Low Latency Model Serving with Late Batching

Pin Gao
Computer Science
Tsinghua University
gao-p12@mails.tsinghua.edu.cn

Lingfan Yu
Computer Science
New York University
lingfan.yu@nyu.edu

Yongwei Wu
Computer Science
Tsinghua University
wuyw@tsinghua.edu.cn

Jinyang Li
Computer Science
New York University
jinyang@cs.nyu.edu

Abstract

Serving pre-trained deep neural network (DNN) models must meet the requirement of low-latency, which is often at odds with achieving high throughput. Existing deep learning systems use batching to improve throughput, but do not perform well when serving models with dynamic dataflow graphs. We propose Late Batching, which improves both the latency and throughput of DNN serving. Late Batching breaks a dataflow graph into a collection of subgraphs (called "blocks") and dynamically assembles a batched block for execution as requests join and leave the system. We implemented our approach on MXNet. Preliminary experiments show that Late Batching brings significant benefits.

CCS Concepts • **Computer systems organization** → *Data flow architectures*;

Keywords deep learning, serving, latency

ACM Reference Format:
Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2017. Low Latency Model Serving with Late Batching. In *Proceedings of Workshop on AI Systems at Symposium on Operating Systems Principles (AISys'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/

1 Introduction

Existing work on deep learning systems (e.g. Tensorflow[2], MXNet[3], Caffe[5], PyTorch[1], Tensorflow Fold[7], Dynet[9]) have so far focused on improving throughput during the training of a DNN model. Unlike training, serving a pre-trained DNN model also places much emphasis on low latency in addition to good throughput[4].

Batching is a key technique in achieving good throughput. However, for an important and increasingly prevalent class of DNN applications, namely, RNN-based sequence models, naive batching harms both the latency and throughput. A RNN-based model consists of a chain of computation blocks whose length is determined by the input request. Thus, it is a challenge how to batch requests of different lengths. Existing systems perform dataflow graph batching. The most common form of dataflow batching (e.g. Tensorflow, MXNet) is to batch incoming requests and pad requests in a batch to the same length. Doing so allows the system to construct a static dataflow graph for executing the batch of requests and return results to users after the entire batch finishes. Two recent systems, DyNet [9] and Tensorflow Fold [8], avoid padding during batching. They adopt dynamic batching techniques which generate a dataflow graph for each request and dynamically merges individual graphs into a single dataflow graph for the entire batch of requests. While these dataflow batching strategies have worked well for DNN training, they are not good enough for serving which cares more about latency. There are two reasons: a) arriving requests need to wait for last batch to finish, even if the last batch is not full. b) all requests in the same batch won't be returned to the users until the entire batch finishes even if some short requests have finished much sooner.

To address the above problems, we propose *Late Batching*, a dynamic batching mechanism that can batch incoming requests with ongoing ones and can return requests to users as soon as they finish. Late Batching does not form a complete dataflow graph for the batch. On the contrary, it breaks the execution into subgraphs and batch these subgraphs lazily. Our approach brings two benefits. First, new requests can *join* the execution of current batch if they can be batched together which enables requests to avoid waiting for last batch to finish. Second, if the execution of some request in the batch has finished, it can *leave* the execution of current batch instead of waiting for the completion of other long requests in the batch.

Taking LSTM as an example, Figure 1 shows the difference between dataflow batching and Late Batching. In this example, each row represents the lifetime of one request. As shown in the figure, we have a batch size of 4. For dataflow batching, when a request comes, it needs to wait for the completion of last batch. So, the batch of request 10 to 13 starts at time 105. On the other hand, they finish together since they share the same dataflow graph. So none of them can return to user until time 112 even though the shortest one only has sequence length of 2. For Late Batching, at time 102, one request in previous batch left. And request 10 can

AISys'17, October 28, 2017, Shanghai, China
2017.

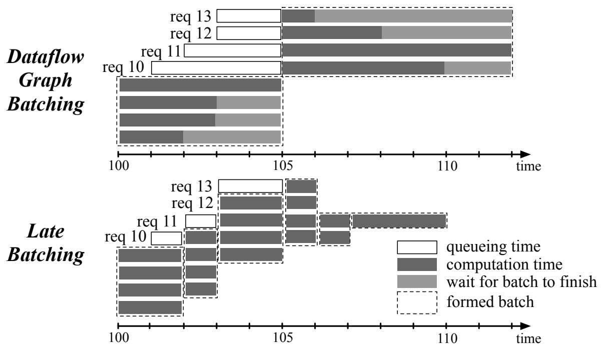

Figure 1. LSTM: Dataflow Batching v.s. Late Batching

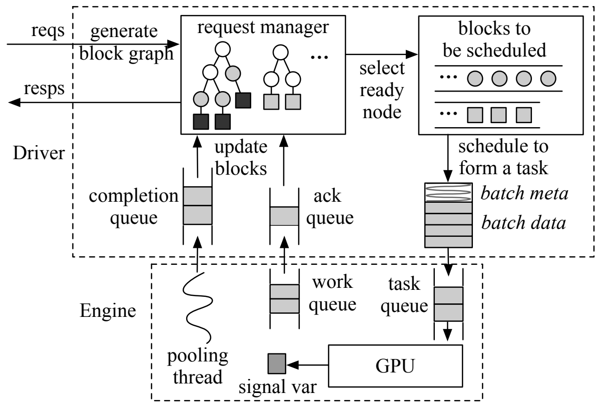

Figure 2. System Architecture

join the execution immediately and finishes and returns at time 105.

2 System Design

Our system batches the DNN execution at the granularity of *block*. A block is a subgraph in the dataflow graph which can be either a single operator or many operators that are grouped together. For example, an RNN cell can be considered as a block. Grouping operators into blocks allows us to make the dataflow graph coarse-grained: computation is represented as a *block graph*, where each node represents a block and each edge depicts the direction in which data flows from one block to another. If two blocks computes the same function and uses the same parameters, they are considered the same block, and thus can be batched together if there is no data dependency between them.

The overall system design is depicted in Figure 2. Our system has two components: Driver and Engine. The Driver is responsible for generating the block graph for each request, tracking the dependency of nodes in a block graph, selecting nodes to form a batched computation task, and submitting the task to the Engine. The Engine is a middle layer lying between the Driver and GPUs. It is responsible for executing a batched computation task and notifying the Driver about task completion.

Our design batches at the block level and utilizes runtime information (specifically the arrival and completion of requests) to dynamically make batching decisions. The key difference between us and existing systems (such as Tensorflow, Tensorflow Fold) is that, instead of submitting an entire dataflow graph to the execution engine and wait for its completion, we are able to join new requests into execution and let finished requests leave immediately.

However, everything comes at a price. Our design does introduce two new challenges. First, we must be able to effectively overlap the computation of the Driver with GPU execution. Second, upon finishing a task, the Engine needs to notify the Driver via some notification mechanism since the communication is now asynchronous. To address the above challenges, our system adopts two novel techniques.

Ahead-of-Time Scheduling. Adjacent nodes in the same block graph, with one depending on the other, have to be scheduled with original order preserved. However, it is also too late to schedule the second block after the completion of the first one because that makes the Engine wait, wasting GPU resources. How to overlap scheduling and execution? We note that the Driver has knowledge about blocks that have been pushed to Engine but does not know which block will finish first. To solve this problem, we exploit the fact that each block's computation is substantial and the GPU can execute only one block a time. Thus the order of blocks pushed to the GPU is the order in which blocks are executed. So upon pushing a block to GPU, Engine notifies Driver immediately and Driver can schedule the next block.

Signal Operator. In order to let the Engine inform the Driver about task completion asynchronously, we wrap notification into a *signal operator* and appends it to the end of each block. The signal operator changes a signal variable, an integer for example. To avoid huge synchronization cost between CPU and GPU, the signal variable is allocated in pinned host memory which can be accessed by zero copy without blocking GPU execution.

3 Preliminary Results

We implemented the Late Batching based on MXNet. Currently, our system is able to run LSTM cells. Our latency improvement mainly comes from two aspects: a) early departure and b) early start. Our LSTM evaluations use the WMT15 Europarl [6] dataset and an artificial dataset in which all sequences have the same fixed length (20). Each workload contains 100k requests, of which the arrival time obeys Poisson distribution. We measured the latency and throughput of these systems under varying request arrival rates. We compare our system against MXNet (using a bucket strategy to batch requests with similar length) and Tensorflow Fold. Our experiments are performed on a Nvidia GeForce GTX Titan X with 12GB memory. Results are compiled into figures in Appendix. Tensorflow Fold has strictly worse performance, because it spends a large proportion of the computation time merging dataflow graphs. Also, as the computation goes on, its effective batch size becomes smaller, making it unable to saturate the GPU.

How early start affects latency Our system gains huge latency improvements on WMT15 dataset as shown in Figure 3 and 4. Considering the sequence length may vary from 1 to 330, bucketing way will introduce a lot of buckets. When there's limited exectution devices, buckets will wait until there's one device available. Thus it introduced a lot of time on waiting. Even in the case of fixed-sequence length dataset, there's only one bucket, our system still have benefits on reducing waiting time. Figure 6 and 7 shows the results on fixex-squence length dataset. Still, these two graphs show that when our system has not reached its max throughput, latency is roughly 25-40% less than MXNet. This is because when our system is not fully loaded, requests can always join the current batch and start execution immediately. Thus, early start saves waiting time and improves latency.

How early leave affects throughput From Figure 3 and 4 we can see that our system get huge throughtput benefits. Thanks to early leaving, requests do not need to pad and they can return immediately upon finishing executions. So the request does not need to wait for the whole batch to finish. Also, by looking at Figure 5, we can see that early leave and early start together avoids useless computation caused by padding when system is fully loaded, and improve the system's max throughput by about 25-33%.

Acknowledgments

This paper was greatly improved by the advices from our extraordinary fellow colleagues Minjie Wang, Chien-Chin Huang, Cheng Tan, and the anonymous reviewers of AISys 2017. We thank Sebastian Angel for his careful readings and suggestions that helped improve our paper quality a lot. Special thanks to Zheng Zhang from NYU Shanghai for providing us with GPU servers.

References

[1] 2017. PyTorch. http://pytorch.org/. (2017). [Online; accessed 1-September-2017].

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th International Symposium on Computer Architecture.*

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia.* ACM, 675–678.

[6] Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, Vol. 5. 79–86.

[7] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[8] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. In *ICLR.*

[9] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980* (2017).

A Appendix

Results on WMT15 Europarl dataset and Fixed-length (20) dataset

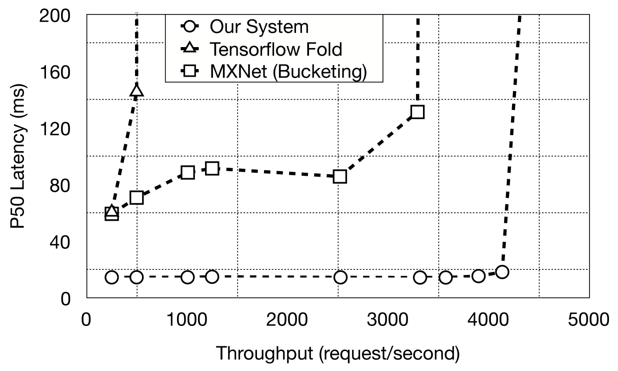

Figure 3. P50 latency vs throughput on WMT15 dataset

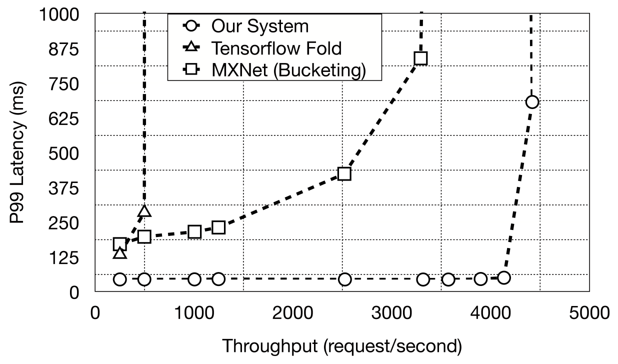

Figure 4. P99 latency vs throughput on WMT15 dataset

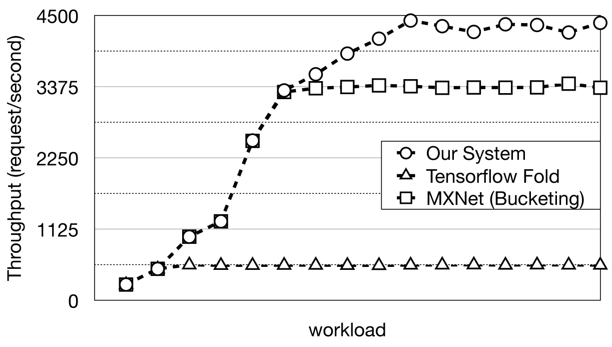

Figure 5. Throughput vs workload on WMT15 dataset

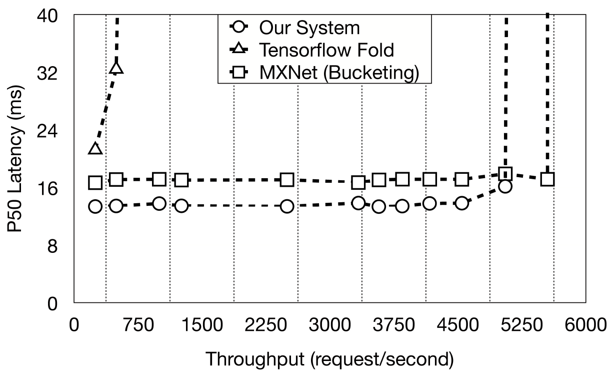

Figure 6. P50 latency vs throughput on fixed-length dataset

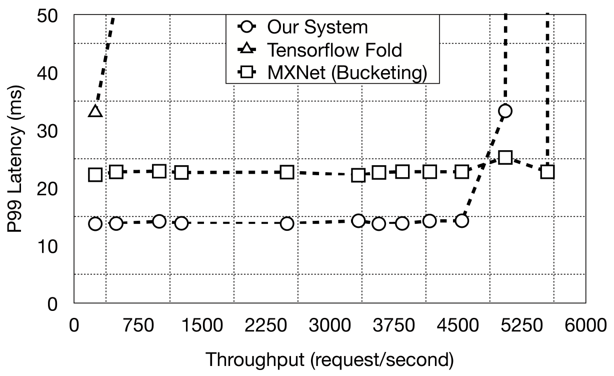

Figure 7. P99 latency vs throughput on fixed-length dataset