ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning

Zhewei Yao[1]*, Amir Gholami[1]*, Sheng Shen[1], Mustafa Mustafa[2], Kurt Keutzer[1], Michael Mahoney[1]

[1]University of California, Berkeley
[2]Lawrence Berkeley National Laboratory
{zheweiy, amirgh, sheng.s, keutzer, mahoneymw}@berkeley.edu, mmustafa@lbl.gov

Abstract

Incorporating second-order curvature information into machine learning optimization algorithms can be subtle, and doing so naïvely can lead to high per-iteration costs associated with forming the Hessian and performing the associated linear system solve. To address this, we introduce ADAHESSIAN, a new stochastic optimization algorithm. ADAHESSIAN directly incorporates approximate curvature information from the loss function, and it includes several novel performance-improving features, including: (i) a fast Hutchinson based method to approximate the curvature matrix with low computational overhead; (ii) a spatial averaging to reduce the variance of the second derivative; and (iii) a root-mean-square exponential moving average to smooth out variations of the second-derivative across different iterations. We perform extensive tests on NLP, CV, and recommendation system tasks, and ADAHESSIAN achieves state-of-the-art results. In particular, we find that ADAHESSIAN: (i) outperforms AdamW for transformers by 0.13/0.33 BLEU score on IWSLT14/WMT14, 2.7/1.0 PPL on PTB/Wikitext-103; (ii) outperforms AdamW for SqueezeBert by 0.41 points on GLUE; (iii) achieves 1.45%/5.55% higher accuracy on ResNet32/ResNet18 on Cifar10/ImageNet as compared to Adam; and (iv) achieves 0.032% better score than Adagrad for DLRM on the Criteo Ad Kaggle dataset. The cost per iteration of ADAHESSIAN is comparable to first-order methods, and ADAHESSIAN exhibits improved robustness towards variations in hyperparameter values. The code for ADAHESSIAN is open-sourced and publicly-available (Yao and Gholami 2020).

Introduction

The high-dimensional and non-convex nature of many machine learning tasks has rendered many classical optimization methods inefficient for training and/or evaluating Neural Network (NN) models. As such, first order methods, in particular variants of Stochastic Gradient Descent (SGD), have become the main workhorse for training NN models. However, they are by no means an ideal solution. For example, there are often many ad-hoc rules that need to be followed very precisely in order to converge (hopefully) to a point with good generalization properties. This includes heuristics for the choice of the many hyperparameters, e.g., learning rate, decay schedule, momentum parameters, number of warmup iterations, etc. Even the choice of the first order optimizer has become an ad-hoc rule, which can significantly affect the performance.

For example, SGD (Robbins and Monro 1951) with momentum is typically used in Computer Vision (CV); Adam (Kingma and Ba 2015) is used for training transformer models for Natural Language Processing (NLP); and Adagrad (Duchi, Hazan, and Singer 2011) is used for Recommendation Systems (RecSys). It is far from obvious *a priori* which variant (if any) is appropriate for a new problem domain, and using the wrong SGD variant can lead to significant performance degradation. Importantly, this may *not* be immediately apparent if one only considers certain popular learning tasks, such as ResNet50 (He et al. 2016) training on ImageNet (Deng et al. 2009). The reason is that, for these tasks, years of industrial scale (namely, brute force) hyperparameter tuning, and years of building systems to exploit first derivative (but not second derivative or other) information, has led to what may be termed *ideal-SGD behaviour*. Such a brute force approach is computationally and financially not possible for many large scale learning problems—certainly it is not possible to do routinely. It points to a failure of our theoretical understanding, and it has made it challenging to train and apply NN models reliably and more generally.

Many of these issues arise since first order methods only use gradient information and do not (reliably[1]) consider the curvature properties of the loss landscape. Second order methods, on the other hand, are specifically designed to capture and exploit curvature properties by using both gradient and Hessian information. They have many favorable properties, including resiliency to *ill-conditioned* loss landscapes, invariance to parameter scaling, and robustness to hyperparameter tuning.[2] The main idea underlying second order methods involves *preconditioning* the gradient vector before using it for weight update. For example, the loss could be very flat in one dimension and very sharp in another. As a result, the (larger/smaller) step size taken by the optimizer should be different for these (flatter/sharper) dimensions. Second or-

*Equal Contribution.

[1]Some people view Adam and Adagrad as quasi-Newton methods, i.e., in between first order and second order methods, but the connection is quite indirect, and others justify them differently.

[2]For NNs, ill-conditioning is described in terms of observed properties, such as "exploding" and "vanishing" gradients, rather than problem-specific structural properties like condition numbers.

der methods (including ADAHESSIAN), however, normalize different dimensions via rotation and scaling of the gradient vector before the weight update, and thus can capture this curvature difference.

Nonetheless, in spite of their advantages, second order methods come with challenges. Most obvious is the higher per-iteration cost. In addition, it isn't obvious how to incorporate moving averages or local averaging, as is common with practical first order algorithms, into the Hessian matrix. Finally, much of the systems infrastructure that has been built is biased towards first order methods. Thus, despite their faster convergence rate and improved robustness properties, second order methods are rarely used for training NNs.

In this paper, we introduce ADAHESSIAN, an adaptive second order optimizer that addresses these problems and that exceeds state-of-the-art performance for a wide range of learning problems. Particularly, our contributions are as follows:

- We approximate the Hessian matrix as a diagonal operator. This is achieved by applying Hutchinson's method to approximate the Hessian diagonal, and it reduces the overhead of second order methods from quadratic to linear in d (the number of parameters).
- We incorporate a spatial averaging to reduce the variance of Hessian diagonal elements. This has no additional overhead in the Hutchinson's method, but it favorably affects the performance of the optimizer.
- We exploit this diagonal form to apply a root-mean-square exponential moving average to smooth out "rugged" loss surfaces. This incurs only $\mathcal{O}(d)$ memory complexity.
- We extensively test ADAHESSIAN on a wide range of learning tasks. In all cases, ADAHESSIAN is comparable to or better than state-of-the-art first order methods, and it outperforms other adaptive optimization methods.
  - **NLP:** ADAHESSIAN improves the performance of transformers for machine translation and language modeling tasks, as compared to AdamW. In particular, ADAHESSIAN significantly outperforms AdamW by 0.13/0.33 BLEU on IWSLT14/WMT14, and by 2.7/1.0 PPL on PTB/WikiText-103. Moreover, for SqueezeBERT (Iandola et al. 2016) fine-tuning on GLUE, ADAHESSIAN achieves 0.41 better points than AdamW.
  - **CV:** ADAHESSIAN achieves significantly higher accuracy, as compared to Adam. For instance, for ResNet32/ResNet18 on Cifar10/ImageNet, ADAHESSIAN achieves 93.08%/70.08%, as opposed to 91.63%/64.53% achieved by Adam, respectively. In all cases, ADAHESSIAN achieves similar performance to the heavily-tuned SGD behavior. See Appendix for more details.
  - **RecSys:** ADAHESSIAN improves the performance of DLRM on the Criteo Ad Kaggle dataset by 0.032% as compared to Adagrad, which is commonly used. See Appendix for more details.
- We measure the sensitivity of ADAHESSIAN to different hyperparameters such as learning rate, spatial averaging size, and delayed Hessian computation. Interestingly, our results show that ADAHESSIAN is robust to those hyperparameters.

We emphasize that our empirical results are achieved even though we use the same learning rate schedule, weight decay, warmup schedule, dropout, batch size, and first/second order moment coefficients as the heavily-tuned default first order baseline optimizers. Additional gains could be achieved if one wanted to extensively optimize these hyperparameters.

Problem Formulation And Related Work

We focus on supervised learning tasks. The goal is to solve a non-convex stochastic optimization problem of the form:

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} l_i(x_i, y_i; \theta), \tag{1}$$

where $\theta \in \mathbb{R}^d$ denotes the model parameters, $l_i(x_i, y_i; \theta)$ is the loss function, (x_i, y_i) is the paired input data and corresponding ground truth label, and N is the total number of data points in the training dataset. We denote the first derivative (gradient) of the loss w.r.t. model parameters as $\mathbf{g} = \frac{1}{N_B} \sum_{i=1}^{N_B} \frac{\partial l_i}{\partial \theta}$ and the corresponding second derivative (Hessian) as $\mathbf{H} = \frac{1}{N_B} \sum_{i=1}^{N_B} \frac{\partial^2 l_i}{\partial \theta^2}$, where N_B is the size of one mini-batch. In general, first order methods (Robbins and Monro 1951; Nesterov 1983; Duchi, Hazan, and Singer 2011; Tieleman and Hinton 2012; Zeiler 2012; Kingma and Ba 2015; Loshchilov and Hutter 2019; Liu et al. 2020) can be represented using the following general update formula:

$$\theta_{t+1} = \theta_t - \eta m_t / v_t, \tag{2}$$

where η is the learning rate, and m_t and v_t are the so-called first and second moment terms, respectively. A summary of the values of m_t and v_t for various methods is given in Appendix.

Using SGD and related first order methods to solve Eq. 1 is often very challenging, due to their strong sensitivity to learning rate, decay schedule, momentum parameters, etc. To address this, several adaptive methods (Duchi, Hazan, and Singer 2011; Kingma and Ba 2015) have been proposed to take into account knowledge of the geometry of the data by scaling gradient coordinates, using the past gradient information. This can be viewed in one of two equivalent ways: either as automatically adjusting the learning rate in Eq. 2; or as an adaptive *preconditioner* of the gradient.

The above first-order adaptive methods have been proposed to improve SGD. However, for some practical machine learning problems they actually perform worse than vanilla SGD with momentum. This is in fact one of the main baffling practical issues in machine learning, and one for which theory has little to say. For example, SGD is currently the best performing optimizer for some CV tasks. That is, using other variants such as AdamW (Loshchilov and Hutter 2019) leads to significantly worse generalization performance. However, for NLP tasks, AdamW has the best performance by a large margin as compared to SGD. The point here is that even the choice of the optimizer has effectively become a hyperparameter.

Another class of preconditioning is to use the second order information (Byrd et al. 1995; Bollapragada, Byrd, and Nocedal 2019; Bollapragada et al. 2018; Yao et al.

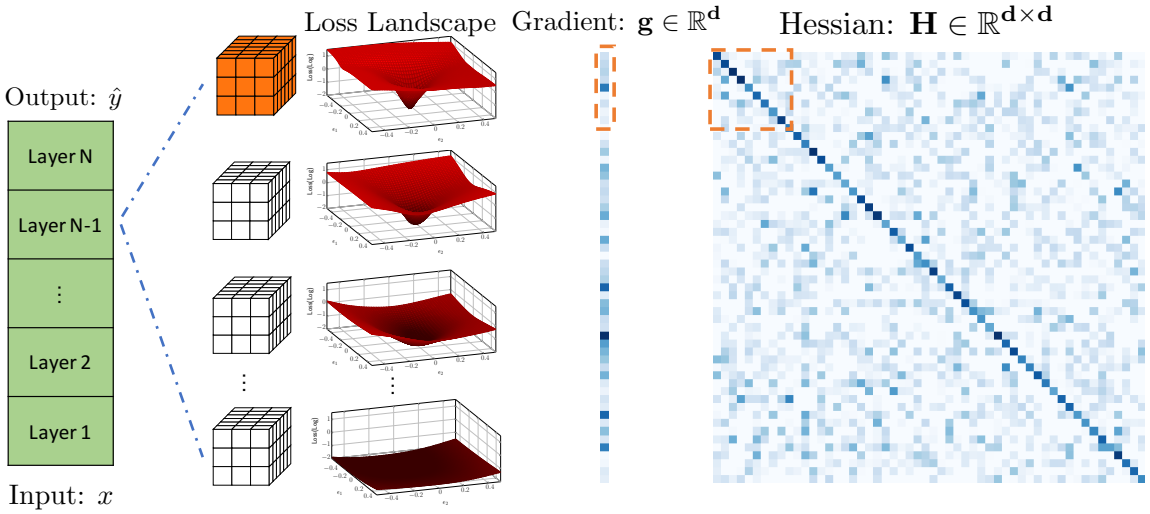

Figure 1: A simple model with N layers (first column); with the convolutional blocks of the N-1 layer shown (second column); and the loss landscape of each block (third column), which can be calculated by perturbing the convolutions's parameters in two different eigen-directions. (See (Yao et al. 2019) for details of how to construct loss landscape.) Note the different loss landscape topologies. First order methods do not explicitly capture this difference. The entries (3D tensors) colored in orange show the components used for calculating the spatial average of Hessian. The part of the gradient (fourth panel) highlighted in the orange box is the corresponding gradient of the orange convolution kernel; and the part of the Hessian diagonal (fifth panel) highlighted in the orange box is used to compute the spatial average.

2018b; Roosta-Khorasani and Mahoney 2016; Xu et al. 2016; Xu, Roosta-Khorasani, and Mahoney 2017a; Xu, Roosta-Khorasan, and Mahoney 2017; Conn, Gould, and Toint 2000; Martens and Grosse 2015; Agarwal, Bullins, and Hazan 2016; Wang et al. 2018b; Agarwal et al. 2016; Carmon et al. 2018; Chen et al. 2019; Gupta, Koren, and Singer 2018; Schaul, Zhang, and LeCun 2013) (we refer the interested reader to (Bottou, Curtis, and Nocedal 2018) for a thorough review of these methods). Second order methods are particularly useful for ill-conditioned problems, and they can have better theoretical convergence rates than their first-order counterparts. However, in practice one of the difficulties in applying second order methods is that noisy estimates of the Hessian could lead to sub-optimal preconditioning (one source of the noise could be the mini-batch used to compute the sub-sampled Hessian). The same problem applies to the mini-batch gradient as well, but it is possible to address it through momentum. However, it is not computationally feasible to (naïvely) incorporate momentum in second order methods as we need to explicitly from the Hessian matrix to average it. Ideally, if there was a way to apply the momentum method to the Hessian, then that would help smooth out the noise to get a better approximation to the non-noisy curvature of the loss landscape.

One way to address the Hessian momentum problem, is to use the Hessian diagonal, instead of the full Hessian operator. Using the diagonal not only allows us to incorporate momentum but also enables the efficient application of inverse Hessian. As we will discuss in the next section, this diagonal approximation along with spatial averaging and momentum of the Hessian proves very helpful for various practical problems.

Methodological Approach

Here, we first provide the formulation for the full Newton method. Then we describe the three components of ADAHESSIAN, namely Hessian diagonal approximation, spatial averaging, and Hessian momentum. Finally, we discuss the overall formulation of ADAHESSIAN.

A General Hessian Based Descent Direction

For the loss function $f(w) : \mathbb{R}^d \to \mathbb{R}$, let us denote the corresponding gradient and Hessian of $f(w_t)$ at iteration t as $\mathbf{g}_t$, and $\mathbf{H}_t$, respectively.[3] A general descent direction can then be written as follows for a positive-definite Hessian:

$$\Delta w_t = \mathbf{H}_t^{-k}\mathbf{g}_t, \quad \text{where } \mathbf{H}_t^{-k} = U_t^T \Lambda_t^{-k} U_t. \tag{3}$$

Here, we refer to $0 \leq k \leq 1$ as *Hessian power*, and $U_t^T \Lambda_t U_t$ is the eigen-decomposition of $\mathbf{H}_t$. Note that for $k = 0$, we recover the gradient descent method; and for $k = 1$, we recover the Newton method. While in our empirical tests we consider non-convex machine learning problems, but we provide a standard convergence behaviour of Eq. 3 in Appendix for a simple strongly convex and strictly smooth function $f(w)$ (we emphasize that the proof is very standard and we are only including it for completeness).

The basic idea of Hessian based methods is to *precondition* the gradient with the $\mathbf{H}^{-k}$ and use $\mathbf{H}^{-k}\mathbf{g}$ for the update

[3] Without confusion, we use the same gradient and Hessian notations for $f(w)$ and $\mathcal{L}(\theta)$. Furthermore, when there is no confusion we will drop subscript t.

direction, instead of using the *bare* gradient $\mathbf{g}$ vector. The preconditioner automatically rotates and rescales the gradient vector. This is important since the loss landscape curvature is generally different across different directions/layers and since these directions need not correspond to the canonical axes. This is illustrated in Fig. 1, where we show a 2D schematic plot of the loss landscape for different convolution channels (Yao et al. 2019). Each channel can have a different loss landscape topology. For example, the last channel has a much flatter loss landscape, as compared to other layers. As a result, it is preferable to take a larger step size for the last channel than for the first channel, which has a very "sharp" loss landscape. Problems that exhibit this behaviour are *ill-conditioned*. The role of the Hessian is to automatically normalize this ill-conditionedness by stretching and contracting different directions to accommodate for the curvature differences (full Newton method also rotates the gradient vector along with adjusting the step size).

However, there are two major problems with this approach. The first problem is that a naïve use of the Hessian preconditioner comes at the prohibitively high cost of applying Hessian inverse to the gradient vector at every iteration ($\mathbf{H}^{-k}\mathbf{g}$ term). The second and more challenging problem is that *local Hessian* (curvature) information has noise when a mini-batch is used to compute it. The same problem exists for the gradient as well, but that can be alleviated by using gradient momentum instead of local gradient information. However, as mentioned before it is computationally infeasible to (naïvely) compute a Hessian momentum. The reason is that we cannot form the Hessian matrix and average it throughout different iterations, as such an approach has quadratic memory complexity in the number of parameters along with a prohibitive computational cost. However, as we discuss next, both problems can be resolved by using Hessian diagonal instead of the full Hessian.

Hessian Diagonal Approximation

To address the issue that applying the inverse Hessian to the gradient vector at every iteration is computationally infeasible, one could use an inexact Newton method, where an approximate Hessian operator is used instead of the full Hessian (Dembo, Eisenstat, and Steihaug 1982; Xu, Roosta-Khorasani, and Mahoney 2017b,a; Yao et al. 2018b; Bollapragada, Byrd, and Nocedal 2019). The most simple and computationally efficient approach is to approximate the Hessian as a diagonal operator in Eq. 3:

$$\Delta w = diag(\mathbf{H})^{-k}\mathbf{g}, \tag{4}$$

where $diag(\mathbf{H})$ is the Hessian diagonal, which we denote as $\boldsymbol{D}$.[4] We show that using Eq. 4 has the same convergence rate as using Eq. 3 for simple strongly convex and strictly smooth function $f(w)$ (see Appendix). Note that we only include the proof for completeness, and our algorithm ADAHESSIAN can be applied for general machine learning problems.

The Hessian diagonal $\boldsymbol{D}$ can be efficiently computed using the Hutchinson's method. The two techniques we use for this approximation are: (i) a Hessian-free method (Yao et al. 2018a); and (ii) a randomized numerical linear algebra (RandNLA) method (Bekas, Kokiopoulou, and Saad 2007, Figure 1). In particular, the Hessian-free method is an oracle to compute the multiplication between the Hessian matrix $\mathbf{H}$ with a random vector z, i.e.,

$$\frac{\partial \mathbf{g}^T z}{\partial \theta} = \frac{\partial \mathbf{g}^T}{\partial \theta} z + \mathbf{g}^T \frac{\partial z}{\partial \theta} = \frac{\partial \mathbf{g}^T}{\partial \theta} z = \mathbf{H}z. \tag{5}$$

Here, the first equality is the chain rule, and the second equality is since z is independent of θ. Eq. 5 effectively allows us to compute the Hessian times a vector z, without having to explicitly forming the Hessian, by backpropotating the $\mathbf{g}^T z$ term, which has the same cost as ordinary gradient backpropogation (Yao et al. 2018a). Then, with the Hessian matvec oracle, one can compute the Hessian diagonal using Hutchinson's method:

$$\boldsymbol{D} = diag(\mathbf{H}) = \mathbb{E}[z \odot (\mathbf{H}z)], \tag{6}$$

where z is a random vector with Rademacher distribution, and $\mathbf{H}z$ is computed by the Hessian matvec oracle given in Eq. 5. This process is illustrated in Appendix. It can be proved that the expectation of $z \odot (\mathbf{H}z)$ is the Hessian diagonal (Bekas, Kokiopoulou, and Saad 2007).

Another important advantage, besides computational efficiency, of using the Hessian diagonal is that we can compute its moving average to resolve the local noisy Hessian as mentioned before. This allows us to smooth out noisy local curvature information, and to obtain estimates that use global Hessian information instead. We incorporate both spatial averaging and momentum (temporal averaging) to smooth out this noisy Hessian estimate as described next.

Spatial Averaging

The Hessian diagonal can vary significantly for each single parameter dimension of the problem. We found it helpful to perform spatial averaging of Hessian diagonal and use the average to smooth out spatial variations. For example, for a convolutional layer, each convolution parameter can have a very different Hessian diagonal. In ADAHESSIAN we compute the average of the Hessian diagonal for each convolution kernel (3×3) as illustrated in Appendix. Mathematically, we perform a simple spatial averaging on the Hessian diagonal as follows:

$$\boldsymbol{D}^{(s)}[ib+j] = \frac{\sum_{k=1}^{b} \boldsymbol{D}[ib+k]}{b}, \text{for } 1 \le j \le b, 0 \le i \le \frac{d}{b} - 1, \tag{7}$$

where $\boldsymbol{D} \in \mathbb{R}^d$ is the Hessian diagonal, $\boldsymbol{D}^{(s)} \in \mathbb{R}^d$ is the spatially averaged Hessian diagonal, $\boldsymbol{D}[i]$ ($\boldsymbol{D}^{(s)}[i]$) refers to the i-th element of $\boldsymbol{D}$ ($\boldsymbol{D}^{(s)}$), b is the spatial average block size, and d is the number of model parameters divisible by b. We show that replacing $\boldsymbol{D}$ in Eq. 4 by $\boldsymbol{D}^{(s)}$ in Eq. 7, the update direction has the same convergence rate as using Eq. 3 for simple strongly convex and strictly smooth function $f(w)$ (see Appendix).

The Appendix provides illustration of spatial averaging for both convolutional and matrix kernels. In general, the block size b is a hyperparameter that can be tuned for different

[4] Note that $\boldsymbol{D}$ can be viewed as a vector, in which case $\boldsymbol{D}^{-k}\mathbf{g}$ is an element-wise product of vectors. Without clarification, $\boldsymbol{D}$ is treated as a vector for the rest of the paper.

Algorithm 1: ADAHESSIAN

Require: Initial Parameter: θ_0
Require: Learning rate: η
Require: Exponential decay rates: β_1, β_2
Require: Block size: b
Require: Hessian Power: k
Set: $m_0 = 0$, $v_0 = 0$
for $t = 1, 2, \ldots$ **do** // Training Iterations
  $\mathbf{g}_t \leftarrow$ current step gradient
  $\boldsymbol{D}_t \leftarrow$ current step estimated diagonal Hessian
  Compute $\boldsymbol{D}_t^{(s)}$ based on Eq. 7
  Update $\bar{\boldsymbol{D}}_t$ based on Eq. 8
  Update $m_t,\ v_t$ based on Eq. 9
  $\theta_t = \theta_{t-1} - \eta m_t / v_t$

tasks. While this is a new hyperparameter that can help the performance, but the performance of ADAHESSIAN is not sensitive to it (we provide sensitivity results in result section).

Next we describe momentum which is another useful method to smooth out Hessian noise over different iterations.

Hessian Momentum

We can easily apply momentum to Hessian diagonal since it is a vector instead of a quadratically large matrix. This enables us to adopt momentum for Hessian diagonal in ADAHESSIAN. More specifically, let $\bar{\boldsymbol{D}}_t$ denote the Hessian diagonal with momentum that is calculated as:

$$\bar{\boldsymbol{D}}_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\boldsymbol{D}_i^{(s)}\boldsymbol{D}_i^{(s)}}{1-\beta_2^t}}, \tag{8}$$

where $\boldsymbol{D}^{(s)}$ is the spatially averaged Hessian diagonal (defined in Eq. 7), and $0 < \beta_2 < 1$ is the second moment hyperparameter. Note that this is exactly the same as the momentum term in Adam (Kingma and Ba 2015) or RMSProp (Tieleman and Hinton 2012) except that we are using the spatial averaging Hessian diagonal instead of the gradient.

AdaHessian

To summarize, instead of only applying momentum for gradient, ADAHESSIAN uses *spatial averaging* and *Hessian momentum* to smooth out local variations in Hessian diagonal. More specifically, the first and second order moments (m_t and v_t) for ADAHESSIAN are computed as follows:

$$\begin{aligned} m_t &= \frac{(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\mathbf{g}_i}{1-\beta_1^t}, \\ v_t &= (\bar{\boldsymbol{D}}_t)^k = \left(\sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\boldsymbol{D}_i^{(s)}\boldsymbol{D}_i^{(s)}}{1-\beta_2^t}}\right)^k, \end{aligned} \tag{9}$$

where $0 < \beta_1,\ \beta_2 < 1$ are the first and second moment hyperparameters that are also used in Adam. Note that Adam uses the same formulation except that the spatial averaging Hessian diagonal $\boldsymbol{D}_i^{(s)}$ is replaced with gradient.

The main overhead of ADAHESSIAN is the Hutchinson's method to approximate Hessian diagonal, $\boldsymbol{D}$. We use one Hutchinson step per iteration to approximate the Hessian diagonal (i.e., one random Rademacher vector z in Eq. 6). The cost of this estimation is one Hessian matvec (to compute Hz), which is equivalent to one gradient backpropagation (Yao et al. 2018a, 2019).

Also note that, it is possible to get a more accurate approximation to Hessian diagonal by using more Hutchinson steps per iteration. However, we found that one step per iteration performs well in practice since the multiple calculations could be performed as Hessian momentum. In fact, as we discuss later, it is possible to skip the Hutchinson calculation for few iterations to further reduce its computational overhead, without significant impact on final accuracy.

Model	**IWSLT14** `small`	**WMT14** `base`
SGD	28.57 ± .15	26.04
AdamW	35.66 ± .11	28.19
ADAHESSIAN	**35.79 ± .06**	**28.52**

Table 1: NMT performance (BLEU) on IWSLT14 De-En and WMT14 En-De testsets (higher is better).

Empirical Performance

We have tested ADAHESSIAN extensively on various learning tasks (NLP, CV, and RecSys). We first describe the experimental setup. Then we present results for Neural Machine Translation (NMT), Language Modeling (LM), and Natural Language Understanding (NLU). We provide further results on CV in and RecSys in Appendix. The reason for such a broad empirical evaluation is that several previous optimization methods were originally tested with very simple models on very few tasks, and when those methods were later tested by the community on more complex models, the results were often worse than popular optimization methods.

Experiment Setup

We start with NLP tasks and compare ADAHESSIAN with SGD and AdamW (Loshchilov and Hutter 2019). For each task we use the optimal hyperparameters reported in the literature for SGD and AdamW to compare with a strong baseline. However, we perform little tuning on ADAHESSIAN since first we do not have access to industrial scale resources to do extensive tuning, and second we want to show the average performance of ADAHESSIAN instead of the absolute best performance achieved with brute force tuning. As such, we directly use the same β_1, β_2, weight decay, batch size, dropout rate and learning rate schedule in ADAHESSIAN as in AdamW for each task (even though tuning those is expected to improve ADAHESSIAN performance). For ADAHESSIAN we only tune the learning rate and the spatial averaging block size b. Please see Appendix for more detailed experimental settings.

Neural Machine Translation

We use BLEU (Papineni et al. 2002) as the evaluation metric for NMT. Following standard practice, we measure tokenized

Model	PTB Three-Layer	Wikitext-103 Six-Layer
SGD	59.9 ± 3.0	78.5
AdamW	54.2 ± 1.6	20.9
ADAHESSIAN	**51.5 ± 1.2**	**19.9**

Table 2: LM performance (PPL) on PTB and Wikitext-103 test datasets (lower is better).

case-sensitive BLEU and case-insensitive BLEU for WMT14 En-De and IWSLT14 De-En, respectively. For a fair comparison, we do not include other external datasets.

The NMT results are shown in Tab. 1. The first interesting observation is that here SGD performs much worse than AdamW (which is opposite to its behaviour for image classification problems where SGD has superior performance; See Appendix). As pointed out in the introduction, even the choice of the optimizer has become another hyperparameter. In particular, note that the BLEU scores of SGD are 7.09 and 2.15 lower than AdamW on IWSLT14 and WMT14, which is quite significant. Similar observations about SGD were also reported in (Zhang et al. 2019).

Despite this, ADAHESSIAN achieves state-of-the-art performance for NMT with transformers. In particular, ADAHESSIAN outperforms AdamW by 0.13 BLEU score on IWSLT14. Furthermore, the accuracy of ADAHESSIAN on WMT14 is 28.52, which is 0.33 higher than that of AdamW. We also plot the training losses of AdamW and ADAHESSIAN on IWSLT14/WMT14 in Appendix. As one can see, ADAHESSIAN consistently achieves lower training loss. These improvements are quite significant for NMT, and importantly these are achieved even though ADAHESSIAN directly uses the same β_1 and β_2, as well as the same number of warmup iterations as in AdamW.

Language Modeling

We report the language modeling results in Tab. 2, using the tensorized transformer proposed in (Ma et al. 2019). Similar to NMT, note that the perplexity (PPL) of SGD is more than 57 points worse than AdamW on Wikitext-103. That is similar to the NMT task, SGD performs worse than AdamW. However, ADAHESSIAN achieves more than 1.8/1.0 better PPL than that of AdamW on PTB/Wikitext-103, respectively.

We also show the detailed training loss curves in Appendix. ADAHESSIAN achieves consistently lower loss values than AdamW throughout the training process on both PTB and Wikitext-103. Similar to NMT, the β_1/β_2 as well as the warmup phase of ADAHESSIAN are kept the same as AdamW.

Natural Language Understanding

We report the NLU results in Tab. 3, using the SqueezeBERT model (Iandola et al. 2016) tested on GLUE datasets (Wang et al. 2018a). As can be seen, ADAHESSIAN has better performance than AdamW on 5 out of 8 tasks. Particularly, on RTE and MPRC, ADAHESSIAN achieves more than 1 point as compared to AdamW. On average, ADAHESSIAN outperforms AdamW by 0.41 points. Note that similar to NMT and LM, except learning rate and block size, ADAHESSIAN directly uses the same hyperparameters as AdamW. Interestingly note that these results are better than those reported in SqueezeBERT (Iandola et al. 2020), even though we only change the optimizer to ADAHESSIAN instead of AdamW.

Sensitivity Analysis

Here, we study the sensitivity of ADAHESSIAN to learning rate and spatial averaging block size, and then discuss the computational overhead of ADAHESSIAN and how reducing the frequency of Hutchinson calculation can reduce the overhead.

Learning Rate and Block Size Effects

Here, we explore the effects of the learning rate and block size b on ADAHESSIAN. We first start with the effect of learning rate, and test the performance of ADAHESSIAN and AdamW with different learning rates. The results are reported in Tab. 4 for IWSLT14 dataset, where we scale the original learning rate with a constant factor, ranging from 0.5 to 20 (the original learning rate is the same as previous section. It can be seen that ADAHESSIAN is more robust to the large learning rates. Even with $10\times$ learning rate scaling, ADAHESSIAN still achieves 32.48 BLEU score, while AdamW diverges even with $6\times$ learning rate scaling. This is a very desirable property of ADAHESSIAN as it results in reasonable performance for such a wide range of learning rates.

We also test the effect of the spatial averaging block size (parameter b in Eq. 7). As a reminder, this parameter is used for spatially averaging the Hessian diagonal as illustrated in Appendix. The sensitivity results are shown in Tab. 5 where we vary the block size from 1 to 128. While the best performance is achieved for the block size of 32, the performance variation for other block sizes is rather small. Moreover, all the results are still no worse than the result with AdamW.

ADAHESSIAN Overhead

Here we discuss and measure the overhead of ADAHESSIAN. In terms of computational complexity, ADAHESSIAN requires twice the flops as compared to SGD. This $2\times$ overhead comes from the cost of computing the Hessian diagonal, when one Hutchinson step is performed per optimization iteration. Each Hutchinson step require computing one Hessian matvec (the $\mathbf{H}z$ term in Eq. 6). This step requires one more gradient backpropagation, hence leading to twice the theoretical complexity. Please refer to (Yao et al. 2019) for more details.

We have also measured the actual runtime of ADAHESSIAN in PyTorch on a single RTX Titan GPU machine, as reported in the second column of Tab. 6. For ResNet20, ADAHESSIAN is $2.42\times$ slower than SGD (and $2.27\times$ slower than Adam). As one can see, ADAHESSIAN is not orders of magnitude slower than first order methods. The gap between the measured and theoretical speed is likely due to the fact that Pytorch (Paszke et al. 2019) (and other existing frameworks) are highly optimized for first order methods. But even then,

	RTE	MPRC	STS-B	SST-2	QNLI	QQP	MNLI-m	MNLI-mm	Avg.
AdamW$^+$ (Iandola et al. 2020)	71.8	89.8	89.4	**92.0**	**90.5**	89.4	82.9	82.3	86.01
AdamW*	79.06	90.69	90.00	91.28	90.30	**89.49**	82.61	81.84	86.91
ADAHESSIAN	**80.14**	**91.94**	**90.59**	91.17	89.97	89.33	**82.78**	**82.62**	**87.32**

Table 3: Comparison of AdamW and ADAHESSIAN for SqueezeBERT on the development set of the GLUE benchmark. The result of AdamW$^+$ is directly from (Iandola et al. 2020) and the result of AdamW* is reproduced by us.

LR Scaling	**0.5**	**1**	**2**	**3**	**4**	**5**	**6**	**10**
AdamW	**35.42 ± .09**	35.66 ± .11	**35.37 ± .07**	**35.18 ± .07**	**34.79 ± .15**	14.41 ± 13.25	0.41 ± .32	Diverge
ADAHESSIAN	35.33 ± .10	**35.79 ± .06**	35.21 ± .14	34.74 ± .10	34.19 ± .06	**33.78 ± .14**	**32.70 ± .10**	**32.48 ± .83**

Table 4: Robustness of AdamW and ADAHESSIAN to the learning rate on IWSLT14. As can be seen, ADAHESSIAN is much more robust to large learning rate variability as compared to AdamW.

Block Size	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
ADAHESSIAN	35.67 ± .10	35.66 ± .07	35.78 ± .07	35.77 ± .08	35.67 ± .08	**35.79 ± .06**	35.72 ± .06	35.67 ± .11

Table 5: Block Size effect of ADAHESSIAN on IWSLT14.

Hutchinson Calculation Frequency	**1**	**2**	**3**	**4**	**5**
Theoretical Cost (×SGD)	2×	1.5×	1.33×	1.25×	1.2×
ResNet20 (Cifar10)	92.13 ± .08	92.40 ± .04	92.06 ± .18	92.17 ± .21	92.16 ± .12
Measured Cost (×SGD)	2.42×	1.71×	1.47×	1.36×	1.28×
Measured Cost (×Adam)	2.27×	1.64×	1.42×	1.32×	1.25×
ResNet32 (Cifar10)	93.08 ± .10	92.91 ± .14	92.95 ± .17	92.93 ± .24	93.00 ± .10
Measured Cost (×SGD)	3.23×	2.12×	1.74×	1.56×	1.45×
Measured Cost (×Adam)	2.91×	1.96×	1.64×	1.48×	1.38×

Table 6: Comparison between theoretical and measured speed of ADAHESSIAN, as compared to SGD and Adam, tested on Cifar10. We also measured the speed up for different Hutchinson calculation frequencies. The real time measurement is performed on one RTX Titan GPU.

if one considers the fact that SGD needs a lot of tuning then this overhead may not be large.

It is also possible to reduce the ADAHESSIAN overhead. One simple idea is to reduce the Hutchinson calculation frequency from 1 Hessian matvec per iteration to multiple iterations. For example, for a frequency of 2, we perform the Hutchinson step at every other optimization iteration. This reduces the theoretical computational cost to $1.5\times$ from $2\times$. One can also further reduce the frequency to 5, for which this cost reduces to $1.2\times$.

We studied how such reduced Hutchinson calculation frequency approach would impact the performance. We report the results for training ResNet20/ResNet32 on the Cifar10 in Tab. 6, when we vary the Hutchinson frequency from 1 to 5. As one can see, there is a small performance variation, but the ADAHESSIAN overhead significantly decreases as compared to SGD and Adam.

Conclusions

In this work, we proposed ADAHESSIAN, an adaptive Hessian based optimizer. ADAHESSIAN incorporates an approximate Hessian diagonal, with spatial averaging and momentum to precondition the gradient vector. This automatically rescales the gradient resulting in better descent directions. One of the key novelties in our approach is the incorporation spatial averaging for Hessian diagonal along with an exponential moving average in time. These enable us to smooth noisy local Hessian information which could be highly misleading. We extensively tested ADAHESSIAN on various datasets and tasks, using state-of-the-art models. These include IWSLT14 and WMT14 for neural machine translation, PTB and Wikitext-103 for language modeling, GLUE for natural language understanding, Cifar10 and ImageNet for image classification, and Criteo Ad Kaggle for recommendation system. ADAHESSIAN consistently achieves comparable or higher generalization performance as compared to the highly tuned default optimizers used for these different tasks.

Acknowledgments

This work was supported by a gracious fund from Amazon Machine Learning Research Award (MLRA). The UC Berkeley team also acknowledges gracious support from Intel corporation, Intel VLAB team, Google Cloud, Google TFTC team, and Nvidia. Amir Gholami was supported through through funding from Samsung SAIT. Michael Mahoney would like to acknowledge the DARPA, IARPA, NSF, and ONR via its BRC on RandNLA for providing partial support. Our conclusions do not necessarily reflect the position or the policy of our sponsors, and no official endorsement should be inferred.

References

Agarwal, N.; Allen-Zhu, Z.; Bullins, B.; Hazan, E.; and Ma, T. 2016. Finding approximate local minima for nonconvex optimization in linear time. *arXiv preprint arXiv:1611.01146* .

Agarwal, N.; Bullins, B.; and Hazan, E. 2016. Second-order stochastic optimization in linear time. *Journal of Machine Learning Research* 1050: 15.

Bekas, C.; Kokiopoulou, E.; and Saad, Y. 2007. An estimator for the diagonal of a matrix. *Applied numerical mathematics* 57(11-12): 1214–1229.

Bollapragada, R.; Byrd, R. H.; and Nocedal, J. 2019. Exact and inexact subsampled Newton methods for optimization. *IMA Journal of Numerical Analysis* 39(2): 545–578.

Bollapragada, R.; Mudigere, D.; Nocedal, J.; Shi, H.-J. M.; and Tang, P. T. P. 2018. A progressive batching L-BFGS method for machine learning. *arXiv preprint arXiv:1802.05374* .

Bottou, L.; Curtis, F. E.; and Nocedal, J. 2018. Optimization methods for large-scale machine learning. *SIAM Review* 60(2): 223–311.

Byrd, R. H.; Lu, P.; Nocedal, J.; and Zhu, C. 1995. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing* 16(5): 1190–1208.

Carmon, Y.; Duchi, J. C.; Hinder, O.; and Sidford, A. 2018. Accelerated methods for nonconvex optimization. *SIAM Journal on Optimization* 28(2): 1751–1772.

Chen, C.; Reiz, S.; Yu, C.; Bungartz, H.-J.; and Biros, G. 2019. Fast Evaluation and Approximation of the Gauss-Newton Hessian Matrix for the Multilayer Perceptron. *arXiv preprint arXiv:1910.12184* .

Conn, A. R.; Gould, N. I.; and Toint, P. L. 2000. *Trust region methods*. Series on Optimization. SIAM.

Dembo, R. S.; Eisenstat, S. C.; and Steihaug, T. 1982. Inexact Newton methods. *SIAM Journal on Numerical analysis* 19(2): 400–408.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12(Jul): 2121–2159.

Gupta, V.; Koren, T.; and Singer, Y. 2018. Shampoo: Preconditioned stochastic tensor optimization. *arXiv preprint arXiv:1802.09568* .

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Iandola, F. N.; Han, S.; Moskewicz, M. W.; Ashraf, K.; Dally, W. J.; and Keutzer, K. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size. *arXiv preprint arXiv:1602.07360* .

Iandola, F. N.; Shaw, A. E.; Krishna, R.; and Keutzer, K. W. 2020. SqueezeBERT: What can computer vision teach NLP about efficient neural networks? *arXiv preprint arXiv:2006.11316* .

Kingma, D. P.; and Ba, J. 2015. Adam: A method for stochastic optimization. *International Conference on Learning Representations* .

Liu, L.; Jiang, H.; He, P.; Chen, W.; Liu, X.; Gao, J.; and Han, J. 2020. On the Variance of the Adaptive Learning Rate and Beyond. In *International Conference on Learning Representations*.

Loshchilov, I.; and Hutter, F. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

Ma, X.; Zhang, P.; Zhang, S.; Duan, N.; Hou, Y.; Zhou, M.; and Song, D. 2019. A tensorized transformer for language modeling. In *Advances in Neural Information Processing Systems*, 2229–2239.

Martens, J.; and Grosse, R. 2015. Optimizing neural networks with Kronecker-factored approximate curvature. In *International conference on machine learning*, 2408–2417.

Nesterov, Y. 1983. A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2). In *Doklady an ussr*, volume 269, 543–547.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, 311–318. Association for Computational Linguistics.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, 8024–8035. Curran Associates, Inc.

Robbins, H.; and Monro, S. 1951. A stochastic approximation method. *The annals of mathematical statistics* 400–407.

Roosta-Khorasani, F.; and Mahoney, M. W. 2016. Sub-sampled Newton methods I: globally convergent algorithms. *arXiv preprint arXiv:1601.04737* .

Schaul, T.; Zhang, S.; and LeCun, Y. 2013. No more pesky learning rates. In *International Conference on Machine Learning*, 343–351.

Tieleman, T.; and Hinton, G. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4(2): 26–31.

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. R. 2018a. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* .

Wang, S.; Roosta, F.; Xu, P.; and Mahoney, M. W. 2018b. GIANT: Globally improved approximate Newton method for distributed optimization. In *Advances in Neural Information Processing Systems*, 2332–2342.

Xu, P.; Roosta-Khorasan, F.; and Mahoney, M. W. 2017. Second-order optimization for non-convex machine learning: An empirical study. *arXiv preprint arXiv:1708.07827* .

Xu, P.; Roosta-Khorasani, F.; and Mahoney, M. W. 2017a. Newton-type methods for non-convex optimization under inexact Hessian information. *arXiv preprint arXiv:1708.07164* .

Xu, P.; Roosta-Khorasani, F.; and Mahoney, M. W. 2017b. Newton-Type Methods for Non-Convex Optimization Under Inexact Hessian Information. *arXiv preprint arXiv:1708.07164* .

Xu, P.; Yang, J.; Roosta-Khorasani, F.; Ré, C.; and Mahoney, M. W. 2016. Sub-sampled Newton methods with non-uniform sampling. In *Advances in Neural Information Processing Systems*, 3000–3008.

Yao, Z.; and Gholami, A. 2020. https://github.com/amirgholami/ADAHESSIAN.git. *Github Online System* .

Yao, Z.; Gholami, A.; Keutzer, K.; and Mahoney, M. W. 2019. PyHessian: Neural Networks Through the Lens of the Hessian. *arXiv preprint arXiv:1912.07145* .

Yao, Z.; Gholami, A.; Lei, Q.; Keutzer, K.; and Mahoney, M. W. 2018a. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries. *Advances in Neural Information Processing Systems* .

Yao, Z.; Xu, P.; Roosta-Khorasani, F.; and Mahoney, M. W. 2018b. Inexact non-convex Newton-type methods. *arXiv preprint arXiv:1802.06925* .

Zeiler, M. D. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* .

Zhang, J.; Karimireddy, S. P.; Veit, A.; Kim, S.; Reddi, S. J.; Kumar, S.; and Sra, S. 2019. Why ADAM Beats SGD for Attention Models. *arXiv preprint arXiv:1912.03194* .