Using Second-Order Information in Training Deep Neural Networks

Yi Ren

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2022

Abstract

Using Second-Order Information in Training Deep Neural Networks

Yi Ren

In this dissertation, we are concerned with the advancement of optimization algorithms for training deep learning models, and in particular about practical second-order methods that take into account the structure of deep neural networks (DNNs). Although first-order methods such as stochastic gradient descent [75] have long been the predominant optimization algorithm used in deep learning, second-order methods are of interest because of their ability to use curvature information to accelerate the optimization process.

After the presentation of some background information in Chapter 1, Chapters 2 and 3 focus on the development of practical quasi-Newton methods for training DNNs. We analyze the Kronecker-factored structure of the Hessian matrix of multi-layer perceptrons and convolutional neural networks and consequently propose block-diagonal Kronecker-factored quasi-Newton methods named K-BFGS and K-BFGS(L). To handle the non-convexity nature of DNNs, we also establish new double damping techniques for our proposed methods. Our K-BFGS and K-BFGS(L) methods have memory requirements comparable to first-order methods and experience only mild overhead in terms of per-iteration time complexity.

In Chapter 4, we develop a new approximate natural gradient method named Tensor Normal Training (TNT), in which the Fisher matrix is viewed as the covariance matrix of a tensor normal distribution (a generalized form of the normal distribution). The tractable Kronecker-factored ap-

proximation to the Fisher information matrix that results from this approximation enables TNT to enjoy memory requirements and per-iteration computational costs that are only slightly higher than those for first-order methods. Notably, unlike KFAC [62] and K-BFGS/K-BFGS(L), TNT only requires the knowledge of the shape of the trainable parameters of a model and does not depend on the specific model architecture.

In Chapter 5, we consider the subsampled versions of Gauss-Newton and natural gradient methods applied to DNNs. Because of the low-rank nature of the subsampled matrices, we make use of the Sherman-Morrison-Woodbury formula along with backpropagation to efficiently compute their inverse. We also show that, under rather mild conditions, the algorithm converges to a stationary point if Levenberg-Marquardt damping is used.

The results of a substantial number of numerical experiments are reported in Chapters 2, 3, 4 and 5, in which we compare the performance of our methods to state-of-the-art methods used to train DNNs, that demonstrate the efficiency and effectiveness of our proposed new second-order methods.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my advisor Professor Donald Goldfarb. This thesis would not have been possible without his guidance. He has been so supportive to me in every aspect, not only for research, but also for my career and personally. I'm deeply honored to be his student.

I would like to thank the committee members of this dissertation, including Professor Daniel Bienstock, Professor Chrsitian Kroer, Professor Cedric Josz, and Professor Katya Scheinberg. I really appreciate their helpful feedback for the thesis and also learned a lot from the discussion with them.

I would like to thank the IEOR department staff, especially Winsor Yang and Lizbeth Morales, who take care all of the PhD students in the deparment so well, including myself.

I would like to thank all the faculty and fellow students in the department, for being such a supportive community.

Lastly, I would like to thank my girlfriend, for giving me unlimited support during my PhD study and going through the pandemic time with me together. I also would like to thank my parents, for supporting me unconditionally throughout my life.

To my parents.

Preface

The main chapters of this thesis are closely adapted from the following papers:

- Chapter 2 is based on the article *Practical quasi-Newton methods for training deep neural networks* [35]. This article is joint with Donald Goldfarb and Achraf Bahamou.
- Chapter 3 is based on the article *Kronecker-factored quasi-Newton methods for convolutional neural networks* [73]. This article is joint with Donald Goldfarb and Achraf Bahamou.
- Chapter 4 is based on the article *Tensor normal training for deep learning models* [74]. This article is joint with Donald Goldfarb.
- Chapter 5 is based on the article *Efficient subsampled Gauss-Newton and natural gradient methods for training neural networks* [72]. This article is joint with Donald Goldfarb.

Chapter 1: Introduction and Background

1.1 Introduction

The field of deep learning has thrived and advanced tremendously in the past two decades. In the past few years, in particular, many important deep learning models, including ResNet [40], Transformer [85], BERT [28], GPT-3 [16], DALL·E 2 [70], have been proposed and proven to be successful in many applications such as computer vision and natural language processing. However, optimization, an indispensable component of any deep learning application, has been studied less than other aspects of deep learning, even though it plays an important part in the performance and efficiency of deep learning models.

In training deep learning models, practitioners predominantly use first-order methods, such as stochastic gradient descent (SGD) [75] and its variants that use adaptive learning rates [29, 42, 46]. As their name suggests, these methods update the parameters of the model using only gradient information. On the other hand, second-order methods, which take the curvature of a problem into account and are able to optimize much faster than first-order methods when applied to classical optimization problems, are used much less often in practice for training deep neural networks (DNNs).

The number of trainable parameters n in a DNN is usually enormous. As a result, for second-order methods, computing the Hessian matrix, which takes at least $O(n^2)$ time and storage, is prohibitive, let alone inverting the Hessian, which takes $O(n^3)$ time. Quasi-Newton methods, including BFGS [17, 31, 34, 81] and limited-memory BFGS (L-BFGS) [54], avoid the Hessian inversion computation by maintaining an approximation to the inverse Hessian. Nevertheless, computing and storing a full $n \times n$ BFGS approximation or storing a modest number of (step, change in gradient) vector pairs for use in an L-BFGS implementation is still impractical.

Because of the difficulty of computing the Hessian matrix in DNN settings, the use of other surrogates for the Hessian, including Gauss-Newton, Fisher, and empirical Fisher matrices have been proposed and used in training DNNs, balancing the trade-off between utilizing curvature information and ease of computation. Martens [59] proposed approximately solving the linear equations appearing in the Gauss-Newton method by applying the linear conjugate gradient method, leading to a "Hessian-free" method. KFRA [15] is another approximate Gauss-Newton method that recursively computes the Kronecker-factored Gauss-Newton matrices for each layer in the DNN.

Natural gradient methods that precondition the gradient direction with the Fisher (information) matrix are proposed in Amari [2] and Amari *et al.* [3]. To mitigate the high computational cost and memory requirements of using a Fisher pre-conditioner in DNNs, KFAC [62] approximates the Fisher matrix with a block-diagonal Kronecker-factored approximation for DNNs, which was later extended to more sophisticated deep learning models [38, 90, 61] and also led to other variants such as EKFAC [33] and SKFAC [84].

Empirical Fisher, a further approximation to the Fisher matrix, is also related to several well studied first- and second-order methods. The class of adaptive learning rate methods, including AdaGrad [29], RMSprop [42], and Adam [46], can all be viewed as using the "square root" of a diagonal approximation to the empirical Fisher matrix as the preconditioning matrix. Shampoo [39, 5] extended the adaptive learning rate methods to a block-diagonal Kronecker-factored version and, unlike KFAC, does not depend on the specific type of DNN models, using only its tensor representation. Bahamou *et al.* [9] proposed a mini-block (empirical) Fisher method, which consists of even finer blocks and lies in between the adaptive learning rate methods and Kronecker-factored methods such as KFAC and Shampoo.

Besides utilizing various preconditioning matrices, there are several other important features about second-order methods for training DNNs that distinguish them from classical optimization algorithms. First and foremost, many of them, including KFAC [62], KFRA [15], and Shampoo [39, 5], use block-diagonally preconditioning matrices, where each block corresponds to one layer or one chunk of trainable parameters in the model. This "block-diagonal approximation" helps

second-order methods reduce their storage and time complexities in deep learning settings and is naturally aligned with the "modular" nature of deep learning models.

Secondly, Kronecker-factored and/or low-rank approximations to the preconditioning matrices are usually used to further reduce the complexities of the algorithms, which are always motivated by the structure of the model and the use of mini-batches in the training process.

Lastly, backpropagation is heavily used in training DNNs. Besides computing the gradient with backpropagation, which is essentially required by every method, backpropagation can also be used in computing Gauss-Newton and Fisher matrices, as well as in computing the product between the preconditioning matrices and a given vector (see e.g., Martens [59] and Yao *et al.* [93]).

1.1.1 K-BFGS

In Chapters 2 and 3, we consider the development of practical stochastic quasi-Newton, and in particular Kronecker-factored block-diagonal BFGS and L-BFGS methods, for training DNNs.

In Chapter 2, we propose approximating the Hessian of the multi-layer perceptron models by a block-diagonal matrix and use the structure of the gradient and Hessian to further approximate these blocks, each of which corresponds to a layer, as the Kronecker product of two much smaller matrices. This is analogous to the approach in KFAC [62], which computes a Kronecker-factored block-diagonal approximation to the Fisher matrix in a stochastic natural gradient method. Because of the indefinite and highly variable nature of the Hessian in a DNN, we also propose a new damping approach to keep the upper as well as the lower bounds of the BFGS and L-BFGS approximations bounded. In tests on autoencoder feed-forward neural network models with either nine or thirteen layers applied to three datasets, our methods outperformed or performed comparably to KFAC and state-of-the-art first-order stochastic methods.

In Chapter 3, we extend the methods proposed in Chapter 2 to enable them to be applied to convolutional neural networks (CNNs), by analyzing the Kronecker-factored structure of the Hessian matrix of convolutional layers. Several improvements to the methods in Chapter 2 are also proposed that can be applied to both MLPs and CNNs. These new methods have memory

requirements comparable to first-order methods and much less per-iteration time complexity than those in Chapter 2. Moreover, convergence results are proved for a variant under relatively mild conditions. Finally, we compared the performance of our new methods against several state-of-the-art (SOTA) methods on MLP autoencoder and CNN problems, and found that they outperformed the first-order SOTA methods and performed comparably to the second-order SOTA methods.

1.1.2 Tensor Normal Training

In Chapter 4, we propose and analyze a new approximate natural gradient method, *Tensor Normal Training* (TNT), based on the so-called *tensor normal* distribution [58], which like Shampoo, only requires knowledge of the shape of the training parameters. By approximating the probabilistically based Fisher matrix, as opposed to the empirical Fisher matrix, our method uses the block-wise covariance of the sampling based gradient as the pre-conditioning matrix. Moreover, the assumption that the sampling-based (tensor) gradient follows a tensor normal distribution ensures that its covariance has a Kronecker separable structure, which leads to a tractable approximation to the Fisher matrix. Consequently, TNT's memory requirements and per-iteration computational costs are only slightly higher than those for first-order methods. In our experiments, TNT exhibited superior optimization performance to SOTA first-order methods, and comparable optimization performance to the SOTA second-order methods KFAC and Shampoo. Moreover, TNT demonstrated its ability to generalize as well as first-order methods, while using fewer epochs.

1.1.3 Subsampled Gauss-Newton and Natural Gradient Methods

In Chapter 5, we present practical Levenberg-Marquardt variants of Gauss-Newton and natural gradient methods for solving non-convex optimization problems that arise in training deep neural networks involving enormous numbers of variables and huge data sets. Our methods use subsampled Gauss-Newton or Fisher information matrices and either subsampled gradient estimates (fully stochastic) or full gradients (semi-stochastic), which, in the latter case, we prove convergent to a stationary point. By using the Sherman-Morrison-Woodbury formula with automatic differ-

entiation (backpropagation) we show how our methods can be implemented to perform efficiently. Finally, numerical results are presented to demonstrate the effectiveness of our proposed methods.

1.2 Background

1.2.1 Empirical Risk Minimization (ERM)

The most common optimization problems that need to be solved in training deep neural networks, and to a broader extent, in general machine learning, can be formulated as an *Empirical Risk Minimization* (ERM) problem

$$\min_{\theta \in \mathbb{R}^d} L(\theta) := \frac{1}{I} \sum_{i=1}^{I} l_i(\theta). \tag{1.1}$$

In applying ERM to machine learning, the index i usually corresponds to one data-point in a given (training) dataset $S = \{(x_i, y_i) \mid i = 1, ..., I\}$.

Moreover, in machine learning, the goal is usually to obtain an output $\hat{y}_i$ based on the input x_i, that is as close to a known value y_i as possible. Particular examples include regression problems, where y_i is a real number or a vector consisting of real numbers, and classification problems, where y_i denotes a label (or class). In order to achieve such a goal, we first define a model f such that $\hat{y}_i = f_\theta(x_i)$ where the vector θ embodies the parameters of the model. Here f can denote any kind of parametric model, from the simplest linear regression or logistic regression model, to deep neural networks, e.g., multilayer perceptrons and convolutional neural networks.

Given $f_\theta(x_i))$, we define

$$l_i(\theta) := \mathcal{L}(y_i, \hat{y}_i) = \mathcal{L}(y_i, f_\theta(x_i)),$$

where $\mathcal{L}$ is a loss function that measures the difference between the ground truth y_i and the output of the model $\hat{y}_i$. There are numerous loss functions that are used in machine learning problems, (probably) the most common ones that are used being mean squared error for regression problems

and cross entropy for classification problems.

In order to solve (1.1), practitioners usually use an iterative method of the form

$$\theta_t = \theta_{t-1} - \alpha_t H_t g_t \tag{1.2}$$

for $t = 1, 2, ...$, where $\alpha_t \in R$ is the step size (i.e., learning rate) and $\theta \in \mathbb{R}^d$ is the vector of parameter, which is initialized as θ_0. $g_t \in \mathbb{R}^d$ is the gradient $\nabla_\theta L$ of $L(\theta)$ or an estimate of $\nabla_\theta L$, whereas $H_t \in R^{d\times d}$ is the *preconditioning matrix*. Note that if $H_t = I$ and $g_t = \nabla_\theta L$, the above method is the gradient descent method. If $H_t = I$ and g_t is a stochastic estimate to $\nabla_\theta L$, the above method is the stochastic gradient descent (SGD) [75] method. When $H_t \neq I$, the last method is referred to as a *preconditioned SGD* method. H_t is usually associated with an estimate of the "curvature" of the objective function L. Different choices of H_t, including the Hessian and its surrogates, will be discussed in Section 1.2.2 and other chapters. In particular, in the deep learning setting, because the models always have a very large number of parameters, one needs to be mindful about the amount of memory required to store H_t and the cost of computing $H_t g_t$. For example, a naive implementation of Newton's method [67], i.e., letting H_t to be the inverse of the Hessian, would incur (at least) $O(d^2)$ memory to store H_t and $O(d^3)$ time to compute the inverse of the Hessian.

1.2.2 A Brief Introduction on Various Preconditioning Matrices

Hessian

The Hessian matrix of the second derivatives of L w.r.t. the parameters θ, is directly related to the landscape of the function $L(\theta)$. Hence, the inverse Hessian is naturally the first choice for the preconditioning matrix H_t in (1.2).

Numerous methods have been proposed to mitigate the high storage and computational complexity of a straight-forward Newton method, where H_t is computed as the exact inverse of the Hessian. These methods include the conjugate gradient (CG) method [67], Newton-CG [77], the

trust region methods [67, 11], and quasi-Newton methods [17, 31, 34, 81, 54].

One of the major difficulties with using the inverse Hessian as a preconditioning matrix for training DNNs is that the function $L(\theta)$ is not guaranteed to be convex. As a result, the Hessian matrix is not necessarily even positive semi-definite, and many of the aforementioned methods need to be modified to deal with non-convexity. To take a closer look, we first expand the gradient $\frac{\partial L}{\partial \theta}$ by the chain rule:

$$\frac{\partial l}{\partial \theta} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta}. \tag{1.3}$$

Note that we now focus on a single date-point and omit the index i. We then apply the chain rule to (1.3) again, resulting in the expression for Hessian (see, e.g., Martens [60]):

$$\begin{aligned} \frac{\partial^2 l}{\partial \theta^2} &= \frac{\partial}{\partial \theta}\left(\frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta}\right) = \left(\frac{\partial \hat{y}}{\partial \theta}\right)^\top \frac{\partial^2 l}{\partial \hat{y}^2} \frac{\partial \hat{y}}{\partial \theta} + \sum_{j=1}^{m} \left[\frac{\partial l}{\partial \hat{y}}\right]_j \frac{\partial^2 [\hat{y}]_j}{\partial \theta^2} \\ &:= J_f^\top H_l J_f + \sum_{j=1}^{m} \left[\frac{\partial l}{\partial \hat{y}}\right]_j \frac{\partial^2 [\hat{y}]_j}{\partial \theta^2}, \end{aligned} \tag{1.4}$$

where we define $J_f := \frac{\partial \hat{y}}{\partial \theta}$ as the *Jacobian* matrix, and $H_l := \frac{\partial^2 l}{\partial \hat{y}^2}$ is the Hessian of the loss function l w.r.t. the output $\hat{y}$ of the model. For common loss functions used in deep learning, H_l is usually positive semi-definite (see more discussion about this below). Hence, the non-convexity of Hessian comes from the second term of (1.4).

Lastly, for the given dataset S, the curvature of the function $L(\theta)$ is associated with the average Hessian $\frac{1}{I}\sum_{i=1}^{I} \frac{\partial^2 l_i(\theta)}{\partial \theta^2}$.

Gauss-Newton Matrix

One way to resolve the non-convexity issue of the Hessian is to only keep the first term in (1.4), leading to the *(Generalized) Gauss-Newton (GGN)* matrix

$$G = J_f^\top H_l J_f,$$

which is usually guaranteed to be positive semi-definite (PSD). To be more specific, H_l is usually PSD for commonly used loss functions, e.g.,

- when the loss function $\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|_2^2$ is mean squared error, we have that $H_l = I$;
- when $\mathcal{L}(y, \hat{y})$ is binary cross entropy with a sigmoid function, i.e.,

$$\mathcal{L}(y, \hat{y}) = \sum_{j=1}^{m} y_j \log\left(\text{sigmoid}(\hat{y}_j)\right) + (1 - y_j) \log\left(1 - \text{sigmoid}(\hat{y}_j)\right), \tag{1.5}$$

  we have that $H_l = \text{diag}\left(\text{sigmoid}(\hat{y})\right) \succeq 0$;

- when $\mathcal{L}(y, \hat{y})$ is cross entropy combined with a softmax function, i.e.,

$$\mathcal{L}(y, \hat{y}) = \sum_{j=1}^{m} y_j \log\left(\text{softmax}(\hat{y})_j\right) := \sum_{j=1}^{m} y_j \log a_j, \tag{1.6}$$

  where $a := \text{softmax}(\hat{y})$, we have that $H_l = \text{diag}(a) - aa^\top \succeq 0$.

As a result, the GGN matrix G is also guaranteed to be PSD and the updating direction in (1.2) is guaranteed to be a descending direction.

Besides being positive semi-definite, GGN also approximates the Hessian matrix well, especially near the optimal point. To see this, note that if $\left[\frac{\partial l}{\partial \hat{y}}\right]_j \approx 0$ for any $j = 1, ..., m$, i.e., changing any $\hat{y}_j$ will not further decrease the value of l, we have that $H \approx G$, i.e., the GGN is almost the same as the Hessian. See Martens [60] for more discussion on this.

Lastly, the word "generalized" in the name "(Generalized) Gauss-Newton" is because traditionally, Gauss-Newton matrix is only defined when loss function is mean squared error, i.e., $\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|_2^2$ and $G = J_f^\top J_f$, which we will refer to as classical Gauss-Newton.

Fisher Matrix

The Fisher (information) matrix [71, 1], another preconditioning matrix that is closely related to GGN, is defined as

$$F = \mathbb{E}_{x,y\sim P_{x,y}(\theta)}\left[\frac{\partial \log p(y \mid x,\theta)}{\partial \theta}\left(\frac{\partial \log p(y \mid x,\theta)}{\partial \theta}\right)^{\top}\right], \tag{1.7}$$

where $p(y \mid x, \theta)$ is the density function of the model's *learned distribution of* y, given x and θ, Moreover, $P_{x,y}(\theta)$ is the *learned distribution of* (x, y), given θ, with its density function $p(x, y \mid \theta) = p(y \mid x, \theta)q(x)$, where $q(x)$ is the density function of x. When we use Fisher as the preconditioning matrix H_t in (1.2), the updating direction $H_t g_t$ is called the *natural gradient* direction.

To see the connection between the Fisher matrix and GGN, we first note that commonly used loss function can usually be interpreted as the negative log likelihood of some distribution (see, e.g., Martens [60] and Martens and Grosse [62]). For example,

- minimizing the mean squared error loss function $\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|_2^2$ is equivalent to minimizing the negative log likelihood of a normal distribution with fixed variance and $\hat{y}$ as the mean parameter of the distribution;

- minimizing the binary cross entropy (with a sigmoid function) $\mathcal{L}(y, \hat{y})$ as defined in (1.5) is equivalent to minimizing the negative log likelihood of a Bernoulli distribution with $\text{sigmoid}(\hat{y}_j)$ $(j = 1, ..., m)$ as the parameter of the distribution;

- minimizing the cross entropy combined (with a softmax function) as define in (1.6) is equivalent to minimizing the negative log likelihood of a categorical distribution with $\text{softmax}(\hat{y})$ as the parameters of the distribution.

In other words, for these aforementioned loss functions, we have that $\mathcal{L}(y, \hat{y}) = \mathcal{L}(y, f_\theta(x)) = -\log p(y \mid x, \theta)$, ignoring some constant terms in the right hand side.

As a result of the above interpretation of the loss function being negative log likelihood, Fisher matrix is equivalent to GGN for certain loss functions (again, see Martens [60] and Martens and

Grosse [62]). Loosely speaking, we can rewrite $\frac{\partial \log p(y \mid x,\theta)}{\partial \theta}$ by chain rule as

$$\frac{\partial \log p(y \mid x, \theta)}{\partial \theta} = \frac{\partial \hat{y}}{\partial \theta} \frac{\partial \log p(y \mid x, \theta)}{\partial \hat{y}} = J_f \frac{\partial \log p(y \mid x, \theta)}{\partial \hat{y}}.$$

Then, the Fisher matrix becomes

$$F = \mathbb{E}_{x,y \sim P_{x,y}(\theta)} \left[J_f \frac{\partial \log p(y \mid x, \theta)}{\partial \hat{y}} \left(\frac{\partial \log p(y \mid x, \theta)}{\partial \hat{y}} \right)^\top J_f^\top \right].$$

After some manipulations, one can prove that this is exactly the expression for GGN.

Notably, the equivalence between Fisher and GGN allows one to approximate them more efficiently. Without computing the full Jacobian matrix, one can first sample x from the density function $q(x)$ of x, which is usually approximated by simply taking x from the training set. Then, one performs the forward pass of the model to obtain $p(y \mid x, \theta)$, sample y from $p(y \mid x, \theta)$, and compute $\frac{\partial \log p(y \mid x,\theta)}{\partial \theta}$ with a backward pass of the model.

Lastly, it is worth pointing out that the learned distribution $p(y \mid x, \theta)$ is different from the *empirical* distribution of y from the training set, i.e., given x_i, $y = y_i$ with probability 1. If the learning distribution is indeed replaced with the empirical distribution, the Fisher matrix will become the "empirical" Fisher matrix, which we will now discuss.

Empirical Fisher matrix

As mentioned above, if we replace the learned distribution $P_{x,y}(\theta)$ of (x, y) in (1.7) with the empirical distribution of (x, y), we obtain the expression for the *empirical Fisher* matrix

$$F_{\text{empirical}} := \frac{1}{I} \sum_{i=1}^{I} \frac{\partial \log p(y_i \mid x_i, \theta)}{\partial \theta} \left(\frac{\partial \log p(y_i \mid x_i, \theta)}{\partial \theta} \right)^\top = \frac{1}{I} \sum_{i=1}^{I} \frac{\partial l_i}{\partial \theta} \left(\frac{\partial l_i}{\partial \theta} \right)^\top.$$

Admittedly, the empirical Fisher matrix requires less computation than the Fisher matrix, mainly because we do not need to perform an additional backward pass for computing $\frac{\partial l_i}{\partial \theta}$, as these are already computed in the gradient computation. More importantly, however, it is believed

that the empirical Fisher matrix captures less curvature information compared with the Fisher matrix (see [50]), which is aligned with the fact that Fisher is more closely connected with Hessian than empirical Fisher.

Chapter 2: A First Practical Quasi-Newton Method for Training Multi-layer Perceptrons

2.1 Introduction

We consider in this section the development of practical stochastic quasi-Newton (QN), and in particular Kronecker-factored block-diagonal BFGS [17, 31, 34, 81] and L-BFGS [54], methods for training deep neural networks (DNNs). Recall that the BFGS method starts each iteration with a symmetric positive definite matrix B (or $H = B^{-1}$) that approximates the current Hessian matrix (or its inverse), computes the gradient $\nabla \mathbf{f}$ of f at the current iterate $\mathbf{x}$ and then takes a step $\mathbf{s} = -\alpha H \nabla \mathbf{f}$, where α is a step length (usually) determined by some inexact line-search procedure, such that $\mathbf{y}^\top \mathbf{s} > 0$, where $\mathbf{y} = \nabla \mathbf{f}^+ - \nabla \mathbf{f}$ and $\nabla \mathbf{f}^+$ is the gradient of f at the new point $\mathbf{x}^+ = \mathbf{x} + \mathbf{s}$. The method then computes an updated approximation B^+ to B (or H^+ to H) that remains symmetric and positive-definite and satisfies the so-called *quasi-Newton* (QN) condition $B^+\mathbf{s} = \mathbf{y}$ (or equivalently, $H^+\mathbf{y} = \mathbf{s}$). A consequence of this is that the matrix B^+ operates on the vector $\mathbf{s}$ in exactly the same way as the average of the Hessian matrix along the line segment between $\mathbf{x}$ and $\mathbf{x}^+$ operates on $\mathbf{s}$.

In DNN training, the number of variables and components of the gradient n is often of the order of tens of millions and the Hessian has n^2 elements. Hence, computing and storing a full $n \times n$ BFGS approximation or storing p $(\mathbf{s}, \mathbf{y})$ pairs, where p is approximately 10 or larger for use in an L-BFGS implementation, is out of the question. Consequently, in our methods, we approximate the Hessian by a block-diagonal matrix, where each diagonal block corresponds to a layer, further approximating them as the Kronecker product of two much smaller matrices, as in Martens and Grosse [62], Botev *et al.* [15], Gupta *et al.* [39], and Dangel *et al.* [24].

Literature Review on Using Second-order Information for DNN Training. For solving the stochastic optimization problems with high-dimensional data that arise in machine learning (ML),

stochastic gradient descent (SGD) [75] and its variants are the methods that are most often used, especially for training DNNs. These variants include such methods as AdaGrad [29], RMSprop [42], and Adam [46], all of which scale the stochastic gradient by a diagonal matrix based on estimates of the first and second moments of the individual gradient components. Nonetheless, there has been a lot of effort to find ways to take advantage of second-order information in solving ML optimization problems. Approaches have run the gamut from the use of a diagonal re-scaling of the stochastic gradient, based on the secant condition associated with quasi-Newton (QN) methods [14], to sub-sampled Newton methods (e.g. see Xu *et al.* [91], and references therein), including those that solve the Newton system using the linear conjugate gradient method (see Byrd *et al.* [18]).

In between these two extremes are stochastic methods that are based either on QN methods or generalized Gauss-Newton (GGN) and natural gradient [3] methods. For example, a stochastic L-BFGS method for solving strongly convex problems was proposed in Byrd *et al.* [19] that uses sampled Hessian-vector products rather than gradient differences, which was proved in Moritz *et al.* [66] to be linearly convergent by incorporating the variance reduction technique (SVRG [44]) to alleviate the effect of noisy gradients. A closely related variance reduced block L-BFGS method was proposed in Gower *et al.* [36]. A regularized stochastic BFGS method was proposed in Mokhtari and Ribeiro [64], and an online L-BFGS method was proposed in Mokhtari and Ribeiro [63] for strongly convex problems and extended in Lucchi *et al.* [57] to incorporate SVRG variance reduction. Stochastic BFGS and L-BFGS methods were also developed for online convex optimization in Schraudolph *et al.* [80]. For nonconvex problems, a damped L-BFGS method which incorporated SVRG variance reduction was developed and its convergence properties was studied in Wang *et al.* [87].

GGN methods that approximate the Hessian have been proposed, including the Hessian-free method [59] and the Krylov subspace method [86]. Variants of the closely related natural gradient method that use block-diagonal approximations to the Fisher information matrix, where blocks correspond to layers, have been proposed in e.g. Heskes [41], Desjardins *et al.* [27], Martens and

Grosse [62], and Fujimoto and Ohira [32]. Using further approximation of each of these (empirical) Fisher matrix and GGN blocks by the Kronecker product of two much smaller matrices, the efficient KFAC [62], KFRA [15], EKFAC [33], and Shampoo [39] methods were developed. See also Ba *et al.* [7], Dangel *et al.* [24], and Roux and Fitzgibbon [76], which combine both Hessian and covariance (Fisher-like) matrix information in stochastic Newton type methods, Also, methods are given in Li and Montúfar [53] and Wang and Li [88] that replace the Kullback-Leibler divergence by the Wasserstein distance to define the natural gradient, but with a greater computational cost.

Our Contributions. The main contributions in this section can be summarized as follows:

1. New BFGS and limited-memory variants (i.e. L-BFGS) that take advantage of the structure of feed-forward DNN training problems;

2. Efficient non-diagonal second-order algorithms for deep learning that require a comparable amount of memory and computational cost per iteration as first-order methods;

3. A new damping scheme for BFGS and L-BFGS updating of an inverse Hessian approximation, that not only preserves its positive definiteness, but also limits the decrease (and increase) in its smallest (and largest) eigenvalues for non-convex problems;

4. A novel application of Hessian-action BFGS;

5. The first proof of convergence (to the best of our knowledge) of a stochastic Kronecker-factored quasi-Newton method.

2.2 Kronecker-factored Quasi-Newton Method for DNN

After reviewing the computations used in DNN training, we describe the Kronecker structures of the gradient and Hessian for a single data point, followed by their extension to approximate expectations of these quantities for multiple data-points and give a generic algorithm that employs BFGS (or L-BFGS) approximations for the Hessian.

Deep Neural Networks. We consider a feed-forward DNN with L layers, defined by weight matrices W_l (whose last columns are bias vectors b_l), activation functions ϕ_l for $l \in \{1 \dots L\}$ and

loss function $\mathcal{L}$. For a data-point (x, y), the loss $\mathcal{L}(a_L, y)$ between the output a_L of the DNN and y is a non-convex function of $\theta = \left[\text{vec}(W_1)^\top, \ldots, \text{vec}(W_L)^\top\right]^\top$. The network's forward and backward pass for a single input data point (x, y) is described in Algorithm 1.

Algorithm 1 Forward and backward pass of DNN for a single data-point

1: Given input (x, y), weights (and biases) W_l, and activations ϕ_l for $l \in [1, L]$
2: $\mathbf{a}_0 = x$; **for** $l = 1, .., L$ **do** $\bar{\mathbf{a}}_{l-1} = (\mathbf{a}_{l-1}, 1)$; $\mathbf{h}_l = W_l \bar{\mathbf{a}}_{l-1}$; $\mathbf{a}_l = \phi_l(\mathbf{h}_l)$
3: $\mathcal{D}\mathbf{a}_L \leftarrow \frac{\partial \mathcal{L}(z,y)}{\partial z}\Big|_{z=\mathbf{a}_L}$
4: **for** $l = L, .., 1$ **do** $\mathbf{g}_l = \mathcal{D}\mathbf{a}_l \odot \phi_l'(\mathbf{h}_l)$; $\mathcal{D}W_l = \mathbf{g}_i \bar{\mathbf{a}}_{i-1}^\top$; $\mathcal{D}\mathbf{a}_{l-1} = W_l^\top \mathbf{g}_l$

For a training dataset that contains multiple data-points indexed by $i = 1, ..., I$, let $f(i; \theta)$ denote the loss for the ith data-point. Then, viewing the dataset as an empirical distribution, the total loss function $f(\theta)$ that we wish to minimize is

$$f(\theta) := \mathbb{E}_i[f(i; \theta)] := \frac{1}{I} \sum_{i=1}^{I} f(i; \theta).$$

Single Data-point: Layer-wise Structure of the Gradient and Hessian. Let $\nabla \mathbf{f}_l$ and $\nabla^2 f_l$ denote, respectively, the restriction of $\nabla \mathbf{f}$ and $\nabla^2 f$ to the weights W_l in layer $l = 1, \ldots, L$. For a single data-point $\nabla \mathbf{f}_l$ and $\nabla^2 f_l$ have a tensor (Kronecker) structure, as shown in Martens and Grosse [62] and Botev *et al.* [15]. Specifically,

$$\nabla \mathbf{f}_l(i) = \mathbf{g}_l(i)(\mathbf{a}_{l-1}(i))^\top, \quad \text{equivalently,} \quad \text{vec}(\nabla \mathbf{f}_l(i)) = \mathbf{a}_{l-1}(i) \otimes \mathbf{g}_l(i), \tag{2.1}$$

$$\nabla^2 f_l(i) = (\mathbf{a}_{l-1}(i)(\mathbf{a}_{l-1}(i))^\top) \otimes G_l(i), \tag{2.2}$$

where the pre-activation gradient $\mathbf{g}_l(i) = \frac{\partial f(i)}{\partial \mathbf{h}_l(i)}$, and the pre-activation Hessian $G_l(i) = \frac{\partial^2 f(i)}{\partial \mathbf{h}_l(i)^2}$. Our algorithm uses an approximation to $(G_l(i))^{-1}$, which is updated via the BFGS updating formulas based upon a secant condition that relates the change in $\mathbf{g}_l(i)$ with the change in $\mathbf{h}_l(i)$.

Although we focus on fully-connected layers here, the idea of Kronecker-factored approximations to the diagonal blocks $\nabla^2 f_l$, $l = 1, \ldots, L$ of the Hessian can be extended to other layers used in deep learning, such as convolutional and recurrent layers.

Multiple Data-points: Kronecker-factored QN Approach. Now consider the case where we have a dataset of I data-points indexed by $i = 1, \ldots, I$. By (2.2), we have

$$\mathbb{E}_i[\nabla^2 f_l(i)] \approx \mathbb{E}_i\left[\mathbf{a}_{l-1}(i)(\mathbf{a}_{l-1}(i))^\top\right] \otimes \mathbb{E}_i\left[G_l(i)\right] := A_l \otimes G_l \tag{2.3}$$

Note that the approximation in (2.3) that the expectation of the Kronecker product of two matrices equals the Kronecker product of their expectations is the same as the one used by KFAC [62]. Now, based on this structural approximation, we use $H^l = H_a^l \otimes H_g^l$ as our QN approximation to $\left(\mathbb{E}_i[\nabla^2 f_l(i)]\right)^{-1}$, where H_a^l and H_g^l are positive definite approximations to A_l^{-1} and G_l^{-1}, respectively. Hence, using our layer-wise block-diagonal approximation to the Hessian, a step in our algorithm for each layer l is computed as

$$\text{vec}(W_l^+) - \text{vec}(W_l) = -\alpha H^l \text{vec}\left(\widehat{\nabla \mathbf{f}}_l\right) = -\alpha (H_a^l \otimes H_g^l)\text{vec}\left(\widehat{\nabla \mathbf{f}}_l\right) = -\alpha \text{vec}\left(H_g^l \widehat{\nabla \mathbf{f}}_l H_a^l\right), \tag{2.4}$$

where $\widehat{\nabla \mathbf{f}}_l$ denotes the estimate to $\mathbb{E}_i[\nabla \mathbf{f}_l(i)]$ and α is the learning rate. After computing W_l^+ and performing another forward/backward pass, our method computes or updates H_a^l and H_g^l as follows:

1. For H_g^l, we use a damped version of BFGS (or L-BFGS) (See Section 2.3) based on the $(\mathbf{s}, \mathbf{y})$ pairs corresponding to the average change in $\mathbf{h}_l(i)$ and in the gradient with respect to $\mathbf{h}_l(i)$; i.e.,

$$\mathbf{s}_g^l = \mathbb{E}_i[\mathbf{h}_l^+(i)] - \mathbb{E}_i[\mathbf{h}_l(i)], \qquad \mathbf{y}_g^l = \mathbb{E}_i[\mathbf{g}_l^+(i)] - \mathbb{E}_i[\mathbf{g}_l(i)]. \tag{2.5}$$

2. For H_a^l we use the "Hessian-action" BFGS method described in Section 2.4. The issue of possible singularity of the positive semi-definite matrix A_l approximated by $(H_a^l)^{-1}$ is also addressed there by incorporating a **Levenberg-Marquardt (LM)** damping term.

Algorithm 2 gives a high-level summary of our **K-BFGS** or **K-BFGS(L)** algorithms (which use BFGS or L-BFGS to update H_g^l, respectively). In order to distinguish the K-BFGS and K-

BFGS(L) methods here from those that will be described in Chapter 3, we refer to the methods in Chapter 2 as K-BFGS-20 and K-BFGS(L)-20 outside Chapter 2. See Algoirthm 4 for a detailed pseudocode. The use of mini-batches is described in Section 2.8. Note that, an additional forward-backward pass is used in Algorithm 2 because the quantities in (2.5) need to be estimated using the same mini-batch.

Algorithm 2 High-level summary of K-BFGS / K-BFGS(L)

Require: Given initial weights θ, batch size m, learning rate α

1: **for** $k = 1, 2, \ldots$ **do**
2: Sample mini-batch of size m: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$
3: Perform a forward-backward pass over the current mini-batch M_k (see Algorithm 1)
4: **for** $l = 1, \ldots, L$ **do** $p_l = H_g^l \widehat{\nabla \mathbf{f}}_l H_a^l$; $W_l = W_l - \alpha \cdot p_l$
5: Perform another forward-backward pass over M_k to get $(\mathbf{s}_g^l, \mathbf{y}_g^l)$
6: Use damped BFGS or L-BFGS to update H_g^l ($l = 1, ..., L$) (see Section 2.3, in particular Algorithm 3)
7: Use Hessian-action BFGS to update H_a^l ($l = 1, ..., L$) (see Section 2.4)
8: **end for**

2.3 BFGS and L-BFGS for G_l

Damped BFGS Updating. It is well-known that training a DNN is a non-convex optimization problem. As (2.2) and (2.3) show, this non-convexity manifests in the fact that $G_l > 0$ often does not hold. Thus, for the BFGS update of H_g^l, the approximation to G_l^{-1}, to remain positive definite, we have to ensure that $(\mathbf{s}_g^l)^\top \mathbf{y}_g^l > 0$. Due to the stochastic setting, ensuring this condition by line-search, as is done in deterministic settings, is impractical. In addition, due to the large changes in curvature in DNN models that occur as the parameters are varied, we also need to suppress large changes to H_g^l as it is updated. To deal with both of these issues, we propose a **double damping (DD)** procedure (Algorithm 3), which is based upon **Powell's damped-BFGS** approach [69], for modifying the $(\mathbf{s}_g^l, \mathbf{y}_g^l)$ pair. To motivate Algorithm 3, consider the formulas used for BFGS updating of B and H:

$$B^+ = B - \frac{B\mathbf{s}\mathbf{s}^\top B}{\mathbf{s}^\top B \mathbf{s}} + \rho \mathbf{y}\mathbf{y}^\top, \quad H^+ = (I - \rho \mathbf{s}\mathbf{y}^\top) H (I - \rho \mathbf{y}\mathbf{s}^\top) + \rho \mathbf{s}\mathbf{s}^\top, \tag{2.6}$$

where $\rho = \frac{1}{\mathbf{s}^\top \mathbf{y}} > 0$. If we can ensure that $0 < \frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_1}$ and $0 < \frac{\mathbf{s}^\top \mathbf{s}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_2}$, then we can obtain the following bounds:

$$\|B^+\| \leq \|B - \frac{B\mathbf{s}\mathbf{s}^\top B}{\mathbf{s}^\top B \mathbf{s}}\| + \|\rho \mathbf{y}\mathbf{y}^\top\| \leq \|B\| + \|\frac{B^{1/2}H^{1/2}\mathbf{y}\mathbf{y}^\top H^{1/2}B^{1/2}}{\mathbf{s}^\top \mathbf{y}}\| \tag{2.7}$$

$$\leq \|B\| + \|B\|\frac{\|H^{1/2}\mathbf{y}\|^2}{\mathbf{s}^\top \mathbf{y}} \leq \|B\|\left(1 + \frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}}\right) \leq \|B\|\left(1 + \frac{1}{\mu_1}\right) \tag{2.8}$$

and

$$\|H^+\| \leq \|H^{1/2} - \frac{\mathbf{s}\mathbf{y}^\top H^{1/2}}{\mathbf{s}^\top \mathbf{y}}\|^2 + \|\frac{\mathbf{s}\mathbf{s}^\top}{\mathbf{s}^\top \mathbf{y}}\| \leq \left(\|H^{1/2}\| + \frac{\|\mathbf{s}\|\|H^{1/2}\mathbf{y}\|}{\mathbf{s}^\top \mathbf{y}}\right)^2 + \frac{\|\mathbf{s}\|^2}{\mathbf{s}^\top \mathbf{y}} \tag{2.9}$$

$$\leq \left(\|H^{1/2}\| + (\frac{\mathbf{s}^\top \mathbf{s}}{\mathbf{s}^\top \mathbf{y}})^{1/2}(\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}})^{1/2}\right)^2 + \frac{\mathbf{s}^\top \mathbf{s}}{\mathbf{s}^\top \mathbf{y}} \leq \left(\|H^{1/2}\| + \frac{1}{\sqrt{\mu_1 \mu_2}}\right)^2 + \frac{1}{\mu_2}. \tag{2.10}$$

Thus, the change in B (and H) is controlled if $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_1}$ and $\frac{\mathbf{s}^\top \mathbf{s}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_2}$. Our DD approach is a two-step procedure, where the first step (i.e. Powell's damping of H) guarantees that $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_1}$ and the second step (i.e., Powell's damping with $B = I$) guarantees that $\frac{\mathbf{s}^\top \mathbf{s}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_2}$. Note that there is no guarantee of $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{1}{\mu_1}$ after the second step. However, we can skip updating H in this case so that the bounds on these matrices hold. In our implementation, we always do the update, since in empirical testing, we observed that at least 90% of the pairs satisfy $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{2}{\mu_1}$. See Section 2.3.1 for more details on damping.

Algorithm 3 Double Damping (DD)

1: **Input:** $\mathbf{s}$, $\mathbf{y}$; **Output:** $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$; **Given:** H, μ_1, μ_2
2: **if** $\mathbf{s}^\top \mathbf{y} < \mu_1 \mathbf{y}^\top H \mathbf{y}$ **then** $\theta_1 = \frac{(1-\mu_1)\mathbf{y}^\top H \mathbf{y}}{\mathbf{y}^\top H \mathbf{y} - \mathbf{s}^\top \mathbf{y}}$ **else** $\theta_1 = 1$
3: $\tilde{\mathbf{s}} = \theta_1 \mathbf{s} + (1 - \theta_1) H \mathbf{y}$ {Powell's damping on H}
4: **if** $\tilde{\mathbf{s}}^\top \mathbf{y} < \mu_2 \tilde{\mathbf{s}}^\top \tilde{\mathbf{s}}$ **then** $\theta_2 = \frac{(1-\mu_2)\tilde{\mathbf{s}}^\top \tilde{\mathbf{s}}}{\tilde{\mathbf{s}}^\top \tilde{\mathbf{s}} - \tilde{\mathbf{s}}^\top \mathbf{y}}$ **else** $\theta_2 = 1$
5: $\tilde{\mathbf{y}} = \theta_2 \mathbf{y} + (1 - \theta_2)\tilde{\mathbf{s}}$ {Powell's damping with $B = I$}
6: **return** $\tilde{s}$, $\tilde{y}$

L-BFGS Implementation. L-BFGS can also be used to update H_g^l. However, implementing L-BFGS using the standard "two-loop recursion" (see Algorithm 7.4 in Nocedal and Wright [67]) is not efficient. This is because the main work in computing $H_g^l \widehat{\nabla \mathbf{f}}_l H_a^l$ in line 4 of Algorithm 2 would

require $4p$ matrix-vector multiplications, each requiring $O(d_i d_o)$ operations, where p denotes the number of $(\mathbf{s}, \mathbf{y})$ pairs stored by L-BFGS. (Recall that $\widehat{\nabla \mathbf{f}}_l \in R^{d_o \times d_i}$.) Instead, we use a "non-loop" implementation [20] of L-BFGS, whose main work involves 2 matrix-matrix multiplications, each requiring $O(p d_i d_o)$ operations. When p is not small (we used $p = 100$ in our tests), and d_i and d_o are large, this is much more efficient, especially on GPUs.

2.3.1 Powell's Damped BFGS Updating

For BFGS and L-BFGS, one needs $\mathbf{y}^\top \mathbf{s} > 0$. However, when used to update H_g^l, there is no guarantee that $(\mathbf{y}_g^l)^\top \mathbf{s}_g^l > 0$ for any layer $l = 1, \ldots, L$. In deterministic optimization, positive definiteness of the QN Hessian approximation B (or its inverse) is maintained by performing an inexact line search that ensures that $\mathbf{s}^T \mathbf{y} > 0$, which is always possible as long as the function being minimized is bounded below. However, this would be expensive to do for DNN. Thus, we propose the following heuristic based on Powell's damped-BFGS approach [69].

Powell's Damping on B. Powell's damping on B, proposed in Powell [69], replaces $\mathbf{y}$ in the BFGS update, by $\tilde{\mathbf{y}} = \theta \mathbf{y} + (1 - \theta) B\mathbf{s}$, where

$$\theta = \begin{cases} \frac{(1-\mu)\mathbf{s}^\top B\mathbf{s}}{\mathbf{s}^\top B\mathbf{s} - \mathbf{s}^\top \mathbf{y}}, & \text{if } \mathbf{s}^\top \mathbf{y} < \mu \mathbf{s}^\top B\mathbf{s}, \\ 1, & \text{otherwise.} \end{cases}$$

It is easy to verify that $\mathbf{s}^\top \tilde{\mathbf{y}} \geq \mu \mathbf{s}^\top B\mathbf{s}$.

Powell's Damping on H. In Powell's damping on H (see e.g. Badreddine *et al.* [8]), $\tilde{\mathbf{s}} = \theta \mathbf{s} + (1 - \theta) H\mathbf{y}$ replaces $\mathbf{s}$, where

$$\theta = \begin{cases} \frac{(1-\mu_1)\mathbf{y}^\top H\mathbf{y}}{\mathbf{y}^\top H\mathbf{y} - \mathbf{s}^\top \mathbf{y}}, & \text{if } \mathbf{s}^\top \mathbf{y} < \mu_1 \mathbf{y}^\top H\mathbf{y}, \\ 1, & \text{otherwise.} \end{cases}$$

This is used in lines 2 and 3 of the DD (Algorithm 3). It is also easy to verify that $\tilde{\mathbf{s}}^\top \mathbf{y} \geq \mu_1 \mathbf{y}^\top H\mathbf{y}$.

Powell's Damping with $B = I$. Powell's damping on B is not suitable for our algorithms

because we do not keep track of B. Moreover, it does not provide a simple bound on $\frac{\tilde{\mathbf{s}}^\top \tilde{\mathbf{s}}}{\tilde{\mathbf{s}}^\top \tilde{\mathbf{y}}}$ that is independent of $\|B\|$. Therefore, we use Powell's damping with $B = I$, in lines 4 and 5 of the DD (Algorithm 3). It is easy to verify that it ensures that $\tilde{\mathbf{s}}^\top \tilde{\mathbf{y}} \geq \mu_2 \tilde{\mathbf{s}}^\top \tilde{\mathbf{s}}$.

Powell's damping with $B = I$ can be interpreted as adding an Levenberg-Marquardt (LM) damping term to B. Note that an LM damping term μ_2 would lead to $B \geq \mu_2 I$. Then, the secant condition $\tilde{\mathbf{y}} = B\tilde{\mathbf{s}}$ implies

$$\tilde{\mathbf{y}}^\top \tilde{\mathbf{s}} = \tilde{\mathbf{s}}^\top B \tilde{\mathbf{s}} \geq \mu_2 \tilde{\mathbf{s}}^\top \tilde{\mathbf{s}},$$

which is the same inequality as we get using Powell's damping with $B = I$. Note that although the μ_2 parameter in Powell's damping with $B = I$ can be interpreted as an LM damping, we recommend setting the value of μ_2 within $(0, 1]$ so that $\tilde{\mathbf{y}}$ is a convex combination of $\mathbf{y}$ and $\tilde{\mathbf{s}}$. In all of our experimental tests, we found that the best value for the hyperparameter λ for both K-BFGS and K-BFGS(L) was less than or equal to 1, and hence that $\mu_2 = \lambda_G = \sqrt{\lambda}$ was in the interval $(0, 1]$.

Double Damping (DD)

Our double damping (Algorithm 3) is a two-step damping procedure, where the first step (i.e. Powell's damping on H) can be viewed as an interpolation between the current curvature and the previous ones, and the second step (i.e. Powell's damping with $B = I$) can be viewed as an LM damping.

Recall that there is no guarantee that $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{2}{\mu_1}$ holds after DD. While we skip using pairs that do not satisfy this inequality, when updating H_g^l in proving the convergence of the K-BFGS(L) variant Algorithm 5 , we use all $(\mathbf{s}, \mathbf{y})$ pairs to update H_g^l in our implementations of both K-BFGS and K-BFGS(L) . However, whether one skips or not makes only slight difference in the performance of these algorithms, because as our empirical testing has shown, at least 90% of the iterations satisfy $\frac{\mathbf{y}^\top H \mathbf{y}}{\mathbf{s}^\top \mathbf{y}} \leq \frac{2}{\mu_1}$, even if we don't skip. See Figure 2.1 which reports results on this for K-BFGS(L) when tested on the MNIST, FACES and CURVES datasets, where the legends in each plot assign different colors to represent each layer l.

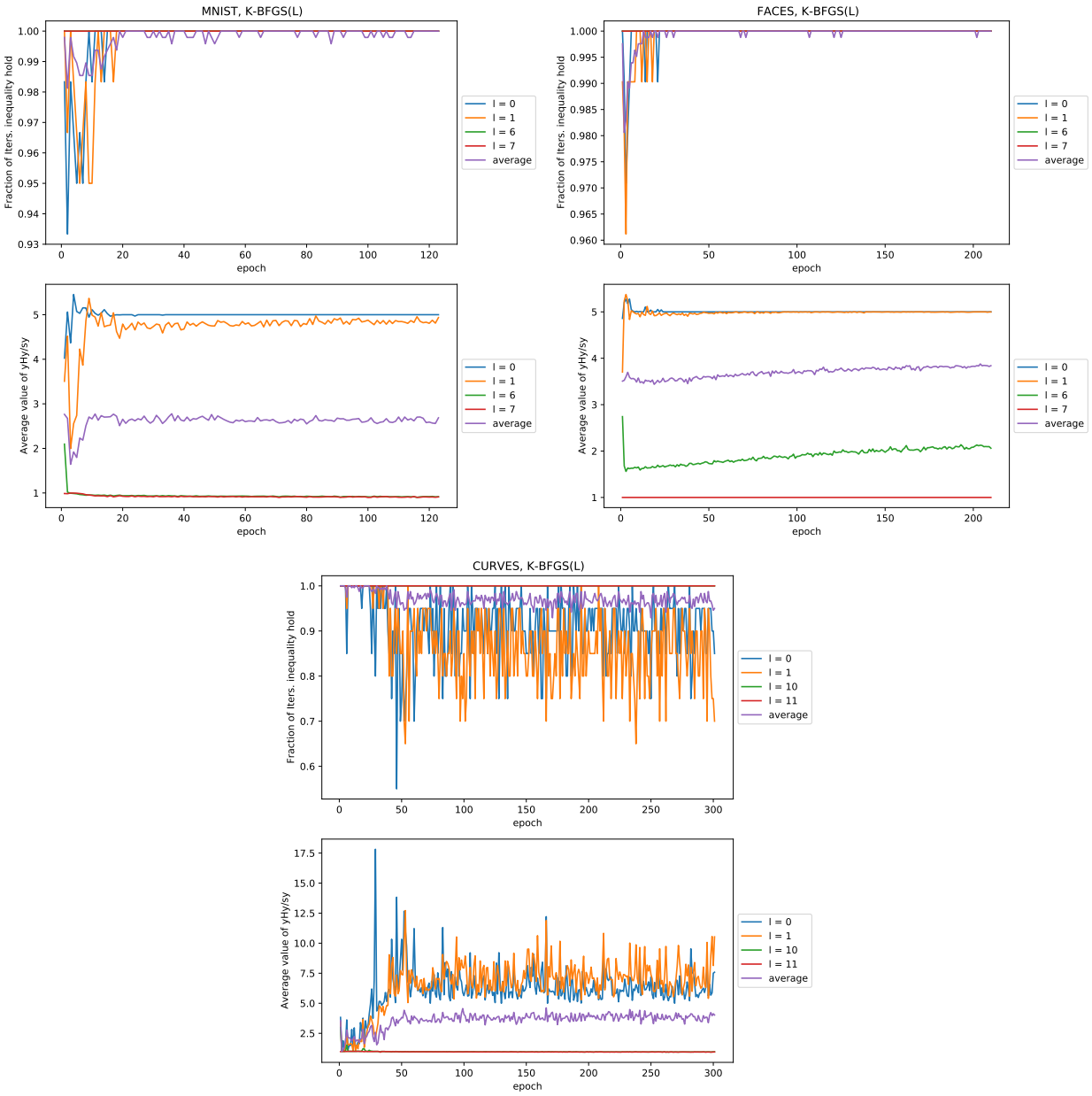


Figure 2.1: Fraction of the number of iterations in each epoch, in which the inequality $\frac{\mathbf{y}^\top H\mathbf{y}}{\mathbf{s}^\top\mathbf{y}} \leq \frac{2}{\mu_1}$ holds (upper plots), and the average value of $\frac{\mathbf{y}^T H\mathbf{y}}{\mathbf{s}^\top\mathbf{y}}$ (lower plots) in each epoch

2.4 "Hessian action" BFGS for A_l

In addition to approximating G_l^{-1} by H_g^l using BFGS, we also propose approximating A_l^{-1} by H_a^l using BFGS. Note that A_l does not correspond to some Hessian of the objective function. However, we can generate $(\mathbf{s}, \mathbf{y})$ pairs for it by "Hessian action" (see e.g. Byrd *et al.* [19], Gower *et al.* [36], and Gower and Richtárik [37]).

Connection between Hessian-action BFGS and Matrix Inversion. In our methods, we choose $\mathbf{s} = H_a^l \cdot \mathbb{E}_i[\mathbf{a}_{l-1}(i)]$ and $\mathbf{y} = A_l\mathbf{s}$, which as we now show, is closely connected to using the Sherman-Morrison modification formula to invert A_l. In particular, suppose that $A^+ = A + c{\cdot}\mathbf{a}\mathbf{a}^\top$; i.e., only a rank-one update is made to A. This corresponds to the case where the information of A is accumulated from iteration to iteration, and the size of the mini-batch is 1 or $\mathbf{a}$ represents the average of the vectors $\mathbf{a}(i)$ from multiple data-points.

Theorem 1. *Suppose that A and H are symmetric and positive definite, and that $H = A^{-1}$. If we choose $\mathbf{s} = H\mathbf{a}$ and $\mathbf{y} = A^+\mathbf{s}$, where $A^+ = A + c \cdot \mathbf{a}\mathbf{a}^\top (c > 0)$. Then, the H^+ generated by any QN update in the **Broyden family***

$$H^+ = H - \sigma H\mathbf{y}\mathbf{y}^\top H + \rho\mathbf{s}\mathbf{s}^\top + \phi(\mathbf{y}^\top H\mathbf{y})\mathbf{h}\mathbf{h}^\top, \tag{2.11}$$

where $\rho = 1/\mathbf{s}^\top\mathbf{y}$, $\sigma = 1/\mathbf{y}^\top H\mathbf{y}$, $\mathbf{h} = \rho\mathbf{s} - \sigma H\mathbf{y}$ and ϕ is a scalar parameter in $[0, 1]$, equals $(A^+)^{-1}$. Note that $\phi = 1$ yields the BFGS update (2.6) and $\phi = 0$ yields the DFP update.

Proof. If $\mathbf{s} = H\mathbf{a}$ and $\mathbf{y} = A^+\mathbf{s}$, then $\mathbf{h} = 0$, so all choices of ϕ yield the same matrix H^+. Since $H^+A^+\mathbf{s} = H^+\mathbf{y} = \mathbf{s}$ and for any vector $\mathbf{v}$ that is orthogonal to $\mathbf{a}$, $H^+A^+\mathbf{v} = H^+A\mathbf{v} = \mathbf{v}$, since $\mathbf{s}^\top A \rightleftharpoons 0$ and $\mathbf{y}^\top HA \rightleftharpoons 0$, it follows that $H^+A^+ = I$, using the fact that $\mathbf{s}$ together with any linearly independent set of $n - 1$ vectors orthogonal to $\mathbf{a}$ spans R^n. (Note that $\mathbf{s}^\top\mathbf{a} = \mathbf{a}^\top H\mathbf{a} > 0$, since $H > 0 \Rightarrow$ that $\mathbf{s}$ is not orthogonal to $\mathbf{a}$.) □

In fact, all updates in the Broyden family are equivalent to applying the Sherman-Morrison modification formula to $A^+ = A + c{\cdot}\mathbf{a}\mathbf{a}^\top$, given $H = A^{-1}$, since after substituting for $\mathbf{s}$ and $\mathbf{y}$ in

(2.11) and simplifying, one obtains

$$H^+ = H - H\mathbf{a}(c^{-1} + \mathbf{a}^\top H\mathbf{a})^{-1}\mathbf{a}^\top H.$$

When using momentum, $A^+ = \beta A + (1-\beta)\mathbf{a}\mathbf{a}^\top$ $(0 < \beta < 1)$. Hence, if we still want Theorem 1 to hold, we have to scale H by $1/\beta$ before updating it. This, however, turns out to be unstable. Hence, in practice, we use the non-scaled version of "Hessian action" BFGS.

Levenberg-Marquardt Damping for A_l. Since $A_l = \mathbb{E}_i\left[(\mathbf{a}_{l-1}(i)(\mathbf{a}_{l-1}(i))^\top)\right] \succeq 0$ may not be positive definite, or may have very small positive eigenvalues, we add an **Levenberg-Marquardt (LM) damping** term to make our "Hessian-action" BFGS stable; i.e., we use $A_l + \lambda_A I_A$ instead of A_l, when we update H_a^l. Specifically, "Hessian action" BFGS for A_l is performed as

1. $A_l = \beta \cdot A_l + (1-\beta) \cdot \mathbb{E}_i\left[\mathbf{a}_{l-1}(i)\mathbf{a}_{l-1}(i)^\top\right]$; $A_l^{\mathrm{LM}} = A_l + \lambda_A I_A$.
2. $\mathbf{s}_a^l = H_a^l \cdot \mathbb{E}_i[\mathbf{a}_{l-1}(i)]$, $\mathbf{y}_a^l = A_l^{\mathrm{LM}}\mathbf{s}_a^l$; use BFGS with $(\mathbf{s}_a^l, \mathbf{y}_a^l)$ to update H_a^l.

2.5 Pseudo-code and Algorithm Details for K-BFGS/K-BFGS(L)

2.5.1 Pseudo-code for K-BFGS/K-BFGS(L)

Algorithm 4 gives pseudocode for K-BFGS/K-BFGS(L), which is implemented in the experiments. For details see Sections 2.3, 2.4, and Section 2.3.1.

2.5.2 Algorithm Details: Mini-batch and Moving Average

Clearly, using the whole dataset at each iteration is inefficient; hence, we use a mini-batch to estimate desired quantities. We use $\overline{X}$ to denote the averaged value of X across the mini-batch for any quantity X. To incorporate information from the past as well as reducing the variability, we use an exponentially decaying moving average to estimate desired quantities with decay parameter $\beta \in (0,1)$:

1. To estimate the gradient $\mathbb{E}_i[\nabla\mathbf{f}(i)]$, at each iteration, we update $\widehat{\nabla\mathbf{f}} = \beta \cdot \widehat{\nabla\mathbf{f}} + (1-\beta) \cdot \overline{\nabla\mathbf{f}}$.
2. H_a^l: To estimate A_l, at each iteration we update $\widehat{A}_l = \beta \cdot \widehat{A}_l + (1-\beta) \cdot \overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}$. Note that

Algorithm 4 Pseudocode for K-BFGS / K-BFGS(L)

Require: Given initial weights $\theta = \left[\text{vec}\,(W_1)^\top, \ldots, \text{vec}\,(W_L)^\top\right]^\top$, batch size m, learning rate α, damping value λ, and for K-BFGS(L), the number of $(\mathbf{s}, \mathbf{y})$ pairs p that are stored and used to compute H_g^l at each iteration

$\mu_1 = 0.2, \beta = 0.9$ {set default hyper-parameter values}

$\lambda_A = \lambda_G = \sqrt{\lambda}$ {split the damping into A and G}

$\widehat{\overline{\nabla \mathbf{f}}}_l = 0$, $A_l = \mathbb{E}_i\left[\mathbf{a}_{l-1}(i)\mathbf{a}_{l-1}(i)^\top\right]$ by forward pass, $H_a^l = (A_l + \lambda_A I_A)^{-1}$, $H_g^l = I$ $(l = 1, ..., L)$ {Initialization}

for $k = 1, 2, \ldots$ **do**

  Sample mini-batch of size m: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$

  Perform a forward-backward pass over the current mini-batch M_k to compute $\overline{\nabla \mathbf{f}}_l$, $\mathbf{a}_l$, $\mathbf{h}_l$, and $\mathbf{g}_l$ $(l = 1, \ldots, L)$ (see Algorithm 1)

  for $l = 1, \ldots, L$ **do**

    $\widehat{\overline{\nabla \mathbf{f}}}_l = \beta \widehat{\overline{\nabla \mathbf{f}}}_l + (1 - \beta)\overline{\nabla \mathbf{f}}_l$

    $p_l = H_g^l \widehat{\overline{\nabla \mathbf{f}}}_l H_a^l$

    {In K-BFGS(L), when computing $H_g^l \left(\widehat{\overline{\nabla \mathbf{f}}}_l H_a^l\right)$, L-BFGS is initialized with an identity matrix}

    $W_l = W_l - \alpha \cdot p_l$

  end for

  Perform another forward-backward pass over M_k to compute $\mathbf{h}_l^+$, $\mathbf{g}_l^+$ $(l = 1, \ldots, L)$

  for $l = 1, ..., L$ **do**

    {Use damped BFGS or L-BFGS to update H_g^l (see Section 2.3)}

    $\mathbf{s}_g^l = \beta \cdot \mathbf{s}_g^l + (1 - \beta) \cdot \left(\overline{\mathbf{h}_l^+} - \overline{\mathbf{h}_l}\right)$, $\mathbf{y}_g^l = \beta \cdot \mathbf{y}_g^l + (1 - \beta) \cdot \left(\overline{\mathbf{g}_l^+} - \overline{\mathbf{g}_l}\right)$

    $(\tilde{\mathbf{s}}_g^l, \tilde{\mathbf{y}}_g^l) = \text{DD}(\mathbf{s}_g^l, \mathbf{y}_g^l)$ with $H = H_g^l$, $\mu_1 = \mu_1$, $\mu_2 = \lambda_G$ {See Algorithm 3}

    Use BFGS or L-BFGS with $(\tilde{\mathbf{s}}_g^l, \tilde{\mathbf{y}}_g^l)$ to update H_g^l

    {Use Hessian-action BFGS to update H_a^l (see Section 2.4)}

    $A_l = \beta \cdot A_l + (1 - \beta) \cdot \overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}$

    $A_l^{\text{LM}} = A_l + \lambda_A I_A$

    $\mathbf{s}_a^l = H_a^l \cdot \overline{\mathbf{a}_{l-1}}$, $\mathbf{y}_a^l = A_l^{\text{LM}} \mathbf{s}_a^l$

    Use BFGS with $(\mathbf{s}_a^l, \mathbf{y}_a^l)$ to update H_a^l

  end for

end for

although we compute $\mathbf{s}_a^l$ as $H_a^l \cdot \overline{\mathbf{a}_{l-1}}$, we update $\widehat{A}_l$ with $\overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}$ (i.e. the average $\mathbf{a}_{l-1}(i)\mathbf{a}_{l-1}(i)^\top$ over the minibatch, not $\overline{\mathbf{a}_{l-1}} \cdot (\overline{\mathbf{a}_{l-1}})^\top$).

3. H_g^l: BFGS "uses" momentum implicitly incorporated in the matrices H_g^l. To further stabilize the BFGS update, we also use a moving-averaged $(\mathbf{s}_g^l, \mathbf{y}_g^l)$ (before damping); i.e., We update $\mathbf{s}_g^l = \beta \cdot \mathbf{s}_g^l + (1-\beta) \cdot \left(\overline{\mathbf{h}_l^+} - \overline{\mathbf{h}_l}\right)$, and $\mathbf{y}_g^l = \beta \cdot \mathbf{y}_g^l + (1-\beta) \cdot \left(\overline{\mathbf{g}_l^+} - \overline{\mathbf{g}_l}\right)$.

Finally, when computing $\overline{\mathbf{h}_l^+}$ and $\overline{\mathbf{g}_l^+}$, we use the same mini-batch as was used to compute $\overline{\mathbf{h}_l}$ and $\overline{\mathbf{g}_l}$. This doubles the number of forward-backward passes at each iteration.

2.6 Storage and Computational Complexity

Tables 2.1 and 2.2 compare the storage and computational requirements, respectively, for a layer with d_i inputs and d_o outputs for K-BFGS, K-BFGS(L), KFAC, and Adam/RMSprop. We denote the size of mini-batch by m, the number of $(\mathbf{s}, \mathbf{y})$ pairs stored for L-BFGS by p, and the frequency of matrix inversion in KFAC by T. Besides the requirements listed in Table 2.1, all algorithms need storage for the parameters W_l and the estimate of the gradient, $\widehat{\nabla \mathbf{f}_l}$, (i.e. $O(d_i d_o)$). Besides the work listed in Table 2.2, all algorithms also need to do a forward-backward pass to compute $\nabla \mathbf{f}_l$ as well as updating W_l, (i.e. $O(m d_i d_o)$). Also note that, even though we use big-O notation in these tables, the constants for all of the terms in each of the rows are roughly at the same level and relatively small.

In Table 2.2, for K-BFGS and K-BFGS(L), "Additional pass" refers to Line 5 of Algorithm 2; under "Curvature", $O(m d_i^2)$ arises from "Hessian action" BFGS to update H_a^l (see the algorithm at the end of Section 2.4), $O(m d_o)$ arises from (2.5), $O(d_o^2)$ arises from updating H_g^l (only for K-BFGS); and "Step ΔW_l" refers to (2.4). For KFAC, referring to Algorithm 7, "Additional pass" refers to Line 8; under "Curvature", $O(m d_i^2 + m d_o^2)$ refers to Line 9, and $O(\frac{1}{T} d_i^3 + \frac{1}{T} d_o^3)$ refers to Line 11; and "Step ΔW_l" refers to Line 5.

From Table 2.1, we see that the Kronecker property enables K-BFGS and K-BFGS(L) (as well as KFAC) to have storage requirements comparable to those of first-order methods. Moreover, from Table 2.2, we see that K-BFGS and K-BFGS(L) require less computation per iteration than KFAC,

Table 2.1: Storage Requirement of K-BFGS(-20), K-BFGS(L)(-20), KFAC, and Adam/RMSprop beyond that required for SGD

Algorithm	$\nabla f_l \odot \nabla f_l$	A	G	Total
K-BFGS	—	$O(d_i^2)$	$O(d_o^2)$	$O(d_i^2 + d_o^2 + d_i d_o)$
K-BFGS(L)	—	$O(d_i^2)$	$O(pd_o)$	$O(d_i^2 + d_i d_o + pd_o)$
KFAC	—	$O(d_i^2)$	$O(d_o^2)$	$O(d_i^2 + d_o^2 + d_i d_o)$
Adam/RMSprop	$O(d_i d_o)$	—	—	$O(d_i d_o)$

Table 2.2: Computation per iteration of K-BFGS(-20), K-BFGS(L)(-20), KFAC, and Adam/RMSprop beyond that required for SGD

Algorithm	Additional pass	Curvature	Step ΔW_l
K-BFGS	$O(md_i d_o)$	$O(md_i^2 + md_o + d_o^2)$	$O(d_i^2 d_o + d_o^2 d_i)$
K-BFGS(L)	$O(md_i d_o)$	$O(md_i^2 + md_o)$	$O(d_i^2 d_o + pd_i d_o)$
KFAC	$O(md_i d_o)$	$O(md_i^2 + md_o^2 + \frac{1}{T}d_i^3 + \frac{1}{T}d_o^3)$	$O(d_i^2 d_o + d_o^2 d_i)$
Adam/RMSprop	—	$O(d_i d_o)$	$O(d_i d_o)$

since they only involve matrix multiplications, whereas KFAC requires matrix inversions which depend cubically on both d_i and d_o. The cost of matrix inversion in KFAC (and singular value decomposition in Gupta *et al.* [39]) is amortized by performing these operations only once every T iterations; nonetheless, these amortized operations usually become much slower than matrix multiplication as models scale up.

2.7 Convergence Analysis

Following the framework for stochastic quasi-Newton methods (SQN) established in [87] for solving nonconvex stochastic optimization problems, we prove that, under fairly standard assumptions, for our K-BFGS(L) algorithm with skipping DD and exact inversion on A_l (see Algorithm 5), the number of iterations N needed to obtain $\frac{1}{N}\sum_{k=1}^{N}\mathbb{E}[\|\nabla \mathbf{f}(\theta_k)\|^2] \leq \epsilon$ is $N = O(\epsilon^{-\frac{1}{1-\beta}})$, for step size α_k chosen proportional to $k^{-\beta}$, where $\beta \in (0.5, 1)$ is a constant.

Algorithm 5 is very similar to our actual implementation of K-BFGS(L) (i.e. Algorithm 4), except that

- we skip updating H_g^l if $(\tilde{\mathbf{s}}_g^l)^\top \tilde{\mathbf{y}}_g^l < \mu_1 (\tilde{\mathbf{y}}_g^l)^\top H_g^l \tilde{\mathbf{y}}_g^l$ (see Line 17);
- we set H_a^l to the exact inverse of A_l^{LM} (see Line 23);

Algorithm 5 K-BFGS(L) with DD-skip and exact inversion of A_l^{LM}

Require: Given initial weights $\theta = \left[\mathrm{vec}\,(W_1)^\top, \ldots, \mathrm{vec}\,(W_L)^\top\right]^\top$, batch size m, learning rate $\{\alpha_k\}$, damping value λ, and the number of $(\mathbf{s}, \mathbf{y})$ pairs p that are stored and used to compute H_g^l at each iteration

$\mu_1 = 0.2, \beta = 0.9$ {set default hyper-parameter values}
$\lambda_A = \lambda_G = \sqrt{\lambda}$ {split the damping into A and G}
$A_l(0) = \mathbb{E}_i\left[\mathbf{a}_{l-1}(i)\mathbf{a}_{l-1}(i)^\top\right]$ by forward pass, $H_a^l(0) = (A_l(0) + \lambda_A I_A)^{-1}$, $H_g^l(0) = I$ $(l = 1, ..., L)$ {Initialization}
for $k = 1, 2, \ldots$ **do**
  Sample mini-batch of size m: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$
  Perform a forward-backward pass over the current mini-batch M_k to compute $\overline{\nabla \mathbf{f}}_l$, $\mathbf{a}_l$, $\mathbf{h}_l$, and $\mathbf{g}_l$ $(l = 1, \ldots, L)$ (see Algorithm 1)
  for $l = 1, \ldots, L$ **do**
    $p_l = H_g^l(k-1)\widehat{\nabla \mathbf{f}}_l H_a^l(k-1)$, where $\widehat{\nabla \mathbf{f}}_l = \overline{\nabla \mathbf{f}}_l$
    {When computing $H_g^l\left(\widehat{\nabla \mathbf{f}}_l H_a^l\right)$, L-BFGS is initialized with an identity matrix}
    $W_l = W_l - \alpha_k \cdot p_l$
  end for
  Perform another forward-backward pass over M_k to compute $\mathbf{h}_l^+$, $\mathbf{g}_l^+$ $(l = 1, \ldots, L)$
  for $l = 1, ..., L$ **do**
    {Use damped L-BFGS with skip to update H_g^l (see Section 2.3)}
    $\mathbf{s}_g^l = \beta \cdot \mathbf{s}_g^l + (1-\beta) \cdot \left(\overline{\mathbf{h}_l^+} - \overline{\mathbf{h}_l}\right)$, $\mathbf{y}_g^l = \beta \cdot \mathbf{y}_g^l + (1-\beta) \cdot \left(\overline{\mathbf{g}_l^+} - \overline{\mathbf{g}_l}\right)$
    $(\tilde{\mathbf{s}}_g^l, \tilde{\mathbf{y}}_g^l) = \mathrm{DD}(\mathbf{s}_g^l, \mathbf{y}_g^l)$ with $H = H_g^l(k-1)$, $\mu_1 = \mu_1$, $\mu_2 = \lambda_G$ {See Algorithm 3}
    if $(\tilde{\mathbf{s}}_g^l)^\top \tilde{\mathbf{y}}_g^l \geq \mu_1 (\tilde{\mathbf{y}}_g^l)^\top H_g^l \tilde{\mathbf{y}}_g^l$ **then**
      Use L-BFGS with $(\tilde{\mathbf{s}}_g^l, \tilde{\mathbf{y}}_g^l)$ to update $H_g^l(k)$
    end if
    {Use exact inversion to compute H_a^l}
    $A_l(k) = \beta \cdot A_l(k-1) + (1-\beta) \cdot \overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}$
    $A_l^{\mathrm{LM}}(k) = A_l(k) + \lambda_A I_A$
    $H_a^l(k) = \left(A_l^{\mathrm{LM}}(k)\right)^{-1}$
  end for
end for

Algorithm 6 SQN method for nonconvex stochastic optimization.

Require: Given $\theta_1 \in \mathbb{R}^n$, batch sizes $\{m_k\}_{k\geq 1}$, and step sizes $\{\alpha_k\}_{k\geq 1}$

for $k = 1, 2, \ldots$ **do**
  Calculate $\widehat{\nabla \mathbf{f}}_k = \frac{1}{m_k}\sum_{i=1}^{m_k} \nabla \mathbf{f}(\theta_k, \xi_{k,i})$
  Generate a positive definite Hessian inverse approximation H_k
  Calculate $\theta_{k+1} = \theta_k - \alpha_k H_k \widehat{\nabla \mathbf{f}}_k$
end for

- we use decreasing step sizes $\{\alpha_k\}$ as specified in Theorem 2;
- we use the mini-batch gradient instead of the momentum gradient (see Line 8).

Our proofs make use of the following assumptions, the first two of which, were made in [87].

Assumption 1. *$f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable. $f(\theta) \geq f^{low} > -\infty$, for any $\theta \in \mathbb{R}^n$. $\nabla \mathbf{f}$ is globally L-Lipschitz continuous; namely for any $x, y \in \mathbb{R}^n$, $\|\nabla \mathbf{f}(x) - \nabla \mathbf{f}(y)\| \leq L\|x - y\|$.*

Assumption 2. *For any iteration k, the stochastic gradient $\widehat{\nabla \mathbf{f}}_k = \widehat{\nabla \mathbf{f}}(\theta_k, \xi_k)$ satisfies: a) $\mathbb{E}_{\xi_k}\left[\widehat{\nabla \mathbf{f}}(\theta_k, \xi_k)\right] = \nabla \mathbf{f}(\theta_k)$, b) $\mathbb{E}_{\xi_k}\left[\left\|\widehat{\nabla \mathbf{f}}(\theta_k, \xi_k) - \nabla \mathbf{f}(\theta_k)\right\|^2\right] \leq \sigma^2$, where $\sigma > 0$, and $\xi_k, k = 1, 2, \ldots$ are independent samples that are independent of $\left\{\theta_j\right\}_{j=1}^{k}$.*

Assumption 3. *The activation functions ϕ_l have bounded values: $\exists \varphi > 0$ s.t. $\forall l, \forall h, |\phi_l(h)| \leq \varphi$.*

To prove the convergence, we prove Lemmas 1-3, which in addition to Assumptions AS.1-2, ensure that all of the assumptions in Theorem 2.8 in [87] are satisfied, and hence that the generic stochastic quasi-Newton (SQN) method, i.e. Algorithm 6, below converges. Specifically, Theorem 2.8 in [87] requires, in addition to Assumptions AS.1-2, the assumption

Assumption 4. *There exist two positive constants $\underline{\kappa}, \bar{\kappa}$, such that $\underline{\kappa} I \preceq H_k \preceq \bar{\kappa} I, \forall k$; for any $k \geq 2$, the random variable H_k depends only on $\xi_{[k-1]}$.*

To be more specific, in order to use the convergence analysis in [87], we need to show that the block-diagonal approximation of the inverse Hessian used in Algorithm 5 satisfies the assumption that it is bounded above and below by positive-definite matrices. Given the Kronecker structure of our Hessian inverse approximation, it suffices to prove boundness of both $H_a^l(k)$ and $H_g^l(k)$ for all iterations k. Making the additional assumption AS.3, we are able to prove Lemma 1, and hence Lemma 3, below. Note that many popular activation functions satisfy AS.3, such as sigmoid and tanh. In the following proofs, $\|\cdot\|$ denotes the 2-norm for vectors, and the spectral norm for matrices.

Lemma 1. *Suppose that AS.3 holds. There exist two positive constants $\underline{\kappa}_a$, $\bar{\kappa}_a$ such that $\underline{\kappa}_a I \preceq H_a^l(k) \preceq \bar{\kappa}_a I, \forall k, l$.*

Proof. Because $A_l^{\mathrm{LM}}(k) \succeq \lambda_A I_A$, we have that $H_a^l(k) \preceq \bar{\kappa}_a I_A$, where $\bar{\kappa}_a = \frac{1}{\lambda_A}$.

On the other hand, for any $\mathbf{x} \in \mathbb{R}^{d_l}$, by Cauchy-Schwarz, $\langle \mathbf{a}_{l-1}(i), \mathbf{x}\rangle^2 \leq \|\mathbf{x}\|^2 \|\mathbf{a}_{l-1}(i)\|^2 \leq \|\mathbf{x}\|^2(1+\varphi^2 d_l)$. Hence, $\left\|\overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}\right\| \leq 1+\varphi^2 d_l$; similarly, $\|A_l(0)\| \leq 1+\varphi^2 d_l$. Because $\|A_l(k)\| \leq \beta\|A_l(k-1)\| + (1-\beta)\left\|\overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}\right\|$, by induction, $\|A_l(k)\| \leq 1+\varphi^2 d_l$ for any k and l. Thus, $\|A_l^{\mathrm{LM}}(k)\| \leq 1+\varphi^2 d_l + \lambda_A$. Hence, $H_a^l(k) \succeq \underline{\kappa}_a I_A$, where $\underline{\kappa}_a = \frac{1}{1+\varphi^2 d_l+\lambda_A}$.

□

Lemma 2. *There exist two positive constants $\underline{\kappa}_g$ and $\bar{\kappa}_g$, such that $\underline{\kappa}_g I \preceq H_g^l(k) \preceq \bar{\kappa}_g I, \forall k, l$.*

Proof. To simplify notation, we omit the subscript g, superscript l and the iteration index k in the proof. Hence, our goal is to prove $\underline{\kappa}_g I \preceq H = H_g^l(k) \preceq \bar{\kappa}_g I$, for any l and k. Let $(\mathbf{s}_i, \mathbf{y}_i)$ $(i = 1, ..., p)$ denote the pairs used in an L-BFGS computation of H. Since $(\mathbf{s}_i, \mathbf{y}_i)$ was **not skipped**, $\frac{\mathbf{y}_i^\top \bar{H}^{(i)} \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_1}$, where $\bar{H}^{(i)}$ denotes the matrix H_g^l used at the iteration in which $\mathbf{s}_i$ and $\mathbf{y}_i$ were computed. Note that this is not the matrix H_i used in the recursive computation of H at the current iterate θ_k.

Given an initial estimate $H_0 = B_0^{-1} = I$ of $(G_g^l(\theta_k))^{-1}$, the L-BFGS method updates H_i recursively as

$$H_i = \left(I - \rho_i \mathbf{s}_i \mathbf{y}_i^\top\right) H_{i-1} \left(I - \rho_i \mathbf{y}_i \mathbf{s}_i^\top\right) + \rho_i \mathbf{s}_i \mathbf{s}_i^\top, \quad i = 1, \ldots, p, \tag{2.12}$$

where $\rho_i = (\mathbf{s}_i^\top \mathbf{y}_i)^{-1}$, and equivalently,

$$B_i = B_{i-1} + \frac{\mathbf{y}_i \mathbf{y}_i^\top}{\mathbf{s}_i^\top \mathbf{y}_i} - \frac{B_{i-1}\mathbf{s}_i \mathbf{s}_i^\top B_{i-1}}{\mathbf{s}_i^\top B_{i-1} \mathbf{s}_i}, \quad i = 1, \ldots, p,$$

where $B_i = H_i^{-1}$. Since we use DD with skipping, we have that $\frac{\mathbf{s}_i^\top \mathbf{s}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_2}$ and $\frac{\mathbf{y}_i^\top \bar{H}^{(i)} \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_1}$. Note that we don't have $\frac{\mathbf{y}_i^\top H_{i-1} \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_1}$, so we cannot direct apply (2.10). Hence, by (2.8), we have that $||B_i|| \leq ||B_{i-1}|| \left(1 + \frac{1}{\mu_1}\right)$. Hence, $||B|| = ||B_p|| \leq ||B_0|| \left(1 + \frac{1}{\mu_1}\right)^p = \left(1 + \frac{1}{\mu_1}\right)^p$. Thus, $B \preceq \left(1 + \frac{1}{\mu_1}\right)^p I$, $H \succeq \left(1 + \frac{1}{\mu_1}\right)^{-p} I := \underline{\kappa}_g I$.

On the other hand, since $\underline{\kappa}_g$ is a uniform lower bound for $H_g^l(k)$ for any k and l, $\bar{H}^{(i)} \succeq \underline{\kappa}_g I$.

Thus,

$$\frac{1}{\mu_1} \geq \frac{\mathbf{y}_i^\top \bar{H}^{(i)} \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \geq \underline{\kappa}_g \frac{\mathbf{y}_i^\top \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \Rightarrow \frac{\mathbf{y}_i^\top \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_1 \underline{\kappa}_g}.$$

Hence, using the fact that $||uv^\top|| = ||u|| \cdot ||v||$ for any vectors u, v, $||\rho_i \mathbf{s}_i \mathbf{s}_i^\top|| = \rho_i ||\mathbf{s}_i|| ||\mathbf{s}_i|| = \frac{\mathbf{s}_i^\top \mathbf{s}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_2}$,

$$\begin{aligned}
||H_i|| &= || \left(I - \rho_i \mathbf{s}_i \mathbf{y}_i^\top\right) H_{i-1} \left(I - \rho_i \mathbf{y}_i \mathbf{s}_i^\top\right) + \rho_i \mathbf{s}_i \mathbf{s}_i^\top || \\
&= ||H_{i-1} + \rho_i^2 (\mathbf{y}_i^\top H_{i-1} \mathbf{y}_i) \mathbf{s}_i \mathbf{s}_i^\top - \rho_i \mathbf{s}_i \mathbf{y}_i^\top H_{i-1} - \rho_i H_{i-1} \mathbf{y}_i \mathbf{s}_i^\top + \rho_i \mathbf{s}_i \mathbf{s}_i^\top || \\
&\leq ||H_{i-1}|| + ||\rho_i^2 (\mathbf{y}_i^\top H_{i-1} \mathbf{y}_i) \mathbf{s}_i \mathbf{s}_i^\top || + ||\rho_i \mathbf{s}_i \mathbf{y}_i^\top H_{i-1}|| + ||\rho_i H_{i-1} \mathbf{y}_i \mathbf{s}_i^\top || + ||\rho_i \mathbf{s}_i \mathbf{s}_i^\top || \\
&\leq ||H_{i-1}|| + ||H_{i-1}|| \cdot ||\rho_i^2 (\mathbf{y}_i^\top \mathbf{y}_i) \mathbf{s}_i \mathbf{s}_i^\top || + 2\rho_i ||\mathbf{s}_i|| \cdot ||\mathbf{y}_i^\top H_{i-1}|| + \frac{1}{\mu_2} \\
&\leq ||H_{i-1}|| + ||H_{i-1}|| \cdot \frac{1}{\mu_1 \underline{\kappa}_g} \frac{1}{\mu_2} + 2\rho_i ||\mathbf{s}_i|| \cdot ||\mathbf{y}_i^\top|| \cdot ||H_{i-1}|| + \frac{1}{\mu_2} \\
&\leq ||H_{i-1}|| \left(1 + \frac{1}{\mu_1 \underline{\kappa}_g} \frac{1}{\mu_2} + 2 \frac{1}{\sqrt{\mu_1 \mu_2 \underline{\kappa}_g}}\right) + \frac{1}{\mu_2} \\
&= \hat{\mu} ||H_{i-1}|| + \frac{1}{\mu_2}, \quad \text{where} \quad \hat{\mu} = \left(1 + \frac{1}{\sqrt{\mu_1 \mu_2 \underline{\kappa}_g}}\right)^2.
\end{aligned}$$

From the fact that $H_0 = I$, and induction, we have that $||H|| \leq \hat{\mu}^p + \frac{\hat{\mu}^p - 1}{\hat{\mu} - 1} \frac{1}{\mu_2} \equiv \bar{\kappa}_g$.

□

Lemma 3. *Suppose that AS.3 holds. Let $\theta_{k+1} = \theta_k - \alpha_k H_k \widehat{\nabla \mathbf{f}}_k$ be the step taken in Algorithm 5. There exists two positive constants $\underline{\kappa}$, $\bar{\kappa}$ such that $\underline{\kappa} I \preceq H_k \preceq \bar{\kappa} I, \forall k$.*

Proof. By Lemma 1, 2 and the fact that $H_k = \text{diag}\{H_a^1(k-1) \otimes H_g^1(k-1), ..., H_a^L(k-1) \otimes H_g^L(k-1)\}$. □

Using Lemma 3, we can now apply Theorem 2.8 in [87] to prove the convergence of Algorithm 5:

Theorem 2. *Suppose that assumptions AS.1-3 hold for $\{\theta_k\}$ generated by Algorithm 5 with mini-batch size $m_k = m$ for all k, and α_k is chosen as $\alpha_k = \frac{\underline{\kappa}}{L\bar{\kappa}^2}k^{-\beta}$, with $\beta \in (0.5, 1)$. Then*

$$\frac{1}{N}\sum_{k=1}^{N}\mathbb{E}\left[\|\nabla \mathbf{f}(\theta_k)\|^2\right] \leq \frac{2L\left(M_f - f^{low}\right)\bar{\kappa}^2}{\underline{\kappa}^2}N^{\beta-1} + \frac{\sigma^2}{(1-\beta)m}\left(N^{-\beta} - N^{-1}\right)$$

where N denotes the iteration number and $M_f > 0$ depends only on f. Moreover, for a given $\epsilon \in (0, 1)$, to guarantee that $\frac{1}{N}\sum_{k=1}^{N}\mathbb{E}\left[\|\nabla \mathbf{f}(\theta_k)\|^2\right] < \epsilon$, the number of iterations N needed is at most $O\left(\epsilon^{-\frac{1}{1-\beta}}\right)$.

Proof. To show that Algorithm 5 lies in the framework of Algorithm 6, it suffices to show that H_k generated by Algorithm 5 is positive definite, which is true since $H_k = \text{diag}\{H_a^1(k-1) \otimes H_g^1(k-1), ..., H_a^L(k-1) \otimes H_g^L(k-1)\}$ and $H_a^l(k)$ and $H_g^l(k)$ are positive definite for all k and l. Then by Lemma 3, and the fact that H_k depends on $H_a^l(k-1)$ and $H_g^l(k-1)$, and $H_a^l(k-1)$ and $H_g^l(k-1)$ does not depend on random samplings in the kth iteration, AS.4 holds. Hence, Theorem 2.8 of [87] applies to Algorithm 5, proving Theorem 2. □

Note: other theorems in [87], namely Theorems 2.5 and 2.6, also apply here under our assumptions.

2.8 Experiments

2.8.1 Major Numerical Experiments

We tested K-BFGS and K-BFGS(L), as well as KFAC, Adam/RMSprop and SGD-m (SGD with momentum) on three autoencoder problems, namely, MNIST [51], FACES, and CURVES, which are used in e.g. Hinton and Salakhutdinov [43], Martens [59], and Martens and Grosse [62], except that we replaced the sigmoid activation with ReLU. See Section 2.8.2 for a complete description of these problems and the competing algorithms.

Since one can view Powell's damping with $B = I$ as LM damping, we write $\mu_2 = \lambda_G$, where λ_G denotes the LM damping parameter for G_l. We then define $\lambda = \lambda_A\lambda_G$ as the overall damping

term of our QN approximation. To simplify matters, we chose $\lambda_A = \lambda_G = \sqrt{\lambda}$, so that we needed to tune only one hyper-parameter (HP) λ.

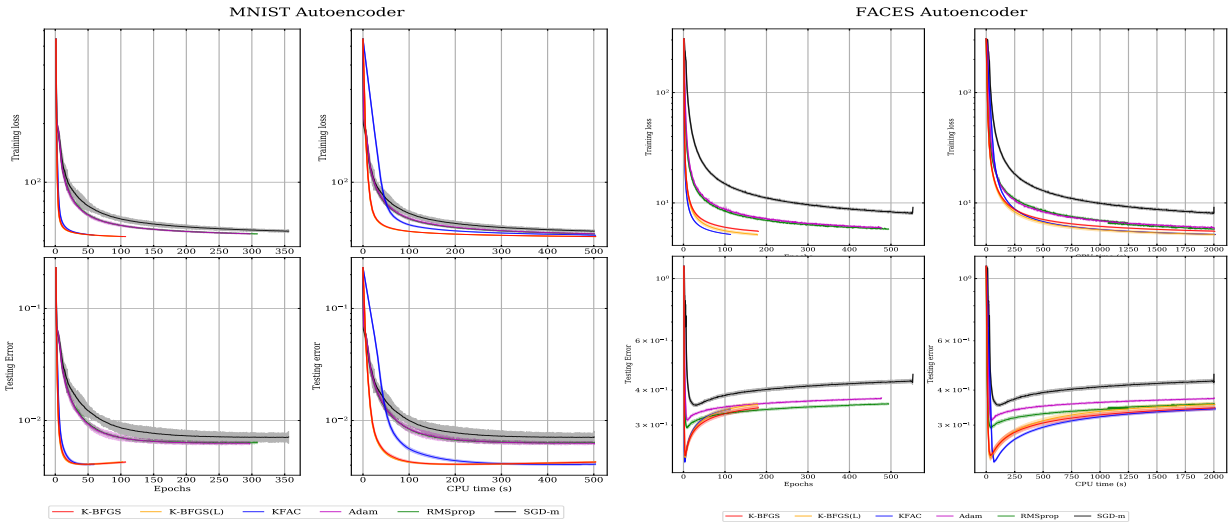
To obtain the results in Figure 2.2, we first did a grid-search on (learning rate, damping) pairs for all algorithms (except for SGD-m, whose grid-search was only on learning rate), where damping refers to λ for K-BFGS/K-BFGS(L)/KFAC, and ϵ for RMSprop/Adam. We then selected the best (learning rate, damping) pairs with the lowest training loss encountered. The range for the grid-search and the best HP values (as well as other fixed HP values) are listed in Section 2.8.2. Using the best HP values that we found, we then made 20 runs employing different random seeds, and plotted the mean value of the 20 runs as the solid line and the standard deviation as the shaded area.[1] For each problem, the upper (lower) rows depicts training loss (testing (mean square) error), whereas the left (right) column depicts training/test progress versus epoch (CPU time), respectively. After each epoch, the training loss/testing error from the whole training/testing set is reported (the time for computing this loss is not included in the plots). For each problem, algorithms are terminated after the same amount of CPU time.

From the training loss plots in Figure 2.2, our algorithms clearly outperformed the first-order methods, except for RMSprop/Adam on CURVES, with respect to CPU time, and performed comparably to KFAC in terms of both CPU time and number of epochs taken. The testing error plots in Figure 2.2 show that our K-BFGS(L) method and KFAC behave very similarly and substantially outperform all of the first-order methods in terms of both of these measures. This suggests that our algorithms not only optimize well, but also generalize well.

To further demonstrate the robustness of our algorithms, we examined the loss under various HP settings, which showed that our algorithms are stable under a fairly wide range for the HPs (see Section 2.8.2).

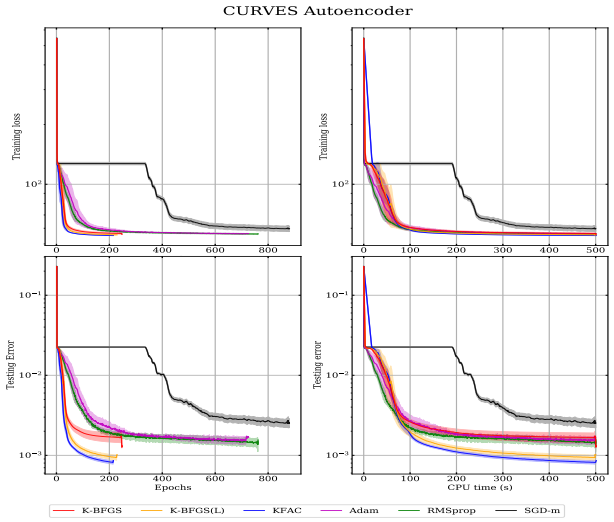
We also repeated our experiments using mini-batches of size 100 for all algorithms (see Figures 2.4, 2.5, and 2.6, where HPs are optimally tuned for batch sizes of 100) and our proposed methods continue to demonstrate advantageous performance, both in training and testing. For these experi-

[1] Results were obtained on a machine with 8 x Intel(R) Xeon(R) CPU @ 2.30GHz and 1 x NVIDIA Tesla P100. Code is available at `https://github.com/renyiryry/kbfgs_neurips2020_public`.



(a) MNIST

(b) FACES



(c) CURVES

Figure 2.2: Comparison between K-BFGS(-20), K-BFGS(L)(-20), and other comparing methods on MNIST, FACES, and CURVES

ments, we did not experiment with 20 random seeds. These results show that our approach works as well for relatively small mini-batch sizes of 100, as those of size 1000, which are less noisy, and hence is robust in the stochastic setting employed to train DNNs.

Compared with other methods mentioned in this section, our K-BFGS and K-BFGS(L) have the extra advantage of being able to double the size of minibatch for computing the stochastic gradient with almost no extra cost, which might be of particular interest in a highly stochastic setting. See Section 2.8.3 for more discussion on this and some preliminary experimental results.

2.8.2 Implementation Details of the Experiments

Description of Competing Algorithms

KFAC

We first describe KFAC in Algorithm 7. Note that G_l in KFAC refers to the G matrices in Martens and Grosse [62], which is different from the G_l in K-BFGS.

Algorithm 7 KFAC

Require: Given θ_0, batch size m, and learning rate α, damping value λ, inversion frequency T

for $k = 1, 2, \ldots$ **do**
  Sample mini-batch of size m: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$
  Perform a forward-backward pass over the current mini-batch M_k (see Algorithm 1)
  for $l = 1, 2, \ldots L$ **do**
    $p_l = H_g^l \widehat{\nabla \mathbf{f}}_l H_a^l$
    $W_l = W_l - \alpha \cdot p_l$.
  end for
  Perform another pass over M_k with y sampled from the predictive distribution
  Update $A_l = \beta \cdot A_l + (1-\beta) \cdot \overline{\mathbf{a}_{l-1}\mathbf{a}_{l-1}^\top}$, $G_l = \beta \cdot G_l + (1-\beta) \cdot \overline{\mathbf{g}_l \mathbf{g}_l^\top}$
  if $k \leq T$ or $k \equiv 0 \pmod{T}$ **then**
    Recompute $H_a^l = (A_l + \sqrt{\lambda} I)^{-1}$, $H_g^l = (G_l + \sqrt{\lambda} I)^{-1}$
  end if
end for

Adam/RMSprop

We implement Adam and RMSprop exactly as in Kingma and Ba [46] and Hinton *et al.* [42], respectively. Note that the only difference between them is that Adam does bias correction for the 1st and 2nd moments of gradient while RMSprop does not.

Initialization of Algorithms

We now describe how each algorithm is initialized. For all algorithms, $\widehat{\nabla \mathbf{f}}$ is always initialized as zero.

For second-order information, we use a "warm start" to estimate the curvature when applicable. In particular, we estimate the following curvature information using the entire training set before we start updating parameters. The information gathered is

- A_l for K-BFGS and K-BFGS(L);
- A_l and G_l for KFAC;
- $\nabla f \odot \nabla f$ for RMSprop;
- Not applicable to Adam because of the bias correction.

Lastly, for K-BFGS and K-BFGS(L), H_a^l is always initially set to an identity matrix. H_g^l is also initially set to an identity matrix in K-BFGS; for K-BFGS(L), when updating H_g^l using L-BFGS, the incorporation of the information from the p $(\mathbf{s}, \mathbf{y})$ pairs is applied to an initial matrix that is set to an identity matrix. Hence, the above initialization/warm start costs are roughly twice as large for KFAC as they are for K-BFGS and K-BFGS(L).

Autoencoder Problems

Table 2.3 lists information about the three datasets, namely, MNIST[2], FACES[3], and CURVES[4]. Table 2.4 specifies the architecture of the 3 problems, where binary entropy $\mathcal{L}(a_L, y) = \sum_n [y_n \log a_{L,n} + (1 - y_n) \log(1 - a_{L,n})]$, MSE $\mathcal{L}(a_L, y) = \frac{1}{2} \sum_n (a_{L,n} - y_n)^2$. Besides the loss function in Table 2.4, we further add a regularization term $\frac{\eta}{2}||\theta||^2$ to the loss function, where $\eta = 10^{-5}$.

[2]Downloadable at `http://yann.lecun.com/exdb/mnist/`
[3]Downloadable at `www.cs.toronto.edu/~jmartens/newfaces_rot_single.mat`
[4]Downloadable at `www.cs.toronto.edu/~jmartens/digs3pts_1.mat`

Table 2.3: Info for the MNIST, FACES, and CURVES datasets

Dataset	# data points	# training examples	# testing examples
MNIST	70,000	60,000	10,000
FACES	165,600	103,500	62,100
CURVES	30,000	20,000	10,000

Table 2.4: Architecture of 3 auto-encoder problems

Dataset	Layer width & activation	Loss function
MNIST	[784, 1000, 500, 250, 30, 250, 500, 1000, 784] [ReLU, ReLU, ReLU, linear, ReLU, ReLU, ReLU, sigmoid]	binary entropy
FACES	[625, 2000, 1000, 500, 30, 500, 1000, 2000, 625] [ReLU, ReLU, ReLU, linear, ReLU, ReLU, ReLU, linear]	MSE
CURVES	[784, 400, 200, 100, 50, 25, 6, 25, 50, 100, 200, 400, 784] [ReLU, ReLU, ReLU, ReLU, ReLU, linear, ReLU, ReLU, ReLU, ReLU, ReLU, sigmoid]	binary entropy

Specification of Hyper-parameters

In our experiments, we focus our tuning effort onto learning rate and damping. The range of the tuning values is listed below:

- learning rate $\alpha_k = \alpha \in \{$ 1e-5, 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, 1e-2, 3e-2, 1e-1, 3e-1, 1, 3, 10 $\}$.
- damping:
  - λ for K-BFGS, K-BFGS(L) and KFAC: $\lambda \in \{$ 3e-3, 1e-2, 3e-2, 1e-1, 3e-1, 1, 3 $\}$.
  - ϵ for RMSprop and Adam: $\epsilon \in \{$ 1e-10, 1e-8, 1e-6, 1e-4, 1e-3, 1e-2, 1e-1 $\}$.
  - Not applicable for SGD with momentum.

The best hyper-parameters (HPs) were chosen by computing the deterministic loss function encountered at the end of every epoch until the algorithm was terminated, and then choosing the lowest of these values as the criterion for evaluating the efficacy of that choice of HPs. These values were used in Figure 2.2 and are listed in Table 2.5. Besides the tuning hyper-parameters, we also list other fixed hyper-parameters with their values:

Table 2.5: Best HP values (learning rate, damping) for Figure 2.2

	K-BFGS	K-BFGS(L)	KFAC	Adam	RMSprop	SGD-m
MNIST	(0.3, 0.3)	(0.3, 0.3)	(1, 3)	(1e-4, 1e-4)	(1e-4, 1e-4)	(0.03, -)
FACES	(0.1, 0.1)	(0.1, 0.1)	(0.1, 0.1)	(1e-4, 1e-4)	(1e-4, 1e-4)	(0.01, -)
CURVES	(0.1, 0.01)	(0.3, 0.3)	(0.3, 0.3)	(1e-3, 1e-3)	(1e-3, 1e-3)	(0.1, -)

- Size of minibatch $m = 1000$, which is also suggested in Botev *et al.* [15].
- Decay parameter:
  - K-BFGS, K-BFGS(L): $\beta = 0.9$;
  - KFAC: $\beta = 0.9$;
  - RMSprop, Adam: Following the notation in Kingma and Ba [46], we use $\beta_1 = \beta_2 = 0.9$;[5]
  - SGD with momentum: $\beta = 0.9$.
- Other:
  - $\mu_1 = 0.2$ in double damping (DD):

    We recommend to leave the value as default because μ_1 represents the "ratio" between current and past, which is scaling invariant;
  - Number of $(\mathbf{s}, \mathbf{y})$ pairs stored for K-BFGS(L) $p = 100$:

    It might be more efficient to use a smaller p for the narrow layers. We didn't investigate this for simplicity and consistency;
  - Inverse frequency $T = 20$ in KFAC.

[5]The default value of β_2 recommended in Kingma and Ba [46] is 0.999. Hence, we also tested $\beta_2 = 0.999$, and obtained results that were similar to those presented in Figure 2.2 (i.e., with $\beta_2 = 0.9$). For the sake of fair comparison, we chose to report the results with $\beta_2 = 0.9$.

Sensitivity to Hyper-parameters

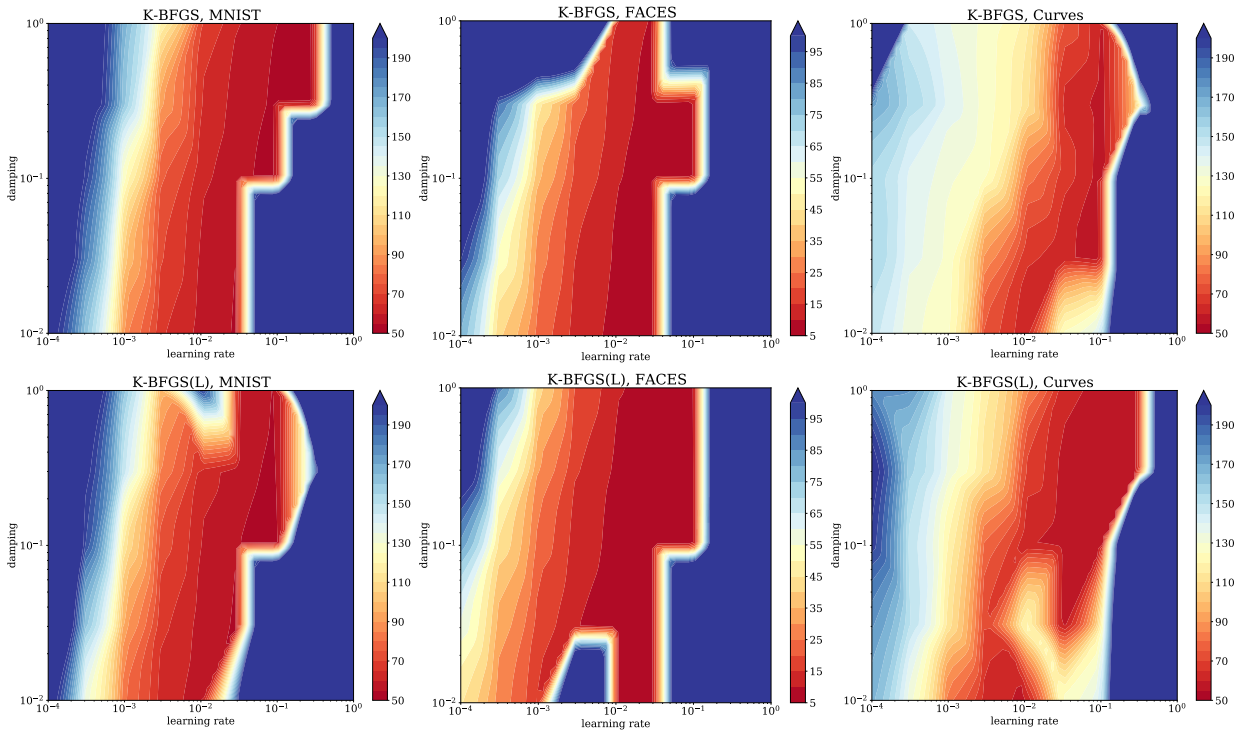


Figure 2.3: Landscape of loss achieved by K-BFGS(-20) and K-BFGS(L)(-20) w.r.t hyper-parameters (i.e. learning rate and damping).

Figure 2.3 shows the sensitivity of K-BFGS and K-BFGS(L) to hyper-parameter values (i.e. learning rate and damping). The left, middle, right columns depict results for MNIST, FACES, CURVES, which are terminated after 500, 2000, 500 seconds (CPU time), respectively, for K-BFGS (upper) and K-BFGS(L) (lower) row. The x-axis corresponds to the learning rate α, while the y-axis correspond to the damping value λ. Color corresponds to the loss after a certain amount of CPU time. We can see that both K-BFGS and K-BFGS(L) are robust within a fairly wide range of hyper-parameters.

To get the plot, we first obtained training loss with $\alpha \in$ {1e-4, 3e-4, 1e-3, 3e-3, 1e-2, 3e-2, 1e-1, 3e-1, 1} and $\lambda \in$ {1e-2, 3e-2, 1e-1, 3e-1, 1}, and then drew contour lines of the loss within the above ranges.

Table 2.6: Best HP values (learning rate, damping) for Figures 2.4, 2.5, 2.6

	K-BFGS	K-BFGS(L)	KFAC	Adam	RMSprop	SGD-m
MNIST	(0.1, 0.3)	(0.1, 0.3)	(0.1, 0.3)	(1e-4, 1e-4)	(1e-4, 1e-4)	(0.03, -)
FACES	(0.03, 0.03)	(0.03, 0.3)	(0.03, 0.3)	(3e-5, 1e-4)	(3e-5, 1e-4)	(0.01, -)
CURVES	(0.3, 1)	(0.3, 0.3)	(0.03, 0.1)	(3e-4, 1e-4)	(3e-3, 1e-4)	(0.03, -)

2.8.3 Additional Numerical Experiments

Experimental Results Using Mini-batches of Size 100

We repeated our experiments using mini-batches of size 100 for all algorithms (see Figures 2.4, 2.5, and 2.6). For each figure, the upper (lower) rows depict training loss (testing (mean square) error), whereas the left (right) column depicts training/test progress versus epoch (CPU time), respectively.

The best hyper-parameters were those that produce the lowest value of the deterministic loss function encountered at the end of every epoch until the algorithm was terminated. These values were used in Figures 2.4, 2.5, 2.6 and are listed in Table 2.6.

Our proposed methods continue to demonstrate advantageous performance, both in training and testing. It is interesting to note that, whereas for a minibatch size of 1000, KFAC slightly outperformed K-BFGS(L), for a minibatch size of 100, K-BFGS(L) clearly outperformed KFAC in training on CURVES.

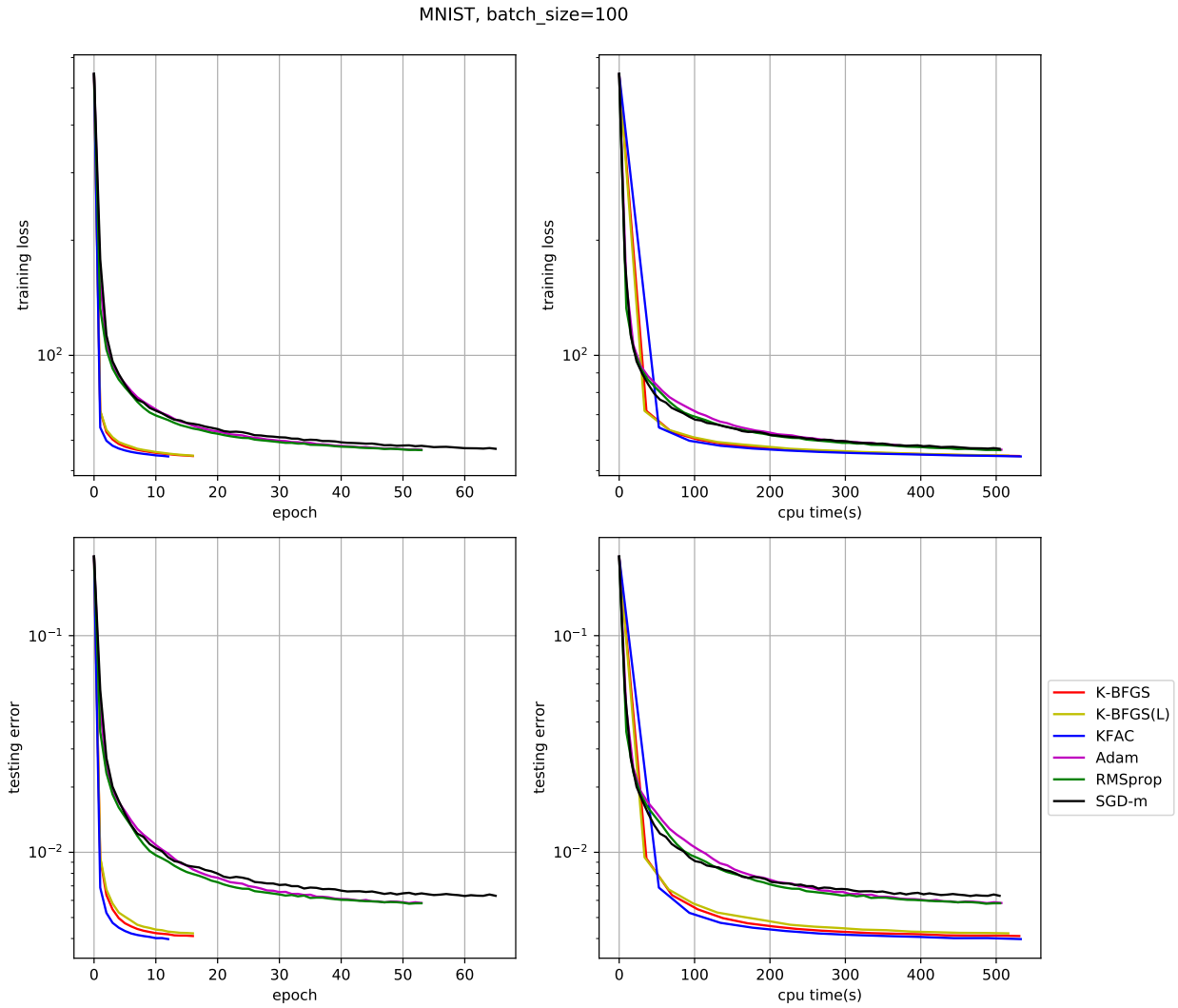


Figure 2.4: Comparison between K-BFGS(-20), K-BFGS(L)(-20), and other comparing methods on MNIST with batch size 100

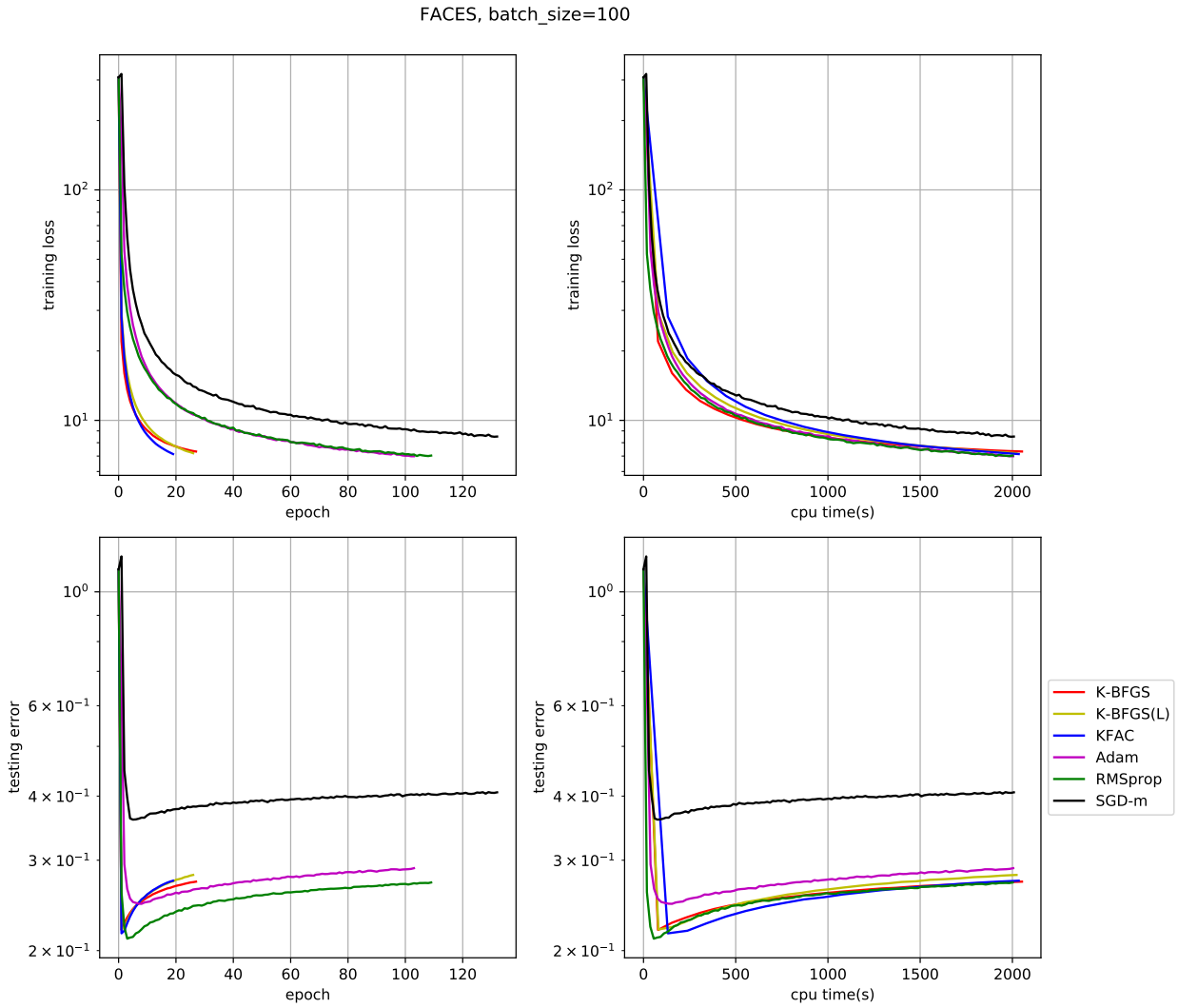


Figure 2.5: Comparison between K-BFGS(-20), K-BFGS(L)(-20), and other comparing methods on FACES with batch size 100

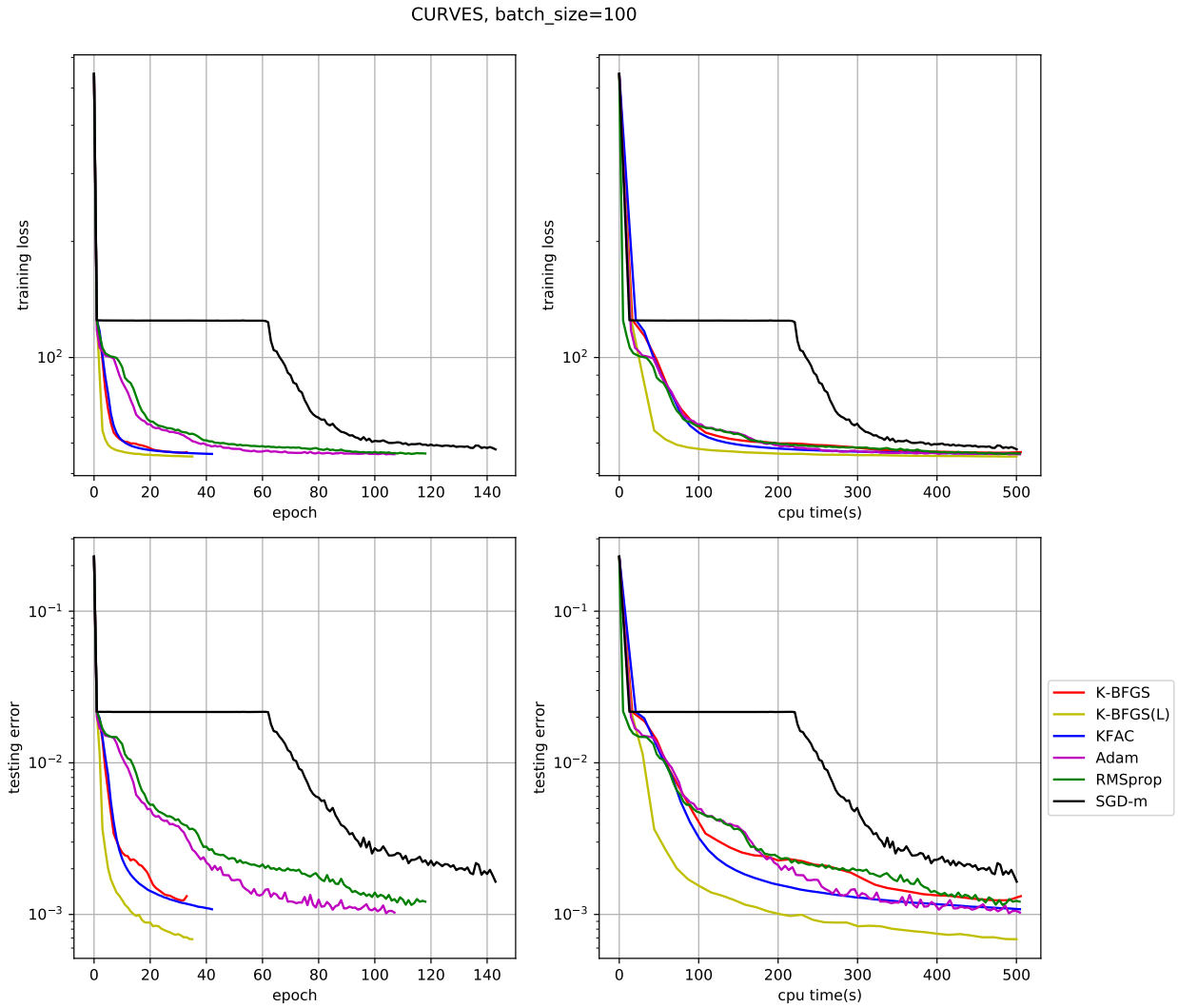


Figure 2.6: Comparison between K-BFGS(-20), K-BFGS(L)(-20), and other comparing methods on CURVES with batch size 100

Doubling the Mini-batch for the Gradient at Almost No Cost

Compared with other methods mentioned in this chapter, our K-BFGS and K-BFGS(L) methods have the extra advantage of being able to double the size of the minibatch used to compute the stochastic gradient with almost no extra cost, which might be of particular interest in a highly stochastic setting. To accomplish this, we can make use of the stochastic gradient $\overline{\nabla \mathbf{f}}^{+}$ computed in the **second** pass of the previous iteration that is needed for computing the $(\bar{\mathbf{s}}, \bar{\mathbf{y}})$ pair for applying the BFGS or L-BFGS updates, and average it with the stochastic gradient $\overline{\nabla \mathbf{f}}$ of the current

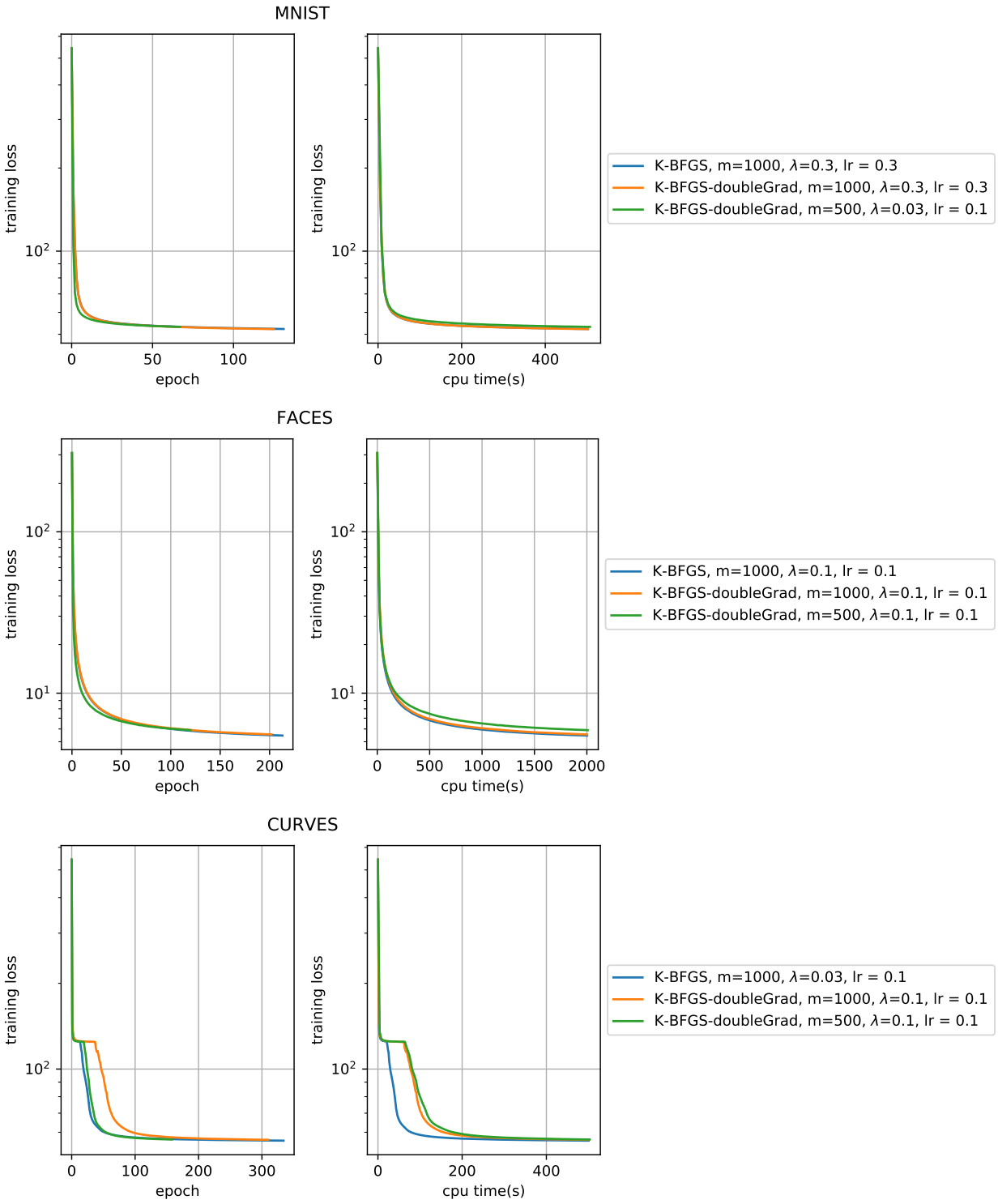


Figure 2.7: Comparison between K-BFGS(-20) and its "double-grad" variants

iteration. For example if the size of minibatch is $m = 1000$, the above "double-grad-minibatch" method computes a stochastic gradient from 2000 data points at each iteration, except at the very first iteration.

The results of some preliminary experiments are depicted in Figure 2.7, where we compare an earlier version of the K-BFGS algorithm (Algorithm 4), which uses a slightly different variant of Hessian-action to update H_a^l, using a size of $m = 1000$ for mini-batches, with its "double-grad-minibatch" variants for $m = 500$ and 1000. Even though "double-grad" does not help much in these experiments, our K-BFGS algorithm performs stably across these different variants. These results indicate that there is a potential for further improvements; e.g., a finer grid search might identify hyper-parameter values that result in better performing algorithms.

2.9 Conclusion

We proposed Kronecker-factored QN methods, namely, K-BFGS and K-BFGS(L), for training multi-layer feed-forward neural network models, that use layer-wise second-order information and require modest memory and computational resources. Experimental results indicate that our methods outperform or perform comparably to the state-of-the-art first-order and second-order methods. Our methods can also be extended to convolutional and recurrent NNs.

Chapter 3: Kronecker-factored Quasi-Newton Methods for Deep Learning

3.1 Introduction and Summary of Contributions

In this chapter, we continue the study and development of the Kronecker-factored quasi-Newton (QN) methods described in Chapter 2. See Section 2.1 for the motivation and relevant literature for these Kronecker-factored QN methods.

To be more specific, we propose new versions of K-BFGS, that are substantial extensions to the Kronecker-factored quasi-Newton methods for MLPs proposed in Chapter 2. Not only can they be applied to CNNs, but they also incorporate several generic improvements beyond what was proposed in Chapter 2.

To enable K-BFGS to train CNN models, we first show that for a convolutional layer, the gradient and Hessian restricted to that layer can be approximated by the Kronecker product of two vectors and matrices, respectively, extending the results in Botev *et al.* [15] and Wu *et al.* [89], etc. We then formalize exactly how K-BFGS should be applied to optimize the parameters in convolutional layers, with the Kronecker-factored approximation of the Hessian as the basis.

Our generic improvements to Chapter 2 include a new double damping technique $D_P D_{LM}$ and a "minibatched" Hessian-action BFGS, both of which are applicable to convolutional and fully-connected layers.

Our proposed methods have comparable memory requirements to those of first-order methods, while their per-iteration time complexities are smaller, and in many cases, much smaller than those of Chapter 2 and other popular second-order methods such as KFAC. Further, we prove convergence results for a limited memory K-BFGS(L) variant under relatively mild conditions.

We conducted experiments on several MLP autoencoder problems, which demonstrated that our improved versions of K-BFGS outperformed the ones proposed in Chapter 2. Moreover, on

several well-studied CNN problems, our proposed method outperformed the 1st-order SOTA methods SGD-m (i.e., SGD with momentum) and Adam and performed comparably to the 2nd-order SOTA method KFAC.

3.2 Kronecker-factored Structures in CNNs

In this secton, after first discussing the computations involved in a CNN model, we describe the Kronecker structures of the gradient and Hessian of the loss function with respect to a convolutional layer's parameters.

3.2.1 Convolutional Neural Networks (CNNs)

We consider a CNN with L trainable layers (for simplicity, assume they are all convolutional layers), with parameters consisting of a weight tensor w_l and a bias vector b_l (shapes specified later) for $l \in \{1, ..., L\}$ and a loss function $\mathcal{L}$. For a data-point (x, y), x is fed into the CNN as input, yielding $\hat{y}$ as the output. The loss $\mathcal{L}(\hat{y}, y)$ between the output $\hat{y}$ and y is a non-convex function of the set of all trainable parameters $\theta := \{w_1, b_1, ..., w_L, b_L\}$.

For a dataset that contains multiple data-points indexed by $n = 1, ..., N$, let $f(n; \theta)$ denote the loss from the nth data-point. Thus, viewing the dataset as an empirical distribution, the actual loss function that we wish to minimize is

$$f(\theta) := \mathbb{E}_n[f(n; \theta)] := \frac{1}{N} \sum_{n=1}^{N} f(n; \theta).$$

Let us now focus on a single convolutional layer of the CNN, with its own weight tensor w and bias vector b as the trainable parameters. For simplicity, we omit the layer index l, and assume that:

1. the convolutional layer is 2-dimensional;
2. the filters are of size $(2R + 1) \times (2R + 1)$, with spatial offsets from the centers of each filter indexed by $\delta \in \Delta := \{-R, ..., R\} \times \{-R, ..., R\}$;

3. the stride is of length 1, and the padding is equal to R, so that the sets of input and output spatial locations ($t \in \mathcal{T} \subset \mathbf{R}^2$) are the same.[1];
4. the layer has J input channels indexed by $j = 1, ..., J$, and I output channels indexed by $i = 1, ..., I$.

The weight tensor $w \in \mathbf{R}^{I \times J \times (2R+1) \times (2R+1)}$ corresponds to the elements of all of the filters in the layer. Hence, an element of w is denoted as $w_{i,j,\delta}$, where the first two indices i, j are the output/input channels, and the last two indices δ are the spatial offset within a filter. The bias $b \in \mathbf{R}^I$ is a length-I vector.

Let a, with components $a_{j,t}$, denote the input to the layer after padding is added, where t denotes the spatial location of the padded input and $j = 1, ..., J$; and let h, with components $h_{i,t}$, denote the output of the layer, where t denote the spatial location of the output and $i = 1, ..., I$. Given a, h is computed as

$$h_{i,t} = \sum_{j=1}^{J} \sum_{\delta \in \Delta} w_{i,j,\delta} a_{j,t+\delta} + b_i, \quad t \in \mathcal{T}, \, i = 1, ..., I. \tag{3.1}$$

Note that we only consider the linear transformation of the convolutional layer. In other words, if there is any activation or any other operations such as batch normalization afterwards, we view it as being separate from the layer.

3.2.2 Kronecker-factored Structure of Gradient and Hessian for Convolutional Layers

Recent work has shown that curvature information in DNNs has the property of being a sum (or average, equivalently) of Kronecker products, beginning with the development of KFAC [62, 38], which showed this for Fisher matrices. Botev *et al.* [15] and Chapter 2 showed that Hessian matrices are also a sum of Kronecker product for fully-connected layers, while Bakker *et al.* [10] and Wu *et al.* [89] extended this result to convolutional layers.

Based on the above Kronecker-factored structures, practical second-order methods for deep

[1]The derivations in this chapter can also be extended to the case where stride is greater than 1.

learning models were proposed, by approximating the curvature as a single Kronecker product, including KFAC [62, 38], KFRA [15], and K-BFGS-20 (Chapter 2).

Without developing any training method, Wu *et al.* [89] proposed a single Kronecker product approximation to the Hessian of a DNN that consists of alternating convolutional layers and ReLU activation functions, without any further modifications, such as batch normalization or skip connections Moreover, assuming all activation functions are ReLU results in the "second order" term in the Hessian being zero and the Hessian being equivalent to a Gauss-Newton matrix (or equivalently, a Fisher matrix). In contrast, we show that the Hessian has a Kronecker-factor structure for convolutional layers, without any assumptions about the activation functions or model architectures.

Case 1: Single Data-point

We now consider a single data-point, omitting the index n for simplicity, and derive the structure of gradient and Hessian with respect to the loss function $f(\cdot;\theta)$.

We define $\mathcal{D}X := \frac{\partial f}{\partial X}$ for any variable X and $\text{vec}(\cdot)$ to be the vectorization of a matrix. For the output and input of the layer, we define, respectively, the vectors

$$\begin{aligned}\mathbf{h}_t &:= \left(h_{1,t}, ..., h_{I,t}\right)^\top \in \mathbb{R}^I,\\ \mathbf{a}_t &:= \left(a_{1,t+\delta_1}, ..., a_{J,t+\delta_{|\Delta|}}, 1\right)^\top \in \mathbb{R}^{J|\Delta|+1},\end{aligned}$$

for $t \in \mathcal{T}$. Note that a homogeneous coordinate is concatenated at the end of $\mathbf{a}_t$.

For the weights and biases, we define the vectors

$$\mathbf{w}_i := \left(w_{i,1,\delta_1}, ..., w_{i,J,\delta_{|\Delta|}}, b_i\right)^\top \in \mathbb{R}^{J|\Delta|+1},$$

for $i = 1, ..., I$, and from them the matrix

$$W := (\mathbf{w}_1, ..., \mathbf{w}_I)^\top \in \mathbb{R}^{I\times(J|\Delta|+1)}, \tag{3.2}$$

which contains all the parameters of the convolutional layer.

In Theorem 3, we now give the structure of the gradient and Hessian of W for a single data-point.

Theorem 3. *For a single data-point,*

i) (Structure of gradient)

$$vec(\mathcal{D}W) = \sum_{t\in\mathcal{T}} \mathbf{a}_t \otimes \mathcal{D}\mathbf{h}_t$$

is the sum of $|\mathcal{T}|$ Kronecker products;

ii) (Structure of Hessian)

$$\frac{\partial^2 f}{\partial vec(W)^2} = \sum_{t,t'\in\mathcal{T}} A_{t,t'} \otimes G_{t,t'}$$

is the sum of $|\mathcal{T}|^2$ Kronecker products, where

$$\begin{aligned} A_{t,t'} &:= \mathbf{a}_t \mathbf{a}_{t'}^\top \in \mathbb{R}^{(J|\Delta|+1)\times(J|\Delta|+1)}, \\ G_{t,t'} &:= \frac{\partial^2 f}{\partial \mathbf{h}_t \partial \mathbf{h}_{t'}} \in \mathbb{R}^{I\times I}. \end{aligned} \tag{3.3}$$

for $t, t' \in \mathcal{T}$.

Proof. We first derive the structure of the gradient. By (3.1),

$$\begin{aligned} \frac{\partial f}{\partial w_{i,j,\delta}} &= \sum_{t\in\mathcal{T}} \frac{\partial f}{\partial h_{i,t}} \frac{\partial h_{i,t}}{\partial w_{i,j,\delta}} = \sum_{t\in\mathcal{T}} \mathcal{D}h_{i,t} a_{j,t+\delta}, \\ \frac{\partial f}{\partial b_i} &= \sum_{t\in\mathcal{T}} \frac{\partial f}{\partial h_{i,t}} \frac{\partial h_{i,t}}{\partial b_i} = \sum_{t\in\mathcal{T}} \mathcal{D}h_{i,t}. \end{aligned}$$

Hence,

$$\mathcal{D}\mathbf{w}_i = \sum_{t\in\mathcal{T}} \mathcal{D}h_{i,t}\mathbf{a}_t \Rightarrow \mathcal{D}W = \sum_{t\in\mathcal{T}} \mathcal{D}\mathbf{h}_t \left(\mathbf{a}_t\right)^\top.$$

and $\mathrm{vec}(\mathcal{D}W) = \sum_{t\in\mathcal{T}} \mathbf{a}_t \otimes \mathcal{D}\mathbf{h}_t$.

To derive the Hessian of $f(\cdot;\theta)$ for a single data-point, it follows from (3.1) that

$$\frac{\partial(\mathcal{D}h_{i,t})}{\partial w_{i',j,\delta}} = \sum_{t'} \frac{\partial(\mathcal{D}h_{i,t})}{\partial h_{i',t'}} \frac{\partial h_{i',t'}}{\partial w_{i',j,\delta}} = \sum_{t'} \frac{\partial^2 f}{\partial h_{i,t} \partial h_{i',t'}} a_{j,t'+\delta},$$
$$\frac{\partial(\mathcal{D}h_{i,t})}{\partial b_{i'}} = \sum_{t'} \frac{\partial(\mathcal{D}h_{i,t})}{\partial h_{i',t'}} \frac{\partial h_{i',t'}}{\partial b_{i'}} = \sum_{t'} \frac{\partial^2 f}{\partial h_{i,t} \partial h_{i',t'}}.$$

Hence,

$$\frac{\partial(\mathcal{D}h_{i,t})}{\partial \mathbf{w}_{i'}} = \sum_{t'} \frac{\partial^2 f}{\partial h_{i,t} \partial h_{i',t'}} \mathbf{a}_{t'},$$

and

$$\frac{\partial^2 f}{\partial \mathbf{w}_i \partial \mathbf{w}_{i'}} = \frac{\partial}{\partial \mathbf{w}_{i'}} \left(\sum_t \mathcal{D}h_{i,t} \mathbf{a}_t \right) = \sum_t \frac{\partial(\mathcal{D}h_{i,t})}{\partial \mathbf{w}_{i'}} \mathbf{a}_t^\top = \sum_t \sum_{t'} \frac{\partial^2 f}{\partial h_{i,t} \partial h_{i',t'}} A_{t,t'}.$$

Hence,

$$\frac{\partial^2 f}{\partial \mathrm{vec}(W)^2} = \sum_{t,t'} A_{t,t'} \otimes G_{t,t'}.$$

□

If we **assume** that $G_{t,t'} = 0$ for $t \neq t'$, we have that

$$\frac{\partial^2 f}{\partial \mathrm{vec}(W)^2} \approx \sum_{t \in \mathcal{T}} A_{t,t} \otimes G_{t,t}. \tag{3.4}$$

As in other methods that have been proposed for training DNNs that use Kronecker factored approximations to the Hessian or other pre-conditioning matrices [62, 38, 15], we further approximate $\frac{\partial^2 f}{\partial \mathrm{vec}(W)^2}$ by a single Kronecker product. To achieve this, we now approximate the average of the Kronecker products of a set of matrix pairs $\{(U_t, V_t)\}$ by the Kronecker product of the averages

of individual sets of matrices $\{U_t\}$, $\{V_t\}$, i.e.,

$$\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} U_t \otimes V_t \approx \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} U_t\right) \otimes \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} V_t\right). \tag{3.5}$$

Applying (3.5) to (3.4), we have that

$$\begin{aligned}\frac{\partial^2 f}{\partial \mathrm{vec}(W)^2} &\approx |\mathcal{T}| \cdot \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} A_{t,t}\right) \otimes \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} G_{t,t}\right) \\ &= \left(\sum_{t\in\mathcal{T}} A_{t,t}\right) \otimes \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} G_{t,t}\right). \end{aligned} \tag{3.6}$$

Note that the assumptions we made in deriving (3.6) are analogous to the IAD (Independent Activations and Derivatives), SH (Spatial Homogeneity), and SUD (Spatially Uncorrelated Derivatives) assumptions in Grosse and Martens [38]. Lastly, one can similarly derives a single Kronecker approximation for the gradient, using Theorem 3 and (3.5).

Case 2: Multiple Data-points

In the case of multiple data-points, we use (n) to denote the index of a data-point. To approximate the average Hessian across multiple data-points as a single Kronecker product, we again use (3.5), but averaging over the data points this time. By (3.6), we have that

$$\begin{aligned}\frac{\partial^2 f}{\partial \mathrm{vec}(W)^2} &= \mathbb{E}_n\left[\frac{\partial^2 f(n)}{\partial \mathrm{vec}(W)^2}\right] \\ &\approx \mathbb{E}_n\left[\left(\sum_{t\in\mathcal{T}} A_{t,t}(n)\right) \otimes \left(\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} G_{t,t}(n)\right)\right] \qquad (3.7) \\ &\approx A \otimes G, \qquad (3.8)\end{aligned}$$

where

$$A := \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} A_{t,t}(n)\right], \quad G := \mathbb{E}_n\left[\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} G_{t,t}(n)\right]. \tag{3.9}$$

(3.8) will serve as the foundation for us to develop the Kronecker-factored QN method for CNNs below.

3.3 Our New K-BFGS Method

3.3.1 K-BFGS-20 in Chapter 2

The Kronecker-factored quasi-Newton method proposed in Chapter 2 for training multilayer perceptrons (MLPs), approximated the Hessian of the loss function by a block diagonal matrix, where each block corresponds to the Hessian w.r.t. the parameters of a fully-connected layer. As a result, the parameters of each layer can be updated separately.

For a single fully-connected layer in the MLP, Chapter 2 approximates the Hessian restricted to that layer as $H_A \otimes H_G$, where H_A and H_G are some approximations to some matrices A^{-1} and G^{-1}, respectively. As a result, by the property of Kronecker product, the parameters of this fully-connected layer can be updated by computing

$$W^+ = W - \alpha H_G(\mathcal{D}W)H_A, \tag{3.10}$$

where W denotes the parameters (including weights and biases) in the fully-connected layer and α denotes the learning rate.

Furthermore, in Chapter 2, H_A and H_G, as the approximations to A^{-1} and G^{-1}, are estimated with the BFGS (or L-BFGS) updating formula. To be more specific, given an approximation H_G to the inverse of a symmetric matrix G, the BFGS updating formula computes

$$H_G^+ = (I - \rho \mathbf{s}_G \mathbf{y}_G^\top) H (I - \rho \mathbf{y}_G \mathbf{s}_G^\top) + \rho \mathbf{s}_G \mathbf{s}_G^\top, \tag{3.11}$$

with given vectors $\mathbf{s}_G$, $\mathbf{y}_G$ and $\rho = \frac{1}{\mathbf{y}_G^\top \mathbf{s}_G}$. H_A is similarly computed with BFGS updating.

Lastly, the $(\mathbf{s}_G, \mathbf{g}_G)$ pairs used by H_G is derived from the definition of G as a Hessian matrix w.r.t. the output of the fully-connected layer. A double damping procedure with damping term λ_G

is proposed to deal with the non-convexity of G. For H_A, Chapter 2 keeps track of an estimation to A and generates $(\mathbf{s}_A, \mathbf{y}_A)$ pairs with a "Hessian-action" approach, i.e. letting $\mathbf{s}_A = A\mathbf{y}_A + \lambda_A \mathbf{s}_A$, where λ_A is the damping term.

3.3.2 What's New in Our K-BFGS Method?

In this part, we describe the generic improvements of our new methods beyond K-BFGS-20, as well as how to extend the methods to convolutional layers. The complete pseudo-code and other implementation details are described in Sec 3.4.

Generic Improvements beyond K-BFGS-20

Algorithm 8 $D_P D_{LM}$ (P stands for Powell's damping and LM stands for Levenberg-Marquardt damping)

Input: $\mathbf{s}$, $\mathbf{y}$; **Output:** $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$; **Given:** H, $0 < \mu_1 < 1$, $\mu_2 > 0$
if $\mathbf{s}^\top \mathbf{y} < \mu_1 \mathbf{y}^\top H \mathbf{y}$ **then**
    $\theta_1 = \frac{(1-\mu_1)\mathbf{y}^\top H \mathbf{y}}{\mathbf{y}^\top H \mathbf{y} - \mathbf{s}^\top \mathbf{y}}$
else
    $\theta_1 = 1$
end if
$\tilde{\mathbf{s}} = \theta_1 \mathbf{s} + (1 - \theta_1) H\mathbf{y}$ {Powell's damping on H}
$\tilde{\mathbf{y}} = \mathbf{y} + \mu_2 \tilde{\mathbf{s}}$ {Levenberg-Marquardt damping on H^{-1}}
return: $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$

Improvement #1: $D_P D_{LM}$. In the double damping procedure proposed in Chapter 2, the parameter μ_2 can only take values in $(0, 1]$, which restricts its interpretation as a Levenberg-Marquardt (LM) damping term.

We propose a new procedure $D_P D_{LM}$ (Algorithm 8), in which the parameter μ_2 ($= \lambda_G$) is more directly related to LM damping and can take any values in $(0, \infty)$. To see this connection, note that in Algorithm 8, after Powell's damping on H, $\tilde{\mathbf{s}}^\top \mathbf{y} \geq \mu_1 \mathbf{y}^\top H \mathbf{y} \geq 0$. Hence, $\tilde{\mathbf{s}}^\top \tilde{\mathbf{y}} \geq \mu_2 ||\tilde{\mathbf{s}}||^2$, which can be viewed as LM damping with a parameter of μ_2, since G is then lower bounded by $\mu_2 I$.

Improvement #2: "minibatched" Hessian-action BFGS. In approximating A^{-1}, Chapter 2 uses the so-called "Hessian-action" approach, in which they computes $A\mathbf{s}_A$ with an estimation of

A from a moving average scheme with a given hyper-parameter on decaying. In other words, one needs to compute A from each minibatch and always keep track of a moving average of it, which could be time consuming when A is large. The large size of A is particularly true for convolutional layers.

In this chapter, we propose a "minibatched" version of Hessian-action BFGS, i.e. computing $A\mathbf{s}$ with A estimated from only the current minibatch. By doing so, we avoid the explicit computation of A, replacing it with a direct matrix-vector product $A\mathbf{s}$. Moreover, the hyper-parameter on decaying is no longer needed, which saves some effort in hyper-parameter tuning. This improvement is applicable to both fully-connected and convolutional layers, the latter of which is described in Section 3.3.2.

Extension to Convolutional Layers

For convolutional layers, we similarly use (3.10) to update the parameters W defined in (3.2), where H_G and H_A correspond to approximations to the inverses of G and A defined in (3.9).

To approximate the inverse of G defined in (3.9), i.e. compting H_G, we use the BFGS updating formula (3.11), or L-BFGS. (We name our method K-BFGS and K-BFGS(L), respectively, when BFGS or L-BFGS is used for estimating H_G.) For a fixed data-point index n and $t \in \mathcal{T}$, the $(\mathbf{s}, \mathbf{y})$ pair for $G_{t,t}(n) = \frac{\partial^2 f(n)}{\partial \mathbf{h}_t(n)^2}$ is $(\mathbf{s}, \mathbf{y}) = (\mathbf{h}_t^+(n) - \mathbf{h}_t(n), \mathcal{D}\mathbf{h}_t^+(n) - \mathcal{D}\mathbf{h}_t(n))$, where the "+" sign denotes that we compute the value after a step of the parameters has been taken. Hence, for a fixed n, the $(\mathbf{s}, \mathbf{y})$ pair for $\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} G_{t,t}(n)$ is $\left(\overline{\mathbf{h}^+}(n) - \overline{\mathbf{h}}(n), \overline{\mathcal{D}\mathbf{h}^+}(n) - \overline{\mathcal{D}\mathbf{h}}(n)\right)$, where $\overline{X}$ denotes the value of X_t averaged over the spatial locations $\mathcal{T}$ for any quantity X, i.e., $\overline{X} := \frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} X_t$. Finally, the $(\mathbf{s}, \mathbf{y})$ for G in (3.9) is

$$\mathbf{s}_G = \mathbb{E}_n\left[\overline{\mathbf{h}^+}(n) - \overline{\mathbf{h}}(n)\right], \tag{3.12}$$

$$\mathbf{y}_G = \mathbb{E}_n\left[\overline{\mathcal{D}\mathbf{h}^+}(n) - \overline{\mathcal{D}\mathbf{h}}(n)\right]. \tag{3.13}$$

Combining the above with the $D_P D_{LM}$ approach described in Section 3.3.2, the final $(\mathbf{s}, \mathbf{y})$ pair

we use is $D_P D_{LM}(\mathbf{s}_G, \mathbf{y}_G)$.

To approximate the inverse of A defined in (3.9), i.e., computing H_A, we use the "minibatched" Hessian-action BFGS described in Section 3.3.2. Given the current estimate H_A of A^{-1}, the $(\mathbf{s}, \mathbf{y})$ pair for updating H_A are computed as:

$$\mathbf{s}_A = H_A \hat{\mathbf{a}}, \quad \mathbf{y}_A = A\mathbf{s}_A + \lambda_A \mathbf{s}_A, \tag{3.14}$$

where $\hat{\mathbf{a}} = \mathbb{E}_n[\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} \mathbf{a}_t(n)] = \mathbb{E}_n\left[\overline{\mathbf{a}(n)}\right]$ and λ_A is the damping term. Since A is estimated from a minibatch, we compute As_A without explicitly computing A. To be specific, by (3.3) and (3.9),

$$\begin{aligned} A\mathbf{s}_A &= \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} \mathbf{a}_t(n)\mathbf{a}_t(n)^\top\right]\mathbf{s}_A \\ &= \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} (\mathbf{a}_t(n)^\top \mathbf{s}_A)\mathbf{a}_t(n)\right]. \end{aligned} \tag{3.15}$$

Lastly, we propose to set the damping terms $\lambda_A = \sqrt{|\mathcal{T}|}\sqrt{\lambda}$, $\lambda_G = \frac{1}{\sqrt{|\mathcal{T}|}}\sqrt{\lambda}$ for a given overall damping hyper-parameter λ, which is shown to be better than setting $\lambda_A = \lambda_G = \sqrt{\lambda}$, which was proposed in Chapter 2 for fully-connected layers. (See Section 3.4.3 for more discussion on this.)

3.4 Pseudo-code and Algorithm Details for K-BFGS / K-BFGS(L)

3.4.1 Pseudo-code for K-BFGS / K-BFGS(L)

Algorithm 9 gives the pseudo-code for our proposed methods K-BFGS and K-BFGS(L). Note that one can use either BFGS or L-BFGS update for H_G, in which case we name the algorithm K-BFGS and K-BFGS(L), respectively. For simplicity, we assume that all layers in the model are convolutional layers. However, the algorithm can easily be adapted to fully-connected layers, hence applicable to MLP models or CNN models that contain fully-connected layers.

Algorithm 9 Pseudo-code for K-BFGS / K-BFGS(L)

Require: Given learning rates $\{\alpha_k\}$, damping value λ, curvature update frequency T, batch size m

$\mu_1 = 0.2, \beta = 0.9$

$\lambda_A^l = \sqrt{|\mathcal{T}^l|}\sqrt{\lambda}$, $\lambda_G^l = \frac{1}{\sqrt{|\mathcal{T}^l|}}\sqrt{\lambda}$ $(l = 1, ..., L)$ {$\mathcal{T}^l$ denotes the sets of spatial locations in layer l}

$\widetilde{\mathcal{D}W_l} = 0$, $A_l = \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} \mathbf{a}_t^l(n)\mathbf{a}_t^l(n)^\top\right]$, $H_A^l = (A_l + \lambda_A^l I_A)^{-1}$, $H_G^l = (\lambda_G^l)^{-1} I$, $\mathbf{s}_G^l = \mathbf{y}_G^l = 0$ $(l = 1, ..., L)$ {Initialization}

for $k = 1, 2, ...$ **do**

  Sample mini-batch M_k of size m

  Perform a forward-backward pass over M_k to compute stochastic gradient $\widetilde{\mathcal{D}W_l}$ $(l = 1, ..., L)$

  for $l = 1, ..., L$ **do**

    $\widetilde{\mathcal{D}W_l} = \beta\widetilde{\mathcal{D}W_l} + \widetilde{\mathcal{D}W_l}$

    $p_l = H_G^l \widetilde{\mathcal{D}W_l} H_A^l$ {if L-BFGS is used for H_G^l, it is initialized as $\lambda_G^{-1} I$}

    $W_l = W_l - \alpha_k p_l$

  end for

  if $k \equiv 0 \pmod{T}$ **then**

    Perform another forward-backward pass over M_k to compute $\overline{\mathbf{h}_l^+}$ and $\overline{\mathcal{D}\mathbf{h}_l^+}$ $(l = 1, \ldots, L)$

    for $l = 1, ..., L$ **do**

      {Update H_A^l by BFGS}

      $\mathbf{s}_A^l = H_A^l \widetilde{\overline{\mathbf{a}}_l}$, $\mathbf{y}_A^l = \widetilde{A}_l \mathbf{s}_A^l + \lambda_A^l \mathbf{s}_A^l$ using (3.15)

      Use BFGS updating (3.11) with $(\mathbf{s}_A^l, \mathbf{y}_A^l)$ to update H_A^l

      {Update H_G^l by BFGS or L-BFGS}

      $\mathbf{s}_G^l = \beta\mathbf{s}_G^l + (1-\beta)\left(\widetilde{\overline{\mathbf{h}_l^+}} - \widetilde{\overline{\mathbf{h}_l}}\right)$, $\mathbf{y}_G^l = \beta\mathbf{y}_G^l + (1-\beta)\left(\widetilde{\overline{\mathcal{D}\mathbf{h}_l^+}} - \widetilde{\overline{\mathcal{D}\mathbf{h}_l}}\right)$.

      $(\tilde{\mathbf{s}}_G^l, \tilde{\mathbf{y}}_G^l) = D_P D_{LM}(\mathbf{s}_G^l, \mathbf{y}_G^l)$ with $H = H_G^l$, $\mu_1 = \mu_1$, $\mu_2 = \lambda_G^l$ {See Algorithm 8}

      Use BFGS or L-BFGS with $(\tilde{\mathbf{s}}_G^l, \tilde{\mathbf{y}}_G^l)$ to update H_G^l {We name the algorithm K-BFGS or K-BFGS(L), respectively, when BFGS or L-BFGS is used.}

    end for

  end if

end for

3.4.2 Usage of Minibatches and Moving Averages

Because there is usually a large amount of data, we use minibatches to estimate the quantities needed at every iteration. We use $\widetilde{X}$ to denote the average value of X over a minibatch for any quantity X, which is usually used as an estimate to $\mathbb{E}_n[X(n)]$. Moreover, we use moving averages to both reduce the stochasticity and incorporate more information from the past:

- **Gradient.** At every iteration, the gradient $\widetilde{\mathcal{D}W}$ is estimated from a minibatch. We use a momentum scheme to get a better estimate $\widehat{\mathcal{D}W}$ of the gradient, i.e. we update

$$\widehat{\mathcal{D}W} = \beta\widehat{\mathcal{D}W} + \widetilde{\mathcal{D}W}.$$

- **BFGS updating for H_G.** By (3.12) and (3.13), we use both a minibatch and moving averages to estimate the $(\mathbf{s}, \mathbf{y})$ for H_G, i.e. we update

$$\begin{aligned}\mathbf{s}_G &= \beta\mathbf{s}_G + (1-\beta)\left(\widetilde{\overline{\mathbf{h}^+}} - \widetilde{\overline{\mathbf{h}}}\right),\\ \mathbf{y}_G &= \beta\mathbf{y}_G + (1-\beta)\left(\widetilde{\overline{\mathcal{D}\mathbf{h}^+}} - \widetilde{\overline{\mathcal{D}\mathbf{h}}}\right).\end{aligned}$$

- **BFGS updating for H_A.** In (3.14), we estimate the value of A from the current minibatch, i.e. $\widetilde{\sum_{t\in\mathcal{T}} A_{t,t}}$, as well as $\hat{\mathbf{a}} = \widetilde{\overline{\mathbf{a}}}$. Note that $A\mathbf{s}_A$ can be computed without forming A.

3.4.3 Other Details

Unlike Chapter 2, in which the whole K-BFGS process is performed at every iteration, we introduce a curvature update frequency T, controlling how frequently the algorithm updates its curvature matrices. In other words, when $k \not\equiv 0 \pmod{T}$, only from Line 5 to Line 11 of Algorithm 9 is incurred.

Note that Algorithm 9 contains only one damping hyper-parameter (HP) λ, and sets $\lambda_A^l = \sqrt{|\mathcal{T}^l|}\sqrt{\lambda}$, $\lambda_G^l = \frac{1}{\sqrt{|\mathcal{T}^l|}}\sqrt{\lambda}$ for each convolutional layer $l = 1, ..., L$, where $\mathcal{T}^l$ denotes the sets of spatial locations in layer l. This can be viewed as "rebalancing" A and G, i.e. setting A to be

Table 3.1: Storage Requirement of K-BFGS, K-BFGS(L), KFAC, and Adam

Algorithm	$\mathcal{D}W$	$\mathcal{D}W \odot \mathcal{D}W$	A / H_A	G / H_G	Total
K-BFGS	$O(IJ\|\Delta\|)$	—	$O(J^2\|\Delta\|^2)$	$O(I^2)$	$O(J^2\|\Delta\|^2 + IJ\|\Delta\| + I^2)$
K-BFGS(L)	$O(IJ\|\Delta\|)$	—	$O(J^2\|\Delta\|^2)$	$O(pI)$	$O(J^2\|\Delta\|^2 + IJ\|\Delta\| + pI)$
KFAC	$O(IJ\|\Delta\|)$	—	$O(J^2\|\Delta\|^2)$	$O(I^2)$	$O(J^2\|\Delta\|^2 + IJ\|\Delta\| + I^2)$
Adam	$O(IJ\|\Delta\|)$	$O(IJ\|\Delta\|)$	—	—	$O(IJ\|\Delta\|)$

$\mathbb{E}_n\left[\frac{1}{\sqrt{|\mathcal{T}|}}\sum_{t\in\mathcal{T}} A_{t,t}(n)\right]$ and G to be $\mathbb{E}_n\left[\frac{1}{\sqrt{|\mathcal{T}|}}\sum_{t\in\mathcal{T}} G_{t,t}(n)\right]$. For fully-connected layers (if there are any), we set $\lambda_A^l = \lambda_G^l = \sqrt{\lambda}$, as in Chapter 2. Since $\lambda_A^l\lambda_G^l = \lambda$, adding $\lambda_A^l\mathbf{s}_A^l$ and $\lambda_G^l\mathbf{s}_G^l$, respectively, to the vectors $\mathbf{y}_A^l$ and $\mathbf{y}_G^l$ before applying BFGS (or L-BFGS) to H_A^l and H_G^l, can be viewed as an approximation to adding the overall LM damping factor λI to $(H^l)^{-1} = (H_A^l)^{-1} \otimes (H_G^l)^{-1}$ prior to updating.

Moreover, a warm start computation of A_l is included, i.e. A_l is computed from the whole dataset before first iteration, which is then used to initialize H_A^l. This only introduces a mild overhead, as this warm start computation take no more than the time for one full epoch, and gives a good starting point for H_A^l.

When using the L-BFGS derived matrix H_G^l to compute $H_G^l\widehat{\mathcal{D}W_l}$, instead of using the classical two-loop recursion of L-BFGS, we follow the "non-loop" implementation in Byrd *et al.* [20], which is faster in practice because $\widehat{\mathcal{D}W_l}$ is a matrix, not a vector.

For CNN models that have batch normalization layers, we use the momentum gradient directions to update its parameters, with its own learning rate α_k/λ. (Note that α_k/λ can be viewed as the "effective" learning rate in K-BFGS, roughly speaking.)

In terms of default hyper-parameters, as shown in Algorithm 9, we set decay parameters $\beta = 0.9$, and $\mu_1 = 0.2$ in $D_P D_{LM}$. For K-BFGS(L), the number of $(\mathbf{s}, \mathbf{y})$ pairs stored for L-BFGS was set to be 100. These settings are exactly the same as in Chapter 2.

Table 3.2: Computation per iteration of K-BFGS, K-BFGS(L), KFAC, and Adam beyond that required for SGD

Algorithm	Additional pass	Curvature	Step ΔW_l
K-BFGS	$O\left(\frac{mIJ\|\Delta\|\|\mathcal{T}\|}{T}\right)$	$O\left(\frac{mJ\|\Delta\|\|\mathcal{T}\|+J^2\|\Delta\|^2+mI\|\mathcal{T}\|+I^2}{T}\right)$	$O(IJ^2\|\Delta\|^2 + I^2J\|\Delta\|)$
K-BFGS(L)	$O\left(\frac{mIJ\|\Delta\|\|\mathcal{T}\|}{T}\right)$	$O\left(\frac{mJ\|\Delta\|\|\mathcal{T}\|+J^2\|\Delta\|^2+mI\|\mathcal{T}\|+pI}{T}\right)$	$O(IJ^2\|\Delta\|^2 + pIJ\|\Delta\|)$
KFAC	$O\left(\frac{mIJ\|\Delta\|\|\mathcal{T}\|}{T_1}\right)$	$O\left(\frac{m(J^2\|\Delta\|^2+I^2)\|\mathcal{T}\|}{T_1} + \frac{J^3\|\Delta\|^3+I^3}{T_2}\right)$	$O(IJ^2\|\Delta\|^2 + I^2J\|\Delta\|)$
Adam	—	$O(IJ\|\Delta\|)$	$O(IJ\|\Delta\|)$

3.5 Space and Computational Requirements

In this section, we compare the space and computational requirements of the proposed K-BFGS and K-BFGS(L) methods with KFAC (see Algorithm 12) and Adam, which are among the predominant 2nd- and 1st-order methods, respectively, used to train CNNs. (One can also easily make similar comparison for MLPs.)

We focus on one convolutional layer, with J input channels, I output channels, kernel size $|\Delta|$, and $|\mathcal{T}|$ spacial locations. Moreover, let m denote the size of minibatches, p denote the number of $(\mathbf{s}, \mathbf{y})$ pairs for L-BFGS, T denote the curvature update frequency for K-BFGS/K-BFGS(L), and T_1 and T_2 denote the frequency of statistics update and inverse update for KFAC, respectively.

From Table 3.1, one can see that K-BFGS/K-BFGS(L) requires roughly the same amount of memory as KFAC. Note that I and J are usually much larger than $|\Delta|$ in CNNs. For example, in VGG16 [82], I and J can be as large as 512 whereas $|\Delta| = 9$. Hence, as Table 3.1 shows, the memory required by K-BFGS/K-BFGS(L) is of the same order as that of Adam in terms of I and J.

In Table 3.2, besides the operations listed, each algorithm also needs to compute the mini-batch gradient requiring $O(mIJ|\Delta||\mathcal{T}|)$ time. (Note that $|\mathcal{T}|$ is usually much larger than I or J.) Comparing with K-BFGS-20, K-BFGS improves time complexity due to the usage of "mini-batched" Hessian-action BFGS. If K-BFGS-20 were used (with original Hessian-action BFGS), the first term $O\left(\frac{mJ|\Delta||\mathcal{T}|}{T}\right)$ of the "Curvature" column for K-BFGS and K-BFGS(L) would increase

to $O\left(\frac{mJ^2|\Delta|^2|\mathcal{T}|}{T}\right)$. K-BFGS requires considerably less time to compute curvature information than KFAC. First, K-BFGS avoids matrix inversion, whose complexity is $O(I^3)$ and $O(J^3|\Delta|^3)$ (although this is amortized in KFAC by $\frac{1}{T_2}$ by using the same inverse for T_2 iterations). Second, K-BFGS avoids computing the Ω and Γ matrices of KFAC from minibatch data, whose complexity is $m(J^2|\Delta|^2 + I^2)|\mathcal{T}|$. Instead, we directly compute the $(\mathbf{s}, \mathbf{y})$ pairs for H_A and H_G, without explicitly forming A or G.

3.6 Convergence Results

Algorithm 10 K-BFGS(L) with $D_{P(I)}D_{LM}$ and exact inversion of A

Require: Given learning rates $\{\alpha_k\}$, damping values $\lambda_A^l, \lambda_G^l > 0$ $(l = 1, ..., L)$, batch size m, $0 < \mu_1 < 1, 0 < \beta < 1$

$A_l = \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} \mathbf{a}_t^l(n)\mathbf{a}_t^l(n)^\top\right]$, $H_A^l = (A_l + \lambda_A^l I_A)^{-1}$, $H_G^l = (\lambda_G^l)^{-1} I$, $\mathbf{s}_G^l = \mathbf{y}_G^l = 0$ $(l = 1, ..., L)$ {Initialization}

for $k = 1, 2, ...$ **do**

  Sample mini-batch M_k of size m

  Perform a forward-backward pass over M_k to compute stochastic gradient $\widetilde{\mathcal{D}W_l}$ $(l = 1, ..., L)$

  for $l = 1, ..., L$ **do**

    $p_l = H_G^l \widetilde{\mathcal{D}W_l} H_A^l$ $\{H_G^l$ is initialized as $\lambda_G^{-1} I$ in L-BFGS$\}$

    $W_l = W_l - \alpha_k p_l$

  end for

  Perform another forward-backward pass over M_k to compute $\overline{\mathbf{h}_l^+}$ and $\overline{\mathcal{D}\mathbf{h}_l^+}$ $(l = 1, \ldots, L)$

  for $l = 1, ..., L$ **do**

    {Compute H_A^l}

    Compute $\hat{A}_l = \widetilde{\sum_t A_{t,t}^l}$, $H_A^l = \left(\hat{A}_l + \lambda_A^l I\right)^{-1}$

    {Update H_G^l by L-BFGS}

    $\mathbf{s}_G^l = \beta \mathbf{s}_G^l + (1-\beta)\left(\widetilde{\overline{\mathbf{h}_l^+}} - \widetilde{\overline{\mathbf{h}_l}}\right)$, $\mathbf{y}_G^l = \beta \mathbf{y}_G^l + (1-\beta)\left(\widetilde{\overline{\mathcal{D}\mathbf{h}_l^+}} - \widetilde{\overline{\mathcal{D}\mathbf{h}_l}}\right)$.

    $(\tilde{\mathbf{s}}_G^l, \tilde{\mathbf{y}}_G^l) = D_{P(I)}D_{LM}(\mathbf{s}_G^l, \mathbf{y}_G^l)$ with $H = H_G^l$, $\mu_1 = \mu_1$, $\mu_2 = \lambda_G^l$ {See Algorithm 11}

    Use L-BFGS with $(\tilde{\mathbf{s}}_G^l, \tilde{\mathbf{y}}_G^l)$ to update H_G^l

  end for

end for

In this section, we present convergence results for a variant of K-BFGS(L) (specifically, Algorithm 10), following the framework in Wang *et al.* [87]. For the purpose of simplicity, we assume

Algorithm 11 $D_{P(I)}D_{LM}$

Input: $\mathbf{s}$, $\mathbf{y}$; **Output:** $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$; **Given:** $0 < \mu_1 < 1, \mu_2 > 0$
if $\mathbf{s}^\top\mathbf{y} < \mu_1\mathbf{y}^\top(\mu_2^{-1}I)\mathbf{y}$ **then**
    $\theta_1 = \frac{(1-\mu_1)\mathbf{y}^\top\mathbf{y}/\mu_2}{\mathbf{y}^\top\mathbf{y}/\mu_2 - \mathbf{s}^\top\mathbf{y}}$
else
    $\theta_1 = 1$
end if
$\tilde{\mathbf{s}} = \theta_1\mathbf{s} + (1-\theta_1)\mu_2^{-1}\mathbf{y}$ {Powell's damping with $H = \mu_2^{-1}I$}
$\tilde{\mathbf{y}} = \mathbf{y} + \mu_2\tilde{\mathbf{s}}$ {Levenberg-Marquardt damping on H^{-1}}
return: $\tilde{s}$, $\tilde{y}$

that all layers are convolutional. (Our results also hold for MLPs or if the model contains both convolutional and fully-connected layers.) For simplicity, we also assume that the curvature update frequency $T = 1$ in Algorithm 10. However, all the proofs and results still hold if $T > 1$.

There are several minor differences between Algorithm 10 and our actual implementation of K-BFGS(L), i.e. the one in Algorithm 9, including

1. $D_{P(I)}D_{LM}$ (Algorithm 11) is used instead of D_PD_{LM} (Algorithm 8). $D_{P(I)}D_{LM}$ differs from D_PD_{LM} by replacing H by a scaled identity matrix $\mu_2^{-1}I$, where it appears in Algorithm 8. This is justifiable partly because, in our actual implementation of L-BFGS, H_G is always initialized with the scaled identity matrix $\mu_2^{-1}I$ where $\mu_2 = \lambda_G^l$;

2. H_A^l is computed by simply inverting $\hat{A}_l + \lambda_A^l I$, instead of using minibatched Hessian-action BFGS;

3. Gradient is estimated from the current minibatch without momentum.

In particular, $D_{P(I)}D_{LM}$ (see Algorithm 11), rather than D_PD_{LM}, is used, which leads to the following lemma:

Lemma 4. *The output of $D_{P(I)}D_{LM}$ satisfies: $\frac{\tilde{\mathbf{s}}^\top\tilde{\mathbf{s}}}{\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}}} \leq \frac{1}{\mu_2}$, $\frac{\tilde{\mathbf{y}}^\top\tilde{\mathbf{y}}}{\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}}} \leq \frac{1}{\mu_3}$, where $\mu_3 = \frac{\mu_1}{\mu_2(1+2\mu_1)}$.*

Proof. First, similar to Powell's damping on H, we can show that $\tilde{\mathbf{s}}^\top\mathbf{y} \geq \frac{\mu_1}{\mu_2}\mathbf{y}^\top\mathbf{y} \geq 0$. Hence, $\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}} = \tilde{\mathbf{s}}^\top\mathbf{y} + \mu_2\tilde{\mathbf{s}}^\top\tilde{\mathbf{s}} \geq \mu_2\tilde{\mathbf{s}}^\top\tilde{\mathbf{s}}$.

To see the second inequality, by using that $\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}} \geq \tilde{\mathbf{s}}^\top\mathbf{y}$ it follows that

$$\begin{aligned}\tilde{\mathbf{y}}^\top\tilde{\mathbf{y}} &= \mathbf{y}^\top\mathbf{y} + 2\mu_2\tilde{\mathbf{s}}^\top\mathbf{y} + {\mu_2}^2\tilde{\mathbf{s}}^\top\tilde{\mathbf{s}} = \mathbf{y}^\top\mathbf{y} + 2\mu_2\tilde{\mathbf{s}}^\top(\tilde{\mathbf{y}} - \mu_2\mathbf{s}) + {\mu_2}^2\tilde{\mathbf{s}}^\top\tilde{\mathbf{s}} \\ &\leq \mathbf{y}^\top\mathbf{y} + 2\mu_2\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}} \leq \mu_2(\frac{1}{\mu_1} + 2)\tilde{\mathbf{s}}^\top\tilde{\mathbf{y}}.\end{aligned}$$

□

Consequently, one can prove the following two lemmas:

Lemma 5. *Suppose that we use* $(\mathbf{s}, \mathbf{y})$ *for the BFGS update (3.11). If* $\frac{\mathbf{s}^\top\mathbf{s}}{\mathbf{s}^\top\mathbf{y}} \leq \frac{1}{\mu_2}$, $\frac{\mathbf{y}^\top\mathbf{y}}{\mathbf{s}^\top\mathbf{y}} \leq \frac{1}{\mu_3}$, *then* $\|B^+\| \leq \|B\| + \frac{1}{\mu_3}$ *and* $\|H^+\| \leq (1 + \frac{1}{\sqrt{\mu_2\mu_3}})^2\|H\| + \frac{1}{\mu_2}$, *where* B *denotes the inverse of* H.

Proof. Corresponding to the BFGS update (3.11) of H, the update of B is

$$B^+ = B - \frac{B\mathbf{s}\mathbf{s}^\top B}{\mathbf{s}^\top Bs} + \rho\mathbf{y}\mathbf{y}^\top.$$

Hence,

$$\|B^+\| \leq \|B - \frac{B\mathbf{s}\mathbf{s}^\top B}{\mathbf{s}^\top B\mathbf{s}}\| + \|\rho\mathbf{y}\mathbf{y}^\top\| \leq \|B\| + \frac{\mathbf{y}^\top\mathbf{y}}{\mathbf{s}^\top\mathbf{y}} \leq \|B\| + \frac{1}{\mu_3}.$$

Also, using the fact that for the spectral norm $\|\cdot\|$, $\|I - \rho\mathbf{s}\mathbf{y}^\top\| = \|I - \rho\mathbf{y}\mathbf{s}^\top\|$, we have that

$$\|H^+\| \leq \|H\|\|I - \rho\mathbf{s}\mathbf{y}^\top\|^2 + \|\frac{\mathbf{s}\mathbf{s}^\top}{\mathbf{s}^\top\mathbf{y}}\| \leq \|H\|(\|I\| + \frac{\|\mathbf{s}\|\|\mathbf{y}\|}{\mathbf{s}^\top\mathbf{y}})^2 + \frac{\mathbf{s}^\top\mathbf{s}}{\mathbf{s}^\top\mathbf{y}} \leq (1 + \frac{1}{\sqrt{\mu_2}}\frac{1}{\sqrt{\mu_3}})^2\|H\| + \frac{1}{\mu_2}.$$

□

Lemma 6. *In Algorithm 10, for a given layer index* $l = 1, ..., L$, *there exist two positive constants* $\underline{\kappa}_G^l$ *and* $\bar{\kappa}_G^l$, *such that* $\underline{\kappa}_G^l I \preceq H_G^l(k) \preceq \bar{\kappa}_G^l I$, $\forall k$.

Proof. To simplify notation, we omit the subscript G, superscript l and the iteration index k in the proof. Hence, our goal is to prove $\underline{\kappa}_G I \preceq H = H_G^l(k) \preceq \bar{\kappa}_G I$, for any k. Let $(\mathbf{s}_i, \mathbf{y}_i)$ $(i = 1, ..., p)$

denote the pairs used in an L-BFGS computation of H.

Given an initial estimate $H_0 = B_0^{-1} = \lambda_G^{-1} I$ of $(G_l(\theta_k))^{-1}$, the L-BFGS method updates H_i recursively as

$$H_i = \left(I - \rho_i \mathbf{s}_i \mathbf{y}_i^\top\right) H_{i-1} \left(I - \rho_i \mathbf{y}_i \mathbf{s}_i^\top\right) + \rho_i \mathbf{s}_i \mathbf{s}_i^\top, \tag{3.16}$$

where $\rho_i = (\mathbf{s}_i^\top \mathbf{y}_i)^{-1}$, $i = 1, ..., p$, and equivalently,

$$B_i = B_{i-1} - \frac{B_{i-1}\mathbf{s}_i \mathbf{s}_i^\top B_{i-1}}{\mathbf{s}_i^\top B_{i-1} \mathbf{s}_i} + \rho_i \mathbf{y}_i \mathbf{y}_i^\top, \quad i = 1, \ldots, p,$$

where $B_i = H_i^{-1}$. Since we use $D_{P(I)} D_{LM}$, by Lemma 4, we have that $\frac{\mathbf{s}_i^\top \mathbf{s}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_2}$ and $\frac{\mathbf{y}_i^\top \mathbf{y}_i}{\mathbf{s}_i^\top \mathbf{y}_i} \leq \frac{1}{\mu_3}$.

Hence, by Lemma 5, we have that $||B_i|| \leq ||B_{i-1}|| + \frac{1}{\mu_3}$. Hence, $||B|| = ||B_p|| \leq ||B_0|| + \frac{p}{\mu_3} = \lambda_G + \frac{p}{\mu_3}$. Thus, $B \leq \left(\lambda_G + \frac{p}{\mu_3}\right) I$, and $H \geq \left(\lambda_G + \frac{p}{\mu_3}\right)^{-1} I \equiv \underline{\kappa}_G I$.

On the other hand, by Lemma 5, we have that $\|H_i\| \leq (1 + \frac{1}{\sqrt{\mu_2 \mu_3}})^2 \|H_{i-1}\| + \frac{1}{\mu_2}$. Hence, from the fact that $H_0 = \lambda_G^{-1} I$, and induction, we have that $||H|| \leq \lambda_G^{-1} \hat{\mu}^p + \frac{\hat{\mu}^p - 1}{\hat{\mu} - 1} \frac{1}{\mu_2} \equiv \bar{\kappa}_G$, where $\hat{\mu} = (1 + \frac{1}{\sqrt{\mu_2 \mu_3}})^2$.

□

To apply the convergence results in Wang *et al.* [87] to Algorithm 10, we need to have that H_A^l, and hence that $H_l = H_A^l \otimes H_G^l$, is bounded above and below by positive definite matrices, in addition to H_G^l. For this purpose and for satisfying other requirements needed to apply the theory in Wang *et al.* [87], we make the following assumptions:

Assumption 5. *$f : \mathbb{R}^d \to \mathbb{R}$ is continuously differentiable. $f(\theta)$ is lower bounded by a real number f^{low} for any $\theta \in \mathbb{R}^d$. ∇f is globally Lipschitz continuous with Lipschitz constant L, i.e., for any $\theta, \theta' \in \mathbb{R}^d$, $\|\nabla f(\theta) - \nabla f(\theta')\| \leq L\|\theta - \theta'\|$.*

Assumption 6. *For every iteration k, we have*

$$a)\ \mathbb{E}_{\xi_k}\left[g(\theta_k, \xi_k)\right] = \nabla f(\theta_k),$$
$$b)\ \mathbb{E}_{\xi_k}\left[\|g(x_k, \xi_k) - \nabla f(\theta_k)\|^2\right] \leq \sigma^2,$$

where g is the minibatch gradient and $\sigma > 0$ is the noise level of the gradient estimation, and $\xi_k, k = 1, 2, \ldots$ are independent samples, and for a given k the random variable ξ_k is independent of $\{\theta_j\}_{j=1}^k$.

Assumption 7. *The inputs $a^l_{j,t}$'s to any layers are bounded, i.e. $\exists \varphi > 0$ s.t. $\forall l, j, t, |a^l_{j,t}| \leq \varphi$.*

Note that AS. 7 is relatively mild, in the sense that it is fulfilled if the activation functions of the model are all bounded (e.g. sigmoid, tanh, binary step), or some appropriate "normalization" is performed before the data are fed into each layer.

We now show that our block-diagonal approximation to the Hessian is bounded below and above by positive definite matrices in Lemma 7, and after that, applying Theorem 2.8 in Wang *et al.* [87] we obtain our main convergence result, Theorem 4.

Lemma 7. *For Algorithm 10, under Assumption AS. 7, (i) $\hat{A}_l \leq (J_l|\Delta|\varphi^2 + 1)|\mathcal{T}^l|I$, $\forall l$, and (ii) there exist two positive constants $\underline{\kappa}$ and $\bar{\kappa}$, such that $\underline{\kappa} I \leq H = diag\{H_1, ..., H_L\} \leq \bar{\kappa} I$.*

Proof. In proving part (i), we omit the layer index l for simplicity. First, because $\hat{A}$ is the averaged value across the minibatch, it suffices to show that for any data-point n, $\sum_t A_{t,t}(n) \leq (J|\Delta|\varphi^2 + 1)|\mathcal{T}|I$.

By AS. 7, $||\mathbf{a}_t(n)||^2 = \sum_{j,\delta} a_{j,t+\delta}(n)^2 + 1 \leq J|\Delta|\varphi^2 + 1$. Hence, for any vector $\mathbf{x}$,

$$\mathbf{x}^\top A_{t,t}(n)\mathbf{x} = \mathbf{x}^\top(\mathbf{a}_t(n)\mathbf{a}_t(n)^\top)\mathbf{x} = (\mathbf{a}_t(n)^\top\mathbf{x})^2 \leq ||\mathbf{a}_t(n)||^2||\mathbf{x}||^2 \leq (J|\Delta|\varphi^2 + 1)||\mathbf{x}||^2.$$

Hence, $A_{t,t}(n) \leq (J|\Delta|\varphi^2 + 1)I$ and $\sum_t A_{t,t}(n) \leq (J|\Delta|\varphi^2 + 1)|\mathcal{T}|I$, proving part (i).

Note that $\hat{A}_l + \lambda^l_A I \geq \lambda^l_A I$ because $\hat{A}_l$ is PSD. On the other hand, $\hat{A}_l + \lambda^l_A I \leq ((J_l|\Delta|\varphi^2 +$

$1)|\mathcal{T}^l| + \lambda_A^l)I$. Hence,

$$((J_l|\Delta|\varphi^2 + 1)|\mathcal{T}^l| + \lambda_A^l)^{-1} I \preceq H_A^l = (\hat{A}_l + \lambda_A^l I)^{-1} \preceq (\lambda_A^l)^{-1} I. \tag{3.17}$$

By (3.17) and Lemma 6, we have that $\underline{\kappa}^l I \preceq H_A^l \otimes H_G^l = H_l \preceq \overline{\kappa}^l I$ where $\underline{\kappa}^l = ((J_l|\Delta|\varphi^2 + 1)|\mathcal{T}^l| + \lambda_A^l)^{-1}\underline{\kappa}_G^l$, $\overline{\kappa}^l = (\lambda_A^l)^{-1}\overline{\kappa}_G^l$. Finally, $\underline{\kappa} I \preceq H = \text{diag}\{H_1, ..., H_L\} \preceq \overline{\kappa} I$, where $\underline{\kappa} = \min\{\underline{\kappa}^1, ..., \underline{\kappa}^L\}$ and $\overline{\kappa} = \max\{\overline{\kappa}^1, ..., \overline{\kappa}^L\}$.

□

Theorem 4. *Suppose that Assumptions AS.5, AS.6, AS.7 hold for $\{\theta_k\}$ generated by Algorithm 10. We also assume that α_k is specifically chosen as $\alpha_k = \frac{\underline{\kappa}}{L\bar{\kappa}^2} k^{-\beta}$ with $\beta \in (0.5, 1)$. Then*

$$\frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[\|\nabla f(\theta_k)\|^2\right] \leq \frac{2L\left(M_f - f^{low}\right)\bar{\kappa}^2}{\underline{\kappa}^2} K^{\beta-1} + \frac{\sigma^2}{(1-\beta)m}(K^{-\beta} - K^{-1}),$$

where K denotes the iteration number and M_f is a positive constant. Moreover, for a given $\epsilon \in (0, 1)$, to guarantee that $\frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[\|\nabla f(\theta_k)\|^2\right] < \epsilon$, the number of iterations K needed is at most $O\left(\epsilon^{-\frac{1}{1-\beta}}\right)$.

Proof. First, Algorithm 10 falls in the general framework of the Stochastic Quasi-Newton (SQN) method (Algorithm 2.1) in Wang *et al.* [87]. Second, by Lemma 7, Assumption AS.3 in Wang *et al.* [87] is satisfied. Also, by the way H_A and H_G are updated, AS.4 in Wang *et al.* [87] is satisfied. Hence, since Assumptions AS.1 and AS.2 are identical to the other two assumptions made in Wang *et al.* [87], we are able to apply Theorem 2.8 in that paper to Algorithm 3 in this Section.

□

Theorem 4 shows that Algorithm 10 converges to a stationary point for a (possibly) non-convex function f. We note that under very similar assumptions, Theorems 2.5 and 2.6 in Wang *et al.* [87] also hold for Algorithm 10.

3.7 Numerical Results

In this section, we describe two sets of experiments, namely, on three MLP autoencoder problems and four CNN problems, comparing our proposed methods to other relevant methods (see Sec 3.7.2 for the detailed description of them) mentioned in this chapter.

The results reported in the tables and plots are all based on runs using 5 different random seeds and the tuned best hyper-parameters (HPs) from a grid search specified below. The values reported in the tables and the solid curves depicted in the plots are derived from the averages of the 5 runs, while the shaded areas in the plots depict the $\pm \text{std}/\sqrt{5}$ range for the runs. All experiments were run on a machine with 8 Xeon Gold 6248 CPUs with one Nvidia V100 GPU.

3.7.1 Major Numerical Experiments

Comparison with Chapter 2

Table 3.3: Average of training loss achieved by K-BFGS, K-BFGS(L), their counterpart in Chapter 2, KFAC, Adam, and SGD-m using 5 different random seed with best HP values

	MNIST	FACES	CURVES
K-BFGS	51.60	5.00	55.46
K-BFGS†[2]	52.01	4.62	55.06
K-BFGS(L)	51.53	4.83	55.31
K-BFGS-20	52.38	5.46	56.00
K-BFGS(L)-20	54.27	4.92	55.94
KFAC	51.24	4.71	55.12
Adam	52.76	5.33	55.24
SGD-m	54.75	6.47	55.97

[2]The dagger sign (†) denotes that the curvature update frequency $T = 20$

Our first set of experiments are on three MLP autoencoder problems with MNIST [51], FACES, and CURVES [43] datasets, that have become standard for testing the performance of algorithms to train DNNs. We tested our proposed K-BFGS and K-BFGS(L) methods, their counterpart in Chapter 2 (i.e., K-BFGS-20 and K-BFGS(L)-20), as well as three SOTA methods, SGD with momentum (SGD-m), Adam, and KFAC. See Section 3.7.2 for the model architecture and dataset details.

Minibatches of size 1000 were used for all three problems, as in Chapter 2. Each algorithm was run for a fixed amount of time for each problem (500 seconds for MNIST and CURVES, 2000 seconds for FACES). For all Kronecker-factored QN method, we set $T = 1$, except the one denoted as K-BFGS†, which used $T = 20$. For KFAC, we set $T_1 = 1$ and $T_2 = 20$. These settings are exactly the same as those in Chapter 2.

As we are primarily interested in optimization performance in these experiments, we conducted a grid search on two hyper-parameters (HPs), namely, learning rate and damping, for all methods (only learning rate for SGD-m), and selected the best HP values that achieved the smallest loss on the training set. (See Section 3.7.2 for the searching range and best HP values selected.) Finally, we ran each algorithm with their best HPs, using 5 different random seeds, and reported the average loss in Table 3.3. (See Figures 3.1, 3.2, and 3.3 for the training curves.)

From Table 3.3, we can clearly see that K-BFGS and K-BFGS(L) consistently outperformed their counterparts in Chapter 2, justifying the effectiveness of the generic improvements we incorporated in K-BFGS. (See Sec 3.7.3 for a more complete ablation study.) Moreover, K-BFGS and K-BFGS(L) also performed better than the first-order methods in most cases except for Adam on CURVES. Besides, our proposed methods performed similarly to KFAC, particularly when amortization was used (see K-BFGS†). Note that the KFAC baseline that we implemented was better than the one in Chapter 2, since it splits the overall damping term adaptively (see Line 19 of Algorithm 12), rather than simply setting $\pi_l = 1$, which turned out to be an important factor for KFAC.

Table 3.4: Average of validation classification accuracy (%) achieved by K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m using 5 different random seeds with best HP values on VGG16 and ResNet32

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
K-BFGS	94.28	93.45	75.65	71.46
K-BFGS(L)	94.24	93.38	75.43	70.98
KFAC	94.39	93.31	76.21	71.14
Adam	94.28	93.35	75.64	70.35
SGD-m	94.14	93.13	75.41	70.18

CNNs: Generalization Performance

We tested K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on two CNN models that have been found to be effective, namely, VGG16 [82] and ResNet32 [40]. We experimented on both models, using two datasets, CIFAR-10 and CIFAR-100 [48], each of which includes 50,000 training samples and 10,000 testing samples (we view them as the validation set in our experiments). For both datasets, we applied the data augmentation techniques in Krizhevsky *et al.* [49], including random horizontal flip and random crop. (See Sec 3.7.2 for more details about the experimental set-up.) The above model/dataset choices have been used and endorsed in many papers, e.g. Zhang *et al.* [96] and Choi *et al.* [23].

Minibatches of size 128 were used for all experiments. For SGD-m and Adam, we ran the algorithms for 200 epochs and decay the learning rate by a factor of 0.1 every 60 epochs, which has been shown to be an effective learning rate schedule for these 1st-order methods on these problems. For K-BFGS, K-BFGS(L), and KFAC, we ran the algorithms for 150 epochs and decay the learning rate by a factor of 0.1 every 50 epochs, so that their overall running time is approximately the same as that of the 1st-order methods. For K-BFGS, we set the curvature update frequency $T = 20$. For KFAC, we set $T_1 = 10$ and $T_2 = 100$, as in Zhang *et al.* [96].

As we are interested in generalization performance in these experiments, we incorporated the weight decay technique (see Sec 3.7.2) and conducted a grid search on three hyper-parameters (HPs), namely, initial learning rate, weight decay factor, and damping for all methods (only initial

learning rate and weight decay for SGD-m). Then, we selected HP values that achieved the largest classification accuracy on the validation set (the grid search ranges and the best HP values so determined, are listed in Sec 3.7.2), and reported the average classification accuracy on the validation sets in Table 3.4. See Figures 3.4, 3.5, 3.6, and 3.7 for the training and validation curves. In these figures, we plot training cross entropy loss (the upper row) and validation classification error (the lower row) against number of epochs (the left column) and process time (the right column).

Results in Table 3.4 indicate that K-BFGS clearly outperformed Adam and SGD-m in terms of generalization, with the exception that K-BFGS and Adam achieved the same accuracy on VGG16+CIFAR10. K-BFGS outperformed K-FAC on the two ResNet32 problems, and underperformed K-BFGS on the other two VGG16 problems. Lastly, K-BFGS(L) seemed to consistently underperform K-BFGS in the experiments that we conducted. As a result, we recommend prioritize using K-BFGS, rather than K-BFGS(L), to training CNN models in practice.

Finally, by comparing the process times reported in Figures 3.4, 3.5, 3.6, and 3.7, we can see that the per-iteration time of K-BFGS is only about 1/3 more than it is for the first-order methods. Moreover, the per-iteration time of K-BFGS (with $T = 20$) is roughly the same as KFAC (with $T_2 = 100$), which demonstrates the effectiveness of our QN approach.

3.7.2 Implementation Details of the Experiments

Specification on Comparing Algorithms

We describe the version of KFAC that we implemented in Algorithm 12. Note that Ω_l in KFAC is the same as A_l in K-BFGS. Similar to the pseudo-code of K-BFGS, we assume that all layers are convolutional. However, one can easily derive our KFAC implementation for fully-connected layers from Algorithm 12 and Martens and Grosse [62].

Note that KFC-pre in Grosse and Martens [38] differs from Algorithm 12 in the following ways:

- KFC-pre uses clipping for the approximated natural gradient direction p;

Algorithm 12 KFAC

Require: Given learning rates $\{\alpha_k\}$, damping value λ, batch size m, statistics update frequency T_1, inverse update frequency T_2

$\beta = 0.9$

$\widehat{\mathcal{D}W_l} = 0$, $\Omega_l = \mathbb{E}_n\left[\sum_{t\in\mathcal{T}} \mathbf{a}_t^l(n)\mathbf{a}_t^l(n)^\top\right]$, $\Gamma_l = \mathbb{E}_n\left[\overline{\mathcal{D}\mathbf{h}^l(n)(\mathcal{D}\mathbf{h}^l(n))^\top}\right]$ with y sampled from the predictive distribution ($l = 1, ..., L$) {Initialization}

for $k = 1, 2, \ldots$ **do**

  Sample minibatch M_k of size m

  Perform a forward-backward pass over the current minibatch M_k to compute $\widetilde{\mathcal{D}W_l}$ for $l = 1, ..., L$

  for $l = 1, 2, \ldots L$ **do**

    $\widehat{\mathcal{D}W_l} = \beta\widehat{\mathcal{D}W_l} + \widetilde{\mathcal{D}W_l}$

    $p_l = H_\Gamma^l \widehat{\mathcal{D}W_l} H_\Omega^l$

    $W_l = W_l - \alpha_k p_l$.

  end for

  if $k \equiv 0 \pmod{T_1}$ **then**

    Perform another pass over M_k with y sampled from the predictive distribution to compute $\mathcal{D}\mathbf{h}_t^l$ for $l = 1, ..., L$

    for $l = 1, 2, \ldots L$ **do**

      Update $\Omega_l = \beta \cdot \Omega_l + (1-\beta) \cdot \sum_{t\in\mathcal{T}} \widetilde{\mathbf{a}_t^l(\mathbf{a}_t^l)^\top}$, $\Gamma_l = \beta \cdot \Gamma_l + (1-\beta) \cdot \widetilde{\overline{\mathcal{D}\mathbf{h}_t^l(\mathcal{D}\mathbf{h}_t^l)^\top}}$

    end for

  end if

  if $k \equiv 0 \pmod{T_2}$ **then**

    for $l = 1, 2, \ldots L$ **do**

      Recompute $H_\Omega^l = (\Omega_l + \pi_l\sqrt{\lambda}I)^{-1}$, $H_\Gamma^l = (\Gamma_l + \frac{1}{\pi_l}\sqrt{\lambda}I)^{-1}$, where $\pi_l = \sqrt{\frac{\text{trace}(\Omega_l \otimes I)}{\text{trace}(I \otimes \Gamma_l)}}$

    end for

  end if

end for

- KFC-pre uses momentum for p;
- KFC-pre uses parameter averaging on θ.

All of these techniques can also be applied to K-BFGS. Since we are primarily interested in comparing different pre-conditioning matrices, we chose not to include such techniques in our implementation.

In our KFAC implementation, for the CNN problems that have batch normalization (BN) layers, we update the parameters of the BN layers with the gradient direction, along with the same learning rate α, as was done in Zhang *et al.* [96].

Also note that in Algorithm 12, a warm start computation of Ω_l and Γ_l is included, i.e. initial estimates of Ω_l and Γ_l are computed from the whole dataset before the first iteration. A similar warm start computation of A_l was also included in K-BFGS. Since these warm start computations take no more than the time for one full epoch, we did not include the times for warm starts in the figures.

Finally, Adam was implemented exactly as in Kingma and Ba [46], with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, as suggested in the paper. We view the hyper-parameter ϵ in Adam as the damping term, and tune it in our experiments (specified below).

In SGD with momentum, the momentum of gradient is computed as in K-BFGS (Algorithm 9) and KFAC (Algorithm 12). In other words, if g denotes the minibatch gradient, we update the momentum of graident by $\hat{g} = \beta\hat{g} + g$ with $\beta = 0.9$ at every iteration.

Details on the Autoencoder Experiments

Table 3.5: Best HP values (learning rate, damping) for Table 3.3 as well as Figures 3.1, 3.2, and 3.3

	MNIST	FACES	CURVES
K-BFGS	(0.03, 0.3)	(0.03, 1)	(0.03, 0.3)
K-BFGS†[3]	(0.3, 30)	(0.03, 3)	(0.3, 30)
K-BFGS(L)	(0.03, 0.3)	(0.03, 1)	(0.1, 3)
K-BFGS-20	(0.01, 0.1)	(0.01, 0.3)	(0.01, 0.03)
K-BFGS(L)-20	(0.003, 0.1)	(0.01, 0.3)	(0.003, 0.03)
KFAC	(0.03, 1)	(0.01, 0.1)	(0.3, 30)
Adam	(1e-4, 1e-4)	(1e-4, 1e-4)	(1e-3, 1e-3)
SGD-m	(0.003, -)	(0.001, -)	(0.003, -)

[3]The dagger sign (†) denotes that the curvature update frequency $T = 20$

Table 3.6: Model architectures for the MLP autoencoder problems

	Layer width
MNIST	[784, 1000, 500, 250, 30, 250, 500, 1000, 784]
FACES	[625, 2000, 1000, 500, 30, 500, 1000, 2000, 625]
CURVES	[784, 400, 200, 100, 50, 25, 6, 25, 50, 100, 200, 400, 784]

The autoencoder architectures are exactly the same as in Chapter 2. The only difference is that we did not include the regularization term $\frac{\eta}{2}||\theta||^2$, because we focus on optimization performance so it is better to avoid this compounding factor and it is hard to know how to set the value of η unless we include it in the HP tuning process. To be more specific, Table 3.6 describes the model architectures. For MNIST and CURVES datasets, we use binary cross entropy with sigmoid as the loss functions, whereas for FACES, we use mean squared error. The activation functions of the hidden layers are always ReLU, except that there is no activation for the very middle layer.

As in Chapter 2, we only use the training sets of the datasets, which contains 60k (MNIST[4]), 103.5k (FACES[5]), and 20k (CURVES[6]) training samples, respectively.

In order to obtain the results in Table 3.3, we first conducted a grid search for each algorithm based on the following ranges:

- learning rate: { 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, 0.01, 0.03, 0.1, 0.3, 1 };
- damping:
  - Kronecker-factored QN methods (i.e., λ in Algorithm 9) and KFAC (i.e., λ in Algorithm 12): { 0.03, 0.1, 0.3, 1, 3, 10, 30 };
  - Adam (i.e., the ϵ HP in Kingma and Ba [46]): { 1e-8, 1e-4, 1e-3, 0.01, 0.1 }.

We selected the HP values that achieves the smallest training loss (see Table 3.5). We then ran each algorithm with their corresponding best HP values and 5 different random seed, and reported

[4] `http://yann.lecun.com/exdb/mnist/`
[5] `http://www.cs.toronto.edu/~jmartens/newfaces_rot_single.mat`
[6] `http://www.cs.toronto.edu/~jmartens/digs3pts_1.mat`

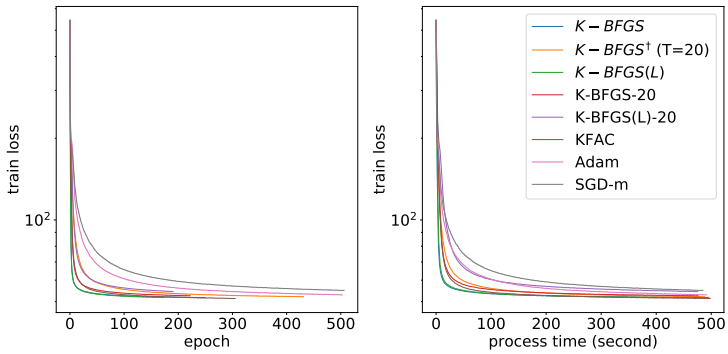


Figure 3.1: Optimization performance of K-BFGS, K-BFGS(L), their counterpart in Chapter 2, KFAC, Adam, and SGD-m on MNIST

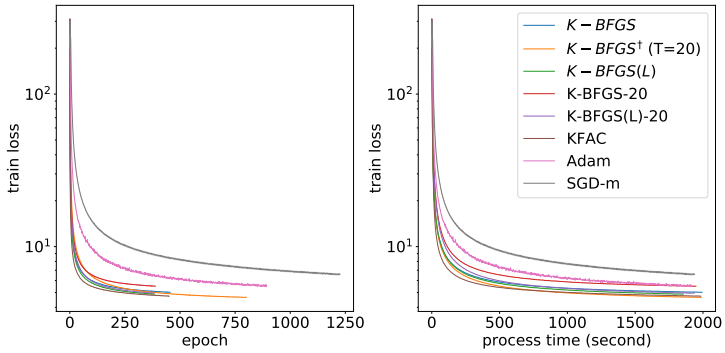


Figure 3.2: Optimization performance of K-BFGS, K-BFGS(L), their counterpart in Chapter 2, KFAC, Adam, and SGD-m on FACES

the average loss in Table 3.3. The training curves are also included in Figures 3.1, 3.2, and 3.3, where the training loss is reported against number of epochs (left) and process time (right).

Details on the CNN Experiments

The VGG16 model refers to the "model D" in Simonyan and Zisserman [82], with the modifications that the 3 fully-connected (FC) layers at the end of the model being replaced with only one FC layer (input size equal the size of the output size of the last conv layer, and output size equal number of classes of the dataset), and a batch normalization layer is added after each of the convolutional layers in the model. These changes are usually adopted nowadays on top of the original VGG models. The ResNet32 model refers to the one in Table 6 of He *et al.* [40].

For all the algorithms that we tested, we use the weight decay technique to help improve gen-

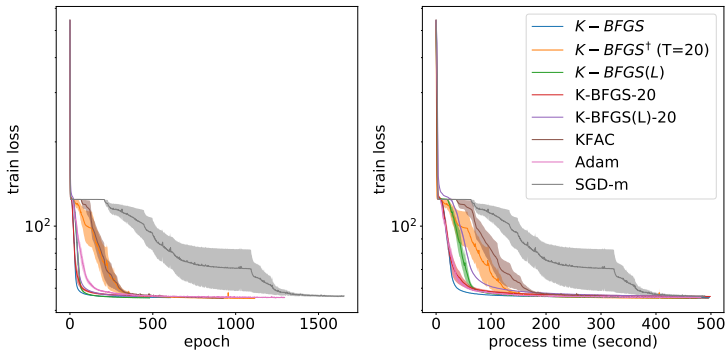


Figure 3.3: Optimization performance of K-BFGS, K-BFGS(L), their counterpart in Chapter 2, KFAC, Adam, and SGD-m on CURVES

eralization, which has shown to be effective for both 1st-order [55] and 2nd-order methods [96]. To be more specific, take K-BFGS/K-BFGS(L) (Algorithm 9) as an example, we replace Line 10 with $W_l = W_l - \alpha_k(p_l + \gamma W_l)$ where γ is the weigh decay factor. The same modification is done for SGD-m, Adam, and KFAC as well.

In order to obtain the results in Table 3.4, we first conducted a grid search for each algorithm based on the following ranges:

- K-BFGS and K-BFGS(L):
  - initial learning rate: { 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300, 1e3, 3e3 }
  - weight decay γ: { 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 0.01, 0.1 }
  - damping (i.e., λ in Algorithm 9): { 1, 10, 100, 1e3, 1e4, 1e5 }
- KFAC:
  - initial learning rate: { 1e-3, 3e-3, 0.01, 0.03, 0.1, 0.3 }
  - weight decay γ: { 0.001, 0.01, 0.1, 1 }
  - damping (i.e., λ in Algorithm 12): { 1e-4, 0.001, 0.01, 0.1, 1, 10, 100 }
- Adam:
  - initial learning rate: { 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, 0.01, 0.03, 0.1 }

Table 3.7: Best HP values (initial learning rate, weight decay, damping) for Table 3.4 as well as Figures 3.4, 3.5, 3.6, and 3.7

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
K-BFGS	(30, 1e-5, 1e4)	(100, 1e-5, 1e3)	(0.1, 0.01, 10)	(1e3, 1e-6, 1e4)
K-BFGS(L)	(1, 1e-3, 100)	(1e3, 1e-6, 1e4)	(0.3, 0.001, 100)	(10, 1e-4, 100)
KFAC	(3e-4, 1, 0.001)	(0.01, 0.1, 0.1)	(0.01, 0.1, 1)	(0.1, 0.01, 0.1)
Adam	(0.003, 0.1, 0.1)	(0.003, 0.1, 0.01)	(3e-4, 1, 0.01)	(0.01, 0.1, 0.01)
SGD-m	(0.003, 0.1, -)	(0.03, 0.01, -)	(0.003, 0.1, -)	(0.03, 0.01, -)

  - weight decay γ: { 0.01, 0.1, 1, 10 }
  - damping (i.e., the ϵ HP in Kingma and Ba [46]): { 1e-8, 1e-4, 0.01, 0.1, 1 }

- SGD-m:
  - initial learning rate: { 3e-4, 1e-3, 3e-3, 0.01, 0.03, 0.1, 0.3 }
  - weight decay γ: { 1e-3, 0.01, 0.1, 1 }

We selected the HP values that achieves the largest classification accuracy on the validation set (see Table 3.7). We then ran each algorithm with their corresponding best HP values and 5 different random seeds, and reported the average validation classification accuracy in Table 3.4. The training cross entropy loss (upper rows) and validation classification error (lower rows) against number of epochs (left columns) and process time (right columns) are also included in Figures 3.4, 3.5, 3.6, and 3.7.

From Table 3.7, we can see that, the optimal damping value for K-BFGS and K-BFGS(L) tends to be larger than that for KFAC, which is somewhat reasonable since they use quasi-Newton approaches to estimate curvature information. Hence, a stronger damping term (regularization) is needed. Moreover, in our experiments, for K-BFGS and K-BFGS(L), there was a strong positive correlation between the optimal learning rate and damping values, and a strong negative correlation between the optimal learning rate and weight decay values. These are not surprising because the

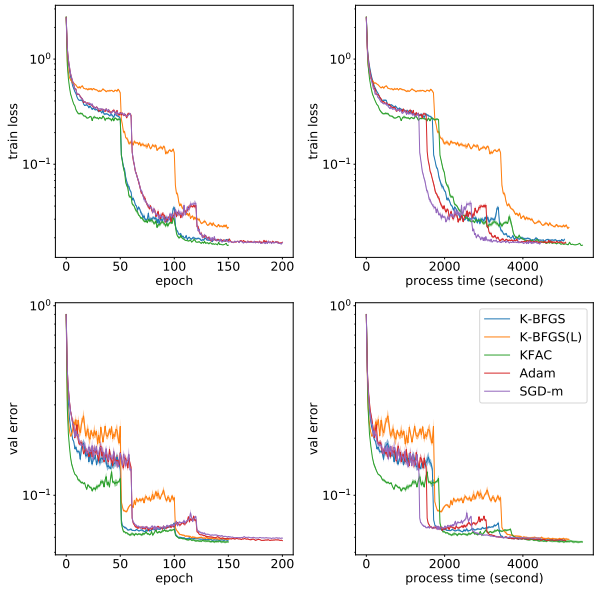


Figure 3.4: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on VGG16 with CIFAR10.

Table 3.8: Training loss of K-BFGS and K-BFGS-20 with two improvements turned on or off

Name of algorithm	Improvement #1	Improvement #2	MNIST	FACES	CURVES
K-BFGS	yes	yes	51.60	5.00	55.46
K-BFGS (#1 off)	no	yes	51.92	5.39	55.86
K-BFGS (#2 off)	yes	no	51.45	5.26	55.88
K-BFGS-20	no	no	52.38	5.46	56.00

"effective" learning rate involves the ratio of the learning rate to the damping, and the "effective" weight decay factor is the product of the weight decay value and the learning rate.

3.7.3 Additional Numerical Experiments

An Ablation Study

Besides the comparison presented in Section 3.7.1, we also conducted an ablation study on the two generic improvements we presented in Section 3.3.2. To be more specific, if both of the improvements are turned on, the algorithm is exactly the same as the one named "K-BFGS" in Table 3.3, whereas if both are turned off, it is the same as "K-BFGS-20" in Table 3.3.

We repeated the same MLP autoencoder experiments described in Section 3.7.1, and presented

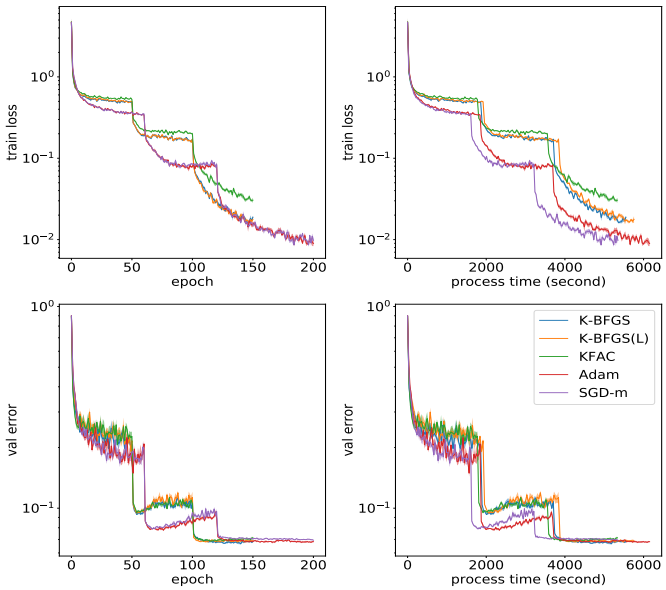


Figure 3.5: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on ResNet32 with CIFAR10

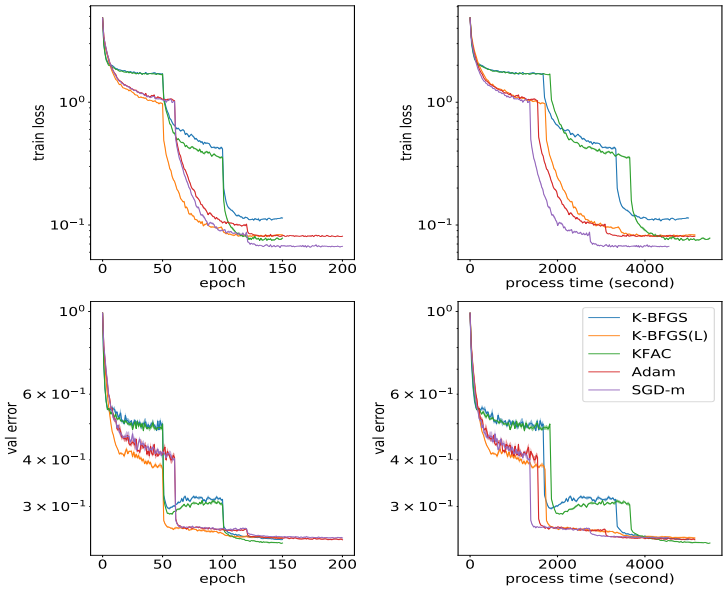


Figure 3.6: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on VGG16 with CIFAR100

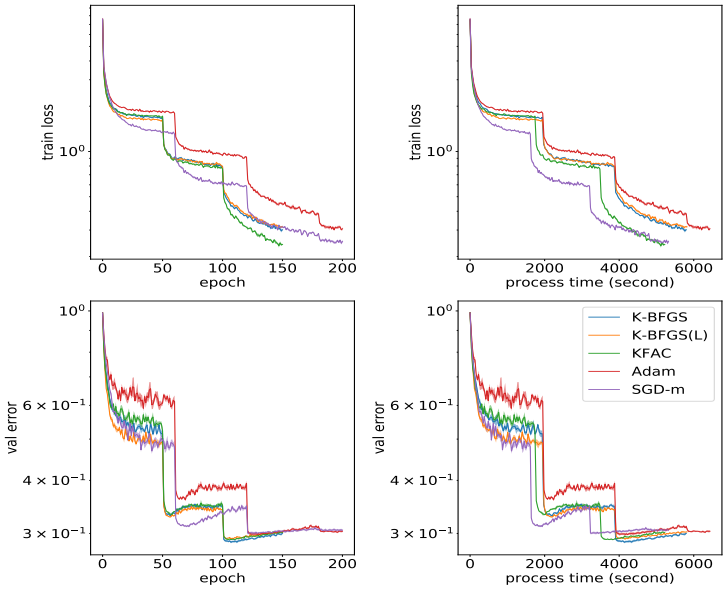


Figure 3.7: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on ResNet32 with CIFAR100

the results on four different variants in Table 3.8, where improvement #1 refers to the use of $D_P D_{LM}$, whereas improvement #2 refers to the use of "minibatched" Hessian-action BFGS, and the reported values are averaged across 5 different random seeds, using the best HP values for each algorithm. Table 3.8 shows that using each one of the improvements alone yields better results than the variant without improvements (i.e. K-BFGS-20), and using the two together (i.e. K-BFGS) usually yields the best results. This ablation study, along with the reasoning in Section 3.3.2, justifies the inclusion of the improvements we proposed.

Experiments on Wide ResNet

We also repeated the experiments on the ResNet32 model described in Section 3.7.1, except that we widened the width of the model by a factor of 4 (hence, we call this model ResNet32-4), which was proposed in Zagoruyko and Komodakis [94], and later adopted as a common variant of the original ResNet model (see e.g., Zhang *et al.* [96]).

We first conducted the same grid search as for the original ResNet32 model. The best HPs values are shown in Table 3.10. We then ran each algorithm with their corresponding best HPs

Table 3.9: Average of validation classification accuracy (%) achieved by K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m using 5 different random seeds with best HP values on ResNet32-4

Dataset Model	CIFAR10 ResNet32-4	CIFAR100 ResNet32-4
K-BFGS	95.61	79.41
K-BFGS(L)	95.74	79.18
KFAC	95.65	79.45
Adam	95.64	78.03
SGD-m	95.59	78.24

Table 3.10: Best HP values (initial learning rate, weight decay, damping) for Table 3.9 as well as Figures 3.8 and 3.9

	ResNet32-4, CIFAR10	ResNet32-4, CIFAR100
K-BFGS	(1e3, 1e-6, 1e4)	(100, 1e-5, 1e3)
K-BFGS(L)	(1e3, 1e-6, 1e4)	(1e3, 1e-6, 1e4)
KFAC	(0.1, 0.01, 1)	(0.01, 0.1, 0.01)
Adam	(0.03, 0.01, 0.1)	(0.003, 0.1, 0.01)
SGD-m	(0.03, 0.01, -)	(0.01, 0.1, -)

with 5 different random seeds and reported the average validation accuracy achieved in Table 3.9. We also plotted the average training and validation curves in Figures 3.8 and 3.9.

As shown in Table 3.9, all algorithms performed significantly better after the width of the layers were widened. Moreover, the performance of K-BFGS and K-BFGS(L) remains competitive, in particular for CIFAR-100, where K-BFGS and K-BFGS(L) achieved the best validation accuracy among all the comparing algorithms. Lastly, as shown in Figures 3.8 and 3.9, K-BFGS and K-BFGS(L) are able to scale up as well as the first-order methods in terms of per-iteration time complexity, when the model becomes larger.

3.8 Conclusion

In this chapter, we proposed a new class of Kronecker-factored quasi-Newton methods that are applicable to both MLP and CNN models, and that substantially improve upon the methods described in Chapter 2. We believe that our new methods are the first ones within the scope of

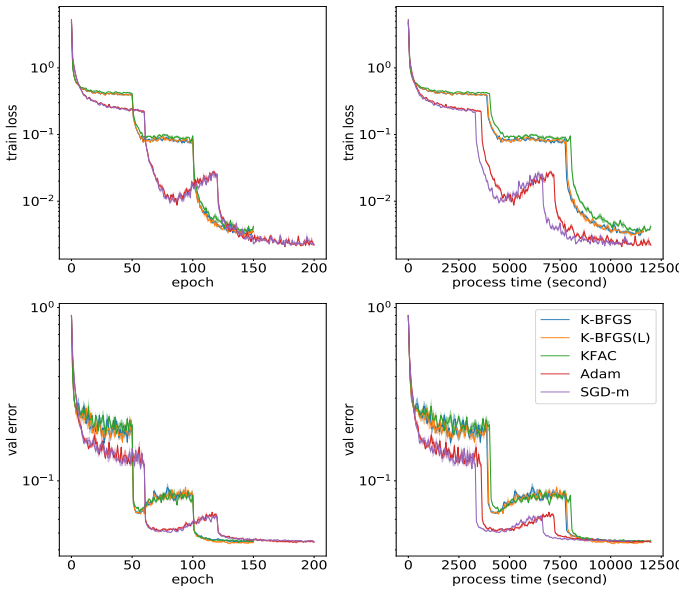


Figure 3.8: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on ResNet32-4 with CIFAR10

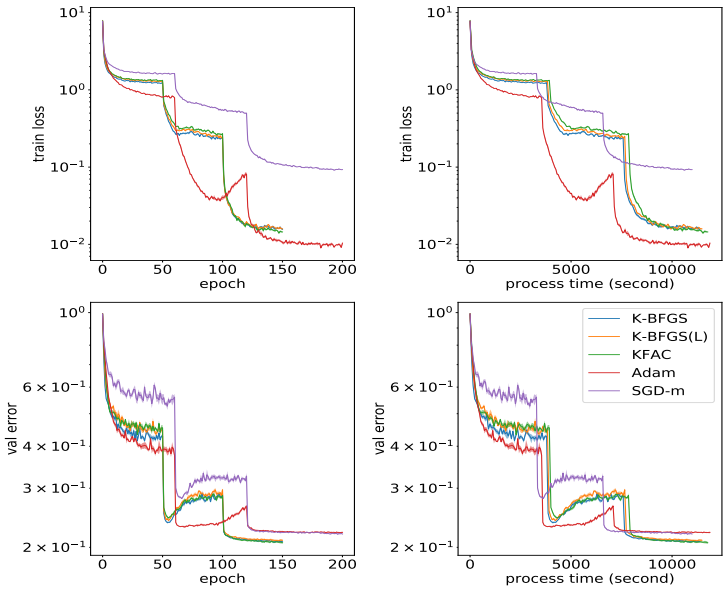


Figure 3.9: Generalization performance of K-BFGS, K-BFGS(L), KFAC, Adam, and SGD-m on ResNet32-4 with CIFAR100

quasi-Newton methods that use Kronecker-factored curvature approximations and are practical for training CNNs.

With extensive numerical experiments, our new methods are shown to be better than the ones in Chapter 2. On several standard CNN models, K-BFGS outperforms SOTA first-order methods and performs similarly to KFAC.

Chapter 4: Tensor Normal Training for Deep Learning Models

4.1 Introduction

Recently, there has been considerable advancement in the development of second-order methods that are suitable for deep learning models with huge numbers of parameters. These methods usually approach pre-conditioning of the gradient in a modular way, resulting in block-diagonal pre-conditioning matrices, where each block corresponds to a layer or a set of trainable parameters in the model. Inspired by the idea of the natural gradient (NG) method [2, 92, 3], Martens and Grosse [62] proposed KFAC, an NG method that uses a Kronecker-factored approximation to the Fisher matrix as its pre-conditioning matrix that can be applied to multilayer perceptrons, and which has subsequently been extended to other architectures, such as convolutional neural networks [38] and recurrent neural networks [61]. Kronecker-factored preconditioners (see Chapters 2 and 3) based on the structure of the Hessian and quasi-Newton methods have also been developed. Despite the great success of these efficient and effective second-order methods, developing such methods requires careful examination of the structure of the preconditioning matrix to design appropriate approximations for each type of layer in a model.

Another well-recognized second-order method, Shampoo [39, 5], extends the adaptive learning rate method AdaGrad, so that the gradient is pre-conditioned along every dimension of the underlying tensor of parameters in the model, essentially replacing the diagonal pre-conditioning matrix of the adaptive learning rate methods by a block diagonal Kronecker-factored matrix which can be viewed as an approximation to a fractional power of the empirical Fisher (EF) matrix. However, while estimating the Fisher matrix, in a deep learning setting, by the EF matrix saves some computational effort, it usually does not capture as much valuable curvature information as the Fisher matrix [50].

Variants of the normal distribution, i.e. the matrix-normal distribution [25] and the tensor-normal distribution [58], have been proposed to estimate the covariance of matrix and higher-order tensor observations, respectively. By imposing a Kronecker structure on the covariance matrix, the resulting covariance estimate requires a vastly reduced amount of memory, while still capturing the interactions between the various dimensions of the respective matrix or tensor. Iterative MLE methods for estimating the parameters of matrix-normal and tensor-normal distributions have been examined in e.g. Dutilleul [30] and Manceur and Dutilleul [58], and various ways to identify the unique representation of the distribution parameters have been proposed in Singull *et al.* [83] and Dees and Mandic [26]. However, to the best of our knowledge, this advanced statistical methodology has not been used to develop optimization methods for deep learning. In this chapter, we describe a first attempt to do this and demonstrate its great potential.

Our Contributions. In this chapter, we propose a new approximate natural gradient (NG) method, Tensor-Normal Training (TNT), that makes use of the tensor normal distribution to approximate the Fisher matrix. Significantly, the TNT method can be applied to any model whose training parameters are a collection of tensors without knowing the exact structure of the model.

To achieve this, we first propose a new way, that is suitable for optimization, to identify the covariance parameters of tensor normal (TN) distributions, in which the average eigenvalues of the covariance matrices corresponding to each of the tensor dimensions are required to be the same (see Section 4.3).

By using the Kronecker product structure of the TN covariance, TNT only introduces mild memory and per-iteration computational overhead compared with first-order methods. Also, TNT's memory usage is the same as Shampoo's and no greater than KFAC's, while its per-iteration computational needs are no greater than Shampoo's and KFAC's (see Section 4.6).

The effectiveness of TNT is demonstrated on deep learning models. Specifically, on standard autoencoder problems, when optimization performance is compared, TNT converges faster than the benchmark first-order methods and roughly the same rate as the benchmark second-order methods. Moreover, on standard CNN models, when generalization is concerned, TNT is able

to achieve roughly the same level of validation accuracy as the first-order methods, but using far fewer epochs (see Section 4.7).

We also prove that, if the statistics used in TNT can be estimated ideally, it converges to a stationary point under mild assumptions (see Section 4.4).

4.2 Preliminaries

Supervised Learning. Throughout this chapter, we consider the classic supervised learning setting where we learn the parameters θ of a model, by minimizing $\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f_\theta(x_i))$, where $\{(x_i, y_i)\}_{i=1}^{N}$ denotes a given dataset (x_i being the input to the model and y_i being the target), $f_\theta(x_i)$ denotes the output of the model when x_i is provided as the input, and l denotes a loss function (e.g. least-squares loss for regression and cross entropy loss for classification) that measures the discrepancy between the model output $f_\theta(x_i)$ and the target y_i.

Natural Gradient Method and the Fisher Matrix. In a first-order method, say SGD, the updating direction is always derived from an estimate to the gradient direction $\nabla_\theta \mathcal{L}(\theta)$. In a natural gradient (NG) method [3], however, the Fisher matrix is used as a pre-conditioning matrix that is applied to the gradient direction. As shown in Martens and Grosse [62], the Fisher matrix is defined as

$$F = \mathbb{E}_{x\sim Q_x, y\sim p(\cdot|x,\theta)}\left[\nabla_\theta \log p(y|x,\theta)\,(\nabla_\theta \log p(y|x,\theta))^\top\right], \tag{4.1}$$

where Q_x is the data distribution of x and $p(\cdot|x,\theta)$ is the density function of the conditional distribution defined by the model with a given input x.

In many cases, such as when p is associated with a Gaussian distribution and the loss function l measures least-squares loss, or when p is associated with a multinomial distribution and l is cross-entropy loss, $\log p$ is equivalent to l (see e.g. Martens [60] and Martens and Grosse [62]). Hence, if $\mathcal{D}\theta$ denotes the gradient of l w.r.t. θ for a given x and y, we have that $F = \mathbb{E}_{x\sim Q_x, y\sim p}[\mathcal{D}\theta\mathcal{D}\theta^\top]$. Consequently, one can sample x from Q_x and perform a forward pass of the model, then sample

y from $p(\cdot|x,\theta)$, and perform a backward pass to compute $\mathcal{D}\theta$, and then use $\mathcal{D}\theta\mathcal{D}\theta^\top$ to estimate F. We call $\mathcal{D}\theta$ a *sampling-based gradient*, as opposed to the *empirical gradient* $\nabla_\theta l(y_i, f_\theta(x_i))$ where (x_i, y_i) is one instance from the dataset.

It is worth noting that the first moment of $\mathcal{D}\theta$ is zero. To see this, note that, with given x,

$$\begin{aligned}&\mathbb{E}_{y\sim p}[\nabla_\theta \log p(y|x,\theta)] = \int \nabla_\theta \log p(y|x,\theta) p(y|x,\theta) dy = \int \nabla_\theta p(y|x,\theta) dy \\ =&\nabla_\theta \left(\int p(y|x,\theta) dy \right) = \nabla_\theta 1 = 0.\end{aligned}$$

Hence, $\mathbb{E}_{x\sim Q_x, y\sim p}[\mathcal{D}\theta] = \mathbb{E}_{x\sim Q_x}\left\{\mathbb{E}_{y\sim p}[\nabla_\theta \log p(y|x,\theta)] \mid x\right\} = 0$. Thus, the Fisher matrix F can be viewed as the covariance matrix of $\mathcal{D}\theta$. Note that the empirical Fisher CANNOT be viewed as the covariance of the empirical gradient, because the first moment of the latter is, in general, NOT zero.

Tensor-Normal Distribution. The development of our new method makes use the so-called *tensor-normal* distribution [58, 26]:

Definition 1. *An arbitrary tensor $G \in \mathbb{R}^{d_1\times\cdots\times d_k}$ is said to follow a tensor normal (TN) distribution with mean parameter $M \in \mathbb{R}^{d_1\times\cdots\times d_k}$ and covariance parameters $U_1 \in \mathbb{R}^{d_1\times d_1}$, ..., $U_k \in \mathbb{R}^{d_k\times d_k}$ if and only if $\overline{vec}(G) \sim Normal(\overline{vec}(M), U_1 \otimes \cdots \otimes U_k)$.*

In the above definition, the $\overline{\text{vec}}$ operation refers to the *vectorization* of a tensor, whose formal definition can be found in Sec 4.2.1. Note that *matrix-normal* distribution can be viewed as a special case of TN distribution, where $k = 2$. Compared with a regular normal distribution, whose covariance matrix has $\prod_{i=1}^k d_i^2$ elements, the covariance of a k-way tensor-normal distribution is stored in k smaller matrices with a total number of elements equal to $\sum_{i=1}^k d_i^2$.

To estimate the covariance submatrices $U_1, \ldots, U_k$, the following property (e.g., see Dees and Mandic [26]) is used:

$$\mathbb{E}[G^{(i)}] = U_i \cdot \prod_{j\neq i} \text{tr}(U_j), \tag{4.2}$$

where $G^{(i)} := \text{mat}_i(G)\text{mat}_i(G)^\top \in \mathbb{R}^{d_i \times d_i}$ denotes the *contraction* of G with itself along all but the ith dimension and mat_i refers to *matricization* of a tensor (see Section 4.2.1 for the formal definitions). By (4.2), we can sample G to obtain estimates of the $G^{(i)}$'s, and hence, estimates of the U_i's. The complexity of computing $G^{(i)}$ is $d_i \prod_{j=1}^{k} d_j$, which is also far less than the complexity of computing $\overline{\text{vec}}(G)\overline{\text{vec}}(G)^\top$ needed to estimate the covariance of a regular normal distribution.

4.2.1 Some Tensor Definitions and Properties

We present in this section fairly standard notation and definitions regarding tensors, e.g., see Gupta *et al.* [39] and Chapter 3 of Lu *et al.* [56], that we use throughout this chapter. Let $A \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ denote a tensor of order k.

- *slices* of A along its i-th dimension: for $i = 1, ..., k$ and $j = 1, ..., d_i$, the j-th slice of A along its i-th dimension, A^i_j denotes the $d_1 \times \cdots \times d_{i-1} \times d_{i+1} \times \cdots \times d_k$ tensor of order $k - 1$, composed from all of the entries of A whose ith index is j.

- *vectorization* of A: denoted as $\overline{\text{vec}}(A)$, is defined recursively as

$$\overline{\text{vec}}(A) = \begin{pmatrix} \overline{\text{vec}}(A^1_1) \\ \vdots \\ \overline{\text{vec}}(A^1_{d_1}) \end{pmatrix},$$

  where for the base case, in which A is one-dimensional tensor (i.e., a vector), $\overline{\text{vec}}(A) = A$. Note that when A is a matrix, this corresponds to the row-major vectorization of A.

- *matricization* of A: denoted as $\text{mat}_i(A)$, for $i = 1, ..., k$, is defined as

$$\text{mat}_i(A) = \begin{pmatrix} \overline{\text{vec}}(A^i_1)^\top \\ \vdots \\ \overline{\text{vec}}(A^i_{d_i})^\top \end{pmatrix}.$$

  Note that $\overline{\text{vec}}(A) = \overline{\text{vec}}(\text{mat}_1(A))$.

- *contraction* of A with itself along all but the ith dimension: denoted as $A^{(i)}$, is defined as $\text{mat}_i(A)\text{mat}_i(A)^\top$.

- *mode-i product* of A by a matrix $U \in \mathbb{R}^{d_i' \times d_i}$: the operation is denoted as $\times_i$. Let $B = A \times_i U \in \mathbb{R}^{d_1 \times \cdots \times d_{i-1} \times d_i' \times d_{i+1} \times \cdots \times d_k}$ denote the resulting tensor. $B_{j_1,\ldots,j_{i-1},j_i',j_{i+1},\ldots,j_k} = \sum_{j_i} A_{j_1,\ldots,j_k} U_{j_i',j_i}$. Note that in the matrix case ($k = 2$), $A \times_1 U = UA$, $A \times_2 U = AU^\top$.

Lemma 8. *Let $X \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times m}$, $B \in \mathbb{R}^{n \times n}$. Then, we have*

$$(A \otimes B^\top)\overline{vec}(X) = \overline{vec}(AXB).$$

Note that the above lemma is slightly different from the most common version of it, which uses a column-major vectorization of the matrix X.

Proposition 1. *Let $G \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ and $U_i \in \mathbb{R}^{d_i \times d_i}$ for $i = 1, \ldots, k$. Then, we have*

$$\left(\otimes_{i=1}^{k} U_i\right) \overline{vec}(G) = \overline{vec}(G \times_1 U_1 \times_2 U_2 \cdots \times_k U_k). \tag{4.3}$$

Proof of Proposition 1:

Proof. Our proof, which is largely inspired by the one in Gupta *et al.* [39], is by induction on k. When $k = 1$, it is easy to see that (4.3) holds by the definition of the mode-i product. When $k = 2$, (4.3) follows from Lemma 8.

Now assume that (4.3) holds for $1, 2, \ldots, k-1$. For k, we let $H = \otimes_{i=2}^{k} U_i$

By the induction hypothesis,

$$\mathrm{mat}_1(G)H^\top = \left(H\mathrm{mat}_1(G)^\top\right)^\top = \left(H\left(\overline{\mathrm{vec}}(G_1^1) \quad \cdots \quad \overline{\mathrm{vec}}(G_{d_1}^1)\right)\right)^\top \tag{4.4}$$

$$= \left(H\overline{\mathrm{vec}}(G_1^1) \quad \cdots \quad H\overline{\mathrm{vec}}(G_{d_1}^1)\right)^\top \tag{4.5}$$

$$= \left(\overline{\mathrm{vec}}(G_1^1 \times_1 U_2 \cdots \times_{k-1} U_k) \quad \cdots \quad \overline{\mathrm{vec}}(G_{d_1}^1 \times_1 U_2 \cdots \times_{k-1} U_k)\right)^\top \tag{4.6}$$

$$= \begin{pmatrix} \overline{\mathrm{vec}}(G_1^1 \times_1 U_2 \cdots \times_{k-1} U_k)^\top \\ \vdots \\ \overline{\mathrm{vec}}(G_{d_1}^1 \times_1 U_2 \cdots \times_{k-1} U_k)^\top \end{pmatrix} = \mathrm{mat}_1(G \times_2 U_2 \cdots \times_k U_k) \tag{4.7}$$

By Lemma 8 and (4.7),

$$\begin{aligned}
\left(\otimes_{i=1}^k U_i\right)\overline{\mathrm{vec}}(G) &= (U_1 \otimes H)\,\overline{\mathrm{vec}}(\mathrm{mat}_1(G)) = \overline{\mathrm{vec}}(U_1\mathrm{mat}_1(G)H^\top) \\
&= \overline{\mathrm{vec}}(U_1\mathrm{mat}_1(G \times_2 U_2 \cdots \times_k U_k)) \\
&= \overline{\mathrm{vec}}(\mathrm{mat}_1(G \times_2 U_2 \cdots \times_k U_k \times_1 U_1)) \\
&= \overline{\mathrm{vec}}(G \times_2 U_2 \cdots \times_k U_k \times_1 U_1) \\
&= \overline{\mathrm{vec}}(G \times_1 U_1 \times_2 U_2 \cdots \times_k U_k),
\end{aligned}$$

where the third from last equality comes from the fact that $B\mathrm{mat}_i(A) = \mathrm{mat}_i(A \times_i B)$, and the last equality comes from the fact that mode-i products are commutative.

□

4.3 Tensor-Normal Training

In this section, we propose Tensor-Normal Training (TNT), a new variant of the natural gradient (NG) method that makes use of the tensor-normal distribution.

4.3.1 Block Diagonal Approximation

In this chapter, we consider the case where the parameters of the model θ consists of multiple tensor variables $W_1, ..., W_L$, i.e. $\theta = (\overline{\text{vec}}(W_1)^\top, ..., \overline{\text{vec}}(W_L)^\top)^\top$. This setting is applicable to most common models in deep learning such as multi-layer perceptrons, convolutional neural networks, recurrent neural networks, etc. In these models, the trainable parameter W_l ($l = 1, \ldots, L$) come from the weights or biases of a layer, whether it be a feed-forward, convolutional, recurrent, or batch normalization layer, etc. Note that the index l of W_l refers to the index of a tensor variable, as opposed to a layer.

To obtain a practical NG method, we assume, as in KFAC and Shampoo, that the pre-conditioning Fisher matrix is block diagonal. To be more specific, we assume that each block corresponds to the covariance of a tensor variable in the model. Hence, the approximate Fisher matrix is:

$$F \approx \text{diag}_{l=1}^{L} \left\{ \mathbb{E}_{x\sim Q_x, y\sim p} \left[\overline{\text{vec}}(\mathcal{D}W_l)(\overline{\text{vec}}(\mathcal{D}W_l))^\top \right] \right\} = \text{diag}_{l=1}^{L} \left\{ \text{Var}(\overline{\text{vec}}(\mathcal{D}W_l)) \right\}.$$

The remaining question is how should one approximate $\text{Var}(\overline{\text{vec}}(\mathcal{D}W_l))$ for $l = 1, ..., L$.

4.3.2 Computing the Approximate Natural Gradient Direction by TNT

We consider a tensor variable $W \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ in the model and assume that $G := \mathcal{D}W \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ follows a TN distribution with zero mean and covariance parameters $U_1, ..., U_k$ where $U_i \in \mathbb{R}^{d_i \times d_i}$. Thus, the Fisher matrix corresponding to W is $F_W = \mathbb{E}_{x\sim Q_x, y\sim p}[\text{Var}(\overline{\text{vec}}(G))] = U_1 \otimes \cdots \otimes U_k$. Loosely speaking, the idea of relating the Fisher matrix to the covariance matrix of some normal distribution has some connections to Bayesian learning methods and interpretations of NG methods (see e.g., Khan and Rue [45]). Let $\nabla_W \mathcal{L} \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ denote the gradient of $\mathcal{L}$ w.r.t. W. The approximate NG updating direction for W is computed as

$$F_W^{-1} \overline{\text{vec}}(\nabla_W \mathcal{L}) = (U_1^{-1} \otimes \cdots \otimes U_k^{-1}) \overline{\text{vec}}(\nabla_W \mathcal{L}) = \overline{\text{vec}} \left(\nabla_W \mathcal{L} \times_1 U_1^{-1} \times_2 \cdots \times_k U_k^{-1} \right), \quad (4.8)$$

where $\times_i$ ($i = 1, ..., k$) denotes a mode-i product (see Section 4.2.1). Note that the last equality of (4.8) makes use of the following proposition, which also appears in Gupta *et al.* [39] (see Sec 4.2.1 for a proof):

Proposition 1. *Let $G \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ and $U_i \in \mathbb{R}^{d_i \times d_i}$ for $i = 1, ..., k$. Then, we have*

$$\left(\otimes_{i=1}^{k} U_i\right) \overline{vec}(G) = \overline{vec}(G \times_1 U_1 \times_2 U_2 \cdots \times_k U_k). \tag{4.9}$$

To summarize, the generic Tensor-Normal Training algorithm is described in Algorithm 13.

Algorithm 13 Generic Tensor-Normal Training (TNT)

Require: Given batch size m, and learning rate α

for $t = 1, 2, \ldots$ **do**
  Sample mini-batch M_t of size m
  Perform a forward-backward pass over M_t to compute the mini-batch gradient
  Perform another backward pass over M_t with y sampled from the predictive distribution to compute $G_l = \mathcal{D}W_l$ ($l = 1, ..., L$) averaged across M_t
  for $l = 1, ...L$ **do**
    Estimate $\mathbb{E}[G_l^{(i)}]$ ($i = 1, ..., k_l$) from G_l
    Determine $U_1^{(l)}, ..., U_{k_l}^{(l)}$ from $\mathbb{E}[G_l^{(1)}], ..., \mathbb{E}[G_l^{(k_l)}]$
    Compute the inverses of $U_1^{(l)}, ..., U_{k_l}^{(l)}$
    Compute the updating direction p_l by (4.8)
    $W_l = W_l - \alpha \cdot p_l$.
  end for
end for

4.3.3 Identifying the Covariance Parameters of the Tensor Normal Distribution

By (4.2), U_i can be inferred from $\mathbb{E}[G^{(i)}]$ up to a constant multiplier. However, different sets of multipliers can generate the same F, i.e. the same distribution. This is less of a problem if one only cares about F. However, we need F^{-1} to compute the approximate natural gradient. That is, we first must choose a representation of $F = c(\tilde{U}_1 \otimes \cdots \otimes \tilde{U}_k)$ (see below), and then compute $F^{-1} = c^{-1}((\tilde{U}_1 + \epsilon I)^{-1} \otimes \cdots \otimes (\tilde{U}_k + \epsilon I)^{-1})$ with a proper choice of $\epsilon > 0$, where ϵI plays a damping role in the preconditioning matrix. In this case, different representations of F will lead to different F^{-1}.

The statistics community has proposed various representations for $\tilde{U}_i$'s. For example, Singull *et al.* [83] imposed that $c = 1$ and the first element of $\tilde{U}_i$ to be one for $i = 1, ..., k-1$, whereas Dees and Mandic [26] imposed that $\text{tr}(\tilde{U}_i) = 1$ for $i = 1, ..., k$. Although these representations have nice statistical properties, they are not ideal from the perspective of inverting the covariance for use in a NG method in optimization.

We now describe one way to determine $\tilde{U}_1, ..., \tilde{U}_k$, and c from $\mathbb{E}[G^{(1)}], ..., \mathbb{E}[G^{(k)}]$. In particular, we first set $c = 1$, so that F^{-1} has a constant upper bound $\epsilon^{-k} I$. We then require that $\frac{\text{tr}(\tilde{U}_i)}{d_i}$ is constant w.r.t i. In other words, the average of the eigenvalues of each of the $\tilde{U}_i$'s is the same. This helps the $\tilde{U}_i$'s have similar overall "magnitude" so that a suitable ϵ can be found that works for all dimensions. Moreover, this shares some similarity with how KFAC splits the overall damping term between KFAC matrices, although KFAC adjusts the damping values, whereas TNT adjusts the matrices. A bit of algebra gives the formula

$$\tilde{U}_i = \frac{\mathbb{E}[G^{(i)}]}{c_0^{k-1} \prod_{j \neq i} d_j}, \tag{4.10}$$

where $c_0 = \left(\frac{\text{tr}(\mathbb{E}[G^{(i)}])}{\prod_j d_j} \right)^{1/k}$.

4.3.4 Comparison with Shampoo and KFAC

Shampoo, proposed in Gupta *et al.* [39], and later modified and extended in Anil *et al.* [5], is closely related to TNT. Both methods use a block-diagonal Kronecker-factored preconditioner based on second-order statistics of the gradient and are able to handle all sorts of tensors, and hence, can be applied to all sorts of deep neural network models, easily and seamlessly. The major differences between them are:

(i) The TN distribution cannot be directly applied to EF, which is used in Shampoo, because the empirical gradient does not have a zero mean; hence its covariance and second moment are different. It is also believed that EF does not capture as much valuable curvature information as Fisher [50].

(ii) Using the statistics $\mathbb{E}[G^{(i)}]$'s, TNT approximates the Fisher matrix as the covariance of the block-wise sampling-based gradients assuming that they are TN distributed. On the other hand, Shampoo computes $1/2k$-th power of the statistics of each direction of the tensor-structured empirical gradient and forms a preconditioning matrix from the Kronecker product of them. It is unclear to us how to interpret statistically such a matrix other than by its connection to EF. We further note that Shampoo was developed as a Kronecker-factored approximation to the full-matrix version of AdaGrad [29], whereas TNT was developed as a NG method using a TN-distributed approximation to the Fisher matrix.

(iii) TNT computes the updating direction using the inverse (i.e. power of -1) of the Kronecker factors of the approximate Fisher matrix, whereas Shampoo uses the $-1/2k$-th power[1] of the Kronecker factors of the EF matrix.

Another method related to TNT is KFAC [62, 38], which, like TNT, uses Fisher as its preconditioning matrix. Their major differences are:

(i) KFAC develops its approximation based on the structure of the gradient and Fisher matrix for each type of layer. Admittedly, this could lead to better approximations. But it is relatively hard to implement (e.g. one need to store some intermediate variables to construct the KFAC matrices). Also, if new types of layers with different structures are considered, one needs to develop suitable Kronecker factorizations, i.e., KFAC matrices. On the contrary, TNT, like Shampoo, is a model-agnostic method, in the sense that, TNT can be directly applied as long as the shape of the tensor variables are specified.

(ii) Each block of TNT corresponds to a tensor variable whose shape needs to be specified, whereas each block of KFAC corresponds to all variables in a layer. For example, for a linear or convolutional layer, the KFAC block would correspond to the Fisher of both its weights and bias (and their correlation), whereas TNT would produce two blocks corresponding to the weights and bias, respectively.

In order to gain more insight into how well TNT approximates the Fisher matrix compared

[1] In Anil *et al.* [5], for autoencoder problems involving tensors of order 2, the power was set to $-\frac{\alpha}{2}$, where $\alpha \in [0, 1]$ was treated as a hyper-parameter which required tuning, and was set to $\alpha = 1$ after tuning.

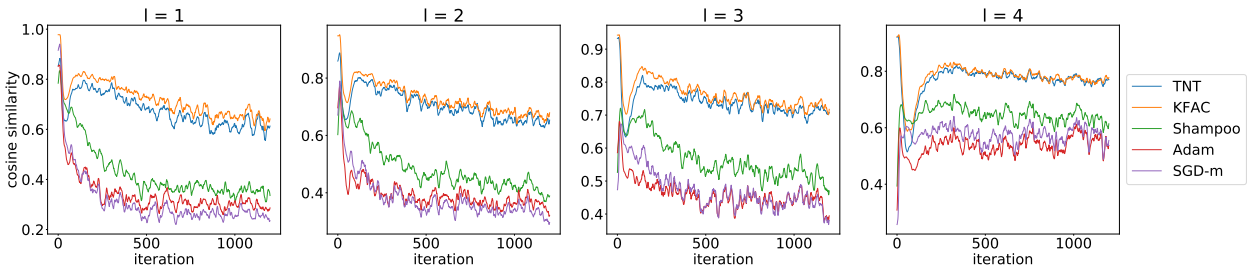


Figure 4.1: Cosine similarity between the directions produced by TNT, KFAC, Shampoo, Adam, and SGD-m and that of a block Fisher method

with other methods, we computed the cosine similarity between the direction produced by each method and that by a block Fisher method, where each block corresponded to one layer's full Fisher matrix in the model (see Figure 4.1). The algorithms were run on a 16×16 down-scaled MNIST [52] dataset and a small feed-forward NN with layer widths 256-20-20-20-20-20-10 described in Martens and Grosse [62]. As in Martens and Grosse [62], we only show the middle four layers. For all methods shown in Figure 4.1, we always followed the trajectory produced by the block Fisher method. In our implementation of the block Fisher method, both the gradient and the block-Fisher matrices were estimated with a moving-average scheme, with the decay factors being 0.9. In all of the other methods compared to the block Fisher method, moving averages were also used, with the decay factors being 0.9, as described in Section 4.7.2, to compute the relevant gradients and approximate block-Fisher matrices used by them, based on values computed at points generated by the block-Fisher method.

As shown in Figure 4.1, the cosine similarity for TNT is always around 0.6 to 0.8, which is similar to than KFAC, which is quite significant considering the fact that KFAC is aware of the structure of the Fisher matrix. On the other hand, the cosine similarity for TNT is always much better than that of Shampoo, which could attribute to the fact that TNT approximates Fisher whereas Shampoo approximates empirical Fisher, or that TNT and Shampoo use different ways to construct their approximations. To provide more information, we also include SGD with momentum and Adam, whose similarity to the block Fisher direction is usually lower that of the second-order methods.

4.4 Convergence

Algorithm 14 Idealized Version of TNT

Require: Given $\theta_1 \in \mathbb{R}^n$, batch sizes $\{m_t\}_{t\geq 1}$, step sizes $\{\alpha_t\}_{t\geq 1}$, and damping value $\epsilon > 0$

1: **for** $t = 1, 2, \ldots$ **do**
2: Sample mini-batch of size m_t: $M_t = \{\xi_{t,i}, i = 1, \ldots, m_t\}$
3: Calculate $\widehat{\nabla \mathcal{L}}_t = \frac{1}{m_t} \sum_{\xi_{t,i} \in M_t} \nabla l(\theta_t, \xi_{t,i})$
4: Compute $\tilde{U}_i$ $(i = 1, ..., k)$ by formula (4.10), using the **true values** of $\mathbb{E}_{x\sim Q_x, y\sim p}[G^{(i)}]$ $(i = 1, ..., k)$ at the current parameter θ_t.
5: Compute $p_t = \overline{\text{vec}}\left(\widehat{\nabla \mathcal{L}}_t \times_1 (\tilde{U}_1 + \epsilon I)^{-1} \times_2 \cdots \times_k (\tilde{U}_k + \epsilon I)^{-1}\right)$
6: Calculate $\theta_{t+1} = \theta_t - \alpha_t p_t$
7: **end for**

In this section, we present results on the convergence of an idealized version of TNT that uses the actual covariance of $\mathcal{D}\theta$, rather than a statistical estimate of it (see Algorithm 14). In particular, our results show that Algorithm 14, with constant batch size and decreasing step size, converges to a stationary point under some mild assumptions. For simplicity, we assume that the model only contains one tensor variable W. However, our results can be easily extended to the case of multiple tensor variables. To start with, our proofs require the following assumptions:

Assumption 8. *$\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable. $\mathcal{L}(\theta)$ is lower bounded by a real number $\mathcal{L}^{low}$ for any $\theta \in \mathbb{R}^n$. $\nabla \mathcal{L}$ is globally Lipschitz continuous with Lipschitz constant L; namely for any $\theta, \theta' \in \mathbb{R}^n$, $\|\nabla \mathcal{L}(\theta) - \nabla \mathcal{L}(\theta')\| \leq L\|\theta - \theta'\|$.*

Assumption 9. *For any iteration t, we have*

$$a)\ \ \mathbb{E}_{\xi_t}\left[\nabla l(\theta_t, \xi_t)\right] = \nabla \mathcal{L}(\theta_t) \qquad b)\ \ \mathbb{E}_{\xi_t}\left[\|\nabla l(\theta_t, \xi_t) - \nabla \mathcal{L}(\theta_t)\|^2\right] \leq \sigma^2$$

where $\sigma > 0$ is the noise level of the gradient estimation, and $\xi_t, t = 1, 2, \ldots,$ are independent samples, and for a given t the random variable ξ_t is independent of $\{\theta_j\}_{j=1}^t$

Assumption 10. *Let $G := \mathcal{D}\theta$. For any $\theta \in \mathbb{R}^n$, the norm of the Fisher matrix*

$$F = \mathbb{E}_{x\sim Q_x, y\sim p}[\overline{vec}(G)\overline{vec}(G)^\top]$$

is bounded above.

Since F represents the curvature of the KL divergence of the model's predictive distribution, Assumption 10 controls the change of predictive distribution when the model's parameters change; hence, it is a mild assumption for reasonable deep learning models. Essentially, we would like to prove that, if the Fisher matrix is upper bounded, our approximated Fisher (by TNT) is also upper bounded.

We now present two lemmas and our main theorem.

Lemma 9. $\|\mathbb{E}_{x\sim Q_x, y\sim p}[G^{(i)}]\| \leq \left(\frac{1}{d_i}\prod_{i'=1}^{k} d_{i'}\right)\|\mathbb{E}_{x\sim Q_x, y\sim p}[\overline{vec}(G)\overline{vec}(G)^\top]\|,\ \forall\, i = 1, \ldots, k.$

Proof. Let $X \in \mathbb{R}^{m\times n}$ be a random matrix, and $x_i \in \mathbb{R}^m$ denote its ith column ($i = 1, ..., n$). Because $\overline{\text{vec}}(X)$ is a vector containing all the elements of all the x_i's, $x_i x_i^\top$ is a square submatrix of $\overline{\text{vec}}(X)\overline{\text{vec}}(X)^\top$. Hence, $\|\mathbb{E}[x_i x_i^\top]\| \leq \|\mathbb{E}[\overline{\text{vec}}(X)\overline{\text{vec}}(X)^\top]\|$, and we have that

$$\|\mathbb{E}[XX^\top]\| = \|\mathbb{E}[\sum_{i=1}^{n} x_i x_i^\top]\| = \|\sum_{i=1}^{n}\mathbb{E}[x_i x_i^\top]\| \leq \sum_{i=1}^{n}\|\mathbb{E}[x_i x_i^\top]\| \leq n\|\mathbb{E}[\overline{\text{vec}}(X)\overline{\text{vec}}(X)^\top]\|.$$

Letting $X = \text{mat}_i(G) \in \mathbb{R}^{d_i\times(d_1\cdots d_{i-1}d_{i+1}\cdots d_k)}$, it then follows that

$$\begin{aligned}\|\mathbb{E}_{x\sim Q_x, y\sim p}[G^{(i)}]\| &\leq (d_1\cdots d_{i-1}d_{i+1}\cdots d_k)\|\mathbb{E}[\overline{\text{vec}}(\text{mat}_i(G))\overline{\text{vec}}(\text{mat}_i(G))^\top]\| \\ &= (\frac{1}{d_i}\prod_{i'=1}^{k} d_i')\|\mathbb{E}[\overline{\text{vec}}(G)\overline{\text{vec}}(G)^\top]\|.\end{aligned}$$

□

Lemma 10. *Suppose Assumption 10 holds. Let $F_{TNT} := (\tilde{U}_1 + \epsilon I)\otimes\cdots\otimes(\tilde{U}_k + \epsilon I)$ where $\tilde{U}_i$'s are defined in (4.10). Then, the norm of F_{TNT} is bounded both above and below.*

Proof. It is clear that $||F_{\text{TNT}}|| = \prod_{i=1}^{k}||\tilde{U}_i + \epsilon I|| \geq \epsilon^k$. On the other hand, for $i = 1, ..., k$, if we

denote the eigenvalues of $\mathbb{E}[G^{(i)}]$ by $\lambda_1 \leq \cdots \leq \lambda_{d_i}$, we have from (4.10) that

$$\begin{aligned}\|\tilde{U}_i\| &= \frac{\|\mathbb{E}[G^{(i)}]\|}{\left(\frac{\mathrm{tr}(\mathbb{E}[G^{(i)}])}{\prod_j d_j}\right)^{(k-1)/k} \prod_{j\neq i} d_j} = \frac{\lambda_{d_i}}{\left(\frac{\lambda_1+\cdots+\lambda_{d_i}}{\prod_j d_j}\right)^{(k-1)/k} \prod_{j\neq i} d_j} \\ &\leq \frac{\lambda_{d_i}}{\left(\frac{\lambda_{d_i}}{\prod_j d_j}\right)^{(k-1)/k} \prod_{j\neq i} d_j} = \frac{d_i \lambda_{d_i}^{1/k}}{(\prod_j d_j)^{1/k}} = \frac{d_i \|\mathbb{E}[G^{(i)}]\|^{1/k}}{(\prod_j d_j)^{1/k}}.\end{aligned}$$

Thus, since $||F_{\mathrm{TNT}}|| = \prod_{i=1}^k ||\tilde{U}_i + \epsilon I|| = \prod_{i=1}^k (||\tilde{U}_i|| + \epsilon)$, by the above and Lemma 9,

$$||F_{\mathrm{TNT}}|| \leq \prod_{i=1}^k \left(\frac{d_i \|\mathbb{E}[G^{(i)}]\|^{1/k}}{(\prod_j d_j)^{1/k}} + \epsilon \right) \leq \prod_{i=1}^k \left(d_i^{1-1/k} ||\mathbb{E}[\overline{\mathrm{vec}}(G)\overline{\mathrm{vec}}(G)^\top]||^{1/k} + \epsilon \right).$$

Then, by Assumption 10, we have that $||F_{\mathrm{TNT}}||$ is bounded above.

□

Theorem 5. *Suppose that Assumptions 8, 9, and 10 hold for $\{\theta_t\}$ generated by Algorithm 14 with batch size $m_t = m$ for all t. If we choose $\alpha_t = \frac{\kappa}{L\bar{\kappa}^2} t^{-\beta}$, with $\beta \in (0.5, 1)$, then*

$$\frac{1}{N}\sum_{t=1}^N \mathbb{E}_{\{\xi_j\}_{j=1}^\infty}\left[\|\nabla \mathcal{L}(\theta_t)\|^2\right] \leq \frac{2L\left(M_{\mathcal{L}} - \mathcal{L}^{low}\right)\bar{\kappa}^2}{\underline{\kappa}^2} N^{\beta-1} + \frac{\sigma^2}{(1-\beta)m}\left(N^{-\beta} - N^{-1}\right),$$

where N denotes the iteration number and the constant $M_{\mathcal{L}} > 0$ depends only on $\mathcal{L}$. Moreover, for a given $\delta \in (0,1)$, to guarantee that $\frac{1}{N}\sum_{t=1}^N \mathbb{E}_{\{\xi_j\}_{j=1}^\infty}\left[\|\nabla \mathcal{L}(\theta_t)\|^2\right] < \delta$, N needs to be at most $O\left(\delta^{-\frac{1}{1-\beta}}\right)$.

Proof. The proof of Theorem 1 follows from Theorem 2.8 in Wang *et al.* [87]. Clearly, Algorithm 14 falls under the scope of the stochastic quasi-Newton (SQN) method in Wang *et al.* [87]. In particular, by Proposition 1, the pre-conditioning matrix $H = F_{\mathrm{TNT}}^{-1}$. Moreover, to apply Theorem 2.8 in Wang *et al.* [87], we need to show that AS.1 - AS.4 in Wang *et al.* [87] hold. First, AS.1 and AS.2 in Wang *et al.* [87] are the same as Assumption 8 and Assumption 9, respectively in Section 4.4. Second, by Lemma 10, since $||F_{\mathrm{TNT}}||$ is both upper and lower bounded, so is $||F_{\mathrm{TNT}}^{-1}||$. Hence, AS.3 in Wang *et al.* [87] is ensured. Finally, Algorithm 14 itself ensures AS.4 in Wang *et al.* [87]

holds. Hence, by Theorem 2.8 of Wang *et al.* [87], the result is guaranteed.

□

Algorithm 15 Tensor-Normal Training

Require: Given batch size m, learning rate $\{\alpha_t\}_{t\geq 1}$, weight decay factor γ, damping value ϵ, statistics update frequency T_1, inverse update frequency T_2

$\mu = 0.9, \beta = 0.9$
Initialize $\widehat{G_l^{(i)}} = \mathbb{E}[G_l^{(i)}]$ $(l = 1, .., k,\ i = 1, ..., k_l)$ by iterating through the whole dataset, $\widehat{\nabla_{W_l}\mathcal{L}} = 0$ $(l = 1, ..., L)$
for $t = 1, 2, \ldots$ **do**
  Sample mini-batch M_t of size m
  Perform a forward-backward pass over M_t to compute the mini-batch gradient $\overline{\nabla\mathcal{L}}$
  if $t \equiv 0 \pmod{T_1}$ **then**
    Perform another backward pass over M_t with y sampled from the predictive distribution to compute $\overline{G_l} = \overline{\mathcal{D}W_l}$ averaged across M_t $(l = 1, ..., L)$
  end if
  for $l = 1, ...L$ **do**
    $\widehat{\nabla_{W_l}\mathcal{L}} = \mu\widehat{\nabla_{W_l}\mathcal{L}} + \overline{\nabla_{W_l}\mathcal{L}}$
    if $t \equiv 0 \pmod{T_1}$ **then**
      Update $\widehat{G_l^{(i)}} = \beta\widehat{G_l^{(i)}} + (1-\beta)\overline{G_l}^{(i)}$ for $i = 1, ..., k_l$
    end if
    if $t \equiv 0 \pmod{T_2}$ **then**
      Determine $\tilde{U}_1^{(l)}, ..., \tilde{U}_{k_l}^{(l)}$ from $\widehat{G_l^{(1)}}, ..., \widehat{G_l^{(k_l)}}$ by (4.10)
      Recompute $(\tilde{U}_1^{(l)} + \epsilon I)^{-1}, ..., (\tilde{U}_{k_l}^{(l)} + \epsilon I)^{-1}$
    end if
    $p_l = \widehat{\nabla_{W_l}\mathcal{L}} \times_1 (\tilde{U}_1^{(l)} + \epsilon I)^{-1} \times_2 \cdots \times_k (\tilde{U}_k^{(l)} + \epsilon I)^{-1}$
    $p_l = p_l + \gamma W_l$
    $W_l = W_l - \alpha_t \cdot p_l.$
  end for
end for

4.5 Algorithm Details and Pesudo-code on TNT

In practice, we compute $\overline{G} = \overline{\mathcal{D}W}$ averaged over a minibatch of data at every iteration, and use the value of $\overline{G}^{(i)}$ to update a moving average estimate $\widehat{G^{(i)}}$ of $\mathbb{E}[G^{(i)}]$. The extra work for these computations (as well as for updating the inverses of $\tilde{U}_i$) compared with a stochastic gradient descent method is amortized by only performing them every T_1 (and T_2) iterations, which is also

Table 4.1: Memory and per-iteration time complexity of TNT and other comparing methods beyond that required by SGD

Name	Memory	Time (per-iteration)
TNT	$O(\sum_{i=1}^k d_i^2)$	$O((\frac{1}{T_1}m + \sum_{i=1}^k d_i) \prod_{i=1}^k d_i + \frac{1}{T_2} \sum_{i=1}^k d_i^3)$
Shampoo	$O(\sum_{i=1}^k d_i^2)$	$O((\sum_{i=1}^k d_i) \prod_{i=1}^k d_i + (\frac{1}{T_2} \sum_{i=1}^k d_i^3$ *- if using SVD*))
Adam-like	$O(\prod_{i=1}^k d_i)$	$O(\prod_{i=1}^k d_i)$
Newton-like	$O(\prod_{i=1}^k d_i^2)$	*- depends on specific algorithm*

the approach used in KFAC and Shampoo, and does not seems to degrade the overall performance of the TNT algorithm. Moreover, we compute $\mathbb{E}[G^{(i)}]$ using the whole dataset at the initialization point as a warm start, which is also done in our implementations of Shampoo and KFAC.

In Algorithm 15, we present a detailed pseudo-code for our actual implementation of TNT. The highlighted parts, i.e., Lines 7, 15 and 16, indicate where TNT differs significantly from Shampoo.

4.6 A Comparison on Memory and Per-iteration Time Complexity

To compare the memory requirements and per-iteration time complexities of different methods, we consider the case where we optimize one tensor variable of size $d_1 \times \cdots \times d_k$ using minibatches of size m at every iteration. A plain SGD method requires $O(\prod_{i=1}^k d_i)$ to store the model parameters and the gradient, whereas its per-iteration time complexity is $O(m \prod_{i=1}^k d_i)$. Table 4.1 lists the memory requirements and per-iteration time complexities in excess of that required by SGD for different methods.

Compared with a classic Newton-like method (e.g. BFGS), TNT (as well as Shampoo) reduces the memory requirement from $O(\prod_{i=1}^k d_i^2)$ to $O(\sum_{i=1}^k d_i^2)$, which is comparable to that of Adam-like adaptive gradient methods. In fact, if the d_i's are all equal to d and $3 \leq k << d$, the Kronecker-factored TNT pre-conditioning matrix requires kd^2 storage, which is less than that required by the diagonal pre-conditioners used by Adam-like methods. On the other hand, in terms of per-iteration time complexity, TNT (as well as Shampoo) only introduces a mild overhead for estimating the statistics $\mathbb{E}[G^{(i)}]$'s, inverting the pre-conditioning matrices, and computing the updating direction. Also, the first two of these operations can be amortized by only performing them every T_1 and T_2

iterations. Lastly, the extra work of $O(\frac{1}{T_1} m \prod_{i=1}^{k} d_i)$ required by TNT relative to Shampoo is due to the extra backward pass needed to estimate the true Fisher, as opposed to the EF.

Moreover, although TNT and Shampoo both incur $\frac{1}{T_2} \sum_{i=1}^{k} d_i^3$ amortized time to invert the preconditioning matrices, the SVD operation in Shampoo can take much more time than the matrix inverse operation in TNT, especially when the matrix size is large[2].

The per-iteration computational complexity of KFAC is more complicated because it depends on the type of the layer/variable. For linear layers, TNT and KFAC both use two matrices, whose sizes are the number of input nodes and output nodes, respectively. For convolutional layers, TNT uses three matrices, whose sizes are the size of filter, number of input channels, and number of output channels, whereas KFAC uses two matrices whose sizes are the size of the filter times the number of input channels, and the number of output channels. As a result, the first KFAC matrix requires much more memory. In general, the per-iteration complexity of KFAC is no less than that of TNT.

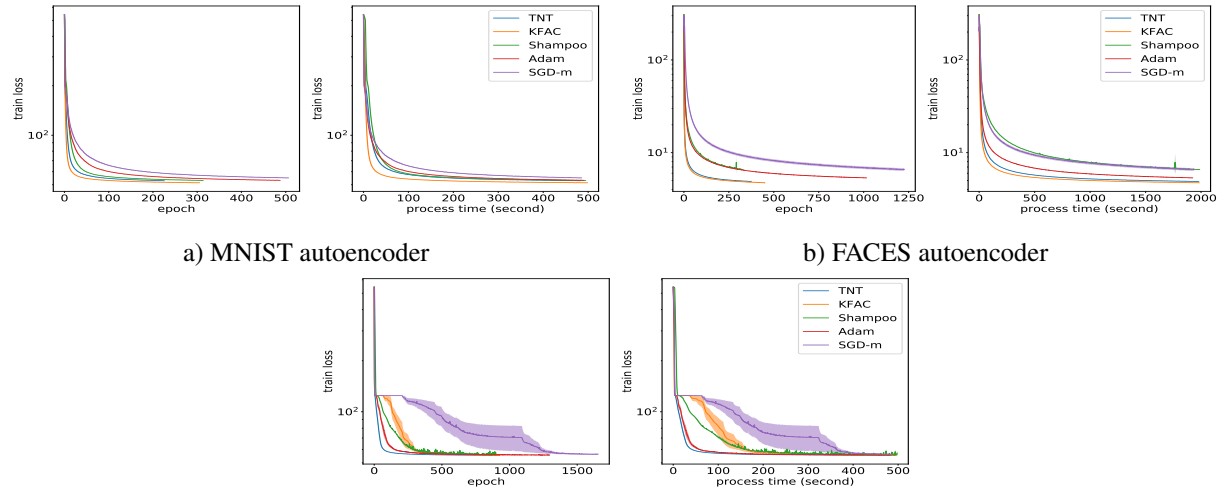
4.7 Experiments

In this section, we compare TNT with some state-of-the-art second-order (KFAC, Shampoo) and first-order (SGD with momentum, Adam) methods (see Section 4.7.2 on how these methods were implemented).

Approximate Hessian-based K-BFGS methods are also state-of-the-art Kronecker-factored second-order method for training deep learning models. Since our focus is on optimizers that use Fisher or empirical Fisher as the preconditioning matrix, we did not include K-BFGS in our comparison. See Chapters 2 and 3 for the performance of K-BFGS methods.

Our experiments were run on a machine with one V100 GPU and eight Xeon Gold 6248 CPUs using PyTorch [68]. Each algorithm was run using the best hyper-parameters, determined by an appropriate grid search (specified below), and 5 different random seeds. In Figures 4.2 and 4.3 the

[2]In Anil *et al.* [5] it is shown that replacing the SVD operation by a coupled Schur-Newton method saves time for matrices of size greater than 1000×1000. In our experiments, we used the coupled Newton method implementation of Shampoo.



a) MNIST autoencoder

b) FACES autoencoder

c) CURVES autoencoder

Figure 4.2: Optimization performance of TNT, KFAC, Shampoo, Adam, and SGD-m on three autoencoder problems

performance of each algorithm is plotted: the solid curves give results obtained by averaging the 5 different runs, and the shaded area depicts the ±standard deviation range for these runs. In Figure 4.3, upper row depicts the training loss whereas lower row depicts the validation classification error. Our code is available at `https://github.com/renyiryry/tnt_neurips_2021`.

4.7.1 Major Numerical Experiments

Optimization: Autoencoder Problems

We first compared the optimization performance of each algorithm on two autoencoder problems [43] with datasets MNIST [52], FACES[3], and CURVES, which were also used in Martens [59], Martens and Grosse [62], and Botev *et al.* [15] as benchmarks to compare different algorithms. For each algorithm, we conducted a grid search on the learning rate and damping value based on the criteria of minimal training loss. We set the Fisher matrix update frequency $T_1 = 1$ and inverse update frequency $T_2 = 20$ for all of the second-order methods. Details of our experiment settings are listed in Section 4.7.2.

From Figure 4.2, it is clear that TNT outperformed SGD with momentum, Adam and Shampoo,

[3] `https://cs.nyu.edu/~roweis/data.html`

Table 4.2: Average of validation classification accuracy (%) achieved by TNT, KFAC, Shampoo, Adam, and SGD-m using 5 different random seeds with best HP values

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
TNT	94.31	93.48	76.01	71.70
KFAC	94.39	93.31	76.21	71.14
Shampoo	93.88	93.13	76.02	70.35
Adam	94.28	93.35	75.64	70.35
SGD-m	94.14	93.13	75.41	70.18

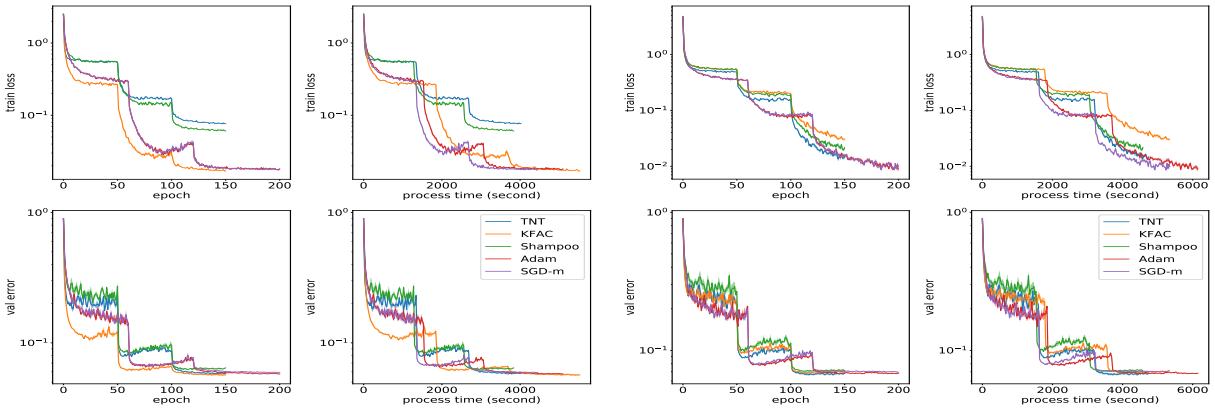
both in terms of per-epoch progress and process time. The comparison between TNT and KFAC is problem-dependent; TNT outperforms KFAC on CURVES, performs similarly as KFAC on FACES, and underperforms KFAC on MNIST.

We repeated these experiments using a grid search on more hyper-parameters, and obtained results (see Figure 4.6 in Sec 4.7.3) that further support our observations based on Figure 4.2.

Generalization: Convolutional Neural Networks

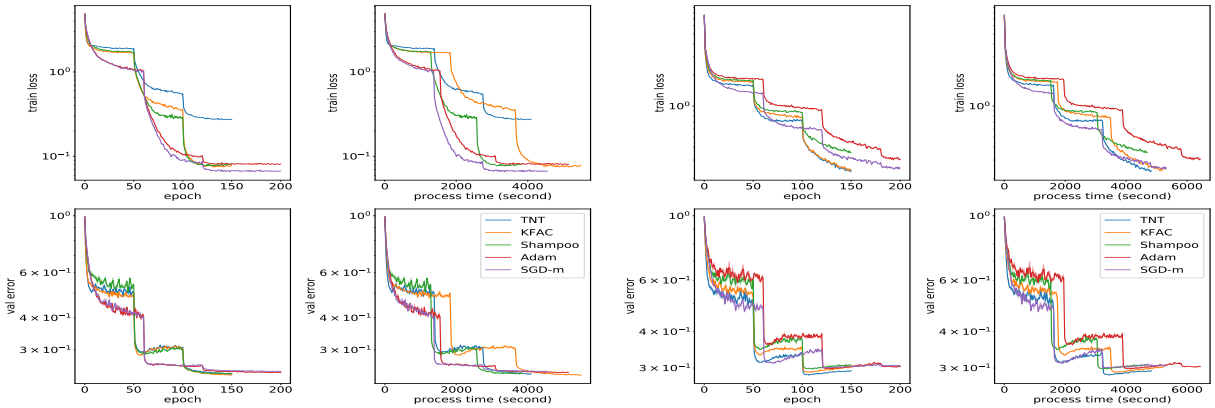
We then compared the generalization ability of each algorithm on two CNN models, namely, ResNet32 [40] and VGG16 [82], using both the CIFAR10 and CIFAR100 [48] dataset. The first-order methods were run for 200 epochs during which the learning rate was decayed by a factor of 0.1 every 60 epochs, whereas the second-order methods were run for 150 epochs during which the learning rate was decayed by a factor of 0.1 every 50 epochs. Moreover, as indicated in Loshchilov and Hutter [55] and Zhang *et al.* [96], weight decay, different from the L_2 regularization added to the loss function, helps improve generalization across different optimizers. Thus, for each algorithm, we conducted a grid search on the initial learning rate and the weight decay factor based on the criteria of maximal validation classification accuracy. The damping parameter was set to 1e-8 for Adam (following common practice), and 0.03 for KFAC[4]. For TNT and Shampoo, we set $\epsilon = 0.01$. We set $T_1 = 10$ and $T_2 = 100$ for the three second-order methods (same as in Zhang *et al.* [96]). Details of our experiment settings and a further discussion of the choice of damping

[4]The value of 0.03 is suggested in `https://github.com/alecwangcq/KFAC-Pytorch`, a github repo by the authors of Zhang *et al.* [96].



a) CIFAR-10, VGG16

b) CIFAR-10, ResNet-32



c) CIFAR-100, VGG16

d) CIFAR-100, ResNet-32

Figure 4.3: Generalization performance of TNT, KFAC, Shampoo, Adam, and SGD-m on four CNN models

hyper-parameters can be found in Section 4.7.2.

The results in Table 4.2 and Figure 4.3 indicate that, with a proper learning rate and weight decay factor, second-order methods and Adam exhibit roughly the same generalization performance as SGD with momentum, which corroborate the results in Loshchilov and Hutter [55] and Zhang *et al.* [96]. In particular, TNT has a similar (and sometimes better) generalization performance than the other methods. For example, comparing TNT with SGD-m, TNT (SGD-m) achieves 71.70% (70.18%) validation accuracy with ResNet32 on CIFAR100 (see Table 4.2 for the accuracy achieved by the other algorithms). Moreover, in terms of process time, TNT is roughly twice (equally) as fast as SGD with momentum on ResNet32 model in Figure 4.3b and d (on VGG16 in Figure 4.3a and c). This illustrates the fact that TNT usually requires only moderately more computational effort per-iteration but fewer iterations to converge than first-order methods.

Also, as shown on the VGG16 model, KFAC seems to be considerably slower than TNT and Shampoo on larger models. This is because KFAC needs to compute and invert larger matrices compared with TNT and Shampoo for convolutional layers. Moreover, the VGG16 models we tested has lager convolutional layers than the ResNet32 model, which makes the difference in running time clearer.

We also compared TNT with a variant of it that uses the empirical rather than the true Fisher as the preconditioning matrix. The results of this comparison, which are presented in Section 4.7.3, suggest that it is preferable to use Fisher rather than empirical Fisher as pre-conditioning matrices in TNT.

4.7.2 Implementation Details of the Experiments

In our implementations of the algorithms that we compared to TNT, we included in all of the techniques like weight decay and momentum, so that our numerical experiments would provide a FAIR comparison. Consequently, we did not include some special techniques that have been incorporated in some of the algorithms as described in previously published papers, since to keep the comparisons fair, we would have had to incorporate such techniques in all of the algorithms

(see Section 4.7.2 for more details).

Competing Algorithms

In SGD with momentum, we updated the momentum of the gradient $m = \mu \cdot m + g$ at every iteration, where g denotes the minibatch gradient at current iteration. The gradient momentum is also used in the second-order methods, in our implementations.

For Adam, we follow exactly the algorithm in Kingma and Ba [46] with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In particular, we follow the approach in Kingma and Ba [46] in estimating the momentum of gradient by $m = \beta_1 \cdot m + (1 - \beta_1) \cdot g$. The role of β_1 and β_2 is similar to that of μ and β in Algorithm 15 and Algorithm 16, as we will describe below.

In the experiments on CNNs, we use weight decay (same as in Algorithms 15 and 16) on SGD and Adam, similar to SGDW and AdamW in Loshchilov and Hutter [55] (for further details, see Section 4.7.2).

Shampoo

In Algorithm 16, we present our implementation of Shampoo, which mostly follows the description of it given in Gupta *et al.* [39]. Several major improvements are also included, following the suggestions in Anil *et al.* [5], including:

1. In Line 9 of Algorithm 16, a moving average is used to update the estimates $\widehat{G_l^{(i)}}$, as is done in our implementations of TNT and KFAC. This approach is also used in Adam, whereas summing the $G_l^{(i)}$'s over all iterations, as in Gupta *et al.* [39], is analogous to what is done in AdaGrad, upon which Shampoo is based.

2. In Line 12 of Algorithm 16, we use a coupled Newton method to compute inverse roots of the matrices (as proposed in Anil *et al.* [5]), rather than using SVD. The coupled Newton approach has been shown to be much faster than SVD, and also preserves relatively good accuracy in terms of computing inverse roots. The coupled Newton method performs reasonably well (without tuning) using a max iteration number of 100 and an error tolerance of 1e-6.

Algorithm 16 Shampoo

Require: Given batch size m, learning rate $\{\alpha_t\}_{t\geq 1}$, weight decay factor γ, damping value ϵ, statistics update frequency T_1, inverse update frequency T_2

$\mu = 0.9, \beta = 0.9$

Initialize $\widehat{G_l^{(i)}} = \mathbb{E}[G_l^{(i)}]$ $(l = 1, .., k,\ i = 1, ..., k_l)$ by iterating through the whole dataset, $\widehat{\nabla_{W_l}\mathcal{L}} = 0$ $(l = 1, ..., L)$

for $t = 1, 2, \ldots$ **do**

  Sample mini-batch M_t of size m

  Perform a forward-backward pass over the current mini-batch M_t to compute the minibatch gradient $\overline{\nabla\mathcal{L}}$

  for $l = 1, ...L$ **do**

    $\widehat{\nabla_{W_l}\mathcal{L}} = \mu\widehat{\nabla_{W_l}\mathcal{L}} + \overline{\nabla_{W_l}\mathcal{L}}$

    if $t \equiv 0 \pmod{T_1}$ **then**

      Update $\widehat{G_l^{(i)}} = \beta\widehat{G_l^{(i)}} + (1-\beta)\overline{G_l}^{(i)}$ for $i = 1, ..., k_l$ where $\overline{G_l} = \overline{\nabla_{W_l}\mathcal{L}}$

    end if

    if $t \equiv 0 \pmod{T_2}$ **then**

      Recompute $\left(\widehat{G_l^{(1)}} + \epsilon I\right)^{-1/2k_l}, ..., \left(\widehat{G_l^{(k_l)}} + \epsilon I\right)^{-1/2k_l}$ with the coupled Newton method

    end if

    $p_l = \widehat{\nabla_{W_l}\mathcal{L}} \times_1 \left(\widehat{G_l^{(1)}} + \epsilon I\right)^{-1/2k_l} \times_2 \cdots \times_k \left(\widehat{G_l^{(k_l)}} + \epsilon I\right)^{-1/2k_l}$

    $p_l = p_l + \gamma W_l$

    $W_l = W_l - \alpha_t \cdot p_l$

  end for

end for

Some other modifications proposed in Anil *et al.* [5] are not included in our implementation of Shampoo, mainly because these modifications can also be applied to TNT, and including them only in Shampoo would introduce other confounding factors.

(i) We did not explore multiplying the damping term in the pre-conditioner by the maximum eigenvalue λ_{max} of the contraction matrix. Moreover, this modification is somewhat problematic, since, if the model contains any variables that always have a zero gradient (e.g. the bias in a convolutional layer that is followed by a BN layer), the optimizer would become unstable because the pre-conditioner of the zero-gradient variables would be the zero matrix, (note that in this case $\lambda_{max} = 0$).

(ii) We did not explore the diagonal variant of Shampoo, as we mainly focused on the comparison between different pre-conditioning matrices, and TNT can also be extended to a diagonal version; similarly, we did not explore the variant proposed in Anil *et al.* [5] that divides large tensors into small blocks.

KFAC

In this subsection, we briefly describe our implementation of KFAC. The preconditioning matrices that we used for linear layers and convolutional layers are precisely as those described in Martens and Grosse [62] and Grosse and Martens [38], respectively. For the parameters in the BN layers, we used the gradient direction, exactly as in `https://github.com/alecwangcq/KFAC-Pytorch`.

As in our implementations of TNT and Shampoo, and as suggested in Grosse and Martens [38], we did a warm start to estimate the pre-conditioning KFAC matrices in an initialization step that iterated through the whole data set, and adopted a moving average scheme to update them with $\beta = 0.9$ afterwards.

In inverting the KFAC matrices and computing the updating direction, we inverted the damped KFAC matrices and used them to compute the updating direction, where the damping factors for both A and G were set to be $\pi\sqrt{\lambda}$ and $\frac{1}{\pi_l}\sqrt{\lambda}$, where $\pi = \sqrt{\frac{\text{trace}(A\otimes I)}{\text{trace}(I\otimes G)}}$ and λ is the overall damping

Table 4.3: Hyper-parameters (learning rate, damping) used to produce Figure 4.2

Name	MNIST	FACES	CURVES
TNT	(1e-4, 0.1)	(1e-6, 0.003)	(3e-6, 0.01)
KFAC	(0.03, 1)	(0.01, 0.1)	(0.3, 30)
Shampoo	(3e-4, 3e-4)	(3e-4, 3e-4)	(3e-4, 3e-4)
Adam	(1e-4, 1e-4)	(1e-4, 1e-4)	(1e-3, 1e-3)
SGD-m	(0.003, -)	(0.001, -)	(0.003, -)

value; this choice is the same as the KFAC algorithm we tested in Chapter 3 (see Algorithm 12), and is also suggested in Martens and Grosse [62] and Grosse and Martens [38].

Further, we implemented weight decay exactly as in TNT (Algorithm 15) and Shampoo (Algorithm 16).

Experiment Settings for the Autoencoder Problems

MNIST has 60,000 training data, FACES[5] has 103,500 training data, and CURVES[6] has 20,000 training data. The model architectures, including layer widths, activation functions, loss functions, are exactly the same as the ones used in Chapter 3 (see Section 3.7.2). The above settings largely mimic the settings in Martens [59], Martens and Grosse [62], and Botev *et al.* [15]. Since we primarily focused on optimization rather than generalization in these tasks, we did not include L_2 regularization or weight decay. For all algorithms, we used a batch size of 1,000 at every iteration.

In order to obtain Figure 4.2, we first conducted a grid search on the learning rate (lr) and damping value based on the criteria of minimizing the training loss. The ranges of the grid searches used for the algorithms in our tests were:

- SGD-m:
  - lr: 1e-4, 3e-4, 0.001, 0.003, 0.01, 0.03
  - damping: not applicable
- Adam:

[5]Downloadable at `www.cs.toronto.edu/~jmartens/newfaces_rot_single.mat`.
[6]Downloadable at `http://www.cs.toronto.edu/~jmartens/digs3pts_1.mat`

Table 4.4: Hyper-parameters (initial learning rate, weight decay factor, damping) used to produce Figure 4.3

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
TNT	(1e-4, 10, 0.1)	(1e-4, 10, 0.01)	(1e-4, 10, 0.1)	(1e-4, 10, 0.01)
KFAC	(3e-4, 1, 0.001)	(0.01, 0.1, 0.1)	(0.01, 0.1, 1)	(0.1, 0.01, 0.1)
Shampoo	(0.001, 1, 0.1)	(0.01, 0.1, 0.01)	(0.001, 1, 0.01)	(0.01, 0.1, 0.01)
Adam	(0.003, 0.1, 0.1)	(0.003, 0.1, 0.01)	(3e-4, 1, 0.01)	(0.01, 0.1, 0.01)
SGD-m	(0.003, 0.1, -)	(0.03, 0.01, -)	(0.003, 0.1, -)	(0.03, 0.01, -)

  - lr: 1e-5, 3e-5, 1e-4, 3e-4, 0.001, 0.003, 0.01
  - damping (i.e. the ϵ hyperparameter of Adam): 1e-8, 1e-4, 1e-2

- Shampoo:
  - lr: 1e-5, 3e-5, 1e-4, 3e-4, 0.001, 0.003
  - damping (i.e. ϵ in Algorithm 16): 1e-4, 3e-4, 0.001, 0.003, 0.01

- TNT:
  - lr: 1e-7, 3e-7, 1e-6, 3e-6, 1e-5, 3e-5, 1e-4, 3e-4, 0.001
  - damping (i.e. ϵ in Algorithm 15): 0.001, 0.003, 0.01, 0.03, 0.1, 0.3

- KFAC:
  - lr: 1e-4, 3e-4, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3
  - damping: 0.01, 0.03, 0.1, 0.3, 1, 3, 10

The best hyper-parameter values determined by our grid searches are listed in Table 4.3.

Experiment Settings for the CNN Problems

Both CIFAR-10 and CIFAR-100 have 50,000 training data and 10,000 testing data (used as the validation set in our experiments). For all algorithms, we used a batch size of 128 at every iteration.

In training, we applied data augmentation as described in Krizhevsky *et al.* [49], including random horizontal flip and random crop.

The ResNet32 model refers to the one in Table 6 of He *et al.* [40]. The VGG16 model refers to the "model D" in Simonyan and Zisserman [82], with the same modifications used in Chapter 3, i.e., that the 3 fully-connected (FC) layers at the end of the model being replaced with only one FC layer (input size equal the size of the output size of the last conv layer, and output size equal number of classes of the dataset), and a batch normalization layer is added after each of the convolutional layers in the model.

It is worth noting that, in TNT and Shampoo, for the weight tensor in the convolutional layers, instead of viewing it as a 4-way tensor, we view it as a 3-way tensor, where the size of its 3 ways (dimensions) corresponds to the size of the filter, the number of input channel, and the number of the output channel, respectively. As a result, the preconditioning matrices of TNT and Shampoo will come from the Kronecker product of three matrices, rather than four matrices.

Weight decay, which is related to, but not the same as L_2 regularization added to the loss function, has been shown to help improve generalization performance across different optimizers [55, 96]. In our experiments, we adopted weight decay for all algorithms. The use of weight decay for TNT and Shampoo is described in Algorithm 15 and Algorithm 16, respectively, and is similarly applied to KFAC. Also note that weight decay is equivalent to L_2 regularization for pure SGD (without momentum). However, the equivalence does not hold for SGD with momentum. For the sake of a fair comparison, we also applied weight decay for SGD-m.

In order to obtain Figure 4.3, we first conducted a grid search on the initial learning rate (lr), weight decay (wd) factor and damping (when applicable) based on the criteria of maximizing the classification accuracy on the validation set. The range of the grid searches for the algorithms in our tests were:

- SGD-m:
  - lr: 3e-4, 1e-3, 3e-3, 0.01, 0.03, 0.1, 0.3

  - wd: 0.001, 0.01, 0.1, 1

- Adam:
  - lr: 3e-5, 1e-4, 3e-4, 1e-3, 3e-3, 0.01, 0.03, 0.1
  - wd: 0.01, 0.1, 1, 10
  - damping: 1e-8, 1e-4, 0.01, 0.1, 1

- Shampoo:
  - lr: 3e-5, 1e-4, 3e-4, 0.001, 0.003, 0.01, 0.03, 0.1
  - wd: 0.01, 0.1, 1, 10
  - damping (i.e., ϵ in Algorithm 16): 0.001, 0.01, 0.1, 1

- TNT:
  - lr: 1e-6, 3e-6, 1e-5, 3e-5, 1e-4, 3e-4, 0.001
  - wd: 1, 10, 100
  - damping (i.e., ϵ in Algorithm 15): 0.001, 0.01, 0.1, 1

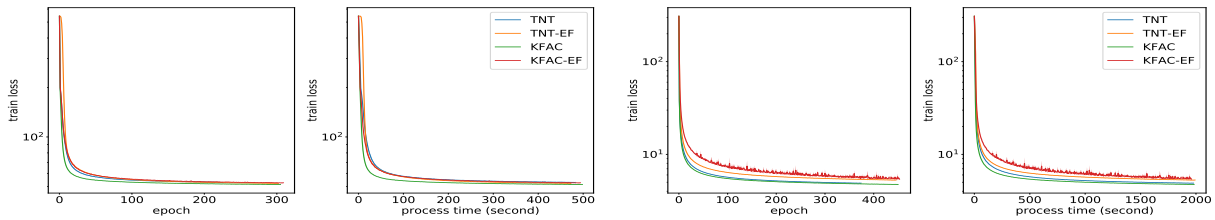
- KFAC:
  - lr: 1e-3, 3e-3, 0.01, 0.03, 0.1, 0.3
  - wd: 0.001, 0.01, 0.1, 1
  - damping: 1e-4, 0.001, 0.01, 0.1, 1, 10, 100

The best hyper-parameter values are listed in Table 4.4.

4.7.3 Additional Numerical Experiments

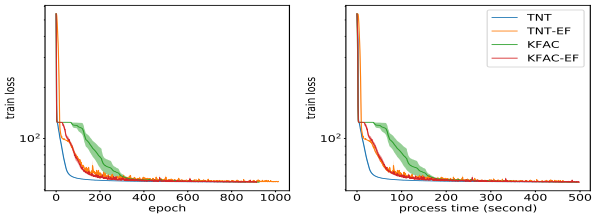
A Comparison between TNT and TNT-EF

In this subsection, we compare our proposed TNT algorithm against a variant of it, TNT-EF, which uses an empirical Fisher (EF) preconditioning matrix in place of the true Fisher matrix. In



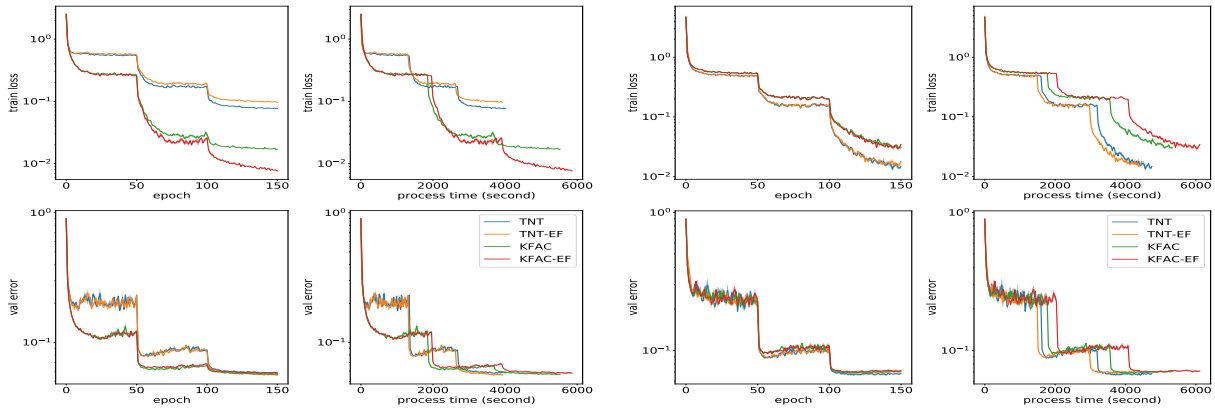
a) MNIST autoencoder

b) FACES autoencoder



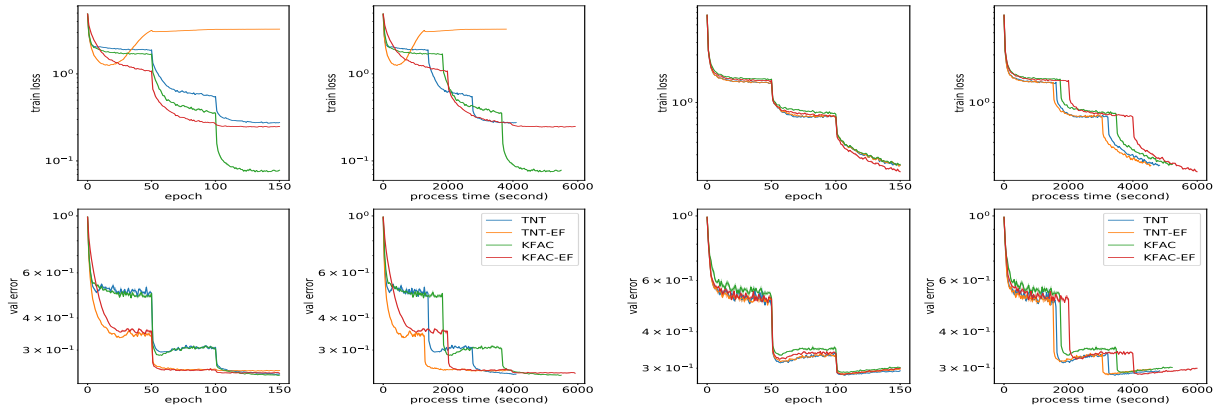
c) CURVES autoencoder

Figure 4.4: Optimization performance of TNT, KFAC and their EF counterparts on three autoencoder problems.



a) CIFAR-10, VGG16

b) CIFAR-10, ResNet-32



c) CIFAR-100, VGG16

d) CIFAR-100, ResNet-32

Figure 4.5: Generalization performance of TNT, KFAC, and their EF counterparts on four CNN models.

Table 4.5: Hyper-parameters (learning rate, damping) used to produce Figure 4.4

Name	MNIST	FACES	CURVES
TNT-EF	(3e-6, 0.01)	(3e-6, 0.01)	(0.003, 1)
KFAC-EF	(1, 100)	(0.3, 30)	(1, 100)

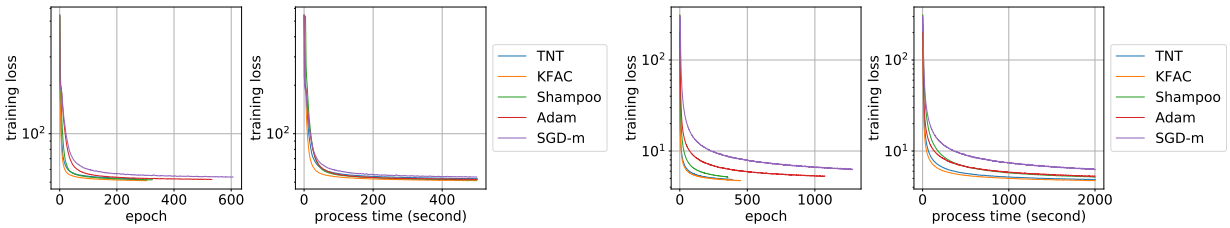
Table 4.6: Hyper-parameters (initial learning rate, weight decay factor, damping) used to produce Figure 4.5

	TNT-EF	KFAC-EF
CIFAR-10 + VGG16	(1e-4, 10, 0.1) $\rightarrow$ 94.50%	(3e-4, 1, 0.001) $\rightarrow$ 94.29%
CIFAR-10 + ResNet32	(1e-4, 10, 0.01) $\rightarrow$ 93.24%	(0.01, 0.1, 0.1) $\rightarrow$ 93.16%
CIFAR-100 + VGG16	(3e-5, 10, 0.1) $\rightarrow$ 75.27%	(3e-4, 1, 0.1) $\rightarrow$ 75.55%
CIFAR-100 + ResNet32	(1e-4, 10, 0.01) $\rightarrow$ 71.58%	(0.01, 0.1, 0.001) $\rightarrow$ 71.61%

other words, TNT-EF does everything specified in Algorithm 15, except that it does not perform the extra backward pass in Line 7 of Algorithm 3. When updating the matrices $\widehat{G_l^{(i)}}$, TNT-EF uses the empirical minibatch gradient, rather than the sampling-based minibatch gradient, i.e. the one coming from the extra backward pass. To provide more information, we also include KFAC and KFAC-EF in our comparison, where KFAC-EF uses the empirical Fisher as its preconditioning matrix.

We conducted a hyper-parameter grid search for TNT-EF and KFAC-EF, following the same procedure as the one that was used for TNT and KFAC, whose performance was plotted in Figures 4.2 and 4.3. The best values for the TNT-EF and KFAC-EF hyper-parameters that we obtained are listed in Tables 4.5 and 4.6. We then plotted in Figures 4.4 and 4.5, the performance of TNT-EF and KFAC-EF, along with that of TNT and KFAC, using for it the hyper-parameters given in Tables 4.3 and 4.4. The upper row of Figure 4.5 depicts the training loss, whereas the lower row depicts the validation classification error.

As shown in Figure 4.4, TNT performed at least as well as TNT-EF, on the MNIST problem, and performed somewhat better on the FACES and CURVES problems. For the four CNN problems shown in Figure 4.5, TNT achieved better validation accuracy than TNT-EF on 3 of them. We can similarly observe the comparison between KFAC and KFAC-EF. These observations confirm



a) MNIST autoencoder b) FACES autoencoder

Figure 4.6: Optimization performance of TNT, KFAC, Shampoo, Adam, and SGD-m on two autoencoder problems, with more extensive tuning

Table 4.7: Hyper-parameters (learning rate, damping, μ, β) used to produce Figure 4.6

Problem	Algorithm	(learning rate, damping, μ, β)
MNIST	TNT	(1e-4, 0.1, 0.9, 0.9)
MNIST	KFAC	(0.001, 0.1, 0.99, 0.99)
MNIST	Shampoo	(1e-4, 3e-4, 0.99, 0.99)
MNIST	Adam	(1e-4, 1e-4, 0.99, 0.99)
MNIST	SGD-m	(0.001, -, 0.99, -)
FACES	TNT	(1e-6, 0.003, 0.9, 0.9)
FACES	KFAC	(0.01, 0.1, 0.9, 0.9)
FACES	Shampoo	(1e-4, 3e-4, 0.99, 0.999)
FACES	Adam	(1e-4, 1e-4, 0.9, 0.9)
FACES	SGD-m	(0.001, -, 0.9, -)

the widely held opinion that the Fisher matrix usually carries more valuable curvature information than the empirical Fisher metrix.

More on Hyper-parameter Tuning

In this subsection, we expand on the experiments whose results are plotted in Figure 4.2, by incorporating the tuning of more hyper-parameters. To be more specific, we tuned the following hyper-parameters jointly:

1. SGD-m: learning rate and μ;

2. all other algorithms[7]: learning rate, damping, μ, and β.

The searching range for learning rate and damping is the same as in Sec 4.7.2, whereas the

[7]For Adam, μ and β refer to β_1 and β_2, respectively.

searching range for μ and β were set to be $\{0.9, 0.99, 0.999\}$. The obtained values for the hyper-parameters are listed in Table 4.7.

Figure 4.6 depicts the performance of different algorithms with hyper-parameters obtained from the aforementioned more extensive tuning process. Comparing the performance of different algorithms in Figure 4.6, we can see that the observations we made from Figure 4.2 still hold to a large extent. Moreover, with extensive tuning, second-order methods seem to perform similarly with each other, and are usually better than well-tuned first order methods on these problems.

As a final point, we would like to mention that one could also replace the constant learning rate for all of the algorithms tested with a "warm-up, then decay" schedule, which has been shown to result in good performance on these problems in Anil *et al.* [5]. Also, one could perform a more extensive tuning for the CNN problems. In particular, one could tune the initial learning rate, weight decay factor, damping, μ, and β jointly for the CNN problems.

See more in Choi *et al.* [23] and Schmidt *et al.* [78] for the importance and suggestions on hyper-parameter tuning. Moreover, see Amid *et al.* [4] for other relevant numerical results, in particular for KFAC and Shampoo. In Amid *et al.* [4], KFAC is shown to work extremely well with a higher frequency of inversion, another direction for experiments that could be explored.

4.8 Conclusion and Further Discussions

In this chapter, we proposed a new second-order method, and in particular, an approximate natural gradient method TNT, for training deep learning models. By approximating the Fisher matrix using the structure imposed by the tensor normal distribution, TNT only requires mild memory and computational overhead compared with first-order methods. Our experiments on various deep learning models and datasets, demonstrate that TNT provides comparable and sometimes better results than the state-of-the-art (SOTA) methods, both from the optimization and generalization perspectives.

Due to space and computational resource constraints, we did not run experiments on even larger models such as ImageNet and advanced models for NLP tasks. However, the results in this chapter

already show very strong evidence of the potential of the TNT method. We also did not explore extending our method to a distributed setting, which has been shown to be a promising direction for second-order methods such as KFAC and Shampoo [6, 5]. Since TNT already performs very well on a single machine, we expect that it will continue to do so in a distributed setting. These issues will be addressed in future research.

As a final note, the preconditioning matrices of TNT (as well as those of Shampoo) are derived from the specific shape of the (tensor) parameters of the particular deep learning model that is being trained. One can, of course, reshape these parameters, e.g., by flattening the tensors into vectors, which gives rise to very different preconditioning matrices.

The method proposed in this chapter can be applied to any deep learning or machine learning model. If the model and/or data has a flawed design or contains bias, this could potentially have negative societal impacts. However, this possibility is beyond the scope of the work presented in this chapter.

Chapter 5: Efficient Subsampled Gauss-Newton and Natural Gradient Methods for Training Neural Networks

5.1 Contributions and Closely Related Work

5.1.1 Our Contributions

Our main contribution is the development of methods for training deep NNs that incorporate partial, but substantial, second-order information, while keeping the computational cost of each iteration comparable to that required by first-order methods. To achieve this, we propose new generic subsampled generalized Gauss-Newton and natural gradient methods that can be implemented efficiently and are provably convergent. Our methods add a Levenberg-Marquardt (LM) damping term to the Gauss-Newton and Fisher information matrices and invert the resulting matrices using the Sherman-Morrison-Woodbury formula. Moreover, by taking advantage of the Kronecker factored structure in these matrices, we are able to form and invert them in $O(n)$ time. Furthermore, we prove that semi-stochastic versions of our algorithms (i.e., those that use a full gradient combined with mini-batch stochastic Gauss-Newton or Fisher information matrices) converge to a stationary point. We demonstrate the effectiveness of our methods with numerical experiment, comparing both first-order method (SGD) and second-order methods (Hessian-free, KFAC).

5.1.2 Closely Related Work

Our methods were initially motivated by the Hessian-free approach of Martens [59], which approximates the Hessian by the generalized Gauss-Newton matrix and then approximately solves the huge $n \times n$ linear system involving that matrix and an LM damping term to update the n parameters of the NN by an "early-termination" linear CG method. Other closely related methods

include the Krylov subspace descent method of Vinyals and Povey [86], which generalizes the Hessian-free approach by constructing a Krylov subspace; the KFAC method [62], which uses the block-diagonal part of the Fisher matrix to approximate the Hessian; and the Kronecker Factored Recursive Approximation method [15], which uses a block-diagonal approximation of the Gauss-Newton matrix. For very recent work on properties of the natural gradient method and the Fisher matrix in the context of NNs see Bernacchia *et al.* [12], Cai *et al.* [21], and Zhang *et al.* [95].

5.2 Background

5.2.1 Feed-forward Neural Networks

Although our methods are applicable to a wide range of NN architectures, for simplicity, we focus on feed-forward fully-connected NNs with $L+1$ layers. At the l-th layer, given the vector of outputs from the preceding layer $v^{(l-1)}$ as input, $v^{(l)}$ is computed as $v^{(l)} = \phi^{(l)}(W^{(l)}v^{(l-1)} + b^{(l)})$, where $W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$, $b^{(l)} \in \mathbb{R}^{m_l}$, and $\phi^{(l)} : \mathbb{R}^{m_l} \to \mathbb{R}^{m_l}$ is a nonlinear activation function. Hence, given the input $x = v^{(0)}$, the NN outputs $\hat{y} = v^{(L)}$. To train the NN we minimize an empirical average loss

$$f(\theta) = \frac{1}{N}\sum_{i=1}^{N} f_i(\theta) = \frac{1}{N}\sum_{i=1}^{N} \varepsilon(\hat{y}_i(\theta), y_i), \tag{5.1}$$

where $\theta = \left(\text{vec}(W^{(1)})^T, (b^{(1)})^T, \cdots, \text{vec}(W^{(L)})^T, (b^{(L)})^T\right)^T$ ($\text{vec}(W)$ vectorizes the matrix W by concatenating its columns) and $\varepsilon(\hat{y}_i(\theta), y_i)$ is a loss function based on the differences between $\hat{y}_i$ and y_i, for the given set $\{(x_1, y_1), ..., (x_N, y_N)\}$ of N data points. Note $\theta \in R^n$, where $n = \sum_{l=1}^{L}(m_l m_{l-1} + m_l)$ can be extraordinarily large.

5.2.2 Approximations to the Hessian matrix

At iteration t, at the point $\theta = \theta^{(t)}$, Newton-like methods compute $p_t = -G_t^{-1}g_t$, where G_t is an approximation to the Hessian of $f(\theta^{(t)})$, and g_t is $\nabla f(\theta^{(t)})$, or an approximation to it, and then set $\theta^{(t+1)} = \theta^{(t)} + p_t$. Computing G_t and inverting it (solving $G_t p_t = -g_t$) is the core step of

such methods. Finding a balance between the cost of computing p_t and determining an accurate direction p_t is crucial to developing a good algorithm.

Gauss-Newton Method

In order to get a good approximation to the Hessian of $f(\theta)$, we first examine the Hessian of $f_i(\theta)$ corresponding to a single data point. By (5.1) it follows from the chain rule that

$$\frac{\partial^2 f_i(\theta)}{\partial \theta^2} = J_i^\top H_i J_i + \sum_{j=1}^{m_L} \left(\frac{\partial f_i(\theta)}{\partial \hat{y}_i} \right)_j \frac{\partial}{\partial \theta} (J_i)_j,$$

where $J_i = \frac{\partial \hat{y}_i}{\partial \theta}$, and $H_i = \frac{\partial^2 f_i(\theta)}{\partial (\hat{y}_i)^2}$. The Gauss-Newton (GN) method (e.g., see Nocedal and Wright [67] and Martens [59]) approximates the Hessian matrix by ignoring the second term in the above expression, i.e., the GN approximation to $\frac{\partial^2 f_i(\theta)}{\partial \theta^2}$ is $J_i^T H_i J_i$. Note that $J_i \in R^{m_L \times n}$ and $H_i \in R^{m_L \times m_L}$, and hence that H_i is a relatively small matrix. Finally, $\frac{\partial^2 f(\theta)}{\partial \theta^2}$ is approximated by

$$B_t = \frac{1}{N} \sum_{i=1}^{N} J_i^T H_i J_i. \tag{5.2}$$

Natural gradient method

The natural gradient (NG) method [2] modifies the gradient $\nabla f(\theta)$ by multiplying it by the inverse of the Fisher (information) matrix, which serves as an approximation to $\frac{\partial^2 f(\theta)}{\partial \theta^2}$:

$$B_t = F_t \equiv \frac{1}{N} \sum_{i=1}^{n} \nabla f_i(\theta^{(t)}) \nabla f_i(\theta^{(t)})^\top. \tag{5.3}$$

Properties of the approximations

There are several reasons why the NG and GN methods are well-suited for training NNs. First, even though the loss function (5.1) is a non-convex function of θ, H_i is positive semi-definite ($H_i \succeq 0$) for commonly-used loss functions (e.g., least-squared loss, cross entropy loss). Hence,

$J_i^T H_i J_i \succeq 0$. Also, $F_t \succeq 0$. p_t is a descent direction, as long as $-g_t$ is a descent direction and g_t is not in the null space of B_t. Second, the multiplication of an arbitrary vector by the matrix B_t or F can be done efficiently by backpropagation (see Section 5.4 or, e.g., Schraudolph [79]).

5.2.3 Mini-batch and damping

The prohibitively large amount of data and (relative) difficulty in computing the GN and Fisher matrices suggests simplifying these approximations to the Hessian matrix further. Consequently, as in Martens [59] and Martens and Grosse [62], we estimates (5.2) and (5.3) using a mini-batch of indices $S_2^t \subset \{1, 2, ..., N\}$ at iteration t where $|S_2^t| = N_2$.

Mini-batch approximations make the GN and Fisher matrices low-rank. Hence, we add λI to them to make them invertible (namely, the Levenberg-Marquardt (LM) method [65]). Thus, the approximation to the Hessian becomes

$$G_t = B_t + \lambda I, \tag{5.4}$$

where B_t is either the Fisher information matrix or the Gauss-Newton matrix.

Viewing the LM method as a trust-region method, the magnitude of λ is inversely related to the size of the region $||p|| \leq \Delta_t$ in which we are confident about the ability of the quadratic model

$$m_t(p) = f(\theta^{(t)}) + g_t^T p + \frac{1}{2} p^T B_t p$$

to approximate $f(\theta^{(t)} + p)$. Note that solving $B_t p = -g_t$ is equivalent as minimizing $m_t(p)$.

To determine the value of λ, let $\lambda = \lambda_{\text{LM}} + \tau$, where λ_{LM} is updated at each iteration, and $\tau > 0$. τ is typically very small and can be viewed as coming from an l_2 regularization term in the objective function, which is a common practice in training deep NNs to avoid possible over-fitting. It also ensures that $\lambda_t \geq \tau > 0$, which guarantees that the smallest eigenvalue of G_t is strictly positive.

To update λ_{LM}, we consider the ratio of the actual reduction in $f(\cdot)$ to the reduction in the

quadratic model $m_t(\cdot)$

$$\rho_t = \frac{f(\theta^{(t)}) - f(\theta^{(t)} + p_t)}{m_t(0) - m_t(p_t)} \tag{5.5}$$

to measure how "good" that model is. If ρ_t is positive and large, it means that the quadratic model is a good approximation. Hence, we enlarge the "trust region", by decreasing the value of λ_{LM}. If ρ_t is small, we increase the value of λ_{LM} (see Section 5.6 for more intuition). Specifically, λ_{LM} is updated as follows: **if** $\rho_t < \epsilon$: $\lambda_{\text{LM}}^{(t+1)} = boost \times \lambda_{\text{LM}}^{(t)}$; **else if** $\rho_t > 1 - \epsilon$: $\lambda_{\text{LM}}^{(t+1)} = drop \times \lambda_{\text{LM}}^{(t)}$; **else**: $\lambda_{\text{LM}}^{(t+1)} = \lambda_{\text{LM}}^{(t)}$, where $0 < \epsilon < \frac{1}{2}$, $drop < 1 < boost$. Finally, $\lambda_{t+1} = \lambda_{\text{LM}}^{(t+1)} + \tau$.

5.3 Our Innovation: a general framework for computing p_t

In the NN context, it is very expensive to compute (5.2) or (5.3); and even given G_t, computing p_t still requires $O(n^3)$ time, which is prohibitive. For these reasons Martens [59] proposed a Hessian-free method that uses an "early termination" linear conjugate gradient method to compute p_t approximately. Here we propose an alternative approach, that is both potentially faster, and is also exact.

5.3.1 Using the Sherman-Morrison-Woodbury (SMW) Formula

The matrix G_t for both the GN and NG methods has the form $G_t = \lambda I + \frac{1}{N_2} J^\top H J$, where $J^\top = (J_1^\top, \cdots, J_{N_2}^\top)$ and $H = \text{diag}\{H_1, \cdots, H_{N_2}\}$ for GN and $J^\top = (\nabla f_1(\theta), \cdots, \nabla f_{N_2}(\theta))$ and $H = I$ for NG. Using the well-known SMW formula,

$$G_t^{-1} = \frac{1}{\lambda}\left(I - \frac{1}{N_2} J^\top D_t^{-1} J\right), \text{ where } D_t = \lambda H^{-1} + \frac{1}{N_2} J J^\top. \tag{5.6}$$

Note that the matrix D_t in (5.6) is $N_2 m_L \times N_2 m_L$ in the GN case and $m_L \times m_L$ in the NG case, much smaller than the $n \times n$ LM matrix G_t, assuming N_2 is not too large in the GN case.

In cases where the H_i are not invertible (e.g., softmax regression with GN method), we can still

use SMW to obtain

$$G_t^{-1} = \frac{1}{\lambda}\left(I - \frac{1}{N_2}J^\top H D_t^{-1} J\right), \text{ where } D_t = \lambda I + \frac{1}{N_2} J J^\top H. \tag{5.7}$$

Because the analysis for these cases are similar to those where H_i is invertible, we will restrict our analysis to the symmetric expressions in (5.6).

5.3.2 Backpropagation in SMW

For an arbitrary vector $V \in R^{m_L}$, $J_i^\top V = \left(\frac{\partial \hat{y}_i}{\partial \theta}\right)^\top V = \frac{\partial\left((\hat{y}_i)^\top V\right)}{\partial \theta}$. Hence, we can compute the vector $J_i^\top V$ by backpropagating through the customized function $(\hat{y}_i)^\top V$. The other vectors needed in (5.6) can be computed similarly (See Section 5.4).

5.3.3 Computing D_t

We first demonstrate how to compute D_t in (5.6) in an efficient way. For a given data point i, let $DW_i^{(l)}$ denote the gradient of $f_i(\theta)$ w.r.t $W^{(l)}$. As shown in Section 5.4, $DW_i^{(l)}$ is a rank-one matrix, i.e., $DW_i^{(l)} = (g_i^{(l)})(v_i^{(l-1)})^\top$. Hence, the (i, j) element of D_t can be computed as

$$\begin{aligned}
\nabla f_i(\theta)^\top \nabla f_i(\theta) &= \sum_{l=1}^{L} \operatorname{vec}\left(DW_i^{(l)}\right)^\top \operatorname{vec}\left(DW_j^{(l)}\right) \\
&= \sum_{l=1}^{L} \operatorname{vec}\left((g_i^{(l)})(v_i^{(l-1)})^\top\right)^\top \operatorname{vec}\left((g_j^{(l)})(v_j^{(l-1)})^\top\right) \\
&= \sum_{l=1}^{L} \left((g_i^{(l)})^\top (g_j^{(l)})\right)\left((v_i^{(l-1)})^\top v_j^{(l-1)}\right)
\end{aligned}$$

For simplicity, we have ignored the b's in the above. Therefore, we compute D_t without explicitly writing out any $DW_i^{(l)}$, where all the vectors needed have been computed when doing backpropagation for the gradient.

Similarly, in the case of the GN matrix where D_t is defined in (5.6), we need to compute $J_{i_1} J_{i_2}^\top$ for all $i_1, i_2 = 1, ..., N$. The (j_1, j_2) element of $J_{i_1} J_{i_2}^\top$, namely $e_{j_1}^\top J_{i_1} J_{i_2}^\top e_{j_2}$, is the dot product of two

"backpropagated" gradients $J_{i_1}^\top e_{j_1}$ and $J_{i_2}^\top e_{j_2}$, and hence can be computed efficiently. [1]

5.4 Computational techniques

In this section, we present the major computational techniques used by our algorithm, and present their pseudo-codes.

5.4.1 Network computation (forward pass)

We have the Algorithm 17.

Algorithm 17 Forward Pass: Compute the neural network w.r.t. a single input x

Input: θ, x
Output: $\hat{y}$ or $h^{(l)}$, $v^{(l)}$ $(l = 1, ..., L)$
unpack θ to be $W^{(l)}$, $b^{(l)}$ $(l = 1, ..., L)$
$v^{(0)} = x$
for $l = 1, ..., L$ **do**
    $h^{(l)} = W^{(l)} v^{(l-1)} + b^{(l)}$
    $v^{(l)} = \phi^{(l)}(h^{(l)})$
end for
$\hat{y} = v^{(L)}$

5.4.2 Gradient computation (backward pass)

In order to compute the gradient $\nabla f(\theta)$, it suffices to compute $\nabla f_i(\theta)$ for $i = 1, ..., N$.

For $i = 1, ..., N$,

$$\nabla f_i(\theta) = \frac{\partial f_i(\theta)}{\partial \theta} = \frac{\partial \varepsilon(\hat{y}_i(\theta), y_i)}{\partial \theta} = \frac{\partial \varepsilon(\hat{y}_i(\theta), y_i)}{\partial \hat{y}_i(\theta)} \frac{\partial \hat{y}_i(\theta)}{\partial \theta} = \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i} \frac{\partial v_i^{(L)}}{\partial \theta}.$$

[1] There are other ways to compute and "invert" D_t, e.g., solving $D_t d_t = -J g_t$ by the linear conjugate gradient method as in Hessian-free, with either the explicit value of D_t, or an oracle to compute the product of D_t with an arbitrary vector. We tried both of these approaches and neither performed better that inverting D_t, i.e., computing d_t exactly.

Hence,

$$\frac{\partial f_i(\theta)}{\partial b^{(l)}} = \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial b^{(l)}} = \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}\frac{\partial h_i^{(L)}}{\partial v_i^{(L-1)}}\cdots\frac{\partial h_i^{(l+1)}}{\partial v_i^{(l)}}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}\frac{\partial h_i^{(l)}}{\partial b^{(l)}}$$
$$=\frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}W^{(L)}\cdots W^{(l+1)}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}. \tag{5.8}$$

Since

$$\frac{\partial h_i^{(l)}}{\partial \mathrm{vec}(W^{(l)})} = \left(\frac{\partial h_i^{(l)}}{\partial W_{:,1}^{(l)}}\cdots\frac{\partial h_i^{(l)}}{\partial W_{:,m_{l-1}}^{(l)}}\right) = \left((v_i^{(l-1)})_1 I_{m_l\times m_l} \quad \cdots \quad (v_i^{(l-1)})_{m_{l-1}} I_{m_l\times m_l}\right),$$

similarly,

$$\begin{aligned}\frac{\partial f_i(\theta)}{\partial \mathrm{vec}(W^{(l)})} =& \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}\frac{\partial h_i^{(L)}}{\partial v_i^{(L-1)}}\cdots\frac{\partial h_i^{(l+1)}}{\partial v_i^{(l)}}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}\frac{\partial h_i^{(l)}}{\partial \mathrm{vec}(W^{(l)})}\\ =& \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}W^{(L)}\cdots W^{(l+1)}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}\\ &\cdot\left((v_i^{(l-1)})_1 I_{m_l\times m_l} \quad \cdots \quad (v_i^{(l-1)})_{m_{l-1}} I_{m_l\times m_l}\right).\end{aligned}$$

Thus,

$$\frac{\partial f_i(\theta)}{\partial W^{(l)}} = \frac{\partial \varepsilon(\hat{y}_i, y_i)}{\partial \hat{y}_i}\frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}W^{(L)}\cdots W^{(l+1)}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}(v_i^{(l-1)})^T. \tag{5.9}$$

By (5.8) and (5.9), we have the following recursion

$$\begin{aligned}\frac{\partial f_i(\theta)}{\partial b^{(l)}} &= \frac{\partial f_i(\theta)}{\partial b^{(l+1)}}W^{(l+1)}\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}},\\ \frac{\partial f_i(\theta)}{\partial W^{(l)}} &= \frac{\partial f_i(\theta)}{\partial b^{(l)}}(v_i^{(l-1)})^T.\end{aligned}$$

Combining all of the above yields Algorithm 18.

Algorithm 18 Backward Pass: Compute the gradient $\nabla f_i(\theta)$

Input: θ, $h^{(l)}$, $v^{(l)}$ $(l = 1, ..., L)$, x, y
Output: $\nabla f_i(\theta)$
unpack θ to be $W^{(l)}$, $b^{(l)}$ $(l = 1, ..., L)$
$g^{(L)} = \frac{\partial \varepsilon(v^{(L)}, y)}{\partial \hat{y}} \frac{\partial v_i^{(L)}}{\partial h_i^{(L)}}$
$b_1^{(L)} = g^{(L)}$
$W_1^{(L)} = g^{(L)}(v^{(L-1)})^T$
for $l = L - 1, ..., 1$ **do**
    $g^{(l)} = g^{(l+1)} W^{(l+1)} \frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}$
    $b_1^{(l)} = g^{(l)}$
    $W_1^{(l)} = g^{(l)}(v^{(l-1)})^T$
end for
pack $W_1^{(l)}$, $b_1^{(l)}$ $(l = 1, ..., L)$ to be $\nabla f_i(\theta)$
return $\nabla f_i(\theta)$

5.4.3 J_i

Although we do not explicitly compute J_i in our algorithms, deriving an expression for J_i will help us in deriving expressions for the quantities we need.

Noticing that $J_i = \frac{\partial \hat{y}_i}{\partial \theta} = \frac{\partial v_i^{(L)}}{\partial \theta}$, we'd like to get an recursion w.r.t. $\frac{\partial v_i^{(0)}}{\partial \theta}$, ..., $\frac{\partial v_i^{(L)}}{\partial \theta}$. Because $v_i^{(0)} \equiv x_i$, we have that $\frac{\partial v_i^{(0)}}{\partial \theta} = 0$. For $l = 1, ..., L$,

$$\begin{aligned} \frac{\partial h_i^{(l)}}{\partial \theta} &= \frac{\partial \left(W^{(l)} v_i^{(l-1)} + b^{(l)} \right)}{\partial \theta} = \frac{\partial W^{(l)}}{\partial \theta} v_i^{(l-1)} + W^{(l)} \frac{\partial v_i^{(l-1)}}{\partial \theta} + \frac{\partial b^{(l)}}{\partial \theta}, \\ \frac{\partial v_i^{(l)}}{\partial \theta} &= \frac{\partial v_i^{(l)}}{\partial h_i^{(l)}} \frac{\partial h_i^{(l)}}{\partial \theta}, \end{aligned} \tag{5.10}$$

where $\frac{\partial W^{(l)}}{\partial \theta}$, $\frac{\partial b^{(l)}}{\partial \theta}$ are some abstract notions that will be specified later.

5.4.4 $J_i\theta_1$

We use the subscript 1 to denote the directional derivative of some variables as a function of θ along the direction θ_1. Because $J_i = \frac{\partial v_i^{(L)}}{\partial \theta}$, we have that

$$J_i\theta_1 = \frac{\partial v_i^{(L)}}{\partial \theta}\theta_1 = v_{i,1}^{(L)}.$$

We can also decompose θ_1 into $\text{vec}(W_1^{(l)})$ (hence, $W_1^{(l)}$) and $b_1^{(l)}$ (for all $l = 1, ..., L$), which agrees with the directional derivative notation.

Note that $v_{i,1}^{(0)} = \frac{\partial v_i^{(0)}}{\partial \theta}\theta_1 = 0$. Then, recursively, by (5.10), for $l = 1, ..., L$,

$$\begin{aligned}
h_{i,1}^{(l)} &= \frac{\partial h_i^{(l)}}{\partial \theta}\theta_1 = \left(\frac{\partial W^{(l)}}{\partial \theta}v_i^{(l-1)} + W^{(l)}\frac{\partial v_i^{(l-1)}}{\partial \theta} + \frac{\partial b^{(l)}}{\partial \theta}\right)\theta_1 \\
&= \frac{\partial W^{(l)}}{\partial \theta}\theta_1 v_i^{(l-1)} + W^{(l)}\frac{\partial v_i^{(l-1)}}{\partial \theta}\theta_1 + \frac{\partial b^{(l)}}{\partial \theta}\theta_1 = W_1^{(l)}v_i^{(l-1)} + W^{(l)}v_{i,1}^{(l-1)} + b_1^{(l)}, \\
v_{i,1}^{(l)} &= \frac{\partial v_i^{(l)}}{\partial \theta}\theta_1 = \frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}\frac{\partial h_i^{(l)}}{\partial \theta}\theta_1 = \frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}h_{i,1}^{(l)}.
\end{aligned} \tag{5.11}$$

This leads to Algorithm 19.

Algorithm 19 Compute the product of J_i and a vector θ_1

Input: θ_1, θ, h, v
Output: $J_i\theta_1$
unpack θ to be $W^{(l)}$, $b^{(l)}$ ($l = 1, ..., L$)
unpack θ_1 to be $W_1^{(l)}$, $b_1^{(l)}$ ($l = 1, ..., L$)
$v_1^{(0)} = 0$
for $l = 1, ..., L$ **do**
    $h_1^{(l)} = W_1^{(l)}v^{(l-1)} + W^{(l)}v_1^{(l-1)} + b_1^{(l)}$
    $v_1^{(l)} = \frac{\partial v^{(l)}}{\partial h^{(l)}}h_1^{(l)}$
end for
return $v_1^{(L)}$

5.4.5 $J_i^T x$

The idea behind computing $J_i^T x$ (x being an arbitrary vector) is even more tricky than $J_i \theta_1$. For given J_i and x, we define $s(\theta_2) = \theta_2^T (J_i^T x) = (J_i \theta_2)^T x = (v_{i,2}^{(L)})^T x$.

We denote the transpose of the partial derivative of s w.r.t. a variable by adding a hat on the variable, e.g, $\hat{\theta}_2 = \left(\frac{\partial s}{\partial \theta_2}\right)^T$. Because $\hat{\theta}_2 = J_i^T x$, it suffices to compute

$$\hat{\theta}_2 = (\text{vec}\left(\hat{W}_2^{(1)}\right)^T, \left(\hat{b}_2^{(1)}\right)^T, \cdots, \text{vec}\left(\hat{W}_2^{(L)}\right)^T, \left(\hat{b}_2^{(L)}\right)^T)^T.$$

Notice that $\hat{v}_{i,2}^{(L)} = x$, which is given. For $l = L, L-1, ..., 2$, when $\hat{v}_{i,2}^{(l)}$ is given, by (5.11), we have that

$$\begin{aligned}
v_{i,2}^{(l)} &= \frac{\partial v_i^{(l)}}{\partial h_i^{(l)}} h_{i,2}^{(l)} \\
\Rightarrow \hat{h}_{i,2}^{(l)} &= \left(\frac{\partial s}{\partial h_{i,2}^{(l)}}\right)^T = \left(\frac{\partial s}{\partial v_{i,2}^{(l)}} \frac{\partial v_{i,2}^{(l)}}{\partial h_{i,2}^{(l)}}\right)^T = \left(\frac{\partial v_{i,2}^{(l)}}{\partial h_{i,2}^{(l)}}\right)^T \hat{v}_{i,2}^{(l)} \\
h_{i,2}^{(l)} &= W_2^{(l)} v_i^{(l-1)} + W^{(l)} v_{i,2}^{(l-1)} + b_2^{(l)} \\
\Rightarrow \hat{v}_{i,2}^{(l-1)} &= \left(\frac{\partial s}{\partial v_{i,2}^{(l-1)}}\right)^T = \left(\frac{\partial s}{\partial h_{i,2}^{(l)}} \frac{\partial h_{i,2}^{(l)}}{\partial v_{i,2}^{(l-1)}}\right)^T = \left(W^{(l)}\right)^T \hat{h}_{i,2}^{(l)}
\end{aligned}$$

$$\hat{W}_2^{(l)} = \text{vec}^{-1}\left(\widehat{\text{vec}\left(W_2^{(l)}\right)}\right) = \text{vec}^{-1}\left(\left(\frac{\partial s}{\partial \text{vec}\left(W_2^{(l)}\right)}\right)^T\right) = \text{vec}^{-1}\left(\left(\frac{\partial s}{\partial h_{i,2}^{(l)}}\frac{\partial h_{i,2}^{(l)}}{\partial \text{vec}\left(W_2^{(l)}\right)}\right)^T\right)$$

$$= \text{vec}^{-1}\left(\left(\frac{\partial h_{i,2}^{(l)}}{\partial \text{vec}\left(W_2^{(l)}\right)}\right)^T\left(\frac{\partial s}{\partial h_{i,2}^{(l)}}\right)^T\right) = \text{vec}^{-1}\left(\left(\left(v_i^{(l-1)}\right)^T \otimes I_{m_l \times m_l}\right)^T \hat{h}_{i,2}^{(l)}\right)$$

$$= \text{vec}^{-1}\left(\left(v_i^{(l-1)} \otimes I_{m_l \times m_l}\right)\hat{h}_{i,2}^{(l)}\right) = \hat{h}_{i,2}^{(l)}\left(v_i^{(l-1)}\right)^T$$

$$\hat{b}_2^{(l)} = \left(\frac{\partial s}{\partial b_2^{(l)}}\right)^T = \left(\frac{\partial s}{\partial h_{i,2}^{(l)}}\frac{\partial h_{i,2}^{(l)}}{\partial b_2^{(l)}}\right)^T = \hat{h}_{i,2}^{(l)},$$

where $\text{vec}^{-1}()$ is the inverse map of the "vectorization" map vec().

Then, we have Algorithm 20.

Algorithm 20 Compute the product of J_i^T and a vector x

Input: x, θ, h, v
Output: $J_i^T x$ or $\hat{h}_2^{(l)}$ $(l = 1, ..., L)$
unpack θ to be $W^{(l)}$, $b^{(l)}$ $(l = 1, ..., L)$
$\hat{v}_2^{(L)} = x$
for $l = L, ..., 1$ **do**
    $\hat{h}_2^{(l)} = \left(\frac{\partial v^{(l)}}{\partial h^{(l)}}\right)^T \hat{v}_2^{(l)}$
    $\hat{v}_2^{(l-1)} = (W^{(l)})^T \hat{h}_2^{(l)}$
    $\hat{W}_2^{(l)} = \hat{h}_2^{(l)}(v^{(l-1)})^T$
    $\hat{b}_2^{(l)} = \hat{h}_2^{(l)}$
end for
pack $\hat{W}_2^{(l)}$, $\hat{b}_2^{(l)}$ $(l = 1, ..., L)$ to be $J_i^T x$

Note that we have an option of outputting $J_i^T x$ or $\hat{h}_2^{(l)}$ $(l = 1, ..., L)$. In the latter case (partial-computing mode), some operations can be skipped to save time.

Computing $J_i^T V$

We present the algorithm for computing $J_i^T V$, where $V \in R^{m_L}$ is an arbitrary vector whose dimension matches the column dimension of J_i^T. The vector $J_i^T V$ is of length n, which corre-

sponds to the parameters θ of the neural network. We use $\hat{W}_2^{(l)}$ and $\hat{b}_2^{(l)}$ to denote the part in $J_i^T V$ corresponding to the part $W^{(l)}$ and $b^{(l)}$ in θ, for $l = 1, ..., L$.

Algorithm 21 Compute $J_i^T V$ by backpropagation

1: $\hat{v}_2^{(L)} = V$
2: **for** $l = L, ..., 1$ **do**
3: $\quad \hat{h}_2^{(l)} = \left(\frac{\partial v_i^{(l)}}{\partial h_i^{(l)}}\right)^T \hat{v}_2^{(l)}$
4: $\quad \hat{v}_2^{(l-1)} = (W^{(l)})^T \hat{h}_2^{(l)}$
5: $\quad \hat{W}_2^{(l)} = \hat{h}_2^{(l)} (v_i^{(l-1)})^T$
6: $\quad \hat{b}_2^{(l)} = \hat{h}_2^{(l)}$
7: **end for**

We can compute $J_i^T V$ by a backpropagation, described in Algorithm 21 in $O(n)$ time. From Algorithm 21, it is clear that the part of $J_i^T V$ that corresponds to a $W^{(l)}$ is the outer product of two vectors, which can be expressed as the Kronecker product of a column vector with a row vector. This observation was also made in Martens and Grosse [62] and Botev *et al.* [15] and can be useful when we compute $J_{i_1} J_{i_2}^T$, as shown in Section 5.3.3.

5.4.6 $(J_{i_1}^T x_1)^T J_{i_2}^T x_2$

The straightforward way to form $(J_{i_1}^T x_1)^T J_{i_2}^T x_2$ is to compute both $J_{i_1}^T x_1$ and $J_{i_2}^T x_2$ using Algorithm 20, and then compute their dot product. We now present a much more efficient way to do this. In the following, we use superscripts to distinguish variables associated with $J_{i_1}^T x_1$ and $J_{i_2}^T x_2$. Since $\text{vec}(a_1 b_1^T)^T \text{vec}(a_2 b_2^T) = (b_1 \otimes a_1)^T (b_2 \otimes a_2) = (b_1^T \otimes a_1^T)(b_2 \otimes a_2) = (b_1^T b_2) \otimes (a_1^T a_2) = (b_1^T b_2)(a_1^T a_2)$, we have that

$$
\begin{aligned}
(J_{i_1}^T x_1)^T J_{i_2}^T x_2 &= (\hat{\theta}_1^{(1)})^T \hat{\theta}_2^{(2)} = \sum_{l=1}^{L} \left[\left(\text{vec} \left(\hat{W}_1^{(l),(1)} \right) \right)^T \text{vec} \left(\hat{W}_2^{(l),(2)} \right) + (\hat{b}_1^{(l),(1)})^T \hat{b}_2^{(l),(2)} \right] \\
&= \sum_{l=1}^{L} \left[\left(\text{vec} \left(\hat{h}_{i_1,2}^{(l)} (v_{i_1}^{(l-1)})^T \right) \right)^T \text{vec} \left(\hat{h}_{i_2,2}^{(l)} (v_{i_2}^{(l-1)})^T \right) + (\hat{h}_{i_1,2}^{(l)})^T \hat{h}_{i_2,2}^{(l)} \right] \\
&= \sum_{l=1}^{L} \left((v_{i_1}^{(l-1)})^T v_{i_2}^{(l-1)} + 1 \right) \cdot \left((\hat{h}_{i_1,2}^{(l)})^T \hat{h}_{i_2,2}^{(l)} \right).
\end{aligned}
$$

Hence, we can compute $(J_{i_1}^T x_1)^T J_{i_2}^T x_2$ without actually forming these two vectors. On the contrary, we can simply use the vectors $\hat{h}_{i,2}^{(l)}$ and $v_i^{(l)}$ (defined in Section 5.4.5).

5.4.7 $\left(J_{i_1} J_{i_2}^T\right)_{i_1,i_2=1,...,N}$

First, consider computing a single matrix $J_{i_1} J_{i_2}^T$ for $i_1, i_2 = 1, ..., N$. If we denote $V_{i_1,i_2}^{(l-1)} = (v_{i_1}^{(l-1)})^T v_{i_2}^{(l-1)} + 1$, the (j_1, j_2)-th element of it is computed as

$$e_{j_1}^T J_{i_1} J_{i_2}^T e_{j_2} = (J_{i_1}^T e_{j_1})^T J_{i_2}^T e_{j_2} = \sum_{l=1}^{L} \left((v_{i_1}^{(l-1)})^T v_{i_2}^{(l-1)} + 1\right) \left((\hat{h}_{i_1,2}^{(l),(j_1)})^T \hat{h}_{i_2,2}^{(l),(j_2)}\right)$$
$$= \sum_{l=1}^{L} V_{i_1,i_2}^{(l-1)} \left((\hat{h}_{i_1,2}^{(l),(j_1)})^T \hat{h}_{i_2,2}^{(l),(j_2)}\right).$$

Furthermore, if we denote $\hat{H}_{i,2}^{(l)} = \left(\hat{h}_{i,2}^{(l),(1)} \quad \dots \quad \hat{h}_{i,2}^{(l),(m_L)}\right)$, we have that

$$J_{i_1}^T J_{i_2} = \sum_{l=1}^{L} V_{i_1,i_2}^{(l-1)} (\hat{H}_{i_1,2}^{(l)})^T \hat{H}_{i_2,2}^{(l)}.$$

Furthermore, when computing B, we can use the following shortcut:

$$\begin{pmatrix} J_1 J_1^T & \cdots & J_1 J_N^T \\ \cdots & \cdots & \cdots \\ J_N J_1^T & \cdots & J_N J_N^T \end{pmatrix} = \left(J_{i_1} J_{i_2}^T \right)_{i_1,i_2=1,...,N} = \left(\sum_{l=1}^{L} V_{i_1,i_2}^{(l-1)} (\hat{H}_{i_1,2}^{(l)})^T \hat{H}_{i_2,2}^{(l)} \right)_{i_1,i_2=1,...,N}$$
$$= \sum_{l=1}^{L} \left(V_{i_1,i_2}^{(l-1)} (\hat{H}_{i_1,2}^{(l)})^T \hat{H}_{i_2,2}^{(l)} \right)_{i_1,i_2=1,...,N}$$
$$= \sum_{l=1}^{L} \left(V_{i_1,i_2}^{(l-1)} 1_{m_L \times m_L} \right)_{i_1,i_2=1,...,N} \odot \left((\hat{H}_{i_1,2}^{(l)})^T \hat{H}_{i_2,2}^{(l)} \right)_{i_1,i_2=1,...,N}$$

(where $\odot$ denotes pointwise multiplication, and $1_{m \times m}$ denotes an $m \times m$ matrix of all ones)

$$= \sum_{l=1}^{L} \left(V^{(l-1)} \otimes 1_{m_L \times m_L}\right) \odot \left((\hat{H}_2^{(l)})^T \hat{H}_2^{(l)}\right)$$

(let $\hat{H}_2^{(l)} = \left(\hat{H}_{1,2}^{(l)} \quad \dots \quad \hat{H}_{N,2}^{(l)}\right)$, let $V^{(l)} = \left(V_{i_1,i_2}^{(l)} \right)_{i_1,i_2=1,...,N}$)

The cost of computing the above expression is $O(Lm_L^2N^2+\sum_{l=1}^N \left(m_l m_L^2 N^2 + m_L^2 N^2 + m_{l-1}N^2\right)) = O(m_L^2 N^2 \sum_{l=1}^N m_l)$.

Then, we have Algorithm 22.

Algorithm 22 Compute a $|S| \times |S|$ block matrix $\left(J_{i_1} J_{i_2}^T\right)_{i_1,i_2 \in S}$

Input: θ, h_i, v_i $(i \in S)$, S
Output: $\left(J_{i_1} J_{i_2}^T\right)_{i_1,i_2 \in S}$
for $i \in S$ **do**
  for $j = 1, ..., m_L$ **do**
    $\left(\hat{h}_{i,2}^{(l),(j)} \right)_{l=1,...,L}$ = Compute_J_transpose_V(e_j, θ, h_i, v_i) (partly-computing mode)
    {see Algorithm 20}
  end for
end for
for $l = 1, ..., L$ **do**
  for $i \in S$ **do**
    $\tilde{v}_i^{(l-1)} = \begin{pmatrix} v_i^{(l-1)} \\ 1 \end{pmatrix}$
    $\hat{H}_{i,2}^{(l)} = \left(\hat{h}_2^{(l),(1)} \quad \cdots \quad \hat{h}_2^{(l),(m_L)} \right)$
  end for
  $\tilde{v}^{(l-1)} = \left(\tilde{v}_i^{(l-1)}\right)_{i \in S}$ {arranged in a row}
  $\hat{H}_2^{(l)} = \left(\hat{H}_{i,2}^{(l)}\right)_{i \in S}$ {arranged in a row}
  $B_l = \left(\left((\tilde{v}^{(l-1)})^T \tilde{v}^{(l-1)} \right) \otimes 1_{m_L \times m_L} \right) \odot \left((\hat{H}_2^{(l)})^T \hat{H}_2^{(l)} \right)$
end for
return $\sum_{l=1}^L B_l$

5.5 Algorithm for Subsampled Second-Order Methods

In this section, we summarize our subsampled GN and NG methods. Since we are focused on very large data sets, we estimate the gradient $\nabla f()$, and $f()$ in the reduction ratio ρ_t (see (5.5)) using a mini-batch S_1^t.

The above algorithm works for both the GN and NG methods, the only differences being in computing and inverting D_t and the backpropagations needed for computing $G_t^{-1} g_t$.

Algorithm 23 Sub-sampled Gauss-Newton / Natural Gradient method

1: **Parameters:** N_1, N_2, $0 < \epsilon < \frac{1}{2}$, learning rate α
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Randomly select a mini-batch $S_1^t \subseteq [N]$ of size N_1 and $S_2^t \subseteq S_1^t$ of size N_2
4: Compute $g_t = \frac{1}{|S_1^t|} \sum_{i \in S_1^t} \nabla f_i(\theta^{(t)})$
5: Compute D_t and $p_t = -G_t^{-1} g_t$ as in (5.6) or (5.7) with mini-batch S_2^t
6: Update λ using the LM style rule { see Section 5.2.3 } with S_1^t mini-batch estimates of $f()$ to compute ρ_t in (5.5)
7: set $\theta^{(t+1)} = \theta^{(t)} + \alpha \cdot p_t$
8: **end for**

5.6 Convergence

Recall that the LM direction that we compute is $p_t = -(B_t + \lambda_t I)^{-1} g_t$. If we let $\Delta_t = ||p_t||$, it is well known that p_t is the global solution to the trust-region (TR) problem

$$\min_p m_t(p) \text{ s.t. } ||p|| \leq \Delta_t.$$

As in the classical TR method, we evaluate the quality of the quadratic model $m_t(\cdot)$ by computing ρ_t defined by (5.5). However, while the classical TR method updates Δ_t depending on the value of ρ_t, we follow the LM approach of updating λ_t instead. Loosely speaking, there is a reciprocal-like relation between λ_t and Δ_t. While Martens [59] proposed this way of updating λ_t as a "heuristic", we are able to show that Algorithm 23, with a exact (full) gradient (i.e., $N_1 = N$) and only updating θ_t when ρ_t is above a certain threshold (say η), converges to a stationary point under the following assumptions:

Assumption 11. $||B_t|| \leq \beta$.

Assumption 12. $||\nabla^2 f(\theta)|| \leq \beta_1$.

Our proof is similar to that used to prove convergence of the standard trust-region method (e.g., see Nocedal and Wright [67]), and in particular makes use of the following:

Lemma 11. *Under Assumption 11, there exists a constant $c_1 > 0$ such that*

$$m_t(0) - m_t(p_t) \geq c_1 ||g_t|| \Delta_t.$$

Proof. Because $p_t = -(B_t + \lambda_t I)^{-1} g_t$, $-g_t^T p_t = p_t^T (B_t + \lambda_t I) p_t$. Then since $B_t \geq 0$, we have

$$m_t(0) - m_t(p_t) = -g_t^T p_t - \frac{1}{2} p_t^T B_t p_t \geq \lambda_t ||p_t||^2.$$

On the other hand, since $\frac{\lambda_t}{\beta + \lambda_t} \geq \frac{\tau}{\beta + \tau} = c_1 > 0$, $||B_t|| \leq \beta$ and $\Delta_t = ||p_t||$,

$$c_1 ||g_t|| \Delta_t = c_1 || - (B_t + \lambda_t I) p_t || || p_t || \leq c_1 (\beta + \lambda_t) ||p_t||^2 \leq \lambda_t ||p_t||^2.$$

□

Using Lemma 11, we now prove the global convergence of the full-gradient variant of Algorithm 23:

Theorem 6. *Suppose in Algorithm 23, we set $N_1 = N$, $\alpha = 1$ and only update θ_t when $\rho_t \geq \eta$ where $0 < \eta < \epsilon$. Then, under Assumptions 11 and 12, if f is bounded below. we have that* $\lim_{t \to \infty} ||g_t|| = 0$.

Proof. We first show that λ_t is bounded above by some constant Λ_1: Recalling (5.5), at iteration t, we have

$$|\rho_t - 1| = \left| \frac{m_t(p_t) - f(\theta^{(t)} + p_t)}{m_t(0) - m_t(p_t)} \right|. \tag{5.12}$$

By Taylor's theorem, $f(\theta^{(t)} + p_t) = f(\theta^{(t)}) + \nabla f(\theta^{(t)})^T p_t + \frac{1}{2} p_t^T \nabla^2 f(\theta^{(t)} + \mu p_t) p_t$ for some $\mu \in (0, 1)$. Hence,

$$\left| m_t(p_t) - f(\theta^{(t)} + p_t) \right| = \left| \frac{1}{2} p_t^T B_t p_t - \frac{1}{2} p_t^T \nabla^2 f(\theta^{(t)} + \mu p_t) p_t \right| \leq \frac{1}{2} (\beta + \beta_1) ||p_t||^2. \tag{5.13}$$

By (5.12) and (5.13), we have that

$$|\rho_t - 1| \leq \frac{\frac{1}{2}(\beta + \beta_1)||p_t||^2}{\lambda_t||p_t||^2} = \frac{\frac{1}{2}(\beta + \beta_1)}{\lambda_t}.$$

Hence, there exists a $\Lambda > 0$ such that for all $\lambda_t \geq \Lambda$, we have $|\rho_t - 1| \leq \epsilon$, and thus, $\rho_t \geq 1 - \epsilon$. Consequently, by the way λ_{LM} is updated, for all t, $\lambda_t \leq boost \cdot \Lambda + \tau = \Lambda_1$; i.e., λ_t is bounded above. By Assumption 11, $||B_t + \lambda_t I||$ is also bounded, i.e., $||B_t + \lambda_t I|| \leq \beta + \Lambda_1$. Hence, the minimum eigenvalue of $(B_t + \lambda_t)^{-1}$ is no less than $\frac{1}{\beta + \Lambda_1}$. Finally, by the Cauchy-Schwarz inequality and the fact that $\Delta_t = ||p_t|| = ||-(B_t + \lambda_t I)^{-1} g_t||$, we have that $||g_t||\Delta_t \geq ||g_t(B_t + \lambda_t I)^{-1} g_t|| \geq \frac{1}{\beta + \Lambda_1}||g_t||^2$.

Let $T_1 = \{t = 0, 1, ... \mid \rho_t \geq \eta\}$ denote the set of indices t such that step p_t is accepted. For any $t \in T_1$, by definition of ρ_t and Lemma 11,

$$f(\theta^{(t)}) - f(\theta^{(t+1)}) > \eta c_1 ||g_t||\Delta_t \geq \eta c_1 \frac{1}{\beta + \Lambda_1}||g_t||^2. \tag{5.14}$$

We now show that $|T_1| = \infty$ (unless for some t, $g_t = 0$ and Algorithm 23 stops finitely): Suppose that this is not the case. Then there exists a $T > 0$ such that for all $t \geq T$, p_t is rejected (i.e., $\rho_t \leq \eta < \epsilon$). Then, $\lambda_t \to \infty$, contradicting the fact that λ_t is bounded. Because $|T_1| = \infty$, $\lim_{t \to \infty} ||g_t|| = \lim_{t \in T_1} ||g_t||$. Because f is bounded below and $f(\theta^{(t)})$ is non-increasing, the left-hand-side of (5.14) goes to zero. Hence, the right-hand-side also goes to zero, which implies $\lim_{t \in T_1} ||g_t|| = 0$. □

5.7 Computational Costs of Proposed Algorithms

In this section we discuss the computational cost of our SMW-based GN and NG algorithms, and compare them with SGD, Hessian-free (HF) and KFAC. First, several basic operations including computing $f_i(\theta)$, $\nabla f_i(\theta)$, JV and $J^T v$ all requires $O(n)$ time for a single data point. Hence, all algorithms have a cost of $O(N_1 n)$ for computing the stochastic gradient g_t.

For the second order methods, the following table summarizes the extra costs for computing the LM direction p_t, where n_{HF} denotes the number of CG iterations used in Hessian-free.

Algorithm	Cost
SMW-GN	$O(m_L N_2 n + m_L^2 N_2^2 \sum_l m_l + m_L^3 N_2^3)$
SMW-NG	$O(N_2 n + N_2^2 \sum_l m_l + N_2^3)$
HF	$n_{\text{HF}} \times O(N_2 n)$
KFAC	$O(\sum_l m_l^3 + N_2 \sum_l m_l^2)$

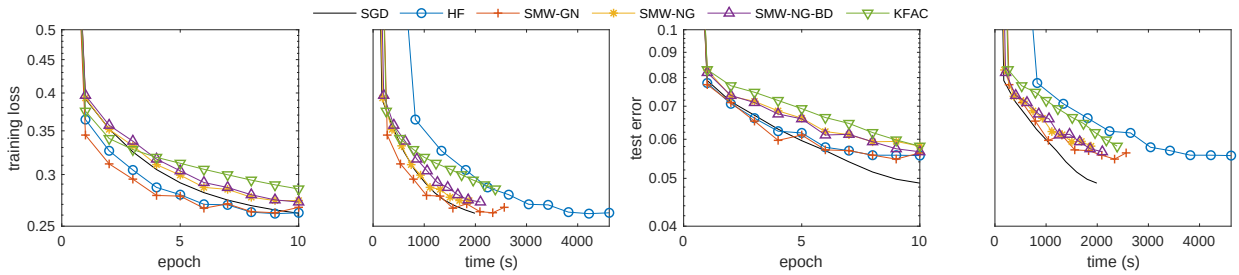


Figure 5.1: Results of SMW-GN, SMW-NG, SMW-NG-BD, KFAC, HF, and SGD on a MNIST classification problem

5.7.1 Comparison Between Algorithms

Since n is usually extremely large in NNs, we see that in SMW-GN the multiplier of the term involving n is reduced from $n_{\text{HF}} N_2$ in HF to $m_L N_2$. KFAC has a term proportional to $\sum_l m_l^3$, which is of an even higher order than n.

For all of the second-order methods, when $N_2 \ll N_1$, the overhead for each iteration is usually compensated for by the better direction generated by these methods for updating the parameters. However, even if the condition $N_2 \ll N_1$ is not met, as long as N_2 is reasonably small, the overhead is controllable. Consequently, one should choose a relatively small N_2 when implementing our SMW-based algorithms.

5.8 Numerical Experiments

We compared our algorithms SMW-GN and SMW-NG with SGD, HF and KFAC. The KFAC algorithm was implemented using block-diagonal approximaton, without re-scaling and momentum (see section 6.4 and 7 of Martens and Grosse [62]). We also included a block diagonal version

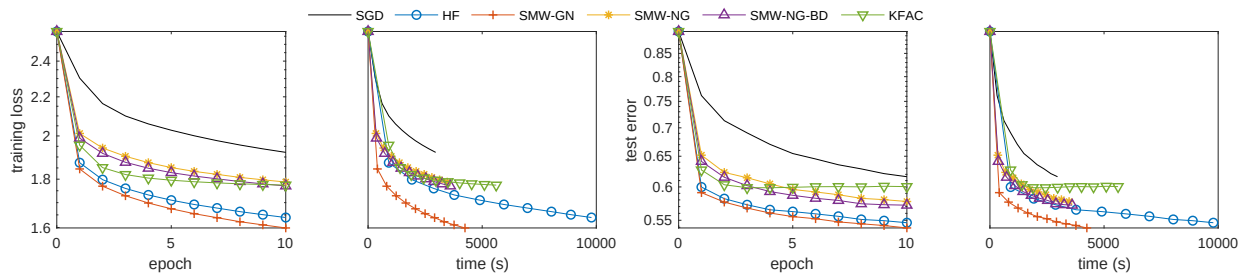


Figure 5.2: Results of SMW-GN, SMW-NG, SMW-NG-BD, KFAC, HF, and SGD on a CIFAR classification problem

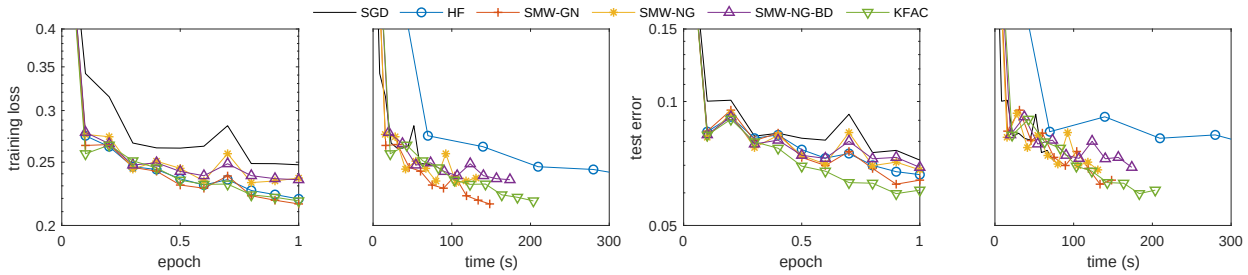


Figure 5.3: Results of SMW-GN, SMW-NG, SMW-NG-BD, KFAC, HF, and SGD on a webspam classification problem

of SMW-NG, namely, SMW-NG-BD, where in each block, the matrix is inverted by SMW, in order to mimic the block diagonal approximation used by KFAC.

For all of the experiments reported in this section, we set the initial value of λ_{LM} to be 1, $boost = 1.01$, $drop = 0.99$, $\epsilon = 1/4$, $\tau = 0.001$, same as in Kiros [47]. All algorithms were implemented in MATLAB R2019a and run on an Intel Core i5 processor. We tested the performance of the algorithms on several classification problems. We reported both training loss and testing error. The data sets were scaled to have zero means and unit variances.

MNIST [51]: The training set is of size $N = 6 \times 10^4$. We used a NN with one hidden layer of size 500 and logistic activation, i.e., $(m_0, m_1, m_2) = (784, 500, 10)$, where the first and last layers are the size of input and output. The output layer was softmax with cross entropy. We use $N_1 = 60$, $N_2 = 30$. The learning rate for SGD was set to be 0.1, tuned from $\{0.01, 0.05, 0.1, 0.5\}$. Initial learning rates for other methods were also set to be 0.1 for purposes of comparison. We did not tune the learning rates for second-order methods because they adaptively modify the rate by

updating λ_{LM} as they proceed. We ran each algorithm for 10 epochs. The results are presented in Figure 5.1.

CIFAR-10 [48]: The training set is of size $N = 5 \times 10^4$. We used a NN with two hidden layers of size 400 and logistic activation, i.e., $(m_0, m_1, m_2, m_3) = (3072, 400, 400, 10)$. The output layer was softmax with cross entropy. We use $N_1 = 100$, $N_2 = 50$. The learning rate was set to be 0.01. We ran each algorithm for 10 epochs. The results are presented in Figure 5.2.

webspam [22]: The training set is of size $N = 3 \times 10^5$. We used a NN with two hidden layers of size 400 and logistic activation, i.e., $(m_0, m_1, m_2, m_3) = (254, 400, 400, 1)$. Because this is a binary classification, we set the output layer to be logistic with binary cross entropy. We use $N_1 = 60$, $N_2 = 30$. We tuned the learning rate for SGD and initial learning rates for other algorithms separately, which turned out to be 0.4 for SGD, 0.05 for HF and SMW-GN, and 0.1 for SMW-NG, SMW-NG, and KFAC. The results shown in Figure 5.3 all used their corresponding best learning rates. We ran each algorithm for 1 epoch.

5.8.1 Discussion of results

Interestingly, the relative ranking of the algorithms changes from one problem to another, indicating that the relative performance of the algorithms depends upon the data set, structure of the NN, and parameter settings.

From our experimental results, we see that SMW-GN is always faster than HF in terms of both epochs and clock-time, which is consistent with our analysis above. KFAC sometimes performs very well, not surprisingly, because it accumulates more and more curvature information with each new mini-batch. But it also slows down considerably when the NN has wide layers (see Figure 5.2). Moreover, the three experiments were done differently, mimicing the different practices used when training a NN model, namely, tuning learning rates for all algorithms, tuning learning rates for one algorithm and then using it for all, or simply choosing a conservative learning rate. After carefully tuning the learning rate, SGD can perform as well as second-order methods as shown in Figure 5.1. However, if learning rate is chosen to be more conservative or typical (e.g., 0.01), it

may suffer from slow convergence compared with second-order methods (see Figure 5.2). If we want to get lower training loss or testing error, we may have to run it for far more epochs / time.

The key take-away from our numerical results is that our SMW-based algorithms that are based on the Gauss-Newton and natural gradient methods are competitive with their Hessian-free and Kronecker-factor implementations, HF and KFAC, as well as SGD. In particular, SMW-GN performs extremely well without requiring any parameter tuning.

5.9 Summary and Future Research Directions

In this chapter, we proposed efficient LM-NG/GN methods for training neural networks, semi-stochastic versions of which are provably convergent, while fully stochastic versions are competitive with off-the-shelf algorithms including SGD and KFAC. A promising future research topic is the study of how to adapt gradient (diagonal) rescaling techniques like Adam [46] and AdaGrad [29], that are based on running averages of the first and second moments of the stochastic gradients encountered during the course of the algorithm, to our GN and NG algorithms. This is a challenging topic, since both NG and NG based algorithms already incorporate non-diagonal rescalings. A second promising future research topic is the study of how to, starting from relatively small gradient and GN and Fisher matrix mini-batches, increase their sizes as needed, by evaluating their variances (e.g., see Bollapragada *et al.* [13] and references therein). Finally, the structure of our algorithms is well-suited for parallel computation. Besides the common approach of distributing the data across different processors, one can compute terms involving D_t in parallel, so that the cost of second-order computations becomes comparable to that for evaluating gradients.

References

[1] S.-i. Amari, *Differential-geometrical methods in statistics*. Springer Science & Business Media, 2012, vol. 28.

[2] S.-I. Amari, "Natural gradient works efficiently in learning," *Neural computation*, vol. 10, no. 2, pp. 251–276, 1998.

[3] S.-I. Amari, H. Park, and K. Fukumizu, "Adaptive method of realizing natural gradient learning for multilayer perceptrons," *Neural computation*, vol. 12, no. 6, pp. 1399–1409, 2000.

[4] E. Amid, R. Anil, and M. K. Warmuth, "Locoprop: Enhancing backprop via local loss optimization," *arXiv preprint arXiv:2106.06199*, 2021.

[5] R. Anil, V. Gupta, T. Koren, K. Regan, and Y. Singer, "Scalable second order optimization for deep learning," *arXiv preprint arXiv:2002.09018*, 2021. arXiv: 2002.09018 [cs.LG].

[6] J. Ba, R. Grosse, and J. Martens, "Distributed second-order optimization using Kronecker-factored approximations," 2016.

[7] J. Ba, R. B. Grosse, and J. Martens, "Distributed second-order optimization using kronecker-factored approximations," in *ICLR*, 2017.

[8] H. Badreddine, S. Vandewalle, and J. Meyers, "Sequential quadratic programming (sqp) for optimal control in direct numerical simulation of turbulent flow," *Journal of Computational Physics*, vol. 256, pp. 1–16, 2014.

[9] A. Bahamou, D. Goldfarb, and Y. Ren, "A mini-block natural gradient method for deep neural networks," *arXiv preprint arXiv:2202.04124*, 2022.

[10] C. Bakker, M. J. Henry, and N. O. Hodas, "The outer product structure of neural network derivatives," *arXiv preprint arXiv:1810.03798*, 2018.

[11] A. Beck and Y. Vaisbourd, "Globally solving the trust region subproblem using simple first-order methods," *SIAM Journal on Optimization*, vol. 28, no. 3, pp. 1951–1967, 2018.

[12] A. Bernacchia, M. Lengyel, and G. Hennequin, "Exact natural gradient in deep linear networks and its application to the nonlinear case," in *Advances in Neural Information Processing Systems*, 2018, pp. 5941–5950.

[13] R. Bollapragada, R. Byrd, and J. Nocedal, "Adaptive sampling strategies for stochastic optimization," *SIAM Journal on Optimization*, vol. 28, no. 4, pp. 3312–3343, 2018.

[14] A. Bordes, L. Bottou, and P. Gallinari, "Sgd-qn: Careful quasi-newton stochastic gradient descent," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1737–1754, 2009.

[15] A. Botev, H. Ritter, and D. Barber, "Practical Gauss-Newton optimisation for deep learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 557–565.

[16] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[17] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms 1. general considerations," *IMA Journal of Applied Mathematics*, vol. 6, no. 1, pp. 76–90, 1970.

[18] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal, "On the use of stochastic hessian information in optimization methods for machine learning," *SIAM Journal on Optimization*, vol. 21, no. 3, pp. 977–995, 2011.

[19] R. H. Byrd, S. L. Hansen, J. Nocedal, and Y. Singer, "A stochastic quasi-Newton method for large-scale optimization," *SIAM Journal on Optimization*, vol. 26, no. 2, pp. 1008–1031, 2016.

[20] R. H. Byrd, J. Nocedal, and R. B. Schnabel, "Representations of quasi-Newton matrices and their use in limited memory methods," *Mathematical Programming*, vol. 63, no. 1-3, pp. 129–156, 1994.

[21] T. Cai *et al.*, "A gram-gauss-newton method learning overparameterized deep neural networks for regression problems," *arXiv preprint arXiv:1905.11675*, 2019.

[22] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, 27:1–27:27, 3 2011, Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[23] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," *arXiv preprint arXiv:1910.05446*, 2019.

[24] F. Dangel, P. Hennig, and S. Harmeling, "Modular block-diagonal curvature approximations for feedforward architectures," *arXiv preprint arXiv:1902.01813*, 2019.

[25] A. P. Dawid, "Some matrix-variate distribution theory: Notational considerations and a Bayesian application," *Biometrika*, vol. 68, no. 1, pp. 265–274, 1981.

[26] B. S. Dees and D. P. Mandic, "A statistically identifiable model for tensor-valued Gaussian random variables," *arXiv preprint arXiv:1911.02915*, 2019.

[27] G. Desjardins, K. Simonyan, R. Pascanu, *et al.*, "Natural neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2071–2079.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[29] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[30] P. Dutilleul, "The mle algorithm for the matrix normal distribution," *Journal of statistical computation and simulation*, vol. 64, no. 2, pp. 105–123, 1999.

[31] R. Fletcher, "A new approach to variable metric algorithms," *The computer journal*, vol. 13, no. 3, pp. 317–322, 1970.

[32] Y. Fujimoto and T. Ohira, "A neural network model with bidirectional whitening," in *International Conference on Artificial Intelligence and Soft Computing*, Springer, 2018, pp. 47–57.

[33] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a Kronecker factored eigenbasis," in *Advances in Neural Information Processing Systems*, 2018, pp. 9550–9560.

[34] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of computation*, vol. 24, no. 109, pp. 23–26, 1970.

[35] D. Goldfarb, Y. Ren, and A. Bahamou, "Practical quasi-Newton methods for training deep neural networks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 2386–2396.

[36] R. Gower, D. Goldfarb, and P. Richtárik, "Stochastic block BFGS: Squeezing more curvature out of data," in *International Conference on Machine Learning*, 2016, pp. 1869–1878.

[37] R. M. Gower and P. Richtárik, "Randomized quasi-newton updates are linearly convergent matrix inversion algorithms," *SIAM Journal on Matrix Analysis and Applications*, vol. 38, no. 4, pp. 1380–1409, 2017.

[38] R. Grosse and J. Martens, "A Kronecker-factored approximate fisher matrix for convolution layers," in *International Conference on Machine Learning*, 2016, pp. 573–582.

[39] V. Gupta, T. Koren, and Y. Singer, “Shampoo: Preconditioned stochastic tensor optimization,” in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1842–1850.

[40] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] T. Heskes, “On "natural" learning and pruning in multilayered perceptrons,” *Neural Computation*, vol. 12, Jan. 2000.

[42] G. Hinton, N. Srivastava, and K. Swersky, “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,” *Cited on*, vol. 14, no. 8, 2012.

[43] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[44] R. Johnson and T. Zhang, “Accelerating stochastic gradient descent using predictive variance reduction,” in *Advances in neural information processing systems*, 2013, pp. 315–323.

[45] M. E. Khan and H. Rue, “The Bayesian learning rule,” *arXiv preprint arXiv:2107.04562*, 2021.

[46] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, 2014.

[47] R. Kiros, “Training neural networks with stochastic hessian-free optimization,” *arXiv preprint arXiv:1301.3641*, 2013.

[48] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105.

[50] F. Kunstner, P. Hennig, and L. Balles, “Limitations of the empirical Fisher approximation for natural gradient descent,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019.

[51] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[52] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[53] W. Li and G. Montúfar, "Natural gradient via optimal transport," *Information Geometry*, vol. 1, no. 2, pp. 181–214, 2018.

[54] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.

[55] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2019.

[56] H. Lu, K. N. Plataniotis, and A. Venetsanopoulos, *Multilinear subspace learning: dimensionality reduction of multidimensional data*. CRC press, 2013.

[57] A. Lucchi, B. McWilliams, and T. Hofmann, "A variance reduced stochastic newton method," *arXiv preprint arXiv:1503.08316*, 2015.

[58] A. M. Manceur and P. Dutilleul, "Maximum likelihood estimation for the tensor normal distribution: Algorithm, minimum sample size, and empirical bias and dispersion," *Journal of Computational and Applied Mathematics*, vol. 239, pp. 37–49, 2013.

[59] J. Martens, "Deep learning via hessian-free optimization.," in *ICML*, vol. 27, 2010, pp. 735–742.

[60] ——, "New insights and perspectives on the natural gradient method," *Journal of Machine Learning Research*, vol. 21, no. 146, pp. 1–76, 2020.

[61] J. Martens, J. Ba, and M. Johnson, "Kronecker-factored curvature approximations for recurrent neural networks," in *International Conference on Learning Representations*, 2018.

[62] J. Martens and R. Grosse, "Optimizing neural networks with Kronecker-factored approximate curvature," in *International conference on machine learning*, 2015, pp. 2408–2417.

[63] A. Mokhtari and A. Ribeiro, "Global convergence of online limited memory bfgs," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 3151–3181, 2015.

[64] ——, "Res: Regularized stochastic bfgs algorithm," *IEEE Transactions on Signal Processing*, vol. 62, no. 23, pp. 6089–6104, 2014.

[65] J. J. Moré, "The levenberg-marquardt algorithm: Implementation and theory, numerical analysis," *Lecture notes in mathematics 630*, pp. 105–116, 1977.

[66] P. Moritz, R. Nishihara, and M. Jordan, "A linearly-convergent stochastic l-bfgs algorithm," in *Artificial Intelligence and Statistics*, 2016, pp. 249–258.

[67] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[68] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[69] M. J. Powell, "Algorithms for nonlinear constraints that use lagrangian functions," *Mathematical programming*, vol. 14, no. 1, pp. 224–248, 1978.

[70] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents," *arXiv preprint arXiv:2204.06125*, 2022.

[71] C. Rao, *Information and accuracy attainable in the estimation of statistical parameters. kotz s & johnson nl (eds.), breakthroughs in statistics volume i: Foundations and basic theory, 235–248*, 1945.

[72] Y. Ren and D. Goldfarb, "Efficient subsampled Gauss-Newton and natural gradient methods for training neural networks," *arXiv preprint arXiv:1906.02353*, 2019.

[73] ——, "Kronecker-factored quasi-Newton methods for convolutional neural networks," *arXiv preprint arXiv:2102.06737*, 2021.

[74] ——, "Tensor normal training for deep learning models," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021.

[75] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[76] N. L. Roux and A. W. Fitzgibbon, "A fast natural newton method," in *ICML*, 2010.

[77] C. W. Royer, M. O'Neill, and S. J. Wright, "A newton-cg algorithm with complexity guarantees for smooth unconstrained optimization," *Mathematical Programming*, vol. 180, no. 1, pp. 451–488, 2020.

[78] R. M. Schmidt, F. Schneider, and P. Hennig, "Descending through a crowded valley-benchmarking deep learning optimizers," in *International Conference on Machine Learning*, PMLR, 2021, pp. 9367–9376.

[79] N. N. Schraudolph, "Fast curvature matrix-vector products for second-order gradient descent," *Neural computation*, vol. 14, no. 7, pp. 1723–1738, 2002.

[80] N. N. Schraudolph, J. Yu, and S. Günter, "A stochastic quasi-newton method for online convex optimization," in *Artificial intelligence and statistics*, 2007, pp. 436–443.

[81] D. F. Shanno, "Conditioning of quasi-Newton methods for function minimization," *Mathematics of computation*, vol. 24, no. 111, pp. 647–656, 1970.

[82] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[83] M. Singull, M. R. Ahmad, and D. von Rosen, "More on the Kronecker structured covariance matrix," *Communications in Statistics-Theory and Methods*, vol. 41, no. 13-14, pp. 2512–2523, 2012.

[84] Z. Tang *et al.*, "Skfac: Training neural networks with faster kronecker-factored approximate curvature," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 13 479–13 487.

[85] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[86] O. Vinyals and D. Povey, "Krylov subspace descent for deep learning," in *Artificial Intelligence and Statistics*, 2012, pp. 1261–1268.

[87] X. Wang, S. Ma, D. Goldfarb, and W. Liu, "Stochastic quasi-Newton methods for nonconvex stochastic optimization," *SIAM Journal on Optimization*, vol. 27, no. 2, pp. 927–956, 2017.

[88] Y. Wang and W. Li, "Information newton's flow: Second-order optimization method in probability space," *arXiv preprint arXiv:2001.04341*, 2020.

[89] Y. Wu, X. Zhu, C. Wu, A. Wang, and R. Ge, "Dissecting hessian: Understanding common structure of hessian in neural networks," *arXiv preprint arXiv:2010.04261*, 2020.

[90] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," *Advances in neural information processing systems*, vol. 30, pp. 5279–5288, 2017.

[91] P. Xu, F. Roosta, and M. W. Mahoney, "Newton-type methods for non-convex optimization under inexact hessian information," *Mathematical Programming*, pp. 1–36, 2019.

[92] H. H. Yang and S.-i. Amari, "Complexity issues in natural gradient descent method for training multilayer perceptrons," *Neural Computation*, vol. 10, no. 8, pp. 2137–2157, 1998.

[93] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. W. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," *arXiv preprint arXiv:2006.00719*, 2020.

[94] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proceedings of the British Machine Vision Conference (BMVC)*, E. R. H. Richard C. Wilson and W. A. P. Smith, Eds., BMVA Press, 2016, pp. 87.1–87.12, ISBN: 1-901725-59-6.

[95] G. Zhang, J. Martens, and R. Grosse, "Fast convergence of natural gradient descent for overparameterized neural networks," *arXiv preprint arXiv:1905.10961*, 2019.

[96] G. Zhang, C. Wang, B. Xu, and R. Grosse, "Three mechanisms of weight decay regularization," in *International Conference on Learning Representations*, 2019.