AROM-DRL: An Adaptive Routing Optimization Model for QoS-Aware Software Defined Networks using Deep Reinforcement Learning

Tareq Tayeh, *Member, IEEE,* Michael Rist, *Member, IEEE,*
{ttayeh, mrist}@uwo.ca

***Abstract*—Software Defined Networking (SDN) has been recognized as the next-generation networking paradigm that decouples the data plane and control plane, allowing network resources to be managed by a logically centralized controller. The inclusion of Machine Learning (ML) techniques can improve network optimization and the automated provisioning of the network's service capabilities, as well as enhancing the SDN's ability to fulfil Quality of Service (QoS) requirements in a variety of applications. In particular, the recent emergence of Deep Reinforcement Learning (DRL) allowed more complex problems with high-dimensional state and action space to be solved, making them ideal for Routing Optimization (RO) in complex network environments with rapid changes in continuous time. In this paper, we introduce an Adaptive RO Model for QoS-aware SDNs using DRL (AROM-DRL). AROM-DRL dynamically considers various QoS parameters, such as latency, throughput, packet loss, and jitter, in combination with statically determined parameters, to generate a powerful and dynamically determined action-reward strategy for the DRL system as part of an iterative RO mechanism. In a QoS-aware SDN system, network operators and service providers can use AROM-DRL to assist in offering high-quality services to increase customer satisfaction and reduce customer churn. Future work will include an AROM-DRL implementation to be evaluated against both SDN and non-SDN benchmarks, which have been discussed, implemented, and evaluated in this paper. All code can be found at https://github.com/TareqTayeh/Adaptive-Routing-Optimization-for-QoS-aware-Software-Defined-Networks-using-Deep-Reinforced-Learning.**

***Index Terms*—Software Defined Networking, Machine Learning, Deep Reinforced Learning, Routing Optimization, QoS.**

I. Introduction

Recently, with the explosive growth in the use of data-driven smart devices and communication technologies, the amount of data traversing the world's networks has been on an exponential trajectory. Furthermore, the nature of these smart technologies has led to increasingly more complex, heterogeneous network infrastructure and communication requirements, which in turn create a number of challenges. One of these challenges is traffic Routing Optimization (RO), which plays a key role in traffic engineering. RO is concerned with organizing and managing network traffic and resources, and finding efficient traffic routes to maximize network performance. A possible solution to the RO challenge is to bring intelligence into the traditional network for improved network efficiency and user experience [1], [2]. The highly distributed nature of traditional networks, however, make the application of any overarching network intelligence difficult. Fortunately, Software Defined Networking (SDN) allows us to bridge this problematic gap.

SDN has been recognized as the next-generation networking paradigm that decouples the data plane and control plane. The network resources in SDN are managed by a logically centralized controller, which has a global view of the network by monitoring and collecting real-time network state and configuration data, as well as packet and flow-granularity information, resulting in a dynamically programmed network [3]. However, in spite of the SDN's promising features, developing a reliable end-to-end transportation network upon them is difficult due to the requirements of various Quality of Service (QoS) provisioning and fast route configuration [4]. SDNs should fulfil various QoS requirements, such as delay, packet loss, jitter, and throughput, in a variety of applications. It should also provide fast and adaptive data transportation in order to react to changing network topology and traffic statistics in real-time. Therefore, in order to overcome these challenges, it is vital that any solution should provide a time-efficient, QoS-aware traffic routing mechanism.

One efficient way of overcoming the aforementioned challenge involves integrating SDNs with Machine Learning (ML) techniques, due to the latter's data-driven nature, efficiency, and ability to perform data analysis without explicitly programming the application [5]. In addition, ML provides the ability for network optimization and automated provision of network services. This powerful ML-SDN integration enables optimal network solutions, such as configurations and resource allocations, to be executed on the network in real-time [6].

There are three broad categories of ML methods: supervised learning, unsupervised learning, and semi-supervised learning [7]. Supervised learning is where an algorithm is trained to learn a mapping function from the input data to the output variables, whereas in unsupervised learning an algorithm is trained on unlabeled input data with not output variables. Semi-supervised aims to provide a medium between supervised and unsupervised learning, utilizing a combination of labeled and unlabeled data in the training phase. This is particularly useful in scenarios where labeled data is difficult or expensive to obtain, such as in the medical industry. However, this approach focuses on learning underlying structures and does not learn from previous experiences, making it not adaptable to lag-free, real-time decision making [8].

The recent emergence of Reinforcement Learning (RL) [9] focuses on making decisions based on previous experiences in real-time. RL is an iterative behavioral learning model that does not undergo a training phase with any labeled training data, where the system learns through trial and error instead. Deep Reinforcement Learning (DRL) [10] combines RL with the more complex, layered Neural Networks (NNs) associated with Deep Learning (DL) methods. DRL addresses some of the challenges faced by RL by leveraging the deep NNs capabilities to approximate the value function [6]. DRL possesses the ability to solve problems with high-dimensional state and action space, as well as a more robust convergence rate to optimal behavior policy. These abilities make integrating SDNs with DRL ideal for RO in more complex environments with rapid changes in continuous time.

Many previous approaches that utilize DRL as an RO mechanism chose statically determined parameters that form the reward for the DRL agent in the SDN system [11], [12]. Examples of these parameters often include current traffic conditions, traffic configurations, network status or individual metrics such as latency, throughput or packet loss. While these parameters are extremely useful features to use for DRL-SDN decisions, the use of a statically set strategy does not fully realize the potential of traffic RO.

To increase the efficiency and fully realize the potential of traffic RO in a network, we propose an Adaptive RO Model for QoS-aware SDNs using DRL (AROM-DRL). By dynamically considering various QoS parameters, such as latency, throughput, packet loss, and jitter, in combination with statically determined parameters, a powerful and dynamically determined action-reward strategy is generated for the DRL system as part of an iterative RO mechanism, further enhancing the overall RO strategy, network performance and traffic delivery. AROM-DRL aims to minimize latency, jitter, and packet loss, and aims to maintain a consistent throughput value and bandwidth availability. In a QoS-aware SDN system, network operators and service providers can use AROM-DRL to assist in offering high-quality services to increase customer satisfaction and reduce customer churn.

The proposed AROM-DRL solution includes the preliminary development of a mathematical model that can be used to adaptively generate a reward-strategy. The Deep Deterministic Policy Gradient (DDPG) [13] algorithm will be integrated within AROM-DRL to optimize the model. DDPG is an off-policy algorithm that can be used for environments with continuous action spaces. DDPG concurrently learns a Q-function, the tail distribution function of the standard normal distribution, and a policy.

At this stage, analysis and implementation of AROM-DRL has only been conducted mathematically and not via simulations or emulations due to time restrictions. In future work, AROM-DRL will be evaluated and compared against two benchmarks: A non-SDN controller-free environment and an SDN controlled environment. However, analysis of the two benchmarks have been conducted via simulations already and results will be discussed in this paper. Both networks were simulated using Mininet [14] in a Ubuntu [15] Virtual Machine (VM) configured with Open vSwitch [16], and Floodlight [17] was utilized as the SDN controller in the SDN controlled environment. Distributed Internet Traffic Generator (D-ITG) [18] was utilized to generate realistic multi-flow, multi-host traffic between hosts in the environment [19].

The remainder of this paper is organized as follows. Section II presents the motivation behind the use of DRL and the related work in the field of RO. Section III illustrates the technical contribution of the paper. Section IV contains background information about the tools used. Section V details the general principles of AROM-DRL. Section VI discusses the methodology and implementation of the two benchmarks. Section VII demonstrates and discusses the obtained results and performance evaluation of the two benchmarks. Section VIII concludes the paper. Finally, Section IX thoroughly discusses opportunities for future work.

II. Related Work

Stampa, *et al* [11], uses an off-policy, DDPG [13] approach to develop their DRL [10] agent. The DRL agent takes in a state in the form of a Traffic Matrix (the bandwidth request between each source-destination pair), then takes action through link-weights that act to determine a path for all source-destination pairs. Finally, the reward is based on mean network delay. Although the proposed solution is effective towards the goal of minimizing network delay, the reward strategy is static and unable to effectively address other common network QoS requirements. This is the case in Maheswari, *et al* [12] as well, where the reward function is based on a single QoS metric (delay). Maheswari, *et al* state that how to delvelop a strategy is still an open question.

Yu, *et al* [20], as with Maheswari, *et al*, demonstrate that generating a strategy is an open question for their DRL-based routing mechanism. They believe that by considering QoS-aware traffic classification, alongside state network measurement, they can develop a strategy that is adaptively generated for a reliable and effective end-to-end transport. Bouzidi, *et al* [21] lay out a weighted reward formula $r = \alpha$ .W - β.L - γ.PL, where QoS parameters: Latency (L), Rate (W) and Packet Loss (PL) are considered. α, β and γ (contained in [0,1]) are the adjustable weights determined by the routing strategy. The objective is to determine the optimal policy for mapping a correlation of states to actions to maximize the reward *r*, but their QoS parameters are statically determined. Pham, *et al* [22] use a DRL agent, which interacts with its SDN network through state, action and reward. In their framework, state is represented with a Traffic Matrix, action is realized through a link-weight vector, and reward is the mean of pre-determined QoS metrics and number of qualified flows. Thus, the reward-strategy is to minimize the mean of QoS metrics and maximize the number of qualified flows. However, their reward-strategy iss fixed and not adaptively generated.

Lin, *et al* [23] propose a QoS-aware Adaptive Routing (QAR) via reinforcement learning in multi-layer hierarchical SDNs by dynamically considering various QoS metrics. They demonstrate that QAR has fast adaptation to the network states, distributes traffic loads well and has great scalability. This work is the closest to our proposed solution, however, we will be utilizing different routing strategies and considering different features in our approach.

III. Contribution

Through our proposed solution and the preliminary implementation of two comparative benchmarks, the main contributions of this paper include:

1) Introduce and analyze AROM-DRL, an adaptive RO model for QoS-aware SDNs using DRL [10] and DDPG [13], that aims to minimize latency, jitter, and packet loss, and aims to maintain a consistent throughput value and bandwidth availability.
2) Effectively simulate a realistic multi-flow, multi-host traffic environment using Mininet [14], a Floodlight [17] SDN controller, and D-ITG [18] for both a non-SDN, controller-free environment and an SDN controlled environment.
3) Effectively extract and visualize QoS metrics from continuous multi-flows via the D-ITG toolset.

IV. Background

In this section, we introduce and discuss the tools used to emulate the networks for the two benchmarks.

A. Mininet

There are various software platforms that allows network topologies to be emulated. In this paper, Mininet [14] was used as our evaluation test bed, due to its widely recognized realistic emulator for deploying large networks. Mininet is a Linux-based emulation software that creates a realistic virtual network, running real kernel, switch and application code, on a single machine in seconds. The Linux containers are interconnected via virtual links that can be configured with delay, jitter, packet loss and bitrate parameters. Mininet provides performance accuracy and scalability without the need for real simulations and shared hardware test beds. Furthermore, the Mininet emulation software can run predefined network topologies with a single command, or user-defined network topologies via Python scripts.

B. Floodlight SDN Controller

Floodlight [17] is an open-source, community-developed, Java-based SDN controller that supports OpenFlow protocols and orchestrates traffic flows in an SDN environment. It is one of the most popular open source SDN controllers due to its nature and functionalities. The Floodlight controller has a modular architecture and exposes a set of powerful REST APIs that can be utilized effectively by separate, northbound applications for network management and control applications. Figure 1 visualizes the Floodlight SDN controller architecture.

C. D-ITG

D-ITG [18] is a platform capable of producing IPv4 and IPv6 traffic by accurately replicating the workload of current Internet applications and by following stochastic models for packet size and inter-departure time that mimic application-level protocol behavior. Moreover, D-ITG is capable of measuring the most common QoS performance metrics at the packet level, acting as a network measurement tool. It supports Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), and a couple other protocols at the transport layer. Figure 2 visualizes the architecture of D-ITG, which comprises of five main components utilized in this paper: ITGSend, ITGRecv, ITGLog, ITGDec, and ITGPlot.

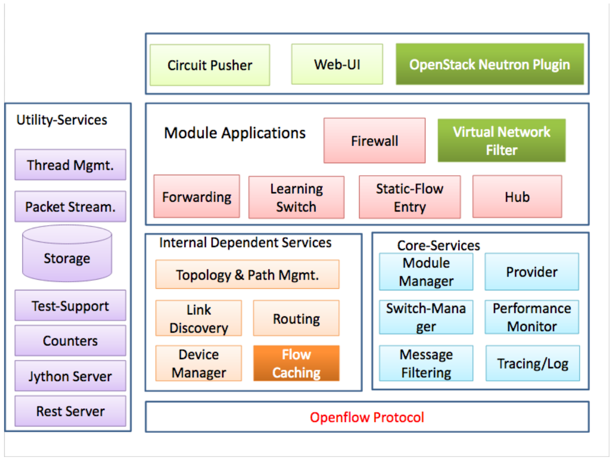


Fig. 1: Floodlight SDN Controller Architecture

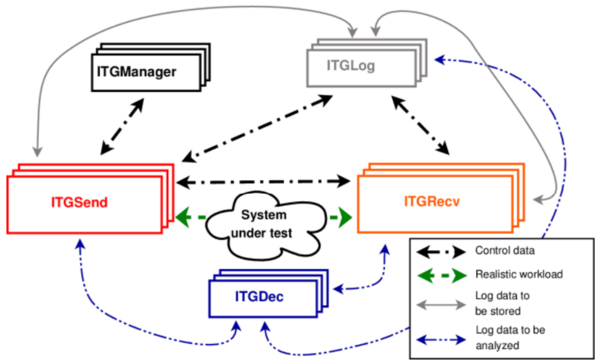


Fig. 2: D-ITG Architecture

1) ITGSend: The component responsible for generating single-flow or multi-flow traffic.
2) ITGRecv: The component responsible for receiving multiple parallel traffic flows generated by one of multiple ITGSend instances.
3) ITGLog: The component responsible for receiving and storing log information sent by ITGSend and ITGRecv.
4) ITGDec: The component responsible for decoding and analyzing the log files stored by ITGLog. The analyzed log file produces synthetic reports and sample QoS metrics timeseries for each flow and for the whole set of flows.
5) ITGPlot: The component that enables plots to be generated for the QoS metrics data stored by ITGLog and decoded by ITGDec.

V. Principles of AROM-DRL

The following section will first introduce and discuss the related principles of DRL [10], before detailing the implementation of the modified DDPG [13] algorithm used. Afterwards, the QoS-aware reward function design is introduced and detailed. Lastly, we introduce the AROM-DRL framework, which combines DRL, the modified DDPG, and the QoS-aware reward function all together.

A. General Principles of DRL

DRL is a subfield of ML that combines RL [9] and DL. RL tries to imitate the way a human or other intelligent being might interact with a new environment, by making decisions via trial and error. It is simply an agent learning entity, which interacts with its environment to identify the best action series in order to maximize a given objective function. The RL framework operates through 3 feedback signals: action, state and reward. The action is what the agent selects from a list of potential actions when interacting with the environment, where the agent can observe this change and use it as a feedback signal to learn from. These changes that the agent observes after an action takes place are changes to the state of the environment. The new state that the agent observes may generate a reward signal, which would either positively or negatively reinforce the mechanism for the agent. The more the agent interacts with the environment, the more it learns. Combining together the action that the agent took, the change in state and the potential reward received from the change in state, the agent begins to build a working model for the environment that they are exploring.

DRL incorporates DL into the RL mechanism, allowing agents to make decisions from unstructured input data without manual engineering of state spaces. This powerful integration allows DRL algorithms to optimize an objective by deciding the actions to perform when inputs are very large and high-dimensional, a problem that cannot be solved by traditional RL algorithms. Furthermore, with DRL agents, different reward functions can be used to implement different target policies, without the need of designing a new algorithm [11].

B. Modified DDPG

AROM-DRL uses a modified DDPG algorithm as its DRL algorithm. DDPG is an off-policy algorithm proposed by DeepMind to combine the continuous-time control and optimization capabilities found in Deterministic Policy Gradient (DPG) [24] with Deep Q Networks' (DQNs) [25] capabilities to generate more complex, non-linear Q-functions [20]. DDPG uses NNs to generate the strategy and Q functions, forming an efficient and stable discrete action control model that can be used for environments with continuous action spaces. Figure 3 depicts our modified DDPG algorithm's pseudocode.

```
Algorithm 1 Deep Deterministic Policy Gradient
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:   Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:   Execute $a$ in the environment
6:   Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:   Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:   If $s'$ is terminal, reset environment state.
9:   if it's time to update then
10:    for however many updates do
11:      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:      Compute targets
             $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:      Update Q-function by one step of gradient descent using
             $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:      Update policy by one step of gradient ascent using
             $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:      Update target networks with
             $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
             $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$
16:    end for
17:  end if
18: until convergence
```

Fig. 3: DDPG Pseudocode

The modified DDPG implementation used in this paper performs an enhanced exploration at the beginning of training. For a particular number of steps at the beginning, the agent takes actions that are sampled from a uniform random distribution over valid actions, before returning to the normal DDPG exploration. In this actor-critic architecture of DDPG, the actor module adopts the DPG method, while the critic module adopts the DQN method. The DDPG parameter updating process consists of updating the actor module of the neural network and updating the critic module of the DQN neural network, where the policy gradient for the actor module's NN is backpropagated.

C. QoS-Aware Reward Design

In this section, we propose a QoS-aware reward function that suits our design of QoS-aware routing. The agent finds the routing path with the maximum QoS-aware reward with regards to minimizing latency, jitter, and packet loss, and maintaining a consistent throughput value and bandwidth availability. Towards this, our QoS-aware reward function is proposed as follows:

$$
\begin{aligned}
&R_t := R(i \to j | s_t, a_t) \\
&= -g(a_t) \\
&+\beta_1 \Big[\theta_1\big(\tfrac{1}{delay_{ij}+\epsilon}\big) + \theta_2\big(\tfrac{1}{jitter_{ij}+\epsilon}\big)\Big] \\
&+\beta_2 \Big(1 - loss_{ij}\Big) \\
&+\beta_3 \Big(1 - \tfrac{throughput_{tij} - throughput_{t-1ij}}{throughput_{tij} - throughput_{t+1ij}}\Big) \\
&+\beta_4 \big(throughput_{tij} > \alpha\big)
\end{aligned} \tag{1}
$$

where R is the reward, i and j are the nodes, s_t is state s at time t, a_t is action a at time t, $g()$ denotes the cost to take action a_t that reveals the action input to switch operations, β_1, β_2, β_3, β_4, θ_1, $\theta_2 \in (0, 1)$ are the learnable weights determined by the QoS requirements of flow, ϵ is an extremely small number to avoid dividing by zero, and α is a predetermined minimum threshold set.

The DRL mechanism utilizes this reward formula to reinforce the aims of this paper. If we look into the formula line by line starting from β_1, delay and jitter are minimized by taking their inverse value, meaning smaller delay and jitter values result in a higher reward. Next, as loss is a percentage, the lower the packet loss, the lower the value that is subtracted from one, and the higher the resulting value inside the brackets will be, resulting in a higher reward. Next, a consistent throughput is positively reinforced, as a consistent throughput achieved between time intervals results in a higher value inside the brackets and the higher the reward. Finally, having throughput over a minimum threshold set allows the value inside the brackets to be a one rather than zero, reinforcing a positive mechanism for a high reward.

D. AROM-DRL

AROM-DRL combines all the previous subsections together. The novel model is an adaptive RO model for QoS-aware SDNs using DRL and the modified DDPG algorithm discussed in V-B, with an aim to minimize latency, jitter, and packetloss, and to maintain a consistent throughput value and bandwidth availability via the QoS-aware reward design discussed in V-C. The DRL component is integrated with the SDN controller via the exposed REST APIs to form the main agent in this model.

The general process of AROM-DRL can be summarized as followed: The SDN controller's network analysis and network environment interaction allows the AROM-DRL agent to obtain the network state and determine the best action series for a long-term reward. The optimal action is selected from the list of available actions, which is a set of link weights [w1, w2; : : :wn] used to route traffic between two nodes. The new paths of flows are recalculated based on the set of updated weights, the βs and θs, and the SDN controller generates new rules to establish the new paths. After the paths are updated, the reward and the new network state, which includes the current traffic matrix, the action taken, and the agent, are retrieved via the next network analysis and measurement. Hence, AROM-DRL allows the network performance to be iteratively optimized. Figure 4 visualizes AROM-DRL's architecture.

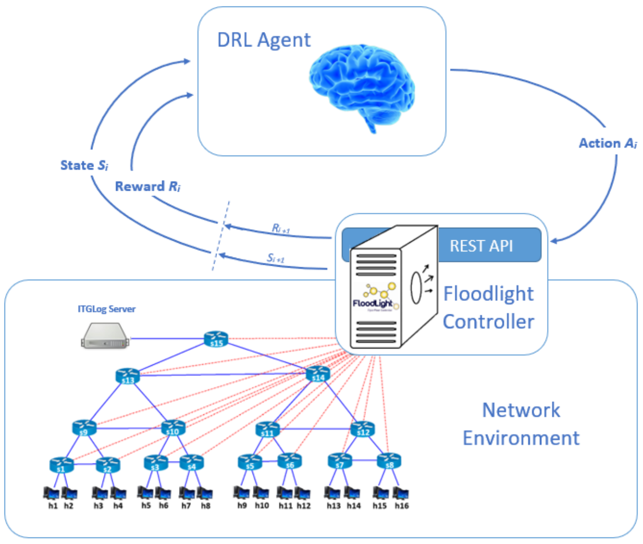


Fig. 4: AROM-DRL Architecture

VI. Comparative Benchmarks Methodology

In future work, AROM-DRL will be implemented and evaluated against two benchmarks: a non-SDN, controller-free environment and an SDN controlled environment. The following section will describe the non-SDN and SDN architectures respectively, followed by the D-ITG flows generated for evaluating the benchmarks. Afterwards, the benchmarks' implementations and evaluation metrics were discussed.

A. Comparative Benchmark Architectures

In order to ensure a level comparison across these environments, we have developed a standardized topology that can be implemented across both non-SDN and SDN architectures. The topology consists of 15 switches and 17 hosts arranged in a hybridization of tree and mesh topologies to emulate the structure of a real-world network with access, distribution/aggregation, edge, and core routers. Hosts 1 through 16 represent the clients in access networks connected to the Distribution Routers (DRs) represented by switches s1 through s8. The DRs are then connected to Edge Routers (PEs) s9 through s12, which in turn are connected to the Core Routers (P) s13 through s15. Host 17 is reserved as the D-ITG Log Server.

B. Non-SDN Architecture

Figure 5 describes the structure of the non-SDN, controller-free architecture. In Mininet, by default, topologies have a controller attached to all switches when launched and built. For the non-SDN architecture, in order to avoid this, all OpenFlow [16] switches were manually configured. This was done by manually adding flows to the flow tables and changing the bridge setting to normal, thus allowing them to work as L2 devices. Important to note that the bridge settings does not allow loops or spanning trees within the architecture, hence no links were added between switches on the same level.

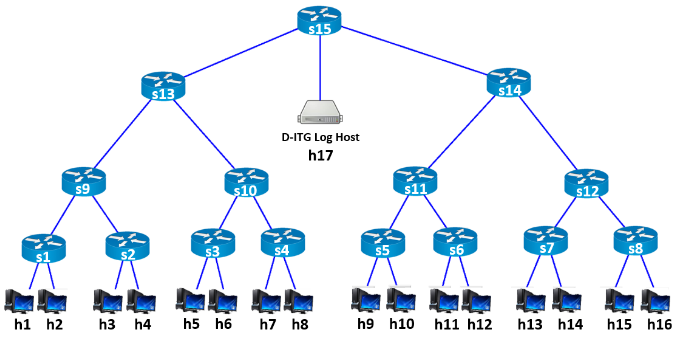


Fig. 5: Non-SDN Topography

C. SDN Architecture

Figure 6 describes the structure of the SDN benchmark architecture. Although similar topographically to the Non-SDN architecture, the SDN architecture detaches the control plane. A Floodlight controller (c0) was established and connected to each switch in the network, where it was set to default routing configuration, which uses Open Shortest Path First (OSPF). In this architecture, all functions and processes related to routing have been centralized to the controller.

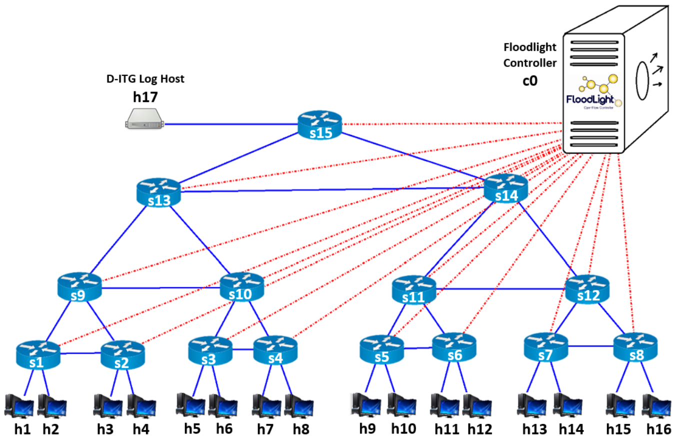
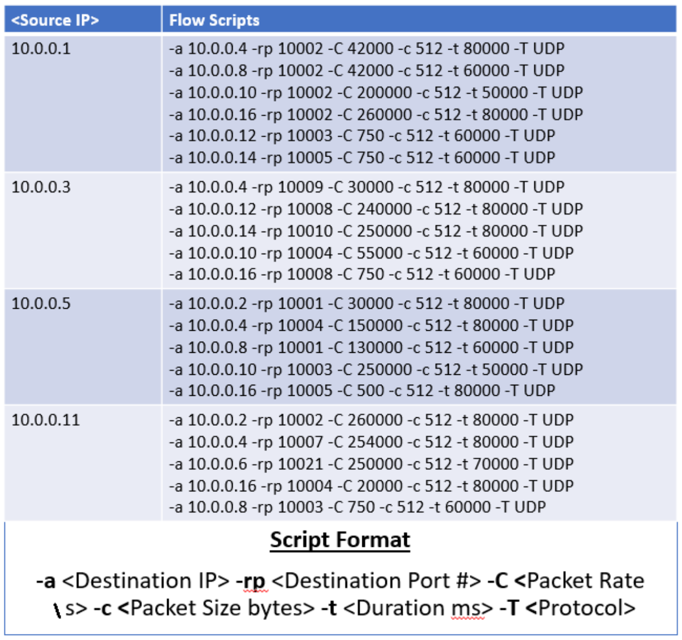


Fig. 6: SDN Topography



<Source IP>	Flow Scripts
10.0.0.1	-a 10.0.0.4 -rp 10002 -C 42000 -c 512 -t 80000 -T UDP -a 10.0.0.8 -rp 10002 -C 42000 -c 512 -t 60000 -T UDP -a 10.0.0.10 -rp 10002 -C 200000 -c 512 -t 50000 -T UDP -a 10.0.0.16 -rp 10002 -C 260000 -c 512 -t 80000 -T UDP -a 10.0.0.12 -rp 10003 -C 750 -c 512 -t 60000 -T UDP -a 10.0.0.14 -rp 10005 -C 750 -c 512 -t 60000 -T UDP
10.0.0.3	-a 10.0.0.4 -rp 10009 -C 30000 -c 512 -t 80000 -T UDP -a 10.0.0.12 -rp 10008 -C 240000 -c 512 -t 80000 -T UDP -a 10.0.0.14 -rp 10010 -C 250000 -c 512 -t 80000 -T UDP -a 10.0.0.10 -rp 10004 -C 55000 -c 512 -t 60000 -T UDP -a 10.0.0.16 -rp 10008 -C 750 -c 512 -t 60000 -T UDP
10.0.0.5	-a 10.0.0.2 -rp 10001 -C 30000 -c 512 -t 80000 -T UDP -a 10.0.0.4 -rp 10004 -C 150000 -c 512 -t 80000 -T UDP -a 10.0.0.8 -rp 10001 -C 130000 -c 512 -t 60000 -T UDP -a 10.0.0.10 -rp 10003 -C 250000 -c 512 -t 50000 -T UDP -a 10.0.0.16 -rp 10005 -C 500 -c 512 -t 80000 -T UDP
10.0.0.11	-a 10.0.0.2 -rp 10002 -C 260000 -c 512 -t 80000 -T UDP -a 10.0.0.4 -rp 10007 -C 254000 -c 512 -t 80000 -T UDP -a 10.0.0.6 -rp 10021 -C 250000 -c 512 -t 70000 -T UDP -a 10.0.0.16 -rp 10004 -C 20000 -c 512 -t 80000 -T UDP -a 10.0.0.8 -rp 10003 -C 750 -c 512 -t 60000 -T UDP

Script Format

-a <Destination IP> **-rp** <Destination Port #> **-C** **<**Packet Rate \ s> **-c** **<**Packet Size bytes> **-t** <Duration ms> **-T** **<**Protocol>

Fig. 7: D-ITG Multi-Flows

D. D-ITG Flows for Non-SDN and SDN Models

The D-ITG traffic generation toolset was utilized to support the evaluation of both the comparative benchmark architectures, and ultimately, the AROM-DRL solution once implemented. The multi-flow mode was used to generate multiple, concurrent flows from a single ITGSend host instance to multiple ITGRecv host instances. Multi-flow scripts were developed to effectively create a cross network traffic when run concurrently.

Figure 7 details four multi-flows coming out of four different sender hosts destined for multiple different receiver hosts across the network. The left column in the figure lists the source Internet Protocols (IPs) from which the multi-flow script will be run, while the right columns details the individual flows in the script. As an example, consider the first line of the first multi-flow script, which represents an individual flow from IP 10.0.0.1. Looking at the flow, it has a destination IP (-a) of 10.0.0.4, a destination port address (-rp) of 10002, an inter-departure time rate (-C) of 42000 packets/s, a packet size (-c) of 512 bytes/packet, a flow duration (-t) of 80000 ms, and uses (-T) the UDP Protocol for transport.

E. Implementation

The benchmark architectures, along with the supporting testbed environment, have been implemented on a Virtual Machine (VM) running Ubuntu 14.04, based on the pre-configured Floodlight VM vmdk image file. This environment is created through an Oracle VM Virtual Box v6.1. To launch and run simulations for the two comparative benchmark architectures, the VM is running Floodlight v1.2, Java 7 JDK, Python v2.7.6, Mininet v2.2.1, as well as D-ITG v2.8.1.

Simulations for the two benchmark architectures were launched and built in Mininet using the respective topology Python script. This sets up the network topology, starts the network, connects the switches to the Floodlight controller (SDN architecture), then builds the entire network. Important to note that for the SDN architecture, the Floodlight controller was run separately on the same VM prior to launching the network simulation.

Once the network simulation for either benchmark was running, individual host instances were made accessible using xterm. All the hosts in the Mininet simulation ran on a shared file system within the VM environment and had access to all applications therein. This meant that D-ITG could be installed in and accessed from the shared file system and could run via multiple hosts. In order to generate traffic using D-ITG multi-flow mode in the running simulation, an ITGLog host instance was initialized. The ITGLog host instance was the hub for all send and receive logs generated by the multi-flow traffic. ITGRecv receiver host instances were initialized before any flows can be generated, where they ran on the designated host listening for any incoming traffic. These receiver host instances opened transport protocol-specific ports, based on the incoming traffic flow. A single receiver instance is able to accommodate multiple incoming flows concurrently, provided the requested ports do not overlap. Finally, multi-flow ITGSend host instances were launched and ran to the completion of the multiple concurrent flows contained within the referenced script.

Its important to note that when creating the D-ITG scripts, the individual flow options were entered in the following order: [signaling_options] [flow_options] [inter-departure time_options] [packet sizes_options] [application_options], as depicted in Figure 7. Application options are always added at the end of each individual flow and can not be used in conjunction with any inter-departure time or packet size options.

F. Evaluation metrics

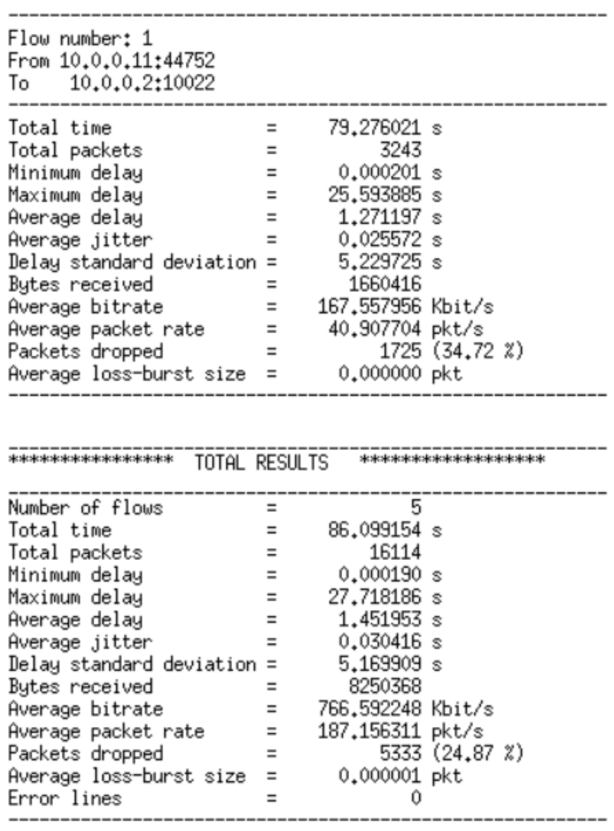
To measure the effectiveness of the two comparative benchmark architectures, and ultimately AROM-DRL in future work, several performance metrics were considered:

- Latency (delay), which should be minimized as close to zero as possible.
- Jitter, which should be minimized as close to zero as possible.
- Packet loss, which should be minimized as close to zero as possible.
- Bandwidth and throughput, which should show consistency, minimizing variance.
- Bandwidth availability, which should not fall below a specified threshold.

VII. Performance Evaluation

Metrics were collected by the ITGLog host through the generation of ITG log files. These were then decoded and analyzed with ITGDec. Afterwards, the decoded log files were visualized and further analyzed with the use of ITGplot.

An example of a partial ITGDec report for multi-flows from Host 11 to Host 2 is described in Figure 8. The report provides many QoS metrics measurements per flow and for the aggregate average of all the multi-flows.



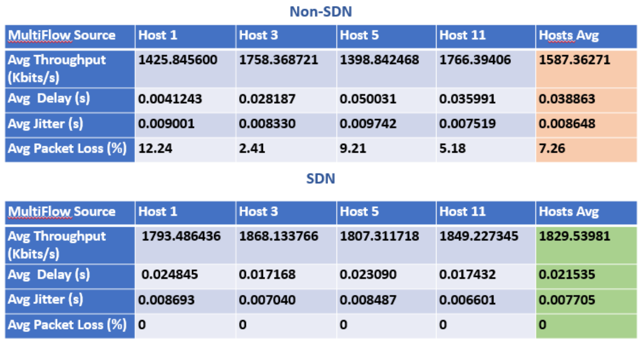
```
----------------------------------------------------------
Flow number: 1
From 10.0.0.11:44752
To    10.0.0.2:10022
----------------------------------------------------------
Total time                =     79.276021 s
Total packets             =          3243
Minimum delay             =      0.000201 s
Maximum delay             =     25.593885 s
Average delay             =      1.271197 s
Average jitter            =      0.025572 s
Delay standard deviation =      5.229725 s
Bytes received            =       1660416
Average bitrate           =    167.557956 Kbit/s
Average packet rate       =     40.907704 pkt/s
Packets dropped           =          1725 (34.72 %)
Average loss-burst size   =      0.000000 pkt
----------------------------------------------------------

----------------------------------------------------------
****************  TOTAL RESULTS   ******************
----------------------------------------------------------
Number of flows           =             5
Total time                =     86.099154 s
Total packets             =         16114
Minimum delay             =      0.000190 s
Maximum delay             =     27.718186 s
Average delay             =      1.451953 s
Average jitter            =      0.030416 s
Delay standard deviation =      5.169909 s
Bytes received            =       8250368
Average bitrate           =    766.592248 Kbit/s
Average packet rate       =    187.156311 pkt/s
Packets dropped           =          5333 (24.87 %)
Average loss-burst size   =      0.000001 pkt
Error lines               =             0
----------------------------------------------------------
```

Fig. 8: Partial ITGDec Report for Multi-Flows from Host 11 to Host 2

Furthermore, ITGDec was able to further probe into the specific metrics on a per flow and aggregate basis for each multi-flow script. The format was "ITGDec [log file] [metric option] [sample interval in ms] [output.dat file name]", where metric option was one of the following: -b (throughput), -d (delay), -j (jitter), -p (packet loss). For example, we were able to specify "ITGDec log-file -b 1000 outfile.dat". That took in the specified "log-file" and sampled the throughput data every 1000ms (1s), writing the resultant data to a .dat file. Armed with the metric-specific data in a .dat file, ITGplot was then used to convert and plot this data into an .eps file for visual interpretation and analysis. Similar to rest of the ITGDec, ITGplot was run outside of the simulation on the VM. The command format was "ITGplot [input.dat] [# of the flow to be plotted]"; if no flow number was specified all flows and their aggregate were included in the plot. For example "ITGplot input-file.dat 3", took in "input-file.dat" and created a plot (and .eps file) for flow "3".

During the simulations for the non-SDN and SDN comparative benchmark architectures, the same four cross-network multi-flows discussed in Section VI-D were run concurrently. The results were captured, decoded and analysed using the D-ITG toolset as specified earlier. Figure 7 shows four outgoing host multi-flows and their average QoS metrics for both the non-SDN and SDN environments. For both tables, the column to the right represents the total aggregate hosts' average, where the column is displayed in orange for non-SDN and in green for SDN. Overall, with the exception of average delay in the h1 sourced multi-flow, the SDN network outperformed the non-SDN network across the board. The aggregate average throughput for the SDN network was 15.26% greater than for the non-SDN. The aggregate average delay and jitter for the SDN network were 44.59% and 10.90% lower than for the non-SDN respectively. Finally, the tables showed that the SDN suffered no packet loss, while the non-SDN had an aggregate average of 7.26% packet loss.



Non-SDN

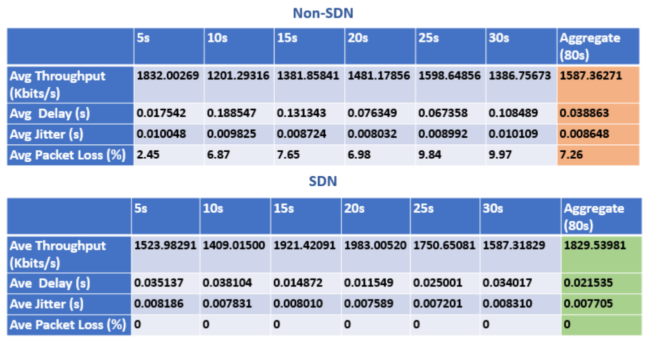
MultiFlow Source	Host 1	Host 3	Host 5	Host 11	Hosts Avg
Avg Throughput (Kbits/s)	1425.845600	1758.368721	1398.842468	1766.39406	1587.36271
Avg Delay (s)	0.0041243	0.028187	0.050031	0.035991	0.038863
Avg Jitter (s)	0.009001	0.008330	0.009742	0.007519	0.008648
Avg Packet Loss (%)	12.24	2.41	9.21	5.18	7.26

SDN

MultiFlow Source	Host 1	Host 3	Host 5	Host 11	Hosts Avg
Avg Throughput (Kbits/s)	1793.486436	1868.133766	1807.311718	1849.227345	1829.53981
Avg Delay (s)	0.024845	0.017168	0.023090	0.017432	0.021535
Avg Jitter (s)	0.008693	0.007040	0.008487	0.006601	0.007705
Avg Packet Loss (%)	0	0	0	0	0

Fig. 9: Non-SDN vs SDN Multi-Flows

To further dissect and analyze the time component in the data, the data was broken down into time segments with 5 second intervals, alongside the aggregate average, as displayed in Figure 10. This will be an important view for future comparison against the AROM-DRL solution, as DRL is time dependent as well. Looking into Figure 10, it can be noticed that the non-SDN got off to a slightly quicker start in the first 5 seconds. It is possible that was due to the fact that the non-SDN was manually configured with additions to the flow tables, allowing it to initially handle traffic flows more efficiently, while the SDN may initially take slightly longer to auto-update forwarding tables. However, in spite of this, the SDN quickly outperformed the non-SDN from second 10 till the end.



Non-SDN

	5s	10s	15s	20s	25s	30s	Aggregate (80s)
Avg Throughput (Kbits/s)	1832.00269	1201.29316	1381.85841	1481.17856	1598.64856	1386.75673	1587.36271
Avg Delay (s)	0.017542	0.188547	0.131343	0.076349	0.067358	0.108489	0.038863
Avg Jitter (s)	0.010048	0.009825	0.008724	0.008032	0.008992	0.010109	0.008648
Avg Packet Loss (%)	2.45	6.87	7.65	6.98	9.84	9.97	7.26

SDN

	5s	10s	15s	20s	25s	30s	Aggregate (80s)
Ave Throughput (Kbits/s)	1523.98291	1409.01500	1921.42091	1983.00520	1750.65081	1587.31829	1829.53981
Ave Delay (s)	0.035137	0.038104	0.014872	0.011549	0.025001	0.034017	0.021535
Ave Jitter (s)	0.008186	0.007831	0.008010	0.007589	0.007201	0.008310	0.007705
Ave Packet Loss (%)	0	0	0	0	0	0	0

Fig. 10: Non-SDN vs SDN Aggregate Time Segment

Figures 11 and 12 further represented the QoS metrics in the data as time series plots for the non-SDN and SDN network simulations, respectively. In comparing between the two networks, we can see how some patterns emerged. As described earlier, the initial throughput and delay of the non-SDN simulation seemed to be better initially, however, once past the 10 second mark, the SDN performance improved for both of these metrics. Furthermore, throughput appeared to remain more consistent throughout the simulation for the SDN network. Although delay spiked around the 60 second mark into the simulation for both SDN and non-SDN, the SDN network appeared to better suppress the extent and duration of the increased delay. In examining jitter, although the SDN network outperformed the non-SDN network, the differences present were more subtle. Finally, in looking at packet loss, the difference was clear, with the non-SDN presenting more loss, specifically during the same period where delay increased and throughput decreased.

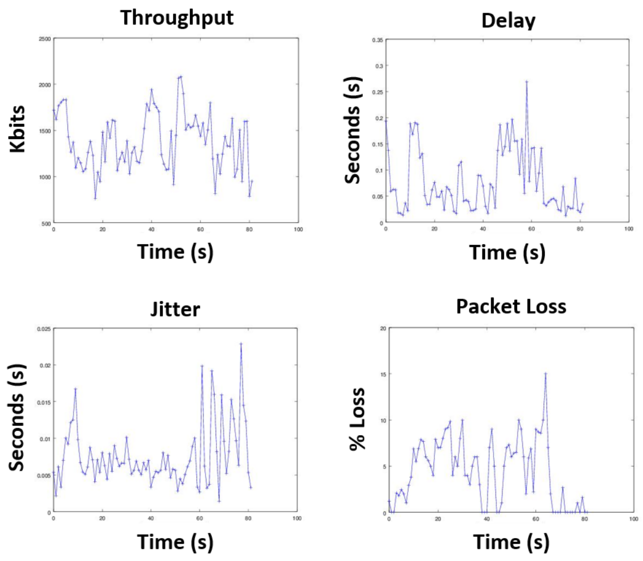


Fig. 11: Non-SDN ITGPlots

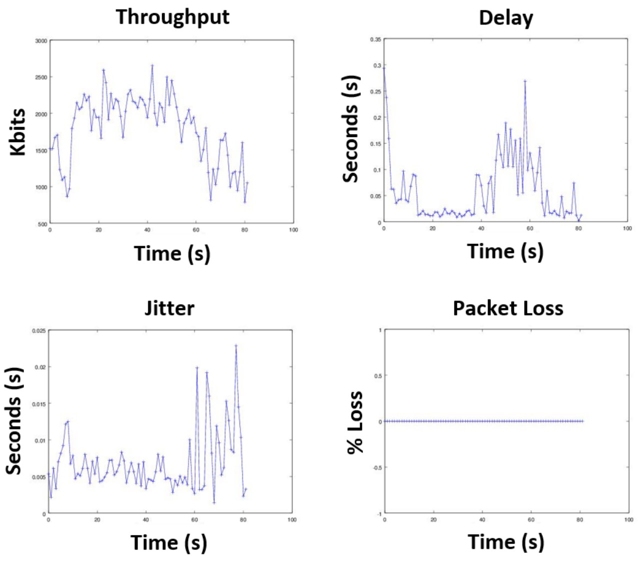


Fig. 12: SDN ITGPlots

VIII. Conclusion

In this paper, we introduced and detailed an adaptive RO model for QoS-aware SDNs using DRL called AROM-DRL. AROM-DRL aims to increase the efficiency and fully realize the potential of traffic RO in a network, by dynamically considering various QoS parameters in the action-reward strategy. In addition, we effectively demonstrated to implement and simulate a realistic multi-flow, multi-host traffic environment using Mininet, a Floodlight SDN controller, and D-ITG for the two benchmarks: a non-SDN controller-free environment and an SDN controlled environment. Performance evaluation for the two benchmarks demonstrated how the SDN architecture performed better than the non-SDN architecture, where QoS metrics from continuous multi-flows for both benchmarks were effectively extracted and visualized via the D-ITG toolset.

IX. Future Work

A. AROM-DRL Implementation and Evaluation

In order to evaluate and validate AROM-DRL, it has to be implemented. AROM-DRL will be implemented in Python, and there are many different Python-based packages out there that support a DRL implementation, such as Keras [26], TensorFlow [27], and PyTorch [28]. Keras is a powerful open-source library that provides high-level ML APIs and can act as the interface for the backend Tensorflow library, which is an open-source library developed by the Google Brain Team for numerical computation and large-scale ML. PyTorch is an open source ML framework developed by Facebook's AI Research lab that accelerates the path from research prototyping to production deployment.

The DRL mechanism will be integrated with the Floodlight SDN via the exposed set of REST APIs, which can be used to extract many useful statistics and metrics from the network environment, as well as modify certain network parameters on the fly. For example, Floodlight provides a Static Flow Entry Pusher application and a Circuit Pusher application that allow users to proactively install forwarding paths in the network. Static Flow Entry Pusher allows installing flow entries switch by switch, thereby creating forwarding paths based on the explicit choice of switch-ports by the AROM-DRL agent. CircuitPusher builds upon Static Flow Entry Pusher, Device Manager, and Routing services based on their REST API to build single shortest path circuits within a single OpenFlow island. Afterwards, AROM-DRL will be evaluated against the two benchmarks discussed in Section VI and against the evaluation metrics detailed in Section VI-F.

B. More Advanced Network Topology

In order to further evaluate and contrast the two benchmarks and AROM-DRL, more complex topologies will be explored.

C. More Advanced D-ITG Flows

Once AROM-DRL has been implemented, we are interested in looking to continue our exploration of more advanced D-ITG flows that are capable of emulating even more realistic traffic driven down from the application layer. We believe this would provide an interesting tool for further evaluating our solution and benchmark architectures. D-ITG is capable of generating application layer traffic which is passed down through transport protocols, such as TCP and UDP, to create more dynamic and realistic flows. These flows follow stochastic modelling of authentic traffic, which include Telnet sessions, DNS, several VoIP standards, and Quake3 Arena protocol traffic, where the latter emulates actual network gaming traffic. By creating and running these concurrent multi-flows with destinations spanning the network topology, we can also generate base level training data for AROM-DRL. There are limitations to this approach, as it does not represent the full spectrum of real network traffic, however, it has the potential to provide a much stronger starting point for generating training data than would have been possible otherwise.

References

[1] G. Xu, Y. Mu, and J. Liu, "Inclusion of artificial intelligence in communication networks and services," *ITU J., ICT Discoveries, Special*, no. 1, pp. 1–6, 2017.

[2] S. Sendra, A. Rego, J. Lloret, J. M. Jimenez, and O. Romero, "Including artificial intelligence in a routing protocol using software defined networks," in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 670–674, IEEE, 2017.

[3] I. F. Akyildiz, P. Wang, and S.-C. Lin, "Softair: A software defined networking architecture for 5g wireless systems," *Computer Networks*, vol. 85, pp. 1–18, 2015.

[4] A. Yassine, H. Rahimi, and S. Shirmohammadi, "Software defined network traffic measurement: Current trends and challenges," *IEEE Instrumentation & Measurement Magazine*, vol. 18, no. 2, pp. 42–50, 2015.

[5] S. Kumar, G. Bansal, and V. S. Shekhawat, "A machine learning approach for traffic flow provisioning in software defined networks," in *2020 International Conference on Information Networking (ICOIN)*, pp. 602–607, IEEE, 2020.

[6] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2018.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] T. T. Lu, "Fundamental limitations of semi-supervised learning," Master's thesis, University of Waterloo, 2009.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[11] G. Stampa, M. Arias, D. Sánchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[12] D. N. Maheswari, C. Sujitha, and K. Ramana, "Routing optimization in sdn using deep reinforcement learning,"

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

[14] M-Team, "Mininet: A virtual network on your desktop," *Available: http://mininet.org/*.

[15] M. G. Sobell, *A practical guide to Ubuntu Linux*. Pearson Education, 2015.

[16] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 117–130, 2015.

[17] Project-Floodlight, "Floodlight controller: community-developed, java openflow controller," *Available: http://floodlight.openflowhub.org/*.

[18] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre, "D-itg distributed internet traffic generator," in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 316–317, IEEE, 2004.

[19] D. D. Kyuchukova and G. V. Hristov, "Conducting experiments using a platform for evaluation and assessment of sdn performance,"

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

[21] E. H. Bouzidi, A. Outtagarts, and R. Langar, "Deep reinforcement learning application for network latency management in software defined networks," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2019.

[22] T. A. Q. Pham, Y. Hadjadj-Aoul, and A. Outtagarts, "Deep reinforcement learning based qos-aware routing in knowledge-defined networking," in *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pp. 14–26, Springer, 2018.

[23] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "Qos-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *2016 IEEE International Conference on Services Computing (SCC)*, pp. 25–33, IEEE, 2016.

[24] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," 2014.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[26] F. Chollet *et al.*, "Keras," 2015.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, pp. 8026–8037, 2019.

Acknowledgment

We would like to thank Dr. Anwar Haque and TA Kirk Vander Ploeg for all their assistance in this project and all their hard work in the course. We would like to thank them for providing the space and platform for us to learn about Mininet, Floodlight, D-ITG, and the field of DRL; all which were areas we had no to very little knowledge of prior to the course. We are looking forward to further working on this project to either publish a conference or journal paper.

Tareq Tayeh received the B.E.Sc degree in Software Engineering from Western University, London, Canada, in 2018. He is currently pursuing the M.E.Sc degree in Software Engineering, with the Vector Institute Accredited collaborative specialization in Artificial Intelligence, with the Department of Electrical and Computer Engineering, Western University, London, Canada. He worked for IBM, Markham, Canada as a Software Developer intern between May 2016 and August 2017, and as a full time Lead DevOps Engineer between May 2018 and September 2019. His current research interests are in the areas of artificial intelligence, machine learning, anomaly detection, computer vision, image processing, time series, IoT, and cloud computing.

Michael Rist is currently pursuing an M.Sc degree in Computer Science, in the area of Software Engineering and Human Computer Interaction, from Western University, London, Canada. He has more than 12 years of progressive leadership experience within the IT Services and HCM Sectors (2004-2016), and more recently worked as a Technical Lead on the Schulich School of Medicine and Dentistry's 2019 Website Design and Refresh project. His current research interests are in the areas of swarm intelligence applications in software and cyberphysical systems, machine learning, model-driven design and development, and edge computing.