AdaInf: Data Drift Adaptive Scheduling for Accurate and SLO-guaranteed Multiple-Model Inference Serving at Edge Servers

Sudipta Saha Shubha
University of Virginia
USA

Haiying Shen
University of Virginia
USA

ABSTRACT

Various audio and video applications rely on multiple deep neural network (DNN) models deployed on edge servers to conduct inference with ms-level latency service-level-objectives (SLOs). To avoid accuracy decreases caused by data drift, continual retraining is necessary. However, this poses a challenge for GPU resource allocation to satisfy the tight SLOs while maintaining high accuracy in this scenario. There has been no research devoted to tackling this issue. In this paper, we conducted trace-based experimental analysis in this particular scenario, which shows that different models have varying degrees of impact from data drift, incremental retraining (proposed by us that retrains certain samples before inference) and early-exit model structures can help increase accuracy, and the interdependencies among tasks may lead to significant CPU-GPU memory communications. Leveraging these unique observations, we propose a data drift Adaptive scheduler for accurate and SLO-guaranteed Inference serving at edge servers (`AdaInf`). `AdaInf` uses incremental retraining and allocates GPU amount among applications based on their SLOs. For each application, it splits GPU time between retraining and inference to satisfy its SLO, and then allocates GPU time among retraining tasks based on their impact degrees. In addition, `AdaInf` proposes strategies that leverage the job features in this scenario to reduce the impact of CPU-GPU memory communications on latency. Our real trace-driven experimental evaluation shows that `AdaInf` can increase accuracy by up to 21% and reduce SLO violations by up to 54% compared to existing methods. Achieving similar accuracy as `AdaInf` requires 4× more GPU resources on the edge server for the existing method.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Edge server, deep learning, data drift, retraining, inference serving

ACM Reference Format:
Sudipta Saha Shubha and Haiying Shen. 2023. AdaInf: Data Drift Adaptive Scheduling for Accurate and SLO-guaranteed Multiple-Model Inference Serving at Edge Servers. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3603269.3604830

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604830

1 INTRODUCTION

Deep neural networks (DNNs) have revolutionized various domains by leveraging powerful GPUs and large datasets. With the availability of DNN models, application developers have started building multi-model applications (e.g., video surveillance, traffic monitoring, real-time analysis of drone footage, and language translation) that process audio or video streams in real time [1]. A multi-model application consists of several DNN models organized in a directed acyclic graph (DAG). For example, as shown in Fig. 1, in a video surveillance application, the output of an object detection model is used as the input of a vehicle type recognition model (to recognize personal car, ambulance, etc.) and a person activity recognition model (to recognize walking, fighting, etc.).

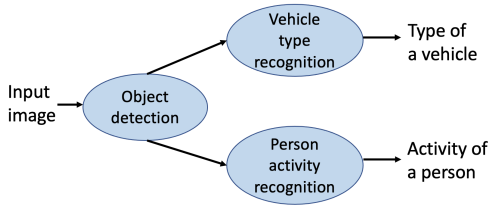


Figure 1: DAG of the video surveillance application.

To ensure that inference requests meet ms-level latency service-level-objectives (SLOs) for each application, developers typically deploy the application's models on an edge server within a small-scale data center that serves a specific locality [2]. Unlike distant cloud servers, edge servers can provide low latency services and offer privacy protection for user data as regulations such as GDPR mandate against sending videos or audios outside of a specific locality [3]. As shown in Fig. 2, an edge server is connected to a cellular access network on one end and to the cloud through a wide-area network on the other end [4]. Through the cellular network, users submit inference requests to the edge server, and the edge server returns the inference results. Various commercial options are currently available for public edge servers, such as Amazon AWS Wavelength and Microsoft Azure Stack Edge.

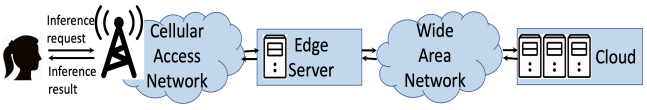


Figure 2: Edge server deployment in real world.

However, edge servers often have limited resources with low-capacity or only a few GPUs [5], which is further compounded by the growing computational demands of DNN applications outpacing the capabilities of GPUs. Model compression techniques are applied to address this problem, but the accuracy of the compressed models can significantly drop due to data drift, where live audio or video data diverges significantly from the training data [3]. For example, in the video surveillance application, cameras in streets encounter varying scenarios over short time intervals, such as sudden changes in lighting or occlusion conditions [6], and rapid changes in the distribution of vehicle types or person activities [3]. Because compressed DNNs have shallower architectures and fewer weights, they are not generalizable to new data distributions or changed conditions, which can affect accuracy [7].

To ensure high accuracy for applications with rapidly changing situations, it is important to perform continual learning or retraining [3, 8, 9] at short time intervals (e.g., 50s) [3]. In the retraining, the DNNs are retrained on new samples (i.e., inference requests) collected during the previous time period. The class labels of the retraining data are obtained from a "golden model" hosted in the cloud [3]. However, retraining on local edge servers [3] can negatively impact inference accuracy and latency due to limited GPU resources and resource competition with inference executions. Fig. 3 illustrates the timeline of the continual learning process, which starts at 0ms and completes at 40s. During this time, inference requests that arrive cannot benefit from the retrained models. To reduce the latency of the retraining process, we could allocate more resources to it. However, this would leave fewer resources for inference tasks, which may cause them to miss their SLOs. Therefore, conducting both retraining and inference serving for multi-model applications in an edge server poses a challenge for GPU resource allocation to tasks to satisfy inference latency SLOs while maximizing the average accuracy of all applications.

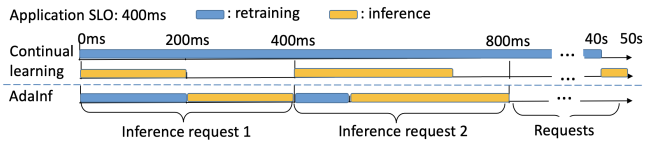


Figure 3: Continual learning and incremental retraining in AdaInf.

The challenge of resource allocation becomes even more formidable in the context of multi-model applications for several reasons. First, each application requires significantly more computing resources for both retraining and inference compared to single-model applications. Second, addressing the challenge while considering the dependencies of inference tasks on the retraining tasks, as well as the interdependencies between the inference tasks themselves, is non-trivial. Finally, when scheduling resources for multiple DAGs, there can be an enormous number of potential solutions. A heuristic [3] enumerating solutions to find the best one, or solving an optimization problem [10], cannot meet the short scheduling time requirement, which is critical for satisfying SLOs.

Within our knowledge, there has been no research devoted to tackling this challenge. To address the research gap, we propose a method called *incremental retraining*. It retrains a model as much as possible every time before it is used by an inference task, utilizing spare time within the SLO. For example, as shown in Fig. 3, if an application has a 400ms SLO and the inference needs 200ms to complete, the retraining is executed for 200ms. If the inference needs 300ms to complete, the retraining is executed for 100ms. To gain more time for retraining within the SLO, we propose using early-exit structures [11].

In this paper, we first conducted real trace-driven experimental analysis for the multi-model application scenario running both inference and retraining tasks for each model (§2). As far as we know, this is the first study of its kind and the key findings include:

(1) Not all models of an application are impacted by data drift, and different models are impacted by different degrees.
(2) Incremental retraining can significantly improve accuracy.
(3) Early-exit structures can help improve accuracy by saving more time for retraining.
(4) CPU-GPU memory communication constitutes a significant portion of the inference latency, and different data types have different reuse time latencies.

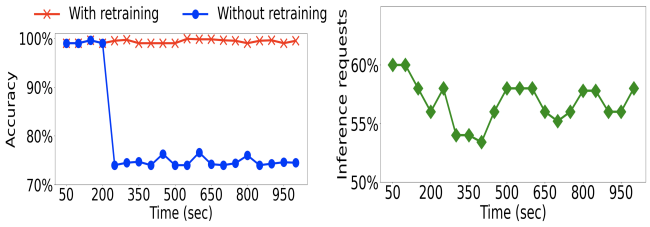
By leveraging these observations, we propose a data drift Adaptive scheduler for accurate and SLO-guaranteed Inference serving at edge servers (`AdaInf`) for multi-model applications. `AdaInf` novelly uses the incremental retraining. Specifically, it consists of the following major methods.

(i) **Data drift-aware retraining-inference DAG generation (§3.2).** `AdaInf` periodically (e.g., 50s) identifies the DNN models of each application that are impacted by data drift and their impact degrees, which are used to determine the number of retraining samples to expedite retraining while retaining accuracy. It builds a DAG that incorporates both retraining tasks and inference tasks for each application to facilitate GPU resource allocation and task executions.

(ii) **Data drift-aware GPU space and time allocation (§3.3).** Before each time session (e.g., 5ms), `AdaInf` allocates GPU resource to the tasks of all applications in the session. Specifically, `AdaInf` splits the total GPU computation space (e.g., 4 GPUs) among the applications based on their SLOs. For each application, it splits the GPU time (i.e., latency SLO) between the retraining tasks and inference tasks (based on the incremental retraining idea). Then, it further splits the GPU time for retraining among the retraining tasks based on their impact degrees and decides the optimal request batch size for the inference tasks to minimize inference latency. Additionally, `AdaInf` chooses an optimal early-exist structure of each application to leave more time for retraining to increase accuracy.

(iii) **CPU-GPU memory communication minimization (§3.4).** Contrary to the existing works [12–17] that focus on minimizing the CPU-GPU memory communication for the training of a single-model application, `AdaInf` leverages the interdependency between multiple retraining and inference tasks of a multi-model application for this purpose. Specifically, it maximizes the usage of GPU memory contents before they are evicted to the CPU memory, and first evicts GPU memory contents that will be reused later.

Our extensive real trace-driven experimental evaluation (§5) shows that `AdaInf` increases up to 21% accuracy and reduces up to 54% SLO violation compared to existing methods, with a 2ms scheduling time. The current method requires four times the amount of GPU resource on the edge server to achieve a comparable accuracy of 96% as `AdaInf`. Note that `AdaInf` is also applicable to single-model applications. *This work does not raise any ethical issues.*



(a) Impact of data drift on accuracy **(b) Inference requests using updated model in Ekya**

Figure 4: Impact of data drift on the application.

2 EXPERIMENTAL ANALYSIS

We conducted experimental analysis based on the video surveillance application shown in Fig. 1. As [3], we used the TinyYOLOv3 model for object detection, the MobileNetV2 model for vehicle type recognition, and the ShuffleNet model for person activity recognition. We used the Jackson Hole dataset [11], which contains 200 hours of images from a real video surveillance system deployed in Jackson Hole, Wyoming, USA. The first 40% of the dataset was used to initially train the models. The inference requests were generated for 1000s, where each request is an image from the dataset. The experiments were conducted on an AWS EC2 instance of type p3.2xlarge. It has 1 Nvidia V100 GPU with 5760 computation cores, 16 GB GPU memory, and 2.3 GHz Intel processor with 8 CPU cores and 61 GB of CPU memory. Unless otherwise specified, in each 50s time period, we executed the retrainings of all the models first and then all the inference tasks. As [18], the inference request rate followed the Twitter trace [19], which resembles real-world inference workload. The percentage of all inference requests (for vehicle type and person activity outputs in Fig. 1) in a time period that are predicted correctly is the accuracy of the time period.

In our experimental analysis, we measured Ekya [3] as it is the related work closest to `AdaInf`. To handle data drift, Ekya aims to satisfy the resource demands of both retraining and inference tasks and maximize the average accuracy for single-model applications. It achieves this by using a heuristic that moves resources between tasks if this increases the average accuracy and finally chooses the solution with the maximum average inference accuracy. It also determines the configuration for each task such as the number of iterations for retraining and the video frame resolution for inference. However, Ekya executes a retraining task entirely, so during the retraining time, any inference request received cannot benefit from the retrained models [3], leading to decreased accuracy. Furthermore, Ekya is not currently equipped to handle multi-model applications.

2.1 Impact of Data Drift on Accuracy

Fig. 4a shows the accuracy of each time period with and without retraining. The video surveillance application with retraining provides 0%-27% higher accuracy than that without retraining. This is because street situation may rapidly change. For example, at a certain timestamp, most of the vehicles may be personal cars, but after a short interval, e.g., 50 seconds, most of the vehicles are police cars and ambulances due to an accident, thus changing the distribution of the vehicle types. The model that is trained only on the initial training data cannot capture the distribution change, thus resulting in low accuracy. This result indicates the importance of performing retrainings at short time intervals to handle data drift. We used Ekya to represent the continual learning methods. Fig. 4b shows that only 53%-60% inference requests use the updated model at each time period in Ekya.

Observation 1: There is a sizable drop in the inference accuracy due to data drift but only 53%-60% inference requests can use the retrained model.

Fig. 5 shows the accuracy of the three models in the application in each time period with and without retraining. Fig. 6 shows the Jensen-Shannon divergence [20] of the class label distributions of the tasks in consecutive time periods, indicating the data distribution change over time [21]. The object detection model did not suffer accuracy loss from data drift while the person activity and vehicle type recognition models experienced around 0%-9% and 0%-15% accuracy loss, respectively. This can be attributed to the data distribution change from Fig. 6. The figure shows that for the object detection task, the overall distribution between vehicles and persons (i.e., the percentages of objects that fall under the vehicle category and the person category) remained almost constant across all time periods, and the data drift changed the distribution of vehicle types by 0.1%-26% more than that of person activities over the time periods.

Observation 2: Not all models of an application are impacted by data drift.

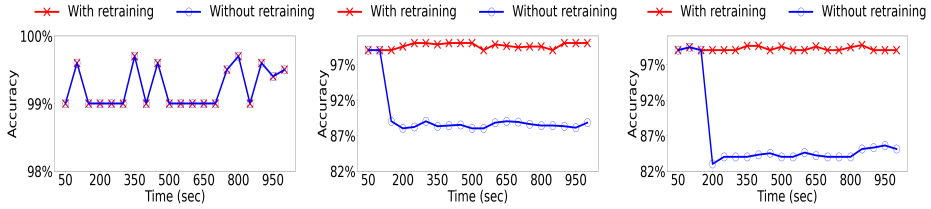
Our results show that the vehicle type recognition model suffered around 0%-6% more accuracy loss compared to the person activity recognition model.

Observation 3: In an application, among the models impacted by data drift, different models are impacted differently.

2.2 Early-Exit Structure for Inference

For this experiment, we considered the inference requests of an application received within a time session of 5ms as a job [10], which has 400ms SLO. We created all possible early-exit structures of the application, where each structure includes an early-exit structure for each model of the application. We created the early-exit structures of a model by choosing the layer after every 3 layers of the full structure as an early-exit point as in [22]. There were total 81 early-exit structures of the application. Due to space limit, we only report the performance of the early-exit structure that achieves the highest accuracy. It is possible to develop a framework to automate the creation of the early-exit structures of a model. It is out of the scope of this paper.

We tested three methods: Early-exit structure with incremental retraining (Early-inc), Full structure with incremental retraining (Full-inc), and Early-exit structure without any retraining (Early-w/o). Fig. 7a shows the accuracy of these methods at each time period. Fig. 7b shows the total retraining time and percent of retraining samples used in retraining at each time period of Early-inc and Ekya. Full-inc achieves 0%-4% higher accuracy than Ekya since in Ekya, the inference requests received before the retraining completion time (20-23s from Fig. 7b) cannot use the retrained model. The incremental retraining enables each job to use a model that has been retrained to a certain extent, leading to higher accuracy.



(a) Object detection model (b) Person activity recognition model (c) Vehicle type recognition model

Figure 5: Impact of data drift on each model of the application.

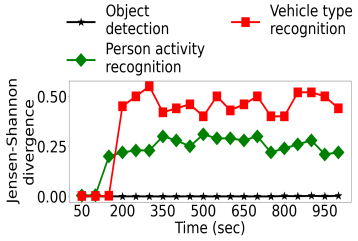
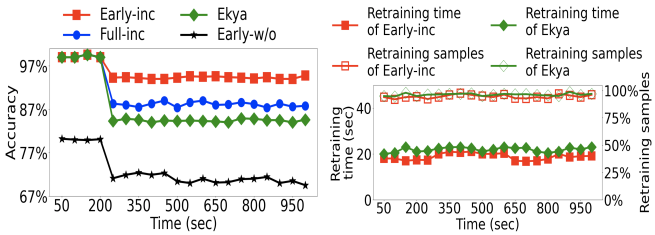


Figure 6: Change in data distribution across time.



(a) Accuracy (b) Retraining time and percentage of retraining samples

Figure 7: Early-exit structure with incremental retraining provides better accuracy.

Early-inc achieves 0%-6% higher accuracy than Full-inc, 0%-10% higher accuracy than Ekya, and 21%-23% higher accuracy than Early-w/o. These results show that though an early-exit structure reduces accuracy, using the spare time gained from the shorter inference latency of the early-exit structure for incremental retraining increases the accuracy significantly. Also, retraining is important to ensure high accuracy.

The total retraining time and percentage of retraining samples at a time period of Early-inc are close to those of Ekya. This shows that by performing retraining incrementally in multiple steps at a time period, incremental retraining ultimately does not receive less time for retraining compared to continual learning.

Overall, the results show that it is possible to find an early-exit structure to improve accuracy in incremental retraining. Note that this observation might not hold for a randomly chosen early-exit structure. Hence, we need to carefully choose the optimal early-exit structure as indicated in §3.3.2.

Observation 4: Incremental retraining improves accuracy, and using an optimal early-exit structure with incremental retraining could significantly improve accuracy.

2.3 Optimal Request Batch Size

In order to maximize the resource utilization of GPU, the requests in a job are executed in batches [23]. The worst-case latency [23] refers to the time required to complete all the request batches in a job. Fig. 8 illustrates the average per-batch latency and the average worst-case latency for different request batch sizes. The figure shows that there exists an optimal request batch size of 16 which generates the lowest worst-case latency.

Observation 5: In a multi-model application scenario, there exists an optimal request batch size that minimizes the worst-case latency.

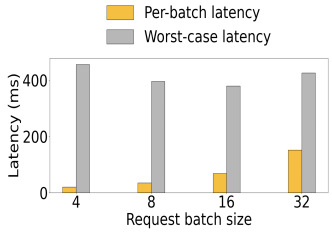


Figure 8: Average latency at a time session vs. request batch size.

Fig. 9 shows the average worst-case latency for different request batch sizes when the allocated GPU space changes. The optimal batch size is 4, 8, 16, and 16 when the allocated GPU computation space (space in short) equals to 25%, 50%, 75%, and 100%, respectively. Fig. 10 shows the average worst-case latency for different request batch sizes for the full structure and three randomly chosen early-exit structures of the application. The optimal batch size is 16, 32, 32, and 4 for the different structures, respectively.

Observation 6: The optimal request batch size is influenced by the allocated GPU space and the early-exit structure of an application.

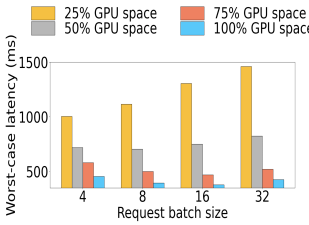


Figure 9: Average latency at a time session with varying GPU space.

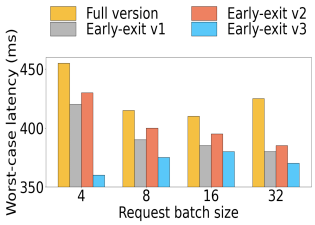


Figure 10: Average latency at a time session with varying structures.

2.4 GPU Memory Communications

GPU memory contention is common in DNN jobs, requiring some contents to be evicted to CPU memory. Existing work on CPU-GPU memory communication [12–14] or memory content reuse [15–17] only focuses on single-model training and does not consider multi-model applications or concurrent running of inference and retraining. Our scenario is uniquely featured by multi-model applications including both retraining and inference tasks of multiple models with interdependencies expressed by a DAG. As [17], we call the inputs to the first layer and the outputs generated by the layers (i.e., first, hidden, and last layer) as intermediate outputs. In our scenario, the GPU memory is more likely to be insufficient to host all the parameters and intermediate outputs generated by the tasks that are needed later on, thus causing CPU-GPU memory

communications. First, parameter values updated by the retraining of a model will be used during the inference of the same model. Second, the intermediate output generated by the last layer during the inference of a model will be used in the inference of the next model(s) in the DAG sequence. In addition, different inference requests in a request batch run the same models in the DAG. Also, the next job of an application may reuse the contents of the previous job of the application.

Fig. 11 decomposes the per-batch inference latency (shown in Fig. 8) into communication time between CPU memory and GPU memory and the actual computation time in GPU space for different batch sizes. The communication time accounts for around 24% of the latency. For inference in a single-model application scenario, this time accounts for around 17% of the latency [17].

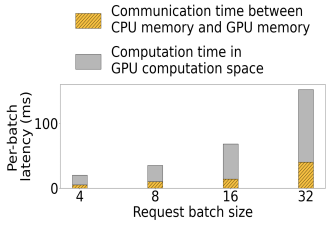
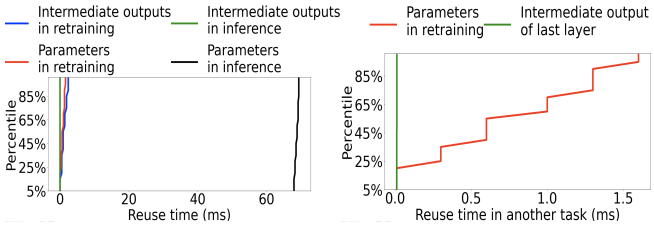


Figure 11: CPU-GPU memory communication latency.

Observation 7: When executing the retraining and inference tasks of multi-model applications in an edge server, the CPU-GPU memory communication time constitutes a significant portion of the overall inference latency.

Next, we study the memory content reuse in the Early-inc method. While the GPU was retraining the models of the application, we chose a random timestamp and measured the time latency after which each content (i.e., a layer's intermediate output or parameter values) in the GPU memory at that timestamp is reused by any model. We did the same while the GPU was executing the inference of the models. Fig. 12a shows the CDF of the reuse time of different memory contents. We see that the intermediate outputs in inference, parameters in retraining, intermediate outputs in retraining, and parameters in inference are reused within 0.01ms-1.6ms, 0.02ms-6ms, 0.02ms-7.5ms, and 67ms-68.6ms, respectively.



(a) Reuse time of memory contents. **(b) Reuse time of memory contents across tasks in a DAG.**

Figure 12: Reuse time latency of memory contents.

Fig. 12b shows the CDF of the reuse time latency of the memory contents between tasks in the DAG. The updated parameter values from the retraining of the vehicle recognition model were reused by its inference task, and the last layer's intermediate output of the object detection model was reused by the inference of the vehicle recognition model. The intermediate output and updated parameters are reused within 0.01ms-0.02ms and 0.01ms-1.65ms, respectively.

Observation 8: Memory content reuse occurs not only inside one model (between retraining and inference) but also between the dependent tasks in a DAG. Also, the reuse time latencies of different types of data from different tasks in a DAG are different.

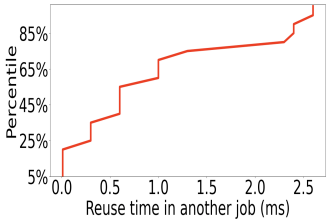


Figure 13: Reuse time of parameters across jobs.

Next, we study the reuse of the memory contents of a job by the next job of the same application in Early-inc. When the first job is completed, we measured when the current memory contents are reused. Fig. 13 shows the CDF of the reuse time latency for the parameters of the first job. We do not show the intermediate outputs used in retraining or inference of the first job here as we found they were never reused.

Observation 9: The parameters from a job will be reused by the next job but the intermediate outputs will not be reused.

3 SYSTEM DESIGN OF AdaInf

3.1 Overview

Motivated by *Observation 1*, and by leveraging other observations, we propose `AdaInf`. For each application, at the beginning of each time period T (e.g., 50s), `AdaInf` calculates the impact degree of each model and generates a retraining-inference DAG to facilitate retraining and inference task resource scheduling and execution, as shown in Fig. 14. Before each time session (e.g., 5ms), `AdaInf` schedules resource allocation to inference requests in the time session (predicted based on request rate as in [10]) and their model retrainings. The scheduling takes around 2ms (shown in §5), so at timestamp τ, `AdaInf` performs scheduling for time session $[\tau+2, \tau+7)$ms. A time session has multiple jobs and each job is for one application. In a job, each model has an inference task (consisting of all requests in the time session) and possibly a retraining task. `AdaInf` uses spare time after inference task execution for incremental retraining, while satisfying its SLO.

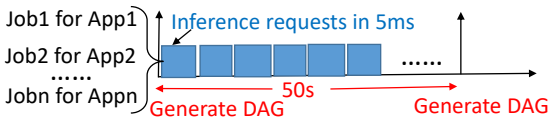


Figure 14: Time periods for generating DAGs and time sessions for scheduling.

In scheduling, `AdaInf` decides:

(1) the request batch size for each job,
(2) how much GPU space is allocated to each job to be used by each task (including retraining and inference) in the job's DAG,
(3) the optimal DNN structure of each inference task, and GPU time allocated to the retraining task and the inference task of every model in a job,
(4) and the number of retraining samples and which samples to use for each retraining task given allocated GPU space and time.

AdaInf has three major methods detailed as follows.

3.2 Data Drift-aware Retraining-Inference DAG Generation

Determining Data Drift Impact. Guided by *Observation 2*, for each application, AdaInf periodically judges whether each model will be impacted by data drift in the new training data (generated in the previous time period) and hence the necessity to retrain the model, and calculates the impact degrees of the models that need retraining (based on *Observation 3*). To do this, AdaInf first identifies S (a pre-defined value) new training samples that diverge the most from the old training samples from time period $T-1$. These samples are more prone to be misclassified by the current model [8]. To find these samples, AdaInf calculates the Cosine distance of the feature vector of every new sample with the mean feature vector of the old training samples, and chooses the S new training samples whose Cosine distances are the highest. As the feature vectors are high-dimensional, before calculating the Cosine distances, AdaInf reduces the number of features by applying principal component analysis [24] on the vectors to get more accurate distance results. Next, AdaInf gets the prediction output of these S samples using the current full structure of model m and calculates its accuracy I'_m. Let us denote the accuracy of the initially trained version of model m by I_m. If $I'_m < I_m$, AdaInf decides that model m will be impacted by the data drift and needs retraining. To ensure the correctness of the process identifying the models requiring retraining, AdaInf then gradually increases the value of S each time to repeat the process. It stops when there is no change in the results for consecutive n (e.g., 4) times. Finally, for the identified models, AdaInf calculates the impact degree of each model m by $I_m - I'_m$.

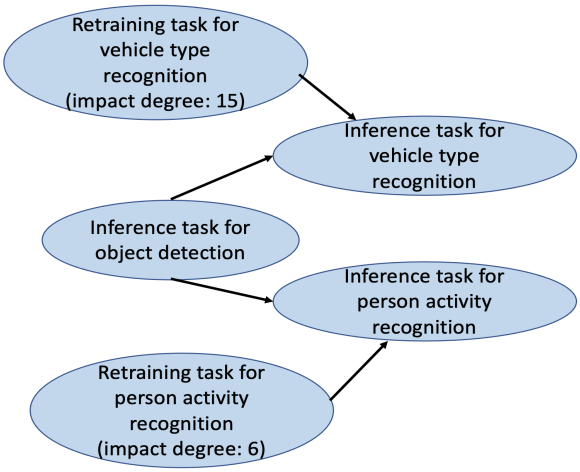


Figure 15: Retraining-inference DAG of the video surveillance application.

Unlike previous continual learning that retrains all models and uses all retraining samples to retrain model m, AdaInf does not retrain the models that will not be impacted by the data drift and determines the number of retraining samples for the retraining of model m based on its impact degree to reduce the negative impact on the accuracy from the data drift (details are in §3.3).

Generating Retraining-Inference DAG. Unlike previous work [3] in which many inference requests cannot use the retrained model, AdaInf aims to retrain a model as much as possible before using it for the next inference task in order to increase inference accuracy while meeting the SLO. To this end, for each application, AdaInf forms a DAG to denote the dependency between the retraining and inference tasks of multiple models, as illustrated in Fig. 15. Each vertex of the DAG represents a retraining task or an inference task of a DNN model of the application. A model's retraining task always points to its inference task. Note that if model m does not need retraining at the time period, the DAG does not have its retraining task. The retraining vertex of a model has an attribute "impact degree". During each time session, the tasks in a job are executed based on the DAG, and each task uses its allocated GPU space and time resources determined by the methods introduced in §3.3 and the execution latency is further reduced by the methods presented in §3.4.

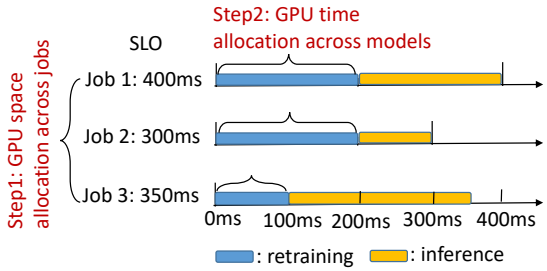


Figure 16: Steps of GPU space and time allocation.

3.3 Data Drift-aware GPU Space and Time Allocation

Before each time session t, AdaInf conducts scheduling (for (1)-(4) in §3.1) for retraining tasks and inference tasks predicted to occur in the time session. AdaInf has two steps in the scheduling as shown in Fig. 16.

(i) **GPU space division among applications (§3.3.1).** First, for the jobs at time session t, AdaInf divides the GPU resource space among the jobs (i.e., applications) based on their required GPU resource space to satisfy SLOs.

(ii) **GPU time division among the DAG vertices of an application (§3.3.2).** Second, for each job, AdaInf splits the GPU time of the allocated resource between retraining and inference with the constraint of satisfying the SLO, further splits the GPU time among retraining tasks based on their impact degrees, and then determines training samples for each retraining task accordingly and the optimal request batch size for the inference tasks. To leave enough time for retraining, AdaInf chooses an early-exit model structure for inference to satisfy its SLO without compromising accuracy.

3.3.1 GPU Space Division among Applications. Let's use $T_a$ms to denote the average time to complete all inference requests in a time session. Then, since each time session equals 5ms, there will be $s = T_a/5$ time sessions with concurrently running jobs. Hence, AdaInf distributes the total GPU resource (denoted by G) evenly among the s sessions. For each session, AdaInf then distributes the

allocated GPU space to different jobs in proportion to each job's required GPU space to complete all of its requests to satisfy its SLO. To determine the GPU space required by a job, AdaInf first finds the time latency of the job if it is allocated with one GPU and then scales the GPU space to satisfy its SLO. The details of this approach are presented below.

Based on *Observation 5*, for each application, AdaInf performs offline profiling to find an application's per-batch inference latency based on the initial DAG of the application that does not include any retraining task (e.g., the DAG in Fig. 1) for a set of request batch sizes when it is allocated with an entire GPU. Based on this profiling, during scheduling, AdaInf calculates the worst-case inference latency for a job for every request batch size when an entire GPU is allocated to the job. It equals the product of the per-batch inference latency and the total number of batches. Then, AdaInf chooses the request batch size that results in the lowest worst-case inference latency of job i, denoted by L^i_w.

Next, AdaInf calculates the required GPU space of each job i (denoted by G^i) to adjust the worst-case inference latency from L^i_w to its latency SLO (denoted by L^i_s) using a non-linear regression model as described in [3]. The model takes the chosen batch size, L^i_w and L^i_s as inputs, and outputs G^i. Finally, job i receives $\frac{G^i}{\sum_i G^i}$ fraction of the GPU resource allocated to time session t, denoted by G^i_a.

Observation 6 shows that the allocated GPU space affects the optimal batch size. Therefore, AdaInf adjusts the batch size of the job given the allocated GPU space. To do this, AdaInf iterates through the set of request batch sizes to scale the worst-case latency when the GPU space changes from an entire GPU to the actually allocated GPU space using a regression model as described in [3]. The regression model takes the batch size, the actually allocated GPU space, and worst-case latency for the batch size with an entire GPU allocation as inputs, and outputs the scaled worst-case latency for the batch size. Finally, AdaInf selects the request batch size that provides the lowest-scaled worst-case latency.

3.3.2 *GPU Time Division among the DAG Vertices of an Application.*

The execution of the tasks of a job follows the sequence in its DAG as shown in Fig. 15 and each task uses its allocated resource space G^i_a. AdaInf splits the total SLO latency of the job among the DAG vertices. It first splits the SLO latency between the inference operation and retraining operation. Then, it further distributes the allocated time for retaining operation among the retraining tasks in order to increase accuracy. The details of these two steps will be presented in the following after we explain the selection of the early-exit structure.

Based on *Observation 4*, AdaInf employs the early-exit technique. For each DNN model of an application, AdaInf creates several early-exit structures offline in the same manner as described in [22]. Then, AdaInf performs offline profiling to find the per-batch inference latency of every early-exit structure of a model for a set of request batch sizes if an entire GPU is allocated to the structure. Also, AdaInf stores the accuracy of every early-exit structure on the initial test data, and the stored accuracy is updated by the accuracy of the structure on the S new training samples (introduced in §3.2) at each time period.

During scheduling, AdaInf traverses each inference task vertex of the DAG of an application and chooses a structure for it. Suppose m is the corresponding model. If there is no retraining task vertex in the DAG for model m, AdaInf uses the full structure of m during inference execution since it does not need to save time for retraining. Otherwise, AdaInf first takes only those early-exit structures of model m, whose accuracy values are no less than the application-specific pre-known threshold A_m (the impact of A_m is discussed in §5.3) and selects the one with the lowest per-batch inference latency. To do this, for each structure, AdaInf uses a similar non-linear model as described previously to scale the profiled per-batch inference latency when the GPU space changes from the entire GPU to the actually allocated GPU space.

After a structure is chosen for the model of each inference task, we get an early-exit structure of the whole application. Based on *Observation 6*, AdaInf adjusts the batch size of the application given the chosen early-exit structure using a non-linear model as mentioned above. Finally, AdaInf selects the request batch size that provides the lowest-scaled worst-case latency.

Next, AdaInf calculates the total inference time of the job, and then the spare time in the SLO latency used for retraining in order to satisfy the SLO while maximizing accuracy. We present the details below. After allocating GPU space to an inference job, for the parallel tasks in the DAG, the GPU kernel schedules these tasks in a time-sliced manner [25] in the allocated space. Therefore, the total time taken by the set of all inference tasks of the job i (denoted by I^i) is calculated by $\sum_{k\in I^i} l^i_k$, where l^i_k is the latency execution of the k^{th} inference task of job i. Then, the total spare GPU time that is allocated to the retraining tasks of job i equals: $T^i_r = L^i_s - \sum_{k\in I^i} l^i_k$.

Then, this remaining time is divided among the retraining tasks according to their impact degrees to increase accuracy. That is, retraining tasks with higher impact degrees receive more time and training samples for retraining, thus mitigating more influence from data drift. Specifically, if the degree is d^i_j for the j^{th} retraining task of job i, its allocated time is $T^i_r \cdot \frac{d^i_j}{\sum_j d^i_j}$.

Based on the allocated GPU time to a retraining task, AdaInf then decides a retraining setting (i.e., the number of retraining samples, retraining data batch size, and the number of epochs) to maximize accuracy. To do this, AdaInf performs an offline profiling to find the retraining latency and accuracy for each retraining setting when an entire GPU is allocated to retrain the model. Based on this profiling and the allocated GPU time for the retraining task, during scheduling, AdaInf chooses a retraining setting for the retraining task using a non-linear model as mentioned above. Suppose the number of retraining samples is R^i_j for the j^{th} retraining task of job i, then AdaInf selects R^i_j samples from the new retraining samples that deviate the most from the old training samples using the approach introduced in §3.2.

Now, due to concurrently running jobs (in different sessions) of an application, when a job i starts to retrain a model of the application, there may be other jobs that have already completed or are retraining the same model. In such a case, job i starts the retraining by taking the average of the current parameter values of the different versions of the model as the initial parameter values to generate better accuracy [26]. To avoid redundant training, the job

does not use retraining samples that have been used or are being used by other jobs.

3.4 CPU-GPU Memory Communication Minimization

At a time, the GPUs on an edge server concurrently run the jobs from different time sessions, each of which has different jobs. Each job has several inference requests and retraining task for each of its models. All the tasks running on a GPU share the same GPU memory, which is limited. To address the issue in *Observation 7*, we leverage the features of the scenario of both retraining and inference running for multi-model applications (*Observation 8* and *Observation 9*) to propose the following strategies to reduce the CPU-GPU memory communications and decrease latency.

3.4.1 Maximizing GPU Memory Usage. Different inference requests in a batch run the same models in the DAG of the application. Therefore, all requests use the same parameter values of a model layer. Each request is executed concurrently with other requests, without knowledge about which model layers are executed in each of the other requests. When a request completes its execution of a model layer, the layer's parameters may be sent to the CPU memory due to lack of space in GPU memory. After some time, another request has to fetch them from the CPU memory to the GPU memory.

To avoid such CPU-GPU memory communications, `AdaInf` runs the execution of a single model layer for all the requests in a batch at the same time. This way, all the inference executions of a model layer are completed before the parameters of the layer are evicted to CPU memory. `AdaInf` applies the same strategy for the retraining batch while retraining a model. In model retraining, all retraining samples in a batch use the same parameter values of the model. `AdaInf` does not evict the parameters of a model layer from GPU memory until the layer is retrained on all the samples in the batch.

In addition, based on *Observation 9*, when a job completes, `AdaInf` evicts all intermediate outputs of the job but retains the updated parameters of the job, as they may be needed again by the next job of the same application.

Consequently, in order to release enough space in the GPU memory, `AdaInf` prioritizes evicting the contents that should be evicted first and leaves the contents that should not be evicted until the end. However, determining which contents should be selected for eviction in each category is non-trivial. To address this issue, we propose a method described below.

3.4.2 Priority-based Eviction of GPU Memory Contents. Considering *Observation 8* and the fact that the latency SLOs of different applications can vary, we propose a method for determining the priority of contents to remain in GPU memory. Specifically, `AdaInf` prioritizes the memory contents that will be used much earlier than other contents and/or have tighter SLOs to remain in the GPU memory. `AdaInf` determines task priority based on various factors, including the causes for data reuse as described in §2.4. Specifically, `AdaInf` takes into account the interdependency between tasks in the DAG, as well as the data type and content reuse time latency.

We use R_c to denote the reuse time latency of content unit c, which can be estimated using profiling, as described in [15]. Then, the score of content c is calculated as follows: $S_c = (1-\alpha)R_c + \alpha L_s^i$, where $\alpha \in (0, 1)$ is a weight. Contents with higher scores are evicted from the GPU memory. However, calculating S_c for each content and finding the contents with the highest S_c values take time. To reduce the time, guided by *Observation 8*, `AdaInf` chooses a certain data type (as shown in Fig. 12) to evict first and then among the contents of this type, it chooses those with the highest S_c values for eviction. Note that the data type here is also distinguished by its task in the DAG. Through offline analysis for each application, `AdaInf` obtains a range for the reuse time latency of each data type. `AdaInf` takes the mean value of the range as the value of R_c of the data type. Thus, each data type also has a S_c value, and those types with higher S_c values will be chosen for eviction first.

Among the evicted memory contents, lower-scoring contents are kept in the PIN memory portion of the CPU memory so the contents can be sent back to the GPU memory earlier than higher-scoring memory contents. PIN memory is a small portion of the CPU memory that takes less time to communicate with the GPU memory compared to the remaining portion of the CPU memory [13].

Reduced CPU-GPU memory communications not only helps to reduce task latency, but also saves more time for retraining. To integrate these strategies into the scheduling in §3.3, `AdaInf` uses these strategies in offline profilings mentioned in §3.3. Hence, the profilings help to calculate the inference task latency with these strategies, and then the spare time in SLO latency for retraining.

4 EXPERIMENTAL SETUP AND IMPLEMENTATION

Applications and Datasets. Unless otherwise specified, the experiment settings are the same as those in §2. The latency SLO of each application (with a range [400, 600]ms) was taken from [10]. By default, we experimented with eight concurrent applications, which include the video surveillance application described in §2, six applications from [10], and one application from [27] that has a complex DAG. The datasets of the applications were from [10, 27–33].

The applications are shown in Fig. 17. The social media application has a more complex DAG than the other applications. The application categorizes posts (i.e., safe for viewing or not) based on the linked images and suggests a person to tag in the images using an image recognition model. Additionally, the application translates the post language to English if necessary using a language translation model. In each oval, we show the model name in bracket for the corresponding inference task. The MobileNet and ShuffleNet models are designed to fit into the limited resources of end devices and edge servers. We compressed the remaining models using DeepSpeed compression library [34].

For the experiment with a varying number of applications, we obtained the additional applications and datasets from [23] as follows.

- *Analyzing video games:* It first runs object detection using the SSDLite model, and then runs text recognition using STN-OCR and object recognition using ResNet18.
- *Rating dance performance:* It first runs person detection using TinyYOLOv3, and then runs pose recognition using Shufflenet.
- *Estimating response to public bill boards:* It first runs object detection using SSDLite, and then runs face recognition using MobileNetv2 and gaze recognition using ResNet18.

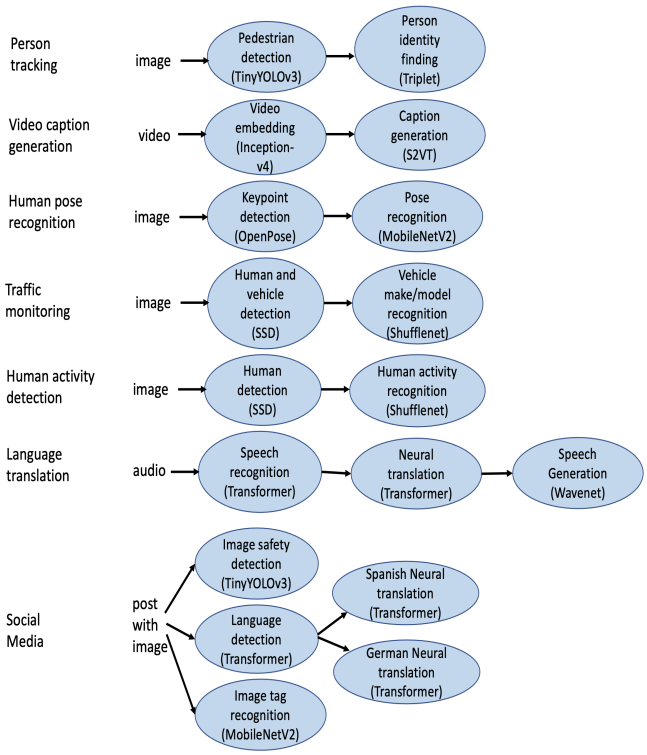


Figure 17: DAGs of the applications in the experiments.

• *Finding bike-rack occupancy on buses:* It runs object detection using TinyYOLOv3.
• *Matching vehicle to amber alert description:* It first runs text recognition using STN-OCR and object detection using SSDLite, then runs vehicle make/model recognition using ResNet18.
• *Rating corporate logo placement:* It first runs object detection using TinyYOLOv3, then runs icon recognition using MobileNetv2 and human pose recognition using Shufflenet.

Testbed. As [3], we used an Amazon AWS EC2 p3.8xlarge server with 4 GPUs as the edge server. The total memory of all the GPUs combined is 64GB, and the GPUs are interconnected via NVLinks. The server has a 2.3 GHz Intel processor with 32 CPU cores and 244 GB of CPU memory. We also varied the number of GPUs and used an AWS server of type p3.2xlarge and p3.16xlarge to perform the 1-GPU and 8-GPU experiments, respectively. We used two AWS servers of type p3.8xlarge to perform the 16-GPU experiments.

Comparison Methods.
• *Ekya [3]*: At the beginning of each 50s time period, Ekya conducts scheduling as explained in §2. It uses a heuristic that tries various possible resource allocations by moving resources between tasks and chooses the one with the maximum average inference accuracy for all applications.
• *Scrooge [10]*: For inference requests in 5ms, Scrooge solves an optimization formulation that outputs the instance type, GPU amount and batch size to satisfy latency SLOs and minimize monetary cost of the cloud instances. The optimization takes around 100ms so Scrooge schedules for all the 5ms time sessions within this 100ms. After every 50s [3], it performs retraining on the cloud, which is an AWS EC2 p3.16xlarge server with around 20Gbps bandwidth between the edge server. As Scrooge considers unlimited resource, we made two changes. First, we modified the optimization formulation by adding a constraint that the allocated GPU amount is no more than the edge server's GPU amount, and we denote this solution by Scrooge. Second, after obtaining the optimization solution, we allocated $\frac{G_i}{\sum_i G_i}$ fraction of the total GPU amount to application i, where G_i denotes its required GPU amount, and we denote this solution by Scrooge*.

Implementation. We developed `AdaInf` using C++ and Python. We used Nvidia MPS [35] to divide the GPU resource. Each application is treated as a `CUDA Context` and we used `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` to assign a percentage of the total concurrent threads to an application. The computation of one request in a request batch or one retraining in a retraining batch was executed in a `CUDA Stream`. From CUDA toolkit, we used `cudaStreamWaitEvent` to make a stream wait for the completion of a model layer execution of the other streams, `cudaMalloc` for memory allocation in GPU, `cudaMemcpyAsync` to execute CPU-GPU memory communications, `cudaMallocHost` for memory allocation in PIN memory, and `nvprof` to measure the time for CPU-GPU memory communications. All offline profilings were performed on a Nvidia V100 GPU using Nvidia profiling tool [36]. Keras and Tensorflow were used for the initial training, retraining, and inference executions. The value of α (in §3.4.2) was set to 0.4. A_m (in §3.2) was varied within [80%, 95%] for the models of different applications. S was initially set to 3% with an increase rate of 3% in each step. These parameters are empirically determined. Determination and impact of these values are discussed in §5.3.

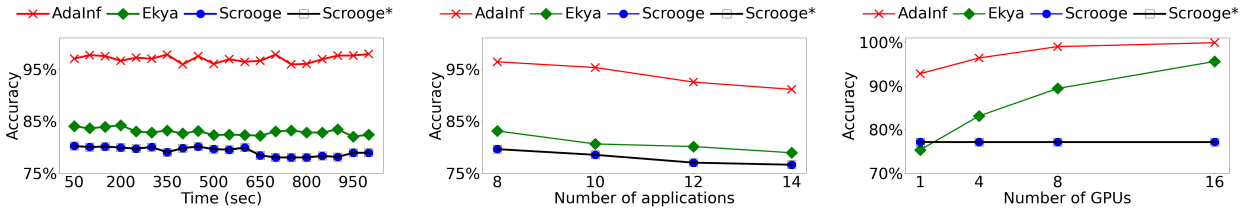
5 PERFORMANCE EVALUATION

5.1 Comparison Results

The results show that Scrooge* performs similarly to Scrooge, so we use Scrooge to represent both in the presentation below. We report the average accuracy across all applications.

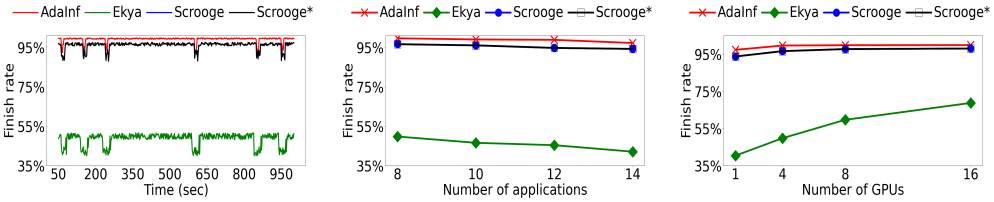
Accuracy. Fig. 18 shows the accuracy of the methods in different experiment settings. `AdaInf` provides 11%-14% higher accuracy than Ekya. This is because unlike Ekya, `AdaInf` uses the incremental retraining. Also, it uses the retraining samples that deviate the most from the previous training data. In Ekya, the inference can use the retrained model only when the model's retraining has finished on all the retraining samples (which takes 20s-23s in Fig. 7b). Also, `AdaInf` does not retrain the models that are not impacted by data drift, and also determines the retraining time of the retraining tasks based on their impact degrees. These leave more GPU time for the retraining tasks that are more impacted by data drift, thus achieving higher accuracy. Additionally, the early-exit structures and the strategies for CPU-GPU memory communication reduction reduce the inference tasks' execution latency and leave more time for retraining, which also helps achieve higher accuracy. `AdaInf` provides around 19%-21% higher accuracy than Scrooge. The long communication time between the cloud and edge server prevents many inference requests from using the retrained model. From Table 1, the transmission time takes 34.1 seconds.

Fig. 18b shows as the number of applications increases, the average accuracy (over all the time periods) decreases due to higher



(a) Accuracy vs. time. (b) Accuracy vs. no. of applications. (c) Accuracy vs. number of GPUs.

Figure 18: Accuracy comparison of different methods.



(a) Finish rate vs. time. (b) Finish rate vs. no. of applications. (c) Finish rate vs. number of GPUs.

Figure 19: Finish rate comparison of different methods.

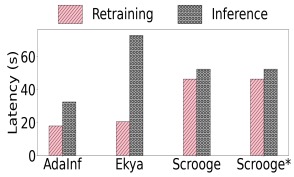


Figure 20: Latency.

workload and limited resource for retraining. Fig. 18c shows that with the increase of the number of GPUs in the edge server, the accuracies of both AdaInf and Ekya increase because of more GPU resource for retraining. However, Scrooge's accuracy does not change because its retraining is performed on cloud. We also see that only when Ekya uses 16 GPUs, its accuracy (95.6%) is similar as that of AdaInf when it uses 4 GPUs (96.4%). This implies that AdaInf achieves a four-fold higher GPU resource efficiency while maintaining a similar accuracy level of 96% as Ekya.

Finish Rate. The finish rate for an application is defined as the ratio (expressed in percentage) between the number of requests that satisfy the latency SLO and the number of requests that come to the edge server within a 1s window Fig. 19 shows the finish rate comparison of different methods in different experiment settings. AdaInf achieves around 50%-54% higher finish rate than Ekya. This is because Ekya does not focus on maximizing latency SLO fulfillment while this is a goal of AdaInf. AdaInf allocates GPU resources based on application SLOs, and early-exit structures and strategies for minimizing CPU-GPU memory communication reduce inference latency, resulting in a higher finish rate. Although Scrooge offloads the retraining tasks to the cloud, AdaInf still generates 2%-4% higher finish rate than Scrooge by using early-exit structures and reducing CPU-GPU memory communication to expedite inference. The reasons for the finish rate differences between the methods are illustrated in Fig. 20, which shows the average latency for retraining and inference, respectively, for all applications over a 50s time period.

Fig. 19b shows that as the number of applications increases, the average finish rate (over all the time periods) of each method decreases due to higher workload and GPU resource limitation. Fig. 19c shows that as the number of GPUs increases, the average finish rate of each method increases due to more GPU resources for inference. The influence on AdaInf and Scrooge is not obvious since AdaInf aims to satisfy SLOs and Scrooge offloads retraining to the cloud. The finish rate cannot reach 100% because the methods schedule for the predicted number of inference requests, which may differ from the actual number.

Time Overhead. Table 1 presents the average time overhead of different methods for executing various components. AdaInf takes 4.2s for periodical DAG update, but it does not affect the job scheduling or executions as it independently runs in the CPU. AdaInf takes 2ms and Scrooge takes 100ms for scheduling jobs in 5ms. Scrooge solves a time-consuming optimization formulation. Ekya takes 8.4s to make the scheduling decision for the jobs in 50s. It uses a time-consuming heuristic that traverses every pair of tasks to check whether transferring resource between the pair can improve accuracy. Scrooge takes 34.1s communication time to transfer 85.7GB of data between edge and cloud during each retraining. The data contents are the retraining samples sent from the edge server to the cloud and the updated models from the cloud to the edge server. AdaInf's strategies are beneficial in reducing the CPU-GPU memory communication in a job, with only a 1ms overhead.

GPU Utilization. Fig. 21 shows the utilization of the GPUs at each second for different methods. We measured the utilization using Nvidia-smi. We report the average utilization of a GPU. All methods result in close or equal to 100% utilization across time because each method executes multiple concurrent applications in the same GPU. The inference request workload of an application is usually insufficient to saturate the computational capability of a GPU [37]. However, GPU space multiplexing technologies (e.g., Nvidia MPS) enable concurrent execution

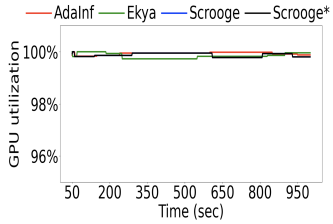


Figure 21: GPU utilization.

Table 1: Time overheads of methods.

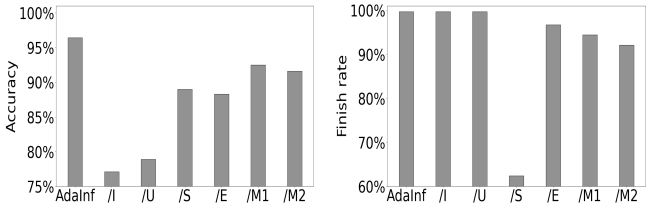
	Periodical DAG update in CPU	Scheduling in CPU	Edge-cloud communication time	Edge-cloud transferred data amount	CPU-GPU memory communication minimization
AdaInf	4.2s	2ms	0	0	1ms
Ekya	0	8.4s	0	0	0
Scrooge	0	100ms	34.1s	85.7GB	0
Scrooge*	0	100ms	34.1s	85.7GB	0

of multiple applications in a GPU, which helps fully utilize the capacity of a GPU.

5.2 Effectiveness of Each Proposed Method

We created several variants of AdaInf as follows.

- AdaInf/I: It evenly divides the spare time among all the models for retraining instead of considering impact degrees.
- AdaInf/U: It creates the retraining-inference DAG once and does not update the models' impact degrees.
- AdaInf/S: It evenly divides the GPU space allocated to a time session among the jobs in the session.
- AdaInf/E: It uses the full structure instead of the early-exit structure of a model.
- AdaInf/M1: It does not apply the strategy to maximize GPU memory usage.
- AdaInf/M2: It does not apply the strategy of priority-based eviction of GPU memory contents.



(a) Accuracy. **(b) Finish rate.**

Figure 22: Performance of different variants of AdaInf.

Accuracy. Fig. 22a shows the average accuracy across all time periods for AdaInf and its different variants. The results follow: AdaInf>AdaInf/M1>AdaInf/M2>AdaInf/S>AdaInf/E>AdaInf/U>AdaInf/I. AdaInf achieves around 4% and 5% higher accuracy than AdaInf/M1 and AdaInf/M2, respectively, which indicates the effectiveness of the strategies in reducing CPU-GPU memory communications to reduce inference latency and give more time to retraining. AdaInf achieves around 7% higher accuracy than AdaInf/S, which indicates the importance of considering the resource demands of different jobs to satisfy SLOs in resource allocation. AdaInf achieves around 8% higher accuracy than AdaInf/E, indicating the efficacy of using early-exit structures to leave more time for retraining. AdaInf achieves around 17% higher accuracy than AdaInf/U, indicating the importance of considering the dynamic change of the impact degree of a model across different time periods. AdaInf achie-ves around 18% higher accuracy than AdaInf/I, which indicates the effectiveness of considering the impact degrees.

Finish Rate. Fig. 22b presents the average finish rate across all time periods for AdaInf and its different variants. The results follow: AdaInf=AdaInf/I=AdaInf/U>AdaInf/E>AdaInf/M1>AdaInf/M2>AdaInf/S. AdaInf attains a finish rate approximately 3% higher than AdaInf/E. This result demonstrates the effectiveness of using an early-exit model in achieving higher latency SLO fulfillment. AdaInf achieves around 5% and 7% higher finish rate than AdaInf/M1 and AdaInf/M2, respectively due to the effectiveness of the two strategies for minimizing CPU-GPU memory communication. AdaInf achieves a 37% higher finish rate than AdaInf/S, indicating the importance of dividing GPU resource based on jobs' resource requirements to meet their latency SLOs. AdaInf/I and AdaInf/U achieve similar finish rates as AdaInf but incur lower accuracy as described above since they missing components help increase accuracy instead of reducing latency, and they initially allocate resources to satisfy SLOs.

5.3 Effect of Parameters

Determination of S (in §3.2). Table 2 displays the detected models impacted by data drift for the video surveillance application at the second time period. "Person" and "Vehicle" refer to the person activity recognition and vehicle type recognition models, respectively. When S=18%, the result is the same as the previous 4 values, and then AdaInf stops the process. We see the results are the same as those when S=100%. Thus, our process can correctly identify which models require retraining and it is also time-efficient as it does not check all 100% of samples. We also verified the correctness of the results from our approach for all other applications.

Table 2: Determination of parameter S

Value of S (% of retraining samples)	3%	6%	9%	12%	15%	18%	100%
Models impacted by data drift	X	Person	Person, Vehicle	Person, Vehicle	Person, Vehicle	Person, Vehicle	Person, Vehicle

Influence of α (in §3.4.2). While scoring each GPU memory content for eviction, α determines which factor should be given more importance between reuse time latency and SLO. Fig. 23 displays the average accuracy and average finish rate of AdaInf for all the applications across all the time periods for different values of α. Accuracy does not change much with the variation of the α value. For the finish rate, $\alpha = 0.4$ produces the highest finish rate. When $\alpha < 0.4$, the finish rate decreases because the SLO is given less importance. It also decreases when $\alpha > 0.4$ because the reuse time latency of a memory content is given less importance. Thus, we set

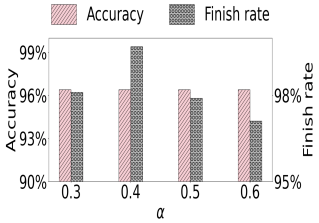


Figure 23: Influence of α.

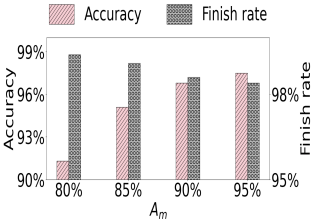


Figure 24: Influence of A_m.

$\alpha = 0.4$ since it achieves an optimal tradeoff between reuse time and SLO.

Influence of A_m (in §3.3.2). A_m is an application-specific pre-known accuracy threshold for choosing early-exit structure options of a model. Fig. 24 displays the average accuracy and average finish rate across all the time periods of the video surveillance application for varying values of A_m for the vehicle recognition model of the application. The A_m values for the object detection and person recognition models of the application were fixed at 90% and 95%, respectively. As A_m increases, the accuracy of the application increases but the finish rate decreases. This is due to the fact that as A_m increases, `AdaInf` selects an early-exit structure with higher accuracy, but this structure requires more time for inference.

6 LIMITATIONS AND DISCUSSION

Design Challenge. While allocating GPU space among applications, `AdaInf` chooses the optimal request batch size of an application assuming that it receives an entire GPU, and then adjusts the size after allocating the GPU space and choosing an early-exit structure of the application. We will explore how to directly decide the optimal request batch size without any adjustment.

DNN Execution in CPUs. `AdaInf` does not focus on executing the DNN models in CPUs. CPU execution incurs less monetary cost than GPU execution. Hence, if the number of requests is low for scheduling, we may execute the requests in CPUs, which may be enough to satisfy SLOs.

GPU Type Heterogeneity. Currently, in AdaInf, retraining and inference tasks for a job are executed within a single edge server with GPUs of the same type. We will extend `AdaInf` to allocate tasks to heterogeneous GPUs across multiple servers.

Offline Profiling. Similar to the existing inference serving systems [10, 18, 38], `AdaInf` needs to perform offline profiling. We profiled for each combination of a batch size and an early-exit structure in a separate GPU of the same architecture (Nvidia V100). Note that if we replace any of the constituent models of an application with a new model, we need to do the profiling again.

7 RELATED WORK

Inference Serving for Single-Model Applications. Various methods have been proposed for inference serving in single-model applications on the cloud or edge servers to meet latency SLOs and/or minimize costs [3, 18, 39–42]. Romero *et al.* [18] proposed a system that uses integer linear programming formulation along with pre-calculated performance-cost profiles on different hardware platforms (i.e., CPU, GPU, ASIC) to choose the optimal hardware platform for each inference request execution in order to minimize the monetary cost while satisfying SLOs. Gujarati *et al.* [39] proposed a system that drops the inference requests predicted to miss their SLOs to leave more GPU resource to other requests in order to minimize the tail latency. Gunasekaran *et al.* [40] proposed a system that finds the optimal number of models in an ensemble model to maximize accuracy and satisfy SLOs. A group of methods [41, 42] propose to use profiling to find the optimal configurations for the inference tasks of each application to maximize accuracy with minimal GPU resource usage. These methods do not consider model retraining to handle data drift. Bhardwaj *et al.* [3] proposed Ekya that jointly conducts retraining and inference serving, but it may suffer from lower accuracy and SLO violation in the multi-model applications, as verified in §5.1. All of these methods do not consider multi-model applications, which pose more formidable challenges for both retraining and inference executions, as indicated in §1. Several methods [15–17] use eviction technique by prioritizing the memory contents that will be used much earlier than other contents. However, these methods focus on single-model training instead of multi-model applications that involve both retraining and inference.

Inference Serving for Multi-Model Applications. A group of methods focus on the inference serving of multi-model applications by considering the DAG dependency [10, 23, 27, 37, 38, 43–49]. These methods also aim to satisfy SLOs and/or minimize monetary cost. Hu *et al.* [10] used an optimization formulation to decide the optimal GPU type and amount of the public cloud for each application to maximize SLO fulfillment and minimize the monetary cost. Shen *et al.* [23] proposed a modified bin-packing algorithm to allocate GPU time to multiple applications in order to maximize GPU resource utilization and SLO fulfillment. A group of methods [27, 37, 43–46, 48] use profiling to find the optimal request batch size for each application with the goal of achieving both overall high GPU utilization and SLO fulfillment of the application. Another group of methods [47, 49] use regression-based profiling to determine which models to run concurrently to minimize inter-model GPU memory interference. Zhang *et al.* [38] proposed a periodic planner that aggregates request streams into moderately-sized groups for high utilization and an online scheduler that employs a novel online algorithm to provide guaranteed SLO fulfillment. However, none of these methods focus on executing both retraining and inference of the models in the same server.

8 CONCLUSION

Edge servers are widely used to serve inference requests of multi-model applications with tight SLOs. The applications suffer from data drift and hence the models need to be retrained periodically but it causes resource contention, impacting accuracy and SLO fulfillment. To handle this, based on our experimental analysis, we propose `AdaInf` which generates data drift-aware retraining-inference DAGs, allocates GPU space and time based on data drift awareness, and minimizes CPU-GPU memory communication. Our experimental evaluations demonstrate that `AdaInf` achieves up to 21% higher accuracy, reduces latency SLO violation by up to 54%, and achieves 4× higher resource efficiency compared to existing methods. We plan to study how to balance the workloads of retraining and inference serving tasks among multiple edge servers in the future.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers and our shepherd, Ganesh Ananthanarayanan, for their invaluable feedback. This research was supported in part by U.S. NSF grants NSF-1827674, NSF-2206522, NSF-1822965, FHWA grant 693JJ31950016, Microsoft Research Faculty Fellowship 8300751, and Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation and workforce development. For more information about CCI, please visit cyberinitiative.org.

REFERENCES

[1] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *Proc. of ATC*, 2019.

[2] Darek Fanton. Edge server. https://www.onlogic.com/company/io-hub/what-are-edge-servers/, 2021.

[3] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *Proc. of NSDI*, 2022.

[4] Enrique Saurez, Harshit Gupta, Alexandros Daglis, and Umakishore Ramachandran. Oneedge: An efficient control plane for geo-distributed infrastructures. In *Proc. of SoCC*, 2021.

[5] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. Mu: an efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proc. of SoCC*, 2021.

[6] Mickael Cormier, Aris Clepe, Andreas Specker, and Jürgen Beyerer. Where are we with human pose estimation in real-world surveillance? In *Proc. of IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022.

[7] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[8] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proc. of CVPR*, 2017.

[9] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7), 2021.

[10] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proc. of SoCC*, 2021.

[11] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proc. of SenSys*, 2020.

[12] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *Proc. of OSDI*, 2020.

[13] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proc. of SC*, 2021.

[14] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient dnn training with tick-tock scheduling. *Proc. of MLSys*, 2021.

[15] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proc. of ASPLOS*, 2020.

[16] Xiaonan Nie, Xupeng Miao, Zhi Yang, and Bin Cui. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. In *Proc. of ICDE*, 2022.

[17] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. Oc-dnn: Exploiting advanced unified memory capabilities in cuda 9 and volta gpus for out-of-core dnn training. In *Proc. of HiPC*, 2018.

[18] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *Proc. of ATC*, 2021.

[19] Twitter. Twitter streaming traces. https://archive.org/details/archiveteam-twitter-stream-2018-04, 2018.

[20] Bent Fuglede and Flemming Topsoe. Jensen-shannon divergence and hilbert space embedding. In *Proc. of ISIT*, 2004.

[21] Harshit Daga, Patrick K Nicholson, Ada Gavrilovska, and Diego Lugones. Cartel: A system for collaborative transfer learning at the edge. In *Proc. of SoCC*, 2019.

[22] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proc. of MobiCom*, 2020.

[23] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proc. of SOSP*, 2019.

[24] Rasmus Bro and Age K Smilde. Principal component analysis. *Analytical methods*, 6(9):2812–2831, 2014.

[25] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. of OSDI*, 2018.

[26] Dan Hendrycks, Kimin Lee, and Mantas Mazeika. Using pre-training can improve model robustness and uncertainty. In *ICML*, 2019.

[27] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proc. of SoCC*, 2020.

[28] Alexander Hermans, Lucas Beyer, and Bastian Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737*, 2017.

[29] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proc. of ICCV*, 2015.

[30] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *Proc. of NSDI*, 2017.

[31] Xiaochen Liu, Pradipta Ghosh, Oytun Ulutan, BS Manjunath, Kevin Chan, and Ramesh Govindan. Caesar: cross-camera complex activity recognition. In *Proc. of SenSys*, 2019.

[32] Yao Feng, Fan Wu, Xiaohu Shao, Yanfeng Wang, and Xi Zhou. Joint 3d face reconstruction and dense alignment with position map regression network. In *Proc. of ECCV*, 2018.

[33] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. Rim: Offloading inference to the edge. In *Proc. of IoTDI*, 2021.

[34] DeepSpeed. Deepspeed compression library. https://www.deepspeed.ai/compression/, 2023.

[35] Nvidia. Nvidia multi-process service (mps). https://docs.nvidia.com/deploy/mps/index.html, 2021.

[36] Nvidia. Nvidia profiling tool. https://developer.nvidia.com/nvidia-system-management-interface, 2012.

[37] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *Proc. of NSDI*, 2023.

[38] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *Proc. of NSDI*, 2023.

[39] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *Proc. of OSDI*, 2020.

[40] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *Proc. of NSDI*, 2022.

[41] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proc. of DAC*, 2022.

[42] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proc. of SIGCOMM*, 2018.

[43] Vinod Nigade, Ramon Winder, Henri Bal, and Lin Wang. Better never than late: Timely edge video analytics over the air. In *Proc. of SenSys*, 2021.

[44] Zhou Fang, Dezhi Hong, and Rajesh K Gupta. Serving deep neural networks at the cloud edge for vision applications on mobile platforms. In *Proc. of MMSys*, 2019.

[45] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proc. of EuroSys*, 2019.

[46] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *Proc. of NSDI*, 2017.

[47] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *Proc. of ATC*, 2022.

[48] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *Proc. of NSDI*, 2022.

[49] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *Proc. of OSDI*, 2022.