UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ISADORA PEDRINI POSSEBON

Look-Ahead Reinforcement Learning: an application for load balancing network traffic

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Alberto Egon Schaeffer-Filho

Porto Alegre
August 2021

CIP — CATALOGING-IN-PUBLICATION

Possebon, Isadora Pedrini

Look-Ahead Reinforcement Learning: an application for load balancing network traffic / Isadora Pedrini Possebon. – Porto Alegre: PPGC da UFRGS, 2021.

96 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Alberto Egon Schaeffer-Filho.

1. Network traffic. 2. Reinforcement learning. 3. Network traffic prediction. 4. Load balancing. 5. Network flow. 6. Machine learning. I. Schaeffer-Filho, Alberto Egon. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof[a]. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Enquanto não alcances Não descanses, De nenhum fruto queiras só metade.”

— MIGUEL TORGA

ACKNOWLEDGEMENTS

To my advisor, Prof. Dr. Alberto Egon Schaeffer-Filho for the guidance that was vital to the conclusion of this work. I am grateful for your patience and the time you invested in this project, for the motivation and passion for science you inspire.

To my parents, - Agradeço aos meus pais, principalmente, por todo o esforço e dedicação que tiveram comigo e na minha educação ao longo desses 26 anos. Vocês me motivaram a ir longe e, mais do que isso, me proporcionaram a possibilidade de ir longe. Tenham a certeza de que o resultado desse trabalho é fruto de um sonho que vocês me ajudaram a sonhar e a colocar em prática. Sem vocês, nada disso seria possível. Essa vitória é nossa!

To my sister and brother for all these years of sharing and compassion. You were always the people who understood and helped me get through difficult times. I appreciate having you not only as siblings but also as part of my life. Together we can do remarkable things.

To Guilherme, that started this journey with me as my boyfriend and is now my fiance. I thank you for all your patience and companionship. Thank you for the technical discussions, science critiques, and wine shared while working on our projects. You are the best science/life partner I could ever ask for.

To my friends, Amanda e Victória, who helped me with technical discussions and motivational speeches since our first year as undergraduate students at UFRGS. It is an honor to share these experiences and knowledge with such inspirational women.

Finally, I am grateful for all the teachers that I had at Instituto de Informática and at UFRGS. Thanks for sharing your knowledge and inspiring me to always search for answers. You helped me accomplish a dream I had since I was 13 years old.

ABSTRACT

Considering the growth in complexity and scale of computer networks and that the leading cause of failures is human error, there is an increasing interest in minimizing the role of humans in network management tasks. In this context, we propose a two-step, machine learning approach for automatically balancing network flows that can compromise network performance. In particular, firstly, we rely on identifying elephant flows, which more heavily impact network resources. Secondly, we use a reinforcement learning mechanism to determine the best action to be performed in the network, given its current status. The intuition for this two-step approach is to amortize the computational costs of reinforcement learning and apply it only to flows which can cause a high impact on network performance.

To evaluate our work, we firstly perform a functional evaluation to discuss different reward functions for load balancing using reinforcement learning. Secondly, we evaluate the elephant flow identification, discussing the impact of looking to elephant flows on reinforcement learning strategies.

For the first set of experiments, results indicate that the RL approach is better than the baseline (controller with no RL intervention). The reward function with better results used a harmonic mean heuristic. This reward function was able to reduce FCT and be scalable concerning the number of switches. For the second set of experiments, we showed the importance of using an elephant flow intelligence: reward function with this factor was able to reduce FCT by 91%, considering a 50/50 workload (50% mice flows, and a 50% elephant flows proportion, with a 15 seconds interval between connections).

Our main contributions are (i) problem modeling as a function of states and actions in a system that aims to balance network traffic and (ii) an architecture that more judiciously uses reinforcement learning on flows of interest for load balancing.

Keywords: Network traffic. reinforcement learning. network traffic prediction. load balancing. network flow. machine learning.

Look-Ahead Reinforcement Learning: uma aplicação para balanceamento de fluxos de rede usando aprendizado por reforço

RESUMO

Considerando o crescimento de complexidade e escala das redes de computadores e que a principal causa de falhas é o erro humano, há um interesse crescente em minimizar o papel dos humanos nas tarefas de gerenciamento de rede. Nesse contexto, propomos uma abordagem de aprendizado de máquina em duas etapas para balancear automaticamente os fluxos de rede que podem comprometer o desempenho da rede. Em primeiro lugar, contamos com a identificação de fluxos de elefantes, que impactam mais fortemente os recursos da rede. Em segundo lugar, utilizamos um mecanismo de aprendizagem por reforço para determinar a melhor ação a ser realizada na rede, dado o seu estado atual. A intuição para esta abordagem em duas etapas é amortizar os custos computacionais do aprendizado por reforço e aplicá-los apenas aos fluxos que podem causar um alto impacto no desempenho da rede.

Para avaliar nosso trabalho, primeiramente fazemos uma avaliação funcional para discutir diferentes funções de recompensa usadas no balanceamento de carga com aprendizagem por reforço. Em segundo lugar, avaliamos a identificação de fluxos elefante, discutindo o impacto de observar esse tipo de fluxo nas estratégias de aprendizagem por reforço.

Para o primeiro conjunto de experimentos, os resultados indicam que a abordagem RL é melhor do que a solução de base (controlador sem RL). A função de recompensa com melhores resultados utilizou uma heurística de média harmônica e foi capaz de reduzir o FCT, sendo escalável em relação ao número de switches na topologia. Para o segundo conjunto de experimentos, mostramos a importância de usar uma inteligência de fluxos elefantes: a função de recompensa com esse fator foi capaz de reduzir o FCT em 91 %, considerando uma carga de trabalho de 50/50 (50 % de fluxos ratos e 50 % de fluxos elefantes, com intervalo de 15 segundos entre as conexões).

Nossas principais contribuições são (i) modelagem de problemas em função de estados e ações em um sistema que visa balancear o tráfego da rede e (ii) uma arquitetura que usa de forma mais criteriosa a aprendizagem por reforço nos fluxos de interesse para o balanceamento de carga.

Palavras-chave: tráfego de rede, aprendizado por reforço, previsão de tráfego de rede, balanceamento de carga, fluxo de rede, aprendizado de máquina.

LIST OF FIGURES

Figure 2.1 Diagram illustrating how the agent interacts with the environment.............21
Figure 2.2 Diagram illustrating the difference between Q-Learning and Deep Q-Learning. 24

Figure 4.1 Look-Ahead Reinforcement Learning for load balancing network traffic approach overview........38
Figure 4.2 Example network topology used to model our Reinforcement Learning agent. 39
Figure 4.3 Network topology for state when H1 routes workload to H2 through a, b, f and i links. 40
Figure 4.4 Reinforcement Learning agent steps to select the best action. The agent iterates over each step to find the best actions. The chosen action is the one highlighted. 43
Figure 4.5 Diagram showing how Deep Q-Learning approximates Q-value for the actions based on the current state........44
Figure 4.6 Look-Ahead Reinforcement Learning for Load Balancing architecture.......45
Figure 4.7 Output generated by Flow Action Translator: OpenFlow rule for switch S1 49

Figure 5.1 Topology S1 used as baseline for experimental analysis. H1 is the source host, and H2 is the target. 55
Figure 5.2 Topology S2 used to analyze the impact of more switches on experimental results. H1 is the source host, and H2 is the target. 56
Figure 5.3 Average total flow completion time results for experiments with agents *WeightedUsage*, *UsageHarmonicMean*, *UsageStandardDeviation* and Floodlight Controller. Error bars represent the standard deviation for each set of replications. Each set of experiments considered 5 flows with the same size (10 MB, 50 MB, 100 MB, 200 MB, and 500 MB, respectively)........62
Figure 5.4 Memory usage results for experiments with different reward functions.......64
Figure 5.5 Total flow completion time for experiments using agents *UsageHarmonicMean* and *UsageHarmonicMean-EFI*........66

Figura A.1 Visão geral da arquiteura de Look-Ahead Reinforcement Learning para balanceamento de carga. 82
Figura A.2 Arquitetura Look-Ahead Reinforcement Learning para balanceamento de tráfego de rede........84
Figura A.3 Topologia S1 usada como base para os experimentos. H1 é o host de origem e H2 é o host de destino........86
Figura A.4 Topologia S2 usada para analisar o impacto de switches adicionais no resultado dos experimentos. H1 é o host de origem e H2 é o host de destino.........87
Figura A.5 Média do tempo total de completude dos fluxos em segundos (*average total flow completion time*), em função do tamanho dos fluxos em MBytes (*flow size*). Os resultados são mostrados para os agentes *WeightedUsage*, *UsageHarmonicMean*, *UsageStandardDeviation* e para o controlador Floodlight. As barras de erro representam o desvio padrão de cada conjunto de replicação dos experimentos........91
Figura A.6 Resultados de utilização média de memória em KBytes (*average memory usage*), em função do tamanho dos fluxos em MBytes (*flow size*) utilizando agentes com diferentes funções de recompensa. 92

Figura A.7 Resultados de tempo total para conclusão dos fluxos (FCT) usando os agentes *UsageHarmonicMean* e *UsageHarmonicMean-EFI*. 94

LIST OF TABLES

Table 3.1 Comparison between research efforts discussed in this document. 35

Table 4.1 Representation of snapshot 1, with a 5Mbps flow. 39
Table 4.2 Representation of state 1, generated from snapshot 1 illustrated on Table 4.1...39

Table 4.3 Network statistics stored at each network snapshot, considering an active flow between links *a*, *b*, *f*, and *i*. 46
Table 4.4 Set of parameters set for each switch as the output of Flow Action Translator module. 49

Table 5.1 Set of different parameters used on the model and its correspondent values..57
Table 5.2 Set of different agents trained for the experiments. 60
Table 5.3 Set of different parameters for training the agents. 60
Table 5.4 Experiments configuration for functional analysis..61
Table 5.5 Homogeneous state representation. 63
Table 5.6 Heterogeneous state representation. 63
Table 5.7 Comparison between reward function values for each state. 64
Table 5.8 Experiments configuration for EFI analysis...65
Table 5.9 Workload distribution, where #MF is the number of mice flows, and #EF is the number of elephant flows. 66

Tabela A.1 Conjunto de diferentes parâmetros e seus valores correspondentes. 86
Tabela A.2 Configuração dos experimentos para a análise funcional. Baseline corresponde ao uso do controlador Floodlight sem intervenções de aprendizado de máquina. 90
Tabela A.3 Distribuição da carga de trabalho, onde #FR é o número de fluxos ratos e #FE é o número de fluxos elefantes. 92

LIST OF ABBREVIATIONS AND ACRONYMS

ANN	Artificical Neural Network
AR	Auto-Regressive
BP	Back-propagation
DC	Data Center
DQL	Deep Q-Learning
DQN	Deep Q-Network
ECMP	Equal-Cost Multi-Path
FIB	Forwarding Information Base
FNN	Feedforward Neural Network
GPR	Guassian Process Regression
HMM	Hidden Markov Model
HMM	Hidden Markov Model
ISP	Internet Service Providers
KBR	Kernel Bayes Rule
KNN	K-Nearest Neighbors
LPM	Longest-Prefix Matches
LSTM	Long Short-Term Memory
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
MLP-NN	Multi-Layer Perceptron Neural Network
MSE	Mean Squared-Error
NDN	Named Data Networking
NN	Neural Network
NNE	Neural Network Ensemble

NWS	Network Weather Service
oBMM	Online Bayesian Moment Matching
OSPF	Open Shortest Path First
PIT	Pending Interest Table
QoE	Quality of Experience
QoS	Quality of Service
RIB	Routing Information Base
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RRMSE	Relative RMSE
SARSA	State-Action-Reward-State-Action
SDN	Software-defined Network
SNMP	Simple Network Management Protocol
SVM	Support Vector Machines
TDBA	Tensor-based Deep Belief Architecture
TNR	True Negative Rate
TPR	True Positive Rate
TSF	Time Series Forecasting

CONTENTS

1 INTRODUCTION **14**
1.1 Context **14**
1.2 Motivation **15**
1.3 Objectives and contributions **16**
1.4 Document organization **16**
2 BACKGROUND **17**
2.1 Traffic prediction **17**
2.1.1 Time Series Forecast Problems 18
2.1.2 Non-Time Series Forecast Problems 18
2.2 Load balancing **19**
2.3 Reinforcement Learning **20**
2.3.1 Q-Learning 22
2.3.2 Deep Q-Learning (DQL) 24
2.3.3 On-Policy and Off-Policy Learning 25
3 RELATED WORK **27**
3.1 Machine learning for traffic prediction **27**
3.1.1 Traffic prediction modeled as a Time Series Forecast problem 27
3.1.2 Traffic prediction modeled as a non-Time Series Forecast problem 29
3.2 Traffic engineering **32**
3.2.1 Traffic routing for load balancing 32
3.2.2 Machine learning for traffic routing 33
3.3 Discussion **35**
4 LOOK-AHEAD REINFORCEMENT LEARNING FOR LOAD BALANCING **37**
4.1 Approach Overview **37**
4.2 Reinforcement Learning Agent Modelling **38**
4.2.1 State 39
4.2.2 Reward function 40
4.2.3 Actions 43
4.2.4 Deep Q-Learning Agent 43
4.3 Look-Ahead Reinforcement Learning Architecture **44**
4.3.1 Statistics Manager 45
4.3.2 Elephant Flow Identification 46
4.3.3 Reinforcement Learning Agent 48
4.3.4 Traffic Engineering Rules 48
4.4 Summary **49**
5 PROTOTYPING AND EVALUATION **50**
5.1 Prototyping **50**
5.1.1 Statistics Manager 51
5.1.2 Elephant flow identification 51
5.1.3 Reinforcement Learning Agent 51
5.1.3.1 Step function 51
5.1.3.2 Observation space 52
5.1.3.3 Action space 52
5.1.3.4 Reward function 53
5.1.4 Flow Action Translator 54
5.2 Experiments configuration **55**
5.2.1 Network topologies 55
5.2.2 Environment setup 56

5.2.3 Experiments setup 57
5.3 Functional analysis 61
5.4 Elephant Flow Intelligence (EFI) analysis 65
6 FINAL CONSIDERATIONS AND FUTURE WORK 68
6.1 Final considerations 68
6.2 Limitations and Future work 69
REFERENCES 71
APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS 76
A.1 Introdução 76
A.2 Background 77
A.2.1 Predição de tráfego 78
A.2.2 Balanceamento de carga 78
A.2.3 Aprendizado por reforço 78
A.3 Trabalhos relacionados 79
A.3.1 Previsão de tráfego 79
A.3.2 Balanceamento de carga 79
A.4 Look-Ahead Reinforcement Learning for Load Balancing: abordagem de aprendizado por reforço para balanceamento de carga de tráfego de rede 81
A.4.1 Visão geral 81
A.4.2 Agente de aprendizado por reforço 81
A.5 Prototipação e análise experimental 85
A.5.1 Prototipação 85
A.5.2 Análise experimental 86
A.5.2.1 Avaliação funcional 89
A.5.2.2 Avaliação do fator EFI 91
A.6 Considerações finais e trabalhos futuros 93

1 INTRODUCTION

In this work, we aim to explore network traffic prediction to improve load balancing techniques. In the next sections, we present the following aspects of this work: context, motivation, objectives, contributions, and, finally, the organization of the document that describes it.

1.1 Context

The complexity, heterogeneity, and scale of computer networks have grown beyond the limits of manual administration to such a degree that the leading cause of failures in network environments is human error (DOBSON et al., 2019). Besides, the impact of failures in network environments can be costly, compounded by the delayed reaction and low accuracy of traditional fault tolerance methods. This triggered a change in the design philosophy of network management systems to minimize the role of humans in the control loop. Moreover, there is great diversity in users demands, each with different capacity requirements during different periods (CHEN; HU; MIN, 2019). The dynamic nature of the system and the existence of different requirements make it difficult to allocate resources efficiently, which remains an open challenge (GUO et al., 2017).

In this context, network traffic engineering becomes a crucial task: how to optimize the network performance by dynamically analyzing, predicting, and controlling the behavior of data transmitted over this network? This is the goal behind traffic engineering. This method is used to achieve some operational tasks, such as balancing workload among different paths on the network, increasing the quality of service perceived by the user, and improving fault tolerance.

Routing network traffic is critical and involves selecting the path for transmitting packets. The selection criteria are diverse and depend mainly on the policies and objectives of the operation, such as minimizing costs, maximizing the use of links, and quality of service (QoS) provisioning. In this context, the main challenges are the following: scalability of traffic routing techniques for complex networks, identification of the correlation between the chosen path and the quality of service or perception of improvement provided, and the ability to predict the consequences of a routing decision (BOUTABA et al., 2018). Allied, these points motivate efforts to use machine learning to predict behavior and balance network traffic.

1.2 Motivation

On the one hand, traffic prediction approaches play a crucial role in network operations and management and attempt to anticipate traffic load, volume, packet size, routing, and more. This can help providers optimize network resources (NAREJO; PASERO, 2018). On the other hand, load balancing tries to infer the classification and division of network flows to achieve the best usage of the transmission links. This is typically used to achieve higher transfer rates and lower transmission delays, as well as for reducing adverse effects such as retransmissions (PIZZUTTI; SCHAEFFER-FILHO, 2019).

When devising a load balancing strategy, one must be aware of different types of network flows. For example, elephant flows represent a large (in the number of bytes) and continuous stream of traffic, whereas mice flows tend to be small and short-lived (HAMDAN et al., 2020a). Considering that elephant flows tend to occupy a network path for much longer than mice flows, there is a risk that the number of active flows in some links becomes imbalanced. For instance, simple heuristics that ignore flow size and distribute flows evenly over all equal length paths often lead to congestion, and load balancing heuristics must be used to detect and correct imbalances (POUPART et al., 2016). Furthermore, detecting and preventing network abuse is becoming challenging with a growing amount of traffic and the complexity of networks.

We propose a network traffic load balancing architecture that we call Look-Ahead Reinforcement Learning to address these issues. It combines two main insights. Firstly, the idea of using Reinforcement Learning (RL) to balance workload. This paradigm consists of programming agents based on rewards associated with each action taken by this agent, without the need to specify how to perform a task. Thus, we use a RL agent to identify the more suitable action to provide balanced network usage. Secondly is the idea of adding a previous step before using RL: a phase that indicates whether the agent should interfere in the environment or not.

We advocate that this traffic analysis is essential to ensure that there will be less overhead for the agent's observations. The agent will focus on specific behavior (elephant flows) to balance the workload among the network. Moreover, this extra step could be used to predict the consequences of a routing decision - for instance, avoid splitting small workloads among several paths, reducing the cost of packet reordering. This means that we should prevent the RL agent from being called every time - especially when the network flow is not going to last enough time to compensate for the cost of RL agent

decision-making.

Therefore, the architecture we propose provides load balancing between flows that risk the network's performance. This architecture considers a Software Defined Network (SDN), and is composed of two levels: elephant flow identification and reinforcement learning for load balancing. We use the first to indicate whether it is necessary to act on a specific flow. If so, the system uses the RL architecture to determine the best action for the given network status. We believe this first step can absorb RL calculations' computational cost and latency.

1.3 Objectives and contributions

This work is a RL architecture that aims to balance the network load, providing fewer network failures with less human intervention. We developed this architecture as a module that an SDN controller consults. The experiments were carried out on a software-defined network, using the OpenFlow protocol, evaluating the technique's performance in terms of total flow completion time. We analyzed the method in terms of (i) ability to balance workload, (ii) scalability in terms of switches in the topology, and (iii) impact of using an elephant flow intelligence for reinforcement learning.

Our main contributions are (i) problem modeling as a function of states and actions in a system that aims to balance network traffic, and (ii) an architecture that more judiciously uses reinforcement learning on flows of interest for load balancing. We advocate that this two-step machine learning approach, which we call look-ahead reinforcement learning, can reduce human errors by avoiding unnecessary interactions.

1.4 Document organization

This document is organized as follows: Chapter 2 presents the background concepts and theoretical foundation for this research. Chapter 3 brings the main works related to this topic. Chapter 4 presents details about the proposed architecture. In Chapter 5, we detail the prototype implementation and discuss our experimental evaluation. Finally, Chapter 7 presents the conclusions and final considerations of our work.

2 BACKGROUND

Considering the objectives of this work, we highlight three principal areas: traffic prediction, load balancing, and reinforcement learning. The concept of traffic engineering is related to methods for optimizing network performance through analysis and prediction of network behavior. For this, one could analyze aspects such as bandwidth distribution, network resources, quality of service, load balancing, and packet routing. In our work, we consider traffic engineering in the context of load balancing. In the following sections, we present the main concepts of each area.

2.1 Traffic prediction

Network traffic prediction plays a crucial role in network operations and management for today's increasingly complex and diverse networks (BOUTABA et al., 2018). Advance knowledge of future events in a dynamic system can often be leveraged to improve the system's performance and efficiency. For instance, such knowledge could potentially benefit many problems in data center networks, including routing and flow scheduling, circuit switching, packet scheduling in switch queues, and transport protocols. Indeed, past work on each of the above topics has explored this, and in some cases, claimed significant improvements (ĐUKIĆ et al., 2019)

The predictability of network traffic is essential in many areas, such as dynamic bandwidth allocation, network security, network planning, and predictive congestion control. Network traffic prediction could address several features: available bandwidth and capacity, flow size or volume, flow time, and even application behavior (JOSHI; HADI, 2015).

Measuring available bandwidth and capacity is vital for many types of network applications. Activities such as quality of service management (PAUL; TACHIBANA; HASEGAWA, 2016), admission control (CAVUSOGLU; ORAL, 2014), resource provisioning, and even network security (GUERRERO; LABRADOR, 2010) depend on knowledge of one or both of the available bandwidths and capacity. The link capacity is sometimes considered static and known a priori, but this premise cannot be guaranteed. Currently, capacities fluctuate on mobile networks and radio networks - due to variations in range, interference, and workload (PAKZAD; PORTMANN; HAYWARD, 2015).

In this context, two significant approaches stand out: techniques that consider traf-

fic prediction a Time Series Forecasting (TSF) problem and the ones that do not model it as time series forecasting problem (Non-TSF) (BOUTABA et al., 2018). The main difference between these approaches is the characteristics of the prediction and the methods used for it.

2.1.1 Time Series Forecast Problems

When modeled as a TSF problem, the system predicts the available bandwidth on a given path based on characteristics observed on that path in the last sample of the observation (such as the average number of bits per second). In other words, these techniques consider the traffic history of predicting its behavior. The objective is to construct a regression model capable of drawing an accurate correlation between future traffic volume and previously observed traffic volumes.

Time series models have three main types: autoregressive model (AR), moving average model (MA), and autoregressive moving average model (ARMA). An autoregressive model predicts future behavior based on past behavior. It is used for forecasting when there is some correlation between values in a time series and the values that precede and succeed. It consists of a linear regression of the data in the current series against one or more values in the same series. In contrast, a moving average is a technique to get an overall idea of the trends in a data set; it averages any subset of numbers. Lastly, an ARMA model is used to describe weakly stationary stochastic time series in terms of two polynomials. The first of these polynomials is for autoregression, the second for the moving average. ARMA model can deeply understand the structure of time series and achieve the optimal prediction of the minimum variance (WANG; LIU; GAN, 2018).

2.1.2 Non-Time Series Forecast Problems

In contrast, when we do not use TSF to model prediction problems, we consider other methods and characteristics; for instance, works such as (CHEN; WEN; GENG, 2016), (LI et al., 2016), and (POUPART et al., 2016). We can use different techniques to forecast network behavior, but in this work, we highlight the approaches described by these authors.

Chen et al. (CHEN; WEN; GENG, 2016) propose a Hidden Markov Model

(HMM) to describe the relationship between flow count, flow volume, and dynamic behavior. Li et al. (LI et al., 2016) focus on predicting incoming and outgoing traffic volume; for this means, they propose a frequency domain-based method. Lastly, Poupart et al. (POUPART et al., 2016) explore three different approaches to predict elephant flows (GUO; MATTA, 2001): Gaussian Process Regression (GPR), online Bayesian Moment Matching (oBMM), and Multi-Layer Perceptron Neural Network(MLP-NN). We discuss these works in Chapter 3.

The main idea when using Non-Time Series Forecast techniques is not working with stochastic data. This would be a solution to overcome the TSF limitation: only working with predictable data (based on its history). In this context, when using non-TSF approaches, one could choose from various methods to predict network behavior.

On the one hand, when modeled as a TSF problem, it is not easy to obtain data and use the proposed solution in a controlled environment (such as data centers). Conversely, when not modeled as a TSF problem, we usually obtain a low predictive power, as we will discuss in Chapter 3. Allied, these issues raise the need for new techniques that are highly accurate and cost-effective.

2.2 Load balancing

Load balancing refers to the efficient distribution of traffic within the network. This technique infers the classification and division of traffic flows to achieve the best usage of the network resources, and can minimize adverse effects such as packet retransmissions (PIZZUTTI; SCHAEFFER-FILHO, 2018). Load balancing has become a necessity as applications become complex, user demand grows, and traffic volume increases. This workload distribution is essential to achieve highly available infrastructures and to optimize network performance (ALIZADEH et al., 2014).

Load balancers run as hardware appliances or are software-defined. Hardware appliances often run proprietary software optimized to run on custom processors. As traffic increases, the vendor simply adds more load balancing appliances to handle the volume. On the other hand, software-defined load balancers provide flexibility to adjust for changing needs and lower costs than purchasing and maintaining physical machines. In this work, we focus on software-defined load balance. Software load balancers could provide benefits like predictive analytics that determine traffic bottlenecks before they happen. However, the main challenge for this approach is the possible delay while configuring

software during scaling activities.

We discuss load balancing schemes along two key dimensions: (i) the information used to make load balancing decisions; and (ii) the decision granularity. Firstly, the information used for decision making might include local traffic, topology, or global traffic. More specifically, global traffic information could be based on per-flow feedback, centralized controller, centralized arbiter, per-path feedback, or hop-by-hop probes. Secondly, the decision granularity is usually in terms of a packet, flow, or flowlet (VANINI et al., 2017). Load balancing designs that rely on global information about traffic conditions on different paths can handle asymmetry. There are many ways to collect this information with varying precision and complexity, ranging from transport-layer signals, centralized controllers, and in-network hop-by-hop or end-to-end feedback mechanisms.

An example is CONGA (ALIZADEH et al., 2014), which uses explicit feedback loops between the top-of-rack (also called leaf) switches to collect per-path congestion information. By contrast, schemes that are oblivious to traffic conditions generally have difficulty with asymmetry. This is the case even if different paths are weighted differently based on the topology, as some designs ((ZHOU et al., 2014) (CAO et al., 2013)) have proposed. Using the topology is better than nothing, but it does not address the fundamental problem of the optimal traffic splits depending on real-time traffic conditions.

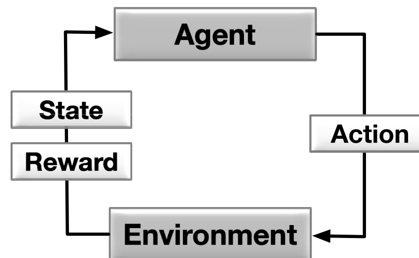
Because we want to collect statistics and have global traffic information to make better decisions, we use a software-defined load balancing on the SDN controller. Moreover, the granularity considered here is in terms of flows: we explore network traffic prediction to determine which network flows represent a risk to network load balancing. Hence, we discuss Machine Learning (ML) approaches to predict network traffic volume and apply actions to provide balanced network usage.

2.3 Reinforcement Learning

Reinforcement Learning is an agent-based iterative process for modeling problems for decision making (SUTTON, 2018). In order to make these decisions, this approach considers an agent that interacts with the environment. Instead of this agent being taught by examples, it learns by exploring the environment and exploiting the knowledge acquired. The agent will then take actions, and these actions will be rewarded or penalized based on their impact on the environment (BOUTABA et al., 2018). Therefore, the training data in RL constitutes a set of state-action pairs and rewards. We illustrate this concept

on Figure 2.1. At each time step, the agent acts on the environment. The environment then returns the reward associated with this action and the resulting state after the action. The agent will use this data to make better decisions on the next time steps.

Figure 2.1: Diagram illustrating how the agent interacts with the environment.



Source: the author.

Reinforcement Learning models are defined based on a set of states S, a set of actions A, and a set of corresponding rewards. Next, we discuss the terminology followed by most RL algorithms.

- **Environment**: the world through which the agent moves. The environment takes the agent's current state and action as input and returns as output the agent's reward and its next state.
- **agent**: an agent takes actions and chooses the best action for the current environment state. Briefly, the agent is the algorithm that chooses what actions to take to achieve its goal.
- **set of actions** (A): set of all possible moves the agent can make. Agents usually choose from a list of possible discrete actions. When defining the set of actions the agent could take, we need to consider how each of these actions will modify the current state.
- **state** (S): a state is a concrete and immediate situation in which the agent finds itself, *i.e.*, a specific place and moment, an instantaneous configuration that puts the agent concerning other significant things. The state is defined when modeling the problem we are trying to solve, and it should indicate how far the environment is from its goal.
- **Reward** (R): a reward is a feedback by which we measure the success or failure of an agent's actions in a given state. This reward is used to evaluate the agent's action. For example, using a power-saving model, if the agent chooses an action that makes the environment save energy, he wins a positive reward; otherwise, he earns a negative reward. From any given state, an agent sends output in the form of

actions to the environment. The environment returns the agent's new state (which resulted from acting on the previous state) and rewards if there are any.

- **policy** (π): the strategy that the agent employs to determine its next action based on the current state. In other words, policy specifies an action a that is taken in a state s; more precisely, π is the probability of action a being taken in a state s. In this context, we can divide reinforcement learning algorithms into on-policy learning, and off-policy learning. In on-policy learning, the reward function is learned from action that we took using a policy $\pi(a|s)$ (Monte Carlo, SARSA). On the other hand, in off-policy learning, the reward is learned from taking different actions (Q-Learning).
- **Value (V)**: the expected long-term return, as opposed to the short-term reward R. *Value* is defined as a function $V\pi(s)$, and corresponds to the expected long-term return of the current state under policy π.
- **Q-value or action-value** (Q): Q-value is similar to Value, except that it takes an extra parameter: the current action a. Q-Value is defined as a function $Q\pi(s, a)$, that refers to the long-term return of the current state s, taking action a under policy π.

An agent is then responsible for considering the current state, the set of possible actions, and their corresponding rewards to define the best action for the problem: the action that provides the greatest reward. In this context, an RL algorithm is an algorithm that iterates over these actions, selecting the best to accomplish a goal (SUTTON, 2018). The main RL algorithms are Q-Learning, State-Action-Reward-State-Action (also called SARSA), and Deep Q-Network (DQN), discussed in the following subsections.

2.3.1 Q-Learning

Q-learning is an off-policy RL algorithm used to find the best action that an agent should perform given a current state. This is an off-policy algorithm because the Q-learning function learns from actions outside the current policy. More specifically, Q-learning seeks to learn a policy that maximizes the total reward. The foundation of the algorithm is the presented the following equation, where $Q(s, a)$ is the expected value of

state s when taking action a.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} p_0(s'|s,a) max_{a'} Q(s',a') \tag{2.1}$$

which its stochastic version is

$$Q(s,a) = Q(s,a) + \alpha(R(s,a) + max_{a'} Q(s',a') - Q(s,a)) \tag{2.2}$$

where $p_0(s'|s,a)$ is the probability to the transition of s to s' under action a. γ is a discount factor, and it balances immediate and future rewards. A factor of 0 will make the agent only consider current rewards, while a factor close to 1 will make it strive for a long-term high reward. Typically this value can range anywhere from 0.8 to 0.99.

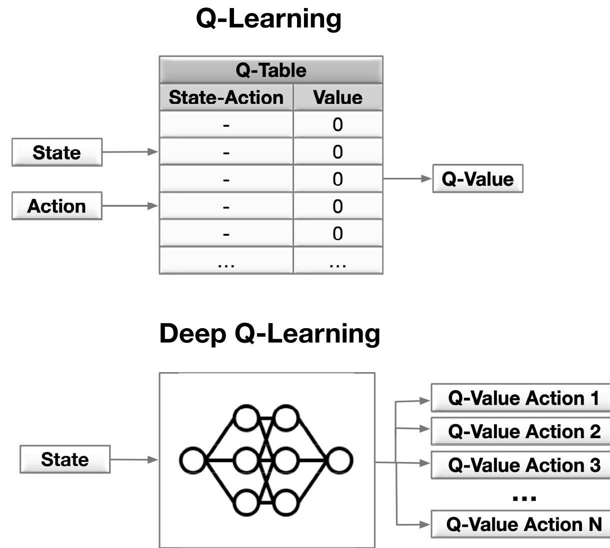
α is the learning rate, and it determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge). In contrast, a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities). Typically this value is usually set in the inteval from 0.1 and 0.3.

When performing Q-learning, we create a table called Q-table. This table stores the Q-value for each $(state, action)$ pair - initialized to 0. After each episode (also knonw as iteration), we update and store these Q-values. Hence, this Q-table becomes a reference table for the agent to select the best action based on the Q-value. The equation used to update the Q-value is presented next.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3}$$

An agent interacts with the environment by exploring or exploiting. The first is to use the Q-table to reference and view all possible actions for a given state. The agent then selects the action based on the maximum value of those actions. This method is known as exploiting since we use the available information to make a decision. The second way to take action is to act randomly - called exploring. Instead of selecting actions based on the maximum future reward, we select a random action. Random actions are vital because they allow the agent to explore new states that otherwise may not be selected during the exploitation process. We can then balance exploration and exploitation using an ϵ factor and set the value of how often we want to explore instead of exploit (SUTTON, 2018).

Figure 2.2: Diagram illustrating the difference between Q-Learning and Deep Q- Learning.



Source: the author.

One of the biggest challenges about RL usage is to find the balance between the exploration of new states and the exploration of prior knowledge. That is, finding the balance between considering previous experiences and exploring new possibilities. In particular, concerning traffic routing, we must explore many routes until we find the optimal one. However, exploring new routes is extremely computationally costly and can harm the system when evaluated by the quality of service. Works such as Arroyo-Valles et. al (ARROYO-VALLES et al., 2007), Bhorkar et. al (BHORKAR et al., 2012), and Li et. al (LIN; SCHAAR, 2011) seek to solve this problem in the context of traffic routing and explore these issue.

2.3.2 Deep Q-Learning (DQL)

A major limitation of Q-learning is that it only works in environments with discrete and finite state and action spaces. To address this issue, deep Q-learning combines reinforcement learning and deep neural networks. More specifically, it uses a neural network to approximate the Q-value function. More specifically, it uses the state as input for a neural network, and the output it the Q-value of all possible actions (FRANÇOIS-LAVET et al., 2018). We illustrate this concept in Figure 2.2.

To avoid computing the full expectation in the DQL loss, we can minimize it using stochastic gradient descent. When computing the loss using just the last transition

s, a, r, s', we reduce it to standard Q-Learning. This technique is called Experience Replay, and it is used to make the network updates more stable. At each time step of data collection, the transitions are added to a circular buffer called replay buffer. During training, instead of using just the latest transition to compute the loss and its gradient, we compute them using a mini-batch of transitions sampled from the replay buffer. This process has two main advantages: better data efficiency by reusing each transition in many updates and better stability using uncorrelated transitions in a batch (XU et al., 2018).

2.3.3 On-Policy and Off-Policy Learning

Considering that almost all reinforcement learning algorithms involve estimating value functions, formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience. In this context, solving a reinforcement learning task means finding a policy that achieves a great amount of reward over time (SUTTON; BARTO, 2018).

With this in mind, it is important to consider two concepts: update policy (also known as target policy) and behavior policy. The first is how the agent learns the optimal policy, and the latter is how the agent behaves (how the agent selects an action). To illustrate this concept, consider the following equations.

Q-Learning equation to update Q-value:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \lambda max_a Q(S', a) - Q(S, A)] \tag{2.4}$$

equation to update Q-value:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \lambda Q(S', a) - Q(S, A)] \tag{2.5}$$

When using Q-Learning, the agent learns optimal policy using absolute greedy policy and behaves using other policies such as $\epsilon - greedy$ policy. Because the updated policy is different from the behavior policy, Q-Learning is off-policy. More specifically, Q-learning is off-policy because it updates its Q-table using the Q-value of the following state S' and the greedy action a. In other words, it estimates the *return* (total discounted future reward) for state-action pairs assuming that a greedy policy was followed, although

it is not following a greedy policy.

In contrast, when using SARSA, the agent learns optimal policy and behaves using the same policy. Because the updated policy is the same as the behavior policy, SARSA is on-policy. It is on-policy because it updates its Q-values using the next state S' and the current policy's action. It estimates the return for state-action pairs assuming the current policy continues to be followed.

In short, an off-policy learner learns the value of the optimal policy from actions taken using the current policy. An on-policy learner learns the value of the policy being carried out by the agent, including the exploration steps (random actions) (SCHULMAN et al., 2017).

3 RELATED WORK

This chapter presents the main related work to the research presented in this document. We highlight the following areas: machine learning for traffic prediction and machine learning for load balancing and traffic routing. In the following subsections, we discuss these efforts and how they lead us to the questions we address in this document.

3.1 Machine learning for traffic prediction

Network traffic prediction entails forecasting future traffic and traditionally has been addressed via Time Series Forecasting (BOUTABA et al., 2018). However, considering that modeling a time series is not always possible, several research efforts do not consider this a TSF problem. Instead, the authors leverage other methods and features to address traffic prediction issues.

In the next subsections, we highlight research efforts that considered traffic prediction as a TSF problem and other works that do not consider it a TSF problem. We present their most relevant aspects concerning the work presented in this document.

3.1.1 Traffic prediction modeled as a Time Series Forecast problem

One of the approaches used for preventive control is to predict the near future traffic in the network and then take appropriate actions - such as controlling buffer sizes. Several works developed in the literature are interested in resolving the problem of improving network traffic monitoring's efficiency and effectiveness by forecasting data packet flow in advance. In this context, an accurate traffic prediction model should have the ability to capture prominent traffic characteristics.

The literature has shown that neural networks are one of the best alternatives for modeling and predicting traffic parameters, possibly because they can approximate almost any function regardless of their degree of nonlinearity and without prior knowledge of its functional form (BOUTABA et al., 2018). In fact, according to Boutaba et al., Yu et al. (YU; CHEN, 1993) were the first to apply ML in traffic prediction using Multi-Layer Perceptron Neural Networks (MLP-NN). The primary motivation was to improve accuracy over traditional Auto-Regressive (AR) methods. After that, several works began

to explore different neural networks to address network traffic forecasting.

Motivated by the issue of how to conduct network resource planning (NRP), Wu et al. (WU et al., 2019) propose a novel time-series framework to model and forecast traffic dynamics in machine-to-machine (M2M) communications (GAZIS, 2017). The authors base their approach on existing research efforts that suggest that network traffic modeling and forecasting are capable of aiding with the NRP processes to manage network resources (AL-KHATIB; HARDJAWANA; VUCETIC, 2014) (KOSEOGLU, 2017). This TSF model utilizes the statistical techniques INGARCH (p, q) (integer-valued generalized autoregressive conditional heteroskedasticity) and βARMA (p, q) (beta autoregressive moving average). The authors use this model to capture both the internal impact factors (previous traffic dynamics, random errors, etc.) and external impact factors (topology, bandwidth, M2M device moving speed, etc.) asynchronous and synchronous M2M traffic dynamics over a large time scale. The proposed framework is composed of three components: (i) Traffic Dynamics Modeling (TDM), to imitate the characteristics of the asynchronous and synchronous M2M real-time traffic dynamics, by leveraging INGARCH and βARMA; (ii) Parameter estimation, to specify values for the set of parameters in their scheme; and (iii) Forecasting, that provides outlooks for multiple upcoming M2M network traffic by using conditional maximum-likelihood estimators (CMLE). To evaluate their model, the authors performed theoretical analysis and simulations and validated its forecasting efficiency. They consider two metrics: prediction accuracy and goodness-of-fit (identifying how well the models fit their corresponding sets of observations). Experiments demonstrate that TSF achieves superior performance regarding both metrics.

Wang et al. (WANG; LIU; GAN, 2018) studied how to establish a valuable and accurate forecasting model for network traffic. The authors propose a Grey Model First Order One Variable (GM (1,1)) model (LIU; FORREST, 2006) to improve prediction accuracy and combine it with the Auto-Regressive Moving Average Model (ARMA). Then, the authors propose that the reformed GM(1,1) model would generate higher prediction precision. The combination weighting method is then adopted to combine the ARMA(p,q) model with the reformed GM(1,1), resulting in a new prediction model. The combination forecasting model is a kind of forecasting method that selects appropriate weights to weigh and average the results obtained from several forecasting methods. Overall, experiment results show the validity of the model.

Cortez et al. (CORTEZ et al., 2006) chose a NN ensemble (NNE) of five MLP-NN

with one hidden layer each. For training, they use resilient back-propagation (BP), and, for data, Simple Network Management Protocol (SNMP) traffic data collected from two different Internet Service Providers (ISP) networks. The first data represents the traffic on a transatlantic link, while the second represents the aggregated traffic in the ISP backbone. To complete the missing SNMP data, the authors used linear interpolation. They test the NNE for real-time forecasting (online forecasting on a few-minute sample), short-term (one-hour to several-hour sample), and mid-term forecasting (one-day to several-day sample). The comparison amongst the TSF methods shows that, in general, the NNE produces the lowest Mean Absolute Percentage Error (MAPE) for both datasets. It also shows that NNE outperforms the other methods with an order of magnitude in terms of time and computational complexity and is well suited for real-time forecasting.

As shown in this subsection, several research efforts model network traffic prediction a TSF problem and obtain successful results. However, this approach is very restrictive: to apply these models, we need to work with stationary data. Stationary data is the kind of data with all statistical properties (such as mean and variance) remaining constant over time. This need is because if we take a particular behavior over time, this behavior must be the same in the future to forecast the series. With this in mind, several research efforts consider traffic prediction on non-TSF models focusing on non-stationary series. We highlight some of them in the following subsection.

3.1.2 Traffic prediction modeled as a non-Time Series Forecast problem

As explained in the previous subsection, using TSF models to predict network traffic features is not always possible, especially because of the need for stationary data. With this in mind, several works use different methods and characteristics to predict traffic (CHEN; WEN; GENG, 2016) (LI et al., 2016) (POUPART et al., 2016).

In work developed by Chen et al. (CHEN; WEN; GENG, 2016), the authors investigate the possibility of reducing the cost of monitoring and collecting traffic volume by inferring the volume of future traffic based on flow counting. For this, they propose a Hidden Markov Model (HMM) to describe the relationship between flow count, flow volume, and dynamic behavior over time. To predict future traffic volume based on current flow count, they use Kernel Bayes Rule (KBR) and Recurrent Neural Network (RNN) with long short-term memory (LSTM). The experiments use semi-simulated data and real network traffic data to demonstrate the feasibility of inferring and predicting network traffic

volume based on simple flow statistics such as flow counts. The authors use a normalized dataset composed of network traffic volumes and corresponding flow counts collected every 5 min over 24 weeks. The RNN shows a prediction mean squared error (MSE) of 0.3 at best, 0.05 higher than KBR, and twice as much as the prediction error of an RNN fed with traffic volume instead of flow count. These results indicate that these techniques provide useful information to predict traffic volume. However, considering that the motive was to promote flow count-based traffic prediction to lower the monitoring cost, the experiments also show that the performance is compromised. Besides, the authors highlight an issue as future work: the nonstationarity of the network traffic. In this context, it would be essential to develop online learning algorithms for KBR and RNN such that the model adjusts and adapts itself to the dynamic network traffic.

Considering that traditional inter-DC transfers suffer from both low utilization and congestion, Li et al. (LI et al., 2016) suggest that traffic prediction is a crucial method to optimize these transfers. To address this issue, the authors propose a frequency domain-based method for network traffic flows instead of just traffic volume. They emphasize practical issues in the prediction model design, especially the cost of measurements, and show that it is possible to reduce the flow sampling overhead using interpolation methods. The focus is on predicting incoming and outgoing traffic volume on an inter-data center link dominated by elephant flows. Their model combines wavelet transform with Feedforward Neural Network (FNN) to improve prediction accuracy. This FNN was trained with BP using gradient descent and wavelet transform to capture both the time and frequency features of the traffic time series. The authors add interpolation to fill in the elephant flows' unknown values to reduce the amount of monitoring overhead for the elephant flow information. In the experiments, the dataset contains the total incoming and outgoing traffic collected in 30 seconds intervals using SNMP counters on the data center (DC) edge routers and inter-DC links over six weeks. Then, they decompose the time series using level 10 wavelet transform, leading to 120 features per timestamp. Consequently, k-step ahead predictions feed $k * 120$ features into the NN and show a relative RMSE (RRMSE) ranging from 4 to 10% for the NN-Wavelet transformation model, as the prediction horizon varies 30 seconds to 20 minutes. Results show that their approach can reduce prediction errors over existing methods (such as ARIMA (AKAIKE, 1969)) by values between 5% and 30%. This work is vital to the context of our research because it indicates that the use of artificial neural networks can reduce prediction errors when used to predict network traffic characteristics.

Poupart et al. (POUPART et al., 2016) explore the use of different ML techniques for flow size prediction and elephant flow detection. They propose to use data mining to estimate each flow's size as it starts and to detect elephant flows without modifying any application or end host. To this end, they use three machine techniques: gaussian processes regression (GPR), online bayesian moment matching (oBMM), and a neural network (MLP-NN). For each flow, the authors consider seven features: source IP, destination IP, source port, destination port, protocol, server versus client (if the protocol is TCP), and the size of the first three data packets after the protocol/synchronization packets. The datasets consist of three public datasets, with over nine million flows. To detect elephant flows, they use a flow size threshold that varies from 10KB to 1MB. The experiments show discrepancies in the performance of the approaches with different datasets. Although oBMM outperforms all other methods in one dataset with an average true positive rate (TPR) and true negative rate (TNR) very close to 100%, it fails with an average TPR below 50% dataset. In one of the datasets, oBMM seems to suffer the most from class imbalance: as the threshold increases, fewer flows are tagged as elephant flows, creating a class imbalance in the training dataset and lower TPR. In short, oBMM outperforms all other approaches in terms of average TNR in all datasets. On the other hand, NN and GPR, have an average TPR consistently above 80%. The motive for flow size prediction in (POUPART et al., 2016), is to discriminate elephant flows from mice flows in routing to speed up elephant flow completion time. Mice flows are routed through Equal-cost multi-path routing (ECMP), while re-routing elephant flows through the least congested path. Results show that the resulting routing policy reduced the average completion time of elephant flows while keeping the average flow completion time of mice flows roughly the same.

The approach presented in Poupart et al. (POUPART et al., 2016) is especially important for the work we present in this document because it addresses flow size prediction to identify elephant flows and route them via a least congested path, which is also the motivation for our work. In the context of flow size prediction, the authors show that the use of six additional features that can be extracted from the first packet's header provides helpful information. They also show the benefits of elephant flow detection in routing by assigning the least congested path to elephant flows, which is essential for our research.

The Non-TSF approaches presented in this subsection infer traffic volumes from flow count and packet header fields. Predicting traffic volume is helpful in several contexts, such as traffic routing aiming load balancing. We explore this subject in the follow-

ing subsection.

3.2 Traffic engineering

Traffic routing technique is quite challenging because it requires that machine learning models, when dealing with complex (and sometimes dynamic) network topologies, learn the relationship between the selected path and the perceived impact on the network. Also, it is necessary to assess the ability to predict the consequences of routing decisions. With this in mind, two primary research efforts inspired our work: "Let It Flow" by Vanini et al. (VANINI et al., 2017), and "Learning to route" by Valadarsky et al. (VALADARSKY et al., 2017). We present their proposals as it follows, together with other research efforts that address similar issues.

3.2.1 Traffic routing for load balancing

Network traffic load balancing is essential in data centers because they must provide large bisection bandwidth to support the increasing traffic demands of applications (such as big data analytics, web services, and cloud storage). With this in mind, Vanini et al. (VANINI et al., 2017) published a work addressing this issue. The authors present the concept of flowlet: a burst of packets separated in time from other bursts by a sufficient gap. Their work then shows that flowlet switching is a powerful technique for resilient load balancing with asymmetry (link failures and heterogeneity in network equipment, for example). Their method is called LetFlow, and it is an instance of a more general approach to load balancing: it randomly re-routes flows with a probability that decreases as a function of the flow's rate. However, schemes that lack visibility into path congestion have a critical drawback: they perform poorly in asymmetric topologies. The reason is that the optimal traffic split across asymmetric paths depends on (dynamically varying) traffic conditions; hence, traffic-oblivious schemes are fundamentally unable to make optimal decisions and can perform poorly in asymmetric topologies (ALIZADEH et al., 2014). Flowlets have a remarkable elasticity property: their size changes automatically based on traffic conditions on their path. Hence, LetFlow picks routes randomly for flowlets and lets their elasticity naturally balance the traffic on different paths. The authors show that LetFlow is a significant improvement over ECMP and could deploy it today to improve

resilience to asymmetry. This technique is trivial to implement in hardware, does not require any changes to end-hosts, and is incrementally deployable.

This work raises the question: could we have better results if we applied an intelligent routing instead of randomly re-route flows? In this context, we highlight the need for using intelligence to route, explored in the next section ((MEKINDA; MUSCARIELLO, 2016), (MAO et al., 2017), (VALADARSKY et al., 2017), and (BARROS et al., 2019)).

3.2.2 Machine learning for traffic routing

The work presented by Medkinda et al. (MEKINDA; MUSCARIELLO, 2016) acts in the context of Named Data Networking (NDN), where routers forward Interests for content after finding Longest-Prefix Matches (LPM) of content names in their Forwarding Information Base (FIB) (ZHANG et al., 2014). In this kind of network, scalability is a challenge because of the vast global Internet namespace. Hence, they propose a novel approach to interest forwarding that compresses the FIB data structure into NNs. Their work investigates the consequences of training routers to "guess" paths - they qualify the information retrieval via NN-FIBs as guessing because of the unpredictable nature of the query outcome. To this aim, the control plane would access the Pending Interest Table (PIT) for actual names, find their longest matching prefix in the Routing Information Base (RIB) - populated by a name-based routing protocol - and find the NN-RIB in charge. These NNs are offline trained by the control plane from the RIB and matching interests. Finally, these NNs are made available to the data plane for interrogation. Experiments show that NN-FIBs are orders of magnitude smaller and faster, with accurate egress face guess. Conversely, it appears that supervised learning has to be carried out offline, preferably inside the control plane, which might be local or centralized within an SDN controller.

Mao et al. (MAO et al., 2017) propose a smart packet routing strategy with Tensor-based Deep Belief Architectures (TDBAs) that considers multiple parameters of network traffic. In TDBAs, they use tensors to represent the units in every layer and the weights and biases. The proposed TDBAs can be trained to predict the whole paths for every edge router. The packets are supposed to be generated inside the network and destined for edge routers, while the internal routers just forward packets. Every edge router obtains the traffic patterns of the other edge routers through the signaling process. Then, the edge routers input the traffic patterns to the TDBA and construct the paths to all other

edge routers. The whole paths are attached to the headers of corresponding packets. Consequently, other routers forward the packets according to the labeled paths. Simulation results demonstrate that this proposal outperforms the conventional Open Shortest Path First (OSPF) protocol regarding overall packet loss rate and average delay per hop.

More recently, Valadarsky et al. (VALADARSKY et al., 2017) investigate how ideas and techniques from machine learning could be leveraged to generate routing configurations automatically. The authors focus on the classical setting of intradomain traffic engineering and observe that this context poses significant challenges for data-driven protocol design. They study ML-guided routing by examining the environment of intradomain traffic engineering (TE), that is, the optimization of routing within a single, single-administered network. The investigation focused on two main questions: (1) How should routing be formulated as an ML problem? Furthermore, (2) What are suitable representations for the inputs and outputs? Their preliminary results regarding the power of data-driven routing suggest that applying ML to this context yields high performance and is a promising direction for further research. More specifically, they evaluate several supervised learning schemes for predicting traffic demands. Preliminary results indicate that supervised learning might be ineffective if the traffic conditions do not exhibit very high regularity. Next, they focus on reinforcement learning: instead of explicitly learning future traffic demands and optimizing these demands, the goal is to learn a proper mapping from the observed history of traffic demands to routing configurations. Results suggest that this is a better approach. The authors leave several questions to be answered and leveraged more deeply in future work. From these questions, we highlight (1) the possibility of using a different objective function than minimizing max-link-utilization (for instance, flow-completion-time) to yield better results; and (2) how scalable is the ML approach in this context.

Lastly, Barros et al. (BARROS et al., 2019) proposes a novel mechanism for management, orchestration, and flow control in the context of the device-to-device (D2D) to deal with load balancing using the deep Q-learning (DQN) technique. The authors implemented a D2D network simulation environment, using the ParticiptAct dataset to evaluate the load of the cell towers in a region of Italy. They used Gauss-Markov and Gilbert-Elliott models for mobility and packet loss, respectively, where it was considered that the towers had a separate coverage area, hence forming a Voronoi space. They also used a Gaussian process to predict the load of the towers when they receive the packet and a DQN to perform the balance of load of the network. They consider delivery rate,

Table 3.1: Comparison between research efforts discussed in this document.

Work	Traffic prediction	Traffic routing	Traffic routing with prediction results	Machine learning techniques	Network issues adressed
Wu et. al (WU et al., 2019)	TSF	No	No	INGARCH, Beta-ARMA, CMLE	Network resource planning
Wang et. al (WANG; LIU; GAN, 2018)	TSF	No	No	ARMA, GM(1,1)	Network forecast
Cortez et. al (CORTEZ et al., 2006)	TSF	No	No	Neural Network Ensemble	Network forecast, anomaly detection
Chen et. al (CHEN; WEN; GENG, 2016)	Non-TSF	No	No	Hidden Markov Model, Kernel Bayes Rule, Recurrent Neural Networks	Reducing the cost of network monitoring, network forecast
Li et. al (LI et al., 2016)	Non-TSF	No	No	Wavelet transform, ANN	Congestion control, elephant flow detection
Poupart et. al (POUPART et al., 2016)	Non-TSF	Yes	Yes	Gaussian Processes Regression, Online Bayesian Moment Matching, MLP neural network	Flow size prediction, elephant flow prediction
Vanini et. al (VANINI et al., 2017)	-	Yes	-	-	Resilient asymetric load balancing
Mekinda et. al (MEKINDA; MUSCARIELLO, 2016)	-	Yes	-	ANN	Accelerate packet forwarding, NDN
Mao et. al (MAO et al., 2017)	-	Yes	-	Deep Learning with tensors	Intelligent packet routing
Valadarsky et. al (VALADARSKY et al., 2017)	-	Yes	-	Deep neural networks, Reinforcement Learning	Intelligent packet routing
Barros et. al (BARROS et al., 2019)	Non-TSF	Yes	Yes	Gaussian-Markov, Deep Q-Learning	Device-to-device (D2D) load balancing
Look-Ahead Reinforcement Learning	Non-TSF	Yes	Yes	Reinforcement Learning, Deep Q-Learning	Network forecast, load balancing

delay, and a load-balancing metric based on a custom utility function to evaluate their proposal. Their approach presents better results than the baseline approach used in their experiments.

3.3 Discussion

We use Table 3.1 to illustrate how the presented research efforts relate to the work presented in this document. Existing research efforts tend to focus either on network traffic forecast or network routing (chosing the best route for a flow). Our proposal, however, is to consolidate both into a single architecture. In this context, we would be able to use network traffic prediction to enhance network routing decisions. We can split the problem into two issues: network traffic prediction and network routing with this prediction.

Network traffic prediction. To address this issue, based on existing work on the literature, we focus on network forecast as a non-TSF problem, mainly because we understand the limitations of working with stochastic data. Also, recent research efforts show that neural networks can solve this kind of problem.

Network routing. Several approaches have been proposed to monitor flows in the network and detect elephants based on thresholds concerning the amount of data transmitted so far or the bandwidth utilized. These approaches can only detect elephant flows after they have been flowing for a while, and therefore they only permit re-routing and re-

scheduling. This is not ideal since congestion may occur until elephant flows are detected, and new routes are chosen by load balancing, or the priorities of the elephant flows are decreased to allow mice flows to complete without any delays (POUPART et al., 2016). With this in mind, and considering that reinforcement learning showed promising results for choosing the best routes for a given flow, we propose adding an extra step into the reinforcement learning algorithm for routing: a detection step. This is the concept of "look-ahead" reinforcement learning. Instead of taking action on flows and evaluate their impact on the environment, we would look ahead for elephant flows. Then, these elephant flows would be the subject of the routing decision.

In short, we propose a new framework for traffic routing and load balancing using the look-ahead reinforcement learning technique. We hypothesize that this extra step could prevent unnecessary actions in the environment. We discuss this proposal in the next chapter.

4 LOOK-AHEAD REINFORCEMENT LEARNING FOR LOAD BALANCING

In this chapter, we present the Look-Ahead Reinforcement Learning architecture for load balancing. We start with an approach overview in Section 4.1, followed by the modeling and fundamentals of this work in Section 4.2. Lastly, in section Section 4.3, we discuss how we integrated the presented concepts and came up with the architecture proposed in this work.

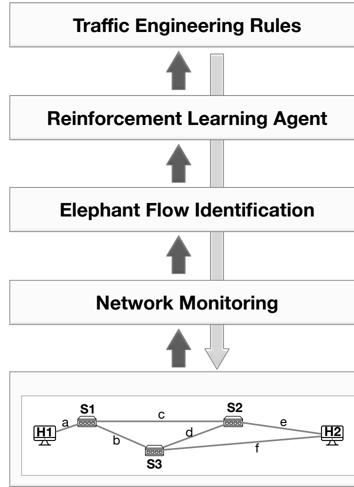
4.1 Approach Overview

This work presents a mechanism for load balancing network flows using a strategy based on reinforcement learning. This mechanism attempts to explore alternative paths in the topology in order to ensure low overall bandwidth utilization. However, instead of unwisely trying to load balance any traffic flow, our reinforcement learning agent will concentrate only on flows that can significantly impact bandwidth utilization, e.g., elephant flows (HAMDAN et al., 2020b).

We advocate that adding an extra step to identify whether the flow will be an elephant flow (since early stages) before applying a reinforcement learning algorithm to load balance could improve routing decisions. The primary motivation is to avoid wasting resources on rerouting flows that would not represent a risk to network traffic balance. Secondly, we aim to improve load balancing results by only acting on the flows that have a higher impact on bandwidth utilization.

This architecture consists of four main components: Network Monitoring, Elephant Flow Identification, Reinforcement Learning Agent, and Traffic Engineering Rules, illustrated in Figure 4.1. These components are responsible for collecting network usage information to identify if a reinforcement learning agent should intervene in the network's functioning. In positive cases, we have a final component responsible for translating this agent's actions into network operation actions (OpenFlow rules). This architecture is especially suitable for massive traffic volumes because it can suppress latency costs when considering elephant flows.

Figure 4.1: Look-Ahead Reinforcement Learning for load balancing network traffic approach overview.



Source: the author.

4.2 Reinforcement Learning Agent Modelling

The main goal of the reinforcement learning agent we modeled is to balance the workload among the network resources. More specifically, to distribute the flows among different routes to have a more homogeneous usage of the links. With this in mind, we represent the network as a directed graph $G = (V, E)$. V is a set of vertices (or nodes), where each vertex corresponds to either a switch or a host. E is the set of edges (or links) that connect a pair of vertices defined below.

$$E \subseteq \{\{x, y\} | (x, y) \in V^2 \wedge x \neq y\}$$

Where (x, y) is an ordered pair of distinct vertices.

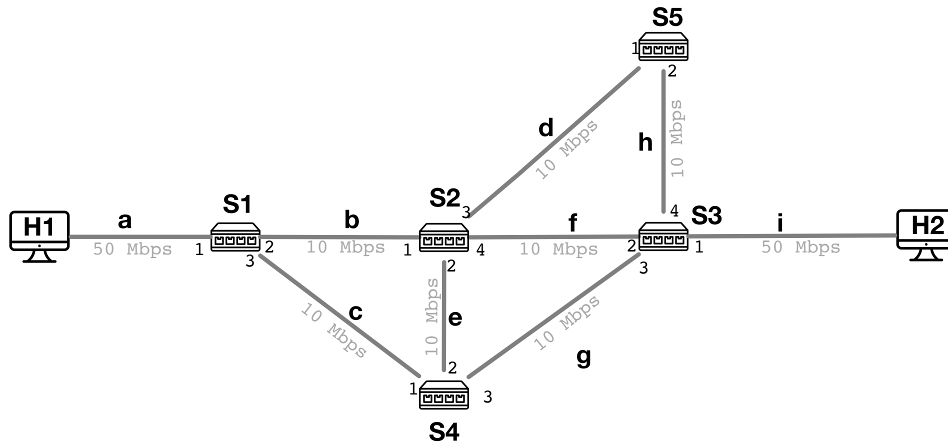
For example, considering the topology in Figure 4.2, we have the following sets:

- Set of vertices = $V = \{h_1, h_2, s_1, s_2, s_3, s_4, s_5\}$
- Set of edges = $E = \{a, b, c, d, e, f, g, h, i\}$

To illustrate our approach, consider the network topology presented in Figure 4.2. In this topology, we consider that the flow source is always H1, and the flow target is always H2, and that we have several concurrent (possible elephant) flows in this network. Thus, we aim to prioritize the homogeneous use of links.

We can adapt this definition and the topology according to the network problem we want to solve - this is where we combine the knowledge of networks with the knowledge of machine learning. Considering that the goal of this work is to maintain a homogeneous

Figure 4.2: Example network topology used to model our Reinforcement Learning agent.



Source: the author.

Table 4.1: Representation of snapshot 1, with a 5Mbps flow.

Port	S1.1	S1.2	S1.3	S2.1	S2.2	S2.3	S2.4	S3.1	S3.2	S3.3	S3.4	S4.1	S4.2	S4.3	S5.1	S5.2
Bits-per-second	5120	5120	0	5120	0	0	5120	5120	5120	0	0	0	0	0	0	0

use of the topology links concerning traffic occupation, we base the created model entirely on load balancing criteria. In this context, our objective function is to maximize the results of the reward function.

The main components of this model are state, reward function, and actions discussed next.

4.2.1 State

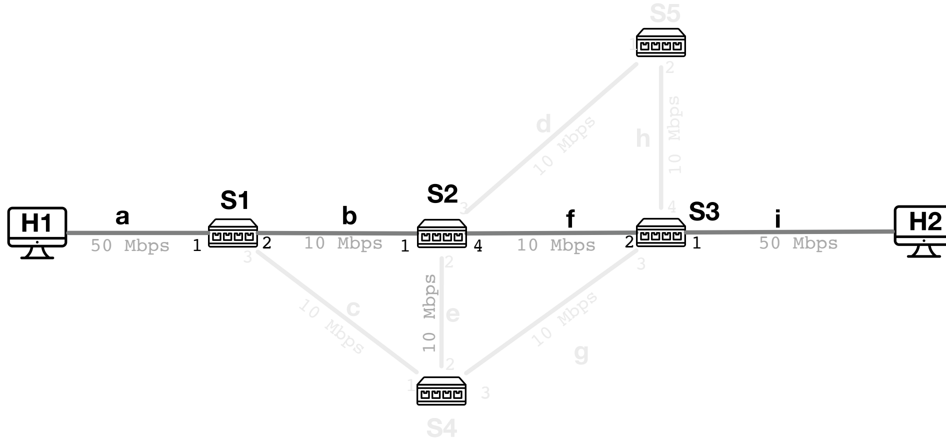
As discussed in Chapter 2, the environment state should be a concrete representation of what the agent finds. We consider the state as the number of bits transferred per second (Mbps). We calculate this value based on network statistics collected from the network periodically. We define the state at time t, S_t, to be an n-dimensional vector $[x_1, \dots, x_n]$, where n corresponds to the number of switch ports and where $x_i, for 1 \leq i \leq n$, corresponds to the number of bits being transferred per second by the $i-th$ switch port.

To illustrate the snapshot and resulting state, consider a 5Mbps flow through links a, b, f and i, as shown in Figure 4.3. Table 4.1 represents the snaspshot that captured this moment (snapshot 1) and Table 4.2 illustrates the resulting state.

Table 4.2: Representation of state 1, generated from snapshot 1 illustrated on Table 4.1.

State	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t(1)	5	5	0	5	0	0	5	5	5	0	0	0	0	0	0	0

Figure 4.3: Network topology for state when H1 routes workload to H2 through a, b, f and i links.



Source: the author.

As shown in Table 4.2, the resulting state is the primary tool used for the agent to identify whether the network could become overloaded. With information about the network topology, the agent will evaluate how costly a state could be to this network. That is, how homogeneous is the use of the topology links regarding traffic occupation. This cost is represented as the reward function, discussed next. Naturally, when the network has only one flow, there is a high cost involved. We consider this approach for networks with a high amount of flows.

4.2.2 Reward function

A reward function should let the agent know how close it is to its goal. We address load balance issues; therefore, we want to evaluate each network state regarding resource usage. More specifically, considering that we define the state of our problem as the number of bits transferred among the network, we need to evaluate how to distribute the workload. To address this issue, we propose three different reward functions. These functions have a different foundation: weighted switch ports usage, switch ports usage harmonic mean, and switch ports usage standard deviation. Considering that each reward function creates a different agent, we call the generated agents *WeightedUsage*, *UsageHarmonicMean*, and *UsageStandardDeviation*, respectively. We selected these three functions to measure how homogenous is the use of the network resources. Then, one way of defining the reward R_t, at time t, is in terms of an application and context-dependent function, f,

as follows:

$$R_t(S_t) = f(S_t)\mu \tag{4.1}$$

where f is defined by a designer depending on the specific criterion that they wish to optimize when trying to maintain a more homogeneous use of the topology links: to seek homogeneity by distribution traffic according to (1) weighted usage of switch ports; (2) harmonic mean of switch ports usage; or (3) usage standard deviation switch ports usage. Finally, in Equation 4.1, μ is defined as the sum of bits transferred by the source host output port and the target host input port. Intuitively, μ is an additive reward penalty reflecting whether possible loops exist in the network.

With regards to loop control, we consider the factor presented in Equation 4.2. This factor is the sum of bytes transferred by the source host (B_0) with the bytes received by the target host (B_1). We then multiply it by the result of the Equation 4.1. With this factor, we can penalize the agent for choosing actions that might lead to a loop state, where the source is sending a large amount of data, and the target is having trouble receiving it. Equation 4.2 represent how to calculate the discount factor for all agents.

$$\mu = B_0 + B_1 \tag{4.2}$$

In addition to these equations, each reward function has its specific heuristic to calculate how far the agent is from its final goal. *WeightedUsage* uses Equation 4.3, *UsageHarmonicMean* uses Equation 4.4, and *UsageStandardDeviation* uses Equation 4.5.

$$f(S_t) = \sum_{i=1}^{n} p(x_i) \tag{4.3}$$

where $p(x_i) = 2x_i$ if $x > 1$, and 1 if $x <= 1$. An ϵ factor is used to address division by 0 issues.

Weighted usage heuristics: Equation 4.3 presents the weighted usage heuristics, where we consider the final reward function as the sum of all switch ports. This function aims to give higher rewards to states that use more resources (switch ports) because we would be spreading the traffic and reducing the total flow completion time intuitively.

$$f(S_t) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}} \tag{4.4}$$

Harmonic mean usage heuristics: Equation 4.4 present the switch ports usage

harmonic mean, used to identify homogenous network usage. Thus, we consider all n switch ports in the topology and calculate the harmonic mean of the number of bits transferred by all ports.

$$f(x) = \sigma_{x_{\mathbf{n}}} \tag{4.5}$$

Standard deviation usage heuristics: Equation 4.5 corresponds to the switch ports usage standard deviation. In this function, we consider the standard deviation of the number bits x transferred by each of the n ports. Once again, we use this value in Equation 4.1 to calculate the final reward for state S_t.

To better understand these functions, consider the network state presented in Table 4.2. μ has the same value for all heuristics:

$$\mu = B_0 + B_1 = 5120 + 5120 = 10240$$

Considering the weighted usage heuristics (Equation 4.3), we calculate the reward as it follows.

$$f(x) = \sum_{i=0}^{n} g(y)_i = (6 * (2 * 5) + 10 * (1)) = 70$$

$$Reward(S_t) = f(x) * \mu = 70 * 10240 = 716800$$

In the same way, Equation 4.4 would yield the following reward:

$$f(x) = \frac{16}{6 * \frac{1}{5} + 10 * \frac{1}{0.00001}} = 1.66 * e^{-5}$$

$$Reward(S_t) = f(x) * \mu = 1.66 * e^{-5} * 10240 = 0.1638398$$

where 0.00001 is the epsilon used for minimum values.

Lastly, Equation 4.5 would yield the following reward:

$$\sigma_{x_n} = 2.5354576932600215$$

$$Reward(s_t) = 2596.31 * 10240 = 2658621440$$

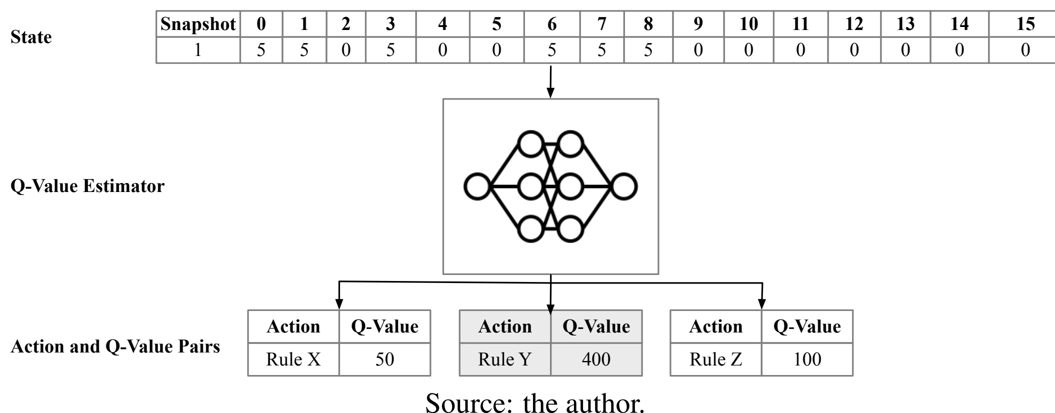
4.2.3 Actions

Since our goal is to balance network traffic, our actions should allow the agent to do so. Hence, we define the set of possible actions as the set of tuples in the format $< switch_id, in_port, out_port >$. Thus, an action is an OpenFlow rule to be installed on a switch. $switch_id$ corresponds to the switch where a rule will be installed; in_port is the port from where the flow will be coming; and out_port is the port to where the flow will be routed.

As an example, consider a flow $F1$ starting on host $H1$ to host $H2$ in the topology illustrated in Figure 4.2. A possible action for this topology would be $S2, 1, 2$, indicating that a flow coming into port 1 on switch S2 should use output port 4.

Combining state, actions, and reward function, we have our RL agent. This agent is responsible for interacting with the environment and choosing the best action to accomplish its goal. In this context, the RL agent will use an RL algorithm to analyze the network state and choose the best action. Figure 4.4 illustrates this process.

Figure 4.4: Reinforcement Learning agent steps to select the best action. The agent iterates over each step to find the best actions. The chosen action is the one highlighted.



Source: the author.

With the definition of a reinforcement learning problem, an agent is then responsible for considering the current state, the set of possible actions, and their corresponding rewards for defining the best action for the problem. That is the action that provides the greatest reward: the one that will maintain a more homogeneous use of the topology links.

4.2.4 Deep Q-Learning Agent

A significant limitation of Q-learning is that it only works in environments with discrete and finite state and action spaces. One solution for extending Q-learning to more

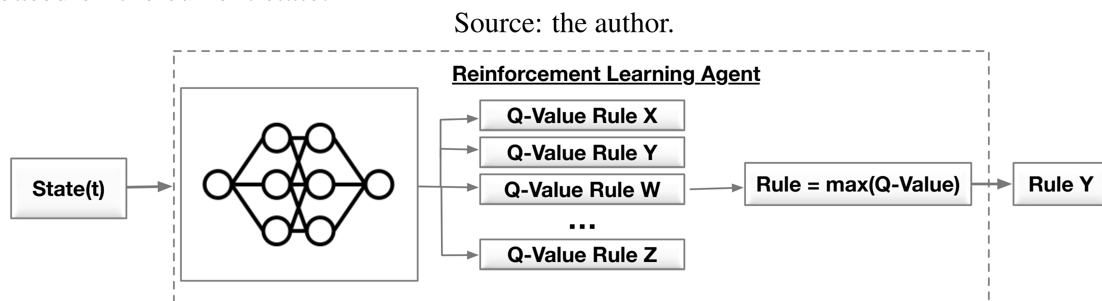
complex environments is to apply function approximators to learn the value function. This function would take states as inputs instead of storing the full state-action table - often infeasible. Since deep neural networks are powerful function approximators, several works in the literature often adapt neural networks for this role. This technique is known as Deep Q-Network or Deep Q-Learning, as discussed in Chapter 2 (YU et al., 2018).

Since the number of bits in each port is unbounded, there is a possibly infinite (but countable) number of possible states in the system. For this reason, applying tabular versions of standard reinforcement learning algorithms, such as tabular Q-Learning (SUTTON, 2018), is infeasible given that we cannot store a Q-table defined over infinite states.

Therefore, we rely on a Deep Q-Learning agent (YU et al., 2018), which uses a neural network to approximate the Q-value for the set of possible actions based on the current state. The current state is the input value for this neural network. The output is the Q-Value for the set of possible actions so that the agent can choose the most suitable one, as illustrated in Figure 4.5.

Figure 4.5: Diagram showing how Deep Q-Learning approximates Q-value for the actions based on the current state.

Source: the author.

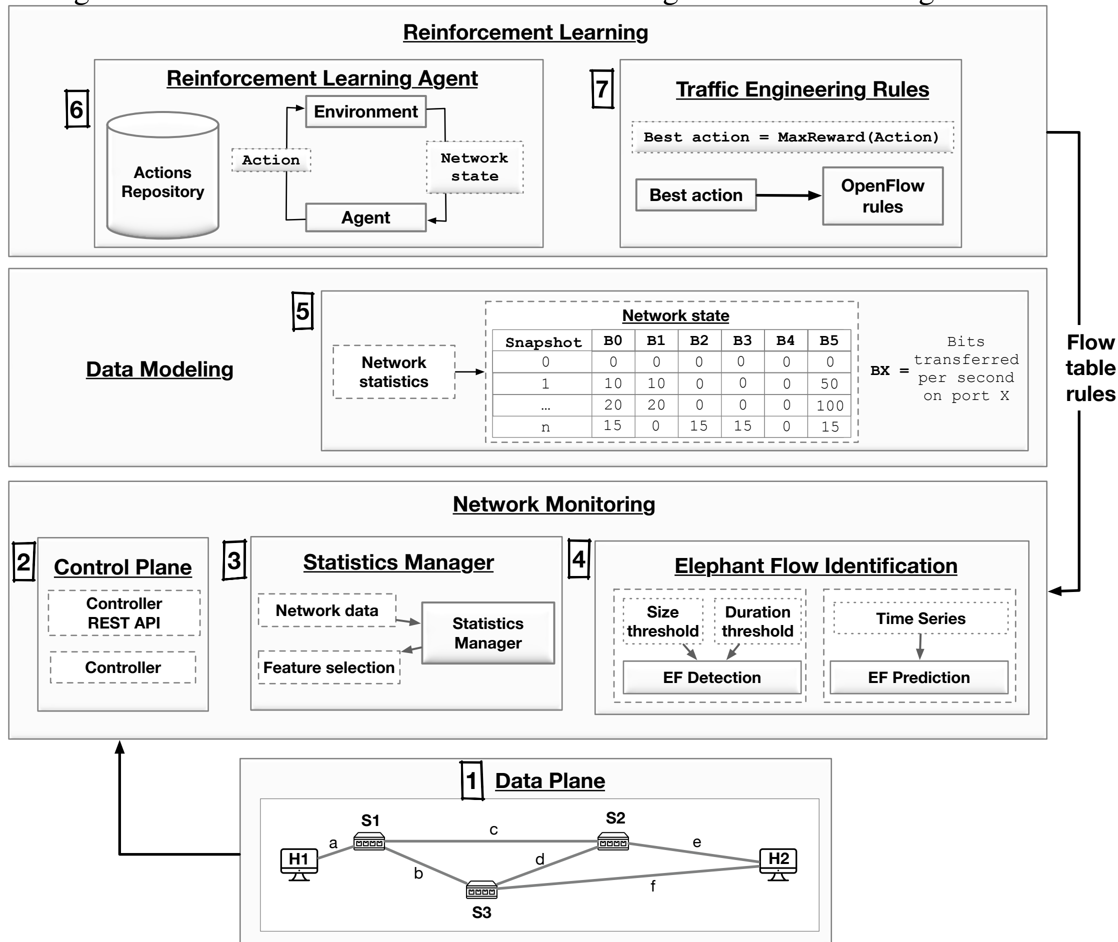


4.3 Look-Ahead Reinforcement Learning Architecture

Based on what we discussed in previous chapters and the definitions discussed in Section 4.2, we developed the architecture illustrated in Figure 4.6. This figure illustrates the main modules that make up the proposed technique: (1) Statistics Manager, (2) Elephant Flow Identification, (3) Reinforcement Learning Agent, and (4) Traffic Engineering Rules. Together, these modules represent an architecture capable of collecting, processing, and analyzing network data to predict whether a flow represents a risk to load balance. In case of an affirmative answer, an RL agent would be responsible for, based on the current network state (composed of collected statistics), suggest the best action for

this specific flow. Lastly, the Flow Action Translator module would get this action and translate it into rules that the control plane would understand and install. In the following subsections, we discuss each of these modules.

Figure 4.6: Look-Ahead Reinforcement Learning for Load Balancing architecture.



Source: the author.

4.3.1 Statistics Manager

The Statistics Manager is responsible for asking for a network snapshot periodically, for example, every 10 seconds. However, we can adjust this frequency according to each network infrastructure's needs. This snapshot comprises network features such as active flows, bits received by switch port per second, and bits transferred by a switch port per second. Next, these statistics are grouped by flow and stored.

We consider flow as a 5-tuple consisting of the source IP address, destination IP address, TCP/UDP source port, TCP/UDP destination port, and transport protocol identification. It is important to emphasize that we can also adjust this tuple according to

Table 4.3: Network statistics stored at each network snapshot, considering an active flow between links *a*, *b*, *f*, and *i*.

Snapshot (t)	Switch Port	Bits-transferred-per-sec
0	S1.1	10 Mbits
	S1.2	10 Mbits
	S1.3	0
	S2.1	10 Mbits
	S2.2	0
	S2.3	0
	S2.4	10 Mbits
	S3.1	10 Mbits
	S3.2	10 Mbits
	S3.3	0
	S3.4	0
	S4.1	0
	S4.2	0
	S4.3	0
	S5.1	0
	S5.2	0

each network infrastructure's needs. We could also adjust the requested statistics according to the network needs. However, we only consider statistics derived from switch flow tables and controller information. Regarding flow table statistics, our proof-of-concept prototype only considers bits per second received by each switch port.

Table 4.3 illustrates how these statistics are stored. For each snapshot, we store the number of bits received per second by each switch port. In the case illustrated in the table, consider that at time 0, we have got the first snapshot of a flow originated on H1 targeting H2. This flow took the path from links *a*, *b*, *f*, and *i* - Figure 4.3. The first snapshot registered the flow transferring 100 Mbits through ports S1.1, S1.2, S2.1, S2.4, S3.2, and S3.1

These statistics will be used in the following stages (Elephant Flow Identification and Reinforcement Learning Agent) to identify elephant flows and model the problem as a reinforcement learning problem.

4.3.2 Elephant Flow Identification

The Elephant Flow Identification is a component that attempts to determine whether a flow is an elephant or not. A flow is considered an elephant flow if it is a large continuous flow that can occupy an unbalanced share of the total bandwidth over time - that is, if

it has a long duration and generates a significant amount of traffic (GUO; MATTA, 2001). With this in mind, elephant flow identification is crucial when optimizing network usage resources and performance. In this context, managing elephant flows involves adequate identification and eventually rerouting such flows to more appropriate locations, minimizing the possible negative impact on the other flows (KNOB et al., 2017). Previous studies show that detecting and rerouting elephant flows effectively can lead to a 113% improvement in aggregate throughput compared with the traditional routing (LIU et al., 2017). With this in mind, this module is responsible for identifying if a flow is an elephant flow. One could use two different approaches for this identification: elephant flow prediction or elephant flow detection.

Elephant flow prediction identifies this kind of flows in the early stages. To achieve this, one could use traffic prediction techniques, as discussed in Chapter 2. In most cases, these techniques would use the statistics gathered by the Statistics Manager. The problem with this technique is that we need a controlled environment to correctly predict, in the early stages, if a given flow will be an elephant flow. This would be a good approach for applications where we have an expected behavior, such as datacenters. For instance, a data center that has scheduled backup processes every Friday 10 PM. In this case, an elephant flow prediction would be helpful. Otherwise, we would only rely on network statistics without having the history or expected traffic behavior.

On the other hand, elephant flow detection identifies this kind of flows when they are already happening (LIU et al., 2017). According to network necessities, it is possible to identify elephant flows by defining volume and size thresholds for these flows. For instance, the network operator could determine that the threshold for a flow to be considered elephant is 10MB for size and 10 seconds for time. In this context, whenever a flow reaches the size of 10MB or has been active for at least 10 seconds, it would be considered an elephant flow, similar to research efforts presented in Silva et al. (SILVA et al., 2018).

We base the decision of whether to use elephant flow detection or prediction on network needs. The architecture presented in this work accepts either one. However, the final output should be an indication of whether this flow could harm the network balance or not. Thus, if the current flow is considered an elephant flow, its features would be the input for the next stage: the reinforcement learning agent. This stage will consult the agent's best action for the current network scenario.

4.3.3 Reinforcement Learning Agent

This module is responsible for gathering all data and format it as a reinforcement learning problem. In this step, we use the model defined in Section 4.2:

- **State** (S): current network usage, represented by the number of bits transferred by each switch port on the topology.
- **Set of actions** (A): set of all possible actions to install a path from H1 to H2. More specifically, the set of all possible rules necessary to install a path from H1 to H2. We obtain all these possible paths by consulting the controller.
- **Reward** (R): $Reward(s_t)$ is a function of S_t.

The reinforcement learning agent is responsible for running its learning algorithm and selecting the best action for the current state. More specifically, this module receives the current state as input, normalizes the state values and outputs the best action to apply on the network.

To choose this action, the agent runs the algorithm with its pre-defined parameters and selects the action that provides the best reward. Next, we send this action to the Flow Action Translator.

4.3.4 Traffic Engineering Rules

This module is responsible for translating the selected action to a flow rule and installing it in the flow table of a subset of switches. This module identifies which switches are related to the chosen action and maps to flow routing rules. After installing a rule, we observe its impact in the new network state (in terms of how homogeneous link usage is) and calculate the reward for this new state, which will indicate whether the action contributed to our goal or not.

The input for this module is the chosen rule, and the output is the payload used for installing this OpenFlow rule through the Controller. Figure 4.7 illustrates an example of this payload, and Table 4.4 illustrates the meaning of each parameter in this payload.

This step is the final step of the cycle that we repeat according to the network needs.

Figure 4.7: Output generated by Flow Action Translator: OpenFlow rule for switch S1

```
{
  switch: S1,
  name: flow-match,
  priority: 32768,
  ingress-port: 1,
  active: true,
  actions: output=3
}
```

Table 4.4: Set of parameters set for each switch as the output of Flow Action Translator module.

Parameter	**Description**
switch	ID of the switch (data path) that this rule should be added to
name	Name of the flow entry. This is the primary key, so it must be unique
priority	The priority of this rule. Maximum value is 32767, which indicates that this rule cannot be overwritte
ingress-port	Switch port on which the packet is received
active	Indicates if this is an active rule
actions	Specify multiple actions using a comma-separated list (<key>=<value>). Specifying no actions will cause the packets to be dropped

4.4 Summary

In this chapter, we discussed the architecture proposed in this work. This architecture comprises the following modules: Statistics Manager, Elephant Flow Identification, Reinforcement Learning Agent, and Traffic Engineering Rules. This model is used to create a prototype and execute experiments to evaluate the architecture proposed in this work.

5 PROTOTYPING AND EVALUATION

In this chapter, we discuss the prototype and experimental results. In Section 5.1 we explain the prototype developed, in Section 5.2 we present the experiments configuration, and in Sections 5.3 and 5.4 we discuss the experimental results.

5.1 Prototyping

Considering the proposal of our work, we highlight the following as the main goals of the prototype.

- To validate the model proposed for reinforcement learning - discussed in Section 5.3;
- Evaluate if the model achieves load balancing and which reward function yields satisfactory results;
- Evaluate if the additional machine learning step we propose (look-ahead) could yield a more intelligent agent and, hence, better network resource usage for balancing the workload;
- Analyze the scalability of the agents.

In the context of this work, a flow is represented by the tuple `<ip_src, ip_dst, port_src, port_dst, protocol>`, where `ip_src` and `ip_dst` are the IP adresseses source and destination, respectively, `port_src` is the source port and `port_dst` is the destination port; protocol indicates the protocol.

For prototyping, we used the Mininet VM to emulate the SDN topology, Floodlight controller on version 1.2, and a docker application with Python 3.7 to run the application with statistics collection, elephant flow identification, and reinforcement learning agent. In the following subsections, we explain how we implemented each of these components work.

Following, we discuss the main stages of prototyping. Section 5.1.1 details the Statistics Manager module. Section 5.1.2 is about how the Elephant Flow Identification module works. Section 5.1.3 describes the reinforcement learning agents used. Finally, Section 5.1.4 details the Flow Action Translator module.

5.1.1 Statistics Manager

To collect statistics from the switches, we used the Floodlight Statistics module[1]. By default, Floodlight starts a thread that updates the switch ports statistics every 10 seconds. We changed this value to every 5 seconds to emulate a real-time application. In this context, we use Floodlight REST API endpoint for bandwidth `/wm/statistics/bandwidth/<switchId>/<portId>/json`. We collect the `bits-per-second-tx` of each switch port to update our statistics. We call this endpoint every time we need to update the network snapshot and derive the environment state (discussed in Section 5.1.3.2).

5.1.2 Elephant flow identification

Considering that we do not have historical data or pre-defined behavior of the workload in our network, we chose to use elephant flow detection for prototyping. We used a size threshold defined by the parameter `ef_threshold`.

5.1.3 Reinforcement Learning Agent

For the reinforcement learning agent, we developed an OpenAI gym environment[2] that would allow us to test our agent with different reinforcement learning algorithms. We chose this tool because it is widely used among works with Machine Learning research (MNIH et al., 2013).

In our experiments, we chose the algorithms made available by Stablebaselines[3]. An OpenAI gym environment is essentially composed of four components: a step function, an observation space, an action space, and a reward function, discussed next.

5.1.3.1 Step function

We show the pseudocode of a reinforcement learning agent on Listing 5.1 to understand better how the step function works.

[1] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API
[2] https://gym.openai.com
[3] https://stable-baselines.readthedocs.io/en/v2.3.0/guide/quickstart.html

```
model = loadTrainedAgent()
state = env.getState()

for step in range(timesteps):
    action = model.predict(state)
    state, reward = env.step(action)
```

Listing 5.1 – Reinforcement learning algorithm pseudo-code.

As shown in the excerpt above, the agent only looks to the current state and chooses the best action. The step function only accepts the $action$ parameter, applies it to the environment, and returns the updated state and the generated reward. This means that using the model proposed in Chapter 4 we can not apply an action for a specific flow (call `env.step(action, flow)`). Instead, the agent should look to the state and identify which flow should be rerouted (`env.step(action)`). We discuss how we adapt our model to this implementation in the action space subsection.

5.1.3.2 Observation space

The observation space is the state, which in our case is the network state: the number of bits transferred by each switch port. In the context of this work, we want to evaluate how network resources (switches and links) are used. Thus, we use switch port statistics to indicate how much data each switch port is transferring. For example, considering S1 Topology (Figure 5.1) the state is a 16-dimensioned array containing the `bits-per-second-tx` of each switch port. Using the Statistics Manager module, this state is updated: we collect the statistics using the Floodlight Statistics module and model it as the expected array.

5.1.3.3 Action space

As discussed in Chapter 4, the ideal approach would implicate using the step function as it follows on on Listing 5.2.

```
model = loadTrainedAgent()
state = env.getState()

for flow in active_flows:
  if isElephantFlow(flow):
    action = model.predict(state)
    state, reward = env.step(action, flow)
```

```
8    else:
9      continue
```

Listing 5.2 – Reinforcement learning algorithm pseudo-code choosing an action for all active flow considered elephant.

However, as we showed in Section 5.1.3.1, the `step` function only accepts an action for an already defined flow. To overcome this issue, we adapt our model to consider the rule and the flow for which the rule will be installed. This means that our new action for this prototype will be the tuple `<switch_id, in_port, out_port, flow_match`. This new model implicates having a pre-established set of flows so that each agent can install an OpenFlow rule. Thus, we consider the network can manage a maximum of a given number of concurrent flows - we call this parameter `max_sim_flows`.

Besides, we had to adapt pre-processing a flow to identify if it is an elephant or mice flow only to apply actions for elephant flows. Because we can no longer do this, we use the reward function to penalize actions that act on mice flows.

Furthermore, considering that a continuous action space is a more challenging problem due to computational resources and available algorithms, we implemented this action space as a discrete and finite action space. This means that we mapped every possible path and every possible rule that had to be installed for these paths. Next, we mapped these rules (tuple configuration) to a number.

Lastly, note that we need to apply the action for calculating the reward of a given action, installing the rule. So, besides installing a rule, the step function waits for a given time (in seconds) to collect the resulting state and then calculate the generated reward. This value is set by the parameter `install_wait_time`. We have to wait so the controller can install new rules, and these rules reflect on the traffic. After this time, we evaluate the new state with the reward function, as we discuss next.

5.1.3.4 Reward function

As discussed previously, in the context of the OpenAI gym environment, the agent should only observe the state's environment and choose the best action for it. We cannot observe the state and choose the best action for a given flow match. Consequently, we understand that it is not possible to add an extra step before calling the reinforcement learning. Instead, we use the reward function to penalize the agent when it chooses to install rules for mice flows because we advocate that we should only use the reinforcement

learning actions to act on elephant flows. In this context, we adapt our reward function to this behavior and expect that the agent will learn to concentrate on routing elephant flows after a few iterations. We introduce the Elephant Flow Intelligence (EFI), which will be summed to the reward function. The EFI factor is a parameter defined for each experiment, and it is essential because it is how we penalize the agent for choosing actions for mice flows. Thus, the final reward functions used in this prototype have the format presented in Equation 5.1.

$$Reward(S_t) = f(x) * \mu + EFI \tag{5.1}$$

Lastly, for our prototype, we used the Deep Q-Learning algorithm from Stable Baselines[4]. We chose this algorithm because literature has shown remarkable results with it (MNIH et al., 2013).

5.1.4 Flow Action Translator

This is the final step in our architecture. Here, we translate the chosen action to an OpenFlow rule, using the tuple `<switch_id, in_port, out_port, flow_match>`. We describe an example rule below, as Floodlight Controller expects.

```
{
    "name": "rule_name",
    "switch": "S1",
    "active": "true",
    "eth_type": "0x0800",
    "ipv4_src": "10.0.0.1",
    "ipv4_dst": "10.0.0.2",
    "ip_proto": "0x06",
    "tcp_src": "46110",
    "tcp_dst": "5201",
    "actions": "output=3",
    "idle_timeout": "60",
    "hard_timeout": 10
}
```

The values of `idle_timeout` and `hard_timeout` are parameters defined for each experiment. As explained before, after sending the install request to the controller,

[4]https://stable-baselines.readthedocs.io/en/master/modules/dqn.html

we wait for a given amount of seconds to check the resulting state. The resulting state is then presented to the agent, and the cycle continues.

5.2 Experiments configuration

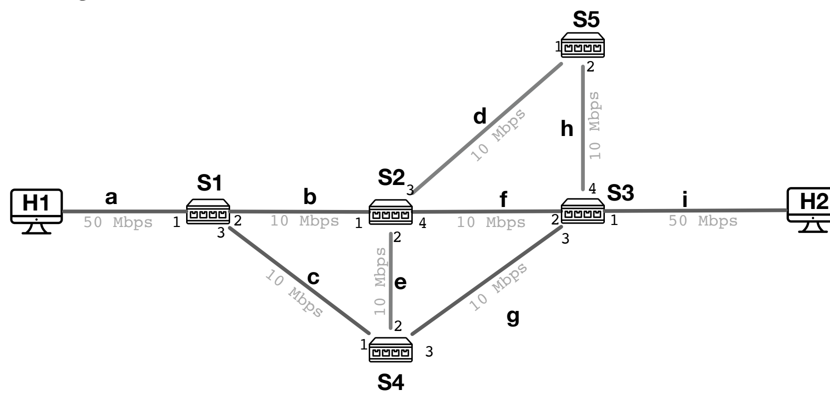
In this section, we detail the experiments configuration. Firstly, network topologies used during our analysis, followed by the environmnet setup for running the experiments, and lastly, the parameters used for the experiments.

5.2.1 Network topologies

For these experiments, we consider a specific topology to facilitate modeling, testing, and evaluation. We discuss the chosen topologies in this section.

Figure 5.1 presents the baseline topology used for modeling this prototype. We chose S1 topology as the baseline topology for the experiments because it has fewer switches, resulting in minor action and observation spaces. This topology was created to facilitate listing all possible paths between the two hosts and consider the possibility of inserting rules that could generate a loop. Note that 50 Mbps is the highest bandwidth, located at the topology ends, to induce a bottleneck effect.

Figure 5.1: Topology S1 used as baseline for experimental analysis. H1 is the source host, and H2 is the target.

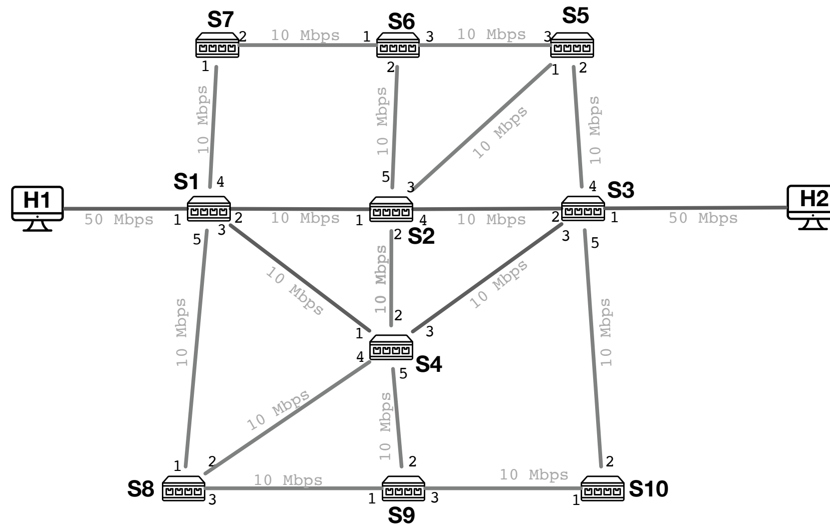


Source: the author.

We also consider a second topology in our prototype. This second topology was used to evaluate the impact of adding extra switches in the topology. We call this second topology S2, illustrated in Figure 5.2. We kept the 50 Mbps end links to reproduce the bottleneck effect on a larger scale. We discuss the result of the experiments using these

topologies in Sections 5.3 and 5.4.

Figure 5.2: Topology S2 used to analyze the impact of more switches on experimental results. H1 is the source host, and H2 is the target.



Source: the author.

These simplifications allowed us to use a broader range of algorithms to test our agent. However, it is essential to highlight that we should use a continuous action space for a dynamic topology.

5.2.2 Environment setup

We executed all experiments presented in this chapter on a MacBook Pro Mid 2014, processor 2,6 GHz Dual-Core Intel Core i5, with 8 GB 1600 MHz DDR3 memory. To reproduce the experiments presented in this chapter, one would need the components listed below.

- **Mininet VM**[5];
- **Floodlight Controller**[6] on version 1.2, running on Mininet VM;
- **Flow Switch Monitor**[7] debugging project;
- **Docker Python3.7 image**[8] (with all necessary dependencies);
- The **Look-Ahead RL application**[9], running on the Docker Python3.7 image.

[5] http://mininet.org/download/#option-1-mininet-vm-installation-easy-recommended
[6] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview?homepageId=1343545
[7] https://github.com/guilhermealles/floodlight-switch-monitor
[8] https://github.com/ippossebon/docker-look-ahead-rl
[9] https://github.com/ippossebon/floodlight-api-look-ahead-rl

To reproduce the experiments we presented, one needs to set the environment with the dependencies listed above and follow the steps below.

1. On Mininet VM, run the Floodlight controller version 1.2;
2. On Mininet VM, run Mininet with the desired topology (S1 or S2, both under the directory `floodlight-api-look-ahead-rl/topologies/`;
3. Initialize flow statistics adding static entries that send the flows to the controller. This can be done by REST API requests to Floodlight REST API. The data for these requests is available at `floodlight-api-look-ahead-rl/initial_flow_entries.csv`;
4. For running the experiments considering the topology **S1**, on Docker Python3.7 image, use the script `floodlight-api-look-ahead-rl/run-experiments-s5.py`. On the other hand, for considering *S2* topology, use the script `floodlight-api-look-ahead-rl/run-experiments-s10.py`;
5. Create traffic between the hosts (e.g. using `iperf`);
6. Use `floodlight-switch-monitor` to analyze switch traffic.

Since all mentioned projects are open source, further details can be found at all repositories `README.md` file.

5.2.3 Experiments setup

Considering the model defined on the previous chapter, Table 5.1 shows the parameters configuration for this model on our experimental evaluation.

Table 5.1: Set of different parameters used on the model and its correspondent values.

Parameter	**Value**
`max_sim_flows`	16 flow matches
`install_wait_time`	7 seconds
`ef_threshold_duration`	10 seconds
`ef_threshold_size`	100 MBytes
`idle_timeout`	60 seconds
`hard_timeout`	10 seconds
`EFI`	100

The number of simultaneous flows the agent knows (`max_sim_flows`) was set to 16 due to time restriction. Considering the link bandwidth on the network topologies,

we needed to have a significant amount of mice and elephant flows that would finish in a reasonable amount of time. Our time was limited because we understood the need of reproducing each set of experiments several times to observe the variation of the results. Also, because Floodlight only provides statistics for previously installed flows, our prototype had to initially install the set of flow matches that would be used on our experiments. More specifically, we install an initial rule for each flow match the controller will analyze that would send this flow match through the default route suggested by Floodlight. This means that for a flow routed through a path of n switches, we would have to install *n* rules. The total number of initial rules to be installed in the agent initialization would be $n * m$, where m would be the number of flow matches. Since the topologies we used for these experiments had 5 and 10 switches ($n = 5$ and $n = 10$), to minimize the initial setup time and still evaluate the model with elephant flows, we considered 16 simultaneous flows appropriately value.

As explained previously, Floodlight has a thread for collecting statistics periodically. We set this time to be every 5 seconds because we wanted to get the most updated values possible - to have an accurate network state. We tested this thread getting statistics every 1 and 2 seconds, but we did not get satisfactory results: the counters were inaccurate. However, when setting this time to 5 seconds, we were able to get accurate results for our experiments, so we set Floodlight to update the statistics counters every 5 seconds. Because we needed some time between installing a rule, routing packets through this rule, and observe its impact on the network statistics, we added a 2 seconds threshold, resulting in a total `install_wait_time` of 7 seconds.

For the elephant flow duration threshold (`ef_threshold_duration`), we used the same value used on SDEFIX (KNOB et al., 2016) and IDEAFIX (SILVA et al., 2018). Because we consider a 10 seconds duration and the topologies used for the experiments had the majority (bottlenecks) links with 10 Mbps capacity, we considered (`ef_threshold_size`) as 100 MBytes. We considered this value appropriate as threshold size because we also needed the flows to be (i) big enough to observe the impact of the agent's choices and (ii) small enough considering the time restriction we had to run experiments. More specifically, regarding the time needed to complete the experiments, we had two major factors: `install_wait_time` and the number of timesteps needed for the agent to complete its task. Considering `install_wait_time` as 7 seconds, in the worst-case scenario, we would need $7 * num_timesteps$ to complete an experiment. Yet, the number of timesteps needed depends on the number of flows and the size of these

flows - the higher the values, the higher the number of timesteps. Preliminary experiments showed that we needed an average of 700 timesteps to complete all five flows (considering 5 flows with the same total flow byte count - 10 MBytes, 50 MBytes, 100 MBytes, 200 MBytes, and 500 MBytes) for an experiment configuration using a reinforcement learning agent. This implicates that we needed $7*700 = 4900$ seconds to complete one experiment - for the worst case. Considering at least five replications for this experiment (to analyze mean and standard deviation values), we needed $4900 * 5 = 24500$ seconds to complete one experiment with its replications. Moreover, considering that we executed these experiments for 22 scenarios (for functional evaluation: 3 agents for functional evaluation + 1 baseline, and for EFI evaluation: 2 agents $*3$ workloads $*3$ intervals $= 18$ scenarios), our worst-case scenario involved needing $24500 * 18 = 441000$ seconds $= 122,5$ hours. This assuming that none of the experiment scenarios would have to be executed again due to some inconsistency or error. In short, considering all these factors, a flow is considered an elephant flow based on a `ef_threshold_size` of 100 MBytes.

The `idle_timeout` parameter is used for installing a rule on a switch. This value indicates how long a rule should remain active after it is no longer being used. The default value for this parameter is 0, but we set this value to 60 seconds for all rules installed. We use this value because we want to make sure the agent is learning the best rules for our goal. Because it is not feasible to increment the priority of every new rule (so this would be used instead of the previously installed rule for the exact flow match), we use the timeout as an alternative. Suppose the agent understands this rule as an action that will collaborate to more homogeneous resource usage. In that case, the agent will keep choosing this action and renovating its active time. If not, the rule would be removed because it is no longer being used.

The `hard_timeout` parameter is also used for installing a rule on a switch. However, this value indicates for how long this rule is valid, despite being used or not. The default value for this parameter is also 0, but we set it to 10 seconds for one rule: the controller rule. The controller rule should only be chosen by the agent in extreme cases: loop control. This means that if the agent could not learn a good path for a given flow and, instead, installed rules that generate a loop on the network, the controller rule can be used to recover from this loop state, as explained previously. Because we understand this rule should only be used in extreme cases (due to its overhead of sending packets to the controller), we set a hard timeout of 10 seconds, which would be enough time for a flow to recover from a loop state. We also tested this rule with 5 and 20 seconds, but with 10

seconds, we got the best results.

Lastly, `EFI` parameter is used for penalizing the agent when it chooses actions for mice flows, as explained previously. The value used in our experiments was $EFI = 100$ because of the range of possible reward values. More specifically, considering the values on Table 5.1 and that the base reward function used for EFI evaluation experiments was harmonic mean usage, we have the maximum reward value explained next.

Consider $maxHMean$ as the maximum harmonic mean usage value for topology S1, which was calculated considering the usage of all links. $maxHMeanReward$ is the maximum value a state could yield using this reward function.

$$maxHMean = 0.05292$$

$$maxHMeanReward = 5.29$$

For EFI experiments, the minimum reward value was set to $minEFIReward = -1 * maxHMeanReward$, which in this case would be $minEFIReward = -5.29$. We set this as the minimum value because experiments using the minimum value calculated by the harmonic mean usage heuristic were not low enough for the agent to learn how bad this state would be. Considering that we had rewards varying from -5.29 to 5.29, the EFI factor should be high enough to potentialize the choice of using an action for an elephant flow. That is why we used $EFI = 100$, so the reward values could vary from -5.29 to 105.29.

Considering we analyze three different reward functions, we trained a different agent for each reward function, resulting in the agents described in Table 5.2. We describe the training parameters for each of these agents on Table 5.3.

Table 5.2: Set of different agents trained for the experiments.

Agent	**Flows for training**	**Learning rate**	**Gamma**	**Initial Epsilon**	**Epsilon decay**	**Final Epsilon**	**Buffer Size**	**Batch Size**	**Number of timesteps**	**Reward function**	**Topology**	**Training time**
WeightedUsage t	5	0.05	0.95	1.0	0.9	0.01	56	50	700	Weighted usage	S1	01:03:06.168
UsageHarmonicMean	5	0.05	0.95	1.0	0.9	0.01	56	50	700	Harmonic Mean	S1	00:33:07.165
UsageStandardDeviation.	5	0.05	0.95	1.0	0.9	0.01	56	50	700	Standard deviation	S1	00:34:00.839

Table 5.3: Set of different parameters for training the agents.

Parameter	**Description**
Learning rate	Parameter used in neural networks training that controls how quickly the model adapts to the problem
Gamma	Discount factor
Initial epsilon	Intial probability of agent choosing a random action
Epsilon decay	raction of entire training period over which the epsilon is annealed
Final epsilon	Final probability of agent choosing a random action
Buffer size	Size of the replay buffer used to enhance the performance training
Batch size	Size of a batched sampled from replay buffer for training
Timesteps	Considering a continuous learning, indicates the number of timesteps for running the algorithm

5.3 Functional analysis

This first set of experiments aims to evaluate if an RL agent can effectively balance workload, considering the different reward functions discussed earlier, namely: *weighted usage*, *usage harmonic mean*, and *usage standard deviation*. We compare the performance of these variations with a baseline approach, which is the use of the Floodlight controller without using any load balancing method. In this set of experiments, we consider $EFI = 0$.

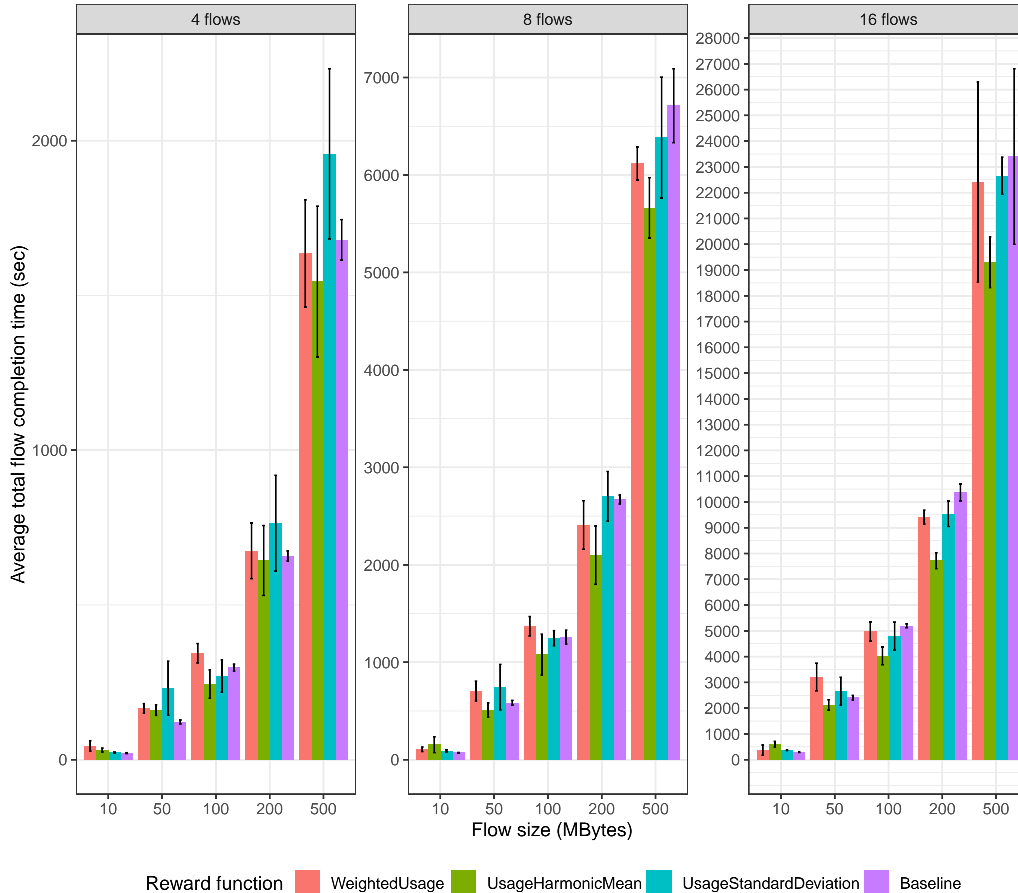
To compare these alternatives, we consider the total flow completion time (FCT) as the metric we want to minimize. Intuitively, when elaborating these reward functions, we want more switches being used to carry all traffic sent by the source to the destination. Splitting traffic among different paths would then use more resources for less time, yielding lower total flow completion time. We run each experiment configuration five times to analyze average and standard deviation values. We calculate total flow completion time as the sum of all flows completion time. We use this metric to evaluate (i) if the agent can balance workload, (ii) if there is one reward strategy that can balance this workload better compared to a default route provided by the Floodlight controller, and (iii) scalability in terms of the number of flows. On the other hand, memory usage is the metric we use to evaluate the computational costs of each agent. We illustrate the configuration of these experiments on Table 5.4. We illustrate the results of these experiments regarding total flow completion time in Figure 5.3, and memory usage in Figure 5.4.

Table 5.4: Experiments configuration for functional analysis.

Agent	**Reward Function**	**Number of EF**	**Topology**
WeightedUsage	Usage heuristics	5	S1
UsageHarmonicMean	Harmonic mean	5	S1
UsageStandardDeviation	Standard deviation	5	S1
Baseline	-	5	S1

In our work, we consider effective a load balance strategy if this strategy could use network resources for less time - that is, if the total flow completion time is less than an approach that was not using a load balance approach. In the context of our experiments, Floodlight Controller was not using any load-balancing method, and all flows were using the same default route. Intuitively, this would result in a higher total flow completion time because all flows were disputing for the same resources. Splitting traffic among different paths would then use more resources (paths, i.e., switches) for less time, which would

Figure 5.3: Average total flow completion time results for experiments with agents *WeightedUsage*, *UsageHarmonicMean*, *UsageStandardDeviation* and Floodlight Controller. Error bars represent the standard deviation for each set of replications. Each set of experiments considered 5 flows with the same size (10 MB, 50 MB, 100 MB, 200 MB, and 500 MB, respectively).



Source: the author.

yield a shorter total flow completion time.

4 simultaneous flows. The agents were not able to balance the workload effectively - except for 4 simultaneous 100 MB flows, which had an average total flow completion time shorter than the default Floodlight route. A possible explanation for these results is that the overhead of choosing the best action (a rule that could generate a state of homogenous resource usage) does not pay off for small flows. That is, we need bigger flows to compensate for the time spent chosing the best action.

8 simultaneous flows. Regarding flows smaller than 200 MB, the agents were not able to provide shorter total flow completion time. In fact, for this set of experiments, the default route for every flow simultaneously on the network was faster than using the rules chosen by the agents. Again, we believe this is because these flows are too short to overcome the overhead of eventual explorations of the agent - which might need some

time to recover from evetual failures. When we look to 8 simultaneous 500 MB flows, on the other hand, we see that the default Floodlight route was slower than the resulting paths chosen by the agents. Specially agent *UsageHarmonicMean*, which was able to get the fastest combination of paths, resulting on the shorter average total flow completion time.

16 simultaneous flows. Considering this set of experiments, the threshold for agents yielding better results was shorter: 100 MB. Flows bigger than 100 MB were able to use the agents to be rerouted to paths that generated a shorter total completion time - again, we highlight agent *UsageHarmonicMean*, with the shortest total flow completion time. Once again, we attribute these results to the fact that smaller flows are not able to overcome the overhead of chosing the best action and cope with any failures.

Considering the results shown in Figure 5.3, we consider agent *UsageHarmonicMean* as the best agent for solving the problem proposed in this work. This agent chose the best set of paths that generated the fastest routes and, consequently, the shortest average total flow completion time for flows bigger than 100 MBytes. We believe this might be associated with the fact that the difference of rewards between a good and a bad state is lower when using UsageHarmonicMean than when using the other rewards functions. For instance, consider a homogeneous state s0 on topology S1, illustrated in Table 5.5 - this would happen in a case where two 5 MB flows would use the route *a-b-f-i*. Also, consider a heterogeneous state s1 on the same topology, illustrated on Table 5.6 - this would happen in a case where two 5 MB flows would use each one of the two routes: *a-b-f-i* and *a-c-g-i*. The reward function for each agent is illustrated on Table 5.7, where we illustrate the difference between a good and a bad state on the column *Difference(s0,s1)*, and *UsageHarmonicMean* has the lowest value. However, we also believe additional experiments could help understanding if this difference impacts the results.

Table 5.5: Homogeneous state representation.

Switch port	S1.1	S1.2	S1.3	S2.1	S2.2	S2.3	S2.4	S3.1	S3.2	S3.3	S3.4	S4.1	S4.2	S4.3	S5.1	S5.2
Bits transferred	10383360	10383360	0	10383360	0	0	10383360	10383360	10383360	0	0	0	0	0	0	0

Table 5.6: Heterogeneous state representation.

Switch port	S1.1	S1.2	S1.3	S2.1	S2.2	S2.3	S2.4	S3.1	S3.2	S3.3	S3.4	S4.1	S4.2	S4.3	S5.1	S5.2
Bits transferred	10383360	5191680	5191680	5191680	0	0	5191680	10383360	5191680	5191680	0	5191680	0	5191680	0	0

Lastly, we also analyze memory usage to evaluate the scalability of the agent, considering the two topologies discussed. We collected the memory usage using *tracemalloc*[10] Python module.

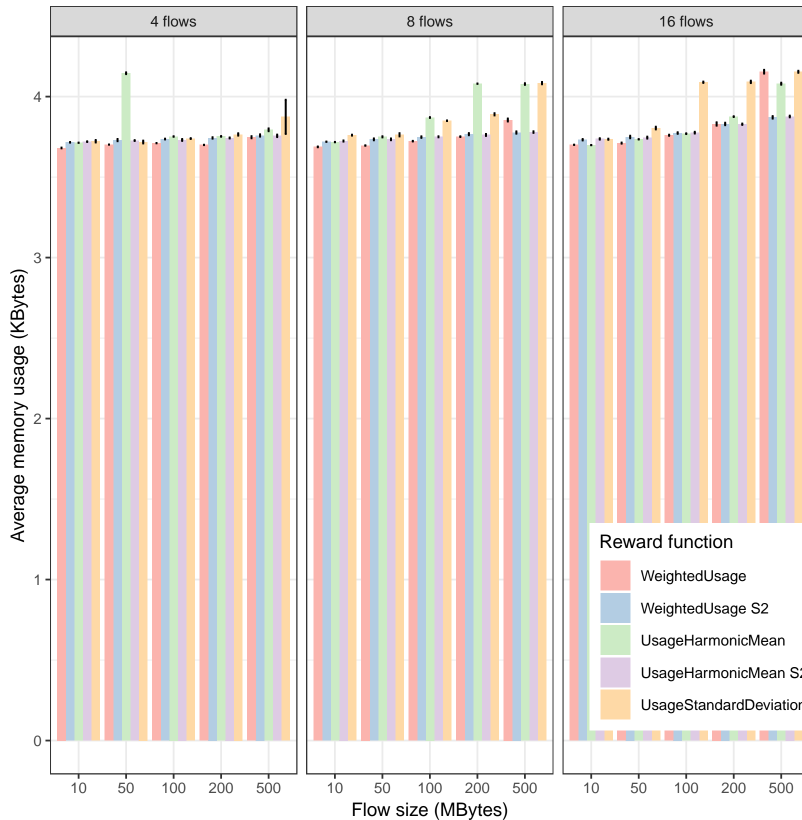
[10] https://docs.python.org/3/library/tracemalloc.html

Table 5.7: Comparison between reward function values for each state.

Reward function	**Reward(s0)**	**Reward(s1)**	**Difference(s0,s1)**
WeightedUsage	$2.58754 * 10^{16}$	$2.58754 * 10^{16}$	$8.306688 * 10^{8}$
UsageHarmonicMean	$3,322675 * 10^{15}$	$5.537792 * 10^{15}$	$2.215117 * 10^{16}$
UsageStandardDeviation	$1.043906 * 10^{16}$	$7.131237 * 10^{15}$	$3.307825 * 10^{15}$

As shown in Figure 5.4, the memory usage as we increase the number of flows is very low. We use more memory for agents that need more steps to complete the flow due to continuous learning. We have some outliers (especially for four flows of 50 MB), showing a higher memory usage because of this - *i.e.,* there were needed more time steps to complete all active flows.

Figure 5.4: Memory usage results for experiments with different reward functions.



Source: the author.

Considering that agent *UsageHarmonicMean* reduced the FCT and is scalable concerning the number of switches, we use this agent as the baseline for the elephant flow intelligence analysis, discussed next.

5.4 Elephant Flow Intelligence (EFI) analysis

The previous analysis showed that the agent with a reward function that uses *harmonic mean* managed to achieve reduced FCT without a significant impact on memory usage. Thus, we use this agent (which we call *UsageHarmonicMean-EFI*) as the baseline for the experiments in this section, assessing the EFI factor. For this analysis, we performed the experiments by analyzing two variables: workload and interval between connections (PIZZUTTI; SCHAEFFER-FILHO, 2018). We used three different workloads: 25/75, 50/50, and 75/25 as mice and elephant flow proportion; three different intervals between connections: 5, 10, and 15 seconds.

Table 5.8: Experiments configuration for EFI analysis.

Agent	Interval	Workload	Topology	Timesteps
UsageHarmonicMean	5	25/75	S1	600
UsageHarmonicMean	5	50/50	S1	700
UsageHarmonicMean	5	75/25	S1	1500
UsageHarmonicMean	10	25/75	S1	600
UsageHarmonicMean	10	50/50	S1	700
UsageHarmonicMean	10	75/25	S1	1000
UsageHarmonicMean	15	25/75	S1	600
UsageHarmonicMean	15	50/50	S1	700
UsageHarmonicMean	15	75/25	S1	1000
UsageHarmonicMean-EFI	5	25/75	S1	600
UsageHarmonicMean-EFI	5	50/50	S1	700
UsageHarmonicMean-EFI	5	75/25	S1	1500
UsageHarmonicMean-EFI	10	25/75	S1	600
UsageHarmonicMean-EFI	10	50/50	S1	700
UsageHarmonicMean-EFI	10	75/25	S1	1000
UsageHarmonicMean-EFI	15	25/75	S1	600
UsageHarmonicMean-EFI	15	50/50	S1	700
UsageHarmonicMean-EFI	15	75/25	S1	1000

We illustrate these experiments configuration on Table 5.8. We executed the experiments with all possible combinations, considering topology S1. Because we did not have enough time, we could not execute the experiments for topology S2 - on average, a 700 timesteps experiment runs in 2 hours. The number of timesteps used for each experiment configuration is based on several executions, where we observed which would be the appropriate number. Note that this number may vary according to the workload and interval between connections: the higher the number of elephant flows, the higher is the number of timesteps necessary to complete all flows. Similarly, the higher the inter-

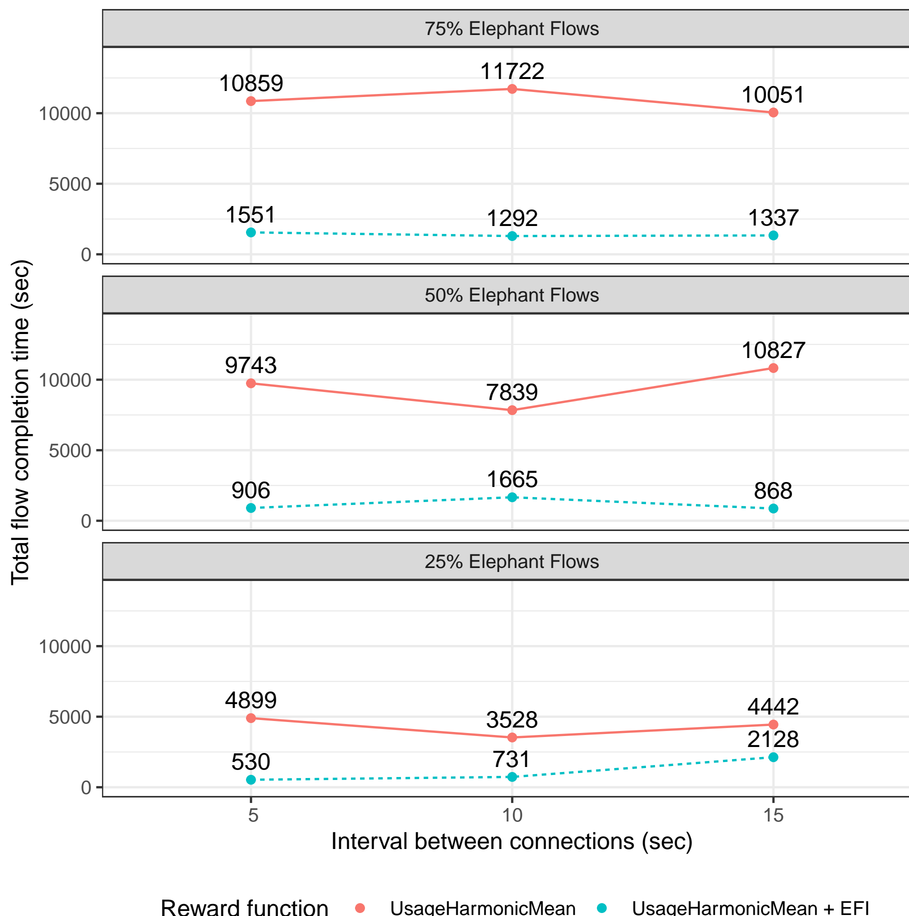
val between connections, the higher the number of time steps. Table 5.9 shows how we distributed the workload.

Table 5.9: Workload distribution, where #MF is the number of mice flows, and #EF is the number of elephant flows.

Workload	#MF	#EF	Flow sizes
25/75	2	6	50 MB, 80 MB, 100 MB, 200 MB, 300 MB, 400 MB, 800 MB, 1024 MB
50/50	4	4	50 MB, 60 MB, 80 MB, 90 MB, 100 MB, 400 MB, 800 MB, 1024 MB
75/25	6	2	50 MB, 60 MB, 70 MB, 80 MB, 90 MB, 95 MB, 100 MB, 1024 MB

The workload values shown in Table 5.9 indicate 25/75 as 25% of the flows in the network as mice flows and 75% as elephant flows. Similarly, 50/50 corresponds to 50% mice flows and 50% elephant flows, and 75/25 to 75% mice flows and 25% elephant flows.

Figure 5.5: Total flow completion time for experiments using agents *UsageHarmonicMean* and *UsageHarmonicMean-EFI*.



Source: the author.

Results shown in Figure 5.5 demonstrate that the elephant flow identification (EFI) factor has a significant influence on reducing total flow completion time. The *UsageHarmonicMean-EFI* agent yielded the lowest FCT in all scenarios compared to the baseline *harmonic*

mean agent. We understand these results as proof that when using RL approaches, the agent should concentrate on elephant flows as they represent the main risk when balancing workload among network resources. The RL agent will concentrate on balancing these flows and ignore the less relevant flows in our approach.

In all scenarios, the impact of using EFI is significant: in the worst case, we can reduce the FCT on 52% (25% workload with a 15 seconds interval between connections). This indicates that even considering the smallest amount of EF and the most extensive time interval between connections, we could reduce FCT when using EFI. Our most suitable scenario was considering a 50/50 workload and a 10 seconds interval between connections, which could reduce FCT by 91%. Still, the scenario where intuitively it would be needed more time (75% of EF workload and 15 seconds interval between connections) yielded excellent results: an 87% reduction on FCT.

6 FINAL CONSIDERATIONS AND FUTURE WORK

This dissertation presented an approach for load balancing using reinforcement learning and analyzing the impact of adding logic to penalize actions that reroute mice flows instead of actions that reroute elephant flows. We discuss our final considerations in Section 6.1 and future work on Section 6.2.

6.1 Final considerations

We advocate that using RL techniques to reroute mice flows can be inefficient because these are low-impact flows in terms of network usage, and the overhead for managing them would not pay off. Thus, we proposed a two-step approach based on reinforcement learning and network traffic prediction to balance network flows and ensure efficient network resource usage. The first step is network traffic prediction and is used to determine which flows have the highest impact on network resources and may cause network imbalance. The second step is composed of a reinforcement learning agent and is used to re-establish this balance, focusing on making the best usage of resources given the current state of the network.

Our main contributions are (i) problem modeling as a function of states and actions in a system that aims to balance network traffic and (ii) an architecture that more judiciously uses reinforcement learning on flows of interest for load balancing. We advocate that this two-step machine learning approach, which we call look-ahead reinforcement learning, can reduce human errors regarding the management of the network by preventing unnecessary interactions.

We evaluate our research using two analysis groups: functional analysis and elephant flow intelligence analysis. For the first analysis, we consider the same model with three different reward functions: network usage heuristics, harmonic mean heuristics, and standard deviation heuristics. We then evaluate whether the proposed model could balance the workload among the network and, if so, which of the reward functions would give better results. The second analysis consisted of evaluating the impact of adding an Elephant Flow Intelligence (EFI) to the reward function of this model. This modification would allow the agent to receive higher rewards when choosing a rule for an elephant flow and lower rewards when choosing a rule for a mice flow.

Experimental results show that in all scenarios, the impact of using EFI is signif-

icant. Our worst result reduced the flow completion time (FCT) by 52%, considering a 75/25 workload (75%mice flows, and 75% elephant flows proportion, with a 15 seconds interval between connections). As our best result, the EFI reduced FCT by 91%, considering a 50/50 workload (50% mice flows, and a 50% elephant flows proportion, with a 10 seconds interval between connections).

We believe the results showed in this research prove the importance of only considering elephant flows on reinforcement learning architectures for load balancing network traffic.

6.2 Limitations and Future work

The proposed solution aims at simplicity in terms of implementation. However, some conditions are limiting for the proper functioning of real networks. This is the case for three factors on our implementation, namely: (1) model based on a static topology, (2) mice flows can still be using an action chosen by the agent, and (3) the limited number of active flows considered on the topology. In this section, we discuss some suggestions for addressing each of these limitations.

Regarding the prototype based on a static topology, a possible alternative would be still use the model proposed in this work (state represented by the links occupation) and use the controller to get information about the topology. Our prototype already uses the Floodlight topology discovery module to discover the network topology and its possible paths. The challenge would be using a dynamic OpenAI gym environment to represent the links and possible actions.

Concerning still being able to reroute mice flows, we believe this would require a change in the prototype. To apply a specific action for a specific flow, the flow that needs to be routed should be part of the environment state. In other words, if the agent only looks at the state for choosing the best action (considering the step function of OpenAI environments), this means the state should be the flow that needs to be rerouted. Hence, the state would represent the active elephant flows on the network at a given time. If there are no active elephant flows in the network, the agent would not install a new rule; it would simply continue with the currently installed rules. This indicates that we were first thinking about our prototype without considering the agent model; we could get the same results with this modification. We believe a most suitable approach would be choosing an action for a given flow - if this is an elephant flow, choose a reroute action; if not,

continue.

Since the limitation of only looking at a pre-defined number of active flows is related to our prototype, we believe that if we consider a machine learning tool that allows us to consider a dynamic state, this would no longer be a problem. With this feature, every elephant flow could be a candidate for being rerouted by the agent.

Additional research opportunities involve changing the model state to (i) the number of flows passing through a switch and (ii) only consider the hosts involved as the state. For the first, a possible state of being studied would be considering how many flows are active on each switch - so we could maximize the number of flows passing through each switch instead of maximizing the number of links used. The state would be represented as the set of active elephant flows, its correspondent routes, and statistics for the second. We would change the model to only look at elephant flows and do not consider mice flows.

Finally, our experimental results demonstrate that our hypothesis of only considering elephant flows on a reinforcement learning approach for load balancing network traffic is valid (using EFI). Hence, to answer the question we raised when we started our work - *how to optimize the network performance by dynamically analyzing, predicting, and controlling the behavior of data transmitted over this network?* - we highlight the use of an elephant flow intelligence approach. It is crucial to identify which flows can harm the balance of the network and use this knowledge to better use the network resource.

We believe a good research opportunity would be using the model proposed in our work to network environments where the traffic is predictable, such as data centers. We believe our approach could significantly improve network resource usage and reinforcement learning agent performance in this scenario.

REFERENCES

AKAIKE, H. Fitting autoregressive models for prediction. **Annals of the Institute of Statistical Mathematics**, Springer Science and Business Media LLC, v. 21, n. 1, p. 243–247, dec. 1969. Available from Internet: <https://doi.org/10.1007/bf02532251>.

AL-KHATIB, O.; HARDJAWANA, W.; VUCETIC, B. Traffic modeling for machine-to-machine (M2M) last mile wireless access networks. In: **2014 IEEE Global Communications Conference**. [S.l.: s.n.], 2014. p. 1199–1204.

ALIZADEH, M. et al. Conga: Distributed congestion-aware load balancing for datacenters. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 4, p. 503–514, aug. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2740070.2626316>.

ARROYO-VALLES, R. et al. Q-probabilistic routing in wireless sensor networks. In: **2007 3rd International Conference on Intelligent Sensors, Sensor Networks and Information**. [S.l.: s.n.], 2007. p. 1–6.

BARROS, P. H. et al. Load balancing in d2d networks using reinforcement learning. In: **2019 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2019. p. 1–6.

BHORKAR, A. A. et al. Adaptive opportunistic routing for wireless ad hoc networks. **IEEE/ACM Transactions on Networking**, v. 20, n. 1, p. 243–256, Feb 2012. ISSN 1063-6692.

BOUTABA, R. et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. **Journal of Internet Services and Applications**, v. 9, n. 1, p. 16, Jun 2018. ISSN 1869-0238. Available from Internet: <https://doi.org/10.1186/s13174-018-0087-2>.

CAO, J. et al. Per-packet load-balanced, low-latency routing for clos-based data center networks. In: **Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2013. (CoNEXT ’13), p. 49–60. ISBN 9781450321013. Available from Internet: <https://doi.org/10.1145/2535372.2535375>.

CAVUSOGLU, B.; ORAL, E. A. Estimation of available bandwidth share by tracking unknown cross-traffic with adaptive extended kalman filter. **Computer Communications**, v. 47, p. 34–50, 2014.

CHEN, Y.-T.; LI, C.-Y.; WANG, K. A fast converging mechanism for load balancing among sdn multiple controllers. In: **2018 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2018. p. 00682–00687.

CHEN, Z.; HU, J.; MIN, G. Learning-based resource allocation in cloud data center using advantage actor-critic. In: **ICC 2019 - 2019 IEEE International Conference on Communications (ICC)**. [S.l.: s.n.], 2019. p. 1–6.

CHEN, Z.; WEN, J.; GENG, Y. Predicting future traffic using hidden markov models. In: **2016 IEEE 24th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2016. p. 1–6.

CORTEZ, P. et al. Internet traffic forecasting using neural networks. In: **The 2006 IEEE International Joint Conference on Neural Network Proceedings**. [S.l.: s.n.], 2006. p. 2635–2642. ISSN 2161-4393.

DOBSON, S. et al. Self-organization and resilience for networked systems: Design principles and open research issues. **Proceedings of the IEEE**, v. 107, n. 4, p. 819–834, 2019.

FRANÇOIS-LAVET, V. et al. An introduction to deep reinforcement learning. **Foundations and Trends® in Machine Learning**, Now Publishers, v. 11, n. 3-4, p. 219–354, 2018. Available from Internet: <https://doi.org/10.1561%2F2200000071>.

GAZIS, V. A survey of standards for machine-to-machine and the internet of things. **IEEE Communications Surveys Tutorials**, v. 19, n. 1, p. 482–511, 2017.

GUERRERO, C. D.; LABRADOR, M. A. Traceband: A fast, low overhead and accurate tool for available bandwidth estimation and monitoring. **Comput. Netw.**, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 6, p. 977–990, abr. 2010. ISSN 1389-1286. Available from Internet: <http://dx.doi.org/10.1016/j.comnet.2009.09.024>.

GUO, J. et al. Energy-efficient resource allocation for multi-user mobile edge computing. In: **GLOBECOM 2017 - 2017 IEEE Global Communications Conference**. [S.l.: s.n.], 2017. p. 1–7.

GUO, L.; MATTA, I. The war between mice and elephants. In: **Proceedings Ninth International Conference on Network Protocols. ICNP 2001**. [S.l.: s.n.], 2001. p. 180–188.

HAMDAN, M. et al. Flow-aware elephant flow detection for software-defined networks. **IEEE Access**, v. 8, p. 72585–72597, 2020.

HAMDAN, M. et al. Flow-aware elephant flow detection for software-defined networks. **IEEE Access**, v. 8, p. 72585–72597, 2020.

JOSHI, M.; HADI, T. H. A review of network traffic analysis and prediction techniques. **ArXiv**, abs/1507.05722, 2015.

KNOB, L. A. D. et al. Sdefix — identifying elephant flows in sdn-based ixp networks. In: **NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2016. p. 19–26.

KNOB, L. A. D. et al. Mitigating elephant flows in sdn-based ixp networks. In: **2017 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2017. p. 1352–1359.

KOSEOGLU, M. Pricing-based load control of m2m traffic for the lte-a random access channel. **IEEE Transactions on Communications**, v. 65, n. 3, p. 1353–1365, 2017.

LI, Y. et al. Inter-data-center network traffic prediction with elephant flows. In: **NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2016. p. 206–213. ISSN 2374-9709.

LIN, Z.; SCHAAR, M. van der. Autonomic and distributed joint routing and power control for delay-sensitive applications in multi-hop wireless networks. **IEEE Transactions on Wireless Communications**, v. 10, n. 1, p. 102–113, January 2011. ISSN 1536-1276.

LIU, S.; FORREST, J. **Grey Information:Theory and Practical Applications Springer-Verlag, Londun Ltd,2006**. [S.l.: s.n.], 2006. ISBN -10:1-85233-995-0.

LIU, Z. et al. An adaptive approach for elephant flow detection with the rapidly changing traffic in data center network: An approach for elephant flow detection with the changing traffic. **International Journal of Network Management**, p. e1987, 07 2017.

MAO, B. et al. A tensor based deep learning technique for intelligent packet routing. In: . [S.l.: s.n.], 2017. p. 1–6.

MEKINDA, L.; MUSCARIELLO, L. Supervised machine learning-based routing for named data networking. In: **2016 IEEE Global Communications Conference (GLOBECOM)**. [S.l.: s.n.], 2016. p. 1–6. ISSN null.

MNIH, V. et al. **Playing Atari with Deep Reinforcement Learning**. 2013.

NAREJO, S.; PASERO, E. An application of internet traffic prediction with deep neural network. In: ____. **Multidisciplinary Approaches to Neural Computing**. Cham: Springer International Publishing, 2018. p. 139–149. ISBN 978-3-319-56904-8. Available from Internet: <https://doi.org/10.1007/978-3-319-56904-8{_}.>

PAKZAD, F.; PORTMANN, M.; HAYWARD, J. Link capacity estimation in wireless software defined networks. **2015 International Telecommunication Networks and Applications Conference (ITNAC)**, p. 208–213, 2015.

PAUL, A. K.; TACHIBANA, A.; HASEGAWA, T. Implementation design of available bandwidth measurement scheme: A proxy based approach. In: **Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services**. New York, NY, USA: ACM, 2016. (MOBIQUITOUS 2016), p. 257–262. ISBN 978-1-4503-4759-4. Available from Internet: <http://doi.acm.org/10.1145/3004010.3004047>.

PIZZUTTI, M.; SCHAEFFER-FILHO, A. E. An efficient multipath mechanism based on the flowlet abstraction and p4. In: **2018 IEEE Global Communications Conference (GLOBECOM)**. [S.l.: s.n.], 2018. p. 1–6.

PIZZUTTI, M.; SCHAEFFER-FILHO, A. E. Adaptive multipath routing based on hybrid data and control plane operation. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2019. p. 730–738.

POUPART, P. et al. Online flow size prediction for improved network routing. In: **2016 IEEE 24th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2016. p. 1–6.

SCHULMAN, J. et al. Proximal policy optimization algorithms. **CoRR**, abs/1707.06347, 2017. Available from Internet: <http://arxiv.org/abs/1707.06347>.

SILVA, M. V. B. da et al. Ideafix: Identifying elephant flows in p4-based ixp networks. In: **2018 IEEE Global Communications Conference (GLOBECOM)**. [S.l.: s.n.], 2018. p. 1–6.

SUTTON, R.; BARTO, A. **Reinforcement Learning: An Introduction**. MIT Press, 2018. (Adaptive Computation and Machine Learning series). ISBN 9780262039246. Available from Internet: <https://books.google.com.br/books?id=6DKPtQEACAAJ>.

SUTTON, R. S. **Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning series)**. A Bradford Book, 2018. ISBN 0262039249. Available from Internet: <https://www.xarg.org/ref/a/0262039249/>.

ĐUKIĆ, V. et al. Is advance knowledge of flow sizes a plausible assumption? In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. Boston, MA: USENIX Association, 2019. p. 565–580. ISBN 978-1-931971-49-2. Available from Internet: <https://www.usenix.org/conference/nsdi19/presentation/dukic>.

VALADARSKY, A. et al. Learning to route. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2017. (HotNets-XVI), p. 185–191. ISBN 9781450355698. Available from Internet: <https://doi.org/10.1145/3152434.3152441>.

VANINI, E. et al. Let it flow: Resilient asymmetric load balancing with flowlet switching. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 407–420. ISBN 978-1-931971-37-9. Available from Internet: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini>.

WANG, Y.; LIU, Y.; GAN, Y. Research on combination network traffic forecasting model. In: **2018 IEEE International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)**. [S.l.: s.n.], 2018. p. 311–314.

WU, Y. et al. Modeling and forecasting of timescale network traffic dynamics in m2m communications. In: **2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.: s.n.], 2019. p. 711–721.

XU, Z. et al. Experience-driven networking: A deep reinforcement learning based approach. In: **IEEE INFOCOM 2018 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2018. p. 1871–1879.

YU, E. S.; CHEN, C. Y. R. Traffic prediction using neural networks. In: **Proceedings of GLOBECOM '93. IEEE Global Telecommunications Conference**. [S.l.: s.n.], 1993. p. 991–995 vol.2.

YU, W. et al. Historical best q-networks for deep reinforcement learning. In: **2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.: s.n.], 2018. p. 6–11.

ZHANG, L. et al. Named data networking. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 66–73, jul. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656887>.

ZHOU, J. et al. Wcmp: Weighted cost multipathing for improved fairness in data centers. In: **Proceedings of the Ninth European Conference on Computer Systems**. New York, NY, USA: Association for Computing Machinery, 2014. (EuroSys '14). ISBN 9781450327046. Available from Internet: <https://doi.org/10.1145/2592798.2592803>.

APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS

This appendix presents a extended summary of the work in the Portuguese language. The structure of this appendix is the same of the English document.

Neste apêndice é apresentado um resumo expandido do trabalho na língua portuguesa. A estrutura deste apêndice é a mesma do documento na língua inglesa.

A.1 Introdução

A complexidade, heterogeneidade e escala das redes de computadores cresceram além dos limites da administração manual, a tal ponto que a principal causa de falhas em ambientes de rede é o erro humano (DOBSON et al., 2019). Além disso, o impacto de falhas em ambientes de rede pode ser caro, agravado pelo tempo de reação atrasado e baixa precisão dos métodos tradicionais de tolerância a falhas. Isso desencadeou uma mudança na filosofia de design dos sistemas de gerenciamento de rede para minimizar o papel dos humanos no circuito de controle. Aliados, esses pontos motivam os esforços para usar o aprendizado de máquina para prever o comportamento e equilibrar o tráfego da rede.

Por um lado, as abordagens de previsão de tráfego desempenham um papel crucial nas operações e gerenciamento de rede e tentam antecipar a carga, o volume, o tamanho do pacote, o roteamento e muito mais do tráfego. Isso pode ajudar os provedores a otimizar os recursos de rede (NAREJO; PASERO, 2018). Por outro lado, o balanceamento de carga tenta inferir a classificação e divisão dos fluxos da rede para obter o melhor aproveitamento dos links de transmissão. Isso normalmente é usado para atingir taxas de transferência mais altas e atrasos de transmissão menores, bem como para reduzir efeitos adversos, como retransmissões (PIZZUTTI; SCHAEFFER-FILHO, 2019).

Ao conceber uma estratégia de balanceamento de carga, deve-se estar ciente dos diferentes tipos de fluxos de rede. Por exemplo, os fluxos elefantes representam um grande (em número de bytes) e um fluxo contínuo de tráfego, enquanto os fluxos ratos tendem a ser pequenos e de curta duração (HAMDAN et al., 2020a). Considerando que os fluxos elefantes tendem a ocupar um caminho de rede por muito mais tempo do que os fluxos ratos, existe o risco de que o número de fluxos ativos em alguns links fique desequilibrado. Por exemplo, heurísticas simples que ignoram o tamanho do fluxo e distribuem fluxos uniformemente em todos os caminhos de comprimento igual geralmente

levam ao congestionamento, e heurísticas de balanceamento de carga devem ser usadas para detectar e corrigir desequilíbrios (POUPART et al., 2016). Além disso, detectar e prevenir abusos na rede está se tornando um desafio com o crescente volume de tráfego e a complexidade das redes.

Neste trabalho, exploramos a previsão de tráfego de rede para melhorar as técnicas de balanceamento de carga, quando usadas em conjunto. Existem várias abordagens de previsão de tráfego de rede. Podemos dividir essas abordagens em duas categorias principais: métodos de previsão de série temporal (*Time Series Forecast* - TSF) e métodos de previsão não-série temporal (*non-TSF*).

Em particular, propomos uma abordagem em duas etapas com base no aprendizado por reforço e previsão de tráfego de rede para equilibrar os fluxos de rede e garantir o uso eficiente dos seus recursos. Em primeiro lugar, a previsão do tráfego de rede é usada para determinar quais fluxos têm o maior impacto nos recursos da rede e podem causar o seu desequilíbrio. Em segundo lugar, a aprendizagem por reforço é usada para restabelecer esse equilíbrio, com foco em fazer o melhor uso dos recursos, dado o estado atual da rede. Nossas principais contribuições são (i) modelagem do problema em função de estados e ações em um sistema que visa balancear o tráfego da rede e (ii) uma arquitetura que usa de forma mais criteriosa a aprendizagem por reforço nos fluxos de interesse para o balanceamento de carga. Defendemos que essa abordagem de aprendizado de máquina em duas etapas, que chamamos de aprendizado por reforço antecipado (*Look-Ahead Reinforcement Learning*), pode reduzir os erros humanos evitando interações desnecessárias.

A.2 Background

No contexto deste trabalho, destacamos três conceitos principais: previsão de tráfego, balanceamento de carga e aprendizado por reforço. Utilizamos ideias de previsão de tráfego para identificar antecipadamente fluxos que podem representar um risco para o equilíbrio da rede (dado que o objetivo da arquitetura proposta é balanceamento de carga). O conceito de aprendizado por reforço, por sua vez, é utilizado na modelagem do sistema de aprendizado de máquina.

A.2.1 Predição de tráfego

As técnicas de predição de tráfego de rede podem ser divididas em duas categorias amplas: predição como um problema de previsão de série temporal (em inglês, *Time Series Forecast*, ou TSF) e predição como um problema não linear (em inglês, *non-TSF*). O objetivo no TSF é construir um modelo de regressão capaz de traçar uma correlação precisa entre o volume de tráfego futuro e os volumes de tráfego previamente observados (BOUTABA et al., 2018). Em contraste com o TSF, podemos prever o tráfego de rede usando outros métodos, chamados de abordagens não lineares. As equações dinâmicas não lineares geram séries temporais não lineares. Eles exibem características que não podem ser modeladas por processos lineares, como variação de mudança de tempo, ciclo assimétrico, limites e quebras. Este modelo é geralmente usado para prever o tráfego de rede com técnicas como redes neurais e lógica *fuzzy*.

A.2.2 Balanceamento de carga

O balanceamento de carga se refere à distribuição eficiente do tráfego na rede. Esta técnica infere a classificação e divisão do fluxo de tráfego para obter o melhor aproveitamento dos recursos da rede e pode minimizar efeitos adversos como retransmissões de pacotes (PIZZUTTI; SCHAEFFER-FILHO, 2018). Esta distribuição de carga de trabalho é essencial para obter infraestruturas altamente disponíveis e otimizar o desempenho da rede (ALIZADEH et al., 2014).

A.2.3 Aprendizado por reforço

A aprendizagem por reforço (em inglês, *Reinforcement Learning*, ou RL) depende de um agente cujo comportamento é definido com base em estados, um conjunto de ações e um conjunto de recompensas correspondentes (SUTTON, 2018). Nesse contexto, um estado é uma configuração que relaciona o agente a outros elementos do ambiente. Uma ação é um movimento possível que um agente pode fazer para atingir seu objetivo. Por fim, uma recompensa é um *feedback* pelo qual medimos o sucesso ou o fracasso das ações de um agente em um determinado estado. Dessa forma, um agente é responsável por considerar o estado atual, as ações possíveis e as recompensas correspondentes para

definir o melhor movimento para a solução do problema.

A.3 Trabalhos relacionados

Nesta seção, destacamos os principais trabalhos da literatura relacionados ao trabalho proposto neste documento. Assim, discutimos sobre trabalhos na área de previsão de tráfego e trabalhos na área de balanceamento de carga.

A.3.1 Previsão de tráfego

O trabalhado apresentado por Chen et al. (CHEN; WEN; GENG, 2016) modela a relação entre o volume de tráfego e estatísticas de fluxo usando um modelo de Markov. Assim, é possível evitar a metrificação direta do volume de tráfego através de estimativas. Os resultados experimentais demonstram a viabilidade e eficácia do método proposto.

Poupart et al. (POUPART et al., 2016) descreve como formular um problema de previsão de tráfego como uma tarefa de aprendizado de máquina online, para que o modelo seja adaptado continuamente às mudanças no fluxo de tráfego. Com isso, os autores: (i) avaliam a natureza preditiva de um conjunto de recursos e a precisão de três técnicas de predição online e (ii) demonstram como usar esses preditores online para melhorar o roteamento. Os autores consideram sete estatísticas que estão disponíveis para cada fluxo: IP de origem, IP de destino, porta de origem, porta de destino, protocolo, indicação de servidor ou cliente e os tamanhos dos três primeiros pacotes de dados. Os resultados experimentais mostram que o método de roteamento resultante reduziu o tempo médio de conclusão dos fluxos elefantes, enquanto manteve o tempo médio de conclusão de fluxos ratos.

A.3.2 Balanceamento de carga

O trabalho feito por Vanini et al. (VANINI et al., 2017) introduz o conceito de *flowlet*: uma rajada de pacotes separados de outras rajadas por um intervalo no tempo. Os autores mostram que o uso de *flowlets* é uma técnica poderosa para balanceamento de carga com assimetria (falhas de link e heterogeneidade em equipamentos de rede). O método é uma abordagem mais geral para o balanceamento de carga: redirecionar os

fluxos aleatoriamente, permitindo que sua elasticidade equilibre naturalmente o tráfego. Essa técnica é uma melhoria significativa em relação ao ECMP.

Pizzutti et al. (PIZZUTTI; SCHAEFFER-FILHO, 2018) apresenta uma abordagem que se baseia na abstração do *flowlet* para gerenciar a divisão e o monitoramento dos fluxos da rede. Os autores adicionam uma função que monitora e atua dinamicamente no plano de dados para encontrar o intervalo mínimo possível que permite a alternância de um fluxo entre as rotas sem causar efeitos adversos. Eles avaliam seu trabalho usando P4 e destacam que estratégias que identificam classes de fluxo podem ser aplicadas para liberar caminhos específicos que sofram com gargalos.

Barros et al. (BARROS et al., 2019) propõe um mecanismo de gerenciamento, orquestração e controle de fluxo no contexto de dispositivo a dispositivo (D2D) para lidar com o balanceamento de carga, usando a técnica de *Deep Q-Learning* (DQN). O trabalho usa aprendizagem por reforço para minimizar a chance de mover a carga para um nó que provavelmente ficará sobrecarregado em um futuro próximo. Os autores usam um processo gaussiano para prever a carga de um nó em uma rede D2D e comparar o método proposto com uma abordagem de base. Os resultados mostram que a abordagem proposta promove um equilíbrio adequado entre desempenho de balanceamento de carga e velocidade.

Chen et al. (CHEN; LI; WANG, 2018) considera o problema do tempo de convergência do balanceamento de carga. Para isso, os autores propõem um mecanismo de balanceamento de carga de convergência rápida. A abordagem é baseada em algoritmos genéticos e, de acordo com os resultados experimentais, a simulação convergiu 20,7 % mais rápido que os outros mecanismos, conseguindo assim um melhor desempenho de equilíbrio de carga.

Diferimos desses trabalhos em dois fatores: (i) diferentemente das técnicas apresentadas, usamos uma etapa adicional para identificar fluxos de alto impacto (ou seja fluxos elefantes) e determinar se a possível sobrecarga desses fluxos compensará os custos de uma abordagem com aprendizado por reforço; (ii) usamos a técnica de aprendizado por reforço, que captura o estado instantâneo da rede, para balancear a carga de trabalho.

A.4 Look-Ahead Reinforcement Learning for Load Balancing: abordagem de aprendizado por reforço para balanceamento de carga de tráfego de rede

Apresentamos agora o mecanismo proposto para balanceamento de carga de fluxos de rede usando aprendizado por reforço. Esse mecanismo tenta explorar caminhos alternativos na topologia da rede para garantir a baixa utilização geral da largura de banda.

A.4.1 Visão geral

Em vez de balancear qualquer fluxo de tráfego, nossa abordagem depende de um agente de aprendizado por reforço que se concentrará apenas nos fluxos elefantes (HAMDAN et al., 2020b), o que tende a impactar a utilização da largura de banda de forma mais significativa. Defendemos que adicionar esta etapa extra pode melhorar as decisões de roteamento. A motivação principal é evitar o desperdício de recursos no redirecionamento de fluxos que não representem um risco para o equilíbrio do tráfego da rede. Em segundo lugar, nosso objetivo é melhorar os resultados do balanceamento de carga atuando apenas nos fluxos que podem impactar significativamente a utilização da largura de banda.

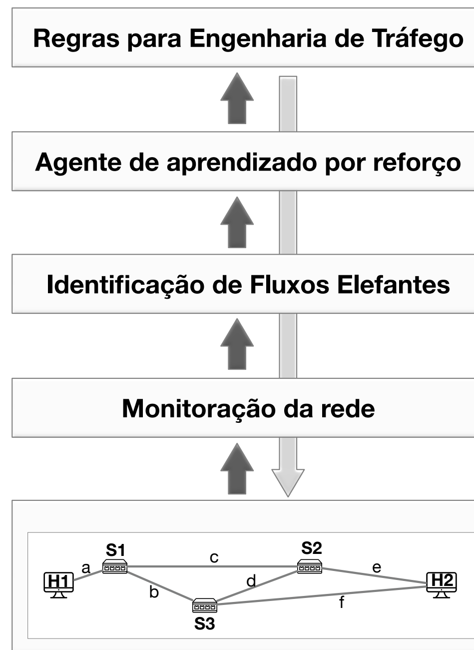
Em nossa abordagem, coletamos informações de utilização da rede para identificar se um agente de aprendizado por reforço deve intervir nas decisões de roteamento padrão do controlador. Em casos positivos, as ações do agente são traduzidas em ações de redirecionamento de tráfego (regras *OpenFlow* para determinados fluxos). Nossa abordagem é especialmente adequada para grandes volumes de tráfego porque pode suprimir os custos de latência ao considerar os fluxos elefantes. A visão geral dessa arquitetura está ilustrada na Imagem A.1.

A.4.2 Agente de aprendizado por reforço

Modelamos o problema de balanceamento de carga de tráfego de rede como um problema de aprendizagem por reforço que visa manter uma ocupação homogênea dos enlaces da topologia. Os principais componentes desse modelo são os seguintes: estado, função de recompensa e conjunto de ações.

Representamos o estado pelo número de bits transferidos por cada porta de *switch* na topologia considerada, que é calculado com base nas estatísticas coletadas da rede. O

Figura A.1: Visão geral da arquiteura de Look-Ahead Reinforcement Learning para balanceamento de carga.



Fonte: a Autora.

estado resultante é um vetor com n valores, correspondendo ao número de portas de *switch* na topologia da rede. Cada posição do vetor corresponde ao número de bits transferidos por segundo pela porta em questão. O estado resultante é a principal ferramenta usada pelo agente para identificar se a rede pode ficar sobrecarregada. Com as informações disponíveis sobre a topologia da rede, o agente avaliará o quão custoso um estado pode ser para esta rede. Ou seja, quão homogêneo é o uso dos enlaces da topologia.

Uma função de recompensa deve modelar recompensas por meio de *feedback* contínuo e deixar o agente saber o quão perto está de seu objetivo. Portanto, queremos avaliar cada tupla estado-ação da rede em termos de uso de recursos. Mais especificamente, considerando que definimos um estado em nosso problema como o uso de portas de *switch* em um momento específico, precisamos avaliar como podemos distribuir melhor o tráfego. Intuitivamente, queremos valores de recompensa mais altos para estados que representam o uso homogêneo de portas de *switch*. Nosso agente oferece uma gama de funções de recompensa diferentes para avaliar como o tráfego é distribuído: média ponderada da utilização dos enlaces, média harmônica da utilização dos enlaces e desvio padrão da utilização dos enlaces.

Definimos o estado no tempo t, S_t, como um vetor n-dimensional $[x_1, \ldots, x_n]$, onde n corresponde ao número de portas de *switch* e onde $x_i, para 1 \leq i \leq n$, corresponde ao número de bits sendo transferidos por segundo pela porta do *switch* de índice i. Assim,

uma maneira de definir a recompensa R_t, no momento t, é em termos de uma aplicação e função dependente do contexto, f, como segue:

$$R_t(S_t) = f(S_t)\mu \tag{A.1}$$

onde f é definido por um designer dependendo do critério específico que deseja otimizar ao tentar manter um uso mais homogêneo dos links de topologia: buscar homogeneidade pelo tráfego de distribuição de acordo com (1) uso ponderado de portas de switch; (2) média harmônica do uso das portas do switch; ou (3) uso de portas de switch de desvio padrão. Finalmente, em A.1, μ é definido como a soma dos bits transferidos pela porta de saída do host de origem e a porta de entrada do host de destino. Intuitivamente, μ é uma penalidade de recompensa aditiva que reflete se existem possíveis loops na rede.

De acordo com a estratégia de recompensa, definimos equações específicas para calcular a que distância um agente está de seu objetivo final (estas serão comparadas e avaliadas na seção de prototipação e análise experimental).

A Equação A.2 corresponde à média ponderada de utilização das portas dos *switches*, onde consideramos a função de recompensa final como a soma de todas as portas do *switch*.

$$f(S_t) = \sum_{i=1}^{n} p(x_i) \tag{A.2}$$

onde $p(x_i) = 2x_i$ se $x > 1$, e 1 se $x <= 1$.

A Equação A.3 corresponde à média harmônica de utilização das portas do *switch*:

$$f(S_t) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}} \tag{A.3}$$

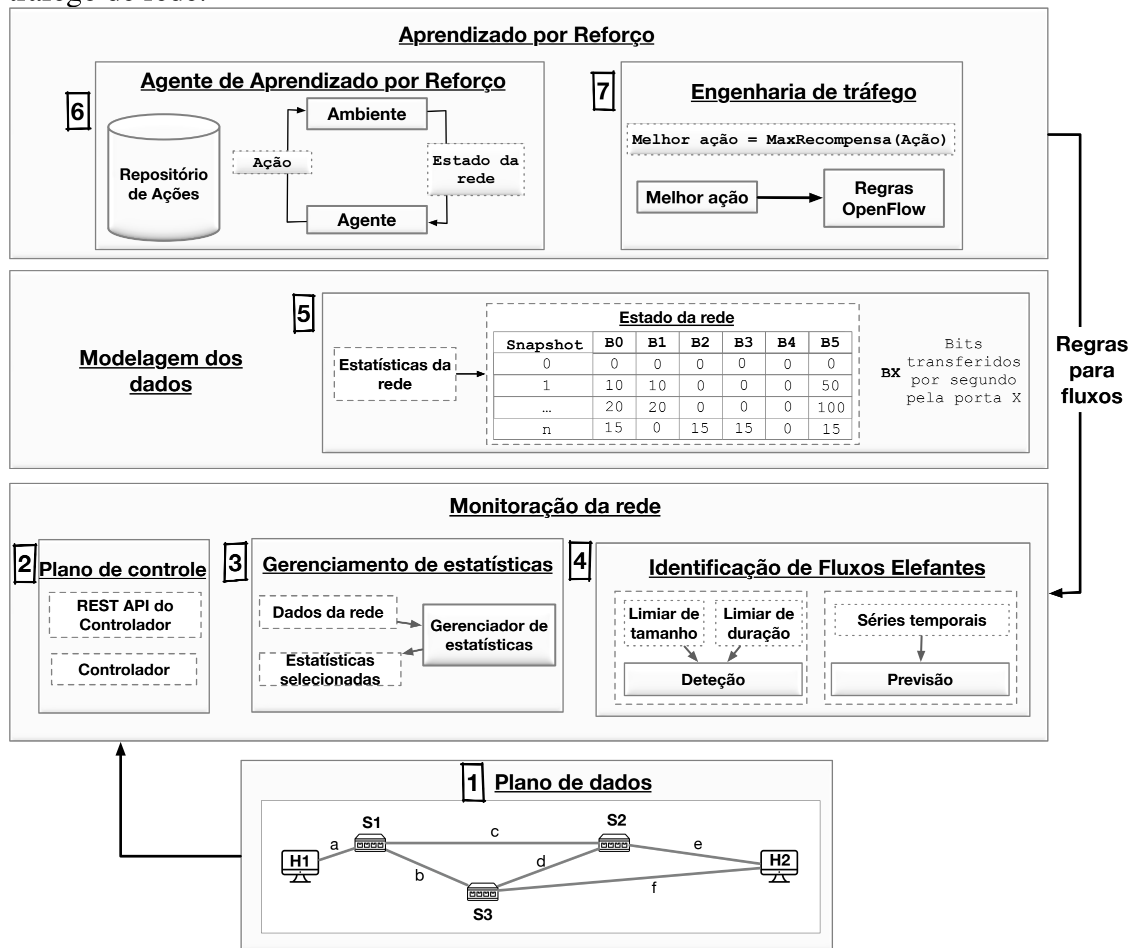
A Equação A.4 corresponde ao desvio padrão de utilização das portas do *switch*.

$$f(x) = \sigma_{x_n} \tag{A.4}$$

Por fim, também definimos uma ação em nosso modelo como a tupla $< switch_id, in_port, out_port >$. O agente resultante é responsável por interagir com o ambiente e escolher a melhor ação para cumprir seu objetivo. O agente usará um algoritmo de aprendizado por reforço para analisar o estado da rede e determinar o melhor movimento. Isto é, a ação que oferece a maior recompensa: aquela que manterá um uso mais homogêneo dos enlaces da topologia.

Como o número de bits em cada porta é ilimitado, existe um número possivelmente infinito (mas contável) de estados possíveis no sistema. Por esta razão, a aplicação de

Figura A.2: Arquitetura Look-Ahead Reinforcement Learning para balanceamento de tráfego de rede.



Fonte: a Autora.

versões tabulares de algoritmos de aprendizagem por reforço padrão, como *Q-Learning* tabular (SUTTON, 2018), é inviável, uma vez que não podemos armazenar uma tabela definida com um número infinito de estados. Portanto, contamos com um agente de *Q-Learning* profundo (*Deep Q-Learning*) (YU et al., 2018), que usa uma rede neural para aproximar o valor Q para o conjunto de ações possíveis com base no estado atual. O estado atual é o valor de entrada para esta rede neural. A saída é o Q-Valor para o conjunto de ações possíveis para que o agente escolha a mais adequada. Para a implementação do agente de aprendizagem por reforço, desenvolvemos um ambiente de OpenAI [1]. O algoritmo *Deep Q-Learning* de Stable Baselines [2] com 2 camadas ocultas mostra ótimos resultados para aplicações semelhantes na literatura (MNIH et al., 2013), e por isso optamos pela implementação deste algoritmo neste trabalho.

[1]https://gym.openai.com

[2]https://stable-baselines.readthedocs.io/en/master/modules/dqn.html

A.5 Prototipação e análise experimental

Nesta seção, descrevemos o protótipo implementado e apresentamos uma avaliação inicial. Em particular, avaliamos a capacidade da abordagem de aprendizado de reforço para balancear a carga do tráfego de rede, analisamos a escalabilidade do protótipo e avaliamos se a identificação de fluxos elefantes produz um melhor uso da rede para balancear a carga de trabalho.

A.5.1 Prototipação

Considerando a proposta do nosso trabalho, destacamos os seguintes como principais objetivos do protótipo: (i) validar o modelo proposto para aprendizagem por reforço, (ii) avaliar se o modelo é capaz de balancear a carga de tráfego na rede e qual função de recompensa apresenta melhores resultados, (iii) avaliar se a etapa adicional de identificar fluxos elefantes (look-ahead) poderia gerar um agente mais inteligente e, portanto, melhorar o uso de recursos de rede para balancear a carga de trabalho, e (iv) analisar a escalabilidade dos agentes.

Para a prototipagem, usamos o Mininet VM para emular a topologia SDN, o controlador Floodlight na versão 1.2 e um aplicativo docker com Python 3.7 para executar a aplicação de coleta de estatísticas, identificação de fluxo elefante e agente de aprendizado por reforço. Os principais componentes do protótipio são os seguintes: Statistics Manager (gerenciador de estatísticas), Elephant Flow Identification (identificação de fluxos elefantes), Reinforcement Learning Agent (agente de aprendizado por reforço), e Flow Action Translator (tradutor de ações para fluxos).

Executamos todos os experimentos em um MacBook Pro Mid 2014, processador 2,6 GHz Dual-Core Intel Core i5, com 8 GB de memória DDR3 de 1600 MHz. Para reproduzir os experimentos apresentados neste capítulo, seriam necessários os componentes listados abaixo. Todos os projetos mencionados são de código aberto, e mais detalhes podem ser encontrados no arquivo `README.md` do repositório no GitHub.

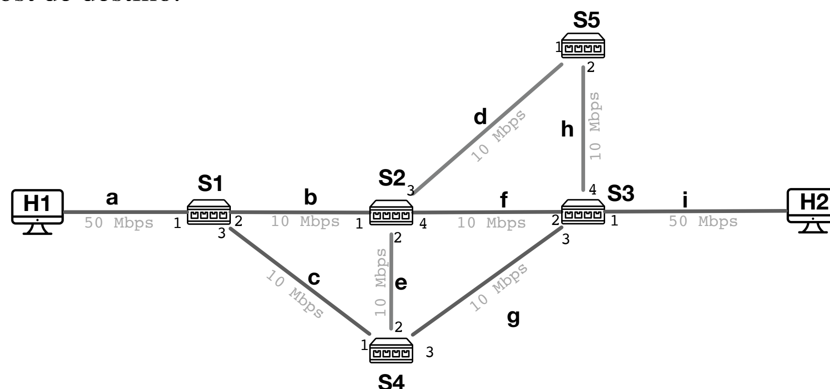
Tabela A.1: Conjunto de diferentes parâmetros e seus valores correspondentes.

Parâmetro	Valor
`max_sim_flows`	16 flow matches
`install_wait_time`	7 segundos
`ef_threshold_duration`	10 segundos
`ef_threshold_size`	100 MBytes
`idle_timeout`	60 segundos
`hard_timeout`	10 segundos
`EFI`	100

A.5.2 Análise experimental

Para a análise experimental, utilizamos 2 topologias diferentes. A Figura A.3 apresenta a topologia de base usada no protótipo. Essa foi a configuração escolhida porque possui um número menor de rotas possíveis, gerando menores conjuntos de estado e ação. Esta topologia foi criada para facilitar a listagem de todos os caminhos possíveis entre os dois hosts e considerar a possibilidade de inserir regras que possam gerar um loop. Observe que 50 Mbps é a largura de banda mais alta, localizada nas extremidades da topologia, para induzir um efeito de gargalo. Também consideramos uma segunda topologia em nosso protótipo, para avaliar o impacto da adição de switches extras na topologia. Chamamos essa segunda topologia de S2, ilustrada na Figura A.4.

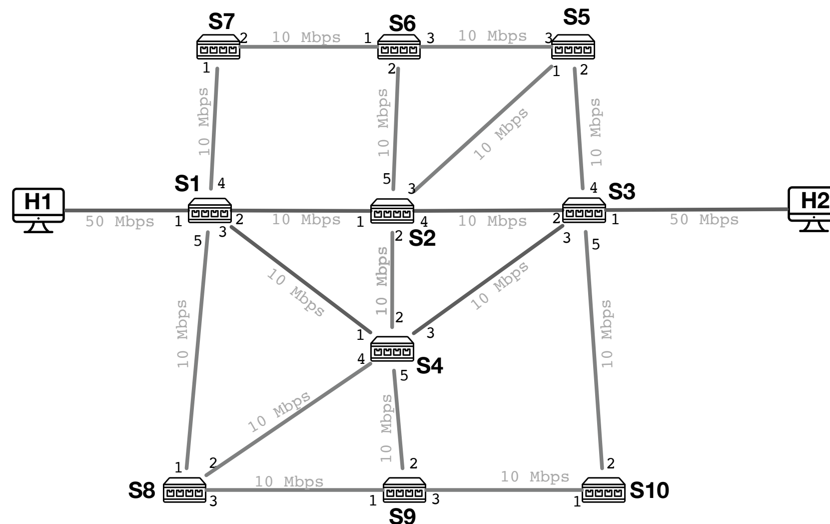
Figura A.3: Topologia S1 usada como base para os experimentos. H1 é o host de origem e H2 é o host de destino.



Fonte: a Autora.

A Tabela A.5.2 apresenta os parâmetros e os valores de cada um deles utilizados nos experimentos. O número de fluxos simultâneos que o agente conhece (`max_sim_flows`) foi definido como 16 devido à restrição de tempo. Considerando a largura de banda do enlace nas topologias de rede, precisávamos ter uma quantidade significativa de fluxos ratos e elefantes que terminariam em um período de tempo razoável.

Figura A.4: Topologia S2 usada para analisar o impacto de switches adicionais no resultado dos experimentos. H1 é o host de origem e H2 é o host de destino.



Fonte: a Autora.

Nosso tempo foi limitado porque entendemos a necessidade de reproduzir cada conjunto de experimentos várias vezes para observar a variação dos resultados. Além disso, como o Floodlight só fornece estatísticas para fluxos instalados anteriormente, nosso protótipo teve que instalar inicialmente o conjunto de fluxos que seriam usados em nossos experimentos. Mais especificamente, instalamos uma regra inicial para cada fluxo que o controlador analisaria, essa regra enviava os pacotes dos fluxos pela rota padrão do controlador. Isso significa que para um fluxo roteado por um caminho de n switches, teríamos que instalar n regras. O número total de regras iniciais a serem instaladas na inicialização do agente seria $n*m$, onde m seria o número de fluxos. Como as topologias que usamos para esses experimentos tinham 5 e 10 *switches* ($n = 5$ e $n = 10$), para minimizar o tempo de configuração inicial e ainda avaliar o modelo com fluxos elefantes, consideramos 16 fluxos simultâneos um valor adequado.

Conforme explicado anteriormente, o Floodlight executa uma *thread* para coletar estatísticas periodicamente. Definimos esse tempo para ser a cada 5 segundos porque queríamos obter os valores mais atualizados possíveis - para ter um estado preciso. Testamos este valor como 1 e 2 segundos, mas não obtivemos resultados satisfatórios: os contadores estavam imprecisos. Ao definir esse tempo para 5 segundos, conseguimos obter resultados precisos para nossos experimentos. Como precisávamos de algum tempo entre a instalação de uma regra, o roteamento de pacotes por meio desta regra e ainda observar o impacto nas estatísticas da rede, adicionamos 2 segundos extras, resultando em um `install_wait_time` total de 7 segundos.

Para o limite de duração do fluxo elefante (ef_threshold_duration), usamos o mesmo valor usado em SDEFIX e IDEAFIX (KNOB et al., 2016) e IDEAFIX (SILVA et al., 2018). Como consideramos uma duração de 10 segundos e as topologias usadas para os experimentos tinham a maioria (gargalos) de links com capacidade de 10 Mbps, consideramos (ef_threshold_size) como 100 MBytes. Consideramos esse valor apropriado como tamanho limite porque também precisávamos que os fluxos fossem (i) grandes o suficiente para observar o impacto das escolhas do agente e (ii) pequenos o suficiente considerando a restrição de tempo que tínhamos para executar os experimentos. Mais especificamente, em relação ao tempo necessário para completar os experimentos, tivemos dois fatores principais: install_wait_time e o número de passos de tempo necessários para o agente completar sua tarefa. Considerando install_wait_time como 7 segundos, no pior cenário, precisaríamos de $7 * num_timesteps$ para completar um experimento. No entanto, o número de passos de tempo necessários depende do número de fluxos e do tamanho desses fluxos - quanto mais altos os valores, maior o número de passos de tempo. Experimentos preliminares mostraram que precisávamos de uma média de 700 passos de tempo para completar todos os cinco fluxos (10 MBytes, 50 MBytes, 100 MBytes, 200 MBytes e 500 MBytes) para uma configuração de experimento usando um agente de aprendizagem por reforço. Isso implica que precisamos de $7 * 700 = 4900$ segundos para concluir um experimento - para o pior caso. Considerando pelo menos cinco repetições para este experimento (para analisar os valores de média e desvio padrão), precisávamos de $4.900 * 5 = 24.500$ segundos para completar um experimento com suas replicações. Além disso, considerando que executamos esses experimentos para 22 cenários (para avaliação funcional: 3 agentes para avaliação funcional + 1 linha de base e para avaliação EFI: 2 agentes $*3$ cargas de trabalho $*3$ intervalos $=$ 18 cenários), nosso pior cenário envolveu a necessidade de $24.500 * 18 = 441.000$ segundos $= 122,5$ horas. Isso assumindo que nenhum dos cenários do experimento teria que ser executado novamente devido a alguma inconsistência ou erro. Resumindo, considerando todos esses fatores, um fluxo é considerado um fluxo elefante baseado em uma ef_threshold_size de 100 MBytes.

O parâmetro idle_timeout é usado para instalar uma regra em um *switch*. Este valor indica quanto tempo uma regra deve permanecer ativa depois de não estar mais sendo usada. O valor padrão para este parâmetro é 0 [3], mas definimos esse valor para 60 segundos para todos regras instaladas. Usamos esse valor porque queremos ter

[3] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343518/Static+Entry+Pusher+API

certeza de que o agente está aprendendo as melhores regras para nosso objetivo. Como não é viável incrementar a prioridade de cada nova regra, usamos o tempo limite como alternativa. Suponha que o agente entenda essa regra como uma ação que colaborará para o uso de recursos mais homogêneo. Nesse caso, o agente continuará escolhendo esta ação e renovando seu tempo ativo. Caso contrário, a regra seria removida porque não está mais sendo usada.

O parâmetro `hard_timeout` também é usado para instalar uma regra em um *switch*. Porém, este valor indica por quanto tempo essa regra é válida, independente de ser usada ou não. O valor padrão para este parâmetro também é 0, mas nós o definimos para 10 segundos para uma regra: a regra do controlador. A regra do controlador só deve ser escolhida pelo agente em casos extremos: controle de *loop*. Isso significa que se o agente não conseguiu aprender um bom caminho para um determinado fluxo e, em vez disso, instalou regras que geram um *loop* na rede, a regra do controlador pode ser utilizada para se recuperar deste estado de *loop*, conforme explicado anteriormente. Como entendemos que essa regra só deve ser usada em casos extremos (devido à sobrecarga de envio de pacotes para o controlador), definimos um tempo limite rígido de 10 segundos, o que seria tempo suficiente para um fluxo se recuperar de um estado de *loop*. Também testamos essa regra com 5 e 20 segundos, mas com 10 segundos, obtivemos os melhores resultados.

Por fim, o parâmetro `EFI` é utilizado para penalizar o agente na hora de escolher ações para os fluxos ratos, conforme explicado anteriormente. O valor usado em nossos experimentos foi $EFI = 100$ por causa da gama de valores de recompensa possíveis.

A.5.2.1 Avaliação funcional

Este primeiro conjunto de experimentos visa avaliar se um agente pode equilibrar efetivamente a carga de trabalho, considerando as diferentes funções de recompensa discutidas anteriormente. Comparamos o desempenho dessas variações com uma abordagem base, que é o uso do controlador do Floodlight sem usar nenhum método de balanceamento de carga.

Para comparar essas alternativas, consideramos o tempo total de conclusão do fluxo (em inglês, *Flow Completion Time*, ou FCT) como a métrica que queremos minimizar. Intuitivamente, ao elaborar essas funções de recompensa, queremos que mais *switches* sejam usados para transportar todo o tráfego enviado pela origem ao destino. Dividir o tráfego entre caminhos diferentes usaria mais recursos por menos tempo, resul-

Tabela A.2: Configuração dos experimentos para a análise funcional. Baseline corresponde ao uso do controlador Floodlight sem intervenções de aprendizado de máquina.

Agente	**Função de recompensa**	**Número de FE**	**Topologia**
WeightedUsage	Média ponderada da utilização dos enlaces	5	S1
UsageHarmonicMean	Média harmônica da utilização dos enlaces	5	S1
UsageStandardDeviation	Desvio padrão da utilização dos enlaces	5	S1
Baseline	-	5	S1

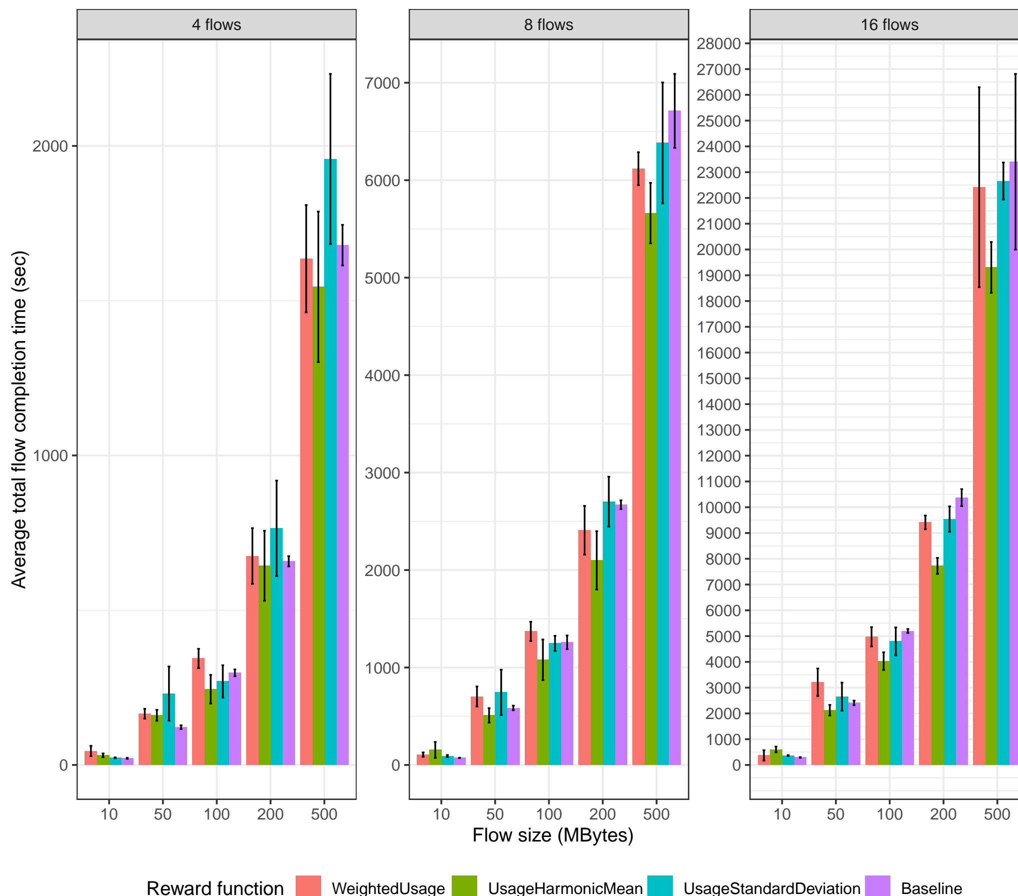
tando em menor tempo de conclusão do fluxo total. Executamos cada configuração de experimento cinco vezes para analisar os valores de média e desvio padrão. Calculamos o tempo total de conclusão do fluxo como a soma de todos os tempos de conclusão dos fluxos. Usamos essa métrica para avaliar (i) se o agente pode equilibrar a carga de trabalho, (ii) se há uma estratégia de recompensa que pode equilibrar essa carga de trabalho melhor em comparação com uma rota padrão fornecida pelo controlador Floodlight e (iii) escalabilidade em termos de o número de fluxos. Por outro lado, a utilização de memória é a métrica que usamos para avaliar os custos computacionais de cada agente. Ilustramos a configuração desses experimentos na Tabela A.2.

Em nosso trabalho, consideramos uma estratégia de balanceamento de carga eficaz se essa estratégia puder usar recursos de rede por menos tempo - ou seja, se o tempo total de conclusão do fluxo for menor do que uma abordagem que não estava usando uma abordagem de balanceamento de carga. No contexto de nossos experimentos, o controlador Floodlight não estava usando nenhum método de balanceamento de carga e todos os fluxos estavam usando a mesma rota padrão. Intuitivamente, isso resultaria em um tempo de conclusão de fluxo total mais alto porque todos os fluxos estavam disputando os mesmos recursos. Dividir o tráfego entre caminhos diferentes usaria mais recursos por menos tempo, o que resultaria em um tempo de conclusão de fluxo total mais curto.

Considerando os resultados apresentados na Figura A.5, consideramos o agente *UsageHarmonicMean* como o melhor agente para resolver o problema proposto neste trabalho. Esse agente escolheu o melhor conjunto de caminhos que gerou as rotas mais rápidas e, consequentemente, o menor tempo médio de conclusão do fluxo total para fluxos maiores que 100 MBytes. Acreditamos que isso pode estar associado ao fato de que a diferença de recompensas entre um estado bom e um estado ruim é menor ao usar o *UsageHarmonicMean* do que ao usar as outras funções de recompensa.

Por fim, também analisamos o uso de memória para avaliar a escalabilidade do agente, considerando as duas topologias discutidas. Conforme mostrado na Figura A.6, o uso de memória conforme aumentamos o número de fluxos é muito baixo. Usamos

Figura A.5: Média do tempo total de completude dos fluxos em segundos (*average total flow completion time*), em função do tamanho dos fluxos em MBytes (*flow size*). Os resultados são mostrados para os agentes *WeightedUsage*, *UsageHarmonicMean*, *UsageStandardDeviation* e para o controlador Floodlight. As barras de erro representam o desvio padrão de cada conjunto de replicação dos experimentos.



Fonte: a Autora.

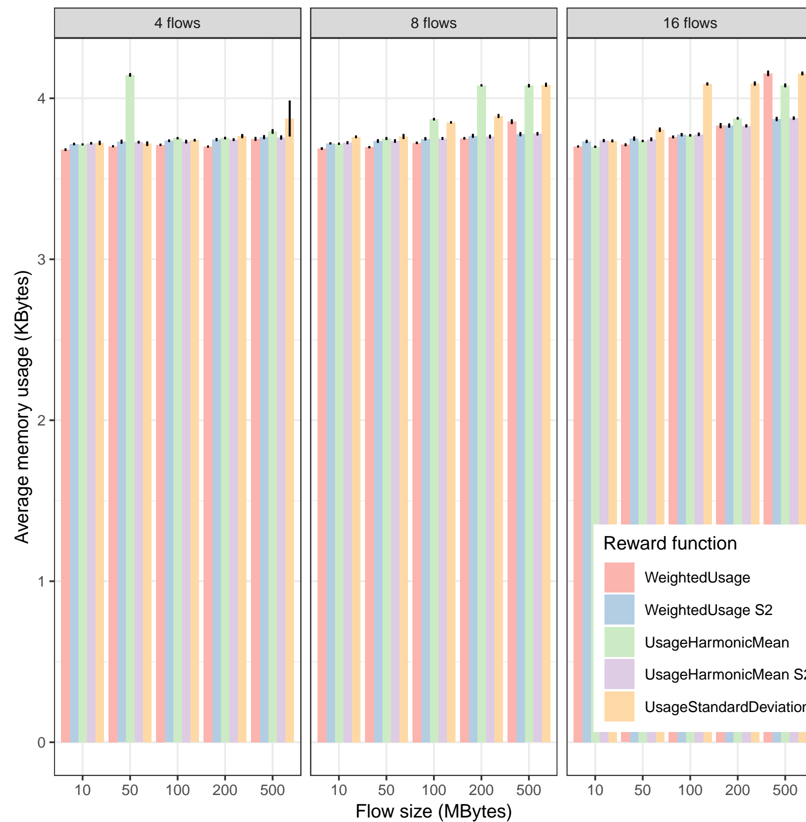
mais memória para agentes que precisam de mais etapas para concluir o fluxo devido ao aprendizado contínuo. Temos alguns *outliers* (especialmente para quatro fluxos de 50 MB), mostrando um maior uso de memória por causa disso - foram necessários mais passos de tempo para completar todos os fluxos ativos.

Considerando que o agente *UsageHarmonicMean* reduziu o FCT e é escalável em relação ao número de *switches*, usamos este agente como base para a análise de inteligência de fluxo elefante (fator EFI), discutida a seguir.

A.5.2.2 Avaliação do fator EFI

A análise anterior mostrou que o agente com função de recompensa que usa média harmônica conseguiu atingir FCT reduzida sem um impacto significativo no uso de memória. Assim, usamos este agente (que chamamos de *UsageHarmonicMean-EFI*) como

Figura A.6: Resultados de utilização média de memória em KBytes (*average memory usage*), em função do tamanho dos fluxos em MBytes (*flow size*) utilizando agentes com diferentes funções de recompensa.



Fonte: a Autora.

Tabela A.3: Distribuição da carga de trabalho, onde #FR é o número de fluxos ratos e #FE é o número de fluxos elefantes.

Carga de trabalho	#FR	#FE	Tamanho dos fluxos
25/75	2	6	50M, 80M, 100M, 200M, 300M, 400M, 800M, 1024M
50/50	4	4	50M, 60M, 80M, 90M, 100M, 400M, 800M, 1024M
75/25	6	2	50M, 60M, 70M, 80M, 90M, 95M, 100M, 1024M

base para os experimentos desta seção, avaliando o fator EFI. Para esta análise, realizamos os experimentos analisando duas variáveis: carga de trabalho e intervalo entre conexões (PIZZUTTI; SCHAEFFER-FILHO, 2018). Usamos três cargas de trabalho diferentes: 25/75, 50/50 e 75/25 como proporção de fluxos ratos e elefantes; três intervalos diferentes entre as conexões: 5, 10 e 15 segundos. Os valores de carga de trabalho mostrados na Tabela A.3 indicam 25/75 como 25 % dos fluxos na rede como fluxos ratos e 75 % como fluxos elefantes. Da mesma forma, 50/50 corresponde a 50 % fluxos ratos e 50 % fluxos elefantes, e 75/25 a 75 % fluxos ratos e 25 % fluxos elefantes.

Executamos os experimentos com todas as combinações possíveis, considerando a topologia S1. Como não tivemos tempo suficiente, não pudemos executar os experimentos

para a topologia S2 - em média, um experimento de 700 passos de tempo é executado em 2 horas. O número de passos de tempo usados para cada configuração de experimento é baseado em várias execuções, onde observamos qual seria o número adequado. Observe que esse número pode variar de acordo com a carga de trabalho e o intervalo entre as conexões: quanto maior o número de fluxos elefantes, maior é o número de passos de tempo necessários para completar todos os fluxos. Da mesma forma, quanto maior o intervalo entre as conexões, maior o número de etapas de tempo. A tabela A.3 mostra como distribuímos a carga de trabalho.

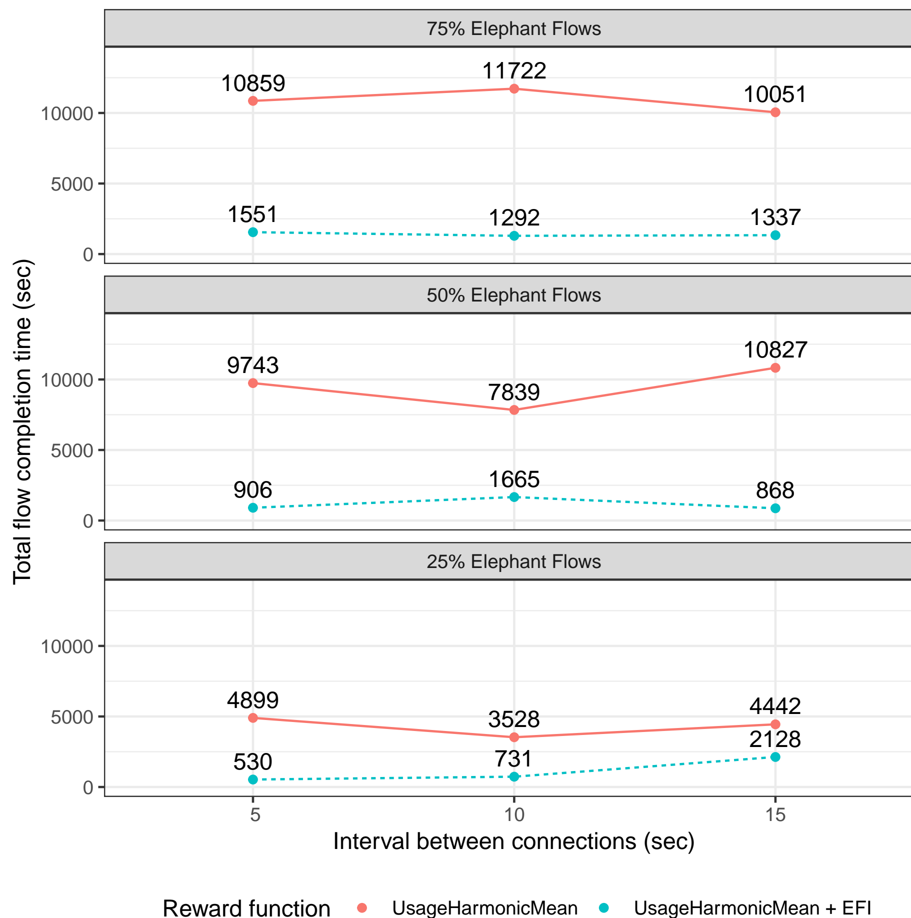
Os resultados mostrados na Figura A.7 demonstram que o fator de identificação de fluxo elefante (EFI) tem uma influência significativa na redução do tempo de conclusão do fluxo total. O agente *UsageHarmonicMean-EFI* rendeu a FCT mais baixa em todos os cenários em comparação com o agente média harmônica base. Entendemos esses resultados como prova de que ao usar abordagens RL, o agente deve se concentrar nos fluxos elefantes, pois eles representam o principal risco ao equilibrar a carga de trabalho entre os recursos da rede. O agente RL se concentrará em equilibrar esses fluxos e ignorar os fluxos menos relevantes em nossa abordagem.

Em todos os cenários, o impacto do uso de EFI é significativo: no pior caso, podemos reduzir o FCT em 52 % (25 % de carga de trabalho com um intervalo de 15 segundos entre as conexões). Isso indica que mesmo considerando a menor quantidade de fluxos elefantes e o intervalo de tempo mais extenso entre as conexões, poderíamos reduzir a FCT ao usar o EFI. Nosso cenário mais favorável ao uso do fator EFI foi considerar uma carga de trabalho de 50/50 e um intervalo de 10 segundos entre as conexões, o que poderia reduzir a FCT em 91 %. Ainda assim, o cenário em que intuitivamente seria necessário mais tempo (75 % da carga de trabalho EF e intervalo de 15 segundos entre as conexões) produziu excelentes resultados: uma redução de 87 % no FCT.

A.6 Considerações finais e trabalhos futuros

Defendemos que o uso de técnicas de aprendizado por reforço para redirecionar os fluxos ratos pode ser ineficiente porque são fluxos de baixo impacto em termos de uso da rede e a sobrecarga para gerenciá-los não compensaria. Assim, propusemos uma abordagem em duas etapas com base no aprendizado por reforço e na previsão do tráfego da rede para equilibrar os fluxos da rede e garantir o uso eficiente dos recursos da rede. A primeira etapa é a previsão do tráfego da rede e é usada para determinar quais fluxos têm

Figura A.7: Resultados de tempo total para conclusão dos fluxos (FCT) usando os agentes *UsageHarmonicMean* e *UsageHarmonicMean-EFI*.



Fonte: a Autora.

o maior impacto nos recursos da rede e podem causar desequilíbrio na rede. A segunda etapa é composta por um agente de aprendizagem por reforço e é utilizada para restabelecer esse equilíbrio, com foco em fazer o melhor uso dos recursos dado o estado atual da rede.

Nossas principais contribuições são (i) modelagem do problema em função de estados e ações em um sistema que visa balancear o tráfego da rede e (ii) uma arquitetura que usa de forma mais criteriosa a aprendizagem por reforço nos fluxos de interesse para o balanceamento de carga. Defendemos que essa abordagem de aprendizado de máquina em duas etapas, que chamamos de aprendizado por reforço antecipado (*Look-Ahead Reinforcement Learning*), pode reduzir erros humanos relacionados ao gerenciamento da rede, evitando interações desnecessárias.

Avaliamos nossa pesquisa usando dois grupos de análise: análise funcional e análise de inteligência de fluxo elefante. Para a primeira análise, consideramos o mesmo modelo com três funções de recompensa diferentes: média ponderada da utilização dos

enlaces da rede, média harmônica da utilização dos enlaces da rede e desvio padrão da utilização dos enlaces da rede. Em seguida, avaliamos se o modelo proposto poderia equilibrar a carga de trabalho entre a rede e, em caso afirmativo, qual das funções de recompensa daria melhores resultados. A segunda análise consistiu em avaliar o impacto da adição de um uma inteligência de fluxos elefante (*Elephant Flow Intelligence* ou EFI) à função de recompensa deste modelo. Essa modificação permitiria ao agente receber recompensas maiores ao escolher uma regra para um fluxo elefante e recompensas menores ao escolher uma regra para fluxos rato.

Os resultados experimentais mostram que em todos os cenários, o impacto do uso de EFI é significativo. Nosso pior resultado reduziu o tempo de conclusão de fluxo (FCT) em 52 %, considerando uma carga de trabalho de 75/25 (75 % de fluxos ratos e 75 % de proporção de fluxos elefantes, com um intervalo de 15 segundos entre as conexões). Como nosso melhor resultado, o EFI reduziu a FCT em 91 %, considerando uma carga de trabalho de 50/50 (50 % de fluxos ratos e 50 % de fluxos elefantes, com um intervalo de 10 segundos entre as conexões).

A solução proposta visa a simplicidade em termos de implementação. No entanto, algumas condições são limitantes para o bom funcionamento de redes reais. Este é o caso de três fatores em nossa implementação: (1) protótipo baseado em uma topologia estática, (2) fluxos ratos ainda podem estar usando uma ação escolhida pelo agente e (3) o número limitado de fluxos ativos considerados na topologia.

Em relação ao protótipo baseado em topologia estática, uma possível otimização seria alterar o estado apenas para considerar os *hosts* envolvidos. Nesse caso, ainda usaríamos o modelo proposto (estado representado pela ocupação dos enlaces) e usaríamos o controlador para obter informações sobre a topologia. Nosso protótipo já usa o módulo de descoberta de topologia do Floodlight para descobrir a topologia de rede e seus caminhos possíveis. O desafio seria usar uma estrutura dinâmica da ferramenta OpenAI para representar os enlaces e possíveis ações.

Em relação a ainda ser capaz de redirecionar os fluxos ratos, exigiria uma mudança no protótipo. Para aplicar uma ação específica para um fluxo específico, o fluxo que precisa ser roteado deve fazer parte do estado do ambiente. Em outras palavras, se o agente apenas olha o estado para escolher a melhor ação, isso significa que o estado deve ser o fluxo que precisa ser redirecionado. Portanto, o estado representaria os fluxos elefantes ativos na rede em um determinado momento. Se não houver fluxos elefantes ativos na rede, o Como a limitação de olhar apenas para um número predefinido de fluxos ativos

está relacionada ao nosso protótipo, acreditamos que, se considerarmos uma ferramenta de aprendizado de máquina que nos permite considerar um estado dinâmico, isso não seria mais um problema. Com esse recurso, todo fluxo elefante pode ser um candidato a ser redirecionado pelo agente.

Finalmente, nossos resultados experimentais demonstram que nossa hipótese de considerar apenas os fluxos elefantes em uma abordagem de aprendizado por reforço para balanceamento de carga de tráfego de rede é válida (usando EFI). Assim, acreditamos que uma boa oportunidade de pesquisa seria utilizar o modelo proposto em nosso trabalho para ambientes de rede onde o tráfego é previsível, como data centers. Acreditamos que nossa abordagem pode melhorar significativamente o uso de recursos de rede e o desempenho do agente de aprendizagem de reforço neste cenário.

Oportunidades de pesquisa adicionais envolvem alterar o estado do modelo para (i) número de fluxos que passam por um *switch* e (ii) considerar apenas os fluxos ativos no estado. Para o primeiro, um possível estado a ser estudado seria considerar quantos fluxos estão ativos em cada *switch* - então poderíamos maximizar o número de fluxos que passam por cada *switch*, ao invés de maximizar o número de links usados. Para o segundo, o estado seria representado como o conjunto de fluxos elefantes ativos, suas rotas e estatísticas correspondentes. Com isso, mudaríamos o modelo para olhar apenas os fluxos elefantes, desconsiderando fluxos ratos.