QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks

Zhou Fang
University of California San Diego

Tong Yu
Carnegie Mellon University

Ole J. Mengshoel
Carnegie Mellon University

Rajesh K. Gupta
University of California San Diego

ABSTRACT

Deep neural networks (DNNs) are popular in diverse fields such as computer vision and natural language processing. DNN inference tasks are emerging as a service provided by cloud computing environments. However, cloud-hosted DNN inference faces new challenges in workload scheduling for the best Quality of Service (QoS), due to dependence on batch size, model complexity and resource allocation. This paper represents the QoS metric as a utility function of response delay and inference accuracy. We first propose a simple and effective heuristic approach that keeps low response delay and satisfies the requirement on processing throughput. Then we describe an advanced deep reinforcement learning (RL) approach that learns to schedule from experience. The RL scheduler is trained to maximize QoS, using a set of system statuses as the input to the RL policy model. Our approach performs scheduling actions only when there are free GPUs, thus reduces scheduling overhead over common RL schedulers that run at every continuous time step. We evaluate the schedulers on a simulation platform and demonstrate the advantages of RL over heuristics.

1 INTRODUCTION

Deep neural networks (DNNs) have found widespread use due to their versatility and demonstrated value. Problems in efficient deployment of DNN models are starting to attract research interest from both academia and industry [5, 11]. DNN *inference* refers to the phase of forward propagation through the DNN model that returns prediction results. Despite excellent performance of DNN models, high overhead of computation and memory makes their deployment on the client-end a challenging task, especially for resource limited mobile platforms such as smartphones and wearable devices. Executing DNNs on cloud-hosted GPU servers provides an attractive alternative to reduce the latency of inference. DNN inference as a cloud service also simplifies application development as well as DNN model management.[1]

This paper studies scheduling of DNN inference queries on servers that require sub-second delay. We consider two Quality of Service (QoS) metrics for query responses: ***accuracy*** and ***delay***. The scheduler dynamically selects a batch size and a DNN model to process on a GPU in a cluster. Its goal is to maximize the real-time system performance that is a utility function of inference accuracy, end-to-end (E2E) delay, and query deadline.

Accuracy in cloud computing is represented differently for different types of workload, such like quality of web response [9], errors of big data analytic query [1], data staleness [13], and so on. For DNN inference tasks, for example, accuracy is measured by errors of predicted labels for object recognition tasks, and localization errors are considered in addition for detection tasks. A higher inference accuracy usually requires more complex networks, thus a longer processing delay. The scheduler should handle **model selection** to maximize QoS of query responses considering the two contradictory objectives: accuracy and delay.

Furthermore, batching inputs can effectively improve the throughput of DNNs, because of better utilization of the parallel computing resources, and less data copy between CPU and GPU memories [11]. Because batching leads to higher processing latency in the inference phase, the scheduler must consider **batch size selection** to satisfy the requirements on both latency and throughput. Although adaptive batching for big data workloads has been studied [2, 3], these works do not consider tasks with sub-second processing delay and DNN specific tasks.

In this paper, we describe two approaches to schedule DNN inference tasks on a cluster of GPU servers. The servers may be heterogeneous in GPU types, which leads to a more complex machine dependent scheduling problem. We start with a heuristic scheduler: it selects the minimal batch size that satisfies the throughput requirement to keep E2E delays low. Although the scheduler is simple and efficient, its performance is sensitive to careful parameter tuning, and it tends to frequently use faster DNN models that provide lower accuracy. To overcome the drawbacks, we further propose an advanced approach based on deep reinforcement learning (RL) [6]. Our RL scheduler uses real-time system status as the input to a policy DNN model that selects scheduling actions, which is trained to maximize QoS of query responses. To reduce scheduling overhead, it performs actions on demand instead of running at a fixed time step. We conduct simulation-based experiments to evaluate the schedulers, and show that the RL approach achieves better performance.

2 DNN INFERENCE SERVER

2.1 System Overview

The system model is shown in Figure 1 where a server processes a continuous stream of incoming DNN inference queries sent by clients. A query comes with soft and hard deadlines (t^s_{ddl} and t^h_{ddl})

[1]For example, Google Cloud Vision API: https://cloud.google.com/vision/, and Microsoft Cognitive Services: https://www.microsoft.com/cognitive-services/.

CIKM'17 , November 6–10, 2017, Singapore, Singapore
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3133045

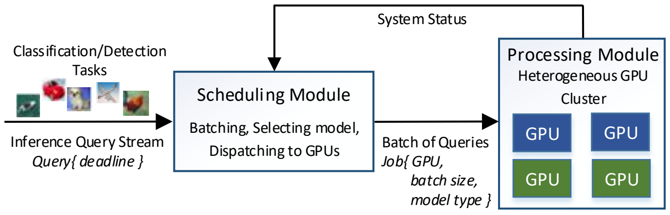

System Status
Classification/Detection Tasks
Inference Query Stream
Query{ deadline }
Scheduling Module
Batching, Selecting model, Dispatching to GPUs
Batch of Queries
Job{ GPU, batch size, model type }
Processing Module
Heterogeneous GPU Cluster
GPU
GPU
GPU
GPU

Figure 1: The architecture of DNN inference server.

specified by the application, which give the time budget to process the query (described in Section 2.3). We assume that the client clock is synchronized to the server, as clock synchronization is generally available nowadays, even on mobile devices. The *processing module* in Figure 1 is a GPU cluster that may contain different types of GPUs. Each GPU processes batches of queries sequentially and returns the inference results. The *scheduling module*, which is the core of this work, packs queries as a batch and selects the suitable DNN model to run on GPUs.

2.2 The Impact of Batch Size

First we illustrate the impact of batching on processing throughput of DNNs with measurement data. We denote the forward propagation delay of a DNN as $D_{B,M,G}$, where B is the batch size, M is the DNN model, G is the type of GPU. There are generally two ways to speed up a model: training a simpler model with fewer parameters, or simplifying the original model using approximation techniques such like matrix factorization, pruning, and quantization [5]. This work considers the former approach and uses YOLO [10], a model for real-time object detection, as the testcase. Two models are used: the *full* model achieves a mean average precision (mAP) of 63.4% on the PASCAL VOC dataset [4] and the mAP is 57.1% for the less complex *tiny* model.[2] We measure $D_{B,M,G}$ on AWS `g2.2xlarge` (g2) and more powerful `p2.xlarge` (p2) GPU instances.[3] The measured throughputs in queries per second (QPS) are given in Table 1, with B ranging from 1 to 64, $M \in$ {tiny, full}, and $G \in$ {g2, p2}. It shows that batching effectively improves the throughputs.[4]

Table 1: DNN Inference Throughputs (QPS)

G	M	B=1	2	4	8	16	32	64
g2	tiny	25.6	35.7	51.9	66.1	73.7	84.7	91.2
	full	9.4	9.4	12.3	14.7	16.0	16.8	-
p2	tiny	23.8	35.7	63.5	90.9	109.6	131.1	141.9
	full	14.5	18.9	24.2	28.4	31.5	33.9	34.6

2.3 Performance Metrics of Query Response

The quality of a query response (Q), defined as a function of the end time of query processing (t_{end}), the deadlines ($t^{s,h}_{ddl}$) and the DNN model (M), is computed as:

$$Q(t_{end}, t^{s,h}_{ddl}, M) = \kappa_M \cdot V(t_{end}, t^{s,h}_{ddl}), \qquad (1)$$

where $\kappa_M \in (0, 1]$ is a factor to penalize less accurate models, and V is a time-utility function (TUF) to represent the influence of delay on response quality. The scheduler is designed to maximize the real-time response quality Q, therefore it is essential to select κ_M and V that accurately model the application performance. Choosing a proper TUF for time-sensitive applications has been extensively studied by the real-time system community [8].

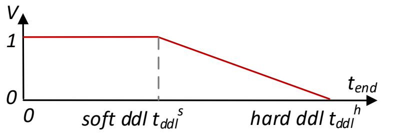


Figure 2: Utility of query response as a function of time delay $V(t_{end}, t^{s,h}_{ddl})$.

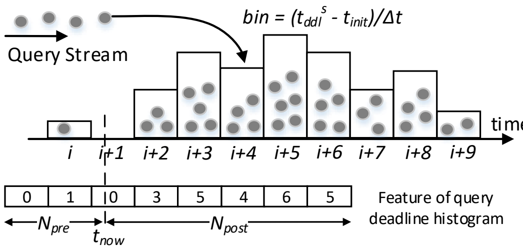


Figure 3: Buffering incoming queries in bins to approximately order queries according to deadline t^s_{ddl}.

The TUF (V) used in this work is illustrated in Figure 2: the utility value of a response is 1 if t_{end} meets the soft deadline, otherwise it decreases linearly until zero at the hard deadline. This work does not consider downstream latency that is unknown at the scheduling time. Timecard [9] details a solution to predict downstream latency based on response content and network condition.

3 SCHEDULER DESIGN

We consider two QoS-aware schedulers: a simple heuristic scheduler and a more advanced scheduler using deep reinforcement learning. Both approaches prioritize the queries with closer deadlines to avoid missing deadlines. For a heavily loaded server with a high query arrival rate, it is inefficient to order every query based on deadline in the query buffer. As an alternative, we approximate query ordering by quantizing time into discrete bins, as shown in Figure 3. The duration of each bin is Δt. A query with soft deadline t^s_{ddl} is allocated to bin i with $i = (t^s_{ddl} - t_{init})/\Delta t$, where t_{init} is the system initialization time. The bins are created and deleted dynamically. The scheduler fetches queries starting from the bins with the smallest indexes.

3.1 Simple Scheduling by Heuristic

The heuristic scheduler always selects the smallest batch size (B) that satisfies the requirement on throughput:

$$B \geq [\alpha \cdot (D_{B,M,G} \cdot R_t) + (1 - \alpha) \cdot N_t] \cdot \beta \cdot \xi_G, \qquad (2)$$

where R_t is the query arrival rate at time t, and $\xi_G \in (0, 1]$ is the portion of queries allocated to GPU G. The term $D_{B,M,G} \cdot R_t$ represents the number of incoming queries. N_t is the total number of buffered queries. $\alpha \in [0, 1]$ and $\beta > 0$ are tuning parameters. The scheduler allocates queries to GPUs proportionally to processing capability. In this work we use the throughput with $B = 8$ as the processing capability of a GPU, and obtain ξ_G accordingly. The scheduler uses the most accurate DNN model in default. If the batch size is not less than a threshold $B_{thrd} = 8$, in order to improve QoS by reducing delay, the scheduler checks all simplified model

[2]YOLO models: https://pjreddie.com/darknet/yolo/.
[3]GPU type is NVIDIA GRID for `g2.2xlarge` and NVIDIA GK210 for `p2.xlarge`.
[4]Schedulers that handle more classes of DNNs, model accuracy-speed trade-offs, and GPU types lie in our future works.

types and selects the model maximizing the total quality value of this batch $Q_B = \kappa_M \cdot \sum_{q \in B} Q(t_{now} + D_{B,M,G}, t^{s,h}_{ddl}, M)$.

3.2 Advanced Scheduling by Deep Reinforcement Learning

In reinforcement learning, at time t, the agent observes the state s_t, chooses the action a_t, and obtains a reward r_t with this action. The learning process aims at maximizing the expected cumulative reward: $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in (0, 1]$ is the discount factor of future reward. The agent selects actions based on a policy $\pi(s, a)$ that gives the probability to perform a in state s. A deep RL agent uses a DNN to approximate the policy $\pi(s, a)$ (DNN details are in Section 4.2). To customize this deep RL agent in our scheduling problem, here we describe state, action, and reward. We follow the routine of a policy gradient RL algorithm to train the model [12].

State represents the run-time status of the system, including the following four parts:

Histogram of query deadlines: We use a histogram of query deadlines as the feature representing urgency of buffered queries. The feature vector is obtained from the sizes of query bins. As shown in Figure 3, at the current time t that corresponds to the bin i, the histogram feature comprises of N_{pre} bins before and N_{post} bins after the bin i. The size is zero for a non-existent bin, hence this feature has a fixed length $N_{pre} + N_{post}$. Because the histogram state has been truncated to a fixed length, it may be less than the real total number of buffered queries. Thus, we add the total number of buffered queries as an additional feature.

Arriving rate: The state contains the query arrival rate (R_t) to estimate the number of incoming queries in future.

GPU type: The scheduler must consider the processing capability of the GPU that processes the batch. We use a binary vector to represent the GPU type: an element e of the vector is 1 if the GPU matches the type that e is associated with, otherwise e is 0.

Availability of GPUs: When dispatching a batch to a GPU, the optimal batch size is influenced by the availability of other GPUs. We represent availability as the expected rest computing time of each device ($t_{end} - t_{now}$), where t_{end} is estimated using the measured mean computing time (see Table 1). The state includes the availabilities of all GPUs.

An **action** selects a batch size B and a model type M, then runs the selected model with the batch of queries on a GPU. For a heterogeneous cluster, it is possible that an action is feasible for some types of GPUs but not for the others, for example, out of memory may happen when it runs a complex model with a large batch on a GPU with low memory capability. The scheduler automatically selects a smaller batch size when the action is not feasible.

The **reward** is defined as the average quality (Equation 1) of the latest W query responses. It represents the system's real-time QoS. Consider that several GPUs may process queries in parallel, for the reward of the action a_t, the window W excludes any batches generated after t, to eliminate the noise on response quality caused by other GPUs.

Reinforcement learning typically computes actions at regular time intervals (*e.g.*, [6]). The interval should be short enough for the scheduler to perform timely actions. For example, in this scheduling problem, the inference delay of the smallest batch $B = 1$ is around 60ms, the scheduling interval should be even smaller to dispatch queries to newly available GPUs quickly. Using a small

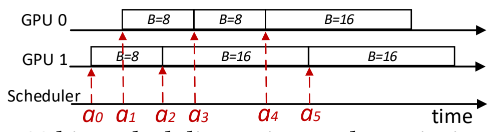


Figure 4: Taking scheduling actions at dynamic time points.

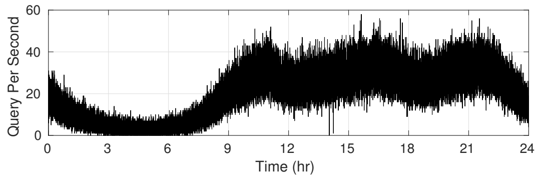


Figure 5: QPS of web queries in SogouQ dataset.

fixed scheduling interval exerts high computation overhead. To resolve this problem, as shown in Figure 4, we compute the next action to take using the policy DNN model only when there are free GPUs. It leads to a large reduction of the scheduling overhead when incoming workloads keep all GPUs busy. In the evaluation using a single GPU (described in Section 4.2), taking actions at dynamic time points needs only 3% action computations compared with using a fixed interval of 10ms.

4 EVALUATION

4.1 Experiment Setting

We emulate the DNN inference server (Figure 1) on a simulation platform. The simulator generates newly arriving queries, schedules new batches of queries to run, and checks whether each GPU has finished the previous batch. A query is a YOLO object detection task that uses either the full or the tiny model. We use $\kappa_{full} = 1$ and $\kappa_{tiny} = 0.5$ for the models. The processing delay of a batch ($D_{B,M,G}$) is generated by a normal distribution fitting the measurement data (Table 1). The query stream is generated from a web query trace provided by the SogouQ query log dataset[5]. The QPS of query stream over a day is plotted in Figure 5. We generate new queries every 10ms. Because the timestamp resolution of the original query log is 1s, queries are allocated to finer 10ms intervals using a normal distribution. For each query, there is an upstream network delay before it arriving at the server. We consider a mobile computing scenario where clients send queries to the server via LTE networks. The delays are generated from a *Pareto* distribution which is fitted to LTE delay data produced by the Mahimahi tool [7]. Deadlines $t^s_{ddl} = 1$s and $t^h_{ddl} = 2$s are applied to all queries.

A baseline scheduler (denoted as *fixed*) is tested for performance comparison. It always batches all buffered queries to run on a GPU, with 32 as the maximal batch size. We use the query data in the time range from 9hr to 12hr (Figure 5) as the training data. The continuous query log is split into training jobs with a duration of 10min to train the RL scheduler. The training data is also used to optimize the parameters of the heuristic scheduler. Because the scheduling problem is trivial at low load level (time 0hr to 9hr), we are mostly interested in the scheduling performance with high load. Hence, the evaluation is done using 4 testing jobs generated from the time 12hr to 24hr, each lasts for 3hr (*e.g.*, see Table 2).

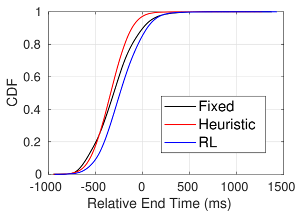
[5]SogouQ: http://www.sogou.com/labs/resource/q.php.

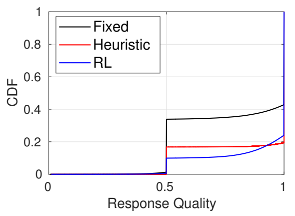
Table 2: Mean (SD) Response Quality of Single GPU

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	0.883 (0.204)	0.818 (0.235)	0.860 (0.218)	0.863 (0.218)
Heuristic	0.968 (0.117)	0.915 (0.185)	0.950 (0.146)	0.939 (0.161)
RL	0.978 (0.084)	0.931 (0.149)	0.961 (0.114)	0.949 (0.132)

Table 3: P_{tiny} (%) of Single GPU

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	20.2	33.8	25.0	25.4
Heuristic	5.5	16.3	9.2	11.7
RL	2.0	8.2	4.1	6.1





(a) CDF of relative end time **(b) CDF of response quality**

Figure 6: Performance comparison of Fixed, Heuristic and RL schedulers.

4.2 Performance of Schedulers

Single GPU: We first evaluate the schedulers using a processing module with one GPU (p2). Both schedulers use batch sizes as given in Table 1, and the RL scheduler has $B = 0$ as an additional option. The bin size of query buffer is $\Delta t = 200$ms. The arrival rate of queries (R_t) is measured in a sliding window of 5s. The heuristic schedule use the parameters $\alpha = 0.5$ and $\kappa = 0.5$ that give the highest mean response quality on the training data. The policy model of the RL scheduler has 1 fully connected hidden layer with 64 neurons. The histogram of deadlines feature in the state uses $N_{pre} = 4$ and $N_{post} = 6$ (Section 3.2). The reward is measured using the qualities of last $W = 1000$ queries.

The scheduling performance is measured with two metrics: (1) mean response quality (Equation 1) over 3 hours, given in Table 2 with standard deviation (SD); (2) P_{tiny}, the percentage of queries that use the tiny model and have lower inference accuracy, given in Table 3. The fixed scheduler that does not adapt batch size has the lowest response quality. The heuristic scheduler effectively improves the mean quality, and the RL scheduler achieves better performance. The Cumulative Distribution Functions (CDFs) of end time relative to soft deadline (t_{end} - t^s_{ddl}) of all queries are plotted in Figure 6a. It illustrates that compared with the heuristic approach, the RL approach has larger response delays, whereas it achieves higher quality by using the tiny model for less times (as in Figure 6b). The scheduling actions performed by the heuristic and RL schedulers in a duration of 10min are visualized in Figure 7.

Heterogeneous GPU Cluster: To evaluate the schedulers for more servers, we use a cluster of 5 g2 GPUs and 5 p2 GPUs. The query arrival rate is scaled up by 8 times in the simulation to fill up the processing capability of the cluster. The policy DNN model of the RL agent has the same hidden layer as the model for one GPU. The results are given in Table 4 and 5. We find that similar to the single GPU case, the RL scheduler delivers higher response quality by using the tiny YOLO model less frequently.

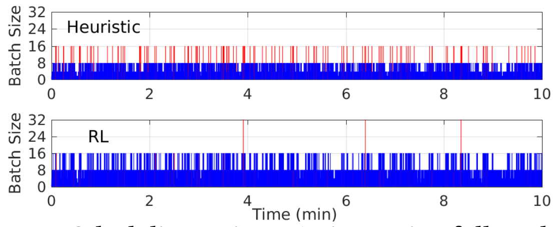


Figure 7: Scheduling actions. Actions using full model are plotted in blue and tiny model in red.

Table 4: Mean (SD) Response Quality of Multi-GPUs

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	0.820 (0.231)	0.767 (0.242)	0.801 (0.237)	0.813 (0.234)
Heuristic	0.948 (0.148)	0.883 (0.210)	0.925 (0.176)	0.916 (0.186)
RL	0.955 (0.139)	0.895 (0.202)	0.933 (0.168)	0.922 (0.179)

Table 5: P_{tiny} (%) of Multi-GPUs

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	32.6	43.5	36.5	34.4
Heuristic	9.6	22.8	14.4	16.4
RL	8.2	20.6	12.8	14.8

5 CONCLUSIONS

This paper examines the QoS-aware scheduling problem for DNN inference workloads in cloud computing. We describe a simple heuristic scheduler and a more advanced scheduler based on deep reinforcement learning. In addition, we propose to perform actions on demand at dynamic points in time, which greatly reduces the scheduling overhead. Through simulation experiments, we demonstrate that the RL scheduler achieves higher mean response quality and uses the simplified DNN model less frequently.

REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[2] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
[3] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *ACM SoCC*, 2014.
[4] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal visual object classes (VOC) challenge. *International journal of computer vision*, 88(2):303–338, 2010.
[5] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.
[6] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, 2016.
[7] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement. In *SIGCOMM*, 2014.
[8] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *ISORC*, 2005.
[9] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *SOSP*, 2013.
[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, June 2016.
[11] M. Song, Y. Hu, H. Chen, and T. Li. Towards pervasive and user satisfactory CNN across GPU microarchitectures. In *HPCA*, 2017.
[12] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999.
[13] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.