Increasing the Efficiency of the Sophia Optimizer: Continuous Adaptive Information Averaging

Stanford CS224N Custom Project

Jason D. Lazar
Department of Computer Science
Stanford University
jasonl24@stanford.edu

Caia M. Costello
Department of Computer Science
Stanford University
caia@stanford.edu

Abstract

Pre-training large language models (LLMs) is incredibly time-consuming and costly. Our project aims to improve the efficiency of Sophia, a second-order optimizer proposed by Stanford researchers that achieves 2x convergence efficiency over Adam ([1]). To do this, we propose a new way to estimate the Hessian diagonal that Sophia requires, by collecting the average of the square of the gradients directly. This approach is referenced in the original research, but it was not implemented. This significantly improves Sophia, offering an improved optimizer for language model pre-training. On a 30M GPT2 model, our optimizer (CAIA **C**ontinuous **A**daptive **I**nformation **A**veraging) converges more than twice as fast as Sophia and more than four times faster than AdamW.

1 Key Information to Include

- Mentor: Kaylee C. Burns
- Team Contributions: Caia and Jason worked equally together to produce this report and research. Caia primarily conducted the training process and validated the technical implementation of the optimizer. Jason worked on the presentation of the results and writing the background and related works for the project. That said, many tasks were distributed equally between the team members, and overall they both contributed equally.

2 Introduction

Pre-training large language models (LLMs) is incredibly time-consuming and costly. PaLM cost 10 million dollars for two months of training [2]. Sam Altman estimated that the cost to train GPT-4 was about $100 million.[1] Dario Amodei, the CEO of Antrophic, predicts that single models will cost billions in the next few years.[2] Thus, pre-training efficiency is a major bottleneck for scaling up LLMs. AdamW (and its variants) is the primary optimizer that is used to train the best LLMs to date.

Adagrad [3] and RMSprop [4] introduced the idea of collecting a running average of the square of the gradient and using this average to adjust the step size in updates. The Adam paper explicitly states that this value is an attempt to estimate the *variance* of the gradient. Almost all popular optimizers collect an average of the square of the gradient and use this information in various ways. However, all these optimizers use the average batch gradient, that is, they collect the squares of the batch gradients.

[1] Knight, Will. "OpenAI's CEO Says the Age of Giant AI Models Is Already Over". Wired.

[2] "People have trained models that cost, I think, of order 100 million. But I think billion dollar models will be trained in 2024. And my guess is in 2025, 2026, several billion dollar, maybe even 10 billion models will be trained." https://www.theloganbartlettshow.com/archive/ep-82-dario-amodeis-ai-predictions-through-2030

We propose collecting the average of the square of each individual gradient, using PyTorch hooks, and using this information - the actual variance of the gradient - rather than the approximations earlier approaches have used.

A group of Stanford researchers proposed a lightweight, second-order optimizer called Sophia, or **S**econd-**o**rder Cli**p**ped Stoc**h**astic Opt**i**miz**a**tion [1]. Sophia uses an efficient, stochastic estimate of the Hessian diagonal. This optimization strategy is new, and it outperforms both Adam [5] and Lion [6] by reaching convergence 2x faster. However, these researchers did not compute the Hessian estimate but rather an imperfect approximation.

Our research adapts Sophia and estimates the Hessian diagonal by collecting the average of the square of the gradients directly. By doing this, we achieve 2x better optimization over Sophia and 4x over AdamW. The inspiration for this work was an aside in the Sophia paper. [1]

> "The most natural way to build an estimator for the diagonal of the Gauss-Newton matrix for the Hessian of the mini-batch loss is using
>
> $$\frac{1}{B}\sum_{b=1}^{B}\nabla\ell_{ce}(f(\theta,x_b),\hat{y}_b)\cdot\nabla\ell_{ce}((f(,x_b),\hat{y}_b)$$
>
> where $\hat{y}_b$'s are labels sampled from the model on inputs x_b's respectively. However, as noted by Grosse (2022), implementing this estimator is inconvenient under the current auto-differentiation frameworks, where the users only have access to the average gradient over a mini-batch (as opposed to the individual ones)"

This aside led us to Grosse's draft manuscript of Neural Net Training Dynamics, [7] where he writes (referring to the above expression as equation 2):

> In practice, like SGD, adaptive gradient methods are typically run on batches of data. In principle, the proper thing to do would be to find a way to estimate Eqn. 2 in a way that uses batch operations. Unfortunately, this is hard to do in practice, so instead one simply substitutes the batch gradients for the per-example gradients in Eqn. 2.

He then adds:

> "Given that there are such profound differences between the empirical and true Fisher matrices, as well as between the per-example and per-batch versions of the empirical Fisher, you'd expect someone to have measured the effects of these various choices on optimization performance. But to the best of my knowledge, this hasn't been done. Good opportunity for a class project!"

The Sophia paper suggests that collecting the average of the squares is inconvenient, and Grosse suggests it is "hard to do in practice", as well as a good idea for a project, so we aim to improve Sophia by collecting the average of the squares of the gradients as proposed by Grosse and the authors of the Sophia paper

3 Related Work

Stochastic Adaptive First Order Optimizers: Adagrad was first optimizer that introduced the idea of adaptive learning rates in 2011 to increase optimization efficiency. [8] *Root Mean Squared Propagation* (RMSProp) added this adaptive information to the earlier RProp approach. [4] *Adaptive Moment Estimation*, or (Adam), was introduced in 2014 [5].

Adam and its variants are accepted as the industry standard optimizer for language model pre-training since they improve the efficiency of fixed learning rate methods like stochastic gradient descent (SGD) while remaining relatively cheap in terms of computational cost. Adam adjusts the learning rate for each parameter based on a moving average calculated for each parameter throughout the optimization. This effectively significantly improves the efficiency of convergence. Adam is a first-order optimizer that can be improved using second-order techniques.

Second Order Optimizers: There has been a breadth of research devoted to addressing the major bottleneck of language model pre-training to reach convergence faster through second-order optimization techniques. This often involves computing and leveraging an estimate for the Hessian matrix along with the gradient.

The original Sophia research that inspired our project is a continuation of prior research into LLM pre-training efficiency efforts. [1]. For example, classical second-order optimization algorithms pre-condition the gradient with curvature information [9][10]. Researchers have used second-order optimization techniques to apply them to deep learning in numerous ways. For example, in 2021, Jahani et. al (2021) proposed to use the EMA of diagonal Hessian estimator as the pre-conditioner [11]. The researchers in our paper cite many more papers that aim to improve the efficiency of training by using the Hessian Estimate, all with the same goal to create better optimizers for LLMs.

As mentioned in the original Sophia research paper, despite the improvements over first-order optimizer, Adam remains the most popular in the field of deep learning because approximating the Hessian consumes a considerable amount of wall clock time and computational resources, often requiring specific model architecture or hardware structures.

4 Our Approach

As we have discussed, optimizers use the batch gradient, the sum of the gradient over a batch, and this is coded into Pytorch with the autograd system.

To estimate the variance of the gradient, optimizers collect a moving average of the square of this batch gradient. Commonly, this is collected with a beta value of 0.99, so this estimate is made over hundreds of steps. However, it is possible to collect the variance directly by summing the **square** of each gradient in the same way that the gradients are summed.

The exponential moving average will eventually converge to the actual variance if the variance does not change over time, but it will do so slowly. Furthermore, the estimate is very slow to adapt when the variance changes. **Our method collects the actual variance immediately.**

Adagrad, RMSprop, and Adam use the square of the gradient to adjust the step size. Sophia uses the square of another gradient (over labels sampled from the model), which better approximates the Hessian. This other squared gradient is an improvement, so in what follows, we collect the square of the gradients over the loss from labels sampled from the model (what Grosse calls the true Fisher information).

Algorithm 1 Sophia

Input: θ_1, learning rate $\{n_t\}_{t=1}^{T}$, hyperparameters $\lambda, \gamma, \beta_1, \beta_2, \epsilon$, and Estimator $\in$ {Hutchinson, Gauss-Newton-Barlett}
Set $m_0 = 0, v_0 = 0, h_{1-k} = 0$
for $t = 1$ to T **do**
  Compute minibach loss $L_t(\theta_t)$
  Compute $g_t(\Delta\theta_t)$
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  if t mod $k = 1$ **then**
    Compute $\hat{h}_t =$ Estimator(θ_t)
    $h_t = \beta_2 h_{t-k} + (1 - \beta_2)\hat{h}_t$
  else $h_t = h_{t-1}$
  end if
  $\theta_t = \theta_t - \eta_t \lambda \theta_t$(weight decay)
  $\theta_{t+1} = \theta_t - \eta_t \cdot \text{clip}(m_t / \max\{\gamma \cdot h_t, \epsilon\}, 1)$
end for

The focus of our project is on computing $\hat{h}_t$, or the Hessian estimate differently than Sophia. The researchers in the original paper used the Gauss-Newton-Barlett (GNB) estimator to compute $\hat{h}_t$ (among other methods), and maintain a rolling average that is incorporated in the parameter update.

The GNB has a loose estimation of the Hessian where the square of the summed gradients is used as an estimate of the sum of the squared gradients. Approximating like this slows convergence.

We propose collecting the average of the square of the gradients directly, which should improve the efficiency of optimization. Our project uses most of the same experimentation and methods described in the original Sophia research paper, namely nanoGPT, Karpathy's[12] re-implemention of GPT2.

4.1 Preliminary Explanation of Sophia with an Emphasis on $\hat{h}_t$

In Algorithm 1, θ_t is a given parameter at time step t. At each step, they sample a mini-batch from the data distribution and calculate the mini-batch loss, $L_t(\theta_t)$. g_t is the gradient of $L_t(\theta_t)$, i.e. $g_t = \nabla L_t(\theta_t)$. m_t is the Exponential Moving Average (EMA) of gradients, $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$, which is the numerator of the update.

At time step t with $t \mod k = 1$, the estimator returns an estimate $\hat{h}_t$ of the diagonal of the Hessian of the mini-batch loss. The Sophia algorithm computes an estimate of the diagonal of the Hessian by computing the batch loss over the gradients of labels sampled from the model.

They call this $\hat{L}(\theta)$ and define it as

$$\hat{L}(\theta) = 1/B \sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta x_b), \hat{y}_b)$$

Sophia updates the Hessian every k steps, resulting in the following update rule for the diagonal Hessian estimate:

$$h_t = \beta_2 h_{t-k} + (1-\beta_2)\hat{h}_t \text{ if } t \mod k = 1; \text{ else } h_t = h_{t-1}.$$

4.2 Sophia's Original Equation for the Hessian Approximation

As mentioned previously, the Sophia paper states $\hat{L}(\theta) = \frac{1}{B}\sum_{b=1}^{B} \ell_{ce}(f(\theta, x_b), \hat{y}_b)$ is the mini-batch loss on the *sampled* labels (as opposed to the original labels). Thus,

$$\mathbb{E}_{\hat{y}_b' s}\left[\nabla \hat{L}(\theta) \odot \nabla \hat{L}(\theta)\right] \approx \frac{1}{B}\sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta, x_b), y_b) \odot \frac{1}{B}\sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta, x_b), y_b),$$

The researchers argue that in terms of expectation

$$\frac{1}{B}\sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta, x_b), y_b)^2 \approx (\frac{1}{B}\sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta, x_b), y_b))^2$$

However, we argue that this approximation is unnecessary and the original expression can be computed efficiently.

4.3 Our Equation

For our project, we use the estimate that the researchers would use if they had access to the individual gradients in training. Instead of an approximation, we compute the expression itself:

$$\hat{h}_t = E[Hess] = \frac{1}{B}\sum_{b=1}^{B} \nabla \ell_{ce}(f(\theta, x_b) y_b)^2,$$

We use this as the Hessian approximation instead of doing a momentum solution.

We intend to find out if this leads to convergence efficiency gains. This is because, as mentioned in the original research "given a mini-batch of inputs $\{(x_b, y_b)\}_{b=1}^{B}$, the above formula is the most natural way to build an estimator for the diagonal of the Gauss-Newton matrix for the Hessian of the mini-batch loss"

4.4 Collecting Individual Squared Gradients

Most of our code is taken from the repository from the original Sophia research paper ([1]), which in turn is a small modification of nanoGPT. However, in order to collect the gradients necessary for our Hessian approximation, we implemented PyTorch hooks to gather the values in the inputs/outputs from the forward and backward passes during training.

Machine learning frameworks like Pytorch have tools that allow the collection of the summed batch gradient over a set of samples. However, Pytorch also allows hooks on each module that expose the input and output gradients for each layer for both the forward and backward passes, and by leveraging these we can compute the summed squared gradients efficiently. We implement a new backward pass for all of the layers which computes the sum of the squared gradient. We first cache the input values for each forward layer.

```
def fc_forward_hook(module, name, m_input, m_output):
with torch.no_grad():
    if enable_hooks:
        finputs[name] = m_input[0]
```

When we are called from a backward layer, we first square the input for linear layers to the forward layer and then the output gradient. Finally, we take their matrix product. This yields the sum of the squared gradients. For LayerNorm layers, Linear layers and the embedding layers WPE, WTE we compute the gradient and its square. These backward passes are all different, but for an example of how we do this, let's take the RMSnorm[3] layers we have

$$g = \frac{\partial L}{\partial y_i} \cdot \frac{1}{\sqrt{\frac{1}{N}\sum_{j=1}^{N}(x_j)^2 + \epsilon}}$$

where

g is the gradient L is the loss function, x_i is the input to the RMSNorm layer.
y_i is the output of the RMSNorm layer, N is the number of inputs to the RMSNorm layer, ϵ is a small constant added for numerical stability, The term $\frac{\partial L}{\partial y_i}$ is the gradient of the loss with respect to the output of the RMSNorm layer (i.e., the grad output which is passed as an argument to the hook).

The backward hook code for RMSNorm looks like

```
def fc1_backward_hook(module,name, grad_input, grad_output):
with torch.no_grad():
    if enable_hooks:
        hess = hesses[name] #state["hess"]
        fin = finputs[name] # state["last_grad"]
        if  None != grad_output[0] and module == 'RMSNorm':
            mean_x2 = (fin  ** 2).mean(dim=[2], keepdim=True)
            x_norm = fin/ torch.sqrt(mean_x2  + 10e-7)
            grad = x_norm *  grad_output[0]
```

[3]The GPT2 model does not use RMSNorm, but more modern transformer implementations have replaced the more complicated LayerNorm with RMSNorm. We use LayerNorm in our experiments, but present RMSNorm here due to its simplicity.

```
gradsq = grad **2
if no_square: # don't square the gradients for testing.
    sum_gradsq = grad.sum([0,1])
else:
    sum_gradsq = gradsq.sum([0,1])
hess.add_(sum_gradsq)
```

4.5 Validating Correct Gradient Estimates: A Sanity Check

In our implementation, we rigorously validated that our gradient estimates are accurately computed. In order to do this, we compared the autograd gradients `p.grad` vs our computations.

Our code prints out the difference between `p.grad` and our computations to ensure that it is small, and that our computation checks out correctly. Here is an excerpt from the printout for running the model with `no square` turned on:

```
max diff  1.4551915228366852e-11
name:  transformer.h.4.ln_1.weight
max diff  3.183231456205249e-12
name:  transformer.h.4.ln_2.weight
max diff  1.5802470443304628e-11
name:  transformer.h.5.ln_1.weight
```

As you can see, the difference between our gradients and the autograd gradients is incredibly small, which is a very good thing. It validates that our computations are correct. The small differences are due to rounding.

5 Experiments

5.1 Baselines

Though covered in other sections, it is worth mentioning explicitly that the baselines for evaluation is the Adam optimizer, as well as the Sophia optimizer presented in the original paper. We ran all of them on the same architecture and size model, GPT-2 30M. We are comparing validation loss and speed of convergence (more on this in section 2.3). By comparing our new optimizer to these baselines, we will be able to assess whether our efforts were worth it to improve pre-training efficiently.

5.2 Data

The dataset we are using to evaluate our optimizer remains the same as the original Sophia research, this is because we wanted to see our preformance againsty the same data, and we also ran them on wandb and on a H100. We are training GPT-2 (nanoGPT's reproduction of the 30M model), using our updated optimizer on OpenWebText([13]). We ran it on a single 8XA100 40GB node.

5.3 Evaluation method

We make sure that we are computing the backward pass correctly by collecting the plain sum of the gradients, (as opposed to the sum of the squared gradients) for testing and seeing if this matches the PyTorch autograd values.

We also graph the convergence of this optimizer alongside others, in wandb to see if convergence is affected. We expect to get a closer estimate of the variance, but maybe have slower convergence time because of the added compute.

5.4 Experimental details

We trained 30M GPT-2 on AdamW, Sophia and our new optimizer, CAIA on an H100.

Optimizers:	Adam	Sophia	Caia
ρ	n/a	0.03	500
Size	30M	30M	30M
weight decay	1e-1	2e-1	2e-1
lr	6e-4	3e-4	3.2e-3
steps	50k	50/100k	25k
train time	32hrs	34hrs	35hrs

We ran these on an H100 computer, and all of them on the same model architecture of GPT-2, karpathy's nanoGPT.

For more info on the GPT-2 model, the GPT-2 model contains 6 Transformer decoder blocks. Each decoder block includes a multi-head masked attention layer, a multi-layer perceptron layer, normalization, and dropout layers. We do not use dropout during pre-training.

batch_size = 8
block_size = 1024
gradient_accumulation_steps = 6
total_bs = 480
n_layer = 6
n_head = 8
n_embd = 256

5.5 Results

Our optimizer, CAIA, reached 3.55 loss in less steps than both AdamW and Sophia. We ran Sophia twice, once with 50k steps and once with 100k steps. We ran Adam to 50k steps and our optimizer to 25k steps.

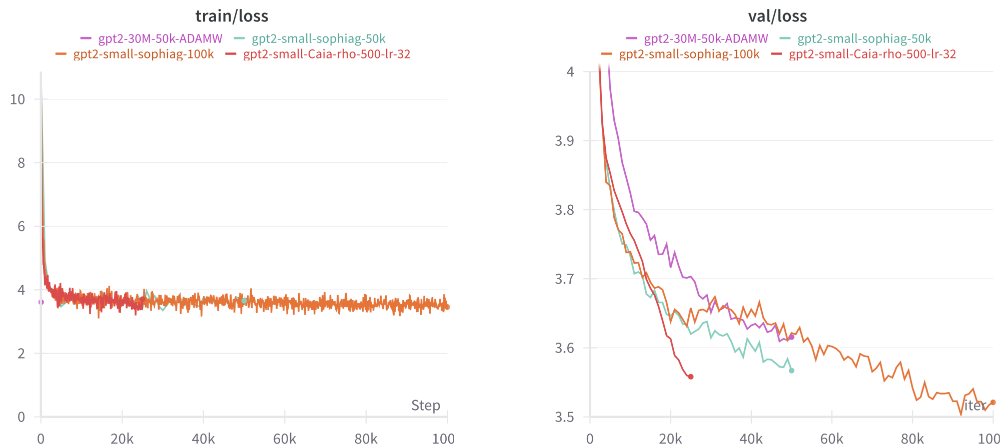

Figure 1: The training loss and the validation loss plots

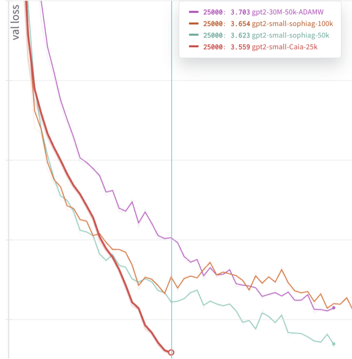

Figure 2: Plots for 25k steps

Optimizers:	Adam	Sophia	Caia
loss 10k	3.82	3.74	3.76
loss 25k	3.703	3.65	3.559
loss 50k	3.615	3.6	n/a
loss 73k	n/a	3.557	n/a

Figure 3: Table of loss for 30M models at different step counts

- Our validation loss at 25k steps outperformed that of Sophia and Adam at 50k steps.

- Our optimizer reached 3.559 loss at 25k steps and it took Sophia 73000 steps to reach the same loss. This is striking as we were able to almost 3x Sophia's convergence on small model.
- An issue we saw was that it did take a long time, but it was made up in convergence speed. Also because we used hooks, the model could not be compiled, so the run-time was slower.

6 Analysis

By the scaling hypothesis, whereby adding more parameters, layers, or data—the model's performance generally improves. This means that this optimizer could speed up training by a factor of three. As billions will be spent on training large language models, this means saving billions in compute costs.

Our optimizer was built on the assumption that using a better estimate for the Hessian we could avoid oscillations. Our graph seems to go mostly straight downwards, whereas Adam and Sophia had a bit more oscillation and this could be due to our enhanced estimate.

Our optimizer was slower to compute per step due to our use of hooks, which prevent compilation, which was expected, but we converged in fewer steps.

7 Conclusion

This research required a significant amount of computing power and resources to train even the 30M model. Moving forward, we desire to research whether our optimizer works better than Sophia and Adam on the larger models presented in the original research, and what that would look like in practice. Overall, it was fascinating to learn the specifics of optimization techniques and optimizers through our work. Sophia is at the cutting edge of optimizer research, and we hope that our results demonstrates an advancement in the field, especially in Natural Language Processing (NLP).

References

[1] Hong Liu, Zhiyuan Li, and Percy Liang. Sophia: A scalable stochastic second-order optimizer for language model pre-training. In *Association for Computational Linguistics (ACL)*, 2018.

[2] Narang S. Devlin J. Bosma M. Mishra G. Roberts A. Barham P. Chung H. W. Sutton C. Gehrmann S. et al. Chowdhery, A. Palm: Scaling language modeling with pathways. 2022.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. volume 12, pages 2121–2159, 2011.

[4] Srivastava N. Hinton, G. and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. 2012.

[5] Ba Jimmy Kingmar, Diederik P. Adam: A method for stochastic optimization. 2014. Accessed 12 Mar. 2024.

[6] Lizhang Chen, Bo Liu, Kaizhao Liang, and Qiang Liu. Lion secretly solves constrained optimization: As lyapunov predicts. 2023.

[7] Roger Grosse. CSC2541 Winter 2022 Topics in Machine Learning: Neural Net Training Dynamics. In *CSC2541 Winter 2022*, 2022. Accessed 13 Feb. 2024.

[8] H. Braun and M. Riedmiller. Rprop: a fast adaptive learning algorithm. In *In Proceedings of the International Symposium on Computer and Information Science VII*, 1992. Accessed 12 Mar. 2024.

[9] C. G. Broyden. The convergence of a class of double-rank minimization algorithms. 1970.

[10] Y. Nesterov and B. T. Polyak. Cubic regularization of newton method and its global performance. 2006.

[11] Rusakov S. Shi Z. Richtárik P. Mahoney M. W. Jahani, M. and M. Taká˘c. Doubly adaptive scaled algorithm for machine learning using second-order information. 2021.

[12] Andrej Karpathy. nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts. GitHub, 2023.

[13] A. Gokaslan and V. Cohen. Openwebtext corpus. 2019.

A Appendix

B Criticism of Sophia Implementation

During our deep study of Sophia and it's code we discovered some issues.

The Sophia algorithm computes an estimate of the diagonal of the Hessian by computing the batch loss over the gradients of labels sampled from the model. They call this $\hat{L}(\theta)$ and define it as $\frac{1}{B}\sum l_{ce}(f(\theta, x_b), \hat{y}_b)$. Naively, one might assume that `grad` is $\sum l_{ce}(f(\theta, x_b), \hat{y}_b)$, the sum of the gradients over the entire batch.

However, nanoGPT and the Sophia code that uses it do not compute the batch in one go. Rather, they use gradient accumulation steps. In the default configuration, Sophia uses 6 gradient accumulation steps per GPU. This is multiplied by 8 when running on one GPU, so a single GPU uses 48 steps.

The summed gradient that nanoGPT and Sophia compute is the average of the gradient for a single mini-batch of size 1024 tokens by a mini-batch size of 8. This is 8192 tokens. The code adjusts the loss (by dividing by the number of gradient steps) so that the final summed gradient is the average gradient for a mini-batch. Furthermore, the Cross-Entropy loss averages the gradients on the last step when it does an all_reduce and copies the gradients from each GPU. Thus, in both the single GPU and multiple GPU cases, the value computed is the average (over the number of gradient accumulation steps) of the sum over a mini-batch of size 8192.

The Sophia code treats the final gradient as the sum of over half a million tokens (48 * 8192) rather than the average gradient of a minibatch of 8192 tokens. On lines 199 and 194 of sophiag.py, the gradient (over the sampled labels) is multiplied by `bs`, a value passed into step (whose default value is 5120). However, this default is overridden on line 309 of train_sophia.py, where it is set to "`bs=total_bs * block_size`". The `block_size` is 1024 (the number of tokens in a document) while `total_bs` is set to be 480 (on line 34 of train_sophia.py) and over-ridden by the config file to be 480. In a comment, 480 is explained to be 6 (the number of gradient accumulation steps) times 8

(the batch size) times 10 (the number of GPUs). Perhaps, 10 is a typo, as in other places, the number of GPUs is taken to be 8, most notable on line 86 of `train_sophia.py`.

However, to match the theoretical design, the summed gradient (over the total batch size $B = 48 * 8192$) should be divided by $1/B$ times the square of the summed gradient over the sampled labels. Thus, we should multiply the gradients by the number of gradient accumulation steps, and then divide the Hessian estimate by the total batch size, not multiply it.

We have `grad`, the average (over 48 mini-batches) of the summed gradient over a minibatch (of size 8192). To get the total summed gradient, we must multiply by the number of gradient accumulation steps (48). We also have $\hat{g}$ the average summed gradient over a mini-batch of sampled labels. To get the correct value for the square of the summed gradient over the sampled labels, we must multiply $\hat{g}^2$ by 48^2.

Finally, we divide the square by the total batch size $(48 * 8192)$ to get the Hessian estimate. The paper writes $B * \hat{L}(\theta) * \hat{L}\theta$ but each $\hat{L}$ is $1/B$ times that summed gradients, so we end up with $\frac{1}{B}(\sum l_{ce}(f(\theta, x_b), \hat{y}_b))^2$. This gives the formula:

$$\frac{48 * g}{\frac{1}{48*8192} * 48^2 * \hat{g}^2}$$

We can cancel two pairs of 48s to get:

$$\frac{8192 * g}{\hat{g}^2}$$

The Sophia code uses `(exp_avg.abs() / (rho * bs * hess + 1e-15))`, or in our terms:

$$\frac{g}{\rho * 480 * 1024 * \hat{g}^2 + 1e - 15}$$

This differs from the paper's proposal as it is 4 billion times too large. As this is adjusted by the factor ρ this does not change the actual experimental results, but demonstrates a lack of understanding of the autograd system.