A Comparative Study of Preconditioner Update Strategies in the SOAP Optimizer

Herman Efraimsson 404233, Ryosei Okamoto 395004, Riccardo Piantoni 403996
School of Computer and Communication Sciences, EPFL

***Abstract*—Preconditioners improve first-order optimizers by incorporating second-order information, but often require expensive computations like matrix inversions and decompositions. To save time and resource costs, preconditioners are typically updated at fixed intervals.**

The SOAP optimizer, a recent method inspired by Shampoo, shows stable performance even with less frequent updates, unlike the latter. However, previous work only considered fixed update frequencies: we investigate alternative update scheduling strategies for the frequency of updating SOAP's eigenspace. Our experiments show that, in specific settings, choosing the right schedule for the update frequency can reduce computational costs while maintaining or improving training performance.

I. Introduction

In recent years, preconditioners have gained attention for enhancing first-order optimizers by improving gradient quality — modifying its direction, magnitude, or both. While classical methods like Newton or quasi-Newton achieve this, their high computational and memory demands make them impractical for large-scale models. As a result, research has shifted toward more efficient preconditioners.

Despite these advances, computing such preconditioners still involves expensive operations (e.g., root inverse matrices, SVD, QR decompositions). To reduce overhead, implementations often avoid frequent recomputation by exploiting the relative stability of gradient statistics between steps. A common strategy is to update the preconditioner periodically, introducing a new hyperparameter: the update frequency.

This project investigates how this frequency affects optimizer performance, and whether it can be tuned or scheduled similarly to how the learning rate is. We focus on the SOAP optimizer, known for its effectiveness in training deep neural networks.

II. Model and Method

A. Optimizer

We consider, the second-order optimization method, SOAP [1], which builds upon the Shampoo algorithm [2]. The key idea behind Shampoo is to approximate the Hessian H using a preconditioner constructed from the Kronecker products of gradient matrices[3]:

$$H^2 \propto \mathbb{E}[GG^T] \otimes \mathbb{E}[G^T G],$$

where $G \in \mathbb{R}^{m \times n}$ denotes the gradient matrix.

It has been shown that Shampoo is equivalent to performing Adafactor [4] in the eigenspace of the second-moment matrices of the gradients, $L = \mathbb{E}[GG^T]$ and $R = \mathbb{E}[G^T G]$[1]. In contrast, SOAP performs Adam [5] which enables more general updates than Adafactor in this eigenspace. Unlike Shampoo, SOAP continues to update the preconditioner using Adam even at steps when the eigenspace is not refreshed. Since computing eigenvectors is computationally expensive, SOAP schedules eigenspace updates to reduce this overhead. Accordingly, the parameters are updated as described in Algorithm 1.

B. Related work

As noted by Anil et al.[6], the most computationally expensive operation in Shampoo is updating the preconditioners. They evaluated Shampoo with different update frequencies and reported that its performance degrades when the update interval becomes too large. Vyas et al [1] compared SOAP with fixed eigenspace update intervals to Shampoo with preconditioner updates at the same frequency. Their experiments demonstrated that although SOAP's performance deteriorated when the eigenspace was updated infrequently, the degradation was less severe compared to Shampoo under the same conditions. In addition, Shi et al. [7] proposed delaying preconditioning during the early phase of training and adopting the grafted method to mitigate the adverse effects caused by stale preconditioners. However, all of these studies adopted fixed-interval update schemes and, to the best of our knowledge, no prior work has explored the use of dynamic or adaptive update schedules. Therefore, in this work, we investigate more flexible eigenspace update strategies in the SOAP algorithm to improve optimization performance.

C. Schedules

We test and compare 5 different frequency schedules to determine when to update the preconditioner. All the associated pseudocode algorithms are shown in Appendix A.

1) *Constant frequency:* First of all, we test different fixed values for the preconditioner update frequency, supporting the claims from related works with experimental evidence. We choose the values 1, 10, 100 and 300.
2) *Doubling frequency:* We conjecture that the preconditioner is more useful early on in training, instead of when the training has stabilized. We therefore double our frequency f every time that we perform an update, resulting in updates to the preconditioner getting less and less frequent as the training progresses. Eventually, we clip the updates in order to prevent them from getting too sparse, testing 1024 and 256 as clip values.

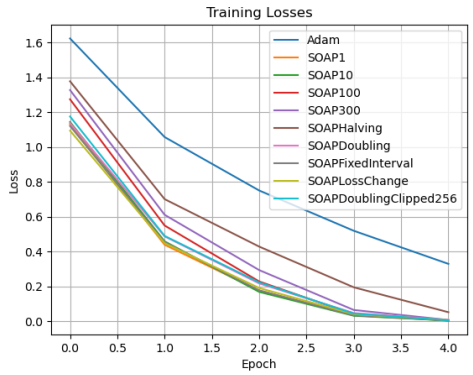

Fig. 1. ResNet18 trained on CIFAR-10 using ADAM and SOAP with different update strategies.

3) *Halving frequency:* We hypothesize that the opposite is true, namely that having an up-to-date preconditioner is more useful later on in training.
4) *Fixed-interval doubling frequency:* We divide the iterations in fixed-size slots, so that the frequency gets doubled at the end of each of them. In each slot, the preconditioner gets updated more than once according to the current frequency, which stays the same throughout the whole slot. We choose an interval size of 256 to analyze how it behaves.
5) *Loss-dependent frequency:* The idea we propose is that it is not necessary to update the preconditioner until the loss landscape changes significantly. Therefore, we test a routine in which the preconditioner is updated whenever the measured loss function has changed more than a certain threshold from the last update. In such a setting, the frequency of the updates behaves like a "sensitivity" value for the change in the measured loss function: we choose the value 0.3 for our tests.

D. Experimental Setup

To evaluate the behavior of different strategies in a real-world scenario, we conducted a simple image classification task using the CIFAR-10 dataset, which includes 50,000 training and 10,000 test images of size 32×32 across 10 classes. The chosen architecture is a ResNet-18 CNN trained from scratch, modified to accept 32×32 inputs and output 10 classes through a final linear layer. Training was performed with a batch size of 200 over 5 epochs, and with a learning rate of 0.005 (scheduled to decrease via cosine annealing).

To induce a more dynamic loss landscape and stress the update strategies more, we then introduced in the CNN dropout and stochastic depth (each with a 0.3 probability), which randomly drop neurons or layers per iteration using a Bernoulli distribution. This causes frequent changes in network architecture and loss curvature, along with the need for more frequent preconditioner updates. Due to the increased stochasticity in this new setting, we trained 5 identical models per optimizer configuration, and averaged their results for more reliability.

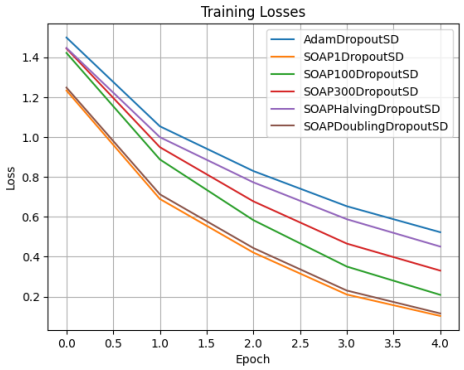

Fig. 2. ResNet18 w/dropout+SD trained on CIFAR-10 using ADAM and SOAP with different update strategies - average training loss per model configuration, averaged then again per epoch - configurations' batch 1

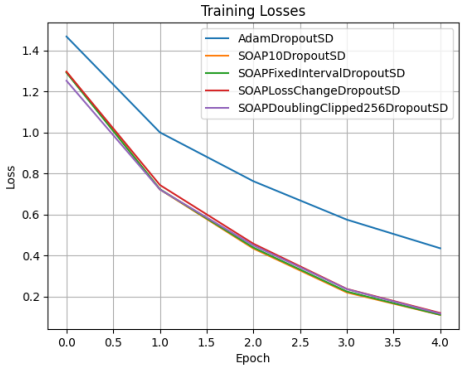

Fig. 3. ResNet18 w/dropout+SD trained on CIFAR-10 using ADAM and SOAP with different update strategies - average training loss per model configuration, averaged then again per epoch - configurations' batch 2

As a baseline, we use the Adam optimizer, along with SOAP running different constant preconditioner update frequencies whose performance has already been considered, but not shown, in earlier works [1].

III. Results

First of all, we ran SOAP training with all the update strategies on a standard ResNet18, with the result shown in Figure 1. We can observe that our update strategies all performed better than Adam, but had little effect on the resulting convergence speed on the chosen network architecture when compared to using SOAP with a constant frequency.

On the other hand, when applying the different schedules to a more dynamic setting such as the ResNet18 with dropout and stochastic depth, the update frequency of the preconditioner becomes more and more significant in both the speed of the training dynamics and the overall time elapsed. As Figures 2 and 3 show (with results being split in order for the plots to be more clear), the SOAP optimizer still manages to outperform ADAM with any update frequency; however, it becomes very clear that some strategies, such as continuously updating the preconditioner, or doubling the frequency, or updating based on the change in the measured loss function, lead to faster

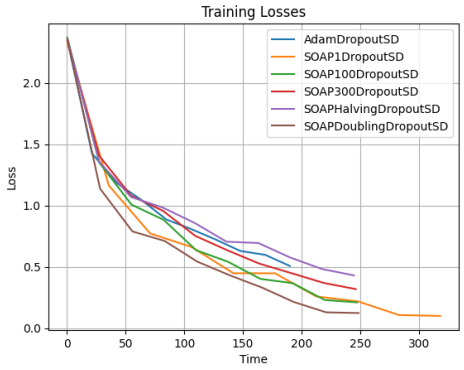

Fig. 4. ResNet18 w/dropout/SD trained on CIFAR-10 using ADAM and SOAP with different update strategies - average training loss per model configuration, sampled every half epoch, over the time elapsed - configurations' batch 1

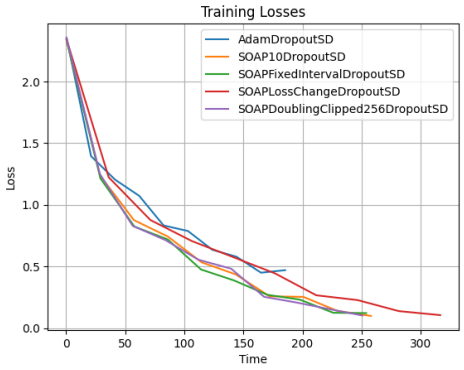

Fig. 5. ResNet18 w/dropout/SD trained on CIFAR-10 using ADAM and SOAP with different update strategies - average training loss per model configuration, sampled every half epoch, over the time elapsed - configurations' batch 2

training speed. On the contrary, halving the frequency after each update has led to significant performance degradation.

Nonetheless, it is important to make a distinction between the routines which manage to achieve better training efficiency without also requiring a significant additional time effort, and the ones which don't. We can observe from Figures 4 and 5 that the continuous updates (with constant frequency equal to 1) and the loss-dependent updates do take more time to perform the same iterations, even though they figure among the lowest training losses observed. The doubling frequency, instead, is as efficient as them, even without requiring considerably more time than other schedules performing way worse, such as updating the preconditioner every 300 steps. This might be a good tradeoff when compared to the ADAM optimizer, as we can see that it still takes a little less time to compute the same number of iterations, but has double the training loss with respect to other update strategies.

IV. Discussion

When using our update strategies to train an ordinary ResNet18, we saw little to no improvement in the training dynamics when changing the frequency of the preconditioner updates, no matter the chosen frequency or schedule. We conjecture that this is likely a consequence of the problem setting being too simple, which might lead to the curvature of the loss function being stable between iterations, and therefore updating the preconditioner more or less frequently doesn't result in relevant performance differences in any case (aside from standard training noise).

However, after introducing dropout and stochastic depth, the differences in the choices of the frequency schedule and hyperparameters become substantial, either accelerating or slowing training down by affecting the total number of iterations needed to reach the same training loss. In particular, with constant frequencies, performance is greatly affected by the frequency value: continuous updates yield the best loss, but take more time when compared to updating only once every 100 or 300 steps, which is nevertheless more time-efficient. This implies that, in such quickly-shifting settings, we cannot use stale conditioners lightheartedly, confirming that dynamic alterations of the landscape of the loss function during training magnify the variability of the preconditioners. Moreover, we have observed that doubling the frequency at each update yields comparable performance to updating the preconditioner at every iteration, while halving it after each update leads to a great loss of training efficiency. This result suggests that having an up-to-date preconditioner is more important towards the start of training than at the end, at least for the SOAP algorithm: the reason might lie in the fact that preconditioners are scaling and rotating the basis space in a way that reduces the "skewedness" of the function, speeding up progress along "unbalanced" directions.

Also, updating the preconditioner depending on the change in magnitude of the measured loss function might be worth exploring more, since it seems to be yielding promising results, but probably needs better tuning of the "sensitivity" parameter.

Although not explored in this experiment, one could try using different adaptive update strategies, which might be able to better predict when it is necessary to update the preconditioner. We did not consider such strategies, since these would require more knowledge about the training landscape, which in turn could possibly make tuning harder.

V. Summary

In this work, we implemented various scheduling strategies for updating preconditioners in the SOAP algorithm. Experimental results show that it is, in some cases, more beneficial to update the preconditioner in the early stages of training and that, by gradually reducing the frequency of updates toward the end, it is possible to lower computational costs while maintaining model performance. In general, scheduling the update frequency might prove more beneficial than it seems, and future works should explore additional, less trivial strategies for doing so.

REFERENCES

[1] N. Vyas, V. Gupta, T. Koren, and Y. Singer, "Soap: Improving and stabilizing shampoo using adam," *arXiv preprint arXiv:2409.11321*, 2024. [Online]. Available: https://arxiv.org/abs/2409.11321

[2] V. Gupta, T. Koren, and Y. Singer, "Shampoo: Preconditioned stochastic tensor optimization," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*. PMLR, 2018, pp. 1842–1850. [Online]. Available: https://proceedings.mlr.press/v80/gupta18a.html

[3] D. Morwani, I. Shapira, N. Vyas, E. Malach, S. Kakade, and L. Janson, "A new perspective on shampoo's preconditioner," 2024. [Online]. Available: https://arxiv.org/abs/2406.17748

[4] N. Shazeer and M. Stern, "Adafactor: Adaptive learning rates with sublinear memory cost," 2018. [Online]. Available: https://arxiv.org/abs/1804.04235

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[6] R. Anil, V. Park, R. Pascanu, X. Zhang, and G. Gur-Ari, "Scalable second order optimization for deep learning," *arXiv preprint arXiv:2002.09018*, 2020. [Online]. Available: https://arxiv.org/abs/2002.09018

[7] H.-J. M. Shi, T.-H. Lee, S. Iwasaki, J. Gallego-Posada, Z. Li, K. Rangadurai, D. Mudigere, and M. Rabbat, "A distributed data-parallel pytorch implementation of the distributed shampoo optimizer for training neural networks at-scale," 2023. [Online]. Available: https://arxiv.org/abs/2309.06497

APPENDIX

A. Algorithms

This appendix contains the pseudocode algorithms representing the frequency update routines analyzed.

Algorithm 1 SOAP for a $m \times n$ layer (adapted from [1])

Per-layer matrices: $L \in \mathbb{R}^{m \times m}, R \in \mathbb{R}^{n \times n}, V, M \in \mathbb{R}^{m \times n}$
Hyperparameters: Learning rate η, betas = (β_1, β_2), epsilon ϵ, preconditioning frequency f
Note: Initialization and bias correction are ignored for simplicity.

for *timestep* t **do**

Sample batch B_t
$G \in \mathbb{R}^{m \times n} \leftarrow -\nabla_W \phi_{B_t}(W_t)$
// Compute gradient
$G' \leftarrow Q_L^T G Q_R$ // Rotate gradient
$M \leftarrow \beta_1 M + (1 - \beta_1) G$ // Update 1st moment
$M' \leftarrow Q_L^T M Q_R$ // Rotate 1st moment
$V \leftarrow \beta_2 V + (1 - \beta_2)(G' \odot G')$
// Apply Adam in rotated space
$N' \leftarrow \frac{M'}{\sqrt{V} + \epsilon}$ // Adam preconditioning
$N \leftarrow Q_L N' Q_R$ // Revert to original space
$W \leftarrow W - \eta N$ // Update weights
$L \leftarrow \beta_2 L + (1 - \beta_2) G G^T, R \leftarrow \beta_2 R + (1 - \beta_2) G^T G$
// Update covariance matrices
if *Precondition_Frequency_Routine()* **then**
$Q_L \leftarrow \text{Eigenvectors}(L, Q_L)$
$Q_R \leftarrow \text{Eigenvectors}(R, Q_R)$
// Recompute evecs
end
end

Algorithm 2 Doubling Frequency

if $current\_step - last\_update == f$ **then**
  $f \leftarrow \min\{clip\_value,\ 2f\}$
  return True
else
  return False

Algorithm 3 Halving Frequency

if $current\_step - last\_update == f$ **then**
  $f \leftarrow \max\{1,\ f/2\}$
  return True
else
  return False

Algorithm 4 Fixed-interval Doubling Frequency

if $current\_step \mod interval\_size == f$ **then**
  $f \leftarrow 2f$
if $(current\_step - last\_update) \mod f == 0$ **then**
  return True
else
  return False

Algorithm 5 Loss-dependent Frequency

if $|last\_loss - current\_loss| \geq f$ **then**
  return True
else
  return False