Cost-Efficient Sharing Algorithms for DNN Model Serving in Mobile Edge Networks

Hao Dai, Jiashu Wu, Yang Wang, Jerome Yen, Yong Zhang, Chengzhong Xu *Fellow, IEEE*

Abstract—With the fast growth of mobile edge computing (MEC), the deep neural network (DNN) has gained more opportunities in application to various mobile services. Given the tremendous number of learning parameters and large model size, the DNN model is often trained in cloud center and then dispatched to end devices for inference via edge network. Therefore, maximizing the cost-efficiency of learned model dispatch in the edge network would be a critical problem for the model serving in various application contexts. To reach this goal, in this paper we focus mainly on reducing the total model dispatch cost in the edge network while maintaining the efficiency of the model inference. We first study this problem in its off-line form as a baseline where a sequence of n requests can be pre-defined in advance and exploit dynamic programming techniques to obtain a fast optimal algorithm in time complexity of $O(m^2n)$ under a semi-homogeneous cost model in a m-sized network. Then, we design and implement a 2.5-competitive algorithm for its online case with a provable lower bound of 2 for any deterministic online algorithm. We verify our results through careful algorithmic analysis and validate their actual performance via a trace-based study based on a public open international mobile network dataset.

Index Terms—Mobile Edge Computing, Online Algorithm, Cost Efficiency, Deep Neural Network, Model Sharing

1 Introduction

With the fast growth of smart devices and ubiquitous sensors, massive amounts of data are being generated in edge network [1]. Meanwhile, the exponential multiplication of data is also driving the rapid development of the *deep neural network* (DNN) model for wide uses in our daily lives [2]–[4]. However, if such a large amount of data were always shipped to cloud center for model processing, traditional cloud architecture would suffer from considerable challenges in communication, storage, and computation [5]. Much worse, many new types of applications (e.g., cooperative autonomous driving) have fairly strict latency requirements, which would generate an additional burden to the center. Therefore, an alternative computational paradigm—mobile edge computing (MEC) [6]–[8]—is advocated, which allows the deployment of the computation in proximity to user equipment (UE) for the time bounded responses [9]. With MEC, the DNN workloads can be (partially) pushed to the edge of the cloud network, which in turn could effectively mitigate the strict requirements on the communication, and consequently, fully unleash the potentials of edge computing for cost reduction [10], [11].

However, given the large number of model parameters, high model computational loads, and heterogeneous capacities of different computing platforms, fulfilling the requirements of the DNN model processing in the edge network is not a trivial matter [12]. Firstly, for the mobile device, due to its constrained resources [13] and finite amount of generated data, the model training, in general, cannot be fully deployed on it. Secondly, for the edge server, although installed close to users, its computational capacity is still relatively low, compared to the cloud, to train the DNN models for serving, not only in terms of efficiency but also in regards to coverage area. Therefore, fully deploying DNN model to share in the edge servers, in our opinion, is not always effective. Finally, for the cloud, it is usually far from mobile devices and incurs long time latency in its service path, which could compromise the quality of time-bounded services. As such, given these issues, it is not reasonable to place the entire DNN model computation in a single place. Instead, we are in favor of combining the advantages of both cloud and edge to collaboratively complete the DNN task.

To address the foregoing issues, at present, the mainstream of DNN deployment mode in the edge is so-called *In-Cloud Training and In-Edge Co-inference* (ICIE), which means the model training is accomplished in the cloud while the model co-inference is conducted between the edge and mobile device [10], [14]. This mode can work as part of the model serving process with some distinct merits. On the one hand, given large compute resources, the model can be continuously trained and updated in the cloud based on the data generated from edges and devices. On the other hand, the trained model can be dispatched to the MEC server, which facilitates its download to the mobile device for inference in two aspects: 1) the model cached in the MEC server can be effectively syned with the one in the cloud to ensure the correct inference made by the devices; 2) the

∗ *Yang Wang is the corresponding author*

- *Hao Dai, Jiashu Wu, Yang Wang, Yong Zhang are with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China, and with University of Chinese Academy of Sciences, Beijing, China. E-mail: {hao.dai, js.wu, yang.wang1, zhangyong}@siat.ac.cn*
- *Jerome Yen is with Department of Computer and Information Science, Faculty of Science and Technology, University of Macau, Macau SAR, China, E-mail: jeromeyen@um.edu.mo*
- *Chengzhong Xu is with State Key Lab of IoTSc, Department of Computer Science, University of Macau, Macau SAR, China, E-mail: czxu@um.edu.mo*

updated model can be cached, replicated, and transferred between the MEC servers in the edge network to support the model sharing between the mobile users in a wide area.

Although it is crucial to the exploitation of the full strength of DNN in a wide range of applications, the ICIE mode has not been fully exercised or studied in existing solutions where the mode is only used to dispatch the updated DNN directly from the cloud center to the mobile terminals without resorting to the edge network [15]–[17] or with the edge network, it is often in lack of theoretical analysis on the service cost-efficiency [18], [19].

In this paper, we intend to fill in void by studying from an algorithmic point of view: how cost efficient it is for the model to be dispatched from the cloud to multiple edge sites for inference, which is defined in our context as a *model sharing problem* in the edge network.

Superficially, the sharing problem described bears certain similarities to the Content Delivery Network (CDN) [20], [21]. However, they have essential differences in several aspects. First, the existing CDN solutions rarely take the interplay between the edge servers and the cloud center into account, as well also lack relevant theoretical analysis. Second, since the model performs long-term training continuously based on newly generated data, handling inplace updates is a crucial issue in the DNN model sharing problem, which is often missing in conventional CDN where the delivered contents are often static [22]. Since we usually only share and update the serialization of model parameters, the transmission cost of this part is generally homogenous, making this issue worth discussing separately from conventional cache problems. Last, the mainstream CDN algorithms, especially for those learning-based algorithms [9], [18], [21], require a lot of experience to train a scheduling model, resulting in bare compatibility with generalized real-world scenarios. Given these differences, the model sharing problem in our case is much more complicated and imposed great challenges, compared to CDN.

In this paper, we study the model sharing problem based on the ICIE deployment mode to maximize the cost-efficiency for the DNN model serving in a crowd of geographically dispersed mobile users. More specifically, we study the problem of sharing a trained model, pulled from the cloud, in an edge network by caching or transfer, with possible multiple copies, in a collection of cache servers in the edge so that the overall cost of the time-series requests to it is minimized. We investigate the sharing problem in both off-line and online forms based on an often-used *semi-homogeneous cost model* [23]—all pairs of cache nodes in the edge network have the same transfer cost, but each cache node has its own caching cost rate.[1] The rationale behind this model is that it is often adopted in the settings where the billing rates of the edge resources are partially fixed across its different edge servers in a region as studied in [6], [24].

The off-line form is defined to model the case that a stream of requests to a shared model can be predicted prior to the sharing of the model. This is particularly true when some model is accessed regularly among a set of network nodes. However, the off-line form is usually ideal. A more practical situation is that the request sequence is unpredictable. The online algorithm can cope well with this case by serving the incoming requests in a timely manner with minimum cost. In this paper, we first design a fast optimal off-line algorithm for the predictable case and then propose our 2.5-competitive online algorithm to tackle the unpredictability in the more practical case. We provably achieve these results with deep insights into the problem and careful analysis of the solution algorithms. Note that our algorithms are generic enough, it is not designed for any particular type of DNN model, thus is applicable to all kinds of DNN models as long as they can be stored in a data file (e.g., PMML [25] and PFA [26]).

1. Note that the cost can be a very general concept, it can refer to the time latency, monetary cost, or others, depending on how the cost is defined.

In summary, we made the following contributions in this paper:

- We present a dynamic programming-based optimal algorithm for the model sharing problem that can minimize the total transfer and caching costs within $O(m^2n)$ time for the off-line case, here m represents the number of nodes in the network, while n is the length of the request stream.
- Our online algorithm for this problem is designed by extending the anticipatory caching idea [27] whereby a 2.5-competitive ratio as well as its tightness are also obtained by giving a lower bound of the ratio as 2 for any deterministic online algorithm.
- With the proposed off-line and online algorithms, we extend the synchronizing mechanism to support the active model update from the cloud center, and show that the extended mechanism does not impact the theoretical bound of the algorithm.
- We validate our results through an intensive trace-based empirical study, whose results reveal our algorithms are cost-efficiently feasible and practical in reality.

The organization of the paper is as follows: we introduce some background knowledge and related work regarding the DNN model serving and the mobile edge network in Section 2. We describe the formulation and notation of the model sharing problem in Section 3 and propose an off-line algorithm and an online algorithm with their critical analysis in Section 4. We present the simulation studies to validate our findings in Section 5, followed by the conclusion of the paper in the last section.

2 Background Knowledge

In this section, we introduce some background knowledge to help understand our work, and then show the motivation of our work.

2.1 DNN Model and Its Serving

A deep neural network (DNN), as one of the most cutting-edge machine learning techniques, is typically composed of multiple layers between the input and output layers. It is often used to mimic the workings of the human brain in processing data for successful use in many applications, including object detection, speech recognition, language translation, and decision making. A DNN model is fully characterized by its layer's components, in particular, the weight parameters.

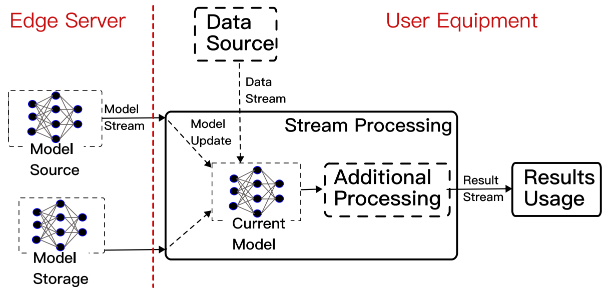


Fig. 1. The architecture of model serving. The model source and storage, allocated on edge servers, are requested by the UE, which leverages the current model to make inferences based on the data from the data source. The model on the edge servers is periodically synchronized with the one updated in the cloud.

In general, the DNN model applications consist of two phases: *model training* and *model serving*. The model training in general is a procedure in deep learning that creates a designed model by estimating its parameters from a large number of training samples and in our particular case it typically learns the weights for each pair of connected neurons using some iterative methods [10].

In practice, the trained model is often recorded in standardized document-based intermediate representation (i.e., PMML [25] and PFA [26]) and then used to serve in data-processing context by following a certain model serving framework as shown in Fig.1. The inputs of the serving framework are from two data streams: one containing the data that needs to be scored, and the other containing the model updates, which are either from the model data blob itself or from the reference to the model data in a database or a file system. The stream engine uses the current model for the actual scoring in memory and delivers the scoring results as inference outputs to its users. Since it is represented as data rather than code, the DNN model in general and its weight parameters in particular can be manipulated as a special type of data, which is fundamental for our proposed algorithms.

2.2 Mobile Edge Network

The mobile edge network we considered is part of the cellular network infrastructure as shown in Fig.2 where the *Radio Access Network* (RAN) covers a wide geographical area, which is divided into a number of cells, each taking a base station (BS) as a fixed access point to cover its mobile devices by translating the radio signals into data packets, which are then routed through the wired mobile backbone network (CN) to external packet data networks (e.g., cloud data centers) via packet data network gateway (P-GW). The inter-connected BSs are typically connected to a *Mobility Management Entity* (MME) and a *Serving Gateway* (S-GW) via different technologies. The MME is used to handle the control information, including mobility management and authentication functions, for the mobile terminals while the S-GW is deployed to process the data information, such as data packet routing/forwarding and handover management.

The mobile edge is created on the edge of the networking infrastructure, where a number of MEC servers (also called *edge server* or *server* thereafter) are deployed in close proximity of the BSs, each being physically attached to one edge

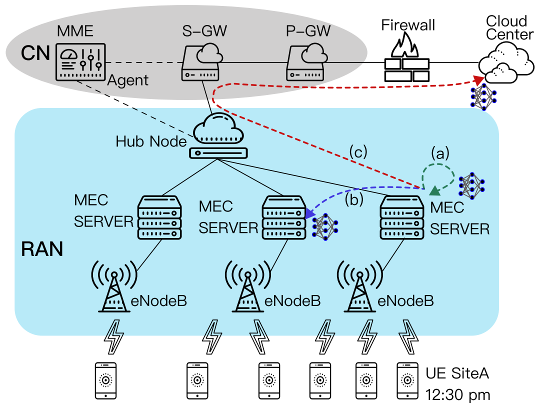


Fig. 2. Mobile edge network where MEC servers are integrated with RAN to perform edge computing. The inter-connected BSs are connected to a *Mobility Management Entity* (MME) and a *Serving Gateway* (S-GW) via different technologies (e.g., microwave or landlines).

server that is inter-connected with others via a *Hub Node* as shown in Fig. 2.

2.3 DNN Serving in Edge Network

The mobile-edge based infrastructure is aligned well with the model serving framework. One can train the DNN model in the cloud center and dispatch the trained model to the edge servers, which are capable of manipulating it directly at the edge network in terms of caching, replication and transfer with the minimum service cost to serve the stream engine in each mobile device. Note that this serving process is not a one-off, rather, it is repeated when the model is updated in the cloud to adapt to the new settings [28]. As such, the models cached in the edge network, also in the mobile device, need to be synced with the one in the cloud for the correctness of the inference.

Overall, the mobile edge can take the place of the cloud to some extent to facilitate latency-sensitive services. Additionally, the mobile edge can also interplay with the cloud to deliver cost effective, ubiquitous and scalable mobile services for inference applications. Consequently, the success of this computing mode is largely dependent on the cost efficiency of sharing the model among multiple edge sites for inference.

3 Model Sharing Problem

In this section, we formulate the model sharing problem with respect to the mobile edge network described in Fig. 2. To this end, we first model the edge network into a system model whereby the sharing problem is formulated based on a defined cost model, and its complexity is also analyzed. For quick reference, we summarize the frequently used symbols in *Tab.*1 in Appendix.

3.1 System Model

As shown in Fig. 2, a mobile edge network is composed of m networked edge servers $S = \{s^1, \ldots, s^{\mathbb{M}}\}$ that can collaborate with cloud center s^0 to carry out well defined computational tasks to serve a large number of mobile devices, connecting to the eNodeB of a selected edge server.

In particular for the DNN model serving, according to the ICIE deployment, the model training is conducted in the cloud by periodically using the newly generated data, while the model inference is accomplished by mobile devices. To this end, the edge server in this deployment often acts as a dual role. On the one hand, it communicates with the cloud to download (pull) the updated model, and on the other hand, it works as a relay *cache server*, which is used to cache and transfer the model inside the network for its cost-efficient dispatch on demands to mobile devices. The mobile device asks for the updated model by sending syn-requests to a specific edge server in its own period.

As the billing rates of edge server vendors are currently fixed in terms of network transmission[2], in this paper, we regard the transfer cost as a constant concerning the data size of the DNN model. Therefore, in some realistic settings, we can simplify the cost model to be semi-homogeneous, in which the transfer cost and the pull cost are denoted as λ and β, respectively. Furthermore, since deletion is instantaneous on the edge server, we can assume that the deletion costs are trivial and can be ignored in the system model.

Based on the defined cost model, we can explicitly define four operations as well as their associate costs for the model sharing in the edge network as follows:

- *Caching*: ${}_{\theta}s^m \overset{\mu_m \delta t_{i,j}}{\Longrightarrow} {}_{\theta}s^m, \delta t_{i,j} = t_j - t_i$, caching model θ from s^m from t_i to t_j.
- *Transfer*: ${}_{\theta}s^q \overset{\lambda}{\Longrightarrow} {}_{\theta}s^m$, transferring model θ from server s^q to server s^m while keeping the model at s^q.
- *Pull*: ${}_{\theta}s^0 \overset{\beta}{\Longrightarrow} {}_{\theta}s^m$, pulling model θ from cloud center s^0 to server s^m.
- *Deletion:* ${}_{\theta}s^m \overset{0}{\Longrightarrow} s^m$, removing model θ at s^m.

As thus, the edge servers in the network can receive n random syn-requests, each being made at any time instance, denoted as $\mathcal{R} = \{\boldsymbol{r}_0, \ldots, \boldsymbol{r}_n\}$, where request $\boldsymbol{r}_i = (e_i, t_i)$, $e_i \in S$, represents that $\boldsymbol{r}_i$ is made from server e_i at time t_i. Thus, for a request sequence as shown by black dots in Fig.3, the four operations are combined to serve it: a) caching the model on edge server (black line); b) transferring the model to other edge servers (blue line); c) pulling the updated model from the cloud (red line); d) deleting the model from edge servers (vertices at the end of each horizontal Line).

3.2 Problem Formulation

With the system model, we can further describe how to formulate the sharing problem with an attempt to reduce the sharing cost through scheduling in the sequel.

3.2.1 Schedule Model

As stated above, our target is to combine a set of defined operations for each server to manipulate the model along the time line so that all the requests are satisfied with minimal cost. We name such a set as a *schedule*, which is defined as follows:

Definition 1 (Schedule). A *schedule* $\mathcal{P}$ is any set of caches, transfers, and pulls satisfying: 1) the synchronization service is available for $\boldsymbol{r}_i = (e_i, t_i), 0 \leq i \leq n$; 2) The transferring and caching only happen when there is at least one edge server caching the updated model and running the synchronization service at any time instance. Otherwise, a pulling is incurred.

2. Say, the cost model from Bell Network charges \$0.3078 for transferring $1GB$ over its OC3 link [29]

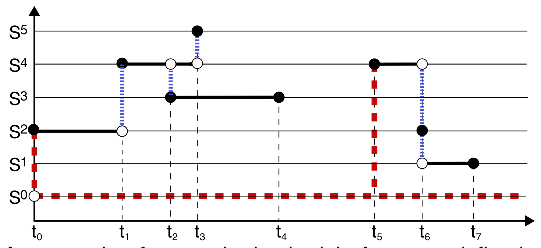


Fig. 3. An example of a standard schedule for a pre-defined request sequence. The black dots represent requests, and the black lines represent caching that end on request, while the blue and red lines represent transferring and pulling, respectively.

By following the method in [30], we also describe the schedule using a *space-time diagram*, where the edges are caching intervals, transferring, or pulling, and the vertices are requests, endpoints of either transferring or pulling. More formally, we have

Definition 2 (Space-Time Graph). We define a space-time graph as a weighted directed graph $G = (V, E, W)$. The vertex set is denoted by $V = \{v_{mi} \mid 0 \leq m \leq \mathbb{M}, 0 \leq i \leq n\}$, where vertex $v \in \{v_{mi} \mid \boldsymbol{r}_i \in R, e_i = s^m\}$ corresponds to request $\boldsymbol{r}_i$ made on edge server s^m at time t_i, vertex $v \in \{v_{mi} \mid \boldsymbol{r}_i \notin \mathcal{R}, e_i = s^m\}$ denotes transfer vertex and $v \in \{v_{0i} \mid 0 \leq i \leq n\}$ represents pull vertex. The edge set E consists of three subsets:

1) a set of *cache edges* $E_C = \{(v_{mj}, v_{mj}) \mid 0 \leq i < j \leq n, 1 \leq m \leq \mathbb{M}\}$;
2) a set of *transfer edges* $E_T = \{(v_{qi}, v_{mi}), (v_{mi}, v_{qi}) \mid 0 \leq i \leq n, m \neq q$, and $s^q, s^m \in S\}$, and
3) a set of *pull edges* $E_P = \{(v_{0i}, v_{mi}) \mid 0 \leq i \leq n, 1 \leq m \leq \mathbb{M}\}$.

Combined with the cost model, the edge weights W can be defined as $W(e) = \lambda$ for edges $e \in E_T$, $W(e) = \mu_m(t_j - t_i)$ for edges $(v_{mi}, v_{mj}) \in E_C$, and $W(e) = \beta$ for edges $e \in E_P$.

Based on the defined space-time graph, we can make an observation by following the same arguments in [30] that each schedule instance has a standard form, which is defined as follows:

Observation 1 (Standard Form). For any instance of the graph, there exists at least one optimal schedule in which every transfer and pull occurs at a request time t_i with its output ends on edge server e_i.

Fig. 3 shows a standard form schedule in a space-time diagram, where all the transfers (white dots) and pulls (red dots) connecting with the requests (black dots) at different edge servers. By this example, one can see that the DNN model sharing problem is equivalent to finding a standard-form tree in its off-line or online fashion to serve all the requests made along timeline at the minimum cost on space-time graph G.

3.2.2 Formulation and Complexity

Given a schedule $\mathcal{P}$ in standard form, we can regard its cost as the sum of the edge weights of the standard form tree. According to the definition of the space-time graph, the cost of a scheduling $\mathcal{P} = \{E_C \cup E_T \cup E_P\}$ can be denoted as:

$$\mathcal{C}(\mathcal{P}) = \sum_{(v_{mi},v_{mj})\in E_C} \mu_m(t_j - t_i) + \sum_{(v_{mi},v_{qi})\in E_T} \lambda + \sum_{(v_{0i},v_{mi})\in E_P} \beta \tag{1}$$

Note that there could be many feasible schedules for request sequence $\mathcal{R}$, we use Γ to represent the feasible schedules for up to $\boldsymbol{r}_i$. The goal of our problem is to find an optimal schedule $\mathcal{P}^*$, which can be formalized as follows:

$$\mathcal{P}^* = \arg\min_{\mathcal{P}\in\Gamma}\{\mathcal{C}(\mathcal{P})\} \tag{2}$$

This problem is solvable in polynomial time, and we will give a polynomial optimal algorithm in the next section. However, its general form with heterogeneous cost model is a variant of the rectilinear Steiner tree problem [31] and the rectilinear Steiner arborescence problem [32], both of them are NP-complete. Thus, it is believed that the model sharing problem is still NP-complete, but its formal proof is still open [33]. Moreover, due to the lack of prior knowledge about the request sequence in the online form, designing a reliable online algorithm with the heterogeneous cost model is also barely achievable.

Notably, the proposed model is generic enough to adapt to other appropriate machine learning models than DNN.

4 Sharing Algorithms

Given the problem definition, in this section we investigate the sharing algorithm for both off-line and online cases. The off-line algorithm targets the scenario in which the request sequence is highly predictable from history and can be available in advance while in the online case the requests are usually not predictable and the algorithm for this case is more realistic.

4.1 An Optimal off-line Algorithm

Firstly, given the standard form of schedules, we can define sub-schedule as follows:

Definition 3 (Sub-schedule). The *sub-schedule* $\mathcal{P}^{(j)}$ of $\mathcal{P}$ is a schedule for $\boldsymbol{r}_j$ that consists of the set of caching intervals, transferring and pulling from $\mathcal{P}$ required to satisfy all requests $\boldsymbol{r}_0, \ldots, \boldsymbol{r}_j$.

Note that the sub-schedule $\mathcal{P}^{(j)}$ of the optimal schedule $\mathcal{P}$ may not be an optimal schedule for $\{\boldsymbol{r}_0 \ldots \boldsymbol{r}_j\}$, and it may not be unique.

Based on the concepts presented above, we can further make an analysis of this problem and then derive our optimal algorithm. To this end, we first obtain a lower bound on the marginal costs to satisfy each individual request, which is defined by

Definition 4 (Marginal Cost Bound). The marginal cost bound of request $\boldsymbol{r}_i$ on s^m is $b_i = \min\{\alpha\lambda + (1-\alpha)\beta, \mu_j\delta_{p(i),i}\}, 1 \leq i \leq n$, here, $\alpha = 1$ or 0, depending on whether or not there is an updated model kept in the group of edge servers.

Given the marginal cost bound, we further have a lower bound on the total costs to satisfy a request sequence, which is defined by

Definition 5 (Running Bound). The running bound of the marginal costs up to request i is $B_i = \sum_{j=1}^{i} b_j$.

As a result, for a segment of the request sequence from $\boldsymbol{r}_j$ to $\boldsymbol{r}_i$, its running bound of the marginal costs can be computed as $B_i - B_j$, denoted by B_i^j in the sequel.

Definition 6 (Optimal Cost $C(i)$). We define $C(i), 0 \leq i \leq n$, is the cost of the optimal schedule S^*. When $t = t_0$, it is necessary to pull the model from the cloud to serve $\boldsymbol{r}_0$, so the cost of $\boldsymbol{r}_0$ must be β, i.e., $C(0) = \beta$.

Our goal is to create a recurrence for $C(i)$ that we can solve dynamically. To this end, by analyzing the standard form of the schedule, we can immediately derive the optimality of the trivial case when the last request is served by pulling from s^0(cloud center).

Lemma 1. If $\mathcal{P}^*$ is an optimal schedule in which the last operation is a pulling, then $\mathcal{P}^{(i-1)}$ is an optimal schedule up to request $\boldsymbol{r}_{i-1}$ (i.e.,$\mathcal{P}^{(i-1)} \subset \mathcal{P}^*$), and we have $C(i) = C(i-1) + \beta$.

Proof: If the optimal $\mathcal{P}^*$ ends in a pulling, we can directly derive the optimality of $\mathcal{P}^{(i-1)}$ since in this case caching cost is more expensive than β. □

We now consider the other two non-trivial cases, that is, $\boldsymbol{r}_i$ is served either by transferring or caching. In these cases, the last transferring or caching involved to serve $\boldsymbol{r}_i$ may impact all the requests made in the caching interval since a caching is extended from its starting point to t_i which allows the requests to re-adjust the sources of the model (e.g., a caching may be changed to transferring for cost reduction). As a consequence, no request $\boldsymbol{r}_j, 0 < j < i$ is guaranteed to be optimal for the sub-schedule of $\mathcal{P}^{(i)}$ with respect to the interval $[t_1, t_{i-1}]$. To deal with this, we define two auxiliary recurrences that help compute $C(i)$.

Definition 7 (Semi-Optimal Cost $T(i)$ and $D(i)$). We define $T(i)$ and $D(i)$ to be the semi-optimal cost of a schedule $\mathcal{P}^{(i)}$ that $\boldsymbol{r}_i$ is served by transferring and caching on edge server e_i, respectively. Clearly, $C(i) \leq T(i)$ and $C(i) \leq D(i)$.

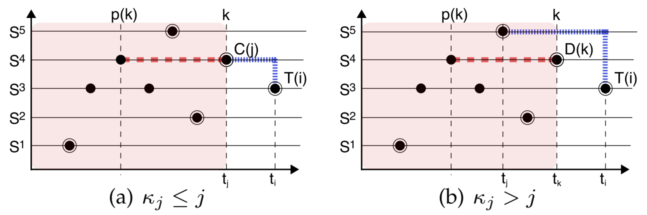
The basic idea of auxiliary recurrence is to establish the relationships between $C(i)$, certain $T(j)$ and certain $D(j)$ that has been available, or vice versa, whereby the most recent $C(i)$ can be computed. To this end, we also define the following concepts in our algorithm design.

Definition 8 (Feeding Set). For each request $\boldsymbol{r}_i$, we define its feeding set as $\mathcal{F}(i) = \{\boldsymbol{r}_j | e_j \neq e_i, t_j < t_i, \text{and } \boldsymbol{r}_j \text{ is the most recent request on } e_j\}$.

That is, $\mathcal{F}(i)$ is composed of the most recent requests on each edge server except e_i. $\mathcal{F}(i)$ designates the set of candidate models that could be used to satisfy $\boldsymbol{r}_i$ via a transfer in the optimal schedule. Given $\mathcal{F}(i)$, we further define *cover index set* and *pivot index* for each element in $\mathcal{F}(i)$ as follows,

Definition 9 (Cover Index Set). We define the cover index set $\pi_j(i)$ with respect to each $\boldsymbol{r}_j \in \mathcal{F}(i)$ as

$$\pi_j(i) = \{k | H(e_k, t_{p_k}, t_k), p_k < j \leq k < i\} \tag{3}$$



(a) $\kappa_j \le j$ (b) $\kappa_j > j$

Fig. 4. The examples of the case when $\kappa_j \le j$ and $\kappa_j > j$. The final caching $H(e_i, t_{p(i)}, t_i)$ impacts the serving path (shown in bold blue line) of the requests $[t_{p(i)}, t_{i-1}]$ in both cases. The cycled vertices represent the requests in $\mathcal{F}(i)$.

here, $H(e_k, t_{p_k}, t_k)$ represents a caching from t_{p_k} to t_k on server e_k.

Definition 10 (Pivot Index). The pivot index κ_j for $\boldsymbol{r}_j \in \mathcal{F}(i)$ is defined by either 0 or the maximum in $\pi_j(i)$, depending on whether or not $\pi_j(i) = \varnothing$, i.e.,

$$\kappa_j = \begin{cases} \max\{\pi_j(i)\} & \pi_j(i) \neq \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The definition of $\kappa_j \neq 0$ is important as it signifies the last request in $[t_j, t_{i-1}]$ that is served by the caching $H(e_{\kappa_j}, t_{p(\kappa_j)}, t_{\kappa_j})$ other than the transfer from $H(e_i, t_{p_i}, t_i)$, which forms the basis for the $C(i)$ recurrences. We distinguish two cases: 1) $\kappa_j \le j$, and 2) $\kappa_j > j, \boldsymbol{r}_j \in \mathcal{F}(i)$. The first is the boundary case, as illustrated in Fig. 4(a), which is trivial.

Lemma 2. For the pivot index $\kappa_j, \boldsymbol{r}_j \in \mathcal{F}(i)$ as defined in Definition 10, if $\kappa_j \le j$ then the optimal *restricted* cost $T(i) = C(j) + \min(\mu_j, \mu_i)\delta_{j,i} + \lambda + B^{j}_{i-1}$.

Proof: $C(j)$ is the minimal cost of satisfying all requests up to t_j. The cost of serving $\boldsymbol{r}_j$ at t_j is $\min(\mu_j, \mu_i)\delta_{j,i} + \lambda$, and with the caching involved we can satisfy all requests $\{\boldsymbol{r}_k \mid j < k < i\}$ by transferring and short caching intervals with a cost of B^{j}_{i-1}. Since the cost of this path is a lower bound of serving these requests, the total cost is optimal under the stated conditions of the lemma. □

Now let's examine the case that $\kappa_j > j$. In this case, both $H(e_{\kappa_j}, t_{p(\kappa_j)}, t_{\kappa_j})$ and $H(e_i, t_j, t_i)$ are in the final schedule as shown in Fig. 4(b), then we have

Lemma 3. For κ_j as defined in Definition 10, if $\kappa_j \neq 0$ then the optimal *restricted* cost

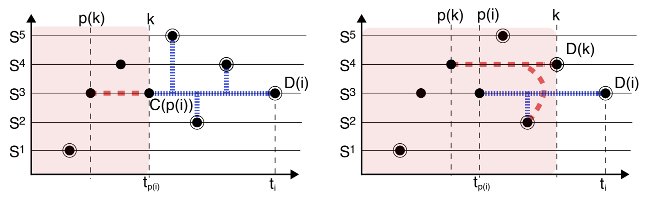
$$T(i) = D(\kappa_j) + \min(\mu_j, \mu_i)\delta_{j,i} + \lambda + B^{\kappa_j}_{i-1} \tag{5}$$

Proof: We can construct a schedule up to $\boldsymbol{r}_{\kappa_j}$ that ends up with a caching. Since $D(\kappa_j)$ is a lower bound on the cost of this schedule, we have $D(\kappa_j) \le \mathcal{C}(\mathcal{P}^{(\kappa_j)})$. Since $B^{\kappa_j}_{i-1}$ is a lower bound on adding the requests between t_{κ_j} and t_{i-1}, and we must add $\min(\mu_j, \mu_i)\delta_{j,i}$ to cover the interval $[t_j, t_i]$. Then, we see that $D(\kappa_j) + \min(\mu_j, \mu_i)\delta_{j,i} + B^{\kappa_j}_{i-1} \le C(i)$.

If we start with a restricted optimal schedule to $\boldsymbol{r}_{\kappa_j}$ with cost $D(\kappa_j)$, then we can similarly construct a restricted schedule with a transferring from t_j to t_i to serve $\boldsymbol{r}_i$ at the cost of $D(\kappa_j) + \min(\mu_j, \mu_i)\delta_{j,i} + \lambda + B^{\kappa_j}_{i-1}$, which is greater than $C(i)$, and then conclude the lemma. □

By combining these lemmas, we enumerate all the request indexes on the interval $[t_{p(i)}, t_{i-1}]$ to derive $T(i)$ recurrence as follows:

$$T(i) = \begin{cases} +\infty & -m \le i \le 0 \\ \lambda + \min\limits_{r_j \in \mathcal{F}(i)} \begin{cases} C(j) + \min(\mu_j, \mu_i)\delta_{j,i} + B^{j}_{i-1} \\ \min\limits_{k \in \pi_j(i)} \{D(k) + \min(\mu_j, \mu_i)\delta_{j,i} + B^{k}_{i-1}\} \end{cases} \end{cases} \tag{6}$$



(a) $\kappa_{p(i)} \le p(i)$ (b) $\kappa_{p(i)} > p(i)$

Fig. 5. The examples of the trivial case when $\kappa_{p(i)} \le p(i)$ and the non-trivial case when $\kappa_{p(i)} > p(i)(\pi(i) \neq \varnothing)$. The final caching $H(e_i, t_{p(i)}, t_i)$ impacts the serving path (shown in bold blue line) of the requests $[t_{p(i)}, t_{i-1}]$ in both cases.

Now we establish the relationships between $D(i)$ and certain $C(\kappa_j)$ that have been available. To reach this goal, we first define $\pi_{p(i)}(i)$ and $\kappa_{p(i)}$ concerning $\boldsymbol{r}_{p(i)}$, as special cases of Definition 9 and 10. Afterward, as with the case in computing the $T(i)$ recurrences, we also distinguish two cases: 1) $\kappa_{p(i)} \le p(i)$, and 2) $\kappa_{p(i)} > p(i)$. As the same, the first case is the trivial boundary case, and an illustrative example of the trivial case is shown in Fig.5(a).

Lemma 4. For the pivot index $\kappa_{p(i)} \le p(i)$, the optimal restricted cost $D(i) = C(p(i)) + \mu_i\delta_{p(i),i} + B^{p(i)}_{i-1}$.

Proof: We can conclude this lemma by following similar arguments in the proof of Lemma 2. □

Now we examine the non-trivial case that $\kappa_{p_i} > p_i$. In this case, both $H(e_{\kappa_{p(i)}}, t_{p(\kappa_{p(i)})}, t_{\kappa_{p(i)}})$ and $H(e_i, t_{p(i)}, t_i)$ are in the final schedule, as illustrated in Fig.5(b). Then we have

Lemma 5. For any $\kappa_{p(i)} \neq 0$, the optimal restricted cost $D(i) = D(\kappa_{p(i)}) + \mu_i\delta_{p(i),i} + B^{\kappa_{p(i)}}_{i-1}$.

Proof: We can conclude this lemma by following the arguments in Lemma 3. □

By combining the two lemmas above, we enumerate all the request indexes in interval $[t_{p(i)}, t_{i-1}]$ to derive $D(i)$ recurrence:

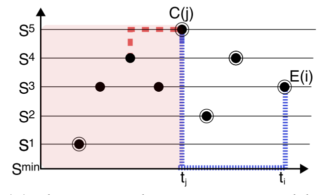
$$D(i) = \begin{cases} +\infty & -m \le i \le 0 \\ \min \begin{cases} C(p(i)) + \mu_i\delta_{p(i),i} + B^{p(i)}_{i-1} & 1 \le i \le n \\ \min\limits_{j \in \pi_{p(i)}(i)} \{D(j) + \mu_i\delta_{p(i),i} + B^{j}_{i-1}\} \end{cases} \end{cases} \tag{7}$$

Since unit cost μ_i is heterogeneous on different edge servers, we find a particular case that the request is served by the path transferring twice. Specifically, we let s^{min} denote the edge server with the minimum unit cost. The specific route is to transfer the model from e_j to s^{min} at first and then transfer it to e_i at the time of t_i to serve $\boldsymbol{r}_i$, as shown in Fig.6(a). For convenience, we call such routes "double-transferring."

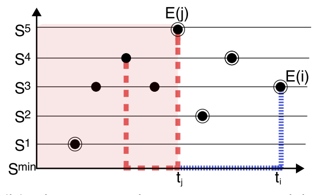
For double-transferring, we can define Semi-Optimal Cost $E(i)$ for it as well, and establish its relationship with the $C(i)$, $T(i)$ and $D(i)$ by the following lemma.

Lemma 6. For some $\boldsymbol{r}_j \in \mathcal{F}(i), e_i = s^{min}$ and $e_j \neq s^{min}$, the optimal restricted cost $E(i) = \min(T(j), D(j)) + 2\lambda + \mu_{min}\delta_{j,i} + B^{j}_{i-1}$.

Proof: Since μ_i is heterogeneous, we can construct a case of $\mu_j\delta_{j,i} > \lambda + \mu_{min}\delta_{j,i}$. In this case, we have $C(j) + 2\lambda + \mu_{min}\delta_{j,i} < C(j) + \lambda + \mu_i\delta_{j,i}$. Therefore, $T(i)$ is greater than $E(i)$. Similarly, we can prove that $C(j) + 2\lambda + \mu_{min}\delta_{j,i} < C(j) + \mu_i\delta_{j,i}$, $D(i)$ is greater than $E(i)$. □



(a) the case when $\boldsymbol{r}_j$ is served by transferring.



(b) the case when $\boldsymbol{r}_j$ is served by double-transferring.

Fig. 6. The examples of the cases when $\boldsymbol{r}_j$ is served by transferring or caching (a) and served by double-transferring (b), there are specific paths to serve $\boldsymbol{r}_i$. In case (a), e_j transfers the model to s^{min} at first, and then s^{min} caches it for a period at the cost of $H(s^{min}, t_j, t_i)$ before being transferred to e_i. In case (b), s^{min} keeps caching for a period at the cost of $H(s^{min}, t_j, t_i)$ and then transfers model to e_i.

$E(i)$ can be directly superimposed on the new path based on $D(j)$ or $T(j)$ when $e_i \neq s^{min}$ and $e_j \neq s^{min}$. Besides, there is another special case when $\boldsymbol{r}_j$ is served by $E(j)$, as shown in Fig.6(b). In this case, $E(i) = E(j) + \lambda + \mu_{min}\delta_{j,i} + B^j_{i-1}$.

Given these analyses, we can complete the recurrence for E(i) as follows,

$$E(i) = \begin{cases} +\infty & -m \leq i \leq 0 \\ \min\limits_{r_j \in \mathcal{F}(i)} \begin{cases} min(T(j), D(j)) + 2\lambda + \mu_{min}\delta_{j,i} \\ \quad +B^j_{i-1} \qquad e_j \neq s^{min}, e_i \neq s^{min} \\ E(j) + \lambda + \mu_{min}\delta_{j,i} + B^j_{i-1} \\ \qquad\qquad e_j \neq s^{min}, e_i \neq s^{min} \\ C(j) + \lambda + \mu_{min}\delta_{j,i} + B^j_{i-1} \\ \qquad\qquad e_j = s^{min} \\ C(j) + \mu_{min}\delta_{j,i} + B^j_{i-1} \\ \qquad\qquad e_i = s^{min} \end{cases} \end{cases} \tag{8}$$

Given these considerations, we can complete the recurrence for $C(i)$ in terms of the $T(i)$, $D(i)$ and $E(i)$ as follows,

$$C(i) = \begin{cases} \beta & i = 0 \\ \min \begin{cases} T(i) \\ D(i) \\ E(i) \\ C(i-1) + \beta \end{cases} & 1 \leq i \leq n \end{cases} \tag{9}$$

After considerable analysis, we have the following theorem for our proposed algorithm.

Theorem 1. Given the semi-homogeneous cost model, the recurrence algorithm can correctly compute the minimum cost of the off-line DNN model sharing problem, with the time complexity of $O(m^2 n)$.

Proof: The correctness proof can be directly obtained by combining Lemma 2 to Lemma 6. And during the next pass over the requests to compute the recurrences, these pointers can be used to precisely identify each of the intervals required by Eq. (6)–(9) in $O(m^2)$, $O(m)$, $O(m)$ and $O(1)$ time, respectively, on the per-request basis, thus taking at most $O(m^2 n)$ time for n requests. □

4.2 2.5-Competitive Online Algorithm

In this section, we give a 2.5-competitive algorithm for the online version of this problem. The basic idea is to serve the next request by keeping an updated model on server e_j for a period of time. As the same in the off-line case, each request can be served by three methods: caching, transferring, and pulling. Besides, we can decompose the problem into each edge server to accomplish the global solution. Since there are two different cost variables λ and β, without loss of generality, we will discuss and design the online algorithm in three cases: 1) $\beta \leq \lambda$; 2) $\lambda < \beta \leq 2\lambda$; and 3) $\beta > 2\lambda$.

4.2.1 *Case 1: $\beta \leq \lambda$*

In this case, the request won't be served by transferring. Instead, the optimal solution will consist of only caching and pulling. Therefore, we merely need to make a trade-off between caching and pulling costs. When the cost of caching is greater than pulling, the algorithm will retrieve the model from the cloud to the edge server.

Algorithm 1: OTSharing-v1 algorithm

```
Input: $t$ : current time ;
       $\vec{t_p}$ : array of the last visit time for each server.
Output: $\vec{t_{p'}}$ : updated array of last visit times.
for $t_{pi} \leftarrow \vec{t_p}$ do
    $\Delta t_i \leftarrow t_i - t_{pi}$;
    if request $i$ takes place on server $i$ then
        if $\Delta t_i > \frac{\beta}{\mu_i}$ then
            serving request $i$ through pulling;
        end
        if $\Delta t_i \leq \frac{\beta}{\mu_i}$ then
            serving request $i$ through caching;
        end
        Set active time $\frac{\beta}{\mu_i}$ for the updated model on $e_i$,
          and it should be deleted when active time is
          expired ;
        $t_{pi'} \leftarrow t_i$ ;
    end
end
```

Theorem 2. Algorithm1 is 2-competitive.

Proof:

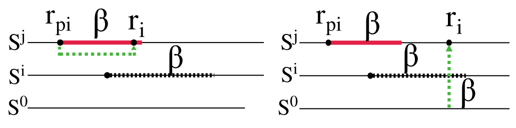


Fig. 7. In case (a), the request is served by caching; in case (b), the request is served by pulling.

a) If $\Delta t \leq \frac{\beta}{\mu_i}$, request $\boldsymbol{r}_i$ is served by caching in both the optimal solution (represented by *Optimal* in the sequel) and *Algorithm1*, as illustrated in Fig. 7(a). The competitive ratio (denoted by *c.r.* in the sequel) is

$$c.r.(a) = \frac{ALG}{OPT} = \frac{\mu_i \Delta t}{\mu_i \Delta t} = 1 \tag{10}$$

b) If $\Delta t > \frac{\beta}{\mu_i}$, request $\boldsymbol{r}_i$ is served by pulling with a cost β in both *Optimal* and *Algorithm1*. After that, edge server e_i will cache the updated model with an extra cost β (Fig. 7(b)). Thus, the competitive ratio is

$$c.r.(b) = \frac{ALG}{OPT} = \frac{\beta + \beta}{\beta} = 2 \tag{11}$$

For all requests in $\mathcal{R}$, the whole competitive ratio is

$$C.R. = \frac{\sum^{k_1} \mu_i \Delta t + \sum^{k_2} (\beta + \beta)}{\sum^{k_1} \mu_i \Delta t + \sum^{k_2} \beta} = 1 + \frac{k_2 \beta}{k_1 \mu_i \Delta t + k_2 \beta} \leq 2 \tag{12}$$

here, k_1 and k_2 represent the number of instances of each case.

□

4.2.2 *Case 2:* $\lambda < \beta \leq 2\lambda$

When $\mu_j \Delta t \leq \lambda$, it's evident that serving request by caching is the optimal choice; When Δt increases to $\mu_j \Delta t > \lambda$ and multiple active models are present, the serving cost would be less costly by transferring than by caching. Meanwhile, when there are no other updated models, the best solution is the same as *Algorithm1*: request $\boldsymbol{r}_i$ should be served by pulling. Therefore, the improved algorithm is shown in *Algorithm.2*.

Algorithm 2: OTSharing-v2 algorithm

```
Input: $t$ : current time ;
       $\vec{t_p}$ : array of the last visit time for each server ;
       $\vec{c}$ : array of servers having active cache items.
Output: $\vec{t_{p'}}$ : updated array of last visit times ;
        $\vec{c'}$ : updated array of cache servers.
for $t_{pi} \leftarrow \vec{t_p}$ do
    if request $i$ takes place on server $i$ then
        if $\vec{c}.size = 0$ then
            serving request $i$ through pulling ;
            $\vec{c'} \leftarrow \vec{c} + e_i$
        end
        if $e_i \in \vec{c}$ then
            serving request $i$ through caching;
        end
        if $e_i \notin \vec{c}$ and $\vec{c}.size \neq 0$ then
            serving request $i$ through transferring ;
            $\vec{c'} \leftarrow \vec{c} + e_i$
        end
        $t_{pi'} \leftarrow t_i$ ;
    end
    $\Delta t_i \leftarrow t_i - t_{pi'}$ ;
    if $\vec{c}.size > 1$ and $\Delta t_i \geq \frac{\lambda}{\mu_i}$ then
        Delete the model from edge server $e_i$ ;
        $\vec{c'} \leftarrow \vec{c} - e_i$
    end
    if $\vec{c}.size = 1$ and $\Delta t_i \geq \frac{\beta}{\mu_i}$ then
        Delete the model from edge server $e_i$ ;
        $\vec{c'} \leftarrow \vec{c} - e_i$
    end
end
```

Theorem 3. Algorithm2 is 2-competitive.

Proof:

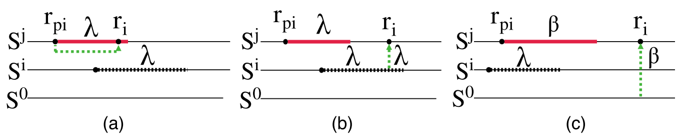


Fig. 8. In case (a), the request is served by caching; in case (b), the request is served by transferring; in case (c), the request is served by pulling.

a) If $\Delta t \leq \frac{\lambda}{\mu_i}$, request $\boldsymbol{r}_i$ is served by caching in both *Optimal* and *Algorithm2* (Fig. 8(a)), $c.r.(a)$ is

$$c.r.(a) = \frac{ALG}{OPT} = \frac{\mu_i \Delta t}{\mu_i \Delta t} = 1 \tag{13}$$

b) If $\Delta t > \frac{\lambda}{\mu_i}$, and there are updated models on other edge servers, the *Algorithm2* still caches the model with a cost λ at first, then serves this request by transferring, while the previous caching is wasted. By contrast, the optimal choice is serving this request by transferring without caching, whose cost is λ (Fig. 8(b)). Thus, $c.r.(b)$ is

$$c.r.(b) = \frac{ALG}{OPT} = \frac{\lambda + \lambda}{\lambda} = 2 \tag{14}$$

c) In the case of $c = 0$, it's evident that the request is served by pulling, and the last active model has been held with a cost β. On the contrary, *Optimal* would serve request by pulling without any caching (Fig. 8(c)), indicating $c.r.(c)$ is

$$c.r.(c) = \frac{ALG}{OPT} = \frac{\beta + \beta}{\beta} = 2 \tag{15}$$

For all requests in $\mathcal{R}$, the whole competitive ratio is

$$\begin{aligned} C.R. &= \frac{\sum^{k_1} \mu_i \Delta t + \sum^{k_2} (\lambda + \lambda) + \sum^{k_3} (\beta + \beta)}{\sum^{k_1} \mu_i \Delta t + \sum^{k_2} \lambda + \sum^{k_3} \beta} \\ &= 1 + \frac{k_2 \lambda + k_3 \beta}{k_1 \mu_i \Delta t + k_2 \lambda + k_3 \beta} \leq 2 \end{aligned} \tag{16}$$

□

4.2.3 *Case 3:* $\beta > 2\lambda$

When $c = 1$, *Algorithm2* will store the model with a cost of β on edge server e_i. And it will serve the request by pulling if the next request time is expired. When $\beta \leq 2\lambda$, pulling is guaranteed to be the optimal solution, but when $\beta > 2\lambda$, due to $\mu_i \neq \mu_j$, the performance of Algorithm 2 is going to be terrible.

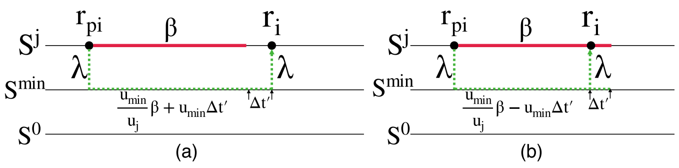


Fig. 9. A new optimal solution in case 3. In the case (a), $\Delta t = \frac{\beta}{\mu_j} + \Delta t'$; in the case (b), $\Delta t = \frac{\beta}{\mu_j} - \Delta t'$. The optimal solutions are green routes.

As shown in Fig. 9, when $\Delta t = \frac{\beta}{\mu_j} \pm \Delta t'$, the optimal route for any edge server $e_j (e_j \neq s^{min})$ would be: transfer the model to the server with the minimal caching cost rate, cache the model until the subsequent request, and then serve the request by transferring. Let $\alpha = \frac{\mu_{min}}{\mu_{max}}$, the competitive ratio of *Algorithm2* in this case will be

$$\lim_{\alpha \to 0, \Delta t' \to 0} \frac{2\beta}{2\lambda + \alpha\beta + \mu_{min} \Delta t'} = \frac{\beta}{\lambda}. \tag{17}$$

Since $\beta > 2\lambda$, *Algorithm2* cannot give a constant upper bound of the competitive ratio. To address this issue, we propose a new bounded online algorithm (*Algorithm3*) as follow.

The major difference between *Algorithm2* and *Algorithm3* lies in the condition of deleting the models from edge servers. In *Algorithm3*, The only model left will be held with a cost of 2λ on e_j, and then transferred to s^{min} and maintained there with another cost of $\beta - 2\lambda$.

Theorem 4. Algorithm3 is 2.5-competitive.

Proof:

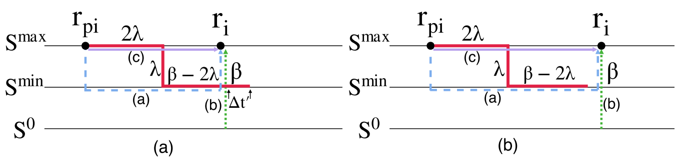


Fig. 10. Two different cases when $c = 1$. The red line represents the schedule of the online algorithm. In case (a), the next request is coming before the active model on s^{min} expired while opposite in case (b).

Algorithm 3: OTSharing-v3 algorithm

```
Input: $t$ : current time ;
       $\vec{t_p}$ : array of the last visit time for each server ;
       $\vec{c}$ : array of servers having active cache items ;
       $s^{min}$ : edge server with minimal cache rate.
Output: $\vec{t_{p'}}$ : updated array of last visit times ;
        $\vec{c'}$ : updated array of cache servers.
for $t_{pi} \leftarrow \vec{t_p}$ do
    if request $i$ takes place on server $i$ then
        if $\vec{c}.size = 0$ then
            serving request $i$ through pulling ;
            $\vec{c'} \leftarrow \vec{c} + e_i$
        end
        if $e_i \in \vec{c}$ then
            serving request $i$ through caching;
        end
        if $e_i \notin \vec{c}$ and $\vec{c}.size \neq 0$ then
            serving request $i$ through transferring ;
            $\vec{c'} \leftarrow \vec{c} + e_i$
        end
        $t_{pi'} \leftarrow t_i$ ;
    end
    $\Delta t_i \leftarrow t_i - t_{pi'}$;
    if $\vec{c}.size > 1$ and $\Delta t_i \geq \frac{\lambda}{\mu_i}$ then
        delete the model from edge server $e_i$ ;
        $\vec{c'} \leftarrow \vec{c} - e_i$
    end
    if $\vec{c}.size = 1$ and $e_i \neq s^{min}$ and $\Delta t_i \geq \frac{2\lambda}{\mu_i}$ then
        transfer the model from $e_i$ to $s^{min}$ ;
        delete the model from edge server $e_i$ ;
        $\vec{c'} \leftarrow \vec{c} - e_i$ ;
        $\vec{t_{p'}}[s^{min}] \leftarrow t_i - \frac{2\lambda}{\mu^{min}}$ ;
        $\vec{c'} \leftarrow \vec{c} + s^{min}$ ;
    end
    if $\vec{c}.size = 1$ and $e_i = s^{min}$ and $\Delta t_i \geq \frac{\beta}{\mu^{min}}$ then
        delete the model from edge server $e_i$ ;
        $\vec{c'} \leftarrow \vec{c} - e_i$
    end
end
```

a) When $c \neq 1$, the competitive ratio of *Algorithm3* is as the same as *Algorithm2*, which is at most 2;

b) When $c = 1$ and $\Delta t \leq \frac{2\lambda}{\mu_i}$, the request is served by caching, which has been proved optimal.

c) When $c = 1$ and $\frac{2\lambda}{\mu_i} < \Delta t \leq \frac{2\lambda}{\mu_i} + \frac{\beta-2\lambda}{\mu_{min}}$ (Fig. 10(a)), the cost of algorithm is:

$$ALG = 2\lambda + \lambda + \lambda + (\beta - 2\lambda - \mu_{min}\Delta t') \tag{18}$$

As illustrated in Fig. 10(a), there are three potential optimal solutions in this case: solution(a), solution(b) and solution(c). The competitive ratio is

$$\begin{aligned} c.r. &= \frac{ALG}{\min\{cost(a), cost(b), cost(c)\}} \\ &= \max\{\frac{ALG}{cost(a)}, \frac{ALG}{cost(b)}, \frac{ALG}{cost(c)}\} \end{aligned} \tag{19}$$

$$\begin{aligned} \frac{ALG}{cost(a)} &= \frac{2\lambda + \lambda + \lambda + (\beta - 2\lambda - \mu_{min}\Delta t')}{\lambda + \lambda + \frac{2\lambda}{\mu_{max}}\mu_{min} + (\beta - 2\lambda - \mu_{min}\Delta t')} \\ &< \lim_{(\beta-2\lambda-\mu_{min}\Delta t')\to 0, \alpha\to 0} \frac{ALG}{cost(a)} = \frac{4\lambda}{2\lambda} = 2 \end{aligned} \tag{20}$$

$$\begin{aligned} \frac{ALG}{cost(b)} &= \frac{2\lambda + \lambda + \lambda + (\beta - 2\lambda - \mu_{min}\Delta t')}{\beta} \\ &\leq \lim_{\Delta t'\to 0} \frac{ALG}{cost(b)} = 1 + \frac{2\lambda}{\beta} < 2 \end{aligned} \tag{21}$$

$$\begin{aligned} \frac{ALG}{cost(c)} &= \frac{2\lambda + \lambda + \lambda + \mu_{min}(\frac{\beta-2\lambda}{\mu_{min}} - \Delta t')}{2\lambda + \mu_{max}(\frac{\beta-2\lambda}{\mu_{min}} - \Delta t')} \\ &< \lim_{(\frac{\beta-2\lambda}{\mu_{min}} - \Delta t')\to 0} \frac{ALG}{cost(c)} = \frac{4\lambda}{2\lambda} = 2 \end{aligned} \tag{22}$$

For all the potential optimal solutions, we proof the competitive ratio satisfies that: $c.r.' < 2$, hence $c.r. = \max(c.r.') < 2$ is proved.

d) When $c = 1$ and $\Delta t > \frac{2\lambda}{\mu_i} + \frac{\beta-2\lambda}{\mu_{min}}$ (Fig. 10(b)), the cost of the algorithm is:

$$ALG = 2\lambda + \lambda + \beta - 2\lambda + \beta \tag{23}$$

As the same as case c), there are also three potential optimal solutions in this case: solution(a), solution(b), and solution(c). For solution(a),

$$cost(a) = \lambda + \lambda + \frac{2\lambda}{\mu_{max}}\mu_{min} + \beta - 2\lambda > \beta, \tag{24}$$

for solution(b), $cost(b) = \beta$. And for solution(c), we have

$$cost(c) = 2\lambda + \mu_{max}\frac{\beta - 2\lambda}{\mu_{min}} > \beta \tag{25}$$

Apparently, the optimal solution is solution(b) with the minimum cost β. Therefore, the competitive ratio is :

$$c.r. = \frac{2\lambda + \lambda + \beta - 2\lambda + \beta}{\beta} = 2 + \frac{\lambda}{\beta} < 2 + \frac{\lambda}{2\lambda} = 2.5 \tag{26}$$

To sum up, for all the requests in $\mathcal{R}$, the overall competitive ratio is

$$\begin{aligned} C.R. &= \frac{\sum^{k_1}\mu_i\Delta t + \sum^{k_2}(\lambda+\lambda) + \sum^{k_3}(\beta+\beta)}{\sum^{k_1}\mu_i\Delta t + \sum^{k_2}\lambda + \sum^{k_3}\beta + \sum^{k_4}\beta} \\ &+ \frac{\sum^{k_4}(2\lambda + \lambda + \beta - 2\lambda + \beta)}{\sum^{k_1}\mu_i\Delta t + \sum^{k_2}\lambda + \sum^{k_3}\beta + \sum^{k_4}\beta} \\ &= \frac{(2k_1\mu_i\Delta t + 2k_2\lambda + 2k_3\beta + 2k_4\beta) + k_4\lambda - k_1\mu_i\Delta t}{k_1\mu_i\Delta t + k_2\lambda + k_3\beta + k_4\beta} \\ &\leq 2 + \frac{k_4\lambda - k_1\mu_i\Delta t}{k_4\beta + k_1\mu_i\Delta t} \\ &\leq 2 + \frac{\frac{1}{2}k_4\beta - k_1\mu_i\Delta t}{k_4\beta + k_1\mu_i\Delta t} \\ &= 2.5 - \frac{3(k_1\mu_i\Delta t)}{2(k_4\beta + k_1\mu_i\Delta t)} \leq 2.5 \end{aligned} \tag{27}$$

□

Besides, we analyze the lower bound of this online problem closely and introduce the following theorem:

Theorem 5. The competitive ratio of the online DNN model sharing problem is at least 2 for any deterministic online algorithm.

Proof: Assume there is an adaptive adversary that produces a synchronization request sequence and tries to mislead the online algorithm and make its competitive ratio worse. For any deterministic online algorithm $\mathcal{A}$, the adversary can construct a sequence to counter the online algorithm. To construct the worst case, we assume that all the requests occur on s^{min}. In this case, all algorithms could only lever caching and pulling to serve requests instead of transferring.

It is reasonable to assume the caching cost of $\mathcal{A}$ is h after the i-th request is satisfied, and the adversary makes a follow-up request at time $\frac{h}{\mu_{min}} + \tau$, where τ is a very tiny interval. When $h < \beta$, we have:

$$c.r. = \frac{h+\beta}{h} > \frac{2h}{h} = 2 \tag{28}$$

Similarly, when $h \geq \beta$, we have:

$$c.r. = \frac{h+\beta}{\beta} \geq \frac{2\beta}{\beta} = 2 \tag{29}$$

Let h_s be any caching cost of $\mathcal{A}$ after request $\boldsymbol{r}_i$ is satisfied in the case $h < \beta$, and h_l for $h \geq \beta$ correspondingly. Suppose there are m h_s and n h_l for a request sequence $\mathcal{R}$, then the competitive ratio is:

$$\begin{aligned} C.R. &= \frac{\sum^m(\beta+h_s) + \sum^n(\beta+h_l)}{\sum^m(h_s) + \sum^n(\beta)} \geq \frac{\sum^m(\beta+h_s) + 2n\beta}{\sum^m(h_s) + n\beta} \\ &> \frac{2mh_s + 2n\beta}{mh_s + n\beta} > \frac{2mh_s + 2nh_s}{mh_s + nh_s} = 2 \end{aligned} \tag{30}$$

In summary, we construct a worst-case where any online algorithm cannot attain a competitive ratio of more than 2. Therefore, we declare that the lower bound on the problem is 2, as well as conclude the theorem. □

This theorem shows the high values of our algorithm in practical uses.

4.3 Cloud-Edge Model Synchronization

According to the system model, the cloud center would proactively notify an edge server to download the latest updated model for serving its syn-requests upon the completion of its training phase. In this section, we will introduce how to extend the off-line and online algorithm to support such update requests.

4.3.1 Preliminary

For a periodically updated model, we can specify the proactive notifications issued by the cloud center as a *notification sequence*, denoted by $\mathcal{U} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_k\}$, where notification $\boldsymbol{u}_i$ is made by the cloud center s^0 at time instance t_i. The notified server first pulls the updated model from the cloud, then passes it to other servers via transferring to serve the requests from mobile devices in the sequel.

4.3.2 Extended off-line Algorithm

Note that all requests, as well as updates, are given in advance, in the off-line mode. Given the deduced update sequence $\mathcal{U}$, we can derive an optimal off-line algorithm to support the updates based on the proposed off-line algorithm in Section 4.1.

As the same as $\boldsymbol{r}_i$, $\boldsymbol{u}_i = (e_i, t_i)$, e_i represents the server that needs to be updated, and t_i denotes the corresponding time. It is natural to assume that there is an update $\boldsymbol{u}_0$ before the request sequence $\mathcal{R}$. And $\mathcal{R}$ can be divided into multiple sub-sequences combined with the update sequence. The request sequence with updates can be denoted as follow:

$$\mathcal{R} \cup \mathcal{U} = \{\boldsymbol{u}_0, \boldsymbol{r}_0, ..., \boldsymbol{r}_{\boldsymbol{u}_1}\} \cup \{\boldsymbol{u}_1, \boldsymbol{r}_{\boldsymbol{u}_1+1}, ..., \boldsymbol{r}_{\boldsymbol{u}_2}\} \cup ... \cup \{\boldsymbol{u}_k, ..., \boldsymbol{r}_n\}$$

To deal with the request sequence with updates in off-line form, an intuitive and reasonable idea is to leverage the proposed off-line algorithm on each sub-sequence separately, which obviously leads to the optimal solution. Therefore, the extension of the off-line algorithm is to decompose the request sequence concerning the update time and solve the sub-sequence separately.

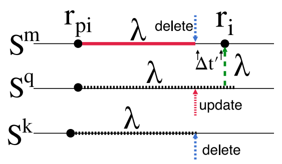


Fig. 11. When the update operation occurs on s^q, the models on other edge servers (s^m, s^k) will be deleted, and the subsequent requests on other edge servers are satisfied via transferring or pulling.

4.3.3 Extended Online Algorithm

In the online version, it is reasonable to assume that updates only occur on servers with active caches for the availability guarantee. Besides, this assumption would not affect the discussion in terms of the competitive ratio.

We make the following extension to the online algorithm to handle updates: once the update operation occurs on s^q, the algorithm must expire other stored models on edge servers ($s^m \neq s^q$) right now. As shown in Fig. 11, the stored model on s^m and s^k should be deleted when the update has happened on s^q.

Theorem 6. The extended online algorithm with update operation is 2.5-competitive.

Proof: In the case of $c = 1$, since the cost of deleting is negligible, the update operation brings barely any extra cost when there is only one active cache. In contrast, it is a non-trivial case when $c \neq 1$, because the occurred updating results in deleting on $c - 1$ active servers, bringing about a change in the cost of subsequent requests. It can be found that when subsequent requests are supposed to be served by caching, inserting an update operation would force them to be served by transferring or pulling. Analyses conducted on caching in this case is shown as follow:

$$c.r. = \frac{\lambda - \mu_i \Delta t' + \lambda}{\lambda} < \lim_{\Delta t' \to 0} \frac{\lambda - \mu_i \Delta t' + \lambda}{\lambda} = \frac{2\lambda}{\lambda} = 2 \tag{31}$$

here, $\Delta t'$ is the offset between the time updating occurred and the time the cache should be deleted originally. Note that the update does not affect the cost of this algorithm when $c = 1$. And the competitive ratio of the algorithm when $c \neq 1$ has been proved to be less than 2. Thus, the competitive ratio of the entire algorithm is still 2.5. □

In summary, by considering the pulling and transferring operations in the model sharing problem, we discussed the relationships between β and λ and proposed online algorithms for three cases, respectively. We also analyzed and proved that the presented algorithms could achieve a competitive ratio of 2.5, even combined with update operations.

5 Evaluation

We conducted experiments to evaluate the actual performance of the designed algorithms and validate the proposed theorems. The simulator efficiently implements the proposed algorithms and the model upon which the algorithms are built.

5.1 Experimental Setup

We investigated a practical scenario where the user performs DNN inference (e.g., image recognition, voice input, etc.) through a mobile device while the DNN training is deployed in the cloud center. When users need to update the model, they can synchronize it through edge servers provided by the cloud service provider (e.g., *Amazon Web Services*, AWS[3]). Considering the charging mechanism of cloud service providers, we hope to turn down the model serving service when there is no user synchronization to reduce storage and communication costs.

Dataset: We adopted a public open international mobile network dataset as the experimental trace data [34]. The dataset tracks mobile users' access to web services and is collected by the MONROE platform spanning six countries, 27 mobile network operators, and 120 measurement nodes. We sampled 7000 records from it to simulate the model synchronizations. As a simulation of workloads, we leveraged the CIFAR-10 [35] as the training and test dataset.

Baselines: For convenience, we named the proposed off-line and online algorithms as *DTSharing* and *OTSharing*, respectively, and compared them with the off-line *Greedy* and online ***A**ctive **C**aching with **3**-competitive ratio* (*AC3*) algorithms developed in [36]. In particular, we used *UOTSharing* to denote the *OTShring* algorithm with an active update mechanism. To this end, we stipulated that the training and updates of the DNN model are performed every 500 requests. In addition, we took a DNN model *ResNet50* as the workload.

Parameters: We took the billing policy of AWS as a reference to model the cache cost of servers. For instance, "m4.xlarge" costs 48¢ per hour (i.e., 0.8¢ per minute) [37]. Therefore, we set the unit cache cost of edge servers as a uniform distribution μ_i $U[0.4, 1.6]$. Similarly, we made AWS data transfer price as the cost of data transmission in the simulation, which takes approximately 14¢ per GB transfer out to the Internet and 6¢ per GB between regions within Asia [38]. The size of a pre-trained *ResNet50* model is 102MB, which means that the transmission and pull costs are approximately 1.4¢ and 0.6¢ respectively ($\beta = 1.4, \lambda = 0.6$).

Moreover, we defined the performance ratio as $\rho = \mathcal{C}(\mathcal{A})/\mathcal{C}(DTSharing)$, which is used to measure the cost-efficiency of algorithms. Obviously, the lower the performance ratio (approx to 1) of an algorithm, its cost is more approx to the optimal (i.e., the cost of *DTSharing*), which indicates better performance.

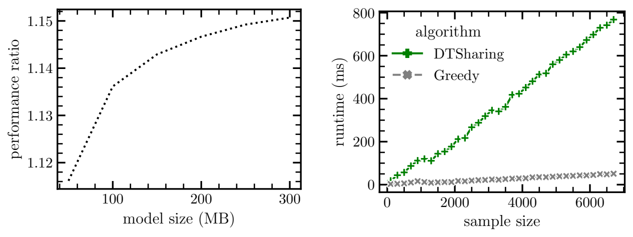
Regarding the hardware of the platform, we treated Raspberry Pi 4b as the edge servers, Tesla-V100 as the cloud center, and leveraged *PyTorch* for DNN training and inference.

5.2 The off-line algorithm

To validate the correctness and performance of the off-line algorithm, we designed a greedy algorithm *Greedy* as a baseline for comparison. The main idea of the greedy algorithm is to start with the last request of all edge servers and find a path with the lowest cost as the solution. The time complexity of the greedy algorithm is $O(mn)$.

Firstly, we investigated the performance of *DTSharing* and *Greedy* by calculating the performance ratio, which is defined by the cost of *Greedy* over the cost of *DTSharing*. As shown in Fig.12(a), in all our experiments with different

3. https://aws.amazon.com



(a) How the performance ratio is relatively changed under different model sizes. The y-axis is the performance ratio.

(b) How the running time of the compared algorithms is varied with the number of requests.

Fig. 12. Performance of the compared algorithms.

model sizes, the performance ratios are more significant than 1, indicating *DTSharing* is better than *Greedy*. Moreover, the ratio of *Greedy* gradually rises with the increase of the model size, indicating that the performance of *Greedy* gets worse as the model increases.

Afterward, we compared the running time of the algorithms and took the speedup ratio (i.e., the running time of *DTSharing* over the running time of *Greedy*) as a metric. To this end, we sampled the request sequence to investigate that the running time varies with the number of sampled requests (n) with an increment of every 100 samples. As shown in Fig.12(b), the running times of both algorithms grow up linearly with the increasing number of requests, which is consistent with the complexity we analyzed.

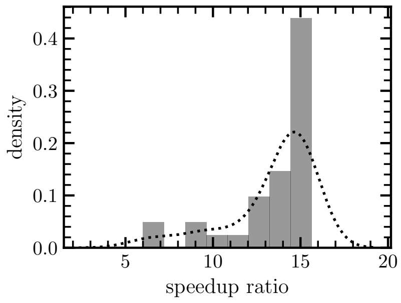


Fig. 13. Distribution of the speedup ratio of the greedy algorithm, representing the comparison of the running time of the two algorithms. The x-axis is the speedup ratio, and the y-axis is the density of the speed ratio.

Since the number of edge servers (m) is fixed, the speedup ratio of the greedy algorithm should be a constant from our proposition in complexity analysis. The distribution of the speedup ratios is demonstrated in Fig.13, which is circa 15×, implying that the speedup ratio is stable even though the number of requests is changed. This illustrates that *DTSharing* only takes a constant multiply of the time over *Greedy* to attain the optimal solution.

5.3 The online algorithm

With the optimal cost achieved by *DTSharing*, we studied the cost-efficiency of both *OTSharing* and *UOTSharing*. We first investigated the impact of different DNN models, then examined their performance by changing the model size from 50MB to 300MB, and letting λ and β vary with the model size correspondingly.

As shown in Fig.14, the performance ratios of the compared algorithms increase moderately but eventually flatten out as the DNN model size grows. This observation is consistent with our expectation that the performance ratios

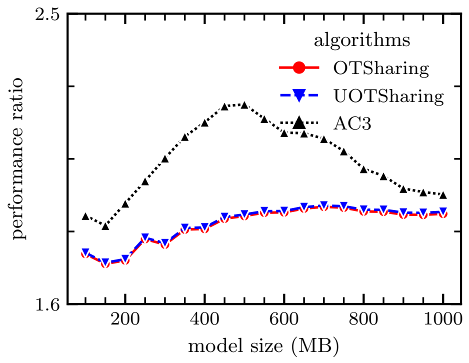


Fig. 14. Performance ratio changes with respect to different model sizes. A lower ratio is better than a higher one, implying more cost-efficient.

gradually converge to a constant, demonstrating that the online algorithm's competitive ratio (performance ratio) is bounded. Meanwhile, it is worth noting that the performance of *OTSharing* is always slightly better than *AC3* (the lower, the better). Regarding the descent issue of the *UOTSharing*'s performance as the model size increases, we speculated that since the period pulling brought by the update sequence, the costs of transfer and pull are increased correspondingly. For deep insight, we decomposed the cost composition of different algorithms as shown in Fig.15.

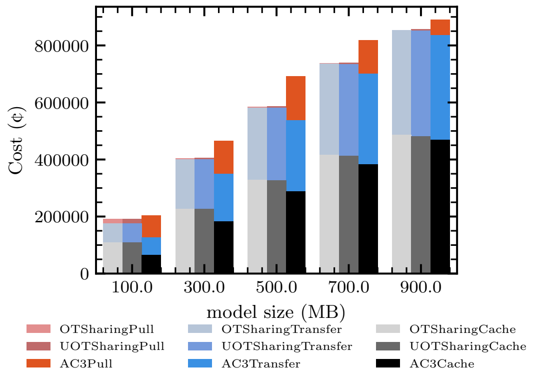
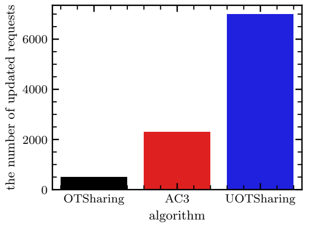


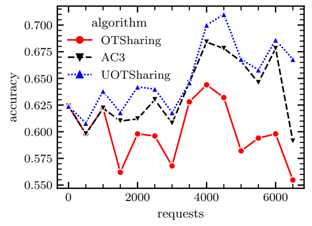
Fig. 15. The cost of request sequence is changed with respect to model size in different algorithms. The total cost consists of three parts–caching, transferring, and pulling cost, and the breakdowns of each bar correspond to these three parts from the bottom to the top.

The bar diagram in Fig.15 illustrates how the cost is profiled for all the investigated algorithms. We can find that a large model would lead to a significant increase in the proportion of pulling and transferring in *UOTSharing*, which is consistent with our speculation. On the contrary, the edge servers perform pulling barely in *OTSharing*, and cache cost accounts for a higher proportion than other algorithms. Due to the total cost of *OTSharing* being the lowest, the cache in *OTSharing* is significantly more cost-efficient.

Despite reducing the use of pulling results in higher cost efficiency, the frequency of model updating is also reduced, which would affect the accuracy of the inference. To compare the accuracy of baselines, we constructed an experiment by leveraging *ResNet50* [39] to classify the *CIFAR-10* dataset. To emulate the ICIE architecture, we opted to attach a photo to each request for inference. Afterward, devices transmit the photo to the cloud center, where the pre-trained ResNet-50 resides. The GPU trains one epoch with every 500 new data pieces (500 requests) received. Upon completion of the training, the updated model can be retrieved by means of a pull, resulting in improved accuracy compared to the outdated model in practice.



(a) The number of requests to retrieve updated model.



(b) The classification accuracy per 500 request intervals.

Fig. 16. The number of requests to retrieve the newest trained *ResNet50* model, and the corresponding accuracy of the retrieved models on *CIFAR-10* in different algorithms.

The numerical results are shown in Fig.16. First, we investigated the number of requests retrieved after the current latest model was constructed, which means that these requests occurred after a pull and before the next training. As illustrated in Fig.16(a), it is evident that all the requests in *UOTSharing* can be satisfied by the latest model, while *OTSharing* almost takes the expired model. As such, an intuitive and reasonable conjecture is that the accuracy of *OTSharing* should be lower than others, and *UOTSharing* should exhibit the best. This conclusion is confirmed by Fig.16(b). Although the accuracies of the algorithms are the same initially, with the leverage of the pull, the accuracy of *UOTSharing* is significantly higher than the others. Therefore, it can be concluded that the *UOTSharing* with the update mechanism is superior to *AC3* in terms of cost efficiency and model update frequency (which directly affects the accuracy).

6 Related Work

Currently, many mainstream model serving platforms (e.g., TensorFlow Serving [15], SageMaker [16], and DLHub [17]) focus mainly on serving the inference tasks by efficiently scheduling server-side resources [15]–[17]. In contrast, we concentrate on model sharing in this paper, which is often dedicated to model serving for providing mobile users with on-demand model inference-based applications in a cost-efficient way. Liu *et al.* [40] studied federated reinforcement learning in a cloud robotic system by using model sharing to achieve knowledge transfer among different robots. In contrast, Jiang *et al.* [41] presented a model sharing framework in the edge to facilitate cross-domain object detection in autonomous driving. Though oriented to the edge, unlike this study, we paid more attention to exploiting the edge server as a cache in a cost-efficient way, rather than computing resources as mentioned in [40] [41].

In comparison to our work, some studies also take the networked edge servers as caches for data or knowledge sharing. For example, Guo *et al.* [42] developed an intelligence-sharing vehicular network for sharing the knowledge acquired from DNN in MEC-enabled vehicular networks. As with our serving framework, the network can transfer the established model trained at the cloud center to the edge servers if necessary. However, most of these studies mainly focus on the mechanism of model sharing, instead of the cost issue we concerned.

In spirit, the model sharing in our problem has some similarities with the *data sharing* in CDN [20]. Huang *et al.* [43] formalized the data sharing problem as a multiple *Connected Facility Location problem* [44] and developed a 6.55-approximate algorithm to solve the fairness of this problem while Luo *et al.* [45] studied the problem in edge settings, and proposed an approximate balanced greedy algorithm to make the content distribution more balanced. Both algorithms are off-line, each with its own goal, lacking the notion of online for cost reduction as in our case.

Wang *et al.* [30] proposed a homogeneous cost model for data sharing in the cloud, thereby presenting an efficient off-line optimal algorithm and a 3-competitive online algorithm to reduce the overall sharing cost. We improved their cost model to a semi-homogeneous one whereby the model sharing, together with its off-line and online algorithms for cost reduction, is deployed in the edge network. In addition, to adapt the algorithms to the model sharing scenario, we took both pulling and update mechanisms into account, making the problem more complicated than the previous. On this basis, we designed an online algorithm that is better than the original (lower competitive ratio) and conducted experimental comparisons.

In addition to the foregoing related studies, there are also some learning-based methods, which focus on the problem related to ours [18], [19], [46], [47]. For example, Yang *et al.* [46], [47] proposed a multi-task framework to address the resource allocation and offloading decision issues in the MEC networks. However, these methods are beyond the scope of our consideration as they cannot provide theoretical guarantees for the performance to deal with our sharing problem.

7 Conclusion

In this paper, we investigated the cost-efficient model sharing algorithms for the DNN serving in the edge network. We first formulated the problem and analyzed its complexity. Then based on a semi-homogeneous cost model, we proposed a $O(m^2n)$ (which can be reduced to $O(m \log m\ n)$ with a well-designed data structure) optimal algorithm for its off-line case and a 2.5-competitive algorithm for its online case, respectively. To evaluate the quality of the competitive ratio, we also proved a lower bound of 2 based on the same cost model for any deterministic online algorithm. We also considered the update operations and designed off-line and online algorithms supporting the update mechanism. We achieved these results with our in-depth insights and careful analysis of this problem. Finally, we validated our algorithms through a trace-based simulation study. Again, although the DNN is a concern in this paper, the proposed algorithms are generic and simple enough to adapt to other appropriate content delivery-like scenarios.

In summary, the off-line algorithm is optimal in service cost minimization but only practical in some instances, while the online algorithm is sub-optimal but more practical in reality. In addition, Although both off-line and online algorithms are efficient, they are under global control, which could be deemed as the major limitation in the current design. We plan to design the distributed version in our near future work. Meantime, to align with the characteristics of DNN training, we are also contemplating an integration of a penalty mechanism into the synchronization process, with the aim of increasing accuracy while preserving the cost of the end devices.

Acknowledgements

This work was supported in part by Key-Area Research and Development Program of Guangdong Province (2020B010164002), and Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0015/2019/AKP.

References

[1] G. Huang, C. Luo, K. Wu, Y. Ma, Y. Zhang, and X. Liu, "Software-defined infrastructure for decentralized data lifecycle governance: Principled design and open challenges," in *Proceedings - International Conference on Distributed Computing Systems*, 2019.

[2] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys*, vol. 52, no. 1, 2019.

[3] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of Edge Computing and Deep Learning: A Comprehensive Survey," *IEEE Communications Surveys and Tutorials*, vol. 22, no. 2, pp. 869–904, 2020.

[4] J. Wu, H. Dai, Y. Wang, Y. Zhang, D. Huang, and C. Xu, "Pack-cache: An online cost-driven data caching algorithm in the cloud," *IEEE Transactions on Computers*, pp. 1–8, 2022.

[5] T. Ouyang, Z. Zhou, and X. Chen, "Follow Me at the Edge: Mobility-Aware Dynamic Service Placement for Mobile Edge Computing," *JSAC*, 2018.

[6] M. Du, Y. Wang, K. Ye, and C. Xu, "Algorithmics of Cost-Driven Computation Offloading in the Edge-Cloud Environment," *IEEE Transactions on Computers*, 2020.

[7] X. Wang, Z. Ning, and S. Guo, "Multi-Agent Imitation Learning for Pervasive Edge Computing: A Decentralized Computation Offloading Algorithm," *TPDS*, 2021.

[8] X. Xia, F. Chen, Q. He, J. C. Grundy, M. Abdelrazek, and H. Jin, "Cost-Effective App Data Distribution in Edge Computing," *TPDS*, 2021.

[9] H. Peng and X. Shen, "Multi-Agent Reinforcement Learning Based Resource Management in MEC- And UAV-Assisted Vehicular Networks," *JSAC*, 2021.

[10] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing," *Proceedings of the IEEE*, vol. 107, no. 8, 2019.

[11] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge," *Proceedings - IEEE INFOCOM*, vol. 2019-April, pp. 1423–1431, 2019.

[12] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, "In-edge AI: Intelligentizing mobile edge computing, caching and communication by federated learning," *IEEE Network*, 2019.

[13] Y. Chen, N. Zhang, Y. Zhang, X. Chen, W. Wu, and X. S. Shen, "TOFFEE: Task Offloading and Frequency Scaling for Energy Efficiency of Mobile Devices in Mobile Edge Computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.

[14] J. Park, S. Samarakoon, M. Bennis, and M. Debbah, "Wireless Network Intelligence at the Edge," *Proceedings of the IEEE*, vol. 107, no. 11, pp. 2204–2239, 2019.

[15] Tensorflow, "Tensorflow serving," [EB/OL], 2021, http://tensorflow.org/.

[16] Amazon, "Amazon sagemaker," [EB/OL], 2021, https://aws.amazon.com/sagemaker/.

[17] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. Foster, "DLHub: Model and data serving for science," in *IPDPS*, 2019.

[18] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman, "Rl-cache: Learning-based cache admission for content delivery," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2372–2385, 2020.

[19] C. Cho, S. Shin, H. Jeon, and S. Yoon, "Ttl-based cache utility maximization using deep reinforcement learning," in *2021 IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6.

[20] G. Tang, K. Wu, and R. Brunner, “Rethinking cdn design with distributee time-varying traffic demands,” in *INFOCOM*, 2017.

[21] F. Wang, F. Wang, J. Liu, R. Shea, and L. Sun, “Intelligent Video Caching at Network Edge: A Multi-Agent Deep Reinforcement Learning Approach,” *INFOCOM*, 2020.

[22] T. Taleb, P. A. Frangoudis, I. Benkacem, and A. Ksentini, “CDN slicing over a multi-domain edge cloud,” *TMC*, 2020.

[23] Y. Wang, H. Dai, X. Han, P. Wang, Y. Zhang, and C.-Z. Xu, “Cost-driven data caching in edge-based content delivery networks,” *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[24] M. Eriksson, “Cost modelling of edge compute,” in *The Edge Event*, 09 2020.

[25] R. Pechter, “What’s PMML and what’s new in PMML 4.0?” *ACM SIGKDD Explorations Newsletter*, 2009.

[26] J. Pivarski, C. Bennett, and R. L. Grossman, “Deploying analytics with the portable format for analytics (PFA),” in *KDD*, 2016.

[27] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, “Dynamic Service Migration in Mobile Edge Computing Based on Markov Decision Process,” *TON*, 2019.

[28] L. Bottou and Y. Le Cun, “Large scale online learning,” in *Advances in Neural Information Processing Systems*, 2004.

[29] A. Mehta, W. Tarneberg, C. Klein, J. Tordsson, M. Kihl, and E. Elmroth, “How beneficial are intermediate layer data centers in mobile edge networks?” in *FAS-W*, 2016.

[30] Y. Wang, S. He, X. Fan, C. Xu, and X. H. Sun, “On Cost-Driven Collaborative Data Caching: A New Model Approach,” *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[31] M. R. Garey and D. S. Johnson, “The Rectilinear Steiner Tree Problem is \$NP\$-Complete,” *SIAP*, 1977.

[32] W. Shi and C. Su, “The rectilinear Steiner arborescence problem is NP-complete,” *SIAM Journal on Computing*, 2005.

[33] X. Xia, F. Chen, Q. He, J. C. Grundy, M. Abdelrazek, and H. Jin, “Cost-Effective App Data Distribution in Edge Computing,” *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[34] A. S. Khatouni, M. Trevisan, D. Giordano, M. Rajiullah, S. Alfredsson, A. Brunstrom, C. Midoglu, and Ö. Alay, “An Open Dataset of Operational Mobile Networks,” in *MobiWac*, 2020.

[35] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” *University of Toronto*, 2009.

[36] Y. Wang, S. He, X. Fan, C. Xu, J. Culberson, and J. Horton, “Data Caching in Next Generation Mobile Cloud Services, Online vs. Off-Line,” *ICPP*, pp. 412–421, 2017.

[37] L. N. Hyseni and A. Ibrahimi, “Comparison of the cloud computing platforms provided by amazon and google,” in *2017 Computing Conference*, 2017, pp. 236–243.

[38] A. W. Services, “Amazon cloudfront pricing,” [EB/OL], 2021, https://aws.amazon.com/cloudfront/pricing/.

[39] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *CVPR*, 2016.

[40] B. Liu, L. Wang, and M. Liu, “Lifelong federated reinforcement learning: A learning architecture for navigation in cloud robotic systems,” *IEEE Robotics and Automation Letters*, 2019.

[41] X. Jiang, F. Richard Yu, T. Song, Z. Ma, Y. Song, and D. Zhu, “Blockchain-Enabled Cross-Domain Object Detection for Autonomous Driving: A Model Sharing Approach,” *IEEE Internet of Things Journal*, 2020.

[42] J. Guo, W. Luo, B. Song, F. R. Yu, and X. Du, “Intelligence-sharing vehicular networks with mobile edge computing and spatiotemporal knowledge transfer,” *IEEE Network*, 2020.

[43] Y. Huang, X. Song, F. Ye, Y. Yang, and X. Li, “Fair Caching Algorithms for Peer Data Sharing in Pervasive Edge Computing Environments,” in *ICDCS*, 2017.

[44] C. Swamy and A. Kumar, “Primal-dual algorithms for connected facility location problems,” in *Algorithmica*, 2004.

[45] G. Luo, H. Zhou, N. Cheng, Q. Yuan, J. Li, F. Yang, and X. Shen, “Software-Defined Cooperative Data Sharing in Edge Computing Assisted 5G-VANET,” *IEEE Transactions on Mobile Computing*, 2021.

[46] B. Yang, X. Cao, C. Yuen, and L. Qian, “Offloading optimization in edge computing for deep-learning-enabled target tracking by internet of uavs,” *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 9878–9893, 2021.

[47] B. Yang, X. Cao, J. Bassey, X. Li, and L. Qian, “Computation offloading in multi-access edge computing: A multi-task learning approach,” *IEEE Transactions on Mobile Computing*, vol. 20, no. 9, pp. 2745–2762, 2021.

Hao Dai received the M.Sc. degree in Communication and Electronic Technology from Wuhan University of Technology, in 2017. He is currently working toward the PhD degree in the Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interests include cloud computing, big data processing, mobile edge computing systems.

Jiashu Wu received BSc. degree in Computer Science and Financial Mathematics & Statistics from the University of Sydney, Australia (2018), and M.IT degree in Artificial Intelligence from the University of Melbourne, Australia (2020). He is currently pursuing his Ph.D at the University of Chinese Academy of Sciences (Shenzhen Institute of Advanced Technology, CAS). His research interests include big data and cloud computing.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China (1989), and the MSc. and PhD. degrees in computer science from Carlton University (2001) and University of Alberta, Canada (2008), respectively. He is currently with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, as a full professor and with Xiamen University, China as an adjunct professor. His research interests include service and cloud computing, programming language implementation, and software engineering. He is an Alberta Industry R&D Associate (2009-2011), and a Canadian Fulbright Scholar (2014-2015).

Jerome Yen received his Ph.D. degree in Systems Engineering and Management Information Systems from the University of Arizona. He is now a Distinguished Professor of Department of Computer and Information Science, Faculty of Science and Technology, University of Macau, China. Prior to his current post, he was a Professor of Accounting and Finance department of Tung Wah College, and a Honorary Professor of Department of Electrical and Electronic Engineering, The University of Hong Kong. His main research interests include financial engineering, investment management, market and credit risk management, financial product development, trading strategies and hedge funds.

Yong Zhang received his Ph.D. in the Department of Computer Science and Engineering at Fudan University in 2007. He is now a Professor in SIAT, CAS, Honorary Professor at the University of Hong Kong. Before joining SIAT, he worked as Post-Doctoral Fellow and Senior Researcher in TU-Berlin and HKU. He has published more than 100 papers in refereed journals and conferences. His research interests include design and analysis of algorithms, combinatorial optimization, and wireless networks.

Chengzhong Xu obtained B.Sc. and M.Sc. degrees from Nanjing University in 1986, and 1989, respectively, and a Ph.D. degree from the University of Hong Kong in 1993, all in Computer Science and Engineering. Currently, he is a Chair Professor of Computer Science and the Dean of Faculty of Science and Technology, University of Macau, China. His recent research interests are in cloud and distributed computing, systems support for AI, smart city and autonomous driving. He has published more than 400 papers in journals and conferences. He serves on a number of journal editorial boards and the Chair of IEEE TCDP from 2015 to 2020. He is a fellow of the IEEE.