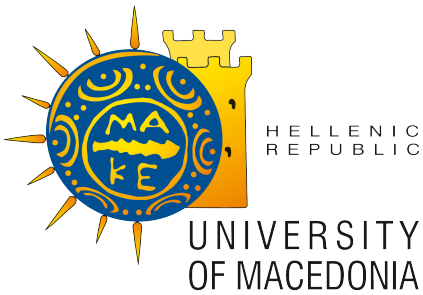
University of Macedonia, Greece
School of Information Science, Dept. of Applied Informatics

# Development of Network Services Embedding method using Reinforcement Learning




Anastasios Karageorgiadis

Thesis Committee:
Professor Panagiotis Papadimitriou (UOM)
Professor Nikolaos Samaras (UOM)
Professor Ioannis Refanidis (UOM)

Thessaloniki, August 2023

Πανεπιστημιο Μακεδονιας,Ελλαδα
Σχολη Πληροφορικης, Τμημα Εφαρμοσμενης Πληροφορικης

# Ανάπτυξη Μεθόδου Ενσωμάτωσης Δικτυακών Υπηρεσιών μέσω Ενισχυτικής Μάθησης


Αναστάσιος Καραγεωργιάδης

Εξεταστική Επιτροπή:
Καθηγητής Παναγιώτης Παπαδημητρίου (ΠΑΜΑΚ)
Καθηγητής Νικόλαος Σαμαράς (ΠΑΜΑΚ)
Καθηγητής Ιωάννης Ρεφανίδης (ΠΑΜΑΚ)

Θεσσαλονίκη, Αύγουστος 2023

# Abstract

In the last few years, the Network Functions Virtualization (NFV), a network architecture approach, has become essential for all the services provider companies. With NFV architectures, providers can reduce the requirements for specialized hardware [1], which may stay unused for most of the time if it serves only a few requests. But in order to use most of the cloud infrastructure, they require methods for mapping a service onto the virtualized infrastructure. There's where Network Service Embedding comes into play, to help providers optimize the distribution of the physical resources to fulfill the customers' needs as fast as possible and in a more reliable way. Network Service Embedding [2] (NSE) methods can take into account more complex needs that a client may specify, such as low latency, bandwidth limits except for CPU or memory demands. Also, NSE helps providers to manage their resources efficiently, therefore, serving as many clients in a given period of time, is giving them the ability to increase their profits. This is also important for the clients as they can experience the quality of service and lower costs based on their needs. The purpose of this Master's thesis is to develop a method for the optimized embedding of network services onto a virtualized infrastructure (e.g., data center) using supportive learning techniques based on Reinforcement Learning algorithms, as opposed to heuristic methods that are mostly employed. For the implementation of this work, Python [3] was used as programming language, the DRL models developed using Tensorflow [4] framework and the generated service graph were created with NetworkX [5] framework.

# Περίληψη

Τα τελευταία χρόνια η Εικονικοποίηση Δικτυακών Λειτουργιών, μία αρχιτεκτονική προσέγγιση δικτύων, έχει γίνει απαραίτητη για όλους τους παρόχους υπηρεσιών [1]. Με την Εικονικοποίηση των Δικτυακών Λειτουργιών [2], οι πάροχοι μπορούν να μειώσουν τις ανάγκες τους για εξειδικευμένο υλικό, το οποίο μπορεί να μην αξιοποιείται το περισσότερο χρόνο, αν εξυπηρετεί μόνο μερικά αιτήματα. Αλλά για να αξιοποιηθεί στο μεγαλύτερο βαθμό η υποδομή νέφους, απαιτούνται μέθοδοι τοποθέτησης μίας υπηρεσίας σε μία εικονικοποιημένη υποδομή. Εκεί είναι που εμφανίζονται οι μέθοδοι Ενσωμάτωσης Δικτυακών Υπηρεσιών, για να βοηθήσουν τους παρόχους να βελτιστοποιήσουν την κατανομή των φυσικών πόρων και να εκπληρώσουν τις ανάγκες των πελατών τους όσο το δυνατόν γρηγορότερα και πιο αξιόπιστα. Οι μέθοδοι Ενσωμάτωσης Δικτυακών Υπηρεσιών μπορούν να λάβουν υπόψιν πιο περίπλοκες ανάγκες που ορίζουν οι πελάτες, όπως η χαμηλή καθυστέρηση, όρια στο εύρος ζώνης εκτος από τις απαιτήσεις επεξεργαστικής ισχύς ή μεγέθους μνήμης. Επιπλέον, η Ενσωμάτωση Δικτυακών Υπηρεσιών, βοηθάει τους παρόχους να διαχειριστούν πιο αποδοτικά τους πόρους τους, καθώς η ταυτόχρονη εξυπηρέτηση περισσότερων χρηστών, δίνει την δυνατότητα για περισσότερα οφέλη. Αυτό είναι εξίσου σημαντικό και για τους πελάτες γιατί έχουν καλύτερες εμπειρίες χρήσης και χαμηλότερο κόστος ανάλογα με τις ανάγκες τους. Ο σκοπός αυτής της εργασίας είναι η ανάπτυξη μίας μεθόδου για την όσο το δυνάτον βέλτιστη ενσωμάτωση υπηρεσιών δικτύου σε μία εικονικοποιημένη υποδομή (π.χ. κέντρο δεδομένων) χρησιμοποιώντας υποστηριζόμενες τεχνικές μάθησης βασισμένες στην Ενισχυτική Μάθηση, σε αντίθεση με τις προτεινόμενες ερευτικές μεθόδους που ως επι το πλείστον χρησιμοποιούνται κατα κόρον. Για την υλοποίηση της παρούσας εργασίας χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python [3], σε συνδυασμό με το εργαλείο Tensorflow [4] για την ανάπτυξη του μοντέλου καθώς επίσης και του εργαλείου NetworkX [5] για την δημιουργία δικτυακών υπηρεσιών με την μορφή γράφου.

# Acknowledgements

First, I would like to thank my advisor Mr. Panagiotis Papadimitriou for his guidance, and Mr. Angelos Pantelas for his help and his useful comments.

I am also grateful to my family, which has been supporting me all the time, so as to achieve my dreams.

# Contents

# List of Figures

# Chapter 1

# Introduction

The current status quo of everyday life, requires people to be connected through the internet, in order to work, communicate, etc. These kinds of needs are increasing the demands for network infrastructures, in a way that more and more users could be added from day to day, and the performance of the provided services must also be improved. In order to address these uprising client requests, service providers are looking for new ways to make better use of their physical resources, as a matter of serving more clients faster, and more reliably while reducing the overall costs and of course, increasing their profits. All the above lead the path for Software Defined Networks [8] to appear, which are having as the main goal to reduce the need for specialized hardware equipment, and move onto general purpose equipment like typical personal computers or servers. This means that a PC has to reproduce the operation of an actual network component like a switch, or a firewall, this is done by specific software by using a virtualized environment that simulates the physical components. As a further step, this virtualization has to take into account all the big or small network functions that are required for a network to operate properly.

So, this implies that multiple continuous or not, network functions that may or may not be dependent on each other, must be mapped accordingly in a way that relies on some restrictions. A set of these network functions is represented as a graph. The problem that arises from this, is how to map these network functions inside a virtualized infrastructure and how to distribute them in an efficient way across the infrastructure's components. This problem is called Network Service Embedding better known as Virtual Network Embedding, and it is the subject of the current master thesis. In a nutshell, this work is about finding and testing a different approach for solving the Network Service Embedding problem, based on supportive machine learning techniques.

## 1.1 Thesis Contribution

In this Master thesis, the main objective is to quantify the potential benefits of using Reinforcement Learning methods for Network Service Embedding, such as better resource utilization or lower latency between nodes of a network service graph, compared to state-of-the-art heuristic and exact methods. The main idea is to extract some useful conclusions about the Reinforcement Learning methods, which as an approach can lead to an easier scaling to real-world scenarios. For example, by adding more requirements the dimensionality will be increased, this is easier to adapt in a Reinforcement Learning agent than to a classic heuristic approach. Also, as a further step, it would be better to experiment with our Reinforcement Learning methods on service graphs that stem from vehicles or robotics services.

## 1.2 Thesis Outline

In Chapter 2 presents all the background information needed for this master thesis. There is also an overview of Network Function Virtualization (NFV), and Virtualized Network Functions (VNFs), the difference between them and the basic idea for the Embending of Virtualized Network Functions (VNFs). In Chapter 3 the problem of Embending Virtual Network Functions using Reinforcement Learning is stated and a reference to the most recent related approaches that have been published by other authors. In Chapter 4 the overall design and implementation of the proposed work in this master thesis, is described. In Chapter 5 the performance of the proposed approach is evaluated compared to a simple heuristic baseline algorithm. Finally, Chapter 6 acts as an epilogue for this thesis, presenting some useful conclusions along with the possible future improvements and all the lessons learned throughout this process.

# Chapter 2

# Background

## 2.1 Network Function Virtualization

### 2.1.1 Introduction

Over the last few years, Internet Service Providers [9] are faced a lot of challenges arising from the composition of their network infrastructure, which is mainly built up from physical network devices (hardware equipment), are dedicated to a specific function or are related to only a specific group of a network operation (e.g. firewall, load balancer, NAT, etc). This approach is causing the network infrastructure to be very difficult to maintain, as it becomes harder to expand and upgrade it; while the customers' needs are increasing each day (e.g. especially during the Covid-19 period, Internet/network demands have met a historic high). Furthermore, VNE's purpose is to find the appropriate threshold between making online embedding decisions faster and pursuing a long-term objective, like increasing ISP revenues with better resource utilization. In an effort to secure the best possible level of service, an ISP usually deals with this increase by adding more and more hardware equipment. This way of handling the situation is leading them to raise the overall operational and sustainability costs, having one eye on not burdening the customers with further expenses, while the other is on keeping up with the competitors. Taking all these into account, a Network Function Virtualization solution manages to redesign the way that network architecture is conceived, utilizing all the benefits deriving from the virtualization process. In a nutshell, it transforms all the dedicated hardware equipment into virtual instances, which as a result, are executable on general-purpose machines whose computational power and memory capacity make them able to host a quite impressive number of those kinds of instances. With the general purpose machines to be servers, more powerful personal computers.

### 2.1.2 Definition

With the term Network Function Virtualization [2], we referred to the process that decouples network functions [10] from the underlying dedicated hardware and "realizes" them into a form of software, named Virtual Network Functions, that is enabled to run in any resource-sufficient virtual machines. In a nutshell, NFV refers to an overall concept as a framework for running software-defined network functions.

### 2.1.3 Use case

The Network Function Virtualization (NFV) approach adopted by Telecommunication and Services provides companies with deployment management and scaling of the network functions. The NFV also allows for decoupling and virtualization of existing Operational Support Systems, systems used by operators to manage their communication networks, or legacy and dedicated hardware to make them software-driven by using standardized hardware. The advantages of NFV are the reduced overall costs both for ISPs and customers, and the ability to have a faster evolution of service since the interconnection of network functions no longer demands any new hardware addition or rearranging/modifying the infrastructure components.

## 2.2 Virtual Network Functions

### 2.2.1 Definition

The Virtual Network Function is the implementation of a physical network function by utilizing software decoupled from the underlying hardware infrastructure. To simplify it, any virtual device usually is called a Virtual Network Function (VNF), or in other words, a VNF replaces network hardware with software [11] and it can be freely moved in the physical infrastructure inside commodity servers. Many VNFs create a Network Service [12].

### 2.2.2 Examples

Usually, nowadays, VNFs are deployed in the cloud infrastructure as microservices, that can work independently or can be combined to provide essential networking functionality to be used by service providers or even some third-party end users. Some examples of virtual network functions include: a) Network routing, such as Domain Name Service (DNS), b) Natural Address Translation (NAT). There are also VNFs for security reasons, such as c) malware detection, d) intrusion detection (ID), and e) Virtual Private Network (VPN) services. In addition, there are network functions that are also virtualizable for

f) traffic analysis, prediction, and Quality of Service (QoS) measurement purposes. At last, there are also g) VNFs for network and resource load balancing[11].

## 2.3 Network Service Embedding

### 2.3.1 Introduction to Network Service Embedding

The Network Service Embedding also referred to as Virtual Network Embedding [13] deals with formulating the best way to place Virtual Network Functions, which represent the network's functional components, in the most suitable location of a Substrate Network(SN) [14]. More particularly, the VNE problem deals with the challenge in network virtualization that is about optimizing the resource allocation[12, 15] inside an SN. The solution to the problem implies ensuring that the resource requirements are not violated and furthermore that the proposed mapping is optimal according to some specific objectives, for example, inter-rack traffic minimization.

## 2.4 Substrate Network

A substrate network is a network represented as a graph. More precisely a substrate network is a directed graph notated as `G(N, F)`, where **N** refers to the set of the nodes or the physical routers, switches, servers, etc that are modeled as vertices of the graph, and **F** is referred to the set of edges or the physical links between the actual components of a network [14] (e.g. core switches).

## 2.5 Network Topologies

### 2.5.1 What is a Network Topology?

Network topology is the architecture of all the physical components of a Network. More precisely, with the term network topology, we refer to the way that all the hardware equipment of a network is connected or structured in a top-down design approach. In most cases, this is represented by a tree schema but there are some exceptions, like peer-to-peer networks, which are structured like a circle. Lately, there are also virtual networks that are simulating a specific type of network topology. So, in conclusion, a more accurate definition of network topology would be this, a network topology is the representation of the way that any kind of physical or virtualized network equipment is placed and connected with each other, in an order that forms a network.

### 2.5.2 The main goal

The main goal of constructing a network infrastructure is to connect as many as possible endpoints (mainly with endpoints referring to servers) by using switches that have a limitation on the number of ports. The basic concept is to use switches with the smallest number of ports that can cover the

requirements, in order to reduce the overall cost. So, by connecting switching elements and forming a topology, in a smart and efficient way, a network can interconnect a large number of endpoints.

### 2.5.3 CLOS Topology

The CLOS network topology [7] is a hierarchical topology that has multistage switching with the same cost for each route and that's why it is also sometimes referred to as Equal Cost Multiple Paths topology. The main benefit of the CLOS network is that reduces the number of ports required from a switch. CLOS networks were invented by Edson Erwin [16] back in 1938 as a more scalable approach to connect network nodes and it was formalized in 1952 by Charles Clos [17], who used this topology to redesign the telephone nodes' interconnection in order to optimize telephony network systems. The CLOS topology was used before IP networks show up, and in our days is used to create a Leaf and Spine system of interconnecting Edge switches (Top of the Rack switches) together through Spine switches. It is built so that, each Leaf or Edge switch is connected to all Spine switches. The CLOS topology is shown in the following figure 2.1.

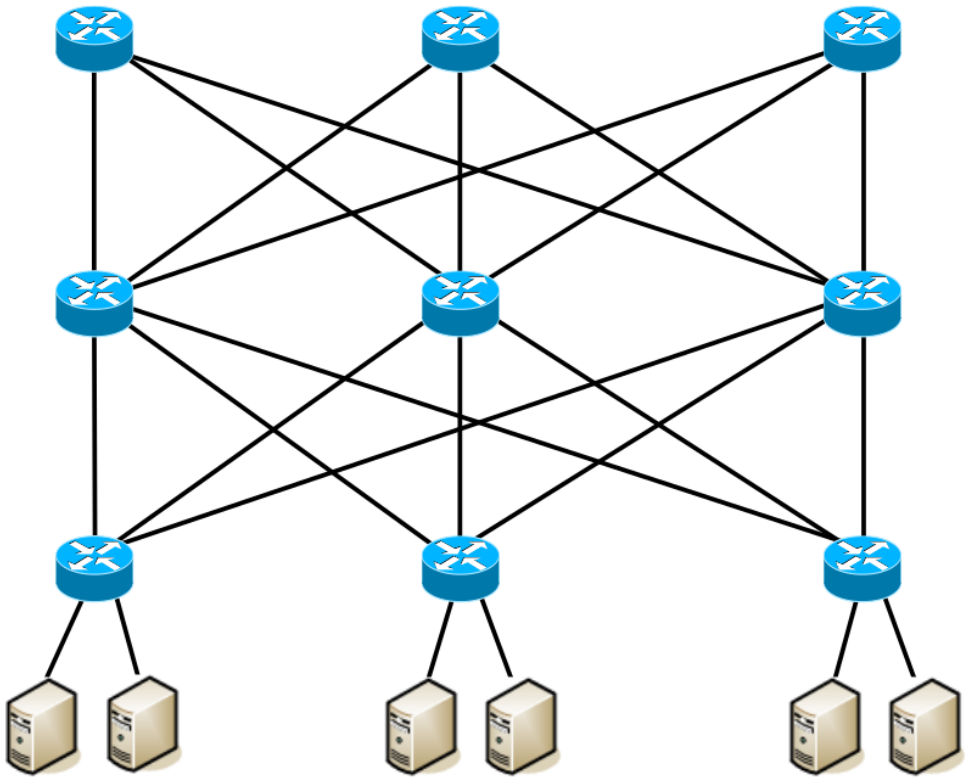


Figure 2.1: Clos Topology

### 2.5.4 Fat Trees Topologies

A type of network topology that managed to serve the above, goal is the Fat-Tree network topology. It came as an expansion of the classic CLOS topology. The Fat-Tree networks were proposed for the first time back in 1985, by Charles E. Leiserson [18]. This type of network forms a tree, and servers are connected to the lowest layer. The basic feature of a fat tree, that distinguishes it from other tree-like topologies, is that for any switch in a layer, the number of links going down to its children nodes is equal to the number of links going up to its parent nodes in the first layer above it. As a result, the links are getting "fatter", moving towards the top of the tree, and the switch or the switches in the root (named core switches) of the tree have the most links compared to any other switch below them.

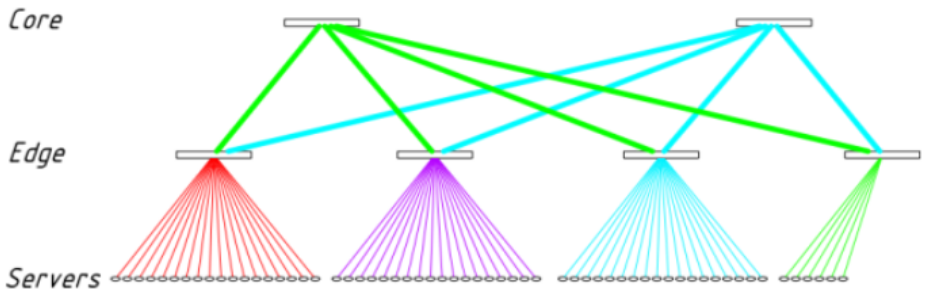



Figure 2.2: 2 layered Fat Tree Topology [6]

### 2.5.5 3-Layered Fat Tree Topology

#### 2.5.5.1 Introduction

As the need to have a huge amount of servers (endpoints), the complexity in the middle layers increases, so an extra layer of switches is added to the aggregation one. In the two-layered Fat Tree network topology, there are two layers the core switches layer and the edge or tor (Top of Rack) layer where the servers are attached. In the three-layered one, the edge layer has as a parent layer the aggregation layer, in which the switches in it are used for interconnection between multiple edge switches with multiple core switches. A part of the aggregation layer and part of the edge layer connected together, they are forming an abstract section of network topology called a pod. By using this extra layer the Fat Tree topology becomes more robust to failures, for example, if an aggregation or a core switch link fails there are alternative routes to connect a server to the rest of the network.

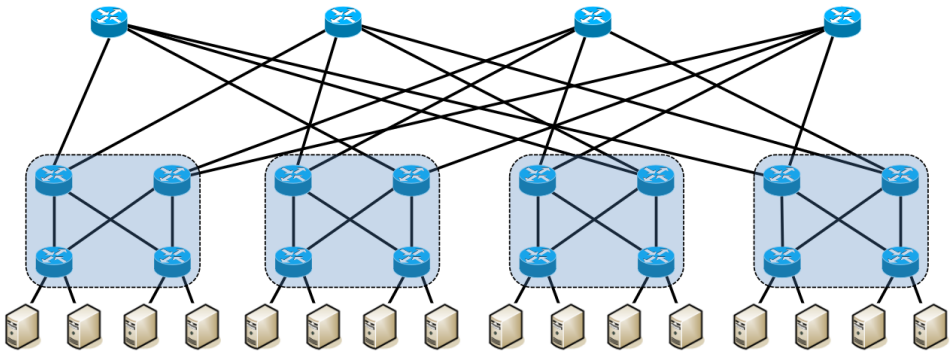


Figure 2.3: 3 layered Fat Tree Topology [7]

#### 2.5.5.2 Description

The size of a 3-layered Fat tree topology is relatively connected to the size of a switch component, in terms of the number of ports that it has. In this stage, is useful to notice that every switch or router in the topology is assumed to have the same size. So, given a switch's number of ports is $k$, the network will have $\frac{k^2}{4}$ core switches, followed by $k$ pods as children nodes. The pod is an abstraction of two more layers the aggregation and the edge one. Each core switch must have one link to each pod, this means that a core switch is connected with one aggregation switch of the pod, as there are multiple aggregation switches inside a single pod. By moving down to inside the pod abstraction, each node has $\frac{k}{2}$ aggregation switches and $\frac{k}{2}$ edge/ToR switches. As a result, an aggregation switch is connected to $\frac{k}{2}$ core and $\frac{k}{2}$ edge or ToR switches. The total number of aggregation switches sums up to $\frac{k^2}{2}$, and it is the same for ToR switches. At the final layer, there is where the servers are connected to the network, as it entails from the above, there are $\frac{k}{2}$ servers per ToR switch. A ToR switch has multiple servers connected to it and this group of servers is known as a rack, that's why the switch is called ToR (Top of the Rack). Finally, the result is to have a physical or virtual network with a total of $\frac{k^3}{4}$ servers. Last but not least, all these nodes are somehow connected, and this is made with the links, the total number of those links inside the network is $\frac{k^2}{2}(k + \frac{k}{2})$. The links normally have a specific bandwidth/capacity that is increasing as they move closer to the core switches, this is also referred to as oversubscription of the network usually a ratio of 4:1.

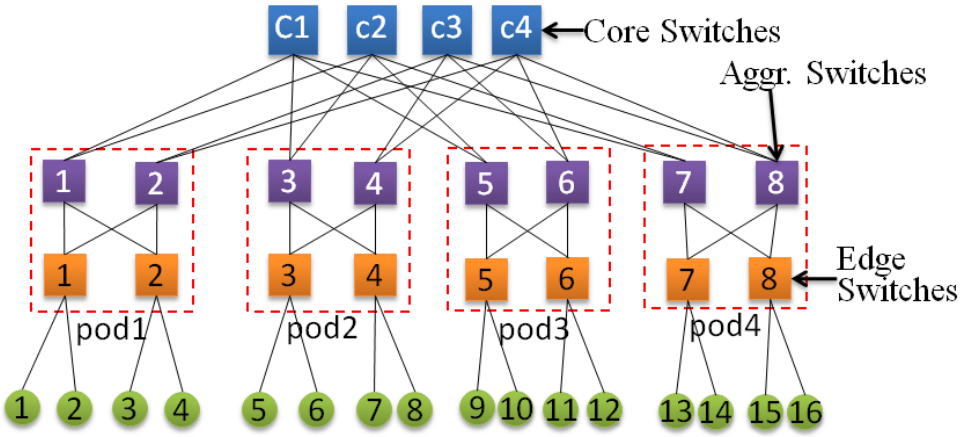



Figure 2.4: 3 layered Fat Tree Topology with labels [6]

## 2.6 Reinforcement Learning

### 2.6.1 Introduction

Reinforcement Learning [19] belongs to the field of Artificial Intelligence and is specifically a part of Machine Learning techniques. The basic concept of Reinforcement Learning assumes having an agent (AI software) that by taking some actions inside an environment tries to increase its rewards in order to achieve a goal. Nowadays, RL, and especially DRL has offered a new way of solving difficult problems and has made some important breakthroughs in a variety of domains. This is also expected to help with the VNE problem also, as there is a need for better performance in high-complexity environments. Overall the Reinforcement Learning methods are all about making decisions sequentially that will lead toward a specific goal, and that is exactly what the VNE problem requires. Furthermore, Reinforcement learning is trying to find the appropriate balance between, exploration, where the agent gathers more information that might lead it to better decisions in the future steps (action selection), and exploitation where the agent makes the most profitable decision given the current information.

### 2.6.2 Categories of Reinforcement Learning Algorithms

In general, there are two main categories, that the reinforcement learning methods are classified into, the first one is Model-free and the second one is Model-based. The key difference between the two of them is that Model-based methods involve learning an explicit model of the environment and its dynamics, including transition probabilities and all the reward functions, in order to train an agent to act according to the objectives of the problem. By proceeding further the Model-free methods are split into the value-based or value iteration and the policy-based. The Value-based methods in reinforcement learning aim to estimate the value or more precisely the quality of different state-action pairs. These algorithms use a value function to determine the optimal policy that an agent must follow toward a goal. One popular value-based algorithm is Q-learning, which iteratively updates a value function, known as the Q-function, to approximate the maximum expected rewards after an action. By using the Q-function, the agent can choose the action that maximizes its expected return. This is represented in the form of a table where each action-state pair has a value that is constantly updated as the agent moves closer to the final objective. An alternative is to use Deep Q-Networks (DQNs)[20, 21] to extend Q-learning by employing deep neural networks to approximate the Q-function, enabling reinforcement learning in high-dimensional and continuous state spaces, where the use of a large matrix would be inefficient. The Policy-based methods, on the other hand, try to directly optimize the policy of an agent, which is a

mapping from states to actions, without estimating the value function explicitly. In a nutshell, policy-based methods are used to find the best possible sequence of actions to be made by the agent, that maximizes the expected cumulative reward and leads to the final goal. The algorithm uses gradient ascent to iteratively update the policy parameters. By sampling actions from the current policy and estimating the gradient of the expected return, policy-based methods learn to improve the policy over time. In addition, they are able to handle larger and more complex action spaces. Regarding the Model-based reinforcement learning algorithms. These algorithms use the learned model to simulate interactions with the environment, enabling planning and decision-making based on simulated experiences. Overall a model-based reinforcement learning algorithm can be advantageous when direct interaction with the real environment is costly or time-consuming. The learned model can be used for predicting future states and rewards but also is used for generating sample trajectories for policy optimization. Model-based algorithms, such as I2A[22], or AlphaZero[23] combine planning and exploration to find optimal actions in a given environment.

### 2.6.3 Bellman Equation

In order to describe a Reinforcement Learning problem, there are some variables that need to be specified like, an environment inside where the actor-agent will operate, the Actions list which is a set of all possible actions that an agent/learner algorithm can take inside the environment, the States table which represents all the states that the agent may be in by taking an action. There is also a positive Reward when the agent reaches a goal and there is a negative Reward if the agent makes a mistake and reaches a non-wanted state. In addition to supporting the agent to reach the goal as soon as possible or with fewer actions, there is a $\gamma$ factor that discounts the reward in each step, this also allows the agent to choose actions in a better way. The above description refers to Value iteration, where the agent updates each state's reward value iteratively in order to extract a sequence of actions that lead to the goal. The value iteration is formulated by the following mathematical equation [24, 25]:

$$V(s) = \max_{a} \left(R(s,a) + \gamma V(s')\right) \tag{2.1}$$

where $s$ is the current state,$s'$ is the next state, $a$ is the action taken, $R$ represent the reward for the current action,state pair, the discount factor $\gamma$, $0 < \gamma < 1$.

### 2.6.4 Markov Decision Process

The above equation describes the value for each state if the environment in which the agent is moving/acting is deterministic, which means that every action occurs every time. But in a real-world

environment, there is also some kind of randomness, that makes the environment a non-deterministic one. So, if the agent chose an action to take there is a chance that this action will not happen/occur. For example, if the agent is a self-driving car and it wants to move forward (action: move forward) there is a 70% that the car will move forward without any problem, and there is a chance of 20% that another car is blocking its way and there is a 10% chance that the car stops working. As a result, the action selected might not lead to the preferred state. In order to model this randomness in Reinforcement Learning, Markov Processes [26] are used, based on Markov's property that every state is independent of the previous one. There for now the agent is making decisions based on some probabilities per action. This concept is formulated from the following equation, which is a variation of the previous one 2.1.

$$V(s) = \max_{a} \left( R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right) \tag{2.2}$$

### 2.6.5 Q-Learning

The Q-Learning is an `Off-policy` Reinforcement Learning method, that tries to find the best action given a state. The reason why it is considered an `Off-policy` RL algorithm is that the Q-learning function learns from actions that are outside the current policy, with other words it is taking random actions, and therefore a policy isn't necessary. The best action is defined by a value, named `q-value`, that shows the quality of an action. In simpler words, this means that the higher the `q`, the better the action. With Q-learning [27, 28, 20] instead of calculating the value of the current state that an agent is in, now the purpose is to expose the quality of an action that updates an agent's state and the purpose of the Q-Learning approach is to expose the best sequence of actions that leads to the final goal, where the maximum reward is given to the agent. The states and actions form a table that shows a corresponding q-value $Q(state, action) = q$, where $q \in R$. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} (P(s,a,s')V(s')) \tag{2.3}$$

The above equation 2.3 is transformed to only described by Q iteratively in the following form:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} \left( P(s,a,s') \max_{a'} Q(s',a') \right) \tag{2.4}$$

where $s$ is the symbol of current state, $a$ the current action, $s'$ is the next possible state, after action $a$ was occur, $a'$ is the action given the state $s'$ and $\gamma$ is a discount factor of the expected reward as the agent moves forward and choose actions.

### 2.6.6 Deep Reinforcement Learning

In general the term Deep is referred to the use of a Neural Network model, as it wide common in the machine learning field. The same thing happens into the Reinforcement Learning field, which is consists of a set of algorithms that are focusing onto function approximations. For example, in the previous section the Q-learning method uses matrices/tables to store a specific and deterministic states, actions and rewards sets. This approach is limited to handle low-dimension problems with discrete states and actions. So, with the use of Deep Reinforcement Learning instead of those tables there is a ANN, usually a kind of MLP [29, 30] (Multi-Level Perceptron) model, which takes as input a vector /tensor that represents the state of the environment and gives as output a set of actions or the q-values of these actions. To be more accurate the output is the Q value per action, showing the quality of the agent taking this specific action given the current state. After the NN model has made an approximation the agent uses another method in order to pick an action, usually with some greedy approach that includes a randomness or uses a probability distribution per action and picks the one with the highest probability to occur (see softmax [31] selection).

### 2.6.7 Deep Q-Learning

Deep Q-learning [21] is an alternative approach to implementing the Q-learning [20] algorithm using neural networks. The basic concept is to encode the state space into $R^n$ dimension in order to feed them into an Artificial Neural Network, as output you get $Q_i$ value of each action. In the Deep Q-learning process, there are 3 stages: the first one is the learning phase where the model is trained by updating its weights given a state as input and gets a vector of actions by using a loss function (Section: 2.7.4) based on output and Q-target set of actions, the second phase is about acting, where the agent uses the model to make decisions inside the environment and the last phase is the way an action is selected, this called action selection policy.

The Loss function can be described by the following equation,

$$L_\theta = E\left[r_t + \gamma * \max_\alpha Q_\theta(s_{t+1}, \alpha) - Q_\theta(s_t, \alpha_t)\right] \tag{2.5}$$

where $\theta$ represents the Neural Network's parameters, $r_t$ is the current reward, $\gamma$ is the discount factor $s$,$s_{t+1}$ are the current and next state variables according and $\alpha$ represent the action.

**Algorithm 1** DQN Algorithm with Experience Replay

*Initialize replay memeory D to capacity N*
*Initialize action − value function Q with random weights*
**for** *episode* $= 1, M$ **do**
    *Initialize sequence* $s_1 = x_1$&*preprocessed sequenced* $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        *With probability* $\epsilon$ *select random action, otherwise select* $a_t = max_a Q^*(\phi(s_t), a; \theta)$
        *Execute action* $a_t$ *in emulator & observe reward* $r_t$ *&image* $x_{t+1}$
        *Set* $s_{t+1} = s_t, a_t, x_{t+1}$ *& preprocess* $\phi_{t+1} = \phi(s_{t+1})$
        *Sample random minibatchoftransitions*$(\phi_t, a_t, r_t, \phi_{t+1})$ *trasition from D*
        *Set* $y_j = \begin{cases} r_j, & \text{for terminal state } \phi_{j+1} \\ r_j + \gamma * \max_{\alpha'} Q(\phi_{j+1}, \alpha'; \theta), & \text{for non-terminal} \phi_{j+1} \end{cases}$
        *Perform a gradient descent step on* $(y_i - Q(\phi_j, \alpha_j; \theta))^2$

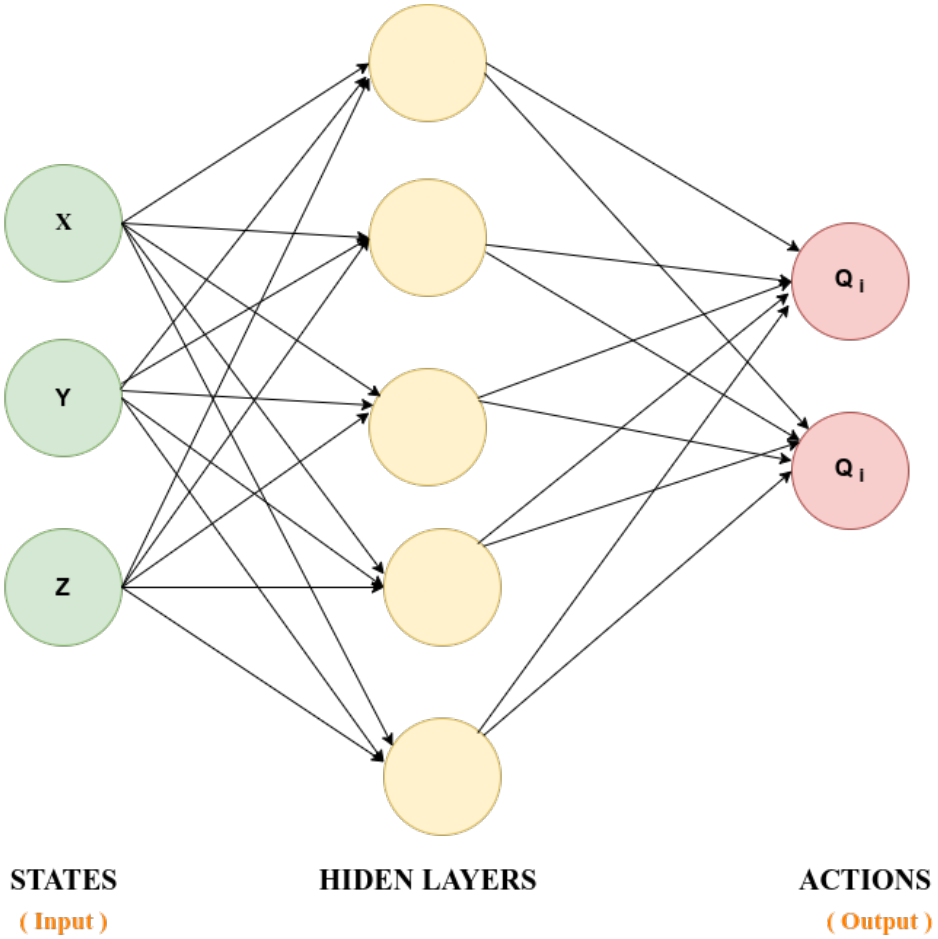



Figure 2.5: Example of Deep Q Neural Network

#### 2.6.7.1 Replay Memory

The concept of the replay buffer or experience memory is crucial for Deep Q Learning, as the agent acts inside the environment and explores the world, the old experiences are stored in a memory buffer. Old experiences refer to vectors that store the current state, action, reward, and the next state, action set. The agent interacts with the environment e.g. takes action, gets rewards, etc., for a number of episodes, completely at random at the beginning of the process, and stores these interactions' data as vectors in the memory. Then the agent uses this memory by selecting a batch of it, to feed and train the model upon these data, and it updates the memory with new experiences. The sampling of experiences can be done randomly or they can have a priority on which will be picked up each time. This sampling

approach of experiences makes the model more robust to over-fit and that prevents it from not acting properly in a new state.

#### 2.6.7.2 Selection Policy

As a part of the process, the output of the Neural Network model has to give the agent the ability to choose an action. For the purpose of taking action, there are a few options available to do so, like $\epsilon$-greedy,$\epsilon$-soft, and softmax.

- `ϵ-greedy:` A greedy approach to pick an action. An action is chosen with probability $\epsilon$, while the best action, the one with the highest Q-value, is chosen, with probability$1 - \epsilon \in [0, 1]$.
The main problem of this approach is that the choice is uniformly distributed so it considers that all actions are equal good.
- `ϵ-soft:` This is compliment method of the $\epsilon$-greedy. In general, this policy is any policy given a state $s$ that the probability of all actions is greater than some minimum value $\epsilon$. The soft decision of not picking an action.

$$\pi(\alpha|s) \geq \frac{\epsilon}{|A(s)|} \tag{2.6}$$

where $\forall \alpha \in A(s)$.

- `Softmax:` The output of Q-values is using a softmax function in order to transpose the Q-values into a probability distribution, so the agent in each states selects the action that is most probably to occur.

$$\sigma(x_i) = \frac{\exp^{x_i}}{\sum_{j=1}^{N} \exp^{x_i}} \tag{2.7}$$

This is the standard softmax function $\sigma$: $\mathbb{R}^{\mathbb{N}} \to \mathbb{R}^{\mathbb{N}}$, where $i = 1, \cdots N$ and $\mathbf{x} = (x_1 \cdots, x_N) \in \mathbb{R}^{\mathbb{N}}$.

All the above selection policies are trying to find the balance between exploration and exploitation, to create a policy for the agent that gets the best actions. For example, the $\epsilon$-greedy can be expanded to have an adaptive $\epsilon$, so at the start of the training the $\epsilon$ is high, so as the agent explores the environment, and as the iteration goes by, there is a decay to $\epsilon$ value, in a way that eventually it will exploit the optimal action in the end.

## 2.7 Artificial Neural Networks

### 2.7.1 Introduction

As part of this work, the focus is on using neural networks as the model for the Reinforcement Learning Agent. Neural networks are an essential building block of deep learning models, and they are inspired by the biological neural network of the human brain. Artificial Neural Networks (ANN) are mathematical models that are designed to mimic the structure and function of biological neurons in the brain. The basic building block of an ANN is the neuron, which receives input signals, processes them, and produces an output signal. ANNs are typically composed of layers of neurons, with each layer performing a specific task in the overall learning process.

The basic idea behind ANNs is to learn a function that maps input data to output data, similar to the way humans learn from experience. This makes ANNs particularly effective at tasks such as classification, regression, and pattern recognition. ANNs consist of a series of interconnected nodes, called artificial neurons or units, organized into layers. Each unit in a layer receives input from the previous layer, performs a simple computation, and outputs a signal to the next layer. The strength of these connections between units can be adjusted during training to enable the network to learn to perform a specific task. Additionally, ANNs can learn to recognize complex patterns and relationships that might be difficult or impossible for humans to discern.

There are many different types of ANN architectures, each with its own strengths and weaknesses. Some of the most common types of architectures include feedforward networks [29], recurrent networks [32], and convolutional networks [33, 34]. Each of these architectures is suited to different types of tasks and data, and selecting the appropriate architecture is an important step in designing an effective neural network.

The development of ANNs has been a major focus of research in the field of machine learning for several decades, owing to their ability to learn complex patterns and relationships in data, and their success in a wide range of applications, such as computer vision [34], natural language processing [35, 36], and speech recognition [37, 34]. However, the design and training of ANNs can be a complex and challenging process, requiring a deep understanding of both the mathematics and the underlying data. Furthermore, the effectiveness and the success of the model are heavily dependent on the quality and quantity of the training data, the architecture of the network, and the hyperparameters used during the learning process.

Overall, ANNs are a powerful tool for developing intelligent systems that can learn from data, adapt to changes in the environment, and make informed decisions. And as for the role of ANNs in the current

master thesis is to develop a model that can learn and adapt to changes in the environment and make informed decisions based on the learned experiences.

### 2.7.2 Dense Layer

The simplest kind of an Artificial Neural Network (ANN) is a Multilevel Perceptron Model [29], which only contains a couple of fully connected dense layers. Dense layers, also known as fully connected layers, are the most commonly used layers in deep learning models, including Multilevel Perceptron Models. In general, a dense layer is a layer where each neuron is connected to every neuron in the previous layer. The output of each neuron is a weighted sum of the inputs, plus a bias term, which is passed through an activation function. One important aspect of the dense layer is the initialization of the weights and biases (Section 2.7.5). The initialization of the weights and biases can affect the overall performance of the model during training. The weights and biases in the dense layer are learned during training using an optimization algorithm to minimize a loss function. The activation function introduces non-linearity into the model, allowing it to learn complex relationships between inputs and outputs. The number of neurons in a dense layer is a hyperparameter that needs to be tuned during model selection. Is good to mention that the last layer in a neural network model is usually a dense layer where the number of neurons in this layer is typically equal to the number of classes in a classification task the number of features in a regression task, or the number of actions in a Reinforcement Learning task. As for the example of Multilevel Perceptron Models [29] typically contain one or more dense layers, followed by one or more non-linear activation functions, and an output layer. The output layer may have a different number of neurons and a different activation function than the hidden layers. Overall, dense layers are essential building blocks of many deep learning models, including Multilevel Perceptron Models. Their purpose is to enable the model to learn complex relationships between inputs and outputs and is typically followed by an activation function (Section 2.7.3) and an output layer.

### 2.7.3 Activation Functions

Activation functions [31, 30, 34] are used to obtain the output of a neural network layer's node. It actually maps the node's result values in [0, 1] or [-1, +1] depending on the function. There are two types of activation functions, Linear and Non-Linear ones. For example, a linear function would be something like $y = x$. A non-linear example is an exponential or logarithmic function.

In neural networks, we mostly use non-linear functions, one of these types is the ***Sigmoid***(Figure 2.6). But we mostly use a variant of the ***Sigmoid***, the ***ReLU*** –*Rectify Linear Unit*(Figure 2.7)–, which is half rectified in $x \in [-\infty, 0]$. The function and its directive are both monotonic (increasing).

•**The Sigmoid function:**

$$\sigma(x) = \frac{1}{1 + \exp^{-x}} \tag{2.8}$$

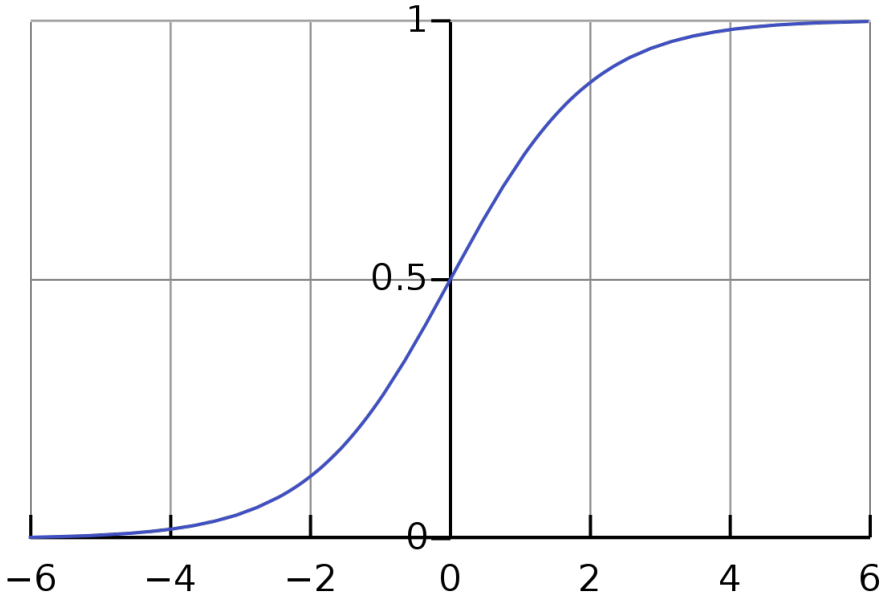



Figure 2.6: Sigmoid plot

•**The ReLU function:**

$$R(z) = \max(0, x) \tag{2.9}$$

or otherwise we can say:

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \quad (2.10) \\ x, & \text{if } x \geq 0 \quad (2.11) \end{cases}$$

The ***ReLU***'s drawback is that all negative values become zero immediately, which decreases the model's ability to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turn affects the resulting graph by not mapping the negative values appropriately. It is also known as the zero-stacked neurons problem and it's more likely to occur when we have a high learning rate or there is a large negative bias. This problem means that many neurons will not add useful information to our network, so a solution to that is adding a dropout parameter, where neurons are disabled with a random probability in order to retrieve information from different neurons each time and improve the fitting of the model.

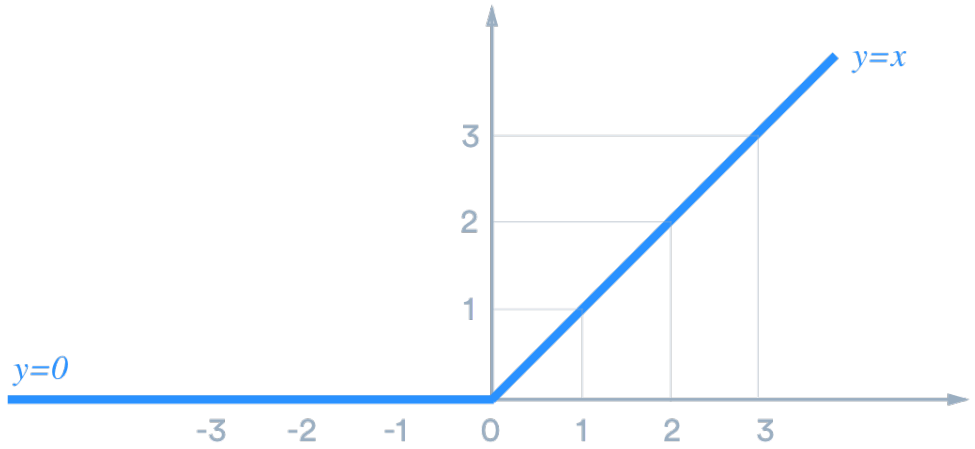



Figure 2.7: ReLU plot

Engineers have come up with some other activation functions to avoid this problem, like ***LReLU***, ***PReLU***, and ***ELU***. Which are shown in Figures 2.8, 2.9 along with their equations:

- **Leaky–ReLU equation:**

$$f(x) = \begin{cases} 0.01 \times x, & \text{if } x < 0 \quad (2.12) \\ x, & \text{if } x \geq 0 \quad (2.13) \end{cases}$$

if we generalize the equation of LReLU it comes to PReLU,

- **Parametric–PReLU equation:**

$$f(x, a) = \begin{cases} a \times x, & \text{if } x < 0 \quad (2.14) \\ x, & \text{if } x \geq 0 \quad (2.15) \end{cases}$$

where $a$ is added to set of the trainable variables of the model.

- **Exponential Linear Unit (ELU) equation:**

$$f(x, a) = \begin{cases} a \times (\exp^x - 1), & \text{if } x \leq 0 \quad (2.16) \\ x, & \text{if } x > 0 \quad (2.17) \end{cases}$$

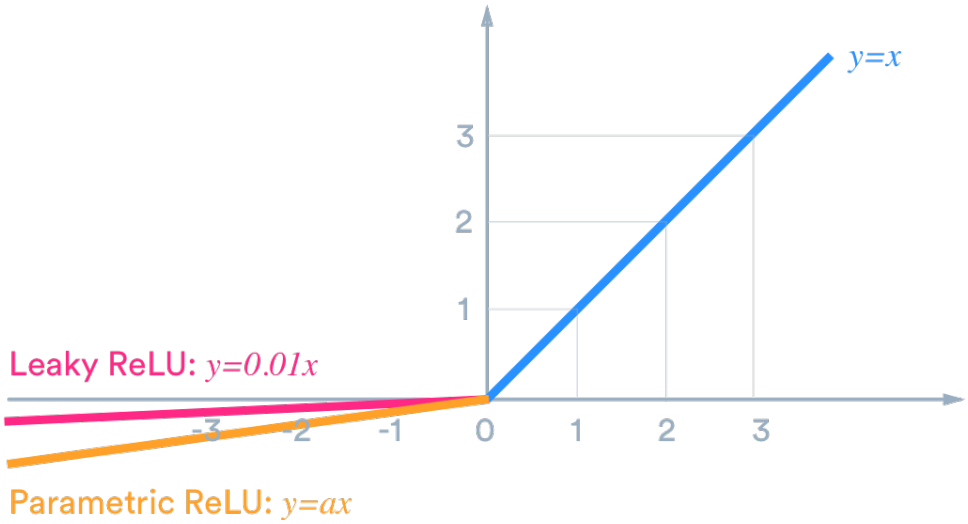



Figure 2.8: Leaky and Parametric–ReLU plot

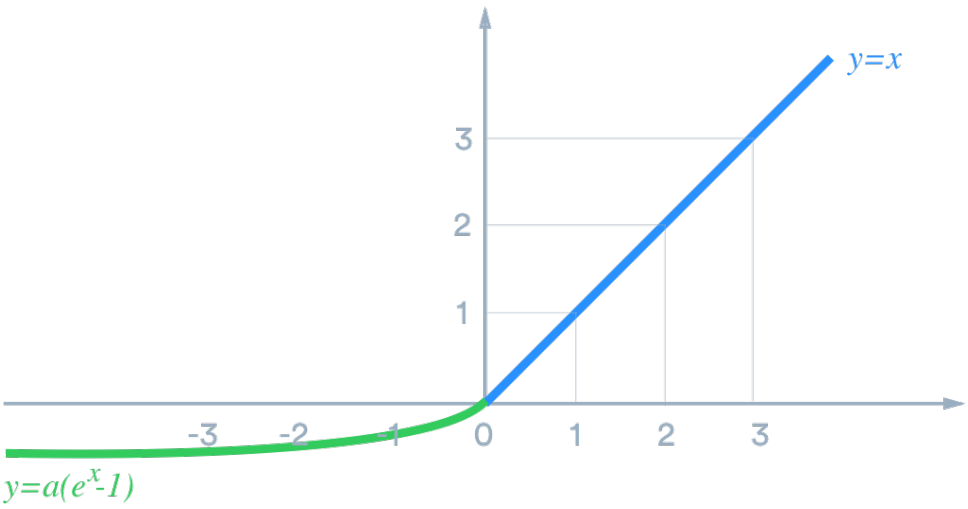



Figure 2.9: ElU plot

Another well-known activation function is the ***Softmax***. ***Softmax*** is a function that takes as argument a vector of **N** real numbers and converts it into a probability distribution consisting of **N** probabilities proportional to the exponentials of the input numbers (Figure 2.10). This method actually normalizes the input between [0,1] by respecting the main rule of probability theory that $\sum_i^N P(N_i) = 1$. In neural networks *Softmax* is used in most cases after the last layer –logits– to map the non-normized output to a probability distribution over the predicted logits or classes. Also, Softmax is used in the calculation of the loss in such cases by using softmax–cross-entropy loss.

- ***Softmax Equation :***

$$\sigma(x_i) = \frac{\exp^{x_i}}{\sum_{j=1}^{N} \exp^{x_i}} \tag{2.18}$$

this is the standard softmax function $\sigma$: $\mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{N}}$, where $i = 1, \cdots N$ and $\mathbf{x} = (x_1 \cdots, x_N) \in \mathbb{R}^{\mathbb{N}}$ .

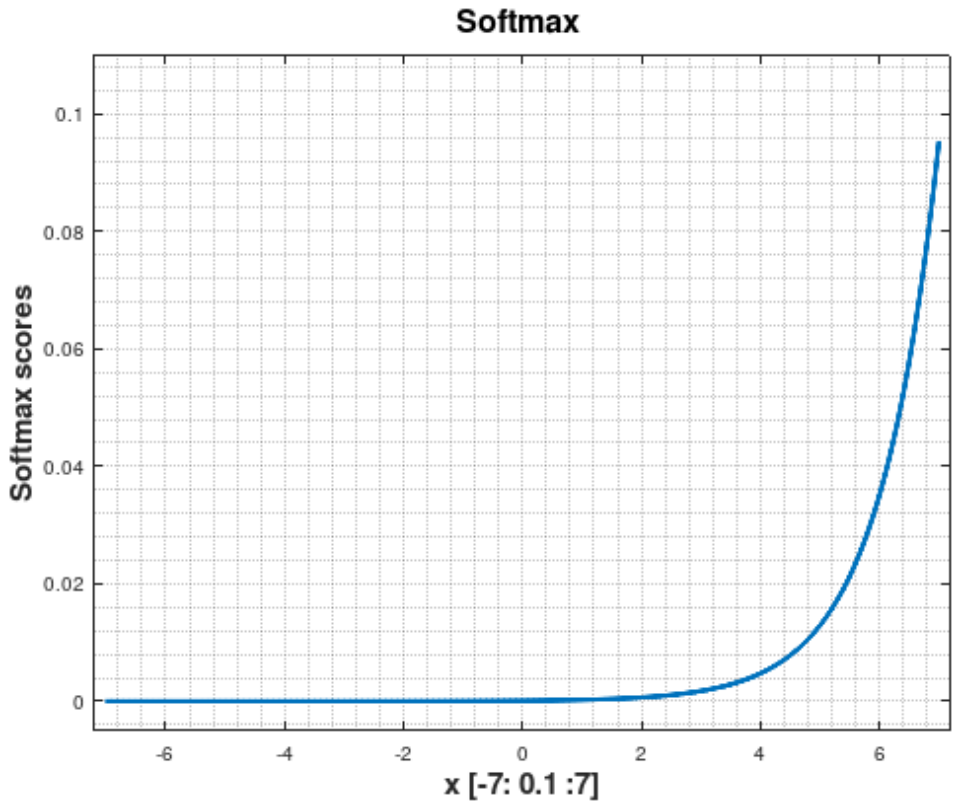



Figure 2.10: Softmax example

### 2.7.4 Loss Function

The most important part of a Neural Network model is the appropriate selection of a loss function. A loss function is a mathematical expression that tries to calculate the error between the predicted output of the model given the input, in such a way that the model updates its weights inside each node of all layers, to get closer to the true values. The choice of a loss function depends on the type of machine learning problem being solved. Based on the machine learning field if the task is a problem of Supervised learning, where the true labels are provided, the model calculates the error/loss between the prediction and the ground truth ones, that a user feeds the model. In a nutshell, it is responsible for measuring the difference between the predicted output of the model and the true values. So, in Supervised learning tasks, the model learns by minimizing the loss function, which measures the discrepancy between the predicted and true values. Some common loss functions used in supervised learning tasks include mean squared error (MSE) [38, 39], cross-entropy [40], and binary cross-entropy [41]. But in the case of Unsupervised Learning, where Reinforcement Learning is laid, on the other hand, there are not any labels, instead, the model trains itself through a system of penalties or rewards that are given for each action, and help it to move closer to the final objective. This means that the model trains itself by maximizing or minimizing some objective function that is indicative of performance, such as reconstruction error or contrastive loss or maximizing the profits/score. Unsupervised learning has been used extensively in several machine learning applications, such as clustering, dimensionality reduction, anomaly detection, or any other problem where the classic methods failed to do so, either because of a lack of labeled datasets or because of the complexity of the task. The choice of an appropriate loss function for reinforcement learning is crucial for the success of the model. The loss functions used in reinforcement learning include mean squared error, policy gradients, and Q-learning.

For the VNE problem, which is the focus of this work, the Double Deep Q-Network (DDQN) model will be used as a reinforcement learning agent. The objective of the DDQN model is to learn an optimal policy that maximizes the expected cumulative reward obtained by placing virtual network functions (VNFs) in a virtual network. The choice of an appropriate loss function for the DDQN model is critical for the success of the VNE problem.

#### 2.7.4.1 Mean Square Error

The mean squared error (MSE) is a common loss function used in regression problems in machine learning [39]. It is used to measure the average squared difference between the predicted and true values of a model's output. Mathematically, it is calculated by taking the average of the squared differences between the predicted values and the true values.

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(y_i - y_{pred_i})^2 \tag{2.19}$$

where $N$ is the number of data points $y_i$ are the observed values and $y_{pred_i}$ are the predicted ones.

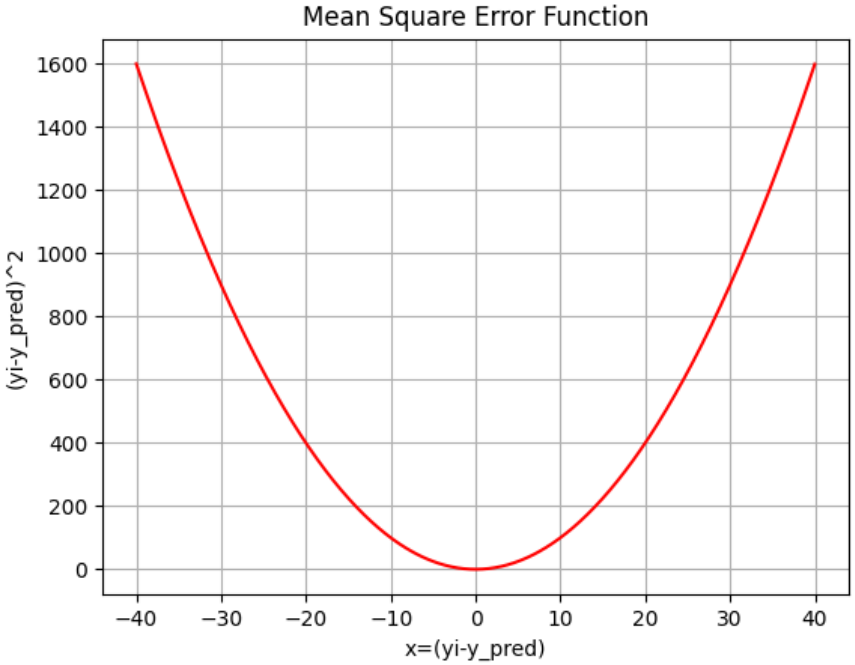



Figure 2.11: Mean Square Error plot

The MSE loss function has several desirable properties that make it a popular choice for regression problems. First of all is continuous and differentiable, which means that it can be easily optimized using gradient-based methods, like neural networks which use the backpropagation technique. Additionally, it gives higher weight to large errors, which can be useful in situations where large errors are more significant than small errors. This has also a significant drawback, and that is MSE is very sensitive to outliers. If the data that feeds the model are not cleaned or they have missing values or in general the quality of them is not as it should be this definitely will affect the final model and its performance.

In general MSE loss has been widely used in a variety of regression-based applications, including finance, healthcare, and transportation [42, 43]. For example, in financial forecasting, MSE can be used to measure the accuracy of stock price predictions. In healthcare, it can be used to evaluate the

performance of medical diagnostic models. In transportation, it can be used to evaluate the accuracy of traffic flow models.

The MAE can be expressed mathematically as:

$$MAE = \frac{1}{N}\sum_{i=0}^{N}|y_i - y_{pred_i}| \tag{2.20}$$

where $y_i$ is the actual value, $y_{pred_i}$ is the predicted value, and $N$ is the number of samples in the dataset.

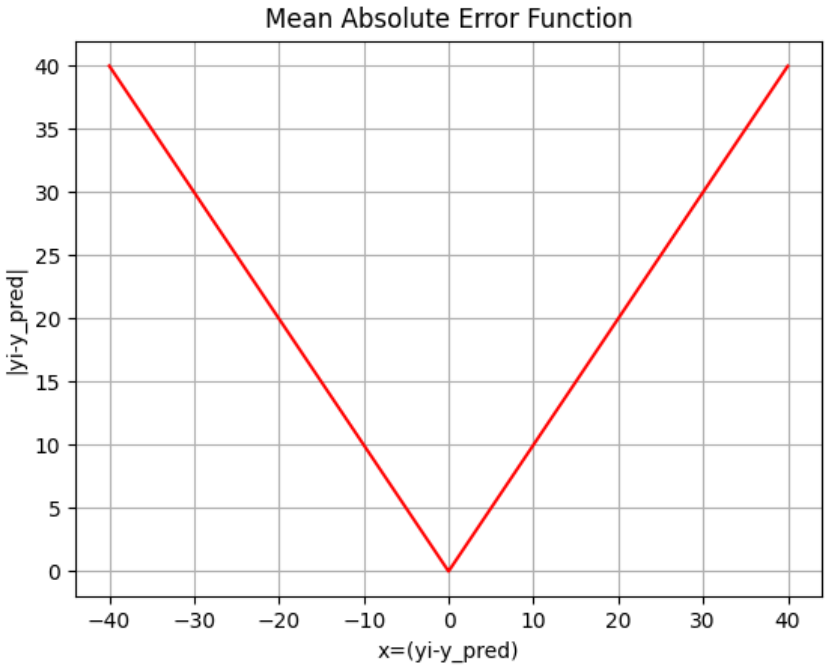



Figure 2.12: Mean Absolute Error plot

The MAE is often used in situations where outliers can have a large impact on the performance of the model. This is because the MAE is less sensitive to outliers than the MSE, since the absolute value operation makes the difference between the predicted and actual values insensitive to the direction of the difference.

One potential downside of the MAE is that it treats all errors equally, whereas the MSE places a higher weight on larger errors. As a result, the MAE may not be the best choice if you want to minimize the impact of large errors on your model's performance.

#### 2.7.4.2 Huber Loss

The Huber loss [44] is a popular choice of loss function in machine learning due to its ability to provide a balance between the Mean Squared Error (MSE) and Mean Absolute Error (MAE) losses. The Huber loss is robust to outliers in the data, meaning it is less affected by data points that are far from the majority of the data. It is a smooth and continuous function that behaves like the MSE loss for small errors and like the MAE loss for larger errors.

The Huber loss function is commonly used in regression problems where the goal is to predict a continuous value output. The Huber loss function is similar to other loss functions such as the MSE and MAE loss functions, but it provides a trade-off between the two. In situations where the dataset may contain outliers, the Huber loss function can be particularly useful as it can downweight the impact of these outliers on the overall loss. The Huber loss function is particularly useful when modeling data from real-world situations where the noise levels may be higher or the dataset may contain errors.

The Huber loss function is a loss function used in regression tasks that is less sensitive to outliers than the mean squared error (MSE) loss function. It is defined as a combination of MSE and MAE loss functions and is typically used when the data contains outliers or noise. The Huber loss function can be written as:

$$L_\delta(\alpha) = \begin{cases} \frac{1}{2}\alpha^2, & \text{if } |\alpha| \leq \delta. \\ \delta(|\alpha| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases} \tag{2.21}$$

If $\alpha = y - f(x)$ the above equation is expanded into this form,

$$L_\delta(y, y_{pred}) = \begin{cases} \frac{1}{2}(y_i - y_{pred})^2, & \text{if } |y - f(x)| \leq \delta. \\ \delta(|y - y_{pred}| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases} \tag{2.22}$$

where $y$ is the true value, $y_{pred}$ is the predicted value, and $\delta$ is a hyperparameter that determines the threshold at which the function switches from quadratic to linear. When the absolute error $|y - y_{pred}|$ is less than or equal to delta, the function is quadratic, similar to MSE. When the absolute error is greater than the delta, the function is linear, similar to MAE. This equation essentially can be described as: for loss values less than the $\delta$, use the MSE formula. In the case that loss values are greater than $\delta$, use the MAE. Using the MAE for larger loss values mitigates the weight that is put on outliers so that get a well-rounded model. At the same time, the MSE is used for the smaller loss values to maintain a quadratic function near the center. Overall, this method manages effectively to combine the best of both worlds from the two loss functions.

The advantage of using the Huber loss function over MSE is that it is less sensitive to outliers. Outliers can cause the weights of the model to be updated in such a way that they overfit the outlier


Figure 2.13: Huber Loss plot

data, leading to poor generalization performance. By using the Huber loss function, the effect of outliers is reduced and the model can learn to fit the majority of the data.

### 2.7.5 Weights Initializers

Weight initialization is a crucial step in the training process of a neural network model. It involves setting the initial values of the weights in each layer of the network to appropriate values that can help the network avoid getting stuck in a sub-optimal solution and improve the overall convergence towards minimizing the loss. Weight initialization can have a significant impact on the performance of the network and can play a crucial role in reducing the training time and improving accuracy. This section provides an overview of weight initialization approaches in machine learning models.

There are several weight initialization techniques that can be used in neural networks. The choice of initialization method is dependent on the network architecture, the activation functions used, and the nature of the problem being solved. The most common method used in deep learning is random weight initialization. With random initialization, the weights of the network are set to random values drawn usually from a uniform or a normal distribution. While this method is simple and easy to implement, it can lead to slow convergence and poor performance in deeper networks. In recent years, researchers have proposed many weight initialization methods to address the limitations of random initialization. These methods are designed to set the weights in a way that can speed up the convergence rate, improve the accuracy of the model, and prevent overfitting. The choice of initialization method can also depend on the optimization algorithm used to train the network, as some methods are better suited for certain optimization techniques.

In the following section, some of the most commonly used weight initialization methods in neural networks will be discussed such as He initialization (Subsection 2.7.5.1) and Glorot (Subsection 2.7.5.2) initialization.

#### 2.7.5.1 He Weight Initialization

The He initialization is a weight initialization technique that is widely used in deep learning networks. This method was introduced in 2015 by He [45] and is an improvement over previous techniques, such as random initialization, that were not optimized for use in deep neural networks. The main goal of He initialization is to prevent the vanishing and exploding gradients problems that can occur during the training phase in deep neural networks.

In general, the He initialization method works by setting the weights of a layer to random values drawn from a Gaussian distribution with a mean of 0 and a variance of $\frac{2}{n}$, where $n$ is the number of inputs to the layer. This approach ensures that the variance of the outputs of each layer is roughly the same as the variance of the inputs. The importance of the above initialization lies in the need that the gradients in a neural network are proportional to the variance of the inputs, and if the variance changes

significantly from layer to layer, the gradients can become too small or too large, which can cause the network to learn slowly or not at all or stack at sub-optimal solutions.

In practice, He initialization has been shown to work well in a wide range of deep learning models, including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and other types of architectures that use ReLu activation (Section 2.7.3) functions. It has also been found to be more effective than other methods, such as Glorot (also known as Xavier) initialization, which was designed to be used with sigmoid activation functions (Section 2.7.3). However, the choice of initialization method should be carefully considered for each specific model, and there are other methods that can be used in conjunction with He initialization to further improve the performance of the network. It is common that some layers may have different activation functions from others, even in the same model. For example, it's common in a CNN classifier where all the convolutional layers have a ReLu activation function, and the last Dense layer which makes the classification may have a sigmoid or a softmax activation function. The He initialization method [45, 46] as is implemented and described by the TensorFlow documentation is split into to sub-methods the He normal, which draws samples from a truncated normal distribution centered on 0 with $stddev = \sqrt{\frac{2}{fan_{in}}}$. And the He uniform where the weights are sampling values from a uniform distribution within [-limit, limit], and the limit is given by this formula $limit = \sqrt{\frac{6}{fan_{in}}}$, where $fan_{in}$ is the number of input units in the weight tensor, for both methods.

#### 2.7.5.2 Glorot Weight Initialization

Glorot initialization is a commonly used method for weight initialization in neural networks. It was introduced by Xavier Glorot and Yoshua Bengio in 2010 [47] as a way to address the vanishing and exploding gradient problem that can occur during training. The idea behind Glorot initialization is to set the initial weights to values that take into account the number of input and output connections for a given layer, as well as the activation function being used.

The Glorot initialization method involves randomly initializing the weights with values drawn from a uniform or normal distribution that is centered around zero. However, the range of the distribution is scaled by a factor that depends on the number of inputs and outputs for the layer. Specifically, the weights are initialized to values in the range [-r, r], where $r$ is calculated as the square root of 6 divided by the sum of the number of inputs and outputs. This range is chosen to ensure that the activations of the neurons in the layer are not too small or too large, which can help to prevent vanishing or exploding gradients during training.

Glorot initialization is a popular choice for weight initialization in neural networks due to its simplicity and effectiveness. It has been shown to improve the performance of deep neural networks compared to

other initialization methods, particularly in networks with large numbers of layers. However, it may not always be the best choice for every type of network or problem, and other initialization methods such as He initialization may be more appropriate in some cases. Inside the the official Tensorflow documentation there are implemented two versions of the above technique one mentioned as Glorot normal, which draws samples from a truncated normal distribution centered on 0 with $stddev = \sqrt{\frac{2}{fan_{in}+fan_{out}}}$. The other is the Glorot uniform where draws samples from a uniform distribution within [-limit, limit], where $limit = \sqrt{\frac{6}{fan_{in}+fan_{out}}}$, where in both cases the $fan_{in}$ is the number of input units in the weight tensor and $fan_{out}$ is the number of output units.

#### 2.7.5.3 LeCun Weight Initialization

LeCun weight initialization method, also known as LeCun initialization, is a weight initialization technique commonly used in deep neural networks. This method was introduced by Yann LeCun [48, 46]. The aim of this technique is to provide an efficient initialization of the weights in a neural network by taking into account the number of input and output connections of each layer. Following a similar logic as the previous two methods. The LeCun initialization method considers the distribution of the inputs to each neuron and the non-linearity of the activation function used in each layer. In this method, the weights are initialized using a normal distribution with a mean of zero and a standard deviation that is determined by the size of the input and output connections of each layer. This means that the weights are scaled based on the number of incoming and outgoing connections, which helps to preserve the variance of the signal throughout the network. For example, as described from the Tensorflow documentation [46], there is the LeCun Normal method, which takes samples from a truncated normal distribution centered on 0 with $stddev = \sqrt{\frac{1}{fan_{in}}}$. There is the LeCun Uniform method which samples from a uniform distribution within [-limit, limit], $limit = \sqrt{\frac{3}{fan_{in}}}$. The $fan_{in}$ is the number of input units in the weight tensor.

The LeCun initialization method also takes into account the non-linear activation functions used in each layer. The idea is to ensure that the variance of the output of each layer is equal to the variance of its input. This is done by scaling the weights with a factor that depends on the type of activation function used. For instance, the factor for the hyperbolic tangent (tanh) activation function is approximately 1.25, while for the Rectified Linear Unit (ReLU) activation function it is approximately 2.

One of the main advantages of the LeCun initialization method is that it helps to avoid the vanishing or exploding gradient problem. This problem arises when the gradients in a deep neural network become too small or too large, which can lead to slow convergence or divergence during training. By initializing

the weights in each layer based on the input and output connections and the activation function used, the LeCun method ensures that the gradients remain within a reasonable range throughout the network.

In summary, the LeCun weight initialization method is a powerful technique that can help to improve the performance and training time of deep neural networks. By taking into account the number of input and output connections and the non-linear activation functions used in each layer, this method provides an efficient initialization of the weights that can help to avoid the vanishing or exploding gradient problem and improve the overall convergence of the network.

# Chapter 3

# Problem Statement

## 3.1 Embedding of Virtual Network Functions

The embedding of VNFs is the problem of finding the optimal fitted virtual node, inside a virtualized network represented by a Substrate Network (SN), where a VNF can be placed [14, 12, 15, 49]. In the case of multiple sequential VNFs that are parts of a network service, further from node placement, there are links that must be mapped inside SN; so, there is another step that is similar to the path-finding problem, where usually the optimal shortest path is selected. The VNFs are served by the network operator in the form of Virtual Network Requests (VNRs). In the real world, a VNE method has to handle the VNRs upon arrival rather than serving them at once after arrival (off-line). With the on-line embedding decisions, the ISP aims to maximize its long-term revenue. The Network Service Embedding overall is an NP-hard [15, 50] problem. With this master thesis, an alternative method of NSE is implemented and tested in order to compare it with the basic heuristic or linear programming methods. As the demands for more capacity or better user experience over a network infrastructure have been increasing over the last few years, the need for an efficient, faster, and more robust change approach is necessary. With the use of virtualized networks and the ability to use any commodity server as a physical node, which can simulate a large variety of multiple network functions, classical Linear programming methods are getting too difficult to maintain and respond to more complex requirements. This is opening the path to machine learning approaches that are easier to expand for higher-dimension constraint problems.

The goal of this work is to create another proof of concept to point out all the benefits that a Reinforcement Learning method has to offer to a difficult and time-demanding problem. More precisely, this work tries to quantify the improvements in time and complexity that a Reinforcement Learning(RL) method has instead of a simple heuristic or Integer Linear Programming (ILP) methods that were

proposed in previous publications (see [12, 15]).

## 3.2 Related Work

The present work was inspired by previously published research on network service embedding titled "Network Service Embedding Across Multiple Resource Dimensions" [12]. In that paper, the main problem discussed is the classic Network Service Embedding over multiple resource constraints. The resources that a VNF has, are CPU and memory (2-D) there is also the link bandwidth between two VNF nodes. The goal is to compare a MILP [51] method and a simple heuristic baseline algorithm with a more sophisticated heuristic method that was developed to solve NSE in a more efficient way for multiple dimensions. The higher dimensionality of the problem refers to the extra resource demands of a VNF sequence, resources demands like CPU, memory, and bandwidth. So, the Mixed Integer Linear Program has the objective of minimizing the Equation 3.1, and the formulation is described with the following mathematical expressions:

$$\sum_{u \in V} z_u + \frac{1}{\sum_{(i,j) \in E_F} d^{ij}} \sum_{(u,v) \in E} \sum_{(i,j) \in E_F} f_{uv}^{ij} \tag{3.1}$$

The 3.1 must subject to these constraints:

$$\sum_{u \in V} x_u^j = 1 \, \forall i \in V_F \tag{3.2}$$

$$\sum_{v \in V_s} (f_{uv}^{ij} - f_{vu}^{ij}) = d^{ij}(x_u^i - x_u^j) \tag{3.3}$$

Where $(u \neq v)$, $(i \neq j, \forall (i,j) \in E_F, \forall u \in V)$.

$$\sum_{i \in V_F} d_i^k x_u^i \leq r_u^k \cdot z_u \; \forall u \in V, \forall k \in \{1,2\} \tag{3.4}$$

$$\sum_{i,j \in E_F} f_{uv}^{ij} \leq r_{uv}, \; \forall (u,v) \in E \tag{3.5}$$

$$x_u^i, z_u \in \{0,1\}, \; \forall i \in V_F, \forall u \in V \tag{3.6}$$

$$f_{uv}^{ij} \geq 0, \; \forall (u,v) \in E, \forall (i,j) \in E_F \tag{3.7}$$

| Symbol | Description |
|---|---|
| V | the set of physical nodes within the substrate topology |
| $V_F$ | the set of virtual nodes comprising a network service |
| E | the set of physical edges within the substrate topology |
| $E_F$ | the set of virtual edges between virtual nodes |
| $d_i^k$ | demand of virtual node `i` on resource `k` |
| $d^{ij}$ | bandwidth demand of edge (i,j) in *Mbps* |
| $r_u^k$ | residual capacity of physical node `u` on resource `k` |
| $r_{uv}$ | residual capacity of physical edge (u,v) *Mbps* |
| $x_u^i$ | assignment of virtual node $i$ to physical node `u` |
| $f_{uv}^{ij}$ | amount of bandwidth assisgned to link (u,v) for the virtual edge (i,j) in `Mbps` |
| $z_u$ | usage indicator of server u |
| $s_u^i$ | suitability value of mapping VNF i to server u |

Table 3.1: Symbol Explanation Table[12]

Regarding the baseline heuristic, that was used in this paper, it sorts the racks of the substrate network by using the server's CPU load. Then this baseline is expanded in order to be used for an n-dimension problem, where the resource is represented as 2D vectors and the constraints formulation uses the Manhattan distance in order to check if there is any violation of the constraints. In the case a Virtual Node can be placed, then there is a priority to keep, on where to place this node. The priority has to do with putting a VN first in the same rack on a different or in the same server and then if it can't be placed there, the algorithm tries for a server on a different rack. This process simply takes into account the intra-rack and inter-rack traffic, so as to better manage the substrate network resources and balance the load.

In order to think a level further the [15] was used as a guideline to design the Reinforcement Learning approach presented in these pages. In [15], the problem of NSE is described and a basic formulation is presented that shows how to handle the mapping of a VNF service. More precisely they define the state of the network, as a multi-dimension vector/ array,

$$State = \begin{pmatrix} S\_CPU\_MAX \\ S\_BW\_MAX \\ S\_CPU\_Free \\ S\_BW\_Free \\ Current\_Embedding \\ V\_CPU\_Request \\ V\_BW\_Request \\ Pending\_V\_Nodes \end{pmatrix} \tag{3.8}$$

The above vector is explained by the Table,

| State Representation Feature | Description |
|---|---|
| S_CPU_Max | The maximum of CPU resources over all SN nodes |
| S_BW_Max | The max bandwidth of each substrate node. We define the bandwidth of a node as the sum of all links' bandwidth that directly link to this node. |
| S_CPU_Free | The amount of CPU resources that are currently free on every substrate node. |
| S_BW_Free | The bandwidth resources that are yet to be allocated on all substrate nodes. |
| Current_Embedding | The (partial) embedding result of the current VNR. Each substrate node is set to 1 if it host a virtual node in the current VNR and 0 otherwise. This feature works as a mask to prevent virtual nodes in the same VNR from sharing one substrate node, as most previous works did. |
| V_CPU_Request | The number of virtual CPUs the current virtual needs to fulfil its requirements. |
| V_BW_Request | The total bandwidth the current virtual node demands according to the current VNR. |
| Pending_V_Nodes | The number of unallocated virtual nodes in the current VNR. |

Table 3.2: State Vector Features Explanation Table[15]

As for the mapping steps the authors of [15] split it, into two separate sub-problems one is to place a VNF node on the substrate network if the CPU constraints are valid, and then to map the linking between two VNF nodes with those of virtual network's nodes, or with other words to find the optimal path between all possible layers/nodes of the virtual network, that also satisfy the bandwidth request of the VNF nodes. There is also this work [52], where the authors are focusing on the 'Online' procedure of placing VNFs, which is closer to the real world scenarios. They try to fully automating Virtual Network Embedding by using neural networks and reconstructing the substrate network state into a 2D array that contains the CPU and bandwidth features of all nodes. This array is handled as an image vector and feed into a Convolutional Neural Network [53]. At the end there is also an extra logic in order to

reduce the action space, so as the model to converge faster and give a feasible solution at the end. Last but not least, as for the evaluation or simulation process a lot of the information was extracted from [54, 10, 55, 56] and it will be discussed further at the Chapter 5.

| Author | Methology | Features | Spatial Features | Features Extraction | DRL usage | Problem Solving Time |
|---|---|---|---|---|---|---|
| Chowdhury et al [57] | MIP modelling and LP relaxation | CPU & bandwidth | Yes | No | No | Massive with larger networks |
| Shahriar et al [58] | ILP formulation with heuristic solver | mean substrate and virtual path lengths | Yes | No | No | Affordable |
| Dehury et al [59] | MIP formulation | CPU, memory and bandwidth | Yes | No | No | Affordable |
| Cheng et al [60] | Node Ranking | CPU & bandwidth | No | Yes | No | Affordable |
| Yao et al [61] | Deep RL | CPU & bandwidth | Yes | No | Yes | Affordable |
| Yuan et al [56] | Q-Learning | CPU & bandwidth | Yes | No | No | Affordable |
| Sciancalepore et al [62] | Q-Learning | CPU & bandwidth | Yes | Yes | No | Costly due to huge solution complexity |
| Xiao et al [63] | Deep RL & policy gradient training | Throughput & operation cost | No | Yes | Yes | Not mentioned |
| Wang et al [64] | Deep RL | CPU, memory & bandwidth | No | Yes | Yes | Computational overhead increases in larger scale networks |
| Pham et al [65] | DDPG | CPU & QoS | No | No | Yes | Not mentioned |
| Dolati et al [52] | Deep RL | CPU & bandwidth | Yes | Yes | Yes | Not mentioned |
| Wang et al [66] | TD | CPU & QoS | No | No | Yes | Not mentioned |

Table 3.3: State of the Art Comparison Table [15]

The above table shows an overview of the main characteristics of the similar and well-known approaches for Network Service Embeddings [1], and it was used as a form of guideline to get inspiration before selecting am Reinforcement Learning method.

[1] This is a part of the original table, you can see more at [15]

# Chapter 4

# Our Approach

## 4.1 Introduction

In this Master thesis, the main objective is to quantify the potential benefits of using Reinforcement Learning methods for Network Service Embedding, such as better resource utilization or lower latency between nodes of a network service graph, compared to state-of-the-art heuristic or exact methods. The first step is to define the optimization objectives and constraints, taking into account the resource demands of our services. As such, the CPU utilization or bandwidth limitations, that may be expressed in service graphs, will be investigated. Subsequently, a heuristic algorithm was developed as a baseline to measure the overall gains through the comparison with the RL method. The idea is to extract some useful conclusions about the Reinforcement Learning methods, which as an approach can lead to an easier scaling to real-world scenarios. For example, by adding more resource requirements to a VNF, the dimensionality of the input features will be increased, so as the complexity of the problem too, this is easier to adapt in a Reinforcement Learning agent than to a classic heuristic or exact approach. In addition, the selected RL method is examined for its efficiency (e.g. time) in various network conditions, e.g. CPU load, number of requests arriving in a specific moment, etc. As for the scale of the problem, regarding network topology, the initial step is using a small network for a micro ($\mu$) datacenter with let's say 4 servers per rack, and then expanding to a larger scale with a few hundred servers per rack. For the evaluation of the performance a network service generator is developed and a Fat Tree model of network topology is built. Moreover, as a further step, there is research to experiment with the Reinforcement Learning method on service graphs that stem from actual datacenter data for service graphs of network functions.

## 4.2 Service Graphs Generator

To generate the data that simulates network service graphs, the NetworkX [5] framework was used, to construct a list of graphs. This list represents the list of pending requests that the algorithm must try to place inside the virtualized network. A service graph has a length that is given from a uniform distribution $randint(2, 5)$. The CPU demands of a VNF node are selected randomly from this list $[0.1, 0.2, 0.3, 0.4]$. As for the link bandwidth demand between the VNF nodes is randomly distributed between $(20, 250)$ Mbps. Both the CPU and Bandwidth demands are normalized in space $[0, 1]$, this is necessary for the later data generation that would feed to the RL model. The total number of services to be generated is given as input, and it is dependent on the total number of servers in the virtual network. A simple example of a VNFs service graph is shown below in Figure 4.1

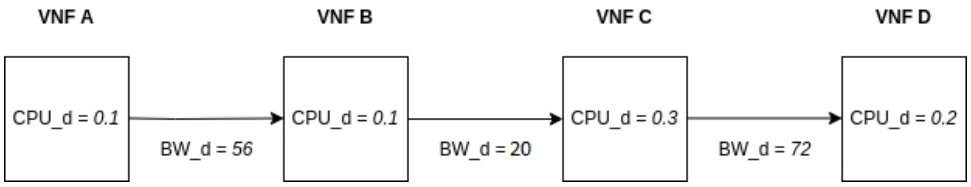



Figure 4.1: Example of VNFs request.

## 4.3 Heuristic Method As Baseline

For the purpose of this work, a heuristic algorithm was developed as a baseline to evaluate the results of the Reinforcement Learning (RL) method. The idea of the heuristic algorithm was inspired by a similar approach referenced in [12]. The algorithm was designed based on a specific network topology 2-layer Fat Tree, while in this case a 3-layered Fat Tree is used. With this change, a different complexity level is added to the problem. The objective of the baseline algorithm is to sort the available servers/nodes based on CPU load and choose the one with the lowest load first. Looking at this from an abstract view, a naive approach would be that the algorithm must classify the servers in groups depending on the network layer, so as to reduce the sorting time. In a nutshell, starting from the higher layer and moving forward the algorithm, must choose the pod, then the rack, and finally, the server where a node will be placed. This process is repeated for each Virtual Network Function (VNF) node of a service graph. Placing sequential nodes of a Virtual Network Request (VNR) also keeps a priority in order to decide where to put the next node of the service chain, firstly trying into the same server as the previous node of the chain, then trying inside the same rack but in a different server, then in the same pod but in a

different rack and so as into a different server. As for the sorting process, the algorithm would have to calculate the average CPU load of each group. So, there would be a variable for storing the CPU average load of servers inside a pod, then one similar variable for racks inside a pod, and finally the same metric for the list of servers inside a rack.

But here, a more straightforward approach is followed as the number of servers in the datacenter is quite manageable, the algorithm referenced below sorts all the servers of the entire datacenter picks the one with the most available capacity, and maps the current VNR's node.

**Algorithm 2** Heuristic Baseline CPU Algorithm

```
1: procedure nseBaselineCPU( VNF_req, datacenter)
2:     vnf_map = [ ]
3:     sorted_servers ← sort_rack_server(datacenter)
4:     rack_id ← sorted_servers[0][0][0]
5:     server_id ← sorted_servers[0][0][1]
6:     for VNF_node in VNFs_req do
7:         if first_node_of_VNFs then
8:             placed, map_node = placement(datacenter, rack_id, server_id, node_name, cpu_d)
9:             if placed then
10:                vnf_map.append(map_node)
11:            else
12:                print("st node  can't be inserted, drop..".format(i+1,node))
13:                break
14:        else
15:            placed, map_node = placement(datacenter, vnf_map[−1][2], vnf_map[−1][3], node_name, cpu_d)
16:            if node_placed then
17:                link_path = find_substate_path(datacenter, vnf[−1], map_node)
18:                if None in link_path then
19:                    vnf_map.append(map_node)
20:                else
21:                    valid_map_link = check_bw(edges[(vnf_map[−1][0], node[0])], link_path)
22:                    if valid_link_map then
23:                        vnf_map_links[(vnf_map[−1][0], node[0])] = link_path
24:                        vnf_map.append(map_node)
25:                    else
26:                        print("loop , link can't be fitted".format(i, (vnf_map[−1][0], node[0])))
27:                        vnf_map, vnf_map_links = revert_map(datacenter, vnf_map, vnf_map_links, nodes, edges)
28:                        break
29:            else
30:                picked_i+ = 1
31:                rack_id ← sorted_servers[picked_i][0][0]
32:                server_id ← sorted_servers[picked_i][0][1]
33:                placed, map_node = placement(datacenter, rack_id, server_id, node_name, cpu_d)
34:                if placed then
35:                    vnf_map.append(map_node)
36:                else
37:                    revert_placement(vnf_map, vnf_map_links, vnf_req_nodes, vnf_req_edges)
38:                    break
```

**Algorithm 3** Heuristic Baseline NSE Algorithm

```
procedure NSEBASELINECPU( VNF_req, datacenter)
    vnf_map = [ ]
    sorted_servers ← sort_rack_server(datacenter)
    rack_id ← sorted_servers[0][0][0]
    server_id ← sorted_servers[0][0][1]
    for VNF_node in VNFs_req do
        if first_node_of_VNFs then
            placed, map_node = placement(datacenter, rack_id, server_id, node_name, cpu_d)
            if placed then
                vnf_map.append(map_node)
            else
                print("st node  can't be inserted, drop..".format(i+1,node))
                break
        else
            placed, map_node = placement(datacenter, vnf_map[−1][2], vnf_map[−1][3], node_name, cpu_d)
            if placed then
                link_path = find_substate_path(datacenter, vnf[−1], map_node)
                if None in link_path then
                    vnf_map.append(map_node)
                else
                    valid_map_link = check_bw(edges[(vnf_map[−1][0], node[0])], link_path)
                    if valid_link_map then
                        vnf_map_links[(vnf_map[−1][0], node[0])] = link_path
                        vnf_map.append(map_node)
                    else
                        print("loop , link can't be fitted".format(i, (vnf_map[−1][0], node[0])))
                        vnf_map, vnf_map_links = revert_map(datacenter, vnf_map, vnf_map_links, nodes, edges)
                        break
            else
                sorted_servers = datacenter.sort_rack_server()
                placed, map_node = placement(datacenter, sorted_servers[0][0][0], sorted_servers[0][0][1]
,node[0], node[1])
                if placed then
                    link_path = find_substrate_path(datacenter, vnf_map[−1], map_node)
                    if None in link_path then
                        vnf_map.append(map_node)
                    else
                        valid_map_link = check_bw(edges[(vnf_map[−1][0], node[0])], link_path)
                        if valid_link_map then
                            vnf_map_links[(vnf_map[−1][0], node[0])] = link_path
                            vnf_map.append(map_node)
                        else
                            vnf_map, vnf_map_links = revert_map(datacenter, vnf_map, vnf_map_links, nodes, edges)
                            break
```

**Algorithm 4** Find Link Path Between Substrate Network's Nodes

```
1: procedure FIND_LINK_PATH(datacenter, nodeA, nodeB)
2:     nodeA_rack = datacenter.rack[nodeA[2]]
3:     nodeB_rack = datacenter.rack[nodeB[2]]
4:     nodeA_server = nodeA_rack.servers[nodeA[3]]
5:     nodeB_server = nodeB_rack.servers[nodeB[3]]
6:     podA = nodeA[1]
7:     podB = nodeB[1]
8:     torA = nodeA_rack.tor_id
9:     torB = nodeB_rack.tor_id
10:    links_path = []
11:    if nodeA_server == nodeB_server then
12:        links_path.append(None)
13:    else if torA == torBandnodeA_server! = nodeB_server then
14:        links_path.append(datacenter.links_d[(torA, server1)])
15:        links_path.append(datacenter.links_d[(torA, server2)])
16:    else if podA_id == podA_id : then
17:        links_path.append(datacenter.links_d[(torA, serverA)])
18:        links2select = find_link_by_child(datacenter, torA)
19:        selected_agg_torA = pick_best_fit_link(links2select)
20:        links_path.append(selected_agg_torA[1])
21:        liks_path.append(datacenter.links_d[selected_agg_torA[0][0], tor2])
22:        links_path.append(datacenter.links_d[(torB, serverB)])
23:    else
24:        links_path.append(datacenter.links_d[(torA, serverA)])
25:        links2select = find_link_by_child(datacenter, torA)
26:        selected_agg_torA = pick_best_fit_link(links2select)
27:        links_path.append(selected_agg_torA[1])
28:        links2select = find_link_by_child(datacenter, selected_agg_torA[0][0])
29:        selected_core_agg = pick_best_fit_link(links2select)
30:        links_path.append(selected_core_agg[1])
31:        links2select = find_link_by_parent(datacenter, selected_core_agg[0][0])
32:        selected_agg_sw = get_agg_both_way(datacenter, links2select, torB)
33:        links_path.append(selected_core_agg[0][0], selected_agg_sw)
34:        links_path.append(datacenter.link_d[selected_agg_sw, torB])
35:        links_path.append(datacenter.links_d[(torB, serverB)])
36:    return links_path
```

**Algorithm 5** Placement Algorithm

```
1: procedure PLACEMENT( datacenter, rack_id, server_id, node_name, cpu_d)
2:     placed = False
3:     map_node = ()
4:     rack_obj = datacenter.find_rack(rack_id)
5:     server_obj = rack_obj.find_server(server_id)
6:     if (server_obj.node_fits(cpu_d))h then
7:         placed = True
8:         map_node = (vnf_node_id, rack_obj.pod_id, rack_id, server_id)
```

A more realistic approach needs to take into account a link's bandwidth demand between two consecutive nodes. So the initial algorithm Heuristic Baseline $NSE_{CPU}$ is extended with a few more steps to validate that the bandwidth constraints, of a given node, are satisfied too, before continuing further with the placement process. This generates the Heuristic Baseline $NSE_{CPU+BW}$ algorithm, where the extra step contains the link mapping.

For the purpose of link mapping implementation, a new subroutine is developed. This routine examines mapped nodes in consecutive pairs, to check where the two physical nodes are located. The available locations that two virtual nodes may be mapped are, into the same server or two different servers. In the case of the same server, there is no need for any link constraint, but in the case of different servers, there is a need for specification of the link's path. The link path varies in length since there are three cases, the two servers are located in the same rack, the servers are in the same pod but different racks, and finally the case where the two servers are in completely different pods. All the above, says that a link path between two physical servers where virtual nodes are mapped can have a length of 2,4 or 6 intermediate nodes.

All the above presents the necessary information for the Find Link Path algorithm.

### 4.3.1 Mapping of VNF

The mapping is done by storing the VNF's node name/id, and the virtual network's node signature. This signature contains the pod's, rack's, and server's identity, and across with the VNF node identity are stored in a tuple data structure. Each VNF node mapping tuple is appended into a list, till every node of the VNF sequence is placed. In case one node can't be placed, the list is discarded and the CPU resources of the server are updated back into the previous values.

$$VNF\_map = (VNF\_node\_name, pod\_id, rack\_id, server\_id) \tag{4.1}$$

$$VNF\_map\_links[vnf\_node_i, vnf\_node_{i+1}] = [Link_1, Link_2..] \tag{4.2}$$

The Equation 4.2 represents a Python dictionary that has as keys two consecutive VNF nodes' names, and its value is a list of Link objects. The Link objects are composing the path on the substrate network, from the server that $vnf\_node_i$ to the server where the $vnf\_node_j$ are placed. So, in order to have the full information of a VNF request that is mapped on a virtual network, there is an extra dictionary data structure that stores the list of all $VNF\_map$ tuples for each VNF node, and a list with all the links between each substrate network's node. It's also important to mention that if a VNF request's nodes can be placed on the same server there is not a $VNF\_map\_links$ list. In conclusion, the first task is to put as many VNF nodes of a service request into the same server and then examine placement into different servers, not considering any priority between pods or racks, just picking the server with the most available CPU. This logic allows a faster decision if a node can be placed into the substrate network or not. This way also reduces the times that servers are sorted based on CPU load before picking one, the sorting process is done once in the beginning and once at the end of the placement.

To better understand the mapping process and the info that is stored in order to construct the mapped VNF node on the substrate network, the whole procedure is visualized by the Figure 4.2

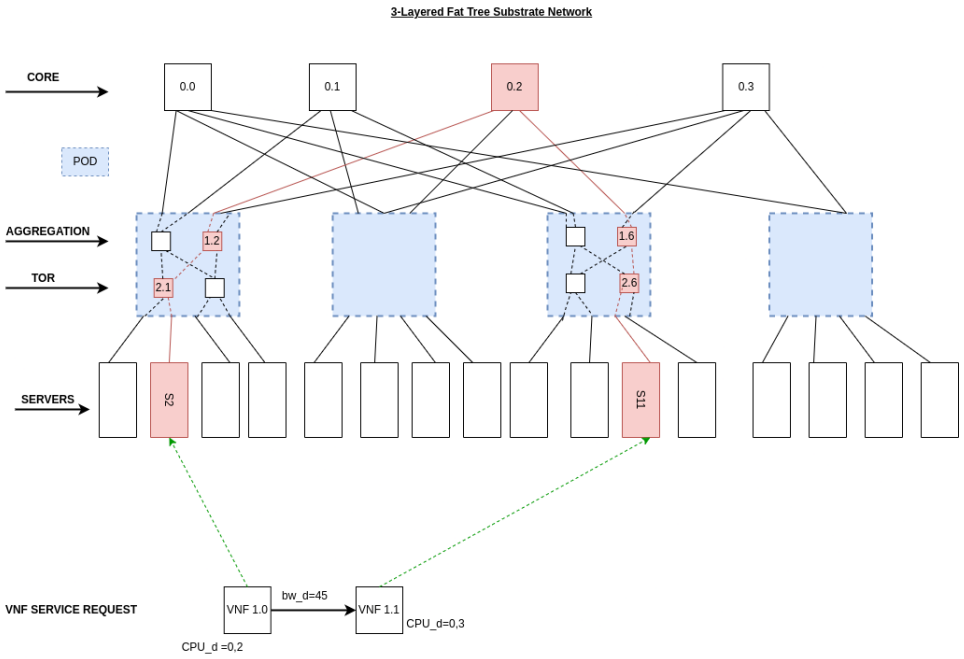



Figure 4.2: Mapping of VNF visual

## 4.4 Reinforcement Learning Method

### 4.4.1 Reinforcement Learning Modeling for VNE

In order to build a Reinforcement Learning approach for the Virtual Network Embedding problem, the initial step is to try and formulate the problem as a Markov Decision Process (MDP). Towards this direction the MDP consists of four sections, the first one is the state, the second one is the action space, the third is the transition probabilities from one state to another, and finally, the rewards that are given to the agent as an action is taken or a state is reached. Let's now define the state section, which in the VNE problem can be the current status of the resources' load of the datacenter, in other words, the average available resources e.g. CPU, Bandwidth of the whole datacenter. In addition to that the state includes the demands of the current VNR, which need to be mapped on the datacenter's servers. As for the action space, this represents the different choices that the agent can make in each state, so in VNE a possible action set is composed of all available datacenter's servers. The agent's action is the selection of one server where the current VNR's node can be mapped. Then as for the transition from one state to another as the result of an action that is done by the agent, it could be determined based on the availability of resources. Last but not least the rewards that the agent gets from executing an action could reflect the desired behaviour of the VNE agent based on the final goal, for example maximizing the VNE request acceptance ratio or minimizing the resource usage or in general better resource utilization.

### 4.4.2 Deep Reinforcement Learning agent

The methodology employed in this study relies on the application of Deep Reinforcement Learning (DRL) techniques, by aiming to address the challenge of efficiently embedding virtual network requests onto physical networks while maintaining the quality of service and minimizing resource consumption. The selected technique particularly focuses on the development of a Virtual Network Embedding (VNE) agent based on a double Deep Q-Network (DQN) architecture. The main purpose of using a Double Deep Q Neural Network approach is because of the overestimation problem that comes along with the simpler DQN architecture [21]. In the DQN model the max operator that is used for calculating the Bellman equation (see 2.6.3,2.6.5), is prone to cause overestimation of Q-values. The intuition behind the overestimation is that the agent may overemphasize some actions, even if they are not the optimal selection. This causes slower convergence or stack at suboptimal policies. In general, this problem may be a huge challenge when the agent explores better solutions. So, a way to address this is by using a DDQN model, which can be shown as an expansion of DQN model. The DDQN as an expansion of DQN, has a restriction which is, that it can be used only for problems with a finite action space. So in a

more simplistic way, if the action space of the VNE mapping problem, is all the possible physical nodes that a datacenter has, this can lead to a large action space, as the datacenter scales up. In a nutshell, it is more difficult for a model to be trained and also to be robust, if the action space is large. Furthermore, it makes the operation of the VNE agent computationally intensive as the agent needs to evaluate many different action combinations and learn optimal policies to follow. This leads to further examining ways of reducing the action space, down to an optimal minimum that is enough for the agent to select from. The solution to this problem consists of the first section of the described methodology's pipeline.

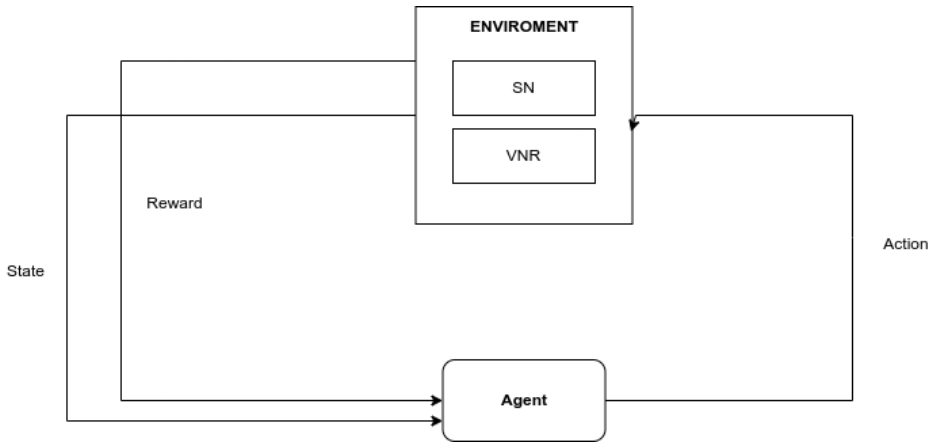



Figure 4.3: Abstract Pipeline of the RL VNE problem

### 4.4.3 DDQN architecture description

In this Section, the overall architecture and functionality of a Double Deep Q Network will be described. The concept of Double Deep Q Network (DDQN) is to build two similar neural networks, the main model and the target model. The architecture of the model is composed of two Dense Layers with a size of hidden layers equal to 24. Each layer has as an activation layer the ReLu 2.7.3, while the selected weights' initializer method used is the HeUniform 2.7.5.1.

The main model is used for the part of the action selection while the target model is used for the evaluation of the selected action. This can be described by the following equation

$$Q(s,\alpha) = r + \gamma * Q_{target}\left(s', \arg\max\left(Q(s',\alpha',\theta),\theta_{target}\right)\right) \tag{4.3}$$

where $Q_{target}(s',\alpha',\theta)$ represents the estimated Q values for the next state $s'$ using the target network with parameters $\theta_{target}$, the $argmax(Q(s',\alpha',\theta)$ is showing the action that maximizes the Q-values for

the next state $s'$ based on the main network with parameters $\theta$.

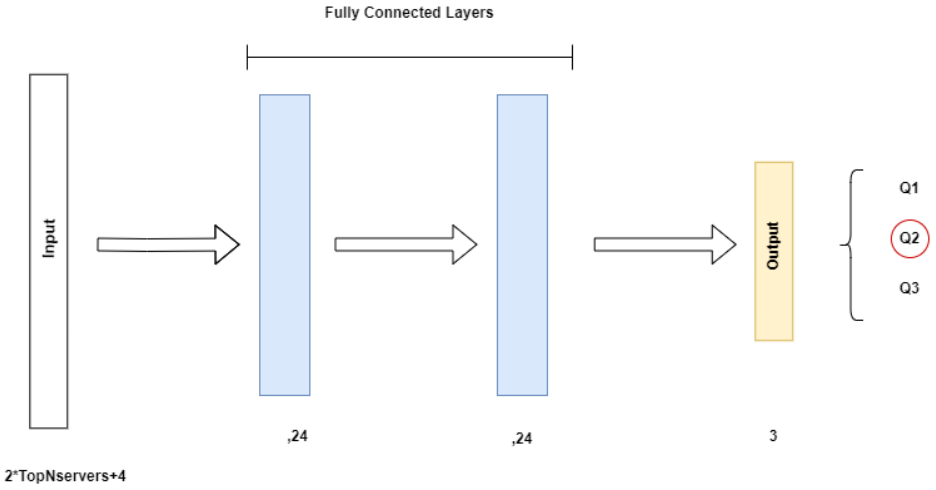



Figure 4.4: Abstract Pipeline of Neural Network Architecture

In the above Figure 4.4 the input size (see: 4.4.4) is defined from the actual size of the action space after is pruned with the help of the Heuristic algorithm (see:4.4.5). The red circle is used to represent the Q-value with the highest value, which is selected at the end using the arg max.

To help further understand the model architecture, a second part of the information is shown in the next Figure 4.5 , and shows the total size of the model, and it is called model's summary.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 24)                264
_________________________________________________________________
dense_1 (Dense)              (None, 24)                600
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 75
=================================================================
Total params: 939
Trainable params: 939
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.5: Model's summary

### 4.4.4 Define Input State

In the presented approach, a vector state representation that encapsulates critical information about the current virtual network and the physical network resources at hand is utilized by the DDQN agent. The main focus of this Section is the ¡state¿ part of the Reinforcement Learning design, which centers on the development of an appropriate state representation capturing the essential features of the VNE problem. Starting from the work referenced in Chapter 3, and specifically at Section 3.2 where the state vector 3.8 is presented, a similar formulation for the state is developed. This state representation must include details about a virtual node that is pending to be mapped. These details are first and foremost the node's resource demands, the physical nodes and their available resources, and the length of the VNE request that is to be handled by the agent, while also there is a value for the order of the node that is currently processed, and ready to be mapped. By looking at this in a more abstract way, as shown in Figure 4.3, the state of the agent is combined by two parts, one is the datacenter's state (SN), and the other is the VNE request's information(VNR).

Breaking down this abstraction and starting from the datacenter's state, it is important to mention that the state's vector size can vary according to the number of actions that the agent is allowed to have. For example, by using the heuristic 4.4.5 algorithm, the action space is pruned down to a specific number. This number is equal to the number of possible physical node candidates that a VNE node can be mapped onto. For the requirements of this project, given that a node has only CPU and Bandwidth demands, so a physical node has two values. This means if the number of servers candidate is equal to 4, the datacenter state has a length of $4 \times 2$. On the other hand, the VNE request's state size is fixed, as it only includes 4 values. To provide additional details, there are two values that correspond to the CPU and Bandwidth demands, one value for the current node index, meaning that there is a linear sequence of nodes in a VNE request, plus a value showing the size of this VNE request.

Let's say that the selected number of server candidates using the 4.4.5 algorithm is 2, so the state

vector looks like this,

$$State = \begin{pmatrix} dc\_CPU_i \\ dc\_BW_i \\ dc\_CPU_{i+1} \\ dc\_BW_{i+1} \\ CPU\_d \\ BW\_d \\ Current\_node\_index \\ Length\_VNE\_req \end{pmatrix} \tag{4.4}$$

| State Representation Feature | Description |
|---|---|
| $dc\_CPU_i$ | The i-th server's CPU load status. |
| $dc\_BW_i$ | The i-th server's Bandwidth load status. |
| $dc\_CPU_{i+1}$ | The i+1-th server's CPU load status. |
| $dc\_BW_{i+1}$ | The i+1-th server's Bandwidth load status. |
| $CPU\_d$ | The CPU demand of the current node to be mapped. |
| $BW\_d$ | The Bandwidth demand of the current node to be mapped. |
| $Current\_node\_index$ | The node of the VNE request that is pending to be mapped. |
| $Length\_VNE\_req$ | The number of total nodes that the current VNE request has. |

Table 4.1: State Vector Features Explanation Table

where $i$ is the index of each server candidate.

### 4.4.5 Heuristic Algorithm for Candidate Servers Selection

The necessity of a finite and manageable action space, in order to train an efficient and robust RL agent, requires reducing the actual action space into a more feasible one. In the Virtual Network Embedding the action space is consisted of all the available servers/nodes of the physical network. This might be a huge amount of nodes, that the agent has to select from so as to make a mapping action, which is increasing the complexity and the execution time. To deal with this uprising problem, a good solution is to select a subset of nodes (servers) out of the total, in a way that let's say the $\kappa$ top candidate nodes to be selected. The selection is made by combing the server's current resource status, in such a way that the selected $\kappa$ servers are those with the highest probability of a successful mapping.

For this reason, a heuristic algorithm was developed, this algorithm is described below:

**Algorithm 6** Heuristic Rank Servers Algorithm

```
1:  procedure GRC_RANK
2:      get_servers_status()
3:      weights = {}
4:      for server, loads in servers_resources.items() do
5:          weights[server] = 1/(loads[0] + loads[1])
6:      total_weight = sum(weights.values())
7:      for server in servers_resources do
8:          weights[server] = weights[server]/total_weight
9:      sorted_servers = sorted(weights.items(), key = lambda x : x[1])
10:     k_selected = [server[0] for server in sorted_servers[−top_N :]
```

As a guideline each time a datacenter object is created we must define the size of the action space, in order to set the boundary on how many servers must be selected. By trying to minimize the range of the available options the agent has, in such a way that will help it to make better decisions. Furthermore, before the ranking of the servers takes place, the current status of all servers' load, in terms of CPU load and Bandwidth availability, must be updated. As the bandwidth of a server, we define the link's capacity between the server and the ToR switch. To do so, a dictionary data structure is used named *servers_resources*, where the key is the server's id and as values are the list of float values, representing the two load variables.

### 4.4.6 Train Process Description

In this section, the training process of the agent is described in the form of pseudocode. The basic idea is that given a datacenter and a number of generated VNRs, representing the episodes, iterating over all the VNRs one at a time, and by calling the agent's train process. Afterwards, the agent's train process uses a node of the passed VNR, and follows the algorithm 7 described below. The training process makes use of an Experience Replay buffer, in which new experiences are added. When the memory size reaches a specific limit above the *batch_size* the actual train is happening, so the DDQN model updates its parameters. The first approach uses the classic selection policy with $\epsilon$-greedy (Section 2.6.7.2).

**Algorithm 7** Training Process Abstract Pseudocode

**procedure** TRAIN_AGENT($vnr, vnr_i, numOfVNRs, upd\_target\_step$)
    $score = 0$
    $done = False$
    $state = get\_state(vnr, 0)$
    **for** $node_i$ in $range(2, len(vnr), 2)$ **do**
        $action = select_action(state)$
        $valid\_action = check\_valid\_action(vnr, action, state)$
        $next\_state = get\_state(vnr, node\_i)$
        $vnr\_state(valid\_action, node_i, len(vnr))$
        $ar = acceptance\_ratio(noVNRs)$
        $reward = calculate\_reward(state, action, valid\_action, len(vnr), ar)$
        $done = is\_done(state, actionvalid)$
        $score+ = reward$
        $add\_mem\_samples(state, action, reward, next\_state, done)$
        $state = next\_state$
        **if** $len(memory) > batch\_size$ **then**
            $replay(batch\_size)$
        **if** $node_i \,\%\, upd\_target\_step == 0$ **then**
            $update\_target\_model()$
            $save\_model()$
            $print("episode|score|e".format(vnr_i, score, epsilon))$
        **if** *done* **then**
            *break*

The following algorithm showing the actual training/updating model procedure. This part is executed only after there is sufficient data inside the memory.

**Algorithm 8** Experience Replay Pseudocode

```
procedure REPLAY_MEMORY(batch_size)
    minibatch = random.sample(memory, batch_size)
    for state, action, reward, next_state, done in minibatch do
        q = model.predict(state)
        if done then
            q[0][action] = reward
        else
            α' = model.predict(next_state)[0]
            q_t = target_model.predict(next_state)[0]
            q[0][action] = reward + γ × q_t[arg max(α')]
        model.fit(state, q)
        if epsilon > min_epsilon then
            epsilon* = epsilon_decay
```

### 4.4.7 Reward function

This section is referred to the Reward and Penalty system that is applied to the agent, to guide it closer to the final goal. For purposes of simplicity, the goal of the agent is to reach a specific Request Acceptance Ratio. While the agent is learning and exploring for solutions, the reward system is split into two different parts. The first part is to reward the agent in each step after a successful action is made, and then give an extra reward for moving towards the goal.

As mentioned before the goal to catch is to reach a specific Request Acceptance Ratio. With that being said, the Acceptance Ratio threshold for this problem is set to be 0.95. So, the Acceptance Ratio reward function is described as :

$$R_{RAR}(x) = \begin{cases} 1, & \text{if } x \geq 0.95 \quad (4.5) \\ 1 - (0.95 - x), & \textbf{otherwise} \quad (4.6) \end{cases}$$

In general, the threshold is a variable and can be tuned up accordingly to the specific task. In the above equation, the $x$ represents the current calculated Request Acceptance Ratio.

The second part of the reward system is defined by the length of the VNR, in terms of how many nodes it has. The concept is that each node is equally important in a way where the maximum reward, here is 1, is split equally for each node.

$$R_{vnr}(status) = \begin{cases} 1/length(VNR_i), & \text{if } status == success \quad (4.7) \\ -1/length(VNR_i), & \text{otherwise} \quad (4.8) \end{cases}$$

The above shows that if a node is successfully mapped by the agent, a reward is given accordingly, in this way if a whole VNR is successfully mapped, the agent gets a reward of 1. In case the agent maps fail, an equivalent negative reward is given.

With all these being said, the total reward function is given by the equation:

$$R = \alpha * R_{vnr}(status) + \beta * R_{RAR}(x) \tag{4.9}$$

where $\alpha, \beta$ are the weights that can be used to balance the importance of each reward in the training process. For simplicity purposes, both weights are equal to 1. In this case, because as is defined by the type of rewards one is related to the number of total VNR generated (epochs) and the other to the length of each VNR, the length of the VNR is much smaller than the total number of VNRs. This means that the $R_{RAR}$ is relatively small at the start of the training and as the agent moves forward, and gets closer to the goal it gets higher $R_{RAR}$ values.

### 4.4.8 Model Parameters

Here the rest of the agent parameters are briefed and discussed, to help explain the Results that are following. A Neural Network model requires a few parameters to be defined before even starting. First thing first the Loss function that was used was the MSE 2.7.4.1 and also an alternative of it called Huber 2.7.4.2, both choices had similar performances, so the MSE was kept as the final choice. The learning step was set 0.001 and the selected optimizer was Adam [34]). The batch size was set into different values, according to the size of the training process in terms of the number of generated VNRs (epochs), so the batch size was set either to 20,32 or 64, the mid one was the selected one. Now, coming into the part of DDQN architecture there are some extra parameters that are used for the Q-function 2.6.5. These are the $\gamma$, known as the discount factor which was set to 0.95, then there is the $\epsilon$, since the DDQN by default follows an epsilon-greedy policy, so the $\epsilon$ initial value was set as 1, with the final or minimum value of $\epsilon$ to be 0.01, while the $\epsilon$ decays with a step of 0.99, as the agent moves into the environment and tries to balance exploration and exploitation of new actions.

# Chapter 5

# Results

In this chapter, the results of this work are presented according to each of the methods described in the previous Chapter 4. First, of, it would be important to mention that this work is a simple proof of concept, with the purpose of quantifying any possible outcomes of Reinforcement Learning methods for the Virtual Network Embedding process, with its goal to be another valuable pointer toward the state of the art path for the field. It would be more than necessary to examine, the above methodology at deeper levels, and compare it with other similar approaches. In order to evaluate the proposed methods, some well-known metrics would be used. These are the Request Acceptance Ratio, the CPU utilization, and the Bandwidth utilization, which have been referenced in other similar works [67].

In the first section of Chapter 5, the environment for the experiments of the Baseline algorithm (Section: 4.3) is described, and then the results in terms of VNE request acceptance ratio, CPU, and link utilization are presented, along with the necessary comments. Then in the next Section, the training progress of the agent is presented, and finally the overall evaluation of the agent across with the comments.

The training and the evaluation process was made by using randomly synthesized data, that were fabricated by taking into account the recent bibliography like [54].

When it comes to hardware specifications to train the RL agent and to run the simulation, because of the complexity of Neural Networks in general, a GPU was necessary. This was a MSI Nvidia 1080 8 Gb and the PC used has also a AMD Ryzen 2600X 3.6GHz CPU with a total memory size of 16 Gb.

### 5.0.1 Experiments Description

The characteristics of the generated data are specified in the first part of this chapter. In this case, VNE requests are utilized, which are represented in two forms. The first form is the actual VNE graph, which is solely used for plotting purposes. The second form is a simplified representation consisting of a

list of float values, which are handled as pairs. Each pair comprises the CPU and bandwidth demands of a VNE node. For the purpose of symmetry, a simple zero padding is required. This means that the final element of the list has a value of zero, as the last node does not have any link and, therefore, no bandwidth demand.

The CPU demand of a VNE node is randomly sampled between $0.1 - 0.4$, and corresponds to the percentage of CPU the node needs to occupy in order to operate. The CPU available load is normalized in $0 - 1$, so the maximum available CPU is 1. As for the bandwidth demand, is randomly selected too, but between $20 - 250$, in the magnitude of Mbps. The datacenter's links have a capacity of 1Gbps, and this is also normalized between $0 - 1$. So, the randomly selected bandwidth value must be divided by 1000, before being assigned as a node's demand.

To generate different sizes of VNE requests, a list with possible up-limits, such as $max\_length = [3, 5, 10]$, is utilized. The minimum size of a VNE request is 2. This approach generates a variety of VNE requests that are evaluated. Furthermore, it is worth noting that the number of VNE requests changes dynamically, ranging from a few hundred up to a few thousand, ranging from 100-8000 requests.

The datacenter is also scaled up by a factor of k, which represents the number of switch's ports. Specific k values are selected from a list such as $k \in [8, 24]$. Generally, it is considered a good practice for k to be a factor of a power of 2.

### 5.0.2 Define Evaluation Mid-Case scenario using the Baseline Algorithm

In the Figures below, the idea is that there is a plot for each metric under consideration, Request Acceptance Ratio (RAR), Central Process Unit (CPU), and bandwidth usage. In each chart, the x-axis is the same and shows the number of VNE requests generated. The y-axis is the value of each measurement. Also, as is obviously observed, there are three different coloured graphs in the same graph. This is done to show the relationship of each measurement to the length of the VNE request. In short, the example of the magenta-coloured graph in the RAR plot shows how RAR changes as the number of VNE requests increases, while a VNE request has a random length of 2-3 nodes. There are six graphs in total, three per value of the $k$ that is the parameter for the scale of the datacenter.

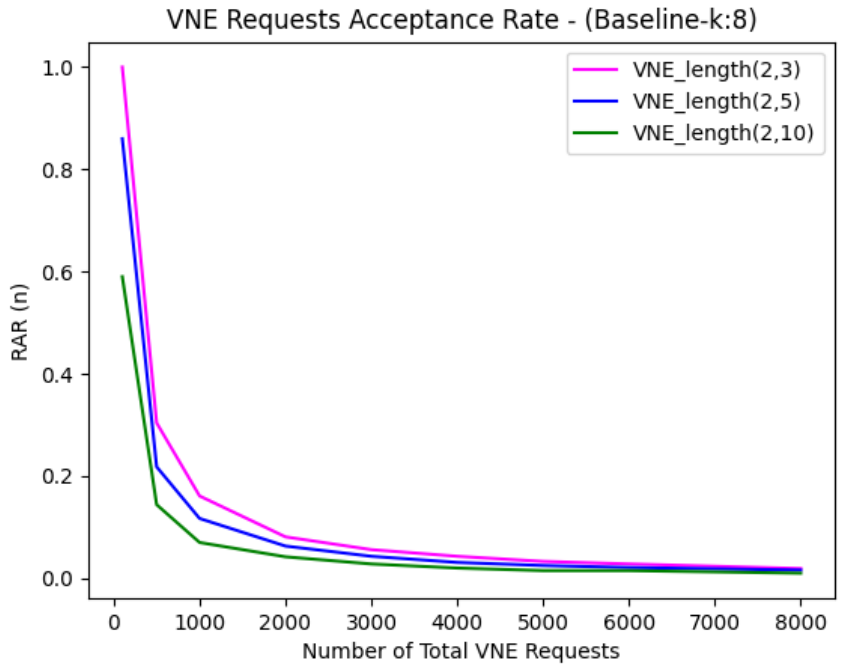



Figure 5.1: VNE Request Acceptance Ratio per VNE size

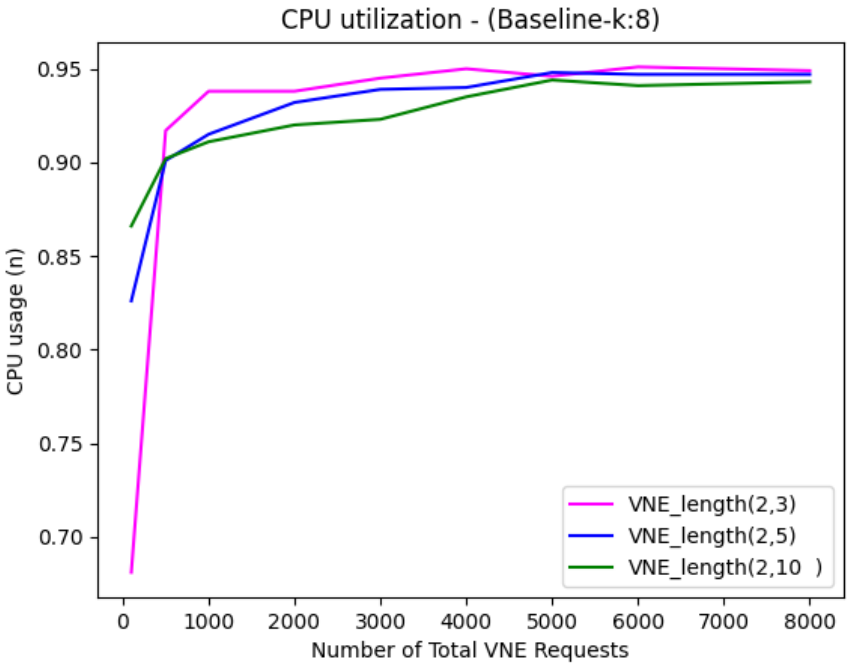



Figure 5.2: CPU utilization per VNE size

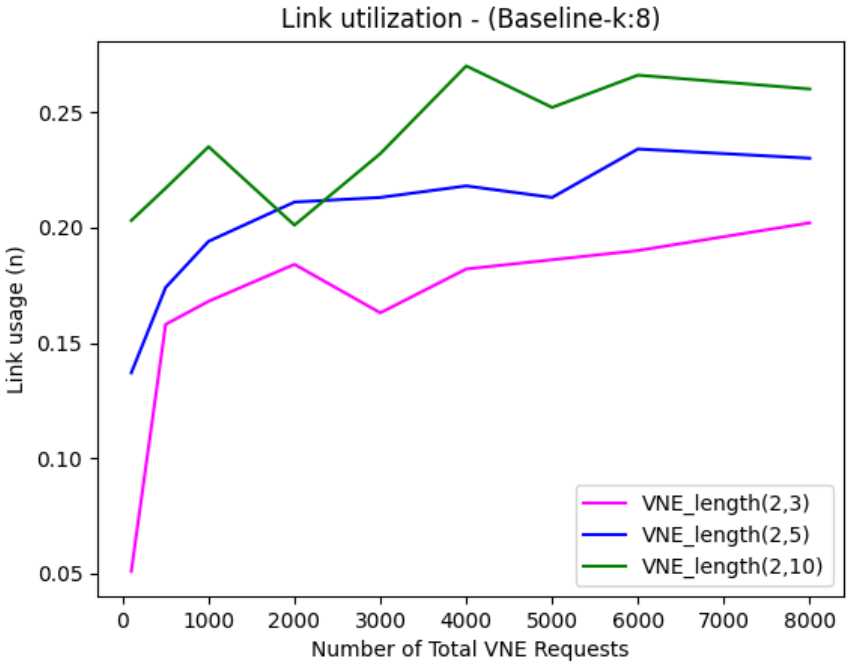



Figure 5.3: Bandwidth utilization per VNE size

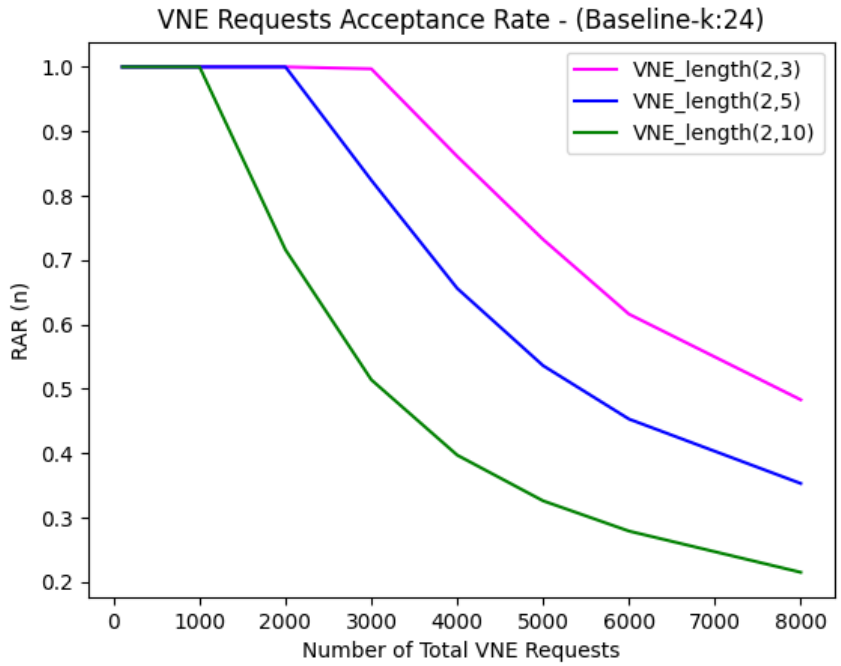



Figure 5.4: VNE Request Acceptance Ratio per VNE size

The relationship between the request acceptance ratio (RAR) and the total number of VNE (virtual network embedding) requests is illustrated in figures (5.4,5.1). As the total number of VNE requests increases, the RAR decreases due to the limited capacity of the datacenter resources. Specifically, in the conducted experiments, the scale-up of the datacenter in terms of the number of servers by a parameter $k$. The total number of servers in the datacenter is $\frac{k^3}{4}$, where $k \in [8, 24]$. In Figure (5.1), the RAR plot is shown for $k = 8$, while in Figure (5.4), the $k$ is increased to 24. Furthermore, both figures demonstrate that as the length of VNE requests increases, the RAR decreases.

The Figure (5.2) displays the CPU utilization, respectively, for $k = 8$. It is worth noting that each request has multiple nodes with various resource demands. It is evident from the figure that the Baseline(4.3) algorithm prioritizes mapping the nodes of a VNE request onto the same server if possible, resulting in higher CPU usage. It follows that as the length of VNE requests increases, more nodes are mapped onto the same server, resulting in higher CPU usage. Therefore, VNE requests with a maximum length of 10 exhibit greater CPU usage. This is mostly because the VNE request that comes first allocates the server's CPU proportional to the number of nodes, so the next VNE request might not fit in the server due to bigger demands, and the CPU available capacity is reduced significantly from

the previous request.

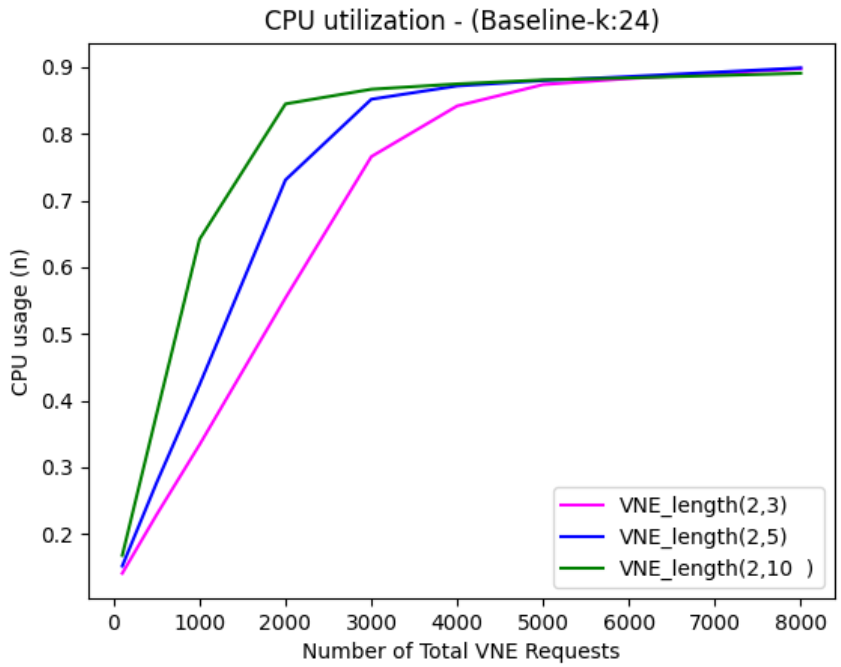



Figure 5.5: CPU utilization per VNE size

The graph in Figure (5.3) illustrates that the bandwidth utilization of links is relatively low since most of the requests are mapped onto the same server. While this eliminates latency between nodes in VNE requests, it creates a single point of failure. If one node operation fails, the entire VNE request fails. It is worth noting that the generated VNE requests have larger CPU demands than bandwidth. For instance, a VNE node may require 0.4 of the CPU capacity while the bandwidth requirement is only 0.04. The difference in resource usage between Figures (5.2,5.5, 5.3,5.6) is due to this discrepancy in resource demands.

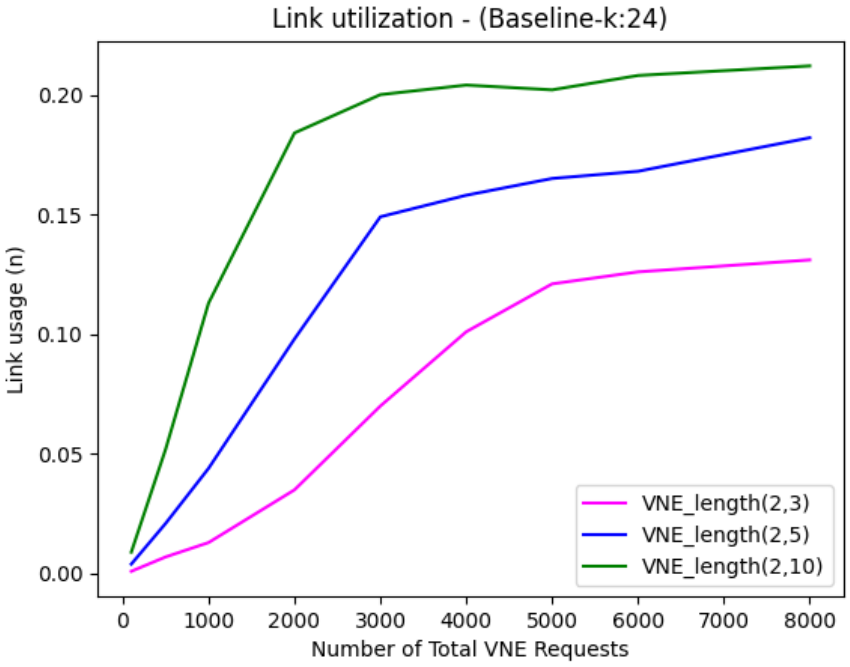



Figure 5.6: Bandwidth utilization per VNE size

The results presented in this Section 5.0.2 are used in order to get an intuition about the behavior of the problem and its environment. More precisely, the objective is to use the above simulations as a guideline to pick a mid-case scenario to be used for the performance evaluation of the two proposed methods. So as a conclusion, for the next Section 5.2 the size of VNE request is set to $2-5$ nodes, and the maximum number of VNE requests is set to 8000 for the case where $k = 24$. The $k$ is either 8 or 24, so as to show the difference as the datacenter scales up.

## 5.1 Agent Train Results

For this Section, the generated VNR data, has the length of the (2,5), where the minimum number of nodes is 2 and the maximum is 5. As was examined in the previous Section the VNE case with length (2,5) was selected as a mid-case scenario, to test the agent built in the current work.

### 5.1.1 Learning Process

For the training process of the agent two different scales $k = [8, 24]$ of the training datacenter-instance were tested, where at the end the smaller one was selected in order to evaluate its performance.

Where $k$ is the scale-up factor of the datacenter as described above (see: Section 5.0.1) and the number of VNRs can be seen as the number of episodes.

The above plots were examined for a variety of parameter configurations and came out that the action space size had a huge impact on the learning process. More precisely for the architecture that is described fair well in the previous Chapter, the optimal training progress is achieved by using an action space of 3 or 4. The action space size defines the number of servers that the Heuristic Algorithm 4.4.5 pre-selects in order to minimize the action space. From the training process is shown that for an action space with size 2 (binary selection) or greater than 4, the model doesn't converge properly, actually, the MSE loss shows larger deviations from the optimal. With all the above being said the final choice for the actual action space size was 3. So, the agent actually has three servers as candidates to select from, in order to map a VNE node.

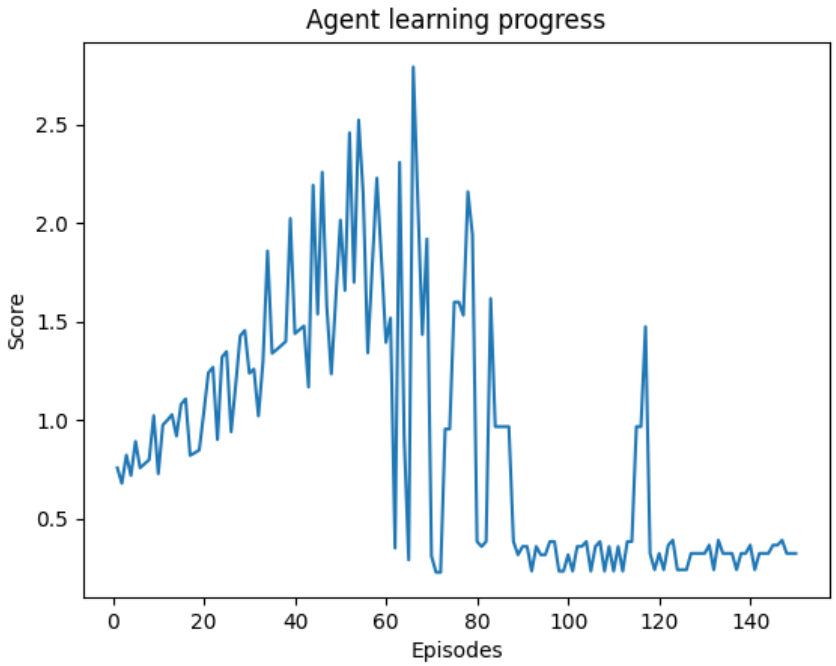



Figure 5.7: Score per Epoch for k=8 and number Of VNRs 150

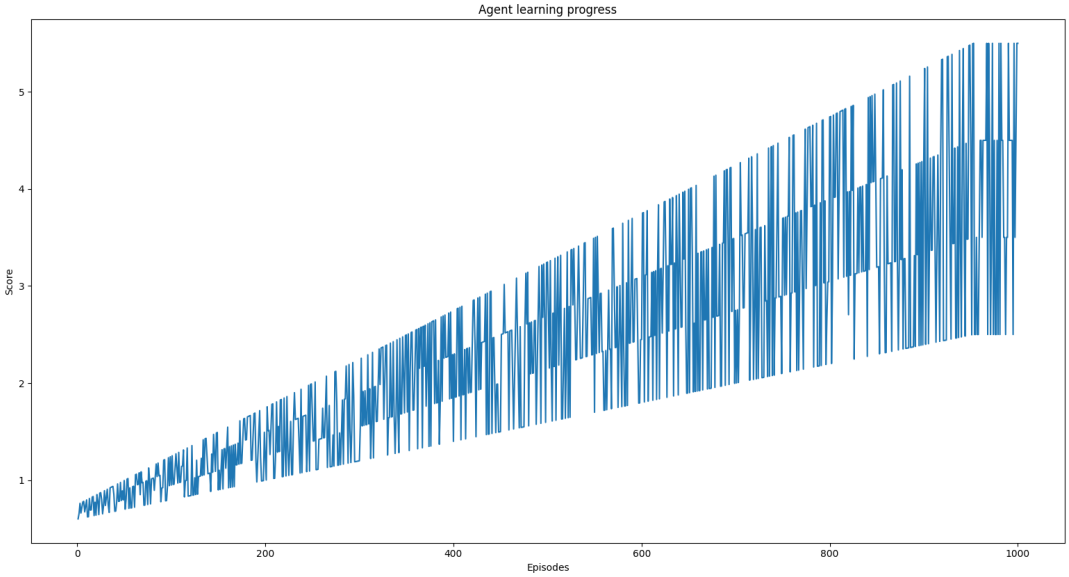



Figure 5.8: Score per Epoch for k=24 and number Of VNRs 1000

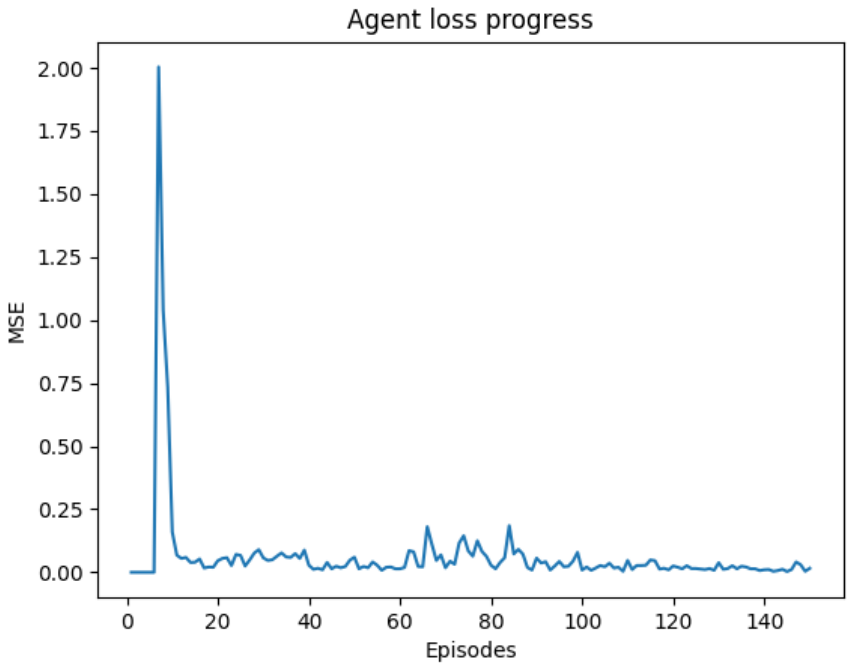



Figure 5.9: MSE Loss per Epoch for k=8 and number Of VNRs 150

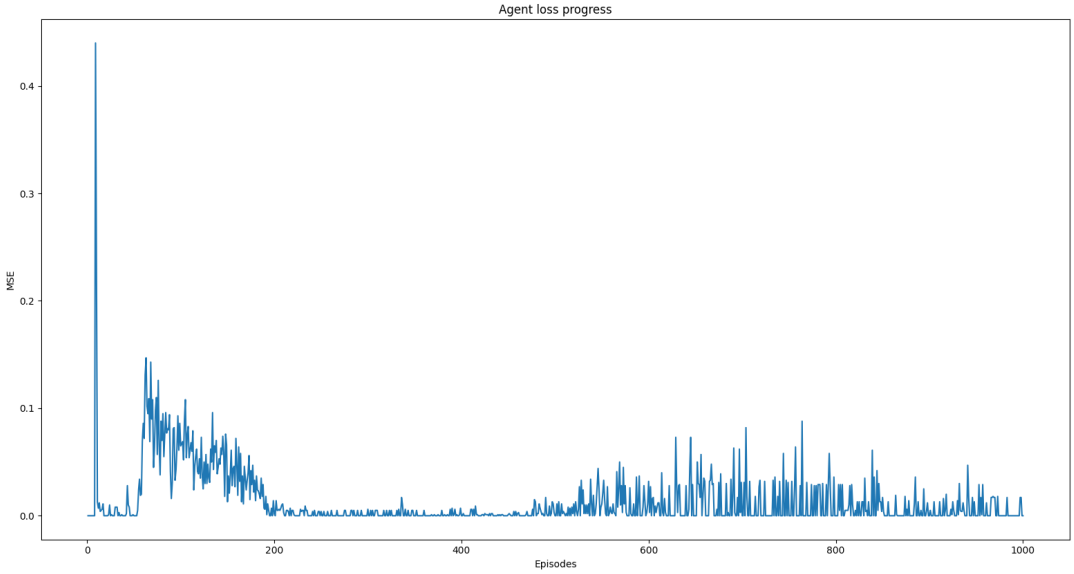



Figure 5.10: MSE Loss per Epoch for k=24 and number Of VNRs 1000

## 5.2 Evaluation Plots Baseline vs DagNeQ Agent

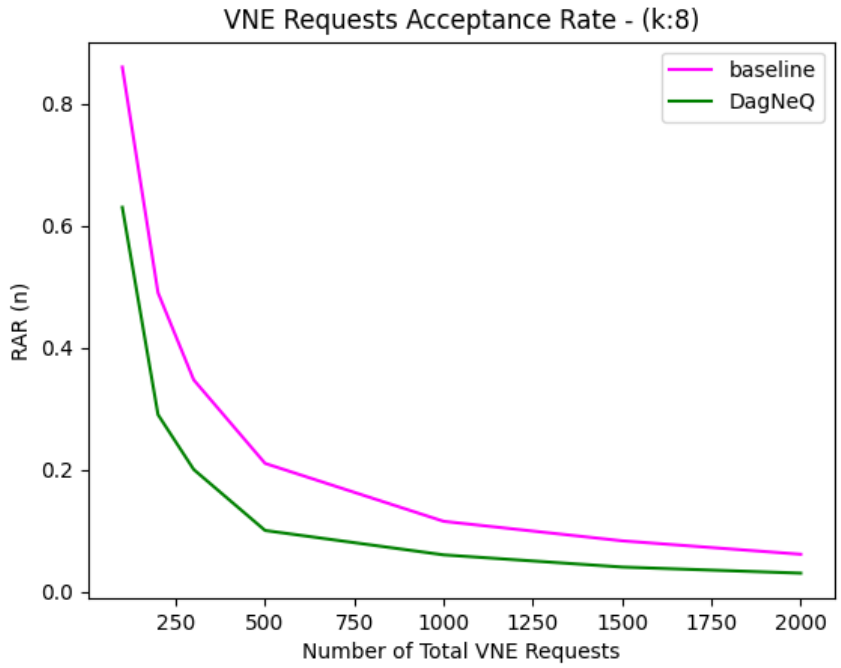



Figure 5.11: Request Acceptance Ratio

The Figures 5.11, 5.12, 5.13 are the showcase, for the VNE mapping using the proposed agent in Chapter 4 at Section 4.4.1. Starting from Figure 5.11, which is the base characteristic to compare the Agent with the Baseline method, as shown from the plot the agent has similar or worse behavior than the baseline, so it seems the model described in this work doesn't outperform the Baseline algorithm. But at this point is good to take into account that the agent is very dependent on selecting a suitable reward function, and in the case of this work, it might need some improvements. Moreover, the randomly generated data lacks the necessary quality and there are some general assumptions like the size of the VNR length or the resource demands for each node, that affect the overall performance. That being said, it points out the necessity of exploring and further research for more realistic data, that are closer to the real-world problem of VNE mapping. As for the Figures 5.12, 5.13 the results are way worse than the Baseline. But need to consider that the agent does not have any information about the CPU or Bandwidth utilization while training.

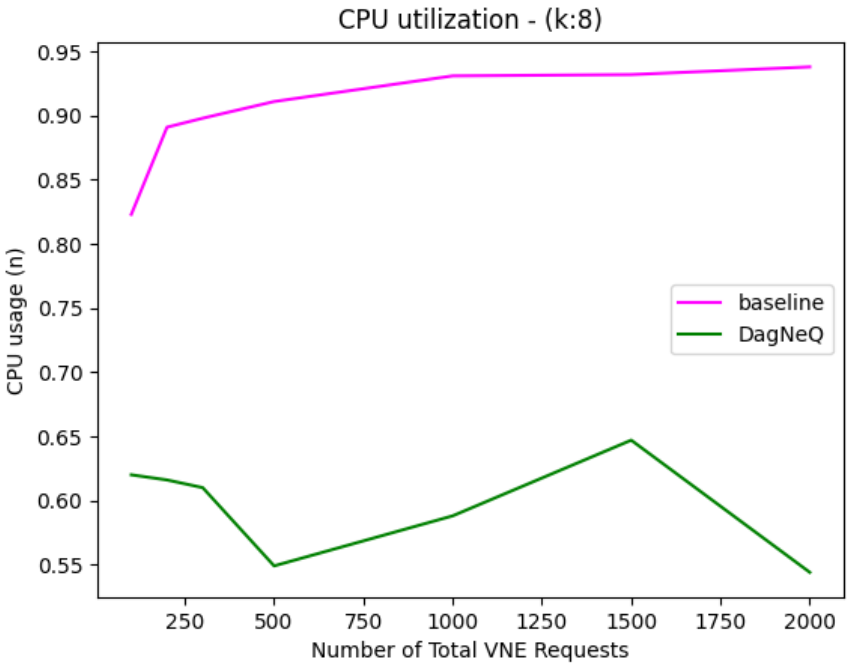



Figure 5.12: CPU utilization

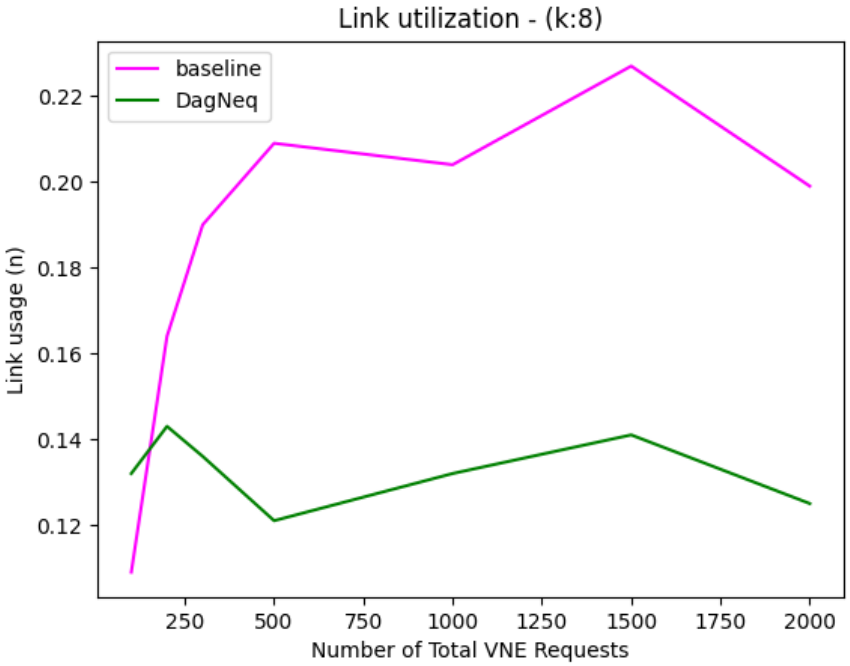



Figure 5.13: Bandwidth utilization

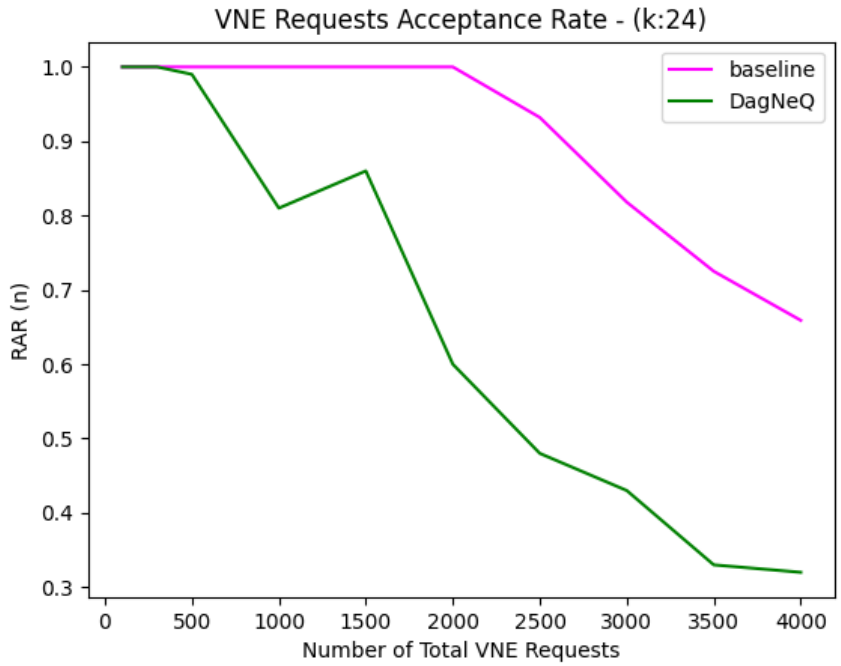



Figure 5.14: Request Acceptance Ratio k:24

As the datacenter is scaled up in Figures 5.14,5.15,5.16, and the number of VNE requests, the gap between the two approaches becomes more apparent. For a large number of VNE requests the Baseline algorithm seems to make a better decision regarding the mapping process. From the utilization plots for CPU and Bandwidth, it is pretty obvious that the Baseline is also doing better. The Figures 5.15,5.16 if closely looked together, may help to observe that the DagNeQ agent doesn't develop any specific hierarchical preference in terms of which server has an advance over another. As was described in the Baseline algorithm, it was designed to select firstly the same server then a server that belongs to the same rack, and then anyone else. This is why the DaqNeQ seems to use more bandwidth in the beginning while the Baseline after it starts to exhaust the other options.

As for the CPU and Bandwidth utilization is expected behavior, as mentioned above at 5.12, 5.13 , because in the above test case scenario of the agent (DagNeQ) the reward function does not include any reward point system for reaching a CPU or Bandwidth-based objective. In a nutshell, the agent is not trained to look for any resource utilization. This causes the agent to have a suboptimal solution in comparison with the Baseline algorithm.

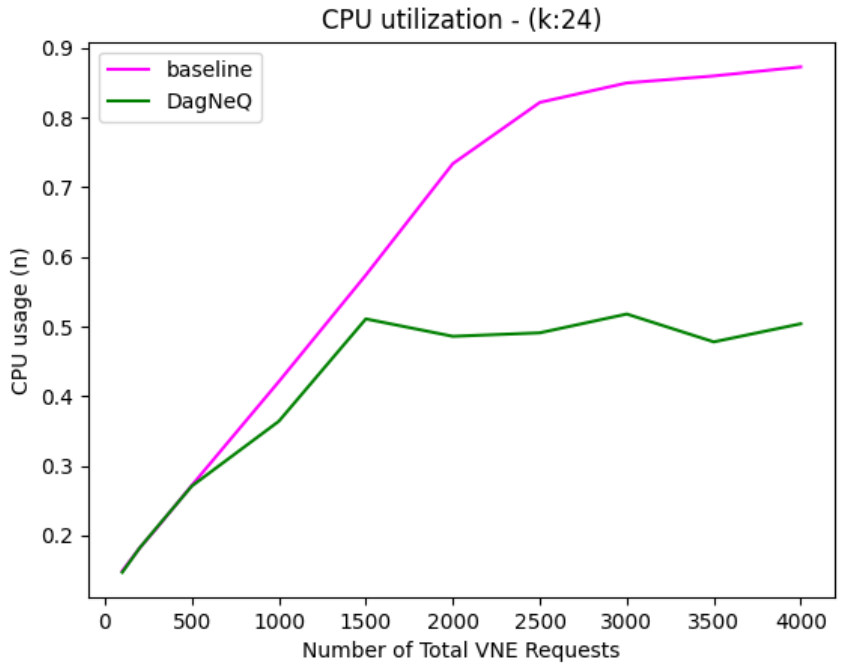



Figure 5.15: CPU utilization k:24

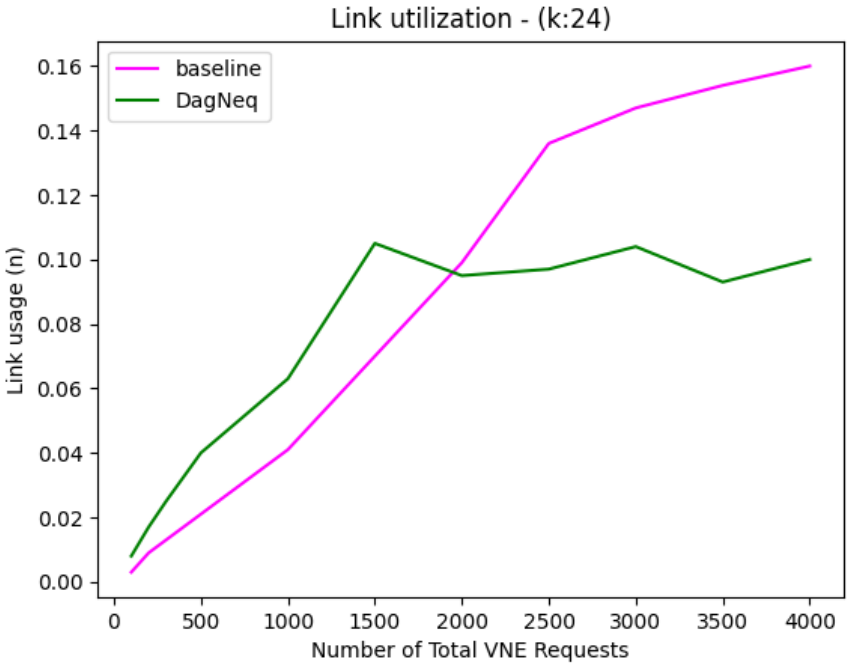



Figure 5.16: Bandwidth utilization

# Chapter 6

# Conclusion

## 6.1 Conclusion

This master thesis describes an alternative approach for Virtual Network Embedding by using Reinforcement Learning. The developed model is a variant of the Deep Q-Learning model implemented with the help of *Tensorflow* framework and *Keras API* based on Artificial Neural Networks. The Network Service Embedding agent gets as input the current state of the Virtual Network combined with the Network Service request. The goal is to select the appropriate action: pick the suitable server to map a VNR's node. As the process goes on, the agent aims to map the whole VNR. The final objective is set to it through the reward function. In this work, the objective is to reach a Request Acceptance Ratio of 0.95 and over. So, is important to define a specific reward system according to the desired objective. For example, an alternative would be to have a reward function that shows the agent to achieve a specific resource utilization threshold. Furthermore, it could be a more complex task to solve, like a combination of Request Acceptance Ratio and CPU utilization thresholds that must be reached. As the results from the previous Chapter 5 indicate, there is a need for deeper research on building a reward function. By designing and implementing a more sophisticated reward system, the goal is to build an optimal agent that solves the VNE problem. One more final thought would be after testing different reward scenarios to implement and evaluate different Neural Network Models, in terms of alternative architectures. It may seem like, a too narrow-visioned approach, following the Neural Network path, for the model, but the level of abstraction that has to offer can make things a lot easier.

## 6.2 Future Work

With a simplistic overview of the VNE problem, there are a few ideas that are worth investing more time for further research. For example, expanding the resource demands by adding extra features like memory demand, time duration of a VNE request, or more complex relations between a VNE's nodes. Furthermore, to move closer to the real-world scenarios, a VNE request's demands and properties are closely investigated to get a good-quality of data. In more detail, a VNE request may contain information about the size of a VNE request, the resource demands values that are needed per type of VNE or the type of graph that the VNE's nodes form with respect to node dependencies. As a final thought the time factor is crucial also, meaning in the online mapping process will requests keep coming, how much time does the agent have to do the mapping before the subsequent request comes? This also raises the possibility of parallelizing the process of the mapping, to achieve better timing.

### 6.2.1 Multi Dimension resources demands

As a new path to be discovered in the future, it would be useful to examine the above proposed solution to a higher dimensionality. In this work, the service graphs that were used, had only demands on CPU and Bandwidth resources, while in real-world applications there are more needs like RAM, GPU, etc. So, it is quite interesting to further explore and analyze the results of the above work by adding complexity to it and also by choosing a different method as the baseline, in order to quantify any possible gains.

### 6.2.2 Improve Efficiency of the proposed method

Further experimentation and research must be done, so as to improve the efficiency of the proposed VNE agent. The first thing to be re-examined should be the reward system, which will provide multiple objectives except the Request Acceptance Ratio. The results point out the necessity for better resource utilization, which can be achieved through a represented reward function. Secondly, a possible improvement would be to add an extra action into the available selection set, and that would be something like "no-op". This action, "no-operation" or simpler "no-action" will help the agent avoid mapping a VNE node if it doesn't apply to the resource constraints of any of the available servers. This will give the agent the ability to instantly reject a request.

### 6.2.3 Evaluate the system into a real world application

The main concern for future work will be the further testing and scaling of the proposed solution, into a synchronous datacenter and using service graphs from real-world scenarios. At the same time, it will be

necessary to re-evaluate the results of this work with more similar approaches. This will give us results for comparison to mark how good the above implementation is and of course to provide a more robust conclusion about the benefits of it.

### 6.2.4 Outcome

If we look beyond the implementation of this project, the next goal to chase is to build an agent as a software package, to be used in modern datacenter applications. An actual software product that will be useful to Service Provider companies, and will help them to improve the quality of their services while providing a better customer experience like cheaper subscriptions and a more fair balance between cost and demands for each use-case.

## 6.3 Lessons Learned

The period of engagement with this Master's thesis on Virtual Network Embedding methods utilizing Reinforcement Learning has unveiled valuable insights into virtual network resource optimization. The research and experimentation, that took place in this work, gave some critical lessons. Firstly, the choice of the DQN algorithm and its hyper-parameters significantly influences the performance of the VNE agent. Furthermore, this study underscores the importance of the accurate virtual network environment modeling. The reward system design in a Reinforcement Learning-based VNE problem is one more factor that profoundly impacts the learning process, subsequent embedding decisions, and the overall efficiency of the approach. In addition, a noteworthy observation pertains to the scalability of the proposed DQN method for VNE, which may encounter challenges when dealing with large-scale networks. As a final conclusion, this research offers a comprehensive understanding of the potential of Reinforcement Learning methods, underlining the significance of careful algorithmic choices, robust network modeling, and resource-aware reward design in achieving efficient and adaptable network embedding solutions, while exposing any difficulties or problems during the development and design process.

# References


[1] VMware: Network functions virtualization (2013) https://www.vmware.com/topics/glossary/content/network-functions-virtualization-nfv.html. v, vi

[2] ETSI-ESO: NFV ETSI, GS NFV 002 V1.1.1 Network Functions Virtualisation (NFV), Architectural Frame- work (2013) https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf. v, vi, 4

[3] Van Rossum, G., Drake, F.L.: Python 3 Reference Manual, Scotts Valley, CA. (2009) v, vi

[4] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015) Software available from tensorflow.org. v, vi

[5] Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In Varoquaux, G., Vaught, T., Millman, J., eds.: Proceedings of the 7th Python in Science Conference, Pasadena, CA USA (2008) 11 – 15 v, vi, 42

[6] Konstantin S. Solnushkin: Fat Tree Design (May 2013) https://clusterdesign.org/fat-trees/. xi, 8, 10

[7] Panagiotis, P.: Big data networks lectures (2021) University of Macedonia. xi, 7, 9

[8] Urrea C, B.D.: Software-Defined Networking Solutions, Architecture and Controllers for the Industrial Internet of Things: A Review (October 2021) 1

[9] Varvara, A.: A framework for virtual network functions(vnf) modeling and service graph verification in sdn/cloud context. Master's thesis, Politecnico Di Torino (2018) 3

[10] Wang, L., Mao, W., Jin Zhao, Y.X.: DDQP: A Double Deep Q-Learning Approach to Online Fault-Tolerant SFC Placement. IEEE Transactions on Network and Service Management (2021) 4, 39

[11] Cohen, P.: What is Virtual Network Function? (October 2021) https://www.rcrwireless.com/20211021/fundamentals/what-is-a-virtual-network-function. 4, 5

[12] Pentelas, A., Papathanail, G., Fotoglou, I., Papadimitriou, P.: Network Service Embedding Across Multiple Resource Dimensions. IEEE Transactions on Network and Service Management (2020) 4, 6, 33, 34, 35, 36, 42

[13] Fischer, A., Botero, J.F., Beck, M.T., de Meer, H., Hesselbach, X.: Virtual Network Embedding: A Survey. IEEE Communications Surveys and Tutorials **15**(4) (2013) 1888–1906 6

[14] Alkmim, G., Batista, D., Fonseca, N.: Mapping virtual networks onto substrate networks. Journal of Internet Services and Applications **4** (01 2013) 6, 33

[15] Yan, Zhongxia and Ge, Jingguo and Wu, Yulei and Li, Liangxiong and Li, Tong: Automatic virtual network embedding: A deep reinforcement learning approach with graph convolutional networks. IEEE Journal on Selected Areas in Communications **38**(6) (2020) 1040–1057 6, 33, 34, 37, 38, 39

[16] E. L. Erwin: Telephone system (February 1938) US2244004A. 7

[17] Clos, Charles: A study of non-blocking switching networks. The Bell System Technical Journal **32**(2) (1953) 406–424 7

[18] Leiserson, C.E.: Fat-trees: Universal networks for hardware-efficient supercomputing. IEEE Transactions on Computers **C-34**(10) (1985) 892–901 8

[19] Andrew, A.M.: Reinforcement learning:: An introduction. Kybernetes (1998) 11

[20] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540) (2015) 529–533 11, 13, 14

[21] Li, Y.: Deep reinforcement learning: An overview (2017) 11, 14, 50

[22] Racanière, S., Weber, T., Reichert, D., Buesing, L., Guez, A., Jimenez Rezende, D., Puigdomènech Badia, A., Vinyals, O., Heess, N., Li, Y., et al.: Imagination-augmented agents for deep reinforcement learning. Advances in neural information processing systems **30** (2017) 12

[23] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T.P., Simonyan, K., Hassabis, D.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. CoRR **abs/1712.01815** (2017) 12

[24] Sammut, C., Webb, G.I., eds. In: Bellman Equation. Springer US, Boston, MA (2010) 97–97 12

[25] Jordi Torres: The Bellman Equation (June 2020) https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7. 12

[26] van Otterlo, M., Wiering, M. In: Reinforcement Learning and Markov Decision Processes. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 3–42 13

[27] Fan, J., Wang, Z., Xie, Y., Yang, Z.: A theoretical analysis of deep q-learning. In Bayen, A.M., Jadbabaie, A., Pappas, G., Parrilo, P.A., Recht, B., Tomlin, C., Zeilinger, M., eds.: Proceedings of the 2nd Conference on Learning for Dynamics and Control. Volume 120 of Proceedings of Machine Learning Research., PMLR (10–11 Jun 2020) 486–489 13

[28] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In Balcan, M.F., Weinberger, K.Q., eds.: Proceedings of The 33rd International Conference on Machine Learning. Volume 48 of Proceedings of Machine Learning Research., New York, New York, USA, PMLR (20–22 Jun 2016) 1928–1937 13

[29] Deep Learning: MultiLevel Perceptron. http://deeplearning.net/tutorial/mlp.html. 14, 18, 19

[30] Karlik, B., Olgac, A.V.: Performance analysis of various activation functions in generalized mlp architectures of neural networks. International Journal of Artificial Intelligence and Expert Systems **1**(4) (2011) 111–122 14, 19

[31] MissingLink.AI: 7 Types of Neural Network Activation Functions: How to Choose? https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/. 14, 19

[32] Sherstinsky, A.: Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network (2018) cite arxiv:1808.03314Comment: 39 pages, 10 figures, 66 references. 18

[33] O'Shea, K., Nash, R.: An introduction to convolutional neural networks. CoRR **abs/1511.08458** (2015) 18

[34] Karageorgiadis, A.: FACESiR: Face and Speaker identity recognition in video streams. Master's thesis, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece (2019) https://doi.org/10.26233/heallink.tuc.84791. 18, 19, 58

[35] Rogers, A., Kovaleva, O., Rumshisky, A.: A primer in bertology: What we know about how bert works (2020) 18

[36] Kalyan, K.S., Rajasekharan, A., Sangeetha, S.: Ammus : A survey of transformer-based pretrained models in natural language processing (2021) 18

[37] McLaren, M., Yun, L., Scheffer, N., Ferrer, L.: Application of convolutional neural networks to speaker recognition in noisy conditions. INTERSPEECH-2014 (2014) 686–690 18

[38] George Seif: Understanding the 3 most common loss functions for Machine Learning Regression (2019) https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3. 24

[39] Sammut, C., Webb, G.I., eds. In: Mean Squared Error. Springer US, Boston, MA (2010) 653–653 24

[40] Zhang, Z., Sabuncu, M.: Generalized cross entropy loss for training deep neural networks with noisy labels. Advances in neural information processing systems **31** (2018) 24

[41] Ruby, U., Yendapalli, V.: Binary cross entropy with deep learning technique for image classification. Int. J. Adv. Trends Comput. Sci. Eng **9**(10) (2020) 24

[42] Jadon, A., Patil, A., Jadon, S.: A comprehensive survey of regression based loss functions for time series forecasting (2022) 25

[43] NΓIΓ±ez, E., Steyerberg, E.W., NΓIΓ±ez, J.: Regression modeling strategies. Revista EspaΓ±ola de CardiologΓa (English Edition) **64**(6) (2011) 501–507 25

[44] Huber, P.J.: Robust estimation of a location parameter. Breakthroughs in statistics: Methodology and distribution (1992) 492–518 27

[45] He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: Surpassing human-level performance on imagenet classification (2015) 29, 30

[46] TensorFlow: https://www.tensorflow.org/api_docs/python/tf/keras/initializers/. 30, 31

[47] Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In Teh, Y.W., Titterington, M., eds.: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Volume 9 of Proceedings of Machine Learning Research., Chia Laguna Resort, Sardinia, Italy, PMLR (13–15 May 2010) 249–256 30

[48] LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.R. In: Efficient BackProp. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 9–48 31

[49] Dräxler, S., Karl, H., Mann, Z.Γ.: Jasper: Joint optimization of scaling, placement, and routing of virtual network services. IEEE Transactions on Network and Service Management **15**(3) (2018) 946–960 33

[50] Yu, M., Yi, Y., Rexford, J., Chiang, M.: Rethinking virtual network embedding: substrate support for path splitting and migration. ACM SIGCOMM Computer Communication Review **38**(2) (2008) 17–29 33

[51] Magatao, L.: Mixed integer linear programming and constraint logic programming: towards a unified modeling framework. (2005) 35

[52] Dolati, M., Hassanpour, S.B., Ghaderi, M., Khonsari, A.: Deepvine: Virtual network embedding with deep reinforcement learning. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE (2019) 879–885 38, 39

[53] O'Shea, K., Nash, R.: An introduction to convolutional neural networks. CoRR **abs/1511.08458** (2015) 38

[54] Hongzi Mao, Mohammad Alizadeh, I.M.: Resource Management with Deep Reinforcement Learning. IEEE (2016) 39, 59

[55] Quang, P.T.A., Hadjadj-Aoul, Y., Outtagarts, A.: A Deep Reinforcement Learning Approach for VNF Forwarding Graph Embedding. IEEE Transactions on Network and Service Management **16** (December 2019) 39

[56] Yuan, Y., Tian, Z., Wang, C., Zheng, F., Lv, Y.: A Q-learning-based approach for virtual network embedding in data center. Neural Computing and Applications **32**(7) (2020) 1995–2004 39

[57] Chowdhury, M., Rahman, M.R., Boutaba, R.: Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. IEEE/ACM Transactions on networking **20**(1) (2011) 206–219 39

[58] Shahriar, N., Chowdhury, S.R., Ahmed, R., Khan, A., Fathi, S., Boutaba, R., Mitra, J., Liu, L.: Virtual network survivability through joint spare capacity allocation and embedding. IEEE Journal on Selected Areas in Communications **36**(3) (2018) 502–518 39

[59] Dehury, C.K., Sahoo, P.K.: Dyvine: Fitness-based dynamic virtual network embedding in cloud computing. IEEE Journal on Selected Areas in Communications **37**(5) (2019) 1029–1045 39

[60] Cheng, X., Su, S., Zhang, Z., Wang, H., Yang, F., Luo, Y., Wang, J.: Virtual network embedding through topology-aware node ranking. ACM SIGCOMM Computer Communication Review **41**(2) (2011) 38–47 39

[61] Yao, H., Chen, X., Li, M., Zhang, P., Wang, L.: A novel reinforcement learning algorithm for virtual network embedding. Neurocomputing **284** (2018) 1–9 39

[62] Sciancalepore, V., Yousaf, F.Z., Costa-Perez, X.: Z-TORCH: An automated NFV orchestration and monitoring solution. IEEE Transactions on Network and Service Management **15**(4) (2018) 1292–1306 39

[63] Xiao, Y., Zhang, Q., Liu, F., Wang, J., Zhao, M., Zhang, Z., Zhang, J.: NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning. In: Proceedings of the International Symposium on Quality of Service. (2019) 1–10 39

[64] Wang, H., Wu, Y., Min, G., Xu, J., Tang, P.: Data-driven dynamic resource scheduling for network slicing: A deep reinforcement learning approach. Information Sciences **498** (2019) 106–116 39

[65] Quang, P.T.A., Hadjadj-Aoul, Y., Outtagarts, A.: Evolutionary actor-multi-critic model for VNF-FG embedding. In: 2020 IEEE 17th Annual Consumer Communications and Networking Conference (CCNC), IEEE (2020) 1–6 39

[66] Wang, S., Bi, J., Wu, J., Vasilakos, A.V., Fan, Q.: VNE-TD: A virtual network embedding algorithm based on temporal-difference learning. Computer Networks **161** (2019) 251–263 39

[67] Dietrich, D., Papagianni, C., Papadimitriou, P., Baras, J.S.: Network function placement on virtualized cellular cores. In: 2017 9th International conference on communication systems and networks (COMSNETS), IEEE (2017) 259–266 59