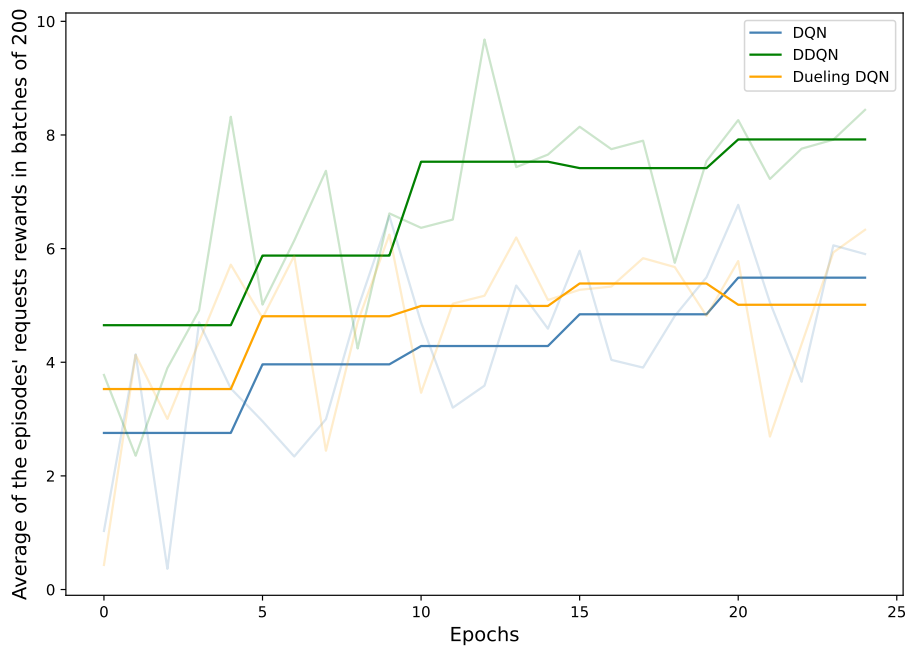


Figure 5.5: Comparison of training results between agents in environment setup 3.

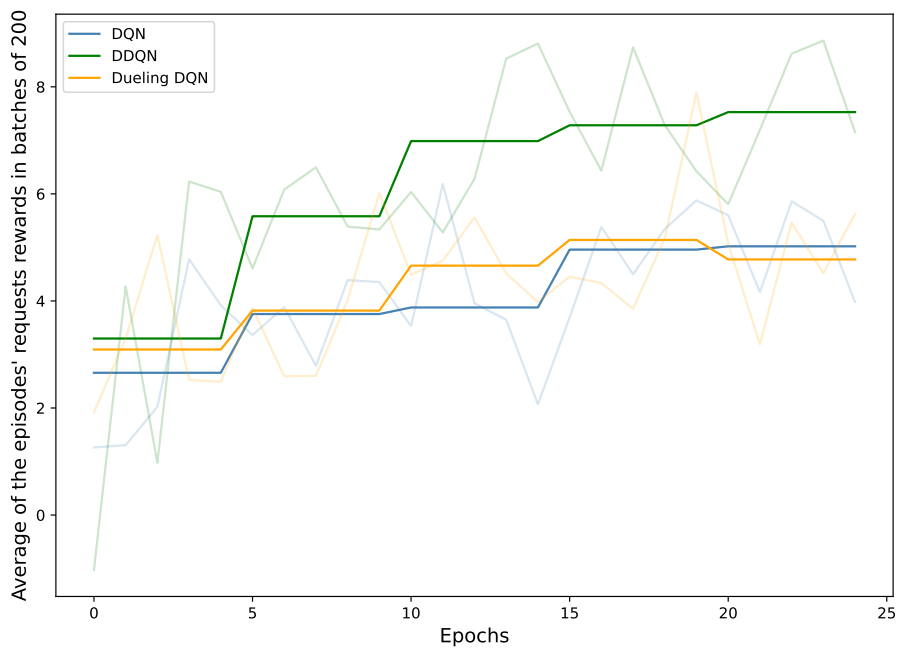


Figure 5.6: Comparison of training results between agents in environment setup 4.

at around -525. These values are negative due to the chosen reward function and its application to a worst-case scenario, where even if the best paths for each request are chosen, some links will always be overly booked since multiple requests are concurrently being supported.

The following three setups demonstrate the average reward of each request in an episode. In these configurations, the resulting plots cannot be as smooth as the first, since the variable number of requests per episode impacts such average. For instance, considering an episode with three requests, followed by an episode with ten. To the three requests in the first episode, bad actions can be chosen (e.g. the best path can be the one at index 0 and the agent returned 4 instead, for each of the requests), however, three requests have a low impact on the network state and, thus, even though the actions were not the best, the reward can be high. On the other hand, given the request concurrency, if the episode has ten requests, since the actions of the previous requests impact the state on a current request, the reward limit of request ten is not the maximum possible but the current maximum considering the already active requests. For this reason, even though the plots should, and do, still show an increasing tendency, it was expected some higher variance in the results. Despite these characteristics, [DDQN](#) continued showing better results than the remaining agents, achieving higher average request rewards per episode.

When the results obtained in setup 1 were presented it has been said that the rewards are negative due to the use of the worst-case scenario requests. In this setup's episode requests, the reward is positive in the beginning and starts decreasing as more flows are allocated due to the increase in congestion. Since our reward function is not linear and severely penalizes congestion, the requests allocated last result in exponentially lower rewards. In setups 2 to 4, the average of the episodes' requests' rewards presented are positive. This occurs because the number of requests per episode is lower, leading to lower congestion and, consequently, higher rewards. This approach does not suffer as much from an episode's last requests, which were the ones with the lowest rewards and that greatly decreased the total rewards achieved in setup 1.

The performance differences between the agents can be explained by their distinct behaviour as covered in Section 3.4. [DDQN](#)'s greater performance can be related to its use of a double estimator for Q-values, which prevents some actions from overshooting and getting constantly selected. This provides a better exploration of the network and a finer tuning of the agent which, in this case, is highly recommended because the order in which requests happen in an episode, the amount of requests it uses and the request specifications are variable. With this increased exploration capability, the agent will be able to test different actions when others have already assumed the best action, an assumption that might be flawed for the specific network state and incoming request. Overall, the [DDQN](#) agent is more robust. As for [Dueling DQN](#), its decoupling of the state and action values' prediction facilitates, in some cases, the learning of action values and improves its performance in comparison to [DQN](#).

5.4.2 Agents' learning process in Network RW

In the second topology, only setup 1 was tested, not just because it is the closest to what will be applied in real-world situations, but also to keep the study concise. Given the distinct network dimensions, also the environment specifications are different in this case (i.e. 24 requests instead of 32 with a 20 Mb requirement instead of 15 Mb). However, the agents' parameters are equal to [ARPANET's](#) (Table 5.5). Using the training specifications previously mentioned, the agents took 4000 epochs and between 6 to 8 hours to learn the environment.

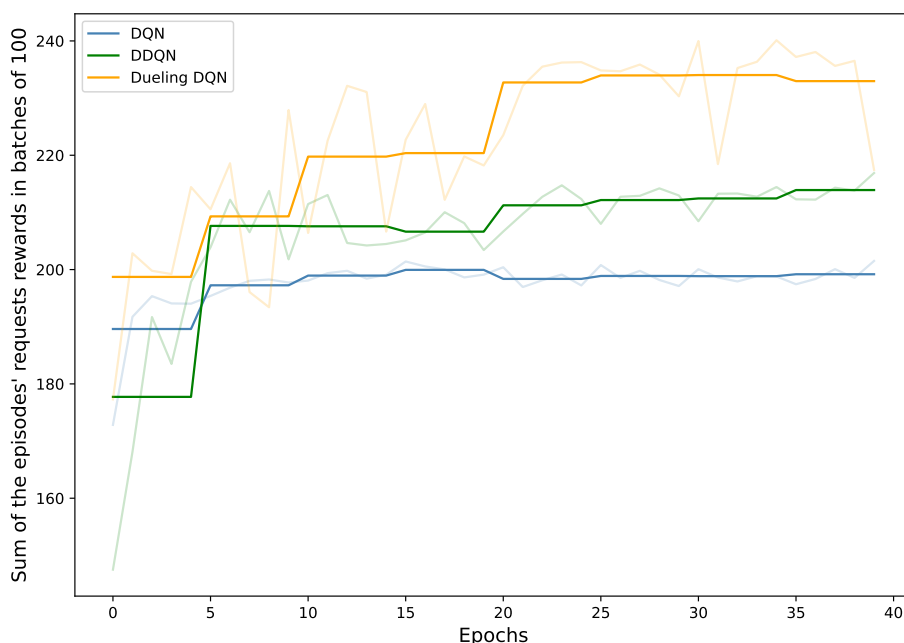


Figure 5.7: Comparison of training results between agents in environment setup 1 and Network RW.

Figure 5.7 exhibits the training process of the three agents in Network RW. In this case, the [Dueling DQN](#) shows a significantly better reward tendency than what it did in [ARPANET](#) and than [DDQN](#). This can be explained by the topology itself, which allows for the [Dueling DQN](#) to better benefit from the state action values separation. Moreover, [DDQN](#) is still superior to [DQN](#).

Contrarily to what was observed in the learning processes of the different agents in [ARPANET](#) and using setup 1, the rewards achieved in Figure 5.7 are positive. This indicates that the agents were able to assign paths that are better at avoiding congestion than in [ARPANET](#). This phenomenon will be further analysed, but it can already be understood that Network RW has a topology that is more suitable for load balancing than [ARPANET](#). Additionally, the different environment settings, such as the number

of requests per episode and their bandwidth utilization, might also positively affect the agents' performances.

5.5 Performance comparison between routing solutions

This topic will be divided into four segments:

1. Comparison of static weight Dijkstra to the dynamic link weight techniques studied.
2. Performance evaluation of the three agents trained in [ARPANET](#) and the four setups presented, measured against static weight Dijkstra.
3. Assessment of the best agent's suitability to unknown environment conditions.
4. Performance evaluation of the three agents trained for setup 1 and Network RW, compared to static weight Dijkstra.

5.5.1 Link cost optimization techniques

The first simulations conducted involved two selected link cost equations from [49], [DSP](#) and [LIOA](#), which were compared to static weight Dijkstra, specifically [MHA](#). The following results were obtained using 32 180-seconds-long Iperf requests in [TCP](#) with a requirement of 15 Mbits/s bitrate.

Table 5.6: Comparison between link cost optimization algorithms and Dijkstra ([MHA](#)).

Metric	MHA	DSP	LIOA
Bitrate (Mbits/s)	9.99	10.50	11.04
RTT (s)	0.331	0.349	0.314
Uncongested requests	6	6	7

From Table 5.6, which compiles the results of Table I.1, found in Annex I, it is possible to understand that both dynamic solutions show a slight improvement over Dijkstra's performance. This is explained by how [DSP](#) and [LIOA](#) can better balance the network traffic across different paths, avoiding the concentration of all requests in the same popular links. As such, this will result in higher bitrate levels because there will be less congestion. However, these approaches cannot fully explore the network's resources capabilities, which makes them unable to significantly improve on the number of supported requests without congestion metric. Additionally, this solution does not fit well into DataMiner's workflow, since the periodic interfaces' weight update introduces an undesirably high computational load into the system, as mentioned earlier in this document.

5.5.2 DRL versus Dijkstra in ARPANET

In this section, the performance of every [DRL](#) agent and setup combination is assessed using the settings presented in Section 5.5.1. With the goal of keeping the text organized

and uncluttered, the Iperf report results of the **DRL** agents simulations are enclosed in tables found under Annex II. Comparison graphs are presented here.

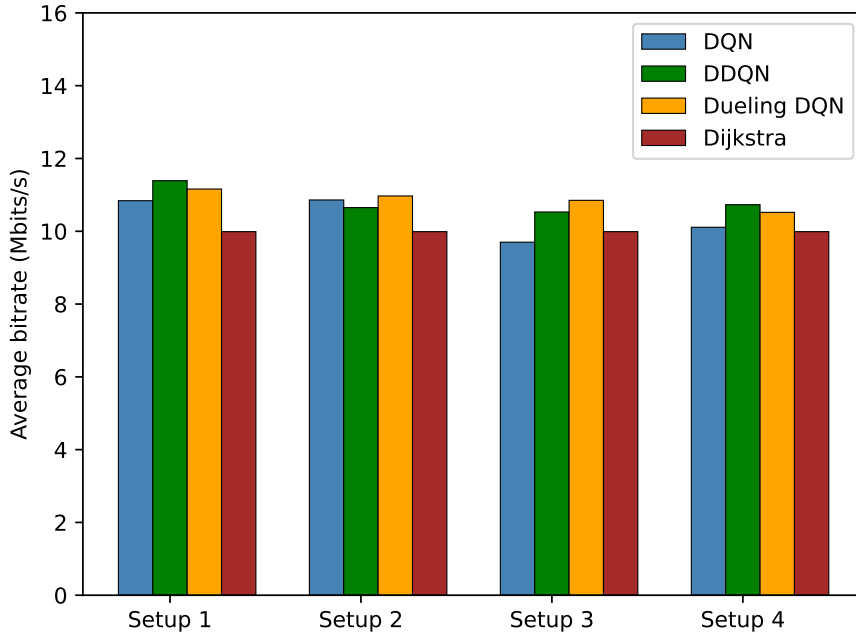


Figure 5.8: Average bitrate of each agent in each setup and Dijkstra’s.

After an analysis of the results, it can be concluded that **DDQN** is the best performing agent across all setups, as was expected due to its better performance in the training phase.

Regarding the average bitrate, in setups 1 and 2, all the agents outperform Dijkstra. Additionally, between agents, there is not much difference. In setups 3 and 4, however, **DQN** starts to fall back from the remaining agents’ performances and to match Dijkstra’s.

In terms of average **RTT**, the trend continues. In setups 1 and 2, the **DRL** agents are much better than Dijkstra but tend to deteriorate in setups 3 and 4.

The last evaluation metric is the number of uncongested requests. This value’s optimization is the primary goal of this work. In regard to such metric, a strong increase over Dijkstra, specifically in the **DDQN** agent, was achieved in setup 1. The **DQN** and **Dueling DQN** agents are also capable of outperforming the baseline routing mechanism in almost every scenario expect setup 4 (**DQN**).

A common trend for all metrics and agents is the algorithm degradation in setups 3 and 4, with emphasis in the **RTT** values. This can be explained by the introduction of bitrate variation in the agents’ training. Since bandwidth is the indicator of the network state used in the **DRL** environment, changes to its value impact the performance of the agents by making it more difficult for them to understand the underlying network. Setup

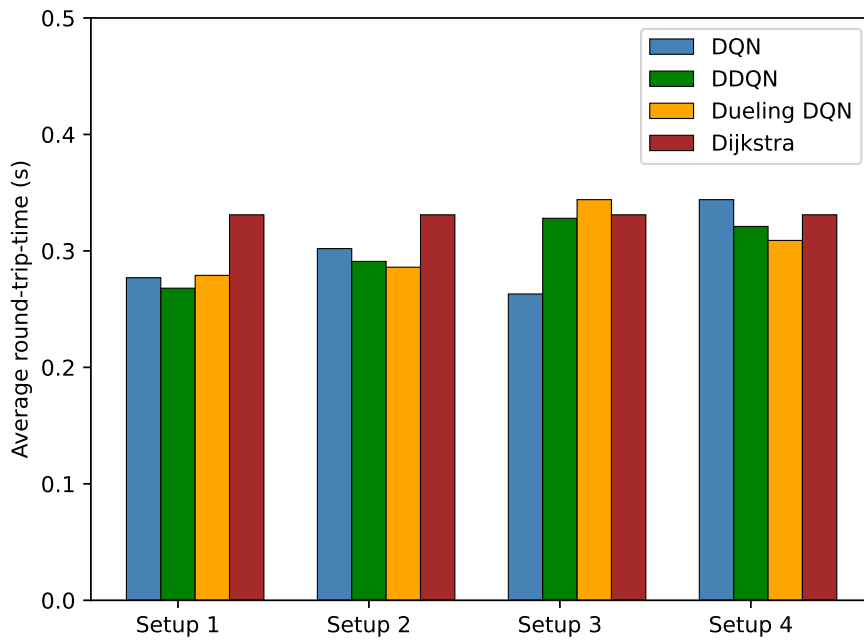


Figure 5.9: Average RTT of each agent in each setup and Dijkstra's.

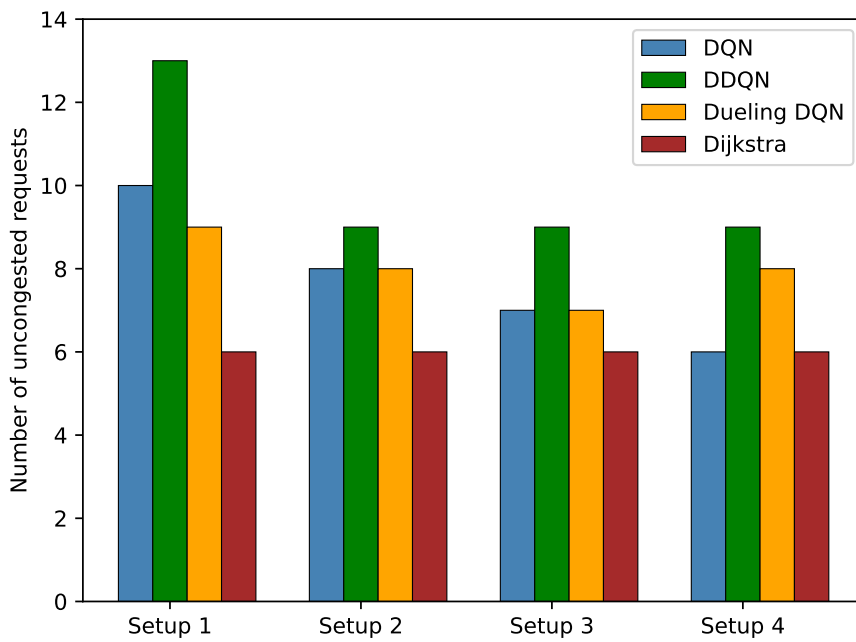


Figure 5.10: Number of uncongested requests of each agent in each setup and Dijkstra's.

1 is also clearly the most favourable environment for the algorithms. Setup 2, despite using a delimited random number of requests per episode, still manages to achieve better results than the last two, which supports the idea of the bitrate requirement per request having a stronger influence on the agents' performance than the number of active sessions.

Based on these observations, it is right to state that this [DRL](#) approach to routing is most suitable for patterned use networks, where the communication sessions usually occur between the same endpoints and maintain their requirements (i.e. bitrate).

5.5.3 DDQN's performance in unknown environment conditions

Although the results shown before demonstrate the algorithm's capability to surpass Dijkstra's performance, the requests used to train it are the same as the ones used to test it. Even though this is fairly close to what will be Skyline's reality, simulations were also ran on different settings to evaluate the algorithm's adaptability. Such settings are:

1. 14 requests.
2. 15 Mbits/s bitrate per request.
3. Hosts 1 and 2 as sources and the hosts 6, 8, 9, 10, 11, 12, and 13 as destinations.

For this test, only the best agent/setup pair was used (i.e. [DDQN](#) in setup 1).

Table 5.7: Performance comparison between [DDQN](#) (setup 1) and Dijkstra.

Metric	Dijkstra	DDQN
Bitrate (Mbits/s)	9.93	10.49
RTT (s)	0.365	0.371
Uncongested requests	1	4

From these results, it can be understood that even for unseen data, the developed algorithm can still make decisions that make it produce better results than Dijkstra.

This test is purely for reference. There can be situations where the best course of actions is always to choose the path with index 0 between two hosts, which corresponds to the path Dijkstra would use for that data transport. However, Dijkstra does not scale that well when congestion or network size increases, which gives a [DRL](#) agent a lot of room to explore.

5.5.4 DRL versus Dijkstra in Network RW

In Table 5.8, the [DRL](#) agents' performance in Network RW and setup 1's environment specifications is presented and can be used to understand this routing approach's adequacy to real-world situations. Such results were obtained through 24 Iperf sessions between the source hosts and every destination host in the network, with a bitrate requirement of 20 Mbits/s.

Table 5.8: Performance of the [DRL](#) agents compared to Dijkstra in Network RW.

Metric	Bitrate (Mbits/s)	RTT (s)	Uncongested requests
Dijkstra	14.84	0.412	2
DQN	18.88	0.192	12
DDQN	19.12	0.187	15
Dueling DQN	19.20	0.166	16

It is clear that all [DRL](#) agents greatly exceed Dijkstra’s performance. In all the developed agents, the average bitrate of the Iperf requests increases significantly and reaches levels close to their bitrate requirement. This is possible due to the placing of flows that start after others are active in more available paths, balancing the load evenly across the network and allowing the links to not get congested. Consequently, the [RTT](#) of these flows also decreases. These results prove the algorithm’s applicability to real-world situations, since not only the network capacity is maximized (i.e. more uncongested bookings), but each request’s bitrate and [RTT](#) are optimized.

Additionally, the results achieved in Network RW reveal a much more pronounced improvement than in [ARPANET](#). Network RW, since it belongs to a network operator, is built to enable load balancing. Thus, between two endpoints, more paths with the same length (i.e. number of hops) will exist, than in [ARPANET](#). Furthermore, with the baseline routing algorithm, even though five paths could all have the same cost (i.e. the same length), the chosen path would still always be the first returned in a Dijkstra search. This [DRL](#) solution, on the other hand, makes better use of the network topology and distributes traffic along multiple paths. Moreover, Network RW, given the larger scale of the network, has more path possibilities between nodes, hence the use of $k = 10$, which also spikes network exploration and increases its usability.

CONCLUSIONS AND FUTURE WORK

6.1 Final remarks

This dissertation explores new approaches to multimedia traffic routing in a [SDN](#) environment. An iterative approach was followed for the developed work. First, dynamic weight routing algorithms were studied and applied to DataMiner. However, these solutions entailed a few downsides, such as the high computational load of updating the routers' interfaces weights and the possible naive use of booked links with low current bandwidth utilization. Ultimately, a [DRL](#) algorithm was proposed that considers resources' contracted capacity in the allocation of traffic flows in a distributed and efficient manner. This way, the full capacities of a network's resources can be explored while attempting to increase the number of supported uncongested bookings and, consequently, optimizing the average bitrate and end-to-end delay verified in active bookings at a certain time period. This simple and computational efficient solution is expandable, however, the algorithm hereby presented is already capable of outperforming both static link cost solutions and dynamic link cost optimization algorithms (e.g. [DSP](#) and [LIOA](#)). These results were obtained on topologies built with Mininet, using a proactive Ryu controller to install forwarding rules and Iperf to inject traffic flows into the network. By analysing the Iperf results, one can conclude that all [DRL](#) agents developed can outperform the remaining routing optimization techniques in the simulated scenarios, both in the number of uncongested bookings, and the average bitrate and end-to-end delay metrics. Such approach thrives in networks with a well-tailored use, is scalable and can be applied to all sorts of network topologies.

6.2 Future work

The developed [DRL](#) algorithms are, on their own, capable of selecting accurate paths for given network topologies, considering network utilization levels. However, this dissertation's work is merely a baseline tool that can be further modified and optimized. There are several possible approaches on how to progress from this project onwards, which are

proposed in the following subsections.

6.2.1 Request tailoring

An important factor that impacts the training and, consequently, performance of the [DRL](#) agents is the set of requests used during those processes. The closer the two are, the better the agents will behave. Thus, one way to achieve this is through communication between Skyline and their customers. This algorithm will be incorporated in [SRM](#) so, when Skyline sells its software to a customer, besides retrieving the customer's physical network topology specifications, it is crucial to communicate and tune important parameters that have been mentioned earlier in this document. An alternative to this would be to have an historical record of services that could be analysed and used to create request templates for training purposes.

6.2.2 Traffic forecasting

This topic, covered in Section [2.3](#), can also be a great addition to this solution. One way to use it is to apply it to the services' history and have it predict new requests to be used to train the [DRL](#) agents in an online fashion. Not only that, but using traffic forecasting, it could be feasible to have predicted future services impact a path decision that is being made at a certain moment. If there was a way to reflect that expected upcoming communication in the environment state, the achieved performance could be even better than it is right now.

6.2.3 Complex neural networks

To accomplish what was just proposed, the use of more advanced neural network layers, such as [RNNs](#) or [CNNs](#), could be required. Even in the current solution, a refined neural network architecture could be explored to see if better results would be achieved.

6.2.4 Reward function

A major component of a [DRL](#) algorithm is the reward function used. In the present solution, such function was defined to target the selection of paths with more available links. However, besides the room that exists to optimize the presented reward function, which was designed by experimentation, other factors could be added to it to improve the agents' awareness of the impact their choices have on the network. Through actual network observation, which did not apply to this dissertation's use case but can for other situations, metrics like the end-to-end delay of services could be continuously analysed and taken into consideration in the selection of paths for upcoming requests.

6.2.5 Agent optimization

Despite this dissertation presenting results for three different agents, given the vast range of options that the field of [DRL](#) includes, there were still multiple agents (some of which are listed in [89]), from the [DQN](#) family (e.g. Quantile Regression [DQN](#) or Categorical 51-Atom [DQN](#)), or others (e.g. actor-critic algorithms' category), that could be tested and techniques (e.g. [PER](#) and noise) that could be applied to this problem's solution and be able to improve the results obtained.

6.2.6 Problem of topology change

The [DRL](#) agents presented in this document are trained for specific network topologies. In this case, migrating from [ARPANET](#) to Network RW requires a new training process. A valuable addition to this solution would be the ability to take a trained agent and apply it to any kind of topology. One way to achieve this could be through [Transfer Learning \(TL\)](#), which is a [ML](#) field that explores the possibility of using gathered knowledge from a given problem to apply it to a different but similar situation [90].

BIBLIOGRAPHY

- [1] R. T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. <http://www.rfc-editor.org/rfc/rfc2616.txt>. RFC Editor, June 1999. URL: <http://www.rfc-editor.org/rfc/rfc2616.txt> (cit. on p. xv).
- [2] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor, Dec. 2017 (cit. on p. xv).
- [3] BillWagner. *Language-Integrated Query (LINQ) (C#)*. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/> (cit. on p. xvi).
- [4] R. Enns et al. *Network Configuration Protocol (NETCONF)*. RFC 6241. June 2011. DOI: [10.17487/RFC6241](https://doi.org/10.17487/RFC6241). URL: <https://rfc-editor.org/rfc/rfc6241.txt> (cit. on p. xvi).
- [5] A. Bierman, M. Björklund, and K. Watsen. *RESTCONF Protocol*. RFC 8040. Jan. 2017. DOI: [10.17487/RFC8040](https://doi.org/10.17487/RFC8040). URL: <https://rfc-editor.org/rfc/rfc8040.txt> (cit. on p. xvi).
- [6] *About Skyline Communications*. Accessed October 20, 2020. URL: <https://skyline.be/skyline/facts> (cit. on p. 1).
- [7] B. A. A. Nunes et al. “A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks”. In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634. DOI: [10.1109/SURV.2014.012214.00180](https://doi.org/10.1109/SURV.2014.012214.00180) (cit. on pp. 5–7).
- [8] T. D. Nadeau and K. Gray. *SDN: Software Defined Networks*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O’Reilly Media, Inc, 2013 (cit. on p. 5).
- [9] L. Peterson et al. *Software-Defined Networks: A Systems Approach*. 2020. URL: <https://github.com/SystemsApproach/SDN> (cit. on p. 5).
- [10] D. Kreutz et al. “Software-Defined Networking: A Comprehensive Survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: [10.1109/JPROC.2014.2371999](https://doi.org/10.1109/JPROC.2014.2371999) (cit. on pp. 6, 9).

-
- [11] W. Xia et al. “A Survey on Software-Defined Networking”. In: *IEEE Communications Surveys Tutorials* 17.1 (2015), pp. 27–51. DOI: [10.1109/COMST.2014.2330903](https://doi.org/10.1109/COMST.2014.2330903) (cit. on pp. 6, 7, 18).
- [12] O. N. Foundation. *SDN Architecture 1.0 Overview*. 2014. URL: https://opennetworking.org/wp-content/uploads/2014/11/TR_SDN-ARCH-1.0-Overview-12012016.04.pdf (cit. on p. 6).
- [13] Y. Sung et al. “FS-OpenSecurity: A Taxonomic Modeling of Security Threats in SDN for Future Sustainable Computing”. In: *Sustainability* 919.9 (2013). DOI: [10.3390/su8090919](https://doi.org/10.3390/su8090919) (cit. on p. 6).
- [14] K. Cabaj et al. “SDN Architecture Impact on Network Security”. In: Sept. 2014. DOI: [10.15439/2014F473](https://doi.org/10.15439/2014F473) (cit. on p. 6).
- [15] O. N. Foundation. *SDN Architecture 1.0*. 2014. URL: https://opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf (cit. on pp. 6, 7).
- [16] O. N. Foundation. *SDN Architecture 1.1*. 2016. URL: https://opennetworking.org/wp-content/uploads/2014/10/TR-521_SDN_Architecture_issue_1.1.pdf (cit. on pp. 6, 7, 9).
- [17] A. Lara, A. Kolasani, and B. Ramamurthy. “Network Innovation using OpenFlow: A Survey”. In: *IEEE Communications Surveys Tutorials* 16.1 (2014), pp. 493–512. DOI: [10.1109/SURV.2013.081313.00105](https://doi.org/10.1109/SURV.2013.081313.00105) (cit. on p. 7).
- [18] *OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06) for information on specification licensing through membership agreements*. Tech. rep. 2015. URL: <http://www.opennetworking.org/%5C%7D> (cit. on pp. 7, 8).
- [19] K. Suzuki et al. “A Survey on OpenFlow Technologies”. In: *IEICE Transactions on Communications* E97.B (Feb. 2014), pp. 375–386. DOI: [10.1587/transcom.E97.B.375](https://doi.org/10.1587/transcom.E97.B.375) (cit. on p. 7).
- [20] M. P. V. Manthena. “Network-as-a-Service Architecture with SDN and NFV: A Proposed Evolutionary Approach for Service Provider Networks”. PhD thesis. Feb. 2015. DOI: [10.13140/RG.2.1.2446.2883](https://doi.org/10.13140/RG.2.1.2446.2883) (cit. on pp. 7, 8).
- [21] J. H. Cox et al. “Advancing Software-Defined Networks: A Survey”. In: *IEEE Access* 5 (2017), pp. 25487–25526. DOI: [10.1109/ACCESS.2017.2762291](https://doi.org/10.1109/ACCESS.2017.2762291) (cit. on p. 8).
- [22] H. Hawilo et al. “NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC)”. In: *IEEE Network* 28.6 (2014), pp. 18–26. DOI: [10.1109/MNET.2014.6963800](https://doi.org/10.1109/MNET.2014.6963800) (cit. on pp. 8, 9).
- [23] *SDN vs. NFV - What’s the difference? Software Defined Networking (SDN)*. Mar. 2020. URL: <https://www.cisco.com/c/en/us/solutions/software-defined-networking/sdn-vs-nfv.html> (cit. on pp. 8, 9).

- [24] L. S. Skyline Communications. “Software-Defined Networking – A White Paper”. In: *DataMiner Dojo* (Mar. 2021). URL: <https://skyline.be/learn/publications/software-defined-networking-white-paper> (cit. on p. 9).
- [25] M. Sanaei and S. Mostafavi. “Multimedia Delivery Techniques over Software-Defined Networks: A Survey”. In: *2019 5th International Conference on Web Research (ICWR)*. 2019, pp. 105–110. DOI: [10.1109/ICWR.2019.8765278](https://doi.org/10.1109/ICWR.2019.8765278) (cit. on p. 10).
- [26] Cisco. *Cisco Annual Internet Report (2018–2023)*. 2017. URL: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf> (cit. on p. 10).
- [27] R. Molla. *An explosion of online video could triple bandwidth consumption again in the next five years*. June 2017. URL: <https://www.vox.com/2017/6/8/15757594/future-internet-traffic-watch-live-video-facebook-google-netflix> (cit. on p. 10).
- [28] T. G. Skyline Communications. “The production Media Data Center: end-to-end monitoring and orchestration for ALL-IP Broadcast Infrastructures”. In: *DataMiner Dojo* (Apr. 2020). URL: <https://community.dataminer.services/the-production-media-data-center-end-to-end-monitoring-and-orchestration-for-all-ip-broadcast-infrastructures/> (cit. on pp. 10, 11).
- [29] E. Gallier. *The Migration Path to All-IP Infrastructures*. Accessed November 3, 2020. URL: <https://www.harmonicinc.com/insights/blog/migration-path-to-all-ip-infrastructures/> (cit. on pp. 10, 11).
- [30] P. Cortez et al. “Internet Traffic Forecasting using Neural Networks”. In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. 2006, pp. 2635–2642. DOI: [10.1109/IJCNN.2006.247142](https://doi.org/10.1109/IJCNN.2006.247142) (cit. on pp. 12–17).
- [31] R. Madan and P. S. Mangipudi. “Predicting Computer Network Traffic: A Time Series Forecasting Approach Using DWT, ARIMA and RNN”. In: *2018 Eleventh International Conference on Contemporary Computing (IC3)*. 2018, pp. 1–5. DOI: [10.1109/IC3.2018.8530608](https://doi.org/10.1109/IC3.2018.8530608) (cit. on pp. 12, 13, 16, 25).
- [32] C. Chatfield. *Time-series forecasting*. Chapman and Hall, 2001 (cit. on pp. 12–14).
- [33] P. J. Brockwell and R. A. Davis. *Introduction to time series and forecasting*. Springer, 2008 (cit. on pp. 12–14).
- [34] Jian-Chang Lu, Dong-Xiao Niu, and Zheng-Yuan Jia. “A study of short-term load forecasting based on ARIMA-ANN”. In: *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*. Vol. 5. 2004, 3183–3187 vol.5. DOI: [10.1109/ICMLC.2004.1378583](https://doi.org/10.1109/ICMLC.2004.1378583) (cit. on p. 13).
- [35] *Forecasting: Principles and Practice*. Accessed December 10, 2020. URL: <https://otexts.com/fpp2> (cit. on pp. 14, 15).

- [36] S. N. A. M. Razali et al. "Forecasting of Water Consumptions Expenditure Using Holt-Winter's and ARIMA". In: *Journal of Physics: Conference Series* 995 (Apr. 2018), p. 012041. DOI: [10.1088/1742-6596/995/1/012041](https://doi.org/10.1088/1742-6596/995/1/012041). URL: <https://doi.org/10.1088/1742-6596/995/1/012041> (cit. on p. 15).
- [37] C. Veiga et al. "Demand forecasting in food retail: A comparison between the Holt-Winters and ARIMA models". In: *WSEAS Transactions on Business and Economics* 11 (Jan. 2014), pp. 608–614 (cit. on pp. 15, 16).
- [38] M. Omane-Adjepong, F. Oduro, and S. Oduro. "Determining the Better Approach for Short-Term Forecasting of Ghana's Inflation: Seasonal ARIMA Vs Holt-Winters". In: *International Journal of Business, Humanities and Technology* 3 (Jan. 2013), pp. 69–79 (cit. on p. 15).
- [39] J. Kumar, R. Goomer, and A. K. Singh. "Long Short Term Memory Recurrent Neural Network (LSTM-RNN) Based Workload Forecasting Model For Cloud Datacenters". In: *Procedia Computer Science* 125 (2018). The 6th International Conference on Smart Computing and Communications, pp. 676–682. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2017.12.087>. URL: <http://www.sciencedirect.com/science/article/pii/S1877050917328557> (cit. on p. 16).
- [40] N. Lobontiu. "1.7 Linear And Nonlinear Dynamic Systems". In: *System dynamics for engineering students: concepts and applications*. Academic Press, an imprint of Elsevier, 2018 (cit. on p. 16).
- [41] J. Brownlee. *Deep Learning for Time Series Forecasting Predict the Future with MLPs, CNNs and LSTMs in Python*. Jason Brownlee, 2018 (cit. on pp. 16, 24).
- [42] A. Azzouni and G. Pujolle. "NeuTM: A neural network-based framework for traffic matrix prediction in SDN". In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. 2018, pp. 1–5. DOI: [10.1109/NOMS.2018.8406199](https://doi.org/10.1109/NOMS.2018.8406199) (cit. on p. 17).
- [43] A. Lazaris and V. K. Prasanna. "Deep Learning Models For Aggregated Network Traffic Prediction". In: *2019 15th International Conference on Network and Service Management (CNSM)*. 2019, pp. 1–5. DOI: [10.23919/CNSM46954.2019.9012669](https://doi.org/10.23919/CNSM46954.2019.9012669) (cit. on p. 17).
- [44] N. Ramakrishnan and T. Soni. "Network Traffic Prediction Using Recurrent Neural Networks". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 187–193. DOI: [10.1109/ICMLA.2018.00035](https://doi.org/10.1109/ICMLA.2018.00035) (cit. on p. 17).
- [45] K. Rusek et al. "Unveiling the Potential of Graph Neural Networks for Network Modeling and Optimization in SDN". In: *Proceedings of the 2019 ACM Symposium on SDN Research*. SOSR '19. San Jose, CA, USA: Association for Computing Machinery, 2019, pp. 140–151. ISBN: 9781450367103. DOI: [10.1145/3314148.3314357](https://doi.org/10.1145/3314148.3314357). URL: <https://doi.org/10.1145/3314148.3314357> (cit. on p. 17).

- [46] K. Rusek et al. “RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN”. In: *IEEE Journal on Selected Areas in Communications* 38.10 (2020), pp. 2260–2270. DOI: [10.1109/JSAC.2020.3000405](https://doi.org/10.1109/JSAC.2020.3000405) (cit. on p. 17).
- [47] S. Tomovic et al. “A new approach to dynamic routing in SDN networks”. In: *2016 18th Mediterranean Electrotechnical Conference (MELECON)*. 2016, pp. 1–6. DOI: [10.1109/MELCON.2016.7495433](https://doi.org/10.1109/MELCON.2016.7495433) (cit. on p. 18).
- [48] S. Tomovic et al. “A new approach to dynamic routing in SDN networks”. In: *2016 18th Mediterranean Electrotechnical Conference (MELECON)*. 2016, pp. 1–6. DOI: [10.1109/MELCON.2016.7495433](https://doi.org/10.1109/MELCON.2016.7495433) (cit. on p. 18).
- [49] E. Akin and T. Korkmaz. “Comparison of Routing Algorithms With Static and Dynamic Link Cost in Software Defined Networking (SDN)”. In: *IEEE Access* 7 (2019), pp. 148629–148644. DOI: [10.1109/ACCESS.2019.2946707](https://doi.org/10.1109/ACCESS.2019.2946707) (cit. on pp. 18, 50, 74).
- [50] M. Beshley et al. “Adaptive flow routing model in SDN”. In: *2017 14th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*. 2017, pp. 298–302. DOI: [10.1109/CADSM.2017.7916140](https://doi.org/10.1109/CADSM.2017.7916140) (cit. on pp. 20, 36).
- [51] S. Tomovic et al. “A new approach to dynamic routing in SDN networks”. In: Apr. 2016, pp. 1–6. DOI: [10.1109/MELCON.2016.7495433](https://doi.org/10.1109/MELCON.2016.7495433) (cit. on p. 20).
- [52] M. Huang et al. “Dynamic routing for network throughput maximization in software-defined networks”. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016, pp. 1–9. DOI: [10.1109/INFOCOM.2016.7524613](https://doi.org/10.1109/INFOCOM.2016.7524613) (cit. on p. 20).
- [53] X. Huang et al. “Deep Reinforcement Learning for Multimedia Traffic Control in Software Defined Networking”. In: *IEEE Network* 32.6 (2018), pp. 35–41. DOI: [10.1109/MNET.2018.1800097](https://doi.org/10.1109/MNET.2018.1800097) (cit. on pp. 21, 31, 33, 34, 36).
- [54] J. Xie et al. “A Survey of Machine Learning Techniques Applied to Software Defined Networking (SDN): Research Issues and Challenges”. In: *IEEE Communications Surveys Tutorials* 21.1 (2019), pp. 393–430. DOI: [10.1109/COMST.2018.2866942](https://doi.org/10.1109/COMST.2018.2866942) (cit. on pp. 21, 27).
- [55] X.-D. Zhang. “Machine Learning”. In: *A Matrix Algebra Approach to Artificial Intelligence* (2020), pp. 223–440. DOI: [10.1007/978-981-15-2770-8_6](https://doi.org/10.1007/978-981-15-2770-8_6) (cit. on p. 21).
- [56] *Framing: Key ML Terminology | Machine Learning Crash Course*. Accessed December 28, 2020. URL: <https://developers.google.com/machine-learning/crash-course/framing/ml-terminology> (cit. on p. 21).
- [57] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. The MIT Press, 2018 (cit. on p. 21).

- [58] Z. M. Fadlullah et al. “State-of-the-Art Deep Learning: Evolving Machine Intelligence Toward Tomorrow’s Intelligent Network Traffic Control Systems”. In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2432–2455. DOI: [10.1109/COMST.2017.2707140](https://doi.org/10.1109/COMST.2017.2707140) (cit. on p. 22).
- [59] Q. Mao, F. Hu, and Q. Hao. “Deep Learning for Intelligent Wireless Networks: A Comprehensive Survey”. In: *IEEE Communications Surveys Tutorials* 20.4 (2018), pp. 2595–2621. DOI: [10.1109/COMST.2018.2846401](https://doi.org/10.1109/COMST.2018.2846401) (cit. on p. 22).
- [60] *What are Neural Networks?* Accessed December 27, 2020. URL: <https://www.ibm.com/cloud/learn/neural-networks> (cit. on pp. 22–24).
- [61] DeepAI. *Neural Network*. Accessed December 27, 2020. May 2019. URL: <https://deepai.org/machine-learning-glossary-and-terms/neural-network> (cit. on pp. 22, 23).
- [62] J. B. Ahire. *The Artificial Neural Networks handbook: Part 1*. Accessed December 27, 2020. Sept. 2020. URL: <https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4> (cit. on p. 23).
- [63] *What Is a Neural Network?* Accessed December 27, 2020. URL: <https://www.mathworks.com/discovery/neural-network.html> (cit. on p. 23).
- [64] Dansbecker. *Rectified Linear Units (ReLU) in Deep Learning*. May 2018. URL: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning> (cit. on p. 24).
- [65] A. Mozo, B. Ordozgoiti, and S. Gómez-Canaval. “Forecasting short-term data center network traffic load with convolutional neural networks”. In: *Plos One* 13.2 (2018). DOI: [10.1371/journal.pone.0191939](https://doi.org/10.1371/journal.pone.0191939) (cit. on p. 24).
- [66] S. Hosein and P. Hosein. “Load forecasting using deep neural networks”. In: *2017 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT) 2017*, pp. 1–5. DOI: [10.1109/ISGT.2017.8085971](https://doi.org/10.1109/ISGT.2017.8085971) (cit. on p. 25).
- [67] W. He. “Load Forecasting via Deep Neural Networks”. In: *Procedia Computer Science* 122 (2017). 5th International Conference on Information Technology and Quantitative Management, ITQM 2017, pp. 308–314. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2017.11.374>. URL: <http://www.sciencedirect.com/science/article/pii/S1877050917326170> (cit. on p. 25).
- [68] J. Kumar, R. Goomer, and A. K. Singh. “Long Short Term Memory Recurrent Neural Network (LSTM-RNN) Based Workload Forecasting Model For Cloud Datacenters”. In: *Procedia Computer Science* 125 (2018). The 6th International Conference on Smart Computing and Communications, pp. 676–682. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2017.12.087>. URL: <http://www.sciencedirect.com/science/article/pii/S1877050917328557> (cit. on p. 25).

- [69] A. A. Zai and B. Brown. *Deep Reinforcement Learning in Action*. Manning Publications Company, 2020 (cit. on pp. 25–30, 61).
- [70] Z. Xu et al. “Experience-driven Networking: A Deep Reinforcement Learning based Approach”. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 1871–1879. DOI: [10.1109/INFOCOM.2018.8485853](https://doi.org/10.1109/INFOCOM.2018.8485853) (cit. on pp. 27, 30, 31, 34–36).
- [71] N. C. Luong et al. “Applications of Deep Reinforcement Learning in Communications and Networking: A Survey”. In: *IEEE Communications Surveys Tutorials* 21.4 (2019), pp. 3133–3174. DOI: [10.1109/COMST.2019.2916583](https://doi.org/10.1109/COMST.2019.2916583) (cit. on pp. 27, 28).
- [72] N. C. Luong et al. “Applications of Deep Reinforcement Learning in Communications and Networking: A Survey”. In: *IEEE Communications Surveys Tutorials* 21.4 (2019), pp. 3133–3174. DOI: [10.1109/COMST.2019.2916583](https://doi.org/10.1109/COMST.2019.2916583) (cit. on pp. 29, 44, 45).
- [73] C. Yu et al. “DROM: Optimizing the Routing in Software-Defined Networks With Deep Reinforcement Learning”. In: *IEEE Access* 6 (2018), pp. 64533–64539. DOI: [10.1109/ACCESS.2018.2877686](https://doi.org/10.1109/ACCESS.2018.2877686) (cit. on pp. 31, 33, 36, 45).
- [74] Y.-R. Chen et al. “RL-Routing: An SDN Routing Algorithm Based on Deep Reinforcement Learning”. In: *IEEE Transactions on Network Science and Engineering* 7.4 (2020), pp. 3185–3199. DOI: [10.1109/TNSE.2020.3017751](https://doi.org/10.1109/TNSE.2020.3017751) (cit. on pp. 31, 36).
- [75] P. Sun et al. “A Scalable Deep Reinforcement Learning Approach for Traffic Engineering Based on Link Control”. In: *IEEE Communications Letters* 25.1 (2021), pp. 171–175. DOI: [10.1109/LCOMM.2020.3022064](https://doi.org/10.1109/LCOMM.2020.3022064) (cit. on pp. 31, 35).
- [76] Nsnam. 3. URL: <https://www.nsnam.org/> (cit. on p. 34).
- [77] S. Communications. *Dataminer documentation* (cit. on pp. 38–43, 53).
- [78] D. Harrington, R. Presuhn, and B. Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. STD 62. <http://www.rfc-editor.org/rfc/rfc3411.txt>. RFC Editor, Dec. 2002. URL: <http://www.rfc-editor.org/rfc/rfc3411.txt> (cit. on pp. 40, 41).
- [79] *Welcome to RYU the network Operating system (NOS)*. URL: <https://ryu.readthedocs.io/en/latest/index.html> (cit. on pp. 43, 44).
- [80] V. GUEANT. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP Test the limits of your network Internet neutrality test*. URL: <https://iperf.fr/> (cit. on p. 44).
- [81] Mininet. *Documentation · mininet/mininet wiki*. URL: <https://github.com/mininet/mininet/wiki/Documentation> (cit. on p. 44).
- [82] K. Arulkumaran et al. “Deep Reinforcement Learning: A Brief Survey”. In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: [10.1109/MSP.2017.2743240](https://doi.org/10.1109/MSP.2017.2743240) (cit. on pp. 44, 45).

-
- [83] *Production Quality, Multilayer Open Virtual Switch*. URL: <http://www.openvswitch.org/> (cit. on p. 56).
- [84] F. Schur. *DDQN with PyTorch for OpenAI Gym*. <https://github.com/fschur/DDQN-with-PyTorch-for-OpenAI-Gym>. 2020 (cit. on p. 61).
- [85] gouxiangchen. *Dueling-DQN-pytorch*. <https://github.com/gouxiangchen/dueling-DQN-pytorch>. 2020 (cit. on p. 61).
- [86] P. Mary Mammen and H. Kumar. *Explainable AI: Deep Reinforcement Learning Agents for Residential Demand Side Cost Savings in Smart Grids*. Oct. 2019 (cit. on p. 62).
- [87] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 63).
- [88] G. Sahin and S. Subramaniam. “Providing Quality-of-Protection Classes Through Control-Message Scheduling in DWDM Mesh Networks With Capacity Sharing”. In: *Selected Areas in Communications, IEEE Journal on* 22 (Dec. 2004), pp. 1846–1858. DOI: 10.1109/JSAC.2004.833834 (cit. on p. 65).
- [89] OpenAI. *Part 2: Kinds of RL Algorithms*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (cit. on p. 81).
- [90] V. D. Markova and V. K. Shopov. “Knowledge Transfer in Reinforcement Learning Agent”. In: *2019 International Conference on Information Technologies (InfoTech)*. 2019, pp. 1–4. DOI: 10.1109/InfoTech.2019.8860881 (cit. on p. 81).

ANNEX 1 - LINK COST OPTIMIZATION ALGORITHMS' PERFORMANCE

Table I.1: Comparison of link cost optimization algorithms in 32 TCP flow requests.

Request	MHA (Dijkstra)		DSP		LIOA	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H7	13.10	0.320	14.30	0.397	13.54	0.390
H1 ->H8	7.63	0.414	11.30	0.494	15.00	0.282
H1 ->H9	12.00	0.208	12.30	0.286	10.00	0.237
H1 ->H11	13.53	0.226	12.00	0.292	11.87	0.339
H1 ->H12	9.13	0.333	11.24	0.375	7.91	0.413
H1 ->H13	9.34	0.317	10.83	0.370	8.91	0.300
H2 ->H7	10.89	0.282	14.50	0.279	9.36	0.454
H2 ->H8	3.65	0.418	7.59	0.424	10.14	0.296
H2 ->H9	5.69	0.395	7.33	0.401	12.16	0.254
H2 ->H10	9.79	0.432	5.24	0.445	13.90	0.341
H2 ->H11	12.85	0.180	6.27	0.366	10.80	0.265
H2 ->H12	10.05	0.286	10.96	0.285	9.60	0.324
H2 ->H13	9.03	0.381	6.17	0.382	11.25	0.248
H3 ->H7	8.80	0.395	12.10	0.407	13.90	0.397
H3 ->H8	12.00	0.370	11.20	0.382	9.59	0.391
H3 ->H9	8.77	0.366	9.02	0.366	7.63	0.438
H3 ->H10	10.10	0.392	9.89	0.371	11.20	0.327
H3 ->H11	13.48	0.356	11.73	0.353	7.94	0.297
H3 ->H12	14.99	0.397	15.00	0.372	15.00	0.319
H3 ->H13	9.49	0.392	11.00	0.382	6.60	0.441
H4 ->H6	14.99	0.192	15.00	0.221	15.00	0.152
H4 ->H7	5.70	0.387	7.96	0.416	6.36	0.462
H4 ->H8	6.53	0.343	6.00	0.339	12.21	0.320
H4 ->H9	6.51	0.413	5.76	0.414	7.70	0.417
H4 ->H10	6.35	0.424	7.43	0.429	10.00	0.313
H4 ->H11	7.30	0.407	8.97	0.407	10.26	0.297
H4 ->H12	3.53	0.471	7.50	0.443	5.31	0.466
H4 ->H13	4.39	0.427	7.54	0.443	10.00	0.316
H5 ->H6	14.97	0.049	15.00	0.063	15.00	0.039
H5 ->H8	14.99	0.161	15.00	0.140	15.00	0.135
H5 ->H11	14.99	0.225	15.00	0.192	15.00	0.225
H7 ->H6	14.99	0.218	15.00	0.219	15.00	0.161
Average:	9.99	0.331	10.50	0.349	11.04	0.314
Uncongested:	6		6		7	

ANNEX 2 - DRL AGENTS' PERFORMANCE

Table II.1: Performance comparison between DRL agents in 32 TCP flow requests (setup 1, ARPANET).

Request	DQN		DDQN		Dueling DQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H7	14.77	0.314	14.96	0.267	14.62	0.290
H1 ->H8	9.55	0.348	12.99	0.296	9.69	0.346
H1 ->H9	14.71	0.170	11.78	0.182	12.10	0.266
H1 ->H11	13.69	0.245	14.58	0.190	13.52	0.181
H1 ->H12	14.76	0.261	9.48	0.313	11.45	0.290
H1 ->H13	13.39	0.257	9.53	0.295	11.14	0.283
H2 ->H7	10.69	0.257	14.77	0.244	9.26	0.307
H2 ->H8	6.41	0.328	6.89	0.320	11.86	0.245
H2 ->H9	7.11	0.316	7.33	0.308	8.15	0.299
H2 ->H10	9.24	0.314	9.76	0.307	14.58	0.311
H2 ->H11	13.41	0.140	14.69	0.112	13.37	0.182
H2 ->H12	4.80	0.425	10.17	0.277	11.07	0.262
H2 ->H13	7.69	0.301	8.27	0.298	7.80	0.285
H3 ->H7	14.77	0.317	14.87	0.322	7.63	0.500
H3 ->H8	14.28	0.253	15.00	0.165	9.09	0.306
H3 ->H9	7.08	0.422	9.39	0.310	10.42	0.291
H3 ->H10	13.65	0.311	8.37	0.338	10.54	0.318
H3 ->H11	8.79	0.319	14.87	0.273	10.02	0.315
H3 ->H12	15.00	0.284	14.99	0.325	15.00	0.290
H3 ->H13	10.97	0.349	13.94	0.323	14.78	0.322
H4 ->H6	15.00	0.173	15.00	0.194	15.00	0.151
H4 ->H7	6.12	0.338	5.12	0.365	9.89	0.346
H4 ->H8	7.49	0.269	10.24	0.323	8.86	0.328
H4 ->H9	8.15	0.329	8.53	0.313	9.58	0.265
H4 ->H10	7.03	0.323	8.27	0.314	4.45	0.445
H4 ->H11	8.04	0.322	8.90	0.312	8.43	0.346
H4 ->H12	5.15	0.427	3.73	0.444	4.98	0.431
H4 ->H13	5.11	0.399	7.96	0.317	9.71	0.314
H5 ->H6	15.00	0.025	15.00	0.030	15.00	0.052
H5 ->H8	15.00	0.048	15.00	0.205	15.00	0.099
H5 ->H11	15.00	0.141	15.00	0.238	15.00	0.118
H7 ->H6	15.00	0.146	15.00	0.043	15.00	0.147
Average:	10.84	0.277	11.39	0.268	11.16	0.279
Uncongested:	10		13		9	

Table II.2: Performance comparison between DRL agents in 32 TCP flow requests (setup 2, ARPANET).

Request	DQN		DDQN		Dueling DQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H7	12.26	0.300	15.00	0.246	4.23	0.496
H1 ->H8	10.57	0.370	12.32	0.331	15.00	0.283
H1 ->H9	13.20	0.176	11.27	0.192	10.95	0.242
H1 ->H11	13.03	0.203	14.50	0.226	13.16	0.173
H1 ->H12	12.42	0.314	7.98	0.314	9.40	0.351
H1 ->H13	11.34	0.296	8.23	0.300	12.39	0.279
H2 ->H7	12.52	0.255	14.17	0.280	4.71	0.436
H2 ->H8	5.82	0.375	9.18	0.315	9.40	0.321
H2 ->H9	7.99	0.353	6.81	0.383	10.64	0.305
H2 ->H10	9.12	0.385	8.61	0.366	10.24	0.333
H2 ->H11	14.50	0.146	14.68	0.189	14.50	0.138
H2 ->H12	11.72	0.260	9.94	0.290	7.98	0.353
H2 ->H13	7.66	0.341	7.59	0.368	9.26	0.294
H3 ->H7	11.90	0.408	9.71	0.322	12.29	0.304
H3 ->H8	14.59	0.322	13.86	0.295	11.22	0.226
H3 ->H9	11.20	0.366	8.80	0.297	8.10	0.377
H3 ->H10	12.54	0.407	8.93	0.313	9.68	0.387
H3 ->H11	9.59	0.372	15.00	0.273	10.73	0.302
H3 ->H12	15.00	0.376	14.61	0.322	15.00	0.184
H3 ->H13	9.19	0.411	7.20	0.317	12.38	0.312
H4 ->H6	15.00	0.169	15.00	0.105	14.71	0.297
H4 ->H7	6.65	0.336	11.24	0.342	11.51	0.328
H4 ->H8	8.00	0.302	8.18	0.320	9.65	0.253
H4 ->H9	7.81	0.369	6.25	0.382	8.20	0.315
H4 ->H10	6.47	0.369	6.97	0.394	6.52	0.372
H4 ->H11	7.32	0.361	7.40	0.378	9.06	0.307
H4 ->H12	4.71	0.428	6.65	0.322	8.80	0.350
H4 ->H13	6.90	0.373	6.96	0.390	11.25	0.267
H5 ->H6	15.00	0.040	9.29	0.318	15.00	0.054
H5 ->H8	15.00	0.100	15.00	0.146	15.00	0.104
H5 ->H11	15.00	0.193	15.00	0.119	15.00	0.203
H7 ->H6	15.00	0.191	15.00	0.144	15.00	0.216
Average:	10.87	0.302	10.67	0.291	10.97	0.286
Uncongested:	8		9		8	

Table II.3: Performance comparison between DRL agents in 32 TCP flow requests (setup 3, ARPANET).

Request	DQN		DDQN		Dueling DQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H7	6.91	0.312	15.00	0.266	12.43	0.297
H1 ->H8	9.05	0.326	9.59	0.411	10.09	0.367
H1 ->H9	10.98	0.209	13.41	0.192	10.92	0.360
H1 ->H11	15.00	0.034	14.60	0.210	8.90	0.431
H1 ->H12	6.36	0.308	9.02	0.325	9.50	0.362
H1 ->H13	6.60	0.307	11.06	0.313	8.26	0.414
H2 ->H7	10.58	0.292	14.70	0.277	7.91	0.396
H2 ->H8	7.39	0.311	5.22	0.430	9.00	0.416
H2 ->H9	12.71	0.169	8.45	0.376	9.20	0.366
H2 ->H10	7.69	0.329	8.23	0.407	8.51	0.363
H2 ->H11	9.79	0.340	14.87	0.136	12.22	0.271
H2 ->H12	2.66	0.527	4.48	0.370	14.87	0.352
H2 ->H13	12.69	0.163	7.44	0.386	8.32	0.333
H3 ->H7	15.00	0.024	12.02	0.411	10.97	0.415
H3 ->H8	9.23	0.329	14.97	0.248	12.39	0.339
H3 ->H9	5.80	0.306	6.85	0.435	8.70	0.440
H3 ->H10	6.62	0.311	7.84	0.461	7.64	0.443
H3 ->H11	7.67	0.290	12.36	0.383	12.80	0.277
H3 ->H12	14.58	0.318	12.04	0.274	14.93	0.421
H3 ->H13	7.15	0.308	9.74	0.303	14.73	0.294
H4 ->H6	15.00	0.042	15.00	0.233	15.00	0.256
H4 ->H7	15.00	0.040	11.78	0.375	9.00	0.324
H4 ->H8	7.73	0.353	7.89	0.303	9.66	0.312
H4 ->H9	7.26	0.302	5.79	0.406	13.50	0.213
H4 ->H10	4.47	0.502	5.86	0.416	5.09	0.480
H4 ->H11	8.61	0.292	7.36	0.402	9.40	0.365
H4 ->H12	9.91	0.318	10.72	0.396	11.35	0.373
H4 ->H13	7.34	0.329	6.23	0.422	6.19	0.428
H5 ->H6	7.63	0.311	14.99	0.131	10.65	0.251
H5 ->H8	12.96	0.045	15.00	0.265	15.00	0.210
H5 ->H11	15.00	0.315	9.41	0.330	15.00	0.236
H7 ->H6	15.00	0.026	15.00	0.214	15.00	0.214
Average:	9.70	0.262	10.53	0.328	10.85	0.344
Uncongested:	7		9		7	

Table II.4: Performance comparison between DRL agents in 32 TCP flow requests (setup 4, ARPANET).

Request	DQN		DDQN		Dueling DQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H7	11.28	0.350	11.20	0.347	14.56	0.292
H1 ->H8	8.75	0.448	14.62	0.274	9.06	0.398
H1 ->H9	10.24	0.335	7.65	0.339	14.20	0.175
H1 ->H11	10.47	0.353	10.38	0.235	12.74	0.207
H1 ->H12	13.08	0.347	8.75	0.392	11.71	0.295
H1 ->H13	4.63	0.487	8.81	0.334	10.96	0.281
H2 ->H7	14.60	0.273	8.07	0.367	7.71	0.360
H2 ->H8	7.77	0.393	7.01	0.442	4.88	0.415
H2 ->H9	7.82	0.389	6.67	0.407	7.17	0.389
H2 ->H10	9.87	0.339	11.16	0.360	5.57	0.395
H2 ->H11	11.02	0.327	14.61	0.173	14.69	0.144
H2 ->H12	11.18	0.347	12.07	0.399	11.00	0.267
H2 ->H13	9.12	0.373	8.72	0.296	7.34	0.379
H3 ->H7	10.30	0.361	9.75	0.354	10.55	0.386
H3 ->H8	11.51	0.307	10.48	0.257	10.76	0.363
H3 ->H9	5.70	0.456	9.26	0.345	8.84	0.348
H3 ->H10	6.11	0.473	9.15	0.361	10.15	0.373
H3 ->H11	9.67	0.331	6.56	0.390	10.89	0.341
H3 ->H12	15.00	0.256	15.00	0.325	15.00	0.354
H3 ->H13	10.76	0.370	10.18	0.374	13.27	0.393
H4 ->H6	9.65	0.387	15.00	0.301	15.00	0.145
H4 ->H7	11.36	0.300	10.44	0.372	8.11	0.324
H4 ->H8	8.98	0.335	8.55	0.258	6.96	0.338
H4 ->H9	6.49	0.391	9.09	0.352	5.32	0.402
H4 ->H10	10.34	0.424	6.30	0.482	5.28	0.405
H4 ->H11	6.22	0.387	7.41	0.359	8.87	0.396
H4 ->H12	5.08	0.445	15.00	0.362	10.19	0.379
H4 ->H13	6.49	0.394	11.54	0.355	5.81	0.414
H5 ->H6	15.00	0.153	15.00	0.360	15.00	0.070
H5 ->H8	15.00	0.166	15.00	0.092	15.00	0.141
H5 ->H11	15.00	0.141	15.00	0.109	15.00	0.175
H7 ->H6	15.00	0.155	15.00	0.108	15.00	0.143
Average:	10.11	0.344	10.73	0.321	10.52	0.309
Uncongested:	6		9		8	

Table II.5: Performance comparison between Dijkstra and DDQN in unseen scenario settings (setup 1, ARPANET).

Request	DQN		DDQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H6	7.32	0.355	9.52	0.301
H1 ->H8	8.05	0.376	6.48	0.407
H1 ->H9	10.31	0.350	12.74	0.348
H1 ->H10	8.50	0.421	10.70	0.456
H1 ->H11	14.17	0.385	9.46	0.374
H1 ->H12	10.19	0.376	9.96	0.371
H1 ->H13	13.07	0.354	15.00	0.359
H2 ->H6	8.64	0.320	9.81	0.325
H2 ->H8	7.21	0.347	7.94	0.353
H2 ->H9	9.07	0.390	14.73	0.384
H2 ->H10	8.93	0.408	9.93	0.403
H2 ->H11	9.57	0.392	6.60	0.337
H2 ->H12	9.00	0.421	9.02	0.376
H2 ->H13	14.98	0.218	15.00	0.402
Average:	9.93	0.365	10.49	0.371
Uncongested:	1		3	

Table II.6: Performance comparison between Dijkstra and the three agents (setup 1, Network RW).

Request	Dijkstra		DQN		DDQN		Dueling DQN	
	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)	Bitrate (Mbits/s)	RTT (s)
H1 ->H6	17.27	0.232	19.35	0.070	19.99	0.050	19.92	0.05
H1 ->H7	18.60	0.446	19.99	0.297	19.99	0.079	19.99	0.12
H1 ->H8	16.48	0.474	19.99	0.173	18.96	0.257	20.00	0.12
H1 ->H9	14.43	0.408	19.09	0.083	19.03	0.233	18.91	0.24
H1 ->H10	13.42	0.418	19.99	0.136	19.99	0.062	19.99	0.12
H2 ->H6	13.31	0.362	19.98	0.161	17.61	0.209	19.59	0.12
H2 ->H7	12.72	0.379	16.48	0.266	17.05	0.234	19.51	0.14
H2 ->H8	17.32	0.447	16.99	0.288	19.99	0.238	19.99	0.13
H2 ->H10	13.45	0.344	18.20	0.196	18.19	0.200	18.64	0.15
H3 ->H6	10.80	0.415	18.86	0.194	19.99	0.116	19.99	0.16
H3 ->H7	13.04	0.396	14.77	0.295	19.99	0.138	19.99	0.17
H3 ->H8	19.03	0.486	19.99	0.252	19.98	0.325	19.99	0.23
H3 ->H9	11.04	0.491	18.45	0.171	15.08	0.286	16.84	0.31
H3 ->H10	14.25	0.475	19.99	0.215	19.99	0.245	19.99	0.19
H4 ->H6	13.95	0.348	17.24	0.153	19.92	0.099	18.68	0.13
H4 ->H7	11.93	0.415	16.94	0.275	19.99	0.085	19.82	0.17
H4 ->H8	9.65	0.461	19.99	0.196	16.55	0.238	18.29	0.20
H4 ->H9	11.10	0.482	18.53	0.120	18.24	0.243	16.63	0.27
H4 ->H10	19.99	0.390	19.98	0.212	19.99	0.241	19.99	0.16
H5 ->H6	16.71	0.247	18.81	0.099	19.89	0.057	19.53	0.09
H5 ->H7	19.62	0.455	19.99	0.248	19.99	0.080	19.99	0.10
H5 ->H8	18.39	0.459	19.99	0.211	19.99	0.292	16.99	0.18
H5 ->H9	16.04	0.428	19.57	0.121	18.94	0.266	17.55	0.28
H5 ->H10	13.71	0.423	20.00	0.180	19.63	0.213	19.99	0.16
Average:	14.84	0.412	18.88	0.192	19.12	0.187	19.20	0.166
Uncongested:	2		12		15		16	