Deep Q-Learning for Load Balancing Traffic in SDN Networks

Vasileios Tosounidis
University of Macedonia
Department of Applied Informatics
Thessaloniki 54636, Greece
vasilis.tos@gmail.com

Georgios Pavlidis
University Of Macedonia
Department of Applied Informatics
Thessaloniki 54636, Greece
george030796@gmail.com

Ilias Sakellariou
University Of Macedonia
Department of Applied Informatics
Thessaloniki 54636, Greece
iliass@uom.edu.gr

ABSTRACT

Load balancing is a widely used technique that aims to enable large network topologies, most commonly found in large data centers, to handle the constantly varying load of service requests. Traditional networks built on multi-vendor hardware and software present significant difficulties in the efficient and flexible application of load balancing techniques. Usually, solutions rely on high cost dedicated hardware and thus are used only for a subset of the tasks, resulting to limited flexibility for network administrators. Software Defined Networking (SDN) is a relatively new approach that enables flexible network management solutions to a number of problems, including that of efficient load-balancing. The key characteristics of decoupled centralized network control combined with programmability, allows the seamless integration of AI techniques to network management. Toward this direction, this paper employs deep reinforcement learning to effectively load balance requests to services in a data center network, resulting to an approach that is able to dynamically adapt to varying request loads, including changes in the infrastructure's capabilities. The proposed approach is experimentally evaluated in order to support its feasibility, with very promising results.

CCS CONCEPTS

• **Computing methodologies → Reinforcement learning**; • **Networks → Traffic engineering algorithms**; Network simulations.

KEYWORDS

Deep Q-Learning, Convolutional Neural Network, Machine Learning, Artificial Intelligence, Deep Learning, SDN, Network Analytics, Knowledge-Defined Networking

ACM Reference Format:
Vasileios Tosounidis, Georgios Pavlidis, and Ilias Sakellariou. 2020. Deep Q-Learning for Load Balancing Traffic in SDN Networks. In *11th Hellenic Conference on Artificial Intelligence (SETN 2020), September 2–4, 2020, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3411408.3411423

1 INTRODUCTION

Traditional network architectures are effectively distributed systems, in which control is spread among the nodes (i.e. routers, switches, etc.); hence each node has a very limited view and control over the complete network, making solutions like load-balancing, which require a global view of the network, very difficult to implement. Often solutions resort to dedicated hardware that has a very high cost and rather limited compatibility. In such a setting, network management is a difficult task since changes need to be implemented separately to each device, which in turn leads to higher maintenance costs. In addition, traditional networks lack in flexibility and resource utilization, due to the fact that after the policies of the network have been set it cannot dynamically allocate resources responding to hardware or load changes. To overcome these obstacles a new network architecture has emerged, most commonly known as Software-Defined Networking (SDN). This new paradigm decouples control from data devices (i.e. routers, switches) [8] and centralizes it in the control plane, which can be seen as a single logical point that has a complete view of the network and is responsible for all the operational decision-making. The data-plane, that consists only of data devices, is responsible for blindly executing the commands sent by the control plane. Being software-based means that this architecture can programmatically adapt to varying requirements of the network and allocate resources dynamically without the need of further human interference. In addition to its flexibility, the SDN paradigm proved to yield lower maintenance cost than traditional networks, because it does not require the addition of new hardware in order to scale the network.

The term "Knowledge plane" was introduced in [3] by David Clark, who argued that the Internet was in need of a mechanism that would enable it to use knowledge collected from what it was asked to do by learning from it in order to take care of itself, going as far as re-configuring its core. Unfortunately, very little effort has been done towards this approach. The rise of SDN, offers the possibility to use this principle in the Software-Defined Networking paradigm, through an adaptation of the Knowledge plane that would complement the SDN infrastructure as proposed by A. Mestres et al. in [12]. In the current paper, following the principles proposed in that work, we propose a Neural Network to take full control of the operational decision-making in an SDN. The centralized control of the SDN, in addition to network analytics run as an external program providing a richer view of the system, enable the use of a Deep Reinforcement Learning model, in order to gain knowledge on how the network responds to different requirements and dynamically control the former through the control plane. The problem addressed in this work is that of the maximum utilization of network resources through a mechanism most commonly known as load balancing.

The rest of the paper is structured as follows. Section 2 offers the necessary background by describing the basic principles of SDN, KDN and Deep Learning. Section 3 presents the problem of load balancing networks. Section 4 presents the Deep Neural Network used to achieve load balancing. The system architecture used to conduct all experiments is briefly described in Section 5. Various load balancing experiments conducted on a simulated SDN with the use of a Deep Q Learning model and compared against traditional load balancing algorithms are described in Section 6. Finally, Section 7 concludes the paper and provides directions for future work.

2 BACKGROUND AND RELATED WORK

The term *Software-Defined Networking* was introduced in an online article by K. Greene [6] to refer to the work done in the context of a new standard, namely OpenFlow [11], in Stanford University. This new standard allowed access to flow tables, which are essentially a set of rules that routers and switches abide to, in order to control the network through software.

In the SDN architecture the control and data planes are decoupled, meaning that network devices assume only the role of forwarding packets through the network based on the rules imposed by the control plane. Thus, this architecture comprises of 3 distinct planes [8]:

(1) the *Management Plane*, that consist of applications that send instructions to the network such as firewalls, load balancers etc.
(2) the *Control Plane*, i.e. the SDN controller that has complete information on the state of the underlying network devices, the links between these, as well as the flows that are set by the management plane,
(3) the *Data Plane*, consisting of all the network devices (i.e. routers, switches) responsible of forwarding the packets throughout the system by executing instructions sent by the controller.

The SDN controller is not a physical device but rather a software application that resides in a dedicated server and has control over the network's devices [5]. It is the core of a software defined network, so any communication that occurs in the network between applications and network devices must go through the controller. In essence a controller manages the flow control to the switches/routers/modems and the applications to deploy automated intelligent networks. The controller communicates with the network devices using a dedicated interface, most commonly one like the OpenFlow protocol [11] that was released by the ONF (Open Networking Foundation) in 2011 which allows the controller to (re)configure the individual network devices and choose the most efficient network path for an application.

The principle of *Knowledge-Defined Networking* was initially presented by D. Clark et al. [3] as a construct that would have a higher-level view over a network and be able to provide instructions as needed to all the other network elements. In addition to the three distinct SDN planes, the KDN paradigm introduces the *Knowledge Plane* (KP), which has the ability to integrate behavioral models and reasoning processes oriented to decision making in the SDN network [12]. The KP works by leveraging the amount of information collected by the management and control planes and uses it to learn how the underlying network behaves in various situations. This enables it to make decisions based on Machine Learning and, in some cases, automatically configure the network to achieve optimal operation. Since SDN supports the concept of automation, applying advanced operational methods based on Machine Learning and Artificial Intelligence is not a difficult task. The centralized nature of SDN helps in applying such models, since data-plane components are controlled by a single controller that is aware of the current state of the network. The latter offers the ability to an ML/AI model to decide on the network's operational parameters based on the requirements or even the limitations of the current network state, i.e. the state of servers, devices and links in terms of load, bandwidth, etc. This can eventually lead to self-driven networks that require little to almost non continuous maintenance.

2.1 Q-Learning

Q-Learning is a reinforcement learning algorithm, often mentioned as a method of asynchronous dynamic learning [18], whose goal is to learn a policy based on which an agent "knows" which action to take under given circumstances. This method enables an agent to discover a solution in Markov Decision Process (MDP) [15], which usually describes an environment for reinforcement learning, without the need for the environment's model (model-free), simply by experiencing the outcomes of its actions. The MDP model is described by the tuple (S, A, P, R, γ) and a policy π, where S is the set of states, A is the set of actions, $P^{\alpha}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = \alpha\right]$ is the state transition probability matrix, $R : S\,x\,A \rightarrow \mathbb{R}$ is a reward function and $\gamma \in [0, 1)$ is called a discount factor. The MDP model includes a *policy* that guides the choice of an action in a given state, and is described as $\pi(\alpha \mid s) = \mathbb{P}[A_t = \alpha \mid S_t = s]$. The Q-Learning process takes place with the help of rewards and penalties for each action performed by the agent. By repeatedly trying all actions in all states it is able to learn the best pairs of state-action. The agent chooses an action at each state from a matrix of states-actions (Q-table) that includes reward estimations for each action. These estimations (Q-Values) are calculated by the Q-formula. After the actions have been taken the agent is either rewarded or penalised based on the outcome of the action and the states-actions matrix is updated with new Q-Values as follows (Eq. 1):

$$Q_{t+1}(s_t, a_t) := Q_t(s_t, a_t) + \alpha \cdot [r_t + \gamma \cdot maxQ_t(s_{t+1}, \alpha) - Q(s_t, a_t)] \quad (1)$$

where $\gamma \in [0, 1)$ is the discount factor that determines the importance of future rewards, $\alpha \in [0, 1)$ is the learning rate and r_t is the reward at time t. This process is repeated until there is no further improvement in the model's learning. The goal of the training is to make the policy (matrix with Q-Values) as accurate as possible, in order to always select the best action for a given state.

The existence of a plethora of problems that cannot be described by discrete states, or in which there are so many states that is inefficient to create a Q-table of states-actions, led DeepMind [13] into developing and presenting Deep Q-Network (DQN), an evolution of the Q Learning algorithm, that successfully combined a Convolutional Neural Network (CNN) and Q-Learning in order to create a more powerful Deep Neural Network. Simply put, with the help of a Neural Network the agent can make a prediction of the Q Value, instead of creating a matrix with all the different Q values for each pair of state-action, thus greatly limiting the amount of calculations that need to be done for each state. The Neural Network estimates the probable value of Q, by taking data that describe the current state as input and returning the estimated future rewards as output. In addition it introduces a mechanism of learning from experience, by implementing a “replay memory” that stores all of the agent’s “experiences” and then randomly samples from that dataset to train. In the field of networking, where it is expected that the NN will need to take a lot of actions, the fact that it doesn’t have to calculate the Q Table each time, drastically improves it’s performance and the overall performance of the network in general.

2.2 Related Work

A number of research works considered the use of a Q-Learning model, sometimes in addition to Neural Networks (NN), in order to load balance the traffic in an SDN network. Ruelas in [17] proposed the introduction of OpenFlow switches that would include a NN, able to control the flows inside the SDN network. This work was mostly aimed at data-centers with topologies like that of a fat-tree (see Section 3.1), in order to benefit from the multiple paths between a source host and the destination. However, this proposal presents difficulties in its implementation, since each switch and router needs to host the NN in order to control the flows. This goes against a fundamental principle of the SDN paradigm, that of a single centralized point of control, by reintroducing the control function in data-plane devices. Work reported in [2] attempts to solve the problem of load balancing between the links of the network in order to minimize load on the paths, using a Back Propagation Artificial Neural Network (BPANN). The ANN model is used as an external module to the SDN that monitors the network through the SDN controller. This way it is able to continuously train on the data reported by the controller. After its initial training the algorithm can successfully load balance the traffic in the links of the network. Tennakoon, Karunarathna and Udugama presented a simple Q-Learning algorithm [19] that is concentrated on large Wireless SDNs. This algorithm is used to load balance networks that involve mobile devices and remote stations, thus targeting a rather limited problem. In addition, by enforcing a tabular QLearning algorithm, it is not able to detect unseen network states, that require something like a Convolutional Neural Network (CNN) to recognize the different patterns. Shih-Chun Lin et. Al. [10] presented a reinforcement learning algorithm that is Quality of Service (QoS) aware. This means that for each request the algorithm will choose the path that produces the highest QoS-aware reward, thus minimizing the number of unsatisfied users in the long term.

In the current work, we present an agent that combines a Convolutional Neural Network (CNN) with a Q-Learning algorithm in order to load balance the traffic between HTTP servers. This solution differs from previous works in that it introduces a CNN responsible for predicting the state of the network after each action. This contributes to an increased performance since the Q-Learning algorithm does not have to generate the state table each time, but also enables the agent to be topology agnostic. Thus, it can determine, without prior knowledge, all aspects of the network and how the latter behaves to varying loads, by simply collecting network data. Even though it can be applied to any topology, it is best suited for large network topologies that have significant traffic and provide multiple paths from source to a destination, such as those in large Data Centers.

3 LOAD BALANCING IN DATA CENTERS

In computer systems such as data centers (DC), the load balancing mechanism is responsible for distributing requests in order to ensure that all available resources are used with maximum utilization [16]. More specifically this mechanism improves the distribution of workload across multiple computing and network resources, choosing both the best route in the DC network and the most capable server to respond, such that the total amount of answered requests is maximised, the response time is minimised and overloading of any single resource is avoided. A load balancer can either be a software or a hardware artifact. Typically, the advantage of hardware-based balancers, is that with the right combination of specialized software and specialized hardware higher performance levels are achieved compared to the corresponding software-based solutions. However, hardware load balancers are vendor specific, do not offer interoperability with network devices from other vendors and most of the time they require greater financial resources to acquire and maintain.

Software-based load balancers can be divided into two main categories, static and dynamic [14]. *Static* algorithms are not affected by the changes that occur in the network during its operation; they instead base their decisions on a predetermined static policy. Examples of static algorithms include:

- *Random* – It is the simplest approach to distribute traffic across servers, since the balancer just selects randomly a machine each time to send a request.
- *Round Robin* – It redirects requests to all servers in turn and in the long run distributes evenly the traffic across the available machines. This algorithm works best when the servers are of roughly equal capacity.
- *Weighted Round Robin* – A variation of Round Robin, with the difference that a weight assigned to each server is taken into account. A higher value in the former, indicates a server of higher capacity, thus one that can serve more requests. This optimized version of round robin maintains the advantages of the original and at the same time solves the problem of unbalanced distribution in case of an asymmetric system.

Dynamic algorithms consider the current state of both the network and the servers in order to optimize the decision, in accordance to changes that may appear in the system. Some examples include:

- *Dynamic Round Robin* – The dynamic Round Robin works in the same way as the weighted Round Robin, there is weight attached to each server based on which, the balancer choose where to send a request. However, in this case, the weights are not defined by the administrator but automatically generated in real time. The balancer constantly monitors the system and depending on its current state, it adjusts ratio weights of servers, thus ratio weights are continually adapted to the conditions.
- *Least Response Time* – This algorithm uses information from a server’s health check, i.e. monitoring information, to determine which server should be selected to respond to a current request. The balancer checks which server is responding faster and redirects the request to this one, thus, the balancer ensures that machines with the most connections that will respond slowly, do not accept new requests.

This paper concentrates on a solution that falls under the category of dynamic load balancing algorithms and more specifically a subset of these algorithms that involves predictive methods. These methods try to predict the traffic and the state of machines and network in order to maximize the availability of the network. To accomplish this task, they require a plethora of information acquired from the network and the more information available to the balancer, the more accurate the predictions will be. Under this we considered that the Software-Defined Networking (SDN) paradigm is ideal since the SDN controller, being the single point of operational control over the SDN, can provide all the information about the status of both the underlying network (i.e servers, routes, switches) and the applications that run on top of it, hence servers responding to requests. We explore the possibility of solving the problem of load balancing an SDN by applying a Neural Network that will extract knowledge from the data collected by the network's operation and eventually be able to instruct the SDN controller on how to set the flows in the network, in order to evenly distribute load to individual servers. To ensure that our Neural Network can have access to a significant amount of data we decided to use the fat tree network topology for our SDN.

3.1 Fat Tree Network Topology

Since load balancing has to take into account both server capacity as well as the state of the underlying network, it is important to discuss how connectivity is organised in Data Centers [1]. The fat tree topology, is a special version of the Clos [4] network topologies based on a complete binary tree and designed by Charles E. Leiserson in 1985 [9]. The main use of this network is to connect the processors of a parallel, general-purpose, supercomputer, as it allows to communicate large amounts of information concurrently, in comparison with other types of networks. Nowadays, fat-tree topology is largely used in Data Center Networks as it can provide low hardware cost and high performance, therefore many big companies such as Facebook, Amazon, Google, and Microsoft prefer it for their gigantic data centers. The main entities of fat-tree topology are switches with the same number of ports and equal bandwidth (Figure 1). These switches are interconnected in such a way that going up the tree the number of "wires" connecting a node with its parent increases, thus there are multiple paths between switches, and as a result both the bandwidth and the tolerance to losses and errors increases.

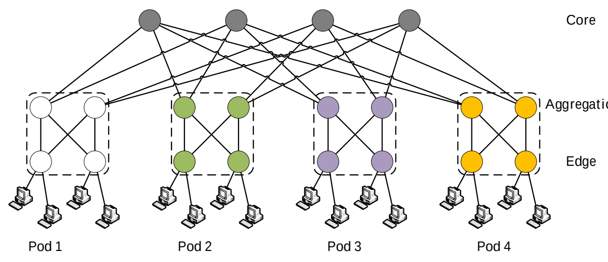

Figure 1: Simple k-ary Fat-Tree network topology with k=4

A fat-tree network is organized in 4 layers and k pods, where k is the number of ports per switch. The upper, root layer is called the core layer and it contains $(\frac{k}{2})^2$ switches. Next, there are aggregation layers, edge layer and in the end, are the hosts, located on the leaves. A fat-tree built with k-ports switches can support up to $\frac{3k}{4}$ hosts. The pods of the fat-tree network consist of $\frac{k}{2}$ aggregation switches and $\frac{k}{2}$ edge switches. Each switch of the lower layer is connected to $\frac{k}{2}$ hosts and from the remaining $\frac{k}{2}$ ports, each is connected to a port of a separate aggregation switch. An aggregation switch, in turn, is connected to $\frac{k}{2}$ edge switches and $\frac{k}{2}$ core switches. So, summarizing, with this connectivity in each pod, every aggregation switch is connected with all edge switches and all the edge switches with all aggregation switches, each pod is connected to $\frac{2k}{4}$ hosts and with all core switches and eventually, each core switch is connected to all pods.

4 LOAD BALANCING USING CNN AND Q-LEARNING

The proposed load balancing approach relies on two different types of monitoring data: network traffic on connections among switches in the fat tree topology, and server utilization. Each set is represented by a different tensor.

More specifically, the tensor representing the network state, is an NxN matrix, where N is the total number of switches in the network. The matrix contains a value related to the bandwidth of each link that connects two switches. This value is set to 0 if there is no connection between the switches, or the link has already been represented (i.e. s1 - s2 is the same as s2 - s1). Otherwise the value will range between 0 and 1, where 1 represents a 100% availability of the link's bandwidth, whereas 0 means that the link has reached its maximum capacity.

The second data input is a vector with values representing the CPU and memory utilization for each server. These two tensors combined make up for the complete state that our Q-Learning model will use.

Table 1: Output of the model as a tuple of (Server, Duration).

		Server		
		1	2	3
Duration	2	(1,2)	(2,2)	(3,2)
	4	(1,4)	(2,4)	(3,4)
	6	(1,6)	(2,6)	(3,6)
	8	(1,8)	(2,8)	(3,8)

In essence, a load balancing algorithm must decide where to direct each request received. Thus, the output of the model is an index of a server from the load balancing server pool that will respond to a certain number of subsequent responses as shown in Table 1. It should be noted that the total number of the subsequent responses is also set by the Q-Learning model, we will refer to this number as "duration" in the following. The output is in a form of a vector with a length of $A * D$ where A is the total number of servers and D is the number of available durations to answer continuous requests, which means that each position of the output vector corresponds to a unique combination of server-duration values.

The Deep Neural Network proposed (Figure 2) consists of three separate Neural Networks, two of which are used as input and processing of the data collected from the network. The first one is a Convolutional Neural Network (CNN) made up of two convolutional layers (convolutional layer, ReLu activation function) and one flat layer. The CNN is used to identify patterns in the network state. The second one is a simple Neural Network (NN) that consists of three hidden layers and is used to identify the CPU and memory load of the servers in the network. The outputs of the first two networks are directed to the third Neural Network, which has two hidden fully connected layers and one output layer, and produces the output vector. The Agent, State and Action in the proposed solution are as follows:

- *Agent* - that plays the role of the SDN Controller.
- *State* - We define the state as a combination of the bandwidth capacity of the links between the network devices and the CPU and memory load of the servers.
- *Action* - A tuple (S, D), where S is the server that will handle the request and D is the number of subsequent requests this server will handle.

Algorithm 1 was used to train the Deep Q Learning model.

Algorithm 1: Deep Q Learning training

```
1  Initialize Q(s,a) and weights with random normal
     distribution and with Xavier initializer
2  With probability $\epsilon$, select random action a, otherwise
     $max(Qs, a)$
3  Execute action a in the environment, obtain reward r and
     next state s'
4  Store transition(s, a, r, s',done) in the replay buffer (done is
     true if the episode has ended)
5  Observe for the first 1000 timesteps
6  Sample a random mini-batch of transitions from the replay
     buffer
7  For every transition in the mini-batch , calculate
     $targetQ = r + \gamma * max(Qs, a)$ , if the episode has ended
     $targetQ = r$
8  Calculate $cost = (Qs, a - targetQ)^2$
9  Update Q(s,a), minimizing the cost function
10 Repeat steps 2 to 8 until no further learning
```

The training parameters are described in Table 2. These values were defined through experimentation and appear to maximize the training of the model for this particular work. In addition to these we opted to use the Xavier initializer for the initial Q-Values matrix since it proved to be best suited for this solution.

The effectiveness of each action is measured by the average number of transactions in the course of 1 minute that the network was able to successfully serve by using the DQN load balancer. Towards this the reward function employed is shown in Eq. 2:

$$R_{(a_t)} := \frac{1}{60} \cdot \sum_{n=t-60}^{n=t} transactions(n) \tag{2}$$

where a_t is the action taken at time t and $transactions(n)$ is the number of transactions observed at n-th point in time. The overall effectiveness of the model is measured by a cost function (Eq. 3) defined as:

$$cost = [Q_{(s,a)} - (r_{(s,a)} + \gamma \cdot max(Q(s', a))]^2. \tag{3}$$

This is basically the Mean Square Error function (MSE) where $Q_{(s,a)}$ is the prediction and the immediate and future rewards are the target (Q-Target).

Table 2: Training Hyperparameters

Parameter	Meaning	Value
γ	Discount factor	0.9
η	Learning rate	0.000002
initial ε	Starting exploration probability	1.0
min ε	Minimum exploration probability	0.05
ε decay rate	Exponential decay rate of ε	0.00011875
dropout	Dropout to reduce overfitting	0.4
replay memory	Size of agent's memory	5000
batch	Batch size to train on	64
episode size	Size of each training episode	600 (s)

5 EXPERIMENTAL SETUP

This section describes the environment that was setup for the experiments. Due to limited resources there were some cutbacks in the scale of the network but were compensated by developing solutions that can easily be scaled to larger sizes.

5.1 Network Environment

To emulate the SDN we used Mininet, an open-source tool that is popular among researchers in the field of SDN. Mininet allows to create virtual networks deploying controllers, switches and hosts [7]. All the virtual devices support the OpenFlow protocol communication, making Mininet the perfect emulator for SDN.

We opted to use the Floodlight SDN controller [20], which is an open-source Openflow SDN controller. This decision was made because it provides well-defined REST APIs that allow applications to get and set the state of the controller, thus enabling our agent to dynamically perform such tasks. The controller acts as a middleware between the Neural Network (NN) and the underlying network devices providing the NN with all the available data on the state of the network after each action is taken. It is able to save flows in memory and each time there is uncertainty for the action that needs to be taken it will ask the NN for instructions.

All tests, experiments, and training were done on a fat tree topology, since it is used extensively in large data centers and thus simulates the actual conditions in which such a system should theoretically operate at its full potential. Figure 3 shows an abstract view of the network architecture that was used for the experiments of this paper.

We implemented this topology in the Mininet emulator, in such a way that it is possible to easily adjust different topology parameters such as network size or the capabilities of links and hosts so that it could be used in different experiments without the need to rewrite the topology.

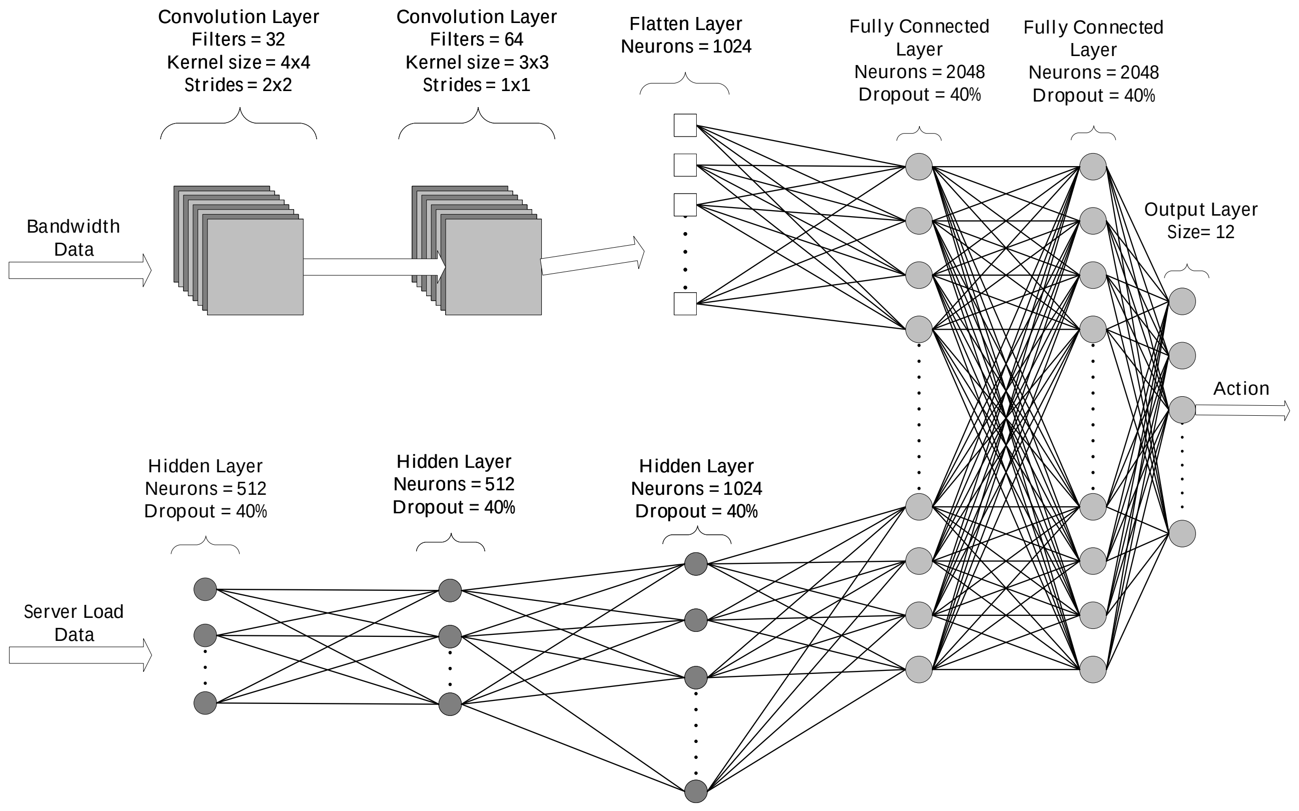

Figure 2: Proposed Deep Neural Network

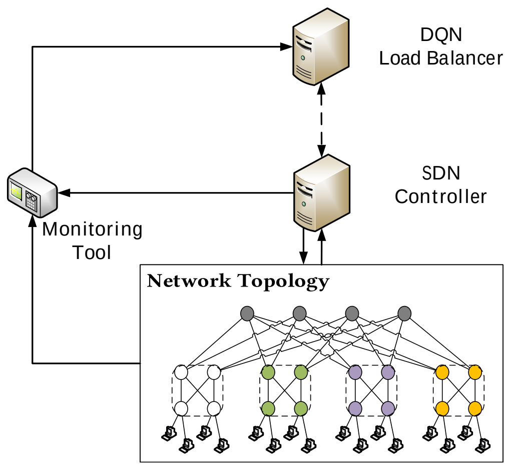

Figure 3: Network architecture for the proposed experiment.

5.2 Configuration

Since the maximum bandwidth of connections and processing power of hosts are directly influenced by the machine on which the Mininet is running we decided to limit the capabilities of our network in order to have more reliable analytics. In particular we limit the computational power of hosts to 20% of CPU. With the exception of hosts that are members of the load balancer pool, their power was set to 20%, 45% and 75% of CPU respectively, so that there is no homogeneity among the members of the pool. The bandwidth of links from edge layer to the aggregate layer was set to 500 Mbits with 10 ms delay. Links from aggregate switches to core switches have a little more bandwidth, as they have more network traffic, hence the boundaries were set to 1000 Mbits with 10ms delay and 800 Mbits with 20ms delay.

6 EXPERIMENTAL RESULTS

The first set of our experiments involved the training of the DQN model. The network was flooded with multiple requests and concurrent users and the model had no prior knowledge of the network topology, the quantity of the servers or the capacity of each server. By making decisions and taking a reward based on the overall wellness of the network's state after said decision, the model was able to train itself to select the best server to handle each request.

The second set involved experimentation with the trained model; we sent multiple requests from multiple concurrent users for a long period of time and observed how the load was distributed by the model between the servers. Figure 4 shows the traffic from the

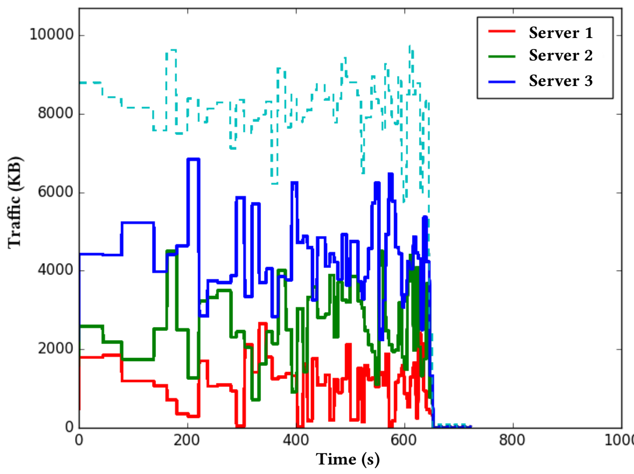

Figure 4: Traffic distributed between servers by DQN

requests made by 25 concurrent users for a period of 10 minutes; each line represents the load handled by each server for that period of time. From the figure we can see that the load is successfully distributed between the servers by the DQN model, based on their respective capacity (Server 1 20%, Server 2 45%, Server 3 75%).

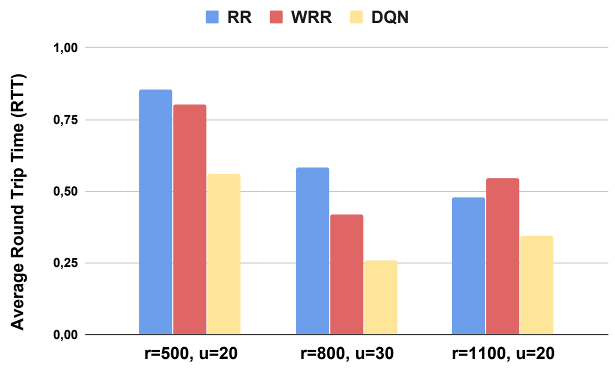

Figure 5: Average request latency for 3 different load scenarios (r: total number of requests, u: concurrent users)

6.1 Comparison with Traditional Load Balancing Algorithms

In order to evaluate the efficiency of the DQN, we "stressed" the servers by generating a large number of requests from multiple concurrent users and compared the results with other methods. We choose to compare DQN with Round Robin (RR) as one of the most popular methods and the Weighted Round Robin (WRR) as a more advanced method of load balancing. Experiments are divided into two main categories, where the first involves a varying number of requests over a specified time period and the second a constant number of requests; we ran a total of 10 iterations per experiment and extracted the average performance of each load balancing algorithm in order to have more accurate results to support our proposal. The values in Table 3 are the results of the first set of experiments where the network was flooded with requests from 30 concurrent users in a period of 4 minutes. In all metrics recorded the DQN model achieved higher performance, since it was able to distribute the requests between the servers in such a way that the amount of requests served was approximately 40% higher than both the other algorithms. Another notable aspect of this experiment is the reduced average number of failed transactions were the DQN model ensures a very high availability of the network.

Table 3: Average performance of load balancing algorithms

	RR	WRR	DQN
Transactions	5158	5404	7170
Data transferred (MB)	73,75	85,28	106,26
Response time (s)	0,97	0,72	0,46
Transactions/s	21,34	22,46	29,91
Throughput (MB/s)	0,31	0,34	0,44
Concurrency	13,72	14,70	19,79
Failed transactions	33	17	4

For the second part of the experiments we specified a total number of requests that the network should handle and a number of concurrent users. The evaluation results for 500 requests from 20 concurrent users are presented in Table 4. The DQN model was able to complete serving the requests sent to the network in almost half the time the other algorithms were able to.

Table 4: Average performance for 500 requests from 20 concurrent users

	RR	WRR	DQN
Time to complete (s)	52,64	50,20	28,10
Requests/s	12,38	13,09	23,42
Time per request (s)	85,29	80,41	56,21
Transfer rate	190,12	200,98	216,15
Throughput (MB/s)	0,31	0,34	0,44
Concurrency	19,72	14,70	13,79
Failed tranctions	17,00	59,00	2

Figure 5 shows the average responsiveness of the network on varying number of requests and concurrent users. This is calculated by the time each request needs to go from host to server and back. It is usually a way to measure the overall performance of the network since it also involves the route a request follows inside it. DQN "inferred" that in order to get a higher reward this time should be minimized, resulting in much lower Round Trip Time (RTT) than the other algorithms.

7 CONCLUSION

Current network architectures used in data centers are reaching their limit in supporting the constantly increasing load of data and traffic flowing through them. This led the networking community in the introduction of many new technologies and architectures to tackle this problem, one of which is software defined networking, commonly known as SDN. This proved to be a big step towards better network management, but it might not be enough to withstand the future increase in data and traffic. The use of Artificial Intelligence is able to provide this architecture with the required power to become a better and more future proof solution. Some of the advantages that we are able to see from our experiments with a DQN model on the SDN paradigm are that:

- Since it belongs to a set of Artificial Intelligence algorithms called "General Purpose Algorithms", makes it perfect for use with SDN because the same algorithm can be applied to a large range of problems and challenges that appear in data centers with the same efficiency and effectiveness since the goal remains the same, i.e. better network traffic management.
- It is environment driven and thus dynamic. It relies on a continuous analysis on the state of the network to make better decisions that will affect the network positively.
- It is very often that the efficiency of the network relies on hidden or unpredictable factors where for example in some cases part of the network is hidden from the core. Unlike traditional algorithms that are not able to detect such occurrences, Neural Networks are able to make use of all the available information derived from collected data, to detect hidden patterns or issues, adapt in those situations and provide decisions that will overcome the problems.

Our load balancing method was based on decisions taken by the trained NN model that used monitoring data collected from the underlying network while traffic was flowing through it. The model is able to accurately decide the most appropriate server to handle requests coming from applications. The network shows an overall better performance during tests that were made by stressing it continuously sending multiple packets. The model seems to almost always choose the fastest option to serve a packet and the load is distributed evenly in accordance to the individual capacity of the servers. Compared to traditional algorithms for load-balancing, it was noticed that there was a big increase in the total number of packets served by the network. It should be noted that that this method can be optimized even more to produce much better results that are almost impossible to be produced by the traditional algorithms.

For instance, if the full set of capabilities that Software Defined Networks offer is considered, such as information regarding the client application, client preferences, etc, in conjunction with the ability to manage multiple types of services the potential of the approach is greatly enhanced. This can results to a network, which depending on the needs of the client and the requirements of the specific application, will be able to adapt how the data center delivers its services. This way we can achieve a fully automated SDN that is capable of making important network and traffic decisions autonomously, without human intervention, which combined with newer approaches, such as intent based networking can prove to bring significant improvements in the area.

The most significant limitation of the current approach lies in its centralised nature, which in addition to the introduction of a single point of control makes it prone to errors. Future research towards this direction could provide a synchronization of additional points of control in order to minimize such effects. Another important research direction is that of detecting the addition or removal of network devices making the solution suitable for dynamically changing network topologies.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.* 38, 4 (Aug. 2008), 63–74. https://doi.org/10.1145/1402946.1402967

[2] Cui Chen-Xiao and Xu Ya-Bin. 2016. Research on load balance method in SDN. *International Journal of Grid and Distributed Computing* 9, 1 (2016), 25–36.

[3] David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. 2003. A Knowledge Plane for the Internet. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/863955.863957

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

[5] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[6] Kate Greene. 2009. TR10: Software-Defined Networking. http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/#comments

[7] Faris Keti and Shavan Askar. 2015. Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. In *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*. IEEE, Kuala Lumpur, 205–210. https://doi.org/10.1109/ISMS.2015.46

[8] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2014. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103, 1 (2014), 14–76.

[9] Charles E. Leiserson. 1985. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Trans. Comput.* 34, 10 (Oct. 1985), 892–901.

[10] Shih-Chun Lin, Ian F Akyildiz, Pu Wang, and Min Luo. 2016. QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach. In *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, San Francisco, 25–33. https://doi.org/10.1109/SCC.2016.12

[11] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling innovation in campus networks. *Computer Communication Review* 38 (04 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[12] Albert Mestres, Alberto Rodriguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J. Hibbett, Giovani Estrada, Khaldun Ma'ruf, Florin Coras, Vina Ermagan, Hugo Latapie, Chris Cassar, John Evans, Fabio Maino, Jean Walrand, and Albert Cabellos. 2017. Knowledge-Defined Networking. *SIGCOMM Comput. Commun. Rev.* 47, 3 (Sept. 2017), 2–10. https://doi.org/10.1145/3138808.3138810

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs.LG]

[14] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. 2012. A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms. In *Proceedings of the 2012 Second Symposium on Network Cloud Computing and Applications (NCCA '12)*. IEEE Computer Society, USA, 137–142. https://doi.org/10.1109/NCCA.2012.29

[15] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[16] Sheetanshu Rajoriya. 2014. Load Balancing Techniques in Cloud Computing: An Overview. *International Journal of Science and Research (IJSR)* 3, 7 (2014), 1616–1623.

[17] Alex M. R. Ruelas and Christian Esteve Rothenberg. 2018. A Load Balancing Method Based on Artificial Neural Networks for Knowledge-Defined Data Center Networking. In *Proceedings of the 10th Latin America Networking Conference* (São Paulo, Brazil) *(LANC '18)*. Association for Computing Machinery, New York, NY, USA, 106–109. https://doi.org/10.1145/3277103.3277135

[18] Richard S. Sutton. 1992. Introduction: The Challenge of Reinforcement Learning. In *Reinforcement Learning*, Richard S. Sutton (Ed.). Springer US, Boston, MA, 1–3. https://doi.org/10.1007/978-1-4615-3618-5_1

[19] Deepal Tennakoon, Suneth Karunarathna, and Brian Udugama. 2018. Q-learning approach for load-balancing in software defined networks. In *2018 Moratuwa engineering research conference (MERCon)*. IEEE, Moratuwa, 1–6.

[20] Liehuang Zhu, Md Monjurul Karim, Kashif Sharif, Fan Li, Xiaojiang Du, and Mohsen Guizani. 2019. SDN Controllers: Benchmarking & Performance Evaluation. arXiv:1902.04491 [cs.NI]