Knowledge and Information Systems
https://doi.org/10.1007/s10115-020-01487-8

REGULAR PAPER

Fast LSTM by dynamic decomposition on cloud and distributed systems

Yang You[1] · Yuxiong He[2] · Samyam Rajbhandari[2] · Wenhan Wang[2] · Cho-Jui Hsieh[3] · Kurt Keutzer[4] · James Demmel[4]

Received: 20 June 2020 / Accepted: 27 June 2020

Abstract
Long short-term memory (LSTM) is a powerful deep learning technique that has been widely used in many real-world data-mining applications such as language modeling and machine translation. In this paper, we aim to minimize the latency of LSTM inference on cloud systems without losing accuracy. If an LSTM model does not fit in cache, the latency due to data movement will likely be greater than that due to computation. In this case, we reduce model parameters. If, as in most applications we consider, the LSTM models are able to fit the cache of cloud server processors, we focus on reducing the number of floating point operations, which has a corresponding linear impact on the latency of the inference calculation. Thus, in our system, we dynamically reduce model parameters or flops depending on which most impacts latency. Our inference system is based on singular value decomposition and canonical polyadic decomposition. Our system is accurate and low latency. We evaluate our system based on models from a series of real-world applications like language modeling, computer vision, question answering, and sentiment analysis. Users of our system can use either pre-trained models or start from scratch. Our system achieves 15× average speedup for six real-world applications without losing accuracy in inference. We also design and implement a distributed optimization system with dynamic decomposition, which can significantly reduce the energy cost and accelerate the training process.

Keywords LSTM · Fast inference · Dynamic decomposition

1 Introduction

Long short-term memory (LSTM) is a powerful deep learning technique that has been used in many real-world applications like language modeling [26], machine translation [56], speech recognition [57], and visual question answering [23]. Because of LSTM's good performance in these applications, there is a growing interest in using LSTM inference for recommendations systems in domains such as movie recommendation [55] or search-based online advertising [62]. While researchers keep improving the training speed [60,61], the inference

✉ Yang You
youyang@cs.berkeley.edu

Extended author information available on the last page of the article

Published online: 19 July 2020

speed for these LSTM-based systems can be a major bottleneck and delays degrade the user experience. Our specific target is to reduce the LSTM inference latency to less than 50 ms to ensure a good interactive user experience.

The previous work on improving the speed of LSTMs has focused on reducing the number of parameters in the LSTM. Although reducing number of parameters is important, we find number of floating point operations (flops) is the major overhead when the model can fit the cache of cloud servers. In this situation, the LSTM inference speed is dependent on the number of flops. When the model cannot fit in the cache, the data movement overhead will usually be higher than the flops overhead. In order to reduce number of flops, we study a series of matrix and tensor decomposition techniques. We design and implement our inference system based on singular value decomposition (SVD) [11] and canonical polyadic decomposition (CPD) [10]. Our system provides dynamic selection support to find the trade-off between data movement and flops.

We evaluate our approaches based on a series of real-world applications such as compositional modular network (CMN), sentiment analysis for IMDB dataset, language modeling for the PTB dataset, end-to-end module network (N2NMN), and one billion word language modeling by BIG-LSTM. For ease of comparison, we also include the results on the well-known handwritten digits recognition for MNIST dataset. Our first approach was to directly apply decomposition in LSTM inference. However, because the range of singular values in some applications are narrow, this approach does not work as well for them. The error rates for these applications are high. To reduce the LSTM latency for all the applications, we combine the regular training and decomposition-aware training together. Once we have a well-trained LSTM model, we use the SVD-aware training to fine-tune it. By training the same number of epochs, we can achieve 11× speedup without losing accuracy in inference for MNIST dataset; however, in practice, we usually train one model and use it for weeks on a variety of platforms. Thus, training time is not the principal concern for us. By training longer and building a smaller but still accurate model, we can achieve 43× speedup without losing accuracy in inference for the MNIST dataset. To allow the users to start from scratch (without a well-trained model or enough hardware memory), our system also provides the alternative of using only decomposition-aware training. Overall, our system achieves 15× average speedup for six real-world applications without losing accuracy in inference. Our results are discussed in detail in Sect. 4. The data flow of our system is shown in Fig. 1. Our contributions include:

- We are not the first team to apply the low-rank technique in processing neural networks models. But we believe we are the first to decide what factorization to use at run-time in a dynamic way.

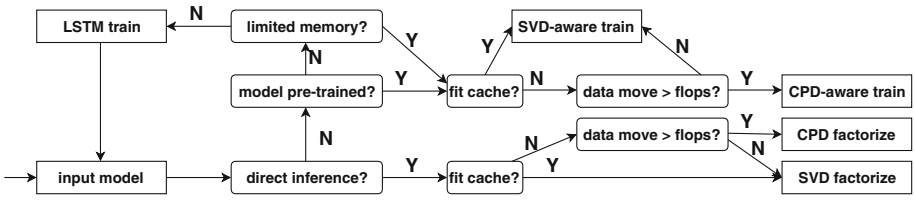


Fig. 1 Data flow of our system. The input is a well-trained or non-trained LSTM model. The output is a low-rank LSTM model for cloud system. Our target is to minimize the latency of LSTM inference at run time. We take both the data movement overhead and floating-point operation overhead into consideration. Our system dynamically picks the best online approach. For the users without enough hardware memory for training LSTM model, our system is able to start from scratch

- We believe that we implemented the first dynamic inference system that can achieve more than 10 times speedup for several real-world applications. People can easily use our system in their own applications.
- Even users with limited hardware resources can use our system. We provide low-latency (< 50 ms) responses for LSTM-inference on cloud servers.
- We design and implement a distributed optimization system with dynamic decomposition, which can significantly reduce the energy cost and accelerate the training process.

2 Background and related work

The target of this paper is to speed up LSTM inference on Cloud systems. We review the computational pattern of LSTM and previous techniques in this section.

2.1 Long short-term memory (LSTM)

Because of the exploding and vanishing gradient problems, it is hard to learn the long-term dependencies with the approach of gradient descent that is universally used in training deep recurrent neural networks (RNN) [3,38]. To solve this problem, Hochreiter and Schmidhuber [21] proposed the long short-term memory (LSTM) technique. The mathematical definition is presented in Eqs. (1)–(6) where σ denotes the Sigmoid activation function and $\odot$ denotes the element-wise operation.

$$\boldsymbol{i_t} = \sigma(\boldsymbol{W_{xi}}\boldsymbol{x_t} + \boldsymbol{W_{hi}}\boldsymbol{h_{t-1}} + \boldsymbol{b_i}) \tag{1}$$

$$\boldsymbol{f_t} = \sigma(\boldsymbol{W_{xf}}\boldsymbol{x_t} + \boldsymbol{W_{hf}}\boldsymbol{h_{t-1}} + \boldsymbol{b_f}) \tag{2}$$

$$\boldsymbol{o_t} = \sigma(\boldsymbol{W_{xo}}\boldsymbol{x_t} + \boldsymbol{W_{ho}}\boldsymbol{h_{t-1}} + \boldsymbol{b_o}) \tag{3}$$

$$\boldsymbol{g_t} = \tanh(\boldsymbol{W_{xc}}\boldsymbol{x_t} + \boldsymbol{W_{hc}}\boldsymbol{h_{t-1}} + \boldsymbol{b_c}) \tag{4}$$

$$\boldsymbol{c_t} = \boldsymbol{f_t} \odot \boldsymbol{c_{t-1}} + \boldsymbol{i_t} \odot \boldsymbol{g_t} \tag{5}$$

$$\boldsymbol{h_t} = \boldsymbol{o_t} \odot \tanh(\boldsymbol{c_t}) \tag{6}$$

LSTM has the following recurrent computational pattern. The input of LSTM is a squence of vectors, and we denote T as the sequence length. Here, $t \in \{1, 2, 3, \ldots, T\}$ denotes the time step. $\boldsymbol{i_t}$, $\boldsymbol{f_t}$, and $\boldsymbol{o_t}$ denotes the input gate, forget gate, and output gate, respectively. $\boldsymbol{g_t}$ controls the update of the state. At step t, $\boldsymbol{c_t}$ denotes the LSTM cell state, $\boldsymbol{x_t}$ denotes the input vector, and $\boldsymbol{h_t}$ denotes the hidden state. $\boldsymbol{x_t}$ is a l-by-1 vector, and $\boldsymbol{h_t}$ is a d-by-1 vector. $\boldsymbol{W_{xi}}$, $\boldsymbol{W_{hi}}$, $\boldsymbol{W_{xf}}$, $\boldsymbol{W_{hf}}$, $\boldsymbol{W_{xo}}$, $\boldsymbol{W_{ho}}$, $\boldsymbol{W_{xc}}$, and $\boldsymbol{W_{hc}}$ are the cell matrices, with the dimensions of d-by-l, d-by-d, d-by-l, d-by-d, d-by-l, d-by-d, d-by-l, and d-by-d, respectively. In the practical implementation like TensorFlow [1], these eight matrices are merged together to a $4d$-by-$(l+d)$ matrix $\boldsymbol{W}$. We may also write it as $(l+d)$-by-$4d$ in this paper. Meanwhile, x_t and h_{t-1} are concatenated together to a $(l+d)$-by-1 vector x. Therefore, the kernel operation of LSTM inference is a dense matrix dense vector multiplication $\boldsymbol{Wx}$. In order to minimize the inference latency, we need to make this matrix operation as fast as possible. We want to explore the special structure within matrix $\boldsymbol{W}$.

2.2 Previous work

To speed up LSTM inference, we study low-rank techniques or matrix factorization approaches. The matrix or tensor decomposition technique has been used for designing

CNNs. In VGG [47], GoogleNet [47], and ResNet [19] models, the researchers factorize a large convolutional layer into the stack of layers with smaller filters. Low-rank techniques were also used to accelerate neural network training and reduce the number of parameters [13,18,29,40,50,51]. The block-diagonal technique is also a possible technique for low- rank recurrent neural network with fewer parameters [48]. The pruning [18] can reduce model size, but matrix multiplication with irregular sparsity often is slower than dense multiplication on the general-purpose processors like CPUs or GPUs. The irregular sparsity requires special-purpose hardware to achieve a high speed and a low power [17]. This paper is focused on the deep learning inference devices used on cloud systems (i.e., CPUs or GPUs). So pruning technique does not work well in reducing inference latency. There are three main problems for using irregular sparsity [36]:

- **Additional storage from indices** Sparse formats use additional memory to track the location of each nonzero element. For example, the format like compressed-sparse-column (CSC) needs at least one additional indices for each nonzero element. The precision of these indices is dependent on the matrix dimension. For example, a 64K-by-64K needs a 16-bit index. In this way, each nonzero element brings at least 16-bit storage overhead. If we use 16-bit precision to store the neural network weight, it will bring 150% overhead.
- **Random memory accesses** Caches lines, DRAM row buffers, and TLBs get the best performance when memory is accessed in large contiguous units. A typical cache line is 64 bytes, and a typical DRAM row is 4 KB. The platform's performance is lower when the memory is accessed by irregular fine-grained operations.
- **Array-data-paths** The array-data-paths are highly important for achieving high performance in current architectures. However, fine-grained sparsity is not able to make full use of array-data-paths (e.g., the 16×16 TensorCore units in Volta GPU).

It is also interesting to explore sparse structures within LSTM models [53]; however, most of our experimental results show the cell matrices are totally dense. The recent work on softmax approximation [16,46] is another technique to speed up LSTM training, which is useful for language modeling problems. This paper is focused on inference speed rather than training speed. Since our target applications are broader than language modeling, this paper is focused on the LSTM cell matrix rather than softmax layer. Moreover, the softmax layer computation is an one-time cost and can be finished offline.

Application-specific integrated circuits (ASICs) are another direction of accelerating the training and inference of deep neural networks. For example, field programmable gate array (FPGA) has been used to increase the throughput of deep learning systems. FPGA is promising because it can maximize the parallelism and reduce the energy consumption. In 1996, Cloutier et al. [8] built the virtual image processor (VIP) on Altera EPF81500 FPGA, which was the first ConvNet (convolutional neural network) implementation on FPGA. The architecture of VIP is based on SIMD (single-instruction stream multiple-data streams). The processing elements (PEs) of VIP were connected by a 2D torus topology. The key novelty of VIP is to employ the low-accuracy arithmetic without full-fledged multipliers, which can be much more efficient.

In recent years, digital signal processing (DSP)-based FPGA is becoming more popular. This kind of architecture had a lot of multiply-and-accumulate (MAC) units, which made the fast and low-energy deep learning systems possible. Since then, the FPGA-based deep learning systems are focused on optimizing the ConvNet operation, which is the kernel part of the learning computational engine. There are numerous research papers on this direction [4,6,15,37,42,54,64]. Most of them reported very impressive performance improvement by implementing FPGA-based systems. An example of state-of-the-art systems implemented

in recent years is Caffeine [63]. Caffeine is a software/hardware interface system for fast ConvNet on FPGAs. Caffeine is able to automatically pick the optimal hardware parameters for the users. However, the users of Caffeine system are required to provide deep learning framework and FPGA device specifications to the system. Different from VIP [8], Caffeine system is built on top of high-level synthesis systolic-like computer architecture (similar to Google's TPU [25]). The systolic-like architecture is optimized for conducting fast matrix–matrix multiply. Caffeine also is a multi-level data parallelism system. More related work on FPGA-based deep learning system can be found in the survey by Shawahna et al. [44].

2.3 Singular value decomposition (SVD)

Single value decomposition (SVD) is a stable matrix decomposition technique with many successful applications [11]. Given a matrix $\boldsymbol{W}$ with the dimension of m-by-n, SVD factorizes $\boldsymbol{W}$ into two matrices $\boldsymbol{U}$, $\boldsymbol{V^T}$, and a diagonal matrix $\boldsymbol{\Sigma}$ (Eq. (7)). The columns of $\boldsymbol{U}$, $\boldsymbol{V^T}$ are the singular vectors of matrix $\boldsymbol{M}$. The dimension of $\boldsymbol{U}$ and $\boldsymbol{V^T}$ are m-by-R and R-by-n, respectively. $\boldsymbol{\Sigma}$ is a R-by-R diagonal matrix with nonnegative real singular values in descending order. Here, R is the rank of $\boldsymbol{W}$. SVD can be rewritten as $\boldsymbol{W} = \boldsymbol{W_1 W_2}$ where $\boldsymbol{W_1}$ is $\boldsymbol{U}\sqrt{\boldsymbol{\Sigma}}$ and $\boldsymbol{W_2}$ is $\sqrt{\boldsymbol{\Sigma}}\boldsymbol{V^T}$. The number of parameters in the original matrix $\boldsymbol{W}$ is mn. The number of parameters after SVD is reduced to only $R(m+n)$. If R is much smaller than $\min(m, n)$, then we essentially reduce the number of parameters from $\Theta(mn)$ to $\Theta(m+n)$.

$$\boldsymbol{W} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V^T} = (\boldsymbol{U}\sqrt{\boldsymbol{\Sigma}})(\sqrt{\boldsymbol{\Sigma}}\boldsymbol{V^T}) = \boldsymbol{W_1 W_2} \tag{7}$$

2.4 Tensor decomposition

A tensor is a generalization of vector and matrices to n dimensions. When n is one, a tensor is a vector. When n is two, a tensor is a matrix. Typically, only when n is larger than two, we refer to the data representation as a tensor. The most widely used tensor decomposition techniques include canonical polyadic decomposition (CPD) [10] and Tucker decomposition [52]. In our experiments, CPD performs better than Tucker decomposition, so this paper is focused on CPD. We use $\boldsymbol{X}$ to denote a tensor with the dimension (d_1, d_2, d_3). The algorithm decomposes the tensor $\boldsymbol{X}$ into R sets. Each set has three vectors $\boldsymbol{a_i}$, $\boldsymbol{b_i}$, and $\boldsymbol{c_i}$ where $i \in \{1, 2, \ldots, R\}$. The length of $\boldsymbol{a_i}$, $\boldsymbol{b_i}$, and $\boldsymbol{c_i}$ is d_1, d_2, and d_3, respectively. The number of parameters in the original tensor $\boldsymbol{X}$ is $d_1 d_2 d_3$. The number of parameters after CPD is reduced to only $R(d_1+d_2+d_3)$. If R is much smaller than the minimum of d_1, d_2 and d_3, then we essentially reduce the number of parameters from $\Theta(d_1 d_2 d_3)$ to $\Theta(d_1+d_2+d_3)$ Fig 2.

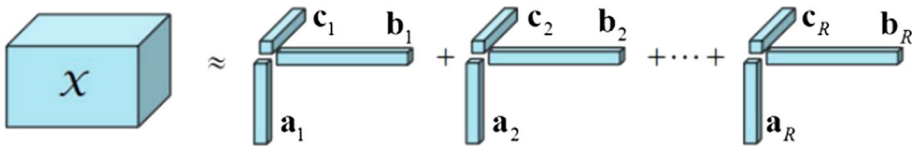


Fig. 2 CP decomposition (CPD) approach. If the rank is R, the algorithm decomposes a d_1-by-d_2-by-d_3 tensor $\boldsymbol{X}$ into R sets. Each set is made up of three vectors $\boldsymbol{a_i}$, $\boldsymbol{b_i}$, and $\boldsymbol{c_i}$ where $i \in \{1, 2, \ldots, R\}$. The length of $\boldsymbol{a_i}$, $\boldsymbol{b_i}$, and $\boldsymbol{c_i}$ is d_1, d_2, and d_3, respectively

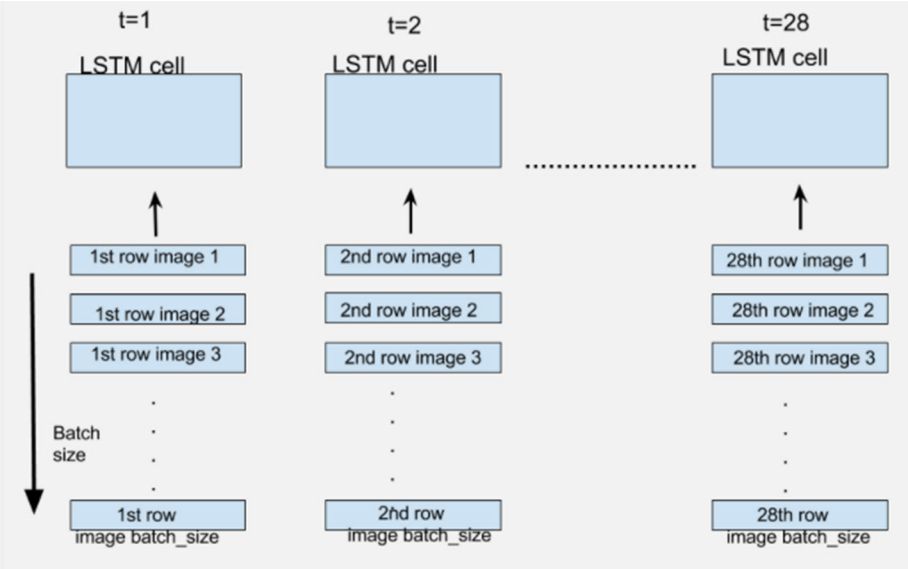


Fig. 3 We partition a 28-by-28 image into 28-step sequence. Each row (a vector with 28 elements) of the image is an input step. Totally, we have 60K images. Batch size means the number of vectors we process at each step

3 Applications

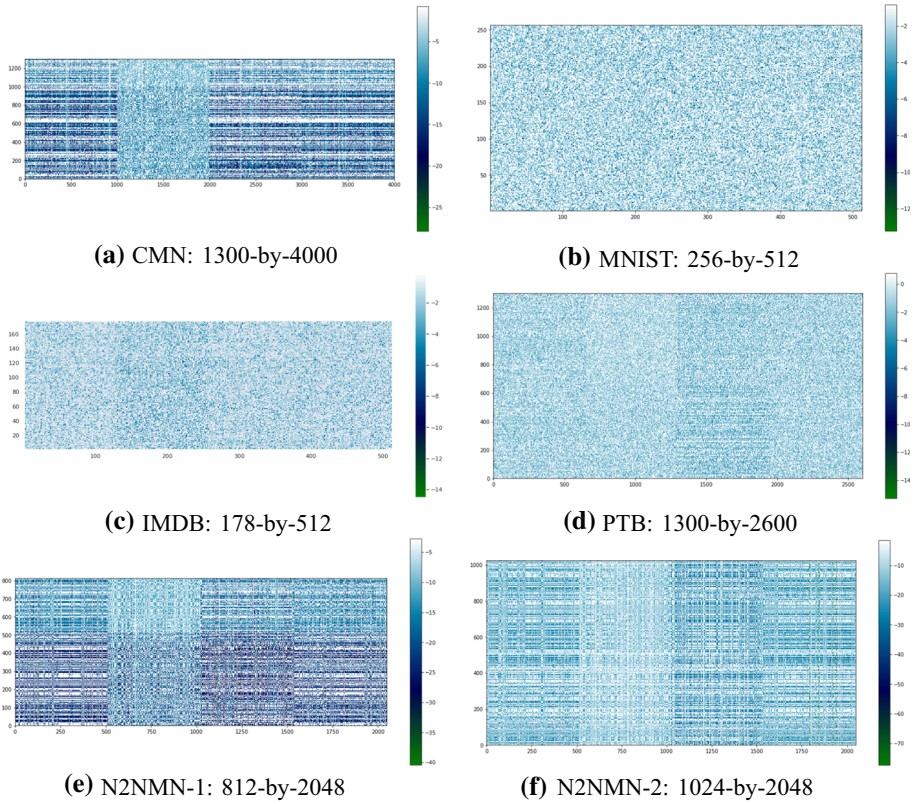
We evaluate our methods on a series of real-world deep learning applications on cloud systems. Our goal is to achieve fast inference at runtime.

3.1 Application1: compositional modular network (CMN)

Compositional modular network (CMN) [23] does linguistic analysis and visual inference end-to-end. The architecture is built on top of LSTM modules. It inspects local regions and pairwise interactions between regions. Given an image and an expression, it learns to parse the expression into vector representation of subject, relationship and object with attention, and align these textual components to image regions with two types of modules. The localization module outputs scores over each individual region, while the relationship module produces scores over region pairs. These outputs are integrated into final scores over region pairs, producing the top region pair as grounding result. The $\boldsymbol{W}$ is a 1300-by-4000 dense matrix, which is shown in Fig. 4a.

3.2 Application2: handwritten digits recognition for MNIST dataset

MNIST dataset [30] has 60k training examples and 10k test examples. Each sample is a 28-by-28 handwritten digit image. We use this dataset to train a pure-LSTM model. We partition each image as 28-step input vectors. The dimension of each input vector is 28-by-1 (Fig. 3). Then, we have a 128-by-28 transform layer before the LSTM layer, which means the actual LSTM input vector is 128-by-1. The hidden dimension of LSTM layer is 128. Thus, the $\boldsymbol{W}$



(a) CMN: 1300-by-4000

(b) MNIST: 256-by-512

(c) IMDB: 178-by-512

(d) PTB: 1300-by-2600

(e) N2NMN-1: 812-by-2048

(f) N2NMN-2: 1024-by-2048

Fig. 4 Cell matrices of our LSTM models (log scale). All these matrices are totally dense. We observe that there are many relatively large numbers in the matrices of MNIST, IMDB, and PTB. For other three matrices, most of the numbers are relatively tiny

of LSTM layer is a 256-by-512 matrix. From Fig. 4b, we observe that the matrix is totally dense.

3.3 Application3: sentiment analysis for IMDB dataset

IMDB [34] is a dataset for binary sentiment classification. The dataset provides a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. The training dataset has 12,500 positive reviews and 12,500 negative reviews. The vocabulary of this dataset has 400,000 words. To reduce the storage overhead, we use a 400,000-by-50 embedding matrix to hold all of the word vector values. So the input vector length is 50. The maximum sequence is 250. The hidden dimension of LSTM layer is 128. Thus, the $\boldsymbol{W}$ is a 178-by-512 dense matrix. From Fig. 4c, we observe that the matrix is totally dense.

3.4 Application4: language modeling for PTB dataset

The Penn Treebank (PTB) [35] dataset selected 2,499 stories from a three year Wall Street Journal (WSJ) collection of 98,732 stories for syntactic annotation. These 2,499 stories have

been distributed in both Treebank-2 (LDC1999T42) and Treebank-3 (LDC1999T42) releases of PTB. The vocabulary has 10,000 words. After word embedding, the input vector length is 650. The sequence length is 35. Our LSTM model has two layers. The hidden dimensions of both these two layers are 650. For both layers, the $\boldsymbol{W}$ is an 1300-by-2600 dense matrix. Since these two matrices are similar to each other, we only show one of them in Fig. 4d. We use accuracy to evaluate the correctness of our LSTM model. We first apply a softmax operation to get the predicted probabilities of each word for each output of the LSTM network. We then make the network predictions equal to those words with the highest softmax probability by using the argmax function. These predictions are then compared to the actual target words and then averaged to get the accuracy.

3.5 Application5: end-to-end module network (N2NMN)

End-to-end module network (N2NMN) [22] is a model for visual question answering. Given an image and an expression, N2NMN predicts a computational expression and a sequence of attentive module parameterizations. It uses these to assemble a concrete network architecture and then executes the assembled neural module network to output an answer for visual question answering. The LSTM model of N2NMN has two layers. For the first layer, the input vector length is 300 and the hidden dimension is 512. Thus, the $\boldsymbol{W}$ of the first LSTM layer is an 812-by-2048 matrix. For the second layer, the input vector length is 512 and the hidden dimension is 512. Thus, the $\boldsymbol{W}$ of the second LSTM layer is an 1024-by-2048 matrix. The matrix of the first layer is shown in Fig. 4e. The matrix of the second layer is shown in Fig. 4f.

3.6 Application6: one billion word language modeling

To comprehensively evaluate our approaches, we additionally pick the one billion word language modeling dataset [7], which is similar to ImageNet dataset [12] in computer vision. We use the BIG-LSTM model [26] as the baseline. The LSTM projection technique [41] was used in BIG-LSTM model. Let us denote p is projection length. In this situation, both the input vector (l-by-1) and hidden state vector (d-by-1) are projected onto lower-dimensional vectors (p-by-1) before entering the LSTM cell. The dimension of the LSTM cell matrix is transformed from (($l + d$)-by-4d) to (2p-by-4d). For BIG-LSTM, p is 1024 and d is 8192. Thus, we need to handle a 2K-by-32K dense matrix in LSTM inference.

4 Fast LSTM inference

4.1 Terms

Let us use SVD here to explain the terms. We use our system to convert the LSTM cell matrix $\boldsymbol{W}$ into two smaller matrices $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$. We define $|W|$ as the number of parameters in W. Then, the **compression rate** is $|W|/(|W_1| + |W_2|)$. For a given input vector, the original solution is $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$. The SVD approximate solution is $\tilde{\boldsymbol{y}} = \boldsymbol{W_1}\boldsymbol{W_2}\boldsymbol{x}$. We use $E = (||\boldsymbol{y} - \tilde{\boldsymbol{y}}||_2)/||\boldsymbol{y}||_2$ to denote the **compression error rate**.

4.2 LSTM inference bottleneck

GPU is a popular hardware option for deep learning training. Likewise, CPU is a popular hardware option for inference on cloud systems. Since we focus on reducing the inference latency of LSTM applications, we examine three widely used CPUs on today's cloud systems and further note that their ample cache sizes make them well suited to our approach. It is worth noting that our system also works for GPU-based or TPU-based hardware. The particulars of memory sizes are shown in Table 1, and in Table 2 we observe that most of the cell matrices of regular LSTM models can fit the caches of these devices. The key operation of LSTM is a dense matrix vector multiplication, which has a regular memory access pattern. Altogether this implies that memory access is not the major bottleneck if the model can fit into cache. Our experimental results confirmed this assumption (i.e., floating-point operations dominate). Thus, to reduce the latency of LSTM inference, our goal is to minimize the number of floating-point operations (flops). In this situation, the application is computation-bound rather than memory-bound. We design and implement our system based on two fundamental factorization approaches: SVD and CPD.

4.3 Dynamic selection between SVD and CPD

The kernel operation of LSTM is a dense matrix vector multiplication $\boldsymbol{Wx}$. We assume the dimension of $\boldsymbol{W}$ and $\boldsymbol{x}$ is m-by-n and n-by-1, respectively. We define two integers n_1 and n_2 where $n_1 \times n_2 = n$. The number of parameters in model $\boldsymbol{W}$ is mn. Suppose we want a rank R, the number of parameters is reduced to $R(m+n)$ by SVD and $R(m+n_1+n_2)$ by CPD. Typically, n is larger than (n_1+n_2). For example, m is 1300 and n is 4000. If rank is 64, SVD can reduce the number of parameters from five million to 0.3 million. If we set $n_1 = 20$ and $n_2 = 20$, CPD can reduce the number of parameters to 0.1 million.

The number of flops in computing $\boldsymbol{y} = \boldsymbol{Wx}$ is $\Theta(mn)$. As mentioned in Sect. 2.3, the SVD factorizes $\boldsymbol{W}$ into $\boldsymbol{W_1}$ (m-by-R) and $\boldsymbol{W_2}$ (R-by-n). Computing $\tilde{\boldsymbol{y}} = \boldsymbol{W_1 W_2 x}$ needs $\Theta(R(m+n))$ operations. For CPD (Sect. 2.4), we get a total of R sets. In each set, we need to finish $\Theta(m+n_1n_2)$ operations and get a n_1-by-n_2 matrix. To sum up these R different n_1-by-n_2 matrices, we need to finish $\Theta(Rn_1n_2)$ operations. Thus, the total number of floating-point operations by CPD is $\Theta(R(m+n_1n_2)) + \Theta(Rn_1n_2) = \Theta(R(m+2n))$. Table 3 gives the summary of comparison between CPD and SVD. Picking n_1 and n_2 for CPD will have an influence on both the number of parameters and the approximate error rate. Typically, higher compression rate will lead to higher error rate. In our system, we first make sure CPD is able to meet the compression error rate requirement (e.g., 5%). Then, we search the best n_1 and n_2 to achieve highest compression rate. We use CMN application (introduced in Sect. 3) as an example to illustrate the key idea. There are three situations:

- **We set n_1 or n_2 to 1. CPD and SVD use the same rank** CPD will get the same number of parameters and the same application error rate with SVD. On the other hand, CPD is slower than SVD because CPD needs to conduct more floating-point operations. There is no benefit in using CPD. Our system picks SVD in this situation.
- **We set neither n_1 nor n_2 to 1, CPD and SVD use the same rank** 5% error rate is our target. For rank = 512, SVD is qualified (No. 3 in Table 3). However, No. 9 and No. 10 are also qualified and they have a higher compression rate. In this situation, CPD beats SVD in reducing the number of parameters. For this application in this situation, our system picks CPD.

Table 1 Device memory on current cloud systems and mobile systems for LSTM inference

Device	Generation (nm)	Year	L1 cache	L2 cache (MB)	L3 cache (MB)	Memory (GB)
Haswell CPU (Intel Xeon E5-2698 V3)	22	2013	512 KB	4	40	768
Broadwell CPU (Intel Xeon E5-4660 v4)	14	2014	1 MB	4	35	768
Skylake CPU (Intel Xeon Platinum 8168)	14	2015	768 KB	24	33	768
iPhone 7+ (Apple A10)	16	2016	64 KB	3	4	3
iPhone X (Apple A11)	10	2017	64 KB	8	—	3

Table 2 Does the model Matrix W (m-by-n) fit device last-level cache of CPU? most of the model matrices can fit the last-level cache of cloud sever. Only the tiny model matrices can fit the last-level cache of the mobile devices like cell phone

Model matrix W	Haswell (45 MB)	Boardwell (40 MB)	Skylake (58 MB)	iPhone 7+ (7 MB)	iPhone X (8 MB)
CMN: 1300-by-4000 (20 MB)	Yes	Yes	Yes	No	No
MNIST: 256-by-512 (512 KB)	Yes	Yes	Yes	No	No
IMDB: 178-by-512 (356 KB)	Yes	Yes	Yes	No	No
PTB1: 1300-by-2600 (12.9 MB)	Yes	Yes	Yes	No	No
PTB2: 1300-by-2600 (12.9 MB)	Yes	Yes	Yes	No	No
N2NMN1: 812-by-2048 (6 MB)	Yes	Yes	Yes	Yes	Yes
N2NMN2: 1024-by-2048 (8 MB)	Yes	Yes	Yes	No	Yes
BIG-LSTM: 2k-by-32k (256 MB)	No	No	No	No	No

- **We set neither n_1 nor n_2 to 1. CPD and SVD use different ranks** For rank = 256, SVD is qualified (No. 2 in Table 3). However, No. 10 is also qualified and it has a higher compression rate. In this situation, CPD beats SVD in reducing the number of parameters. Our system picks CPD at run time.

Although CPD can do a better job in reducing the number of parameters, our focus is to reduce the computational complexity if the model can fit the platform's cache. For the mobile device like cell phone, both the cache size and memory size are much smaller than the cloud server (Table 1). In this situation, the LSTM model often is not able to fit the device cache (Table 2). If the model can not fit in the cache, our system does a quantitative evaluation to compare the overhead of the data movement and and the overhead of flops (Fig. 1). If the data movement overhead is higher, our system picks the CPD-based approach. Otherwise, the system picks SVD-based approach. This decision is made online because it depends on both the hardware and the application.

4.4 Direct factorization inference

To allow the users with well-trained LSTM models to finish the development in a short time, our system provides an direct factorization inference engine. In this section, we use SVD-based approach to illustrate the key idea. The CPD-based approach works in the same way (Fig. 1). For a well-trained input model, the first step of our system is to directly apply SVD in LSTM inference. This approach does not need to retrain the model. We use SVD to factorize the LSTM cell matrix $\boldsymbol{W}$ into $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$. For a given input vector, the original solution is $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$. The SVD approximate solution is $\tilde{\boldsymbol{y}} = \boldsymbol{W_1}\boldsymbol{W_2}\boldsymbol{x}$. As mentioned before, we use $E = (||\boldsymbol{y} - \tilde{\boldsymbol{y}}||_2)/||\boldsymbol{y}||_2$ to denote the compression error rate or error rate. In practice (based on the feedback from the users), we observe that the final test accuracy will not be influenced if E is less than 5%. From Fig. 5, we observe that direct SVD inference only works for CMN, N2NMN-1, and N2NMN-2. For these three applications, direct SVD inference can achieve an order of magnitude speedup with less than 5% compression error rate. However, for other model matrices, the compression error rates are extremely high. We sum up the results in Table 4. For CMN, N2NMN-1, and N2NMN-2, the singular values decrease very fast and there are many relatively tiny singular values in the model matrices (Fig. 6). For the model matrices of IMDB, MNIST, and PTB, the singular values decrease very slowly and all of them are larger than 0.1. Thus, we think the matrices of IMDB, MNIST, and PTB are not suitable for the direct SVD inference. To solve this problem for IMDB, MNIST, and PTB, our system needs to use the dynamic decomposition technique in the training process and train a suitable model with a lower rank.

4.5 Dynamic-aware model training for inference

In this section, we also use SVD-based approach to illustrate the key idea, the CPD-based approach works in the same way (Fig. 1). In practice, many users often do not have enough hardware memory to train the large LSTM models. In this situation, our system will automatically train the model from scratch. On the other hand, although some users have enough hardware memory, they do not have any trained model. Our system also needs to start from scratch for these users. For the users starting from scratch, our system first does a regular LSTM training and then does a SVD-aware training.

Table 3 Comparison between SVD and CPD for CMN model (1300-by-4000). In practice (based on the feedback from our customers), we observe that the final test accuracy will not be influenced if compression error rate is less than 5%

No	Case	#Parameters	Compression rate	#Flops	Speedup	Compression error rate (%)
1	SVD $R = 128$	678,400	7.7×	678,400	7.7×	14.1
2	SVD $R = 256$	1,356,800	3.8×	1,356,800	3.8×	2.27
3	SVD $R = 512$	2,713,600	1.9×	2,713,600	1.9×	0.01
4	CPD $R = 128$ ($n_1 = 4000, n_2 = 1$)	678,528	7.7×	1,190,400	4.4×	14.1
5	CPD $R = 256$ ($n_1 = 4000, n_2 = 1$)	1,357,056	3.8×	2,380,800	2.2×	2.27
6	CPD $R = 256$ ($n_1 = 2000, n_2 = 2$)	845,312	6.1×	2,380,800	2.2×	6.92
7	CPD $R = 256$ ($n_1 = 1000, n_2 = 4$)	589,824	8.8×	2,380,800	2.2×	11.3
8	CPD $R = 512$ ($n_1 = 4000, n_2 = 1$)	2,714,112	1.9×	4,761,600	1.1×	0.01
9	CPD $R = 512$ ($n_1 = 2000, n_2 = 2$)	1,690,624	3.1×	4,761,600	1.1×	1.02
10	CPD $R = 512$ ($n_1 = 1000, n_2 = 4$)	1,179,648	4.4×	4,761,600	1.1×	5.38

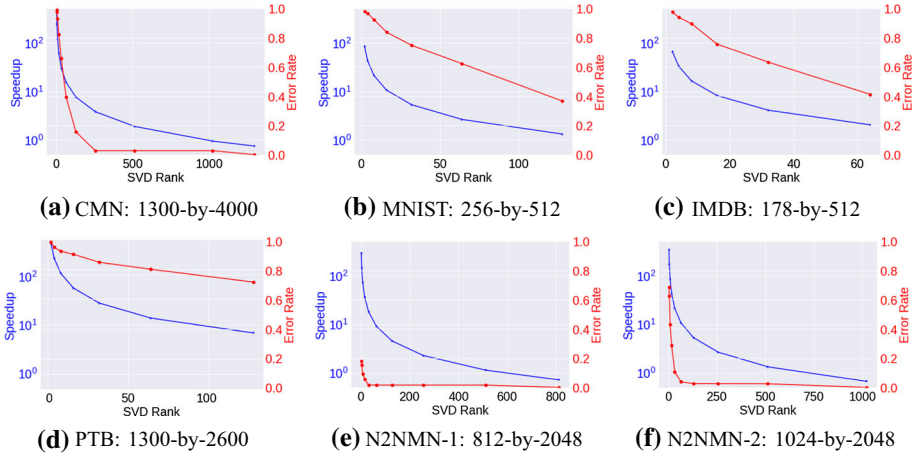


Fig. 5 Effect of direct SVD LSTM inference. The blue line represents the speedup corresponding to the blue horizontal axis (left). The red line represents the error rate corresponding to the red horizontal axis (right). CMN, N2NMN-1, and N2NMN-2 can achieve an order of magnitude speedup with less than 5% error rate. For other model matrices, direct SVD does not work well

If the users already have a well-trained regular model and have a system with enough memory, our system directly does the SVD-aware training based on the regular model. The output of our system is a SVD-aware model. Our system uses the SVD-aware model in inference process. As mentioned in Sect. 2.1, l is the input vector length and d is the LSTM cell hidden dimension. The LSTM cell matrix $\boldsymbol{W}$ is a $4d$-by-$(l+d)$ matrix. Our system has the following three main steps:

- Step 1: Our system gets the model $\boldsymbol{W}$ by regular LSTM training based on the input data. The regular LSTM training runs the optimization solver iteratively until the algorithm is converged. For most of the applications, our system picks momentum SGD [5] or Adam [28] as the optimization solver. It usually takes thousands of iterations. Each iteration includes the forward propagation and backward propagation. This step will be skipped if the users start from scratch.
- Step 2: Our system factorizes the trained model matrix W by decomposition approach to the smaller model matrices. In SVD, they are $\boldsymbol{W_1}$ ($4l$-by-R) and $\boldsymbol{W_2}$ (R-by-$(l+d)$).
  - For example, in SVD we have $W = U\Sigma V^T = U\sqrt{\Sigma} \times \sqrt{\Sigma}V^T = W_1 W_2$
- Step 3: The system fine-tunes $\boldsymbol{W_1}$ and $\boldsymbol{W_2}$ in SVD-aware LSTM training based on the input data. The system essentially transfers a k-layer-wide neural networks into a $2k$-layer narrow neural network. This process also runs the optimization solver iteratively until the algorithm is converged. It usually takes more iterations than regular LSTM training.

The implementation details of SVD-aware LSTM training are shown in Sect. 4.6. It is worth noting that the regular LSTM training and the SVD-aware LSTM training are two different optimization problems. They may need different hyper-parameters or even different optimization solvers. Our system will automatically tune these hyper-parameters. The users do not need to tune them manually. From Fig. 7, we observe that combining regular training and SVD-aware training together can achieve 21× speedup in inference without losing accuracy (for MNIST dataset).

Table 4 Does direct decomposition inference meet the requirement? in practice (based on the feedback from our users), we observe that the final test accuracy will not be influenced if compression error rate is less than 5%

Model	Compression error rate (%)	Speedup	Singular values range	Satisfied?
N2NMN-1: 812-by-2K	1.6	18×	10^{-13}–10^{0}	Yes
N2NMN-2: 1K-by-2K	4.3	11×	10^{-15}–10^{0}	Yes
CMN: 1300-by-4000	2.3	4×	10^{-8}–10^{1}	Yes
MNIST: 256-by-512	36.6	1.3×	10^{-1}–10^{1}	No
IMDB: 178-by-512	41.2	2×	10^{-1}–10^{1}	No
PTB-1: 1300-by-2600	39.3	1.7×	10^{0}–10^{2}	No
PTB-2: 1300-by-2600	38.0	1.7×	10^{0}–10^{2}	No

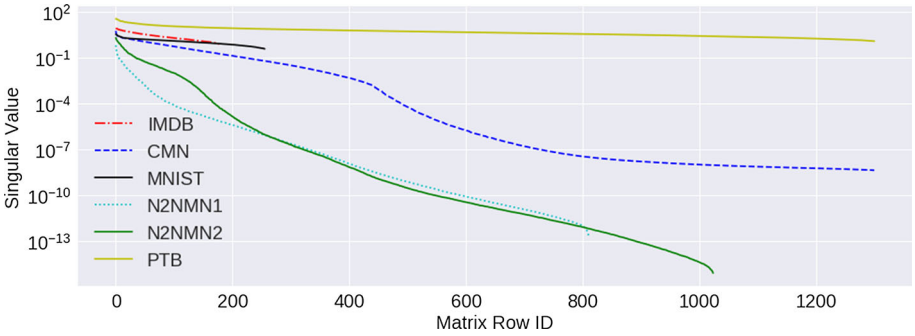
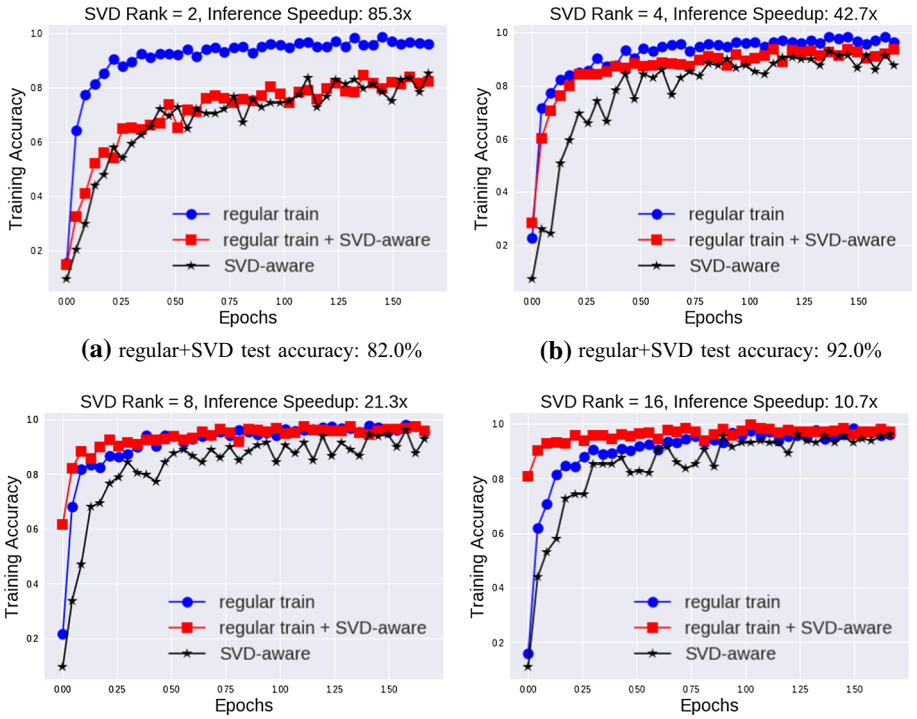


Fig. 6 For CMN, N2NMN-1, and N2NMN-2, singular values decrease very quickly and there are many tiny singular values in these model matrices. For other model matrices, the singular values decrease slowly and all of them are larger than 0.1

In practice, the users can pass a well-trained model to our system or start from scratch to use our system. If we skip steps 1 and 2 and only run step 3 (for users who do not have enough hardware memory to finish regular LSTM training), our system can also get a reasonable accuracy. From Fig. 7 and Table 5, we observe that only running SVD-aware training can achieve 10.7× speedup but with a lower testing accuracy (95.9% vs. 94.5%). On the other hand, since our focus is on LSTM inference time rather than training time, we often train a good model and use it for weeks or even months on different platforms. Thus, our system can train longer and generate a much better model. Figure 8 shows that our system can achieve the same or even a higher test accuracy by just training longer. In this way, we can achieve 42.7× speedup in inference without losing accuracy.

By using this approach, we also achieved good speedup for the IMDB and PTB models. For the IMDB dataset, the model trained by regular LSTM achieves 85.2% test accuracy. Our system achieves 86.0% accuracy and 17× speedup (SVD rank = 8 for 178-by-512 matrix). For PTB dataset, the model trained by regular LSTM achieves 28% test accuracy. Our system achieves 28% accuracy and 7× speedup (SVD rank = 128 for the 1300-by-2600 matrix). The Big-LSTM model uses perplexity rather than accuracy for the correctness metric. Lower perplexity means higher accuracy. The perplexity of the baseline is 35.1 after 10 days of training on eight NVIDIA Tesla M40 GPUs. Since the model cannot fit cache and its data



(a) regular+SVD test accuracy: 82.0%

(b) regular+SVD test accuracy: 92.0%

(c) regular+SVD test accuracy: 95.9%

(d) regular+SVD test accuracy: 96.4%

Fig. 7 Baseline's test accuracy is 95.9% for MNIST dataset. We observe that combining regular training and SVD-aware training together can achieve 21× speedup in inference without losing accuracy (for MNIST dataset)

Table 5 Testing accuracy and inference speedup for MNIST (256-by-512 matrix). Regular train and SVD-aware run the same number of epochs

Rank	2	4	8	16
Regular train	0.959	0.959	0.959	0.959
Regular train + SVD-aware	0.820	0.920	0.959	0.964
SVD-aware	0.791	0.908	0.919	0.945
Inference speedup	85.3×	42.7×	21.3×	10.7×

movement overhead is higher than the flops overhead, our system picks CPD-aware training for it. If we set CPD rank as 512 for the 2k-by-32k matrix, our system achieves 35.9 perplexity and 4× speedup in inference. The details are summed up in Table 6.

4.6 Implementation details

The industry frameworks such as TensorFlow [1] and PyTorch [39] can directly use our implementation. We provide Python interface for them. To achieve a higher performance, we use C/C++, OpenMP threading, and SIMD instructions in low-level implementation. For instance, the users only need to use our LSTM-Factorize class to replace the framework's LSTM class (e.g., tf.nn.rnn_cell.BasicLSTMCell in TensorFlow framework). The users can

Fig. 8 By training longer, SVD-aware achieves 42.7× speedup in inference. This function is useful for users who do not have enough hardware memory to finish regular LSTM training. The test accuracy (higher is better) for MNIST dataset: regular (95.9%), regular + SVD-aware (96.3%), SVD-aware (96.4%)

Table 6 Test accuracy and inference speedup by using SVD/CPD-aware training

Application	MNIST	IMDB	PTB	One billion
Optimizer	Adam	Adam	SGD	Adagrad
Learning rate	0.001	0.001	0.3	0.2
Baseline accuracy	95.9%	85.2%	28%	perplexity = 35.1
Our accuracy	96.4%	86.0%	28%	perplexity = 35.9
Inference speedup	42×	17×	7×	4×

also directly use our interface to conduct the training and inference. In this way, the users will not need to use any framework.

4.7 Experimental platforms

For training the small and medium models, we use a server powered by eight NVIDIA Tesla M40 GPUs. For training the extremely large model, we use TACC's Stampede2 supercomputer, NERSC's Cori2 supercomputer, and CSCS's Piz Daint supercomputer. For Stampede2, we use the CPU cluster. Each Stampede2 node has 48 Intel Xeon Platinum 8160 (Skylake) cores on two sockets (24 cores/socket). The nominal frequency is 2.1 GHz (1.4-3.7 GHz depending on instruction set and number of active cores). Hyper-threading is enabled: There are two hardware threads per core, for a total of $48 \times 2 = 96$ hardware threads per node. For NERSC Cori2, each node contains an Intel Xeon Phi Processor 7250 @ 1.40 GHz. 68 cores per node with support for 4 hardware threads each (272 threads total). For the LSTM inference, we use the server on Google cloud and Microsoft Azure cloud. Our inference hardware information is available in Table 1.

5 Distributed training with dynamic decomposition

We also want to improve the training efficiency in a deep learning system. To our knowledge, training is a non-trivial overhead in the real-world applications with millions of training

samples. It can take days or even weeks to train a deep learning model. State-of-the-art approaches currently are using distributed systems or supercomputers [61] to train the huge deep learning models.

5.1 Implementation

HPC and supercomputing techniques are becoming increasingly popular among Internet tech giants. Amazon AWS provides an elastic and scalable cloud infrastructure to run HPC applications.[1] Google released its first 100-petaFlop supercomputer, named the TPU Pod.[2] Facebook made a submission on the Top500 supercomputer list.[3] Microsoft is heavily investing in supercomputing techniques.[4] The reason is that a large change in the computing world started in the mid-2000s: not only are the fastest computers parallel, but nearly all computers are becoming parallel, because the physics of semiconductor manufacturing will no longer let conventional sequential processors get faster year after year, as they have for so long (roughly doubling in speed every 18 months for many years). So all programs that need to run faster will have to become parallel programs. Thus, modern supercomputers are made up of thousands of chips or tens of thousands of processors, which require an extremely high parallelism to reach their peak performance. However, where does the parallelism come from for deep learning?

There are two kinds of parallelism for deep neural network training: data parallelism and model parallelism [9].

For data parallelism (Fig. 9), developers can partition the dataset and process each data subset independently. We assume the system has n samples and P nodes. In this toy example, n is 6 and P is 3. We need to finish five steps during each iteration. At the first step, we evenly and randomly partition the data to all the nodes. At the second step, each machine node will finish a local forward propagation and a local backward propagation on its own local data. At the third step, each machine node will get a local gradient. At the fourth step, the system will do an all-reduce operation to get the global gradients by averaging all the local gradients. The system will send a copy of the global gradients to all the nodes. At the fifth step, each node will use the global gradients (it own copy) to update its own local weights.

For model parallelism, developers can parallelize across different layers or within each layer. Due to the data dependency between different layers in forward propagation and backward propagation, developers cannot efficiently parallelize across different layers except for pipelining [24]. Thus, developers usually need to parallelize within each layer to implement model parallelism. In this way, a wider neural network provides higher parallelism. However, given the same number of parameters, a deep model can achieve better results than a wide model [14]. Thus, modern neural networks are deep rather than wide. For example, a typical layer of BERT (state-of-the-art NLP model) is a 1024-by-1024 matrix. The widest layer of BERT is a 1024-by-4096 matrix. If we disable data parallelism (i.e., set batch size as one), we cannot make full use of the computational power of even one GPU or CPU chip to accelerate a $1024 \times 1024 \times 1024$ matrix multiply.

For applications with an extremely wide model or a large single sample, model parallelism can work with data parallelism (e.g., [45]). Assume we have P nodes in this situation, and we partition these P nodes into G groups (e.g., G=256, P=1024). We use model-parallelism

[1] https://d1.awsstatic.com/HPC2019/Amazon-HyperionTechSpotlight-190329.FINAL-FINAL.pdf.

[2] https://techcrunch.com/2019/05/07/googles-newest-cloud-tpu-pods-feature-over-1000-tpus.

[3] https://www.top500.org/site/50701.

[4] https://thenextweb.com/artificial-intelligence/2019/07/23/openai-microsoft-azure-ai/.

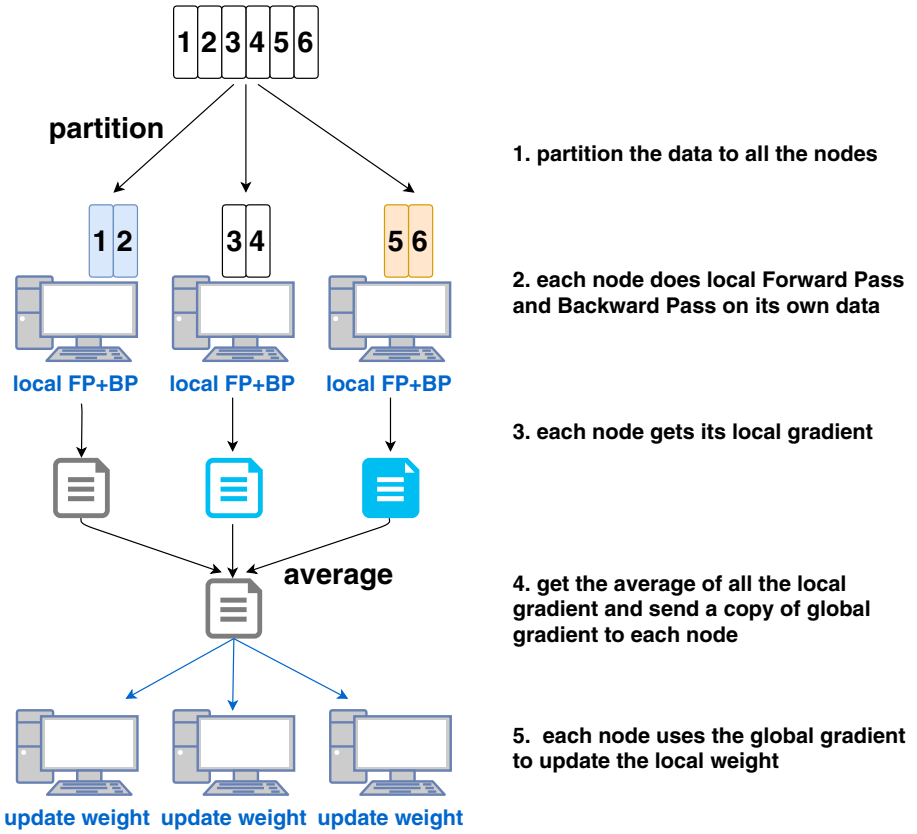


Fig. 9 Synchronous implementation for distributed deep learning. The communication between different nodes in each iteration is an all-reduce operation. The system computes the global gradients by averaging all the local gradients on each node. Then, each node will get a copy of the global gradients and use it to update the local weights

within each group and data-parallelism across different groups. Data parallelism dominates in most of our applications (i.e., $G \gg P/G$). Thus, we study the data-parallel deep learning method, which includes synchronous and asynchronous approaches. Recent studies [2,59] show that the synchronous approach can be much faster and more stable (e.g., easy to reproduce the results and debug) than the asynchronous approach. The recent successes [60,61] of large-scale distributed deep learning are also based on a synchronous data-parallel approach. Thus, we design and implement the data-parallel approach with dynamic decomposition.

As most algorithms, the deep learning workload includes two parts: communication and computation. First, we evaluate the communication time, which is referred to as t_{comm}. Here, the communication time is actually the total time spent in the network along the critical path. We do not study the communication within a single node. The communication time includes two parts:

α or network latency is the time to send an empty message (zero bytes). For TCP system based on gigabit Ethernet, the latency is around 50μs per message, which is extremely inefficient. The reason is that the message needs to travel a long way in this situation: (1) from the CPU to the north-bridge, (2) then to the PCI controller, (3) then to the network card,

(4) then to the network, (5) then to the switch, (6) then to the other network card, (7) then to the CPU interrupt controller through the OS, and (8) finally to the application. The more efficient and expensive networks such as Infiniband or Myrinet have the latency $\alpha \approx 2\mu$s per message. Starting a new TCP connection may cost hundreds of milliseconds, especially on the condition that we need to resolve the DNS name. People usually use α or alpha to denote the network latency.

β or the inverse of network bandwidth is the time to send each byte of the message (s/byte). For 500 MB/s gigabit ethernet (4000 Mb/s), it is 2 ns per byte. The bandwidth of an $4\times$ Infiniband is 1000 MB/s, whose β is 1ns/byte. People usually use β or beta to denote the inverse of network bandwidth.

As mentioned before, we use Sync SGD approach for large-scale DL training. The key communication part in Sync SGD is an all-reduce operation. There are three widely used algorithms for all-reduce: butterfly, tree, and ring [49]. The tree algorithm is optimized for latency. The ring algorithm is optimized for bandwidth. The butterfly algorithm is optimized for both of them. The butterfly algorithm is implemented with a recursive halving reduce-scatter followed by a recursive doubling all-gather. The complexities of these three algorithms are described in Eqs. (8), (9), and (10) where M_s is the message size. In our experiments, we found butterfly algorithm performs better than ring and tree algorithms for our deep learning workloads on several supercomputers (e.g., NERSC Cori2, Piz Daint, TACC Stampede2). We choose butterfly algorithm in our implementation.

$$t_{\text{tree}} = 2(\alpha \log P + \beta M_s \log P) \tag{8}$$

$$t_{\text{ring}} = 2\left(\alpha P + \frac{P-1}{P}\beta M_s\right) \tag{9}$$

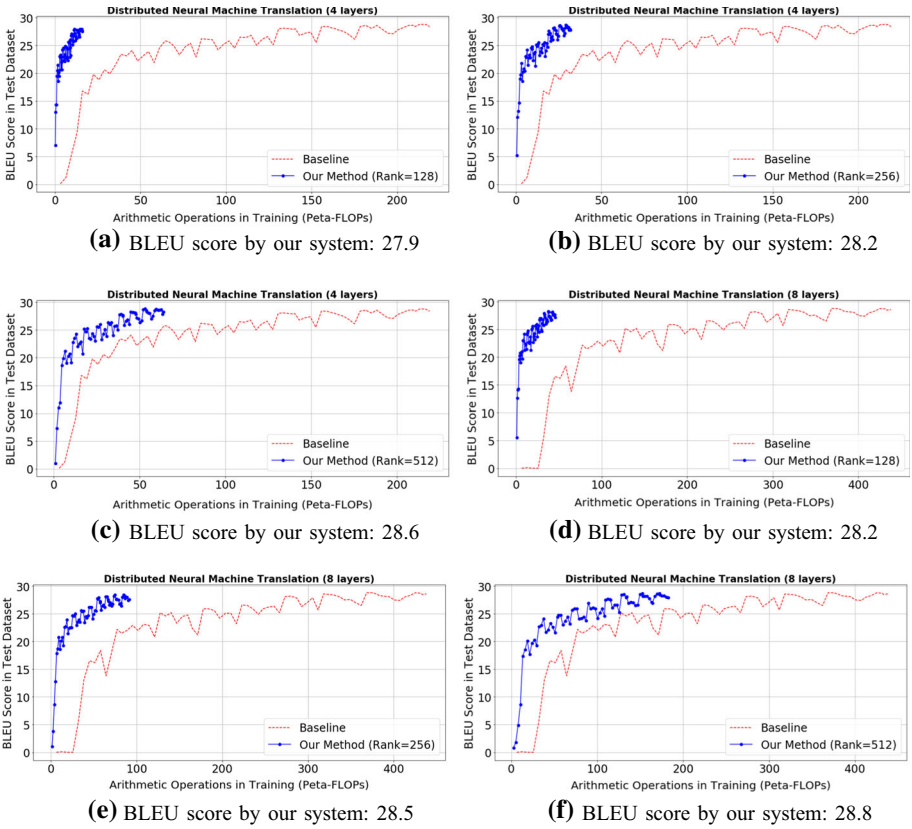
$$t_{\text{butterfly}} = 2\left(\alpha \log P + \frac{P-1}{P}\beta M_s\right) \tag{10}$$

5.2 GNMT: Google's neural machine translation

To evaluate the efficiency of our distributed system with dynamic decomposition, we use GNMT (Google's Neural Machine Translation) [33,56]. GNMT is a large-scale deep learning model that being used in a wide range of machine translation applications. The shared embeddings are used in the encoder layers and decoder layers. In our experiments, we used both the four-layer GNMT and eight-layer GNMT. Let us use four-layer GNMT as an example to illustrate the structure of the model. There are four LSTM layers in the encoder part, which has a hidden dimension of 1K. For these four layers, only the first layer is bidirectional, and the rest are undirectional. The residual connections start from the third layer. There are four unidirectional LSTM layers with a hidden dimension of 1K and a fully connected classifier in the decoder part. The residual connections start from the third layer. The normalized Bahdanau attention (gnmt_v2 attention mechanism) was used in our model. State-of-the-art implementations use WMT16 English–German translation dataset to train the GNMT. We use the same dataset in our experiments. As the baseline, we use BLEU score on newstest2014 dataset as the quality metric (higher is better). We use sacrebleu package to collect the BLEU score. In our experiments, the baseline can roughly get a BLEU score of 28.

5.3 Convergence speed

A recent study [27] shows that the distributed optimization is a sharp minimal problem. The reason is that we have to make the batch size very large on a distributed system. Even though



(a) BLEU score by our system: 27.9 **(b)** BLEU score by our system: 28.2

(c) BLEU score by our system: 28.6 **(d)** BLEU score by our system: 28.2

(e) BLEU score by our system: 28.5 **(f)** BLEU score by our system: 28.8

Fig. 10 From these figures, we can observe that our approach can achieve the same BLEU score (higher is better) as the baseline with much fewer arithmetic operations

we can train the model well, we can hardly get any good results in the testing dataset, which means it is hard to generalize the model trained by distributed systems. After combining the distributed optimization together with the dynamic decomposition technique, we need to verify the convergence speed. For a fair comparison, we fix the number of iterations and batch size (which means the number of epochs is fixed). In this way, our system will be much faster than the baseline because we reduced the number of arithmetic operations by the dynamic decomposition technique. We finished fewer number of operations to process each image. Our experimental results in Fig. 10 show that our approach can achieve the same BLEU score (higher is better) as the baseline with much fewer arithmetic operations. Thus, our distributed system with dynamic decomposition can achieve a faster convergence rate than the baseline.

5.4 Communication overhead

After implementing our system, we found that in some situations the communication (over the network) cost is much higher than the computation cost. Here, we collect some data from real-world hardware:

Table 7 Communication cost under α-β Model

Network	α (latency)	β (1/bandwidth)
Mellanox 56 Gb/s FDR IB	0.7×10^{-6} s	0.2×10^{-9} s
Intel 40 Gb/s QDR IB	1.2×10^{-6} s	0.3×10^{-9} s
Intel 10 GbE NetEffect NE020	7.2×10^{-6} s	0.9×10^{-9} s

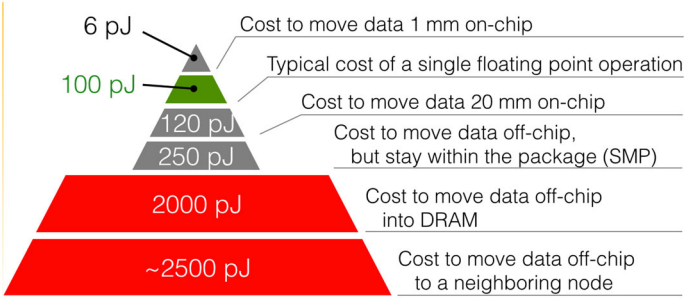


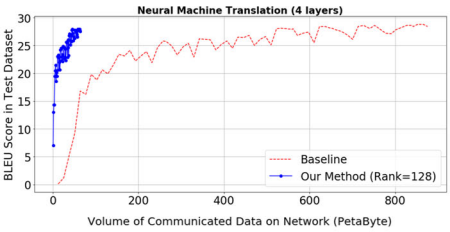
Fig. 11 From this figure, we can see the communication operations cost much more energies than computation operations

- Time-per-flop (γ) $\ll$ 1/ bandwidth (β) $\ll$ latency (α)
  - $\gamma = 0.9 \times 10^{-13}$s for NVIDIA P100 GPUs
  - α and β are in Table 7

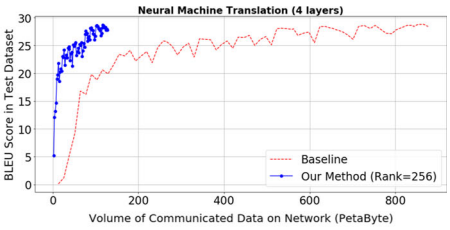
From the data, we can see that communicating one word will be much slower than finishing one floating-point operation. The situation will become worse for the large models with billions of parameters, which can be a significant overhead for communication. On the other hand, communication (data movement) also costs much more energies than computation (Fig. 11). Energy consumption is a big concern for DL, especially the search and tuning in AutoML. The estimated CO_2 emissions per person of one airline trip between New York and San Francisco is 1,984 lbs. However, the estimated CO_2 emissions of tuning a state-of-the-art NLP model is 626,155 lbs [43]. This motivates us to minimize the communication overhead to reduce the energy consumption. Figure 12 shows that our approach can achieve the same BLEU score (higher is better) as the baseline with much less data transferred over the network, which will also significantly reduce the energy cost and accelerate the training.

5.5 Discussion

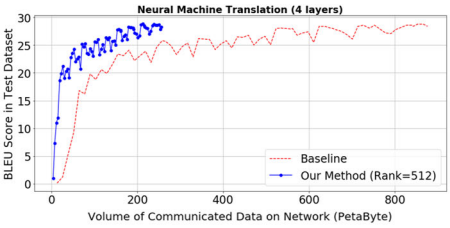
We evaluate our method and implementation on both regular cloud systems and mobile devices. Mobile devices are an important workload. There is a concern in current deep learning systems. We need to collect the data from the users and train our model on the cloud. A user can be a person. For a person, something like the photo in the cell phone is the data. Because data are confidential, some users probably do not want to share the data in the future. Federated learning is a new approach that allows the users to train their own model without sharing the data. So we think in the future many users will train the deep learning model on mobile devices.



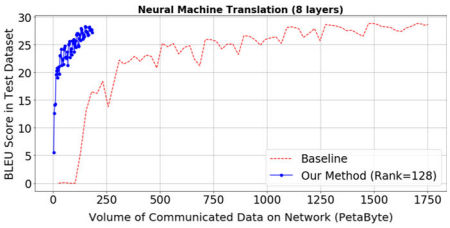
12.1 BLEU score by our system: 27.9



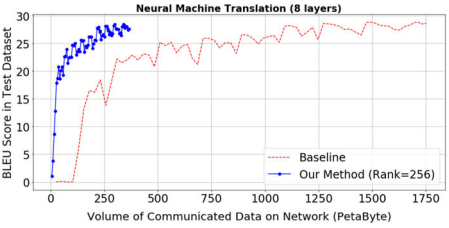
12.2 BLEU score by our system: 28.2



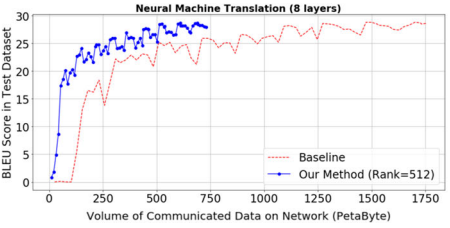
12.3 BLEU score by our system: 28.6



12.4 BLEU score by our system: 28.2



12.5 BLEU score by our system: 28.5



12.6 BLEU score by our system: 28.8

Fig. 12 From these figures, we can observe that our approach can achieve the same BLEU score (higher is better) as the baseline with much less data transferred over the network, which will also significantly reduce the energy cost

6 Conclusions and future work

Inference speed for LSTMs is a critical factor in many applications such as movie recommendation or online advertising. In our study, we found exploring the trade-off between flops and parameters is an important issue for low-latency LSTM inference. In this paper, we built a system based on dynamic low-rank techniques to accelerate LSTM inference. The users with or without a trained model can apply our system on any platform (with or without enough hardware memory). Our system works on all types of the hardware chips (e.g., CPU, GPU, or TPU). We demonstrated that our system can achieve an average of 15× speedup for six real-world applications without losing accuracy. We also design and implement a distributed optimization system with dynamic decomposition, which can significantly reduce the energy cost and accelerate the training process. We wish to continue to accelerate LSTM inference, and there are a number of other interesting directions to explore such as quantization [58], parallel timesteps [31], wide LSTM [20], and Sparsified SGD [32,65]. Due to limited space in this paper, we defer investigation of these to future studies.

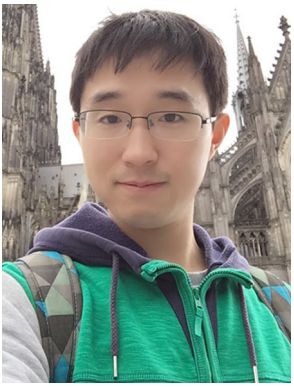
Acknowledgements We thank CSCS for granting us access to Piz Daint resources. We thank Ronghang Hu at UC Berkeley for useful discussions.

References

1. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI), vol 16. pp 265–283
2. Amodei D, Ananthanarayanan S, Anubhai R, Bai J, Battenberg E, Case C, Casper J, Catanzaro B, Cheng Q, Chen G, et al (2016) Deep speech 2: end-to-end speech recognition in english and mandarin. In: International conference on machine learning, pp 173–182
3. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw 5(2):157–166
4. Benkrid K, Belkacemi S (2002) Design and implementation of a 2d convolution core for video applications on fpgas. In: Third international workshop on digital and computational video, 2002. DCV 2002. proceedings. IEEE, pp 85–92
5. Bottou L (2012) Stochastic gradient descent tricks. In: Montavon G, Orr G, Müller K-R (eds) Neural networks: tricks of the trade. Springer, New York, pp 421–436
6. Cardells-Tormo F, Molinet P-L (2005) Area-efficient 2-d shift-variant convolvers for fpga-based digital image processing. In: IEEE workshop on signal processing systems design and implementation. IEEE, pp 209–213
7. Chelba C, Mikolov T, Schuster M, Ge Q, Brants T, Koehn P, Robinson T (2013) One billion word benchmark for measuring progress in statistical language modeling. arXiv:1312.3005
8. Cloutier J, Cosatto E, Pigeon S, Boyer FR, Simard PY (1996) Vip: An fpga-based processor for image processing and neural networks. In: Proceedings of fifth international conference on microelectronics for neural networks. IEEE, pp 330–336
9. Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, Ranzato M, Senior A, Tucker P, Yang K, et al (2012) Large scale distributed deep networks. In: Advances in neural information processing systems, pp 1223–1231
10. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391
11. Demmel JW (1997) Applied numerical linear algebra, vol 56. Siam, USA
12. Deng J, Dong W, Socher R, Li L-J, Li K, Fei-Fei L (2009) Imagenet: a large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. IEEE, pp 248–255
13. Denton EL, Zaremba W, Bruna J, LeCun Y, Fergus R (2014) Exploiting linear structure within convolutional networks for efficient evaluation. In: Advances in neural information processing systems, pp 1269–1277
14. Eldan R, Shamir O (2016) The power of depth for feedforward neural networks. In: Conference on learning theory, pp 907–940
15. Gironés RG, Palero RC, Boluda JC, Cortés AS (2005) Fpga implementation of a pipelined on-line backpropagation. J VLSI Signal Process Syst Signal Image Video Technol 40(2):189–213
16. Grave E, Joulin A, Cissé M, Grangier D, Jégou H (2016) Efficient softmax approximation for gpus. arXiv:1609.04309
17. Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ (2016) Eie: efficient inference engine on compressed deep neural network. In: 2016 ACM/IEEE 43rd annual international symposium on, computer architecture (ISCA). IEEE, pp 243–254
18. Han S, Mao H, Dally WJ (2015) Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv:1510.00149
19. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778
20. He Z, Gao S, Xiao L, Liu D, He H, Barber D (2017) Wider and deeper, cheaper and faster: tensorized lstms for sequence learning. In: Advances in neural information processing systems, pp 1–11
21. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
22. Hu R, Andreas J, Rohrbach M, Darrell T, Saenko K Learning to reason: end-to-end module networks for visual question answering
23. Hu R, Rohrbach M, Andreas J, Darrell T, Saenko K (2017) Modeling relationships in referential expressions with compositional modular networks. In: 2017 IEEE conference on computer vision and pattern recognition (CVPR). IEEE, pp 4418–4427
24. Huang Y, Cheng Y, Bapna A, Firat O, Chen D, Chen M, Lee H, Ngiam J, Le QV, Wu Y, et al (2019) Gpipe: Efficient training of giant neural networks using pipeline parallelism. In: Advances in neural information processing systems, pp 103–112

25. Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, et al (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture, pp 1–12
26. Jozefowicz R, Vinyals O, Schuster M, Shazeer N, Wu Y (2016) Exploring the limits of language modeling. arXiv:1602.02410
27. Keskar NS, Mudigere D, Nocedal J, Smelyanskiy M, Tang PTP (2016) On large-batch training for deep learning: Generalization gap and sharp minima. arXiv:1609.04836
28. Kingma DP, Ba J (2014) Adam: a method for stochastic optimization. arXiv:1412.6980
29. Kuchaiev O, Ginsburg B (2017) Factorization tricks for lstm networks. arXiv:1703.10722
30. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324
31. Lei T, Zhang Y (2017) Training rnns as fast as cnns. arXiv:1709.02755
32. Lin Y, Han S, Mao H, Wang Y, Dally WJ (2017) Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv:1712.01887
33. Luong M, Brevdo E, Zhao R (2017) Neural machine translation (seq2seq) tutorial. https://github.com/tensorflow/nmt
34. Maas AL, Daly RE, Pham PT, Huang D, Ng AY, Potts C (2011) Learning word vectors for sentiment analysis. In: Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies, Portland, Oregon, USA association for computational linguistics, pp 142–150
35. Marcus MP, Marcinkiewicz MA, Santorini B (1993) Building a large annotated corpus of english: The penn treebank. Computational linguistics 19(2):313–330
36. Narang S, Undersander E, Diamos G (2017) Block-sparse recurrent neural networks. arXiv:1711.02782
37. Nichols KR, Moussa MA, Areibi SM (2002) Feasibility of floating-point arithmetic in fpga based artificial neural networks. In: In CAINE. Citeseer
38. Pascanu R, Mikolov T, Bengio Y (2013) On the difficulty of training recurrent neural networks. In: International conference on machine learning, pp 1310–1318
39. Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in pytorch
40. Sainath TN, Kingsbury B, Sindhwani V, Arisoy E, Ramabhadran B Low-rank matrix factorization for deep neural network training with high-dimensional output targets
41. Sak H, Senior A, Beaufays F (2014) Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Fifteenth annual conference of the international speech communication association
42. Savich AW, Moussa M, Areibi S (2007) The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. IEEE Trans Neural Netw 18(1):240–252
43. Schwartz MO (2020) Groundwater contamination associated with a potential nuclear waste repository at yucca mountain. USA Bull Eng Geol Environ 79(2):1125–1136
44. Shawahna A, Sait SM, El-Maleh A (2018) Fpga-based accelerators of deep learning networks for learning and classification: A review. IEEE Access 7:7823–7859
45. Shazeer N, Cheng Y, Parmar N, Tran D, Vaswani A, Koanantakool P, Hawkins P, Lee H, Hong M, Young C, et al (2018) Mesh-tensorflow: Deep learning for supercomputers. In: Advances in neural information processing systems, pp 10414–10423
46. Shim K, Lee M, Choi I, Boo Y, Sung W (2017) Svd-softmax: Fast softmax approximation on large vocabulary neural networks. In: Advances in neural information processing systems, pp 5469–5479
47. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556
48. Sivakumar SC, Robertson W, Phillips WJ (1999) Online stabilization of block-diagonal recurrent neural networks. IEEE Trans Neural Netw 10(1):167–175
49. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in mpich. The Intern J High Perform Comput Appl 19(1):49–66
50. Tjandra A, Sakti S, Nakamura S (2017) Compressing recurrent neural network with tensor train. In: 2017 international joint conference on, Neural networks (IJCNN). IEEE, pp 4451–4458
51. Tjandra A, Sakti S, Nakamura S (2018) Tensor decomposition for compressing recurrent neural network. In: 2018 international joint conference on neural networks (IJCNN), IEEE, pp 1–8
52. Tucker LR (1966) Some mathematical notes on three-mode factor analysis. Psychometrika 31(3):279–311
53. Wen W, He Y, Rajbhandari S, Wang W, Liu F, Hu B, Chen Y, Li H (2017) Learning intrinsic sparse structures within long short-term memory. arXiv:1709.05027
54. Wolf DF, Romero RA, Marques E (2001) Using embedded processors in hardware models of artificial neural networks. In: Simpósio Brasileiro de Automação Inteligente, vol 9. Brasil

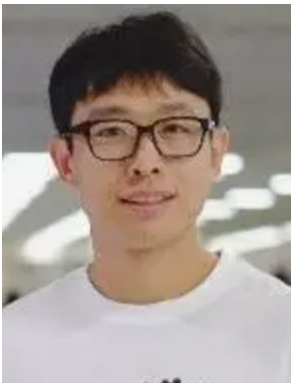
55. Wu C-Y, Ahmed A, Beutel A, Smola AJ, Jing H (2017) Recurrent recommender networks. In: Proceedings of the tenth ACM international conference on web search and data mining. ACM, pp 495–503
56. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, et al (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:1609.08144
57. Xiong W, Droppo J, Huang X, Seide F, Seltzer M, Stolcke A, Yu D, Zweig G (2016) Achieving human parity in conversational speech recognition. arXiv:1610.05256
58. Xu C, Yao J, Lin Z, Ou W, Cao Y, Wang Z, Zha H (2018) Alternating multi-bit quantization for recurrent neural networks. arXiv:1802.00150
59. You Y, Buluç A, Demmel J (2017) Scaling deep learning on gpu and knights landing clusters. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. ACM, p 9
60. You Y, Li J, Reddi S, Hseu J, Kumar S, Bhojanapalli S, Song X, Demmel J, Hsieh C-J (2019) Large batch optimization for deep learning: training bert in 76 minutes. arXiv:1904.00962
61. You Y, Zhang Z, Hsieh C-J, Demmel J, Keutzer K (2018) Imagenet training in minutes. In: Proceedings of the 47th international conference on parallel processing. ACM, p 1
62. Zhai S, Chang K-h, Zhang R, Zhang ZM (2016) Deepintent: Learning attentions for online advertising with recurrent neural networks. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 1295–1304
63. Zhang C, Sun G, Fang Z, Zhou P, Pan P, Cong J (2018) Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. IEEE Trans Comput-aided Des Integr Circuits and Syst 38(11):2072–2085
64. Zhang H, Xia M, Hu G (2007) A multiwindow partial buffering scheme for fpga-based 2-d convolvers. IEEE Trans Circuits Syst II: Express Br 54(2):200–204
65. Zhu M, Rhu M, Clemons J, Keckler SW, Xie Y (2016) Training long short-term memory with sparsified stochastic gradient descent



Yang You is a tenure-track assistant professor at National University of Singapore. He received his PhD in Computer Science from UC Berkeley. His advisor is Prof. James Demmel, who was the former chair of the Computer Science Division and EECS Department. Yang You's research interests include Parallel/Distributed Algorithms, High Performance Computing, and Machine Learning. The focus of his current research is scaling up deep neural networks training on distributed systems or supercomputers. In 2017, his team broke the world record of ImageNet training speed, which was covered by the technology media like NFS, ScienceDaily, Science NewsLine, and i-programmer. He is a winner of IPDPS 2015 Best Paper Award (0.8%), ICPP 2018 Best Paper Award (0.3%) and ACM/IEEE George Michael HPC Fellowship. Yang You is a Siebel Scholar and a winner of Lotfi A. Zadeh Prize.

Yuxiong He is a researcher in MSR Technology. Her research interests include resource management, algorithms, modeling, and performance evaluation of parallel and distributed systems. She received her Ph.D. in Computer Science from Singapore-MIT Alliance in 2008, working with Wen-Jing Hsu and Charles E. Leiserson. She has a BE degree in Computer Engineering from NTU.

Samyam Rajbhandari is a Research Software Development Engineer at Microsoft AI and Research.



Wenhan Wang is a Senior Engineering Manager at Microsoft.

Cho-Jui Hsieh is an assistant professor of Computer Science at UCLA since Fall 2018. Cho-Jui Hsieh was a Ph.D. student at UT Austin working with Prof. Inderjit Dhillon. He worked closely with Prof. Pradeep Ravikumar. He received his master degree in National Taiwan University under supervision of Prof. Chih-Jen Lin. Before joining UCLA, Cho-Jui Hsieh worked as an Assistant Professor at UC Davis Computer Science and Statistics for three years and was a visiting scholar in Google since summer 2018. Cho-Jui Hsieh is interested in developing new algorithms and optimization techniques for large-scale machine learning problems. Currently, he is working on developing new machine learning models as well as improving the model size, training speed, prediction speed, and robustness of popular (deep learning) models.

Kurt Keutzer is a professor in the EECS department of UC Berkeley. Kurt received his Ph.D. degree in Computer Science from Indiana University in 1984 and then joined the research division of AT&T Bell Laboratories. In 1991, he joined Synopsys, Inc. where he ultimately became Chief Technical Officer and Senior Vice-President of Research. In 1998, Kurt became Professor of Electrical Engineering and Computer Science at the University of California at Berkeley. Kurt's research group is currently focused on using parallelism to accelerate the training and deployment of Deep Neural Networks for applications in computer vision, speech recognition, multi-media analysis, and computational finance. Kurt has published six books, over 250 refereed articles, and is among the most highly cited authors in Hardware and Design Automation. Kurt was elected a Fellow of the IEEE in 1996. Kurt's earlier research has had a lasting impact on Electronic Design Automation. For example, at the 50th Design Automation Conference Kurt received a number of awards reflecting achievements over the 50 year history of the conference. These awards included for "Top Ten Cited Author", "Top Ten Cited Paper" and he was also recognized as among one of only three people to have received four Best Paper Awards in the history of the conference. While at Synopsys, he worked with the General Counsel to co-develop the company's intellectual property strategy. Since joining Berkeley, he has consulted directly with high-tech companies to do assessments of large (100+ patent) portfolios to assess their defensibility and monetary value. He has also worked, as time has allowed, as an IP consultant on individual patent infringement cases with many of Silicon Valley's top IP law firms. As an angel investor Kurt has participated in the capital formation of twelve start-up companies including Coverity and Tensilica. Of these twelve, six have already seen profitable exits and two are still independent and growing. Kurt has also served as an advisor to nine other start-ups including two that went on to become public companies.

James Demmel is the Dr. Richard Carl Dehmel Distinguished Professor of Computer Science and Mathematics at the University of California at Berkeley. Prof. Demmel is the chair of the EECS Department. His personal research interests are in numerical linear algebra, high performance computing, and communication avoiding algorithms in general. He is known for his work on the LAPACK and ScaLAPACK linear algebra libraries. He is a member of the National Academy of Sciences, National Academy of Engineering, American Academy of Arts and Sciences, a Fellow of the ACM, IEEE and SIAM, and winner of the IEEE Computer Society Sidney Fernbach Award, the SIAM J. H. Wilkinson Prize in Numerical Analysis and Scientific Computing, IPDPS Charles Babbage Award, and numerous best paper prizes, including being the only three-time winner of the SIAM Linear Algebra (SIAG/LA) Prize. He was an invited speaker at the 2002 International Congress of Mathematicians and the 2003 International Congress on Industrial and Applied Mathematics.

Affiliations

Yang You[1] · Yuxiong He[2] · Samyam Rajbhandari[2] · Wenhan Wang[2] · Cho-Jui Hsieh[3] · Kurt Keutzer[4] · James Demmel[4]

Yuxiong He
yuxhe@microsoft.com

Samyam Rajbhandari
samyamr@microsoft.com

Wenhan Wang
wenhanw@microsoft.com

Cho-Jui Hsieh
chohsieh@cs.ucla.edu

Kurt Keutzer
keutzer@berkeley.edu

James Demmel
demmel@cs.berkeley.edu

1 Department of Computer Science, National University of Singapore, Singapore, Singapore

2 Microsoft Research, Microsoft Corporation, Redmond, WA, USA

3 Department of Computer Science, UCLA, Los Angeles, CA, USA

4 Computer Science Division, UC Berkeley, Berkeley, CA, USA