University of BRISTOL

Application of Machine Learning in Heterogeneous Telecommunication Networks

Francesco Venerandi

May 2019

Final year project thesis submitted in support of the degree of

Master of Engineering in Electrical & Electronic Engineering

(H606)

Department of Electrical & Electronic Engineering

University of Bristol

DECLARATION AND DISCLAIMER

Unless otherwise acknowledged, the content of this thesis is the original work of the author. None of the work in this thesis has been submitted by the author in support of an application for another degree or qualification at this or any other university or institute of learning.

The views in this document are those of the author and do not in any way represent those of the University.

The author confirms that the printed copy and electronic version of this thesis are identical.

Signed:

Dated:

Acknowledgements

First and foremost, I would like to give a huge thank you to my project supervisor Dr Reza Nejabati for both giving me the opportunity to research such an exciting topic, and for his continued guidance over the past year. Secondly, I would like to thank Douglas Harewood-Gill for his numerous suggestions and for being a consistent source of counsel. I wish him the best of luck in pursuing his PHD. I would also like to thank my personal tutor Judy Rorison for being such a dependable source of encouragement and advice over the past four years.

Finally, I would like to thank my parents and brother for being so supportive of me throughout my undergraduate degree, I could not have done it without you.

Abstract

The strain put on modern-day networks is ever-expanding and traditional approaches are struggling to deliver the high quality of service that customers demand. Software Defined Networking (SDN) is an emerging network architecture which promises to revolutionise the industry with the implementation of more dynamic systems. SDN enables having a centralised controller which provides a convenient platform for state-of-the-art machine learning techniques to help manage complex networks. This research paper focuses on how a specific type of reinforcement learning technique, Q-learning, can be used within SDN to solve the shortest path problem within a given topology.

It explores how two methods of Q-learning, Q-tables and artificial neural networks, compare when written using the Python packages NumPy and TensorFlow. Specifically, the number of episodes taken for the algorithms to converge onto the shortest path is evaluated. The experimental results provide strong evidence that Q-learning has the potential to replace more traditional path finding algorithms (such as Dijkstra's) within a SDN architecture. Q-learning has the additional benefit of needing little to no prior knowledge of the network topology in a time when data privacy is increasingly important. Furthermore, the successful implementations of said algorithms within a SDN architecture provide a use case that reinforcement learning should be further investigated as it has the potential to play a critical role in the networks of the future.

Table of Contents

1 Introduction

1.1 Background

The number of IoT (Internet of Things) smart devices that will be connected to the internet is estimated to be twenty-one billion by 2025. As demand for an excellent quality of service on more devices increases, networks must be able to adapt to accommodate for these changes. Software Defined Networking (SDN) is an emerging network architecture which should be able to assist in this development. SDN allows for a more dynamic, cost-effective and manageable computer network by decoupling control from forwarding functions. This, amongst other things, allows the controller to be directly programmable leading to a plethora of additional benefits as will be discussed in chapter two. The shortest path problem is perhaps one of the most fundamental aspects of computer networking and this thesis explores how it can be solved within a SDN architecture. In its basic form, the shortest path problem consists of finding the shortest route between two nodes within a network. The term shortest can relate to different things depending on the type of network under consideration; for instance, it could mean the shortest physical distance between two points in a topology.

Machine learning has become a buzzword in recent years as people propose it to be the answer to almost every industry problem. Although this promise is unlikely to be entirely true, it is certainly very powerful and will definitely revolutionise the networking industry. Networks consist of huge amounts of data which humans have trouble making sense of. Machines however, can easily sift through this data and find patterns ranging from the optimum route to take through a network, to discovering anomalies and so mitigating potential threats [1]. However, algorithms may be able to learn the solution to such problems without the use of such data.

1.2 Research Problem

Only recently has reinforcement learning (a type of machine learning that will be explained in chapter three) become a viable method in the aiding of computer networks [2], [3]. This has become possible with increased computational power along with easy to implement artificial neural networks.

Both supervised and unsupervised learning provide powerful methods to solve the shortest path problem within a network. However, many people are becoming more concerned with data privacy in recent years following such events as the Cambridge Analytica Scandal, and so, methods which do not rely on data collection and usage are more appealing. This is where a reinforcement learning approach is attractive because in theory, the application could be completely model-free and so not use any pre-existing knowledge to route packets to their intended destinations.

1.2.1 Aims and Objectives

The primary aim of this project was to investigate how feasible a reinforcement learning, specifically a Q-learning algorithm, would be in finding the shortest paths within a network topology. The algorithms were first tested on a simple topology, and then it was seen if they would be scalable for a more complex network. The same applications were built using two different Python packages, NumPy and TensorFlow, to see how they affected performance. Once it had been confirmed that they could successfully find the shortest paths, one was implemented within a simulated Mininet environment using a Ryu controller. These mentioned software packages will be explained in more detail in chapter three.

1.3 Thesis Structure

The structure of this thesis is organised as follows. Chapter two introduces the concept of Software Defined Networking, what it is and how it works. It then discusses the preferred methods regarding shortest path finding. Chapter three gives an overview of machine learning and explains the three

types: supervised, unsupervised, and reinforcement. It then goes into further detail on reinforcement learning and how specifically a generalised Q-learning algorithm works. Finally, it describes the different software packages used throughout the project. Chapter four starts by outlining the implemented Mininet and Ryu controller architecture and then introduces the network topology that was used to test the various algorithms. It proceeds to describe the implementation of the four separate Q-learning algorithms that were created. Chapter five gives a comprehensive summary of the results obtained from testing both the overall architecture, and each individual algorithm. Chapter six investigates where potential future research should be directed and finally, chapter seven concludes the thesis and research project.

2 Software Defined Networking

2.1 Introduction

The following chapter explores Software Defined Networking. It then goes into more detail on the shortest path problem which is at the basis of this thesis.

2.2 Overview

Software Defined Networking (SDN) was born out of Stanford University's Ethane project [4] in 2007 and by its primary adoption in 2011, had developed into an emerging network architecture which was manageable, adaptable and cost-effective. With the ever-increasing number of devices needing connectivity and the rise of cloud computing, complex traffic patterns continue to require more agile networks. SDN's defining feature was the decoupling of the control plane from the data plane.

The control plane can be broadly defined as the place where traffic routing decisions are made. The data plane (sometimes known as the forwarding plane) then forwards this traffic to the next hop in the network, according to the instructions given by the control plane. SDN acts to centralise the control plane allowing a single software program to regulate multiple data-plane elements. There are many benefits of SDN compared to traditional networking in an era where the IoT is placing more and more strain upon networks. Firstly, consolidation of network management becomes possible which allows all of the networking components to be operated via a single device. This centralised controller is directly programmable by the operator, allowing more flexibility within the network. Secondly, SDN enables the administrator to have a global view of the network allowing traffic flow to be easily controlled. This brings with it the additional benefit of being able to distribute security and policy information consistently throughout the network. Finally, analysis of traffic can easily uncover and mitigate threats and so SDN increases network security. However, this centralised security brings with it the risk of having a single point of failure. Therefore, it is important that security measures are kept up to date. **Figure 2-1** below shows

a graphical representation of the separation between control and forwarding planes.

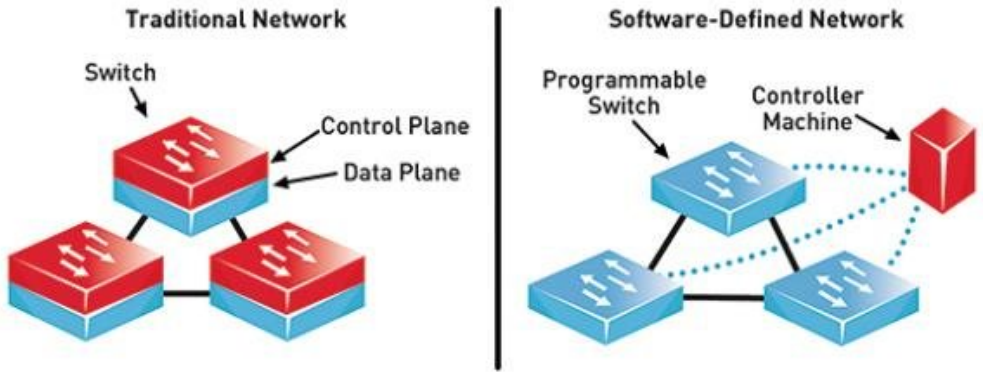

***Figure 2-1**: Traditional Vs SDN Network Topology [5]*

2.3 OpenFlow

OpenFlow [6] originated from the same Ethane project as SDN but became more widely adopted in 2012. OpenFlow, an instance of the SDN architecture, is a protocol that separates the control of a switch from the switch itself. A switch is simply an abstract packet processing machine which receives packet instructions from the central controller. In essence, the OpenFlow protocol sends and receives messages to and from an OpenFlow switch. These messages collectively allow a SDN controller to control network traffic.

An OpenFlow switch, shown in **Figure 2-2**, comprises of the following three core functions which collaboratively enable it to work:

1. Flow Tables: Flow tables are a subset of forwarding tables. They may use the information within a packet to decide its next hop destination. Examples of information used are incoming switch port, TCP port / IP address etc.
2. Controllers: The controller can set up flows through the network based upon a number of different characteristics including: number of hops, latency and jitter. It is up to the network designer to decide how packets are routed and this becomes more manageable when implemented within a SDN architecture.

3. The OpenFlow Protocol: The OpenFlow enabled controller functions as a type of operating system for the network. Controllers talk to the switches via the OpenFlow protocol and so impose policies on flows.

In addition to having one or more flow tables, the switch also consists of group tables. These enable more complex packet operations to be executed on multiple packets at once.

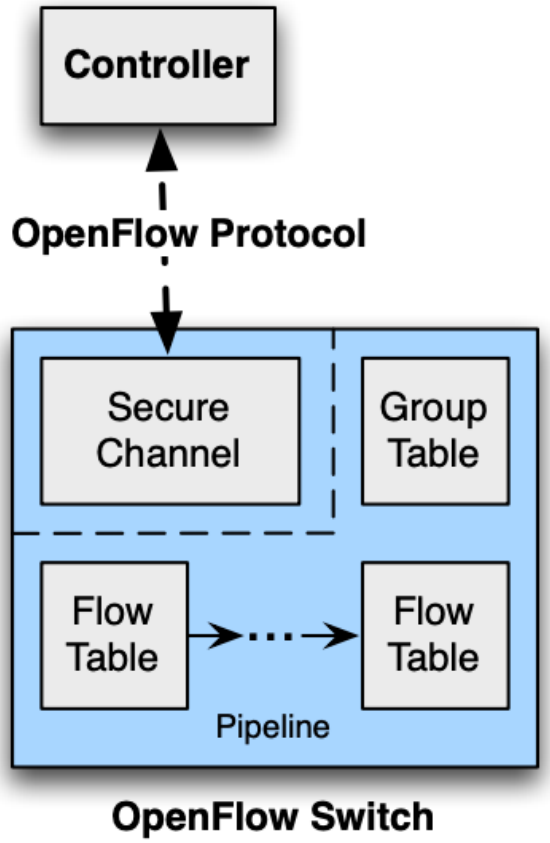

Figure 2-2: *OpenFlow Switch [7]*

2.4 The Shortest Path Problem

2.4.1 Overview

The shortest path problem, put simply, requires finding of the shortest path between two nodes in a graph. The terms shortest and graph, can have different meanings depending on the problem specification. For instance, it could mean the shortest physical distance in kilometres between two places on a map, or the shortest route by car from point A to B through an urban

cityscape. It could also mean the shortest amount of time taken for a packet to be sent between two switches over an internet network. In the same way that there are only certain routes that a car driving through a city can take to get between destinations, i.e. roads, there are only certain links that exist between switches that packets can take. When it comes to modelling graphs consisting of nodes, there are a few different scenarios that may arise. Firstly, the graph may be unweighted, as seen in **Figure 2-3**. This essentially means that the weight between each node in the network is zero (this is an unrealistic assumption to make in the real world). Therefore, the shortest path between *A* and *C* in **Figure 2-3** is the path with the least number of hops i.e. *A* → *C* with a hop count of one. **Figure 2-4** shows an example of the same network but using non-negligible weights. If the weights of the paths between *A* and *C* are summated, the shortest path becomes *A* → *B* → *C* with an overall weight of two (although its hop count is three). The final option can be seen in **Figure 2-5**; this network also includes negative weights and so requires a different algorithm to find the shortest path. (An example of a negative weight in the real world looking at Figure 2-5; a chauffeur gets paid to drive their employer from *A* to *D*, but lives at location *B*. Therefore, their journey to work from *B* to *A* incurs a cost and so it has a positive weight. But once at work, the chauffeur gets paid for driving to *D* and hence this is shown as a negative weight). **Figure 2-5** therefore shows the shortest path to be *A* → *D* → *C*, with a weight of one.

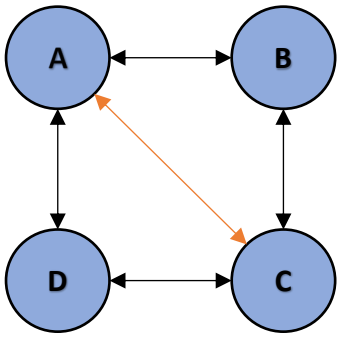

Figure 2-3: *Unweighted Network*

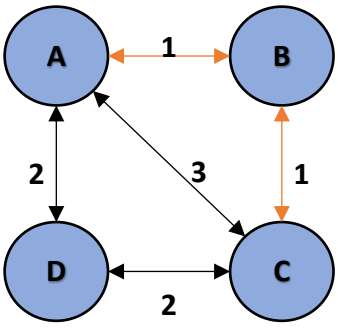

Figure 2-4: *Positive Weighted Network*

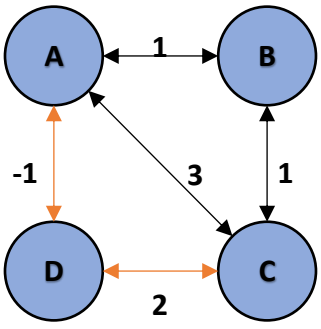

Figure 2-5: *Negative Weighted Network*

These are very simplistic graphs; in reality, there is a very high upper limit on the number of nodes and connections within a network. It is therefore

important to have ways of finding the shortest path in a network that is realistic both in terms of time taken and computational intensity.

2.4.2 Breadth-first Search

Breadth-first search (BFS) is perhaps the simplest method, and so it fails to work well in a complex network. The algorithm traverses all the nodes within the network and updates its queue (which is a list of nodes it hasn't visited). Once it visits a node, it removes it from the queue and adds the nodes' children to the queue (assuming these children nodes have not been previously visited). Once the queue is empty, the algorithm knows that it has explored the entire graph.

2.4.3 Dijkstra's Algorithm

Dijkstra's algorithm is a commonly used method which enables finding of the shortest path between any chosen node and every other node in the graph. Dijkstra's uses a form of the BFS to solve this problem. The algorithm works as follows:

1. A shortest path tree set is created (this is simply a list which keeps track of nodes in the network and their distances from the starting node).
2. Assign a distance of zero to the starting node *A*, and a distance of infinity to all other nodes in the set.
3. While the set does not include all the distances:
    a. See what the distances of the connected nodes are from *A* and pick the shortest one not already in the set, for example *B*.
    b. *B* is then added to the set and the distance values of the nodes connected to *B* are updated. Distances are updated only if the following criteria is met:
        i. *If the sum of distance values from A* ***and*** *the weight of the connection A → B is less than the distance value of B.*

An example of Dijkstra's algorithm in operation is described below shown in the starting state **Figure 2-6** .

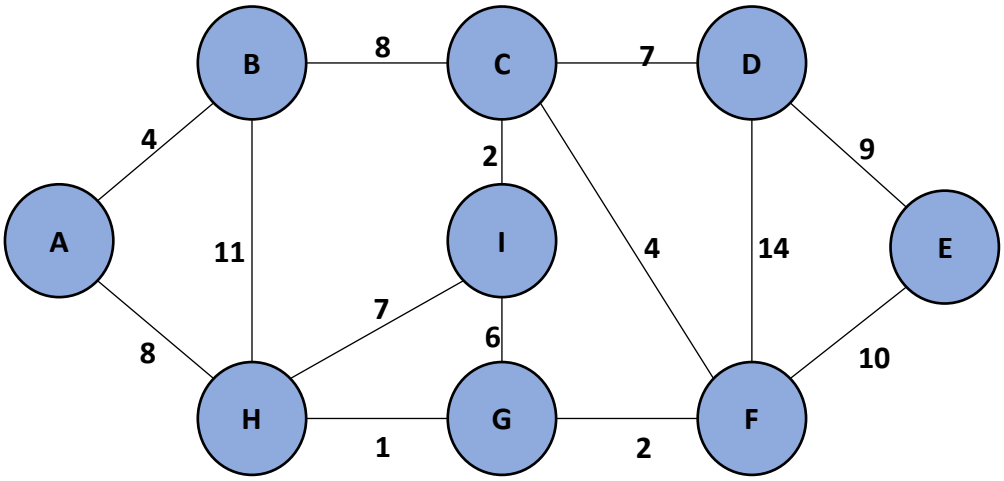

Figure 2-6: *Dijkstra's Algorithm start*

The set is initially empty. A is the starting node and so is set to zero and the other nodes in the set are set to infinity; $A = [0, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty]$. A is selected as it has the shortest distance of zero; $A = [0]$. The adjacent nodes from A are; B and H with distances of four and eight respectively. B is picked as its distance is shorter; $A = [A, \mathrm{B}]$. The nodes connected to B; C and H are now inspected. C now has a distance of twelve from A, but H only has a distance of eight, therefore H is picked; $A = [A, \mathrm{B}, \mathrm{H}]$.

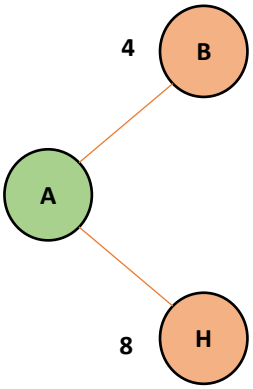

Figure 2-7: Dijkstra's Algorithm step 1

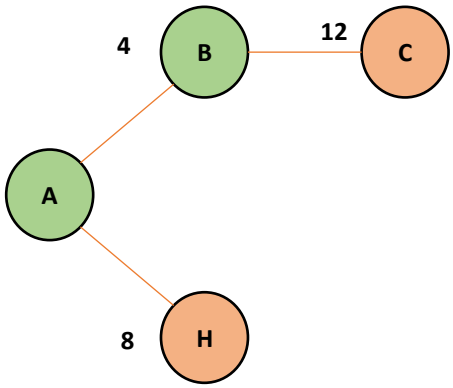

Figure 2-8: Dijkstra's Algorithm step 2

This process is repeated from node H, updating subsequent nodes with their distance from A as the algorithm works through the graph.

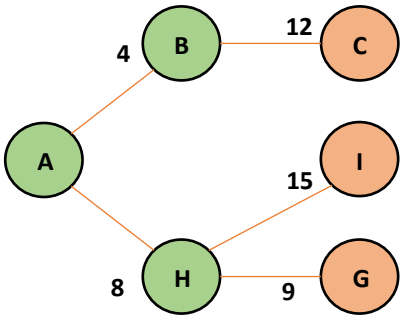

Figure 2-9: Dijkstra's Algorithm step 3

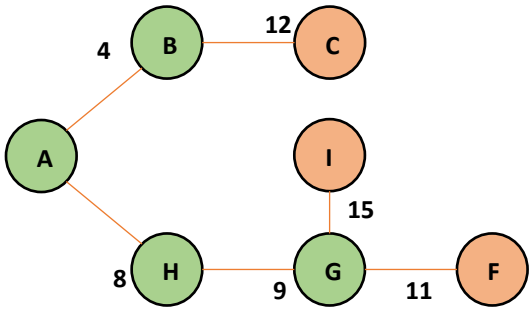

Figure 2-10: Dijkstra's Algorithm step 4

Figure 2-7, Figure 2-8, Figure 2-9 and Figure 2-10 show the first four steps of this algorithm in action. The nodes are highlighted in green once they have been visited. Figure 2-11 shows the completed algorithm once it has traversed the graph and every node is accounted for in the set. It shows the shortest path from $A \rightarrow E$ being $A \rightarrow H \rightarrow G \rightarrow F \rightarrow E$ with a distance of twenty-one.

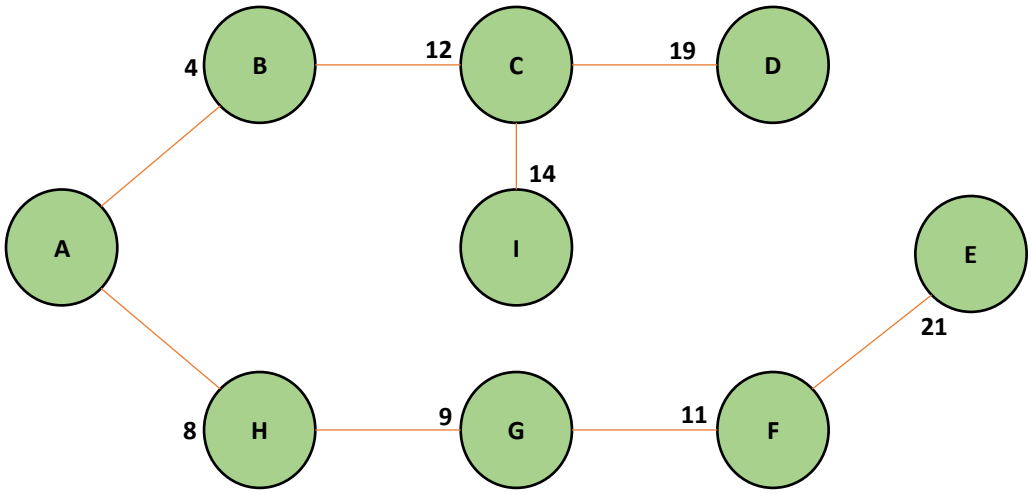

Figure 2-11: *Dijkstra's Algorithm end*

Where Dijkstra's algorithm fails however, is in a network which has negative weights. This is because the algorithm gets stuck in a loop where it constantly updates the path with a negative weight and so decreases the path cost to negative infinity.

2.4.4 Floyd-Warshall's Algorithm

The Floyd-Warshall algorithm (FWA) solves the shortest path problem by utilising dynamic programming. Dynamic programming refers to a problem-solving approach which stores related and simpler sub-problems. A negatively weighted edge network can be solved with FWA using the following approach: the shortest path from $A \rightarrow C$ can either be the shortest path from $A \rightarrow B + B \rightarrow C$, or simply the shortest path from $A \rightarrow C$ that has already been found. As seen in Figure 2-12, if each node is taken individually, the route from $A \rightarrow C$ includes a negative weight (negative four). However, when the two routes are summed together, the overall cost is positive (one). FWA takes advantage of dynamic programming in order to overcome the negative weight problem.

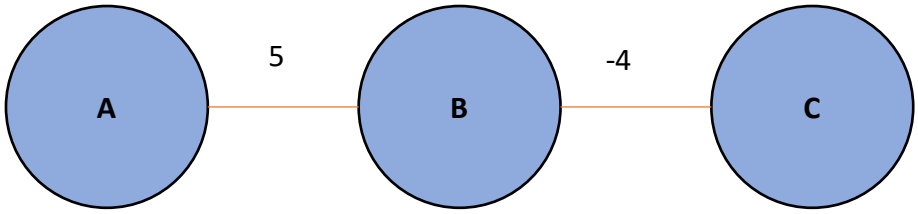

Figure 2-12: *Example Network*

3 Machine Learning

3.1 Introduction

This chapter introduces the different variations of machine learning techniques before going into more detail on Q-learning; the chosen method of machine learning utilised for this project due to its model-free implementation. It concludes with a brief summary of the software packages/applications which were used throughout the project.

3.2 Overview

Machine learning (ML) has gained a lot of attention in recent times due to its ability to perform tasks which were once thought to be impossible. Although ML can be traced back to the first discoveries of statistical methods and the invention of the neural network by neurophysiologist Warren McCulloch and mathematician Walter Pitts in 1943 [8], adoption was slow due to comparatively weak computer processing power. By the late 20th century however, great leaps in ML had been made which accumulated in IBM's Deep Blue beating world chess champion Garry Kasparov [9] in a game of chess.

ML is today used in almost every industry as it has the capability of performing both hugely complex tasks, as well as learning more mundane tasks and so potentially replacing human workers. Machine learning today encompasses a variety of different techniques; however, they all essentially have the same aim which is for a computer algorithm to learn to perform a certain task without being given explicit instructions. These tasks can range from recognising faces and road signs, to independently learning and executing operations simply by interacting with a previously unseen environment.

3.3 Types of Machine Learning

ML can be roughly separated into three categories; supervised learning (SL), unsupervised learning (UL) and reinforcement learning (RL).

SL uses trained data that contains both the input and the desired output and learns to map the two. For example, a SL algorithm could be fed images of cats with the label '*cat*' and other images with the label '*not cat*'. Over time, the algorithm learns to associate certain characteristics with the label '*cat*'. When presented with an unseen image of a cat, it would be able to correctly classify the image.

UL is trained on unlabelled data which contains only the input. The algorithm can learn patterns in the data which it uses to map inputs to outputs. For example, an UL algorithm could be fed images of different animals in which it finds patterns relating to '*whiskers*' or '*four-legged*'. When presented with an unseen image of a cat, it could match these properties with other similar animals it has trained on and therefore group them together.

Finally, RL is the type of machine learning which this project explores. A RL algorithm learns through experience in the form of positive or negative reinforcement in order to perform the desired action. As a RL agent explores an environment, it is given rewards depending on the actions it takes within that environment. Over time, the algorithm will independently learn the best actions to take as they lead to the highest reward.

3.3.1 Deep Learning

Deep learning (DL) is a sub-field of ML which is inspired by the human brain and so uses algorithms called artificial neural networks (ANN) to propagate learning. DL can be implemented in conjunction with the three aforementioned types of ML to make them more powerful. It was the breakthrough that has allowed applications to handle huge amounts of data and is the future of ML. It is referred to as deep because the neural networks contain various hidden layers; each performing different mathematical functions that enable learning. A paper by F. Richard Yu and Ying He gives a comprehensive review of both RL and DL [10].

DL gained traction in 2015 when Google's AlphaGo [11] algorithm developed by its DeepMind team beat a human professional Go player. Go is a strategy board game which has an estimated lower-bound of 2×10^{170} legal board positions (the observable universe contains 1×10^{80} atoms). It was the implementation of DL within a RL algorithm which allowed AlphaGo to perform so well.

3.3.2 Artificial Neural Networks

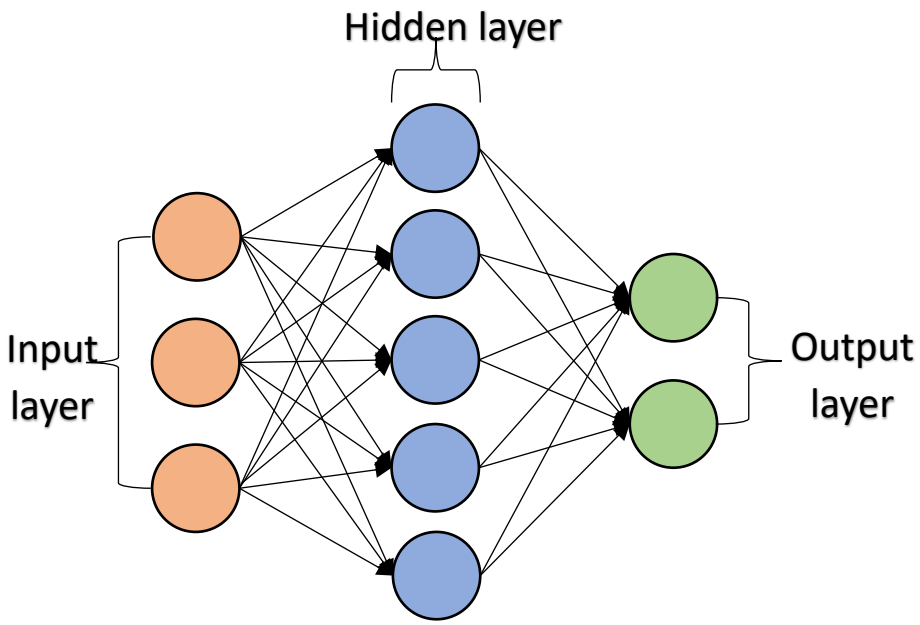

Figure 3-1: *Artificial Neural Network*

An ANN can be described visually as seen in Figure 3-1 and consists of at least three layers; the input layer, the hidden layer(s) and the output layer. The input layer permits data to be fed into the model of the system for further processing. The hidden layer(s) takes a set of weighted inputs, runs them through a certain activation function, and outputs the result. There can be several hidden layers in a model, each one performing a different mathematical operation. Finally, the output layer is the layer of neurons that produces the desired outputs. The arrows between each node in Figure 3-1 represent what are known as edges. These each hold a specific weight which is updated as the model trains. By the end of this training period, the edges with the highest weights represent the most reinforced paths; much like

how the human brain learns by building stronger connections between neurons in the brain. Figure 3-2 shows additional hidden layers, which turns the network into a deep neural network.

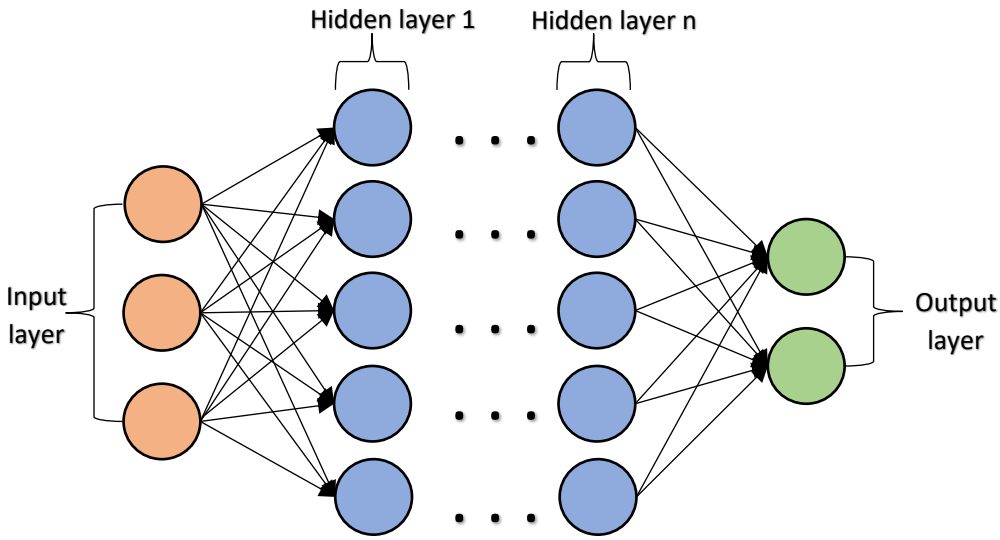

Figure 3-2: *Artificial Neural Network with Hidden Layers*

3.3.3 Q-learning

Q-learning (QL) is a model-free method of RL; meaning that it does not need any prior information to learn from. For any finite Markov Decision Process (MDP), QL can find the action which maximises the total reward over all future steps. A MDP is simply described as decision-making where decisions are partly random, and partly chosen by the decision maker. Figure 3-3 shows the flow chart for QL which can be explained as followed.

An agent starts at state $\boldsymbol{s_t}$ and takes the action $\boldsymbol{a_t}$ which has the highest cumulative future reward $\boldsymbol{r_t}$. It receives the reward $\boldsymbol{r_t}$ and progresses to the next state $\boldsymbol{s_{t+1}}$. The value of this reward is used to update what is known as a Q-table; this is simply somewhere where the algorithm can store the values of rewards for each state-action pair (Q-tables will be described in more detail later on). This process can also be described mathematically as shown in a version of the Bellman Equation (
Equation **1**).

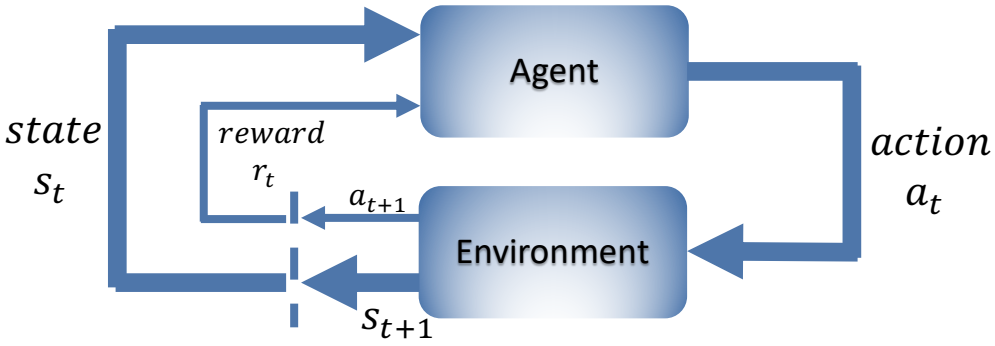

Figure 3-3: *Q-learning Flow Chart*

Equation 1: *Q-learning Equation*

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[\,r_{t+1} + \gamma\, maxQ(s_{t+1}, a_t) - Q(s_t, a_t)]$$

The maths behind this equation can seem quite complicated and so the main features will be described along with the overarching result and how it is used in Q-learning.

The agents' objective is to pick the policy that maximises the total reward over the training period. Assume that the agent is initially at state s_t within the environment:

At time $t = 0$, the agent picks action a_t, progresses to state s_{t+1} and receives the reward r_{t+1}.
At time $t = 1$, the agent picks action a_{t+1}, progresses to state s_{t+2} and receives reward r_{t+2}.

This process is repeated until the agent has completed its training of the environment and so the total cumulative reward is given as:

$$total\ reward = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} \ldots$$

However, this is not an accurate representation because it gives equal probability to all rewards. As time increases, the chance of something happening within the environment that could change the value of a future

reward increases; therefore, rewards in the future should have less of an effect on the decisions of the agent. A discount factor of γ is applied to discount rewards in the future. The value of γ is between zero and one and can be chosen by the designer to fit the specific environment.

$$\boldsymbol{total\ reward} = \boldsymbol{r_{t+1}} + \boldsymbol{\gamma(r_{t+2})} + \boldsymbol{\gamma^2(r_{t+3})} + \boldsymbol{\gamma^3(r_{t+4})} \ldots = \sum_{\boldsymbol{i=1}}^{\boldsymbol{T}} \boldsymbol{\gamma^{i-1} r_i}$$

The notion of a value-function; $\boldsymbol{V(s)}$ is often used in RL. This represents how good a state is based upon the possible cumulative future rewards that can be received from that state. Among all the possible value-functions that exist, there is one; $\boldsymbol{maxV(s)}$ which has the highest value and therefore relates to the most preferred action to take.

One other function; $\boldsymbol{Q(s,a)}$ can be defined for convenience which describes the state-action pair and it is this which populates the Q-table. Therefore, it can be seen that $\boldsymbol{\gamma\ maxQ(s_{t+1},\ a_t)}$ in
Equation **1** represents the action $\boldsymbol{a_t}$ taken by the agent from state $\boldsymbol{s_{t+1}}$, which gives it the maximum possible future reward. Finally, α represents a hyper-parameter known as the learning rate. This can have a value of between zero and one (as chosen by the designer) and represents how quickly new information overrides old information. This needs to be set carefully in order that the Q-values converge at a sensible rate (optimal learning rates are themselves an active area of research).

The Q-table is updated at each step of training with the new Q-value calculated using
Equation **1**. As time increases, these values converge once the agent has found the optimal policy and this same policy is then repeated over and over i.e. reinforced. However, this presents a problem; how does the agent know the policy it is reinforcing is the policy which yields the highest Q-value? This is where the exploration-exploitation parameter ε comes into play.

Initially, the agent should explore the environment randomly; $\varepsilon = 1$, so as to find as many state-action pairs as possible. Over time, the value of ε is reduced and the agent begins to exploit the best policies. The value of ε should stop decreasing around a small number, usually $\varepsilon = 0.1$. This is so the agent exploits the best policy most of the time but there is still a small probability that it explores alternative policies which may yield a higher Q-value.

Figure 3-4 below illustrates this problem in a form which may be easier to understand. The cat explores the environment randomly and finds the single bottle of milk on its first try. A classical reinforcement learning algorithm would then associate this path (shown in orange) as the optimum path yielding the highest reward. So, in all subsequent episodes, the cat follows this path. However, there may be somewhere else in the environment which contains two bottles of milk; if the algorithm simply exploits the path with the highest Q-value, it will never know that there is a different path which yields a higher reward. Therefore, it makes sense for the cat to exploit a path which it knows has a high reward most of the time, but sometimes explore the environment to see if it can find a higher reward.

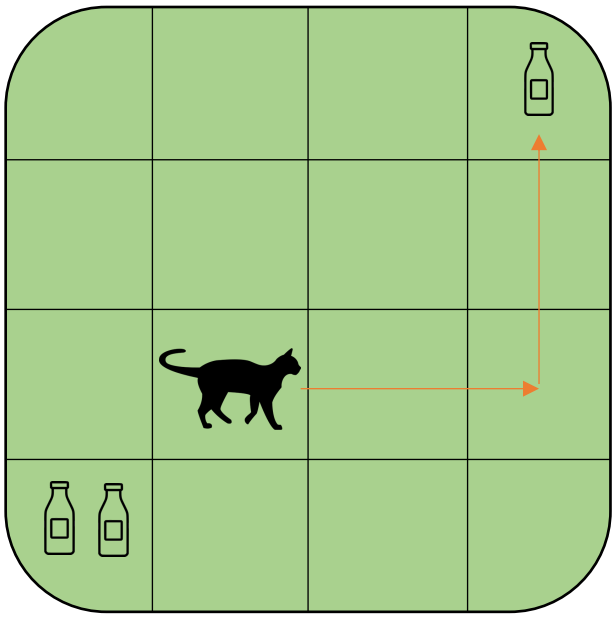

Figure 3-4: *Exploration vs Exploitation Dilemma*

This exploration vs exploitation is very relevant for a network topology. Networks are ever-changing environments; links between nodes may sometimes break or new nodes with additional links may be added. The agent should be allowed to sometimes explore this environment so as to adapt to the changing topology. The value of what ε should be within a SDN architecture is a research project on its own.

3.4 Software Packages / Applications Used

3.4.1 Mininet

Mininet [12] is a software emulator used for rapid prototyping of networks on a single machine. It provides an extensible Python API and can be used to simulate realistic network topologies that use OpenFlow enabled switches. The virtualised switches and hosts act like real devices within a network and allow computer-generated traffic to be sent between them. Mininet has a host of useful inbuilt commands which can be used to analyse the network such as *iperf* and *pingall*. It was used in this project to create a simulated environment and test connections between switches running OpenFlow. It was installed on a virtual machine running Linux which enabled it to emulate the network locally on a laptop / PC.

3.4.2 Ryu SDN Controller

In addition to Mininet, in order to complete the virtual SDN architecture, it was required to leverage an OpenFlow controller framework. There were several external controller options to choose from, each having its own benefits ranging from the level of documentation and community support, to which programming language it was coded in. Ryu [13] was the chosen controller as it could be written in Python and fully supported the latest version of OpenFlow.

3.4.3 NetworkX

NetworkX [14] is a well-designed Python software package which can be used for the creation and manipulation of complex network architectures. It has integrated functions such as one which finds the shortest path

between any two nodes within a network. This function was used to check the accuracy of the created Q-learning algorithms.

3.4.4 NumPy

NumPy [15] is a package which works alongside Python and provides the ability to modify high-performance multi-dimensional arrays. The first two of the four Q-learning algorithms were written using NumPy.

3.4.5 TensorFlow

TensorFlow [16] is an all-purpose programming package which was developed by the Google Brain Team in 2015. It uses dataflow graphs for numerical computation and operations which change shared states. TensorFlow's most important function allows modelling of deep artificial neural networks, i.e. it is a very powerful machine learning framework. TensorFlow provides a suitable front-end API written in Python whilst executing applications in the more high-performance language of C++. It was used to create the second two Q-learning algorithms.

4 Design and Methodology

4.1 Introduction

Chapter four begins by describing the overall SDN architecture used before going into further detail on the chosen network topology. It then explores the designs of the various Q-learning algorithms.

4.2 Network Architecture

The overarching design of the project is shown visually in Figure 4-1. The Ryu controller accesses the appropriate Q-learning algorithm which exports the shortest path from each switch to every other switch in the network topology. This exported list of shortest paths is then used by the controller to route packets to their intended destination.

Existing code made available by the grotto-networking team [17] was used to create the architecture. It was modified by the author to work with the desired network topology and the Q-learning algorithms were then embedded within the Ryu controller code. The original application implemented NetworkX to find the shortest paths in the network; this function was replaced with the appropriate Q-learning application. It was important to export the list of shortest paths in the same format as NetworkX's, in order for the system to function as desired.

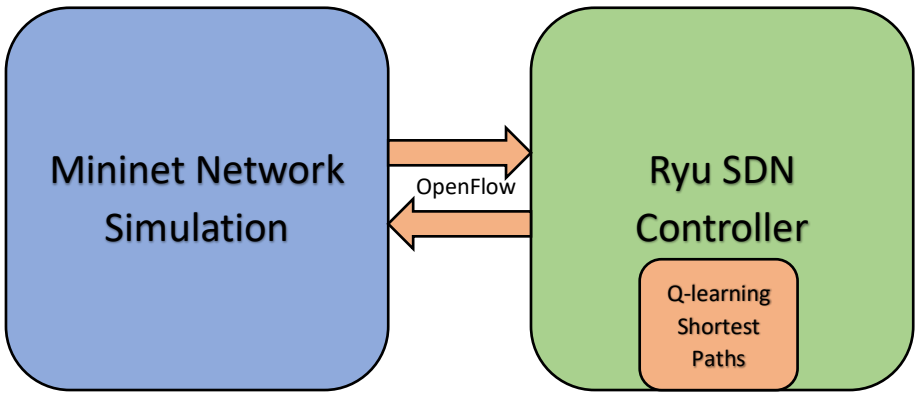

Figure 4-1: *Overarching SDN Network Architecture*

4.3 Network Topology

The only required design criterion when creating the network topology was that loops had to be present within the architecture. A loop within a network exists when there is more than one path that can carry information from the same source to destination. The COST 239 pan-European Network was chosen due to its real-world application and can be seen in its two forms in Figure 4-2 and Figure 4-3. The numbers on the links represented the distance in kilometres between cities based on physical cable length. This distance could have been modelled in a variety of ways; for example, it could have been represented as delays in milliseconds that switches would wait before sending a packet. This would relate to the time taken for a packet to reach its intended destination in the real-world topology.

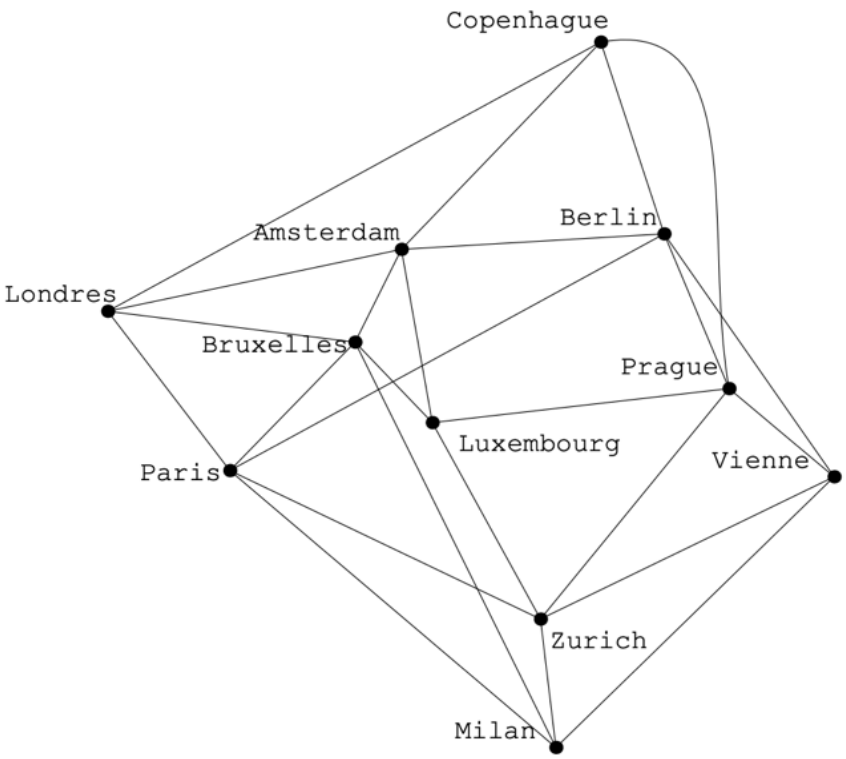

Figure 4-2: *COST 239 pan-European Network a [18]*

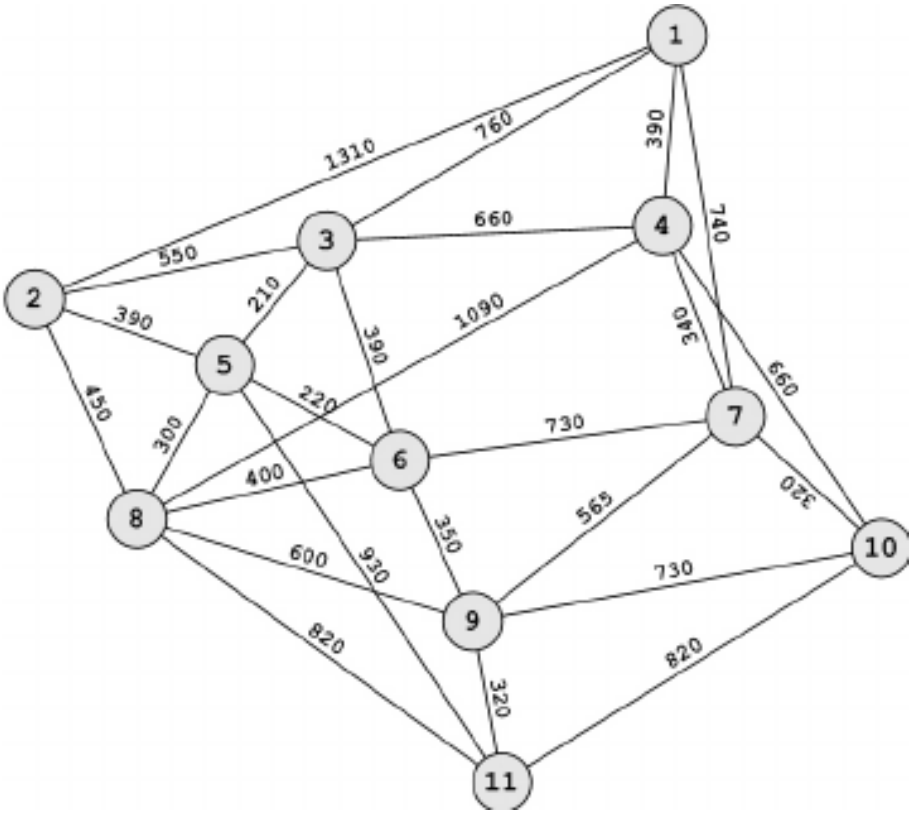

Figure 4-3: *COST 239 pan-European Network b [19]*

The network topology was created using a program developed by the grotto-networking team called ComNetViz [20]. This web application allowed the design of a topology with the following features:

1. The ability to add hosts that have their own discrete MAC address port number.
2. The ability to add switches, both connected to the hosts and to other switches via output ports.
3. The ability to add links between host-switch and switch-switch consisting of both weight and capacity. Weight related to the delay placed on each link in milliseconds and capacity to the amount of data that could traverse the link.

Once created, the topology was exported as a .JSON file which could be saved onto the virtual machine and run together with the entire architecture. The final topology can be seen in Figure 4-4. Note that the link capacities have been reduced so as to display just the structure of the topology. If one loads the actual .JSON file, they will see the links between

switches as being much thicker than in Figure 4-4. These capacities related to how much traffic each link could handle. As this was not the direct scope of the project, all the capacities between switches were set to 1000Mbps (the maximum that Mininet could handle).

It was important that for the network to perform correctly, each host had its own individual MAC address which could identify it from the others. (A MAC address can be thought of as an IP address for hardware which uniquely identifies all devices within a network). Another useful built-in application of ComNetVis was its '*path*' function. This executed '*Shortest Paths*' which found the shortest path between a host and all the other hosts within the topology and exported these features as a .JSON file. This file provided yet another way to check the accuracy of the algorithms detailed in the following section.

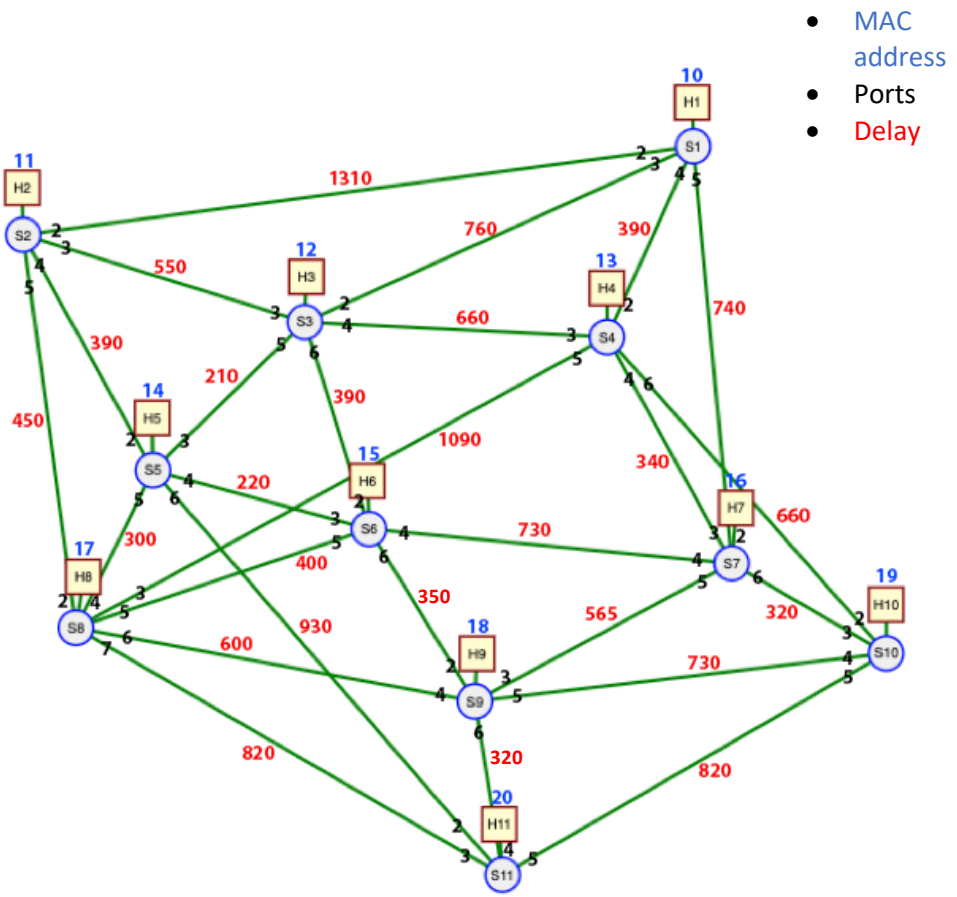

Figure 4-4: *Final Network Topology*

4.4 Algorithm Design

4.4.1 Overview

There were four separate applications created for this research project. They relate to the two coding packages used: NumPy and TensorFlow, and the two topologies used: unweighted and weighted. The reinforcement learning section of the algorithm was based on a series of tutorials by Arthur Juliani [21] and the topology setup was based on a blog post by The Beginner Programmer [22]. Although each of the applications seem quite different, they all follow a similar procedure which can be described in the following pseudocode. (A flow chart depicting the same process can be found in Appendix B).

Algorithm 1: Q-learning base algorithm

```
1: GET: List of switches and weights from the network topology
2: CREATE: R-Matrix of rewards
3: CREATE: Q-Matrix of zeros
4: INITIALISE: e, as a number close to 1
5: CHOOSE: Starting state
6: for: number of episodes
7:     CHOOSE: current state
8:     if random (0,1) < e then
9:         CHOOSE: random next state
10:    else
11:        CHOOSE: next state which yields the highest Q-value
12:    end if
13:    CALCULATE: Q-value using Bellman's equation for RL
14:    UPDATE: Q-Matrix
15:    REDUCE: e, stop at small value e.g. e= 0.1
16: end for
17: OUTPUT: Shortest path
18: REPEAT: For all starting switches
```

The network topology for all four algorithms needed to be converted into a matrix which the algorithm could access to find a list of possible next moves. This was done by creating an R (reward) matrix of size $(number\ of\ switches)^2$. Negative ones were placed into the illegal moves and zeros into the positions which connected switches. An example of this matrix can be seen in Figure 4-5. Finally, a large reward e.g. one hundred, was placed in the position that related to the final move reaching the goal state.

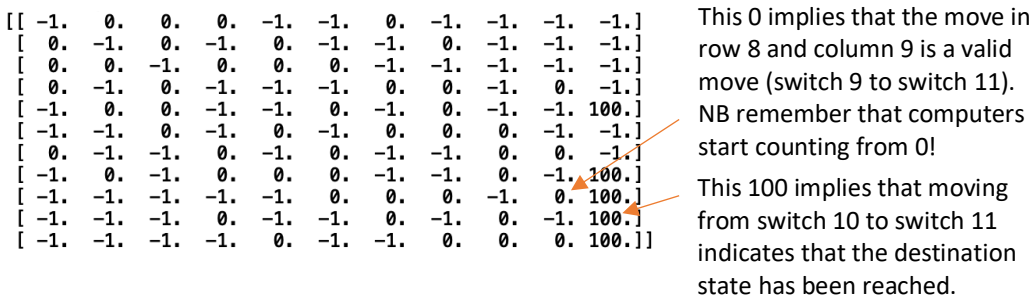

Figure 4-5: *Generic Reward Matrix*

Once training had been completed, the Q-Matrix could be inspected to find the shortest path from start to finish.

4.4.2 Unweighted Vs Weighted, NumPy

There was one main difference between the unweighted and weighted versions of the Q-learning algorithm when implemented in NumPy. This materialised when calculating Q-values for a state-action pair. The unweighted algorithm favoured shortest path finding by number of hops from start to finish. Looking at Figure 4-4, it can be seen that there are seven possible shortest paths from $S1 \rightarrow S11$ with an overall length of three hops. (For example: $S1 \rightarrow S7 \rightarrow S9 \rightarrow S11$). The unweighted algorithms gave a higher Q-value to paths with three hops than to those with more than three. Therefore, once the algorithm had finished its training, the completed Q-Matrix held all of the information needed to find the shortest path. This was done by firstly finding the largest value in the first row of the Q-Matrix (which indicates the best move to take from starting state $S1$). The position

of this value in the matrix related to the next hop that the agent should take. This process was repeated until the final state was reached.

When using the weighted topology however, the weights were factored into the Q-equation and so paths which had a larger weight gave higher Q-values (i.e. smaller Q-values, as opposed to larger ones, indicated shorter paths). Figure 4-6 shows an example of a Q-Matrix upon completion of all its training episodes within a weighted network topology. When working out the shortest path, the smallest value at each hop was chosen as it related to the best action to take from that state. This path is shown in Figure 4-6 highlighted in orange. Starting at row zero, column three has the smallest value (not including illegal moves of negative one) and so the agent makes the move $S1 \rightarrow S4$ (remember that computers start counting from zero and hence row zero relates to switch one). Row three would therefore be the next row inspected and so on. The final path is therefore $S1 \rightarrow S4 \rightarrow S7 \rightarrow S9 \rightarrow S11$.

	0	1	2	3	4	5	6	7	8	9	10
0	-1	32033	31753	31471	-1	-1	31477	-1	-1	-1	-1
1	32673	-1	31388	-1	30927	-1	-1	30906	-1	-1	-1
2	31914	30760	-1	31716	30591	30590	-1	-1	-1	-1	-1
3	31912	-1	31328	-1	-1	-1	30870	31521	-1	31159	-1
4	-1	31281	30635	-1	-1	30387	-1	30458	-1	-1	30437
5	-1	-1	30746	-1	30362	-1	31364	30522	30057	-1	-1
6	31389	-1	-1	30919	-1	31004	-1	-1	30354	30423	-1
7	-1	30925	-1	32462	30246	30658	-1	-1	30374	-1	30246
8	-1	-1	-1	-1	-1	30436	30421	31079	-1	30919	29540
9	-1	-1	-1	31855	-1	-1	30606	-1	30566	-1	30267
10	-1	-1	-1	-1	31702	-1	-1	30704	29538	30977	29085

Figure 4-6: *Q-Matrix Upon Episode Completion*

Interestingly, the algorithm found this path even though it had an extra hop in it; because the overall cost was lower. This can be seen graphically in Figure 4-7, the red path has a cost of 1615 compared to the cost of the blue path at 1625.

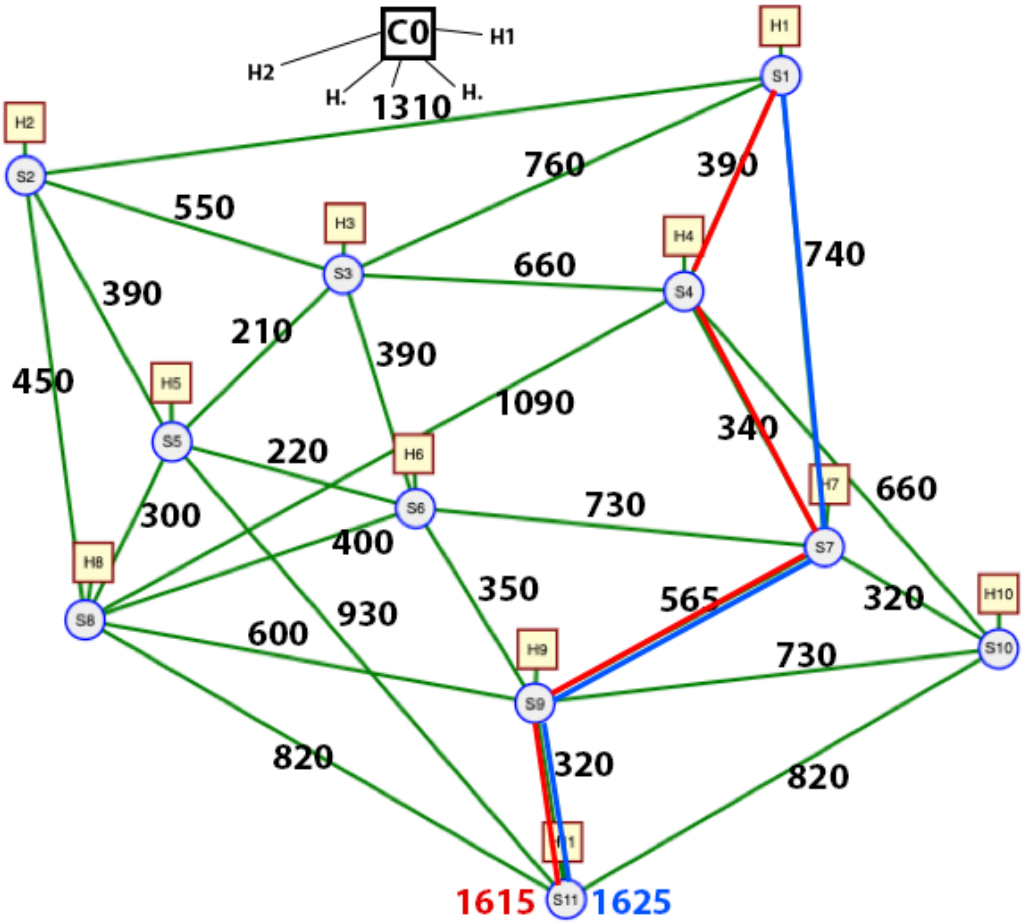

Figure 4-7: *Shortest Path Routes*

4.4.3 NumPy Vs TensorFlow

The key difference between these two methods was that the TensorFlow algorithms used an artificial neural network to store the Q-values instead of a Q-table. This had the advantage of being scalable for more complex networks as a large Q-Matrix would not have to be stored in memory. However, these applications took far longer to train due to the increased computational complexity needed to create and update the neural networks. The two applications written in TensorFlow were not implemented into the overall architecture as they only found the shortest path from one start to one end node. Loops could have been implemented into the code (as it was done for the NumPy applications) to find all the shortest paths for all nodes in the topology. However, the training time would be excessive and so it was enough to show functionality of the two algorithms for a single start/end path.

4.4.4 TensorFlow Weighted

The algorithm for the weighted topology in TensorFlow was slightly different to the other three. For its NumPy equivalent, the weights were incorporated into the Q-equation. But when made in TensorFlow, the weights were incorporated into the R-Matrix to see if this affected performance. The weights were normalised between zero and one (the closer to one, the lower the weight), and placed instead of the zeros into the R-Matrix as seen below in **Figure 4-8**.

```
[[ -1.    0.    0.5   0.8  -1.   -1.    0.5  -1.   -1.   -1.   -1. ]
 [  0.   -1.    0.7  -1.    0.8  -1.   -1.    0.8  -1.   -1.   -1. ]
 [  0.5   0.7  -1.    0.6   1.    0.8  -1.   -1.   -1.   -1.   -1. ]
 [  0.8  -1.    0.6  -1.   -1.   -1.    0.9   0.2  -1.    0.6  -1. ]
 [ -1.    0.8   1.   -1.   -1.    1.   -1.    0.9  -1.   -1.  100.3]
 [ -1.   -1.    0.8  -1.    1.   -1.    0.5   0.8   0.9  -1.   -1. ]
 [  0.5  -1.   -1.    0.9  -1.    0.5  -1.   -1.    0.7   0.9  -1. ]
 [ -1.    0.8  -1.    0.2   0.9   0.8  -1.   -1.    0.6  -1.  100.4]
 [ -1.   -1.   -1.   -1.   -1.    0.9   0.7   0.6  -1.    0.5 100.9]
 [ -1.   -1.   -1.    0.6  -1.   -1.    0.9  -1.    0.5  -1.  100.4]
 [ -1.   -1.   -1.   -1.    0.3  -1.   -1.    0.4   0.9   0.4 101. ]]
```

Figure 4-8: *Reward Matrix for TensorFlow Weighted*

By using this form of the R-Matrix, the weight relating to each move had already been accounted for when calculating the appropriate Q-value using the Bellman Equation.

4.4.5 Formatting

As mentioned previously, the output of these algorithms had to be in a format that the Mininet application could both read and use. The following format was required:

$$\{first\ switch\colon \{first\ switch\colon [shortest\ route\ to\ starting\ switch],$$
$$second\ switch[shortest\ route\ to\ starting\ switch],$$
$$third\ switch[shortest\ route\ to\ starting\ switch],$$
$$\dots etc\}$$

For example:

$$\{'S1': \{'S1': ['S1'],$$
$$'S2': ['S1', 'S2'],$$
$$'S3': ['S1', 'S3'],$$
$$\ldots etc\}$$

One problem encountered was that the external code was written in Python2 (Mininet is currently only supported by Python2) whereas the algorithms were written in the more up to date Python3. The main issue here was exporting switch lists between programs. Python2 exported them in Unicode format (Python3 automatically handles Unicode), and so a switch became $u'S11'$ instead of $'S11'$. Therefore, certain functions were written to manipulate these outputs into their desired format.

5 Results

5.1 Introduction

The following section of this thesis begins with a walkthrough of the network setup. More details of this setup can be found in the README.txt file. It then analyses the results that were gathered from the four algorithms. As mentioned previously, all four algorithms outputted the same result, i.e. the shortest paths. Therefore, only one (NumPy unweighted) was implemented together with the overall architecture to show usability. It is advisable to refer to the code provided when reading this section.

5.2 Network Setup

In order for the network to work correctly, the following setup should be used from within the virtual machine. The commands below shown in **bold** translate to what the user should enter on their local machine. Before starting, a virtual machine (VM), e.g. VirtualBox, running Mininet should be installed. The Ryu controller should be downloaded (pip install) within the VM along with the associated required packages. The six scripts in the folder *SDN Mininet Simulation* provided in conjunction with this thesis should also be downloaded and saved within the folder *ryu/ryu/app* in the VM. Open the README.txt file and follow instructions 1-6 before executing the following commands.

1. Start the virtual machine running Mininet. A Linux window running Ubuntu will open. (This window can be used to simulate the network, but usability is improved if a *ssh* connection is used from the local machine). Login with the credentials:
    a. Login: ***mininet***
    b. Password: ***mininet***
2. Open two instances of your virtual machine via a *ssh* connection from your local machine e.g. Terminal (mac), Command Prompt (Windows) and log in to both:
    a. Login: ***ssh -X mininet@192.168.56.101***
    b. Password: ***mininet***

c. (The IP address can be found by typing ***ifconfig*** within the first Ubuntu window and should be seen under *eth0/eth1, inet addr: XXX*).

3. Navigate to the correct folder in both ssh windows:
    a. ***cd ryu/ryu/app***
4. Clean the old network topology from the local system in both windows (it is necessary to do this before running all new simulations):
    a. ***sudo mn -c***
5. In the first *ssh* window, start the network:
    a. ***sudo python NetRunnerNS.py -f PanEuroNet.json -ip 192.168.56.101***
    b. (Use the appropriate IP address. A confirmation message should be displayed, the hosts and switches will be initiated).
6. In the second *ssh* window, start the SDN controller which runs the Q-learning algorithm along with the appropriate network topology:
    a. ***python l2DestForwardStaticRyuNS.py --netfile=PanEuroNet.json***
    b. (A confirmation message that forwarding tables have been set up for each switch should be displayed).
7. Test if the network is working correctly:
    a. ***pingall***

5.2.1 Network Test

Figure 5-1 shows a successful *pingall* command execution with the implementation of the NumPy, unweighted algorithm. 0% of the packets were dropped which means that the SDN controller successfully installed all the correct routing tables based on the Q-learning algorithm.

```
mininet> pingall
*** Ping: testing ping reachability
H1 -> H2 H3 H4 H5 H6 H7 H8 H9 H10 H11
H2 -> H1 H3 H4 H5 H6 H7 H8 H9 H10 H11
H3 -> H1 H2 H4 H5 H6 H7 H8 H9 H10 H11
H4 -> H1 H2 H3 H5 H6 H7 H8 H9 H10 H11
H5 -> H1 H2 H3 H4 H6 H7 H8 H9 H10 H11
H6 -> H1 H2 H3 H4 H5 H7 H8 H9 H10 H11
H7 -> H1 H2 H3 H4 H5 H6 H8 H9 H10 H11
H8 -> H1 H2 H3 H4 H5 H6 H7 H9 H10 H11
H9 -> H1 H2 H3 H4 H5 H6 H7 H8 H10 H11
H10 -> H1 H2 H3 H4 H5 H6 H7 H8 H9 H11
H11 -> H1 H2 H3 H4 H5 H6 H7 H8 H9 H10
*** Results: 0% dropped (110/110 received)
```

Figure 5-1: *Mininet pingall Execution*

5.3 Algorithm Analysis

The following four figures display the rate of shortest path conversion for each Q-learning algorithm; NumPy unweighted (**Figure 5-2**), NumPy weighted (**Figure 5-3**), TensorFlow unweighted (**Figure 5-4**) and TensorFlow weighted (**Figure 5-5**). Note that the score (y-axis) for each algorithm was arbitrary in comparison to the other three. These scores were calculated differently for each algorithm and hence the scale was not uniform across the four. It was enough to show that the scores converged around a maximum value implying that the shortest path (highest score) had been found.

The unweighted algorithms were run for a total of 2,500 episodes and the weighted ones for 50,000. The reasoning behind this design choice was because the unweighted topology had seven possible shortest routes from $S1 \rightarrow S11$ with the same distance (three hops). Therefore, the unweighted algorithms managed to find one of these shortest paths much faster. However, the weighted topologies had only a single path with a cumulative lowest weight (implying the shortest path), and so these algorithms needed

additional episodes to explore the network topology. It should also be noted that these algorithms converged onto the shortest path in much fewer episodes than they were trained for. In this simulation, the designer had access to the topology and so could have stopped training once the algorithm had found the shortest path in the network. However, this prior knowledge is not very realistic and so the prolonged training was left in to simulate an environment where the designer didn't know when the algorithm would converge onto the shortest path.

Figure 5-2 shows that the NumPy algorithm in an unweighted network topology successfully found the shortest path within 400 episodes. After this, it continually used the same path (exploited) and so its score did not increase. In the weighted NumPy algorithm (Figure 5-3), the shortest path converged much later on at around 10,000 episodes. This was due to the need for increased exploration of the environment; there was only a single shortest path in the weighted topology and hundreds of possible paths from start to finish to explore. Note that the runtime for this program was still less than a minute.

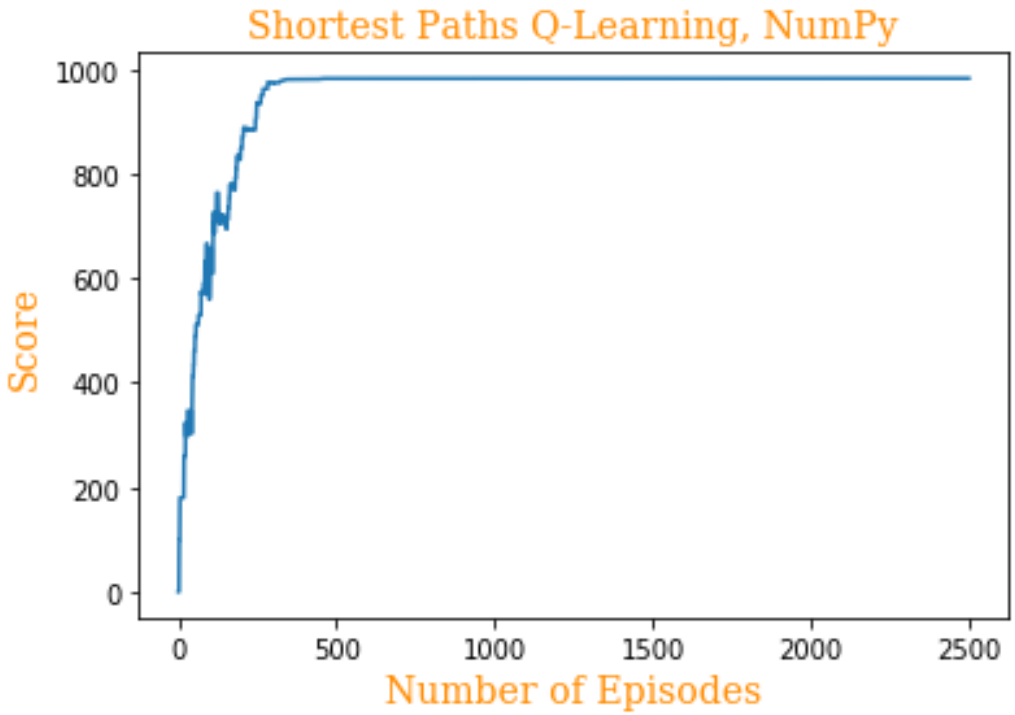

Figure 5-2: *Shortest Paths Graph, Unweighted NumPy*

The jitter that can be seen in **Figure 5-3** and **Figure 5-4** relates to the exploration-exploitation parameter. This jitter was associated with the algorithm occasionally exploring alternative paths, although these paths may not have necessarily produced a higher score. The exploration-exploitation parameter was left out of the NumPy unweighted algorithm as one of the seven shortest paths could be quickly found and once found, there was no need to explore to find an alternative route.

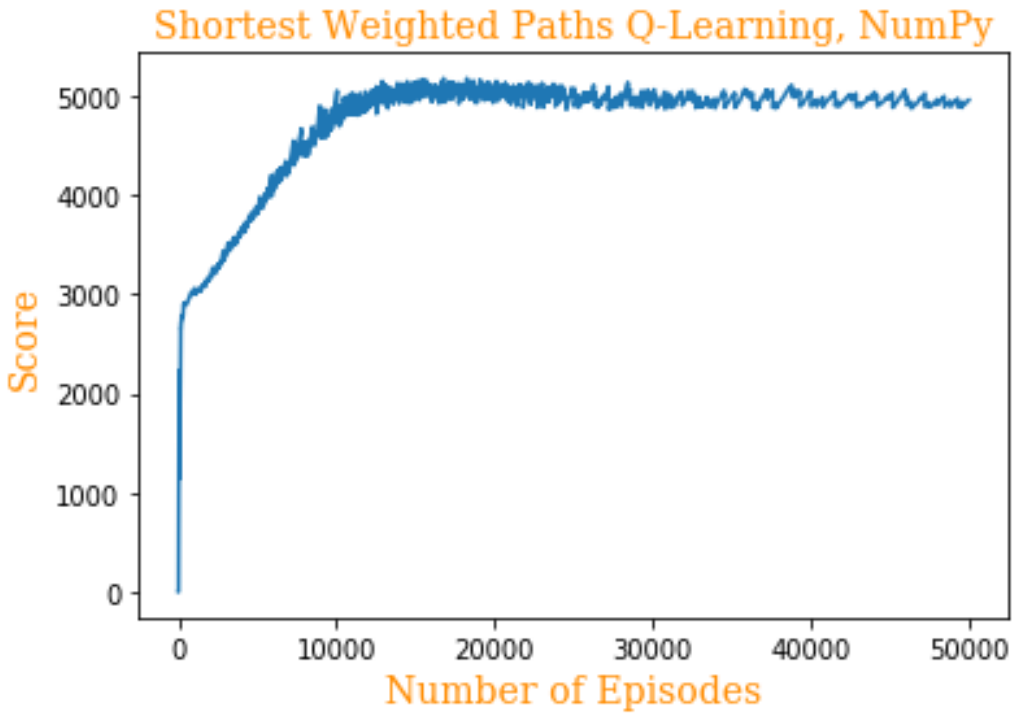

Figure 5-3: *Shortest Paths Graph, Weighted NumPy*

The TensorFlow implementations on average performed better than their NumPy equivalents. Although they converged a lot faster, the run time for each episode was longer due to the increased computational complexity required to update the neural network (as opposed to updating a Q-table). Both TensorFlow algorithms solved the shortest path problem within 500 episodes. **Figure 5-4** shows that the score sometimes increased even after the apparent convergence. The reason behind this was because the algorithm would sometimes find an alternative shortest path in its exploration phase (one of the seven containing three hops) and exploit it until it gave a higher Q-value than previously. This would be repeated each time an alternative

shortest path was found. If the exploration-exploitation parameter had been reduced to $\varepsilon = 0$ instead of $\varepsilon = 0.1$, the score would have plateaued at its maximum value like in **Figure 5-5**. However, it was good practice to leave $\varepsilon = 0.1$ so that the algorithm was scalable for more complex topologies and would be able to handle a dynamically changing network.

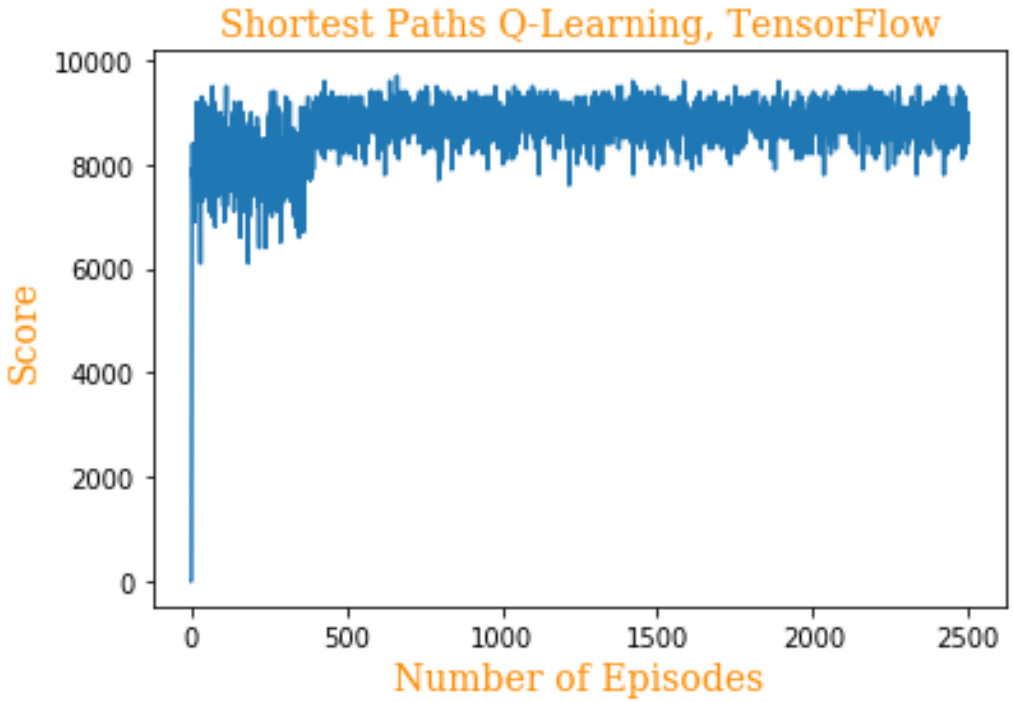

Figure 5-4: *Shortest Paths Graph, Unweighted TensorFlow*

In the weighted TensorFlow algorithm (**Figure 5-5**), there was only a single shortest path associated with the highest score. Therefore, the fluctuations of scores in the exploration phase only ever dropped below the maximum score and never rose above i.e. it could only find worse paths to follow.

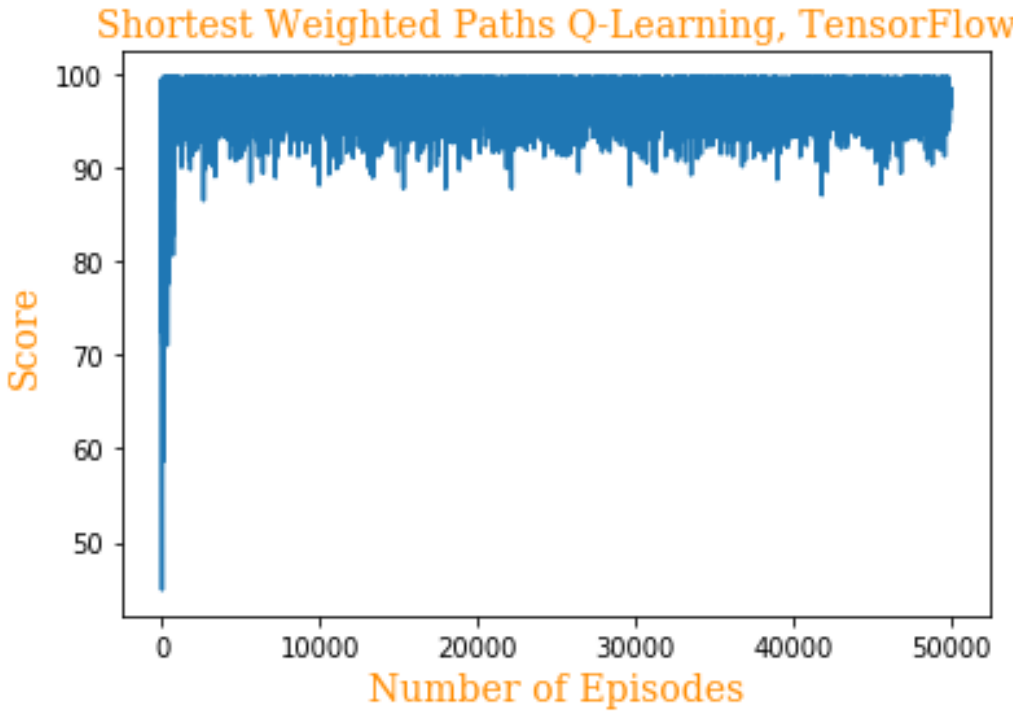

Figure 5-5: *Shortest Paths Graph, Weighted TensorFlow*

5.4 The Problem of Loops

There was one main problem that was encountered when the weighted TensorFlow application was implemented. The algorithm worked by assigning larger rewards to links with smaller weights and so the shortest path (lowest cumulative weight) was characterised by the highest score. Each episode consisted of a maximum of ninety-nine moves around the topology so as to ensure that the final state would be reached by chance. However, if this final state was reached before the ninety-ninth iteration, the episode would end. The algorithm independently learnt that it could exploit loops within the network to cheat the system. By repeating these loops, it would gain the maximum possible score before reaching its final state. What is remarkable is that it not only figured this out on its own, but it also found the best loop to travel around ($S2 \rightarrow S4 \rightarrow S5 \rightarrow S2 \ldots etc$) and ensured that its final move, reaching the goal state, occurred on its ninety-ninth move. This was obviously undesirable; however, it shows the power of a very simple self-learning algorithm. The solution to this was to make the weights negative so that an increased episode length would not produce a higher

cumulative score. Once this was implemented, the algorithm began behaving appropriately.

6 Future Work

6.1 Introduction

This chapter of the thesis provides an in-depth analysis of where future research should be directed. It evaluates some techniques which should be explored in order to improve the performance of the Q-learning algorithms as well as some additional tests which could be executed.

6.2 Overview

There are several exciting potential streams of research which would build on this project. Out of algorithms created, the TensorFlow weighted application should be the one where the majority of this further research is conducted as it is the most scalable and realistic of the four.

Primarily, these algorithms should be tested on far more complex networks to see how well they perform. If they were left to run for a few hours on a powerful computer, they should be robust enough to find the shortest path in a network with thousands of switches. It might however, be advisable to firstly update the algorithms and include some improved reinforced learning techniques as described below.

6.3 Deep Q-learning, Experience Replay, Target Q-networks and More

Additional reinforcement learning techniques could be applied to the models in order to improve performance. It was these innovations that allowed the Google DeepMind team to achieve superhuman performance on their simulations of several Atari games [23]. Firstly, the TensorFlow weighted application should be altered in order to accommodate for Deep Q-learning. The deep simply means the implementation of additional hidden layers which would change the model from a single-layered network, to a multi-layered convolutional network. This would be far more robust in handling larger topologies without taking up excessive memory in the form of a large Q-table. Secondly, the implementation of Experience Replay should be investigated. This relates to the idea that the network can train

itself on stored memories from several past experiences within the environment, instead of just the previous episode. The agent selects a random batch of past experiences and so learns the task in a more robust manor as the previous episode may contain biases. By drawing from random episodes, it trains via a varied set of experiences and so performance should increase. The experiences would be stored inside a buffer in memory, and once full, the oldest experiences would be replaced by newer ones. Thirdly, a separate Target Neural Network should be used to compute Q-values during training. This network would be used to produce target Q-values which calculate the loss for every state-action pair. It may seem strange to use this second network, but it can have a significant impact on performance. The Q-values in the network continually change during each iteration of that episode (the movement from one switch to another). Therefore, if these changing values are used to update the network, the value approximations can accelerate out of control. To reduce this risk, the target networks' weights are fixed and are slowly updated in order to facilitate for a more stable training period. A second DeepMind paper [24] explains this solution in further detail.

The world of reinforcement learning is an ever-changing area and so techniques that affect performance are constantly being discovered. Two of these techniques which have gained popularity in recent years are Double Deep Q-learning and Dual Deep Q-learning. These could both be additional streams of research.

6.4 Graph Nets and Unsupervised Learning

DeepMind released Graph Nets [25] to the public in late 2018. It provides a powerful library which can be built on top of TensorFlow to simulate large networks of interconnected nodes. This could be implemented into the Q-learning application as it allows for easily created random network topologies which algorithms could train on. It also allows for the experimentation of unsupervised learning regarding the shortest path finding problem. The idea behind this would be to create an algorithm which is shown lots of random network topologies and independently learns how

to find the shortest path based on this data. This algorithms performance could be compared with the reinforcement learning alternative.

6.5 Independent Learning of the Network

Further scope for research should analyse whether the Q-learning algorithms could be implemented into the weighted network topology without any previous knowledge of the network. An ideal situation would be for the application to explore the network, finding switches and weights as it goes, and at the same time train its reinforcement learning model. Currently, the SDN network has knowledge of the topology before it is implemented but there are benefits of being able to train on an unseen network.

7 Conclusion

7.1 Conclusion

This thesis explores the potential of implementing different reinforcement learning algorithms to find the shortest path within a Software Defined Networking framework. It introduces the core concepts surrounding both SDN and reinforcement learning and explores the implementation of four separate algorithms. Two Python-based packages; NumPy and TensorFlow are investigated and compared on both a simplified unweighted network topology, and a more realistic weighted one. The four applications are built in such a way that they can be easily implemented into a SDN architecture. Each one shows potential in accurately discovering the shortest paths in a network topology within a timely manner. One of these algorithms has also been successfully incorporated into a SDN architecture and simulated in Mininet using a Ryu based controller to demonstrate usability.

The two applications implemented in TensorFlow use neural networks which has the advantage of scalability. This is beneficial as it eliminates the need of having excessive memory to store a matrix of Q-values. They also, on average, perform better than their NumPy equivalents i.e. converge onto the shortest path within less training episodes.

SDN is still a relatively new concept and its uptake has been slow. Networks are faced with unprecedented pressure as millions of new devices require connectivity every day and handling this increase is becoming a difficult task. However, SDN has the potential to assist in this and become the future of networking. SDN, being relatively new, also provides the opportunity to perfect network systems and exploit powerful machine learning techniques before the architectures are fully deployed. This large network of interconnected devices allows for the possibility of huge data sets to be collected (assuming no data protection acts are breached) and used to train machine learning models.

However, as personal data security is becoming more of a priority in recent years, with the likes of the EU General Data Protection Regulation (GDPR) [26], the application of reinforcement learning that can train without the use of large data sets may become more viable than both its supervised and unsupervised alternatives.

One shortcoming of this research is that the algorithms created have access to knowledge of the network topology in which they are being trained (in the form of the number of switches and the weights between those switches). However, this knowledge is not built into the algorithm, rather the appropriate part of the algorithm can access it as needed.

There is a plethora of potential research which could be investigated to improve the performance of these models. Primarily, the reinforcement learning algorithms should be altered so as to accommodate for deep learning. Following this, the additional techniques discussed in chapter six should be implemented. It may then be worthwhile to compare their performance with similar algorithms which use both supervised and unsupervised learning techniques.

References

[1] N. Sultana, N. Chilamkurti, W. Peng and R. Alhadad, "Survey on SDN based network intrusion detection system using machine learning approaches," *Peer-to-Peer Networking and Applications,* vol. 12, p. 493, 2019.

[2] Q. Li, N. Huang, D. Wang, X. Li, Y. Jiang and Z. Song, "HQTimer: A Hybrid Q -Learning-Based Timeout Mechanism in Software-Defined Networks," *IEEE Transactions on Network and Service Management ,* vol. 16, pp. 153-166, 2019.

[3] T. Mu, A. Al-Fuqaha, K. Shuaib, F. M. Sallabi and J. Qadir, "SDN Flow Entry Management Using Reinforcement Learning," *ACM Transactions on Autonomous and Adaptive Systems (TAAS),* vol. 13, no. 2, 2018.

[4] M. Casado, M. Freedman, J. Pettit, N. Luo, J. Luo, N. McKeown and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM Computer Communications Review: ACM,* vol. 37, pp. 1-12, 2007.

[5] A. Maleki, M. Hossain, J. Georges, E. Rondeau and T. Divoux , "An SDN Perspective to Mitigate the Energy Consumption of Core Networks - GEANT2," in *International Seeds Conference*, Leeds, 2017.

[6] T. N. McKeown, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, J. Shenker and J. Turner, "OpenFlow enabling innovation in campus networks," *SIGCOMM Comput. Commun,* vol. 38, pp. 69-74, 2008.

[7] Open Networking Foundation, "openflow-spec-v1.3.0," 2012. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[8] W. S. McCulloch, "Bulletin of Mathematical Biophysics," Kluwer Academic Publishers, 1943.

[9] M. Campbell, A. J. Hoane and F. Hsu, "Deep Blue," *Artificial Intelligence,* vol. 134, pp. 57-83, 2002.

[10] F. R. Yu and Y. He, Reinforcement Learning and Deep Reinforcement Learning, Spinger, Cham, 2019.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Drissche, J. Schrittwieser, I. Antonglou, V. Panneershelvam, M. Lanctot, M. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and H, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature,* vol. 529, pp. 484-489, 2016.

[12] B. B. Lantz and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks ACM,* p. 19, 2010.

[13] I. Yamahata and K. Morita, "Ryu: Network operating system".

[14] A. A. Hagberg, D. A. Schult and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," *Proceedings of the 7th Python in Science Conference (SciPy2008),* pp. 11-15, 2008.

[15] T. E. Oliphant, "A guide to NumPy," Trelgol Publishing, 2006.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin and S. Ghemawat, "TensorFlow: Large-scale machine learning on hetrogenous systems," 2015.

[17] G. M. Bernstein, "Grotto Networking," 2015. [Online]. Available: https://www.grotto-networking.com/SDNfun.html.

[18] A. Ferreira, H. Pérennes, H. Rivano, A. W. Richa and N. Stier-Moses, "Models Complexity and Algorithms for the Design of Multi-fiber WDM Networks," *Telecommunications Sytems: OAI,* 2003.

[19] W. He, J. Fang and A. Somani, "A p-cycle based survivable design for dynamic traffic in WDM networks," *Global Telecommunications Conference, GLOBECOM:IEEE,* vol. 4, 2005.

[20] G. M. Bernstein, "ComNetViz," 2015. [Online]. Available: http://www.grotto-networking.com/ComNetViz/ComNetViz.html.

[21] A. Juliani, “Simple Reinforcement Learning with TensorFlow Part 0: Q-Learning with Tables and Neural Networks,” 2016. [Online]. Available: https://gist.github.com/awjuliani.

[22] The Beginner Programmer, “firsttimeprogrammer.blogspot.com,” 2016. [Online]. Available: http://firsttimeprogrammer.blogspot.com/2016/09/getting-ai-smarter-with-q-learning.html.

[23] V. Mnih, K. Kavukcuoglu, D. Silver , A. A. Resu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller , A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie , A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wiestra, S. Legg and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature,* vol. 518, pp. 529-533, 2015.

[24] T. P. Lillicrap, J. J. Hunt, A. Pritzel, A. Heess, Y. Tassa, D. Silver and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv,* 2015.

[25] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Taccetti , D. Raposo, A. Santoro, R. Faulkner and C. Gulcehre, “Relational inductive biases, deep learning, and graph networks,” *arXiv,* 2018.

[26] European Union, “Regulation (EU) 2016/679 of the European Parliament and of the council of 27 April 2016, on the protection of natural persons with regard to the processing of personal data and on the free-movement of such data and repealing Directive 95/46/EC,” *Official Journal of the European Union: EU,* 2016.

Appendix A

Software Packages Used

Table 1: Software / Packages / Code used

Filename/Algorithm/Package	Supplier/Source/Author/ Website	Use/Modifications made/ Student written
g_switchesOut.pickle	Francesco Venerandi.	Student written. Needed to run simulation. (Empty file).
Getting AI smarter with Q-learning: a simple first step in Python	The Beginner Programmer. *https://firsttimeprogrammer.blogspot.com/*	Student used. Code can be found at: *http://firsttimeprogrammer.blogspot.com/2016/09/getting-ai-smarter-with-q-learning.html* Sets up simple Q-learning algorithm using NumPy.
json	Included in python library.	Used to import .json files into simulation.
l2DestForwardStaticRyuNS.py	Greg M. Bernstein. *https://www.grotto-networking.com/SDNfun.html#*	Code can be found at: *https://www.grotto-networking.com/code/SDNfun/l2DestForwardStaticRyuNS.py* Used in SDN simulation.
MatPlotLib	Python library. matplotlib.org	Used to generate plot graphs.
Mininet	SDN network simulator. mininet.org	Used to simulate network topology.
net_NP_weighted.py	Francesco Venerandi. Based on code written by Arthur Juliani & The Beginner Programmer.	Student written using parts of code: *https://gist.github.com/awjuliani/9024166ca08c489a60994e529484f7fe#file-q-table-learning-clean-ipynb* and: *http://firsttimeprogrammer.blogspot.com/2016/09/getting-ai-smarter-with-q-learning.html* NumPy weighted graph.
net_NP.py	Francesco Venerandi. Based on code written by Arthur Juliani & The Beginner Programmer.	Student written using parts of code: *https://gist.github.com/awjuliani/9024166ca08c489a60994e529484f7fe#file-q-table-learning-clean-ipynb* and: *http://firsttimeprogrammer.blogspot.com/2016/09/getting-ai-smarter-with-q-learning.html* NumPy unweighted graph.
net_TF_simple.py	Francesco Venerandi. Based on code written by Arthur Juliani & The Beginner Programmer.	Student written using parts of code: *https://gist.github.com/awjuliani/4d69ead4d0ed9a5884f3cdcf0ea0874#file-q-net-learning-clean-ipynb* and: *http://firsttimeprogrammer.blogspot.com/2016/09/getting-ai-smarter-with-q-learning.html* TensorFlow unweighted graph.
net_TF_weighted_simple.py	Francesco Venerandi.	Student written using parts of code:

	Based on code written by Arthur Juliani & The Beginner Programmer.	*https://gist.github.com/awjuliani/4d69ead4d0ed9a5884f3cdcf0ea0874#file-q-net-learning-clean-ipynb* TensorFlow weighted graph.
NetRunnerNS.py	Greg M. Bernstein. *https://www.grotto-networking.com/SDNfun.html#*	Student modified lines 81-82. Code can be found at: *https://www.grotto-networking.com/code/SDNfun/NetRunnerNS.py* Used in SDN simulation.
NetworkX	Python library for graphs. networkx.github.io	Used to create visual network topology & shortest_paths algorithm to check validity of results.
NumPy	Python library. numpy.org	Used to create Q-learning algorithms.
PanEuroNet.json	Francesco Venerandi. ComNetVis created by Greg M. Bernstein.	Used to create network topology within simulation. Created using application ComNetVis (grotto-networking): *http://www.grotto-networking.com/ComNetViz/ComNetViz.html*
Pickle	Included in Python library.	Used to transfer files between applications.
Pylab	Python library. scipy.github.io	Used to generate plot graphs.
Python	Python2 & Python3 used.	Used as base language for all code.
README.py	Francesco Venerandi	Student written. Details on how to run code.
Ryu	osrg.github.io/ryu/	Used as SDN controller within simulation.
ShortestPathBridgeNet_NP.py	Greg M. Bernstein. *https://www.grotto-networking.com/SDNfun.html#*	Student modified lines 44-61, 127. Code can be found at: *https://www.grotto-networking.com/code/SDNfun/ShortestPathBridge.py* Used in SDN simulation.
TensorFlow	Python library. tensorflow.org	Used to create Q-learning algorithms.
Virtual Box	virtualbox.org	Used to run Linux virtual machine on host PC.

Appendix B

Reinforcement Learning Flow Chart

Table 2: *Flow Chart for Q-learning*