HESSCALE: SCALABLE COMPUTATION OF HESSIAN DIAGONALS

Anonymous authors
Paper under double-blind review

ABSTRACT

Second-order optimization uses curvature information about the objective function, which can help in faster convergence. However, such methods typically require expensive computation of the Hessian matrix, preventing their usage in a scalable way. The absence of efficient ways of computation drove the most widely used methods to focus on first-order approximations that do not capture the curvature information. In this paper, we develop *HesScale*, a scalable approach to approximating the diagonal of the Hessian matrix, to incorporate second-order information in a computationally efficient manner. We show that HesScale has the same computational complexity as backpropagation. Our results on supervised classification show that HesScale achieves high approximation accuracy, allowing for scalable and efficient second-order optimization.[1]

1 INTRODUCTION

First-order optimization offers a cheap and efficient way of performing local progress in optimization problems by using gradient information. However, their performance suffers from instability or slow progress when used in ill-conditioned landscapes. Such a problem is present because first-order methods do not capture curvature information which causes two interrelated issues. First, the updates in first-order have incorrect units (Duchi et al. 2011), which creates a scaling issue. Second, first-order methods lack parameterization invariance (Martens 2020) in contrast to second-order methods such as natural gradient (Amari 1998) or Newton-Raphson methods. Therefore, some first-order normalization methods were developed to address the invariance problem (Ba et al. 2016, Ioffe & Szegedy 2015, Salimans & Kingma 2016). On the other hand, some recent adaptive step-size methods try to alleviate the scaling issue by using gradient information for first-order curvature approximation (Luo et al. 2019, Duchi et al. 2011, Zeiler 2012, Reddi et al. 2018, Kingma & Ba 2015, Tran & Phong 2019, Tieleman et al. 2012). Specifically, such methods use the empirical Fisher diagonals heuristic by maintaining a moving average of the squared gradients to approximate the diagonal of the Fisher information matrix. Despite the huge adoption of such methods due to their scalability, they use inaccurate approximations. Kunstner et al. (2019) showed that the empirical Fisher does not generally capture curvature information and might have undesirable effects. They argued that the empirical Fisher approximates the Fisher or the Hessian matrices only under strong assumptions that are unlikely to be met in practice. Moreover, Wilson et al. (2017) presented a counterexample where the adaptive step-size methods are unable to reduce the error compared to non-adaptive counterparts such as stochastic gradient descent.

Although second-order optimization can speed up the training process by using the geometry of the landscape, its adoption is minimal compared to first-order methods. The exact natural gradient or Newton-Raphson methods require the computation, storage, and inversion of the Fisher information or the Hessian matrices, making them computationally prohibitive in large-scale tasks. Accordingly, many popular second-order methods attempt to approximate less expensively. For example, a type of truncated-Newton method called Hessian-free methods (Martens 2010) exploits the fact that the Hessian-vector product is cheap (Bekas et al. 2007) and uses the iterative conjugate gradient method to perform an update. However, such methods might require many iterations per update or some tricks to achieve stability, adding computational overhead (Martens & Sutskever 2011).

[1]Code will be available.

Some variations try to approximate only the diagonals of the Hessian matrix using stochastic estimation with matrix-free computations (Chapelle & Erhan 2011, Martens et al. 2012, Yao et al. 2021). Other methods impose probabilistic modeling assumptions and estimate a block diagonal Fisher information matrix (Martens & Grosse 2015, Botev et al. 2017). Such methods are invariant to reparametrization but are computationally expensive since they need to perform matrix inversion for each block.

Deterministic diagonal approximations to the Hessian (LeCun et al. 1990, Becker & Lecun 1989) provide some curvature information and are efficient to compute. Specifically, they can be implemented to be as efficient as first-order methods. We view this category of approximation as *scalable second-order methods*. In neural networks, curvature backpropagation (Becker & Lecun 1989, Mizutani & Dreyfus 2008) can be used to backpropagate the curvature vector. Although these methods show a promising direction for scalable second-order optimization, the approximation quality is sometimes poor with objectives such as cross-entropy (Martens et al. 2012). A scalable second-order method with high quality approximation is still needed.

In this paper, we present *HesScale*, a high-quality approximation method for the Hessian diagonals. Our method is also scalable and has little memory requirement with linear computational complexity while maintaining high approximation accuracy.

2 BACKGROUND

In this section, we describe the Hessian matrix for neural networks and some existing methods for estimating it. Generally, Hessian matrices can be computed for any scalar-valued function that are twice differentiable. If $f : \mathbb{R}^n \to \mathbb{R}$ is such a function, then for its argument $\boldsymbol{\psi} \in \mathbb{R}^n$, the Hessian matrix $\boldsymbol{H} \in \mathbb{R}^{n \times n}$ of f with respect to $\boldsymbol{\psi}$ is given by $H_{i,j} = {\partial^2 f(\boldsymbol{\psi})}/{\partial\psi_i \partial\psi_j}$. Here, the ith element of a vector $\boldsymbol{v}$ is denoted by v_i, and the element at the ith row and jth column of a matrix $\boldsymbol{M}$ is denoted by $M_{i,j}$. When the need for computing the Hessian matrix arises for optimization in deep learning, the function f is typically the objective function, and the vector $\boldsymbol{\psi}$ is commonly the weight vector of a neural network. Computing and storing an $n \times n$ matrix, where n is the number of weights in a neural network, is expensive. Therefore, many methods exist for approximating the Hessian matrix or parts of it with less memory footprint, computational requirement, or both. A common technique is to utilize the structure of the function to reduce the computations needed. For example, assuming that connections from a certain layer do not affect other layers in a neural network allows one to approximate a block diagonal Hessian. The computation further simplifies when we have piece-wise linear activation functions (e.g., ReLU), which result in a *Generalized Gauss-Newton* (GGN) (Schraudolph 2002) approximation that is equivalent to the block diagonal Hessian matrix with linear activation functions. The GGN matrix is more favored in second-order optimization since it is positive semi-definite. However, computing a block diagonal matrix is still demanding.

Many approximation methods were developed to reduce the storage and computation requirements of the GGN matrix. For example, under probabilistic modeling assumptions, the *Kronecker-factored Approximate Curvature* (KFAC) method (Martens & Grosse 2015) writes the GGN matrix $\boldsymbol{G}$ as a Kronecker product of two matrices of smaller sizes as: $\boldsymbol{G} = \boldsymbol{A} \otimes \boldsymbol{B}$, where $\boldsymbol{A} = \mathbb{E}[\boldsymbol{h}\boldsymbol{h}^\top]$, $\boldsymbol{B} = \mathbb{E}[\boldsymbol{g}\boldsymbol{g}^\top]$, $\boldsymbol{h}$ is the activation output vector, and $\boldsymbol{g}$ is the gradient of the loss with respect to the activation input vector. The $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices can be estimated by Monte Carlo sampling and an exponential moving average. KFAC is more efficient when used in optimization since it requires inverting only the small matrices using the Kronecker-product property $(\boldsymbol{A} \otimes \boldsymbol{B})^{-1} = \boldsymbol{A}^{-1} \otimes \boldsymbol{B}^{-1}$. However, KFAC is still expensive due to the storage of the block diagonal matrices and computation of Kronecker product, which prevent it from being used as a scalable method.

Computing the Hessian diagonals can provide some curvature information with relatively less computation. However, it has been shown that the exact computation for diagonals of the Hessian typically has quadratic complexity with the unlikely existence of algorithms that can compute the exact diagonals with less than quadratic complexity (Martens et al. 2012). Some stochastic methods provide a way to compute unbiased estimates of the exact Hessian diagonals. For example, the AdaHessian (Yao et al. 2021) algorithm uses the Hutchinson's estimator $\text{diag}(\boldsymbol{H}) = E[\boldsymbol{z} \circ (\boldsymbol{H}\boldsymbol{z})]$, where $\boldsymbol{z}$ is a multivariate random variable with a Rademacher distribution and the expectation can be estimated using Monte Carlo sampling with an exponential moving average. Similarly, the GGN-MC method (Dangel et al. 2020) uses the relationship between the Fisher information matrix and

the Hessian matrix under probabilistic modeling assumptions to have an MC approximation of the diagonal of the GGN matrix. Although these stochastic approximation methods are scalable due to linear or $O(n)$ computational and memory complexity, they suffer from low approximation quality, improving which requires many sampling and factors of additional computations.

3 THE PROPOSED HESSCALE METHOD

In this section, we present our method for approximating the diagonal of the Hessian at each layer in feed-forward networks, where a backpropagation rule is used to utilize the Hessian of previous layers. We present the derivation of the backpropagation rule for fully connected and convolutional neural networks in supervised learning. Similar derivation for fully connected networks with mean squared error is presented before (LeCun et al. 1990, Becker & Lecun 1989). However, we use the exact diagonals of the Hessian matrix at the last layer with some non-linear and non-element-wise output activations such as softmax and show that it can still be computed in linear computational complexity. We show the derivation for Hessian diagonals for fully connected networks in the following and provide the derivation for the convolutional neural networks in Appendix B.

We use the supervised classification setting where there is a collection of data examples. These data examples are generated from some *target function* f^* mapping the input $\boldsymbol{x}$ to the output y, where the k-th input-output pair is $(\boldsymbol{x}_k, y_k)$. In this task, the *learner* is required to predict the output class $y \in \{1, 2, ..., m\}$ given the input vector $\boldsymbol{x} \in \mathbb{R}^d$ by estimating the target function f^*. The performance is measured with the cross-entropy loss, $\mathcal{L}(\boldsymbol{p}, \boldsymbol{q}) = -\sum_{i=1}^{m} p_i \log q_i$, where $\boldsymbol{p} \in \mathbb{R}^m$ is the vector of the target one-hot encoded class and $\boldsymbol{q} \in \mathbb{R}^m$ is the predicted output. The learner is required to reduce the cross-entropy by matching the target class.

Consider a neural network with L layers that outputs the predicted output $\boldsymbol{q}$. The neural network is parametrized by the set of weights $\{\boldsymbol{W}_1, ..., \boldsymbol{W}_L\}$, where $\boldsymbol{W}_l$ is the weight matrix at the l-th layer, and its element at the ith row and the jth column is denoted by $W_{l,i,j}$. During learning, the parameters of the neural network are changed to reduce the loss. At each layer l, we get the activation output $\boldsymbol{h}_l$ by applying the activation function $\boldsymbol{\sigma}$ to the activation input $\boldsymbol{a}_l$: $\boldsymbol{h}_l = \boldsymbol{\sigma}(\boldsymbol{a}_l)$. We simplify notations by defining $\boldsymbol{h}_0 \doteq \boldsymbol{x}$. The activation output $\boldsymbol{h}_l$ is then multiplied by the weight matrix $\boldsymbol{W}_{l+1}$ of layer $l+1$ to produce the next activation input: $a_{l+1,i} = \sum_{j=1}^{|\boldsymbol{h}_l|} W_{l+1,i,j} h_{l,j}$. We assume here that the activation function is element-wise activation for all layers except for the final layer L, where it becomes the softmax function. The backpropagation equations for the described network are given as follows Rumelhart et al. (1986):

$$\frac{\partial \mathcal{L}}{\partial a_{l,i}} = \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} \frac{\partial a_{l+1,k}}{\partial h_{l,i}} \frac{\partial h_{l,i}}{\partial a_{l,i}} = \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} W_{l+1,k,i} \sigma'(a_{l,i}), \tag{1}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l,i,j}} = \frac{\partial \mathcal{L}}{\partial a_{l,i}} \frac{\partial a_{l,i}}{\partial W_{l,i,j}} = \frac{\partial \mathcal{L}}{\partial a_{l,i}} h_{l-1,j}. \tag{2}$$

In the following, we write the equations for the exact Hessian diagonals with respect to weights $\partial^2 \mathcal{L} / \partial W_{l,i,j}^2$, which requires the calculation of $\partial^2 \mathcal{L} / \partial a_{l,i}^2$ first:

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial a_{l,i}^2} &= \frac{\partial}{\partial a_{l,i}} \left(\sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} W_{l+1,k,i} \sigma'(a_{l,i}) \right) \\
&= \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \sum_{p=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial^2 \mathcal{L}}{\partial a_{l+1,k} \partial a_{l+1,p}} \frac{\partial a_{l+1,p}}{\partial a_{l,i}} W_{l+1,k,i} \sigma'(a_{l,i}) + \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} W_{l+1,k,i} \sigma''(a_{l,i}) \\
&= \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \sum_{p=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial^2 \mathcal{L}}{\partial a_{l+1,k} \partial a_{l+1,p}} W_{l+1,p,i} W_{l+1,k,i} \sigma'(a_{l,i})^2 + \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} W_{l+1,k,i} \sigma''(a_{l,i}),
\end{aligned}$$

$$\frac{\partial^2 \mathcal{L}}{\partial W_{l,i,j}^2} = \frac{\partial}{\partial W_{l,i,j}} \left(\frac{\partial \mathcal{L}}{\partial a_{l,i}} h_{l-1,j} \right) = \frac{\partial}{\partial a_{l,i}} \left(\frac{\partial \mathcal{L}}{\partial a_{l,i}} \right) \frac{\partial a_{l,i}}{\partial W_{l,i,j}} h_{l-1,j} = \frac{\partial^2 \mathcal{L}}{\partial a_{l,i}^2} h_{l-1,j}^2. \tag{3}$$

Since, the calculation of $\partial^2 \mathcal{L} / \partial a_{l,i}^2$ depends on the off-diagonal terms, the computation complexity becomes quadratic. Following Becker and Lecun (1989), we approximate the Hessian diagonals by

ignoring the off-diagonal terms, which leads to a backpropagation rule with linear computational complexity for our estimates $\widehat{\frac{\partial^2 \mathcal{L}}{\partial W_{l,i,j}^2}}$ and $\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{l,i}^2}}$:

$$\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{l,i}^2}} \doteq \sum_{k=1}^{|\boldsymbol{a}_{l+1}|} \left[\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{l+1,k}^2}} W_{l+1,k,i}^2 \sigma'(a_{l,i})^2 + \frac{\partial \mathcal{L}}{\partial a_{l+1,k}} W_{l+1,k,i} \sigma''(a_{l,i}) \right], \tag{4}$$

$$\widehat{\frac{\partial^2 \mathcal{L}}{\partial W_{l,i,j}^2}} \doteq \widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{l,i}^2}} h_{l-1,j}^2. \tag{5}$$

However, for the last layer, we use the exact Hessian diagonals $\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{L,i}^2}} \doteq \frac{\partial^2 \mathcal{L}}{\partial a_{L,i}^2}$ since it can be computed in $O(n)$ for the softmax activation function and the cross-entropy loss. More precisely, the exact Hessian diagonals for cross-entropy loss with softmax is simply $\boldsymbol{q} - \boldsymbol{q} \circ \boldsymbol{q}$, where $\boldsymbol{q}$ is the predicted probability vector and $\circ$ denotes element-wise multiplication. We found empirically that this small change makes a large difference in the approximation quality, as shown in Fig. 1a. Hence, unlike Becker and Lecun (1989) who use a Hessian diagonal approximation of the last layer by Eq. 4, we use the exact values directly to achieve more approximation accuracy. We call this method for Hessian diagonal approximation *HesScale* and provide its pseudocode for supervised classification in Algorithm 1. HesScale is not specific to cross-entropy loss as the exact Hessian diagonals can

Algorithm 1 HesScale: Computing Hessian diagonals of a neural network layer in classification

Require: Neural network f and a layer number l
Require: First and second order information $\widehat{\frac{\partial \mathcal{L}}{\partial a_{l+1,i}}}$ and $\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{l+1,i,j}^2}}$, unless $l = L$
Require: Input-output pair $(\boldsymbol{x}, y)$
  Set loss function $\mathcal{L}$ to cross-entropy loss
  Compute preference vector $\boldsymbol{a}_L \leftarrow f(\boldsymbol{x})$ and target one-hot-encoded vector $\boldsymbol{p} \leftarrow \texttt{onehot}(y)$
  Compute the predicted probability vector $\boldsymbol{q} \leftarrow \boldsymbol{\sigma}(\boldsymbol{a}_L)$ using softmax function $\boldsymbol{\sigma}$
  Compute the error $\mathcal{L}(\boldsymbol{p}, \boldsymbol{q})$
  if $l = L$ **then** ▷ Computing Hessian diagonals exactly for the last layer
    Compute $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_L} \leftarrow \boldsymbol{q} - \boldsymbol{p}$ ▷ $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_L}$ consists of elements $\frac{\partial \mathcal{L}}{\partial a_{L,i}}$
    Compute $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_L}$ using Eq. 2 ▷ $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_L}$ consists of elements $\frac{\partial \mathcal{L}}{\partial W_{L,i,j}}$
    $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{a}_L^2}} \leftarrow \boldsymbol{q} - \boldsymbol{q} \circ \boldsymbol{q}$ ▷ $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{a}_L^2}}$ consists of elements $\widehat{\frac{\partial^2 \mathcal{L}}{\partial a_{L,i}^2}}$
    Compute $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{W}_L^2}}$ using Eq. 5 ▷ $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{W}_L^2}}$ consists of elements $\widehat{\frac{\partial^2 \mathcal{L}}{\partial W_{L,i,j}^2}}$
  else if $l \neq L$ **then**
    Compute $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_l}$ and $\partial \mathcal{L} / \partial \boldsymbol{W}_l$ using Eq. 1 and Eq. 2
    Compute $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{a}_l^2}}$ and $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{W}_l^2}}$ using Eq. 4 and Eq. 5
  end if
  return $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_l}$, $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{W}_l^2}}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_l}$, and $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{a}_l^2}}$

be calculated in $O(n)$ for some other widely used loss functions as well. We show this property for negative log-likelihood function with Gaussian and softmax distributions in Appendix A. The computations can be reduced further using a linear approximation for the activation functions (by dropping the second term in Eq. 4), which corresponds to an approximation of the GGN matrix. We call this variation of our method *HesScaleGGN*.

Based on HesScale, we make a stable optimizer, which we call *AdaHesScale*, given in Algorithm 2. We use the same style introduced in Adam (Kingma & Ba 2015), using the squared diagonal approximation instead of the squared gradients to update the moving average. Moreover, we introduce another optimizer based on HesScaleGGN, which we call *AdaHesScaleGGN*.

Algorithm 2 AdaHesScale for optimization

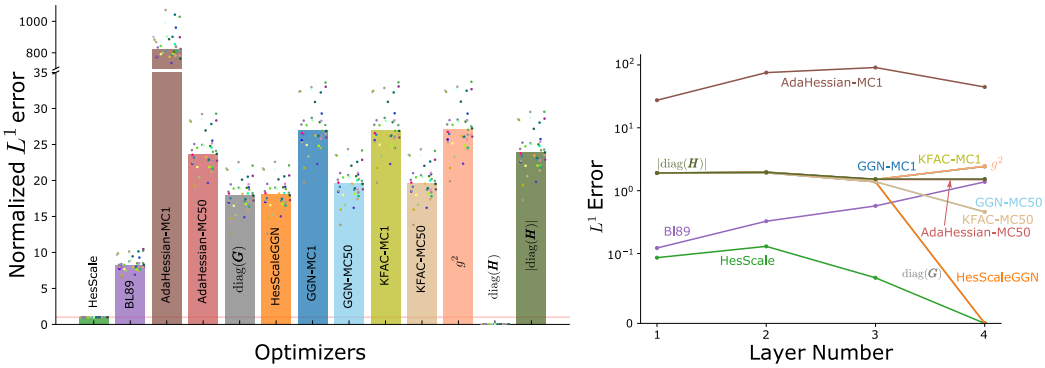
Require: Neural network f with weights $\{\boldsymbol{W}_1, ..., \boldsymbol{W}_L\}$ and a dataset $\mathcal{D}$
Require: Small number $\epsilon \leftarrow 10^{-8}$
Require: Exponential decay rates $\beta_1, \beta_2 \in [0, 1)$
Require: step size α
Require: Initialize $\{\boldsymbol{W}_1, ..., \boldsymbol{W}_L\}$
  Initialize time step $t \leftarrow 0$.
  for l in $\{L, L-1, ..., 1\}$ **do** ▷ Set exponential moving averages at time step 0 to zero
    $\boldsymbol{M}_l \leftarrow 0$; $\boldsymbol{V}_l \leftarrow 0$ ▷ Same size as $\boldsymbol{W}_l$
  end for
  for $(\boldsymbol{x}, y)$ in $\mathcal{D}$ **do**
    $t \leftarrow t + 1$
    $\boldsymbol{r}_{L+1} \leftarrow \boldsymbol{s}_{L+1} \leftarrow \emptyset$ ▷ $\boldsymbol{r}_l$ and $\boldsymbol{s}_l$ stand for $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_l}$ and $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{a}_l^2}}$, respectively
    for l in $\{L, L-1, ..., 1\}$ **do**
      $\boldsymbol{F}_l, \boldsymbol{S}_l, \boldsymbol{r}_l, \boldsymbol{s}_l \leftarrow$ `HesScale`$(f, \boldsymbol{x}, y, l, \boldsymbol{r}_{l+1}, \boldsymbol{s}_{l+1})$. ▷ Check Algorithm 1
      $\boldsymbol{M}_l \leftarrow \beta_1 \boldsymbol{M}_l + (1 - \beta_1)\boldsymbol{F}_l$ ▷ $\boldsymbol{F}_l$ stands for $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_l}$
      $\boldsymbol{V}_l \leftarrow \beta_2 \boldsymbol{V}_l + (1 - \beta_2)\boldsymbol{S}_l$ ▷ $\boldsymbol{S}_l$ stands for $\widehat{\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{W}_l^2}}$
      $\hat{\boldsymbol{M}}_l \leftarrow \boldsymbol{M}_l / (1 - \beta_1^t)$ ▷ Bias-corrected estimate for $\boldsymbol{F}_l$
      $\hat{\boldsymbol{V}}_l \leftarrow \boldsymbol{V}_l / (1 - \beta_2^t)$ ▷ Bias-corrected estimate for $\boldsymbol{S}_l$
      $\boldsymbol{W}_l \leftarrow \boldsymbol{W}_l - \alpha \hat{\boldsymbol{M}}_l \oslash (\hat{\boldsymbol{V}}_l + \epsilon)^{\circ \frac{1}{2}}$ ▷ $\oslash$ is element-wise division
      ▷ $\boldsymbol{A}^{\circ \frac{1}{2}}$ is element-wise square root of $\boldsymbol{A}$
    end for
  end for

4 Approximation quality & scalability of HesScale

In this section, we evaluate HesScale for its approximation quality and computational cost and compare with other methods. These measures constitute the criteria we look for in scalable and efficient methods. For our experiments, we implemented HesScale using the *BackPack* framework (Dangel et al. 2020), which allows easy implementation of backpropagation of statistics other than the gradient.

We start by studying the approximation quality of Hessian diagonals compared to the true values. To measure the approximation quality of the Hessian diagonals for different methods, we use the L^1 distance between the exact Hessian diagonals and their approximations. Our task here is supervised classification, and data examples are randomly generated. We used a network of three hidden layers with *tanh* activations, each containing 16 units. The network weights and biases are initialized randomly. The network has six inputs and ten outputs. For each example pair, we compute the exact Hessian diagonals for each layer and their approximations from each method. All layers' errors are summed and averaged over 1000 data examples for each method. In this experiment, we used 40 different initializations for the network weights, shown as colored dots in Fig. 1a. Each point represents the summed error over network layers, averaged over 1000 examples for each different initialization. In this figure, we show the average error incurred by each method normalized by the average error incurred by HesScale. Any approximation that incurs an averaged error above 1 has a worse approximation than HesScale, and any approximation with an error less than 1 has a better approximation than HesScale. Moreover, we show the layer-wise error for each method in Fig. 1b.

Different Hessian diagonal approximations are considered for comparison with HesScale. We included several deterministic and stochastic approximations for the Hessian diagonals. We also include the approximation of the Fisher Information Matrix done by squaring the gradients and denoted by g^2, which is highly adopted by many first-order methods (e.g., Kingma and Ba, 2015). We compare HesScale with three stochastic approximation methods: AdaHessian (Yao et al. 2021), Kronecker-factored approximate curvature (KFAC) (Martens & Grosse 2015), and the Monte-Carlo (MC) estimate of the GGN matrix (GGN-MC) (Dangel et al. 2020). We also compare HesScale with two deterministic approximation methods: the diagonals of the exact GGN matrix (Schraudolph 2002) (diag$(\boldsymbol{G})$) and the diagonal approximation by Becker and Lecun (1989) (BL89). In KFAC, we extract the diagonals from the block diagonal matrix and show the approximation error



(a) Normalized L^1 error with respect to HesScale

(b) Layer-wise L^1 error

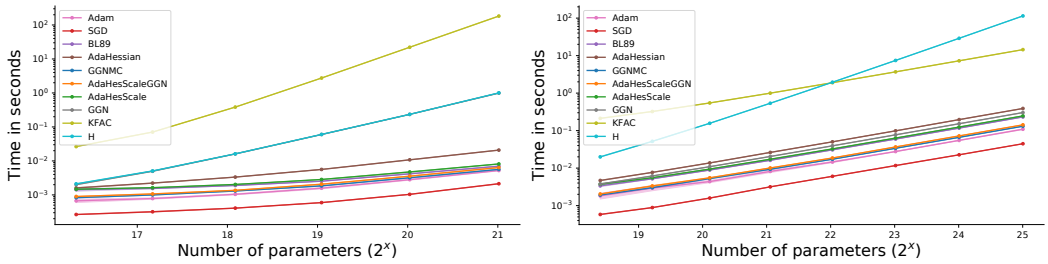
Figure 1: The averaged error for each method is normalized by the averaged error incurred by HesScale. We show 40 initialization points with the same colors across all methods. The norm of the vector of Hessian diagonals $|\text{diag}(\boldsymbol{H})|$ is shown as a reference.

averaged over 1 MC sample (KFAC-MC1) and over 50 MC samples (KFAC-MC50), both per each data example. Since AdaHessian and GGN-MC are already diagonal approximations, we use them directly and show the error with 1 MC sample (GGN-MC1 & AdaHessian-MC1) and with 50 MC samples (GGN-MC50 & AdaHessian-MC50).

HesScale provides a better approximation than the other deterministic and stochastic methods. For stochastic methods, we use many MC samples to improve their approximation. However, their approximation quality is still poor. Methods approximating the GGN diagonals do not capture the complete Hessian information since the GGN and Hessian matrices are different when the activation functions are not piece-wise linear. Although these methods approximate the GGN diagonals, their approximation is significantly better than the AdaHessian approximation. And among the methods for approximating the GGN diagonals, HesScaleGGN performs the best and close to the exact GGN diagonals. This experiment clearly shows that HesScale achieves the best approximation quality compared to other stochastic and deterministic approximation methods.

Next, we perform another experiment to evaluate the computational cost of our optimizers. Our Hessian approximation methods and corresponding optimizers have linear computational complexity, which can be seen from Eq. 4 and Eq. 5. However, computing second-order information in optimizers still incur extra computations compared to first-order optimizers, which may impact how the total computations scale with the number of parameters. Hence, we compare the computational cost of our optimizers with others for various number of parameters. More specifically, we measure the update time of each optimizer, which is the time needed to backpropagate first-order and second-order information and update the parameters.

We designed two experiments to study the computational cost of first-order and second-order optimizers. In the first experiment, we used a neural network with a single hidden layer. The network has 64 inputs and 512 hidden units with tanh activations. We study the increase in computational time when increasing the number of outputs exponentially, which roughly doubles the number of parameters. The set of values we use for the number of outputs is $\{2^4, 2^5, 2^6, 2^7, 2^8, 2^9\}$. The results of this experiment are shown in Fig. 2a. In the second experiment, we used a neural network with multi-layers, each containing 512 hidden units with *tanh* activations. The network has 64 inputs and 100 outputs. We study the increase in computational time when increasing the number of layers exponentially, which also roughly doubles the number of parameters. The set of values we use for the number of layers is $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The results of this experiment are shown in Fig. 2b. The points in Fig. 2a and Fig. 2b are averaged over 30 updates. The standard errors of the means of these points are smaller than the width of each line. On average, we notice that the cost of AdaHessian, AdaHesScale and AdaHesScaleGGN are three, two and 1.25 times the cost of Adam, respectively. It is clear that our methods are among the most computationally efficient approximation method for Hessian diagonals.



(a) Increasing number of outputs in a neural network (b) Increasing number of layers in a neural network

Figure 2: The average computation time for each step of an update is shown for different optimizers. The computed update time is the time needed by each optimizer to backpropagate gradients or second-order information and to update the parameters of the network.

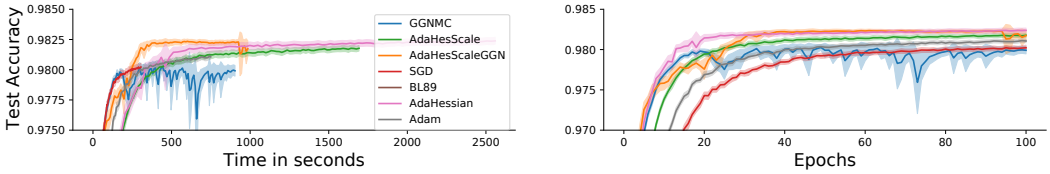
5 Empirical performance of HesScale in Optimization

In this section, we compare the performance of our optimizers—AdaHesScale and AdaHesScaleGGN—with three second-order optimizers: BL89, GGNMC, and AdaHessian. We also include comparisons to two first-order methods: Adam and SGD. We exclude KFAC and the exact diagonals of the GGN matrix from our comparisons due to their prohibitive computations.

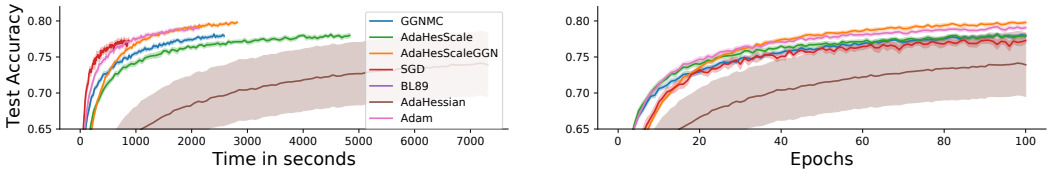
Our optimizers are evaluated in the supervised classification problem with a series of experiments using different architectures and three datasets: MNIST, CIFAR-10, and CIFAR-100. Instead of achieving the state-of-the-art performance with specialized techniques and architectures, we follow the DeepOBS benchmarking work (Schneider et al. 2019) and compare the optimizers in their generic and pristine form using relatively simple networks. It allows us to perform a more fair comparison without extensively utilizing specialized knowledge for a particular task. In the first experiment, we use the MNIST-MLP task from DeepOBS. The images are flattened and used as inputs to a network of three fully connected layers (1000, 500, and 100 units) with *tanh* activations. We train each method for 100 epochs with a batch size of 128. We show the training plots in Fig. 6a with their corresponding sensitivity plots in Appendix C, Fig. 8a. In the second experiment, we use the CIFAR10-3C3D task from the DeepOBS benchmarking tasks. The network consists of three convolutional layers with *tanh* activations, each followed by max pooling. After that, two fully connected layers (512 and 256 units) with *tanh* activations are used. We train each method for 100 epochs with a batch size of 128. We show the training plots in Fig. 6b with their corresponding sensitivity plots in Fig. 8b. In the third experiment, we use the CIFAR100-3C-3D task from DeepOBS. The network is the same as the one used in the second task except for the activations are ELU. We train each method for 200 epochs with a batch size of 128. We show the training plots in Fig. 7b with their corresponding sensitivity plots in Fig. 9b. In the fourth experiment, we use the CIFAR100-ALL-CNN task from DeepOBS with the ALL-CNN-C network, which consists of 9 convolutional layers (Springenberg et al. 2015). We use ELU activations insead of ReLU to differentiate between the performance of AdaHesScale and AdaHesScaleGGN. We show the training plots in Fig. 7a with their corresponding sensitivity plots in Fig. 9a.

In the MNIST-MLP and CIFAR-10-3C3D experiments, we performed a hyperparameter search for each method to determine the best set of β_1, β_2, and α. The range of β_2 is $\{0.99, 0.999, 0.9999\}$ and the range of β_1 is $\{0.0, 0.9\}$. The range of step size is selected for each method to create a convex curve. Our criterion was to find the best hyperparameter configuration for each method in the search space that minimizes the area under the validation loss curve. The performance of each method was averaged over 30 independent runs. Each independent run had the same initial representation for the algorithms used in an experiment. Using each method's best hyperparameter configuration on the validation set, we show the performance of each method against the time in seconds needed to complete the required number of epochs, which better depicts the computational efficiency of the methods. Fig. 3a and Fig. 3b show these results on MNIST-MLP and CIFAR-10 tasks. Moreover, we show the sensitivty of each method to the step size in Fig. 5a and Fig. 5b.

In the CIFAR-100-ALL-CNN and CIFAR-100-3C3D experiments, we used the set of β_1 and β_2 that achieved best roboustness in the previous two tasks. The values used for β_1 and β_2 were 0.9 and

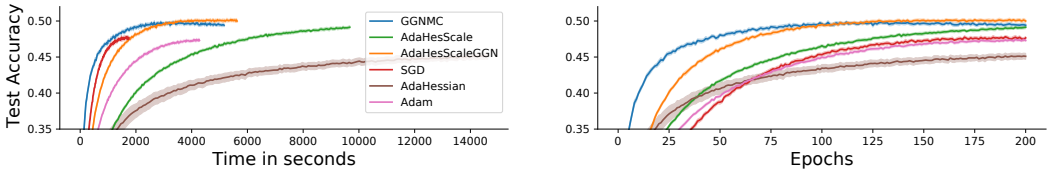


(a) MNIST-MLP

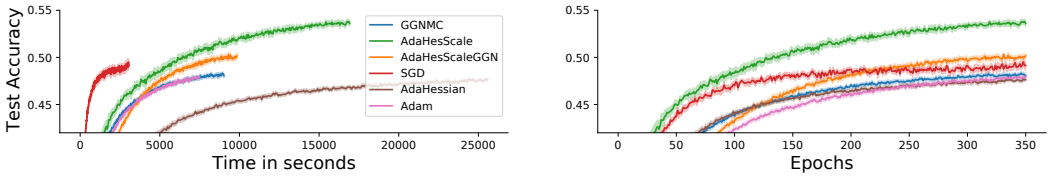


(b) CIFAR-10 3C3D

Figure 3: MNIST-MLP and CIFAR-10 3C3D classification tasks. Each method is trained for 100 epochs. We show the time taken by each algorithm in seconds (left) and we show the learning curves in the number of epochs (right). The performance of each method is averaged over 30 independent runs. The shaded area represents the standard error.



(a) CIFAR-100 3C3D

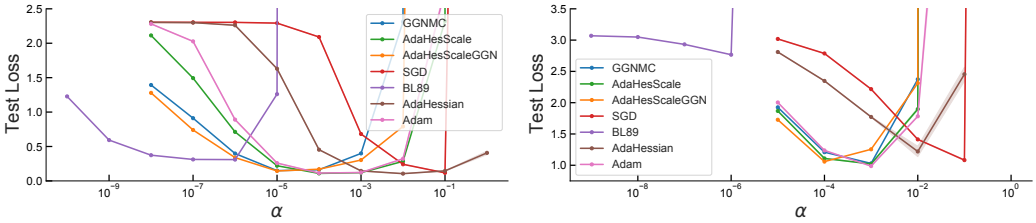


(b) CIFAR-100 All-CNN-C

Figure 4: CIFAR-100 3C3D and CIFAR-100 ALL-CNN classification tasks. Each method from the first task is trained for 200 epochs and each method from the second task is trained for 350 epochs. We show the time taken by each algorithm in seconds (left) and we show the learning curves in the number of epochs (right). The performance of each method is averaged over 30 independent runs. The shaded area represents the standard error.

0.999 respectively. We did a hyperparameter search for each method to determine the best step size using the specified β_1 and β_2. The range of step size is selected for each method to create a convex curve, while ensuring that each method uses a similar search budget and the same interval between search points. Our criterion was to find the best hyperparameter configuration for each method in the search space that minimizes the area under the validation loss curve. The performance of each method was averaged over 30 independent runs. Each independent run had the same initial representation for the algorithms used in an experiment. Using each method's best hyperparameter configuration on the validation set, we show the performance of each method against the time in seconds needed to complete the required number of epochs. Fig. 4a and Fig. 4b show these results on CIFAR-100-ALL-CNN and CIFAR-100-3C3D tasks.

Our results show that all optimizers except for BL89 performed well on the MNIST-MLP task. However, in CIFAR-10, CIFAR-100 3c3d and CIFAR-100 ALL-CNN, we notice that AdaHessian performed worse than all methods except BL89. This result is aligned with AdaHessian's inability to accurately approximate the Hessian diagonals as shown in Fig. 1. Moreover, AdaHessian re-



(a) MNIST (b) CIFAR-10

Figure 5: Sensitivity of the step size for each method on MNIST-MLP and CIFAR-10 3C3D tasks. We select the best values of β_1 and β_2 for each step size α.

quired more computational time compared to all methods, which is also reflected in Fig. 2. While being time-efficient, AdaHesScaleGGN consistently outperformed all methods in CIFAR-10-3C3D and CIFAR-100-3C3D, and it outperformed all methods except AdaHesScale in CIFAR-100 ALL-CNN. This result is aligned with our methods' accurate approximation of Hessian diagonals. Our experiments indicate that incorporating HesScale and HesScaleGGN approximations in optimization methods can be of significant performance advantage in both computation and accuracy. AdaHesScale and AdaHesScaleGGN outperformed other optimizers likely due to their accurate approximation of the diagonals of the Hessian and GGN, respectively.

6 Conclusion

HesScale is a scalable and efficient second-order method for approximating the diagonals of the Hessian at every network layer. Our work is based on the previous work done by Becker and Lecun (1989). We performed a series of experiments to evaluate HesScale against other scalable algorithms in terms of computational cost and approximation accuracy. Moreover, we demonstrated how HesScale can be used to build efficient second-order optimization methods. Our results showed that our methods provide a more accurate approximation and requires small additional computations.

7 Broader Impact

Second-order information is used in domains other than optimization. For example, some works alleviating catastrophic forgetting use a utility measure for the network's connections to protect them. Typically, an auxiliary loss is used between such connections and their old values are weighted by their corresponding importance. Such methods (LeCun et al. 1990, Hassibi & Stork 1993, Dong et al. 2017, Kirkpatrick et al. 2017, Schwarz et al. 2018, Ritter et al. 2018) use the diagonal of the Fisher information matrix or the Hessian matrix as a utility measure for each weight. The quality of these algorithms depends heavily on the approximation quality of the second-order approximation. Second-order information can also be used in neural network pruning. Molchanov et al. (2019) showed that second-order approximation with the exact Hessian diagonals could closely represent the true measure of the utility of each weight.

The accurate and efficient approximation for the diagonals of the Hessian at each layer enables HesScale to be used in many important lines of research. Using this second-order information provides a reliable measure of connection utility. Therefore, using HesScale in these types of problems can potentially improve the performance of neural network pruning methods and regularization-based catastrophic forgetting prevention methods.

References

Amari, S. (1998). Natural Gradient Works Efficiently in Learning. *Neural Computation*, *10*(2), 251–276.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Becker, S. & Lecun, Y. (1989). Improving the convergence of back-propagation learning with second-order methods. *Proceedings of the 1988 Connectionist Models Summer School* (pp. 29–37).

Bekas, C., Kokiopoulou, E., & Saad, Y. (2007). An estimator for the diagonal of a matrix. *Applied Numerical Mathematics*, *57*(11), 1214–1229.

Botev, A., Ritter, H., & Barber, D. (2017). Practical Gauss-Newton optimisation for deep learning. *Proceedings of the 34th International Conference on Machine Learning*, *70*, 557–565.

Chapelle, O. & Erhan, D. (2011). Improved preconditioner for hessian free optimization. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Dangel, F., Kunstner, F., & Hennig, P. (2020). BackPACK: Packing more into backprop. *International Conference on Learning Representations*.

Dong, X., Chen, S., & Pan, S. J. (2017). Learning to prune deep neural networks via layer-wise optimal brain surgeon. Proceedings of the 31st International Conference on Neural Information Processing Systems (pp. 4860-4874).

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(61), 2121–2159.

Hassibi, B. & Stork, D. (1993). Second order derivatives for network pruning: Optimal brain surgeon. *Advances in Neural Information Processing Systems*, *5*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning* (pp. 448-456).

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

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, *114*(13), 3521–3526.

Kunstner, F., Hennig, P., & Balles, L. (2019). Limitations of the empirical fisher approximation for natural gradient descent. *Advances in Neural Information Processing Systems*, *32*.

LeCun, Y., Denker, J., & Solla, S. (1990). Optimal brain damage. *Advances in Neural Information Processing Systems*, *2*.

Luo, L., Xiong, Y., & Liu, Y. (2019). Adaptive gradient methods with dynamic bound of learning rate. *International Conference on Learning Representations*.

Martens, J. (2010). Deep learning via hessian-free optimization. *International Conference on Machine Learning* (pp. 735–742).

Martens, J. (2020). New insights & perspectives on the natural gradient method. *Journal of Machine Learning Research*, *21*(146), 1-76.

Martens, J. & Grosse, R. (2015). Optimizing neural networks with kronecker-factored approximate curvature. *International Conference on Machine Learning* (pp. 2408–2417).

Martens, J. & Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. *International Conference on Machine Learning* (pp. 1033-1040).

Martens, J., Sutskever, I., & Swersky, K. (2012). Estimating the hessian by back-propagating curvature. *International Conference on International Conference on Machine Learning*(pp. 963–970).

Mizutani, E. & Dreyfus, S. E. (2008). Second-order stagewise backpropagation for hessian-matrix analyses & investigation of negative curvature. *Neural Networks*, *21*(2), 193–203.

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., & Kautz, J. (2019). Importance estimation for neural network pruning. *IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 11264–11272).

Reddi, S. J., Kale, S., & Kumar, S. (2018). On the convergence of Adam and beyond. *International Conference on Learning Representations*.

Ritter, H., Botev, A., and Barber, D. (2018). Online structured Laplace approximations for overcoming catastrophic forgetting. *International Conference on Neural Information Processing Systems* (pp. 3742–3752).

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, *29*, 901–909.

Schneider, F., Balles, L., and Hennig, P. (2019). DeepOBS: A deep learning optimizer benchmark suite. *International Conference on Learning Representations*.

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, *14*(7), 1723–1738.

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2018). Progress & compress: A scalable framework for continual learning. *International Conference on Machine Learning* (pp. 4528–4537).

Springenberg, J., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2015). Striving for simplicity: The all convolutional net. *International Conference on Learning Representations* [Workshop].

Tieleman, T., Hinton, G., et al. (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.

Tran, P. T. and Phong, L. T. (2019). On the convergence proof of AMSGrad and a new version. *IEEE Access*, *7*, 61706–61716.

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. *International Conference on Neural Information Processing Systems* (pp. 4151–4161).

Yao, Z., Gholami, A., Shen, S., Keutzer, K., and Mahoney, M. W. (2021). Adahessian: An adaptive second order optimizer for machine learning. *AAAI Conference on Artificial Intelligence*, *35*(12), 10665-10673.

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

A COMPUTATIONAL COMPLEXITY OF THE HESSIAN DIAGONALS OF NEGATIVE LOG-LIKELIHOOD LOSS W.R.T THE PARAMETERS

Here, we analyze the diagonals of the Hessian matrix of the log-likelihood loss function with probabilities coming from a normal distribution or a softmax distribution. We show that the exact computations of the diagonal can be computed with linear complexity since computing the diagonal elements does not depend on the off-diagonal in some cases. In the following, we consider the softmax and Gaussian functions, and we write the exact Hessians diagonals of the log-likelihood function with respect to some parameter vector $\boldsymbol{\theta}$ in both cases.

A.1 SOFTMAX FUNCTION

Consider a cross-entropy function with probabilities coming from a softmax distribution as: $f = -\sum_{i=1}^{|\boldsymbol{q}|} p_i \log q_i(\boldsymbol{\theta})$, where $\boldsymbol{q}$ is some probability vector that depend on $\boldsymbol{\theta}$ and $\boldsymbol{p}$ is the target vector of one-hot encoded class. When $\boldsymbol{q}(\boldsymbol{\theta})$ is parametrized by a softmax function $\boldsymbol{\sigma}$, we can write the gradient of the cross-entropy function with respect to $\boldsymbol{\theta}$ as:

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \boldsymbol{q} - \boldsymbol{p}.$$

Next, we write the exact diagonal element of the Hessian matrix as follows:

$$\text{diag}(\boldsymbol{H}_f) = \text{diag}(\nabla_{\boldsymbol{\theta}}(\boldsymbol{q} - \boldsymbol{p})) = \boldsymbol{\sigma}(\boldsymbol{\theta}) - \boldsymbol{\sigma}(\boldsymbol{\theta}) \circ \boldsymbol{\sigma}(\boldsymbol{\theta}) = \boldsymbol{q} - \boldsymbol{q} \circ \boldsymbol{q},$$

where $\circ$ denotes element-wise multiplication. Computing the exact diagonals of the Hessian matrix depends only on vector operations, which means that we can compute it in $O(n)$. The cross-entropy loss is used with softmax in many important tasks, such as supervised classification and discrete reinforcement learning control.

A.2 MULTIVARIATE GAUSSIAN FUNCTION WITH DIAGONAL COVARIANCE

When $\boldsymbol{q}(\boldsymbol{\theta})$ is parametrized by a Gaussian function. The probability vector $\mathbf{q}$ is given by the multivariate normal distribution. We can write the log-likelihood function as:

$$\log \mathbf{q}(\boldsymbol{\theta}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta} - \boldsymbol{\mu}) + c,$$

where $\boldsymbol{\Sigma}$ is the covariance matrix, c is some constant, and $\boldsymbol{\mu}$ is the mean vector. We can write the gradient of the log-likelihood function as follows:

$$\nabla_{\boldsymbol{\theta}} \log \mathbf{q}(\boldsymbol{\theta}) = -\boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta} - \boldsymbol{\mu}).$$

Finally, we write the exact diagonals of the Hessian matrix as:

$$\text{diag}(\boldsymbol{H}_f) = \text{diag}(\nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta} - \boldsymbol{\mu})) = -\boldsymbol{\Sigma}^{-1}.$$

When the covariance matrix is diagonal, computing the gradient and exact Hessian diagonals can be done in $O(n)$. Log-likelihood functions with Gaussian distributions are used in many important problems, such as continuous reinforcement learning control.

B HESCALE WITH CONVOLUTIONAL NEURAL NETWORKS

Here, we derive the Hessian propagation for convolutional neural networks. Consider a convolutional neural network with $L-1$ layers followed by a fully connected layer that outputs the predicted output $\boldsymbol{q}$. The neural network is parametrized by the set of weights or filters $\{\boldsymbol{W}_1, ..., \boldsymbol{W}_L\}$, where $\boldsymbol{W}_l$ is the filter matrix at the l-th layer with the dimensions $k_{l,1} \times k_{l,2}$, and its element at the ith row and the jth column is denoted by $W_{l,i,j}$. For the simplicity of this proof, we assume that the number of weights at each layer is one, the proof can be extended easily to the general case. The learning algorithm learns the target function f^* by optimizing the loss $\mathcal{L}$. During learning, the parameters of the neural network are changed to reduce the loss. At the layer l, we get the activation output matrix $\boldsymbol{H}_l$ by applying the activation function $\boldsymbol{\sigma}$ to the activation input $\boldsymbol{A}_l$: $\boldsymbol{H}_l = \boldsymbol{\sigma}(\boldsymbol{A}_l)$. We assume here that the activation function is element-wise activation for all layers except for the final layer L, where it becomes the softmax function. We simplify notations by defining $\boldsymbol{H}_0 \doteq \boldsymbol{X}$, where $\boldsymbol{X}$ is

the input sample. The activation output $\boldsymbol{H}_l$ is then convoluted by the weight matrix $\boldsymbol{W}_{l+1}$ of layer $l+1$ to produce the next activation input: $A_{l+1,i,j} = \sum_{m=0}^{k_{l,1}-1} \sum_{n=0}^{k_{l,2}-1} W_{l+1,m,n} H_{l,(i+m),(j+n)}$. We denote the size of the activation output at the l-th layer by $h_l \times w_l$.

The backpropagation equations for the described network are given as follows Rumelhart et al. (1986):

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial A_{l,ij}} &= \sum_{m=0}^{k_{l+1,1}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} \frac{\partial A_{l+1,(i-m),(j-n)}}{\partial A_{l,ij}} \\
&= \sum_{m=0}^{k_{l+1,1}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} \sum_{m'=0}^{k_{l+1,1}-1} \sum_{n'=0}^{k_{l+1,2}-1} W_{l+1,m',n'} \frac{\partial H_{l,(i-m+m'),(j-n+n')}}{\partial A_{l,i,j}} \\
&= \sum_{m=0}^{k_{l+1,1}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} W_{l+1,m,n} \sigma'(A_{l,i,j}),
\end{aligned} \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l,i,j}} = \sum_{m=0}^{h_l-k_{l,1}} \sum_{n=0}^{w_l-k_{l,2}} \frac{\partial \mathcal{L}}{\partial A_{l,m,n}} \frac{\partial A_{l,m,n}}{\partial W_{l,i,j}} = \sum_{m=0}^{h_l-k_{l,1}} \sum_{n=0}^{w_l-k_{l,2}} \frac{\partial \mathcal{L}}{\partial A_{l,i,j}} H_{l-1,(i+m),(j+n)}. \tag{7}$$

In the following, we write the equations for the exact Hessian diagonals with respect to weights $\partial^2\mathcal{L}/\partial W^2_{l,i,j}$, which requires the calculation of $\partial^2\mathcal{L}/\partial A^2_{l,i,j}$ first:

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial A^2_{l,i,j}} &= \frac{\partial}{\partial A_{l,i,j}} \left[\sum_{m=0}^{k_{l+1,1}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} W_{l+1,m,n} \sigma'(A_{l,i,j}) \right] \\
&= \sum_{m,p=0}^{k_{l+1,2}-1} \sum_{n,q=0}^{k_{l+1,2}-1} \frac{\partial^2 \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)} \partial A_{l+1,(i-p),(j-q)}} \frac{\partial A_{l+1,(i-p),(j-q)}}{\partial A_{l,i,j}} W_{l+1,m,n} \sigma'(A_{l,i,j}) \\
&+ \sum_{m=0}^{k_{l+1,2}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} W_{l+1,m,n} \sigma''(A_{l,i,j})
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial W^2_{l,ij}} &= \frac{\partial}{\partial W_{l,i,j}} \left[\sum_{m=0}^{h_l-k_{l,1}} \sum_{n=0}^{w_l-k_{l,2}} \frac{\partial \mathcal{L}}{\partial A_{l,i,j}} H_{l-1,(i+m),(j+n)} \right] \\
&= \sum_{m,p=0}^{h_l-k_{l,1}} \sum_{n,q=0}^{w_l-k_{l,2}} \frac{\partial^2 \mathcal{L}}{\partial A_{l,i,j} \partial A_{l,p,q}} \frac{\partial A_{l,p,q}}{\partial W_{l,i,j}} H_{l-1,(i+m),(j+n)}
\end{aligned}$$

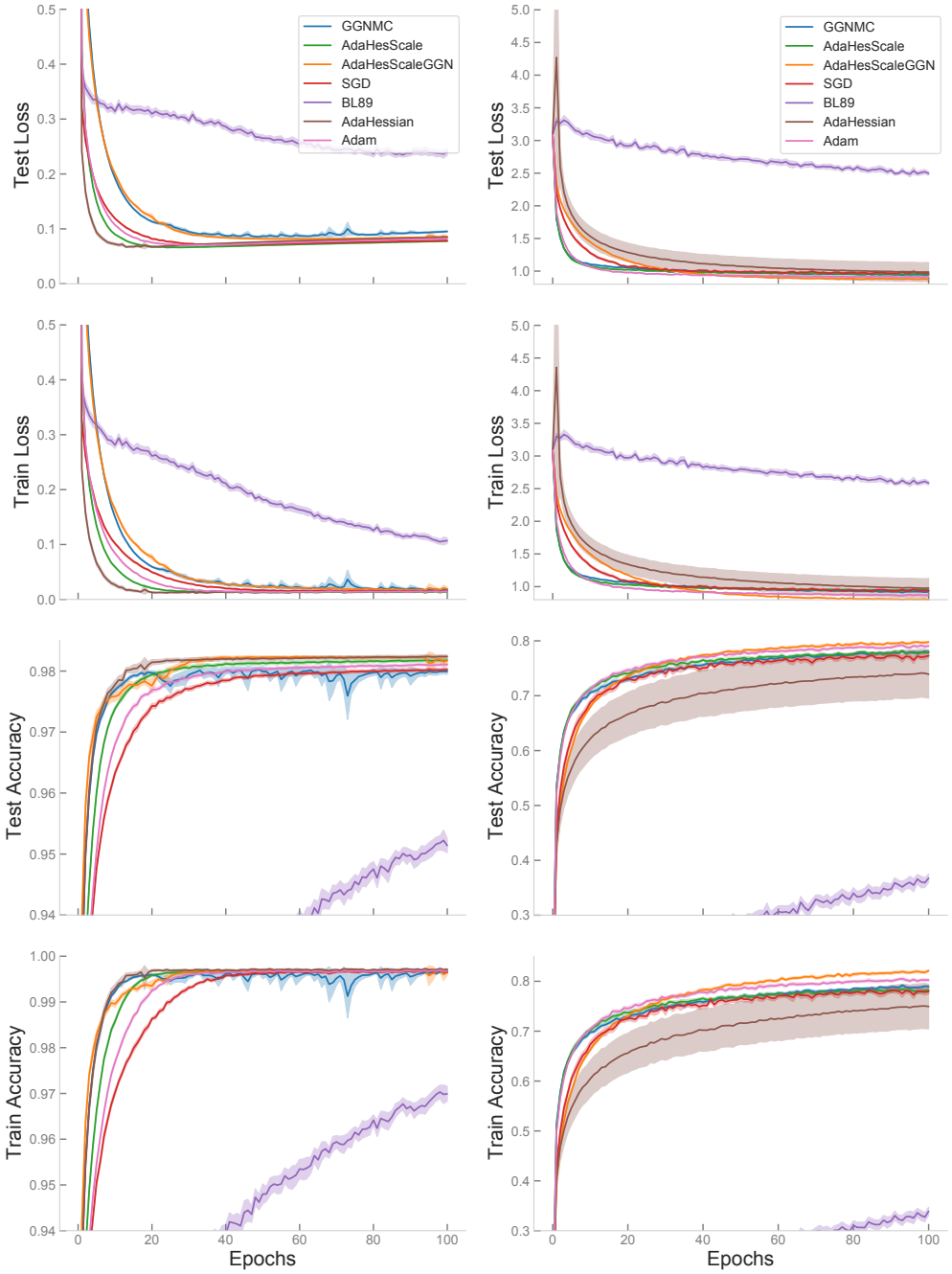
Since, the calculation of $\partial^2\mathcal{L}/\partial A^2_{l,i,j}$ and $\partial^2\mathcal{L}/\partial W^2_{l,i,j}$ depend on the off-diagonal terms, the computation complexity becomes quadratic. Following Becker and Lecun (1989), we approximate the Hessian diagonals by ignoring the off-diagonal terms, which leads to a backpropagation rule with linear computational complexity for our estimates $\widehat{\frac{\partial^2\mathcal{L}}{\partial W^2_{l,i,j}}}$ and $\widehat{\frac{\partial^2\mathcal{L}}{\partial A^2_{l,i,j}}}$:

$$\begin{aligned}
\widehat{\frac{\partial^2 \mathcal{L}}{\partial A^2_{l,i,j}}} &\doteq \sum_{m=0}^{k_{l+1,2}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\widehat{\partial^2 \mathcal{L}}}{\partial A^2_{l+1,(i-m),(j-n)}} W^2_{l+1,mn} \sigma'(A_{l,ij})^2 \\
&+ \sum_{m=0}^{k_{l+1,2}-1} \sum_{n=0}^{k_{l+1,2}-1} \frac{\partial \mathcal{L}}{\partial A_{l+1,(i-m),(j-n)}} W_{l+1,mn} \sigma''(A_{l,i,j}),
\end{aligned} \tag{8}$$

$$\widehat{\frac{\partial^2 \mathcal{L}}{\partial W^2_{l,i,j}}} \doteq \sum_{m=0}^{h_l-k_{l,1}} \sum_{n=0}^{w_l-k_{l,2}} \widehat{\frac{\partial^2 \mathcal{L}}{\partial A^2_{l,i,j}}} H^2_{l-1,(i+m),(j+n)}. \tag{9}$$

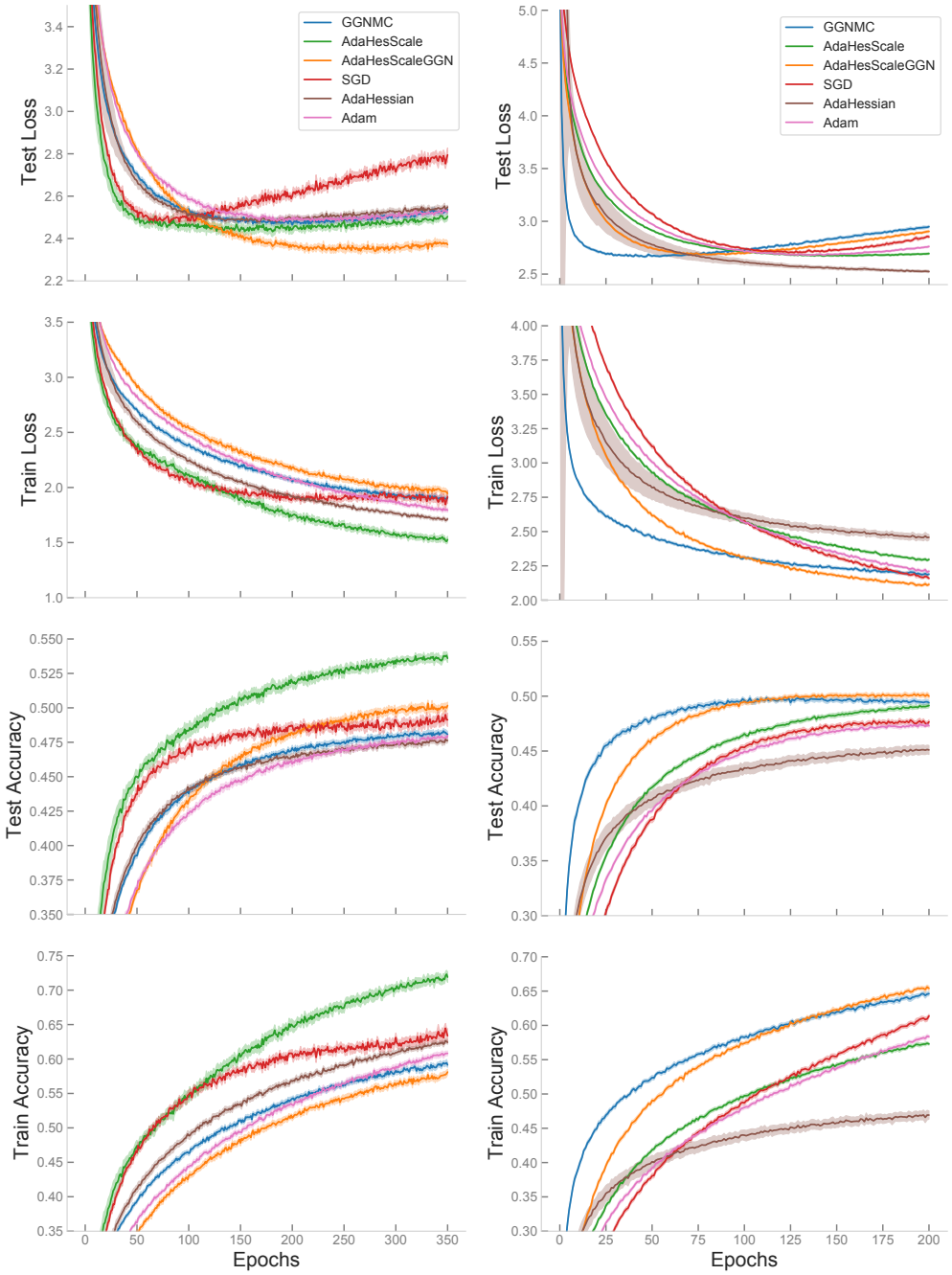
C OPTIMIZATION PLOTS IN THE NUMBER OF EPOCHS

We give the training loss, training accuracy, validation loss, validation accuracy, test loss, and test accuracy for each of the methods we include in our comparison in Fig. 6 and Fig. 7. Moreover, we give the sensitivity plots for β_1, β_2, and α for each method in Fig. 8 and Fig. 9.



(a) MNIST (b) CIFAR-10

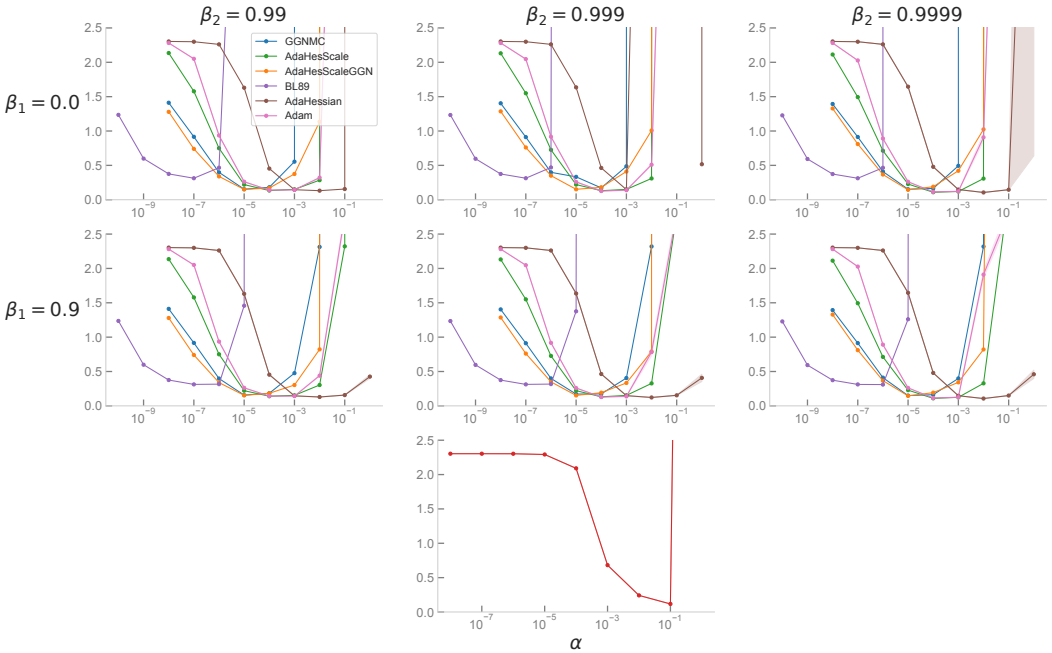
Figure 6: Learning curves of each algorithm on two tasks, MNIST-MLP and CIFAR-10 3C3D, for 100 epochs. We show the best configuration for each algorithm on the validation set. The best parameter configuration for each algorithm is selected based on the area under the curve for the validation loss.



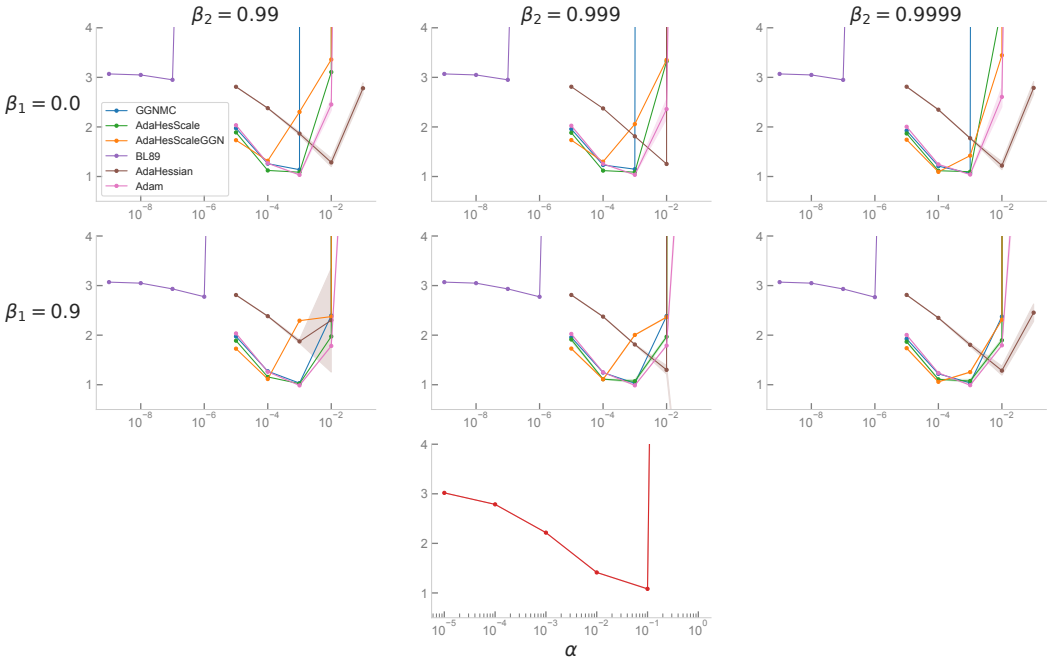
(a) CIFAR-100 All CNN

(b) CIFAR-100 3C3D

Figure 7: Learning Curves of each algorithm on CIFAR-100 with All-CNN and 3C3D architectures, for 100 epochs. We show the best configuration for each algorithm on the validation set. The best parameter configuration for each algorithm is selected based on the area under the curve for the validation loss.

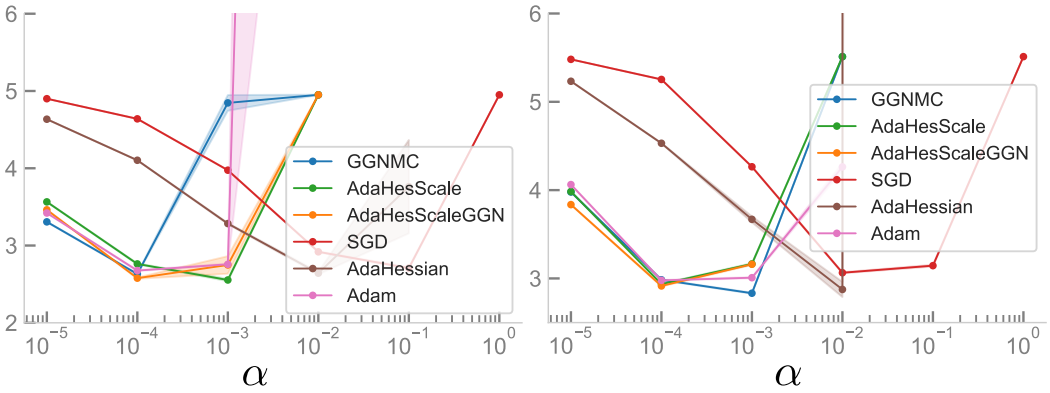


(a) MNIST



(b) CIFAR-10

Figure 8: Parameter Sensitivity study for each algorithm on two data sets, MNIST and CIFAR-10. The range of β_2 is $\{0.99, 0.999, 0.9999\}$ and the range of β_1 is $\{0.0, 0.9\}$. Each point for each algorithm represents the average test loss given a set of parameters.



(a) CIFAR-100 All-CNN

(b) CIFAR-100 3C3D

Figure 9: Parameter Sensitivity study for each algorithm on CIFAR-100 with All-CNN and 3C3D architectures. The range of step size is $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}\}$. We choose β_1 to be equal to 0.9 and β_2 to be equal to 0.999. Each point for each algorithm represents the average test loss given a set of parameters.