Time and Memory Efficient Variance Reduced Optimization and Sampling for Deep Learning

Ayoub El Hanchi
McGill University

Abstract

Variance reduced optimization and sampling algorithms are known to enjoy faster convergence rates compared to standard stochastic optimization and sampling algorithms. The application of these algorithms to deep learning problems has been limited due to their prohibitive memory cost and the difficulty to obtain per-example gradients in most deep learning software platforms. We propose a new set of methods to efficiently compute, store, and manipulate the per-example gradients. Using these methods, we evaluate the effectiveness of variance reduced algorithms on standard image classification tasks. We implemented these ideas in PyTorch. The code can be found here and here.

1 Introduction

Stochastic gradient descent (SGD) and stochastic gradient Langevin dynamics (SGLD) are the two main algorithms used in the optimization and sampling of deep neural network models. These algorithms replace the full gradient by an estimate computed on small mini-batches of the data to achieve a sub-linear per iteration time complexity in the number of data points. The variance of the gradient estimator plays a crucial role in the quality of the solutions that these algorithms arrive at.

Variance reduction techniques such as SAGA [4], SAG [12], and SVRG [8] attempt to reduce this variance. When applied to convex problems, they are shown both theoretically and empirically to outperform the regular SGD and SGLD [2, 5]. Not surprisingly, the analysis of these algorithms becomes more difficult in the non-convex case, with more modest theoretical improvements [11].

A significant drawback of these variance reduction techniques is their high memory cost, usually requiring the storage of previous per-example gradients, leading to an $O(Nd)$ memory usage, where N is the number of data points, and d the dimension of the parameter vector. For modern neural networks with millions of parameters, this is not feasible.

In this work, we propose a new set of techniques to reduce the memory cost of these algorithms. We leverage the linear structures present within neural networks to propose an efficient mechanism to construct and store the per-example gradients. We also propose the use of a mini-batch sampling strategy that reduce the memory cost of these algorithms. Finally, we study empirically the effect of variance reduction on the training of deep neural networks. To our knowledge only [3] study variance reduced optimization for deep neural networks, although they circumvent the memory problem treated here by recomputing the necessary gradients when needed.

2 Obtaining the per-example gradients

Obtaining per-example gradients from modern automatic differentiation engines is difficult. Aside from JAX [1], no other engine to my knowledge offers a vectorized version of this computation. The naive way of requesting each gradient separately requires as many forward and backward passes as the mini-batch size. The overhead from this computation is large enough to offset any potential

gains from variance reduction, and is therefore a serious limitation in the implementation of these algorithms.

In this section we show how to reconstruct the per-example gradients for linear and convolutional layers using only the single forward/backward pass needed for the computation of the gradient of the full mini-batch.

Throughout this section, let $\mathcal{L}$ be the loss function or the negative log-posterior:

$$\mathcal{L}(\{x_i\}_{i=1}^N, \theta) = \sum_{i=1}^{N} l_i(x_i, \theta)$$

Let $S \subseteq [N]$ denote the mini-batch and let $\mathcal{L}_S$ denote the estimate of the loss computed using S:

$$\mathcal{L}_S = \sum_{i \in S} l_i(x_i, \theta)$$

For a matrix X, and indices I, let $X[I, :]$ denote the rows corresponding to the indices in I, and let $X[:, I]$ denote the columns corresponding to the indices in I.

2.1 Linear layers

Denote by $A_S \in \mathbb{R}^{d_{in} \times |S|}$ the input of the linear layer. Let $W \in \mathbb{R}^{d_{out} \times d_{in}}$ be the weight matrix of the layer and $b \in \mathbb{R}^{d_{out}}$ the bias of the linear layer. Then the output of the linear layer is:

$$Z_S := W A_S + b$$

and for any subset $I \subseteq S$:

$$\begin{aligned} \frac{\partial \mathcal{L}_I}{\partial W} &= \frac{\partial \mathcal{L}_\mathcal{S}}{\partial Z_S}[:, I] A_S^T[I, :] \\ \frac{\partial \mathcal{L}_I}{\partial b} &= \frac{\partial \mathcal{L}_\mathcal{S}}{\partial Z_S}[:, I]\ 1_{|I|} \end{aligned} \tag{1}$$

where $1_{|I|}$ is the vector of ones of length $|I|$. In particular when $I = \{i\}$ is a single index we have:

$$\begin{aligned} \frac{\partial l_i}{\partial W} &= \frac{\partial \mathcal{L}_\mathcal{S}}{\partial Z_S}[:, i] A_S^T[i, :] \\ \frac{\partial l_i}{\partial b} &= \frac{\partial \mathcal{L}_\mathcal{S}}{\partial Z_S}[:, i] \end{aligned} \tag{2}$$

Therefore to reconstruct the per-example gradients, it is enough to cache A_S in the forward pass, intercept the gradient of the outputs Z_S during the backward pass, and use (2) to compute the needed gradients.

2.2 Convolutional layers

To obtain the individual gradients for a convolutional layer, we first rewrite it as a linear layer and use the above formulas. Denote by $A_S \in \mathbb{R}^{h_{in} \times w_{in} \times c_{in} \times |S|}$ the input of the convolutional layer. Let $W \in \mathbb{R}^{c_{out} \times k_h \times k_w \times c_{in}}$ be the layer's weights, and let $b \in \mathbb{R}^{c_{out}}$ be the bias of the convolutional layer. We start by making the following transformations:

- Decompose $A_S[:, :, :, i]$ into $\{a_i^j\}_{j=1}^{h_{out} w_{out}}$, the components of shape $k_h \times k_w \times c_{in}$ that are transformed using W and b to produce the output of the i^{th} data point.
- Vectorize the $\{a_i^j\}_{j=1}^{h_{out} w_{out}}$ to produce $\{vec(a_i^j)\}_{j=1}^{h_{out} w_{out}}$ of length $k_h k_w c_{in}$.
- Do this for all $i \in S$, and collect the $\{\{vec(a_i^j)\}_{j=1}^{h_{out} w_{out}}\}_{i \in S}$ in the columns of a matrix $A_S^{'}$.
- Vectorize the last three dimensions of W to produce $W^{'}$ of shape $c_{out} \times k_h k_w c_{in}$.

With these transformations, we can write the output of the convolutional layer as:

$$Z_S := W^{'} A_S^{'} + b$$

Which brings us back to the case of linear layers, but with one important difference. Each data point now produces $h_{out}w_{out}$ vectors that are transformed by W and b, instead of just one in the case of a linear layer. The gradient due to the i^{th} data point is therefore the sum of the gradients due to each a_i^j in $\{a_i^j\}_{j=1}^{h_{out}w_{out}}$, and similarly for subsets of S.

3 Storing the per-example gradients

The results of the previous section allow us to efficiently construct the per-example gradients using only one forward and backward pass. We now turn to the problem of storing them.

3.1 Linear layers

The naive way of storing the per-example gradients would be to explicitly construct them and store them. This requires $N \times d_{in} \times d_{out}$ memory, which is not doable for all but small layers. From formulas (1) and (2), we see that to store the per-example gradients, it is enough to store $\frac{\partial \mathcal{L}}{\partial Z}$ and A, and update them as new gradients are evaluated. This only requires $N \times d_{in}$ for A and $N \times d_{out}$ for $\frac{\partial \mathcal{L}}{\partial Z}$ for a total of $N \times (d_{in} + d_{out})$.

3.2 Convolutional layers

We might be tempted to use the same trick above for convolutional layers, but this is not a very good idea since typically $\frac{\partial \mathcal{L}}{\partial Z}$ and A are the most memory hungry components of such layers. The naive way of explicitly constructing and storing the gradients is the better option in this case.

To better understand this difference from linear layers, we can see from (2) that each per-example gradient in a linear layer is a rank-one matrix, which is storable using only two vectors of sizes d_{in} and d_{out}. In convolutional layers however, each per-example gradient is a sum of $h_{out}w_{out}$ outer products as mentioned in the previous section. Typically $h_{out}w_{out} \gg c_{out}$, so that this outer product decomposition requires significantly more memory than the explicit construction and storage of the matrix.

3.3 SVD compression

The discussion of the previous paragraph suggests an alternative way of storing the per-example gradients in convolutional layers. If we cannot store the exact gradient, perhaps we can store a low rank approximation of it that requires less memory.

In particular, we can look for a low rank matrix that best approximates the gradient with respect to an appropriate matrix norm. This problem admits a closed form solution if we use the spectral norm or the Frobenius norm. The result is known as the Eckart–Young–Mirsky theorem, and its solution is easily obtained from the singular value decomposition of the original matrix. Furthermore, we can use the outer-product decomposition provided by the singular value decomposition to efficiently store this low-rank approximation.

If we let r be the rank of our approximation, then this method reduces the memory cost from $N \times c_{out} \times c_{in} \times k_h \times k_w$ to $N \times r \times (c_{out} + (c_{in} \times k_h \times k_w))$, which can be significant for small enough r. Computing the singular value decomposition is the price to pay for this reduction in memory usage, but efficient randomized algorithms for finding the first few largest singular values and their corresponding singular vectors are known. See [7] for more details.

3.4 Partition sampling

Up until now we have assumed that per-example gradients are necessary, and looked for ways to generate them and store them. Another way to reduce the memory cost of variance reduction algorithms is by constraining the distribution of the mini-batches. In particular, we can first partition the dataset into M disjoint subsets of equal size $\frac{N}{M}$, and then treat each such subset as a single

Table 1: The memory cost in gigabytes (GB) when using single precision of the variance reduction algorithms for standard deep neural networks. The 'Normal' column refers to the use of the storing strategy for linear layers described in section 3.1. Partition-128 refers to the partitioning of the dataset into disjoint subsets of size 128.

Network	Naive	Normal	rank-1 approx	Partition-128	rank-1 approx + Partition-128
LeNet-5	246	13.5	4.15	1.92	0.03
VGG16	$5.53 \cdot 10^5$	$5.90 \cdot 10^4$	337	630	2.63
Resnet34	$18.7 \cdot 10^5$	$8.45 \cdot 10^4$	421	764	3.29

example. Using this sampling strategy, it is enough to store the gradients for each of the $\frac{N}{M}$ subsets. This reduces the memory cost of these algorithm by a factor of $\frac{1}{M}$, but can come at the cost of lower variance reduction. See [6] for an analysis of SAGA using partition sampling.

4 Implementation and Experiments

We implemented the methods described above in PyTorch [10] in a package we called "torch_vr". The implementation can be found here. We also wrote a package we called "torch_sample" in the same flavour as torch.optim which implements both Metropolis adjusted and unadjusted Markov chain Monte Carlo algorithms for sampling. This implementation can be found here.

To test our implementation, we first compared the performance of the SAGA algorithm to that of regular SGD, both the unaccelerated version and the accelerated one on a softmax regression model applied on MNIST, CIFAR10, and CIFAR100. The results are shown in figure 1. The same initialization and step size was used for all algorithms for each dataset. The variance reduction effect is clearly seen, resulting in faster convergence for both SAGA and accelerated SAGA compared to SGD and accelerated SGD.

For each of these datasets, we also trained a small convolutional neural network with six convolutional layers and relu activations. To exhibit the variance reduction effect, we first ran the Adam optimizer [9] to reach regions close enough to an optimum so that the variance reduction effect can be clearly seen. Here again we used the same initialization and step size for all algorithms. The results are shown in figure 2. For the CIFAR10 and CIFAR100 datasets, we see that the variance reduction effect is noticeable, resulting in faster convergence. For the MNIST dataset however, SGD and SAGA seem to be producing the same results, both in the accelerated and the unaccelerated case. This is explained by the fact that the model is overparametrized, allowing it to fit the training data perfectly. Around the optimum, all the per-example gradients are almost zero, and the SAGA update reduces to the SGD update, resulting in the same convergence.

5 Conclusion

We presented a new set of techniques to adapt the variance reduced optimization and sampling algorithms to deep learning models. We used the linear structures present within neural networks to develop efficient ways to construct, store, and manipulate the per-example gradients, and showed the effectiveness of variance reduction in the training of a small convolutional neural network on standard image classification tasks. We believe that these techniques can help in the training of very large networks and reach state of the art performance and we are currently running more experiments to support this.

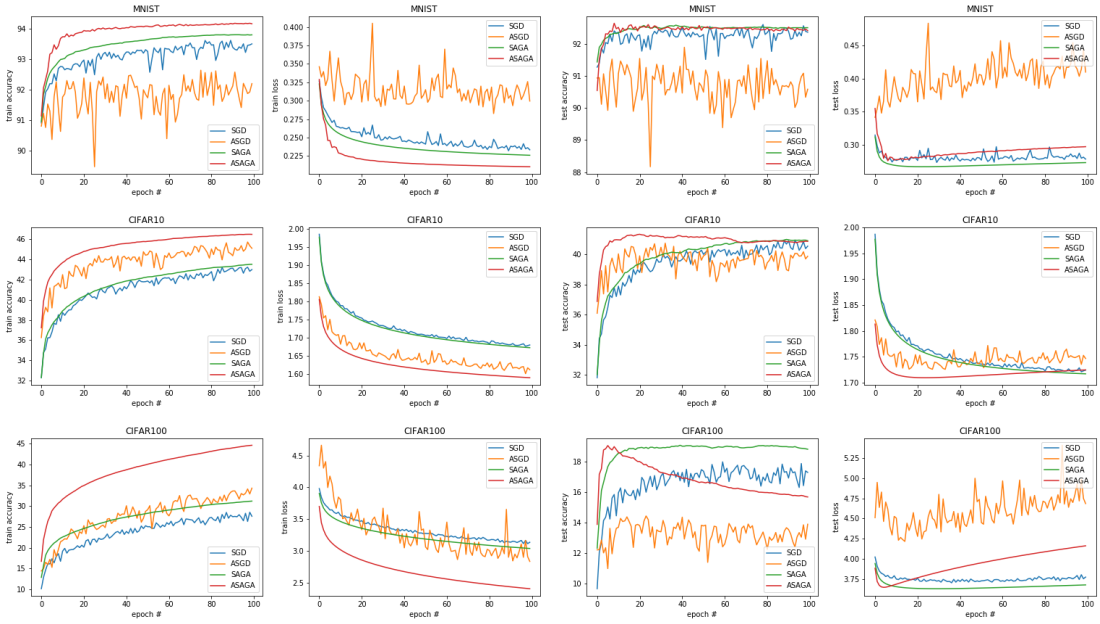

Figure 1: Evolution of (from left to right) the accuracy on the training set, the loss on the training set, the accuracy on the test set, the loss on the test set of (from top to bottom) MNIST, CIFAR10, and CIFAR100 for a softmax regression model trained using SGD (blue), SAGA (green), nesterov accelerated SGD (orange), nesterov accelerated SAGA (red).

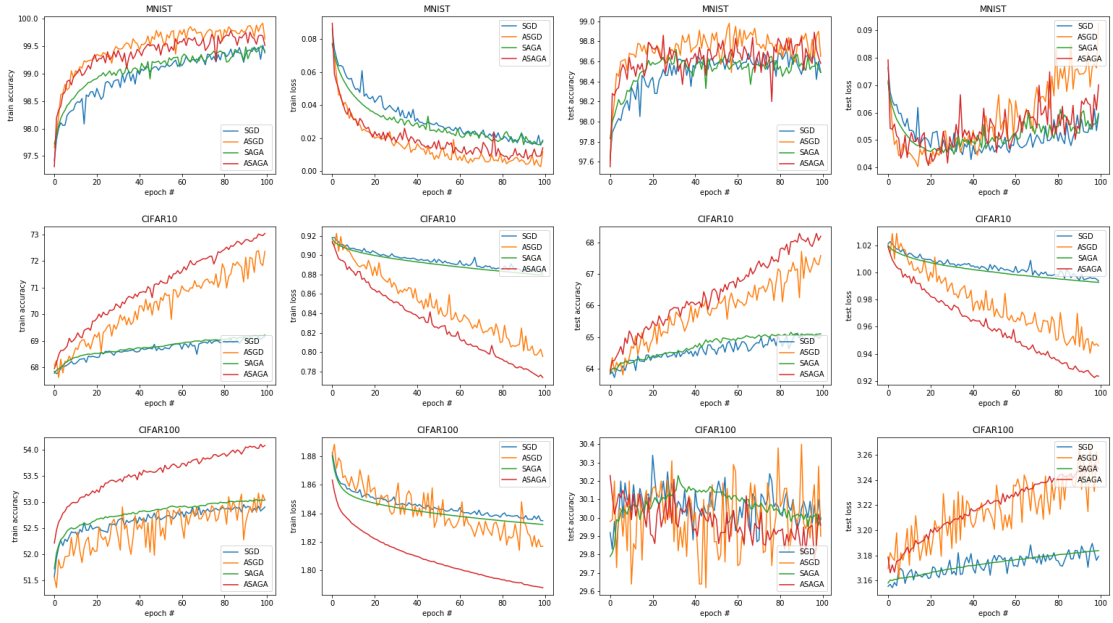

Figure 2: Evolution of (from left to right) the accuracy on the training set, the loss on the training set, the accuracy on the test set, the loss on the test set of (from top to bottom) MNIST, CIFAR10, and CIFAR100 for a convolutional neural network with six convolutional layers trained using SGD (blue), SAGA (green), nesterov accelerated SGD (orange), nesterov accelerated SAGA (red).

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[2] Niladri S. Chatterji, Nicolas Flammarion, Yi-An Ma, Peter L. Bartlett, and Michael I. Jordan. On the Theory of Variance Reduction for Stochastic Gradient Monte Carlo. feb 2018.

[3] Aaron Defazio and Léon Bottou. On the Ineffectiveness of Variance Reduced Optimization for Deep Learning. Technical report.

[4] Aaron Defazio Ambiata, Francis Bach, and Simon Lacoste-Julien. SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives. Technical report.

[5] Avinava Dubey, Sashank J Reddi, Barnabás Póczos, Alexander J Smola, Eric P Xing, and Sinead A Williamson. Variance Reduction in Stochastic Gradient Langevin Dynamics. Technical report.

[6] Robert M. Gower, Peter Richtárik, and Francis Bach. Stochastic Quasi-Gradient Methods: Variance Reduction via Jacobian Sketching. may 2018.

[7] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. sep 2009.

[8] Rie Johnson and Tong Zhang. Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. Technical report.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary Devito Facebook, A I Research, Zeming Lin, Alban Desmaison, Luca Antiga, Orobix Srl, and Adam Lerer. Automatic differentiation in PyTorch. Technical report.

[11] Sashank J. Reddi, Suvrit Sra, Barnabas Poczos, and Alex Smola. Fast Incremental Method for Nonconvex Optimization. mar 2016.

[12] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing Finite Sums with the Stochastic Average Gradient. sep 2013.