Spring 2023

Container Caching Optimization based on Explainable Deep Reinforcement Learning

Divyashree Jayaram
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Artificial Intelligence and Robotics Commons, and the OS and Networks Commons

Recommended Citation
DOI: https://doi.org/10.31979/etd.rmr2-9qcf
https://scholarworks.sjsu.edu/etd_projects/1231

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
Container Caching Optimization based on Explainable Deep Reinforcement Learning

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Divyashree Jayaram

May 2023
The Designated Project Committee Approves the Project Titled

Container Caching Optimization based on Explainable Deep Reinforcement Learning

by

Divyashree Jayaram

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Genya Ishigaki  Department of Computer Science

Dr. Faranak Abri  Department of Computer Science

Dr. Fabio Di Troia  Department of Computer Science
ABSTRACT

Container Caching Optimization based on Explainable Deep Reinforcement Learning

by Divyashree Jayaram

Serverless edge computing environments use lightweight containers to run IoT services on a need basis i.e only when a service is requested. These containers experience a cold start up latency when starting up. One probable solution to reduce the startup delay is container caching on the edge nodes. Edge nodes are nodes that are closer in proximity to the IoT devices. Efficient container caching strategies are required since the resource availability on these edge devices is limited. Because of this constraint on resources, the container caching strategies should also take proper resource utilization into account. This project tries to further improve the startup latency and provides explanations of a resulting caching strategy by using Explainable Reinforcement Learning (XRL). This project proposes a method that uses Deep Reinforcement Learning (DRL) to learn a policy to efficiently perform container caching by maximizing the expected cumulative discounted reward. The proposed method uses a probability distribution function to distribute IoT service requests to the other edge devices by taking advantage of the heterogeneous and distributed character of edge computing environments. An XRL method called the action-influence model will be integrated into the DRL framework so that a service provider of the edge computing service can understand the logic behind the caching and further improve the performance. A comprehensive analysis of the impact of caching decisions on overall performance of the DRL algorithm is provided.
ACKNOWLEDGMENTS

I would like to thank my family for being understanding and supportive throughout the duration of this project. I would like to thank Professor Genya Ishigaki for his invaluable support and guidance throughout the duration of this project, as his expertise, insights, and feedback have been crucial to the success of this project. I also would like to thank the committee members for their time and insightful comments during the project’s evaluation process, which helped to determine its final form and ensure its quality. Finally, I would like to thank my friends for their constant support and encouragement.
# TABLE OF CONTENTS

## CHAPTER

1 Introduction ......................................................... 1
   1.1 Serverless Edge Computing .................................. 2
   1.2 Deep Reinforcement Learning ............................... 4
   1.3 Need for Explainablity in Deep Reinforcement Learning .... 5
   1.4 Project Statement ........................................... 5

2 Related Works ...................................................... 7

3 System Architecture ............................................... 9
   3.1 Network Topology ............................................. 10
   3.2 Request ....................................................... 11
   3.3 Delay ........................................................ 11
   3.4 Request Redistribution Policy .............................. 12

4 Problem Formulation with Expected Explanation ............... 13

5 Method ............................................................. 15
   5.1 DQN .......................................................... 15
      5.1.1 State Space ............................................. 16
      5.1.2 Action Space .......................................... 16
      5.1.3 Reward ............................................... 17
   5.2 XRL framework for DQN ................................. 17

6 Experiments and Results .......................................... 19
   6.1 Experiment Setting: Topology and Computing Nodes .......... 19
6.2 Experiment Setting: Request Generation ......................... 19
6.3 Experiment Setting: Reward ........................................ 19
6.4 DQN Hyper-parameters .............................................. 21
6.5 Performance on DRL-based Caching Decisions ..................... 21
6.6 Explanation of DRL-based Caching Decisions ...................... 22
6.7 Explanation of Request Arriving Behaviors and Topological
Importance of Nodes ................................................... 27

7 Conclusion and Future Work .......................................... 28

LIST OF REFERENCES .................................................... 29

APPENDIX

Appendix ................................................................. 32

A.1 An Upper Bound for the Caching Rewards .......................... 32
CHAPTER 1

Introduction

Over the past decade, the Internet of Things (IoT) has developed quickly, connecting an increasing number of machines, sensors, and devices to the internet. The IoT is anticipated to reach new levels of innovation and growth with the introduction of 5G networks and serverless edge computing. IoT devices, generally generate a large volume of data, which needs to be processed and analyzed in real-time for critical applications. For instance, in a smart city application, sensors that track traffic flow produce data that must be immediately processed to improve traffic flow and lessen congestion. The traditional cloud computing environment, however, is unable to meet the IoT applications’ high responsiveness and low latency requirements since doing so necessitates the transmission of data over the network for processing and analysis, which adds to network latency. Additionally, due to the need for specialized hardware and software to handle the high volume of streaming data generated by IoT devices, traditional cloud computing may not be able to do so.

IoT applications therefore require a computing environment that can support the aforementioned requirements. Edge computing is the perfect option for these kinds of applications because it moves computing power closer to the point of data generation, eliminates the need for data transfer to the cloud, and allows for quicker response times [1]. The edge computing environment utilizes the processing capacity of edge nodes, such as switches and routers, that are fully distributed over multiple geographic locations and network typologies, ensuring optimal proximity to data sources and end users [2]. These edge nodes, however, only have an insufficient amount of processing and storage power. Thus, the Serverless computing paradigm emerges.

With Serverless computing, developers may create and deploy apps without worrying about the underlying infrastructure because the cloud provider will handle
the management of the resources required to execute and scale the programs. Serverless computing helps mitigate the resource limitation of edge computing by providing a more efficient way of deploying applications to edge nodes. Instead of deploying the application to each node, developers can deploy a serverless function that is executed on demand whenever needed, reducing the need for expensive and complex software updates on the edge nodes.

1.1 Serverless Edge Computing

Serverless edge computing is a cutting-edge computing method that combines the benefits of serverless computing and edge computing making it a solid framework for constructing real-time, responsive applications that can provide seamless user experiences for data-intensive applications such as Internet of Things (IoT) devices [3].

Serverless computing environments typically use a function as a fundamental unit of computation. These functions tend to be event-driven. i.e., They are triggered in response to a specific event, for instance, an IoT sensor reading or a user request. Therefore serverless computing environment are also known as functions-as-a-service, or FaaS [4]. Whenever, a function request is made by an IoT device or a user, the function code is executed by encasing it inside a lightweight container. However, the container must first be instantiated before the function can be executed. This entails preparing the necessary resources, getting the container up and running, launching the function, and then returning any output to the user [5]. In a naive implementation of serverless edge computing, the containers are deleted after the user’s request has been fulfilled.

However, the container-based function execution strategy results in a considerable delay due to the container instantiation process known as cold start delay. Along
with the instantiation process’s delay, the heterogeneity and resource availability limitations at the edge also contribute to an increase in cold start up latency [6]. Cold start up latency has a significant negative influence on critical IoT applications because it delays data processing, which might affect the application’s overall performance. For instance, even a small delay of just few milliseconds, in processing sensor data could cause missing or delayed alarms for maintenance concerns or safety dangers in a real-time monitoring system for industrial equipment. Hence, the cold startup latency has to be mitigated in order to create a platform for IoT applications that is low latency and highly responsive. Figure 1 shows a table with cold start up latency in milliseconds (ms) computed for 50,000 functions on three serverless platforms such as AWS lambda, Azure and Google cloud using Nodejs6*, indicating a delay of several milliseconds which is not acceptable by time critical applications such as the ones mentioned in the example above [7].

![Table](image)

<table>
<thead>
<tr>
<th>Provider-Memory</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS-128</td>
<td>265.21</td>
<td>189.87</td>
<td>7048.42</td>
<td>354.43</td>
</tr>
<tr>
<td>AWS-1536</td>
<td>250.07</td>
<td>187.97</td>
<td>5368.31</td>
<td>273.63</td>
</tr>
<tr>
<td>Google-128</td>
<td>493.04</td>
<td>268.5</td>
<td>2803.8</td>
<td>345.8</td>
</tr>
<tr>
<td>Google-2048</td>
<td>110.77</td>
<td>52.66</td>
<td>1407.76</td>
<td>124.3</td>
</tr>
<tr>
<td>Azure</td>
<td>3640.02</td>
<td>431.58</td>
<td>45772.06</td>
<td>5110.12</td>
</tr>
</tbody>
</table>

![Figure 1](image)

Therefore, most of serverless platforms cache the used container instance at the edge nodes so they can be reused again for subsequent requests of the same function. Reusing the containers, known as warm start, can significantly improve the delay incurred during the cold start and increase the responsiveness of the system [8]. An effective container caching approach is essential to significantly reduce the cold start up latency and improve the performance of serverless edge computing environment. First-in-first-out (FIFO), least-frequently-used (LFU), and least-recently-used (LRU),
among other standard caching approaches, can greatly increase the cache hit rate [9]. However, because they are generic caching methods, they might not take into account the IoT applications need for low latency and high responsiveness [10]. Also the generic container caching approaches do not take into account dynamic workload patterns and heterogeneity of the serverless edge platform. As a result, we want intelligent container caching strategies that dynamically adjust to workload patterns and optimize the use of available resources, lowering cold start times and enhancing system performance as a whole.

1.2 Deep Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique that enables an agent to learn how to make the best decisions possible based on feedback in the form of rewards from the environment. It entails teaching the agent how to take actions in a way that maximizes a cumulative reward signal over time. Over the past few years, RL has made considerable strides, and the merging of RL with Deep Neural Networks (DNN) has resulted in Deep Reinforcement Learning (DRL). Compared to traditional RL, DRL performs better in complex and dynamic situations and can handle high-dimensional input data and learn optimal decision making policies. Due to this learning ability of DRL algorithms, they can be trained on the edge nodes enabling real-time decision-making capabilities.

This project incorporates a Deep Reinforcement Learning (DRL) approach which learns from experiences to design more complex container caching strategies that take into account a variety of elements, including workload patterns, edge computing environment, and IoT device characteristics. These strategies can improve system performance by reducing latency, increasing throughput, and enhancing energy efficiency.
1.3 Need for Explainability in Deep Reinforcement Learning

Given the complexity of DRL algorithm and its decision-making strategy through trial and error, it might be challenging to grasp how the algorithm arrived at its decisions. This can make it challenging to understand the logic underlying the algorithm’s decisions, which can be problematic in safety-critical applications in particular. In order to increase accountability, openness, and trust in these potent decision-making systems, we therefore require approaches to better comprehend and explain the justification for the decision-making process of DRL algorithms [11]. Motivated by these challenges, eXplainable Deep Reinforcement Learning (XRL) has been introduced. XRL provides clear and transparent explanation of DRL’s decision making process.

According to [11], a number of cutting-edge XRL approaches are available that explain various DRL components. These XRL methods are divided into different categories, including model inspection, policy explanation, objective explanation, result explanation, and more. Along with providing explanations, XRL also offers the ability to increase the effectiveness and efficiency by identifying potential biases or areas for improvement.

1.4 Project Statement

This project first aims to train a DRL algorithm, specifically a Deep Q Network (DQN), to learn an efficient container caching policy on a simple serverless edge computing environment for IoT requests with small number of edge nodes and requests. Further, it showcases the usage of XRL in the container caching problem. In particular, we employ Action Influence Model (AIM) [12] to reveal the relationship between caching actions and the associated cumulative rewards. However, there are many existing research available that focus on optimizing container caching strategies.
This project aims to emulate and incorporate the concepts and principles of existing container caching optimization approaches to train a DRL algorithm. The major contribution of this project is to demonstrate a method to extract the most impactful caching actions taken by a complex DRL agent and provide a part of the reason why the agent is able to improve the cache hit rate using XRL. This type of explanation could enhance the accountability of edge computing systems (and their operators) and make it easier to make extra human intervention to further improve the performance. The proposed XRL-based method is evaluated on aforementioned simple scenario so that the extracted impactful actions can be compared with the theoretically optimal actions.
CHAPTER 2
Related Works

Recent years have seen a considerable increase in interest in serverless edge computing as a result of the growth of IoT devices and the requirement for processing and response times that are low-latency. The effective management of container caching, which entails storing containers at the edge devices for quick access and execution, is one of the important challenges in this context.

Container caching in serverless edge computing for IoT devices has been studied extensively and has shown to be effective in lowering latency and enhancing system responsiveness [13][14][15][16]. However, due to the serverless edge computing system’s limited resource availability, retaining the containers in the edge devices impacts the system’s overall performance [17] [18]. To mitigate the impact caused by retaining containers several existing studies have used machine learning algorithms to learn the dynamicity of the workload and optimally place the containers to minimize the impact.

In a serverless edge computing context, Jeon et al. [10] employ a DRL technique to develop a caching policy to cache containers. The Multi-agent Deep Deterministic strategy Gradient (MADDPG) approach is used by the authors to successfully train a Multi Agent RL (MARL) model to cooperate with one another and develop caching strategy in order to meet the response time QOS. The current study, however, is different from the aforementioned study in that it focuses on single centralized RL agent that effectively learns caching policy using the experience gained from running a DQN algorithm. The current study also takes into account the request distribution similar to the approach used in [13] for load balancing and enhances the reliability and availability of the edge devices to fulfill an incoming function request.

Many studies [19][20][21] have used Explainability to understand the decision
made by learning algorithms. Explainability can have a number of advantages, including increased responsibility and compliance, increased efficiency and productivity, improved decision-making and problem-solving, and better collaboration between humans and machines. However, majority of these studies so far have been on OpenAI gym environments. The current study is unique in that it employs an XRL approach to better understand the caching decisions made by the DRL algorithm on the state variables by generating a graph for caching containers in a serverless edge computing environment. Explainable reinforcement learning (XRL) is becoming more and more popular as a result of the growing requirement to comprehend and analyze the decision-making process of RL algorithms in critical domains such as autonomous driving and healthcare in order to construct a transparent and reliable RL model. According to [22], there are several studies that have used different XRL approaches for different purposes such as visualization, queriable explanations, policy summerization etc., to understand the decision making process of RL agent on different custom environments and OpenAI gym environments. Structural Causal Model (SCM), is one of the policy summerization techniques which uses a causal graph to explain the causal relationships between variables. The nodes of graph represents the variables and the edges represent the relationship between variables. AIM extends the causal graph of SCM by using associating specific actions to each of the edges of the graph indicating the causal influence of these actions on the variables [12].
CHAPTER 3
System Architecture

This project considers a three tier system architecture as shown in Figure 2.

• **Tier 1** consists of the edge devices. Edge devices are devices that are present at the edge of the network generally responsible for sending a function request to the edge nodes. The edge devices can range from sensors to IoT devices such as security cameras, smart watches or smart traffic sensors etc.

• **Tier 2** consists of edge nodes. The edge nodes are typically computing nodes that are located near to the edge devices and spread across a wide geographical area. The edge nodes are connected to each other and to the cloud data center via communication links. Each edge node has some cache capacity to cache a limited number of containers of different function types that can cater to the request for a function type from edge devices.

In the current system architecture a subset of edge nodes are designated to directly receive requests for a function type from the edge devices and are known as entry nodes. These entry nodes either fulfill the request from the edge device or distribute the request by selecting in random one of the remaining edge nodes (non-entry nodes) depending on whether there exists a container cached of the requested function type in it.

The selected edge nodes fulfills the request for a function type if it has a container cached of the requested type. If it does not have a container cached and has not yet maxed on the capacity to cache containers, it instantiates a container of the requested function type and caches it for future use. Otherwise, it forwards the request to the cloud nodes.

• **Tier 3** consists of cloud nodes. The cloud nodes are assumed to have unlimited cache capacity and have containers to fulfill the request for any function type
Figure 2: System Architecture.

generated by the edge devices.

3.1 Network Topology

A communication network \( G = (N, L) \) consists of nodes \( N \) and links \( L \). The set of nodes includes three types of nodes such as edge devices \( N_D \), edge nodes \( N_E \), and cloud nodes \( N_C \). Here, edge and cloud nodes are called computing nodes.

Each edge computing node \( n \in N_E \) has a cache size of \( S(n) \) and a list \( A^{(r)}(n) \) of numbers of container caches available at time \( r \) by function types. A link \( e_{i,j} \) represents a connection between two nodes \( n_i \) and \( n_j \).

Among the edge nodes \( N_E \), a subset of nodes are designated as entry nodes, that act as a point of entry to the edge network. They are responsible for handling the arriving request or redistributing them.
3.2 Request

The system is assumed to be logically time slotted, and there are $M$ requests arriving at each discrete time slot $r$.

A request $R_t^{(r)}$ generated by an edge device at time slot $r$ contains two attributes: (Request-type $t_i$, max$_{hop}$). Request-type indicates a requested function type. The max$_{hop}$ is a predefined value, similar to Time-To-Live (TTL), that determines how many hops can a request take among the edge nodes, before forwarding it to the cloud. The max$_{hop}$ is decremented whenever the request makes a hop.

3.3 Delay

There are two types of delays considered in our model: request propagation delay and container instantiation delay. Let $P(R_t^{(r)}) = (n_1, e_{1,2}, n_2, ..., n_K)$ denote a path (a sequence of nodes and edges) from a device that generated a request $R_t^{(r)}$ to a computing node that processed the request. The request propagation delay $d_p$ of a request $R_t^{(r)}$ is

$$d_p(R_t^{(r)}) = \sum_{e_{i,j} \in L(P(R_t^{(r)}))} w(e_{i,j}),$$

where the function $L$ extracts all the links $e_{i,j}$ in a given path, and $w$ shows the propagation delay of a given link.

The container instantiation delay $d_c$ of a request $R_t^{(r)}$ indicates how long it takes to prepare a requested container at a specific computing node.

$$d_c(R_t^{(r)}) = \begin{cases} 0 & \text{if } A^{(r)}(n)[t] > 0 \\ C & \text{if } A^{(r)}(n)[t] = 0 \end{cases},$$

(1)

where $n$ is a computing node that the request resides at the current moment, and $C \in \mathbb{N}$ is a constant relative to the warm start scenario.
3.4 Request Redistribution Policy

This paper assumes a random request redistribution policy similar to the approach in [13] for load balancing among edge nodes. For a given request $R^{(r)}_t$, the request handling and redistribution at each computing node $n \in N_E$ is conducted as follows.

- When a computing node is an entry node and $A^{(r)}(n)[t] > 0$, node $n$ fulfills the request (warm start).
- When a computing node is an entry node and $A^{(r)}(n)[t] = 0$, the entry node selects one of its neighbor nodes at random, decrements the $max_{hop}$ value by 1, and forwards the request.
- When a non-entry computing node receives a request from entry or peer computing nodes, the node checks $A^{(r)}(n)[t]$ and $max_{hop}$.
  - If $A^{(r)}(n)[t] > 0$, then it fulfills the request (warm start with request distribution).
  - If $A^{(r)}(n)[t] = 0$ and $max_{hop} > 0$, then it selects one of its neighbor nodes at random, decrements the $max_{hop}$, and forwards the request.
  - If $A^{(r)}(n)[t] = 0$ and $max_{hop} = 0$, the node checks if there is a spare capacity to instantiate the requested function.
    * If $\sum_{t=t_1}^{t_m} (A^{(r)}(n)[t]) < S(n)$, the node instantiates the requested function container (cold start). It caches the container for future use after this time step.
    * If $\sum_{t=t_1}^{t_m} (A^{(r)}(n)[t]) = S(n)$, the node forwards the request to the cloud for processing, accepting extra propagation delay (cloud forward).

This random distribution at each computing node collectively defines a path $P(R^{(r)}_t)$ from a request source to the request destination.
CHAPTER 4

Problem Formulation with Expected Explanation

The overall objective of this project is divided into two distinct targets: Target 1 and Target 2.

• **Target 1: Training a DRL algorithm to optimize container caching**
  
  The goal of Target 1 is to train a DRL agent to choose an optimal set of containers $A^{(r)}(n)[t]$ cached at each edge node $n$ to increase the number of requests served by warm starts or warm starts with request distribution. This leads to the reduction of overall latency and improves the responsiveness of the system, as pointed out in [18, 10].

  For a function request type $R^{(r)}_t$ at a given node $n$, the container caching decision depends on variable $A^{(r)}(n)[t]$, which indicates if a requested container is available to process the request or not. Also, the capacity constraint also depends on the available caches in the node. Therefore, the objective function is formulated as a minimization problem of the sum of propagation and instantiation delays over all requests.

  \[
  \text{Minimize } \sum_{\{R^{(r)}_t\}} \left( d_p(R^{(r)}_t) + \alpha d_c(R^{(r)}_t) \right), \tag{2}
  \]

  where $\alpha$ is a scaling constant.

  The propagation delay $d_p(R^{(r)}_t)$ depends on the placement of container caches on the edges nodes, and the container instantiation delay $d_c(R^{(r)}_t)$ depends on the availability of the container caches on computing nodes. Therefore, the goal is to find an optimal container caching decision that takes into account the request generation pattern and strategically places the container caches at demanded computing nodes.
• **Target 2: Explaining the learning of DRL using explainability**

When solving the problem with DRL, it is difficult to understand why a certain set of caching actions provides a minimal total delay. This is because the caching decisions at every computing node and the random redistribution process collectively map each request to a specific computing node. To enhance the accountability of such DRL decision-making, we extract the most important caching actions that have been taken in a long run of DRL over multiple time slots. By providing the list of significant actions, edge computing operators should be able to validate and further improve the automated caching decisions.
CHAPTER 5

Method

In this section, we first describe a popular DRL technique, Deep Q Network (DQN) that solves the optimization problem defined in eq. (2). Furthermore, we summarize an EXplainable RL method, called Action Influence Model (AIM) that reveals the underlying relationship between RL actions and cumulative rewards (the objective function).

5.1 DQN

A DQN is a type of RL agent that combines the RL algorithm with Deep Neural Networks (DNN) [23]. It is a variant of the Q-learning technique that maps a state-action pair to the expected discounted future reward using a deep neural network as the Q-function. The algorithm can learn complex correlations between states and actions due to DQN’s ability to automatically extract relevant characteristics from the input and represent them in a high-dimensional space. Through a series of observations, actions, and rewards, the agent in DQN interacts with the environment. The agent wants to develop the ability to choose actions that will result in the greatest overall reward in the future.

The goal of the DQN agent is to learn an optimized caching policy to realize low startup latency. The typical working of a DQN agent is as follows:

- During the exploration phase, at each time step $r$, the DQN agent observes a state $s_r$, takes an action $a_r$ using an epsilon greedy policy, receives a reward $Reward_r$ reaches the next state $s_{r+1}$. The DQN agent stores this information in a replay buffer.

- After sufficient experience have been collected in the training phase, DQN agent then samples the data in replay buffer to update the Q-value for each possible action given in equation (3). The Q-value is the expected cumulative reward
with a discount factor $\gamma$, if the agent selects a particular action $a$ for a policy $\pi$.

$$Q^*(s,a) = \max \mathbb{E} [\text{reward}_r + \gamma \text{reward}_{r+1} + \gamma^2 \text{reward}_{r+2} + ... | s_r = s, a_r = a, \pi]$$  \hspace{1cm} (3)

- During the exploitation phase, the DQN agent uses this learned Q-values to take actions in real time.

The following are state, action and reward models used for the current study.

### 5.1.1 State Space

A caching state at each time step $r$ is defined as a $(T \times N_E)$ two-dimensional vector $V_s^{(r)}$, where $T$ is the number of predefined function types. Each element of the vector can be indexed by a computing node and a function type and represents the number of available function (container) caches at the corresponding node.

For example, assume the environment where there are two function types $\text{ReqType0}$ and $\text{ReqType1}$, and five computing nodes $n_0, ..., n_4$ in the network. Each node $n_i$ caches containers for the two function types. Therefore, at a time slot $r$, if the state vector, $V_s^{(r)}$ is $[[0, 2], [3, 1], ...]$, it means that node $n_0$ caches zero $\text{ReqType0}$ and two $\text{ReqType1}$ containers, and node $n_1$ caches three $\text{ReqType0}$ and one $\text{ReqType1}$. Note that, when the state vector is fed to DQN, we use a vector flattening function to make it an equivalent one-dimensional vector.

### 5.1.2 Action Space

At each time step $r$, the agent builds an action vector $V_a^{(r)}$ that contains the numbers of each container type to be cached after the current time slot. Since this vector naturally becomes the succeeding state, this vector is also $(T \times N_E)$ two-dimensional.

Using the capacity constraint: $\sum_{t=t_1}^{t_m} (A^{(r)}(n)[t]) \leq S(n)$, the size of the action space can be reduced by explicitly removing the invalid caching decisions that exceeds
the node capacity $S(n)$.

5.1.3 Reward

After each time slot $r$, the agent receives a reward $Reward^{(r)}$ for each function request type $R_t^{(r)}$ generated at $r$.

The reward function should be defined to encourage the agent to increase the number of warm start cases, considering the propagation and instantiation delays. For each request $R_t^{(r)}$, a reward $Reward^{(r)}(R_t^{(r)})$ is defined as follows.

$$Reward^{(r)}(R_t^{(r)}) = \begin{cases} U & \text{if warm start at entry node} \\ \frac{1}{d_p(R_t^{(r)}) + d_c(R_t^{(r)})} & \text{else} \end{cases}$$

(4)

Here, the value $U$ can be chosen from any numbers larger than 1.0 since the tight upper bound for the other conditional case is 1.0. The reward decreases as either delay term(s), the propagation delay $d_p(R_t^{(r)})$ or the container instantiation delay $d_c(R_t^{(r)})$, increases. The total reward $Reward^{(r)}$ at time step $r$ is

$$Reward^{(r)} = \sum_{\{R_t^{(r)}\}} Reward^{(r)}(R_t^{(r)}).$$

(5)

When a request is served by a warm start process, the instantiation delay $d_c(R_t^{(r)})$ will be significantly smaller than the one with a cold start.

5.2 XRL framework for DQN

Structural Causal Model (SCM), is one of the policy summarization techniques which uses a causal graph to explain the causal relationships between variables. Nodes in the causal graph represent the variables, and the edges represent the relationship between variables. Action Influence Model (AIM) [12], an extension of SCM, explains the behaviors of an RL agent in terms of the effects that its actions cause. AIM generates an explanation to answer questions like “why action A was taken” and also
counterfactual questions like “why action B was NOT taken.” AIM generates a graph called *Action Influence Graph* (AIG) to explain the influence of actions.

To draw the graph, we first split the state and the successor state vector $V_s^{(r)}$ and $V_s^{(r+1)}$ into the respective state features. Each state feature represents a part of the state of a computing node at time step $r$. Note that the state and successor state can be sampled from a replay buffer of a DQN agent. Next, we calculate the cache hit rate of every computing node as the weighted average of the hit rate for each function type $t_1, t_2, ..., t_M \in T$ as follows:

$$\sum_{t=1}^{M} \min \left\{ 1.0, \frac{A^{(r)}(n) |t|}{\sum R^{(r)}_{t}} \right\}$$

Here, the min function enforces that the hit rate does not exceed 1.0 when there are caches available more than requests.

We also compute the p-values between state features to identify statistically significant pairs of features. Also, the p-values between rewards and state features are computed. A table containing all the p-values is called a p-value table.

State features that have higher statistical significance, when compared to the corresponding statistical significance threshold value of the p-value table, are selected and connected by edges that correspond to DRL actions. This forms an Action Influence Graph (AIG).
CHAPTER 6

Experiments and Results

This section showcases the performance of our XRL approach in a simple network topology, shown in Figure 3. We built a network simulator using Python and NetworkX.

6.1 Experiment Setting: Topology and Computing Nodes

The network topology has a setup of five edge nodes $N_E = (n_0, n_1, ..., n_4)$ connected by seven communication links. Among the five edge nodes, two of the nodes $n_0$ and $n_1$ are designated as entry nodes. We also assume that a cloud node $N_C$ is located at a centralized data center. For every computing node $n$ except for the cloud node, the maximum number of container caches $S(n)$ is set to 4.

6.2 Experiment Setting: Request Generation

At each time slot $r$, 30 requests of different function request types are generated at random by the edge devices $n_d \in N_D$, which are assumed to be connected to the two entry nodes. The generated requests are forwarded to the entry nodes $n_0$ and $n_1$ at random. For simplicity, we consider two function request types: $R_{0}(r)$ and $R_{1}(r)$ for this experimental setup. As the experimental graph has a small diameter, the TTL for a request, $max_{\text{hop}}$ is set to 1.

In this experiment, we intentionally create an extreme scenario where the generation probabilities for the two types of requests are extremely imbalanced. In particular, the generation probability for $Req\text{Type}0$ is set to 0.01 while the probability for $Req\text{Type}1$ is 0.99. With these extreme settings, obviously, the effective caching strategy is to have more type-1 containers cached with a few type-0 containers.

6.3 Experiment Setting: Reward

For the propagation delay metrics, this experiment assumes that a cold start takes ten more units of time than a warm start with caching. This assumption can be
Figure 3: A Logical Network Topology for an Experiment: $n_0$ and $n_1$ are entry nodes, solid links represent connections among edge nodes, and the dotted lines represent connections between edge nodes and a cloud node.

represented with $C = 10$ in eq. (1). Furthermore, the reward for the warm start at entry nodes is set to 2: $U = 2$ in eq. (4).

With the settings, there are four scenarios (and the corresponding reward values) possible for the requests in this experiment.

- (i) A request is processed at the entry node with the demanded type of container cache. The request gets a reward of 2.

- (ii) With a warm start, the reward is $\frac{1}{1+0} = 1$, where the propagation delay is 1 and the instantiation delay is 0.

- (iii) With a cold start, the reward is $\frac{1}{1+10} = \frac{1}{11}$, where the propagation delay
is 1 and the instantiation delay is 10.

- (iv) A request cannot be processed at neither an entry node nor an edge computing node and needs to be sent to the cloud node. The reward is \( \frac{1}{51} + 0 = \frac{1}{51} \). The propagation delay is 1 plus 50 from the first hop to the edge computing node and the second hop to the cloud node. The instantiation delay is 0 since the cloud node has a theoretically infinite capacity to cache all types of containers.

### 6.4 DQN Hyper-parameters

A DQN agent is trained with a learning rate of 0.01 with the epsilon-greedy exploration. For the exploration, \( \epsilon \) is initially set to 1.0 and decayed by 0.5 after every episode until it reaches the lower limit of 0.01. The training batch size is 64.

### 6.5 Performance on DRL-based Caching Decisions

This section presents the simulation results corresponding to Target 1 in the project’s objective.

Figure 4 shows the reward per episode that the DQN agent earned over the course of 400 episodes with each episode consisting of 10 time steps. During the exploration phase, the agent continues improving its caching policy to increase cumulative reward. This continuous improvement is shown as an overall upward trend of the average rewards (the orange line in the figure). Over the 400 episodes, the DQN agent earns an average cumulative reward of 198 per episode with the rewards ranging from a minimum of 46 to a maximum of 281. The theoretical upper bound for the cumulative reward per episode is 282. (See the Appendix Section for calculation of this upper bound.) The DQN agent achieving the near-upper bound reward indicates and reaffirms that DRL can effectively learn an optimal caching policy over time.

Figure 5 shows reward distribution among the four possible reward values averaged over every 250 time steps. The trend in this distribution indicates that the agent
learns an optimal caching decision that is to increase the requests served by warm
starts and decrease the use of cold starts. In particular, the upward trends of the blue
(Case i) and orange (Case ii) lines empirically proves the improvement in the policy.

6.6 Explanation of DRL-based Caching Decisions

This section presents the simulation results corresponding to Target 2 in the
project’s objective.

We extract explanations of the agent behaviors based on the Action Influence
Model (AIM). The AIM is constructed based on the replay buffer data collected
during the exploration phase. We create a causal network by following the procedures
detailed in Section 5.2. An AIG of the data gathered for 400 episodes is displayed in

Figure 4: Reward received by DQN Agent per episode.
Figure 5: A Reward Category Distribution: Over time, the RL actions encourage more warm starts, avoiding cloud usage.

Figure 6.

Overall, the AIG shows the potential interaction between the states. A close observation of the graph nodes $n_1$, $n_2$, $n_3$ and $reward$ shows that an increased hit rate at these nodes is causally linked to a significant reward. This is also theoretically correct because warm starts at these nodes provide the best reward value among the four cases.

The edges, $(n_1, reward),(n_2, reward)$, and $(n_3, reward)$ are associated with actions 5 and 3. This indicates that the these two actions result in an increased hit rate and significant reward. We use figure 7, a frequency graph of the actions taken by the DQN agent obtained during the training, a side by side comparison of figures 6 and 7 show that the actions learned by the DQN agent resulting in maximum
Figure 6: Causal Graph (AIG) generated based on DQN experiences - only statistically significant edges are shown.

The cumulative reward is action 3 and 5. This can be used as an evidence to corroborate the explanation provided by the AIM.

We also provide additional evidence to verify the explanation provided in figure 6 by comparing the impact of each action using a numerical analysis. For this, we consider two expected request arriving scenarios in our experiment which is derived from the request generation probability of $\text{ReqType0}$ and $\text{ReqType1}$. Since the request generation probability of $\text{ReqType0} = 0.01$ and $\text{ReqType1} = 0.99$, with 30 requests generated at a given time slot $r$ for the current experimental set up, we have, for each time step, the number of each type of request would become either (0 $\text{ReqType0}$, 30 $\text{ReqType1}$) or (1 $\text{ReqType0}$, 29 $\text{ReqType1}$) due to the extreme distribution. For the...
Figure 7: An action frequency graph generated for actions taken by DQN agent for 400 episodes

For the purpose of illustrating this calculation, we will provide cache hit rate calculations for only four out of the nine available actions, specifically including action 5 and action 3. The actions and their respective states considered for this example are:

\[
\text{action 5} = [0, 4]. \quad (7) \\
\text{action 3} = [1, 3]. \quad (8) \\
\text{action 2} = [2, 2]. \quad (9) \\
\text{action 1} = [3, 1]. \quad (10)
\]

The numerical analysis of cache hit rates for the aforementioned actions for the expected request-arriving scenarios is as follows:
Case 1: Number of requests for $\text{Reqtype}_0 = 0$ and $\text{ReqType}_1 = 30$ - The cache hit rates are 0.066, 0.055, 0.033 and 0.033 for actions 5, 3, 2 and 1 respectively.

Case 2: Number of requests for $\text{Reqtype}_0 = 1$ and $\text{ReqType}_1 = 29$ - The cache hit rates are 0.068, 0.55, 0.53 and 0.51 for actions 5, 3, 2 and 1 respectively.

By comparing hit rates given in case 1 and case 2, we can observe that action 5 and action 3 exhibit higher rates when compared to the other actions. When finding the significance of these hit rates with the reward obtained by the DQN agent, action 5 and 3 seem to have higher significance on the respective nodes than any other actions.

Also, the effectiveness of the two actions can be supported by the fact that the actions prepare more caches for more frequent request type ($\text{ReqType}_1$). Note that,
although this analysis seems very trivial in small cases, it is infeasible to compute the expected hit rates of all actions and decide the best action in practical scenarios with unknown request distribution and large action space.

We can also extract more detailed explanations from the AIG as shown in 8, to focus on a particular action as well as on top x% of statistically significant edges. Fig 8, shows top 80% statistically significant edges of action 5.

6.7 Explanation of Request Arriving Behaviors and Topological Importance of Nodes

In addition to identifying actions with a significant impact on positive rewards, the causal graph can infer the relative importance of nodes in terms of their topological location and the request arrival process.

In the causal graph, nodes $n_1$, $n_2$, and $n_3$ are *causally linked* to a significant reward (the reward node). Based on Figure 3, it is trivial that $n_2$ has more topological significance than others since the random redistribution process will bring more requests to it. However, it is nontrivial that $n_1$ and $n_3$ hold more significance than $n_0$ and $n_4$ since the graph is symmetric.

When we analyzed the actual request distribution in the experiments, slightly more requests (+0.46%) were assigned to the entry node $n_1$ compared to $n_0$ even though the generation process uses uniformly random assignment. Therefore, it seems that the causal graph captures the slight difference in the request arrival and identified that $n_1$ and $n_3$ are more important than the other side of the graph. This indicates the capability of the causal model to reveal the hidden patterns of actions that lead to a higher reward.
CHAPTER 7

Conclusion and Future Work

In this project, we have (i) reaffirmed the effectiveness of Deep Reinforcement Learning (DRL) for the container caching optimization problem and (ii) demonstrated the power of an Action Influence Graph (AIG) in explaining the caching policies learned by the DRL agent. Our DQN agent learns a caching policy that reduces the number of cold starts, which incur more delay than warm starts, by adjusting the distribution of container types cached in edge computing nodes. In particular, the agent learns to cache commonly used containers at topologically key locations. The explanations provided by the AIG are verified by theoretical and numerical analysis of the hit rates and request distributions. Based on the analysis, we have shown that the causal model-based explanations can improve the accountability of the autonomous caching control by DRL.

For future work, it is important to consider the scalability aspect for the system architecture as the number of edge devices and data generation continues to grow. Scalability consideration will also help in improving the request redistribution policy. Further, we can use the explanation provided by the AIM as a feedback to iteratively improve the decision making process of the DQN agent. Finally, AIM is one of the policy explanation approaches, we can augment this approach with other XRL approaches to provide a more comprehensive understanding about the DQN agent.
LIST OF REFERENCES


Appendix

A.1 An Upper Bound for the Caching Rewards

For a network with 5 nodes, each node with a maximum container caching capacity of 4 and 30 requests at each time step, the reward a DQN agent can get for an ideal scenario is calculated as follows:

- Requests reaching $n_0$ and $n_1$ (entry nodes): Each entry node can serve 4 requests per node. $(4 \times 2 = 8$ requests can be severed.) The corresponding reward is $8 \times d_p(R_t^{(r)}) = 8 \times 2 = 16$.

- Requests redistributed to $n_2$, $n_3$, and $n_4$: Each node can serve 4 requests per node. $(4 \times 3 = 12$ requests can be severed.) The corresponding reward is $12 \times d_p(R_t^{(r)}) = 12 \times 1 = 12$.

- Requests forwarded to the cloud: Since 30 requests arrive at each time step and 20 of them will be processed at either entry or edge computing nodes, the cloud should take care of the remaining 10 requests. The reward is $10 \times M = 10 \times \frac{1}{51} = 0.196$

Therefore, the upper bound for the total reward per time step is 28.196. Since we have 10 time steps per episode, the upper bound for the cumulative reward that a DQN agent can obtain per episode is 281.96.