Fast Parallel Training of Neural Language Models

Tong Xiao[†], Jingbo Zhu[†], Tongran Liu[‡], Chunliang Zhang[†]
[†]NiuTrans Lab., Northeastern University, Shenyang 110819, China
[‡]Institute of Psychology (CAS), Beijing 100101, China
{xiaotong,zhujingbo,zhangcl}@mail.neu.edu.cn, liutr@psych.ac.cn

Abstract

Training neural language models (NLMs) is very time consuming and we need parallelization for system speedup. However, standard training methods have poor scalability across multiple devices (e.g., GPUs) due to the huge time cost required to transmit data for gradient sharing in the back-propagation process. In this paper we present a sampling-based approach to reducing data transmission for better scaling of NLMs. As a "bonus", the resulting model also improves the training speed on a single device. Our approach yields significant speed improvements on a recurrent neural network-based language model. On four NVIDIA GTX1080 GPUs, it achieves a speedup of 2.1+ times over the standard asynchronous stochastic gradient descent baseline, yet with no increase in perplexity. This is even 4.2 times faster than the naive single GPU counterpart.

1 Introduction

Neural language models (NLMs) use continuous representations of words to predict their probability distributions. Several good models have been developed, showing very promising results in many natural language processing (NLP) tasks. The simplest of these uses feedforward neural networks to learn a distribution over the vocabulary given a number of history words [Bengio *et al.*, 2003], whereas others resort to recurrent neural networks (RNNs) for modeling sequences of arbitrary length [Mikolov *et al.*, 2010].

As in standard neural networks, a neural language model consists of multiple layers of neurons (or neural units). It represents each word using a group of neurons, and the number of model parameters increases linearly as more words are involved. Thus, the parameter set is huge for non-toy models trained on real world data though they have stronger ability for prediction. E.g., for a language model with 50k vocabulary size and 1k internal layer size, the parameter number is larger than 100 million. It is well known that it is slow to train such big models using stochastic gradient descent (SGD). Several research groups have been aware of this and designed good methods to speed up the learning process of NLMs [Bengio and Senécal, 2003; Morin and Bengio, 2005; Mnih and Hinton, 2008; Mnih and Teh, 2012; Zoph *et al.*, 2016].

However, training time is still unsatisfactory on a single device due to its limited computation capability. The next obvious step is toward parallelized training, e.g., running the work on a machine with two or more GPU cards. To do this, a popular way is data parallelism where different GPUs process different minibatches of samples in parallel. Unfortunately, a naive adaptation of existing NLMs in the high-latency parallelization scenario is inefficient [Bengio and Senécal, 2008]. The reason is that we have to accumulate the gradient for each processor and then share the gradient for model update. The process of gradient sharing requires huge data transmission which is extremely slow between two GPUs or between a GPU and a CPU. It is found to take up to 50% of the time of the back-propagation process in a minibatch[1].

A solution to this problem is to minimize the data transmission time. This motivates us to develop a sampling-based approach to reducing data transmission for better scaling of NLM training. In our approach, we sample a small portion of data for transmission and share gradients on the selected data. Beyond this, we apply the sampling method to gradient computation of a single minibatch for further system speedup. We experiment with the proposed approach in a state-of-the-art RNN-based language model on three different sized tasks. Experimental results show that it yields a significant scaling improvement over the baseline. On four NVIDIA GTX1080 GPUs, it achieves a 2.1x speedup over the standard asynchronous SGD method, yet with no increase in perplexity. This is 4.2 times faster than the naive single GPU system.

2 Background

2.1 Neural Language Model

Neural language models resemble the general architecture of neural network-based systems. They operate by creating connections between neurons which are organized into different layers. A layer can be defined as a procedure that maps an input vector $x = (x_1, ..., x_m)^T$ to an output vector $y = (y_1, ..., y_n)^T$, like this

$$y = f(s) \quad (1)$$

[1]The result was obtained on a machine with an Intel X99 mainboard and 4 GPUs. The minibatch size was set to 64.

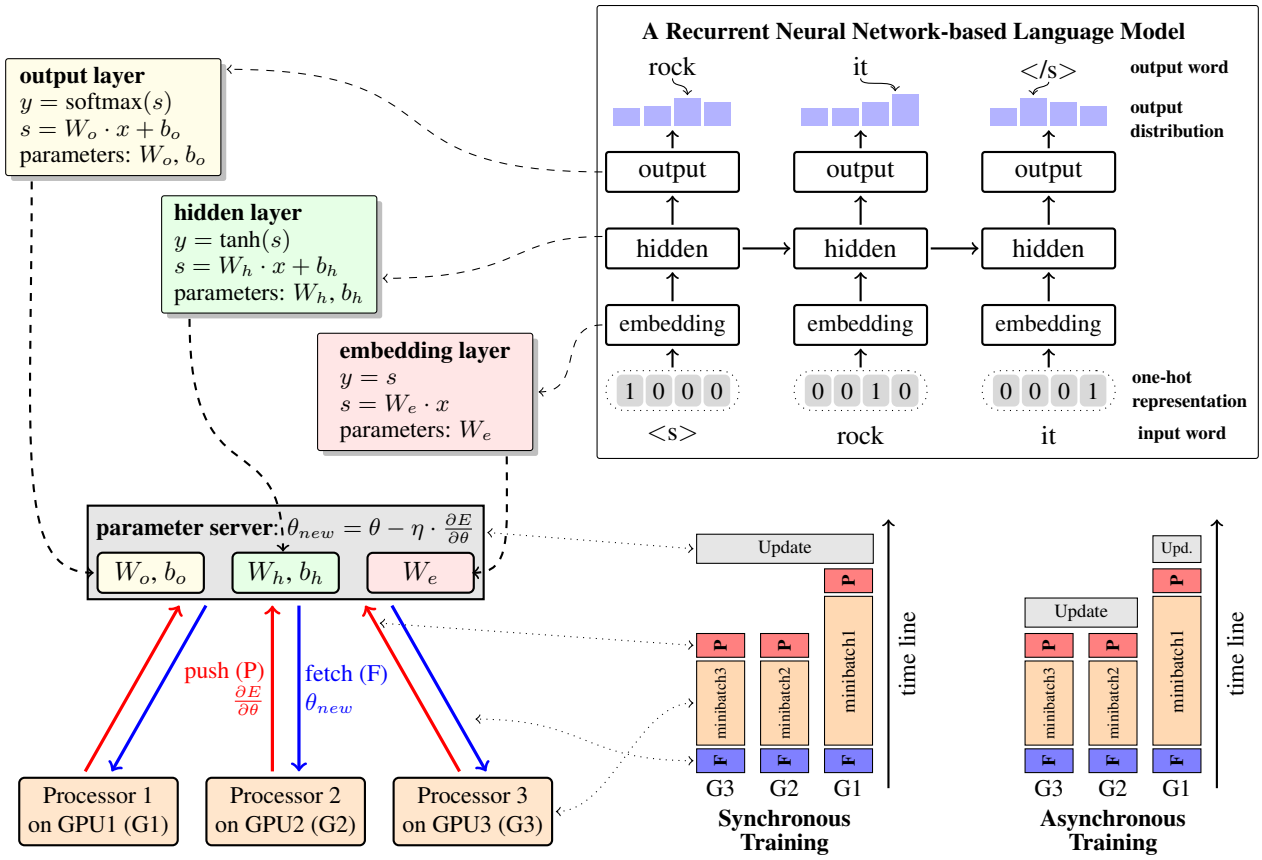


Figure 1: An illustration of RNN-based language models and parallel training. tanh(·) = hyperbolic tangent function. <s> and </s> represent the beginning and the end of a word sequence.

$$s = W \cdot x + b \quad (2)$$

where the model parameters are the $n \times m$ weight matrix W and the n-dimensional bias term b. $W_{i,j}$ represents the weight of the connection from the j-th neuron in the input to the i-th neuron in the output. $f(\cdot)$ is an activation function that generally performs a non-linear transformation on s before passing it to the output.

A neural language model consists of three types of layers

- **Input layer**. It receives the one-hot representation of a word and outputs its real-valued multi-dimensional representation (called word embedding).
- **Output layer**. This layer produces the probability for each word in the vocabulary by using the softmax function - $f(s_i) = \exp(s_i) / \sum_{i'} \exp(s_{i'})$
- **Hidden layer**. This layer is sandwiched between the input and output layers. Its output is regarded as a high-level representation of the input sequence. For a deep representation, several hidden layers can be stacked.

Different topologies of these layers can lead to different networks. A feedforward neural network-based language model takes a fixed number of history words as input and predicts the next one. For variable-length sequences of words, the recurrent neural network-based language model applies the same recurrent layer over the sequence of word embeddings, and the representation of the entire sequence is encoded in the output of the final hidden layer. See Figure **??** (top) for a running example of a RNN-based language model.

The inference process (or forward process) of an NLM is straightforward. E.g., in RNN-based LMs, we can pass various vectors through the network and obtain $\text{P}(w_k|w_1...w_{k-1})$ in the output layer corresponding to the k-th word. A popular method of training RNN models is stochastic gradient descent. We run a backward process to calculate the gradients for model parameters θ with respect to the error (denoted as $\frac{\partial E}{\partial \theta}$), namely back-propagation. Then we update the model by $\theta_{new} = \theta - \eta \cdot \frac{\partial E}{\partial \theta}$, where η is the learning rate.

2.2 Parallel Training

For faster training, a popular way is parallel SGD. In this work we follow the framework presented in [Dean *et al.*, 2012]. It distributes training across multiple model instances. Each model instance is assigned to a processor and is responsible for processing a minibatch of samples. The processors communicate with a centralized parameter server for model

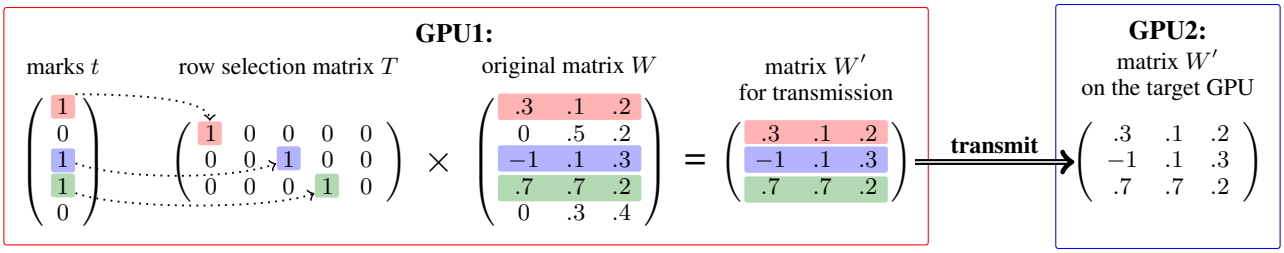


Figure 2: Sampling-based data transmission.

update, which keeps the latest state of all parameters of the model. The parameter server can be sharded across several devices or machines, where each shard stores and applies updates to a portion of the model parameters.

For a processor, before processing the minibatch, it asks the parameter server for a current copy of the model parameters. After processing the minibatch, it pushes the gradients to the server and sends an update message. See Figure **??** (bottom) for an illustration of the parallel training model used in this work.

In general, there are two paradigms for parameter update. The server can wait until all the processors finish their jobs, and then refresh the model parameters after receiving the gradients from the processors, i.e., the training runs in a synchronous manner. Alternatively, processors can run in an asynchronous way in which the server starts the update once a processor finishes the upload of the gradient information (Figure **??**). Theoretically, asynchronous parallel training can not guarantee good convergency as its synchronous counterpart - but it is faster and shows good empirical results in many large-scale systems.

Another way for fast parallel training is to overlap communication and computation. This can be done by using the multi-way data transmission model of modern GPUs. Note that the problem discussed in this paper is orthogonal to the method of overlapping data transmission and computation. In the following we restrict ourselves to the model where no such overlaps are employed.

3 Data Transmission via Sampling

In parallel training of NLMs, processing a minibatch requires two programs: 1) we first employ the back-propagation method to obtain the gradients for the minibatch, and then 2) we upload the gradients onto the parameter server and wait until the server finishes the update and sends back the new parameters. The second program takes little time if we run it on a single GPU with a high speed of on-chip data copy. But it is not the case when we switch to the multi-GPU environment. We observe that in a normal NLM setting, the data transmission between two GPUs takes 50% of the time of the back-propagation process. This means that we roughly have 50% inefficiency at best if we transmit the parameters with no other cost.

A possible solution to the issue is that processors communicate updates with the parameter server occasionally, rather than performing parameter update for every minibatch [Zhang *et al.*, 2015]. Though it can reduce the total amount of the transmitted data, we find that it is harmful to NLMs if we delay the update of the model parameters. Because the gradient matrices of softmax layers and hidden layers are big and dense, we need to keep the update in a relatively frequent fashion so that the learning can have reasonable convergence.

Here we present a method that balances the sending over time. The idea is pretty simple - we only select a small (necessary) portion of the data for transmission. Let W be an $n \times m$ matrix we need to copy from a GPU to another[2], and $W_{(i)}$ be the i-th row of W. For rows of W, we generate an n-dimensional vector $t = (t_1, ..., t_n)^T$, where $t_i = 0$ or 1. Then we transmit $W_{(i)}$ only if $t_i = 1$, and do not perform data copy for $W_{(i)}$ if $t_i = 0$. Here we call t the row-based mark for data transmission.

More formally, let n' be the number of the non-zero entries of t, and $\omega_{i'}$ be the index of the i'-th non-zero entry of t. We define an $n' \times n$ matrix T subject to $T_{i',j} = 1$ if $j = \omega_{i'}$, otherwise $T_{i',j} = 0$. In other words, $T_{i',j} = 1$ means that t_j is the i'-th non-zero entry of t. Then, we define the transmitted matrix W' to be:

$$W' \quad = \quad T' \times W \tag{3}$$

Eq. (3) actually performs a transformation on W by selecting rows of W with a non-zero mark in t. To implement this way of data copy between processor and server, all we need is to transmit W' from the source device and unpack it to W on the target device. Other parts of the training procedure can proceed as usual. See Figure 2 for an illustration of the data selection and transmission in our method.

Note that the matrix multiplication in Eq. (3) is not heavy because W' is very sparse. Also, transmitting W' from a processor to the server can be implemented via sparse matrix transmission. Here we choose a straightforward method that packs W' into a continuous block of data and unpacks it on the server end. Then the server can do sparse matrix operations after collecting the gradients from the processors. This process is basically the same as that of the baseline system and we can reuse the parameter fetching and pushing strategies.

To determine which rows of a matrix are selected, we choose different methods for different layers. In the softmax layer, each row of the weight matrix corresponds to a word.

[2]The matrix keeps either the model parameters or the gradients.

Rather than random sampling, a better way is to keep updating the model on the most frequent words, which can lead to fast convergence. Also, the words in the minibatch can be seen as positive samples that we intend to learn from. Thus, we design the following sets of words to make the vector t

- We select the words in the minibatch, denoted as V_{base}.
- We select $p[\%]$ of the most frequent words in the vocabulary, denoted as V_α.
- We randomly select $q[\%]$ of the words in the vocabulary, denoted as V_β.

Then, we generate the union set $V_{all} = V_{base} \bigcup V_\alpha \bigcup V_\beta$ and define that $t_i = 1$ only if $word_i \in V_{all}$. The use of V_β introduces randomness into training and makes the model behave more stable on the test data. For hidden layers, we generate t in a random manner, where we only use parameter q to control how often a row is selected to make t.

Another note on our method. For the embedding layer, the gradient is non-zero only for columns of the input words. In this case we employ a column-based method that transmits the gradients or model parameters for the active columns of the words in the minibatch. Though column-based data transmission is applicable to all other layers, using row-based transmission has several benefits (for softmax layers and hidden layers). First, a row in W is for an output neuron of the layer. Row selection is essentially a process that randomly omits the update of some neurons. This idea is similar to that of Dropout [Srivastava *et al.*, 2014], which has been proven to improve the robustness of neural network-based models on the unseen data. In addition, row-based copy is efficient because the data elements are stored in a block of contiguous memory space[3]. This can ease its implementation in practical systems.

4 Gradient Computation via Sampling

As we only transmit gradients for some of the neurons in a layer, it is natural to save more time by discarding the computing work of those neurons whose gradients are not sent to the parameter server. This makes more sense for the softmax layer whose weight matrix is huge. Generally, the big softmax layer is a bottleneck of the forward and backward processes for NLMs. Therefore, we apply the sampling method to gradient computation in the softmax layer.

Let $g = (g_1, ..., g_n)^T$ be the gold-standard word probability distribution for the softmax layer ($g_i = 1$ if $word_i$ is the correct answer), and $E(g, y)$ be the error function given the gold standard g and the layer output y. Here we choose cross entropy as the error function. By using Eqs. (1) and (2), we compute the gradient of the error with respect to $W_{i,j}$:

$$\frac{\partial E(g,y)}{\partial W_{i,j}} = (y_i - g_i) \cdot x_j \quad (4)$$

Eq. (4) implies a two-step procedure - for each word (with index i), we first calculate its output probability y_i in the forward phase, and then calculate $\frac{\partial E(g,y)}{\partial W_{i,j}}$ for each (i, j) in the backward phase. Given the fact that we only transmit $\frac{\partial E(g,y)}{\partial W_{i,j}}$ for $word_i \in V_{all}$, there is no need to compute the gradient for $word_i \notin V_{all}$ in the backward phase.

Also, we can obtain y_i based on a sampled set of words in the forward phase. Given the word indices for data transmission V_{all}, we randomly select additional $\mu[\%]$ of the words in the vocabulary (V_γ). Then, we compute y_i over the union set of the two: $V_f = V_{all} \bigcup V_\gamma$, like this

$$y_i = \begin{cases} \frac{\exp(s_i)}{\sum_{word_{i'} \in V_f} \exp(s_{i'})} & \text{if } word_i \in V_f \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

An efficient implementation of Eq. (5) is that we compute $\{s_i \mid word_i \in V_f\}$ and exclude $\{s_i \mid word_i \notin V_f\}$ from producing the output. It can benefit from faster operations on smaller matrices[4]. In this work we also apply this method to obtaining $\frac{\partial E(g,y)}{\partial b_i}$ and $\frac{\partial E(g,y)}{\partial x_j}$ for further speedup.

Note that the method presented here basically shares a similar idea as presented in [Bengio and Senécal, 2003; 2008]. Unlike previous work, we employ two different samplers for the forward-backward process. We use a slightly larger word set (V_f) to compute y_i in the forward phase, and a smaller word set (V_{all}) to compute $\frac{\partial E(g,y)}{\partial W_{i,j}}$ in the backward phase. It seems like a procedure that introduces some dummy words (i.e., words in V_γ controlled by μ) into backpropagation. In this way, the probability can be distributed to these dummy words. It is very similar to the way of generating noise for robust training [Hoaglin *et al.*, 1983]. In general, such a method is very beneficial to learning statistical models with good generalization ability.

entry	PTB	FBIS	Xinhua
training (words)	952K	4.94M	110M
validation (words)	76K	101K	115K
test (words)	85K	104K	118K
vocabulary size	15K	30K	40K
embedding size	512	512	1024
hidden size	512	512	1024
minibatch size	64	64	64

Table 1: Data and model settings.

5 Experiments

5.1 Experimental setup

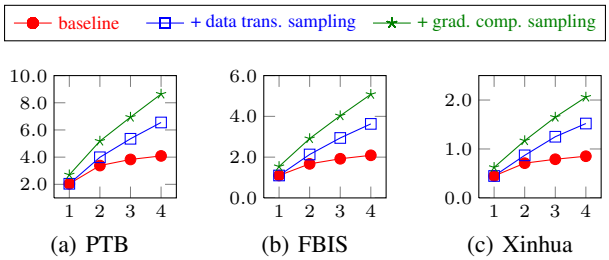
We experimented with the proposed approach in a recurrent neural network-based language model. We chose asynchronous SGD for the training framework. We ran all experiments on a machine with four NVIDIA GTX1080 GPUs. The quality of language modeling was measured by the average per-word log-scale probability (perplexity, or PPL for short). Three different sized tasks were chosen for training and evaluation.

[3]We assume matrices are in a row-major order for storage.

[4]Instead, we can use sparse matrix operations, e.g, multiplication on sparse matrices. But we found that it was faster to operate slimmed dense matrices.

Entry		PTB			FBIS			Xinhua		
		Valid.	Test	Speed	Valid.	Test	Speed	Valid.	Test	Speed
1 GPU	baseline	85.2	90.8	20.4	45.2	50.7	11.0	48.3	55.7	4.48
	+ gradient comp. sampling	86.7	87.2	26.9	45.0	49.1	15.4	48.6	55.7	6.27
2 GPUs	baseline	84.3	88.9	33.6	46.2	51.8	16.7	48.5	56.8	7.03
	+ data trans. sampling	88.1	86.2	39.1	48.8	50.2	21.4	50.1	54.2	8.64
	+ gradient comp. sampling	86.4	87.3	56.1	47.7	50.0	29.0	49.4	55.1	11.6
4 GPUs	baseline	87.0	87.4	40.8	45.5	51.0	20.9	47.6	56.0	8.46
	+ data trans. sampling	86.9	87.0	65.4	48.0	49.8	36.3	47.9	56.3	15.1
	+ gradient comp. sampling	85.0	88.2	86.2	47.3	51.2	50.8	46.2	55.7	20.4

Table 2: Perplexity and speed [k words/second] results of different systems.



(a) PTB (b) FBIS (c) Xinhua

Figure 3: Speed [10k words/second] against GPU number.

- **Penn Treebank (PTB)**. It is the standard data set used in evaluating LMs. We used sections 00-20 as the training data, sections 21-22 as the validation data, and sections 23-24 as the test data.
- **FBIS**. We used the Chinese side of the FBIS corpus (LDC2003E14) for a medium sized task. We extracted a 4,000 sentence data set for validation and a 4,027 sentence data set for test. We left the rest for training.
- **Xinhua**. For large-scale training, we generated a 4.5 million sentence set from the Xinhua portion of the English Gigaword (LDC2011T07). The validation and test sets (5,050 and 5,100 sentences) were from the same source but with no overlap with the training data.

For the FBIS and Xinhua tasks, we removed sentences of more than 50 words from the training data. All Chinese sentences in the FBIS data were word segmented using the tool provided within NiuTrans [Xiao *et al.*, 2012]. We used vocabularies of 15K, 30K and 40K most frequent words for the three tasks. The out of vocabulary (OOV) words were marked with a special UNK token. See Table 1 for a summary of the data and model settings.

For RNN-based LM, we chose long short-term memory (LSTM) for the recurrent unit [Hochreiter and Schmidhuber, 1997]. The weights in all the networks were initialized with a uniform distribution in [-0.1, 0.1]. The gradients were clipped so that their norm was bounded by 3.0. For all experiments, training was iterated for 20 epochs. We started with a learning rate of 0.7 and then halved the learning rate if the perplexity increased on the validation set. In our sampling-based approach, we set $\mu = 5\%$ by default. For softmax layers, we set $p = 10\%$ and $q = 10\%$. For hidden layers, we set $q = 90\%$, which could prevent the system from disturbing the training of the LSTM layer too much.

5.2 Results

We first compare the system speed by varying the number of GPUs. Figure 3 shows that the standard asynchronous SGD method (baseline) has poor scaling due to the time cost of data transmission. The NLM can be scaled up better when sampling-based data transmission is employed. Also, the system runs faster when the sampling method is applied to gradient computation in the back-propagation process. It achieves over 1.3x speedups on a single GPU for all three of the tasks. Under the 4-GPU setting, it achieves a speedup of 2.1x over the asynchronous SGD baseline, and is even 4.2 times faster than the naive 1-GPU counterpart. More interestingly, it is observed that more speed improvements can be obtained on bigger models (as in Xinhua). This is because that sampling over a larger parameter set discards more data for transmission and thus saves more time.

Then we study the quality of various NLMs as measured by perplexity (Table 2). We see that the perplexity of sampling-based models is comparable to that of the baseline, and the result is consistent under different settings of GPU number. Moreover, we plot the learning curve on the validation set of the Xinhua task. Figure 4 shows that our sampling-based method has good convergence. The perplexity drops significantly in the first 5 epochs and tends to coverage in 8$\sim$10 epochs.

As the percentage of time spent exchanging parameters depends on the size of minibatch, it is worth a study on how our method performs for different sized minibatches. Figure 5 shows that the speed improvement persists under different settings of minibatch size. For smaller minibatches, larger speed improvements are obtained because more time (in percentage) is spent on communicating gradients.

Also, we do sensitivity analysis on the hyper-parameters of our method (p, q, and μ). We find that it helps if we sample less data in data transmission for the softmax layer. The system can run faster by using smaller p and q while it in turn results in more training epochs to converge. The interesting observation here is that using two-stage sampling in gradient computation is promising (see Section 4). Figure 6 shows that setting μ around 0.03$\sim$0.05 leads to better convergence, which indicates that introducing some noise into training is

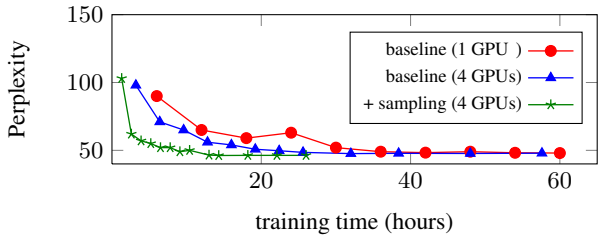


Figure 4: Perplexity against training time (Xinhua).

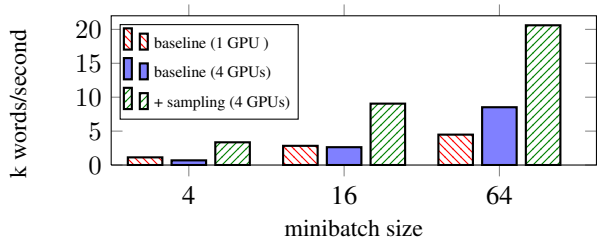


Figure 5: Speed against minibatch size (Xinhua).

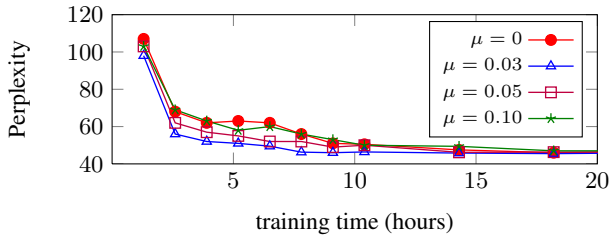


Figure 6: Learning curves in different settings of μ (Xinhua).

good for generalizing the model on the unseen data.

In addition to vanilla SGD, the sampling-based method is applicable to other training and data transmission paradigms. For a further improvement we apply the 1-bit method [Seide *et al.*, 2014] on top of our system. We find that the 1-bit method can speed up our sampling-based system, but with a decrease of language modeling quality in the default setting of sampling. We suspect that it is due to the loss of accuracy for the model when we quantize a small portion of gradients. For a comparable perplexity result, we enlarge both p and q to 25% when the 1-bit method is employed. The final speed up is 2.6x over the baseline on 4 GPUs. It is 5.2 times faster than the single GPU baseline.

6 Related Work

It has been observed that increasing the scale of learning with respect to training samples can drastically improve the performance of NLMs [Józefowicz *et al.*, 2016]. In response, it is natural to scale up training algorithms through parallelization. In machine learning, many researchers have investigated parallel training methods for statistical models [Mann *et al.*, 2009; McDonald *et al.*, 2010; Zinkevich *et al.*, 2010; Dean *et al.*, 2012]. Some of this research focuses on delayed gradient updates, which can reduce the communication between machines [Langford *et al.*, 2009; Agarwal and Duchi, 2011]. Other groups work on problems with sparse gradients and develop faster asynchronous stochastic gradient descent in a lock-less manner [Niu *et al.*, 2011]. These methods work well for convex and/or sparse problems but have not been well studied in the training of large-scale NLMs. We note that, despite significant development effort, we were not able to have good scaling for our NLM using existing methods; it was this experience that led us to develop a model-specific method in this paper.

In the context of language modeling, several research groups have addressed the computational challenge in learning large vocabulary NLMs. They proposed good solutions to the problem, including importance sampling [Bengio and Senécal, 2003; 2008], noise contrastive estimation [Mnih and Teh, 2012], and hierarchical softmax [Morin and Bengio, 2005; Mnih and Hinton, 2008]. Most of these focus on the scaling problem on a single processor or running small models in a multi-CPU environment. More recently, Zoph et al. [2016] trained their 4-layer LSTM-based NLM by laying each layer on a GPU. Though their model parallelism method works for multi-layer RNNs, it is only applicable to certain architecture where different parts of the model can be processed in parallel. Another interesting study reported that NLMs could be run on a GPU cluster [Józefowicz *et al.*, 2016], but the details were missing. It is still rare to see in-depth studies on scaling up the training of NLMs over modern GPUs through data parallelism.

Another related study is [Seide *et al.*, 2014]. They used the 1-bit method to ease data transmission by quantizing gradients. Actually, the 1-bit method and our sampling-based method are two research lines that are compatible to each other. As is discussed in Section 4, they can work together for a further speedup. Note that the idea of sampling is not new in language modeling. E.g., our method of sampling-based gradient computation and importance sampling [Bengio and Senécal, 2003] are two variants on a theme. Unlike previous work, we develop a different sampling strategy and use two samplers for the forward and backward processes.

7 Conclusions

We have presented a sampling-based approach to scaling up the training of NLMs over multiple devices. It reduces cost data transmission and gradient computation in a distributed training environment. We have demonstrated that on 4 GPUs the proposed approach can lead to a speedup of 2.1+ times over the standard asynchronous SGD, with no loss in language modeling quality. It is even 4.2 times faster than the single GPU baseline without sampling.

Acknowledgments

This work was supported in part by the National Science Foundation of China (61672138 and 61432013) and the Fundamental Research Funds for the Central Universities. The authors would like to thank anonymous reviewers, Fuxue Li, Yaqian Han, Ambyer Han and Bojie Hu for their comments.

References

[Agarwal and Duchi, 2011] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 873–881, Granada, Spain, 2011.

[Bengio and Senécal, 2003] Yoshua Bengio and Jean-Sbastien Senécal. Quick training of probabilistic neural nets by importance sampling. In *Proceedings of AISTATS*, 2003.

[Bengio and Senécal, 2008] Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Network*, 4:713–722, 2008.

[Bengio *et al.*, 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[Dean *et al.*, 2012] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Q Le, Mark Z Mao, Marcaurelio Ranzato, Andrew W Senior, Paul Tucker, et al. Large scale distributed deep networks. *Information processing systems*, pages 1223–1231, 2012.

[Hoaglin *et al.*, 1983] David C. Hoaglin, Frederick Mosteller, and John W. Tukey. *Understanding Robust and Exploratory Data Analysis*. John Wiley, 1983.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–17808, 1997.

[Józefowicz *et al.*, 2016] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[Langford *et al.*, 2009] John Langford, Alexander J Smola, and Martin Zinkevich. Slow learners are fast. In *Proceedings of the 23rd Annual Conference on Neural Information Processing Systems (NIPS)*, pages 2331–2339, Vancouver, Canada, 2009.

[Mann *et al.*, 2009] Gideon Mann, Ryan T. McDonald, Mehryar Mohri, and Dan Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *Proceedings of the 23rd Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1231–1239, Vancouver, Canada, 2009.

[McDonald *et al.*, 2010] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 456–464, Los Angeles, California, June 2010.

[Mikolov *et al.*, 2010] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Černocký Jan, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of INTERSPEECH*, pages 1045–1048, Makuhari, Chiba, Japan, September 2010.

[Mnih and Hinton, 2008] Andriy Mnih and Geoffrey Hinton. A scalable hierarchical distributed language model. In *Proceedings of NIPS*, pages 1081–1088, Vancouver, Canada, December 2008.

[Mnih and Teh, 2012] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. In *Processings of the 29th International Conference on Machine Learning (ICML)*, pages 1751–1758, Edinburgh, Scotland, June 2012.

[Morin and Bengio, 2005] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Proceedings of AISTATS*, pages 246–252, 2005.

[Niu *et al.*, 2011] Feng Niu, Benjaminm Recht, Christopher Ré, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, Granada, Spain, 2011.

[Seide *et al.*, 2014] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Proceedings of INTERSPEECH*, pages 1058–1062, Singapore, September 2014.

[Srivastava *et al.*, 2014] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[Xiao *et al.*, 2012] Tong Xiao, Jingbo Zhu, Hao Zhang, and Qiang Li. NiuTrans: An Open Source Toolkit for Phrase-based and Syntax-based Machine Translation. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistic (ACL): System Demonstrations*, pages 19–24, Jeju Island, Korea, July 2012.

[Zhang *et al.*, 2015] Sixin Zhang, Anna Choromanska, and Yann Lecun. Deep learning with elastic averaging sgd. In *Proceedings of the Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, pages 685–693, Montréal, Canada, 2015.

[Zinkevich *et al.*, 2010] Martin A. Zinkevich, Alex Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *Proceedings of the 24th Annual Conference on Neural Information Processing Sytems (NIPS)*, pages 2595–2603, Vancouver, Canada, 2010.

[Zoph *et al.*, 2016] Barret Zoph, Ashish Vaswani, Jonathan May, and Kevin Knight. Simple, fast noise-contrastive estimation for large rnn vocabularies. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1217–1222, San Diego, California, June 2016.