DSA4212 Optimisation for Large-Scale Data-Driven Inference

Academic Year 2021/22 Semester 2

Group 3 Assignment 2 Report

Exploration of Variance Reduction Techniques for Stochastic Gradient Descent (SGD) with Practical Implementation

Group members:

Name	Matric No.
Li Zhuoran	A0194510L
Pan Yuting	A0201267N
Zhang Simian	A0204709H
Zhong Zhaoping	A0194519U

1. Introduction

The stochastic gradient descent algorithm (SGD) is widely applied in optimization problems, in particular large-scale classification problems. The goal is to minimize an objective function in the form of a *finite-sum*:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x),$$

where $x \in R^d$. In addition, our group also consider the composite objective function with an additional *regularization term*:

$$F(x) = f(x) + h(x),$$

where $h(x)$ often measures the magnitude of model parameters to control for model complexity.

However, SGD has its main limitation of high variance of update which leads to slow convergence in many cases. In the original paper of SVRG (Johnson & Zhang, 2013), the authors make the observation that the variance of the standard SGD update direction can only go to zero if decreasing step sizes are used, thus preventing a linear convergence rate unlike for full gradient descent.

To resolve this challenge, there has been remarkable development in the research community which proposes variance reduction(VR) algorithms like SAG (Schmidt et al., 2016), SVRG (Johnson & Zhang, 2013), SAGA (Defazio et al., 2014) to minimize the objective function with a faster convergence rate than SGD, both theoretically and empirically, which allows a linear convergence rate with constant step size. While the paper of references provides sound results for VR algorithms implemented with single-data gradient calculations, our group is interested in finding out whether these algorithms have practical values to be implemented with minibatch gradient calculations, since most algorithms in deployment harness GPU for their capability of fast matrix computation. Our group validated the performance of different algorithms in single-version, and investigated their practical value in minibatch-version. Our results prove that the VR algorithms indeed have faster convergence in single-version, but most do not see significant improvements in minibatch-version.

2. Methods

2.1 Stochastic Average Gradient (SAG)

Stochastic Average Gradient (SAG) method is an improved version of the basic SGD. Like SGD methods, the SAG method's iteration cost is independent of the number of terms in the sum. However, by incorporating a memory of previous gradient values the SAG method achieves a faster convergence rate than black-box SG methods. It is shown that the convergence rate of SAG is improved from $O\ (1\ /\sqrt{k})$ to $O(1/k)$ in general, and when the sum is strongly-convex the convergence rate is improved from the sub-linear $O(1/k)$) to a linear convergence rate of the form $O(p^k)$ for $\rho < 1$ (Schmidt et al., 2016).

Stochastic Average Gradient (SAG) method is designed based on the incremental aggregated gradient (IAG) method of (Blatt et al., 2007). The main idea of IAG is to introduce an aggregated gradient d^k, which is initialized by the sum of the gradients given by the first L data points (i.e. $d^L = \sum_{l=1}^{L} f(x^l)$).

when we do a new iteration (let's say the k^{th} iteration, where K > L), we subtract d^k by the gradient given at the $(k - l)^{th}$ iteration, and add the gradient of the new iteration into d^k. Thus, d^k is always an aggregated gradient of L data points.

$$x^{k+1} = x^k - \alpha\frac{1}{L}d^k$$

$$\text{where} \quad d^k = d^{k-1} - \nabla f_{(k)}(x^{k-L}) + \nabla f_{(k)}(x^k)$$

Intuitively, we can treat d^k as a memory table of size L, which keeps adding gradients generated from new data points and dropping the gradient generated by the earliest data point in the table. Moreover, a factor $\frac{1}{L}$ is explicitly included here to make the approximated descent direction $\frac{1}{L} d^k$ comparable in magnitude to the one used for standard full gradient descent.

Based on the above method, SAG is a randomized variant of IAG, which has the following modifications:

1. Instead of using a cyclic choice of data points, the SAG method randomly samples new data points with replacement.
2. The size of aggregated gradient L is set to be the size of the training dataset n.
3. In order to be more computationally efficient, the initialized d^k is set to be 0 (treating all data points as unobserved in the initialization step).

The formal definition for SAG is

$$x^{k+1} = x^k - \frac{\alpha}{n}\sum_{i=1}^{n} y_i^k$$

where at each iteration a random index i_k is selected and we set

$$y_i^k = \begin{cases} f_i'(x^k) & \text{if } i = i_k, \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

In comparison with IAG, the term $\sum_{i=1}^{n} y_i^k$ used in SAG is indeed the d^k in IAG. We can interprete y_i^k as a memory table which records the gradient of each y_i calculated in the k^{th} iteration. For each iteration, only one y_i is updated. This can be interpreted as a special case of the dropping and adding process of d^k in IAG. The following algorithm is a full implementation of SAG method:

SAG Algorithm

```
Initialize: aggregated gradient $d = 0$, and memory table $y_i = 0$ for $i = 1, 2, \dots, n$
for $k = 0, 1, \dots$ do
    Sample $i$ from $\{1, 2, \dots, n\}$
    $d = d - y_i + \nabla f_i(x)$
    $y_i = \nabla f_i(x)$
    $x = x - \frac{\alpha}{n} d$
end
```

One noticeable issue for the above SAG algorithm is the slow early learning steps. In the above SAG algorithm, we normalize the direction d by the total number of data points n and the initialized y_i are all zero. This leads to very small steps during early iterations of the algorithm where we have only seen a fraction of the data points, because many y_i variables contributing to d are set to the uninformative zero-vector.

Hence, we introduce re-weighting on early iterations, which is a more logical normalization method to to divide d by m, the number of data points that we have seen at least once (which converges to n once we have seen the entire data set), leading to the update $x = x - \frac{\alpha}{m} d$.

Another noticeable point is that using the traditional SAG algorithm is not the best practical choice for real world large scale data because it does not make use of the parallel computation functionalities provided by GPUs. Therefore, we came up with a batch-version SAG algorithm by ourselves to resolve this challenge. Specifically, now each time we do not sample only a single data point, but sample a stochastic batch of data instead. The batch-version SAG algorithm is shown as following:

```
Minibatch SAG Algorithm
Initialize: aggregated gradient $d = 0$, and batch memory table $b_i = 0$ for $i$ = 1, 2, …, L where L is the number of batches
for $k$ = 0, 1, … do
    Sample $i$ from {1, 2, …, L}
    $d = d - b_i + \nabla f_i(x)$
    $b_i = \nabla f_i(x)$
    $x = x - \frac{\alpha}{n} d$
end
```

2.2 Stochastic Variance Reduced Gradient (SVRG)

Stochastic Variance Reduced Gradient (SVRG) is an explicit variance reduction method for stochastic gradient descent. SVRG adopted an inner-outer loop structure to update the gradient (Sebbouh et al., 2019). A reference full gradient (snapshot) is evaluated in the outer loop, following *m* steps of the gradient updates in the inner loop, where the reference gradient is used to construct a variance-reduced estimate of the current gradient.

Specifically, in the outer loop, an optimal estimate $\widehat{x}$ of the true x is computed. Moreover, an average gradient $\widehat{\mu}$ is maintained in the outer loop as $\widehat{\mu} = \nabla P(\widehat{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\widehat{x})$ by one pass over the data using $\widehat{\omega}$. In the inner loop, *m* times of gradient updates will be executed. Each of the *m* gradient updates takes up a random number *i* from *n*, the total number of data, and updates the weight as $x^{(t)} = x^{(t-1)} - \eta_t(\nabla f_i(x^{(t-1)}) - \nabla f_i(\widehat{x}) + \widehat{\mu})$. After *m* iterations of the inner loop to be executed, update $\widehat{x} = x_m$ (option 1) or $\widehat{x} = x_t$ for randomly chosen t from 0 to *m*-1 (option 2). In practical implementations, option 1 is favoured to simplify the algorithm. The value of *m* should be the same order of *n* but slightly larger (for example *m* = 2*n* for convex problems) since for each data point requires 2*m* + *n* gradient computations (Johnson & Zhang, 2013). The pseudo-code of the procedure of SVRG can be found below.

```
SVRG Algorithm
Parameters update frequency $m$ and learning rate $\eta$
Initialize $\widehat{x_0}$
Iterate: for $s$ = 1, 2, 3, …
    $\widehat{x} = \widehat{x_{s-1}}$
    $\widehat{\mu} = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\widehat{x})$
    $x_0 = \widehat{x}$
    Iterate: for $t$ = 1, 2, … , $m$
        Randomly pick $i_t \in \{1, ..., n\}$ and update weight
        $x_t = x_{t-1} - \eta(\nabla f_{it}(x_{t-1}) - \nabla f_{it}(\widehat{x}) + \widehat{\mu})$
    end
option I: set $\widehat{x_s} = x_m$
option II: set $\widehat{x_s} = x_t$ for randomly chosen $t \in \{0, ... , m - 1\}$
end
```

The random sampling of *i* and the use of the average gradient enable SVRG to reduce its variance. To be specific, it can be noticed that the expectation of $\nabla f_i(\hat{x}) - \hat{\mu}$ over *i* is zero (Johnson & Zhang, 2013), hence, the update rule $x^{(t)} = x^{(t-1)} - \eta_t(\nabla f_i(x^{(t-1)}) - \nabla f_i(\hat{x}) + \hat{\mu})$ can be regarded as generalized SGD. It can be shown that $\hat{\mu}$ will converge to 0 and $\nabla f_i(\hat{x})$ will converge to $\nabla f_i(x_*)$ when both $\hat{x}$ and $x^{(t)}$ converge to the same parameter x_* (Johnson & Zhang, 2013), and consequently, $\nabla f_i(x^{(t-1)}) - \nabla f_i(\hat{x}) + \hat{\mu} \rightarrow \nabla f_i(x^{(t-1)}) - \nabla f_i(x_*) + \hat{\mu}$ will converge to 0, which shows that the variance of the update rule $x^{(t)} = x^{(t-1)} - \eta_t(\nabla f_i(x^{(t-1)}) - \nabla f_i(\hat{x}) + \hat{\mu})$ is reduced.

Although it has been proven that the aforementioned SVRG procedure can achieve variance reduction, the practicality of SVRG may be compromised considering that the SVRG is often applied to large-scale data that takes a high time cost. Since the SVRG method needs to perform an inner and outer loop update for each data point in the training dataset, it consumes a lot of time to iterate through each data point and hence compromises the practicality of SVRG. To further improve the utility of SVRG, our group implemented the mini-batch version of SVRG. Specifically, SVRG using mini-batches randomly divides the entire training dataset into many mini-batches, and each mini-batch is packaged as a whole to contain hundreds of data points. The structure of the inner and outer loops of the original SVRG is preserved, the difference is that the gradient updates computed by the inner loop are now based on a mini-batch rather than a single data point (Babanezhad et al., 2015) and the mini-batching SVRG procedure can be found below. Such a design still ensures the randomness of SVRG to achieve the purpose of reducing variance, and at the same time effectively reduces the number of loops and the amount of computation required for gradient update, which greatly improves the execution speed of SVRG and makes up for the speed shortcomings of the original SVRG version (Sebbouh et al., 2019). It is recommended to adopt the mini-batch version SVRG for practical use.

```
                    **Minibatch SVRG Algorithm**
**Parameters** update frequency $m$, total number of mini-batches $n$, learning rate $\eta$
**Initialize** $\widehat{x_0}$, $n$ random mini-batches
**Iterate**: for $s = 1, 2, 3, \ldots$
    $\hat{x} = \widehat{x_{s-1}}$
    $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\hat{x})$
    $x_0 = \hat{x}$
    **Iterate**: for $t = 1, 2, \ldots, m$
        Randomly pick a mini-batch $B_i$, $i \in \{1, \ldots, n\}$ and update weight
        $x_t = x_{t-1} - \eta(\nabla f_{it}(x_{t-1}, B_i) - \nabla f_{it}(\hat{x}, B_i) + \hat{\mu})$
    **end**
**option I**: set $\widehat{x_s} = x_m$
**option II**: set $\widehat{x_s} = x_t$ for randomly chosen $t \in \{0, \ldots, m-1\}$
**end**
```

2.3 SAGA

The SAGA method is designed in the spirit of SAG, SDCA, MISO and SVRG, a set of famous incremental gradient algorithms with fast linear convergence rates. SAGA improves on the theory behind SAG and SVRG, with better theoretical convergence rates, and is able to support non-strongly convex problems directly (Defazio et al., 2014).

We start with some known initial vector $x^0 \in R^d$ and known gradients $\nabla f_i(\phi_i^0) \in R^d$ with $\phi_i^0 = x^0$ for each i. These gradients are stored in a table data-structure of length n, or alternatively a $n \times d$ matrix. SAGA uses a step size of γ and makes the following updates, starting with $k = 0$:

SAGA Algorithm

Given the value of x^k and of each $\nabla f_i(\phi_i^k)$ at the end of iteration k, the updates for iteration $k + 1$ is as follows:

```
for $k = 0, 1, 2, \dots$ do
    1. Pick a $j$ uniformly at random.
    2. Take $\phi_j^{k+1} = x^k$, and store $\nabla f_j(\phi_j^{k+1})$ in the table. All other entries in the table remain unchanged. The quantity $\phi_j^{k+1}$ is not explicitly stored.
    3. Update $x$ using $\nabla f_j(\phi_j^{k+1})$, $\nabla f_j(\phi_j^k)$ and the table average:
           $g^k = \nabla f_j(\phi_j^{k+1}) - \nabla f_j(\phi_j^k) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\phi_i^k)$
           $x^{k+1} = x^k - \gamma g^k$
end
```

A notable difference between SAGA and SAG is that SAGA uses an unbiased gradient estimate g^k to update the parameter, whereas SAG uses a biased one. Compared with SVRG, SAGA updates the ϕ_j's value each time index j is picked, whereas SVRG updates all of ϕ's as a batch. Therefore, we would like to hold the idea that SAGA is essentially at the midpoint between SVRG and SAG (Defazio et al., 2014).

As for real world large-scale datasets, using the traditional SAGA algorithm (presented in the original SAGA paper) is not practical because it samples only one data point each time to calculate the gradient estimate, which does not make use of the parallel computation functionalities provided by GPUs. Therefore, we came up with a batch-version SAGA algorithm by ourselves to resolve this challenge. Specifically, now each time we do not sample only a single data point, but sample a stochastic batch of data instead. The batch-version SAGA algorithm is shown as follows:

Minibatch SAGA Algorithm

Given the mini-batch B, the value of x^k, and the gradient memory table $J^0 \in R^{d \times n}$ at the end of iteration k, the updates for iteration $k + 1$ is as follows:

```
for $k = 0, 1, 2, \dots$ do
    1. Sample a random batch $B \subseteq [n]$ $s.t.$ $|B| = b$.
    2. Update the gradient estimate $g^k = \frac{1}{n} J^k e + \frac{1}{b}\sum_{i \in B}(\nabla f_i(x^k) - J_{:i}^k)$
    3. Take an optimization step $x^{k+1} = x^k - \gamma g^k$
    4. Update the gradient memory table:
        a. if $i \in B$, $J_{:i}^{k+1} = \nabla f_i(x^k)$.
        b. if $i \notin B$, $J_{:i}^{k+1} = J_{:i}^k$.
end
```

3. Variance Reduction and Methods Comparison

We first review a slightly more generalized version of the variance reduction approach (we allow the updates to be biased). Suppose that we want to use Monte Carlo samples to estimate $E[X]$ and that we can compute efficiently $E[Y]$ for another random variable Y that is highly correlated with X. One variance reduction approach is to use the following estimator θ_α as an approximation to $E[X]$: $\theta_\alpha := \alpha(X - Y) + E[Y]$, for a step size $\alpha \in [0, 1]$. We have that $E[\theta_\alpha]$ is a

convex combination of $E[X]$ and $E[Y]$: $E[\theta_\alpha] = \alpha E[X] + (1 - \alpha)E[Y]$. The standard variance reduction approach uses $\alpha = 1$ and the estimate is unbiased $E[\theta_\alpha] = E[X]$. The variance of θ_α is:
$Var(\theta_\alpha) = \alpha^2[Var(X) + Var(Y) - 2\,Cov(X, Y)]$, and so if $Cov(X, Y)$ is big enough, the variance of θ_α is reduced compared to X, giving the method its name. By varying α from 0 to 1, we increase the variance of θ_α towards its maximum value (which usually is still smaller than the one for X) while decreasing its bias towards zero (Gower & Gramfort, 2019).

Both SAGA and SAG can be derived from such a variance reduction viewpoint: here X is the SGD direction sample $\nabla f_j(x^k)$, whereas Y is a past stored gradient $\nabla f_j(\phi_j^k)$. SAG is obtained by using $\alpha = 1/n$ (update rewritten in our notation in (1)), whereas SAGA is the unbiased version with $\alpha = 1$ (see (2) below). For the same ϕ's, the variance of the SAG update is $1/n^2$ times the one of SAGA, but at the expense of having a non-zero bias. This non-zero bias might explain the complexity of the convergence proof of SAG and why the theory has not yet been extended to proximal operators. By using an unbiased update in SAGA, we are able to obtain a simple and tight theory, with better constants than SAG, as well as theoretical rates for the use of proximal operators (Defazio et al., 2014).

SAG: $$x^{k+1} = x^k - \gamma\left[\frac{\nabla f_j(x^k) - \nabla f_j(\phi_j^k)}{n} + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\phi_i^k)\right] \quad (1)$$

SAGA: $$x^{k+1} = x^k - \gamma\left[\nabla f_j(x^k) - \nabla f_j(\phi_j^k) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\phi_i^k)\right] \quad (2)$$

SVRG: $$x^{k+1} = x^k - \gamma\left[\nabla f_j(x^k) - \nabla f_j(\hat{x}) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\hat{x})\right] \quad (3)$$

The SVRG update (see (3) above) is obtained by using $Y = \nabla f_j(\hat{x})$ with $\alpha = 1$ (and is thus unbiased - we can observe that SAG is the only method that we present in the report that has a biased update). Essentially SAGA is at the midpoint between SAG and SVRG. It updates the ϕ_j value each time index j is picked, whereas SVRG updates all of ϕ's as a batch. SVRG makes a trade-off between time and space. For the equivalent practical convergence rate it makes 2 - 3 times more gradient evaluations (depending on different implementations), but in doing so it does not need to store a table of gradients, but a single average gradient. The usage of SAGA vs. SVRG is problem dependent. For example, for linear predictors where gradients can be stored as a reduced vector of dimension $p - 1$ for p classes, SAGA is preferred over SVRG both theoretically and in practice (e.g. multi-class logistic regression). For neural networks, where no theory is available for either method, the storage of gradients is generally more expensive than the additional backpropagation, but this is computer architecture dependent. SVRG also has an additional parameter besides step size that needs to be set, namely the number of iterations per inner loop (m). This parameter can be set via the theory, or conservatively as $m = n$, however doing so does not give anywhere near the best practical performance. Having to tune one parameter instead of two is a practical advantage for SAGA (Defazio et al., 2014).

4. Data

Data set	Type	# Training Data	# Test Data	# Variables
adult	Binary	32561	16281	14
covtype	Multi-class	435759	145253	54
ijcnn	Binary	49990	91701	22
mnist	Multi-class	60000	10000	784

To test the performance of different algorithms, our group selected a few large-scale data sets that are publicly available and used in the paper of our references. Each data set was split into a training set and a test set, if not already split in the original source. All data sets have a size of 10^5, with *covtype* having an even larger size of 10^6, and *mnist* having a higher dimension of 10^5. *adult*, *covtype* and *ijcnn* were preprocessed with a standard procedure that standardized all numerical features and one-hot encoded all categorical features. *mnist* was preprocessed by dividing every feature by 255 to ensure a scale of 0 to 1. The details of the data could be found in the notebook with links to sources included.

5. Experiment & Results

While the paper provides theoretical convergence rates of the different variance reduction (VR) techniques, our group decided to explore the empirical performance based on our own implementation. It is to be noted that all the paper provides algorithms in the version of passing once a single data to calculate the gradient, which our group believe is not efficient given the parallel computing power of GPU to handle matrix computation. As such, our group implemented a single-version of the VR algorithms according to the paper, and another minibatch-version for each algorithm as aforementioned.

Our group used logistic regression with cross-entropy loss for all experiments. The logistic regression model is modified such that each class of labels has an associated vector of parameters. The probability of a data belonging to class i, among k classes is $P(Y = Cl_i|X) = \frac{exp(\beta_i * X)}{\sum_{i=1}^{k} exp(\beta_i * X)}$ and the corresponding loss is the *negative log likelihood* of the true class $L(Y, X) = -log(P(Y = Cl_Y))$. This was chosen and implemented because there are multi-class labels in some data sets, and cross-entropy loss could also be applied on binary-class classification tasks.

For the single-version algorithms, our group evaluated the performance based on training loss. As discussed above, the goal of VR algorithms is to reduce the variance of the update gradient so that a faster convergence to the theoretical minima could be achieved. While it is difficult to get the exact minima, our group determined that it was sufficient to compare the training loss, since the distance to the loss minimum is equivalent to subtracting a constant from the training loss.

For the minibatch-version algorithms, our group has two objectives in our experiment. Firstly our group would like to test whether the faster convergence also applies in the minibatch-version, so we compared the training loss similar to the single-version. Secondly it is important to check whether the faster convergence is prone to overfitting, since the generalized performance on unseen data is of paramount significance for deployment. As such, our group also compared the test accuracy of different algorithms as well.

All the metrics mentioned above are compared against the number of gradient calculations divided by the number of data. This is to ensure fairness of comparison since not all algorithms update the parameters immediately after a gradient calculation.

5.1 Single-Version

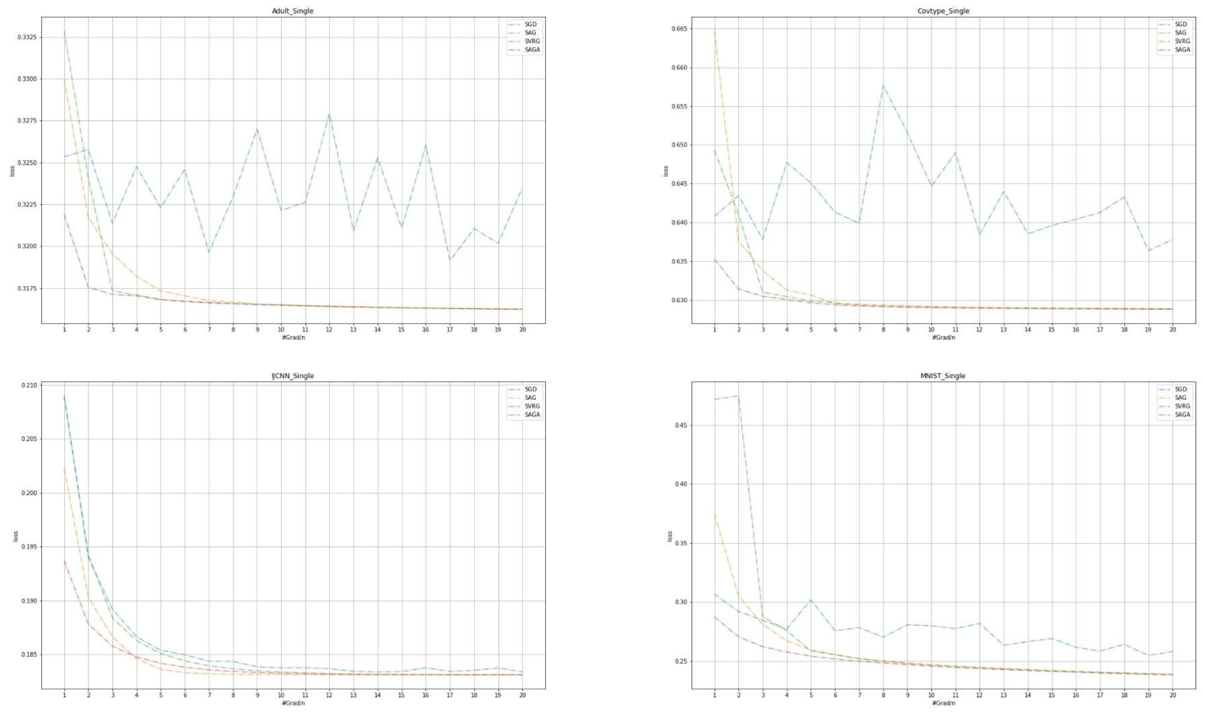

Figure 1. Loss of Single-Version

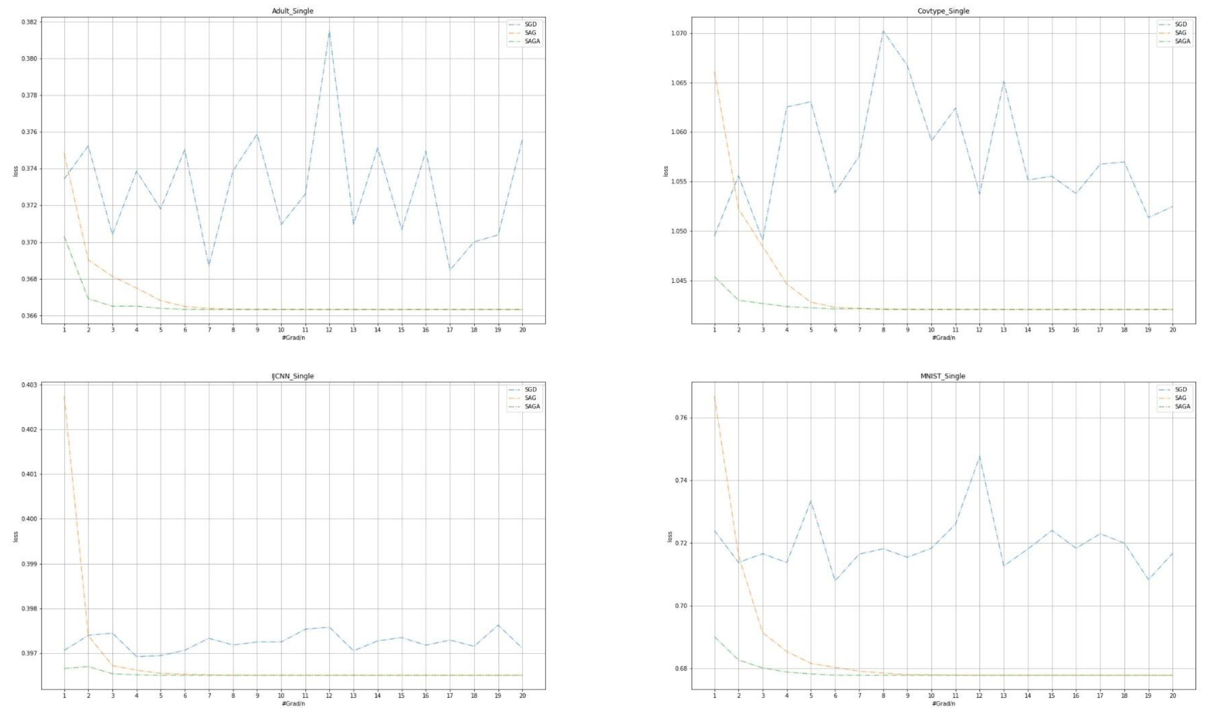

Figure 2. Loss of Single-Version with L2 regularization

From Figure 1 and Figure 2 above, we can easily observe that SAGA outperforms SGD, SAG and SVRG in both with and without L2 regularization. Notably, SAGA can reduce the training loss rapidly within only the first three #Grad/n. In Section 3, we mentioned that SAGA is theoretically preferred over SAG in most of the scenarios, whereas the usage of SAGA vs. SVRG is problem dependent. As for linear predictors like multi-class logistic regression which is exactly our case, SAGA is preferred over SVRG both theoretically and in practice. The empirical results here are exactly aligned with our argument in Section 3 above.

In our experiment, we did not apply L2-regularization to SVRG. This is because L2-regularization cannot be directly applied to SVRG. Specifically, the optimization problem with L2-regularization can be written as min $f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) + h(x)$, where *h(x)* takes the form of $h(x) = \frac{\lambda}{2}||x||^2$. The original update rule of SVRG requires computing the gradient of loss function using $x^{(t-1)}$ and $\hat{x}$, that is $\nabla f_i(x^{(t-1)})$ and $\nabla f_i(\hat{x})$. However, the gradient computation in update rule cannot be applied to the L2-regularization $h(x) = \frac{\lambda}{2}||x||^2$ since the regularization term is not corporated into the SVRG process, it is meaningless to use the SVRG update rule to calculate the gradient of the regularization term i.e., $h(x^{(t-1)})$ and $\nabla h(\hat{x})$, and doing so does not guarantee convergence for SVRG. Hence, in our experiment, we did not apply L2-regularization to SVRG. If L2-regularizer is required, the update rule needs to change accordingly. Specifically, to take an exact gradient step with respect to the regularizer and an SVRG step with respect to the loss functions (Babanezhad et al., 2015) i.e., different gradient computations for regularizer and loss function.

An interesting observation is that SVRG tends to perform poorly for the 1st pass. This is because the weights for SVRG were initialized by performing 1 iteration (convex) or 10 iterations (nonconvex) of SGD (Johnson & Zhang, 2013), which suggests that the 1st effective pass served as a “warm-up” initialization step. Furthermore, the advantage of SVRG over other variance reduction methods such as SAG and SAGA is low storage cost and fast execution speed. Methods such as SAG and SAGA require storing a gradient sample for each training example and the storage of a full table of gradients takes up a memory cost of O(*nd*), where *n* is the number of training data and *d* is the model dimension. In contrast, SVRG does not require storing a gradient sample for each training example and takes up a memory cost of O(3*d*), which is significantly lower than SAG and SAGA. Moreover, the excessive space complexity of SAG and SAGA could potentially cause a slower speed of the updating process. When the dataset is extremely large (> 1 million), reading and updating the memory table may be a heavy burden for practical use and the difference between the execution time of SVRG and SAG or SAGA will be significant. In our experiment, for the largest dataset *covtype* that has 435759 training data, SAG and SAGA need 1200 seconds and more to finish the training process while SVRG takes about 420 seconds, which is about 3 times faster. In summary, the low storage cost and fast execution speed enable SVRG to be executed on as many computing platforms as possible in the context of large-scale data, which greatly increases its universality and practicality.

5.2 Batch-Version

5.2.1 Training Loss

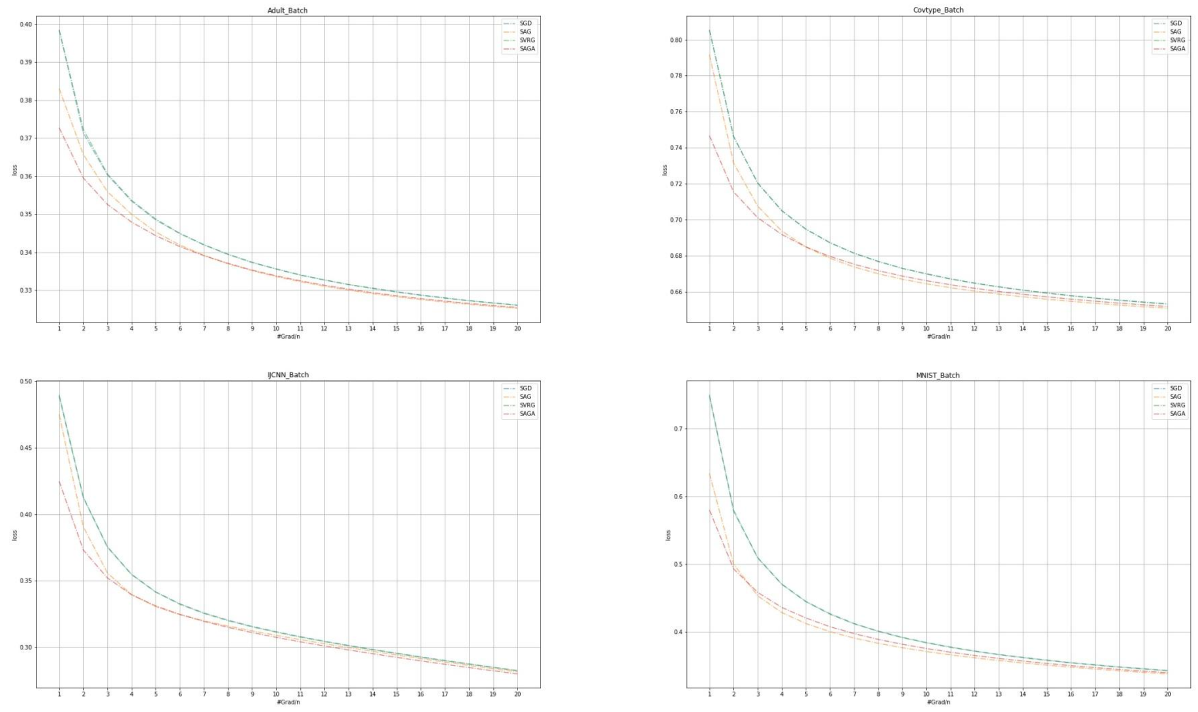

Figure 3. Loss of Minibatch-Version

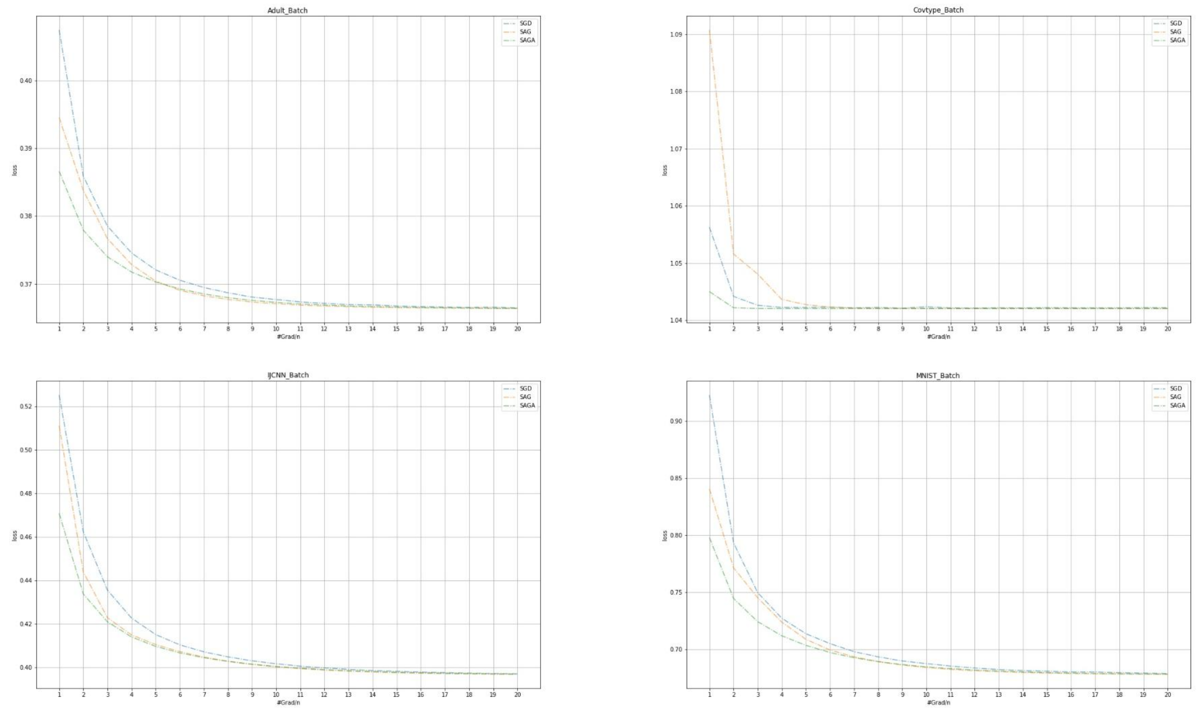

Figure 4. Loss of Minibatch-Version with L2 regularization

From Figure 3 and Figure 4 above, we can easily observe that minibatch-version SAGA outperforms all of the other minibatch-version algorithms in both scenarios, with and without L2 regularization. This empirical result is exactly aligned with the result of experiments of single-version algorithms.

Furthermore, the result above shows that SAG achieves better results than SGD in most of the batch versions with L2 regularization, except for the data set *covtype*. While the original paper shows that L2 regularization outperforms the basic SGD. The exact reason remains unknown, but one noticeable difference might be the pre-set regularization strength and step size of the gradient descent. Therefore, it is not universally true that SAG performs better than SGD in the batch version with L2 regularization.

5.2.2 Test Accuracy

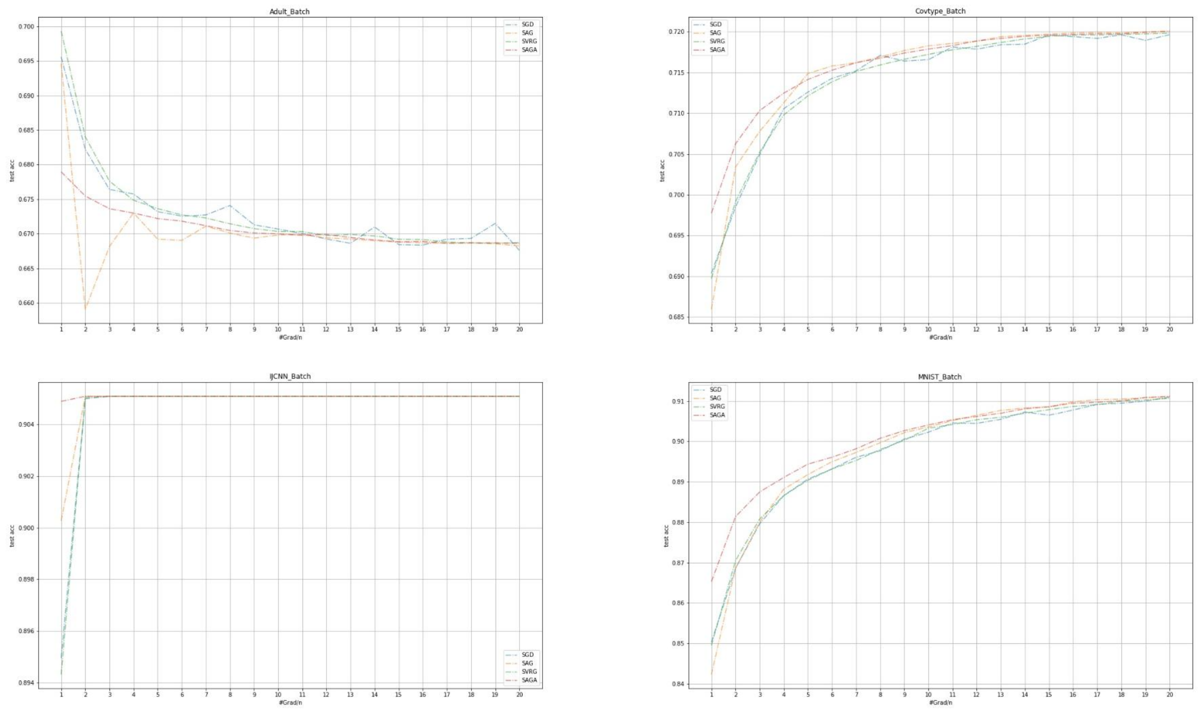

Figure 5. Accuracy of Minibatch-Version

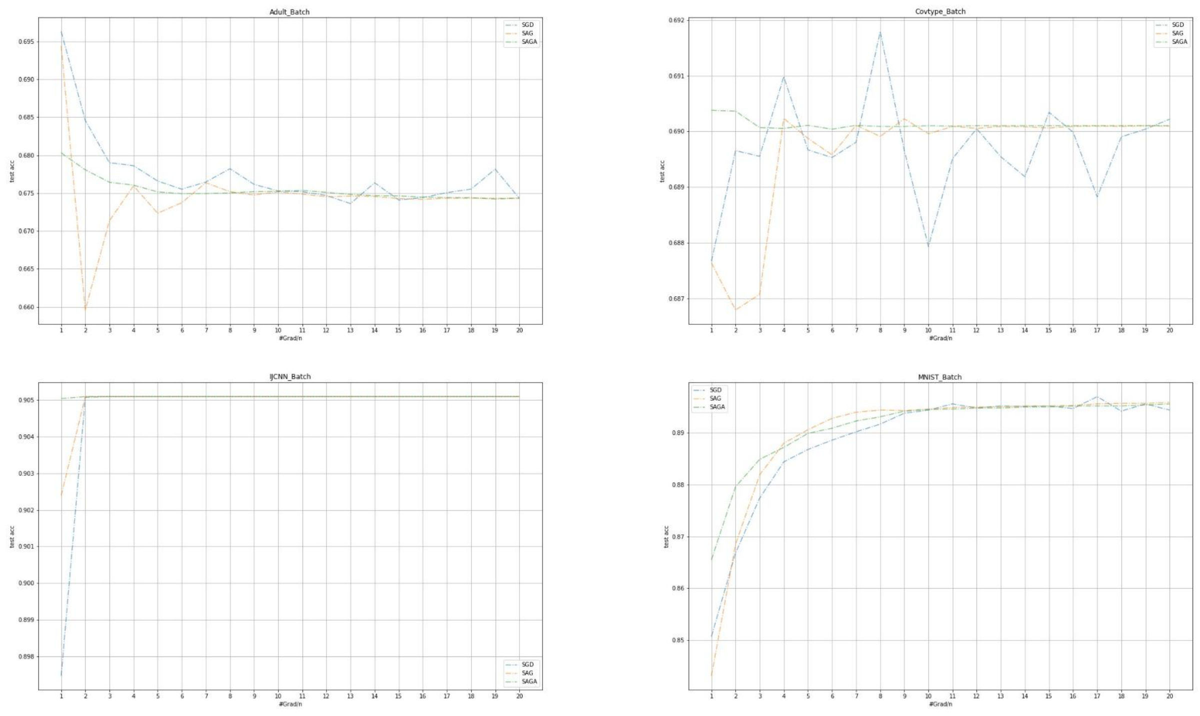

Figure 6. Accuracy of Minibatch-Version with L2 regularization

From the test accuracies, it can be seen that algorithms that show fast convergence (SAG, SAGA) are also prone to overfitting. However, there is self-correction as the algorithms run for more iterations, possibly due to the randomness in the algorithms. Overall, the improvement in test accuracy by VR algorithms is not significant, as a faster convergence to the model parameters that minimize the training loss may result in overfitting of the training data.

6. Conclusion and Discussion

From our experiments, SAGA is the best performing algorithm in most cases, while SVRG provides competitive results at a lower storage cost. While a faster convergence is indeed achieved by the VR algorithms in both single-version as discussed in the referenced paper and minibatch-version implemented by our group, our group determines that the practical value of these algorithms may not be as attractive as their theoretical contribution. On one hand, most commercial packages use minibatch implementation of SGD which already addresses the issue of variance reduction and convergence speed. On the other hand, in real-life situations such as Machine Learning Operations (ML Ops) the generalizing power on unseen data is crucial, and an optimal model for training data does not necessarily provide the best performance on test data.

Additionally, many of the variance reduction methods like SAG, SVRG and SAGA (what we discussed in the report) were raised before the era of Deep Learning. In their original papers, arguments and empirical results are only based on simple linear predictor experiments like linear regression and logistic regression. However, whether these variance reduction methods are still applicable to today's ubiquitous deep neural networks is a doubtful question that is worth thinking about.

References

Babanezhad, R., Ahmed, M. O., Virani, A., Schmidt, M., Konecnˇ y´, J., & Sallinen, S. (2015). Stop wasting my gradients: practical SVRG. *NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems*, *Volume 1*, Pages 2251–2259. https://papers.nips.cc/paper/2015/file/a50abba8132a77191791390c3eb19fe7-Paper.pdf

Blatt, D., Hero, A., & Gauchman, H. (2007). A Convergent Incremental Gradient Method with a Constant Step Size. *SIAM Journal on Optimization*.

Defazio, A., Bach, F., & Lacoste-Julien, S. (2014). SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives. *NIPS'14: Proceedings of the 27th International Conference on Neural Information Processing Systems*, *Volume 1*, Pages 1646–1654. https://arxiv.org/pdf/1407.0202.pdf

Gower, R. M., & Gramfort, A. (2019). *Stochastic Variance Reduced Gradient Methods*. Stochastic Variance Reduced Gradient Methods. Retrieved April 17, 2022, from https://gowerrobert.github.io/pdf/M2_statistique_optimisation/optimization_IV_variance_reduced-expanded.pdf

Johnson, R., & Zhang, T. (2013). Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. *NIPS'13: Proceedings of the 26th International Conference on Neural Information Processing Systems*, *Volume 1*, Pages 315–323. https://papers.nips.cc/paper/2013/file/ac1dd209cbcc5e5d1c6e28598e8cbbe8-Paper.pdf

Schmidt, M., Roux, N. L., & Bach, F. (2016). Minimizing Finite Sums with the Stochastic Average Gradient. *arXiv [math.OC]*. https://arxiv.org/pdf/1309.2388.pdf

Sebbouh, O., Gazagnadou, N., Jelassi, S., Bach, F., & Gower, R. (2019). Towards closing the gap between the theory and practice of SVRG. *NIPS'19: Proceedings of the 33rd International Conference on Neural Information Processing Systems*, *Volume 1*, Pages 648–658. https://arxiv.org/pdf/1908.02725.pdf

Github Repository:

https://github.com/simonCodeZzz/DSA4212_Assignment2