Comparison of second-order optimizers on transformers

Francesca Bettinelli, Luka Radić, Silvia Romanato
CS-439 – Optimization for Machine Learning
Ecole Polytechnique Fédérale de Lausanne

***Abstract*—Second-order optimizers can potentially provide benefits in training deep neural networks (DNNs) compared to first-order optimizers, despite entailing higher computational costs. In this project, we investigate the performance and scalability of two recently proposed second-order optimizers, AdaHessian and AdaSub, on the fine-tuning of a transformer model. In particular, our baseline is the ALBERT model fine-tuned with AdamW on the question-answering dataset SQuAD 2.0. While AdaHessian achieves similar results to AdamW in terms of F1 score and Exact Match – with up to $3\times$ computational overhead –, AdaSub reveals to be too expensive to perform training, hence requiring further improvements to be applied to large models.**

I. Introduction

Deep neural networks (DNNs) have become a fundamental tool for several applications, despite being challenging to optimize because of their high-dimensional and non-convex nature. First-order optimization methods are currently the standard for training DNNs. Still, such optimizers must be carefully tuned for the problem of interest, and they may lead to suboptimal results. Second-order methods have the potential to overcome such issues by leveraging information contained in the curvature of the loss function, typically embedded in some approximation of the Hessian matrix. As a drawback, they usually require additional computational costs; therefore, they may not be scalable to classes of models with a huge number of parameters, such as large language models (LLMs).

In this project, we study two second-order algorithms: AdaHessian [1], which approximates the trace of the Hessian with a randomized technique, and AdaSub [2], which computes the Hessian in a low-dimensional subspace. In particular, we evaluate the performance of AdaHessian and AdaSub on a natural language processing (NLP) application. Specifically, we finetune ALBERT [3], a pre-trained LLM based on the BERT architecture, on SQuAD 2.0 [4], a popular benchmark for question-answering. While AdaHessian already showed promising results on transformers on different tasks [1], AdaSub was only tested on relatively small models. Therefore, in this project, we investigate both the reproducibility of results for AdaHessian and the scalability of AdaSub by comparing them with AdamW [5], a state-of-the-art first-order optimizer for transformers, already implemented in `torch`[1]. The comparison of three optimizers will focus on distinct performance criteria: computational memory usage, execution time, evaluation metrics, and speed of convergence.

The paper is structured as follows. In Section III, we present our algorithms of interest. In Section IV, we describe the numerical experiments performed with ALBERT on SQuAD 2.0. Finally, in Sections V and VI, we summarize our results and discuss potential further developments. Our implementation can be found in this GitHub repository.

[1]https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

II. Related work

First-order optimizers – such as Adam [6] – are nowadays the default choice for training DNNs. However, as they only use gradient information, they may be slow to converge, sensitive to hyperparameters, or suffer from ill-conditioning problems. Second-order optimizers address these issues by exploiting the curvature information of the loss, which is represented by second-order derivatives, i.e., by the Hessian matrix. This class of algorithms can potentially find better descent directions, exhibit faster convergence, and show resiliency to ill-conditioned loss landscapes [7].

The classic example of a second-order optimizer is the so-called Newton method, which preconditions the standard gradient update with the Hessian inverse. Unfortunately, computing and storing the full Hessian is very expensive. Therefore, several techniques have been proposed to compute, instead, approximations of the Hessian. Quasi-Newton methods, for instance, replace the Hessian inverse with a positive definite matrix [8] – different choices of such matrix lead to different methods, like BFGS and L-BFGS [9]. Other approaches approximate the Hessian as a diagonal (or block-diagonal) operator. This can be done with randomized techniques, for instance, by sampling a random vector and computing the Hessian-vector product, at the cost of another backpropagation step [10]. This only doubles the time and storage complexity compared to first-order methods.

III. Models and Methods

A. Optimization Problem

We are interested in the following minimization problem:

$$\min_{\mathbf{w}} f(\mathbf{w}), \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a nonlinear function depending on parameters $\mathbf{w} \in \mathbb{R}^n$. In DNNs, f represents the loss function, while $\mathbf{w}$ collects the trainable parameters. If f is a twice-differentiable function, we can define the gradient (w.r.t. $\mathbf{w}$) as $\mathbf{g} = \nabla f \in \mathbb{R}^n$ and the Hessian as $\mathbf{H} = \nabla^2 f \in \mathbb{R}^{n\times n}$.

B. AdaHessian

AdaHessian [1] computes a diagonal approximation $\mathbf{D} = \mathrm{diag}(\mathbf{H}) \in \mathbb{R}^n$ of the Hessian via a randomized technique known as Hutchinson's method:

$$\mathbf{D} = \mathbb{E}[\mathbf{z} \odot (\mathbf{H}\mathbf{z})], \tag{2}$$

where $\mathbf{z} \in \mathbb{R}^n$ is a Rademacher random vector and $\odot$ denotes the component-wise product.

To mitigate the effects of stochastic noise in (2), $\mathbf{D}$ can be replaced with its spatial average $\mathbf{D}^{(s)}$ computed over a block of size b. Then, we can apply momentum similarly to Adam. In particular, the first moment $\hat{\mathbf{m}}_t$ (at iteration t) is defined exactly as in Adam, while the second moment is computed as

$$\hat{\mathbf{v}}_t = (\bar{\mathbf{D}}_t)^k, \tag{3}$$

where $k > 0$ is an abitrarily chosen parameter, and

$$\bar{\mathbf{D}}_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbf{D}_i^{(s)}\mathbf{D}_i^{(s)}}{1-\beta_2^t}}. \tag{4}$$

For the full pseudocode, see Algorithm 1 in the Appendix.

C. AdaSub

AdaSub [2] computes a second-order approximation of the objective function on a low-dimensional subspace defined adaptively as the linear combination of the current gradient and a small set of past gradients. More precisely, f is restricted to a m-dimensional affine set defined (at iteration t) as $\mathbf{w}_t + \mathrm{Range}(\mathbf{V}_t)$, where $\mathbf{V}_t \in \mathbb{R}^{n\times m}$ is an orthonormal basis of the subspace. Then, the update for AdaSub takes the following form:

$$\mathbf{w}_t \leftarrow \mathbf{w}_t - \eta\mathbf{V}_t(\bar{\mathbf{H}}_t)^{-1}\mathbf{V}_t^T\mathbf{g}_t, \tag{5}$$

where $\bar{\mathbf{H}}_t := \mathbf{V}_t^T\mathbf{H}_t\mathbf{V}_t + \alpha\mathbf{I}_{m\times m}$ is the Hessian estimate in the subspace, $\eta > 0$ is the learning rate, and $\alpha > 0$ is a regularization parameter ensuring that $\bar{\mathbf{H}}_t \succ 0$ (which is needed to obtain a descent direction, as shown in [2]). The update (5) can hence be seen as a regularized Newton step for the restriction of f onto the subspace.

To reduce the cost of the algorithm, when applying AdaSub to DNNs, the Hessian is computed layer-by-layer, which corresponds to finding a block-diagonal approximation. For the full pseudocode, see Algorithm 2 in the Appendix.

D. Benchmarking model and dataset

We decided to focus on the machine question-answering task because it differs from the challenges of the translation, language modeling, and natural language understanding (NLU) tasks on which the AdaHessian algorithm has previously been evaluated in [1]. The Stanford Question Answering Dataset (SQuAD 2.0) [4] was chosen for this task due to its status as a widely recognized benchmark.

The model selected to be finetuned on this task is ALBERT (A Lite BERT) [3], which optimizes the architecture of BERT for better memory usage, speed, and scalability. The reason for this choice is that being a relatively small model with 11.1M parameters, it allows us to perform more experiments, considering the limited computational resources of the project. Precisely, we use the `AlbertForQuestionAnswering` model which consists of the base ALBERT model with a span classification head on top for extractive question-answering tasks.

IV. Experiments

We compare the performance of AdamW, AdaHessian, and AdaSub in terms of the following aspects: computational memory usage, execution time, evaluation metrics, and speed of convergence. The computational memory usage determines how many resources an optimizer needs to store and process data during training. An optimizer requiring less memory can be particularly suitable for training larger models or for devices with limited memory. The execution time measures how quickly an optimizer can complete training cycles, directly impacting the overall training duration. Evaluation metrics are essential for assessing the optimizer's effectiveness in performing the task; in this case, we use standard benchmarks for SQuAD2.0, namely, the F1-score and the Exact Match score. Lastly, the speed of convergence indicates how quickly an optimizer can reach a minimal error state. We will therefore monitor the behavior of the loss, which, for this question-answering task, is the span extraction loss.

All the experiments are performed on a Tesla V100 (32 GB) and a Tesla A100 GPU (40 GB), which show similar characteristics in terms of performance. For the dataset, we use pre-defined splits of 132k training and 12k test samples.

A. Comparison of AdamW and AdaHessian

The first experiment is conducted by comparing AdamW and AdaHessian. For both, we set the number of epochs to 3, use a linear learning rate scheduler with warm-up percent of 0.1, and fix the batch size to 16 due to out-of-memory (OOM) errors on AdaHessian for higher values, which persisted even after implementing gradient accumulation. We use default weight decay and betas, as defined in `torch` implementation of AdamW. The AdamW learning rate is set to $5\ 10^{-5}$ similarly to [3] and [11], both training ALBERT on SQuAD2.0. Our model roughly achieves these baseline results, with a minor metric gap likely due to computational restrictions. In [1], the AdaHessian learning rate is the same as AdamW for the models on which the two optimizers are compared; however, we observe that, in our case, AdaHessian performs badly if we use the same learning rate as AdamW, hence we manually tune this parameter in the range $[5\ 10^{-5}, 10^{-1}]$, obtaining an optimal value of $3\ 10^{-3}$. Furthermore, we experiment with three different values of Hessian power k: 0.25, 0.5, 1. The summary of optimal hyperparameters is shown in Table II.

B. Comparison of AdamW and AdaHessian with a combined training strategy

We tried to investigate whether AdaHessian can find a better optimum when starting from a model partially trained using AdamW, compared to the one fully trained using AdamW. This idea is motivated by the fact that second-order optimizers are known for having superlinear convergence properties when the starting point is already close to the optimum [12]. Therefore, one can use a cheap first-order optimizer at the beginning of training, which will take more radical steps to get closer to the minimum, and then exploit a more expensive but sensitive second-order method only at the later stages. This type of strategy, already applied with success in other domains (see, e.g., [13]), seemed suited for our problem since AdamW shows a fast decrease of the loss at the beginning of the training but then tends to stagnate.

We start from an AdamW checkpoint trained for two epochs and then initialize AdaHessian to run for one more epoch. We use a lower rate compared to the previous experiment,

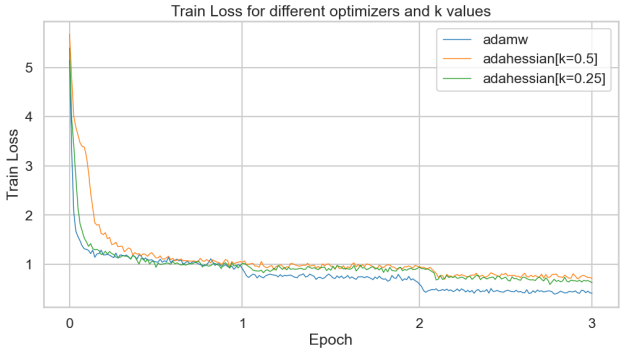

Fig. 1. Loss of the AdaHessian optimizer compared to AdamW. The plot shows the runs done with AdaHessian with hessian power $k = 0.5$ and $k = 0.25$, both with learning rate $3\ 10^{-3}$.

considering the closeness to the local minimum, and test three values with different orders of magnitude: $3\ 10^{-5}$, $3\ 10^{-4}$, and $3\ 10^{-3}$. The Hessian power is set to $k = 0.5$, while the other hyperparameters are the same as the previous experiment.

C. Comparison of the three optimizers on a dataset subsample

The application of the AdaSub optimizer to a LLM presented several issues. Training the model on the full dataset required an unreasonable expected execution time, so we decided to reduce the dataset size to 15k samples. Still, when using the implementation provided by the authors of the original paper [2], an OOM error occurred. We noticed that this issue was due to inefficient usage of the `torch.autograd.grad` function responsible for the computation of Hessian-vector products. Therefore, we restructured the code so that `torch.autograd.grad` was only called once for each group of parameters, following a similar structure as AdaHessian. This way, we managed to reduce the expected execution time of one epoch from 40 to 5 hours on the subsampled dataset. Unfortunately, we encountered more problems (already present in the original code) with the `torch.linalg.eigh` function required for eigenvalue computation, which we did not manage to resolve in the project timeframe, so we ultimately could not perform a full comparison with the AdaSub optimizer. Still, we could test the memory usage required by the algorithm.

V. Results/Discussion

In this section, the results of the different experiments will be presented. As already mentioned, the four dimensions of comparison will be convergence behavior, evaluation metrics, running time, and computational memory. The last two can be discussed here in a general way for both experiments.

The running time for AdamW was around 1 hour per epoch, while for AdaHessian it was around 3 hours. Although AdaHessian should be approximately $2\times$ slower (because of the extra backpropagation step), more overhead comes from the spatial averaging and momentum computation steps. AdaSub, instead, needs to additionally store the subspace and compute its orthogonal basis at each iteration. As a result, AdaHessian and AdaSub consume $1.27\times$ and $1.7\times$ more memory than AdamW, respectively. The required amount of GPU process memory allocation is shown in Figure 4.

A. Comparison on the full dataset

For AdaHessian, we started with the Hessian power $k = 1$. Higher learning rates initially performed better but later led to an increasing loss, likely due to more aggressive steps near the optimum (see Figure 2). By decreasing the Hessian power to $k = 0.5$, we obtained much better results with the same set of learning rates (see Figure 3). We tried to manually find a good balance between these two hyperparameters by comparing the loss between AdamW and AdaHessian with $k \in [0.25, 0.5]$ and learning rate $3\ 10^{-3}$, as shown in Figure 1. From the plot, we can see that AdamW and AdaHessian with $k = 0.25$ have a very similar trend in the first epoch, but AdamW exhibits larger loss drops at epoch switches. Also, while the $k = 0.5$ and $k = 0.25$ behavior seems quite different in the first epoch, this difference is reduced in subsequent epochs. The evaluation metrics are reported in Table I (first block). AdamW outperforms both AdaHessian optimizers, even though the $k = 0.25$ version is still close to AdamW's performance.

TABLE I
Performance Metrics

	F1-score	Exact Match
AdamW	**80.539**	**77.066**
AdaHess[k=0.5]	75.918	72.248
AdaHess[k=0.25]	78.762	75.280
AdaHess[combined]	**80.864**	**77.352**

B. Comparison of AdamW and AdaHessian with a combined training strategy

Regarding the combined training strategy, the loss function is depicted in Figure 5. This plot reveals that, when employing continual training with AdaHessian at a learning rate of $3\ 10^{-4}$, the loss is marginally lower compared to the benchmark performance achieved with AdamW. This model, whose performance is shown in Table I (second block), outperforms the AdamW one, indicating that the integration of two optimization techniques leads to promising improvements. More exhaustive results on the combined performance can be found in Table III.

VI. Conclusions

Throughout this project, we effectively compared the performance of second-order and first-order optimizers on the ALBERT transformer applied to a question-answering task. Our findings revealed that AdaHessian achieved results comparable to AdamW, while a hybrid training approach, incorporating both optimizers, slightly outperformed the benchmark. This possibly underscores better convergence properties of second-order methods near the optimum. Moreover, we suggest that further enhancements to AdaHessian are possible with increased computational resources and more precise hyperparameter tuning. On the other hand, the initial version of AdaSub provided in [2] required excessive training time. After modifying their code, we managed to reduce the training time by $8\times$. However, even after significantly subsampling the dataset, the training remained unfeasible within the project's timeframe. Hence, further adjustments are needed to apply AdaSub to larger models like transformers.

REFERENCES

[1] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. Mahoney, "AdaHessian: An adaptive second-order optimizer for machine learning," in *proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 12, 2021, pp. 10 665–10 673.

[2] J. V. G. Da Mata and M. S. Andersen, "AdaSub: Stochastic Optimization Using Second-Order Information in Low-Dimensional Subspaces," in *2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, Oct. 2023. [Online]. Available: http://dx.doi.org/10.1109/DSAA60987.2023.10302473

[3] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," 2020.

[4] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[5] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in *International Conference on Learning Representations*, 2018.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[7] T.-D. Guo, Y. Liu, and C.-Y. Han, "An overview of stochastic quasi-Newton methods for large-scale machine learning," *Journal of the Operations Research Society of China*, vol. 11, no. 2, pp. 245–275, 2023.

[8] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A survey of optimization methods from a machine learning perspective," *IEEE transactions on cybernetics*, vol. 50, no. 8, pp. 3668–3681, 2019.

[9] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical programming*, vol. 45, no. 1, pp. 503–528, 1989.

[10] C. Bekas, E. Kokiopoulou, and Y. Saad, "An estimator for the diagonal of a matrix," *Applied numerical mathematics*, vol. 57, no. 11-12, pp. 1214–1229, 2007.

[11] S. Li, R. Li, and V. Peng, "Ensemble ALBERT on SQuAD 2.0," 2021.

[12] Q. Jin and A. Mokhtari, "Exploiting local convergence of quasi-newton methods globally: Adaptive sample size approach," *Advances in Neural Information Processing Systems*, vol. 34, pp. 3824–3835, 2021.

[13] P. Rathore, W. Lei, Z. Frangella, L. Lu, and M. Udell, "Challenges in Training PINNs: A Loss Landscape Perspective," 2024.

APPENDIX

TABLE II
OPTIMAL HYPERPARAMETERS FOR FULL DATASET TRAINING. ADAHESS REFERS TO THE MODEL FULLY TRAINED WITH ADAHESSIAN. ADAHESS[COMBINED] DENOTES THE MODEL FIRST TRAINED WITH ADAMW FOR TWO EPOCHS, AND ADAHESSIAN FOR THE LAST EPOCH.

Optimizer	Leaning Rate	Weight Decay	Betas	Batch Size	Warmup Percent	Hessian Power
AdamW	5e-5	0.01	(0.9, 0.999)	16	0.1	/
AdaHess	3e-3	0.01	(0.9, 0.999)	16	0.1	0.25
AdaHess[combined]	3e-4	0.01	(0.9, 0.999)	16	0.1	0.5

Algorithm 1: AdaHessian

Input: Learning rate η, momentum factors β_1, β_2, block size b, Hessian power k

Initialize $\mathbf{w}_0$, $\mathbf{m}_0 \leftarrow \mathbf{0}$, $\mathbf{v}_0 \leftarrow \mathbf{0}$
for $t = 1, 2 \dots$ **do**
  $\mathbf{g}_t \leftarrow \nabla f_t(\mathbf{w}_{t-1})$
  $\mathbf{D}_t \leftarrow \text{Hutchinson}(t)$; // Trace estimation using Hutchinson's method
  $\mathbf{D}_t^{(s)} \leftarrow \text{SpatialAveraging}(b, t)$; // Noise smoothing over a block of size b
  $\bar{\mathbf{D}}_t \leftarrow \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\mathbf{D}_i^{(s)}\mathbf{D}_i^{(s)}}{1-\beta_2^t}}$
  $\hat{\mathbf{m}}_t \leftarrow \frac{(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\mathbf{g}_i}{1-\beta_1^t}$
  $\hat{\mathbf{v}}_t \leftarrow (\bar{\mathbf{D}}_t)^k$
  $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \eta\hat{\mathbf{m}}_t/\hat{\mathbf{v}}_t$
end

Algorithm 2: AdaSub

Input: Learning rate η, subspace dimension m, threshold ρ

for $t = 1, 2 \dots$ **do**
  for $l = 1, \dots, p$ **do**
    if $t = 0$ **then**
      $\mathbf{G}_l \leftarrow \mathbf{g}_{k,l}$
    else
      $\mathbf{G}_l \leftarrow [\mathbf{G}_l, \mathbf{g}_{k,l}]$; // Subspace construction
      if $k > m$ **then**
        Delete 1st column of $\mathbf{G}_l$
      end
    end
    $\mathbf{V}_{t,l} \leftarrow \text{Orhonormalize}(\mathbf{G}_l)$; // Orthonormal basis construction (via QR)
    $\mathbf{Y} \leftarrow \mathbf{H}_{t,l}\mathbf{V}_{t,l}$; // Hessian-vector products
    $\mathbf{\Lambda}, \mathbf{U} \leftarrow \text{SpectralDecomposition}(\mathbf{Y}^T\mathbf{V}_{t,l})$; // $V^THV = U^T\Lambda U$
    $\alpha \leftarrow \max(0, \rho - \min(\Lambda))$
    $\mathbf{w}_{t,l} \leftarrow \mathbf{w}_{t,l} - \eta\mathbf{V}_{t,l}(\mathbf{V}_{t,l}^T\mathbf{H}_{t,l}\mathbf{V}_{t,l} + \alpha\mathbf{I})^{-1}\mathbf{V}_{t,l}^T\mathbf{g}_{t,l}$
  end
end

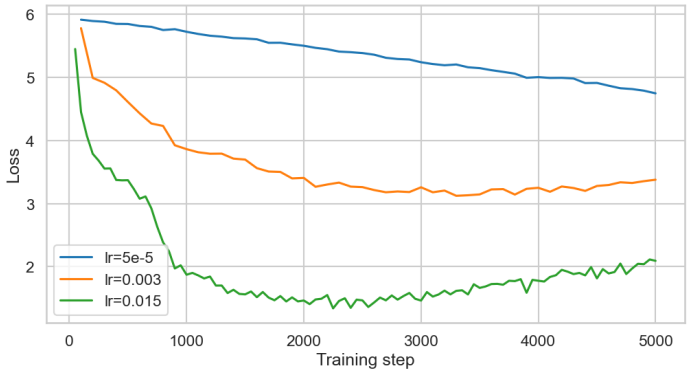

Fig. 2. The loss trend for the first 5k steps using AdaHessian for $k = 1$ and three different learning rates. The effect of the loss first decreasing and then increasing is the most visible for $lr = 0.015$ and present but less exhibited for $lr = 0.003$. This also happens for $lr = 5\ 10^{-5}$, but later during training, after the figure is cut. This effect is likely due to the high hessian power enforcing higher steps as the loss gets closer to the minimum, and more exhibited for higher learning rates amplifying this behaviour.

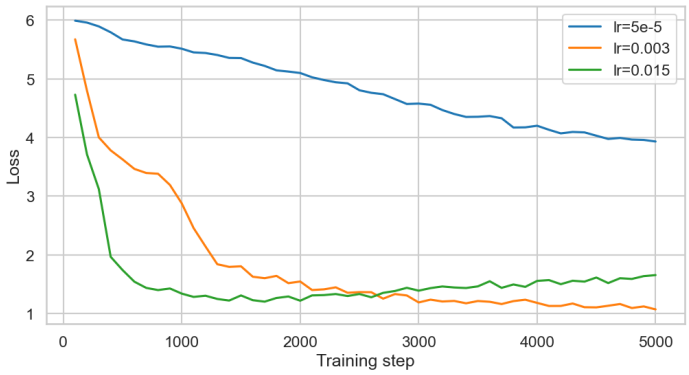

Fig. 3. The loss trend for the first 5k steps using AdaHessian for $k = 0.5$ and three different learning rates. The effect of the loss first decreasing and then increasing is now eliminated for $lr = 5\ 10^{-5}$ and $lr = 0.003$, likely because this value of hessian power is not aggressive anymore. However, the effect is still exhibited for $lr = 0.015$, implicating that the learning rate remains too high.

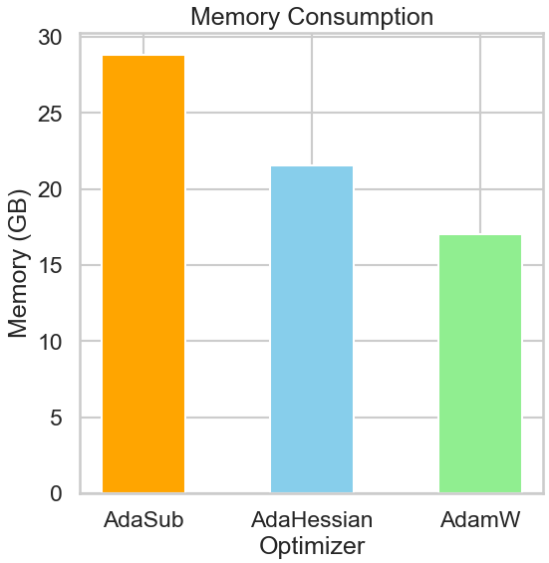

Fig. 4. GPU Memory allocation of AdaHessian, AdaSub and AdamW in GB. It is possible to see how AdamW required the least memory allocation, while AdaHessian required 40% more and Adasub required almost double.

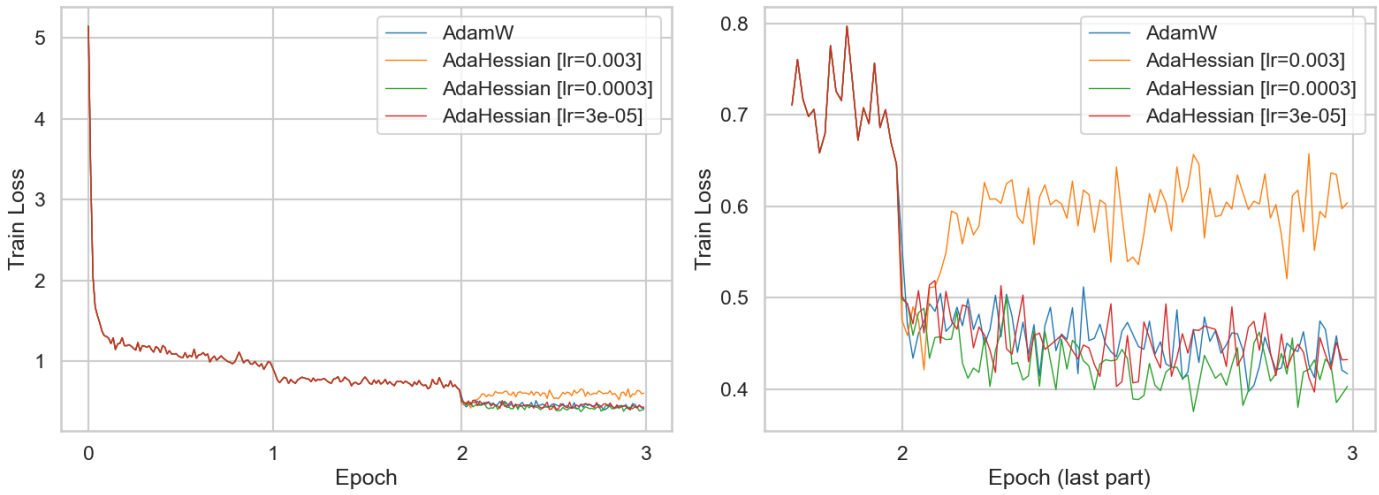

Fig. 5. The combined training strategy of AdamW and AdaHessian is illustrated in the accompanying figures. The left figure displays the training loss for AdamW during the first two epochs, followed by the continuation of training using three variants of AdaHessian. The right figure provides a detailed view of the third epoch. These figures demonstrate that lower learning rates enhance AdaHessian's performance, enabling it to achieve a lower loss compared to AdamW.

TABLE III
PERFORMANCE METRICS OF THE COMBINED TRAINING STRATEGY

	F1-score	**Exact Match**
AdamW	80.539	77.066
AdaHessian[lr=0.003]	78.905	75.785
AdaHessian[lr=0.0003]	**80.864**	**77.352**
AdaHessian[lr=3e-05]	80.397	76.88