POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Experimental Assessment of Deep Reinforcement Learning Assisted Optical DC/HPC Network Reconfiguration Methods

Tesi di Laurea Magistrale in
Telecommunications Engineering - Ingegneria delle Telecomunicazioni


Author: **Massimiliano Sica**

Hosting Institution: **University of California Davis**

Student ID: 10558133
Advisor: Prof. Massimo Tornatore
Co-advisors: Prof. S.J. Ben Yoo, Prof. Roberto Proietti, Dr. Sandeep Kumar Singh


Academic Year: 2021-2022

# Abstract

Data center (DC) and high performance computing (HPC) networks are at the roots of any cloud-computing system and are responsible for interconnecting efficiently the different parts of a system architecture. In the recent years cloud-computing has seen an impressive growth with services like AWS, Azure and Google Cloud becoming the standard for almost any tradition IT service. In particular, the wide diffusion of machine learning has led to an increase in workload for the current data center systems which are now dealing with different types of traffic, with different quality-of-service requirements and an increased number of demands. Current data center networks rely on over provisioned static links which are designed to handle worst case scenarios. The current approach not only is very expensive to maintain, but is also inefficient since most of the current data center traffic is highly unpredictable and prone to oscillations. One of the most promising solutions is optical switch reconfiguration, which allows to provision paths depending on the current network situation with very fast switching times if compared to a classical electrical switch. To drive optical switch reconfiguration several heuristic and integer linear programming (ILP) methods have been tried, however they tend to show limited scalability and poor generalization capabilities. To solve the above problem we are going to present a deep reinforcement learning (DRL) based optical reconfiguration method using an experimental testbed. DRL allows to generalize to different scenarios without explicit training. The specific goal is to show that optical reconfiguration can indeed improve the training performance of distributed machine learning workloads in case of network congestion. By changing the topology of the network dynamically, optical switch reconfiguration can generate new topologies where it is possible for the routing algorithm to route the distributed machine learning traffic on a path which is less congested than the one where it was running pre-reconfiguration. By setting up the testbed with the proper number of servers, a real time network monitoring system and a routing algorithm we were able to show a 5x training time decrease for the deployed distributed computer vision algorithm. In addition, using a self-supervised learning algorithm we were able to improve the training of the agent leading to a 29% less network collapses.

**Keywords:** optical network reconfiguration, data center networks, reinforcement learn-

ing, distributed machine learning, self-supervised learning

# Abstract in lingua italiana

Le reti di data center (DC) e High Performance Computing (HPC) sono alla base di qualsiasi sistema di cloud-computing e sono responsabili interconnettere efficientemente diverse parti di sistema. Negli ultimi anni il cloud-computing ha visto una crescita impressionante con servizi come AWS, Azure e Google Cloud che sono diventati lo standard per quasi tutti i servizi IT tradizionali. In particolare, l'ampia diffusione del machine learning ha comportato un aumento del carico di lavoro per gli attuali sistemi di data center che oggi affrontano tipologie di traffico variegate, con diverse esigenze di qualità del servizio e un numero maggiore di richieste. Le attuali reti di data center si basano su collegamenti statici progettati per gestire i worst case scenarios. L'approccio attuale non solo è molto costoso da mantenere, ma è anche inefficiente poiché la maggior parte del traffico in un data center è altamente imprevedibile. Una delle soluzioni più promettenti si basa sulla riconfigurazione degli switch ottici, che consente di generare nuove connessioni a seconda della situazione della rete con tempi di commutazione molto rapidi rispetto a un classico switch elettrico. Per guidare la riconfigurazione dello switch ottico sono stati testati diversi algoritmi euristici e di integer linear programming (ILP), tuttavia tendono per mostrare una scalabilità limitata e scarse capacità di generalizzazione. Per risolvere il problema di cui sopra, presenterò un metodo di riconfigurazione ottica basato sul deep reinforcement learning (DRL) utilizzando una testbed sperimentale. Il DRL permette di generalizzare in scenari in cui non e' stato esplicitamente addestrato. L'obiettivo è dimostrare che la riconfigurazione ottica può effettivamente migliorare le prestazioni di addestramento dei carichi di lavoro di machine learning distribuiti. Cambiando dinamicamente la topologia della rete, la riconfigurazione ottica può generare nuove toplogie dove è possibile per l'algoritmo di routing instradare il traffico del carico distribuito su un percorso che è meno congestionato di quello pre-reconfigurazione. Configurando il banco di prova con il numero corretto di server, un sistema di monitoraggio della rete in tempo reale e un algoritmo di routing, sono stato in grado di mostrare una riduzione del tempo di addestramento di 5 volte per l'algoritmo di visione artificiale distribuito. Inoltre, usando un algoritmo self-supervised sono stato in grando di migliorare il processo di training dell'agente riducendo il numero di collassi di rete del 29%.

**Parole chiave:** riconfigurazione reti ottiche, reti per data center, reinforcemnt learning, machine learning distribuito, self-supervised learning

# Contents

# 1 | Introduction

This introductory chapter is going to provide an overview on the structure of this thesis and on the reasons why the topic of optical reconfiguration for data center and high performance computing networks was chosen. In the last section, the problem statement, a more detailed explanation of what the developed algorithm does, the results obtained and an example scenario are provided.

## 1.1. Thesis Outline

This thesis is organized in 6 chapters including this one and they are organized as follows:

- Chapter 1: presents the motivation behind this work and the problem statement addressed with this study
- Chapter 2: provides an overview of DC/HPC networks, discusses how optical reconfiguration can help, and presents the related work
- Chapter 3: presents general reinforcement learning (RL) theory, two DRL algorithms and a self-supervised DRL method and analyzes their advantages
- Chapter 4: is the biggest chapter where the proposed approach is presented together with all the tools used to develop it
- Chapter 5: presents and analyzes the experimental results
- Chapter 6: concludes this thesis summarizing the major outcomes and discussing possible future work directions.

## 1.2. Overview and Motivation

*Data center networking is the integration of a constellation of networking resources switching, routing, load balancing, analytics, etc. to facilitate the storage and processing of applications and data* -VMware Glossary [47]

Data center and high performance computing networks are at the roots of any cloud-computing system and are responsible for interconnecting efficiently the different parts of a system architecture. They rely on high bandwidth optical links in order to be able to deal with the massive amount of data generated every second. In the recent years cloud-computing has seen an impressive growth with services like AWS, Azure and Google Cloud becoming the standard for almost any tradition IT service (storage, computing, networking etc...). The growth of these services also led to an increase in the number of users with different needs, and consequently to more complex traffic patterns. For this reason, today's DC and HPC networks are characterized by a progressively more spiky and unpredictable traffic due to the vast number of applications running and the constant demand from users. Each user/ application tries to meet different requirements and meeting all of them can be complex. Current DC/HPC architectures have dealt with the problem by using static, over provisioned architectures which are designed to handle worst case scenarios. The problem with this kind of architectures is that they are not able to efficiently adapt to the modern unpredictable data center traffic and this leads to quality-of-service (QoS) requirements not being met. In addition, worst case static provisioning comes with very expensive useless cabling, excessive heat production and strong power consumption. Current state of the art solutions rely mainly on ILP, which are very hard to solve online even with the most advanced solvers, or heuristics algorithms which can show generalization issues. Even the current DRL based solutions show some problems. Specifically, DRL follows a trial-and-error approach which leads to making a lot of mistakes (taking sub-optimal actions or failure-leading actions) that can slow down the training process. DRL training can be lengthy, energy consuming and data-hungry. This work is going to focus on how to move beyond statically wired networks and develop a smart infrastructure capable of provisioning dynamically and the needed resources for each application leading to an improvement of QoS and to a simpler architecture. First, a DRL agent is developed to improve the training performance of the distributed machine learning algorithm and act as a baseline. After a self-supervised methodology is implemented in order to improve the convergence time of the base agent and reduce the number of visits to failure states (network collapse).

## 1.3. Problem Statement

The explosion of machine learning in the recent years has led to a considerable increase within data center infrastructure of these kinds of workloads which consume a relevant amount of both computer and network resources [9]. According to [20], "Since 2012, the amount of compute in the largest AI training jobs has been increasing exponentially with

a 3.4-month doubling time, 50x times faster than the pace of Moore's Law". In this context, being able to optimize the execution of such type of traffic can lead to great savings for the company owning the data center and to an improvement of quality-of-service for the customers. To support this thesis, we decided to run an experiment to understand how much improvement can be obtained with reconfiguration. We deployed the distributed machine learning (DML) algorithm using the testbed shown in 4.1 with and without background traffic. The DML traffic captured by Sflow is shown in figure 1.2. The background traffic consisted in congesting a single link with 9Gb/s UDP traffic generated by Iperf. we deployed the machine learning training over the following virtual machines (VM): VM1, VM2, VM3 and VM4, and we generated Iperf traffic between VM7 and VM8. The difference in training time between the two is simply impressive, the non congested scenario completed a 100 epoch training in about 1950 seconds while the congested one completed in about 3548 seconds. The increase in training time is about 45% which means that a proper optical reconfiguration algorithm could make a real difference.

Given the above, the goal of this work is to provide a novel algorithm based on deep DRL to provide efficient optical switch reconfiguration to improve the training time of DML applications running in a data center/super computer while also improving the performance of the DRL agent using a self-supervised module.

The input to our algorithm is the physical topology of the testbed in Figure 4.1 that we are using to test the algorithm, and a the current network metrics obtained from the top-of-the-rack (ToR) switches. The system is responsible for the following:

- Traffic monitoring
- Traffic matrix estimation
- Routing
- Optical Switch Reconfiguration

All of the above are implemented by three components: the DRL agent, the traffic monitor and the flow manager which are shown in figure 1.1. The output of the algorithm is a new configuration for the optical switch and the new flow tables for each ToR. For the development of the DRL agent we are going to compare the performance of two well known algorithms: Deep Q-learning and Deep deterministic policy gradient. Once a baseline is obtained, a self-supervised approach is used to improve the performance of the agent itself in terms of convergence time and visits to the failure state.

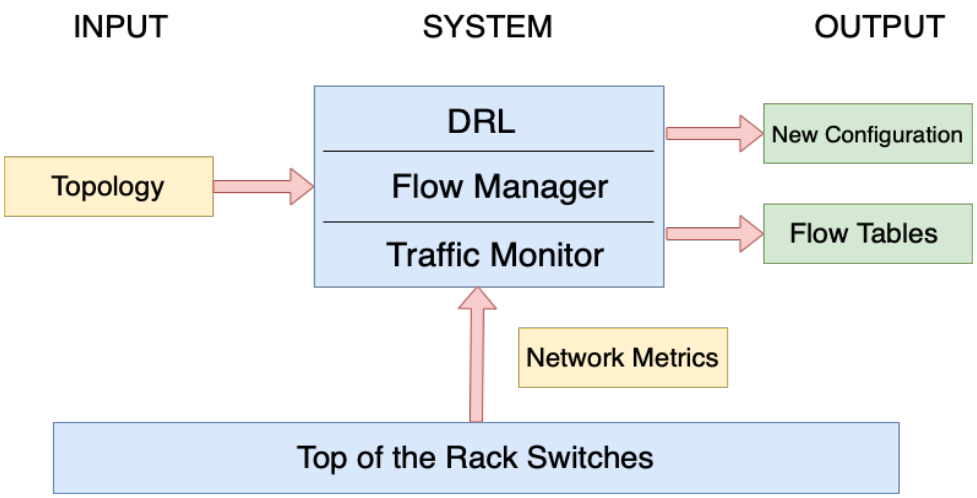


Figure 1.1: Input-output diagram

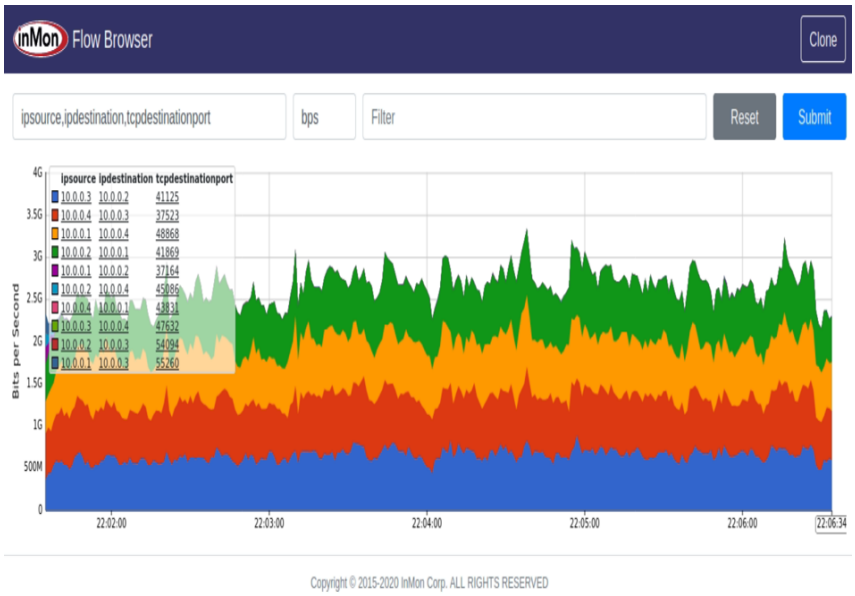


Figure 1.2: Example of distributed machine learning traffic

# 2 | Optical Reconfiguration in DC/HPC Networks

In this chapter some fundamental background topics, useful for the development of the optical reconfiguration algorithm of this thesis, are presented. It is introduced what data center and high performance computing networks are, and how optical reconfiguration and SDN can be game changers for the current cloud- computing architectures. Eventually, the state of the art in machine learning (ML) for optical reconfiguration is described.

## 2.1. DC/HPC Networks in a Nutshell

Data center and high performance computing networks are the backbone infrastructure for any modern cloud-computing system. They provide high speed connectivity between the different servers that compose the system. Given the growing popularity of cloud-computing, DC/HPC networks are responsible to carry high bandwidths and to guarantee QoS in more complex scenarios [19]. These networks need very fast routing to be able to manage the huge amount of network traffic and the velocity at which the demand set changes. The authors of [51], propose an SDN based routing algorithm capable of outperforming current network protocols like OSPF in terms of convergence time and response. In addition to being efficient, routing algorithms need to be able to adapt to changing topologies, since the problem of optical network reconfiguration is a dynamic one [42]. Currently, DC/HPC networks rely on static wiring to provision worst-case scenario bandwidths ([1], [10]), however such solution leads to poorly performing networks since the bandwidth is not allocated dynamically according to the current needs [2]. In addition to being inefficient performance wise [3], they are also lacking in terms of energy efficiency. In [31] and [45], the authors mention that the total power consumption of data centers increased globally from 24GW to 38GW (63%) between 2011 and 2012. This can lead to significant bottlenecks in certain areas of the network, while other links are completely available. In order to solve this issue we need to rethink completely the current architecture and move to a more flexible architecture which allocates bandwidth

dynamically. However, in order to build this system we need to develop a data plane that is easily re-configurable no matter the size and that is reconfigured fast, to minimize the data loss. The two ingredients to obtain such architecture are optical reconfiguration and software defined networking (SDN). One example is [42] where the authors try to develop an efficient routing algorithm to deal with the flexible interconnects of re configurable DC/HPC networks using OpenFlow and Ryu.

## 2.2. The Power of Optical Reconfiguration and SDN

Optical switching technology can enable fast reconfiguration within a data center network since optical interconnections can be provisioned quickly and the bandwidth is much higher that the average electronic packet switch. In this section, two papers that express the potential of these two technologies are presented.

The first paper, Flexspander [43] is an example of successful integration of optical circuit switches (OCS) within mainly electronic packet switches systems. The authors of Flexspander claim that most DC/HPC systems are currently connected using either a Hierarchical Clos [24] or a Dragonfly [21] topology which are statically wired and can lead to lack of flexibility and suffer performance degradation. In this scenario researchers started looking at optical switching and developed a reconfigurable topology called Flexfly [50] capable of leveraging silicon photonics (SiP) switches to build a re configurable architecture with nanosecond switching. The authors of [43] decided to build on top of this architecture using the concept of expander networks ([40] [44]) which have been proven to be capable of outperforming most of other topologies at the same cost [18]. The result is the Flexpander architecture, which is a re configurable network topology that can be built with an arbitrary combination of commercial electrical packet switches and OCS. Thanks to its flexibility, the authors were able to show an improvement of performance over static network topologies in terms of flow completion time (FCT) and prove the superiority of optical reconfiguration in handling the skewed, unpredictable traffic of today's DC/HPC systems.

Thanks to Flexspander it has been shown that optical circuit switches provide the hardware for an efficient data plane, but that is not enough since it is also important to control synchronously the OCS and all the other electronic components in the system, like for example top-of-the-rack (ToR) switches. To do so the concept of software defined networking [22] is exploited, which consist in separating the data plane from the control plane for better management. On top of SDN any application that can do reconfiguration and management for the network can be run, figure 2.1 shows the concept behind

SDN. Other works have successfully shown the capabilities of SDN in this context like "Software-Defined Networking Control Plane for Seamless Integration of Multiple Silicon Photonic Switches in Datacom Networks" [39]. In this paper, the authors have explored how to integrate silicon photonics based switches into a traditional Ethernet or InfiniBand networks in order to study the feasibility and performance of integrating such switches into the current Datacom network architectures. The SDN controller is used to integrate two technologies that are completely different from one another and manages to make them work together in a synchronized fashion. SDN is used to run the reconfiguration algorithm needed to improve the performance of the network by using the OpenFlow protocol to send messages to both the electronic packets switches (EPS) and silicon photonic switches. Using this setup the authors were able to demonstrate a considerable improvement over the state of the art, in fact the final control plane was able to reach a 344 $\mu$s latency.

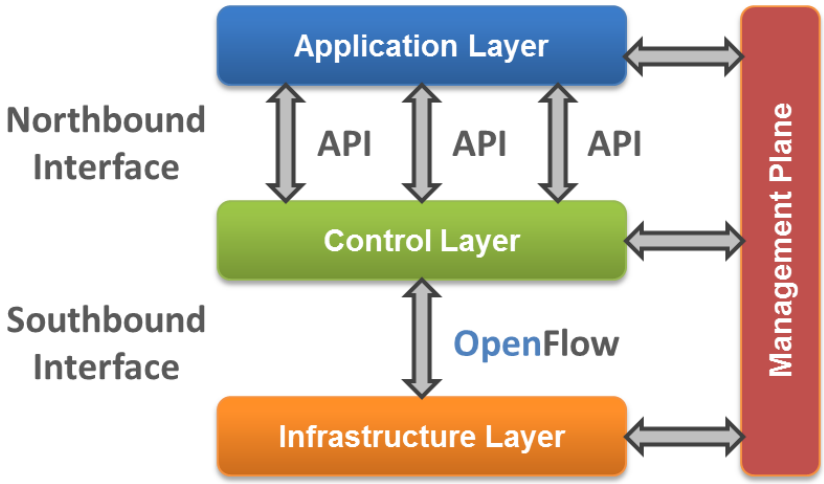


Figure 2.1: Visual explanation of SDN (link)

## 2.3. State of the Art: ML for Optical Reconfiguration

The problem of optical reconfiguration has been treated in literature in multiple occasions. The initial approach taken by most researchers was to write an ILP problem for reconfiguration, however the complexity of these methods did not scale well with the size of the current data center infrastructure as pointed out in ([49], [14], [26], [27]). Also heuristic algorithms have been proven to be non adequate to solve the task since they are often very far from optimal ([49], [5], [8], [48] ). In particular, OSPF [6] and ECMP [46] do not account for current network utilization and lead to sub optimal solutions. In this

context, people started exploring machine learning based solutions that have the potential to scale properly without considerable human intervention [37].

Paper [49] provides an example of machine learning for optical reconfiguration. The authors were able to develop a convolutional neural network architecture capable of finding an optimal or near optimal reconfiguration scheme given as input the current demand matrix and the topology of the network. With respect to an ILP, the proposed neural network does not need to keep solving a problem online. Instead the neural network model is trained offline from from traffic traces which are very common in today's data centers, after the training the new topology inference is found to be quite fast. The overall architecture ends up being quite complex and based on three separate modules responsible for:

- Scoring a traffic demands matrix and topology couple
- Labeling the historical traffic traces with topologies with high scores
- Mapping module to actually map a demand matrix to a topology

Another approach was taken in [13], where the authors developed a DRL based framework for avoiding quality-of-service (QoS) deterioration of applications running within a data center. Whenever a certain application's QoS deteriorates the DRL is triggered and the workloads on the overloaded links are reconfigured. QoS is defined in terms of throughput, latency and packet loss combined together in:

$$QS(t) = T_p - \sum_{s=1}^{S}(k_1^2 La_s + k_2^s Pl_s) \tag{2.1}$$

Where $T_p$ is the throughput, $La_s$ is the latency and $Pl_s$ is the packet loss of reconfigured workloads at time t with QoS s, while $k_1$ and $k_2$ are weighted factors. Every DRL algorithm is characterized, in addition to the reward, by a state and an action space. The state at time t for this solution is made of the following for each workload:

- QoS priority
- Source
- Destination
- Link Utilization : a vector combining the bandwidth utilization of each link normalized between 0 and 1

The action decides how the reconfigured workload is reconfigured in the data plane and is

defined as a distribution ratio vector of length m, where each element is the ratio of traffic allocated to the $m_{th}$ available path. The authors make use of OpenFlow to periodically poll the SDN agents and gather the statistics needed for computing the QS function(2.1) and for checking the QoS status. The authors tested their application using some real world workloads belonging to different categories with very different needs (online serving, scientific computing, offline backup) and obtained significant results up to 6.9% network latency improvement with respect to heuristic reconfiguration methods. However, during the result's collection it emerged that reconfiguring when the QoS is already degraded leads to a decrease in performance. As a follow up for this work the authors in [12] tried to predict a QoS degradation that will lead to a reconfiguration in advance, using recurrent neural networks, in order to avoid the performance deterioration that would follow.

The algorithm presented in this thesis is going to focus on a new self-supervised DRL algorithm capable of optical reconfiguration which is going to exploit real time monitoring of the network traffic to optimize the training time of a DML algorithm. DRL is the way to go in this problem since we want to avoid using ILP and heuristic algorithms to improve generalization capabilities. The real time monitoring is necessary in order to deal with the possible delays that may arise from a polling system and instead react readily to drastic network traffic changes. In addition to the above we will also develop a self-supervised algorithm to improve the convergence of the agent and reduce the number of visits to the failure state. The entire work is going to be carried using an experimental testbed to more closely resemble a real world scenario.

# 3 | Deep Reinforcement Learning Theory and Applications

In this chapter the theory needed to build the general knowledge to understand my thesis work is covered. It will focus on RL in general, Markov Decision processes, deep Q-learning, Deep Deterministic Policy Gradient and a more particular type of RL based on a self-supervised training methodology.

## 3.1. Introduction and Concepts

The concepts of RL explained in this section come from Sutton and Barto [41].

To understand RL we need to start from the concept of "finite Markov Decision Process" (MDP), where we have an agent interacting with an environment which provides back a reward and a new environment state.

MDP can be modeled by:

- States S
- Actions A
- Reward function R
- Discount factor $\gamma \in (0, 1]$
- Policy to learn $\pi$

The dynamics of the MDP can be fully defined by function p (3.1) where $R_t$ and $S_t$ are random variables, for respectively the reward and the state, with well defined discrete probability distributions.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3.1}$$

From the above equation we can derive anything that we want to know about the envi-

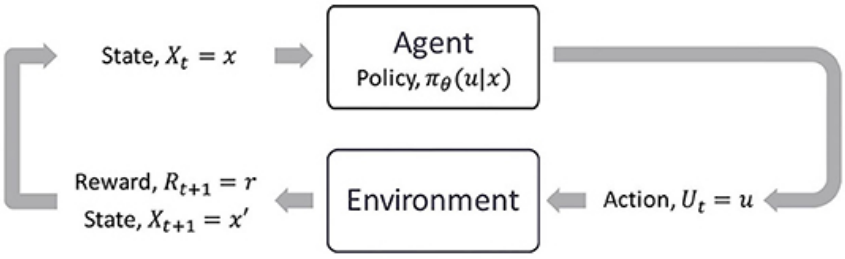


Figure 3.1: Visual explanation of agent environment (link)

ronment like for example the state-transition probabilities:

$$p(s'|s,a) = Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a) \tag{3.2}$$

or the expected reward for a certain state, action, next state triplet:

$$E[R_t|S_{t-1}, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a) \tag{3.3}$$

The idea is to teach the agent how to take actions (find a policy) that are capable of maximizing the cumulative rewards, hence the reward the system provides should be strongly connected to the result we want to achieve. The discount factor $\gamma$ defines how much our algorithm is long or short sighted. For each state we want to evaluate how "good" it is to be there (in terms of expected future rewards) and in this context we define two functions:

- $V_\pi$(s) State Value Function : expected return of being in state s and following policy $\pi$
- $q_\pi$(s,a) Action Value Function : expected return starting from s taking action a and here after following policy $\pi$

$V_\pi$(s) can be estimated using Bellman equation 3.4 and can be used to find the optimal policy, however in reality solving Bellman equation 3.4 is very computationally expensive and often unfeasible since a huge amount of memory would be required to store all the possible states of a real world scenario.

$$V^\pi(s) = \sum_{a \in A} \pi(a|s)[R(s,a) + \gamma \sum_{s \in S} P(s'|s,a)V^{s'}] \tag{3.4}$$

If we want the optimal value for both value ans action value then we get:

$$V^*(s) = max_a\{r(s,a) + \gamma \sum_{s \in S} P(s'|s,a)V^*(s')\} \tag{3.5}$$

$$Q^*_\pi(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)max'_a Q^*(s',a) \tag{3.6}$$

And the reason why we want these is because by rewriting the equation we can get the optimal policy.

The big issue with MDP is that they require a full knowledge of the environment which is, again, almost never the case in real world scenarios.

To tackle the issue of solving Bellman equation, dynamic programming was developed in order to break down the big problem into small recursive sub problems. Dynamic programming starts with policy evaluation where we apply, for every step k, Bellman equation to every state. It can be proved that for k $\rightarrow\infty$ $V_k$ converges to $V_\pi$ for any initial $V_0$ The second component of dynamic programming s policy improvement:

$$\pi^*(s) = argmax_a\{r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V^*(s')\} \tag{3.7}$$

The idea behind policy improvement is acting greedy by choosing the best action at every step. Now if we combine policy evaluation and policy improvement we obtain the so called policy iteration, which converges to the optimal policy by evaluating and improving iteratively.

Dynamic programming despite improving the previous situation still suffers from the curse of dimensionality when the number of states is very big, and it is strongly connected to having full knowledge of the environment which is normally not true. There are other methods called Monte Carlo (MC) methods which rely only on the experience gathered (data) to learn value functions. These methods are still based on policy iteration, however some modifications have to be made:

- Policy Evaluation: MC averages returns observed after visits to s (either every-visit or first-visit)
- Policy Improvement: MC acts greedy with respect to the state action function

The algorithm of policy improvement for MC converges asymptotically only if every state-

action pair is visited, which means that sometimes we need to sacrifice optimality for exploration. This trade off is called exploration-exploitation dilemma.

Before moving on to the final algorithm it is worth mentioning that another problem still needs to be fixed: MC only applies to episodic tasks which is strong limitation. The solution is temporal difference (TD) learning which combines MC with DP.

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{3.8}$$

The equation above shows between squared brackets the temporal-difference error and is what allows us to train online without the need of a complete episode like in MC. Now that we have added this tiny bit of information we can move on to Q-learning, which updates its action value function like:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{3.9}$$

We now have all the tools to understand DRL and how neural networks come into play to improve the performances of these algorithms and provide a tool that could be used on real world use cases.

## 3.2. Deep-Q Learning

*Here we use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network (DQN), that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. [30]*

As mentioned in the previous chapter, solving Bellman's equation extremely complex and it is often impossible in real world scenarios. In this context, neural networks can really help since we can train them to approximate the optimal action-value function 3.6. We are going to refer to the approximate action-value function as $Q(s; a; \theta_i)$ where $\theta_i$ are the weights at iteration i. Introducing a non linear approximator in RL comes at a price since the learning can become unstable due to the high correlation between sequences of observations. The authors of [30] propose two solutions to this problem:

- Experience replay buffer: it allows us to sample data at different time instants to perform training on uncorrelated sequences
- Iterative approach: training is done with respect to a target network which is only periodically updated

To perform experience replay we store the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D$. During learning we apply Q-learning updates on samples of experience drawn uniformly at random from D.

The loss function we use to compute gradient descent looks like:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

where $\theta_i^-$ are the weights of the target network updated every c steps.

Algorithm 3.1 Deep Q-Learning

Randomly initialize neural network $Q(s, a|\theta^Q)$
Initialize target networks Q'
Initialize Replay buffer R
**for** $Episode = 1, M$ **do**
  Receive initial state $s_1$
  **for** $t = 1, T$ **do**
    Generate random probability p
    **if** p >= $\epsilon$ **then**
      Select action $a_t = argmax_a(Q(s, a|\theta^Q))$
    **else**
      Select random action $a_t$
    **end if**
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ from R
    Sample a random mini batch of M transitions from R
    **if** episode terminated at the next step **then**
      $y_i = r_i$
    **else**
      $y_i = r_i + \gamma max_a{}' Q'(s', a'|\theta_i^-)$
    **end if**
    Compute gradient descent $y_i$- $Q(s, a|\theta^Q)$ to update the weights
  **end for**
**end for**

## 3.3. Deep Deterministic Policy Gradient (DDPG)

*We adapt the ideas underlying the success of Deep Q-Learning to the continuous action domain. We present an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces [25].*

Algorithm 3.2 Deep Deterministic Policy Gradient

Randomly initialize critic network and actor network $Q(s, a|\theta^Q)$ and $\mu(s|\theta^\mu)$
Initialize target networks Q' and $\mu'$ with $\theta^{Q'}$ and $\theta^{\mu'}$
Initialize Replay buffer R
**for** $Episode = 1, M$ **do**
    Initialize random process N for exploration
    Receive initial state $s_1$
    **for** $t = 1, T$ **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ from R
        Sample a random mini batch of M transitions from R
    **end for**
**end for**

The main reason behind the development of DDPG [25] is to apply DQN [30] in continuous action spaces where it would be impossible or very complex to evaluate singularly all the possible actions that may occur (greedy policy), since in continuous spaces the amount of possible actions can be outstanding and not realistically manageable computationally. In this context we need to find a different way to estimate the policy function. The solution is to use an actor neural network $\mu(s, \theta^\mu)$ to estimate the current policy(the action to take at every step) and a critic network which provides an estimate of the Bellman's equation (like in DQN) and tells the actor how good it is performing. For what concerns the issue that samples collected close in time are strongly correlated we use a replay buffer like in DQN. To solve the exploration-exploitation dilemma the actor will produce some actions that are going to be affected by some values extracted by sampling an ad hoc random process. Now we should get in the details of how the losses are computed. The critic loss is based on the mean square error formula while the actor loss wants to minimize the mean of the value given by the critic network for the actions taken by the actor network. The target networks for the critic and the actor are updated every $k$ steps using a very

small parameter $\tau$ to track the current weights. The $\tau$ parameter should be much smaller than 1 to avoid divergence.

## 3.4. Self-Supervised Deep Reinforcement Learning

One of the biggest problems of DRL is the lack of awareness when taking certain actions with respect to others. As human beings we are going to treat different actions in different ways [28], for example if we are handling a container made of glass we are going to be more careful with it than if it was made of plastic. The reason behind this is simple, if we dropped a glass container it is going to break and we cannot bring it back to its original state which is not true for the plastic one. The authors of [11] believe that introducing this kind of thinking, known as causal thinking, in the agent's learning process could dramatically speed up the convergence of the rewards. Reversibility estimation allows to teach the agent to maintain a safer behavior in high risk environments, like for example when interacting with elders or handling fragile material. The paper explains how to introduce an auxiliary component to the regular DRL reward which takes into account causality and is estimated using a self-supervised learning process. In order to estimate causality, according to the paper [11] we need to focus our attention on a surrogate task: learning how to predict if an observation (A) comes before another (B). An observation can be considered as a synonym of the combination of state and next state

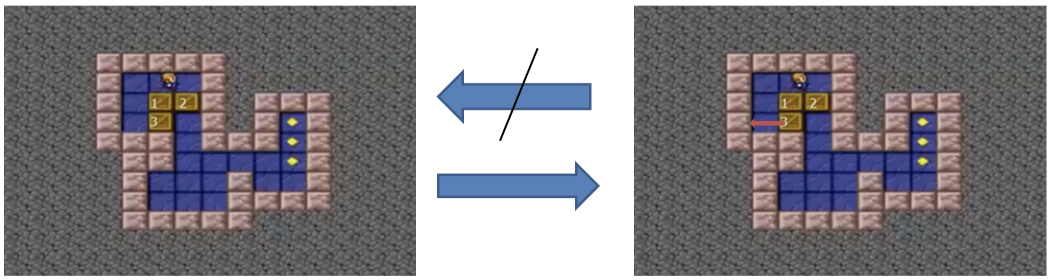

Figure 3.2: Sokoban reversibility example

In figure 3.2 we show how from observation one we move to observation two which leads to failing the task since once the box is placed next to the wall there is no way for the agent to go back and the game has to be restarted. Now that the problem setup is more clear we can move the implementation of the solution . The first step is to find a mathematical way to relate precedence and causality and for doing so we need to define some components:

- Degree of reversibility of an action in K steps: $\phi_{\pi,K}(s,a)$

- Precedence estimator $\psi_\pi(s, s')$ : it provides the probability of visiting state s' after s
- Empirical reversibility $\phi_\pi(s, a) = E[\psi_\pi(s, s')]$

Starting from the above, the authors are able to link precedence and reversibility providing a theoretical bridge between the two.

Now, how can we predict precedence between two observations in a self-supervised way? First of all by self-supervised the authors mean that the neural network is going to train itself without the need of a labeled data set. The neural network is going to take as target two temporally ordered states, distant no more than w steps from each other, and as input the same states but shuffled. The network is supposed to tell which of the two states comes first and it is going to know if the prediction is correct or not thanks to the target.The algorithm generates an embedding for both the states using a Siamese network which are concatenated together and then passed to a feed forward neural network for temporal order prediction. The probability is updated via negative log likelihood against the result of the shuffle so that it matches the actual temporal order. With this setup the authors define two possible ways to operate the algorithm: a version which tries to avoid irreversible actions (RAE: Reversibility Aware Exploration) and a version which prohibits irreversible actions (RAC: Reversibility Aware Control). Both of these algorithms were able to improve considerably the performance of a basic IMPALA agent [7]. For what concerns RAC with offline training, in the use case of the cart pole problem [33] it achieves the maximum score immediately despite the use of a random policy agent. If we deploy RAC online, we are going to experience a slower learning curve, which however leads to a very safe behavior.

# 4 | Solution Development

This chapter is the most relevant, since here the solution and the tools needed to develop it are covered in detail. The overall architecture of the system, which includes the testbed (virtual machines, ToR, optical switch and links) and the external servers that we use to run different applications, is shown. After that, the chapter will dive into the physical structure of the testbed and the configuration procedure for the virtual machines. Following the testbed configuration more information about the workloads deployed for the experiment will be given. After that, the implementation of traffic monitoring, routing and the DRL algorithm is explained. The chapter ends by describing the experimental setup.

## 4.1. Overall Architecture

The whole system architecture is shown in figure 4.1. The data plane is made by 6 virtual machines, 4 ToR switches and an optical switch, while the control plane is handled between two servers, the controller and the monitor. The monitor is the brain of our system and it is where our algorithm is going to run, while the controller takes care of running the Ryu ofctl rest API [38] which allows the monitor to interface with the data plane using OpenFlow. In addition the controller is configured as Sflow [16] collector to provide real time monitoring capabilities to our system.

## 4.2. Testbed Description

In this subsection we will go more in the details for what concerns the structure of the DC testbed available at the NGNCS lab of UC Davis.

The DC testbed is made up by four virtual servers running in two separate physical servers (Node1 and Node2) connected by four virtual ToR switches built inside a physical ToR running PicOS [36]. At the center of the topology there is the microelectromechanical system (MEMS) based optical switch (figure 4.3 ), and the whole infrastructure is controlled by the monitor and the controller which are installed in two different servers.

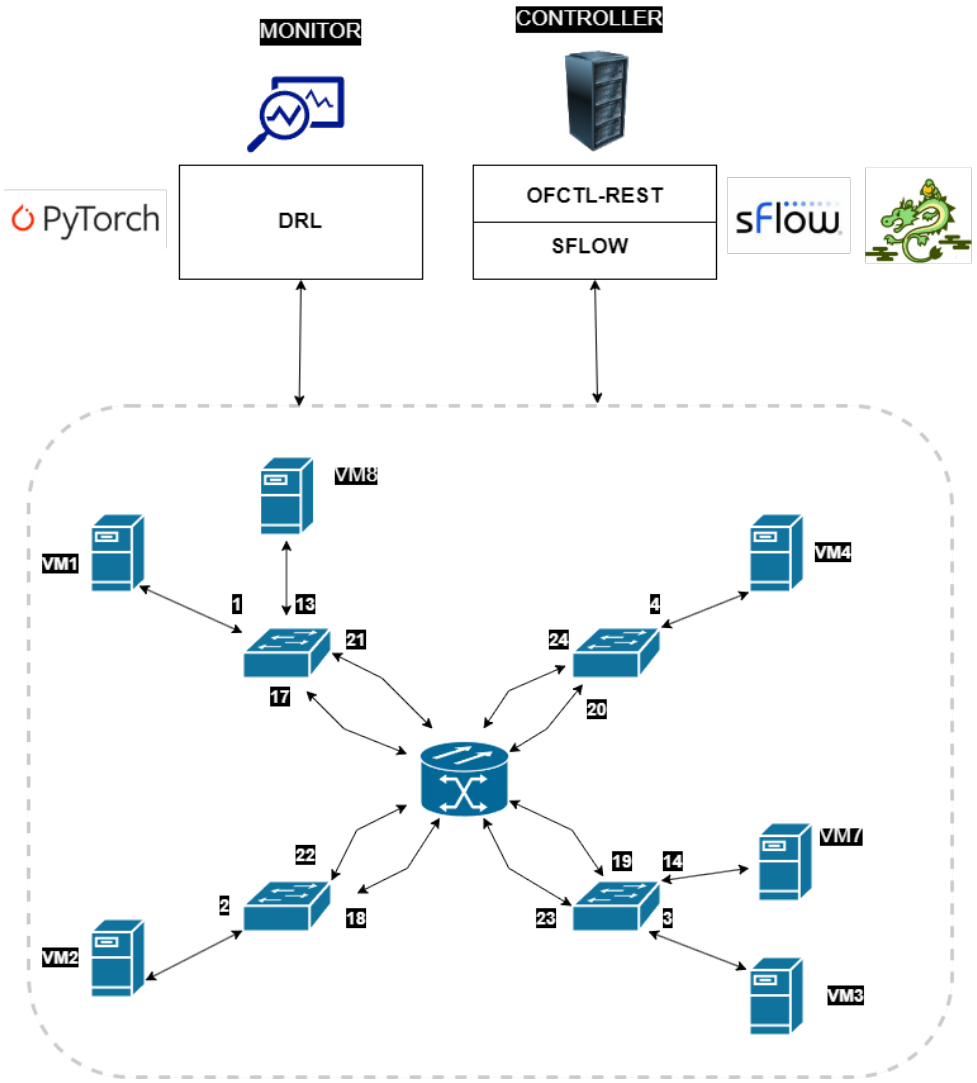


Figure 4.1: System's architecture

The controller provides an interface for the Ryu rest API (ofctl rest) [38] which allows to manage the virtual bridges using a SDN approach. On the other hand, the monitor acts as an orchestrator and sends out the commands to the controller calling the needed ofctl rest method. The monitor is also the server where our deep RL runs and where the reconfiguration commands are sent to the OCS. The optical switch receives configuration instructions via a TCP socket using Standard Commands for Programmable Instruments (SCPI). The optical circuit switch deserves a few additional notes, since it provides a number of ports that can be connected dynamically leading to variations in the topology (number of edges and connected VMs). In particular the VMs are connected like shown in 4.1. This means that every time that a reconfiguration is required, the DRL algorithm will connect the VMs in a different way leading to a new topology graph.

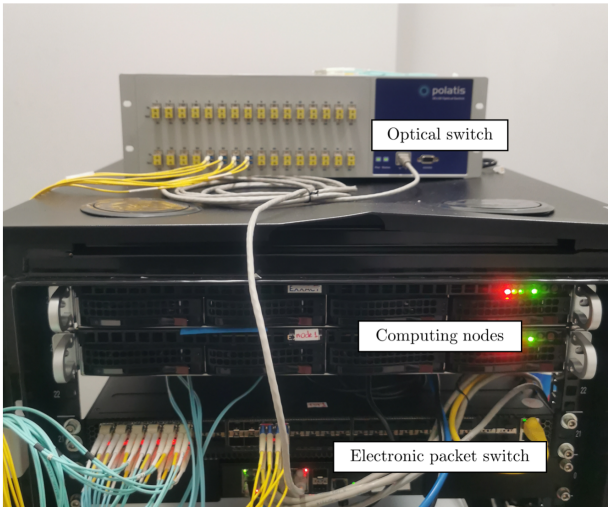


Figure 4.2: Physical testbed [4]

| | VM | In Port | Out Port |
|---|---|---|---|
| **row1** | VM1 | 17 | 49 |
| **row2** | VM1 | 21 | 53 |
| **row3** | VM2 | 18 | 50 |
| **row4** | VM2 | 22 | 54 |
| **row5** | VM3 | 23 | 55 |
| **row6** | VM3 | 19 | 51 |
| **row7** | VM4 | 20 | 52 |
| **row8** | VM4 | 24 | 56 |

Table 4.1: Virtual machine to optical switch port

As mentioned earlier in the section we make use of virtualization in order to create two VMs within each computing node. Since configuration and installation were part of the work at UC Davis we are going to provide a description of the procedure here. The very first step was to physically place the server in the rack, the servers are from EXXACT and come with 32 Gb of RAM and an AMD EPYC 7302P 16-Core Processor. To connect the server with the ToR we used two 10GBase-DWDM ER SFP+ SMF 1537.40nm 40Km transceivers. Once the server was placed in the right spot and connected to the rest of the testbed we installed Ubuntu 20.04 in it and an OpenSSH [34] server to be able to work outside the server room. The installation of the virtual machines was carried out using KVM [23]. First of all we need to create a virtual router (with both L2 and L3

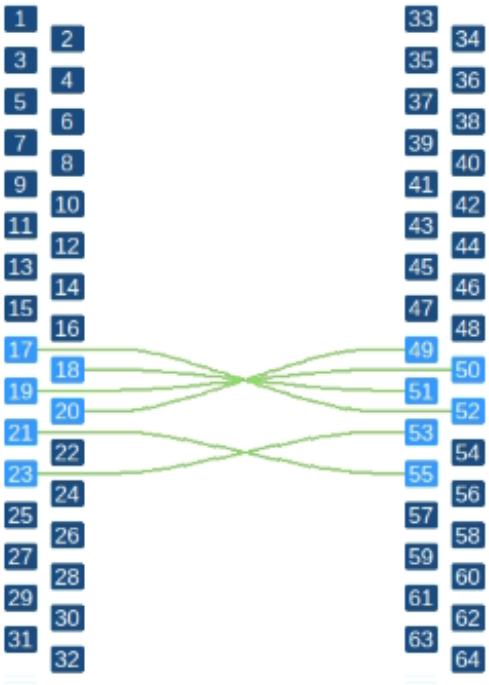


Figure 4.3: Optical switch available cross-connects

configuration) using 1G ports in order to guarantee Internet access to the virtual machines and to act as DNS server. Then we are going to create two bridges linked to the 10G ports via MAC address. Once this whole procedure is completed and correctly running we proceed to create two virtual machines, one for each of the bridges defined before, with 4096 MB of RAM, 10GB of hard disk space and two vCPUs. A picture with a schematic of the internal networking configuration is available in figure 4.4.

## 4.3. Workloads

### 4.3.1. Distributed Machine Learning

For the experiments we run a DML algorithm developed in PyTorch by PhD student Yang Hao at the University of Science and Technology in China. The algorithm is an image classification task of the Cifar10 [32] data set which contains 60000 32x32 colour images in 10 classes, with 6000 images per class. Let us analyze all the keywords one by one. When we talk about image classification we want to be able to assign a new image not belonging to the training set to the correct class (eg: airplane, automobile, bird, cat etc... [32] ). Performing such task is often very demanding in terms of computation and may not be possible to solve it using a single machine and this is when DML comes into play.

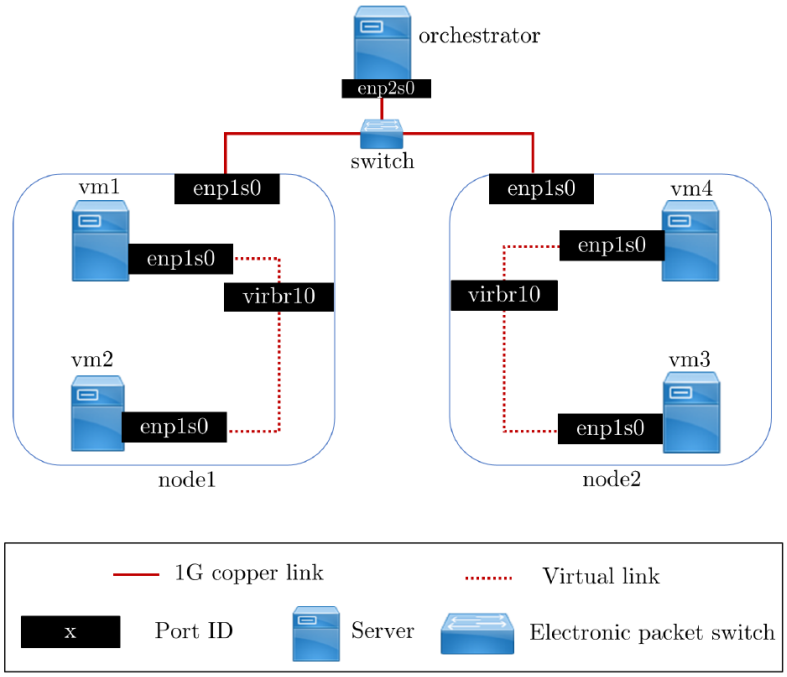


Figure 4.4: Virtual machines configuration visual explanation [4]

DML allows to train the neural network model in different nodes communicating with one another following, in this case, a ring all-reduce pattern (in our topology represented in figure 4.1: $0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$).

### 4.3.2. Iperf and Sflow

In this section we are going to introduce to the reader two fundamental tools used for the development of this work Iperf [17] and Sflow [16].

Iperf is a tool for generating synthetic traffic between an Iperf server and a client. In our specific scenario Iperf has been used to trigger the reconfiguration algorithm by congesting some links using around 9Gb/s of background traffic. Iperf can generate both TCP and UDP traffic, in our experiments we chose to use UDP since we need to congest some links and TCP congestion control would not allow it.

On the other hand, Sflow is a tool to provide real time visibility to software-defined networks and other DevOps systems. It involves the following concepts:

- Agent: embedded in network device (i.e., a switch) converts the raw data into usable metrics
- Collector: responsible for gathering data from all the agents

- Sampling Rate: number of samples taken from an input port of the switch

In our system each switch will be an Agent while the controller will the the collector. Furthermore SFlow offers a rest API which can be easily used from python to gather all the needed metrics [16] and a dashboard that allows to see in real time the traffic evolution in the network.

## 4.4. Traffic Monitoring and Traffic Matrix Estimation

In this section we are going to explain why and how we have implemented traffic monitoring and matrix estimation for this project. The reason why we have implemented such system is to be able to detect when reconfiguration should be triggered in order to improve the performance of the DML and to be able to collect the current demands, IP to IP, which we use to build our traffic matrix. Other groups have explored techniques for network monitoring and metric collection like [35], [29], however we decided to opt for Sflow [16] which substantially simplifies the whole monitoring process.The traffic matrix is estimated by sending an HTTP request to the rest API of Sflow asking for all the flows reported by the agents to the collector. From the answer we build a matrix which we will use later on for routing and for updating the bandwidths at every step (while loop iteration). Traffic monitoring is implemented using a method within the same class that implements traffic matrix estimation. This method takes as input the current topology graph $g$ (made by many edges connecting the different servers) and the traffic matrix. The method will check edge by edge whether the specific edge belongs to a path taken by the DML and it is congested, where by congested we mean that more than 80% of its total bandwidth has been used. If we meet one of such edges we increase a counter called $edges\_non\_conforming$ which we use to determine whether or not we need to reconfigure. Instead, to understand if the network has collapsed (no more DML traffic flowing), we need to keep another counter which will check if for every demand in the traffic matrix there is a path satisfying it, if there isn't we can say that the network has collapsed since the DML requires each of the servers to be reachable by all others in order to exchange information, whenever this condition is not satisfied the traffic stops to flow and the DML training stops. For technical reasons we decided to develop a function that checks whether or not a certain function will lead to a collapse, if it does we are just going to ignore the action and collect the associated reward.

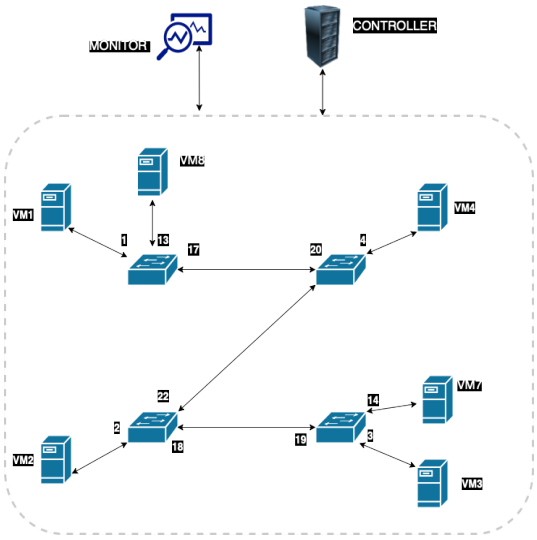


Figure 4.5: Possible topology 1

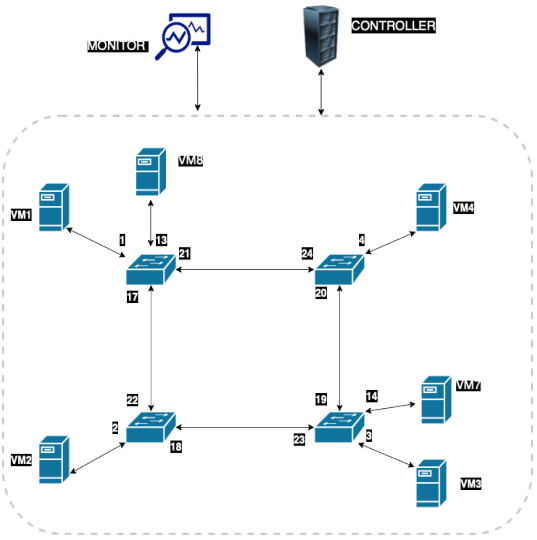


Figure 4.6: Possible topology 2

## 4.5. Routing Algorithm

The routing algorithm is the most crucial component of this system after the DRL since it is meant to provide each demand in the traffic matrix with a path which avoids congestion and allows for a better balancing of the DML traffic. The routing algorithm is responsible for

- Topology generation
- Finding paths for each demand
- Updating bandwidth of each link
- Installing the flows on the ToR switches

Topology generation is the initial step which is needed in order to adjust to the dynamic configuration variations of the OCS. After every reconfiguration, a new topology is generated reflecting the changes in the OCS interconnects. Two examples are given in Fig.4.5 and Fig.4.6

Now let us dive in the procedure described in the second bullet point, which for each demand takes the following steps:

1. Get the first $k$ shortest paths from the tracking topology
2. For each one of them evaluate the bandwidth availability
3. Pick the shortest one which has enough band to satisfy the demand

After we have a path for each demand we update the bandwidth in the tracking topology, which keeps track of the bandwidth usage on each link and it is updated at every time step by sampling the traffic matrix using Sflow [16], this topology is kept in the memory of the program and is used to monitor the congestion levels in the system. The last point is a little more peculiar since once we have a path for each demand we do not install the flows on the ToR immediately, we only install them as a result of a reconfiguration. The reason for doing so is avoiding to install the flows at every monitoring step which leads to instability in the code.

## 4.6. Deep Reinforcement Learning Algorithm

This section is going to take care of the most crucial parts of this work, which are the DRL algorithms, their implementation and testing.

For this work we relied mainly on DDPG [25] and DQN [30] with different setups between one and the other. The DRL agent that we are trying to train is responsible for finding the best reconfiguration for the OCS given a certain traffic condition, which in other words means that it is supposed to generate a new topology where the routing algorithm is capable of finding a path for the DML which is not congested by any other application. Let us start by defining the part of algorithm which are common to both methods. Both the algorithms were trained for a certain number of episodes where each episode is made up by a certain number of steps. For every step the algorithm does the following:

1. Monitor link congestion
2. If the reconfiguration threshold is not met ignore the action and move to the next step
3. Else generate a new configuration
4. If the configuration leads to a collapse ignore it and collect -0.125 reward
5. Else implement it and collect the reward
6. If we reached the best state or failure state start a new episode
7. Else move to the next step

It is important to point out that the reason why we ignore the action is that when working with the testbed actions that collapse the network can create issues at the application level. In fact continuously restarting the DML or simply isolating any of the virtual machines may require to restart the entire program and kill the DML processes via CLI

in the various virtual machines. A flow chart is available in figure 4.7.

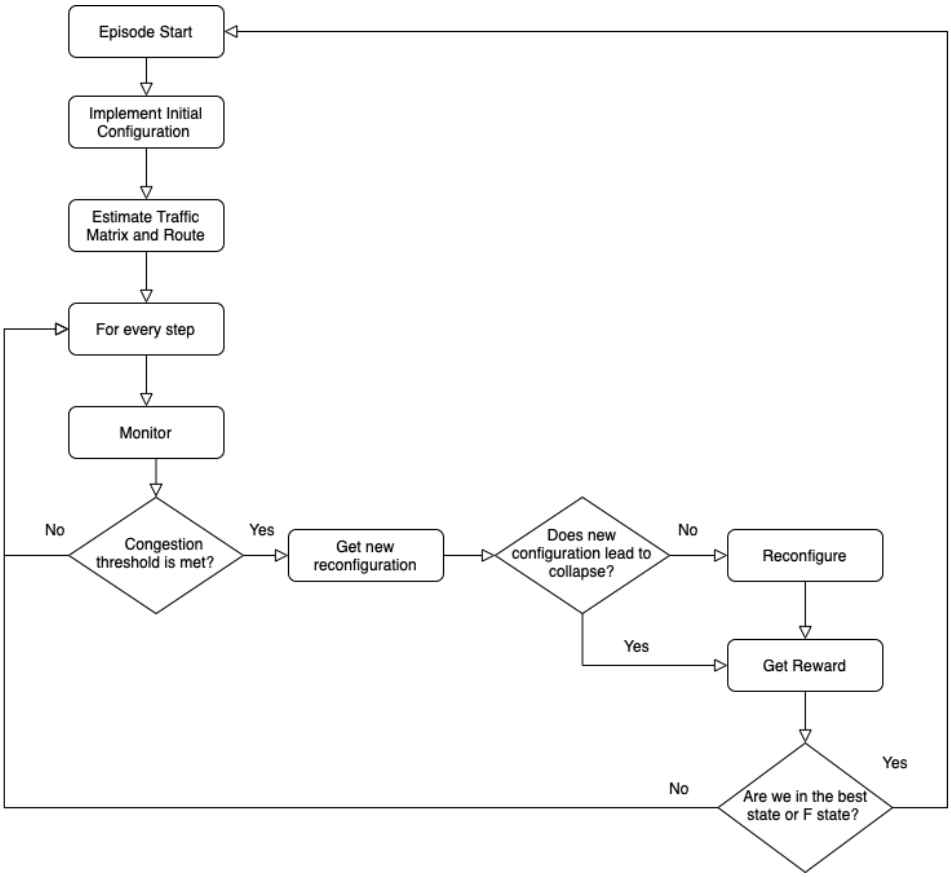


Figure 4.7: Flow chart for DRL training

The reward function A(t) where t is a specific time step of the episode is defined as:

$$A(t) = non\_conforming\_dml(t) - \frac{non\_conforming\_dml(t+1)}{total\_links} \tag{4.1}$$

where $non\_conforming\_dml(t)$ represents the number of links which exceed the congestion threshold and are being used by the DML algorithm at time step $t$. $Total\_links$ instead represents the total number of links in the testbed. The choice of considering only the links used by the DML is of great importance since the final goal of this algorithm is to optimize the performance of the DML not of the other applications running in the testbed. Specifically in DQN the problem definition is as follows:

1. Congestion Threshold: defines the amount of bandwidth usage needed to claim that a link is congested
2. Reconfiguration Threshold: defines the number of congested links needed for reconfiguring

3. State: number of congested links
4. Action: score for each possible action that can be taken
5. Reward: A(t)-A(t+1) for a successful action otherwise zero

Once the neural network outputs the score for every action in the action space, the algorithm acts greedy and picks the highest scoring action. The Markov process associated to the DQN setup is shown in figure 4.9, where each number represents the number of congested links used by the DML. In the setup that we are using, the maximum number of such links is 3, while the minimum is zero. Furthermore each state is connected to the other state by at least one OCS configuration, in the next section we are going to show a more detailed example of the state space and the associated transitions related to the specific experiment we are carrying out. The actions that can be taken to move from one state to another are stored in an array an look like the example in figure 4.2 where the numbers 17 to 24 are the ports of the OCS as can be seen in table 4.1

| | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** |
|---|---|---|---|---|---|---|---|---|
| **17** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **18** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **19** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **20** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **21** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **22** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **23** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **24** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Table 4.2: Optical circuit switch matrix example

We have implemented exploration by following an $\epsilon$ greedy policy where we choose a random action among the available ones with probability less or equal to $\epsilon$ and we decrease $\epsilon$ every four steps.

As for DDPG, the framing of the problem is a little different, since it does not pick an action from a set in a greedy fashion, instead it generates a new OCS matrix made up of continuous values which will need processing in order to be turned into an actionable configuration scheme. While generating a continuous action and then discretize it may sound pointless, the idea behind using DDPG for this problem scenario is to solve the

scalability issues that may incur while working with DQN. DQN evaluates all the available actions in the action space and then chooses greedily which can create some issues if the state space is too big, on the other hand DDPG does not evaluate all the different actions but instead it generates a continuous output that can be interpreted as an action. So the problem definition for DDPG is as follows:

1. Congestion Threshold: defines the amount of bandwidth usage needed to claim that a link is congested
2. Reconfiguration Threshold: defines the number of congested links needed for reconfiguring
3. Action: continuous connectivity matrix of the OCS
4. Reward: $A(t) - A(t+1)$ for a successful action otherwise zero
5. Action post processing: algorithm to turn the continuous matrix into a discrete one

The output of the DDPG actor network is the lower triangular of what will be the connectivity matrix of the OCS after the processing step as shown in figure 4.8.

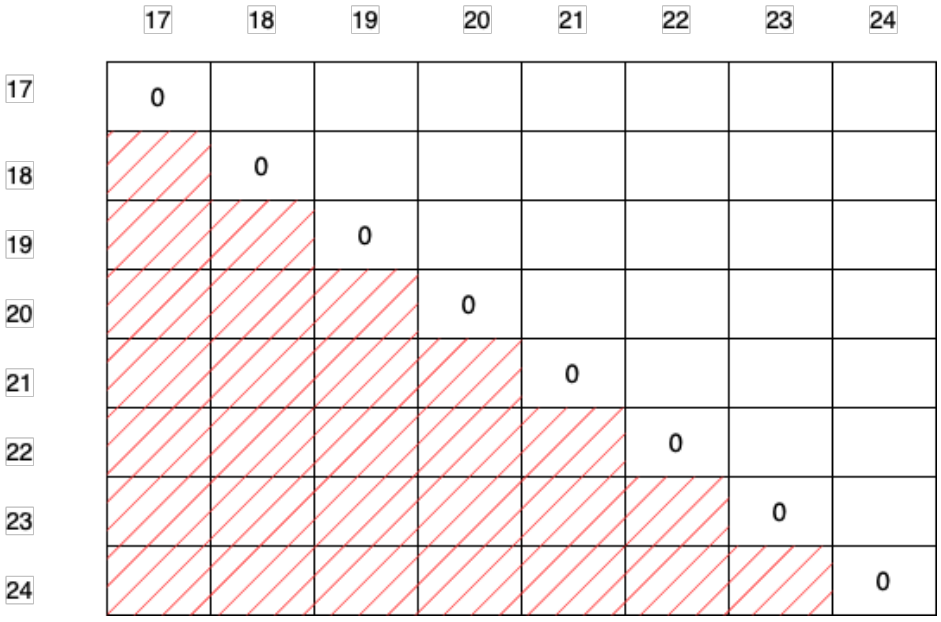


Figure 4.8: Output of DDPG agent (red)

In fact given the algorithm that we use for post processing the action we do not need to output the full matrix because of the proprieties of the latter:

1. The OCS matrix is symmetric since each port should be connected to the other
2. the diagonal is zero since a port cannot be connected to itself

3. Each row and each column can only have a non zero entry to avoid having one port connected to many others.

To enforce the above, for every row we run the following algorithm:

1. Find element in row with the highest value and set it to 1
2. Set to 0 all other elements of the row and of the column
3. Add the zero diagonal and make the matrix symmetric

Once we have generated a proper connectivity matrix we implement it in the OCS and let the routing algorithm take care of serving the demands.

## 4.7. Experimental Setup DQN

Now that all the components of our system have been covered in detail, this section is going to describe the way the experiments were carried out. To train the agent the first step was to deploy the DML on the 4 nodes of the testbed in figure 4.1 to generate the traffic, after two monitoring iterations of the algorithm an Iperf is generated launching around 9 Gigabit/s of UDP traffic between VM 7 and 8. Once the link congestion is detected by the traffic monitor an action is chosen by the agent. Before implementing the action in the OCS however, we check whether or not it will lead to a collapse. If it does we ignore the action and collect a reward of -0.125 otherwise we implement it and collect the associated reward. Exploration is implemented as a random action taken with a probability $\epsilon$ starting from 0.8 and decreased all the way to 0.001 during the various training steps. Choosing the right value of $\epsilon$ is quite critical since a value which is too small may lead to have the agent always stuck taking the same action, while a too big value can make convergence harder. The neural network has the following structure (input x output):

1. input layer: 4x15 ReLu activated
2. normalization layer
3. hidden layer: 15x30 ReLu activated
4. normalization layer
5. output layer: 30x4 ReLu activated

The training is carried out for 2000 episodes with a buffer size of 250, and each episode terminates either after 10 steps or after the optimal (state 0) or failure (F) state is reached.

The reason for terminating the training when the optimal state is reached is because while in the optimal state no reconfiguration is triggered. We decided to train the network over a batch of 4 trajectories (state, action, reward, next state) every 5 time steps and consider it an hyper parameter.

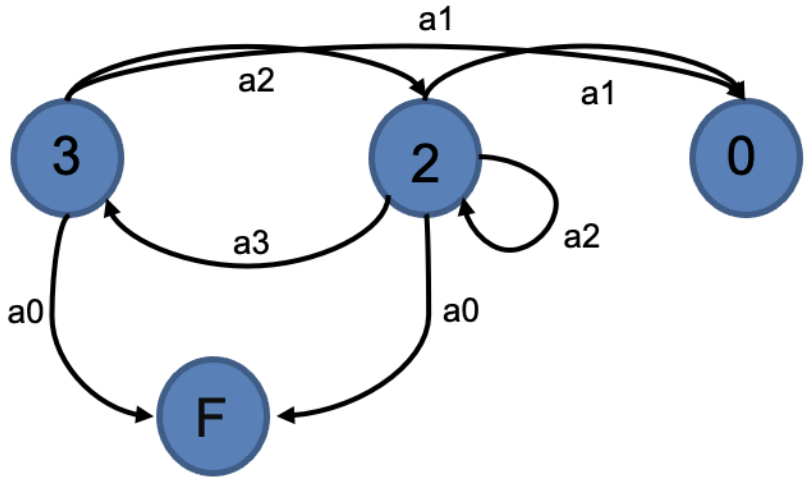


Figure 4.9: Markov process for our experiment

Figure 4.9 shows the Markov process associated to the experiment, every state represents the total number of congested links. We have 4 actions available a0, a1, a2, a3 and a4 which have the following effect:

1. a0: leads to state F
2. a1: leads to state 0
3. a2: leads to state 2
4. a3: leads to state 3

For example state 3 means that there are three congested links in the testbed where the DML is running. In every state we take an action which can lead to any state. The initial state is state 3 whose topology can be seen in figure 4.10 and the optimal state is state 0 (DML is not running over any congested link). The topologies for the different states are shown in 4.10, 4.11, 4.12,4.13. For example in figure 4.10 the Iperf from VM8 to VM7 will take path [13,17,20,22,18,19,14] and the DML will run on exactly the same path since it is the only available one supporting the ring all-reduce patter mentioned in section 4.3. If we analyze state 2 instead (fig. 4.11), the Iperf will congest only the path [13,17,22,18,23,14], leaving link [20,19] free.

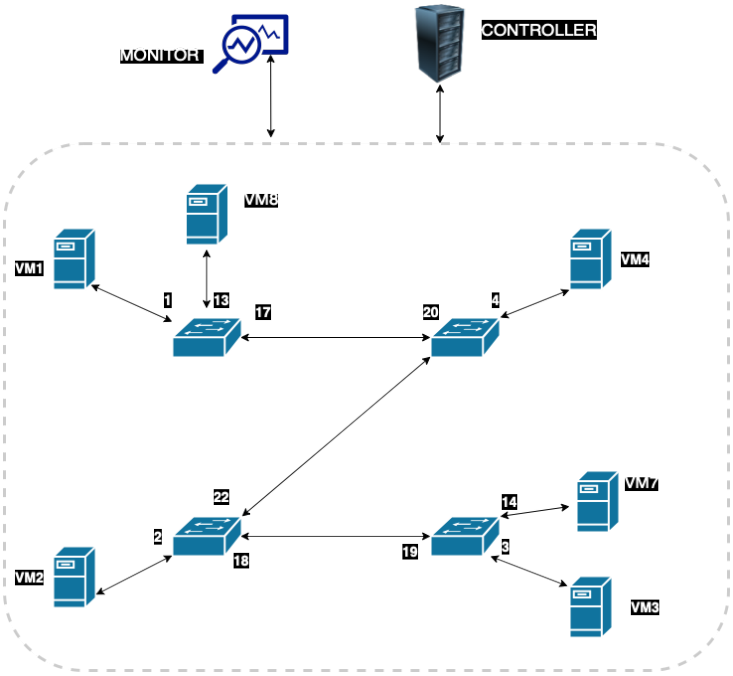


Figure 4.10: State 3

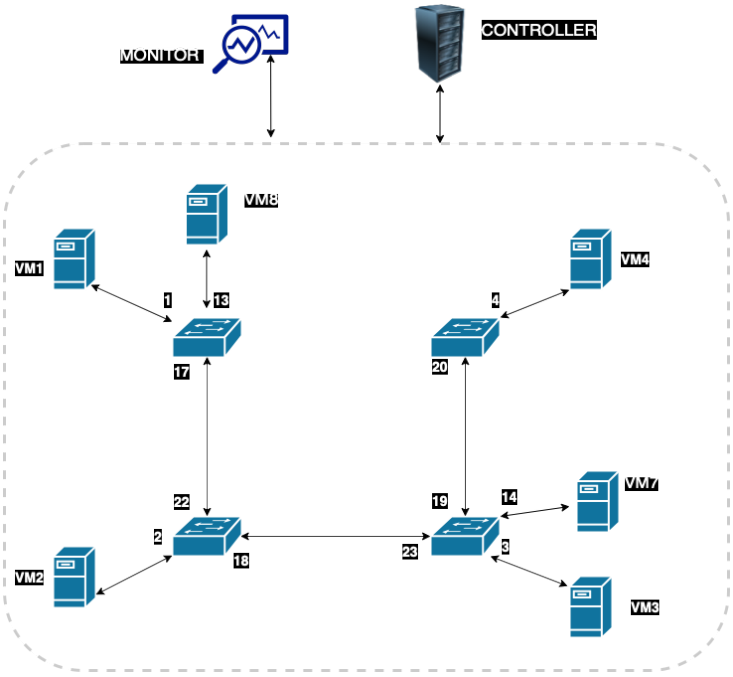


Figure 4.11: State 2

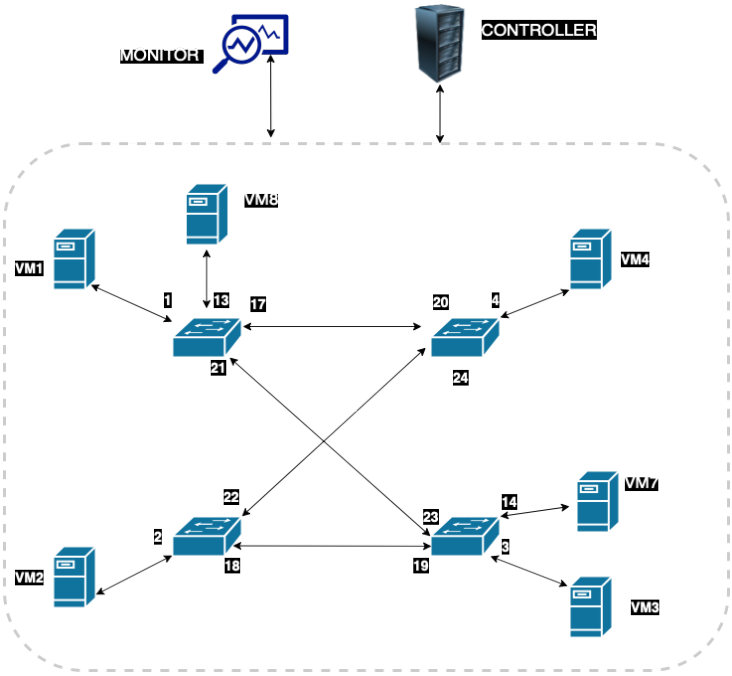


Figure 4.12: State 0

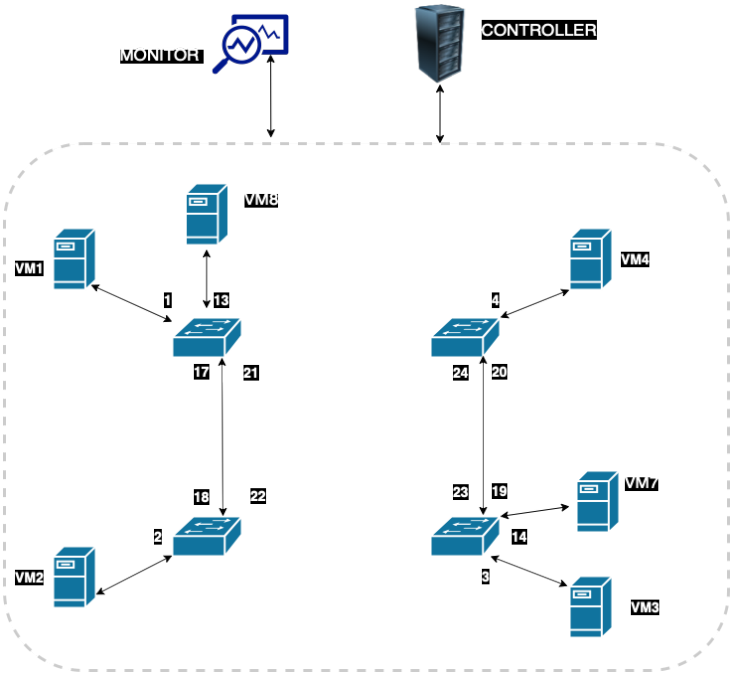


Figure 4.13: State F

The following transitions will lead to these rewards:

1. state 3 to state 2: 0.041
2. state 3 to state 0: 0.125
3. state 2 to state 0: 0.083
4. state 3 or 2 to state F: -0.125
5. state 2 to state 3: -0.041

# 5 | Results Evaluation

This is the final chapter of my thesis, and we are going to show the experimental results of the DQN based reconfiguration algorithm together with a comparison of the performance of the latter with the self-supervised aided version. A small section is also devoted to the results obtained using DDPG which however where not of interest, due to the divergence of the algorithm.

## 5.1. DQN Results

By means of a DQN agent we were able to show an improvement in terms of training time for the DML of about 5x as shown in 5.1

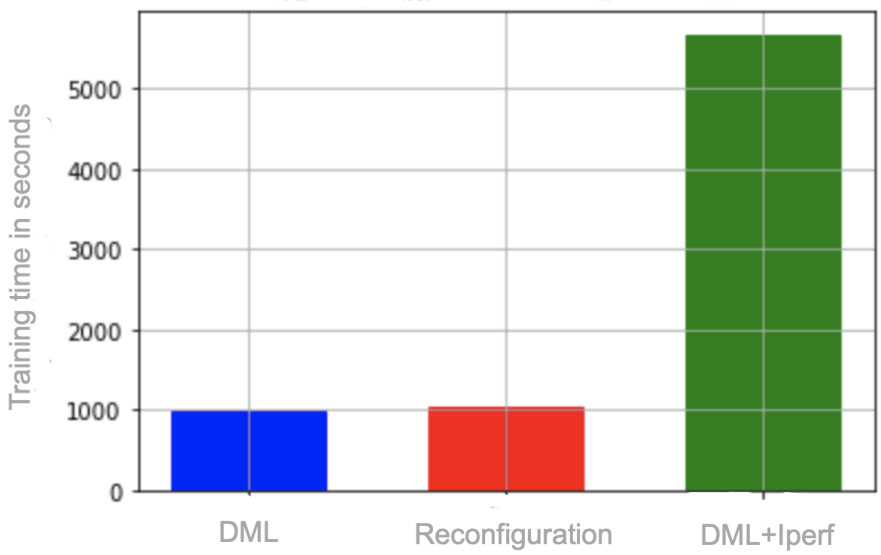


Figure 5.1: Deep Q network performance over 20 epochs

The precise values are:

1. Only DML : 978 seconds (16 minutes)
2. DML and 9 Gbit/s UDP iperf: 56756 (95 minutes)

3. Reconfiguration:1026 seconds (17 minutes)

The results prove the initial assumption made in section 4.1 that using optical reconfiguration can actually lead to significant improvements in the training of machine learning algorithms. Our algorithm is capable, given a certain traffic matrix, to generate a new topology that allows to separate the Iperf and the DML traffic. The agent converges around episode 250, meaning that the test scenario is restarted 250 times before being able to take the best action at the first shot. In figure 5.2 we show the evolution of the rewards, on the x-axis the episodes are shown, while on the y-axis the average reward per episode is shown. Always from figure 5.2 we can see that the rewards oscillate a lot at the beginning, as we would expect, but they stabilize around the maximum (0.125) with some oscillations due to the fact that the exploration probability is never fully zero. The agent's loss is shown in 5.4, the x-axis shows the training epoch while the y-axis shows the value of the loss. A training epoch consists of five episodes, since we train the neural network associated with the agent every five episodes.

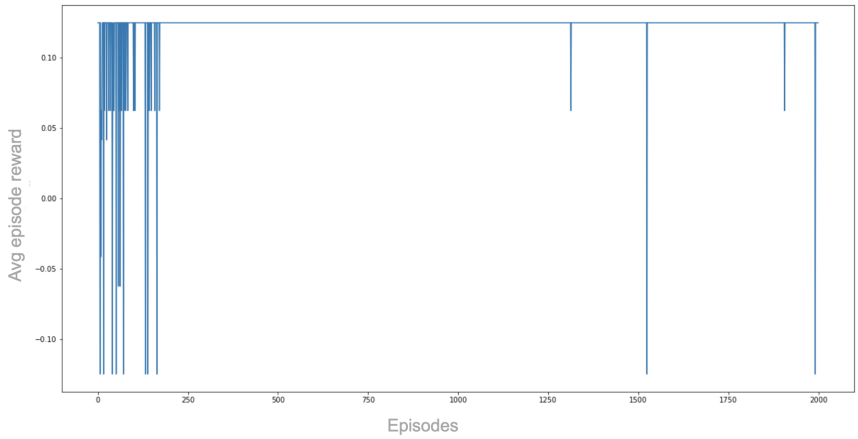


Figure 5.2: Reward evolution for deep Q network agent

It is worth reminding that 0.125 is the best reward attainable by the agent while -0.125 is the worst.In figure 5.3 we show an alternative view for the reward evolution expressed in terms of episode duration, on the x-axis we have the episodes aggregated by a factor five and on the y-axis the duration of the episode (number of collected rewards).Again, after some iterations we reach convergence around a length of 1 since we succeeded to get to the best state at the first attempt. If the algorithm is learning it means that the episodes get shorter since the DRL immediately chooses the optimal action and the episode is terminated just after that. Convergence is attained around epoch 250 with multiple spikes related to exploration. Exploration may lead us to take a non optimal action at a certain time step the episode will run for more steps than it would have if we

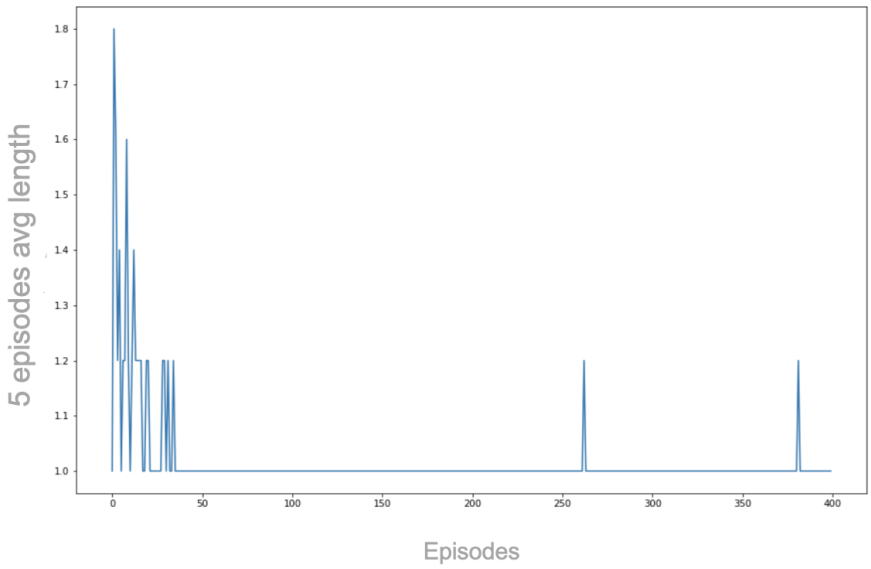


Figure 5.3: Five episode average length evolution for deep Q network agent

picked the best action immediately, since with a probability $\epsilon$ we take a random action from the action set.

## 5.2. Self-Supervised Assisted Results

In order to improve the performance of the DQN algorithm we have tried to implement the self-supervised (ss) reversibility aware algorithm explained in 3.4. To build the self supervised algorithm we need to update the reward function:

$$A(t) = non\_conforming\_dml(t) - \frac{non\_conforming\_dml(t+1)}{total\_links} - SS \qquad (5.1)$$

SS is a variable estimated by the self-supervised neural network and it represents the probability with which a certain state come before another one on average and it is normalized in the range -0.125 and 0.125. Our self supervised network is trained over a batch of four observations and has the following structure:

- one fully connected layer 4x10 ReLu activated
- one fully connected output layer 20x1 Sigmoid activated

The comparison between the self-supervised aided algorithm and the regular DQN is shown in figure 5.5. In the figure is shown how the self-supervised module is capable of

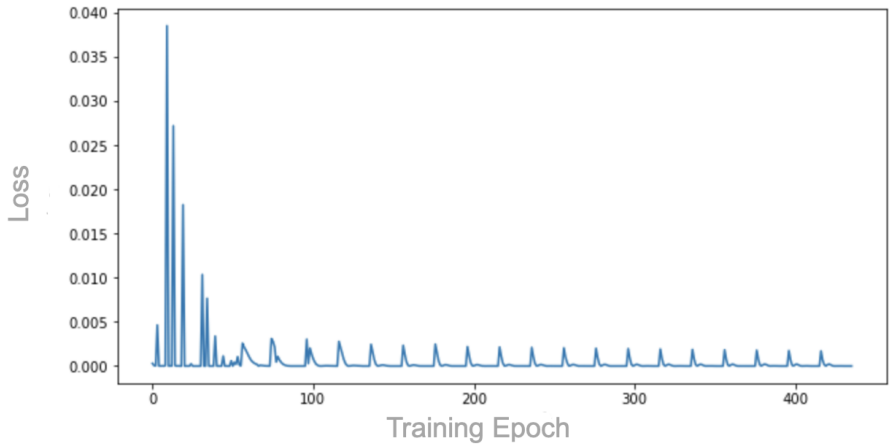


Figure 5.4: Neural network loss per training epoch

speeding up the convergence of the regular DQN module.

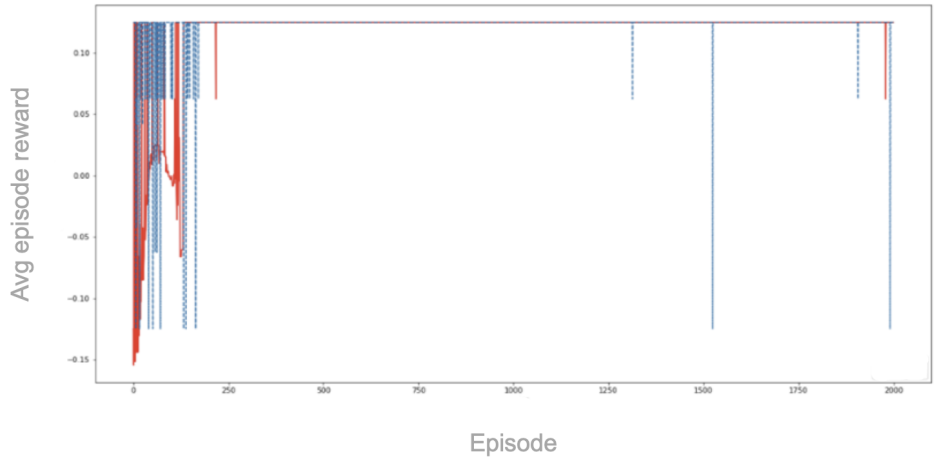


Figure 5.5: Neural network reward evolution

The loss for the self-supervised module is shown in 5.6, however this is the part of the work that needs to be investigate further since the loss for this network is behaving in a slightly unusual way since its convergence is still oscillating around the fixed value of 0.2 instead of smoothly converging there.

To conclude this section we check how many times the failure state is visited in both the self-supervised aided scenario and the regular one. The logic behind this analysis relies on the idea that the self-supervised algorithm is meant to make the agent behave in a more

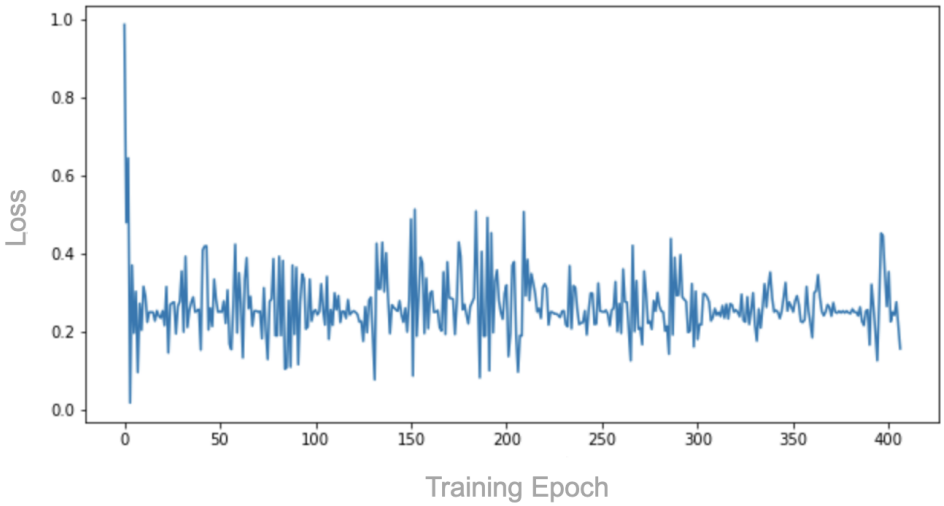


Figure 5.6: Self-supervised network loss per training epoch

"safe" way by reducing the number of times a failure state is visited. We ran 10 training sessions and collected the number of times the failure state was visited and plotted the average value in 5.7. The value for the simple DRL is 27.9 times while for the ss module is 19.8 visits, that's a 29% decrease.

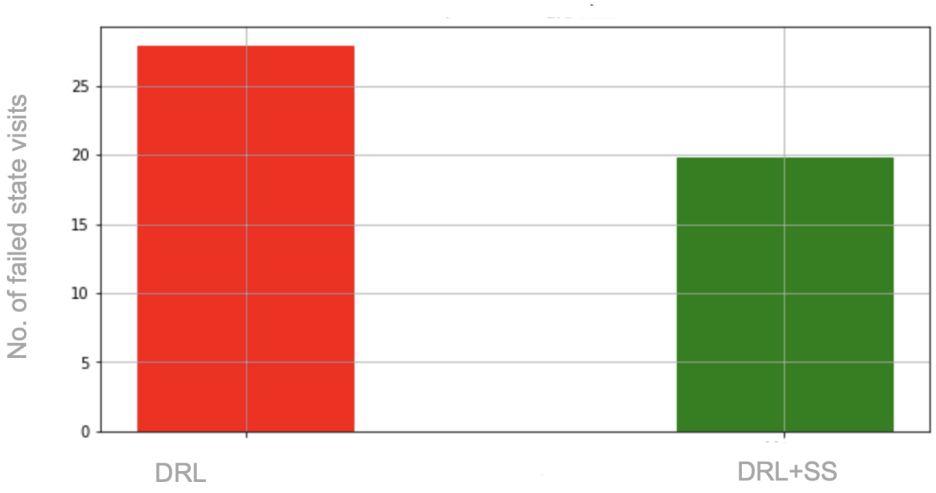


Figure 5.7: Failure state visits per algorithm

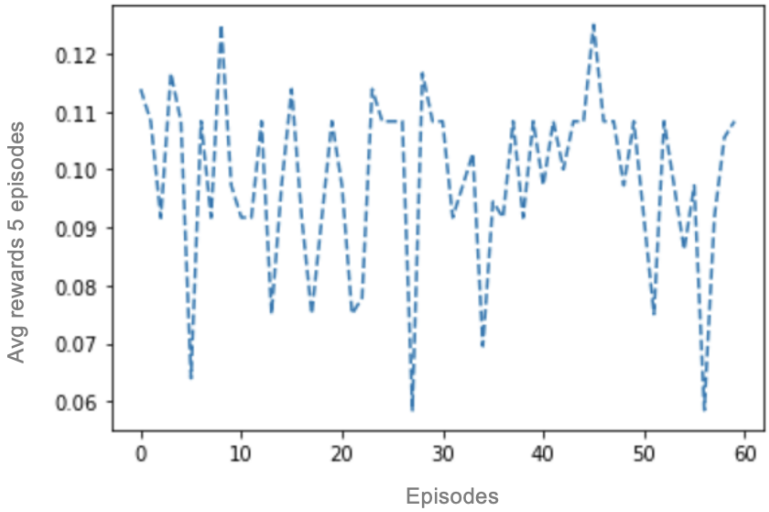


Figure 5.8: DDPG reward evolution

## 5.3. DDPG Results

We are adding this section just for reference since the DDPG agent was not successfully able to converge in our scenario. The main reasons may be related to the very small state and space for this problem and the fact that the action space is naturally discrete, we were forcing it to be continuous in order to deal with the scalability issues of DQN.

Given that the agent did not converge to any reward the result analysis in terms of DML training completion time is not meaningful. However the reward evolution, critic loss and actor loss are shown respectively in figures 5.8, 5.9, 5.10.

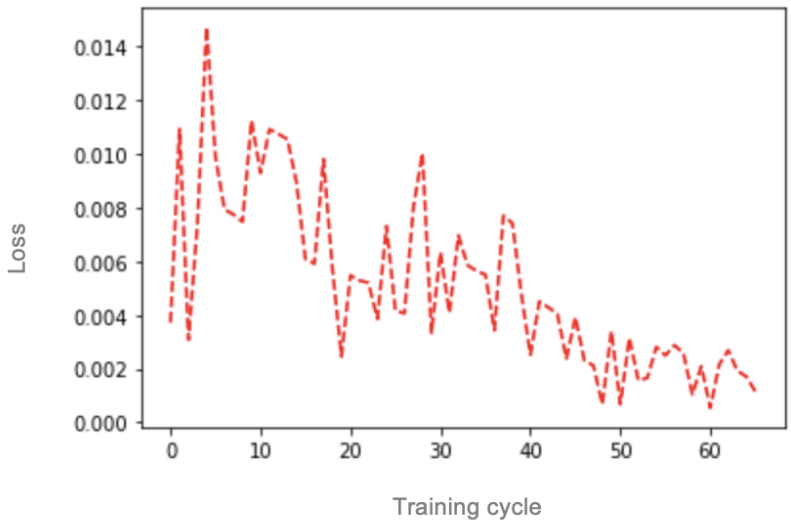


Figure 5.9: DDPG critic loss per training epoch

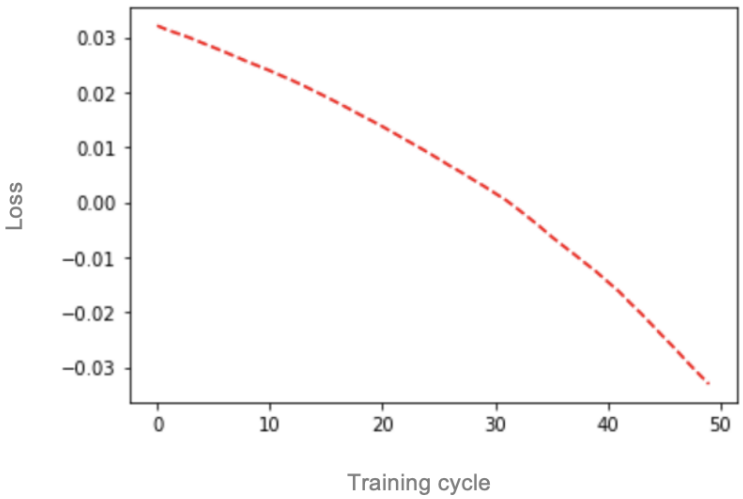


Figure 5.10: DDPG actor loss per training epoch

# 6 | Conclusions and Further Developments

For this thesis we developed a DRL-based optical reconfiguration algorithm to improve the training time of a DML algorithm running over 4 nodes in an experimental testbed in presence of network congestion. We were able to demonstrate a 5x times training time improvement by generating a new topology via optical switch reconfiguration and proper routing to separate completely the DML from the congesting traffic flow. We were also able to improve the performance of the DRL agent using a self-supervised technique leading to faster convergence and 29% less visits to the failure state. The result is very relevant since it provides a proof of concept that DML workloads can have their training time improved thanks to optical reconfiguration and that a self-supervised reversibility aware method can improve the performance of a regular DQN agent and reduce the number of visits to the failure state. The main limitations of our works are related to scalability both in terms of number of nodes and applications running in the testbed. A future improvement could be to deploy other applications on the testbed (Hadoop, media streaming etc...) to check the performance variations, and to increase the state space by simulating our algorithm over an actual data center network to deal with the possible scalability issues that may arise from using DQN. In fact, DQN works by providing a score to each available action and greedily choosing the best one. Such approach may result to be unsuitable when the action space is very very large (millions of actions), which would be a common scenario in a real world data center. In addition to the above, to investigate how other agents can perform in this scenario. A possibility could be the double DQN(DDQN) [15] algorithm which, by getting rid of the max operation in the training of DQN, may improve the scalability and performance of the agent in more complex scenarios.

# Bibliography

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[2] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[3] X. Chen, R. Proietti, M. Fariborz, C.-Y. Liu, and S. B. Yoo. Machine-learning-aided cognitive reconfiguration for flexible-bandwidth hpc and data center networks. *Journal of Optical Communications and Networking*, 13(6):C10–C20, 2021.

[4] W. O. Cifuentes. Testbed demonstration of optical reconfiguration by make before break approach in cloud computing networks. 2022.

[5] Y. Cui, S. Xiao, X. Wang, Z. Yang, S. Yan, C. Zhu, X.-Y. Li, and N. Ge. Diamond: Nesting the data center network with wireless rings in 3-d space. *IEEE/ACM Transactions On Networking*, 26(1):145–160, 2017.

[6] M. Dzida, M. Zagozdzon, M. Pioro, and A. Tomaszewski. Optimization of the shortest-path routing with equal-cost multi-path load balancing. In *2006 International Conference on Transparent Optical Networks*, volume 3, pages 9–12. IEEE, 2006.

[7] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[8] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 339–350, 2010.

[9] M. Ghobadi. Emerging optical interconnects for ai systems. In *2022 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2022.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[11] N. Grinsztajn, J. Ferret, O. Pietquin, M. Geist, et al. There is no turning back: A self-supervised approach for reversibility-aware reinforcement learning. *Advances in Neural Information Processing Systems*, 34:1898–1911, 2021.

[12] X. Guo, X. Xue, F. Yan, B. Pan, G. Exarchakos, and N. Calabretta. Experimental assessment of traffic prediction assisted data center network reconfiguration method. In *2021 European Conference on Optical Communication (ECOC)*, pages 1–4. IEEE, 2021.

[13] X. Guo, F. Yan, X. Xue, B. Pan, G. Exarchakos, and N. Calabretta. Qos-aware data center network reconfiguration method based on deep reinforcement learning. *Journal of Optical Communications and Networking*, 13(5):94–107, 2021.

[14] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 319–330, 2014.

[15] H. Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[16] inMon. Sflow-rt home page. URL `https://sflow-rt.com/index.php`.

[17] Iperf. Iperf home page. URL `https://iperf.fr/`.

[18] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 281–294, 2017.

[19] A. Kaur, V. Singh, and S. S. Gill. The future of cloud computing: opportunities, challenges and research trends. In *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC) I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC), 2018 2nd International Conference on*, pages 213–219. IEEE, 2018.

[20] M. Khani, M. Ghobadi, M. Alizadeh, Z. Zhu, M. Glick, K. Bergman, A. Vahdat, B. Klenk, and E. Ebrahimi. Sip-ml: high-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 657–675, 2021.

[21] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[22] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[23] KVM. Kvm home page. URL `https://www.linux-kvm.org/page/Main_Page`.

[24] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[25] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[26] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, et al. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[27] M. Masdari, S. ValiKardan, Z. Shahi, and S. I. Azar. Towards workflow scheduling in cloud computing: a comprehensive analysis. *Journal of Network and Computer Applications*, 66:64–82, 2016.

[28] D. W. McAllister, T. R. Mitchell, and L. R. Beach. The contingency model for the selection of decision strategies: An empirical test of the effects of significance, accountability, and reversibility. *Organizational behavior and human performance*, 24(2):228–244, 1979.

[29] O. Michel, J. Sonchack, G. Cusack, M. Nazari, E. Keller, and J. M. Smith. Software packet-level network analytics at cloud scale. *IEEE transactions on network and service management*, 18(1):597–610, 2021.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[31] F. A. Moghaddam, P. Lago, and P. Grosso. Energy-efficient networking solutions in cloud-based environments: A systematic literature review. *ACM Computing Surveys (CSUR)*, 47(4):1–32, 2015.

[32] U. of Toronto. Cifar10 and cifar100 datset. URL `https://www.cs.toronto.edu/~kriz/cifar.html`.

[33] OpenAi. Gym: Cartpole. URL `https://www.gymlibrary.dev/environments/classic_control/cart_pole/`.

[34] OpenSSH. Openssh main page. URL `https://www.openssh.com/`.

[35] K. Phemius and M. Bouet. Monitoring latency with openflow. In *Proceedings of the 9th international conference on network and service management (CNSM 2013)*, pages 122–125. IEEE, 2013.

[36] PicOS. Configuration guide. URL `https://docs.pica8.com/`.

[37] D. Rafique and L. Velasco. Machine learning for network automation: Overview, architecture, and applications [invited tutorial]. *Journal of Optical Communications and Networking*, 10(10):D126–D143, 2018.

[38] Ryu. Ryu ofctl rest documentation. URL `https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html`.

[39] Y. Shen, M. H. Hattink, P. Samadi, Q. Cheng, Z. Hu, A. Gazman, and K. Bergman. Software-defined networking control plane for seamless integration of multiple silicon photonic switches in datacom networks. *Optics express*, 26(8):10914–10929, 2018.

[40] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[41] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2015.

[42] M. Y. Teh, Y.-H. Hung, G. Michelogiannakis, S. Yan, M. Glick, J. Shalf, and K. Bergman. Tago: Rethinking routing design in high performance reconfigurable networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[43] M. Y. Teh, Z. Wu, and K. Bergman. Flexspander: Augmenting expander networks in high-performance systems with optical bandwidth steering. *Journal of Optical Communications and Networking*, 12(4):B44–B54, 2020.

[44] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 205–219, 2016.

[45] A. Venkatraman. Global census shows datacentre power demand grew 63% in 2012. *ComputerWeekly. com*, 8, 2012.

[46] A. Verma and N. Bhardwaj. A review on routing information protocol (rip) and open shortest path first (ospf) routing protocol. *International Journal of Future Generation Communication and Networking*, 9(4):161–170, 2016.

[47] VMware. What is data center networking. URL `https://www.vmware.com/topics/glossary/content/data-center-networking.html`.

[48] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 327–338, 2010.

[49] M. Wang, Y. Cui, S. Xiao, X. Wang, D. Yang, K. Chen, and J. Zhu. Neural network meets dcn: Traffic-driven topology adaptation with deep learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–25, 2018.

[50] K. Wen, P. Samadi, S. Rumley, C. P. Chen, Y. Shen, M. Bahadori, K. Bergman, and J. Wilke. Flexfly: Enabling a reconfigurable dragonfly through silicon photonics. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 166–177. IEEE, 2016.

[51] C. Yu, J. Lan, Z. Guo, and Y. Hu. Drom: Optimizing the routing in software-defined networks with deep reinforcement learning. *IEEE Access*, 6:64533–64539, 2018.

# List of Figures

# List of Tables

# Acronyms

| Acronym | Description |
|---|---|
| OCS | Optical Circuit Switch |
| DRL | Deep Reinforcement Learning |
| RL | Reinforcement Learning |
| DML | Distributed Machine Learning |
| DQN | Deep Q Network |
| DDPG | Deep Deterministic Policy Gradient |
| SS | Self-Supervised |
| AWS | Amazon Web Services |
| IT | Information Technology |
| SDN | Software Defined Networking |
| VM | Virtual Machine |
| EPS | Electronic Packet Switch |
| ML | Machine Learning |
| TD learning | Temporal Difference learning |
| ToR | Top of the Rack |
| SiP | Silicon Photonics |
| MC | Monte Carlo |
| DP | Dynamic Programming |
| ILP | Integer Linear Programming |
| MEMS | Micoroelectromechanical System |

# Acknowledgements

I would like to thank Prof. Massimo Tornatore who helped me out during my Master's and gave me the possibility to pursue our thesis at UC Davis. A special thanks goes to Dr. Sandeep Kumar Singh who followed me closely during our work in Davis and taught me how to make it through my first research experience. I would also like to thank Prof. Roberto Proietti and Prof. S.J. Ben Yoo for hosting me in their lab and providing me with funding for my time in California.

On a personal level I would like to thank my family for supporting me throughout my entire journey in both the bachelor's and the master's. I could not have reached this goal without your support.

A special thanks goes to U.U.

To conclude I would like to thank my university colleagues with whom we have shared ups and downs and long days and nights working on a multitude of projects. Thanks to your friendship and positive attitude we were able to keep myself strong and focused throughout my studies.