Intelligent Node-Overload Protection in Mobile Edge Computing using Reinforcement Learning

Anirudha Jitani

School of Computer Science, McGill University

Montreal, Quebec, Canada

May 13, 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Masters of Science

Abstract

Mobile Edge Computing (MEC) refers to the concept of placing computational capability and applications at the edge of the network. Running applications and computations closer to the clients provide benefits such as reduction in network congestion, better performance of applications, and reduced latency in handling client requests. The performance of an edge server is adversely affected when it is overloaded, especially if it crashes due to overload and causes service failures. In this thesis, a solution to prevent nodes from getting overloaded is analyzed by introducing an admission control policy. An adaptive admission control policy based on low complexity Reinforcement Learning (RL), called SALMUT (Structure-Aware Learning for Multiple Thresholds), is validated using several scenarios mimicking real-world deployments. This approach performs as well as state-of-the-art deep RL algorithms such as PPO (Proximal Policy Optimization) and A2C (Advantage Actor Critic), but requires an order of magnitude less time to train, and outputs an easily interpretable policy. We also show that SALMUT performs better than a baseline algorithm in a real-testbed we created, which runs actual workloads similar to real-world deployments.

Sommaire

Le Mobile Edge Computing (MEC) correspond au fait de placer des capacités de calcul et applications à la périphérie du réseau. L'exécution des applications et des calculs plus près des clients offre des avantages tels que la réduction de la congestion du réseau, une meilleure performance des applications et une réduction de la latence dans le traitement des demandes des clients. Les performances d'un serveur edge sont affectés négativement lorsqu'il est surchargé, en particulier s'il se bloque en raison d'une surcharge et provoque des pannes de service. Dans cette thèse, nous analysons une solution pour éviter la surcharge des nœuds en introduisant une stratégie de contrôle d'admission. Une stratégie de contrôle d'admission adaptative basée sur l'apprentissage par renforcement (Reinforcement Learning, RL) de faible complexité appelée SALMUT (Structure-Aware Learning for Multiple Thresholds) est validée à l'aide de plusieurs scénarios imitant des déploiements réels. Cette approche fonctionne aussi bien que les algorithmes RL par apprentissage profond de pointe tels que PPO (Proximal Policy Optimization) et A2C (Advantage Actor Critic), mais l'entraînement est un ordre de grandeur plus rapide et produit une stratégie facilement interprétable. Nous montrons également que SALMUT fonctionne mieux qu'un algorithme de base dans un environnement de test que nous avons créé, exécutant des charges de travail réelles similaires à des déploiements dans le monde réel.

I dedicate this thesis to my family and friends for their

unconditional love and support.

I love you all dearly.

Acknowledgment

I would like to express my sincerest appreciation to my supervisor Professor Aditya Mahajan for the constant support and patient guidance he has provided me throughout the course of my thesis. He convincingly guided and encouraged me to be professional and do the right thing even when the road got tough. Without his persistent help, the goal of this project would not have been realized. I am forever thankful to him for standing by my side throughout the challenges that I faced.

I would like to express gratitude to my other supervisor Professor Doina Precup for her counselling and support even when she was extremely busy. It was after taking her reinforcement learning classes that motivated me to explore the field of reinforcement learning. I am extremely lucky to have such insightful and supportive supervisors by my side.

I would like to thank the MITACS Accelerate program for funding my thesis (grant IT16364) and Ericsson Systems for believing in the project. I would especially like to thank Zhongwen Zhu of Ericsson for coming up with the problem statement and being a constant mentor to me in the past year. I would also like to thank Emmanuel Thepie Fapi, Hatem Abou-zaid, Hakimeh Purmehdi, and Prasad Garigipati from Ericsson for all the intense and productive weekly sync-ups and their advice whenever I was stuck with a problem. Special thanks to Pierre Thibault from Ericsson Systems for setting up the virtual machines to run the second set of experiments.

I would also like to thank Montreal Institute of Learning Algorithms (MILA) for providing me an environment of delightful and remarkable people with whom I could discuss potential solution to problems and attend information sessions that were really helpful for my thesis. They also provided me with a good working space along with tools, tutorials and compute required for my thesis.

I would like to thank Jhelum Chakraborty (post-doc) for helping me narrow down a research topic and providing me with research materials pertinent to my thesis, Professor Muthcumaru Maheswaran for piquing my interest in the application of machine learning in networks and communications, and Amit Sinha my fellow lab-mate for listening to me and providing insights on my solution approaches.

Last but not the least I would like to thank my parents and friends in Montreal who were my constant emotional support-system throughout my masters. Without their unconditional love, it would not have been possible to go through these tough times, especially during the pandemic.

Contents

List of Tables

List of Figures

List of Common Acronyms

A2C	Advantage Actor-Critic
BSs	Base Stations
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
IoT	Internet-of-Things
MEC	Mobile Edge Computing
PPO	Proximal Policy Optimization
QoE	Quality of Experience
QoS	Quality of Service
RL	Reinforcement Learning
SALMUT	Structure-Aware Learning for Multiple Thresholds
SGD	Stochastic Gradient Descent
TRPO	Trust Region Policy Optimization
VoD	Video-on-Demand

List of Common Symbols

Symbol	Description
X_t	Queue length at time t
L_t	CPU load of the system at time t
A_t	Offloading action taken by agent at time t
$P(r)$	Probability Mass function of the CPU resources required
μ	Processing time of a single core in the edge node
λ	Request arrival rate of user
h	Holding cost per unit time
$c(\ell)$	Running cost per unit time
$p(\ell)$	Penalty for offloading the packet
$V^{\pi}(x, \ell)$	Performance of the policy π
$p(x', \ell' \mid x, \ell, a)$	Transition probability function
$\bar{\rho}(x, \ell, a)$	Cost function in the discrete MDP
$Q(x, \ell, a)$	Q-value for state (x, ℓ) and action a
π_{τ}	Optimal Threshold Policy for SALMUT
$f(\tau(x), \ell)$	Probability of accepting new request

Chapter 1

Introduction

1.1 Motivation

In the last decade, we have seen a shift in the computing paradigm, from co-located datacenters and compute servers to cloud computing. Due to the aggregation of resources, cloud computing can deliver elastic computing power and storage to customers without the overhead of setting up expensive datacenters and networking infrastructures. It has especially attracted small-sized and medium-sized businesses who can leverage the cloud infrastructure with minimal setup costs. In recent years, the focus on the quality of the experience of end users has become a very important aspect for network providers, especially with the proliferation of Video-On-Demand (VoD) services, Internet-of-Things (IoT), real-time online gaming platforms, and Virtual Reality (VR) applications. The cloud paradigm is not the ideal candidate for such latency-sensitive applications owing to the delay between the end user and cloud server.

This has led to a new trend in computing called Mobile Edge Computing (MEC) [1, 2], where the compute capabilities are increasingly moving closer to the network edges. It represents an essential building block in the 5G vision of creating a large distributed,

pervasive, heterogeneous, and multi-domain environments. Harvesting the vast amount of the idle computation power and storage space distributed at the network edges can yield sufficient capacities for performing computation-intensive and latency-critical tasks by the end-users. However, it is not feasible to set up huge resourceful edge clusters along all network edges that mimic the capabilities of the cloud, due to the sheer volume of resources that would be required, which would remain underutilized most of the times. Due to the limited resources at the edge nodes and fluctuations in the user requests, an edge cluster may not be capable of meeting the resource and service requirements of all the users it is serving.

Computation offloading methods have gained a lot of popularity as they provide a simple solution to overcome the problems of edge and mobile computing. Data and computation offloading can potentially reduce the processing delay, improve energy efficiency, and even enhance security for computation-intensive applications. The critical problem in computation offloading is to determine the computational workload, and to choose the MEC server from all available servers. Various aspects of MEC from the point of view of the mobile user have been investigated in the literature. For example, the questions of when to offload to a mobile server, to which mobile server to offload, and how to offload have been studied extensively (see [3–7] and references therein).

However, the design questions at the server level have not been investigated as extensively. When an edge server receives a large number of requests in a short period of time (for example due to a sporting event), the edge server can get overloaded, which can lead to service degradation or even node failure. When such service degradation occurs, edge servers are configured to offload requests to other nodes in the cluster in order to avoid crashing. The crash of an edge node leads to reduction of the cluster, which is

catastrophic for the platform operator as well as the end users, who are using the services or the applications. However, performing this migration takes extra time and reduces the resources available for other services provided by the cluster. Therefore, it is paramount to design *proactive* mechanisms that prevent the node from getting overloaded, by using dynamic offloading policies that can adapt to service request dynamics.

In this thesis, we study the problem of node overload protection for a single edge node. We first model the problem to incorporate practical considerations of server holding, processing and offloading costs. Then we develop an offloading policy that rejects and offloads new requests to balance the overall running cost of the system. In the simplest case, when the request arrival process is time-homogeneous, we model the system as a continuous-time Markov decision process (MDP) and use the *uniformization technique* [8, 9] to convert the continuous-time MDP to a discrete-time MDP, which can then be solved using standard dynamic programming algorithms [10].

However, solving a dynamic program requires knowledge of the system parameters, i.e. the MDP model, which is not typically known and may also, in practice, vary with time. In such time-varying environments, the offloading policy must adapt to the environment. Reinforcement learning (RL) [11] is a natural choice to design such adaptive policies and has already been successfully applied in various optimization problems arising in MEC [12–14].

Although RL has achieved considerable success in various application domains, including communication networks, this success is generally achieved by using deep neural networks to model the policy and the value function. Such deep RL algorithms require considerable computational power and time to train, are notoriously brittle to the choice of algorithm hyper-parameters, may not transfer well from simulation to the real-world,

and produce policies which are difficult to interpret. These features make them impractical for deployment on the edge nodes, where continuous adaptation to changing network conditions is necessary.

For the aforementioned reasons, rather than using general purpose deep RL algorithms, we design a node overload protection scheme that uses a recently proposed low-complexity RL algorithm called SALMUT (Structure-Aware Learning for Multiple Thresholds) [15]. SALMUT exploits the structure of the optimal policy, requires considerably fewer computational resources to train and provides policies which are easy to interpret. We compare the performance of deep RL algorithms with SALMUT in a variety of scenarios motivated by real world deployments. Our experiments show that SALMUT performs as well as state-of-the-art deep RL algorithms such as Proximal Policy Optimization (PPO) [16] and Advantage Actor Critic (A2C) [17] but requires an order of magnitude less time to train and provides policies which are easy to interpret.

1.2 Thesis Organization and Contributions

This thesis is organized as follows:

- *Chapter 2 - Background and Related Work*

  This chapter provides background knowledge of Reinforcement Learning necessary for understanding the work presented in later chapters. It also includes a detailed literature review of reinforcement learning applied to networking and communication applications.

- *Chapter 3 - Problem Formulation*

  This chapter provides the system model for the optimal admission control policy in

the node overload protection problem. It also contains the dynamic programming solution for the admission control policy and the structure-aware formulation and solution to the problem. This is joint work with Aditya Mahajan. The problem was proposed by Zhongwen Zhu.

- *Chapter 4 - Experiments*

  This chapter presents the numerical evaluation of the admission control policy in the node overload protection problem. It includes analysis of the choice of parameters for the model, followed by empirical evaluation in two different settings - a simulated environment and a real testbed. It also covers the analysis of training time and policy interpretability for SALMUT and behavioral analysis of the policies learned by different algorithms. Ideas on which experiments to run have originated from discussions with Aditya Mahajan.

- *Chapter 5 - Conclusions and Future Work*

  This chapter presents the conclusions of the experiments conducted in this thesis and suggests potential future avenues of investigation.

1.3 Scope of Thesis and Contributions

Our main contributions are the following:

- We design a node overload-protection scheme that uses a recently proposed low-complexity RL algorithm called SALMUT (Structure-Aware Learning for Multiple Thresholds) [15]. SALMUT exploits the structure of the optimal policy, requires considerably fewer computational resources to train and produces policies which are easy to interpret.

- We compare the performance of deep RL algorithms with SALMUT in a variety of scenarios, in a simulated testbed motivated by real world deployments. Our experiments show that SALMUT performs close to state-of-the-art deep RL algorithms such as PPO [16] and A2C [17] but requires an order of magnitude less time to train, and provides policies which are easy to interpret.

- We also developed a real-testbed where we run actual workloads and compare the performance of the SALMUT algorithm with a baseline algorithm.

- We perform an in-depth analysis of the effects of different parameters of the cost function on the structure of the optimal policy and perform a sensitivity analysis of the chosen cost structure to small changes in the request distribution.

Chapter 2

Background and Related Work

2.1 Reinforcement Learning

In this chapter, we cover ideas in Reinforcement Learning that are necessary for understanding the approach we introduce in chapter 3. For a broader understanding of the area, we refer the reader to [11]. Learning from interaction is a foundational idea that underlies theories of learning and intelligence. Reinforcement learning (RL) is a computational approach towards learning from interactions.

2.1.1 Markov Decision Process

Markov decision processes (MDPs) [10] are a theoretical formalism for modelling sequential decision making, in which an agent interacts with an environment where the states are fully observable. The interaction between the agent and the environment can be seen in Fig. 2.1. The agent is at state S_t at a given time t and interacts with the environment by taking an action A_t. It receives the next state S_{t+1} and the reward R_{t+1} from the environment.

A MDP is a tuple of five elements $M = (S, A, p, r, \gamma)$, as defined in [18], where:

- S is a finite set of states. The state at time t is denoted as $S_t \in S$.

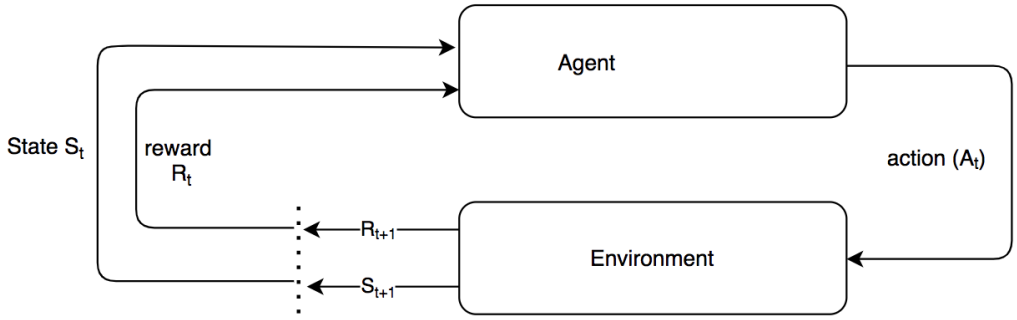


Figure 2.1: Agent-Environment interaction in a MDP (adapted from [11]).

- A is a finite set of possible actions. The action at time t is denoted as $A_t \in A$.
- $p : S \times A \times S \to [0, 1]$ is the state transition probability, where $p(s'|s, a)$ gives the distribution of the possible next state $s' \in S$ upon taking action $a \in A$ in state $s \in S$.
- $r : S \times A \times S \to \mathbb{R}$ is the expected reward obtained as a result of choosing action a at state s. The reward at time $t+1$ is denoted as $R_{t+1} \in \mathbb{R}$. It is the numerical reward received from the environment at state S_t after taking action A_t and transitioning to next state S_{t+1}.
- $\gamma \in [0, 1)$ is temporal discount factor that is used to de-prioritize rewards obtained far away in future.

A MDP satisfies the Markovian property:

$$\Pr(S_{t+1}|S_t, A_t) = \Pr(S_{t+1}|S_t, A_t, ..., S_1, A_1, S_0),$$

which implies that the current state contains all the information needed by the agent to take a decision.

In a finite MDP, the sets of states and actions (S, A) have a finite number of elements.

Given a state s and action a, the probability of next state s' is given by:

$$p(s'|s,a) = \Pr(S_{t+1} = s'|S_t = s, A_t = a). \tag{2.1}$$

An MDP could either be episodic or continuing. In an episodic MDP, the interaction between the agent and the environment breaks naturally into subsequences, which we call episodes. Each episode ends in a special state called the terminal state, followed by a reset to a starting state, which can be fixed or sampled from a given distribution of starting states. In an episodic MDP, there is a natural notion of final time step defined by T, which is the time at which the agent enters the terminal state. Tasks which have episodes are called episodic tasks. The final time step T is a random variable. Continuous tasks are tasks that have no ends i.e. they don't have any terminal state. These types of tasks will never end. We will define all the terms for episodic tasks henceforth, unless explicitly mentioned.

2.1.2 Policies and Value Functions

Policy is a mapping from states to probabilities of selecting each possible action, $\pi(a|s)$. The goal of the agent in RL typically is to find a policy $\pi(a|s)$ that maps from states s to actions a in a way that maximizes the sum of expected future discounted rewards. This sum is called the expected discounted return (or simply return). The return following time step t is denoted by G_t and is given by:

$$G_t = \sum_{i=t+1}^{T} \gamma^{i-t-1} R_i.$$

Reinforcement learning algorithms specify how the agent should change its policy to optimize the expected return. If the policy is deterministic, the agent will have non-zero probabilities for all actions except one, and the policy can be represented as $\pi : S \to A$. If Π is the set of all possible policies in an environment, the goal of the RL agent is to find

$\pi \in \Pi$ that gives the highest expected discounted return possible from every state. This policy is also called the optimal policy and is denoted as π^*.

The value function under a policy π, denoted by $V^\pi : S \to \mathbb{R}$, is the expected return given any state. When starting in state s at time t and following the policy π thereafter, the value function is formally defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{i=t+1}^{T} \gamma^{i-t-1} R_i | S_t = s \right]. \tag{2.2}$$

The action-value function or Q-function under a policy π, denoted by $Q^\pi : S \times A \to \mathbb{R}$, is the expected return conditioned on both state and action. When starting in state s at time t, taking action a, and following the policy π thereafter, it is formally defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{i=t+1}^{T} \gamma^{i-t-1} R_i | S_t = s, A_t = a \right]. \tag{2.3}$$

One can also express the value function as a relationship between the value of any state s and the values of its successor states s', known as the Bellman equation [19]:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V^\pi(s')]. \tag{2.4}$$

The system of Bellman equations can also be written in a matrix form as:

$$\mathbf{V}^\pi = \mathbf{r}^\pi + \gamma \mathbf{P}^\pi \mathbf{V}^\pi, \tag{2.5}$$

where in $\mathbf{V}^\pi \in \mathbb{R}^{|S|}$ represents the value of a state, each element in $\mathbf{r}^\pi \in \mathbb{R}^{|S|}$ represents the reward associated with each state and $\mathbf{P}^\pi \in \mathbb{R}^{|S| \times |S|}$ represents the state transition probability for a fixed policy π. Formally, $\mathbf{V}^\pi = [V^\pi(1).....V^\pi(|S|)]^\top$, is a vector representing the value of each state. Similarly, $\mathbf{r}^\pi = [r^\pi(1).....r^\pi(|S|)]^\top$ is a vector representing the expected numerical reward at each state, which can be computed by marginalizing over the next states and actions, and $\mathbf{P}^\pi = [P^\pi(1).....P^\pi(|S|)]^\top$ is the state-transition matrix, with each row $P^\pi(s)$ representing the probability distribution over the next states given state s,

computed by marginalizing over the actions. The number of states in the environment is denoted as $|S|$.

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$, only if $V^{\pi}(s) \geq V^{\pi'}(s), \forall s \in S$. There is always at least one policy that is better than or equal to all other policies, known as an optimal policy. We denote any optimal policy by π^*. All optimal policies share the same state-value function, called the optimal state-value function, denoted V^{π^*} and defined as:

$$V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s), \quad \forall s \in S. \tag{2.6}$$

Similarly, the optimal action-value function, denoted Q^{π^*}, is defined as:

$$Q^{\pi^*}(s,a) = \max_{\pi} Q^{\pi}(s,a), \quad \forall s \in S, \forall a \in A. \tag{2.7}$$

The Bellman optimality equation [19] for any state s is given by

$$V^{\pi^*}(s) = \max_{a} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V^{\pi^*}(s')]. \tag{2.8}$$

Similarly, the Bellman optimality equation for the Q-function of state s and action a is given by:

$$Q^{\pi^*}(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} Q^{\pi^*}(s',a')]. \tag{2.9}$$

For finite MDPs, the system of Bellman optimality equations (2.8) and (2.9) has a unique solution. In principle, one can obtain $V^{\pi^*}(s)$ or $Q^{\pi^*}(s,a)$ by solving the equations above in an iterative manner, if the dynamics of the environment $p(s'|s,a)$ and the rewards $r(s,a,s')$ are known. Once $V^{\pi^*}(s)$ or $Q^{\pi^*}(s,a)$ are obtained, the optimal policy can be derived. For each state s, there will be one or more actions for which the maximum is obtained in the Bellman optimality equation. Any policy π that assigns non-zero probability only to these actions is an optimal policy π^*.

Dynamic programming refers to the collection of algorithms that can be used to find optimal policies when the model of the environment is available. Iterative methods to solve the Bellman optimality equations include value iteration [20], policy iteration [21] or generalized policy iteration [20].

2.1.3 Learning Algorithms

In many practical applications, $p(s'|s,a)$ is not available, so one cannot obtain the optimal policy using (2.8) or (2.9). In order to address this issue, various methods such as Monte Carlo [22], Temporal Difference (TD) learning [23], n-step TD [22] have been developed to estimate the value functions or Q-functions. These methods do not need the an model of the environment. Instead, they learn the value functions or Q-functions by interacting with the environment.

Monte Carlo (MC) methods are a way of estimating the value function for a given policy (prediction problem) by averaging sampled returns. One way to estimate the value of a state from trajectories in the environment would be to simply average the returns observed after visits to that state.

The value function update equation for MC methods is:

$$V^{\pi}(S_t) \leftarrow V^{\pi}(S_t) + \alpha(G_t - V^{\pi}(S_t)), \tag{2.10}$$

where $\alpha \in (0,1]$ is the learning rate. If α is decayed appropriately, this provides an unbiased estimator of the value function, because the updates are done based on the actual returns that result from the interaction with the environment. However, the agent needs to wait until the end of an episode, when return can be calculated, to update the value functions, which makes this type of method very slow in practice. Monte Carlo estimates can also suffer from high variance.

Temporal difference (TD) methods are an alternative way to estimate the value function. They are generally preferred over MC because the updates are done online (one does not need to wait until the end of the episode to update values) and they can help to reduce the variance of the estimates, compared to MC. These methods use the current estimate of V^π instead of the true value, hence they are also called bootstrap methods. The value function update equation for one-step TD method, or TD(0), is:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)), \tag{2.11}$$

where we assume that actions are generated according to the fixed policy π which is being evaluated. The quantity $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is denoted by δ_t, also known as TD error. It measures the difference between the estimated value of S_t and the estimate $R_{t+1} + \gamma V(S_{t+1})$, which is obtained one time step later

There are two very popular methods to do control (i.e., to approximate optimal policies) using TD-style updates, known as SARSA [24] and Q-learning [25]. In SARSA, bootstrapping is performed from an action-value that is computed according to the current policy (also called on-policy). The agent takes actions in a way which favors the actions with the highest current estimated value. For example, the agent may follow an ϵ-greedy policy, in which the action which currently has the highest value is taken with probability $1 - \epsilon$, and the agent chooses uniformly among the other actions, with total probability ϵ, where $\epsilon \in (0, 1)$ is chosen to be a small number. The SARSA update function is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \tag{2.12}$$

In Q-learning, the bootstrapping uses the highest action-value from the next state, regardless of what the agent actually ends up doing. This means that the agent can act according to any policy, as the update is off-policy. The Q-learning update is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)). \tag{2.13}$$

The n-step TD methods generalize both MC and TD methods, so one can shift from one to the other smoothly as needed to meet the demands of a particular task. n-step methods span a spectrum, with MC methods at one end and one-step TD methods at the other. Instead of the one-step return as in TD or full return as in MC, n-step TD methods utilize n-step returns to compute bootstrap targets:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{2.14}$$

where we use indices to denote the time step at which a value function estimate is queried. The value function update equation for n-step TD is:

$$V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha(G_{t:t+n} - V_{t+n-1}(S_t)). \tag{2.15}$$

If the state space and action space are finite and small, the value functions or Q-functions can be represented as a table with one entry for each state or state–action pair. This represeentation is commonly known as tabular. With larger state and action spaces, tabular representations lead to excessive memory requirements and sample inefficiency. Therefore, we need function approximations that takes examples of transitions and attempts to generalize from them to construct an approximation of the entire function.

There are two main approaches to solve this problem. In the first approach (value-based methods), a value function or Q-function is learned to approximate the expected discounted return. In the second approach (policy-based methods), a function approximator for the policy is directly learned, which then determines the probability of choosing an action for a given state. We review these two types of methods in the next two subsections.

2.1.4 Value-based Approximation Methods

In value-based approximation methods, a parametric function $\tilde{V}^\pi(s;\theta)$ or $\tilde{Q}^\pi(s,a;\theta)$ is learned to estimate $V^\pi(s)$ or $Q^\pi(s,a)$, where $\theta \in \mathbb{R}^d$ ($d \ll |S|$) are the parameters of the

functional form, which are updated as follows:

$$\theta_{t+1} = \theta_t + \alpha[U_t - \tilde{V}^\pi(S_t;\theta_t)]\nabla\tilde{V}^\pi(S_t;\theta_t), \tag{2.16}$$

where U_t is the target of the update. If U_t is an unbiased estimate of the value function, i.e. $\mathbb{E}[U_t|S_t = s] = V^\pi(S_t)$ (e.g. the returns G_t in Monte Carlo methods), then θ_t is guaranteed to converge to a local optimum under stochastic approximation conditions [26], which include decreasing α according to a particular schedule over time and some other technical conditions [27]. However, conveergence cannot always be guaranteed if a bootstraping estimate of U_t is used, as in TD(0) or n-step methods, as these methods are biased and do not produce true gradients. They take into account the effect of changing the parameter θ_t on the estimate, but ignore its effect on the target. They only include a part of the gradient, and therefore they are called semi-gradient methods. Semi-gradient TD methods do converge [11] for linear function approximations to a parameter value called the TD fixed point under the on-policy distribution. However, linear approximators are not powerful enough to capture all complexities of real environments. To address this issue, there has been a tremendous amount of research on non-linear approximators, especially neural networks. One of the most popular techniques which has received tremendous attention in recent years is Deep Q-Network (DQN) [28]

Deep Q-Network (DQN)

DQN utilizes a replay memory buffer and a target network to stabilize the training. The replay buffer holds previous observation tuples $(S_t, A_t, R_t, S_{t+1}, d_t)$ where d_t records if the episode ended with this observation. Then, the approximator is trained by taking a random mini-batch from the replay memory and applying the Q-learning update rule. DQN uses a neural network, such as Convolutional Neural Network (CNN), to input S_t. The network

has $|A|$ outputs to approximate the Q-function for each possible action. The weights θ are trained by taking a mini-batch of size m and minimizing the loss function:

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y_i - Q(S_i, A_i; \theta))^2, \tag{2.17}$$

where

$$y_i = \begin{cases} R_i, & d_t = True, \\ R_i + \gamma \max_{a'} Q(S_i', a'; \theta^-), & d_t = False, \end{cases} \tag{2.18}$$

where θ^- is the weight vector of the target network, which is updated to θ every M iterations. There are other other approaches inspired by DQN, such as Dueling Deep Q-Network (DDQN) [29], Deep Recurrent Q-Networks (DQRN) [30] etc.

2.1.5 Policy-based Approximation Methods

The goal of policy-based methods is to directly learn the parameters of the policy π_θ, parameterized by θ, using a gradient-based algorithm. Let $J(\theta)$ measure the value function for the policy π_θ starting from state s_0:

$$J(\theta) = V^{\pi_\theta}(s_0).$$

Then the policy gradient theorem [11] provides an analytical expression for gradient of $J(\theta)$ as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi(a|s;\theta) Q^{\pi_\theta}(s,a)]. \tag{2.19}$$

The parameters θ of the policy are updated as follows:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} J(\theta_t), \tag{2.20}$$

where $\nabla_\theta J(\theta) \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ and α is the learning rate.

Two commonly used policy-gradient methods are REINFORCE and actor-critic. REINFORCE [31] applies the same idea as the Monte Carlo method and uses the actual return to estimate the objective: in (2.19)

$$G_0 := \sum_{t=0}^{T} \gamma^t R_{t+1}. \tag{2.21}$$

The Actor-Critic (AC) [11] model modifies the REINFORCE algorithm by adding another approximator, called the critic. The actor learns the policy parameters, similar to the REINFORCE method, whereas the critic learns the Q-function, which provides the gradient to the policy. The critic receives state S_t and returns the approximation of $Q_\phi(s, a)$, where ϕ are the parameters of the critic network. The critic is trained by calculating the TD-error and updating ϕ as:

$$\delta_t = R_{t+1} + \gamma Q_\phi(S_{t+1}, A_{t+1}) - Q_\phi(S_t, A_t), \tag{2.22}$$

$$\phi_{t+1} = \phi_t + \alpha_\phi \delta_t \nabla_\phi Q_\phi(S_t, A_t), \tag{2.23}$$

where α_ϕ is the critic's learning rate. The most widely used policy-gradient algorithms include Advantage Actor-Critic (A2C) [32], Proximal Policy Optimization (PPO) [33], Trust Region Policy Optimization (TRPO) [34]. We will now review the first two of these algorithms.

Advantage Actor-Critic (A2C)

A2C maintains a policy $\pi(a|s;\theta)$ and an estimate of the value function $V(s;\phi)$. The algorithm uses the n-step return to update both the policy and the value function. The update performed after every $T_{\max}$ steps is:

$$\theta_{t+1} = \theta_t + \nabla_\theta \log \pi(A_t|S_t;\theta) \times \mathcal{A}(S_t, A_t;\theta,\phi), \tag{2.24}$$

where $\mathcal{A}(S_t, A_t; \theta, \phi)$ is the estimate of the advantage function defined by

$$\mathcal{A}(S_t, A_t; \theta, \phi) = \sum_{i=0}^{j-1} \gamma^i R_{t+i+1} + \gamma^j V(S_{t+j}; \phi) - V(S_t; \phi), \tag{2.25}$$

and j is upper bounded by $T_{\max}$. Even though the parameters θ of the policy and ϕ of the value function are shown as being separate for generality, in practice the policy and value function usually share a common layer of features. Typically, a neural network is used that has one softmax output for the policy $\pi(A_t|S_t; \theta)$ and one linear output for the value function $V(S_t; \phi)$, with all non-output layers shared.

Proximal Policy Optimization (PPO)

Implementing policy gradient methods is challenging because they are sensitive to the choice of stepsize and often have very poor sample efficiency. If the step size is too small, then the progress is very slow whereas if it is too large, then the signal is overwhelmed by the noise. Several methods were developed to eliminate these flaws. The family of trust-region algorithms like TRPO [34] constrain or optimize the size of a policy update. But TRPO is not easily compatible with sharing parameters between a policy and value function, or with auxiliary losses.

PPO is a trust-region policy gradient algorithm that is easy to tune and implement, reduces sample complexity and computes an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. The objective function is as follows:

$$J(\theta) = \tilde{E}_t[\min(\tilde{\rho}_t(\theta)\tilde{\mathcal{A}}_t, clip(\tilde{\rho}_t(\theta), 1-\epsilon, 1+\epsilon)\tilde{\mathcal{A}}_t], \tag{2.26}$$

where θ is the policy parameter, $\tilde{E}_t$ denotes the empirical expectation over timesteps, $\tilde{\rho}_t(\theta)$ is the ratio of probability under the new and old policies respectively, $\tilde{\mathcal{A}}_t$ is the estimated

advantage at time t, as in A2C, and ϵ is a hyperparameter. The $clip(.)$ function is defined as follows:

$$c(x,a,b) = \begin{cases} 1-\epsilon, & x < 1-\epsilon, \\ 1+\epsilon, & x > 1+\epsilon, \\ x, & \text{otherwise}, \end{cases}$$

indicating that if $\tilde{\rho}_t(\theta)$ causes the objective function to increase to a certain extent, its effectiveness will decrease (be clipped). This objective implements a trust region update which is compatible with Stochastic Gradient Descent (SGD).

2.2 RL in Networks and Communications

Reinforcement learning has been recently used as a tool to effectively address various problems and challenges in the areas of networks and communications. As modern networks such as Internet of Things (IoT), Heterogeneous Networks (HetNets), and Unmanned Aerial Vehicle (UAV) networks become more autonomous and decentralized, network entities need to make local and autonomous decisions, such as spectrum access, data rate selection, transmission power control, and base station association, in order to achieve targets such as throughput maximization, energy consumption minimization, latency minimization, etc. In uncertain and stochastic environments, most of the decision-making problems can be modeled by Markov Decision Processes. This section contains a concise summary of the applications of reinforcement learning to different networks and communications problems, as presented in [35].

2.2.1 Data and Computation Offloading

IoT devices need to support advanced applications such as interactive online gaming and face recognition with good user experience, despite the limited computation, memory and power supplies at the edge of the network. In order to address the limited capabilities at the edge of the network, IoT devices can offload computational tasks to nearby Mobile Edge Computing (MEC) servers, Access Points (APs), and even neighboring Mobile Users (MUs). Data and computation offloading can potentially reduce the processing delay, save battery, and even enhance security for computation-intensive IoT applications. However, a critical problem in computation offloading is to determine the amount of computational workload to offload, and to choose the MEC server (from all available servers) to which to offload. If the chosen MEC server experiences heavy workloads and degraded channel conditions, it may take even longer for the IoT devices to offload data and receive the results from the MEC server. Hence, the design of an offloading policy has to take into account time-varying channel conditions, user mobility, energy supply, computation workload and the computational capabilities of different MEC servers.

The authors in [36] use Deep Q-Network (DQN) to minimize the cost of delay and power consumption for all mobile users, by jointly optimizing the offloading decision and computational resource allocation. The system is modelled as a MDP where the states comprise of the sum of the costs in the entire system and the available computational capacity of the MEC server. The action for a base station is to determine the resource allocation and offloading decision for mobile users. To overcome the large action space, the authors introduce a pre-classification step which removes all the non-feasible actions for the mobile users. In [37], the work done by [36] is enhanced by using Dueling Deep Q-Network (DDQN) [29] in an ultra-dense network. They optimize the association between

mobile users and the BSs. They enhance the system states by adding the energy and task queues of the BSs in addition to the channel conditions between the mobile user and the BSs. The cost function is a weighted sum of the execution delay, the handover delay and the computational task dropping costs. The authors in [38] aim to design an optimal offloading policy for IoT devices with energy harvesting capabilities. The system consists of multiple MEC servers, which vary in their computation and communication capabilities. The IoT devices can execute computational tasks locally or offload the tasks to the MEC servers. The IoT device's offloading decision is formulated as a MDP where the states include the battery status, the channel capacity, and the predicted amount of harvested energy in the future. The reward incorporates the overall delay, energy consumption, the task drop loss, and the data sharing gains in each time slot. The authors propose a fast DQN offloading scheme that uses hotbooting to initialize a Convolution Neural Network (CNN), which accelerates the learning speed.

In [5] and [39], the authors study QoS-aware computation offloading in an ad-hoc mobile network. The mobile user can offload computational tasks to nearby mobile users, constituting a mobile cloudlet, by making a certain amount of payment. The mobile user has a queue with limited buffer size to store the arriving tasks. The mobile user selects nearby cloudlets within device-to-device (D2D) communication range for offloading the task. The offloading decision is modeled as a MDP with the states including the number of remaining tasks, the quality of the links between mobile users and the cloudlet, and the availability of the cloudlet's resources. The objective is to maximize a combined utility function, incorporating the mobile user's energy consumption, processing delay, and payment for task offloading. The authors show that the utility function is an non-decreasing function of the total number of tasks that have been processed either locally or remotely by

the cloudlets. They formulate the optimization problem using a MDP and solve it using DQN.

Data and computation offloading is also used in fog computing, where the mobile application requesting data and the computational resources can be hosted in a container. The authors in [40] model the container migration as a multi-dimensional MDP with the system states comprising of the delay, the power consumption and the migration cost. The action includes the selection policy that selects the containers to be migrated from each source node, and the allocation policy that determines the destination node of each container. The action space is optimized for efficient exploration by dividing fog nodes into under-utilization, normal-utilization, and over-utilization groups. The training process is optimized by using DDQN [29] and Prioritized Experience Replay (PER) [41] which assigns different priorities to the transitions in the replay buffer.

2.2.2 Network Access and Rate Control

As IoT networks become more decentralized, sensors and mobile users need to make independent decisions, such as channel and base station selections, to maximize their throughput. Reinforcement Learning has been studied to optimize for the several problems of this type, described below.

Dynamic Spectrum Access

Dynamic spectrum access allows users to dynamically select channels to maximize their throughput. Users typically do not have full channel information. Deep Reinforcement Learning (DRL) can thus be used as a powerful dynamic spectrum access method. The authors in [42] suggest a dynamic channel access scheme of a sensor by formulating the problem as a partially observable Markov Decision Process (POMDP) and leveraging DQN

to come up with an effective allocation scheme. The sensor selects one of the M channels in each time slot to transmit its packet. If the selected channel is in small interference, the sensor receives a positive reward, otherwise the reward is negative. The goal is to find a policy that maximises the expected discounted return of the sensor over time slots. An adaptive scheme is proposed, which assesses the current policy's accumulated reward. DQN is re-trained to find a new good policy when the reward is reduced by a pre-defined threshold.

Vehicle to Vehicle Communication

One of the difficulties of Vehicle to Vehicle (V2V) communication is that, in order to optimise its capacity under a latency constraint, each V2V transmitter must choose a channel and a transmission power level. DQN is used in [43], in which agents (V2V transmitters) share a range of channels. Each V2V transmitter's actions include selecting channels and transmitting power levels. The reward is a function of the capacity and latency of the V2V transmitter. The state observed by the V2V transmitter consists of (a) the instantaneous channel state information (CSI) of the corresponding V2V connection, (b) the interference with the V2V connection in the previous time slot, (c) the channels chosen by the neighbours of the V2V transmitter in the previous time slot, and (d) the time remaining to reach the latency constraint.

Joint User Association and Spectrum Access

User association is introduced to decide which user to delegate to which Base Station (BS). Usually, the joint user association and spectrum access problems are combinatorial and non-convex, requiring almost complete and precise network information to achieve the optimal strategy. The authors consider a heterogeneous network (HetNet) in [44], which

consists of multiple users and BSs - macro base stations and femto base stations. Each user's task is to select one BS and one channel in order to optimise its data rate, while ensuring that the user's Signal-to-Interference-plus-Noise Ratio (SINR) is higher than the minimum requirement for Quality of Service (QoS). To address the problem, the state consists of a vector of the QoS states of all users, and a DQN strategy is explored. The user takes an action at each time slot (selecting BS and channel) and gets a positive reward if the QoS is met and a negative reward otherwise. To learn the optimal strategy, they use DDQN [29].

Adaptive Rate Control

In complex and unpredictable environments such as Dynamic Adaptive Streaming over HTTP (DASH) [45], Adaptive Rate Control refers to data rate control, which enables users to individually select video segments of different bitrates to download. The objective of the client is to maximize the quality of experience (QoE) of the client, such as maximizing the average bitrate and minimizing rebuffering. The problem can be formulated as a MDP [46], where the agent is the client and the action comprises a bitrate codec representation to download. The client state includes (a) the video quality of the last segment downloaded, (b) the current buffer state, (c) the rebuffering time, and (d) the channel capabilities encountered in previous time slots during segment downloads. They use Asynchronous Advantage Actor-Critic (A3C) [32], where bitrates are chosen for the client by the actor network, and the critic network helps train the actor network. The reward is a composite function consisting of (a) the video's visual quality, (b) the consistency of the video quality, (c) the rebuffering event, and (d) the buffer state. The authors in [47] demonstrate that in a potential space communication system that is supposed to function in unpredictable conditions, e.g., orbital dynamics, atmospheric and space atmosphere, and complex networks, deep

RL can be used for rate control to achieve multiple goals. In order to achieve multiple objectives such as low Bit Error Rate (BER), throughput increase, power and spectral performance, the transmitter in the device must be configured with parameters such as symbol rate and encoding rate. The proposed approach uses a set of multiple neural networks in parallel to achieve the multi-objective goal. The input is the current state and the channel conditions, and the output is the predicted action. The neural networks are trained by using the Levenberg-Marquardt backpropagation algorithm [48].

2.2.3 Caching

Studies on wireless caching have shown that access delays, energy consumption, and the total amount of traffic can be reduced significantly by caching content in wireless devices at the edge of the network. By deploying both computational resources and caching capabilities close to end users, one can significantly improve the energy efficiency and QoS for applications that require intensive computations and low latency. Statistically, a small amount of popular content is usually requested by many users during a short time span, which accounts for most of the traffic load. Proactively caching such popular content can avoid the heavy traffic burden of the backhaul links.

Caching in networks has been broadly studied in two categories: with and without perfect content popularity information. The majority of related research assumes that perfect popularity information is known in advance. Even then, the optimal cache placement policy is a NP hard problem. Due to the content dynamics and users' mobility, content popularity may change over time. Therefore, it is necessary to learn the content popularity and users' preference dynamically. There has also been some research which assumes imperfect information about content popularity. In this case, machine learning methods are

exploited to learn the content popularity by observing users' historical content demands and user-content correlations.

Most of the current literature on distributed edge caching assumes that the content popularity distribution is known or that the learned content distribution is stationary. Under these assumption, distributed algorithms to determine the content placement have been proposed in [49–58] using techniques such as integer linear programming, convex relaxations, particle swarm optimization, belief propagation, and non-cooperative games. Most of these works model the content popularity distribution using an Independent Reference Model (IRM). This usually does not incorporate temporal locality and short-term popularity, which are very important for an optimal caching strategy [59].

In recent years, there is some work on using reinforcement learning to design distributed caching algorithms when the content popularity is not known. The authors in [60] propose a deep RL caching algorithm for a single BS with a fixed cache size. For each request, the BS makes a decision on whether or not to store the currently requested content in the cache. If the new content is kept, the BS determines which local content will be replaced. They employ the Wolpertinger architecture [61] to reduce the size of the action space and avoid missing an optimal policy. The Wolpertinger architecture consists of three main parts: an actor network, K-Nearest Neighbors (K-NN), and a critic network. The actor network is meant to avoid a too-large action space, the critic network corrects the decisions made by the actor network, the Deep Deterministic Policy Gradients (DDPG) [62] method is applied to update both the critic and the actor networks, whereas the K-NN can help to explore the set of actions better, in order to avoid poor decisions.

The authors of [63] address caching in 5G cellular networks, where space-time popularity of requested files is modeled via local and global Markov chains. The proposed

framework entails estimation of the popularity profiles both at the local as well as at the global scale - each SB estimates its local vector of popularity profiles based on limited observations, and transmits it to the network operator, where an estimate of the global profile is obtained by aggregating the local ones. They model three types of cost: (a) cost of refreshing cache contents, (b) cost incurred during the operational phase, which penalizes requests for files already cached much less than requests for files not stored, (c) cost that captures the mismatch between the caching action and the global popularity profile. The entire cost is a linear combination of these terms, adjusted according to the characteristics of the network.

2.2.4 Network Security

As networks become more decentralized, they are vulnerable to various attacks such as Denial-of-Service (DoS) and cyber-physical attack. We will review now the application of reinforcement learning in resolving several security concerns.

Jamming Attacks

Attackers or jammers transmit high-powered radio frequency (RF) jamming signals to interfere with legitimate communication networks, thus reducing the SINR of legitimate receivers. It is difficult for users to select an appropriate frequency channel without being aware of the radio channel, model and jamming methods, as well as to decide how to exit and avoid the attack. The authors of [64] suggest the use of DQN to find an optimal anti-jamming power management strategy. The model is an IoT network with one jammer and IoT computers. The jammer is able to observe the transmitter's communications and selects a jamming technique to reduce the SINR at the receiver. To maximise its utility, the transmitter selects the transmit power level as part of its action. The utility is the difference

that the transmission creates between the SINR and the cost of energy consumption. The agent is the transmitter, and the state consists of the SINR calculated at the last time step on its receiver. In order to optimise its expected cumulative discounted reward, DQN using a CNN is used to find an power control policy for the transmitter.

Cyber-physical Attacks

An attacker manipulates data in cyber-physical attacks in order to change control signals in the system. Autonomous systems such as Intelligent Transport Systems (ITSs) are a typical example of targets for such an attack, which raises the risk of accidents in autonomous vehicles (AVs). In order to minimise the spacing deviation, a lot of vehicular communication security algorithms can be used. However, in these algorithms, the attacker's actions are believed to be stable, which is not realistic. Based on the time-varying observations of the behaviour of the attacker, reinforcement learning can improve the action choices. A framework involving a cloud and a collection of IoT devices is modelled by the authors of [65]. The IoT devices produce signals and relay them to the cloud. The cloud uses the signals obtained to estimate and monitor the operation of the IoT devices. The cloud uses Long Short Term Memory (LSTM) [66] units to detect an attack, and uses features or fingerprints such as flatness, skewness, and kurtosis of the signals of the IoT devices for authentication. The cloud can only authenticate a small number of vulnerable IoT devices due to the intensive computing needed in the process. A deep RL algorithm that allows the cloud to determine which IoT devices to authenticate is suggested. The state of the cloud involves the attacker's behaviour in the past time steps on the IoT devices. The reward is a function of the IoT devices' data values. To find an optimal policy, the agent uses an LSTM unit, whose input is the state of the cloud, and whose output contains IoT device attack probabilities.

2.2.5 Traffic Engineering and Routing

Traffic Engineering (TE) refers to Network Utility Maximization (NUM) in communication networks - optimizing the direction of data traffic routing, given the number of network flows from source to destination nodes. Traditional problems associated with NUM are mainly model-based. The network environment, however, is becoming more complex and dynamic with the developments of wireless communication technology, making it difficult to model, forecast, and monitor. Deep RL methods offer a viable and practical way to develop experience-driven and model-free approaches that can learn from past observations and adapt to the complexity of a wireless network. The authors of [67] propose an actor-critic approach for solving the problem of routing optimization. The state of the system is represented by the bandwidth request between each source-destination pair, and the reward is a function of the mean delay in the network. In [68], the same problem is solved by combining two techniques in conjunction with DDPG [62] for TE problems: TE-aware exploration and actor-critic-based PER methods. The TE-aware exploration leverages the shortest path algorithm and NUM-based solution as the baseline during exploration.

The authors model autonomous navigation of a single UAV in a large-scale, unknown, complex environment as a POMDP in [69], which is then solved by an actor-critic approach. The state of the device includes its distances and angles of orientation to surrounding obstacles, and the distance and angle between its current location and the target. The UAV's actions include: turn left, turn right, or keep ahead. The reward is composed of four parts: a) an exponential penalty term based on proximity to an obstacle, b) a linear penalty term to facilitate minimum time delay, c) the transition, and d) direction rewards, if the UAV is getting close to the target position. They use Recurrent Deterministic

Policy Gradient (RDPG) [70] and approximate the actor and critic using Recurrent Neural Netowrks (RNNs).

2.2.6 Resource Sharing and Scheduling

Enhancements to device capability can be focused on optimising the sharing and scheduling of resources between multiple wireless nodes. Integrating reinfrocement learning into 5G networks would revolutionize resource sharing and scheduling schemes from model-based to model-free approaches, and meet various application demands by adapting to the network environment. In a massive multi-user MIMO scheme, the authors of [71] optimise user scheduling, which is responsible for allocating resource blocks to BSs and mobile users, taking into account the channel conditions and QoS specifications. The system state is an indicator of the average spectrum efficiency. The action of the scheduler is a set of scheduling parameters to maximize the return, as a function of the average spectrum efficiency. A policy gradient method is deployed to learn a policy function directly from trajectories generated by the current policy.

The network infrastructure is comparatively static, e.g., cache, computation, and radio resources, while the upper layer Virtualized Network Functions (VNFs) are dynamic to accommodate application-specific service demands that vary in time. Network slicing [72] is the concept of dividing the network resources into multi-layer slices, managed by different service renderers independently with minimal conflicts. The concept of Service Function Chaining (SFC) refers to orchestrating various VNFs to provide the necessary functionality and QoS provisioning. A deep RL approach for QoS/QoE aware SFC in VNF-enabled 5G systems is proposed by the authors of [73].

Resource allocation and scheduling issues in computing clusters and database systems

are studied by the authors of [74]. DeepRM is an online deep RL-based solution that utilises policy gradient methods to control resources in computer systems. In [75], the authors use the actor-critic method to address the scheduling problem in general-purpose, distributed data stream processing systems. There are several threads, processes, and devices in the system model. The state of the system consists of the current scheduling decision and each data source's workload. The scheduling problem is to allocate each thread to a machine's process. The scheduler agent decides the allocation of each thread, with the intention of minimizing the average processing time. Three components are included in the deep RL ageent: (a) an actor network, (b) an optimizer generating the K-Nearest Neighbors (K-NN) set for the output action of the actor network, and (c) the critic network. The action with the maximum Q-value from the K-NN set is selected.

Chapter 3

Problem Formulation

3.1 System model

A simplified mobile edge computing (MEC) system consists of an edge server and several mobile users accessing that server (see Fig. 3.1). Mobile users independently generate service requests according to a Poisson process. The rate of requests and the number of users may also change with time. The edge server takes CPU resources to serve each request from mobile users. The request is buffered in a queue before it is served. When a new request comes, the server has the option to offload the request (see Fig. 3.2). The mathematical model of the edge server and the mobile users is presented below.

Edge server

Let $X_t \in \{0, 1, \ldots, \mathsf{X} - 1\}$ denote the number of service requests buffered in the queue, where X denotes the size of the buffer. Let $L_t \in \{0, 1, \ldots, \mathsf{L} - 1\}$ denote the CPU load at the server where L is the capacity of the CPU. We assume that the CPU has k cores.

We assume that the requests arrive according to a (potentially time-varying) Poisson process with rate λ. If a new request arrives when the buffer is full, the request is offloaded

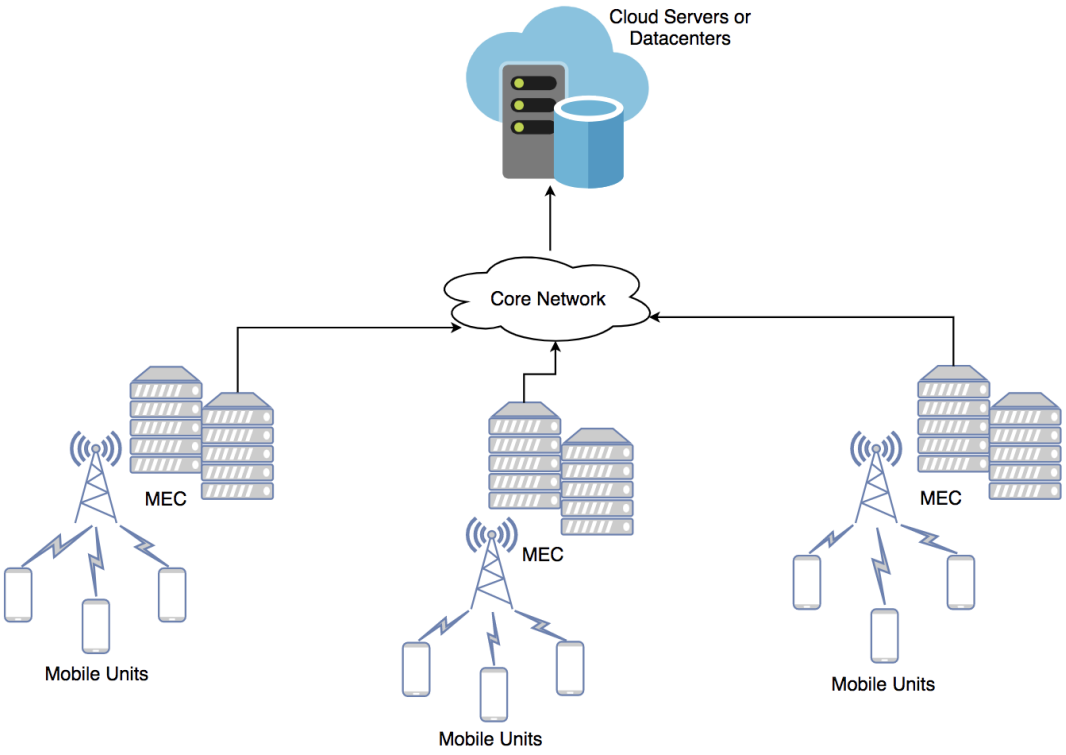


Figure 3.1: A mobile edge computing (MEC) system. User may be mobile and will connect to the closest edge server.The MEC servers are connected to the backend cloud server or datacenters through the core network.

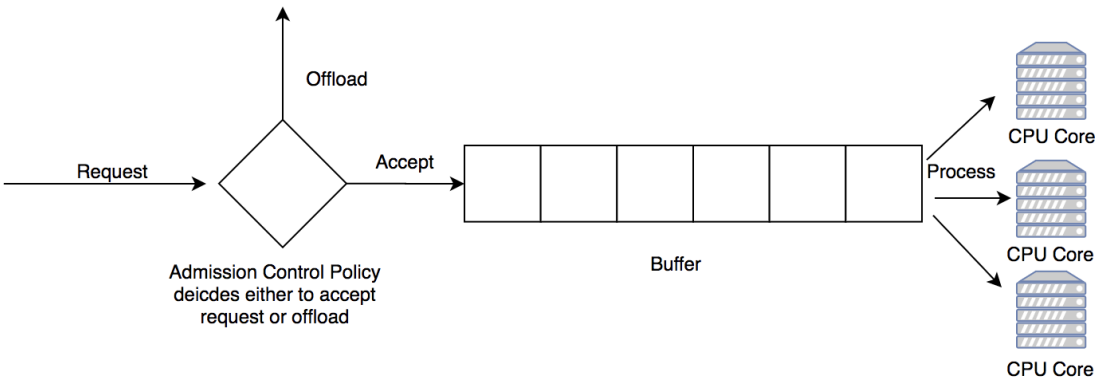


Figure 3.2: System model of admission control in a single edge server.

to another server. If a new request arrives when the buffer is not full, the server has the option to either accept or offload the request.

The server can process up to a maximum of k requests from the head of the queue.

Processing each request requires CPU resources for the duration for which the request is being served. The required CPU resources is a random variable $R \in \{1, \ldots, \mathsf{R}\}$ with probability mass function P. The realization of R is not revealed until the server starts working on the request. The duration of service is an exponentially distributed random variable with rate μ.

Let $\mathcal{A} = \{0, 1\}$ denote the action set. Here $A_t = 1$ means that the server decides to offload the request while $A_t = 0$ means that the server accepts the request.

Traffic model for mobile users

We consider multiple models for traffic.

- **Scenario 1:** All users generate requests according to the same rate λ and the rate does not change over time. Thus, the rate at which requests arrive is λN.
- **Scenario 2:** In this scenario, we assume that all users generate requests according to rate λ_{M_t}, where M_t is a global state which changes over time. Thus, the rate at which requests arrive in a state m is $\lambda_m N$.
- **Scenario 3:** Each user n has a state $M_t^n \in \{1, \ldots, \mathsf{M}\}$. When the user n is in state m, it generates requests according to rate λ_m. The state M_t^n changes over time. Thus, the rate at which requests arrive at the server is $\sum_{n=1}^{N} \lambda_{M_t^n}$.
- **Time-varying users:** In each of the scenarios above, we can consider the case when the number of users is not fixed and changes over time. We call them Scenario 4, 5, and 6 respectively.

Cost and the optimization framework

The system incurs three types of a cost:

- a holding cost of h per unit time when a request is buffered in the queue but is not being served.
- a running cost of $c(\ell)$ per unit time for running the CPU at a load of ℓ.
- a penalty of $p(\ell)$ for offloading a packet at CPU load ℓ.

We combine all these costs in a cost function

$$\rho(x, \ell, a) = h[x - k]^+ + c(\ell) + p(\ell)\mathbb{1}\{a = 1\}, \tag{3.1}$$

where $[x]^+$ is a short-hand for $\max\{x, 0\}$, $\mathbb{1}\{\cdot\}$ is the indicator function and a is the action. Note that to simplify the analysis, we have assumed that the server always serves $\min\{X_t, k\}$ requests.

Whenever a new request arrives, the server uses a memoryless policy $\pi\colon \{0, 1, \ldots, \mathsf{X} - 1\} \times \{0, 1, \ldots, \mathsf{L} - 1\} \to \{0, 1\}$ to choose an action

$$A_t = \pi_t(X_t, L_t).$$

The performance of a policy π starting from initial state (x, ℓ) is given by

$$V^\pi(x, \ell) = \mathbb{E}\left[\int_0^\infty e^{-\gamma t}\rho(X_t, L_t, A_t)dt \;\middle|\; X_0 = x, L_0 = \ell\right], \tag{3.2}$$

where $\gamma > 0$ is the discount rate and the expectation is with respect to the arrival process, CPU utilization, and service completions.

The objective is to minimize the expected cost (3.2) for the different traffic scenarios listed above. We are particularly interested in the setting where the arrival rate and potentially other components of the model such as the resource distribution are not known to the system designer and change during the operation of the system.

Solution framework

When the model parameters (λ, N, μ, P, k) are known and time-homogeneous, the optimal policy π can be computed using dynamic programming. However, in a real system, these parameters may not be known, so we are interested in developing a reinforcement learning algorithm which can learn the optimal policy based on the observed per-step cost.

In principle, when the model parameters are known, Scenarios 2 and 3 can also be solved using dynamic programming. However, the state of such dynamic programs will include the state M_t of the system (for Scenario 2) or the states $(M_t^n)_{n=1}^N$ of all users (for Scenario 3). Typically, these states change at a slow time-scale. So, we will consider reinforcement learning algorithms which do not explicitly keep track of the states of the user and check if the algorithm can adapt quickly whenever the arrival rates change.

3.2 Dynamic programming to identify an optimal admission control policy

When the arrival process is time-homogeneous, the process $\{X_t, L_t\}_{t\geq 0}$ is a finite-state continuous-time Markov decision process (MDP) controlled through $\{A_t\}_{t\geq 0}$. To specify the controlled transition probability of this MDP, we consider the following two cases.

First, if there is a new arrival at time t, then

$$\mathbb{P}(X_t = x', L_t = \ell' \mid X_{t^-} = x, L_{t^-} = \ell, A_t = a) = \begin{cases} P(\ell' - \ell), & \text{if } x' = x+1 \text{ and } a = 0 \\ 1, & \text{if } x' = x,\ \ell' = \ell, \text{ and } a = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

We denote this transition function by $q_+(x', \ell'|x, \ell, a)$. Note that the first term $P(\ell' - \ell)$ denotes the probability that the accepted request required $(\ell' - \ell)$ CPU resources.

Second, if there is a departure at time t,

$$\begin{aligned}&\mathbb{P}(X_t = x', L_t = \ell' \mid X_{t^-} = x, L_{t^-} = \ell)\\ &= \begin{cases} P(\ell - \ell'), & \text{if } x' = [x-1]^+ \\ 0, & \text{otherwise.} \end{cases}\end{aligned} \tag{3.4}$$

We denote this transition function by $q_-(x', \ell'|x, \ell)$. Note that there is no decision to be taken at the completion of a request, so the above transition does not depend on the action. In general, the reduction in CPU utilization will correspond to the resources needed for the request whose service was completed. However, keeping track of those resources would mean that we would need to expand the state to include $(R_1, \ldots, R_k)$, where R_i denotes the resources required by the request processed by CPU i. In order to avoid such an increase in the state dimension, we assume that when a request is completed, CPU utilization reduces by amount $\ell - \ell'$ with probability $P(\ell - \ell')$.

We combine (3.3) and (3.4) into a single controlled transition probability function from state (x, ℓ) to state (x', ℓ') given by

$$\begin{aligned}p(x', \ell' \mid x, \ell, a) = &\frac{\lambda}{\lambda + \min\{x, k\}\mu} q_+(x', \ell' \mid x, \ell, a)\\ &+ \frac{\min\{x, k\}\mu}{\lambda + \min\{x, k\}\mu} q_-(x', \ell' \mid x, \ell).\end{aligned} \tag{3.5}$$

Let $\nu = \lambda + k\mu$ denote the uniform upper bound on the transition rate at the states. Then, using the *uniformization technique* [8, 9], we can convert the above continuous time discounted cost MDP into a discrete time discounted cost MDP with discount factor $\beta = \nu/(\gamma + \nu)$, transition probability matrix $p(x', \ell'|x, \ell, a)$ and per-step cost

$$\bar{\rho}(x, \ell, a) = \frac{1}{\gamma + \nu} \rho(x, \ell, a).$$

Therefore, we have the following.

Theorem 1 *Consider the following dynamic program*

$$V(x,\ell) = \min\{Q(x,\ell,0), Q(x,\ell,1)\}, \tag{3.6}$$

where

$$\begin{aligned} Q(x,\ell,0) = \frac{1}{\gamma+\nu}\big[h[x-k]^+ + c(\ell)\big] \\ + \beta\Bigg[\frac{\lambda}{\lambda+\min\{x,k\}\mu}\sum_{r=1}^{\mathsf{R}} P(r)V([x+1]_{\mathsf{X}},[\ell+r]_{\mathsf{L}}) \\ + \frac{\min\{x,k\}\mu}{\lambda+\min\{x,k\}\mu}\sum_{r=1}^{\mathsf{R}} P(r)V([x-1]^+,[\ell-r]^+)\Bigg], \end{aligned}$$

and

$$\begin{aligned} Q(x,\ell,1) = \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell) + p(\ell)\big] \\ + \beta\frac{\min\{x,k\}\mu}{\lambda+\min\{x,k\}\mu}\sum_{r=1}^{\mathsf{R}} P(r)V([x-1]^+,[\ell-r]^+), \end{aligned}$$

where $[x]_{\mathsf{B}}$ *denotes* $\min\{x,\mathsf{B}\}$.

Let $\pi(x,\ell) \in \mathcal{A}$ *denote the argmin of the right hand side of* (3.6)*. Then, the time-homogeneous policy* $\pi(x,\ell)$ *is optimal for the original continuous-time optimization problem.* □

PROOF The equivalence between the continuous and discrete time MDPs follows from the uniformization technique [8, 9]. The optimality of the time-homogeneous policy π follows from the standard results for MDPs [10]. ■

Thus, for all practical purposes, the decision maker has to solve a discrete-time MDP, where decisions need to be taken at the time points when a new request arrives. From now on, we will ignore the $1/(\gamma+\nu)$ term in front of the per-step cost and assume that it has been absorbed in the constant h, and the functions $c(\cdot)$, $p(\cdot)$.

When the system parameters are known, the above dynamic program can be solved using standard techniques such as value iteration [20], policy iteration [21], or linear programming. However, in practice, the system parameters may slowly change over time. Therefore, instead of pursuing a planning solution, we consider reinforcement learning solutions which can adapt to time-varying environments.

3.3 Structure-aware reinforcement learning

Although, in principle, the optimal admission control problem formulated above can be solved using deep RL algorithms, such algorithms require significant computational resources to train, are brittle to the choice of hyperparameters, and generate policies which are difficult to interpret. For these reasons, we investigate an alternate class of RL algorithms which circumvents these limitations, by leveraging the special structure of the problem that we want to solve.

3.3.1 Structure of the optimal policy

We first establish monotonicity properties of the value function and the optimal policy.

Proposition 1 *For a fixed queue length x, the value function is weakly increasing in the CPU utilization ℓ.* □

PROOF The proof is presented in Appendix A.1 ■

Proposition 2 *For a fixed queue length x, if it is optimal to reject a request at CPU utilization ℓ, then it is optimal to reject a request at all CPU utilizations $\ell' > \ell$.* □

PROOF The proof is presented in Appendix A.2 ■

3.3.2 The SALMUT algorithm

Proposition 2 shows that the optimal policy can be represented by a threshold vector $\tau = (\tau(x))_{x=0}^{\mathsf{X}}$, where $\tau(x) \in \{0, \ldots, \mathsf{L}\}$ is the smallest value of the CPU utilization such that it is optimal to accept the packet for CPU utilization less than or equal to $\tau(x)$ and reject it for utilization greater than $\tau(x)$.

The SALMUT algorithm was proposed in [15] to exploit a similar structure in admission control for multi-class queues. It was originally proposed for the average cost setting. We present a generalization to the discrete-time setting.

We use π_τ to denote a threshold-based policy with the parameters $(\tau(x))_{x=0}^{\mathsf{X}}$ taking values in $\{0, \ldots, \mathsf{L}\}^{\mathsf{X}+1}$. The key idea behind SALMUT is that, instead of deterministic threshold-based policies, we consider a random policy parameterized with parameters taking value in the compact set $[0, \mathsf{L}]^{\mathsf{X}+1}$. Then, for any state (x, ℓ), the randomized policy π_τ chooses action $a = 0$ with probability $f(\tau(x), \ell)$ and chooses action $a = 1$ with probability $1 - f(\tau(x), \ell)$, where $f(\tau(x), \ell)$ is any continuous decreasing function w.r.t ℓ, which is differentiable in its first argument, e.g., the sigmoid function

$$f(\tau(x), \ell) = \frac{\exp((\tau(x) - \ell)/T)}{1 + \exp((\tau(x) - \ell)/T)}, \tag{3.7}$$

where $T > 0$ is a hyper-parameter (often called "temperature").

Fix an initial state (x_0, ℓ_0) and let $J(\tau)$ denote the performance of policy π_τ when starting from the initial state (x_0, ℓ_0). Let V_τ and Q_τ denote the value function and action-value function of policy π_τ.

Now, from the policy gradient theorem [11], we know that

$$\nabla J(\tau) = \sum_{x=0}^{\mathsf{X}} \sum_{\ell=0}^{\mathsf{L}} \mu(x, \ell) \sum_{a \in \mathcal{A}} \frac{\delta \pi_\tau(a|x, \ell)}{\delta \tau} Q_\tau(x, \ell, a) \tag{3.8}$$

where $\mu(x, \ell)$ is the occupancy measure on the states starting from the initial state (x_0, ℓ_0)

and

$$\nabla Q(x,\ell;\tau) = \sum_{x'=0}^{\mathsf{X}} \sum_{\ell'=0}^{\mathsf{L}} \nabla p^{(\tau)}(x',\ell'|x,\ell) \times \sum_{a\in\mathcal{A}} \pi_\tau(a|x,\ell) Q_\tau(x,\ell,a). \tag{3.9}$$

Therefore, an unbiased estimator of $\nabla J(\tau)$ is given by

$$\nabla p^{(\tau)}(x',\ell'|x,\ell) Q_\tau(x,\ell,a), \quad \text{where } a \sim \pi_\tau(\cdot|x,\ell). \tag{3.10}$$

Note that

$$\frac{\delta \pi_\tau(a|x,\ell)}{\delta \tau} = (-1)^a \nabla f(\tau(x),\ell). \tag{3.11}$$

Combining (3.8) with (3.11), we get that

$$(-1)^a \nabla f(\tau(x),\ell)\big[\bar{\rho}(x,\ell,a) + \beta V_\tau(x',\ell')\big], \tag{3.12}$$

where $a \sim \pi_\tau(\cdot|x,\ell)$ is an unbiased estimator of $\nabla J(\tau)$.

Thus, we can use the standard two time-scale Actor-Critic algorithm [11] to simultaneously learn the policy parameters τ and the action-value function Q as follows. We start with an initial guess Q_0 and τ_0 for the action-value function and the optimal policy parameters. Then, we update the action-value function using temporal difference learning:

$$Q_{n+1}(x,\ell,a) = Q_n(x,\ell,a) + b_n^1\big[\bar{\rho}(x,\ell,a) + \beta \min_{a'\in A} Q_n(x',\ell',a') - Q_n(x,\ell,a)\big], \tag{3.13}$$

and update the policy parameters using stochastic gradient descent while using (3.12) as the unbiased estimator of $\nabla J(\tau)$:

$$\tau_{n+1}(x) = \mathrm{Proj}\big[\tau_n(x) + b_n^2 (-1)^a \nabla f(\tau(x),\ell)\big[\bar{\rho}(x,\ell,a) + \beta \min_{a'\in\mathcal{A}} Q(x',\ell',a')\big]\big], \tag{3.14}$$

where Proj is a projection operator which clips the values to the interval $[0,\mathsf{L}]$ and $\{b_n^1\}_{n\geq 0}$ and $\{b_n^2\}_{n\geq 0}$ are learning rates which satisfy the standard conditions on two time-scale learning: $\sum_n b_n^k = \infty$, $\sum_n (b_n^k)^2 < \infty$, $k \in \{1,2\}$, and $\lim_{n\to\infty} b_n^2/b_n^1 = 0$.

The complete approach is presented in Algorithm 1.

Algorithm 1: Two time-scale SALMUT algorithm

Result: τ

Initialize value functions $\forall x, \forall \ell$, $Q(x, \ell, a) \leftarrow 0$

Initialize threshold vector $\forall x$, $\tau(x) \leftarrow \text{rand}(0, \mathsf{L} - 1)$

Initialize start state $(x, \ell) \leftarrow (x_0, \ell_0)$

while TRUE **do**

  if EVENT == ARRIVAL **then**

    Choose action a according to Eq. (3.7)

    Update $V(x, \ell)$ according to Eq. (3.13)

    Update threshold τ using Eqs. (3.14)

    $(x, \ell) \leftarrow (x', \ell')$

  end

end

Theorem 2 *The two time-scale SALMUT algorithm described above converges almost surely and* $\lim_{n\to\infty} \nabla J(\tau_n) = 0$. □

PROOF The proof is present in Appendix B. ■

In the next chapter, we demonstrate the advantage offered by the proposed SALMUT algorithm in terms of the training time and interpretability of policies with respect to traditional DRL algorithms such as PPO and A2C. We perform empirical evaluations of the admission control policy in the node overload protection problem in two different settings - a simulated environment and a real testbed.

Chapter 4

Experimental Setup

4.1 Numerical experiments - Computer Simulations

In this section, we present detailed numerical experiments to evaluate the proposed reinforcement learning algorithm on various scenarios described in Sec. 3.1.

We consider an edge server with buffer size $\mathsf{X} = 20$, CPU capacity $\mathsf{L} = 20$, $k = 2$ cores, service-rate $\mu = 3.0$ for each core, holding cost $h = 0.12$. The CPU capacity is discretized into 20 states for utilization $0 - 100\%$, with $\ell = 0$ corresponding to a state with CPU load $\ell \in [0\% - 5\%)$, and so on.

The CPU running cost is modelled such that it incurs a positive reinforcement for being in the optimal CPU range, and a high cost for an overloaded system.

$$c(\ell) = \begin{cases} 0 & \text{for } \ell \leq 5 \\ -0.2 & \text{for } 6 \leq \ell \leq 17 \\ 10 & \text{for } \ell \geq 18. \end{cases}$$

The offload penalty is modelled such that it incurs a fixed cost for offloading to enable the offloading behavior only when the system is loaded and a very high cost when the

system is idle to discourage offloading in such scenarios.

$$p(\ell) = \begin{cases} 1 & \text{for } \ell \geq 3 \\ \\ 10 & \text{for } \ell \leq 3. \end{cases}$$

The probability mass function of resources needed per request is as follows:

$$P(r) = \begin{cases} 0.6 & \text{if } r = 1 \\ \\ 0.4 & \text{if } r = 2. \end{cases}$$

We have performed a detailed analysis on the structure of the cost function, which will be discussed in the next section.

We created a simulation environment using OpenAI's Gym environment to model the dynamics of the system. Rather than simulating the system in continuous-time, we simulate the equivalent discrete-time MDP by generating the next event (arrival or departure) using a Bernoulli distribution with probabilities and costs described in Sec. 3.2. We assume that the parameter $1/(\gamma + \nu)$ in (3.6) has been absorbed in the cost function. We assume that the discrete time discount factor $\beta = \gamma/(\gamma + \nu)$ equals 0.95.

We use the ADAM optimizer because Stochastic Gradient Descent (SGD) does not adapt well to sudden changes in request arrival distribution. ADAM [76] is a first-order gradient-based optimizer for stochastic objective functions, based on adaptive estimates of lower-order moments. ADAM is able to update its learning rate based on the change in the lower-order moments, i.e. when there is a sudden change in the request distribution. We observed in our initial experiments that the learning rate of SGD saturates to a small value after running the experiments for some time, hence it was unable to provide meaningful updates to the Q-values when the network dynamics changed drastically.

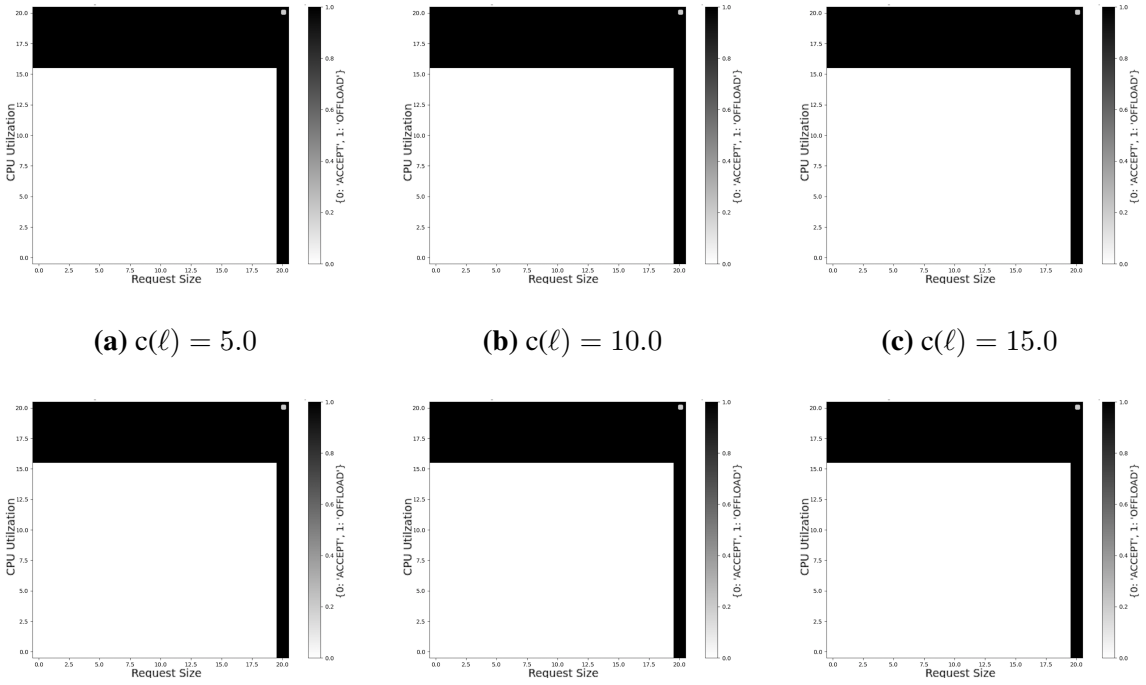
4.1.1 Choice of the Model Parameters

In order to come up with an optimal cost structure, we analyze the structure of the Dynamic Programming solution in 3.2, obtained by using Policy Iteration [11], by varying the overload cost, offload cost, holding cost, rewards, and discount factor (β) one at a time while keeping the others constants. We fix the arrival rate ($\lambda = 6.0$), processing rate ($\mu = 3.0$), number of cores ($k = 2$) and number of users ($N = 24$) throughout these experiments.

The figures in this section plot the optimal policy computed using Policy Iteration for all the states of the system. The x-axis denotes the queue/request size and y-axis denotes the CPU utilization/load. The grid spanning the x-axis and y-axis is the state of the system and the color of the grid represents the optimal action at that state. The color black means that the optimal action is to offload, whereas the color white means that the optimal action is to accept the request.

Varying the overload cost

The overload cost is the cost $c(\ell)$ incurred when the system goes into an overloaded state. In our experiments, we assume that the state goes into an overloaded state whenever the CPU load ℓ is greater that 90%, i.e. $\ell \geq 18$. We vary the overload cost from 5 to 30 at the intervals of 5, while keeping the other costs constant. We need to ensure that the overload cost is much higher than the other costs as it degrades the performance of the edge nodes. We observe in Figure 4.1 that the optimal policy does not change with the change in overload cost, as long as the overload cost is high enough.



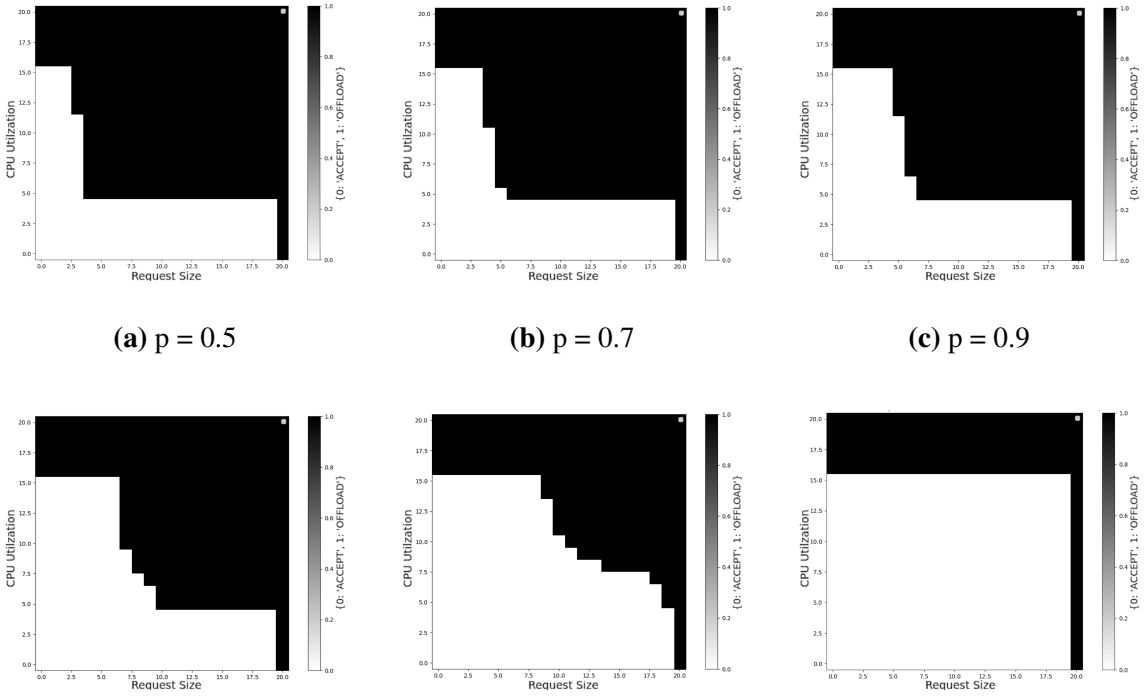
(a) c(ℓ) $= 5.0$ **(b)** c(ℓ) $= 10.0$ **(c)** c(ℓ) $= 15.0$

(d) c(ℓ) $= 20.0$ **(e)** c(ℓ) $= 25.0$ **(f)** c(ℓ) $= 30.0$

Figure 4.1: Dynamic Programming solutions obtained by varying the overload cost and keeping other parameters constant (offload cost = 1, holding cost = 0.03, reward = 0.1, β = 0.95). The x-axis denotes the request size and y-axis denotes the CPU utilization. Each grid is the state space of the system and the color of the grid cells represents the optimal action at that state: black for offload, white for accept.

Varying the Offload Costs

The offload cost is the cost incurred by the system when it offloads a packet to another MEC server, denoted by $p(\ell)\mathbb{1}\{a = 1\}$. We vary the offload cost from 0.5 to 1.5 at the intervals of 0.2, while keeping the other costs constant. We observe in Figure 4.2 that changing the offloading cost results in a significant difference in the structure of the optimal policy. When the offload cost is high, the system offloads only when the system is overloaded (reactive offloading), which is a property we do not desire in our policy (see Fig. 4.2f). When the offload cost is low, the system offloads very often, as accepting



(a) p = 0.5 **(b)** p = 0.7 **(c)** p = 0.9

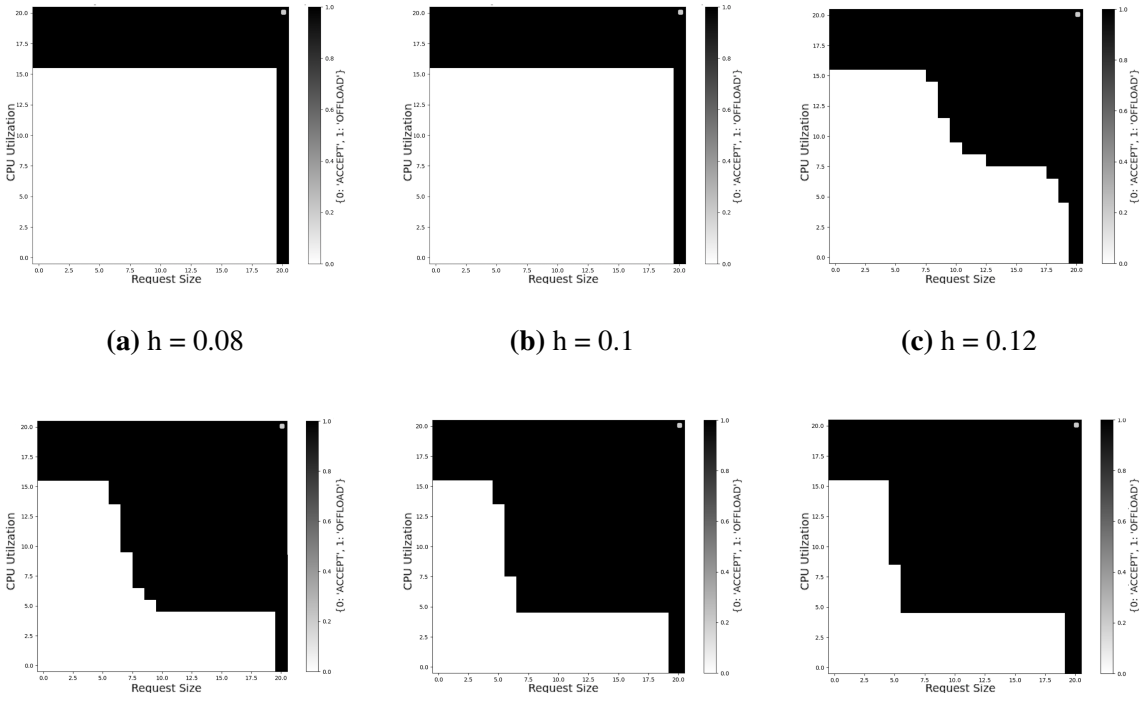
(d) p = 1.1 **(e)** p = 1.3 **(f)** p = 1.5

Figure 4.2: Solution to Dynamic Programming by varying the offload cost and keeping other parameters constant (overload cost = 10.0, holding cost = 0.15, reward = 0.1 β = 0.95)

the request would incur higher cost than offloading it, i.e. the holding cost significantly overpowers the offload cost, causing frequent offloads (see Fig. 4.2a-4.2d). We desire an intermediate value of offload cost, to ensure that offloads happen only when there is a good possibility of the system going into an overloaded state.

Varying the Holding Costs

The holding cost is the cost incurred by the system per unit time when a request is buffered in the queue but is not being served, denoted by $h[x-k]^+$. We vary the holding cost from 0.08 to 0.18 at steps of 0.02, while keeping the other costs constant. We observe in Fig. 4.3 that the optimal structure varies significantly when we change the holding cost. The

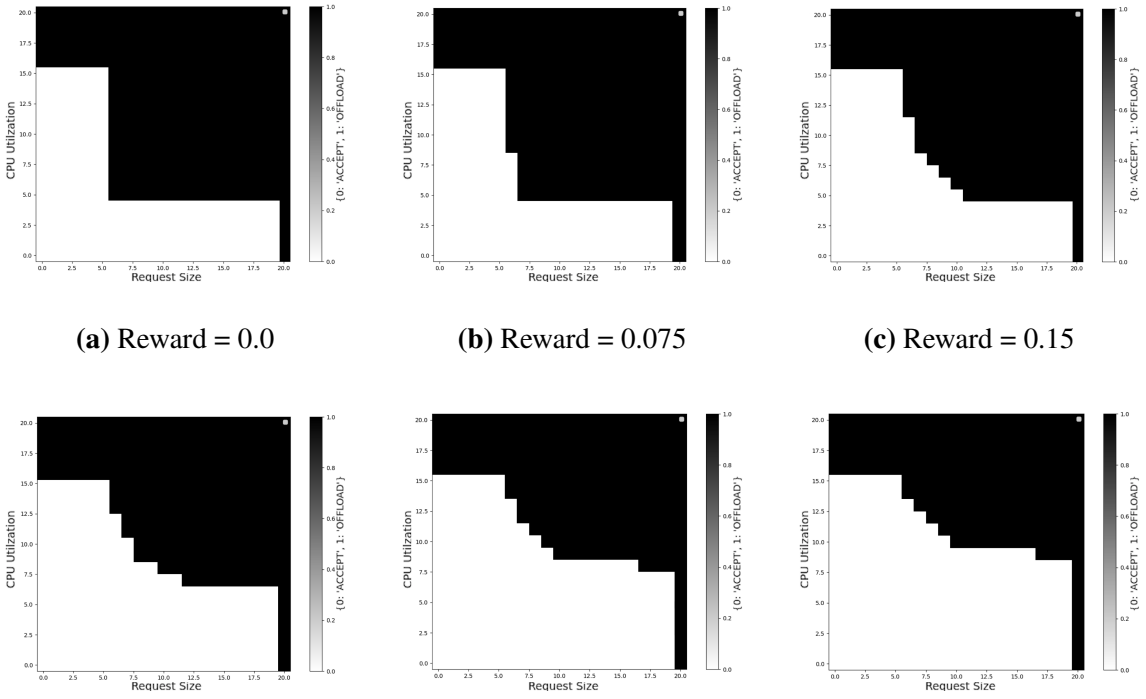


(a) h = 0.08 **(b)** h = 0.1 **(c)** h = 0.12

(d) h = 0.14 **(e)** h = 0.16 **(f)** h = 0.18

Figure 4.3: Solution to Dynamic Programming by varying the holding cost and keeping other parameters constant (overload cost = 10.0, offload cost = 1, reward = 0.1, β = 0.95).

optimal structure remains constant until the holding cost is ≤ 0.1 (Fig. 4.3a - 4.3b). Once we change the value beyond 0.1, there is a drastic change in the policy. It is because the cost incurred for holding the requests in the queue exceeds the cost of offloading those requests, hence more requests are offloaded. When the holding cost is too high, the edge nodes decide to offload most of the requests, which is not a behavior we want in the policy. We desire an intermediate value of holding cost that balances offloading in a way that avoids overloading.



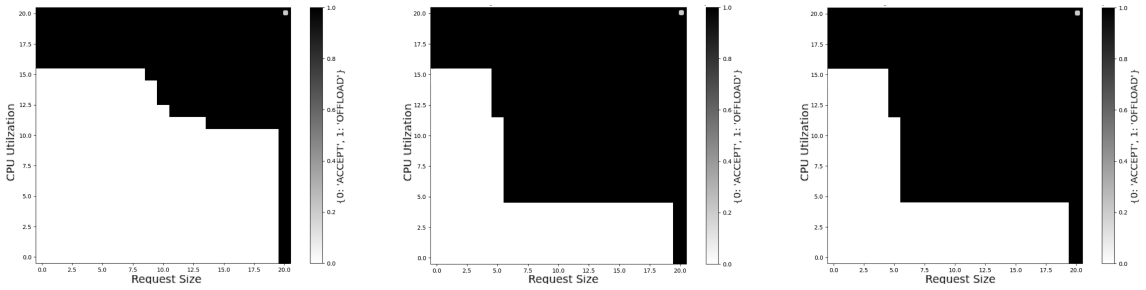
(a) Reward = 0.0 **(b)** Reward = 0.075 **(c)** Reward = 0.15

(d) Reward = 0.225 **(e)** Reward = 0.3 **(f)** Reward = 0.375

Figure 4.4: Solution to Dynamic Programming by varying the rewards and keeping other parameters constant (overload cost = 10.0, offload cost = 1.0, holding cost = 0.15, β = 0.95).

Varying the Rewards

A reward $c(\ell)$ is a positive reinforcement provided to the system if it stays in the optimal load area, i.e. $30\% \leq \ell \leq 85\%$. We vary the rewards from 0.0 to 0.375 at the intervals of 0.075, while keeping the other costs constant. We observe in Fig. 4.4 that the system offloads more when the rewards are low, as there is no incentive to stay in the mid-to-higher load regions when the buffer size is high due to the increasing effect of holding cost. The offloading decreases when the reward increases, as the edge nodes are incentivized not to offload when the load of the system is not very high and the chances of going into an overloaded state is minimal. When the reward is very high, the offloading occurs in a reactive manner, i.e. only when the system is in an overloaded state or on the verge of



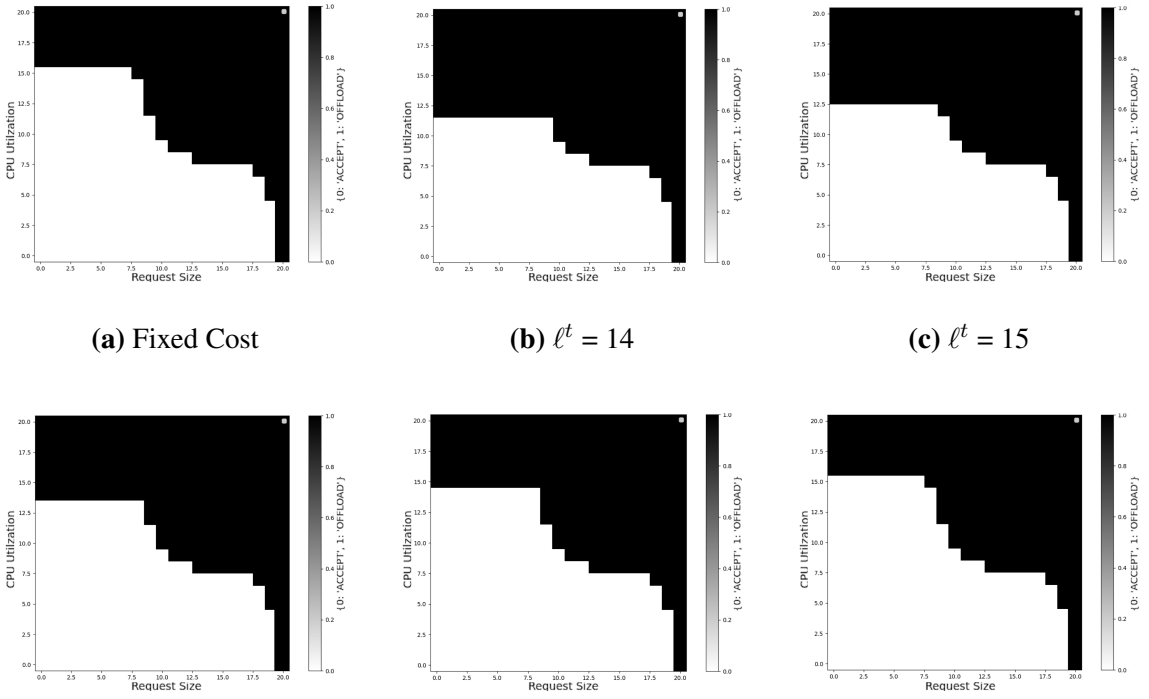
(a) β = 0.95 **(b)** β = 0.99 **(c)** β = 0.999

Figure 4.5: Solution to Dynamic Programming by varying the discount factor (β) and keeping other parameters constant (overload cost = 10.0, offload cost = 1.0, holding cost = 0.15, reward = 0.2).

entering the overloaded state.

Varying the Discount Factor

The discount factor β determines how much the costs incurred in the distant future impact the current action of the agent relative to those in the immediate future. We test for the following values of discount factor: [0.95, 0.99, 0.999], while keeping the other costs constant. We observe in Fig. 4.5 that the system offloads more when the discount factor is high and the offloading decreases when the discount factor decreases. When $\beta = 0.999$, the system offloads even when it is far from being overloaded, because there is a small chance that overload may occur and the agent wants to avoid this situation at all costs. However, this impacts the current performance because the system is underutilized. A discount factor $\beta = 0.95$ gives us a nice staircase model, where the agent performs offloading in a pro-active manner to avoid overloaded situations.



(a) Fixed Cost **(b)** ℓ^t = 14 **(c)** ℓ^t = 15

(d) ℓ^t = 16 **(e)** ℓ^t = 17 **(f)** ℓ^t = 18

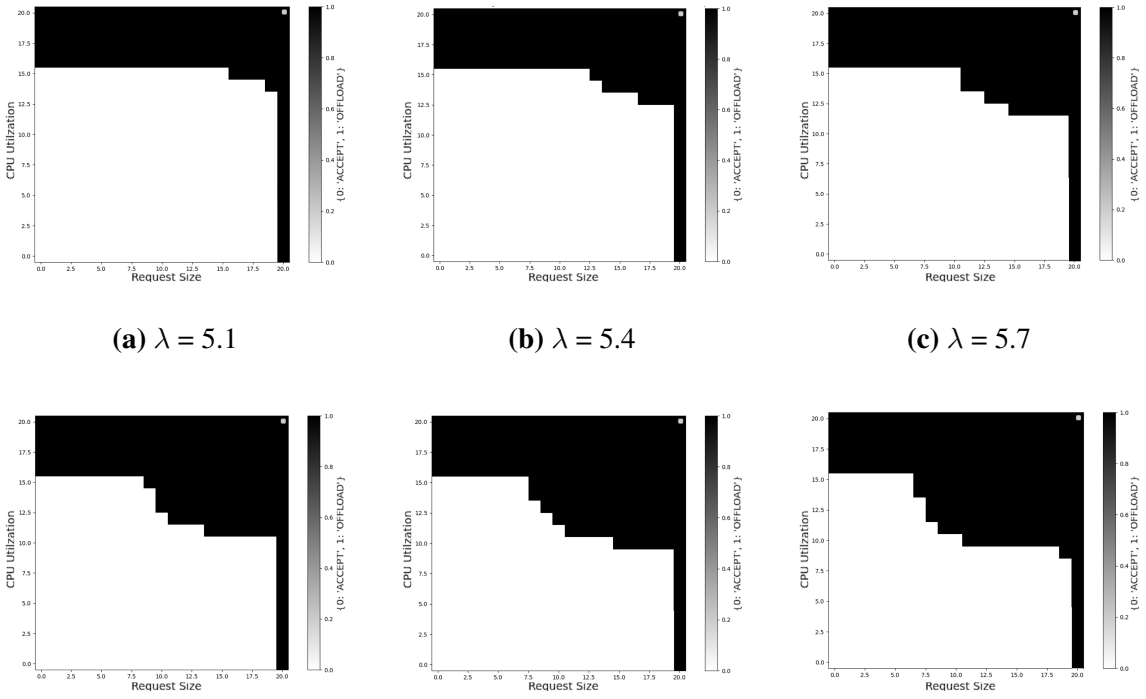
Figure 4.6: Solution to Dynamic Programming by varying the structure of the overload cost and keeping other parameters constant (overload cost = 10.0, offload cost = 1.0, holding cost = 0.12, reward = 0.1, β = 0.95).

Varying the Overload Cost Structure

In this analysis, instead of having a fixed overload cost as described in the previous section, we vary the overload cost as a step function of ℓ beyond a utilization ℓ^t

$$c(\ell) = \begin{cases} 0 & \text{for } \ell \leq 5 \\ -0.2 & \text{for } 6 \leq \ell < \ell^t \\ \ell/2 & \text{for } \ell \geq \ell^t. \end{cases}$$

We observe that changing the cost structure in this manner leads to more offloading when values of ℓ' are low (see Fig. 4.6c-4.6d) and becomes similar to the fixed overloading structure when values of ℓ' approach the threshold value of overload (see Fig. 4.6f and



(a) $\lambda = 5.1$ **(b)** $\lambda = 5.4$ **(c)** $\lambda = 5.7$

(d) $\lambda = 6.0$ **(e)** $\lambda = 6.3$ **(f)** $\lambda = 6.7$

Figure 4.7: Solution to Dynamic Programming by varying the arrival rate (λ) and keeping other parameters constant (overload cost = 10.0, offload cost = 1, holding cost = 0.12, reward = 0.1, $\beta = 0.95$).

Fig. 4.6a). We do not want a policy which will offload aggressively, hence changing the structure of the overload cost this way (with low values of ℓ') does not seem to provide good policies in our problem.

Sensitivity Analysis

In this section, we present the cost structure we choose in Sec. 4.2-A as seen in Fig. 4.7d as it exhibits the ideal structure for an optimal policy. The requests are offloaded in overloaded states, and the system also manifests proactive offloading behavior when the CPU load and queue length are in the upper-mid range, to avoid going into overload. We now analyse the susceptibility of the optimal policy to small changes in arrival rates (λ).

Ideally, we want our policy to be invariant to small changes in the environment.

We change the arrival rate by small amounts and test for values of λ from 5.1 to 6.7 in the intervals of 0.3. We observe in Fig. 4.7 that with small changes in λ, the policy does not change much but is robust enough to adapt to the change in traffic. The policies do not loose the staircase structure we are ideally looking for and adjust the offloading to suit the request arrival distribution. Therefore, we conclude that the parameters of the cost function we chose in Sec. 4.2 are correct for our problem.

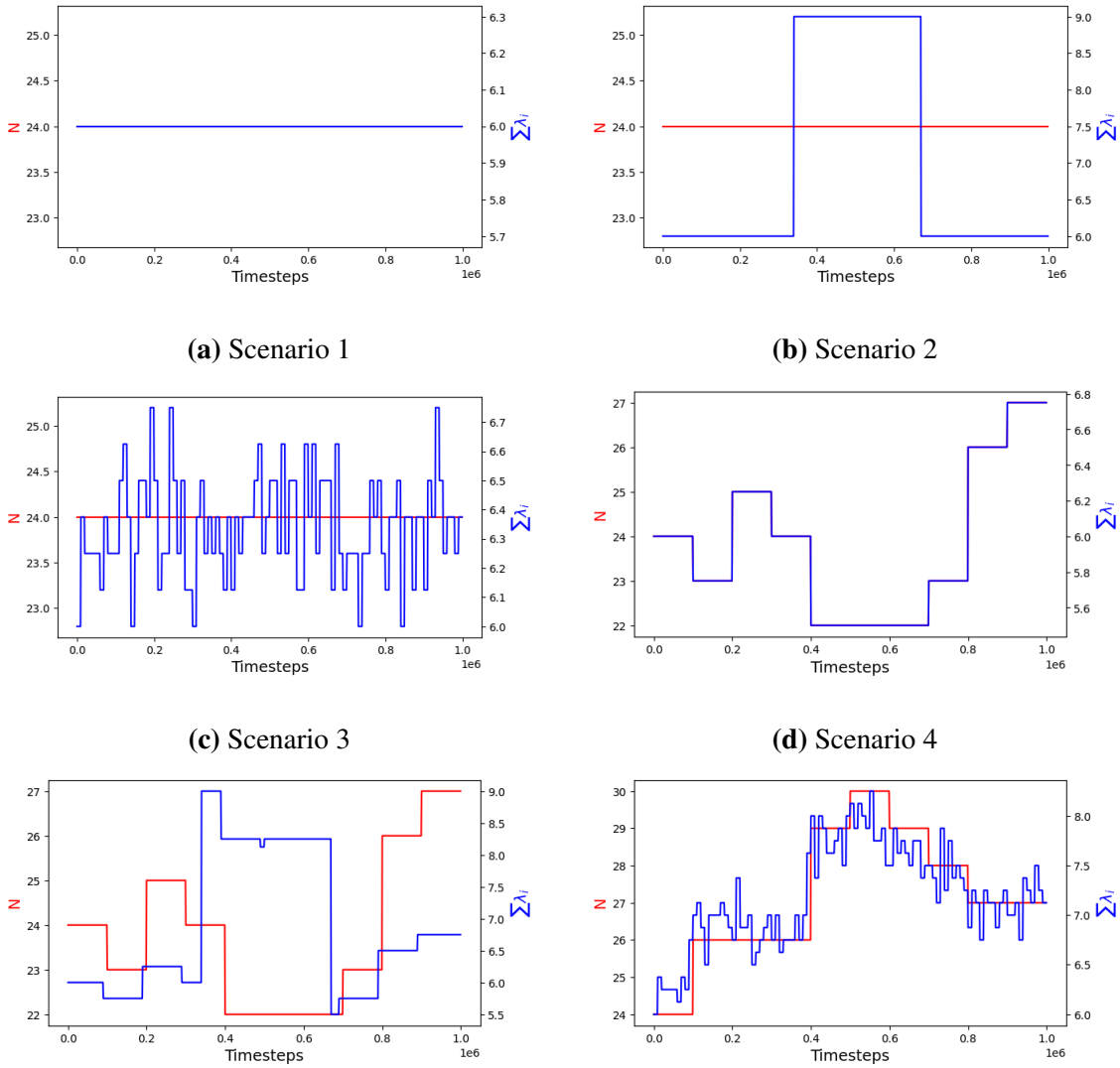
4.1.2 Simulation scenarios

We consider a number of traffic scenarios which increase in complexity and closeness to real-world settings. Each scenario runs for a horizon of $T = 10^6$. The scenarios capture variation in the transmission rate and the number of users over time. The evolution of the arrival rate λ and the number of users N for the different scenarios is shown in Fig. 4.8.

Scenario 1 This scenario tests how the learning algorithms perform in the time-homogeneous setting. We consider a system with $N = 24$ users with arrival rate $\lambda_i = 0.25$. Thus, the overall arrival rate $\lambda = N\lambda_i = 6$.

Scenario 2 This scenario tests how the learning algorithms adapt to occasional but significant changes to arrival rates. We consider a system with $N = 24$ users, where each user generates requests at rate $\lambda_{\text{low}} = 0.25$ for the interval $(0, 3.33 \cdot 10^5]$, then generates requests at rate $\lambda_{\text{high}} = 0.375$ for the interval $(3.34 \cdot 10^5, 6.66 \cdot 10^5]$, and then generates requests at rate λ_{low} again for the interval $(6.67 \cdot 10^5, 10^6]$.

Scenario 3 This scenario tests how the learning algorithms adapt to frequent but small changes to the arrival rates. We consider a system with $N = 24$ users, where each user



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

(e) Scenario 5

(f) Scenario 6

Figure 4.8: The evolution of λ and N for the different scenarios that we described. In scenarios 1 and 4, λ and N overlap in the plots.

generates requests according to rate $\lambda \in \{\lambda_{\text{low}}, \lambda_{\text{high}}\}$ where $\lambda_{\text{low}} = 0.25$ and $\lambda_{\text{high}} = 0.375$. We assume that each user starts with a rate λ_{low} or λ_{high} with equal probability. At time intervals $m \cdot 10^4$, each user toggles its transmission rate with probability $p = 0.1$.

Scenario 4 This scenario tests how the learning algorithm adapts to change in the number of users. In particular, we consider a setting where the system starts with $N_1 = 24$ user. At every 10^5 time steps, a user may leave the network, stay in the network or add another

mobile device to the network with probabilities 0.05, 0.9, and 0.05, respectively. Each new user generates requests at rate λ.

Scenario 5 This scenario tests how the learning algorithm adapts to large but occasional change in the arrival rates and small changes in the number of users. In particular, we consider the setup of Scenario 2, where the number of users changes as in Scenario 4.

Scenario 6 This scenario tests how the learning algorithm adapts to small but frequent change in the arrival rates and small changes in the number of users. In particular, we consider the setup of Scenario 3, where the number of users changes as in Scenario 4.

4.1.3 The RL algorithms

For each scenarios, we compare the performance of the following policies

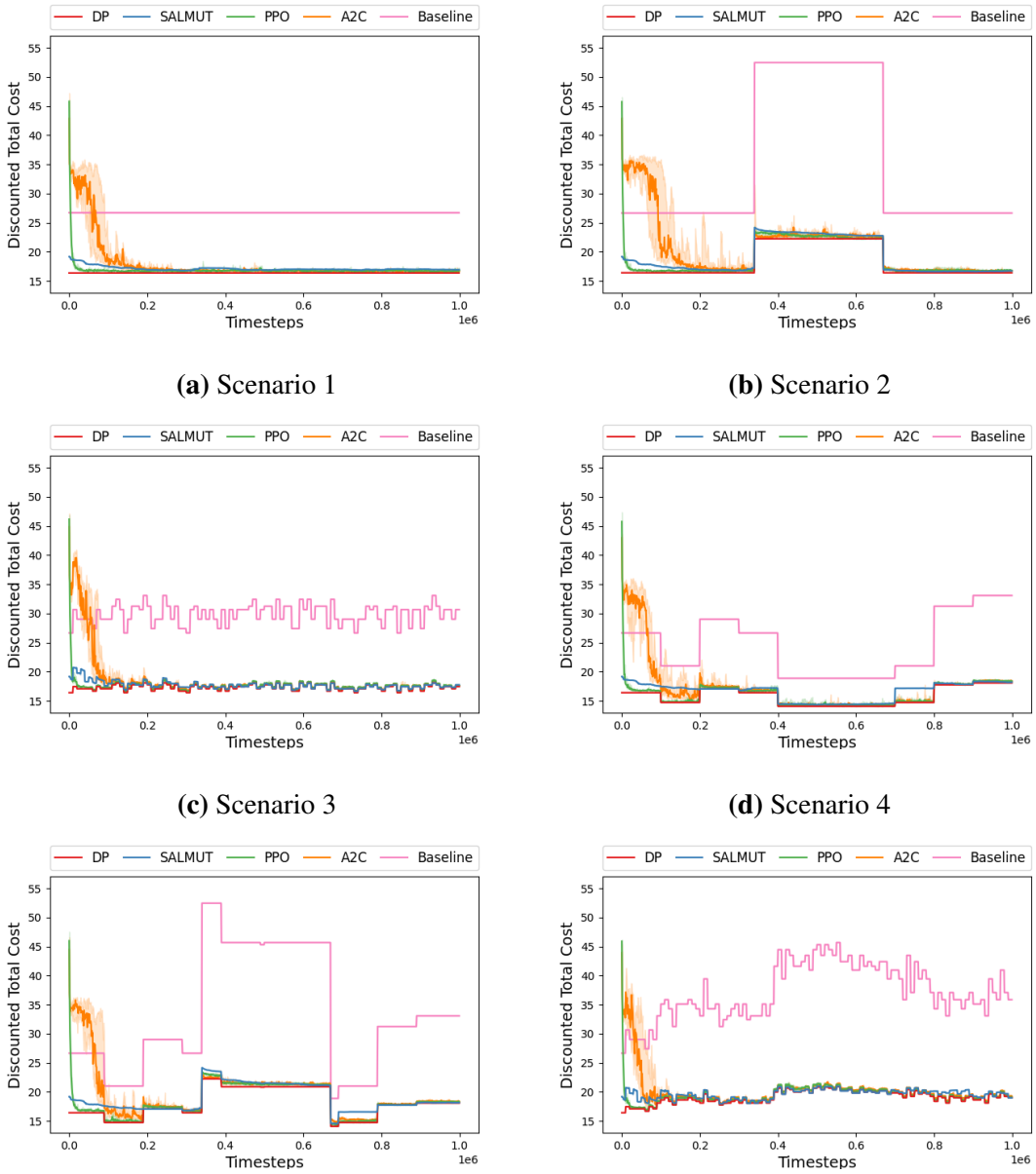
1. Dynamic Programming (DP), which computes the optimal policy using Theorem 1.

2. SALMUT, as described in Sec. 3.3

3. PPO (Proximal Policy Optimization) [16], which is a trust region policy gradient method that optimizes a surrogate objective function using stochastic gradient ascent.

4. A2C (Advantage Actor-Critic) [17], which is a two time-timescale learning algorithms where the critic estimates the value function and the actor updates the policy distribution in the direction suggested by the critic.

5. Baseline, which is a fixed-threshold based policy, where the node accepts requests when $\ell < 18$ (non-overloaded state) and offloads requests otherwise. Such static policies are currently deployed in many real-world systems.



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

(e) Scenario 5

(f) Scenario 6

Figure 4.9: Total cost as a function of time for the different algorithms.

4.1.4 Results

For each of the algorithms described above, we train SALMUT, PPO, and A2C for 10^6 steps. The performance of each algorithm is evaluated every 10^3 steps using independent rollouts of length $H = 1000$ for 100 different random seeds. The experiment is repeated for the 10 sample paths and the median return with an uncertainty band from the first to

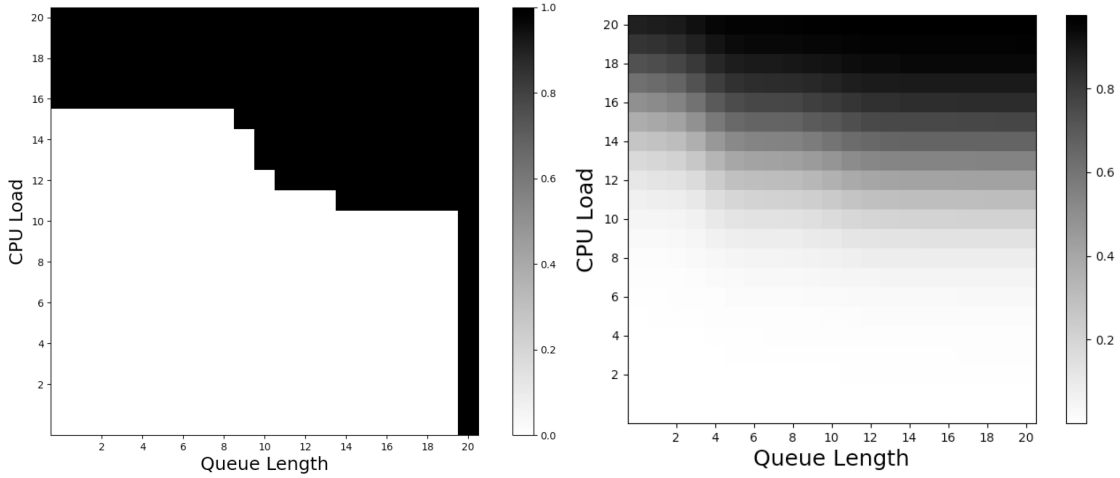
the third quartile is plotted in Fig. 4.9.

For Scenario 1, all RL algorithms (SALMUT, PPO, A2C) converge to the optimal policy relatively quickly and remain stable after convergence. Since all policies converge quickly, they are also able to adapt quickly in Scenarios 2–6 and keep track of the time-varying arrival rates and number of users. There are small differences in the peformance of the RL algorithms, but these are minor. Note that in contrast, the baseline policy of offloading when the server is overloaded performs poorly.

The plots for Scenario 1 (Fig. 4.9a) show that all the algorithms converge to the optimal performance. PPO converges to the optimal policy in less than 10^5 steps, whereas SALMUT and A2C takes around $2 \cdot 10^5$ steps to converge. Upon further analysis on the structure of the optimal policy, we observe that the structure of the optimal policy of SALMUT (Fig 4.10b) differs from that of the optimal policy computed using DP (Fig 4.10a). There is a slight difference in the structure of these policies when the buffer size (x) is high and the CPU load (ℓ) is low, which occurs because these states are reachable with a very low probability and hence SALMUT doesn't encounter these states in the simulations or encounters them too rarely to be able to learn the optimal policy in these states.

The plots from Scenario 2 (Fig. 4.9b) show similar behavior to Scenario 1 when λ is constant. When λ changes significantly, we observe that the RL algorithms are able to adapt to the drastic but stable changes in the environment. Once the load stabilizes, all the algorithms are able to readjust themselves to the changes and perform very close to the optimal policy. The plots from Scenario 3 (Fig 4.9c) show similar behavior to Scenario 1, i.e. small but frequent changes in the environment do not impact the learning performance of reinforcement learning algorithms.

The plots from Scenario 4-6 (Fig. 4.9d-4.9f) show consistent performance with varying



(a) DP $\lambda = 6$ **(b)** SALMUT $\lambda = 6$

Figure 4.10: Comparing the optimal policy and converged policy of SALMUT along one of the sample paths. The colorbar represents the probability of the offloading action.

users. The RL algorithms including SALMUT show similar performance for most of the time-steps except in cases when the load decreases, in which case SALMUT performs slightly worse than PPO and A2C. This could be due to the fact that SALMUT takes longer to adjust its more aggresive offloading policy when the system is in a more loaded state.

4.1.5 Analysis of Training Time and Policy Interpretability

The differentiating factor among these three RL algorithms is the training time and interpretability of policies. We ran our experiments on a server containing Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz. Referring to Table 4.1, SALMUT is about 28 times faster to train than PPO and 17 times faster to train than A2C. SALMUT does not require a non-linear function approximator such as a Neural Network (NN) to represent its policy, making its training very fast.

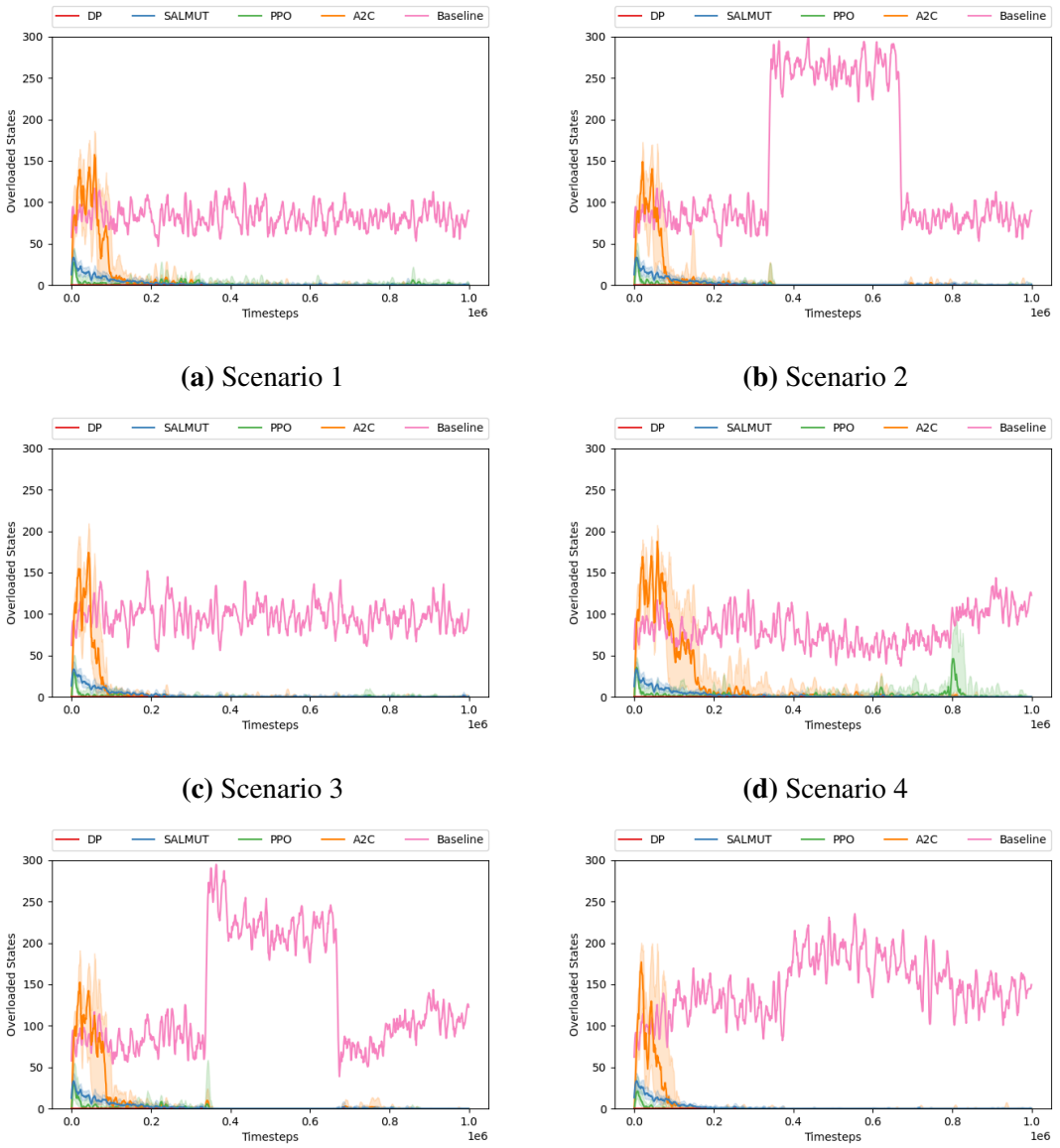
By construction, SALMUT searches for (randomized) threshold based policies. For

Table 4.1: Training time of RL algorithms

Algorithm	**Mean Time (s)**	**Std-dev (s)**
SALMUT	95.67	3.29
PPO	2673.17	23.33
A2C	1677.33	9.99

example, for Scenario 1, SALMUT converges to the policy shown in Fig. 4.10b. It is easy for a network operator to interpret such threshold based strategies and decide whether to deploy them or not. In contrast, in deep RL algorithms such as PPO and A2C, the policy is parameterized using the weights of a neural network r and it is very difficult to visualize the learned weights of such a policy and decide whether the resulting policy is reasonable. Thus, by leveraging the threshold structure of the optimal policy, SALMUT is able to learn faster and at the same time to provide threshold-based policies which are easier to interpret.

The policy of SALMUT is completely characterized by the threshold vector τ, making it storage efficient too. In addition to the vector τ, we need to store the Q-value function of every state-action pair. This results in a storage complexity of $O(|S \times A|)$. The update of the value function involves the computation of a single function corresponding to the current value of threshold (3.14). Therefore, the per iteration computational complexity is O(1). Thus the proposed algorithm provides significant improvements in storage and per-iteration computational complexity compared to traditional reinforcement learning algorithms.



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

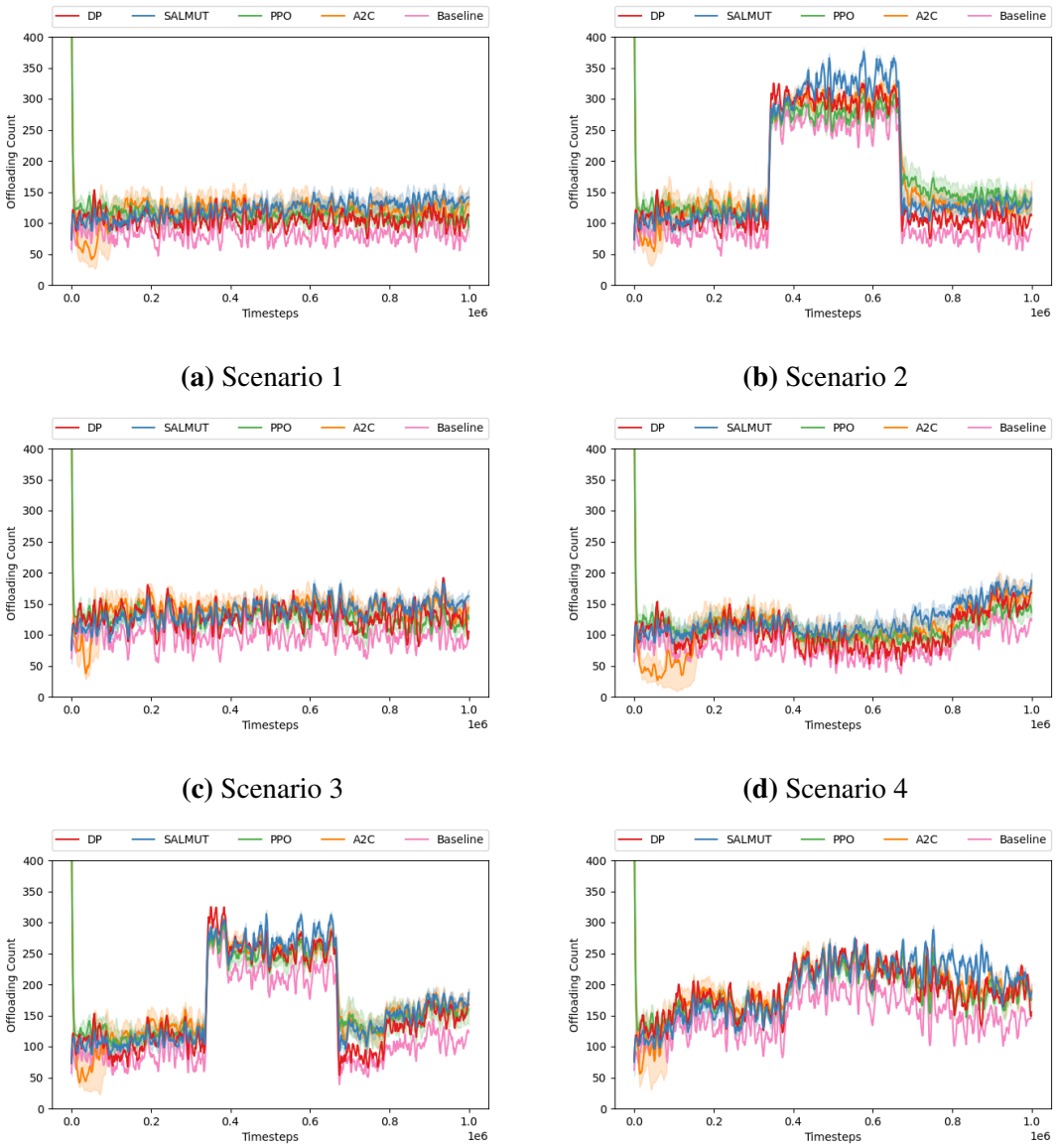
(e) Scenario 5

(f) Scenario 6

Figure 4.11: Comparing the number of times the system goes into the overloaded state at each evaluation step. The trajectory of event arrival and departure is fixed for all evaluation steps and across all algorithms for the same arrival distribution.

4.1.6 Behavioral Analysis of Policies

We performed further analysis on the behavior of the learned policy by observing the number of times the system enters into an overloaded state and offloads incoming request. Let us define C_{ov} to be the number of times the system enters into an overloaded state and



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

(e) Scenario 5

(f) Scenario 6

Figure 4.12: Comparing the number of times the system performs offloading at each evaluation step. The trajectory of event arrival and departure is fixed for all evaluation steps and across all algorithms for the same arrival distribution.

C_{off} to be the number of times the system offloads requests for every 1000 steps of training iteration. We generated a set of 10^6 random numbers between 0 and 1, defined by z_t, where t is the step count. We will use this set of random numbers to fix the trajectory of events (arrival or departure) for all the experiments in this section. Similar to the experiment in

the previous section, the number of users N and the arrival rate λ are fixed for 1000 steps and evolve according to the scenarios described in Fig. 4.8. The event is set to arrival if z_t is less than or equal to $\frac{\lambda_t}{\lambda_t+\min\{x_t,k\}\mu}$, and set to departure otherwise. These experiments were carried out during the training time for 10 different seeds. We plot the median of the number of times a system goes into an overloaded state (Fig. 4.11) and the number of requests offloaded by the system (Fig. 4.12 along with the uncertainty band from the first to the third quartile for every 1000 steps.

We observe in Fig. 4.11, that all the algorithms (SALMUT, PPO, A2C) learn not to enter into the overloaded state. As seen in the case of total discounted cost (Fig. 4.9), PPO learns this instantly, followed by SALMUT and A2C, which takes some time to learn this behavior. The observation is valid for all the different scenarios we tested. We observe that for Scenario-4, PPO enters the overloaded state at around $0.8 \cdot 10^6$, which is due to the fact the $\sum_i \lambda_i$ increases drastically at that point (seen in Fig. 4.8d), and we also see its effect on the cost in Fig. 4.9d at that time. The baseline algorithm, on the other hand, enters into the overloaded state quite often.

We observe in Fig. 4.12, that the algorithms (SALMUT, PPO, A2C) learn to adjust their offloading rate to avoid the overloaded states. The number of times that requests are offloaded is directly proportional to the total arrival rate of all the users at that time. When the arrival rate increases, the number of times the offloading occurs also increases in the interval. We see that even though more requests are offloaded by the RL algorithms than by the baseline algorithm in all scenarios and timesteps, the difference between the number of times they offload is not significantm implying that the RL algorithms learn policies that offload at the right moment, as to not lead the system into an overloaded state. We perform further analysis of this behavior for the real test bed (see Fig. 4.17) and the results

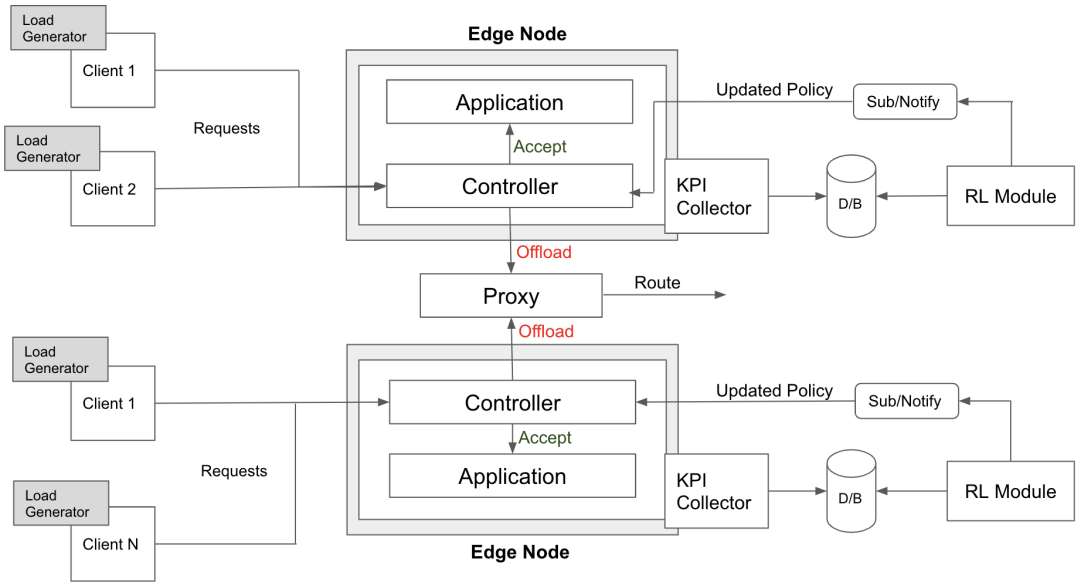


Figure 4.13: Architecture for the realistic testbed we run the experiments on.

are similar for the simulations too.

4.2 Numerical experiments - Docker Testbed

We test our proposed algorithm on a testbed resembling the MEC architecture in Fig. 3.1, but without the core network and backend cloud server for simplicity. We consider an edge node which serves a single application. Both the edge nodes and clients are implemented as containerized environments in a virtual machine. The overview of the testbed is shown in Fig. 4.13. The load generator generates requests for each client independently according to a time-varying Poisson process. The requests at the edge node are handled by the controller which decides either to accept the request or offload the request based on the policy for the current state of the edge node. If the action is ”accept”, the request is added to the request queue of the edge node, otherwise the request is offloaded to another healthy edge node via the proxy network. The Key Performance Indicator (KPI) collector copies the KPI metrics into a database at regular intervals. The RL modules uses these metrics to update

its policies. The Subscriber/Notification (Sub/Notify) module notifies the controller about the updated policy. The controller now uses the updated policy to serve all future requests.

In our implementation, the number of clients N served by an edge node and the request rate of the clients λ is constant for at-least 100 seconds. We define a step to be the execution of the testbed for 100 seconds. Each request runs a workload on the edge node and consumes CPU resources R, where R is a random variable. The states, actions, costs, next states for each step are stored in a buffer in the edge node. After the completion of a step, the KPI collector copies these buffers into a database. The RL module is then invoked, which loads its most recent policy and other parameters, and trains on this new data to update its policy. Once the updated policy is generated, it is copied in the edge node and is used by the controller for serving the requests for the next step.

We run our experiments for a total of 1000 steps, where N and λ evolve according to Fig. 4.8 for different scenarios, similar to the previous set of experiments. We consider an edge server with buffer size $\mathsf{X} = 20$, CPU capacity $\mathsf{L} = 20$, $k = 2$ cores, service-rate $\mu = 3.0$ for each core, holding cost $h = 0.12$. The CPU capacity is discretized into 20 states for utilization $0 - 100\%$, similar to the previous experiment. The CPU running cost is $c(\ell) = 30$ for $\ell \geq 18$, $c(\ell) = -0.2$ for $6 \leq \ell \leq 17$, and $c(\ell) = 0$ otherwise. The offload penalty is $p = 1$ for $\ell \geq 3$ and $p = 10$ for $\ell < 3$. We assume that the discrete time discount factor $\beta = \alpha/(\alpha + \nu)$ equals 0.99.

4.2.1 Results

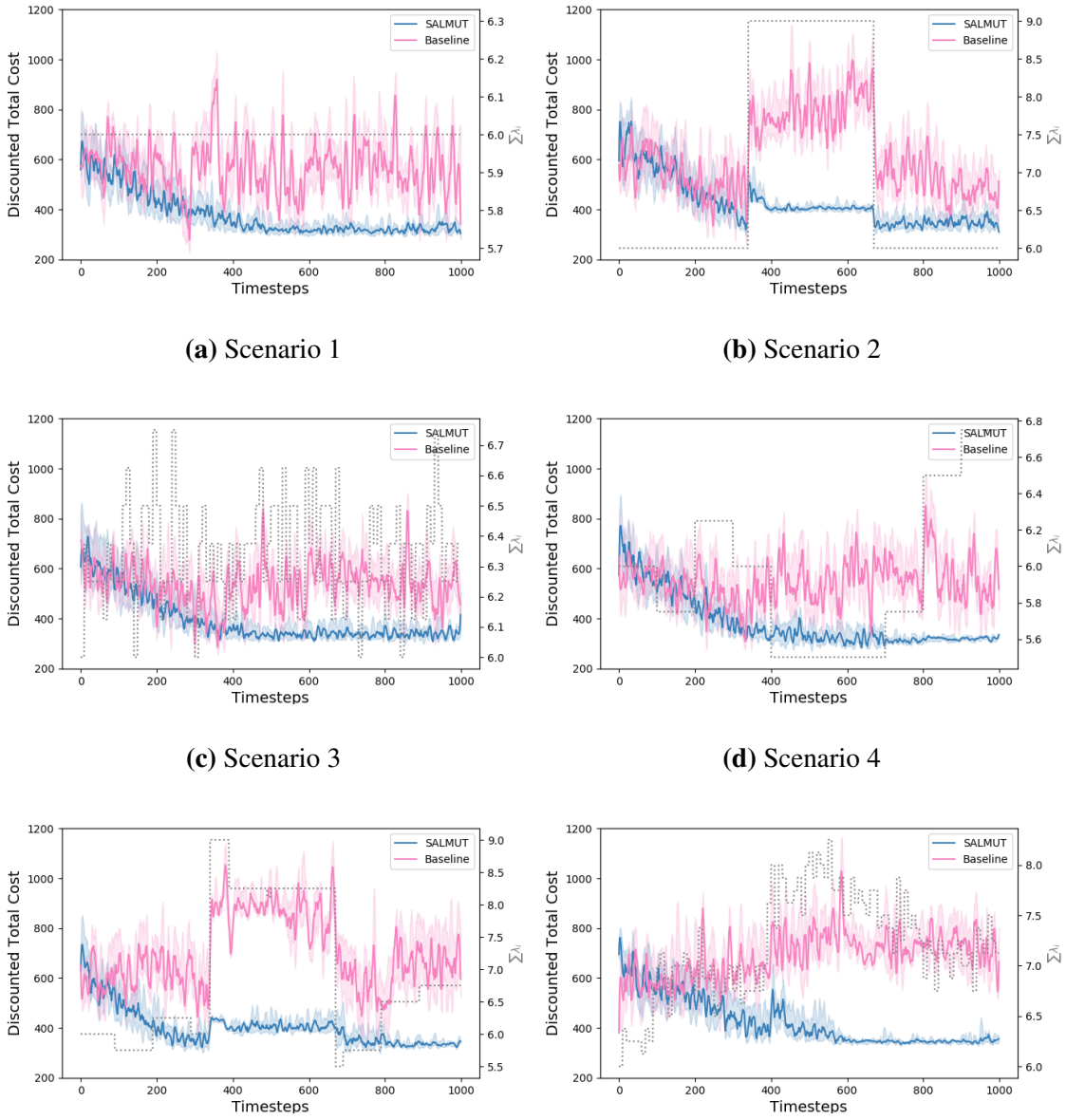
We run the experiments for SALMUT and the baseline algorithm for a total of 1000 steps. We do not run the simulations for the PPO and A2C algorithms in our testbed, as these algorithms cannot be trained in real-time in an online manner, due to the amount of time

they require for training. The performance of SALMUT and of the baseline algorithm is evaluated at every step, by computing the discounted total cost for that step using the cost buffers which are stored in the database. The experiment is repeated 5 times and the median performance with an uncertainty band from the first to the third quartile are plotted in Fig. 4.14 along with the total request arrival rate ($\sum_i \lambda_i$) in gray dotted lines.

For Scenario 1 (Fig. 4.14a), we observe that the SALMUT algorithm outperforms the baseline algorithm right from the start, indicating that SALMUT updates its policy very quickly at the start and slowly converges towards optimal performance after 400 steps, whereas the baseline incurs high cost throughout. Since SALMUT policies converge towards optimal performance after some time, they are also able to adapt quickly in Scenarios 2–6 (Fig. 4.14b-4.14f) and keep track of the time-varying arrival rates and number of users. We observe that SALMUT takes some time to learn a good policy, but once it learns the policy, it adjusts to frequent but small changes in λ and N very well (see Fig. 4.14c and 4.14d). If the request rate changes drastically, the performance decreases a little (which is bound to happen as the total requests to process are much more numerous than the server's capacity) but the magnitude of the performance drop is much smaller in SALMUT compared to the baseline, seen in Fig. 4.14b, 4.14e and 4.14f. This is because the baseline algorithms incur high overloading cost for these requests, whereas SALMUT incurs offloading costs for the same requests. Further analysis on this issue is presented in Section 4.2.2.

4.2.2 Behavioral Analysis

We perform behavior analysis of the learned policy by observing the number of times the system enters into an overloaded state (Fig. 4.15) and the number of incoming request



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

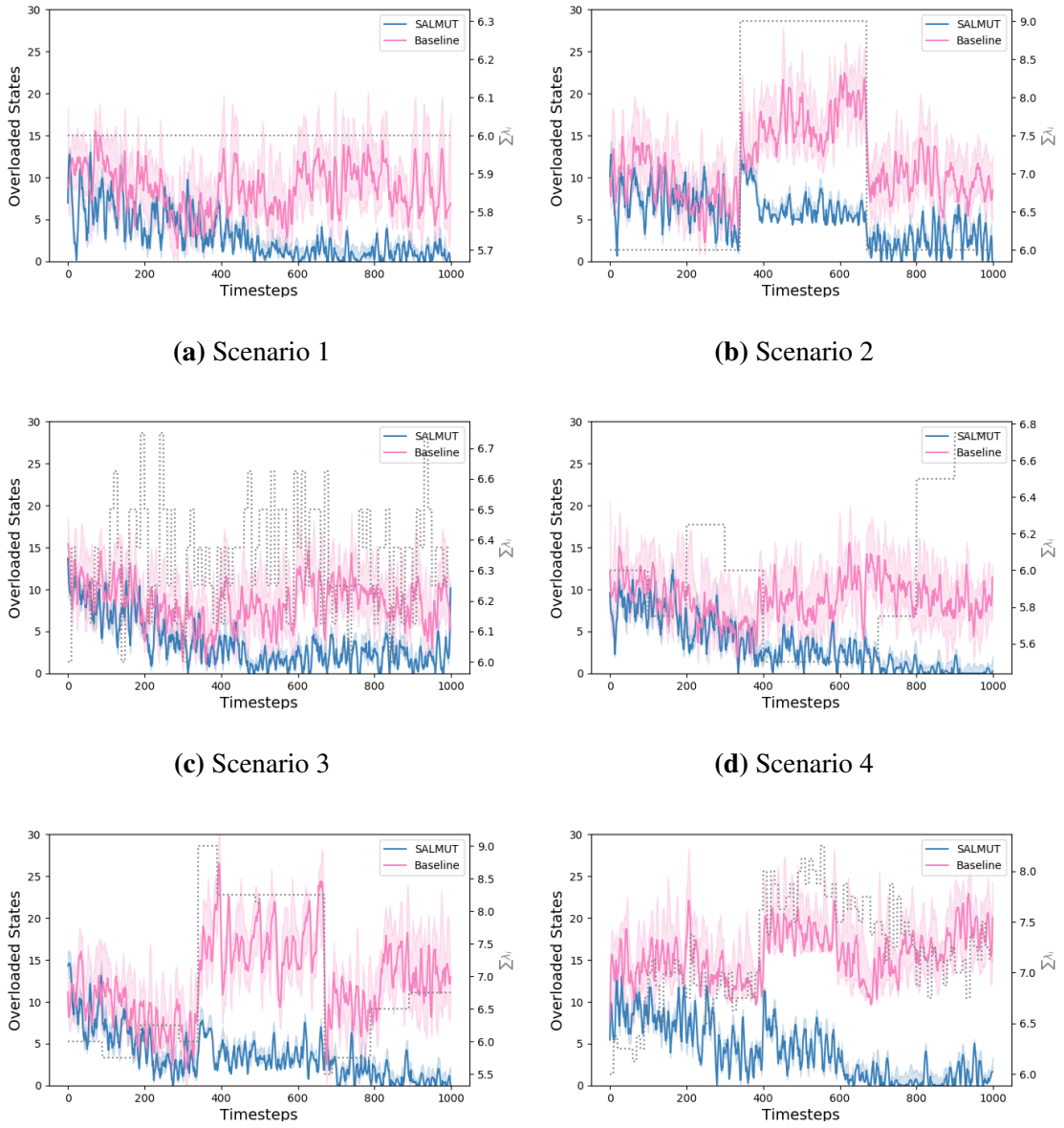
(e) Scenario 5

(f) Scenario 6

Figure 4.14: Performance of RL algorithms for different scenarios in the end-to-end testbed we created. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.

offloaded by the requests (Fig. 4.16) at every step. Let us define C_{ov} to be the number of times the system enters into an overloaded state and C_{off} to be the number of times the system offloads requests for every step (100s) of the training iteration.

Fig. 4.15 shows that the number of times the edge node goes into an overload state while



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

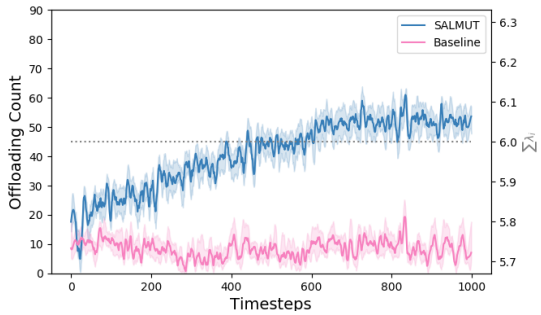
(d) Scenario 4

(e) Scenario 5

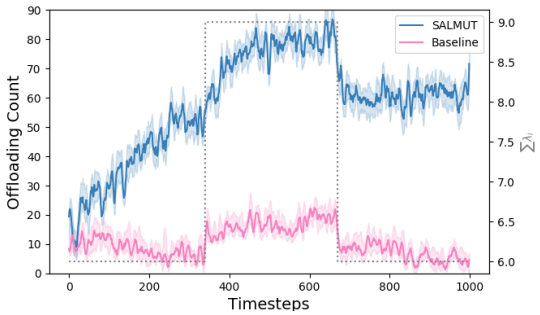
(f) Scenario 6

Figure 4.15: Comparing the number of times the system goes into the overloaded state at each step in the end-to-end testbed we created. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.

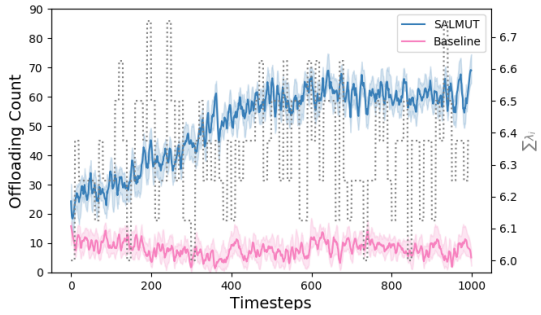
following policy executed by SALMUT is much smaller than for the baseline algorithm. Even when the system goes into an overloaded state, it is able to recover quickly and does not suffer from performance deterioration. From Fig. 4.15b and 4.15e we can observe that in Scenarios 2 and 5, when the request load increases drastically (at around 340 steps), C_{ov}



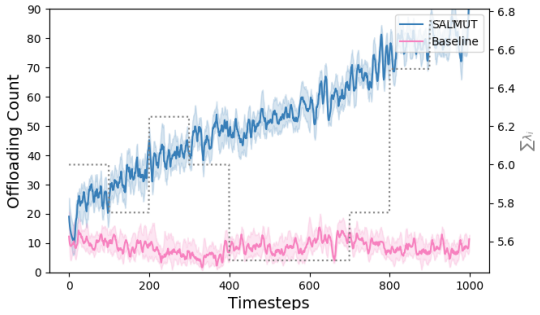
(a) Scenario 1



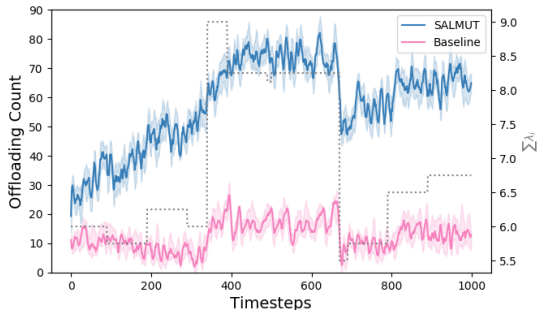
(b) Scenario 2



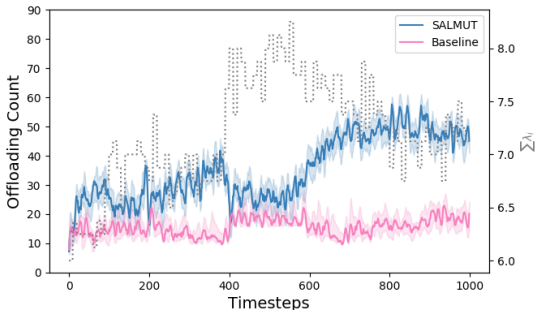
(c) Scenario 3



(d) Scenario 4



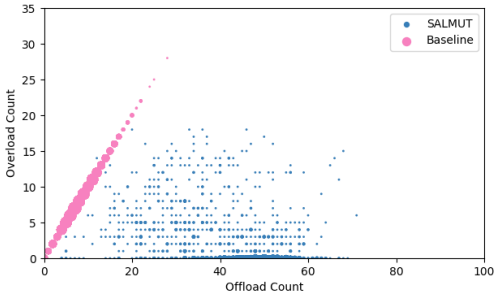
(e) Scenario 5



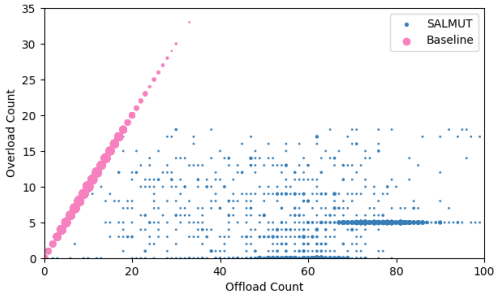
(f) Scenario 6

Figure 4.16: Comparing the number of times the system performs offloading at each step in the end-to-end testbed we created. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.

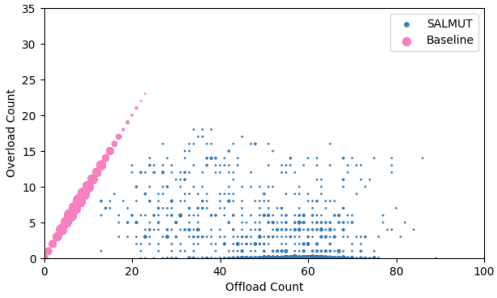
increases and its effects can also be seen in the overall discounted cost in Fig. 4.14b and 4.14e at around the same time. SALMUT is able to adapt its policy quickly and recovers within 50 steps. We observe in Fig. 4.16 that SALMUT performs more offloading as compared to the baseline algorithm.



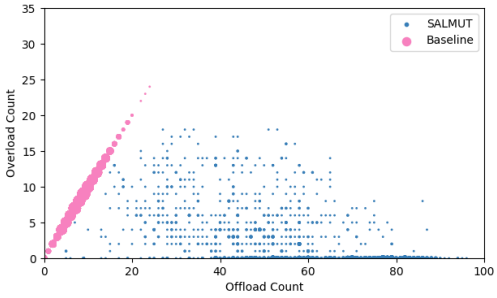
(a) Scenario 1



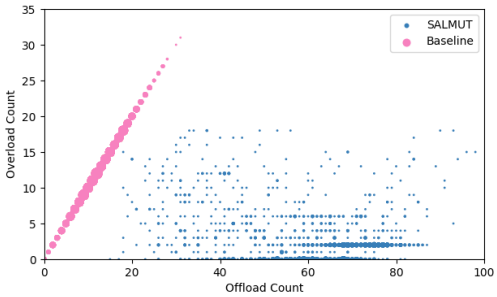
(b) Scenario 2



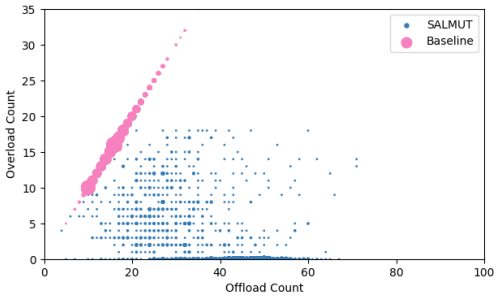
(c) Scenario 3



(d) Scenario 4



(e) Scenario 5



(f) Scenario 6

Figure 4.17: Scatter-plot of C_{ov} Vs C_{off} for SALMUT and baseline algorithm in the end-to-end testbed we created. The width of the points is proportional to its frequency.

A policy that offloads often and does not go into an overloaded state may not necessarily minimize the total cost. We did some further investigation by visualizing the scatter-plot (Fig. 4.17) of the overload count (C_{ov}) on the y-axis and the offload count (C_{off}) on the x-axis for both SALMUT and the baseline algorithm for all the scenarios described in

Fig. 4.8. We observe that SALMUT keeps C_{ov} much lower than the baseline algorithm at the cost of increased C_{off}. We can observe from Fig. 4.17 that the slope for the plot is linear for the baseline algorithms because they are offloading reactively. SALMUT, on the other hand, learns a behavior that is analogous to pro-active offloading, where it benefits from the offloading action it takes by minimizing C_{ov}.

Chapter 5

Conclusion and Future Work

In this thesis, we considered a single node optimal policy for overload protection on edge servers in a time varying environment. We proposed a RL-based, adaptive, low-complexity admission control approach that exploits the structure of the optimal policy and finds a policy that is easy to interpret. Our proposed algorithm performs as well as standard deep RL algorithms but has better computational and memory complexity. Therefore, our proposed algorithm is more suitable for deployment in real systems for online training.

The results we presented can be extended in several directions. In addition to CPU overload, one could consider other resource bottlenecks such as disk I/O, RAM utilization, etc. In such cases, the state dimension would increase and one could use state-dimensionality reduction techniques such as Principal Component Analysis (PCA) [77] and learn the threshold-mapping between the principal component of PCA and the corresponding load. It may be desirable to simultaneously consider multiple resource constraints. Similarly, one could consider multiple applications with different resource requirements and different priority. The framework we developed will be applicable to these more sophisticated setups as well, provided the optimal policy has some kind of a threshold structure.

The empirical evaluations were done for a small state space. The increase in the

state space can be handled by increasing the quantization step of discretization. If the quantization step is too coarse, border effects may arise: depending on the choice of the quantization intervals, the states may not capture the behavior of the system and small transitions close to the border between two states may be perceived as huge changes by the learner, which only knows that the state of the system has changed. There are several approaches presented in the literature to overcome the problem including clustering states, using softer borders [78], and generalizations to states based on fuzzy rules [79]. Soft state borders mean that whenever the learner is close to the border between two states it can use a linear combination of both states' Q-values, avoiding hard transitions and the associated border effects.

The discussion in this thesis was restricted to a single node. These results could also provide a foundation to investigate node overload protection in multi-node clusters where there are additional challenges such as routing, link failures, and changing network topology. One approach could be to develop a two-tier hierarchical reinforcement learning algorithm where the node-level admission control policy decides whether or not to offload a request and the cluster-level admission control policy aggregates information from all the nodes and executes the routing policy, i.e. where to offload the requests.

The problem could be modeled as designing optimal offloading and routing policies using multi-agent multi-stage optimization, where one can view each server as a separate agent. Each server could have a local state (its queue length and CPU load) and take a local action (whether or not to offload a new request). The state dynamics of the servers in this case are coupled through their control actions. In particular, when a request is offloaded by a server, it impacts the arrival rate at the server to which it is re-routed. The servers could also have coupled costs which depend on the end-to-end throughput and delay incurred

by the users. In particular, when a request is offloaded by a server, it impacts the arrival rate at the server to which it is re-routed. The servers also have a coupled reward which depends on the end-to-end throughput and delay incurred by the users. Both the dynamics and the reward couplings depend on the network characteristics such as link availability, delays, and capacity, which vary over time.

Such multi-agent multi-stage optimization problems are studied under the heading of decentralized stochastic control [80] or decentralized partially observable Markov decision problems (dec-POMDPs) [81]. However, in general, finding the optimal solution to such problems is NEXP-complete [82]. Therefore, instead of focusing on optimal policies, one can consider an alternative approach to learn a satisficing policies (i.e., policies which provide satisfactory or adequate performance rather than optimal performance) using multi-agent reinforcement learning (MARL) algorithms (see [83] and references therein for an overview), which has achieved considerable progress in recent years.

One of the difficulties in generalizing the approach discussed in this work to multi-agent systems is the difficulty in identifying the structure of optimal policies. However, since in the MARL framework, the objective is to identify a satisficing rather than optimal policy, one can restrict attention to a specific policy structure because it is easier to interpret even if it is not optimal. One could restrict attention to threshold-based policies and derive the policy gradients for such policies for the MARL algorithms.

Appendix A

Proof of Structural Properties

A.1 Proof of Proposition 1

Let $\delta(x) = \lambda/(\lambda + \min(k, x)\mu)$. We define a sequence of value functions $\{V_n\}_{n\geq 0}$ as follows

$$V_0(x, \ell) = 0$$

and for $n \geq 0$

$$V_{n+1}(x, \ell) = \min\{Q_{n+1}(x, \ell, 0), Q_{n+1}(x, \ell, 1)\},$$

where

$$\begin{aligned}
Q_{n+1}(x,\ell, 0) = \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell)\big] \\
+ \beta\Bigg[\delta(x)\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}}, [\ell+r]_{\mathsf{L}}) \\
+ (1-\delta(x))\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell-r]^+)\Bigg] and \\
Q_{n+1}(x,\ell, 1) = \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell) + p(\ell)\big] \\
+ \beta(1-\delta(x))\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell-r]^+),
\end{aligned}$$

where $[x]_{\mathsf{B}}$ denotes $\min\{x, \mathsf{B}\}$.

Note that $\{V_n\}_{n\geq 0}$ denotes the iterates of the value iteration algorithm, and from [10], we know that

$$\lim_{n\to\infty} V_n(x,\ell) = V(x,\ell), \quad \forall x,\ell \tag{A.1}$$

where V is the unique fixed point of (3.6).

We will show that (see Lemma 1 below) each $V_n(x,\ell)$ satisfies the property of Proposition 1. Therefore, by (A.1) we get that V also satisfies the property.

Lemma 1 *For each $n \geq 0$ and $x \in \{0,...,X\}$, $V_n(x,\ell)$ is weakly increasing in ℓ.* □

PROOF We prove the result by induction. Note that $V_0(x,\ell) = 0$ and is trivially weakly increasing in ℓ. This forms the basis of the induction. Now assume that $V_n(x,\ell)$ is weakly increasing in ℓ. Consider iteration $n+1$. Let $x \in \{0,...,X\}$ and $\ell_1, \ell_2 \in \{0,...,L\}$ such that $\ell_1 < \ell_2$. Then,

$$\begin{aligned}
Q_{n+1}(x,\ell_1,0) &= \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell_1)\big] \\
&\quad + \beta\Bigg[\delta(x)\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}},[\ell_1+r]_{\mathsf{L}}) \\
&\quad + (1-\delta(x))\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+,[\ell_1-r]^+)\Bigg] \\
&\overset{(a)}{\leq} \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell_2)\big] \\
&\quad + \beta\Bigg[\delta(x)\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}},[\ell_2+r]_{\mathsf{L}}) \\
&\quad + (1-\delta(x))\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+,[\ell_2-r]^+)\Bigg] \\
&= Q_{n+1}(x,\ell_2,0),
\end{aligned} \tag{A.2}$$

where (a) follows from the fact that $c(\ell)$ and $V_n(x,\ell)$ are weakly increasing in ℓ.

By a similar argument, we can show that

$$Q_{n+1}(x,\ell_1,1) \leq Q_{n+1}(x,\ell_2,1). \tag{A.3}$$

Now,

$$\begin{aligned} V_{n+1}(x,\ell_1) &= \min\{Q_{n+1}(x,\ell_1,0), Q_{n+1}(x,\ell_1,1)\} \\ &\overset{(b)}{\leq} \min\{Q_{n+1}(x,\ell_2,0), Q_{n+1}(x,\ell_2,1)\} \\ &= V_{n+1}(x,\ell_2) \end{aligned} \tag{A.4}$$

■

where (b) follows from (A.2) and (A.3). Eq. (A.4) shows that $V_{n+1}(x,\ell)$ is weakly increasing in ℓ. This proves the induction step. Hence, the result holds for the induction.

A.2 Proof of Proposition 2

PROOF Let $\delta(x) = \lambda/(\lambda + \min\{x,k\}\mu)$. Consider

$$\begin{aligned} \Delta Q(x,\ell) =& Q(x,\ell,1) - Q(x,\ell,0) \\ &= -\beta\delta(x)\sum_{r=1}^{\mathsf{R}} P(r)V([x]_{\mathsf{X}},[\ell+r]_{\mathsf{L}}) - p. \end{aligned}$$

For a fixed x, by Proposition 1, $\Delta Q(x,\ell)$ is weakly decreasing in ℓ. If it is optimal to reject a request at state (x,ℓ) (i.e., $\Delta Q(x,\ell) \leq 0$), then for any $\ell' > \ell$,

$$\Delta Q(x,\ell') \leq \Delta Q(x,\ell) \leq 0;$$

■

therefore, it is optimal to reject the request.

Appendix B

Proof of Optimality of SALMUT

PROOF The choice of learning rates implies that there is a separation of timescales between the updates of (3.13) and (3.14). In particular, since $b_n^2/b_n^1 \to 0$, iteration (3.13) evolves at a faster timescale than iteration (3.14). Therefore, we first consider update (3.14) under the assumption that the policy π_τ, which updates at the slower timescale, is constant.

We first provide a preliminary result.

Lemma 2 *Let Q_τ denote the action-value function corresponding to the policy π_τ. Then, Q_τ is Lipscitz continuous in τ.*

PROOF This follows immediately from the Lipscitz continuity of π_τ in τ. ∎

Define the operator $\mathcal{M}_\tau : \mathbb{R}^{\mathsf{N}} \to \mathbb{R}^{\mathsf{N}}$, where $\mathsf{N} = (\mathsf{X}+1) \times (\mathsf{L}+1) \times \mathcal{A}$, as follows:

$$[\mathcal{M}_\tau Q](x,\ell,a) \;=\; \big[\bar{\rho}(x,\ell,a) + \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \min_{a' \in \mathcal{A}} Q(x',\ell',a')\big] - Q(x,\ell,a). \tag{B.1}$$

Then, the step-size conditions on $\{b_n^1\}_{n\geq 1}$ imply that for a fixed π_τ, iteration (3.13) may be viewed as a noisy discretization of the ODE (ordinary differential equation):

$$\dot{Q}(t) = \mathcal{M}_\tau[Q(t)]. \tag{B.2}$$

Then we have the following:

Lemma 3 *The ODE* (B.2) *has a unique globally asymptotically stable equilibrium point* Q_τ.

PROOF Note that the ODE (B.2) may be written as

$$\dot{Q}(t) = \mathcal{B}_\tau[Q(t)] - Q(t)$$

where the Bellman operator $\mathcal{B}_\tau : \mathbb{R}^{\mathsf{N}} \to \mathbb{R}^{\mathsf{N}}$ is given by

$$\mathcal{B}_\tau[Q](x,\ell) = \big[\bar{\rho}(x,\ell,a) + \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \times \min_{a' \in \mathcal{A}} Q(x',\ell',a')\big]. \tag{B.3}$$

Note that $\mathcal{B}_\tau$ is a contraction under the sup-norm. Therefore, by Banach fixed point theorem, $Q = \mathcal{B}_\tau Q$ has a unique fixed point, which is equal to Q_τ. The result then follows from [84, Theorem 3.1]. ∎

We now consider the faster timescale. Recall that (x_0, ℓ_0) is the initial state of the MDP. Recall

$$J(\tau) = V_\tau(x_0, \ell_0)$$

and consider the ODE limit of the slower timescale iteration (3.14), which is given by

$$\dot{\tau} = -\nabla J(\tau). \tag{B.4}$$

Lemma 4 *The equilibrium points of the ODE* (B.4) *are the same as the local optima of* $J(\tau)$. *Moreover, these equilibrium points are locally asymptotically stable.*

PROOF The equivalence between the stationary points of the ODE and local optima of $J(\tau)$ follows from definition. Now consider $J(\tau(t))$ as a Lyapunov function. Observe that

$$\frac{d}{dt} J(\tau(t)) = -\big[\nabla J(\tau(t))\big]^2 < 0,$$

as long as $\nabla J(\tau(t)) \neq 0$. Thus, from Lyapunov stability criteria all local optima of (B.4) are locally asymptotically stable. ∎

Now, we have all the ingredients to prove convergence. Lemmas 2-4 imply assumptions (A1) and (A2) of [85]. Thus, the iteration (3.13) and (3.14) converges almost surely to a limit point (Q°, τ°) such that $Q^\circ = Q_{\tau^\circ}$ and $\nabla J(\tau^\circ) = 0$ provided that the iterates $\{Q_n\}_{n\geq 1}$ and $\{\tau_n\}_{n\geq 1}$ are bounded.

Note that $\{\tau_n\}_{n\geq 1}$ are bounded by construction. The boundness of $\{Q_n\}_{n\geq 1}$ follows from considering the scaled version of (B.2):

$$\dot{Q} = \mathcal{M}_{\tau,\infty} Q \tag{B.5}$$

where,

$$\mathcal{M}_{\tau,\infty} Q = \lim_{c\to\infty} \frac{\mathcal{M}_\tau[cQ]}{c}.$$

It is easy to see that

$$[\mathcal{M}_{\tau,\infty} Q](x,\ell,a) = \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \min_{a'\in\mathcal{A}} Q(x',\ell',a') - Q(x,\ell,a) \tag{B.6}$$

■

Furthermore, origin is the asymptotically stable equilibrium point of (B.5). Thus, from [86], we get that the iterates $\{Q_n\}_{n\geq 1}$ of (3.13) are bounded.

Bibliography

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[3] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *IEEE International Symposium on Information Theory*, 2016, pp. 1451–1455.

[4] F. Wang, J. Xu, X. Wang, and S. Cui, "Joint offloading and computing optimization in wireless powered mobile-edge computing systems," *IEEE Transactions on Wireless Communications*, vol. 17, no. 3, pp. 1784–1797, 2017.

[5] D. Van Le and C.-K. Tham, "Quality of service aware computation offloading in an ad-hoc mobile cloud," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 9, pp. 8890–8904, 2018.

[6] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Transactions on Wireless Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[7] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, Jun. 2019.

[8] A. Jensen, "Markoff chains as an aid in the study of markoff processes," *Scandinavian Actuarial Journal*, vol. 1953, no. sup1, pp. 87–91, 1953.

[9] R. A. Howard, *Dynamic Programming and Markov Processes*. The MIT Press, 1960.

[10] M. Puterman, *Markov decision processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. The MIT Press, 2018.

[12] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2018.

[13] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks," *arXiv:1808.01977*, 2018.

[14] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 64–69, 2019.

[15] A. Roy, V. Borkar, A. Karandikar, and P. Chaporkar, "Online reinforcement learning of optimal threshold policies for Markov decision processes," *arXiv:1912.10325*, 2019.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[17] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," *arXiv:1708.05144*, 2017.

[18] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[19] R. Bellman, "On the theory of dynamic programming," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 38, no. 8, p. 716, 1952.

[20] M. L. Puterman and M. C. Shin, "Modified policy iteration algorithms for discounted markov decision problems," *Management Science*, vol. 24, no. 11, pp. 1127–1137, 1978.

[21] J Van Der Wal, "Discounted markov games: Generalized policy iteration method," *Journal of Optimization Theory and Applications*, vol. 25, no. 1, pp. 125–138, 1978.

[22] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge United Kingdom, 1989.

[23] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[24] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[25] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[26] J. N. Tsitsiklis, "Asynchronous stochastic approximation and q-learning," *Machine learning*, vol. 16, no. 3, pp. 185–202, 1994.

[27] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

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

[29] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[30] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *AAAI Fall Symposium Series*, 2015.

[31] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, 2000, pp. 1057–1063.

[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[34] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, 2015, pp. 1889–1897.

[35] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[36] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *IEEE Wireless Communications and Networking Conference*, IEEE, 2018, pp. 1–6.

[37] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2018.

[38] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for IoT devices with energy harvesting," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1930–1941, 2019.

[39] D. Van Le and C.-K. Tham, "A deep reinforcement learning based offloading scheme in ad-hoc mobile clouds," in *IEEE Conference on Computer Communications Workshops*, IEEE, 2018, pp. 760–765.

[40] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[41] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[42] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, "Deep reinforcement learning for dynamic multichannel access in wireless networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 2, pp. 257–265, 2018.

[43] H. Ye, G. Y. Li, and B.-H. F. Juang, "Deep reinforcement learning based resource allocation for V2V communications," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 3163–3173, 2019.

[44] N. Zhao, Y.-C. Liang, D. Niyato, Y. Pei, and Y. Jiang, "Deep reinforcement learning for user association and resource allocation in heterogeneous networks," in *IEEE Global Communications Conference*, IEEE, 2018, pp. 1–6.

[45] T. Stockhammer, "Dynamic adaptive streaming over HTTP– standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, 2011, pp. 133–144.

[46] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 197–210.

[47] P. V. R. Ferreira, R. Paffenroth, A. M. Wyglinski, T. M. Hackett, S. G. Bilén, R. C. Reinhart, and D. J. Mortensen, "Multiobjective reinforcement learning for cognitive satellite communications using deep neural network ensembles," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 5, pp. 1030–1041, 2018.

[48] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[49] C. Li, L. Toni, J. Zou, H. Xiong, and P. Frossard, "QoE-driven mobile edge caching placement for adaptive video streaming," *IEEE Transactions on Multimedia*, vol. 20, no. 4, pp. 965–984, 2017.

[50] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *IEEE International Conference on Computer Communications*, IEEE, 2010, pp. 1–9.

[51] S. Zhang, P. He, K. Suto, P. Yang, L. Zhao, and X. Shen, "Cooperative edge caching in user-centric clustered mobile networks," *IEEE Transactions on Mobile Computing*, vol. 17, no. 8, pp. 1791–1805, 2017.

[52] S. Ioannidis and E. Yeh, "Adaptive caching networks with optimality guarantees," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 113–124, 2016.

[53] C. Liang, F. R. Yu, N. Dao, G. Senarath, and H. Farmanbar, "Enabling adaptive data prefetching in 5G mobile networks with edge caching," in *IEEE Global Communications Conference*, IEEE, 2018, pp. 1–6.

[54] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[55] S. Wang, X. Zhang, K. Yang, L. Wang, and W. Wang, "Distributed edge caching scheme considering the tradeoff between the diversity and redundancy of cached

content," in *IEEE International Conference on Communications in China*, IEEE, 2015, pp. 1–5.

[56] J. Liu, B. Bai, J. Zhang, and K. B. Letaief, "Content caching at the wireless network edge: A distributed algorithm via belief propagation," in *IEEE International Conference on Communications*, IEEE, 2016, pp. 1–6.

[57] C. Fang, F. R. Yu, T. Huang, J. Liu, and Y. Liu, "Energy-efficient distributed in-network caching for content-centric networks," in *IEEE Conference on Computer Communications Workshops*, IEEE, 2014, pp. 91–96.

[58] J. Gao, L. Zhao, and L. Sun, "Probabilistic caching as mixed strategies in spatially-coupled edge caching," in *29th Biennial Symposium on Communications (BSC)*, IEEE, 2018, pp. 1–5.

[59] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 5, pp. 5–12, 2013.

[60] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *52nd Annual Conference on Information Sciences and Systems*, IEEE, 2018, pp. 1–6.

[61] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," *arXiv preprint arXiv:1512.07679*, 2015.

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

[63] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, "Optimal and scalable caching for 5G using reinforcement learning of space-time popularities," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 180–190, 2017.

[64] Y. Chen, Y. Li, D. Xu, and L. Xiao, "DQN-based power control for IoT transmission against jamming," in *IEEE Vehicular Technology Conference*, IEEE, 2018, pp. 1–5.

[65] A. Ferdowsi and W. Saad, "Deep learning-based dynamic watermarking for secure signal authentication in the internet of things," in *IEEE International Conference on Communications*, IEEE, 2018, pp. 1–6.

[66] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

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

[68] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE Conference on Computer Communications Workshops*, IEEE, 2018, pp. 1871–1879.

[69] C. Wang, J. Wang, X. Zhang, and X. Zhang, "Autonomous navigation of UAV in large-scale unknown complex environment with deep reinforcement learning," in *IEEE Global Conference on Signal and Information Processing*, IEEE, 2017, pp. 858–862.

[70] D. R. Song, C. Yang, C. McGreavy, and Z. Li, "Recurrent deterministic policy gradient method for bipedal locomotion on rough terrain challenge," in *IEEE In-*

ternational Conference on Control, Automation, Robotics and Vision, IEEE, 2018, pp. 311–318.

[71] Z. Xu, Y. Wang, J. Tang, J. Wang, and M. C. Gursoy, "A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs," in *IEEE International Conference on Communications*, IEEE, 2017, pp. 1–6.

[72] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5G: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[73] X. Chen, Z. Li, Y. Zhang, R. Long, H. Yu, X. Du, and M. Guizani, "Reinforcement learning–based qos/qoe-aware service function chaining in software-driven 5g slices," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, e3477, 2018.

[74] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016, pp. 50–56.

[75] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 705–718, 2018.

[76] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[77] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[78] S. P. S. T. J. Michael and I Jordan, "Reinforcement learning with soft state aggregation," *Advances in Neural Information Processing Systems*, vol. 7, p. 361, 1995.

[79] P. Y. Glorennec and L. Jouffe, "Fuzzy q-learning," in *Proceedings of 6th International Fuzzy Systems Conference*, IEEE, vol. 2, 1997, pp. 659–662.

[80] A. Nayyar, A. Mahajan, and D. Teneketzis, "Decentralized stochastic control with partial history sharing: A common information approach," *IEEE Transactions on Automatic Control*, vol. 58, no. 7, pp. 1644–1658, 2013.

[81] F. A. Oliehoek and C. Amato, *A concise introduction to decentralized POMDPs*. Springer, 2016.

[82] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The complexity of decentralized control of markov decision processes," *Mathematics of operations research*, vol. 27, no. 4, pp. 819–840, 2002.

[83] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *arXiv preprint arXiv:1911.10635*, 2019.

[84] V. S. Borkar and K Soumyanatha, "An analog scheme for fixed point computation – Part I: Theory," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 44, no. 4, pp. 351–355, 1997.

[85] V. S. Borkar, "Stochastic approximation with two time scales," *Systems & Control Letters*, vol. 29, no. 5, pp. 291–294, 1997.

[86] V. S. Borkar and S. P. Meyn, "The ODE method for convergence of stochastic approximation and reinforcement learning," *SIAM Journal on Control and Optimization*, vol. 38, no. 2, pp. 447–469, 2000.