Spring 5-15-2022

Learning-based Decision Making in Wireless Communications

Chen Zhong
Syracuse University

Abstract

Fueled by emerging applications and exponential increase in data traffic, wireless networks have recently grown significantly and become more complex. In such large-scale complex wireless networks, it is challenging and, oftentimes, infeasible for conventional optimization methods to quickly solve critical decision-making problems. With this motivation, in this thesis, machine learning methods are developed and utilized for obtaining optimal/near-optimal solutions for timely decision making in wireless networks.

Content caching at the edge nodes is a promising technique to reduce the data traffic in next-generation wireless networks. In this context, we in the first part of the thesis study content caching at the wireless network edge using a deep reinforcement learning framework with Wolpertinger architecture. Initially, we develop a learning-based caching policy for a single base station aiming at maximizing the long-term cache hit rate. Then, we extend this study to a wireless communication network with multiple edge nodes. In particular, we propose deep actor-critic reinforcement learning based policies for both centralized and decentralized content caching.

Next, with the purpose of making efficient use of limited spectral resources, we develop a deep actor-critic reinforcement learning based framework for dynamic multichannel access. We consider both a single-user case and a scenario in which multiple users attempt to access channels simultaneously. In the single-user model, in order to evaluate the performance of the proposed channel access policy and the framework's tolerance against uncertainty, we explore different channel switching patterns and different switching probabilities. In the case of multiple users, we analyze the

probabilities of each user accessing channels with favorable channel conditions and the probability of collision.

Following the analysis of the proposed learning-based dynamic multichannel access policy, we consider adversarial attacks on it. In particular, we propose two adversarial policies, one based on feed-forward neural networks and the other based on deep reinforcement learning policies. Both attack strategies aim at minimizing the accuracy of a deep reinforcement learning based dynamic channel access agent, and we demonstrate and compare their performances.

Next, anomaly detection as an active hypothesis test problem is studied. Specifically, we study deep reinforcement learning based active sequential testing for anomaly detection. We assume that there is an unknown number of abnormal processes at a time and the agent can only check with one sensor in each sampling step. To maximize the confidence level of the decision and minimize the stopping time concurrently, we propose a deep actor-critic reinforcement learning framework that can dynamically select the sensor based on the posterior probabilities. Separately, we also regard the detection of threshold crossing as an anomaly detection problem, and analyze it via hierarchical generative adversarial networks (GANs).

In the final part of the thesis, to address state estimation and detection problems in the presence of noisy sensor observations and probing costs, we develop a soft actor-critic deep reinforcement learning framework. Moreover, considering Byzantine attacks, we design a GAN-based framework to identify the Byzantine sensors. To evaluate the proposed framework, we measure the performance in terms of detection accuracy, stopping time, and the total probing cost needed for detection.

LEARNING-BASED DECISION MAKING IN WIRELESS COMMUNICATIONS

By

Chen Zhong

M.S., Stevens Institute of Technology, 2016

B.E., Beijing Institute of Technology, 2014

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University

May 2022

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisors, Dr. Mustafa Cenk Gursoy and Dr. Senem Velipasalar Gursoy, for their patient guidance and constant support throughout my doctoral study. Their enthusiasm and rigor for academics inspired me to overcome one challenge after another in my research. Without their invaluable advice and encouragement, this dissertation would not have been possible.

I would like to extend my deepest gratitude to all my defense committee members Dr. Carlos E. Caicedo Bastidas, Dr. Pramod K. Varshney, Dr. Biao Chen, Dr. Qinru Qiu, and Dr. Sara Eftekharnejad for their valuable comments and suggestions.

I would also like to take this opportunity to thank all my former and current labmates of the Wireless Comm. & Networking Lab and the Smart Vision Systems Lab for their support in my research and life. My appreciation also extends to all my collaborators. It is the intense academic discussion with you that makes me enjoy the process of research and understand the motto of Syracuse University, "Suos Cultores Scientia Corona".

Special thanks to my friends who keep me in a company at Syracuse, and who always support me emotionally in the COVID-19 pandemic.

Most importantly, I would like to express my sincerest gratitude to my family for their unconditional love and support. Thanks to the motivation and inspiration from them since my childhood, and it will fuel me for the rest of my life. Thanks to their understanding and care, I could complete my study with great peace of mind. I dedicate this dissertation to my family.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Next generation wireless networks hold the promise for the development of smart systems, intelligent devices, and new-media. However, the emergence of various novel applications, which increases the demand for reliable, low-latency, and high-data-rate wireless communications, brings challenges to the design of wireless networks.

Specifically, there has been a significant growth in wireless data, with regards to both the total traffic volume and the average size per file. This has led to the necessity to have the scarce computational, storage, and spectral resources be more effectively allocated and efficiently utilized. As a consequence, several novel supporting techniques, such as edge computing, content caching, multichannel access, and non-orthogonal multiple access (NOMA), have been developed and incorporated into wireless networks. In such schemes, decision making plays a important role.

Employing conventional optimization methods to solve decision making problems is challenging and not scalable in large, dynamic, complex wireless networks. Being competitive in effectively controlling the computational complexity and providing optimal/near-optimal policies, machine learning methods are considered as promising

tools to address these problems.

In the literature, how machine learning methods can be applied in wireless communication problems have recently been intensively studied. For instance, in [1] and [2], the authors summarized the applications of machine learning methods in wireless communication networks, including channel estimation, end-to-end communication system design, and resource allocation. In [3], deep learning-based solutions for physical-layer 5G wireless techniques are analyzed. Moreover, the authors in [4] discussed the details of selection between model-based techniques and AI-based techniques, and advocated the application of both techniques in combination. Also, they provided a detailed tutorial on employing artificial neural network-based machine learning algorithms for solving various wireless networking problems. In addition, future directions and open research problems in machine learning for wireless communications are summarized in [5] and [6].

While certain machine learning methods demonstrate outstanding performance in specific problems, there exist challenges and potential risks in applying machine learning techniques in wireless systems. On the one hand, training machine learning algorithms is demanding especially in dynamic and large wireless networks. To solve this problem, new training strategies and new machine learning algorithms are constantly being explored. For example, in [7], federated learning is used for each user to train a policy with the exploitation of limited local computational resources. In [8], conditional GANs are employed to model the channel effects. Capability of machine learning algorithms to handle big data is explored in [9] and [10]. On the other hand, application of machine learning in wireless communications also raises security considerations due to the potential detrimental impact of adversarial attacks. To investigate this aspect, the authors in [11] proposed a stochastic gradient descent (SGD) algorithm to learn a predictive model over blockchains and used an l-nearest aggregation algorithm to protect the system from Byzantine attacks. In [12], a comprehensive

survey is provided to introduce how deep learning algorithms can be employed for anomaly detection.

In this thesis, we initially develop machine learning algorithms for solving specific decision making problems in wireless communications, such as content caching and multichannel access. Subsequently, we analyze security aspects of learning-based systems, by studying the impact of adversarial attacks and investigating anomaly detection.

1.2 Literature Review

1.2.1 Content Caching

The rapid growth in the number of mobile devices and in rich media-enabled applications has led to a 17-fold increase in mobile data traffic from 2012 to 2017 [13]. Mobile video accounts for more than half of this data traffic, and is predicted to further grow by 9-fold, accounting for 79% of the total data traffic, by 2022. However, the increase in mobile network connection speed, which is predicted to be only about three-fold, will not be adequate to satisfy the users' demands on high-quality streaming services.

To better serve the users, content caching strategies have been studied recently. In particular, content caching is considered as a key approach to reduce the data traffic as it enables the content server nodes to store a part of the popular contents locally, so that when the cached contents are requested, the server can deliver the content directly to the users and reduce the delay and congestion in the network. The authors in [14] analyzed proactive content caching from the aspect of big data analytics and demonstrated that with limited cache size, proactive caching can provide 100% user satisfaction while offloading 98% of the backhaul traffic.

Motivated by this, different caching strategies have been studied in the literature. For central content servers, such as the baseband unit in a cloud radio access

network (C-RAN), centralized coded caching and delivery schemes were presented in [15] and [16]. Regarding decentralized caching, the authors in [17] presented a decentralized optimization method for the design of caching strategies that aimed at minimizing the energy consumption of the network. Recently, proactive caching at the wireless network edge, such as at the base stations and user equipments, is proposed. This technique makes it possible to have popular contents to be placed closer to the end users and be directly transmitted, which can effectively reduce the time compared to routing in content delivery networks (CDNs), and apparently save a considerable amount of waiting time for users and offload a portion of the data traffic at the CDN. For instance, authors in [18] and [19] studied edge caching policies aimed at minimizing the transmission delay in cellular networks. In [20], a decentralized framework for proactive caching was proposed based on blockchains considering a game-theoretic point of view. In [21], caching and multicast problems were jointly solved using dynamic programming. In addition, studies on hierarchical caching have also been conducted. For instance, in [22–24], hybrid content caching schemes for joint content caching control at the baseband unit and radio remote heads were presented. And the authors in [25] proposed an edge hierarchical caching policy for caching at small base stations and user equipments. In addition, other caching strategies have been intensively studied recently considering different models and strategies. For instance, the study in [26] proposed an age-based threshold policy which caches all contents that have been requested by more than a certain threshold. Furthermore, popularity-based content caching policies named StreamCache and PopCaching were studied in [27] and [28], respectively. Recently, femtocell caching [29], coded caching [30] and D2D caching [31] have also been investigated.

We in Chapter 2 concentrate on edge caching at small base stations, where the caching policy at the base station is driven by the content popularity. Hence, content popularity is the key to solve the caching problem. In previous works, content popu-

larity is assumed to be either known to the content server as presented in [20, 32], or estimated before the caching actions as proposed in [19, 33]. The former assumption makes the framework less practical when the content popularity is time-varying, and the estimation of the content popularity or the arrival intensity of the users' requests will lead to large overhead. To avoid these drawbacks, machine learning methods have recently been introduced to determine efficient caching policies. For instance, the authors in [34] trained the optimization algorithms for caching through a deep neural network in advance. In other studies, different deep reinforcement learning (DRL) algorithms were used to find the caching policies that can better adapt to changing environments. In [35], authors implemented a Q-learning algorithm to find the optimal caching policy. In [36] and [37], the use of actor-critic deep reinforcement learning frameworks for caching policies were studied. And for the cooperative caching policy in decentralized caching networks, a multi-agent Q-learning solution was proposed in [38], and authors in [39] and [40] presented two different multi-armed bandit based caching schemes. And the authors in [41] proposed an extreme-learning machine framework for content popularity prediction.

As seen in previous studies, content popularity distribution has a key role in content caching problems. Though the DRL algorithms do not require the content popularity information, the agent needs to observe enough features of the environment to ensure the accuracy of its decisions, so we adopt the Wolpertinger architecture based actor-critic DRL framework [42] to deal with large discrete action spaces. And for the decentralized caching system, we propose a multi-agent framework [43–45] for cooperative caching.

1.2.2 Multichannel Access

The dynamic spectrum access problem has been extensively studied in the literature. For instance, the authors in [46] provided a comprehensive survey on spectrum shar-

ing technologies in cognitive radio networks with an outlook towards 5G. In the fixed spectrum assignment policy, a large portion of the assigned spectrum may be used sporadically where another portion of the spectrum can be congested. Allowing users to dynamically choose the available channels, the dynamic spectrum access technology is considered crucial to ensure that the limited spectral resources are allocated appropriately to satisfy the users' demand. For the correlated channel scenarios, the authors in [47] developed an analytical framework for opportunistic spectrum access based on the theory of partially observable Markov decision processes (POMDPs). And for independent channels, the problem can be modeled as a restless multi-armed bandit (RMAB) process [48].

Numerous studies have been conducted to find the spectrum access policies enabling users to effectively probe the channels. For instance, myopic policies were studied in [49] and [50], where the information on channels is collected through sufficient statistics and the user only senses the channel with the highest conditional probability. A stochastic game theory based policy was presented in [51] and [52], where multiple users are in the system but each user can adjust its behavior based only on the individual information. In [53], a joint probing and accessing policy was proposed to allow the user to probe multiple channels at a time.

Inspired by the achievements of reinforcement learning in dynamic control problems, such as the game of Atari [54], and AlphaGo [55], there has been increased interest in seeking reinforcement learning based solutions for problems in wireless communications. As summarized in [56] and [57], deep reinforcement learning algorithms have been applied in various wireless settings. For example, the authors in [58] and [59] investigate the use of Q-learning and SARSA (state-action-reward-state-action) reinforcement learning, respectively, in power control. The base stations' ON-OFF states are controlled by a deep Q-network (DQN) with the goal to improve the energy efficiency in [60]. And authors of [61] introduced the allocation of compu-

tational resources, and proposed a semi-MDP based optimal policy to schedule the cloud computing resources with the purpose to improve the system utility. Moreover, reinforcement learning is also used to perform joint optimizations. For example, the DQN was applied to seek optimal policies to jointly allocate the sub-bands and power in vehicle-to-vehicle communication [62], and an actor-critic reinforcement learning framework was proposed to jointly solve the user scheduling, and subchannel and power allocation problem in order to maximize the energy efficiency [63].

As to the dynamic spectrum access, the control problem is generally modeled as either an MDP [64] [65] or a POMDP [66] depending on whether the environment is completely observable to the users or not. And, there are various different reinforcement learning algorithms being used in solving the spectrum access problems. Authors in [67] proposed a continuous sampling and exploitation (CSE) online learning algorithm for an RMAB model. An application of Q-learning in the sensing order selection, in the presence of imperfect sensing, is presented in [68]. Also, as a typical reinforcement learning framework, DQN has been applied in [66, 69, 70] for different purposes such as to improve the accuracy of selecting the channels in good condition, to maximize the network utility, or to minimize the service blocking probability. Additionally, in order to solve the dynamic spectrum access problem in decentralized systems, different multi-agent reinforcement learning strategies are studied in [69, 71, 72]. For instance, in [69], the authors concentrated on a multi-user scenario in which transmission is successful only if a single user transmits over an accessed channel. The channels themselves do not inherently have time-varying states and correlations, and only collisions lead to transmission failures. And in [73], a comparison between the single-agent reinforcement learning and multi-agent reinforcement learning is provided.

Motivated by the recent research described above, we in this thesis investigate the application of the actor-critic reinforcement learning based framework to multichannel

access, while in the literature more focus has been on deep Q-network and Q-learning frameworks. Besides, our work also considers the multichannel access problem from a different aspect and we address more challenging and general scenarios (e.g., with more channels and more states, agents operating in a decentralized fashion in multi-user scenarios), requiring us to design new learning algorithms. For instance, as also noted above, instead of the value-based deep Q-network algorithm, we use actor-critic framework that takes advantage of value-based and policy-based algorithmic strategies.

1.2.3 Adversarial Jamming Attacks

There has recently been growing interest in employing machine learning to address certain problems in communication systems, such as modulation classification [74], and dynamic multichannel access [75]. However, this increasing interest brings forth potential security risks due to adversarial attacks. Since machine learning methods are highly data-driven algorithms, even a minor modification in the observation data can lead to dramatic changes in the learning-based decision policies. Therefore, adversarial machine learning has been intensively studied to better understand the vulnerabilities of machine learning methods. Motivated by this, we in this thesis investigate the learning-based wireless jamming attacks on deep reinforcement learning policies on dynamic multichannel access.

In the literature, adversarial attacks have been considered and widely applied to deep learning-based classification problems, such as the classification of images [76], time series [77] and sound events [78]. In these cases, the victim models are trained and fixed, and the input data is accessible to the attacker, so that the attack can be realized by crafting adversarial examples to mislead the victim's decisions. This idea is also used in the attack on reinforcement learning-based tasks [79] and [80]. However, in certain control problems, the observations of the reinforcement learning agents are

not available to the attacker, making it infeasible to craft any adversarial examples. To tackle this difficulty, in [81], the authors trained a reinforcement learning-based adversarial policy instead.

A deep learning-based wireless jamming attack has been studied in [82] and [83], in both of which, the system consists of a single transmitter, a receiver, one background traffic source and a deep learning-based jammer. Inspired by the framework presented in the literature, we address a more general channel model introduced in [84], and we propose two different jamming attackers, namely a feed-forward neural network (FNN) attacker and a deep reinforcement learning (DRL) attacker, to perform the jamming attacks on a user performing dynamic multichannel access using a DRL agent itself [84].

1.2.4 Anomaly Detection

State estimation/detection is critical in different applications, involving, for instance, remote health monitoring [85], smart grid [86], assembly lines, structural health monitoring, autonomous systems [87] [88], adaptive radar [89], cognitive radio networks [90], and Internet of Things (IoT). And the authors in [91] provided a survey of anomaly detection techniques for wireless sensor networks. In such applications, it is important to monitor systems *via sensors*, and make reliable and time-sensitive decisions and detect anomalies (e.g., in order to maintain safe operation, identify faulty or compromised components, detect targets or obstacles, avoid collisions, protect incumbent users).

In Chapter 5, we specifically consider active hypothesis testing for the anomaly detection problem in which there are k abnormal processes out of N processes, where $0 \leq k \leq N$. During the detection process, the decision maker is allowed to observe only one of the N processes at a time. The distribution of the observations depends on whether the target is normal or not. In this setting, the objective of the decision

maker is to minimize the observation delay and dynamically determine all abnormal processes.

The original active hypothesis testing problem was investigated in [92]. Based on this work, several recent studies proposed more advanced anomaly detection techniques in more complicated and realistic situations. For example, the authors in [93] considered the case where the decision maker has only limited information on the distribution of the observation under each hypothesis. In [94], the performance measure is the Bayes risk that takes into account not only the sample complexity and detection errors, but also the costs associated with switching across processes. Moreover, authors in [95] considered the scenario that in some of the experiments, the distributions of the observations under different hypotheses are not distinguishable, and extended this work to a case with heterogenous processes [96], where the observation in each cell is independent and identically distributed (i.i.d.). Also, the study of stopping rule has drawn much interest. For instance, in [97], improvements were achieved over prior studies since the proposed decision threshold can be applied in more general cases. The authors in [98] leveraged the central limit theorem for the empirical measure in the test statistic of the composite hypothesis Hoeffding test, so as to establish weak convergence results for the test statistic, and, thereby, derive a new estimator for the threshold needed by the test.

Recently, machine learning-based methods have also been applied to such hypothesis testing problems. For instance, in [99], the deep Q-network has been employed for sequential hypothesis testing and change point detection. Considering the cybersecurity threats, authors in [100] proposed deep reinforcement one-shot learning for change point detection to address scenarios in which only a few training instances are available, for example, in the zero-day attack. And a random forest machine learning algorithm is presented in [101] to effectively detect compromised loT devices at distributed fog nodes. Moreover, in [102] an adversarial statistical learning method

has been proposed to detect slight changes in the statistical parameters caused by the attack data. Besides, there are a number of recent works specifically focus on the anomaly detection conducted among multiple processes. In such cases, active sequential hypothesis testing problem is often modeled as a partially observable Markov decision process, and the reinforcement learning algorithms are applied to dynamically select the processes to be tested. For instance, in [99] and [103], the application of deep Q-networks and actor-critic deep reinforcement learning have been investigated, respectively.

In addition, anomaly detection in multivariate time series has attracted interest recently. In such problems, deep learning algorithms are trained by both normal and abnormal training data, and used as classifiers to detect and diagnose the anomalies [104], [105]. Different from the deep learning-based detectors, the GAN-based framework proposed in [106] and [107] is only trained with the normal dataset, and estimate the probability of the anomaly.

Furthermore, several recent studies have taken the cost of the detection process into consideration. For instance, in [94], the performance measure is the Bayes risk that takes into account not only the sample complexity and detection errors, but also the costs associated with switching across processes. In [108], the cost is expressed as a function of the duration of the anomalous state. Considering the cost of both sampling and errors, authors in [109] proposed a deep reinforcement learning-based policy for significant sampling with applications to shortest path routing.

Motivated by these studies, we in Chapter 6 propose a novel GAN-based anomaly detector that has a prediction capability and detects threshold crossing in a stochastic time series without requiring the knowledge of its statistics. We evaluate the performance of this detector, determine the incurred sampling and delay costs, identify the key tradeoffs, and compare with existing strategies.

Based on the work conducted in Chapter 5 and 6, we consider more practical

noisy observations from sensors and the sensing cost control in Chapter 7. As mentioned before, various machine learning-based methods have been applied to address detection and hypothesis testing problems. For instance, learning approaches include the DQN [99] [100], adversarial statistical learning [102], and deep actor-critic reinforcement learning [103]. Compared with these methods, the recently proposed *soft actor-critic reinforcement learning algorithm* [110] exhibit advantages in exploring the unknown/uncertain environments due to the fact that the soft actor-critic reinforcement learning algorithm (SAC) is based on the maximum entropy reinforcement learning framework which encourages a more evenly distributed probabilities for all actions and attempts to find a balance between exploration and exploitation. And the authors in [111] show the advantages of SAC in handling constrained Markov decision processes. Motivated by this, we in this thesis propose a SAC-based decision-making agent for detection.

Moreover, we also consider the Byzantine attacks on the sensors. If the sensors are compromised, they become Byzantines which always quantize the signals to wrong states and send the distorted samples to the decision-maker. Conventionally, the Byzantine can be identified using Monte-Carlo methods [112]. However, in our setting, both the state of the target process and the state of the sensors may change before sufficient samples can be collected. Inspired by the application of the generative adversarial networks (GANs) in detecting the changes in the statistics of time series data [107] [106], we propose a GAN-based detector to identify the Byzantine sensors.

1.3 Outline and Main Contributions

In Chapter 2, we investigate the application of deep actor-critic reinforcement learning algorithm in edge caching problems. In terms of cache hit rate and transmission delay, we analyze the performance of the proposed algorithm. Also, comparisons between

the proposed framework and other caching policies are provided.

- In Section 2.1, for the first time, we present a deep reinforcement learning framework (that utilizes Wolpertinger architecture) for content caching. We define the state and action spaces and the reward function for the DRL agent, and employ this agent to make proper cache replacement decisions to maximize the cache hit rate. We analyze the performance of this DRL agent in terms of the cache hit rate. And we compare the performance with other caching algorithms, including Least Recently Used (LRU), Least Frequently Used (LFU), and First-In First-Out (FIFO) caching strategies. The results show that the DRL agent is able to achieve improved short-term cache hit rate and improved and stable long-term cache hit rate. We further confirm the effectiveness of the DRL agent through comparisons with deep Q-network. The results show that the DRL agent is able to achieve competitive cache hit rates while having significant advantages in runtime.

- In Section 2.2, we extent the study in Section 2.1 to a more practical situation. Specifically, we first provide a more detailed analysis of the proposed actor-critic DRL framework considering a single-cell wireless scenario with one base station. Subsequently, we extent it to a multi-agent actor-critic framework used for decentralized cooperative caching at multiple base stations. We analyze the performance of the proposed framework for centralized caching in terms of the cache hit rate, and provide comparisons with other caching polices including least recently used (LRU), least frequently used (LFU), and first-in first-out (FIFO) policies. We demonstrate that the DRL agent is able to achieve improved short-term cache hit rate and improved and stable long-term cache hit rate. We analyze the performance of the proposed multi-agent framework in terms of the cache hit rate and also the transmission delay reduction, and again provide comparisons with the LRU, LFU, and FIFO caching strategies.

We show that the proposed multi-agent framework can identify the popular contents effectively, and outperform the other schemes.

In Chapter 3, we propose an actor-critic deep reinforcement learning framework for dynamic multichannel access in a single-user scenario and show that this framework can work with a relatively larger number of channels than other deep reinforcement learning based approaches. We analyze the performance of the proposed framework and compare it with the deep Q-network (DQN) framework presented in [66]. Simulation results demonstrate that our proposed framework can achieve competitive performance in the case of 16 channels, and better performance in the cases of 32 and 64 channels. We test the proposed approach in time-varying scenarios, and the results demonstrate the adaptive ability of actor-critic deep reinforcement learning. Also, our framework leads to significant benefits in terms of time/computational efficiency. We extent the actor-critic algorithm-based framework to a multi-agent framework to solve the dynamic multichannel selection problem in the multi-user model. And this purely distributed multi-agent framework can work without any additional information exchange between the users. We provide the channel selection accuracy for the case in which multiple users make their access decisions simultaneously, and compare with other algorithms (such as DQN, slotted ALOHA, and the optimal policy when the channel dynamics/patterns are known).

In Chapter 4, we propose two adversarial policies, one based on feed-forward neural networks (FNNs) and the other based on deep reinforcement learning (DRL) policies. Both attack strategies aim at minimizing the accuracy of a DRL-based dynamic channel access agent. We first present the two frameworks and the dynamic attack procedures of the two adversarial policies. Then we demonstrate and compare their performances. Finally, the advantages and disadvantages of the two frameworks are identified.

In Chapter 5, we study deep reinforcement learning based active sequential testing

for anomaly detection. We assume that there is an unknown number of abnormal processes at a time and the agent can only check with one sensor in each sampling step. To maximize the confidence level of the decision and minimize the stopping time concurrently, we propose a deep actor-critic reinforcement learning framework that can dynamically select the sensor based on the posterior probabilities. We provide simulation results for both the training phase and testing phase, and compare the proposed framework with the Chernoff test in terms of claim delay and loss.

In Chapter 6, we study anomaly detection by considering the detection of threshold crossings in a stochastic time series without the knowledge of its statistics. To reduce the sampling cost in this detection process, we propose the use of hierarchical generative adversarial networks (GANs) to perform nonuniform sampling. In order to improve the detection accuracy and reduce the delay in detection, we introduce a buffer zone in the operation of the proposed GAN-based detector. In the experiments, we analyze the performance of the proposed hierarchical GAN detector considering the metrics of detection delay, miss rates, average cost of error, and sampling ratio. We identify the tradeoffs in the performance as the buffer zone sizes and the number of GAN levels in the hierarchy vary. We also compare the performance with that of a sampling policy that approximately minimizes the sum of average costs of sampling and error given the parameters of the stochastic process. We demonstrate that the proposed GAN-based detector can have significant performance improvements in terms of detection delay and average cost of error with a larger buffer zone but at the cost of increased sampling rates.

In Chapter 7, we model the sensor probing mechanism in the presence of two types of noise: the noise introduced by the sensors during sensing, and the noise in the communication links. With this practical sensing model, we propose a soft actor-critic reinforcement learning framework to address the detection problem with a sensing cost control. And, we consider the random Byzantine attacks on the sensing

model and design a GAN-based agent to identify the Byzantine sensors. To evaluate the proposed framework, we consider accuracy, stopping time, and total cost as the performance metrics. In the experiments, the proposed SAC framework is compared to the conventional actor-critic algorithm. Via simulation results, we demonstrate that the proposed SAC agent can be more robust in different test cases, and the proposed GAN detector is able to identify the Byzantines with high accuracy and help to recover the detection performance achieved in the absence of Byzantine attacks.

1.4 Bibliographic Note

- The results in Section 2.1 appeared in the conference paper:
  - C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, 2018, pp. 1-6.
- The results in Section 2.2 appeared in the journal paper:
  - C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep reinforcement learning-based edge caching in wireless networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 1, pp. 48-61, 2020.
- The results in Chapter 3 appeared in the journal paper:
  - C. Zhong, Z. Lu, M. C. Gursoy, and S. Velipasalar, "A deep actor-critic reinforcement learning framework for dynamic multichannel access," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1125-1139, 2019.
- The results in Chapter 4 appeared in the conference paper:

- C. Zhong, F. Wang, M. C. Gursoy, and S. Velipasalar, "Adversarial jamming attacks on deep reinforcement learning based dynamic multichannel access," in *2020 IEEE Wireless Communications and Networking Conference (WCNC)*, 2020, pp. 1-6.

- The results in Chapter 5 appeared in the conference paper:
  - C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep actor-critic reinforcement learning for anomaly detection," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1-6.

- The results in Chapter 6 appeared in the conference paper:
  - C. Zhong, M. C. Gursoy and S. Velipasalar, "Anomaly Detection and Sampling Cost Control via Hierarchical GANs," GLOBECOM 2020 - 2020 IEEE Global Communications Conference, 2020, pp. 1-6.

- The content in Chapter 7 will be submitted as a journal paper:
  - C. Zhong, M. C. Gursoy and S. Velipasalar, "Robust Learning-Based Detection with Cost Control and Byzantine Mitigation."

Chapter 2

Deep Reinforcement Learning Based Content Caching Strategies

In this chapter, we propose the deep actor-critic reinforcement learning frameworks for edge caching problems. Considering both single-cell and multi-cell wireless scenarios, we design the DRL-based frameworks and demonstrate their performance respectively.

In Chapter 2.1, we present a DRL-based framework with Wolpertinger architecture for content caching at the base station. The proposed framework is aimed at maximizing the long-term cache hit rate, and it requires no knowledge of the content popularity distribution. To evaluate the proposed framework, we compare the performance with other caching algorithms, including Least Recently Used (LRU), Least Frequently Used (LFU), and First-In First-Out (FIFO) caching strategies. Meanwhile, since the Wolpertinger architecture can effectively limit the action space size, we also compare the performance with Deep Q-Network to identify the impact of dropping a portion of the actions. Our results show that the proposed framework can achieve improved short-term cache hit rate and stable long-term cache hit rate in comparison with LRU, LFU, and FIFO schemes. Additionally, the performance is shown to be competitive in comparison to Deep Q-learning, while the proposed

framework can provide significant savings in runtime.

In Chapter 2.2, with the DRL framework proposed in Chapter 2.1, we conduct more detailed analyses. We initially consider a single-cell wireless scenario with one base station, and study centralized caching where the single base station is the only cache-enabled content server. Subsequently, we address a multi-cell wireless network, and consider a decentralized caching framework, where each base station is equipped with caching storage space. For centralized edge caching, we aim at maximizing the cache hit rate. In decentralized edge caching, we consider both the cache hit rate and transmission delay as performance metrics. The proposed frameworks are assumed to neither have any prior information on the file popularities nor know the potential variations in such information. Via simulation results, the superiority of the proposed frameworks is verified by comparing them with other policies, including LFU, least recently LRU, and FIFO policies.

2.1 A Deep Reinforcement Learning-Based Framework for Content Caching

2.1.1 System Model

Data traffic is triggered by requests from the rapidly increasing number of end-users, and the volume of requests varies over time. In this setting, we propose a deep reinforcement learning framework acting as an agent. Based on the users' requests, this DRL agent makes caching decisions to store the frequently requested contents at local storage. If the requested contents are already cached locally, then the base station can serve the user directly with reduced delay. Otherwise, the base station requests these contents from the original server and updates the local cache based on the caching policy.

In this section, we consider a single base station with cache size of C. We assume that in a given time slot, the total number of contents that users can request from this base station is fixed and denoted as N. We give every content a unique index, and this index acts as the content ID. We assume that all contents have the same size. The list of users' requests is denoted as $Req = \{R_1, R_2, R_3, ...\}$. Here, R_t denotes the ID of the requested content at time t. For each request, the DRL agent makes a decision on whether or not to store the currently requested content in the cache, and if yes, the agent determines which local content will be replaced.

We define $\mathcal{A}$ as the action space, and let $\mathcal{A} = \{a_1, a_2, a_3, ..., a_m\}$, where a_v denotes a valid action. And in our case, m has a finite but generally a large value, describing the total number of possible actions. For each content, there are two cache states: cached, and not cached. The cache state gets updated based on the caching decision. Here, we define two types of actions: the first one is to find a pair of contents and exchange the cache states of the two contents; the second one is to keep the cache states of the contents unchanged. Theoretically, multiple actions can be executed at one decision epoch. To reduce the computational complexity, we need to limit the action space size m and the number of actions to be executed in one decision epoch, which will discussed in detail in Section 2.1.2.

The reward should reflect the objective of the framework, which, in our case, is to reduce the data traffic. In our setting, all requests are served by the base station, all contents have the same size, and there are no priorities for users. Therefore, the reduction in data traffic can be evaluated in terms of the cache hit rate. Here, we define the cache hit rate CHR in T requests as

$$CHR_T = \frac{\sum_{i=1}^{T} \mathbf{1}\left(R_i\right)}{T} \tag{2.1}$$

where indicator function $\mathbf{1}(R_i)$ is defined as

$$\mathbf{1}(R_i) = \begin{cases} 1, & R_i \in \mathcal{C}_T, \\ 0 & R_i \notin \mathcal{C}_T \end{cases} \tag{2.2}$$

where $\mathcal{C}_T$ stands for the cache state in this period. Therefore the reward in T requests can be defined as

$$r^T = CHR_T. \tag{2.3}$$

For each decision epoch t, we obtain reward r_t, which can be a weighted sum of short-term and long-term cache hit rates. We more explicitly introduce the definition of r_t for the proposed framework in Section 2.1.2 below.

The objective of the DRL agent is to find a policy, σ^*, that maximizes the long-term cache hit rate:

$$\underset{\sigma^*}{\text{maximize}} \quad E[r_t|\sigma^*]. \tag{2.4}$$

We are interested in developing a model-free learning algorithm to solve problem (2.4) that can effectively reduce the data traffic with fixed cache capacity at the base station.

2.1.2 DRL-based Content Caching Framework

In this section, we present the DRL-based content caching framework, which is aimed at maximizing the cache hit rate in order to reduce the data traffic. To solve the content caching problem with high-dimensional state and action spaces (due to the large number of contents and cache sizes in practical scenarios), we propose a framework based on the Wolpertinger architecture [42] to narrow the size of the action space and avoid missing the optimal policy at the same time.

2.1.2.1 Algorithm Overview

Based on the Wolpertinger Policy [42], our framework consists of three main parts: actor network, K-nearest neighbors (KNN), and critic network. We train the policy using the Deep Deterministic Policy Gradient (DDPG) [113]. The Wolpertinger architecture is employed for two reasons: 1) as an online algorithm, this framework can adapt to data, and enables us to develop a long-term policy; 2) actor network can avoid the listing and consideration of very large action space, while the critic network can correct the decision made by the actor network, and KNN can help to expand the actions to avoid poor decisions. This algorithm work in three steps. Firstly, the actor network takes cache state and the current content request as its input, and provides a single proto actor $\hat{a}$ at its output. Then, KNN receives the single actor $\hat{a}$ as its input, and calculate the l_2 distance between every valid action and the proto actor in order to expand the proto actor to an action space, denoted by $\mathcal{A}_k$, with K elements and each element being a possible action $a_v \in \mathcal{A}$. And at the last step, the critic network takes the action space $\mathcal{A}_k$ as its input, and refines the actor network on the basis of the Q value. The DDPG is applied to update both critic and actor networks.

Below we provide a more detailed description of the key components of the algorithm.

The actor: The actor network is defined as a function parameterized by θ^μ, mapping the state $\mathcal{S}$ from the state space $\mathbb{R}^s$ to the action space $\mathbb{R}^a$. The mapping provides a proto-actor $\hat{a}$ in $\mathbb{R}^a$ for a given state under the current parameter. Here, we scale the proto-actor to make sure $\hat{a}$ is a valid action that $\hat{a} \in \mathcal{A}$:

$$\mu(s|\theta^\mu) : \mathcal{S} \rightarrow \mathbb{R}^a$$

$$\mu(s|\theta^\mu) = \hat{a}. \tag{2.5}$$

K-nearest neighbors: The generation of proto-actor can help reduce the computa-

tional complexity caused by the large size of the action space. However, reducing the high-dimensional action space to one actor will lead to poor decision making. So, the K-nearest neighbors mapping, g_k, is applied to expand the actor $\hat{a}$ to a set of valid actions in action space $\mathcal{A}$. The set of actions returned by g_k is denoted as $\mathcal{A}_k$:

$$\mathcal{A}_k = g_k(\hat{a}_t)$$

$$g_k = \arg\max_{a \in \mathcal{A}}^{k} |a - \hat{a}|^2. \tag{2.6}$$

The critic: To avoid the actor with low Q-value being occasionally selected, a critic network is defined to refine the actor. This deterministic target policy is described below:

$$Q(s_t, a_j|\theta^Q) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_j) + \gamma Q(s_{t+1}, a_{t+1}|\theta^Q)] \tag{2.7}$$

where θ^Q stands for the parameters of the critic network, and $\gamma \in (0, 1]$ is the discount factor which weigh the future accumulative reward $Q(s_{t+1}, a_{t+1}|\theta^Q)$. Here, the critic takes both the current state s_t and the next state s_{t+1} as its input to calculate the Q value for each action in $\mathcal{A}_k$. Then, the action that provides the maximum Q value will be chosen as a_t, i.e.,

$$a_t = \arg\max_{a_j \in \mathcal{A}_k} Q(s_t, a_j|\theta^Q) \tag{2.8}$$

Update: The actor policy is updated using deep deterministic policy gradient, which is given as

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\mu^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}. \tag{2.9}$$

The update of critic network parameter θ^Q and actor network parameter θ^μ are given as

$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'} \tag{2.10}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \tag{2.11}$$

where $\tau \ll 1$.

2.1.2.2 Workflow

In this part, we introduce the workflow of the proposed framework. The framework consists of two phases, namely offline and online phases.

Offline phase: In the offline phase, the actor and critic networks are constructed and pre-trained with historic transition profiles. This process is the same as in the training of a deep neural network. In the offline phase, when we train the networks with sufficient number of samples, the critic and actor will be sufficiently accurate, and the updated parameters θ^Q and θ^μ will be stored in order to provide a good initial point for the online phase.

Online phase: The online phase is initialized with the parameters determined in the offline phase. The system is dynamically controlled in the online phase. In each epoch t, if the requested content is not cached, the DRL agent observes the state s_t from the environment, and obtains the proto actor and Q value from the actor network and critic network, respectively. Then, an ϵ-greedy policy is applied at selecting the execution action a_t. This policy can force the agent to explore more possible actions. After the action a_t is executed, the DRL agent observes the reward r_t and next state s_t from the base station cache, and the transition (s_i, a_i, r_i, s_{i+1}) will be stored to the memory $\mathcal{M}$ at the end of each epoch. The DRL agent updates the parameters θ^Q and θ^μ with $N_\mathcal{B}$ transition samples from memory $\mathcal{M}$ based on the DDPG.

In our implementation, the actor network has two hidden layers of fully-connected units with 256 and 128 neurons, respectively; and the critic network has two hidden layers of fully-connected units with 64 and 32 neurons, respectively. The capacity of memory $N_\mathcal{M}$ is set as $N_\mathcal{M} = 10000$, and the mini batch size is set as $N_\mathcal{B} = 100$. The discount factor γ introduced in (2.7) is set as 0.9.

Then, we define the state and action spaces, and the reward function of the DRL agent as follows:

State Space: The DRL agent assumes the feature space of the cached contents and the currently requested content as the state. The feature space consists of three components: short-term feature $\mathcal{F}_s$, medium-term feature $\mathcal{F}_m$, and long-term feature $\mathcal{F}_l$, which represent the total number of requests for each content in a specific short-, medium-, long-term, respectively. These features vary as the cache state is updated. For each decision epoch, we assign a temporary index to every content from which we need to extract features. Since we only extract the features from cached contents and the currently requested content, let the index range from 0 to the cache capacity C. The index of the currently requested content is 0, while the index of the cached content varies from 1 to C. This temporary index is different from the content ID and is only used for denoting the feature. Then, we let f_{xj}, for $x \in \{s, m, l\}$ and $j \in [0, C]$, denote the feature of a specific content within a specific term. Thus, the observed state is defined as $s_t = \{\mathcal{F}_s; \mathcal{F}_m; \mathcal{F}_l\}$ where $\mathcal{F}_s = \{f_{s0}, f_{s1}, ..., f_{sC}\}$, $\mathcal{F}_m = \{f_{m0}, f_{m1}, ..., f_{mC}\}$, and $\mathcal{F}_l = \{f_{l0}, f_{l1}, ..., f_{lC}\}$.

Action Space: In order to limit the action size, we restrict that the DRL agent can only replace one selected cached content by the currently requested content, or keep the cache state the same. With this, we define $\mathcal{A}$ as the action space, and let $\mathcal{A} = \{0, 1, 2, ..., C\}$, where C is again the cache capacity at the base station. And we assume that only one action can be selected in each decision epoch. Let a_t be the selected action at epoch t. Note that, for each caching decision, there are $(C + 1)$ possible actions. When $a_t = 0$, the currently requested content is not stored, and the current caching space is not updated. And when $a_t = \upsilon$ with $\upsilon \in \{1, 2, ..., C\}$, the action is to store the currently requested content by replacing the υth content in the cache space.

Reward: As stated in the previous section, we select the cache hit rate as the

reward to represent the objective of the proposed framework. The reward for each decision epoch depends on the short and long-term cache hit rate. For example, we set the short-term reward as the number of requests for local content in the next epoch, i.e., the short-term reward r_t^s can be either 0 or 1. And let the total number of requests for local content within the next 100 requests as the long-term reward $r_t^l \in [1, 100]$. The total reward for each step is defined as the weighted sum of the short and long-term rewards

$$r_t = r_t^s + w * r_t^l$$

where w is the weight to balance the short and long-term rewards, so that we can give more priority to the short-term reward to maximize the cache hit rate at every step given the chosen action.

The major notations are listed in Table 2.1 below.

Table 2.1: Notations

Notation	Description
C	Cache capacity at base station
i	ID, or index of contents
N	Total number of contents
R_t	Content requested at epoch t
$\mathcal{A}$	Action space
a_t	The chosen action in the epoch t
r_t	The reward obtained in the epoch t
s_t	The observation state in the epoch t
$\mathcal{F}$	Feature space
$\mathcal{F}_s$, $\mathcal{F}_m$, $\mathcal{F}_l$	Short/ mid/ long term features
f_{si}, f_{mi}, f_{li}	Short/ mid/ long term feature of content i
k	The number of nearest neighbors

Algorithm 1 DRL-based Content Caching Algorithm

Offline:

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \longleftarrow \theta^Q, \theta^{\mu'} \longleftarrow \theta^\mu$
Initialize replay buffer $\mathcal{M}$ with capacity of $N_\mathcal{M}$
Initialize a random process $\mathcal{N}$ for action exploration
Initialize features space $\mathcal{F}$
Pre-train the actor and critic network with the pairs $< s, a >$ and the corresponding $Q(s, a|\theta^Q)$.

Online:

for $t = 1, T$ **do**
    The base station receive a request R_t
    if Requested content is already cached **then**
        Update cache hit rate and end epoch;
    else
        if Cache storage is not full **then**
            Cache the currently requested content
            Update cache state and cache hit rate
            End epoch;
        end if
        Receive observation state s_t
        Actor:
        Receive proto-ation from actor network $\hat{a}_t = \mu(s_t|\theta^\mu)$.
        KNN:
        Retrieve k approximately closest actions $\mathcal{A}_k = g_k(\hat{a}_t)$
        Critic:
        Select action $a_t = \arg\max_{a_j \in \mathcal{A}_k} Q(s_t, a_j|\theta^Q)$ according to the current policy.
        Execute action a_t, and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{M}$
        Sample a random mini batch of $N_\mathcal{B}$ transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{M}$
        Set target $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
    $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\mu^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \longleftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
        Update the cache state
        Update features space $\mathcal{F}$
        Update cache hit rate
    end if
end for

2.1.3 Simulation Results

2.1.3.1 Simulation Setup

Data Generation: In our simulations, the raw data of users' requests is generated according to the Zipf distribution. We set the total number of files as 5000, and we have collected 10000 requests as the testing data. We generate two types of data sets. Initially, we analyze the performance with fixed content popularity distribution, and the data set was generated with unchanged popularity distribution with Zipf parameter set as 1.3. Subsequently, we study how long-term cache hit rate varies over time as the content popularity distribution changes. In this case, the data set was generated with a varying Zipf parameter, and changing content popularity rank. Note that, although we generate the data using the Zipf distribution, the proposed framework is applicable to arbitrarily distributed popularities, and indeed requires no knowledge regarding the popularity distribution.

Feature Extraction: From the raw data of content requests we extract the feature F and use it as the input state of the network. Here, as features, we consider the number of requests for a file within the most recent 10, 100, 1000 requests.

2.1.3.2 Performance Comparison

To analyze the performance of our algorithm, we evaluate the cache hit rate and provide comparisons with other caching strategies.

2.1.3.2.1 Cache Hit Rate In this part, comparisons are made between our proposed framework and the following caching algorithms:

- **Least Recently Used (LRU) [114]:** In this policy, the system keeps track of the most recent requests for every cached content. And when the cache storage is full, the cached content, which is least requested recently, will be replaced by the new content.

- **Least Frequently Used (LFU) [115]:** In this policy, the system keeps track of the number of requests for every cached content. And when the cache storage is full, the cached content, which is requested the least many times, will be replaced by the new content.

- **First In First Out (FIFO) [116]:** In this policy, the system, for each cached content, records the time when the content is cached. And when the cache storage is full, the cached content, which was stored earliest, will be replaced by the new content.

Here, we consider both short-term and long-term performance. For the short-term analysis, we study the relationship between the cache capacity and cache hit rate. Regarding the long-term performance, we are interested in the stability and robustness of the proposed DRL framework, i.e., we seek to characterize how the cache hit rate changes over time with the changing popularity distribution of contents.

Figure 2.1 shows the overall cache hit rate achieved by the proposed framework and the other caching algorithms introduced above. In this study, we set the Zipf distribution parameter as 1.3. We can see that our proposed framework provides a higher cache hit rate for all cache capacity values. When the cache capacity is small, the performance of LFU is very close to our proposed framework. As the cache capacity increases, the gap between proposed framework and other three caching algorithms increases at first, and then gradually decreases. At cache capacity $C = 500$, the cache hit rate of all four algorithms are close to each other at around 0.8. And at this point, the cache hit rates achieved by different policies tend to converge because the cache capacity is high enough to store all popular contents. From this point on, increasing the cache capacity will not improve the cache hit rate effectively any more, and the cache hit rate is now limited by the distribution of the content popularity.

In Fig. 2.2, we address the long-term cache hit rate, and based on the long-

term performance we evaluate the capability that the policy can maintain the good performance as content popularities vary over time. Specifically, we design a data set with a changing popularity distribution based on the Zipf distribution. In addition to the parameter of the Zipf distribution, the rank of the contents also vary over time. All the Zipf distribution parameter values and the ranks of contents are generated randomly. From the figure, we can observe that the proposed DRL framework doesn't show advantage initially, but soon the cache hit rate increases. This is because the proposed framework needs to update the deep neural network to adapt to the changing content popularity distribution. After that, the hit rate curve of proposed framework reaches the peak and then deceases only slightly, maintaining a relatively stable cache hit rate. Meanwhile, the LFU curve starts at a relative high cache hit rate and then drops rapidly. This poor performance is caused by the frequency pollution, which is an inevitable drawback of the LFU policy. Because the number of requests are accumulative, when the popularity distribution changes, the previous record will mislead the system. For LRU and FIFO, the performance are relatively stable but the performance is not competitive with respect to our DRL agent. Based on the analysis, our proposed framework will be more suitable for applications that require robustness and a long-term high performance.

2.1.3.2.2 Efficiency In this part, we compare our proposed framework with the Deep Q-learning based caching algorithm. The most significant difference between these two algorithms is that our proposed algorithm only considers a set of valid actions expanded from the actor, but the Deep Q-learning based algorithm calculates the value for all valid actions. Intuitively, our proposed framework will reduce the computational complexity, but since the Deep Q-learning algorithm receives more possible actions, it may lead to better performance.

To address this key tradeoff, we compare the cache hit rates and the corresponding

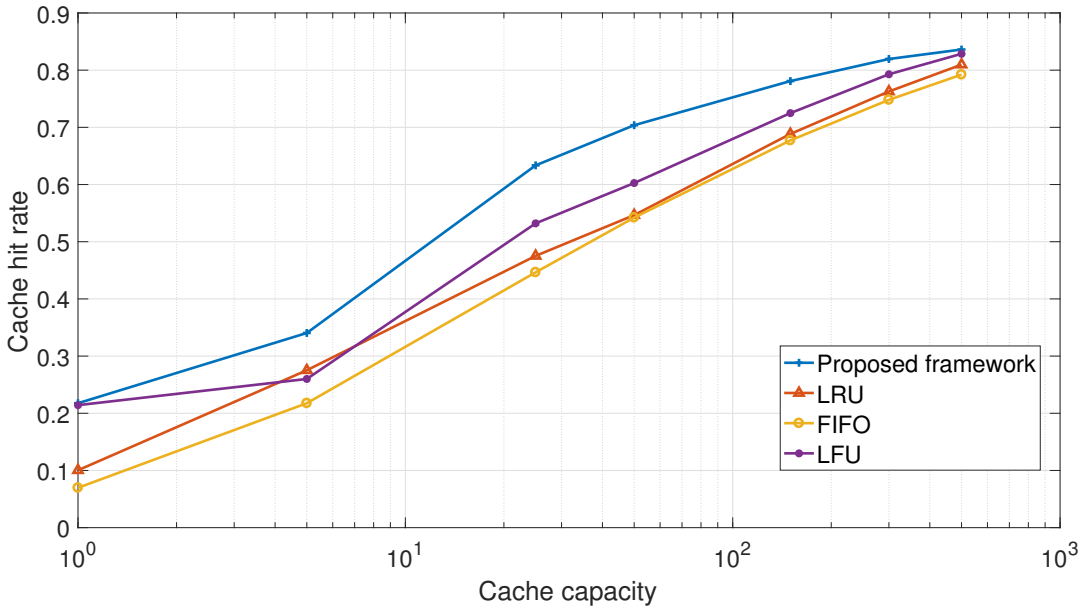


Figure 2.1: Cache hit rate vs. cache capacity. We vary the cache capacity as $C = 1, 5, 25, 50, 150, 300, 500$.

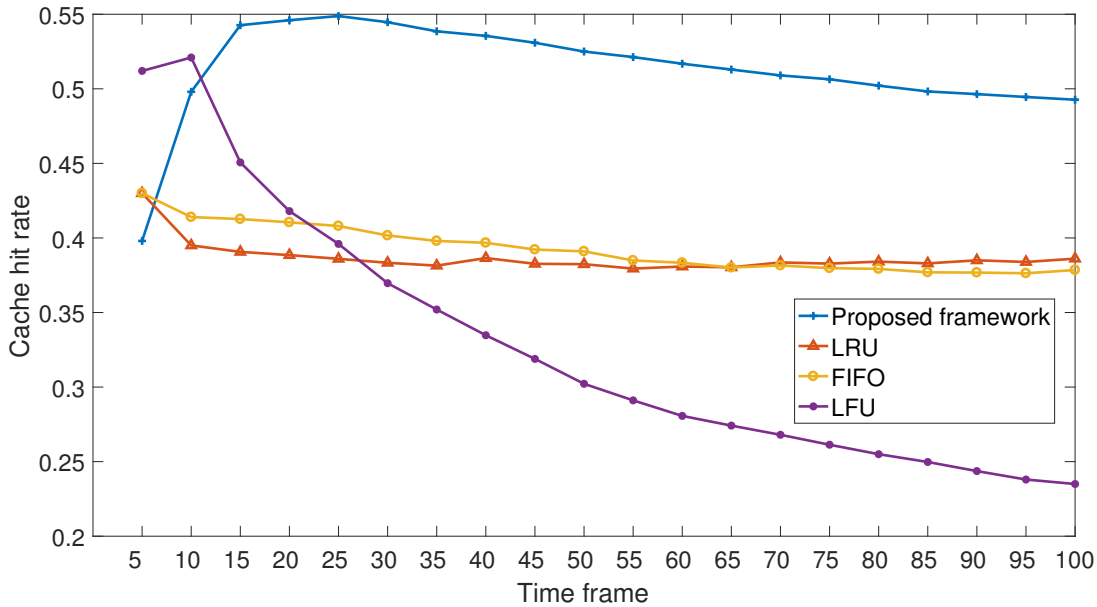


Figure 2.2: Cache hit rate as the content popularity distribution changes over time, with cache capacity fixed at $C = 300$.

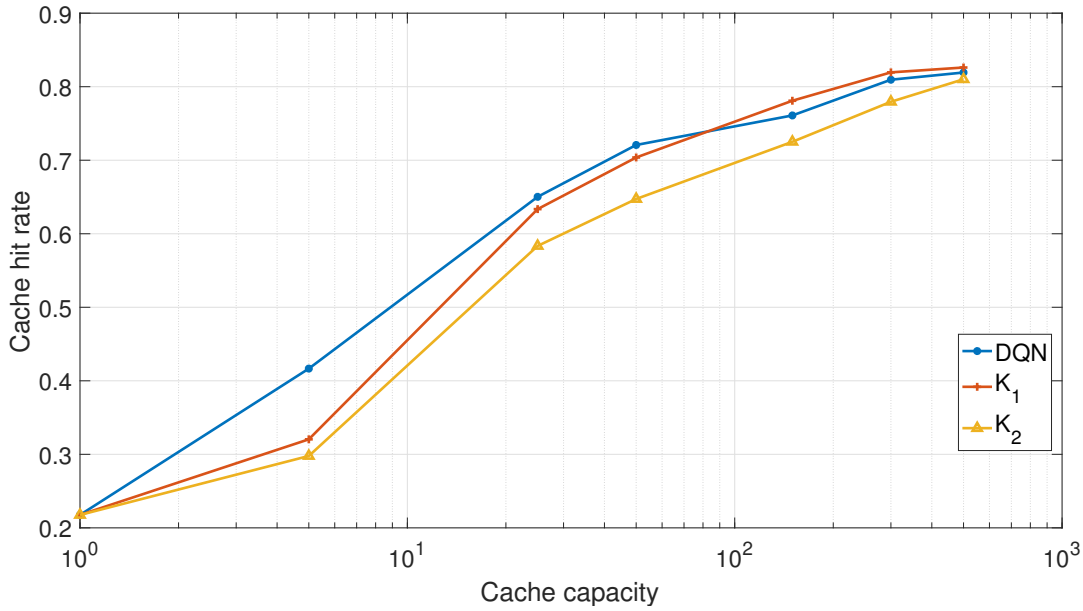


Figure 2.3: Cache hit rate vs. cache capacity.

runtimes of these two deep learning schemes. In Fig. 2.1, the cache capacity values vary as $\{1, 5, 25, 50, 150, 300, 500\}$, and the cache hit rates are plotted when the content requests are generated using the Zipf distribution parameter 1.3. The curve labeled DQN represents the performance of the deep Q-network. K_1 and K_2 denote two different settings of proposed framework. In the case of K_1, the KNN returns $k_1 = \lceil 0.15C \rceil$ actions to the expanded action space $\mathcal{A}_k$. For K_2, the KNN returns $k_2 = \lceil 0.05C \rceil$ actions to the expanded action space $\mathcal{A}_k$. As we can see in the figure, when cache capacity is $C = 1$, all three curves intersect at the same point, because all three policies are trained to find the one most popular content. Then, as cache capacity increases, the gap between this three policies become obvious. Especially when the cache capacity is $C = 5$, DQN consider all possible actions, while both K_1 and K_2 only take the proto actor. The gap between K_1 and K_2 reflects the randomness that might be introduced by the proto action. And then, the gap between K_1 and DQN gradually decreases. These results demonstrate that the proposed framework can achieve competitive cache hit rates compared to DQN.

Moreover, the proposed framework can achieve this competitive performance with significantly lower runtimes. With cache capacity fixed at $C = 300$, we record the time needed for 1000 decision epochs, and show the average runtime results in Table 2.2 below. As can be seen, the DQN needs much more time at each epoch. In practice, this increased computational cost often leads to storage problems, which makes the deep Q network less competitive in solving large scale problems than the proposed framework.

Table 2.2: Runtime/decision epoch

	DQN	K_1	K_2
Runtime (s)	1.2225	0.3224	0.1163

2.2 Deep Reinforcement Learning Based Edge Caching in Wireless Networks

This section is organized as follows. First, in Sections 2.2.1 and 2.2.2, we focus on the study of a centralized caching system. Specifically, in Section 2.2.1, the single-cell system model and the cache hit rate maximization problem is introduced. In Section 2.2.2, the deep actor-critic reinforcement learning based centralized edge caching framework is proposed. Subsequently, we extent our analysis to the decentralized edge caching scenario in Sections 2.2.3 and 2.2.4. In Section 2.2.3, we introduce the multi-cell system and the decentralized edge caching problems based on the overall cache hit rate and transmission delay. And in Section 2.2.4, we demonstrate the multi-agent framework for the decentralized edge caching scenario. Numerical results are presented and discussed in Section 2.2.5.

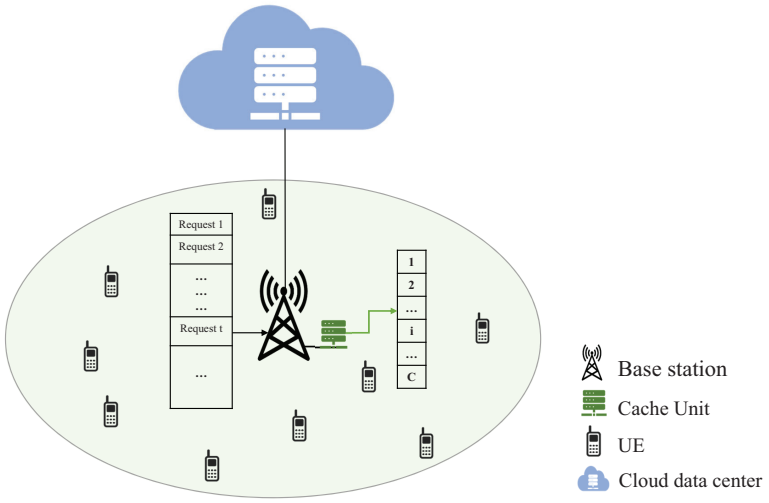


Figure 2.4: System model of a centralized caching system.

2.2.1 Centralized Edge Caching in a Single-Cell Network: System Model and Problem Formulation

2.2.1.1 System Model

In the centralized caching system, shown in Fig. 2.4, we assume that there is only one cache-enabled base station, which is the content server for all users in its coverage. It is also assumed that the total number of contents that can be requested by the user is M and the base station can store C contents at most[1]. These contents have different popularities, which can be quantified by the probability that the content will be requested by the users. In this centralized caching system, we assume that the requests from users arrive at the base station one by one, and a Zipf distribution is used to approximatively describe the popularity distribution of the files at all users. Here, we assign each content a unique index, and use this index as the content ID when users request the content. So, we can denote the requests from the users as $Req = \{R_1, R_2, R_3, ...\}$, where R_t denotes the ID of the requested content at time t.

Since the base station is equipped with cache storage, it first checks if the contents

[1]In this chapter, we assume that all contents have the same size. And based on this assumption, we use the number of contents, C, that can be stored at a base station as the cache capacity.

requested by the users are cached locally. If the requested contents are available in the local cache, then the base station can transmit the contents to the corresponding user without requesting them from the upper level content servers. To avoid requesting content as much as possible, the base station needs to update its cache according to users' preferences. Each time a request arrives, the base station, as noted above, will first check if the requested content is available locally, so that it can decide how to serve the user. Then, the base station has to decide whether or not to update its cache. Therefore, for each content, there are two cache states: cached, and not cached. The cache state gets updated based on the caching decision. Here, we define two types of actions: the first one is to find a pair of contents and exchange the cache states of the two contents and the second one is to keep the cache states of all the contents unchanged. We describe the action space at the base station as $\mathcal{A} = \{a_1, a_2, a_3, ..., a_m\}$, where a_υ, $\upsilon = 1, 2, \ldots, m$, denotes a valid action and m is the size of the action space. Note that the replacement in the cache requires two contents in pair: one is to be added to the cache, and the other one is to be removed from the cache. Hence, the value of m depends on the cache capacity and the number of content files, and has a finite but generally a large value. Theoretically, multiple actions can be executed in one decision epoch. To reduce the computational complexity, we need to limit the action space size m and the number of actions to be executed in one decision epoch. For this purpose, we propose to employ the Wolpertinger architecture based DRL framework as will be discussed in detail in subsection 2.2.2.2.

2.2.1.2 Problem Formulation

In this part, we formulate the caching problem with the objective to maximize the cache hit rate, which describes how frequently the requested content is found in the local cache.

For the centralized caching system, the cache hit rate is computed from the per-

spective of the base station, and the cache hit rate P_{hit}^{c} in T requests is defined as

$$P_{hit}^{c} = \frac{\sum_{t=1}^{T} \mathbf{1}\left(R_t\right)}{T} \tag{2.12}$$

where indicator function $\mathbf{1}\left(R_t\right)$ is defined as

$$\mathbf{1}\left(R_t\right) = \begin{cases} 1, & \text{if the request } R_t \text{ hits in the cache,} \\ 0, & \text{otherwise} \end{cases} \tag{2.13}$$

with the cache being essentially described as the set of indices of the contents in the cache at time t (with cardinality equal to the cache capacity C).

And the problem of the maximization of the cache hit rate over the caching states can be expressed as

$$\textbf{P1:} \qquad \underset{\mathbf{\Phi}}{\text{Maximize}} \qquad P_{hit}^{c} \tag{2.14}$$

$$\text{Subject to} \qquad \sum_{f=1}^{M} \phi_f \leq C \tag{2.15}$$

where $\mathbf{\Phi}$ is the $1 \times M$ dimensional content state vector that records the states of all contents (describing whether they are cached or not), and each element ϕ_f in the content state vector is an indicator to show if the file is cached:

$$\phi_f = \begin{cases} 1 & \text{if the file } f \text{ is cached at the base station} \\ 0 & \text{if the file } f \text{ is not cached at the base station} \end{cases} . \tag{2.16}$$

2.2.2 Deep Actor-Critic Framework for Centralized Edge Caching

To solve the optimization problem **P1**, we in this section propose a Wolpertinger architecture based single-agent actor-critic DRL framework for centralized edge caching. First, we introduce the related definitions in this architecture. Several of these defini-

tions will also be used for the decentralized edge caching framework in Sections 2.2.3 and 2.2.4.

2.2.2.1 Related Definitions

2.2.2.1.1 Agent's Observation and State Space

Observations of the Centralized DRL Agent The centralized DRL caching agent assumes the feature space of the cached contents and the currently requested content as the state. In each decision epoch, we assign a temporary index to each content from which we need to extract features. Since we only extract the features from cached contents and the currently requested content, we let the indices range from 0 to cache capacity C. The index of the currently requested content is 0, while the index of the cached content varies from 1 to C. This temporary index is different from the content ID and is only used for denoting the feature. Thus, the observed state at time t is defined as $s_t = \{\mathcal{F}_s; \mathcal{F}_m; \mathcal{F}_l\}$, where $\mathcal{F}_s$, $\mathcal{F}_m$, $\mathcal{F}_l$ are the features collected at different times, as will be discussed next.

Feature Space The feature space consists of three components: short-term feature $\mathcal{F}_s$, medium-term feature $\mathcal{F}_m$, and long-term feature $\mathcal{F}_l$, which represent the total number of requests for each content in a specific short-, medium-, long-term, respectively. These features are updated as new requests arrive at agents. Then, we let f_{xj}, for $x \in \{s, m, l\}$ and $j \in \{1, \ldots, M\}$, denote the feature of a specific content within a specific term, where M is the total number of contents. As mentioned above, the observation for each agent can be expressed by $\{\mathcal{F}_s; \mathcal{F}_m; \mathcal{F}_l\}$, and we have $\mathcal{F}_s = \{f_{s0}, f_{s1}, ..., f_{sM}\}$, $\mathcal{F}_m = \{f_{m0}, f_{m1}, ..., f_{mM}\}$, and $\mathcal{F}_l = \{f_{l0}, f_{l1}, ..., f_{lM}\}$.

2.2.2.1.2 Action Space In order to limit the action size, we restrict that the DRL agent can either only replace one selected cached content by the currently requested

content, or keep the cache state the same. Thus, each replacement action of the caching agents indicates a pair of content IDs: one is the ID of the cached content to be deleted from the cache, and the other one is the ID of the content which is currently requested. So, all possible replacement actions can be described by a $C \times L$ matrix, where C is the cache capacity and L is the number of requests arriving simultaneously. For the centralized caching agent, since we assume that the users' requests arrive one by one, we have $L = 1$, which means that the replacement decision must be made between only the current requested content and a cached content. Therefore, the action space of the centralized caching agent can be defined as $\mathcal{A} = \{0, 1, 2, ..., C\}$, where C is again the cache capacity at the base station.

And we assume that only one action can be selected in each decision epoch. Let a_t be the selected action in epoch t. Note that for each caching decision, there are $(C + 1)$ possible actions. When $a_t = 0$, the currently requested content is not stored, and the current caching space is not updated. And when $a_t = \upsilon$ for $\upsilon \in \{1, 2, ..., C\}$, the action is to store the currently requested content by replacing the υth content in the cache space.

2.2.2.1.3 Reward As stated in the previous section, the centralized caching agent aims at maximizing the cache hit rate to solve problem **P1**. The reward r_t for each decision epoch depends on the short and long-term cache hit rate. For example, we set the short-term reward, considering the number of requests for local content in the next epoch, i.e., the short-term reward $P^c_{hit,s}$ can be either 0 or 1. And let the total normalized number of requests for local content within the next 100 requests as the long-term reward $P^c_{hit,l} \in [0, 1]$. The total reward for each step is defined as the weighted sum of the short and long-term rewards

$$r_t = P^c_{hit,s} + w * P^c_{hit,l} \tag{2.17}$$

where w is the weight to balance the short and long-term rewards. For instance, if we lower the value of w, we give more priority to the short-term reward to maximize the cache hit rate at every step given the chosen action.

2.2.2.2 Wolpertinger Architecture

Based on the Wolpertinger Policy [42], our framework consists of three main components: actor network, K-nearest neighbors (KNN), and critic network. We train the centralized caching policy using the Deep Deterministic Policy Gradient (DDPG) [113]. This Wolpertinger architecture works in three steps. First, the actor network takes cache state and the current content request as its input, and provides a single proto actor $\hat{a}$ at its output. Then, KNN receives the single actor $\hat{a}$ as its input, and calculate the L_2 distance between every valid action and the proto actor in order to expand the proto actor to an action space, denoted by $\mathcal{A}_K$, with K elements and each element being a possible action $a_v \in \mathcal{A}$. And at the last step, the critic network takes the action space $\mathcal{A}_K$ as its input, and refines the actor network on the basis of the Q value. The DDPG is applied to update both critic and actor networks.

Below we provide a more detailed description of the key components of the algorithm.

The actor: The actor network is designed to choose a proto-actor $\hat{a} \in \mathcal{A}$ from the valid actions. This selection is based on the decision policy of the actor network, and will be updated after each decision.

K-nearest neighbors (KNN): The generation of the proto-actor can help reduce the potentially high computational complexity due to the large size of the action space. However, reducing the high-dimensional action space to one actor will lead to poor decision making. To remedy this, the K-nearest neighbors mapping, g_K, is applied to expand the actor $\hat{a}$ to a set of valid actions selected from the action space $\mathcal{A}$. The

set of actions returned by g_K is denoted by $\mathcal{A}_K$:

$$\mathcal{A}_K = g_K(\hat{a}_t) \tag{2.18}$$

where

$$g_K = \arg\min_{a \in \mathcal{A}}^{K} |a - \hat{a}|^2. \tag{2.19}$$

With (2.19), we determine the K nearest neighbors of the proto-actor. Here, a is a valid action in the action space $\mathcal{A}$, and $|a - \hat{a}|^2$ is the L_2 distance of the features between the action a and the proto-actor $\hat{a}$. When the proto-actor is selected by the actor network, the agent will traverse the action space to find the K nearest feature distances, and the action set will be determined accordingly.

The critic: To avoid the actor with low Q-value being occasionally selected, a critic network is defined to refine the actor. The critic network will evaluate all actions in the expanded action space, and the action that provides the maximum Q value will be chosen as a_t , i.e.,

$$a_t = \arg\max_{a_j \in \mathcal{A}_K} Q(s_t, a_j). \tag{2.20}$$

2.2.2.3 Single-Agent Actor-Critic Framework

In this subsection, we present the details of the single-agent actor-critic framework for the centralized caching system, and introduce the update of the two networks.

The actor: The actor network is defined as a function parameterized by θ^μ, mapping the state $\mathcal{S}$ from the state space to the action space $\mathcal{A}$. The mapping provides a proto-actor $\hat{a}$ in $\mathcal{A}$ for a given state under the current parameter. Here, we scale the proto-actor to make sure $\hat{a}$ is a valid action i.e., $\hat{a} \in \mathcal{A}$:

$$\mu(s|\theta^\mu) : \mathcal{S} \rightarrow \mathcal{A} \tag{2.21}$$

$$\mu(s|\theta^{\mu}) = \hat{a}. \tag{2.22}$$

The critic: The critic is employed as a refining network, and the deterministic target policy is described below:

$$Q(s_t, a_j|\theta^Q) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_j) + \gamma Q(s_{t+1}, a_{t+1}|\theta^Q)] \tag{2.23}$$

where θ^Q stands for the parameters of the critic network, and $\gamma \in (0, 1]$ is the discount factor which weighs the future accumulative reward $Q(s_{t+1}, a_{t+1}|\theta^Q)$. Here, the critic takes both the current state s_t and the next state s_{t+1} as its input to calculate the Q value for each action in $\mathcal{A}_K$. Then, the action that provides the maximum Q value will be chosen as a_t, i.e.,

$$a_t = \arg\max_{a_j \in \mathcal{A}_K} Q(s_t, a_j|\theta^Q). \tag{2.24}$$

Update: To update the parameters of actor and critic, we replay a minibatch of samples randomly selected from the previous transition, with a minibatch size $N_{\mathcal{B}}$. Therefore, the actor policy is updated using deep deterministic policy gradient, which is given as

$$\nabla_{\theta\mu} J \approx \frac{1}{N_{\mathcal{B}}} \sum_i \nabla_a Q(s, a|\mu^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta\mu} \mu(s|\theta^{\mu})|_{s_i} \tag{2.25}$$

and the critic is updated by minimizing the loss:

$$L = \frac{1}{N_{\mathcal{B}}} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{2.26}$$

where $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.

Workflow: In this part, we introduce the workflow of the proposed framework. At the beginning of each epoch t, the agent observes the state s_t from the environment. Then, the proto-actor obtained by the actor network based on the current policy will

be passed to KNN, and expanded action set will be evaluated by the critic network. Then, an ϵ-greedy policy is applied at selecting the action a_t. This policy can force the agent to explore more possible actions. After the chosen action is executed in the environment, the transition (s_i, a_i, r_i, s_{i+1}) will be stored to the memory $\mathcal{M}$ at the end of this epoch. Next, a minibatch with size $N_{\mathcal{B}}$ will be randomly sampled from the memory $\mathcal{M}$ and replayed to update the actor and critic networks. The complete process is presented in Algorithm 2 below.

2.2.3 Decentralized Edge Caching in Multi-Cell Networks: System Model and Problem Formulation

2.2.3.1 System Model

The decentralized caching system considered in this section is depicted in Fig. 2.5. The system consists of a cloud data center and N cache-enabled base stations. Similar to the centralized content server, each base station in this decentralized content caching system also has a fixed cache capacity C and is able to serve the users from the cache when the requested contents are available locally. For the contents not cached locally, a request is generated by the base station to retrieve the content from the cloud data center. Here, we assume that the cloud data center has sufficient storage space to have all content files, and all base stations can connect with the cloud data center. As shown in the figure, each base station covers a fixed cellular region described by a circle with the corresponding base station at the center, and the radii of the cells are fixed and all users in the cell can access the corresponding base station. There are U users randomly distributed in the system, and they are located in at least one cellular region covered by a base station to ensure service. Each base station receives requests from all users in its cellular region simultaneously, and based on the requests, the base station will learn users' preferences for contents and make

Algorithm 2 Single-Agent Actor-Critic Algorithm for Content Caching

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \longleftarrow \theta^Q, \theta^{\mu'} \longleftarrow \theta^\mu$
Initialize replay buffer $\mathcal{M}$ with capacity of $N_\mathcal{M}$
Initialize features space $\mathcal{F}$
for $t = 1, T$ **do**
    The base station receive a request R_t
    if Requested content is already cached **then**
        Update cache hit rate and end epoch;
    else
        if Cache storage is not full **then**
            Cache the currently requested content
            Update cache state and cache hit rate
            End epoch;
        end if
        Receive observation state s_t
        Actor: Receive proto-ation from actor network $\hat{a}_t = \mu(s_t|\theta^\mu)$.
        KNN: Retrieve k approximately closest actions $\mathcal{A}_K = g_K(\hat{a}_t)$
        Critic: Select action $a_t = \arg\max_{a_j \in \mathcal{A}_K} Q(s_t, a_j|\theta^Q)$ according to the current policy.
        Execute action a_t, and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{M}$
        Sample a random mini batch of $N_\mathcal{B}$ transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{M}$
        Set target $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N_\mathcal{B}} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
    $\nabla_{\theta\mu} J \approx \frac{1}{N_\mathcal{B}} \sum_i \nabla_a Q(s, a|\mu^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta\mu} \mu(s|\theta^\mu)|_{s_i}$
        Update the target networks with $\tau \ll 1$:
$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \longleftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
        Update the cache state
        Update features space $\mathcal{F}$
        Update cache hit rate
    end if
end for

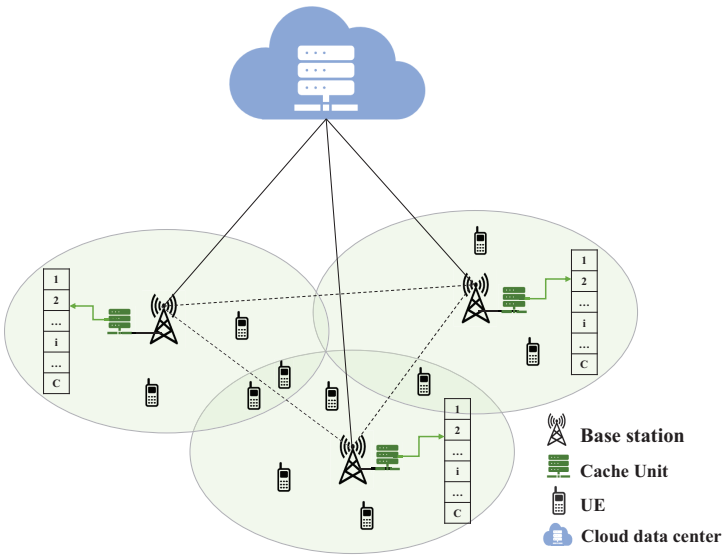


Figure 2.5: System model of a decentralized caching system.

caching decisions.

We assume that in a given time slot, the users' locations do not change and those located in the overlapped regions can be served by any one of the corresponding base stations. Users have their own preferences for contents, and in each time slot, each user can request only one content. Here, we denote the total number of contents as M, and use the content ID to denote the requests for the corresponding content. In each operation cycle, users request contents based on their own preferences. The requests are sent to all base stations that can connect with the user, and, for instance, when delay is the performance metric, the base station that provides the minimum transmission delay will finally transmit the requested content file to the user. In the meantime, all base stations will update their caches to improve the cache hit rate or minimize the average transmission delay based on the users' requests.

The base stations will compete with each other to get the chance to transmit and also cooperate with each other to reduce the overall transmission delay. To realize this framework, we propose a Wolpertinger architecture based multi-agent framework. In

this framework, there are N actor networks and one centralized critic network. We consider each base station as an agent that adopts one of the actor networks to seek its own caching policy. And we assume there are control channels that allow the base stations to send the caching state and data traffic parameters to the cloud date center, so that the cloud data center can act as the centralized critic to evaluate the overall caching state[2]. Similar to the centralized content server, in each operation cycle, the decentralized agent can either keep the cache state the same or replace unpopular contents with the popular ones. Note that there can be more than one request arriving at a base station at the same time, and for different contents, the agent needs to jointly decide which cached content will be deleted and which content requested by which user will be cached. For each agent i, we define the action space as $\mathcal{A}_i$, and let $\mathcal{A}_i = \{a_0, a_1, ..., a_{\mathcal{D}_i}\}$, where a_ν denotes a valid action. In our case, a_0 indicates that the current cache state is unchanged. For $\nu = \{1, 2, ..., \mathcal{D}_i\}$, we define $\mathcal{D}_i = \binom{C_i}{1}\binom{L_i}{1}$, where C_i is the total number of files that can be stored at base station i, and L_i is the number of users that can connect with the base station i. So each a_ν stands for a possible combination to replace one of C_i cached contents with one of L_i currently requested contents. In every time slot, each agent must select its own action from the corresponding action space $\mathcal{A}_i$ and execute.

As seen in the descriptions above, decentralized caching in a multi-cell network is more general and challenging than the centralized caching with a single base station analyzed in the previous sections. In the decentralized caching framework, base stations receive multiple file requests at a given time and differentiate which users generate the requests, each user generates the file requests according to its unique preference, user location and channel conditions are taken into account if the objective is to reduce the transmission delay, and overall multi-agent reinforcement learning is employed. A challenge in this setting is that the action space and observation space

[2]We note that the centralized critic can also be placed at a node or controller (other than the cloud data center) that is connected to the BSs.

grow as the number of users in the coverage of the base stations increases.

2.2.3.2 Problem Formulation

2.2.3.2.1 Cache Hit Rate For the decentralized caching system, the cache hit rate is calculated from the perspective of users in each time slot t. In particular, the cache hit rate is defined as

$$P_{hit}^{d} = \frac{\sum_{j=1}^{U} \xi_j}{U} \tag{2.27}$$

where U is the total number of users, and ξ_i is an indicator defined as

$$\xi_j = \begin{cases} 1 & \text{if user } j \text{ is served by a base station} \\ & \text{(i.e., from the base station's local cache)} \\ 0 & \text{if user } j \text{ is served by the upper level server.} \end{cases}$$

And the maximization of the cache hit rate over the caching decisions is expressed as

$$\textbf{P2:} \quad \underset{\mathbf{\Phi}}{\text{Maximize}} \quad P_{hit} \tag{2.28}$$

$$\text{Subject to} \quad \sum_{f=1}^{M} \phi_{i,f} \leq C_i \quad \forall i \in \{1, \ldots, N\} \tag{2.29}$$

where $\mathbf{\Phi}$ is an $N \times M$ matrix which records the caching states of the N base stations, and each element $\phi_{i,f}$ in the caching state matrix is an indicator to show if the file is cached at base station i:

$$\phi_{i,f} = \begin{cases} 1 & \text{if the file } f \text{ is cached at the base station } i \\ 0 & \text{if the file } f \text{ is not cached at the base station } i \end{cases} . \tag{2.30}$$

2.2.3.2.2 Transmission Delay For the decentralized caching system, we evaluate the caching policy in terms of transmission delay as well. The transmission delay is defined as the number of time frames needed to transmit a content file, and can be expressed as

$$T = \min \left\{ \tilde{t} : F \leq \sum_{\kappa=1}^{\tilde{t}} T_0 \mathbb{C}[\kappa] \right\} \tag{2.31}$$

where F is the size of the content file to be transmitted. T_0 stands for the duration of each time frame, and $\mathbb{C}[\kappa]$ is the instantaneous channel capacity in the κth time frame. And the channel capacity $\mathbb{C}[\kappa]$ is expressed as

$$\mathbb{C}[\kappa] = B \log_2 \left(1 + \frac{P_t}{BN_0} z_\kappa \right) \quad \text{bits/s} \tag{2.32}$$

where P_t is the transmission power, B is the channel bandwidth, N_0 is the (one-sided) noise power spectral density, and z_κ is the channel gain in the κth time frame. In the system, there are two types of transmitters: the cloud data center and the base stations. We assume that all transmitters transmit at their maximum power level to maximize the transmission rate. The transmission power is denoted as

$$P_t = \begin{cases} P_c & \text{if the transmitter is the cloud data center} \\ P_i & \text{if the transmitter is the } i^{\text{th}} \text{ base station} \end{cases} . \tag{2.33}$$

So, if user j requests a content, which is not cached at any base station that can connect with the user, the content file will be first transmitted from the cloud data center to the base station $\hat{i}$, which is the closest base station to the user j, and then from the base station $\hat{i}$ to user j. Thus, the minimum transmission delay $\hat{D}_j$ in the

case of a missing file in the cache can be expressed as

$$\hat{D}_j = T_{c,\hat{i}} + T_{\hat{i},j} \tag{2.34}$$

where $T_{c,\hat{i}}$ stands for the transmission delay form the cloud data center to the base station $\hat{i}$, and $T_{\hat{i},j}$ is the transmission delay from the base station $\hat{i}$ to the user j.

However, if the requested file is cached at a base station i, which can connect to user j, the transmission delay D_j in the case of having a hit in the cache can be expressed as

$$D_j = T_{i,j}. \tag{2.35}$$

Problem Formulation: In the previous section, we have described the transmission delay for both cases of missing and hitting in the cache. In this section, we formulate the caching problem. First, we define the transmission delay reduction ΔD_j as

$$\Delta D_j = \hat{D}_j - D_j. \tag{2.36}$$

Now, the average transmission delay reduction in an operation cycle is

$$\Delta D = \frac{1}{U}\sum_{j=1}^{U} \Delta D_j \tag{2.37}$$

$$= \frac{1}{U}\sum_{j=1}^{U} (\hat{D}_j - D_j) \tag{2.38}$$

$$= \frac{1}{U}\sum_{j=1}^{U} (T_{c,\hat{i}} + T_{\hat{i},j} - T_{i,j}) \tag{2.39}$$

where U is again the total number of users. In this chapter, our goal is to maximize the average transmission delay reduction, and the caching problem is formulated as follows:

$$\textbf{P3:} \quad \underset{\mathbf{\Phi}}{\text{Maximize}} \quad \Delta D \tag{2.40}$$

$$\text{Subject to} \quad \xi_{i,j} = 1 \quad \exists i \ \forall j \tag{2.41}$$

$$\sum_{f=1}^{M} \phi_{i,f} \leq C_i, \forall i \in \{1, \ldots, N\} \tag{2.42}$$

where $\mathbf{\Phi}$ is again the $N \times M$ matrix which records the caching states of the N base stations, and each element $\phi_{i,f}$ in the caching state matrix is an indicator, showing if the file is cached at base station i:

$$\phi_{i,f} = \begin{cases} 1 & \text{if the file } f \text{ is cached at the base station } i \\ 0 & \text{if the file } f \text{ is not cached at the base station } i \end{cases} . \tag{2.43}$$

C_i is the maximum number of files that can be stored at base station i. And $\xi_{i,j}$ is an indicator describing if user j is in the area covered by base station i:

$$\xi_{i,j} = \begin{cases} 1 & \text{if user } j \text{ can connect to base station } i \\ 0 & \text{if user } j \text{ cannot connect to base station } i \end{cases} . \tag{2.44}$$

2.2.4 Deep Actor-Critic Framework for Decentralized Edge Caching

In this section, we introduce the multi-agent deep reinforcement learning framework for the decentralized edge caching problem. In this framework, there will be multiple DRL agents that can make independent caching decisions based on their own observations, and each agent will also incorporate the Wolpertinger architecture as introduced in subsection 2.2.2.2.

2.2.4.1 Related Definitions

2.2.4.1.1 Agents' Observation and State Space Allowing the agents to make their own caching decisions and cooperate with each other, the decentralized caching framework is proposed as a centralized critic network together with a decentralized actor network. Therefore, the agent will feed the actor network with their own observations and feed the critic network with the complete state space. This multi-agent actor critic framework is based on a partially observable Markov decision process. Each agent i, $i = 1, 2, ..., N$, can only observe the requests arriving at itself, and select its own action only based on the observation o_i. In the environment, the agent i can observe the contents' features through its local request history. And for the centralized critic, the state space is defined as $\mathtt{x} = \{o_1, o_2, ..., o_N\}$. Similar to the centralized caching agent's observation, the observation o_i of each decentralized caching agent can be denoted as $o_i = \{\mathcal{F}_s; \mathcal{F}_m; \mathcal{F}_l\}$, where $\{\mathcal{F}_s; \mathcal{F}_m; \mathcal{F}_l\}$ is the feature space as introduced in subsection 2.2.2.1.1.

2.2.4.1.2 Action Space Similar to the action space in the centralized edge caching framework, introduced in subsection 2.2.2.1.2, each agent in the decentralized caching framework can either keep the current caching state unchanged or only make one replacement. However, in the decentralized caching system, different numbers of file requests arrive at different base stations, since each base station serves potentially different number of users. Therefore, the action space size of each decentralized agent is also different depending on the number of users in the base stations' service regions.

2.2.4.1.3 Reward The decentralized caching agents are designed to solve the optimization problems **P2** or **P3**, depending on whether cache hit rate or delay reduction is the ultimate goal.

To solve problem **P2**, in operation cycle t, after the agents update their caches

according to the selected actions, the cache hit rate for requests in the next operation cycle $t+1$ will be received as the reward within the multi-agent framework. So we define the reward in the tth operation cycle as

$$r_t = P_{hit}^{t+1}, \tag{2.45}$$

and to solve the problem **P3**, the reward for each iteration is defined as

$$r_t = \Delta D^{t+1}. \tag{2.46}$$

2.2.4.2 Multi-Agent Actor-Critic Framework

Now, we introduce the decentralized caching framework in detail. Specifically, we have multi-agent actor-critic framework based on the partially observable Markov decision processes with N agents, where the critic network $V(\mathtt{x})$ and N actors $\pi_{\theta_i}(o_i)$, $i = i, 2, ..., N$, are parameterized by $\theta = \{\theta_c, \theta_1, \theta_2, ..., \theta_N\}$.

Actor: The actor network is defined as a function to seek a caching policy $\pi = \{\pi_1, \pi_2, ..., \pi_N\}$, which can map the observation of the agent to a valid action chosen from the action space $\mathcal{A}$. In each time slot, agent i will select an action a_i based on its own observation o_i and policy π_i:

$$a_i = \pi_i(o_i). \tag{2.47}$$

Critic: The critic is employed to estimate the value function $V(\mathtt{x})$, where $\mathtt{x}$ stands for the observation of all agents, $\mathtt{x} = \{o_1, o_2, ..., o_N\}$. At time instant t, after the actions $a_t = \{a_{1,t}, ..., a_{N,t}\}$ are chosen by the actor networks, the agents will execute the actions in the environment and send the current observation $\mathtt{x}_t$ along with the feedback from the environment to the critic. The feedback includes the reward r_t and the next time instant observation $\mathtt{x}_{t+1}$. Then, the critic can calculate the TD

(Temporal Difference) error:

$$\delta^{\pi_\theta} = r_t + \gamma V(\mathtt{x}_{t+1}) - V(\mathtt{x}_t) \tag{2.48}$$

where $\gamma \in (0, 1)$ is the discount factor.

Update: Instead of using DDPG to train the neural networks as we present in the centralized caching framework, the decentralized caching agents are updated using TD error since this approach involves relatively lower computational complexity, helping the decentralized caching agents more easily meet real-time operation requirements.

Specifically, the critic is updated by minimizing the least squares temporal difference (LSTD):

$$V^* = \arg\min_V (\delta^{\pi_\theta})^2 \tag{2.49}$$

where V^* denotes the optimal value function.

The actor i is updated by policy gradient. Here, we use TD error to compute the policy gradient:

$$\nabla_{\theta_i} J(\theta_i) = E_{\pi_{\theta_i}}[\nabla_{\theta_i} \log \pi_{\theta_i}(o_i, a_i)\delta^{\pi_\theta}] \tag{2.50}$$

where $\pi_{\theta_i}(o_i, a_i)$ denotes the score of action a_i under the current policy. Then the weighted difference of parameters in the actor i can be denoted as $\Delta\theta_i = \alpha\nabla_{\theta_i} \log \pi_{\theta_i}(o_i, a_i)\delta^{\pi_\theta}$, where $\alpha \in (0, 1)$ is the learning rate. And the actor network i can be updated using the gradient decent method:

$$\theta_i \longleftarrow \theta_i + \alpha\nabla_{\theta_i} \log \pi_{\theta_i}(o_i, a_i)\delta^{\pi_\theta} \tag{2.51}$$

Workflow: To make the multi-agent system meet real-time operation requirements, we abandon the memory presented in the centralized agent. Consequently, only the current transition will be used in updating the networks. In each iteration, agent i, $i = 1, 2, \ldots, N$, observes the features of users' requests and updates its own

cache space. Each actor network will propose a proto-actor $\hat{a}_{i,t}$, and each of these proto-actors will be expanded to a K-action set respectively by the KNN, and the expanded action set is denoted as $\mathcal{A}_{i_K}$. Then, one action will be chosen from the expanded action set of each agent, and the chosen actions will be combined to form a new action set to be evaluated by the critic network. Each possible action set combination can be expressed as $\mathtt{A}_h = (\hat{a}_{1_K}, \hat{a}_{2_K}, ..., \hat{a}_{N_K})$, where we denote $\hat{a}_{i_K}$ as an element chosen from the ith agent's expanded action set. Therefore, there will overall be K^N possible action combinations considering all agents, and we can also index the possible action combination $\mathtt{A}_h$ with $h = 1, 2, \ldots, K^N$. The critic network will evaluate all K^N possible combinations of action sets. For example, if there are 2 agents, and each agent's proto-actor is expanded to an action set with 3 actions, the critic network will need to evaluate all 3^2 possible combinations. Following this, the action combination that provides the maximum state value will be executed in the environment finally. Then, the critic and actor networks will update their parameters accordingly.

The complete process is presented in Algorithm 3 below.

2.2.5 Numerical Results

To analyze the performance of our algorithms, comparisons are made between our proposed deep reinforcement learning framework and the following caching algorithms:

- **Least Recently Used (LRU) [114]:** In this policy, the system keeps track of the most recent requests for every cached content. And when the cache storage is full, the cached content, which is requested least recently, will be replaced by the new content.

- **Least Frequently Used (LFU) [115]:** In this policy, the system keeps track of the number of requests for every cached content. And when the cache storage

Algorithm 3 Multi-Agent Actor-Critic Algorithm for Content Caching

Initialize critic network $V(\mathtt{x})$ and actor $\pi_{\theta_i}(o_i)$, parameterized by $\theta = \{\theta_c, \theta_1, \theta_2, ..., \theta_N\}$.
Receive initial state $\mathtt{x} = \{o_1, o_2, ..., o_N\}$.
for $t = 1, T$ **do**
    The base station receive users' requests $Req_t = \{req_{1,t}, req_{2,t}, ..., req_{U,t}\}$.
    Extract observation at time t for each agent, and $\mathtt{x}_t = \{o_{1,t}, o_{2,t}, ..., o_{N,t}\}$
    for $i = 1, N$ **do**
        The agent i selects proto-actor $\hat{a}_{i,t} = \pi_{\theta_i}(o_{i,t})$ w.r.t. the current policy
        Expand the proto-actor $\hat{a}_{i,t}$ to a action set with k actions $\mathcal{A}_{i,K}$ via KNN.
    end for
    Extract all possible action set combinations $\mathtt{A}_h$, $h = 1, 2, \ldots, k^N$.
    Critic network calculate the state value V with all possible action set combinations.
    Find the action set combination that can provide maximum state value, set $a_t = \arg\max_{\hat{a}_{i,k} \in \mathcal{A}_{i,k}} V(\mathtt{x}_{t+1}|\mathtt{A})$
    Execute actions a_t to update the cache state of each base station
    Observe reward r_t and new state $\mathtt{x}_{t+1}$
    Critic calculates the TD error based on the current parameter: $\delta^{\pi_\theta} = r_t + \gamma V(\mathtt{x}_{t+1}) - V(\mathtt{x}_t)$
    Update the critic parameter θ_c by minimizing the loss: $\mathcal{L}(\theta) = (\delta^{\pi_\theta})^2$
    for agent $i = 1$ to N **do**
        Update the actor policy by maximizing the action value: $\Delta\theta_i = \alpha \nabla_{\theta_i} \log \pi_{\theta_i}(o_{i,t}, a_i)\delta^{\pi_\theta}$, $\alpha \in (0, 1)$.
    end for
    Update features space $\mathcal{F}$
end for

is full, the cached content, which is requested the least many times, will be replaced by the new content.

- **First In First Out (FIFO) [116]:** In this policy, the system, for each cached content, records the time when the content is cached. And when the cache storage is full, the cached content, which was stored earliest, will be replaced by the new content.

2.2.5.1 Numerical Results for Centralized Edge Caching

2.2.5.1.1 Neural Network In our implementation, the actor network has two hidden layers of fully-connected units with 256 and 128 neurons, respectively; and the critic network has two hidden layers of fully-connected units with 64 and 32 neurons, respectively. The memory capacity $N_{\mathcal{M}}$ is set as $N_{\mathcal{M}} = 10000$, and the mini batch size is set as $N_{\mathcal{B}} = 100$. The discount factor γ introduced in (2.23) is set as 0.9. In the KNN component of the algorithm, we conduct the experiments with the number of neighbors as $K_1 = \lceil 0.15C \rceil$ and $K_2 = \lceil 0.05C \rceil$, where C is the cache capacity.

2.2.5.1.2 File/Content Request Generation In our simulations, the raw data of users' requests is generated according to the Zipf distribution

$$f(k; \beta, M) = \frac{1/k^{\beta}}{\sum_{m=1}^{M}(1/m^{\beta})} \tag{2.52}$$

where k is the rank of the files. For each experiment, we collect 10000 requests as the testing data. The settings of Zipf exponent β and total number of files M are specified before each experiment.

2.2.5.1.3 Feature Extraction From the raw data of content requests, we extract the feature $\mathcal{F}$ and use it as the input state of the network. Here, as features, we

consider the number of requests for a file within the most recent 10, 100, and 1000 requests.

Cache Hit Rate

Fig. 2.6 shows the overall cache hit rate (plotted in percentage) achieved by the proposed framework and the other caching algorithms introduced above. In this figure, we set the total number of files as M as 5000 and the Zipf exponent as β at 1.3 and we vary the cache capacity. However, instead of directly using the cache capacity C, we consider the cache ratio $\sigma = \frac{C}{M}$ (where M is the total number of content files that can be requested by the users), so that we can analyze the impact of the cache capacity normalized by the potential data traffic flow into this system. We observe that the proposed framework with $K_1 = \lceil 0.15C \rceil$ outperforms the other strategies and provides the highest cache rates for all cache capacity values, while the performance of the proposed framework with $K_2 = \lceil 0.05C \rceil$ is relatively close to the LFU policy. This observation demonstrates that increasing the number of neighbors selected in the KNN stage can help improve the decision policy because the value of K dictates how many actions can be learned in one iteration. We also notice that when the cache capacity is small, the performance of LFU is very close to our proposed framework. As the cache capacity increases, the gap between proposed framework with K_1 and other three caching algorithms increases at first, and then gradually decreases. At cache capacity $C = 500$, the cache hit rate of all four algorithms are close to each other at around 80% hit rate. At this point, the cache hit rates achieved by different policies tend to converge because the cache capacity is high enough to store all popular contents. From this point on, increasing the cache capacity will not improve the cache hit rate significantly, and the cache hit rate is now essentially limited by the distribution of the content popularity.

In Fig. 2.7, we study the cache hit rate as a function of the Zipf exponent β. In this

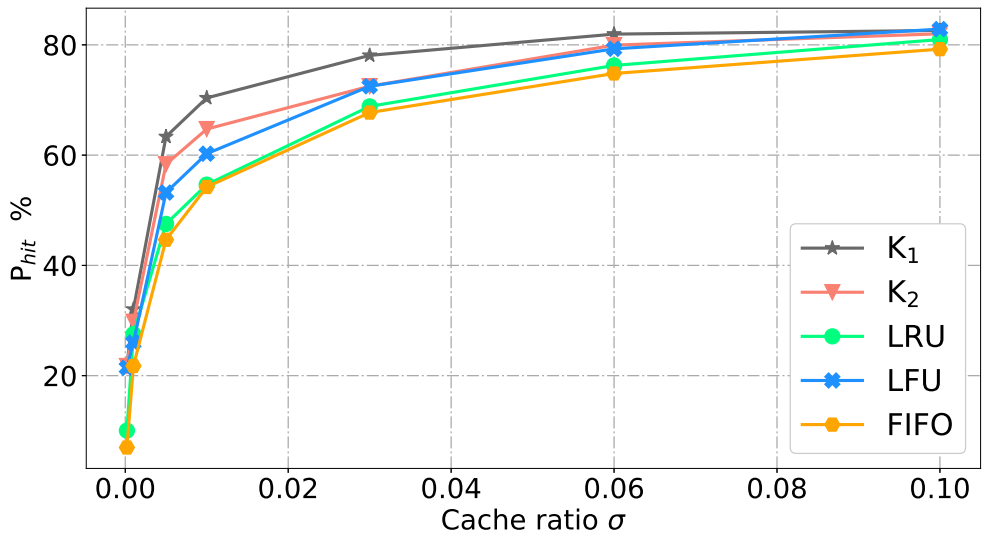


Figure 2.6: Cache hit rate vs. cache ratio $\sigma = \frac{C}{M}$. We vary the cache capacity as $C = 1, 5, 25, 50, 150, 300, 500$.

experiment, we set the total number of files as 1000, fix the cache capacity at 100, and vary the Zipf exponent β. Again, we test the proposed framework with two different K values, i.e., $K_1 = \lceil 0.15C \rceil$ and $K_2 = \lceil 0.05C \rceil$, and compare the cache hit rate with that achieved by the non-learning based caching policies. As β increases, cache hit rates achieved by all caching policies grow. This is due to the fact that with larger β, there are fewer files with larger request probabilities and therefore the popularity of the files is skewed. Consequently, caching these more popular files leads to an increase in the cache hit rates. And with the same cache capacity, chances are higher that the agent can cache all the highly popular files (the number of which has decreased). We also notice that the slopes of all the curves first increase and then decrease. This is because initially when the number of popular files gets small, all caching policies start storing the most popular files, and the larger the β, the smaller the influence of less popular files is. However, we eventually experience diminishing returns as β is further increased. In addition, the gap between the curves corresponding to K_1 and K_2 also increases as β gets larger. This verifies that adopting a larger value of K can

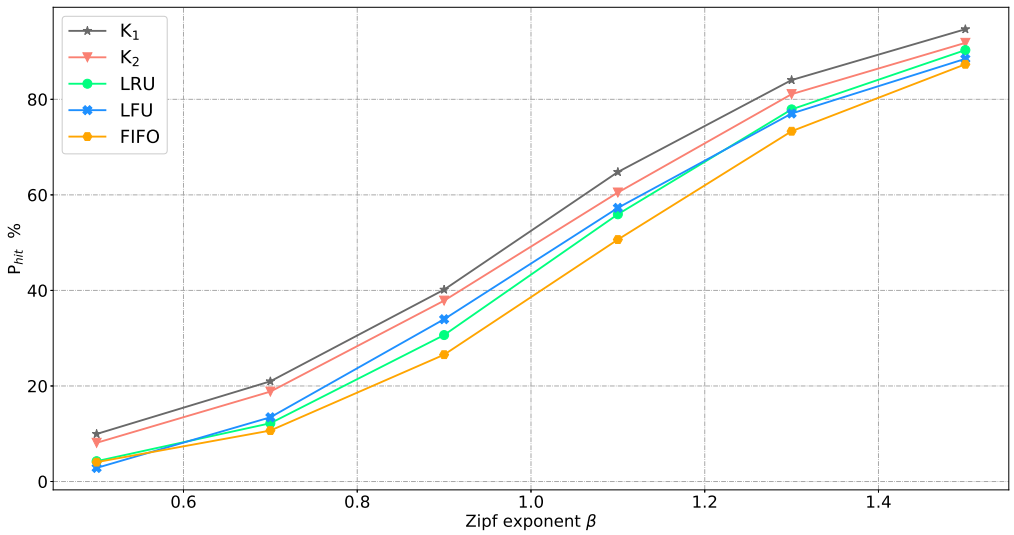


Figure 2.7: Cache hit rate vs. Zipf exponent β. We vary the Zipf exponent as $\beta = 0.5, 0.7, 0.9, 1.1, 1.3, 1.5$.

be helpful for the agent in exploring different action strategies.

In Fig. 2.8, we plot the cache hit rate as a function of the total numbers of files M. In this experiment, we set the cache capacity as 100, fix the Zipf exponent at 1.4, and vary the value of M. Again the proposed framework is tested with two different values of K and the performance is compared with the LRU, LFU, FIFO caching policies. As the number of files increases, the cache hit rate achieved by all policies tend to decrease. This is because when the cache capacity is fixed, increasing the number of files leads to smaller cache ratio. And since the Zipf exponent β is also fixed, the number of popular files is increased. In this figure, we still observe the proposed framework outperforming for all values of M. Besides, observing the difference between the cache hit rate achieved at $M = 1000$ and that achieved at $M = 5000$ with the same caching policy, we find that the proposed framework has better ability in handling cases with larger M (or smaller cache ratio), making it more competitive in practical settings.

In Fig. 2.9, we plot the long term average cache hit rate $\overline{P}_{\text{hit}}(T) = \sum_{t=0}^{T} \mathbf{1}(R_t)$ over time. In this experiment, the number of files is fixed at 1000, and the popularities of

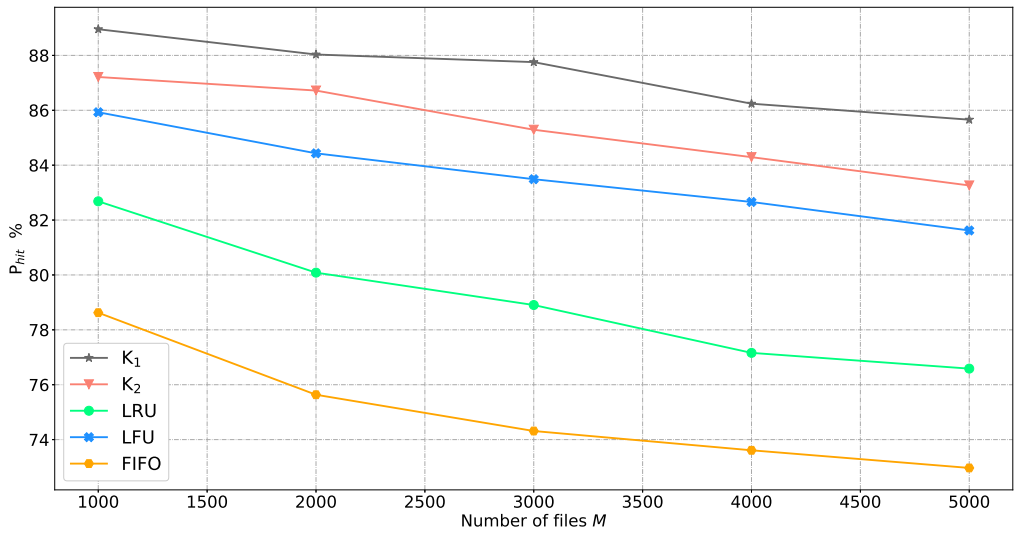


Figure 2.8: Cache hit rate vs. number of files M. We vary the number of files as $M = 1000, 2000, 3000, 4000, 5000$.

the files change every 10000 time slots. Each time the popularity changes, the ranks of the files will vary randomly and the Zipf exponent is randomly generated in the range $[1.0, 1.3]$. Note that the change points and the popularity parameters of the files (both ranks and Zipf exponent) are unknown to the reinforcement learning agent. With the long-term average cache hit rate, we evaluate the ability of the caching policies to maintain a stable performance in the changing environment. We can observe that the proposed framework outperforms all the time and the performance is stable. For the LRU and FIFO caching policies, though the curves are flat and smooth, the cache hit rate is not competitive. And for the LFU policy, the cache hit rate drops quickly after the first change point because of the frequency pollution. With this experiment, we conclude that the proposed framework is more suitable for applications that require high long-term performance and stability.

2.2.5.2 Numerical Results for Decentralized Edge Caching

2.2.5.2.1 Environment Settings As shown in Fig. 2.10, in the experiments, we consider a system with 5 base stations and 30 users randomly distributed in the area,

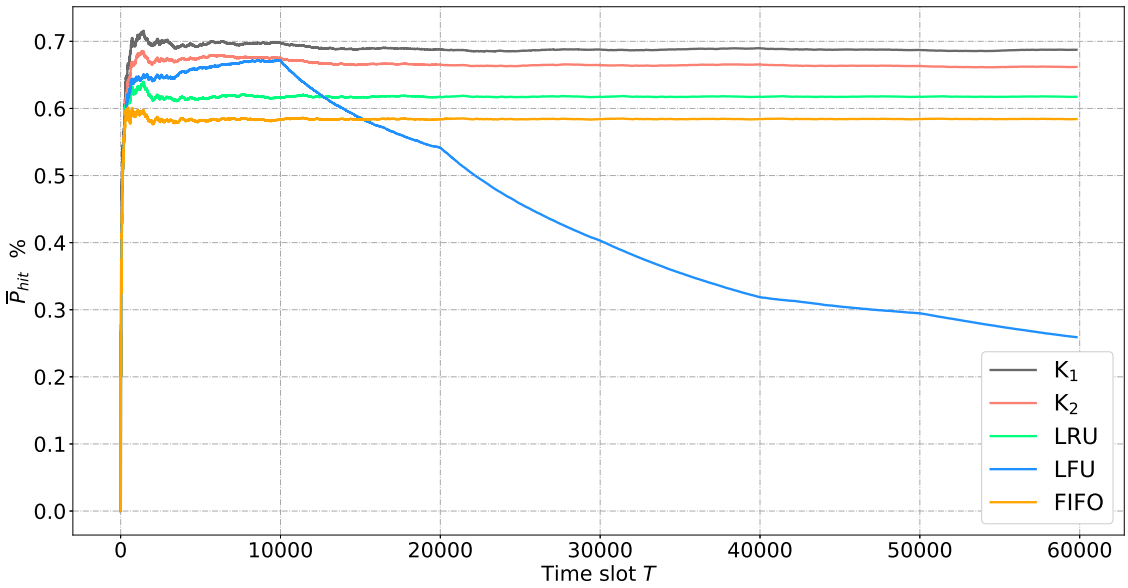


Figure 2.9: Long term average cache hit rate as the popularity of files change over time.

each covered by at least one of the base stations. The cell radius is set as $R = 2.2$km, and the transmission power of all base stations is set as $P_i = 16.9$dB, $i = 1, 2, ..., 5$. The transmission power of the cloud data center is set as $P_c = 20$dB. As assumed, the content files are split into units of the same size, and the size of each unit is set as 96 bits. And we assume Rayleigh fading with path loss $\mathbb{E}\{z\} = d^{-4}$, where d is the distance between the transmitter and receiver.

2.2.5.2.2 Neural Network In the implementation, each actor network has three layers, and the first and hidden layers have 200 and 600 neurons, respectively, as shown in Table 2.3. And for each actor network, the number of neurons in the output layer depends on the size of the action space $\mathcal{A}_i$. For the critic network, the number of neurons in each layer is provided in Table 2.3. To ensure that the critic network can learn faster than the actor networks, we set the learning rate of the critic network as 0.001, and the learning rate of each actor network as 0.0005. And we test this framework with the number of neighbors set as $K = 1$ and $K = 2$ in the KNN

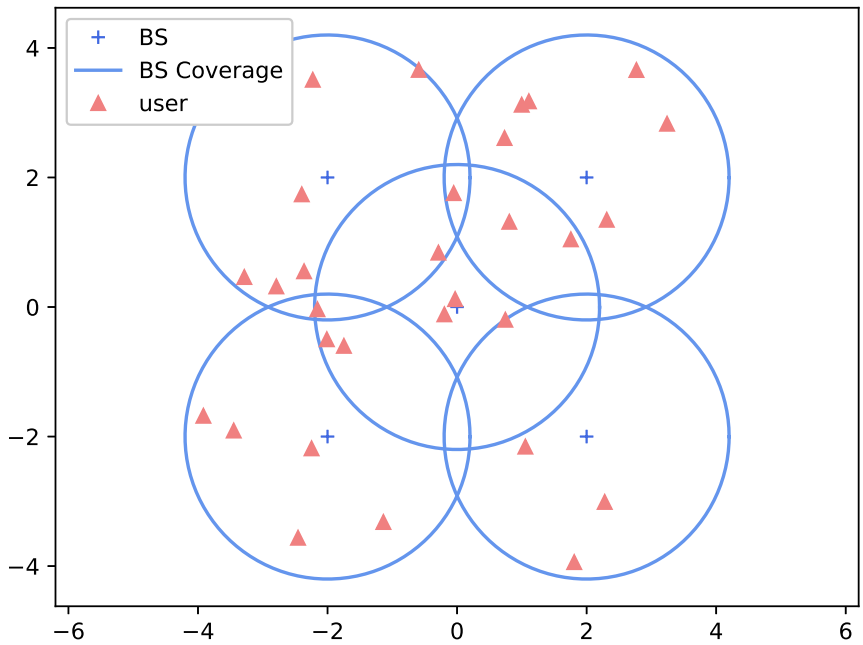


Figure 2.10: Coverage map of a system contains 5 base stations and 30 users

Table 2.3: Architecture of Multi-agent Framework

	Actor Network	Critic Network
Input Layer	200 Neurons	400 Neurons
Hidden Layer	600 Neurons	800 Neurons
Output Layer	$\mathcal{D}_i + 1$ Neurons	1 Neurons
Learning Rate	0.0005	0.001

component of the algorithm. When $K = 2$, the proto-actor of each agent will be expanded to an action set with 2 valid actions, while when $K = 1$, the proto-actor will not be expanded and the Wolpertinger architecture will specialize to the regular actor-critic structure.

2.2.5.2.3 File/Content Request Generation In our simulations, the raw data of users' requests is generated according to the Zipf distribution as shown in (2.52), where the total number of files M is set as 500, and unless state otherwise, the Zipf exponent β is fixed at 1.3 in the study of the cache size and transmission delay. The

rank of the file k is randomly generated for each user so that users' preferences for files can be differentiated. To encourage the base station to cache the files that are popular for more users, the users are randomly divided into 5 groups. It is assumed that the users in the same group will have similar but not exactly the same rank for all files. And the group information will not influence the users' location. It is important to note that we generate the requests with Zipf distribution and also group the users. However, such information is totally unknown to the agents.

2.2.5.2.4 Feature Extraction From the raw data of content requests, we extract the feature $\mathcal{F}$ and use it as the agents' observations of the network. Here, as features, we consider the number of requests for a file within the most recent 10, 100, and 1000 requests.

Cache Hit Rate

In Fig. 2.11, we plot the overall cache hit rate (as a percentage) achieved by the proposed framework and the other caching policies in a multi-cell network. The tendency of the cache hit rate as the cache ratio increases is very similar to that in the case of a single base station as shown in Fig. 2.6. However, in a multi-cell scenario, even the cache hit rate achieved by the regular actor-critic framework (i.e., when $K = 1$, whose curves are labeled as "DRL" in the figures) is always higher than those of the LRU, LFU and FIFO policies, while in the single base station case, when we set the number of neighbors as $K_2 = \lceil 0.05C \rceil$, the cache hit rate achieved by the proposed framework can become slightly lower than that of the LFU policy. This observation indicates the benefits of the agents cooperating with each other to avoid caching the same files so that the limited cache storage can be utilized more effectively. Also, when we increase the number of neighbor to $K = 2$ (whose curves are labeled as "K = 2" in the figures), the gap between the proposed framework

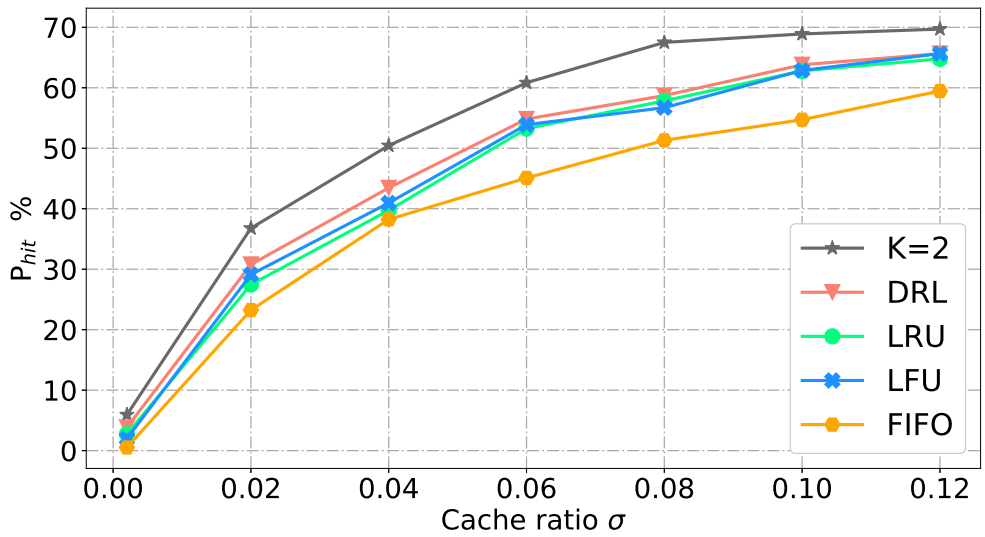


Figure 2.11: Cache hit rate vs. cache capacity. We vary the cache capacity as $C = 1, 10, 20, 30, 40, 50, 60$.

and other policies increases further. This is because when we increase the number of neighbors from $K = 1$ to $K = 2$, the total number of action sets that will be learned by the critic increases from 1 to 2^5, which will provide the critic more information for final action selection. Note that we can continue increasing the value of K, but since the number of action sets will increase exponentially, we need to trade off between the performance and increased computational complexity and runtime.

In Fig. 2.12, we study the relationship between cache hit rate and Zipf exponent β. In this experiment, we fix the cache capacity of all base stations as $C = 40$, and vary the Zipf exponent. As β increases, the cache hit rate of all policies increase because there are now a smaller number of popular files but with higher popularities compared to before when β was smaller. Eventually, for large values of β, performances of different policies tend to converge. This is because with the increasing value of β, the frequencies of popular files being requested is sufficiently high, so that the LFU and LRU policies can always keep these files, and for the proposed learning agents, the features of these files become more distinguishable to learn.

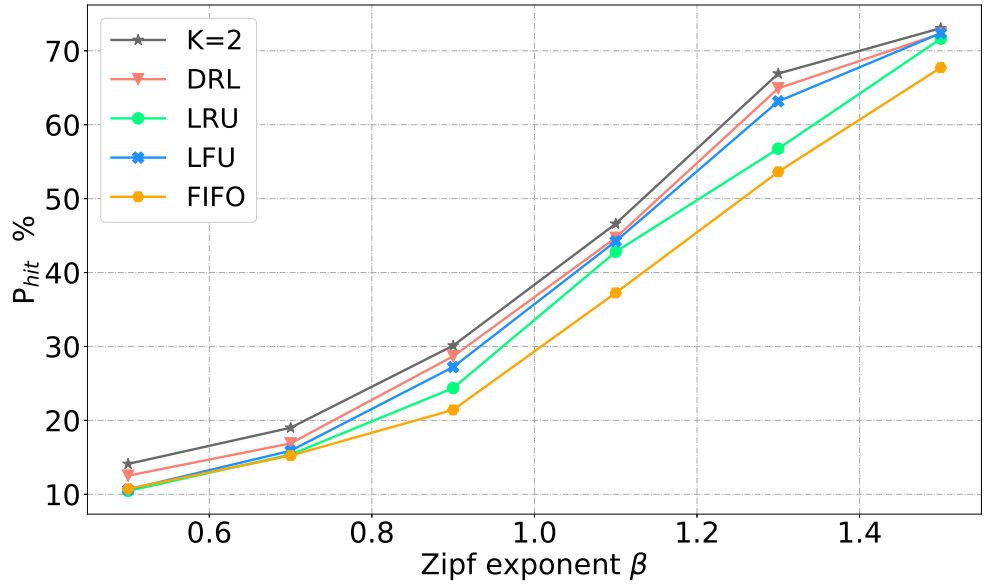


Figure 2.12: Cache hit rate vs. Zipf exponent. We vary the Zipf exponent as $\beta = 0.5, 0.7, 0.9, 1.1, 1.3, 1.5$.

Transmission Delay

In this section, we present the the simulation results addressing the transmission delay. In particular, we evaluate the reduction in transmission delay as a percentage as follows:

$$\eta = \frac{\Delta D}{\frac{1}{U}\sum_{j=1}^{U} \hat{D}_j} \times 100\%. \tag{2.53}$$

Hence, η is the percentage of delay reduction per user in one operation cycle.

To determine the relationship between the transmission delay and cache capacity, in Fig. 2.13, we fix the Zipf exponent at $\beta = 1.3$, and plot the percentage of overall transmission delay reduction η as a function of the cache ratio. It is shown that as the cache ratio σ increases, the reduction in transmission delay achieved by all caching policies first rises quickly because the base stations can cache more files, and then the trend slows down after a certain value of σ. The upward trend starts to slow down because all these caching algorithms are encouraged to cache the most popular files

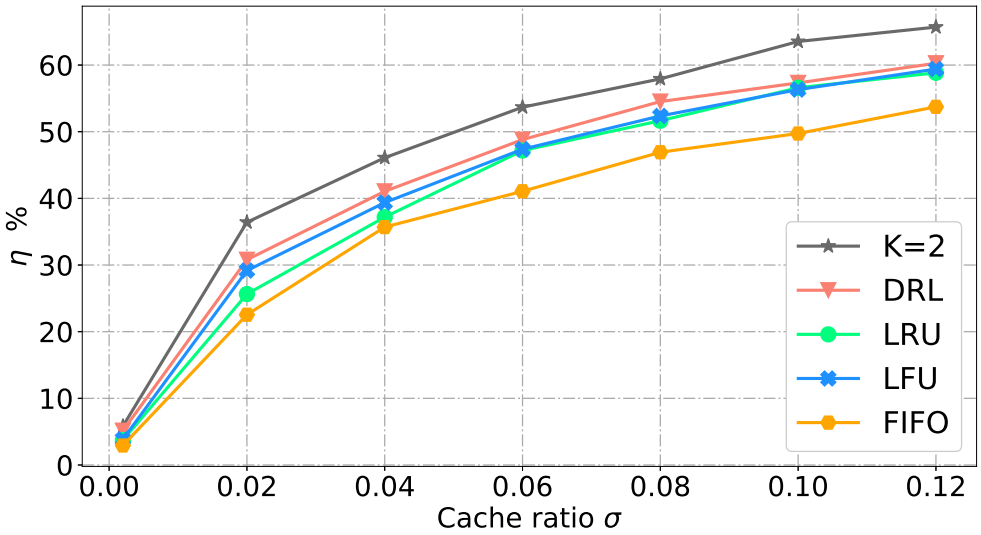


Figure 2.13: Percentage of transmission delay reduction vs. cache capacity. We vary the cache capacity as $C = 1, 10, 20, 30, 40, 50, 60$.

following the statistics they learn. So when the cache ratio grows further and further, the caching agent will start caching the less popular content files. Though more files are cached and transmission delay is further reduced, caching the less popular files at the edge nodes lead to smaller improvements in reducing the transmission delay when compared with the contribution made by caching the most popular files. In other words, when the cache ratio is large enough to cache all of the most popular files, the system does not necessarily have to keep enlarging the cache capacity, considering the price to pay for the storage and the relatively small reduction in transmission delay that will be achieved by storing the less popular files. We also observe that for all values of the cache ratio, the proposed framework achieves better performance for two reasons: First, the proposed framework considers the reduction in the average transmission delay as the reward, so that the caching algorithm does not only focus on finding the most popular files, but also takes into account the users' locations and several less popular files with potentially high delay penalties if not cached; and secondly, the critic network can facilitate the exchange of information among the

base stations so that they can avoid caching the same file to serve the user located in overlapped regions, and in this way, utilize the cache space more efficiently.

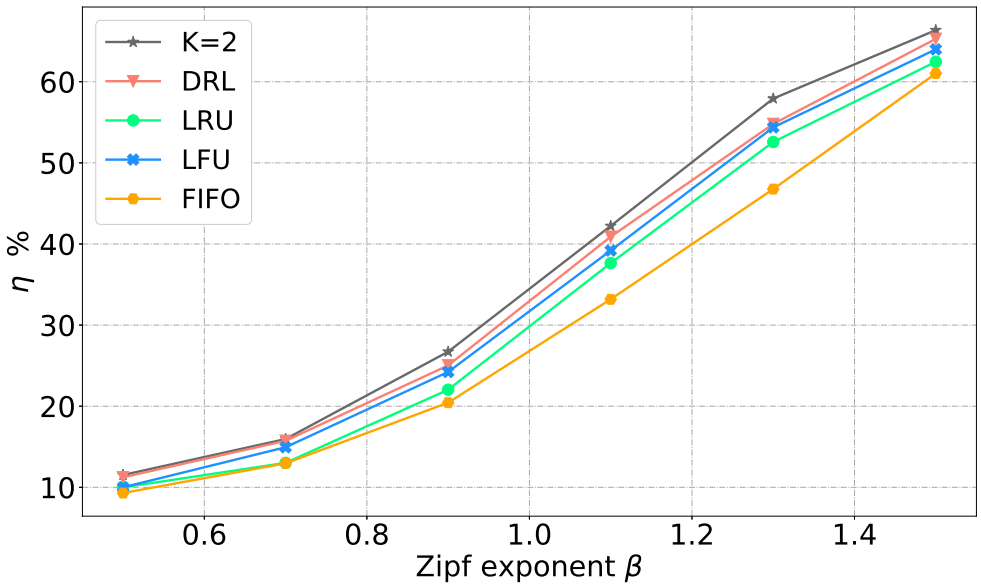


Figure 2.14: Percentage of transmission delay reduction vs. Zipf exponent. We vary the Zipf exponent as $\beta = 0.5, 0.7, 0.9, 1.1, 1.3, 1.5$.

Again, we fix the cache ratio at $\sigma = 0.1$ and demonstrate how the percentage of transmission delay reduction varies as the Zipf exponent β increases. In Fig. 2.14, we observe that when β is small, the gap between the curve with "$K = 2$" and the curve labeled "DRL" (i.e., $K = 1$) is small, implying that when the popularities of the files are close to each other, the actor-critic agent is not able to take advantage of the KNN because the features for all files are relatively similar and therefore it is more difficult to find the most popular files. On the other hand, as β increases, the increasing gap between these two curves points to the advantage of adopting a larger number of neighbors in KNN. And when the value of β approaches 1.5, the actor-critic agents with different K values achieve similar performances again since the features of popular files can be easily distinguished from the non-popular files, and therefore even with smaller number of neighbors, the proposed framework can

learn it well.

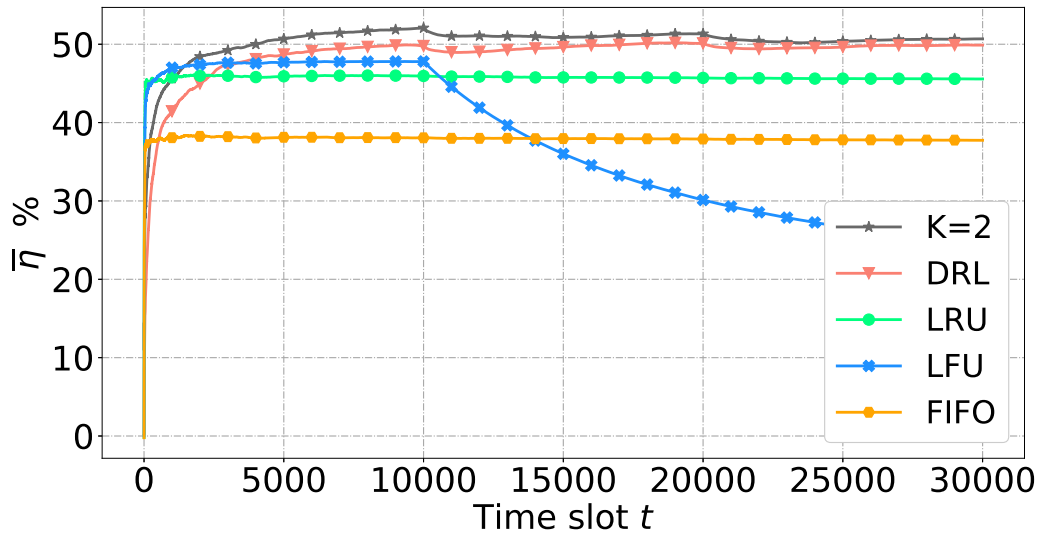


Figure 2.15: Percentage of transmission delay reduction η as the popularity distribution of contents change over time

In Fig. 2.15, we demonstrate the ability of the caching policies to adapt to varying content popularity distributions. In this experiment, the users' preferences for files change at every 10000 time slots. The users' requests are generated using Zipf distributions with their unique ranks of files and Zipf exponents. At each change point, these parameters vary randomly. The change points and Zipf parameters are all unknown to the caching agents. We only limit the Zipf exponent β to be in the range $[1.1, 1.5]$. Then we plot the average of the percentages of the transmission delay reduction over time as $\overline{\eta}_T = \frac{1}{T}\sum_{t=1}^{T}\eta_t$, for $t = 1, 2, ..., 30000$. As shown in Fig. 2.15, the proposed framework with both values of number of neighbors achieve relatively lower performance at the beginning, because unlike the other three caching policies (i.e., LRU, LFU, and FIFO), the proposed framework does not directly collect the statistics from the users' requests, but generally adjust the parameters of the neural networks and learn the popularity patterns of the files. After the neural networks are trained well, the two actor-critic agents are able to achieve better long-term performance over the other policies. As before, having larger number of neighbors (i.e.,

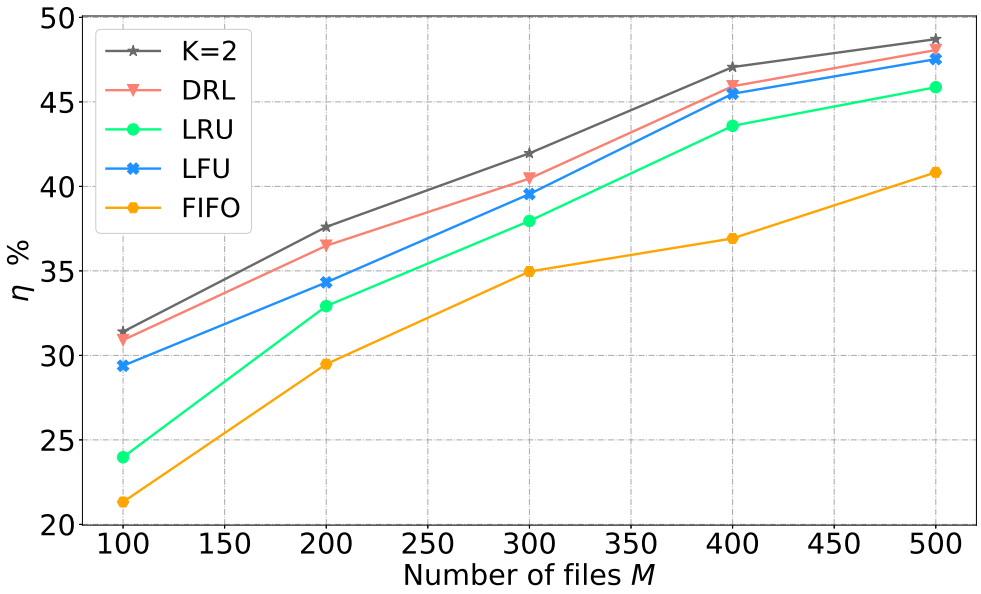


Figure 2.16: Percentage of transmission delay reduction vs. total number of files. We fix the Zipf exponent at $\beta = 1.3$ and the cache ratio $\sigma = 0.1$.

$K = 2$) results in the best performance. And at each time the popularity distribution changes, even though the average transmission delay reduction slightly drops as the actor-critic framework updates the parameters to adapt to the new pattern, the actor-critic agents can keep a stable performance after the re-training process. The LFU policy performs the best at the beginning, but due to the frequency pollution, the performance drops quickly at the first change point and continues diminishing. For the LRU and FIFO policies, the performances are stable, because the cache size is limited and the files that are used to be popular and less popular after the change can be replaced in a relatively short amount of time. However, as evidenced in this figure, their performances are lower and the proposed framework is more suitable be to applied in scenarios that require long-term high performance and stability.

In Fig. 2.16, we plot the percentage of transmission delay reduction as a function of the total number of files. Actor-critic agents again outperform the other caching policies.

Chapter 3

A Deep Actor-Critic Reinforcement Learning Framework for Dynamic Multichannel Access

The scarcity of spectral resources makes it challenging to satisfy the ever-growing demand for high-quality wireless communication services, and increases the importance to improve the spectrum utilization. Dynamic spectrum access, which enables users to proactively choose available channels, is one key approach to address this problem. However, dynamic spectrum access can be very challenging for instance in scenarios in which there is lack of prior information on the channel conditions or especially when the channel conditions in different frequency bands vary over time and multiple users dynamically access the channels simultaneously. Motivated by these considerations, we in this chapter propose a deep reinforcement learning based framework for dynamic multichannel access.

In particular, we in this chapter consider an environment with N correlated chan-

nels, and each channel is assumed to have two possible states: good or bad[1]. The good state indicates better channel conditions and higher channel capacity, ensuring transmission success, while the bad state implies increased chances for transmission failure due to unfavorable channel conditions. It is assumed that the state of each channel can switch between good and bad, and this switching pattern can be modeled as a Markov chain with at most 2^N states. In order to successfully transmit their data, all users aim at selecting the good channels as frequently as possible. Since the channel switching pattern and other users' choices are unknown, each user can only try sensing or accessing different channels at each time and determine the pattern as much as possible based on its own observation. Here, we assume that users can receive a feedback in the channels they selected, and this channel feedback will indicate the channel conditions[2]. In this way, users learn if their selections lead to channels with good or bad states, and based on such previous experience, they predict the channel states in the next time period when they need to choose a channel, and increase the probability of choosing a channel in good state.

Since each user is only able to learn the states of channels selected by itself, the environment is partially observable to the users, making the channel selection problem a partially observable Markov decision process (POMDP). This is to say, to solve the dynamic spectrum access problem, an access policy that only depends on the user's individual information on the state of previously accessed channels after each time of sensing must be determined. However, in theory, POMDP problems are PSPACE-hard, and the increase in the number of states will lead to double-exponential growth in complexity. Hence, it is rather difficult to obtain the optimal solution. Conventionally, heuristic algorithms [117,118] and Monte Carlo methods [119,120] have been used to find acceptable sub-optimal solutions in a reasonable duration of time. In

[1]Having more than two states can also be incorporated in the analysis and algorithms as done in Section 3.4.3.2 with channels having "excellent", "good", and "bad" states.

[2]We describe specific types of feedback that can indicate the channel condition in Section 3.1.2.

both approaches, decisions are made based on previous exploration results.

In this chapter, inspired by the effectiveness of reinforcement learning methods in exploring unknown environments [121, 122], we investigate the use of deep reinforcement learning algorithms in solving the dynamic spectrum access problem. More specifically, we propose a deep actor-critic reinforcement learning framework for dynamic spectrum access, aiming at increasing the accuracy of channel selection with good states.

3.1 System Model

In this chapter, we consider the dynamic multichannel access problem in which users dynamically select channels and learn the channel states. Below, we describe the system model in detail.

3.1.1 Channel State Switching Patterns

In the system we consider, there are N correlated channels in total, and each channel has two possible states: the good channel state, which allows the user to transmit successfully, and the bad channel state, which will lead to transmission failure. We assume that the states of these channels are dynamically switching between good and bad. Since the channels are correlated, we can model the switching pattern of the states of all channels as a Markov chain, denoted as $\mathcal{P}$. In each time slot t, we denote the channel state as $\mathtt{X}_t = \{\mathtt{x}_{1,t}, \mathtt{x}_{2,t}, ..., \mathtt{x}_{N,t}\}$, where N is the total number of channels, $\mathtt{x}_{i,t}$ stands for the state of the ith channel in time slot t. And we assume that the channel state can only change at the beginning of each time slot and remains the same within the time slot. State transition probabilities at the beginning of each time slot can be described as follows: the probability that the channel state will change from current state to a different state in the Markov chain $\mathcal{P}$ is p; and the probability that

the channel state will remain the same is $(1-p)$.

3.1.2 Users' Observations

We assume that the channel switching pattern is unknown to the users. In order to successfully transmit their data, users have to deduce the channel switching pattern from their observations of the channels. Different mechanisms can be used to obtain such observations (or channel feedback). One approach is that the users send pilot or data signals over the selected channels and receive feedback from their corresponding receivers in the form of signal-to-interference-plus-noise ratios (SINRs). Or the users can tune to certain channels and determine the SINRs of signals received in those channels. In order to keep the analysis general in the chapter, we assume that the users learn the conditions of the channels they have selected and accessed without explicitly detailing the particular mechanism. We assume that each user can only select k channels to access, where $1 \leq k < N$, and by accessing the selected channels, the user can learn the corresponding channel states while the states of all other channels that are not selected remain unknown to the user. Thus, from the user's perspective, choosing the channels in good states out of N channels is a POMDP, in which the user aims to learn the pattern of variations in the channel states based on previous decisions. In the following two subsections, we describe the user observations initially in the case in which there is only one user in the system, and then in the case where there are multiple users trying to access the channels simultaneously.

3.1.2.1 Single-User Scenario

For the system with only one user, we denote the user's observation in time slot t as $O_t = \{o_{1,t}, o_{2,t}, ..., o_{N,t}\}$, where N is again the total number of channels, and $o_{i,t}$ stands for the user's observation of the ith channel, where $i = 1, 2, \ldots, N$, in time slot t. We assume that when the user selects a channel to access, the state of the

chosen channel is revealed to the user. In the single-user case, let us define the state of channel i, for $i = 1, 2, ..., N$, in time slot t as

$$\mathtt{x}_{i,t} = \begin{cases} +1 & \text{if the } i^{\text{th}} \text{ channel is in good state in time slot } t \\ -1 & \text{if the } i^{\text{th}} \text{ channel is in bad state in time slot } t \end{cases} . \tag{3.1}$$

Now, for the user, the observation of each channel is

$$\begin{aligned} o_{i,t} &= \phi_{i,t}\, \mathtt{x}_{i,t} \\ &= \begin{cases} \mathtt{x}_{i,t} & \text{if the } i^{\text{th}} \text{ channel is selected in time slot } t \\ 0 & \text{if the } i^{\text{th}} \text{ channel is not selected in time slot } t \end{cases} \end{aligned} \tag{3.2}$$

where $\phi_{i,t}$ is the indicator defined as

$$\phi_{i,t} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ channel is selected in time slot } t \\ 0 & \text{if the } i^{\text{th}} \text{ channel is not selected in time slot } t \end{cases} . \tag{3.3}$$

As seen above, if a channel is not selected for access, its state is not known and we indicate the observation for those channels as zero.

3.1.2.2 Multi-User Scenario

For the system with $M > 1$ users, users make their own decisions to choose which channels to access and can only receive the feedback on the channel states corresponding to the selected channels. We assume that the users are not able to exchange information on their selections and observed channel states among themselves. Therefore, it is unavoidable that in some time slots, more than one user can choose/access the same channel. In these circumstances, even if the selected channels are in good state, the users may experience "degraded" channels due to potential collisions. Taking this

into account, we define the state of the ith channel in time slot t as follows:

$$\mathtt{x}_{i,t} = \begin{cases} +1 \text{ if the } i^{\text{th}} \text{ channel is in good state and no collision occurs} \\ d_{i,t} \text{ if the } i^{\text{th}} \text{ channel is in good state and collision occurs} \\ -1 \text{ if the } i^{\text{th}} \text{ channel is in bad state} \end{cases} \tag{3.4}$$

where $d_{i,t} < 1$ is the discount factor for the good channels that are selected by more than one user. Note that this discount factor is introduced in order to discourage the users to access the same good channel in the same time slot so that collisions in channels with good states can be avoided as much as possible. In practice, different mechanisms can be employed for collision detection and different discount factor formulations can be used.

One approach is to define the discount factor $d_{i,t}$ to be proportional to $\frac{1}{m_i}$, where $m_i > 1$ is the number of users that have selected the ith channel. This choice can be justified as follows. As noted above, let us assume that the users receive SINR feedback from their corresponding receivers after accessing the selected channels. We denote the received power (after having experienced fading) when user j accesses a good channel as $P_{r,j}^{good}$, while the received power when the user j accesses a bad channel is indicated as $P_{r,j}^{bad}$. We further assume that $P_{r,j}^{good} \gg N_0 \gg P_{r,j}^{bad}$, where N_0 is the noise power. Now, we can choose two thresholds Γ_1 and Γ_2 with which the following inequalities with the received SINRs are satisfied:

$$\underbrace{\frac{P_{r,j}^{good}}{N_0}}_{\substack{\text{no interference/}\\ \text{good channel}}} > \Gamma_1 > \underbrace{\frac{P_{r,j}^{good}}{\sum\limits_{k\in\mathbb{I},k\neq j} P_{r,k}^{good} + N_0}}_{\text{interference/good channel}} > \Gamma_2 > \underbrace{\frac{P_{r,j}^{bad}}{N_0}}_{\substack{\text{no interference/}\\ \text{bad channel}}} > \underbrace{\frac{P_{r,j}^{bad}}{\sum\limits_{k\in\mathbb{I},k\neq j} P_{r,k}^{bad} + N_0}}_{\text{interference/bad channel}} \tag{3.5}$$

where the leftmost term in (3.5) is the signal-to-noise ratio (SNR) when user j accesses a good channel and there are no other users in this channel (and hence no interference). Note that since we assume that the received power in a good channel satisfies $P_{r,j}^{good} \gg N_0$, we have $\frac{P_{r,j}^{good}}{N_0} \gg 1$. On the other hand, if multiple users access a good channel and a collision occurs, the SINR of user j becomes $\frac{P_{r,j}^{good}}{\sum_{k\in\mathbb{I},k\neq j} P_{r,k}^{good}+N_0}$, where $\mathbb{I}$ denotes the index set of interfering users. Assuming that the received powers of different users at the same receiver are comparable, we have

$$\frac{P_{r,j}^{good}}{\sum\limits_{k\in\mathbb{I},k\neq j} P_{r,k}^{good} + N_0} \approx \frac{P_{r,j}^{good}}{\sum\limits_{k\in\mathbb{I},k\neq j} P_{r,k}^{good}} \approx \frac{1}{m-1} \tag{3.6}$$

where m is the number of users that select the good channel in the given time slot, and the first approximation is due to received powers being much larger than noise power. The second approximation in (3.6) (which is due to received power levels being comparable) provides a justification for choosing the discount factor $d_{i,t}$ to be proportional to $\frac{1}{m}$. As to the cases in which users select bad channels, since the received power levels are small (e.g., because of potentially strong attenuation in the channel) and the noise is dominant, SNR and SINRs levels will be small. Finally, we note that if there exists thresholds Γ_1 and Γ_2 satisfying the inequalities in (3.5), comparisons with these thresholds would serve as one approach to identify channel states and recognize collisions via SNR/SINR feedback.

Now, we define the observation of user j in time slot t as $O_{j,t} = \{o_{j,1,t}, ..., o_{j,i,t}, ..., o_{j,N,t}\}$, where j is the user index and i is the index of the channel. Similar to the single-user

scenario, the user j's observation of the ith channel is

$$
\begin{aligned}
o_{j,i,t} &= \phi_{i,j,t}\, \mathtt{x}_{i,t} \\
&= \begin{cases} \mathtt{x}_{i,t} & \text{if the } i^{\text{th}} \text{ channel is selected in time slot } t \\ 0 & \text{if the } i^{\text{th}} \text{ channel is not selected in time slot } t \end{cases}
\end{aligned} \tag{3.7}
$$

where $\phi_{i,j,t}$ is an indicator defined as

$$
\phi_{i,j,t} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ channel is selected by user } j \text{ in time slot } t \\ 0 & \text{if the } i^{\text{th}} \text{ channel is not selected by user } j \text{ in time slot } t \end{cases} . \tag{3.8}
$$

Similarly as in the single-user case, the channel states are revealed only for the selected/accessed channels. For the channels that are not selected for access, the observation is set to zero.

3.1.2.3 Users' Action Space

As we noted before, the users can only select k channels to access in each time slot, where $1 \leq k < N$. We consider a discrete action space $\mathcal{A} = \{a_1, a_2, \ldots, a_{\mathcal{D}}\}$, where $\mathcal{D}$ is the total number of valid actions. Each valid action in the action space describes the k indices of the channels that will be accessed. So, for a specific value of k, we have the number of actions equal to $\binom{N}{k}$. For example, if $k = 1$, each action a_i, $i = 1, 2, ..., \mathcal{D} = N$, corresponds to accessing channel i; while if $k = 2$, each valid action a_i, $i = 1, 2, ..., \mathcal{D}$, can be described by the indices of the two chosen channels. Hence, in each time slot, the user will pick one action from the action space $\mathcal{A}$, access the corresponding k channels, and the condition of the chosen channels will be revealed.

3.2 Multichannel Access Problem Formulation

In this section, we formulate the dynamic multichannel access problem based on the channel access mechanisms and the corresponding rewards. To learn the channel switching pattern, we propose an actor-critic algorithm based deep reinforcement learning framework, which works as an agent to make channel selection decisions for the user. In this framework, the agent obtains the user's observation of the channels and makes channel access decisions based on the observation, and subsequently receives the feedback from the channels, and updates the decision policy.

3.2.1 Single-User Scenario

We first consider the case in which there is only one user in the system. The reward $r_{i,t}$ obtained when the ith channel is selected/accessed by the user in time slot t is defined as follows:

$$\begin{aligned} r_{i,t} &= \mathtt{x}_{i,t} \\ &= \begin{cases} +1 & \text{if the } i^{\text{th}} \text{ channel is in good state in time slot } t \\ -1 & \text{if the } i^{\text{th}} \text{ channel is in bad state in time slot } t \end{cases} \end{aligned} \tag{3.9}$$

Since the user aims to select good channels as much as possible to ensure frequent successful transmissions, the agent is designed to find a policy π (which is a mapping from the observation space $\mathcal{O}$ to the action space $\mathcal{A}$) that maximizes the long-term expected reward R of channel access decisions:

$$\pi^* = \arg\max_{\pi} R \tag{3.10}$$

where π^* denotes the optimal decision policy, and in a finite time duration T, we express R as

$$R = \frac{1}{T} \sum_{t=1}^{T} \sum_{i=1}^{N} \phi_{i,t}\, r_{i,t} \tag{3.11}$$

where $\phi_{i,t}$ is the indicator function defined in (3.3).

Now, the problem can be formulated as

$$\textbf{P1:} \qquad \underset{\{\phi_{i,t}\}}{\text{Maximize}} \qquad R \tag{3.12}$$

$$\text{Subject to} \qquad \sum_{i=1}^{N} \phi_{i,t} = k, \forall t \tag{3.13}$$

where k is the number of channels that the user can select in each time slot, and according to the definition of R, we have $R \in [-k, k]$.

3.2.2 Multi-User Scenario

In this case, we assume that there are multiple users, and each user can independently select a channel to access without knowing other users' actions. Thus, each user will employ a separate actor-critic reinforcement learning agent. Since each user has the goal to choose good channels as frequently as possible, the agent of user j is required to find a policy π_j for $j = 1, 2, \ldots, M$ (mapping the observation space $\mathcal{O}_j$ to the action space $\mathcal{A}$) that maximizes the long-term expected reward R_j of the channel access decisions for user j:

$$\pi_j^* = \arg\max_{\pi} R_j. \tag{3.14}$$

Similarly as in the single-user case, π_j^* denotes the optimal decision policy for user j, and in a finite time duration T, we express R_j as

$$R_j = \frac{1}{T} \sum_{t=1}^{T} \sum_{i=1}^{N} \phi_{i,j,t}\, r_{i,t} \tag{3.15}$$

where $\phi_{i,j,t}$ is an indicator defined in (3.8) and the reward $r_{i,t}$ obtained when user j accesses the ith channel in time slot t is

$$
\begin{aligned}
r_{i,t} &= \mathtt{x}_{i,t} \\
&= \begin{cases} +1 & \text{if the } i^{\text{th}} \text{ channel is in good state and no collision occurs} \\ +1 \cdot d_{i,t} & \text{if the } i^{\text{th}} \text{ channel is in good state and collision occurs} \\ -1 & \text{if the } i^{\text{th}} \text{ channel is in bad state} \end{cases} .
\end{aligned} \tag{3.16}
$$

Hence, the optimization problem for user j for $j = 1, 2, \ldots, M$ can be formulated as

$$\textbf{P2:} \qquad \underset{\{\phi_{i,j,t}\}}{\text{Maximize}} \qquad R_j \tag{3.17}$$

$$\text{Subject to} \qquad \sum_{i=1}^{N} \phi_{i,j,t} = k. \tag{3.18}$$

The formulation of each user's optimization problem is similar to that in the single-user case. However, the optimal solution in the multi-user scenario should find the channels in good condition and also avoid collisions at the same time. This means that the agent needs to learn both the channel switching pattern and the other users' channel selection pattern from the channel feedback. When there are not enough channels in good state, users compete for such limited number of good channels. On the other hand, if a sufficient number of good channels exists simultaneously, each user can potentially access a good channel without experiencing a collision.

3.3 Actor-Critic Reinforcement Learning Framework

In this section, we describe the proposed actor-critic deep reinforcement learning framework for dynamic multichannel access and develop algorithms for both single- and multi-user cases.

3.3.1 Actor-Critic Agent's Observation Space, Actions, and Rewards

We first introduce the relevant definitions within the actor-critic framework.

Channel State and Agent's Observation: The channel state is varying as described by a Markov chain and it is a part of the environment, which is unknown to the agent. Therefore, the agent can only take its own observation space $\mathcal{O}$ as the input to the actor-critic framework. The agent can only access the chosen channels in each iteration, and observe the reward that depends on the state of the chosen channels. As defined in the previous section, in time slot t the user observation O_t (or $O_{j,t}$ for multi-user scenario) is a sparse matrix with only k nonzero elements in each column (representing the observation vector at any given time), where k is the number of channels that are selected to be accessed in each time slot. The users will learn on the basis of their previous experiences. We assume the agent keeps an observation space $\mathcal{O}$ that consists of the most recent Ω observations O_t. The observation space is initialized as an all-zero $N \times \Omega$ matrix, and at each time t, the latest observation O_t will be added to the observation space, and oldest observation $O_{t-\Omega}$ will be removed. The updated observation space $\mathcal{O}$ at time $t+1$ is denoted as $\mathcal{O}_{t+1} = \{O_t; O_{t-1}; ...; O_{t-(\Omega-1)}\}$. For instance, Fig. 3.1 depicts a scenario in which $\Omega = 16$ and $k = 1$.

Action: The agent scores all possible actions in the action space $\mathcal{A}$ based on the user's observation, and the action with the highest score will be chosen. In our setting,

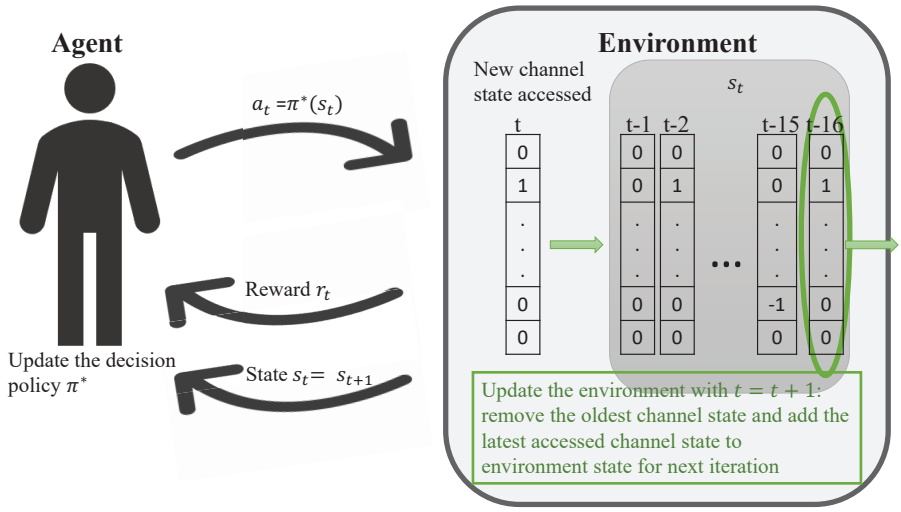


Figure 3.1: In reinforcement learning, the agent constantly observes the environment and makes a decision. The decision will be executed and the corresponding feedback will be used to update the policy.

the action indicates which channel or channels to access.

Reward: The reward is received when the action is executed, meaning that the agent chooses a channel and gets direct feedback from the environment. The reward is defined based on the condition of the chosen channel, as formulated in the optimization problems **P1** and **P2**.

In addition to the average reward formulations given in (3.11) and (3.15), we also define here the reward received by the agent in time slot t since this will be used in the actor-critic reinforcement learning algorithm. In particular, in the single-user scenario, the reward in time slot t is

$$R_t = \sum_{i=1}^{N} \phi_{i,t}\, r_{i,t} \tag{3.19}$$

where $\phi_{i,t}$ is given in (3.3) and $r_{i,t}$ is given in (3.9).

In the multi-user case, the reward for agent j in time slot t can be expressed as

$$R_{j,t} = \sum_{i=1}^{N} \phi_{i,j,t}\, r_{i,t} \tag{3.20}$$

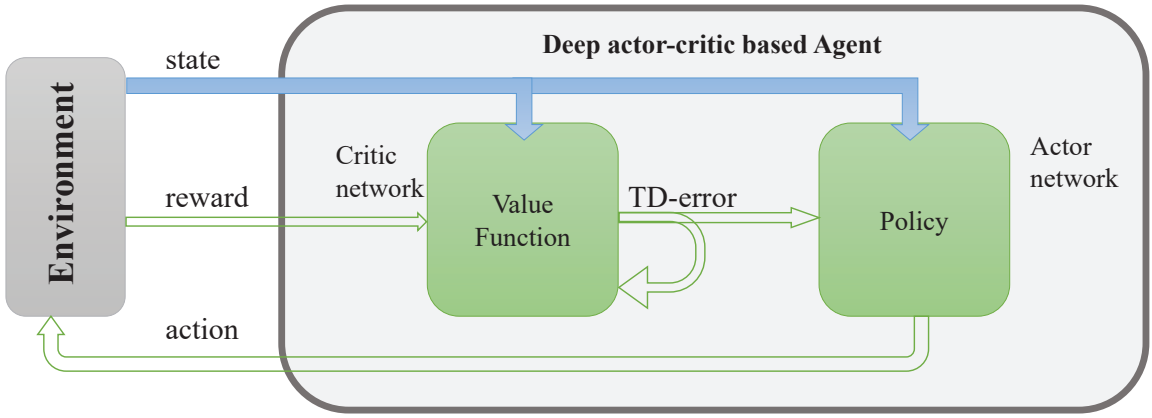


Figure 3.2: Structure of the actor-critic deep reinforcement learning agent

where $\phi_{i,j,t}$ is the channel selection indicator for agent j as defined in (3.8), and $r_{i,t}$ is provided in (3.16).

3.3.2 Algorithm Overview

In this subsection, we describe the architecture of the actor-critic algorithm. The actor-critic architecture consists of two neural networks: actor and critic. In our model, the actor neural network is parameterized by θ, and the critic neural network is parameterized by μ. The structure of the actor-critic deep reinforcement learning agent is depicted in Fig. 3.2

Actor: The actor is employed to explore a policy π, that maps the agent's observation $\mathcal{O}$ to the action space $\mathcal{A}$:

$$\pi_\theta(\mathcal{O}) : \mathcal{O} \rightarrow \mathcal{A} \tag{3.21}$$

So the mapping policy $\pi_\theta(\mathcal{O})$ is a function of the observation $\mathcal{O}$ and is parameterized by θ. And the chosen action can be denoted as

$$a = \pi_\theta(\mathcal{O}) \tag{3.22}$$

where we have $a \in \mathcal{A}$. Since the action space is discrete, we use softmax function at the output layer of the actor network so that we can obtain the scores of each actions. The scores sum up to 1 and can be regarded as the probabilities to obtain a good reward by choosing the corresponding actions.

Critic: The critic is employed to estimate the value function $V(\mathcal{O})$. At time instant t, when the action a_t is chosen by the actor network, the agent will execute it in the environment and send the current observation $\mathcal{O}_t$ along with the feedback from the environment to the critic. The feedback includes the reward r_t and the next time instant observation $\mathcal{O}_{t+1}$. Then, the critic calculates the TD (Temporal Difference) error:

$$\delta_t = R_t + \gamma V_\mu(\mathcal{O}_{t+1}) - V_\mu(\mathcal{O}_t) \tag{3.23}$$

where $\gamma \in (0, 1)$ is the discount factor [3].

Update: The critic is updated by minimizing the least squares temporal difference (LSTD):

$$V^* = \arg\min_{V_\mu} (\delta_t)^2 \tag{3.24}$$

where V^* denotes the optimal value function.

The actor is updated by policy gradient. Here, we use the TD error to compute the policy gradient[4]:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(\mathcal{O}, a)\delta_t] \tag{3.25}$$

where $\pi_\theta(\mathcal{O}, a)$ denotes the score of action a under the current policy. Then, the weighted difference of parameters in the actor at time t can be denoted as $\Delta\theta_t = \alpha\nabla_{\theta_t} \log \pi_{\theta_t}(\mathcal{O}_t, a_t)\delta_t$, where $\alpha \in (0, 1)$ is the learning rate. And the actor network i

[3]We note that the reward R_t in (3.23) is given by (3.19) in the single-user case, and is equal to $R_{j,t}$ in (3.20) when user/agent j is considered in the multi-user scenario.

[4]In (3.25), policy gradient is denoted by $\nabla_\theta J(\theta)$ where $J(\theta)$ stands for the policy objective function, which is generally formulated as the statistical average of the reward.

can be updated using the gradient decent method:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} \log \pi_{\theta_t}(\mathcal{O}_t, a_t)\delta_t. \tag{3.26}$$

3.3.3 Workflow for a Single User

In the case of a single user, there is only one actor-critic network employed as the agent to dynamically select channels. At the beginning of time slot t, the agent will collect the latest Ω observations of channels and the observation space is denoted as $\mathcal{O}_t$. Then the actor network will choose k channels according to the decision policy, i.e., the action with highest score will be selected. Next, the channel reward will be sent from every chosen channel. Based on the reward, the current observation space $\mathcal{O}_t$ and the observation space for the next time slot $\mathcal{O}_{t+1}$, the critic network calculates the TD-error. And finally the critic and actor networks will be updated based on the TD-error.

The full framework is provided in Algorithm 4 below.

3.3.4 Workflow for Multiple Users

In the case that multiple users access the channels simultaneously, we assume that all access decisions and actions are completed at the same time. At the beginning of the time slot t, agent j for $j = 1, 2, \ldots, M$ collects the corresponding user's observation $\mathcal{O}_{j,t}$ of all channels. Then, each user will select the action with highest score according to its own decision policy. Next, the agents receive the rewards from their chosen channels simultaneously. Based on their own rewards and observations, critic networks will calculate the corresponding TD-error to update the critic and actor networks, respectively.

The full framework is provided in Algorithm 5 below.

Algorithm 4 Actor-Critic Deep Reinforcement Learning Algorithm for Single-User Dynamic Multichannel Access

Initialize the critic network $V_\mu(\mathcal{O})$ and the actor $\pi_\theta(\mathcal{O})$, parameterized by μ and θ respectively.
The environment initializes the state of each channel `X`.
The agent initializes its observation as all zero matrix $\mathcal{O}_0$
for $t = 0, T$ **do**
With the observation, the agent selects k channels according to the decision policy $a_t = \pi(\mathcal{O}_t|\theta)$ w.r.t. the current policy
Agent accesses the chosen channels and receives the reward R_t based on the channel state.
Based on the reward, the new observation of channels O_t will be added to the observation space for the next time slot $\mathcal{O}_{t+1}$
Critic calculates the TD error: $\delta_t = R_t + \gamma V(\mathcal{O}_{t+1}) - V(\mathcal{O}_t)$
Update the critic by minimizing the loss: $\mathcal{L}(\mathcal{O}_t, a_t) = (\delta_t)^2$
Update the actor policy by maximizing the action value: $\Delta\theta_t = \alpha \nabla_{\theta_t} \log \pi_{\theta_t}(\mathcal{O}_t, a_t)\delta_t$, $\alpha \in (0, 1)$.
Update the observation $\mathcal{O}_t = \mathcal{O}_{t+1}$.
Update the channel state `X`.
end for

3.4 Experiments and Numerical Results

In this section, we initially describe the simulation setting and then evaluate the performance of the proposed actor-critic framework via numerical results, and provide comparisons with random channel access, the DQN based framework proposed in [66] and also the optimal policy under the assumption that the channel switching patterns are known.

3.4.1 Simulation Setting

In our implementation, the design of the agent for a single user and that of the agent for each user in the multi-user case are similar. The agent consists of two neural networks: actor and critic. Each of the two networks has two layers. For the actor, which scores all actions in the action space, the first layer has 200 neurons with ReLU as the activation function, and second layer has $\mathcal{D}$ neurons with Softmax as

Algorithm 5 Actor-Critic Deep Reinforcement Learning Algorithm for Multi-User Dynamic Multichannel Access

Initialize the critic network $V_{\mu_j}(\mathcal{O}_j)$ and the actor $\pi_{\theta_j}(\mathcal{O}_j)$ for user j, parameterized by μ_j and θ_j respectively, with $j = 1, 2, \ldots, M$.
The environment initializes the state of each channel `X`.
The agent j initializes its observation as all zero matrix $\mathcal{O}_{j,0}$, $j = 1, 2, \ldots, M$.
for $t = 0, T$ **do**
  for $j = 1, M$ **do**
    With the observation, the agent selects an action $a_{j,t} = \pi(\mathcal{O}_{j,t}|\theta_j)$ w.r.t. the current policy
  end for
  Agents start accessing the chosen channels simultaneously, and every agent receives the corresponding reward $R_{j,t}$ based on the channel state and the access collisions.
  for $j = 1, M$ **do**
    Based on the reward, agent j adds the new observation of channels $O_{j,t}$ to the observation space for the next time slot $\mathcal{O}_{j,t+1}$
    Every critic calculates the corresponding TD error: $\delta_{j,t} = R_{j,t} + \gamma V(\mathcal{O}_{j,t+1}) - V(\mathcal{O}_{j,t})$
    Update the critic by minimizing the loss: $\mathcal{L}(\mathcal{O}_{j,t}, a_{j,t}) = (\delta_{j,t})^2$
    Update the actor policy by maximizing the action value: $\Delta\theta_{j,t} = \alpha \nabla_{\theta_{j,t}} \log \pi_{\theta_{j,t}}(\mathcal{O}_{j,t}, a_{j,t}) \delta^{\pi_{\theta_{j,t}}}$, $\alpha \in (0, 1)$.
    Update the observation $\mathcal{O}_{j,t} = \mathcal{O}_{j,t+1}$ for every user.
  end for
  Update the channel state `X`.
end for

the activation function, where N is the total number of channels. For the critic, which computes the value of the chosen action, the first layer has 200 neurons with ReLU as the activation function, and second layer has 1 neuron. Especially, since the critic will evaluate the decision made by the actor, the learning rate of the actor network should be smaller than that of the critic network to make the actor network converge slower than critic network. Here, we set the learning rate of the critic network as 0.0005, and the learning rate of the actor network as 0.0001. To ensure stability, both learning rates will decay exponentially with the decay rate 0.95 for every 250000 time slots. To encourage the agent to explore the environment, we employ ϵ-greedy policies and we set $\epsilon = 0.1$. Moreover, in our implementation, we set $\Omega = 16$, so that the agent has access to observations of the most recent 16 time slots.

3.4.2 Average Reward in the Single-User Case

In this section, we present the results for the single-user model, and compare our framework with the DQN framework, random access, Whittle index heuristic, and also the optimal decision policy under the assumption that the channel switching pattern is known to the user.

DQN framework proposed in [66] consists of two hidden layers and maintains a replay memory with a size of $1,000,000$. To update the network, the DQN framework will replay a minibatch of 32 samples extracted from the memory. For a fair comparison with the actor-critic agent, our DQN algorithm has the same structure as the actor network, i.e., in our implementation, the DQN has only one hidden layer. Besides, we set the memory size as $1,000$ so that the DQN could be more adaptive. Unlike the aforementioned actor-critic structure, the DQN consists of only one neural network that is used to select the channels. Therefore, the neural network in DQN acts as the counterpart of the actor network in the proposed framework. Without the critic network, the DQN replays a batch of previous transitions and use the loss

between the previously estimated Q values and the estimated Q values from replaying to update the neural network.

In the *random access policy*, there is no learning and users randomly select channels at the beginning of each time slot, and all channels will be accessed with the same probability.

In *Whittle index heuristic* [123], all the channels are treated as independent and the transition probability matrix of each channel is assumed to be obtained by observing the channels separately over a certain period (e.g. 10,000 iterations) in advance. Then, the transition probability matrices are used to update the belief vector for final decision making. More details can be found in [123].

We also consider the *optimal policy* [66, Theorem 1] assuming that the channel dynamics is known to the user. The user accesses a channel at the beginning. Then, according to the policy, for instance when the channel state switching probability is greater that 0.5, if the user selects a good channel at time t, the user will choose a channel in the next activated subset of channels according to the known pattern in the next time slot. On the other hand, if the user selects a bad channel at time t, the user will stay at the chosen channel in the next time slot. The reverse strategy is employed if the switching probability is less than 0.5.

3.4.2.1 Single Good Channel

In this experiment, we consider different number of channels, i.e., $N = \{16, 32, 64\}$, and only one channel is in good state in each time slot. To evaluate the performance, we calculate the expected reward R with different Markov chains $\mathcal{P}$. To define a Markov chain for the channel distribution, we need to specify the channel states in order and the state switching probabilities. We assume that, for each state, the probability that current state will transfer to another state is p, and the probability that the current state will be kept is $1 - p$. Our experiments were conducted in two

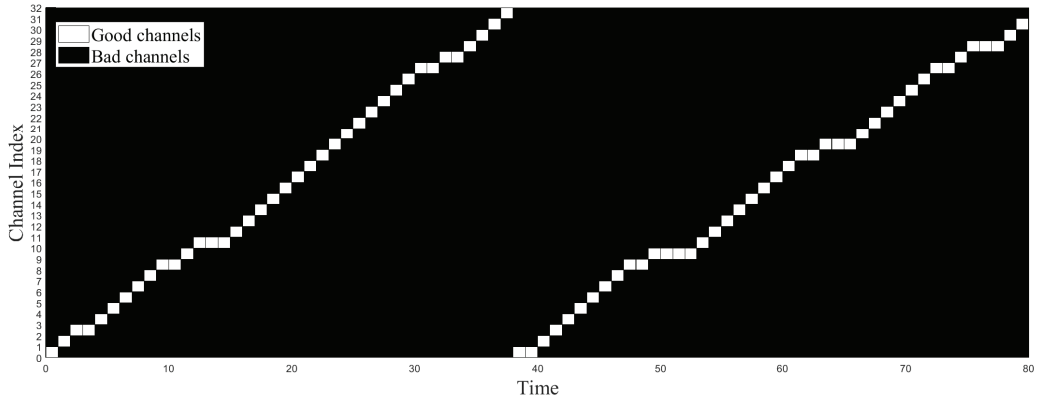


Figure 3.3: Round-robin switching pattern when only one of the 32 channels is in good condition and the switching probability is $p = 0.75$. The channel in good state at a given time is indicated by a white square.

cases:

Round-Robin Switching Scenario: In this experiment, we assume that the index of the only good channel switches from 1 to N according to a round-robin scheduling and we assume the user can only access to one channel at a time. Then we vary the switching probabilities as $p = \{0.75, 0.80, 0.85, 0.90, 0.95\}$. This round-robin pattern with switching probability $p = 0.75$ is depicted in Fig. 3.3, where the channel with the good state is indicated with a white square at the corresponding channel index value at a given time. Since the probability $p = 0.75$ is relatively high, we have an increasing staircase pattern (indicating channels with good state changing from one to the next) more frequently than the flat pattern that occurs when the same channel stays in good state in multiple time slots.

We compare our actor-critic (AC) policy with DQN, random access, Whittle index heuristic and optimal policy in terms of the average reward. Figs. 3.4(a), 3.4(b), and 3.4(c) provide the average rewards of different policies for $N = 16, 32$ and 64 channels, respectively. In all subfigures, we notice that the average rewards of the optimal policy are identical because the channel pattern is assumed to be known in this case, and hence the increase in the number of channels makes no influence on the policy performance. Optimal policy expectedly leads to the highest average rewards.

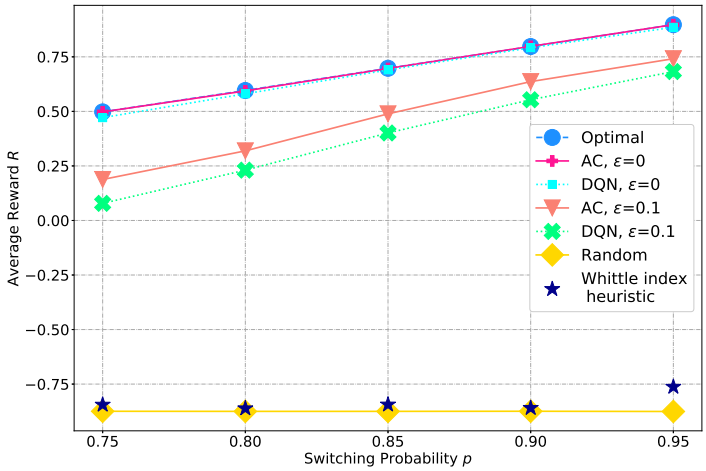
On the other hand, performance curves achieved by Whittle index heuristic and the random access policy are very low, demonstrating the inadequacy of these strategies.

More interesting and competitive performances are displayed by the proposed actor-critic policy and DQN policy. We note that in both actor-critic and DQN agents, we employ the ϵ-greedy exploration strategy with $\epsilon = 0.1$ during training. Once the neural networks are well trained, we can set $\epsilon = 0$ and choose actions only according to the learned policies or keep $\epsilon = 0.1$ in order to preserve the adaptation capabilities of the reinforcement learning agents to a changing environment. We observe in Fig. 3.4(a) that when $N = 16$, actor-critic and DQN policies with $\epsilon = 0$ achieve the optimal performance (at the expense of loss of adaptation abilities to varying conditions). When ϵ is kept equal to 0.1, due to occasional random action selections, average reward is lower but actor-critic agent performs better than DQN. Indeed, the actor-critic policy outperforms DQN with higher margins when the number of channels is increased to $N = 32$ and $N = 64$ in both cases of $\epsilon = 0$ and $\epsilon = 0.1$. In particular, the DQN framework has difficulty handling the case of 64 channels unless the switching probability is relatively large, and for the case of 32 channels, the DQN framework achieves negative rewards when $p = 0.75$. Hence, the proposed actor-critic framework is more suitable especially when the number of channels is relatively large.

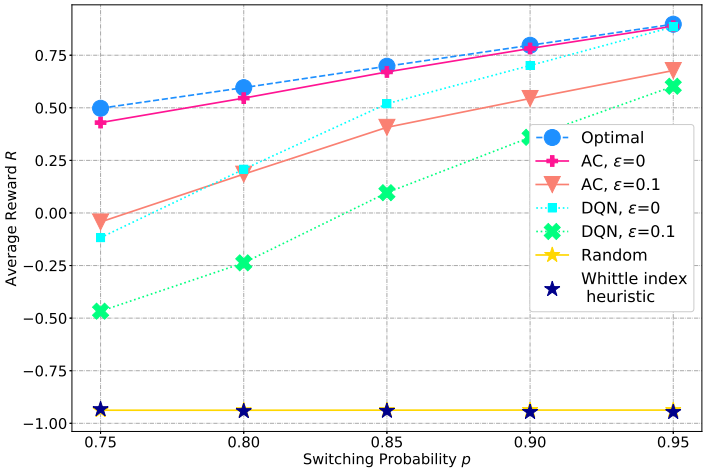
As to the overall tendency in the actor-critic and DQN performances, we can observe increasing average reward as the switching probability increases and the gap between different cases diminishes. Note that p denotes the probability of switching between states, and hence higher switching probability will decrease the uncertainty and will make it easier for the agent to learn the policy. Comparing the performances at the relatively low value of $p = 0.75$, the actor-critic framework demonstrates better performance levels, and therefore it has higher tolerance against uncertainty.

Arbitrary Switching Scenario:

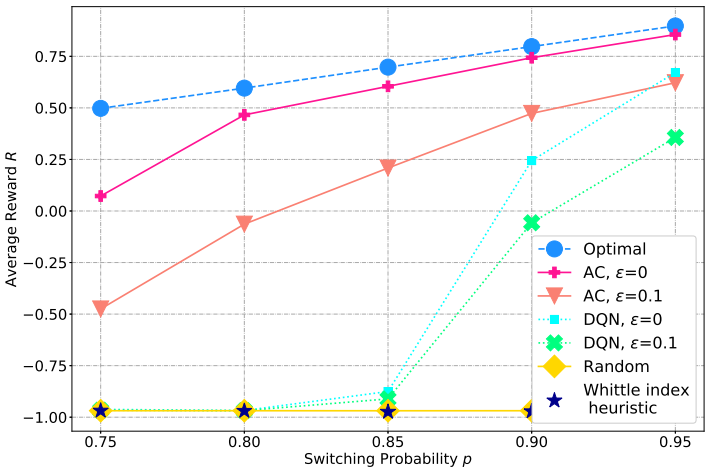
In the round-robin switching scenario, the channel states switch according to a



(a) 16 channels



(b) 32 channels



(c) 64 channels

Figure 3.4: Average reward vs. switching probability. We consider 16, 32, 64 channels cases with the switching probability varies as $p = \{0.75, 0.80, 0.85, 0.90, 0.95\}$

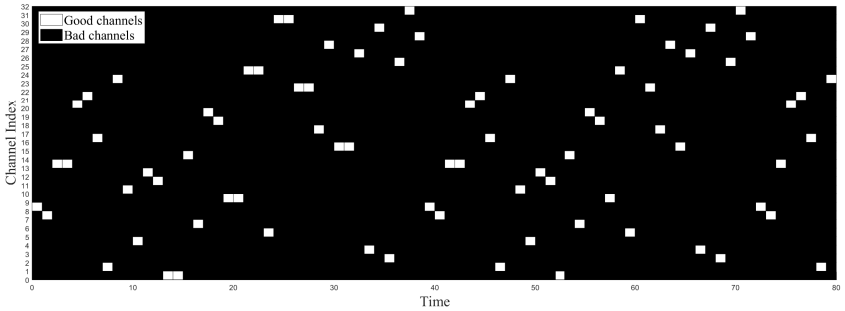


Figure 3.5: A switching pattern when only one of the 32 channels is in good condition at a given time, with a switching probability $p = 0.9$

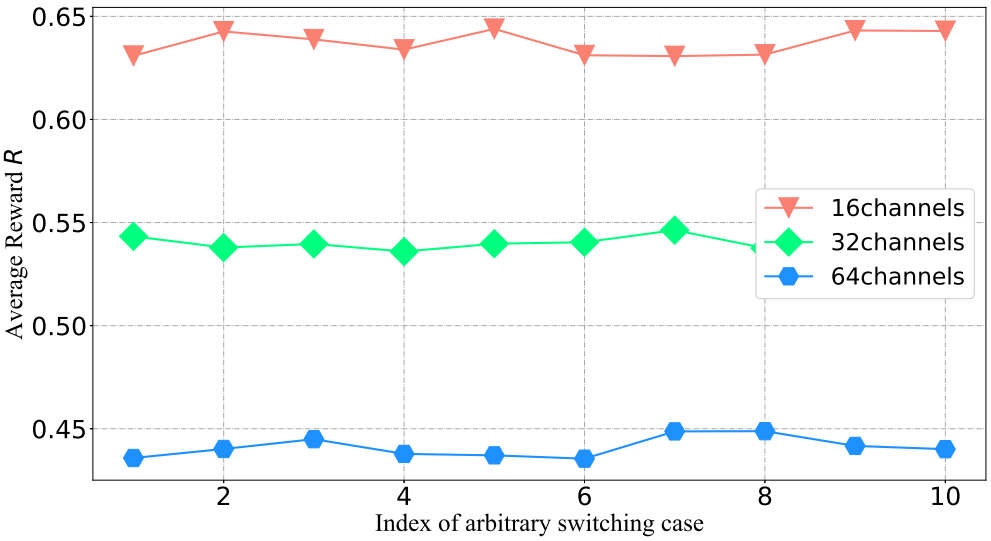


Figure 3.6: The average reward for different arbitrary switching orders

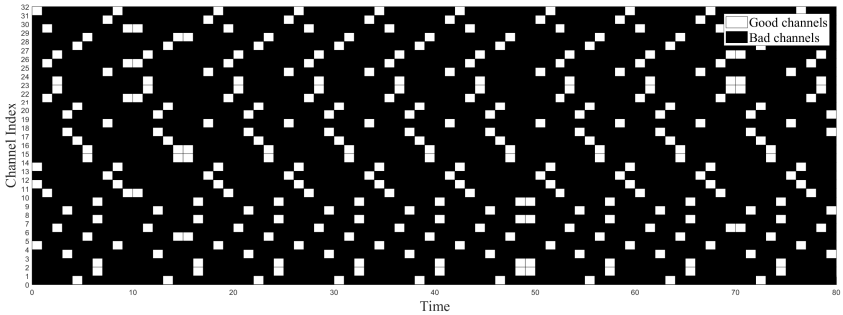


Figure 3.7: A switching pattern when each four channels of the 32 channels are grouped, with a switching probability $p = 0.9$

specific scheduling model. However, this information is unknown to the actor-critic agent, and of course is not being used in the process to find a channel access policy. Moreover, the actor-critic algorithm was proposed as a model-free algorithm. To demonstrate the performance of the proposed framework in a model-free environment, we in this experiment, fix the switching probability p at 0.9 and test the framework with 10 different arbitrary switching orders (i.e., 10 different permutations of N channels). One such switching pattern in the case of $N = 32$ channels is depicted in Fig. 3.5.

Fig. 3.6 plots the performance in the cases of 16, 32 and 64 channels with 10 randomly generated arbitrary switching orders. Still, the user is allowed to access one channel at a time. For any given number of channels, the average reward varies only slightly across different switching cases, showing that our proposed framework can work in a model-free environment. Since we have shown that the switching order will not affect the agent's performance, we assume a round robin switching scheduling in all the following experiments.

3.4.2.2 Multiple Good Channels

Now, we consider the switching pattern of a group channels, and in each state in this pattern, there are multiple channels in good state. For instance, a pattern with

four channels in good state at a given time is shown in Fig. 3.7. In this experiment, we fix the switching probability at $p = 0.9$, and study the performance in terms of the average sum reward. We assume that the user is allowed to access more than one channel at a time, and for each selected channel, the user will receive a reward (1 or -1). To show how many good channels are selected on average in one iteration, we sum the reward received in each iteration and average over time. In the implementation, we assume there are always 6 good channels when the total number of channels is 16, and 12 good channels when the total number of channel is 32.

In Figs. 3.8(a) and 3.8(b), we plot the performance when the user can access 2, 3, and 4 different channels at a time among 16 and 32 channels, respectively. The performances of the optimal policy with known channel dynamics is always around 0.9 times the maximum reward for all scenarios because of the value of the switching probability. We observe that random access overall performs poorly due to not learning the switching patterns. Whittle index heuristic achieves higher average sum rewards. Interestingly, when the number of channels that can be accessed at a time increases, the performance of Whittle index heuristic exceeds that of DQN in the experiment with 32 channels. Among the learning-based policies, the actor-critic agent achieves highest rewards. For both actor-critic and DQN policies, when there are 16 channels, the average sum reward increases as the number of channels that can be accessed increases but with diminishing returns. As introduced in Section 3.1, when the user is allowed to choose k different channels, each action stands for a set of channels to be accessed. Therefore, the size of the action space grows from N to $\binom{N}{k}$. And the performance of the learning based policies is significantly influenced by the size of the action space. For instance, in the case with 32 channels, the average sum reward achieved by the DQN agent diminishes as the number of channels to be accessed increases, demonstrating that the DQN agent is not able to handle the growing size of the action space. On the other hand, the average sum reward received by the

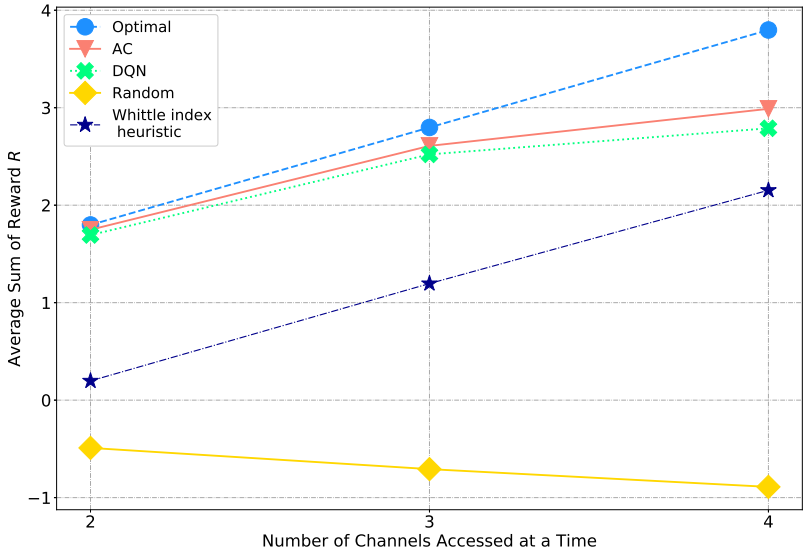
actor-critic agent is still slightly increasing, showing the capability of the actor-critic reinforcement learning algorithm in working with relatively large action spaces.

3.4.3 Average Reward in the Multi-User Case

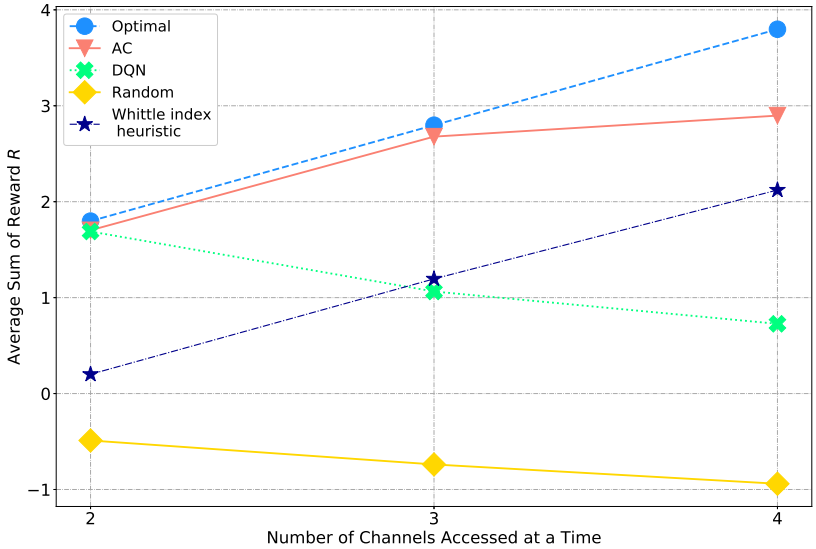
In this subsection, we provide simulation results for the multi-user scenario. As introduced in Algorithm 5, we propose a decentralized multi-agent framework to solve the problem, which allows each user to make its own decision. In this experiment, each user can only access one channel and is unaware of other users' decisions, meaning that there could be collisions. So, to maximize the reward, each agent is required to learn not only the channel switching pattern, but also the other users' channel access patterns to avoid collisions.

3.4.3.1 Multi-User Scenario without Priorities

First, we consider a scenario in which there are m users, where $m = 2, 3, 4$, and no priority is assigned to any user. We run the proposed actor-critic agent and the DQN agent, assuming that there are 16 channels with 6 good channels in each state within the switching pattern, and the switching probability is fixed at $p = 0.9$. As a reference, we also evaluate the performance of the optimal policy with known channel switching patterns, and the slotted-ALOHA where each user employs the random access policy independently. We again consider the average sum reward as the performance metric. As shown in Fig. 3.9, the optimal policy ensures that users avoid choosing the same channel in each time slot, and hence the optimal average sum reward is actually the same as that achieved in the case of a single user accessing multiple different channels. The averaged reward received by slotted-ALOHA keeps decreasing as the number of users increases, which means that in the slotted-ALOHA policy, the collisions cannot be effectively avoided. For the decentralized actor-critic multi-agent policy, the tendency of the performance is very similar to that of the AC



(a) 16 channels



(b) 32 channels

Figure 3.8: Average sum of reward vs. number of channels that can be accessed at a time.

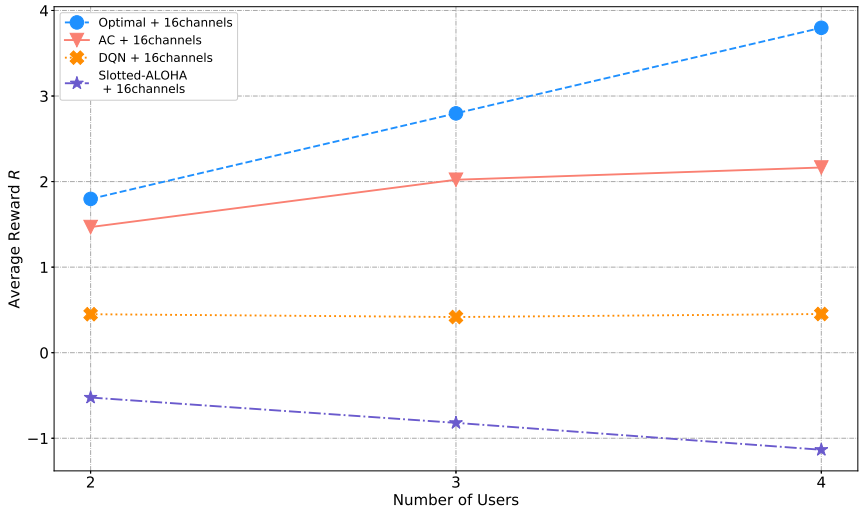


Figure 3.9: Average reward vs. number of users

curve shown in Fig. 3.8(a), but the values of the average sum reward are smaller, due to the absence of information on other users. As to the performance of the DQN agent, the average sum reward is rather low and varies only slightly when the number of users increases, indicating that the agent is not capable in this decentralized multi-user channel selection scenario.

3.4.3.2 Multi-User Scenario with Priorities

Now, we address the multi-user case where there are 3 users and 16 channels, and assume that one of these three users has higher priority than the other two. The user with the higher priority is referred to as the primary user, and the other two are secondary users. Again, we assume that there are always enough channels for users to transmit, however, some of the channels are more favorable compared to the others in the sense that they have improved channel conditions and have greater channel capacity. We refer to the channels that can provide better transmission quality as excellent channels, and the other available channels can again be in good or bad states. Hence, we now have an extended model in which the channels can be in one of the three states: *excellent, good,* and *bad*. The decentralized agents are expected

to be able to find the good channels and take advantages of the excellent channels.

To encourage the users to access the excellent channels, we assume that the reward for the excellent channels are doubled. To give the priority to the primary users, we assume that the reward received by the primary user will also be doubled regardless of whether the reward is positive or negative. In our experiments, we assume 2 excellent channels and 4 good channels in each channel state.

3.4.3.2.1 Primary User Sharing the Channel with Secondary Users In this part, we consider the scenario that the primary user will share the channels with the secondary users in the presence of a collision. Here, we assign each user an index, and the user 1 is chosen as the primary user. Then we record the channel access result of each user. Here, we mark the results using 5 labels:

- Excellent Channels: The user selects an excellent channel and occupies it alone.
- Collision in Excellent Channels: Two or three users access the same excellent channel.
- Good Channels: The user selects a good channel and occupies it alone.
- Collision in Good Channels: Two or three users access the same good channel.
- Bad Channels: The user selects a bad channel.

Fig. 3.10 and Fig. 3.11 present the users' channel access patterns based on the proposed actor-critic agents and the DQN agents in a period of 500 time slots, respectively. And we summarize the distribution (or equivalently the computed probabilities) of different channel access results in Table 3.1 and Table 3.2 for the two channel access frameworks, respectively. It is obvious that the proposed actor-critic is more competitive in selecting excellent channels and good channels. Also the lower probabilities of collisions at excellent and good channels indicate that the proposed

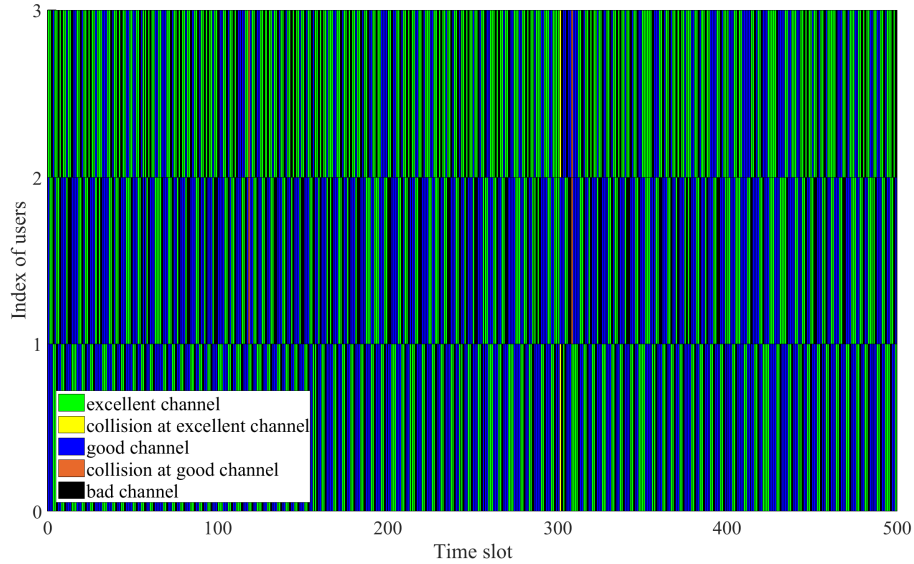


Figure 3.10: The channel selection results based on the decentralized actor-critic agents of all users over time in the case that the primary user shares the channel with secondary users in case of a collision.

Table 3.1: The distribution of different channel access results for decentralized actor-critic agents of all users over time in the case that the primary user shares the channel with secondary users in case of a collision.

User Index	Excellent Channels	Collision at Excellent Channels	Good Channels	Collision at Good Channels	Bad Channels
1	0.4240	0.0020	0.4720	0	0.1020
2	0.3140	0	0.4640	0.0040	0.2180
3	0.5040	0.0020	0.2840	0.0040	0.2060

framework is effective in learning other users' decision patterns to avoid collisions. As to the users' priorities, we find that the proposed actor-critic agents do not necessarily guarantee that the primary user occupies the excellent channels most of the time. One explanation is that even though the reward of the primary user is doubled, the other users still try to access the excellent channels to achieve their own maximum reward. Another reason is that the negative reward of the primary user is also doubled, and hence there is a chance that the primary user will be less aggressive to avoid such increased penalty. This is evidenced by the observation that the primary user attains the minimum probability of experiencing a bad channel.

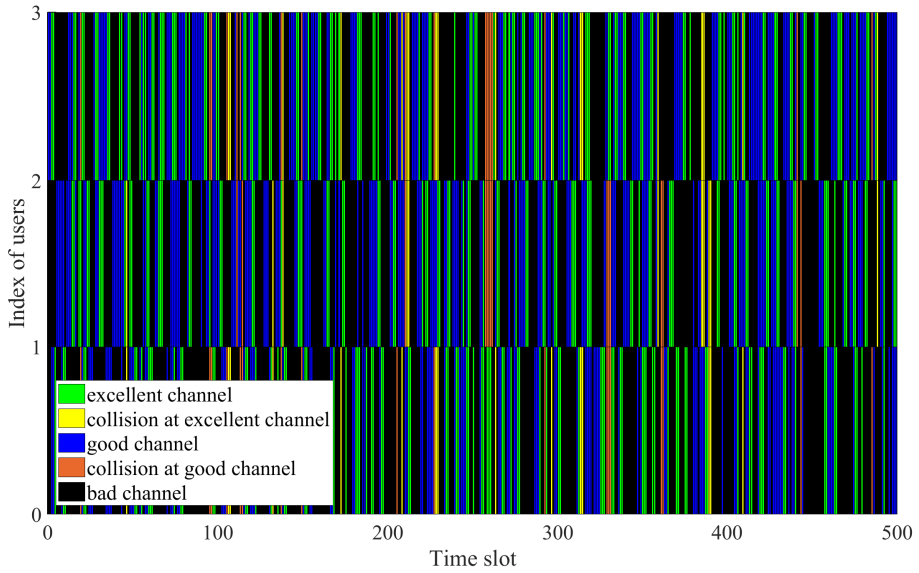


Figure 3.11: The channel selection results based on the decentralized DQN agents of all users over time in the case that the primary user shares the channel with secondary users in case of a collision.

Table 3.2: The distribution of different channel access results for decentralized DQN agents of all users over time in the case that the primary user shares the channel with secondary users in case of a collision.

User Index	Excellent Channels	Collision at Excellent Channels	Good Channels	Collision at Good Channels	Bad Channels
1	0.1900	0.0300	0.2340	0.0360	0.5100
2	0.1900	0.0220	0.3240	0.0300	0.4340
3	0.2360	0.0360	0.3300	0.0380	0.3600

3.4.3.2.2 Primary User Occupying the Channel Alone in case of a Collision In this part, we consider the case in which the primary user has the priority to occupy a channel when the secondary users also select it at the same time. Still, user 1 is assigned to be the primary user, and users 2 and 3 are the secondary users. In Figs. 3.12 and 3.13, we show the channel access results of all users based on the proposed actor-critic framework and DQN, respectively. And the Table 3.3 and Table 3.4 summarize the corresponding distribution of results. Since there will not be any collisions occurring from the perspective of the primary user, we have the following cases:

- Excellent Channels: The user selects an excellent channel and occupies it a alone.
- Good Channels: The user selects a good channel and occupies it alone.
- Collision with the Primary User: The secondary user selects the same excellent/good channel with the primary user.
- Collision with Secondary User: The secondary user selects the same excellent/good channel with the other secondary user.
- Bad Channels: The user selects a bad channel.

With the priority to occupy the channel alone in case of a collision, the probability that the primary user accesses an excellent /good channel is now increased. And from the distribution of the results, we notice that both the actor-critic and DQN polices can effectively enable the secondary user to avoid a collision with the primary user, because in Table 3.3 and Table 3.4, the probability of collision with the primary user is much lower than the probability of collision with a secondary user.

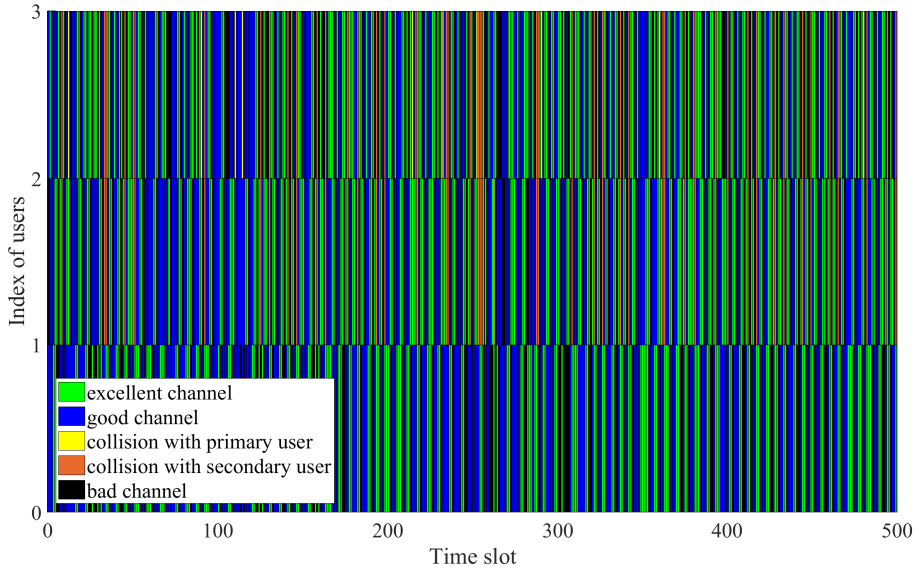


Figure 3.12: Channel selection results based on decentralized actor-critic agents of all users over time in the case that the primary user occupies the channel alone in case of a collision.

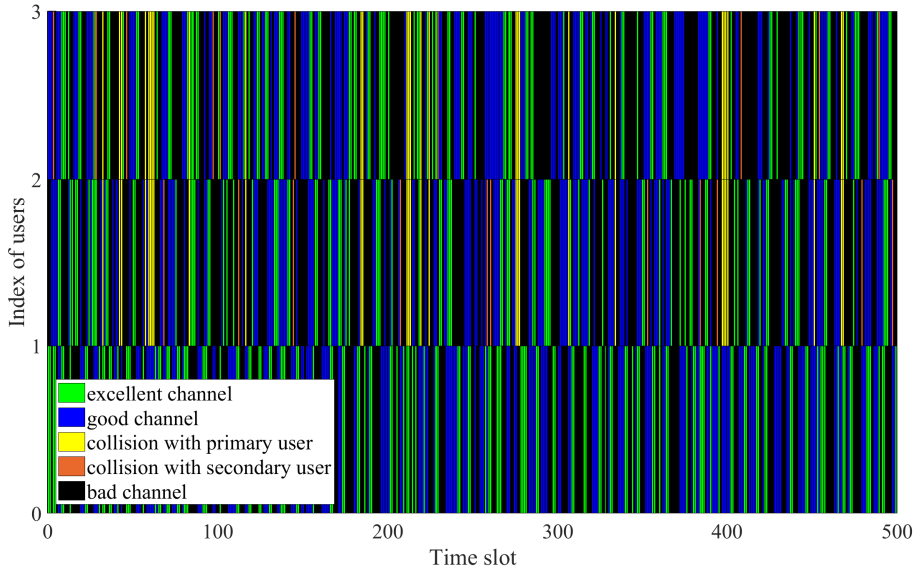


Figure 3.13: Channel selection results based on decentralized DQN agents of all users over time in the case that the primary user occupies the channel alone in case of a collision.

Table 3.3: The distribution of different channel access results for decentralized actor-critic agents of all users over time in the case that the primary user occupies the channel alone in case of a collision.

User Index	Excellent Channels	Good Channels	Collision with Primary User	Collision with Secondary User	Bad Channels
1	0.4420	0.4260	0	0	0.1320
2	0.3780	0.3720	0.0020	0.1000	0.1480
3	0.3580	0.3180	0.0200	0.1000	0.2040

Table 3.4: The distribution of different channel access results for decentralized DQN agents of all users over time in the case that the primary user occupies the channel alone in case of a collision.

User Index	Excellent Channels	Good Channels	Collision with Primary User	Collision with Secondary User	Bad Channels
1	0.2800	0.3520	0	0	0.3680
2	0.1780	0.2960	0.0220	0.0580	0.4460
3	0.2000	0.2960	0.0200	0.0580	0.4260

3.4.4 Time-Varying Environment

As discussed before, both the proposed actor-critic framework and the DQN framework introduced in [66] are reward-driven algorithms which can continually interact with the environment and update the policies. To illustrate the adaptive ability of the proposed framework, we have designed a time-varying environment, where at the beginning, the agent has been trained for pattern $\mathcal{P}_1$, and at time slot $t = 500$, the channel distribution changes to the second pattern $\mathcal{P}_2$, then at time slot $t = 1500$, the channel distribution changes back to pattern $\mathcal{P}_1$. In this process, both change points are unknown to the agent. The experiment was conducted with a fixed switching probability $p = 0.9$, and arbitrary switching order where 32 channels are grouped into 8 subsets randomly and each subset has 4 perfectly correlated channels.

The re-training process is shown in Fig. 3.14 in terms of the reward averaged over every 500 accessing decisions. Considering the learning rate in the actor-critic framework decays as the training process goes by, and the learning rate will influence time needed for the re-training process, we in this experiment test this framework in two different settings: for AC agent I, we allow the agent to reset the learning rate to

the initial value when the agent receives negative average reward; and for AC agent II, as a reference, the learning rate will always decay over time. Before the experiment, all agents are well trained and extra time slots are taken to make sure that the learning rates in AC agents are smaller than initial values. Then we set the time when we start the observation as $t = 0$. When the channel state switching pattern changes at $t = 500$, the average reward achieved by both actor-critic framework and DQN drops to negative values suddenly. And then, in the following re-training process, the policies get updated and adapt to the pattern $\mathcal{P}_2$, and as a result, the average rewards gradually increase and reach to the previous levels before the second change point. In the first re-training process, due to the difference in learning rate, the AC agent I is quicker to learn the pattern $\mathcal{P}_2$ while the AC agent II is slower but experiences slightly less fluctuations in terms of the average reward. Both AC agents eventually perform as well as before the first change point. Comparing the time duration it takes for the agent to get back the previous level and the performance after the first re-training process, we conclude that our proposed framework is very competitive in terms of the adaptive ability, though the actor-critic structure which has two separate neural networks takes slightly more time to converge. We also observe that the DQN agent attains an average reward level that is less than before the first change point. Then, when the second change in the channel pattern occurs, we observe that the rewards of all three agents drop to negative values similarly as after the first change point. However, at this time, since the channel distribution changes back to pattern $\mathcal{P}_1$, which has been learned by the agents, the AC agents are able to reach the previous performance level over a shorter duration compared to after the first change point. On the other hand, for the DQN agent, the duration of the second re-training process is longer than that of the first one and it can hardly perform as before the first change point, which indicates that the DQN agent is less competitive in switching between channel patterns.

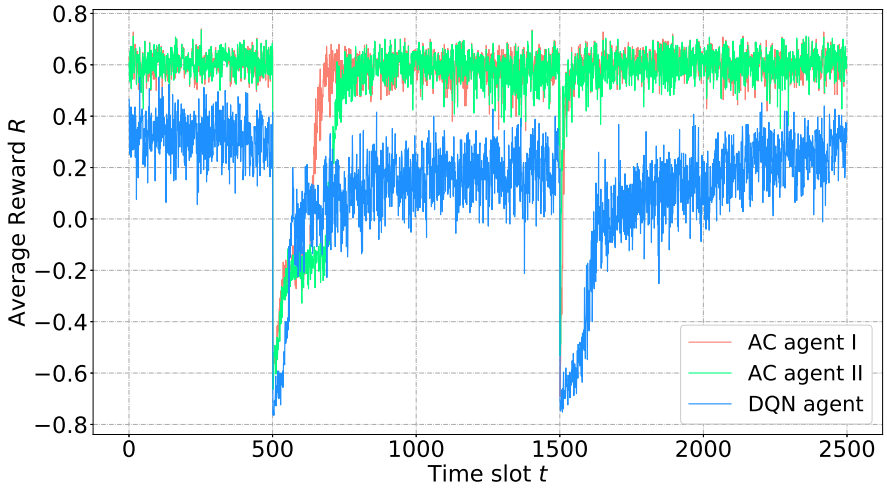


Figure 3.14: The re-training process in a time-varying environment with the change points at $t = 500$ and $t = 1500$.

Table 3.5: The runtime needed for each channel access decision

number of channels	AC agent	DQN agent	% reduced
16	0.002428	0.025381	90.4328
32	0.003998	0.030833	87.0340
64	0.004002	0.059308	93.2527

3.4.5 Study of Runtime

To meet the real-time requirements, the channel access decisions must be made quickly. To highlight the efficiency of the actor-critic framework, we have computed the average runtime needed for making one decision and compared it with that needed in the DQN framework. Table 3.5 shows the runtime for one decision needed by the actor-critic (AC) agent and the DQN agent for the case of having a single good channel out of N channels in total, where $N = \{16, 32, 64\}$.

The proposed actor-critic framework is actually more complicated in architecture because it has two neural networks and hence has more parameters to update. But we only pass one actor to the critic, so that the critic requires less computational resources. Another important reason why our framework can have significant savings in the runtime is that we do not need to replay any experience because the LSTD of the critic network is enough to ensure that the actor policy is updating in the

correct direction, while the DQN proposed in [66] replayed 32 samples for each time of updating to make the channel access policy stable. For the current number of channels and users, the second reason for the substantial improvements in runtime is that the action space is limited. But once that action space increases, as the number of channels increases, the first reason will become more significant.

Indeed, to demonstrate the impact of memory replay, we briefly discuss the computational complexities of AC and DQN agents next. For the actor-critic network, let us assume that the number of neurons in each layer i of actor network is a_i, and the number of neurons in each layer j of critic network is c_j, and there are A layers in actor network and C layers in critic network. We further assume that the input size is K. In each iteration, the number of calculations at neurons is $(Ka_1 + \sum_{i=1}^{A-1} a_i a_{i+1}) + 2 \cdot (Kc_1 + \sum_{j=1}^{C-1} c_j c_{j+1})$.

For the DQN, we assume that the number of neurons in each layer g if d_g, and there are D layers in total. Also, we suppose that the minibatch size is M. With the same input size K, the number of calculations of DQN is $M(Kd_1 + \sum_{g=1}^{D-1} d_g d_{g+1})$.

If we assume that the actor network has the same size as the DQN, and the critic network has the same size except for the output layer (the size of critic output layer is fixed to be 1, and the size of actor network and DQN output layer are fixed to be the number of actions), then the ratio of computational complexity between actor-critic network and DQN is approximately $3/M$, where the typical values of M are 16, 32, 64, and for some cases it can be even greater. Therefore, we conclude that not replaying the minibatch is a important reason that can explain the high time efficiency of actor-critic. Also, when the DQN replays the minibatch sample, the time consumption for importing the data is also nonnegligible.

Chapter 4

Adversarial Jamming Attacks on Deep Reinforcement Learning Based Dynamic Multichannel Access

In this chapter, we propose two adversarial policies, one based on feed-forward neural networks (FNNs) and the other based on deep reinforcement learning (DRL) policies. Both attack strategies aim at minimizing the accuracy of a DRL-based dynamic channel access agent that is proposed in [84]. We first present the two frameworks and the dynamic attack procedures of the two adversarial policies. Then we demonstrate and compare their performances. Finally, the advantages and disadvantages of the two frameworks are identified.

4.1 Dynamic Channel Access Policies of the Victim User

In this section, we introduce the background on dynamic multichannel access. As noted above, we consider an actor-critic DRL agent proposed in [84] as the victim user to be attacked.

4.1.1 Channel Switching Pattern

In the considered dynamic multichannel access problem, the time is slotted and the user selects one channel to access at the beginning of each time slot. We assume that the state of each channel switches between good and bad in a certain probabilistic pattern. When the channel is in good condition, the user can transmit data successfully. Otherwise, a transmission failure will occur. We also assume that the channel switching pattern can be modeled as a Markov chain, and in each state of which, there are k out of the N channels in good condition. At the beginning of each time slot, the channel pattern can either switch to the next state with probability of ρ, or remain to be the same as the state in the last time slot with probability of $(1 - \rho)$. In Fig. 4.1, we display a round-robin switching pattern with two out of 16 channels being good in each time slot and each channel has the same probability to be in good state.

4.1.2 Actor-Critic Agent

It is assumed that the channel switching pattern is unknown to the user, and the user can only observe the channel selected in the current time slot. Hence, the multichannel access is a partially observable Markov decision process (POMDP). To help the user to access the good channels as frequently as possible under such conditions, we proposed in [84] an actor-critic deep reinforcement learning based agent to make the channel

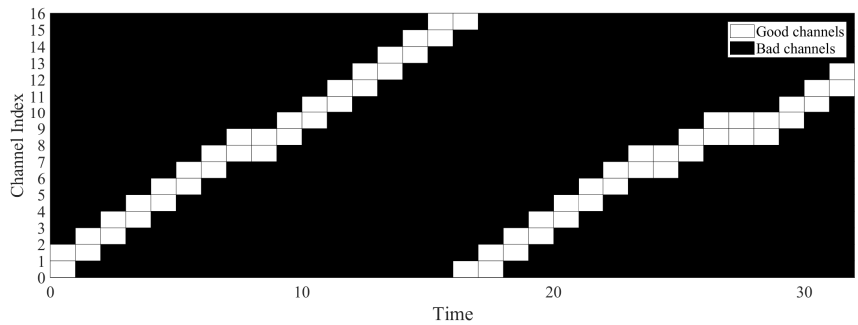


Figure 4.1: Round-robin switching pattern when two of the 16 channels is in good condition and the switching probability is $\rho = 0.95$. The channel in good state at a given time is indicated by white squares.

access decisions in each time slot.

The proposed agent is designed to learn the channel switching pattern through past decisions and the corresponding feedback from the channels. We assume that, at time t, the channel state can be denoted as $\mathtt{X}_t = \{\mathtt{x}_1, \mathtt{x}_2, ..., \mathtt{x}_N\}$, where N is the total number of channels, $\mathtt{x}_i$ stands for the state of the ith channel. For each channel i, where $i = 1, 2, ..., N$, we have $\mathtt{x}_i = 1$ if the channel is in good state, or $\mathtt{x}_i = 0$ if the channel is in bad state. And each time the agent senses a channel, the state of the sensed channel is revealed to be either good or bad. Therefore, we define the reward (feedback) as follows: if a good channel is chosen, the reward r_t will be $+1$; otherwise, the reward r_t will be -1.

The agent's observation can be denoted as $O_t = \{o_1, o_2, ..., o_N\}$, where N is the total number of channels. If channel i, $i = 1, 2, ..., N$, is chosen, the agent senses it and learns its state, so we define $o_i = r_t$; otherwise, the agent will record $o_i = 0$. The agent will learn on the basis of its previous experience. We assume that the agent keeps an observation space $\mathcal{O}$ that consists of the most recent M observations. The observation space is initialized as an all-zero $N \times M$ matrix, and at each time t, the latest observation O_t will be added to the observation space, and the oldest observation O_{t-M} will be removed. The updated observation space $\mathcal{O}$ at time $t+1$ can be denoted as $\mathcal{O}_{t+1} = \{O_t; O_{t-1}; ...; O_{t-(M-1)}\}$.

Next, we consider a discrete action space denoted by $\mathcal{A} = \{1, 2, ..., N\}$, where N is the total number of channels. Each valid action in the action space describes the index of the channel that will be accessed. Hence, when an action is chosen, the agent will access the corresponding channel and receive the reward which reveals the condition of the chosen channel. The agent can only choose one channel to sense/learn in each iteration. The aim of the agent is to find a policy π, which maps the observation space $\mathcal{O}$ to the action space $\mathcal{A}$, that maximizes the long-term expected reward R of channel access decisions:

$$\pi^* = \arg\max_{\pi} R$$

where π^* denotes the optimal decision policy, and in a finite time duration T, we express R as

$$R = \frac{1}{T} \sum_{t=1}^{T} r_t.$$

And according to the definition of R, we have $R \in [-1, 1]$.

4.1.3 Performance in the Absence of Jamming Attacks

We consider the channel switching pattern shown in Fig. 4.1, and evaluate the accuracy of the good channel access by the user with $N = M = 16$. The evaluation is performed in the absence of any jamming attacks and after the DRL agent is well trained. In Fig. 4.2, we test the model in two cases. First, we consider the ϵ-greedy policy with $\epsilon = 0.1$, with which the user accesses a random channel with probability 0.1, and chooses the channel selected by the reinforcement learning policy with probability 0.9. Note that the ϵ-greedy policy allows the model to access bad channels by chance during exploration. In addition, we also consider the case in which ϵ is set to 0 to identify the performance of the pure DRL policy. We note that ϵ-greedy policies with $\epsilon > 0$ are generally employed to enhance the DRL agent's ability against changes in the channel patterns, as will be discussed in detail in Section V. We observe in the

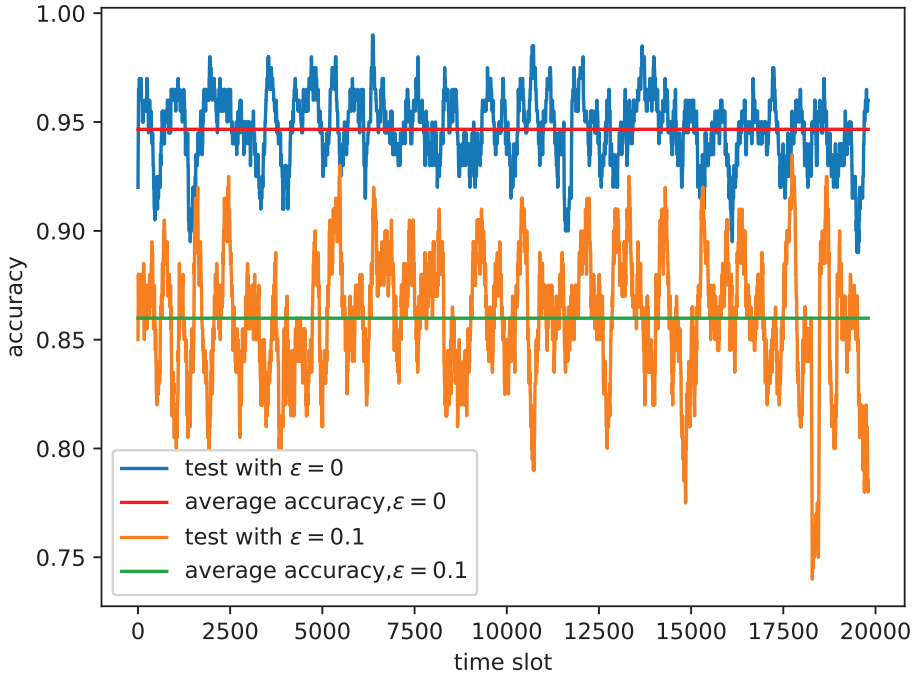


Figure 4.2: Accuracy of the good channel access in the absence of jamming attacks.

figure that high average accuracies (higher than 85% and around 95% with $\epsilon = 0.1$ and $\epsilon = 0$, respectively) are attained in the absence of jamming attacks.

4.2 FNN Jamming Attacker

In this section, we analyze the FNN method to perform jamming attacks on the actor-critic DRL dynamic multichannel access agent described in Section 4.1.2. A presumptive attacker is able to choose and jam a single channel in each time slot to significantly reduce the selection accuracy of the actor-critic agent. We assume that the attacker employs a feed-forward neural network (FNN) to make the decision on which channel to attack.

4.2.1 Initial FNN Model

We build the FNN with TensorFlow as the attacker model. To collect initial training data for this FNN, we assume the attacker has another actor-critic agent which has a similar performance as the victim model does. These two models do not necessarily have the same parameters, as we need to retrain the initial FNN before attack. From this attacker actor-critic agent, we obtain the channel selection during 53 consecutive time slots as training data for FNN.

The FNN model feeds on 3 previous channels as input, and gives the probability among each channel in the next time slot as output. It has 2 hidden layers with 16 hidden neurons in each layer, with sigmoid activation function, RMSProp optimizer, and mean squared error loss function. With 50 sets of 3-previous-1-future data pairs from attacker's actor-critic agent, we set random weights in the FNN, and run 4000 iterations to train the FNN model (hereinafter referred to as the initial FNN), which has 88% accuracy on extra testing data. We intentionally limit the amount of training data and iterations to avoid overfitting to the initial policy of the victim actor-critic agent, which will greatly change under attack.

4.2.2 Channel Observation and Record

Before the attacker starts the jamming attack with FNN, it observes one channel and the reward of the victim user to determine if its attack is successful. The attacker also records the history of channel selection as input to the FNN to predict the next attacking choice. However, if the attacker simply records the attacked channel, once FNN misses to predict the victim user's channel selection, the attacker will lose track of the victim, and it will take some time to accurately predict the victim's chosen channel again. Thus, we suggest an alternative strategy, utilizing the initial FNN as a good channel detector. This initial FNN always keeps the initial parameter, and thus it is different from the adapting FNN which is affected by victim policy during

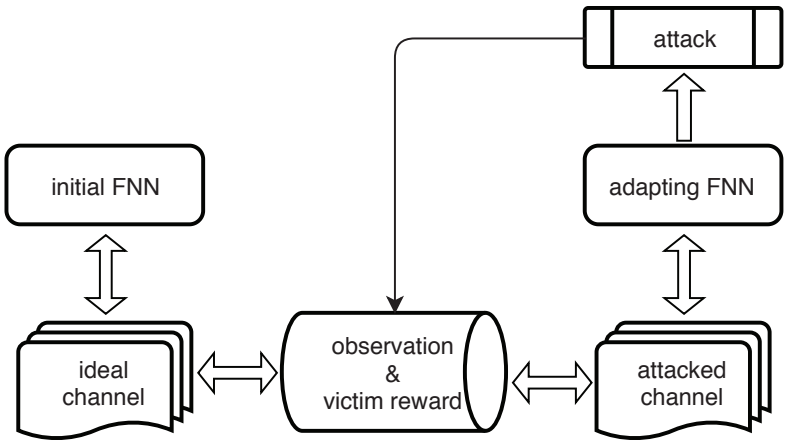


Figure 4.3: The diagram of the history records and FNNs

the dynamic attack, as we will describe in Section 4.2.3.

As depicted in Fig. 4.3, we use two sets of history records and FNNs. First, the initial FNN keeps its record of "ideal channel" and makes its own prediction. Since the initial FNN imitates the high-performance victim, the ideal channel generated by initial FNN is close to the good channels. Second, the adapting FNN keeps another set of "attacked channel" record and decides which channel to attack in the next time period. The observation of the channel and reward determines the next channel to observe, and which channel to enter in both channel records. This is explicitly explained in Algorithm 6 below.

4.2.3 Dynamic Attack

Based on the attacked channel record, we can use FNN to perform real-time jamming attacks against the victim. Although the initial FNN works well with the original policy of the victim actor-critic agent, it is not as accurate when the victim adapts to the attack with a new policy. As a control problem, there are two major considerations.

On the one hand, attacker FNN needs to retrain. When the attacker jams one of the good channels that the victim tends to choose, the victim will have low accuracy for the first few thousand time slots. After that, as the victim's actor-critic agent

Algorithm 6 The loop of predicting, recording and observing

```
during each time slot:
predict using adapting FNN, and attack the channel
record "attacked channel" as adapting FNN prediction
predict using initial FNN
record "ideal channel" as initial FNN prediction
if last time "attacker success" then
    observe the last different attacked channel
else if last time "observed success" then
    observe ideal channel
else if last time "failed" then
    observe the last observed channel
end if
if victim is observed and reward is positive then
    record observation as ideal channel and attacked channel
    mark as "observed success"
else if reward is negative then
    record observation as attacked channel
    mark as "attacker success"
else
    record attacked channel as ideal channel
    mark as "failed"
end if
```

adapts to the attack, it learns a new policy to find the good channel and at the same time, mislead the attacker's FNN attack. Thus, the attacker should retrain the initial FNN (instead of starting with random weights), and attack with this retrained FNN to adapt the new victim policy.

On the other hand, attacker FNN needs to stop the attack, and retrieve the initial FNN parameters before retraining occasionally. If the attacker keeps retraining FNN, the victim accuracy would still recover gradually. There are two reasons for this. First, the parameters of FNN deviates from the initial FNN during long-term retraining, and lose the basic features (for example, taking the difference between channel values). This means that the attacker sets the FNN parameters to their initial values, and retrains to fit to the current victim policy. Second, if the attacker keeps on attacking, the data for retraining would reflect the setting in which the victim is under attack and is operating with low accuracy of good channel access. This prevents the attacker FNN from learning the desired victim pattern. One way to solve this problem is to stop attacking when the victim accuracy begins to recover, so the victim will converge fast to a stable policy with high accuracy. Then the attacker can retrieve, retrain using observations from this converged policy and perform better and more accurate attacks. Another benefit is that the attacker will not stay long in the recovering stage, where the victim average accuracy is up to 50% (which is much higher than the desired accuracy), so that the attacker can significantly reduce the overall average accuracy.

Therefore, we develop a retrieve-retrain-attack-stop (RRAS) procedure as depicted in Fig. 4.4 to perform dynamic attacks. At time ① shown in Fig. 4.4, we start the initial attack with the initial FNN, which is guaranteed to perform well at first. Then, the attacker will gradually lose control of the victim as it adapts to the initial attack. At time ②, the victim accuracy grows up to a lower threshold, so the attacker gives up attacking, and lets the victim recover fast from the initial attack, to reduce the

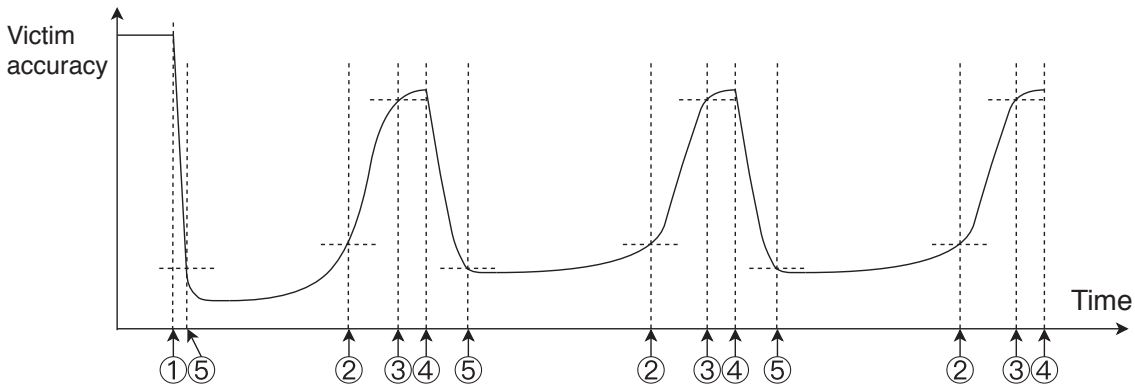


Figure 4.4: Retrieve-retrain-attack-stop procedure of dynamic attack: ① initial attack ② stop attack ③ retrieve parameters and start retrain ④ stop retrain and start attack ⑤ stop updating the model (only for the DRL attacker)

time span between ② and ③ and reach a higher accuracy threshold at time ③. At time ③, the attacker retrieves the initial FNN parameters and collects the retraining data until time ④. Finally, the attacker initiates another attack at time ④, and the entire procedure is repeated as depicted in Fig. 4.4.

4.3 DRL Jamming Attacker

In this section, we introduce an actor-critic deep reinforcement learning (DRL) based agent to perform the jamming attack on the aforementioned victim user without having any prior information about the channel switching pattern or the victim's action policy. The DRL attacker is also assumed to observe only one channel in each iteration. Different from the FNN attacker, however, we assume that the DRL attacker is able to observe the victim's interaction the environment for a period of time that is sufficiently long for the DRL attacker to learn the activity pattern.

4.3.1 Actor-Critic Model

In Fig. 4.5, we show the diagram of the actor-critic structure and the DRL attacker-environment interactions. The actor-critic structure consists of two neural networks,

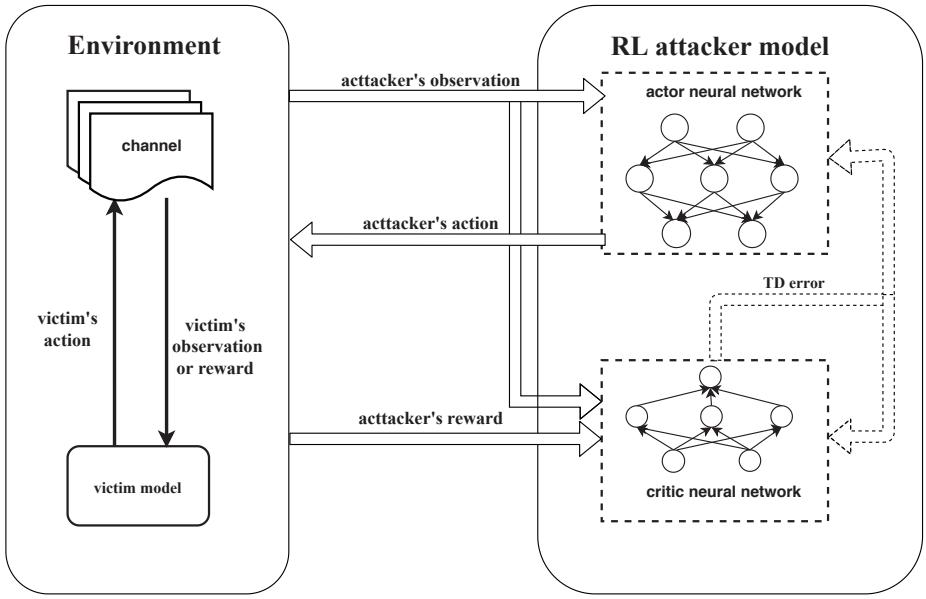


Figure 4.5: Diagram of actor-critic structure and DRL attacker-environmental interactions.

namely the actor network and critic network. The channels and victim's channel selection model form the environment to be observed by the attacker. Each time, after the DRL attacker observes the environment, an action will be selected based on this observation by the actor neural network. Then, the reward and the new state of the environment after executing the chosen action will be sent to the critic neural network to calculate the temporal difference (TD) error. This TD error will be used to update both critic and actor neural networks. When the update of network is completed, the DRL attacker model is ready to make the next decision.

At the beginning of each time slot t, the DRL attacker can select one channel based on its own action policy learned by the actor-critic neural networks. The action of the DRL attacker and the victim at time t are denoted as a_t^A and a_t^V respectively. Since both the DRL attacker and the victim select one out of the N channels, the sizes of their action spaces are the same. We assume that there are proper mechanisms and measurements (such as SINR levels, ACK signals) through which the attacker learns if the victim has selected the same channel as the attacker itself, i.e., $a_t^A = a_t^V$, and

if the victim has transmitted successfully. The goal of the attacking DRL agent is to learn the victim's activity pattern so that it can jam the channels selected by the victim as much as possible. Based on this objective, we define the reward of the DRL attacker at time t as

$$r_t = \begin{cases} +1 & \text{if } a_t^A = a_t^V \text{ and victim selects a good channel,} \\ +0.5 & \text{if } a_t^A = a_t^V \text{ and victim selects a bad channel,} \\ -0.5 & \text{if } a_t^A \neq a_t^V \text{ and victim selects a bad channel,} \\ -1 & \text{if } a_t^A \neq a_t^V \text{ and victim selects a good channel.} \end{cases} \tag{4.1}$$

Within this setting, the DRL agent is encouraged to select the same good channels as the victim as its first priority. We also consider the case in which the attacker and victim select the same bad channel as partial success in terms of jamming.

As mentioned before, the DRL agent has no knowledge about the channels and the victim user. Hence, from the perspective of the DRL agent, the channels and victim form an unknown environment. We assume that in each time slot t, the observation of the DRL attacker is denoted as $\mathbf{S}_t = \{s_{t,1}, s_{t,2}, \ldots, s_{t,N}\}$. Then each element $s_{t,i}$, for $i = 1, 2, \ldots, N$, stands for the observation on the ith channel at time t. As assumed before, the DRL attacker can only choose one channel at a time, so we have

$$s_{i,t} = \begin{cases} r_t & \text{if the } i^{\text{th}} \text{ channel is selected in time slot } t, \\ 0 & \text{if the } i^{\text{th}} \text{ channel is not selected in time slot } t. \end{cases} \tag{4.2}$$

Above, 0 indicates that the corresponding channels are not selected and therefore there is no information on these channels.

4.3.2 Operational Modes

Once the DRL agent is initialized, it switches between two different modes: listening mode and attacking mode.

- *Listening mode:* In this mode, the DRL agent only observes the environment and updates its own policy based on the reward, but does not jam the selected channels so that the victim is not influenced and updates to a new policy.
- *Attacking phase:* In this mode, the DRL agent jams the selected channels and decides whether to update its neural networks based on the victim's performance. When the victim performs well, the DRL agent should evolve its policy as the victim gradually adapts to the attacker's influence. However, when the victim performs poorly, the DRL agent should stop learning from the reward. Because in this situation the victim frequently chooses the bad channels, and the reward may misguide the attacker.

We assume that the victim's model is pre-trained so that the victim's activity pattern is stable when the attacker starts to train its own neural networks. In this training phase, the DRL agent works in the listening mode. And when the DRL agent is well trained, it can start the dynamic attack which we describe in detail in the following subsection.

4.3.3 Dynamic Attack

Similar to the FNN attacker, the DRL attacker also uses the RRAS procedure shown in Fig. 4.4. We note that the DRL agent requires less prior information about the victim' activity pattern than the FNN attacker. However, due to the differences in the learning method, the DRL attacker needs to observe the victim over a longer period to train a reliable policy. DRL attacker also aims at avoiding the situation in

which the victim learns a totally new action policy once the model is well trained. For this purpose, the duration of each cycle of the DRL attacker is fixed at a certain value that prevents the victim to update to a new policy.

As shown in Fig. 4.4, the DRL attacker also starts its first attack at time ① when the victim model has been working in a stable fashion and working well. Before this point, the DRL attacker works in listening phase to learn the victim's activity pattern, and we assume that at time ①, the DRL attacker can also function well with high stability. Once the attack is initiated, the performance of the victim drops rapidly. In this process, the victim keeps updating its model to overcome the influence of the attacks, and at the same time, the attacker also keeps updating its model to adapt to the victim's changing policy. However, we should note that the attacker is always encouraged to choose the same channel as the victim does. Hence, when the victim is forced to explore other channels which are not attacked in order to find a new policy to counteract against the attacks, it cannot avoid but try bad channels in order to find the good ones. From the perspective of DRL attacker, there is no need to follow the victim's selection because the victim's model updates dramatically and the policy may perform worse initially. On the one hand, it is difficult for the attacker to learn an unstable policy. On the other hand, copying the bad policy may give victim the chance to recover its performance. Based on this idea, the DRL attacker stops updating when the performance of the victim is lower than a threshold and we mark this time instant as time ⑤. Though the DRL attacker model stops learning, it still works in attacking mode, so the performance of the victim continues to decrease. As mentioned before, the DRL attacker should stop jamming the channels before the victim adapts to its attacks, because the victim is also a reinforcement learning agent that has the ability to act against attacks naturally. At time ②, the victim's performance starts to recover, meaning that a new policy is being formed in the victim model. To avoid pushing the victim to the new policy further, the DRL attacker needs

to switch to the listening mode at time ② to encourage the victim to return to its old policy as quickly as possible. And at time ③, the victim is able to perform as well as that before the attack, and the DRL attacker will retrieve the initial model and keep working in listening mode to adjust its policy based on the victim's activity until time ④ when the DRL attacker switches to attacking mode and starts a new cycle. In our implementation, the duration of each cycle is fixed to 2000 time slots, and the gap between time ③ and ④ is fixed at 200 time slots. Also, in the experiments, the duration between time ② and ③ is very small.

4.4 Experiments

In this section, we test the proposed FNN attacker and DRL attacker with a well-trained victim model and channel pattern introduced in Section 4.1. In the following experiment, the FNN attacker starts attacking at time slot $t = 0$, and the DRL attacker starts attacking at time slot $t = 2000$.

First, we test both the FNN attacker and DRL attacker under the condition that the victim model works with $\epsilon = 0$ to show its full power. In Figs. 4.6 and 4.7, we plot the victim's accuracy over time to show the attackers' performance. For the FNN attacker, the victim's model crashes after about 5000 time slots and never recovers which means the victim's DRL agent has failed to adapt to the FNN attacker when it is not trying any random channels (due to the fact that $\epsilon = 0$). And for the DRL attacker, the victim's policy crashes immediately after the DRL attacker starts jamming the channels at time slot $t = 2000$. However, the victim's policy can recover for a short period of time after a few thousands of time slots. We should note that as a reinforcement learning-based agent, the DRL attacker always works with an ϵ-greedy policy with $\epsilon = 0.1$. The randomness in the DRL attacker's policy leads to a small chance for the victim to recover its performance from time to time. Overall, it is not

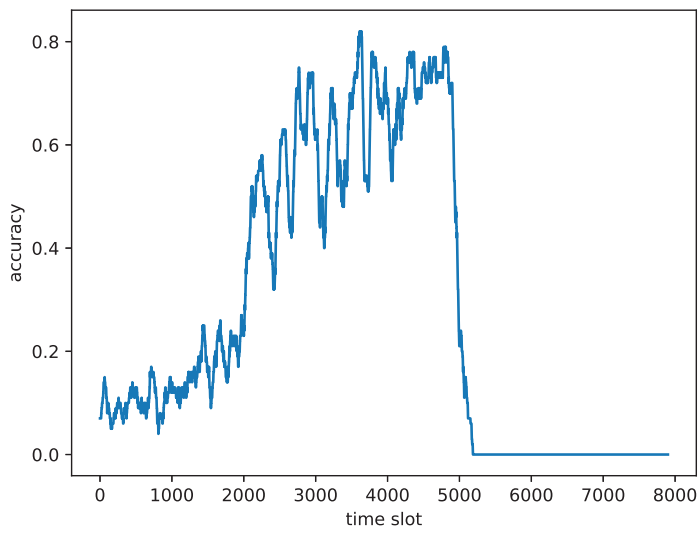


Figure 4.6: Victim's accuracy under FNN attacker's RRAS procedure. The victim works without a ϵ-greedy policy.

challenging for the proposed to attackers to jam the channels selected by the victim most of the time, and considering this, we test the victim model with $\epsilon = 0.1$ in the following experiments to show the performance of the proposed attackers facing with a stronger victim user.

In Figs. 4.8 and 4.9, we plot the accuracy of the victim under FNN attacker's RRAS procedure and DRL attacker's RRAS procedure respectively. The FNN attacker retrieves and retrains when the victim's accuracy reaches 80% , and stops attacking when the victim's accuracy goes below 40%. Each retraining takes 100 samples, and runs over 850 iterations. Under the FNN attacker's RRAS procedure, the victim's accuracy drops rapidly after the attack begins. For the FNN attacker's initial attack cycle, the victim's performance recovers slowly. After that, the victim's performance can recover quickly when the attack is stopped and drops sharply once the attack resumes after retraining in each of the following RRAS cycles. This means that the FNN can make the victim perform poorly as much as possible in each RRAS cycle. The DRL attacker stops updating the policy when the victim's accuracy is

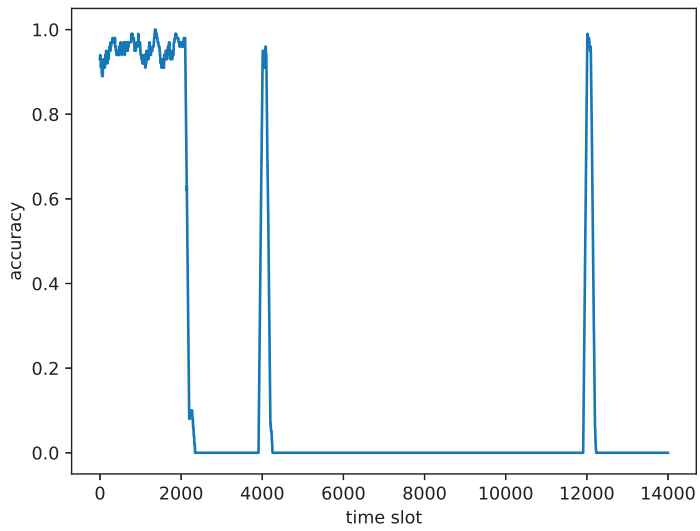


Figure 4.7: Victim's accuracy under DRL attacker's RRAS procedure. The victim works without a ϵ-greedy policy.

lower than 30% and switches to the listening mode when the victim's accuracy recovers to higher than 30% or if the duration of the current cycle is longer that 2000 time slots. In the listening mode, the DRL attacker reloads its initial policy and retrains for 200 time slots before the next attacking mode begins. In Fig. 4.9, the DRL attacker is able to have the victim's performance drop substantially and the recovery occurs over a short period of time but the performance drops again significantly, which means that the victim operates with very low accuracy most of the time. We note that under the DRL attacker's RRAS procedure, the victim's accuracy is more effectively constrained at a lower level.

To further compare the FNN and DRL attackers, we plot the corresponding probability density function (PDF) and cumulative distribution function (CDF) of the moving average of victim's accuracy in Figs. 4.10-4.13 based on the accuracy curves shown in Figs. 4.8 and 4.9. Note that, the PDF and CDF under DRL agent's attacks starts collecting the accuracy data starting from the initial attacking phase at time $t = 2000$.

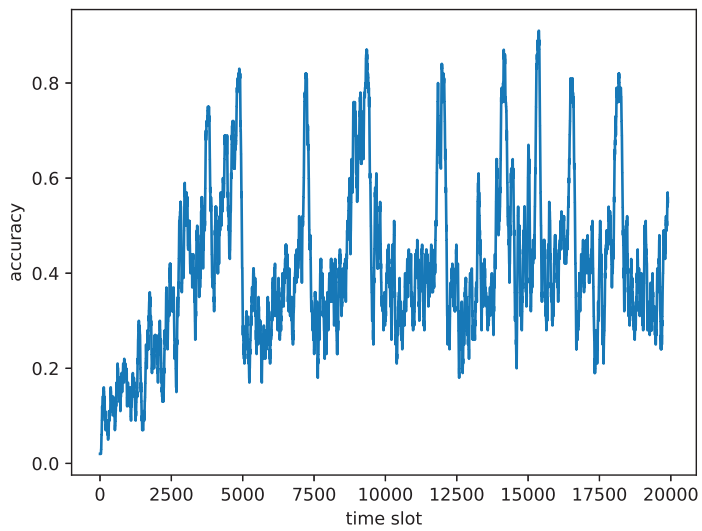


Figure 4.8: Victim's accuracy under FNN attacker's RRAS procedure. The victim works with $\epsilon = 0.1$.

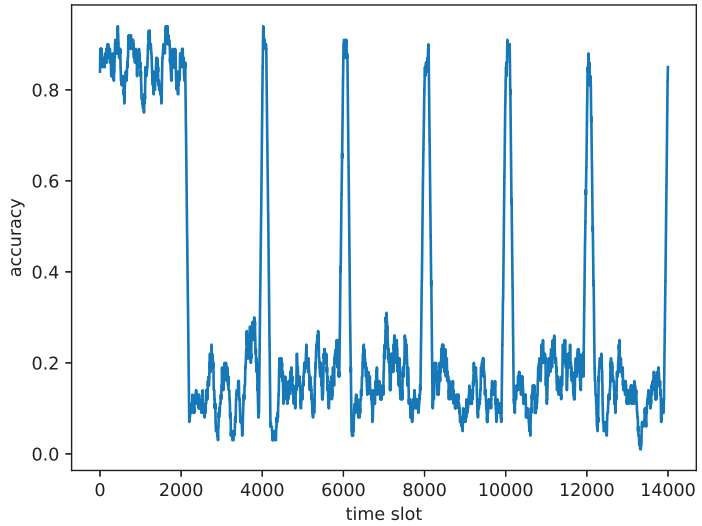


Figure 4.9: Victim's accuracy under DRL attacker's RRAS procedure. The victim works with $\epsilon = 0.1$.

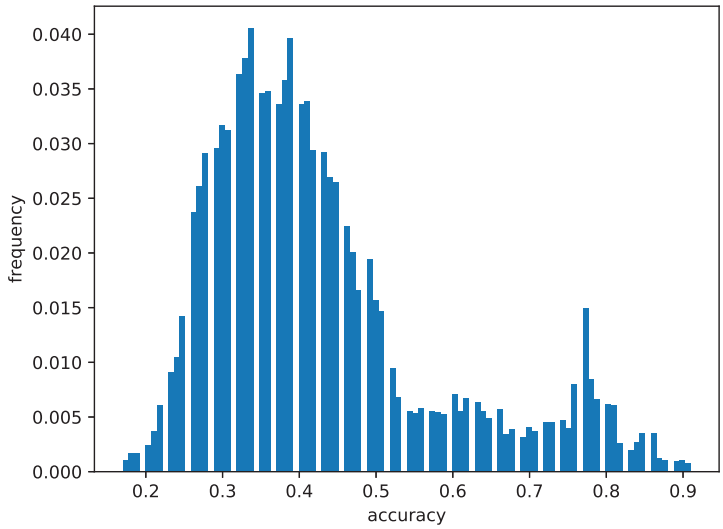


Figure 4.10: PDF of victim's accuracy under FNN attacker's RRAS procedure.

In Fig. 4.10, we observe that with the FNN attacker's jamming attacks, the victim's accuracy is more concentrated in the range of (0.3, 0.4). Correspondingly, in Fig. 4.11, the CDF increases at the fastest rate and approximates to 80% when the accuracy is 50%. For the DRL attacker, the victim's accuracy is highly concentrated at the level of 0.1 as shown in Fig. 4.12, and the corresponding CDF in Fig. 4.13 exceeds 80% when the accuracy is 20%. Since both proposed attackers stop attacking under specific conditions, the victim is able to recover its accuracy periodically. Hence, we can observe the increased distribution of the victim's accuracy at about 80% under both types of attacks.

As analyzed above, the DRL attacker can perform more effectively in the experimental environment. Additionally, the DRL attacker does not require any other auxiliary neural network as the FNN attacker does. However, if we consider the difference in the information regarding the victim-environment interactions required by these two attacker, we note the advantage of the FNN attackers. The FNN attacker only needs to obtain the victim's activity records for a short period of time (50 or 100 time slots) and repeat the learning of the records over thousands of iterations to

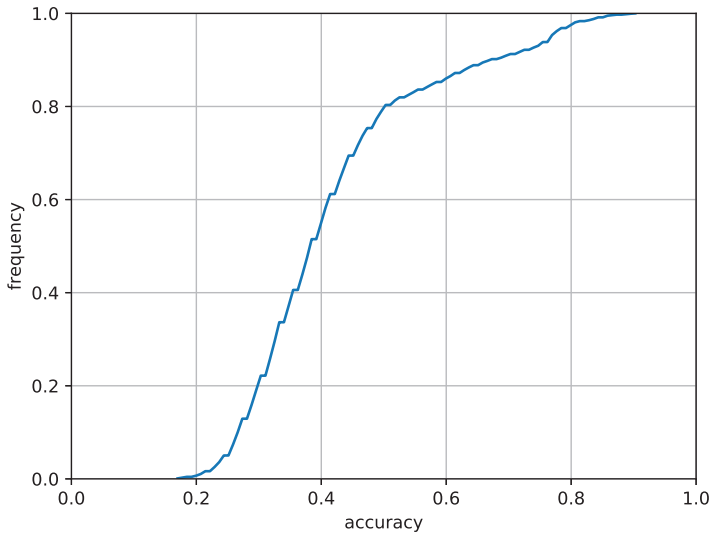


Figure 4.11: CDF of victim's accuracy under FNN attacker's RRAS procedure.

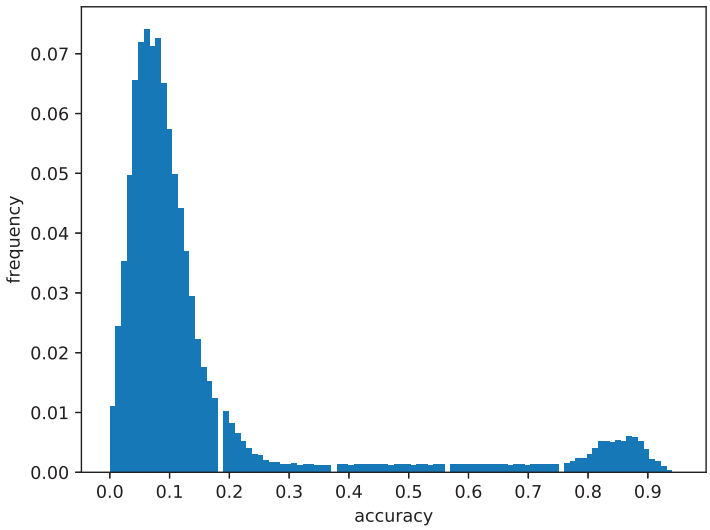


Figure 4.12: PDF of victim's accuracy under DRL attacker's RRAS procedure.

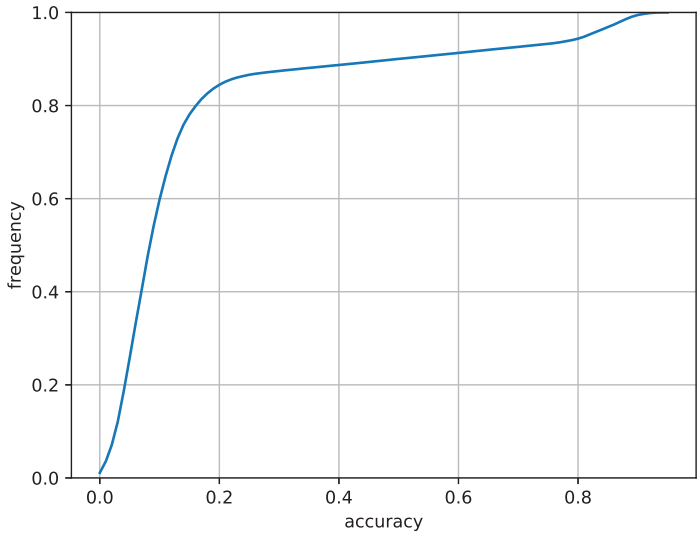


Figure 4.13: CDF of victim's accuracy under DRL attacker's RRAS procedure.

train and retrain its policy. However, the DRL attacker has to observe the victim-environment interactions for about 10000 time slots to train a stable policy after initialization. Therefore, if the channels patterns vary suddenly, the FNN is more promising in terms of adapting to a new policy quicker than the DRL attacker.

Chapter 5

Deep Actor-Critic Reinforcement Learning for Anomaly Detection

In this chapter, we study deep reinforcement learning based active sequential testing for anomaly detection. We assume that there is an unknown number of abnormal processes at a time and the agent can only check with one sensor in each sampling step. To maximize the confidence level of the decision and minimize the stopping time concurrently, we propose a deep actor-critic reinforcement learning framework that can dynamically select the sensor based on the posterior probabilities. We provide simulation results for both the training phase and testing phase, and compare the proposed framework with the Chernoff test in terms of claim delay and loss.

5.1 System Model

In this chapter, we consider N independent processes, where each of the processes could be in either normal or abnormal state. We assume that at any time t, the probability of the process i, for $i = 1, 2, ..., N$, being abnormal is P_i. We denote the number of abnormal processes as k, and since all processes are assumed to be independent, the value of k could be any integer in the range $[0, N]$ at any given

time. It is also assumed that at any time instant, if anomaly occurs in any number of processes, the states of all processes will remain the same until all abnormal processes are detected and fixed.

We assume that there is a single observation target Y_t for all processes, and the samples have different density distributions depending on the states of the processes (e.g., normal or abnormal). For example, we can consider the scenario in which for each process, there is a sensor that can send a state signal to the observer in each time slot. When the process is normal, the sensor should send $Y = 0$, while if the process is abnormal, the sensor should send $Y = 1$. We note that in practical settings the sensors are not always reliable, so in this chapter we assume that the sensor will erroneously send a flipped signal with probability ρ. Now, when the process is normal, the samples are distributed according to the Bernoulli distribution $Y \sim f(Y, \rho)$, and when the process is abnormal, the distribution of the samples follows the Bernoulli distribution $Y \sim g(Y, 1 - \rho)$. Furthermore, we assume that the observer can only observe the sample from one of the N sensors at any given time. Hence, to minimize the time slots needed for detecting the anomalies, it is important to find an effective policy for sensor selection.

Since there are N processes, an unknown number of which can be in abnormal state, we have $M = 1 + \sum_{k=1}^{N} \binom{N}{k}$ hypotheses, where k is the number of abnormal processes at a given time. We say $H_0 = \{\emptyset\}$ is true when none of the N processes is abnormal. And for each of the $M - 1$ possible combinations of unknown numbers of abnormal processes, we define a hypothesis H_m for $m = 1, \ldots, M-1$. Table 5.1 shows the observation models along with the corresponding sample distribution at different sensors when the given hypothesis is true. In the table, we have three processes and we use g and f to denote the real sample density distributions in abnormal and normal states, respectively. For instance, hypothesis H_4 indicates that processes 1 and 2 are abnormal and hence the samples at sensors 1 and 2 follow the distribution g. On

Table 5.1: Observation Model

	sensor 1	sensor 2	sensor 3
$H_0 = \{\emptyset\}$	f	f	f
$H_1 = \{1\}$	g	f	f
$H_2 = \{2\}$	f	g	f
$H_3 = \{3\}$	f	f	g
$H_4 = \{1, 2\}$	g	g	f
$H_5 = \{1, 3\}$	g	f	g
$H_6 = \{2, 3\}$	f	g	g
$H_7 = \{1, 2, 3\}$	g	g	g

the other hand, samples in sensor 3 are distributed according to f since process 3 is normal. It is important to note that we assume that the parameters of the sample density distributions are unknown to the observer. To obtain an approximation of the density distribution, we employ the maximum likelihood estimation. Here, we define Ω^t as the sample space at time t, which contains all samples $\{Y_1, Y_2, \ldots, Y_t\}$. And $\mathcal{F}_{i,m}$ is a subset of Ω^t, and it contains all samples collected from sensor i when the hypothesis H_m is true. And the estimated sample density distributions can be defined as $f(Y_t|\mathcal{F}_{i,m})$ and $g(Y_t|\mathcal{F}_{i,m})$.

We denote the prior probability of each hypothesis being true as $\pi = [\pi_0, \ldots, \pi_{M-1}]$. Since the probability of the process i being abnormal is assumed to be P_i, the prior probabilities are the joint probabilities of the N processes being in the corresponding states. Then, we denote π_m^t as the posterior belief of the hypothesis H_m being true at time t, and the posterior belief is updated as

$$\pi_m^t = \frac{\pi_m \prod_{t=1}^{T} p_m^{i_t}(Y_t|\mathcal{F}_{i_t,m})}{\sum_{l=0}^{M-1} \pi_l \prod_{t=1}^{T} p_l^{i_t}(Y_t|\mathcal{F}_{i_t,l})} \tag{5.1}$$

where we denote the sensor selected at time t by i_t, and

$$p_m^{i_t}(Y_t|\mathcal{F}_{i_t,m}) = \begin{cases} g(Y_t|\mathcal{F}_{i,m}) & \text{if } i_t \in H_m \\ f(Y_t|\mathcal{F}_{i_t,m}) & \text{if } i_t \notin H_m \end{cases} . \tag{5.2}$$

5.2 Problem Formulation

Similar to [99] and [124], we consider the confidence level as the maximization objective. The confidence level on hypothesis H_m being true is given by the Bayesian log-likelihood ratio $\mathcal{C}_{H_m}$:

$$\mathcal{C}_{H_m} := \log \frac{\pi_m}{1-\pi_m}. \tag{5.3}$$

And the average Bayesian log-likelihood ratio is defined as

$$\mathcal{C} = \sum_{m=0}^{M-1} \pi_m \log \frac{\pi_m}{1-\pi_m} = \sum_{m=0}^{M-1} \pi_m \mathcal{C}_{H_m}. \tag{5.4}$$

While maximizing the long term average confidence level, we also aim at minimizing the stopping time, T_{stop}. So we assume that there are upper bound and lower bound on the posterior belief. As shown in Fig.5.1, the hypothesis H_m is claimed to be accepted when the posterior belief π_m is greater than the upper bound π_{up}, or to be rejected when the posterior belief is less than the lower bound π_{low}. And once any of the M hypotheses is accepted, the observer stops receiving samples immediately.

5.3 Deep Actor-Critic Framework

In this section, we describe the proposed deep actor-critic learning framework for the anomaly detection problem.

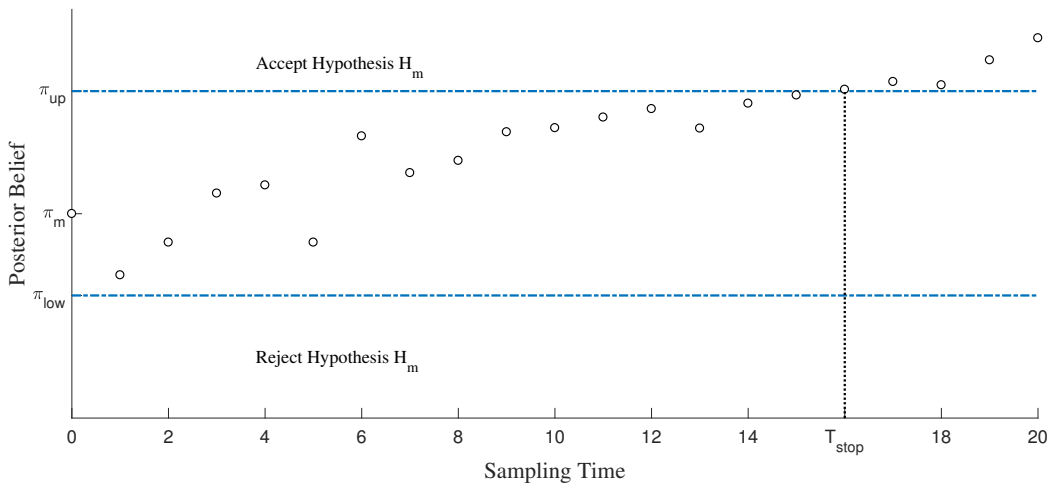


Figure 5.1: An example of stopping time.

5.3.1 Preliminaries

We first introduce the relevant definitions within the framework.

Agent's Observation and State: Since the agent can only observe one sample Y_t from the selected sensor i_t at time t, the problem can be modeled as a partially observable Markov decision process (POMDP). With this sample, the agent can update the posterior belief π^t according to (5.1). And we take the posterior belief vector as the state (or input) of the agent, and we denote the state at time t as $\mathcal{O}_t$, and define it as

$$\mathcal{O}_t = \begin{cases} \pi & \text{t} = 1 \\ \pi^{t-1} & \text{otherwise} \end{cases} . \tag{5.5}$$

Action: We denote the action space as $\mathcal{A}$, in which all valid actions are included. Here, the size of the action space is N, and a valid action stands for selecting the corresponding sensor and receiving the sample to update the posterior belief. In each iteration, the agent will score all valid actions, and choose the one with the highest score to execute.

Reward: As we introduced in the previous sections, the proposed agent has two goals: 1) maximize the average confidence level and 2) minimize the stopping time.

So we define the immediate reward r_t as

$$r_t = \frac{\mathcal{C}^t - \mathcal{C}}{t}, \tag{5.6}$$

where $\mathcal{C}^t = \sum_{m=0}^{M-1} \pi_m^t \log \frac{\pi_m^t}{1-\pi_m^t}$.

Here, we define the state $\mathcal{O}_T$ as the terminal state if any of the M hypothesis is claimed to be accepted, i.e., $\max(\pi^{T-1}) \geq \pi_{\text{up}}$. And when we update the agent, we consider a weighted reward R_t at time $t \leq T$, as a discounted sum of the rewards:

$$R_t = \sum_{\tau=t}^{T} \lambda^{\tau-t} r_\tau, \tag{5.7}$$

so that each previous selection that can lead to better future steps will achieve a greater reward. And in the implementation, the agent will be updated T times after the terminal state has been reached, using the weighted reward achieved at the terminal time T, and all the way back to the initial time $t = 0$.

5.3.2 Algorithm Overview

In this subsection, we describe the architecture of the actor-critic algorithm. The actor-critic architecture consists of two neural networks: actor and critic. In our model, these two networks will not share any neurons but are parameterized by θ.

Actor: The actor is employed to explore a policy μ that maps the agent's observation $\mathcal{O}$ to the action space $\mathcal{A}$:

$$\mu_\theta(\mathcal{O}) : \mathcal{O} \rightarrow \mathcal{A}. \tag{5.8}$$

So the mapping policy $\mu_\theta(\mathcal{O})$ is a function of the observation $\mathcal{O}$ and is parameterized

by θ. And the chosen action can be denoted as

$$a = \mu_\theta(\mathcal{O}) \tag{5.9}$$

where we have $a \in \mathcal{A}$. Since the action space is discrete, we use the softmax function at the output layer of the actor network so that we can obtain the scores of each actions. The scores sum up to 1 and can be regarded as the probabilities of obtaining a good reward when the corresponding actions are chosen.

Critic: The critic is employed to estimate the value function $V(\mathcal{O})$. At time instant t, when action a_t is chosen by the actor network, the agent will execute it in the environment and send the current observation $\mathcal{O}_t$ along with the feedback from the environment to the critic. The feedback includes the reward r_t and the next time instant observation $\mathcal{O}_{t+1}$. Then, the critic calculates the TD (Temporal Difference) error:

$$\delta^{\mu_\theta} = r_t + \gamma V(\mathcal{O}_{t+1}) - V(\mathcal{O}_t) \tag{5.10}$$

where $\gamma \in (0, 1)$ is the discount factor.

Update: The critic is updated by minimizing the least squares temporal difference (LSTD):

$$V^* = \arg\min_V (\delta^{\mu_\theta})^2 \tag{5.11}$$

where V^* denotes the optimal value function.

The actor is updated by policy gradient. Here, we use the TD error to compute the policy gradient:

$$\nabla_\theta J(\theta) = E_{\mu_\theta}[\nabla_\theta \log \mu_\theta(\mathcal{O}, a)\delta^{\mu_\theta}] \tag{5.12}$$

where $\mu_\theta(\mathcal{O}, a)$ denotes the score of action a under the current policy. Then, the weighted difference of parameters in the actor at time t can be denoted as $\Delta\theta_t = \alpha\nabla_{\theta_t} \log \mu_{\theta_t}(\mathcal{O}_t, a_t)\delta^{\mu_{\theta_t}}$, where $\alpha \in (0, 1)$ is the learning rate. And the actor network

i can be updated using the gradient descent method:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} \log \mu_{\theta_t}(\mathcal{O}_t, a_t) \delta^{\mu_{\theta_t}}. \tag{5.13}$$

5.3.3 Training Phase

In the training phase, the actor and critic neural networks are constructed and trained. For each episode, there will be a true hypothesis, generated according to the prior belief π. The agent will observe one sample at a time until it can accept a hypothesis. In the episode, at the beginning of each time slot t, the agent receives the current state $\mathcal{O}_t$, and chooses one out of the N sensors to obtain a sample Y_t. Based on the sample, the agent can update the posterior belief π^t and receive a reward. Then the critic network and actor network will be updated. Since the agent does not know which hypothesis is indeed true, the samples will be added to the corresponding subsets of overall sample space after the ground-truth is revealed, i.e., the posterior belief is always updated by the estimated density distribution based on the samples collected in the previous episodes.

The full framework is provided in Algorithm 7 below on the next page.

5.3.4 Testing Phase

In the testing phase, the agent first reloads the neural network parameters from the training phase, and makes direct use of the well-trained neural networks without further updates. To test the ability of detecting a change point, we assume that at the beginning of every episode, the hypothesis H_0 is true. And to activate the state, H_0 will be true for at least T_1 time slots so that the agent can learn a high posterior probability of H_0. Then, based on the prior belief, a new true hypothesis will be generated, and the agent continues to choose sensors. When the posterior belief of H_0 is less than the lower bound π_{low}, the agent will report a change point and reset

Algorithm 7 Deep Actor-Critic Reinforcement Learning Algorithm for Anomaly Detection: Training Phase

$t = 0$
Initialize the critic network $V_\theta(\mathcal{O})$ and the actor $\mu_\theta(\mathcal{O})$, parameterized by θ.
The agent initializes the sample space Ω^0, and the subsets $\mathcal{F}_{i,m}$, for $i = 1, \ldots, N$ and $m = 0, \ldots, M-1$.
for $T = 1$: Maximum episode **do**
  $t_{\text{start}} = t$
  Generate a new hypothesis H_j to be true according to the prior belief π, and $j \in \{0, 1, \ldots, M-1\}$.
  The agent fetches the prior belief vector π as the initial state.
  while $\mathcal{O}_T$ is not a terminal state **do**
    $t \leftarrow t + 1$
    With the state $\mathcal{O}_t$, the agent selects one out of the N sensors according to the decision policy $a_t = \mu(\mathcal{O}_t|\theta)$ w.r.t. the current policy.
    Agent receives the sample Y_t from the chosen sensor and update the posterior belief vector π^t.
    Agent updates the sample Y_t to the sample space Ω^T.
    With the new state $\mathcal{O}_{t+1}$, the agent obtains a reward r_t.
    Update the state $\mathcal{O}_t = \mathcal{O}_{t+1}$.
  end while
  $R = 0$
  for $\tau = t - 1 : t_{\text{start}}$ **do**
    $R \leftarrow r_\tau + \lambda * R$
    Critic calculates the TD error: $\delta^{\mu_\theta} = R + \gamma V(\mathcal{O}_{\tau+1}) - V(\mathcal{O}_\tau)$
    Update the critic by minimizing the loss: $\mathcal{L}(\theta) = (\delta^{\mu_\theta})^2$
    Update the actor policy by maximizing the action value: $\Delta\theta_\tau = \alpha \nabla_{\theta_\tau} \log \mu_{\theta_\tau}(\mathcal{O}_\tau, a_\tau)\delta^{\mu_{\theta_\tau}}$, $\alpha \in (0, 1)$.
  end for
  Reveal the true hypothesis, and update samples to the corresponding $\mathcal{F}_{i,j}$, and update the estimated sample density distributions.
end for
Save the trained neural networks.

Table 5.2: The settings of actor-critic network

	actor	critic
first layer	200 neurons + ReLU	200 neurons + ReLU
second layer	200 neurons + ReLU	100 neurons + ReLU
output layer	N neurons + Softmax	1 neuron
learning rate	0.0005	0.01

the state to the prior belief. Subsequently, the agent keeps collecting samples until it can claim any of the hypotheses being true.

The full framework is provided in Algorithm 8 below on the next page.

5.4 Simulation Results

5.4.1 Experiment Settings

5.4.1.1 Environment

In our experiments, we set the number of processes as $N = 3$, so that the total number of hypotheses is $M = 8$. The definition of each hypothesis and the distribution of the observations from different sensors under the specific hypothesis being true has been given in Table 5.1 in Section II. Here, we assume that the probabilities of each process being abnormal is $P = [0.2, 0.3, 0.1]$, respectively.

5.4.1.2 Actor-Critic Neural Network

The design of our proposed actor-critic framework is shown in the Table 5.2. This framework consists of two neural networks. Each neural network includes 3 layers, and the layers are connected with ReLU activation function. To ensure that the critic network is able to guide the update of the actor network, we assign larger learning rate to the critic network. And in order to maintain a stable and high performance, the learning rates decay over time so that the network parameters will not change rapidly when the neurons are well trained.

Algorithm 8 Deep Actor-Critic Reinforcement Learning Algorithm for Anomaly Detection: Testing Phase

Initialize the critic network $V_\theta(\mathcal{O})$ and the actor $\mu_\theta(\mathcal{O})$, and reload the trained parameters θ.
The agent initializes the sample space Ω^0, and the subsets $\mathcal{F}_{i,m}$, for $i = 1, \ldots, N$ and $m = 0, \ldots, M-1$.
for $T = 1$: Maximum episode **do**
  Set H_0 as the true hypothesis.
  The agent fetches the prior belief vector π as the initial state.
  for $t = 1 : T_1$ **do**
    With the state $\mathcal{O}_t$, the agent selects one out of the N sensors according to the decision policy $a_t = \mu(\mathcal{O}_t|\theta)$ w.r.t. the current policy.
    Agent receives the sample Y_t from the chosen sensor and update the posterior belief vector π^t.
    Agent updates the sample Y_t to the sample space Ω^T.
  end for
  Generate a new hypothesis H_j to be true according to the prior belief π, and $j \in \{0, 1, \ldots, M-1\}$.
  Set $D = 0$, set $\Gamma = 0$
  Set $\mathcal{O}_{T_1}$ as the new state.
  for $t^{'} = 1$: Maximum sampling time **do**
    With the state $\mathcal{O}_{t'}$, the agent selects one out of the N sensors according to the decision policy $a_{t'} = \mu(\mathcal{O}_{t'}|\theta)$ w.r.t. the current policy.
    Agent receives the sample $Y_{t'}$ from the chosen sensor and update the posterior belief vector $\pi^{t^{'}}$.
    Agent updates the sample $Y_{t'}$ to the sample space Ω^T.
    if $\pi_0^{t^{'}} \leq \pi_{\text{low}}$ **then**
      Agent rejects the hypothesis H_0, and report a change point.
      Agent resets the state as $\mathcal{O}_{t'+1}$ as the prior probability π.
    end if
    if $\max(\mathcal{O}_{t'+1}) \geq \pi_{\text{up}}$ **then**
      Agent accepts the corresponding hypothesis as the true hypothesis.
      Break Loop
    end if
  end for
  Reveal the true hypothesis, and update samples to the corresponding $\mathcal{F}_{i,j}$, and update the estimated sample density distributions.
end for

5.4.2 Training Phase

In the training phase, we set the bound π_{up} as 0.8, and run the procedure shown in Algorithm 7. To check the performance of the agent at different training steps, we conduct a validation testing after every 1000 training steps. The validation set consists of 3 hypotheses randomly selected from the M hypotheses. We denote the validation set as $\mathcal{H} = \{H_{m^1}, H_{m^2}, H_{m^3}\}$, and in the validation testing, we assign the three chosen hypotheses to be true in the order $H_{m^1} \to H_{m^2} \to H_{m^3}$. In the validation phase, each of the three hypotheses will remain to be true for 200 sampling steps, and the agent selects the sensor with its current policy, but the network will not be updated. Each time the agent is tested with the validation set, we record the posterior probabilities of the three hypotheses.

In Fig. 5.2, we plot the posterior probabilities over the sampling time. The posterior probabilities of each hypothesis in the validation set is collected from all validation phases over 15000 training episodes in total. Since each hypothesis in the validation phase remains to be true for 200 sampling steps, each validation phase has a fixed duration of 600 sampling steps. In the figure, the posterior probabilities of different hypotheses are plotted in different colors, and the darkness of the colors stand for the density of the probability at the corresponding value, i.e., the darker the color is, the more frequently that the posterior probability will take the corresponding value at the corresponding sampling time index. We can observe that at the beginning of each change point, the posterior probability of the true hypothesis increases quickly, and remains at a high value that is approximately 1. And when the next hypothesis starts to be true, the posterior probability of the previous hypothesis diminishes. Besides the patterns with increased darkness, there are also some samples of the probabilities in relatively light colors. The difference in the level of darkness indicates the exploration of the agent while trying to find an efficient selection policy. Since all dark colors appear at high values of the posterior probabilities, the agent is able to

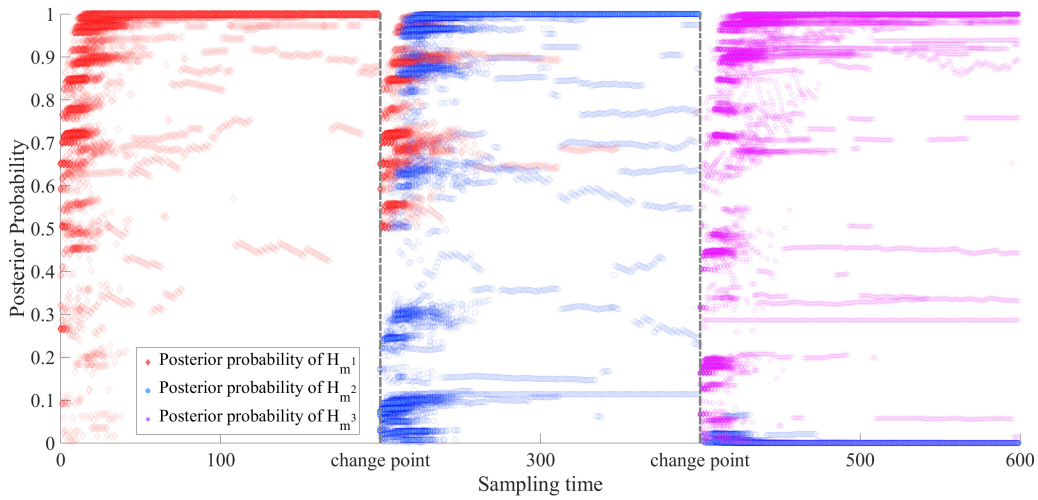


Figure 5.2: Posterior probability over the sampling time in the validation phase.

detect the true hypothesis with high reliability.

5.4.3 Testing Phase

In the testing phase, we investigate the performance of the proposed agent in terms of the detection delay and loss. Here, we define the claim delay as the difference between the time when the agent claims a hypothesis to be true (i.e., when the posterior probability of the hypothesis exceeds π_{up}) and the time when the change occurs. Also, to evaluate the accuracy of the claim, we define the loss as a ratio of the number of wrong claims to the total number of claims. To find a reasonable pair of upper and lower bound for the decision making, in the experiments, we vary the upper bound π_{up} as $\pi_{\text{up}} \in [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]$, and at the same time vary the lower bound π_{low} as $\pi_{\text{low}} \in [0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]$.

In Fig. 5.3 and Fig. 5.4, we plot the average claim delay and average loss, respectively, under each pair of the upper and lower bounds. From the figures we notice that as the upper bound π_{up} increases, the claim delay increases and the loss decreases. This is because when the upper bound is high, the agent accepts a hypothesis more cautiously and hence more observations will be taken, which also

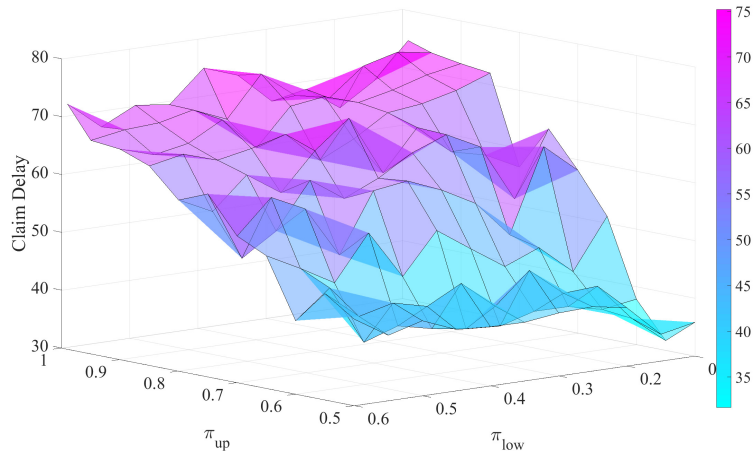


Figure 5.3: Claim delay under different $< \pi_{\text{up}}, \pi_{\text{low}} >$ pairs, when $\pi_{\text{up}} \in [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]$, and $\pi_{\text{low}} \in [0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]$

improves the confidence level of the decision. On the other hand, as the lower bound π_{low} decreases, the loss also decreases slightly, because the lower bound is the threshold to reject the previous hypothesis that the agent considers to be true (which should always be H_0 in the testing phase). When the lower bound is reduced, more stringent conditions are imposed to reject a hypothesis, which results in reduced false alarms. And comparing with the patterns shown in Fig. 5.2, more sampling time is needed in the testing phase. That is because in the testing phase, the detection starts with the posterior probability of H_0 being very high, and hence the agent will need more samples to confirm that the previous hypothesis has turned to be false. This ability to adapt to different initializations makes the agent more practically appealing in dealing with the real anomaly detection cases where all processes are normal at the beginning.

Finally, we compare our proposed framework with the Chernoff test [92]. Chernoff test considers the Kullback-Leibler information of the two distributions of the observations, and decides whether to receive the sample from the sensor with highest accumulated log-likelihood ratio or randomly pick one of the sensors. In our experiments, we assign the lower bound π_{low} to be 0.6, and vary the upper bound

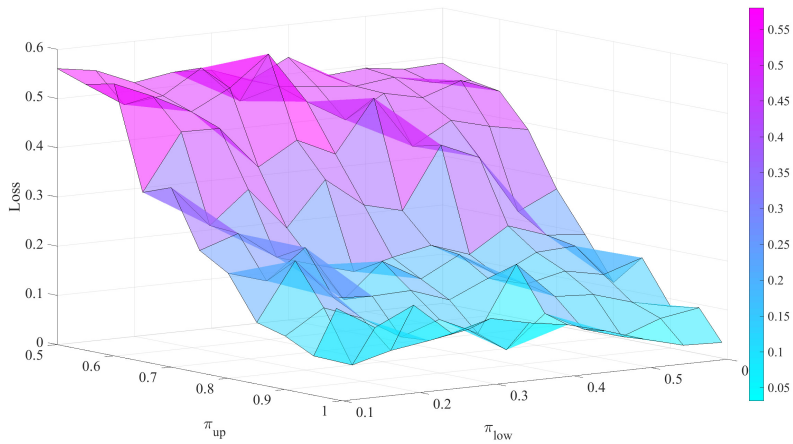


Figure 5.4: Loss under different $< \pi_{\text{up}}, \pi_{\text{low}} >$ pairs, when $\pi_{\text{up}} \in [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]$, and $\pi_{\text{low}} \in [0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]$

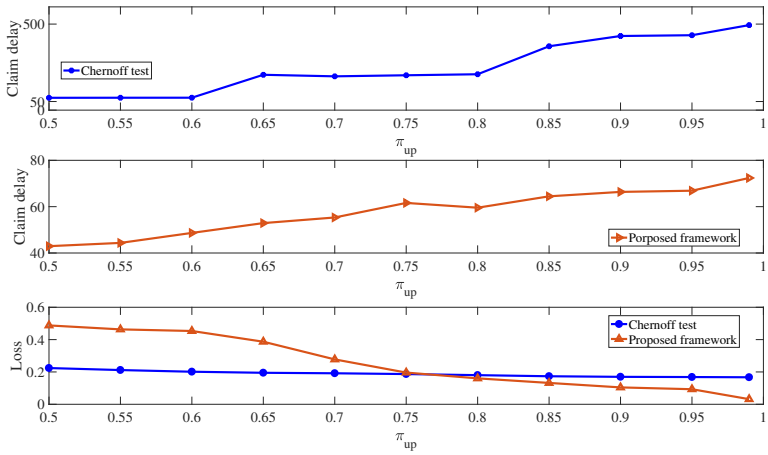


Figure 5.5: Comparison between proposed framework and Chernoff test: claim delay and loss with $\pi_{\text{low}} = 0.6$, and π_{up} varies as $[0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]$.

as $\pi_{\text{up}} \in [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]$. Shown in Fig. 5.5 are the claim delay and decision loss curves achieved by our proposed framework and Chernoff test. For the claim delay, it is obvious that the Chernoff test will need many more samples to reach the stopping criterion. This is because the Chernoff test assumes that all hypotheses are distinguishable under different tests, which means that it requires all hypotheses to have different observation distributions under each test. However, in our system model, just as shown in Table 5.1, different hypotheses can have the same observation distribution. For example, under H_1 being true, if the agent tests with the sample from sensor 1, it will not be able to distinguish hypotheses H_1, H_4, H_5, and H_7, because under all these hypotheses, the process 1 is in abnormal state. And for the loss, it is obvious that the loss from the proposed agent decreases when the upper bound increases. However, the loss from the Chernoff test, though slightly decreases as the upper bound gets larger, is relatively stable. When $\pi_{\text{up}} \geq 0.75$, the performance of the proposed agent is more competitive in terms of both the claim delay and loss. So the proposed agent is more suitable for systems with high sampling cost and require high confidence levels.

Chapter 6

Anomaly Detection and Sampling Cost Control via Hierarchical GANs

In this chapter, we study anomaly detection by considering the detection of threshold crossings in a stochastic time series without the knowledge of its statistics. To reduce the sampling cost in this detection process, we propose the use of hierarchical generative adversarial networks (GANs) to perform nonuniform sampling. In order to improve the detection accuracy and reduce the delay in detection, we introduce a buffer zone in the operation of the proposed GAN-based detector. In the experiments, we analyze the performance of the proposed hierarchical GAN detector considering the metrics of detection delay, miss rates, average cost of error, and sampling ratio. We identify the tradeoffs in the performance as the buffer zone sizes and the number of GAN levels in the hierarchy vary. We also compare the performance with that of a sampling policy that approximately minimizes the sum of average costs of sampling and error given the parameters of the stochastic process. We demonstrate that the proposed GAN-based detector can have significant performance improvements in

terms of detection delay and average cost of error with a larger buffer zone but at the cost of increased sampling rates.

6.1 System Model

As noted above, we consider anomaly detection as the detection of crossing a threshold Γ in a stochastic time series. Specifically, we assume that an anomaly occurs when the monitored stochastic process exceeds or falls below Γ. Such anomaly detection is required, for instance, in remote monitoring using sensors in smart home, smart city, e-Health, and industrial control applications. In these cases, the monitored process can be modeled as a stochastic process and it is very important to accurately and timely detect if the process (describing e.g., the patient's health in remote health monitoring or the operational characteristics of the power grid in a smart grid application) crosses a threshold. To react to the changes immediately, the system can continuously monitor the environment. However, this will lead to very high sampling, sensing and also communication costs (e.g., if the sensing results need to be sent to a remote processing center). Alternatively, the system can observe and sample the process intermittently. In this case, the sampling can be nonuniform with the sampling rate depending on how close the values are to the threshold Γ. With this approach, the sampling/sensing cost will be reduced but there will be a higher risk of delay in the threshold-crossing detection. Therefore, there exists a tradeoff between sampling and delay costs and this should be addressed by taking the delay cost into account when making the sampling decisions.

While our framework is applicable to any process or time series, we consider the Ornstein-Uhlenbeck (OU) process (which has applications in physical sciences, power system dynamics, financial mathematics) in this chapter in order to be more concrete in our discussions. Additionally, a sampling policy for the OU process is previously

derived in [125] under the assumption of complete statistical knowledge, and we will compare the performance of the proposed hierarchical GAN framework with that of this policy. OU process can be expressed as

$$dx(t) = \theta(\mu - x(t))dt + \sigma dW(t) \tag{6.1}$$

where μ is the mean of the time series, θ is the speed of mean reversion that scales the distance between $x(t)$ and the mean μ, and σ is the volatility to scale the Wiener process $W(t)$. Specifically, we set the mean μ as 0.

The cost of error due to delayed detection can be defined as the area enclosed between the actual crossing point $x(T_{\text{true}})$ and the detected crossing point $x(T_{\text{detect}})$, where T_{true} is the actual time instant at which the threshold is crossed, and T_{detect} is the time instant when threshold-crossing is detected. Therefore, when the threshold is $\Gamma = 0$, the cost of error can be computed as

$$C = \int_{T_{\text{true}}}^{T_{\text{detect}}} |x(t)|dt. \tag{6.2}$$

Note that the cost of error due to delayed detection is proportional to the value of the process x and the gap between T_{detect} and T_{true}.

In [125], based on the assumption that the parameters of the OU process are known, a policy is derived to control the sampling time. This policy makes use of the OU process parameters and the current sample to estimate the subsequent sampling time that minimizes the sum of the average costs of error and sampling. Specifically, under certain conditions and assuming $\Gamma = 0$, an approximate solution for the next sampling time is given as

$$T_1^*(x(t)) = T^* + \frac{1 - e^{-\theta T^*}}{\sqrt{1 - e^{-2\theta T^*}}} \frac{\sqrt{\pi}}{\sigma\sqrt{\theta}} x(t) \tag{6.3}$$

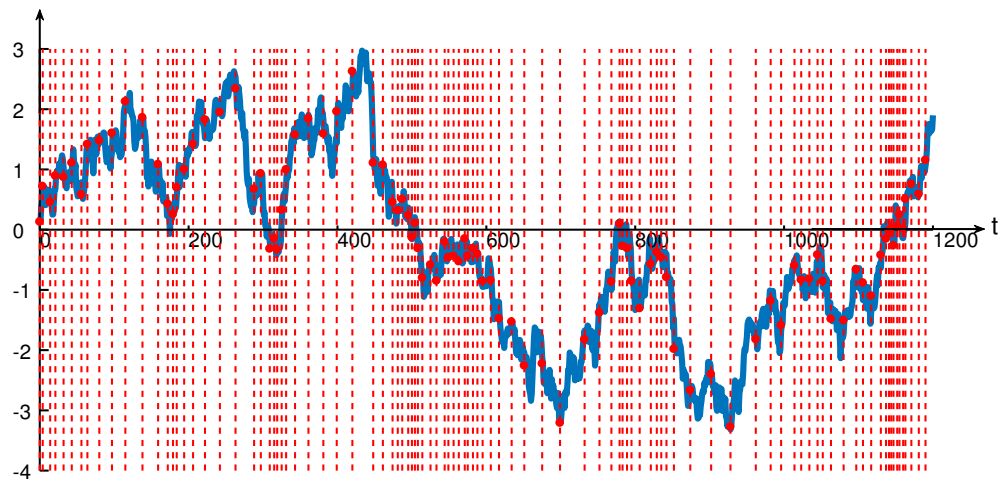


Figure 6.1: Sampling of an OU process with parameters $\mu = 0$, $\sigma = 0.5$, $\theta = 0.025$, and sampling cost $c_s = 0.1$.

where $T^* = (\frac{18\pi c_s^2}{\sigma^2})^{\frac{1}{3}}$, and c_s is a predefined sampling cost.

In Fig. 6.1, we demonstrate such sampling process using (6.3). The blue curve is the time series $x(t)$, red vertical lines indicate the sampling time instants, and the red dots are the samples collected by the policy. It is obvious that when the value of $x(t)$ approaches the threshold $\Gamma = 0$, the policy samples more frequently, and when $x(t)$ moves far away from the threshold, the policy samples less frequently. Such nonuniform sampling policy provides an effective solution, balancing the detection accuracy and the sampling cost, However, the parameters of the OU process may not be known in practice, and this renders the optimal policy inapplicable. In such cases, data-driven approaches are needed. Considering these scenarios, we in this chapter propose a GAN-based framework, that does not require any information on the OU process, to control the sampling time. Indeed, the proposed approach is quite general and applicable to any process. Basically, in this framework, the current sample $x(t)$ will be fed to the GAN, and the samples in the following N time instants will be predicted. And based on the predictions, the next sampling time will be estimated.

We denote the set of predictions obtained at time t as $\{\hat{x}(t+1), \hat{x}(t+2), \ldots, \hat{x}(t+N)\}$, where N denotes prediction length, and $\hat{x}(t+\Delta t)$ is the prediction of $x(t+\Delta t)$

for $\Delta t = 1, 2, \ldots, N$. Then, each element in the prediction set will be compared with the threshold Γ to see if it is a threshold-crossing point, and to decide the next sampling time. We denote the next sampling time as $T(x(t))$, which can be expressed as

$$T(x(t)) = \begin{cases} t + \Delta t, \text{if } \hat{x}(t + \Delta t) \text{ is a crossing point} \\ \qquad\qquad \text{for some } \Delta t \in [1, 2, \ldots, N] \\ t + N + 1, \text{if no crossing point is predicted} \end{cases} . \tag{6.4}$$

Note that the accuracy of such prediction is critical in deciding when to take the new sample. And the mean squared error in the prediction can be expressed as

$$\frac{1}{N} \sum_{\Delta t=1}^{N} (\hat{x}(t + \Delta t) - x(t + \Delta t))^2. \tag{6.5}$$

6.2 Hierarchical GAN Framework

6.2.1 Preliminaries

We first describe the general structure of a GAN [126]. In particular, GAN consists of two neural networks: a generator G, that is used to capture the statistical features of the data; and a discriminator D that is used to estimate the probability that a sample comes from the training data rather than the generative model G to evaluate the generative policy.

We first define a sample space $\mathcal{S}$ with a probability distribution $p(s|x)$, where s is a set of samples corresponding to the real data x in the training data set. The generator G maps the sample into the real data space:

$$G(s; \psi) : s \to \hat{x}, \tag{6.6}$$

where ψ denotes the parameters of the generator neural network, and the $\hat{x}$ is a projection (of real data x) generated by the generator G.

The discriminator $D(\tilde{x};\omega)$ estimates the probability of the input $\tilde{x}$ coming from the real data set, where ω denotes the parameters of the discriminator neural network, and the input $\tilde{x}$ can be either the real data x or the generated data $\hat{x}$. A good discriminator D is expected to be able to distinguish the generated data from the real data, i.e., the estimated probability should be very small if the input is the generated data and should be close to 1 if the input is from the real data. Therefore, the discriminator is aimed at minimizing the objective function given as

$$\mathcal{L}_D = -(\log(D(x)) + \log(1 - D(\hat{x}))). \tag{6.7}$$

On the other hand, for a generator G that has the goal to learn the real data distribution, the evaluation $D(\hat{x})$ acts as a guidance on the update of the generative model. Thus, a good generator G should be able to make the generated data indistinguishable from the real data to the discriminator D. For this purpose, the generator G seeks to maximize $D(\hat{x})$, or equivalently minimize the following objective function:

$$\mathcal{L}_G = \log(1 - D(\hat{x})). \tag{6.8}$$

The workflow of GAN is presented in Algorithm 9 below.

6.2.2 Hierarchical Structure and Anomaly Detection

With the GAN introduced above, the sample $x(t)$ collected at time t can be used to generate the predictions of data in the next N time instants. Therefore, the choice of the value of N determines the maximum gap between the two successive sampling time instants. To control the sampling cost in the anomaly detection, we assume that the system only takes one sample in each sampling time. Note that we can increase

Algorithm 9 Workflow of GAN

Initialize the generator $G(s;\psi)$ with the parameters ψ, and the the discriminator $D(x;\omega)$, parameterized by ω.
for $T = 1$: Maximum episode **do**
Fetch the sample set $s(t)$ and the corresponding real data set $x(t)$.
Use the generator G to generate the projection of the real data: $\hat{x}(t) = G(s(t);\psi(t))$
Feed the projection $\hat{x}(t)$ and the real data $x(t)$ to the discriminator D, and obtain the estimated probabilities of both data being real data.
Update the discriminator by descending the stochastic gradient:
$-\nabla_\omega(\log(D(x(t);\omega(t)) + \log(1 - D(\hat{x}(t);\omega(t)))$
Update the generator by descending the stochastic gradient:
$\nabla_\psi \log(1 - D(\hat{x}(t);\omega(t))$
end for

the prediction length N to enable the detector to potentially sample less frequently. However, since a single sample contains very limited information for the GAN to make predictions, a single GAN will not be able to maintain a high prediction accuracy with increased prediction length. To achieve a better balance in this trade-off, we propose a hierarchical GAN structure as shown in Fig. 6.2.

The hierarchical GAN consists of N GANs, where GAN i takes the sample collected at time t and the predictions from the lower level GANs 1 through $i-1$ as the input and generates the next prediction $\hat{x}(t+i)$. In this way, the hierarchical GAN takes advantage of the accurate predictions generated by the lower level GANs to reconstruct the pattern of the data and improve the reliability of the predictions. Meanwhile, to minimize the loss presented in (6.5), we add a squared error term to the loss function given in (6.8), so the loss of the generator in GAN i is given as

$$\mathcal{L}_{G_i} = \log(1 - D(\hat{x}(t+i))) + (\hat{x}(t+i) - x(t+i))^2. \tag{6.9}$$

While accurate predictions of the lower level GANs can help to reduce the prediction losses of the upper level GANs, if the lower level GANs are not well trained, the errors will propagate to the upper levels which can eventually lead to a large accumu-

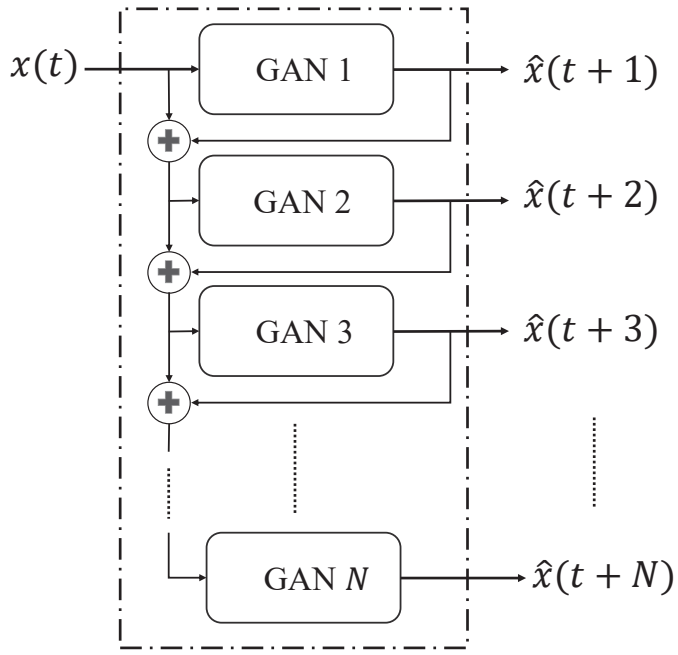


Figure 6.2: Structure of hierarchical GAN.

lated error at the last level. To address this, we train the hierarchical GAN level by level, and freeze the update of well-trained levels to avoid overfitting. The training procedure of hierarchical GAN is shown in Algorithm 10. In the training phase, the OU processes are generated with random parameters σ and θ at the beginning of every episode and are assumed to be available to the system.

After the GANs are trained, the parameters are saved for the testing phase. In the testing phase, the OU processes are also generated with random parameters in every episode, but the parameters as well as the real data are no longer available to the hierarchical GAN. Unlike in the training phase, only the GANs that can make use of the available real samples can update the neural networks. For example, with real data $x(t)$, the system can determine the next sampling time $T(x(t))$. If we denote the next sample as $x(T)$, then the GAN which predicts the corresponding $\hat{x}(T)$ can use this projection and real data pair to update its model.

Algorithm 10 Training procedure of hierarchical GAN

```
for i = 1 : N do
    Initialize the generator G_i(s; ψ_i) with the parameters ψ_i, and the the discriminator D_i(x; ω_i), parameterized by ω_i.
    for τ = 1:Maximum episode do
        Randomly generate an OU process time series with length as L.
        Take x(0) as the first sample and set t = 0.
        while t ≤ (L − i − 1) do
            for j = 1 : i do
                Fetch the input
                    s_j = [x(t), x̂(t + 1), ..., x̂(t + j − 1)].
                Obtain x̂(t + j) = G_j(s_j; ψ_j).
                if j == i then
                    Update GAN j.
                end if
            end for
            Determine the next sampling time T(x(t)) using Eq. (6.4).
            Set t = T(x(t)).
        end while
    end for
end for
```

6.3 Simulation Results

6.3.1 Experiment Settings

6.3.1.1 Environment

In the experiments, we let the mean value μ be fixed at 0, set the range of θ as $[0.02, 0.03]$, and the range of σ as $[0.4, 0.6]$. At the beginning of each episode, the OU process will be generated with θ and σ randomly selected from their corresponding range according to a uniform distribution. We also set the threshold as $\Gamma = 0$ throughout the experiments.

6.3.1.2 Structure of GAN

Each GAN consists of two neural networks: generator network and discriminator network. The input size of the generator network depends on the index of its level in

the hierarchical structure. And in the generator network, there is one long short-term memory (LSTM) layer followed by two fully-connected layers. And the discriminator network consists of three fully-connected layers, and the ReLU activation function is employed in between layers and Sigmoid activation function is adopted after the output layer.

6.3.2 Training Phase

In the experiments, we first train each GAN in the hierarchy for 5000 times, and when all GANs are converged, we freeze the update of GANs but continue feeding the OU process data to the detector. Then we record the squared error $(\hat{x}(t+i) - x(t+i))^2$ from each GAN as the loss. In Fig. 6.3, we plot the loss as a function of the GAN index, and compare the impact of increasing the prediction length N. It is obvious that as the index of the GAN increases, the loss tends to get accumulated. We also notice that when the index is between 2 to 6, the losses first increase and then drop to a lower level, and following this, the losses continue increasing at a fixed rate.

As we mentioned before, a single sample can only provide limited information for the GANs to predict the future samples, and consequently the loss jumps to relatively high levels initially. However, as more predictions are used as input to the upper level GANs, the loss is corrected to some degree by the LSTM layers. This is because of the fact that even though the previous prediction is not perfect, the presence of such prediction can act as a projection of the real data set to provide the upper level GANs with more features of the time series data.

6.3.3 Testing Phase

Considering the losses shown in Fig. 6.3, it is expected that the proposed GAN-based detector will experience errors in the testing phase. To reduce such errors, we introduce a small buffer zone of width ρ around the threshold Γ in the following

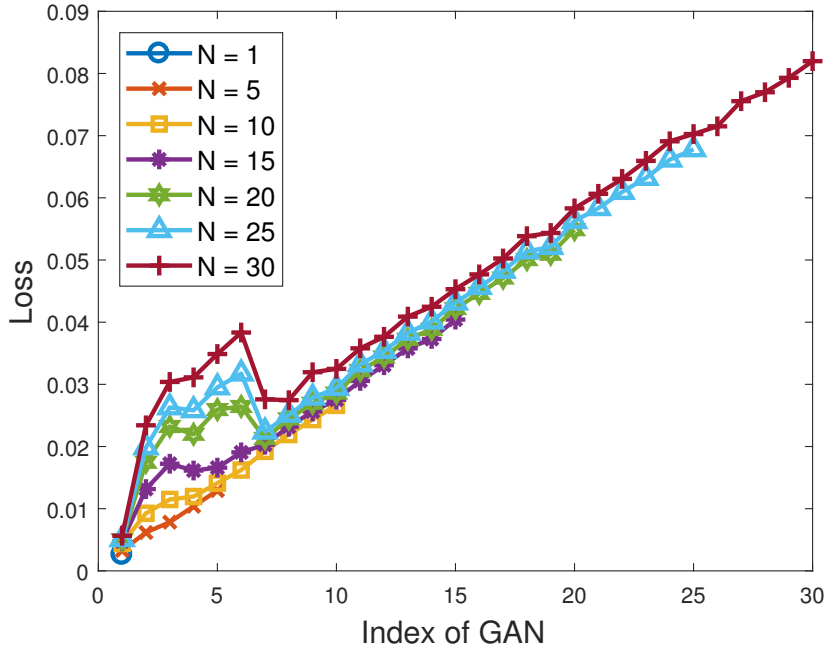


Figure 6.3: The prediction loss in each level of the hierarchical GAN.

experiments, and define the threshold-crossing time as the first time instant at which the predicted sample is within the range $[\Gamma-\rho, \Gamma+\rho]$. With this, the delay in detecting a threshold crossing is reduced at the cost of increased number of samples.

Random fluctuations of the stochastic process imply that the process can potentially cross over the threshold multiple times within a certain time frame, and duration from one threshold crossover to another varies randomly as well. Therefore, there are two potential outcomes of detection: 1) the GAN-based detector successfully detects the threshold crossing potentially with a delay but before another crossover occurs; and 2) the detector fails to detect the threshold crossing before another crossing occurs (and we indicate this outcome as *failed/missed detection*).

We define the detection delay as the difference between the time instant when the change is detected and the time instant the change actually occurs. Thus, the delay varies between 0 (indicating perfect detection) and the time until a new crossing occurs (indicating that detection was not done before a new crossing). In Fig. 6.4,

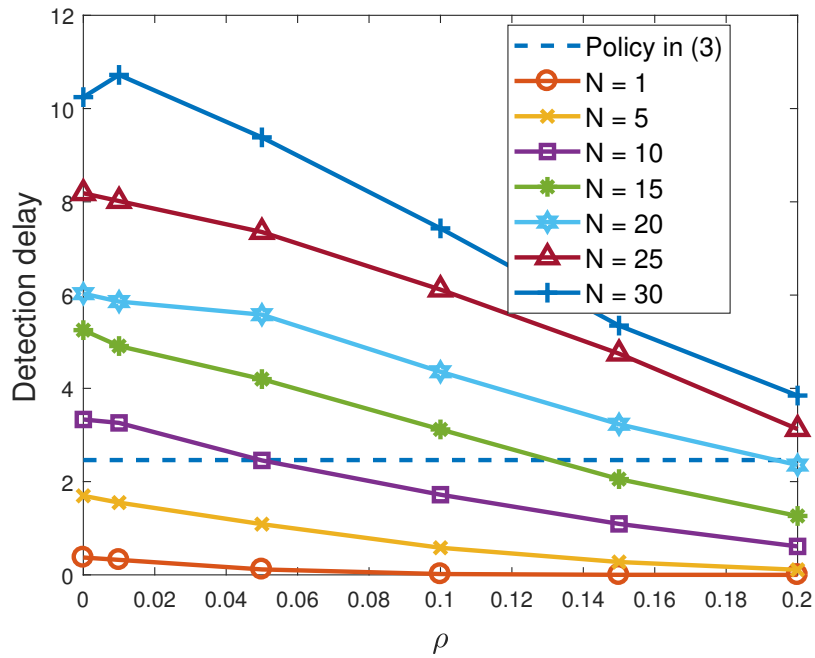


Figure 6.4: The average detection delay vs. buffer zone width ρ.

we plot the average detection delay as a function of the buffer zone width ρ for different values of N. As we increase ρ, the detection delay achieved by the proposed hierarchical GAN-based detector decreases in all cases, and a delay smaller than that achieved by the sampling policy in (6.3) can be attained when the prediction length (or equivalently the number of GANs in the hierarchy) is $N = 1, 5, 10, 15$ or 20 for sufficiently large ρ. Note that the sampling policy in (6.3) assumes complete statistical knowledge (which is not available to the GAN-based detector) but does not perform any explicit predictions. In Fig. 6.4, we further observe that for fixed ρ, delays expectedly grow as we increase the prediction length N and take fewer samples.

Since the detection delay is only considered when the threshold crossings are successfully detected, in Fig. 6.5 we plot the miss rate to have a better understanding on the failed/missed detection rates. The miss rate is defined as the ratio of the number of crossings that are missed by the detector to the total number of crossings. We observe that the miss rates achieved with different values of prediction length N

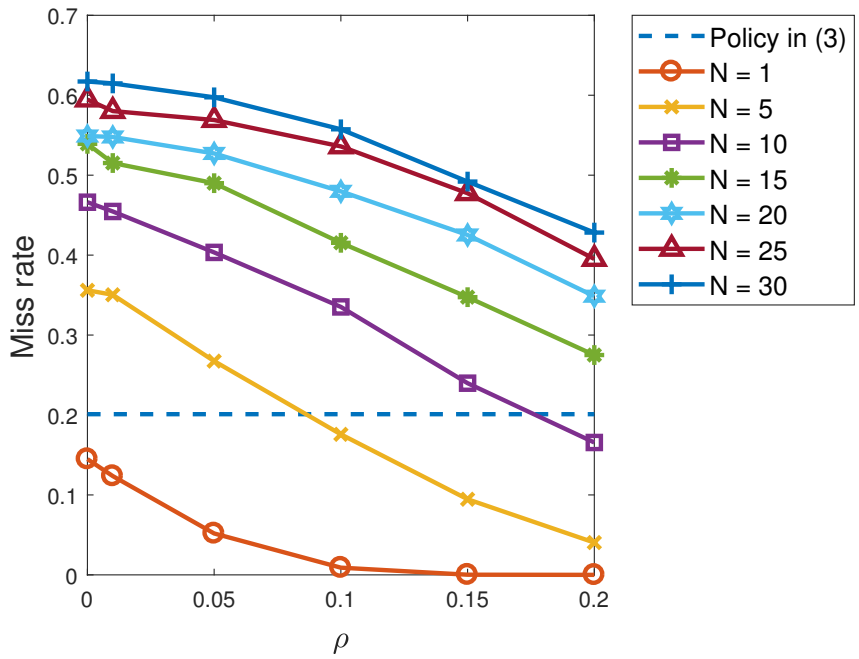


Figure 6.5: The average miss detection ratio vs. buffer zone width ρ.

decrease as the buffer size ρ increases, and the miss rates for $N = 1, 5, 10$ can be lower than that achieved by the sampling policy in (6.3). To further reduce the miss rate, we can continue increasing the buffer size, but this will lead to high sampling rates. We also notice in Fig. 6.5 that miss rate increases as N is increased. As noted above, with larger prediction length, the hierarchical GAN can sample less frequently. However, this increases the risks of miss detection because the duration until a new crossing can be far smaller than the prediction length, and when the short duration is coupled with the low sampling ratio, the changes are missed with an increased probability.

We can also measure the performance of the proposed GAN-based detector by considering the cost of error (due to delayed detection) as formulated in (6.2). Note that even if the miss rates are high with large prediction length, misses might occur due to short durations between consecutive threshold crossings, whose cost with respect to the metric in (6.2) is small. To address this possibility and understand the

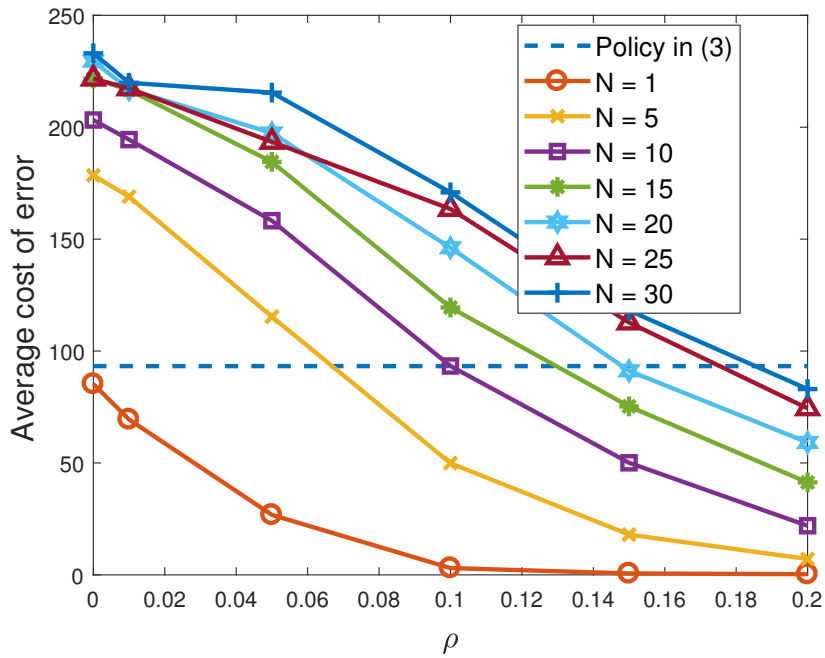


Figure 6.6: The average cost of error (due to delayed detection) vs. buffer zone width ρ.

impact of miss rates, we investigate the cost of error due to delayed detection. In Fig. 6.6, average costs are plotted as a function of ρ for different values of N. Here, cost of error in (6.2) is averaged over 10000 time series. As seen in Fig. 6.6, the GAN-based detectors' performance in terms of costs approaches and exceeds the performance of the policy in (6.3) (i.e., starts achieving lower cost) as the buffer zone width increases. With buffer width set as $\rho = 0.1$, three out of the seven tested GAN-based detectors can perform competitively or better in comparison with the sampling policy in (6.3). The number increases to five when $\rho = 0.15$, and all seven GAN-based detector can work less costly with $\rho = 0.2$. On the other hand, in Fig. 6.5, less than half of the tested GAN-based detectors are able to outperform the policy in (6.3). This confirms that the number of missed detections are primarily due to short durations between consecutive threshold crossings.

In the numerical analysis above, we have primarily addressed the performances in terms of detection delays. Note that prediction lengths also affect the sampling rates,

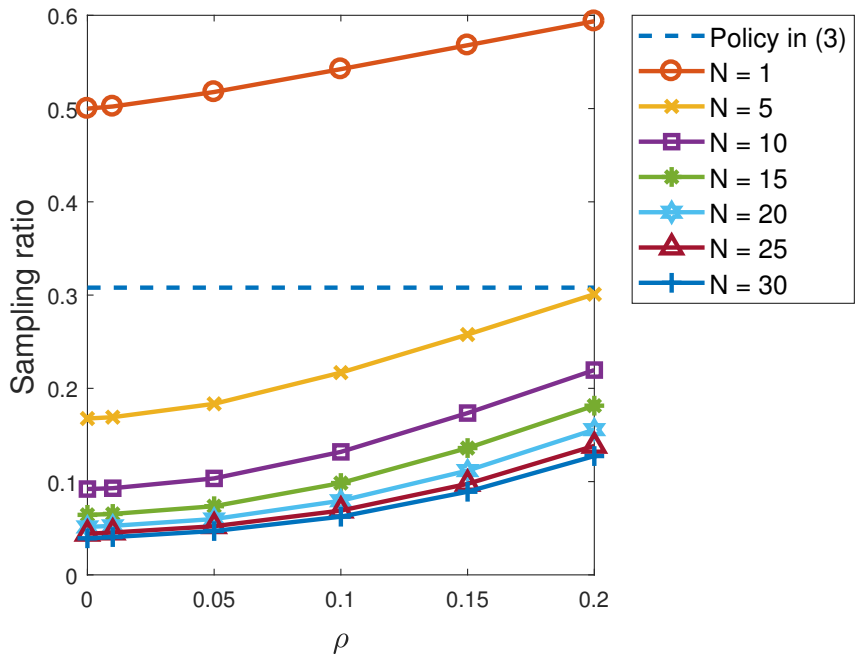


Figure 6.7: The average sampling ratio vs. buffer zone width ρ.

which we investigate next numerically. In particular, we define the sampling ratio as the number of samples taken by the detector over the total number of samples in the time series. In Fig. 6.7, we plot the sampling ratio required by the GAN-based detectors as a function of ρ. We observe that the sampling ratio grows with increasing ρ. We also see that except for the GAN-based detector with $N = 1$, all other GAN-based detectors sample the data less frequently than policy in (6.3) for all values of ρ. Even after the sampling ratios grow with ρ increasing to 0.2, most of the GAN-based detectors still exhibit obvious advantages.

We have seen above that the GAN-based detectors' performance in terms of considered metrics are strongly influenced by the selection of ρ, and this makes ρ a critical parameter enabling us to control the tradeoff between the delay costs and sampling rates. Specifically, we have observed in Figs. 6.4, 6.5 and 6.6 that the detection delays, miss rates, and average cost of error can all be reduced by increasing ρ but at the expense of requiring more samples as seen in Fig. 6.7. We have also noted

that even though the GAN-based detectors do not have statistical knowledge of the OU processes, they can outperform the sampling policy in (6.3) in terms of detection delays and sampling ratio, owing to their well-trained neural networks and prediction capabilities.

Chapter 7

Robust Learning-Based Detection with Cost Control and Byzantine Mitigation

In this chapter, we consider two types of noise: the noise introduced by the sensors during sensing, and the noise in the communication links. The sensing noise affects the quality of sensing, and lower sensing noise generally indicates a more expensive sensor with a higher sensing cost. Noise in communication can be due to distortion in reception, interference and/or adversarial jamming attacks. In this work, we seek a framework to learn the states and detect anomalies fast and accurately with cost control potentially in the presence of adversarial attacks.

7.1 System Model

We consider a scenario in which there are N remote sensors monitoring a target process. The state of the process can switch between M possible states. Here, we assume that the state of the process can be denoted as a signal $S \in \{s_1, \ldots, s_m, \ldots, s_M\}$, and each element s_m $(m = 1, 2, \ldots, M)$ stands for a possible state. We consider that a

decision-making agent dynamically selects the sensors to probe the process state and makes decisions on the process state based on the samples collected by all the selected sensors. When the selected sensors probe the process state, the corresponding state signal $S = s_m$ will be observed by every probing sensor albeit with noise. The sensors estimate the process state and report to the decision-maker individually.

Ideally, the process state S can be detected using only one sample. In practice, however, the sensor observations and/or the communication links are noisy. Hence, the decision-maker needs to observe multiple samples to ensure a detection accuracy. A diagram of the sensing and transmission by a single sensor in the presence of noise is depicted in Fig. 7.1. Due to their types and differences, sensors experience different levels of noise in their observations. The noise introduced in the observation of sensor i is distributed according to a Gaussian noise with zero mean and variance σ_i^2, i.e.,

$$n_i \sim \mathcal{N}(0, \sigma_i^2). \tag{7.1}$$

Therefore, the received sensing/observation signal at sensor i can be expressed as $s + n_i$. We also assume that, at each time when sensor i requests a state signal, there is a probing cost c_i. And for the sensor whose noise power σ_i^2 is lower, the corresponding cost c_i is typically higher. After receiving the observation, the sensors quantize the signal according to a set of predefined thresholds $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_{M+1}\}$. We assume

$$-\infty = \Gamma_1 < s_1 < \Gamma_2 < \cdots < s_m < \Gamma_{m+1} < \cdots < s_M < \Gamma_{M+1} = \infty, \tag{7.2}$$

and the sensor i quantizes the signal as

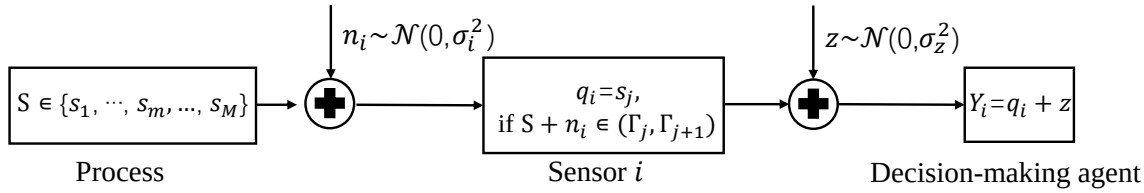


Figure 7.1: A diagram that depicts the noisy observations of a single sensor and the noisy observations of the decision-maker.

$$q_i = s_j, \text{if } S + n_i \in (\Gamma_j, \Gamma_{j+1}), \exists j \in \{1, 2, \ldots, M+1\}. \tag{7.3}$$

Then the sensor i transmits the quantized signal q_i to the decision-maker over a communication channel. Reception over the channel is distorted by another additive Gaussian noise z with mean zero and variance σ_z^2, i.e.,

$$z \sim \mathcal{N}(0, \sigma_z^2), \tag{7.4}$$

Therefore the signal received at the decision-maker from sensor i is denoted as

$$Y = q_i + z. \tag{7.5}$$

In the considered setting with noisy observations, we propose a soft actor-critic based decision-making agent that aims at dynamically selecting sensors in order to make a decision quickly with a certain confidence level at a small sensing cost. Here, while we have assumed without loss of generality that sensed signals directly match the values of the process states, the analysis is general and applicable to any other fixed signalling values that represent different states. Also, the system model can also be extended to cases with multiple processes and multiple states for each process.

7.2 Problem Formulation

7.2.1 Stopping Rule

Since the process state M possible states, we have M hypotheses. We denote the prior probabilities of each hypothesis being true by the probability vector $\pi = [\pi_1, \pi_2, \ldots, \pi_M]$. With this, we further denote by π_m^t the posterior belief of the hypothesis H_m, $m =$ 0 or 1, being true at time t, and update the posterior belief as

$$\pi_m^t = \Pr\{S = s_m \mid Y_1, \ldots, Y_t\} = \frac{\pi_m \prod_{\tau=1}^{t} p(Y_\tau | S = s_m)}{\sum_{l=1}^{M} \pi_l \prod_{\tau=1}^{t} p(Y_\tau | S = s_l)} \tag{7.6}$$

where $p(Y_t|S = s_m)$ is the distribution of Y_t observed at the decision-maker at time t given that the process state is $S = s_m$, and this distribution is derived below. We note that when the agent selects to observe n processes, $n = 0, 1, ..., N$, in a time slot, the posterior probabilities will be updated n times.

The probability distribution $p(Y|S = s_m)$ can be expressed as

$$p(Y|S = s_m) = \sum_{l=1}^{M} p(Y, q = s_l | S = s_m) \tag{7.7}$$

$$= \sum_{l=1}^{M} P(q = s_l | S = s_m) p(Y | q = s_l, S = s_m). \tag{7.8}$$

Since the variables Y and S are conditional independent, (7.8) can be rewritten as

$$p(Y|S = s_m) = \sum_{l=1}^{M} P(q = s_l | S = s_m) p(Y | q = s_m) \tag{7.9}$$

The sensors apply the detection rule in (7.3), and therefore the conditional prob-

abilities $P(q|S)$ can be expressed in terms of the Gaussian Q function. For instance,

$$p(q = s_j|S = s_m) = p(\Gamma_j - s_m < n_i < \Gamma_{j+1} - s_m) = Q\left(\frac{\Gamma_j - s_m}{\sigma_i}\right) - Q\left(\frac{\Gamma_{j+1} - s_m}{\sigma_i}\right) \tag{7.10}$$

The conditional distribution $p(Y|q = s_j)$, is Gaussian with mean s_j and variance σ_z^2, i.e., we have

$$p(Y|q = s_j) = \mathcal{N}(s_j, \sigma_z^2). \tag{7.11}$$

Then, substituting (7.10) and (7.11) into (7.8), we can obtain the conditional probability density function of Y given the source signal S, and utilize it to update the posterior probability in (7.6).

As shown in Fig. 7.2, the hypothesis H_m is claimed to be accepted when the posterior belief π_m is greater than the upper bound π_{upper}, or to be rejected when the posterior belief is less than the lower bound π_{lower}. And once any of the M hypotheses is accepted, the observer stops receiving samples immediately.

7.2.2 Confidence Measures and Rewards

In this chapter, we consider two different confidence measures, and we derive two rewards based on them to be used in the learning algorithms.

7.2.2.1 Log-likelihood Ratio Based Reward

Similarly as in [99] and [124], we consider the confidence level as the maximization objective. One confidence level in terms of the posterior probability at time t is given

by the average Bayesian log-likelihood ratio (LLR)

$$\mathcal{C}(\pi(t)) = \sum_{m=0}^{M-1} \pi_m \log \frac{\pi_m}{1-\pi_m}. \tag{7.12}$$

And the LLR-based reward, which measures the improvement in the confidence level from time $t-1$ to time t, is defined as

$$r_{\mathcal{C}}(t) = \mathcal{C}(\pi(t)) - \mathcal{C}(\pi(t-1)). \tag{7.13}$$

7.2.2.2 Entropy Based Reward

Confidence can also be measured via the entropy. Since the entropy of the posterior probability distribution is minimized by having one of the posterior probabilities to be 1 and all the other probabilities to be 0, we can also consider an entropy-based reward given as

$$r_{\mathcal{H}}(t) = \mathcal{H}(\pi(t-1)) - \mathcal{H}(\pi(t)) \tag{7.14}$$

where entropy is formulated as

$$\mathcal{H}(\pi(t)) = -\sum_{m=0}^{M-1} \pi_m \log \pi_m. \tag{7.15}$$

7.2.3 Cost

As mentioned in the previous section, we consider a sensing cost and incorporate it into the reward function (as described in (7.18) below). This instantaneous cost $c(t)$ depends on the cost of the sensors that are selected in time slot t. We assume a

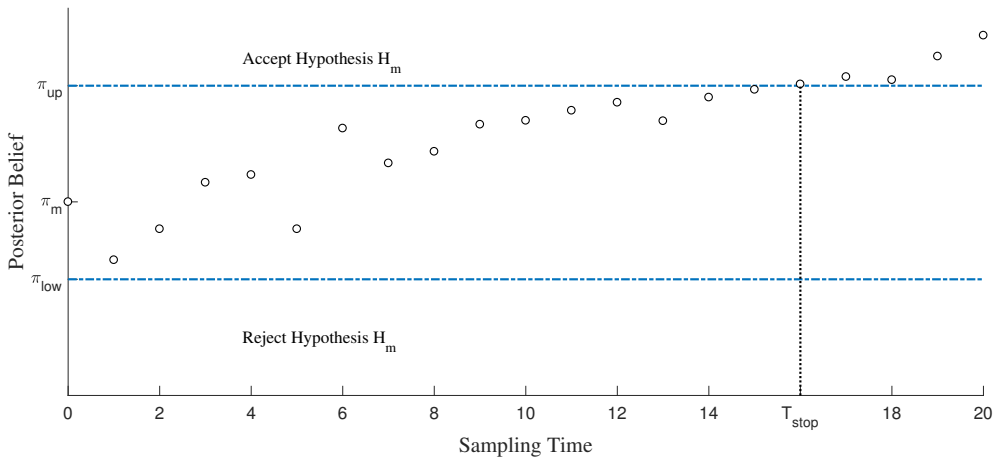


Figure 7.2: An example of stopping time.

weight of λ per process, and define the cost at time t as

$$c(t) = \begin{cases} \lambda \sum(\mathbb{1}(i) \cdot c_i) & \text{without a designed target average cost,} \\ \lambda|\overline{c} - \sum(\mathbb{1}(i) \cdot c_i)| & \text{with a designed target average cost } \overline{c} \end{cases} \tag{7.16}$$

where $\mathbb{1}(i) \in 0, 1$ indicates whether sensor i is selected at time t, c_i is the cost of using sensor i, λ is a predefined weight factor to control the influence that the cost can have on the reward function, and $\overline{c}$ $(0 < \overline{c} < \sum c_i)$ denotes the predefined average cost that the agent targets. That is to say, the predefined average cost is considered as a soft constraint on the cost consumption, and the agent aims at fully utilizing the budget but not exceeding it.

7.3 Learning-Based Solutions

7.3.1 Workflow

To handle the hypothesis testing problem, and jointly control the potential risks during the detection, we in this chapter propose the detection scheme shown in Fig. 7.3. The detection scheme consists of three parts: the environment, a decision-maker,

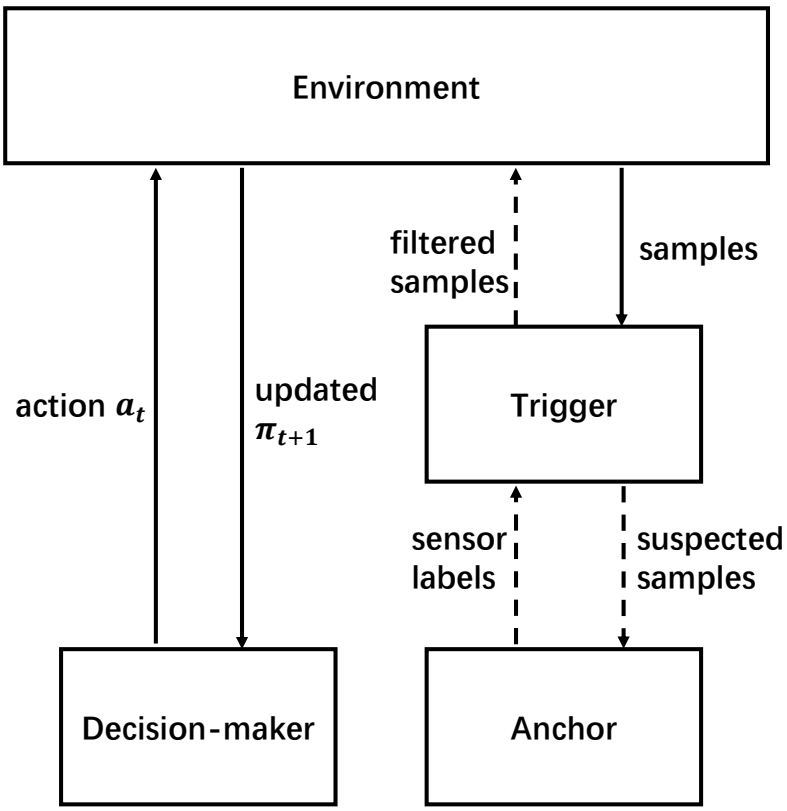


Figure 7.3: Workflow of the learning-based detection scheme.

and a byzantine detector which includes a trigger and an anchor node.

Environment: The environment consists of the process and all the sensors. In the environment, the samples and the feedback after executing the actions selected by the decision-maker can be dynamically generated, and the state of the process updates at the beginning of each episode.

Decision-maker: Based on the observations, the decision-maker is responsible for dynamically selecting sensors to sequentially probe the process, and for terminating the probing when the confidence level exceeds a predefined threshold and detecting the state of the process. The decision-maker is based on the soft actor-critic reinforcement learning algorithm, and it aims to detect the process state as accurately as possible while controlling the probing cost. Also, the algorithm is supposed to be able to work robustly in the presence of additional uncertainty that can be caused by increased

noise/interference (e.g., due to jamming attacks).

Trigger and Anchor Nodes: The trigger and anchor nodes are employed as the Byzantine detectors when there are potential Byzantine sensors in the system. The two parts are designed to identify the Byzantines and eliminate the samples collected by those sensors. Specifically, the trigger will first inspect every newly collected sample, and report the suspected samples to the anchor node. Then, the anchor node will compare the suspected samples with the samples that are collected by the anchor node. Here, the anchor node can be either a unit of the SAC-based decision-maker or an independent remote sensor, and it can collect state signals from the target process. We assume the anchor node is reliable and due to its high reliability, the probing cost of anchor node is very high. Therefore, the anchor node is not used for probing the process, and there is a trigger employed to reduce the usage of anchor node in identifying the Byzantines. In this work, since the distribution of the process states and the sensors' information are unknown to the agents, and the number of samples is limited, we apply the GAN algorithm to reconstruct the distribution of samples in each state and take advantage of the reconstructed features to identify the Byzantines.

7.3.2 Decision-maker: SAC-based Agent

In this section, we describe the proposed soft actor-critic learning framework for the considered detection problem.

7.3.2.1 Preliminaries

We first introduce the relevant definitions within the framework.

Agent's Observation and State: Since the agent can only have observations from the selected sensors/processes, the problem can be modeled as a partially observable Markov decision process (POMDP). With the given observations, the agent can up-

date the posterior belief π^t according to (7.6). And we take the reward r_t obtained by the selected action a_i as the state (or input) of the agent, and we denote the state at time t as $\mathcal{O}_t$. The state $\mathcal{O}_t$ is a $1 \times M$ vector, and each element $\mathcal{O}_{t,i}$ denotes the observation obtained by taking the action a_i at time t, which is defined as

$$\mathcal{O}_{t,i} = \begin{cases} r_t & \text{if action } a_i \text{ is selected at time } t \\ 0 & \text{otherwise} \end{cases} . \tag{7.17}$$

The definitions of action a_i and reward r_t are introduced below. And we assume that the agent can keep at most $\mathcal{M}$ latest observations.

Action: We denote the action space as $\mathcal{A}$, in which all valid actions are included. Since the agent can select any combination of k sensors at a time ($k \in \{1, 2, ..., N\}$), the size of the action space is $|\mathcal{A}| = \sum_{k=1}^{N} \binom{N}{k}$, and a valid action a stands for selecting the corresponding sensors and receiving the samples to update the posterior belief. In each iteration, the agent will estimate the probability distribution of selecting each valid action, and choose the one according to the estimated distribution to execute.

Reward: Since the decision-making agent aims to reach the confidence level as soon as possible, it should maximize the accumulated reward from the first time slot to the stopping time T_{stop} in an episode. So we define the immediate reward r_t as

$$r_t = \begin{cases} r_{\mathcal{C}}(t) - c(t) & \text{if LLR-based reward is employed} \\ r_{\mathcal{H}}(t) - c(t) & \text{if entropy-based reward is employed} \end{cases} \tag{7.18}$$

and the accumulated reward is expressed as

$$r_{1:T} = \begin{cases} \mathcal{C}(\pi(T)) - \mathcal{C}(\pi(1)) - \sum_{t=1}^{T} c(t) & \text{LLR-based reward} \\ \mathcal{H}(\pi(1)) - \mathcal{H}(\pi(T)) - \sum_{t=1}^{T} c(t) & \text{entropy-based reward} \end{cases} . \tag{7.19}$$

Here, we define the state $\mathcal{O}_T$ as the terminal state if any of the M hypothesis is claimed to be accepted, i.e., $\max(\pi^{T-1}) \geq \pi_{\text{upper}}$. And when we update the agent, we consider a weighted reward R_t at time $t \leq T$, as a discounted sum of the rewards

$$R_t = \sum_{\tau=t}^{T} \eta^{\tau-t}(r_\tau), \tag{7.20}$$

so that each previous selection that can lead to better future steps will achieve a greater reward. In the implementation, the agent will be updated T times after the terminal state has been reached, using the weighted reward achieved at the terminal time T, and all the way back to the initial time $t = 0$.

7.3.2.2 Soft Actor-Critic Algorithm

In this subsection, we describe the architecture of the soft actor-critic algorithm. The soft actor-critic architecture consists of three neural networks: policy network, Q network, and value network. These three networks will not share any neurons but exchange information to update each other.

Policy network: The policy network is employed to explore a policy μ that maps the agent's observation $\mathcal{O}$ to the action space $\mathcal{A}$:

$$\mu_\phi(\mathcal{O}) : \mathcal{O} \rightarrow \mathcal{A}. \tag{7.21}$$

So the mapping policy $\mu_\phi(\mathcal{O})$ is a function of the observation $\mathcal{O}$ and is parameterized by ϕ. The chosen action can be denoted as

$$a = \mu_\phi(\mathcal{O}) \tag{7.22}$$

where we have $a \in \mathcal{A}$. Since the action space is discrete, we use the softmax function at the output layer of the policy network so that we can obtain the scores of each

action. The scores sum up to 1 and can be regarded as the probabilities of obtaining a good reward when the corresponding actions are chosen.

Q network: The Q network Q_θ, parameterized by θ, is an approximator to the soft Q function. It is fed the $(\mathcal{O}, a)$ pairs, and it estimates the corresponding Q value. The Q network encourages the policy to converge to the real Q value distribution instead of converging to a promising action. In this way, the agent tends to explore the environment more and engage in effective exploration strategies.

Value network: The value network $V_\psi(\mathcal{O})$ is parameterized by ψ, and it estimates the soft values of the given states. Since the estimated state value indicates the potential future reward, the value network encourages the policy to exploit the promising actions that are learned from the experience.

Update: To update the neural networks, we adopt a memory $\mathcal{D}$ to store the historical transitions, and sample a minibatch at every iteration. And all three neural networks are updated using stochastic gradient descent.

The value network is updated by minimizing the squared residual error

$$J_{V_\psi} = \mathbb{E}_{\mathcal{O}_t \sim \mathcal{D}}[\frac{1}{2}(V_\psi(\mathcal{O}_t) - \mathbb{E}_{a_t \sim \mu_\phi}[Q_\theta(\mathcal{O}_t, a_t) - \log \mu_\phi(a_t|\mathcal{O}_t)])^2]. \tag{7.23}$$

The Q network is updated by minimizing the soft Bellman residual

$$J_{Q_\theta} = \mathbb{E}_{(\mathcal{O}_t, a_t) \sim \mathcal{D}}[\frac{1}{2}(Q_\theta(\mathcal{O}_t, a_t) - \hat{Q}_\theta(\mathcal{O}_t, a_t))^2] \tag{7.24}$$

where $\hat{Q}_\theta(\mathcal{O}_t, a_t) = r(\mathcal{O}_t, a_t) - c_t + \gamma \mathbb{E}[V_\psi(\mathcal{O}_{t+1})]$.

The policy network is trained by minimizing the expected KL-divergence

$$J_{\mu_\phi} = \mathbb{E}_{\mathcal{O}_t \sim \mathcal{D}}[D_{KL}(\mu_\phi(\cdot|\mathcal{O}_t)||\text{softmax}(Q_\theta(\mathcal{O}_t, \cdot)))]. \tag{7.25}$$

The full framework is described in Algorithm 11 below.

Algorithm 11 Soft Actor-Critic Algorithm for Detection

```
t = 0
Initialize the value network V_ψ(O), Q network Q_θ(O, a) and the policy network
μ_ψ(O), parameterized by ψ, θ and ϕ, respectively.
The agent initializes the memory D.
for T = 1 : Maximum episode do
    t_start = t
    Generate a new hypothesis H_j to be true according to the prior belief π, and
    j ∈ {0, 1, ..., M − 1}.
    The agent fetches the prior belief vector π as the initial state.
    while O_T is not a terminal state do
        t ← t + 1
        With the state O_t, the agent selects one out of the N sensors according to
        the decision policy a_t = μ_ϕ(O_t) w.r.t. the current policy.
        Agent receives the samples Y_t from the chosen sensor and update the pos-
        terior belief vector π^t.
        With the new state O_{t+1}, the agent obtains a reward r_t and cost c_t.
        Update the state O_t = O_{t+1}.
    end while
    R = 0
    for τ = t − 1 : t_start do
        R ← r_τ + η ∗ R
        Update the neural networks according to eq. (7.23), (7.24) and (7.25).
    end for
    Reveal the true hypothesis, and check the accuracy of detection.
end for
Save the trained neural networks.
```

7.3.3 Trigger and Anchor: GAN-based Byzantine Detector

In this section, we consider the scenario that the decision-maker is working in the presence of Byzantine attacks. It is assumed that when a sensor is under attack, this honest sensor becomes a Byzantine. And it is also assumed that the number of Byzantines, $k \in \{0, 1, \ldots, N\}$ is unknown to the decision-maker. Same as the honest sensors, the Byzantine sensors quantize the samples using the thresholds in (7.2). However, the Byzantine sensors always flip the samples after quantization according to a pattern. For instance, when the quantized sample is s_i, the Byzantine sensor will send s_j to the decision-maker, and if the quantized sample is s_j, the Byzantine sensor

will send s_i to the decision-maker. Here, we have $i, j \in \{1, 2, \ldots, M\}$ and $i \neq j$.

Ideally, since all the sensors are probing the same process, the samples collected by the sensors should be the same in the absence of noise. Therefore, compared to the sample sent by the anchor node, the sensors that send different signals are the Byzantines. However, we consider both the sensor noise and the noise in transmission channels in this work. Such noise makes the samples collected by each sensor be distributed with different variances. The mean values will also be different when Byzantines send different signals. Hence, we need to estimate the mean value of the samples before comparing with samples from the anchor node. To obtain reliable mean values, a large number of samples may be needed. In practice, the posterior probability may reach the stopping criteria before the decision-maker collects sufficient samples. To solve this problem, we propose a GAN-based detector to estimate the mean values using a single sample.

7.3.3.1 Trigger and Anchor Node

In Fig. 7.4, the structure of the GAN-based detector is depicted. The detector consists of two parts: a trigger and an anchor node. Both parts are based on the GAN: there will be two samples that are collected by different nodes to be compared using the GAN. Here, the input "sample 1" comes from the suspected sensor, and the input "sample 2" comes from the reference sensor which is introduced in the below. At each step, the trigger is employed to screen all the samples and report the suspected samples. Once a suspected sample is reported, the corresponding sensor will be a suspected sensor and all the other sensors will be the reference sensors. The features of "sample 1" and "sample 2" will be reconstructed by the generator. Then, the two features will be the inputs of the discriminator to calculate a loss between these two samples. We assume that in each episode, GAN can keep a record of $\mathcal{B}$ latest samples from each sensor. Hence, the latest sample of each suspected sensor

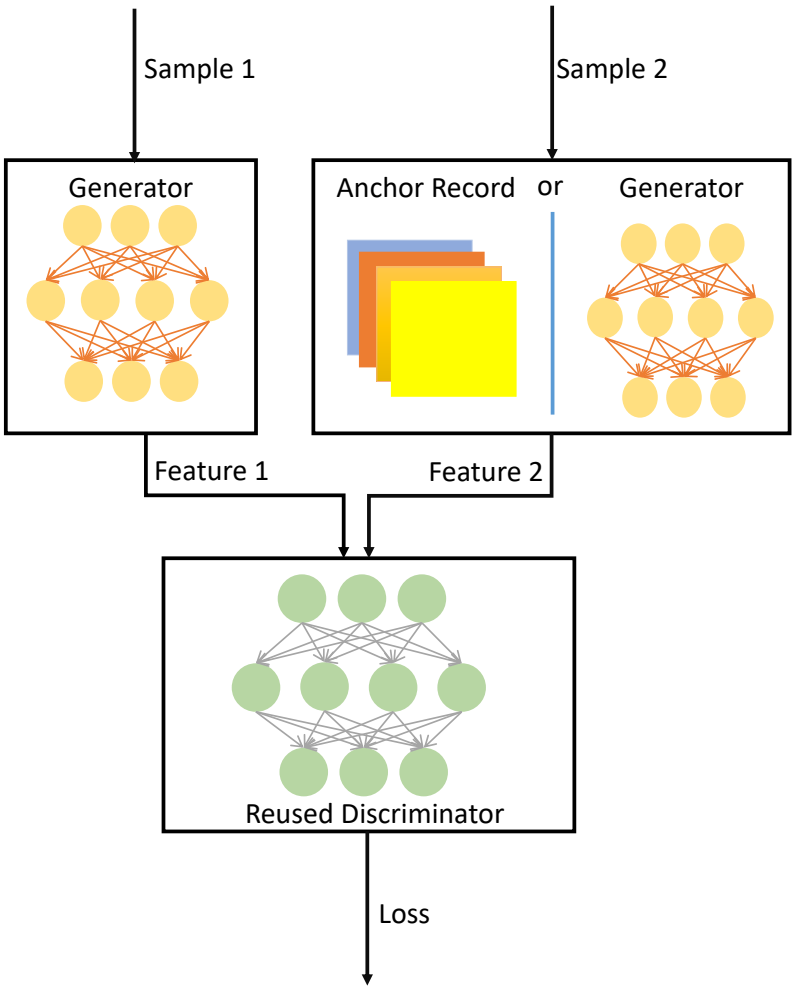


Figure 7.4: Structure of the GAN-based byzantine detector.

will be compared with at most $\mathcal{B}(N-1)$ samples from the reference sensors, and after each comparison, a corresponding loss will be obtained. Based on all these losses, the GAN will decide on whether the "sample 1" is a suspected sample.

As shown in Fig. 7.3, only if a sample is labeled as a suspected sample, it will be sent to the anchor node to be the input "sample 1". And once the suspected sample is received, the anchor node will be triggered and take one sample on the process, and this sample will be the input "sample 2". It is assumed that the anchor node is reliable and the GAN has already collected a record of the sample features during the training process. Therefore, the features of "sample 1" is reconstructed using the

generator, and the features of "sample 2" will be extracted from the anchor record. Similar to the trigger, a loss between the two samples will be calculated and the decision on whether the corresponding sensor is a Byzantine will be made. Once a sensor is identified as a Byzantine, the samples collected by it will be eliminated.

7.3.3.2 GAN

The general structure of a GAN is introduced in [126]. A GAN consists of two neural networks: a generator G, that is used to capture the statistical features of the data; and a discriminator D that is used to estimate the probability that a sample comes from the training data rather than the generative model G to evaluate the generative policy.

We first define a sample space $\mathcal{S}$ with a probability distribution $p(s|x)$, where s is a set of samples corresponding to the real data x in the training data set. The generator G maps the sample into the real data space:

$$G(s; \mathcal{G}) : s \rightarrow \hat{x}, \tag{7.26}$$

where $\mathcal{G}$ denotes the parameters of the generator neural network, and the $\hat{x}$ is a projection (of real data x) generated by the generator G.

The discriminator $D(\tilde{x}; \omega)$ estimates the probability of the input $\tilde{x}$ coming from the real data set, where ω denotes the parameters of the discriminator neural network, and the input $\tilde{x}$ can be either the real data x or the generated data $\hat{x}$.

In the training phase, the GAN is trained to take the samples generated according to both safe sensor data distribution ($z \sim \mathcal{N}(q_i, \sigma_z^2)$) and the attacked sensor data distribution ($z \sim \mathcal{N}(\tilde{q}_i, \sigma_z^2)$), where $\tilde{q}_i$ is the flipped sample when the actual signal is q_i. In particular, the generator aims at reconstructing the samples' statistics and the discriminator compares the reconstructed statistics with the designed one to guide

the generator. Specifically, in the testing phase, the GAN compares the statistics reconstructed from the samples collected by the sensors and the statistics provided by the anchor node, and decides whether the samples collected by the sensors have the same statistics as the anchor node data.

A good discriminator D is expected to be able to distinguish the generated data from the real data, i.e., the estimated probability should be very small if the input is the generated data and should be close to 1 if the input is from the real data. Therefore, the discriminator aims at minimizing the objective function given as

$$\mathcal{L}_D = -(\log(D(x)) + \log(1 - D(\hat{x}))). \tag{7.27}$$

On the other hand, for a generator G that has the goal to learn the real data distribution, the evaluation $D(\hat{x})$ acts as a guidance on the update of the generative model. Thus, a good generator G should be able to make the generated data indistinguishable from the real data to the discriminator D. In addition, since the generator aims at reconstructing the variance of the samples, the difference in the variance of generated data and anchor data should be considered in the loss function. For this purpose, the generator G seeks to minimize the following objective function:

$$\mathcal{L}_G = \log(1 - D(\hat{x})) + w \cdot (\text{mean}(\hat{x}) - \text{mean}(x))^2. \tag{7.28}$$

where w is the weight to rescale the difference in mean of the corresponding data. The workflow of GAN is presented in Algorithm 12 below.

Algorithm 12 Workflow of GAN

Initialize the generator $G(s;\mathcal{G})$ with the parameters $\mathcal{G}$, and the the discriminator $D(x;\omega)$, parameterized by ω.

for $T = 1$: Maximum episode **do**

Fetch the sample set $s(t)$ and the corresponding real data set $x(t)$.

Use the generator G to generate the projection of the real data: $\hat{x}(t) = G(s(t);\mathcal{G}(t))$

Feed the projection $\hat{x}(t)$ and the real data $x(t)$ to the discriminator D, and obtain the estimated probabilities of both data being real data.

Update the discriminator by descending the stochastic gradient:

$-\nabla_{\omega}(\log(D(x(t);\omega(t)) + \log(1 - D(\hat{x}(t);\omega(t)))$

Update the generator by descending the stochastic gradient:

$\nabla_{\mathcal{G}} \log(1 - D(\hat{x}(t);\omega(t)) + w \cdot (\text{mean}(\hat{x}) - \text{mean}(x))^2$

end for

7.4 Simulation Results

7.4.1 Experimental Settings

7.4.1.1 Environment

In the experiments, the target process has four possible states (i.e., $M = 4$) and signal values for these states are denoted as $0, 0.25, 0.5, 1$. We set the number of sensors to be $N = 3$, and the sensor noise power vector is $[0.2, 0.1, 0.05]$ and the corresponding cost vector is $[0.1, 0.5, 1]$. Hence, sensors with smaller noise power have higher cost. The noise power in the communication links can be selected from the set $\{0.05, 0.2, 0.4, 0.6\}$. Normally, the channel noise power is assumed to be known to the decision-maker in the absence of any interference sources. However, when the channel is has interference from other transmitters or experiences jamming attacks, the actual channel noise power becomes unknown to the decision-maker. In this case, the decision-maker underestimates the noise power by assuming it to be equal to 0.05, and uses this noise power level to update the posterior probabilities π. For every episode, if a sensor becomes a Byzantine, that sensor will swap the signal among the two pairs: $(0, 0.5)$ and $(0.25, 1)$.

Table 7.1: Configuration of Soft Actor-Critic algorithm.

	Policy network	Q network[1]	Value network
Input layer	200 neurons + ReLU	(200 + 200) neurons + ReLU	200 neurons + ReLU
Hidden layer	200 neurons + ReLU 200 neurons + ReLU 200 neurons + ReLU	200 neurons + ReLU 64 neurons + ReLU	200 neurons + ReLU 100 neurons + ReLU
Output layer	$(2^N - 1)$ neurons + softmax	1 neuron	1 neuron
Learning rate	e^{-5}	e^{-4}	e^{-4}

[1] Since both the observation and the action are taken as inputs of the Q network, the two components are loaded to the neural network through separate entries. Then, the extracted features of the observation and action will be merged in the second layer, and an estimated soft Q value will be given at the output layer.

7.4.1.2 Neural Networks

The configuration of the proposed soft actor-critic (SAC) framework is provided in Table 7.1. For comparison purposes, we also implement the conventional actor-critic (AC) algorithm. The configuration of the AC framework is provided in Table 7.2. The implementation of the AC framework is also explored in [127]. In the experiments, both the LLR-based reward and entropy-based reward will be considered for the actor-critic framework. The configuration of GAN is provided in Table 7.3.

Table 7.2: Configuration of Conventional Actor-Critic Algorithm

	Actor	Critic
Input layer	200 neurons + ReLU	200 neurons + ReLU
Hidden layer	100 neurons + ReLU	100 neurons + ReLU
Output layer	$(2^N - 1)$ neurons + Softmax	1 neuron
Learning rate	$5e^{-5}$	$1e^{-4}$

Table 7.3: Configuration of GAN

	Generator	Discriminator
Input layer	128 neurons + ReLU	128 neurons + tanh
Hidden layer	128 neurons + ReLU 256 neurons + ReLU 256 neurons + ReLU	128 neurons + tanh 64 neurons + tanh
Output layer	(feature size) neurons	1 neuron + sigmoid
Learning rate	e^{-5}	$5e^{-3}$

7.4.2 Numerical Results

7.4.2.1 Preliminary Results

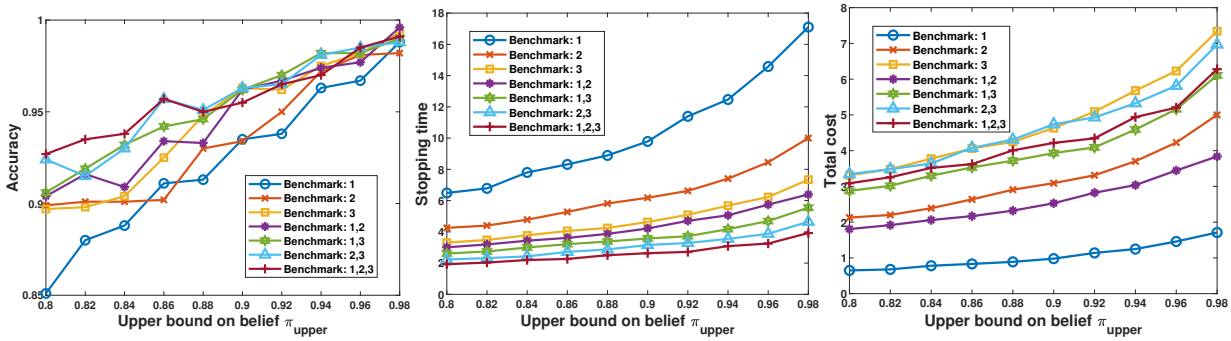


Figure 7.5: Performance of the Benchmarks when π_{upper} is varied from 0.8 to 0.98, $\lambda = 0, \sigma_z^2 = 0.05$.

To illustrate how the selection of sensors influences the performance of the decision-maker, we first investigate the naive policies as a benchmark. The naive policies select their preferred sensors all the time. Since $N = 3$, there are 7 different naive policies, and each of them selects a different set of sensors: {{1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}. We consider the detection accuracy, average stopping time and the average total cost over 10000 episodes as the performance metrics. In Fig. 7.5, we set $\lambda = 0$, and $\sigma_z^2 = 0.05$, and plot the performance metrics as a function of the confidence threshold π_{upper}. As shown in the figure, all the three performance metrics grow as the predefined confidence level increases. As noted before, the sensor 1 has the highest noise power and the lowest cost. So, it can be observed that the policy "Benchmark:1" achieves the lowest accuracy and the lowest cost, but the highest stopping time. Also, it can be observed that when the π_{upper} is low, the accuracy achieved by the policies whose selections include sensor 3 are relatively higher. Correspondingly, the stopping times are lower and the total costs are higher.

These observations indicate that to reach the stopping criteria, the decision-maker needs to obtain sufficient information on the sample distribution, and both the num-

ber of samples and the reliability of the selected sensors can influence the performance. Therefore, the proposed learning-based decision-maker is expected to be able to distinguish the most reliable sensor from the feedback and control the probing cost.

In Fig. 7.6, we illustrate the performance of the proposed SAC decision-maker, and we compare its performance with the AC-based decision-maker. Both DRL-based algorithms are tested with two types of reward functions: LLR and entropy. Generally, the performance of the DRL-based algorithms are competitive. When compared to the benchmarks, the DRL-based algorithms are similar to the "Benchmark:1,2" and "benchmark: 1,3". For the SAC-based algorithms, the policy with the LLR reward performs better than the policy with the entropy reward in terms of the stopping time. For the AC algorithm, the policy with the LLR reward achieves a higher accuracy then the policy with the entropy reward. It can also be observed that for each type of reward, considering the three performance metrics, the SAC-based algorithms are slightly better than the AC-based algorithms.

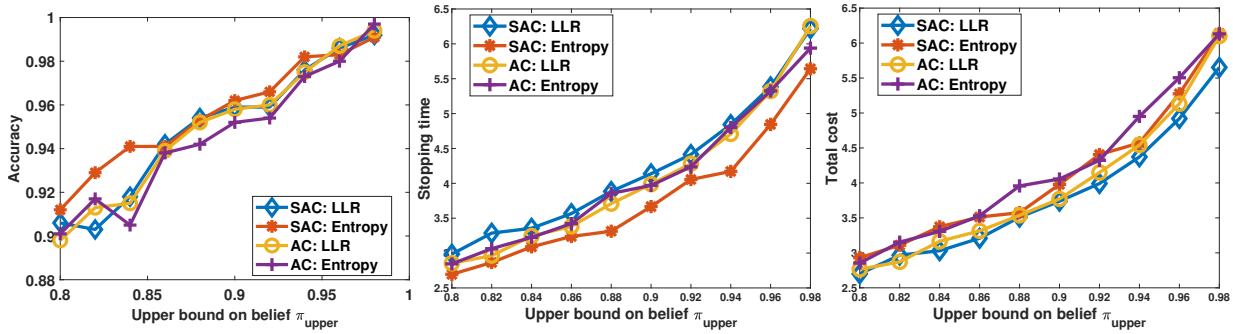


Figure 7.6: Performance of the learning-based decision-makers when π_{upper} is varied from 0.8 to 0.98, $\lambda = 0, \sigma_z^2 = 0.05$.

7.4.2.2 Jamming Attacks and Increased Noise Power

We consider the scenario in which the transmission channel is attacked by a jammer during the experiment and the actual channel noise σ^2 is unknown to the decision-maker. In this experiment, the actual channel noise power varies as $\sigma^2 \in \{0.05, 0.2, 0.4, 0.6\}$,

but the decision-maker always uses $\sigma_z^2 = 0.05$ to update the posterior probabilities. To illustrate the influence of not knowing the actual noise power, we also test the algorithms with the same channel noise powers but under the assumption that this information is known to the decision-makers. In Fig. 7.7, we plot the performance metrics as functions of the actual channel noise power σ^2. For all the decision-makers that we have tested, as σ^2 increases, the performances become worse: the accuracy decreases rapidly and both the stopping time and total cost decrease slightly. The degraded performance can be attributed to the uncertainty introduced by the channel noise. Specifically, the incorrect noise power gives inaccurate computation results in the update of the posterior probabilities. The smaller σ_z^2 that the decision-maker employs, the quicker the growth in confidence level will be. As a consequence, when the channel is attacked, the decision-maker is misled by the unknown noise and it stops taking samples quickly and claims the wrong process states. Nevertheless, if we compare the performances of SAC and AC based policies, we observe that performance degradation in the SAC policies is slightly smaller than that in the AC policies. This observation shows that the SAC algorithms have higher robustness in a noisy environment, and can be more effective against increased interference levels and jamming attacks.

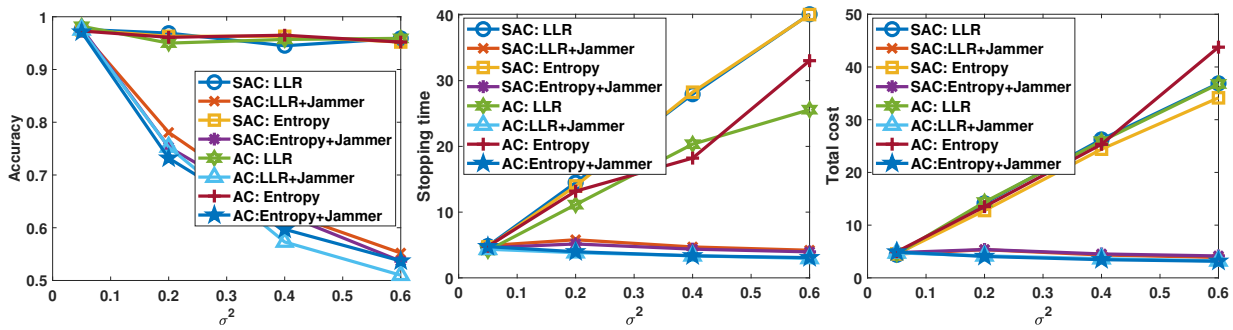


Figure 7.7: Performance of the learning-based decision-makers when the σ^2 varies from 0.05 to 0.6, $\lambda = 0, \pi_{\text{upper}} = 0.94$.

7.4.2.3 Cost Control

We first consider the cost function without a predefined average cost. It should be noted that, in the experiments, the sensor 1 is the least costly but also the least reliable sensor, while the sensor 3 is the most reliable sensor with the highest probing cost. As shown in the previous experiments, there is no obvious difference in the performance between the LLR reward-based policies, and the entropy reward-based policies. So, we only demonstrate the performance of the LLR reward-based policies in experiments in this subsection. The performance metrics are studied as functions of π_{upper}. And the performance achieved by the same algorithm but with different values of λ are grouped. In Figs. 7.8 and 7.9, the performance of SAC algorithm and AC algorithm are plotted respectively.

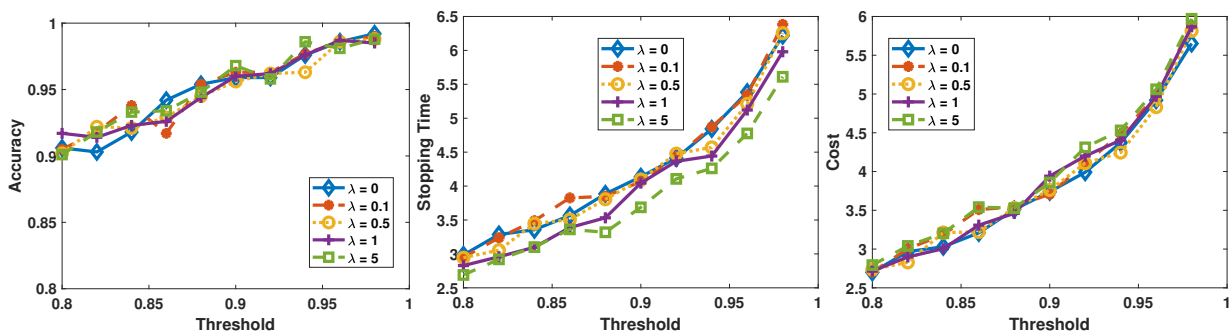


Figure 7.8: Performance of the SAC algorithm when the λ varies from 0 to 5, π_{upper} is varied from 0.8 to 0.98, $\sigma^2 = 0.05$.

Obviously, the AC-based algorithm is more sensitive to the change in λ: when the value of λ increases, the average total cost decreases. Intuitively, the reason for this performance can be that when a higher cost is received, the AC-based algorithm tends to select the sensors with lower cost more frequently. Consequently, the more reliable sensors are selected less frequently, and therefore the decision accuracy decreases and the stopping time increases. As to the SAC-based decision-makers, the performance varies slightly as λ changes because the SAC-based algorithm prefers the most reliable

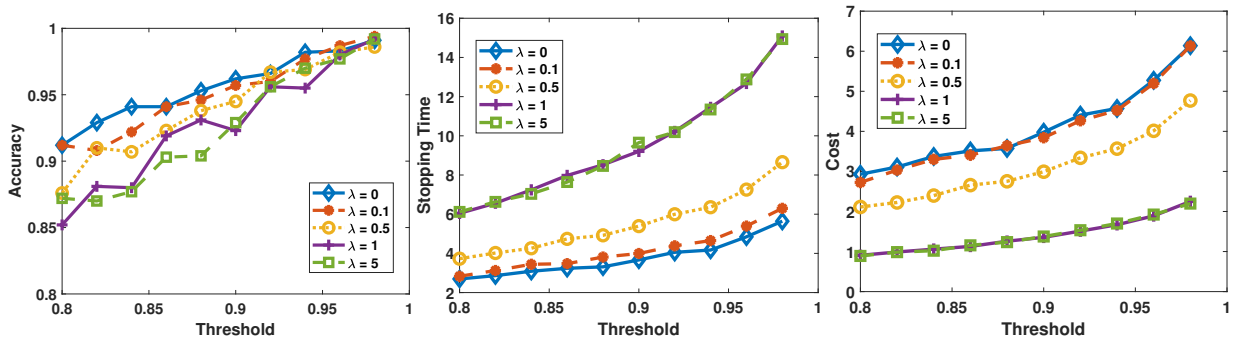
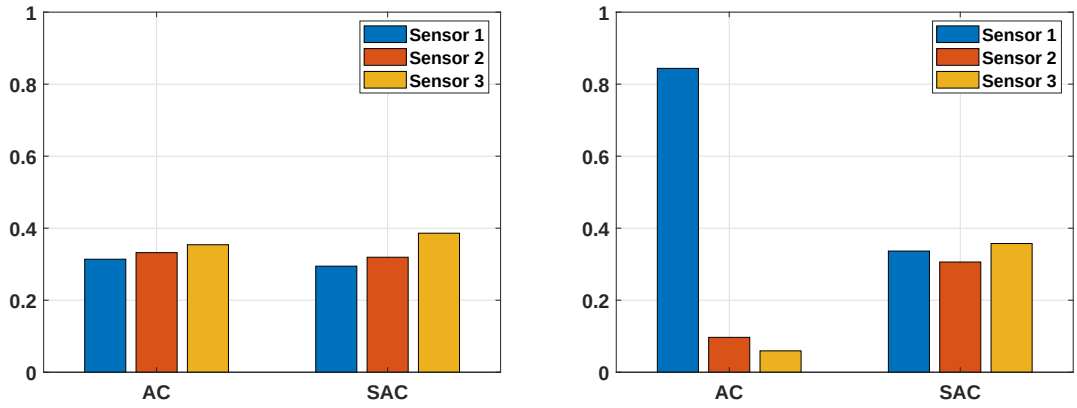


Figure 7.9: Performance of the AC algorithm when the λ varies from 0 to 5, π_{upper} is varied from 0.8 to 0.98, $\sigma^2 = 0.05$.

sensor more frequently. To verify this, in Fig. 7.10 we plot the fractions of samples coming from each sensor. Comparing the two figures, we find that when there is no cost ($\lambda = 0$, in Fig. 7.10a), the two algorithms' policies are similar and the probabilities of selecting each sensor are close to each other. However, when there is a large λ (in Fig. 7.10b), the AC algorithm selects sensor 1 with a probability as high as 0.84, whereas the SAC algorithm still selects sensor 3 with the highest probability even though the probability of selecting sensor 1 increases.

The two algorithms have different reactions to the change in λ, because of the different strategies that are employed by them. The SAC algorithm is an off-policy maximum entropy DRL algorithm. To better explore the environment, the SAC algorithm favors a stochastic policy and aims at obtaining a more dispersed distribution in action probabilities. As to the AC algorithm, a more deterministic policy is pursued, which means that in a specific state, the AC algorithm only considers one action as the optimal solution. The above-mentioned characteristic of the SAC algorithm makes it more stable in diverse settings and have less risk in overftting to any local optimums. In this test case, the AC algorithm changes its policy dramatically as the λ changes. Compared to the AC algorithm, the SAC algorithm is more competitive in handling a change over a wider range.

To take the advantage of the more stochastic policy, we set a designed average cost



(a) $\lambda = 0$ (b) $\lambda = 5$

Figure 7.10: Distribution of sensor selection, $\pi_{\text{upper}} = 0.98, \sigma_z^2 = 0.6$.

$\bar{c} = 1$, and feed back the corresponding cost to the decision-maker. In Fig. 7.11, for the two DRL-based algorithms, we plot the moving average of cost $c_t = \frac{1}{W}\sum_{\tau=t}^{t+W} c(\tau)$, where the window size W is set to be 100. We observe that the moving average costs of both the proposed SAC algorithm and AC algorithm vary in a small range around the designed average cost. The difference is that the moving average cost of the SAC algorithm fluctuates in a wider range but the overall average cost is below the designed average cost, while the moving average cost achieved by the AC algorithm fluctuates in a narrower range but the overall average cost exceeds the designed average cost. In Table 7.4, we provide the corresponding detection accuracy and stopping time achieved by the two algorithms. In terms of both performance metrics, the SAC algorithm shows advantages over the AC algorithm. This is because the SAC algorithm is more flexible in the selection of actions, so that it can take better advantage of selecting the most reliable sensor intermittently to ensure the accuracy and reduce the detection delay at the same time.

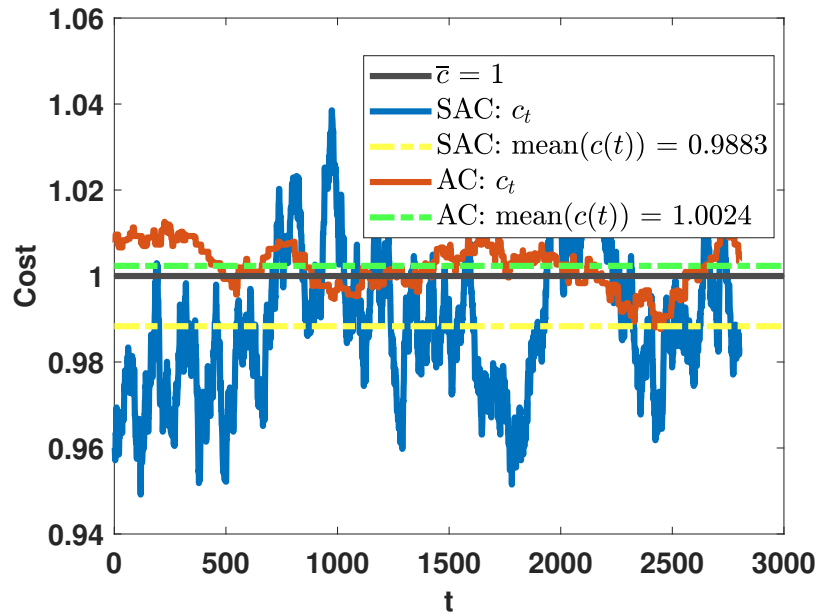


Figure 7.11: Performance in terms of cost, when the designed average cost is set as $\overline{c} = 1$.

Table 7.4: Performance of learning-based decision-makers with the designed average cost is set to be 1.

	Accuracy	Stopping time
SAC	0.9760	4.6020
AC	0.9630	5.5370

7.4.2.4 Byzantine

In this subsection, we consider the scenario in which the decision-maker operates in the presence of Byzantine sensors. It is assumed that at the beginning of each episode, there is an attacker randomly deciding whether to attack an honest sensor to make it a Byzantine or not. It is equally likely for the attacker to select any one from the three sensors or decide not to attack. Once the decision is made, the state of sensors (honest or Byzantine) will remain fixed during the rest of the episode. And as noted before, we assume there is an anchor node which is always honest and the anchor node obtains a record of the features from the training process of the GAN detector.

In Fig. 7.12, we show the performance of the proposed GAN-based Byzantine

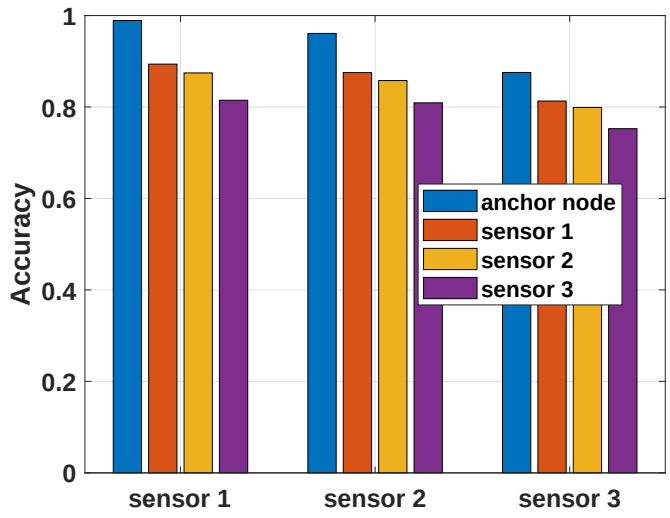


Figure 7.12: Accuracy of the GAN-based detector, when $\sigma_z^2 = 0.1$.

detector in terms of the detection accuracy. Since each sensor has its own noise power, the accuracy of different <suspected sensor, reference sensor> pairs will be different. Here, we consider all possible combination of <suspected sensor, reference sensor> pairs. It should be noted that we also provide the accuracy achieved by the <sensor i, sensor$i$ > pairs for $i = 1, 2, 3$. In our aforementioned assumptions, the sensors cannot be the reference sensors for themselves. Therefore, in such sensor pairs, we assume that the suspected sensor and the corresponding reference sensor are identical and the states of the two sensors are independent. We only consider this situation in the test of GAN detector's accuracy, and in the detection of the process state, the suspected sensor and the reference sensor must be two different sensors. We notice in the figure that employing the anchor node as the reference sensor always achieves the highest accuracy. And it can be also observed that noise power at the suspected sensor and the reference sensor is inversely proportional to the accuracy.

With the pre-trained GAN-based detector, we investigate the performance of the defense strategy. When the sensors are employed as the reference sensors, the detection accuracy actually refers to achieved by the trigger. Since the reference sensors

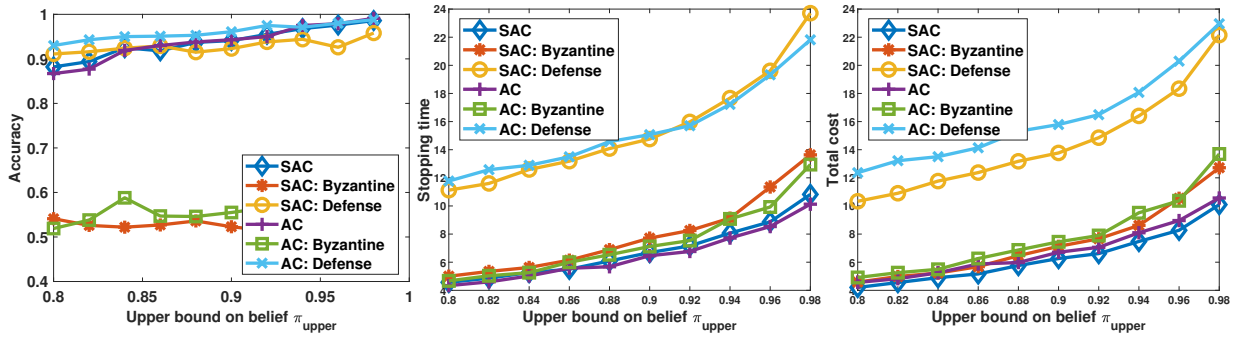


Figure 7.13: Performance of the decision-makers work in absence of Byzantine attacks, with the presence of Byzantine attacks, and with the presence of both Byzantine attack and GAN-based Byzantine detector. when the $\lambda = 0$, $\sigma_z^2 = 0.1$.

are also noisy sensors, there is a probability that the samples from the reference sensors being distorted. To improve the accuracy of the trigger in the testing phase, instead of only comparing the suspected sensor to only one reference sensor, we take all available reference sensors into consideration. Specifically, the sample from the current suspected sensor is compared with samples from all available reference sensors, and each comparison result is a decision on whether the suspected sensor is a Byzantine. Then, the majority decision is the final decision on the identity of the suspected sensor. With this "trigger-anchor" two-level detection, the Byzantines are identified and the corresponding samples are eliminated. In Fig. 7.13, we plot the performance achieved in three cases. Specifically, "SAC/AC" refers to the situation in which the corresponding decision-maker works in the absence of Byzantine attacks, "SAC/AC : Byzantine" stands for the case in which the decision-maker operates in the presence of Byzantine attacks but there is no defense strategy, and "SAC/AC : Defense" denotes the scenario in which the decision-maker works in the presence of Byzantine attacks but employs the defense strategy.

Considering the accuracy of detecting the process state, we observe that the proposed defense strategy can successfully recover the performance to the level achieved when no Byzantine attacks are executed. According to the decisions made by the

GAN-based detector, the samples from the sensors which are labeled as Byzantine are removed. Therefore, to obtain sufficient information to reach the stopping criteria, more probing steps are taken. Consequently, there are obvious increments in both the stopping time and total cost. It can also be observed that with the GAN-based detector, the accuracy achieved by "AC:Defense" scheme is higher than the "AC" scheme. This is because when the samples are too noisy and the sensors quantize them into incorrect states, the GAN-based detector can also take the sensors as Byzantines and remove the samples. With the misleading samples eliminated, the improvement in accuracy is expected.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have studied learning-based decision making strategies in wireless communications, addressing edge caching, dynamic multichannel access, adversarial jamming attacks, and anomaly detection problems. The contributions of this thesis are summarized below.

In Chapter 2, we have investigated the application of actor-critic DRL algorithm in edge caching problems in both single-cell and multi-cell wireless scenarios.

- In Section 2.1, we have proposed and developed a deep reinforcement learning based content caching policy. We built the framework based on the Wolpertinger architecture and trained it using the deep deterministic policy gradient. We have evaluated the performance of the proposed framework and compared it with both short-term and long-term cache hit rates achieved with LRU, LFU, and FIFO policies. The results show that the proposed framework provides improvements on both short-term and long-term performance. Additionally, we have further confirmed the effectiveness of the proposed framework by comparing the cache hit rate and runtime with those achieved with the deep Q-learning

based policy. This comparison has demonstrated that the proposed framework can achieve competitive cache hit rates while effectively reducing the runtime. This makes the proposed framework efficient in handling large-scale data.

- In Section 2.2, we have focused on edge caching in both single-cell and multi-cell scenarios. In particular, we have designed deep actor-critic reinforcement learning frameworks for both centralized and decentralized edge caching scenarios. More specifically, we have employed the Wolpertinger architecture involving an actor neural network, a KNN component, and a critic neural network. We have described in detail how the neural networks are updated. We have developed a single-agent actor-critic algorithm in the single-cell scenario and described its workflow. In the multi-cell setting, we have proposed a decentralized edge caching strategy via a multi-agent framework with multiple actor networks and a single critic network. In this setting, we have designed a multi-agent actor-critic algorithm. We have provided simulation results to test the performance of the proposed frameworks. For the centralized edge caching scenario, we have analyzed the performance in terms of the cache hit rate as a function of the cache ratio, Zipf exponent, and the number of files. For decentralized edge caching in a multi-cell environment, we have considered two objectives: cache hit rate and transmission delay reduction. We have studied the performance in terms of both objectives again as the cache ratio, Zipf exponent, and the number of files vary. We have also evaluated the reinforcement learning agents' adaptation capabilities in the presence of unknown change points where users' preferences change randomly. In all of the experiments, the proposed actor-critic frameworks have shown advantage over the non-learning based policies, leading to benefits and improvements in terms of cache hit rate, transmission delay reduction, adaptation capability and long-term stability.

In Chapter 3, we have considered the dynamic multichannel access problem mod-

eled as a POMDP. To effectively find the channel access policy, we have proposed and implemented model-free actor-critic deep reinforcement learning frameworks in single-user and multi-user scenarios. We have tested the single agent framework on round-robin and arbitrary switching scenarios, and compared the average reward with that of the DQN framework, random access police, Whittle index heuristic and optimal policy. Also, we have studied the performance of the proposed framework in cases in which multiple different channels are selected simultaneously. We have demonstrated the proposed framework's superior ability in handling a large number of channels, high tolerance against uncertainty, and large action spaces. In the multi-user case, we have addressed models with users operating with or without priorities. For users without priority, we have presented results on the average sum reward to demonstrate the decentralized actor-critic agents' capability to learn the channel switching patterns as well as the other users' action patterns. In the case of users with priority, we have computed the distribution of different channel access results under different channel allocation policies and shown that the proposed framework is competitive in various scenarios. To highlight the adaptive ability, we have conducted simulations in a time-varying environment and demonstrated that the proposed framework learns the new patterns effectively in a relatively short period of time. Finally, we have demonstrated the efficiency of the actor-critic framework by computing the percentage of runtime that can be saved compared to the DQN framework.

In Chapter 4, we have proposed two adversarial wireless jamming attackers aimed at minimizing the accuracy of the dynamic multichannel access performed by a DRL agent. We have introduced the frameworks of the proposed FNN and DRL attackers, and then presented their corresponding RRAS working procedures. Via simulations and numerical results, we have evaluated the performances of the two adversarial policies in terms of the victim's accuracy. In this analysis, we have specifically con-

ducted experiments with a stronger victim that applies the ϵ-greedy policy. Finally, we have identified the advantages and disadvantages of the two frameworks.

In Chapter 5, we have considered active sequential testing for anomaly detection, in which an unknown number of processes could be in abnormal states simultaneously. To solve the dynamic problem of how to select sensors based on a partially observable Markov decision process, we have proposed a deep actor-critic reinforcement learning framework, which enables the agent to dynamically select the sensors and minimize the claim delay while maximizing the confidence level based on the posterior probabilities. We have designed the actor-critic sensor selection algorithm, refining the updating procedure. We have analyzed the performance of the proposed framework. In particular, in the training phase, we have conducted validation testing and demonstrated the convergence of the posterior probabilities. In the testing phase, we have investigated the selection of upper and lower thresholds and their influence on the claim delay and loss. Finally, we have provided comparisons between the proposed framework and Chernoff test, and demonstrate the superior performance of the proposed actor-critic deep reinforcement learning framework in terms of lower claim delay. Additionally, while the Chernoff test has lower loss for smaller values of upper threshold, the proposed framework outperforms when higher confidence levels are required (i.e., for larger values of the upper threshold).

In Chapter 6, we have proposed a framework for anomaly detection and sampling cost control based on GANs. First, we have modeled the detection of threshold crossing in a stochastic time series as an anomaly detection problem. Then, we have proposed a hierarchical GAN structure to address such a detection problem. Specifically, we have designed a hierarchical structure to take advantage of the estimated projection of real samples and described the training and testing workflows. The performance of the proposed hierarchical GAN-based detector has been analyzed considering multiple performance metrics, namely the detection delay, miss rate, cost

of error (due delayed detection) and sampling ratio. We have also provided comparisons between the proposed hierarchical GAN detector and the sampling policy derived with complete statistical information of OU processes. We have shown that the proposed GAN-based detector can have improved performance in terms of detection delays, miss rates, and cost of error as the buffer zone width is increased but at the cost of higher sampling ratios.

In Chapter 7, we have proposed a soft actor-critic (SAC) based reinforcement learning framework to address the detection problem with a cost control. First, we have modeled the sensor probing mechanism in the presence of two-level noise (i.e., noise in sensing and noise in the communication link between the sensor and the decision-making agent) and formulated the problem. We have developed the SAC-based algorithm with two types of rewards and two types of cost functions to reflect the objective and costs in the considered setting. Subsequently, we have considered the random Byzantine attacks on the sensors and designed a GAN-based agent to identify the Byzantine sensors. To evaluate the performance, we have considered three performance metrics: accuracy, stopping time, and total cost. In the experiments, we have compared the proposed SAC framework with the conventional actor-critic (AC) algorithm. Via simulation results, we have demonstrated that the proposed SAC agent can be more robust in different test cases, and the proposed GAN detector is able to identify the Byzantines with high accuracy and help to recover the detection performance achieved in the absence of Byzantine attacks.

8.2 Future Research Directions

8.2.1 Learning-based Adversarial Attacks on Remote Sensor Networks

In Chapter 7, we assume random Byzantine attacks on the considered sensor network. In Fig. 7.13, we observe that the GAN-based Byzantine detector can help the decision-maker to eliminate the unreliable samples and maintain a high detection accuracy. To further verify the proposed detection framework's ability to defend against attacks and identify potential risks in the learning-based system, it is of interest to conduct learning-based adversarial attacks on the decision-maker.

8.2.2 Learning-based Identification for Multivariate Attacks on Remote Sensor Networks

In this thesis, we considered jamming attacks and Byzantine attacks on remote sensor networks separately. However, in practice, the two types of attacks can occur simultaneously. Moreover, there can also be unknown types of attacks. Therefore, it is of utmost importance to develop algorithms that can identify different types of attacks using as small number of samples as possible. Given that meta learning algorithms demonstrate great ability in learning unknown tasks, a combination of meta learning algorithms and GAN can be considered to extent our current Byzantine detector to a more advanced agent for multivariate attack classification.

Bibliography

[1] D. Gündüz, P. de Kerret, N. D. Sidiropoulos, D. Gesbert, C. R. Murthy, and M. van der Schaar, "Machine learning in the air," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 10, pp. 2184–2199, 2019.

[2] G. Zhu, D. Liu, Y. Du, C. You, J. Zhang, and K. Huang, "Toward an intelligent edge: Wireless communication meets machine learning," *IEEE Communications Magazine*, vol. 58, no. 1, pp. 19–25, 2020.

[3] H. Huang, S. Guo, G. Gui, Z. Yang, J. Zhang, H. Sari, and F. Adachi, "Deep learning for physical-layer 5g wireless techniques: Opportunities, challenges and solutions," *IEEE Wireless Communications*, vol. 27, no. 1, pp. 214–222, 2020.

[4] A. Zappone, M. Di Renzo, and M. Debbah, "Wireless networks design in the era of deep learning: Model-based, ai-based, or both?" *IEEE Transactions on Communications*, vol. 67, no. 10, pp. 7331–7376, 2019.

[5] M. E. Morocho-Cayamcela, H. Lee, and W. Lim, "Machine learning for 5g/b5g mobile and wireless communications: Potential, limitations, and future directions," *IEEE Access*, vol. 7, pp. 137 184–137 206, 2019.

[6] Y. Sun, M. Peng, Y. Zhou, Y. Huang, and S. Mao, "Application of machine learning in wireless networks: Key techniques and open issues," *IEEE Communications Surveys Tutorials*, vol. 21, no. 4, pp. 3072–3108, 2019.

[7] Z. Yang, M. Chen, W. Saad, C. S. Hong, and M. Shikh-Bahaei, "Energy efficient federated learning over wireless communication networks," *IEEE Transactions on Wireless Communications*, pp. 1–1, 2020.

[8] H. Ye, L. Liang, G. Y. Li, and B. Juang, "Deep learning-based end-to-end wireless communication systems with conditional gans as unknown channels," *IEEE Transactions on Wireless Communications*, vol. 19, no. 5, pp. 3133–3143, 2020.

[9] I. Sabek and M. F. Mokbel, "Machine learning meets big spatial data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1782–1785.

[10] Y. Liu, S. Bi, Z. Shi, and L. Hanzo, "When machine learning meets big data: A wireless communication perspective," *IEEE Vehicular Technology Magazine*, vol. 15, no. 1, pp. 63–72, 2020.

[11] X. Chen, J. Ji, C. Luo, W. Liao, and P. Li, "When machine learning meets blockchain: A decentralized, privacy-preserving and secure design," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 1178–1187.

[12] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," *arXiv preprint arXiv:1901.03407*, 2019.

[13] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update, 2017-2022 white paper," *Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html, [Online]*, Feb. 2019.

[14] M. A. Kader, E. Bastug, M. Bennis, E. Zeydan, A. Karatepe, A. S. Er, and M. Debbah, "Leveraging big data analytics for cache-enabled wireless net-

works," in *Globecom Workshops (GC Wkshps), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[15] Q. Yan, M. Cheng, X. Tang, and Q. Chen, "On the placement delivery array design for centralized coded caching scheme," *IEEE Transactions on Information Theory*, vol. 63, no. 9, pp. 5821–5833, 2017.

[16] Q. Yang, P. Hassanzadeh, D. Gündüz, and E. Erkip, "Centralized caching and delivery of correlated contents over a Gaussian broadcast channel," in *Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt), 2018 16th International Symposium on*. IEEE, 2018, pp. 1–6.

[17] K. Kvaternik, J. Llorca, D. Kilper, and L. Pavel, "A methodology for the design of self-optimizing, decentralized content-caching strategies," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2634–2647, 2016.

[18] 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, 2018.

[19] Y. Li, C. Zhong, M. C. Gursoy, and S. Velipasalar, "Learning-based delay-aware caching in wireless D2D caching networks," *IEEE Access*, vol. 6, pp. 77 250–77 264, 2018.

[20] W. Wang, D. Niyato, P. Wang, and A. Leshem, "Decentralized caching for content delivery based on blockchain: A game theoretic perspective," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[21] B. Zhou, Y. Cui, and M. Tao, "Optimal dynamic multicast scheduling for cache-enabled content-centric wireless networks," *IEEE Transactions on Communications*, vol. 65, no. 7, pp. 2956–2970, 2017.

[22] J. Kwak, Y. Kim, L. B. Le, and S. Chong, "Hybrid content caching in 5G wireless networks: Cloud versus edge caching," *IEEE Transactions on Wireless Communications*, vol. 17, no. 5, pp. 3030–3045, 2018.

[23] M. Chen, W. Saad, C. Yin, and M. Debbah, "Echo state networks for proactive caching in cloud-based radio access networks with mobile users," *IEEE Transactions on Wireless Communications*, vol. 16, no. 6, pp. 3520–3535, 2017.

[24] T. X. Tran, A. Hajisami, and D. Pompili, "Cooperative hierarchical caching in 5G cloud radio access networks," *IEEE Network*, vol. 31, no. 4, pp. 35–41, 2017.

[25] X. Li, X. Wang, P.-J. Wan, Z. Han, and V. C. Leung, "Hierarchical edge caching in device-to-device aided mobile networks: Modeling, optimization, and design," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 8, pp. 1768–1785, 2018.

[26] M. Leconte, G. Paschos, L. Gkatzikis, M. Draief, S. Vassilaras, and S. Chouvardas, "Placing dynamic content in caches with small population," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[27] W. Li, S. M. Oteafy, and H. S. Hassanein, "Streamcache: popularity-based caching for adaptive streaming over information-centric networks," in *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–6.

[28] S. Li, J. Xu, M. Van Der Schaar, and W. Li, "Popularity-driven content caching," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[29] 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.

[30] R. Pedarsani, M. A. Maddah-Ali, and U. Niesen, "Online coded caching," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 836–845, 2016.

[31] J. Song and W. Choi, "Mobility-aware content placement for device-to-device caching systems," *IEEE Transactions on Wireless Communications*, vol. 18, no. 7, pp. 3658–3668, July 2019.

[32] M. S. ElBamby, M. Bennis, W. Saad, and M. Latva-Aho, "Content-aware user clustering and caching in wireless small cell networks," in *2014 11th International Symposium on Wireless Communications Systems (ISWCS)*. IEEE, 2014, pp. 945–949.

[33] H. Zhu, Y. Cao, W. Wang, T. Jiang, and S. Jin, "Deep reinforcement learning for mobile edge caching: Review, new features, and open issues," *IEEE Network*, vol. 32, no. 6, pp. 50–57, 2018.

[34] L. Lei, L. You, G. Dai, T. X. Vu, D. Yuan, and S. Chatzinotas, "A deep learning approach for optimizing content delivering in cache-enabled hetnet," in *2017 International Symposium on Wireless Communication Systems (ISWCS)*. IEEE, 2017, pp. 449–453.

[35] 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, 2018.

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

[37] Y. Wei, F. R. Yu, M. Song, and Z. Han, "Joint optimization of caching, computing, and radio resources for fog-enabled iot using natural actor–critic deep reinforcement learning," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2061–2073, 2018.

[38] J. Sung, K. Kim, J. Kim, and J.-K. K. Rhee, "Efficient content replacement in wireless content delivery network with cooperative caching," in *15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016, pp. 547–552.

[39] J. Song, M. Sheng, T. Q. Quek, C. Xu, and X. Wang, "Learning-based content caching and sharing for wireless networks," *IEEE Transactions on Communications*, vol. 65, no. 10, pp. 4309–4324, 2017.

[40] A. Sengupta, S. Amuru, R. Tandon, R. M. Buehrer, and T. C. Clancy, "Learning distributed caching strategies in small cell networks," in *Wireless Communications Systems (ISWCS), 2014 11th International Symposium on*. IEEE, 2014, pp. 917–921.

[41] S. S. Tanzil, W. Hoiles, and V. Krishnamurthy, "Adaptive scheme for caching youtube content in a cellular network: Machine learning approach," *IEEE Access*, vol. 5, pp. 5870–5881, 2017.

[42] 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.

[43] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in Neural Information Processing Systems*, 2017, pp. 6382–6393.

[44] J. Foerster, I. A. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," in *Advances in Neural Information Processing Systems*, 2016, pp. 2137–2145.

[45] J. K. Gupta, M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning," in *International Conference on Autonomous Agents and Multiagent Systems*. Springer, 2017, pp. 66–83.

[46] F. Hu, B. Chen, and K. Zhu, "Full spectrum sharing in cognitive radio networks toward 5G: A survey," *IEEE Access*, vol. 6, pp. 15 754–15 776, 2018.

[47] Q. Zhao, L. Tong, A. Swami, and Y. Chen, "Decentralized cognitive MAC for opportunistic spectrum access in ad hoc networks: A POMDP framework," *IEEE Journal on selected areas in communications*, vol. 25, no. 3, 2007.

[48] K. Liu and Q. Zhao, "A restless bandit formulation of opportunistic access: Indexablity and index policy," in *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2008. SECON Workshops' 08. 5th IEEE Annual Communications Society Conference on*. IEEE, 2008, pp. 1–5.

[49] Q. Zhao, B. Krishnamachari, and K. Liu, "On myopic sensing for multi-channel opportunistic access: structure, optimality, and performance," *IEEE Transactions on Wireless Communications*, vol. 7, no. 12, 2008.

[50] S. H. A. Ahmad, M. Liu, T. Javidi, Q. Zhao, and B. Krishnamachari, "Optimality of myopic sensing in multichannel opportunistic access," *IEEE Transactions on Information Theory*, vol. 55, no. 9, pp. 4040–4050, 2009.

[51] Y. Xu, J. Wang, Q. Wu, A. Anpalagan, and Y.-D. Yao, "Opportunistic spectrum access in unknown dynamic environment: A game-theoretic stochastic learning solution," *IEEE transactions on wireless communications*, vol. 11, no. 4, pp. 1380–1391, 2012.

[52] J. Zheng, Y. Cai, N. Lu, Y. Xu, and X. Shen, "Stochastic game-theoretic spectrum access in distributed and dynamic environment," *IEEE transactions on vehicular technology*, vol. 64, no. 10, pp. 4807–4820, 2015.

[53] K. Wang, Q. Liu, Q. Fan, and Q. Ai, "Optimally probing channel in opportunistic spectrum access," *IEEE Communications Letters*, 2017.

[54] 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, p. 529, 2015.

[55] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[56] 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," *arXiv preprint arXiv:1810.07862*, 2018.

[57] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *arXiv preprint arXiv:1803.04311*, 2018.

[58] L. Xiao, Y. Li, C. Dai, H. Dai, and H. V. Poor, "Reinforcement learning-based NOMA power allocation in the presence of smart jamming," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 4, pp. 3377–3389, 2018.

[59] A. Ortiz, H. Al-Shatri, X. Li, T. Weber, and A. Klein, "Reinforcement learning for energy harvesting decode-and-forward two-hop communications," *IEEE Transactions on Green Communications and Networking*, vol. 1, no. 3, pp. 309–319, 2017.

[60] H. Li, H. Gao, T. Lv, and Y. Lu, "Deep q-learning based dynamic resource allocation for self-powered ultra-dense networks," in *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2018, pp. 1–6.

[61] H. Liu, S. Liu, and K. Zheng, "A reinforcement learning-based resource allocation scheme for cloud robotics," *IEEE Access*, vol. 6, pp. 17 215–17 222, 2018.

[62] H. Ye and G. Y. Li, "Deep reinforcement learning for resource allocation in v2v communications," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[63] Y. Wei, F. R. Yu, M. Song, and Z. Han, "User scheduling and resource allocation in hetnets with hybrid energy supply: An actor-critic reinforcement learning approach," *IEEE Transactions on Wireless Communications*, vol. 17, no. 1, pp. 680–692, 2018.

[64] A. T. Nassar and Y. Yilmaz, "Reinforcement-learning-based resource allocation in fog radio access networks for various iot environments," *arXiv preprint arXiv:1806.04582*, 2018.

[65] Y. Yu, T. Wang, and S. C. Liew, "Deep-reinforcement learning multiple access for heterogeneous wireless networks," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–7.

[66] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, "Deep reinforcement learning for dynamic multichannel access in wireless networks," *IEEE Transac-*

tions on Cognitive Communications and Networking, vol. 4, no. 2, pp. 257–265, June 2018.

[67] W. Dai, Y. Gai, and B. Krishnamachari, "Online learning for multi-channel opportunistic access over unknown Markovian channels," in *Sensing, Communication, and Networking (SECON), 2014 Eleventh Annual IEEE International Conference on*. IEEE, 2014, pp. 64–71.

[68] Y. Zhang, Q. Zhang, B. Cao, and P. Chen, "Model free dynamic sensing order selection for imperfect sensing multichannel cognitive radio networks: A Q-learning approach," in *Communication Systems (ICCS), 2014 IEEE International Conference on*. IEEE, 2014, pp. 364–368.

[69] O. Naparstek and K. Cohen, "Deep multi-user reinforcement learning for distributed dynamic spectrum access," *IEEE Transactions on Wireless Communications*, vol. 18, no. 1, pp. 310–323, 2019.

[70] S. Liu, X. Hu, and W. Wang, "Deep reinforcement learning based dynamic channel allocation algorithm in multibeam satellite systems," *IEEE ACCESS*, vol. 6, pp. 15 733–15 742, 2018.

[71] H. Li, "Multiagent Q-learning for aloha-like spectrum access in cognitive radio systems," *EURASIP Journal on Wireless Communications and Networking*, vol. 2010, p. 56, 2010.

[72] M. Bkassiny, S. K. Jayaweera, and K. A. Avery, "Distributed reinforcement learning based MAC protocols for autonomous cognitive secondary users," in *Wireless and Optical Communications Conference (WOCC), 2011 20th Annual*. IEEE, 2011, pp. 1–6.

[73] K.-L. A. Yau, P. Komisarczuk, and D. T. Paul, "Enhancing network performance in distributed cognitive radio networks using single-agent and multi-

agent reinforcement learning," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*. IEEE, 2010, pp. 152–159.

[74] Y. Wang, M. Liu, J. Yang, and G. Gui, "Data-driven deep learning for automatic modulation recognition in cognitive radios," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 4074–4077, April 2019.

[75] 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, June 2018.

[76] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[77] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller, "Adversarial attacks on deep neural networks for time series classification," *arXiv preprint arXiv:1903.07054*, 2019.

[78] V. Subramanian, E. Benetos, N. Xu, S. McDonald, and M. Sandler, "Adversarial attacks in sound event classification," *arXiv preprint arXiv:1907.02477*, 2019.

[79] Y. Zhao, I. Shumailov, H. Cui, X. Gao, R. Mullins, and R. Anderson, "Blackbox attacks on reinforcement learning agents using approximated temporal information," *arXiv preprint arXiv:1909.02918*, 2019.

[80] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," *arXiv preprint arXiv:1702.02284*, 2017.

[81] A. Gleave, M. Dennis, N. Kant, C. Wild, S. Levine, and S. Russell, "Adversarial policies: Attacking deep reinforcement learning," *arXiv preprint arXiv:1905.10615*, 2019.

[82] Y. Shi, T. Erpek, Y. E. Sagduyu, and J. H. Li, "Spectrum data poisoning with adversarial deep learning," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 407–412.

[83] T. Erpek, Y. E. Sagduyu, and Y. Shi, "Deep learning for launching and mitigating wireless jamming attacks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 1, pp. 2–14, 2018.

[84] C. Zhong, Z. Lu, M. C. Gursoy, and S. Velipasalar, "Actor-critic deep reinforcement learning for dynamic multichannel access," in *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2018, pp. 599–603.

[85] A. Bujnowski, J. Ruminski, A. Palinski, and J. Wtrorek, "Enhanced remote control providing medical functionalities," in *Proc. Inter. Conf. Pervasive Comput. Tech Healthc. Workshops*, May 2013, pp. 290–293.

[86] F. Passerini and A. M. Tonello, "Smart grid monitoring using power line modems: Effect of anomalies on signal propagation," *IEEE Access*, vol. 7, pp. 27 302–27 312, 2019.

[87] F. Alotibi and M. Abdelhakim, "Anomaly detection for cooperative adaptive cruise control in autonomous vehicles using statistical learning and kinematic model," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–11, 2020.

[88] A. Kanev, A. Nasteka, C. Bessonova, D. Nevmerzhitsky, A. Silaev, A. Efremov, and K. Nikiforova, "Anomaly detection in wireless sensor network of the

"smart home" system," in *2017 20th Conference of Open Innovations Association (FRUCT)*. IEEE, 2017, pp. 118–124.

[89] Z. Li, J. Xie, H. Zhang, H. Xiang, and Z. Zhang, "Adaptive sensor scheduling and resource allocation in netted collocated MIMO radar system for multi-target tracking," *IEEE Access*, vol. 8, pp. 109 976–109 988, 2020.

[90] Q. Zhao, L. Tong, A. Swami, and Y. Chen, "Decentralized cognitive MAC for opportunistic spectrum access in ad hoc networks: A POMDP framework," *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 3, pp. 589–600, 2007.

[91] S. Rajasegarar, C. Leckie, and M. Palaniswami, "Anomaly detection in wireless sensor networks," *IEEE Wireless Communications*, vol. 15, no. 4, pp. 34–40, 2008.

[92] H. Chernoff, "Sequential design of experiments," *The Annals of Mathematical Statistics*, vol. 30, no. 3, pp. 755–770, 1959.

[93] F. Cecchi and N. Hegde, "Adaptive active hypothesis testing under limited information," in *Advances in Neural Information Processing Systems*, 2017, pp. 4035–4043.

[94] D. Chen, Q. Huang, H. Feng, Q. Zhao, and B. Hu, "Active anomaly detection with switching cost," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 5346–5350.

[95] K. Cohen and Q. Zhao, "Active hypothesis testing for anomaly detection," *IEEE Transactions on Information Theory*, vol. 61, no. 3, pp. 1432–1450, 2015.

[96] B. Huang, K. Cohen, and Q. Zhao, "Active anomaly detection in heterogeneous processes," *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 2284–2301, 2019.

[97] M. R. Leonard and A. M. Zoubir, "Robust sequential detection in distributed sensor networks," *IEEE Transactions on Signal Processing*, vol. 66, no. 21, pp. 5648–5662, 2018.

[98] J. Zhang and I. C. Paschalidis, "Statistical anomaly detection via composite hypothesis testing for markov models," *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 589–602, 2018.

[99] D. Kartik, E. Sabir, U. Mitra, and P. Natarajan, "Policy design for active sequential hypothesis testing using deep learning," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2018, pp. 741–748.

[100] A. Puzanov and K. Cohen, "Deep reinforcement one-shot learning for change point detection," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2018, pp. 1047–1051.

[101] I. Alrashdi, A. Alqazzaz, E. Aloufi, R. Alharthi, M. Zohdy, and H. Ming, "AD-IoT: Anomaly detection of IoT cyberattacks in smart city using machine learning," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0305–0310.

[102] N. Moustafa, K.-K. R. Choo, I. Radwan, and S. Camtepe, "Outlier dirichlet mixture mechanism: Adversarial statistical learning for anomaly detection in the fog," *IEEE Transactions on Information Forensics and Security*, 2019.

[103] C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep actor-critic reinforcement learning for anomaly detection," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[104] T. Wen and R. Keyes, "Time series anomaly detection using convolutional neural networks and transfer learning," *arXiv preprint arXiv:1905.13628*, 2019.

[105] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. V. Chawla, "A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1409–1416.

[106] Y. Sun, W. Yu, Y. Chen, and A. Kadam, "Time series anomaly detection based on GAN," in *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 2019, pp. 375–382.

[107] D. Li, D. Chen, B. Jin, L. Shi, J. Goh, and S.-K. Ng, "MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks," in *Artificial Neural Networks and Machine Learning – ICANN 2019: Text and Time Series*, I. V. Tetko, V. Kůrková, P. Karpov, and F. Theis, Eds. Cham: Springer International Publishing, 2019, pp. 703–716.

[108] A. Gurevich, K. Cohen, and Q. Zhao, "Sequential anomaly detection under a nonlinear system cost," *IEEE Transactions on Signal Processing*, vol. 67, no. 14, pp. 3689–3703, 2019.

[109] Y. Shao, A. Rezaee, S. C. Liew, and V. W. S. Chan, "Significant sampling for shortest path routing: A deep reinforcement learning solution," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–7.

[110] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[111] Q. Yang, T. D. Simão, S. H. Tindemans, and M. T. Spaan, "Wcsac: Worst-case soft actor critic for safety-constrained reinforcement learning," in *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence. AAAI Press, online*, 2021.

[112] A. Vempaty, O. Ozdemir, K. Agrawal, H. Chen, and P. K. Varshney, "Localization in wireless sensor networks: Byzantines and mitigation techniques," *IEEE Transactions on Signal Processing*, vol. 61, no. 6, pp. 1495–1508, 2013.

[113] 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.

[114] M. Ahmed, S. Traverso, P. Giaccone, E. Leonardi, and S. Niccolini, "Analyzing the performance of LRU caches under non-stationary traffic patterns," *arXiv preprint arXiv:1301.4909*, 2013.

[115] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[116] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)," *Relatório técnico, Telecom ParisTech*, pp. 1–6, 2011.

[117] T. Smith and R. Simmons, "Heuristic search value iteration for POMDPs," in *Proceedings of the 20th conference on Uncertainty in artificial intelligence*. AUAI Press, 2004, pp. 520–527.

[118] D. Szer, F. Charpillet, and S. Zilberstein, "Maa*: A heuristic search algorithm for solving decentralized POMDPs," *arXiv preprint arXiv:1207.1359*, 2012.

[119] D. Silver and J. Veness, "Monte-carlo planning in large POMDPs," in *Advances in neural information processing systems*, 2010, pp. 2164–2172.

[120] S. Thrun, "Monte carlo POMDPs," in *Advances in neural information processing systems*, 2000, pp. 1064–1070.

[121] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, pp. 1180–1190, 2005.

[122] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press Cambridge, 1998, vol. 1, no. 1.

[123] K. Liu and Q. Zhao, "Indexability of restless bandit problems and optimality of Whittle index for dynamic multichannel access," *IEEE Transactions on Information Theory*, vol. 56, no. 11, pp. 5547–5567, 2010.

[124] M. Naghshvar, "Active learning and hypothesis testing," Ph.D. dissertation, UC San Diego, 2013.

[125] A. Rezaee and V. W. S. Chan, "Cognitive network management and control with significantly reduced state sensing," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 3, pp. 783–794, 2019.

[126] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[127] G. Joseph, M. C. Gursoy, and P. K. Varshney, "Anomaly detection under controlled sensing using actor-critic reinforcement learning," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2020, pp. 1–5.

Vita

Chen Zhong received her B.S. degree in Information Engineering in 2014 from Beijing Institute of Technology, Beijing, China, and her M.S. degree in Electrical Engineering in 2016 from Stevens Institute of Technology, Hoboken, USA. She finished the Ph.D. degree in the Department of Electrical Engineering and Computer Science at Syracuse University, in 2022. During her Ph.D. study, she has 3 first author journal publications and 8 first author conference publications. Her research interests are in the fields of wireless communications and machine learning.