Accelerated CNN Training Through Gradient Approximation

Ziheng Wang
Department of EECS, Massachusetts Institute of Technology
Cambridge, MA, USA
ziheng@mit.edu

Sree Harsha Nelaturu
Department of ECE, SRMIST
Kattankulathur, Chennai, India
sreeharsha_murali@srmuniv.edu.in

Saman Amarasinghe
CSAIL, Massachusetts Institute of Technology
Cambridge, MA, USA
saman@mit.edu

Abstract

Training deep convolutional neural networks such as VGG and ResNet by gradient descent is an expensive exercise requiring specialized hardware such as GPUs. Recent works have examined the possibility of approximating the gradient computation while maintaining the same convergence properties. While promising, the approximations only work on relatively small datasets such as MNIST. They also fail to achieve real wall-clock speedups due to lack of efficient GPU implementations of the proposed approximation methods. In this work, we explore three alternative methods to approximate gradients, with an efficient GPU kernel implementation for one of them. We achieve wall-clock speedup with ResNet-20 and VGG-19 on the CIFAR-10 dataset upwards of 7 percent, with a minimal loss in validation accuracy.

Keywords gradient approximation, accelerated training, CNN

ACM Reference Format:
Ziheng Wang, Sree Harsha Nelaturu, and Saman Amarasinghe. 2019. Accelerated CNN Training Through Gradient Approximation. In *Proceedings of Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC$\hat{2}$, 4th Edition).* ACM, New York, NY, USA, 5 pages.

1 Introduction

Deep convolutional neural networks (CNN) are now arguably the most popular computer vision algorithms. Models such as VGG [15] and ResNet [7] are widely used. However, these models contain up to hundreds of millions of parameters, resulting in high memory footprint, long inference time and even longer training time.

The memory footprint and inference time of deep CNNs directly translate to application size and latency in production. Popular techniques based on model sparsification are able to deliver orders of magnitude reduction in the number of parameters in the network [6]. However, the training of these deep CNNs is still a lengthy and expensive process. Recent research has attempted to address the training time issue by demonstrating effective training on large scale computing clusters consisting of thousands of GPUs [21]. However, these computing clusters are still extremely expensive and labor-intensive to set up or maintain. An alternative to using large computing clusters is to accelerate the computations of the gradients themselves. There has been recent effort to approximate the gradient computation [1, 16, 17, 19]. Other recent works have also suggested that the exact gradient might not be necessary for efficient training of deep neural networks. Studies have shown that only the sign of the gradient is necessary for efficient back propagation [20]. Surprisingly, even random gradients can be used to efficiently train neural networks. [10, 12] However, these methods have not been shown to result in real wall-clock speedups in training for deep CNNs either due to a lack of efficient GPU implementation or because the methods only apply to fully connected networks.

In this work, we hypothesize that we can extend gradient approximation methods to deep CNNs to speed up gradient computations in the training process. We hypothesize that we can apply these approximations to only a subset of the layers and maintain the validation accuracy of the trained network. We validate our hypotheses on three deep CNNs (2-layer CNN [9], ResNet-20 [7], VGG-19 [15]) on CIFAR-10.

We summarize our contributions as follows:

- We present three gradient approximation methods for training deep CNNs, along with an efficient GPU implementations for one of them.
- We explore the application of these methods to deep CNNs and show that they allow for training convergence with minimal validation accuracy loss.

2 Approximation Methods

In a forward-backward pass of a deep CNN during training, a convolutional layer requires three convolution operations: one for forward propagation and two for backward propagation, as demonstrated in Figure 1. We approximate the convolution operation which calculates the gradients of the filter values, which constitutes roughly a third of the computational time. We aim to apply the approximation a quarter of the time across layers/batches. This leads to a theoretical maximum speedup of around 8 percent.

2.1 Zero Gradient

The first method passes back zero as the weight gradient of a chosen layer for a chosen batch. If done for every training batch, it effectively freezes the filter weights.

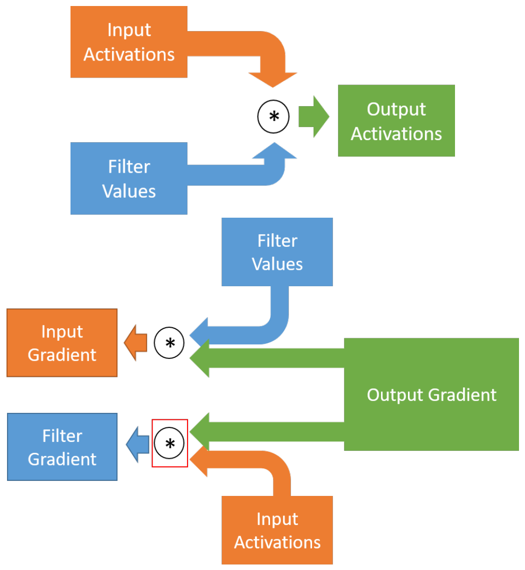

Figure 1. Forward and backward propagation through a convolutional layer during training. Asterisks indicate convolution operations and the operation in the red box is the one we approximate.

2.2 Random Gradient

The second method passes back random numbers sampled from a normal distribution with mean 0 and standard deviation $\frac{1}{128}$ (inverse of batch size) as the weight gradient of a chosen layer for a chosen batch. Different values in the weight gradient are chosen independently. Importantly, this is different from the random feedback alignment method discussed in [10] and [12] as we regenerate the random numbers every training batch. We implement this using tf.py_func, where np.random.normal is used to generate the random values. This approach is extremely inefficient, though surprisingly faster than a naive cuRAND implementation in a custom tensorflow operation for most input cases. We are working on a more efficient implementation.

2.3 Approximated Gradient

The third method we employ is based on the top-k selection algorithms popular in literature [19]. In the gradient computation for a filter in a convolutional layer, only the largest-magnitude gradient value is retained for each output channel and each batch element. They are scaled according to the sum of the gradients in their respective output channels so that the gradient estimate is unbiased, similar to the approach employed in [18]. All other gradients are set to zero. This results in a sparsity ratio of $1 - \frac{1}{HW}$, where H and W are the height and width of the output hidden layer. The filter gradient is then calculated from this sparse version of the output gradient tensor with the saved input activations from the forward pass. The algorithm can be trivially modified to admit the

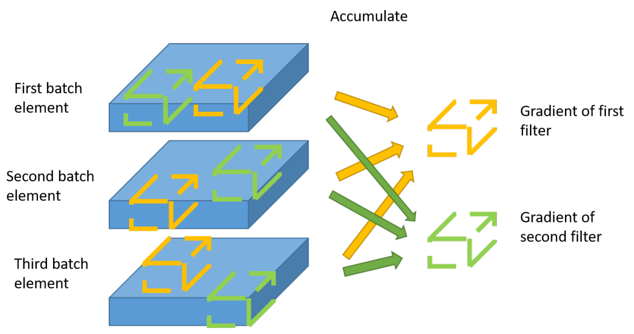

Figure 2. The approximation algorithm illustrated for an example with two filters and three input elements. For each filter, we extract a patch from each batch element's input activations and accumulate the patches.

top-k magnitude gradient values with an adjustment of the scaling parameter, a direction of future research. Similar to the random gradient method, we find that we need to scale our approximated gradient by a factor proportional to the batch size for effective training. In the experiments here, we scale them by $\frac{1}{128}$.

2.4 Efficient GPU Implementation

A major contribution of this work is an implementation of the approximated gradient method in CUDA. This is critical to achieve actual wall-clock training speedups. A naive Tensorflow implementation using tf.image.extract_glimpse does not use the GPU and results in significantly slower training time. Efficient GPU implementations for dense convolutions frequently use matrix lowering or transforms such as FFT or Winograd. Sparse convolutions, on the other hand, is frequently implemented directly on CPU or GPU [3, 13]. Here we interpret the sparse convolution in the calculation of the filter gradient as a patch extraction procedure, as demonstrated in Figure 2. The nonzeros in the output specifies patches in the input that need to be extracted and averaged.

Our kernel implementation expects input activations in $NHWC_i$ format and output gradients in NC_oHW format. It produces the output gradient in C_oKKC_i format. In $NHWC$ format, GPU global memory accesses from the patch extractions can be efficiently coalesced across the channel dimension, which is typically a multiple of 8. Each thread block is assigned to process several batch elements for a fixed output channel. Each thread block first computes the indices and values of the nonzero weight values from the output gradients. Then, they extract the corresponding patches from the input activations and accumulate them to the result.

We benchmark the performance of our code against NVIDIA cuDNN v7.4.2 library apis. Approaches such as cuSPARSE have been demonstrated to be less effective in a sparse convolution setting and are not pursued here [3]. All timing metrics are obtained on a workstation with a Titan-Xp GPU and 8 Intel Xeon CPUs at 3.60GHz.

All training experiments are conducted in $NCHW$ format, the preferred data layout of cuDNN. As a result, we incur a data transpose overhead of the input activations from $NCHW$ to $NHWC$. In addition, we also incur a slight data transpose overhead of the filter gradient from C_oKKC_i to KKC_iC_o.

3 Evaluation

We test our approach on three common neural network architectures (2-layer CNN [9], VGG-19 [15] and ResNet-20[7]) on the CIFAR-10 dataset. The local response normalization in the 2-layer CNN is replaced by the more modern batch normalization method [8]. For all three networks, we aim to use the approximation methods 25 percent of the time. In this work, we test all three approximation methods separately and do not combine. On the 2-layer CNN, we apply the selected approximation method to the second convolutional layer every other training batch. On VGG-19 and ResNet-20, we apply the selected approximation method to every fourth convolutional layer every training batch, starting from the second convolutional layer. We start from the second layer because recent work has shown that approximating the first convolutional layer is difficult [1]. This results in four approximated layers for VGG-19 and five approximated layers for ResNet-20. We refer to a specification of where and when to apply the approximation an **approximation schedule**. For the ResNet-20 model, we train a baseline ResNet-14 model as well. Training a smaller model is typically done in practice when training time is of concern. Ideally, our approximation methods to train the larger ResNet-20 model should result in higher validation accuracy than the ResNet-14 model.

3.1 Performance Comparisons

We compare the performance of our GPU kernel for the approximated gradient method with the full gradient computation for the weight filter as implemented in cuDNN v7.4.2. cuDNN offers state-of-the-art performance in dense gradient computation and is used in almost every deep learning library. Here we demonstrate that our gradient approximation method does yield an efficient GPU implementation that can lead to actual speedups compared to cuDNN.

We present timing comparisons for a few select input cases encountered in the network architectures used in this work in Table 1. We aggregate the two data transpose overheads of the input activations and the filter gradients. We make three observations.

Firstly, in most cases, the gradient approximation, including data transposition, is at least three times as fast as the cuDNN baseline. Secondly, we observe that cuDNN timing scales with the number of input channels times the height and width of the hidden layer, whereas our approximation kernel timing scales with the number of input channels alone. This is expected from the nature of the computations involved: the performance bottleneck of our kernel is the memory intensive patch extractions, the sizes of which scale with the number of input channels times filter size. Thirdly, we observe that in many cases, the data transposition overhead is over fifty percent of the kernel time, suggesting that our implementation can be further improved by fusing the data transpose into the kernel as in SBNet [14]. This is left for future work.

Input Channels	Output Channels	Height / Width	cuDNN	CUDA Kernel	Transpose Overhead	Approx. Total	Speed-up Factor
256	256	8	0.89	0.46	0.05	0.51	1.7
128	128	16	0.89	0.20	0.08	0.28	3.2
512	512	4	1.00	1.25	0.07	1.32	0.8
512	512	2	13.6	1.22	0.08	1.30	10.4
64	64	32	1.51	0.13	0.16	0.29	5.2
16	16	32	0.38	0.04	0.05	0.09	4.2
32	32	16	0.20	0.03	0.02	0.05	4.0
64	64	8	0.22	0.04	0.02	0.06	3.7

Table 1. Performance comparisons. All timing statistics in microseconds. Approx. total column is the sum of the CUDA Kernel time and the transpose overhead.

	Speed-up	Accuracy Loss
Full Gradient	1x	0
Zero Gradient	-5.7%	0.2%
Random Gradient	-38%	-0.8%
Approx. Gradient	-6.6%	1.0%

Table 2. Training speedup and validation accuracy loss for the approximation methods on 2-layer CNN. Negative speedup indicates a slowdown.

3.2 Speedup-Accuracy Tradeoffs

Here, we present the training wall-clock speedups achieved for each network and approximation method. We compare the speedups against the validation accuracy loss, measured from the best validation accuracy achieved during training. Validation accuracy was calculated every ten epochs. As aforementioned, the random gradient implementation is quite inefficient and is pending future work. The speedup takes into account the overhead of defining a custom operation in Tensorflow, as well as the significant overhead of switching gradient computation on global training step. For the 2-layer CNN, we are unable to achieve wall-clock speedup for all approximation methods, even the zero gradient one, because of this overhead. (Table 2) However, all approximation methods achieve little validation accuracy loss. The random gradient method even outperforms full gradient computation by 0.8%.

For ResNet-20, the approximation schedule we choose does not involve switching gradient computations. We avoid the switching overhead and can achieve speedups for both the zero gradient method and the approximated gradient method.

	Speed-up	Accuracy Loss
Full Gradient	1x	0
Zero Gradient	7.1%	1.5%
Random Gradient	-1.4%	1.8%
Approx. Gradient	3.5%	0.75%
ResNet-14	22%	1.0%

Table 3. Training speedup and validation accuracy loss for the approximation methods on ResNet-20. Negative speedup indicates a slowdown.

	Speed-up	Accuracy Loss
Full Gradient	1x	0
Zero Gradient	7.3%	1.0%
Random Gradient	Over -100%	0.1%
Approx. Gradient	1.5%	0.7%

Table 4. Training speedup and validation accuracy loss for the approximation methods on VGG-19. Negative speedup indicates a slowdown.

As shown in Table 3, the zero gradient method achieves roughly a third of the speedup compared to training the baseline ResNet-14 model. The approximated gradient method also achieves a 3.5% wall-clock speedup, and is the only method to suffer less accuracy loss than just using a smaller ResNet-14. In the following section, we demonstrate that with other approximation schedules, the approximated gradient method can achieve as little as 0.1% accuracy loss.

For VGG-19, despite being quicker to converge, the approximation methods all have worse validation accuracy than the baseline method. (Table 4) The best approximation method appears to be the random gradient method, though it is extremely slow due to our inefficient implementation in Tensorflow. The other two methods also achieve high validation accuracies, with the approximated gradient method slightly better than the zero gradient method. Both methods are able to achieve speedups in training.

3.3 Robustness to Approximation Schedule

Here, we explore two new approximation schedules for ResNet-20, keeping the total proportion of the time we apply the approximation to 25 percent. We will refer to the approximation schedule presented in the section above as schedule 1. Schedule 2 applies the selected approximation method every other layer for every other batch. Schedule 3 applies the selected approximation method every layer for every fourth batch. We also present the baseline result of the ResNet-14 model.

As we can see from Table 5, under schedules 2 and 3, both the zero gradient and the approximated gradient method perform well. In fact, for the approximated gradient and the zero gradient methods the validation accuracy loss is smaller than schedule 1. Indeed, in schedule 3, the approximated gradient's best validation accuracy is within 0.1% of that of the full gradient computation. The random gradient method's validation accuracy is in now line with its poor loss curve for these two approximation schedules. This suggests that the random gradient method does not work well for ResNet-20 architecture.

	Schedule 1	Schedule 2	Schedule 3
Full Gradient	91.8%		
Zero Gradient	90.3%	91.6%	91.1%
Random Gradient	90.0%	88.8%	87.9%
Approximated Gradient	91.1%	91.6%	91.7%
Baseline, ResNet-14	90.8%		

Table 5. Validation accuracy for different approximation schedules on ResNet-20. Schedule 1 is the same as presented above.

4 Discussion and Conclusion

While research on accelerating deep learning inference abounds, there is relatively limited work focused on accelerating the training process. Recent works such as PruneTrain prune the neural network in training, but suffers quite serious loss in validation accuracy [11]. Approaches such as DropBack [4] and MeProp [16, 19] show that approximated gradient are sufficient in successfully training neural networks but don't yet offer real wall-clock speedups. In this work, we study three alternative gradient approximation methods.

We are surprised by the consistent strong performance of the zero gradient method. For ResNet-20, for two of the three approximation schedules tested, the validation accuracy loss is better than that of a smaller baseline network. Its performance is also satisfactory on VGG-19 as well as the 2-layer CNN. It admits an extremely fast implementation that delivers consistent speedups. This points to a simple way to potentially boost training speed in deep neural networks, while maintaining their performance advantage over shallower alternatives.

We also demonstrate that random gradient methods can train deep neural networks to good validation accuracy. For the 2-layer CNN and VGG-19, this method leads to the least validation accuracy loss of all three approximation methods. However, its validation accuracy serious lags other methods on ResNet-20. Naive feedback alignment, where the random gradient signal is fixed before training starts, has been shown to be difficult to extend to deep convolutional architectures [2, 5] . We show here that if the random gradients are newly generated every batch and applied to a subset of layers, they can be used to train deep neural networks to convergence. Interestingly, generating new random gradients every batch effectively abolishes any kind of possible "alignment" in the network, calling for a new explanation of why the network converges. Evidently, this method holds the potential for an extremely efficient implementation, something we are currently working on.

Finally, we present a gradient approximation method with an efficient GPU implementation. Our approximation method is consistent in terms of validation accuracy across different network architectures and approximation schedules. Although the training wall clock time speedup isn't large, the validation accuracy loss is also small. We wish to re-emphasize here the small validation accuracy difference observed between the baseline ResNet-14 and ResNet-20, leading us to believe that novel training speed-up methods must incur minimal validation accuracy loss to be more practical than simply training a smaller network.

In conclusion, we show that we can "fool" deep neural networks into training properly while supplying it only very minimal gradient information on select layers. The approximation methods are simple and robust, holding the promise to accelerate the lengthy training process for state-of-the-art deep CNNs.

5 Future Work

Besides those already mentioned, there are several more interesting directions of future work. One direction is predicting the validation accuracy loss that a neural network would suffer from a particular approximation schedule. With such a predictor, we can optimize for the fastest approximation schedule while constraining the final validation accuracy loss before the training run. This would remove the need for arbitrarily selecting an approximation ratio like we did here. We can also examine the effects of mingling different approximation methods and integrating existing methods such as PruneTrain and Dropback [4, 11]. Another direction is approximating the gradient of the hidden activations, as is done in meProp [16]. Finally, we are working on integrating this approach into a distributed training setting, where the approximation schedule is now 3-dimensional (machine, layer, batch). This approach would be crucial for the approximation methods to work with larger scale datasets such as ImageNet, thus potentially allowing for wall-clock speed-up in large scale training.

Acknowledgments

We thank Ajay Brahmakshatriya for helpful advice.

References

[1] Menachem Adelman and Mark Silberstein. 2018. Faster Neural Network Training with Approximate Tensor Operations. *arXiv preprint arXiv:1805.08079* (2018).

[2] Sergey Bartunov, Adam Santoro, Blake Richards, Luke Marris, Geoffrey E Hinton, and Timothy Lillicrap. 2018. Assessing the scalability of biologically-motivated deep learning algorithms and architectures. In *Advances in Neural Information Processing Systems*. 9368–9378.

[3] Xuhao Chen. 2018. Escort: Efficient sparse convolutional neural networks on gpus. *arXiv preprint arXiv:1802.10280* (2018).

[4] Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2018. DropBack: Continuous Pruning During Training. *arXiv preprint arXiv:1806.06949* (2018).

[5] Donghyeon Han and Hoi-jun Yoo. 2019. Efficient Convolutional Neural Network Training with Direct Feedback Alignment. *arXiv preprint arXiv:1901.01986* (2019).

[6] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[8] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[9] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[10] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. 2016. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications* 7 (2016), 13276.

[11] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Mattan Erez, and Sujay Shanghavi. 2019. PruneTrain: Gradual Structured Pruning from Scratch for Faster Neural Network Training. *arXiv preprint arXiv:1901.09290* (2019).

[12] Arild Nøkland. 2016. Direct feedback alignment provides learning in deep neural networks. In *Advances in neural information processing systems*. 1037–1045.

[13] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409* (2016).

[14] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. 2018. Sbnet: Sparse blocks network for fast inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8711–8720.

[15] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[16] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. 2017. meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 3299–3308.

[17] Xu Sun, Xuancheng Ren, Shuming Ma, Bingzhen Wei, Wei Li, Jingjing Xu, Houfeng Wang, and Yi Zhang. 2018. Training simplification and model simplification for deep learning: A minimal effort back propagation method. *IEEE Transactions on Knowledge and Data Engineering* (2018).

[18] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1306–1316.

[19] Bingzhen Wei, Xu Sun, Xuancheng Ren, and Jingjing Xu. 2017. Minimal effort back propagation for convolutional neural networks. *arXiv preprint arXiv:1709.05804* (2017).

[20] Will Xiao, Honglin Chen, Qianli Liao, and Tomaso Poggio. 2018. Biologically-plausible learning algorithms can scale to large datasets. *arXiv preprint arXiv:1811.03567* (2018).

[21] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 1.