Approximate Random Dropout for DNN training acceleration in GPGPU

***Abstract*—The training phases of Deep neural network (DNN) consumes enormous processing time and energy. Compression techniques utilizing the sparsity of DNNs can effectively accelerate the inference phase of DNNs. However, it can be hardly used in the training phase because the training phase involves dense matrix-multiplication using General Purpose Computation on Graphics Processors (GPGPU), which endorse regular and structural data layout. In this paper, we propose the Approximate Random Dropout that replaces the conventional random dropout of neurons and synapses with a regular and predefined patterns to eliminate the unnecessary computation and data access. To compensate the potential performance loss we develop a SGD-based Search Algorithm to produce the distribution of dropout patterns. We prove our approach is statistically equivalent to the previous dropout method. Experiments results on MLP and LSTM using well-known benchmarks show that the proposed Approximate Random Dropout can reduce the training time by 20%-77% (19%-60%) when dropout rate is 0.3-0.7 on MLP (LSTM) with marginal accuracy drop.**

***Index Terms*—training, neural network, dropout**

I. Introduction

Deep Neural Networks (DNNs) have emerged as critical technologies to solve various complicated problems [1], [2]. The inference of DNNs is computational expensive and memory intensive and therefore has an urgent need for acceleration before we can fully embrace DNNs in the power-limited devices. Extensive works are proposed to reduce the computation by compressing the size of synaptic weights, such as weight pruning [3], [4], quantization [5]–[7], low rank [8], [9] and Compact Network Design [10]. The above compression techniques may require retraining the DNN with limited accuracy loss ($< 1\%$). The success of these techniques relies on the sparsity and plasticity of DNNs, however, cannot directly apply to the training phase of DNNs.

The training phase, involving the back-propagation through the network to update the weights, demands three-times more computation effort. GPGPU is suitable for such task which is attributed to the superior parallelism for large matrix multiplication [11], [12]. Extensive works propose to accelerate the training phase on the distributed GPU-based system [13], [14]. Other works [15], [16] focus on accelerating the training phase using gradient pruning and weight quantization, respectively.

Random Dropout technique addresses the over-fitting problem and is widely used in MLP and LSTM training. The most common method [17] randomly dropping some neurons of each layer in every training iteration, while the other (DropConnect [18]) aims the same goal by randomly dropping some synapses connections between layers, namely some elements in weight matrix. Theoretically, we can reduce the number of multiplication to 30%-70% if we can skip the calculation of all the dropped neurons or synapses while the dropout rate changes from 0.3 to 0.7. However, such tremendous saving of multiplication as well as the corresponding data access is hard to exploit. Because the neurons or synapses are randomly and irregularly dropped following the Bernoulli distribution. Such irregularity prevents the GPGPU's single instruction multiple threads (SIMT) architecture to skip the unnecessary multiplication and memory access.

Therefore, in this work, we replace the random dropout with two types of regular dropout patterns to make the choices of dropped neurons or synapses predictable, which allow GPGPU to skip calculation of those dropped neurons or synapses. We further developed an SGD-based Search Algorithm to produce the distribution $\mathcal{K}$ of dropout patterns such that the dropout rate of each neuron is approximately subjected to a Bernoulli distribution (We provide a brief proof). In each iteration, we sample a dropout pattern subjected to $\mathcal{K}$ and then eliminate the redundant computation by omitting the dropped data during the hardware allocation. Our experiments show that applying Approximate Random Dropout during training can reduce the training time by 20%-77% (19%-60%) when dropout rate is 0.3-0.7 on MLP (LSTM) with less than 0.5% accuracy loss. We find that when the batch size increases, the speedup rate increases with accuracy of neural network declines.

The reminder of the paper is organized as follows: Section II introduces the related works and motivates this paper. Section III describes the proposed Approximate Random Dropout Technique. Experiments are shown in Section IV. Section V concludes this paper.

II. Background

A. Accelerating DNN inference and training

There are considerable works pitch into accelerating inference of DNN by leveraging the sparsity of DNN [4], [5], [10], [19]. Han et al. [3] prune synaptic weights which are close to zero and then retrain the DNN to maintain the classification accuracy. The zero weights are then encoded and moved onto the on-chip memory. Special decoder is deployed in the accelerator to decode the zero weights and skip the computation. Consequently, above methods can only benefit ASIC/FPGA based DNN accelerator instead of GPU. Jaderberg et al. [8] and Ioannou et al. [9] use low-rank representations to create computationally efficient neural networks. These methods cannot be used in training phase because of the subtle change of the weights degrades the convergence and accuracy of the training phase.

Extensive works propose to accelerate the training phase on the distributed GPU-based system [13], [14]. Wen et al. [13] propose to use ternary gradients to accelerate distributed deep learning in data parallelism. Zhang et al. [14] propose a variant of the asynchronous SGD algorithm to guarantee the convergence of this algorithm and accelerate the training in a distributed system. Other works are relative to the acceleration in the training process using gradient pruning and weight quantization. Kster et al. [16] share the exponent part of the binary coding of the weights and thereby convert floating-point operations to fixed-point integer operations. Noted that this work is compatible with ours and we leave this topic to further research. Sun et al. [15] prune those comparatively small gradients to speed up training phase. However, their work focuses on software-level optimization and thus yields marginal training acceleration while this work enable computation reduction on hardware-level.

B. Basics of the GPGPU

GPGPU is commonly used for DNN training. It is composed of dozens of streaming multiprocessors (SMs). Each SM consists of single instruction multiple threads (SIMT) cores and a group of on-chip memories including register file, shared memory, L1D cache and etc. Each SM manages and executes multi-threads on it. Those threads are clustered into warps (e.g., 32 threads in NVIDIA GPU), executing the same instruction at the same time. Thus, the branch divergence occurs when programmers write conditional branch (if-else).

Shared memory is a performance-critical on-chip memory. The latency of accessing the global memory (DRAM) is roughly 100x higher than that of accessing the shared memory. Hence, reducing the frequency of accessing global memory is critical for performance. The capacity of the shared memory per block is 48KB in Nvidia GTX 1080Ti, which is much smaller than the capacity of the global memory. Therefore, reducing the superfluous data in shared memory is also important.

The key purpose of this work is to reduce the scale of matrices, by which we can reduce the access frequency of the shared memory and the global memory as well as the computation effort to accelerate the training.

C. Random Dropout

Random dropout is widely used to prevent over-fitting. It randomly omits part of the neurons [17] or synapses [18] on each training iteration. The probability of a neuron or a synapses to be dropped is subjected to a Bernoulli distribution parameterized with *dropout rate* [18], [19]. In a nutshell, the main reason why random dropout can effectively prevent over-fitting is that it generates adequate different sub-models to learn diverse features during the training process and ensembles those sub-models to maximize the capability of DNN for inference.

Existing machine learning frameworks, like Caffe [20] and Tensorflow [21], all adopt the dropout technique. For each layer in the forward propagation, the output matrix is computed and thereafter element-wisely multiplied by a mask matrix composed of randomly generated 0s and 1s, as shown in Fig. 1(a). Similarly, in back-propagation, they first calculate the derivatives of the output matrix. The resulting derivative matrix then multiplies by the same mask matrix.

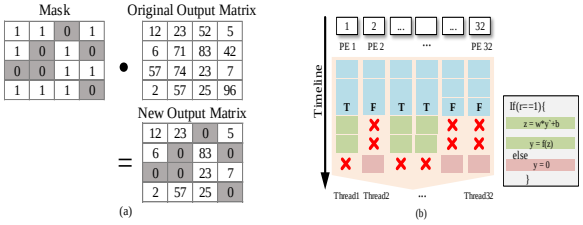


Fig. 1. (a) Implementation of Random Dropout in Forward propagation; (b) GPU divergence happened when directly skipping redundant calculation.

A question arises: why not skipping the calculation of those dropped neurons to reduce the redundant time spent on the matrix multiplication and the data movement. Intuitively, we can write conditional branch (if - else) to skip the redundant calculation. However, such conditional branches incur branch divergence in GPU, which is a great hurdle for performance. As shown in Fig 1(b), 'T' denotes the threads that are satisfying the conditions ($r = 1$) and executing the green function($z = w * y' + b$ and $y = f(z)$), while 'F' refers to those executing the red function($y = 0$). In GPGPUs SIMT architecture, the red threads have to wait for the green threads. Thus, some process elements (PEs) are idle, represented by the red cross. The total execution time is not reduced (even increased) due to the branch divergence. Thus, it is non-trivial to exploit the dropout for speedup in GPU.

III. Approximate Random Dropout

The key idea of accelerating the DNN training is to reduce the scale of matrices involved in multiplication and avoid the brunch divergence of GPU. However, the randomness in conventional dropout methods hamper the scale reduction.

In this work, we define *dropout pattern* as the combination of dropped neurons in each training iteration. As shown in Fig. 2, we design two sets of regular dropout pattern and replace the random dropout with a sampled dropout pattern

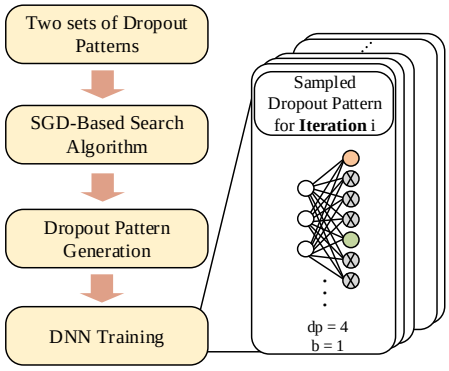
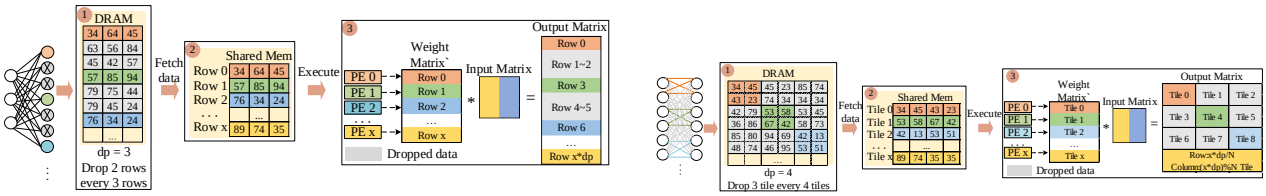


Fig. 2. Overview of the Approximate Random Drouput.



(a) Row-based Dropout Pattern

(b) Tile-based Dropout Pattern

Fig. 3. GPU hardware allocation during training.

sampled from them. Resulted from the replacement, we can forecast which neurons or synapses to be dropped and thereafter assist GPU to skip the calculation and data access of the dropped neurons without incurring the divergence. We modify the caffe source code to reduce the scale of matrices which become feasible due to the predictable dropout.

However, the loss of randomness induced by the regular dropout patterns increases the risk of over-fitting the DNN. To cope with this issue, we further develop a Stochastic Gradient Decedent (SGD) based Search Algorithm (see section III-C), to find a distribution of all possible dropout pattern such that the probability distributions of each neuron or synapse being dropped between our method and conventional method is equivalent. We provide a brief proof of that.

In this section, based on the computation characteristic of GPU, we firstly propose two sets of Dropout Patterns—Row-based Dropout Pattern (RDP) and Tile-based Dropout Pattern (TDP)—and then analyze the mechanisms of the reduction of the time of computation and data access. After that, we introduce our SGD-based Search Algorithm which produce a distribution of possible dropout patterns as well as the dropout pattern generation procedure in each iteration.

A. Row-based Dropout Pattern

In conventional dropout method, the computation relative to a dropped neuron is the multiplication between zero and the correspondent row in the weight matrix of next layer to shrink the scale of matrix, in RDP, we drop the whole row in the weight matrix, which is equivalent to drop all the synapses of a specified neuron.

Concretely, RDP is parameterized by two scalar dp and bias b as follow: we uniformly choose a bias $b \in \{1, ..., dp\}$ and drop all rows in the weight matrix whose indices satisfy

$$i : (i - b) \bmod dp = 0 \tag{1}$$

Consequently, $(i-1)/(i)$ of the neurons are dropped. For instance (the left of Fig. 3(a)), when dp=3, b=1, starting from the first row, we drop two rows (i.e., neurons) in every successive three rows (neurons) in the weight matrix.

Given the size of the output matrix as $M \times N$, the maximum dp is $dp_{max} = M$, and the maximum number of the sub-models is $\sum_{i=1}^{M} i = (M+1)/2$ considering the number of possible bias is i when $dp = i$.

The execution processes in GPU is also shown in Fig. 3(a). DRAM stores the whole weight matrix (as shown in step 1); the gray block denotes the rows of weight matrix correspondent to the dropped neurons. We write the kernel function to prevent GPU from fetching those dropped data into shared memory (as shown in step 2) and build two compact matrices (input matrix and weight matrix) for next step. After data fetch, every PE multiplies one row of the weight matrix by the whole input matrix. Thus, only $\frac{1}{dp}$ of the original weight matrix as well as the input matrix is fetched and calculated. The resulting rows fill $1 \times dp$ rows in the Output Matrix using the same pattern. The rest $\frac{dp-1}{dp}$ of the Output Matrix is set to zero by default. Note that the RDP is agnostic to the matrix-multiplication algorithm as it temporarily compresses the matrices into a compact layout. Therefore, RDP can comply to any optimization method for matrix multiplication.

B. Tile-based Dropout Pattern (TDP)

Tile is a sub-matrix in weight matrix and contains multiple synapses connections. We use tiles as the unit to drop rather than synapse [18] (namely the size of tiles is 1) for the purpose of regularity. TDP is also parameterized by dp and bias b. $dp-1$ tiles are dropped in every dp tiles, resulting in $\frac{dp-1}{dp}$ of synapses connections being dropped. As shown in the left of Fig. 3(b), when $dp = 4, b = 1$, starting from first tile, we drop 3 tiles in every 4 successive tiles.

TDP have similar procedure compare to RDP but different in: (1)TDP fetches non-dropped tiles into the shared memory rather than rows, and builds two compact matrices. (2) each PE conduct the multiplication of one tile of compact weight matrix and the corresponding tile of compact input matrix, according to their PE index. In the right of Fig. 3(b), GPU only conduct multiplication of two compact matrices whose scale is $\frac{1}{4}$ of the original scale. This Dropout Pattern can naturally work with Tiling Method [22] in matrix multiplication, which is an essential optimization technique.

Given the size of the output matrix $M \times N$, the size of the tile $x \times y$, the maximum dp is $dp_{max} = \lfloor M/x \rfloor \times \lfloor N/y \rfloor$ and the maximum number of sub-models is $(1 + dp_{max})/2$. TDP can generate more sub-models than RDP, when N is roughly greater than $x \times y$.

The choice of tile size is critical: the smaller size of the tile, the more number of Dropout Patterns as well as sub-models, while small tile leads to fine-grained control. Under such circumstances, the size of tile is set to be 32×32 to balance the maximization of the number of sub-models and avoiding

Algorithm 1 SGD-based Search Algorithm

Input:

the target global dropout rate p, the maximum number of dropout pattern N, hyper parameters λ_1, λ_2 ($\lambda_1+\lambda_2 = 1$).

Output:

A distribution $\mathcal{K}$ for different dropout patterns.

1: Initialize a N dimension row vector v;
2: Initialize a N dimension constant row vector p_u as [0, 1/2, 2/3, ..., (N-1)/N];
3: **while** $|\Delta loss| \geq threshold$ **do**
4: $d = softmax(v)$;
5: $E_p = ||d^T \cdot p_u - p||_2^2$;
6: $E_n = \frac{1}{N}\sum_{i=1}^{N} d_i \log d_i$;
7: $Loss = \lambda_1 E_p + \lambda_2 E_n$;
8: Calculate gradients of loss over v;
9: Update v with gradient descent;
10: **end while**
11: Return d;

shared memory's bank conflict since the shared memory has 32 banks in NVIDIA GPU.

A typical training process is composed of three steps: fully connected layer computation, activation layer computation and dropout layer computation using the mask matrix. After applying the Dropout Pattern with $dp = 2$, we only need to spend half of the time for fully connected layer computing and skip the dropout layer computing. Consequently, given the dropout pattern, the time spending on training can be overtly reduced.

C. SGD-based Search Algorithm for Dropout Pattern Distribution

For each iteration in training procedure, only one regular dropout pattern is applied to the network. In order to approximate the traditional dropout process [17], the dropout pattern we choose in each iteration should satisfy that: (1)the dropout probability of each neuron should subject to a given Bernoulli distribution, and (2)different sub-models derived from that series of dropout patterns should be adequate.

Therefore, we propose an efficient SGD-based Search Algorithm to find a dropout pattern distribution from which the dropout pattern sampled satisfy the demands. SGD consumes tractable time and is convenient in optimizing the continuous variables. More specifically, the algorithm obtains a probability distribution $\mathcal{K} = \{k_i\}_{i=1}^{dp_{max}}$ which contains the probability k_i of each possible Dropout Pattern $i \in \{1, 2, ..., dp_{max}\}$, who is subjected to $\sum_{i=1}^{dp_{max}} k_i = 1$.

Here we define the *global dropout rate* as the proportion of neurons or synapses who are set to zero. Noted that the global dropout rate is different from the conventional dropout rate which refer to the probability of a single neuron or synapse to be dropped. However, we prove that within our approach the two dropout rate are statistically equivalent.

Given the target global dropout rate p, and the maximum dp as N, we use Algorithm 1 to search for desired distribution $\mathcal{K}$. A vector v with length N is first arbitrary initialized (line 1) and the $softmax(v)$ serve as the final probability distribution of each dropout pattern (line 4). Then we setup a constant vector $p_u = \{\frac{i-1}{i}\}_{i=1}^{N}$ whose element denotes the global dropout rate of a given dropout pattern. Therefore, $d^T \cdot p_u$ is the expected global dropout rate and the difference between it and the target global dropout rate is our optimization objective (line 5). To enforce d to be dense and to produce more diversified sub-models, the negative information entropy of d is added to the loss (line 6, 7). Then the algorithm uses SGD algorithm to update v (line 8, 9) and return the distribution $d \in [0, 1]^N$ when the loss is stuck. By the loss function we defined in line 7, the algorithm aims at finding a distribution $\mathcal{K}$ that (1)make the global dropout rate equal to required value p and (2)maximize the sub-models diversity.

D. Dropout Pattern Generation

The acquired distribution $\mathcal{K}$ is then used to sample dropout pattern in each iteration. Concretely, in each iteration, we randomly sample a dropout pattern (parameterized by dp and b) subjected to the distribution $\mathcal{K}$, and then uniformly choose a bias $b \in \{1, ..., dp\}$. Dropout pattern is then determined.

In our method, global dropout rate is statistically equivalent to the single neurons or synapse dropout rate. For each neuron or synapse, the probability of it to be dropped (conventional dropout rate) is:

$$p_n = \sum_{i=1}^{dp_{max}} p_b k_i = \sum_{i=1}^{dp_{max}} \frac{i-1}{i} k_i \tag{2}$$

The global dropout rate of $\mathcal{K}$ is:

$$p_g = d^T \cdot p_u = \sum_{i=1}^{dp_{max}} k_i \frac{i-1}{i} \approx p \tag{3}$$

Therefore, in terms of the whole training process, the dropout rate p_n of a single neurons or synapse is equal to the global dropout rate p_g and thus is approximately equal to the target dropout rate p by the SGD-based Search Algorithm.

IV. Experiments

To evaluate the effectiveness of proposed approximate random dropout, we compare it with the conventional dropout technique in terms of the DNN accuracy and the training time. In section IV-A, in order to explore the influence of the dropout rate on the performance of a specific 4-layer Multilayer perceptron (MLP), we vary different dropout rate on a MLP. Note that the dropout rate in our method refer to the target dropout rate p as described in Section III-C. In section IV-B, we compare different MLPs with a specific dropout rate. The data set we use with MLP is MNIST [23]. Long short-term memory(LSTM) neural network [24] is used in section IV-C to verify the scalability of our method. The dataset we used with LSTM include a dictionary whose size is 8800, and the Penn Treebank (PTB) [25] data set which has long been a central data set for language modeling. The experiment codes is implemented in Caffe [20] and use a single GTX1080Ti GPU to run.

A. Comparison of different dropout rate

The structure of a specific 4-layer MLP is described as follow: the input layer is shaped according to the batch size; the output layer has 10 neurons for digit 0 to 9; the two hidden layers have 2048 neurons both. During training, we set the following hyper-parameter: the batch size is 128, the learning rate is 0.01, and momentum is 0.9.

We vary the dropout rate from $(0.3, 0.3)$ to $(0.7, 0.7)$ (two hidden layers may have varied dropout rate), and record the accuracy and training time for each dropout rate. The comparison of two metrics of RDP and TDP against the conventional dropout are shown in Fig. 4. The training time of conventional dropout is divided by the new training time of proposed approximate random dropout to obtain the speedup rate.

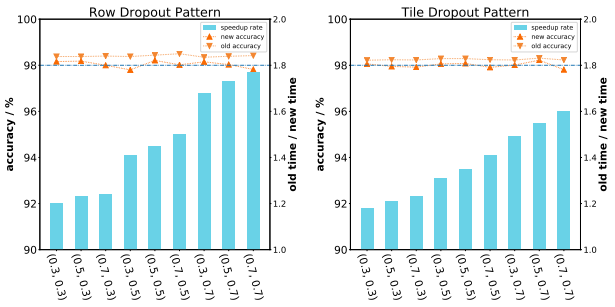


Fig. 4. Comparing different dropout rate combinations on specific network

The results show that RDP can obtain $20\% - 80\%$ speedup compared with the traditional dropout technique when the dropout rate varies between 0.3 and 0.7, which comply to our intuition as the amount of data that require no calculation expands with the increment of the dropout rate. The speedup rate brought by TDP ranges from 1.18 to 1.6. The little slowdown is induced by the calculation of the nonzero positions in the output matrix before matrix multiplication. The accuracy loss of these two classes of dropout patterns is less than 0.47%, which is the evadible concession to the speedup. TDP has less accuracy loss than RDP which can be attributed to the abundance of sub-models in TDP.

B. Comparison of different networks

We investigate the speedup in different MLP structures using a fixed dropout rate (0.7, 0.7). Those MLPs have the same input and output layer as described in section IV-A. Their hidden layer size is shown in Table I. For instance, 1024×64 in the second column means the first and the second hidden layer's size are 1024 and 64, respectively. The hyper-parameters of optimization algorithm follow above experiments.

From Table I, the accuracy degradation is less than 0.5%. In some cases, the accuracy even increases. Moreover, the speedup rate increases as the network size increases. Especially, in the case of 4096×4096 network, both of the proposed dropout patterns reach a $2\times$ speedup.

TABLE I
COMPARING DIFFERENT NETWORK WITH SPECIFIC DROPOUT RATE

Dropout rate	Network size	Dropout pattern	Accuracy(its loss)	Speedup rate
0.7	1024*64	ROW	98.07%(-0.42%)	1.27
		TILE	98.11%(-0.38%)	1.19
	1024*1024	ROW	98.01%(-0.35%)	1.45
		TILE	98.15%(-0.21%)	1.41
	2048*2048	ROW	98.44%(0.37%)	1.77
		TILE	98.5%(-0.31%)	1.60
	4096*4096	ROW	98.00%(-0.47%)	2.16
		TILE	98.16%(-0.31%)	1.95

C. Scaling to Long Short-Term Memory Model

We evaluate the speedup rate and the model performance on LSTM, which predicts the following word based on the given words. Each of the two hidden layers of LSTM contain 1500 neurons. During training, we set the following hyper-parameters: the base learning rate is 1 (the base learning rate will gradually decrease), batch size is 20, the maximum epoch is 50, and the length of the sequence is 35. It should be noted that the execution of LSTM is also performed as matrix multiplication, thus our proposed approximate dropout can be easily applied to LSTM.

As shown in Table II, the accuracy degradation is less than 1%. When dropout rate is increasing, the speedup rate increases without undermining the accuracy loss.

TABLE II
A DICTIONARY DATA SET WHICH CONTAINS 8800 WORDS ON LSTM.

dropout rate		(0.3,0.3)	(0.5,0.5)	(0.7,0.7)
accuracy	original	47.9%	47.3%	45.9%
	ROW	46.9%	46.0%	44.5%
	TILE	47.2%	46.5%	44.4%
speedup	original	1.0	1.0	1.0
	ROW	1.18	1.47	1.53
	TILE	1.18	1.43	1.49

To illustrate the effectiveness of the proposed method, we fix the dropout rate to 0.5 and trace the accuracy of RDP until it's convergence. As shown in Fig. 5, the red curve records our approximate random dropout training process; the blue one records the traditional dropout. The convergence of our method is eariler than the traditional dropout. Moreover, the smoothness of red curve indicates the approximate random dropout is helpful for the training process.

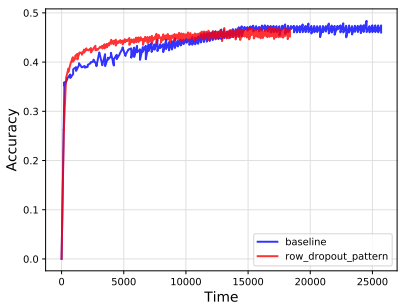


Fig. 5. The training process of RDP and traditional dropout.

The result using the Penn Treebank data set (PTB) [25] on the 3-layer LSTM is shown in Fig. 6(a). The test perplexity using RDP only increases 0.04 given the dropout rate is 0.7, which further shows that our proposed approximate dropout algorithm can generate adequate sub-models for PTB data set. Besides, when dropout rate increases from 0.3 to 0.7, the speedup rate also increases from 24% to 85%.

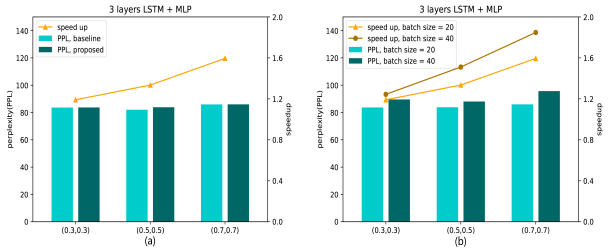


Fig. 6. Speedup rate and accuracy of Row approximate dropout on 3-layer LSTM.

We vary the batch size from 20 to 40. Noted that SGD based search and data initialization are an one-time effort. When the batch size is increased, only the matrix operation and data transmission time increase accordingly. As shown in Fig. 6(b), the speedup rate increases when batch size increases. However, since one dropout pattern is applied to the whole batch, the sub-models generated during training may not be sufficient, which raises the perplexity.

V. Conclusion

Accelerating DNN training is difficult because it is non-trivial to leverage the sparsity of DNN in the dense matrix-multiplication. In this work, we propose a novel approach to eliminate the unnecessary multiplication and data access by replacing the traditional random dropout with an approximate random dropout. The two classes of dropout patterns can avoid the divergence issue in GPU, reduce the scale of the matrix, and thus gain significant improvement on the energy-efficiency with marginal decline of the model performance. The proposed SGD-based search algorithm can guarantee the dropout rate of single neurons or synapse is equivalent to the conventional dropout, as well as the convergence and accuracy of the models. In general, the training time can be reduced by $20\% - 77\%$ and $19\% - 60\%$ when dropout rate is 0.3-0.7 on MLP and LSTM, respectively. Moreover, higher speedup rate is expected on a larger DNN. The proposed method has been wrapped as an API and integrated into Caffe. The speedup can be much higher if the proposed method can be integrated into the cuBLAS Library.

References

[1] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," 2017.

[2] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," 2017.

[3] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[4] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, and G. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulantweight matrices," 2017.

[5] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," 2017.

[6] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *Computer Science*, 2014.

[7] C. Leng, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with admm," 2017.

[8] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *Computer Science*, vol. 4, no. 4, p. XIII, 2014.

[9] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi, "Training cnns with low-rank filters for efficient image classification," *Journal of Asian Studies*, vol. 62, no. 3, pp. 952–953, 2015.

[10] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," 2017.

[11] M. eransk, "Training recurrent neural network using multistream extended kalman filter on multicore processor and cuda enabled graphic processor unit," in *International Conference on Artificial Neural Networks*, pp. 381–390, 2009.

[12] S. Puri, "Training convolutional neural networks on graphics processing units," 2010.

[13] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," 2017.

[14] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *International Joint Conference on Artificial Intelligence*, pp. 2350–2356, 2016.

[15] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," 2017.

[16] U. Kster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, and L. Hornof, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Neural Information Processing Systems*, 2017.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[18] L. Wan, M. D. Zeiler, S. Zhang, Y. Lecun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, pp. 1058–1066, 2013.

[19] W. Wen, Y. He, S. Rajbhandari, M. Zhang, W. Wang, F. Liu, B. Hu, Y. Chen, and H. Li, "Learning intrinsic sparse structures within long short-term memory," 2017.

[20] Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, and Jonathan, "Caffe: Convolutional architecture for fast feature embedding," pp. 675–678, 2014.

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, and M. Devin, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, ACM, 2008.

[23] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[24] A. Graves, *Long Short-Term Memory*. Springer Berlin Heidelberg, 2012.

[25] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.