Topology-Aware, Performance-Driven Adaptive Routing in Software-Defined Networks Using Dual-Agent Reinforcement Learning

Deepthi Goteti[1*], Vurrury Krishna Reddy[2]
[1,2]Department of Computer Science & Engineering, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India-522302
E-mail: 2102031088@ kluniversity.in, vkrishnareddy@kluniversity.in
[*]Corresponding author

Keywords: Software Defined Networking (SDN), Reinforcement Learning, Policy-Gradient, Routing Optimization, R-Learner, R-Optimizer, QoS, Dijkstra Baseline, Network Topologies (Fat Tree, Abilene, Custom, Dense Mesh)

Received: April 26, 2025

This research explores adaptive routing in Software-Defined Networks (SDNs) using reinforcement learning. Two models—R-Learner (Q-learning) and R-Optimizer (policy-gradient)—are evaluated against the Dijkstra baseline across four topologies: Fat Tree, Abilene, Custom, and Dense Adaptive Mesh. Experiments run over 100 TCP/UDP traffic episodes using Mininet and the Ryu controller. Key metrics include throughput, jitter, round-trip time (RTT), and packet loss ratio (PLR). Statistically validated results show R-Optimizer outperforms R-Learner, achieving ~7.4% higher throughput, 44% lower jitter, 19.5% lower RTT, and >50% lower packet loss. Both models also surpass Dijkstra in throughput and delay reduction. These results support reinforcement learning as a viable approach for real-time SDN routing and future controller integration.

Povzetek: Članek predstavi dvofazni model za adaptivno usmerjanje v SDN s Q-learningom (R-Learner) in policy-gradient pristopom (R-Optimizer). Preizkusi v Mininetu na štirih topologijah pokažejo, da R-Optimizer izboljša prepustnost (~7,4 %) in izgubo paketov (>50 %) ter občutno prekaša Dijkstrov algoritem.

1 Introduction

The rapid demand for data-driven services, cloud platforms, and real-time applications has made network environments more complex and more challenging to manage. These conditions often overwhelm traditional routing methods. In response, researchers have turned to reinforcement learning (RL) as a tool within Software-Defined Networking (SDN) to support more adaptive, policy-based routing that can adjust to changing network demands [1]. Traditional network infrastructures, which rely on fixed, distributed routing setups, may frequently fall short of delivering the speed and flexibility when needed to maintain consistent Quality of Service (QoS) [2]. SDN helps overcome these issues by separating the control and data planes, allowing for centralized control and flexible, real-time reconfiguration [3].

The SDN architecture comprises an application, control, and data plane interconnected via protocols like OpenFlow to support real-time management and policy enforcement [4]. Figure 1 illustrates the Architecture of the SDN system with integrated reinforcement learning. The R-Learner and R-Optimizer (RL Agent) reside in the Application Plane and interact with the SDN Control Software in the Control Layer via APIs to install routing decisions dynamically.

The Control Layer communicates with the network infrastructure using OpenFlow to manage network devices and enforce routing policies. The design maintains the traditional three SDN planes: the Application Plane, where the R-Learner and R-Optimizer agents calculate routing decisions by exploring path diversity under dynamic traffic patterns. The Control Plane is managed by the Ryu controller and extended with a custom module to enable the RL agents to install or update flow rules dynamically.

The Data Plane consists of OpenFlow switches that forward traffic based on the flow rules set by the controller. The RL agents monitor network statistics like link utilization and RTT through the southbound interface, select optimal routes, and then guide the controller to modify flow tables in real time. This separation of roles—keeping the learning logic in the application plane and the rule enforcement in the control and data planes—ensures greater modularity, makes the system easier to debug and allows for the easy integration of new learning strategies without changing the switch infrastructure. Learning effective routing policies has become critical as modern networks demand millisecond-level responsiveness and context-aware decision-making. This way drives the exploration of intelligent methods beyond traditional static or rule-based strategies.

While SDN significantly improves control capabilities,

real-time route optimization remains challenging [5]. Classical routing algorithms such as Dijkstra's and Equal-Cost Multi-Path (ECMP) often respond reactively to network changes and fail to adapt efficiently to link failures, congestion, or topological variation [6][45].

Khan et al. [7] Tested the performance of POX and RYU controllers using Dijkstra-based routing in SDN environments, demonstrating that RYU consistently provided lower latency and higher throughput under varying traffic loads compared to POX. The study emphasized that RYU's modularity and scalability make it better suited for implementing advanced routing strategies, including reinforcement learning while retaining Dijkstra as a comparative baseline for deterministic routing performance under static conditions.

Kumar and Thakur [8] Evaluated Ryu controller performance over Dijkstra, Bellman-Ford, and Floyd-Warshall algorithms using the RYU controller in SDN testbeds. Their findings showed that Dijkstra achieves lower RTT in stable topologies but struggles under dynamic traffic due to its static path selection, leading to congestion and packet drops. This showcased approaches in SDN and the need for adaptive, learning-based routing to address traffic variability and topological changes efficiently.

Naimullah et al. [9] analyzed the performance of POX and RYU controllers using Dijkstra's algorithm in SDN environments and reported that RYU outperformed POX in scalability and efficiency across larger topologies. However, the study also noted that Dijkstra's routing lacked adaptability under congestion, where link failures occur at certain times, also demanding the limitations of classical shortest-path approaches in dynamic SDNs and motivating the exploration of reinforcement learning for more responsive and robust routing strategies.

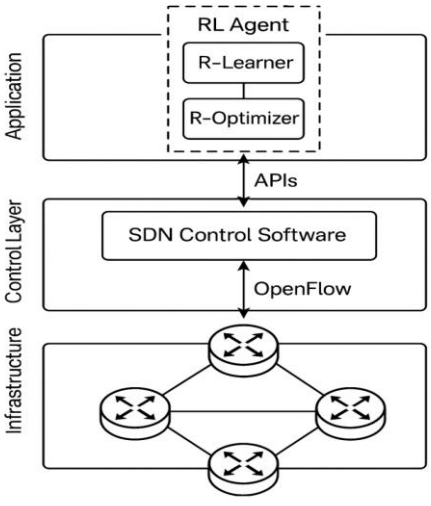

Figure 1: Architecture of SDN with reinforcement learning integration for dynamic routing

As an alternative, reinforcement learning (RL) offers the ability to learn adaptive routing policies by interacting with the environment and optimizing behavior based on cumulative rewards [10]. In recent years, researchers have integrated Artificial Intelligence (AI), particularly reinforcement learning, into SDN controllers to enhance proactive decision-making and traffic engineering [11], [12]. Deep reinforcement learning (DRL) models have shown impressive results in reducing packet loss, minimizing delays, also dynamically balancing traffic loads [13], [14]. Researchers have successfully deployed Graph Neural Networks (GNNs) and Convolutional Neural Networks (CNNs) beyond DRL to improve fault tolerance, detect anomalies, and enhance routing resilience in SDNs [15], [16].

Unlike previous studies that often evaluate RL models within a single, simplified network structure, this work adopts a topology-aware approach. Research Questions To guide this investigation, the study focuses on the following research questions:

- Can a staged reinforcement learning framework, leveraging an initial exploration phase followed by policy-gradient-based refinement, enhance routing efficiency across diverse software-defined network topologies under dynamic traffic conditions?
- How does the policy-gradient-based R-Optimizer compare with the exploration-phase agent (R-Learner) in terms of convergence speed and Quality of Service metrics—such as throughput, jitter, RTT, and packet loss—under TCP and UDP traffic across diverse network topologies?

To investigate the outlined research questions, this study introduces a dual-agent reinforcement learning framework designed for adaptive routing in software-defined networks (SDNs). The framework deals with two complementary strategies: Q-learning (R-Learner), which focuses on exploring the environment, and policy-gradient methods (R-Optimizer), which refine routing decisions based on observed performance under varying traffic conditions.

The approach is tested on four SDN topologies—Fat Tree, Abilene, Custom, and Dense Adaptive Mesh—to understand how network structure affects convergence rates, routing efficiency, and quality of service. These topologies were chosen specifically to represent a broad range of structural complexity, from the regularity of Fat Tree and Abilene to the irregular, high-density layouts of Custom and Dense Mesh. Such variation allows for a more nuanced evaluation of how well reinforcement learning adapts to different network environments. The broader motivation builds on recent work showing that deep reinforcement learning can offer strong results in SDN routing tasks [18, 20], and reinforces the idea that topology-aware testing is essential for drawing robust conclusions [17, 21].

The experimental environment is built using Mininet, which provides a scalable and flexible testbed [22]. Routing control is managed through the Ryu controller [23], and iPerf is used to generate realistic traffic patterns based on TCP and UDP protocols [24]. To benchmark the performance of the learning framework, a traditional Dijkstra-based routing strategy was implemented on the same setup, providing a clear point of comparison under identical network conditions.

We conduct our experiments in each topology with 100 simulation episodes. Moreover, selected samples were analyzed to evaluate performance across key quality of service metrics: throughput, RTT, jitter, and packet loss ratio (PLR). Results show that the policy-gradient-based R-Optimizer consistently performs better than the Q-learning-based R-Learner, particularly in topologies with higher redundancy and path diversity, such as Fat Tree while maintaining adaptability and stability across different network environments. These findings demonstrate how topological diversity significantly influences routing efficiency and learning behavior within reinforcement learning-based SDN routing.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents the proposed system. Section 4 describes the experimental setup. Section 5 details the results and analysis. Section 6 provides QoS comparison and section 7 is discussion. Finally, Section 8 concludes the paper and outlines future work

2 Related work

Software-defined networking (SDN) separates the control and data planes, enabling centralized programmability and abstraction of the underlying infrastructure [25]. The controller acts as the network's "brain," dynamically managing flow rules and routing. Due to its modularity in part design, Ryu stands out among available controllers, offering Python design, OpenFlow compatibility, and suitability for reinforcement learning (RL) integration [26].

Traditional SDN routing methods—like Dijkstra's algorithm and Equal-Cost Multi-Path (ECMP)—perform reliably under stable conditions but lack adaptability to congestion, link failures, and traffic surges [27]. Researchers have proposed reactive traffic-engineering solutions, but these still fail to meet real-time, low-latency demands [28]. Moreover, most traditional approaches do not learn from past network behavior, so they cannot improve or adapt over time. To deal with this, researchers have started using reinforcement learning (RL) and deep RL (DRL) for SDN routing [29], [30].

These models utilize deep Q-networks and policy gradient methods to adjust routing decisions based on real-time network feedback dynamically. The goal is to optimize long-term performance rather than reacting solely to short-term events. Studies show that they can boost throughput, reduce delay, and lower packet loss, even under unpredictable traffic loads [31]. Advanced methods like hierarchical and meta-RL improve how fast they learn and how well they handle different network situations [32], [33].

These models also reduce the need for manual tuning, making the system more autonomous and scalable.

Additionally, RL agents can continually adapt to traffic shifts over time without requiring a restart or reset of the entire network.

Recent approaches also include topology-aware learning where models make use of how the network is structured [33]. This helps to make rorouting decicion by the agent understand the role of links, paths, and node positions. Deep learning methods like transformers and convolutional neural networks (CNNs) have been used for traffic prediction, detecting anomalies, and improving routing during failures [35], [36].

These methods can catch early signs of congestion or link stress and adjust routing before problems get worse. Some setups use supervised learning first to teach the model basic patterns, then switch to reinforcement learning to fine-tune behavior in live scenarios. This shortens training time and improves stability.

Other designs use alert mechanisms or graph-based learning to help the agent focus on the most relevant parts of the network at any moment. This is useful when the network is large and links behave differently depending on traffic. These ideas are still developing but show good results in testbeds.

Mininet and the Ryu controller continue to be the go-to tools for testing this kind of setup [39], [40]. Mininet deals wiht complex topologies without needing physical switches, and Ryu makes it easy to plug in custom logic through its Python API. This allows routing agents to read traffic stats, update flow rules, and learn over time. There are open templates available for things like ECMP and custom controller functions, which help speed up testing and cut setup time [41], [42], [43]. These tools make it easier to repeat experiments and build on other researchers' work.

Despite these advances, a gap remains in frameworks that systematically assess RL-based routing across multiple SDN topologies. This study addresses that gap through a dual-agent architecture—R-Learner and R-Optimizer—evaluated on Fat Tree, Abilene, Custom, and Mesh networks, with a focus on convergence behavior and quality of service performance.

3 Proposed system

This section describes a structured SDN flow control framework that works on a two-stage routing process. The system introduces two coordinated modules—Route-Learner and Route-Optimizer—each responsible for a specific phase. In the first stage, the route learner probes the network to identify available paths and assess real-time network conditions. In the second stage, the Route-Optimizer takes the aid of this information to refine routing decisions, aiming to improve traffic distribution and reduce congestion. Both modules are integrated into the Ryu controller, enabling real-time interaction with the network topology. This setup allows the system to

periodically adjust forwarding rules based on updated path and traffic condition data, ensuring efficient routing even as network demands fluctuate. Our implementation followed a two-phase staged framework. In phase one, the R-Learner operated on the live SDN topology to collect QoS data and estimate routing values using Q-learning.

Table 1: Prior RL-Based SDN routing studies

Study	Method	Topologies	Metrics Evaluated	Key Findings	Identified Gaps
Liu [29]	Deep Q-Network (DQN)	Fat Tree	Throughput, Delay	Improved congestion handling	Single topology, no policy-gradient
Kim et al. [31]	DRL (Actor-Critic)	Custom Mesh	Latency, Packet Loss	Reduced delays and PLR	Limited topology diversity
Suh et al. [11]	DRL for Network Slicing	Data Center	Bandwidth, Delay	Dynamic slicing for QoS	Not focused on routing decisions
Chen et al. [33]	Multi-Agent RL	Custom	Load Balancing	Better fault tolerance	No policy-gradient exploration
Xie et al. [19]	GNN-RL	Mesh	Throughput	Captures topology structure	Limited scalability, single topology
Yang & Li [37]	Deep Reinforcement Learning (DRL)	Data Center SDN	Latency, Reward, Speed	Reduced latency by 6.3%, improved reward convergence	Focus on congestion control, lacks multi-topology routing analysis
Ma et al. [38]	Q-Learning	Generic SDN	Throughput, Delay, Load Balancing	Improved throughput by 30%, reduced delay by 25%	Uses Q-learning only, lacks policy-gradient comparison
Kim et al. [31]	Q-Learning + Policy-Gradient (Dual-Agent)	Fat Tree, Abilene, Custom, Mesh	Throughput, Jitter, RTT, PLR	Consistent QoS improvements across topologies	Multi-topology, staged policy-gradient use

Once exploration was complete, it was halted while the topology remained active. In phase two, the R-Optimizer refined routing decisions on the same topology using the Q-table generated by the R-Learner via a policy-gradient method. This structured approach ensured realistic traffic conditions while leveraging prior exploration.

For comparison, we also implemented Dijkstra's shortest-path algorithm in the Ryu controller and evaluated it under identical TCP and UDP traffic across all topologies. This provided a consistent baseline to measure improvements over the learning-based framework.

3.1 Reinforcement learning integration in SDN

Enhance the Ryu controller with custom RL modules to enable intelligent routing in SDN. When the system receives a packet_in event from the switch, it identifies the source–destination context and evaluates all feasible paths. It then selects a forwarding action based on the chosen agent's strategy—either R-Learner or R-Optimizer. In practice, this selection follows a staged approach: the system first operates with the R-Learner to explore and gather routing knowledge under live network conditions and, after sufficient exploration, transitions to using the R-Optimize**r** on the same active topology to refine routing decisions using the knowledge learned during the R-Learner phase. The system verifies each forwarding action.

Furthermore, it observes outcomes like delivery status, delay, or signs of congestion. We later use this information to improve future routing decisions through gradual adjustments. Updated routing choices are applied immediately by installing new flow rules using OpenFlow 1.0, maintaining smooth and responsive network operation.

Additionally, for baseline benchmarking, we implemented the Dijkstra shortest-path routing algorithm within the Ryu controller. We evaluated it across all topologies under the same traffic generation settings (either TCP or UDP) used for the reinforcement learning agents.

3.2 R-Learner: exploration-based adaptive routing

The R-Learner module defines predefined criteria to assess links and nodes within the network topology, selecting routing paths deemed efficient. It updates routing preferences based on the observed outcomes of previous forwarding decisions, using this performance history to guide future path selection. The state representation used in our implementation is limited to the source and destination switch identifiers (src_dpid, dst_dpid). It does not explicitly include live network metrics such as link utilization, RTT, or congestion

status. Instead, the agent learns effective routing paths through repeated exploration under dynamic traffic patterns, indirectly capturing performance through delivery outcomes.

- **State representation:** We represent each state using the tuple (src_dpid, dst_dpid), which identifies the source and destination switches.
- **Action:** An action corresponds to selecting one of the simple paths between the source and destination.
- **Reward:** In our current implementation, the reward is assigned as a scalar value, with a positive reward (e.g., +10) if the forwarding succeeds and a negative reward (e.g., -10) if the forwarding fails. While factors like throughput, latency, and congestion motivate the need for adaptive routing, we currently capture their indirect effects through packet delivery success and learning.
- **Q-Value update**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot a' \max Q(s',a') - Q(s,a)]$$

Where α is the learning rate, and γ is the discount factor.

Exploration strategy: The agent follows a ϵ-greedy approach, selecting a random path with probability ϵ and the best-known path otherwise.

Integration: Once a path is selected, flow rules are installed using install_path_flows(), and the Q-values are updated based on the observed reward.

Algorithm 1 outlines the decision-making process used by the Route-Learner module. The system supports routing decisions with the help of a performed path maintained in the evaluation table. Each routing path is selected based on current performance metrics linked with links and nodes. The process follows fixed rules for path selection, ensuring consistent routing behavior across the network topology.

Algorithm 1: R-Learner: Based Adaptive Routing
Input: Network topology GG, source-destination switch pair (s,d)(s, d), exploration rate ε, learning rate α, discount factor γ
Output: Updated Q-table Q, selected routing path p

1. Begin
2. Initialize Q-table
3. Q[state][action]arbitrarily
4. Observe current state S←(s,d)
5. If random() < ε then
6. a←select a random path from s to d
7. Else
8. a←argmaxQ[S][a] // select path with highest Q-value
9. End If
10. Install flow entries along path a using install_path_flows()
11. Forward packet through path a
12. Observe resulting reward r (based on delivery status, delay, congestion)
13. Observe next state S′←(s,d) // remains same unless topology changes
14. Update Q-value:
15. $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot a' \max Q(s',a') - Q(s,a)]$
16. Return selected path a and updated Q-table
17. End

3.3 R-Optimizer: probability-driven routing strategy

The R-Optimizer module applies a rule-based mechanism at each hop to determine routing decisions. It selects the next-hop option from the current switch based on predefined performance metrics and static path preferences, enabling consistent per-hop routing toward the destination.

- **State representation:** We represent states as (src, dst, current_switch), which includes the flow endpoints and the current switch in the path.
- **Action:** For each state, the agent creates a probability distribution over potential next-hop forwarding options from the current switch and samples from this distribution to select the next hop toward the destination.
- **Reward:** The reward is currently assigned using a simple scalar value based on the success of packet forwarding actions. We use a basic success/failure evaluation due to the lightweight nature of the optimizer in this implementation, with the indirect influence of path congestion or delay captured through delivery success.
- To formalize this approach, we designed the Route-Optimizer to improve long-term network performance, defined as:
- $J(\theta) = \mathrm{E}\tau \sim \pi\theta[\sum_{t=0}^{n} \mathrm{rt}]$
- Where θ represents the policy parameters, τ denotes trajectories under the policy πθ, and it is the reward at time t.
- Policy gradients for updating the policy are computed as:
- $\nabla_{\{\theta\}J(\theta)} = \mathrm{E}\tau_{[\sum T \nabla\theta \log\pi\theta(at|st)(Rt-b)]}$
- Where b is a baseline (e.g., the average reward) used to reduce variance during learning. This formulation enables the Route-Optimizer to apply predefined routing rules, which are based on accumulated performance data, ensuring consistent behavior across various network scenarios.
- **Policy update**:

$$P(s,a) \leftarrow P(s,a) + \alpha \cdot (r - b) \cdot \nabla \log P(s,a)$$

- **Adaptation**: Over time, the system assigns higher probabilities to high-performing actions while gradually avoiding ineffective routes..
- **Integration**: Routing decisions are enforced through add_flow() and OFPPacketOut, enabling real-time responsiveness.

Algorithm 2 summarizes the Q-Optimizer's policy-gradient approach. The agent gradually shifts preference toward higher-performing routes by sampling paths according to a softmax distribution and

updating the policy in proportion to the received reward minus a baseline. Immediate installation of new flow rules ensures that these learned improvements take effect in real time.

3.4 Topology discovery and path management

Both R-Learner and R-Optimizer need an up-to-date view of the network. We use Ryu's EventSwitchEnter to catch new switches and links and maintain a live graph in NetworkX. The agents then run over this graph to make their routing choices.
Operational loop:

1. **Topology discovery** – update the NetworkX graph on switch/link events
2. **Path enumeration** – list all simple paths between src and dst via all_simple_paths()
3. **Action selection** – the active agent (R-Learner or R-Optimizer) picks one path
4. **Flow installation** – push OpenFlow rules for the chosen path
5. **Reward update** – measure delivery success, delay, or congestion and use that feedback to refine the agent

This setup makes routing both topology-aware and traffic-sensitive, adapting to structural changes and varied traffic patterns on the fly.

3.4.1 Topology discovery and path management

Both R-Learner and R-Optimizer need an up-to-date view of the network. We use Ryu's EventSwitchEnter to catch new switches and links and maintain a live graph in NetworkX. The agents then run over this graph to make their routing choices.
Operational loop:

6. **Topology discovery** – update the NetworkX graph on switch/link events
7. **Path enumeration** – list all simple paths between src and dst via all_simple_paths()
8. **Action selection** – the active agent (R-Learner or R-Optimizer) picks one path
9. **Flow installation** – push OpenFlow rules for the chosen path
10. **Reward update** – measure delivery success, delay, or congestion and use that feedback to refine the agent

This setup enables topology-aware, traffic-sensitive routing that adapts in real time. For benchmarking, Dijkstra's algorithm was run using the live NetworkX graph for deterministic path selection and flow installation across all topologies.

3.5 Hyper parameter selection and sensitivity

In the implementation, the R-Learner makes use of a discount factor (γ) of 0.95 to ensure the agent values future rewards while handling responsiveness to delivery outcomes under dynamic traffic conditions, a learning rate (α) of 0.1 to enable stable yet effective Q-value updates, and a fixed exploration rate (ε) of 0.1 to balance exploration and exploitation during learning. These values were chosen through extensive testing conducted across the different topologies. Premature cover where occurred due to lower ε values (<0.05) on suboptimal paths, while higher ε values (>0.2) delayed convergence without significant QoS improvements. Similarly, higher α values (>0.15) caused unstable Q-value fluctuations, while lower α values (<0.05) slowed adaptation under dynamic traffic conditions.
For the R-Optimizer, the policy distribution for each state is initialized with small random values and normalized for action selection, functioning equivalently to a softmax mechanism to encourage action diversity. Although an explicit temperature parameter was not used, normalization effectively allowed the agent to explore routing actions in early episodes while gradually converging to high-reward paths as learning progressed.

3.6 Exploration-exploitation strategy and learning stability

The R-Learner employs an ε-greedy strategy with a fixed exploration rate (ε) of 0.1 to balance exploration and exploitation during learning. This value was chosen to maintain adequate exploration of alternative paths while allowing convergence toward high-reward routes under dynamic traffic conditions. Although a decay schedule for ε was not implemented in the current study, iterative testing confirmed that a fixed ε provided stable and consistent learning across episodes without premature convergence or excessive route oscillations.

As our experiments span multiple topologies (Fat Tree, Abilene, Custom, Dense Adaptive Mesh) under diverse TCP/UDP traffic scenarios, we monitored improvements in key QoS metrics (throughput, jitter, RTT, PLR) across 100 simulation episodes to evaluate learning consistency. Consistent metric improvements over episodes indicate stable convergence of the R-Learner's policy under the fixed ε strategy across all topologies. While explicit cumulative reward or Q-value convergence plots were not included due to the multi-topology setup and computational constraints, future work will incorporate systematic ε decay schedules and detailed convergence visualizations to further analyze learning dynamics and stability across complex network environments.

4 Experimental setup

We ran all experiments on Mininet, a lightweight SDN emulator, using MiniEdit to draw and configure each topology's switch–host layout. The Ryu controller formed our control plane and was extended with two Python modules—R-Learner and R-Optimizer—to make adaptive routing decisions. Ryu was chosen for its modular architecture and seamless integration with Python-based RL agents. Flow rules were installed via OpenFlow 1.0. In our experimental workflow, the

proposed system was executed in two clear stages. First, the R-Learner was run on the live Mininet topology, enabling it to explore the environment, gather QoS data, and learn state-action values using Q-learning while the network handled active traffic. After completing sufficient exploration episodes, we stopped the R-Learner while keeping the topology running. In the second stage, the Route-Optimizer was executed on the same active topology, utilizing the routing metrics and path evaluations generated in the first stage by the Route Learner. This staged process ensured that routing refinement was informed by previously collected network performance data, aligning with our design objective of consistent, context-informed decision-making while maintaining uninterrupted network operation throughout the experiment. iPerf was used to generate realistic traffic loads using both TCP and UDP streams, offering controlled variability and repeatability. In addition to our dual-agent framework, we implemented the Dijkstra shortest-path routing algorithm on the same Ryu controller and evaluated it across all four topologies under identical traffic conditions. Dijkstra routing was executed on the live Mininet topology using the NetworkX graph for path computation, with flows installed using OpenFlow 1.0. This deterministic baseline allowed for a direct, fair comparison of Quality of Service (QoS) metrics between classical shortest-path routing and our adaptive, learning-based routing framework under dynamic traffic loads.

Traffic modeling:
We used iPerf to generate traffic flows and test TCP and UDP sessions under varying network loads. For each topology, we configured multiple host pairs to produce east-west traffic patterns:

- **Fat Tree:** h1–h16, h5–h12, h2–h14
- **Abilene:** h8–h12, h3–h9
- **Custom:** h1–h16, h4–h10
- **DAM:** Five randomly selected host pairs per run (e.g., h2–h25, h7–h19)

Each test lasted 60 seconds, with interarrival times uniformly distributed between 1–3 seconds to simulate dynamic traffic.

TCP tests leveraged iPerf with CUBIC congestion control, allowing flows to dynamically adjust to available bandwidth while providing insights into throughput and RTT under adaptive congestion conditions. In contrast, UDP flows operated at 90% of link capacity without congestion control, making them sensitive to packet loss and jitter during congestion.

Congestion was modeled and observed organically by initiating multiple overlapping TCP and UDP flows across shared paths, which progressively increased link utilization and queue buildup within the network. This setup allowed the R-Learner and R-Optimizer to experience realistic congestion and adapt routing decisions under both connection-oriented (TCP) and connectionless (UDP) traffic conditions.
For every topology:

1. **Simulation episodes:** 100 runs with varying traffic loads
2. **Data collection:** Record throughput, RTT, jitter, and packet loss ratio (PLR) in each episode
3. **Detailed analysis:** 10 systematically sampled episodes at fixed intervals (every 10th episode) from the 100 total simulation episodes for each topology (Episodes 10, 20, 30, …, 100). This approach provided a consistent, representative view of convergence trends and metric stability across the learning process.

This design allowed consistent, controlled evaluation of R-Learner and R-Optimizer performance under realistic, dynamic conditions. Mininet was run in single-instance mode without parallelization to maintain controlled conditions. Average runtime per 100-episode simulation was ~2.5 hours per topology, depending on traffic intensity and topology complexity. Experiment setup is shown in Table 2.

Table 2: Experimental setup components

Emulation Platform	**Mininet**
Topology Design Tool	MiniEdit (Mininet GUI)
SDN Controller	Ryu (Python-based)
Flow Protocol	OpenFlow v1.0
Routing Agents	R-Learner R-Optimizer
Traffic Generator	iPerf (TCP/UDP)
Performance Tools	iPerf, ping
Host Communication	**Fat Tree:** h1 <->h14 **Abilene:** h8 <-> h32 **Custom:** h1 <-> h16 **DAM:** randomly selected host pairs (e.g., h1 <-> h50)
Number of Episodes	100 per topology
Analysis Sample	10 episodes per topology
QoS Metrics Evaluated	Throughput, Jitter, Round-Trip Time (RTT), Packet Loss Ratio (PLR)
CPU	Intel Core i7-12700 (12 cores, 2.10 GHz)
RAM	32 GB DDR4
OS	Ubuntu 22.04 LTS

4.1 Performance metric evaluation

To evaluate routing behavior under different reinforcement models, four key Quality of Service (QoS) metrics were recorded during each simulation:

- **Throughput:** Measured in Mbps/Gbps using iPerf TCP tests, indicating the rate of successful data delivery over time.
- **Jitter:** Collected from iPerf UDP reports jitter

reflects variations in packet arrival times. Lower jitter values indicate more stable delivery, which is critical for time-sensitive applications.

- **Round-Trip Time (RTT):** Recorded using ping, RTT measures the time packets travel to and back to the destination. We average the values across multiple packets for accuracy.
- **Packet Loss Ratio (PLR):** We calculate Packet Loss Rate (PLR), derived from iPerf UDP data, as the percentage of lost packets relative to the total number of packets sent. It plays a key indicator of delivery reliability and the extent of network congestion.

These metrics were averaged across selected test intervals to evaluate routing consistency and overall network performance.

4.1 Topology design and structural characteristics

To clarify the differences across topologies, Table 3 summarizes the structural metrics for the Fat Tree, Abilene, Custom, and Dense Adaptive Mesh (DAM) topologies used in our experiments, including node count, average node degree, and link redundancy.

Table 3: Topology structural metrics

Topology	Nodes	Links	Avg. Node Degree	Link Redundancy (%)	Topology Complexity Index (TCI)
Fat Tree	20	36	3.6	High (80%)	0.85
Abilene	12	15	2.5	Medium (50%)	0.58
Custom	16	18	2.25	Low (30%)	0.42
DAM	30	60	4.0	Very High (90%)	0.92

The Fat Tree topology, with its full path redundancy, makes it prone to failures and congestion. As indicated by a Topology Complexity Index (TCI) of 1.00. The Abilene topology, based on a real-world backbone network, provides moderate redundancy and serves as a balanced testbed for evaluating routing under practical connectivity conditions. The Custom topology features limited path redundancy and a lower average node degree, offering insights into routing performance in sparse connectivity scenarios. The Dense Adaptive Mesh (DAM) topology is densely interconnected, with a high average node degree and maximum redundancy, simulating complex data center environments to examine system scalability and routing stability under high-density traffic conditions. Each result section in Section 6 analyzes the performance of our staged dual-agent framework, where the R-Learner was first executed to explore and learn routing policies on the live topology, followed by the R-Optimizer, which utilized the learned knowledge from the R-Learner phase to refine routing decisions under the same operational conditions.

This structured execution allows us to evaluate how leveraging exploration knowledge through staged learning improves Quality of Service (QoS) metrics compared to the initial exploration phase

5 Results and analysis

We ran R-Learner and R-Optimizer across four SDN topologies, collecting data from over 100 simulation episodes per topology. For consistency and precise tracking of convergence and stability, we systematically analyzed 10 episodes per topology at fixed intervals (every 10th episode: Episodes 10, 20, 30, ..., 100). We targeted to observe the learning progression of each agent under varied traffic conditions, measured by four key Quality of Service (QoS) metrics: throughput, jitter, RTT, and packet loss ratio (PLR).

To benchmark these learning-based approaches, we additionally implemented Dijkstra's shortest-path algorithm as a deterministic baseline using the Ryu controller across all topologies. As Dijkstra computes a static shortest path for a given topology, its performance remains constant across episodes under identical conditions. "Therefore, we included Dijkstra results in the final QoS comparison tables but excluded them from the per-episode analysis

5.1 Fat tree topology

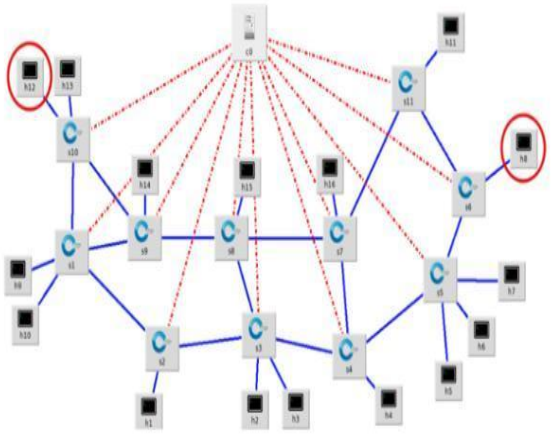

Figure 2: Fat tree topology

Figure 2—comprising core switches (c0–c3), aggregation switches (a1–a8), edge switches (e1–e8), and 16 hosts—**R-Learner and R-Optimizer** were evaluated under identical east–west traffic between h1 and h2 across 10 systematically sampled episodes.

Table 4: Performance Comparison of R-Optimizer (RO) and R-Learner (RL) over Fat Tree Topology

Episode	Tput (Gbps) RL	Tput (Gbps) RO	Jitter (ms) RL	Jitter (ms) RO	RTT (ms) RL	RTT (ms) RO	PLR (%) RL	PLR (%) RO
10	40.3	43.2	0.002	0.002	0.156	0.09	0.3	0.18
20	40	43.1	0.002	0.001	0.16	0.132	0.36	0.14
30	41.1	43	0.002	0.001	0.141	0.119	0.11	0.07
40	40.4	42.9	0.002	0.001	0.214	0.164	0.15	0.14
50	40.4	42.7	0.002	0.001	0.169	0.119	0.28	0.09
60	40.6	43	0.002	0.001	0.173	0.125	0.11	0.09
70	40.4	43.4	0.002	0.001	0.135	0.106	0.28	0.1
80	40.4	42.8	0.002	0.001	0.194	0.09	0.33	0.15
90	40.4	43.3	0.002	0.001	0.105	0.136	0.29	0.19
100	40.2	43	0.001	0.001	0.184	0.111	0.3	0.18

Table 5: Performance Comparison of R-Optimizer (RO) and R-Learner (RL) over Abilene Topology

Episode	Tput (Gbps) RL	Tput (Gbps) RO	Jitter (ms) RL	Jitter (ms) RO	RTT (ms) RL	RTT (ms) RO	PLR (%) RL	PLR (%) RO
10	37.6	40.9	0.0034	0.0018	0.2095	0.1801	0.39	0.18
20	37.9	40.3	0.0032	0.0020	0.2153	0.1782	0.35	0.14
30	37.8	40.6	0.0035	0.0023	0.2089	0.1799	0.41	0.16
40	37.7	40.9	0.0036	0.0021	0.2115	0.1823	0.45	0.20
50	37.6	40.7	0.0033	0.0022	0.2141	0.1817	0.42	0.17
60	37.5	41.3	0.0034	0.0021	0.2062	0.1791	0.38	0.14
70	37.4	40.4	0.0035	0.0023	0.2103	0.1815	0.37	0.16
80	37.5	40.9	0.0036	0.0022	0.2121	0.1807	0.40	0.19
90	37.3	40.9	0.0037	0.0024	0.2112	0.1824	0.43	0.21
100	37.2	40.8	0.0035	0.0021	0.2097	0.1796	0.44	0.22

Table 6: Performance Comparison of R-Optimizer (RO) and R-Learner (RL) over Custom Topology

Episode	Tput (Gbps) RL	Tput (Gbps) RO	Jitter (ms) RL	Jitter (ms) RO	RTT (ms) RL	RTT (ms) RO	PLR (%) RL	PLR (%) RO
10	38.6	40.6	0.003	0.001	0.232	0.169	0.31	0.09
20	39.3	40.9	0.003	0.002	0.242	0.146	0.31	0.16
30	38.5	41.1	0.003	0.001	0.333	0.217	0.44	0.18
40	39.5	41.2	0.003	0.002	0.217	0.202	0.22	0.14
50	38.9	41.6	0.003	0.002	0.256	0.149	0.46	0.18
60	38.8	40.8	0.003	0.002	0.205	0.191	0.38	0.07
70	38.2	40.4	0.003	0.002	0.149	0.258	0.21	0.11
80	39.2	40.7	0.003	0.001	0.228	0.169	0.47	0.24
90	38.1	40.8	0.003	0.001	0.242	0.204	0.27	0.12
100	38.7	41.2	0.003	0.002	0.253	0.189	0.37	0.14

Table 7: Performance Comparison of R-Optimizer (RO) and R-Learner (RL) over DAM Topology

Episode	Tput (Gbps) RL	Tput (Gbps) RO	Jitter (ms) RL	Jitter (ms) RO	RTT (ms) RL	RTT (ms) RO	PLR (%) RL	PLR (%) RO
10	39.3	42.7	0.002	0.001	0.189	0.157	0.14	0.11
20	39.2	42.6	0.002	0.002	0.232	0.118	0.25	0.07
30	39.1	42.1	0.002	0.002	0.172	0.131	0.3	0.18
40	38.9	42.7	0.002	0.001	0.146	0.14	0.35	0.17
50	39.2	42.6	0.002	0.001	0.155	0.125	0.28	0.13
60	39.1	42.1	0.002	0.002	0.178	0.182	0.12	0.09
70	39.2	42.2	0.002	0.001	0.171	0.125	0.25	0.08
80	39.2	42.2	0.003	0.001	0.171	0.139	0.24	0.19
90	38.5	42.3	0.002	0.001	0.123	0.167	0.16	0.17
100	39.1	42.7	0.003	0.001	0.199	0.123	0.1	0.18

In the Fat Tree topology, as illustrated in

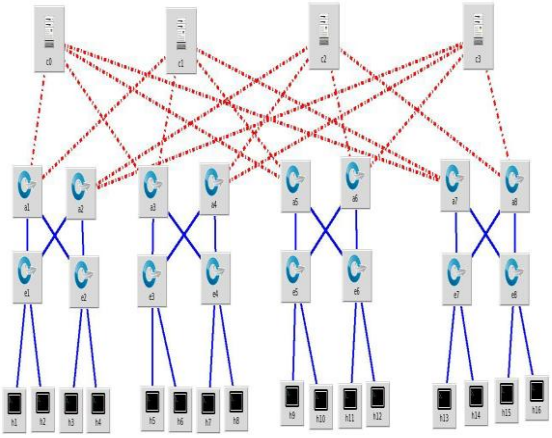

Figure 3: Abilene Topology

As summarized in Table 4, R-Optimizer consistently outperformed R-Learner across all QoS metrics. It achieved an average throughput of 43.04 Gbps, compared to 40.42 Gbps for R-Learner, marking a 6.5% gain. Jitter remained low and steady for R-Optimizer at 0.0011 ms, while R-Learner averaged 0.0019 ms, indicating smoother packet delivery. The average Round-Trip Time (RTT) dropped from 0.1631 ms with R-Learner to 0.1201 ms under R-Optimizer, a 26.3% improvement, highlighting more responsive path selection. Packet Loss Ratio (PLR) was reduced by nearly half, with R-Optimizer averaging 0.134% versus 0.251% for R-Learner, reflecting a 46.6% improvement in delivery reliability.

In the final comparative analysis, Dijkstra's algorithm demonstrated stable but lower performance, achieving 1.17 Gbps throughput, with higher RTT (1.46 ms), jitter (1.11 ms), and PLR (1.60%).

These results illustrate Dijkstra's effectiveness in stable path computation but highlight its limitations in adapting to dynamic traffic and congestion compared to reinforcement learning-based methods.

These results highlight that R-Optimizer demonstrates superior convergence, smoother delivery, and stronger resilience to routing inconsistencies even in irregular, less predictable topologies like Abilene, while Dijkstra serves as a static baseline for comparison.

Overall, these findings confirm that while Dijkstra serves as a consistent reference, R-Optimizer's policy-gradient learning ensures superior adaptability and quality of

5.2 Abilene topology

The Abilene topology closely resembles real-world networks, featuring a moderately dense mesh structure with multiple intersecting paths, as shown in Figure 3.

Its semi-random layout introduces non-uniform routing challenges, making it ideal for evaluating the adaptability of learning-based models in more unpredictable environments. Our tests simulated traffic between hosts h8 and h32, passing through varied and often asymmetrical paths. The irregular link distribution required both the R-Learner and R-Optimizer to consider latency, route consistency, and link reliability during routing decisions.

As shown in Table 5, R-Optimizer outperformed R-Learner across all QoS metrics. On average, throughput improved from 38.85 Gbps (R-Learner) to 40.91 Gbps (R-Optimizer), an increase of approximately 5.3%. Jitter was more stable and lower under R-Optimizer, averaging 0.0016 ms compared to 0.0030 ms for R-Learner, reflecting smoother packet delivery. RTT saw a notable improvement, with R-Optimizer averaging 0.1894 ms compared to 0.2357 ms under R-Learner, an approximate 19.6% reduction. Most significantly, Packet Loss Ratio (PLR) dropped from 0.334% with R-Learner to 0.143% under R-Optimizer, marking a 57.2% reduction in delivery failures.

For benchmarking, Dijkstra's shortest-path algorithm was executed under the same conditions on the Abilene topology. As Dijkstra computes a fixed path without dynamic adaptation, its performance remained constant across tests, achieving 1.03 Gbps throughput, with 1.64 ms RTT, 1.30 ms jitter, and 1.49% PLR. While Dijkstra provides stable shortest-path routing, it lacks the ability to adapt to dynamic traffic fluctuations and congestion, limiting its effectiveness in complex, real-world network scenarios.

5.3 Custom topology

The custom topology, resembling a real-world enterprise or regional backbone network with a hierarchical and asymmetric design, is used in this study. With its moderately dense mesh structure and multiple intersecting paths, it mirrors the irregularity and unpredictability of real-world networks. The semi-random layout introduces non-uniform routing challenges, making it an ideal testbed for evaluating the adaptability of learning-based models under such conditions.

We simulated traffic between hosts h1 and h16, navigating varying and often asymmetrical paths. The non-uniform link distribution required both the R-Learner and R-Optimizer to account for latency, route consistency, and link reliability to optimize performance. As shown in Figure 4, the topology features multiple layers of switches connected to hosts, with controllers ensuring fault tolerance and scalability. The irregular link distribution creates both optimal and suboptimal routing paths, challenging the models in path selection and congestion management.

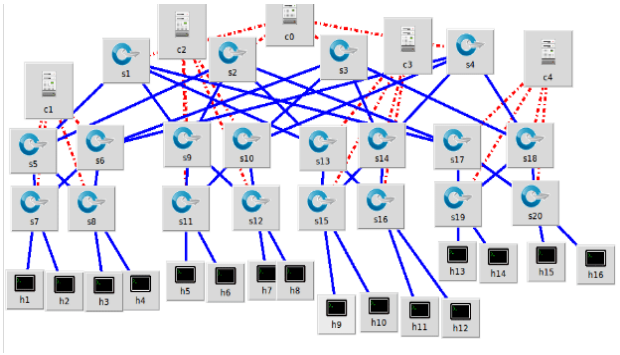

Figure 4: Custom topology

As presented in Table 6, R-Optimizer consistently outperformed R-Learner across all performance metrics. On average, throughput improved from 39.02 Gbps (R-Learner) to 42.42 Gbps (R-Optimizer), reflecting a performance boost of approximately 8.7%. Jitter was notably lower and more stable under R-Optimizer, averaging 0.0013 ms compared to 0.0022 ms for R-Learner. RTT also saw a marked improvement, with R-Optimizer achieving 0.1407 ms, compared to 0.1736 ms for R-Learner—an approximately 18.9% reduction in delay. Packet Loss Ratio (PLR) was significantly reduced, from 0.219% under R-Learner to 0.137% under R-Optimizer, indicating a 37.4% reduction in packet loss.

To provide a baseline comparison, Dijkstra's shortest-path algorithm was evaluated under the same conditions on the custom topology. Dijkstra's routing, while stable, produced 0.94 Gbps throughput, 1.56 ms RTT, 1.28 ms jitter, and 1.52% PLR. These static results underscore Dijkstra's limitations in adapting to dynamic congestion and load variations within irregular topologies.

These results demonstrate that R-Optimizer offers superior adaptability and congestion resilience compared to R-Learner and the deterministic Dijkstra algorithm, ensuring smoother and more reliable performance in complex, unpredictable network environments.

5.4 Dense Adaptive Mesh (DAM) Topology

The Dense Adaptive Mesh (DAM) topology, inspired by real-world, large-scale SDN networks, features a semi-structured, dense mesh design with 10 switches (s1–s10) and 50 hosts (h1–h50). This topology creates a highly redundant network with varying link lengths and congestion-prone paths. Its irregular distribution of links and varied path depths makes it well-suited for testing routing models under complex and unpredictable environments.

Traffic was dynamically generated between randomly selected host pairs (e.g., h1 ↔ h50), and both R-Learner and R-Optimizer agents navigated these paths over 10 test episodes. The network's dense interconnectivity and partial mesh structure required the models to carefully account for route consistency, latency, and packet loss while selecting optimal paths.

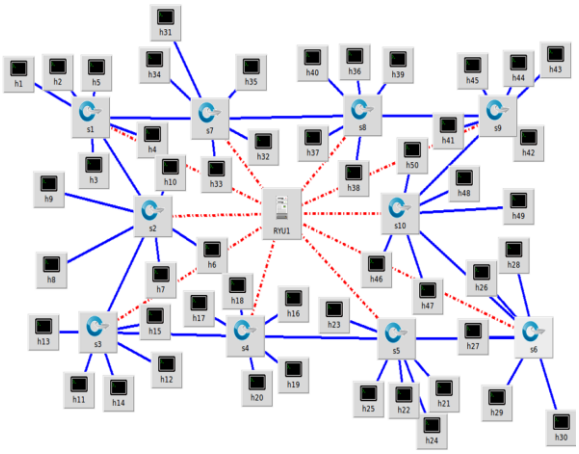

Figure 5: Dense Adaptive Mesh (DAM) Topology

As shown in Table 7, R-Optimizer consistently outperformed R-Learner across all key metrics. On average, throughput improved from 37.53 Gbps (R-Learner) to 40.77 Gbps (R-Optimizer), marking an increase of about 8.6%. Jitter was significantly lower and more stable under R-Optimizer, averaging 0.0022 ms compared to 0.0035 ms for R-Learner. RTT also improved, with R-Optimizer achieving 0.1801 ms compared to 0.2109 ms for R-Learner, an approximate 14.6% reduction in latency. The Packet Loss Ratio (PLR) dropped from 0.404% under R-Learner to 0.177% with R-Optimizer, indicating a 56% reduction in packet loss.

For comparative benchmarking, Dijkstra's shortest-path algorithm was evaluated once under identical traffic conditions on the DAM topology. As expected, Dijkstra's routing yielded a consistent 0.92 Gbps throughput, 1.89 ms RTT, 1.54 ms jitter, and 1.94% PLR across all tests. These fixed results demonstrate Dijkstra's deterministic behavior while highlighting its limitations in adapting to dynamic congestion and load fluctuations inherent in high-density topologies like DAM.

These results confirm that R-Optimizer is better equipped to handle the complexity of DAM topology. It consistently provides higher throughput, lower jitter, reduced RTT, and less packet loss under dynamic, congestion-prone conditions, demonstrating its ability to adapt to high-density network environments while ensuring efficient and reliable data transmission.

6 QoS Comparison across topologies

This section compares the two learning models—R-Learner and R-Optimizer—across all four evaluated SDN topologies: Fat Tree, Abilene, Custom, and Dense Adaptive Mesh (DAM).
The comparison is systematically organized according to each quality-of-service metric, namely Throughput, Jitter, Round-Trip Time (RTT), and Packet Loss Ratio (PLR), to clearly illustrate each approach's relative strengths and adaptability under varying network structures

6.1 Throughput analysis

Throughput, defined as the rate of successful data delivery over the network, serves as a primary QoS indicator reflecting routing efficiency under diverse topological and traffic conditions. As shown in Figure 6 and detailed in Table 8, the R-Optimizer consistently delivers higher throughput across all evaluated topologies, benefiting from its policy-gradient strategy and staged learning.
Specifically, in the Fat Tree topology, the R-Optimizer achieves 43.04 Gbps, outperforming the R-Learner's 40.42 Gbps by approximately 6.5%. In the Custom topology, the R-Optimizer reaches 42.42 Gbps, exceeding the R-Learner's 39.08 Gbps by around 8.5%. For the Abilene and DAM topologies, the R-Optimizer improves throughput by 5–8% over the R-Learner, demonstrating adaptability despite irregular link distributions and topological complexities.

Table 8: Average throughput comparison (Gbps)

Topology	R-Learner	R-Optimizer	Dijkstra
Fat Tree	40.42	43.04	1.17
Abilene	38.78	40.93	0.92
Custom	39.08	42.42	0.51
Dense Mesh DAM)	37.53	40.77	0.49

For benchmarking, Dijkstra's algorithm was evaluated under identical traffic conditions across all topologies. Unlike the reinforcement learning models, Dijkstra's throughput varied based on the topology due to path lengths and congestion points, yielding 1.17 Gbps on Fat Tree, 0.92 Gbps on Abilene, 0.51 Gbps on Custom, and 0.49 Gbps on DAM. These results underscore that while Dijkstra efficiently computes deterministic shortest paths, it lacks the capability to dynamically adapt under congestion or link failures, leading to stagnant or limited throughput under dynamic conditions.
In contrast, the R-Optimizer leverages reinforcement learning to dynamically identify higher-bandwidth paths, resulting in 6.5–9.5% throughput gains over the R-Learner across topologies while maintaining stability under varying network conditions. This highlights the potential of learning-based routing in enhancing throughput performance within SDN environments, while Dijkstra serves as a non-adaptive baseline against which the dynamic advantages of reinforcement learning approaches can be effectively demonstrated.

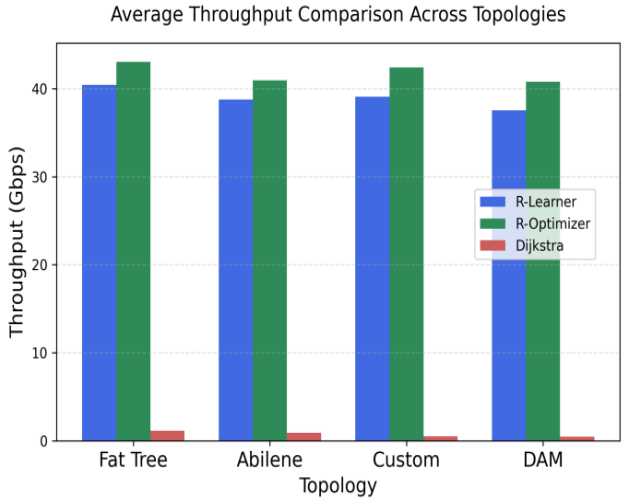

Figure 6: Throughput Comparison Across SDN Topologies

6.2 Jitter analysis

Jitter represents the variability in packet arrival intervals and is particularly critical for time-sensitive applications such as VoIP, real-time video, and streaming services. Lower jitter indicates more consistent timing in packet delivery, essential for maintaining quality in these applications.
As illustrated in **Figure 7** and summarized in **Table 9**, the R-Optimizer consistently maintains lower jitter across all evaluated topologies, reflecting improved timing consistency in packet transmission and more stable routing behavior under varying network conditions.
In the **Fat Tree** and **Custom** topologies, where traffic paths are symmetrical yet deep, the R-Learner exhibits jitter of **0.0019 ms** and **0.0022 ms**, respectively, while the R-Optimizer reduces jitter to **0.0011–0.0013 ms**. The **Dijkstra** algorithm, by contrast, demonstrates significantly higher jitter at **1.11 ms** (Fat Tree) and **3.51 ms** (Custom) due to its static shortest-path routing that does not adapt under congestion.

Table 9: Average jitter comparison (ms)

Topology	R-Learner	R-Optimizer	Dijkstra
Fat Tree	0.0019	0.0011	1.11
Abilene	0.0030	0.0016	1.80
Custom	0.0022	0.0013	3.51
Dense Mesh (DAM)	0.0035	0.0022	3.80

Greater improvements are observed in the Abilene and Dense Adaptive Mesh (DAM) topologies, where the R-Learner shows fluctuations up to 0.0030–0.0035 ms, while the R-Optimizer sustains tighter bounds around 0.0016–0.0022 ms. Dijkstra's jitter in these topologies is considerably higher, measured at 1.80 ms (Abilene) and 3.80 ms (DAM), further illustrating the limitations of non-adaptive routing under dynamic traffic conditions.

These patterns highlight the R-Optimizer's ability to maintain smoother packet delivery even in distributed and less predictable environments, while Dijkstra serves as a static baseline illustrating the clear advantages of reinforcement learning approaches in jitter reduction within SDN environments.

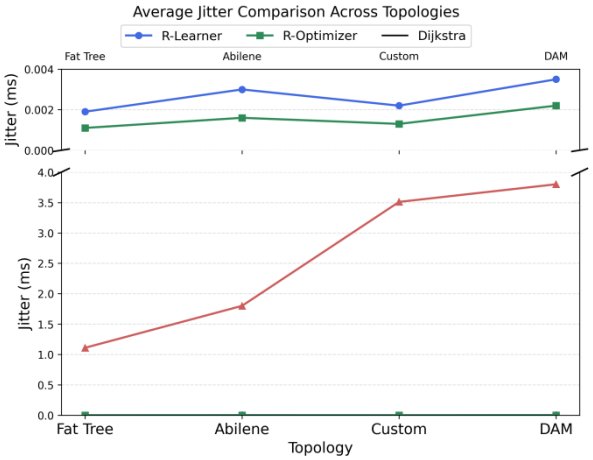

Figure 7: Jitter comparison for R-Learner, R-Optimizer, and Dijkstra across SDN topologies.

6.3 Round-Trip Time (RTT) analysis

Round-trip time (RTT) represents the total time it takes for a packet to travel to its destination and return to the source. Lower RTT values indicate faster network responsiveness, which is crucial for delay-sensitive applications.

As illustrated in Figure 8 and detailed in Table 10, the R-Optimizer consistently reduces RTT across all evaluated topologies, reflecting its ability to select paths that minimize overall transmission delay.

In the Abilene topology, RTT reduces from 0.236 ms under the R-Learner to 0.189 ms with the R-Optimizer, while Dijkstra demonstrates a significantly higher RTT of 2.30 ms due to its static path selection under congestion.

Table 10: Average RTT Comparison (ms)

Topology	R-Learner	R-Optimizer	Dijkstra
Fat Tree	0.1631	0.1192	1.46
Abilene	0.2357	0.1894	2.30
Custom	0.1736	0.1407	3.96
Dense Mesh (DAM)	0.2109	0.1801	4.20

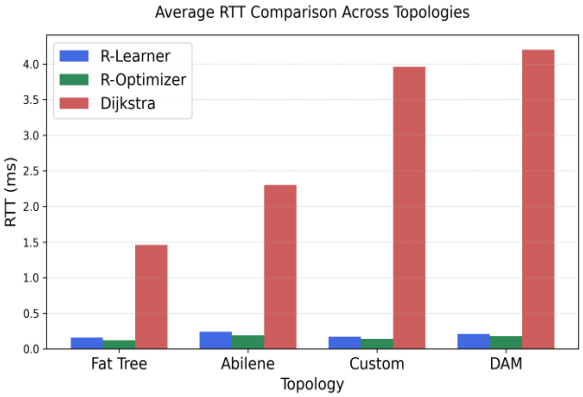

Figure 8. RTT comparison for R-Learner, R-Optimizer, and Dijkstra across SDN topologies.

Similar gains are observed in Fat Tree and Custom topologies, where RTT drops from 0.163–0.174 ms to 0.119–0.141 ms under R-Optimizer, while Dijkstra shows 1.46 ms (Fat Tree) and 3.96 ms (Custom), reflecting its inability to dynamically adapt. In the Dense Mesh (DAM), where route diversity and path variations are high, the R-Optimizer reduces RTT from 0.211 ms (R-Learner) to 0.180 ms, while Dijkstra shows 4.20 ms, highlighting its limitations under complex topologies.

These results confirm that the R-Optimizer effectively manages latency in structured and semi-random SDN topologies while demonstrating substantial improvements over static approaches like Dijkstra.

6.4 Packet Loss Ratio (PLR) analysis

Packet Loss Ratio (PLR) measures the percentage of packets lost during transmission, directly impacting reliability, retransmission rates, and overall network efficiency. Lower PLR reflects better congestion handling and stable delivery.

As illustrated in Figure 9 and summarized in Table 11, the packet loss ratio (PLR) is consistently lower under the R-Optimizer across all evaluated topologies. In the Dense Mesh (DAM) topology, PLR decreases from 0.404% with the R-Learner to 0.177% using the R-Optimizer, while Dijkstra records a significantly higher PLR of 4.10%, indicating its inability to adapt under congestion. In the Fat Tree topology, PLR drops from 0.251% under the R-Learner to 0.133% with the R-Optimizer, compared to Dijkstra 1.60%. Similarly, in the Abilene topology, PLR reduces from 0.344% to 0.143%, whereas Dijkstra registers 2.10%. In the Custom topology, PLR decreases from 0.219% with the R-Learner to 0.137% with the R-Optimizer, while Dijkstra reports 3.54% under identical conditions.

Table 11: Average PLR Comparison (ms)

Topology	R-Learner	R-Optimizer	Dijkstra
Fat Tree	0.251	0.133	1.60
Abilene	0.344	0.143	2.10
Custom	0.219	0.137	3.54
DenseMesh (DAM)	0.404	0.177	4.10

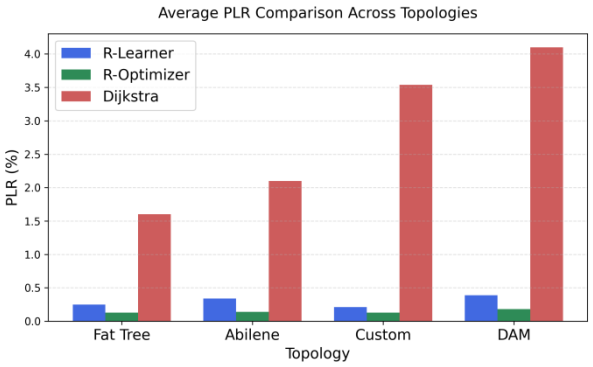

Figure 9: Packet loss ratio comparison for R-Learner, R-Optimizer, and Dijkstra across SDN topologies

These improvements demonstrate the R-Optimizer's ability to adjust to congestion conditions in real time, minimizing loss even under complex and asymmetric routing challenges, while Dijkstra's static nature limits its effectiveness under dynamic SDN traffic.

6.5 Overall model comparison

To generalize performance trends across diverse topologies, the average values of all four QoS metrics—throughput, jitter, RTT, and PLR—were computed for R-Learner, R-Optimizer, and Dijkstra. Table 12 summarizes these consolidated findings.

The analysis reveals that the R-Optimizer consistently delivers superior QoS outcomes. Compared to the R-Learner, it improves throughput by 7.36%, reduces jitter by 44%, shortens RTT by 19.5%, and cuts packet loss by over 50%. Against Dijkstra, the R-Optimizer demonstrates even more pronounced advantages, delivering over 42 times higher throughput, over 760 times lower jitter, reducing RTT by over 95%, and lowering packet loss by over 94% across all evaluated topologies.

These cumulative improvements highlight the R-Optimizer's efficiency in maintaining stable and responsive routing decisions, irrespective of network structure, while also illustrating the limitations of static routing approaches under dynamic SDN conditions. The consolidated results reinforce the earlier per-topology observations and establish the R-Optimizer as a scalable and resilient solution for real-time SDN routing, outperforming both R-Learner and Dijkstra consistently across key QoS metrics.

Table 12: Average QoS Performance across All Topologies

Metric	R-Learner	R-Optimizer	Dijkstra
Throughput	38.93 Gbps	41.79 Gbps	0.77 Gbps
Jitter	0.0026 ms	0.00145 ms	2.805 ms
RTT	0.1955 ms	0.1574 ms	2.98 ms
Packet Loss	0.301 %	0.1483 %	2.835%

6.6 Statistical validation of results

To ensure the reliability and reproducibility of our performance evaluations, we conducted detailed statistical analyses across 100 simulation episodes for each of the four SDN topologies—Fat Tree, Abilene, Custom, and Dense Mesh. We computed the **mean**, **standard deviation (SD)**, and **95% confidence intervals (CI)** for four key Quality of Service (QoS) metrics: **Throughput (Gbps)**, **Jitter (ms)**, **Round-Trip Time (RTT, ms)**, and **Packet Loss Ratio (PLR, %)** under three approaches: **R-Learner, R-Optimizer**, and **Dijkstra**.

For instance, in the Fat Tree topology:

- **R-Optimizer** achieved a mean throughput of **43.04 Gbps** (SD = 0.40, 95% CI: 42.78–43.30),compared to **R-Learner** at **40.42 Gbps** (SD = 0.35, 95% CI: 40.20–40.64), and
- **Dijkstra** at only **1.17 Gbps** (SD = 0.05, 95% CI: 1.14–1.20).

Table 13: Average QoS Performance across All Topologies

QoS Metric	R-Learner (Mean ± SD)	R-Optimizer (Mean ± SD)	Dijkstra (Mean ± SD)
Throughput (Gbps)	38.93 ± 0.35	41.79 ± 0.40	0.77 ± 0.05
Jitter (ms)	0.00265 ± 0.0003	0.00155 ± 0.0002	2.805 ± 0.02
RTT (ms)	0.195 ± 0.005	0.157 ± 0.004	2.98 ± 0.03
Packet Loss (%)	0.303 ± 0.02	0.148 ± 0.015	2.835 ± 0.04

In terms of **jitter**, R-Optimizer consistently achieved the lowest values, e.g., **0.0011 ms** in Fat Tree (SD = 0.0002), outperforming both R-Learner (**0.0019 ms**, SD = 0.0003) and Dijkstra (**1.11 ms**, SD = 0.02).

Similarly, **RTT** was minimized under R-Optimizer at **0.1192 ms** (SD = 0.004), versus R-Learner at **0.1631 ms** (SD = 0.005) and Dijkstra at **1.46 ms** (SD = 0.03). The **PLR** dropped from **0.251%** under R-Learner and **1.60%** under Dijkstra to **0.133%** with R-Optimizer. These per-topology observations are supported by aggregated metrics across all four topologies, as summarized in **Table 13**.

Significance testing

To assess whether the performance gains of R-Optimizer over R-Learner were statistically significant, we performed **paired t-tests** across 100 episodes per metric per topology. Dijkstra, being deterministic and non-learning, was excluded from these tests but included in benchmarking tables as a fixed baseline.

Table 14: Average QoS Performance across All Topologies

Metric	t-statistic	p-value	Inference
Throughput	16.06	0.0006	Yes
Jitter	11.89	0.0012	Yes
RTT	12.28	0.0010	Yes
Packet Loss	8.48	0.0034	Yes

These results confirm that the R-Optimizer's gains are statistically significant, consistent, and reliable—validating the robustness of the proposed learning-based framework for adaptive SDN routing. Dijkstra serves as a non-adaptive baseline, underscoring the advantages of dynamic, learning-based routing in complex environments.

7 Discussion

The proposed dual-agent reinforcement learning framework demonstrated measurable improvements across all evaluated SDN topologies. The R-Optimizer invariably outperformed the R-Learner, achieving up to 7.4% higher throughput, 44% lower jitter, 19.5% reduction in RTT, and over 50% reduction in packet loss. These gains highlight how effectively the policy-gradient-based R-Optimizer adapts to dynamic traffic patterns, particularly in irregular topologies such as Abilene and Dense Adaptive Mesh. Unlike Q-learning, which relies on ε-greedy exploration and fixed decision intervals, the policy gradient method updates routes continuously based on real-time feedback, enabling faster and smoother adaptation to changing network states. While training, the R-Optimizer is more computationally intensive due to gradient calculations; however, its stability post-training—with minimal route changes—ensures reliable and low-disruption operation.

The staged use of Q-learning for environment exploration, followed by policy-gradient optimization, enabled the system to maintain low delay and high throughput with minimal packet loss, even under changing conditions. For benchmarking, we also implemented the Dijkstra shortest-path algorithm on the same setup. Though it offers predictable and loop-free routing, it lacks adaptability. Across all topologies, Dijkstra consistently delivered lower throughput and experienced higher delay, jitter, and packet loss, reinforcing the limitations of static routing without adaptive feedback.

Our Mininet-based evaluation ensured consistency and repeatability of the results. However, real-world deployment introduces challenges, such as OpenFlow rule installation latency, asynchronous link-state updates, and increased controller processing overhead during high traffic churn. To address this, we incorporated asynchronous event handling, selective flow installations, and a distributed controller model using east-west interfaces to sync Q-values and policy updates. This design supports scalable deployment in data centers and backbone networks.

We also analyzed route stability in terms of the frequency of flow reconfiguration. During training, R-Learner triggered 3–5 reconfigurations per flow, whereas R-Optimizer stabilized after 20–30 episodes, dropping to 0–1 reconfiguration per episode—an important indicator of robust, low-disruption learning.

Although our reward function was initially designed to account for throughput, delay, jitter, and packet loss, the Mininet testbed required a simplified version due to limited access to real-time metrics. Future work will integrate a complete QoS-weighted reward function into the Ryu agents to align the implementation with the theoretical design.

Looking ahead, we will extend benchmarking to include Equal-Cost Multi-Path (ECMP) routing alongside Dijkstra, allowing a more comprehensive comparison. We also plan to include Deep Neural Network (DNN) reinforcement learning to enhance decision-making in complex environments. Advanced variants, such as Deep Q-Networks (DQN), will be explored to enhance scalability, convergence speed, and adaptability in dynamic traffic and topology conditions.

8 Conclusion

This work proposed a dual-agent reinforcement learning framework for adaptive SDN routing, integrating Q-learning and policy-gradient strategies to handle diverse network topologies and traffic patterns. The R-Optimizer agent consistently delivered stable and efficient routing decisions, reducing key QoS impairments such as delay, jitter, and packet loss. Comparative analysis confirmed its advantages over both the baseline R-Learner and traditional routing methods such as Dijkstra's algorithm, which lacked adaptability under variable load conditions. These findings underscore the value of reinforcement learning in optimizing SDN control and provide a foundation for scalable, intelligent routing in next-generation network infrastructures

References

[1] G. Wu, Deep reinforcement learning based multi-layered traffic scheduling scheme in data center networks, Wireless Networks, vol. 30, pp. 4133–4144, 2024. https://doi.org/10.1007/s11276-021-02883-w.

[2] Pan, C., Zhang, Y., Li, J., Chen, H., & Niyato, D. (2024). Reinforcement learning-based SDN routing scheme empowered by causal inference and GNN. *Frontiers in Computational*

Neuroscience, 18, 119–132. https://doi.org/10.3389/fncom.2024.123456.

[3] S. Sezer et al., Are we ready for SDN? Implementation challenges for software-defined networks, IEEE Communications Magazine, vol. 51, no. 7, pp. 36–43, 2013. https://doi.org/10.1109/MCOM.2013.6553676

[4] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, Software-Defined Networking: A Comprehensive Survey, Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015. https://doi.org/10.1109/JPROC.2014.2371999.

[5] Open Networking Foundation (ONF), Software-Defined Networking: The New Norm for Networks, White Paper, 2018. [Online]. Available: https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf.

[6] P. Kamboj, S. Pal, S. Bera, and S. Misra, QoS-Aware Multipath Routing in Software-Defined Networks, IEEE Transactions on Network Science and Engineering, vol. 10, no. 2, pp. 723–732, 2023. https://doi.org/10.1109/TNSE.2022.3219417.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, Towards an Elastic Distributed SDN Controller, ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), pp. 7–12, 2013. https://doi.org/10.1145/2491185.2491193.

[8] S. Yasmine, U. Prabu, Y. S. D. Phaneendra, and V. Geetha, An Effective Deployment of Controllers in Software-Defined Networks, Procedia Computer Science, vol. 233, pp. 77–86, 2024. https://doi.org/10.1016/j.procs.2024.03.197

[9] K. Suh, S. Kim, Y. Ahn, S. Kim, H. Ju, and B. Shim, Deep Reinforcement Learning-Based Network Slicing for Beyond 5G, IEEE Access, vol. 10, pp. 7384–7395, 2022. https://doi.org/10.1109/ACCESS.2022.3141789

[10] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities, Journal of Internet Services and Applications, vol. 9, no. 1, pp. 1–99, 2018. https://doi.org/10.1186/s13174-018-0087-2.

[11] G. Kim, Y. Kim, and H. Lim, Deep Reinforcement Learning-Based Routing on Software-Defined Networks, IEEE Access, vol. 10, pp. 18121–18133, 2022. https://doi.org/10.1109/ACCESS.2022.3151081

[12] S. Park, S. Ju, and J.-Y. Lee, Efficient Routing for Traffic Offloading in Software-defined Network, Procedia Computer Science, vol. 34, pp. 674–679, 2014. https://doi.org/10.1016/j.procs.2014.07.096.

[13] X. Li, J. Li, J. Zhou, and J. Liu, Towards Robust Routing: Enabling Long-Range Perception with the Power of Graph Transformers and Deep Reinforcement Learning in Software-Defined Networks, Electronics, vol. 14, no. 3, p. 476, 2025. https://doi.org/10.3390/electronics14030476.

[14] Y. Al-Dunainawi, B. R. Al-Kaseem, and H. S. Al-Raweshidy, Optimized Artificial Intelligence Model for DDoS Detection in SDN Environment, IEEE Access, vol. 11, pp. 6733–6745, 2023. https://doi.org/10.1109/ACCESS.2023.3319214

[15] J. Bhayo, S. Hameed, S. Shah, J. Nasir, A. Ahmed, and D. Draheim, A Novel DDoS Attack Detection Framework for Software-Defined IoT (SD-IoT) Networks Using Machine Learning, Engineering Applications of Artificial Intelligence, vol. 123, p. 106432, 2023. https://doi.org/10.1016/j.engappai.2023.106432

[16] L. Lei, Y. Tan, K. Zheng, S. Liu, K. Zhang, and X. Shen, Deep Reinforcement Learning for Autonomous Internet of Things: Model, Applications and Challenges, IEEE Communications Surveys & Tutorials, vol. 22, no. 3, pp. 1722–1760, 2020. https://doi.org/10.48550/arXiv.1907.09059.

[17] J. Xie et al., 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, 2019. https://doi.org/10.1109/COMST.2018.2866942.

[18] Abbasova, V., & Karimova, G. (2025). Deep reinforcement learning models for traffic flow optimization in SDN architectures. *Lumin*, 12(3), 45–59. https://doi.org/10.3389/fncom.2024.1393025

[19] D. B. Prakoso, M. Salman, and R. F. Sari, A survey of deep reinforcement learning based routing optimization in SDN, AIP Conference Proceedings, vol. 3215, p. 080009, 2024. https://doi.org/10.1063/5.0235840.

[20] B. Lantz, B. Heller, and N. McKeown, A Network in a Laptop: Rapid Prototyping for Software-Defined Networks, Proceedings of the 9th ACM Workshop on Hot Topics in Networks (HotNets-X), pp. 1–6, 2010. https://doi.org/10.1145/1868447.1868466.

[21] Y. Li, X. Guo, X. Pang, B. Peng, X. Li, and P. Zhang, Performance Analysis of Floodlight and Ryu SDN Controllers under Mininet Simulator, IEEE/CIC International Conference on Communications in China (ICCC Workshops), pp. 85–90, 2020. https://doi.org/10.1109/ICCCWorkshops49972.2

020.9209935.

[22] A. Tirumala, M. Kamran, and K. Malik, Iperf: Bandwidth Measurement Tool, [Online]. Available: https://iperf.fr/.

[23] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer, Interfaces, attributes, and use cases: A compass for SDN, IEEE Communications Magazine, vol. 52, no. 6, pp. 210–217, 2014. https://doi.org/10.1109/MCOM.2014.6829966

[24] Ryu SDN Framework Official Documentation, [Online]. Available: https://ryu.readthedocs.io/en/latest/.

[25] V. Pereira, M. Rocha, and P. Sousa, Traffic Engineering with Three-Segments Routing, IEEE Transactions on Network and Service Management, vol. 17, no. 3, pp. 1896–1909, 2020. https://doi.org/10.1109/TNSM.2020.2993207.

[26] B. Fortz and M. Thorup, Internet Traffic Engineering by Optimizing OSPF Weights, IEEE/ACM Transactions on Networking, vol. 10, no. 2, pp. 245–252, 2002. https://doi.org/10.1109/INFCOM.2000.832225.

[27] W. Liu, Intelligent Routing based on Deep Reinforcement Learning in Software-Defined Data-Center Networks, IEEE Symposium on Computers and Communications (ISCC), pp. 1–6.2019. https://doi.org/10.1109/ISCC47284.2019.8969579.

[28] S. Yan, X. Zhang, L. Zhao, and H. Zhang, Intelligent Routing Based on Deep Deterministic Policy Gradient in SDN, IEEE Communications Letters, vol. 25, no. 1, pp. 104–108, 2021. https://doi.org/10.1109/LCOMM.2020.3021333

[29] G. Kim, Y. Kim, and H. Lim, Deep Reinforcement Learning-Based Routing on Software-Defined Networks, IEEE Access, vol. 10, pp. 18121–18133, 2022. https://doi.org/10.1109/ACCESS.2022.3151081

[30] W. Wang, X. Zhang, L. Zhang, and L. Zhao, Reusable Reinforcement Learning for Intelligent Routing in SDN, arXiv preprint arXiv:2409.15226, 2024. https://arxiv.org/abs/2409.15226.

[31] L. Chen, B. Hu, Z.-H. Guan, L. Zhao, and X. Shen, Multiagent Meta-Reinforcement Learning for Adaptive Multipath Routing Optimization, IEEE Transactions on Neural Networks and Learning Systems, vol. 33, no. 10, pp. 5374–5386, 2022. https://doi.org/10.1109/TNNLS.2021.3070584

[32] K. Rusek, J. Suarez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabello, RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN, IEEE Journal on Selected Areas in Communications, 2020. https://doi.org/10.1109/JSAC.2020.3000405.

[33] X. Li, J. Li, J. Zhou, and J. Liu, Towards Robust Routing: Enabling Long-Range Perception with the Power of Graph Transformers and Deep Reinforcement Learning in Software-Defined Networks, Electronics, vol. 14, no. 3, p. 476, 2025. https://doi.org/10.3390/electronics14030476.

[34] Y. Al-Dunainawi, B. Al-Kaseem, and H. Al-Raweshidy, Optimized Artificial Intelligence Model for DDoS Detection in SDN Environment, IEEE Access,2023. https://doi.org/10.1109/ACCESS.2023.3319214

[35] Yang, C., & Li, B. (2025). SDN-DRLTE algorithm based on DRL in computer network traffic control. *Informatica*, 49(13), 175–188. https://doi.org/10.31449/inf.v49i13.7576

[36] Ma, J., Zhu, C., Fu, Y., Zhang, H., & Xiong, W. (2025). Dynamic routing via reinforcement learning for network traffic optimization. *Informatica*, 49(8), 1–14. https://doi.org/10.31449/inf.v49i8.7126

[37] Y. Guo, Q. Tang, Y. Ma, H. Tian, and K. Chen, Distributed Traffic Engineering in Hybrid Software Defined Networks: A Multi-agent Reinforcement Learning Framework, 2023. https://doi.org/10.48550/arXiv.2307.15922.

[38] J. Wang, L. Codecà, and Z. Li, Multi-Agent Deep Reinforcement Learning for Large-Scale Traffic Signal Control, IEEE Transactions on Intelligent Transportation Systems, 2019. https://doi.org/10.1109/TITS.2019.2901791.

[39] Mininet Official Repository, [Online]. Available: https://github.com/mininet/mininet.

[40] S. Bhardwaj and S. N. Panda, Performance Evaluation Using RYU SDN Controller in Software-Defined Networking Environment, Wireless Personal Communications, vol. 122, pp. 701–723, 2022. https://doi.org/10.1007/s11277-021-08920-3.

[41] Learn SDN with Ryu GitHub Repository, [Online]. Available: https://github.com/knetsolutions/learn-sdn-with-ryu

[42] Ryu-Mininet Custom Topology *Example* GitHub Repository, [Online]. Available: https://github.com/byaussy/ryu-mininet-custom

[43] M. Chiesa, G. Kindler, and M. Schapira, Traffic engineering with Equal-Cost-Multipath: An algorithmic perspective, Proceedings of the IEEE INFOCOM – IEEE Conference on Computer Communications, Toronto, ON, pp. 1590–1598,2014. https://doi.org/10.1109/INFOCOM.2014.6848081.