Vol.2018-HPC-165 No.13
2018/7/31

An Efficient Operation Auto-batching Strategy for Neural Networks Having Dynamic Computation Graphs

Yuchen Qiao[1,a)] Kenjiro Taura[1,b)]

Abstract: It has become crucial to improve the speed of training neural networks for the research and development of deep learning models and applications. Organizing the same operations, which can be executed in parallel, in the computation graph of a neural network into batches helps making full use of the available hardware resources. This batching task is usually done by the developers manually. The operations in the neural networks having dynamic computation graphs, however, are difficult to be efficiently grouped by manual because of the data with varying dimensions and structures or the dynamic flow control. Several automatically batching strategy have been proposed, but they don't efficiently group the operations in the backward propagation of training neural networks. This paper tries to apply efficient operation auto-batching in both the forward and backward propagations of neural networks having dynamic computation graphs. We also report the evaluation results of our strategy.

1. Introduction

Recent years, neural networks (NN) have been applied on a lot of machine learning topics and shown their great effects. Natural Language Processing (NLP), which is analyzing, understanding, and deriving meaning from human languages by using computers, has also benefit from applying new neural networks based models on all kinds of its sub topics. In the recent years, neural network based solutions have made impressive advancements in all kinds of NLP tasks like Sentiment Classification [15], [17], Named-Entity Recognition (NER) [10], Machine Translation (MT) [6], [16], [19], Question Answering (QA) [9] and so on. Deep learning with neural networks enables automatic feature extraction and representation learning, which liberates NLP tasks from time-consuming and often incomplete handcrafted features. What's more, the neural networks based NLP gains from incremental dataset, which can be obtained easily in the Big Data Era. As the neural networks used for NLP as well as the training sample dataset become larger and larger, training time for one single network are rising into hours even days [3], [5], [6], [7]. Here we take *Machine Translation*, whose model is very complex and training process is time-consuming, as an example topic to discuss on the time-consuming training procedure of NLP applications. Table. 2 summarized not only the accuracy performance but also the computing performance of the training process of the state-of-the-art works on WMT 2014 English-to-French translation tasks. From this summary we can see the training process of the machine translation is very time-consuming. It takes at least days to finish even equipped with multiple GPUs. Therefore, it's becoming more and more important to take the computation performance of neural networks into consideration.

Several frameworks such as TensorFlow [1], Chainer [18] and DyNet[12] have been developed to help user to build up neural networks easily by reducing engineering work and provide efficient execution of the computation graph of neural networks. Since models for NLP applications are usually trained from sentences with different lengths, the structures of computation graphs for different instances varies a lot. Therefore, frameworks supporting dynamic computation graph definition and execution are more welcome. Usually, the parallel computing is utilized to help with achieving higher computing performance. However, for NLP tasks, which mostly utilize Recurrent Neural Networks (RNNs) or Recursive Neural Networks to extract the sequential and syntactic information from the input words or sentences, are hard to be parallelized because of the dependency between different parts inside of the model as well as the variable lengths and syntactic tree structures of the inputs. Batching, which means organizing the same operations, which can be executed in parallel, in the computation graph of a neural network into batches helps enabling parallelism and making full use of the available hardware resources. This batching task is usually done by the developers manually. However, it's difficult for programmers to group the operations efficiently by manual because of the data with varying dimensions and structures or the dynamic flow control.

Researchers are trying to implement automatic batching in the frameworks for deep learning. Several automatically batching strategy have been proposed, but they don't efficiently group the operations in the backward propagation of training neural networks. In this paper, we discuss on two different automatic batching strategy and their shortcomings and try to apply efficient operation auto-batching in both the forward and backward propaga-

[1] Graduate School of Information Science and Technology, University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo 113-0033, Japan
[a)] qiao@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

Table 1 Accuracy and Performance of State-of-the-art works for NMT on WMT2014 English to French Dataset.

Paper	Model	BLEU	Training Time
(Cho et al., 2014) [4]	Phrase table with neural features	34.50	3 Days
(Sutskever et al., 2014) [16]	Reranking phrase-based SMT best list + LSTM seq2seq	36.5	10 Days-8 GPUs
(Wu et al., 2016) [19]	Residule LSTM seq2seq + RL refining	41.16	6 Days-96GPUs
(Gehring et al., 2017) [7]	seq2seq with CNN	41.29	37 Days-8 GPUs
(Vaswani et al., 2017) [2]	Attention mechanism	41.0	3.5 Days-8 GPUs

tions of neural networks having dynamic computation graphs.

The rest of this paper is organized as follows: Section 2 introduce the basic principle of batching and two automatic batching strategies that have been proposed. We also analyze the shortcomings of them. Section 3 present the automatic batching method proposed by us. An experimental evaluation is shown in Section 4 and we give our analysis. Section 5 introduces our future goal and plan.

2. Batching

Batching is the most common way to enable parallelism in deep learning. Minibatching takes multiple training example and groups them together to be processed simultaneously, often allowing large gains in computation efficiency due to the fact that modern hardware (CPUs and GPUs) have very efficient vector processing instructions that can be exploited with appropriately structured inputs. As shown in Fig. 1, common examples of this in neural networks include grouping together matrix-vector multiplies from multiple examples into a single matrix-matrix multiply, or performing an element-wise operation (such as tanh) over multiple vectors at the same time as opposed to processing single vectors individually.

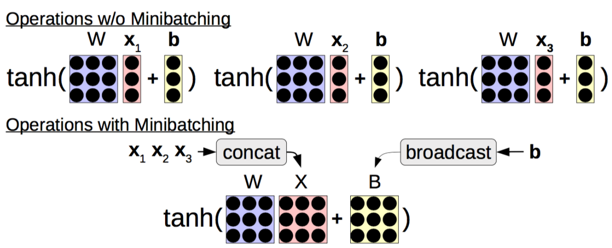

Fig. 1 An example of minibatching for an affine transform followed by a tanh nonlinearity [12].

2.1 Unefficient Batching

It is necessary to batch up all the operations to make the sequences be processed in parallel so as to make good use of efficient data-parallel algorithms and hardware. However, for recurrent and recursive neural networks, it's really hard to apply an ideal batching because the lengths and syntax structure of different inputs varies a lot.

Left part of Fig. 2 shows a computation graph for computing the loss on a minibatch of three training sentences with recurrent neural networks. All these three sentences have different lengths of 2, 3, 4. The operations at Step 1 and 2 can be batched together for parallel execution. However, only operations of sentence 1 and 3 on Step 3 can be batched together and executed in parallel. Given the training samples in this figure, an idea set of batching should be (Op_1^1,Op_1^2,Op_1^3), (Op_2^1,Op_2^2,Op_2^3), (Op_3^1,Op_3^3), (L^1,L^2,L^3). If it can be batched like that, the minibatch of sentences can be processed with the most parallelism.

However, it's really hard to implement an ideal batching for every minibatch since the lengths of inputs vary in each minibatch. We can't count on programmers to implement an ideal batching manually. Usually, users would like to pad the inputs in the same minibatch to let them have the same length, like what the right part of Fig. 2 tells us. Though the inputs with padding have the same length and are easy to be batched, the computation on the padding parts are totally wasted. The wasted computation is considerable and leads to un-efficiency to the model's training.

2.2 Automatic Batching

Since we want to achieve ideal batching without manual implementation, researchers started to explore the possibility to implement automatic batching algorithms. Generally, an automatic batching algorithm consists of several steps [12]:

- *Graph definition*: In this steps, applications define the graph that represents the computation. Nodes in this graph represent different operations (tanh, log, ...)
- *Operation Batching*: First, the algorithm partition the nodes into groups, where nodes in the same group should have the potential for batching. This is done through associating nodes with a signature. Nodes with the same signature can be batched together and are able to be executed simultaneously when their inputs are ready. This signature usually depends on the operation the node represent and also contains the information about its input/output dimension to provide more information for batching. Second, the algorithm schedules an execution order in which nodes that have the same signature and do not depend on each other are scheduled for execution on the same step
- *Forward-backward graph execution and update*: The framework performs the calculation according to the execution order and batching decisions generated in the second step.

Actually, the first and the third step are shared with standard execution of computation graphs. There are two different heuristic strategies for identifying execution orders for the second step:

Depth-based Batching [11] is implemented by assigning each node the depth of it in the original computation graph. The depth of a node is defined as the maximum length from a leaf node to itself. Nodes that have an identical depth and signature (operation) are batched together. With this design, nodes have the same depth don't depend on each other and all the nodes will have a higher depth than its inputs. As a result, batching can be done. However, this heuristic strategy has a shortcoming and will miss

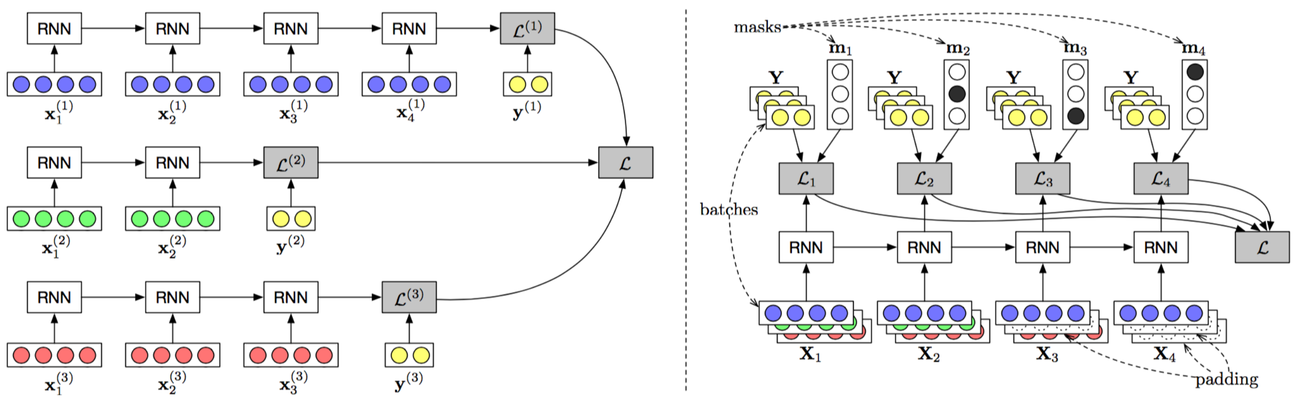

Fig. 2 Two computation graphs for computing the loss on a minibatch of three training instances consisting of a sequence of input vectors paired with a fixed sized output vector. [12].

good opportunities like batching loss function calculation in Fig. 2 because they don't have the same depth.

Agenda-based batching [13] is a way that dose not depend on depth. This method implements and maintains an agenda that record all the available modes which don't have unsolved dependencies. Each node maintains an agenda tracking available nodes (have no unresolved dependencies) in the computation graph. During the initialization, nodes have no coming inputs are put into the agenda. Then at each iteration, the algorithm select nodes with the same signatures from the agenda and group them into a single batch. After the execution of the batched nodes, the algorithm remove these nodes from agenda and decrease the dependency counter of all of their successors. This process is repeated until all the nodes have been processed. During the execution, there may be two groups of batched nodes existing in the agenda at the same iteration. In order to prioritize nodes in the agenda, there is a heuristic method based on the average depth of all nodes with their signatures, such that nodes with a lower average depth will be executed earlier. With this heuristic method, such that nodes with a lower average depth will be executed earlier.

2.3 Shortcomings

According to [13], the agenda-based automatic batching strategy performs better than the depth-based one and shows a very good computing performance. This strategy is implemented based on DyNet and can be easily used by the programmers. However, this automatic batching strategy stills has some shortcomings.

As we know, the training procedure of a neural network actually contains two parts: the forward propagation and the backward propagation. In the agenda-based automatic batching strategy, the framework only do the analysis about the node's batching generation and execution order during the forward propagation. The strategy treats the backward propagation as a simple reverse one of the forward propagation. In the batched execution of the backward propagation, it just transverses the nodes of operations in the reverse order of the batched execution. When it arrives at a group of batched nodes, it calculates all the arguments' derivative of those nodes. However, this will lead to missing some chance to batch operations in the backward propagation.

Fig. 3 illustrates parts of the computation graph of a vanilla RNN language model. We can see that the node Wh_0 and Wh_1 can not be batched together during the forward propagation since there is dependency between each other. In the backward propagation, however, the derivative from Wh_0 to W and that from Wh_1 to W can be batched and calculated together. In the agenda based strategy, this batching strategy will be missed because it just uses a reverse execution order and batch generation of the forward propagation.

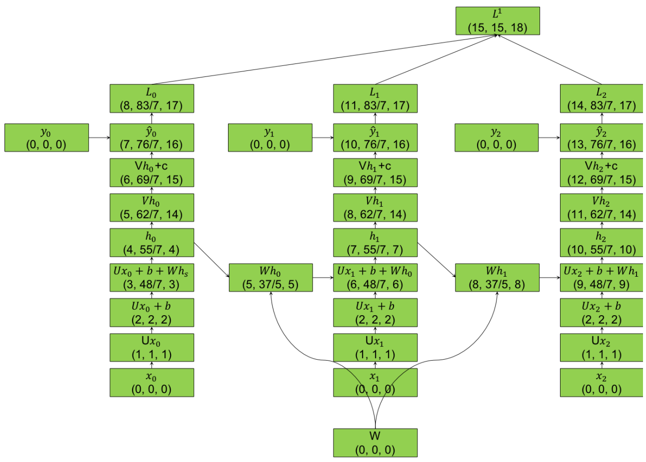

Fig. 3 An example of computation graph.

3. Proposed Approach

Our goal is to make more use of the batching chance in the backward propagation and we developed our proposed approach based the agenda-based strategy. According to our analysis, the missed batching chances in the backward propagation by the agenda-based strategy are mostly the calculation of the parameters weight matrices. Therefore, we implemented some modification in the backward propagation when using the agenda-based strategy.

In the backward propagation, we don't calculate the derivatives to the parameters until the last moment. After other derivatives to other nodes have been all calculated, do the calculation of the parameter's gradients

4. Experimental Evaluation

4.1 Settings

We conducted our experiments on a modern multi-core CPU platform consisting of dual 2.3 GHz Intel Xeon E5-2699 v3 Haswell CPUs. Each CPU has 18 physical cores (36 hardware threads). Thus, the machine has 36 cores (72 hardware threads). It is equipped with 768 GB main memory. In our experiments, we just use 1 single thread to execute the benchmarks so that we can get the pure computing performance gain from the our automatic batching strategy. The operating system is Ubuntu 16.04 and all the code is complied with GNU GCC 5.4.

We used three benchmarks used in [13] and the experiments are based on implementation in the DyNet benchmark repositoryt*1. The information about the three benchmark and the corresponding parameters setting are described below:

- *BiLSTM*: This is a benchmark that trains a tagger using a bi-directional LSTM to extract features from the input sentence, which are then passed through a multi-layer perceptron to predict the tag of the word. The model used in this benchmark is based on the one proposed by Huang et al. [8] and is trained and tested on the WikiNER English Corpus [14]. In the experiments, the word embedding size is set to 128 while the LSTMs in either direction containing 256 hidden states. The size of multi-layer perceptron is set to 32.
- *BiLSTM w/char*: This benchmark is similar to the above one but has something different. In the first benchmark, words that have a frequency of at least five use an embedding specially for that word and other less frequent words use an embedding calculated by running a bi-directional LSTM over the characters in the word. This model can improve generalization with using the spelling of low-frequency words. In the experiments, char embedding size is 64 and the word embedding size is still 128. The size of hidden states in LSTM is 256 and the size of multi-layer perceptron is set to 32. The datasets used are the same with those in the first benchmark.
- *Tree-LSTM*: This benchmark is a sentiment analyzer based on tree-structured LSTMs [17]. Tree LSTMs are trained on the Stanford Sentiment Tree-bank regression task, which is provided in the benchmark repository.

In the experiments, all the benchmarks are executed with the batch size 64. For the first and second benchmark, 100 batches of samples are trained. For the third benchmark, all the samples are trained. All the experiments are implemented and executed on DyNet.

4.2 Performance

We executed the three benchmarks with different automatic batching strategy: By-depth, by-agenda and ours. Table 2 show the computing performance of the three benchmarks. The computing speed is shown as the number of sentences processed per second. The running time of each experiment is also recorded for comparing and reference. According to the table, we can find that the agenda-based automatic batching strategy and what we proposed one beat the depth-based strategy in all cases. The agenda-based one beats ours on *BiLSTM* and the *Tree-LSTM* while our proposed strategy performs faster on the *BiLSTM w/char* benchmark. Please notice that the running time of different benchmarks varies a lot. The *BiLSTM w/char*'s training time is about more than 10 time of the other two benchmarks.

4.3 Analysis

According to the Table. 2, it seems that our proposed strategy doesn't perform better than the agenda-based one. However, please notice that the running time of both these two strategy on the *BiLSTM* and *Tree-LSTM* benchmarks are almost the same. That means the overhead of our proposed strategy is a little higher than the gains from it. However, our proposed strategy is 46 seconds faster than the agenda-based one on the *BiLSTM w/char* benchmark, in which the model is much more complex than that in the first one. That means maybe our proposed strategy will gain more on the complex models and applications.

There are several possible reason that is able to explain why our proposed strategy is slower that the agenda-based one in some cases. The overhead comes from the data copy and additional analysis on the computation graph will reduce the effort of the gain from the batching chance in the backward propagation. When the overhead is larger than the gain, our proposed strategy is slower.

Due to the time limitation, our implementation and experiments focus on CPU. However, we think the our proposed strategy also benefits for GPU platform and the gain should be more since adding batching chance in the backward propagation can reduce the kernel launch times and make better usage of GPU's computation.

5. Conclusion and Future Work

In this work, we focus on automatic batching strategy applied on dynamic computation graphs of the neural networks for NLP. However, the existing strategies will miss some batching chance in the backward propagation during the training a model. Based on the agenda-based automatic batching strategy, we do some modification on it and develop our own one. The experiments shows that our strategy performs better on complex benchmarks.

We believe our strategy will performs much better than the others on GPU platforms since the taking the batching chance in the backward propagation in our way will benefit more on GPU. We will finish the implementation of strategy and execute experiments on GPU to verify what we believe in the future.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N.Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Schazeer. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[3] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *arXiv preprint arXiv:1511.08308*, 2015.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry

*1 https://github.com/neulab/dynet-benchmark

Table 2 Sentences/second and running time on various training tasks for increasingly challenging batching scenarios. (batchsize = 64)

Task	Speed (Sentences/s)			Running time (s)		
	By-depth	By-agenda	Proposed	By-depth	By-agenda	Proposed
BiLSTM	160.824	**172.33**	168.639	39.79 s	**37.14** s	37.95 s
BiLSTM w/char	15.19	17.65	**20.32**	421.44s	362.51 s	**315.01** s
Tree-LSTM	348.85	**359.35**	353.06	24.4 s	**23.68** s	24.11 s

Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*, 2015.

[6] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. Tree-to-sequence attentional neural machine translation. *arXiv preprint arXiv:1603.06075*, 2016.

[7] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[8] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[9] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *International Conference on Machine Learning*, pages 1378–1387, 2016.

[10] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

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

[12] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[13] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. *arXiv preprint arXiv:1705.07860*, 2017.

[14] Joel Nothman, Nicky Ringland, Will Radford, Tara Murphy, and James R Curran. Learning multilingual named entity recognition from wikipedia. *Artificial Intelligence*, 194:151–175, 2013.

[15] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[17] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[18] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, 2015.

[19] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.