Automating Network Operation Centers using Reinforcement Learning

uOttawa

Sa'di Altamimi

School of Electrical Engineering and Computer Science
University of Ottawa

Thesis submitted in partial fulfillment of the requirements for the
PhD degree in
Electrical & Computer Engineering

I would like to dedicate this thesis to
the Palestinian child
Handala!

Acknowledgements

I am deeply grateful to my supervisor, Prof. Shervin Shirmohammadi, for his unwavering trust, flexibility, and continuous support throughout my research journey. I am also grateful for the valuable insights and constructive feedback provided by the rest of my thesis committee: Prof. Emil Petriu, Prof. Hussein Al Osman, Prof. Jiangchuan Liu, and Prof. Peter X. Liu.

I would also like to extend my thanks to Dr. David Cote for the fruitful discussions we had during my internship at Ciena, which greatly enriched my understanding of the field. Additionally, I am grateful to Prof. Martin Bouchard and Dr. Homayoun Kamkar Parsi for the exciting opportunity I had during my internship at WSAudiology, which broadened my perspective and enhanced my research skills.

Furthermore, I wish to express my heartfelt appreciation to Prof. Ghandi Manasrah, an exceptional professor from my undergraduate years, whose mentorship and guidance played a pivotal role in shaping not only my academic growth but also my personal development.

Last but not least, I am profoundly grateful to my dedicated parents, Younes and Sabah, my in-laws Anas and Amal, and my amazing wife, Jana, for their unwavering belief in me and their continuous support throughout this demanding endeavor. Their love, encouragement, and sacrifices have propelled me to overcome challenges and complete this work. To my cutest daughter, Shireen, thanks for filling my life with joy and happiness. To Mohammad, Basel, Samar, and Azeeza, thanks for being an amazing family.

Abstract

Reinforcement learning (RL) has been at the core of recent advances in fulfilling the AI promise towards general intelligence. Unlike other machine learning (ML) paradigms, such as supervised learning (SL) that learn to mimic how humans *act*, RL tries to mimic how humans *learn*, and in many tasks, managed to discover new strategies and achieved super-human performance. This is possible mainly because RL algorithms are allowed to interact with the world to collect the data they need for training by themselves. This is not possible in SL, where the ML model is limited to a dataset collected by humans which can be biased towards sub-optimal solutions.

The downside of RL is its high cost when trained on real systems. This high cost stems from the fact that the actions taken by an RL model during the initial phase of training are merely random. To overcome this issue, it is common to train RL models using simulators before deploying them in production. However, designing a realistic simulator that faithfully resembles the real environment is not easy at all. Furthermore, simulator-based approaches don't utilize the sheer amount of field-data available at their disposal.

This work investigates new ways to bridge the gap between SL and RL through an *offline pre-training* phase. The idea is to utilize the field-data to pre-train RL models in an offline setting (similar to SL), and then allow them to safely explore and improve their performance beyond human-level. The proposed training pipeline includes: (i) a process to convert static datasets into RL-environment, (ii) an MDP-aware data augmentation process of offline-dataset, and (iii) a pre-training step that improves RL exploration phase. We show how to apply this approach to design an action recommendation engine (ARE) that automates network operation centers (NOC); a task that is still tackled by teams of network professionals using hand-crafted rules. Our RL algorithm learns to maximize the Quality of Experience (QoE) of NOC users and minimize the operational costs (OPEX) compared to traditional algorithms. Furthermore, our algorithm is scalable, and can be used to control large-scale networks of arbitrary size.

Table of contents

List of figures

List of tables

List of Abbreviations

AI	Artificial Intelligence
AN	Adaptive Network
ANN	Artificial Neural Network
API	Application Programming Interfaces
ARE	Action Recommendation Engine
AS	Autonomous System
BOLA	Buffer Occupancy based Lyapunov Algorithm
BPA	Back Propagation Algorithm
DASH	Dynamic Adaptive Streaming over HTTP
DES	Discrete Event Simulator
DL	Deep Learning
DNN	Deep Neural Network
HTTP	Hypertext Transfer Protocol
IAT	Inter-arraival Time
IID	Independent and Identically Distributed
ISP	Internet Service Provider
KPI	Key Performance Indicator
LTE	Long-term Evolution (standard)
MDP	Markov Decision Process
ML	Machine Learning

MOS	Mean Opinion Score
MPD	Media Presentation Description
MPLS	Multiprotocol Label Switching
NFV	Network Function Virtualization
NOC	Network Operation Center
OPEX	Operational Expenses
OTT	Over-The-Top
PGA	Policy Gradient Algorithm
QoE	Quality of Experience
QoS	Quality of Service
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RTT	Round-trip Time
SDN	Software-defined Network
SL	Supervised Learning
SLA	Service Level Agreement
T2T	Time to Threshold
ZSM	Zero-touch network and Service Management

Chapter 1

Introduction

The internet now serves 9 billion clients world-wide and consists of a large number of interconnected networks, users, sensors and devices sending petabytes of data through the network every millisecond. The recent developments in the multimedia industry, especially over-the-top (OTT) services have changed the Internet traffic to multimedia traffic. Such a drastic increase of multimedia traffic poses real challenges for internet service providers (ISPs). The task of ensuring the efficient operation of the ISP network typically lies with the Network Operation Centre (NOC). Today, the NOC operators use predefined expert rules, designed from past experience, to take remedial actions when something goes wrong. However, as the network grows larger and becomes more complex, these expert-rules become less efficient and more difficult-to-design. As a result, it is common to attribute the degradation of users' QoE to bottlenecks in the ISPs' networks.

This work proposes an automated algorithm that is capable of managing large networks by (i) identifying the root-cause of network problems and (ii) applying remedial action to resolve the detected problem in real-time.

1.1 Motivations

Traditional NOC operations were often based on heuristics (rule of thumb) rather than true mathematical optimization. The outcome of such heuristics is often limited to simple trigger actions, where as many operation (e.g., network fault resolution) are still highly manual. Next generation NOC systems employ ML to optimize across multiple tunable parameters and carry out autonomous self-healing. The adoption of ML helps NOC operators move from being reactive to becoming predictive, resolving network issues before they affect customers QoE.

Network operators are already sitting on large datasets. However, the task of extracting *actionable* insights from this sheer amount of data is one of the fundamental challenges that makes ML-based ARE hard to achieve. First, curating labeled datasets for training ML models can be costly and labor-intensive task. Second, even if NOC operators are willing to label their dataset, data-labeling can be sub-optimal because the best actions are not typically known due to complex nature of ISP networks. Finally, real networks can experience shifts in the data distribution over time due to the non-stationary nature of the environment. This phenomena is known as *dataset shift*, and it imposes another serious challenge to any ARE algorithm as it render any old dataset useless.

ML algorithms have been widely successful in making predictions or decisions without being explicitly programmed to do so. A major part of what makes ML so valuable is its ability to detect what the human eye misses. ML models can catch complex patterns that would have been overlooked during human analysis. However, this ability alone doesn't necessarily translate to better actions. The ability to take good actions depends on the type of supervision used when training the ML model. For example, the actions learned by a SL model can't outperform the human expert who generated these labels.

In this work, we chose to work with the RL-framework because it has the power to explore new actions unknown to human operator, resulting in superhuman performance. However, the traditional way of training RL models comes with its own limitations. The RL framework is not designed to utilize the rich datasets collected from the field. In contrary, an RL agent has to either interact with the real environment (which is expensive to operate) to collect its own experiences, or with a simulated version that might not be able to fully mimics the real-environment and is usually expensive to develop.

The solution proposed in this work aims at bridging the gap between SL and RL by providing an answer to the following two questions:

1. How to use RL to automate NOC operations with super-human performance?

2. How to utilize the field-datasets, which better capture the dynamics of real networks compared to the simulated environments, to pre-train RL models *safely* and *efficiently* similar to SL?

It is worth noting that, although RL models can be used to fully automate an NOC, its other practical usage is as an Action Recommendation Engine (ARE) that recommends an action to the human operator, still leaving the final decision in the

hands of people and not machines (for more on AI ethics see [18]). In either case, a successful algorithm should consider the following challenges:

- Networks' states can be highly variable, unpredictable, and unobservable. Despite that, ARE must be able to suggest/take *correct* remedial actions.
- ARE algorithm must take into account the long-term effects of its decisions by *proactively* plan for future rather than relaying on instantaneous decisions.
- Maximizing Quality of Service (QoS) objectives (e.g. bandwidth utilization or avoiding network congestion) does not necessarily result in achieving the optimum goal of NOC. ARE must instead consider optimizing an objective function that takes into account *both* user's Quality of Experience (QoE) and OPEX constraints.
- Training an RL model faces many practical problems: it takes a long time to train to achieve good performance, and during that training models make mistakes, which can be costly to make in a real network. As a result, it is important to include an offline stage that allows the model to learn *quickly and safely* before being deployed in real networks.
- Given the high cost of wrong actions, ARE should provide some level of *interpretability* that allows the human expert to assess ARE actions and take-over when necessary.
- ARE should be scalable.

1.2 Approach

The work in this thesis was motivated by the challenges mentioned in §1.1, and is an attempt to automate NOC's remedial actions. To do so, we break down this challenging task into four stages:

First, we formulate the problem of automating NOC actions as a Markov Decision Process (MDP). The main elements of this formulation are defining (i) the network *state*, represented as a combination of end-to-end (E2E) quality of service (QoS) metrics and video streaming metrics of users' QoE, (ii) the high-level remedial *actions*, and (iii) a *reward* signal, which captures users' quality of experience (QoE) as well as operational cost (OPEX).

Second, we provide a practical RL training algorithm that overcomes two of the main limitations of traditional online training: *slow training* and *dangerous exploration*.

In particular, training RL models on real networks takes a long time to train before being able to achieve good performance, and during that training it makes mistakes, which can be costly to make in a real network. To tackle the first problem, we design a fast simulator to mimic the dynamics of a real network. This environment is capable of faithfully capturing the dynamics of the real network and can be used to train an RL model in a matter of minutes rather than months. As for the second problem, we designed a pre-training environment based on field-data to brings the RL model to a level similar to a human expert before allowing it to explore new actions.

Third, we define an objective function that optimizes both the QoE and OPEX. We implement a baseline algorithm, specify testing scenarios, chose evaluation metrics, and validate the feasibility of automating an NOC with superhuman performance. The main two advantages of our formulation are (i) being topology-agnostic given that the RL algorithm acts on overlay networks, and (ii) requiring only E2E metrics which are easier to translate to actions that maintain users' service level agreement (SLA).

Finally, we extend our work to automate large-scale networks and discussed the scalability of our algorithm. This is achieved by breaking the big network into smaller networks, which are topology agnostic (e.g., tunnel groups), and training multiple RL agents to handle these sub-networks in a *decentralized* fashion.

1.3 Contributions

In this work, we design and implement an RL-based closed-loop ARE that can autonomously self-drive a network from raw data. We utilize many concepts from SL to improve RL training and demonstrated that our proposed ARE, not only saves both time and money for NOC tasks, but also achieves superhuman performance. Our contributions can be summarized as follows:

- we formulated the problem of ARE in NOC as MDP process and solve this problem using RL. We used Dynamic Adaptive video Streaming over HTTP (DASH) as a challenging use-case and demonstrated that the model can learn new implicit and non-trivial rules, on its own, achieving better users' QoE while respecting OPEX constraints.

- we designed a training pipeline that utilizes many concepts from SL to improve RL training. This includes: (i) a process to covert static datasets into RL-environment, (ii) an MDP-aware data augmentation process of offline-dataset, and (iii) a pre-training step that improves RL exploration phase.

- we scaled up the network environment, and train multiple identical agents to operate this big network in a decentralized fashion.

1.4 Publications

The contributions listed in §1.3 have led to the following publication and patent:

- (Patent) S. Altamimi; S. Shirmohammadi; D. Cote, C. Barber *"Action Recommendation Engine: Automating NOC actions using multi-agent Reinforcement Learning"*, US patent application (Under review), Feb. 3, 2023
- (Patent) S. Altamimi; B. Altamimi; S. Mohammed; S. Shirmohammadi; D. Cote, *"Action Recommendation Engine (ARE) for Network Operations Center (NOC) solely from raw un-labeled data"*, US patent application 10.2846, Feb. 3, 2021
- (Paper) S. Altamimi, B. Altamimi, D. Côté and S. Shirmohammadi, *"Toward a Superintelligent Action Recommender for Network Operation Centers Using Reinforcement Learning,"* in IEEE Access, vol. 11, pp. 20216-20229, 2023, doi: 10.1109/ACCESS.2023.3248652.

As well as winning the following awards:

- (Award) *"Action Recommendation Engine based on Machine Learning to support closed-loop applications for telecommunications networks"*, Best-idea award in 2020 Ciena's Tech Forum.

The contents of the aforementioned paper will appear in Chapter 4 and 5 and has been reused with permission.

1.5 Organization of Thesis

In Chapter 2 we present the theoretical background from areas related to the topic of this project. This includes DASH standard and RL. Chapter 3 presents a survey of the recent developments in the field of adaptive networks. The related works have been reviewed in two main categories: (i) traditional methods that use expert-rules to resolve network issues by a mean of traffic-engineering and failure recovery, (ii) ML-based approaches with more focus on online and offline RL algorithms. Chapter 4 contains the system design and implementation which details all our contributions. A summary of the obtained results, and some directions for future research in this area appear in Chapter 5, and 6 respectively.

Chapter 2

Background

This chapter provides an overview of the systems and techniques underlying the whole work. It first describes the main concepts behind network automation. Then, it provides a brief introduction to adaptive video streaming and how to measure users' QoE. Finally, it covers the main concepts of the two ML techniques used in this thesis, namely DL and RL.

2.1 The Adaptive Network

The recent developments in the multimedia industry, especially OTT services have changed the Internet traffic to multimedia traffic. Such a drastic increase of multimedia traffic poses real challenges for Network Operation Centers (NOC) who manage large-scale networks that are full of overbuilt legacy systems and protocols, leaving them unable to rapidly scale or adapt. Moreover, The "one-size-fits-all" network paradigm employed in the legacy networks is no longer suited to efficiently address a market model composed of very different applications like machine-type communication, ultra reliable low latency communication and others. This led to the development of new protocols, architectures, and technologies around the concept of *Adaptive Networks* (ANs).

The Adaptive Network is a relatively new approach that expands on autonomous networking concepts to transform the static network into a dynamic, programmable environment driven by analytics and intelligence. It represents the vision of an ideal network that utilizes intelligent automation, guided by streaming telemetry, data-drive analytics, and intent-based policies to rapidly scale, self-configure, self-heal, and self-optimize by constantly assessing network pressures and demands [5].

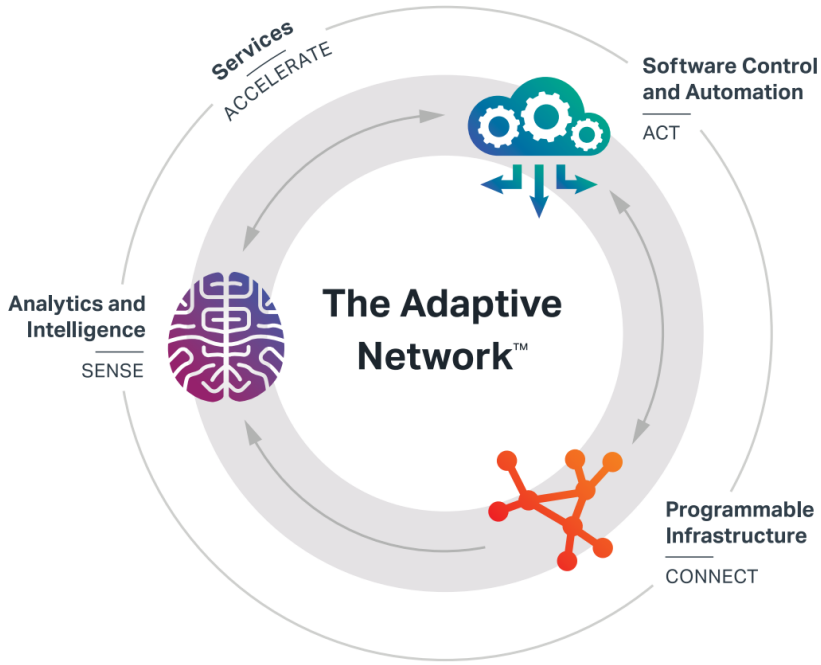


Fig. 2.1 Foundational elements of the Adaptive Network. Source: [5]

Figure 2.1 summarizes the three foundational elements of ANs. The first element is *programmable infrastructures*. It allows ISPs to configure their network devices via common and open APIs, and to access network performance data in real-time. Developing an open, multi-vendor, and multi-domain framework is crucial because it is no secret that no one vendor can deliver best-inbreed hardware, software, and services for every application.

One of the key examples of this elements is *5G slicing*. The current LTE architecture has a rigid framework that is not very flexible or scalable to adapt to diverse use cases. It often lacks customization when it comes to offering any tailored business requirements or to meet specific business demands. With growing mobile data and consumer demands, business needs for faster connectivity and higher throughput cannot be fulfilled by today's 4G LTE network. To overcome these limitation, 5G networks "slice" the physical network architecture into multiple logical and independent networks, that can be programmed and configured on demand, to effectively meet the various services requirements. Traditionally, the concept of a network "slice" is implemented in a form of *overlay networks*, and it has been proposed as a way to improve Internet routing, such as through quality of service guarantees to achieve higher-quality streaming media. 5G networks expands this concepts by decoupling each network function (e.g., routing, etc.) from the network hardware it runs on (i.e., through network virtualization). By

doing so, 5G networks are able to accommodate application with very different and possibly contrasting quality of service (QoS) requirements exploiting a single physical network infrastructure.

The other two elements of The Adaptive Network vision are *network analytics* and *remediation*. The goal is to utilize machine learning (ML) to effectively harnessed the growing wealth of data collected by NOC. Then, use the extracted insights to take more effective decisions to optimize the network performance and deliver a better customer experience. One of the key examples that utilizes these elements is *zero-touch networks* [2]. ML is used to map the collected metrics into actions, without a human in the loop, through three steps:

- ML performs a real-time *anomaly detection* of glitches that can affect multiple capabilities, domains and environments. By using adaptable algorithms that take seasonality, trends, and other behavioral variability into account, anomalies are detected faster and false alarms are reduced to a minimum. ML can also utilize the historical trends to predict a possible network anomaly (and proactively try to avoid it).

- ML perform *root cause analysis* to provide the full context of what is happening, specially when one network issue indirectly affect other network components and services. Correlation analysis across multiple architectural layers is a must if NOC are to effectively determine the probable cause of a problems such as outages, as well as service degradation and slow leaks. Identifying the root-cause of a problem pave the way to find a solution to that problem.

- ML algorithms can learn to fix network issues from labeled examples in a way that match human performance (e.g., using supervised learning). It can also be trained to explore new strategies that can outperform human performance (e.g., using reinforcement learning).

Unlike the first two elements of the adaptive network vision (i.e., programmable infrastructures and network analytics), implementing ML-based algorithms to take automated actions comes with extra requirements. On one hand, network operators call for a high level of reliability and service availability to avoid financial losses due to network outages and SLA violations. On the other hand, transparency and accountability of ML-enabled systems are crucial to build trust in their decisions as well as legal compliance. One way to workaround such a concern is to use ML to recommend actions to human operators and leave the final decision in the hands of people and not machines.

2.2 Video Streaming

Video streaming has rapidly emerged as a dominant type of traffic that accounts for more than 82% of all Internet traffic today. The need for video streaming goes beyond the entertainment industries as countless businesses are transitioning toward a work-from-home model due to the ongoing COVID-19 pandemic that causes the closure of many workplaces and campuses.

2.2.1 DASH Video Streaming

Media *streaming* refers to the technology of transmitting audio and video files in a continuous flow over a wired or wireless connection with little or no intermediate storage in network elements. Before the Internet era, the main platforms for video streaming have been cable, broadcast, and satellite televisions. These platforms enjoyed a controlled experience; there was a single bitrate, and content was purpose-built for one device so there's only one delivery combination possible.

HTTP media streaming services, such as Netflix and YouTube, are referred to as *Over-The-Top* (OTT) platforms because they bypass the traditional distribution channels and directly deliver video content to consumers over the Internet. Traditionally, OTT applications were delivered using a *progressive* download mechanism, where the media content is encoded at one single bitrate and is streamed over the internet as a single file. However, this approach doesn't fit the *best effort* nature of the often-overcrowded public Internet. Therefore, the Dynamic Adaptive Streaming over HTTP (DASH) standard emerged in 2011.

DASH provides a standardized solution for the efficient and easy streaming of multimedia using existing HTTP infrastructure. In a conventional DASH application, several copies of every video at different bitrates are segmented and stored on a streaming server. The DASH client requests an XML configuration file called Media Presentation Description (MPD), containing all the metadata that the client needs in order to stream the video. Among these data is the list of available encoding bitrates supported for the media content being requested.

Modern video players employ complex algorithms to adapt the bitrate of the video that is shown to the user. Bitrate adaptation requires a tradeoff between reducing the probability that the video freezes (rebuffers) and enhancing the quality of the video. High quality, in this context, is a function of *video bitrate* and *network bandwidth*. The choice is trivial in two cases: (i) if the network bandwidth varies but is always more than the bitrate of the highest ABR variant, then the client chooses the highest bitrate

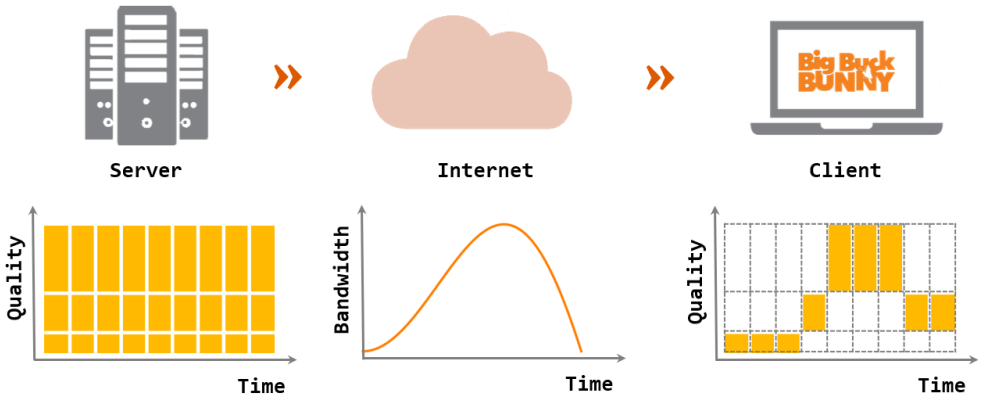


Fig. 2.2 Simplified DASH workflow: (left) a server storing multiple copies of the same video at different video *qualities*, (right) a client adapting its bitrates based on network conditions (middle).

variant. (ii) if the network bandwidth is invariant, then the client chooses the highest bitrate variant that is less of equal to the network bandwidth. In practice, network conditions can be highly variable and difficult to predict. Choosing a bitrate that is too high leads to frequent rebuffering, while choosing a bitrate that is too low leads to poor video quality. Figure 2.2 illustrates this mechanism.

Given the uncertainty in network bandwidth, clients maintain a video buffer of a few seconds duration, to prevent stalling the playback. The buffer is realised as a queue of fixed duration slots, with each slot being filled a video segment of optimally chosen bitrate. Note that segments in different slots may differ by size-in-bits, for the same slot duration (indicated by segment size in Fig. 2.2). Generally speaking, the client can choose the bitrate of video segment to be downloaded next in two ways:

- *Bandwidth-based*: estimates the available video bandwidth and pick the highest bitrate that is less than or equal to the estimate. This is quite intuitive but requires a good prediction of network conditions, which is usually difficult given the complexity of ISP networks. This is even more difficult for DASH clients because their estimate of network bandwidth is solely based on local information given the fact that DASH is a client-side framework.

- *Buffer-based*: measures the duration of video left in the buffer, and accordingly download a higher bitrate segment (if the buffer is sufficiently occupied) or a lower bitrate segment (if the buffer is running low). This class of algorithms do away with the need to estimate bandwidth and use the buffer occupancy as an indirect indicator of the bandwidth. This approach, though not quite intuitive,

is backed by sound theory. The notable BOLA algorithm is an example of this type of algorithms [32].

It is worth mentioning a technique, called *fast switching*, often used by DASH clients to improve users QoE at the expense of a little bit of server-client traffic overhead. To illustrate this process, assume that the buffer is currently filled with optimal bitrate segments. And say the bandwidth suddenly improves, allowing higher bitrate segments to enter the buffer. The viewer gets to see this higher bitrate only after all segments which are already collected in the buffer get played out. This problem becomes more pronounced with larger buffer sizes. Fast switching replaces segments which are already in the buffer, when bandwidth improves. First, it decides between downloading a new segment and a replacement segment (at a higher bitrate). Next, it downloads and replaces a candidate segment that is not too close to the current playout time (i.e., safely away from buffer queue head).

Fast switching allows DASH clients to efficiently compete against other clients in a greedy way. However, as we mentioned before, DASH is a client-side framework where each individual client is not aware of other clients' decisions. This often negatively affect fairness and can cause issues and network problems such as congestion or under-utilization of network links.

2.2.2 Quality of Experience

Quality of Experience (QoE) is the perceptual quality of a service from the end-users' perspective. By definition, video QoE is distinct from network health metrics, transport layer performance (e.g., TCP packet drops, round-trip time, jitter, etc.), bandwidth averages, etc. Ideally, it is desired to assess QoE subjective test, where human viewers evaluate the quality of test videos under a laboratory environment. However, *subjective* tests are costly and time consuming. It is also very challenging for a human observer to providing a QoE score for each individual frame or segment in the video under test.

Therefore, *objective* quality models have been developed to predict QoE, in terms of Mean Opinion Score (MOS), based on available objective parameters (and not QoS metrics). One of the well-accepted QoE models for adaptive streaming provides a prediction or estimation of MOS through a linear combination of the following three parameters:

$$QoE = \alpha\mu + \beta\sigma + \gamma\phi + \delta \tag{2.1}$$

where μ is the average quality of video segments, σ is the standard deviation of video qualities, ϕ is a measure of rebuffering or freezes, and α, β, γ, and δ are tunable

Impairment	Quality	MOS	User Satisfaction	QoE
Imperceptible	Excellent	5	Very satisfied	5
Perceptible but not annoying	Good	4		4.3
			satisfied	4
Slightly annoying	Fair	3	some users dissatisfied	3.6
			Many users dissatisfied	3.1
Annoying	Poor	2	Nearly all users dissatisfied	2.6
Very annoying	Bad	1	Not recommended	1

Fig. 2.3 MOS scale for subjective video quality assessment. Source [1]

parameters [7] that can reflect users' preference. Figure 2.3 shows MOS scale for subjective video quality assessment.

When multiple players share network resources, QoE metrics need to incorporate new factors in addition to the video quality. One important factor is QoE-fairness which can be measures using the well-known Jain's Index defined as follows:

$$J(\mathbf{x}) = \frac{\left[\sum_n x_n\right]^2}{N \sum_n x_n^2} \quad \in [\tfrac{1}{N}, 1] \tag{2.2}$$

where N is the number of video streaming sessions competing over the same network resources. Note that the QoE-fairness doesn't necessarily mean that all clients must receive the same video quality or experience the same amount of buffering time. This is because different users can be more/less sensitive to some of these issues compared to other users. The ultimate goal is to achieve QoE-fairness based on users' preferences as mentioned in Equation. 2.1

2.3 Deep Reinforcement Learning

The Industrial Revolution marks a major turning point in human history. It was the transition from hand production methods to new manufacturing processes using automated special-purpose machines that can *follow* human instructions to achieve specific tasks quickly, precisely, and cheaply compared to manual labour. Following the Digital Revolution, the early digital computers were not that different. Computers

are able to tackle and solve problems that are intellectually difficult for humans as long as they can be described precisely by a list instructions and formal mathematical rules. The true challenge proved to be in solving the tasks that are easy for people to perform but hard for them to describe formally; like recognizing spoken words.

Machine learning (ML) provided and alternative approach to tackle these tasks. Rather than providing the computer with a set of *instructions* on *how* to solve the task, ML provides it with a set of training *examples* demonstrating *what* the task is. In this setup, it is up to the machine to *learn* how to translate these examples into a mathematical formulas that can be used to process new unseen examples. A core topic in ML is that of *sequential decision-making*. This is the task of learning to decide, from past experience, the sequence of *actions* to perform in an uncertain *environment* in order to achieve some goals. Sequential decision-making tasks cover a wide range of possible applications with the potential to impact many domains, such as network automation, robotics, healthcare, smart grids, finance, self-driving cars, and many more.

Deep reinforcement learning (DRL), is a sub-field of ML focuses on teaching the computer how to take decisions under uncertainty. It consists of two sets of algorithms that are inspired by biological systems. The first set of algorithms focuses on modeling the *architecture* of the biological brain, which led to the development of Artificial Neural Networks (ANN). The second set of algorithms focuses on modeling the *behavior* of the brain during the learning process. This led to the development of many learning paradigms, of which, Supervised Learning (SL) and Reinforcement Learning (RL) are considered in this thesis.

In this section, we present the basic concepts of both SL and RL paradigms in parallel to better understand when, where, and how they can be used. In particular, we define the *task* each approach is trying to solve. Then, present the different *sources of information* available for the model to learn from. Next, we describe how to design a *model* that can represent certain policy (strategy) to perform the task. Finally, we compare how SL and RL *training algorithms* help the model minimizing a cost function that measures the performance of these policies.

2.3.1 The Task

Many practical problems require the computer to perform a mapping $g : X \to Y$ from the input space X to the output space Y. Unfortunately, in many cases it is difficult to manually specify the function g by conventional means. The SL paradigm offers an alternative approach that takes advantage of the fact that it is often relatively easy to

obtain examples $(x, y) \in X \times Y$ of the desired mapping. The term supervised learning originates from the view of the correct output (*labels*) being provided by an instructor or teacher who shows the model what to do. The model is required to analyze this training dataset and to tries to learn a mapping function f that best approximates the unknown function g. If the model is good enough, the function f should be able to determine the labels for unseen instances. This requires the model to *generalize* from the training data to unseen situations in a "reasonable" way. This is important because the dataset might be *biased* and might include *noisy examples* that the model should ignore. In RL, however, the true labels are not available. The only available feedback is a "reward" signal that tells the model how good or bad its decisions are. In this case, a typical RL problem can be seen as optimizing the unknown function g that generate these rewards.

Many kinds of tasks can be solved with SL. Some of the most common tasks include classification, regression, prediction, and anomaly detection. In *Classification*, the model is asked to specify which of k-categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathcal{R}^n \rightarrow \{1, ..., k\}$. When $y = f(x)$, the model assigns an input described by vector x to a category identified by numeric code y. The model can also produce a probability distribution over classes. In *Regression*, the model is asked to predict a numerical value given some input. *Prediction* tasks involve data that changes over time. In this task, the model is given a set of historical values and is asked to predict value(s) in the future. In *Anomaly detection*, the model sifts through historical values and flags some set of events as being unusual. Anomaly detection can be performed on time-series data, by comparing current value with previous values and decide if the new value follow the same pattern or not.

It is worth noting that, in practice, a SL model can be asked to solve tasks that involve multiple sub-tasks at the same time. For example, in network automation, the model can be asked to perform real-time anomaly detection and when a problem is detected, the model should select the proper action(s) to take from a list of possible actions (i.e., perform classification). It is also desired to be able to predict problems before they happen to avoid any service loss. In this case, regression can be used to produce a single number that represents the current "health score" of the network based on a set of performance metrics.

RL tackles problems where the solution is not known but rather a form of feedback is still available. For example, consider a task where the model has to learn a sequence of actions in the correct order to achieve a specific goal. In such a case, a single action

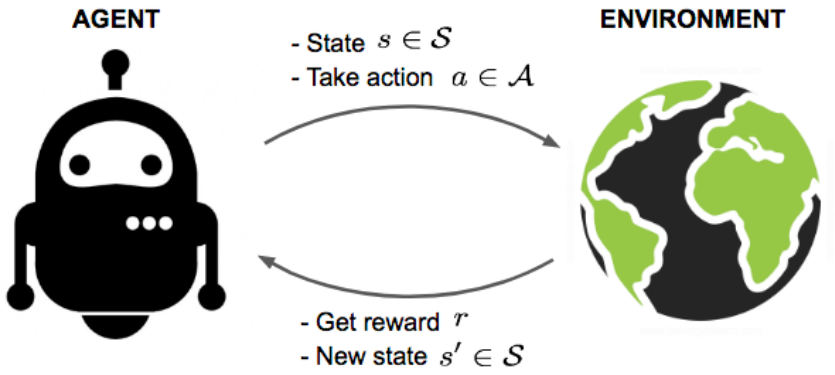


Fig. 2.4 The concept behind Reinforcement Learning

might not be useful or important; what is important is the *policy*. Unlike SL, which has to provide the model with a "good action" to take at each step, RL considers an action as good only if it is part of a "good policy". Thus, it generates a feedback signal that reflects how well a specific policy do in terms of achieving the ultimate goal. For example, to teach a robot how to walk we might not have access to the right amount of force a robot should apply on each joint at each point in time, but we can rate its overall walking skills and provide it with a reward (or a punishment) to encourage it to walk properly.

Planning is another example of tasks suitable for RL. In this type of problems, the goal is to maximize certain type of long-term future reward. Unlike SL, where there is no explicit concept of "agent" or "environment" (and their interaction), the basic mechanism behind RL is to plan to take good actions under uncertainty merely through trial and error. This interaction is formally described using mathematical framework called Markov Decision Process (MDP) which is defined as $\{state, action, reward, transition\}$ tuple. Figure 2.4 illustrates this basic idea. As we can see, an agent observes the *state* of the environment as an input and makes a choice about its *action*. However, rather than receiving the true action as a label to compare with, the RL agent receives some reward and a probabilistic transition to a new state. The transition function helps the RL model predicts how the environment would react in response to its action(s) which in turn is a very instrumental in learning how to plan to take better actions.

It can be observed that many problems can be cast as either SL or RL problems. Take network automation as an example:

- RL formulation (sequential decision-making problem): one can define a NN as agent model, setup a lab as an environment, design a reward function, and let

an RL agent interact with the environment to collect observations, take actions, and learn to plan for the future.

- SL formulation (time-series classification problem): one can also define a NN as agent model with memory (e.g., RNN), setup a lab and collect a set of observation, perform manual labeling (i.e. correct class/action for each observation), and let an DL model learn to predict the correct action from past observations.

It is worth noting that one can expect the SL model to learn more efficiently, compared to RL, because it has access to direct supervision. However, the RL model has an advantage of being able to explore; which can lead to better performance compared to human expert-rules.

2.3.2 The Data

As we mentioned earlier, the examples in SL are presented as a static dataset, where each example is represented as a tuple of {inputs, labels}. The model samples a "batch" of examples from this dataset over and over until it learns the underlying mapping between the inputs and the labels. RL on the other hand packages the examples inside an “environment”. In this setup, the agent is responsible for constructing its own "dataset" (known as episodes) by interacting with the environment. Each episode is represented by a set of $\{state, action, reward, next\ state\}$ tuple.

RL is different from SL in the type of guidance the model will receive. In SL, the model is presented with the true label that represents the solution to the task in hand. This is usually possible because many tasks can be solved easily by humans and thus the solution is known (e.g., face recognition). The challenge might still be in the feasibility of labeling enough amount of data since DNN tends to require huge datasets to perform well. RL tackles problems where the solution is not known but a form of feedback is still available. This form of *indirect* guidance makes training RL agents much harder than SL. RL can also be applicable when the feedback might be delayed. For example, in a game of chess, the agent might not receive a reward after each move but rather a single delayed reward at the end telling it whether it won or lost.

The quality of the input features heavily affects the performance of both SL and RL models. For example, when SL is used to decide whether to dismiss a patient from the hospital or not, the model does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information (called features). A model learns how each of these features correlates with various outcomes. However, it cannot influence the way that the features are defined in any way. Similarly, in RL, the agent

might not be able to fully observe the environment. If the observations were unreliable or incomplete, it will negatively affect the quality of its decisions.

Another issue that affects the performance of SL and RL models alike is the quality of the provided feedback. For example, wrong labels in SL or poorly-designed reward in RL can deceive the model and push it toward the wrong direction. Similarly, imbalanced data can result in a model that is biased towards a subset of actions/classes and is unable to generalize well. While the SL model can't do anything about imbalanced datasets, the RL agent is responsible for collecting its own dataset and has the power to avoid this problem during the training process. Since the state-action pair decides the next state, it is possible that the agent will keep landing in the same set of states because it keeps repeating the same set of actions. It would be desirable to visit diverse set of environment states and *explore* different actions before start to *exploit* the best learned actions. This phenomenon is described in literature as the exploration-exploitation dilemma.

2.3.3 The Learning Algorithm

As mentioned earlier, ML is a learning paradigm that utilizes the concept of learning from a set of examples. "Training" a model simply means determining good values for all its parameters from labeled examples. The *labels* can be explicit (ground truth) as in SL or implicit (reward signal) as in RL. In this section, we'll focus on training a subclass of ML models, namely NN. Training NNs involves working with three elements:

- **model parameters** (θ): are values that control the behavior of the model. They can be tuned, such that when used to transform the input features, they yield the correct output. In NN, those parameters includes the wights and biases of the NN model.
- **objective function** ($J(\theta)$): is a measure of model's performance at specific task. It can be desired to minimize this function (as for the *loss* in SL) or maximize it (as for the *reward* function in RL). The output of this function is almost always a single real-valued number.
- **learning signal** ($\nabla_\theta J(.)$): its an indication of how to change the model parameters in order to improve the objective function. The most commonly used signal is the *gradient* of the objective function with respect to the model parameters.

DL learning models are mostly trained using a famous algorithm called *Back-Propagation Algorithm* (BPA). This algorithm uses the chain-rule to efficiently compute

the gradient of the objective function with respect to NN parameters. The BPA is only applicable if the objective function is differentiable. This is typically the case for SL assuming the activation function is differentiable. However, this is not the case for RL. When the reward function is not differentiable, a *policy gradient* algorithm (PGA) is used to approximate the gradients $\nabla_\theta J(\pi_\theta)$. This process can be noisy and less efficient compared to SL case. However, the ability to optimize non-differentiable objective function is a big plus for RL.

Note that it is common to write the loss function in DL in terms of NN parameters as $J(\theta)$, while in RL as $J(\pi_\theta)$, where π_θ is simply the NN responsible for taking the actions. Note also that once the gradient is calculated (using BPA or PGA), another algorithm, such as *stochastic gradient descent*, is used to perform the actual optimization (i.e., tuning the model parameters given the calculated gradient).

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\pi_\theta) \tag{2.3}$$

where η is the learning rate, $\pi_\theta = \theta$ for SL, and $J = -R$ for RL (the negative sign to emphasise that RL agent *maximizes* the reward)

In order to evaluate the quality of a learning algorithm, it is common to plot the *learning curves* as an indicator of system performance. Learning curves show the model performance over time [1] measured by an objective function. In SL, the most commonly used function is the loss function $J(\theta)$. Other evaluation metrics such as accuracy, precision, and recall can also be used. In RL, the loss function is not useful for two main reasons (thus the reward signal is used instead):

1. Typically, the loss function is defined on a fixed data distribution which is *independent* of the parameters we aim to optimize. This is not the case with RL since the data must be collected (sampled) by the agent itself using its most recent policy.

2. For the loss function to be useful, it should evaluate the performance metric that we care about. In RL, we care about the long term expected reward, but the loss function does not approximate this objective at all. In fact, it is common to use more than one model during the training (e.g., *off-policy* algorithms [33]) where one NN is used to take actions while another model is used for learning. As a result, the loss function can be useful indicator of the model performance only when evaluated at the current parameters, with data generated by the current

[1] In practice, time is measured in "steps" not seconds. A step can be *batch* index or *epoch* number in SL. Similarly, it can be an *experience* index or *episode* number in RL.

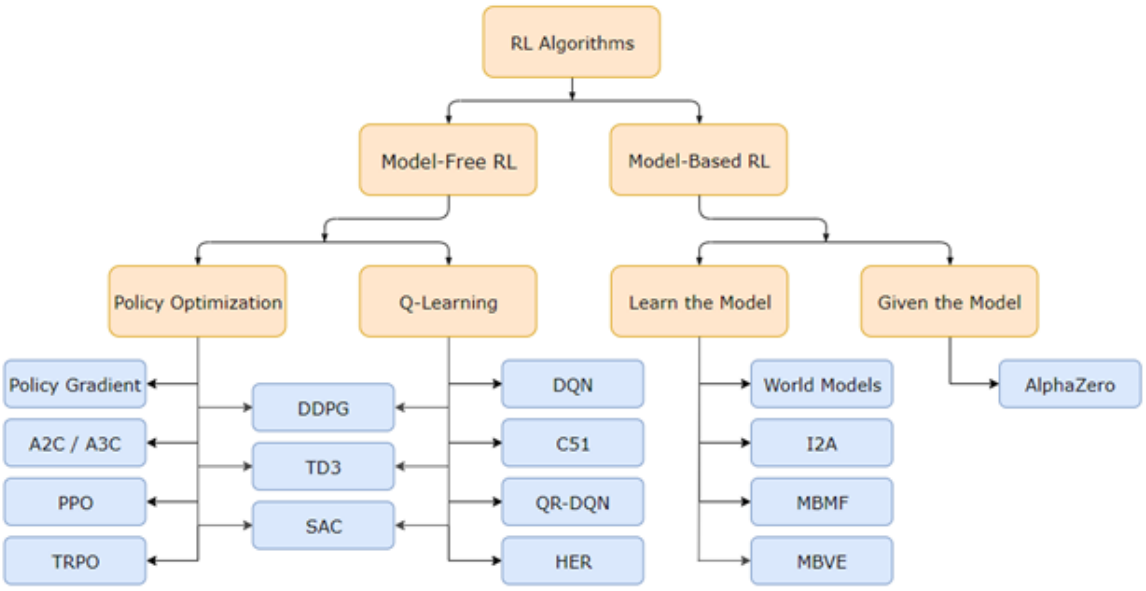


Fig. 2.5 Taxonomy of RL algorithms. [source: spinningup.openai.com]

parameters, to generate the negative gradient of the performance. However, after that first step of gradient descent, there is no more connection to the performance.

RL algorithms can be generally divided based on the *training algorithms* into Model-based and Model-free algorithms (see Figure 2.5). The agent in the former approach tries to model the environment purely from experience. The biggest challenge in this approach goes beyond being expensive and complicated; it is that any bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model but fail in the real environment. For that, Model-free algorithms are more popular.

Model-free algorithms can be further divided into *Q-based* algorithms and *Policy-based* algorithms. Double-DQN and A2C as two famous algorithms that represent those two families respectively. They will be explained in §4.2.3.

2.4 Network Science

Network science is an academic field which studies complex networks such as telecommunication networks, computer networks, and social networks. The field draws on theories and methods including graph theory from mathematics, statistical mechanics from physics, data mining and information visualization from computer science, and

others. The section review the main concepts of network science that are related to telecommunication and computer networks.

A graph is a mathematical structure consisting of a set of vertices (or nodes) and a set of edges that connect the vertices. The vertices in the graph represent the devices in the network, such as servers, routers, and clients. The network devices interact by generating, forwarding, or consuming traffic. These interaction can happen at different rates creating various traffic patterns in the network.

The edges in the graph represent the connections (links) between these devices, and is associated with a set of properties such as:

- capacity: is the maximum amount of data that can be transmitted on the link per unit time.
- utilization: is the actual amount of data that is being transmitted on the link at a given time. If the amount of traffic transmitted through the link exceeded its capacity, the link suffers a congestion problem causing packets to be lost (discarded by the received end).
- cost (distance or weight): is a measure of the cost associated with using a particular link in the network. Different metric can be used to represent the link cost such as monetary cost, latency, or energy consumption. A graph that include edges with associated costs are called weighted graphs.
- direction: links can be bidirectional allowing traffic to pass in both directions with equal cost, or unidirectional where traffic can pass in one direction only. If all links are of the former type, the graph is referred to as undirected graph, otherwise the graph is directed.

Often, networks have certain attributes that can be calculated to analyze the properties and characteristics of the network. Some of these attributes are computed as an average of the attributes of the individual vertices and edges. Others are path related, which depicts the concept of tunnels in telecommunication and computer networks. For a simple undirected graph with N vertices and M edges, it is possible to calculate the following properties:

- Network size: is the number of nodes in a network.
- Network density: is a measure of how many connections exist in a network compared to the total number of possible connections. It is given by: $2M/(N * (N-1))$.

- Network average degree: is the average number of connections per node. It can be calculated by $2M/N$. One of the most exciting discoveries of network science is the fact that, in many real networks, the node degree follows a power law distribution. This implies that, while most of the nodes in the network have only a small number of connections, there is a handful of them, called hubs, that are connected to a very large portion of the network.

- Network diameter: is the longest shortest path between any two nodes in the network. This represents the worst case scenario that a packet can experience when travelled through the network.

- Characteristic path length: is representative of the linear size of a network and is calculated as the average shortest path length between all pairs of nodes in the network. Note that The set of all shortest paths define a distribution of path lengths.

- Network clustering coefficient: is the likelihood of a link existing between two arbitrary neighbors of the same node. It can be calculated as the average of clustering coefficient of all nodes. Where the clustering coefficient of a node is the ratio of existing links connecting a node's neighbors to each other to the maximum possible number of such links.

It is worth noting the in real networks, one or more network properties can change over time (a.k.a., represent random variables). For example, network topology can evolve over time as new nodes and links are added. Furthermore, the cost (delay) of using a certain path (tunnel) in a network during peak period can be much higher than other times of the day. This type of graphs is called stochastic graph (or random network).

Chapter 3

Related Work

In this chapter, we review the related algorithms for NOC. We group the most recent related work in this area into three groups based on how actions are defined: (i) traditional methods that uses hand-crafted expert-rules, (ii) ML-based methods, and (iii) Other trends to NOC but are not the same. We highlight the differences and similarities between our proposed algorithm and related works.

3.1 Traditional Methods

Traditionally, NOCs perform their tasks either manually or, for the simplest tasks, with pre-determined expert rules. These approaches are often based on *network redundancy* where multiple nodes, links, or tunnels are defined to insure a backup path is available when the primary path fails. For example, [29] proposes a fast failure recovery approach based on active probing mechanism. The idea is manually select a subset of nodes as monitoring components which expect to receive at least a certain number of probe packets in a predefined interval; if not, it assumes the link is failed and the restoration phase begins. The main problem of this approach is that it requires meticulous design because any flaw in this component dramatically increases false positive or false negative in failure detection phase.

Another example uses handcrafted rules to achieve fast recovery for OpenFlow networks by actively using backup and primary paths before and after failures to achieve high utilization [19]. In this case, recovery paths are adaptively updated based on the current state of network's load. By providing the backup forwarding rules in advance, fast recovery can be achieved upon a failure. Similarly, the work in [39] adopts segment routing in order to reduce the number of forwarding rules and deal with link failure problem in Software-defined Networks (SDNs). This was done by regarding the

affected flows, through the same link, as an aggregated flow and establish a backup path for the aggregated flow.

While the above methods can help automate NOC, their success is limited because of the intrinsic difficulty in defining expert rules that robustly work in complex and dynamic networks. The authors in [14] found that when a failure occurs, approaches based on network redundancy are only 40% effective in reducing the median impact of the failure.

In this work, we show that it is still possible to utilize the expert rules in a meaningful way. Instead of using the expert-rules to automate NOC, it is possible to pre-train an RL-agent using sub-optimal actions collected by applying the expert-rules in a simulated environment or from historical data. Then allow the RL-agent to explore new policies and improve.

3.2 ML-based Algorithms

Utilizing ML for root cause analysis (RCA) became popular in recent years [31]. The authors in [10] proposed a ML-based RCA algorithms that detects and localizes network problems when a degradation in users' QoE occurs. It does so by placing measurement probes at multiple vantage points along the path, training a supervised ML model on a combination of synthetic and real-world measurements, and using the trained model to diagnose network issues in the wild with around 80% diagnosis accuracy.

One of the main challenges in applying ML for RCA is that ML algorithms are mostly not explainable. The authors in [15] propose a method to include the network operators in the loop. The main idea is to apply ML algorithms on historic events that happened in the network in order to automatically identify dependencies between system events. The output is enhanced with summarization, operations on graphs, and visualization that help network operators identify the root causes of errors.

But, as interesting as these methods are, they only detect the network fault and do not take autonomous actions to fix it. In contrary, the authors in [8] uses raw measurement data and logs to reveal "symptoms"; i.e., degradation conditions, and then applies ML to learn a mapping between these symptoms and remedial actions. It uses 3 months of data collected every 15 minutes from 1800 node, with each sample having 1100 feature, so it has the advantage of being wide scale. On the negative side, the only action it recommends is whether to reboot an interface or not, it only achieves an accuracy of about 40% compared to expert rules, training requires extensive computing resources and time, and any reconfiguration in the network requires retraining.

3.3 Other Approaches

In this thesis, we focus on automating NOC using RL. However, it is also worth mentioning other related trends that help achieve the *adaptive network* vision discussed in §2.1.

Network Routing

Network routing has become increasingly popular in recent years [9] [25] [42]. Routing is the decision making that directs data packets from their source nodes toward their destination nodes through some intermediate nodes. It is different that NOC in the sense that the goal of NOC is to maintain optimal network performance and availability, and to ensure continuous uptime. The actions taken by NOC can include routing traffic (via backup tunnels, change administrative distance, define a static-route, etc.), but it go beyond that. Also, and when routing is applied, NOC are not concerned with deciding the next hop of each individual packet nor optimizing next-hop delay for individual packets.

An example of applying RL in network routing was proposed in [41]. The authors formulated the network routing problem as a POMDP and regarded each router as an independent agent. In this setup, the state is the destination of the first packet in its the queue and the action is one of its outgoing links. Each agent maintains a Q-table, $Q(s, a)$, to estimate the expected time-to-arrival that it takes to send a packet from the current router to the destination s by way of an outgoing link a. RL is applied to optimize the policy of each agent in a distributive way using a global reward. Provided a global feedback signal, the agents act independently but are able to learn cooperative behavior through limited information exchange.

Network routing is different from our work in a sense it requires modifications on the routers to implement these new algorithms. In contrast, our focus is on optimizing the end-to-end experience in the overlay network to maintain high QoE for the end-to-end connections.

Network Slicing & Zero-touch Networks

Network slicing permits the creation of a chain of network functions that logically conform to a dedicated virtual network that fulfills certain requirements. This allows the ISPs to segment the huge amount of data traffic coming through their networks and deal with each segment individually and independently. As a results, ISPs enjoy flexibility in terms adjusting the cost for each slice and can apply individual control

and policy management systems as needed. For example, a QoS-aware network slicing framework for media-centric use cases, where demanding QoS requirements, especially low latency, are expected was introduced in [38].

The inclusion of ML in network slicing enables ISPs to analyze any unknowns and take necessary corrective actions. For example, the authors in [34] applied DL to efficiently and cost effectively apply network load balancing and network slice failure recovery. The DL model triggers automation in the network to modify available resources and make changes on the go and is responsible not only to provide and process raw data, but also make an intelligent decision for network resource adaptation without any human intervention.

While network slicing permits the creation of a chain of network functions that logically conform to a dedicated virtual network that fulfills certain requirements, *Zero-touch network* provides methods to achieve these goals in an unsupervised manner. The main idea lies in using ML to automatically orchestrate and manage network resources while assuring the QoE demanded by users [12]. The authors in [13] examined how ML can be used to address the challenges that SDN and NFV present, specifically highlighting the issues caused by having limited access to available network resources. Similarly, the authors in [22] reviewed the RL-based resource management schemes for 5G heterogeneous networks in various domains, and discussed possible solutions to overcome the complex problem caused by the interference between small and macro cells.

Unlike these contributions, our work targets ISPs running traditional networks infrastructures that does not support SDNs and Network Function Virtualization (NFV) capabilities that can be found in more recent systems such as in 5G networks.

Chapter 4

System Design

Network operation centers control their networks using expert rules, which are a set of predefined instructions that specify how to handle different network situations. These rules are based on the knowledge and experience of network engineers and are manually configured on the networking devices. However, the increasing complexity of networks and the ever-changing nature of network traffic make it challenging for expert rules to keep up. This motivates the idea of automating expert rules using ML.

This chapter explains the design of our ML-based ARE. In §4.1, we describe the testbed setup that is used to run our various algorithms. In Section 4.2, we formulate the ARE problem in a RL framework. Our goal is to use the ARE to discover new policies that can lead to superhuman performance. To achieve this, we leverage offline training to safely explore new policies while avoiding actions that could be costly in real-world environments. We also draw parallels between RL and SL, and discuss the advantages and disadvantages of each approach.

4.1 Network Topology

The quality of training data is crucial for the performance of ML algorithms, as they learn from demonstration. In many cases, it is not feasible to train ML algorithms directly in real-world environments. For instance, DL models are often trained on synthetic datasets and RL agents are typically trained in simulated or laboratory settings. Nevertheless, these models are expected to perform well when deployed in actual environments. Thus, it is essential to ensure that the training data accurately represents real-world scenarios to enable the ML models to generalize well.

This section designs two network topologies: a small-scale network and a scaled-up version. The small-scale network is designed to aid in the analysis of different algorithms against a set of predefined expert rules. The small size of the network makes it feasible to define a nearly-optimal expert rules to compare against. The large-scale network, on the other hand, is used to test the performance of different algorithms in a more realistic setup. As the large network is more complex in nature, we expect the expert rules that we defined to be suboptimal. In this regard, the large network helps evaluate the ability of the different algorithms to learn from the suboptimal labels/actions and the extent of improvement that RL can provide over these hand-crafted rules.

Note that the two networks proposed here can be simulated, emulated, or implemented in the lab. We'll discuss the implementation details in §4.2.2.

4.1.1 The Small Network

In this section, we design a small network topology to help design the network automation algorithm. To aid our design, we used a small network that is neither too simple such that it doesn't sufficiently reflect the real world nor too complicated such that it's difficult to determine a good set of expert rules to automate the network, and potentially use the expert rules as a baseline to evaluate the performance of the ML models.

Following the definition in [11], a network G is called a basic network if it has the following properties: (i) it has at most one source and one sink. (ii) each edge e_i of G has associated positive capacity $C(e_i)$. The traffic flow in such network from the source to the sink satisfies the following two conditions: Feasibility and flow conservation law. The former condition asserts that the transmission rate along any edge e_i can't exceed its capacity. The later condition asserts that the total flow out of the source is equal to the total flow into the sink. A set of candidate topologies are shown in Fig. 4.1.

We chose network A to study the behaviour of our ML algorithms. In this network, a video client (denoted by ⊜) is connected to a video server (denoted by ⊛) through a network consists of five routers (denoted by ®). The network contains three tunnels: the red path (⊛-①-②-⑤-⊜), green path (⊛-①-③-②-⑤-⊜), and yellow path (⊛-①-③-④-⑤-⊜). The traffic can be assigned to a specific tunnel via an access control list (ACL) defined on both ends of the tunnels (routers ① and ⑤).

We used video traffic because of two reasons. Firstly, video is, by far, the dominant IP traffic and is expected to constitute 82% of all IP traffic by 2022 [6]. Secondly, video traffic is inelastic and bulky with a sigmoidal QoE function, which makes it very challenging to optimize users' QoE [17]. As such, it represents the worst-case

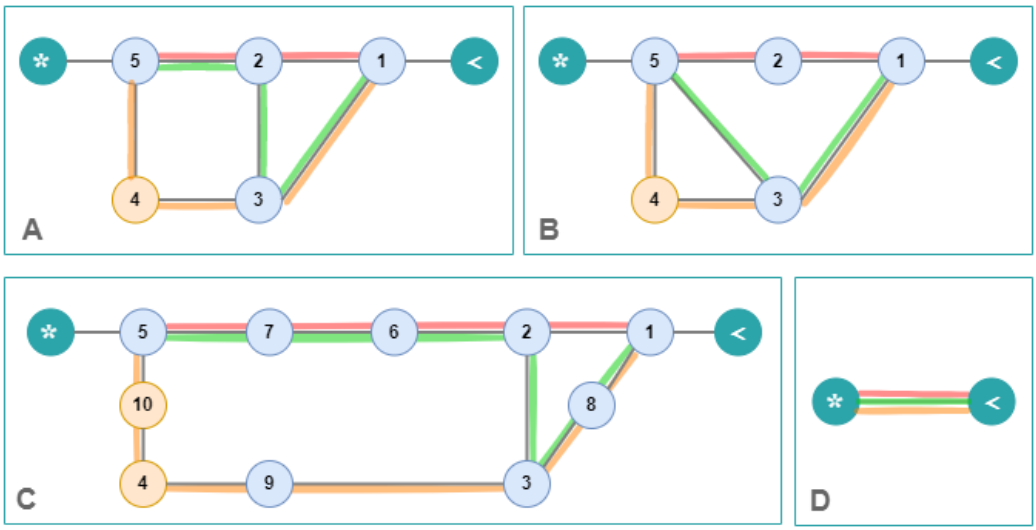


Fig. 4.1 (A,B,C) Basic network setup with clients denoted by ⊜, routers denoted by Ⓡ, and server denoted by ⊛, (B, C) Examples of network variations (D) Corresponding overlay network.

scenario. Two environments were implemented in §4.2.2 using this proposed small network. The first using GNS3 emulator and the second using a probabilistic-based synthetic data generation simulator. One difference from the proposed topology here is that we allowed more than one client to stream video through the network (i.e., multiple flows). However, to keep the design as close to the basic network as possible, we force all clients to be connected to the same router (namely router ①). This has the advantage of allowing the ML agent to aggregate all flows into a single flow as long as the total bitrate from all flows doesn't violate the feasibility principle.

The video server divides the streaming media into short segments, such as two seconds, and store them at multiple bitrates. This approach has several advantages. Firstly, using multiple bitrates allows clients to adaptively request video segments with the best quality possible without causing buffering events. A buffering event occurs when attempting to stream video segments that are larger than the available bandwidth. This can happen when multiple streaming sessions are competing for the limited bandwidth of shared links along the path to the server. As a result, the playback buffer of video clients is depleted at a faster rate than it is being filled, causing the video players to freeze.

Secondly, using short segments allows clients to quickly adapt to changes in network conditions. This can be done by observing the playback buffer level and the average time needed to download the last few segments. These metrics provide an indication

of possible congestion in the network. In such cases, the client can quickly switch to a lower bitrate when downloading the next segment, thus improving the overall streaming experience. Note that if the network conditions improved while the playback buffer of a client is full with low quality video segments, the client can discard some of these segments and download them again at higher bitrate. This process is refereed to as fast-switching in DASH video streaming.

Regarding the traffic patterns, we allowed the clients to connect/disconnect from the network at random times. It is common to model the inter-arrival time (IAT) of video clients using Poisson process. In a Poisson process, the inter-arrival time between events follows an exponential distribution IAT $\sim \mathcal{E}xp(\lambda)$ where the clients are assumed to connect to the server independently from each other at certain rate λ. Estimating the duration of the streaming session is an interesting research problem that has been actively studied in recent years [36]. However, we chose a uniform distribution for simplicity.

We used OSPF [28] as the underlying routing protocol and IP/MPLS to define tunnels (paths) connecting clients to the service provider. The use of overlay network in network automation has many advantages. Firstly, While traditional routing protocols such as OSPF send packets on the shortest path from source to destination, there are many recognized benefits of using multiple non-shortest paths [16], These benefits include failure protection, such as routing around link or node failures, and traffic engineering, such as distributing the traffic load to avoid congestion.

Secondly, it is possible to easily achieve the maximum flow in basic network when using MPLS tunnels even when multiple flows are presented as long as they can be aggregated into one flow. In the case of traditional hop-by-hop routing, the maximum flow cannot always be achieved because the shortest path algorithm computes a single path between a source and a destination. On contrary, the explicit routed MPLS label-switched path allows for traffic disaggregation and therefore the upper bound traffic can be achieved [11].

Finally, when optimizing network performance, it is desired to maximize the end-to-end users' QoE as opposed to being limited to QoS. This can be easily achieved by the overlay network because the traffic is guaranteed to follow a predefined path. Thus, it is possible to guarantee a certain level of performance defined by the service level agreement (SLA).

During a streaming session, we consider two types of problems that can negatively affect the clients' QoE: 1) failure in a router or a link and 2) network congestion. Despite being common, the first problem is not related to the traffic-engineering done

by the network operator. It is due to hardware issues related to the health of physical devices. On the other hand, congestion can happen due to sub-optimal traffic planning were clients demands exceed link capacity. Formally speaking, congestion can be defined as:

$$C_g = min\left\{\frac{r_x + \sum_n r_n}{C_b}\ , 1\right\} \tag{4.1}$$

where r_x, r_n are background traffic and video traffic respectively. C_b is channel capacity.

When network issues are introduced, we consider three types of problems. The study if these types helps up design a more relastic training and testing conditions, as well as realistic expert rules. These types are *persistent*, *transient*, and *recurrent* problems. We define a persistent problem to last for long time period. As a result, we expect the agent to proactively try to fix these issues. Transient problems are resolved automatically within a short time period. Therefore, it might be desired to ignore them in some cases to save unnecessary cost of attempting to fix them. Recurrent problems occurs every certain timesteps. We expect the a trained ML agent to be able to predict them before happening in the future and take some remedial actions to minimize the negative effect of these issues.

Regarding the collected metrics, the ideal scenario would be to have performance indicators from each network device along the data path. However in practice this is not feasible as it involves the cooperation of too many parties (the user, the ISP, and the content provider) and hardware vendors, let alone the amount of overhead it can cause. Therefore, in our approach we focus on capturing the end-to-end metrics. This includes: the application layer, and transport layer. More on how these metrics are used to train our ML models is discussed in §4.2.1.

On the application layer at the video client side, we record statistics concerning the QoE of video play-back. These metrics include the video start-up delay, video stalls, frame skips, the status of the buffer, and video bitrates. These metrics are used to construct an estimated Mean Opinion Score that represents the QoE ground-truth. Notice that while these metrics can indicate the existence of poor QoE, we are not including all of them as inputs to the ML models. They are mostly used to generate an accurate QoE metric to evaluate the performance of the ML models.

On the transport layer, a set of end-to-end network metrics are collected per flow including: RTT, jitter, packetloss, path-length (in hops), number of clients (flows) in the tunnel, number of packets, and flow duration.

4.1.2 The large Network

The proposed network topology in section 4.1.1 represents a realistic building block for larger networks. In this section, we explore methods for scaling up the network to include multiple servers, routers, and a large number of clients. This proposed setup is implemented using a discrete event simulator, as described in section 4.2.2, and serves as a testbed to evaluate the robustness and scalability of our ML models.

In essence, this graph generation problem translates the search problem of finding a graph or set of graphs conforming to a set of potentially interdependent properties within the search space of all potential graphs. Essentially, our graph generator algorithm must be able to generate graphs that closely resemble the characteristics of real-world ISP networks rather than simply generating random graphs.

There are different approaches to synthetically generate graphs with certain properties. One approach is to use *evolutionary algorithms* [37] to gradually add more nodes and edges to the network, simulating a natural growth in the number of devices and connections. This approach allows for a gradual increase in complexity and allows to study how the ML algorithm adapts to the changes.

An evolutionary computation graph algorithm typically follows the following steps to generate a large graph from a small one:

1. *Initialization*: The algorithm starts by creating an initial population of small graphs, usually generated randomly or by using an existing graph as a starting point.

2. Selection The algorithm then selects a subset of the current population of graphs that will be used to create the next generation of graphs. Selection is typically done using a fitness function that assigns a score to each graph based on how well it conforms to the set of desired properties.

3. *Crossover*: The selected graphs are then combined to form new graphs through a process known as crossover. Crossover typically involves randomly selecting edges or nodes from each parent graph to form the new graph.

4. *Mutation*: The new graphs are then subject to random mutations, which involve small changes to their structure such as adding or removing edges or nodes.

5. *Evaluation*: The new generation of graphs is then evaluated using the fitness function and the process is repeated until a graph that meets the desired properties is found or a certain stopping criterion is reached.

Since evolutionary algorithms are stochastic, an important point of concern is the time it takes to converge to a set of acceptable graphs. For example, if the graph is represented in terms of connectivity matrix, it is possible to achieve fine-grain control over the generated topology at the expense of slow convergence speed. On the other hand, if a more efficient representation is used such as defining the generating function, the convergence speed improves but the fine-grain control become limited.

Another approach is to use *generative models*, where an algorithm is given a specific graph and is asked to generate variations of the graph that preserves certain properties such as graph density. Traditional approaches to graph generation focus on various families of random graph models, which typically formalize a simple stochastic generation process. However, due to their simplicity and hand-crafted nature, these random graph models generally have limited capacity to model complex dependencies and are only capable of modeling a few statistical properties of graphs.

More recently, building graph generative models using neural networks has attracted increasing attention [23] [40]. Compared to traditional random graph models, these deep generative models have greater capacity to learn structural information from data and can model graphs with complicated topology and constrained structural properties.

There are generally two approaches to generating synthetic graph based on the type of NN architecture used. The first approach uses an encoder-decoder architectures (such as variational auto-encoders), the encoder network extract useful features about the reference graph, and the decoder network is used to construct a new similar graph in one shot by defining the individual entries in the graph adjacency matrix (i.e., edges). This approach makes the models efficient and generally parallelizable but can seriously compromise the quality of the generated graphs.

The second approach use recurrent networks to model the graph generation as a sequential process. In this approach, the NN starts with a small network and sequentially add nodes and edges, or small graph structures to grow the network to the desired size. One advantage of this this approach is the ability to accommodate for complex dependencies between generated edges.

While generative models based on deep NN is a very powerful tool to generate synthetic graphs, we decided to use a more traditional approach because the process of training these NN models is expensive and require constructing large dataset of network topologies, which is beyond the scope of this work.

We chose to implement a third approach for generating large graphs, where a set of small sub-networks act as building blocks. By randomly sampling these sub-networks and connecting them together, we can construct a large network with specific properties

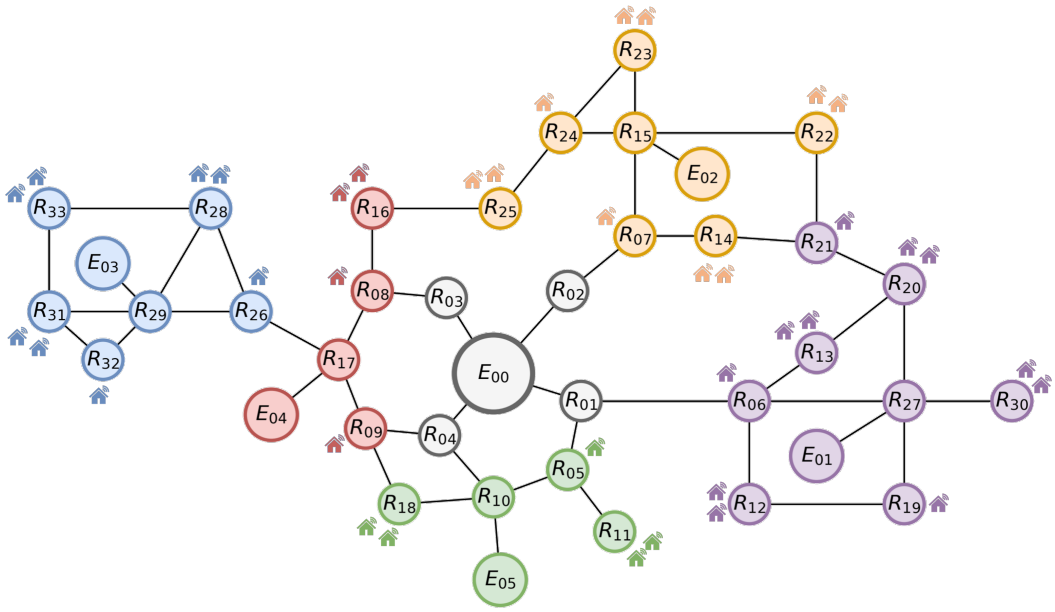


Fig. 4.2 A candidate large network. (E_{00}) represents the cloud, (E_i) are the video servers, and (R_j) the are routers. Variable number of clients (denoted as 🏠). Colors represents different autonomous systems.

such as degree distribution or clustering coefficient. This approach allows for greater control over the properties of the small sub-networks, and thus the generated large network.

One of the main reasons for using this approach was our access to a real ISP network topology. We aimed to study the structure of this network and create synthetic graphs that closely resemble its characteristics. As an example, Figure 4.2 illustrates a candidate network which we simulated using a discrete event simulator, as described in §4.2.2. Additionally, Figure 4.3 depicts a sample of the small sub-networks that can be extracted from this network.

Compared with basic setup, the new topology consists of:

- five edge servers capable of serving a limited number of clients. It is cheaper to use edge servers as they are closer to the clients. However, it is not possible to serve all clients in the neighborhood due to their limited capacity.

- a datacenter (cloud), depicted as a video server that has unlimited capacity. It serves as a default video source when local edge server gets overloaded. However, since the datacenter is far from many clients, streaming videos from this server should be minimized to avoid causing network congestion.

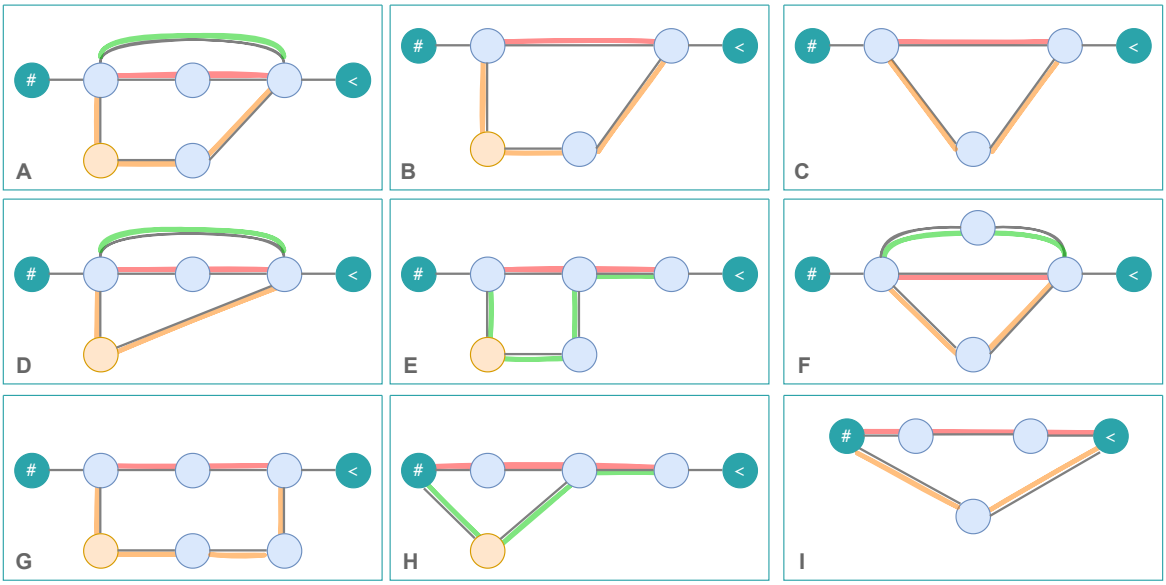


Fig. 4.3 Random subset of tunnel groups extracted from the big network.

- thirty three routers of which twenty-four act as gateways to video clients. For each router, a tunnel group is created and an RL agent is assigned to control this tunnel group. Each tunnel group consists of two main tunnels: one connects the router to the nearest edge server, and the other connects the router to the datacenter through the shortest path. Each one of those two tunnels has a backup tunnel. We chose the backup tunnels such that the overlap with the original tunnels is less than 50%.

- Large number of clients connect to the network at random point in time via the routers. A client streams videos at random bitrates (defined by their adaptation algorithm). We use BOLA algorithm for this purpose [32].

It is worth noting that we chose to setup the backup tunnels with 50% overlap rule to ensure the robustness of the network. If a link is shared among many different tunnels, its removal could cause a very important degradation of the network connectivity. Therefore, when designing the network, our strategy is to ensure that most of the nodes are connected by a set of paths with none or very few nodes in common, thereby boosting the robustness of the whole network and allow the design of expert-rules that protect, maintain, and recover the network from any failure.

Another design choice we made in designing the large network in figure 4.2 is that a portion of the network routers don't act as gateways and have no video clients connected to them. This is to reflect the multi-layer hierarchy in the design of ISP

networks where some routers (in the core and aggregation layers) don't have clients associated with them. The clients are typically served by routers in the access layer of the hierarchy.

In summary, this section introduced a scaled-up version of our basic network introduced in §4.1.1. Compared with basic setup, the new topology have 6 times more servers, 5 times more routers, 8 times more links, 33 times more tunnels, and 200 times more Clients. A method was also proposed to break the network into smaller tunnel groups that are topology agnostic. This approach ensures that no agent is responsible for more than its tunnel group. As a result, the algorithm is scalable to arbitrary network sizes. Also, since these tunnel groups are very similar to the basic network used during the training (except for number of hops which can be fed as part of the input data), the ML agents are expected to perform well on different big networks.

4.2 The Design of ARE

In this section, we propose an OPEX-aware ARE, shown as such in Figure 4.4, using RL, with the goal of outperforming humans by learning new effective rules on its own. ARE uses raw data, tickets, and feedback from its previously recommended actions to either recommend actions to the NOC technicians, shown in the figure with a blue arrow labeled "action recommendation", or directly apply the recommended action in a human-out-of-the-loop fashion, shown in the figure with a blue arrow labeled "autonomous action". In the context of automation, this superhuman performance will lead to an automation system with unprecedented efficiency and optimality in NOCs: efficiency because ARE makes decisions much faster than humans, and optimality because ARE's decisions will lead to much better results, in terms of cost saving and user's quality of experience, compared to human decisions.

4.2.1 Problem Formulation

In response to network problems, ARE must suggest/take actions to achieve and maintain an optimal goal set by the operator. In our case, the goal is to maximize clients' QoE and minimize the ISP's OPEX. We formulate this problem as a decentralized and partially observable Markov Decision Process (MDP) and solve it using RL. We chose RL for 3 reasons: its recent achievements in reaching superhuman performance in computer games, the difficulty of labeling data (needed by other ML methods, say supervised learning) in a complex network because the best actions are not always

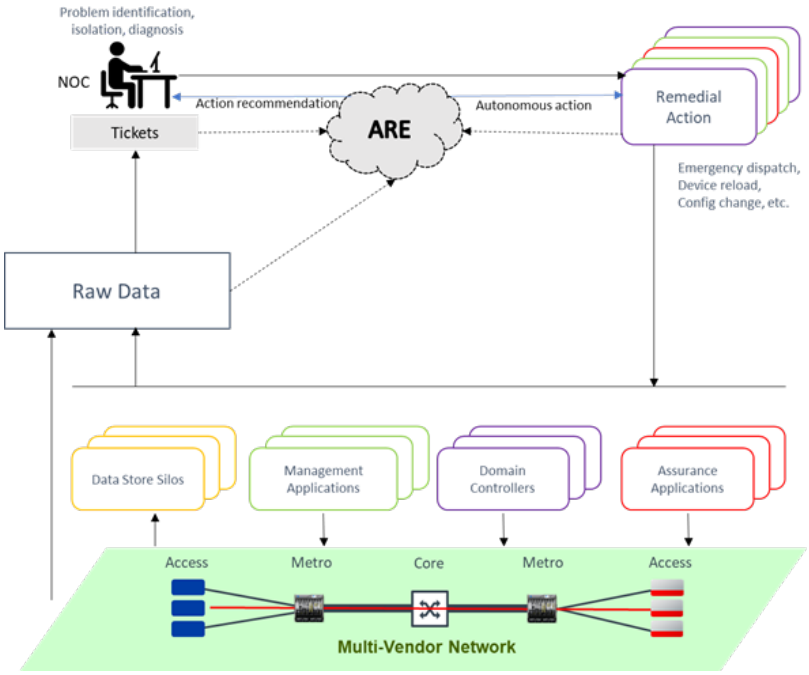


Fig. 4.4 ARE in a typical Network Operation Center (NOC)

known, and because we can satisfy the preconditions of using RL − quantifying a reward function, accessing the environment's variables, and affording RL making mistakes as it's exploring. The process is specified by the RL components $\{\mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{T}_s, \mathcal{T}_o, \mathcal{R}, \gamma\}$ defined as follows:

States $\mathcal{S}$ and State Transitions $\mathcal{T}_s$: our states are set to be the unobservable condition of each link and router within the network, described as *normal*, *congested link*, *broken link*, or *broken router*. The states represent the ground-truth of the root-cause issues in the network. We use the true environment state for two purposes: to calculate the rewards and to serve as an input to the expert rules. $\mathcal{T}_s$ is the state transition function that provides $P(s'|s, a)$, the probability of transition to a next state s' given that the agent starts at state s and takes action a. The Simulator Environment is designed around the idea of trying to mimic this function (see §4.2.2).

Observations $\mathcal{O}$ and Observation Function $\mathcal{T}_o$: the observations are the set of metrics available to the RL agent to infer the unobserved state of the environment (see §4.3). $\mathcal{T}_o$ specifies the probability $P(o'|s', a)$ that the agent will receive an observation o' of state s' after reaching this state through an action a. We use this function to perform the data augmentation step in the Simulator Environment (see §4.2.2).

Actions $\mathcal{A}$: the action taken by an agent at a given timestep t is: *Do Nothing*, *Fix a Link*, or *Reroute* traffic.

Note that rerouting traffic requires specifying two pieces of information. First, the agent needs to select the source and destination tunnels from which the traffic is rerouted. Next, the agent needs to select the client(s) that need to be moved. This means the number of possible actions under the "reroute" label grows with the network size and the number of clients. To simplify our formulation, we allowed the agent to specify the source and destination paths, but we don't let the agent specify which client from the source path to reroute; we simply select a client randomly. This means we don't differentiate between different clients. However, in practice, it's possible to select clients based on a criterion that takes into account the type of traffic (e.g., avoid delay-sensitive applications), traffic load (e.g., target problematic clients), and clients' service class (e.g., improve QoE of premium customers first). Grouping the clients in this way ensures that the number of actions is bounded and the solution to the RL problem is feasible.

It is worth noting that when dealing with large scale networks, it might not be feasible to operate on the link-level using E2E metrics only. To deal with this issue, we propose either including metrics from each hop (a.k.a at port level), or limit the action-space to action that are topology agnostic. We will follow the second approach in our analysis of the big network.

Reward $\mathcal{R}$ and discount factor γ: The reward signal is defined to achieve two goals. On one hand, we would like to maximize the overall QoE of all users. We map the collected QoE to a reward using the function $\Phi(.)$ which assign $+5, 0$, and -5 for high, medium, and low QoE, respectively. On the other hand, we want to minimize the OPEX. We include three main expenses in the OPEX term: the cost of performing an action, the hop-count cost of carrying traffic within the clients' ISP (AS1), and the cost of carrying traffic through the external backup ISP (AS3) which is more expensive because the backup ISP will charge extra. Therefore, the gain metric $\mathcal{G}$ (which we will call QoE-OPEX) to be maximized can be calculated as:

$$\mathcal{G} = \Phi(QoE) - \text{OPEX} \tag{4.2}$$

We adopt the well-accepted QoE model in [7] for adaptive video streaming. The QoE is defined as

$$\begin{aligned} \textbf{QoE} = \alpha \sum_k \frac{q_k}{K} - \beta \sum_k \left| q_{k+1} - q_k \right| \\ - \gamma \sum_k \left(\frac{d(q_k)}{C_k} - B_k \right)^+ - \delta \end{aligned} \tag{4.3}$$

Where q_k is bitrate of k^{th} segment and $d(q_k)$ is its size, B_k is the current buffer level. To define low, medium, and high QoE, we normalize the QoE between $[0, 5]$ to mimic 5-star rating and consider 1-2 starts as low QoE, 3 stars as medium QoE, and 4-5 stars as high QoE.

The expected cumulative discounted reward can be calculated over the horizon h as follows:

$$R = \mathbb{E}[\sum_{i=1}^{h} \gamma^i \mathcal{G}] \tag{4.4}$$

where h specifies how far into the future the agent is trying to forecast. For example, using $h = 100$ and $\gamma = e^{-1/h} = 0.99$ means the actions taken by the RL agent will be influenced by 100 future steps. On the one hand, h should be large enough to provide a foresighted optimization, as opposed to a myopic one. On the other hand, it should be small enough for the problem to remain computationally solvable.

It is worth noting that it is possible to incorporate more objectives into the reward signal. One practical choice includes adding a term to encourage load balancing (e.g., using average link utilization). This can help avoid taking unnecessary risk which can cause congestion problems by discouraging the RL agent from grouping all clients in one tunnel. However, as we'll explain §5.2.2, the RL-agent was able to utilize such a strategy to reduce the OPEX without sacrificing QoE.

4.2.2 RL Environments

In this thesis, we try to answer two main question:

- How to use RL to automate NOC operations with super-human performance?
- How to utilize the field-datasets, which better capture the dynamics of real networks compared to the simulated environments, to pre-train RL models *safely* and *efficiently* similar to SL?

In this section, we proposed three custom OpenAI Gym environments [3] to train and test our RL agent: GNS3 Environment, Simulator Environment, and Batch-RL Environment. These environments aims at providing an answer to the second question as summarized in Table 4.1.

Table 4.1 Summary of the Developed Gym[1] Environments

		Simulator Env.	GNS3 Env.	Batch-RL Env.
Goals:	1.	For fast & safe exploration	To test with real devices	To experience realistic field data
	2.		To aid the design of the Simulator Env.	
Features:	1.	Synthetic traffic (Video & FTP)	Diverse traffic (Video & FTP)	Large amount of traffic (dataset-dependant)
	2.	Random traffic patterns (modeled by $\hat{P}(\hat{s'}\|\hat{s},a)$)	Random traffic patterns (described by $P(s'\|s,a)$)	
	3.	MDP-aware data augmentation	Congestion using real throughput traces	Convert time-series data (metrics and tickets) into $< s,a,r,s' >$ tuples
	4.	Simulate persistent, recurrent, and transient problems	Emulate persistent, recurrent, and transient problems	Field problems (dataset-dependant)

GNS3 Environment

Figure 4.5 is a screenshot of our testbed running in GNS3[2]. The topology shown in the figure is an implementation of the proposed small network in Fig. 4.1(A). It consists of 3 Autonomous Systems (AS) representing consumer-side and its ISP (AS1), an external ISP (AS3) which is used as backup in case of failure, and the service provider (AS2), which in this case is a video service provider like a Netflix or YouTube. In the GNS3 Environment, network problems were generated by randomly breaking links resulting in three types of problems: *persistent*, *transient*, and *recurrent*. We define a persistent problem to last for 15 timesteps, and we expect the agent to have fixed it by then. Transient problems come and go in 3 timesteps, and the agent can ignore them in some cases. Recurrent problems repeat every 100 steps, and we expect the agent to predict them before happening in the future. For traffic generation, we defined a sinusoidal pattern where clients start in random paths. Then for most of the time we have 3 to 4 clients who stream videos and select bitrates independently using DASH[3]. Finally, clients turn off and the cycle repeats.

In order to train the RL agent in GNS3 environment, each $\{s,a,r,s'\}$ step requires around 30 seconds to complete, so the size of our timestep is $30s$. While this environment worked well, it is not practical because of 2 reasons. First, the agent's training might need hundreds of thousands of steps, so the training will take a long time if done in real time. Second and more importantly, RL learns based on trial and error, and in

[2]https://www.gns3.com
[3]https://reference.dashif.org/dash.js/

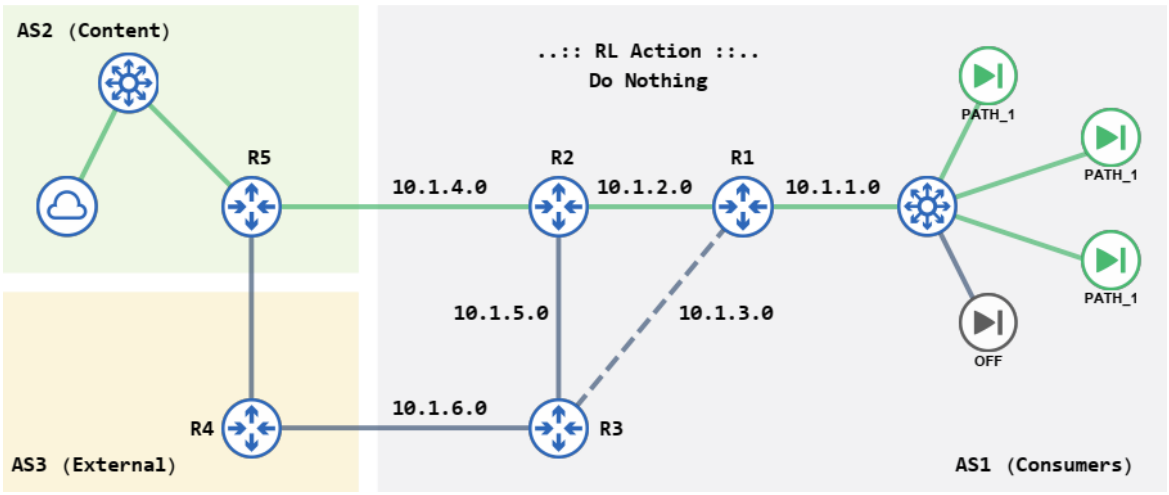


Fig. 4.5 Our system with its network topology. Clients (right) are connected to the video Server (left) through a tunnel group of three possible paths.

a live network we cannot afford to make errors. To deal with those two issues, we designed the other two Gym environments: the Simulator Environment which runs much faster than real time, and the Batch-RL Environment for offline training with a labeled dataset and with no risk of crashing an actual network. Both are described next.

Simulator Environment

For the Simulator Environment to be useful, it has to closely follow the dynamics of the GNS3 devices. In our formulation, environment dynamics are represented by the transition probability $P(s'|s,a)$ of the MDP. However, it is usually difficult to represent $P(s'|s,a)$ explicitly. In such cases, the simulator is used to model the MDP implicitly by providing samples from the transition distributions. One way to implement this is through a *generative model* [20]. Formally speaking, the Simulator Environment is a randomized algorithm that, given an input of a state-action pair $\{\hat{s},a\}$, it outputs a $\{r,\hat{s}'\}$ pair where r is the reward calculated according to a deterministic function given by equation 4.2 and $\hat{s}'$ is randomly drawn according to the estimated transition probability $\hat{P}(.|\hat{s},a)$.

To estimate $\hat{P}$, we let the agent interact with GNS3 and collected 50 hours of $\{s,a,r,s'\}$ transitions. The scenarios were diversified to allow an accurate estimate of transition probability. In particular:

- we collected enough transient, recurrent, and persistent *network problems* on all links. Also, network problems were created on empty and congested links alike.

- we mixed between three types of adaptive bitrate (ABR) algorithms: buffer-base, throughput-based, and fixed bitrate. This ensured all available bitrates had been selected by the DASH clients and diverse *congestion patterns* had been created. Furthermore, we collected hundreds of thousands of video metrics (1 metric every 2 seconds) and calculated the probabilities of switching bitrates between the QoE levels {high, medium, low}.

- we let the agent operate in three modes: random policy (total exploration), gradually improving policy (decaying exploration), and expert rules (total exploitation). This ensured all actions had been presented and that good and bad actions had been chosen as well.

Once $\hat{P}$ is estimated, the Simulator Environment can be used to generate synthetic data $\{\hat{s}, a, r, \hat{s}'\}$ to train the RL agent. Note that what we actually generate is the observation vector $\hat{o}$ that corresponds to the chosen state $\hat{s}$.

In order to make sure all actions are well presented in as many states as possible, we apply data augmentation to the next observation $\hat{o}'$. However, since the agent uses transition probabilities between states $\{\hat{s}, \hat{s}'\}$ to plan future actions, we apply data augmentation in a way that preserves this transition. Formally speaking, let $\mathcal{T}_o^{-1} : \mathcal{O} \to \mathcal{S}$ be a deterministic function that maps observations to corresponding states. We want to perform data augmentation using observation function $\mathcal{T}_o : \mathcal{S} \to \mathcal{O}$ such that the augmented observation $\tilde{o}' = \mathcal{T}_o(s')$ and $\hat{o}'$ are mapped to the same state $\hat{s}'$. For example, if a client was streaming video at rate q_1 at time t with high QoE, and then a link failure happened causing its bitrate to drop to q_2 at time $t+1$ with low QoE, we augmented this by generating examples where q_1 and q_2 are randomly drawn from the set of all bitrates associated with high and low QoE, respectively. In other words, we change the observations in a way that doesn't change the underlying state nor the reward. Figure 4.6 illustrates this process.

One advantage of using the Simulator Environment for training is the ability to put emphasis on rare cases as a way to help the RL agent explore. Also, to avoid overfitting, we tested the RL agent trained with this synthetic data on unseen scenarios in the GNS3 Environment to make sure it can generalize well.

Batch-RL Environment

In practice, an operator might have access to an existing labelled dataset collected from the field, but no access to a simulator. The dataset in this format is suitable for SL-based ARE and is discussed more in §4.3. The Batch-RL Environment is built

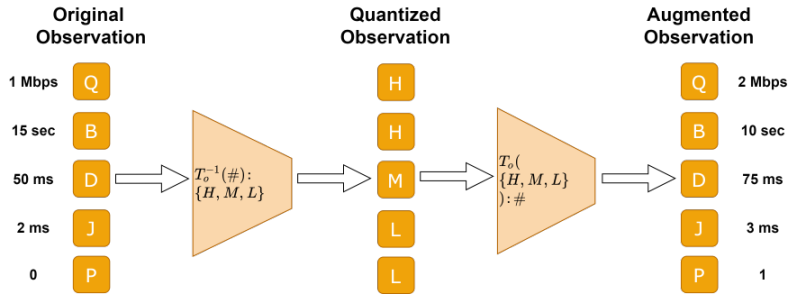


Fig. 4.6 Data augmentation in simulator environment. Observations: $\{Q$: bitrate, B: buffer, D: delay, J: jitter, P: packetloss$\}$. States: $\{H$: High, M: Medium, L: Low $\}$

for such a situation in order to utilize labelled datasets in RL settings. In this case, the environment accepts two CSV files as an input and converts them to a Gym environment. The first file corresponds to time-series data collected from the devices log files. The second file contains the taken actions typically in a form of ticket with timestamps. Depending on dataset format, time alignment between observation logs and actions logs could be needed. Once aligned, the next state can be derived from each row by observing the next row. Finally, a reward signal can be computed and $\{s, a, r, s'\}$ are stored in a CSV file. Each CSV file represents one episode in the Batch-RL environment.

To train the RL agent with the Batch-RL Environment, the agent is put on 100% exploration mode where instead of picking an action randomly, the agent picks its actions according to the CSV file. Each line in the file is read and the reward is provided accordingly. Note that despite restricting the agent's actions in every timestep, Batch-RL can still be valuable if the dataset is rich enough. Once this offline pre-training is done, the agent can be deployed in a real network either for inference or to continue learning online and improve itself beyond the policy learned from the states and actions in the dataset. More information on how to train an RL agent using Batch-RL algorithm is explained in §4.2.4.

Discrete Event Simulator

For the scalability test, we implemented another environment as a discrete event simulator (DES). Unlike the Batch-RL and Simulator environments, this simulator is not derived from the data collected in GNS3. Note that such generative models can be useful in practice given the availability of field dataset. However, since we don't have access to such data for large network, and since running the big network on GNS3 emulator is not feasible, We opted for designing the simulator from scratch by modeling

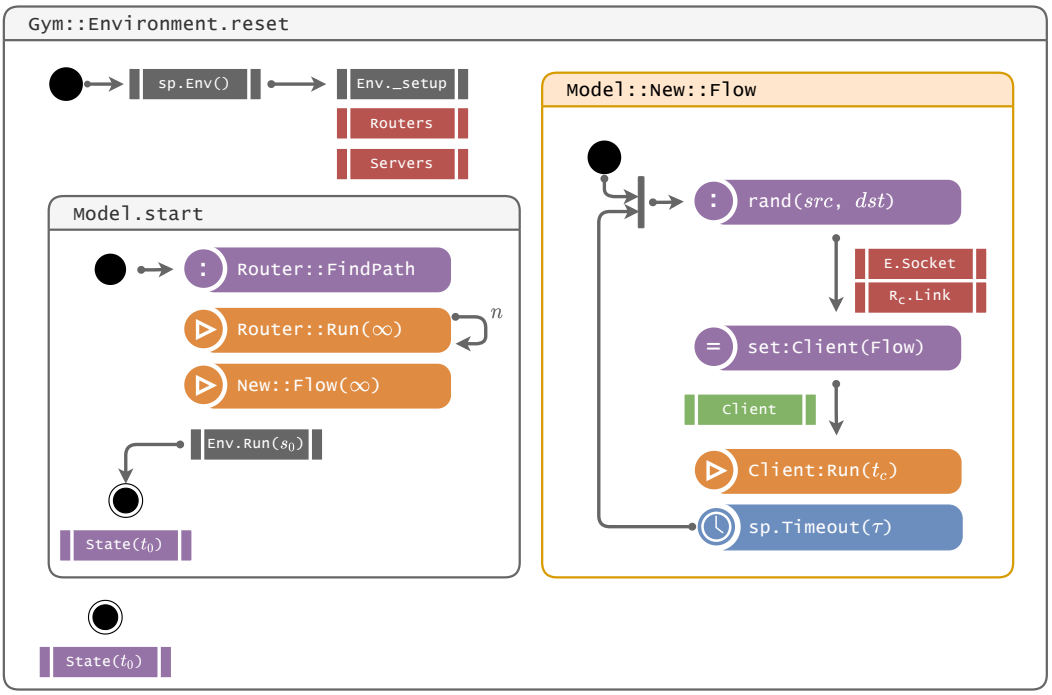


Fig. 4.7 Simulator initialization function. Orange blocks represent a scheduled process, Blue blocks are events to wait for, Red blocks are network resources, while magenta represents RL related blocks

.

each component in the network. Figures 4.7-4.10 present a high-level overview of the main components.

The first important note to recognize in Figure 4.7 is that our simulator represents *time* at two different scales. On one hand, there is a global clock that represents the RL timesteps. The initial state received by the RL agent after "resetting" the environment is called t_0. There is another local clock that represents the DES time. In practice, we found that a client can report its QoE metrics every 2 seconds, a router can report its IPSLA metrics every 15 seconds, and the amount of time needed to produce one RL update is 40 seconds.

At the beginning of the simulation, we create the topology, update the routing tables, and flush all buffers (Routers ports and Servers buffers). We use Dijkstra algorithm to determine the shortest path between routers and servers in the network. Note that a complete implementation for OSPF routing algorithm is not required since we uses overlay network and MPLS tunnels with predetermined tunnels. Once the tunnels are defined, the simulator then run all routers to be ready for incoming packets. Once the routers are ready they will keep running till the end of the simulation. Next, the simulator schedule another parallel process that is responsible for creating video

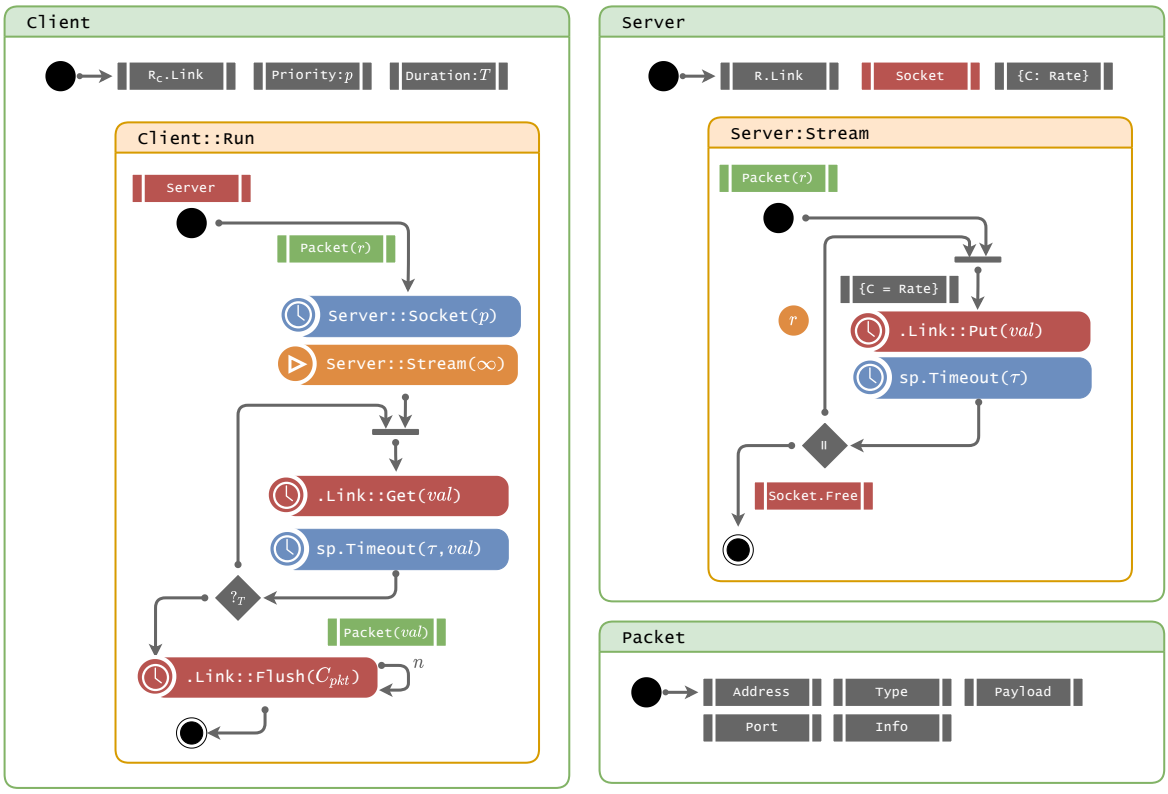


Fig. 4.8 Simulating a video session.

.

sessions at random time intervals. It starts by picking a random server and router, it creates a client and attach the router as its gateway, the client start streaming for a specific session duration, meanwhile, other clients can be created using the same process.

Figure 4.8 illustrates the details of a video streaming session. Video servers are of two types: Edge server with limited capacity or the cloud. Once the client is created, it get assigned to a server (either randomly or by RL agent). Since Edge servers have limited capacity, the client has to send a request to the server and wait for the response before being able to start streaming. Once the connection is approved, the server start sending packets at the requested bitrate r. The server put the packet in its buffer and wait for its gateway to deliver. Similarly, the client starts checking the its gateway buffer at regular interval an yield the result. Once the video session is over,the client sends an interrupt message to the server, clean its old packets and free its reserve socket for others to use it.

Note that the client chooses the desired bitrate of the next segment based on BOLA algorithm. This algorithm is buffer-based algorithm. Meaning, the next bitrate

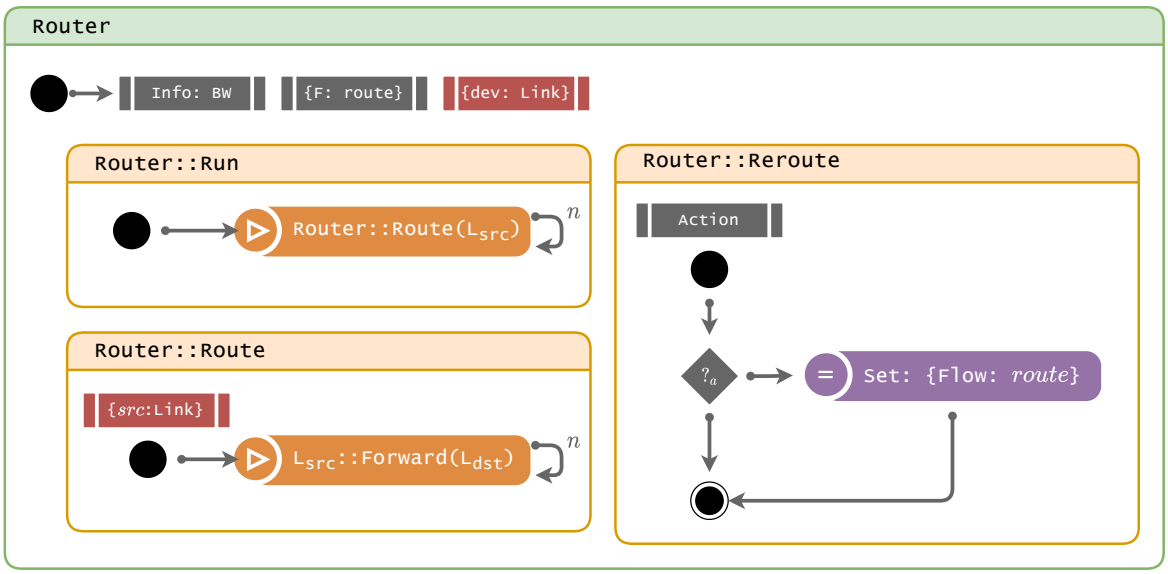


Fig. 4.9 Simulating a router device.

.

depends solely available buffer level. With this in mind, the client can simply calculate its QoE and append it in the packet payload. The packet stores not only the metrics and QoE, it also store information on the whole path it followed, which can be useful for later analysis.

Figure 4.9 illustrates the routing process. As mentioned earlier, the simulator starts by running all routers before any traffic is pushed through the network. Running a router causes it to scan all incoming ports, read any available data, and initiate a route process to the destination port at the end of the link. The reason why it has to initiate a routing process is because the other port can be full. In this case, the link between this source port and that destination port will be blocked (see Timeout event in Figure 4.10). Note that an RL agent can decide to reroute the traffic in two ways: it can switch the path between the main tunnel and the backup tunnel. Or it can decide to assign the client to a different server. In the first case, the client and server won't notice the change. Only the gateways router at the two ends of the tunnels (called Provider Edge router (PE)) are aware of this change. In the second case, the client has to terminate the session, make a new request to the new server, wait for approval, and then continue streaming.

In terms of performance metrics, the QoE metrics are already available for the client: (i) bitrate is decided based on the buffer, the buffer can increase or decrease based on the download speed, the startup delay is measured by the wait time for the server to accept the connection. For the QoS metrics, we approximate the latency

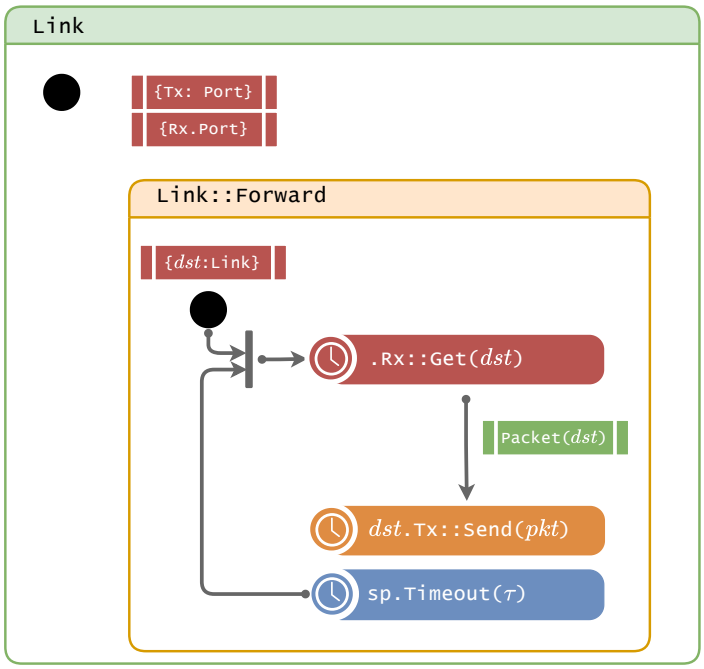


Fig. 4.10 Simulating a router Link.

.

as a function of the number of hops. The jitter and packet loss are defined per link as random variables whose distribution is derived from the GNS3 dataset collected from the Basic network. Once the metrics for each link are measured, the end-to-end metrics are calculated as addition/multiplication of individual link metrics (for delay and packetloss respectively.

Finally, network issues are created directly by reserving some of the resources in the network. This includes: reserve sockets on video server, fill router buffers, etc. It can also be created indirectly by streaming another type of traffic across the network, causing it to compete with the video traffic.

4.2.3 Training Algorithms

In this thesis, we try to answer two main questions:

- How to use RL to automate NOC operations with super-human performance?
- How to utilize the field-datasets, which better capture the dynamics of real networks compared to the simulated environments, to pre-train RL models *safely* and *efficiently* similar to SL?

This section is dedicated to answering the first question. While we are mainly interested in using RL, we dedicate the next subsection to describe some of the advantages of using SL-based ARE to motivate the feasibility of automating NOC actions, and to highlight the benefits of using an offline pre-training step for RL agents.

Deep Learning

In this work, we hypothesized that there are logical relationships between network problems and their remediation actions, even if those relationships are difficult to codify for complex networks. SL is therefore a good tool to model those relationships and take actions autonomously. The use of SL is further justified by the fact that the industry is already moving towards ML tools for network analytics and state detection, such as Ciena's Blue Planet Unified Assurance and Analytics[4], or Blue Planet Route Optimization and Analysis[5], which help the NOC technicians gain deeper insights into the network to make intelligent data-driven decisions that lead to improved efficiency, lowered costs, and providing more personalized services.

We chose DL among other SL algorithms for two reasons: on one hand, DL models are more flexible, powerful, and known to outperform all traditional ML algorithms. On the other hand, we aim at studying the possibility of using SL to pre-train RL models. The idea is to define a NN model for an RL agent, and use DL to efficiently pre-train it on a time-series dataset that was collected and labeled by human experts. This form of training is called *Beauvoir cloning* [30]. In this framework, the agent receives as training data both the encountered states and actions of the demonstrator, and then uses a *classifier* to replicate the expert's policy (where each class represents an action). As we can see in Figure 4.11, the agent has 3 groups of actions that represent 12 classes: "Do Nothing", "Reroute Traffic" between 3 paths, and "Fix Network Issues" on 5 different links. The input features has 18 different performance metrics as detailed in §4.3.

Regarding network architecture, we used Feed-forward NN (FNN) and SeLU activation function to mimic the NN used by our RL agents. We trained the SL model using the standard back-propagation algorithm. One big challenge in our case is the problem of unbalanced classes; i.e., one class including more data points than other classes. This is typical because "Normal" state is the dominant network condition in reality and problems occur less often than Normal. To deal with this issue, we performed data augmentation (see §4.2.2) and used *Focal Loss* [24] that adds a weighting

[4]https://www.blueplanet.com/products/uaa.html
[5]https://www.blueplanet.com/products/route-optimization.html

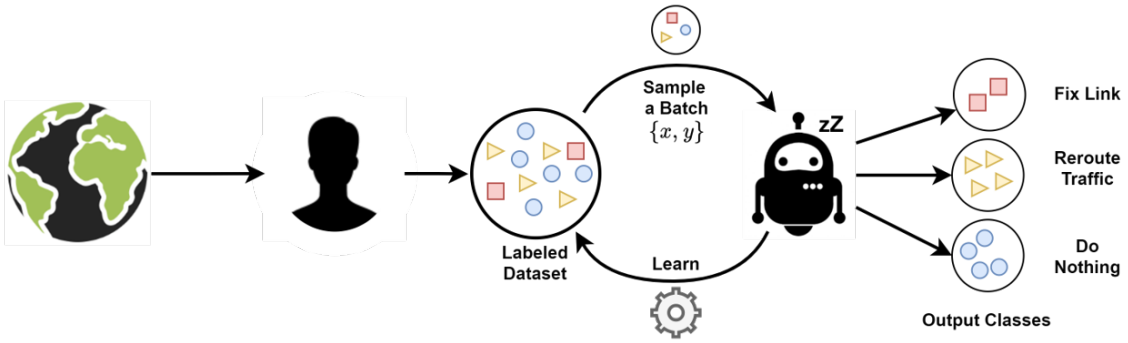


Fig. 4.11 A SL-based ARE trained on labeled dataset generated by human experts.

factor to the standard cross entropy criterion. This factor reduces the relative loss for well-classified examples, putting more focus on hard, misclassified examples.

RL Algorithms

RL algorithms can be generally divided into Model-based and Model-free algorithms. The agent in the former approach tries to model the environment purely from experience. The biggest challenge in this approach goes beyond being expensive and complicated; it is that any bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model but fail in the real environment. For that, we focus on the Model-free algorithms. Model-free algorithms can be further divided based on the into Q-based algorithms and Policy-based algorithms. In this work, we chose Double-DQN and A2C as two candidate algorithms to represent those two families.

Double-DQN [35] is an improved version of the famous Deep Q-Network (DQN) algorithm. As the name implies, DQN uses DL to estimate the Q-function, a function used to estimate the expected value of each possible action given a specific state. Double-DQN uses two NN (see Figure 4.12) to avoid the large overestimation of action values seen in the original algorithm, which leads to poor performance in some stochastic environments. We chose to use the NN form of this algorithm as opposed to the tabular form for two reasons. First, our state space is continuous, and NN can produce better representation compared to simply discretize the state-space. Second, using two NN makes Double-DQN match closely the architecture of A2C algorithm (which also uses two NNs). This makes it easier for us to compare the two algorithms. Finally, since DQN uses a simple FFN to estimate the Q-function, one can benefit from SL to quickly pre-train the DQN model using behaviour cloning as mentioned before.

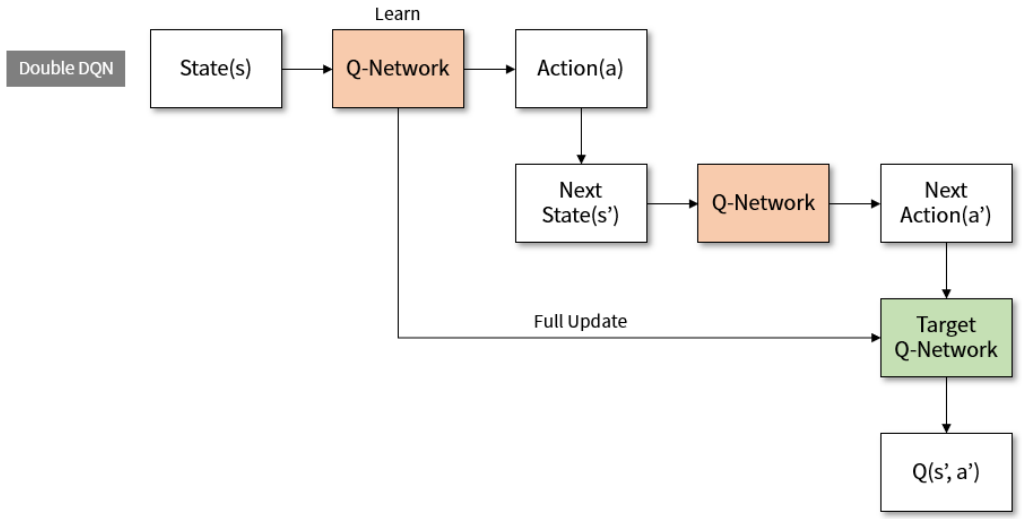


Fig. 4.12 Double Deep Q-Network algorithm. Orange block is the actor network, green block is the learner network. [6]

When the agent interacts with the environment, the collected sequence of experience tuples can be highly correlated. This violates one of the fundamental requirements for a successful training: to have an independent and identically distributed (IID) training data. To break this correlation, our RL agent build a "replay buffer" that contains a collection of past experience tuples (s, a, r, s'). The tuples are gradually added to the buffer during the interaction with the environment (online-step). During the offline-step, the agent randomly samples from the collected episodes in the buffer and learn. The buffer size represents agent's memory; as new data comes in, the oldest experiences are forgotten. In addition to breaking harmful correlations, experience replay allows RL agent to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of its experience.

One key feature of the Batch-RL algorithm is the presence of a replay buffer. This feature is particularly useful because it allows the RL agent to access the buffer and learn from experiences drawn from a teacher policy, rather than relying solely on experiences drawn by the agent itself. This can be highly advantageous, as it enables the agent to learn from a wider range of experiences and can help to reduce the impact of noisy or suboptimal experiences on the learning process.

In the Advantage Actor-Critic (A2C) algorithm [21], each agent is composed of a policy network (actor) and a value network (critic). The two networks interact and learn how to not only take actions but also evaluate their effectiveness.TThe agent's current policy is updated in the direction that would increase the accumulated reward.

The size of the update step depends on the *value* of the advantage function $A(s_t, a_t)$ calculated by the critic given by (4.5). ω and ω_v represent the actor and critic neural networks (NN), respectively.

$$\begin{aligned} A(s_t, a_t) &= Q(s_t, a_t; \omega) - V(s_t; \omega_v) \\ &\approx r_{t+1} + \gamma V(s_{t+1}; \omega_v) - V(s_t; \omega_v) \end{aligned} \tag{4.5}$$

The gradient of the cost function $J(\omega)$ with respect to the policy parameters ω, can be computed as:

$$\begin{aligned} \nabla_\omega J(\omega) &= \nabla_\omega \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \\ &= \mathbb{E}_\pi \Big[\nabla_\omega \ log\Big(\pi(s_t, a_t \ ; \omega)\Big) \ A(s_t, a_t) \Big] \end{aligned} \tag{4.6}$$

and the update rule for the parameters of the actor's NN is given by:

$$\begin{aligned} \omega \leftarrow \omega \ + \ \alpha \ \nabla_\omega log\Big(\pi(s_t, a_t \ ; \omega)\Big) \ A(s_t, a_t) \\ + \ \beta \ \nabla_\omega H(\pi(s_t \ ; \omega)) \end{aligned} \tag{4.7}$$

where $H(\pi_\omega(s_t))$ is an entropy regularization term added to the loss in order to prevent the actor from over-fitting on a small portion of the environment (i.e., keep using the same actions). β controls the strength of the entropy regularization term and is set to a large value at the beginning to encourage exploration and then decreases over time to emphasize improving the rewards.

The update rule for the critic network parameters is derived using the standard Temporal Difference method [33] and is given by:

$$\omega_v \leftarrow \omega_v - \alpha_v \ \nabla_{\omega_v} \Big[\Big(r_t + \gamma V(s_{t+1}; \omega_v) \Big) - V(s_t; \omega_v) \Big]^2 \tag{4.8}$$

where α_v is the learning rate of the critic.

It is worth noting that A2C algorithm has another popular variant called A3C. The two algorithms are similar in the sense that both are scalable. The two algorithms support training multiple agents in parallel by allowing each agent (actor) to sample

[6]Source: *https://greentec.github.io/reinforcement-learning-third-en/*

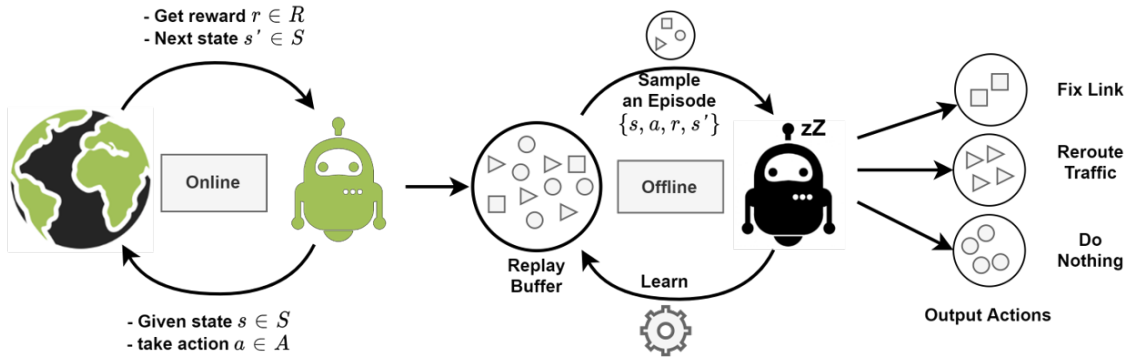


Fig. 4.13 RL-based ARE trained on unlabeled dataset generated continuously by the agent itself. The agent keeps switching between online (data collection) and offline (learning) steps.

its own experiences and push it to a shared buffer. A master model (the learner) then uses this data to improve and update the actors. A2C is a synchronous, deterministic implementation that waits for each actor to finish its segment of experience before performing an update, averaging over all of the actors. A3C is an asynchronous version (hence the name A3C). One advantage of A2C method is that it can more effectively utilize the GPUs and perform best with large batch sizes.

It is worth noting that while other recent improvements on the Actor-Critic algorithm exists, we chose A2C because it is still widely deployed in practice. This is true mainly because it is very efficient to train, and most importantly, very scalable thanks to the fact that A2C can be trained in a decentralized way [4]. For example, in adaptive video streaming, [27] achieved state-of-the-art performance using an asynchronous version of it. And in a recent paper [26], the authors' RL approach outperforms the existing ABR expert rules in a week-long worldwide deployment with more than 30 million video streaming sessions.

4.2.4 Practical Considerations

Training an RL-based NOC automation system poses two main challenges. Firstly, it is crucial to prevent the RL agent from making "bad" decisions during real-network training which can be costly. Secondly, training an RL agent to control a large-scale network is challenging due to the extensive state and action spaces involved. To address the first issue, we propose Batch-RL, a pre-training algorithm that allows us to train the RL agent using a NOC-controlled policy. This encourages the agent to mimic the algorithms used by professional teams before exploring other actions. To tackle

the second issue, we can train the RL agent in a simulated environment and perform transfer learning to the real environment. This approach mitigates the challenges of training an agent to control a large-scale network.

Note that transfer learning is usually needed to account for any differences between the simulated environment and the real-world environment. This is usually the case because of two main issues. First, developing a simulator that exactly models every aspect of the real-world environment is extremely hard and costly. Therefore, developers tend to build simulators that approximate and simplify some aspects that don't have significant contribution to the task in hand. Second, the real-world environments can change in the long term. Therefore, it is important to tune an RL agent trained in the simulator to account for these changes in the wild. Examples of such changes can include the action-space, the state-space, and the meaning of the reward signal.

This section outlines the high-level steps of our algorithm for training a decentralized RL model that addresses the challenges discussed earlier. Firstly, we break down the network topology into topology-agnostic subnetworks. This process involves the following steps:

1. Identify all edge servers in the network as well as the backup media server. Edge servers are the primary source of video content to the neighboring clients.

2. Identify all edge routers in the network. These routers serve as gateways connecting video clients to the nearest video server.

3. For each edge router, construct two tunnel groups starting from the router. The first tunnel group includes two tunnels connected to the edge server. The tunnels are selected by solving an optimization problem that aims at minimizing the number of hops in each tunnel while satisfying the constraint that the two tunnels should not have more than 50% overlap. This requirement reduces the risk of one network problem affecting both the tunnel and the backup tunnel. The second tunnel group includes two tunnels connected to the backup video server. They are selected using the same algorithm as the first tunnel group.

Once the tunnels groups are established from all edge routers to all edge/backup servers, we assign one agent to each edge router to manage its tunnel groups. Each agent has access to information only about the end-to-end KPIs and can optionally obtain aggregated information about the topology, such as the number of hops along a path, the number of clients using a particular path, or the cost of moving traffic to a path. By doing so, the agent's observation space is completely agnostic to the

specific topology of its subnetwork, which enables it to be potentially applied to other subnetworks within the larger network.

Next, we proceed to train a single "master" agent on a subset of these topology-agnostic subnetworks. To maintain consistency, the action space of the training subnetworks should be normalized, with the number and meaning of each action kept the same. However, this constraint is not strictly required for deployment, as we will explain shortly. The training action data sources can come from various sources, such as field data in the form of tickets, logs, and user feedback, which can be further enhanced through a statistical simulator. Alternatively, the training data can be generated by a discrete event simulator, along with expert rules or manual labor to define actions.

It is worth noting that if a simulator is used to generate synthetic data, random background traffic can be added to random hops to represent interactions with external actors, such as the NOC or other automated agents on the network. Additionally, if the number of subnetworks is high, the action space can be explored in parallel using multiple agents (workers), and these interactions can be collected into a central dataset for training the master RL agent. The advantage of using this master-workers approach for training is to ensure that the master agent is exposed to a wider range of network conditions than what a single agent can experience. The action/state space can be explored in parallel using two different methods, depending on the learning approach

In traditional (online) RL, multiple worker agents interact with the environment in parallel to explore the action and state spaces across all possible subnetworks. In batch (offline) RL, mock NOC expert workers interact with the simulator to generate action events. A combination of the above two strategies is also valid. Once the master agent have enough interactions, it updates its policy and sync the new learned policy with the worker agents such that the next collected interactions reflect the latest policy.

Third, we test the master agent on the remaining, unseen subnetworks to validate its performance. at this step, we deploy the RL agent with zero exploration, only exploitation, and confirm its behavior in the production environment. In our work, we expect to approximately reproduce the performance of expert rules at this point.

Finally, once it achieved an acceptable performance, we copy the agent across all subnetworks. During this step, it is possible to allow small RL exploration during run time, to learn new and better action policies. At this stage, we expect the RL agent to significantly outperform the expert rules' performance, eventually leading to superhuman performance.

Note that during transferring the knowledge to the production environment, the action space might be different than the simulated environment. If the action space is

identical between the training and test subnetwork, then the agent can be used as-is (zero-shot learning). However, if they differ, the agent can be tuned to the new action space as follows.

- If there are any actions missing from the new action space, a strategy called action masking can be used to ignore the actions that are no longer applicable to new environment.
- If the actions have different meanings, a strategy called reward shaping can be performed. In this case, one customizes the reward signal that the agent receives to align more closely with the new meaning of the action. This option requires a human in the loop to determine a mapping between the old and new reward signal.
- It is also possible to replace the last layer of the policy network which encode the actions, reusing the features learned by the actor. This requires less data than relearning from scratch.

It's worth noting that while this thesis is focused on video streaming applications using MPLS tunnels, the approach should still be applicable to other use cases and other technologies. For example, to adapt our algorithm for other applications such as IoT or gaming applications, we have two options. The first is to define suitable KPIs and design a reward signal. Then train the model from scratch using our algorithm. The second option is to apply transfer learning as mentioned earlier (mainly through rewards shaping).

As for the networking technology, we chose to work with MPLS tunnels (despite being an aging technology) as opposed to newer alternatives such as SD-WAN or 5G for two reasons. On one hand, MPLS technology is still widely used in ISP networks. Therefore, we aim at helping ISPs and their NOC teams to automate these legacy networks. On the other hand, our RL algorithm is transparent to the choice of underlying infrastructure technology because we choose to work with end-to-end metrics at the transport layer, such requirement is fulfilled by all these technologies and the general concepts still apply.

To conclude this section, it is important to note that both the Simulator Environment and the Batch-RL Environment have their own advantages and disadvantages. The Simulator Environment offers the advantage of generating synthetic data in virtually unlimited amounts, whereas the Batch-RL Environment relies on historical data from the real network, which is finite in quantity and can be expensive to collect. However,

the disadvantage of the Simulator Environment is that the transfer from the simulator to the real network can be sensitive to simulation defects. Thus, developing a high-quality simulator is crucial but challenging. On the other hand, the Batch-RL Environment learns from real network situations, which may reduce the sensitivity to simulation defects, but may not provide as much flexibility in terms of generating diverse data.

4.3 Input Metrics

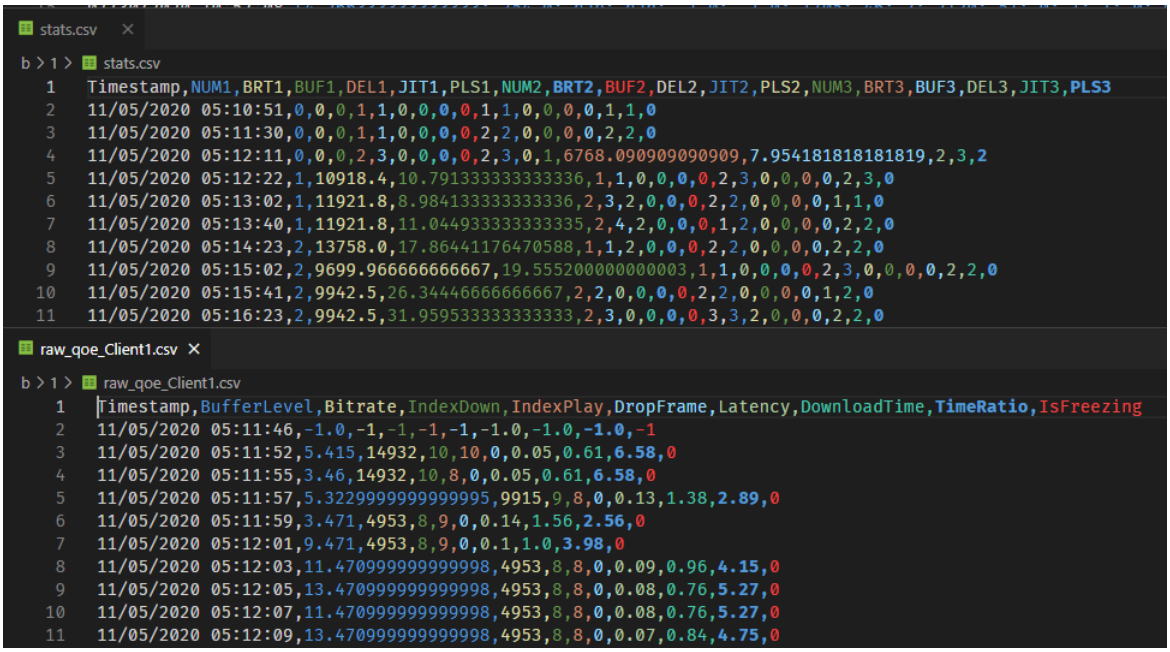
We collect three types of metrics within AS1, each serving a specific purpose. Firstly, we gather end-to-end QoS metrics, which encompass IPSLA metrics for each path, such as delay, jitter, and packet loss. These metrics provide the agent with vital information about the network state, enabling the identification of any issues within the ISP network, such as congestion.

Secondly, we capture end-to-end QoE metrics, which focus on video-related aspects including the client's bitrate, buffer, and download-time. These metrics offer insights into the quality of video sessions from an OTT perspective. By collecting this information, the agent gains a comprehensive understanding of the video session's performance.

Lastly, we gather network-related metrics, which encompass any relevant information about the network topology. This can include details such as the number of clients sharing a tunnel or the number of hops in a tunnel. These metrics play a crucial role in eliminating any ambiguity surrounding the collected data. They are particularly important due to our utilization of end-to-end metrics instead of per-hop metrics and the aggregation of metrics per-tunnel instead of using raw metrics.

Figure 4.14 shows a sample of QoS metrics as well as QoE metrics. Note that the QoS file contains the end-to-end metrics for all 3 paths. However, the QoE file represents the metrics for a single DASH client (i.e., there are 4 files in total). Moreover, the data in the QoS file are all fed to the RL agent. However, the agent doesn't observe the individual QoE metrics. Instead, the QoE metrics and averaged over a fixed time-interval (to match the rate in which IPSLA statistics were collected) and then aggregated per path. We also collected per-port metrics such as port packet loss, but we found that we didn't need them, and our ARE worked fine with E2E QoS & QoE metrics only.

It is worth noting that while the traffic aggregation step is performed per path to mimic a real-world scenario, this poses a real challenge to the RL agent. In particular, we calculate the reward signal based on QoE metrics from individual clients. This



stats.csv

```
Timestamp,NUM1,BRT1,BUF1,DEL1,JIT1,PLS1,NUM2,BRT2,BUF2,DEL2,JIT2,PLS2,NUM3,BRT3,BUF3,DEL3,JIT3,PLS3
11/05/2020 05:10:51,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0
11/05/2020 05:11:30,0,0,0,1,1,0,0,0,0,2,2,0,0,0,0,2,2,0
11/05/2020 05:12:11,0,0,0,2,3,0,0,0,0,2,3,0,1,6768.090909090909,7.954181818181819,2,3,2
11/05/2020 05:12:22,1,10918.4,10.791333333333336,1,1,0,0,0,0,2,3,0,0,0,0,2,3,0
11/05/2020 05:13:02,1,11921.8,8.984133333333336,2,3,2,0,0,0,2,2,0,0,0,0,1,1,0
11/05/2020 05:13:40,1,11921.8,11.044933333333335,2,4,2,0,0,0,1,2,0,0,0,0,2,2,0
11/05/2020 05:14:23,2,13758.0,17.86441176470588,1,1,2,0,0,0,2,2,0,0,0,0,2,2,0
11/05/2020 05:15:02,2,9699.966666666667,19.555200000000003,1,1,0,0,0,0,2,3,0,0,0,0,2,2,0
11/05/2020 05:15:41,2,9942.5,26.34446666666667,2,2,0,0,0,0,2,2,0,0,0,0,1,2,0
11/05/2020 05:16:23,2,9942.5,31.959533333333333,2,3,0,0,0,0,3,3,2,0,0,0,2,2,0
```

raw_qoe_Client1.csv

```
Timestamp,BufferLevel,Bitrate,IndexDown,IndexPlay,DropFrame,Latency,DownloadTime,TimeRatio,IsFreezing
11/05/2020 05:11:46,-1.0,-1,-1,-1,-1,-1.0,-1.0,-1.0,-1
11/05/2020 05:11:52,5.415,14932,10,10,0,0.05,0.61,6.58,0
11/05/2020 05:11:55,3.46,14932,10,8,0,0.05,0.61,6.58,0
11/05/2020 05:11:57,5.3229999999999995,9915,9,8,0,0.13,1.38,2.89,0
11/05/2020 05:11:59,3.471,4953,8,9,0,0.14,1.56,2.56,0
11/05/2020 05:12:01,9.471,4953,8,9,0,0.1,1.0,3.98,0
11/05/2020 05:12:03,11.470999999999998,4953,8,8,0,0.09,0.96,4.15,0
11/05/2020 05:12:05,13.470999999999998,4953,8,8,0,0.08,0.76,5.27,0
11/05/2020 05:12:07,11.470999999999998,4953,8,8,0,0.08,0.76,5.27,0
11/05/2020 05:12:09,13.470999999999998,4953,8,8,0,0.07,0.84,4.75,0
```

Fig. 4.14 Sample of collected dataset. Top: QoS metrics measured every 30 seconds. Bottom: QoE metrics measured every 2 seconds

information is not available to the RL agent. For example, assume that the agent receives a reward of +5 for every client stream at high quality (say at bitrate } 3000 Kbps). When the agent observes an aggregated traffic of 7000 Kbps from 2 clients, it can't easily tell whether it will receive +10 (if both were streaming at 3500Kbps), +5 (if one was streaming at 6000 Kbps while the other at 1000Kbps), or even 0 (if one client is streaming at 7000 kbps while the other is hanging.

Figure 4.15 shows the resulted dataset after applying the aforementioned pre-processing steps. As we can see, the file also contains the true state of the environment. This information is not available to the RL agent and is only being used for evaluation purposes. The main two pieces of information stored are (i) the root-cause of any problem and (ii) the routing-table that tells which client is using what path.

We tried training both SL and RL agent on this dataset in its current form, but the results were not very good. The main two problems with the data are: First, different metrics have different ranges that need to be normalized. For example, the range of possible bitrates are from $0 - 15000kbps$ while the range of video buffer is barely $0 - 15sec$. Such a huge difference between different metrics causes the gradient of the NN to explode. Second, different metrics have different meanings, and they follow different trends. For example, it is more convenient to have all metrics normalized in a way such that the larger the metrics the better (or vice versa). This is not the case

Fig. 4.15 Sample of processed dataset. It includes state, action, rewards, and next state. As well as true information about the true environment states.

with the current setup. As mentioned before, the larger the bitrate value the better, however, the opposite is true for the jitter. To handle those two issues, we normalized all metrics to have values between $(-1, 1)$ and multiplied delay, jitter, and packetloss with -1 such that 1 represents the best case.

It is worth noting that this dataset serves two purposes: firstly, to gather experiences from high-quality policies, such as expert rules, in the GNS3 environment as a basis for Batch-RL learning. Secondly, to encourage the agent to explore as many actions as possible in all states to better understand the underlying dynamics of the environment. The resulting dataset provides the foundation for the simulator environment, which is capable of generating an infinite amount of synthetic data.

Chapter 5

System Evaluation

In this section, we present an evaluation of the performance of our system. We begin by introducing the evaluation metrics and the baseline algorithm used to benchmark our approaches. For each use case, we provide an explanation of the experimental setup, including details on the environment used and the implementation specifics of the machine learning algorithm. Subsequently, we analyze the obtained results and provide a conclusion based on our findings.

5.1 Evaluation Metrics

In order to evaluate the effectiveness of our algorithms, we utilized two different metrics. The first metric we employed was the *Gain*, which is described in Equation 4.2, to compare the performance of RL-based methods. Additionally, we used the Confusion Matrix to evaluate the effectiveness of SL-based methods. We also measured the convergence speed of each algorithm by determining the number of steps required to achieve at least 90% of the baseline algorithm's performance.

To simulate the actions of a NOC operator, we implemented the expert rules presented in Algorithm 1. Although this algorithm is still understandable, it demonstrates why expert rules can quickly become complicated as the network size increases. Therefore, we opted not to choose a network larger than the one depicted in Figure 4.5.

The rationale behind the baseline algorithm can be explained as follows. Initially, the algorithm examines the network metrics and recognizes two types of issues: traffic-related problems (also known as congestion) and device-related problems (also known as link problems). In reality, fixing congestion problems can be less expensive as traffic rerouting can be performed remotely. Therefore, the algorithm prioritizes rerouting as the primary means of resolving the problem.

In our scenario, we consider a situation where two backup paths are available, but one of them is more expensive. This scenario simulates a situation where some of the network resources are owned by the ISP while others are leased from a third party. In this case, the algorithm favors the less expensive route. A less expensive route is also obtained by prioritizing paths with fewer hops.

Moreover, the algorithm is intelligent enough not to disrupt the destination path while performing traffic rerouting. This is accomplished by checking if the destination tunnel can accommodate the additional traffic without compromising the QoE of the clients being served there. We also consider the duration for which a problem may persist. Some network issues may be transient and resolve themselves after a certain time. In such cases, the algorithm strives to avoid taking costly measures unless they are absolutely necessary.

It should be noted that the inputs to the baseline algorithm include the network's state. However, this is not practical in real networks. This is because the true state of large networks is not known, and only observations (i.e., KPI metrics) are available. We were able to include the true state in the baseline algorithm since we utilized it in a controlled emulated environment. In this case, we created the network issues and knew the type, time, and location of all issues within the network. Therefore, the purpose of using the true state was to simplify the implementation of the baseline algorithm, which will be employed as a reference to evaluate the performance of the other ML algorithms. As for the ML algorithms, they do not have access to the true state; they can only learn from observations. Being able to surpass the performance of the baseline algorithm using ML without access to the true state of the network is one of the principal contributions of this thesis.

5.2 Test cases

5.2.1 Feasibility of NOC Automation

Environment Setup

To assess the feasibility of automating NOC using ML, we utilized the same topology depicted in Figure 4.5. In this setup, the neural network (NN) has complete knowledge of the network topology and is capable of taking corrective actions at the link-level, such as repairing a broken link. We emulated the network in GNS3, consisting of 5 Cisco routers and 3 IP/MPLS tunnels connecting varying numbers of AS1 DASH video clients to the AS2 video server. We intentionally varied the number of clients to create

Algorithm 1 Baseline

Input: State $\mathcal{S}$, Observations $\mathcal{O}$
Input: Paths($I_1 : R_{1,2,5}, I_2 : R_{1,2,4,5}, E_3 : R_{1,3,4,5}$)
Output Action
if an internal path I_j is congested **then**,
    $I_k \leftarrow$ Select(I_k s.t. $k \neq j$ **AND** I_k is not congested)
    client $\leftarrow$ Select(random client from path I_j)
    if path $I_k \neq \phi$ **then**
        action $\leftarrow$ Reroute(client to path I_k)
    else if external path E_3 is not congested **then**
        action $\leftarrow$ Reroute(client through E_3)
    else
        action $\leftarrow$ Do Nothing
    end if
else if Network issue **then**,
    action $\leftarrow$ fix the issue (if persisting for 3 time steps)
else if External path E_3 is being used **then**,
    $I_k \leftarrow$ Select(I_k s.t. I_k isn't congested)
    if $I_k \neq \phi$ **then**,
        action $\leftarrow$ Select(random client from path E_3)
        action $\leftarrow$ Reroute(client to path I_k)
    end if
else
    Do Nothing.
end if

Table 5.1 Summary of the Environment Parameters.

	Parameter	Value
Video clients	Type	DASH palyer in Docker container.
	Count	$\sim \mathcal{U}\{1, 4\}$ instances per container.
	ABR Algorithm	{Buffer-based, Throughput based, and Fixed-rate}, randomly selected for every session.
	Arrival	$\sim \mathcal{E}xp(1/60)$.
	Duration	$\sim \mathcal{U}\{15, 45\}$ time steps.
Routers	Type	Cieco IOSv images.
	Routing	MLPS tunnels with ACL for each tunnel.
	Link bandwidth	Implemented as Layer-3 policy of 30 Mbps on egress ports.
	Impairments	Delay $\sim \mathcal{N}(0.2, 0.75)$, Jitter $\sim \mathcal{N}(0.05, 0.2)$, Packetloss $\sim \mathcal{N}(0.01, 0.1)$.
Network Issue	Transient	Rate $\sim \mathcal{E}xp(1/30)$ for Duration $\sim \mathcal{U}\{1, 3\}$ time steps.
	Persistent	Rate $\sim \mathcal{E}xp(1/60)$ for Duration $\sim \mathcal{U}\{15, 45\}$ time steps.
	Recurrent	Rate $\sim \mathcal{E}xp(1/90)$ for Duration $\sim \mathcal{U}\{3, 6\}$ time steps.
MDP	Transition time	15 seconds from state to hidden state + 15 seconds from hidden state to next state.

occasional congestion, and we randomly introduced router issues, as listed in Table 5.1. All three ASs used OSPF, and we utilized MPLS tunnels and ACL per path. In total, we had 3 paths, 5 links, and 4 clients, each running multiple instances of Dash.js to ensure that links had sufficient traffic.

Regarding the DASH video clients, we compiled a diverse set of videos to account for differences in scene characteristics, such as high motion in sports or low detail in animations. All videos were encoded using the H.264/MPEG-4 codec at bitrates ranging from 254 to 14931 kbps. The length of each video segment was set to 2 seconds, a common value in DASH applications.

We trained two models with the goal of achieving a performance comparable to that of the expert rules using an SL approach. The expectation was that a successful model should be able to mimic the expert rules, but not necessarily outperform them in terms of cumulative long-term rewards. This is because in the SL framework, the

model attempts to replicate the behavior of its teacher, even if the teacher makes suboptimal decisions.

We tested two neural network models in our study to achieve a comparable performance to expert rules using a SL approach. The first model we tested was a Self-normalized feedforward Neural Network (SNN) that served as the policy network (π-Net) responsible for taking actions. This model was formulated as a multi-class classification problem, where the different possible classes were the actions. The network had a compact size with three hidden layers of size [input=18, 128, 64, 64, output=12] neurons. The input vector represented observations, including a set of metrics for each of the three tunnels, consisting of six metrics such as QoS metrics (Delay, Jitter, and Packetloss), QoE metrics (Bitrate, Buffer), and other parameters such as the total number of clients in each tunnel. We stacked 8 frames of past metrics and fed them to the network to improve its performance. The network had a total size of around 32k trainable parameters.

The second model we tested was a simple Recurrent Neural Network (RNN). While the SNN closely matched the network used in the RL setup, it was not well equipped to model time-series data. Therefore, we also trained an RNN model to eliminate any bottleneck that could be caused by the inefficiency of the SNN in modeling time-series data.

We collected around 50 hours of data at a rate of three examples per minute by running the baseline algorithm on GNS3. The actions taken by the baseline algorithm were used as labels for the collected dataset, which we split into training, validation, and testing sets. We then performed data augmentation on the training dataset using our simulator environment, as illustrated in Figure 4.6. The percentage of each action taken is depicted in Figure 5.1.

Results

In Figure 5.2 (right), we see that the agent learned to classify most actions correctly (i.e., confusion matrix is almost diagonal). However, some of the confusion between actions can be attributed to the fact that two different actions could lead to same end result. For example, if all 3 paths were occupied and a network issue appeared on the link R1-R2, then rerouting clients from Path 1 to either Path2 or 3 has the exact same cost. This is true because

- both tunnels have same number of hops. As a result, the costs of carrying the traffic in those paths are equal.

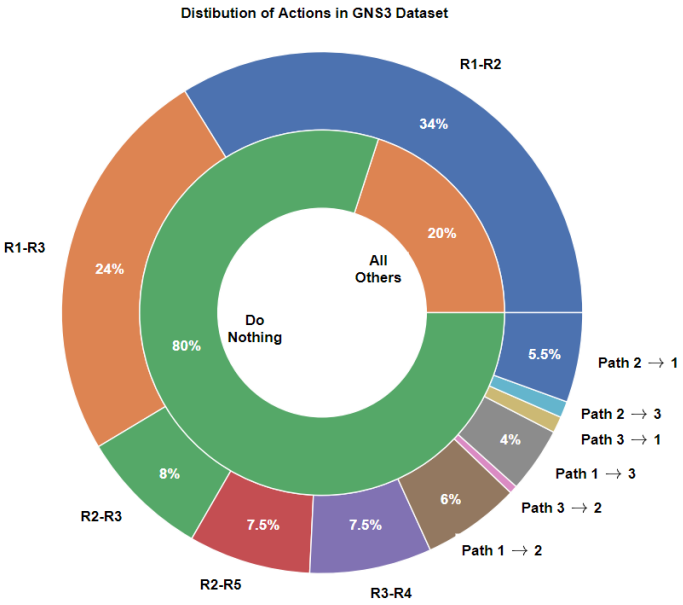


Fig. 5.1 Distribution of different classes withing GNS3 dataset. Inside: overall percentage. Outside: minority class breakdown.

- the broken link affects only Path 1. Therefore, rerouting to either Path 2 or 3 will help the clients to stream videos at a good quality.
- since Path 3 is already occupied, the extra cost for using this tunnel is already paid. In this case, rerouting new clients to this path doesn't incur additional costs compared to Path 2.

To address this issue, we allowed the SL model to pick the top two best actions. If any of them matches the true label, we declare this action as correct. Figure 5.2(left) is a proof that SL-based ARE system can, to high-extend, match the performance of expert rules.

It is important to note that during the training and testing stages, we took measures to prevent the agent from overfitting. Specifically, we ensured that the traffic pattern was different in each stage. This was achieved in several ways: (i) clients were randomly turned on and off, (ii) when clients were first turned on, they were directed to a random path, and (iii) the bitrates selected by clients' ABR algorithm were always different. Additionally, network issues were randomly introduced, which could result in a change in network topology. For example, if a link was broken, the network topology would change from having three different paths to two or even just one (if the link was shared among two paths). By incorporating these measures, we were able to test the ML agent in new scenarios that it had not previously encountered, and it performed well.

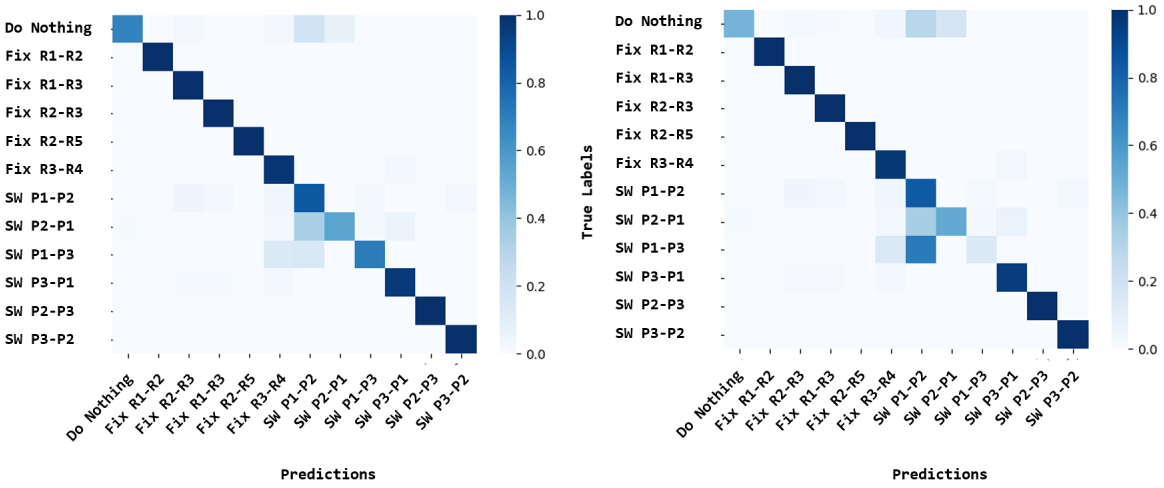
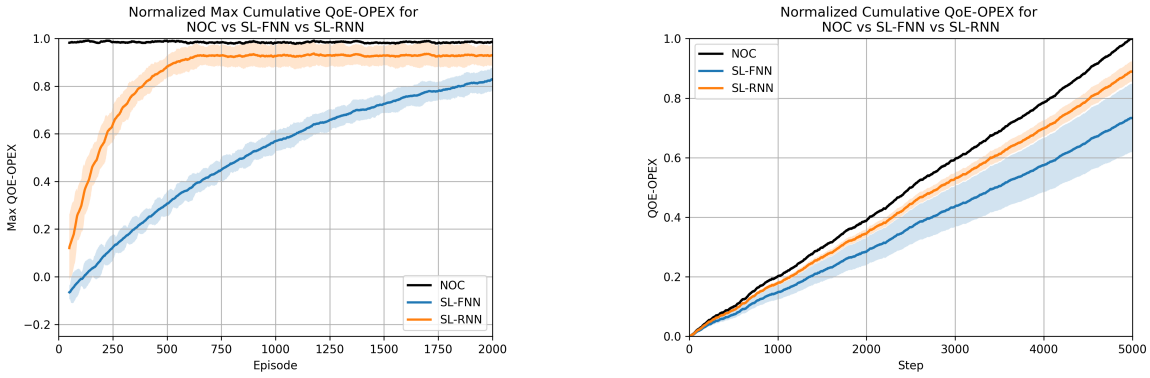


Fig. 5.2 Confusion matrix of SL-based ARE by taking into account top-k actions sorted by their probabilities. (right) $k = 1$ (left) $k = 2$

Next, we study the convergence speed of the SL models. As seen in Figure 5.3(a), both the SNN and RNN models display convergence towards the baseline performance. However, the RNN shows faster convergence, achieving the 90% goal after only 500 episodes, with each episode being truncated at a maximum of 1000 steps. Additionally, Figure 5.3(b) illustrates the cumulative reward gained by both models during a testing phase that lasted for one episode of length 5000 steps. The results demonstrate that both models are capable of achieving high positive rewards, even in highly dynamic network conditions. However, it's worth mentioning that the RNN model is more robust, displaying less variability across different runs. With this, the first test concludes, and we proceed to use RL on the same environment to surpass the expert rules.



(a) Convergence Speed (b) Long-term Reward

Fig. 5.3 SL algorithms can successfully learn to automate NOC.

5.2.2 Outperform Expert Rules

In this section, we leverage the RL framework to train an agent capable of surpassing the baseline algorithm. We compare traditional RL training with Batch-RL to demonstrate that offline datasets can be used to pre-train an RL agent, and subsequently, continue training to exceed the baseline algorithm. This approach allows the agent to benefit from the expert rules, eliminating the initial phase of exploration where actions are entirely random. By doing so, we avoid taking any unnecessary risks that could harm the system's performance due to random action selection.

Environment Setup

Similar to our previous work on SL, we train two RL agents from two different families of model-free RL algorithm to demonstrate the flexibility of our approach. In particular, we chose a variation of DQN algorithm called Double-DQN, which represents the Q-learning based family of algorithms. As well as A2C algorithm which is part of the policy optimization family. Both networks conists of two NN, one of which is the policy network (π-Net) of size [input=18, 128, 64, 64, output=12] with the ReLU activation function. The other NN in Double-DQN algorithm is an exact copy of the policy network. It servers as a reference NN to stabilize the learning. On contrary, the other NN in A2C (called V-Net) consists of three layers of size [input=18, 256, 128, 18, output=1] and is used to predict the value-function. This NN doesn't share weights with the π-Net in order to allow the two networks to update their weights at different rates according to how difficult the tasks they try to model. Note that once the model is trained, only the policy network is needed in production to perform inference.

Regarding the other hyper-parameters, we set the discount factor to $\gamma = 0.99$, and used actor and critic learning rates of $\alpha = 10^{-4}$ and $\alpha' = 10^{-3}$, respectively. Additionally, we controlled the entropy factor to decay from 1 to 0.1. As outlined in §4.2.1, we assigned a reward of +5, 0, and -5 to high, medium, and low QoE levels, respectively, for the QoE model. Furthermore, we considered the running costs of carrying traffic inside AS1 to be cheaper than passing through external AS3. Consequently, we set the cost of the shortest and longest paths in AS1 to 2 and 3, respectively, while going through AS3 had a cost of 5. Finally, the cost of "doing nothing," "rerouting the traffic," and "fixing a link" were set to 0, 2, and 5, respectively.

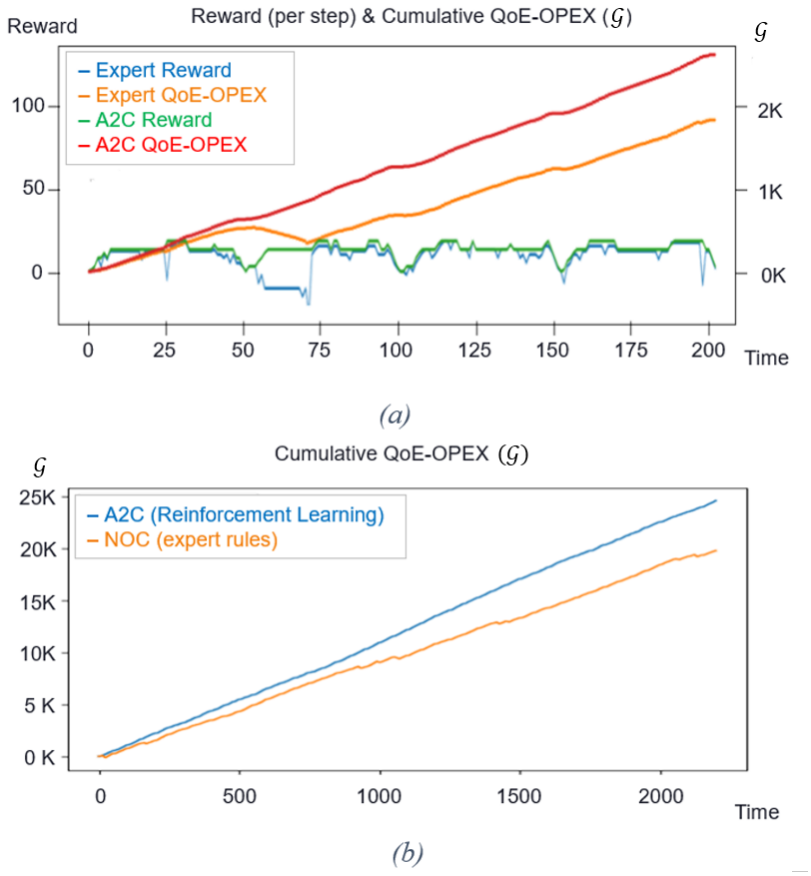


Fig. 5.4 ARE pre-trained using synthetic data on the Simulator Environment (not shown here) and tested on the GNS3 Environment. (a) Comparing both reward and gain. (b) testing the stability of ARE for 18 hours.

Results

Figure 5.4a shows the performance of ARE (A2C) versus that of the expert rules. Here, the agent was pretrained using synthetic data on the Simulator Environment and then tested on the GNS3 Environment. During pretraining, the agent required approximately 500K steps (4167 hours in real time) to outperform expert rules, but this was accomplished in about 6 minutes on the simulator because training on the simulator is much faster than real time. Once this pretrained ARE was deployed in GNS3, we can see in Figure 5.4a that it immediately had better and ever-increasing *Gain* performance while also its Reward was never worse than expert rules' at any point. But can ARE keep up its performance and not collapse? To answer that, we ran the same agent in GNS3 for 18 hours. The results are shown in Figure 5.4b, where we can see that ARE clearly maintains its stability and archives superhuman performance.

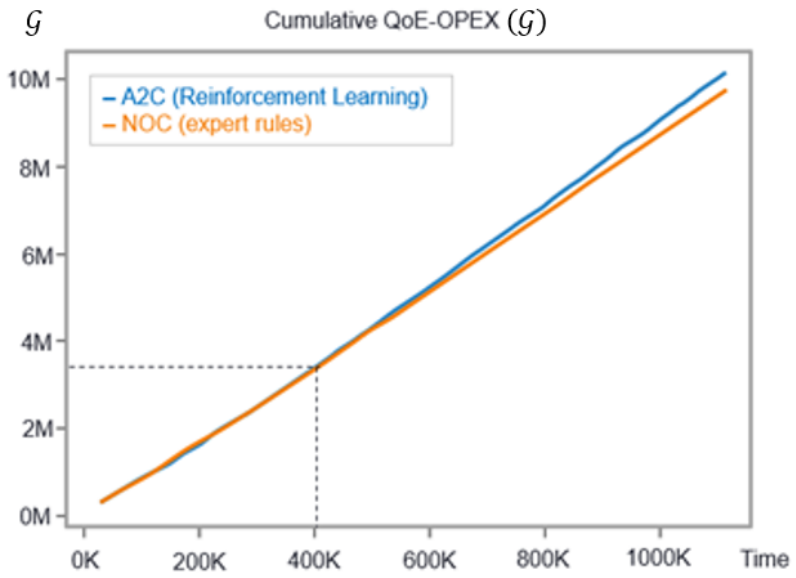


Fig. 5.5 ARE pre-trained on the Batch-RL environment using labelled data and tested on the Simulator environment.

Similarly, results presented in Figure 5.5 demonstrate that pretraining an ARE using Batch-RL with an offline dataset generated by the expert rules on the Simulator Environment is a viable alternative to RL in the real network. The agent's performance is evaluated in the Simulator Environment by allowing it to apply only the rules learned during pretraining for the first 400K steps and then allowing small non-zero exploration from step 400k onward. The Simulator Environment is chosen over the GNS3 Environment because the former is much faster for training, and the goal is to allow ARE to continue training/exploring. As expected, ARE progressively outperforms the expert rules and never falls behind them at any point, demonstrating again that ARE is able to improve and learn new rules by itself.

To gain insight into how the RL agent outperformed the expert rules, we analyzed the network issues and the actions taken by both the RL agent and the expert rules. The complete interactions between the RL agent and the environment are presented in Figure 5.6, while those between the expert rules and the environment are shown in Figure 5.7. Subplot A in both figures shows the true root-cause of network issue in terms of time and location. For example, in the span of the 200 time steps, 20 network issues where introduced, which are indicated by the blue spikes. This information is not available to the RL agent since it is part of the true state of the environment. In order to simplify the analysis, we allow one network issue to be presented in the

environment at any certain time. Therefore, whenever a new issue is introduced, the previous issue is cleared out first.

The dashed line indicates the status of the link since the problem first appears. For example, there was an network problem on the link connecting the routers R1-R3 that lasted for about 5 time steps and was not resolved. This is indicated by the fact that the dashed line value is not None. If the agent didn't take any actions, (see subplot (B)), or took an action that doesn't resolve the issue (e.g., attempting to fix the wrong link or choosing to reroute traffic instead), the problem will persist for a certain number of time steps depends on its type (i.e., transient, persistent, etc.). Once this time is up, the problem is cleared automatically and new issue is introduced. By inspecting the number of actions taken by the RL agent and baseline algorithm, we see that the RL agent is taking less actions, allowing some problems to persist. As we will discuss next, this strategy is the result of including the OPEX into the reward signal. By doing this, the agent learns to postpone fixing problems that doesn't have immediate negative effect on the performance.

Subplot (H) represents the distribution of clients among the different tunnels. We create traffic by allowing four clients to start/stop video streaming at different times. In this scenario, we chose a Sinusoidal-like pattern where the network starts with no active clients. Then, the number of clients joining the network increase, and for a certain period of time, the agent is responsible to optimize the QoE of the four clients under different network conditions. Finally, clients disconnect gradually and the cycle repeats. Note that when a client is first introduced, it is assigned to a random tunnel. Similar to network issues, this behaviour is reproduced for both the RL agent and the baseline algorithm by manually setting the random generator initial seed. However, since the agent is allowed to reroute the traffic, we see different patterns in Figure 5.6(H) and Figure 5.7(H).

Subplots 5.7(C-G) show the E2E metrics that are provided to the agent as part of the observation vector. This is the information that the agent works with to determine the root-cause of any issue shown in subplot (A)

To enhance clarity, we will reuse and emphasize specific sections of Figure 5.6 and 5.7 in the subsequent three figures to illustrate the three main strategies employed by the RL agent. These strategies resulted in fewer but more effective actions, as demonstrated below.

1. The RL-agent learned to ignore any network issue that is not likely to affect any client. Such cases include problems in empty paths as well as transient problems. This can be seen in Figure 5.8 around $t = 25$ and $t = 100$. As we can see in (a),

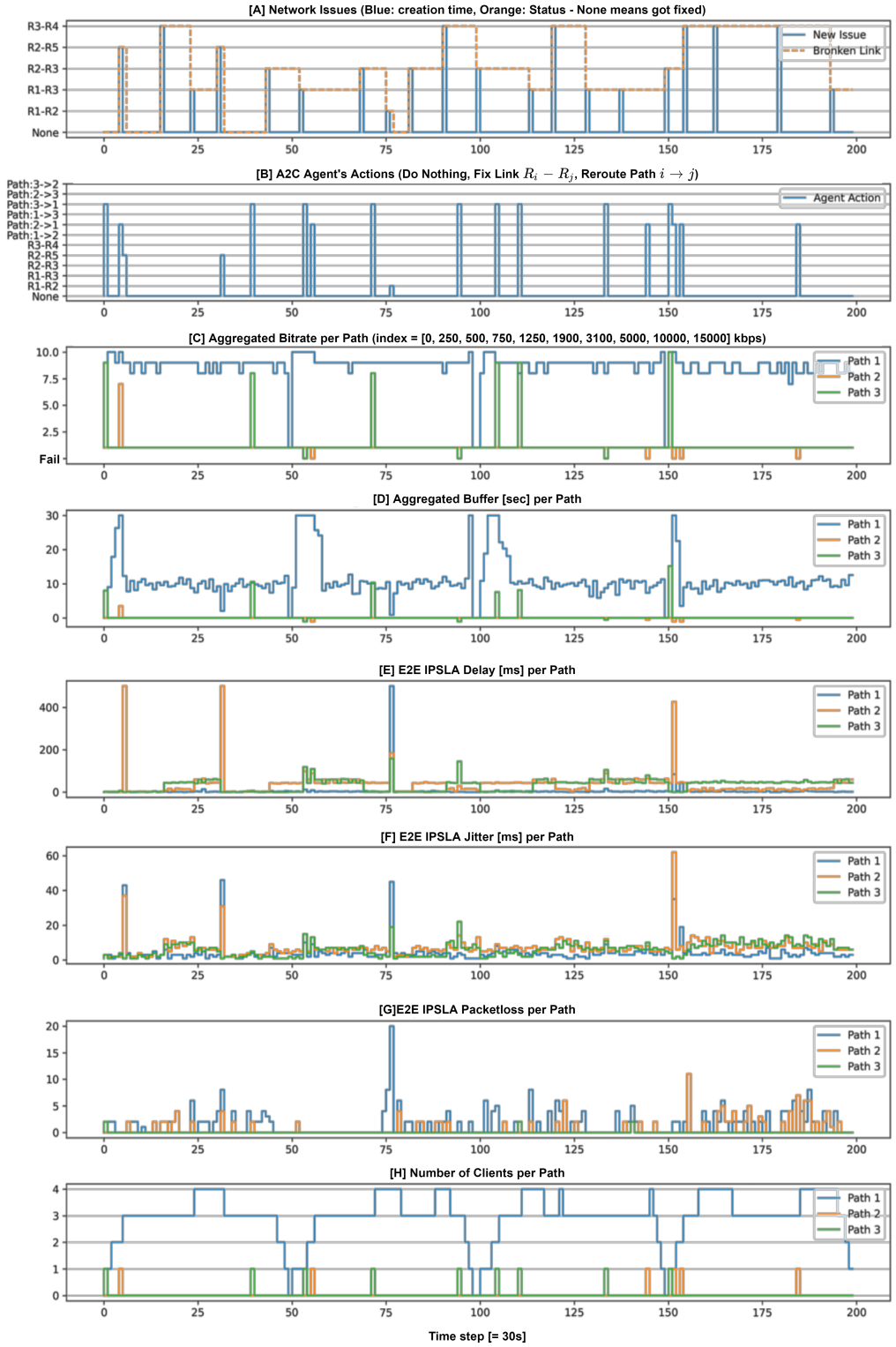


Fig. 5.6 Detailed performance of A2C algorithm tested on GNS3.

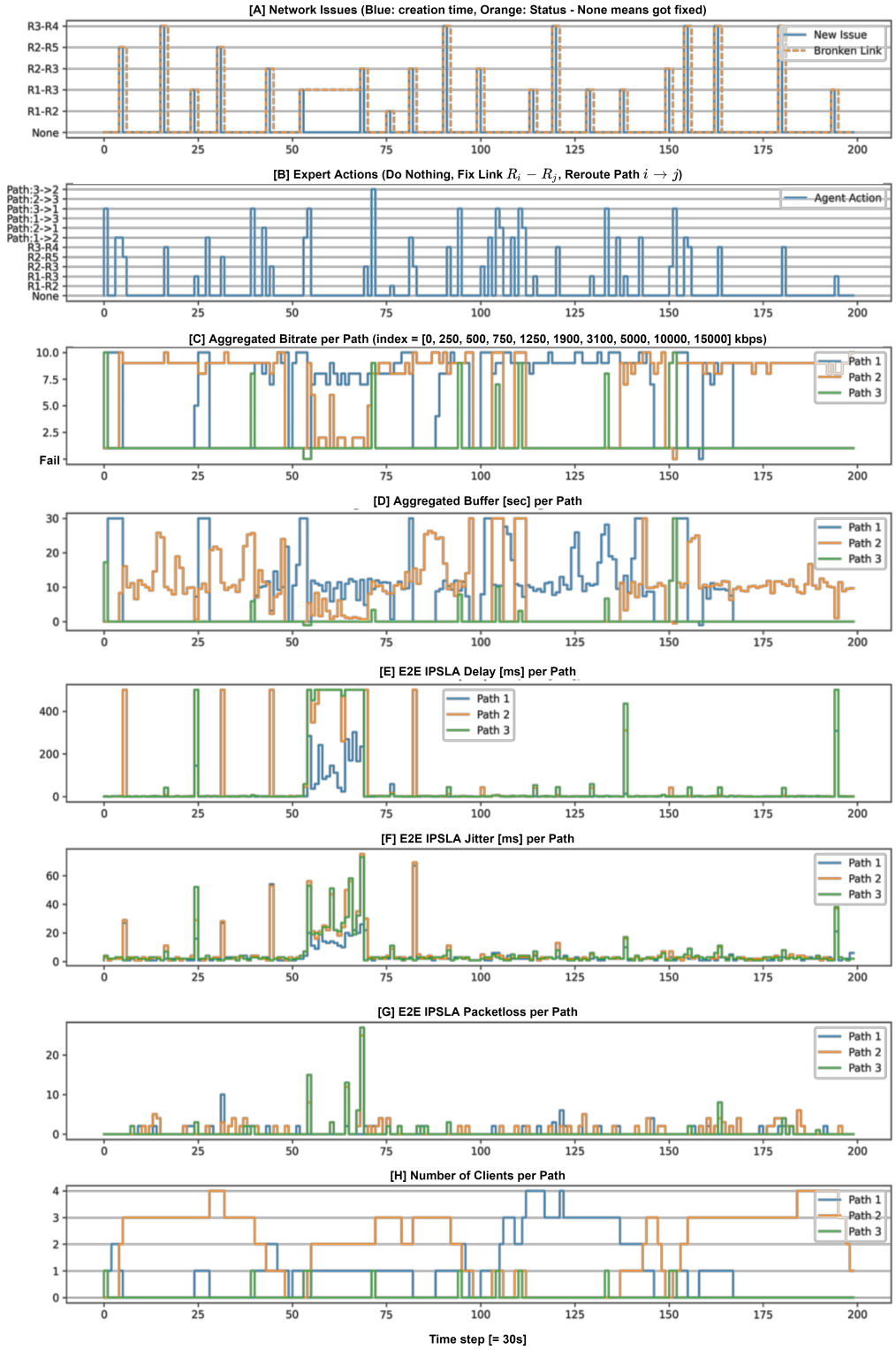


Fig. 5.7 Detailed performance of Baseline algorithm tested on GNS3.

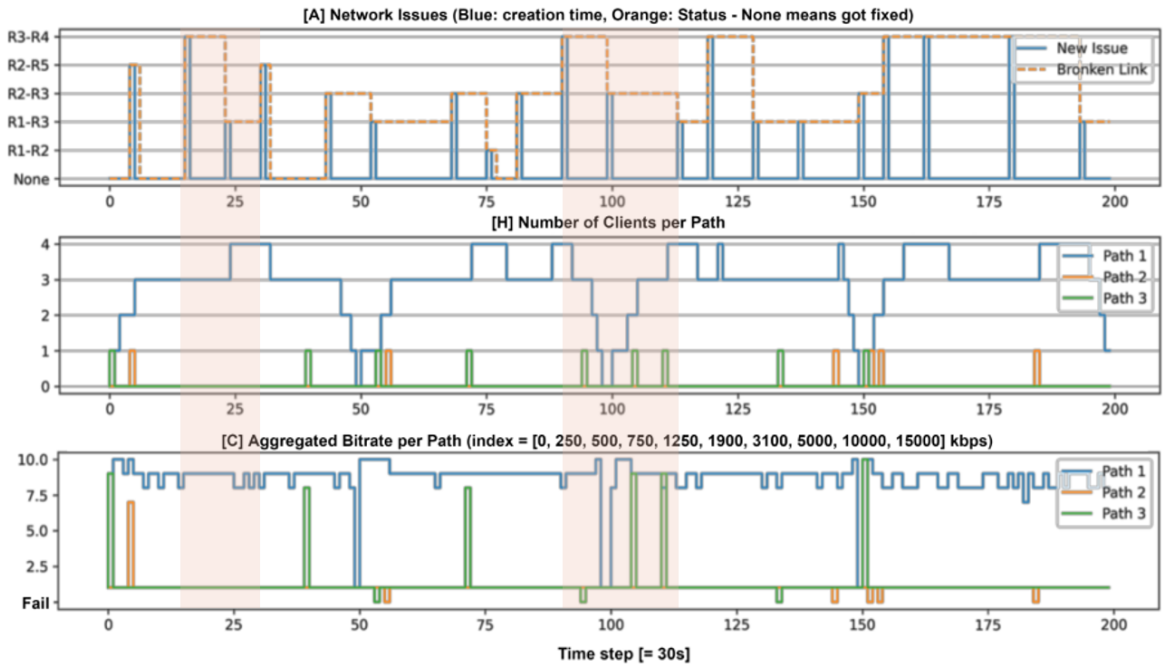


Fig. 5.8 RL agent learned to ignore issues that don't affect clients in order to minimize OPEX.

two network issues affected Path 3 (R1-R3-R4-R5) around $t = 25$: first at the link connecting R3-R4 and then at R1-R3 (see blue line). Meanwhile, all clients were on path 1 as seen in (h). We can verify that no client was affected by observing (c) which shows that all clients were streaming at the highest bitrate. As a results, the agent ignored those issues and didn't attempt to fix them (orange line in (a) was not set to *None* indicating that the issue persisted).

2. It learned that it is better to pack all traffic on path 1 (R1-R2-R5) as much as possible, instead of load-balancing between path 1 and path 2 (R1-R3-R2-R5), because path 1 has fewer hops (lower cost). While on the surface one might think that packing all traffic into one path can cause congestion, the DASH algorithm inside the clients does tend to lower the video bitrate when it senses reduced available bandwidth, and this avoids congestion to some extent. ARE learned that it only needs to move traffic out of path 1 if clients are no longer able to maintain high QoE. In our tests, the link capacity allows all 4 clients to simultaneously achieve high QoE only if all clients are at one bitrate: 3134 kbps; anything above that bitrate will result in congestion, and anything below that bitrate will result in a significant drop in the reward signal. It is remarkable that ARE learned to "live dangerously" by allowing all clients to be in path 1 as long

Fig. 5.9 RL agent achieved better service quality while taking fewer actions.

as their QoE is high and maintained. This is illustrated around $t = 50 - 75$ in Figure 5.9(b) where expert rules cause much fluctuations in the buffer size while ARE maintains the buffer sizes with more stability using fewer actions.

3. It paid particular attention to links that are shared between different paths. For example, link R2-R5 is shared among paths 1 and 2, while link R1-R3 is shared among paths 2 and path 3. As shown in Figure 5.10(a), ARE immediately set the orange line to *None* (i.e. normal state) every time an issue affect R2-R5 and never left the link broken. Furthermore, we can also see that ARE also rushes to reroute any client in Path 3 to avoid the costs of using AS3 infrastructure.

It is interesting to note that none of the above strategies were part of the expert rules; ARE learned them by itself, validating our choice of using RL for complicated

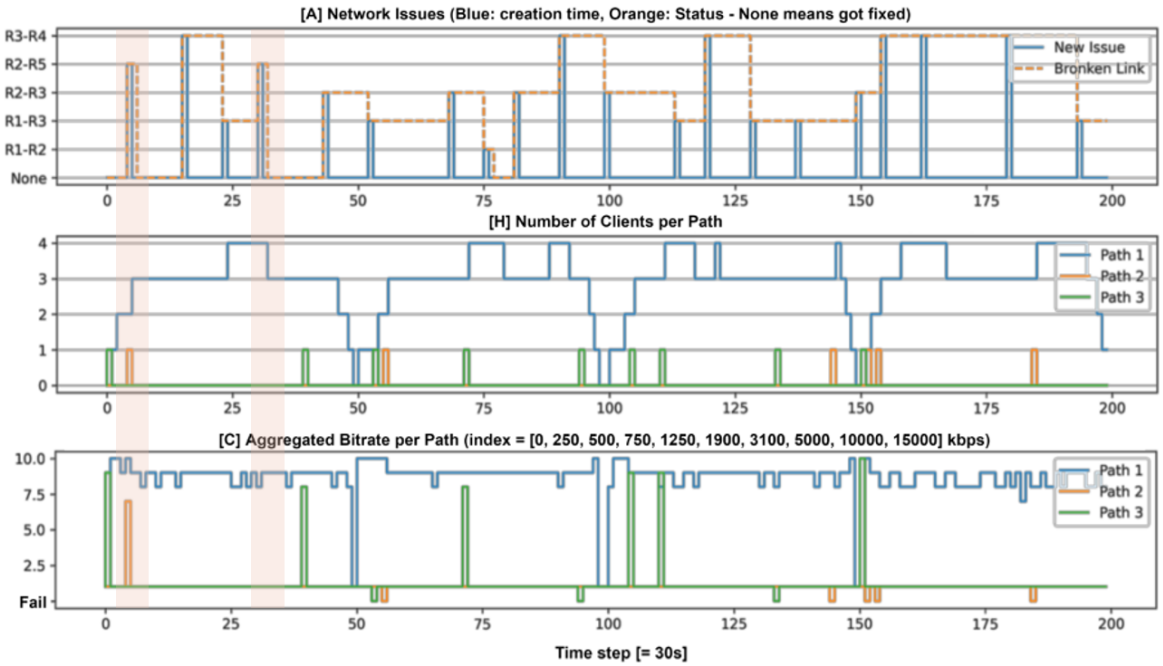


Fig. 5.10 The RL agent learned to quickly address any issues with links that are common to multiple paths.

networks. This demonstrates that ARE can indeed progress beyond its initial training to achieve exceedingly high performance.

So far, we have been focused on designing an algorithm that achieves the highest possible reward. However, it is also important to study the speed of convergence of such algorithms. To investigate the impact of network size on the convergence speed of RL algorithms, we chose to work with A2C algorithm to train various NN and varied the number of layers and width of each layer. We selected the A2C algorithm in this experiment due to its efficiency compared to Double-DQN, but the findings here are applicable to other algorithms as well.

Figure 5.11 compares the speed of which the QOE-OPEX metric increase for A2C algorithms with different network sizes. One interesting observation is that agents who have shallow value-network ($v_\pi = [256]$) and medium policy network ($\pi = [64, 64]$) seems to learn faster than others for the first 20K steps. One major obstacle that RL agent has to face is the high fluctuation in the reward signal, caused by the adaptive bitrate of DASH clients. The value network serves as a reward function approximator, but suboptimal behavior by DASH clients can result in huge fluctuations in the reward signal, making it difficult for the agent to plan. Figure 5.12 illustrates the difficulty of predicting the reward signal, as client bitrate on path 3 fluctuates rapidly between the lowest and highest bitrates around $t = 90 - 100$, leading to frequent traffic rerouting

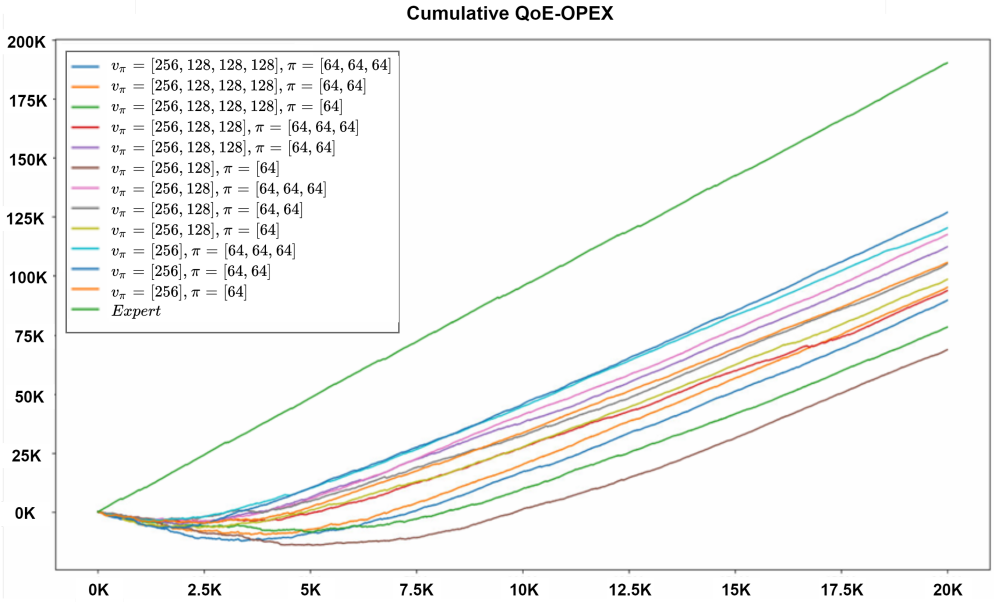


Fig. 5.11 Convergence speed of A2C algorithm for 20K steps on Simulator Environment.

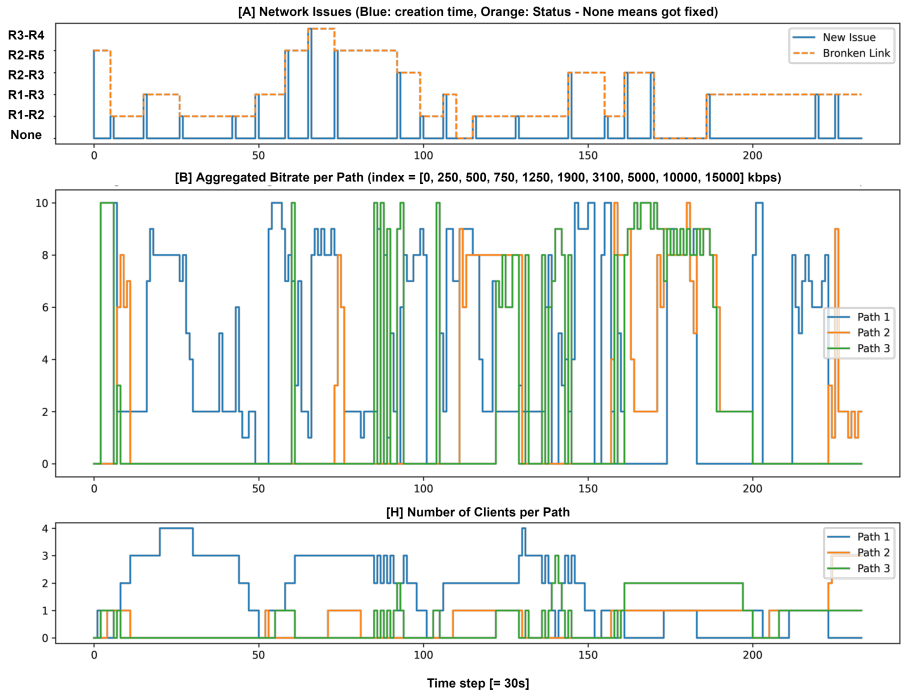


Fig. 5.12 DASH behaviour and the reward function. Baseline algorithm is running on GNS3

Table 5.2 Effect of NN size on convergence speed. Gain is measured at 500K steps and normalized relative to Expert gain.

10 Models			Best Model (10 runs)	
V-Net	Pi-Net	Gain	test	Gain
[256,128,18]	[128,64,64]	1.259	1	1.331
[256]	[64]	1.131	2	1.318
[256]	[64, 64]	1.141	3	1.318
[256]	[64, 64, 64]	1.152	4	1.320
[256]	[128]	1.142	5	1.319
[256, 128]	[64]	1.316	6	1.330
[256, 128]	[64, 64]	1.137	7	1.320
[256, 128]	[128, 64]	1.134	8	1.333
[256, 128]	**[128]**	**1.331**	9	1.321
Expert	-	**1**	**Avg**	**1.323**

between paths 1 and 3. To address this, collaboration between clients and the RL agent may be necessary, as described in [1].

The results in Table 5.2 demonstrate that the A2C algorithm consistently outperforms expert rules in terms of long-term performance. Interestingly, we found that training the critic network is more challenging than training the actor network, which is reflected in the need for deeper and wider value networks compared to policy networks. This underscores the importance of accurately evaluating the situation for effective RL, and once the critic network achieves this, it becomes easier for the actor network to respond appropriately. However, it is important to note that the initial fast convergence achieved by the shallow value network and medium-size policy network does not continue and falls behind by 15% after 500K steps, as shown in Figure 5.11. In contrast, the models with larger value networks and simpler policy networks seem to be the winners in the end.

5.2.3 Transfer Learning

Environment Setup

In this section, we evaluate the performance of the RL agent on the large network depicted in Figure 4.2. In this setup, the multiple agents are trained to control portions of the network in a distributed fashion. The agents don't assume any knowledge of the network topology and can take corrective actions at the tunnel level, such as rerouting

traffic between tunnels and switching to a backup server. We simulate the network in the DES environment developed in §4.2.2. The DES parameters are listed in Table 5.3.

The network consists of 33 routers of which 24 are dedicated to serve the video clients (called Edge routers). This simulates the hierarchy in ISP network. We configure each edge router as follows:

- There are 4 tunnels representing a tunnel group.
- There is one RL agent responsible for controlling the tunnel group.
- There is an edge server located at a distance of at most 3 hops away. The path to this server is the considered to be the main tunnel.
- There is a backup tunnel that has 50% or less overlap with the main tunnels.
- There are variable number of active clients connected through this router.

In terms of the training algorithm, we compares three groups of algorithms trained on on a subset of small networks that are not part of the large network:

1. Train NN using SL.
2. Train NN using traditional (online) RL
3. Pre-train NN using Batch-RL first.

For each of the aforementioned algorithms, we compared the performance of the following variations:

1. Test the NN as is (a.k.a zero-shot transfer learning) against expert rules.
2. For SL: perform a fine-tuning step of each agent on its small network. For RL: allow each agent to continue exploration on its small network.
3. For RL: (i) use a single shared agent during the training step then make independent copies of the agent during testing. (ii) use multiple agents during the training each learning from its local network.

   Finally, to compare effectiveness of SL against Batch-RL, we also train the NN using two versions of the expert rules: an optimal and non-optimal versions.

Table 5.3 Summary of the DES Parameters.

	Parameter	Value
Video clients	Count	1000.
	ABR Algorithm	BOLA Algorithm.
	Buffer	30 seconds
	Arrival	$\sim \mathcal{E}xp(1/60)$.
	Duration	$\sim \mathcal{U}\{15, 45\}$ time steps.
	Video segment	0.2 second (packet size = segment Bitrate).
Routers	Type	Core and Edge
	Count	9 Core and 24 Edge.
	Routing	4 Tunnels of length 2-6 hops.
	Link bandwidth	30 Mbps on egress ports.
	Clients	30-200 active client at a time.
	Impairments	Delay $\sim \mathcal{N}(0.2, 0.75)$, Jitter $\sim \mathcal{N}(0.05, 0.2)$, Packetloss $\sim \mathcal{N}(0.01, 0.1)$.
Servers	Type	Cloud and Edge
	Count	1 Cloud and 5 Edge.
	Capacity	Cloud: ∞ and Edge:100 client (socket).
	Bandwidth	∞.
Network Issue	Transient	Rate $\sim \mathcal{E}xp(1/30)$ for Duration $\sim \mathcal{U}\{1, 3\}$ time steps.
	Persistent	Rate $\sim \mathcal{E}xp(1/60)$ for Duration $\sim \mathcal{U}\{15, 45\}$ time steps.
	Recurrent	Rate $\sim \mathcal{E}xp(1/90)$ for Duration $\sim \mathcal{U}\{3, 6\}$ time steps.

Results

The results of automating the large network using Multi-agent ML approach are summarized in Table 5.4. The table shows that RL can outperform the baseline algorithm by up to 28%, and with fine-tuning, this improvement can match the performance on the small network, resulting in a 33% improvement over the baseline algorithm.

The table highlights another important finding: Batch-RL significantly improves training efficiency by 16% compared to traditional RL, assuming a nearly optimal

Table 5.4 Summary of Multi-agent NOC Automation Results.

Configuration			Results				
Model	Training	Shared Agent	Zero-shot Gain	$T2T_{90\%}$	$T2T_{130\%}$	Success Rate	Fine-tuning Gain
SNN	SL	Yes	0.76 $_{\pm 0.09}$	2.24M	-	89%	.86 $_{\pm 0.03}$
RNN	SL	Yes	0.87 $_{\pm 0.06}$	763K	-	93%	.93 $_{\pm 0.03}$
DQN	RL	Yes	0.98 $_{\pm 0.04}$	832K	-	94%	1.05 $_{\pm 0.02}$
	RL	No	0.85 $_{\pm 0.17}$	1.09M	-	90%	-
	BRL	Yes	1.06 $_{\pm 0.03}$	649K	1.27M	96%	1.11 $_{\pm 0.03}$
A2C	RL	Yes	1.12 $_{\pm 0.08}$	587K	732K	95%	-
	RL	No	1.02 $_{\pm 0.14}$	761K	832K	92%	1.24 $_{\pm 0.02}$
	BRL	Yes	1.28 $_{\pm 0.05}$	476K	534K	99%‘	1.33 $_{\pm 0.02}$

teacher. This improvement is also evident in the reduced variance among different runs. One possible explanation for this is that the teacher algorithm controls the exploration process. Another factor that affects the variance among agents is that if agents are trained independently, they may not have enough time to explore the entire search space. However, if agents share a centralized buffer, a master agent can be trained on all the experiences collected by the worker agents. This allows one agent to experience a diverse set of sub-networks. During testing, the agent can be replicated multiple times, and each copy can be fine-tuned independently on its tunnel group if necessary.

In addition, we present the Time-to-Threshold (T2T) and success rate metrics. The former represents the number of steps required for agents to reach 90% of the baseline performance, and the later represent the percentage of agents (in the multi-agent framework) that were able to cross the 90% threshold during testing. Those two metrics serve as a reliable indicator of the sampling efficiency of variance algorithms. Although all models achieved comparable performance to the baseline algorithm, both SL and RL with no shared exploration struggled to exceed the 90% threshold within 750K steps. This can be attributed in part to the limited capacity of the neural network, which reduces the efficiency of data collection and negatively impacts overall rewards. Notably, the SNN model performed worse than the RNN model, and the DQN model performed worse than the A2C model.

Among all combinations of hyper-parameters, we found that training a cenrtilized A2C agent using Ratch RL, then perform fine-tuning to yeald the best resuts overall. This is expected because of the following factors:

- RL is better than SL because it can utilize the sub-optimal actions taken by the teacher network.
- The use of a centralized RL learning agent for training improves performance without compromising training and testing speed since all agents contribute to data collection and testing.
- The utilization of Batch-RL for training not only enhances convergence speed but also minimizes variance in agents' performance. This is attributed to the ability of the agents to learn useful strategies from expert actions, which would otherwise take longer to discover

Figure 5.13 depicts the cumulative reward achieved by agents during testing on the large network. The agents were derived from a master agent trained using Batch-RL on the small network. The figure clearly shows that all agents were able to manage the networks effectively without any fine-tuning, achieving a 100% success rate. In contrast, if Batch-RL is not utilized, some agents may learn policies that are not useful in all network conditions. This is illustrated in Figure 5.14, where approximately 50% of the agents struggle to perform well in the vicinity of Servers E1, E2, and E5. We argue that this weak performance is not related to the network topology of these sub-networks, as we used a topology-agnostic formulation for the training process. To support our claim, one can examine the agents' performance in the E3 neighborhood, which is similar to E01 and E2. Similarly, agents in the vicinity of E4 perform well, and the topology of this sub-network is similar to the one around E05.

We will conclude this section by demonstrating a case that highlights some of the limitations of the Batch-RL algorithm. Specifically, we used a sub-optimal teacher to train RL agents using Batch-RL, and the results are shown in Figure 5.15. As expected, many agents failed to perform well on the large network, mainly because during training, they did not have sufficient opportunities to explore the best actions. However, we observed that some agents managed to learn a useful policy and achieved good performance. This phenomenon can be explained as follows: unlike supervised learning (SL), Batch-RL does not necessarily try to mimic all the actions taken by the expert, since the RL agent can observe the reward associated with these actions. In other words, the RL agent can differentiate between good and bad actions by inspecting the rewards associated with them. In contrast, an SL agent considers all labels as optimal actions.

However, if an RL agent is trained using a sub-optimal teacher, it can fail due to the lack of alternative actions. If the agent tries a different action, it may still receive

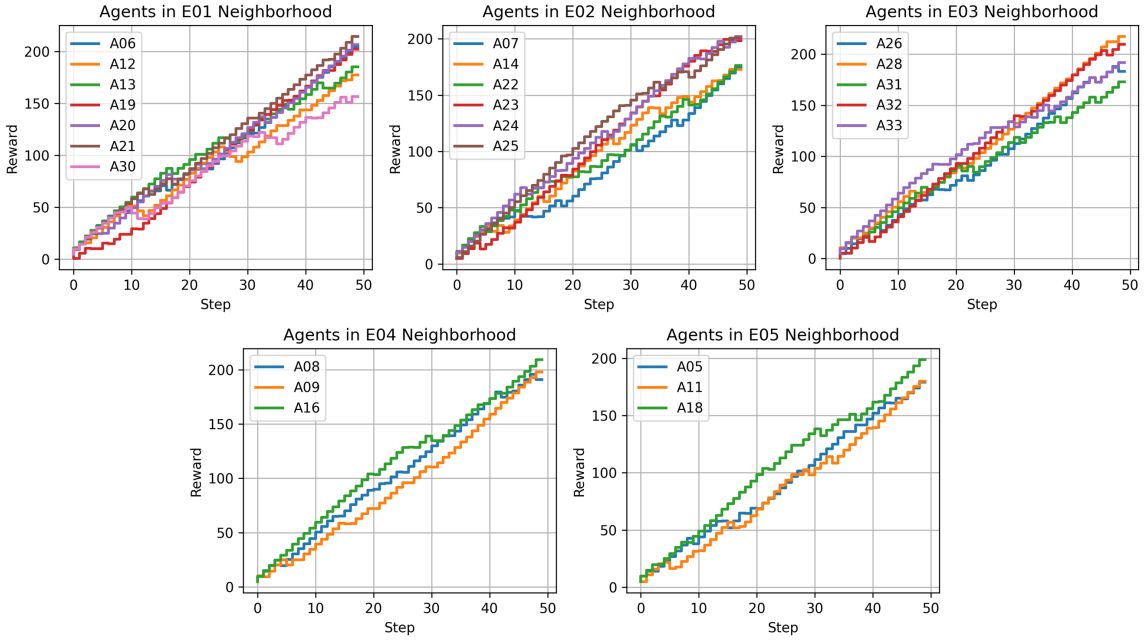


Fig. 5.13 Performance of different RL-agents on the large network. Training done on various small networks in DES environment using Batch-RL.

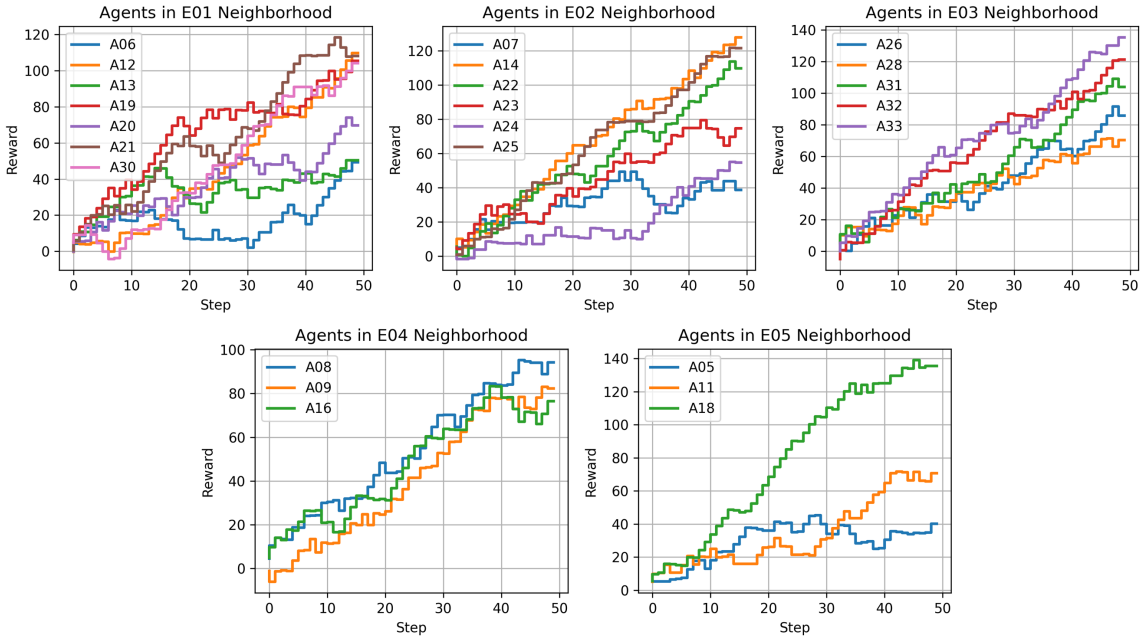


Fig. 5.14 Performance of different RL-agents on the large network. Training done on various small networks in DES environment using online RL.

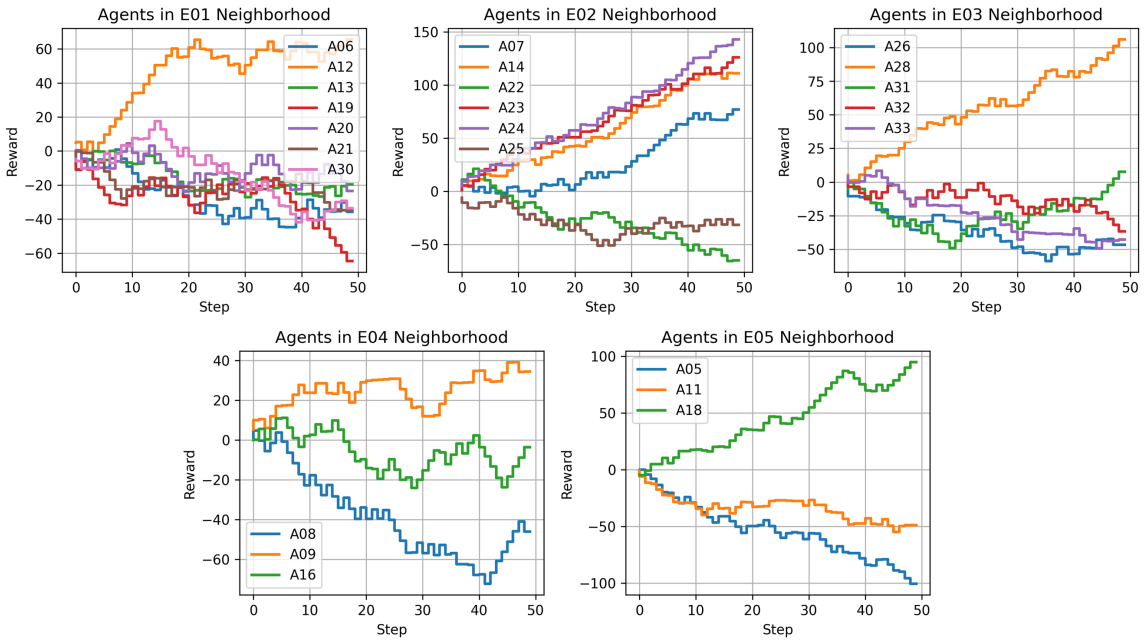


Fig. 5.15 Performance of different RL-agents on the large network. Training done on various small networks in DES environment using Batch-RL with suboptimal expert-rules.

a bad reward. Therefore, it is important to use an expert with a good performance to train RL agents using Batch-RL, so that the agents can learn from a variety of good actions and avoid getting stuck in sub-optimal policies.

5.3 Scalability

In terms of scalability, we argue that our proposed algorithm for ARE is scalable for the following four reasons:

1. The used RL algorithm is scalable: as mentioned before, there are many examples of A2C algorithm deployed in large scale such as [4][26]. This is mainly because the NN in Actor-critic algorithms tend not to be very deep (which is very beneficial for inference), and it can be trained in a multi-agent setting where agent can be trained in a decentralized fashion and synchronize their experience together.

2. The network topology we chose, despite its small size, is not trivial and it includes many advanced concepts that won't change with scale. For example, the agent will experience (i) a network with multiple paths, (ii) with different path lengths i.e. number of hops (iii) some links are shared between (and will affect) multiple

paths (iv) clients randomly join (or drop from) streaming sessions to generate different traffic patterns (v) client stream a real video traffic at adaptive bitrates (vi) some traffic can be rerouted through an external AS which can cost more.

3. Our approach to dividing a large, complex network into smaller tunnel groups is independent of the network's topology, and offers two significant benefits. Firstly, it simplifies the training process. The creation of a dataset that includes all possible subnetwork topologies can be an incredibly expensive and time-consuming task, requiring a significant amount of planning and preparation. However, our topology-agnostic approach eliminates the need for this, reducing the complexity of the training process and making it much more streamlined and efficient. Secondly, our approach also simplifies the deployment process. By reducing the amount of planning and decision-making required when breaking down the network, our topology-agnostic approach streamlines the deployment process, making it easier and more efficient to implement. This can save valuable time and resources, allowing for a more effective and efficient deployment of the network.

4. The pre-training step is very fast. As we mentioned earlier, the agent can experience more than 4000 hours of real time in about 6 minutes.

Chapter 6

Future Work

We studied the problem of automating ARE for NOCs and how to allow RL algorithms to safely discover new rules that can outperform human experts. In this concluding chapter, we summarize the contribution and achieved results and show possible directions for future research in this area.

6.1 Summary of the Results

We showed that using RL, it is possible to automate NOC operations with super-human performance, and this is significant for building autonomous and self-healing networks. We also showed that training such RL systems is practical, because it can be trained orders of magnitude faster than real time in either simulators or offline, without disturbing normal operations of a live network. We summarize our contributions as follows:

- we formulated the problem of ARE in NOC as MDP process and solve this problem using RL exceeding human performance with expert rules.
- we proposed an objective function that goes beyond QoS-based goals to include ISP's OPEX and clients' QoE. Our RL algorithm was trained to anticipate network issues that might degrade the users' QoE and to act within the OPEX constraints.
- we designed a training solution with simulation that can train the RL model in a matter of minutes compared to thousands of hours if trained in real time. We equipped the simulator with MDP-aware data-augmentation capabilities that

enrich the collected dataset without changing the underlying dynamics of the real environment we are simulating.

- Our solution outperformed the human experts by a large margin of 25% on a challenging task (DASH video streaming) without the risk of making costly mistakes.

6.2 Plan for Future Work

- using our algorithm to perform 5G slicing. This is specially exciting given the similarities between MLPS tunnels and overlay and the network slices.
- training RL-agents to decide on what KPI metrics to collect from the network and at what rate. This is important because collecting all type of possible metrics at high-rate results in huge overhead on the network. On the other side, deciding what KPI to collect depends on the QoE metric we want to optimize, and this metric can change over time.
- extend the multi-agent framework to enable collaboration between agents. This collaboration can be highly beneficial, as it can reduce congestion on links that are shared between multiple tunnels that are controlled by different agents. By working together, agents can coordinate their actions to ensure that the overall network operates more efficiently.

References

[1] Altamimi, S. and Shirmohammadi, S. (2020). Qoe-fair dash video streaming using server-side reinforcement learning. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 16(2s):1–21.

[2] Benzaid, C. and Taleb, T. (2020). Ai-driven zero touch network and service management in 5g and beyond: Challenges and research directions. *IEEE Network*, 34(2):186–194.

[3] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[4] Chu, T., Wang, J., Codecà, L., and Li, Z. (2019). Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1086–1095.

[5] Ciena, I. (2018). Introducing the adaptive network vision. *Ciena White Paper*.

[6] Cisco, I. (2016-2021). Cisco visual networking index: Forecast and methodology. *Cisco White Paper*.

[7] De Vriendt, J., De Vleeschauwer, D., and Robinson, D. (2013). Model for estimating qoe of video delivered using http adaptive streaming. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1288–1293. IEEE.

[8] Deb, S., Ge, Z., Isukapalli, S., Puthenpura, S., Venkataraman, S., Yan, H., and Yates, J. (2017). Aesop: Automatic policy learning for predicting and mitigating network service impairments. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1783–1792, New York, NY, USA. Association for Computing Machinery.

[9] Dey, A. J. and Sarma, H. K. D. (2020). Routing techniques in internet of things: A review. In Sarma, H. K. D., Bhuyan, B., Borah, S., and Dutta, N., editors, *Trends in Communication, Cloud, and Big Data*, pages 41–50, Singapore. Springer Singapore.

[10] Dimopoulos, G., Leontiadis, I., Barlet-Ros, P., Papagiannaki, K., and Steenkiste, P. (2015). Identifying the root cause of video streaming issues on mobile devices. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA. Association for Computing Machinery.

[11] Dragos, R. and Avram, S.-M. (2006). A graph theoretical approach to traffic management in isp networks using the overlay mpls network.

[12] Gallego-Madrid, J., Sanchez-Iborra, R., Ruiz, P. M., and Skarmeta, A. F. (2021). Machine learning-based zero-touch network and service management: A survey. *Digital Communications and Networks*.

[13] Gebremariam, A. A., Usman, M., and Qaraqe, M. (2019). Applications of artificial intelligence and machine learning in the area of sdn and nfv: A survey. In *2019 16th International Multi-Conference on Systems, Signals Devices (SSD)*, pages 545–549.

[14] Gill, P., Jain, N., and Nagappan, N. (2011). Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361.

[15] Gonzalez, J. M. N., Jimenez, J. A., Lopez, J. C. D., and Parada G, H. A. (2017). Root cause analysis of network failures using machine learning and summarization techniques. *IEEE Communications Magazine*, 55(9):126–131.

[16] He, J. and Rexford, J. (2008). Toward internet-wide multipath routing. 22(2):16–21.

[17] Hemmati, M., McCormick, B., and Shirmohammadi, S. (2017). Qoe-aware bandwidth allocation for video traffic using sigmoidal programming. *IEEE MultiMedia*, 24(4):80–90.

[18] IEEE (2021). Ethically aligned design for business. *The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems*.

[19] Kanagavelu, R. and Zhu, Y. (2019). A pro-active and adaptive mechanism for fast failure recovery in sdn data centers. In Arai, K., Kapoor, S., and Bhatia, R., editors, *Advances in Information and Communication Networks*, pages 239–257, Cham. Springer International Publishing.

[20] Kearns, M., Mansour, Y., and Ng, A. Y. (2002). A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine learning*, 49(2):193–208.

[21] Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014.

[22] Lee, Y. L. and Qin, D. (2019). A survey on applications of deep reinforcement learning in resource management for 5g heterogeneous networks. In *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 1856–1862.

[23] Liao, R., Li, Y., Song, Y., Wang, S., Hamilton, W., Duvenaud, D. K., Urtasun, R., and Zemel, R. (2019). Efficient graph generation with graph recurrent attention networks. *Advances in neural information processing systems*, 32.

[24] Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988.

[25] Mammeri, Z. (2019). Reinforcement learning based routing in networks: Review and classification of approaches. *IEEE Access*, 7:55916–55950.

[26] Mao, H., Chen, S., Dimmery, D., Singh, S., Blaisdell, D., Tian, Y., Alizadeh, M., and Bakshy, E. (2020). Real-world video adaptation with reinforcement learning. *arXiv preprint arXiv:2008.12858*.

[27] Mao, H., Netravali, R., and Alizadeh, M. (2017). Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210.

[28] Moy, J. (1998). Ospf version 2. *RFC 2328*.

[29] Raeisi, B. and Giorgetti, A. (2016). Software-based fast failure recovery in load balanced sdn-based datacenter networks. In *2016 6th International Conference on Information Communication and Management (ICICM)*, pages 95–99.

[30] Ross, S. and Bagnell, D. (2010). Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668. JMLR Workshop and Conference Proceedings.

[31] Soldani, J. and Brogi, A. (2022). Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys*, 55(3):1–39.

[32] Spiteri, K., Urgaonkar, R., and Sitaraman, R. K. (2020). Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711.

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

[34] Thantharate, A., Paropkari, R., Walunj, V., and Beard, C. (2019). Deepslice: A deep learning approach towards an efficient and reliable network slicing in 5g networks. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 0762–0767.

[35] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. 30(1).

[36] Vasiloudis, T., Vahabi, H., Kravitz, R., and Rashkov, V. (2017). Predicting session length in media streaming. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, page 977–980, New York, NY, USA. Association for Computing Machinery.

[37] Verstraaten, M., Varbanescu, A. L., and de Laat, C. (2017). Synthetic graph generation for systematic exploration of graph structural properties. In *Euro-Par 2016: Parallel Processing Workshops*, pages 557–570, Cham. Springer International Publishing.

[38] Wang, Q., Alcaraz-Calero, J., Ricart-Sanchez, R., Weiss, M. B., Gavras, A., Nikaein, N., Vasilakos, X., Giacomo, B., Pietro, G., Roddy, M., Healy, M., Walsh, P., Truong, T., Bozakov, Z., Koutsopoulos, K., Neves, P., Patachia-Sultanoiu, C., Iordache, M., Oproiu, E., Yahia, I. G. B., Angelo, C., Zotti, C., Celozzi, G., Morris, D., Figueiredo, R., Lorenz, D., Spadaro, S., Agapiou, G., Aleixo, A., and Lomba, C. (2019a). Enable advanced qos-aware network slicing in 5g networks for slice-based media use cases. *IEEE Transactions on Broadcasting*, 65(2):444–453.

[39] Wang, S., Xu, H., Huang, L., Yang, X., and Liu, J. (2019b). Fast recovery for single link failure with segment routing in sdns. In *2019 IEEE 21st International Conference on High Performance Computing and Communications*, pages 2013–2018.

[40] You, J., Ying, R., Ren, X., Hamilton, W., and Leskovec, J. (2018). Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR.

[41] Zeng, S., Xu, X., and Chen, Y. (2020). Multi-agent reinforcement learning for adaptive routing: A hybrid method using eligibility traces. In *2020 IEEE 16th International Conference on Control Automation (ICCA)*, pages 1332–1339.

[42] Zhang, Y., Xin, J., Li, X., and Huang, S. (2020). Overview on routing and resource allocation based machine learning in optical networks. *Optical Fiber Technology*, 60:102355.