A Deep Reinforcement Learning-Based Caching Strategy for Internet of Things Networks with Transient Data

Ali Nasehzadeh

A Thesis Submitted to the Faculty of Graduate
Studies in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Graduate Program in Electrical Engineering & Computer Science
York University
Toronto, Ontario

May 2021

© Ali Nasehzadeh, 2021

Abstract

Internet of Things (IoT) has been on a continuous rise in the past few years, and its potentials are now more apparent. Transient data generation and limited energy resources are the major bottlenecks of these networks. Besides, conventional quality of service measurements are still valid requirements that should be met, such as throughput, jitter, error rate, and delay or latency. An efficient caching policy can help meet the standard quality of service requirements while bypassing the specific limitations of IoT networks. Adopting deep reinforcement learning (DRL) algorithms enables us to develop an effective caching scheme without the need for any prior knowledge or contextual information such as popularity distributions or the lifetime of files. In this thesis, we propose DRL-based caching schemes that improve the cache hit rate and energy consumption of IoT networks compared to some well-known conventional caching schemes. We also propose a hierarchical caching architecture that enables parent nodes to receive the requests from multiple edge nodes and make caching decisions independently. The results of comprehensive experiments show that our proposed method outperforms the well-known conventional caching policies and an existing DRL-based method in cache hit and energy consumption rates by considerable margins. In the fifth chapter, we seek machine learning-based caching solutions for the cases in which the file popularity distribution is not static but changing over time. Taking the same system model into consideration, we propose a concept drift detection method based on clustering and cluster similarity measurements. The drift

detection can trigger a process that leads the DRL agent to adapt to the new popularity distribution; we have based this process on transfer learning techniques. Transfer learning can help us leverage the existing knowledge in trained models and significantly speed up the training phase of our machine learning models.

Table of Contents

Abstract..II

List of Figures……………………………………………………………VI

List of Acronyms……………………………………………………….....VIII

Chapter 1. Introduction 1

1.1 Caching in IoT Networks 1

1.2 Structure of the Thesis 3

1.3 Contributions.. 4

Chapter 2. Background Knowledge 5

2.1 Deep Reinforcement Learning 5

2.2 K-Means Clustering 6

2.3 Transfer Learning.. 7

Chapter 3. Literature Review 9

Chapter 4. Deep Reinforcement Learning-based Caching Strategy for IoT Networks 16

4.1 System Model and Problem Formulation 16

4.2 Proposed Methods.. 21

4.2.1 The Centralized Approach 21

4.2.2 The Distributed Approach 25

4.2.3 Proximal Policy Optimization Algorithm (PPO) 29

4.3 Evaluation 33

4.3.1 Evaluation of the Centralized Caching Approach 33

4.3.2 Evaluation of the Distributed Caching Approach.................... 38

4.4 Summary 46

Chapter 5. Transfer Learning-based Caching Strategy for IoT Networks .48

5.1 Problem Statement 48

5.2 Proposed Approach 49

5.2.1 Concept Drift Detection 49

5.2.2 DQN-based Caching Solution with Transfer Learning 52

5.3 Evaluation of the Dynamic Caching Scheme 57

5.4 Summary 61

Chapter 6. Conclusion 62

6.1 Contributions of the Thesis 62

6.2 Future Works 63

References...65

List of Figures

Figure 4.1 System model: a two-layer IoT network 17

Figure 4.2 expiration reward function 27

Figure 4.3 Actor-Critic architecture 30

Figure 4.4 PPO algorithm workflow 32

Figure 4.5 Accumulated reward for the DRL agent and LRU 36

Figure 4.6 Average freshness of the stored files in the memory 36

Figure 4.7 Average memory utilization... 37

Figure 4.8 comparison between energy consumption and cache hit rates..... 37

Figure 4.9 cache hit rate v.s. request rates 41

Figure 4.10 cache hit rate v.s. skewness factors 41

Figure 4.11 Comparison of energy consumption rates for different W's...... 42

Figure 4.12 Comparison of energy consumption rates for different α's 42

Figure 4.13 Hit rate vs files' life-time.. 43

Figure 4.14 Hit rate vs files' popularity 44

Figure 4.15 Comparison of hierarchical architecture (proposed) and flat architecture (edge caching only). The figure shows the hit rate vs request rates ω 46

Figure 4.16 Comparison of hierarchical architecture (proposed) and flat architecture (edge caching only) vs skewness 46

Figure 5.1 Illustration of Jaccard similarity index 51

Figure 5.2 Jaccard similarity index... 51

Figure 5.3 Q-Learning and Deep Q-learning.. 53

Figure 5.4 DQN workflow 54

Figure 5.5 Transfer learning illustration 57

Figure 5.6 The realized reward during the training phase, before and after transfer 60

List of Acronyms

AI……………………………………………..………Artificial Intelligence

BS……………………………………………..………………..Base Station

CDN……………………………….................Content Delivery Network

DQN…………………………………………………...Deep Q-Networks

DRL……………………………………….Deep Reinforcement Learning

LFU…………………………………………….…Least Frequently Used

LRU………………………………………….………Least Recently Used

MDP………………………………………………Markov Decision Process

MRI…………………………………….…..Magnetic Resonance Imaging

NN…………………………………..……………………….Neural Network

PPO………………………………………..Proximal Policy Optimization

RL………………………………………………….Reinforcement Learning

RSU……………………………………………………………Road-Side Unit

SGD…………………………………………….Stochastic Gradient Descent

TL…………………………………………………………...Transfer Learning

TRPO……………………………………Trust Region Policy Optimization

Chapter 1. Introduction

In this chapter, we briefly introduce the unique characteristics of Internet of Things networks; we then elaborate on why a caching strategy can improve an IoT network's performance; we also briefly review some challenges of the current caching method, and clarify our goals and motivations for this thesis work. In section 1.2, we describe the structure of this thesis, and finally, in section 1.3, we present our publications.

1.1 Caching in IoT Networks

The world is flooding with data and data transmissions; simultaneously, the demand for more reliable and faster connection is souring. This concurrency points to either terribly congested networks or the need for more resources and smarter networking schemes. With the emergence of the Internet of Things (IoT), there is an exponential growth in the number of connected devices in the world, which again emphasizes the importance of better networking schemes [1] [2]. Some examples of IoT networks are smart homes, automation in manufacturing, smart cities and smart routing and traffic control. IoT networks have some unique attributions, e.g. most IoT devices are battery-powered and have limited energy to work with.

Moreover, data generated by the IoT sensors are generally valid for a limited duration of time, e.g. temperature readings can be valid for few minutes, or smoke detection sensors' readings might be valid for only a few seconds. This attribute leads to frequent data transmissions, which is challenging for the limited power

source. To help with these problems, we can leverage a caching technique. Caching is a networking technique that enables networking nodes close to end-users to store frequently requested data (in the thesis, we refer to the data as files), thus mitigating the traffic in networks' links and improving the response time. For example, temperature data can be stored in a networking node, and whenever a user asks for the temperature, the node can forward the data itself instead of fetching it from the source, as long as the data is valid.

Edge caching is a new technology that enables edge nodes such as base stations and users' equipment to be a part of the caching scheme and store files. Being closer to the end-users means that the request does not necessarily go all the way up to the source or routers farther away, to fetch the response because one of the edge nodes might already have the desired file in its caching memory and is ready to fulfill the request locally, which in turn will significantly shorten the response time and simultaneously ease the load on the backhaul links. In this thesis, we are enabling edge nodes in the caching process, however we are not considering users' equipment as a part of the caching scheme.

Caching is not a new technique, but it has been continuously evolving, and with the recent advancement in Artificial Intelligence (AI), some machine learning-based caching methods have been proposed. These methods are more efficient and they can also update themselves to better meet the needs of non-stationary environments. In this thesis, we also propose a machine learning-based caching method.

Assuming the file popularity distribution as a known or a stationary one is not a realistic assumption, but most of the works on caching do so in the literature. In reality, however, file popularity may change with time. For example, in an IoT-based smart home network, security sensors might be read with higher frequency during nights than days. This dynamic, non-stationary behaviour is often a challenging attribute when we are working with machine learning algorithms. Generally, machine learning algorithms seek to model the "*true distribution*"; this term itself implies a single and not changing distribution. So, the question that naturally presents itself here is how we can handle this changing behaviour while leveraging a machine learning-based solution.

The challenges mentioned above form our motivations to search for a caching strategy that improves the cache hit rate[1] and energy consumption in an IoT network while also considering the limited lifetime of data. Moreover, we want to find a way to handle the changing file popularity distribution. We propose a deep reinforcement learning and transfer learning based caching strategy without assuming any unrealistic assumptions or prior knowledge about the data distribution or users' behaviours.

1.2 Structure of the Thesis

The rest of this thesis is organized as follows: Chapter 2 presents some background knowledge about the machine learning techniques we are using in this thesis; we briefly introduce deep reinforcement learning, clustering, and transfer

[1] $cache\ hit = \frac{number\ of\ requests\ responded\ locally}{total\ number\ of\ requests}$

learning. Chapter 3 contains a comprehensive literature review on the domain. Chapter 4 presents the formulation of the caching problem in IoT networks and the proposed deep reinforcement learning-based solutions, along with exhaustive performance evaluations. Chapter 5 addresses the caching problem in IoT networks where the file popularity changes with time; a concept drift detection and a transfer learning based caching scheme are presented in this chapter along with their evaluations. Finally, Chapter 6 concludes this thesis.

1.3 Contributions

- Nasehzadeh, A. and P. Wang. A Deep Reinforcement Learning-Based Caching Strategy for Internet of Things. in 2020 IEEE/CIC International Conference on Communications in China (ICCC). 2020. IEEE.
- (Under review) A. Nasehzadeh and P. Wang, "Deep Reinforcement Learning-Based Caching Strategy for IoT Networks with Transient Data" to IEEE Internet of Things Journal.

Chapter 2. Background Knowledge

To address the challenges of the caching problem, we have chosen deep reinforcement learning as the main solver of this problem. In this chapter, we briefly introduce the primary machine learning techniques we use in this thesis.

2.1 Deep Reinforcement Learning

Reinforcement Learning (RL) [3], a relatively more recent paradigm in machine learning, has gained considerable attention in various optimization problems, including the ones imposed in the field of computer networks. It differs from conventional supervised learning methods where the agent is explicitly told what to do or how to classify; because there is no labelled dataset in reinforcement learning. RL also differs from unsupervised learning since the objective is to maximize a reward signal, but there is no such signal in unsupervised learning [3]. In reinforcement learning, an agent is trained by exploring its environment through multiple interactions, evaluating its options, and searching for the most rewarding actions. In an RL workflow, at each time step, the agent would choose an action, the action will affect the environment, and the state of the environment will change in response to that, and then a reward would be generated and returned to the agent. The agent's objective is to figure out how to behave in each state of the environment to maximize the receiving rewards. This mapping of states to actions is called policy; in other words, any RL algorithm's goal is to find the policy that maximizes the accumulated rewards in the long term. Conventional RL methods have shown to be beneficial in some domains, but unfortunately, they have restricted

applicability. One can only apply them to the domains where either essential features can be handcrafted or fully-observed low-dimensional state spaces are available. However, with the advances in deep learning techniques, it turns out that it is possible to successfully train an agent to learn a rewarding policy directly from high dimensional inputs using end-to-end reinforcement learning [4]. In the context of caching problems, many existing caching schemes require some prior knowledge, such as content popularity, that may not be practically available. This constraint may limit their application in practice. On the contrary, DRL-based caching methods do not need any prior knowledge because such information can be learned through interactions with the environment; this attribute is one of the significant advantages of DRL methods.

As great as DRL algorithms can be, they are not able to handle non-stationary environments easily. Changing the file popularity distribution is an example of a non-stationary environment. As a result, we need to use other tools to detect and handle a change in the environment. We use K-Means clustering and transfer learning techniques to overcome these challenges.

2.2 K-Means Clustering

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. Unsupervised algorithms make inferences from datasets using only input vectors without referring to the labels. The objective of the K-means algorithm is to group similar items together. To achieve this objective, K-means looks for a fixed number (K) of clusters in a dataset, where a cluster refers to a collection of data points aggregated together because of certain similarities.

K-Means algorithm takes the value K as input and identifies K number of centroids, and then allocates every data point to the nearest cluster. The ‘means’ in the K-means refers to averaging of the data; that is, finding the centroids.

In this thesis, we use K-means to detect when a change happens in the file popularity distribution. However, to handle the detected changes, we need another tool. We have chosen a transfer learning approach to address this challenge, and in what follows, we briefly introduce the main concepts of transfer learning.

2.3 Transfer Learning

In our daily lives, we face many examples of transfer learning, e.g. learning the German language would make it easier to learn Dutch, or learning to play the piano would be easier if one already knows how to play another instrument. Observing people's ability to transfer their knowledge from one domain to an entirely different one and solve new problems formed the transfer learning paradigm's motivation. One of the broadly accepted definitions of transfer learning (TL) is as follows: transfer learning aims to extract the knowledge from one or more source tasks and apply the knowledge to a target task, where the performance in the target task is of higher importance [5] [6].

One of the main assumptions in most machine learning algorithms is that the training and testing (current and future) data are sampled from the same distribution. While this assumption has helped solve many problems, it does not hold true for many other applications [5] [6]. In such cases, we can benefit from TL techniques. To further illustrate TL's usefulness, assume we have developed and

trained an image classifier for portraits, and now we want to develop a medical magnetic resonance image (MRI) classifier for diagnostic purposes. The problem with the new task is that there are very few samples for the MRI photos, whereas there are millions of portrait images we can use for training. In this case, we can transfer knowledge from the first classifier to a new one and, for example, benefit from the edge detection skills which is already available and fine-tune the rest of the detection skills on the few data samples that we have for the MRI images.

In this thesis, we use transfer learning techniques to make a DRL-based caching algorithm better respond to the changes in the file popularity distribution in an IoT network. Without applying transfer learning techniques, we had to either keep using the previous agent and accept the significantly lower cache hit rate and higher energy consumption, or wait a long time to train an agent from the beginning and miss the chance to successfully respond locally in the meanwhile.

Chapter 3. Literature Review

Numerous caching replacement policies have been proposed previously; some of the well-known methods are Least Frequently Used (LFU) and Least Recently Used (LRU) methods. The basic idea for both caching algorithms is to somehow rank the files in the nodes' caching memories, and when a node has to store a new file that is not already stored in the caching memory, the lowest ranking file will be replaced with the new one. The difference between these two methods is in how they approach the ranking part. As the name suggests, LFU ranks the files based on how frequently they have been used, whereas the LRU ranks them based on how recently they have been used. In recent years with the promising advancements in artificial intelligence and machine learning, researchers have utilized new and smarter algorithms to tackle the caching problem. Reinforcement learning (RL) [3], a machine learning paradigm that has recently gained significant attention and has been successfully deployed to address various optimization problems, is also a good approach to be applied to the caching problem.

The goal of any RL algorithm is to maximize a reward signal in the long run. Such a reward signal is missing from both supervised and unsupervised learning methods [3]. An RL algorithm starts by exploring its environment, and after iterating the observation-interaction cycle numerous times, it gains some insight about which action might lead to the maximum reward in the long run. To put it in other words, after multiple interactions, the algorithm will develop a mapping from states to actions, which we call policy; and the goal of an RL algorithm is to find

the optimal policy that leads to the highest cumulative reward in the long term. RL is not a newly found approach, but until just recently, its algorithms were not entirely practical. One can apply RL methods only to the problems with low-dimensional search spaces or cases where the main features could have been easily handcrafted. However, thanks to the deep learning techniques, it is now possible to address problems with very large search spaces without explicitly defining essential features using deep reinforcement learning algorithms [4].

Some of the existing caching strategies demand unrealistic information or prior knowledge about the network, e.g., files' popularity distribution, making them infeasible solutions. On the contrary, deep reinforcement learning (DRL) based caching methods do not assume prior knowledge and base their learning on numerous rounds of interactions with the environment. This attribute is one of the main advantages of the DRL approach.

Some works such as [7], [8] [9] [10] have leveraged DRL algorithms to find a good caching policy in content delivery networks with huge files. The authors of [7] considered a hierarchical system model consisting of parent and leaf nodes. They developed a cooperative caching scheme where each node's caching decision is affected by the other nodes' decisions. The proposed DRL method is named Hyper Deep Q-Network (HDQN), which requires stacks of deep Q-networks (DQN) to produce a caching decision. More specifically, the output of DQNs forms a cost vector and then this cost vector leads the DRL agent to output a caching decision. The authors of [8] deployed an actor-critic architecture for their DRL algorithm named Wolpertinger, aiming to shrink the action space using a K-nearest

neighbour algorithm. The smaller action space allows the algorithm to have a faster run time while it does not adversely affect the policy's performance. In [8], the authors deployed a DRL algorithm to do caching in a content delivery network, and they have implemented the relationship between edge nodes as both cooperative and competitive. In [9], Wang *et al.* applied an A3C DRL algorithm to address the caching problem in a content-centric network while assuming different priorities for different requests.

DRL methods have also been deployed to tackle the caching problem in mobile networks. The authors of [10] proposed a DRL-based caching strategy for vehicular networks leveraging the multi-view videos from street cameras to proactively cache the required contents and deliver a higher video quality. Multi-view videos are recorded from a single location but in multiple directions of view. The term “proactively” refers to the caching policy in which a networking node can ask for a file and store the file in its caching memory, just because it is anticipating receiving a request for that file in the near future instead of actually receiving the request. The authors of [11] proposed a joint computation offloading, caching, and resource allocation optimization by taking advantage of mobile social networks’ data. Their work's main feature is the definition of a trust index based on the interaction history among users of the network. This index indicates which user can share its resources, such as its memory, to help with the caching. The authors of [12] proposed the use of double duelling deep Q-network to solve the joint problem of networking, caching, and computation offloading in a software-defined virtualized vehicular network. Addressing a very similar problem, the authors of

[13] implemented a DRL strategy to address the joint optimization problem of caching and computation offloading in a vehicular network where both vehicles and roadside units (RSUs) are capable of caching and computing in a mobility-aware cooperative fashion.

While the majority of related works concentrate on the caching problem within the context of mobile cloud networks or CDNs, in recent years exponential growth has brought the IoT-related problem into the spotlight; and there are some works that address the caching problem in IoT networks. However, IoT networks introduce their unique set of needs, which bring new challenges to the caching problem. One feature is the periodic generation of data by IoT devices, e.g., an IoT sensor might broadcast the room temperature every five minutes; this means that the information is valid for only five minutes. Therefore, data has a limited lifetime in this scenario. Thus, data freshness must be considered for caching in these networks. Furthermore, energy consumption is also problematic in IoT networks. Because IoT devices are generally battery-powered and hence, periodically retrieving data from them causes the device to leave its energy-saving mode, generate the data, transmit the data, and eventually go back to the energy-saving state. Obviously, iterating through such a cycle consumes significantly more energy than just resting in the energy-saving state.

The authors of [14] proposed a joint optimization for resource allocation, computation offloading and caching for IoT networks using a DRL-based solution. Nevertheless, there is no mention of the limited lifetime of data or energy efficiency in their work. The authors of [15] brought a new perspective to the caching

problem; they used a DRL algorithm to assign proper caching capacity and transmission rate to the nodes in a content-centric IoT network. In [16] the authors proposed a DRL-based caching mechanism for a content-centric IoT network without considering the limited lifetime of data. In another recent work, the authors of [17] deployed a federated deep reinforcement learning method in order to realize cooperative edge caching. Federated learning trains a global model across multiple decentralized edge devices holding local data samples. In this work, cooperation exists between base stations to share their cached data to better handle the requests. Their method outperforms the conventional methods such as LRU and LFU, and it achieves similar performance to the centralized DRL approach with a low loss [17].

In [18], authors considered the data transiency in IoT networks. They have also deployed an actor-critic DRL algorithm to address the caching problem. To the best of our knowledge, [18] is the only work that simultaneously applies DRL and considers the limited lifetime and data freshness to find a solution to the caching problem. On the other hand, there is no mention of the energy consumption rate or how it is affected by the caching policy in [18]. Moreover, the system model considered in [18] comprises only one networking node (a single router). Even though it is an innovative work, the scalability of such a model is of question. Assume a two-layer network with a parent-child structure, the parent node who observes all users' requests can be an influential part of the caching scheme. To illustrate that, assume a file that is not the most popular one in any location, but it has moderate popularity across all of the regions; in this case, none of the edge nodes see any reason to cache this file, but the parent node who can observe this

general popularity will cache this file and improve the cache hit rate significantly. This crucial role has not been considered in [18] because there is no parent node in the system model. It might be good to test a DRL-based caching method with such a system model against the well-known conventional methods such as LRU and LFU.

Moreover, most of the works mentioned above implicitly assume a static file popularity distribution and do not consider the impact of the dynamics of file popularity distribution on the caching performance. To address the challenges caused by the dynamically changing file popularity distribution, transfer learning (TL) can be adopted. The authors of [5] provided a comprehensive summary of applications of transfer learning in wireless networks. Reviewing related works, there are some papers on the applications of transfer learning to caching problems. However, they mainly focus on content popularity prediction, meaning that the underlying machine learning method is a supervised learning one. In [19], the authors proposed using TL to extract the knowledge from the content-access history of device-to-device (D2D) users and then the learning process for the content popularity matrix can be improved using the extracted knowledge. Their study shows a 22% improvement in the satisfaction rating, which is proportionate to the number of requests that are served immediately over the total requests.

In [20] [21], the application of TL was studied in a heterogeneous network, where a macro base station tries to predict the content popularity while the small base stations (BSs) use a random caching approach. By transferring important knowledge from the user-content interaction (source domain) to the users' request

pattern (target domain), it is possible to minimize the required training time. They have shown that increasing the source domain's data points can be linearly proportionate to the training time decrease.

In a more related publication, the authors of [22] combined DRL and TL to address the video rate allocation mechanism. The authors modelled the problem into a Markov decision process and solved it with deep Q-networks (DQN) algorithm, and then TL techniques helped with fine-tuning the DQN's performance to map the video rate from the source to the target domain. [22] focuses on addressing the video rate allocation problem in mobile networks which is different from our work that focuses on caching in IoT networks.

Chapter 4. Deep Reinforcement Learning-based Caching Strategy for IoT Networks

This chapter tackles the caching problem in IoT networks, assuming there is only one stationary file popularity distribution. In the first section of this chapter, we introduce the system model and problem formulation. In the next section, we present our proposed caching methods, one with a centralized approach and one with a distributed approach. We elaborate on the details of these two methods in sections 4.2.1 and 4.2.2, respectively. The DRL solver for both approaches is the proximal policy optimization algorithm (PPO) which is thoroughly explained in section 4.2.3. The results of these caching schemes are investigated in section 4.3.

4.1 System Model and Problem Formulation

In real-world IoT network architectures, there may exist multiple layers of network nodes. In this thesis, we take a simple model consisting of two layers, which are both capable of caching. The layer closer to the data sources (IoT devices/sensors) is the parent layer, and the layer closer to the end-users is the leaf/edge layer. Each edge node is potentially connected to multiple users, while a parent node (as an aggregator) can be connected to many data sources. Figure 4.1 depicts such a configuration. In such a network, when caching technique is not being used, users send their request for a particular type of data (in this thesis, we

refer to each requested data as a file) to their local leaf node, the leaf node will pass the request up to the parent, and the parent will fetch the file from the corresponding data source. However, if the nodes are cache-enabled, the edge node will check its cache memory for the requested file. If the requested file is available in its memory, instead of passing the request to the upper layer, the edge node will respond to the request itself. Otherwise, the parent node will receive the request, the parent now must check its cache for the requested file, and the file will be fetched from the original data source only when neither the edge nor the parent has the file in their cache. This approach helps reduce the load on the backhaul links. Moreover, it is considerably faster to respond to the requests locally rather than fetching the data from the sources. Besides, using the cache memory instead of fetching the data from IoT sensors, we are consuming less energy because producing, transmitting data, and alternating between different power states consumes much energy.

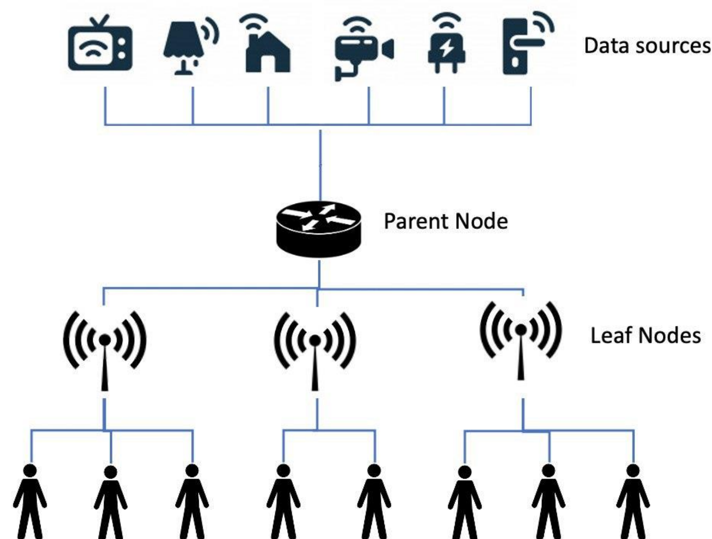

Figure 4.1) System model: a two-layer IoT network

Without loss of generality, we assume N different files in the system, and each data source node can produce only one of the N files. When a source transmits a file, the file's unique id, generation time, and lifetime are sent along with it. Lifetime is the time duration in which that content stays valid after being generated. These lifetimes are different from file to file; for example, the room temperature data might be valid for 5 minutes, while the smoke detector's data might be valid for only 20 seconds due to its more vital nature. The generation time and the lifetime are then used to calculate the freshness of a file, and if the file exceeds its expiration point (i.e., generation time plus the lifetime), it should be removed from the cache memory.

It is needless to say that each node has a limited caching capacity, and as a result, it should avoid storing identical contents in its memory. Note that each edge node (e.g., base station) belongs to a different location, and files can have different popularity from one point to another. Popularity is defined as the frequency that a file is being requested for, and the distribution of this frequency is unknown to the whole system.

Let us now formulate the caching problem as a Markov decision process. Markov decision process (MDP) problems are represented in the forms of a tuple as $\{s, a, p(s' \mid s, a), r\}$, where s denotes the current state, a denotes to action, r is the reward resulted from taking the action a in the state s, and $p(s'|s, a)$ is the transition function which outputs the probability of the following state s' given the current state s and action a. While formulating the caching problem into this framework, the state can refer to the status of the caching nodes in a network, or

the latest user's request, or a combination of all the factors that influence the caching decisions (we will define the details of the state later in this chapter). Actions are the decisions that a node can make, e.g., whether to cache or not or which file should be replaced with the new content. A reward is a scalar value calculated instantly after taking each action and is a measure of the achieved value as the result of the taken action(s). A caching policy $\pi(a|s)$ is also defined, which outputs the probability of choosing action a given the state s. Since actions can affect the reward in later time steps, we should consider the long-term reward. For this reason, we first define the cumulative reward G_n as in (4.1), where n is the time step, r_t is the received reward in time step t, γ is the discount factor, and T is the final time step.

$$G_n = \sum_{t=0}^{T} \gamma^t \, r_{t+n} \qquad (4.1)$$

Discount factor γ is a value in $[0, 1]$; smaller values of γ denote the higher importance of the near-term rewards over long-term ones. The ultimate goal here is to find the optimal policy π^* so that:

$$\pi^* = argmax_{\pi} \;\; \mathbb{E}[G_n| \, \pi] \qquad (4.2)$$

where $\mathbb{E}[G_n \,|\pi]$ denotes the expected value of the random variable G_n given that the DRL agent follows policy π, and n is any time step. To find the optimal policy, we define the value function $V^{\pi}(s)$ and the action-value function $Q^{\pi}(s, a)$ as in (4.3) and (4.4), respectively. $V^{\pi}(s)$ denotes the expected cumulative reward

when starting from the state $\boldsymbol{s}$ and following the policy π after that. The action-value function $Q^{\pi}(s,a)$, also referred to as Q-function, denotes the expected cumulative reward starting from the state s, taking the action a, and after that following the policy π. Using Bellman equations [3], we can write these two functions as in (4.3) and (4.4), where $p(s'|s,a)$ denotes the transition probability to the next state s' given the current state s and the action a, r is the immediate reward which is received after taking the action a, γ is the same discount factor introduced in (4.1).

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} p(s'|s,a)[r + \gamma V^{\pi}(s')] \tag{4.3}$$

$$Q^{\pi}(s,a) = \sum_{s'} p(s'|s,a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a') \right] \tag{4.4}$$

In case one of these two functions can be accurately obtained for all states, it is easy to achieve the optimal policy π^{*}. All we need to do is to choose the action which results in a new state s' so that the $V(s')$ plus the immediate reward is maximized or equivalently choose the action which maximizes the $Q^{\pi}(s,a)$. Using value and Q functions to find the optimal policy is known as the value-based approach to reinforcement learning. The challenge with this approach is to efficiently and accurately calculate the value function or Q function, especially when the state and action spaces are large. Moreover, when the transition probability function is unknown (as in our case), it incurs more difficult or even impossible to calculate the value function or Q function correctly. It is when deep neural networks prove their worth again; by leveraging neural networks in RL

workflow, we would be able to efficiently approximate these functions even for the problems with huge search spaces. It is why we have chosen "deep" reinforcement learning algorithms, but we get to the details of that later in this chapter. Now that we have formulated the caching problem let us present our proposed caching methods.

4.2 Proposed Methods

We are presenting two caching approaches; in the centralized approach, the parent node would act as the oracle and makes the caching decisions for all of the nodes, and in the distributed approach, each node makes its own caching decisions. The difference in system models causes different formulation and reward functions, which we present in detail.

4.2.1 The Centralized Approach

We take the network illustrated in Figure 4.1 as our sample network. Assume that an edge node has received a request for the file f, and this file is not available in the edge node or the parent's cache, and thus, the file should be fetched from the source. In this case, the parent node is the first one to have the file, and it is going to make a centralized caching decision for the requested file. Every node of the network can potentially store this file. Assuming there are M nodes including one parent node and M-1 edge nodes in the network, the action space is defined as $\{a_1, a_2, \ldots, a_M\}$ where $a_i \in \{0,1\}$. If $a_i = 1$, it means that the ith node of the network is supposed to cache the file at hand, and otherwise, the file is not going to be stored. When a caching node is assigned to store a file, it might have enough free space in its memory, and in that case, there is no problem. However, it is also

possible that the memory is full; in this case, the requested file should replace an existing file in the cache. In the latter scenario, we will choose a file to replace following our proposed caching strategy. We would like to clarify that we do not consider using a proactive caching strategy (i.e., pre-caching content before it is requested) for our considered system model. Despite the benefits that such an approach might have in a CDN where the contents are typically static and without a limited lifetime, in our case in which even some popular items may have a very short lifetime, proactive caching might even put an extra load on the backhaul links.

Let us now discuss the reward functions that we consider when implementing the DRL algorithm. As we mentioned earlier about the data freshness, in IoT networks, files have a limited lifetime, and an expired file cannot be returned to the users as a response to their request. Consequently, repeatedly caching a file with a short lifetime might not be an efficient decision. As a result, we consider the freshness of each file in the cache memory as a part of the reward function.

Mathematical definition for freshness is presented in (4.5) where $t_{current}$ and $t_{generation}$ denote the current time and the time instant that the file was generated, respectively. The valid range for the freshness is [0,1), and a smaller value indicates a fresher file. For values greater than or equal to 1, we consider the content as expired and thus invalid.

$$r_1 = \frac{t_{current} - t_{generation}}{life\ time} \qquad (4.5)$$

Increasing the cache hit rate is one of the main objectives of any good caching policy. Cache hit refers to the situation in which a request is addressed by the cache and not by the original data source. In IoT networks, the cache hit rate significantly affects the networks' average response time and energy consumption. Average response time, also known as delay, is the time-span users have to wait before getting their requested files. This value is directly proportional to the distance a request should traverse up towards the data sources in order to fetch the response. The best-case scenario is when a request is addressed locally by an edge node because, in this way, the minimum distance should be paved, and the delay is minimum. The worst delay happens when the file should be fetched from the original source. As we previously explained, the cache hit rate is inversely proportional to the number of times IoT sensors are forced to change their power state, produce and transmit data. The higher the cache hit rate, the lower the energy consumption of IoT devices. The following reward encourages a higher cache hit rate (thus lower delay and lower energy consumption).

$$r_2 = \begin{cases} MAX_{reward} & \text{If the edge responds} \\ MID_{reward} & \text{If the parent responds} \\ MIN_{reward} & \text{otherwise} \end{cases} \quad (4.6)$$

By returning a maximum reward when the edge node responds to a request, we are encouraging the minimum delay, and by returning the minimum reward (which can have a negative value), we are rebuking the high energy consumption and the maximum delay. The medium reward means that we are rewarding the energy-saving, but the delay is not rewarded.

Having lots of free space in the cache memory is not very beneficial because having more valid (fresh) files in the cache translates into higher chances of cache hits. Thus, we define a utilization reward as follows:

$$r_3 = \frac{capacity - used\ space}{capacity} \tag{4.7}$$

This reward is equal to the ratio of non-utilized memory to the total memory space. The total memory space is the aggregated memory capacity of all nodes, and the used memory space is the accumulated used memory space of all nodes. Keep in mind that we can assign positive or negative coefficients to these rewards to convey the concept of reward or punishment, respectively. We have chosen the following form for the final reward function:

$$\begin{gathered} reward_t = \eta\, r_2 - \lambda\, r_1 - \beta\, r_3 \\ \text{where } \lambda, \beta, \eta > 0 \end{gathered} \tag{4.8}$$

The state-space (also known as observation space) is another pillar of reinforcement learning. An agent observes the environment, which is a sample of the observation space, and based on its experiences from the previous interactions with the environment, the agent outputs a new action. In order to form our observations, we first define the memory status variable of each caching node. This variable is a two-row matrix where the first row is the contents ID and the second row is their corresponding freshness value:

$$memStatus = \begin{bmatrix} id_1 & \dots & id_K \\ fresh_1 & \dots & fresh_K \end{bmatrix} \tag{4.9}$$

This way, the RL agent can observe whether the requested file is available in the cache memories or not, the freshness of stored items is also available, so the agent might decide to replace a file that is going to expire in the near future with the requested file.

This concludes the formulating caching problem with a centralized approach. Now, we have to solve this problem; we have chosen the same DRL solver for both centralized and the distributed approach. And to avoid repeating ourselves, we explain the details of the PPO algorithm in section 4.2.3 in parallel to the two problem formulations. Before that, let us present the formulation for the distributed approach.

4.2.2 The Distributed Approach

The system model is still as depicted in Figure 4.1. But we should clarify that in contrast to the centralized approach where the parent node makes the caching decision for all of the nodes, in the distributed approach, each of the parents or edge nodes are capable of making their own caching decision since they are leveraging the same DRL agent with the same reward function.

Recalling the data flow in a cache-enabled network from section 4.2.1, we present the reward function that we deploy in this approach. As we mentioned earlier, in IoT networks data may have limited life-times, which raises some concern about data freshness. We cannot use expired data to respond to users' request, and actually we should remove any expired file from cache memory. It is

readily evident that repeatedly caching a file with a limited life-time cannot be very beneficial, even though it might be fairly popular. Moreover, storing files with long life-times and very poor popularity rank is not efficient either since it is very possible that the file will be expired without receiving any request during its life-time. As an example, for the importance of freshness measure, a DRL agent might deduce that since a file is reaching its expiration it can be a beneficial decision to replace it with the new and coming data. But we should also note that freshness does not offer any intrinsic value; files are valid before expiration and a fresher version of the same file does not offer any advantage. To reflect the above points in our work, we keep the freshness of cached files in the observation, but we do not include it in the reward function. However, for each file in the cache memory that expires without being used at least once, we give a negative reward (punishment) to the agent for storing that file. We keep track of the number of times a cached file has been *hit*; thus, we can calculate the reward function defined in (4.10) for each file in the cache memory after it expires. C_1 is the constant we choose as the punishment coefficient, and k_i is the total number of hits for the ith file in the cache memory.

$$r_{expire}^{i} = [sign(k_i - 0.5) - 1] \times C_1 \qquad (4.10)$$

Note that the total number of hits is an integer, thus by using (4.10) we are giving a negative reward only if a file has a total number of hits equal to zero. Figure 4.2 illustrates this function more clearly.

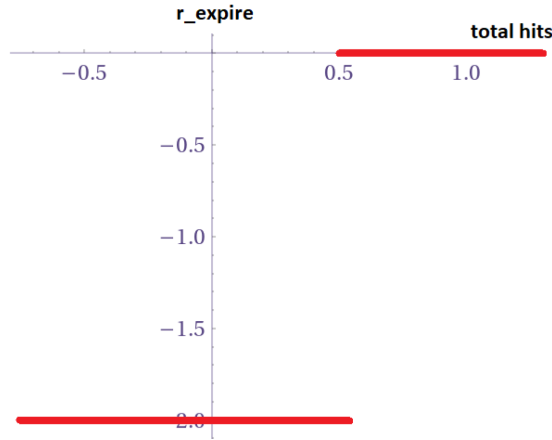

Figure 4.2) Expiration reward function

With the same reasoning from the centralized approach, we want to increase the cache hit rate of the system to avoid communications in the backhaul links and consequently decrease the energy consumption in the IoT devices and improve the response time of the network. So, in order to encourage higher cache hit rates we use the following reward functions:

$$r_{hit} = \frac{K \times C_2}{\tau} \tag{4.11}$$

$$K = \sum_{t=0}^{\tau} \sum_{i=1}^{M} k_i \tag{4.12}$$

where k_i is the total number of hits for the ith file in the cache memory, M is the capacity of files in the memory, τ is the current time step, and K is the sum of all the cache hits up until the current time step. C_2 is a constant hyperparameter which we choose arbitrarily. The final reward function is as follows:

$$r = r_{hit} + \sum_{i=1}^{M} r_{expire}^{i} \tag{4.13}$$

As a part of state-space (also known as observation space) we use the memory status of the caching entities, and a three-row matrix represents the memory status. These three rows contain the content IDs, freshness values as calculated in (4.15), and the total cache hits for each file k_i as defined in (4.12):

$$memStatus = \begin{bmatrix} id_1 & \dots & id_M \\ fresh_1 & \dots & fresh_M \\ k_1 & \dots & k_M \end{bmatrix} \qquad (4.14)$$

$$freshness = \frac{t_{current} - t_{generate}}{t_{age}} \qquad (4.15)$$

The subscript M denotes the maximum number of files we can store in the memory. If there is no content stored in a memory slot, its ID, freshness and k_i values are all set to zero. The requested file's ID and its lifetime account for the other part of the observation. We put these two values together in the request variable as:

$$request = [id, \quad lifetime] \qquad (4.16)$$

This way, the RL agent can observe whether the requested file is available in the cache memories or not. The freshness of stored items is also available, so the agent might decide to replace a file that is going to expire in the near future with the requested file.

This concludes the formulation of the distributed approach. Now, it is time to study the DRL algorithm that solves these two problems. In the following section, we present the details of the proximal policy optimization algorithm (PPO), which we have deployed to tackle the caching problem.

4.2.3 Proximal Policy Optimization Algorithm (PPO)

Proximal Policy Optimization algorithm [23] is a more advanced DRL algorithm that we deploy to address the caching problem when handling a stationary file popularity distribution. Recalling section 4.1, we deploy neural networks to overcome the challenges of approximating the value function or the action-value function in a large search space. In addition to that, with neural networks, it is also possible to parametrize the policy $\pi_\theta(a|s)$, which is also denoted by π_θ, where θ is the set of parameters. Then we can directly tune the parameters of the policy function in search of the optimal one based on the gradient of some performance measure with respect to the policy parameters θ. This approach is known as the policy gradient method, and it has some advantages, such as good convergence and better handling of stochastic policies [3]. Also, there are hybrid versions of RL solutions that take advantage of both value function and policy search, which are generally known as *actor-critic* methods.

We have implemented an actor-critic version of the PPO. This algorithm is developed based on the Trust Region Policy Optimization (TRPO) algorithm [24] and aims to inherit the data efficiency and reliability of the TRPO while avoiding its second-order optimizations. In the PPO algorithm, similar to any other actor-critic method, we simultaneously approximate the policy and value function using two separate neural networks named actor and critic, respectively, as shown in Figure 4.3. PPO deploys sampling from the interactions with the environment, and it also optimizes a clipped Surrogate objective function (will explain in detail later)

using stochastic gradient ascent. PPO is very sample efficient and is relatively easy to implement. PPO algorithm can be implemented in a multi-agent fashion as well.

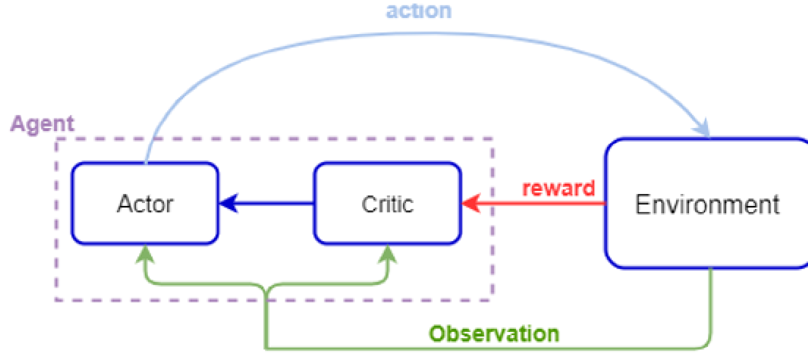

Figure 4.3) Actor-Critic architecture

There are two neural networks (NN) in the architecture of an actor-critic solution. The NN which approximates the value function denoted by $V^{\pi_\theta}(s;\ \theta_v)$ is called critic since its output is given to the actor-network as feedback on the previous action. Then the other network takes in a new observation along with the feedback from the critic to output a new action; this is why this NN is called the actor. In order to update the actor network's parameters denoted by θ, we use the clipped Surrogate objective function as in (4.17).

$$\mathcal{L}^{CLIP}(\theta) = \mathbb{E}_n[min\{b_n(\theta)A_n\,, clip(\,b_n(\theta), 1-\epsilon, 1+\epsilon)A_n\}] \qquad (4.17)$$

$$b_n(\theta) = \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)} \qquad (4.18)$$

The $clip(\cdot)$ function takes in the probability ratio $b_n(\theta)$ defined in (4.18) but clips its value to be no more than $1+\epsilon$ and no less than $1-\epsilon$, where ϵ is a

hyperparameter. θ_{old} in (4.18) is the vector of policy parameters before the update, and A_n is the advantage function which can be calculated as (4.19).

$$A_n = r_n + \gamma V^{\pi_\theta}(s_{n+1}) - V^{\pi_\theta}(s) \qquad (4.19)$$

Finally, to update the parameters θ, we use the following rule:

$$\theta \leftarrow \underset{\theta}{argmax}\, \mathcal{L}^{CLIP}(\theta) \qquad (4.20)$$

The $argmax$ is usually calculated by the stochastic gradient ascent [23].

Algorithm 1 gives a detail description of the PPO workflow. In order to update the critic network, we first need the cumulative rewards G_n as defined in (4.1). Then as in step 6, we apply stochastic gradient descent (SGD) on the mean squared error and update the critic network's parameters θ. Putting Algorithm 1 and Figure 4.4 together should give us a clear image on how the PPO performs.

Algorithm 1: PPO Algorithm, Actor-Critic Style

Input: initial actor's and critic's NN parameters (θ and θ_v)

1: **for** $iteration = 1, 2, ...$ **do**

2: Collect set of trajectories $\mathcal{D} = \{\tau_j\}$ by running the policy π_θ in the environment for T times

3: Estimate advantage values $\hat{A}_1, ..., \hat{A}_T$

4: Receive the rewards and compute $\hat{G}_1, ..., \hat{G}_T$

5: Optimize surrogate loss wrt θ, typically using stochastic gradient ascent with Adam:

$$\theta \leftarrow argmax_\theta \ \mathcal{L}^{clip}(\theta)$$

6: Fit value function by regression on Mean-Squared Error (MSE) using SGD:

$$\theta_v \leftarrow argmin_{\theta_v} \ \mathcal{L}^{VF}$$

where

$$\mathcal{L}^{VF} = \frac{1}{|\mathcal{D}|T} \sum_{\tau \in \mathcal{D}} \sum_{n=0}^{T} \left(V_{\theta_v}(s_n) - \hat{G}_n \right)^2$$

7: **end for**

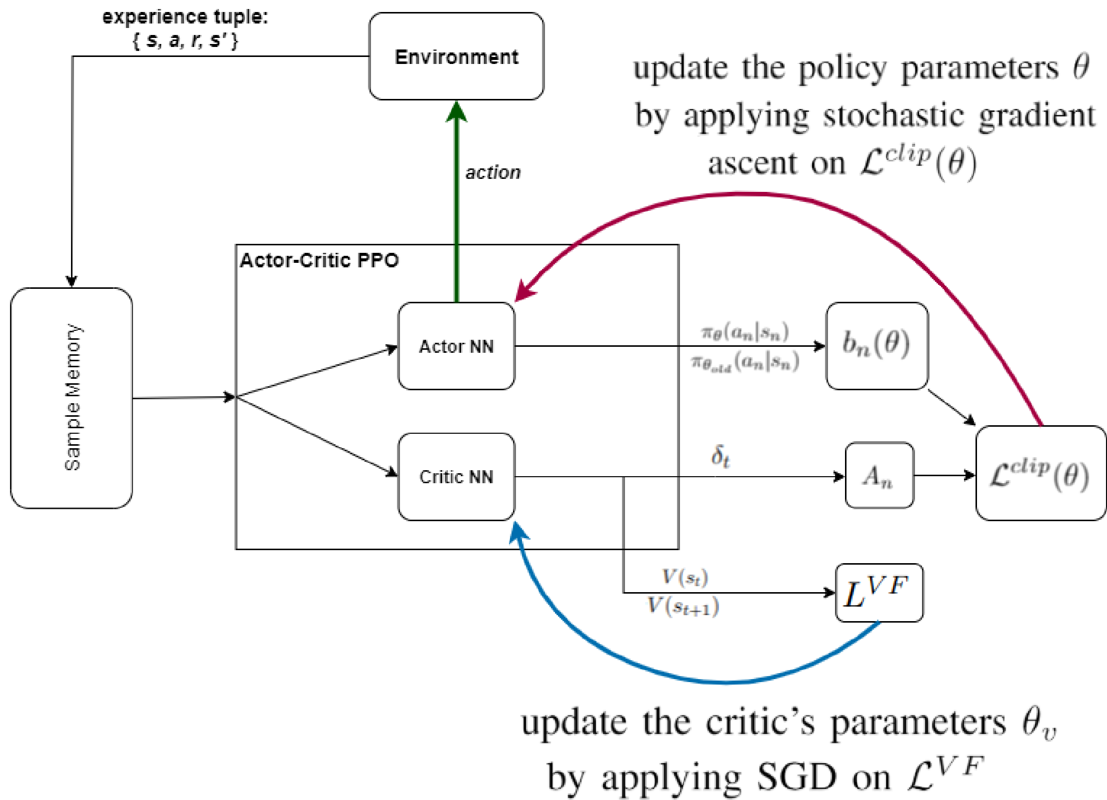

Figure 4.4) PPO algorithm workflow

4.3 Evaluation

The results of employing the PPO algorithm to solve the caching problems defined in section 4.2 are presented in this section. We start with the results of the centralized approach and then continue to the distributed one.

4.3.1 Evaluation of the Centralized Caching Approach

In order to conduct our experiment with the centralized approach, we have used Python version 3.7, TensorFlow 1.14.0, Keras, OpenAI's Gym, and Stable Baselines [25] libraries. Stable Baselines is a library of reinforcement learning algorithm implementations based on OpenAI Baselines. In this experiment, there are one parent node, two edge nodes, and forty IoT devices that produce the data. After users send their requests, their corresponding base station (edge node) checks its cache memory for the requested file and responds to users if the requested content is available. Otherwise, the request is sent to the parent node, and a similar process is executed in the parent node as well; if the parent node could not respond to the request, the file will be fetched from the IoT device. Files have a lifetime in the range of 2 to 14 time-steps. The popularity of files is modeled by Zipf distribution, which means that a file f with a popularity-rank of j is going to be requested with a probability of:

$$p(j;\alpha,F) = \frac{j^{-\alpha}}{\sum_{j=1}^{F} j^{-\alpha}} \tag{4.21}$$

where F is the total number of files in the network, and α is the characterizing parameter to the distribution, which we call the skewness factor in this thesis. As

we assign greater values to the α, the request frequency gap between the most popular and least popular files will grow larger. Note that this distribution is unknown to the agent; it is just used to generate the simulation's file requests.

In the reward function λ, β and η are set to 1 and the static values of the parameters related to r_2 are as follows: $MAX_{reward} = 2$, $MID_{reward} = 1$ and $MIN_{reward} = -5$. These static values do not cause any loss of generality in the evaluations of the proposed method. As for the DRL agents' hyperparameters, the learning rate is initialized from 0.001, and it is scheduled to decrease from that starting point linearly, the discount factor γ is equal to 0.98, and the agent will experience 14 time-steps before performing an update. The actor and critic NNs have two layers with 64 neurons and use $tanh$ activation functions. The hyperparameter values mentioned here have been tuned by trial and error. However, one can use some methods such as Bayesian optimization to search for the optimum hyperparameter values.

The metrics that we used to evaluate our proposed method's performance are the cumulative reward, the average freshness of cached files, the cache hit rate, the energy consumption of IoT devices, and the average utilization of memory space. We are comparing our method with the least recently used (LRU) method. In LRU, the stored files are always ranked based on how recently they have been used, and when a new file should replace one of the files in the memory, it is the least recently used file that will be deleted from the memory.

The cumulative rewards of the two methods are depicted in Figure 4.5. One can quickly notice the major difference in the performance regarding the

cumulative reward; after 1000 requests, LRU results in -1791 points, whereas DRL reaches to -834 points. In Figure 4.6 we can observe the average freshness of the cached files. The DRL method yields an average of about 0.5 throughout the whole simulation, whereas the LRU method has an average freshness of about 0.4. Please note that the smaller value of freshness indicates fresher files. Please also note that we do not have the objective to minimize this freshness value because as long as a file is not expired, it is as good as new, and there is no particular benefit in using a fresher one. The reason that DRL results in higher values for freshness is that there are popular files with a very short lifetime of 2 timesteps; since they are popular, the LRU method will store them, but since they have a lifetime of 2, their freshness cannot even exceed 0.5. On the other hand, there are some popular files with a lifetime of more than 10 timesteps, and thus they can achieve freshness values very close to 1. The DRL agent has learned that a greater reward can be gained by storing the files with longer lifetimes and preferred to pay the small punishment due to higher freshness values. Figure 4.7 shows the average memory utilization over all nodes of the network. The reason that utilization is not always equal to one is that the expired files will be automatically deleted from the cache memory. Since the DRL agent tends to store files with longer lifetimes (as depicted by Figure 4.6, Figure 4.7), fewer expirations happen in the DRL case, and thus the average memory utilization is higher. Figure 4.8 depicts the differences in the cache hit rates and energy consumption in the two methods. It can be seen that the proposed method significantly improves both of these metrics. In our simulation, the DRL method helps to save some energy; relatively speaking, there is a 34.2% improvement over the energy consumption of the LRU method, which is a

significant difference, especially in the battery-powered ecosystem. Please note that the energy consumption is normalized by dividing the absolute consumed values by the energy consumed in the LRU method. This result is consistent with what we have expected as the caching hit rate is inversely proportional to energy consumption.

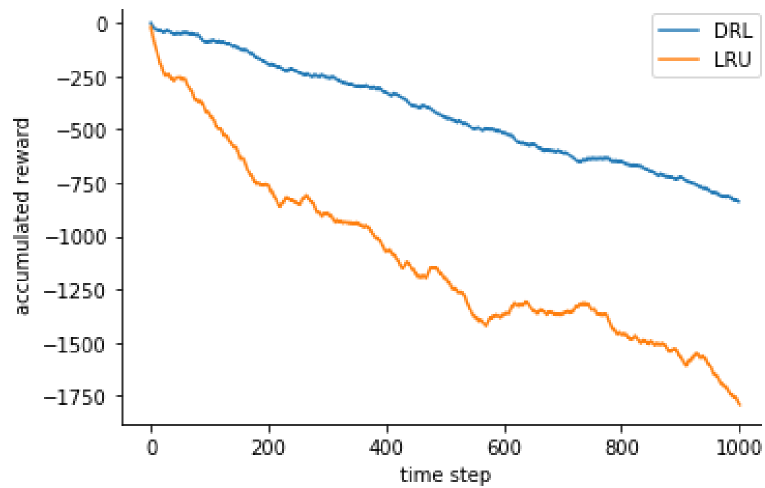

Figure 4.5) Accumulated reward for the DRL agent and LRU

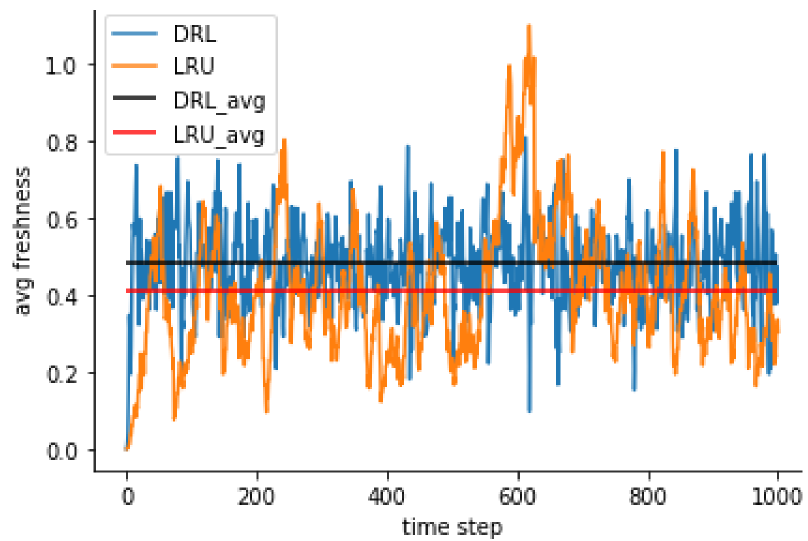

Figure 4.6) Average freshness of the stored files in the memory

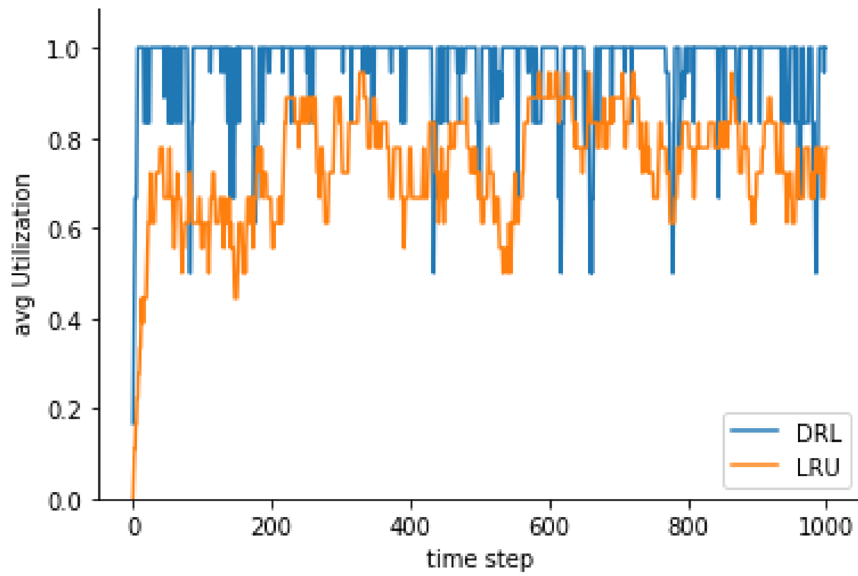

Figure 4.7) Average memory utilization

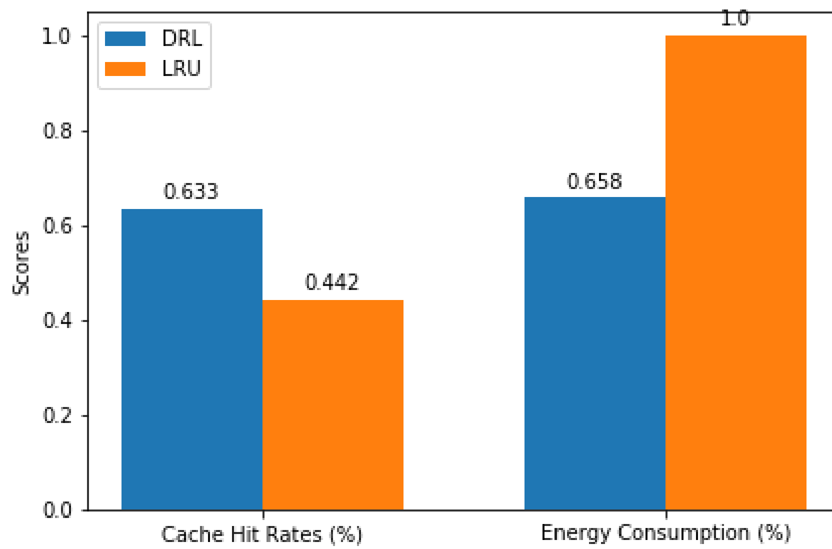

Figure 4.8) Comparison between energy consumption and cache hit rates

4.3.2 Evaluation of the Distributed Caching Approach

In order to conduct our experiment with the distributed approach, we have used Python version 3.7, NumPy [26], TensorFlow 1.14.0 [27], Keras, OpenAI's Gym [28], and Stable Baselines [25] libraries. Stable Baselines is a library of reinforcement learning algorithm implementations based on OpenAI Baselines. We ran the simulation on the workstation with an Intel Core i7-9700 CPU and 16G RAM. The architecture is similar to the one depicted in Figure 4.1; there is one parent node, two edge nodes, and one hundred IoT devices producing the data. Each IoT device only produces one type of file with a unique ID and a lifetime, which is sampled randomly from a uniform distribution with a minimum of 2 and a maximum of 14 time-steps. The popularity distribution follows Zipf's law as explained in (4.21).

We have done our experiments with 24 different settings, resulting from four different request rate values w and six different values of popularity skewness factor α. The request rate of users w varies from 0.5 to 2.0 requests per time step, and the popularity distributions have a varying α between 0.7 to 1.2. Please note that these distributions are unknown to the DRL agent; they are used only for generating requests in the simulations. We assume that each request can be fulfilled before its corresponding user leaves the network. There is no assumption for the users' arrival model since the DRL agent does not use any information except the ones received from the interactions with its environment to make caching decisions.

The architecture of the DRL algorithm is depicted in Figure 4.3 and Figure 4.4. The learning rate is set to 0.001, the discount factor γ is equal to 0.99, and the

agent will experience 16 steps before performing an update, and the number of training mini-batches per update is set to 4. The actor and critic NNs are feedforward neural networks, with two hidden layers and 64 neurons in each layer, and the activation function in use is $tanh$.

We have evaluated our proposed method based on the cache hit rate and energy consumption. We have also monitored the average freshness and popularity of cached files. We compare our method with two of the best-known conventional caching strategies: the least recently used (LRU) and least frequently used (LFU) algorithms. We have also implemented an existing DRL based caching method [18] with a different reward function from ours to observe the effects of our proposed reward function. Now, let us briefly review how each of these methods works:

1) LRU: In LRU, the stored files are always ranked based on how recently they have been used, and when a new file should replace one of the files in the memory, it is the least recently used file that will be deleted from the memory.
2) LFU: Similar to LRU, cached items are ranked, but the ranking criterion is the frequency of their requests. When a new file is meant to be cached, and the memory is full, the new file will replace the file with the least frequency of request.
3) DRL: In our work, we keep track of the number of hits of each file and use this information to design a reward function to encourage a high hit rate. On the other hand, related works such as [18] directly consider the freshness of files in their reward function. To investigate the performance difference

owing to the use of different reward functions, we have implemented the DRL approach in [18] for comparison. We refer to this approach as '**DRL**' in our results.

Figure 4.9 and Figure 4.10 show the cache hit rate versus varying request rate and popularity skewness factor, respectively. It is a clear trend that by increasing those two factors, the cache hit rate will increase as well. Higher request rate gives more chances to the cached files to be requested, and higher skewness factor makes the popularity gap between files even larger and it is easier for the agents to learn the pattern. We can see that our proposed method vastly outperforms the well-known LRU and LFU methods, and it does a better job than the existing DRL method with a different reward function. The proposed method starts with about 38% cache hit, and it goes up to more than 52%. The other DRL approach follows the proposed one with about 1.3% gap, whereas the conventional methods LFU, which outperforms LRU, shows a cache hit range of 16% to 34% for different request rates. The cache hit rate versus the skewness factor has a similar outcome for different approaches.

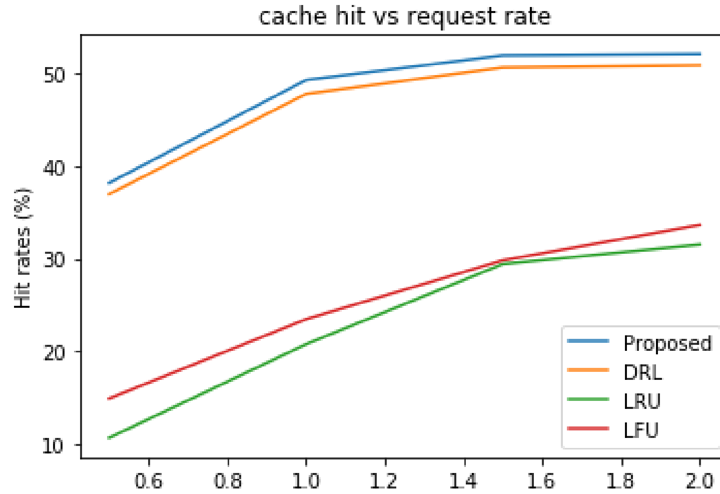

Figure 4.9) Cache hit rate v.s. request rates

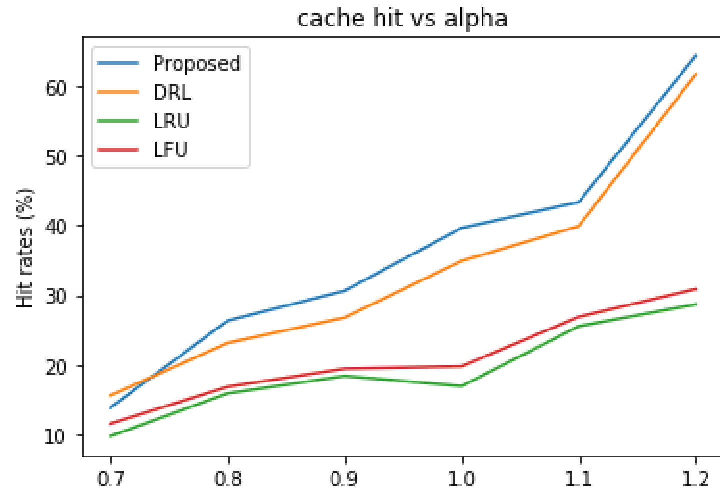

Figure 4.10) Cache hit rate v.s. skewness factors

We have also simulated the energy consumption of the IoT devices for each caching scheme. Figure 4.11 and Figure 4.12 depict the simulation results with varying request rate and popularity skewness factor, respectively. We have normalized the energy consumption rates by our proposed method; values greater than 1 indicate that the energy consumption is higher than our proposed method for that specific configuration, and values less than one denote less energy consumption. We observe that our method uses the least energy for IoT devices

among its counterparts. This result coincides with our expectation that the energy consumption is inversely proportional to the cache hit rate. The higher the cache hit rate, the lower the energy consumption.

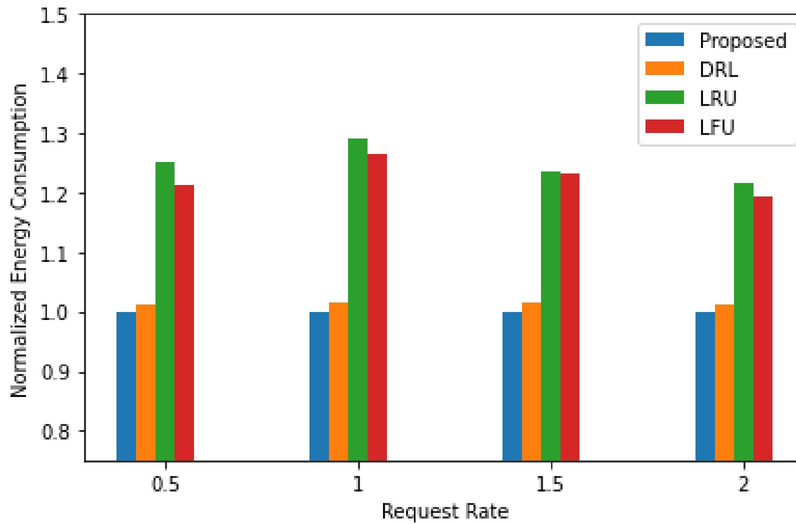

Figure 4.11) Comparison of energy consumption rates for different W's

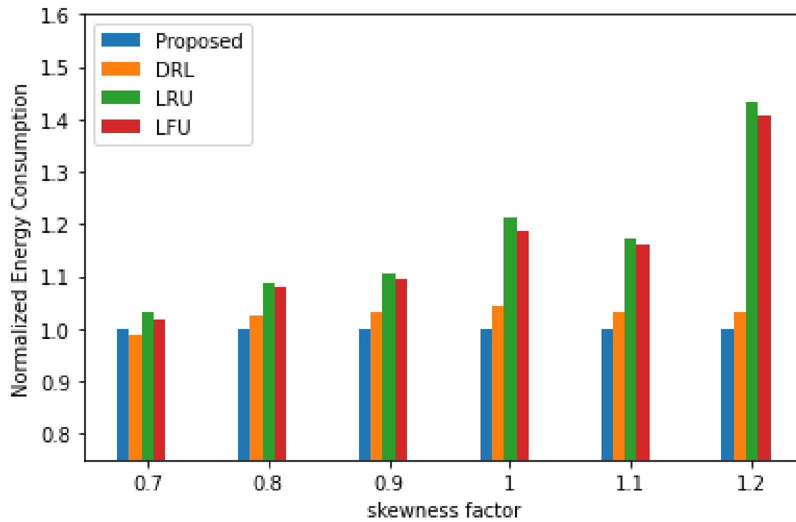

Figure 4.12) Comparison of energy consumption rates for different α's

In order to better understand what makes the proposed algorithm to have a better cache hit rate, we took a look at the requests which have been responded to

by a caching node. We categorize files based on their popularity and lifetime and then keep track of the requested files which have been hit in the cache memories. Figure 4.15 and Figure 4.16 show these results of the average hit rates over all requests and over 24 different settings of our experiments (the 24 settings vary in request rate and the popularity skewness factor as shown in Table 4.1). We can see that the DRL-based methods tend to store popular files, and also, they favor files with a longer life-times, whereas conventional methods (i.e., LRU and LFU) might even favor files with shorter lifetimes. This shows that the DRL agent has learned that storing popular files with a longer lifetime is more beneficial, which is an entirely reasonable argument.

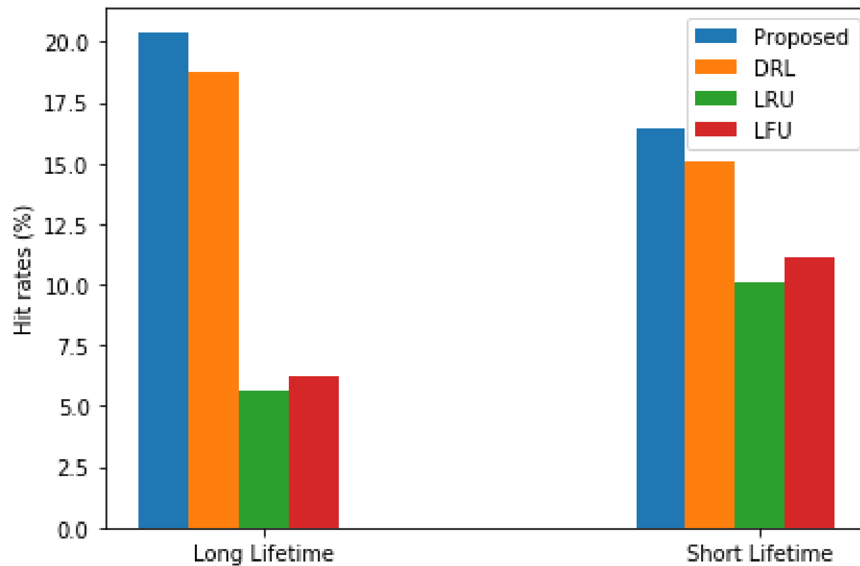

Figure 4.13) Hit rate vs files' life-time

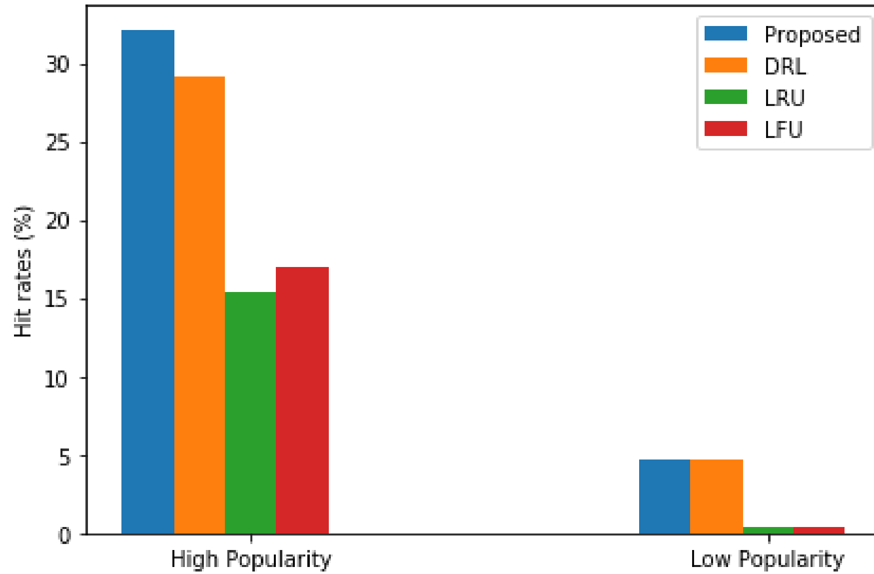

Figure 4.14) Hit rate vs files' popularity

To evaluate the effects of the proposed reward function on the freshness of cached files, we have prepared Table 4.1. This table shows the average freshness for all the cache memory files during the whole simulation for each of the 24 different settings. Note that a freshness value of 1 means the file has expired, and a freshness value of 0 means the file has just been generated. The lower the freshness value, the fresher the file is. In the last row of the table, we can observe the mean value of all those settings. Since in our proposed method, the reward function does not include any punishment for the higher value of freshness, the cached files have slightly older files in the memory compared to the other DRL method, which explicitly considers freshness in its reward function. Nevertheless, the gap between these two methods is marginal.

Table 4.1 Average freshness

Setting	Proposed	DRL
$w = 0.5, \alpha = 0.7$	0.497	0.17
$w = 0.5, \alpha = 0.8$	0.494	0.329
$w = 0.5, \alpha = 0.9$	0.536	0.53
$w = 0.5, \alpha = 1.0$	0.496	0.495
$w = 0.5, \alpha = 1.1$	0.532	0.532
$w = 0.5, \alpha = 1.2$	0.527	0.523
$w = 1, \alpha = 0.7$	0.469	0.468
$w = 1, \alpha = 0.8$	0.457	0.457
$w = 1, \alpha = 0.9$	0.422	0.422
$w = 1, \alpha = 1.0$	0.451	0.450
$w = 1, \alpha = 1.1$	0.411	0.409
$w = 1, \alpha = 1.2$	0.433	0.433
$w = 1.5, \alpha = 0.7$	0.247	0.355
$w = 1.5, \alpha = 0.8$	0.355	0.355
$w = 1.5, \alpha = 0.9$	0.296	0.296
$w = 1.5, \alpha = 1.0$	0.346	0.346
$w = 1.5, \alpha = 1.1$	0.333	0.293
$w = 1.5, \alpha = 1.2$	0.324	0.307
$w = 2, \alpha = 0.7$	0.182	0.27
$w = 2, \alpha = 0.8$	0.279	0.279
$w = 2, \alpha = 0.9$	0.225	0.228
$w = 2, \alpha = 1.0$	0.271	0.275
$w = 2, \alpha = 1.1$	0.249	0.251
$w = 2, \alpha = 1.2$	0.293	0.237
Average	**0.380**	**0.362**

Finally, in order to see the benefits of a hierarchical model architecture, we have implemented a single-layer caching scheme where there are two edge nodes capable of caching, and there is no parent node in the network. For the sake of fair comparison, we keep the sum of memory capacity of these two edge nodes on par with the cumulative memory capacity of the three nodes (one parent node and two edge nodes) in our hierarchical architecture. Figure 4.15 and Figure 4.16 show that the proposed hierarchical architecture can lead to higher cache hit rate and consequently lower energy consumption. The reason is that the parent node in the hierarchical architecture can effectively cache the files, which are commonly popular in multiple regions, thus increasing the overall cache hit rate.

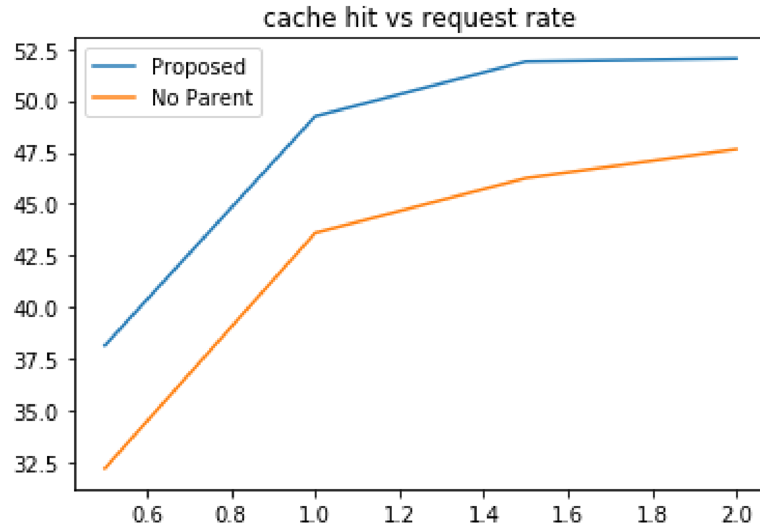

Figure 4.15) Comparison of hierarchical architecture (proposed) and flat architecture (edge caching only). The figure shows the hit rate vs request rates ω

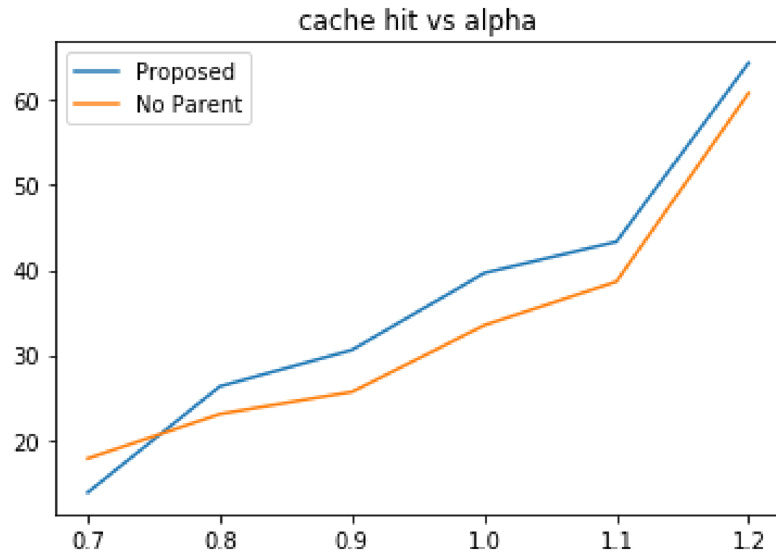

Figure 4.16) Comparison of hierarchical architecture (proposed) and flat architecture (edge caching only) vs skewness

4.4 Summary

In this chapter, we have presented two caching strategies. In one, we have a dominant parent node who receives all of the state information and then chooses a

caching action, acting as an oracle. This centralized architecture enables the parent node to play its part in the caching process, and as a result, we have observed the improvement in cache hit and energy consumption rate. We have also investigated a distributed approach in which every networking node is able to make its own caching decision. In addition to the energy consumption and hit ratio, we have studied the effects of the proposed method on the data freshness, and we have shown that the proposed method tends to store files with a longer lifetime and higher popularity.

Next, we have designed an experiment to show how much a system can benefit from having a hierarchical architecture. We have implemented a single-layer system to compare the caching performance with the two-layer one; we have kept the total available caching memory equal in both systems to make a fair comparison. The experiment results show that the two-layer system model has a clear advantage over the single-layer architecture when comparing the cache hit rates.

Chapter 5. Transfer Learning-based Caching Strategy for IoT Networks

In this chapter, we tackle the challenge of having dynamic file popularity distribution while addressing IoT networks' caching problem. In section 5.1, we first describe the problem and then in section 5.2, we propose a solution to that problem. Section 5.3 evaluates the proposed method's results to see whether it can be an effective solution.

5.1 Problem Statement

Most machine learning algorithms are designed to find a true distribution based on analyzing numerous samples. While these algorithms have proven to be surprisingly successful and accurate, they all share a common assumption that the underlying distribution is stationary and is not going to change. While it might be a valid assumption for some cases, it is not necessarily true everywhere, especially when "time" plays a role. In our caching problem and with realistic settings, the file popularity distribution may change with time. For example, the security sensors' data might be requested with high frequency during nights, but they might not be that popular in the mornings. This is why we need a solution to understand such a system's dynamic needs and adapt itself to meet those needs.

5.2 Proposed Approach

To address this challenge, we propose two modules; one to detect a change in the file popularity distribution and another one to handle the change once it happens. The former is a concept drift detection algorithm, and the latter is a DRL-based caching solution that has adopted transfer learning techniques. We present their workflow in detail in sections 5.2.1 and 5.2.2, respectively.

5.2.1 Concept Drift Detection

To handle concept drift detection, we leverage the most popular clustering technique, the K-Means clustering algorithm [29], and then measure the similarity of the generated clusters. To detect a change in file popularity, we store two batches of recent file requests chronologically, e.g. we store the latest 100 requests and the other 100 requests before that. Then, we cluster those two batches based on two attributes: frequency and lifetime. This phase's result is two sets of K clusters, and we now need to compare the similarity between them. To illustrate the reasoning behind that, imagine that a specific file belongs to the cluster with a highly frequent request rate and long lifetime, but in the second batch, it rests among the least frequently requested files. This difference among clusters might be a good indicator of a change in the file popularity distribution. While the K-Means algorithm is a relatively routine process, measuring the similarity between clusters is not straightforward. Even before starting the process, we can distinguish some problems. For instance, the two K-Means may result in identical clusters but with different labels, and if the similarity measurement process decides based on these labels, the result would not be an accurate one.

Moreover, even sampling from a single distribution does not necessarily lead to the same K-Means clusters. There is an inherent randomness to the problem that we need to consider. One way to influence this randomness is through the batch sizes of the recent requests that we store. Greater batch sizes decrease the randomness, but it also increases the memory consumption and the run-time, which are not desirable. So, there is a trade-off that we need to balance. Furthermore, some randomness is also introduced by the K-Means algorithm itself since it is not a deterministic process. The value of the K in the K-Means can alter this randomness to some extent.

We can approach the relabeling problem in two ways, either relabel clusters based on cluster sizes or rename clusters to have maximal overlap to a target cluster. We have chosen the second solution in which we select the older batch's cluster as the target one and relabel the newer one to have the maximal overlap.

To handle the randomness in the process, we can iterate through multiple trial and error steps to find the batch size and K value that result in consistent results.

Once these initial challenges are addressed, it is time to measure the similarity of the generated clusters. Reviewing the literature in search for related measurements, we came across Jaccard Index [30]. Jaccard index is one of the most popular indices to measure cluster similarity, and it requires reasonably easy calculations. This index returns the ratio of the intersection over the union of two sample sets, as shown in (5.1). This is why the Jaccard index is also known as Intersection-over-Union (IoU). This index can have a value in the range of $[0, 1]$, and for two empty sets, it will return a similarity of one.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \quad (5.1)$$

Figure 5.1 and Figure 5.2 better illustrate how the Jaccard index measures the similarity of two given sets.

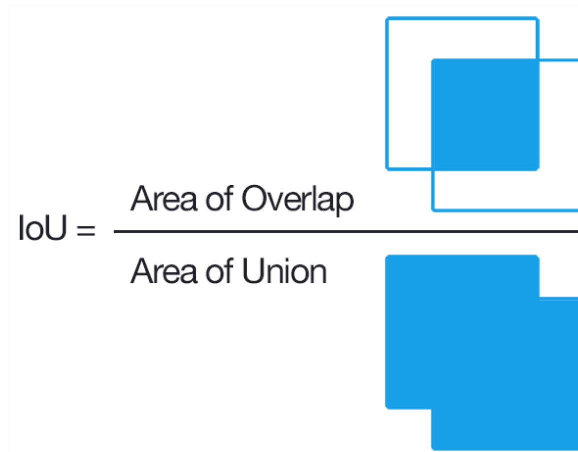

Figure 5.1) Illustration of Jaccard similarity index

Figure 5.2) Jaccard similarity index

Setting a threshold on the Jaccard index's value as a trigger point provides the concept drift detection solution which we are seeking. We will study the performance of such an approach later in this chapter, but for now, let us present how we handle the detected change in the file popularity distribution.

5.2.2 DQN-based Caching Solution with Transfer Learning

Our goal is to maintain excellent caching performance in the presence of a changing environment. Assuming we can effectively detect the changes in file popularity distribution, we still need a solution to efficiently update the caching policy. We have chosen to apply transfer learning techniques to the DRL-based solutions to avoid using the outdated caching policy and increase the speed of training a new DRL agent.

In our specific caching problem in IoT networks, we plan to apply TL to the neural networks inside the DRL solution. For this chapter, we choose the DQN as the DRL algorithm. We chose the DQN for the transfer learning part because it was simpler to implement, and transferring knowledge from a DRL agent to another seemed to be complicated enough, so we tried to avoid unnecessary complexity. Please note, the system model and the caching problem formulation in this chapter is identical to the distributed approach introduced in section 4.2.2. Before we get to the transfer learning part, let us first review how the DQN algorithm works.

In 2015, Google's DeepMind team published [4] and introduced the Deep Q-Network to achieve human-level control in Atari games. Before this work, reinforcement learning applications had been restricted to the problems where valuable features had been crafted for the RL agent or the problem space was fully observable. Mnih *et al.* [4], proposed deep neural networks to change that state of afire and created a deep reinforcement learning agent capable of directly processing the high dimensional sensory input. Recall the definitions of value and Q-functions in (4.3) and (4.4) if any of these two functions are known, we would reach the

optimal policy by simply choosing the action that yields the greatest reward. However, it is usually not the case in real-world scenarios, and thus we use neural networks to approximate them. Figure 5.3 denotes how a neural network is supposed to improve the Q learning algorithm.

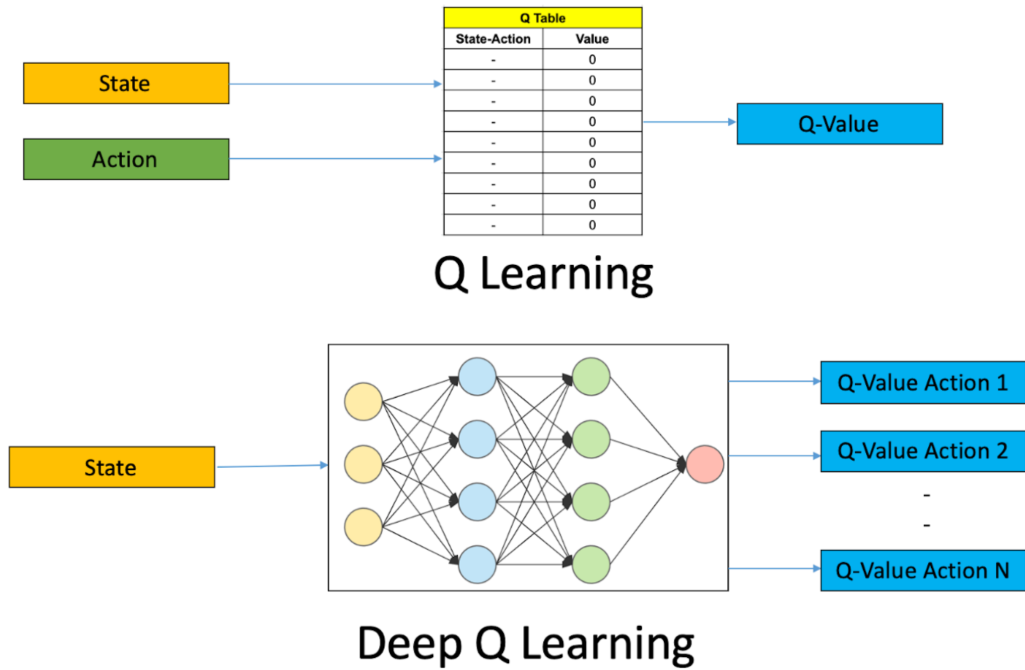

Figure 5.3) Q-Learning and Deep Q-learning

In DQN, the environment's observations are parsed into vectors and then fed to a neural network's input. This NN outputs the Q-value for every action; we will then choose the action with the greatest Q-value, or we can randomly choose an action for the sake of exploration. Figure 5.4 depicts DQN's workflow.

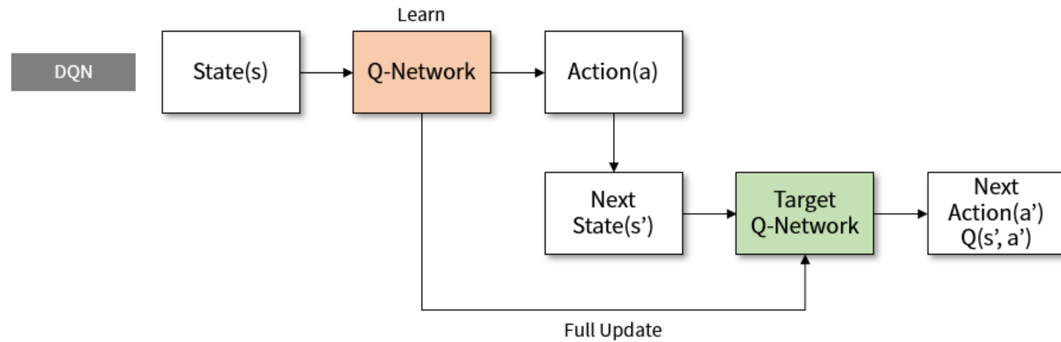

Figure 5.4) DQN workflow

The high-level view of the DQN workflow is as follows:

1- Collect and store samples in the replay buffer with current policy.

2- Randomly select a batch of samples from the experience replay buffer.

3- Use the sampled experience to update the Q neural network.

4- Repeat the previous three steps.

One clever trick used in DQN to further stabilize the system is using two neural networks and updating the parameters after so many steps. One of these NN is called the target network, and the other is the local network. Using a separate target network, updated every so many step with a copy of the latest learned parameters in the local network, helps keep bootstrapping's runaway bias (over-estimates) from dominating the system numerically. Algorithm 2 shows the pseudo-code of the DQN algorithm.

Algorithm 2: Deep Q-learning with Experience Replay

Input: Initial replay memory D to capacity N, action-value function Q with random weights θ, and target action-value function $\hat{Q}$ with weights $\bar{\theta} = \theta$

1: **for** $episode = 1, ..., M$ **do**
2: **for** $t = 1, ..., T$ **do**
3: With probability ϵ select a random action a_t
4: Otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
5: Execute action a_t and observe reward r_t and next state s_{t+1}
6: Store Transaction (s_t, a_t, r_t, s_{t+1}) in
7: Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
8: set y_j as following: ${}_j \leftarrow argmax_\theta \ \mathcal{L}^{clip}(\theta)$
9: Fit value function by regression on Mean-Squared Error (MSE) using SGD:
10: **end for**
11: **end for**

Now that we know how the DQN algorithm works, it is time to study how we can transfer knowledge from a DQN agent to another. So, let us start with the basic concepts of transfer learning.

Transfer learning for neural networks is usually comprised of the following steps:

1. Removing one or a few last layers of the neural net.
2. Freezing the remaining layers, so their weight does not change.
3. Adding new layers.
4. Train the new NN on the new data to update the weight of the new layers.

5. Unfreeze the first layers to fine-tune the whole NN with few iterations.

The last layer is seen as the decision-making layer, e.g. in classification problems, this layer executes the labelling process; this is why we need to replace this layer with a new one when facing a new task. The early layers are seen as the containers of the knowledge gained from the previous task; e.g. the edge detection might be the feature implemented in the first layers, which is why we would like to save them intact.

We then train the recently added layers by the few data points we have for the target task. The good news is that now, the small training data is enough because we have few training parameters.

However, not all of these steps are required; e.g. one might skip the second step and apply backpropagation on all of the layers, it might be problematic since the large gradients would wipe out the good initial values that those layers had held from the source tasks. Step five is also optional, especially if there are not enough data points for the training; the neural net might produce outstanding results even without fine-tuning. Figure 5.5 illustrates the transferring process.

Now that we have presented the problem statement and our solutions, in the next section, let us present the detail of implementing the concept drift detection system and the caching algorithm using DQN and transfer learning. We also investigate the performance of the proposed method from various viewpoints.

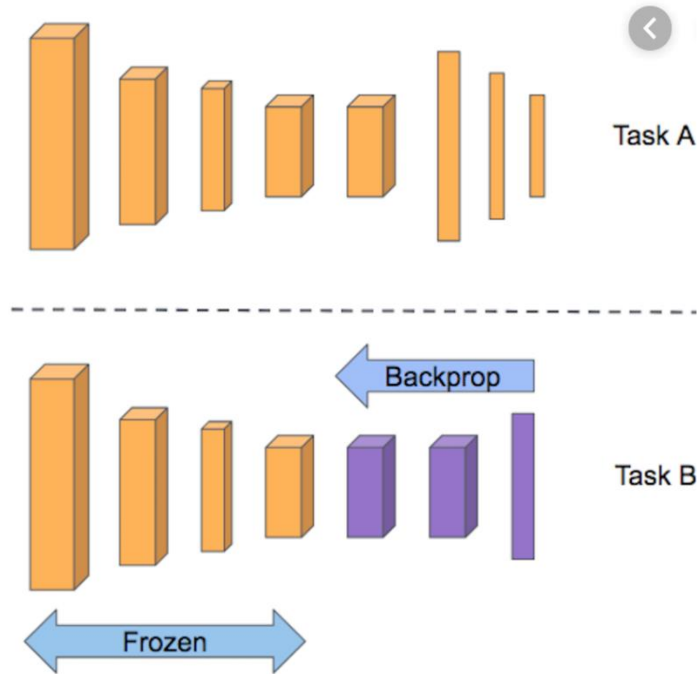

Figure 5.5) Transfer learning illustration

5.3 Evaluation of the Dynamic Caching Scheme

In our experiments we have used Python version 3.8, NumPy [26], TensorFlow 2 [27], Keras, Scikit-learn [31], CluSim [32], and OpenAI's Gym [28] libraries. We ran the simulation on the workstation with an Intel Core i7-9700 CPU and 16G RAM. The architecture is similar to the one depicted in Figure 4.1; there is one parent node, two edge nodes, and one hundred IoT devices producing the data. Each IoT device only produces one type of file with a unique ID and a lifetime; the lifetime is sampled randomly from a uniform distribution with a minimum of 2 and a maximum of 14 time-steps.

The file popularity distribution, the same as before, follows Zipf's law (4.21). In our experiment, we first generate a batch of samples of file requests based on a

specific file popularity distribution. We name this dataset as **main**. Then we alter it to see whether our similarity measurement pipeline is capable of detecting the changes. The **main** dataset is generated with a skewness factor α of 1.2 and the lifetime of the files is an integer value in the range of [2, 14]. The file request rate in our simulation is considered two files per time-step, and of course, this information is unknown to the DRL agent. DRL agent does not use any information except the ones received from the interactions with its environment to make caching decisions.

To generate the **rank 1** altered dataset, we choose the file with the highest popularity ranking and replace it with the least popular file; in the **rank 2** dataset, we replace the second most popular file with the second least popular file; the same logic goes for the **rank 3** dataset. We then feed some subsets of these datasets to the similarity measurement system and obtain the results of Jaccard values shown in Table 5.1:

Table 5.1 Similarity measurement on different datasets

	main	**rank 1**	**rank 2**	**rank 3**
main	1	0.68	0.72	0.81
rank 1	0.68	1	0.54	0.61
rank 2	0.72	0.54	1	0.66
rank 3	0.81	0.61	0.66	1

A Jaccard value of 1 denotes an identical distribution, and a value of 0 denotes no commonality. We can see that the system can effectively distinguish changes in

the distribution, testing on the same distributions returns 1, and as the difference between distribution increases, the Jaccard index value decreases. The **rank 1** and **rank 2** distributions have the most difference (since their top popular files are replaced with two different least popular files), and consequently, the Jaccard value between these two distributions has the smallest value.

To find the optimal value for the sample size, we have run a trial and error experiment, and we found 80 to be a good option for the batch size. It is small enough to does not consume too much run time, and it is large enough for the Jaccard index to be an effective measurement of the similarity.

To test the transfer learning for the DRL-based caching strategy, we have implemented a DQN agent with the following neural network architecture: two hidden layers with the *relu* activation function, and 24 and 16 neurons in the first and second layer, respectively. The discounting factor γ is set to 0.95, and the learning rate is set to 0.001. We also update the target network's parameters every ten steps.

After a training phase is completed, we change the distribution to one of the rank 1-3 datasets. We then remove the last layer of the neural networks and place a new layer with a random initial value. We then freeze the first previous layers and redo the training phase based on the updated environment samples. In Figure 5.6, the blue line is the DRL agent's rewards after the transfer and during its training. The red line is the corresponding one for a regular DRL agent without any knowledge transfer in the same environment. We can see that the transferred agent learns significantly faster than a regular agent.

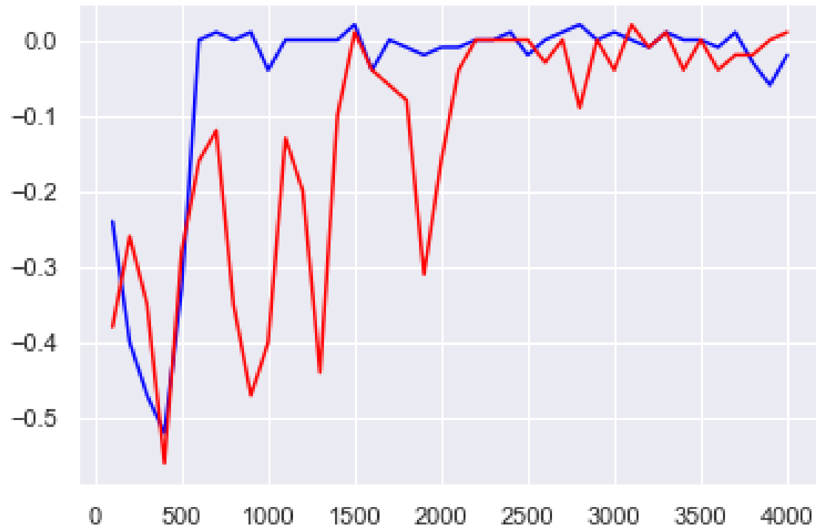

Figure 5.6) The realized reward during the training phase, before and after transfer

In this figure, the vertical axis is the realized reward, and the horizontal axis is time. We can see that it takes a randomly initialized DRL agent about 2000 time-steps to reach convergence while the agent that benefits from transferred knowledge can get to that point in about 600 time-steps.

Table 5.2 shows what would happen if we simply chose to continue caching using the agent trained on the previous environment; we can see that there would be a significant drop in the cache hit rates. On the other hand, transferring knowledge can accelerate the training phase and boost the cache hit ratio in the new environment, which is the ultimate goal of this work.

Table 5.2 Comparison of caching performance before and after knowledge transfer

	Cache hit on Main	**Cache hit on Rank 1**
Trained on Main	**119**	**89**
Trained on main and transferred to rank 1	**-**	**192**

5.4 Summary

In this chapter, we have introduced the concept drift basics, and then we have employed K-Means clustering algorithm alongside the Jaccard similarity measurement to detect whether a change has occurred in the files' popularity distribution. This approach proved to be effective in detecting the changes in the file popularity distribution.

We have also applied transfer learning technique to the DQN algorithm to obtain a caching method that can handle non-stationary environments. We have described our experiments' settings and demonstrated the obtained results. We have shown that transferring knowledge can help the DRL agent significantly speed up its learning process and converge to the same or better performance.

Chapter 6. Conclusion

6.1 Contributions of the Thesis

The following lists our contributions in the thesis:

1. We have formulated the caching problem as a Markov decision process (MDP) while considering IoT networks' restrictions such as data lifetime and limited energy resources.
2. We have proposed the two-layer hierarchical architecture for caching nodes and claimed that such an architecture can further help with caching performance. To provide some evidence for this claim, we have designed some experiments and compared the same caching algorithm's performance in two networks with different architecture, one with a two-layer hierarchical structure and the other with a single layer one. The results of our experiments have shown that the hierarchical architecture significantly helps with caching performance.
3. We have designed two DRL-based caching methods which are especially suitable for IoT networks with transient data. These caching methods are based on a deep reinforcement learning algorithm, namely Proximal Policy Optimization (PPO). We have applied this DRL algorithm with two different approaches, a centralized and a distributed approach.
4. We have conducted extensive experiments and gather comprehensive results and comparisons with different caching methods and architectures. The experimental results indicate that the proposed caching strategies can significantly improve the cache hit rate and reduce energy consumption rate

compared to the conventional caching schemes least recently used (LRU), least frequently used (LFU) and the existing DRL solution in [18]. The proposed algorithms also demonstrate exemplary performance in handling the limited lifetime of data.

5. We have also addressed caching problem where file popularity does not follow a stationary distribution. To detect a change in the distribution, we have leveraged clustering algorithms and apply similarity measurement techniques; and then triggered an update process to retrain the DRL algorithms accordingly.
6. To make the caching policy adapt to the changes in the file popularity distribution, we have implemented a DQN-based caching algorithm and then applied transfer learning techniques to the DRL agent. The preliminary results have shown promising performance in both concept drift detection and the TL-applied training phase.

6.2 Future Works

Firstly, we could not investigate the DQN-based caching methods' performance in more detail because of our limited time. Thus, there is more room to obtain more results for the transferred DQN agent's caching performance, similar to the ones we have in section 4.3.2.

Furthermore, one can try to replace the DQN with more sophisticated DRL algorithms such as PPO. We chose the DQN for the transfer learning part because it is simpler to implement, and transferring knowledge from a DRL agent to another is complicated enough, so we tried to avoid unnecessary complexity. However,

based on our works in chapter 4, we think PPO might deliver a better final caching performance.

We only tried one of the transfer learning methods, but there are more TL approaches to explore. Google's DeepMind team has published their work on progressive neural networks [33] in which the authors proposed some different TL approaches one can apply to DRL algorithms.

Lastly, the whole network can be scaled and tested against a more general setting, e.g. more layers can be added between the sources and the users, or the number of the leaf and parent nodes can be increased. Such a setting might be a great way to compare the distributed and the centralized version against each other in more detail.

References

1. Nasehzadeh, A. and P. Wang. A Deep Reinforcement Learning-Based Caching Strategy for Internet of Things. in 2020 IEEE/CIC International Conference on Communications in China (ICCC). 2020. IEEE.
2. Nordrum, A., Popular internet of things forecast of 50 billion devices by 2020 is outdated. IEEE spectrum, 2016. 18(3).
3. Sutton, R.S. and A.G. Barto, Reinforcement learning: An introduction. 2018: MIT press.
4. Mnih, V., et al., Human-level control through deep reinforcement learning. nature, 2015. 518(7540): p. 529-533.
5. Nguyen, C.T., et al., Transfer Learning for Future Wireless Networks: A Comprehensive Survey. arXiv preprint arXiv:2102.07572, 2021.
6. Pan, S.J. and Q. Yang, A survey on transfer learning. IEEE Transactions on knowledge and data engineering, 2009. 22(10): p. 1345-1359.
7. Sadeghi, A., G. Wang, and G.B. Giannakis, Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. IEEE Transactions on Cognitive Communications and Networking, 2019. 5(4): p. 1024-1033.
8. Zhong, C., 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. IEEE.
9. Zhong, C., M.C. Gursoy, and S. Velipasalar. Deep multi-agent reinforcement learning based cooperative edge caching in wireless networks. in ICC 2019-2019 IEEE International Conference on Communications (ICC). 2019. IEEE.
10. Wang, C., et al. Content-centric caching using deep reinforcement learning in mobile computing. in 2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS). 2019. IEEE.
11. He, Y., et al. Integrated computing, caching, and communication for trust-based social networks: A big data DRL approach. in 2018 IEEE Global Communications Conference (GLOBECOM). 2018. IEEE.
12. He, Y., N. Zhao, and H. Yin, Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach. IEEE Transactions on Vehicular Technology, 2017. 67(1): p. 44-55.
13. Hu, R.Q., Mobility-aware edge caching and computing in vehicle networks: A deep reinforcement learning. IEEE Transactions on Vehicular Technology, 2018. 67(11): p. 10190-10203.

14. Wei, Y., et al., Joint optimization of caching, computing, and radio resources for fog-enabled IoT using natural actor–critic deep reinforcement learning. IEEE Internet of Things Journal, 2018. 6(2): p. 2061-2073.

15. He, X., et al., Green resource allocation based on deep reinforcement learning in content-centric IoT. IEEE Transactions on Emerging Topics in Computing, 2018. 8(3): p. 781-796.

16. He, X., K. Wang, and W. Xu, QoE-driven content-centric caching with deep reinforcement learning in edge-enabled IoT. IEEE Computational Intelligence Magazine, 2019. 14(4): p. 12-20.

17. Wang, X., et al., Federated deep reinforcement learning for internet of things with decentralized cooperative edge caching. IEEE Internet of Things Journal, 2020. 7(10): p. 9441-9455.

18. Zhu, H., et al., Caching transient data for Internet of Things: A deep reinforcement learning approach. IEEE Internet of Things Journal, 2018. 6(2): p. 2074-2083.

19. Bastug, E., M. Bennis, and M. Debbah. Anticipatory caching in small cell networks: A transfer learning approach. in 1st KuVS workshop on anticipatory networks. 2014.

20. Nagaraja, B.B. and K.G. Nagananda. Caching with unknown popularity profiles in small cell networks. in 2015 IEEE Global Communications Conference (GLOBECOM). 2015. IEEE.

21. Bharath, B., K.G. Nagananda, and H.V. Poor, A learning-based approach to caching in heterogenous small cell networks. IEEE Transactions on Communications, 2016. 64(4): p. 1674-1686.

22. Zhang, Z., et al. Cache-enabled adaptive bit rate streaming via deep self-transfer reinforcement learning. in 2018 10th International Conference on Wireless Communications and Signal Processing (WCSP). 2018. IEEE.

23. Schulman, J., et al., Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

24. Schulman, J., et al. Trust region policy optimization. in International conference on machine learning. 2015. PMLR.

25. Hill, A., et al., Stable Baselines. GitHub repository (2018).

26. Harris, C.R., et al., Array programming with NumPy. Nature, 2020. 585(7825): p. 357-362.

27. Abadi, M., et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.

28. Brockman, G., et al., Openai gym. arXiv preprint arXiv:1606.01540, 2016.

29. MacQueen, J. Some methods for classification and analysis of multivariate observations. in Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. 1967. Oakland, CA, USA.

30. Jaccard, P., The distribution of the flora in the alpine zone. 1. New phytologist, 1912. 11(2): p. 37-50.

31. Pedregosa, F., et al., Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 2011. 12: p. 2825-2830.

32. Gates, A.J. and Y.-Y. Ahn, CluSim: a python package for calculating clustering similarity. Journal of Open Source Software, 2019. 4(35): p. 1264.

33. Rusu, A.A., et al., Progressive neural networks. arXiv preprint arXiv:1606.04671, 2016.