NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

AFSHIN NAKHOST LOTFI

Bachelor's Degree in Electrical and Computer Engineering

AUTONOMOUS TRAFFIC ENGINEERING USING DEEP REINFORCEMENT LEARNING

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
⟨September⟩, ⟨2022⟩

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

AUTONOMOUS TRAFFIC ENGINEERING USING DEEP REINFORCEMENT LEARNING

AFSHIN NAKHOST LOTFI

Bachelor's Degree in Electrical and Computer Engineering

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

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
⟨September⟩, ⟨2022⟩

Autonomous Traffic Engineering Using Deep Reinforcement Learning

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.4) [1].

Acknowledgements

The conclusion of this thesis is thanks to all the people that helped and supported me throughout this journey.

First and foremost, I would like to thank my family, especially my mother, who supported me through the tough times. This would not have been possible without you.

I would also like to thank one of my friends, Pedro Oliveira, who helped me start the thesis when I had no idea how to and who was always there to help with me when I had questions. I hope to be able to return the favor one day.

This thesis is dedicated to all of you.

Abstract

The evolution of communication technologies in the past few decades, has led to a huge increase in the complexity and the overall size of telecommunication networks. This phenomenon has increased the need for innovation in the field of Traffic Engineering (TE), as the already existing solutions are not flexible enough to adapt to these changes. With the appearance of *5G* technologies, the urgency to revolutionize the field is higher than ever and the softwarization and virtualization of the infrastructure bring new possibilities for TE optimization, namely the possible use of Artificial Intelligence (AI) based methods for Traffic Management.

The recent advances in AI have provided model-free optimization methods with algorithms like Deep Reinforcement Learning (DRL) that can be used to optimize traffic distributions in complex and hard to model Network scenarios.

This thesis aims to provide a DRL-based solution for TE where an agent is capable of making routing decisions based on the current state of the network, with the goal of balancing the load between the network paths. A DRL agent is developed and trained in two different scenarios where the traffic that already exists in the network is generated randomly or according to a systematic pattern. A simulation environment was developed to train and evaluate the DRL agent.

Keywords: Traffic Engineering, Deep Reinforcement Learning, Load Balancing

Resumo

A evolução das tecnologias de comunicação nas útlimas décadas, tem dado origem a um grande aumento na complexidade e no tamanho das redes de telecomunicações. Este fenómeno tem aumentado a necessidade de inovação na área de *Traffic Engineering* (TE), visto que as soluções já existentes não são flexíveis o suficiente para se adaptarem a estas mudanças. Com a aproximação das tecnologias *5G*, a urgência para revolucionar a área está cada vez maior e a *softwarização* e a virtualização das infraestruturas trazem novas possibilidades para otimizações de TE, nomeadamente o possível uso de métodos baseados em Inteligência Artificial (IA) para gerir o tráfego da rede.

Os avanços recentes de IA têm criado métodos de otimização independentes de modelos (*model-free*), como Deep Reinforcement Learning (DRL) que pode ser usado para otimizar a distribuição de tráfego em cenários de redes complexas e difíceis de modelar.

Esta dissertação tem como objetivo implementar uma solução à base de DRL, em que é desenhado um agente que é capaz de tomar decisões de encaminhamento, com o objetivo de gerir o tráfego pelos caminhos da rede. Um agente DRL é treinado em dois cenários diferentes onde o tráfego já existente na rede é gerado aleatóriamente ou de acordo com um padrão pré-definido. Foi criado um ambiente para treinar e para avaliar o agente DRL.

Palavras-chave: Traffic Engineering, Deep Reinforcement Learning, Load Balancing

Contents

List of Figures

Acronyms

API	Application Programming Interface
ARP	Address Resolution Protocol
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q Network
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
E2E	End-to-End
ECMP	Equal Cost Multi-Path
GPU	Graphics Processing Unit
HPR	Hot Potato Routing
JSON	JavaScript Object Notation
LB	Load Balance
LL	Least Loaded
MAC	Media Access Control
MDP	Markov Decision Process
ML	Machine Learning
MRTE	Multi-Region Traffic Engineering
MSE	Mean Squared Error
NN	Neural Network
NUM	Network Utility Maximization
OSPF	Open Shortest Path First

QoS Quality of Service

ReLU Rectified Linear Unit
RL Reinforcement Learning

SDN Software Defined Networking
SP Shortest Path
SPF Shortest Path First

TCP Transmission Control Protocol
TE Traffic Engineering
TIDE Traffic Information Distributor and Editor
TRPO Trust Region Policy Optimization

UDP User Datagram Protocol

Chapter 1

Introduction

In the past few decades, the rapid evolution of communication technologies has contributed to a significant change in today's society. Communicating over long distances is no longer an issue for most people, as there are more than enough ways of contacting someone. However, the ever-growing number of communication devices and especially, mobile devices, has led to networks of such complexity that it is making them harder than ever to manage. As more and more people get their hands on these devices, the expectation for better services also grows larger. Minimum delays and high availability are two basic examples of requirements for modern networks.

As these networks keep growing larger and keep getting more complex and dynamic, the need for optimizing how data is forwarded and how the network's resources are managed becomes even greater. Traffic Engineering (TE) is a method used to optimize the overall performance of the network by optimizing data forwarding and defining routing rules [2].

The TE technologies used in traditional networks are mostly *IP*-based TE and *MPLS*-based TE [3], which didn't allow full control over the network and didn't offer that much flexibility. The emergence of Software Defined Networking (SDN) has had a significant impact on the field of TE, allowing it to be much more dynamic and easier to implement. In SDN, the control plane of the network is separated from the forwarding plane, breaking the vertical integration of networks. The main characteristics of SDN, compared to traditional networks, are as follows [3]:

- Important network information, such as the network topology, changes in the network state and network application requirements (e.g., QoS, Security) are all stored centrally in the control plane.
- The ability to be programmed to fulfill the network requirements.

- Being open-sourced, meaning it can run on any type of equipment.

This architecture makes it easier to measure network parameters and manage them, creating new possibilities for implementing TE solutions. Typical TE solutions, such as forwarding traffic via the shortest path (e.g., OSPF) or distributing the traffic between multiple paths, do not consider the current network utilization nor the traffic load [4]. They also have greedy approaches and make commitments based on the current state of the network which makes them vulnerable to changes that might occur to the structure of the network, like a link failure for example [5]. This serves as a motivation for designing new algorithms than can cope with the increasing dynamics of modern networks.

Although some better solutions have been developed, they all have flaws that make them sub-optimal for modern networks. Being model-based is an example of a major flaw because it is near impossible to model the environment, user demands and their dynamics in such networks accurately [6]. Network Utility Maximization (NUM) [2, 6] is a model-based solution that formulates multi-hop TE problems, as constrained maximization problems of the utility function. However, this solution may have the following problems:

- Important factors, such as link usage and the network demand, are considered as inputs but in practical terms, they are hard to estimate.
- The End-to-End (E2E) delay cannot be explicitly included in the utility function since an accurate mathematical model of the network parameters is needed.
- Network dynamics are not taken properly into account, leading it to be satisfactory for specific situations but not for dynamic networks.

On the other hand, there has been an increasing effort to leverage Machine Learning (ML), more specifically Deep Reinforcement Learning (DRL), in TE. DRL is the combination of Reinforcement Learning (RL) and Deep Learning (DL), as shown in figure 1.1, and it is especially good at solving control tasks in which an agent has to deal with complex data. In this type of tasks, the agent learns how to act in order to achieve a certain goal [7]. TE problems can be formulated as such, so a DRL agent can be implemented in the control plane of SDN controllers to optimize network performance, while being prepared for changes to the network or to the user demands.

1.1 Objectives

The objective of this thesis is to study the already existing solutions to the TE problem as well as designing a new solution, based on DRL, that is capable of minimizing the consumed bandwidth in a network. This is done by developing and training a DRL agent in two different scenarios, where the traffic that already exists in the network is

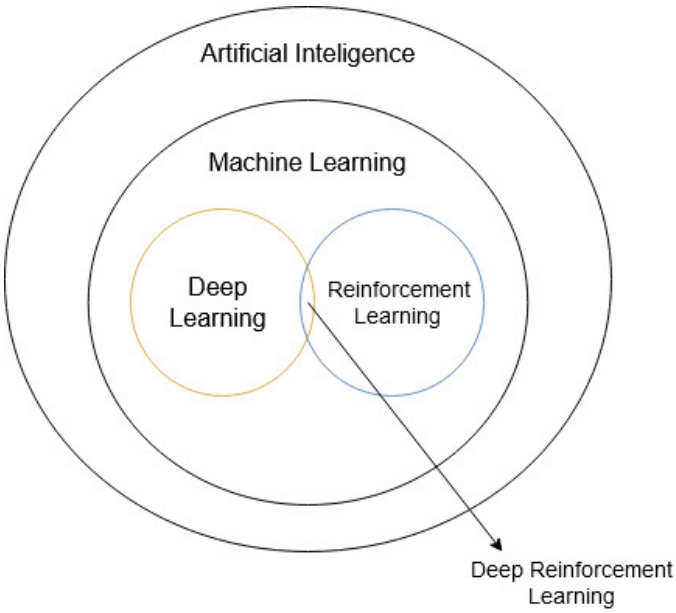

Figure 1.1: The relationship between Artificial Intelligence, Machine Learning, Deep Learning, Reinforcement Learning and Deep Reinforcement Learning.

either generated randomly or according to a specific pattern, and afterwards, creating an environment where the agent can be tested to evaluate its performance.

1.2 Structure

This document is divided into five chapters, the introduction, the state of the art, the proposed model, results and conclusions. In the first chapter, a short description of the problem and the motivation for this thesis is presented. In the second chapter, key concepts about Deep Reinforcement Learning and researches made on the subject are reviewed, with the overall objective of displaying existing solutions at this point in time. In the third chapter, a formal definition of the problem is provided followed by the description of the proposed solution and how it is implemented and what tools were used in the process. The fourth chapter describes how the solution is trained and tested in two different scenarios. The last chapter contains the conclusions that can be taken from this project and what future works can be done after this thesis.

Chapter 2

State of the Art

2.1 Reinforcement Learning

Reinforcement Learning is a subfield of Machine Learning and its application is in solving control tasks. In control tasks, like the name suggests, the agent has to learn how to control its environment and what actions it has to take to achieve its goal. It has its roots in human psychology and it represents a common learning method used especially in educating children. Unlike many other subfields of ML, in RL the agent is not told what to do and instead it learns which actions to take by exploring. The agent tends to take actions that it perceives as being more rewarding, in other words, the actions that result in a desired state are "reinforced positively"and the actions that do the opposite are "reinforced negatively".

The fundamental RL model is made up of an agent, an environment, a state-space, an action-space and rewards. The agent has the central role in this model and it processes observations and decides which actions to take. The environment is the potentially dynamic conditions, where the agent operates. It provides the input data for the sensory inputs of the agent. For example, when implementing RL to play a game, the game itself is the environment. The state-space is the set of all states the agent can be in and in general, states are snapshots of the environment and they are the input data that are given to the agent. They are static representations of certain moments in the potentially dynamic and changing environment. An action-space is the set of all actions the agent can take, with each action changing the environment and producing rewards. Typically a "reward"is a positive quantity but in RL the reward is a numeric quantity that needs to be optimized, so it can be either positive or negative [7].

These components work together in a loop in which the agent makes an observation, receiving data from the environment in the form of a state. The data is processed by

the agent that subsequently makes a decision and takes an action that influences the environment. Next, a reward signal is sent to the agent and the environment enters a new state. Figure 2.1 is a visual representation of how the RL model works in its simplest form.

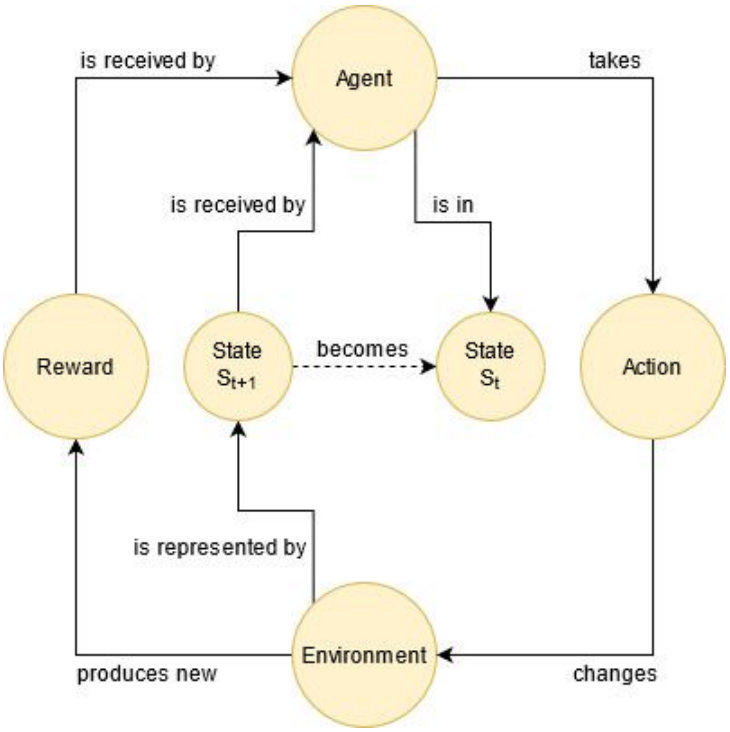

Figure 2.1: Fundamental DRL model. Adapted from [7].

A series of states, actions and rewards with a starting and finishing point (terminal state) is called an episode. A game of chess can be considered an episode and it starts when the game starts, and it reaches the terminal state when either someone wins or the game ends in a draw.

In RL, the agent can either have a complete model of its environment or have none at all, making it a model-based algorithm or model-free algorithm, respectively. A model can be anything from a Neural Network (NN) to a Bayesian graphical model [7]. Knowing the complete model of an environment makes it easy to predict its behavior but however, this is not the case in almost none of real world problems. Model-based algorithms are used in deterministic environments [8]. Model-free RL algorithms learn by trial and error just like human beings when an accurate model of a given environment is not available. These algorithm are generally easier to train because building precise model of complex environments is a really hard task [9].

In order to achieve optimal results, there has to be a balance between exploring different actions and exploiting already discovered strategies. At the beginning, it is better for the agent to explore so it does not get stuck in local maximums because the overall objective is to reach the absolute maximum. However, after finding some good strategies, the agent may opt to exploit these strategies to try and maximize its rewards. Finding a

balance between exploring and exploiting is essential if an agent wants to maximize its rewards.

2.1.1 The Markov Property

Another important notion about RL is the Markov Property. This is a property of any task in which, at any given state, the agent has enough information to make decisions that maximize the future rewards, as shown in the example in figure 2.2. Any task that has this property is called a Markov Decision Process (MDP) and it is essential to model problems as a MDP in the RL domain. Also every action that the agent takes, can affect not only the immediate reward, but every other reward from that point forward. The way the agent learns and how the chosen actions influence future rewards and future actions are the defining characteristics of Reinforcement Learning [9].

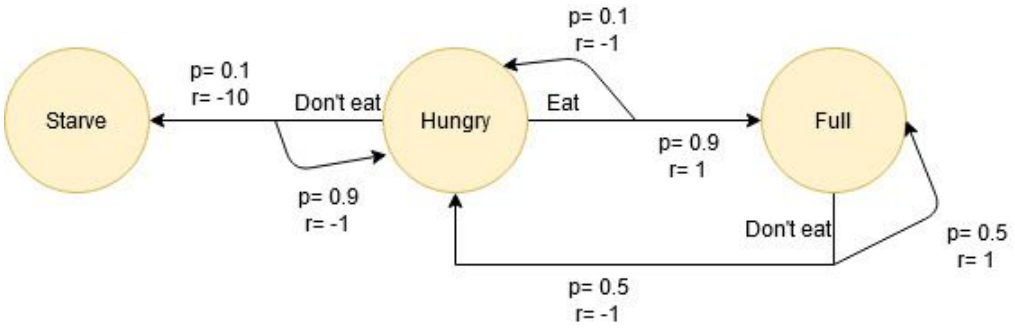

Figure 2.2: An example of a Markov Decision Process. Adapted from [10]

The example that was shown is that of a model-based algorithm, since we have the transition probabilities. This does not happen in real life as it is almost impossible to model the system of a person being hungry, being full or starving to death with accurate transition probabilities, because there are a lot of factors that have to be considered. In some cases of model-free algorithms, the agent needs to create a model of the environment and make it as close as possible to the real one [7].

2.1.2 Policy functions vs. Value functions

In any given state, the agent has to take actions with the overall objective of maximizing the rewards. The strategy that the agent adopts in order to achieve this is referred to as the policy (π) [7]. In mathematical terms, the policy is a function that maps a state to a probability distribution over the set of the actions that can be taken in that state, as shown in expression 2.1.

$$\pi, s \rightarrow P(A|s), s \in S \tag{2.1}$$

In the expression 2.1, π is the policy, s is a state from the state-space S and P(A|s) is the probability distribution over the set of actions. An optimal policy, referred to as π^*, is basically the policy that leads to the most expected rewards.

An important concept to mention is that learning algorithms can either be on-policy or off-policy. The former is a type of algorithm that learns a policy while using it at the same time to train itself. The latter is an algorithm that does not require a policy to learn, meaning that any policy can be used in the learning process and different policies only affect the efficiency of the algorithm [7]. An advantage of off-policy methods is the ability to learn from data that is gathered, either by another entity or by the same agent at a previous time [8].

Training a RL agent to take the best actions to maximize its rewards can be done either directly or indirectly. In the indirect approach (state-value functions), there is an emphasis on the most valuable states and the agent is taught to take actions that lead to those states. On the other hand, in the direct approach (action-value functions), the agent is taught which actions are the most desired ones, given the state it is in [7]. The idea of value functions arises from the indirect method. Value functions, usually referred to as state-value functions, map either a state or a state-action pair to the expected rewards that are associated with being in that state and taking actions according to a policy. The mathematical notation for this function is presented in equation 2.2.

$$V_\pi : s \rightarrow E(R|s, \pi) \tag{2.2}$$

State-value is a term used to describe the value of a given state or in other words, the expected sum of rewards when starting from a state and following a certain policy. High state-value for a given state means that we can expect high rewards if we start from that state. However, action-value functions describe the expected sum of rewards when taking an action. Action-value is used to express the value that taking a certain action can have for the agent.

2.1.3 Q-learning

Q-learning is a popular off-policy learning method that uses a Q-function, which maps a state-action pair to a Q value (weighted average of rewards), given a certain policy, as shown in figure 2.3.

Q-functions are a combination of state-value functions and action-value functions. In this learning method, the Q value (expected reward) associated with a state-action pair is stored and it is occasionally compared with the rewards that were actually received to update the parameters of the algorithm. In other words, we learn how to estimate the expected reward in each state. Equation 2.3 demonstrates how the Q values are updated.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.3}$$

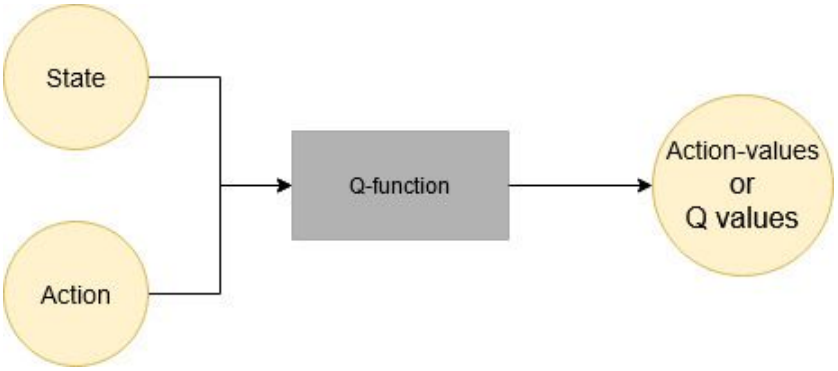

Figure 2.3: Q-function. Adapted from [7].

In the update rule that was mentioned, the first $Q(S_t, A_t)$ is the updated Q value with the second one being the current one, α is the learning rate (step size), R_{t+1} is the received reward, γ is the discount factor and $maxQ(S_{t+1}, a)$ is the maximum Q value for all actions. The step size and the discount factor are called *hyper-parameters* that only affect how the algorithm learns [7]. The discount factor determines how much the agent discounts future rewards. Its value is between 0 and 1. A discount factor of 0 implies that the agent will only consider immediate rewards and if it is equal to 1, the agent would have to consider rewards infinitely in the future, which is impossible in practical terms. Choosing the discount factor is a trade-off between short-term and long-term results. The learning rate defines how much the algorithm updates after each step, so a small learning rate indicates that the updating process has to be gradual and a large learning rate does the opposite. This is really important because in some cases, a large learning rate can stop the algorithm from converging because of the big updates that are made to the parameters at every step. A small learning rate can also be harmful in some cases as it can make the learning process a really lengthy one.

2.1.4 Epsilon Greedy

One strategy is to focus more on exploiting known strategies and choosing actions with high expected rewards, rather than choosing random actions to explore new opportunities. ϵ-greedy (Epsilon-greedy) is a more balanced variant of this greedy approach. Instead of always choosing the action that gives out the largest reward, we choose a random action with a probability of ϵ. The rest of the time, the action with the highest expected reward will be chosen [7].

There is a question of how the known experiences have to be stored, given that storing every action-reward pair is practically impossible as the training continues. There is a workaround that involves storing the actions with the mean of all received rewards for taking that action. Upon taking an action and receiving a reward, we can update the average value of rewards for that action, following equation 2.4, in which μ is the average value of rewards already received, k is the number of observations already made and x is

the new reward.

$$\mu_{new} = \frac{k.\mu_{old} + x}{k+1} \tag{2.4}$$

2.1.5 Softmax Selection Policy

Another method is the Softmax selection policy, that replaces the random selection of actions during exploration, like in ϵ-greedy, with choosing actions according to a probability distribution based on their action-value ranking. This way we can explore whilst having less possibility of choosing actions with the least values [7]. The probability distribution is calculated as shown below in equation 2.5.

$$P(A) = \frac{e^{\frac{Q_k(A)}{\tau}}}{\sum_{i=1}^{n} e^{\frac{Q_k(i)}{\tau}}} \tag{2.5}$$

The input of this function will be the action-value array that is initially set to an array of zeros. The numerator outputs an array of the same length as the input array and it is composed by the exponentiation of the action-value divided by a parameter called the temperature (τ). The denominator outputs a single number that is the sum of the exponentiation of every action-value divided by the temperature parameter. The probability distribution will favor actions that yield larger rewards. The temperature scales the probability distribution. A large value creates similar probabilities and a small value creates probabilities with an exaggerated discrepancy between them.

One disadvantage of this method is how hard it is to manually assign a value to the τ parameter. Although ϵ-greedy also has a parameter to be set, it is a more intuitive process and for this case, there needs to be a lot of trial and error in order to find an appropriate value.

2.2 Deep Learning

Dealing with complex data, classifying them and recognizing patterns in big sets of data are difficult tasks. Deep Learning (DL) is a well-known subfield of Machine Learning that is exceptionally good at doing all of those, as they can represent complex data as a set of simpler components. In this learning method, Deep Neural Network (DNN) are utilized and they essentially resemble the human nervous system and the neurons in the brain, as shown in figure 2.4 [11]. Neural networks can be leveraged in various problems, such as pattern recognition, classification, computer vision, natural language processing, regression, predictive analysis, etc.

The "deep" in DL refers to the multi-layered architecture of these neural networks that are organized by input layer, hidden and output layers. The hidden layers are composed of all the layers between the input and the output. This organization is for the sake of abstraction. The hidden layers having multiple layers makes it a lot more efficient than

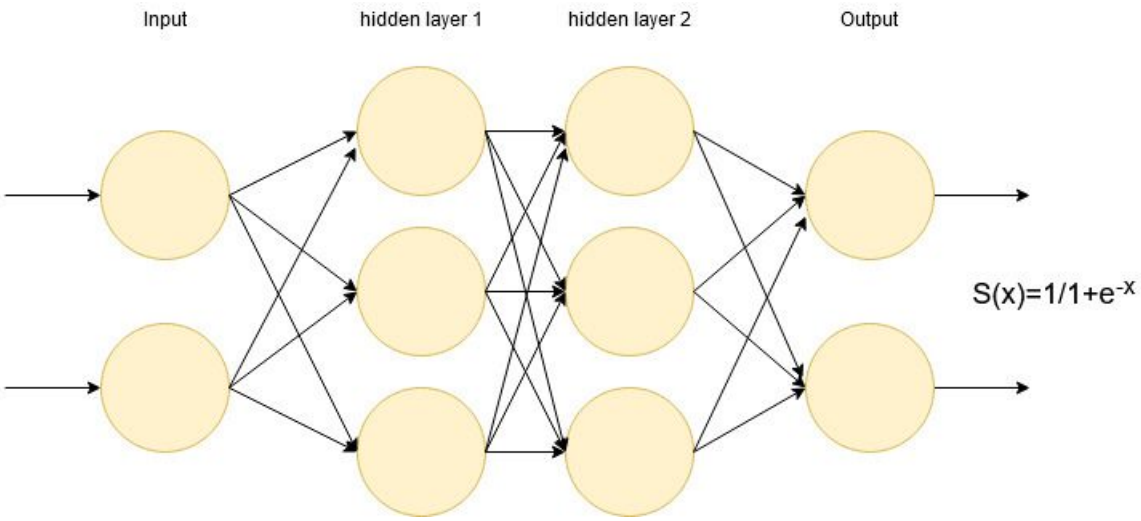

Figure 2.4: An example of a Deep Neural Network.

a hidden layer with only one layer. Also, this deeper model allows it to deal with more complex data and to successfully map non-linear functions [11]. Each layer can deal with a certain feature of the input data, like in the case of face recognition, one layer deals with the edges, one with the nose, another with the ears and so on.

In each layer, there are a set of nodes, or neurons, that are each connected to nodes in adjacent layers with every connection having a certain weight. The first one is the bias. The inputs at every node are multiplied by their weight and then summed together. The result is then passed through an activation function, which in most cases is the Sigmoid Function. Other popular activation functions like Tan Hyperbolic or Rectified Linear Unit (ReLU), all have a mathematically favorable derivative that makes them popular. This characteristic makes it easier to compute partial derivatives of the error delta with respect to individual weights [11]. Figure 2.5 demonstrates how the output of each neuron is obtained.

Initially the weights are random and they get more accurate and more similar to optimal values as the training process goes on. A set of data, called the training set, is put through the neural network and the weights are updated according to the output and the desired results. After the training, a test set goes through the neural network in order to test the overall performance of the network. However, manually updating them is not a good idea because it is seriously inefficient and in tasks like pattern recognition, more automatic learning and reduced manually designed heuristics works better [11]. Each passage of the training set through the NN and the backpropagation (updating the parameters of the NN) afterwards is called an *epoch*. Typically, training a NN takes thousands of epochs and it is a very time-consuming task. Having said that, the rapid evolution of Graphics Processing Unit (GPU) in the last two decades has helped to reduce the training time dramatically.

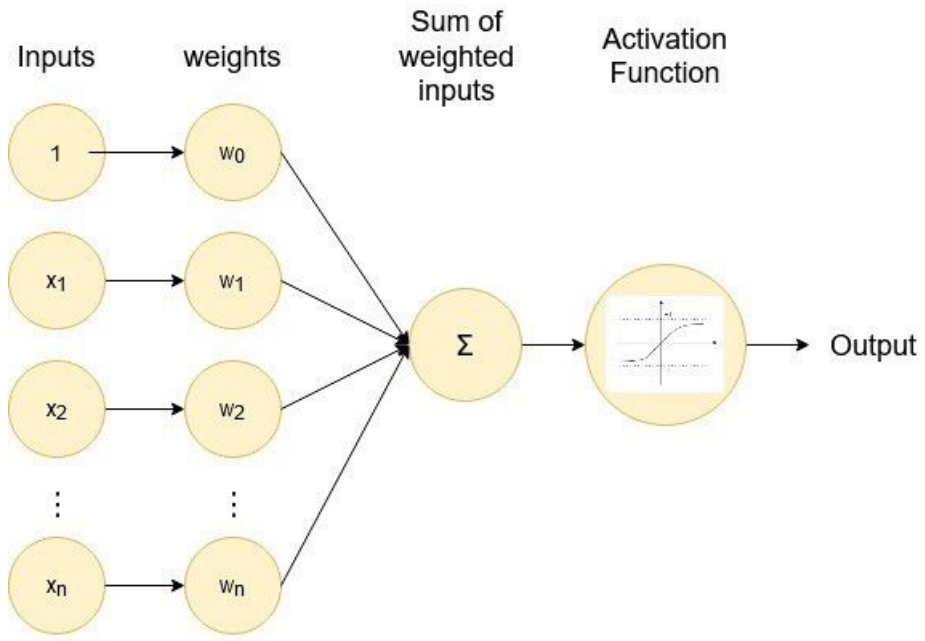

Figure 2.5: Calculation of the output of a neuron, in a Neural Network. Adapted from [7].

2.2.1 Supervised vs. Unsupervised

DL can be divided into two main categories, Supervised and Unsupervised Learning. In Supervised Learning, the agent learns by being given a certain input together with the correct output with the purpose of generalizing its outputs so that they act correctly when the test set is put through the model. One big downside of this learning method is the data collection that has to be done beforehand. Also this model lacks the dynamism of RL in the sense that, it does not adapt well to unknown environments. On the other hand, in Unsupervised Learning, there is no supervision and instead, the model has to learn hidden structures in the data without being given the correct outputs. At first glance, Unsupervised Learning seems similar to RL but they have different purposes. Although finding hidden structure in a data set is advantageous, it is not the only thing that matters when solving a RL problem. The goal is to maximize the rewards the agent receives [9].

2.2.2 Backpropagation

Feedforward Neural Networks are a type of NN in which the data only flows from the input to the output and does not form any loops. Training these networks can be done by using a method called Backpropagation. In this method, the gradient of the loss function is calculated, one layer at a time, starting from the last layer. This is to avoid redundant calculations of intermediate terms. Calculating the gradient is not the challenge while training these networks. The real challenge is numerically evaluating them, which backpropagation does perfectly using a simple procedure [11, 12].

However, backpropagation only calculates the gradient, so another algorithm, such as Gradient Decent or Stochastic Gradient Decent has to be utilized to minimize the gradient and do the actual learning.

2.2.3 Gradient Descent and Stochastic Gradient Decent

Gradient Decent is the algorithm that does the learning in Neural Networks. After the gradient has been calculated, it is up to the Gradient Decent to minimize it and it does so by adopting Newton's algorithm for finding a 2D function's zero. The process consist of descending along a specific path in a multi-dimensional weight space until the loss function can no longer decrease. One of the problems associated with Gradient Descent is getting stuck in a local minimum that can give the impression that the loss function is optimized when it really is not [11].

In the training process, all of the training set is put through the network before adjusting the weights in order to reduce the cost function. This is repeated until reaching a point in which the cost function does not decrease anymore.

A variant of the Gradient Decent, is the Stochastic Gradient Decent that does basically the same thing but more frequently. After going through a certain amount of data from the training set, it will update the weights and by doing so it can reduce the training time and converge to a local minimum faster than Gradient Decent.

2.3 Deep Reinforcement Learning

Deep Reinforcement Learning is the combination of Deep Learning and Reinforcement Learning. It combines the best characteristics of both, resulting in a powerful learning method that can be used to solve control tasks with complex state space. These tasks have a really dynamic nature and solving them requires an algorithm that adapts well to changes. Reinforcement Learning is an ideal framework for solving control tasks but for some specific cases, we can choose to incorporate another algorithm, like DL in this case.

Typically in RL, the agent stores every experience and then analyzes the results, but this quickly becomes impossible when we move on from basic problems like playing Tic-Tac-Toe, to a more complex problem like playing Chess [7]. Tic-Tac-Toe has 255,168 distinct games that exist but in comparison, chess has around 10^{120} valid distinct games. To put this into perspective, if someone at the beginning of the Universe, decided to register all possible chess games by playing a new game every second, they still would not even have gone through half of them. That is where DRL comes into play. Thanks to DRL, it is possible to solve dynamic control tasks where the state space is simply too large for a simple RL algorithm [7].

2.3.1 Deep Q Networks

One way DL can be used alongside RL is when a DNN is used to approximate the Q values in Q-learning, resulting in an algorithm called Deep Q Network (DQN). It was first proposed by *DeepMind* and it started to draw attention when it started to surpass human beings at playing classic Atari games without having any previous knowledge of those games [13].

The first step of this algorithm is initializing the Q values by feeding the initial state, s_t, into the Q-network. This will create a distribution of the expected values over all actions, in that initial state and following a policy π, as discussed in chapter 2.1.3. After that, an action, a_t, is taken based on the policy. After reaching the next state and receiving the reward for that action, s_{t+1} and r_{t+1} respectively, it is time to update the parameters of the DNN to make better predictions. Training methods in which, the parameters are updated after each state transition are known as online training methods. The parameters are updated (backpropagation) with the purpose of minimizing a loss function that will guide the algorithm. When the episode does not end after taking the action a_t, the loss function is given by equation 2.6.

$$L = (Q(s_t, a_t) - (r_{t+1} + \gamma \max Q(S_{t+1}, a)))^2 \tag{2.6}$$

The maximum Q value available across all actions is $maxQ(S_{t+1}, a)$ and it is multiplied by the Discount Factor which was discussed earlier in chapter 2.1.3. However, if the episode ends after taking that action, the loss function that has to be minimized turns into equation 2.7.

$$L = (Q(s_t, a_t) - r_{t+1})^2 \tag{2.7}$$

Finally, the Q values can be updated using algorithms like Stochastic Gradient Descent. All of these steps, except the initial prediction, have to be repeated until the algorithm converges [8].

Despite being a revolutionary approach, there are a lot of problems that are associated with DQN, like training instabilities and also a phenomenon known as *Catastrophic Forgetting*. Moreover, this algorithm requires an external policy to choose the actions, like the epsilon greedy. This only complicates the training process and it gives an edge to algorithms that do not require a separate policy.

2.3.1.1 Catastrophic Forgetting and Experience Replay

A big issue associated with gradient descent-based online training is the Catastrophic Forgetting [7]. Catastrophic Forgetting occurs when the algorithm is presented with two seemingly similar states, but when it takes the same action in both, it gets totally different results from them. Upon encountering a state similar to a known state, the algorithm might decide to take the most valuable action from that known state, which can lead to

horrible results in the new state. This can be very confusing for the algorithm and might prevent it from learning correctly from the situation.

This issue can be addressed by using a method called the experience replay [7]. Using experience replay, implicates using a buffer called the experience replay memory. Each experience is a tuple of the current state, the next state, the action and the reward. The size of the buffer is up to its designer. It is going to be filled with the experiences the algorithm gathers over time and when it is full, a subset of those experiences is chosen and used for backpropagation. This is called *batch* training and it is more beneficial than backpropagating after every epoch. The size of the batch is optional too so it can be adjusted to the needs of the designer. When the buffer is full, the oldest experience is replaced by the new one. A visual representation of the experience replay scheme is shown in figure 2.6.

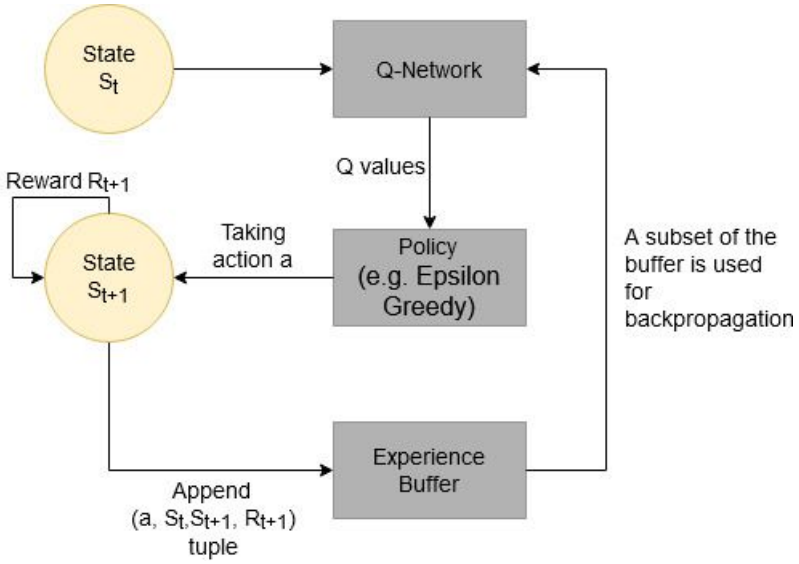

Figure 2.6: A Deep Q Network with Experience Replay. Adapted from [7].

2.3.1.2 Target Network

Another issue that arises from the online training method is the instability in the learning process that can occur, if the rewards are sparse. For example, if the agent receives high rewards for taking a certain action in some state, it will consider the action to be a good one. However, if the same action in another state returns a sub-optimal reward, it can lead the agent to think the opposite of what it believed to be true. This instability can be prevented if a target network is used. Using a target network is like making a backup of your data, before starting a risky update on your computer.

The basic idea of this method is to make a copy of the Q-network, which will be the target network. This way, we can train the original network without worrying about the changes that are made. After a number of iterations, the target network is updated with

the parameters of the original network and the whole process starts over again. Figure 2.7 demonstrates how this works.

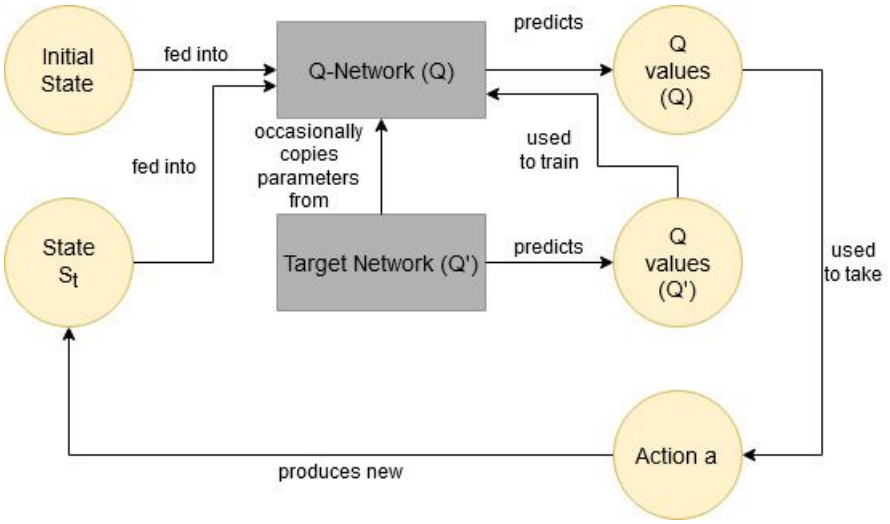

Figure 2.7: A Deep Q Network with a Target Network. Adapted from [7].

2.3.1.3 Double Q-learning

A problem that is associated with using DQN is the overestimation of Q-values, because the values that are used to take actions are the same ones that are used to evaluate those actions. This problem can be solved by employing the Double Q learning method, in which the selection and the evaluation are separated. In this method, two value functions are used but they are not both updated with every experience that the agent gains. Instead, each experience is assigned to only one of them at random. This leads to two different sets of weights where one of them is used to determine the greedy policy and the other determines its value [14].

2.3.1.4 Dueling Architecture

Traditional DQNs learn the value of every action at a given state and this can be inefficient in some cases. There are states where the actions that the agent takes, do not affect the environment in a significant way. To address this problem, a new architecture for these networks was proposed in [15], by the name of Dueling Architecture. It is composed of two streams, one for the state-value (value stream) and for the advantage of each action (advantage stream). The two streams are combined together at the end to estimate the state-action value function.

Instead of learning every action-value each state, this architecture can learn which states are important and actually make decisions when it has to. In [15], a simple example of the Atari game *Enduro* is given to demonstrate the idea. In this racing game, if the car has a clear path in front of it, there is really no need to take any particular action. However, if an obstacle appears, the car has to avoid colliding with it. For this to work,

the value stream has to pay attention to the road and the advantage stream has to pay attention whenever there is an obstacle in its way.

2.3.2 Policy Gradient Methods

An alternative to DQN is a class of algorithms called Policy Gradient. Unlike DQN and other algorithms that approximate the value function, PG approximates the policy function, $\pi(s)$. Instead of being given the values of the actions and then having a policy to tell us which action to take, the algorithm chooses an action by sampling a probability distribution. The DNN is going to be the policy function. Actions that yield higher rewards have a higher probability, so they are more likely to be selected [7].

Policy Gradient methods have some advantages over DQN. First of all, there is no need to implement a policy on top of the existing algorithm as the DNN will be the policy function. It also does not require all the extra methods that were used, in order to prevent instabilities when training the DQN.

2.3.2.1 Stochastic Policy Gradient vs Deterministic Policy Gradient

There is a variant of the Policy Gradient method called the Stochastic Policy Gradient method. In this method, the output is a vector made up of actions that represents the probability distribution over all actions. Then, an action is selected at random. This can lead to different decisions being made in the same state, which is great if we want to explore more. In fact, if the probability distribution is uniform initially, there can be enough exploration to find the optimal action for a given state. This means that the probability distribution converges to a distribution where every action has a probability of 0, except the optimal action that has a probability of 1. This is called a *degenerate* distribution [7].

Another way to approach this is by choosing the same action every time, which is known as the Deterministic Policy Gradient method. Basically it is as if we had a degenerate distribution from the beginning. This method does not allow exploration like the Stochastic Policy Gradient, which is a limiting factor in its application.

2.3.2.2 REINFORCE

Another variant of the policy gradient method is the REINFORCE algorithm. This is an episodic algorithm so the rewards are only analyzed at the end of the episode. At the beginning of the episode, the NN which is the policy function for the algorithm, outputs a vector of probabilities and an action is chosen by sampling the vector. At the end of the episode, when all the rewards are gathered, the algorithm can backpropagate to update the policy function (NN) based on the obtained rewards.

One important aspect of this algorithm is deciding how much influence every action has on the outcome. This is really important because the parameters are only updated

after ending each episode, so it would be foolish to assume that every action had the same effect as the other ones during the episode. This is known as the problem of *credit assignment*. For example in chess, checkmating the opponent is the ultimate objective so it would make sense if it was the most important action. This distinction can be made using the discount factor, which was discussed in chapter 2.1.3. The discount factor tends to be smaller at the beginning of the episode and it increases during the episode until reaching 1 at the terminal state [7].

The policy network is π and θ is the vector that parameterizes the network. The parameters that represent the policy network are the weights of the NN. The policy function is denoted π_θ, and the probability of selecting an action a according to the parameters of the policy network is $\pi(a|\theta)$. Knowing that the sum of all action probabilities is equal to 1, it is easy to conclude that $\pi(a|\theta)$ values will have to be between 0 and 1. This can be a headache, especially when using computers with limited numerical precision, because some of these probabilities can be either too small(0) or too big (1) and this can cause issues when used in an optimization problem. To overcome this issue, $\pi(a|\theta)$ is replaced by $-log\pi(a|\theta)$ in the loss function [4].

The algorithm takes the following steps:

1. Initialize the NN with random parameters and create a random policy.
2. Use the policy to train the algorithm during an episode.
3. Calculate the product of the discount factor and the collected rewards ($\gamma^t * G_t$), also known as the discounted reward for each action.
4. Calculate the product of discounted rewards and the probability of each action at every step.
5. Use this to backpropagate (minimizing the loss function).

The loss function, at each time step is defined by equation 2.8.

$$L = -\gamma_t * G_t * log\pi(a|\theta) \tag{2.8}$$

This algorithm is an example of a Policy Gradient algorithm but it is a really simple one. It has a serious scaling issue because when the action-space is large, reinforcing good actions get harder because there are a lot of possibilities.

2.3.3 Actor-Critic Method

In the previous two sub-chapters, two different classes of DRL algorithms were introduced. Deep Q Networks are a good solution for problems with a discrete action-space but they require a separate policy, like the epsilon greedy, to take actions. Also, the original version of the algorithm has some flaws that have to be addressed by using an experience replay buffer and a target network. DQNs are considered value functions

because they receive a state and return the Q values for taking each action. On the other hand, Policy Gradient method, another type of DRL algorithms, are policy functions that output a probability distribution over all actions. This eliminates the need to have a separate policy. However, this method is episodic which limits its application as there are problems that are not episodic.

These methods are both beneficial in some way but both have flaws that limit their application to certain situations. There is another DRL method called the Actor-critic method, that is a combination of the two and it aims to improve two significant aspects by doing so [7]. The first one is going from the episodic nature of PG to an online training, like in DQN. This way, the updates are more frequent and the efficiency of the sampling is increased. The second improvement is lowering the variance of the received rewards by using the value function.

As the name suggests, actor-critic method consists of two entities, one being the actor (the entity that takes actions) and the other being the critic (the entity that evaluates those actions). In this scheme, the policy is the actor and it is responsible for taking the actions, and DQN is the critic. By doing this, we can update the parameters more frequently now that there is a value function to evaluate the value of all actions in any given state. There is also no need to use a separate policy as the policy function outputs a probability distribution and the action is chosen by sampling it.

2.4 Related Work

The field of DRL has been progressing a lot in the past few years and this has contributed to it being used in many different fields to solve control tasks. If the problems are formulated in a way that gives them the Markov Property, DRL can be leveraged to solve them in an elegant fashion. Traffic Engineering is one of the areas in which there has been an attempt to use DRL to address problems that cannot be addressed with the traditional solutions.

However, DRL is a large field and there are numerous algorithms that follow the guidelines of this learning method but each one has a specific application. There could be situations where an episodic algorithm is the best option compared to an online training method. We can have a model of the environment and only require the agent to predict what is going to happen or we may want to introduce it to an environment without a model and let it learn on its own. Selecting the appropriate algorithm is a problem of its own so it is really important to know what solutions already exist.

In this chapter, other related works will be presented with the purpose of offering a glimpse of what has been done to apply DRL to TE problems and what their conclusions were. They are divided into three categories, single-agent methods, multi-agent methods and SDN-based methods.

2.4.1 Single-Agent Methods

In [6], the authors propose creating a DRL-based control framework called DRL-TE, which is specific to TE problems like the name suggests. Inside this framework, they suggest two different techniques:

1. TE-aware exploration
2. Actor-critic-based prioritized experience replay

They consider a communication network with K end-to-end communication sessions, with the throughput and delay (x_k and z_k, respectively) of each of the K session being the two components of each state. The actions are the set of split ratios for the communication sessions and the reward is the total utility of all sessions, formally represented by equation 2.9.

$$r = \sum_{k=1}^{K} U(x_k, z_k) \tag{2.9}$$

In the equation 2.9, the reward that the agent receives is defined as being the sum of the utility function for each communication session, which is described by equation 2.10.

$$U(x_k, z_k) = U_{\alpha_1}(x_k) - \sigma U_{\alpha_2}(z_k) \tag{2.10}$$

The factor σ defines the relative importance of delay vs. throughput.

The algorithm they chose was the Deep Deterministic Policy Gradient (DDPG) that was first proposed by *DeepMind*, to tackle problems with a continuous action-space. Of course, algorithms like DQN were not used because the action-space in TE problems are continuous and DQN can only deal with problems that have a discrete action-space. However, they identified two issues that this algorithm has in this case. The first is the lack of exploration and the second is the algorithm ignoring the importance of transition samples in the replay buffer. The first could be addressed by using a random noise based method to ensure a certain level of exploration but that does not work well in TE problems.

To overcome these two issues, they used a TE-aware exploration technique that utilizes an already existing TE solution as the baseline during exploration, and also an actor-critic-based prioritized experience replay that indicates the importance of samples. The TE solutions that serve as the baseline for exploration can be anything from evenly distribution the traffic load to forwarding along the shortest path. The prioritized experience replay scheme is implemented by simply giving more priority to certain sample than the others.

The framework and the environment were implemented in ns-3 and the DNNs were implemented using *TensorFlow*. They ran the tests on two different topologies, NSF Network and ARPANET. Then they tested it again on a randomly generated network

topology, using BRITE. The constant K was set to 20, σ and α were set to 1, 3-shortest paths were chosen as candidate paths, link capacities were set to 100 Mbps and packet arrival at the source nodes (not the intermediate nodes) followed a Poisson distribution. They compared their solution to some other solutions such as Shortest Path, Load Balance, NUM and the basic DDPG algorithm. The metrics for this comparison were the end-to-end throughput, end-to-end delay and the network utility.

The final results indicate that DRL-TE, consistently outperforms every one of the other solutions, at every aspect. Even the end-to-end delay, which cannot be mathematically modeled with precision, was significantly reduced. The overall utility of the network was also superior for this method, in the two topologies that were used as well the random topology. The basic form of DDPG could not do nearly as good as DRL-TE and often got stuck in local minimums, showing the difference that the modifications really made. All in all, the proposed algorithm showed great results and adapted well to changes in the traffic load and network topology.

2.4.2 Multi-Agent Methods

In [2], a multi-agent reinforcement learning solution for distributed control is proposed, with the purpose of minimizing the End-to-End delay. Their solution is a framework consisting of modules and extensions that can be combined together to create a multi-agent reinforcement learning algorithm to be used in multi-hop networks. Each agent has three interleaving modules, a policy evaluation module, a policy improvement module and a policy execution module. The policy evaluation module estimates the action value functions, $q_i^{\pi_i}(s,a)$. The policy improvement uses these values to update the policy and optimize it for better results. Finally the policy execute module executes a behavior policy, b_i, which is the policy that generates the actual actions of the agent to generate new action-reward experiences for the next round of policy evaluation and improvement. In this work, a distributed actor-critic-executor architecture was chosen.

In this actor-critic-executor scheme, the critic (policy evaluation module) estimates action-value functions and based on these estimations, the actor (policy improvement module) improves the target policy which is what the algorithm wants to optimize. The executor (policy execution module) executes the actions according to the behavior policy, which can either be identical to the target network or similar in such a way that allows more exploration. If the learning is on-policy, then the behavior policy is the same as the target policy and if it the learning is off-policy, the behavior policy is slightly less greedy which allows exploration. This is demonstrated in figure 2.8.

In the experimental part, they include various methods such as, off-policy, on-policy, expected value estimation, double learning, softmax action selection and n-hop learning, and also shortest path routing algorithm and an off-policy greedy algorithm (Q-routing). The first round of simulations considered a non-stationary network, where the network loads vary. Network load is represented by λ and it is the average number of packets that

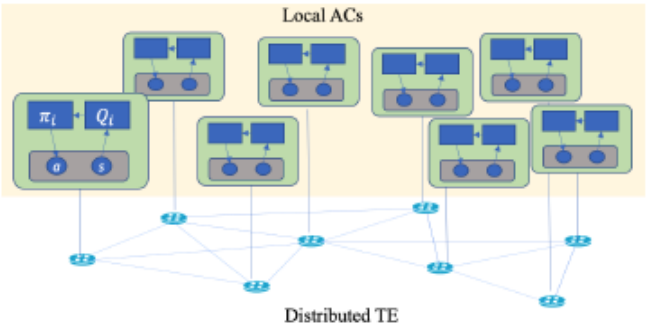

Figure 2.8: Distributed Actor-Critic-Executor. Taken from [2].

arrive at any time step. For the first round, this number went from 0.5 up to 4.5. In the second round, λ went from 3.5 to 4.4 to test the algorithms' adaptability to higher traffic load. It was supposed to represent a stationary network with high network load. The results of the first round of simulations indicate the following:

- The shortest path algorithm did not adapt well to the increase in network load.
- Off-policy algorithms perform better than on-policy algorithms when the network load is at a medium range. ($2 \leqslant \lambda \leqslant 3$).
- On-policy algorithms outperform off-policy algorithms when the network load is higher than 3.5.
- Expected value evaluation decreases the delay due to the smaller variance in the return estimation.
- In almost every case, double learning improves the performance of the algorithms because it reduces the maximization bias. It also improves the overall E2E delay. However, it does not perform well with low network loads because it requires a higher amount of experiences to learn efficiently.
- Softmax action selection helps to balance the network load and reduce packet delivery time by finding secondary paths through exploration.
- 2-hop learning did not prove to be beneficial
- A high learning rate leads to a better performance.

The second round of simulations led to the following conclusions:

- A low learning rate of 0.1 led to better delay performance but a longer learning time.
- A high learning rate of 0.9 reduced the learning time but led to results that were not optimized.

- Double learning displays the best results, with lower and higher learning rates. Double learning with a high learning rate is three times faster to converge than double learning with a low learning rate.

This work provides a thorough evaluation of different algorithms and offers the opportunity to combine different ones to create more elaborate algorithms that can be used in specific scenarios.

In [16], the authors approached the TE problem similarly to [2], justifying that a single agent algorithm is not scalable in practice and that it is preferable to divide the whole network in regions and control each region locally. Each region independently computes its routing decisions based on local observations but they all work towards optimizing a global TE objective. In this approach, there are two types of network traffic, one that circulates inside the region it originated from and the other is the network traffic that passes through various regions. With this in mind, they proposed two separate learning agents. The T-agent observes the status of the local network and controls the routing of terminal demands. The O-agent however, controls the routing of outgoing demands and it interacts with neighbor regions. Having two separate learning agent reduces the action-space. The training phase consists of both online and offline training. In the offline stage, the agent is trained in a simulated network similar to the real one. In the online stage, the agent has to make decisions and improve its abilities with almost no communications between regions required.

The state-space for the T-agent is described as being the status of the network which is expressed by the edge utilization. Edge utilization is determined by both traffic demand and routing decisions. It reflects both the traffic demand change and the effects of the actions that the agent takes. For example, if the edge utilization is zero it could indicate that the edge is broken down. The action-space is the set of routing decisions to distribute traffic onto multiple forwarding paths. To reduce the overhead of path maintenance and to further reduce the action-space, some of these forwarding paths are calculated beforehand. To reduce the action-space even further, they used the same set of forwarding paths and splitting ratios for the terminal demands between the same ingress and egress nodes. Also, considering the fact that larger loads of traffic are more likely to cause congestion, they consider different policies for small flows and large flows. The larger flows, also known as elephant flows are the agent's responsibility but the smaller flows, also known as mice flows, are routed according to a static routing protocol like Equal Cost Multi-Path (ECMP). The rewards are calculated through a reward function according to the current status of the network. To avoid misleading rewards, rewards are calculated based on the status of the region where the agent is located at.

The state-space for the O-agent is identical to that of the T-agent. The action-space is made up of splitting ratios and forwarding paths but for traffic that leaves the region. To reduce the action-space, the same set of forwarding paths and splitting ratios are used for outgoing demand coming from the same ingress node and going to the same neighboring

region, just like it was done for the T-agents. However, there is no distinction between mice flows and elephant flows as a small flow that passes through a large amount of links can be more important than a large flow that passes through a short distance. The rewards for O-agents reflect not only the status of the local region, but the status of the whole network. At each step, neighboring regions share their T-agent's rewards to allow the O-agents to calculate theirs.

The DRL algorithm used in this paper is the DDPG. The framework was implemented using TensorFlow. They used the Telstra topology, the topology from Google cloud and a randomly generated topology using BRITE. Their framework, Multi-Region Traffic Engineering (MRTE), was compared to Hot Potato Routing (HPR), ECMP and Trust Region Policy Optimization (TRPO).

Overall, MRTE proved to be the strongest in various tests that were done on all four algorithms. In the normalized congestion test, MRTE had the lowest congestion rate. In the normalized congestion test with a random link failure, MRTE and TRPO performed similarly in the first two topologies but TRPO performed really poorly in the last topology which also happened to be the largest network. This proves that using multiple agents, as opposed to using a single agent per region is a better solution. Another test which set out to discover the effects of having multiple regions came to the conclusion that the higher the number of regions, the better is the performance of the algorithms. Nevertheless, there is a limit to the number of regions. The actual limit is not stated in the paper but it was proven that in an extreme case where every node is a separate region, there is a lot more congestion than in any other case.

2.4.3 SDN-based Methods

In [4], the authors propose an algorithm that runs in the decision plane of a SDN architecture, as shown in figure 2.9. This solution, which is named SDN_DRLTE, allows a TE solution to be output by the algorithm. The solution is then converted in flow entries that are distributed to all the switches without having to program them one by one. This is the most appealing aspect of integrating DRL algorithms into a SDN.

Traditionally, a routing algorithm like OSPF is used in the decision plane to decide what actions need to be taken at each discrete timestamp. The information that these algorithm base their decision on, comes from the forwarding plane. It is typically the throughput or the delay in the communication channels of the network in function of the network demand. The decision plane, the control plane and the forwarding plane all work together in a closed loop [4]. The control plane upon receiving the actions from the decision plane, converts them into forwarding rules which are then sent to each switch by using open source programs like OpenFlow. The forwarding plane is the layer that is responsible for forwarding the data packets. It also sends performance reports to the decision plane, as it was mentioned before.

This algorithm aims to replace the traditional routing algorithms and optimize a SDN

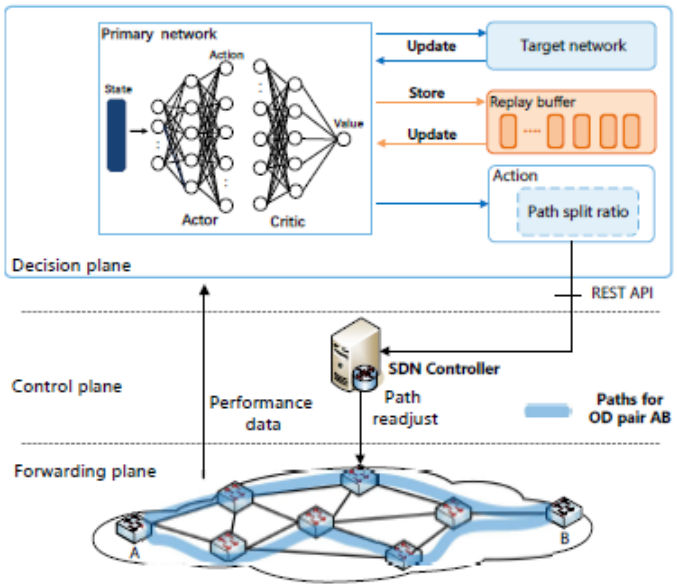

Figure 2.9: DRL integrated in a SDN architecture. Taken from [4].

traffic control. The traffic is split on a flow level and not on a packet level, because it can cause packet arrival out of its original order and service degradation. The authors consider K pairs of origins and destinations and for each pair with m flows with the same traffic demand for each pair. The flows can be forwarded on a collection of paths P_K. The ratio of the traffic of each pair to P_K is the output of the algorithm. The delay and the rate of transmission (d_k and x_k, respectively) are the two metrics that were used in this paper. They are the mean value of the delay and the transmission rate of each flow from the origin-destination pair, respectively. The performance of each pair is given by equation 2.11, where x_k is the transmission rate and d_k is the delay.

$$U(x_k, d_k) = log(x_k) - log(d_k) \tag{2.11}$$

In this method, the significance of these two metrics is considered to be equal. The overall objective is to maximize an objective function, which like in the last case, is the sum of the performance scores of all K pairs.

$$r_t = \sum_{i=1}^{K} U(x_i, d_i) \tag{2.12}$$

States are defined as a combination of the normalized values of the delay and the transmission rate. Using the normalized values can contribute to a faster convergence. Actions are the set of split ratios for each origin-destination pair. The sum of split ratios for each pair equals one. The reward, at any given time, is given by equation 2.12. In this work, the authors opted for the DDPG algorithm with TE-aware and prioritized replay. Mininet was used as the emulator of the forwarding plane and the controller was Ryu. The DNNs were implemented using Keras. The network topologies that were used for the

simulations were NSFNET and GEANT2. Furthermore, they considered 3-shortest paths as candidate paths, 50 traffic flows with the traffic load following a Poisson distribution. Just like in [6], they compared their results to the performance of Shortest Path (SP), Load Balance (LB) and DDPG_TE algorithms, in the same conditions.

The results were in favor of SDN_DRLTE. In terms of transmission rate, even though it suffered some inevitable reduction, it was at a slower rate than both SP and LB because SDN_DRLTE can learn to dynamically split the traffic flow according to the state of the network. The delay was much lower compared to the other three algorithms and the overall rewards that it received were higher, but this is normal because it's the objective of this algorithm to maximize the rewards. Overall, the algorithm performed much better than the others and adapted well to changes in the network load and the network topology.

In [5], the authors proposed a SDN-based algorithm that differs from the last one in that it uses a destination-based forwarding strategy to solve a TE problem at a packet level. In this strategy, instead of sending data via multiple paths, the data is forwarded via a single path and this path can be changed according to the network state. The algorithm is called RL-Routing.

In this work, new features for state representation are proposed. These features include link trust level, switch throughput rate and link-to-switch rate. The first feature makes it possible to classify the links based on their reliability. This allows the algorithm to identify links where there are a lot of collisions or packet-loss and avoid them if possible. The other two provide information on the status of the links and the switches. Congestion in the network can be avoided when this information exists. They deploy one agent per switch by configuring the network to have a one-to-many relationship. In other words, instead of connecting a source switch to a destination switch, they connect the source to all destination switches and vice-versa. The reward function is also adjustable for either upward or downward network throughput.

States are a set of ten features and they summarize a lot of information about the network which can lead to a better management of the network. The list of all features is presented below.

- $f_1 = \{c_{i,j} : \forall e_{i,j} \in \epsilon\}$ - link capacity rate set.
- $f_2 = \{x_{i,j} : \forall e_{i,j} \in \epsilon\}$ - link throughput rate set.
- $f_3 = \{delay_t(i,j) : \forall e_{i,j} \in \epsilon\}$ - link delay set.
- $f_4 = \{status_{i,j} : \forall e_{i,j} \in \epsilon\}$ - link status set, where the status is one if the link is up and zero if otherwise.
- $f_5 = \{T_{i,j} : \forall e_{i,j} \in \epsilon\}$ - link trust level set.
- f_6 - upward switch throughput rate.

- f_7 - downward switch throughput rate.
- $f_8 = \{cx_{i,j} : \forall e_{i,j} \in \epsilon\}$ – link-to-switch rate set.
- f_9 - 7-dimensional vector that indicates the day of the week.
- f_{10} - 4-dimensional vector that the time of day, each position corresponds to a quarter of the day (e.g. the 2nd position is 6AM-12AM).

In the expressions above, $e_{i,j}$ is a link from the set of all links, ϵ. These features allow the agent to have a very detailed idea of the network, even allowing it to predict certain aspects of the network based on the time of day.

The action-space is all of the paths that are available between a source switch and a destination switch, ordered by their total length with the first one being the shortest one. However, if the number of actions is very large, the complexity of the algorithm and the computation cost rise dramatically. The reward is the sum of delay and throughput with the same weight.

The algorithm that is used in this work is the Dueling Double Deep Q Network (Dueling DDQN) with prioritized memory experience. The chosen policy was the ϵ-greedy. Mininet was used to emulate the forwarding plane and Ryu was used as the SDN controller. The network topologies that were used to test the algorithm were Fat-tree, NSFNET and ARPANET. OSPF and Least Loaded (LL) were tested too in order to put the performance of the algorithm into perspective. The evaluation of the three algorithms was based on the following three metrics.

- Reward.
- File transmission time.
- Utilization rate, which is the ratio of the link bandwidth that is used for data transmission from a source to a destination, to the link's maximum bandwidth. It is calculated by the destination switch and the bigger its value, the faster is the data transmission.

The results indicate that RL-Routing cuts the file transmission time of OSPF and LL by more than half. Also, the average utilization rate of RL-Routing is higher than the other two algorithms. OSPF and LL are greedy algorithms that do not always make the best decisions so they change their course of action many times during a single file transfer, whereas RL-Routing rarely needs to change its course. The amount of times source switches have to retransmit data is also higher in OSPF and LL which increases the delay of the network significantly.

All in all, this work presented many fresh ideas and a very interesting approach to solving TE problems using DRL.

In [17], a scalable DRL method, based on the pinning control theory, called the Scale-DRL, is proposed. In this approach, some of the links are considered to be critical. The algorithm allocates weights to all of the links based on this information and then the traffic is forwarded according to a weighted shortest path algorithm. By doing this, the action-space shrinks and this can increase its scalability. ScaleDRL is a SDN-based algorithm, like the algorithms in [5] and [2]. The training consists of an online stage and an offline stage. The offline stage is when the critical link selection occurs. The online stage is when the algorithm assigns weights to the links in order to achieve optimized results.

The links that are considered to be critical are not actually the ones that are the most central. After determining the centrality of all of the links, they are sorted in a descending order and the middle k links are chosen as the critical links. The ratio of the critical links to the total number of links is known as the link selection ratio (α). The DRL framework that was used in this paper is a variant of the actor-critic method, called ACKTR. The states are defined as being the traffic distribution on each link, the actions are the paths that the algorithm chooses and the rewards are the average E2E delays.

The simulations were set up in OMNET++4.6 and the DNNs were implemented using Keras. The selection ratio was 0.3. The network topologies that were used were NSFNET, OS3E and two other generated by BRITE. The algorithms that were tested to serve as a comparison were the Shortest Path First (SPF), ECMP, Traffic Information Distributor and Editor (TIDE) and DRL-TE from the work in [6].

The results indicate that ScaleDRL suffers from less delay than the others, while TIDE and DRL-TE suffer from the Curse of Dimensionality, which is when the action-space becomes so large that the algorithm struggles to converge. A different test was carried out to test if the strategy of selecting the k middle links as the critical links is the most efficient. Between choosing those links randomly, choosing the top links, choosing the bottom links or selecting the k middle links, the last came out on top. The link selection ratio was also analyzed and the best ratio turned out to be the one that was used. A high link selection ratio performed well in a network with the least number of nodes but in the larger networks, it increased the E2E delay.

Chapter 3

Proposed Model

As discussed earlier in the first chapter, the goal of this thesis is to provide a solution to the TE problem, by leveraging DRL algorithms' ability to solve complex control tasks. This thesis proposes a SDN-based system that implements an agent for a designated switch that is in a one-to-many configuration with the other network switches. This agent assumes that the rest of the network switches are controlled by their own agents, so this work will only focus on developing a single agent. This approach is similar to the one used in [5], and it aims to provide a solution that takes advantage of DRL's ability to learn in complex environments to balance the load of the network in the most efficient manner possible.

The proposed model in this thesis consists of a customized environment, that is built around a telecommunications network, and a Deep Reinforcement Learning agent that engages with it over discrete time intervals, Δt, that are called *steps* in the context of this work. During each *step*, the agent observes the environment and receives a state, $s \in S_t$, that represents the state of the paths of the network. In order to make the network more realistic and to diversify the states that can be produced, traffic has to be generated from the other hosts. The state space, S_t, is the set of all available states the environment can be in at any given time interval, Δt. Based on its observation of the environment, the agent takes an action, $a \in A_t$. An action is the path that the agent chooses to install between the source and the destination of a specific flow, and the action space, A_t, is the set of all available actions the agent can choose from. A reward, R_{t+1}, is given to the agent as a feedback for its decision and a new state, s_{t+1}, is generated. The agent is developed with the intention of maximizing the expected future rewards, given by equation 3.1.

$$R = \sum_{t=0}^{\infty} \gamma R_{t+1} \tag{3.1}$$

The Discount Factor, $\gamma \in [0,1]$, was discussed earlier in the second chapter and it defines how much the agent cares about immediate and future rewards. As the action space is the set of all paths that connect the source host to the destination host, the goal of the agent can be defined more intuitively as finding the best set of paths that connects the source host to its destination host, given a certain state of link bandwidth in the network.

The interaction between the agent and the environment is ran in a loop, called an *episode*, and it only comes to an end when the agent encounters a terminal state. The agent is trained by interacting with the environment over a course of many episodes. Then it is tested in a specific testing scenario that evaluates its performance. This chapter focuses on defining the problem, how the problem is simulated and how the proposed model approaches it.

3.1 Problem Definition

In order to give a formal definition of the problem, a few notions have to be presented beforehand. First and foremost is the Markov Property that was discussed earlier in the second chapter. Problems of this nature are control tasks and they can be solved by DRL algorithms, only if they have this property. Having this property means that, at any given state, the agent has enough information to maximize its future rewards. This was taken into consideration during the problem definition, as not doing so could result in the agent not having access to crucial information and not being able to learn properly.

The formal definition of the network is the other important notion that is presented before the problem itself. The network can be represented by a *graph*, so that representing paths between the elements of the network becomes more intuitive. This way, the network of hosts, switches and links can be represented by a directed graph *G=(N, E)*, where N is the set of *Nodes* that correspond to the hosts, N_H, and switches, N_S, and E are the edges that connect these nodes. The designated switch, $N_{S_{src}}$, is connected to the source host, $N_{H_{src}}$, that generates the flows that have to be managed by the agent. The rest of the switches in the network are known as destination switches, $N_{S_{destinations}} \equiv N - N_H - N_{S_{src}}$. The destination hosts, $N_{H_{destinations}} \equiv N - N_S - N_{H_{src}}$, are all the hosts that the source host, $N_{H_s rc}$, communicates with during the training process. All the links, $e_{x,y}$, that connect any pair of nodes, x and y, are considered to be full duplex so $e_{y,x}$ is equivalent to $e_{x,y}$. A path, $p_{src,dst}$, between the source host and a destination host, can be defined as a set of nodes and edges that connects the two hosts. The nodes along a path between two hosts are switches and they cannot be repeated, as this would create loops in the paths. The bandwidth of any given path during a step, $bw_{p_{src,dst}}$, is equal to the bandwidth of the link with the lowest bandwidth along that path.

The problem can be formally defined as follows. Given a network graph, *G=(N, E)*, a source host, $N_{H_{src}}$, a destination host, $N_{H_{dst}} \in N_{H_{destinations}}$, find the path, $p_{src,dst}$, between the source-destination pair that best distributes the network load between the links of the network.

3.2 Environment

The environment is built around the network and it simulates the settings of the problem that was just introduced. The environment provides a network which is controlled by a SDN controller. Using the information and the statistics from this network, it is possible to produce all the possible states the agent can find itself in (state space), all the decisions it can make (action space) and a reward system that evaluates those decisions.

The *step* function of the environment defines what happens at every discrete time interval, Δt. During each episode, the agent receives the current state of the network, which is the bandwidth of every path in the network. It also receives a *flow request* containing a destination host and the desired bandwidth of the flow. The generated bandwidth of the requests follows an exponential distribution, but the destination is chosen randomly. Based on this information, the agent makes a decision that produces a reward, which indicates the quality of its decision, given the circumstances.

The state space, S_t, consists of all the states the agent can find itself in and because in this work, a state contains the bandwidth of a path between a source-destination pair, the state space contains the bandwidths of every path between the source host and all destination hosts. Equation 3.2 represents the tensor that represents the dimensions of a state, s.

$$s = (N, k, 1) \tag{3.2}$$

N represents the number of destination hosts, $N_{destinations}$, k represents the number of paths between a source-destination pair and and the last index of the tensor represents the bandwidth which is the evaluation metric that is used in this work. The dimensions of the state space can be calculated by multiplying N, k and 1. Given that the only metric used in the state representation is the bandwidth of paths, the last index has a dimension of 1. If other metrics were to be used, the state tensor can be easily modified by increasing the dimension of the last index.

The only difference between the state space tensor and the state tensor is that the first one will have a continuous range of values, between the minimum and maximum value of possible bandwidths, and the second one has a discrete value that represents the bandwidth of a given path in that step.

The action space, A_t, is the set of all paths that exist between the source and any given destination. After the agent receives the flow request with the destination of the flow, it can choose an action, a_x, from the action space with size, x, that represents a path between the designated source and the given destination. The action space can be represented by equation 3.3.

$$A_t(Destination) = (a_1, a_2, \ldots, a_x) \tag{3.3}$$

In networks where there are a lot of paths between hosts, the action space can get excessively large which greatly impacts the computation time of training the model. To avoid this problem, the number of available paths between any given pair of hosts, x in equation 3.3, is always limited to 5. This way, only the five shortest paths are presented to the agent as viable actions. This reduces computation time and although it can slightly affect the learning process, the trade off is worth it. These paths are ordered in ascending order of their lengths, so the first one is the shortest path between the two hosts and the last one represents the longest one of all the paths. The chosen action is sent to the controller where it is installed for usage during that *step*.

After a *step* is finished, a reward is given to the agent. In this work, the bandwidth of the paths are the main focus so the rewards reflect how well the chosen action affects this specific metric. The reward system is based on the remaining bandwidth of the chosen path between the source and the destination. Equation 3.4 represents the reward system, where *BW percentage* is the percentage of available bandwidth in the chosen path, at the end of a *step*.

$$Reward = \begin{cases} +5 & \text{if } BW percentage \geqslant 90 \\ +3 & \text{if } 90 > BW percentage \geqslant 75 \\ +1 & \text{if } 75 > BW percentage \geqslant 60 \\ -2 & \text{if } 60 > BW percentage \geqslant 50 \\ -10 & \text{if } 50 > BW percentage \end{cases} \tag{3.4}$$

Using this system, it is possible to motivate the agent to choose actions that do not overload paths. This is done by giving positive rewards to actions that lead to a *BW percentage* of 60% or higher. A negative reward is given when the *BW percentage* falls below 60%. If any action results in a path's bandwidth to fall below the 50% threshhold, it is heavily punished by a reward of -10. The values of the rewards were chosen empirically.

3.2.1 Implementation

The tools that were used to simulate and control the network are *Mininet* and the *Ryu* SDN controller, respectively. The environment itself, which is built around this network, is created by using the open-source Python library, *OpenAI Gym*. The following two subsections provide the necessary explanation of how they were implemented.

3.2.1.1 Mininet and Ryu

Mininet is utilized to create virtual networks with hosts, switches, and links that are very similar to their physical counterparts [18]. It is possible to build networks with *Open vSwitch* switches and *TCLinks* with specific link bandwidth, delay, and packet loss. The accuracy with which *Mininet* simulates real-life networks and the availability of a Python API, makes it the perfect software for SDN projects such as this thesis. Another perk of

using *Mininet* is the *Iperf* integration. This allows the network to simulate TCP or UDP traffic, with any given bandwidth usage over a desired period. It also generates log files in JSON format, so it is possible to evaluate the performance of the network [19]. This is what will be used throughout this work to simulate traffic between hosts.

To create the underlying network, the topology is imported from a text file containing the hosts, switches and links and all the necessary information about the links, such as bandwidth, loss and delay and then it is built using *Mininet*. It is necessary for the hosts to know each other's MAC addresses if they wish to communicate with each other, so after building the network, it is necessary to run the Address Resolution Protocol (ARP) to build ARP tables. ARP is a communication protocol that is typically used to find the link layer address (e.g. MAC address) associated with an internet layer address (e.g. IPv4) [20].

The controller, *Ryu*, is a component-based, open-source, SDN controller that was chosen for this work because it has a Python API and it supports popular communication protocols such as *OpenFlow* [21]. The SDN controller communicates with the switches using the *OpenFlow* protocol, more specifically, by creating *OpenFlow* events that are responded to by *event handlers*. There are 3 *event handlers* that are implemented in this controller, *switch_features_handler*, *state_change_handler* and *port_stats_reply_handler*. They are responsible for installing the table-miss flow entry, detecting existing switches and for getting statistics from the ports, respectively. The table-miss flow entry indicates what happens to packets that do not match with any flow entries in the switches. It installs rules with priority 0 and wildcards all match fields [22]. Several statistics from the ports, such as the utilization rate of the links, are collected every three second by a thread that sends *Port_stats_request* to the switches and when they respond with *Port_stats_reply*, the handler is called and the controller receives the statistics. There is also another thread that is responsible for maintaining the *active paths* which are basically the paths that are installed between each host pair, during each *step*. *Active paths* are sent to the controller by the environment every time the agent chooses a path that is different from the one that is already installed. Connecting the controller to the network, allows the installation and removal of flow rules, as well as centralized network monitoring. Unfortunately, due to an unknown problem with the output statistics gathered by the RYU controller, they could not be used to their full potential in this work. The problem was related to how the utilization rate was calculated which resulted in the statistics being useless for training and testing. Instead, they were only used for diagnosing problems with the network during the development of the project.

To provide a more intuitive view of the network, *NetworkX* was used to create a graph corresponding to the network's topology, where each host or switch is a node and each link is an edge [23]. The graph is created from the same topology file that was used to create the *Mininet* network. The paths between every pair of nodes are created by using the built-in *all_simple_paths* function from *NetworkX*. They are then converted to tuples of (*s*, *in_port*, *out_port*) where *in_port* is the port from which the packet arrives at

the switch *s*, and *out_port* is the port from which the packet must leave the switch. The controller translates the path into a set of flow rules that are installed in the switches. These rules dictate that if at any switch along the path, a packet with the MAC addresses of the source and the destination hosts arrives from the *in_port*, it should be forwarded to the next switch in path or the destination host, through the *out_port*. The controller receives the tuples of (*s*, *in_port*, *out_port*) as paths and installs the corresponding routing rules in the switches that are along those paths.

3.2.1.2 OpenAI Gym

As mentioned earlier, the environment is built using *OpenAI Gym* [24]. The Python API provided by *OpenAI Gym* serves as an interface between learning algorithms and environments and it is used to develop Reinforcement Learning algorithms. The default functions that are required by *OpenAI Gym* are the following : *init*, *reset*, *render*, and *step*. They are in charge of initializing the environment, resetting it, creating visual effects if necessary and defining what happens during each *step*, respectively. The render function is ignored as the environment does not need to have any visual output.

The *init* function initializes the parameters of the environment as well as the observation space which is a *Box* object that contains the lower and higher bound for what the state could look like. The observation space is equivalent to the state space. The *reset* function resets the environment to its initial state at the beginning of every episode. The *step* function is the most important part of the environment as it defines what happens in the environment during each step. After the agent chooses an action, it calls the *step* function, where the chosen path is sent to the controller. The controller then installs the path, if it is different than the one that is already installed. After this, *Iperf* is called to simulate traffic with the bandwidth that was requested at the beginning of the *step*, with a thread waiting for it to end to calculate the reward. *Iperf* denotes the source as the client and the destination as the server and creates a JSON log file for each one of them. Upon receiving the log from the client, the reward of that step is calculated based on the performance of the network.

In order to make the network behave like a real-life network would, traffic is generated in the network in each *step*. Before the state is given to the agent at the beginning of each *step*, a number of flows, which depends on the number of destinations, is created according to equation 3.5.

$$\mathrm{N}umberOfFlows = \begin{cases} (N_{H_{destinations}} - 1)/2 & \text{if number of } N_{H_{destinations}} \text{ is odd} \\ N_{H_{destinations}}/2 & \text{if number of } N_{H_{destinations}} \text{ is even} \end{cases} \tag{3.5}$$

These flows can originate from any destination host, $N_{H_{destinations}}$, and can have any destination, including the source host, $N_{H_{src}}$. The source host cannot generate flows at this stage so that it is only used to generate the flows that are the focus of this work.

The source and the destination of these flows, the path between them and the desired bandwidth of the flow are either generated randomly or they are generated according to a predefined traffic pattern. The first set of generated flows in an episode are given by the *reset* function, that provides the agent with the initial state of the network. During the rest of the episode, the *step* function is responsible for creating these flows. The flowchart corresponding to the *step* function is shown in figure 3.1.

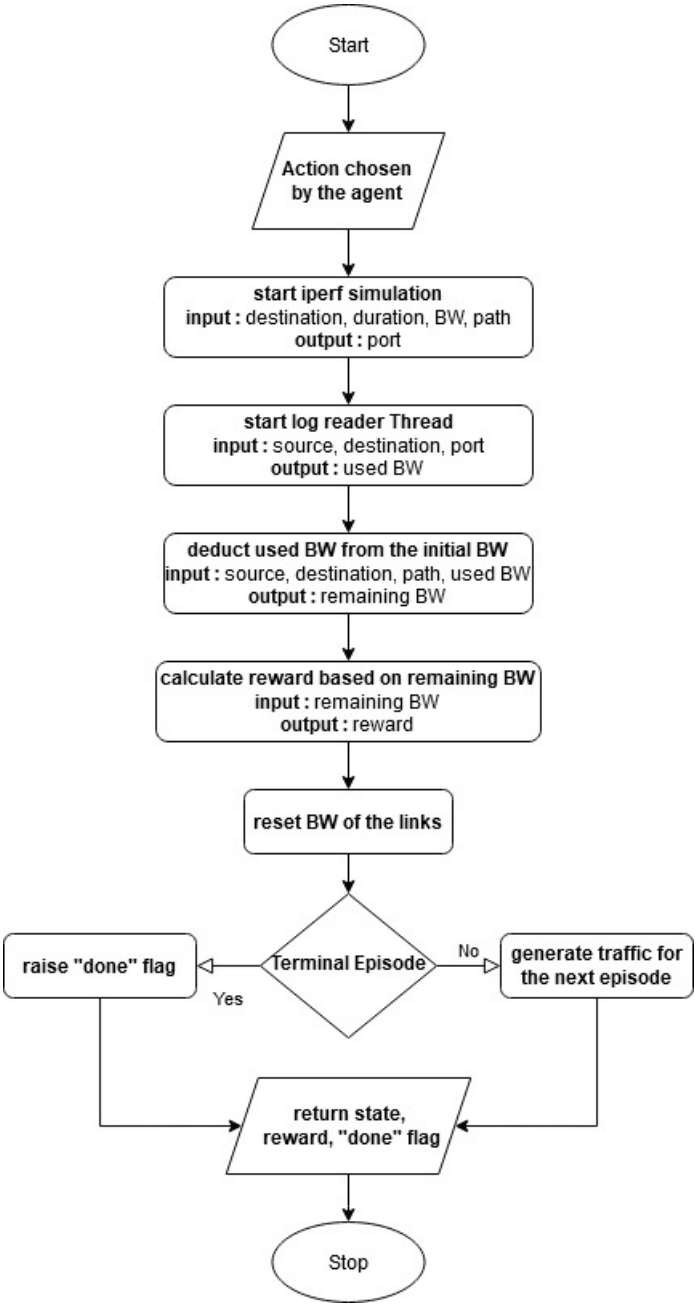

Figure 3.1: Flowchart of the environment's *step* function

3.3 Agent

After the problem definition and creating the model of the environment, the next step is to design the agent that will try to solve the problem. The chosen algorithm is the classic DQN. The Q-network that is designed in this algorithm, takes the current state of the network as input and outputs the Q-values associated with the actions that it can take. The agent can then either use these Q-values to base its decisions on or it can decide to discover new strategies, depending on if it wants to explore or if it wants to exploit known strategies. This strategy is implemented using the *epsilon-greedy* policy, so depending on the value of *epsilon*, the agent either takes random decisions or takes decisions based on the Q-values of the actions.

Due to a few problems that DQN is prone to suffer from, a few optimizations had to be added to the base algorithm to make it more robust. One of the most notorious flaws of DQN is its vulnerability to the *Catastrophic Forgetting* phenomenon, so to avoid it, an experience replay buffer is added to the agent's design. This buffer is used during the training process and it keeps the agent from forgetting what it has already learned after learning something new. This buffer keeps the results from every *step* in the form of a tuple, (s, a, r, s_{t+1}, d), that contains the state, the action, the reward and the subsequent state that is produced, as well as the *done* flag that indicates if that state was the terminal state or not. After gathering enough samples, the experience replay buffer can be used.

As the algorithm is also prone to behaving erratically because its parameters are updated after every episode, there will also be a target network alongside the original network to control and stabilize the learning process [25]. This target network is identical to the original network initially, but it is not updated at the same rate so if there are any major undesirable changes to the original network, this strategy reduces the impact that it has on the learning process.

3.3.1 Training

The training of the agent is done in an episodic manner. During each episode, there are a predefined number of *steps*. At the beginning of an episode, the environment is reset to its initial state, the replay buffer is initialized and the Q-network is created alongside a copy, which serves as the target network. After the initial routine, the agent has everything it needs to start the training process. The Q-values corresponding to the initial state are calculated, after which an action is chosen according to the *epsilon-greedy* policy. Based on the state and the action, a reward is calculated and the next state is generated. This process, that takes the agent from the current state to the next state, is stored in the replay buffer, which can be used after an enough number of instances, called *batch size*, are stored. After gathering enough experiences, the agent samples a batch of size *batch size*, from those experiences in each *step*. All the states in those experiences, $(s_{t_1}, s_{t_2}, ..., s_{t_{batchsize}})$ are passed through the original Q-Network to get the Q-values of the actions for each of

those states. The batch of "next states", $(s_{t+1_1}, s_{t+1_2}, ..., s_{t+1_{batchsize}})$, are passed through the target network where the Q-values are calculated according to equation 3.6, where γ is the discount factor and $(1 - done)$ only sets the Q-values to 0 when reaching the terminal state of an episode.

$$Q(s_t, a_t) = r_t + \gamma * ((1 - done) * max(\hat{Q}(S_{t+1}, a_{t+1}))) \tag{3.6}$$

The Q-values from the target network, are the desired Q-values for the original network and the goal is to approximate the Q-values of the original to the target network. After getting the Q-values from both networks, the loss function is called to calculate the error between them. The result is used to calculate the *gradient descent* and afterwards, the *backpropagation* of the parameters is done. After reaching a predefined number of *steps*, called the *synchronization frequency*, the target network's parameters are again set to be the same as the original network's parameters. Also, after each episode, the value of *epsilon* decays, so the agent gradually becomes more reliant on exploiting the strategies that it has already learned. The rate at which *epsilon* decays is inversely proportional to the total number of episodes. The whole training process is demonstrated in the following piece of pseudo-code.

Algorithm 1 Agent

```
for episode ← 1toMaxEpisodes do
  done ← False
  initialize Qnetwork
  TargetNetwork ← Qnetwork
  initialize ReplayBuffer
  while done ≠ True do
    Qvalues ← Qnetwork(state)
    destination, bw ← flow_generator
    if Epsilon > RandomInteger then
      action ← RandomAction
    else
      action ← argmax(Qvalues)
    end if
    NextState, reward, done ← env.step(action)
    memory += (state, action, reward, nextState, done)
    if memory.len() ≥ BatchSize then
      QNetwork ← update_parameters(memory.sample())
      if step%SynchronizationFrequency == 0 then
        TargetNetwork ← QNetwork
      end if
    end if
  end while
end for=0
```

3.3.2 Implementation

PyTorch, which is an open source machine learning framework based on the Python Torch library, was used to develop the agent [26]. First of all, the Q-Network is created with the *nn.Sequential* module of Torch. This network has four layers, the input layer, the output layer and the two hidden layers between the other two. The input layer has the same dimension as the state size, which depends on the number of destination hosts and the number of paths between those hosts and the source host. The last layer has the same dimension as the action space which is equal to the number of paths between each host pair. The output is equivalent to the action space. The activation function for all layers is Rectified Linear Unit (ReLU). The target network is a copy of the original Q-Network in its initial state and it is periodically synchronised with the original. The experience replay buffer is initialized at the same time, as well as the loss function and the optimizer. The chosen loss function is the most widely used loss function in regression which is Mean Squared Error (MSE). As MSE penalizes large error more than small errors, it can identify outliers and stabilize the learning process. The optimizer for the agent is Adam, which is an extension to the classic stochastic gradient descent. This optimizer is very straightforward to implement, with intuitive parameters that are easy to pick out that is also computationally efficient.

The implemented DQN agent has various parameters, called the "training parameters"in this work, that can be modified in the training process. Changing these parameters influences the learning process so they have to be chosen carefully. These parameters are listed below.

1. Number of episodes : The number of times the agent is exposed to the training loop. More episodes leads to more experiences, which can lead to a better learning process.
2. Learning rate
3. *Epsilon*
4. *Epsilon*'s decay rate : A faster decay rate means that the agent will explore less.
5. Maximum replay buffer size : When the maximum number of experiences has been reached, older experiences are replaces so this can be potentially useful, depending on the context of the project.
6. Batch size : A larger batch size means that more experiences are used for training each time.
7. Discount factor
8. Synchronization frequency : Frequency with which the target network is copied from the original Q-network.

Chapter 4

Results

This chapter covers the description of the training process and the tests that were performed. Before describing the training and testing processes and their respective results, the topology and the neural network that were used for this work are presented.

4.1 Preparation

The chosen topology for this work is an ARPANET topology with 12 switches and 12 hosts, with each host being connected to one of the switches. Figure 4.1 demonstrates how these switches are connected to each other.

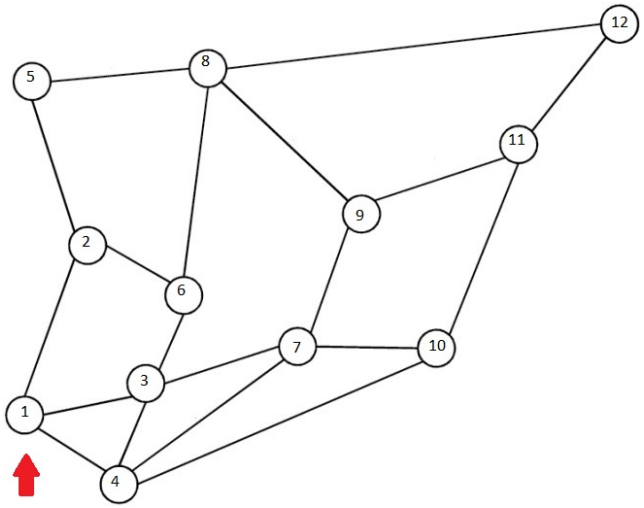

Figure 4.1: ARPANET Topology with 12 switches. The 12 hosts are not shown but each one is connected to one of the switches in the topology. Adapted from [27]

Switch "1"(S1) is the designated switch and the host connected to it, *H1*, is the source host. Although this topology is not nearly as big as a real-life network, it still provides a

decent amount of hosts and switches to make it challenging for the agent to learn how to balance the load efficiently. Every link, even the ones connecting the hosts to their respective switches, has a bandwidth of 100 Mbps. The delay and the loss parameters of the links are set to 1 ms and 0, respectively. This topology is stored in a text file where the links between nodes and their details are present. This text file is the one used to create the topology in *Mininet* and also, to create the *graph* that is used to represent network paths.

The neural network that was chosen has 4 layers, the input and the output layers and two hidden layers between them, as shown in figure 4.2. The input layer and the output layer have the dimensions of the state and the action space, respectively. The two hidden layers between them have 1800 and 1200 neurons, respectively. As mentioned before, all layers use the ReLU activation function. These values were chosen by means of trial and error.

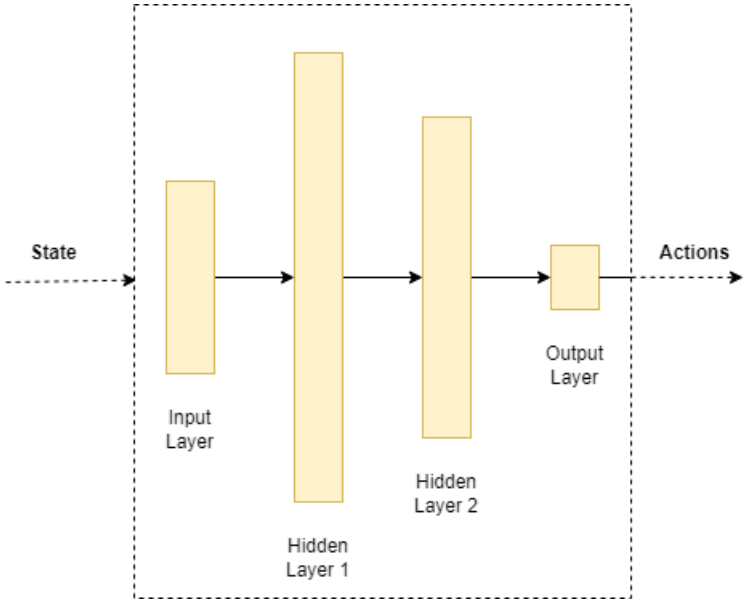

Figure 4.2: A simplistic view of the agent, with the neural network inside.

Considering that the topology, which is used in this project, has 11 destination hosts (12 in total where one is the source host) and there are 5 paths between each source-destination pair, the input layer's size is 55. The output layer has a size of 5 because there are 5 paths to choose from between the source and any given destination.

4.2 Training

Training is divided into two parts, the first covers the training of an agent in an environment where the generated traffic follows the same predefined pattern, every episode. The second part is for the environment where the generated traffic by the destinations hosts is randomized. After training the agent in each of these scenarios, it is tested in a set of specific conditions to evaluate its performance.

As mentioned earlier in chapter 3, the simulation of network traffic is done by using *Iperf*. However, one downside of using this approach is that it is extremely time-consuming and it requires high computation power to accelerate the process. Due to limited computation power and limited time for training and testing, another approach had to be adopted, at least for training which requires a lot more iterations than testing the agent.

This new approach consists of simulating the results of a successful *Iperf* data transmission. Basically, every time the agent chooses its desired action for a flow, it sends the action to the environment where the desired bandwidth of the flow is simply subtracted from the bandwidth of the path that was chosen. This drastically reduces computation time and allows the agent to be trained in a few hours instead of a few weeks. Every traffic simulation during the training process is done by using this strategy.

The following sections cover the description of how the traffic is generated by the other hosts, the first training scenario and the second training scenario, respectively, as well as a discussion section, where the results are discussed.

4.2.1 Artificial Traffic Generation

Using equation 3.5, it is possible to calculate that for the chosen topology with 11 destination hosts, there are 5 generated flows every step, which is the same for both of the training scenarios. These flows can only originate from the set of hosts known as destination hosts, but can have any destination, even the source host. This ensures that the source host only generates one flow per episode. This tactic of creating flows before the agent observes the environment, either randomly or according to a pattern, creates a wide variety of possible states the agent can find itself in, although the latter is more diverse. These flows have varying bandwidths, destinations and paths. This means that it is possible to have links with a lot of traffic, links that carry no traffic at all and links with a moderate amount of traffic in each step. This means that the agent can either choose paths that have heavily occupied links and get punished for it, or it can choose paths that pass along links with low traffic and get a positive reward for it.

In the scenario where traffic follows a predefined pattern, the number of different states the agent can find itself in is equal to the number of steps in an episode, because the pattern repeats itself in each episode. This approach allows the agent to go through the same states many times, which can be beneficial because it can learn from its mistakes, or it can be detrimental because the agent gets used to that scenario and it becomes impossible for it to adapt to other conditions.

In the other scenario, everything is generated randomly. This creates a lot of chaos that is the opposite of the other scenario. Going through a large number of states can make the agent very robust and prepare it for any kind of situation it can face. It is also possible to see how the agent reacts to different loads of traffic by analysing the rewards during times of lower or heavier traffic load.

4.2.2 First Training Scenario

The first training scenario is done with traffic that follows a predefined pattern. The traffic generation pattern is created using a simple *script* that creates a text file with the source, the destination, the path between them and the desired bandwidth of every flow that is going to be created. Each episode of the training process contains 50 flow requests, or in other words, there are 50 iterations of the *step* function per episode.

The training parameters that were used for the agent are shown below.

1. Number of episodes : 5000
2. Learning rate : 0.001
3. *Epsilon* : 0.4
4. *Epsilon*'s decay rate : 1/5000
5. Maximum replay buffer size : 1000
6. Batch size : 200
7. Discount factor : 0.9
8. Synchronization frequency : Every 100 steps

Some of these values, like the learning rate and batch size, were inspired by [7] but others were chosen mostly through experimentation with different values. After defining the parameters, the agent was trained for the specified number of episodes. The results include a graphic representation of the obtained rewards in function of episodes and also the average rewards in function of groups of 25 episodes, as shown in figures 4.3 and 4.4.

As seen in figure 4.3, the rewards that the agent receives increase slightly over time, with the lowest rewards reaching values smaller than -20 and the highest rewards reaching values of approximately 100. The moving average of the values of the rewards, with a period of 15, is also presented. Figure 4.4 makes it easier to visualize the gradual improvement of the rewards, although it is not a noticeable improvement. It is also possible to see that the initial episodes' rewards are significantly lower than the rest. At first, the agent does not know the environment and it also explores more often initially because of the value of *Epsilon*, which after decaying becomes less frequent. These reasons can justify the difference between the values of the rewards, the longer the training process goes on. This increase in rewards is a good sign because the agent is doing what it is supposed to do which is maximizing the rewards it receives.

However, there are a few episodes, after the 1000 episodes mark, that received significantly lower rewards compared to the rest of the episodes. Some of them received the same reward as the first few episodes which is strange, but justifiable because the generation of flow requests is not systematic and this adds randomness to the training

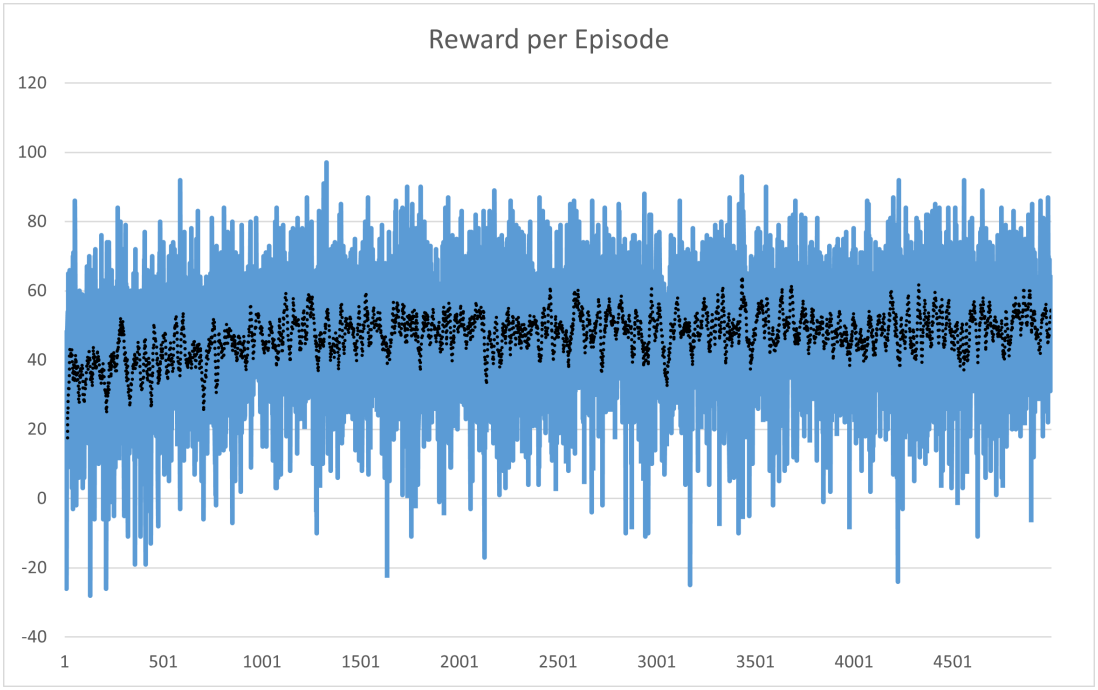

Figure 4.3: Total rewards per episode for the agent with predefined traffic generation (Blue) with the moving average (Black).

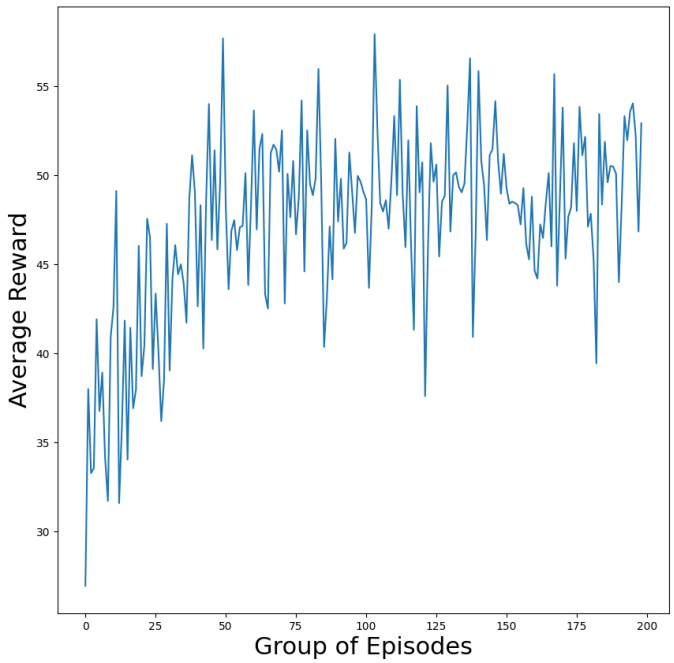

Figure 4.4: Average rewards for groups of 25 episodes for the agent with predefined traffic generation.

process. The desired outcome would be to have a more stable set of rewards the longer training goes on but there are a lot of fluctuations in the rewards, even after a few thousand episodes have passed. The agent is learning how to deal with the environment in this scenario but the results are sub-optimal. Nonetheless, its performance can only be truly evaluated after the testing process.

4.2.3 Second Training Scenario

The second training scenario involves randomized traffic generation which can be challenging for the agent. The training parameters are identical to the first scenario, with the only difference between the two being the generation of traffic from the other hosts. In order to track how different network loads could affect the results of training, another graphic is created to show the average load per 25 episodes to be compared with the average rewards graphic. The following figures demonstrate total rewards per episode, average rewards and the overall load in the network per episode, respectively.

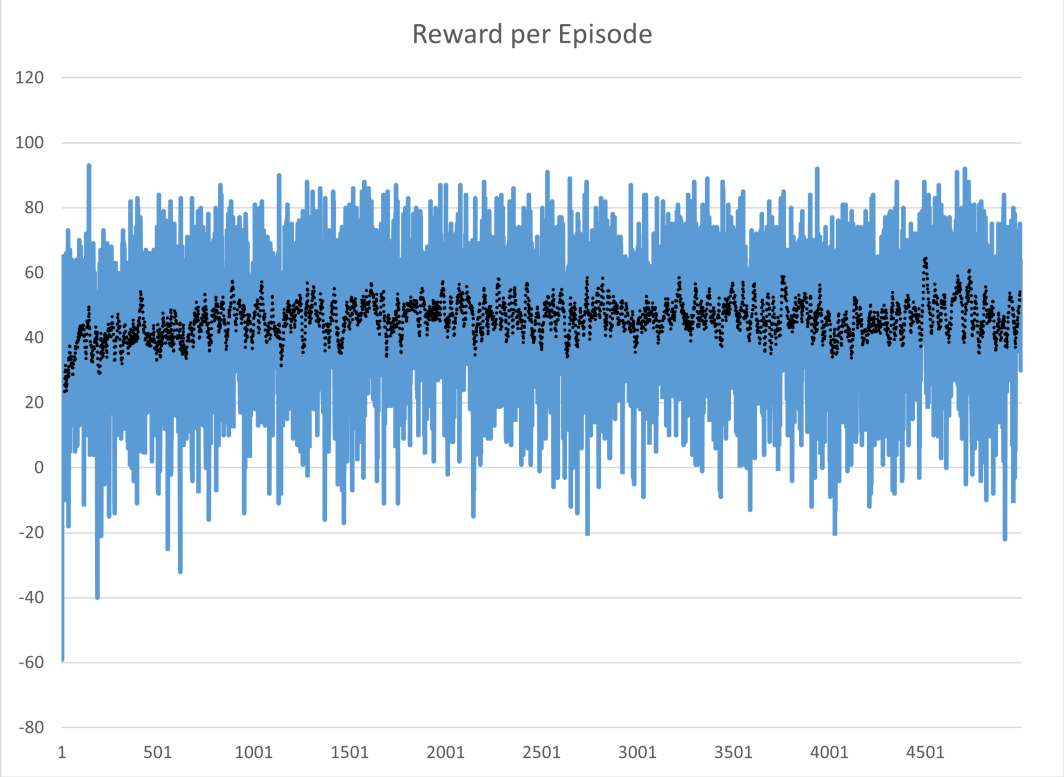

Figure 4.5: Total rewards in each episode for the agent with random traffic generation (Blue) with the moving average (Black).

As seen in figures 4.5 and 4.6, the obtained rewards in the second scenario are similar to the ones in the first scenario. There are a few small differences, like the lowest reward being smaller than the lowest reward in the previous scenario, but the most eye-catching one is the stability of the rewards in the second scenario. The second one seems to have a more stable training than the first one, which is odd considering that the second scenario

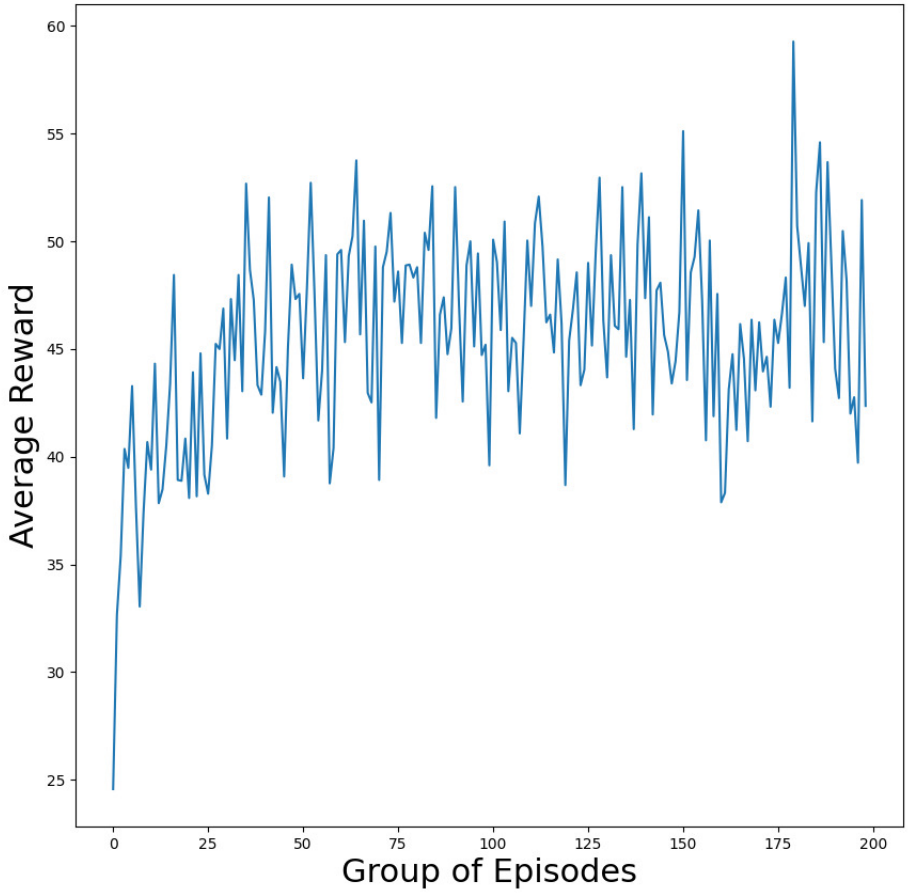

Figure 4.6: Average rewards for groups of 25 episodes with random traffic generation.

trains the agent with network load that is randomly generated. It is also possible to see that the average of rewards does not increase a lot during the training process. All of these factors point towards possible problems with the training process.

In Figure 4.7, it is possible to see that the total generated load varies from approximately 3050 Mbps to almost 3300 Mbps per episode. This means that the agent is exposed to a wide range of different network loads which is what was intended. In order to visualize the effects of the network load on the rewards that the agent gains, the average rewards were compared to the average load of the network, as shown in figure 4.8.

This comparison makes it possible to see that network load actually affects the rewards that are gained. Looking at specific examples, like just past the $x = 100$ mark where there is a huge spike in network load, it is possible to see that rewards indeed get lower, but the relationship between these two entities may not be linear. This could be due to the fact that the generated flows are randomized which makes it hard to predict exactly how a change in network load affects the rewards.

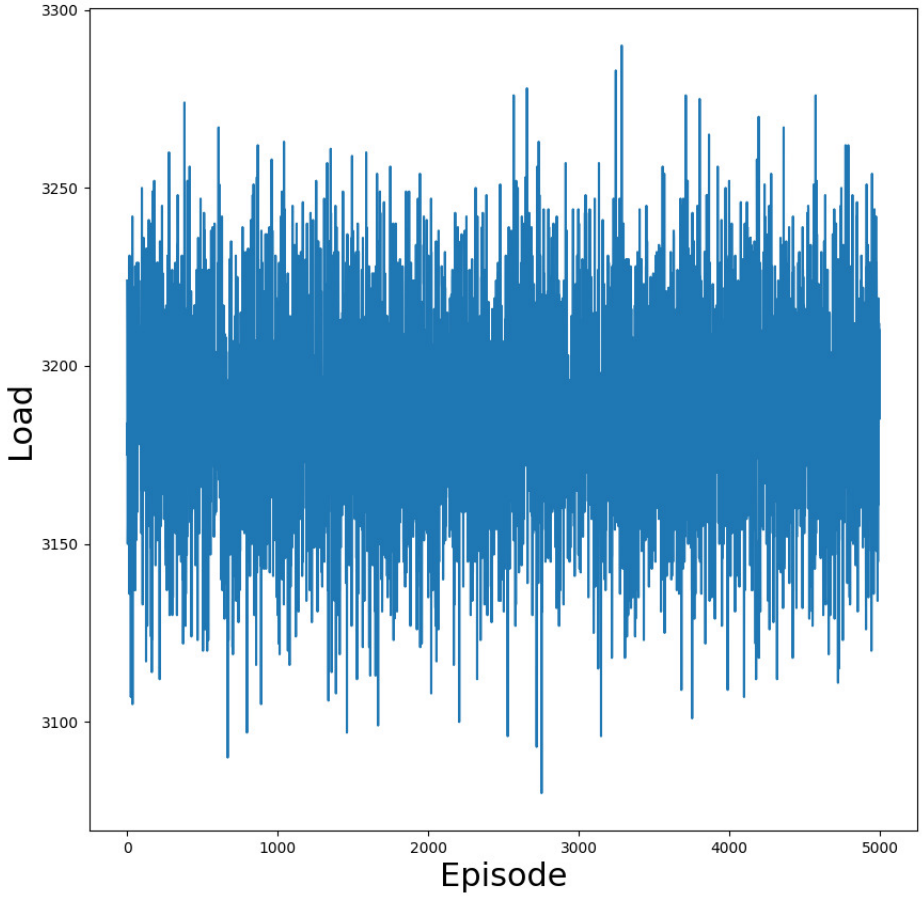

Figure 4.7: Total generated load per episode.

After the training of the agent is done, it is time to test the performance of the agent in a testing scenario. Overall, the results were positive but they show room for improvement.

4.3 Testing

The testing scenario that was considered, consists of 100 flows that have to be managed during 100 steps. These flows are managed by the agent that is trained in scenario 1, the agent trained in scenario 2 and also on a simple routing algorithm, Dijkstra, to serve as benchmark for the agent. The evaluation metric of these tests is the reward they obtain during that episode. In total there are 3 tests to be done, so in order to make the testing process equal for all of them, a file was created with the traffic generation pattern during every step, as well as another file with the flow request generation pattern. This allows the same conditions to be guaranteed for all tests, allowing the results to be comparable. It is possible to evaluate the performance of the DRL agent and determine the effects

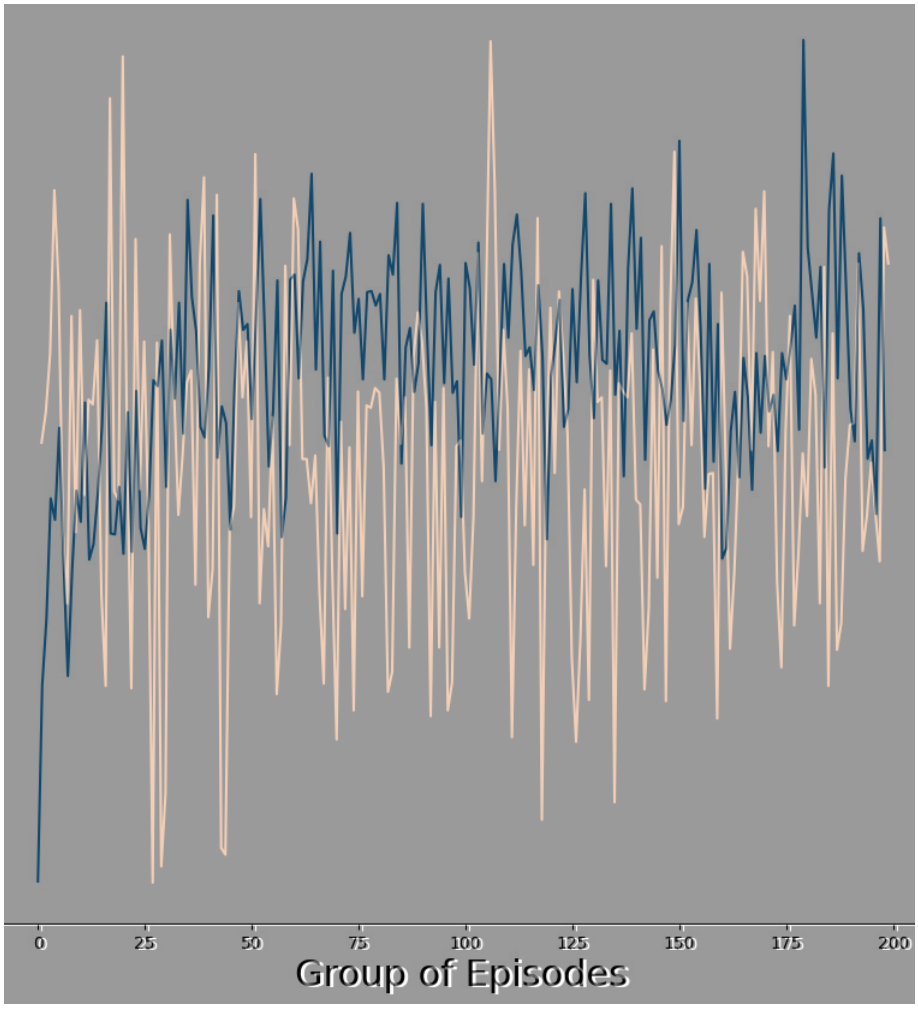

Figure 4.8: Average reward (blue) compared to average network load (orange).

of having different training scenarios by comparing the rewards from the agent in each training scenario with the results obtained from Dijkstra, as well as by comparing the results from both training scenarios.

As mentioned before, unlike in the training process, *Iperf* was used in the testing scenario. However, it was only used to simulate the traffic from the source host, not the rest of the traffic that is generated in the network. This is because the rest of the traffic is not as important as the traffic from the source host.

To compare the agent's performance to Dijkstra's, the testing scenario was run using this algorithm as the decision maker. This is fairly easy to implement due to the way the environment was set up, because running Dijkstra is equivalent to choosing the path with the lowest index from the action space. The rewards that each approach gained per step, during testing are shown below.

As seen in figures 4.9 and 4.10, the two different training scenarios have led to the agent behaving differently in the testing scenario. The test results are better when the agent is trained in the first scenario, with the maximum reward per step reaching 5 and

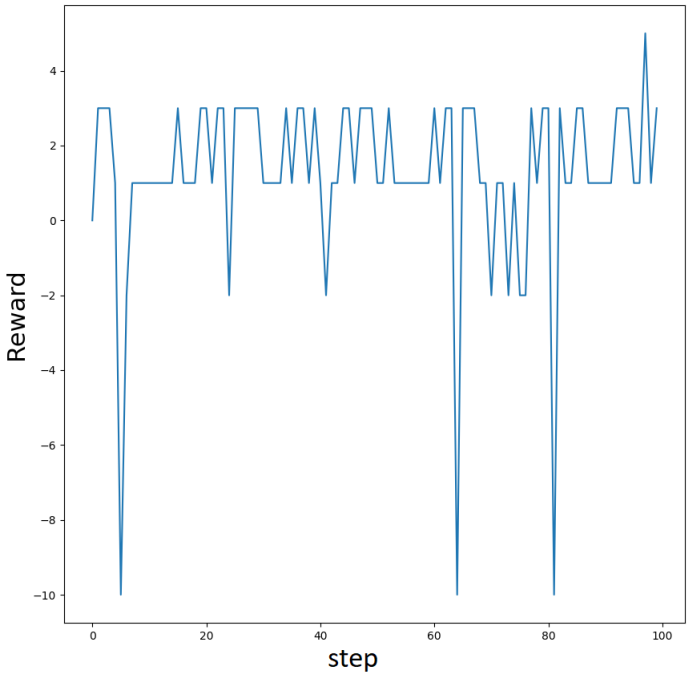

Figure 4.9: Reward per step in the first scenario.

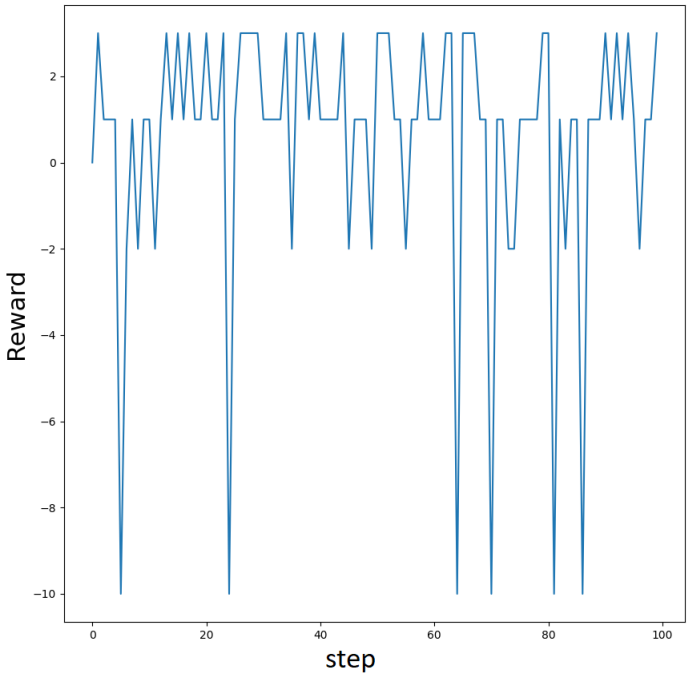

Figure 4.10: Reward per step in the second scenario.

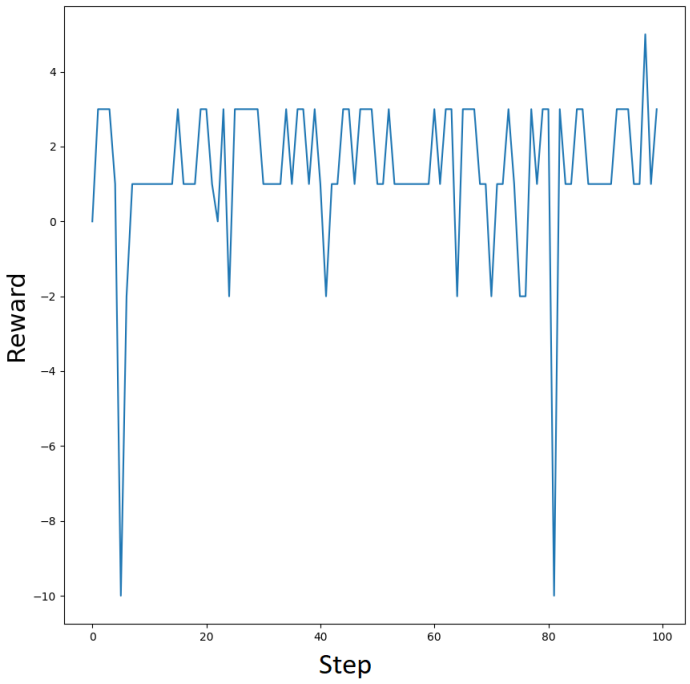

Figure 4.11: Reward per step for the Dijkstra algorithm.

the minimum reward reaching -10. The sum of rewards earned during the episode is 127. The results of the test performed on the agent trained in the second scenario show that the maximum reward, per step, is never greater than 3, but the minimum reward received is -10. The sum of rewards during the episode only reaches 60. Figure 4.11 depicts the Dijkstra algorithm's results, with rewards reaching a maximum of 5 and a minimum of -10, and an overall sum of 137.

By comparing figures 4.9 and 4.11, it is evident that the performance of the agent trained in the first scenario is similar to the Dijkstra algorithm. Given that the results obtained by the two algorithms are so alike, it is possible to realize that the main strategy of the agent is to choose the shortest path. However, there were instances where the chosen action was not according to this strategy, hence the difference in the total reward between the two algorithms. These results are not desirable but they can be improved by finding the right setup for the agent.

The comparison between the results of the agent trained in the second scenario and Dijkstra show that, although the agent performed worse, the adopted strategy was totally different from the agent in the other scenario. The performance of the agent in this scenario, was the worst overall. However, even though the adopted strategy might not have been the best, it is possible to see that if the algorithm is pointed in the right direction, it can manage the bandwidth of the network without only choosing the shortest path. If the right training conditions are given to the agent, it can learn a different strategy that

can actually be useful for the problem that this work is trying to resolve.

The difference between the two scenarios comes down to how the agent reacted to each one of them. The first scenario exposed the agent to the same states which made the agent adopt selecting shortest paths as its main strategy. Perhaps if the topology was more complex, the agent would have been obligated to adopt a more complicated approach. The second scenario exposed the agent to a large number of different states, which led it to adopt a different strategy during the testing process.

4.4 Discussion

The results that were obtained from the testing process show a sub-optimal performance by the DRL agent. Although the end results are unsatisfactory, it is possible to avoid these issues in future works by identifying the possible sources of the problem. In this work, various parameters, such as the training parameters, and simulation conditions, such as the traffic creation pattern and how flow requests are generated, were chosen subjectively, so it is hard to simply identify what the problem is. This can only be done by focusing on the possible causes of the problem and experimenting with different settings.

The first possible cause is the topology that was used during the development of this work. As the DRL agent's performance depends on the experiences that it gathers during training, the first thing to question is if the topology is big enough and if it had the right complexity. A bigger and more complex network leads to the creation of longer, non-trivial paths that can make the training process more challenging for the agent. If the paths that exist in the topology all have the same number of hops (same length), the agent is not able to learn properly. Considering this, using a bigger, more complex topology could be beneficial to the agent's training.

Another contributing factor to the poor results could be the way the action space and the state space are defined. By defining a more complex state space or action space, a lot more information can be given to the agent, which could have resulted in a better performance. If this is in fact the root of the problem, it means that the designed solution does not have the Markov Property.

The training process can also be designed in many different ways, because it contains numerous parameters and conditions that can be modified to produce different results. It can be a simple change to the training parameters, such as the learning rate or batch size, that solves the problem. The problem can also be related to the simulated settings in the environment which can be optimized by trial and error, or by adopting already existing strategies used in other works.

The agent's test results were compared to Dijkstra, which can also be considered an unfair comparison, as the former's aim is to balance the load in the network and the latter only aims to choose the shortest paths. A more fair way of evaluating the agent would be to use the *Fairness Index* to see how well the agent manages the load of the network.

Chapter 5

Conclusions

This chapter includes a summary of the findings from the dissertation as well as recommendations for further study and potential system upgrades.

5.1 Conclusion

Traffic Engineering is a field in need of innovation, in order to battle the rapid expansion of the telecommunications networks and their rising complexity. Deep Reinforcement Learning is a viable option for tackling this problem, so there are a lot of efforts to design solutions that can be used in the field of Traffic Engineering.

This thesis aims to provide a DRL-based solution that is capable of performing *Load Balancing* in a network, where the proposed agent is implemented in the "designated source"that is in a one-to-many configuration with the rest of the switches in the network. Primarily, a state-of-the-art study of the subject matter was conducted, which revealed that already existing solutions are producing positive results in addressing the TE problem.

The problem was then formally defined, and the proposed model was presented. Afterwards, the design was implemented using a number of different tools including *PyTorch*, *OpenAI Gym*, *Mininet* and *Ryu*. The designed solution employs a DQN that is trained in two scenarios: one in which the network's existing traffic is generated randomly, and one in which the generated traffic is systematic, with the goal of creating the same agent in two different environments and observing the results. The results of the training show that the agent learns how to deal with environment, in both scenarios. However, when the agent was put to test against the classic routing algorithm, Dijkstra, the results were different than what was expected. In the first scenario, the results were similar to Dijkstra's, as the sum of rewards for the agent and Dijkstra were 127 and 137, respectively.

The results were similar but slightly worse because the agent's main strategy was the same as Dijkstra's, which was to take the shortest path. The results in the second scenario were poor when compared to the agent in the first scenario, as the sum of rewards only reached 60. The results were poor when compared to Dijkstra, but the agent adopted a different strategy than in the first scenario. This is a clear indication that the agent adapts itself to the environment it is trained in, so the training process is crucial to the development of the agent. Since the first scenario produced better results than the second, it is possible to conclude that the random traffic generation strategy is ineffective for the training process designed in this work. All in all, the obtained results were not desirable, but this thesis paves the way for future works that can use it to avoid having the same problems.

5.2 Future Works

This thesis covers a small part of the world of DRL solutions applied to TE problems. There are a lot of possibilities for future works as there are a lot of optimizations that can be made to various parts of this work. The first would be to recreate this work with the training scenario implemented differently while also using a more complex topology. Another interesting idea would be to try using more complex versions of the basic DQN algorithm, such as the DDQN and Dueling DDQN, given that these algorithms are better than DQN. The last one is to implement a playground where different configurations, topologies and parameters can be easily tested in order to facilitate Traffic Engineering using Deep Reinforcement Learning.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] P. Pinyoanuntapong, M. Lee, and P. Wang. "Delay-Optimal Traffic Engineering through Multi-agent Reinforcement Learning". In: *INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS 2019* (2019), pp. 435–442. DOI: 10.1109/INFCOMW.2019.8845154 (cit. on pp. 1, 2, 20–22, 27).

[3] Z. Shu et al. "Traffic Engineering in Software-Defined Networking: Measurement and Management". In: *IEEE Access* 4 (2016), pp. 3246–3256. ISSN: 21693536. DOI: 10.1109/ACCESS.2016.2582748 (cit. on p. 1).

[4] H. An et al. "Dynamically Split the Traffic in Software Defined Network Based on Deep Reinforcement Learning". In: *2020 International Wireless Communications and Mobile Computing, IWCMC 2020* (2020), pp. 806–811. DOI: 10.1109/IWCMC48107.2020.9148369 (cit. on pp. 2, 17, 23, 24).

[5] Y.-R. Chen et al. "RL-Routing: An SDN Routing Algorithm Based on Deep Reinforcement Learning". In: *IEEE Transactions on Network Science and Engineering* 7.4 (2020), pp. 3185–3199. ISSN: 2327-4697. DOI: 10.1109/tnse.2020.3017751 (cit. on pp. 2, 25, 27, 28).

[6] Z. Xu et al. "Experience-driven Networking: A Deep Reinforcement Learning based Approach". In: *arXiv* (2018), pp. 1871–1879. ISSN: 23318422 (cit. on pp. 2, 19, 25, 27).

[7] B. Brown and A. Zai. *Deep Reinforcement Learning in Action*. 2020, p. 384. ISBN: 9781617295430 (cit. on pp. 2, 4–9, 11–18, 41).

[8] M. Lapan. *Deep Reinforcement Learning Hands-On*. 2019, p. 546. ISBN: 9781788834247 (cit. on pp. 5, 7, 13).

[9] A. M. Andrew. "Reinforcement Learning: An Introduction". In: *Kybernetes* 27.9 (1998), pp. 1093–1096. ISSN: 0368492X. DOI: 10.1108/k.1998.27.9.1093.3 (cit. on pp. 5, 6, 11).

[10] E. Barnett. *Understanding Markov Decision Processes*. 2019. URL: https://towardsdatascience.com/understanding-markov-decision-processes-b5862c192ddb (cit. on p. 6).

[11] A. Shrestha and A. Mahmood. "Review of deep learning algorithms and architectures". In: *IEEE Access* 7 (2019), pp. 53040–53065. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2912200 (cit. on pp. 9–12).

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 11).

[13] T. P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings* (2016). arXiv: 1509.02971 (cit. on p. 13).

[14] Y. Xiao et al. "Learning Multi-Robot Decentralized Macro-Action-Based Policies via a Centralized Q-Net". In: *Proceedings - IEEE International Conference on Robotics and Automation* (2020), pp. 10695–10701. ISSN: 10504729. DOI: 10.1109/ICRA40945.2020.9196684. arXiv: 1909.08776 (cit. on p. 15).

[15] Z. Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *33rd International Conference on Machine Learning, ICML 2016* 4.November 2015 (2016), pp. 2939–2947. arXiv: 1511.06581 (cit. on p. 15).

[16] N. Geng et al. "A Multi-agent Reinforcement Learning Perspective on Distributed Traffic Engineering". In: *Proceedings - International Conference on Network Protocols, ICNP* 2020-Octob (2020). ISSN: 10921648. DOI: 10.1109/ICNP49622.2020.9259413 (cit. on p. 22).

[17] P. Sun et al. "A Scalable Deep Reinforcement Learning Approach for Traffic Engineering Based on Link Control". In: *IEEE Communications Letters* 25.1 (2020), pp. 171–175. ISSN: 1089-7798. DOI: 10.1109/lcomm.2020.3022064 (cit. on p. 27).

[18] *Mininet*. URL: https://github.com/mininet/mininet (cit. on p. 31).

[19] A. Tirumala. "Iperf: The TCP/UDP bandwidth measurement tool". In: *http://dast.nlanr. net/Projects/Iperf/* (1999) (cit. on p. 32).

[20] D. C. Plummer. *An Ethernet Address Resolution Protocol*. 1982-11 (cit. on p. 32).

[21] R. project team. *RYU SDN Framework*. URL: https://book.ryu-sdn.org/en/Ryubook.pdf (cit. on p. 32).

[22] HP. *OpenFlow flow table*. URL: https://techhub.hpe.com/eginfolib/networking/docs/switches/5940/5200-1028b_openflow_cg/content/491966856.htm (cit. on p. 32).

[23] *NetworkX*. URL: https://networkx.org/ (cit. on p. 32).

[24] G. Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540 (cit. on p. 33).

[25] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (7540 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236 (cit. on p. 35).

[26] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 37).

[27] A. de Sousa, P. Monteiro, and C. B. Lopes. "Lightpath admission control and rerouting in dynamic flex-grid optical transport networks". In: *Networks* 69 (1 2017-01), pp. 151–163. ISSN: 00283045. DOI: 10.1002/net.21715 (cit. on p. 38).