Approximate Random Dropout

Zhuoran Song, Li Jiang
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai
songzhuoran@sjtu.edu.cn, jiangli@cs.sjtu.edu.cn

Abstract

The training phases of Deep neural network (DNN) consume enormous processing time and energy. Compression techniques for inference acceleration leveraging the sparsity of DNNs, however, can be hardly used in the training phase. Because the training involves dense matrix-multiplication using GPGPU, which endorse regular and structural data layout. In this paper, we exploit the sparsity of DNN resulting from the random dropout technique to eliminate the unnecessary computation and data access for those dropped neurons or synapses in the training phase. Experiments results on MLP and LSTM on standard benchmarks show that the proposed Approximate Random Dropout can reduce the training time by half on average with ignorable accuracy loss.

1 Introduction

Deep Neural Networks (DNNs) have emerged as critical technologies to solve various complicated problems [3, 18, 30]. The inference of DNNs is computation and memory intensive and therefore has an urgent need for acceleration before we can fully embrace DNNs in the power-limited devices. Extensive works propose to reduce the computation and compress the size of synaptic weights with DNN retraining such as weight pruning [10, 27, 5], quantization [33, 7, 17, 12], low rank [14, 13] and Compact Network Design [11, 23, 32]. The success of these techniques relies on the sparsity and plasticity of DNNs, however, cannot directly apply to the training phase of DNNs.

Training phase, involving an extra back-propagation through the network to update the weights, is more compute and memory intensive. Graphics Processing Unit (GPU) is suitable for such task thanks to its unique capability of large matrix multiplication Černanský [6], Puri [21]. Extensive works, Wen et al. [29], Goyal et al. [8], Zhang et al. [31], Dean et al. [4] propose to accelerate the training phase on the distributed GPU-based system. Other works aim at accelerating the convergence of the training phase of LSTM, by improving the conditioning of the optimization problem Salimans and Kingma [22], and by simplifying LSTM acoustic models Miao et al. [19]. [25, 16] focus on accelerating the training phase using gradient pruning and weight quantization, respectively.

Dropout technique [24] addresses the overfitting problem by randomly dropping some output neurons of each layer in every training iteration. Similarly, DropConnect [26] randomly drop some synapses connections of each layer to address the overfitting problem. When the dropout rate ranges from 0.3 to 0.7, we can reduce the number of multiplication to 30% to 70%, if we can skip the calculation of all the dropped neurons or synapses. Such tremendous saving of multiplication as well as the corresponding data access, however, is not exploited. Because the training phase randomly drops the neurons or synapses following the Bernoulli distribution, zero values are randomly and irregularly distributed in the input matrix. Such irregularity prevents the GPU architecture to skip the unnecessary multiplication and memory access.

*Use footnote for providing further information about author (webpage, alternative address)—not for acknowledging funding agencies.

Preprint. Work in progress.
Therefore, in this paper, we first design two deterministic dropout patterns—the choice of dropped neurons or synapses—to allow GPU to skip calculation of dropped neurons or synapses. To maintain the accuracy of the DNN, we then propose a random generator to produce as many regular dropout patterns—sub-models—as possible. The random generator employs an SGD-based Search Algorithm to determine the distribution of regular dropout patterns that is approximately close to the Bernoulli distribution. Finally, a mathematical proof is provided to show the expected squared error of the ensemble model of the proposed Approximate Random Dropout is far less than the expected squared error of the corresponding sub-model.

2 Background

2.1 Related works

Few works target on accelerating the training by leveraging sparsity of DNN. Sun et al. \[25\] prunes the gradients to speed up training phase. In the back propagation, only the top-k elements (regarding magnitude) in the gradient vectors are used to update about 4% of the weights at each back propagation pass. Köster et al. \[16\] shares the exponent part in the binary coding of the weights and thereby converts floating-point operations into integer fixed-point operations. Their experiment results show that 16-bit Flexpoint closely matches 32-bit floating point, without any need to tune the hyper-parameters. The trade-off between the accuracy and performance is well accepted. For instance, Wen et al. \[27\] proposes a Structured Sparsity Learning (SSL) method to regularize the structures of DNN. It achieves almost 3x speedup on GPU with 2% loss in model accuracy compared with L1 norm neural network model. Zhou et al. \[33\] convert pre-trained full-precision neural network model into a low-precision version whose weights are constrained to be either power of two or zero. It gains 89X compression rate by decreasing 1.47% accuracy compared with the original neural network model.

This paper approaches an alternative way to tradeoff the accuracy for the reduction of training time. This method is compatible with any previous methods and thus can be used simultaneously with previous methods.

2.2 Motivation

The basic idea of dropout is to randomly omit part of the neurons or synapses on each training iteration, based on the Bernoulli distribution \[26, 20, 28\]. In a nutshell, the main reason why dropout can effectively prevent overfitting is dropout generates adequate sub-models to learn diverse features during training and ensembles those sub-models to maximize the capability of DNN for inference. Existing machine learning frameworks, like Caffe \[15\] and Tensorflow \[1\], first calculate the output matrix, and then multiplies each output neuron by a mask matrix composed of randomly generated 0s and 1s, as shown in Fig. 1(a). Similarly, in back-propagation, those frameworks first calculate the derivatives of output matrix, and multiplies each output neuron by the same mask matrix generated by the forward-propagation.

![Motivation of Example](https://example.com/motivation.png)

**Figure 1:** Motivation of Example: (a) Implementation of Random Dropout in Forward propagation; (b) GPU divergence happened when directly skipping redundant calculation.
Intuitively, we can write conditional branch (if - else) to skip the calculation and data access for those dropped neurons. However, such conditional branches incur divergence in GPU. The high-parallelism of GPU results from the Single-Instruction-Multiple-Data architecture. Multiple threads of execution (e.g., 32 threads in NVIDIA GPU) are grouped in warps, 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). As threads in one warp executing the same instruction at the same time, the red threads have to wait for the green threads. Thus, the process element (PE) is idle, represented by the red cross. The resulting divergence prevents us from reducing the computation time.

3 Proposed Approximate Random Dropout Algorithm

The key idea of avoiding the divergence is to replace the random dropout with deterministic dropout, as shown in Fig. 2. We denote dropout pattern as the choices of dropped neurons in each training iteration. Before training, we generate N dropout patterns subjected to a specified distribution described in section 3.2, wherein N is the total training iterations. Then when training, the correspondent patterns is retrieved and apply to current network.

We use two classes of dropout patterns to replace the union random dropout (see section 3.1). These dropout patterns are friendly to GPU. The GPU can skip the calculation and data access involving the dropped neurons without incurring the divergence. However, the loss of randomness induced by the deterministic dropout undermines the capability of overfitting reduction. To cope with this issue, we further develop a random dropout Pattern Generator, guided by a Stochastic-Gradient-Decedent (SGD) based search algorithm (see section 3.2), to mimic a reasonable sequence of dropout patterns. The probability of each neuron and/or synapse dropped is approximately equal to that in the random dropout method. Although the dropout pattern in each iteration is deterministic, we prove that each neuron and/or synapses have statistically dropped for sufficient times (see subsection 3.3).

![Figure 2: General process overview of Approximate Random Dropout Algorithm.](image)

3.1 Two class of dropout patterns

Based on the computation characteristic of GPU, we propose two dropout patterns—row dropout pattern and tile dropout pattern—and analyze the mechanisms to reduce the computation and data access.

3.1.1 Row Dropout Pattern

To avoid divergence, we drop the whole row in the weight matrix, which is equivalent to drop all the synapses of the neuron. We defined dp as follows: when dp = i, we drop i − 1 rows in every i successive rows in the weight matrix. Consequently, \( \frac{i-1}{i} \) neurons are dropped. For instance, as shown in the left of Fig. 3, when dp = 3, we drop two rows (i.e., neurons) in every successive three rows (neurons) in the weight matrix.

The execution processes in GPU is shown in Fig. 3. DRAM stores the whole weight matrix (as shown in procedure 1); the gray block denotes dropped the data. We write the kernel function to prevent
3.1.2 Tile Dropout Pattern

Tiles are sub-matrices in weight matrix. Previous work \cite{26} shows set some tiles to zeros can also perform regularization. In this section, we also use $dp$ to represent the parameter of this pattern. It means when $dp = 1$, $i-1$ tiles are dropped every $i$ tiles, resulting $\frac{i-1}{i}$ synapses connections being dropped. For instance, as shown in the left of Fig. 4, when $dp = 4$, we drop 4 synapses connections every 5 synapses connections.

The way how GPU do matrix multiplication is shown in Fig. 4. Firstly, the whole weight matrix (corresponding to the actual synapses in the neural network) is stored in DRAM as shown in procedure 1: the gray blocks denote dropped weights, which do not participate in computing. Secondly, during fetch data process, GPU does not fetch those dropped data (synapses connections) into shared memory, as shown in procedure 2. Therefore, GPU only needs to consider $\frac{1}{i}$ of the original weight matrix. Thirdly, every PE conduct the computing between one tile of input matrix 1 and corresponding tile of input matrix 2, according to their PE index.

Our tile dropout pattern can generate more sub-models than row dropout pattern on most occasions. Given the output matrix’s size $M \times N$, the size of tile $x \times y$, the number of tile approximate dropout pattern $W$, the number of dropout patterns is $\frac{M \times N}{x \times y}$. The choice of tile size is critical. The smaller size of the tile, the more number of dropout patterns. However, if the size of tile is too small, it will lead to fine-grained control. Under the circumstances, the size of tile is set to be 32 here when considering the number of dropout patterns and bank conflicts in the meantime.

3.2 SGD Based Search Algorithm for random dropout pattern distribution

Two requirements should be satisfied to approximate the traditional dropout process \cite{24}: the probability of drop patterns should be subject to a given Bernoulli distribution; different sub-models should be adequate.
Therefore, we propose an efficient local search algorithm based on SGD to guide the generation of dropout patterns, because SGD algorithm consumes traceable time and is convenient in optimizing the continuous variables. The search algorithm obtains a distribution $K$ which contains the probability of each possible dropout pattern $k$, wherein $K \in \mathbb{R}^N$, $N$ denotes the largest dropout pattern and $\sum_{k \in K} k = 1$. Given the dropout probability $p$ and the number of neurons for this layer, we use the following algorithm to search for $K$.

**Algorithm 1:** Approximate Random Dropout Pattern Search Algorithm

| Input: | $p$, $N$, hyper parameters $\lambda_1$, $\lambda_2$ ($\lambda_1 + \lambda_2 = 1$). |
|---|---|
| Output: | A distribution $K$ 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** $|\Delta \text{loss}| \geq \text{threshold}$ **do**
   4. $d = \text{softmax}(v)$;
   5. $E_p = \langle (d, p_u) - p \rangle^2$;
   6. $E_n = \langle \frac{1}{2n} d, \log(d) \rangle$;
   7. $\text{loss} = \lambda_1 E_p + \lambda_2 E_n$;
   8. Calculate gradient of loss over $v$;
   9. Update $v$ with gradient descent;
4. **end while**
5. Return $d$ as the expected vector $K$;

We use the softmax function to derive the dropout pattern distribution vector $d$ of the current state (line 4). The constant vector $p_u$ is dropout rate of each dropout pattern (in line 2). The inner product of $d$ and $p_u$ is the dropout rate of neurons under the distribution $d$. Then we are trying to approximate the current dropout rate $\langle d, p_u \rangle$ to the expected dropout rate $p$ (line 5-9). The first term of the loss, $E_p$, denotes the squared error between expectation dropout probability of the current distribution and the actual expectation on demand (line 5). The second term of the loss, $E_n$, denotes negative information entropy, the penalty on distribution which leads to few kinds of different sub-models (line 6).

We illustrate the algorithm using Fig. 2 as an example. Given the overall training iteration $I$, after we get a distribution of dropout pattern $K = [ \frac{1}{4}, \frac{3}{4}, 0, \frac{3}{4}, \frac{1}{4} ]$, we can formulate a sequence of dropout pattern. Specifically, we can generate the frequency of dropout pattern $K' = [1, 3, 0, 2, 1, ...]$ by multiplying the probability of a dropout pattern by a scale $I$ (the overall training iteration). We need to unsqueeze (unfold) $K'$ to obtain the sequence of dropout pattern $Seq = [1, 2, 2, 4, 4, 5, ...]$. In order to obtain the maximum randomness, we shuffle the sequence of dropout pattern to $Seq = [2, 10, 4, 18, 24, ...]$. During training phase, we iteratively fetch the dropout pattern from the resulting sequence.

### 3.3 Algorithm Analysis

Dropout trains an ensemble model consisting of all sub-models that can be constructed by removing non-output units from an underlying base network. So we interpret the validity of our algorithm with the idea came from Bagging [2]. Given a number in the sequence of dropout pattern, we can equally formulate a mask $\mu$, and calculate the output with $\mu$. Moreover, the process of selecting a number in the sequence of dropout pattern can be seen as a sampling process of $\mu$ from a given probability distribution.

Consider any $\mu_i$ that can be sampled out, the error of corresponding sub-model is $\epsilon_i$, and the probability of $\mu_i$ being sampled out is $p(\mu_i)$. Assume the errors are drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$, and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Then the error made by the ensemble model is $\sum_i p(\mu_i) \epsilon_i$. The expected squared error of the ensemble model is

$$
\mathbb{E}[\sum_i p(\mu_i) \epsilon_i]^2 = \mathbb{E} \sum_i [p^2(\mu_i) \epsilon_i^2 + \sum_{j \neq i} p(\mu_i) p(\mu_j) \epsilon_i \epsilon_j]] \\
= \sum_i p^2(\mu_i) v + \sum_{i} \sum_{j \neq i} p(\mu_i) p(\mu_j) c 
$$

(1)
With the Eqn.(1) above, we expand the squared error of the ensemble model to the weighted sum of variance $v$ and covariance $c$.

With the knowledge that
\[ \sum_{i} p^2(\mu_i) + \sum_{i} \sum_{j \neq i} p(\mu_i)p(\mu_j) = (\sum_{i} p(\mu_i))^2 = 1 \] (2)

The Eqn.(2) is a trivial expansion of the squared term $(\sum_{i} p(\mu_i))^2$. And it can be achieved with the probability sum of all sub-networks $\sum_{i} p(\mu_i) = 1$.

We can simplify the equation (1) as
\[ \mathbb{E}[(\sum_{i} p(\mu_i)\epsilon_i)^2] = \lambda v + (1 - \lambda)c \] (3)

In the case where the errors are perfectly correlated s.t. $c = v$, the mean squared error reduces to $v$, which means it does not help at all. Obviously, this scenarios will not happen in our algorithm. In case that the errors are perfectly uncorrelated s.t. $c = 0$, the mean squared error is only $\lambda v$. Where the value of $\lambda$ can be restricted as small as possible due to the negative information entropy term $E_{\text{n}}$ of our loss function in the SGD-based search algorithm. So the ensemble model always perform better than each of its members.

4 Experiments

To investigate the performance of the approximate dropout algorithm we provided in Section 3, we first compare different dropout rate on multi-layer perceptron(MLPs) with a specific size in section 4.1. Then we compare different MLPs which have the same dropout rate in section 4.2. The data set we use in section 4.1 and section 4.2 is MNIST. To verify the scalability of the approximate-dropout algorithm, we do some experiments on Long short-term memory(LSTM) neural network [9] in section 4.3. The data set we use in it is a dictionary whose size is 8800. The experiments are performed by Caffe [15]. And we present speedup rate(the ratio of training time) and test accuracy of the two kinds of dropout algorithm.

4.1 Comparing different dropout ratio combinations on specific network

We construct a 4-layer MLP. The input layer is shaped according to the batch size. The output layer has 10 neurons for digit 0 to 9. Both of the hidden layers have 2048 neurons. We apply traditional dropout or proposed approximate dropout to these two hidden layers. Then we record accuracy of those two networks when dropout rate changes from (0.3, 0.3) to (0.7, 0.7). Besides, we first record the original training time when applying the traditional dropout to the network and the dropout rate changes from (0.3, 0.3) to (0.7, 0.7). Then we apply the proposed approximate dropout to the network and record the current training time when dropout rate changes from (0.3, 0.3) to (0.7, 0.7). Finally, we divide the original training time by the current training time to obtain the speedup rate. The speedup rate and the accuracy of two kinds of algorithms are shown in Fig. 5.

According to the experiment result, we can see that both Row Dropout Pattern and Tile Dropout Pattern can achieve speedup compared with the traditional Dropout algorithm. Moreover, as dropout rate changing from (0.3, 0.3) to (0.7, 0.7), the speedup rate is increasing from 1.2 to 1.77. Because as the dropout rate increases, the number of data been dropped is increasing, so the amount of computation that the GPU can save is also increasing. Meanwhile, both of the accuracies of these two Approximate Random Dropout algorithm are around 98%, which is very close to the accuracy of traditional Dropout algorithm.

4.2 Comparing different network with specific drop ratio

In this section, we compare different MLPs which have the same dropout rate (0.5, 0.5) and (0.7, 0.7). The speedup rate and the accuracy of traditional dropout and proposed approximate dropout are shown in Table 1.
Table 1: Result of Row approximate dropout and Tile approximate dropout compared with traditional dropout, iteration = 60K

| dropout ratio | network size | algorithm | original accuracy | new accuracy | speedup rate |
|---------------|--------------|-----------|-------------------|--------------|--------------|
| 0.5           | 1024x64      | ROW       | 98.39%            | 97.87%(-0.52%) | 1.18         |
|               |              | TILE      | 98.32%            | 97.78%(-0.54%) | 1.12         |
|               | 1024x1024    | ROW       | 98.50%            | 98.01%(-0.49%) | 1.19         |
|               |              | TILE      | 98.26%            | 98.02%(-0.24%) | 1.19         |
|               | 2048x2048    | ROW       | 98.44%            | 97.58%(-0.86%) | 1.45         |
|               |              | TILE      | 98.29%            | 98.08%(-0.21%) | 1.54         |
|               | 4096x4096    | ROW       | 98.37%            | 98.00%(-0.37%) | 1.48         |
|               |              | TILE      | 98.26%            | 98.07%(-0.19%) | 1.62         |
| 0.7           | 1024x64      | ROW       | 98.19%            | 97.80%(-0.39%) | 1.27         |
|               |              | TILE      | 98.30%            | 97.51%(-0.79%) | 1.19         |
|               | 1024x1024    | ROW       | 98.33%            | 98.03%(-0.30%) | 1.45         |
|               |              | TILE      | 98.26%            | 97.76%(-0.50%) | 1.22         |
|               | 2048x2048    | ROW       | 98.42%            | 97.82%(-0.60%) | 1.77         |
|               |              | TILE      | 98.23%            | 97.81%(-0.42%) | 1.60         |
|               | 4096x4096    | ROW       | 98.48%            | 98.02%(-0.46%) | 1.95         |
|               |              | TILE      | 98.29%            | 97.58%(-0.71%) | 2.16         |

The network size $1024 \times 64$ in the second column of Table 1 means the neuron numbers of the first and the second hidden layer are 1024 and 64 respectively. From Table 1, both of the Row approximate dropout and Tile approximate dropout algorithm we propose can achieve high accuracy. Concretely, the accuracy degradation is less than 1%. Besides, the speedup rate increases as the network size increases. Especially, in the case of $4096 \times 4096$ network, when dropout rate is (0.7, 0.7), both of the proposed algorithms reach a speedup rate that is around 2, which shows that our algorithm has a chance to reach higher speedup rate on a larger network.

4.3 Scaling to Large-scale Deep Learning

In this section, we evaluated the speedup rate and the model performance of traditional dropout and proposed approximate dropout, considering different sizes of LSTM and different dropout rate setting. The speedup rate and the accuracy of traditional dropout and proposed approximate dropout are shown in Table 2. Also we use the metric $NLL_{test}$ [34] to evaluate the capacity of learned language model to fit real test data. To provide a reference of such metric, we list out extra $NLL_{test}$ value for random initialized model without training, as shown in 5th row of Table 2.

At each time step, the previous word $x_t$ and hidden state $h_{t-1}$ is provided as inputs to the sequence model, in this case, a stack of LSTMs (each with 1000 hidden units), which is used to learn a language model. $K$ is the number of words in the vocabulary, we set $K=8800$ here, plus one additional entry.
for the <BEOS> (beginning or ending of sequence), which is taken as previous word of sentence beginning or ending mark of a sentence.

Table 2: Result of Row approximate dropout and Tile approximate dropout applying to LSTM

| iter | 110K |
|------|------|
| layer number | 2 | 3 |
| drop ratio | 0.3 | 0.5 | 0.7 | 0.3 | 0.5 | 0.7 |
| NLL | | | | | | |
| random | 30.7 | 30.7 | 30.6 | 30.8 | 30.8 | 31.1 |
| row | 35.7 | 35.6 | 35.6 | 32.8 | 32.6 | 32.3 |
| tile | 35.6 | 35.7 | 35.6 | 32.9 | 32.6 | 32.2 |
| accuracy | | | | | | |
| original | 47.6% | 47.7% | 46.2% | 47.9% | 47.3% | 45.9% |
| row | 46.8% | 46.6% | 44.7% | 46.9% | 46.0% | 44.5% |
| tile | 46.9% | 46.3% | 44.9% | 47.2% | 46.5% | 44.4% |
| speedup | | | | | | |
| row | 1.16 | 1.23 | 1.31 | 1.18 | 1.43 | 1.49 |
| tile | 1.15 | 1.22 | 1.30 | 1.18 | 1.47 | 1.53 |

The neuron numbers for each LSTM layer is identically 1000 in our experiment. As shown in Table 2, both of the Row approximate dropout and Tile approximate dropout algorithm we propose can achieve close accuracy and $NLL_{test}$ value in comparison with original model. Concretely, the accuracy degradation is less than 1.5%. When dropout rate is increasing, the speedup rate increases too. Furthermore, the speedup rate increases when the network size increases (from 2 layer LSTM to 3 layer LSTM), which also shows that the proposed approximate algorithm has a chance to reach higher speedup rate on a larger network.

To further verify the effectiveness of proposed approximate dropout algorithm, we set dropout rate $= 0.5$. Then we record the row approximate dropout training process and the tile approximate dropout training process in 3 layer LSTM, as shown in Fig. 6.

The red curve records our approximate dropout training process; the blue one records the traditional dropout. The convergence speed of our approximate dropout is quicker than the traditional dropout; the speedup rate is approximately 1.42X in row approximate dropout and 1.47X in tile approximate dropout. Moreover, the red curve is more smooth than the blue one, which indicates our approximate dropout algorithm is helpful for the training process.

Figure 6: Train log of row approximate random dropout and tile approximate random dropout.

5 Conclusion

In this work, we propose a novel approach to eliminate the unnecessary multiplication and data access by replacing the traditional random dropout mechanism with an approximate random dropout patterns. The two class of dropout patterns can avoid the divergence issue in GPU and thus gain significant improvement on the performance and energy-efficiency. The proposed SGD-based search algorithm can generate sufficient dropout patterns (sub-models) to guarantee the convergence and
accuracy of the models. The efficiency of the proposed technique is proved mathematically and experimentally. In general, the training time can be reduced by half.

References

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

[2] Leo Breiman. Bagging predictors. Kluwer Academic Publishers, 1996.

[3] Francois Chollet. Xception: Deep learning with depthwise separable convolutions. In IEEE Conference on Computer Vision and Pattern Recognition, pages 1800–1807, 2017.

[4] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew Senior, and Paul Tucker. Large scale distributed deep networks. In International Conference on Neural Information Processing Systems, pages 1223–1231, 2012.

[5] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, and Geng Yuan. Circtnn: Accelerating and compressing deep neural networks using block-circulantweight matrices. 2017.

[6] Michal Čerňanský. 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, pages 381–390, 2009.

[7] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. Computer Science, 2014.

[8] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. 2017.

[9] Alex Graves. Long Short-Term Memory. Springer Berlin Heidelberg, 2012.

[10] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In ICLR, 2016.

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.

[12] Qinghao Hu, Peisong Wang, and Jian Cheng. From hashing to cnns: Training binaryweight networks via hashing. 2018.

[13] Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. Journal of Asian Studies, 62(3):952–953, 2015.

[14] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. Computer Science, 4(4):XIII, 2014.

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

[16] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oguz H. Elibol, Scott Gray, Stewart Hall, and Luke Hornof. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In Neural Information Processing Systems, 2017.

[17] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. 2017.
[18] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. 2017.

[19] Yajie Miao, Jinyu Li, Yongqiang Wang, Shi Xiong Zhang, and Yifan Gong. Simplifying long short-term memory acoustic models for fast training and decoding. In IEEE International Conference on Acoustics, Speech and Signal Processing, pages 2284–2288, 2016.

[20] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In International Conference on Frontiers in Handwriting Recognition, pages 285–290, 2014.

[21] Siddhartha Puri. Training convolutional neural networks on graphics processing units, 2010.

[22] Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. 2016.

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. 2018.

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

[25] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meprop: Sparsified backpropagation for accelerated deep learning with reduced overfitting. 2017.

[26] Li Wan, Matthew D Zeiler, Sixin Zhang, Yann Lecun, and Rob Fergus. Regularization of neural networks using dropconnect. In International Conference on Machine Learning, pages 1058–1066, 2013.

[27] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. 2016.

[28] Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhao Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. 2017.

[29] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. 2017.

[30] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. 2017.

[31] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. In International Joint Conference on Artificial Intelligence, pages 2350–2356, 2016.

[32] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. 2017.

[33] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. 2017.

[34] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. 2018.