Reproducing Empirical Results and Assessing Theoretical Guarantees of PAGE

Machine Learning Project 2 - Reproducibility Challenge

Francesco Borg, Arturo Cerasi, Davide Mazzali
School of Computer and Communication Sciences, EPFL, Switzerland

Abstract—**Deep learning is a very practical and applied science, hence it is crucial for newly proposed optimization methods, promising to achieve greater efficiency, to be empirically assessed. In particular, the goal of this work is to test the novel Probabilistic Gradient Estimator Algorithm (PAGE [1]), so to verify empirical results and assess theoretical guarantees presented by the authors. Firstly, since there currently is no publicly available implementation, we create a custom optimization method in PyTorch. Then, we use this to reproduce some of the experiments performed by the authors, in particular, comparing PAGE against vanilla SGD on popular classification problems. Moreover, we compare PAGE with ProxSARAH [2], another non-convex optimization method featuring very similar convergence rate in theory. Finally, we evaluate the running time of our implementation. Overall, our results confirm the claims made by the authors stating superior performance of PAGE compared to SGD. In addition, we found ProxSARAH to be comparable to PAGE in terms of training loss matching the authors' claims, even though the former outperforms the latter in terms of accuracy. We also show how implementing the plain theoretical formulation of PAGE does not yield good solutions in practice. Lastly, present interesting findings about the running times of PAGE.**

I. Introduction

Given the wide-spread and increasing employment of deep learning, efficient methods that allow for training massive neural networks are of utmost importance. In particular, in the case of deep learning models, one is often confronted with an optimization problem where the function at hand is highly non-convex. This makes the task much more challenging then in the convex regime, and indeed extensive efforts have been made by the research community to tackle it. Specifically, the training task involves finding the (close to) best parameters for the model at hand as to minimize the training error on a given dataset. Mathematically, this translates to minimizing a non-convex real-valued objective function, which in ML context is usually referred to as *loss function*. More precisely, in this context one is confronted with a function $f : \mathbb{R}^d \to \mathbb{R}$ given by

$$f(w) = \sum_{i \in [n]} f_i(w) \quad \forall w \in \mathbb{R}^d \,,$$

where, for each $i \in [n]$, one also has $f_i : \mathbb{R}^d \to \mathbb{R}$ and f_i being differentiable. In a ML context one can think of each f_i as the training loss on the i-th element of the training set. Then the problem to solve is:

$$\operatorname{argmin}_{w \in \mathbb{R}^d} f(w) \,. \tag{1}$$

Since the training phase is one of the major bottlenecks in deep learning pipelines, many optimization methods have been proposed. In this work, we focus on PAGE [1], a novel optimization algorithm with fascinating theoretical guarantees. It is a variation of vanilla SGD, parametrized by $T, b \in \mathbb{N}^*, p \in (0,1], \eta \in (0,1)$, where T is the number of optimization steps, b the batch-size (upon which a secondary b' will be defined) and η is the step size. For convenience, let us denote $f_I = \frac{1}{|I|}\sum_{i \in I} \nabla f_i$ for all $I \subset [n]$. Then the algorithm can be outlined as follows[1]:

- let $b' = \sqrt{b}$, and take $w^0 \in \mathbb{R}^d$;
- sample b elements from $[n]$ and call it I;
- compute $g^0 = \nabla f_I(w^0)$;
- updates $w^1 = w^0 - \eta g^0$;
- for $t \in [T]$, do the following:
  - toss a p-biased coin;
  - if head, sample b elements from $[n]$ and call them I. Then compute $g^t = \nabla f_I(w^t)$;
  - otherwise, sample b' elements, call them I'. Compute $g^t = g^{t-1} + \nabla f_{I'}(w^t) - \nabla f_{I'}(w^{t-1})$;
  - update $w^{t+1} = w^t - \eta g^t$;

[1]We remark that this is a slightly different, though equivalent, formulation than the one given by the authors, where we fix some of the parameters as they are used in Corollary 2 of their paper [1] and in their experiments. We also simplified expressions in a way that is more convenient to our discussion.

- return $w \sim \text{unif}(\{w^t\}_{t\in[T]})$.

As noted by the authors, setting $p = 1$ makes PAGE the same as mini-batch SGD with batch size b (besides for the last step).

One of the most popular measures for assessing the theoretical effectiveness of optimization algorithms in this context, is the *gradient complexity*, which is the number of gradient computations performed by an algorithm before meeting a stopping criteria or needed to achieve some approximation guarantee. For example for $I \subset [n]$ and $w \in \mathbb{R}^d$, the gradient complexity of computing $\nabla f_I(w)$ is $|I|$. The authors show many such guarantees in their paper, and the one which is most relevant to our work can be restated as follows[2].

Theorem 1: Assume that for all $w, w' \in \mathbb{R}^d$ we have

$$\mathbb{E}_i\left[\left\|\nabla f_i(w) - \nabla f_i(w')\right\|^2\right] \leq L^2\|w - w'\|^2$$

for some $L > 0$. Take $\eta \leq 1/(2L), b = n, p = b'/(b+b')$ (where $b' = \sqrt{b}$ as defined in Section I). Then, in order to find an ϵ-approximate solution $\hat{w}$ to (1), the gradient complexity of PAGE is bounded by:

$$O\left(n + \frac{\sqrt{n}}{\epsilon^2}\right).$$

On top of these theoretical contributions, the authors also give an empirical evaluation of the performance achieved by PAGE compared to vanilla mini-batch SGD, applied to solving classification on images. Hence we tried to reproduce the same findings, and to do so we implemented PAGE in PyTorch. We also empirically investigate how PAGE compares to an existing algorithm with the same guarantee as in Theorem 1. In addition, we assess the effectiveness of returning a random weight (as in the algorithm) compared to returning the last one. Finally, we assess the running times achieved by our implementation and discuss our findings.

II. Implementation

Although authors do experimental evaluation of PAGE, they do not provide code. Therefore, to reproduce their experiments and investigate the practical aspects of PAGE further, we implement PAGE from scratch using PyTorch. In particular, we provide it in two flavours:

1) the first implements the algorithm outlined in Section I plainly, except we return w^T instead of a uniformly random weight from $\{w^t\}_{t\in[T]}$;
2) the second, more practical, follows the established practice in ML of running an optimization algorithm in *epochs*. Each epoch corresponds to the algorithm computing as many gradients as the size of the training set. In the specific case of PAGE, this requires particular attention, given that inside every epoch, gradients are computed on differently sized batches of the training data, depending on the relative coin toss, size b if heads, b' otherwise.

Both our implementations have been tested in the use case of optimizing a loss function (from the `torch.nn` module) over a neural network (extending `torch.nn.Module`), and the training set is provided as a `torch.Tensor`. Moreover, we provide support also to use `cuda`.

Although simple in the idea, and easy to implement disregarding performances, an efficient implementation of PAGE turned out to be non-trivial. In particular, one can see from Section I that PAGE needs to (1) remember previous updates g^t's, and (2) compute gradients of $f_{I'}$ on both current w^t and previous w^{t-1}. This peculiarity hits PyTorch's gradient computation mechanism: whenever a gradient computation is performed on a neural network's loss functions, the previous gradient value gets overwritten by the new one. Also, of course updating weight w in-place (instead of creating a new vector w^{t+1} out of w^t as in the algorithm) overwrites the old value. An easy work-around would be to maintain two neural networks, update the two independently, and use the second one as a back-up for the first one. However, this is not practical, since it would nearly double running times, and double memory usage: undesirable drawbacks for current standards in neural network sizes. To circumvent this issue, we developed PAGE in the following fashion: We start by tossing T coins ahead of the loop over $t \in [T]$, which allows to check at every t if the following iteration $t + 1$ is going to be in the tails case. Imagine that the coin toss $t + 1$ is indeed tails, then:

- At step t:
  - the algorithm makes a temporary copy of the update g^t;
  - compute the gradient of $f_{I'}$ on current weights w^t, using $t + 1$'s batch I';
  - update w^t using g^t;
- At the following step $t + 1$ (tails), the procedure:
  - finds $\nabla f_{I'}$ evaluated at w^t in the `grad` attribute of the neural network weights, as computed y the previous iteration;

[2]Again, we have made a few simplifications that fit our scenario.

– next, computes $\nabla f_{I'}$ evaluated at current weights w^{t+1}
– update the weights using $g^{t+1} = g^t + \nabla f_{I'}(w^{t+1}) - \nabla f_{I'}(w^t)$;
– if $t+2$ is also tails, the same mechanism is applied again.

Further details are deferred to the implementation itself[3].

We remark that our implementation maintains the property that setting $p = 1$ recovers exactly mini-batch SGD with batch size b.

III. Experimental Results

Our experiments have been run on Google Colaboratory.

A. Comparison with mini-batch SGD

In this section we use our implementation to test PAGE against mini-batch SGD, in order to reproduce the authors' experiments. In particular, we focus on the setting of LeNet [3] neural network applied to MNIST dataset [4], using a cross entropy loss function and normalizing the input from the range $\{0, \dots, 255\}$ to $[0, 1]$. We remark that this experiment is practical in nature, in that it aims at investigating how PAGE performs on real world problems where one ultimately wants to maximize test accuracy. Hence we instantiate our *second* implementation of PAGE.

Specifically, we use the same hyper-parameters employed in the authors' experiments. These are batch sizes $b = 64, 256$, and $p = b'/(b + b')$. As long as η is concerned, no hyper-parameter optimization should be needed (as claimed by the authors) in that results of the authors give upper-bounds on the correct value for this hyper-parameter. In fact, they do not state which numerical value they used for η. However, these bounds depend on the smoothness L of the loss function, a quantity which is impractical to calculate in our case. After experimenting, we found $\eta = 0.005$ to give the best learning speed. Interestingly, this was the value we found for both batch sizes 64 and 256, matching the claim in Theorem 1, where one sees that the upper-bound on η indeed does not depend on b.

For mini-batch SGD, the batch sizes and step size used are the same as for PAGE. In fact also the authors claim to have used the same step size for both algorithms. We remark that the implementation of SGD that we used is nothing but our PAGE implementation with $p = 1$, as to ensure consistent results that would not depend on implementation but rather only on the core algorithm.

[3]https://github.com/CS-433/ml-project-2-stochasticgnocchidescent_project2

With this hyper-parameters setting in place, we run each algorithm-batch size combination for a number of times, which we then average. Our findings are plotted in Figures 1, 2, where on the horizontal axis we find the number of gradient computations (divided by the number of training data points), i.e. the gradient complexity.

Fig. 1. PAGE & SGD test accuracy compared on MNIST with LeNet

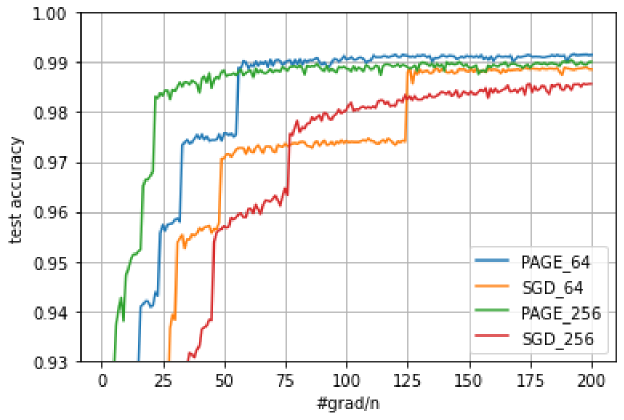
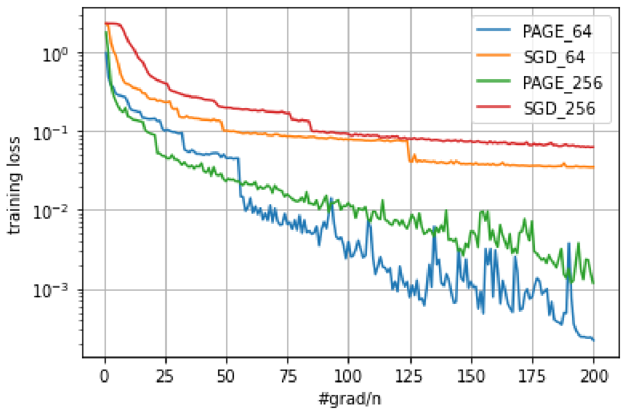


Fig. 2. PAGE & SGD training loss compared on MNIST with LeNet



We are thus able to confirm the main claim of the authors: PAGE outperforms SGD with both batch sizes 64, 256 in terms of training loss and also test accuracy, especially for larger gradient computations, where the gap between the two gets wider. In Figure 1 for large gradient numbers, we see from bottom to top SGD 256, SGD 64, PAGE 256, PAGE 64, which is the same ordering one sees in the authors' plot. Consistently with accuracy, we see the reverse order in Figure 2, which again is the same as in the original paper. Interestingly, also the numerical values that we obtained are very close to those presented by the authors.

While processing the data, we tried to leave out some of the runs where the improvements in accuracy and training loss appeared to come in big steps at selected points. More specifically, we removed those runs where the algorithm got stuck in particular configurations for a while before converging to high accuracy values. What we obtained doing so is shown in Figures 3, 4. Remarkably, our results after this selection, resemble very closely those obtained by the authors, not only in the numerical values, but also in the overall behaviour. Also, in this case it is more clear that the runs feature a much wider gap between PAGE and SGD with $b = 64$ than the one between PAGE and SGD with $b = 256$.

Fig. 3. PAGE & SGD test accuracy compared, keeping smooth runs

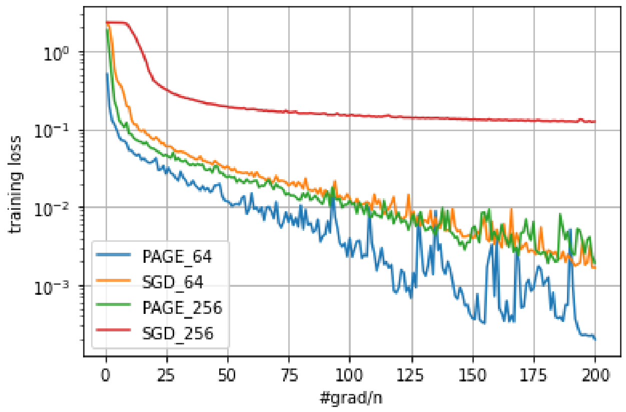
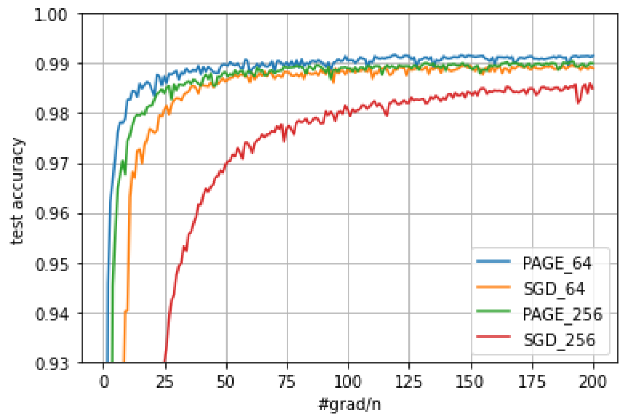


Fig. 4. PAGE & SGD training losses, keeping smooth runs



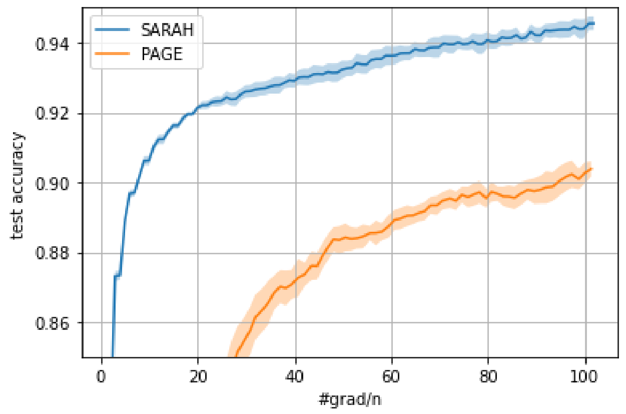
In their work, authors also test PAGE on other datasets (CIFAR10 [5]) and neural networks (ResNet [6], VGG16 [7]). However this turned out not to be practical for our limited computational power.

B. Comparison with ProxSARAH

To take the empirical evaluation of PAGE further, we compared it against ProxSARAH [2], which PAGE's authors mentioned as one of the available non-convex optimization algorithms achieving the same gradient complexity as PAGE (note that among these alternative previously existing methods, ProxSARAH is the only featuring publicly available code). The specific problem we solve in this experiment is that of classifying MNIST images using a simple neural network with a single hidden layer with 100 nodes and ReLu activation function. The code for running ProxSARAH on this problem is already publicly available[4] and is implemented in TensorFlow. We ensured that our implementation and the one provided by ProxSARAH's authors use the same loss function, that is cross entropy, and do the same pre-processing on the dataset, that is ℓ_2 normalization of the raw dataset with entries in range $\{0, \dots, 255\}$.

The purpose of this experiment is to assess if two algorithms featuring the exact same guarantee indeed achieve the same performance. Hence, since this experiment is motivated by theoretical similarities between the two methods, we run it under the hyper-parameter regime where such guarantees hold. Specifically, we will run our *first* implementation of PAGE, that is the one closely mimicking the algorithm, with $b = n$ as per Theorem 1. As long as step size is concerned, we set $\eta = 0.045$ after some trial-and-error. As long as ProxSARAH is concerned, we run their `example_neural_net_1.py` with command line parameters `-d mnist -a 1 -so 2`, which executes a version of their algorithm with hyper-parameters corresponding to those under which their guarantee holds (see Theorems 6, 8 in their work [2]), applied to MNIST dataset on the simple neural network-loss function context described above. Again, we ran each algorithm a few times, and then average the results. On the horizontal axis of our plots we find the gradient complexity.

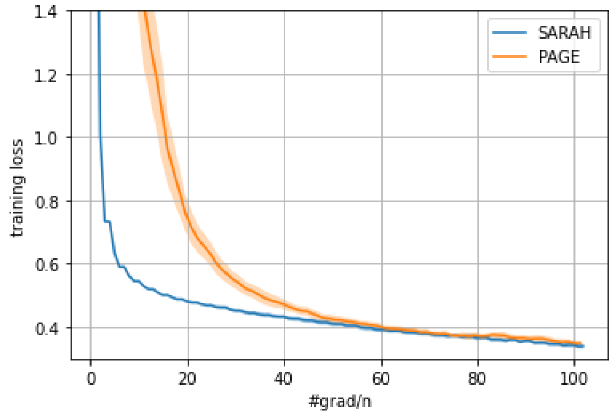
Fig. 5. SARAH and PAGE test accuracy comparison on MNIST with a simple NN made by a single hidden layer of 100 nodes.



Interestingly, ProxSARAH greatly outperforms PAGE in terms of test accuracy, as shown in Figure 5. However, recall once more that the purpose of this experiment is to compare optimization algorithms with the same theoretical guarantee, where the latter speaks about their capability to optimize a function. Hence, we claim

[4]https://github.com/unc-optimization/StochasticProximalMethods

Fig. 6. SARAH and PAGE train loss comparison on MNIST with a simple NN made by a single hidden layer of 100 nodes



that the most relevant result to this experiment is the one comparing the training loss. Despite a considerable gap in the beginning, one can see from Figure 6 that indeed ProxSARAH and PAGE nearly achieve the same values. This empirically confirms that both algorithms are capable of finding ϵ-approximate minima roughly within the same number of gradient computations.

As a side note, we observe from the plots that ProxSARAH is more stable than PAGE in both loss and accuracy: the shaded area represents the 95% confidence interval over the experiments we ran. This area is indeed considerably narrower for ProxSARAH.

C. Returning Random Weights

At this point one might argue that in our experiments in Section III-B the setting was not truly resembling the one where Theorem 1 holds, in that the used implementation returns the last weight w^T instead of a random one. To confirm that our choice is however the most insightful, we run the same experiment in Section III-B comparing our *first* implementation against the one returning a random weight.

To do so, we keep the same hyper-parameters as in the previous section, and run experiments a number of times which we average. Plots 7, 8 show 95% confidence interval over these experiments. One can in fact see how returning random weights generally results in much worse, as well as unreliable, loss and accuracy. From this experiment, it seems that the original formulation of PAGE returning a uniformly random weight from the history is merely a trick useful in proofs (e.g. in the proof of their Theorem 1 [1]), but definitely not practical.

D. Running times

While running our experiments, we noticed the running times of our implementation to behave oppositely

Fig. 7. Test Accuracy of PAGE (1st implementation) returning last & random weights on a 1-hidden layer network

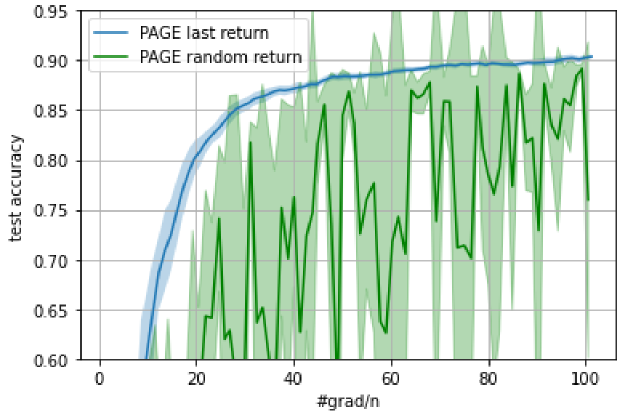
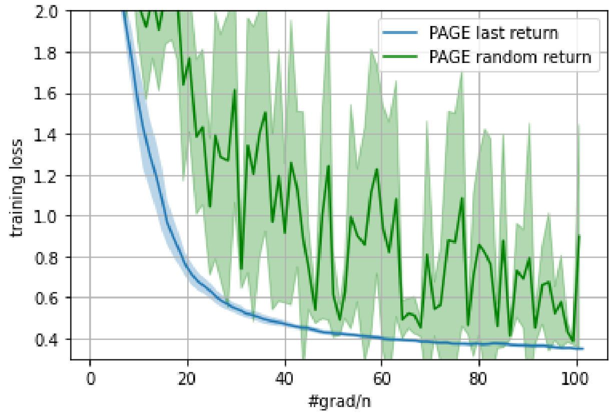


Fig. 8. Training Loss of PAGE (1st implementation) returning last & random weights on a 1-hidden layer network



to what one might expect at first sight. Indeed, one might first think that the smaller the considered batch size, the lower the running time. For the same reason, one should expect SGD to be slower than PAGE for the same value of b, as also suggested by the authors, because SGD deterministically picks batches of size b, whereas the expected batch size for PAGE is

$$O(pb + (1-p)b'),$$

which is smaller than b.

As a benchmark, we measure the time it takes for the various configurations of the considered methods to perform roughly n gradient computation. To do so, we used the LeNet neural network on the MNIST dataset and cross entropy loss function (note that step size is not relevant to this experiment). If the theoretical running time analysis was matched in practice, we would expect to see close figures across different methods. However, the observed running times behave oppositely, as one can see in Tables I, II where the smaller the (expected) batch size, the longer the running time.

Notoriously, when doing matrix-like computing one should avoid loops at all costs, because these are much slower then an equivalent formulation using only matrix operations, since the latter are very optimized in frameworks such as PyTorch. In our case, if the batch size is smaller we need more iterations over the loop for $t \in [T]$ outlined in Section I in order to accumulate roughly n gradient computations, for both PAGE and SGD. Hence, the smaller the batch size, the less we exploit the power of matrix operations in PyTorch and the more loop iterations we perform, resulting in higher running times.

This is a remarkable example of when asymptotic analysis of running times fails to capture the actual behaviour of implementations.

TABLE I
AVERAGE TIME FOR $\approx n$ GRADIENT COMPUTATIONS FOR EACH ALGORITHM, USING GPUS (CUDA)

Algorithm	Avg. Time (sec.s)	95% Confidence Interval
Page ($b = 64$)	30.33	(30.08,30.57)
Page ($b = 256$)	14.93	(14.50,15.36)
Page ($b = n$)	2.48	(1.99,2.96)
SGD ($b = 64$)	3.95	(3.91,3.98)
SGD ($b = 256$)	2.87	(2.83,2.91)

TABLE II
AVERAGE TIME FOR $\approx n$ GRADIENT COMPUTATIONS FOR EACH ALGORITHM, NOT USING GPUS (CUDA)

Algorithm	Avg. Time (sec.s)	95% Confidence Interval
Page ($b = 64$)	37.01	(36.39,37.64)
Page ($b = 256$)	26.48	(25.98,26.99)
Page ($b = n$)	20.75	(18.47,23.03)
SGD ($b = 64$)	14.39	(13.98, 14.80)
SGD ($b = 256$)	12.47	(12.06, 12.88)

IV. CONCLUSIONS

After conducting our experiments and analysing the relative results we can conclude that as claimed by Li, Bao, Zhang and Richtarik in the original paper [1], PAGE algorithm achieves higher effectiveness. In particular SGD is consistently outperformed by PAGE in both accuracy and loss. In addition, from our experiment comparing in identical conditions PAGE against ProxSARAH, we noticed similar train loss convergence, consistently with the identical gradient complexity mentioned in the papers. Moreover, the choice to uniformly sample the weights from all the iterations proves to be a mathematical trick rather than a practical feature of PAGE. Finally, the results we presented regarding the running time of various configurations of PAGE and SGD, shows the fact that asymptotic running time analysis does not always reflect empirical running time, especially when matrix computing is involved.

Future work could extend our result, using other neural network architectures and datasets, to fully replicate the experimental findings of the original paper. In addition, interesting insight could be extracted by exhaustively comparing PAGE to other algorithm with identical gradient complexity, as we did with ProxSARAH.

References

[1] Z. Li, H. Bao, X. Zhang, and P. Richtarik, "Page: A simple and optimal probabilistic gradient estimator for nonconvex optimization," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 6286–6295. [Online]. Available: https://proceedings.mlr.press/v139/li21a.html

[2] N. Pham, L. Nguyen, D. Phan, and Q. Tran-Dinh, "Proxsarah: An efficient algorithmic framework for stochastic composite non-convex optimization," *Journal of Machine Learning Research*, vol. 21, pp. 1–48, 05 2020.

[3] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[4] Y. LeCun and C. Cortes, "MNIST handwritten digit database," http://yann.lecun.com/exdb/mnist/, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[5] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv 1409.1556*, 09 2014.