Dealing with Changes: Resilient Routing via Graph Neural Networks and Multi-Agent Deep Reinforcement Learning

Sai Shreyas Bhavanasi *Student Member, IEEE*, Lorenzo Pappone *Student Member, IEEE*, and Flavio Esposito, *Member, IEEE*

***Abstract*—The computer networking community has been steadily increasing investigations into machine learning to help solve tasks such as routing, traffic prediction, and resource management. The traditional best-effort nature of Internet connections allows a single link to be shared among multiple flows competing for network resources, often without consideration of in-network states. In particular, due to the recent successes in other applications, Reinforcement Learning has seen steady growth in network management and, more recently, routing. However, if there are changes in the network topology, retraining is often required to avoid significant performance losses. This restriction has chiefly prevented the deployment of Reinforcement Learning-based routing in real environments. In this paper, we approach routing as a reinforcement learning problem with two novel twists: minimize flow set collisions, and construct a reinforcement learning policy capable of routing in dynamic network conditions without retraining. We compare this approach to other routing protocols, including multi-agent learning, with respect to various Quality-of-Service metrics, and we report our lesson learned.**

***Index Terms*—Routing protocols, Machine learning algorithms, Reinforcement learning, IP networks, Network Management.**

I. Introduction

Machine learning (ML) has recently seen many applications within computer networking. Many ML techniques and algorithms have been proven to enhance performance for many tasks, ranging from network traffic prediction to resource management and anomaly detection. One of the most prolific areas of ML research in recent years has been Reinforcement Learning (RL). Since the authors in [1] demonstrated how deep neural networks could be trained to approximate a Q-function efficiently, RL became a topic of significant prominence. Since then, RL has been in the spotlight due to a slew of recent artificial intelligence breakthroughs, including defeating humans in games (e.g., Go, chess, StarCraft), self-driving cars, smart-home automation, and service robots, to name a few applications. Computer networks in general, and packet routing problems in particular, have also been solved using RL, although with a single agent to learn the environment and cope with the lack of performance awareness of commonly deployed routing protocols based on Dijkstra and Bellman-Ford algorithms. Despite all recent RL achievements, many simple tasks can still elude a single agent. The main limitation of existing single-agent RL-based routing is that the agent has to implicitly learn the topology through a centralized strategy. While such a strategy may work for logically centralized Software-Defined Networks, it may be impractical for many other computer networks. To overcome this limitation, we propose to tackle the routing problem using Graph Convolutional Networks (GCN) and distributed learning approaches, such as multi-agent RL.

In multi-agent RL, agents share a joint objective function and can cooperate to learn faster, safeguard privacy to a certain extent, and rely on a less failure-prone algorithm, overcoming some of the limitations of a single RL agent. Both GCN and multi-agent RL help tame the complexity of creating a fully distributed routing learning strategy that balances picking short and less congested paths based on local observations and being informed about global and dynamic network states.

Recent literature has shown different techniques to embed RL models into routing algorithms, achieving promising results. Nonetheless, no prior solution considered the resiliency against network changes as a primary concern. A practical example is the failure of one or multiple links or nodes in the network, which leads to unwanted and unexpected topology variations. If such an event occurs after the training phase has been completed, the RL model should be retrained from scratch to learn the new change in the environment.

In this paper, we approach the problem of RL-based routing with resiliency in mind. In particular, by resiliency in this work, we mean the ability to dynamically adapt to computer network changes without the need for retraining the RL model. Our results can be applied to Wide-Area Networks, within intra and inter-domain routing (e.g., iBGP and eBGP), and to Software-Defined networks. In some cases, e.g., at network edge [2], to fine-tune traffic engineering policies, it is desirable to overwrite classical interior BGP routing rules, such as Equal Cost Multi-Path (ECMP) and Open Shortest Path First (OSPF). In other cases, a distributed approach is the only viable solution [3].

In particular, in this paper we present the design, implementation, and evaluation of two resilient RL-based routing schemes with different learning algorithms, a single agent, and a multi-agent solution. Both approaches show benefits when dealing with drastic network topology changes. The single-agent RL routing algorithm leverages Graph Neural Networks [4] to minimize retraining needs. Instead, the multi-agent RL solution is based on Deep Q-Networks, where federated routing agents cooperate to achieve a shared optimization goal.

The idea behind our *Single-Agent Graph Convolutional Network* algorithm (SA-GCN) is to operate directly on network

All authors are affiliated with the Department of Computer Science, Saint Louis University, St. Louis, MO 63103 USA (e-mail: first.last@slu.edu).

This work has been supported by NSF Awards # 1908574 and # 2201536. Bhavanasi completed this work as an undergraduate student. We extend our gratitude to Hunter Parks, for his contributions as an undergraduate student during the initial stages of this project.

traffic datasets encoded in a graph format, instead of a traditional vector and matrix data representations. The advantage of training a RL routing system whose input is a graph lies in the ability to train such policy on any network topology, represented by a computer network adjacency matrix. This advantage naturally permits a RL policy (i.e., a given reward function) to operate in any topology and learn from changes in routing and congestion events, without having to retrain the neural network. In our *Multi-Agent routing with Deep Q-Network* algorithm instead (MA-DQN), each agent makes its routing decisions locally.

We found that when RL agents learn about the network topology explicitly, the training time significantly improves, not surprisingly. This is because, by compiling the network topology into the RL agent's state space, the machine learning model is able to handle changes in latency and bandwidth more efficiently. We also found some surprising results by evaluating several RL models. In particular, we test their ability to optimize Quality of Service (QoS) computer networks metrics, such as latency and bandwidth, while considering different network topology factors (i.e., size, number of competing flows on the same physical or virtual link, congested links, and link and node failures scenarios).

We report that our RL-based algorithms outperform the standard routing algorithms with an overall improvement evaluated with different metrics. Furthermore, we compare our solutions and show that our MA-DQN routing algorithm can achieve the optimal policy much faster than our SA-GCN model. Nonetheless, one of the major drawbacks of MA-DQN is that the model cannot easily be translated to other topology configurations, as such topology is encoded into the neural network model itself, whereas the SA-GCN algorithm results in a more adaptive solution with regards to topology changes. In summary, we analyze and dissect the application of RL on the routing problem as follows: (i) reporting the pros and cons of using single-agent and multi-agent approaches individually; (ii) comparing them in terms of network change resiliency and retraining needs; (iii) discussing our lesson learned and leaving several insights regarding the overall optimization process when dealing with routing with network topology changes; (iv) comparing their performance against the existing standard and RL-based routing algorithms.

The rest of the paper is structured as follows. Section II highlights the state-of-art RL-based solutions in routing problems, focusing on single and multi-agent approaches. Section III formulates our problem more formally and reports the mathematical model for both GCN and DQN-based algorithms. Section IV reports the results achieved with an extensive evaluation of performance in terms of QoS metrics and gives a detailed comparison between our algorithms and different baselines. In Section V we summarize our work and the take-away messages.

II. Related Work

Routing protocols pose challenging requirements for ML models. Examples of such challenges include the capacity to deal with and scale complex and dynamic network topologies, the ability to learn the correlation between the selected path and the perceived QoS, and the ability to forecast the repercussions of routing decisions. Traditional RL algorithms, particularly Q-learning, have been used to route traffic in a variety of network scenarios, given their scarce compute and communication needs and their ability to identify an ideal solution and adapt to changes in the environment. Different techniques to apply RL to the traffic routing problem have been proposed in literature. See e.g., these examples [5, 6, 7] or this recent survey [8]. These techniques differ in terms of (i) learning capability distribution and (ii) the amount of collaboration among numerous learners. Different techniques lend themselves better to specific network topologies and utility purposes. The presence of a central node —the controller in Software-Defined Networks (SDN) and the sink in Internet of Things (IoT) networks, respectively — allows for centralized learning in SDN [9] and IoT. Routing in IoT, on the other hand, necessitates decentralized RL [10, 11], with the learning capability dispersed across the routing nodes. In the rest of this section, we focus on solutions that most closely match our contribution, with respect to two dimensions: Single-Agent Q-learning solutions for traffic engineering and routing with Multi-Agent learning.

Single-Agent Deep Q-learning for Traffic Engineering. A few recent solutions have proposed employing Deep Q-Learning to tackle traffic engineering problems. Most of these approaches investigated the use of RL specifically for real-time routing optimization [12] congestion control [13], and resource management [14]. The authors in [15] implement a deep-RL solution to improve the performance of baseline TE algorithms of an SD-WAN-based network in terms of service availability. Specifically, they evaluated three deep-RL methods: Deep Q-Learning, policy optimization, and TD-λ (Temporal-Difference value function algorithm). Results show that Deep Q-Learning achieves better performance with respect to the other deep-RL algorithms and the baselines in terms of the percentage of time in which the service is up.

In [16], a deep Q-Learning is used to specifically build a greedy online routing algorithm, improving different QoS metrics in SDNs. They implemented a greedy online QoS routing method based on dueling deep Q-network with prioritized experience replay, proving that this solution can learn the network topology to solve multiple QoS metrics optimization tasks. The approach reduced delay, cost, and loss while maximizing bandwidth, outperforming existing learning-based methods. Differently from all these sound solutions, in this work we investigate the application of GCNs to Deep Q-Learning for a QoS routing optimization problem.

Routing with Multi-Agent Reinforcement Learning. Although previous studies of DRL-based techniques have demonstrated the ability to deploy routing configurations in dynamic networks autonomously, some researchers have argued that centralized controller approaches are likely to face challenges in large-scale networks due to the difficulties of collecting widely distributed network status in real-time.

Multi-Agent Reinforcement Learning has rapidly become an important research direction. A few authors have proposed bringing these approaches to optimize routing protocols. See e.g., [17, 18, 19]. In particular, [19] proposes a multi-agent reinforcement learning framework for adaptive routing in communication networks, which takes advantage of both the real-time Q-learning and the actor-critic methods.

Pinyoanuntapong *et al.* [20] formulated the traffic engineering decision-making problem as a Multi-Agent Markov decision process (MA-MDP) instead of a Partially Observable Markov Decision Process (POMDP). Similar to these approaches, we use Q-routing technique for traffic-aware routing, but our solution uses a fully distributed multi-agent Deep Q-Learning Reinforcement Learning algorithm to deal with QoS requirements and topological changes.

III. Model and Background on GCN and MA-DQN

In this section, we detail the design of our single-agent and multi-agent routing approaches. Specifically, we describe the RL model and define the state and action space along with the selected reward functions both for GCN-based algorithm (Section III-A) and DQN (Section III-B).

A. Single Agent GCN Policy Background and Settings

While Convolutional Neural Networks (CNNs) have performed well with structured data, they cannot be used for unstructured data such as graphs. In these situations, Graph Convolutional Neural Networks (GCNs) have shown great promise. GCNs use convolutional operations to extract features from nodes and edges in a graph. The main idea is to propagate information from neighboring nodes to update the representation of each node in the graph. This is achieved by defining a convolution operation on the graph, where the filters are learned using backpropagation. The filters capture local patterns in the graph and are used to update the feature representations of the nodes. We leverage GCNs as the policy network for our DRL algorithm. A policy network is a type of neural network that takes in the current state of an environment as input and outputs a probability distribution over actions that the agent can take in that state.

The application of RL-based approaches to routing have been criticized for the inability to generalize across different topologies used for training. If there are any changes in the network topology, retraining is required. This restriction has mostly prevented the deployment of RL-based routing in real environments. The principal benefit of incorporating Graph Convolutional Networks (GCNs) into the policy network lies in its capacity to deliver high performance even when operating on previously unseen network topologies. This is primarily due to the fact that the topology of the network is explicitly provided as input to the policy network. In our work, each episode of our RL algorithm is trained on a new network topology that the model has not encountered during the training phase.

As a network experiences congestion when several flow sets coexist on the same underlying path, *we address this problem by learning how to minimize the coexistence of flow sets as long as alternative routes exist. We define such flow coexistence as a collision, when two or more flow sets coexist in the same forwarding application process, i.e, a (virtual) router or switch, simultaneously.*

Since an action that an RL agent chooses influences subsequent actions, we model the problem as a Sequential Decision-making Problem (SDP). The problem instance in our study is characterized by the tuple: $M =< S, A, R, \gamma >$, where S is a finite set of states, A is a finite set of actions, R is the immediate reward, and γ is the discount factor. We denote f as the number of flow sets, and N as the number of nodes in the system. In order to optimize the policy network output (i.e., the probability distribution of actions), we employ an "on-policy approach". Specifically, we use the PPO (*Proximal Policy Optimization* [21]) reinforcement learning algorithm to facilitate stable training and prevent divergence.

We next describe each element of the Sequential Decision-making Problem tuple in detail.

State Space. Let S denote the finite set of states that are admissible in the environment. An arbitrary state contains three parts. The first part is the adjacency matrix of the network. The second part is a 2-column matrix containing a one-hot encoding of the source and destination nodes of the flow set to be routed by the GCN. The third part is a $f \times 2$ matrix where each layer of the matrix is a similar 2-column matrix but of a competing flow set in the network. It is important to clarify that the state does not contain what path other flow sets are taking across the network but merely their source and sink nodes. If there are n nodes and up to f other flows, there are $(f + 1) \times n^2$ possible states.

Action Space. Let A denote the finite set of actions that the agent can take. We construct the action space as a one-hot vector the size of the highest degree node in the graph. Having the action space be $dim(G)$ as opposed to $|E|$ offers a significant action space compression. When the agent is at a node i, given that $dim(n_i) < dim(G)$, only the first $dim(n_i)$ output indices are considered when sampling from the policy.

RL Reward Functions. Let $R(s, a)$ denote the immediate reward (or expected immediate reward) received for selecting an action (routing decision) $a \in A$ at a state $s \in S$ which causes the state transition $s \to s'$, and $R(s, a) \in [-1, 1]$. A dimension of our GCN-based routing policy evaluation is analyzing reward functions and their influence on the strength of such policy concerning what it was supposed to optimize. The first reward function considered is shown in Equation 1. This discrete, sparse reward function, while simple, is deceptive. The goal of RL algorithms is to select actions that will yield the largest reward by the end of the episode and attempts to achieve the largest reward in as few actions as possible. If an agent were to use this reward function to learn to route within a network, the agent would learn how to get to the destination node in as few hops as possible while oblivious to each link's link capacity and delay.

$$R(s, a) = \begin{cases} -1 & \text{could not reach destination} \\ 1 & \text{reached destination} \\ 0 & \text{still in transit} \end{cases} \tag{1}$$

The natural second reward structure is defined in Equation 2; in such equation, r is the proportion of the best arbitrary QoS metric possible to the QoS realized by the path the agent chose from source to destination.

$$R(s,a) = \begin{cases} -1 & \text{could not reach destination} \\ \text{r} & \text{reached destination} \\ 0 & \text{still in transit} \end{cases} \tag{2}$$

For routing with other flow sets coexisting in the network, r is the ratio between the measured metric and the optimal QoS metric (Equation 3). If we consider end-to-end latency as a metric, C_1 represents the measured latency on a given path taken into consideration as an RL policy, and C_2 is the latency of the shortest path when no other flow sets coexist in the network. If instead we wish to optimize over bandwidth, C_1 and C_2 represent the bandwidth available on the route considered and the best bandwidth that can theoretically be achieved when no other flows are present in the network:

$$r = \frac{C_2}{C_1}. \tag{3}$$

The agent operates in an environment with three inputs: (1) an adjacency matrix representing the connections between nodes in the network, (2) a two-column matrix indicating the current node and the destination node, and (3) a matrix of size $numflows \times 2 \times dim(G)$ representing the start and endpoints of other flow sets in the network, which are one-hot encoded.

The first input is processed by a graph convolutional network (GCN) specifically designed to handle graph-structured data. The GCN architecture is inspired by traditional convolutional networks and consists of a graph convolution layer, a rectified linear unit (ReLU) activation function for non-linearity, and a graph pooling layer to aggregate information. This implementation is described in detail in the GCN implementation [22].

The second and third inputs are processed by a feed-forward network with two hidden layers of size $\frac{|E|}{2}$ with ReLU activation. The output of the GCN is flattened, and the other two sub-networks' outputs are concatenated lengthwise. This resulting vector is then propagated through another sub-network, a feed-forward neural network with three hidden layers of size $\frac{|E|}{2}$. The last layer of the policy has $dim(G)$ nodes. The value function is another feed-forward network with three hidden layers of size $\frac{|E|}{2}$ with ReLU activation, and the last layer contains one neuron.

Reward Engineering and Considerations over Large Networks. While Equation 2 would properly train an agent to optimize an arbitrary QoS metric r, the sparse nature of non-zero rewards in the functions is undesirable. Mathematically, as the average eccentricity of a network increases, the RL agent's horizon organically grows, making the *credit assignment* of actions increasingly difficult. Our reward structure is semi-sparse, as the agent is given a small reward when it successfully moves closer to the destination node and receives a much larger reward at the end of the episode. Naively, if the agent takes the shortest path to the destination node, it would receive positive rewards at every timestep. However, the final reward would not be as significant if the agent collided with other flow sets along the way.

Another challenge when operating on large networks is the use of $\epsilon - greedy$ exploration, namely a strategy for balancing exploration and exploitation by randomizing actions. With random exploration, a packet is increasingly unlikely to stumble upon the destination node, therefore delaying the reception of a non-zero reward, hence prolonging the learning phase. To address such limitations, we consider the following more specific reward function (Equation 4):

$$R(s,a) = \begin{cases} -1 & \text{could not reach destination} \\ \text{r} & \text{reached destination} \\ 0.1 & \text{moved closer to target} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

B. Multi Agent DQN Background and Settings

One of the first approaches to Deep Learning from high dimensional input is outlined in [1]. Their model, a Convolutional Neural Network trained with a variant of Q Learning, could surpass some of the previous approaches on the Atari 2006 games. The authors in [23] took this approach further: they used reinforcement learning combined with deep neural networks to develop a deep Q-Network (DQN).

While tabular methods come with convergence guarantees and work well with smaller state-action spaces, we use neural networks and Deep Reinforcement Learning as we aim to develop a more flexible and scalable approach that can be easily adapted to larger and more complex problems. Deep RL provides a framework that allows us to handle these more challenging future scenarios without significant alterations to the underlying algorithm.

Using neural networks for RL can cause instability during training. This instability is attributed to correlations caused by sequences of observations, minor changes to the neural network that can change the policy, and correlations between the action and target values [23].

To counter these issues, we use the following techniques with DQN to stabilize training. First, we employ a replay buffer in our algorithm that keeps track of the (s, a, r, s', a') tuple. A batch of these tuples is sampled randomly to adjust the neural network, preventing data correlations. Secondly, DQN uses an iterative update that periodically adjusts action values to the target values to avoid correlations.

The computer network in which the RL agent operates is described by a standard graph $G(V, E)$ where V represents the set of routing nodes and E represents the set of transmission links. Each transmission link contains some traffic that we simulate. The goal is to find the path that minimizes the travel time between each source (s) and destination (d).

Each node acts as an independent agent in the MA-DQN model. Each agent has a separate neural network and makes routing decisions locally.

State Space. Each agent receives the destination as a one-hot vector with size equal to $|V|$, i.e. the number of routers. The state space remains consistent across all agents for each episode.

Action Space. We also create the action space as a one-hot vector whose size is equal to the number of agents' neighbors.

The agent selects a neighbor at each time step t after looking at the state space.

Reward Functions. We employ a dense reward function to route in larger networks. The agent receives either a positive or a negative reward for every routing decision taken based on Equation 5-6, We find that this structure significantly increases the rate at which the model converges. We also discover that the only factors that significantly alter the model's performance are the positive and negative rewards' relative sign and magnitude differences. When the magnitude of the negative reward is greater than the magnitude of the positive reward, the model avoids making those judgments and its behavior is more strongly reinforced.

Therefore, if the packet is more than one step away from the destination:

$$R(s,a) = \begin{cases} r & \text{packet moved closer to the destination} \\ -.75 & \text{otherwise,} \end{cases} \tag{5}$$

where r represents the time difference between the current node and the previous node in terms of packet delivery time to the destination. When a packet can reach the destination using a single link, the reward is defined as:

$$R(s,a) = \begin{cases} 1.5 & \text{best link to destination chosen} \\ -.75 & \text{sub-optimal link to destination chosen} \end{cases} \tag{6}$$

The agent is given a one-hot encoding of the destination as input. A feed-forward neural network with a ReLU activation receives the input after that. The size of the input layer is the same as the number of graph nodes. Then, there are two hidden layers with a combined size of $\frac{|V|}{2}$, followed by an output layer with a size equal to the number of neighbors for the agent.

The DQN algorithm that we used can be found in Algorithm 1. We first initialize the replay buffer and the action and target Neural Networks for each agent (lines 1-4). For each episode, we randomly determine the source and destination nodes. We then assign the packet to the agent at the source (line 8). For each epoch in each episode, the agent observes the state s at timestep t. The agent then selects an action a based on ϵ. The agent chooses either a random action or the best action a^* based on the following maximization problem:

$$a^* = \arg\max_a Q(s_t, a; \theta),$$

depending on the value of ϵ (lines 9-11). The value of epsilon decays as the episodes pass so that the agent can explore at the beginning of the training and exploit the best-known values toward the end. We find that the rate at which ϵ decays plays a significant role in the model's performance. If ϵ decays too fast, the model converges to a poor value. If the model decays too slowly, the model never converges. Once the action is selected, we pass the packet to the neighbor based on the selected action and observe the reward r_t and the next state s_{t+1} (lines 12-13).

Algorithm 1 Deep Q-Network Running on Routing Agent

```
1:  for agent $i = 1, N$ do
2:      Initialize replay buffer $D_i = \emptyset$
3:      Initialize action-value function $Q_i$ with weights $\theta_i$
4:      Initialize target action-value function $\hat{Q}_i$ with random
        weights $\theta_i^- = \theta_i$
5:  end for
6:  for episode = 1, $M$ do
7:      for each decision epoch $t$ do
8:          Assign current agent $n$ current packet $p$
9:          Observe current state $s_t$
10:         Select and execute an action
11:         $a_t = \begin{cases} \text{a random action, with probability}\epsilon \\ \arg\max_a Q_n(s_t, a; \theta_n), \text{with prob.} 1-\epsilon \end{cases}$
12:         Forward $p$ to next agent $v_t$
13:         Observe reward $r_t$ and next state $s_{t+1}$
14:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D_n$
15:         Sample minibatch $(s_j, a_j, r_j, s_{j+1})$ from $D_n$
16:         $y_j = \begin{cases} r_j, \text{ if episode terminates at step } j+1 \\ r_j + \gamma \ \max_{a'} \hat{Q}_n(s_{j+1}, a'; \theta_n^-), \text{ otherwise} \end{cases}$
17:         Grad. descent on $(y_j - Q_n(s_j, a_j; \theta_n))^2$ w.r.t. $\theta_n$
18:         Every $C$ steps reset $\hat{Q_n} = Q_n$
19:     end for
20: end for
```

We store the transition (s_t, a_t, r_t, s_{t+1}) in the agent's replay buffer (line 14). We then randomly sample a batch of transitions (s_j, a_j, r_j, s_{j+1}) from the agent's replay buffer and calculate the expected value y_j from the "older" target network (lines 15 - 16). We then perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ for the agent to optimize the weights of the neural network (line 17). Periodically, the weights of the target Neural Network are reset to the "newer" Neural Network (line 18). By taking samples of transitions, we can avoid correlations in training. Additionally, this replay buffer technique is more efficient as each step can be used in many neural network updates [23].

IV. Evaluation

In this section, we evaluate both proposed algorithms, single-agent via GCN and multi-agent via DQN, with respect to Quality of Service (QoS) metrics for traffic engineering.

In particular, in Section IV-A we describe our multi-agent evaluation settings. In Section IV-B we discuss the evaluation results of the multi-agent algorithm, showing how it outperforms two widely deployed algorithms, Open Shortest Path First (OSPF) and Equal-Cost Multi-Path (ECMP). We also discuss a few lessons learned from reward engineering and the impact of over-training the RL algorithm in our case. In Section IV-D, we discuss the evaluation of the centralized routing solutions using GCN, showing interesting and surprising results on its ability to learn not only how to route, but also on how to avoid collisions (i.e. coexistance and hence competition) with other flow sets.

The MA-DQN model is able to train faster than SA-GCN and its ability to re-train can be used over new topologies with different parameters. Indeed, we found that MA-DQN performs a fast re-training when there are small changes in topologies. The MA-DQN model is also able to learn an optimal policy without explicit knowledge of other flows in the network.

A. MA-DQN Evaluation Settings

In all our experiments, we captured transmission and queuing delays, and we considered network topologies following either Waxman (random-biased) or Barabasi-Albert (preferential attachment) topologies. Our network configurations consist of three sizes: 50 vertices and 100 edges (50V, 100E), 100 vertices and 200 edges (100V, 200E), and 150 vertices and 300 edges (150V, 300E). These topologies were generated using the BRITE topology generator [24]. We used a uniform bandwidth distribution from 0 to .99 Mbps. For the Waxman topology [24], we used a 0.15 alpha parameter and a 0.2 beta parameter (Waxman-specific exponents), given in this equation:

$$P(u,v) = \alpha e^{-d/(\beta L)}, \tag{7}$$

where $P(u,v)$ is the probability of having an edge between nodes u and v, $0 < \alpha$, $\beta <= 1$, d is the Euclidean distance between nodes u and v, and L is the maximum distance between any two nodes. We simulate several traffic scenarios in static and dynamic topology conditions after generating these topologies. Each link has a bandwidth as generated by the BRITE topology generator before the simulation. We simulate traffic by generating flows that take the shortest path from randomly generated source-destination pairs following a uniform distribution. For each packet on a link, we reduce the corresponding residual capacity and increase the latency proportionally to the flow packet size. In all our network simulations, we do capture propagation delay as well as queueing delay. Unless otherwise specified, we use a replay buffer size of 5000 — the memory that stores state-action tuples of the RL algorithms, a batch size of 192, the RMSprop [25] optimizer — RMSprop maintains a moving average of the squared gradients thus resulting in an adaptive learning rate. We use the default learning rate of 10^{-2}. All experiments for the MA-DQN were performed on a Dell Inspiron 15. Convergence times for the MA-DQN model w.r.t to topology size are as follows: a) (50N, 100E) took 5 minutes, (100N, 200E) took 10 minutes, (150N, 300E) took 17 minutes.

B. Evaluation of Multi-Agent DQN

1) Multi Agent DQN based policies outperforms OSPF and ECMP: In this section, we compare the latency and bandwidth of converged DQN policies, OSPF, and ECMP. Figures 1(a-c) and 2(a-c) compare the latency of the aforementioned policies on networks of various sizes and topologies. Figures 1(a-c) and 2(a-c) show that Multi-Agent DQN-based policies learn near-optimal routing policies with networks of sizes up to (150V, 300E). Our results presented in Figure 3 demonstrate the ability of the MA-DQN model to balance bandwidth effectively, consistently outperforming OSPF and ECMP.

One of the most significant results from our experiments is the model's scalability. We find that the multi-agent model can outperform classically adopted routing protocols such as OSPF and ECMP even on large networks. From Figures 1(d), 2(d), 5 (a-b) we observe that the training time scales well with respect to the topology size, and from Figure 5 (c) we observe that the MA-DQN is able to minimize latency across various topology sizes and types. We attribute this result to the fact that each agent is responsible only for local routing decisions. This means that such an agent does not need to receive the current location of the packet in the state space, thus reducing the size of the inputs to the model. Our application of a dense reward function also plays a significant role in performance.

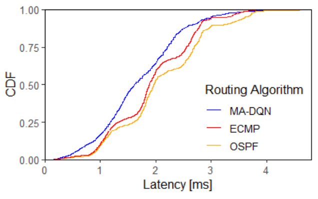
2) Impact of Retraining Needs: The present set of experiments aims to investigate the performance of a model when the network topology changes, specifically when nodes and links become unavailable. The objective is to comprehend how well the model adapts to these alterations and to assess whether adjusting the exploration-exploitation balance could improve the model's performance in such scenarios.

To simulate topology changes, we randomly remove a percentage of nodes and links from a 50-node network generated using the Waxman topology model after the RL model training had reached convergence. For clarity, the average performance over 20 runs is plotted in Figures 4(a-b). Similar results are obtained on the Barabasi-Albert topologies.

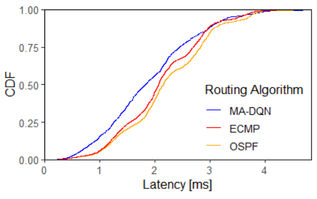
Before training or testing the model with a source-destination pair, we verify if a valid route is available. When a valid route is unavailable, sampling is continued until a connected pair is found. After training the model, we route 2000 packets for evaluation. However, in some cases, heavy reductions of nodes result in a topology change too large for the MA-DQN model to relearn an optimal policy. To address this limitation, we adjust the model's exploration-exploitation balance by changing the exploration value (ϵ), thus prompting the model to explore more instead of exploiting known routes.

In a separate set of experiments, we examine the impact of varying the ϵ value which determines the rate at which the RL agent makes random moves during its exploration phase (results shown in Figure 5d). However, we find that encouraging the model to explore more does not result in any performance improvement. The model's knowledge which was optimized for the existing environment, may not be suitable for the new environment, which is drastically different from what it had learned.

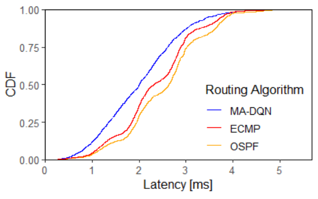
3) Lesson Learned from Reward Engineering in Routing: We explored several reward functions to determine the best. We first consider the performance of the MA-DQN model by using the reward function described in Equation 4. Figure 6(c) describes the error rate obtained when using (4). We observe that the model was *unable* to converge. *We also observe that what produced optimal results for a single agent model performed poorly in a multi-agent model.* The reward function does not penalize poor routing decisions enough but rewards the agent for reaching the destination regardless of what route it took when it was one step away from it. Such behavior leads to positive rewards for greedy routing decisions. Additionally, we noted how, since each routing decision is independent, punishing the model for failing to send the packet yielded poor routing performance results as only the last agent's weights



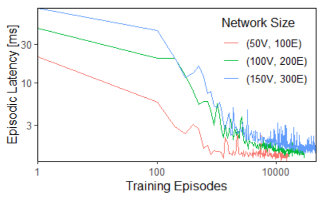
(a)



(b)

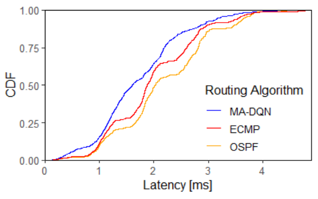


(c)

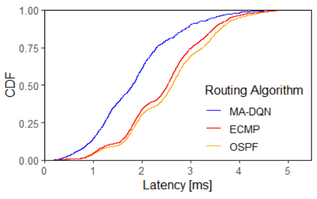


(d)

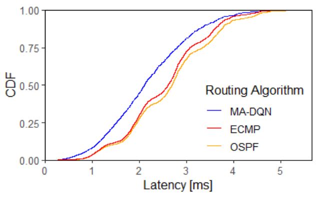
Figure 1: CDF of the latency of a converged MA-DQN model on (a) (50V, 100E), (b) (100V, 200E), and (c) (150V, 300E) Barabasi-Albert Network. We observe that the MA-DQN model is able to consistently outperform OSPF and ECMP. (d) Episodic latency of the MA-DQN model during training in Barabasi-Albert Network.



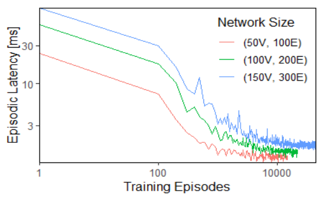
(a)



(b)

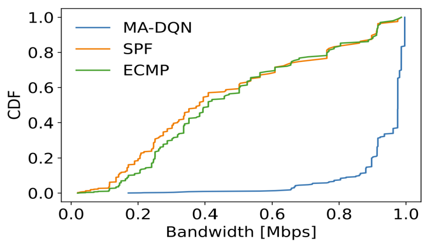


(c)

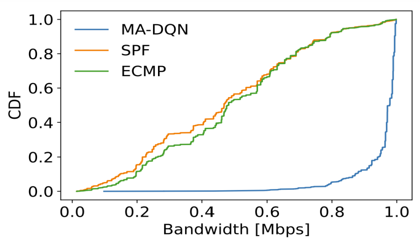


(d)

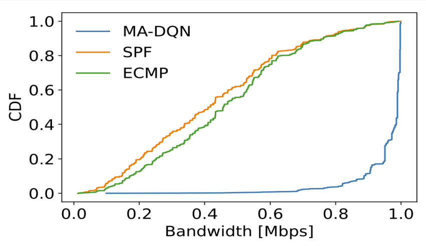
Figure 2: CDF of the latency of a converged MA-DQN model on (a) (50V, 100E), (b) (100V, 200E), and (c) (150V, 300E) Waxman Network. We observe that the MA-DQN model is able to consistently outperform OSPF and ECMP.(d) Episodic latency of the MA-DQN model during training in Waxman.



(a)



(b)



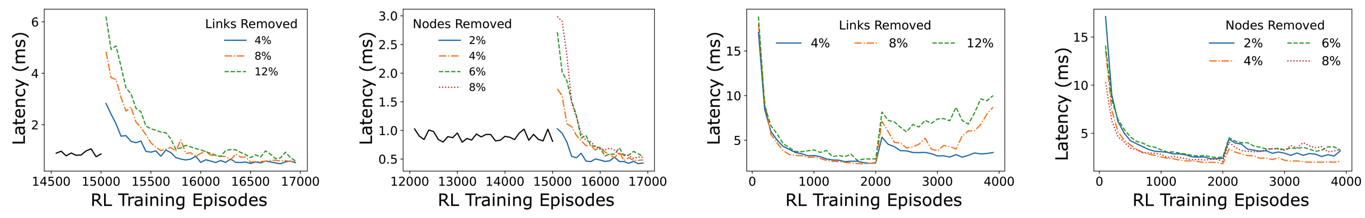
(c)

Figure 3: CDF of the bandwidth of a converged MA-DQN model on (a) (50V, 100E), (b) (100V, 200E), and (c) (150V, 300E) Waxman Network. We observe that the MA-DQN model is able to consistently outperform OSPF and ECMP.

are affected by the punishment. The previous agents' weights are not impacted if the packet is not routed. Additionally, the model gave the same reward regardless of how much closer it reached the destination. Since the reward function of Equation (4) did not provide enough incentives for better routes, the model took longer to reach the destination. After learning these lessons, we redesigned the reward functions (Equations 5 - 6). To determine the negative constant used in the reward function defined in (Equation 5 - 6), we run a grid search in the range $[-1, -.75, -.5, -.25, -.1]$ on a 100 Node Waxman topology. This allowed us to find the optimal constant. As shown in Figures 6(a) and 6(b), we find that for the agent to learn how to avoid poor routes effectively, *the magnitude of the negative reward needs to be larger than the magnitude of the positive rewards.* The model with the negative constant of -.1 did not learn the best policy. The value of the positive reward received by non-terminal packet forwarding, a, can only be in the range $(0, 1)$ since each link has a latency between $(0, 1)$. The positive terminal constant encourages the agent to send the packet via the best possible link when the packet is one step away from the destination. We choose the value of 1.5 as it is sufficiently larger than a. This reward structure brings an incentive for the agent to choose the best possible link when the packet is one step away from the destination.

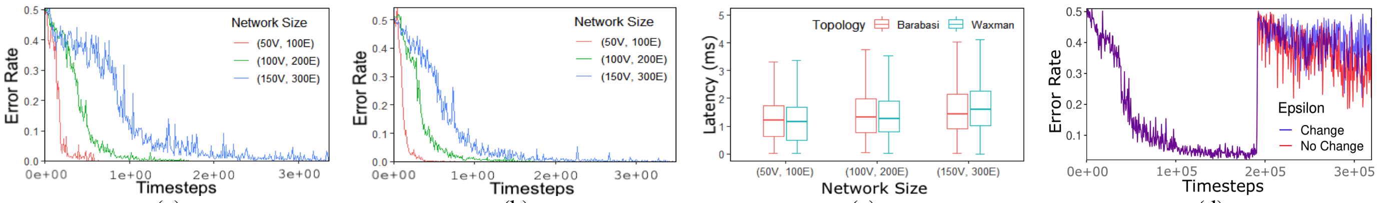
After modifying the reward, we assessed the performance of the new reward function described in Equations 5 - 6. The last agent to make a packet routing decision does not receive a punishment for failure to route the packet, as a single agent cannot be the only one responsible for such failure to deliver the packet. Additionally, the agent receives punishment for making a greedy decision despite being one step away from the destination. Additionally, the agent receives a larger reward for decisions that bring it much closer to the destination than decisions that bring it slightly closer to the destination. We found that such modifications to the reward structure lead to a significant training convergence speed-up.

4) Over-training Avoidance: In this section, we analyze the results of over-fitting the model on a 100 Node following a Barabasi-Albert network topology. One of the most significant parameters for training a RL model is the so-called ϵ: such parameter represents the trade-off between exploring and ex-



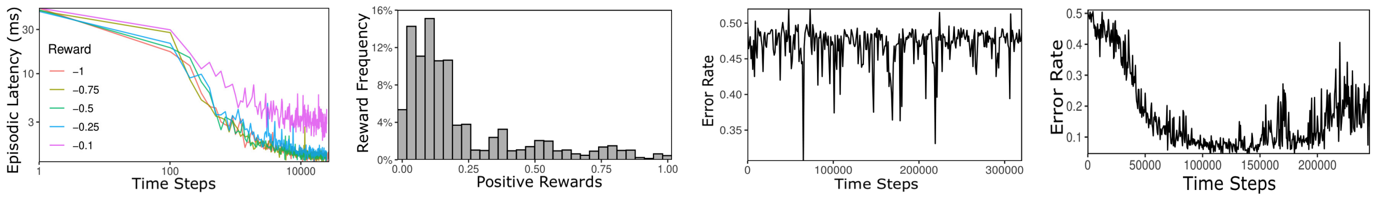
(a) (b) (c) (d)

Figure 4: Performance comparison of MA-DQN and GNN when network topology changes. (a) MA-DQN performance with links removed, (b) MA-DQN performance with nodes removed, (c) GNN performance with links removed, and (d) GNN performance with nodes removed. Both models are evaluated on 50-node Waxman networks, and the results illustrate how each model adapts to changes in network topology.



(a) (b) (c) (d)

Figure 5: Network Rate at which the agent makes poor decisions in (a) Barabasi-Albert and (b) Waxman. (c) Comparison of the latency of converged MA-DQN models across Barabasi-Albert and Waxman topologies. We observe that the models' performance remains similar across network sizes and topologies. (d) Evaluating model's performance when ϵ - the rate at which model explores - is set to an initial value of 1 after 5 nodes are removed from a 100 Node Barabasi-Albert Network.



(a) (b) (c) (d)

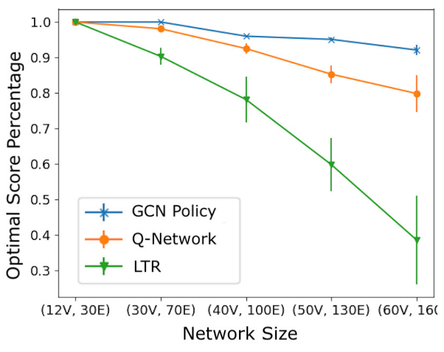
Figure 6: (a) Latency of the MA-DQN model during training when r=-.75 in Equation. 5 - 6 is replaced with different reward values. (b) Frequency of positive non-terminal rewards (a in Equation. 5 - 6) the MA-DQN model received during training when the reward for a poor routing decision is -.1. (c) The error rate for the MA-DQN model when the reward function in Equation. 4 on a (100V, 200E) Barabasi-Albert network is used. (d) Evaluating the error rate when the MA-DQN model model continues to be trained even after it has converged.

ploiting the RL environment. This parameter decays toward 0 during the training phase. When the parameter decays to exactly 0, the type of accuracy loss in Figure 6(d) does not occur. The value of ϵ decays to exactly 0 and yields results shown in Figure 5(a,b). In Figure 6(d), ϵ decayed to 0.05, causing the model to pick up on the noise and hence perform poorly.

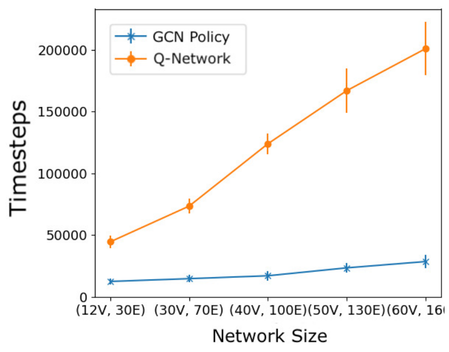
C. GCN Evaluation Settings

In reinforcement learning, two factors are commonly considered when comparing different approaches: (i) timesteps to policy convergence and (ii) strength of a learned policy. In a purely theoretical lens, the latter is the more critical metric. However, in systems where retraining is required, the former metric becomes essential. The traditional unattractive view towards machine learning-based routing comes from the protocol's inability to quickly adjust to a change in network state. This inability requires full retraining of the policy, given that transfer learning is an open and unsolved problem in reinforcement learning.

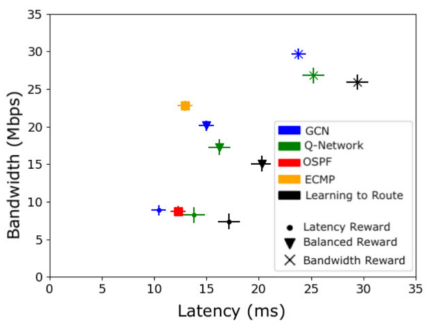
We improve the vanilla Q-Routing [26] by implementing two different approaches: Double Q Learning [27] and Dueling DQN [28]. Double Q learning fixes Q-network's tendency to overestimate the value of actions, which introduces a maximization bias in learning and ultimately leads to unstable training and negatively impacts the quality of the policy. Dueling DQN is an innovation in policy architecture: separating the value and advantage estimators into two policies, then rejoining them to select the action. The key concept behind this design decision is that it is unnecessary to know each action's value at every time step. These modifications result in more stable training (regardless of the RL application domain). In our experiments, network topologies are generated by keeping the settings used in IV-A. We train the SA-GCN model on



(a)

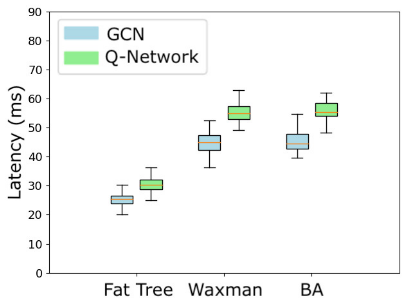


(b)

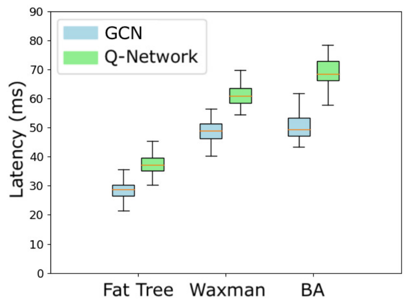


(c)

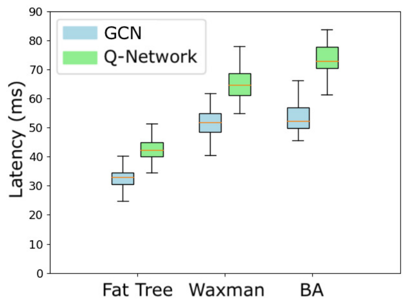
Figure 7: (a) Comparing the reward strength between a GCN policy, a Q-Network, and the reward used in the Learning to Route paper. (b) Comparing the number of training timesteps required for convergence between a GCN policy and a Q-Network. (c) Analysis of reward functions with respect to latency and bandwidth in a (150V, 350E) Waxman topology.



(a)



(b)



(c)

Figure 8: Analysis of latency for a GCN policy and Q-Network for three different network topology classes: fat tree, Waxman, and Barabasi-Albert (BA). (a): (50V, 100E), (b): (100V, 200E), and (c): (150V, 350E).

NVIDIA RTX A4000 GPUs, with 8GB of RAM, and train the model for 1 million episodes. The model requires 2GB of RAM.

D. Evaluation of GNN-Based Routing

1) GCN based Policies leads to Faster Convergence and Stronger Reward Signals: In this experiment set, we compare the characteristics of converged GCN-based policies, Q-routing policies, and the policy used by the Learning to Route paper [29], used as a benchmark (Figure 7). We also explore how the use of Equation 3 to optimize a particular QoS metric influences the routing performance in several networks.

Once the model is fully trained, we test the model using a test set of a hundred different topologies for every network size. We randomly choose the source-destination pairs. In Figures 7-a and 7-b, the reward function used is described in Equation 3, where we optimize the reward r to prioritize low-latency paths.

Figure 7-b compares the time steps required for policy convergence which requires the policy to successfully route every source-destination pair in the test set of 100 randomized topologies for every network size. The results show that the GCN approach achieves a significantly higher percentage score. Indeed, graphical convolutions are excellent at feature extraction on graph-based data. We believe that such results are due to the application of GCN to a network routing task, combined with a reinforcement learning policy update algorithm that outperforms Q-Learning in both efficiency and ability to obtain better rewards.

Lastly, we explore how various reward functions operate with respect to throughput and latency on a Barabasi-Albert graph sized (100V, 350E) — Figure 7-c. For additional comparison, the performance of two of the most commonly used iBGP routing protocols, OSPF and ECMP, are plotted as well. The three reward structures explored are as follows: (i) optimize latency only, (ii) optimize bandwidth only, and (iii) co-optimize latency and bandwidth. We find that each algorithm optimizes what each respective reward function sought to be optimized, showing that RL-based approaches can potentially be customized according to a particular set of QoS metrics [30]. In Figure 11, we compare the performance of MA-DQN with GNN-based methods inspired by [31], along with the reward function in equations 5 - 6. We observe that in smaller networks, MA-DQN and GNN perform similarly. However, in networks of size 75 nodes and above, MA-DQN outperforms GNN-based methods.

2) Latency: Figure 8-a shows that GCN-based policies can learn near-optimal routing policies with respect to the latency in networks with sizes up to (60V, 160E), whereas Q-Network's ability to minimize latency when routing within large networks is limited. In particular, to avoid re-training, the GCN policy is first trained on random topologies and separately evaluated on previously unseen ones. On the contrary,

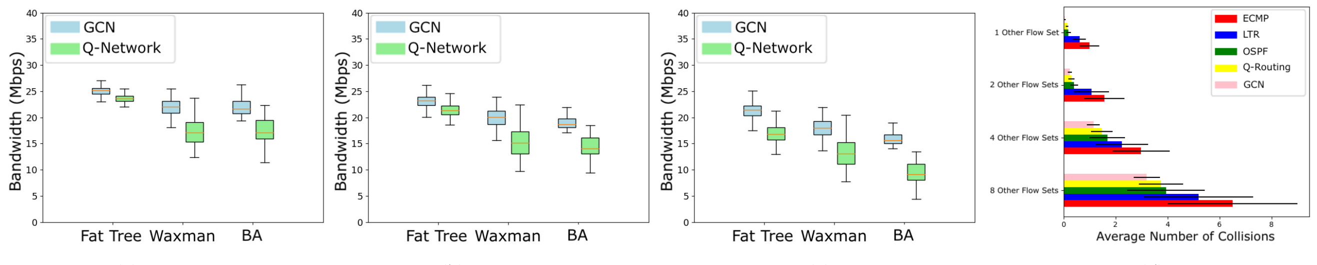

(a) (b) (c) (d)

Figure 9: Analysis of bandwidth for a GCN policy and Q-Network for three different network topology classes: fat tree, Waxman, and BA. (a): (50V, 100E), (b): (100V, 200E), and (c): (150V, 350E). (d) Comparing the average number of collisions per routing for the SA-GCN model between the routed flow and the other flow sets in the network.

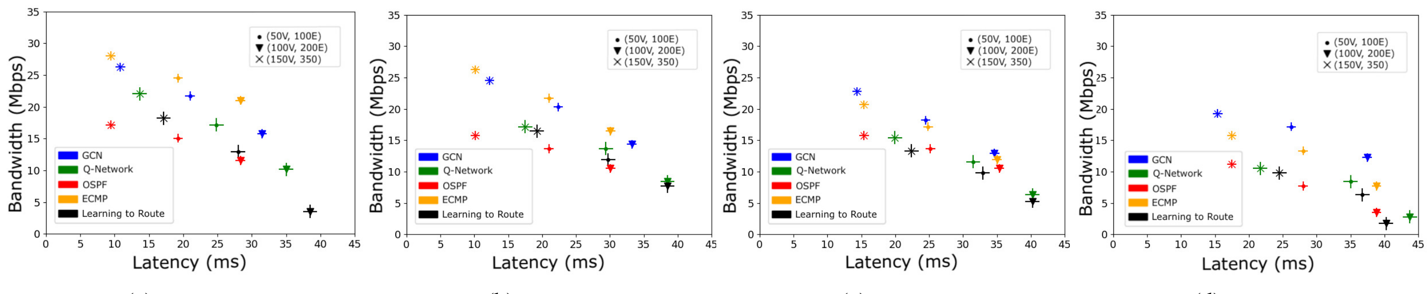

(a) (b) (c) (d)

Figure 10: Comparing QoS metrics for GCN (blue), Q-Network (green), Learning to Route (black) [29], OSPF (red), and ECMP (yellow) with a variety of flow sets: (a) 1, (b) 2, (c) 4, and (d) 8, among 3 different network sizes: (50V, 100E), (100V, 200E), and (150V, 350E).

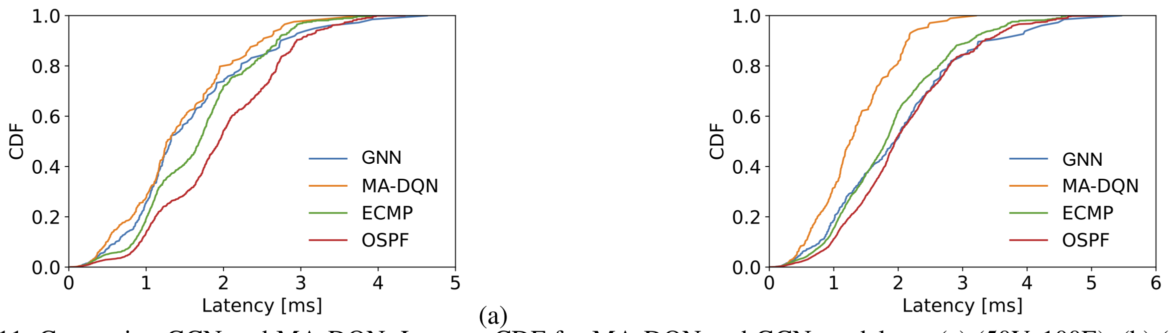

(a) (b)

Figure 11: Comparing GCN and MA-DQN: Latency CDF for MA-DQN and GCN models on (a) (50V, 100E), (b) (75V, 150E) Waxman Network.

both Q-routing and LTR algorithms are explicitly trained and evaluated on each topology. Such expressive power can be attributed to the policy construction of Q-Networks. Densely connected neural networks do not have as much expressive power as their convolutional counterpart. The performance deficiency is compounded when the data is graphical and compounded even further when the neural network has to learn the state transitions implicitly, s', from all state-action pairs, (s, a).

In Figure 8 we show the network performance on a test set of 100 networks for each topology type. We selected a Fat Tree topology to simulate data center networks. Waxman and Barabasi-Albert topology generators were used for the other two topology classes, given their popularity for modeling real-world network topologies such as intra-domains and the World Wide Web [32, 33]. In Figure 8, for each network topology of all three sizes, a trained GCN-based policy outperforms a trained Q-Network.

3) Bandwidth Improvements: Similar to the latency analysis of GCN and Q-Routing, Figure 9 (a-c) compares the two approaches and their ability to optimize bandwidth across the same three network topologies, considering the exact three network sizes. It is observed that our GCN policy construction outperforms Q-Routing in all nine scenarios.

The GCN policy is only trained once for each network topology size and then tested the previously unseen test topologies for each class without any adjustment. Contrarily, Q-Routing is trained explicitly on the test topologies. Despite this, GCN was capable of achieving better performance in terms of bandwidth than the Q-Routing solution.

4) Reward Engineering: In this experiment set, we investigate how different reward functions influence both latency and bandwidth of the routes selected to complete our analysis. The three reward functions observed minimize latency, maximize bandwidth, and simultaneously optimize both with equal weights. In these experiments, no other flow sets coexisted in the network. As an additional comparison, we included OSPF and ECMP to highlight how our policies compare to industry-standard algorithms. For single flow protocols (i.e., all algorithms expect ECMP), GCN shows to learn more robust policies than Q-routing and LTR. This evaluation shows promising observations in reinforcement learning routing protocols that are fine-tuned adequately according to the desired QoS metrics.

While GCNs outperform Q-Routing and LTR in several metrics, the evaluation thus far has been under non-stressed network conditions. That is, we assumed that there are enough resources and no competition among flows to be routed. While such an assumption may be appropriate in eBGP, we cannot route unaware of other flows for transit links within each Autonomous System or on other routing use cases. In an even small data center network, tens of thousands of flows may compete across server racks under the administration of a single domain. However, not all flow collisions are equally damaging. To assess the impact of other flow and see how our algorithm would learn how to route *while avoiding collisions*, we conducted two closely related experiments: comparing the average number of collisions each protocol endured as a function of other flow sets in the network and how said collisions impact the latency and bandwidth of the flow set being routed by the policy.

Figure 9(d) depicts the average number of collisions as a function of the number of other flow sets in a network size of $(100V, 200E)$. We measured how ECMP resulted in the highest number of collisions due to the flow being split among several distinct paths from source to destination. Our other benchmark based on Q-Routing achieved a similar amount of average collisions with the GCN policy. To quantify how the number of *packet collisions* impacts QoS metrics in Figure 10 we provided a more in-depth analysis regarding such an impact on latency and bandwidth. In the case of 1 and 2 competing flow sets (say mice and elephant), ECMP outperformed all other protocols. However, when there were 4 and 8 competing flow sets, the GCN policy's learned ability to avoid the other flow sets produced the best-realized bandwidth and latency among all the other protocols.

5) Impact of retraining needs for GNN: Similar to Section IV-B2, in Figures 4(c-d) we replicate the experiment for GNN and observe that the GNN can relearn an optimal policy for minor changes in the environment. However, with larger network topology changes, GNN is unable to relearn an optimal policy in a timely fashion. Hence we conclude that in those cases, retraining the model from scratch leads to better results.

V. Conclusion

In this paper, we explored packet routing with reinforcement learning with a few novel twists. Our objective has been to study the impact of topology and traffic changes on a trained neural network, evaluating the ability of the model to dynamically adapt without the need for retraining. To do so, we focused on single-domain routing, suitable, e.g., for interior BGP, and on multi-domain routing, valuable for larger scale routing protocols such as exterior BGP. We proposed a Single Agent RL model, based on a Graph Convolutional Network [4] (GCN) to fit the former, and a multi-agent Deep Q-Learning Network model for the latter.

We report reward engineering considerations on our routing algorithms, evaluating both single and multi-agent solutions in extensive experimental settings with different network sizes and connectivity models. Our results show that single-agent with GCN improves the ability to achieve high QoS metrics when the computer network topology changes after the RL training converges. Moreover, the multi-agent model is able to scale well with larger networks, consistently outperforming OSPF and ECMP. Our findings indicate that the MA-DQN model is capable of adapting to changes in topology in networks with up to 50 nodes, but we also observed limitations in its ability to relearn an optimal policy on larger networks. These observations highlight interesting directions for future research in this area. Our code is available with an MIT license at [34].

References

[1] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013).

[2] Brandon Schlinker et al. "Engineering egress with edge fabric: Steering oceans of content to the world". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 418–431.

[3] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. "Policy disputes in path-vector protocols". In: *Proceedings. Seventh International Conference on Network Protocols*. IEEE. 1999, pp. 21–30.

[4] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). Palais des Congrès Neptune, Toulon, France, 2017.

[5] Justin A Boyan and Michael L Littman. "Packet routing in dynamically changing networks: A reinforcement learning approach". In: *Advances in neural information processing systems*. 1994, pp. 671–678.

[6] Michael Littman and Justin Boyan. "A distributed reinforcement learning scheme for network routing". In: *Proceedings of the international workshop on applications of neural networks to telecommunications*. Psychology Press. 2013, pp. 55–61.

[7] Samuel PM Choi and Dit-Yan Yeung. "Predictive Q-routing: A memory-based reinforcement learning approach to adaptive traffic control". In: *Advances in Neural Information Processing Systems*. 1996, pp. 945–951.

[8] Zoubir Mammeri. "Reinforcement learning based routing in networks: Review and classification of approaches". In: *IEEE Access* 7 (2019), pp. 55916–55950.

[9] Shih-Chun Lin et al. "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach". In: *2016 IEEE International Conference on Services Computing (SCC)*. IEEE. 2016, pp. 25–33.

[10] Abhijeet A Bhorkar et al. “Adaptive opportunistic routing for wireless ad hoc networks”. In: *IEEE/ACM Transactions On Networking* 20.1 (2011), pp. 243–256.

[11] Zhichu Lin and Mihaela van der Schaar. “Autonomic and distributed joint routing and power control for delay-sensitive applications in multi-hop wireless networks”. In: *IEEE Transactions on Wireless Communications* 10.1 (2010), pp. 102–113.

[12] Xiaohong Huang et al. “Deep reinforcement learning for multimedia traffic control in software defined networking”. In: *IEEE Network* 32.6 (2018), pp. 35–41.

[13] Dehao Lan et al. “A deep reinforcement learning based congestion control mechanism for NDN”. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–7.

[14] Rongpeng Li et al. “Deep reinforcement learning for resource management in network slicing”. In: *IEEE Access* 6 (2018), pp. 74429–74441.

[15] Sebastian Troia et al. “On deep reinforcement learning for traffic engineering in sd-wan”. In: *IEEE Journal on Selected Areas in Communications* (2020).

[16] Syed Qaisar Jalil et al. “DQR: Deep Q-Routing in Software Defined Networks”. In: *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2020, pp. 1–8.

[17] Xinyu You et al. “Toward packet routing with fully distributed multiagent deep reinforcement learning”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 52.2 (2020), pp. 855–868.

[18] Ruijin Ding et al. “Deep reinforcement learning for router selection in network with heavy traffic”. In: *IEEE Access* 7 (2019), pp. 37109–37120.

[19] Siliang Zeng, Xingfei Xu, and Yi Chen. “Multi-Agent Reinforcement Learning for Adaptive Routing: A Hybrid Method using Eligibility Traces”. In: *2020 IEEE 16th International Conference on Control Automation (ICCA)*. 2020, pp. 1332–1339.

[20] Pinyarash Pinyoanuntapong, Minwoo Lee, and Pu Wang. “Delay-optimal traffic engineering through multi-agent reinforcement learning”. In: *Proc. of IEEE INFOCOM Workshops*. IEEE. 2019, pp. 435–442.

[21] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[22] Mikael Henaff, Joan Bruna, and Yann Lecun. “Deep Convolutional Networks on Graph-Structured Data”. In: (June 2015).

[23] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[24] A. Medina et al. “BRITE: an approach to universal topology generation”. In: *Proc, of MASCOTS*. 2001, pp. 346–353.

[25] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[26] Rocio Arroyo-Valles et al. “Q-probabilistic routing in wireless sensor networks”. In: *2007 3rd International Conference on Intelligent Sensors, Sensor Networks and Information*. IEEE. 2007, pp. 1–6.

[27] Hado Hasselt. “Double Q-learning”. In: *Advances in neural information processing systems* 23 (2010).

[28] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.

[29] Asaf Valadarsky et al. In: *31st Conference of Neural Information Processing Systems*. 2017.

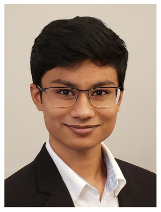
[30] Doron Zarchy et al. “Axiomatizing Congestion Control”. In: SIGMETRICS ’19. Phoenix, AZ, USA: ACM, 2019. DOI: 10.1145/3309697.3331501.

[31] Xuan Mai, Quanzhi Fu, and Yi Chen. *Packet Routing with Graph Attention Multi-agent Reinforcement Learning*. 2021. arXiv: 2107.13181 `[cs.AI]`.

[32] B. M. Waxman. “Routing of multipoint connections”. In: *IEEE Journal on Selected Areas in Communications* 6.9 (1988), pp. 1617–1622.

[33] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *science* 286.5439 (1999), pp. 509–512.

[34] Sai Shreyas Bhavanasi, Lorenzo Pappone, and Flavio Esposito. *https://github.com/routing-drl/main/*. online. 2023.



Sai Shreyas Bhavanasi is a graduate student at Saint Louis University pursuing a Master’s in Artificial Intelligence. He received his Bachelor of Science in Computer Science and Data Science from Saint Louis University. His research interests include applied machine learning, reinforcement learning, and computer networking.

Lorenzo Pappone received the M.Sc. degree in Computer Engineering from University of Naples Federico II in 2021. He is currently pursuing a Ph.D. degree in Computer Science at Saint Louis University, USA. His main research interests include applied machine learning, deep learning, traffic engineering, and distributed systems.

Flavio Esposito is an Associate Professor of Computer Science at Saint Louis University. He obtained his BS and MS in Telecommunication Engineering from the University of Florence, Italy, and his Ph.D. in Computer Science from Boston University. His research centers on the intersection of networked systems and artificial intelligence. Before joining academia, Flavio was a senior software engineer and worked in a few research laboratories in Europe and USA. He is a Principal Investigator on several research awards from the National Science Foundation. His funded projects include edge computing, machine learning for network management, next-generation wireless networks, distributed artificial intelligence, and computer security. Flavio’s main research interests include network management, network virtualization, and distributed systems. Flavio is the recipient of several awards, including several National Science Foundation awards and the Comcast Innovation Award in 2021.