論文 / 著書情報
Article / Book Information

題目(和文)	大規模深層学習のための二次最適化
Title(English)	Second-order Optimization for Large-scale Deep Learning
著者(和文)	大沢和樹
Author(English)	Kazuki Osawa
出典(和文)	学位:博士（工学）, 学位授与機関:東京工業大学, 報告番号:甲第11964号, 授与年月日:2021年3月26日, 学位の種別:課程博士, 審査員:横田 理央,篠田 浩一,岡﨑 直観,村田 剛志,下坂 正倫
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第11964号, Conferred date:2021/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Second-order Optimization for Large-scale Deep Learning

Kazuki Osawa
Department of Computer Science
School of Computing
Tokyo Institute of Technology

A thesis submitted for the degree of
Doctor of Engineering
2021

Tokyo Tech

Abstract

Information matrices that contain information about the curvature and noise have played a key role in mathematical optimization, statistical learning theory, and Bayesian statistics. In classical machine learning, principled matrix-based algorithms inspired by these fields have been developed for fast and robust learning and better model selection. In Deep Learning (DL), however, these information matrices are often approximated under strong assumptions that are not well-justified, as the underlying deep neural networks contain far more parameters than traditional machine learning models, and the matrix computations are expensive. When such approximations are used without quantitative analysis of the errors they introduce, it could lead to erroneous conclusions and paint only a limited part of the whole picture. Therefore, understanding the real benefits and approximability of the matrix-based algorithms has great potential for gaining insights on developing more principled and efficient learning methods in deep learning.

The primary goal of this research is to overcome the infeasibility of evaluating accurate information using High-Performance Computing (HPC) and realize principled matrix-based algorithms for DL. This thesis describes the advantages and computational difficulties of optimization methods with information matrices (i.e., second-order optimization methods) for "large-scale" DL involving a massive number of neurons and data, and proposes efficient algorithms and implementations in GPU supercomputers. We demonstrate that second-order optimization can leverage more parallelism than widely-used optimization methods (e.g., stochastic gradient descent). The research also explores this space based on different approaches to parallelism and data partitioning in distributed memory for fast and robust training. We also discuss generalization measures, that quantify the generalization error of deep neural network models, and approximate Bayesian inference using information matrices for better model selection in DL. For this purpose, an extension of second-order optimization methods for Bayesian DL and fast computation of information matrices in large-scale DL will be discussed. Experiments are conducted in a wide range of practical DL settings (e.g., ImageNet classification). The research aims to find a novel way to utilize HPC for DL research and generate a broad impact on both theoretical understanding and the development of practical and reliable learning methods in DL.

Acknowledgements

I am grateful to my adviser Rio Yokota for his constant support and guidance over the years, and for giving me the freedom and motivation to pursue my own research.

I have been fortunate to collaborate with great students, engineers, and researchers: Yuichiro Ueno, Yohei Tsuji, Akira Naruse, Chuan-Sheng Foo, Satoshi Matsuoka, Siddharth Swaroop, Anirudh Jain, Runa Eschenhagen, Richard E. Turner, Mohammad Emtiyaz Khan, Yaroslav Bulatov, and Ryo Karakida. My research has been supported by the help of the students in the Yokota laboratory: Hana Hoshino and Hikaru Nakata. I have been a Research Fellow of JSPS and my research has been supported by JSPS KAKENHI Grant Number JP19J13477.

The large-scale experiments in my research have been realized on the TSUBAME3.0 supercomputer at the Tokyo Institute of Technology and the ABCI supercomputer at the National Institute of Advanced Industrial Science and Technology, Japan. The Yokota laboratory's server systems have been a great help in the prototyping of the experiments. I would like to thank Hiroyuki Ootomo and Yuichiro Ueno, the server administrators, for their great efforts and contributions to a comfortable and stable server experience.

I have visited the Agency for Science, Technology and Research (A*STAR), Singapore, for a year during my doctoral course, where I have been advised by Chuan-Sheng Foo and supported by the A*STAR Research Attachment Programme. I have been able to produce multiple research results thanks to his help and the wonderful environment where I could devote myself to the research. I would like thank Vijay Chandrasekhar (A*STAR) and Satoshi Matsuoka (RIKEN-CCS) for giving me this opportunity in Singapore that has had a great impact on defining the direction of my research and career.

I would like to thank Ikuro Sato and Kohta Ishikawa (DENSO IT Lab, Inc.), Mohammad Emtiyaz Khan (RIKEN AIP), and Ryo Karakida (AIST) for their help and advice during the research internship. The experience of working with these machine learning researchers has been a great help to me in studying the intersection of the machine learning and high-performance computing.

Finally, I would like to thank Toshiyuki Tsurumi and Satoshi Iwazaki (Nefrock Inc.) for inspiring me to pursue a doctoral degree and providing me great role models as a computer engineer.

Contents

Notation

a, A	scalar
$\boldsymbol{a}$	(column) vector
$[\boldsymbol{a}]_i$, a_i	i-th element of a vector $\boldsymbol{a}$
$\boldsymbol{A}$	matrix
$[\boldsymbol{A}]_{i,j}$	(i,j)-th element of a matrix $\boldsymbol{A}$
A	tensor
$[\mathsf{A}]_{i,j,k,\dots}$	$(i,j,k,\dots)$-th element of a tensor A
$\mathrm{diag}\,(\boldsymbol{A})$	vector consisting of diagonal elements of a matrix $\boldsymbol{A}$
$\mathrm{diag}\,(\boldsymbol{a})$	diagonal matrix $\boldsymbol{A}$ satisfying $[\boldsymbol{A}]_{i,i} = [\boldsymbol{a}]_i$
$\boldsymbol{a} \odot \boldsymbol{b}$	element-wise (Hadamard) product between vectors (or matrices) $\boldsymbol{a}$ and $\boldsymbol{b}$
$\boldsymbol{a} \otimes \boldsymbol{b}$	Kronecker product between a vector (or a matrix) $\boldsymbol{a}$ and a vector (or a matrix) $\boldsymbol{b}$
$\mathbb{R}$	the real numbers
$\mathscr{D}$	data distribution
$\mathcal{S}$	training set drawn i.i.d. from $\mathscr{D}$
$\mathcal{B} \subset \mathcal{S}$	mini-batch
N ($N_{\mathcal{B}}$)	number of data points in $\mathcal{S}$ ($\mathcal{B}$)
$\langle \cdot \rangle$	average over the $\mathcal{S}$ (or $\mathcal{B}$, depending on the context)
L	number of layers in a neural network
P	number of parameters of a neural network
P_l	number of parameters in the l-th layer
M_0 (C_0)	input dimensionality (number of input channels) of a neural network
M_l (C_l)	number of units (number of output channels) in the l-th layer
K	output dimensionality of a neural network
$\boldsymbol{\theta} \in \mathbb{R}^P$	parameters of a neural network
$\boldsymbol{\theta}_l \in \mathbb{R}^{P_l}$	parameters in the l-th layer
$\boldsymbol{\theta}^{(t)}$	parameters after t updates
$\mathcal{X} \subset \mathbb{R}^{M_0}$	input space of a neural network
$\mathcal{Z} \subset \mathbb{R}^K$	output space of a neural network
$\mathcal{Y}$	space in which a ground truth label exists
$\boldsymbol{x} \in \mathcal{X}$	input data point
y (or $\boldsymbol{y}$) $\in \mathcal{Y}$	label data point
$f : \mathcal{X} \to \mathcal{Z}$	function mapping a neural network's input to its output (pre-softmax)
$\ell : \mathcal{X} \times \mathcal{Y} \times \mathbb{R}^P \to \mathbb{R}_{\geq 0}$	per-example loss
$\mathcal{L}$	objective function
$\nabla_{\boldsymbol{a}} h$	gradient of a scalar function h w.r.t. $\boldsymbol{a}$ ($= \boldsymbol{\theta}$ if not specified)
$\boldsymbol{J}_{h,\boldsymbol{a}}$	Jacobian of a vector-valued function h w.r.t. $\boldsymbol{a}$ ($= \boldsymbol{\theta}$ if not specified)
$\nabla^2_{\boldsymbol{a}} h$	Hessian of a scalar function h w.r.t. $\boldsymbol{a}$ ($= \boldsymbol{\theta}$ if not specified)
$\boldsymbol{H} := \nabla^2 \mathcal{L}$	Hessian of $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$
$\boldsymbol{F}$	Fisher information matrix (FIM) of a neural network
$\boldsymbol{F}_{\mathrm{mmc}}$	FIM estimated by m Monte-Carlo samples from the predictive distribution
$\boldsymbol{C}$	covariance of per-example gradient $\nabla \ell$
$\boldsymbol{I}_d$	identity matrix of shape $d \times d$

Chapter 1

Introduction

Our theoretical understanding of Deep Learning (DL) [LeCun et al., 2015] – how to train deep neural networks (DNNs) fast and stably for a given number of samples; how to select DNN models (combination of DNN architecture and trained parameters) that better generalize – is limited. Thus, research and development of DL require extensive trial and error of DNN architectures, training algorithms, and hyperparameters. Moreover, data to be observed and problems to be solved are increasing and changing day by day, and therefore, the realization of a fast and reliable learning method which does not rely on heuristics or "craftsmanship" for specific problems is a challenging open problem.

1.1 Information matrices in deep learning

Information matrices [Thomas et al., 2019], such as the Hessian matrix, Gauss-Newton matrix, Fisher information matrix (FIM), and gradient covariance matrix, have played a key role in mathematical optimization, statistical learning theory, and Bayesian statistics. Principled matrix-based methods for fast and robust training and better model selection have been developed in classical machine learning [Nocedal and Stephen, 2006, Hastie et al., 2009, Mohri et al., 2018] and have great potential in DL [Thomas et al., 2019]. For a DNN, the information matrix is a symmetric block matrix, where each diagonal block represents the correlation between parameters in a layer, and each off-diagonal block represents the correlation between two different layers of the DNN (Figure 1.1). Since the number of parameters in a DNN (P) is significant, it is impractical to handle the "full" information matrix (a $P \times P$ matrix). A simple and common strategy for handling this is to ignore the correlation between different layers and approximate it with a layer-wise block-diagonal matrix. Yet, each diagonal block is still large for most DNNs. Therefore, it is a common practice to approximate each diagonal block under even stronger assumptions (*e.g.*, low-rank, Kronecker-factored, sparse-factored, and diagonal) for practical computation (Figure 1.1). However, even with these approximations, few studies have tested the effectiveness of matrix-based algorithms in practical and "large-scale" DL involving a massive number of DNN parameters and training data. This is due to

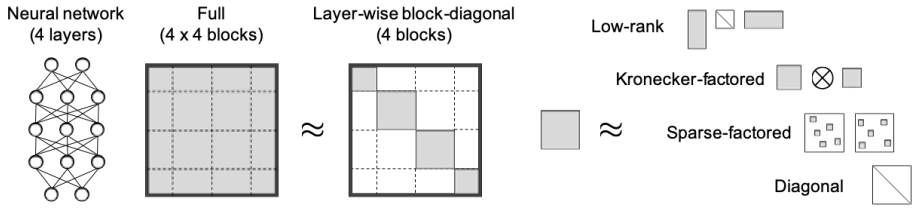


Figure 1.1: Information matrix of a deep neural network and its approximations

the fact that standard DL software libraries, such as PyTorch [Paszke et al., 2019], TensorFlow [Abadi et al., 2016], and JAX [Bradbury et al., 2018], do not support matrix calculation and that the cost of matrix calculation is quite high for large-scale DL.

High-Performance Computing (HPC) has made significant contributions to the advancement of DL. For example, distributed-memory training with multiple accelerators (e.g., GPUs, TPUs) speeds up DNN training (e.g., data-parallelism, model-parallelism) [Ben-Nun and Hoefler, 2019]; out-of-core and recomputation scale up DNN models and tasks by making better use of accelerator and CPU memory [Rajbhandari et al., 2019]; and large-scale computing clusters realize a comprehensive hyperparameter and DNN architecture search [Shallue et al., 2019].

1.2 Research direction and contributions

In this thesis, we mainly discuss the effectiveness of second-order optimization methods (information matrices as preconditioners) for large-scale DL and its efficient implementation with HPC. In order to realize fast information matrix calculations and practical matrix-based methods in large-scale DL, we propose novel uses of HPC in DL that differ from the conventional approaches (Figure 1.2).

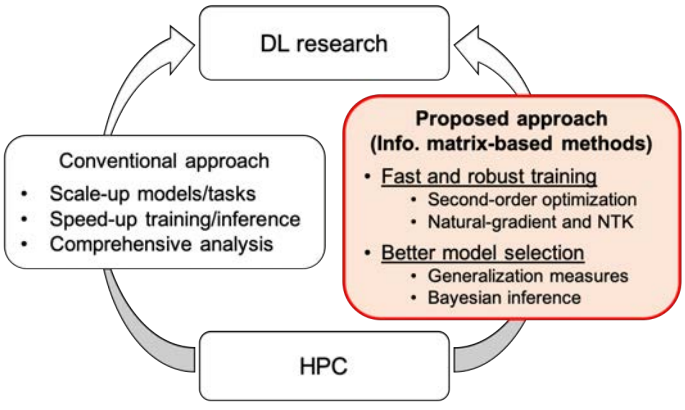


Figure 1.2: Research direction in this thesis: Novel use of HPC in DL.

1.2.1 Distributed parallel second-order optimization

Large-scale distributed training of deep neural networks results in models with worse generalization performance as a result of the increase in the effective mini-batch size. Previous approaches attempt to address this problem by varying the learning rate and batch size over epochs and layers, or ad hoc modifications of batch normalization. We propose *Scalable and Practical Natural Gradient Descent* (SP-NGD), a principled approach for training models that allows them to attain similar generalization performance to models trained with first-order optimization methods, but with accelerated convergence. Furthermore, SP-NGD scales to large mini-batch sizes with a negligible computational overhead as compared to first-order methods. We evaluated SP-NGD on a benchmark task where highly optimized first-order methods are available as references: training a ResNet-50 model for image classification on ImageNet. We demonstrate convergence to a top-1 validation accuracy of 75.4% in 5.5 minutes using a mini-batch size of 32,768 with 1,024 GPUs, as well as an accuracy of 74.9% with an extremely large mini-batch size of 131,072 in 873 steps of SP-NGD.

1.2.2 Distributed parallel second-order optimization for approximate Bayesian inference in deep learning

Bayesian methods promise to fix many shortcomings of deep learning, but they are impractical and rarely match the performance of standard methods, let alone improve them. In this paper, we demonstrate practical training of deep networks with natural-gradient variational inference. By applying techniques such as batch normalisation, data augmentation, and distributed training, we achieve similar performance in about the same number of epochs as the Adam optimizer, even on large datasets such as ImageNet. Importantly, the benefits of Bayesian principles are preserved: predictive probabilities are well-calibrated, uncertainties on out-of-distribution data are improved, and continual-learning performance is boosted. This work enables practical deep learning while preserving benefits of Bayesian principles.

Our contributions are:

- **Extremely large mini-batch training.** We were able to show for the first time that approximate Natural Gradient Descent (NGD) can achieve similar generalization capability compared to highly optimized SGD, by training ResNet-50 on ImageNet classification as a benchmark. We converged to over 75% top-1 validation accuracy for large mini-batch sizes of 4,096, 8,192, 16,384, 32,768 and 65,536. We also achieved 74.9% with an extremely large mini-batch size of 131,072, which took only 873 steps.

- **Scalable natural gradient.** We propose a distributed NGD design using data and model hybrid paralelism that shows *superlinear* scaling up to 64 GPUs.

- **Practical natural gradient.** We propose practical NGD techniques based on analysis of the FIM estimation in large mini-batch settings. Our practical techniques make the overhead of NGD compare to SGD almost negligible. Combining these techniques with our distributed NGD.

- **Training ResNet-50 on ImageNet in 5.5 minutes.** Using 1024 NVIDIA Tesla V100, we achieve 75.4 % top-1 accuracy with ResNet-50 for ImageNet in 5.5 minutes (1760 steps = 45 epochs, including a valida- tion after each epoch).

- **Large-scale Bayesian deep learning.** We successfully train deep neural networks with a Natural Gradient Variational Inference method, VOGN [Khan et al., 2018], on a variety of architectures and datasets, even scaling up to ImageNet for the first time. This is made possible due to the similarity of VOGN to Adam [Kingma and Ba, 2015], enabling us to boost performance by borrowing deep learning techniques and the application of our distributed NGD while preserving the benefits of the Bayesian principles.

1.3 Overview of this thesis

Chapter 2 gives notations and basics on supervised learning, neural networks and backpropagation that are used in Chapter 3.

Chapter 3 summarizes the information matrices in DL and their approximations. We discuss the difference in the time/space complexities of calculating matrices of various types/shapes and how they are used in DL research.

Chapter 4 reviews existing second-order optimization methods, Natural Gradient method, and its approximations in DL.

Chapter 5 describes an efficient implementation of distributed-memory DNN training by second-order optimization on a large-scale GPU cluster.

Chapter 6 extends the distributed second-order optimization implementation proposed in Chapter 5 to DL with Bayesian principles.

Chapter 7 proposes a software library for calculating the information matrices in large-scale DL and investigates the behaviors of the information matrices of various types/shapes when they are used as generalization measures.

Chapter 2

Fundamentals of deep learning

2.1 Training objective

In a supervised learning task, the goal is to minimize the *population loss*:

$$\mathcal{L}_{\mathscr{D}}(\boldsymbol{\theta}) := \mathbb{E}_{(\boldsymbol{x},y)\sim\mathscr{D}}\left[\ell(\boldsymbol{x},y,\boldsymbol{\theta})\right] , \tag{2.1}$$

where $\mathscr{D}$ is the data distribution we want to fit a model parameterized by $\boldsymbol{\theta} \in \mathbb{R}^P$. $\ell : \mathcal{X}\times\mathcal{Y}\times\mathbb{R}^P \to \mathbb{R}_{\geq 0}$ is the per-example loss:

$$\ell(\boldsymbol{x},y,\boldsymbol{\theta}) = h(y, f(\boldsymbol{x})) ,$$

where $f : \mathcal{X} \to \mathcal{Z}$ is the function mapping a model's input $\boldsymbol{x} \in \mathcal{X}$ to its output in $\mathcal{Z}$, and $h : \mathcal{Y}\times\mathcal{Z} \to \mathbb{R}_{\geq 0}$ measures the "difference" between the output $f(\boldsymbol{x}) \in \mathcal{Z}$ and the ground truth label $y \in \mathcal{Y}$. h ("difference") is defined for each task (*e.g.*, regression, classification).

Regression. In a regression task, $\mathcal{Y} = \mathcal{Z} = \mathbb{R}^K$, and

$$h(\boldsymbol{y},\boldsymbol{z}) := \frac{1}{2}\|\boldsymbol{y}-\boldsymbol{z}\|_2^2 \tag{2.2}$$

is the mean-squared-error (MSE) loss. This h can be replaced with

$$h(\boldsymbol{y},\boldsymbol{z}) = -\log p(\boldsymbol{y}|\boldsymbol{z}) ,$$

where $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{z}, \sigma^2\boldsymbol{I}_K)$ (for an arbitrary $\sigma > 0$), without changing the minimal solution of (2.1).

Classification. In a K-class classification task, $\mathcal{Y} = \{1, 2, \ldots, K\}$, $\mathcal{Z} = \mathbb{R}^K$, and

$$h(y,\boldsymbol{z}) := -\log p(y|\boldsymbol{z}) \tag{2.3}$$

is the softmax cross-entropy loss, where $p(k|\boldsymbol{z})$ ($k \in \mathcal{Y}$) is given by the softmax activation:

$$p(k|\boldsymbol{z}) = \exp\left([\boldsymbol{z}]_k\right) / \sum_{k'=1}^{K} \exp\left([\boldsymbol{z}]_{k'}\right) .$$

Usually, we do not know the true distribution $\mathscr{D}$. We are given a training set $\mathcal{S} = \bigcup_{n=1}^{N} \{(\boldsymbol{x}_n, y_n)\}$ drawn i.i.d. from distribution $\mathscr{D}$, and minimize the *empirical loss*[1]:

$$\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) := \frac{1}{|\mathcal{S}|} \sum_{(\boldsymbol{x}_n, y_n) \in \mathcal{S}} \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) = \langle \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \rangle \,, \tag{2.4}$$

where $\langle \cdot \rangle$ is the average over $\mathcal{S}$.

2.2 Deep neural networks

We consider to train a L-layered feed-forward deep neural network parameterized by $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} = \left[\begin{array}{cccc} \boldsymbol{\theta}_1^\top & \boldsymbol{\theta}_2^\top & \dots & \boldsymbol{\theta}_L^\top \end{array} \right]^\top \in \mathbb{R}^P,$$

where $\boldsymbol{\theta}_l \in \mathbb{R}^{P_l}$ $(l = 1, \dots, L)$ is the parameters for the l-th layer and $\sum_{l=1}^{L} P_l = P$.

Given an input $\boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^{M_0}$, we get the model's output by a *forward pass* (Figure 2.1):

$$\begin{aligned} \boldsymbol{h}_0 &= \boldsymbol{x} \in \mathbb{R}^{M_0} \,, \\ \boldsymbol{u}_l &= \boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l \in \mathbb{R}^{M_l} \;\; (l = 1, \dots, L-1) \,, \\ \boldsymbol{h}_l &= \phi_l(\boldsymbol{u}_l) \in \mathbb{R}^{M_l} \;\; (l = 1, \dots, L-1) \,, \\ f(\boldsymbol{x}) &= \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L \in \mathbb{R}^{K} \,, \end{aligned}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{M_l \times M_{l-1}}$ and $\boldsymbol{b}_l \in \mathbb{R}^{M_l}$ are the weights and bias in the l-th layer, respectively ($l = 1, \dots, L$, $M_L = K$), and

$$\boldsymbol{\theta}_l = \left[\begin{array}{c} \mathrm{vec}\,(\boldsymbol{W}_l) \\ \boldsymbol{b}_l \end{array} \right] \in \mathbb{R}^{P_l} \,,$$

where we define:

$$\mathrm{vec}\,(\boldsymbol{A}) := \mathrm{vec}\left(\left[\begin{array}{cccc} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_d \end{array} \right] \right) = \left[\begin{array}{c} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_d \end{array} \right] \in \mathbb{R}^{d^2}$$

for a matrix $\boldsymbol{A} \in \mathbb{R}^{d \times d}$. $\boldsymbol{u}_l \in \mathbb{R}^{M_l}$ is the pre-activation, and $\boldsymbol{h}_l \in \mathbb{R}^{M_l}$ is the activation calculated by the element-wise activation function $\phi_l : \mathbb{R}^{M_l} \to \mathbb{R}^{M_l}$ $(l = 1, \dots, L-1)$. The number of operations (additions and multiplications) of a forward pass is:

$$\begin{aligned} F &:= \sum_{l=1}^{L-1} \left(\underbrace{M_l(M_{l-1}+1)}_{\boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l} + \underbrace{M_l}_{\phi_l(\boldsymbol{u}_l)} \right) + \underbrace{K(M_{L-1}+1)}_{\boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L} + \underbrace{\mathcal{O}(K)}_{h(y, f(\boldsymbol{x}))} \\ &= \sum_{l=1}^{L} P_l + \sum_{l=1}^{L-1} M_l + \mathcal{O}(K) \\ &= P + \sum_{l=1}^{L-1} M_l + \mathcal{O}(K) \,. \end{aligned} \tag{2.5}$$

[1] Although the objective $\mathcal{L}_{\mathcal{S}}$ is often defined with regularization terms, we ignore them for simplicity.

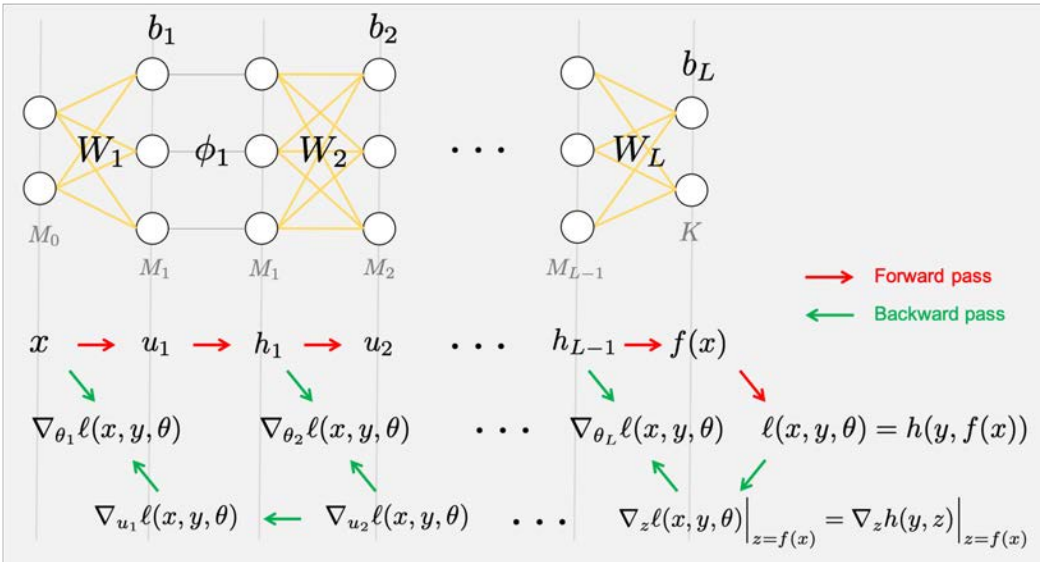


Figure 2.1: Feed-forward neural network and its forward/backward pass

2.3 Backpropagation

To minimize the objective $\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})$, the parameters $\boldsymbol{\theta}$ are often updated iteratively by the gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}) \,, \tag{2.6}$$

where $\boldsymbol{\theta}^{(t)}$ is the parameters after t updates, $\boldsymbol{\theta}^{(0)}$ is the initial parameters, $\eta > 0$ is the learning rate, and

$$\nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) = \nabla \left\langle \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \right\rangle = \left\langle \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \right\rangle \,.$$

The gradient w.r.t. $\boldsymbol{\theta}_l$ $(l = 1, \ldots, L)$ is calculated by *backward pass* (a.k.a. *backpropagation*, Figure 2.1). For $l = L$,

$$\nabla_{\boldsymbol{\theta}_L} \ell(\boldsymbol{x}, y, \boldsymbol{\theta}) = \begin{bmatrix} \operatorname{vec}\left(\nabla_{\boldsymbol{W}_L} \ell(\boldsymbol{x}, y, \boldsymbol{\theta})\right) \\ \nabla_{\boldsymbol{b}_L} \ell(\boldsymbol{x}, y, \boldsymbol{\theta}) \end{bmatrix} = \left[\nabla_{\boldsymbol{z}} h(y, \boldsymbol{z}) \Big|_{\boldsymbol{z}=f(\boldsymbol{x})} \otimes \boldsymbol{h}_{L-1}^{+} \right] \in \mathbb{R}^{P_L} \,,$$

where we define $\boldsymbol{h}_l^{+} := \begin{bmatrix} \boldsymbol{h}_l^{\top} & 1 \end{bmatrix}^{\top} \in \mathbb{R}^{M_l+1}$. For $l = 1, \ldots, L-1$,

$$\nabla_{\boldsymbol{\theta}_l} \ell(\boldsymbol{x}, y, \boldsymbol{\theta}) = \left[\nabla_{\boldsymbol{u}_l} \ell\left(\boldsymbol{x}, y, \boldsymbol{\theta}\right) \otimes \boldsymbol{h}_{l-1}^{+} \right] \in \mathbb{R}^{P_l} \,.$$

For $\boldsymbol{A} \in \mathbb{R}^{d_1 \times d_2}$ and $\boldsymbol{B} \in \mathbb{R}^{d_3 \times d_4}$, the Kronecker product between them is defined as:

$$\boldsymbol{A} \otimes \boldsymbol{B} := \begin{bmatrix} [\boldsymbol{A}]_{1,1} \boldsymbol{B} & \cdots & [\boldsymbol{A}]_{1,1} \boldsymbol{B} \\ \vdots & \ddots & \vdots \\ [\boldsymbol{A}]_{d_1,1} \boldsymbol{B} & \cdots & [\boldsymbol{A}]_{d_1,d_2} \boldsymbol{B} \end{bmatrix} \in \mathbb{R}^{d_1 d_3 \times d_2 d_4} \,.$$

By defining $\boldsymbol{J}_{f,\boldsymbol{u}_l} := \frac{\partial f}{\partial \boldsymbol{u}_l} \in \mathbb{R}^{K \times M_l}$, the gradient w.r.t. the pre-activation $\boldsymbol{u}_l$ can also be written as a transposed Jacobian-vector product:

$$\nabla_{\boldsymbol{u}_l} \ell\left(\boldsymbol{x}, y, \boldsymbol{\theta}\right) = \boldsymbol{J}_{f,\boldsymbol{u}_l}(\boldsymbol{x})^{\top} \nabla_{\boldsymbol{z}} h(y, \boldsymbol{z}) \Big|_{\boldsymbol{z}=f(\boldsymbol{x})} \in \mathbb{R}^{M_l} \,.$$

We can get this transposed Jacobian-vector product efficiently by using *automatic-differentiation* libraries, such as JAX [Bradbury et al., 2018], TensorFlow [Abadi et al., 2016], and PyTorch [Paszke et al., 2019]. For a vector $\boldsymbol{v} \in \mathbb{R}^K$, an automatic-differentiation library calculates the transposed Jacobian-vector product $\boldsymbol{J}_{f,\boldsymbol{u}_l}(\boldsymbol{x})^\top \boldsymbol{v}$ by *backpropagating* $\boldsymbol{v}$:

$$\begin{aligned}
\boldsymbol{J}_{f,\boldsymbol{h}_{L-1}}(\boldsymbol{x})^\top \boldsymbol{v} &= \boldsymbol{W}_L^\top \boldsymbol{v} \in \mathbb{R}^{M_{L-1}}\,, \\
\boldsymbol{J}_{f,\boldsymbol{h}_l}(\boldsymbol{x})^\top \boldsymbol{v} &= \boldsymbol{W}_{l+1}^\top \boldsymbol{J}_{f,\boldsymbol{u}_{l+1}}(\boldsymbol{x})^\top \boldsymbol{v} \in \mathbb{R}^{M_l} \quad (l = 1, \dots, L-2)\,, \\
\boldsymbol{J}_{f,\boldsymbol{u}_l}(\boldsymbol{x})^\top \boldsymbol{v} &= \phi_l'(\boldsymbol{u}_l) \odot \boldsymbol{J}_{f,\boldsymbol{h}_l}(\boldsymbol{x})^\top \boldsymbol{v} \in \mathbb{R}^{M_l} \quad (l = 1, \dots, L-1)\,.
\end{aligned}$$

Note that **there is no need to explicitly construct the Jacobian matrices** $\boldsymbol{J}_{f,\boldsymbol{u}_l}(\boldsymbol{x}) \in \mathbb{R}^{K \times M_l}$ and $\boldsymbol{J}_{f,\boldsymbol{h}_l}(\boldsymbol{x}) \in \mathbb{R}^{K \times M_l}$ $(l = 1, \dots, L-1)$ during the backpropagation, and we can calculate the gradient $\nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})$ in a memory-efficient way. The number of operations (additions and multiplications) of a backward pass is:

$$B := B_{\text{hidden}} + B_{\text{params}}\,, \tag{2.7}$$

where we define

$$\begin{aligned}
B_{\text{hidden}} &:= \sum_{l=1}^{L-1} \underbrace{M_l}_{\phi_l'(\boldsymbol{u}_l) \odot \boldsymbol{J}_{f,\boldsymbol{h}_l}(\boldsymbol{x})^\top \boldsymbol{v}} + \sum_{l=1}^{L-2} \underbrace{M_l M_{l+1}}_{\boldsymbol{W}_{l+1}^\top \boldsymbol{J}_{f,\boldsymbol{u}_{l+1}}(\boldsymbol{x})^\top \boldsymbol{v}} + \underbrace{M_{L-1} K}_{\boldsymbol{W}_L^\top \boldsymbol{v}} + \underbrace{\mathcal{O}(K)}_{\boldsymbol{v} := \nabla_{\boldsymbol{z}} h(y, \boldsymbol{z})\big|_{\boldsymbol{z} = f(\boldsymbol{x})}} \\
&= \sum_{l=2}^{L} P_l + \mathcal{O}(K) = P - P_1 + \mathcal{O}(K)\,,
\end{aligned} \tag{2.8}$$

and

$$B_{\text{params}} := \sum_{l=1}^{L-1} \underbrace{M_l(M_{l-1}+1)}_{\boldsymbol{J}_{f,\boldsymbol{u}_l}(\boldsymbol{x})^\top \boldsymbol{v} \otimes \boldsymbol{h}_{l-1}^+} + \underbrace{K(M_{L-1}+1)}_{\boldsymbol{v} \otimes \boldsymbol{h}_{L-1}^+} = \sum_{l=1}^{L} P_l = P\,. \tag{2.9}$$

Chapter 3

Information matrices of deep neural networks

Matrices that contain information about the curvature and noise play a key role in predicting optimal hyper-parameters [McCandlish et al., 2018, Lorraine et al., 2019], estimating the generalization gap [Thomas et al., 2020, Maddox et al., 2020], preconditioning the optimizers [Amari, 1998], Bayesian inference [Zhang et al., 2018a, Osawa et al., 2019a], continual learning [Kirkpatrick et al., 2017, Ritter et al., 2018a], and meta learning [Rajeswaran et al., 2019].

3.1 Matrix types

The Hessian matrix of the empirical loss w.r.t. the parameter $\boldsymbol{\theta} \in \mathbb{R}^P$ is defined as:

$$\boldsymbol{H}(\boldsymbol{\theta}) := \nabla^2 \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|} \sum_{(\boldsymbol{x}_n, y_n) \in \mathcal{S}} \nabla^2 \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) = \left\langle \nabla^2 \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \right\rangle \in \mathbb{R}^{P \times P}, \tag{3.1}$$

where

$$\begin{aligned}
\left[\nabla^2 \ell(\boldsymbol{x}, y, \boldsymbol{\theta})\right]_{i,j} &= \frac{\partial^2 \ell(\boldsymbol{x}, y, \boldsymbol{\theta})}{\partial \theta_i \partial \theta_j} = \frac{\partial}{\partial \theta_j} \sum_{k=1}^{K} \frac{\partial z_k}{\partial \theta_i} \frac{\partial h(y, \boldsymbol{z})}{\partial z_k}\Big|_{\boldsymbol{z}=f(\boldsymbol{x})} \\
&= \sum_{k=1}^{K} \sum_{k'=1}^{K} \frac{\partial z_k}{\partial \theta_i} \frac{\partial^2 h(y, \boldsymbol{z})}{\partial z_k \partial z_{k'}} \frac{\partial z_{k'}}{\partial \theta_j}\Big|_{\boldsymbol{z}=f(\boldsymbol{x})} + \sum_{k=1}^{K} \frac{\partial^2 z_k}{\partial \theta_i \partial \theta_j} \frac{\partial h(y, \boldsymbol{z})}{\partial z_k}\Big|_{\boldsymbol{z}=f(\boldsymbol{x})}.
\end{aligned}$$

By ignoring the second term, we get the generalized Gauss-Newton approximation of the Hessian matrix [Schraudolph, 2002]:

$$\boldsymbol{G}(\boldsymbol{\theta}) := \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \nabla^2_{\boldsymbol{z}} h(y_n, \boldsymbol{z})\Big|_{\boldsymbol{z}=f(\boldsymbol{x}_n)} \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \approx \boldsymbol{H}(\boldsymbol{\theta}), \tag{3.2}$$

where $\nabla^2_{\boldsymbol{z}} h(y, \boldsymbol{z}) \in \mathbb{R}^{K \times K}$ and $\boldsymbol{J}_f := \frac{\partial f}{\partial \boldsymbol{\theta}} \in \mathbb{R}^{K \times P}$.

For the MSE loss (Equation 2.2) and softmax cross-entropy loss (Equation 2.3), $\boldsymbol{G}$ is equivalent to the Fisher information matrix:

$$\boldsymbol{F}(\boldsymbol{\theta}) := \left\langle \mathbb{E}_{k \sim p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)} \left[\nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)^\top \right] \right\rangle, \tag{3.3}$$

where $p_{\boldsymbol{\theta}}(k|\boldsymbol{x}) := p(k|f(\boldsymbol{x}))$. For a K-class classification task, the expectation in (3.3) can be evaluated through the backpropagation of $\log p_{\boldsymbol{\theta}}(k|\boldsymbol{x})$ for each $k \in \mathcal{Y} = \{1, 2, \ldots, K\}$ (K-times loop):

$$\boldsymbol{F}(\boldsymbol{\theta}) = \left\langle \sum_{k=1}^{K} p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)^\top \right\rangle .$$

However, this becomes infeasible when K is large (*e.g.*, $K = 100, 1000$ for CIFAR-100, ImageNet). Hence, it is a common strategy to estimate it by Monte-Carlo samplings:

$$\boldsymbol{F}_{\text{mmc}}(\boldsymbol{\theta}) := \left\langle \frac{1}{m} \sum_{i=1}^{m} \nabla \log p_{\boldsymbol{\theta}}(k^{(i)}|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(k^{(i)}|\boldsymbol{x}_n)^\top \right\rangle \approx \boldsymbol{F}(\boldsymbol{\theta}) , \tag{3.4}$$

where $m \geq 1$ is the number of the samples and i-th sample $k^{(i)} \sim p_{\boldsymbol{\theta}}(k|\boldsymbol{x})$ is a label in $\mathcal{Y}$. When m is small enough, $\boldsymbol{F}_{\text{mmc}}$ can be practically computed, and $m = 1$ is often chosen [Martens and Grosse, 2015, Martens, 2017, Zhang et al., 2018a].

The uncentered gradient covariance (a.k.a. *empirical Fisher* [Martens, 2017]) is defined as:

$$\begin{aligned} \boldsymbol{C}(\boldsymbol{\theta}) &:= \left\langle \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta})^\top \right\rangle \\ &= \left\langle \nabla \log p_{\boldsymbol{\theta}}(y_n|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(y_n|\boldsymbol{x}_n)^\top \right\rangle \in \mathbb{R}^{P \times P} . \end{aligned} \tag{3.5}$$

The computation of (3.5) is very similar to those of (3.3) and (3.4) because they all involve the average of the (expected) outer product of $\nabla \log p_{\boldsymbol{\theta}}$, but the statistical features are different, *i.e.*, $\boldsymbol{C}$ is not a good approximation of $\boldsymbol{F}$ nor $\boldsymbol{F}_{\text{mmc}}$ [Kunstner et al., 2019, Thomas et al., 2020]. Having said that, it is worth noting that we can compute $\boldsymbol{C}$ at the lowest cost (lower than that for $\boldsymbol{F}_{\text{1mc}}$) because we can get $\nabla \log p_{\boldsymbol{\theta}}(y_n|\boldsymbol{x}_n)$ during the calculation of the gradient of the objective $\nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) = \langle \nabla \log p_{\boldsymbol{\theta}}(y_n|\boldsymbol{x}_n) \rangle$ while the others require additional backpropagation. This makes difference in second-order optimization [Osawa et al., 2020].

3.1.1 Generalized Gauss-Newton matrix and Fisher information matrix

For the standard training objectives in deep learning, $\boldsymbol{G}$ is equivalent to $\boldsymbol{F}$ [Martens, 2020].

Regression. For the MSE loss (Equation 2.2),

$$\nabla_{\boldsymbol{z}}^2 h(\boldsymbol{y}, \boldsymbol{z}) = \nabla_{\boldsymbol{z}}^2 \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{z}\|_2^2 = \boldsymbol{I}_K .$$

Hence

$$\begin{aligned} \boldsymbol{G}(\boldsymbol{\theta}) &= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \nabla_{\boldsymbol{z}}^2 h(\boldsymbol{y}, \boldsymbol{z}) \Big|_{\boldsymbol{z}=f(\boldsymbol{x})} \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \\ &= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{I}_K \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \\ &= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle . \end{aligned}$$

On the other hand, by assuming that $\boldsymbol{y} \sim \mathcal{N}(f(\boldsymbol{x}), \boldsymbol{I}_K)$ ($\sigma^2 = 1$),

$$\nabla \log p(\boldsymbol{y}|f(\boldsymbol{x})) = \boldsymbol{J}_f(\boldsymbol{x})^\top (f(\boldsymbol{x}) - \boldsymbol{y}) ,$$

and

$$\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}) &= \left\langle \mathbb{E}_{\boldsymbol{y}\sim p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}_n)} \left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}_n)^\top\right] \right\rangle \\
&= \left\langle \mathbb{E}_{\boldsymbol{y}\sim \mathcal{N}(f(\boldsymbol{x}_n), \boldsymbol{I}_K)} \left[\nabla \log p(\boldsymbol{y}|f(\boldsymbol{x}_n)) \nabla \log p(\boldsymbol{y}|f(\boldsymbol{x}_n))^\top\right] \right\rangle \\
&= \left\langle \mathbb{E}_{\boldsymbol{y}\sim \mathcal{N}(f(\boldsymbol{x}_n), \boldsymbol{I}_K)} \left[\boldsymbol{J}_f(\boldsymbol{x}_n)^\top \left(f(\boldsymbol{x}_n) - \boldsymbol{y}\right) \left(f(\boldsymbol{x}_n) - \boldsymbol{y}\right)^\top \boldsymbol{J}_f(\boldsymbol{x}_n)\right] \right\rangle \\
&= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \mathbb{E}_{\boldsymbol{y}\sim \mathcal{N}(f(\boldsymbol{x}_n), \boldsymbol{I}_K)} \left[\left(\boldsymbol{y} - f(\boldsymbol{x}_n)\right) \left(\boldsymbol{y} - f(\boldsymbol{x}_n)\right)^\top\right] \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \\
&= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{I}_K \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \\
&= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle = \boldsymbol{G}(\boldsymbol{\theta}) \,.
\end{aligned}$$

Classification. For the softmax cross-entropy loss (Equation 2.3),

$$\left[\nabla_{\boldsymbol{z}} h(y, \boldsymbol{z})\right]_k = \frac{\partial - \log p(y|\boldsymbol{z})}{\partial z_k} = -\frac{1}{p(y|\boldsymbol{z})} \frac{\partial p(y|\boldsymbol{z})}{\partial z_k} \,,$$

and

$$\frac{\partial p(y|\boldsymbol{z})}{\partial z_k} = \frac{\partial \exp(z_y) / \sum_{k'=1}^K \exp(z_{k'})}{\partial z_k} = \begin{cases} p(y|\boldsymbol{z})(1 - p(y|\boldsymbol{z})) & \text{if } k = y, \\ -p(y|\boldsymbol{z}) p(k|\boldsymbol{z}) & \text{otherwise.} \end{cases}$$

Therefore,

$$\left[\nabla_{\boldsymbol{z}} h(y, \boldsymbol{z})\right]_k = \begin{cases} p(y|\boldsymbol{z}) - 1 & \text{if } k = y, \\ p(k|\boldsymbol{z}) & \text{otherwise,} \end{cases}$$

$$\left[\nabla^2_{\boldsymbol{z}} h(y, \boldsymbol{z})\right]_{k,k'} = \begin{cases} p(k|\boldsymbol{z})(1 - p(k|\boldsymbol{z})) = p(k|\boldsymbol{z}) - p(k|\boldsymbol{z})^2 & \text{if } k = k', \\ -p(k|\boldsymbol{z}) p(k'|\boldsymbol{z}) & \text{otherwise,} \end{cases} \,.$$

and

$$\nabla^2_{\boldsymbol{z}} h(y, \boldsymbol{z}) = \operatorname{diag}(\boldsymbol{p}) - \boldsymbol{p}\boldsymbol{p}^\top \in \mathbb{R}^{K \times K} \,,$$

where $\boldsymbol{p} \in \mathbb{R}^K$ is a vector satisfying $[\boldsymbol{p}]_k = p(k|\boldsymbol{z})$ $(k = 1, \ldots, K)$. Using this Hessian, we get

$$\begin{aligned}
\boldsymbol{G}(\boldsymbol{\theta}) &= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \nabla^2_{\boldsymbol{z}} h(y_n, \boldsymbol{z}) \Big|_{\boldsymbol{z}=f(\boldsymbol{x}_n)} \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \\
&= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \left(\operatorname{diag}(\boldsymbol{p}) - \boldsymbol{p}\boldsymbol{p}^\top\right) \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \,.
\end{aligned}$$

Whereas

$$\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}) &= \left\langle \mathbb{E}_{k\sim p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)} \left[\nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n) \nabla \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)^\top\right] \right\rangle \\
&= \left\langle \mathbb{E}_{k\sim p(k|\boldsymbol{z})} \left[\boldsymbol{J}_f(\boldsymbol{x}_n)^\top \nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z}) \nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z})^\top \boldsymbol{J}_f(\boldsymbol{x}_n)\right] \Big|_{\boldsymbol{z}=f(\boldsymbol{x}_n)} \right\rangle \\
&= \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \mathcal{F}(f(\boldsymbol{x}_n)) \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \,,
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{F}(\boldsymbol{z}) &:= \mathbb{E}_{k\sim p(k|\boldsymbol{z})} \left[\nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z}) \nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z})^\top\right] \\
&= \sum_{k=1}^K p(k|\boldsymbol{z}) \nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z}) \nabla_{\boldsymbol{z}} \log p(k|\boldsymbol{z})^\top \in \mathbb{R}^{K \times K} \,.
\end{aligned}$$

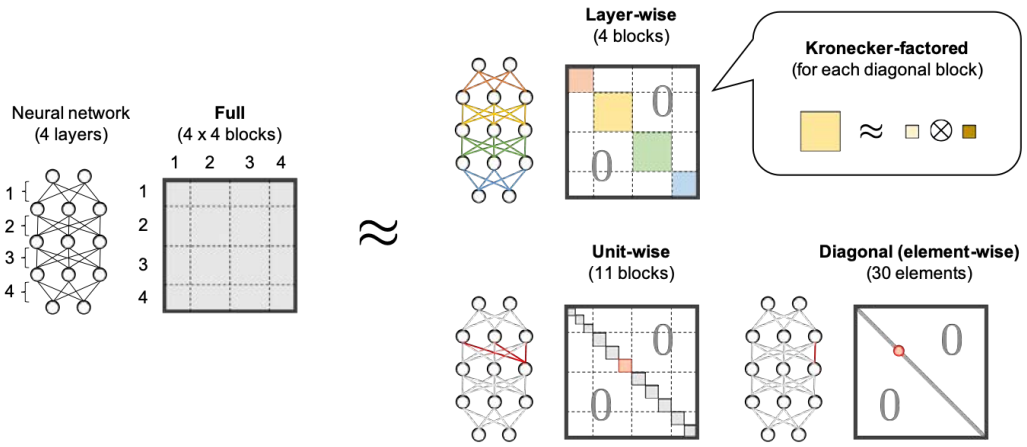


Figure 3.1: Illustration of information matrix shapes for feed-forward deep neural networks.

By using the partial derivatives of $h(y, \boldsymbol{z}) = -\log p(y|\boldsymbol{z})$, we get

$$\begin{aligned}
[\mathcal{F}(\boldsymbol{z})]_{k,k} &= p(k|\boldsymbol{z})(p(k|\boldsymbol{z}) - 1)^2 + \sum_{k' \neq k} p(k'|\boldsymbol{z}) p(k|\boldsymbol{z})^2 \\
&= p(k|\boldsymbol{z}) - 2p(k|\boldsymbol{z})^2 + \sum_{k'=1}^{K} p(k'|\boldsymbol{z}) p(k|\boldsymbol{z})^2 \\
&= p(k|\boldsymbol{z}) - p(k|\boldsymbol{z})^2 \,,
\end{aligned}$$

and for $k \neq k'$,

$$\begin{aligned}
[\mathcal{F}(\boldsymbol{z})]_{k,k'} &= p(k|\boldsymbol{z})(p(k|\boldsymbol{z}) - 1)p(k'|\boldsymbol{z}) + p(k'|\boldsymbol{z})(p(k'|\boldsymbol{z}) - 1)p(k|\boldsymbol{z}) + \sum_{k'' \neq k,k'} p(k''|\boldsymbol{z}) p(k|\boldsymbol{z}) p(k'|\boldsymbol{z}) \\
&= -p(k|\boldsymbol{z})p(k'|\boldsymbol{z}) - p(k'|\boldsymbol{z})p(k|\boldsymbol{z}) + \sum_{k''=1}^{K} p(k''|\boldsymbol{z}) p(k|\boldsymbol{z}) p(k'|\boldsymbol{z}) \\
&= -p(k|\boldsymbol{z})p(k'|\boldsymbol{z}) \,.
\end{aligned}$$

Finally, we get

$$\mathcal{F}(\boldsymbol{z}) = \operatorname{diag}(\boldsymbol{p}) - \boldsymbol{p}\boldsymbol{p}^\top = \nabla_{\boldsymbol{z}}^2 h(y|\boldsymbol{z}) \,,$$

and

$$\boldsymbol{F}(\boldsymbol{\theta}) = \boldsymbol{G}(\boldsymbol{\theta}) \,.$$

3.2 Matrix shapes

Full. Since the number of parameters P of a deep neural network is significant (*e.g.*, $P = 26\text{M}$ for the ResNet-50 [He et al., 2016] on ImageNet), it is impractical to handle "full" information matrices $\boldsymbol{M} \in \mathbb{R}^{P \times P}$ (*e.g.*, 2.7 Petabytes memory for the ResNet-50 if it is stored with 32-bit). For analyzing the eigenvalues of such matrices, we can use the power method or the Lanczos algorithm without explicitly constructing the matrices [Yao et al., 2020]. For second-order optimization where

we need the preconditioned gradients $\boldsymbol{M}^{-1}\nabla\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})$, we can avoid evaluating $\boldsymbol{M}^{-1}$ explicitly by using the conjugate gradient algorithm [Martens, 2010]. However, these approaches tend to be computationally heavy in many deep neural networks as they require iterative processing depending on the number of parameters P.

By definition, all the information matrices $\boldsymbol{H}$,$\boldsymbol{F}$,$\boldsymbol{F}_{\text{mmc}}$, and $\boldsymbol{C}$ are symmetric and block matrices that have $L \times L$ blocks (Figure 3.1):

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{M}_{1,1} & \cdots & \boldsymbol{M}_{1,L} \\ \vdots & \ddots & \vdots \\ \boldsymbol{M}_{L,1} & \cdots & \boldsymbol{M}_{L,L} \end{bmatrix}.$$

The (i,j)-th block $\boldsymbol{M}_{i,j} \in \mathbb{R}^{P_i \times P_j}$ represents the correlation between the i-th and j-th layer.

Layer-wise. A simple and common strategy for handling these matrices is to ignore the correlation between different layers and to approximate to a layer-wise block-diagonal matrix (Figure 3.1):

$$\boldsymbol{M} \approx \begin{bmatrix} \boldsymbol{M}_{1,1} & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{M}_{L,L} \end{bmatrix}.$$

Yet, each diagonal block $\boldsymbol{M}_{l,l} \in \mathbb{R}^{P_l \times P_l}$ is still large for most of practical deep neural networks, *e.g.*, the largest block of Fully Connected on MNIST requires 4.4 Terabytes.

Kronecker-factored. Kronecker-factored Approximate Curvature (K-FAC) [Martens and Grosse, 2015] is an approximate method of $\boldsymbol{F}$ ($\boldsymbol{F}_{\text{mmc}}$) for an efficient natural-gradient learning [Amari, 1998]:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{F}(\boldsymbol{\theta}^{(t)})^{-1}\nabla\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}). \tag{3.6}$$

Recently, K-FAC is also widely used in deep learning research to approximate various information matrices (including $\boldsymbol{H}$ and $\boldsymbol{C}$), not only for the natural-gradients [Botev et al., 2017, Ritter et al., 2018a, Wang et al., 2019]. K-FAC approximates each diagonal block $\boldsymbol{M}_{l,l}$ to a Kronecker product. For $\boldsymbol{F}$,$\boldsymbol{F}_{\text{1mc}}$, and $\boldsymbol{C}$:

$$\begin{aligned}
\boldsymbol{M}_{l,l} &= \left\langle \mathbb{E}\left[\nabla_{\boldsymbol{\theta}_l}\ell \nabla_{\boldsymbol{\theta}_l}\ell^\top\right]\right\rangle \\
&= \left\langle \mathbb{E}\left[\left(\nabla_{\boldsymbol{u}_l}\ell \otimes \boldsymbol{h}^+_{l-1}\right)\left(\nabla_{\boldsymbol{u}_l}\ell \otimes \boldsymbol{h}^+_{l-1}\right)^\top\right]\right\rangle \\
&= \left\langle \mathbb{E}\left[\nabla_{\boldsymbol{u}_l}\ell \nabla_{\boldsymbol{u}_l}\ell^\top \otimes \boldsymbol{h}^+_{l-1}{\boldsymbol{h}^+_{l-1}}^\top\right]\right\rangle \\
&\approx \boldsymbol{B}_l \otimes \boldsymbol{A}_{l-1}, \\
\boldsymbol{B}_l &:= \left\langle \mathbb{E}\left[\nabla_{\boldsymbol{u}_l}\ell \nabla_{\boldsymbol{u}_l}\ell^\top\right]\right\rangle \in \mathbb{R}^{M_l \times M_l}, \\
\boldsymbol{A}_l &:= \left\langle \boldsymbol{h}^+_l {\boldsymbol{h}^+_l}^\top \right\rangle \in \mathbb{R}^{(M_l+1)\times(M_l+1)}
\end{aligned}$$

(Figure 3.1). Note that $\mathbb{E}[\cdot]$ and ℓ in $\boldsymbol{M}_{l,l}$ and $\boldsymbol{B}_l$ depend on the definition of matrix type ($\boldsymbol{F}$ (3.3), $\boldsymbol{F}_{\text{mmc}}$ (3.4), or $\boldsymbol{C}$ (3.5)) while $\boldsymbol{A}_l$ is the same for all matrix types. K-FAC for $\boldsymbol{H}$ and its variants were proposed by Botev et al. [2017] that also have the same $\boldsymbol{A}_l$. K-FAC for convolutional layers was

proposed by Grosse and Martens [2016]. A Kronecker product has preferable features for inverting it and analyzing its eigenvalues efficiently:

$$(\boldsymbol{B} \otimes \boldsymbol{A})^{-1} = \boldsymbol{B}^{-1} \otimes \boldsymbol{A}^{-1} \text{ and}$$
$$\mathrm{eig}(\boldsymbol{B} \otimes \boldsymbol{A}) = \mathrm{eig}(\boldsymbol{B}) \otimes \mathrm{eig}(\boldsymbol{A}) \,,$$

where $\mathrm{eig}(\boldsymbol{X})$ is the eigenvalues of the matrix $\boldsymbol{X}$. When a Kronecker-factored matrix is used for preconditioning a vector $\boldsymbol{v} \in \mathbb{R}^{(M_{l-1}+1)M_l}$,

$$(\boldsymbol{B}_l \otimes \boldsymbol{A}_{l-1})^{-1} \boldsymbol{v} = \boldsymbol{B}_l^{-1} \otimes \boldsymbol{A}_{l-1}^{-1} \boldsymbol{v} = \mathrm{vec}\left(\boldsymbol{A}_{l-1}^{-1} \boldsymbol{V} \boldsymbol{B}_l^{-1}\right) \,,$$

where $\boldsymbol{V} \in \mathbb{R}^{(M_{l-1}+1) \times M_l}$ and $\mathrm{vec}(\boldsymbol{V}) = \boldsymbol{v}$. Hence, we need not to construct the matrix $\boldsymbol{B}_l^{-1} \otimes \boldsymbol{A}_{l-1}^{-1} \in \mathbb{R}^{P_l \times P_l}$, which is often too large to put on memory, explicitly for evaluating the natural-gradient (3.6).

Unit-wise. The i-th element of the pre-activation $\boldsymbol{u}_l = \boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l \in \mathbb{R}^{M_l}$ is evaluated as:

$$u_{l,i} = \boldsymbol{w}_{l,i}^\top \boldsymbol{h}_{l-1}^+ \,,$$

where

$$\begin{bmatrix} \boldsymbol{W}_l & \boldsymbol{b}_l \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_{l,1}^\top \\ \vdots \\ \boldsymbol{w}_{l,M_l}^\top \end{bmatrix} \in \mathbb{R}^{M_l \times (M_{l-1}+1)} \,,$$

and $\boldsymbol{w}_{l,i} \in \mathbb{R}^{M_{l-1}}$ is the weight in the i-th unit ($i = 1, \ldots, M_l$). By assuming that the units are independent to each other, each (layer-wise) diagonal block $\boldsymbol{M}_{l,l}$ is further approximated with a unit-wise block diagonal matrix (Figure 3.1):

$$\boldsymbol{M}_{l,l} \approx \begin{bmatrix} \boldsymbol{D}_{l,1} & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{D}_{l,M_l} \end{bmatrix} \,,$$

where we define

$$\boldsymbol{D}_{l,i} := \left\langle \mathbb{E}\left[\nabla_{\boldsymbol{w}_{l,i}} \ell \nabla_{\boldsymbol{w}_{l,i}} \ell^\top\right] \right\rangle = \left\langle \mathbb{E}\left[\left(\nabla_{u_{l,i}} \ell\right)^2 \boldsymbol{h}_{l-1}^+ \boldsymbol{h}_{l-1}^{+\top}\right] \right\rangle \in \mathbb{R}^{(M_{l-1}+1) \times (M_{l-1}+1)} \,.$$

The unit-wise approximation of $\boldsymbol{F}$ has been studied for evaluating the natural-gradient practically [Amari et al., 2019].

Diagonal (element-wise). Although the Kronecker-factored approximation and the unit-wise approximation significantly reduce the time/space complexity of constructing and inverting matrices, they are not implemented in standard deep learning software libraries (*e.g.*, JAX, TensorFlow, and PyTorch). This is because they require non-trivial implementations on top of the automatic-differentiation framework and the time/space complexity can be still significant for large-scale deep neural networks such as the ResNet. Therefore, a crude approximation that extracts only the

diagonal elements $\mathrm{diag}(\boldsymbol{M}) \in \mathbb{R}^P$ has often been used in deep learning research (Figure 3.1). The diagonal elements of the (uncentered) covariance (empirical Fisher) can be calculated as *the average of the squared per-example gradient*:

$$\mathrm{diag}\left(\boldsymbol{C}\right) = \langle \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \odot \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \rangle \ .$$

On the other hand, the value used by the *Adam* [Kingma and Ba, 2015] is *the square of the averaged gradient*:

$$\boldsymbol{g}^2 := \langle \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \rangle \odot \langle \nabla \ell(\boldsymbol{x}_n, y_n, \boldsymbol{\theta}) \rangle \ .$$

We can get $\boldsymbol{g}^2$ by simply taking the square of the gradient calculated by an automatic-differentiation library while we need to access the per-example gradient for $\mathrm{diag}\left(\boldsymbol{C}\right)$.

3.3 Deep learning research with the information matrices

There has been an increased attention to information matrices in recent deep learning research. Information matrices are often used for DNN training and generalization performance analysis. Eigenvalue analysis of various types ($\boldsymbol{H}, \boldsymbol{F}, \boldsymbol{F}_{\mathrm{mmc}}$, and $\boldsymbol{C}$) of information matrices is also studied. In this section, we review some of those studies on information matrices in deep learning.

Training. $\boldsymbol{F}$ is used for the natural-gradient descent [Amari, 1998], an approximate second-order optimization as a gradient descent in the Riemannian space. This is often estimated by $\boldsymbol{F}_{\mathrm{mmc}}$ [Martens and Grosse, 2015, Zhang et al., 2018a]. Osawa et al. [2020] further approximated it with $\boldsymbol{C}$ and showed empirically that the convergence of the second-order optimization for ImageNet training is maintained while reducing the computational cost. Jastrzebski et al. [2020] observed the effect of the initial learning rate selection on the condition of $\boldsymbol{H}$ and $\boldsymbol{C}$ at each phase of training. McCandlish et al. [2018] proposed the *gradient noise scale*: $\mathcal{B}_{\mathrm{noise}} := \mathrm{Tr}\left\{\boldsymbol{H}\boldsymbol{C}\right\} / \nabla\mathcal{L}^\top \boldsymbol{H} \nabla\mathcal{L} - 1$, for predicting the *critical batch size*, the largest batch size at which we can train a model without sacrificing data efficiency. This is useful for large-batch training [Goyal et al., 2017].

Generalization. Numerous generalization measures using $\boldsymbol{H}, \boldsymbol{F}$ and $\boldsymbol{C}$ to predict the generalization gap of deep models have been revisited in recent years [Jiang et al., 2020, Thomas et al., 2020, Maddox et al., 2020]. The "sharpness" of the loss surface [Keskar et al., 2017] is thought to correlate with the generalization, which is evaluated by measures using $\boldsymbol{H}$ [Tsuzuku et al., 2019, He et al., 2019, Jiang et al., 2020]. Wen et al. [2020] showed empirically that adding $\boldsymbol{C}$-derived noise to the update vector of the large-batch SGD increased the variance of gradients and improved the generalization performance of the converged model. Interestingly, they reported that the diagonal $\boldsymbol{C}$ maintained the original convergence speed of the large-batch SGD while the full $\boldsymbol{C}$ did not.

Analyses of the information matrices. Thomas et al. [2020] discussed the similarities and differences between the information matrices, such as the Hessian matrix ($\boldsymbol{H}$), the Fisher information

matrices ($\boldsymbol{F}$), and the uncentered covariance of the gradients ($\boldsymbol{C}$), in cases where they are used for second-order optimization and generalization measures. Kunstner et al. [2019] argued that $\boldsymbol{C}$ (a.k.a empirical Fisher) can not approximate $\boldsymbol{F}$ in second-order optimization.

Others. $\boldsymbol{F}$, $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$ are used for the variational Bayesian inference [Zhang et al., 2018a, Osawa et al., 2019a], the precision of the weights' posterior distribution, and for continual learning [Kirkpatrick et al., 2017, Ritter et al., 2018a]. Inverse of $\boldsymbol{H}$ on training samples naturally appear in influence function, which is one of the well-known methods for improving the interpretation of black-box models [Koh and Liang, 2017]. The inverse of $\boldsymbol{H}$ attracts broader interests in hyper-parameter estimation. As is reviewed in Table 1 of [Lorraine et al., 2019], various hyper-parameters search methods are based on the implicit function theorem, and it requires the inverse of $\boldsymbol{H}$. Then computation of $\boldsymbol{H}$ also naturally appears in meta-learning. For example, iMAML depends on the inverse of $\boldsymbol{H}$ and solves it by the conjugate gradient method [Rajeswaran et al., 2019]. Although the information matrices show advantages in these topics, most studies use only approximations or are only tested on small settings due to the heavy computational costs. The results provided are not comprehensive enough to claim their effects and their approximability in practical settings.

Since the computation of different type/shape matrices requires different time/space complexity (discussed in Section 3.4), it is important to be aware of which type/shape information matrix and what size DNN/task each study is targeting. Typically, for practical computations, experiments are often restricted to very small DNNs or diagonal approximations of matrices are made.

3.4 Complexities of information matrices

In this section, we discuss the time/space complexities of the information matrices of various types/shapes for constructing them, solving the inverses, and evaluating the matrix-vector products.

3.4.1 Complexities of various shapes

First, we discuss the number of elements (space complexity) and time complexity of solving the inverse of the different shapes of the matrix. Table 3.1 lists the number elements and complexity for a fully-connected feed-forward neural network. We can see that, asymptotically, a Kronecker factorized matrix requires only the same amount of space as a diagonal matrix. Although Kronecker factor decomposed matrices require a higher computational cost than diagonal matrices for inversion, this cost can be greatly reduced by distributing the inversion for each Kronecker factor using multiple GPUs (see Chapter 5). For the BatchNorm layer [Ioffe and Szegedy, 2015], a unit-wise matrix takes a very simple form, so the inverse of this is fast (see Chaper 5).

Table 3.1: The number elements (=: S) and space complexity of a matrix (*storage*) and the time complexity of solving the inverse (*inverse*) for a fully-connected neural network. For a Kronecker-factored matrix (*kron*), $S = S_{\text{kron},A} + S_{\text{kron},B}$. To evaluate the complexities, we assume that $P_1 = \cdots = P_L = P/L$ and $M_0 = \cdots = M_{L-1} = K = \sqrt{P/L}$.

shape	# elements (=: S)	*storage*	*inverse*
full	P^2	$\mathcal{O}(P^2)$	$\mathcal{O}(P^3)$
layer-wise	$\sum_{l=1}^{L} P_l^2$	$\mathcal{O}(P^2/L)$	$\mathcal{O}(P^3/L^2)$
unit-wise	$\sum_{l=1}^{L-1} \left(M_l(M_{l-1}+1)^2\right) + K(M_{L-1}+1)^2$	$\mathcal{O}(P\sqrt{P/L})$	$\mathcal{O}(P^2/L)$
diagonal	P	$\mathcal{O}(P)$	$\mathcal{O}(P)$
kron	$\underbrace{\sum_{l=1}^{L}(M_{l-1}+1)^2}_{=:S_{\text{kron},A}} + \underbrace{\sum_{l=1}^{L-1} M_l^2 + K^2}_{=:S_{\text{kron},B}}$	$\mathcal{O}(P)$	$\mathcal{O}(P\sqrt{P/L})$

3.4.2 Complexities of various types/shapes

Next, we discuss the computational costs associated with constructing a matrix of different types and shapes. First, we consider the number of operations for $\boldsymbol{H}$ of various shapes.

Diagonal block of Hessian. The gradient of per-example loss w.r.t. the parameters in layer l is a Kronecker product of two vectors:

$$\nabla_{\boldsymbol{\theta}_l} \ell = \nabla_{\boldsymbol{u}_l} \ell \otimes \boldsymbol{h}_{l-1}^{+} \,,$$

and the diagonal block of $\boldsymbol{H}$ in layer l is a (averaged) Kronecker product of two matrices:

$$\boldsymbol{H}_{l,l} = \left\langle \nabla^2_{\boldsymbol{\theta}_l} \ell \right\rangle = \left\langle \nabla^2_{\boldsymbol{u}_l} \ell \otimes \boldsymbol{h}_{l-1}^{+} \boldsymbol{h}_{l-1}^{+\top} \right\rangle .$$

Hence, the Kronecker-factored form of $\boldsymbol{H}_{l,l}$ [Botev et al., 2017] will be

$$\boldsymbol{H}_{l,l} \approx \left\langle \nabla^2_{\boldsymbol{u}_l} \ell \right\rangle \otimes \left\langle \boldsymbol{h}_{l-1}^{+} \boldsymbol{h}_{l-1}^{+\top} \right\rangle ,$$

where $\left\langle \boldsymbol{h}_{l-1}^{+} \boldsymbol{h}_{l-1}^{+\top} \right\rangle \in \mathbb{R}^{M_{l-1} \times M_{l-1}}$ is the same matrix as $\boldsymbol{A}_{l-1}$ in K-FAC for $\boldsymbol{F}$, $\boldsymbol{F}_{\text{mmc}}$, and $\boldsymbol{C}$. Here, we assume that the activation function ϕ_l ($l = 1, \ldots, L-1$) is an *element-wise* and *piecewise linear* ($\phi_l'' = 0$) function. For example, the *ReLU* function falls into this category:

$$\boldsymbol{h}_l = \phi_l(\boldsymbol{u}_l) \,,$$
$$[\boldsymbol{h}_l]_i = \max([\boldsymbol{u}_l]_i, 0) \,.$$

This makes the calculation of $\boldsymbol{H}_{l,l}$ simpler:

$$\begin{aligned}
\frac{\partial z_k}{\partial [\boldsymbol{u}_l]_m} &= \sum_{m'=1}^{M_l} \frac{\partial z_k}{\partial [\boldsymbol{h}_l]_{m'}} \frac{\partial [\boldsymbol{h}_l]_{m'}}{\partial [\boldsymbol{u}_l]_m} = \frac{\partial z_k}{\partial [\boldsymbol{h}_l]_m} \phi_l'([\boldsymbol{u}_l]_m) \,, \\
\frac{\partial^2 z_k}{\partial [\boldsymbol{u}_l]_m \partial [\boldsymbol{u}_l]_{m'}} &= \phi_l'([\boldsymbol{u}_l]_m) \frac{\partial^2 z_k}{\partial [\boldsymbol{h}_l]_m \partial [\boldsymbol{h}_l]_{m'}} \phi_l'([\boldsymbol{u}_l]_{m'}) \\
&= \sum_{n=1}^{M_{l+1}} \sum_{n'=1}^{M_{l+1}} \phi_l'([\boldsymbol{u}_l]_m) \frac{\partial [\boldsymbol{u}_{l+1}]_n}{\partial [\boldsymbol{h}_l]_m} \frac{\partial^2 z_k}{\partial [\boldsymbol{u}_{l+1}]_n \partial [\boldsymbol{u}_{l+1}]_{n'}} \frac{\partial [\boldsymbol{u}_{l+1}]_{n'}}{\partial [\boldsymbol{h}_l]_{m'}} \phi_l'([\boldsymbol{u}_l]_{m'}) \,,
\end{aligned}$$

where z_k is the k-th elements of $\boldsymbol{z} = f(\boldsymbol{x}) \in \mathbb{R}^K$, and we get a recursion:

$$\nabla^2_{\boldsymbol{u}_l} z_k = \boldsymbol{D}_l^\top \boldsymbol{W}_{l+1}^\top \nabla^2_{\boldsymbol{u}_{l+1}} z_k \boldsymbol{W}_{l+1} \boldsymbol{D}_l \in \mathbb{R}^{M_l \times M_l} ,$$

for $l = 1, \dots, L-1$, where $\boldsymbol{D}_l$ is a diagonal matrix which m-th diagonal element is $\phi_l'([\boldsymbol{u}_l]_m)$ and we define

$$\nabla^2_{\boldsymbol{u}_L} z_k := \nabla^2_{\boldsymbol{z}} z_k = \boldsymbol{0} \in \mathbb{R}^{K \times K} .$$

Hence, for $l = 1, \dots, L-1$,

$$\nabla^2_{\boldsymbol{u}_l} z_k = \boldsymbol{0} ,$$

$$\frac{\partial z_k}{\partial [\boldsymbol{\theta}_l]_i \partial [\boldsymbol{\theta}_l]_j} = \sum_{m=1}^{M_l} \sum_{m'=1}^{M_l} \frac{\partial [\boldsymbol{u}_l]_m}{\partial [\boldsymbol{\theta}_l]_i} \frac{\partial^2 z_k}{\partial [\boldsymbol{u}_l]_m \partial [\boldsymbol{u}_l]_{m'}} \frac{\partial [\boldsymbol{u}_l]_{m'}}{\partial [\boldsymbol{\theta}_l]_j} = 0 ,$$

and

$$\begin{aligned}
\frac{\partial^2 \ell(\boldsymbol{x}, y, \boldsymbol{\theta})}{\partial [\boldsymbol{\theta}_l]_i \partial [\boldsymbol{\theta}_l]_j} &= \sum_{k=1}^K \sum_{k'=1}^K \frac{\partial z_k}{\partial [\boldsymbol{\theta}_l]_i} \frac{\partial^2 h(y, \boldsymbol{z})}{\partial z_k \partial z_{k'}} \frac{\partial z_{k'}}{\partial [\boldsymbol{\theta}_l]_j} \Big|_{\boldsymbol{z} = f(\boldsymbol{x}_n)} + \underbrace{\sum_{k=1}^K \frac{\partial^2 z_k}{\partial [\boldsymbol{\theta}_l]_i \partial [\boldsymbol{\theta}_l]_j} \frac{\partial h(y, \boldsymbol{z})}{\partial z_k} \Big|_{\boldsymbol{z} = f(\boldsymbol{x})}}_{=0} \\
&= \sum_{k=1}^K \sum_{k'=1}^K \frac{\partial z_k}{\partial [\boldsymbol{\theta}_l]_i} \frac{\partial^2 h(y, \boldsymbol{z})}{\partial z_k \partial z_{k'}} \frac{\partial z_{k'}}{\partial [\boldsymbol{\theta}_l]_j} \Big|_{\boldsymbol{z} = f(\boldsymbol{x}_n)} .
\end{aligned}$$

Therefore, the diagonal block $\boldsymbol{H}_{l,l}$ is equivalent to that of the GGN $\boldsymbol{G}$ (and the FIM $\boldsymbol{F}$):

$$\boldsymbol{H}_{l,l} = \boldsymbol{G}_{l,l} = \boldsymbol{F}_{l,l} ,$$

when the activation function ϕ_l ($l = 1, \dots, L-1$) is an element-wise and piecewise linear function (*e.g.*, ReLU).

Table 3.2 summarizes the number of operations and time complexity for constructing a matrix of various types/shapes. Basically, the complexity for construction depends on the number of data (N), the number of parameters (P), and the number of matrix elements (S; defined in Table 3.1). $\boldsymbol{F}$ requires backpropagations depending on the number of outputs of DNN (K), and $\boldsymbol{F}_{\text{mmc}}$ requires backpropagations depending on the number of MC samples (m). F, B_{hidden}, and B_{params} (colored in gray) appear during the evaluation of $\nabla \mathcal{L}_{\mathcal{S}}$. Therefore, if $\nabla \mathcal{L}_{\mathcal{S}}$ has already been evaluated, $\boldsymbol{C}$ can be calculated at a lower cost than $\boldsymbol{F}$ (even if $K = 1$) and $\boldsymbol{F}_{\text{mmc}}$ (even if $m = 1$). Chapter 5 discusses how to take advantage of this to speed up second-order optimization.

3.4.3 Complexities of matrix-vector products

We now discuss the computational cost required for (full) matrix-vector products. Since matrix-vector products are frequently used in the power method and Lanczos algorithm for eigenvalue analysis of information matrices in deep learning [Ghorbani et al., 2019, Yao et al., 2020], and the

Table 3.2: The number of operations (additions and multiplications) and time complexity for constructing a matrix (*construction*) for a fully-connected neural network. F, B_{hidden}, and B_{params} are defined at (2.5), (2.8), and (2.9), respectively. S and *storage* (for each shape) and $S_{\text{kron},A}$ and $S_{\text{kron},B}$ (for *kron*) are defined in Table 3.1. F, B_{hidden}, and B_{params} (colored in gray) appear during the evaluation of $\nabla\mathcal{L}_{\mathcal{S}}$. Therefore, if $\nabla\mathcal{L}_{\mathcal{S}}$ has already been evaluated, $\boldsymbol{C}$ can be calculated at a lower cost than $\boldsymbol{F}$ (even if $K = 1$) and $\boldsymbol{F}_{\text{mmc}}$ (even if $m = 1$). To evaluate the complexities, we assume that $P_1 = \cdots = P_L = P/L$ and $M_0 = \cdots = M_{L-1} = K = \sqrt{P/L}$. We assume that the activation function ϕ_l $(l = 1, \ldots, L-1)$ is an element-wise and piecewise linear function that ensures that the diagonal blocks of $\boldsymbol{H}$ are equivalent to those of $\boldsymbol{G}$ (and $\boldsymbol{F}$).

shape	type	# operations per example	*construction*
-	$\nabla\mathcal{L}_{\mathcal{S}}$	$F + B_{\text{hidden}} + B_{\text{params}}$	$\mathcal{O}(NP)$
lw, uw, diag	$\boldsymbol{H}$	$F + K(B_{\text{hidden}} + B_{\text{params}} + S)$	$\mathcal{O}\left(NK(P + \textit{storage})\right)$
full, lw, uw, diag	$\boldsymbol{F}$	$F + K(B_{\text{hidden}} + B_{\text{params}} + S)$	$\mathcal{O}\left(NK(P + \textit{storage})\right)$
	$\boldsymbol{F}_{\text{mmc}}$	$F + m(B_{\text{hidden}} + B_{\text{params}} + S)$	$\mathcal{O}(Nm(P + \textit{storage}))$
	$\boldsymbol{C}$	$F + B_{\text{hidden}} + B_{\text{params}} + S$	$\mathcal{O}(N(P + \textit{storage}))$
kron	$\boldsymbol{H}, \boldsymbol{F}$	$F + S_{\text{kron},A} + K(B_{\text{hidden}} + S_{\text{kron},B})$	$\mathcal{O}(NKP)$
	$\boldsymbol{F}_{\text{mmc}}$	$F + S_{\text{kron},A} + m(B_{\text{hidden}} + S_{\text{kron},B})$	$\mathcal{O}(NmP)$
	$\boldsymbol{C}$	$F + S_{\text{kron},A} + B_{\text{hidden}} + S_{\text{kron},B}$	$\mathcal{O}(NP)$

conjugate gradient method for second-order optimization [Martens, 2010], it is useful to understand the computational cost per type ($\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{\text{mmc}}$, and $\boldsymbol{C}$). In this section, we discuss the computational cost of the fast matrix-vector product calculation method introduced by Schraudolph [2002]. Table 3.3 lists the number of operations and time complexity for a matrix-vector product. The Hessian vector product can be computed efficiently on top of the auto-differentiation framework without the need to compute the full Hessian $\boldsymbol{H}$ explicitly:

$$\boldsymbol{H}(\boldsymbol{\theta})\boldsymbol{v} = \nabla^2\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})\boldsymbol{v} = \nabla\left(\nabla\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})^\top\boldsymbol{v}\right) .$$

Therefore, the Hessian vector product can be calculated with a backpropagation of the inner product $\nabla\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})^\top\boldsymbol{v}$ (a scalar value). With the definitions of $\boldsymbol{F}$, $\boldsymbol{F}_{\text{mmc}}$, and $\boldsymbol{C}$, the matrix-vector product can also be calculated without constructing the full matrix:

$$\boldsymbol{M}\boldsymbol{v} = \left\langle \mathbb{E}\left[\nabla\ell\nabla\ell^\top\right]\right\rangle\boldsymbol{v} = \left\langle \mathbb{E}\left[\nabla\ell\left(\nabla\ell^\top\boldsymbol{v}\right)\right]\right\rangle .$$

Once we get the per-example gradient $\nabla\ell$, we can calculate the inner product $\nabla\ell^\top\boldsymbol{v}$ (a scalar value) and $\nabla\ell(\nabla\ell^\top\boldsymbol{v})$ (a vector), each costs P operations. $\mathbb{E}[\cdot]$ and ℓ depend on the definition of matrix type ($\boldsymbol{F}$ (3.3), $\boldsymbol{F}_{\text{mmc}}$ (3.4), or $\boldsymbol{C}$ (3.5)).

Table 3.3: The number of operations (additions and multiplications) and time complexity for evaluating a matrix-vector product (*mvp*) for an L-layered fully-connected neural network. $\boldsymbol{v} \in \mathbb{R}^P, \boldsymbol{u} \in \mathbb{R}^{NK}$. To estimate the complexity, we assume that $P_l = P/L$ for $l = 1, \ldots, L$ for simplicity.

type	# operations per example	*mvp*
$\boldsymbol{F}\boldsymbol{v}$	$F + K(B_{\text{hidden}} + B_{\text{params}} + 2P)$	$\mathcal{O}(NKP)$
$\boldsymbol{F}_{\text{mmc}}\boldsymbol{v}$	$F + m(B_{\text{hidden}} + B_{\text{params}} + 2P)$	$\mathcal{O}(NmP)$
$\boldsymbol{C}\boldsymbol{v}$	$F + B_{\text{hidden}} + B_{\text{params}} + 2P$	$\mathcal{O}(NP)$
$\mathcal{J}^\top \mathcal{J} \boldsymbol{v}$	$F + 2(B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K)) + \sum_{l=1}^{L} \sum_{l'=l}^{L} P_{l'}$	$\mathcal{O}(NLP)$
$\mathcal{J} \mathcal{J}^\top \boldsymbol{u}$	$F + B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K) + \sum_{l=1}^{L} \sum_{l'=l}^{L} P_{l'}$	$\mathcal{O}(NLP)$

3.5 GGN/FIM-vector product by standard auto-differentiation libraries

3.5.1 GGN/FIM-vector product

Given a vector $\boldsymbol{v} \in \mathbb{R}^P$, we consider to evaluate the GGN/FIM-vector product:

$$\boldsymbol{G}\boldsymbol{v} = \boldsymbol{F}\boldsymbol{v} = \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{J}_f(\boldsymbol{x}_n) \right\rangle \boldsymbol{v} = \mathcal{J}^\top \mathcal{J} \boldsymbol{v} \in \mathbb{R}^P \,,$$

where we define

$$\mathcal{J} := \frac{1}{\sqrt{N}} \begin{bmatrix} \boldsymbol{J}_f(\boldsymbol{x}_1) & \cdots & \boldsymbol{J}_f(\boldsymbol{x}_N) \end{bmatrix} \in \mathbb{R}^{NK \times P} \,.$$

First, the Jacobian-vector product $\mathcal{J}\boldsymbol{v} \in \mathbb{R}^{NK}$ is obtained as the partial derivative of $\boldsymbol{u}^\top \mathcal{J} \boldsymbol{v}$ (a scalar) w.r.t. a vector $\boldsymbol{u} \in \mathbb{R}^{NK}$:

$$(\mathcal{J}\boldsymbol{v})^\top = \boldsymbol{v}^\top \mathcal{J}^\top = \frac{\partial}{\partial \boldsymbol{u}} \boldsymbol{v}^\top \mathcal{J}^\top \boldsymbol{u} \in \mathbb{R}^{NK} \,. \tag{3.7}$$

For $\boldsymbol{u} \in \mathbb{R}^{NK}$, we can calculate the transposed Jacobian-vector product $\mathcal{J}^\top \boldsymbol{u}$ by a backward-pass on the neural network. Then, we get the GGN/FIM-vector product $\mathcal{J}^\top \mathcal{J} \boldsymbol{v} = \mathcal{J}^\top (\mathcal{J}\boldsymbol{v})$ (transposed Jacobian-vector product) by another backward-pass.

Now we discuss the number of operations required for evaluating the (per-examples) Jacobian-vector product $\boldsymbol{J}_f(\boldsymbol{x})\boldsymbol{v}$, where

$$\begin{aligned} \boldsymbol{v} &= \begin{bmatrix} \boldsymbol{v}_1^\top & \cdots & \boldsymbol{v}_L^\top \end{bmatrix}^\top \in \mathbb{R}^P \,, \\ \boldsymbol{v}_l &= \begin{bmatrix} \boldsymbol{v}_{l,1}^\top & \cdots & \boldsymbol{v}_{l,M_l}^\top \end{bmatrix}^\top \in \mathbb{R}^{P_l} \;\; (l = 1, \ldots, L) \,, \end{aligned}$$

$\boldsymbol{v}_{l,i} \in \mathbb{R}^{M_{l-1}+1}$ $(i = 1, \ldots, M_l)$, and

$$\boldsymbol{J}_f \boldsymbol{v} = \boldsymbol{J}_{f,\boldsymbol{\theta}} \boldsymbol{v} = \sum_{l=1}^{L} \boldsymbol{J}_{f,\boldsymbol{\theta}_l} \boldsymbol{v}_l \in \mathbb{R}^K \,.$$

For layer l,

$$\underbrace{\boldsymbol{J}_{f,\boldsymbol{\theta}_l} \boldsymbol{v}_l}_{K \times 1} = \underbrace{\boldsymbol{J}_{f,\boldsymbol{h}_{L-1}}}_{K \times M_{L-1}} \underbrace{\boldsymbol{J}_{\boldsymbol{h}_{L-1},\boldsymbol{u}_{L-1}}}_{M_{L-1} \times M_{L-1}} \cdots \underbrace{\boldsymbol{J}_{\boldsymbol{u}_{l+1},\boldsymbol{h}_l}}_{M_{l+1} \times M_l} \underbrace{\boldsymbol{J}_{\boldsymbol{h}_l,\boldsymbol{u}_l}}_{M_l \times M_l} \underbrace{\boldsymbol{J}_{\boldsymbol{u}_l,\boldsymbol{\theta}_l}}_{M_l \times P_l} \underbrace{\boldsymbol{v}_l}_{P_l \times 1} \,,$$

and we can calculate $\boldsymbol{J}_{f,\boldsymbol{\theta}_l}\boldsymbol{v}_l$ by *forward-propagating* $\boldsymbol{v}_l$:

$$
\begin{aligned}
[\boldsymbol{J}_{\boldsymbol{u}_l,\boldsymbol{\theta}_l}\boldsymbol{v}_l]_i &= {\boldsymbol{h}_{l-1}^+}^\top \boldsymbol{v}_{l,i} \quad (i = 1, \dots, M_l)\,, \\
\boldsymbol{J}_{\boldsymbol{h}_{l'},\boldsymbol{\theta}_l}\boldsymbol{v}_l &= \phi'_{l'}(\boldsymbol{u}_{l'}) \odot \boldsymbol{J}_{\boldsymbol{u}_{l'},\boldsymbol{\theta}_l}\boldsymbol{v}_l \in \mathbb{R}^{M_{l'}} \quad (l' = l, \cdots, L-1)\,, \\
\boldsymbol{J}_{\boldsymbol{u}_{l'},\boldsymbol{\theta}_l}\boldsymbol{v}_l &= \boldsymbol{W}_{l'}\boldsymbol{J}_{\boldsymbol{h}_{l'-1},\boldsymbol{\theta}_l}\boldsymbol{v}_l \in \mathbb{R}^{M_{l'}} \quad (l' = l+1, \cdots, L-1)\,, \\
\boldsymbol{J}_{f,\boldsymbol{\theta}_l}\boldsymbol{v}_l &= \boldsymbol{W}_L\boldsymbol{J}_{\boldsymbol{h}_{L-1},\boldsymbol{\theta}_l}\boldsymbol{v}_l \in \mathbb{R}^K\,.
\end{aligned}
$$

The number of operations (additions and multiplications) for this is:

$$
\begin{aligned}
&\underbrace{(M_{l-1}+1)M_l}_{\boldsymbol{J}_{\boldsymbol{u}_l,\boldsymbol{\theta}_l}\boldsymbol{v}_l} + \sum_{l'=l}^{L-1} \underbrace{M_{l'}}_{\phi'_{l'}(\boldsymbol{u}_{l'})\odot\boldsymbol{J}_{\boldsymbol{u}_{l'},\boldsymbol{\theta}_l}\boldsymbol{v}_l} + \sum_{l'=l+1}^{L} \underbrace{M_{l'}M_{l'-1}}_{\boldsymbol{W}_L\boldsymbol{J}_{\boldsymbol{h}_{L-1},\boldsymbol{\theta}_l}\boldsymbol{v}_l} \\
&= P_l + \sum_{l'=l+1}^{L} P_{l'} = \sum_{l'=l}^{L} P_{l'}\,,
\end{aligned}
$$

and the total number of operations for $\boldsymbol{J}_f\boldsymbol{v}$ is:

$$
\sum_{l=1}^{L}\sum_{l'=l}^{L} P_{l'}\,.
$$

Hence, the number of operations for the GGN/FIM-vector product $\mathcal{J}^\top\mathcal{J}\boldsymbol{v} = \left\langle \boldsymbol{J}_f(\boldsymbol{x}_n)^\top \boldsymbol{J}_f(\boldsymbol{x}_n)\boldsymbol{v} \right\rangle$ is:

$$
C_{ggn-vec} = N\left(F + 2(B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K)) + \sum_{l=1}^{L}\sum_{l'=l}^{L} P_{l'}\right)\,.
$$

By assuming that $P_l = P/L$ for $l = 1, \dots, L$,

$$
\begin{aligned}
\sum_{l=1}^{L}\sum_{l'=l}^{L} P_{l'} &= \sum_{l=1}^{L}\sum_{l'=l}^{L} P/L \\
&= \frac{1}{L}\sum_{l=1}^{L}(L+1-l)P \\
&= LP + P - \frac{L+1}{2}P \\
&= \frac{L+1}{2}P\,,
\end{aligned}
$$

and

$$
C_{ggn-vec} = N\left(F + 2(B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K)) + \frac{L+1}{2}P\right)\,.
$$

3.5.2 NTK-vector product

Using the property of the eigenvalues of the product of two matrices

$$
\text{eig}\left(\boldsymbol{A}\boldsymbol{B}\right) = \text{eig}\left(\boldsymbol{B}\boldsymbol{A}\right)\,,
$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times m}$, we get

$$\mathrm{eig}\left(\mathcal{J}^\top \mathcal{J}\right) = \mathrm{eig}\left(\mathcal{J}\mathcal{J}^\top\right) .$$

The matrix $\mathcal{J}\mathcal{J}^\top \in \mathbb{R}^{NK \times NK}$ is known as the *Neural Tangent Kernel* (NTK) [Jacot et al., 2018], which is used for theoretical analysis of training dynamics and generalization performance of infinite-width neural networks. Hence, the eigenvalues of the GGN/FIM are equivalent to that of the NTK, and *we can use the NTK-vector product $\mathcal{J}\mathcal{J}^\top \boldsymbol{u}$ in iterative methods (e.g., power method, Lanczos algorithm) for evaluating the eigenvalues of the GGN/FIM.* In this section, we show that *the NTK-vector product is faster than the GGN/FIM-vector product.*

First, we evaluate the transposed Jacobian-vector product $\mathcal{J}^\top \boldsymbol{u} \in \mathbb{R}^P$ by a backward-pass. Then we replicate the resulting vector as $\boldsymbol{g}$:

$$\boldsymbol{g} \leftarrow \mathcal{J}^\top \boldsymbol{u} ,$$

where $\boldsymbol{g}$ is a constant vector and *does not hold the computational graphs* constructed at the time of backward-pass. Since only $\mathcal{J}^\top \boldsymbol{u}$ holds the computational graph, computing the partial derivative of $\boldsymbol{g}\mathcal{J}^\top \boldsymbol{u}$ w.r.t. $\boldsymbol{u}$ gives us the NTK-vector product:

$$\frac{\partial}{\partial \boldsymbol{u}} \boldsymbol{g}^\top \mathcal{J}^\top \boldsymbol{u} = \boldsymbol{g}^\top \mathcal{J}^\top = \boldsymbol{u}^\top \mathcal{J}\mathcal{J}^\top = \left(\mathcal{J}\mathcal{J}^\top \boldsymbol{u}\right)^\top . \tag{3.8}$$

Therefore, *the NTK-vector product $\mathcal{J}\mathcal{J}^\top \boldsymbol{u}$ can be calculated with one less backward-pass than the GGN/FIM-vector product $\mathcal{J}^\top \mathcal{J}\boldsymbol{v}$.* Hence, the number of operations for the NTK-vector product $\mathcal{J}\mathcal{J}^\top \boldsymbol{u}$ is:

$$C_{ntk-vec} = N \left(F + B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K) + \sum_{l=1}^{L} \sum_{l'=l}^{L} P_{l'} \right) .$$

By assuming $P_l = P/L$ for $l = 1, \ldots, L$,

$$C_{ntk-vec} = N \left(F + B_{\text{hidden}} + B_{\text{params}} - \mathcal{O}(K) + \frac{L+1}{2} P \right) .$$

Chapter 4

Second-order optimization in deep learning

4.1 Second-order optimization

In second-order optimization, the function is approximated using up to the second-order terms of the Taylor expansion. Then, the next update step is chosen to minimize the quadratic function within the specific step size:

$$\min_{\|\epsilon\|_2 \leq const} \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta} + \epsilon) \approx \min_{\|\epsilon\|_2 \leq const} \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \epsilon^\top \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \frac{1}{2} \epsilon^\top \boldsymbol{H}(\boldsymbol{\theta}) \epsilon \,.$$

This leads to the second-order optimization:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{H}(\boldsymbol{\theta}^{(t)})^{-1} \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}) \,, \tag{4.1}$$

where $\eta > 0$ is the learning rate ($\eta = 1$ for the Newton method). In deep learning, $\boldsymbol{H}$ is know to have the negative eigenvalues [Dauphin et al., 2014]. So the generalized Gauss-Newton matrix $\boldsymbol{G}$, which is positive semidefinite, is often used for approximate second-order optimization in deep learning Martens [2010], Botev et al. [2017]:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{G}(\boldsymbol{\theta}^{(t)})^{-1} \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}^{(t)}) \,. \tag{4.2}$$

4.2 Natural gradient descent

Natural Gradient Descent (NGD)[Amari, 1998] is an optimizer which updates the parameters using the first-order gradient of the loss function preconditioned by the *Fisher information matrix* (FIM) of the probabilistic model. NGD can also be referred to as first-order optimization in the Riemannian space. In NGD, we use the steepest descent method (using up to the first-order terms of the Taylor expansion) to find the solution that minimizes the objective within a range that the KL divergence between the prediction distributions before and after the parameter update does not exceed a specific

value.

$$\min_{\mathrm{KL}(p_{\boldsymbol{\theta}} \,\|\, p_{\boldsymbol{\theta}+\epsilon}) \leq const} \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}+\epsilon) \approx \min_{\mathrm{KL}(p_{\boldsymbol{\theta}} \,\|\, p_{\boldsymbol{\theta}+\epsilon}) \leq const} \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \epsilon^{\top} \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) \,. \tag{4.3}$$

When we define $p_{\boldsymbol{\theta}}$ as following:

$$p_{\boldsymbol{\theta}}(y|\boldsymbol{x}) = p(y|f(\boldsymbol{x};\boldsymbol{\theta})) \,,$$

the Kullback-Leibler (KL) divergence will be:

$$\mathrm{KL}(p_{\boldsymbol{\theta}} \,\|\, p_{\boldsymbol{\theta}+\epsilon}) := \mathbb{E}_{y \sim p_{\boldsymbol{\theta}}(y|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(y|\boldsymbol{x}) - \log p_{\boldsymbol{\theta}+\epsilon}(y|\boldsymbol{x}) \right] \,.$$

Assuming $\epsilon \to \mathbf{0}$,

$$\mathbb{E}\left[\log p_{\boldsymbol{\theta}+\epsilon}\right] \approx \mathbb{E}\left[\log p_{\boldsymbol{\theta}}\right] + \epsilon^{\top} \mathbb{E}\left[\nabla \log p_{\boldsymbol{\theta}}\right] + \frac{1}{2} \epsilon^{\top} \mathbb{E}\left[\nabla^2 \log p_{\boldsymbol{\theta}}\right] \epsilon \,,$$

and

$$\begin{aligned} \mathrm{KL}(p_{\boldsymbol{\theta}} \,\|\, p_{\boldsymbol{\theta}+\epsilon}) &\approx -\epsilon^{\top} \mathbb{E}\left[\nabla \log p_{\boldsymbol{\theta}}\right] - \frac{1}{2} \epsilon^{\top} \mathbb{E}\left[\nabla^2 \log p_{\boldsymbol{\theta}}\right] \epsilon \\ &= \frac{1}{2} \epsilon^{\top} \mathbb{E}\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right] \epsilon \\ &= \frac{1}{2} \epsilon^{\top} \boldsymbol{F}(\boldsymbol{\theta}) \epsilon \end{aligned}$$

the last equation comes from the fact that the expected Hessian is equivalent to the FIM [Pascanu and Bengio, 2014]. Using this approximation of the KL divergence, the NGD finds the solution which minimizes the linear function within the range specified by $\| \cdot \|_{\boldsymbol{F}}$:

$$\min_{\|\epsilon\|_{\boldsymbol{F}} \leq const} \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \epsilon^{\top} \nabla \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) \,,$$

where we define

$$\|\boldsymbol{v}\|_{\boldsymbol{M}} := \boldsymbol{v}^{\top} \boldsymbol{M} \boldsymbol{v} \,,$$

for a vector $\boldsymbol{v} \in \mathbb{R}^d$ and a matrix $\boldsymbol{M} \in \mathbb{R}^{d \times d}$. This minimization results in the following NGD update rule:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{F}(\boldsymbol{\theta}^{(t)})^{-1} \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \,, \tag{4.4}$$

Hence, $\boldsymbol{F}$ is the curvature matrix in the Riemannian space [Amari, 1998]. As discussed in Chapter 3, the generalized Gauss-Newton matrix $\boldsymbol{G}$ is equivalent to the FIM $\boldsymbol{F}$ for the MSE loss and softmax cross-entropy loss. This allows us to interpret the NGD as an approximate second-order method [Martens, 2020].

4.2.1 Common approaches to approximate natural gradient in deep learning

To realize a practical NGD training for deep neural networks, it is common to make the following approximations to the $\boldsymbol{F}$ [Martens and Grosse, 2015, Botev et al., 2017, Zhang et al., 2018a]:

- **Layer-wise block-diagonal approximation.** We assume that the correlation between parameters in different layers is negligible and can be ignored. Hence, $\boldsymbol{F}$ is approximated as a layer-wise block diagonal matrix:

$$\boldsymbol{F} \approx \begin{bmatrix} \boldsymbol{F}_1 & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{F}_L \end{bmatrix}.$$

  This assumption significantly reduces the computational cost of inverting $\boldsymbol{F}$ especially when the number of parameters P is large (See also Chapter 3).

- **Stochastic natural gradient.** We approximate the average over the entire dataset $\langle \cdot \rangle_{\mathcal{S}}$ using the average over a mini-batch $\langle \cdot \rangle_{\mathcal{B}}$. This enables calculation of $\boldsymbol{F}$ and evaluation of NGD during mini-batch stochastic learning.

- **Monte Carlo estimation.** We approximate the expectation over the model predictive distribution $\mathbb{E}_{k \sim p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)}[\cdot]$ using a single Monte Carlo sample (a single backward-pass): $\boldsymbol{F} \approx \boldsymbol{F}_{\mathrm{1mc}}$. We note that for a K-class classification model, K backward-passes are required to calculate $\boldsymbol{F}$, while $\boldsymbol{F}_{\mathrm{1mc}}$ requires one backward-pass (See also Chapter 3).

Using these approximations, we estimate the FIM $\boldsymbol{F}_l \in \mathbb{R}^{P_l \times P_l}$ for the l-th layer using a Monte Carlo sample $k \sim p_{\boldsymbol{\theta}}(k|\boldsymbol{x})$ for each input $\boldsymbol{x}$ in a mini-batch $\mathcal{B}$:

$$\boldsymbol{F}_l(\boldsymbol{\theta}) \approx \boldsymbol{F}_{\mathrm{1mc}l}(\boldsymbol{\theta}) := \left\langle \nabla_{\boldsymbol{\theta}_l} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}) \nabla_{\boldsymbol{\theta}_l} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x})^\top \right\rangle_{\mathcal{B}}. \tag{4.5}$$

With this $\boldsymbol{F}_{\mathrm{1mc},l}$, the parameters $\boldsymbol{\theta}_l \in \mathbb{R}^{P_l}$ for the l-th layer are then updated using the FIM preconditioned gradients:

$$\boldsymbol{\theta}_l^{(t+1)} \leftarrow \boldsymbol{\theta}_l^{(t)} - \eta \left({\boldsymbol{F}_{\mathrm{1mc}}}_l^{(t)}(\boldsymbol{\theta}^{(t)}) + \lambda \boldsymbol{I} \right)^{-1} \nabla_{\boldsymbol{\theta}_l} \mathcal{L}^{(t)}(\boldsymbol{\theta}^t), \tag{4.6}$$

where the damping value $\lambda > 0$ is added to the diagonal of the matrix to avoid the divergence of the gradient. Osawa et al. [2020] proposed to further approximate this $\boldsymbol{F}_{\mathrm{1mc}l}$ with the gradient covariance $\boldsymbol{C}_l$, which can be evaluated in a cheaper cost (See also Chapter 3).

4.3 Conjugate gradient method for second-order optimization

Given a positive semidefinite matrix $\boldsymbol{M} \in \mathbb{R}^{P \times P}$ ($\boldsymbol{M} \succeq 0$), we consider to update the parameters $\boldsymbol{\theta} \in \mathbb{R}^P$ of a deep neural network with the preconditioned gradient:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \left(\boldsymbol{M}^{-1} + \lambda \boldsymbol{I}\right) \nabla \mathcal{L}(\boldsymbol{\theta}^{(t)}) \,,$$

where $\lambda > 0$ is the damping value. To do this, we need to solve the following system of linear equations w.r.t. $\boldsymbol{x} \in \mathbb{R}^P$:

$$(\boldsymbol{M} + \lambda \boldsymbol{I})\, \boldsymbol{x} = \nabla \mathcal{L}(\boldsymbol{\theta}) \,.$$

Since the number of parameters of deep neural networks, P, is huge, it is infeasible to solve the inverse matrix explicitly ($\mathcal{O}(P^3)$). One of the common approaches to solve this problem in deep learning is to use the *conjugate gradient (CG) method*. Martens [2010] adopted the CG method with $\boldsymbol{M} = \boldsymbol{G}$ for optimizing deep auto-encoders. The CG method is an iterative method and requires matrix-vector product calculation at each iteration. See Chapter 3 for matrix-vector product calculations for various information matrices ($\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{\text{mmc}}$, and $\boldsymbol{C}$). Taking into about the damping value λ, the matrix-vector product is calculated as:

$$(\boldsymbol{M} + \lambda \boldsymbol{I})\, \boldsymbol{v} = \boldsymbol{M}\boldsymbol{v} + \lambda \boldsymbol{v} \,.$$

The calculations for the CG is described in Algorithm 1. At the beginning of the algorithm, we calculate the gradient of loss $\nabla \mathcal{L}$. Note that because the matrix-vector product calculation for the information matrix of a neural network involves backpropagation, the computation graphs constructed during the gradient calculation must be retained for reuse. The advantage of the CG method is that it relies only on the matrix-vector product and thus saves memory without the need to explicitly build the matrix or solve the inverse. However, it is known that the information matrix of a neural network has a long tail eigenvalue distribution [Amari et al., 2019], i.e. all but the upper eigenvalues take almost zero values, and the condition number of $\boldsymbol{M}$ (the maximum eigenvalue divided by the minimum eigenvalue), which determines the speed of convergence of the CG method, is large and it takes a lot of iterations until the convergence. Since matrix-vector product calculations require higher computational costs than the usual backpropagation (see Chapter 3), realizing second-order optimization by CG methods during training is not practical in most cases.

4.4 Kronecker-factored Approximate Curvature (K-FAC)

In deep learning, instead of CG methods, the predominant method is to approximate the matrix to make it easier to compute the inverse matrix [Roux et al., 2008, Pascanu and Bengio, 2014,

Algorithm 1: Conjugate gradient method for second-order optimization

input: input data $\mathcal{X} \in \mathbb{R}^{N \times M_0}$
forward and backward $\nabla\mathcal{L}(\boldsymbol{\theta}) \in \mathbb{R}^P$; `// computational graph has to be retained`
$\boldsymbol{x}^{(0)} \leftarrow \mathbf{0} \in \mathbb{R}^P$;
$\boldsymbol{p}^{(0)}, \boldsymbol{r}^{(0)} \leftarrow \nabla\mathcal{L}(\boldsymbol{\theta}) \in \mathbb{R}^P$;
$r^{(0)} \leftarrow {\boldsymbol{r}^{(0)}}^\top \boldsymbol{r}^{(0)}$;
$k \leftarrow 0$;
repeat
  $\boldsymbol{u} \leftarrow \boldsymbol{M}\boldsymbol{p} \in \mathbb{R}^P$;
  $\boldsymbol{u} \leftarrow \boldsymbol{u} + \gamma \boldsymbol{p}^{(k)}$; `// damping`
  $m \leftarrow {\boldsymbol{p}^{(k)}}^\top \boldsymbol{u}$;
  $\alpha \leftarrow r^{(k)}/m$;
  $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha \boldsymbol{p}^{(k)}$;
  $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{r}^{(k)} - \alpha \boldsymbol{u}$;
  $r^{(k+1)} \leftarrow {\boldsymbol{r}^{(k+1)}}^\top \boldsymbol{r}^{(k+1)}$;
  $\beta \leftarrow r^{(k+1)}/r^{(k)}$;
  $\boldsymbol{p}^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)} + \beta \boldsymbol{p}^{(k)}$;
  $k \leftarrow k + 1$;
until $r^{(k+1)}$ *is small enough*;
return $\boldsymbol{x}^{(k+1)}$

Thomas, 2014, Grosse and Salakhutdinov, 2015, Martens and Grosse, 2015, Grosse and Martens, 2016, Botev et al., 2017, Mishkin et al., 2018, George et al., 2018, Martens and Johnson, 2018]. Kronecker-Factored Approximate Curvature (K-FAC) Martens and Grosse [2015] is one of the most poputer second-order optimization method in today's deep learning research, that is based on an approximation of $\boldsymbol{F}_{1\mathrm{mc}}$. K-FAC further approximates the layer-wise FIM $\boldsymbol{F}_{1\mathrm{mc}l}$ as a Kronecker product of two matrices:

$$\boldsymbol{F}_{1\mathrm{mc}l} \approx \boldsymbol{B}_l \otimes \boldsymbol{A}_{l-1} =: \tilde{\boldsymbol{F}}_l \,. \tag{4.7}$$

This is called *Kronecker factorization* and $\boldsymbol{B}_l$ and $\boldsymbol{A}_{l-1}$ are called *Kronecker factors*. $\boldsymbol{B}_l$ is computed from the gradient of the loss w.r.t. the output of the l-th layer, and $\boldsymbol{A}_{l-1}$ is computed from the activation of the $(l-1)$-th layer (the input of l-th layer). The definition and the sizes of the Kronecker factors $\boldsymbol{B}_l/\boldsymbol{A}_{l-1}$ depend on the dimension of the output/input and the type of layer [Martens and Grosse, 2015, Grosse and Martens, 2016, Martens and Johnson, 2018, Zhang et al., 2019a].

4.4.1 K-FAC for fully-connected layers

In a fully-connected (FC) layer of a feed-forward DNN, the output $\boldsymbol{s}_l \in \mathbb{R}^{d_l}$ is calculated as

$$\boldsymbol{u}_l \leftarrow \boldsymbol{W}_l \boldsymbol{h}_{l-1}, \tag{4.8}$$

where $\boldsymbol{a}_{l-1} \in \mathbb{R}^{d_{l-1}}$ is the input to this layer (the activation from the previous layer), and $\boldsymbol{W}_l \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix (the bias is ignored for simplicity). The Kronecker factors for this FC

layer are defined as

$$\begin{aligned} \boldsymbol{B}_l &:= \left\langle \nabla_{\boldsymbol{u}_l} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}) \nabla_{\boldsymbol{u}_l} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x})^\top \right\rangle , \\ \boldsymbol{A}_{l-1} &:= \left\langle \boldsymbol{h}_{l-1} \boldsymbol{h}_{l-1}^\top \right\rangle , \end{aligned} \tag{4.9}$$

and $\boldsymbol{B}_l \in \mathbb{R}^{d_l \times d_l}, \boldsymbol{A}_{l-1} \in \mathbb{R}^{d_{l-1} \times d_{l-1}}$ [Martens and Grosse, 2015]. From this definition, we can consider that K-FAC is based on an assumption that the input to the layer and the gradient w.r.t. the layer output are statistically independent.

4.4.2 K-FAC for convolutional layers

In a convolutional (Conv) layer of a feed-forward DNN, the output $\mathcal{S}_l \in \mathbb{R}^{c_l \times h_l \times w_l}$ is calculated as

$$\begin{aligned} \boldsymbol{M}_{\mathsf{H}_{l-1}} &\leftarrow \mathrm{im2col}(\mathsf{H}_{l-1}) \in \mathbb{R}^{C_{l-1} F_l^2 \times H_l W_l} , \\ \boldsymbol{M}_{\mathsf{U}_l} &\leftarrow \boldsymbol{W}_l \boldsymbol{M}_{\mathsf{H}_{l-1}} \in \mathbb{R}^{C_l \times H_l W_l} , \\ \mathsf{U}_l &\leftarrow \mathrm{reshape}(\boldsymbol{M}_{\mathsf{U}_l}) \in \mathbb{R}^{C_l \times H_l \times W_l} , \end{aligned} \tag{4.10}$$

where $\mathsf{H}_l \in \mathbb{R}^{C_l \times H_{l-1} \times W_{l-1}}$ is the input to layer l, and $\boldsymbol{W}_l \in \mathbb{R}^{C_l \times C_{l-1} F_l^2}$ is the weight matrix (the bias is ignored for simplicity). c_l, c_{l-1} are the number of output, input channels, respectively, and k_l is the kernel size (assuming square kernels for simplicity). For each example in the mini-batch, the `im2col` operator[1] converts the input tensor to a matrix so that we can get the result of a convolution by a matrix multiplication. The Kronecker factors for this Conv layer are defined as

$$\begin{aligned} \boldsymbol{B}_l &:= \left\langle \nabla_{\boldsymbol{M}_{\mathsf{U}_l}} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n) \nabla_{\boldsymbol{M}_{\mathsf{U}_l}} \log p_{\boldsymbol{\theta}}(k|\boldsymbol{x}_n)^\top \right\rangle , \\ \boldsymbol{A}_{l-1} &:= \frac{1}{H_l W_l} \left\langle \boldsymbol{M}_{\mathsf{H}_{l-1}} \boldsymbol{M}_{\mathsf{H}_{l-1}}^\top \right\rangle , \end{aligned} \tag{4.11}$$

and $\boldsymbol{B}_l \in \mathbb{R}^{C_l \times C_l}, \boldsymbol{A}_{l-1} \in \mathbb{R}^{C_{l-1} F_l^2 \times C_{l-1} F_l^2}$ [Grosse and Martens, 2016].

4.4.3 Inverting Kronecker-factored FIM

By the property of the Kronecker product and the *factored Tikhonov damping technique* used in [Martens and Grosse, 2015], the inverse of $\tilde{\boldsymbol{F}}_l + \lambda \boldsymbol{I}$ is approximated by the Kronecker product of the inverse of each Kronecker factor

$$\begin{aligned} \left(\tilde{\boldsymbol{F}}_l + \lambda \boldsymbol{I} \right)^{-1} &= \left(\boldsymbol{B}_l \otimes \boldsymbol{A}_{l-1} + \lambda \boldsymbol{I} \right)^{-1} \\ &\approx \left(\boldsymbol{B}_l + \frac{1}{\pi_l} \sqrt{\lambda} \boldsymbol{I} \right)^{-1} \otimes \left(\boldsymbol{A}_{l-1} + \pi_l \sqrt{\lambda} \boldsymbol{I} \right)^{-1} , \end{aligned} \tag{4.12}$$

where π_l^2 is the average eigenvalue of $\boldsymbol{A}_{l-1}$ divided by the average eigenvalue of $\boldsymbol{B}_l$ (which can be computed using the trace). $\pi > 0$ because both $\boldsymbol{B}_l$ and $\boldsymbol{A}_{l-1}$ are positive-semidefinite matrices as defined above.

[1] See Chainer documentation (`https://docs.chainer.org/en/stable/reference/generated/chainer.functions.im2col.html`) for detail.

4.5 Natural gradient and neural tangent kernels

The *Neural Tangent Kernel* (NTK) [Jacot et al., 2018] is used for theoretical analysis of training dynamics and generalization performance of infinite-width neural networks, which is defined as:

$$\Theta := \mathcal{J}\mathcal{J}^\top \in \mathbb{R}^{NK \times NK},$$

where we define

$$\mathcal{J} = \frac{1}{\sqrt{N}} \begin{bmatrix} \boldsymbol{J}(x_1)^\top & \cdots & \boldsymbol{J}(x_N)^\top \end{bmatrix}^\top \in \mathbb{R}^{NK \times P}.$$

Using this $\mathcal{J}$, the FIM can be written as:

$$\boldsymbol{F} = \mathcal{J}^\top \boldsymbol{\Lambda} \mathcal{J} \in \mathbb{R}^{P \times P},$$

where

$$\boldsymbol{\Lambda} = \operatorname{diag}\begin{pmatrix} \mathcal{H}_f(\boldsymbol{x}_1) & \cdots & \mathcal{H}_f(\boldsymbol{x}_N) \end{pmatrix} \in \mathbb{R}^{NK \times NK}$$
$$\mathcal{H}_f(\boldsymbol{x}) := \nabla^2_{\boldsymbol{z}} h(y, \boldsymbol{z})\Big|_{\boldsymbol{z}=f(\boldsymbol{x})} = \operatorname{diag}(\boldsymbol{p}) - \boldsymbol{p}\boldsymbol{p}^\top \in \mathbb{R}^{K \times K}.$$

Using the Sherman-Morrison-Woodbury identity [Petersen and Pedersen, 2012]:

$$(AB + I)^{-1} A = A(BA + I)^{-1},$$

we can get the natural gradient descent (NGD) by the NTK:

$$\begin{aligned}(\boldsymbol{F}(\boldsymbol{\theta}) + \gamma \boldsymbol{I}_P)^{-1} \frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &= \underbrace{\left(\frac{1}{N} \mathcal{J}^\top \boldsymbol{\Lambda} \mathcal{J} + \gamma \boldsymbol{I}_P \right)^{-1}}_{P \times P} \mathcal{J}^\top \frac{\partial}{\partial f} \mathcal{L}(\boldsymbol{\theta}) \\ &= \mathcal{J}^\top \underbrace{\left(\frac{1}{N} \boldsymbol{\Lambda} \mathcal{J} \mathcal{J}^\top + \gamma \boldsymbol{I}_{NK} \right)^{-1}}_{NK \times NK} \frac{\partial}{\partial f} \mathcal{L}(\boldsymbol{\theta}).\end{aligned}$$

This "NGD-by-NTK" transforms the time and space complexity of standard NGD – from depending on the number of DNN parameters P to depending on the number of input samples N and number of classes K – enabling a faster evaluation of NGD for a DNN with massive parameters. NGD-by-NTK is especially useful in situations where the number of data N per iteration is small (*e.g.*, mini-batch learning) and therefore can be practically evaluated. This idea has already adopted by some work for a practical second-order gradient computation [Ren and Goldfarb, 2019], for model compression [Singh and Alistarh, 2020], and for theoretical analysis of convergence of NGD in wide neural networks [Bernacchia et al., 2018, Zhang et al., 2019b, Cai et al., 2019, Karakida and Osawa, 2020].

4.5.1 Distributed NGD-by-NTK

Algorithm 2 describes how to realize the CG method for NGD-by-NTK using multiple processes (p processes) in a data-parallel fashion. As we will discuss in Chapter 5, distributed training using multiple accelerators (GPUs) speedup a DNN training by enabling processing a large number (N) of data in one update step. This approach has the potential to realize the exact natural gradients in large-scale deep learning using multiple accelerators. Due to the iterative nature of the CG method, it is unlikely that this approach will be used directly to speed up the DNN training, but it is expected to be used for the purpose of verifying the accuracy of existing natural gradient approximations (*e.g.*, Layer-wise NGD, K-FAC). Fast implementation of this method and the exact natural gradient analysis is the future work of this study.

Algorithm 2: Distributed CG for NGD-by-NTK

input: rank i, *global* mini-batch size N, *local* mini-batch $\mathcal{X}_i \in \mathbb{R}^{B \times M_0}$, $N = Bp$ $(i = 1, \ldots, p)$

forward propagation (pre-softmax) $\boldsymbol{f} = f(\mathcal{X}_i; \boldsymbol{\theta}) \in \mathbb{R}^{BK}$;

calculate the Hessian of cross-entropy loss $\boldsymbol{\Lambda}(\mathcal{X}_i; \boldsymbol{\theta}) = \frac{\partial^2}{\partial \boldsymbol{f} \partial \boldsymbol{f}^\top} \ell(\mathcal{X}_i; \boldsymbol{\theta}) \in \mathbb{R}^{BK \times BK}$;

$\boldsymbol{x}_i^{(0)} \leftarrow \boldsymbol{0} \in \mathbb{R}^{BK}$;

$\boldsymbol{p}_i^{(0)}, \boldsymbol{r}_i^{(0)} \leftarrow \frac{1}{N} \frac{\partial}{\partial \boldsymbol{f}} \ell(\mathcal{X}_i; \boldsymbol{\theta}) \in \mathbb{R}^{BK}$;

$r^{(0)} \leftarrow {\boldsymbol{r}_i^{(k)}}^\top \boldsymbol{r}_i^{(k)}$;

AllReduce$\big(n^{(0)}\big)$;

$k \leftarrow 0$;

repeat

  $\boldsymbol{g} \leftarrow \underbrace{\mathcal{J}(\mathcal{X}_i)^\top \boldsymbol{p}_i^{(k)}}_{backprop} \in \mathbb{R}^P$;

  Detach $\boldsymbol{g}$ from the computational graph;

  AllReduce($\boldsymbol{g}$);

  $\boldsymbol{u}_i \leftarrow \frac{1}{N} \boldsymbol{\Lambda}(\mathcal{X}_i, \boldsymbol{\theta}) \mathcal{J}(\mathcal{X}_i) \boldsymbol{g} = \frac{1}{N} \boldsymbol{\Lambda}(\mathcal{X}_i, \boldsymbol{\theta}) \underbrace{\frac{\partial {\boldsymbol{p}_i^{(k)}}^\top \mathcal{J}(\mathcal{X}_i) \boldsymbol{g}}{\partial \boldsymbol{p}_i^{(k)}}}_{forward-prop} \in \mathbb{R}^{BK}$;

  $\boldsymbol{u}_i \leftarrow \boldsymbol{u}_i + \gamma \boldsymbol{p}_i^{(k)}$; `// damping`

  $m \leftarrow {\boldsymbol{p}_i^{(k)}}^\top \boldsymbol{u}_i$;

  AllReduce(m);

  $\alpha \leftarrow r^{(k)} / m$;

  $\boldsymbol{x}_i^{(k+1)} \leftarrow \boldsymbol{x}_i^{(k)} + \alpha \boldsymbol{p}_i^{(k)}$;

  $\boldsymbol{r}_i^{(k+1)} \leftarrow \boldsymbol{r}_i^{(k)} - \alpha \boldsymbol{u}_i$;

  $r^{(k+1)} \leftarrow {\boldsymbol{r}_i^{(k+1)}}^\top \boldsymbol{r}_i^{(k+1)}$;

  AllReduce$\big(n^{(k+1)}\big)$;

  $\beta \leftarrow r^{(k+1)} / r^{(k)}$;

  $\boldsymbol{p}_i^{(k+1)} \leftarrow \boldsymbol{r}_i^{(k+1)} + \beta \boldsymbol{p}_i^{(k)}$;

  $k \leftarrow k + 1$;

until $r^{(k+1)}$ *is small enough*;

$\mathcal{G} \leftarrow \underbrace{\mathcal{J}(\mathcal{X}_i)^\top \boldsymbol{x}_i^{(k+1)}}_{backprop} \in \mathbb{R}^P$;

AllReduce($\mathcal{G}$);

return $\mathcal{G}$

Chapter 5

Scalable and practical natural gradient for large-scale deep learning

Large-scale distributed training of deep neural networks results in models with worse generalization performance as a result of the increase in the effective mini-batch size. Previous approaches attempt to address this problem by varying the learning rate and batch size over epochs and layers, or ad hoc modifications of batch normalization. We propose *Scalable and Practical Natural Gradient Descent* (SP-NGD), a principled approach for training models that allows them to attain similar generalization performance to models trained with first-order optimization methods, but with accelerated convergence. Furthermore, SP-NGD scales to large mini-batch sizes with a negligible computational overhead as compared to first-order methods. We evaluated SP-NGD on a benchmark task where highly optimized first-order methods are available as references: training a ResNet-50 model for image classification on ImageNet. We demonstrate convergence to a top-1 validation accuracy of 75.4% in 5.5 minutes using a mini-batch size of 32,768 with 1,024 GPUs, as well as an accuracy of 74.9% with an extremely large mini-batch size of 131,072 in 873 steps of SP-NGD.

5.1 Distributed deep learning

5.1.1 Mini-batch training

When the size of the training dataset $|\mathcal{S}|$ is large, it is a common strategy to use a *mini-batch* $\mathcal{B} \subset \mathcal{S}$ for estimating $\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})$:

$$\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) \approx \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) := \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{B}} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{\theta}) \,.$$

In a stochastic *mini-batch* gradient descent, we refresh the mini-batch $\mathcal{B}$ every step and update the parameters using the *mini-batch gradient* $\nabla \mathcal{L}_{\mathcal{B}}$:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \nabla \mathcal{L}_{\mathcal{B}^{(t)}}(\boldsymbol{\theta}^{(t)}) \,. \tag{5.1}$$

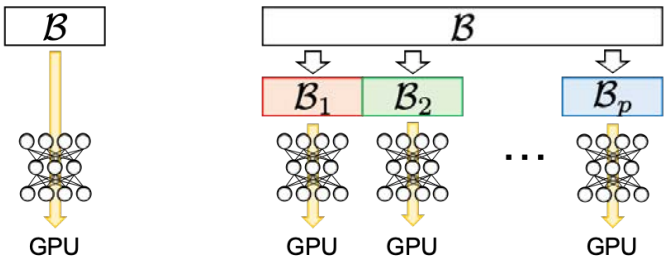


Figure 5.1: (Left) training w/ a single GPU. (Right) distributed data parallel training w/ multiple (p) GPUs.

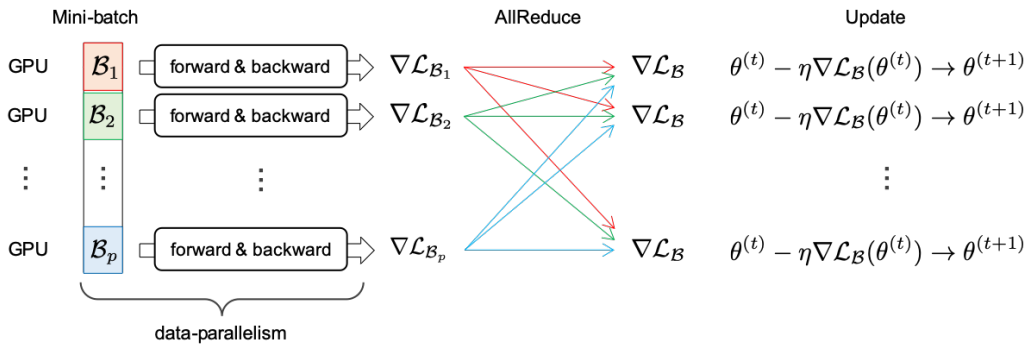


Figure 5.2: Distributed mini-batch stochastic gradient descent.

With this strategy, we can easily increase the number of updates t and speed up the training. The largest mini-batch size $|\mathcal{B}|$ we can use is often limited by the memory capacity of an accelerator (*e.g.*, GPU), and the optimal mini-batch size is determined based on the throughput (the number of examples processed per specific time). For instance, $|\mathcal{B}| = 32$ (32 images per step) is often chosen for training a deep neural network on the ImageNet-1K [Deng et al., 2009] with a single NVIDIA GPU (*e.g.*, Tesla P100/V100).

It is also a common strategy to choose mini-batches so that the model looks at all examples in the dataset $\mathcal{S}$ equally:

$$\mathcal{S} = \mathcal{B}^{(t)} \cup \mathcal{B}^{(t+1)} \cup \cdots \cup \mathcal{B}^{(t+|\mathcal{S}|/|\mathcal{B}|-1)} .$$

Hence, it takes $|\mathcal{S}|/|\mathcal{B}|$ steps to process all the examples once (*i.e.*, *an epoch*). Processing an epoch with a mini-batch size $|\mathcal{B}| = 32$ takes about 40K steps for the ImageNet-1K (about 1.28M examples).

5.1.2 Distributed data parallel training

Now we consider to utilize multiple accelerators (*e.g.*, GPUs) to speedup the mini-batch training. A simple and common strategy is to divide the mini-batch into multiple subsets and let the different processes handle the different subsets (*distributed data parallel* training, Figure 5.1):

$$\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2 \cup \cdots \cup \mathcal{B}_p ,$$

where p is the number of processes (accelerators), and $|\mathcal{B}_i| = |\mathcal{B}|/p$ $(i = 1, \ldots, p)$. We can get the *global* mini-batch gradient $\nabla\mathcal{L}_{\mathcal{B}}$ as the average of the *local* mini-batch gradients:

$$\nabla\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{p}\left(\nabla\mathcal{L}_{\mathcal{B}_1}(\boldsymbol{\theta}) + \nabla\mathcal{L}_{\mathcal{B}_2}(\boldsymbol{\theta}) + \cdots + \nabla\mathcal{L}_{\mathcal{B}_p}(\boldsymbol{\theta})\right)$$

Therefore, the p processes can calculate the local mini-batch gradients $\nabla\mathcal{L}_{\mathcal{S}_i}$ independently, and, as depicted in Figure 5.1, we can process p-times larger number of examples in a step if we have p accelerators. Note that all the processes need to have the same model. To ensure this, the initial parameters $\boldsymbol{\theta}^{(0)}$ is synchronized among all the process at the beginning of the training. Also, the global mini-batch $\nabla\mathcal{L}_{\mathcal{B}}$ needs to be synchronized every time we apply the update rule (Equation 5.1). This can be realized by calling `AllReduce` collective (illustrated in Figure 5.6). The distributed fashion of stochastic mini-batch gradient descent is illustrated in Figure 5.2.

5.1.3 Large-batch training

As the size of deep neural network models and the data which they are trained on continues to increase rapidly, the demand for distributed parallel computing is increasing. A common approach for achieving distributed parallelism in deep learning is to use the data-parallel approach, where the data is distributed across different processes while the model is replicated across them. When the mini-batch size per process is kept constant to increase the ratio of computation over communication, the effective mini-batch size over the entire system grows proportional to the number of processes.

Keskar et al. [2017] report that when the mini-batch size is "large", generalization performance is worse than when a smaller mini-batch is used. Hoffer et al. [2018] attribute this generalization gap to the limited number of updates, and suggest to train longer. Goyal et al. [2017] adopt strategies such as scaling the learning rate proportional to the mini-batch size, while using the first few epochs to gradually warmup the learning rate. Such methods have enabled the training for mini-batch sizes of 8K, where ImageNet [Deng et al., 2009] with ResNet-50 [He et al., 2016] could be trained for 90 epochs with little reduction in generalization performance (76.3% top-1 validation accuracy) in 60 minutes. They also report that when the mini-batch size is increased beyond a certain point (>8K,) the generalization performance starts to degrade even with their strategies. Shallue et al. [2019], with thorough hyper-parameter tuning for various models and datasets, empirically show that there is no evidence that larger mini-batch sizes degrade the generalization performance. However, for training ResNet-50 on ImageNet, their results agree that larger mini-batch sizes (>8K) degrade the generalization performance without the use of label smoothing [Szegedy et al., 2015].

Since Goyal et al. [2017]'s results were presented, there have been many attempts to train ResNet-50 on ImageNet with mini-batch sizes larger than 16K. Combining the learning rate scaling with other techniques such as RMSprop [Tieleman and Hinton, 2012] warm-up, BatchNorm [Ioffe and Szegedy, 2015] without moving averages, and a slow-start learning rate schedule, Akiba et al. [2017]

Table 5.1: Training time and top-1 single-crop validation accuracy of ResNet-50 for ImageNet reported by related work and this work.

	Hardware	Software	Batch size	Optimizer	# Steps	Time/step	Time	Accuracy
Goyal et al. [2017]	Tesla P100 × 256	Caffe2	8,192	SGD	14,076	255 ms	1 hr	**76.3** %
You et al. [2017b]	KNL × 2048	Intel Caffe	32,768	SGD	3,519	341 ms	20 min	75.4 %
Akiba et al. [2017]	Tesla P100 × 1024	Chainer	32,768	RMSprop/SGD	3,519	255 ms	15 min	74.9 %
You et al. [2017b]	KNL × 2048	Intel Caffe	32,768	SGD	2,503	335 ms	14 min	74.9 %
Jia et al. [2018]	Tesla P40 × 2048	TensorFlow	65,536	SGD	1,800	220 ms	6.6 min	75.8 %
Ying et al. [2018]	TPU v3 × 1024	TensorFlow	32,768	SGD	3,519	**37 ms**	2.2 min	**76.3** %
Mikami et al. [2018]	Tesla V100 × 3456	NNL	55,296	SGD	2,086	57 ms	2.0 min	75.3 %
Yamazaki et al. [2019]	Tesla V100 × 2048	MXNet	81,920	SGD	1,440	50 ms	**1.2 min**	75.1 %
This work	Tesla V100 × 128	Chainer	4,096	SP-NGD	10,948	178 ms	32.5 min	74.8 %
	Tesla V100 × 256		8,192		5,434	186 ms	16.9 min	75.3 %
	Tesla V100 × 512		16,384		2,737	149 ms	6.8 min	75.2 %
	Tesla V100 × 1024		32,768		1,760	187 ms	**5.5 min**	75.4 %
	n/a		65,536		1,173	n/a	n/a	75.6 %
	n/a		**131,072**		**873**	n/a	n/a	74.9 %

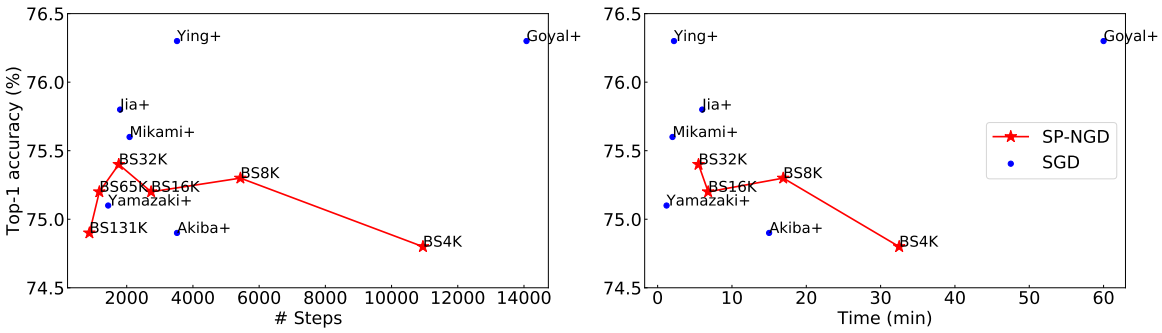


Figure 5.3: Top-1 validation accuracy vs the number of steps to converge (left) and vs training time (right) of ResNet-50 on ImageNet (1000 class) classification by related work with SGD and this work with *Scalable and Practical NGD* (SP-NGD).

were able to train the same dataset and model with a mini-batch size of 32K to achieve 74.9% accuracy in 15 minutes.

More complex approaches for manipulating the learning rate were proposed, such as LARS [You et al., 2017a], where a different learning rate is used for each layer by normalizing them with the ratio between the layer-wise norms of the weights and gradients. This enabled the training with a mini-batch size of 32K without the use of *ad hoc* modifications, which achieved 74.9% accuracy in 14 minutes (64 epochs) [You et al., 2017b]. It has been reported that combining LARS with counter intuitive modifications to the batch normalization, can yield 75.8% accuracy even for a mini-batch size of 65K [Jia et al., 2018].

The use of small batch sizes to encourage rapid convergence in early epochs, and then progressively increasing the batch size is yet another successful approach. Using such an adaptive batch size method, Mikami et al. [2018] were able to train in 122 seconds with an accuracy of 75.3%, and Yamazaki et al. [2019] were able to train in 75 seconds with a accuracy of 75.1%. The hierarchical synchronization of mini-batches have also been proposed [Lin et al., 2020], but such methods have not been tested at scale to the extent of the authors' knowledge.

5.2 Natural gradient descent for large-batch training

While first-order stochastic gradient descent (SGD) has been the dominant approach for training deep neural networks, we explore the potential for second-order optimization methods such as Natural Gradient Descent (NGD) [Amari, 1998] that leverage curvature information to accelerate optimization in large mini-batch settings. We found that NGD enables training with a fewer number of steps than what was previously thought to be possible with SGD, while retaining competitive generalization performance with the help of stronger data augmentation i.e. *mixup* [Zhang et al., 2018c]. Note that this kind of data augmentation helps NGD much more than it does SGD, so we are not simply increasing the baseline accuracy. We minimize the extra per-step computational overhead associated with inverting the curvature matrices using an efficient distributed layer-wise NGD design that scales to massively parallel settings and large mini- batch sizes. In particular, we demonstrate scalability to batch sizes of 32,768 over 1024 GPUs across 256 nodes.

We adopt the Kronecker-Factored Approximate Curvature (K-FAC) method [Martens and Grosse, 2015] for its ability to efficiently approximate the curvature, in contrast to iterative approaches like Hessian-free optimization [Martens, 2010], even though these may yield more accurate curvature approximation. The two main characteristics of K-FAC are that it converges faster than first-order stochastic gradient descent (SGD) methods, and that it can tolerate relatively large mini-batch sizes without any *ad hoc* modifications. K-FAC has been successfully applied to convolutional neural networks [Grosse and Martens, 2016], distributed memory training of ImageNet [Ba et al., 2017], recurrent neural networks [Martens and Johnson, 2018], Bayesian deep learning [Zhang et al., 2018a], reinforcement learning [Wu and Ma, 2018] , and Transformer models [Zhang et al., 2019a].

Our contributions are:

- **Extremely large mini-batch training.** We were able to show for the first time that approximated NGD can achieve similar generalization capability compared to highly optimized SGD, by training ResNet-50 on ImageNet classification as a benchmark. We converged to over 75% top-1 validation accuracy for large mini-batch sizes of 4,096, 8,192, 16,384, 32,768 and 65,536. We also achieved 74.9% with an extremely large mini-batch size of 131,072, which took only 873 steps.

- **Scalable natural gradient.** We propose a distributed NGD design using data and model hybrid parallelism that shows *superlinear* scaling up to 64 GPUs.

- **Practical natural gradient.** We propose practical NGD techniques based on analysis of the FIM estimation in large mini-batch settings. Our practical techniques make the overhead of NGD compare to SGD almost negligible. Combining these techniques with our distributed NGD, we see an ideal scaling up to 1024 GPUs as shown in Figure 5.7.

- **Training ResNet-50 on ImageNet in 5.5 minutes.** Using 1024 NVIDIA Tesla V100, we achieve 75.4 % top-1 accuracy with ResNet-50 for ImageNet in 5.5 minutes (1760 steps = 45 epochs, including a validation after each epoch). The comparison is shown in Figure 5.3 and Table 5.1.

5.3 Related work

With respect to large-scale distributed training of deep neural networks, there have been very few studies that use second-order methods. At a smaller scale, there have been previous studies that used K-FAC to train ResNet-50 on ImageNet [Ba et al., 2017]. However, the SGD they used as reference was not showing state-of-the-art Top-1 validation accuracy (only around 70%), so the advantage of K-FAC over SGD that they claim was not obvious from the results. In the present work, we compare the Top-1 validation accuracy with state-of-the-art SGD methods for large mini-batches mentioned in the introduction (Table 5.1).

The previous studies that used K-FAC to train ResNet-50 on ImageNet Ba et al. [2017] also were not considering large mini-batches and were only training with mini-batch size of 512 on 8 GPUs. In contrast, the present work uses mini-batch sizes up to 131,072, which is equivalent to 32 per GPU on 4096 GPUs, and we are able to achieve a much higher Top-1 validation accuracy of 74.9%. Note that such large mini-batch sizes can also be achieved by accumulating the gradient over multiple iterations before updating the parameters, which can mimic the behavior of the execution on many GPUs without actually running them on many GPUs.

The previous studies using K-FAC also suffered from large overhead of the communication since they used a parameter-server approach for their TensorFlow Abadi et al. [2016] implementation of K-FAC with a single parameter-server[1]. Since the parameter server requires all workers to send the gradients and receive the latest model's parameters from the parameter server, the parameter server becomes a huge communication bottleneck especially at large scale. Our implementation uses a decentralized approach using MPI/NCCL[2] collective communications among the processes. The decentralized approach has been used in high performance computing for a long time, and is known to scale to thousands of GPUs without modification. Although, software like Horovod[3] can alleviate the problems with parameter servers by working as a TensorFlow wrapper for NCCL, a workable realization of K-FAC requires solving many engineering and modeling challenges, and our solution is the first one that succeeds on a large scale task.

[1] The current version of the TensorFlow K-FAC implementation (`https://github.com/tensorflow/kfac`) supports other methods for distributing computations, including a decentralized approach for TPUs.

[2] https://developer.nvidia.com/nccl

[3] https://github.com/horovod/horovod

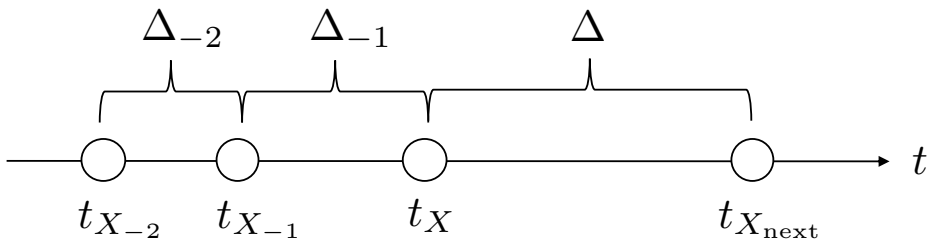


Figure 5.4: Estimate the interval (steps) Δ until the next step $t_{X_{\text{next}}}$ to refresh the statistics X.

5.4 Practical natural gradient

5.4.1 Fast Estimation with Empirical Fisher

Instead of using an estimation by a single Monte Carlo sampling ($\boldsymbol{F}_{1\text{mc}l}$), we adopt the *empirical Fisher* [Martens and Grosse, 2015] to estimate the layer-wise FIM $\boldsymbol{F}_l$:

$$\boldsymbol{F}_l \approx \boldsymbol{C}_l := \left\langle \nabla_{\boldsymbol{\theta}_l} \log p_{\boldsymbol{\theta}}(y|\boldsymbol{x}) \nabla_{\boldsymbol{\theta}_l} \log p_{\boldsymbol{\theta}}(y|\boldsymbol{x})^\top \right\rangle . \tag{5.2}$$

We implemented an efficient $\boldsymbol{C}_l$ computation on top of the Chainer framework [Tokui et al., 2019] that allows us to compute $\boldsymbol{C}_l$ during the forward-pass and the backward-pass for the loss $\mathcal{L}(\boldsymbol{\theta})$[4] Therefore, we *do not need an extra backward-pass to compute* $\boldsymbol{C}_l$, while *an extra backward-pass is necessary for computing* $\boldsymbol{F}_{1\text{mc}l}$. This difference is critical especially for a deeper network which takes longer time for a backward-pass. Although it is insisted that $\boldsymbol{C}$ is not a proper approximation of the FIM, and $\boldsymbol{F}_{1\text{mc}}$ is better estimation in the literature Thomas et al. [2019], Kunstner et al. [2019], we do not see any difference in the convergence behavior nor the final model accuracy between NGD with $\boldsymbol{C}$ and that with $\boldsymbol{F}_{1\text{mc}}$ in training deep ConvNets for ImageNet classification as shown in Section 5.7. This could be because $\boldsymbol{C}$ is effectively being used as a pre-conditioning matrix to accelerate optimization, which only requires it to capture aspects of the FIM; in particular it also leads to parameterization invariant updates as discussed in [Martens, 2020].

5.4.2 Practical FIM Estimation for BatchNorm Layers

It is often the case that a deep ConvNet has Conv layers that are followed by BatchNorm layers [Ioffe and Szegedy, 2015]. The l-th BatchNorm layer after the $(l-1)$-th Conv layer has scale $\boldsymbol{\gamma}_l \in \mathbb{R}^{C_{l-1}}$ and bias $\boldsymbol{\beta}_l \in \mathbb{R}^{C_{l-1}}$ to be applied to the normalized features. When we see these parameters as learnable parameters, we can define

$$\boldsymbol{\theta}_l := \begin{pmatrix} \gamma_{l,1} & \beta_{l,1} & \cdots & \gamma_{l,C_{l-1}} & \beta_{l,C_{l-1}} \end{pmatrix}^\top \in \mathbb{R}^{2C_{l-1}} , \tag{5.3}$$

where $\gamma_{l,i}$ and $\beta_{l,i}$ are the i-th element of $\boldsymbol{\gamma}_l$ and $\boldsymbol{\beta}_l$, respectively $(i = 1, \ldots, C_{l-1})$. For this BatchNorm layer, the FIM $\boldsymbol{F}_l$ is a $2C_{l-1} \times 2C_{l-1}$ matrix, and the computation cost of inverting this

[4]The same empirical Fisher computation can be implemented on PyTorch [Paszke et al., 2019]

matrix can not be ignored when C_{l-1} (the number of output channels of the previous Conv layer) is large (e.g. $C_{\text{out}} = 1024$ for a Conv layer in ResNet-50 [He et al., 2016]).

5.4.2.1 Unit-wise Natural Gradient

We approximate this FIM by applying unit-wise natural gradient [Amari, 1998] to the learnable parameters of BatchNorm layers. A "unit" in a neural network is a collection of input/output nodes connected to each other. The "unit-wise" natural gradient only takes into account the correlation of the parameters in the same node. Hence, for a BatchNorm layer, we only consider the correlation between $\gamma_{l,c}$ and $\beta_{l,c}$ of the same channel c:

$$\begin{aligned} \boldsymbol{F}_l \approx \tilde{\boldsymbol{F}}_l &= \boldsymbol{F}_{l,\text{unit BN}} \\ &:= \text{diag}\left(\boldsymbol{F}_l^{(1)} \dots \boldsymbol{F}_l^{(i)} \dots \boldsymbol{F}_l^{(C_{l-1})}\right) \in \mathbb{R}^{2C_{l-1} \times 2C_{l-1}}, \end{aligned} \tag{5.4}$$

where

$$\boldsymbol{F}_l^{(i)} = \left\langle \mathbb{E}\left[\begin{array}{cc} {\nabla_{\boldsymbol{\gamma}_l}^{(i)}}^2 & \nabla_{\boldsymbol{\gamma}_l}^{(i)} \nabla_{\boldsymbol{\beta}_l}^{(i)} \\ \nabla_{\boldsymbol{\beta}_l}^{(i)} \nabla_{\boldsymbol{\gamma}_l}^{(i)} & {\nabla_{\boldsymbol{\beta}_l}^{(i)}}^2 \end{array} \right] \right\rangle \in \mathbb{R}^{2\times 2}. \tag{5.5}$$

$\nabla_{\boldsymbol{\gamma}_l}^{(i)}$, $\nabla_{\boldsymbol{\beta}_l}^{(i)}$ are the i-th element of $\nabla_{\boldsymbol{\gamma}_l} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$, $\nabla_{\boldsymbol{\beta}_l} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$, respectively. The number of the elements to be computed and communicated is significantly reduced from $4c_{l-1}^2$ to $4c_{l-1}$, and we can get the inverse $(\boldsymbol{F}_{l,\text{unit BN}} + \lambda \boldsymbol{I})^{-1}$ with little computation cost using the inverse matrix formula:

$$\left[\begin{array}{cc} a & b \\ c & d \end{array} \right]^{-1} = \frac{1}{ad - bc} \left[\begin{array}{cc} d & -b \\ -c & a \end{array} \right]. \tag{5.6}$$

We observed that the unit-wise approximation on $\boldsymbol{F}_l$ of BatchNorm does not affect the accuracy of a deep ConvNet on ImageNet classification as shown in Section 5.7.

5.4.3 Natural Gradient with Stale Statistics

To achieve even faster training with NGD, it is critical to utilize stale statistics in order to avoid re-computing the matrices $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_{l,\text{unitBN}}$ at every step. Previous work on K-FAC [Martens and Grosse, 2015] used a simple strategy where they refresh the Kronecker factors only once in 20 steps. However, as observed in [Osawa et al., 2019b], the statistics rapidly fluctuate at the beginning of training, and this simple strategy causes serious defects to the convergence. It was also observed that the degree of fluctuation of the statistics depends on the mini-batch size, the layer, and the type of the statistics (*e.g.*, statistics with a larger mini-batch fluctuates less than that with a smaller mini-batch). Although the previous strategy [Osawa et al., 2019b] to reduce the frequency worked without any degradation of the accuracy, it requires the prior observation on the fluctuation of the statistics, and its effectiveness on training time has not been well studied.

Algorithm 3: Natural gradient with the stale statistics.

input : set of the statistics $\mathcal{S}$ (damped inverses)
input : initial parameters $\boldsymbol{\theta}$
output: parameters $\boldsymbol{\theta}$

$t \leftarrow 1$
foreach $X \in \mathcal{S}$ **do**
  $t_X \leftarrow 1$
end
while *not converge* **do**
  foreach $X \in \mathcal{S}$ **do**
    if $t == t_X$ **then**
      refresh statistics X
      $\Delta, \Delta_{-1} \leftarrow$ Algorithm 4$(X, X_{-1}, X_{-2}, \Delta, \Delta_{-1})$
      $t_X \leftarrow t_X + \Delta$
      $X_{-1}, X_{-2} \leftarrow X, X_{-1}$
  end
  update $\boldsymbol{\theta}$ by natural gradient (3.6) using $\mathcal{S}$
  $t \leftarrow t + 1$
end
return $\boldsymbol{\theta}$

5.4.3.1 Adaptive Frequency To Refresh Statistics

We propose an improved strategy which adaptively determines the frequency to refresh each statistics based on its staleness during the training. Our strategy is shown in Algorithm 3. We calculate the timing (step) to refresh each statistics based on the *acceptable* interval (steps) estimated in Algorithm 4 (Figure 5.4). In Algorithm 4, matrix A is considered to be *similar to* matrix B when $\|A - B\|_F / \|B\|_F < \alpha$, where $\| \cdot \|_F$ is Frobenius norm, and $\alpha > 0$ is the threshold[5]. We tuned α by running training for a few epochs to check if the threshold is not too large (preserves the same convergence). And it aims to find the *acceptable* interval Δ where the statistics X calculated at step $t = t_X + \Delta$ is *similar to* that calculated at step $t = t_X$. With this strategy, we can estimate the *acceptable* interval for each statistics and skip the computation efficiently — it keeps almost the same training time as the original, while reducing the cost for constructing and inverting $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_{l,\mathrm{unitBN}}$ as much as possible. Note that $\mathcal{S}$ in Algorithm 1 is the set of the inverses of damped matrices $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_{l,\mathrm{unitBN}}$. That is, we observe the staleness of the inverse matrices. This significantly reduces the overhead of NGD. We observe the effectiveness of our approach in Section 5.7[6].

Algorithm 4: Estimate the *acceptable* interval until next refresh based on the staleness of statistics.

input : statistics X, X_{-1} (last), X_{-2} (before the last)
input : last interval Δ_{-1}, interval before the last Δ_{-2}
output: next interval Δ, last interval Δ_{-1}

if X *is not similar to* X_{-1} **then**
  $\Delta \leftarrow \max\{1, \lfloor \Delta_{-1}/2 \rfloor\}$
else if X *is not similar to* X_{-2} **then**
  $\Delta \leftarrow \Delta_{-1}$
else
  $\Delta \leftarrow \Delta_{-1} + \Delta_{-2}$
end
return Δ, Δ_{-1}

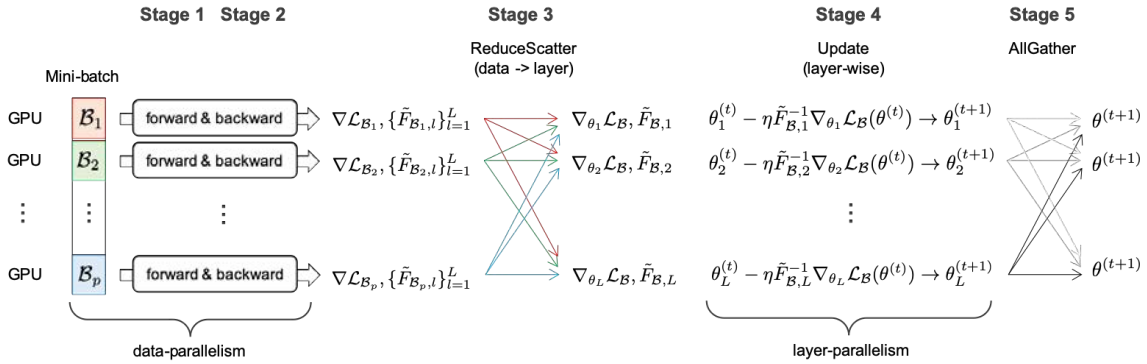


Figure 5.5: Distributed layer-wise approximate natural gradient.

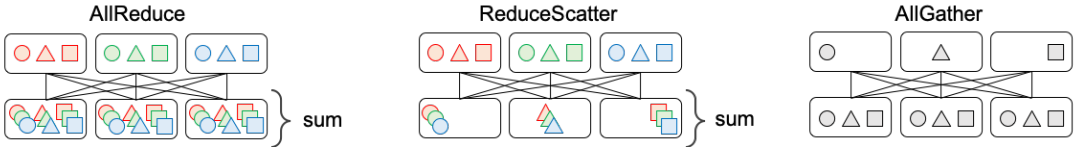


Figure 5.6: Illustrations of `AllReduce`, `ReduceScatter(V)`, and `AllGather(V)` collective. Different colors correspond to data from different batches $\mathcal{B}_i$.

5.5 Scalable natural gradient

Based on the practical estimation of the FIM proposed in the previous section, we designed a distributed parallelization scheme among multiple GPUs so that the overhead of NGD compare to SGD decreases as the number of GPUs (processes) is increased.

Algorithm 5: Distributed Natural Gradient

while *not converge* **do**
  `// Stage 1`
  foreach $l = 1, \cdots, L$ **do**
    forward in l-th layer
    if l-th layer is Conv or FC **then**
      compute $\boldsymbol{A}_{l-1}$
  end
  `// Stage 2`
  ReduceScatterV($\boldsymbol{A}_{0:L-1}$)
  foreach $l = L, \cdots, 1$ **do**
    backward in l-th layer
    if l-th layer is Conv or FC **then**
      compute $\boldsymbol{B}_l$
    else if l-th layer is BatchNorm **then**
      compute $\boldsymbol{F}_{l,\text{unitBN}}$
  end
  `// Stage 3`
  ReduceScatterV($\boldsymbol{B}_{1:L}/\boldsymbol{F}_{1:L,\text{unitBN}}$ and $\nabla_{\boldsymbol{\theta}_{1:L}}\mathcal{L}$)
  `// Stage 4`
  for $l = 1, \cdots, L$ **do in parallel**
    update $\boldsymbol{\theta}_l$ by natural gradient (3.6)
  end
  `// Stage 5`
  AllGatherV($\boldsymbol{\theta}_{1:L}$)
end
return $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_1^\top & \cdots & \boldsymbol{\theta}_L^\top \end{bmatrix}^\top$

5.5.1 Distributed Natural Gradient

Figure 5.5 shows the overview of our design, which shows a single step of training with our distributed NGD. We use the term `Stage` to refer to each phase of computation and communication, which is indicated at the top of the figure. Algorithm 5 shows the pseudo code of our distributed NGD design.

In `Stage 1`, each process (GPU) receives a different part of the mini-batch and performs a forward-pass in which the Kronecker factor $\boldsymbol{A}_{l-1}$ is computed for the received samples, if l-th layer is a Conv layer or a FC layer.

In `Stage 2`, two procedures are performed in parallel — communication among all the processes and a backward-pass in each process. Since `Stage 1` is done in a **data-parallel** fashion, each process computes the statistics only for the different parts of the mini-batch. In order to compute these statistics for the entire mini-batch, we need to average these statistics over all the processes. This

[5] $\alpha = 0.1$ for all the experiments in this work.

[6] For each mini-batch size, we do not use the decayed average of the FIM (Kronecker factors) as opposed to the previous K-FAC papers [Martens and Grosse, 2015, Grosse and Martens, 2016, Ba et al., 2017]. When training with extremely large batches, we observed that the decayed average has little effect.

is performed using a `ReduceScatterV` collective communication, which transitions our approach from data-parallelism to model-parallelism by reducing (taking the sum of) $\boldsymbol{A}_{l-1}$ for different l to different processes. This collective is much more efficient than an `AllReduce` , where $\boldsymbol{A}_{l-1}$ for all l are reduced to all the processes (Figure 5.5). While $\boldsymbol{A}_{l-1}$ is communicated, each process also performs a backward-pass to get the gradient $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$, the Kronecker factor $\boldsymbol{B}_l$ for Conv, FC layers, and $\boldsymbol{F}_{l,\text{unitBN}}$ for BatchNorm layer, for each l.

In `Stage 3`, $\boldsymbol{B}_l$,$\boldsymbol{F}_{l,\text{unitBN}}$ and $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$ are communicated in the same way as $\boldsymbol{A}_l$ by `ReduceScatterV` collective. At this point, only a single process has the FIM estimation $\hat{\boldsymbol{F}}_l$ and the gradient $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$ with the statistics for the entire mini-batch for the l-th layer. In `Stage 4`, only the process that has the FIM computes the matrix inverse and applies the NGD update (3.6) to the weights $\boldsymbol{\theta}_l$ of the l-th layer. Hence, these computations are performed in a **model-parallel** fashion. When the number of layers is larger than the number of processes, multiple layers are handled by a process.

Once the weights $\boldsymbol{\theta}_l$ of each l are updated, we synchronize the updated weights among all the processes by calling an `AllGatherV` (Figure 5.5) collective, and we switch back to data-parallelism. Combining the practical estimation of the FIM proposed in the previous section, we are able to reduce a significant amount of communication required for the Kronecker factors $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_{l,\text{unitBN}}$. Therefore, the amount of communication for our distirbuted NGD is similar to distributed SGD, where the `AllReduce` for the gradient $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$ is implemented as a `ReduceScatter+AllGather`.

5.5.2 Further acceleration

Our data-parallel and model-parallel hybrid approach allows us to minimize the overhead of NGD in a distributed setting. However, NGD still has a large overhead compared to SGD. There are two hotspots in our distributed NGD design. The first is the construction of the statistics $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$, and $\boldsymbol{F}_{l,\text{unitBN}}$, that cannot be done in a model-parallel fashion. The second is the communication (`ReduceScatterV`) for distributing these statistics. In this section, we discuss how we accelerate these two hotspots to achieve even faster training time.

Mixed-precision computation. We use the Tensor Cores in the NVIDIA Volta Architecture [7]. This more than doubles the speed of the calculation for this part. One might think that this low-precision computation affects the overall accuracy of the training, but in our experiments we do not find any differences between training with half-precision floating point computation and that with full-precision floating point computation.

Symmetry-aware communication. The statistics matrices $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$, and $\boldsymbol{F}_{l,\text{unitBN}}$ are symmetric matrices. We exploit this property to reduce the amount of communication without loss of information. To communicate a symmetric matrix of size $N \times N$, we only need to send the upper triangular matrix with $N(N+1)/2$ elements.

[7] https://www.nvidia.com/en-us/data-center/tensorcore/

In addition to these two optimizations, we also adopted the performance optimizations done by [Tsuji et al., 2019]:

- Explicitly use NHWC (mini-batch, height, width, and channels) format for the input/output data (tensor) of Conv layers instead of NCHW format. This makes cuDNN API to fully benefit from the Tensor Cores.
- Data I/O pipeline using the NVIDIA Data Loading Library (DALI) [8].
- Hierarchical `AllReduce` collective proposed by Ueno and Yokota [2019], which alleviates the latency of the `ring-AllReduce` communication among a large number of GPUs.
- Half-precision communication for `AllGatherV` collective.

5.6 Training for ImageNet Classification

The behavior of NGD on large models and datasets has not been studied in depth. Also, there are very few studies that use NGD (K-FAC) for large mini-batches (over 4K) using distributed parallelism at scale [Ba et al., 2017]. Contrary to SGD, where the hyperparameters have been optimized by many practitioners even for large mini-batches, there is very little insight on how to tune hyperparameters for NGD. In this section, we have explored some methods, which we call training schemes, to achieve higher accuracy in our experiments. In this section, we show those training schemes in our large mini-batch training with NGD for ImageNet classification.

5.6.1 Data augmentation

To achieve good generalization performance while keeping the benefit of the fast convergence that comes from NGD, we adopt the data augmentation techniques commonly used for training networks with large mini-batch sizes. We resize all the images in ImageNet to 256×256 ignoring the aspect ratio of original images and compute the mean value of the upper left portion (224×224) of the resized images. When reading an image, we randomly crop a 224×224 image from it, randomly flip it horizontally, subtract the mean value, and scale every pixel to $[0, 1]$.

Running mixup. We extend *mixup* [Zhang et al., 2018c] to increase its regularization effect. We synthesize virtual training samples from raw samples and virtual samples from the previous step (while the original *mixup* method synthesizes new samples only from the raw samples):

$$\tilde{\boldsymbol{x}}^{(t)} = \lambda \cdot \boldsymbol{x}^{(t)} + (1 - \lambda) \cdot \tilde{\boldsymbol{x}}^{(t-1)}\,, \tag{5.7}$$

$$\tilde{\boldsymbol{y}}^{(t)} = \lambda \cdot \boldsymbol{y}^{(t)} + (1 - \lambda) \cdot \tilde{\boldsymbol{y}}^{(t-1)}\,. \tag{5.8}$$

[8] https://developer.nvidia.com/DALI

$\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}$ is a raw input and label (one-hot vector), and $\tilde{\boldsymbol{x}}^{(t)}, \tilde{\boldsymbol{y}}^{(t)}$ is a virtual input and label for t th step. λ is sampled from the Beta distribution with the beta function

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,\mathrm{d}t\,. \tag{5.9}$$

where we set $\alpha = \beta = \alpha_{\text{mixup}}$.

Random erasing with zero value. We also implemented *Random Erasing* [Zhong et al., 2017]. We set elements within the erasing region of each input to zero instead of a random value as used in the original method. We set the erasing probability $p = 0.5$, the erasing area ratio $S_e \in [0.02, 0.25]$, and the erasing aspect ratio $r_e \in [0.3, 1]$. We randomly switch the size of the erasing area from (H_e, W_e) to (W_e, H_e).

5.6.2 Learning rate and momentum

The learning rate used for all of our experiments is scheduled by *polynomial decay*. The learning rate $\eta^{(e)}$ for e th epoch is determined as follows:

$$\eta^{(e)} = \eta^{(0)} \cdot \left(1 - \frac{e - e_{\text{start}}}{e_{\text{end}} - e_{\text{start}}}\right)^{p_{\text{decay}}}. \tag{5.10}$$

$\eta^{(0)}$ is the initial learning rate and $e_{\text{start}}, e_{\text{end}}$ is the epoch when the decay starts and ends. The decay rate p_{decay} guides the speed of the learning rate decay.

We use the momentum variant [Goyal et al., 2017] for NGD updates. Because the learning rate decays rapidly in the final stage of the training with the polynomial decay, the current update can become smaller than the previous update. We adjust the momentum rate $m^{(e)}$ for e th epoch so that the ratio between $m^{(e)}$ and $\eta^{(e)}$ is fixed throughout the training:

$$m^{(e)} = \frac{m^{(0)}}{\eta^{(0)}} \cdot \eta^{(e)}\,, \tag{5.11}$$

where $m^{(0)}$ is the initial momentum rate. The weights are updated as follows:

$$\boldsymbol{\theta}_l^{(t+1)} \leftarrow \boldsymbol{\theta}_l^{(t)} - \eta^{(e)}\left(\tilde{\boldsymbol{F}}_l^{(t)} + \lambda \boldsymbol{I}\right)^{-1} \nabla_{\boldsymbol{\theta}_l} \mathcal{L}^{(t)} + m^{(e)} \boldsymbol{v}^{(t)}\,, \tag{5.12}$$

where $\boldsymbol{v}^{(t)} = \boldsymbol{\theta}_l^{(t)} - \boldsymbol{\theta}_l^{(t-1)}$.

5.6.3 Weights rescaling

We found that the choice of learning rate is very sensitive to the ratio of the (approximate) natural gradient norm to the weight norm. Although it is argued that weight decay regularization helps to control the effective damping value for NGD [Zhang et al., 2018b], tuning the learning rate with weight decay is difficult because it *indirectly* controls the weight norm. To alleviate this difficulty, we

Table 5.2: The hyperparameters of the training with large mini-batch size (BS) used for our schemes in Section 5.6 and top-1 single-crop validation accuracy of ResNet-50 for ImageNet. `reduction` and `speedup` correspond to the reduction rate of the communication amount and the speedup comes from that, respectively, for `emp+unitBN+stale` compare to `emp+unitBN` in Figure 5.7.

BS	α_{mixup}	p_{decay}	e_{start}	e_{end}	$\eta^{(0)}$	$m^{(0)}$	λ	# steps	top-1 accuracy	reduction ↓	speedup ↑
4K	0.4	11.0	1	53	$8.18 \cdot 10^{-3}$	0.997	$2.5 \cdot 10^{-4}$	10,948	75.2 % → 74.8 %	23.6 %	× 1.33
8K	0.4	8.0	1	53.5	$1.25 \cdot 10^{-2}$	0.993	$2.5 \cdot 10^{-4}$	5,434	75.5 % → 75.3 %	15.1 %	× 1.32
16K	0.4	8.0	1	53.5	$2.5 \cdot 10^{-2}$	0.985	$2.5 \cdot 10^{-4}$	2,737	75.3 % → 75.2 %	**5.4** %	× **1.68**
32K	0.6	3.5	1.5	49.5	$3.0 \cdot 10^{-2}$	0.97	$2.0 \cdot 10^{-4}$	1,760	75.6 % → 75.4 %	7.8 %	× 1.40
65K	0.6	2.9	2	64.5	$4.0 \cdot 10^{-2}$	0.95	$1.5 \cdot 10^{-4}$	1,173	75.6 %	n/a	n/a
131K	1.0	2.9	3	100	$7.0 \cdot 10^{-2}$	0.93	$1.0 \cdot 10^{-4}$	873	74.9 %	n/a	n/a

adopt the *Normalizing Weights* [van Laarhoven, 2017] technique, which *directly* controls the weight norm, to the $\boldsymbol{\theta}_l$ of FC and Conv layers. We rescale the $\boldsymbol{\theta}_l$ to have a norm $\sqrt{2 \cdot C_{\text{out}}}$ after (5.12):

$$\boldsymbol{\theta}_l^{(t+1)} \leftarrow \sqrt{2 \cdot C_{\text{out}}} \cdot \frac{\boldsymbol{\theta}_l^{(t+1)}}{\|\boldsymbol{\theta}_l^{(t+1)}\| + \epsilon}. \tag{5.13}$$

where we use $\epsilon = 1 \cdot 10^{-9}$ to stabilize the computation. C_{out} is the output dimension or channels of the layer.

5.7 Experiments

We train ResNet-50 [He et al., 2016] for ImageNet [Deng et al., 2009] in all of our experiments. We use the same hyperparameters for the same mini-batch size. The hyperparameters for our results are shown in Table 5.2. We implement all of our methods using Chainer [Tokui et al., 2019]. Our Chainer extenstion is available at https://github.com/tyohei/chainerkfac. We initialize the weights by the `HeNormal` initializer of Chainer [9] with the default parameters.

5.7.1 Experiment Environment

We conduct all experiments on the ABCI (AI Bridging Cloud Infrastructure) [10] operated by the National Institute of Advanced Industrial Science and Technology (AIST) in Japan. ABCI has 1088 nodes with four NVIDIA Tesla V100 GPUs per node. Due to the additional memory required by NGD, all of our experiments use a mini-batch size of 32 per GPU. We were only given a 24 hour window to use the full machine so we had to tune the hyperparameters on a smaller number of nodes while mimicking the global mini-batch size of the full node run. For large mini-batch size experiments which cannot be executed directly (BS=65K, 131K requires 2048, 4096 GPUs, respectively) ,we used an accumulation method to mimic the behavior by accumulating the statistics $\boldsymbol{A}_{l-1}, \boldsymbol{B}_l, \boldsymbol{F}_{l,\text{unitBN}}$, and $\nabla_{\boldsymbol{\theta}_l} \mathcal{L}$ over multiple steps.

[9] https://docs.chainer.org/en/stable/reference/generated/chainer.initializers.HeNormal.html

[10] https://abci.ai/

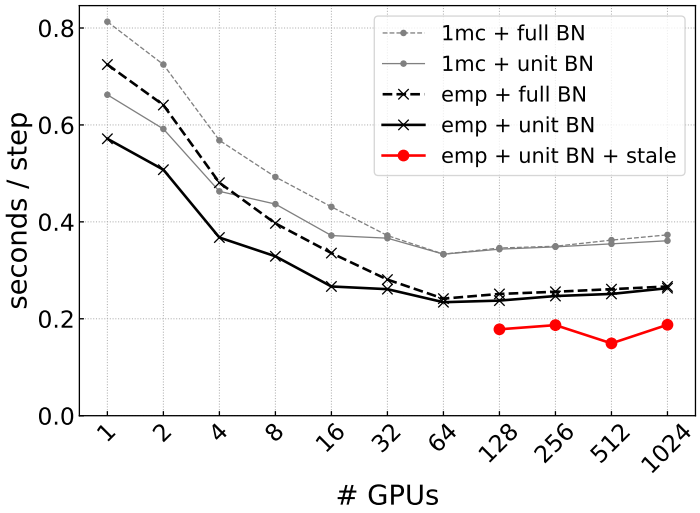


Figure 5.7: Time per step for trainig ResNet-50 (107 layers in total) on ImageNet with our scalable and practical NGD. Each GPU processes 32 images. `1mc` and `emp` correspond to NGD with $\hat{\boldsymbol{F}}_{l,\mathrm{1mc}}$ and that with $\hat{\boldsymbol{F}}_{l,\mathrm{emp}}$, respectively. `fullBN` and `unitBN` correspond to NGD and unit-wise NGD on BatchNorm parameters, respectively. `stale` corresponds to NGD with the stale statistics.

5.7.2 Extremely Large Mini-batch Training

We trained ResNet-50 for ImageNet classification task with extremely large mini-batch size BS={4,096, 8,192, 16,384, 32,768, 65,536, 131,072} and achieved a competitive top-1 validation accuracy ($\geq$ 74.9%) compared to highly-tuned SGD training. The summary of the training is shown in Table 5.1. For BS={4K, 8K, 16K, 32K, 65K}, the training converges in much less than 90 epochs, which is the usual number of epochs required by SGD-based training of ImageNet [Akiba et al., 2017, Goyal et al., 2017, You et al., 2017b, Jia et al., 2018, Mikami et al., 2018]. For BS={4K,8K,16K}, the required epochs to reach higher than 75% top-1 validation accuracy is 35 epochs. Even for a relatively large mini-batch size of BS=32K, NGD still converges in 45 epochs, half the number of epochs compared to SGD. Note that the calculation time is still decreasing while the number of epochs is less than double when we double the mini-batch size, assuming that doubling the mini-batch corresponds to doubling the number of GPUs (and halving the execution time). When we increase the mini-batch size to BS={32K,65K,131K}, we see a significantly small number of steps it takes to converge. At BS=32K and 65K, it takes 1760 steps (45 epochs) and 1173 steps (60 epochs), respectively. At BS=131K, there are less than 10 iterations per epoch since the dataset size of ImageNet is 1,281,167, and it only takes 873 steps to converge (90 epochs). None of the SGD-based training of ImageNet have sustained the top-1 validation accuracy at this mini-batch size. Furthermore, this is the first work that uses NGD for the training with extremely large mini-batch size BS={16K,32K,65K,131K} and achieves a competitive top-1 validation accuracy.

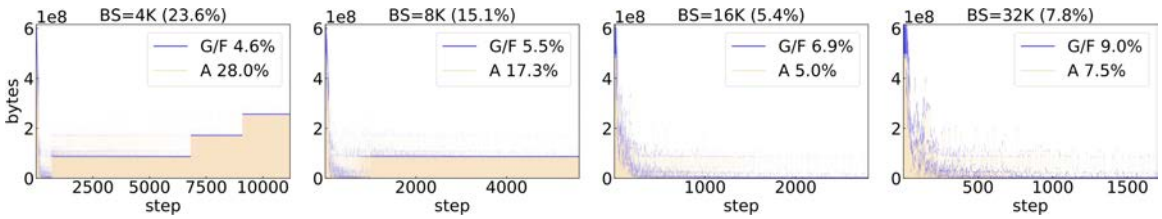


Figure 5.8: The communication amount (bytes) for the statistics ($\boldsymbol{A}_{l-1}, \boldsymbol{B}_l, \boldsymbol{F}_{l,\text{unitBN}}$) in each step in training ResNet-50 on ImageNet with BS={4K,8K,16K,32K} (stacked graph — the amount for `G/F` is stacked on the amount for `A`). `A` and `G/F` correspond to the communication amount for $\boldsymbol{A}_{l-1}$ and $\boldsymbol{B}_l/\boldsymbol{F}_{l,\text{unitBN}}$, respectively. The reduction rate (smaller is better) of the communication amount for all the statistics throughout the training is shown with the percentage (%).

5.7.3 Scalability

We measure the scalability of our distributed NGD implementation for training ResNet-50 on ImageNet. Figure 5.7 shows the time for one step with different number of GPUs and different techniques proposed in Section 5.4. Note that since we fix the number of images to be processed per GPU (=32), doubling the number of GPUs means doubling the total number of images (mini-batch size) to be processed in a step (*e.g.*, 32K images are processed with 1024 GPUs in a step). In a distributed training with multiple GPUs, it is considered ideal if this plot shows a flat line parallel to the x-axis, that is, the time per step is independent of the number of GPUs, and the number of images processed in a certain time increases linearly with the number of GPUs. From 1 GPU to 64 GPUs, however, we observed a *superlinear* scaling. For example, the time per step with 64 GPUs is 300% faster than that with 1 GPU for `emp+fullBN`. This is the consequence of our model-parallel design since ResNet-50 has 107 layers in total when all the Conv, FC, and BatchNorm layers are accounted for. With more than 128 GPUs, we observe slight performance degradation due to the communication overhead comes from `ReduceScatterV` and `AllGatherV` collective. Yet for `emp+unitBN+stale`, we see almost the ideal scaling from 128 GPUs to 1024 GPUs. Moreover, with 512 GPUs, which corresponds to BS=16K, we see a *superlinear* scaling, again. We discuss this in the next sub-section.

5.7.4 Effectiveness of Practical Natural Gradient

We examine the effectiveness of our practical NGD approaches proposed in Section 5.4 for training ResNet-50 on ImageNet with extreamely large mini-batch. We show that our practical techniques makes the overhead of NGD close to a negligible amount and improves training time significantly. The summary of the training time is shown in Table 5.1 and Figure 5.3.

Natural Gradient by Empirical Fisher. We compare the time and model accuracy in training by NGD with empirical Fisher and that with a Fisher estimation by a single Monte Carlo sampling ($\boldsymbol{C}$ vs $\boldsymbol{F}_{1\text{mc}}$) . In Figure 5.7, the time per a step by each training is labeled as `emp` and `1mc`, respectively. Due to the extra backward-pass required for constructing $\boldsymbol{F}_{1\text{mc}}$, `1mc` is slower than `emp` at any number

of GPUs. We do not see any difference in the convergence behavior (accuracy vs steps) and the final accuracy for training ResNet-50 on ImageNet with BS={4K,8K,16K,32K,65K,131K}. Note that we used the same hyperparameters tuned for `emp` for each BS (shown in Table 5.2) for the limitation of computational resource to tune for `1mc`.

Unit-Wise Natural Gradient. We also compare training with natural gradient and that with unit-wise natural gradient on BatchNorm parameters ($\boldsymbol{F}_l$ vs $\boldsymbol{F}_{l,\mathrm{unitBN}}$) . In Figure 5.7, the time per step for each method is labeled as `fullBN` and `unitBN`, respectively. From 1 GPU to 16 GPUs, we can see that `unitBN` effectively accelerates the time per step compare to `fullBN`. For more than 32 GPUs, we can see only a slight improvements since inverting statistics ($\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_l$) for all the layers are already distributed among enough number of processes, and inverting $\boldsymbol{F}_l$ is no longer a bottleneck of processing a step. We do not see any difference in the convergence behavior (accuracy vs steps) and the final accuracy for training ResNet-50 on ImageNet with BS={4K,8K,16K,32K,65K,131K}.

Natural Gradient with Stale Statistics. We apply our adaptive frequency strategy described in Section 5.4 to training ResNet-50 on ImageNet with BS={4K,8K,16K,32K}. In Figure 5.7, the time per a step with all the practical techniques is labeled as `emp+unitBN+stale`. The model accuracy before and after applying this technique, reduction rate (smaller is better) of the communication volume for the statistics, and speedup (`emp+unitBN` vs `emp+unitBN+stale`) is shown in Table 5.2. The communication amount (bytes) in the `ReduceScatterV` collective in each step during a training and the reduction rate are plotted in Figure 5.8. With BS=16K,32K, we can reduce the communication amount for the statistics ($\boldsymbol{A}_{l-1}, \boldsymbol{B}_l$ and $\boldsymbol{F}_l$) to 5.4%,7.8%, respectively. We might be able to attribute this significant reduction rate to the fact that the statistics with larger BS (16K,32K) is more stable than that with smaller BS (4K,8K). Note that though we show the reduction rate of the amount of communication, this rate is also applicable to estimate the reduction rate of the amount of computation for the statistics, and the cost for inverting them is also removed. With these improvements on NGD, we see almost an ideal scaling from 128 GPUs to 1024 GPUs, which corresponds to BS=4K to 32K.

5.7.5 Training ResNet-50 on ImageNet with NGD in 5.5 minutes

Finally, we combine all the practical techniques — empirical Fisher, unit-wise NGD and NGD with stale statistics. Using 1024 NVIDIA Tesla V100, we achieve 75.4 % top-1 accuracy with ResNet-50 for ImageNet in 5.5 minutes (1760 steps = 45 epochs, including a validation after each epoch). We used the same hyperparameters shown in Table 5.2. The training time and the validation accuracy are competitive with the results reported by related work that use SGD for training (the comparison is shown in Table 5.1). We refer to our training method as *Scarable and Practical NGD* (SP-NGD).

5.8 Discussion

In this work, we proposed a *Scalable and Practical Natural Gradient Descent* (SP-NGD), a framework which combines i) a large-scale distributed computational design with data and model hybrid parallelism for the Natural Gradient Descent (NGD) [Amari, 1998] and ii) practical Fisher information estimation techniques including Kronecker-Factored Approximate Curvature (K-FAC) [Martens and Grosse, 2015], that alleviates the computational overhead of NGD over SGD.

Using our SP-NGD framework, we showed the advantages of the NGD over first-order stochastic gradient descent (SGD) for training ResNet-50 on ImageNet classification with extremely large mini-batches. We introduced several schemes for the training using the NGD with mini-batch sizes up to 131,072 and achieved over 75% top-1 accuracy in much fewer number of steps compared to the existing work using the SGD with large mini-batch. By using strong data augmentation (i.e., mixup [Zhang et al., 2018c]), we were able to show that solutions found by a second-order method generalize as well as that of SGD, even for extremely large mini-batches. Note that this data augmentation helps second-order methods more than they do SGD, so it is not a matter of simply increasing the baseline accuracy through data augmentation. Our SP-NGD framework allowed us to train on 1024 GPUs and achieved 75.4% in 5.5 minutes. This is the first work which observes the relationship between the FIM of ResNet-50 and its training on large mini-batches ranging from 4K to 131K.

More accurate and efficient FIM estimation. We showed that NGD using the empirical Fisher matrix (`emp`) is much faster than that with an estimation using a single Monte Carlo (`1mc`), which is widely used by related work on the approximate natural gradient. Although it is stated that `emp` is not a good approximation of the NGD in the literature [Kunstner et al., 2019, Thomas et al., 2019], we observed the same convergence behavior as `1mc` for training ResNet-50 on ImageNet. We might be able to attribute this result to the fact that `emp` is a good enough approximation to keep the behavior of the *true* NGD or that even `1mc` is not a good approximation. To know whether these hypotheses are correct or not and to examine the actual value of the true NGD, we need a more accurate and effcient estimation of the NGD with less computational cost.

Towards Bayesian deep learning. NGD has been applied to Bayesian deep learning for estimating the posterior distribution of the network parameters. For example, K-FAC [Martens and Grosse, 2015] has been applied to Bayesian deep learning to realize *Noisy Natural Gradient* [Zhang et al., 2018a], and our distributed NGD has been applied to that at ImageNet scale [Osawa et al., 2019a]. We similarly expect that our SP-NGD framework will accelerate Bayesian deep learning research using natural gradient methods.

Chapter 6

Practical deep learning with Bayesian principles

Bayesian methods promise to fix many shortcomings of deep learning, but they are impractical and rarely match the performance of standard methods, let alone improve them. In this paper, we demonstrate practical training of deep networks with natural-gradient variational inference. By applying techniques such as batch normalisation, data augmentation, and distributed training, we achieve similar performance in about the same number of epochs as the Adam optimizer, even on large datasets such as ImageNet. Importantly, the benefits of Bayesian principles are preserved: predictive probabilities are well-calibrated, uncertainties on out-of-distribution data are improved, and continual-learning performance is boosted. This work enables practical deep learning while preserving benefits of Bayesian principles. A PyTorch implementation[1] is available as a plug-and-play optimizer.

6.1 Bayesian inference in deep learning

Deep learning has been extremely successful in many fields such as computer vision [Krizhevsky et al., 2012], speech processing [Hinton et al., 2012], and natural-language processing [Mikolov et al., 2013], but it is also plagued with several issues that make its application difficult in many other fields. For example, it requires a large amount of high-quality data and it can overfit when dataset size is small. Similarly, sequential learning can cause forgetting of past knowledge [Kirkpatrick et al., 2017], and lack of reliable confidence estimates and other robustness issues can make it vulnerable to adversarial attacks [Bradshaw et al., 2017]. Ultimately, due to such issues, application of deep learning remains challenging, especially for applications where human lives are at risk.

Bayesian principles have the potential to address such issues. For example, we can represent uncertainty using the posterior distribution, enable sequential learning using Bayes' rule, and reduce overfitting with Bayesian model averaging Hoeting et al. [1999]. The use of such Bayesian principles

[1]The code is available at `https://github.com/team-approx-bayes/dl-with-bayes`.

for neural networks has been advocated from very early on. Bayesian inference on neural networks were all proposed in the 90s, e.g., by using MCMC methods [Neal, 1995], Laplace's method [Mackay, 1991], and variational inference (VI) [Hinton, 1993, Barber and Bishop, 1998, Saul et al., 1996, Peterson and Anderson, 1987]. Benefits of Bayesian principles are even discussed in machine-learning textbooks [Mackay, 2003, Bishop, 2006]. Despite this, they are rarely employed in practice. This is mainly due to computational concerns, unfortunately overshadowing their theoretical advantages.

The difficulty lies in the computation of the posterior distribution, which is especially challenging for deep learning. Even approximation methods, such as VI and MCMC, have historically been difficult to scale to large datasets such as ImageNet [Deng et al., 2009, Russakovsky et al., 2015]. Due to this, it is common to use less principled approximations, such as MC-dropout [Gal and Ghahramani, 2016], even though they are not ideal when it comes to fixing the issues of deep learning. For example, MC-dropout is unsuitable for continual learning [Kirkpatrick et al., 2017] since its posterior approximation does not have mass over the whole weight space. It is also found to perform poorly for sequential decision making [Riquelme et al., 2018]. The form of the approximation used by such methods is usually rigid and cannot be easily improved, e.g., to other forms such as a mixture of Gaussians. The goal of this paper is to make more principled Bayesian methods, such as VI, practical for deep learning, thereby helping researchers tackle its key limitations.

We demonstrate practical training of deep networks by using recently proposed natural-gradient VI methods. These methods resemble the Adam optimizer, enabling us to leverage existing techniques for initialization, momentum, batch normalisation, data augmentation, and distributed training. As a result, we obtain similar performance in about the same number of epochs as Adam when training many popular deep networks (e.g., LeNet, AlexNet, ResNet) on datasets such as CIFAR-10 and ImageNet (see Fig. 6.1). The results show that, despite using an approximate posterior, the training methods preserve the benefits coming from Bayesian principles. Compared to standard deep-learning methods, the predictive probabilities are well-calibrated, uncertainties on out-of-distribution inputs are improved, and performance for continual-learning tasks is boosted. Our work shows that practical deep learning is possible with Bayesian methods and aims to support further research in this area.

Related work. Previous VI methods, notably by Graves [2011] and Blundell et al. [2015], require significant implementation and tuning effort to perform well, *e.g.*, on convolution neural networks (CNN). Slow convergence is found to be especially problematic for sequential problems [Riquelme et al., 2018]. There appears to be no reported results with complex networks on large problems, such as ImageNet. Our work solves these issues by applying deep-learning techniques to natural-gradient VI [Khan et al., 2018, Zhang et al., 2018a].

In their paper, Zhang et al. [2018a] also employed data augmentation and batch normalisation for a natural-gradient method called *Noisy K-FAC* (see Section 6.5) and showed results on VGG

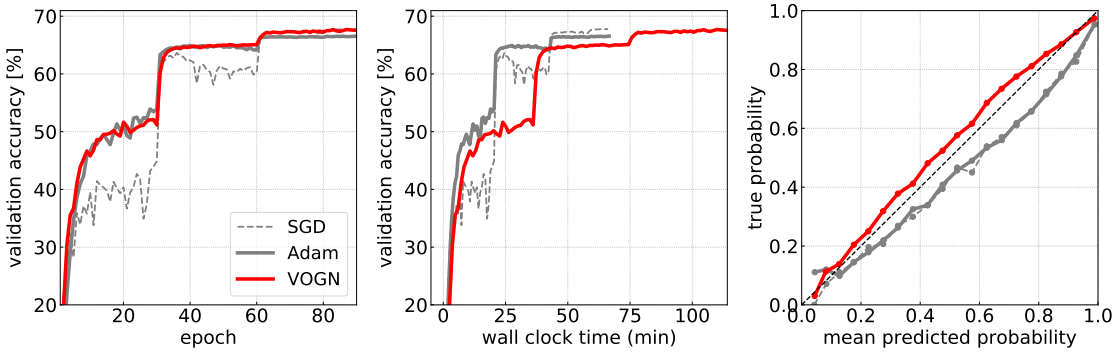


Figure 6.1: Comparing VOGN [Khan et al., 2018], a natural-gradient VI method, to Adam and SGD, training ResNet-18 on ImageNet. The two left plots show that VOGN and Adam have similar convergence behaviour and achieve similar performance in about the same number of epochs. VOGN achieves 67.38% on validation compared to 66.39% by Adam and 67.79% by SGD. Run-time of VOGN is 76 seconds per epoch compared to 44 seconds for Adam and SGD. The rightmost figure shows the calibration curve. VOGN gives calibrated predictive probabilities (the diagonal represents perfect calibration).

on CIFAR-10. However, a mean-field method called Noisy Adam was found to be unstable with batch normalisation. In contrast, we show that a similar method, called *Variational Online Gauss-Newton* (VOGN), proposed by Khan et al. [2018], works well with such techniques. We show results for distributed training with Noisy K-FAC on Imagenet, but do not provide extensive comparisons since tuning it is time-consuming. Many of our techniques can speed-up Noisy K-FAC, which is promising.

Many other approaches have recently been proposed to compute posterior approximations by training deterministic networks [Ritter et al., 2018b, Maddox et al., 2019, Mandt and Hoffman, 2017]. Similarly to MC-dropout, their posterior approximations are not flexible, making it difficult to improve the accuracy of their approximations. On the other hand, VI offers a much more flexible alternative to apply Bayesian principles to deep learning.

6.2 Deep learning with Bayesian principles and its challenges

The success of deep learning is partly due to the availability of scalable and practical methods for training deep neural networks (DNNs). Network training is formulated as an optimization problem where a loss between the data and the DNN's predictions is minimised. For example, in a supervised learning task with a dataset $\mathcal{S}$ of N inputs $\boldsymbol{x}_i$ and corresponding outputs $\boldsymbol{y}_i$ of length K, we minimise a loss of the following form:

$$\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \delta \boldsymbol{\theta}^\top \boldsymbol{\theta},$$

where

$$\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) := \frac{1}{N} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathcal{S}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) = \frac{1}{N} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathcal{S}} h(\boldsymbol{y}_n, f(\boldsymbol{x}_n)) \,,$$

$f(\boldsymbol{x}) \in \mathbb{R}^K$ denotes the DNN outputs with weights $\boldsymbol{\theta}$, $h(\boldsymbol{y}, \boldsymbol{z}))$ denotes a differentiable loss function between a label $\boldsymbol{y} \in \mathbb{R}^K$ and the output $\boldsymbol{z} \in \mathbb{R}^K$, and $\delta > 0$ is the L_2 regularizer.[2] Deep learning relies on stochastic gradient (SG) methods to minimize such loss functions. The most commonly used optimizers, such as stochastic gradient descent (SGD), RMSprop [Tieleman and Hinton, 2012], and Adam [Kingma and Ba, 2015], take the following form[3] (all operations below are element-wise):

$$\begin{aligned} \boldsymbol{\theta}^{(t+1)} &\leftarrow \boldsymbol{\theta}^{(t)} - \alpha_t \frac{\hat{\boldsymbol{g}}(\boldsymbol{\theta}^{(t)}) + \delta \boldsymbol{\theta}^{(t)}}{\sqrt{\boldsymbol{s}^{(t+1)}} + \epsilon}, \\ \boldsymbol{s}^{(t+1)} &\leftarrow (1 - \beta_t) \boldsymbol{s}^{(t)} + \beta_t \left(\hat{\boldsymbol{g}}(\boldsymbol{\theta}^{(t)}) + \delta \boldsymbol{\theta}^{(t)} \right)^2, \end{aligned} \tag{6.1}$$

where t is the number of iteration, $\alpha_t > 0$ and $0 < \beta_t < 1$ are learning rates at t, $\epsilon > 0$ is a small scalar, and $\hat{\boldsymbol{g}}(\boldsymbol{\theta})$ is the stochastic gradients at $\boldsymbol{\theta}$ defined as follows:

$$\hat{\boldsymbol{g}}(\boldsymbol{\theta}) := \frac{1}{|\mathcal{B}_t|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta})$$

using a minibatch $\mathcal{B}_t \subset \mathcal{S}$. This simple update scales extremely well and can be applied to very large problems. With techniques such as initialization protocols, momentum, weight-decay, Batch Normalization [Ioffe and Szegedy, 2015], and data augmentation, it also achieves good performance for many problems.

In contrast, the full Bayesian approach to deep learning is computationally very expensive. The posterior distribution can be obtained using Bayes' rule:

$$p(\boldsymbol{\theta}|\mathcal{S}) = \exp\left(-N \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})/\tau\right) p(\boldsymbol{\theta}) / p(\mathcal{S}) \,,$$

where $0 < \tau \leq 1$.[4] This is costly due to the computation of the marginal likelihood $p(\mathcal{S})$, a high-dimensional integral that is difficult to compute for large networks. Variational inference (VI) is a principled approach to more scalably estimate an approximation to $p(\boldsymbol{\theta}|\mathcal{S})$. The main idea is to employ a parametric approximation, *e.g.*, a Gaussian $q(\boldsymbol{\theta})$:

$$p(\boldsymbol{\theta}|\mathcal{S}) \approx q(\boldsymbol{\theta}) := \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. The parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ can then be estimated by maximising the *evidence lower bound (ELBO)*:

$$L(\boldsymbol{\mu}, \boldsymbol{\Sigma}) := -N \mathbb{E}_q \left[\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})\right] - \tau \mathrm{KL}(q(\boldsymbol{\theta}) \,\|\, p(\boldsymbol{\theta})) \,, \tag{6.2}$$

[2] This regularizer is sometimes set to 0 or a very small value.

[3] Alternate versions with weight-decay and momentum differ from this update [Loshchilov and Hutter, 2019]. We present a form useful to establish the connection between SG methods and natural-gradient VI.

[4] This is a tempered posterior [Vovk, 1990] setup where τ is set $\neq 1$ when we expect model misspecification and/or adversarial examples [Vaart and Ghosal, 2017]. Setting $\tau = 1$ recovers standard Bayesian inference.

where $\mathrm{KL}(\cdot \,\|\, \cdot)$ denotes the Kullback-Leibler divergence. By using more complex approximations, we can further reduce the approximation error, but at a computational cost. By formulating Bayesian inference as an optimization problem, VI enables a practical application of Bayesian principles.

Despite this, VI has remained impractical for training large deep networks on large datasets. Existing methods, such as Graves [2011] and Blundell et al. [2015], directly apply popular SG methods to optimize the variational parameters in the ELBO, yet they fail to get a reasonable performance on large problems, usually converging very slowly. The failure of such direct applications of deep learning methods to VI is not surprising. The techniques used in one field may not directly lead to improvements in the other, but it will be useful if they do, *e.g.*, if we can optimize the ELBO in a way that allows us to exploit the tricks and techniques of deep learning and boost the performance of VI. The goal of this work is to do just that. We now describe our methods in detail.

6.3 Practical deep learning with natural gradient variational inference

We propose Natural Gradient VI (NGVI) methods for practical deep learning with Bayesian principles. The Natural Gradient [Amari, 1998] update takes a simple form when estimating exponential-family approximations [Khan and Nielsen, 2018, Khan and Lin, 2017]. When $p(\boldsymbol{\theta}) := \mathcal{N}(\boldsymbol{\theta}|0, \boldsymbol{I}/\delta)$, the update of the natural-parameter $\boldsymbol{\lambda}$ is performed by using the stochastic gradient of the *expected regularised-loss*:

$$\boldsymbol{\lambda}^{(t+1)} = (1 - \tau\rho)\boldsymbol{\lambda}^{(t)} - \rho \nabla_{\boldsymbol{\mu}} \mathbb{E}_q \left[\mathcal{L}(\boldsymbol{\theta}) + \frac{1}{2}\tau\delta\boldsymbol{\theta}^\top\boldsymbol{\theta} \right], \tag{6.3}$$

where $\rho > 0$ is the learning rate, and we note that the stochastic gradients are computed with respect to $\boldsymbol{\mu}$, the *expectation parameters* of q. The *moving average* above helps to deal with the stochasticity of the gradient estimates, and is very similar to the moving average used in deep learning (see equation 6.1). When τ is set to 0, the update essentially minimises the regularised loss (see Section 5 in Khan et al. [2018]). These properties of NGVI makes it an ideal candidate for deep learning.

Recent work by Khan et al. [2018] and Zhang et al. [2018a] further show that, when q is Gaussian, the update equation 6.3 assumes a form that is strikingly similar to the update equation 6.1. For example, the Variational Online Gauss-Newton (VOGN) method of Khan et al. [2018] estimates a Gaussian with mean $\boldsymbol{\mu}^{(t)}$ and a diagonal covariance matrix $\boldsymbol{\Sigma}^{(t)}$ using the following update:

$$\begin{aligned}
\boldsymbol{\mu}^{(t+1)} &\leftarrow \boldsymbol{\mu}^{(t)} - \alpha_t \frac{\hat{\boldsymbol{g}}(\boldsymbol{\theta}^{(t)}) + \tilde{\delta}\boldsymbol{\mu}^{(t)}}{\boldsymbol{s}^{(t+1)} + \tilde{\delta}}, \\
\boldsymbol{s}^{(t+1)} &\leftarrow (1 - \tau\beta_t)\boldsymbol{s}^{(t)} + \beta_t \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \left(\boldsymbol{g}_n(\boldsymbol{\theta}^{(t)}) \right)^2,
\end{aligned} \tag{6.4}$$

where

$$\boldsymbol{g}_n(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \,,$$
$$\boldsymbol{\theta}^{(t)} \sim \mathcal{N}(\boldsymbol{\theta} | \boldsymbol{\mu}^{(t)}, \boldsymbol{\Sigma}^{(t)})$$

with $\boldsymbol{\Sigma}^{(t)} := \text{diag}(1/(N(\boldsymbol{s}^{(t)} + \tilde{\delta})))$, $\tilde{\delta} := \tau\delta/N$, $\alpha_t, \beta_t > 0$ are learning rates, and $\boldsymbol{v}^2 := \boldsymbol{v} \odot \boldsymbol{v}$. Operations are performed element-wise. Similarly to equation 6.1, the vector $\boldsymbol{s}^{(t)}$ adapts the learning rate and is updated using a moving average. A major difference in VOGN is that the update of $\boldsymbol{s}^{(t)}$ is now based on a Gauss-Newton approximation [Graves, 2011] which uses

$$\frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \left(\boldsymbol{g}_n(\boldsymbol{\theta}^{(t)}) \right)^2 = \text{diag}\left(\boldsymbol{C}(\boldsymbol{\theta}^{(t)}) \right) .$$

This is fundamentally different from the SG update in equation 6.1 which instead uses the gradient-magnitude [Bottou et al., 2018].

$$\left(\frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \boldsymbol{g}_n(\boldsymbol{\theta}^{(t)}) + \delta \boldsymbol{\theta}^{(t)} \right)^2 .$$

The first approach uses the sum *outside* the square while the second approach uses it *inside*. VOGN is therefore an approximate second-order method (diagonal $\boldsymbol{C}$ as the preconditioner) and, similarly to Newton's method, does not need a square-root over $\boldsymbol{s}^{(t)}$. Implementation of this step requires an additional calculation (see Section 6.6) which makes VOGN a bit slower than Adam, but VOGN is expected to give better variance estimates (see Theorem 1 in Khan et al. [2018]).

The main contribution of this paper is to demonstrate practical training of deep networks using VOGN. Since VOGN takes a similar form to SG methods, we can easily borrow existing deep learning techniques to improve performance. We will now describe these techniques in detail. Pseudo-code for (distributed-memory version of) VOGN is shown in Algorithm 6.

Batch Normalization. Batch Normalization (BatchNorm) [Ioffe and Szegedy, 2015] has been found to significantly speed up and stabilize training of neural networks, and is widely used in deep learning. BatchNorm layers are inserted between neural network layers. They help stabilize each layer's input distribution by normalizing the running average of the inputs' mean and variance. In our VOGN implementation, we simply use the existing implementation with default hyperparameter settings. We do not apply L2 regularisation and weight decay to BatchNorm parameters, like in Goyal et al. [2017], or maintain uncertainty over the BatchNorm parameters. This straightforward application of batch normalization works for VOGN.

Data Augmentation. When training on image datasets, data augmentation (DA) techniques can improve performance drastically [Goyal et al., 2017]. We consider two common real-time data augmentation techniques: random cropping and horizontal flipping. After randomly selecting a minibatch at each iteration, we use a randomly selected cropped version of all images. Each image in the minibatch has a 50% chance of being horizontally flipped.

We find that directly applying DA gives slightly worse performance than expected, and also affects the calibration of the resulting uncertainty. However, DA increases the effective sample size. We therefore modify it to be ρN where $\rho \geq 1$, improving performance (see step 10 in Algorithm 6). The reason for this performance boost might be due to the complex relationship between the regularization δ and N. For the regularized loss $\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) + \delta \boldsymbol{\theta}^\top \boldsymbol{\theta}$, the two are unidentifiable, i.e., we can multiply δ by a constant and reduce N by the same constant without changing the minimum. However, in a Bayesian setting (like in equation 6.2), the two quantities are separate, and therefore changing the data might also change the optimal prior variance hyperparameter in a complicated way. This needs further theoretical investigations, but our simple fix of scaling N seems to work well in the experiments.

We set ρ by considering the specific DA techniques used. When training on CIFAR-10, the random cropping DA step involves first padding the 32x32 images to become of size 40x40, and then taking randomly selected 28x28 cropped images. We consider this as effectively increasing the dataset size by a factor of 5 (4 images for each corner, and one central image). The horizontal flipping DA step doubles the dataset size (one dataset of unflipped images, one for flipped images). Combined, this gives $\rho = 10$. Similar arguments for ImageNet DA techniques give $\rho = 5$. Even though ρ is another hyperparameter to set, we find that its precise value does not matter much. Typically, after setting an estimate for ρ, tuning δ a little seems to work well (see Section 6.9).

Momentum and initialization. It is well known that both momentum and good initialization can improve the speed of convergence for SG methods in deep learning [Sutskever et al., 2013]. Since VOGN is similar to Adam, we can implement momentum in a similar way. This is shown in step 25 of Algorithm 6, where β_1 is the momentum rate. We initialize the mean $\boldsymbol{\mu}$ in the same way the parameters are initialized in Adam (we use `init.xavier_normal` [Glorot and Bengio, 2010] in PyTorch [Paszke et al., 2019]). For the momentum term $\boldsymbol{m}$, we use the same initialization as Adam (initialized to 0). VOGN requires an additional initialization for the variance $\boldsymbol{\sigma}^2$. For this, we first run a forward pass through the first minibatch, calculate the average of the squared gradients and initialize the scale $\boldsymbol{s}_0$ with it (see step 9 in Algorithm 6). This implies that the variance is initialized to $\boldsymbol{\sigma}_0^2 = \tau/(N(\boldsymbol{s}_0 + \tilde{\delta}))$. For the tempering parameter τ, we use a schedule where it is increased from a small value (*e.g.*., 0.1) to 1. With these initialization protocols, VOGN is able to mimic the convergence behaviour of Adam in the beginning.

Learning rate scheduling. A common approach to quickly achieve high validation accuracies is to use a specific learning rate schedule [Goyal et al., 2017]. The learning rate (denoted by α in Algorithm 6) is regularly decayed by a factor (typically a factor of 10). The frequency and timings of this decay are usually pre-specified. In VOGN, we use the same schedule used for Adam, which works well.

Distributed-memory training. We also employ distributed-memory training for VOGN to perform large experiments quickly. We can parallelize computation both over data and Monte-Carlo (MC) samples. Data parallelism is useful to split up large minibatch sizes. This is followed by averaging over multiple MC samples and their losses on a single GPU. MC sample parallelism is useful when minibatch size is small, and we can copy the entire minibatch and process it on a single GPU. Algorithm 6 illustrates our distributed-memory scheme. We use a combination of these two parallelism techniques with different MC samples for different inputs. This theoretically reduces the variance during training (see Equation 5 in Kingma and Ba [2015]), but sometimes requires averaging over multiple MC samples to get a sufficiently low variance in the early iterations. Overall, we find that this type of distributed-memory training is essential for fast training on large problems such as ImageNet.

Implementation of the Gauss-Newton update in VOGN. As discussed earlier, VOGN uses the Gauss-Newton approximation, which is fundamentally different from Adam. In this approximation, the gradients on individual data examples are first squared and then averaged afterwards (see step 20 in Algorithm 6 which implements the update for $\boldsymbol{s}^{(t)}$ shown in equation 6.4). We need extra computation to get access to individual gradients, due to which, VOGN is slower Adam or SGD (*e.g.*, in Fig. 6.1). However, this is not a theoretical limitation and this can be improved if a framework enables an easy computation of the individual gradients. Details of our implementation are described in Section 6.6. This implementation is much more efficient than a naive one where gradients over examples are stored and the sum over the square is computed sequentially. Our implementation usually brings the running time of VOGN to within 2-5 times of the time that Adam takes.

Tuning VOGN. Currently, there is no common recipe for tuning the algorithmic hyperparameters for VI, especially for large-scale tasks like ImageNet classification. One key idea we use in our experiments is to start with Adam hyperparameters and then make sure that VOGN training closely follows an Adam-like trajectory in the beginning of training. To achieve this, we divide the tuning into an *optimization part* and a *regularization part*. In the *optimization part*, we first tune the hyperparameters of a deterministic version of VOGN, called the online Gauss-Newton (OGN) method. This method, described in Section 6.7, is more stable than VOGN since it does not require MC sampling, and can be used as a stepping stone when moving from Adam/SGD to VOGN. After reaching a competitive performance to Adam/SGD by OGN, we move to the *regularization part*, where we tune the prior precision δ, the tempering parameter τ, and the number of MC samples K for VOGN. We initialize our search by setting the prior precision δ using the L2-regularisation parameter used for OGN, as well as the dataset size N. Another technique is to warm-up the parameter τ towards $\tau = 1$ (also see the "momentum and initialization" part). Setting τ to smaller values usually stabilizes the training, and increasing it slowly also helps during tuning. We also add

an *external damping factor* $\gamma > 0$ to the moving average $\boldsymbol{s}^{(t)}$. This increases the lower bound of the eigenvalues of the diagonal covariance $\boldsymbol{\Sigma}^{(t)}$ and prevents the noise and the step size from becoming too large. We find that a mix of these techniques works well for the problems we considered.

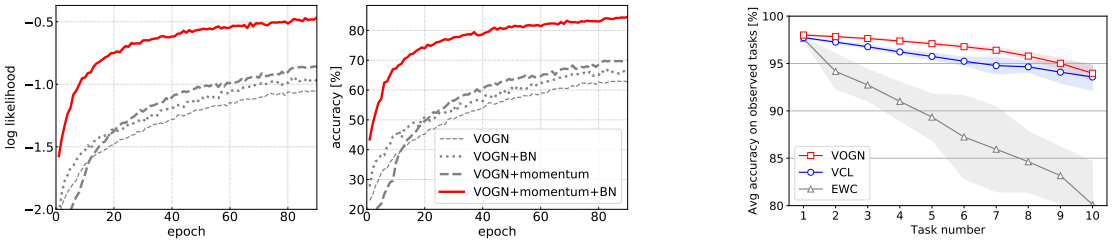
Algorithm 6: Distributed-memory Variational Online Gauss Newton (VOGN)

Learning rate α
Momentum rate β_1
Exp. moving average rate β_2
Prior precision δ
External damping factor γ
Tempering parameter τ
MC samples for training K
Data augmentation factor ρ
Initialize $\boldsymbol{\mu}$, $\boldsymbol{s}$, $\boldsymbol{m}$.
$N \leftarrow \rho N$, $\tilde{\delta} \leftarrow \tau\delta/N$.
repeat
  Sample a minibatch $\mathcal{B} \subset \mathcal{S}$ of size M.
  Split $\mathcal{B}$ into each GPU (local minibatch $\mathcal{B}_{local}$).
  for each GPU in parallel **do**
    for $k = 1, 2, \ldots, K$ **do**
      Sample $\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.
      $\boldsymbol{\theta}^{(k)} \leftarrow \boldsymbol{\mu} + \epsilon\boldsymbol{\sigma}$ with $\boldsymbol{\sigma} \leftarrow (1/(N(\boldsymbol{s} + \tilde{\delta} + \gamma)))^{1/2}$.
      Compute $\boldsymbol{g}_n^{(k)} \leftarrow \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}^{(k)}), \forall i \in \mathcal{B}_{local}$ using the method described in Section 6.6.
      $\hat{\boldsymbol{g}}_k \leftarrow \frac{1}{M} \sum_{n \in \mathcal{B}_{local}} \boldsymbol{g}_n^{(k)}$.
      $\hat{\boldsymbol{g}}_k^2 \leftarrow \frac{1}{M} \sum_{i \in \mathcal{B}_{local}} (\boldsymbol{g}_n^{(k)})^2$.
    end for
    $\hat{\boldsymbol{g}} \leftarrow \frac{1}{K} \sum_{k=1}^{K} \hat{\boldsymbol{g}}_k$ and $\hat{\boldsymbol{g}}^2 \leftarrow \frac{1}{K} \sum_{k=1}^{K} \hat{\boldsymbol{g}}_k^2$.
  end for
  AllReduce $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{g}}^2$.
  $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (\hat{\boldsymbol{g}} + \tilde{\delta}\boldsymbol{\mu})$.
  $\boldsymbol{s} \leftarrow (1 - \tau\beta_2)\boldsymbol{s} + \beta_2 \hat{\boldsymbol{g}}^2$.
  $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha \boldsymbol{m}/(\boldsymbol{s} + \tilde{\delta} + \gamma)$.
until stopping criterion is met

6.4 Experiments

In this section, we present experiments on fitting several deep networks on CIFAR-10 and ImageNet. Our experiments demonstrate practical training using VOGN on these benchmarks and show performance that is competitive with Adam and SGD. We also assess the quality of the posterior approximation, finding that benefits of Bayesian principles are preserved.

CIFAR-10 [Krizhevsky and Hinton, 2009] contains 10 classes with 50,000 images for training and 10,000 images for validation. For ImageNet, we train with 1.28 million training examples and validate on 50,000 examples, classifying between 1,000 classes. We used a large minibatch size $|\mathcal{B}| = 4,096$ and parallelise them across 128 GPUs (NVIDIA Tesla P100). We compare the following methods on CIFAR-10: Adam, MC-dropout [Gal and Ghahramani, 2016]. For ImageNet, we also compare



(a) Effect of momentum and Batch Normalization.

(b) Continual Learning

Figure 6.2: Figure (a) shows that momentum and Batch Normalization improve the performance of VOGN. The results are for training ResNet-18 on CIFAR-10. Figure (b) shows comparison for a continual-learning task on the Permuted MNIST dataset. VOGN performs at least as well (average accuracy) as VCL over 10 tasks. We also find that, for each task, VOGN converges much faster, taking only 100 epochs per task as opposed to 800 epochs taken by VCL (plots not shown).

to SGD, K-FAC [Martens and Grosse, 2015], and Noisy K-FAC [Zhang et al., 2018a]. We do not consider Noisy K-FAC for other comparisons since tuning is difficult. We compare 3 architectures: LeNet-5 [LeCun et al., 1998a], AlexNet [Krizhevsky et al., 2012], ResNet-18 [He et al., 2016]. We only compare to Bayes by Backprop (BBB) [Blundell et al., 2015] for CIFAR-10 with LeNet-5 since it is very slow to converge for larger-scale experiments. We carefully set the hyperparameters of all methods, following the best practice of large distributed training [Goyal et al., 2017] as the initial point of our hyperparameter tuning. The full set of hyperparameters is in Section 6.8.

6.4.1 Performance on CIFAR-10 and ImageNet

We start by showing the effectiveness of momentum and BatchNorm for boosting the performance of VOGN. Figure 6.2a shows that these methods significantly speed up convergence and performance (in terms of both accuracy and log likelihoods).

Figures 6.1 and 6.3 compare the convergence of VOGN to Adam (for all experiments), SGD (on ImageNet), and MC-dropout (on the rest). VOGN shows similar convergence and its performance is competitive with these methods. We also try BBB on LeNet-5, where it converges prohibitively slowly, performing very poorly. We are not able to successfully train other architectures using this approach. We found it far simpler to tune VOGN because we can borrow all the techniques used for Adam. Figure 6.3 also shows the importance of DA in improving performance.

Table 6.1 gives a final comparison of train/validation accuracies, negative log likelihoods, epochs required for convergence, and run-time per epoch. We can see that the accuracy, log likelihoods, and the number of epochs are comparable. VOGN is 2-5 times slower than Adam and SGD. This is mainly due to the computation of individual gradients required in VOGN (see the discussion in Section 6.3). We clearly see that by using deep learning techniques on VOGN, we can perform practical deep learning. This is not possible with methods such as BBB.

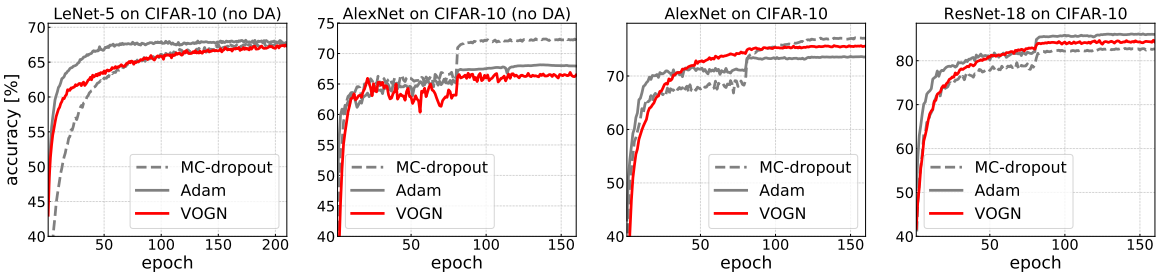


Figure 6.3: Validation accuracy for various architectures trained on CIFAR-10 (DA: Data Augmentation). VOGN's convergence and validation accuracies are comparable to Adam and MC-dropout.

Due to the Bayesian nature of VOGN, there are some trade-offs to consider. Reducing the prior precision (δ in Algorithm 6) results in higher validation accuracy, but also larger train-test gap (more overfitting). This is shown in Section 6.9 for VOGN on ResNet-18 on ImageNet. As expected, when the prior precision is small, performance is similar to non-Bayesian methods. We also show the effect of changing the effective dataset size ρ in Section 6.9: note that, since we are going to tune the prior variance δ anyway, it is sufficient to set ρ to its correct order of magnitude. Another trade-off concerns the number of Monte-Carlo (MC) samples, shown in Section 6.10. Increasing the number of training MC samples (up to a limit) improves VOGN's convergence rate and stability, but also increases the computation. Increasing the number of MC samples during testing improves generalisation, as expected due to averaging.

Finally, a few comments on the performance of the other methods. Adam regularly overfits the training set in most settings, with large train-test differences in both validation accuracy and log likelihood. One exception is LeNet-5, which is most likely due to the small architecture which results in underfitting (this is consistent with the low validation accuracies obtained). In contrast to Adam, MC-dropout has small train-test gap, usually smaller than VOGN's. However, we will see in Section 6.4.2 that this is because of underfitting. Moreover, the performance of MC-dropout is highly sensitive to the dropout rate (see Section 6.11 for a comparison of different dropout rates). On ImageNet, Noisy K-FAC performs well too. It is slower than VOGN, but it takes fewer epochs. Overall, wall clock time is about the same as VOGN.

6.4.2 Quality of the Predictive Probabilities

In this section, we compare the quality of the predictive probabilities for various methods. For Bayesian methods, we compute these probabilities by averaging over the samples from the posterior approximations (see Section 6.12 for details). For non-Bayesian methods, these are obtained using the point estimate of the weights. We compare the probabilities using the following metrics: validation negative log-likelihood (NLL), area under ROC (AUROC) and expected calibration curves (ECE) [Naeini et al., 2015, Guo et al., 2017]. For the first and third metric, a lower number is

Table 6.1: Performance comparisons on different dataset/architecture combinations. Out of the 15 metrics (NLL, ECE, and AUROC on 5 dataset/architecture combinations), VOGN performs the best or tied best on 10 ,and is second-best on the other 5. Here DA means 'Data Augmentation', NLL refers to 'Negative Log Likelihood' (lower is better), ECE refers to 'Expected Calibration Error' (lower is better), AUROC refers to 'Area Under ROC curve' (higher is better). BBB is the Bayes By Backprop method. For ImageNet, the reported accuracy and negative log likelihood are the median value from the final 5 epochs. All hyperparameter settings are in Section 6.8. See Table 6.3 for standard deviations. † BBB is not parallelized (other methods have 4 processes), with 1 MC sample used for the convolutional layers (VOGN uses 6 samples per process).

Dataset/ Architecture	optimizer	Train/Validation Accuracy (%)	Validation NLL	Epochs	Time/ epoch (s)	ECE	AUROC
CIFAR-10/ LeNet-5 (no DA)	Adam	71.98 / **67.67**	**0.937**	210	6.96	**0.021**	0.794
	BBB	66.84 / 64.61	1.018	800	11.43†	0.045	0.784
	MC-dropout	68.41 / **67.65**	0.99	210	6.95	0.087	**0.797**
	VOGN	70.79 / **67.32**	**0.938**	210	18.33	0.046	**0.8**
CIFAR-10/ AlexNet (no DA)	Adam	100.0 / 67.94	2.83	161	3.12	0.262	0.793
	MC-dropout	97.56 / **72.20**	1.077	160	3.25	0.140	**0.818**
	VOGN	79.07 / 69.03	**0.93**	160	9.98	**0.024**	0.796
CIFAR-10/ AlexNet	Adam	97.92 / 73.59	1.480	161	3.08	0.262	0.793
	MC-dropout	80.65 / **77.04**	**0.667**	160	3.20	0.114	0.828
	VOGN	81.15 / 75.48	0.703	160	10.02	**0.016**	**0.832**
CIFAR-10/ ResNet-18	Adam	97.74 / **86.00**	0.55	160	11.97	0.082	**0.877**
	MC-dropout	88.23 / 82.85	0.51	161	12.51	0.166	0.768
	VOGN	91.62 / 84.27	**0.477**	161	53.14	**0.040**	**0.876**
ImageNet/ ResNet-18	SGD	82.63 / **67.79**	**1.38**	90	44.13	0.067	0.856
	Adam	80.96 / 66.39	1.44	90	44.40	0.064	0.855
	MC-dropout	72.96 / 65.64	1.43	90	45.86	**0.012**	0.856
	OGN	85.33 / 65.76	1.60	90	63.13	0.128	0.854
	VOGN	73.87 / **67.38**	**1.37**	90	76.04	0.029	0.854
	K-FAC	83.73 / 66.58	1.493	60	133.69	0.158	0.842
	Noisy K-FAC	72.28 / 66.44	1.44	60	179.27	0.080	0.852

better, while for the second, a higher number is better. See Section 6.12 for an explanation of these metrics. Results are summarised in Table 6.1. VOGN's uncertainty performance is more consistent and marginally better than the other methods, as expected from a more principled Bayesian method. Out of the 15 metrics (NLL, ECE and AUROC on 5 dataset/architecture combinations), VOGN performs the best or tied best on 10, and is second-best on the other 5. In contrast, both MC-dropout's and Adam's performance varies significantly, sometimes performing poorly, sometimes performing decently. MC-dropout is best on 4, and Adam is best on 1 (on LeNet-5; as argued earlier, the small architecture may result in underfitting). We also show calibration curves [DeGroot and Fienberg, 1983] in Figures 6.1 and 6.16. Adam is consistently over-confident, with its calibration curve below the diagonal. Conversely, MC-dropout is usually under-confident. On ImageNet, MC-dropout performs well on ECE (all methods are very similar on AUROC), but this required an excessively tuned dropout rate (see Section 6.11).

We also compare performance on out-of-distribution datasets. When testing on datasets that are different from the training datasets, predictions should be more uncertain. We use experimental protocol from the literature [Hendrycks and Gimpel, 2017, Lee et al., 2018, DeVries and Taylor, 2018, Liang et al., 2018] to compare VOGN, Adam and MC-dropout on CIFAR-10. We also borrow metrics from other works [Hendrycks and Gimpel, 2017, Lakshminarayanan et al., 2017], showing predictive entropy histograms and also reporting AUROC and FPR at 95% TPR. See Section 6.13 for further details on the datasets and metrics. Ideally, we want predictive entropy to be high on out-of-distribution data and low on in-distribution data. Our results are summarised in Figure 6.4 and Section 6.13. On ResNet-18 and AlexNet, VOGN's predictive entropy histograms show the desired behaviour: a spread of entropies for the in-distribution data, and high entropies for out-of-distribution data. Adam has many predictive entropies at zero, indicating Adam tends to classify out-of-distribution data too confidently. Conversely, MC-dropout's predictive entropies are generally high (particularly in-distribution), indicating MC-dropout has too much noise. On LeNet-5, we observe the same result as before: Adam and MC-dropout both perform well. The metrics (AUROC and FPR at 95% TPR) do not provide a clear story across architectures.

6.4.2.1 Performance on a Continual-learning task

The goal of continual learning is to avoid forgetting of old tasks while sequentially observing new tasks. The past tasks are never visited again, making it difficult to remember them. The field of continual learning has recently grown, with many approaches proposed to tackle this problem [Kirkpatrick et al., 2017, Lopez-Paz and Ranzato, 2017, Nguyen et al., 2018, Rusu et al., 2016, Schwarz et al., 2018]. Most approaches consider a simple setting where the tasks (such as classifying a subset of classes) arrive sequentially, and all the data from that task is available. We consider the same setup in our experiments.

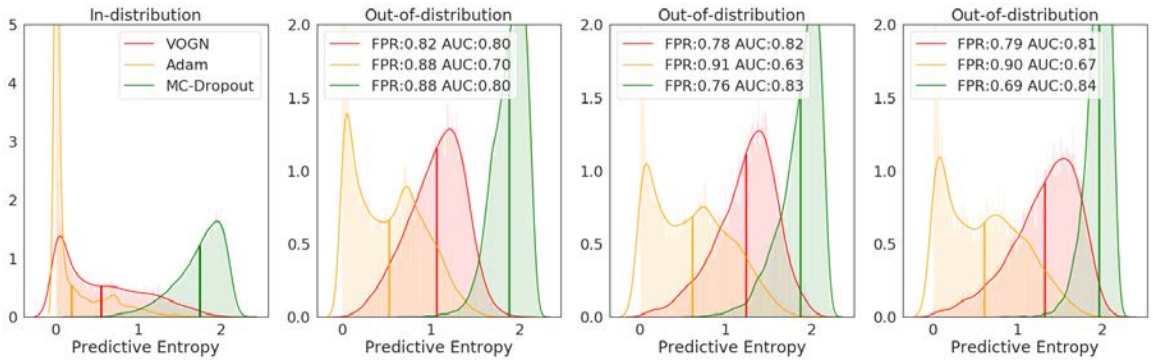


Figure 6.4: Histograms of predictive entropy for out-of-distribution tests for ResNet-18 trained on CIFAR-10. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the FPR at 95% TPR metric (lower is better) and the AUROC metric (higher is better), averaged over 3 runs. We clearly see that VOGN's predictive entropy is generally low for in-distribution and high for out-of-distribution data, but this is not the case for other methods. Solid vertical lines indicate the mean predictive entropy. The standard deviations are small and therefore not reported.

We compare to *Elastic Weight Consolidation* (EWC) [Kirkpatrick et al., 2017] and a VI-based approach called *Variational Continual Learning* (VCL) [Nguyen et al., 2018]. VCL employs BBB for each task, and we expect to boost its performance by replacing BBB by VOGN. Figure 6.2b shows results on a common benchmark called Permuted MNIST. We use the same experimental setup as in Swaroop et al. [2019]. In Permuted MNIST, each task consists of the entire MNIST dataset (10-way classification) with a different fixed random permutation applied to the input images' pixels. We run each method 20 times, with different random seeds for both the benchmark's permutations and model training. See Section 6.8.2 for hyperparameter settings and further details. We see that VOGN performs at least as well as VCL, and far better than a popular approach called EWC [Kirkpatrick et al., 2017]. Additionally, as found in the batch learning setting, VOGN is much quicker than BBB: we run VOGN for only 100 epochs per task, whereas VCL requires 800 epochs per task to achieve best results [Swaroop et al., 2019].

6.5 Noisy K-FAC algorithm

Noisy K-FAC [Zhang et al., 2018a] attempts to approximate the structure of the full covariance matrix, and therefore the updates are a bit more involved than VOGN (see Equation 6.4). Assuming a fully-connected layer, we denote the weight matrix of layer by $\boldsymbol{W}$. The Noisy K-FAC method estimates the parameters of a matrix-variate Gaussian distribution $q_t(\boldsymbol{W}) = \mathcal{MN}(\boldsymbol{W}|\boldsymbol{M}^{(t)}, \boldsymbol{\Sigma}_2^{(t)} \otimes \boldsymbol{\Sigma}_1^{(t)})$ by using the following updates:

$$\boldsymbol{M}^{(t+1)} \leftarrow \boldsymbol{M}^{(t)} - \alpha_t \left[\boldsymbol{A}_\gamma^{(t+1)}\right]^{-1} \left(\nabla_W E\left[\ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}^{(t)}\right] + \tilde{\delta}\boldsymbol{W}^{(t)}\right) \left[\boldsymbol{B}_\gamma^{(t+1)}\right]^{-1}, \tag{6.5}$$

$$\boldsymbol{A}^{(t+1)} \leftarrow (1-\tilde{\beta}_t)\boldsymbol{A}^{(t)} + \tilde{\beta}_t E\left[\boldsymbol{h}^{(t)}\boldsymbol{h}^{(t)\top}\right], \quad \boldsymbol{B}^{(t+1)} \leftarrow (1-\tilde{\beta}_t)\boldsymbol{B}^{(t)} + \tilde{\beta}_t E\left[\boldsymbol{g}^{(t)}\boldsymbol{g}^{(t)\top}\right], \tag{6.6}$$

where $\boldsymbol{W}^{(t)} \sim q_t(\boldsymbol{W})$, $\boldsymbol{g}^{(t)} := \nabla_{\boldsymbol{u}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}^{(t)})$ with $\boldsymbol{u} = \boldsymbol{u}^{(t)} := \boldsymbol{W}^{(t)\top} \boldsymbol{h}^{(t)}$, $\boldsymbol{h}^{(t)}$ is the input vector (the activation of the previous layer), $E[\cdot]$ is the average over the minibatch. $\tilde{\beta} := \beta\tau/N$, and $\gamma := \tilde{\gamma} + \gamma_{ex}$ with some *external* damping factor γ_{ex}. The covariance parameters are set to $\boldsymbol{\Sigma}_2^{(t)^{-1}} := \tau \boldsymbol{A}_\gamma^{(t)}/N$ and $\boldsymbol{\Sigma}_1^{(t)^{-1}} := \boldsymbol{B}_\gamma^{(t)}$, where $\boldsymbol{A}_\gamma^{(t)} := \boldsymbol{A}^{(t)} + \pi_t \sqrt{\gamma} \boldsymbol{I}$ and $\boldsymbol{B}_\gamma^{(t)} := \boldsymbol{B}^{(t)} + \frac{1}{\pi_t} \sqrt{\gamma} \boldsymbol{I}$. $\pi_t^2 (\pi_t > 0)$ is the average eigenvalue of $\boldsymbol{A}^{(t)}$ divided by that of $\boldsymbol{B}^{(t)}$. Similarly to the VOGN update in Equation 6.4, the gradients are scaled by matrices $\boldsymbol{A}^{(t)}$ and $\boldsymbol{B}^{(t)}$, which are related to the precision matrix of the approximation.

6.6 Details on fast implementation of the Gauss-Newton approximation

Current deep learning codebases are only optimized to directly return the average of gradients over the minibatch. In order to efficiently compute the Gauss-Newton (GN) approximation, we modify the backward-pass to efficiently calculate the gradient per example in the minibatch, and extend the solution in Goodfellow [2015] to both convolutional and Batch Normalization layers.

6.6.1 Convolutional layer

Consider a convolutional layer with a weight matrix $\boldsymbol{W} \in \mathbb{R}^{C_{out} \times C_{in} F^2}$ (ignore bias for simplicity) and an input tensor $\mathsf{H} \in \mathbb{R}^{C_{in} \times H_{in} \times W_{in}}$, where C_{out}, C_{in} are the number of output, input channels, respectively, H_{in}, W_{in} are the spatial dimensions, and F is the filter size. For any stride and padding, by applying `torch.nn.functional.unfold` function in PyTorch[5], we get the extended matrix $\boldsymbol{M}_{\mathsf{H}} \in \mathbb{R}^{C_{in} F^2 \times H_{out} W_{out}}$ so that the output tensor $\boldsymbol{S}$ is calculated by a matrix multiplication:

$$\boldsymbol{M}_{\mathsf{H}} \leftarrow \text{unfold}(\mathsf{H}) \in \mathbb{R}^{C_{in} F^2 \times H_{out} W_{out}}, \tag{6.7}$$

$$\boldsymbol{M}_{\mathsf{U}} \leftarrow \boldsymbol{W} \boldsymbol{M}_{\mathsf{H}} \in \mathbb{R}^{C_{out} \times H_{out} W_{out}}, \tag{6.8}$$

$$\mathsf{U} \leftarrow \text{reshape}(\boldsymbol{M}_{\mathsf{U}}) \in \mathbb{R}^{C_{out} \times H_{out} \times W_{out}}, \tag{6.9}$$

where H_{out}, W_{out} are the spatial dimensions of the output feature map. Using the matrix $\boldsymbol{M}_{\mathsf{H}}$, we can also get the gradient per example by a matrix multiplication:

$$\nabla_{\boldsymbol{M}_{\mathsf{U}}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \leftarrow \text{reshape}(\nabla_{\mathsf{U}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta})) \in \mathbb{R}^{C_{out} \times H_{out} W_{out}}, \tag{6.10}$$

$$\nabla_{\boldsymbol{W}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \leftarrow \nabla_{\boldsymbol{M}_{\mathsf{U}}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \boldsymbol{M}_{\mathsf{H}}^\top \in \mathbb{R}^{C_{out} \times C_{in} F^2}. \tag{6.11}$$

Note that in PyTorch, we can access to the inputs H and the gradient $\nabla_{\mathsf{U}} \ell(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{\theta})$ per example in the computational graph during a forward-pass and a backward-pass, respectively, by using the `Function Hooks` [6]. Hence, to get the gradient $\nabla_{\boldsymbol{W}} \ell(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{\theta})$ per example, we only need to perform (6.7), (6.10), and (6.11) after the backward-pass for this layer.

[5] `https://pytorch.org/docs/stable/nn.functional.html#torch.nn.functional.unfold`

[6] `https://pytorch.org/tutorials/beginner/former_torchies/nnft_tutorial.html#forward-and-backward-function-hooks`

6.6.2 Batch Normalization layer

Consider a Batch Normalization layer follows a fully-connected layer, which activation is $\boldsymbol{h} \in \mathbb{R}^d$, with the scale parameter $\boldsymbol{\gamma} \in \mathbb{R}^d$ and the shift parameter $\boldsymbol{\beta} \in \mathbb{R}^d$, we get the output of this layer $\boldsymbol{s} \in \mathbb{R}^d$ by,

$$\boldsymbol{\mu} \leftarrow E\left[\boldsymbol{h}\right] \in \mathbb{R}^d\,, \tag{6.12}$$

$$\boldsymbol{\sigma}^2 \leftarrow E\left[\left(\boldsymbol{h} - \boldsymbol{\mu}\right)^2\right] \in \mathbb{R}^d\,, \tag{6.13}$$

$$\hat{\boldsymbol{a}} \leftarrow \frac{\boldsymbol{h} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2}} \in \mathbb{R}^d\,, \tag{6.14}$$

$$\boldsymbol{u} \leftarrow \boldsymbol{\gamma}\hat{\boldsymbol{h}} + \boldsymbol{\beta} \in \mathbb{R}^d\,, \tag{6.15}$$

where $E\left[\cdot\right]$ is the average over the minibatch and $\hat{\boldsymbol{h}}$ is the normalized input. We can find the gradient with respect to parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ per example by,

$$\nabla_\gamma \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \leftarrow \nabla_{\boldsymbol{u}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \circ \hat{\boldsymbol{h}}\,, \tag{6.16}$$

$$\nabla_\beta \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta}) \leftarrow \nabla_{\boldsymbol{u}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta})\,. \tag{6.17}$$

We can obtain the input $\boldsymbol{h}$ and the gradient $\nabla_{\boldsymbol{u}} \ell(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{\theta})$ per example from the computational graph in `PyTorch` in the same way as a convolutional layer.

6.6.3 Layer-wise block-diagonal Gauss-Newton approximation

Despite using the method above, it is still intractable to compute the Gauss-Newton matrix $\boldsymbol{C}$ (and its inverse) with respect to the weights of large-scale deep neural networks. We therefore apply two further approximations. First, we view the $\boldsymbol{C}$ as a layer-wise block-diagonal matrix. This corresponds to ignoring the correlation between the weights of different layers. Hence for a network with L layers, there are L diagonal blocks, and $\boldsymbol{C}_l$ is the diagonal block corresponding to the l-th layer ($l = 1, \ldots, L$). Second, we approximate each diagonal block $\boldsymbol{C}_l$ with $\tilde{\boldsymbol{C}}_l$, which is either a Kronecker-factored or diagonal matrix. Using a Kronecker-factored matrix as $\tilde{\boldsymbol{C}}_l$ corresponds to K-FAC; a diagonal matrix corresponds to a mean-field approximation in that layer. By applying these two approximations, the update rule of the Gauss-Newton method can be written in a layer-wise fashion:

$$\boldsymbol{\theta}_l^{(t+1)} = \boldsymbol{\theta}_l^{(t)} - \alpha_t \tilde{\boldsymbol{C}}_l(\boldsymbol{\theta}^{(t)})^{-1} \nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\boldsymbol{\theta}^{(t)}) \;\; (l = 1, \ldots, L)\,, \tag{6.18}$$

where $\boldsymbol{\theta}_l \in \mathbb{R}^{P_l}$ is the parameters in l-th layer,

$$\boldsymbol{\theta} = \left[\begin{array}{ccc} \boldsymbol{\theta}_1^\top & \cdots & \boldsymbol{\theta}_L^\top \end{array}\right]^\top \in \mathbb{R}^P\,, \tag{6.19}$$

and $P = \sum_l P_l$. Since the cost of computing $\tilde{\boldsymbol{C}}_l^{-1}$ is much cheaper compared to that of computing $\boldsymbol{C}^{-1}$, our approximations make Gauss-Newton much more practical in deep learning.

Table 6.2: The approximation used for each layer type's diagonal block $\tilde{C}_l$ for the different optimizers tested this paper.

optimizer	convolution	fully-connected	BatchNorm
OGN	diagonal	diagonal	diagonal
VOGN	diagonal	diagonal	diagonal
K-FAC	Kronecker-factored	Kronecker-factored	diagonal
Noisy K-FAC	Kronecker-factored	Kronecker-factored	diagonal

In the distributed setting, each parallel process (corresponding to 1 GPU) calculates the GN matrix for its local minibatch. Then, one GPU adds them together and calculates the inverse. This inversion step can also be parallelized after making the block-diagonal approximation to the GN matrix. After inverting the GN matrix, the standard deviation $\boldsymbol{\sigma}$ is updated (line 9 in Algorithm 6), and sent to each parallel process, allowing each process to draw independently from the posterior.

In the Noisy K-FAC case, we use the same distributed K-FAC strategy as proposed in Chapter 5. When using K-FAC approximations to the Gauss-Newton blocks for other layers, Osawa et al. [2020] empirically showed that the BatchNorm layer can be approximated with a diagonal matrix without loss of accuracy, and we find the same. We therefore use diagonal $\tilde{C}_\ell$ with K-FAC and Noisy K-FAC in BatchNorm layers (see Table 6.2). For further details on how to efficiently parallelise K-FAC in the distributed setting, please see Osawa et al. [2019b, 2020] or Chapter 5.

6.7 OGN: A deterministic version of VOGN

To easily apply the tricks and techniques of deep-learning methods, we recommend to first test them on a deterministic version of VOGN, which we call the online Gauss-Newton (OGN) method. In this method, we approximate the gradients at the mean of the Gaussian, rather than using MC samples[7]. This results in an update without any sampling, as shown below (we have replaced $\boldsymbol{\mu}^{(t)}$ by $\boldsymbol{\theta}^{(t)}$ since there is no distinction between them):

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}_t - \alpha_t \frac{\hat{\boldsymbol{g}}(\boldsymbol{\theta}^{(t)}) + \tilde{\delta}\boldsymbol{\theta}^{(t)}}{\boldsymbol{s}^{(t+1)} + \tilde{\delta}}\,, \tag{6.20}$$

$$\boldsymbol{s}^{(t+1)} \leftarrow (1 - \tau\beta_t)\boldsymbol{s}^{(t)} + \beta_t \frac{1}{|\mathcal{B}_t|} \sum_{n\in\mathcal{B}_t} \left(\boldsymbol{g}_n(\boldsymbol{\theta}^{(t)})\right)^2 . \tag{6.21}$$

At each iteration, we still get a Gaussian $\mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\theta}^{(t)}, \hat{\boldsymbol{\Sigma}}^{(t)})$ with $\hat{\boldsymbol{\Sigma}}^{(t)} := \mathrm{diag}(1/(N(\boldsymbol{s}^{(t)} + \tilde{\delta})))$. It is easy to see that, like SG methods, this algorithm converges to a local minima of the loss function, thereby obtaining a Laplace approximation instead of a variational approximation. The advantage of OGN is that this can be used as a stepping stone, when switching from Adam to VOGN. Since it does not involve sampling, the tricks and techniques applied to Adam are easier to apply to OGN

[7]This gradient approximation used here is referred to as the *zeroth-order delta approximation* where $\mathbb{E}_q[\hat{\boldsymbol{g}}(\boldsymbol{\theta})] \approx \hat{\boldsymbol{g}}(\boldsymbol{\mu})$ (see Appendix A.6 in Khan [2012] for details).

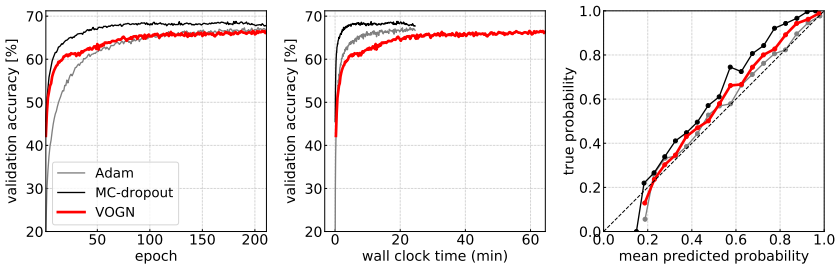


Figure 6.5: LeNet-5 on CIFAR-10 (no DA)

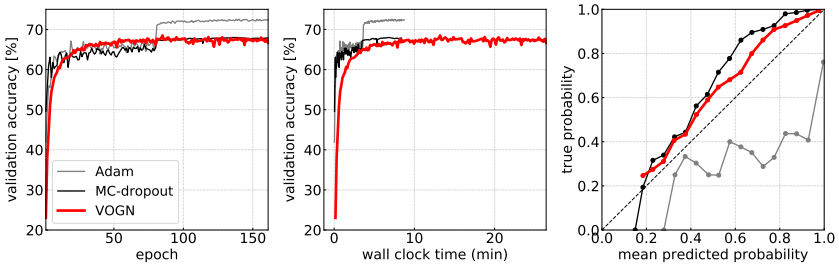


Figure 6.6: AlexNet on CIFAR-10 (no DA)

than VOGN. However, due to the lack of averaging over samples, this algorithm is less effective to preserve the benefits of Bayesian principles, and gives slightly worse uncertainty estimates.

6.8 Hyperparameter settings

Hyperparameters for all results shown in Table 6.1 are given in Table 6.4. The settings for distributed VI training are given in Table 6.5. Please see Goyal et al. [2017], Osawa et al. [2020], and Chapter 5 for best practice on these hyperparameter values.

6.8.1 Bayes by Backprop for CIFAR-10/LeNet-5 training

We use hyperparameter settings and training procedure for Bayes by Backprop (BBB) [Blundell et al., 2015] as suggested by Swaroop et al. [2019]. This includes using the local reparameterization

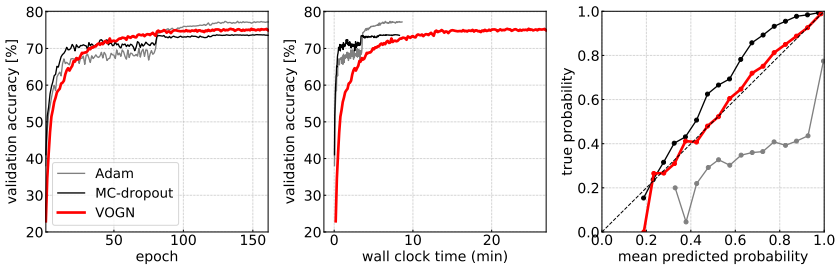


Figure 6.7: AlexNet on CIFAR-10

Table 6.3: Comparing optimizers on different dataset/architecture combinations, means and standard deviations over three runs. DA: Data Augmentation, Acc.: Accuracy (higher is better), NLL: Negative Log Likelihood (lower is better), ECE: Expected Calibration Error (lower is better), AUROC: Area Under ROC curve (higher is better), BBB: Bayes By Backprop. For ImageNet results, the reported accuracy and negative log likelihood are the median value from the final 5 epochs.

Dataset/ Architecture	optimizer	Train Acc (%)	Train NLL	Validation Acc (%)	Validation NLL	ECE	AUROC	Epochs	Time/ epoch (s)
CIFAR-10/ LeNet-5 (no DA)	Adam	71.98 ± 0.117	0.733 ± 0.021	**67.67 ± 0.513**	**0.937 ± 0.012**	**0.021 ± 0.002**	0.794 ± 0.001	210	6.96
	BBB	66.84 ± 0.003	0.957 ± 0.006	64.61 ± 0.331	1.018 ± 0.006	0.045 ± 0.005	0.784 ± 0.003	800	11.43
	MC-dropout	68.41 ± 0.581	0.870 ± 0.101	**67.65 ± 1.317**	0.99 ± 0.026	0.087 ± 0.009	**0.797 ± 0.006**	210	6.95
	VOGN	70.79 ± 0.763	0.880 ± 0.02	**67.32 ± 1.310**	**0.938 ± 0.024**	0.046 ± 0.002	**0.8 ± 0.002**	210	18.33
CIFAR-10/ AlexNet (no DA)	Adam	100.0 ± 0	0.001 ± 0	67.94 ± 0.537	2.83 ± 0.02	0.262 ± 0.005	0.793 ± 0.001	161	3.12
	MC-dropout	97.56 ± 0.278	0.058 ± 0.014	**72.20 ± 0.177**	1.077 ± 0.012	0.140 ± 0.004	**0.818 ± 0.002**	160	3.25
	VOGN	79.07 ± 0.248	0.696 ± 0.020	69.03 ± 0.419	**0.931 ± 0.017**	**0.024 ± 0.010**	0.796 ± 0	160	9.98
CIFAR-10/ AlexNet	Adam	97.92 ± 0.140	0.057 ± 0.006	73.59 ± 0.296	1.480 ± 0.015	0.262 ± 0.005	0.793 ± 0.001	161	3.08
	MC-dropout	80.65 ± 0.615	0.47 ± 0.052	**77.04 ± 0.343**	**0.667 ± 0.012**	0.114 ± 0.002	0.828 ± 0.002	160	3.20
	VOGN	81.15 ± 0.259	0.511 ± 0.039	75.48 ± 0.478	0.703 ± 0.006	**0.016 ± 0.001**	**0.832 ± 0.002**	160	10.02
CIFAR-10/ ResNet-18	Adam	97.74 ± 0.140	0.059 ± 0.012	**86.00 ± 0.257**	0.55 ± 0.01	0.082 ± 0.002	**0.877 ± 0.001**	160	11.97
	MC-dropout	88.23 ± 0.243	0.317 ± 0.045	82.85 ± 0.208	0.51 ± 0	0.166 ± 0.025	0.768 ± 0.004	161	12.51
	VOGN	91.62 ± 0.07	0.263 ± 0.051	84.27 ± 0.195	**0.477 ± 0.006**	**0.040 ± 0.002**	**0.876 ± 0.002**	161	53.14
ImageNet/ ResNet-18	SGD	82.63 ± 0.058	0.675 ± 0.017	**67.79 ± 0.017**	**1.38 ± 0**	0.067	0.856	90	44.13
	Adam	80.96 ± 0.098	0.723 ± 0.015	66.39 ± 0.168	1.44 ± 0.01	0.064	0.855	90	44.40
	MC-dropout	72.96	1.12	65.64	1.43	**0.012**	0.856	90	45.86
	OGN	85.33 ± 0.057	0.526 ± 0.005	65.76 ± 0.115	1.60 ± 0.00	0.128 ± 0.004	0.8543 ± 0.001	90	63.13
	VOGN	73.87 ± 0.061	1.02 ± 0.01	**67.38 ± 0.263**	**1.37 ± 0.01**	0.0293 ± 0.001	0.8543 ± 0	90	76.04
	K-FAC	83.73 ± 0.058	0.571 ± 0.016	66.58 ± 0.176	1.493 ± 0.006	0.158 ± 0.005	0.842 ± 0.005	60	133.69
	Noisy K-FAC	72.28	1.075	66.44	1.44	0.080	0.852	60	179.27

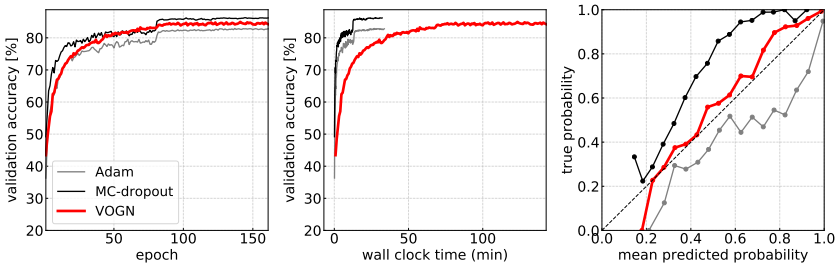


Figure 6.8: ResNet-18 on CIFAR-10

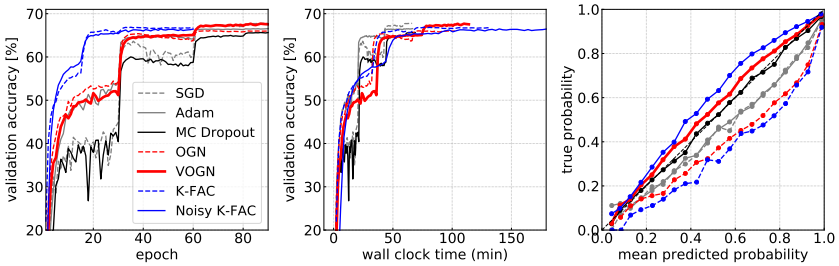


Figure 6.9: ResNet-18 on ImageNet

Table 6.4: Hyperparameters for all results in Table 6.1

Dataset/ Architecture	optimizer	α_{init}	α	Epochs to decay α	β_1	β_2	Weight decay	L2 reg
CIFAR-10/ LeNet-5 (no DA)	Adam	-	1e-3	-	0.1	0.001	1e-2	-
	BBB	-	1e-3	-	0.1	0.001	-	-
	MC-dropout	-	1e-3	-	0.9	-	-	1e-4
	VOGN	-	1e-2	-	0.9	0.999	-	-
CIFAR-10/ AlexNet (no DA)	Adam	-	1e-3	[80, 120]	0.1	0.001	1e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
CIFAR-10/ AlexNet	Adam	-	1e-3	[80, 120]	0.1	0.001	1e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
CIFAR-10/ ResNet-18	Adam	-	1e-3	[80, 120]	0.1	0.001	5e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
ImageNet/ ResNet-18	SGD	1.25e-2	1.6	[30, 60, 80]	0.9	-	-	1e-4
	Adam	1.25e-5	1.6e-3	[30, 60, 80]	0.1	0.001	1e-4	-
	MC-dropout	1.25e-2	1.6	[30, 60, 80]	0.9	-	-	1e-4
	OGN	1.25e-5	1.6e-3	[30, 60, 80]	0.9	0.9	-	1e-5
	VOGN	1.25e-5	1.6e-3	[30, 60, 80]	0.9	0.999	-	-
	K-FAC	1.25e-5	1.6e-3	[15, 30, 45]	0.9	0.9	-	1e-4
	Noisy K-FAC	1.25e-5	1.6e-3	[15, 30, 45]	0.9	0.9	-	-

Table 6.5: Settings for distributed VI training

optimizer	Dataset/ Architecture	M	# GPUs	K	τ	ρ	N_{orig}	δ	$\tilde{\delta}$	γ
VOGN	CIFAR-10/ LeNet-5 (no DA)	128	4	6	$0.1 \to 1$	1	50,000	100	2e-4 $\to$ 2e-3	1e-3
	CIFAR-10/ AlexNet (no DA)	128	8	3	$0.05 \to 1$	1	50,000	0.5	5e-7 $\to$ 1e-5	1e-3
	CIFAR-10/ AlexNet	128	8	3	$0.5 \to 1$	10	50,000	0.5	5e-7 $\to$ 1e-5	1e-3
	CIFAR-10/ ResNet-18	256	8	5	1	10	50,000	50	1e-3	1e-3
	ImageNet/ ResNet-18	4096	128	1	1	5	1,281,167	133.3	2e-5	1e-4
Noisy K-FAC	ImageNet/ ResNet-18	4096	128	1	1	5	1,281,167	133.3	2e-5	1e-4

trick, initializing means and variances at small values, using 10 MC samples per minibatch during training for linear layers (1 MC sample per minibatch for convolutional layers) and 100 MC samples per minibatch during testing for linear layers (10 MC samples per minibatch for convolutional layers). Note that BBB has twice as many parameters to optimize than Adam or SGD (means and variances for each parameter in the deep neural network). The fewer MC samples per minibatch for convolutional layers speed up training time per epoch while empirically not reducing convergence rate.

6.8.2 Continual learning experiment

Following the setup of Swaroop et al. [2019], all models are run with two hidden layers, of 100 hidden units each, with ReLU activation functions. VCL is run with the same hyperparameter settings as in Swaroop et al. [2019]. We perform a grid search over EWC's λ hyperparameter, finding that $\lambda = 100$ performs the best, exactly like in Nguyen et al. [2018].

VOGN is run for 100 epochs per task. Parameters are initialized before training with the default PyTorch initialization for linear layers. The initial precision is 1e6. A standard normal initial prior is used, just like in VCL. Between tasks, the mean and precision are initialized in the same way as for the first task. The learning rate $\alpha = 1e-3$, the batch size $M = 256$, $\beta_1 = 0$, $\beta_2 = 1e-3$, 10 MC samples are used during training and 100 for testing. We run each method 20 times, with different random seeds for both the benchmark's permutation and for model training.

6.9 Effect of prior variance and dataset size reweighting factor

We show the effect of changing the prior variance (δ^{-1} in Algorithm 6) in Figures 6.10 and 6.11. We can see that increasing the prior variance improves validation performance (accuracy and log likelihood). However, increasing prior variance also always increases the train-test gap, without

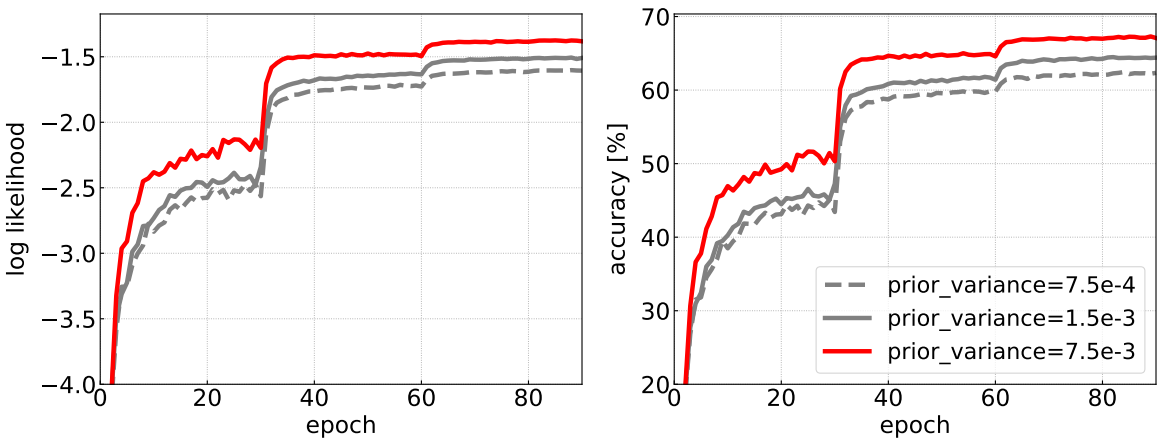


Figure 6.10: Effect of prior variance on VOGN training ResNet-18 on ImageNet.

exceptions, when the other hyperparameters are held constant. As an example, training VOGN on ResNet-18 on ImageNet with a prior variance of $7.5e-4$ has train-test accuracy and log likelihood gaps of 2.29 and 0.12 respectively. When the prior variance is increased to $7.5e-3$, the respective train-test gaps increase to 6.38 and 0.34 (validation accuracy and validation log likelihood also increase, see Figure 6.10).

With increased prior variance, VOGN (and Noisy K-FAC) reach converged solutions more like their non-Bayesian counterparts, where overfitting is an issue. This is as expected from Bayesian principles.

Figure 6.12 shows the combined effect of the dataset reweighting factor ρ and prior variance. When ρ is set to a value in the correct order of magnitude, it does not affect performance so much: instead, we should tune δ. This is our methodology when dealing with ρ. Note that we set ρ for ImageNet to be smaller than that for CIFAR-10 because the data augmentation cropping step uses a higher portion of the initial image than in CIFAR-10: we crop images of size 224x224 from images of size 256x256.

6.10 Effect of number of Monte Carlo samples on ImageNet

In the paper, we report results for training ResNet-18 on ImageNet using 128 GPUs, with 1 independent Monte-Carlo (MC) sample per process during training (`mc=128x1`), and 10 MC samples per validation image (`val_mc`= 10). We now show that increasing either of training or testing MC samples improves performance (validation accuracy and log likelihood) at the cost of increased computation time. See Figure 6.13.

Increasing the number of training MC samples per process reduces noise during training. This effect is observed when training on CIFAR-10, where multiple MC samples per process are required to stabilise training. On ImageNet, we have much larger minibatch size (4,096 instead of 256) and

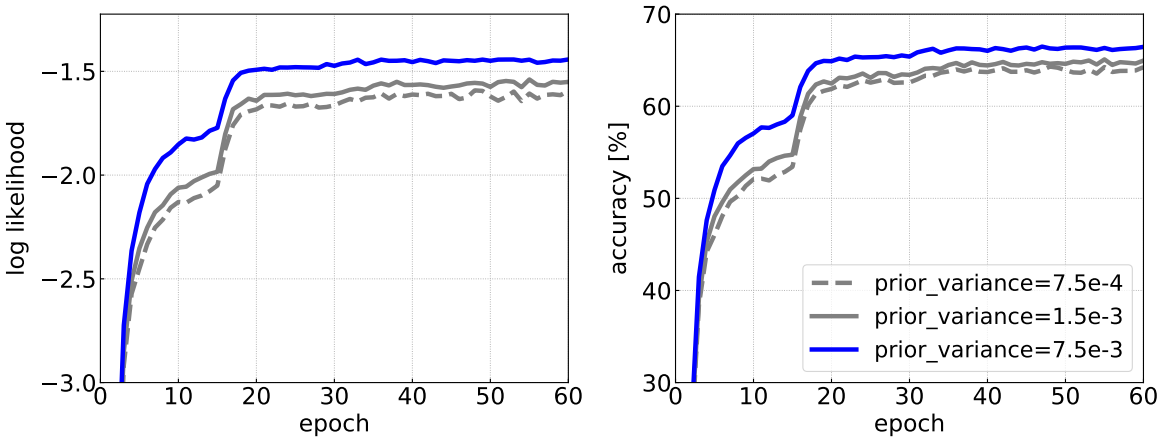


Figure 6.11: Effect of prior variance on Noisy K-FAC training ResNet-18 on ImageNet.

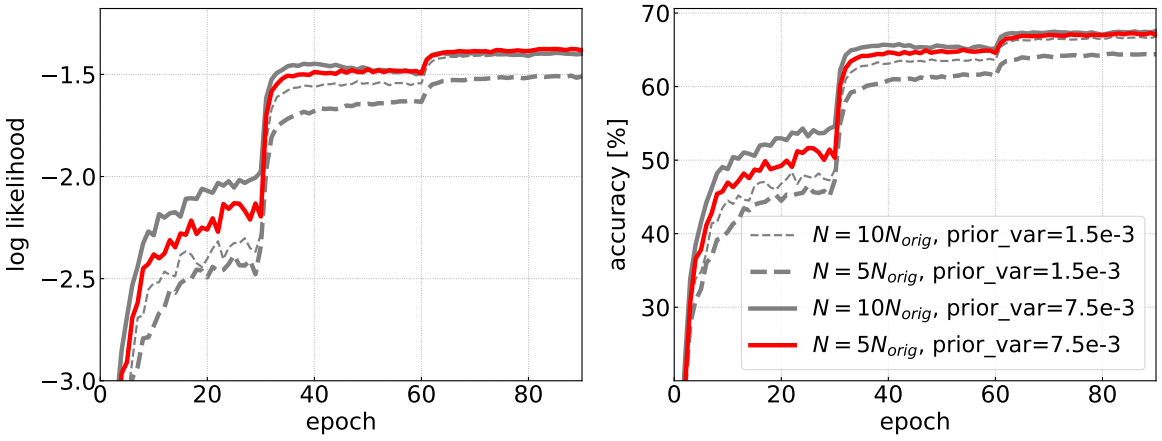


Figure 6.12: Effect of changing the dataset size reweighting factor ρ and prior variance on VOGN training ResNet-18 on ImageNet.

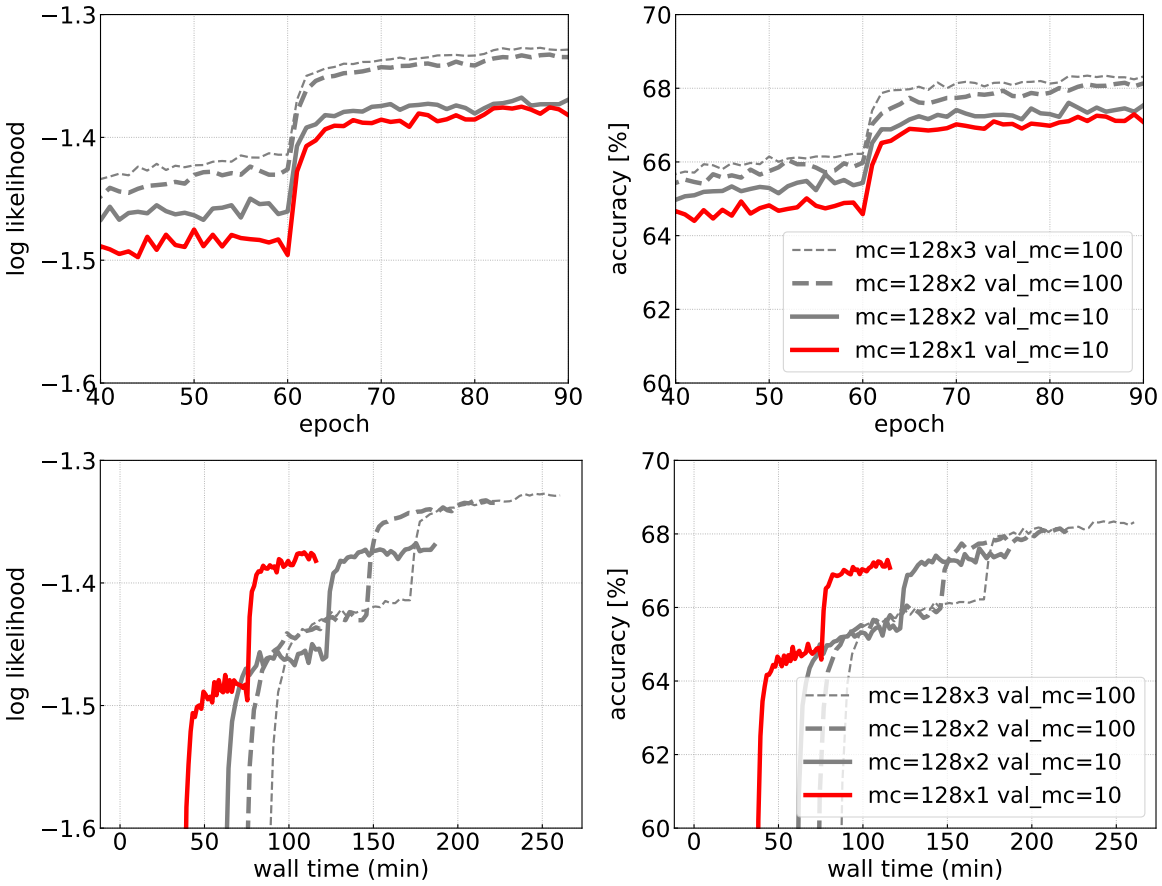


Figure 6.13: Effect of number of training and testing Monte Carlo samples on validation accuracy and log loss for VOGN on ResNet-18 on ImageNet.

more parallel processes (128 not 8), and so training with 1 MC sample per process is still stable. However, as shown in Figure 6.13, increasing the number of training MC samples per process to from 1 to 2 speeds up convergence per epoch, and reaches a better converged solution. The time per epoch (and hence total runtime) also increases by approximately a factor of 1.5. Increasing the number of train MC samples per process to 3 does not increase final test performance significantly.

Increasing the number of testing MC samples from 10 to 100 (on the same trained model) also results in better generalization: the train accuracy and log likelihood are unchanged, but the validation accuracy and log likelihood increase. However, as we run an entire validation on each epoch, increasing validation MC samples also increases run-time.

These results show that, if more compute is available to the user, they can improve VOGN's performance by improving the MC approximation at either (or both) train-time or test-time (up to a limit).

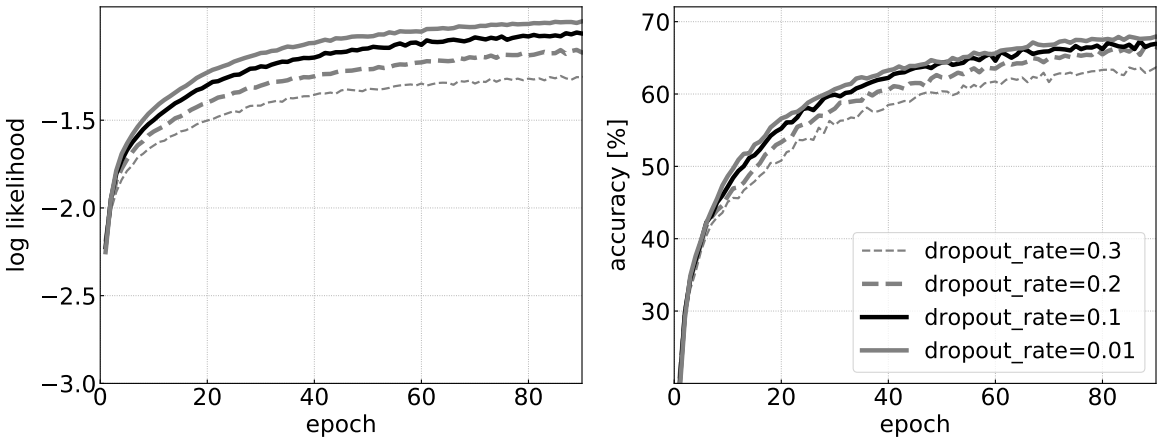


Figure 6.14: Effect of changing the dropout rate in MC-dropout, training LeNet-5 on CIFAR-10. When $p = 0.01$, the train-test gap on accuracy and log likelihood is very high (10.3% and 0.34 respectively). When $p = 0.1$, gaps are 1.4% and 0.04 respectively. When $p = 0.2$, the gaps are -7.71% and -0.02 respectively. We therefore choose $p = 0.1$ as it has high accuracy and log likelihood, and small train-test gap.

6.11 MC-dropout's sensitivity to dropout rate

We show MC-dropout's sensitivity to dropout rate, p, in this section. We tune MC-dropout as best as we can, finding that $p = 0.1$ works best for all architectures trained on CIFAR-10 (see Figure 6.14 for the dropout rate's sensitivity on LeNet-5 as an example). On ResNet-18 trained on ImageNet, we find that MC-dropout is extremely sensitive to dropout rate, with even $p = 0.1$ performing badly. We therefore use $p = 0.05$ for MC-dropout experiments on ImageNet. This high sensitivity to dropout rate is an issue with MC-dropout as a method.

6.12 Uncertainty metrics

We use several approaches to compare uncertainty estimates obtained by each optimizer. We follow the same methodology for all optimizers: first, tune hyperparameters to obtain good accuracy on the validation set. Then, test on uncertainty metrics. For multi-class classification problems, all of these are based on the predictive probabilities. For non-Bayesian approaches, we compute the probabilities for a validation input $\boldsymbol{x}_i$ as $\hat{p}_{ik} := p(y_i = k|\boldsymbol{x}_i, \boldsymbol{\theta}_*)$, where $\boldsymbol{\theta}_*$ is the weight vector of the DNN whose uncertainty we are estimating. For Bayesian methods, we can compute the predictive probabilities for each validation example $\boldsymbol{x}_n$ as follows:

$$\hat{p}_{nk} := \int p(y_n = k|\boldsymbol{x}_n, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{S}) d\boldsymbol{\theta} \approx \int p(y_n = k|\boldsymbol{x}_n, \boldsymbol{\theta}) q(\boldsymbol{\theta}) d\boldsymbol{\theta} \approx \frac{1}{m} \sum_{i=1}^{m} p(y_n = k|\boldsymbol{x}_n, \boldsymbol{\theta}^{(i)}),$$

where $\boldsymbol{\theta}^{(i)} \sim q(\boldsymbol{\theta})$ are samples from the Gaussian approximation returned by a variational method. We use 10 MC samples at validation-time for VOGN and MC-dropout (the effect of changing number

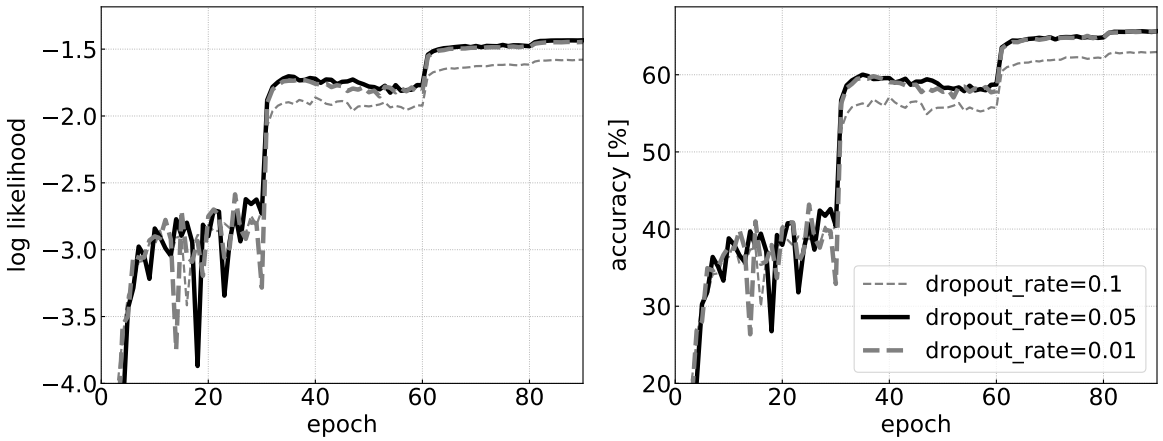


Figure 6.15: Effect of changing the dropout rate in MC-dropout, training Resnet-18 on ImageNet. We use $p = 0.05$ for our results.

of validation MC samples is shown in Section 6.10). This increases the computational cost during testing for these methods when compared to Adam or SGD.

Using the estimates $\hat{p}_{nk}$, we use three methods to compare uncertainties: validation log loss, AUROC and calibration curves. We also compare uncertainty performance by looking at model outputs when exposed to out-of-distribution data.

Validation log likelihood. Log likelihood (or log loss) is a common uncertainty metric. We consider a validation set of N_{Va} examples. For an input $\boldsymbol{x}_n$, denote the true label by $\boldsymbol{y}_n$, a 1-of-K encoded vector with 1 at the true label and 0 elsewhere. Denote the full vector of all validation outputs by $\boldsymbol{y}$. Similarly, denote the vector of all probabilities $\hat{p}_{nk}$ by $\boldsymbol{p}$, where $k \in \{1, ..., K\}$. The validation log likelihood is defined as $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{p}}) := \frac{1}{N_{Va}} \sum_{n=1}^{N_{Va}} \sum_{k=1}^{K} y_{nk} \log \hat{p}_{nk}$.

Tables 6.1 and 6.3 show final validation (negative) log likelihood. VOGN performs very well on this metric (aside from on CIFAR-10/AlexNet, with or without DA, where MC-dropout performs the best). All final validation log likelihoods are very similar, with VOGN usually performing similarly to the other best-performing optimizers (usually MC-dropout).

Area Under ROC curves (AUROC). We consider Receiver Operating Characteristic (ROC) curves for our multi-way classification tasks. A potential way that we may care about uncertainty measurements would be to discard uncertain examples by thresholding each validation input's predicted class' softmax output, marking them as too ambiguous to belong to a class. We can then consider the remaining validation inputs to either be correctly or incorrectly classified, and calculate the True Positive Rate (TPR) and False Positive Rate (FPR) accordingly. The ROC curve is summarised by its Area Under Curve (AUROC), reported in Table 6.1. This metric is useful to compare uncertainty performance in conjunction with the other metrics we use. The AUROC results are very similar between optimizers, particularly on ImageNet, although MC-dropout performs marginally better than the others, including VOGN. On all but one CIFAR-10 experiment (AlexNet, without

DA), VOGN performs the best, or tied best. Adam performs the worst, but is surprisingly good in CIFAR-10/ResNet-18.

Calibration Curves. Calibration curves [DeGroot and Fienberg, 1983] test how well-calibrated a model is by plotting true accuracy as a function of the model's predicted accuracy $\hat{p}_{nk}$ (we only consider the predicted class' $\hat{p}_{nk}$). Perfectly calibrated models would follow the $y = x$ diagonal line on a calibration curve. We approximate this curve by binning the model's predictions into $M = 20$ bins, as is often done. We show calibration curves in Figures 6.1 and 6.16. We can also consider the **Expected Calibration Error (ECE)** metric [Naeini et al., 2015, Guo et al., 2017], reported in Table 6.1. ECE calculates the expected error between the true accuracy and the model's predicted accuracy, averaged over all validation examples, again approximated by using M bins. Across all datasets and architectures, with the exception of LeNet-5 (which we have argued causes underfitting), VOGN usually has better calibration curves and better ECE than competing optimizers. Adam is consistently over-confident, with the calibration curve below the diagonal. Conversely, MC-dropout is usually under-confident, with too much noise, as mentioned earlier. The exception to this is on ImageNet, where MC-dropout performs well: we excessively tuned the MC-dropout rate to achieve this (see Section 6.11).

6.13 Out-of-distribution experimental setup and additional results

We use experiments from the out-of-distribution tests literature [Hendrycks and Gimpel, 2017, Lee et al., 2018, DeVries and Taylor, 2018, Liang et al., 2018], comparing VOGN to Adam and MC-dropout. Using trained architectures (LeNet-5, AlexNet and ResNet-18) on CIFAR-10, we test on SVHN, LSUN (crop) and LSUN (re-size) as out-of-distribution datasets, with the in-distribution data given by the validation set of CIFAR-10 (10,000 images). The entire training set of SVHN (73,257 examples, 10 classes) [Netzer et al., 2011] is used. The test set of LSUN (Large-scale Scene UNderstanding dataset [Yu et al., 2015], 10,000 images from 10 different scenes) is randomly cropped to obtain LSUN (crop), and is down-sampled to obtain LSUN (re-size). These out-of-distribution datasets have no similar classes to CIFAR-10.

Similar to the literature [Hendrycks and Gimpel, 2017, Lakshminarayanan et al., 2017], we use 3 metrics to test performance on out-of-distribution data. Firstly, we plot histograms of predictive entropy for the in-distribution and out-of-distribution datasets, seen in Figure 6.4, 6.17, 6.18 and 6.19. Predictive entropy is given by $\sum_{k=1}^{K} -\hat{p}_{nk} \log \hat{p}_{nk}$. Ideally, on out-of-distribution data, a model would have high predictive entropy, indicating it is unsure of which class the input image belongs to. In contrast, for in-distribution data, good models should have many examples with low entropy, as they should be confident of many input examples' (correct) class. We also compare AUROC and FPR at 95% TPR, also reported in the figures. By thresholding the most likely class' softmax

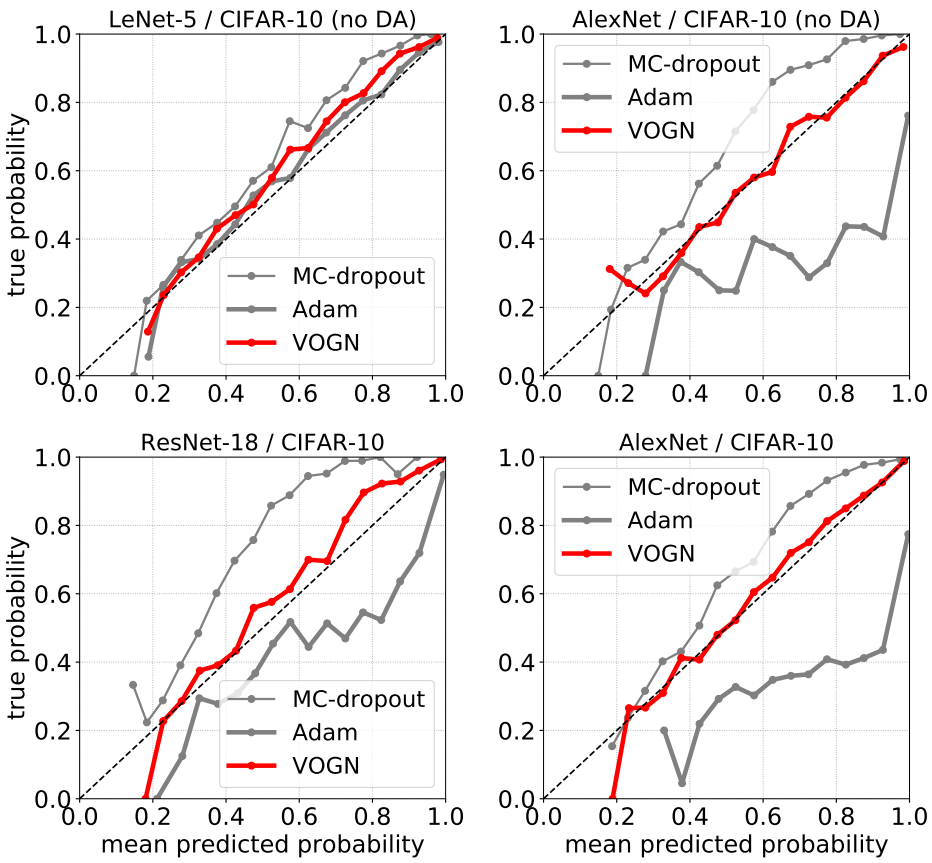


Figure 6.16: Calibration curves comparing VOGN, Adam and MC-dropout for final trained models trained on CIFAR-10. VOGN is extremely well-calibrated compared to the other two optimizers (except for LeNet-5, where all optimizers peform well). The calibration curve for ResNet-18 trained on ImageNet is in Figure 6.1.

output, we assign high uncertainty images to belong to an unknown class. This allows us to calculate the FPR and TPR, allowing the ROC curve to be plotted, and the AUROC to be calculated.

We show results on AlexNet in Figure 6.17 and 6.18 (trained on CIFAR-10 with DA and without DA respectively) and on LeNet-5 in Figure 6.19. Results on ResNet-18 is in Figure 6.4. These results are discussed in Section 6.4.2.

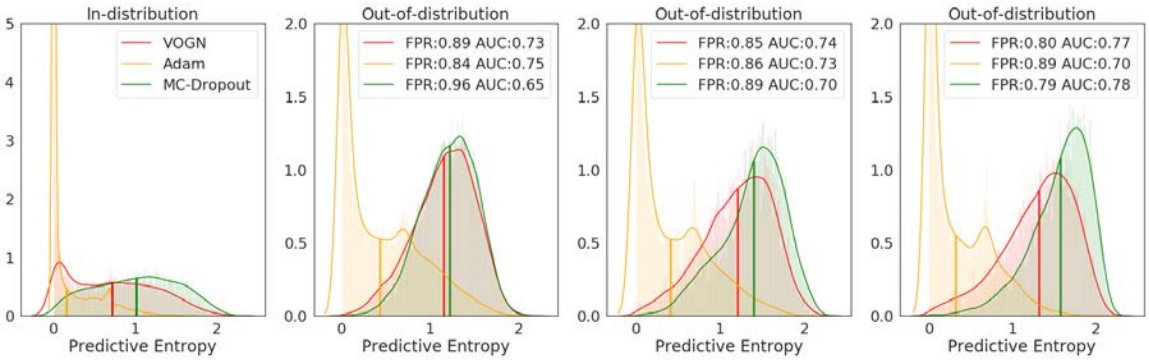


Figure 6.17: Histograms of predictive entropy for out-of-distribution tests for AlexNet trained on CIFAR-10 with data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

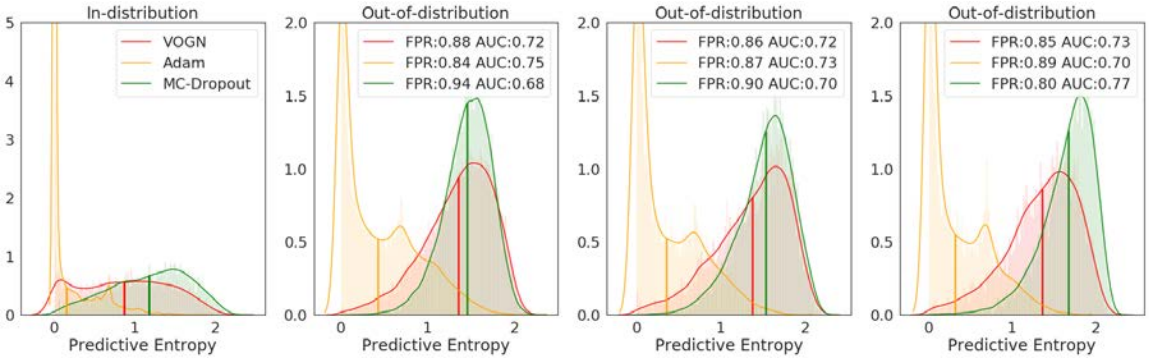


Figure 6.18: Histograms of predictive entropy for out-of-distribution tests for AlexNet trained on CIFAR-10 without data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

6.14 Discussion

We successfully train deep networks with a Natural Gradient Variational Inference method, VOGN, on a variety of architectures and datasets, even scaling up to ImageNet. This is made possible due to the similarity of VOGN to Adam, enabling us to boost performance by borrowing deep learning techniques. Our accuracies and convergence rates are comparable to SGD and Adam. Unlike them,

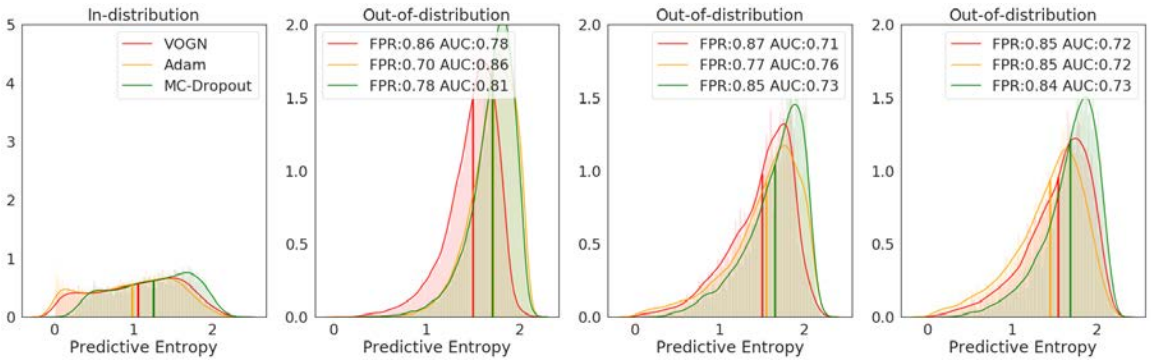


Figure 6.19: Histograms of predictive entropy for out-of-distribution tests for LeNet-5 trained on CIFAR-10 without data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

however, VOGN retains the benefits of Bayesian principles, with well-calibrated uncertainty and good performance on out-of-distribution data. Better uncertainty estimates open up a whole range of potential future experiments, for example, small data experiments, active learning, adversarial experiments, and sequential decision making. Our results on a continual-learning task confirm this. Another potential avenue for research is to consider structured covariance approximations.

Chapter 7

Information matrix in large-scale deep learning

The information matrices of neural networks, such as the Hessian matrix ($\boldsymbol{H}$), the Fisher information matrix ($\boldsymbol{F}$), and the covariance of the gradients ($\boldsymbol{C}$), play a key role in analysis and understanding of the training and generalization in deep learning. Recent developments in approximation methods and software libraries have made practical computation of these matrices possible. However, few empirical studies have exhaustively tested the characteristics and approximability of these matrices. The reason for this is that existing libraries do not widely support various types of matrices in both exact and approximate form. Moreover, the huge computational cost of preparing a large number of neural network models, especially in large-scale setups, prevents the realization of comprehensive comparisons and limits our understanding of these matrices.

Comprehensive empirical studies with various neural network architectures, data sets, and training settings (*e.g.*, mini-batch size, hyper-parameters) are becoming increasingly popular for analysis and understanding of deep learning [Shallue et al., 2019, McCandlish et al., 2018, Choi et al., 2019, Jiang et al., 2020, Metz et al., 2020]. We believe it would be of interest to the community if such comprehensive empirical studies were performed for the various information matrices and their approximations. Libraries such as BACKPACK [Dangel et al., 2020] and PyHessian [Yao et al., 2020] offer the capability to compute such matrices, but provide support for a limited set of matrix and neural network types. The present work extends those capabilities and performs a comprehensive empirical study for various networks, data sets, and training settings. The target tasks, deep networks, and matrices are described in Table 7.1.

7.1 Software libraries for information matrices analysis

. Table 7.2 lists existing software libraries for information matrix analysis in deep learning built on top of PyTorch [Paszke et al., 2019]. While most libraries only support the computation of

[1] BackPACK does not support $\boldsymbol{C}_{\text{kron}}$. Its (averaged) `SumGradSquared` extension corresponds to $\boldsymbol{C}_{\text{diag}}$.

Table 7.1: Models and matrices of deep neural networks for image classification tasks. $\triangle_k$ represents that only the top-k eigenvalues are stored. Matrix types and shapes are defined in chapter 3. $\boldsymbol{H}$ of shape blk-diag, kron, and diag are not evaluated because they are equivalent to those of $\boldsymbol{F}$ in case of piecewise linear activation functions (*e.g.*, ReLU) [Botev et al., 2017]. *F-MNIST*: Fashion-MNIST. *bn*: Batch Normalization. See section 7.3 for more information.

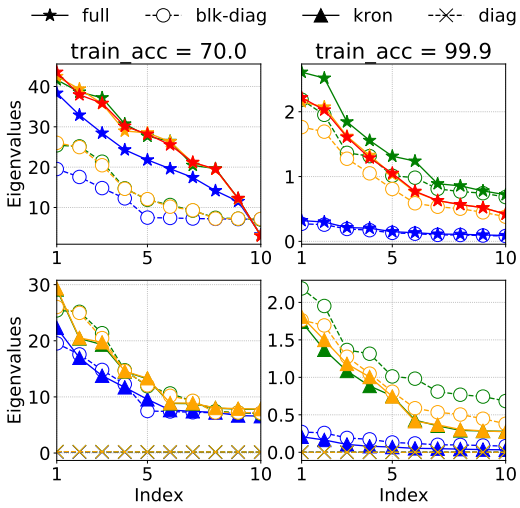
Data set	Networks	# params	Matrix types			Matrix shapes	
		P	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{\text{1mc}}/\boldsymbol{C}$	full/blk-diag	kron/diag
MNIST	Fully Connected	13K - 54M	✓	✓	✓	$\triangle_{15}$	✓
	Simple CNN	6K - 13M					
F-MNIST	Fully Connected	0.1M - 54M	✓	✓	✓	$\triangle_{15}$	✓
	Simple CNN	13M					
SVHN	Simple CNN	17M	✓	✓	✓	$\triangle_{15}$	✓
CIFAR-10	VGG-11 w/o bn	9.8M	✓	✓	✓	$\triangle_{15}$	✓
	ResNet-18	11M					
	WideResNet-50	67M					
CIFAR-100	VGG-11	9.8M	✓	-	✓	$\triangle_{15}$	✓
	ResNet-18	11M					
	WideResNet-50	67M					
ImageNet	VGG-11/19	133M / 143M	✓	-	✓	$\triangle_{1}$	✓
	ResNet-18/50	12M / 26M					
	EfficientNet-B0/3	5.3M / 12M					

the eigenvalues of the Hessian ($\boldsymbol{H}$), BACKPACK [Dangel et al., 2020] was the first to support the computation of the generalized Gauss-Newton approximation of $\boldsymbol{H}$, which is equivalent to the Fisher information matrix ($\boldsymbol{F}$), with user-friendly interfaces. The novelty of BACKPACK lies in its original back-propagation implementation exploiting an efficient Jacobian matrix ($\boldsymbol{J}$) computation which is useful for computing $\boldsymbol{F}$. However, BACKPACK limits the range of networks it supports and creates unnecessary computational overhead due to its original back-propagation.

In this work, we propose Automatic & Scalable Differentiation Library (ASDL), an extension library of PYTORCH, which supports computation of matrices of various types and shapes (described

Table 7.2: Software libraries for the information matrices analysis. $\triangle$ represents that only the eigenvalues computation is supported. *Res*: the library can be used for networks with residual connections [He et al., 2016]. *Dist*: the library supports distributed run with multiple CPU/GPUs.

Software library	Matrices supports						Res	Dist
	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{\text{mmc}}$	$\boldsymbol{C}$	shapes	$\boldsymbol{J}$		
GPyTorch [Gardner et al., 2018]	$\triangle$	-	-	-	full	-	✓	-
MLRG DC [Granziol et al., 2019]	$\triangle$	-	-	-	full	-	✓	-
PyHessian [Yao et al., 2020]	$\triangle$	-	-	-	full	-	✓	✓
BackPACK [Dangel et al., 2020]	-	✓	✓	✓[1]	kron, diag	✓	-	-
This work	✓	✓	✓	✓	**all shapes**	-	✓	✓



Operation	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{\text{1mc}}$	$\boldsymbol{C}$
$X_{\text{full}} \cdot v$	3.70s	114s	13.1s	12.9s
$X_{\text{blk-diag}} \cdot v$	-	114s	13.1s	12.9s
X_{kron}	-	24.7s (25.6s)	**4.97**s (25.6s)	**4.99**s
X_{diag}	-	26.2s (7.52s)	**4.74**s (7.54s)	**4.76**s (5.28s)
$X_{\text{kron}} \& X_{\text{diag}}$	-	31.9s (29.4s)	**5.83**s (29.4s)	**5.85**s
$\text{eig}(X_{\text{kron}})$	-		544 ms	
$\text{eig}(X_{\text{diag}})$	-		14 ms	

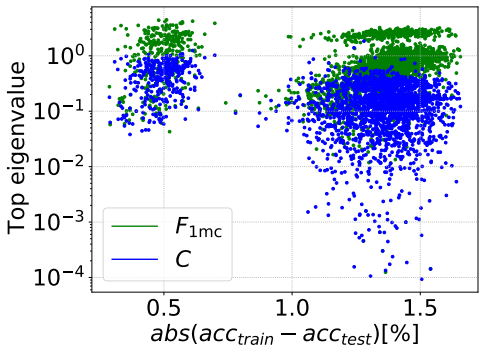
Figure 7.1: **Left**: Top-10 eigenvalues of $\boldsymbol{H}, \boldsymbol{F}, \boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$ of various shapes of Fully Connected at each phase (training accuracy (%)) of training on MNIST classification with batch size of 256. Note that each matrix is evaluated using the entire training set. **Right**: Averaged computation time (100 trials by a NVIDIA Tesla P100) by ASDL for the same model using the entire training set with batch size of 64. The times in parentheses () are measured using BACKPACK [Dangel et al., 2020]. $X \cdot v$: a matrix-vector product. X: computation of an approximate matrix. $\text{eig}(X)$: computation of sorted eigenvalues. ASDL can efficiently compute X_{kron} and X_{diag} at the same time. $\text{eig}(X_{\text{kron}}) = \text{eig}(\boldsymbol{B} \otimes \boldsymbol{A}) = \text{eig}(\boldsymbol{B}) \otimes \text{eig}(\boldsymbol{A})$, and $\text{eig}(X_{\text{diag}})$ is just sorting diagonal elements. Due the huge size of X_{full} and $X_{\text{blk-diag}}$, their eigenvalues are evaluated by the power method using $X \cdot v$ (about 10-20 iterations for each eigenvalue). Hence, the eigenvalues of $X_{\text{kron}}, X_{\text{diag}}$ are much easier to calculate than those of $X_{\text{full}}, X_{\text{blk-diag}}$.

in Chapter 3) and extends the standard automatic-differentiation library (ADL). ASDL support the widest range of matrix types and shapes. We found that, in practical deep learning setups, an estimation of $\boldsymbol{F}$ by Monte-Carlo samplings ($\boldsymbol{F}_{m\text{mc}}$) reproduces $\boldsymbol{F}$'s eigenspectrum well and can be computed efficiently without $\boldsymbol{J}$ computation (Figure 7.1). ASDL also supports networks with residual connections [He et al., 2016] and distributed GPUs run that are essential for analyses in practical setups.

7.2 Experiments

7.2.1 Matrices and eigenvalues computation

In Figure 7.1, we compare the eigenspectrum and computation time of the matrices of each type and shape for Fully Connected models trained on MNIST. Since the sizes of full and block-diagonal matrices are immense, it cannot be constructed on memory. Thus, we calculate their eigenvalues by the power method. The eigenvalues of block-diagonal, Kronecker-factored, and diagonal $\boldsymbol{H}$ are not evaluated as they are equivalent to those of $\boldsymbol{F}$ in case of piecewise linear activation functions (*e.g.*, ReLU) [Botev et al., 2017]. In the middle of the training (*i.e.*, training accuracy = 70%), the matrices of types other than $\boldsymbol{C}$ have almost identical eigenspectrum in each shape. At the end of the



Measure	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{1mc}$	$\boldsymbol{C}$
$\mathrm{Tr}(X_{\mathrm{full}})$	0.150	0.150	0.148	-0.068
$\mathrm{Tr}(X_{\mathrm{kron}})$	-	0.140	0.137	-0.082
$N_{\mathrm{eff}}(X_{\mathrm{kron}})$	-	0.159	0.156	-0.074
$N_{\mathrm{eff}}(X_{\mathrm{diag}})$	-	**0.181**	0.178	-0.057
$\lambda_{\max}(X_{\mathrm{full}})$	0.144	0.130	**0.193**	-0.112
$\lambda_{\max}(X_{\mathrm{blk\text{-}diag}})$	-	0.167	**0.213**	-0.100
$\lambda_{\max}(X_{\mathrm{kron}})$	-	0.147	0.178	-0.060
$\lambda_{\max}(X_{\mathrm{diag}})$	-	-0.083	-0.078	-0.262

Figure 7.2: **Left**: Correlation between $\lambda_{\max}(\boldsymbol{F}_{1mc,\mathrm{blk\text{-}diag}})$, $\lambda_{\max}(\boldsymbol{C}_{\mathrm{blk\text{-}diag}})$ and accuracy gap (train vs test) of 2,568 converged (training accuracy $\geq 99.9\%$) MNIST models (left clusters: Simple CNN, right clusters: Fully Connected). **Right**: Kendall's rank coefficient τ [Jiang et al., 2020] for each measure of the same models. A larger value indicates a stronger positive correlation. $\mathrm{Tr}(\cdot)$: trace. $N_{\mathrm{eff}}(\cdot)$: effective dimensions [Maddox et al., 2020]. $\lambda_{\max}(\cdot)$: top eigenvalue.

training (*i.e.*, training accuracy = 99.9%), the eigenvalue of $\boldsymbol{C}$ decays compared to the others. The Kronecker-factored matrices reproduce the eigenvalues of the block-diagonal matrices well. Also, $\boldsymbol{F}_{1mc}$ reproduces those of $\boldsymbol{F}$ well. Both of them can be calculated at a relatively small cost. The eigenvalues of the diagonal matrices are very small compared to the other shapes.

7.2.2 Matrix eigenvalues as generalization measures

We analyze the generalization performance of the models based on the trace, the top eigenvalue and the effective dimensions [Maddox et al., 2020]. To evaluate the quality of such measures, we used Kendall's rank coefficient [Jiang et al., 2020]: $\tau := \frac{2}{n(n-1)} \sum_{i<j} \mathrm{sign}(\mu_i - \mu_j)\mathrm{sign}(g_i - g_j)$, where μ_i is the calculated metric and g_i is the generalization gap (accuracy gap in our experiments) of the i-th model in the n models trained on the same task. A larger value indicates a stronger positive correlation. In Figure 7.2, we compare τ for the matrices of various types and shapes of 2,568 converged (training accuracy $\geq 99.9\%$) models trained on MNIST. The top eigenvalue of $\boldsymbol{F}_{1mc}$ shows the strongest correlation with the gap. On the other hand, the measures by the diagonal matrices and $\boldsymbol{C}$ are generally weakly correlated.

7.3 Training settings

In this section, we describe the detail training settings, *i.e.*, data sets, neural network architectures, search space for the grid search, and the steps taken to record checkpoints during training.

7.3.1 Grid search

MNIST [LeCun et al., 1998b] contains 60,000 training images of 10 classes. Each pixel in an image is normalized. We train the Fully Connected and Simple CNN, that are adopted by Shallue et al.

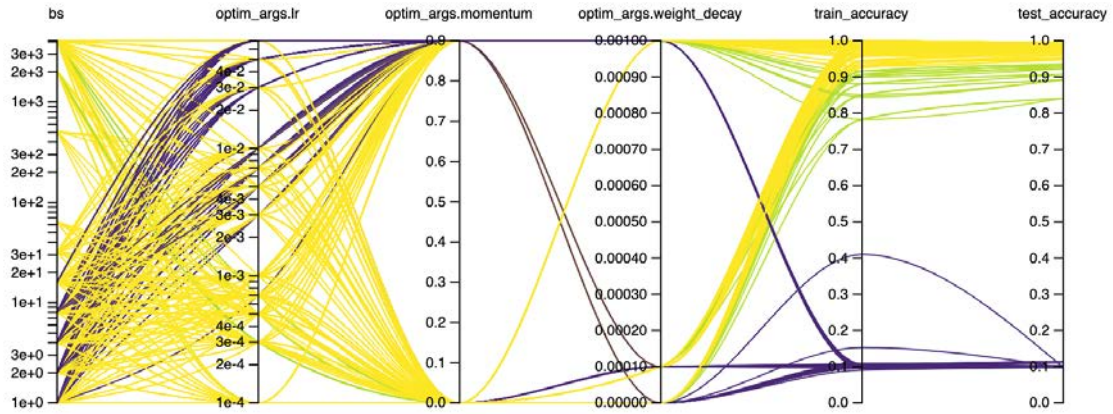


Figure 7.3: Hyper-parameters search space for training Simple CNN on MNIST with SGD/Momentum colored by the last column value (test accuracy). `optim_args` represent arguments to `torch.optim.SGD` of PyTorch [Paszke et al., 2019]. This plot has been exported from the public project of W&B [Biewald, 2020] for this work.

[2019], with the following hyper-parameters search space:

- mini-batch size: $\{2^0, 2^1, \ldots, 2^{12}\}$
- optimizer: SGD / Momentum (`torch.optim.SGD`)
- LR schedule: Constant
- `optim_args.lr` : $\{1, 3, 5\} \times \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$
- `optim_args.momentum` : $\{0, 0.9\}$
- `optim_args.weight_decay` : $\{0, 10^{-3}, 10^{-4}\}$

`optim_args` represent the arguments passed to a PyTorch [Paszke et al., 2019] optimizer class (*e.g.* `torch.optim.SGD`) . For Fully Connected, we scale the width by $\{0.125, 0.25, 0.5, 1, 2, 4\}$ and set the number of fully-connected layers 3 or 5. Figure 7.3 visualizes the hyper-parameters search space for Simple CNN.

Fashion-MNIST [Xiao et al., 2017] contains 60,000 training images of 10 classes. Each pixel in an image is normalized. We train the Fully Connected and Simple CNN with the same hyper-parameters as MNIST.

SVHN [Goodfellow et al., 2014] contains 73,257 training images of 10 classes. Each pixel in an image is normalized for each channel (RGB). We train Simple CNN with the following hyper-parameters search space:

- mini-batch size: $\{2^0, 2^1, \ldots, 2^{12}\}$
- LR schedule: Constant

- optimizer:
  - Momentum (`torch.optim.SGD`)
    - `optim_args.lr` : $\{1, 3\} \times \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$
    - `optim_args.momentum` : 0.9
    - `optim_args.weight_decay` : 10^{-3}
  - Adam [Kingma and Ba, 2015] (`torch.optim.Adam`)
    - `optim_args.lr` : $\{1, 3, 5\} \times \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$
    - `optim_args.weight_decay` : $\{0, 10^{-3}, 10^{-4}\}$

CIFAR-10 [Krizhevsky and Hinton, 2009] contains 50,000 training images of 10 classes. Each pixel in an image is normalized for each channel (RGB). To systematically create models with various learning speed, we adopt the Linear decay learning rate scheduling [Shallue et al., 2019]. The learning rate at t-th step ($\eta^{(t)}$) is given as:

$$\eta^{(t)} = \begin{cases} \eta^{(0)} + (1-\alpha)\eta^{(0)}\frac{t}{T} & \text{if } t \leq T\,, \\ \alpha\eta^{(0)} & \text{otherwise,} \end{cases} \tag{7.1}$$

where $\eta^{(0)}$ is `optim_args.lr` . We train VGG-11 [Simonyan and Zisserman, 2015] (without Batch Normalization [Ioffe and Szegedy, 2015]), ResNet-18 [He et al., 2016], and WideResNet-50 [Zagoruyko and Komodakis, 2016] with the following hyper-parameter search space:

- mini-batch size: $\{2^2, 2^3, \ldots, 2^{12}\}$
- optimizer: Momentum (`torch.optim.SGD`)
- LR schedule: Linear decay
  - α: 0.01
  - T: $\{300000, 400000, 500000, 600000\}$
- `optim_args.lr` : $\{1, 3, 5\} \times \{1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$
- `optim_args.momentum` : 0.9
- `optim_args.weight_decay` : $\{0, 10^{-3}, 10^{-4}\}$

CIFAR-100 [Krizhevsky and Hinton, 2009] contains 50,000 training images of 100 classes. Each pixel in an image is normalized for each channel (RGB). We train VGG-11 with Batch Normalization, ResNet-18, and WideResNet-50 with the same hyper-parameters search space as CIFAR-10 except:

- mini-batch size: $\{2^5, 2^6, \ldots, 2^{12}\}$

Also, we train models with and without the following data augmentation:

- `torchvision.transforms.RandomCrop` (`size`: 32, `padding`: 4) of TorchVision [2].
- `torchvision.transforms.RandomHorizontalFlip` (`p`: 0.5) of TorchVision.

ImageNet [Deng et al., 2009] contains 1,281,167 images of 1000 classes. Each pixel in an image is normalized for each channel (RGB). We train VGG-11/19 [Simonyan and Zisserman, 2015], ResNet-18/50 [He et al., 2016], and EfficientNet-B0/B3 [Tan and Le, 2019]. Due to the computational budget limitations, we do not perform a detail grid search. We only change the mini-batch size:

- mini-batch size (BS): $\{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$
- VGG-11/19:
  - optimizer: Nesterov (`torch.optim.SGD`, `nesterov=True`)
  - LR schedule: Linear decay
    - α: 0.003
    - T: $1.36 \cdot 10^5 \cdot 2^{10}/\mathrm{BS}$
  - `optim_args.lr`: $0.00125 \cdot \mathrm{BS}/2^{10}$
  - `optim_args.momentum`: 0.99
  - `optim_args.weight_decay`: 10^{-4}
- ResNet-18/50:
  - optimizer: Nesterov (`torch.optim.SGD`, `nesterov=True`)
  - LR schedule: Linear decay
    - α: 0.003
    - T: $1.04 \cdot 10^5 \cdot 2^{10}/\mathrm{BS}$
  - `optim_args.lr`: $0.025 \cdot \mathrm{BS}/2^{10}$
  - `optim_args.momentum`: 0.99
  - `optim_args.weight_decay`: 10^{-4}
- EfficientNet-B0/B3: We use the hyper-parameters described by [Tan and Le, 2019]. We apply the fixed data augmentation policies created by the Fast AutoAugment [Lim et al., 2019]. We adopt the PyTorch implementation [3] provided by the authors of the Fast AutoAugment.

Note that we do not apply weight decay to the parameters (γ, β) of Batch Normalization layers. For VGG-11/19 and ResNet-18/50, we apply the Gradual warmup of learning rate [Goyal et al., 2017] for the first 5 epochs of training. Also, we apply the following data augmentation:

[2] `https://github.com/pytorch/vision`
[3] `https://github.com/kakaobrain/fast-autoaugment`

Table 7.3: The range of mini-batch size, the values of training accuracy to save checkpoints, and the number of models for each data set and neural network. Once the moving average $m^{(t)}$ (7.2) of top1-accuracy reaches a checkpoint accuracy, we save the parameters/status of models and optimizer (*e.g.* the moving average of squared gradients for Adam [Kingma and Ba, 2015]).

Data set	Networks	Batch size	Checkpoint accuracy [%]	# models
MNIST	Fully Connected	$[2^0, 2^{12}]$	$70, 90, 95, 97, 99.3, 99.9$	6894
	Simple CNN			1038
F-MNIST	Fully Connected	$[2^0, 2^{12}]$	$70, 90, 95, 97, 99.3, 99.9$	6431
	Simple CNN			681
SVHN	Simple CNN	$[2^0, 2^{12}]$	$80, 90, 95, 97.5, 99.6$	731
CIFAR-10	VGG-11 w/o bn	$[2^2, 2^{12}]$	$50, 70, 90, 95, 97$	3558
	ResNet-18		$50, 70, 90, 95, 97, 98$	2878
	WideResNet-50			792
CIFAR-100	VGG-11	$[2^5, 2^{12}]$	$50, 70, 90, 95, 97$	2785
	ResNet-18			2147
	WideResNet-50			1832
ImageNet	VGG-11	$[2^{10}, 2^{14}]$	$20, 30, 40, 50, 55, 56, 57, 58, 59, 60$	8
	VGG-19		$20, 30, 40, 50, 60, 64, 65, 66, 67, 68$	9
	ResNet-18		$30, 40, 50, 60, 65, 66, 67, 68, 69, 70$	10
	ResNet-50		$30, 50, 70, 73, 75, 76, 77, 78, 79, 80$	18
	EfficientNet-B0		$30, 50, 70, 73, 75, 76, 77, 78, 79, 80$	7
	EfficientNet-B3		$30, 50, 70, 73, 75, 76, 77, 78, 79, 80$	9

- `torchvision.transforms.RandomResizedCrop` (`scale` : (0.08, 1.0), `ratio` : (3/4, 4/3)) of TorchVision.
- `torchvision.transforms.RandomHorizontalFlip` (`p` : 0.5) of TorchVision.

7.3.2 Checkpointing

We analyze information matrices at each phase of training for analyses and understanding of neural network training. Because the importance of the number of steps depends on several factors (*e.g.* batch size, learning rate, and model size), it is preferable to save models based on the metric to minimize/maximize (*e.g.* training accuracy) rather than the number of epochs or steps. Therefore, to obtain only desideratum information, we chose multiple checkpoints during training that depend on top-1 training accuracy for image classification problems. As we do a broad search of batch sizes (*e.g.* varying from as small as $2^0 = 1$ to as big as $2^{12} = 4096$ for MNIST, Fashion-MNIST, and SVHN), we cannot simply compare the results, as they are noisier for smaller batch sizes. Hence it is essential to take care of the criteria by taking the moving average. To smooth the training loss and accuracy, we applied the exponential smoothing algorithm:

$$m^{(t)} = (1 - \beta) \cdot m^{(t-1)} + \beta \cdot E^{(t)}\,, \tag{7.2}$$

where $0 < \beta < 1$ and $E^{(t)}$ is the loss / accuracy over the mini-batch at t-th step. The hyper-parameter β can affect the results greatly, hence we chose the parameter based on the noisiness of the performance, which tends to be proportional to the batch sizes. Therefore we chose the β as follows:

$$\beta = |\mathcal{B}| / \left(|\mathcal{S}_{\text{train}}| \cdot \alpha\right) , \tag{7.3}$$

where α is a smoothing factor, $\mathcal{B}$ is a mini-batch, and $\mathcal{S}_{\text{train}}$ is the training set. In our experiments, $\alpha = 0.5$ is used for CIFAR-10/100 and $\alpha = 0.1$ is used for the other data sets. Table 7.3 lists the values of checkpoint accuracy.

7.4 Additional experiments

7.4.1 Eigenvalue spectrum analyses

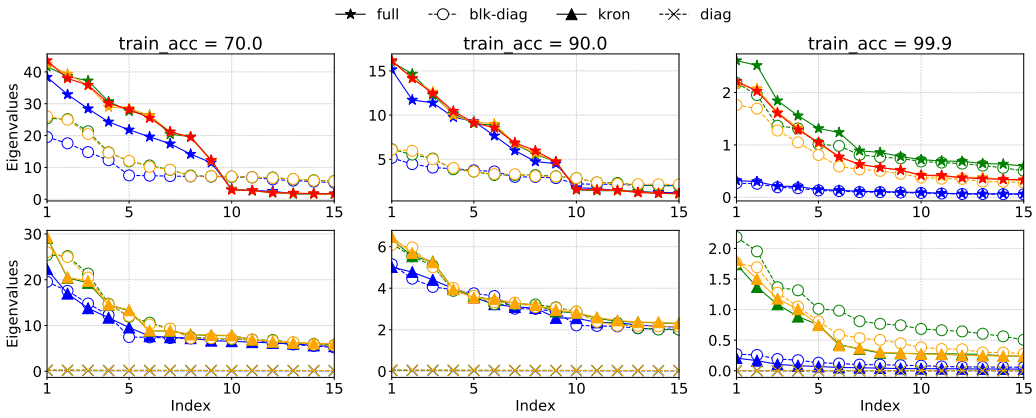
In Figure 7.4, we show the top eigenvalues of $\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$ for various data sets and networks at each phase of the training. Unlike models trained on MNIST, in case of VGG-11 on CIFAR-10/100, the Kronecker-factored matrices do not reproduce the eigenvalues of the block-diagonal matrices very well. The eigenvalues of VGG-11 with Batch Normalization on CIFAR-100 are small. This agrees with the analysis by Karakida et al. [2019]. For ResNet-18 on ImageNet, there is little difference in the eigenvalues of the Kronecker-factored and diagonal $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$.

7.4.2 Eigenvalues as generalization measures

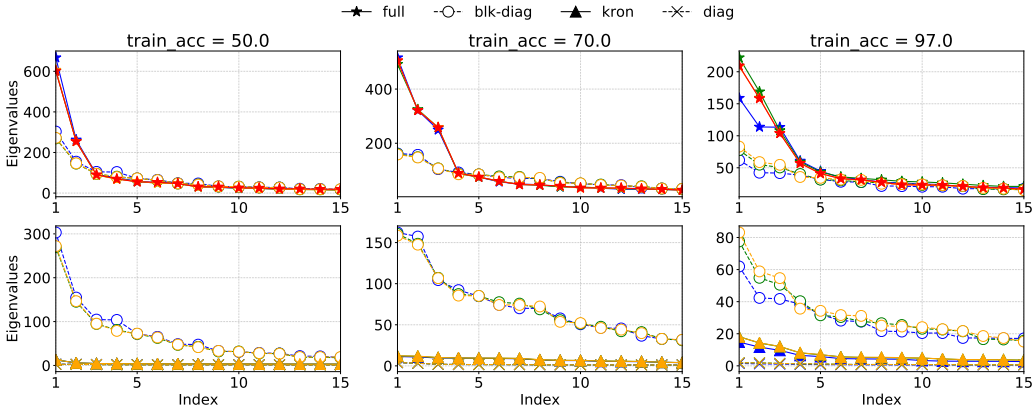
In Figure 7.6, we summarize the correlation between the generalization gap (accuracy gap) and measures by the eigenvalues of $\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$. For MNIST, Fully Connected and Simple CNN are trained using the dropout. For CIFAR-10/100, VGG-11 are trained without any data augmentation nor dropout. This difference probably makes a difference in the accuracy gap, *i.e.* accuracy gaps for MNIST models are around 0.2-1.6% while those for CIFAR-10/100 models are around 15-80%. The top eigenvalue $\lambda_{\max}(\cdot)$ for each plot shows relatively stronger positive correlation to the accuracy gap. Moreover, the measures by $\boldsymbol{F}_{\text{1mc}}$ tend to show the storongest correlation.

7.4.3 Pareto fronts

We show the Pareto fronts [McCandlish et al., 2018], that represent the trade-off between the minimum optimization steps and minimum examples processed required to achieve each checkpoint, and top eigenvalues of $\boldsymbol{H}$ of models trained with batch size of 32 and 4096 for each data set and networks (Figure 7.8-7.17). Note that each eigenvalues are calculated using the entire training set. For each checkpoint of each plot, the eigenvalues of $\boldsymbol{H}$ of the model trained with a large-batch (batch size = 4096) are generally larger than those of models trained with a small-batch (batch size = 32). We do not show the plot for ImageNet as it requires many models for each batch size to calculate the

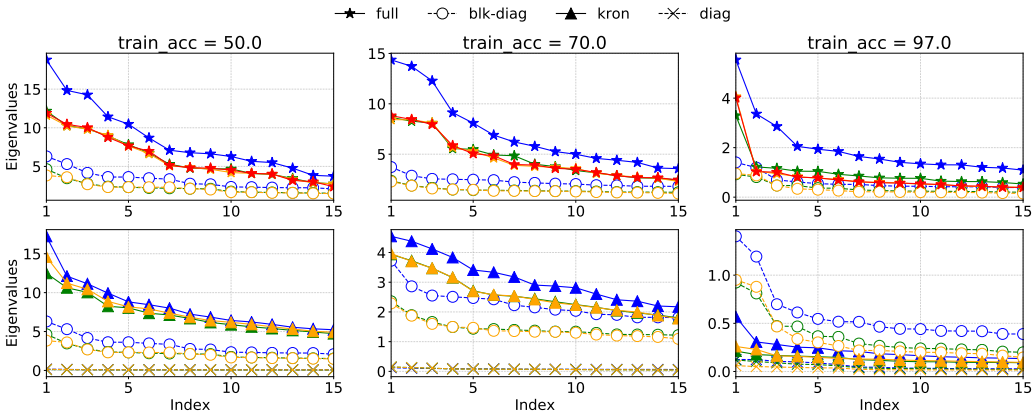


(a) Fully Connected on MNIST trained with BS=256.

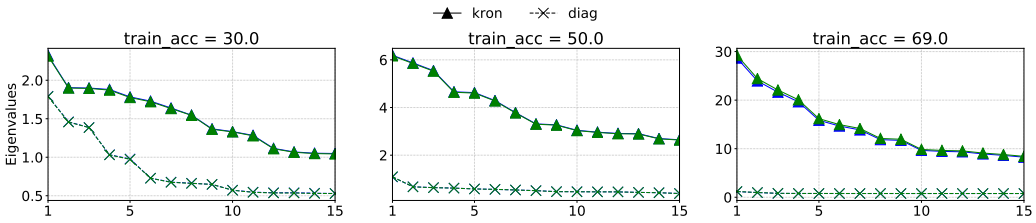


(b) VGG-11 w/o BatchNorm on CIFAR-10 trained with BS=256.

Figure 7.4: Top-15 eigenvalues of $\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{1\text{mc}}$ and $\boldsymbol{C}$ of various shapes at each phase of training.



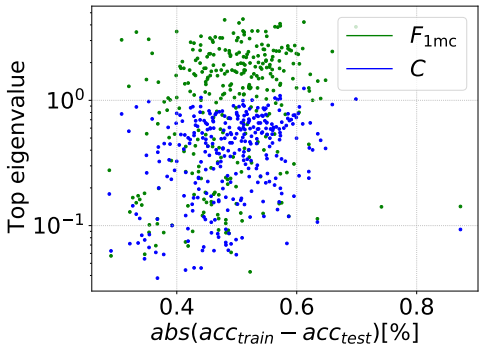
(a) VGG-11 w/ BatchNorm on CIFAR-100 trained with BS=256.



(b) ResNet-18 on ImageNet trained with BS=4096.

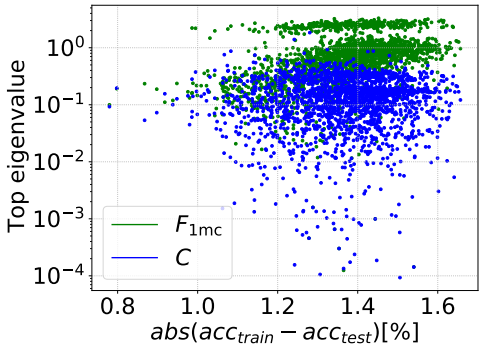
Figure 7.5: Top-15 eigenvalues of $\boldsymbol{H}$, $\boldsymbol{F}$, $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$ of various shapes at each phase of training. We only evaluate the Kronecker-factored and diagonal $\boldsymbol{F}_{\text{1mc}}$ and $\boldsymbol{C}$ due to the huge computational cost for the ImageNet models.

minimum steps and examples while the number of ImageNet models is small due to the limitation of computational budget.



Measure	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{1mc}$	$\boldsymbol{C}$
$\mathrm{Tr}(X_{\mathrm{full}})$	0.172	0.172	0.174	0.132
$\mathrm{Tr}(X_{\mathrm{kron}})$	-	0.178	0.181	0.149
$N_{\mathrm{eff}}(X_{\mathrm{kron}})$	-	0.156	0.157	0.121
$N_{\mathrm{eff}}(X_{\mathrm{diag}})$	-	0.151	0.149	0.106
$\lambda_{\max}(X_{\mathrm{full}})$	**0.194**	0.182	0.184	0.171
$\lambda_{\max}(X_{\mathrm{blk\text{-}diag}})$	-	0.186	0.188	0.193
$\lambda_{\max}(X_{\mathrm{kron}})$	-	0.187	**0.196**	0.184
$\lambda_{\max}(X_{\mathrm{diag}})$	-	0.188	0.186	0.188

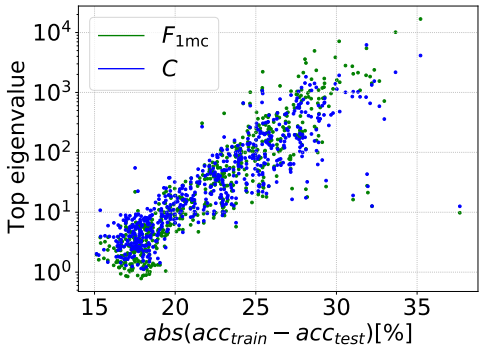
(a) 504 converged (training accuracy $\geq 99.9\%$) Simple CNN models on MNIST.



Measure	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{1mc}$	$\boldsymbol{C}$
$\mathrm{Tr}(X_{\mathrm{full}})$	0.277	0.277	0.274	0.107
$\mathrm{Tr}(X_{\mathrm{kron}})$	-	0.271	0.271	0.097
$N_{\mathrm{eff}}(X_{\mathrm{kron}})$	-	0.219	0.221	0.033
$N_{\mathrm{eff}}(X_{\mathrm{diag}})$	-	0.251	0.253	0.051
$\lambda_{\max}(X_{\mathrm{full}})$	0.288	0.286	**0.336**	0.057
$\lambda_{\max}(X_{\mathrm{blk\text{-}diag}})$	-	0.304	**0.342**	0.054
$\lambda_{\max}(X_{\mathrm{kron}})$	-	0.304	**0.333**	0.125
$\lambda_{\max}(X_{\mathrm{diag}})$	-	0.186	0.200	-0.016

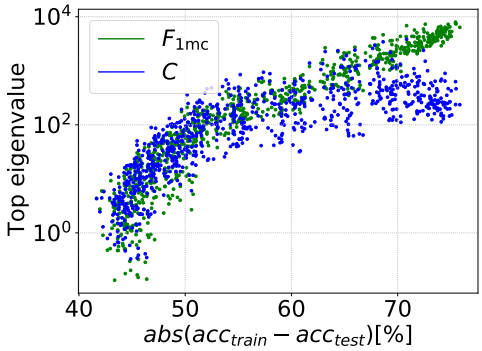
(b) 2400 converged (training accuracy $\geq 99.9\%$) Fully Connected models on MNIST.

Figure 7.6: **Left**: Correlation between $\lambda_{\max}(\boldsymbol{F}_{1mc,\mathrm{blk\text{-}diag}})$, $\lambda_{\max}(\boldsymbol{C}_{\mathrm{blk\text{-}diag}})$ and accuracy gap (train vs test). **Right**: Kendall's rank coefficient τ [Jiang et al., 2020] for each measure of the same models. A larger value indicates a stronger positive correlation. $\mathrm{Tr}(\cdot)$: trace. $N_{\mathrm{eff}}(\cdot)$: effective dimensions [Maddox et al., 2020]. $\lambda_{\max}(\cdot)$: top eigenvalue.



Measure	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{1\text{mc}}$	$\boldsymbol{C}$
$\text{Tr}(X_{\text{full}})$	**0.772**	**0.772**	**0.772**	0.735
$\text{Tr}(X_{\text{kron}})$	-	0.756	0.756	0.670
$N_{\text{eff}}(X_{\text{kron}})$	-	0.496	0.497	0.388
$N_{\text{eff}}(X_{\text{diag}})$	-	0.583	0.584	0.514
$\lambda_{\max}(X_{\text{full}})$	0.726	0.728	0.727	0.698
$\lambda_{\max}(X_{\text{blk-diag}})$	-	0.731	0.733	0.711
$\lambda_{\max}(X_{\text{kron}})$	-	0.666	0.666	0.616
$\lambda_{\max}(X_{\text{diag}})$	-	0.733	0.734	0.710

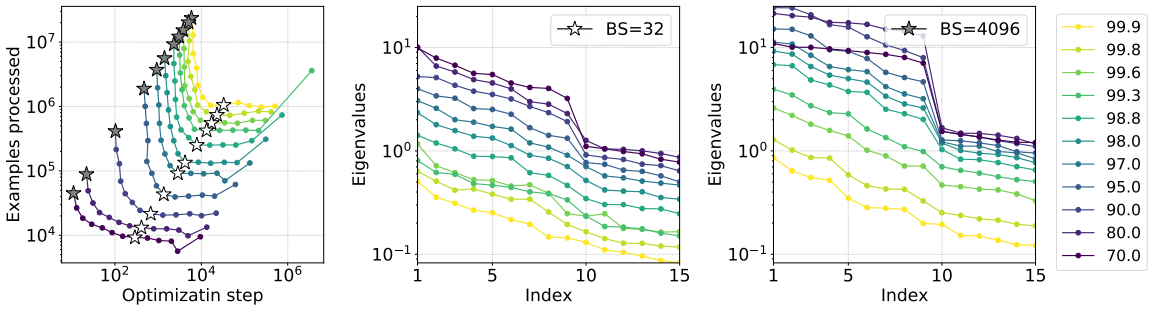
(a) 893 converged (training accuracy $\geq 97\%$) VGG-11 models on CIFAR-10.



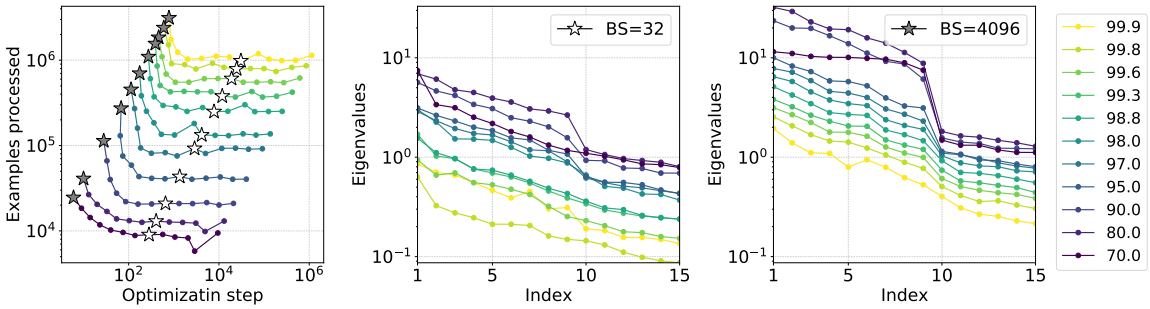
Measure	$\boldsymbol{H}$	$\boldsymbol{F}$	$\boldsymbol{F}_{1\text{mc}}$	$\boldsymbol{C}$
$\text{Tr}(X_{\text{full}})$	-	-	**0.844**	0.552
$\text{Tr}(X_{\text{kron}})$	-	-	**0.843**	0.571
$N_{\text{eff}}(X_{\text{kron}})$	-	-	0.563	0.315
$N_{\text{eff}}(X_{\text{diag}})$	-	-	0.561	0.300
$\lambda_{\max}(X_{\text{full}})$	**0.845**	-	**0.847**	0.616
$\lambda_{\max}(X_{\text{blk-diag}})$	-	-	**0.849**	0.644
$\lambda_{\max}(X_{\text{kron}})$	-	-	**0.840**	0.571
$\lambda_{\max}(X_{\text{diag}})$	-	-	**0.811**	0.469

(b) 943 converged (training accuracy $\geq 97\%$) VGG-11 models on CIFAR-100.

Figure 7.7: **Left**: Correlation between $\lambda_{\max}(\boldsymbol{F}_{1\text{mc,blk-diag}})$, $\lambda_{\max}(\boldsymbol{C}_{\text{blk-diag}})$ and accuracy gap (train vs test). **Right**: Kendall's rank coefficient τ [Jiang et al., 2020] for each measure of the same models. A larger value indicates a stronger positive correlation. $\text{Tr}(\cdot)$: trace. $N_{\text{eff}}(\cdot)$: effective dimensions [Maddox et al., 2020]. $\lambda_{\max}(\cdot)$: top eigenvalue.

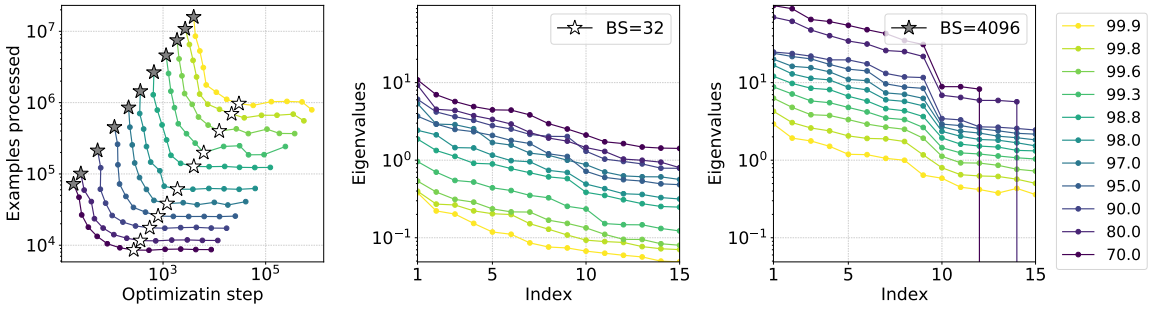


(a) SGD

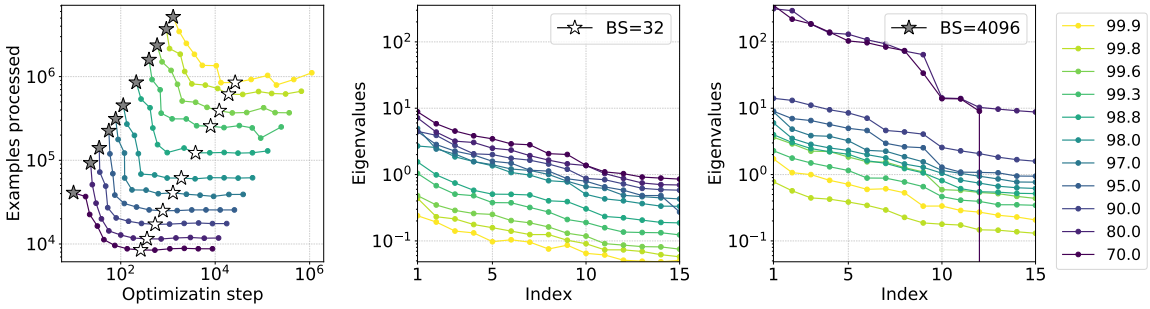


(b) Momentum

Figure 7.8: Fully Connected on MNIST.

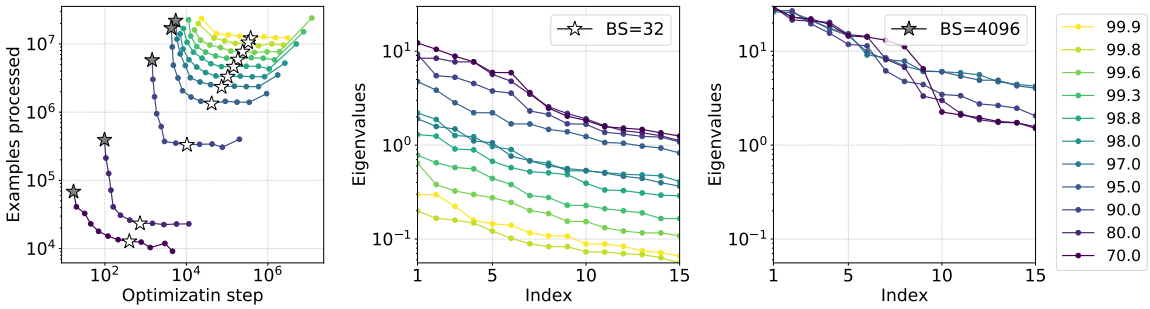


(a) SGD

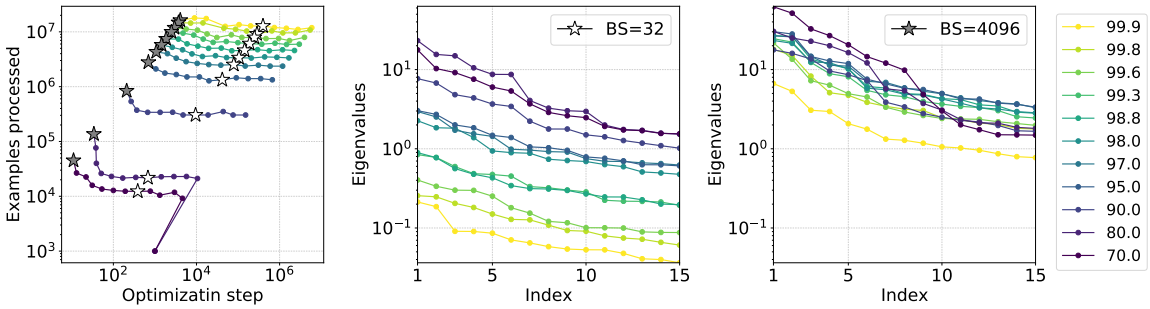


(b) Momentum

Figure 7.9: Simple CNN on MNIST.

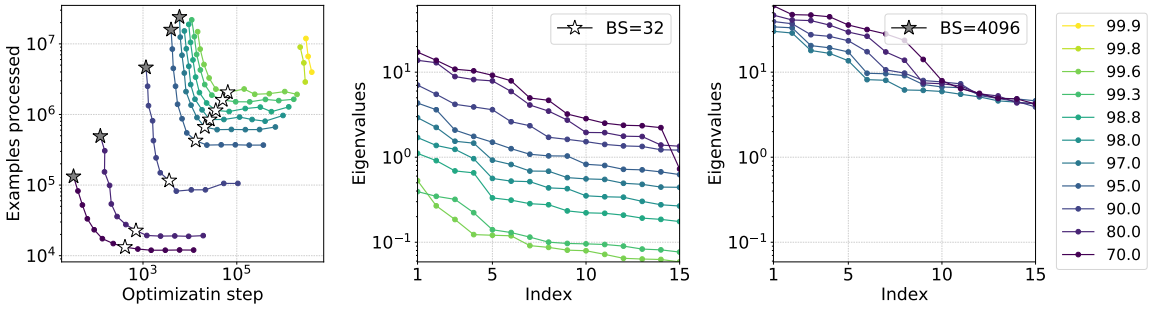


(a) SGD

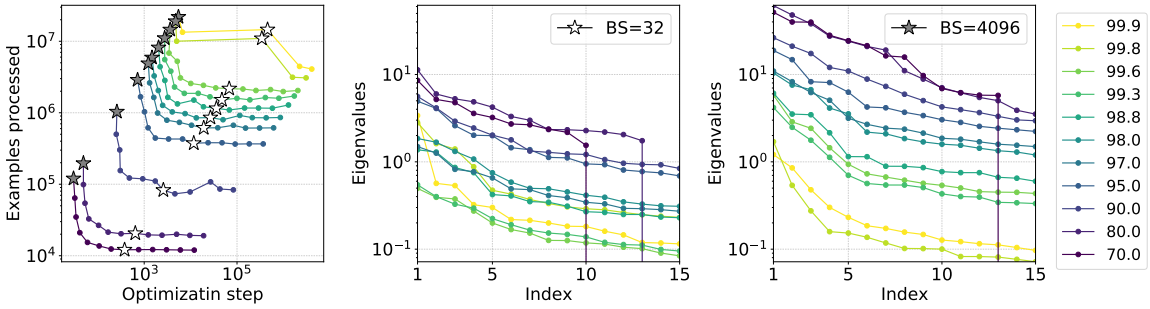


(b) Momentum

Figure 7.10: Fully Connected on Fashion-MNIST.

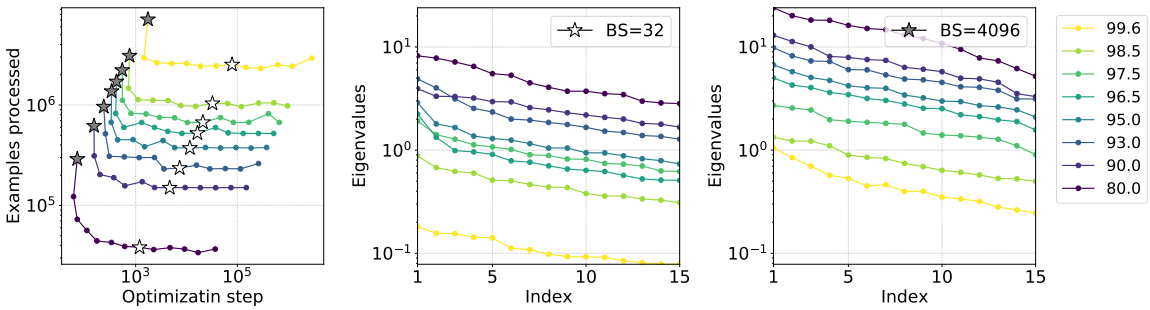


(a) SGD

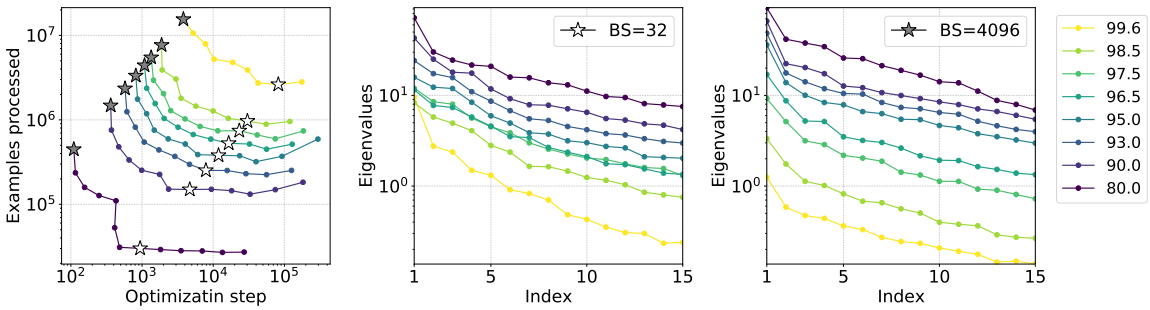


(b) Momentum

Figure 7.11: Simple CNN on Fashion-MNIST.



(a) Momentum



(b) Adam

Figure 7.12: Simple CNN on SVHN.

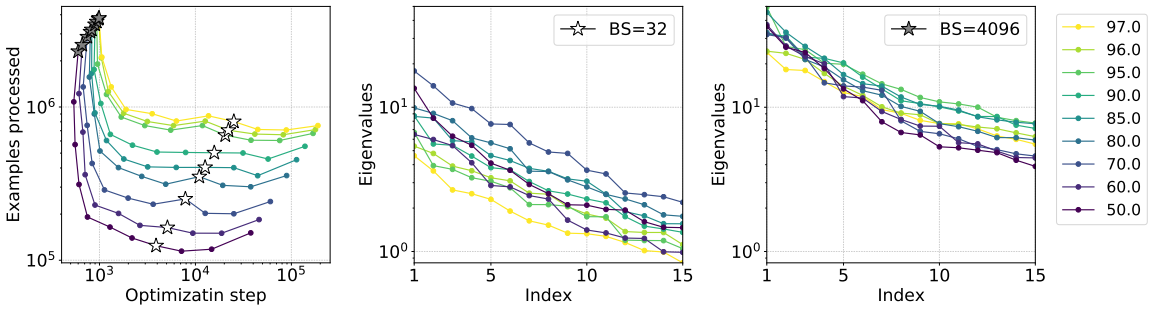


Figure 7.13: VGG-11 (Momentum) on CIFAR-10

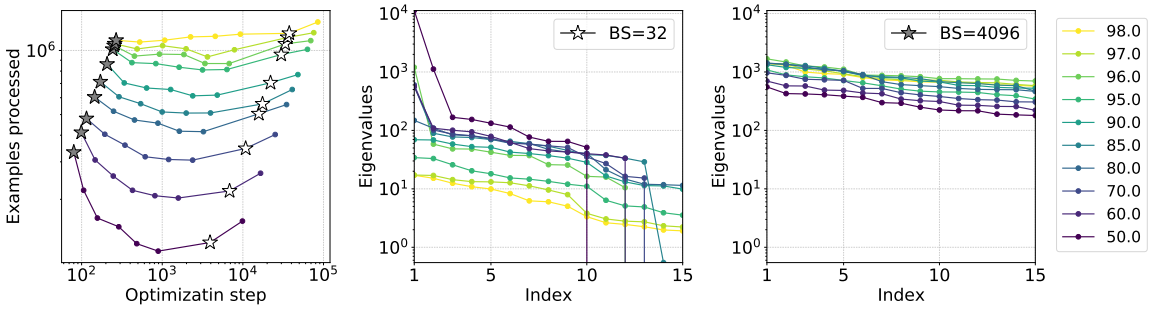
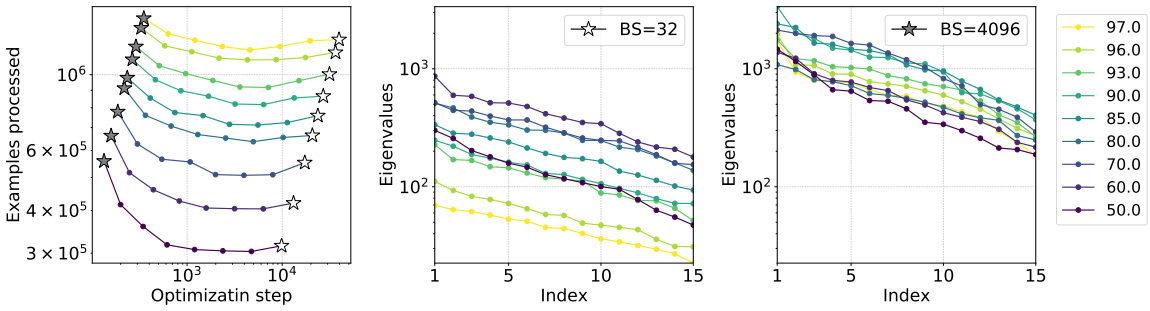
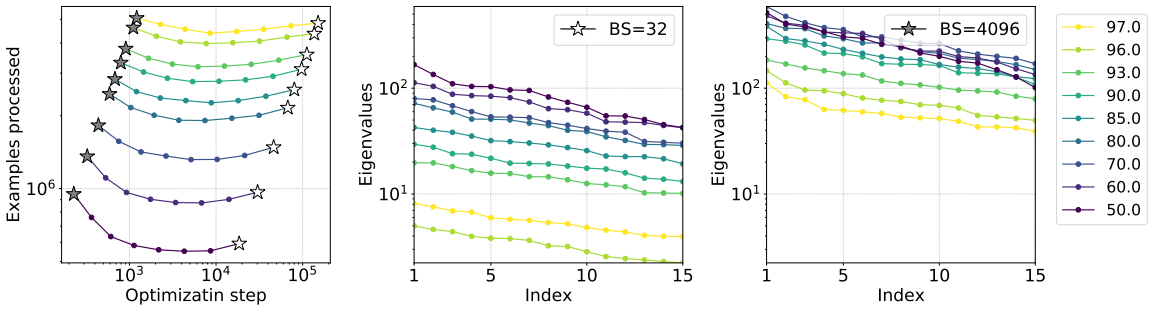


Figure 7.14: WideResNet-50 (Momentum) on CIFAR-10

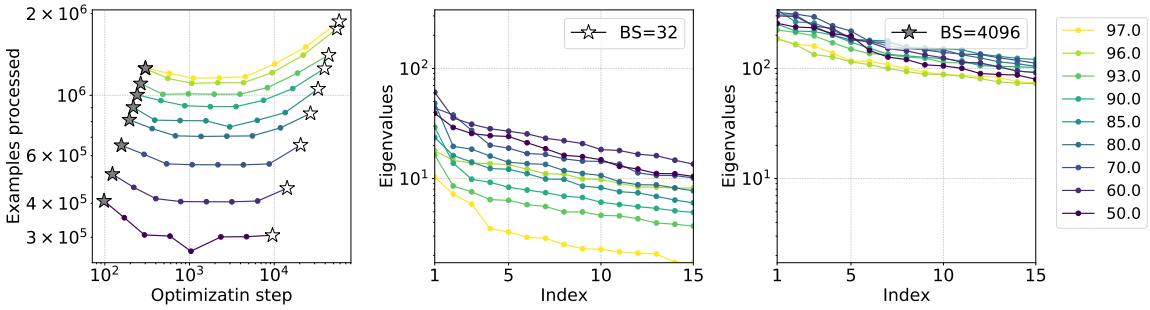


(a) Momentum

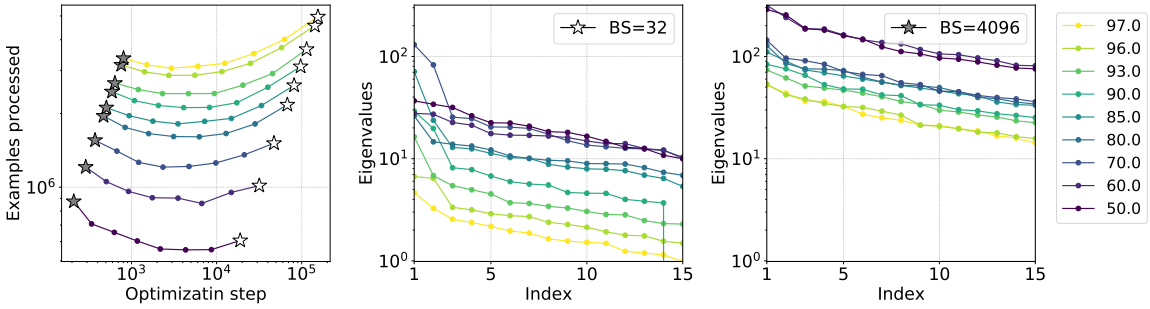


(b) Momentum w/ data augmentation

Figure 7.15: VGG-11 on CIFAR-100

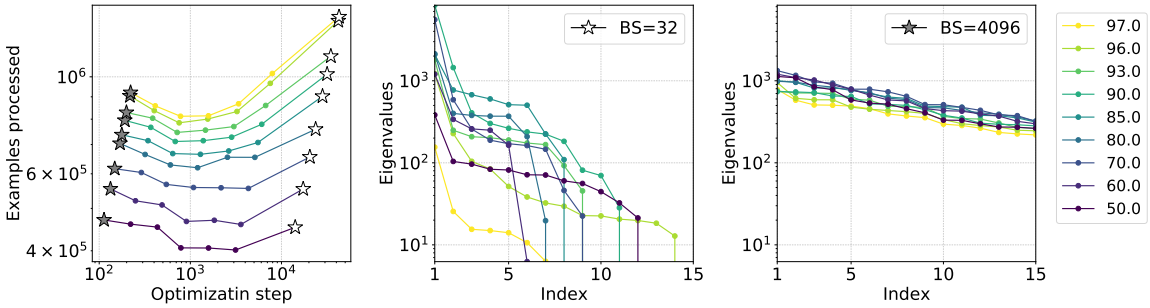


(a) Momentum

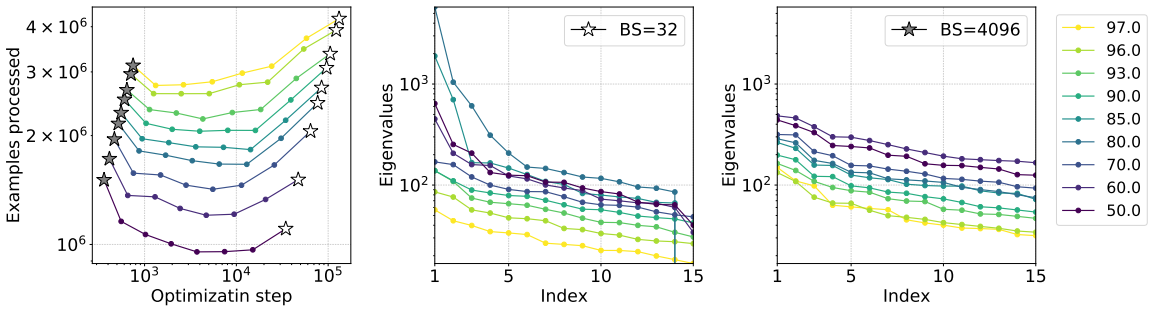


(b) Momentum w/ data augmentation

Figure 7.16: ResNet-18 on CIFAR-100



(a) Momentum



(b) Momentum w/ data augmentation

Figure 7.17: WideResNet-50 on CIFAR-100.

Chapter 8

Conclusion

This thesis discussed information matrices of deep neural networks and their applications in large-scale deep learning and efficient methods of approximation and computation. Information matrices and matrix-based algorithms have a lot of potential in various aspects of deep learning research. Still, their empirical effectiveness in large-scale deep learning settings has been largely unknown due to their computational difficulties. This thesis proposed the efficient realization of second-order optimization methods (information matrices as preconditioners) and validated its effectiveness by a fast implementation on a large GPU cluster. Also, we extended the resulting implementation to a realization of approximate Bayesian inference (i.e. variational inference) in large-scale deep learning settings. These lead to a demonstration of the empirical effectiveness of the matrix-based algorithm (i.e. second-order optimization and variational inference) on an unprecedented scale.

Our empirical findings are:

1. We proposed a distributed parallel *Natural Gradient Descent* (NGD) [Amari, 1998], an approximate second-order optimization method, to train ResNet-50 [He et al., 2016] for the classification of a large image dataset ImageNet [Deng et al., 2009] in 5.5 min. This is the first time to experimentally demonstrate second-order optimization's effectiveness in such a large-scale deep learning setting and has motivated subsequent research on second-order optimization and distributed training.

2. By extending this result, we showed that the distributed NGD could accelerate variational Bayesian inference (Bayesian inference by mathematical optimization) on the probability distribution of DNN parameters to acquire well-calibrated prediction and better generalization performance in large-scale computer vision tasks scaling up to ImageNet classification. This is the first work which gives an empirical result that an approximate Bayesian inference works in a large-scale deep learning setting.

8.1 Future work

This thesis's results use approximated information matrices and do not bring out the full potential of the principled matrix-based methods. There is a lot of room for improving information matrix computation performance by exploiting distributed parallel algorithms; optimized collective communication; low numerical precision operations and quantization; fast solvers for linear systems; and data-centric parallel programming that have long been studied in the HPC field. Once one can evaluate a principled method with accurate information in a broad range of DL settings, it facilitates both theoretical understanding and the development of practical methods in deep learning.

More accurate and efficient FIM estimation. We showed that NGD using he empirical Fisher matrix [Martens, 2020] (`emp`) is much faster than that with an estimation using a single Monte Carlo (`1mc`), which is widely used by related work on the approximate natural gradient. Although it is stated that `emp` is not a good approximation of the NGD in the literature [Thomas et al., 2019, Kunstner et al., 2019], we observed the same convergence behavior as `1mc` for training ResNet-50 on ImageNet. We might be able to attribute this result to the fact that `emp` is a good enough approximation to keep the behavior of the true NGD or that even `1mc` is not a good approximation. To know whether these hypotheses are correct or not and to examine the actual value of the exact NGD, we need a more accurate and efficient estimation of the NGD with less computational cost.

Better uncertainty estimation. Better uncertainty estimates open up a whole range of potential future experiments, for example, small data experiments, active learning, adversarial experiments, and sequential decision making. Our results on a continual-learning task confirm this. Another potential avenue for research is to consider structured covariance approximations.

Comprehensive studies of information matrices in various deep learning settings. We investigated the properties of the eigenvalues of the information matrix of various types/shapes. In the experiments, we only observed the correlation between eigenvalue-based metrics and deep neural networks' generalization performance. Future studies should observe the effects of types/shapes in various matrix-based algorithms.

Bibliography

Martın Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. *arXiv preprint arXiv:1711.04325*, 2017.

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

Shun-ichi Amari, Ryo Karakida, and Masafumi Oizumi. Fisher Information and Natural Gradient Learning in Random Deep Networks. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 694–702, 2019.

Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*, 2017.

David Barber and Christopher M Bishop. Ensemble Learning in Bayesian Neural Networks. *Generalization in Neural Networks and Machine Learning*, 168:215–238, 1998.

Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Computing Surveys*, 52(4):1–43, August 2019. ISSN 03600300. doi: 10.1145/3320060. URL `http://dl.acm.org/citation.cfm?doid=3359984.3320060`.

Alberto Bernacchia, Mate Lengyel, and Guillaume Hennequin. Exact natural gradient in deep linear networks and its application to the nonlinear case. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5941–5950, 2018.

Lukas Biewald. Experiment Tracking with Weights & Biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, Verlag, Berlin, Heidelberg, 2006. ISBN 0-387-31073-8.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight Uncertainty in Neural Networks. In *International Conference on Machine Learning (ICML)*, pages 1613–1622, 2015. arXiv: 1505.05424.

Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical Gauss-Newton Optimisation for Deep Learning. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 557–565, 2017.

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv preprint arXiv:1606.04838*, 2018. URL `http://arxiv.org/abs/1606.04838`. arXiv: 1606.04838.

James Bradbury, Roy Frostig, Peter Hawkins, James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

John Bradshaw, Alexander G. de G. Matthews, and Zoubin Ghahramani. Adversarial Examples, Uncertainty, and Transfer Testing Robustness in Gaussian Process Hybrid Deep Networks. *arXiv preprint arXiv:1707.02476*, 2017.

Tianle Cai, Ruiqi Gao, Jikai Hou, Siyu Chen, Dong Wang, Di He, Zhihua Zhang, and Liwei Wang. Gram-Gauss-Newton Method: Learning Overparameterized Neural Networks for Regression Problems. *arXiv:1905.11675 [cs, math, stat]*, September 2019. URL `http://arxiv.org/abs/1905.11675`. arXiv: 1905.11675.

Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On Empirical Comparisons of Optimizers for Deep Learning. *arXiv preprint arXiv:1910.05446*, 2019.

Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into Backprop. In *International Conference on Learning Representations (ICLR)*, 2020. URL `https://openreview.net/forum?id=BJlrF24twB`.

Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv:1406.2572 [cs, math, stat]*, June 2014. URL `http://arxiv.org/abs/1406.2572`. arXiv: 1406.2572.

Morris H. DeGroot and Stephen E. Fienberg. The comparison and evaluation of forecasters. *The Statistician: Journal of the Institute of Statisticians*, 32:12–22, 1983.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

Terrance DeVries and Graham W. Taylor. Learning Confidence for Out-of-Distribution Detection in Neural Networks. *arXiv preprint arXiv:1802.04865*, 2018.

Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *International Conference on Machine Learning (ICML)*, pages 1050–1059, 2016.

Jacob R. Gardner, Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 7576–7586, 2018.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast Approximate Natural Gradient Descent in a Kronecker Factored Eigenbasis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, pages 9550–9560. Curran Associates, Inc., 2018. URL `http://papers.nips.cc/paper/8164-fast-approximate-natural-gradient-descent-in-a-kronecker-factored-eigenbasis.pdf`.

Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An Investigation into Neural Net Optimization via Hessian Eigenvalue Density. In *International Conference on Machine Learning (ICML)*, pages 2232–2241, 2019.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, 2010.

Ian Goodfellow. Efficient Per-Example Gradient Computations. *arXiv preprint arXiv:1510.01799*, 2015.

Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. *arXiv preprint arXiv:1312.6082*, 2014.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

Diego Granziol, Xingchen Wan, Timur Garipov, Dmitry Vetrov, and Stephen Roberts. MLRG Deep Curvature. *arXiv preprint arXiv:1912.09656*, 2019.

Alex Graves. Practical Variational Inference for Neural Networks. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 2348–2356, 2011.

Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning (ICML)*, pages 573–582, 2016.

Roger B Grosse and Ruslan Salakhutdinov. Scaling Up Natural Gradient by Sparsely Factorizing the Inverse Fisher Matrix. In *International Conference on Machine Learning (ICML)*, pages 2304–2313, 2015.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On Calibration of Modern Neural Networks. In *International Conference on Machine Learning (ICML)*, pages 1321–1330, 2017.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Science & Business Media. Springer, 2009.

Fengxiang He, Tongliang Liu, and Dacheng Tao. Control Batch Size and Learning Rate to Generalize Well: Theoretical and Empirical Evidence. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1143–1152, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

Dan Hendrycks and Kevin Gimpel. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2017. URL `https://openreview.net/forum?id=Hkg4TI9xl`.

E Hinton. Keeping Neural Networks Simple by Minimizing the Description Length of the Weights. In *Annual Conference on Computational Learning Theory*, pages 5–13, 1993.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

Jennifer A Hoeting, David Madigan, Adrian E Raftery, and Chris T Volinsky. Bayesian model averaging: a tutorial. *Statistical science*, pages 382–401, 1999.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *arXiv:1705.08741 [cs, stat]*, January 2018. URL `http://arxiv.org/abs/1705.08741`. arXiv: 1705.08741.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Proceedings of International Conference on International Conference on Machine Learning (ICML)*, pages 448–456, 2015.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural Tangent Kernel: Convergence and Generalization in Neural Networks. *arXiv:1806.07572 [cs, math, stat]*, June 2018. URL `http://arxiv.org/abs/1806.07572`. arXiv: 1806.07572.

Stanislaw Jastrzebski, Maciej Szymczak, Stanislav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho, and Krzysztof Geras. The Break-Even Point on Optimization Trajectories of Deep Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2020. URL `https://openreview.net/forum?id=r1g87C4KwB`.

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018. URL `http://arxiv.org/abs/1807.11205`. arXiv: 1807.11205.

Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic Generalization Measures and Where to Find Them. In *International Conference on Learning Representations (ICLR)*, 2020. URL `https://openreview.net/forum?id=SJgIPJBFvH`.

Ryo Karakida and Kazuki Osawa. Understanding Approximate Fisher Information for Fast Convergence of Natural Gradient Descent in Wide Neural Networks. *arXiv:2010.00879 [cond-mat, stat]*, October 2020. URL `http://arxiv.org/abs/2010.00879`. arXiv: 2010.00879.

Ryo Karakida, Shotaro Akaho, and Shun-ichi Amari. The Normalization Method for Alleviating Pathological Sharpness in Wide Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6403–6413, 2019.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Learning Representations (ICLR)*, 2017. URL `https://openreview.net/forum?id=H1oyRlYgg`.

Mohammad Emtiyaz Khan. *Variational learning for latent Gaussian model of discrete data.* Ph.D. thesis, University of British Columbia, 2012.

Mohammad Emtiyaz Khan and Wu Lin. Conjugate-Computation Variational Inference : Converting Variational Inference in Non-Conjugate Models to Inferences in Conjugate Models. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 878–887, 2017.

Mohammad Emtiyaz Khan and Didrik Nielsen. Fast yet Simple Natural-Gradient Descent for Variational Inference in Complex Models. In *International Symposium on Information Theory and Its Applications (ISITA)*, pages 31–35. IEEE, 2018.

Mohammad Emtiyaz Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam. In *International Conference on Machine Learning (ICML)*, pages 2616–2625, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015. URL `http://arxiv.org/abs/1412.6980`.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Pang Wei Koh and Percy Liang. Understanding Black-box Predictions via Influence Functions. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1885–1894, 2017.

Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images, 2009. URL `http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

Frederik Kunstner, Lukas Balles, and Philipp Hennig. Limitations of the Empirical Fisher Approximation. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4156–4167, 2019.

Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. In *Advances in Neural Information Processing Systems (NIPS)*, pages 6402–6413, 2017.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998a.

Yann LeCun, Corinna Cortes, and CJ Burges. The MNIST database of handwritten digits, 1998b. URL `http://yann.lecun.com/exdb/mnist`.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature14539. URL `http://www.nature.com/articles/nature14539`.

Kimin Lee, Honglak Lee, Kibok Lee, and Jinwoo Shin. TRAINING CONFIDENCE-CALIBRATED CLASSIFIERS FOR DETECTING OUT-OF-DISTRIBUTION SAMPLES. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://openreview.net/forum?id=ryiAv2xAZ`.

Shiyu Liang, R Srikant, and Yixuan Li. ENHANCING THE RELIABILITY OF OUT-OF-DISTRIBUTION IMAGE DETECTION IN NEURAL NETWORKS. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://openreview.net/forum?id=H1VGkIxRZ`.

Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast AutoAugment. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6662–6672, 2019.

Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't Use Large Mini-Batches, Use Local SGD. *arXiv:1808.07217 [cs, stat]*, February 2020. URL `http://arxiv.org/abs/1808.07217`. arXiv: 1808.07217.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient Episodic Memory for Continual Learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 6467–6476, 2017.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing Millions of Hyperparameters by Implicit Differentiation. *arXiv preprint arXiv:1911.02590*, 2019.

Ilya Loshchilov and Frank Hutter. DECOUPLED WEIGHT DECAY REGULARIZATION. In *International Conference on Learning Representations (ICLR)*, 2019. URL `https://openreview.net/forum?id=Bkg6RiCqY7`.

David Mackay. *Bayesian Methods for Adaptive Models*. Ph.D. thesis, California Institute of Technology, 1991.

David Mackay. *Information theory, inference and learning algorithms*. Cambridgeuniversity press, 2003.

Wesley Maddox, Timur Garipov, Pavel Izmailov, Dmitry Vetrov, and Andrew Gordon Wilson. A Simple Baseline for Bayesian Uncertainty in Deep Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 13153–13164, 2019.

Wesley J. Maddox, Gregory Benton, and Andrew Gordon Wilson. Rethinking Parameter Counting in Deep Models: Effective Dimensionality Revisited. *arXiv preprint arXiv:2003.02139*, 2020.

Stephan Mandt and Matthew D Hoffman. Stochastic Gradient Descent as Approximate Bayesian Inference. *Journal of Machine Learning Research*, 18:1–35, 2017.

James Martens. Deep learning via Hessian-free optimization. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 735–742, 2010.

James Martens. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2017.

James Martens. New Insights and Perspectives on the Natural Gradient Method. *Journal of Machine Learning Research*, 21(146):1–76, 2020.

James Martens and Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 2408–2417, 2015.

James Martens and Matthew Johnson. Kronecker-factored Curvature Approximations for Recurrent Neural Networks. In *International Conference on Learning Representations*, 2018.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An Empirical Model of Large-Batch Training. *arXiv preprint arXiv:1812.06162*, 2018.

Luke Metz, Niru Maheswaranathan, Ruoxi Sun, C. Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*, 2020.

Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U-chupala, Yoshiki Tanaka, and Yuichi Kageyama. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233*, 2018. URL `http://arxiv.org/abs/1811.05233`. arXiv: 1811.05233.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*, 2013.

Aaron Mishkin, Frederik Kunstner, Didrik Nielsen, Mark Schmidt, and Mohammad Emtiyaz Khan. SLANG: Fast Structured Covariance Approximations for Bayesian Deep Learning

with Natural Gradient. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6247–6257. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7862-slang-fast-structured-covariance-approximations-for-bayesian-deep-learning-with-natural-pdf.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT Press, 2018.

Mahdi Pakdaman Naeini, Gregory F Cooper, and Milos Hauskrecht. Obtaining Well Calibrated Probabilities Using Bayesian Binning. In *AAAI Conference on Artificial Intelligence*, pages 2901–2907, 2015.

Radford M. Neal. *Bayesian learning for neural networks*. Ph.D. thesis, University of Toronto, 1995.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

Cuong V. Nguyen, Yingzhen Li, Thang D. Bui, and Richard E. Turner. Variational Continual Learning. In *International Conference on Learning Representations (ICLR)*, 2018. URL https://openreview.net/forum?id=BkQqq0gRb.

Jorge Nocedal and Wright Stephen. *Numerical optimization*. Science & Business Media. Springer, 2006.

Kazuki Osawa, Siddharth Swaroop, Anirudh Jain, Runa Eschenhagen, Richard E Turner, Rio Yokota, and Mohammad Emtiyaz Khan. Practical Deep Learning with Bayesian Principles. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4289–4301, 2019a.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12359–12367, 2019b.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and Practical Natural Gradient for Large-Scale Deep Learning. *arXiv preprint arXiv:2002.06015*, 2020.

Razvan Pascanu and Yoshua Bengio. Revisiting Natural Gradient for Deep Networks. In *International Conference on Learning Representations (ICLR)*, 2014. URL https://openreview.net/forum?id=vz8AumxkAfz5U.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8026–8037, 2019.

Kaare Brandt Petersen and Michael Syskind Pedersen. The Matrix Cookbook, 2012. URL `http://matrixcookbook.com`.

Carsten Peterson and James R. Anderson. A Mean Field Theory Learning Algorithm for Neural Networks. *Complex Systems*, 1:995–1019, 1987.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv:1910.02054 [cs, stat]*, October 2019. URL `http://arxiv.org/abs/1910.02054`. arXiv: 1910.02054.

Aravind Rajeswaran, Chelsea Finn, Sham Kakade, and Sergey Levine. Meta-Learning with Implicit Gradients. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 113–124, 2019.

Yi Ren and Donald Goldfarb. Efficient Subsampled Gauss-Newton and Natural Gradient Methods for Training Neural Networks. *arXiv:1906.02353 [cs, stat]*, June 2019. URL `http://arxiv.org/abs/1906.02353`. arXiv: 1906.02353.

Carlos Riquelme, George Tucker, and Jasper Snoek. DEEP BAYESIAN BANDITS SHOWDOWN. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://openreview.net/forum?id=SyYe6k-CW`.

Hippolyt Ritter, Aleksandar Botev, and David Barber. Online Structured Laplace Approximations For Overcoming Catastrophic Forgetting. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3738–3748, 2018a.

Hippolyt Ritter, Aleksandar Botev, and David Barber. A Scalable Laplace Approximation for Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2018b. URL `https://openreview.net/forum?id=Skdvd2xAZ`.

Nicolas L. Roux, Pierre-antoine Manzagol, and Yoshua Bengio. Topmoumoute Online Natural Gradient Algorithm. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 849–856. Curran Associates, Inc., 2008. URL `http://papers.nips.cc/paper/3234-topmoumoute-online-natural-gradient-algorithm.pdf`.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3): 211–252, December 2015. ISSN 0920-5691, 1573-1405. doi: 10.1007/s11263-015-0816-y. URL `http://link.springer.com/10.1007/s11263-015-0816-y`.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive Neural Networks. *arXiv preprint arXiv:1606.04671*, 2016.

Lawrence K Saul, Tommi Jaakkola, and Michael I Jordan. Mean Field Theory for Sigmoid Belief Networks. *Journal of Artificial Intelligence Research*, 4:61–76, 1996.

Nicol N. Schraudolph. Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7):1723–1738, July 2002. ISSN 0899-7667, 1530-888X. doi: 10.1162/08997660260028683. URL `http://www.mitpressjournals.org/doi/10.1162/08997660260028683`.

Jonathan Schwarz, Jelena Luketina, Wojciech M Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & Compress: A scalable framework for continual learning. In *International Conference on Machine Learning (ICML)*, pages 4528–4537, 2018.

Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*, 2015. URL `http://arxiv.org/abs/1409.1556`.

Sidak Pal Singh and Dan Alistarh. WoodFisher: Efficient second-order approximations for model compression. *arXiv:2004.14340 [cs, stat]*, April 2020. URL `http://arxiv.org/abs/2004.14340`. arXiv: 2004.14340.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning (ICML)*, pages 1139–1147, 2013.

Siddharth Swaroop, Cuong V. Nguyen, Thang D. Bui, and Richard E. Turner. Improving and Understanding Variational Continual Learning. *arXiv preprint arXiv:1905.02099*, 2019.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv:1512.00567 [cs]*, December 2015. URL http://arxiv.org/abs/1512.00567. arXiv: 1512.00567.

Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of International Conference on Machine Learning (ICML)*, 2019.

Philip S Thomas. GeNGA: A Generalization of Natural Gradient Ascent with Positive and Negative Convergence Results. In *International Conference on Machine Learning (ICML)*, pages 1575–1583, 2014.

Valentin Thomas, Fabian Pedregosa, Bart van Merriënboer, Pierre-Antoine Mangazol, Yoshua Bengio, and Nicolas Le Roux. Information matrices and generalization. *arXiv preprint arXiv:1906.07774*, 2019. URL http://arxiv.org/abs/1906.07774. arXiv: 1906.07774.

Valentin Thomas, Fabian Pedregosa, Bart van Merriënboer, Pierre-Antoine Mangazol, Yoshua Bengio, and Nicolas Le Roux. On the interplay between noise and curvature and its effect on optimization and generalization. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

Seiya Tokui, Hiroyuki Yamazaki Vincent, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, and Brian Vogel. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, Anchorage, AK, USA, 2019. ACM Press. ISBN 978-1-4503-6201-6. doi: 10.1145/3292500.3330756. URL http://dl.acm.org/citation.cfm?doid=3292500.3330756.

Yohei Tsuji, Kazuki Osawa, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Performance Optimizations and Analysis of Distributed Deep Learning with Approximated Second-Order Optimization Method. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops - ICPP 2019*, pages 1–8, Kyoto, Japan, 2019. ACM Press. ISBN 978-1-4503-7196-4. doi: 10.1145/3339186.3339202. URL http://dl.acm.org/citation.cfm?doid=3339186.3339202.

Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. Normalized Flat Minima: Exploring Scale Invariant Definition of Flat Minima for Neural Networks using PAC-Bayesian Analysis. *arXiv preprint arXiv:1901.04653*, 2019.

Y. Ueno and R. Yokota. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, 2019. doi: 10.1109/CCGRID.2019.00057.

Aad van der Vaart and Ghosal Ghosal. *Fundamentals of nonparametric Bayesian inference*, volume 44. Cambridge University Press, 2017.

Twan van Laarhoven. L2 Regularization versus Batch and Weight Normalization. *arXiv preprint arXiv:1706.05350*, 2017. URL `http://arxiv.org/abs/1706.05350`. arXiv: 1706.05350.

Volodimir G. Vovk. Aggregating strategies. In *Proceedings of the Annual Workshop on Computational Learning Theory*, pages 371–386, 1990.

Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. EigenDamage: Structured Pruning in the Kronecker-Factored Eigenbasis. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 6566–6575, 2019. URL `http://arxiv.org/abs/1905.05934`. arXiv: 1905.05934.

Yeming Wen, Kevin Luk, Maxime Gazeau, Guodong Zhang, Harris Chan, and Jimmy Ba. An Empirical Study of Large-Batch Stochastic Gradient Descent with Structured Covariance Noise. In *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 3621–3631, 2020.

Lei Wu and Chao Ma. How SGD Selects the Global Minima in Over-parameterized Learning: A Dynamical Stability Perspective. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8279–8288, 2018.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael Mahoney. PyHessian: Neural Networks Through the Lens of the Hessian. *arXiv preprint arXiv:1912.07145*, 2020.

Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. *arXiv preprint arXiv:1811.06992*, 2018. URL `http://arxiv.org/abs/1811.06992`. arXiv: 1811.06992.

Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks. *arXiv preprint arXiv:1708.03888*, 2017a. URL `http://arxiv.org/abs/1708.03888`. arXiv: 1708.03888.

Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. *arXiv:1709.05011 [cs]*, September 2017b. URL `http://arxiv.org/abs/1709.05011`. arXiv: 1709.05011.

Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. LSUN: Construction of a Large-Scale Image Dataset using Deep Learning with Humans in the Loop. *arXiv preprint arXiv:1506.03365*, 2015.

Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12, 2016. URL `http://arxiv.org/abs/1605.07146`.

Guodong Zhang, Shengyang Sun, David Duvenaud, and Roger Grosse. Noisy Natural Gradient as Variational Inference. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 5852–5861, 2018a.

Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three Mechanisms of Weight Decay Regularization. *arXiv:1810.12281 [cs, stat]*, October 2018b. URL `http://arxiv.org/abs/1810.12281`. arXiv: 1810.12281.

Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George E. Dahl, Christopher J. Shallue, and Roger Grosse. Which Algorithmic Choices Matter at Which Batch Sizes? Insights From a Noisy Quadratic Model. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019a.

Guodong Zhang, James Martens, and Roger Grosse. Fast Convergence of Natural Gradient Descent for Overparameterized Neural Networks. *arXiv:1905.10961 [cs, stat]*, May 2019b. URL `http://arxiv.org/abs/1905.10961`. arXiv: 1905.10961.

Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond Empirical Risk Minimization. In *International Conference on Learning Representations*, 2018c. URL `http://arxiv.org/abs/1710.09412`. arXiv: 1710.09412.

Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random Erasing Data Augmentation. *arXiv preprint arXiv:1708.04896*, 2017. URL `http://arxiv.org/abs/1708.04896`. arXiv: 1708.04896.