Stochastic Repeated Gradient Descent (SRGD)

Jeff Hajewski
University of Iowa
Iowa City, Iowa
Email: jeffrey-hajewski@uiowa.edu

***Abstract*—Stochastic Repeated Gradient Descent (SRGD) is a derivative of mini-batched SGD that performs multiple parameter updates for a given mini-batch within a single training epoch. The goal of this approach is to capitalize on CPU cache to reduce the wait time between training iterations.**

1. Introduction

We introduce Stochastic Repeated Gradient Descent(SRGD), a modification to Stochastic Gradient Descent that takes advantage of CPU cache to improve compute time and allow the use of large mini-batches, resulting in dramatically faster time to convergence versus mini-batch SGD.

Stochastic Gradient Descent (SGD) has long been the standard optimization method in machine learning. Recent developments such as Adam and AdaGrad solve some of the shortcomings of SGD by using adaptive learning rates and scaled gradients. While these methods have seen substantial success, they can suffer from increased storage requirements, which is problematic for large problems. Additionally, they can be more challenging to implement, as the algorithms are more complex than standard SGD. Stochastic Repeated Gradient Descent aims to reduce compute time by taking advantage of CPU cache and allowing the use of larger mini-batch sizes without degrading convergence results. Additionally, SRGD can easily make use of adaptive learning rate approaches such as that proposed by Xu et. al. [1], or adaptive solvers such as Adam.

1.1. Background

Formally, we are solving the problem

$$\theta = \arg\min_{\theta} \sum_{i=1}^{n} ||f(x_i;\theta) - y_i||_2^2 \tag{1}$$

where we have n number of data points, $x_i \in x \subset \mathbb{R}^d$ and corresponding labels $y_i \in \mathbb{R}$, and $f : \mathbb{R}^d \to \mathbb{R}$.

1.1.1. Stochastic Gradient Descent. Stochastic Gradient Descent is a derivative of Gradient Descent where the gradient $\nabla f(x;\theta)$ is replaced by $\nabla f(x_i;\theta)$, an approximation of the gradient calculated by computing the gradient at a randomly selected point, $x_i \in x$, determined by uniformly sampling (with replacement) over the data set, rather than the true gradient calculated over the entire data set. The parameter update is then performed via the standard update given by equation 2, where x_i is a single data point

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla f(x_i;\theta_k) \tag{2}$$

Each pass over the dataset is considered an epoch and multiple passes over the dataset may occur. This approach, however, has a couple of drawbacks, with the main drawback being an increase in variance of the approximate gradient. This results in slowing the rate of convergence when compared to Gradient Descent. A number of approaches attempt to reduce this variance [refs-here], but the simplest and most common approach to reduce the variance is mini-batching.

1.1.2. Mini-Batch SGD. Mini-batch SGD randomly samples some pre-determined number of points (say b points, and without loss of generality let $m = \frac{n}{b}$, such that b is a multiple of the number of points) from a uniform distribution of the data, without replacement. This mini-batch is used to attain a better (more stable) approximation of the approximate gradient, which is then used in the weight update equation 1.

1.2. Stochastic Repeated Gradient Descent

The goal of SRGD is to improve upon mini-batch SGD by using recently cached data to perform several updates of the weights via 1 before sampling new data. By not sampling new data at each update, the mini-batch data remains in cache. While this increases the variance in the theoretical convergence of our method, the compute time is reduced. This approach is best used with an adaptive learning rate. In this work, we experiemented with stepping the gradient after a certain level of convergence was readed (as proposed in [1]) but decided to use Adam as the solver, as this simplified implmenetation.

We denote the gradient update term by $\Delta\theta_{i,j}^{k}(x_{I_i}, \alpha)$ as simplification in notation. It is defined as:

$$\Delta\theta_{i,j}^{k}(x_{I_i}, \alpha) = -\alpha \nabla f(x_{I_i}; \theta_{i,j}^{k})$$

Algorithm 1

```
Require: $r \geq 1$ and $b \geq 1$
Require: $\theta_0^0 \leftarrow (u_1, u_2, \ldots, u_n)$, where $u_i \sim U[0,1]$
Require: learning rate: $\alpha_0 > 0$
Require: update threshold: $\epsilon > 0$
Require: number of epochs: $l \geq 1$
1: for $k = 1$ to $l$ do
2:     $\theta_0^k = \theta_0^{k-1}$
3:     for $i = 0$ to $\frac{n}{b}$ do
4:         $I_i \sim U[1, n]$ such that $|I_i| = b$
5:         $x_{I_i} \leftarrow \{x_i | i \in I_i, x_i \in x\}$
6:         for $j = 1$ to $r$ do
7:             $\theta_{i+1,j}^k \leftarrow \theta_{i,j}^k + \Delta\theta_{i,j}^k(x_{I_i}, r, \alpha)$
8:         end for
9:     end for
10: end for
```

The term I_i is a set of indices defined

$$I_i = \{i | i \sim U[1, n]\}$$

where n, as noted previously, is the number of data points and the indices i are IID random variable sampled from a uniform distribution on the interval $[1, n]$. We use I_i as an index to denote the set of x_i whose indices are in I_i.

$$x_{I_i} = \{x_i | i \in I_i,\ \ x_i \in x\}$$

2. Algorithm

Algorithm 2 details the SRGD algorithm. Additionally, we check for convergence and stop early if convergence as been achieved. The key step is repeating with weight update r times using the same mini-batch data. This results in r approximate gradient computations along with r weight updates.

3. Theory

4. Experiment

Experiments were performed on a 3.4GHz Intel i7-6700K hexa-core workstation with 64GB of RAM. We only consider time to convergence in our tests. We expect that the total number of steps (i.e., work) is greater than standard SGD, however we capitalize on faster compute time to overcome the greater amount of work.

4.1. Problem Definition

We use the MNIST data set [?] to evaluate our approach. The goal of the MNIST data set is to calssify the input images as one of the digits 0 - 9. The model we fit is a convolutional neural network, consisting of two convolutional layers, both feedinginto max pooling layers before hitting ReLU activation functions. The second convolutional layer also implements dropout[ADD REF]. Lastly, there are two fully connected, linear layers prior to a softmax.

TABLE 1. BATCH-SIZE OF 1 IN 100 DIMENSIONS

Optimizer	$r = 1$		$r = 5$		$r = 10$	
	Error	Time	Error	Time	Error	Time
SGD	7.1752	0.4	7.1752	0.4	7.1752	0.4
ASSGD	0.7824	2.4	0.7824	2.4	0.7824	2.4
SRGD	0.0003	0.4	0.0004	0.1	0.0000	0.1
ASSRGD	0.2148	1.5	0.0077	0.40	0.0022	0.3

TABLE 2. BATCH-SIZE OF 50 IN 100 DIMENSIONS

Optimizer	$r = 1$		$r = 5$		$r = 10$	
	Error	Time	Error	Time	Error	Time
SGD	0.0410	48.5	0.0410	48.5	0.0410	48.5
ASSGD	0.0399	48.7	0.0399	48.7	0.0399	48.7
SRGD	0.0000	45.6	0.0000	9.5	0.0000	7.3
ASSRGD	0.0000	48.6	0.0000	11.8	0.0000	7.1

4.2. Methodology

We use PyTorch [?], a machine learning framework that is an offshoot of Torch, written almost entirely in Python, but also has some low-level linear algebra libraries that can utilize CUDA. It is very easy to use, and since it is written almost entirely in Python, it is very extensible.

4.3. Results

Based on our experiments there appear to be two domains in which SRGD is optimal. In one domain, typically as the mini-batch size approaches the problem size, SRGD takes more compute time but achieves materially better accuracy. In the other domain, SRGD runs at a fraction of the time of SGD and achieves better accuracy (frequently an order of magnitude better).

Additionally (Need to add figure/table), it is interesting to note that the SRGD and ASSRGD methods achieved better accuracy, typically in less time, and they utilized fewer data points than standard SGD. In other words, they were able to do more with less information.

5. Future Work

A key issue in evaluating the efficacy of this new methodology is determining whether the algorithm is getting cache hits on the data while training. This is an

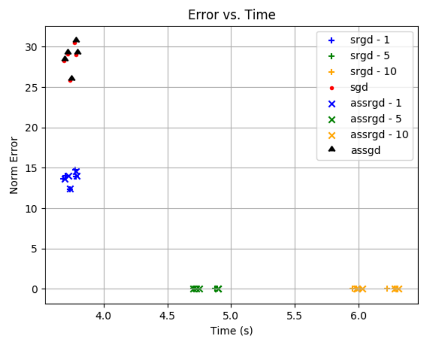

Figure 1. Comparison of error as a function of compute time for a mini-batch of 500 points and a 100 dimensional parameter space.

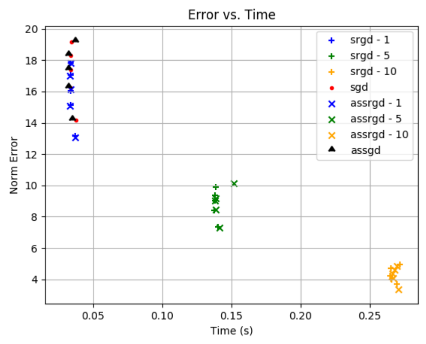

Figure 2. Comparison of error as a function of compute time for a mini-batch of 500 points and a 10 dimensional parameter space.

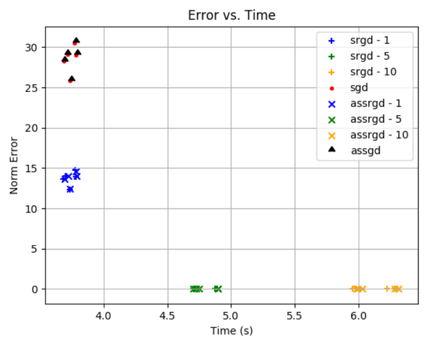

Figure 3. Comparison of error as a function of compute time for a mini-batch of 1,000 points and a 100 dimensional parameter space.

extremely low-level process and not as straightforward to measure. Further compounding the issue is the use of Python makes cache-misses extremely likely (due to the nature of the Python language). At a high level, almost everything in Python is an object and so there is a lot of indirection for seemingly simple Python expressions. This means that by the time we perform an update and go to recalculate the gradient for the same data point, the data has probably been evicted from the cache. At this point, it must be refetched from RAM, which is the exact situation

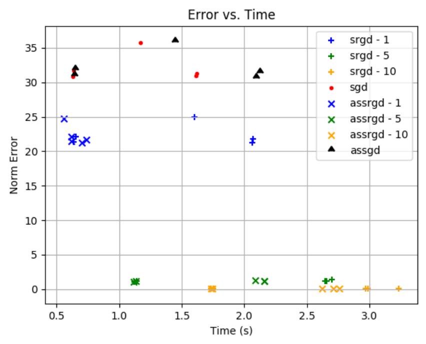

Figure 4. Comparison of error as a function of compute time for a mini-batch of 500 points and a 50 dimensional parameter space.

we are trying to avoid with this new approach.

The question remains: what can be done? The next step is to move to a lower-level language such as C or C++ and implement a smaller test problem. Our intution is that while this started out as a clever approach to squeeze some additional performance out of various optimization algorithms, it appears to effectively utilize the L1 or L2 caches the optimization routines will need to be highly optimized to a point that it is beyond the scope of an average library user. Take, as a comparison, the BLAS library, which is highly optimized and makes extensive use of cache-optimized algorithms. The matrix-multiply routines are substnatially more complicated than the standard approach of three nested for-loops.

The last complicating factor in our approach is that of both hyper-threading and asynchronous data fetching. In the hyper-threaded setting, the CPU actually prefetches the data into cache while performing its standard tasks (actually, these two processes are interleaved) – this has the effect of removing any latency from RAM access. That's not to say there is no room for cache optimization on modern processors, just that it takes quite a bit more planning. The other related issue is that of asynchronous data fetching. The framework Caffe [ADD REF] is a great example of this approach. Caffe creates (at least) two threads, one for main program execution and the other to handle data fetching. While the model is making its forward and backward passes (training phase), the data thread is pulling data from the database (which is typically on disk) into memory, preparing it for use in training. This implementation is quite efficient, and removes a lot of the latency from data access.

6. Conclusion

While we did not achieve the results we were hoping for in this project, a lot was learned. At a minimum, at this level of optimization having a strong understanding of the hardware and OS operations is quite important. Making use of cache optimizations can certainly improve overall run-time, but is much more complex than we initially thought.

It may turn out that these optimzations can be made, just not at the data load phase of the optimizer.

Acknowledgments

References

[1] Y. Xu, Q. Lin, T. Yang, *Accelerated Stochastic Subgradient Methods under Local Error Bound Condition*, arXiv:1607.01027, 2017

[2] Y. LeCun, C. Cortes, *MNIST handwritten digit database*, http://yann.lecun.com/exdb/mnist/, 2010

[3] *PyTorch*, https://pytorch.org