Analysis of the efficiency of hybrid methods to optimize the training of deep neural networks

Nussbaumer Arthur, Poulain--Auzéau Louis & Seiler Emilien
Optimization for Machine Learning - CS-439, EPFL, Switzerland

***Abstract*—In this paper, we study the efficiency of using second-order optimizers and of combining different optimization algorithms on the CIFAR10 classification task, using a ResNet18 network. We first have a look at ATMO, a mix between Adam and SGD ([3]), and compare it to classical first-order optimizers. We then try using AdaHessian, a second order optimizer. Finally, following the model of ATMO, we associate AdaHessian with SGD to mix second and first order optimization. For all the models, we also compare the training times.**

I. Introduction

Theoretically, second order algorithms present superior convergence properties as compared to first order methods such as SGD (Stochastic Gradient Descent) and Adam. However, even for a simple optimization problem, those algorithms are very time and memory consuming. Then, for any machine learning task, given the huge number of parameters to update, using a second order algorithm would be unfeasible, which is why first-order optimizers are popular. However, there are often a lot of rules that need to be followed very precisely to converge to a point with good generalization properties. Indeed, the choice of the first-order optimizer has become an rule which can significantly affect the performance. Besides, even after the choice of the optimizer is made, one need to make sure the its hyper-parameters, such as learning rate or weight decay, are fine tuned.
In this paper, we first analyze a new optimizer and its derivates, that combines Adam and SGD (ATMO, [3]) in order to take advantage of both their desirable properties and we compare to classical SGD and Adam training and to a state of the art second-order method (AdaHessian, [6]). Then, to take advantage of the good convergence properties of AdaHessian and the good generalization properties of SGD, we introduce a mix of those two optimizers, based on the ATMO idea. To get the best results, we will also study the efficiency of using schedulers to reduce the learning rate along the training and hopefully get to better generalizing network. In particular, two of them are studied, CosineAnnealingLR and MultiStepLR.

Our work is based on the library PyTorch ([5]) and the jobs are run on Google Colab using P100 and T4 NVIDIA gpu.

In Section II, we present the model, the schedulers and the dataset that we will use. Section III presents the definitions of our optimizers and their algorithms. We then present our results in Section IV. Finally, the results are discussed in Section V.

II. Experimental setup

A. Model

For our experiment on the different optimizers, we train the ResNet18 network. This is a 18-layers deep convolutional neural network. It combines convolution layers, batch normalization layers and max pooling layers and uses ReLu as activation function, defined as:

$$\text{ReLU}(x) = \max(0, x).$$

B. Dataset

For the data, we used the famous dataset CIFAR10. It contains 60'000 color images of 32x32 pixels that belongs to 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). Classes are mutually exclusive and contain each 6000 different images. The dataset is divided into 50000 training images and 10000 test images.

C. Schedulers

In this work, we used two of PyTorch's schedulers to analyze the gain of performance, compared to keeping the learning rate unchanged.

1) CosineAnnealingLR

This scheduler allows to change the learning rate each epoch, using the rule described in Equation 1. For this scheduler, we chose to change the learning rate until the last epoch, which means it is always changing through the training. The update is defined as

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{t}{T_{max}}\pi\right)\right), \quad (1)$$

with $\eta_{min} = 0$, $\eta_{max} = \eta_{initial}$.

2) MultiStepLR

This optimizer allows to change the learning rate by multiplying it by a fixed value after a given number of epochs. We always use a multiplicative factor of 0.1 for the updates.

III. The different optimizers

In this section, we describe the different optimizers we used in our experiments.

A. First-order methods

The first-order methods are a class of algorithms using the gradient of the loss function to find a minimum. For our experiments we mostly used the SGD and the Adam algorithms. We then used combinations and small modifications of those two optimizers (ATMO, PAdamSGD). The ATMO ([3]) algorithm is presented in Algorithm 2. This optimizer combines the descent directions and step sizes of Adam and SGD, to make more significant updates at the beginning and faster ones at the end of the training.
We also studied PADAM ([1]), a modified version of Adam (the algorithm is provided in Algorithm 1) and its combination with SGD (PADAM-SGD, [2]). The algorithms weights given as parameters can be either constant or dynamic, in the sense that they change during the training, based on the number of epochs completed. In the latter case, the optimizers will be refered as 'Dynamic'.

Algorithm 1: Padam

Input: initial point x_1, learning rate η, adaptive parameters β_1, β_2, $p \in (0,1/2]$

set $m_0 = 0$, $v_0 = 0$, $\hat{v_0} = 0$

for t = *1,...,T* **do**

$g_t \leftarrow \Delta f(x_t, \xi_t)$

$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

$\hat{v}_t \leftarrow max(\hat{v}_{t-1}, v_t)$

$x_{t+1} \leftarrow x_t - \eta_t * \frac{m_t}{\hat{v}_t^p}$

end

Algorithm 2: ATMO

Input: weights w_k, learning rate η, weight decay γ, momentum μ, β_1, β_2, dampening d, λ_{Adam}, λ_{SGD} boolean *nesterov*

for *batches* **do**

$d_{\text{Adam}}, \eta_{\text{Adam}} \leftarrow \Delta_{\text{Adam}}(w_k, \eta, \beta_1, \beta_2)$

$d_{\text{SGD}} \leftarrow \Delta_{\text{SGD}}(w_k, \eta, \gamma, \mu, d, \textit{nesterov})$

$d_{\text{merged}} \leftarrow \lambda_{\text{SGD}}\, d_{\text{SGD}} + \lambda_{\text{Adam}}\, d_{\text{Adam}}$

$\eta_{\text{merged}} \leftarrow \lambda_{\text{SGD}}\, \eta + \lambda_{\text{Adam}}\, \eta_{\text{Adam}}$

$w_{k+1} \leftarrow w_k - \eta_{\text{merged}}\, d_{\text{merged}}$

end

B. Second-order methods

Second-order methods, as opposed to first-order methods, rely on the Hessian also for the steps. Theoretically, one needs less steps to reach a minimum, as each step is more significant. However, the computational cost is often very high (matrix inversion) and training a machine learning algorithm with such algorithms takes a lot of time. We study the performance of the recent second-order optimizer AdaHessian ([6]). In fact, instead of using the full Hessian, which is computationally unfeasible, only the diagonal is computed. A spatial averaging is then performed to smooth local variations in the diagonal. Algorithm 3 presents the algorithm. The structure is the same as Adam's algorithm but the gradient is replaced by the diagonal Hessian, seen as a vector in $\mathbb{R}^d$, where d is the number of parameters.

Algorithm 3: AdaHessian

Input: weights w_t, learning rate η, weight decay γ, β_1, β_2, Hessian power k, block size b, stability constant ε

for *batches* **do**

$g_{t+1} \leftarrow$ current step gradient

$D_{t+1} \leftarrow$ current step estimated diagonal Hessian

for $1 \leq j \leq b$ **do**

for $0 \leq i \leq \frac{d}{b} - 1$ **do**

$D^{(s)}_{t+1}[i + bj] \leftarrow \frac{\sum_{k=1}^{b} D_{t+1}[ib+k]}{b}$

end

end

$m_{t+1} \leftarrow \frac{(1-\beta_1)\sum_{i=1}^{t+1} \beta_1^{t-i} g_i}{1-\beta_1^{t+1}}$

$v_{t+1} \leftarrow \left(\sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t+1} \beta_2^{t-i} D_i^{(s)} D_i^{(s)}}{1-\beta_2^{t+1}}} \right)^k$

$w_{t+1} \leftarrow w_t \left(1 - \eta\,\gamma\right) - \eta \left(\frac{m_{t+1}}{v_{t+1} + \varepsilon} \right)$

end

Finally, we introduce in Algorithm 4 a combination of AdaHessian and SGD. The idea is to use the performance of AdaHessian updates at the beginning and the generalization properties of SGD at the end. For this optimizer, we used both scheduler previously described and a dynamic update of the weights for each algorithm as mentioned in Section III.

Algorithm 4: AdaSGD

Input: weights w_k, learning rate η, weight decay γ, momentum μ, β_1, β_2, dampening d, Hessian power k, λ_{AdaHess}, λ_{SGD}, boolean *nesterov*, stability constant ε

for *batches* **do**

$d_{\text{AdaHess}}, \eta_{\text{AdaHess}} \leftarrow \Delta_{\text{Adam}}(w_k, \eta, \gamma, \beta_1, \beta_2, k, \varepsilon)$

$d_{\text{SGD}} \leftarrow \Delta_{\text{SGD}}(w_k, \eta, \gamma, \mu, d, \textit{nesterov})$

$d_{\text{merged}} \leftarrow \lambda_{\text{SGD}}\, d_{\text{SGD}} + \lambda_{\text{AdaHess}}\, d_{\text{AdaHess}}$

$\eta_{\text{merged}} \leftarrow \lambda_{\text{SGD}}\, \eta + \lambda_{\text{AdaHess}}\, \eta_{\text{AdaHess}}$

$w_{k+1} \leftarrow w_k - \eta_{\text{merged}}\, d_{\text{merged}}$

end

IV. Results

This section aims at presenting our best results for the different optimizers. In practice, we try to optimize these results by changing these parameters based mainly on [2] and [3]. The different parameters we use for our best models for each of these methods are listed in tables I and II. Moreover, we plot the different train and test loss and accuracies for each of these models and the result is shown in Figure 1.

We also studied the different training times per epochs depending on the optimizer and compared those of first-order methods and the ones of second-order optimizers (Figure 2). Note that we use different number of epochs for second-order methods specially because it was too long to run on more epochs and the accuracy was already stable as explained in the discussion.

	lr	Momentum	Weight decay [4]	betas	Batch size
SGD	0.01	0.95	0.0005	-	256
Adam	0.001	-	0.0005	(0.9, 0.999)	256
ATMO	0.01	0.95	0.0005	(0.9, 0.999)	256
Dyn. ATMO	0.001	0.95	0.0005	(0.9, 0.999)	256
PADAM-SGD	0.01	0.9	0.025		128
Dyn. PADAM-SGD	0.01	0.9	0.025		128
AdaHessian	0.15	-	0.0005	(0.9, 0.999)	256
AdaSGD	0.1	0.95	0.0005	(0.9, 0.999)	256

TABLE I: Parameters of our different optimizers.

	Scheduler	Epochs	Optimizer weight
SGD	-	350	-
Adam	-	350	-
ATMO	-	350	$\lambda_{\text{Adam}} = \lambda_{\text{SGD}} = 0.5$
Dyn. ATMO	CosineAnnealingLR	350	Dynamic
PADAM-SGD	MultiStepLR(100, 150)	200	
Dyn. PADAM-SGD	MultiStepLR(100, 150)	200	
AdaHessian	MultiStepLR (80, 120)	160	-
AdaSGD	MultiStepLR(100, 140, 180)	200	Dynamic

Note. Optimizer weight refers to the weights used when combining two optimizers. The numbers inside MultistepLR refer to the epochs when the learning rate was modified.

TABLE II: Number of epochs of our training and schedulers.

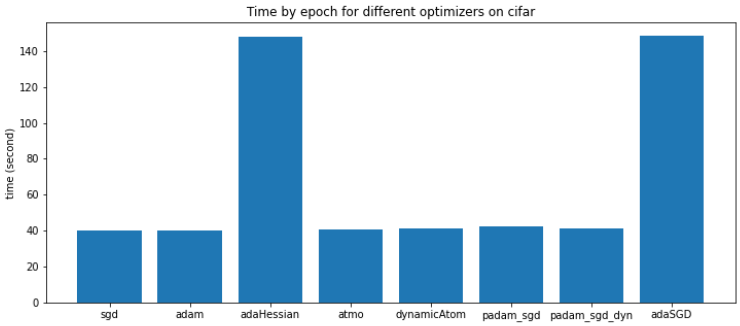

Fig. 2: Training time per epochs for the different optimizers

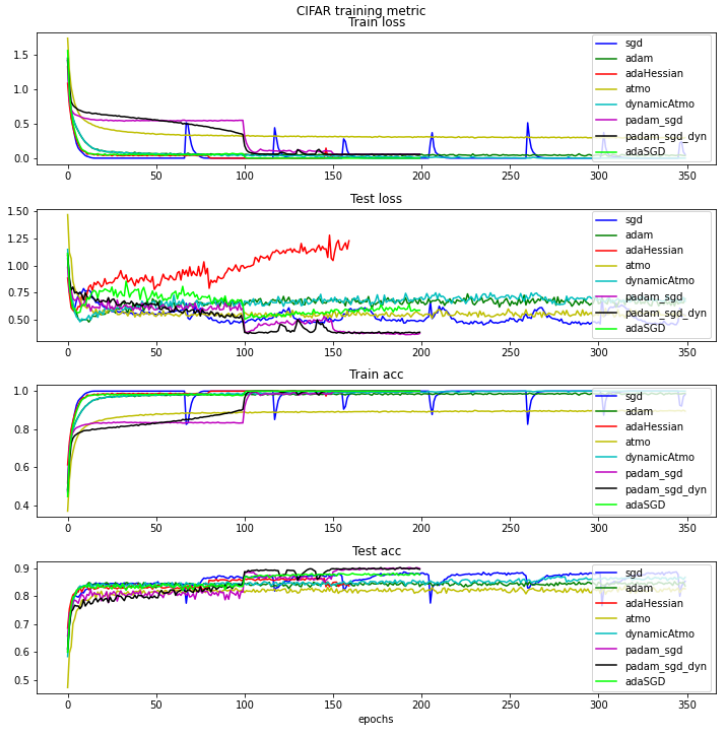

Fig. 1: Evolution of loss and accuracy during the training and the test for our different optimizers.

	test accuracy
SGD	0.8539
Adam	0.8370
ATMO	0.8270
Dyn. ATMO	0.8680
PADAM-SGD	**0.8997**
Dyn. PADAM-SGD	**0.8973**
AdaHessian	0.8372
AdaSGD	0.8771

TABLE III: Best test accuracies for our different optimizers.

V. Discussion

Looking at Figure 1 and at the Table III showing respectively the graph of the different train and test losses and accuracies and the value of these accuracies, one can see clearly that the best optimizers are PADAM-SGD and dynamic PADAM-SGD. In fact, their test losses are still decreasing after 200 epochs, which means that our algorithm still learn and improve itself, and the test accuracies are the highest of all our optimizers. What is interesting to remark is that in the beggining, these last increase slowly compared to the ones of the other methods, but then, it jumps quickly above all the others at 100 epochs. This is clearly an effect of the scheduler which changes the learning rate at 100 and 150 epochs. This can also be seen with adaSGD that leads to good results after 100 epochs. Thus, one can say that changing the learning rate is very good for generalization. However, it seems we could not hope to increase the accuracy by training on more epochs because it does not move a lot from 150 epochs for these three methods.

If we look now at the other first-order methods, we see that momentum SGD and dynamical ATMO perform pretty good on the test data and have a higher accuracy than Adam and ATMO. However, it seems that the SGD optimizer isn't consistent in the long term because its test loss and accuracy moves a lot even after 350 epochs. For dynamical ATMO, the test accuracy seems to increase and it could be maybe useful to increase the number of epochs to have better results.

For second-order methods, AdaHessian is clearly overfitting (as the increasing graph of its test loss shows), but the results are not so bad for the accuracy. However, this would probably leads to a decrease of this measure if we continue to train our model with this algorithm on more epochs. Our hybrid method in comparison has a good stable test accuracy after 200 epochs. However, looking at Figure 2 showing the train time by epochs for the different optimizers, we see that it is much more longer for second-order methods and the results are not really better than other first-order methods in practice. In fact, it takes almost 4 times more time to train an epoch with these methods than with the ones involving only the gradient.

VI. Conclusion

In this report, we study the performance of hybrid methods and compare them to standard first-order and second-order optimizers by training a deep convolutional neural network on CIFAR10 data. What we saw is that the hybrid ATMO method leads to good result only if we use the dynamical algorithm. AdaSGD in comparison leads to good results, but it takes much more time to run and there is thus a tradeoff to make between the training time and the efficiency of our algorithm.

In a future work, it would be interesting to have a more deeper look on PADAM-SGD methods which seem to lead to pretty good results and also to study the hybrid methods on a bigger data set ($e.g.$ CIFAR100) or for a different machine learning task to see if they could be useful in such situations.

Acknowledgements

The authors thank Martin Jäggi and Nicolas Flamarion, who teach us the course in which this project takes part.

References

[1] Jinghui Chen et al. “Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks”. In: (2018). DOI: 10.48550/ARXIV.1806.06763.

[2] Nicola Landro, Ignazio Gallo, and Riccardo La Grassa. *Padam*. 2021. URL: https://github.com/nicolalandro/Padam.git.

[3] Nicola Landro, Ignazio Gallo, and Riccardo La Grassa. “Combining Optimization Methods Using an Adaptive Meta Optimizer”. In: *Algorithms* 14.6 (2021). DOI: 10.3390/a14060186.

[4] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: (2017). DOI: 10.48550/ARXIV.1711.05101.

[5] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] Zhewei Yao et al. “ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning”. In: (2020). DOI: 10.48550/ARXIV.2006.00719.

Appendix

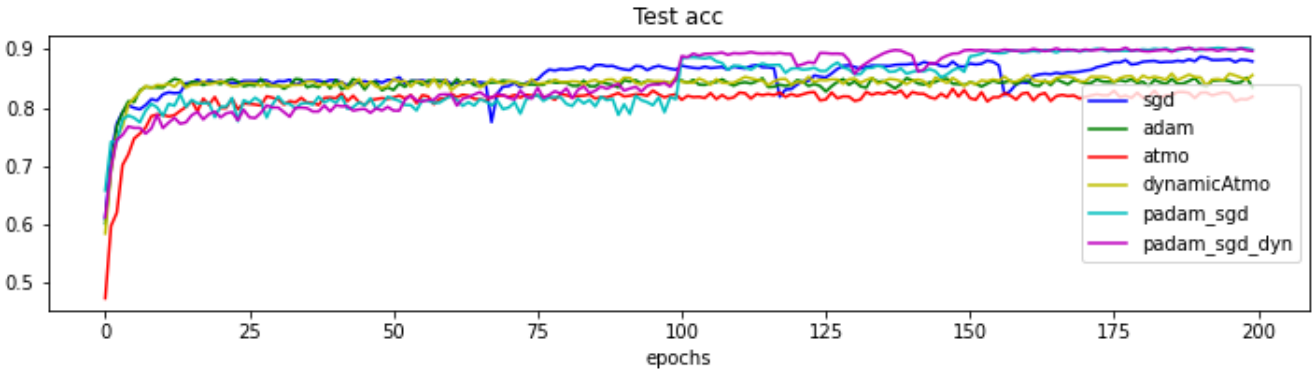

Fig. 3: Test accuracies of our different first-order methods for the firts 200 epochs..

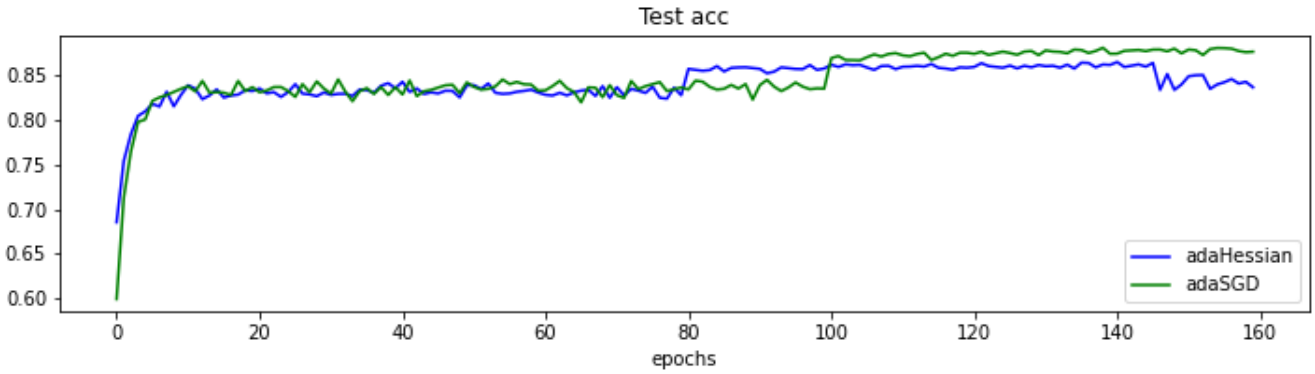

Fig. 4: Test accuracies of our different second-order methods.

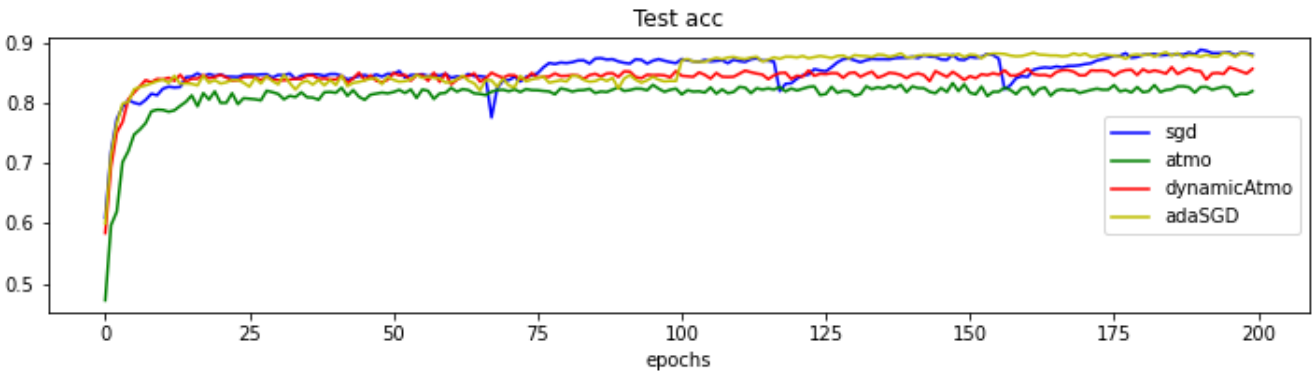

Fig. 5: Test accuracies of the hybrid methods for the first 200 epochs.