

HIGHER ORDER OPTIMIZATION TECHNIQUES FOR MACHINE LEARNING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sudhir Babu Kylasa

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Charlie Y. Hu

School of Electrical and Computer Engineering

Dr. Ananth Y. Grama

School of Computer Science

Dr. David F. Gleich

School of Computer Science

Dr. Petros Drineas

School of Computer Science

Dr. Mithuna S. Thottethodi

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

School of Electrical and Computer Engineering

To my son,

Tanmay Kylasa

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Ananth Grama, for helping me towards my thesis. I have enjoyed working with Prof. Grama for the last 7 years which I will cherish for the rest of my life. His continuous motivation and insightful guidance are the main contributing factors because of which I am able to complete my PhD. I have benefitted immensely with his knowledge and suggestions and I will continue to look up to them throughout my professional career.

I would also like to thank Purdue University for giving me an opportunity to be a part of its student community and I am glad that I embarked on this opportunity at the right time. I am thankful to all the Professors for providing me an opportunity to participate in their courses. I have enjoyed learning new skills and gaining knowledge in these courses. Also, I am indebted to my doctoral committee advisors for their continuous guidance and valuable discussions during the development of my thesis.

I would like to thank my collaborators Dr. Hasan Metin Aktulga, Dr. Giorgiois Kollais, Dr. Shahin Mohammadi and Dr. Fred Roosta for working with me on research publications these past years. I consider myself honored working along with these smart people and have learned new skills and gained knowledge interacting with them. Particularly I am indebted to Mr. Chih-Hao Fang for collaborating with me and extending all the help in last couple of research articles, which are a significant part of this thesis.

And last but not least to my family for their unwavering support and encouragement throughout my PhD.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	x
SYMBOLS	xii
ABSTRACT	xvi
1 INTRODUCTION	1
1.1 Function Approximation	2
1.2 Characteristics of a solution	3
1.2.1 Necessary and Sufficient conditions of a local minimizer	3
1.3 Direct vs. Iterative Solvers	5
1.4 Significance of GPUs in optimization	6
1.5 Iterative solutions for unconstrained minimization problems	10
1.6 Line Search Methods	10
1.6.1 Steepest Descent or Gradient Descent	10
1.6.2 Newton Direction	12
1.6.3 Step-size Estimation for Line search Methods	14
1.7 Trust-region Methods	17
2 GPU ACCELERATED SUB-SAMPLED NEWTON’S METHOD	19
2.1 Introduction	20
2.2 Related Work	23
2.3 Theory, Algorithms and Implementation Details	24
2.3.1 Sub-Sampled Newton’s Method	24
2.3.2 Multi-Class classification	25
2.3.3 Implementation Details	29
2.4 Experimental Results	35

	Page
2.4.1 Experimental Setup and Data	37
2.4.2 Parameterization of Various Methods	37
2.4.3 Computing Platforms	38
2.4.4 Performance Comparisons	38
2.4.5 Sensitivity to Hyper-Parameter Tuning	43
2.5 Conclusions And Future Work	44
2.6 More Details On Softmax Function eq. (3.2)	45
2.6.1 Relationship to Logistic Regression with ± 1 -labels	45
2.7 Tensorflow’s Performance Comparison on Various Compute Platforms .	47
2.8 Additional Performance Comparisons with Quasi-Newton methods . . .	51
2.8.1 Covertypes Dataset	51
2.8.2 Gisette and Real-Sim Datasets	51
3 NEWTON-ADMM: A DISTRIBUTED GPU-ACCELERATED OPTIMIZER FOR MULTICLASS CLASSIFICATION PROBLEMS	52
3.1 Introduction	52
3.1.1 Related Research	54
3.2 Problem Formulation and Algorithm Details	56
3.2.1 Problem Formulation	56
3.2.2 ADMM Framework	57
3.2.3 Inexact Newton-CG Solver	60
3.3 Experimental Evaluation	62
3.4 Conclusions and Future Directions	70
3.5 Appendix	70
3.5.1 On GPU Utilization	71
3.5.2 Hessian Vector Product	73
3.5.3 Newton-type method as a highly efficient subproblem solver for ADMM:	74
3.5.4 Algorithms Parameter Settings	76

	Page
4 FITRE: FISHER INFORMED TRUST-REGION METHOD FOR TRAIN- ING DEEP NEURAL NETWORKS	77
4.1 Introduction and Motivation	78
4.2 Related Work	80
4.3 FITRE	82
4.3.1 Computational Model	84
4.3.2 Algorithm	92
4.4 Experiments	92
4.5 Conclusions and Future Work	110
4.6 Appendix	111
4.6.1 Memory Layout	111
4.6.2 Helper functions	111
4.7 Neural Network Operations	113
4.7.1 Gradient computation	113
4.7.2 Hessian Vector Operation on Neural Networks	117
4.7.3 Loss Functions	122
4.7.4 Activation Functions	124
4.7.5 Pooling Functions	125
4.7.6 Batch Normalization	127
REFERENCES	132

LIST OF TABLES

Table	Page
2.1 Description of the datasets. L indicates the Lipschitz Constant of the dataset.	36
2.2 Performance comparison between first-order and second-order methods. <i>FullNewton</i> uses the entire dataset for gradient and Hessian evaluations.	39
2.3 Performance comparison between first-order and second-order methods on CPU-only and 1-GPU-1-CPU-core compute platforms for <i>coverttype</i> dataset. Batch-size 128 first order methods are compared with second order methods using full gradient and hessian sample size set to 5%. Batch-size 20% first order methods are compared with second order methods using sample sizes of 20% and 5% for gradient and hessian computations respectively.	49
2.4 Performance comparison between our proposed methods and existing quasi-newton methods.	50
3.1 Description of the datasets.	63
3.2 GPU Speedup for Newton-ADMM and SGD.	65
3.3 Performance comparison of Newton-ADMM and GIANT – we present the number of epochs for a solver to reach $\theta < 0.05$. The speedup ratio is defined as the fraction of time taken by GIANT to achieve a specified value of θ to the corresponding time taken by Newton-ADMM on the same hardware platform.	70
4.1 Description of the datasets used in our experiments	94
4.2 Various Convolution Neural Networks used in our experiments. α is 512 when CIFAR10 and CIFAR100 are used, and for Imagenet, it is 2048. β is 10 for CIFAR10, 100 for CIFAR100, and 200 for ImageNet. These networks can be easily adapted for embedding BatchNormalization layers (typically after the convolution layer).	95
4.3 Comparison of VGG11 using ImageNet dataset	96
4.4 Comparison of VGG16 using ImageNet dataset	97
4.5 Comparison of VGG11 using cifar100 dataset	101
4.6 Comparison of VGG16 using cifar100 dataset	102

Table	Page
4.7 Comparison of VGG19 using cifar100 dataset	103
4.8 Comparison of VGG16 using cifar100 dataset with a quasi-Newton Method (L-BFGS).	106
4.9 Behavior of FITRE and SGD without regularization on CIFAR100 dataset. FITRE method uses an update frequency of 5 and “KFAC + gradient” option is turned off in these set of simulations. VGG networks in this table does not use batch-normalization function.	108
4.10 Reparameterization Invariance results for SGD and FITRE methods. VGG networks in these experiments use 0 regularization. And for FITRE method we use the KFAC update frequency of 5 and use only the natural gradient direction as the descent direction (KFAC + gradient option is not used in these experiments).	109

LIST OF FIGURES

Figure	Page
1.1 Architectural overview of CPU and GPU.	6
1.2 CUDA Streaming Multiprocessor Microarchitecture. Provides a birds-eye level view of the components included in a typical SM.	8
1.3 CUDA programming model.	9
2.1 Sensitivity of various first-order methods with respect to the choice of the step-size, i.e., learning-rate. It is clear that, too small a step-size can lead to slow convergence, while larger step-sizes cause the method to diverge. The range of step-sizes for which some of these methods perform reasonably, can be very narrow. This is contrast with Newton-type, which come with a priori “natural” step-size, i.e., $\alpha = 1$, and only occasionally require the line-search to intervene	44
3.1 Training objective function and test accuracy as functions of time for Newton-ADMM and synchronous SGD, both with GPU enabled and GPU disabled, with 4 workers. Overall, Newton-ADMM favors GPUs, enjoys minimal communication overhead, and enjoys faster convergence compared to synchronous SGD.	65
3.2 Training objective function and test accuracy comparison over time for Newton-ADMM, GIANT, InexactDANE, and AIDE on MNIST dataset with $\lambda = 10^{-5}$. We run both Newton-ADMM and GIANT for 100 epochs. Since the computation times per epoch for InexactDANE and AIDE are high, we only run 10 epochs for these methods. We present details of hyperparameter settings in 3.5.4.	66
3.3 Avg. Epoch Time for Strong and Weak Scaling for Newton-ADMM and GIANT.	68
3.4 $\log_{10}(\theta)$ as a function of time for Newton-ADMM and GIANT on MNIST, CIFAR-10, and HIGGS datasets. Newton-ADMM can reach lower θ , given the same amount of time, compared to GIANT. Note that for the HIGGS dataset, both methods can reach low θ soon.	69

Figure	Page
3.5 Training objective function and test accuracy as function of time for Newton-ADMM and GIANT on E18 dataset using 32 nodes. We note that GIANT lingers at higher objective values in the initial iterations, while Newton-ADMM drops to lower objective values rapidly.	69
3.6 Training Objective function comparison over time for different choice of inner-solve for ADMM. For the inner solver, we compare the performance of Inexact Newton solver with L-BFGS (with history size 25, 50, 100). The step size of Inexact Newton method is chosen by linesearch following Armijo rule, whereas the step size of L-BFGS is chosen by linesearch satisfying Strong Wolfe condition. We can see that the per-iteration computation cost of L-BFGS is lower than Inexact Newton with the exception on HIGGS dataset. This is because L-BFGS is sensitive to the scale of step size so that more iterations of Strong Wolfe linesearch procedure are required to satisfy the curvature condition. In general, we observe that L-BFGS performs well on binary class problems, while the performance degrades on multiclass problems, when the number of compute nodes increases.	75
4.1 CNN model and layer composition.	84

SYMBOLS

\mathbf{v}	Vectors in lower-case bold letters
\mathbf{V}	Matrices in upper-case bold letters
$\nabla f(\mathbf{x})$	Gradient of a function, f , evaluated at \mathbf{x}
$\nabla^2 f(\mathbf{x})$	Hessian of a function, f , evaluated at \mathbf{x}
$\mathbf{x}^{(k)}$	Superscript denote iteration count
\mathbf{x}_i	Subscript denotes the <i>local</i> -value of the vector \mathbf{x} at the i^{th} compute node in a distributed setting otherwise specifically mentioned
\mathcal{S}	Collection of indices drawn from a set $\{1, 2, \dots, n\}$
$ \mathcal{S} $	Cardinality of a set \mathcal{S}
$[\mathbf{v}; \mathbf{w}] \in \mathbb{R}^{2p}$	Vertical stacking of two column vectors \mathbf{v}, \mathbf{w}
$[\mathbf{v}, \mathbf{w}]$ a p by 2 matrix	Matrix formed by two column vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^{2p}$
$\ \mathbf{x}\ $	Vector ℓ_2 norm of \mathbf{x}
$\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}^T \mathbf{v}$	Dot product of vectors \mathbf{u} and \mathbf{v}
$\mathbf{A} \odot \mathbf{B}$	Element-wise multiplication of matrices \mathbf{A} and \mathbf{B}
$\mathbf{1}(x)$	Indicator function $x \in \{\text{True}, \text{False}\}$ which evaluates to 1 if $x = \text{True}$ and 0 otherwise
\mathcal{D}	Input dataset
$F_i(\mathbf{x})$	Objective function, $F(\mathbf{x})$, evaluated at point \mathbf{x} using i^{th} — observation
$F_{\mathcal{D}}(\mathbf{x})$	Objective function evaluated on the entire dataset \mathcal{D}
(\mathbf{x}, y)	Sample point and its target (ground truth)
C	Total no. of classes in the dataset
$f(\mathbf{x}, y)$	Function of the network

z	Prediction of the network w.r.t \mathbf{x}
\mathcal{L}	Loss function
\mathbf{g}	Gradient of the network
\mathbf{H}	Hessian of the network
\mathbf{F}	Fisher information matrix of the network
\mathbf{p}	Natural gradient direction
$\ell, (1, \dots, \ell)$	No. of layers in the neural network
\mathbf{a}_{l-1}	Input to layer l
$\tilde{\mathbf{A}}_{l-1}$	Mini-batch input to layer l
$\bar{\mathbf{a}}_{l-1}$	Input to layer l , augmented with homogeneous coordinate
\mathbf{a}_l	Output of layer l
\mathbf{W}_l	Weights of layer l
\mathbf{b}_l	Bias of layer l
$\bar{\mathbf{W}}_l$	Weights and bias folded into single parameter of layer l
$[\text{vec}(\bar{\mathbf{W}}_1)^\top, \dots, \text{vec}(\bar{\mathbf{W}}_\ell)^\top]^\top$	Parameters of the network, $\boldsymbol{\theta}$
\mathbf{s}_l	Output of the convolution function (and input to the activation function) of layer l
\mathbf{p}_l	Output of activation function (and input to pool function) of layer l
\mathbf{a}_l	Output of pool function of layer l
$\mathbf{a}_{out}(= \mathbf{a}_\ell)$	Output of the neural network
\mathbf{a}_{loss}	Output of the loss function
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$	Derivative of the loss function w.r.t a parameter, $\boldsymbol{\theta}$
$\mathbf{g}_{loss} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{out}} \right)$	Gradient terms propagated from the loss function
$\mathbf{g}_l^p \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{p}_l} \right)$	Gradient terms propagated from the pool function of layer l
$\mathbf{g}_l^a \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{s}_l} \right)$	Gradient terms propagated from the activation function of layer l

$\mathbf{g}_l^{conv} \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{a}_l} \right)$	Gradient terms propagated from the convolution function of layer l
\mathbf{A}_{l-1}	Input to the convolution function of layer l
$\tilde{\mathbf{A}}_{l-1}$	Mini-batch input to the convolution function of layer l ($\tilde{}$ is used to indicate mini-batch processing of a given matrix)
\mathbf{C}_l	Input to the activation function of layer l
\mathbf{P}_l	Input to the pool function of layer l
\mathbf{A}_l	Output of layer l (and input to the layer $l + 1$)
\mathbf{A}_{out}	Output of the neural network
\mathbf{A}_{loss}	Output of the loss function
$\mathbf{G}_{loss} \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{A}_{out}} \right)$	Gradient terms propagated from the loss function
$\mathbf{G}_l^p \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{P}_l} \right)$	Gradient terms propagated from the pool function of layer l
$\mathbf{G}_l^a \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{C}_l} \right)$	Gradient terms propagated from the activation function of layer l
$\mathbf{G}_l^{conv} \left(= \frac{\partial \mathcal{L}}{\partial \mathbf{A}_l} \right)$	Gradient terms propagated from the convolution function of layer l
$\mathcal{R}_v \{ \mathbf{A}_l \}$	Input to convolution function of layer l of the neural network
$\mathcal{R}_v \{ \mathbf{C}_l \}$	Input to activation function of layer l of the neural network
$\mathcal{R}_v \{ \mathbf{P}_l \}$	Input to pool function of layer l of the neural network
$\mathcal{R}_v \{ \mathbf{a}_{out} \}$	Output of the neural network
$\mathcal{R}_v \{ \mathbf{G}_{loss} \}$	$\mathcal{R}_v \{ \}$ terms propagated from the loss function
$\mathcal{R}_v \{ \mathbf{G}_l^p \}$	$\mathcal{R}_v \{ \}$ terms propagated from the pool function of layer l
$\mathcal{R}_v \{ \mathbf{G}_l^a \}$	$\mathcal{R}_v \{ \}$ terms propagated from the activation function of layer l

$\mathcal{R}_v \{ \mathbf{G}_l^{conv} \}$

$\mathcal{R}_v \{ \}$ terms propagated from the convolution function
of layer l

ABSTRACT

Kylasa, Sudhir Babu Ph.D., Purdue University, December 2019. Higher Order Optimization Techniques for Machine Learning. Major Professor: Charlie Y. Hu.

First-order methods such as Stochastic Gradient Descent are methods of choice for solving non-convex optimization problems in machine learning. These methods primarily rely on the gradient of the loss function to estimate descent direction. However, they have a number of drawbacks, including converging to saddle points (as opposed to minima), slow convergence, and sensitivity to parameter tuning. In contrast, second order methods that use curvature information in addition to the gradient, have been shown to achieve faster convergence rates, theoretically. When used in the context of machine learning applications, they offer faster (quadratic) convergence, stability to parameter tuning, and robustness to problem conditioning. In spite of these advantages, first order methods are commonly used because of their simplicity of implementation and low per-iteration cost. The need to generate and use curvature information in the form of a dense Hessian matrix makes each iteration of second order methods more expensive.

In this work, we address three key problems associated with second order methods – (i) what is the best way to incorporate curvature information into the optimization procedure; (ii) how do we reduce the operation count of each iteration in a second order method, while maintaining its superior convergence property; and (iii) how do we leverage high-performance computing platforms to significantly accelerate second order methods. To answer the first question, we propose and validate the use of Fisher information matrices in second order methods to significantly accelerate convergence. The second question is answered through the use of statistical sampling techniques that suitably sample matrices to reduce per-iteration cost without impacting conver-

gence. The third question is addressed through the use of graphics processing units (GPUs) in distributed platforms to deliver state of the art solvers.

Through our work, we show that our solvers are capable of significant improvement over state of the art optimization techniques for training machine learning models. We demonstrate improvements in terms of training time (over an order of magnitude in wall-clock time), generalization properties of learned models, and robustness to problem conditioning.

1. INTRODUCTION

The availability of large datasets, combined with ever increasing processing power of current hardware platforms, has motivated a number of new applications in computer vision, natural language processing, recommender systems, optical character recognition, and autonomous vehicles, among others. Optimization has become one of the main tools in data science to train models of the physical world around us to extract meaningful insights from it. At a high level, optimization techniques are used to parametrize models from data with the goal of minimizing error, maximizing generalizability beyond training set, while satisfying application constraints. Given the diversity of applications and models, there is no universal optimization procedure that can be used across all applications and data regimes. Choosing the right model, objective function for optimization, and optimization technique, are critical aspects of machine learning. Optimization methods can be broadly classified as:

Continuous vs. Discrete Optimization problems where variables can only take certain types of data (for instance, integers) are called discrete optimization problems. In contrast, if variables are allowed to take real numbers, the problem is called continuous. Note that discrete optimization problems are relatively difficult to solve because of the hardness of underlying computational problems.

Constrained vs. Unconstrained Optimization problems where restrictions (or constraints) are imposed on the variables used in the underlying model are called constrained optimization problems. In contrast, models without any constraints on the variables are referred to as unconstrained optimization problems. When the function and all its constraints are linear functions in variables, then the problem

is called a linear program. Likewise non-linear programs contain atleast one of the constraints or the function that is non-linear.

Convex vs. Non-Convex This is one of the fundamental properties of functions that influences the choice of the optimization technique. A function, f , is convex if its domain, S , is a convex set. That is, for any two points \mathbf{x} and \mathbf{y} in $S(= \mathcal{R}^n)$, the following property is satisfied:

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y}), \text{ for all } \alpha \in [0, 1]$$

Convex programming is used to describe optimization problems where the function and constraints are all convex functions. Because of the above property convex problems are easier to solve relative to non-convex problems. In the convex regime, a local solution is always a global solution, while in non-convex regime there exist many local solutions, and identifying global solutions is hard for high-dimensional problems.

1.1 Function Approximation

Taylor series approximations of functions are commonly used in optimization problems primarily as a means to approximate a function around a point to estimate a solution when traversed along a certain direction in its neighborhood. Suppose that $f : \mathcal{R}^n \rightarrow \mathcal{R}$ is continuously differentiable, and that $\mathbf{p} \in \mathcal{R}^n$. Then we have:

$$f(\mathbf{x} + \mathbf{p}) = f(\mathbf{x}) + \nabla f(\mathbf{x} + t\mathbf{p})^T \mathbf{p}$$

for some $t \in (0, 1)$. Moreover, if f is twice continuously differentiable, we have:

$$f(\mathbf{x} + \mathbf{p}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{p} + \frac{1}{2} \mathbf{p}^T \nabla^2 f(\mathbf{x} + t\mathbf{p}) \mathbf{p}$$

1.2 Characteristics of a solution

In this subsection, we briefly discuss the conditions which a solution of a minimization problem needs to satisfy in general. More elaborate discussion on the solution characteristics and related theorems and proofs are discussed in Jorge Nocedal's work [1]. Some of the proofs are discussed here for completeness.

A point \mathbf{x}^* is a *global minimizer* if $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all \mathbf{x} .

where \mathbf{x} ranges all over \mathcal{R}^n . The global minimizer is difficult to find because our knowledge of f is local. Since the complete knowledge of f is unavailable, most algorithms only seek to find local minimizers, which is a point that achieves the smallest value of f in a neighborhood. Formally, this is defined as:

A point \mathbf{x}^* is a local minimizer if there is a neighborhood \mathbb{N} of \mathbf{x}^* such that

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \text{ for all } \mathbf{x} \in \mathbb{N}.$$

1.2.1 Necessary and Sufficient conditions of a local minimizer

We briefly discuss the necessary and sufficient conditions that must be satisfied by any solution of a minimization problem. More elaborate discussion on the associated theorems and proofs can be found in [1–3]. We discuss some important properties of the solution, and their proofs are included here for completeness.

Theorem 1.2.1 (First order Necessary Conditions) *If \mathbf{x}^* is a local minimizer and f is a continuously differentiable function in the open neighborhood of \mathbf{x}^* then $\nabla f(\mathbf{x}^*) = 0$.*

Proof Suppose $\nabla f(\mathbf{x}^*) \neq \mathbf{0}$, by contradiction. Define a vector in the opposite direction of the gradient

$$\mathbf{p} = -\nabla f(\mathbf{x}^*), \text{ and note that } \mathbf{p}^T \nabla f(\mathbf{x}^*) = -\|\nabla f(\mathbf{x}^*)\|^2 < 0.$$

Because ∇f is continuous near \mathbf{x}^* there is a scalar $T(> 0)$ such that:

$$\mathbf{p}^T \nabla f(\mathbf{x}^* + t\mathbf{p}) < 0, \quad \text{for all } t \in [0, T] \quad (1.1)$$

For any $\hat{t} \in (0, T]$, now using the Taylor series approximation we have:

$$f(\mathbf{x}^* + \hat{t}\mathbf{p}) < f(\mathbf{x}^*) + \hat{t}\mathbf{p}^T \nabla f(\mathbf{x}^* + t\mathbf{p}) \quad \text{for some } t \in (0, \hat{t}) \quad (1.2)$$

Therefore, $f(\mathbf{x}^* + \hat{t}\mathbf{p}) < f(\mathbf{x}^*)$, for all $\hat{t} \in (0, T]$. This means that we have found a direction away from \mathbf{x}^* along which f decreases, so \mathbf{x}^* cannot be a local minimizer. ■

Theorem 1.2.2 (Second order necessary conditions) *If \mathbf{x}^* is a local minimizer of f and $\nabla^2 f$ exists and is continuous in the neighborhood of \mathbf{x}^* , then $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $\nabla^2 f(\mathbf{x}^*)$ is positive semidefinite.*

Proof We already know that $\nabla f(\mathbf{x}^*) = \mathbf{0}$. Again by contradiction assume that $\nabla^2 f(\mathbf{x}^*)$ is not positive semidefinite. Then choose a vector \mathbf{p} such that $\mathbf{p}^T \nabla^2 f(\mathbf{x}^*) \mathbf{p} < 0$, because $\nabla^2 f$ is continuous near \mathbf{x}^* , then there is a scalar $T > 0$ such that $\mathbf{p}^T \nabla^2 f(\mathbf{x}^* + t\mathbf{p}) \mathbf{p} < 0$ for all $t \in [0, T]$.

By using the Taylor series approximation around \mathbf{x}^* , we have for all $\hat{t} \in (0, T]$ and some $\hat{t} \in (0, \hat{t})$ that:

$$f(\mathbf{x}^* + \hat{t}\mathbf{p}) < f(\mathbf{x}^*) + \hat{t}\mathbf{p}^T \nabla f(\mathbf{x}^* + t\mathbf{p}) + \frac{1}{2}\hat{t}^2 \mathbf{p}^T \nabla^2 f(\mathbf{x}^* + t\mathbf{p}) \mathbf{p} < f(\mathbf{x}^*) \quad (1.3)$$

This indicates that we have found a direction \mathbf{p} away from \mathbf{x}^* along which f is decreasing. Hence \mathbf{x}^* is not a local minimizer anymore. ■

Theorem 1.2.3 (Second order sufficient conditions) *Suppose that $\nabla^2 f$ is continuous in the open neighborhood of \mathbf{x}^* and that $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $\nabla^2 f(\mathbf{x}^*)$ is positive definite. Then \mathbf{x}^* is the strict local minimizer of f .*

Proof Because Hessian is continuous and positive definite at \mathbf{x}^* , we can choose a radius, $r > 0$, so that $\nabla^2 f(\mathbf{x}^*)$ remains positive definite for all \mathbf{x} in the open ball $\mathbb{D} = \{\mathbf{z} \mid \|\mathbf{z} - \mathbf{x}^*\| < r\}$. Then taking an nonzero vector \mathbf{p} with $\|\mathbf{p}\| < r$, we have $\mathbf{x}^* + \mathbf{p} \in \mathbb{D}$, and therefore:

$$\begin{aligned} f(\mathbf{x}^* + \mathbf{p}) &= f(\mathbf{x}^*) + \mathbf{p}^T \nabla f(\mathbf{x}^*) + \frac{1}{2} \mathbf{p}^T \nabla^2 f(\mathbf{z}) \mathbf{p} \\ &= f(\mathbf{x}^*) + \frac{1}{2} \mathbf{p}^T \nabla^2 f(\mathbf{z}) \mathbf{p} \end{aligned}$$

Here, $\mathbf{z} = \mathbf{x}^* + t\mathbf{p}$ for some $t \in (0, 1)$. Since $\mathbf{z} \in \mathbb{D}$, we have $\frac{1}{2} \mathbf{p}^T \nabla^2 f(\mathbf{z}) \mathbf{p} > 0$, and therefore $f(\mathbf{x}^* + \mathbf{p}) > f(\mathbf{x}^*)$, giving the necessary result. ■

Theorem 1.2.4 *When f is convex, any local minimizer \mathbf{x}^* is a global minimizer of f . If in addition, f is differentiable (smooth), then any stationary point \mathbf{x}^* is a global minimizer of f .*

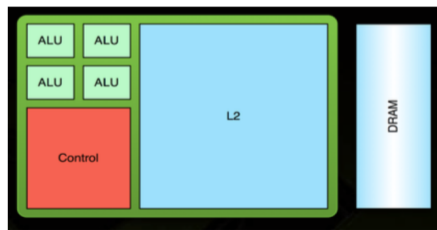
Non-smooth Problems In this thesis, we only consider smooth optimization problems, by which we mean functions whose first (and second) derivatives exist and are continuous. However, there exist many functions that are either non-smooth (whose derivatives does not exist) and/or are discontinuous. In such cases, functions may be a combinations of smooth segments with discontinuities between them. In such cases, minimizers can be found by identifying the individual smooth segments. Subgradients are used in cases where a function is continuous everywhere but non differentiable at some points.

1.3 Direct vs. Iterative Solvers

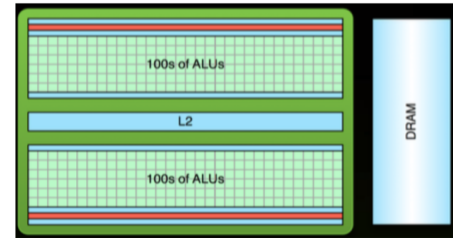
Optimization problems can be solved by using either a direct method, where an analytical solution exists for the underlying problem, or an iterative method, where a

closed form solution does not exist or it is impractical to solve. For instance, consider a linear system of equations $\mathbf{Ax} = \mathbf{b}$ in \mathcal{R}^n . A direct solution can be obtained by using LU decomposition (square matrices), cholesky factorization (symmetric positive definite matrices) and QR factorization (non-singular matrices). Each of these solution techniques has different computational complexity for obtaining a solution, and in high-dimensional space, $n \gg 1$, using any of these methods may become impractical because of amount of time required to obtain a solution and/ or the availability of resources needed for execution. In many high-dimensional problems, matrix \mathbf{A} might not be available, or be impractical to store in physical memory, and only operations on this matrix are supported, such as matrix-vector products. Iterative solvers form a suitable alternative in such scenarios, where a solution vector \mathbf{x}_k , also known as an iterate, is formed initially and repeatedly updated in each iteration of the solver. In contrast to direct solvers, \mathbf{x}_k is always available and the iterative process can be stopped, without running it to completion, if a satisfactory solution is arrived during this process. Iterative solvers can be designed in such a way that the matrix \mathbf{A} is not needed for updating the iterative solution, and only operations such as matrix-vector products, \mathbf{Ax}_k , are used.

1.4 Significance of GPUs in optimization



(a) High level view of a typical CPU and its components.



(b) High level view of a typical GPU and its components.

Fig. 1.1.: Architectural overview of CPU and GPU.

Conventional processors, which we also refer to as CPUs, have been the go-to number crunching hardware platform until recent times. With the availability of Graphical Processing Units (GPUs), the landscape of high-performance hardware has drastically changed, mainly because of the throughput of floating point operations these devices can handle. Conventional CPUs contain one or more processing cores that are made up of processing units (known as Arithmetic and Logic Units (ALUs)), Control Unit, which is responsible for scheduling hardware-level instructions, and a low-latency cache known (or a hierarchy of caches); as shown in 1.1. This central processor is aided by the off-chip high latency random-access memory (RAM) and a high capacity very high latency disk memory. A state-of-the-art CPU can support 16 to 32 thread level parallelism and a motherboard can support up to 8 CPUs. Such a hardware platform is designed to execute instructions operating on data from low-latency caches rapidly, yielding maximum throughput. Sophisticated control unit with pipelined speculative execution consumes much of the real-estate on the die in a typical CPU. Note that at any instant of time only a limited number of threads (few hundreds) can be in execution state, because of number of ALUs available on the processor itself. Compute intensive operations such as large matrix-vector products, ubiquitous in many optimization problems, form the main bottleneck in such hardware platforms.

GPUs have been primarily used for rendering images on to the display units (computer monitors) until recently. Processing images by applying an operation on each pixel (arithmetic or a boolean operation) in very short time period, and delivering these images to be displayed on to the monitor was the primary task of a typical graphical processor. Over the years it has been shown that graphics processors can be used to perform compute intensive tasks such as matrix operations with better speed, relative to conventional CPUs. This led to the evolution of GPUs over the past decade, and have now become the main work-horse in high-performance computing.

GPUs consist of streaming multiprocessors, also known as SM's, and an on-chip memory unit. Each SM, shown in Fig. 1.2 consists of processing units (potentially

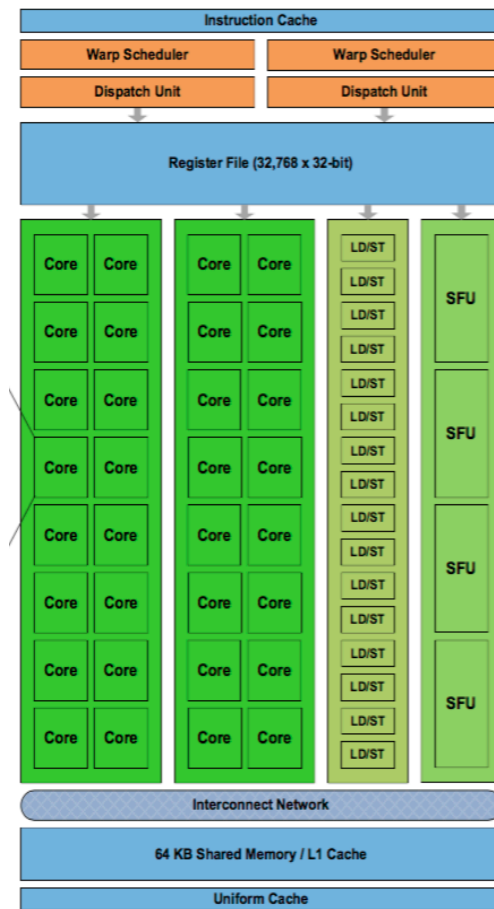


Fig. 1.2.: CUDA Streaming Multiprocessor Microarchitecture. Provides a birds-eye level view of the components included in a typical SM.

few hundreds), which take up much of the physical space on the die, along with associated control-unit, registers and cache memory. The processing unit itself is relatively simple, and can typically handle limited arithmetic operations. Special function units (SFUs) are provided to handle complex arithmetic and logic operations, and are shared by the SM. Various types of memory such as L1/L2 cache, shared memory, register file are available for a streaming multiprocessor, and a high-capacity high-latency off-chip global memory is shared by all the SMs. This hardware platform is optimized for data-parallel high-through computations such as matrix operations

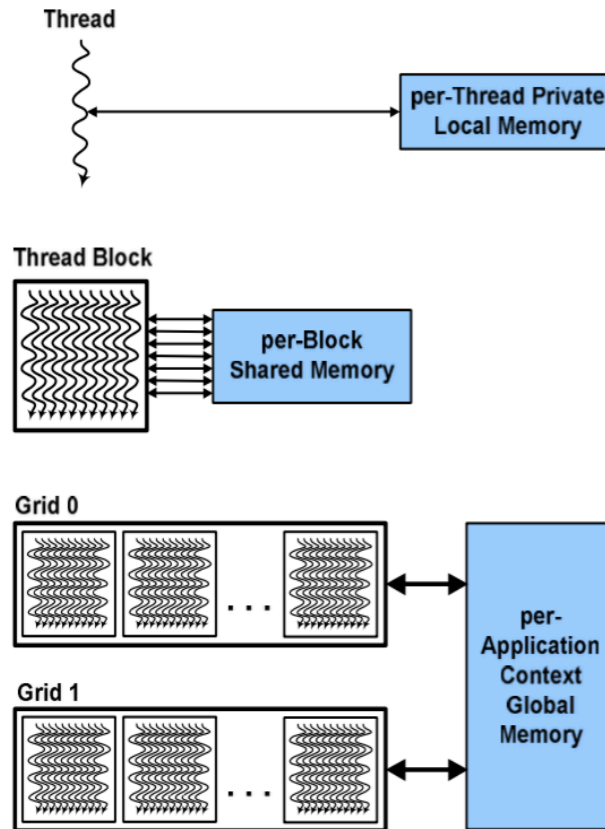


Fig. 1.3.: CUDA programming model.

in an optimization toolkit. The micro-architecture is tolerant to memory latency associated with computations on thousands of processing cores on the GPU. GPUs offer Single-instruction-multi-thread, SIMT, programming semantics, which enables the system software to spawn thousands of threads simultaneously. Scheduling and dispatching is done in units of a small number (typically 32) threads, known as warps. Multiple warps form thread blocks, which can be grouped to form a grids, which are executed inside a kernel, as shown in Fig. 1.3. Once a kernel, which defines the number of grids and thread-blocks, is triggered, low-level threads grouped in thread-blocks are instantiated and scheduled onto an SM. A thread-block resides on an SM throughout

its lifetime, and its memory reads/writes are scheduled per half-warp because of the bus width connecting the processing cores and the memory unit.

1.5 Iterative solutions for unconstrained minimization problems

In this work, we are interested in iterative solutions to unconstrained minimization problems for both convex and non-convex optimization. We present a brief discussion on the two well-known iterative solutions for unconstrained minimization problems, which we use in our research. We show that these methods, when coupled with highly efficient implementations on GPUs, yield significantly better results compared to other alternatives.

1.6 Line Search Methods

While minimizing a function $f(\mathbf{x})$, we need a search direction, \mathbf{p} , and the length of the step along \mathbf{p} to be traversed, to update the solution iterate, \mathbf{x}_k , which represents the solution \mathbf{x} in the k^{th} iteration of the solver. Two popular search directions are *steepest descent* and *Newton* directions. We briefly discuss these two search directions and the behavior of the solver when these directions are used during the minimization process.

1.6.1 Steepest Descent or Gradient Descent

The steepest descent direction, $-\nabla f_k$, is the most obvious choice for search direction, because it represents the direction along which f decreases most rapidly among all the possible directions from \mathbf{x}_k . Using the Taylor series approximation, we know the following:

$$f(\mathbf{x}_k + \alpha \mathbf{p}) = f(\mathbf{x}_k) + \alpha \mathbf{p}^T \nabla f_k + \frac{1}{2} \alpha^2 \mathbf{p}^T \nabla^2 f(\mathbf{x}_k + t \mathbf{p}) \mathbf{p}, \text{ for some } t \in (0, \alpha)$$

where α is the step size. The rate of change of f along the direction of \mathbf{p} is given by the coefficient of α , which is $\mathbf{p}^T \nabla f_k$ (by ignoring the higher order terms of α). Hence the unit direction of \mathbf{p} of most rapid decrease is the solution to the following problem:

$$\min_{\mathbf{p}} \mathbf{p}^T \nabla f_k, \text{ subject to: } \|\mathbf{p}\| = 1$$

Since $\mathbf{p}^T \nabla f_k = \|\mathbf{p}\| \|\nabla f_k\| \cos \theta = \|\nabla f_k\| \cos \theta$, where θ is the angle between \mathbf{p} and ∇f_k . The minimizer is obtained when $\cos \theta = -1$, which is:

$$\mathbf{p} = -\frac{\nabla f_k}{\|\nabla f_k\|}$$

Please note that since the computation of this search direction only uses the gradient, ∇f_k , of the function f , the associated methods are called *first-order* methods.

Convergence Analysis of Steepest Descent Consider the ideal case, where the objective function is convex quadratic and line searches are exact. Suppose that:

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} - \mathbf{b}^T \mathbf{x} \quad (1.4)$$

where \mathbf{Q} is symmetric and positive definite. The gradient is given by $\nabla f(\mathbf{x}) = \mathbf{Q} \mathbf{x} - \mathbf{b}$, and the minimizer \mathbf{x}^* is the unique solution given by the linear solve $\mathbf{Q} \mathbf{x} = \mathbf{b}$.

In order to compute step lengths α_k , which minimize $f(\mathbf{x}_k - \alpha \nabla f_k)$ we differentiate the following function:

$$f(\mathbf{x}_k - \alpha \nabla f_k) = \frac{1}{2} (\mathbf{x}_k - \alpha \nabla f_k)^T \mathbf{Q} (\mathbf{x}_k - \alpha \nabla f_k) - \mathbf{b}^T (\mathbf{x}_k - \alpha \nabla f_k)$$

w.r.t α and set it to zero, which gives:

$$\alpha_k = \frac{\nabla f_k^T \nabla f_k}{\nabla f_k^T \mathbf{Q} \nabla f_k}$$

Thus, we have the iterates using the steepest descent as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left(\frac{\nabla f_k^T \nabla f_k}{\nabla f_k^T \mathbf{Q} \nabla f_k} \right) \nabla f_k$$

In order to quantify the rate of convergence, let us define $\|\mathbf{x}\|_{\mathbf{Q}}^2 = \mathbf{x}^T \mathbf{Q} \mathbf{x}$, we have:

$$\frac{1}{2} \|\mathbf{x} - \mathbf{x}^*\|_{\mathbf{Q}}^2 = f(\mathbf{x}) - f(\mathbf{x}^*)$$

Now, given the above equations, we can establish the convergence relation as follows:

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_{\mathbf{Q}}^2 = \left\{ 1 - \frac{(\nabla f_k^T \nabla f_k)^2}{(\nabla f_k^T \mathbf{Q} \nabla f_k) (\nabla f_k^T \mathbf{Q}^{-1} \nabla f_k)} \right\} \|\mathbf{x}_k - \mathbf{x}^*\|_{\mathbf{Q}}^2 \quad (1.5)$$

This equation is difficult to interpret, but can be rewritten in the following form (using eigenvalues of the symmetric positive definite matrix, \mathbf{Q}) as:

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_{\mathbf{Q}}^2 \leq \left(\frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1} \right) \|\mathbf{x}_k - \mathbf{x}^*\|_{\mathbf{Q}}^2$$

assuming $0 \leq \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ are the eigenvalues of \mathbf{Q} .

This suggests that steepest descent method using exact line searches tends to make slow progress in the neighborhood of the minimizer, \mathbf{x}^* , and this does not improve even when inexact line search is used to estimate the step size. Also, the convergence is sensitive to problem ill-conditioning. For this reason, regularization plays a crucial role in its convergence.

1.6.2 Newton Direction

Newton's method was originally formulated for root finding of a uni-variate function, $\phi(x)$, $x \in \mathcal{R}$ such that:

$$\phi(x^*) = 0$$

For that, it uses a linear approximation (truncated Taylor series). Assuming that we have some t close to t^* , we have:

$$\phi(t + \Delta t) = \phi(t) + \nabla \phi(t) \Delta t + o(|\Delta t|)$$

Equating $\phi(t + \Delta t) = 0$, results in the following linear equation:

$$\phi(t) + \nabla \phi(t) \Delta x = 0$$

Assuming that the displacement, Δt is a good approximation of the optimal displacement ($\Delta t^* = t^* - t$), Newton method's iterates can be written as:

$$t_{k+1} = t_k - \frac{\phi(t_k)}{\nabla \phi(t_k)} t_{k+1} = t_k - (\nabla \phi(t_k))^{-1} \phi(t_k)$$

Hence, the Newton's method for optimization problems can be written in the following form:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\nabla^2 f(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k) \quad (1.6)$$

$[\nabla^2 f(\mathbf{x}_k)]^{-1}$ is commonly known as the *Newton-* direction, which is primarily used in second-order optimization methods, since the second derivative is used to compute the descent direction.

Note that this can also be obtained by using the Taylor series approximation of $f(\mathbf{x}_k + \mathbf{p})$, which is:

$$f(\mathbf{x}_k + \mathbf{p}) \approx f_k + \mathbf{p}^T \nabla f_k + \frac{1}{2} \mathbf{p}^T \nabla^2 f_k \mathbf{p} \stackrel{\text{def}}{=} m_k(\mathbf{p})$$

By setting the derivative of $m_k(\mathbf{p})$ to 0, we obtain the Newton direction as we had earlier.

Newton direction can be used in line search methods because it is a descent direction. Consider the following:

$$\nabla^T f_k \mathbf{p}_k^N = -\mathbf{p}_k^{N^T} \nabla^2 f_k \mathbf{p}_k^N = -\alpha_k \|\mathbf{p}_k^N\|^2 \quad (1.7)$$

for some $\alpha_k > 0$. Since $\nabla^T f_k \mathbf{p}_k^N$ is $-ve$, \mathbf{p}_k is a descent direction. Note that $\nabla^2 f_k$ may not always be positive definite so the Newton direction may not exist. A positive definite approximation of the Hessian can be defined by using either rank-1 (SR-1) or rank-2 (BFGS, DFP) or Gauss-Newton matrices. These methods, also known as *Quasi-Newton* methods, are some of the work-arounds to handle the complexities associated with Hessian computation.

Convergence rate of Newton's Method On the convergence rate of Newton's method, we have the following theorem:

Theorem 1.6.1 *Let $f(\mathbf{x})$ be a smooth and twice differentiable function which is*

1. *gradient is Lipschitz continuous, $\|f'(\mathbf{x}) - f'(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$*
2. *Hessian is Lipschitz continuous, $\|f''(\mathbf{x}) - f''(\mathbf{y})\| \leq M\|\mathbf{x} - \mathbf{y}\|$*
3. *Hessian is positive definite, $f''(\mathbf{x}^*) \geq lI_n$*

Suppose that the initial starting point, \mathbf{x}_0 is close enough to \mathbf{x}^ :*

$$\|\mathbf{x}_0 - \mathbf{x}^*\| \leq \bar{r} \left(\frac{2l}{3M} \right)$$

Then $\|\mathbf{x}_k - \mathbf{x}^\| < \bar{r}$ for all k , and the Newton method converges quadratically:*

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \frac{M\|\mathbf{x}_k - \mathbf{x}^*\|^2}{2(l - M\|\mathbf{x}_k - \mathbf{x}^*\|)}$$

Proof The details of the proof can be found in [1] ■

1.6.3 Step-size Estimation for Line search Methods

Once we have a descent direction, \mathbf{p}_k , (either the anti-gradient or Newton direction) we need to find the step size along this direction so that we can achieve a certain reduction in the objective function. The exact distance needed to traverse along \mathbf{p}_k is obtained by solving the following problem:

$$\min_{\alpha > 0} f(\mathbf{x}_k + \alpha \mathbf{p}_k) \quad (1.8)$$

Ideally, we would like to solve the above problem with minimum compute effort possible. Often, the exact step size is not needed, and in some cases it might lead to divergence or very slow convergence (as in the case of quadratic convex function discussed earlier). Two popular strategies used for this purpose are: Armijo conditions, Armijo-Wolfe conditions, and in some cases polynomial interpolation.

Backtracking line search : For inexact step size computation while finding the approximate solution for the eq. 1.8, following conditions can be enforced; sufficient decrease and curvature conditions. When computing the next iterate, \mathbf{x}_{k+1} , sufficient decrease condition is used to control the amount of decrease in the objective function value achieved by the estimated step size, as given by:

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f_k^T \mathbf{p}_k$$

for some constant $c_1 \in (0, 1)$. Note that it uses the first-order Taylor series approximation at the next iterate.

Using the sufficient decrease condition (also known as *Armijo*-condition), a simple back-tracking line search method can be formulated as follows:

The typical initial value for *alpha* is 1 and it is iteratively decreased by a factor of ρ until the sufficient decrease condition is satisfied, which explains the backtracking nature of this algorithm. Computationally, note that for every line search iteration we only need to evaluate the function value in the near neighborhood of the current iterate, \mathbf{x}_k . Usually in implementation, α is decreased only for a maximum number of line search iterations. This simple mechanism is well suited for Newton method, in practice.

To rule out unacceptably short steps, *curvature condition* is used, which requires α_k to satisfy:

Algorithm 1: Backtracking Linesearch Method

Input : \mathbf{x}_k - current iterate;
 \mathbf{p}_k - descent direction;
 $\bar{\alpha}$ - initial step size;
 ρ - backtracking constant $\in (0, 1)$;
 c - amount of decrease control
 parameter, $\in (0, 1)$
Result: α_k - step size to update the parameters
 $\alpha = \bar{\alpha}$
while $f(\mathbf{x}_k + \alpha \mathbf{p}_k) > f(\mathbf{x}_k) + c_1 \alpha \nabla f_k^T \mathbf{p}_k$ **do**
 | $\alpha = \rho \alpha$
end
 $\alpha_k = \alpha$

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k \geq c_2 \nabla f_k^T \mathbf{p}_k$$

Note that this is a derivative of $\phi(\alpha)$, with left hand side indicating the slope at a step length of α_k and right hand side indicating the slope at $\alpha = 0$. Using the curvature condition, we are trying to enforce that the slope of the function at the next iterate \mathbf{x}_{k+1} is greater by a factor of c_2 relative to the slope at \mathbf{x}_k . If the slope, $\phi'(\alpha)$, is largely negative then it is known that we can further reduce the function value by taking a large step along that direction. On the other hand, if it is slightly negative or positive, then there wouldn't be any significant reduction in the function value along that direction. This condition is popularly known as *Wolfe-condition*, and is used typically with quasi-Newton methods. More sophisticated conditions like *Goldstein-conditions* and *polynomial-interpolation* have been proposed in literature to control the amount of minimum decrease required by the step length. Empirically, these more advanced conditions are computationally expensive because of the repeated evaluation of the gradient. Furthermore, in a mini-batch setting, gradients can be noisy, which makes these advanced conditions difficult to use in practice.

1.7 Trust-region Methods

Another common strategy for computing successive iterates in unconstrained optimization problems is the *trust-region* method. Here, we model the objective function at the current iterate, \mathbf{x}_k , using a target function m_k . As we move farther away from the current iterate, \mathbf{x}_k , this approximation might deviate and fail to represent the original objective function well. A trust-region of some radius around \mathbf{x}_k is used to restrict the search of the minimizer of m_k . Mathematically we find the minimizer of m_k around \mathbf{x}_k to use as the descent direction.

$$\min_p m_k(\mathbf{x}_k + p), \text{ where } \mathbf{x}_k + \mathbf{p} \text{ lies inside the trust region}$$

Trust region radius is iteratively changed depending on the candidate solution, \mathbf{p} . If it is a good solution (resulting in expected decrease in the objective function) we increase the trust region and a bad solution will result in decreasing the trust region. Usually the trust region is defined as a ball defined by $\|\mathbf{p}\|_2 \leq \Delta$, where Δ is the trust region radius.

The model m_k is usually the quadratic approximation of the Taylor series approximation of the objective function itself at the current iterate, i.e.,

$$m_k(\mathbf{x}_k + \mathbf{p}) = f_k + \mathbf{p}^T \nabla_k f + \frac{1}{2} \mathbf{p}^T \mathbf{B} \mathbf{p},$$

where f_k is the function value at \mathbf{x}_k , $\nabla_k f$ the gradient vector and \mathbf{B} is the Hessian (or some approximation of it) matrix of the objective function evaluated at \mathbf{x}_k .

Note that, if the model approximation is used as the first-order Taylor series approximation of the objective function, using the euclidean norm ball for the trust region radius, the trust region subproblem becomes:

$$\min_{\mathbf{p}} f_k + \mathbf{p}^T \nabla f_k, \text{ where subject to: } \|\mathbf{p}\|_2 \leq \Delta$$

and the analytical solution to this problem is given by,

$$\mathbf{p}_k = -\frac{\Delta_k \nabla f_k}{\|\nabla f_k\|} \quad (1.9)$$

This is the steepest descent direction, and the step size is determined by the trust region radius. The trust region and line search approaches coincide in this particular case.

A simple iterative trust-region algorithm can be formulated as follows:

Algorithm 2: Trust-region framework

Input : \mathbf{x}_k - current iterate;
 \mathbf{p}_k - descent direction;
 $\bar{\Delta}$ - trust-region radius;
 η - minimum model reduction parameter,
 $\in [0, \frac{1}{4})$;
 c - amount of decrease control parameter,
 $\in (0, 1)$
Result: Δ - step size to update the parameters
Choose $\Delta_0 \in (0, \hat{\Delta})$
for $k = 0, 1, 2, \dots$ **do**
 $\mathbf{p}_k = \min_{\mathbf{p}} m_k(\mathbf{x}_k + \mathbf{p})$
 $\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m_k(\mathbf{0}) - m_k(\mathbf{p}_k)}$
 if $\rho_k > \frac{3}{4}$ **then**
 $\Delta_{k+1} = \min \{2\Delta_k, \bar{\Delta}\}$
 end
 else
 $\Delta_{k+1} = \Delta_k$
 end
 if $\rho_k > \eta$ **then**
 $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$
 end
 else
 $\mathbf{x}_{k+1} = \mathbf{x}_k$
 end
end

2. GPU ACCELERATED SUB-SAMPLED NEWTON’S METHOD

First order methods, which solely rely on gradient information, are commonly used in diverse machine learning (ML) and data analysis (DA) applications. This is attributed to the simplicity of their implementations, as well as low per-iteration computational/storage costs. However, they suffer from significant disadvantages; most notably, their performance degrades with increasing problem ill-conditioning. Furthermore, they often involve a large number of hyper-parameters, and are notoriously sensitive to parameters such as the step-size. By incorporating additional information from the Hessian, second-order methods, have been shown to be resilient to many such adversarial effects. However, these advantages of using curvature information come at the cost of higher per-iteration costs, which in “big data” regimes, can be computationally prohibitive.

In this chapter, we show that, contrary to conventional belief, second-order methods, when implemented appropriately, can be more efficient than first-order alternatives in many large-scale ML/ DA applications. In particular, in convex settings, we consider variants of classical Newton’s method in which the Hessian and/or the gradient are randomly sub-sampled. We show that by effectively leveraging the power of GPUs, such randomized Newton-type algorithms can be significantly accelerated, and can easily outperform state of the art implementations of existing techniques in popular ML/ DA software packages such as TensorFlow. Additionally these randomized methods incur a small memory overhead compared to first-order methods. In particular, we show that for million-dimensional problems, our GPU accelerated sub-sampled Newton’s method achieves a higher test accuracy in milliseconds as compared with tens of seconds for first order alternatives.

2.1 Introduction

Optimization techniques are at the core of many ML/DA applications. First-order methods that rely solely on gradient of the objective function, have been methods of choice in these applications. The scale of commonly encountered problems in typical applications necessitates optimization techniques that are *fast*, i.e., have low per-iteration cost and require few overall iterations, as well as *robust* to adversarial effects such as problem ill-conditioning and hyper-parameter tuning. First-order methods such as stochastic gradient descent (SGD) are widely known to have low per-iteration costs. However, they often require many iterations before suitable results are obtained, and their performance can deteriorate for moderately to ill-conditioned problems. Contrary to popular belief, ill-conditioned problems often arise in machine learning applications. For example, the “vanishing and exploding gradient problem” encountered in training deep neural nets [4], is a well-known and important issue. What is less known is that this is a consequence of the highly ill-conditioned nature of the problem. Other examples include low-rank matrix approximation and spectral clustering involving radial basis function (RBF) kernels when the scale parameter is large [5]. A subtle, yet potentially more serious, disadvantage of most first-order methods is the large number of hyper-parameters, as well as their high sensitivity to parameter-tuning, which can significantly slow down the training procedure and often necessitate many trial and error steps [6, 7].

Newton-type methods use curvature information in the form of the Hessian matrix, in addition to the gradient. This family of methods has not been commonly used in the ML/ DA community because of their high per-iteration costs, in spite of the fact that second-order methods offer a range of benefits. Unlike first-order methods, Newton-type methods have been shown to be highly resilient to increasing problem ill-conditioning [8–10]. Furthermore, second-order methods typically require fewer parameters (e.g., inexactness tolerance for the sub-problem solver or line-search parameters), and are less sensitive to their specific settings [6, 7]. By incorporating

curvature information at each iteration, Newton-type methods scale the gradient such that it is a more suitable direction to follow. Consequently, although their iterations may be more expensive than those of the first-order counterparts, second-order methods typically require much fewer iterations.

In this context, by reducing the cost of each iteration through efficient approximation of curvature, coupled with hardware specific acceleration, one can obtain methods that are *fast* and *robust*. *In most ML applications, this typically translates to achieving a high test-accuracy early on in the iterative process and without significant parameter tuning*; see Section 2.4. This is in sharp contrast with slow-ramping trends typically observed in training with first-order methods, which is often preceded by a lengthy trial and error procedure for parameter tuning. Indeed, the aforementioned properties, coupled with efficiency obtained from algorithmic innovations and implementations that effectively utilize all available hardware resources, hold promise for significantly changing the landscape of optimization techniques used in ML/DA applications.

With the long-term goal of achieving this paradigm shift, we focus on the commonly encountered finite-sum optimization problem

$$\min_{\mathbf{x} \in \mathbb{R}^d} F(\mathbf{x}) \triangleq \sum_{i=1}^n f_i(\mathbf{x}), \quad (2.1)$$

where each $f_i(\mathbf{x})$ is a smooth convex function, representing a loss (or misfit) corresponding to i^{th} observation (or measurement) [11–13]. In many ML applications, F in eq. (3.1) corresponds to the *empirical risk* [14], and the goal of solving eq. (3.1) is to obtain a solution with small generalization error, i.e., high predictive accuracy on “unseen” data. We consider eq. (3.1) at scale, where the values of n and d are large – millions and beyond. In such settings, the mere computation of the Hessian and the gradient of F increases linearly in n . Indeed, for large-scale problems, operations on the Hessian, e.g., matrix-vector products involved in the (approximate) solution of the sub-problems of most Newton-type methods, typically constitute the main com-

putational bottleneck. In such cases, randomized sub-sampling has been shown to be highly successful in reducing computational and memory costs to be effectively *independent* of n . For example, a simple instance of eq. (3.1) is when the functions f_i 's are quadratics, in which case one has an over-constrained least squares problem. For these problems, randomized numerical linear algebra (RandNLA) techniques rely on random sampling, which is used to compute a data-aware or data-oblivious subspace embedding that preserves the geometry of the entire subspace [15]. Furthermore, non-trivial practical implementations of algorithms based on these ideas have been shown to beat state-of-the-art numerical techniques [16–18]. For more general problems, theoretical properties of sub-sampled Newton-type methods, for both convex and non-convex problems of the form in eq. (3.1), have been recently studied in a series of efforts [8–10, 19–22]. *However, for real ML/ DA applications beyond least squares, practical and hardware-specific implementations that can effectively draw upon all available computing resources, are lacking.*

Contributions: Our contributions in this chapter can be summarized as follows: *Through a judicious mix of statistical techniques, algorithmic innovations, and highly optimized GPU implementations, we develop an accelerated variant of the classical Newton’s method that has low per-iteration cost, fast convergence, and minimal memory overhead. In the process, we show that, for solving eq. (3.1), our accelerated randomized method significantly outperforms state of the art implementations of existing techniques in popular ML/DA software packages such as TensorFlow [23], in terms of improved training time, generalization error, and robustness to various adversarial effects.*

This chapter is organized as follows. Section 2.2 provides an overview of related literature. Section 3.2 presents technical background regarding sub-sampled Newton-type methods, Softmax classifier as a practical instance of eq. (3.1), along with a description of the algorithms and their implementation. Section 2.4 compares and contrasts GPU based implementations of sub-sampled Newton-type methods with

first order methods available in TensorFlow. Conclusions and avenues for future work are presented in Section 3.4.

2.2 Related Work

The class of first-order methods includes a number of techniques that are commonly used in diverse ML/DA applications. Many of these techniques have been efficiently implemented in popular software packages. For example, TensorFlow, [23], has enjoyed considerable success among ML practitioners. Among first-order methods implemented in TensorFlow for solving (3.1) are Adagrad [24], RMSProp [25], Adam [26], Adadelata [27], and SGD with/ without momentum [28]. Excluding SGD, the rest of these methods are adaptive, in that they incorporate prior gradients to choose a preconditioner at each gradient step. Through the use of gradient history from previous iterations, these adaptive methods non-uniformly scale the current gradient to obtain an update direction that takes larger steps along the coordinates with smaller derivatives and, conversely, smaller steps along those with larger derivatives. At a high level, these methods aim to capture non-uniform scaling of Newton’s method, albeit, using limited curvature information.

Theoretical properties of a variety of randomized Newton-type methods, for both convex and non-convex problems of the form eq. (3.1), have been recently studied in a series of results, both in the context of ML applications [6, 8–10, 19–22], as well as scientific computing applications [29–31].

GPUs have been successfully used in a variety of ML applications to speed up computations [32–35]. In particular, Raina et al. [33] demonstrate that modern GPUs can far surpass the computational capabilities of multi-core CPUs, and have the potential to address many of the computational challenges encountered in training large-scale learning models. Most relevant to this chapter, Ngiam et al. [35] show that off-the-shelf optimization methods such as Limited memory BFGS (L-BFGS) and Conjugate Gradient (CG), have the potential to outperform variants of SGD in

deep learning applications. It was further demonstrated that the difference in performance between LBFGS/CG and SGD is more pronounced if one considers hardware accelerators such as GPUs. Extending similar results to full-fledged second-order algorithms, such as Newton’s method, is a major motivating factor for our work here.

2.3 Theory, Algorithms and Implementation Details

2.3.1 Sub-Sampled Newton’s Method

For the optimization problem eq. (3.1), in each iteration, consider selecting two sample sets of indices from $\{1, 2, \dots, n\}$, uniformly at random *with* or *without* replacement. Let $\mathcal{S}_{\mathbf{g}}$ and $\mathcal{S}_{\mathbf{H}}$ denote the sample collections, and define \mathbf{g} and \mathbf{H} as

$$\mathbf{g}(\mathbf{x}) \triangleq \frac{n}{|\mathcal{S}_{\mathbf{g}}|} \sum_{j \in \mathcal{S}_{\mathbf{g}}} \nabla f_j(\mathbf{x}), \quad (2.2a)$$

$$\mathbf{H}(\mathbf{x}) \triangleq \frac{n}{|\mathcal{S}_{\mathbf{H}}|} \sum_{j \in \mathcal{S}_{\mathbf{H}}} \nabla^2 f_j(\mathbf{x}), \quad (2.2b)$$

to be the sub-sampled gradient and Hessian, respectively.

It has been shown that, under certain bounds on the size of the samples, $|\mathcal{S}_{\mathbf{g}}|$ and $|\mathcal{S}_{\mathbf{H}}|$, one can, with high probability, ensure that \mathbf{g} and \mathbf{H} are “suitable” approximations to the full gradient and Hessian, in an algorithmic sense [8, 9]. For each iterate $\mathbf{x}^{(k)}$, using the corresponding sub-sampled approximations of the full gradient, $\mathbf{g}(\mathbf{x}^{(k)})$, and the full Hessian, $\mathbf{H}(\mathbf{x}^{(k)})$, we consider *inexact* Newton-type iterations of the form

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k, \quad (2.3a)$$

where \mathbf{p}_k is a search direction satisfying

$$\|\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k + \mathbf{g}(\mathbf{x}^{(k)})\| \leq \theta \|\mathbf{g}(\mathbf{x}^{(k)})\|, \quad (2.3b)$$

for some inexactness tolerance $0 < \theta < 1$ and α_k is the largest $\alpha \leq 1$ such that

$$F(\mathbf{x}^{(k)} + \alpha \mathbf{p}_k) \leq F(\mathbf{x}^{(k)}) + \alpha \beta \mathbf{p}_k^T \mathbf{g}(\mathbf{x}^{(k)}), \quad (2.3c)$$

for some $\beta \in (0, 1)$. The requirement in eq. (3.11c) is often referred to as Armijo-type line-search [1], and eq. (3.11b) is the θ -relative error approximation condition of the exact solution to the linear system

$$\mathbf{H}(\mathbf{x}^{(k)}) \mathbf{p}_k = -\mathbf{g}(\mathbf{x}^{(k)}), \quad (2.4)$$

which is similar to that arising in classical Newton's Method. Note that in (strictly) convex settings, where the sub-sampled Hessian matrix is symmetric positive definite (SPD), conjugate gradient (CG) with early stopping can be used to obtain an approximate solution to eq. (3.12) satisfying eq. (3.11b). It has also been shown [8, 9], that to inherit the convergence properties of the, rather expensive, algorithm that employs the exact solution to eq. (3.12), the inexactness tolerance, θ , in eq. (3.11b) can only be chosen in the order of the inverse of the *square root* of the problem condition number. As a result, even for ill-conditioned problems, only a relatively moderate tolerance for CG ensures that we indeed maintain convergence properties of the exact update (see also examples in Section 2.4). Putting all of these together, we obtain Algorithm 3, which under specific assumptions, has been shown [8, 9] to be globally linearly convergent¹ with problem-independent local convergence rate².

2.3.2 Multi-Class classification

For completeness, we now briefly review multi-class classification using softmax and cross-entropy loss function, as an important instance of the problems of the form described in eq. (3.1). Consider a p dimensional feature vector \mathbf{a} , with corresponding

¹It converges linearly to the optimum starting from any initial guess $\mathbf{x}^{(0)}$.

²If the iterates are close enough to the optimum, it converges with a constant linear rate independent of the problem-related quantities.

Algorithm 3: Sub-Sampled Newton Method

Input : Initial iterate, $\mathbf{x}^{(0)}$
Parameters: $0 < \epsilon, \beta, \theta < 1$
1 **foreach** $k = 0, 1, 2, \dots$ **do**
2 Form $\mathbf{g}(\mathbf{x}^{(k)})$ as in eq. (2.2a)
3 Form $\mathbf{H}(\mathbf{x}^{(k)})$ as in eq. (2.2b)
4 **if** $\|\mathbf{g}(\mathbf{x}^{(k)})\| < \epsilon$ **then**
 | STOP
 end
5 Update $\mathbf{x}^{(k+1)}$ as in eq. (2.3)
end

labels b , which can belong to one of C classes. In such a classifier, the probability that \mathbf{a} belongs to a class $c \in \{1, 2, \dots, C\}$ is given by $\Pr(b = c \mid \mathbf{a}, \mathbf{w}_1, \dots, \mathbf{w}_C) = e^{\langle \mathbf{a}, \mathbf{w}_c \rangle} / \sum_{c'=1}^C e^{\langle \mathbf{a}, \mathbf{w}_{c'} \rangle}$, where $\mathbf{w}_c \in \mathbb{R}^p$ is the weight vector corresponding to class c . Since probabilities must sum to one, there are in fact only $C - 1$ degrees of freedom. Consequently, by defining $\mathbf{x}_c \triangleq \mathbf{w}_c - \mathbf{w}_C$, $c = 1, 2, \dots, C - 1$, for training data $\{\mathbf{a}_i, b_i\}_{i=1}^n \subset \mathbb{R}^p \times \{1, \dots, C\}$, the cross-entropy loss function for $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \dots; \mathbf{x}_{C-1}] \in \mathbb{R}^{(C-1)p}$ can be written as

$$\begin{aligned}
 F(\mathbf{x}) &\triangleq F(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}) \\
 &= \sum_{i=1}^n \left(\log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) - \sum_{c=1}^{C-1} \mathbf{1}(b_i = c) \langle \mathbf{a}_i, \mathbf{x}_c \rangle \right). \quad (2.5)
 \end{aligned}$$

Note that here, $d = (C - 1)p$. It then follows that the full gradient of F with respect to \mathbf{x}_c is

$$\nabla_{\mathbf{x}_c} F(\mathbf{x}) = \sum_{i=1}^n \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \mathbf{1}(b_i = c) \right) \mathbf{a}_i. \quad (2.6)$$

Similarly, for the full Hessian of F , we have

$$\nabla_{\mathbf{x}_c, \mathbf{x}_c}^2 F = \sum_{i=1}^n \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{a}_i^T, \quad (2.7a)$$

and for $\hat{c} \in \{1, 2, \dots, C-1\} \setminus \{c\}$, we get

$$\nabla_{\mathbf{x}_c, \mathbf{x}_{\hat{c}}}^2 F = \sum_{i=1}^n \left(-\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_{\hat{c}} + \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{a}_i^T. \quad (2.7b)$$

Sub-sampled variants of the gradient and Hessian are obtained similarly. Finally, after training phase, a new data \mathbf{a} is classified as

$$b = \arg \max \left\{ \left\{ \frac{e^{\langle \mathbf{a}, \mathbf{x}_c \rangle}}{\sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle}} \right\}_{c=1}^{C-1}, 1 - \frac{e^{\langle \mathbf{a}, \mathbf{x}_1 \rangle}}{\sum_{c'=1}^C e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle}} \right\}.$$

Numerical Stability

To avoid over-flow in the evaluation of exponential functions in (3.2), we use the “Log-Sum-Exp” trick [36]. Specifically, for each data point \mathbf{a}_i , we first find the maximum value among $\langle \mathbf{a}_i, \mathbf{x}_c \rangle$, $c = 1, \dots, C-1$. Define

$$M(\mathbf{a}) = \max \left\{ 0, \langle \mathbf{a}, \mathbf{x}_1 \rangle, \langle \mathbf{a}, \mathbf{x}_2 \rangle, \dots, \langle \mathbf{a}, \mathbf{x}_{C-1} \rangle \right\}, \quad (2.8)$$

and

$$\alpha(\mathbf{a}) := e^{-M(\mathbf{a})} + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle - M(\mathbf{a})}. \quad (2.9)$$

Note that $M(\mathbf{a}) \geq 0, \alpha(\mathbf{a}) \geq 1$. Now, we have $1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} = e^{M(\mathbf{a}_i)} \alpha(\mathbf{a}_i)$. For computing (3.2), we use $\log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) = M(\mathbf{a}_i) + \log(\alpha(\mathbf{a}_i))$. Similarly, for (2.6) and (2.7), we use

$$\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} = \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle - M(\mathbf{a}_i)}}{\alpha(\mathbf{a}_i)}.$$

Note that in all these computations, we are guaranteed to have all the exponents appearing in all the exponential functions to be negative, hence avoiding numerical over-flow.

Hessian Vector Product

Given a vector $\mathbf{v} \in \mathbb{R}^d$, we can compute the Hessian-vector product without explicitly forming the Hessian. For notational simplicity, define

$$h(\mathbf{a}, \mathbf{x}) := \frac{e^{\langle \mathbf{a}, \mathbf{x} \rangle - M(\mathbf{x})}}{\alpha(\mathbf{a})},$$

where $M(\mathbf{x})$ and $\alpha(\mathbf{x})$ were defined in eqs. (3.13) and (3.14), respectively. Now using matrices

$$\mathbf{V} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{v}_1 \rangle & \langle \mathbf{a}_1, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_1, \mathbf{v}_{C-1} \rangle \\ \langle \mathbf{a}_2, \mathbf{v}_1 \rangle & \langle \mathbf{a}_2, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_2, \mathbf{v}_{C-1} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{v}_1 \rangle & \langle \mathbf{a}_n, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_n, \mathbf{v}_{(C-1)} \rangle \end{bmatrix}_{n \times (C-1)}, \quad (2.10)$$

and

$$\mathbf{W} = \begin{bmatrix} h(\mathbf{a}_1, \mathbf{x}_1) & h(\mathbf{a}_1, \mathbf{x}_2) & \dots & h(\mathbf{a}_1, \mathbf{x}_{C-1}) \\ h(\mathbf{a}_2, \mathbf{x}_1) & h(\mathbf{a}_2, \mathbf{x}_2) & \dots & h(\mathbf{a}_2, \mathbf{x}_{C-1}) \\ \vdots & \vdots & \ddots & \vdots \\ h(\mathbf{a}_n, \mathbf{x}_1) & h(\mathbf{a}_n, \mathbf{x}_2) & \dots & h(\mathbf{a}_n, \mathbf{x}_{C-1}) \end{bmatrix}_{n \times (C-1)}, \quad (2.11)$$

we compute

$$\mathbf{U} = \mathbf{V} \odot \mathbf{W} - \mathbf{W} \odot \left(((\mathbf{V} \odot \mathbf{W}) \mathbf{e}) \mathbf{e}^T \right), \quad (2.12)$$

to get

$$\mathbf{H}\mathbf{v} = \text{vec}(\mathbf{A}^T \mathbf{U}), \quad (2.13)$$

where $\mathbf{v} = [\mathbf{v}_1; \mathbf{v}_2; \dots; \mathbf{v}_{C-1}] \in \mathbb{R}^d$, $\mathbf{v}_i \in \mathbb{R}^p$, $i = 1, 2, \dots, C-1$, $\mathbf{e} \in \mathbb{R}^{C-1}$ is a vector of all 1's, and each row of the matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$ is a row vector corresponding to the i^{th} data point, i.e., $\mathbf{A}^T = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n]$.

Remark 1 *Note that the memory overhead of our accelerated randomized sub-sampled Newton's method is determined by matrices \mathbf{U} , \mathbf{V} , and \mathbf{W} , whose sizes are dictated by the Hessian sample size, $|\mathcal{S}_{\mathbf{H}}|$, which is much less than n . This small memory overhead enables our Newton-type method to scale to large problems, inaccessible to traditional second order methods.*

2.3.3 Implementation Details

We present a brief overview of the algorithmic machinery involved in the implementation of iterations described in eq. (2.3) and applied to the function defined in eq. (3.2) with an added ℓ_2 regularization term, i.e., $F(\mathbf{x}) + \lambda \|\mathbf{x}\|^2/2$. Here, λ is the regularization parameter. We note that for all the algorithms in this section, we assume that matrices are stored in column-major ordering.

Conjugate Gradient For the sake of self-containment, in Algorithm 4, we depict a slightly modified implementation of the classical CG, to approximately solve the linear system in eq. (3.12), i.e., $\mathbf{H}\mathbf{p} = -\mathbf{g}$, to satisfy eq. (3.11b). This routine takes a function (pointer), $H(\cdot)$, which computes the matrix-vector product as $H(\mathbf{v}) = \mathbf{H}\mathbf{v}$, as well as the right-hand side vector, \gg . Lines 2, and 3 initializes the residual vector

Algorithm 4: Conjugate-Gradient

Input :

$H(\cdot)$ - Pointer to Algorithm 9 to compute
Hessian-vector product, $H(\mathbf{v}) = \mathbf{H}\mathbf{v}$
 \mathbf{g} - Gradient

Parameters:

θ - Relative residual tolerance
 T - Maximum no. of iterations

Result: \mathbf{p}_{best} , an approximate solution to $\mathbf{H}\mathbf{p} = -\mathbf{g}$

```

1  $\mathbf{p}_0 = \mathbf{0}$ 
2  $\mathbf{r}_0 = -\mathbf{g}$  // initial residual vector
3  $\mathbf{s}_0 = \mathbf{r}_0$  // initial search direction
4  $\mathbf{p}_{\text{best}} = \mathbf{s}_0$  // best solution so far
5  $\mathbf{r}_{\text{best}} = \mathbf{r}_0$ 
6 foreach  $k = 0, 1, \dots, T$  do
7    $\alpha_k = \mathbf{r}_k^T \mathbf{r}_k / \mathbf{s}_k^T H(\mathbf{s}_k)$ 
8    $\mathbf{p}_{k+1} = \mathbf{p}_k + \alpha_k \mathbf{s}_k$ 
9    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k H(\mathbf{s}_k)$ 
10  if  $\|\mathbf{r}_{k+1}\| \leq \|\mathbf{r}_{\text{best}}\|$  then
11     $\mathbf{r}_{\text{best}} = \mathbf{r}_{k+1}$ 
12     $\mathbf{p}_{\text{best}} = \mathbf{p}_{k+1}$ 
13  end
14  if  $\|\mathbf{r}_{k+1}\| \leq \theta \|\mathbf{g}\|$  then
15    break
16  end
17   $\mathbf{s}_{k+1} = \mathbf{r}_{k+1} + \frac{\|\mathbf{r}_{k+1}\|_2^2}{\|\mathbf{r}_k\|_2^2} \mathbf{s}_k$ 
18 end

```

\mathbf{r} , and search direction \mathbf{s} , respectively, while the best residual is initialized on line 5. Iterations start on line 6, which maintains a counter for maximum allowed iterates to compute. Step-size α for CG iterations is computed on line 7, which is used to update the solution vector, \mathbf{p} and residual vector, \mathbf{r} . The minor modification comes from line 10, which stores the best solution vector thus far. The termination condition eq. (3.11b) is evaluated on line 11. Finally, the search direction, \mathbf{s} , is updated in line 12.

Algorithm 5: Line Search

Input :

\mathbf{x} - Current point
 \mathbf{p} - Newton's direction
 $F(\cdot)$ - Function pointer
 $\mathbf{g}(\mathbf{x})$ - Gradient

Parameters:

α - Initial step size
 $0 < \beta < 1$ - Cost function reduction constant
 $0 < \rho < 1$ - back-tracking parameter
 i_{\max} - maximum line search iterations

```

1  $\alpha = 1$ 
2  $i = 0$ 
3 while  $F(\mathbf{x} + \alpha\mathbf{p}) > F(\mathbf{x}) + \alpha\beta\mathbf{p}^T\mathbf{g}(\mathbf{x})$  do
4   if  $i > i_{\max}$  then
5     break
6   end
7    $i = i + 1$ 
7    $\alpha \leftarrow \rho\alpha$ 
end

```

Line Search method We use a simple back-tracking line search, shown in Algorithm 12 for computing the step size in eq. (3.11c). Step size, α , is initialized in line 1, which is typically set to the “natural” step-size of Newton’s method, i.e., $\alpha = 1$. Iterations start at line 3 by checking the exit criteria, and if required, successively decreasing the step size until the “loose” termination condition is met. In each of these iterations, if the objective function does not reduce by a specified amount, β , step size is reduced by a fraction, ρ , of its current value, until the termination condition is met or specified iterations have been exceeded. It has been shown [8] that this process will terminate after a certain number of iterations, i.e., we are always guaranteed to have $\alpha \geq \alpha_0 > 0$ for some fixed α_0 .

CUDA utility functions Bulk of the work in evaluating the softmax function is done by *ComputeExp* subroutine, shown in Algorithm 6. This function takes a

Algorithm 6: ComputeExp

input : $\hat{\mathbf{A}}$ - where $\hat{\mathbf{A}}_{i,j} = \mathbf{a}_i^T \mathbf{x}_j, \forall i \in \{1 \dots n\}, \forall j \in \{1 \dots C-1\}$
 \mathbf{b} - Training classes
maxPart- memory pointer to store eq. (2.14)
sumExpPart- memory pointer to store eq. (2.15)
linearPart- memory pointer to store eq. (2.16)
n - no. of rows in $\hat{\mathbf{A}}$
C - no. of classes
output: maxPart, sumExpPart, linearPart

```

1 Init. idx ;                                // thread-id
  if idx < n then
2   i ← idx % n ;                            // row no.
3   maxParti = linearParti = sumExpParti = 0
4   foreach j in 1 : C - 1 do
      if maxParti <  $\hat{\mathbf{A}}_{i,j}$  then
      | maxParti =  $\hat{\mathbf{A}}_{i,j}$ 
      end
    end
5   foreach j in 1 : C - 1 do
6   | if  $\mathbf{b}_i == j$  then
      | linearParti =  $\hat{\mathbf{A}}_{i,j}$ 
      end
7   | sumExpParti += exp ( $\hat{\mathbf{A}}_{i,j} - \text{maxPart}_i$ )
    end
  end
end

```

matrix, as an input, and computes the following data structures: “maxPart_{*i*}” stores the maximum component in each of the rows of the input matrix, “linearPart_{*i*}” stores the partial summation of the term $\sum_{j=1}^{C-1} \mathbf{1}(\mathbf{b}_i = j)(\mathbf{a}_i^T \mathbf{x}_j)$, and “sumExpPart” stores the summation in eq. (2.15). Input matrix, $\hat{\mathbf{A}} \in \mathbb{R}^{n \times (C-1)}$, is the product of \mathbf{A} and \mathbf{X} matrices, where $\mathbf{X} \in \mathbb{R}^{p \times (C-1)}$ is a matrix whose i^{th} column is $\mathbf{x}_i \in \mathbb{R}^p$, i.e., $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}]$, and \mathbf{A} is as in eq. (3.18). Line 1 initializes the *idx*, thread-id of a given thread. In the for loop in line 4, we compute the maximum coordinate per row of the input matrix, and the result is stored in array “maxPart”. Line 5

computes “linearPart” and “sumExpPart” arrays, which are later used by functions invoking this algorithm.

Algorithm 7: ComputeFX

input : **A**- Training features
 b - Training classes
 x - Weights vector
 λ - Regularization
 n - no. of rows in **A**
 p - no. of cols in **A**
 C - no. of classes

output: $F(\mathbf{x})$ - Objective function evaluated at **x**

- 1 Initialize **maxPart**, **linearPart**, **sumExpPart** to store
 eqs. (2.14)–(2.16),
- 2 Form $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}]_{p \times (C-1)}$
- 3 $\hat{\mathbf{A}} = \mathbf{A} \times \mathbf{X}$; // matrix-matrix multiplication
- 4 ComputeExp($\hat{\mathbf{A}}$, **b** , **maxPart**, **sumExpPart**, **linearPart**, n, C)
- 5 Reduce(**linearPart**, pLin, n, $t(z) = z$)
- 6 Reduce(**maxPart**, pMax, n, $t(z) = z$)
- 7 Reduce(**sumExpPart**, pExp, n, $t(z) = z$)
- 8 $\text{temp} \leftarrow \mathbf{maxPart} + \mathbf{sumExpPart}$
- 9 Reduce(**temp**, pLog, n, $t(z) = \log(z)$)
- 10 $F(\mathbf{x}) \leftarrow (\text{pMax} + \text{pLog} - \text{pLin}) + \lambda \|\mathbf{x}\|^2 / 2$

Softmax function evaluation Subroutine *ComputeFX*, shown in Algorithm 7, describes the evaluation of objective function at a given point, $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \dots; \mathbf{x}_{C-1}] \in \mathbb{R}^d$. Line 2 initializes the memory to store partial results, and line 3 computes the matrix-matrix product between training set, **A**, and weight matrix, **X**. By invoking the CUDA function, *ComputeExp*, we compute the partial results, *maxPart*, *sumExpPart*, *linearPart*, as described in eqs. (2.14)–(2.16). Lines 5, 6 and, 7 compute the sum of the temporary arrays, and store the partial results in *pLin*, *pMax*, *pExp*, respectively. *Reduce* operation takes a transformation function, $t(\cdot)$, which is applied to the input argument before performing the summation. *Reduce* is a well known

function and many highly optimized implementations are readily available. We use a variation of the algorithm described in [37]. $pLog$ is computed at line 9. Finally, the objective function value is computed at line 10, by adding intermediate results, $pLin$, $pMax$, $pExp$, $pLog$ and the regularization term, i.e.,

$$\begin{aligned} F(\mathbf{x}) &= (pMax + pLog - pLin) + \frac{\lambda}{2} \|\mathbf{x}\|^2 \\ &= \sum_{i=1}^n (\maxPart_i + \logPart_i - \text{linearPart}_i) + \frac{\lambda}{2} \|\mathbf{x}\|^2, \end{aligned}$$

where

$$\maxPart_i = M(\mathbf{a}_i) \quad (\text{cf. eq. (3.13)}), \quad (2.14)$$

$$\text{sumExpPart}_i = \sum_{c=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle - \maxPart_i}, \quad (2.15)$$

$$\text{linearPart}_i = \sum_{c=1}^{C-1} \mathbf{1}(\mathbf{b}_i = c) \langle \mathbf{a}_i, \mathbf{x}_c \rangle, \quad (2.16)$$

$$\logPart_i = \log(e^{-\maxPart_i} + \text{sumExpPart}_i). \quad (2.17)$$

Algorithm 8: Compute ∇F

input : \mathbf{A} - Training features

\mathbf{b} - Training classes

\mathbf{x} - Weights vector

λ - Regularization

output: $\nabla F(\mathbf{x})$ - gradient evaluated at \mathbf{x}

1 Initialize $\mathbf{BInd}_{(n \times C-1)}$

2 Form $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}]_{p \times (C-1)}$

3 Compute $\mathbf{BInd}_{i,c} = \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{z=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_z \rangle}} - \mathbf{1}(\mathbf{b}_i = c)$, similar to Alg. 6

4 $\nabla F(\mathbf{x}) \leftarrow \text{vec}(\mathbf{A}^T \mathbf{BInd} + \lambda \mathbf{X})$

Softmax gradient evaluation Subroutine *Compute ∇F* , shown in Algorithm 8, describes the computation of $\nabla F(x)$. Line 1 initializes the memory to store temporary

results. Algorithm 6 can be easily modified to compute **BInd**. Line 4 computes the gradient of the objective function by matrix multiplication and addition of the regularization term.

Algorithm 9: Compute Hessian-Vector Product, $\nabla^2 F(\mathbf{x})\mathbf{q}$

input : **A**- Training dataset
 λ - Regularization
 \mathbf{x} - Weights vector
 \mathbf{q} - Vector to compute $\nabla^2 F(\mathbf{x})\mathbf{q}$
 n - no. of sample points
 p - no. of features
 C - no. of classes

output: **Hq**: $\nabla^2 F(\mathbf{x})\mathbf{q}$, Hessian-vector product

```

1 Init. idx ;                                // thread-id
2 Form  $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{C-1}]_{p \times (C-1)}$ 
3  $\mathbf{V} = \mathbf{A} \times \mathbf{Q}$ 
4  $\mathbf{W} \leftarrow$  compute as shown in (3.16), similar to kernel
  Alg.6
5  $\mathbf{U} \leftarrow \text{ComputeU}(\mathbf{V}, \mathbf{W}, n, p, C)$ 
6  $\mathbf{Hq} \leftarrow \text{vec}(\mathbf{A}^T \mathbf{U} + \lambda \mathbf{Q})$ 

```

Softmax Hessian-vector evaluation For a given vector, \mathbf{q} , Algorithm 9, computes the *Hessian-vector* product, $\nabla^2 F(\mathbf{x})\mathbf{q}$. Algorithm 9 is heavily used in CG to solve the linear system $\mathbf{H}\mathbf{x} = -\mathbf{g}$. Line 1 computes \mathbf{V} , as shown in eq. (3.15), a matrix multiplication operation. Line 4 computes \mathbf{W} using a function similar to Algorithm 6, and \mathbf{U} is computed using Alg. 10 at line 5. Finally \mathbf{Hq} is computed by multiplying \mathbf{A}^T and \mathbf{U} , and adding the regularization term in line 6.

2.4 Experimental Results

We present a comprehensive evaluations of the performance of Newton-type methods presented in this chapter. We compare our methods to various first-order methods

Algorithm 10: ComputeU

```

input :  $\mathbf{V}$ - matrix  $\mathbf{V}$  as in eq. (3.15)
          $\mathbf{W}$ - matrix  $\mathbf{W}$  as in eq. (3.16)
          $n$  - no. of sample points
          $p$  - no. of features
          $C$  - no. of classes

output:  $\mathbf{U}$  : matrix  $\mathbf{U}$  as shown in (3.17)

Initialize  $\text{idx}$  ;                                // thread-id
sum = 0
if  $\text{idx} < n$  then
     $i = \text{idx} \% n$  ;                                // row no.
    foreach  $j$  in  $1 : C - 1$  do
        | sum +=  $\mathbf{V}_{i,j} \times \mathbf{W}_{i,j}$ ;
    end
    foreach  $j$  in  $1 : C - 1$  do
        |  $\mathbf{U}_{i,j} = \mathbf{V}_{i,j} \times \mathbf{W}_{i,j} - \mathbf{W}_{i,j} \times \text{sum}$ ;
    end
end

```

– SGD with momentum (henceforth referred to as Momentum) [28], Adagrad [24], Adadelta [27], Adam [26] and RMSProp [25] as implemented in Tensorflow [23]. We describe our benchmarking setup, software used for development, and provide a detailed analysis of the results. The code used in this work along with the processed datasets are publicly available [38]. Additionally, raw datasets are also available from the UCI Machine Learning Repository [39].

Table 2.1.: Description of the datasets. L indicates the Lipschitz Constant of the dataset.

Classification	Dataset	Train Size (n)	Test Size	Features (p)	Classes (C)	L
Multi-Class	Coverttype	450000	131012	54	7	1.92
	Drive Diagnostics	50000	8509	48	11	3.95
	MNIST	38000	38000	785	10	28.67
	CIFAR-10	50000	10000	3072	10	534.92
	Newsgroups20	10142	1127	53975	20	128.79
Binary	Gisette	6000	6500	5000	2	751.19
	Real-Sim	65078	7231	20958	2	206.76

2.4.1 Experimental Setup and Data

Newton-type methods are implemented in C/C++ using CUDA/8.0 toolkit. For matrix operations, matrix-vector, and matrix-matrix operations, we use cuBLAS and cuSparse libraries. First order-methods are implemented using Tensorflow/1.2.1 python scripts. All results are generated using an Ubuntu server with 256GB RAM, 48-core Intel Xeon E5-2650 processors, and Tesla P100 GPU cards. For all of our experiments, we consider the ℓ_2 -regularized objective $F(\mathbf{x}) + \lambda \|\mathbf{x}\|^2/2$, where F is as in eq. (3.2) and λ is the regularization parameter. Seven real datasets are used for performance comparisons. Table 2.1 presents the datasets used, along with the *Lipschitz* continuity constant of $\nabla F(\mathbf{x})$, denoted by L . Recall that, an (over-estimate) of the *condition-number* of the problem, as defined in [8], can be obtained by $(L + \lambda)/\lambda$. As it is often done in practice, we first normalize the datasets such that each column of the data matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$ (as defined in Section 3.2.1), has Euclidean norm one. This helps with the conditioning of the problem. The resulting dataset is, then, split into training and testing sets, as shown in the Table 2.1.

2.4.2 Parameterization of Various Methods

The Lipschitz constant, L , is used to estimate the learning rate (step-size) for first order methods. For each dataset, we use a range of learning rates from $10^{-6}/L$ to $10^6/L$, in increments of 10, a total of 13 step sizes, to determine the best performing learning rate (one that yields the maximum test accuracy). Rest of the hyper-parameters required by first-order methods are set to the default values, as recommended in Tensorflow. Two batch sizes are used for first-order methods: a small batch size of 128 (empirically, it has been argued that smaller batch sizes might lead to better performance [40]), and a larger batch size of 20% of the dataset. For Newton-type methods, when the gradient is sampled, its sample size is set to $|\mathcal{S}_g| = 0.2n$.

We present results for two implementations of second-order methods: (a) *Full-Newton*, the classical Newton-CG algorithm [1], which uses the exact gradient and Hessian, and (b) *SubsampledNewton*, sub-sampled variant of Newton-CG using uniform sub-sampling for gradient/Hessian approximations. When compared with first-order methods that use batch size of 128, *SubsampledNewton* uses full gradient and 5% for Hessian sample size, referred to as *SubsampledNewton-100*. When first-order methods’ batch size is set to 20%, *SubsampledNewton* uses 20% for gradient and 5% for Hessian sampling, referred to as *SubsampledNewton-20*. CG-tolerance is set to 10^{-4} . Maximum CG iterations is 10 for all of the datasets except *Drive Diagnostics* and *Gisette*, for which it is 1000. λ is set to 10^{-3} and we perform 100 iterations (epochs) for each dataset.

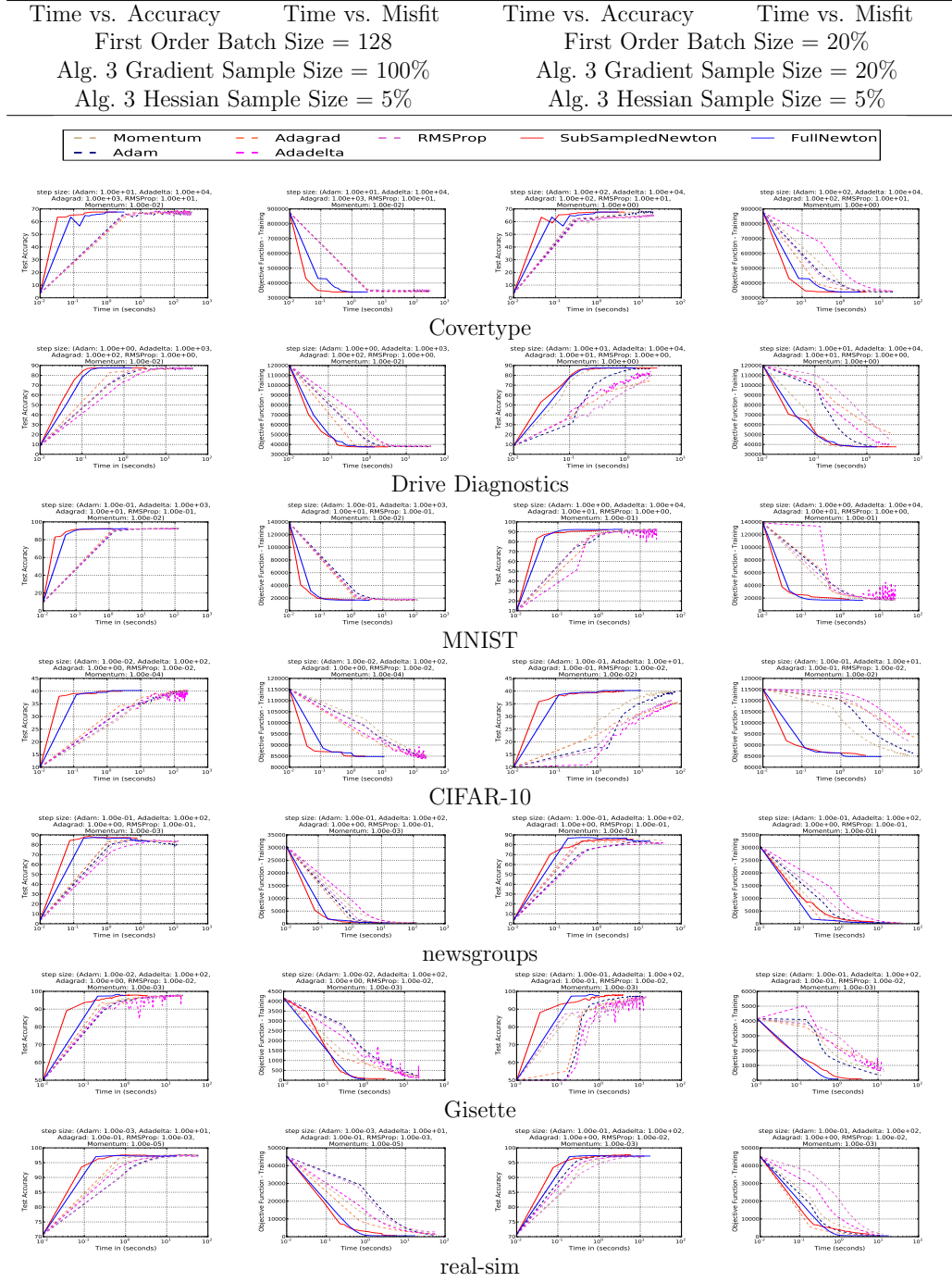
2.4.3 Computing Platforms

For benchmarking first order methods with batch size 128, we use CPU-cores only and for the larger batch size 1-GPU and 1-CPU-core are used. For brevity we only present the best performance results (lowest time-per-epochs); see 2.7 for more detailed discussion on performance results on various compute platforms. Newton-type methods always use 1-GPU and 1-CPU-core for computations.

2.4.4 Performance Comparisons

Table 2.2 presents all the performance results. Columns 1 and 3 show the plots for *cumulative-time vs. test-accuracy* and columns 2 and 4 plot the numbers for *cumulative-time vs. objective function (training)*. Please note that x-axis in all the plots is in “log-scale”.

Table 2.2.: Performance comparison between first-order and second-order methods. *FullNewton* uses the entire dataset for gradient and Hessian evaluations.



Covertypes Dataset

The first row in Table 2.2 shows the plots for *Covertypes* dataset. From the first two columns (batch size 128), we note the following: (i) Newton-type methods minimize the objective function to $\approx 3.4e5$ in a smaller time interval (*FullNewton*: 0.9 secs, *SubsampledNewton-20*: 0.24 secs), compared to first-order alternatives (Adadelta - 91 secs, Adagrad - 183 secs, Adam - 57 secs, Momentum - 285 secs, RMSProp - 40 secs); (ii) Compared to first order algorithms, Newton-type methods achieve equivalent test accuracy, 68%, in a significantly shorter time interval, i.e., 0.9 secs compared with tens of seconds for first order methods (Adadelta: 201 secs, Adagrad: 72 secs, Adam: 285 secs, Momentum: 128 secs, RMSProp: 111 secs); (iii) *SubsampledNewton-100* achieves relatively higher test accuracy earlier compared to the *FullNewton* method in a relatively short time interval (*FullNewton*: 68% in 1.5 secs, *SubsampledNewton-100*: 68% in 204 millisecs). For well-conditioned problems (such as this one), a relaxed *CG-tolerance* and small sample sizes (5% Hessian sample size) yield desirable results quickly.

Columns 3 and 4 present the performance of first-order methods with batch size 20%. Randomized Newton method, *SubsampledNewton-20*, achieves higher test accuracy, 68%, in a very short time, 1.05 secs, compared to any of the first order methods as shown in column 3 (Adadelta: 65% in 21 secs, Adagrad: 65% in 19 secs, Adam: 68% in 20 secs, Momentum: 68% in 18 secs, RMSProp: 65% in 21 secs). First order methods, with batch size 20%, are executed on GPUs resulting in smaller time-per-epoch; see 2.7. This can be attributed to processing larger batches of the dataset by the GPU-cores, yielding higher efficiency.

Drive Diagnostics Dataset

Results for the *Drive Diagnostics* dataset are shown in the second row of Table 2.2. These plots clearly indicate that Newton-type methods achieve their lowest objective function value, $3.75e4$, much earlier compared to first order methods (*FullNewton* -

1.3 secs, *SubsampledNewton-20* - 0.8 secs, *SubsampledNewton-100* - 0.2 secs). Corresponding times for batch size 128 for first order methods are : Adadelta - 16 secs, Adagrad - 34 secs, Adam - 25 secs, Momentum - 32 secs, RMSProp - 35 secs (lowest objective function value for these methods are $\approx 3.8e5$). For batch size 20%, except for Adadelta and Momentum, other first order methods achieve their lowest objective function values, which are significantly higher compared to Newton-type methods, in ≈ 3 seconds. Momentum is the only first order method that achieves almost equivalent objective function value, $3.8e5$ in 0.6 seconds, as Newton-type methods.

All first order methods, with batch size 128, achieve test accuracy of 87% which is same as Newton-type methods but take much longer: *FullNewton* - 0.2 secs, *SubsampledNewton-20* - 0.3 secs, *SubsampledNewton-100* - 0.15 secs vs. Adadelta - 30 secs, Adagrad - 36 secs, Adam - 7 secs, Momentum - 32 secs, RMSProp - 7 secs. Here, except Momentum, none of the first order methods with batch size 20% achieve 87% test accuracy in 100 epochs.

MNIST and CIFAR-10 Datasets

Rows 3 and 4 in Table 2.2 present plots for *MNIST* and CIFAR-10 datasets, respectively. Regardless of the batch size, Newton-type methods clearly outperform first-order methods. For example, with *MNIST* dataset, all the methods achieve a test accuracy of 92%. However, Newton-type methods do so in ≈ 0.2 seconds, compared to ≈ 4 seconds for first order methods with batch size of 128.

CIFAR results are shown in row 4 of Table 2.2. We clearly notice that first order methods, with batch size 128, make slow progress towards achieving their lowest objective function value (and test accuracy) taking almost 100 seconds to reach $8.4e4$ (40% test accuracy). Newton-type methods achieve these values in significantly shorter time (*FullNewton* - 10 seconds, *SubsampledNewton-20* - 4.2 seconds, *SubsampledNewton-100* - 2.6 seconds). The slow progress of first order methods is much more pronounced when batch size is set to 20%. Only Adam and Momentum

methods achieve a test accuracy of $\approx 40\%$ in 100 epochs (taking ≈ 60 seconds). Note that *CIFAR-10* represents a relatively *ill*-conditioned problem. As a result, in terms of lowering the objective function on *CIFAR-10*, first-order methods are negatively affected by the ill-conditioning, whereas all Newton-type methods show a great degree of robustness. This demonstrates the versatility of Newton-type methods for solving problems with various degrees of ill-conditioning.

Newsgroups20 Dataset

Plots in row 5 of Table 2.2 correspond to *Newsgroups20* dataset. This is a sparse dataset, and the largest in the scope of this work (the Hessian is $\approx 1e6 \times 1e6$). Here, *FullNewton* and *SubsampledNewton-100* achieve, respectively, 87.22% and 88.46% test accuracy in the first few iterations. Smaller batch sized first order methods can only achieve a maximum test accuracy of 85% in 100 epochs. Note that average time per epoch for first order methods is ≈ 1 sec compared to 75 millisecs for *SubsampledNewton-100* iteration. When 20% gradient is used, as shown in column 3, we notice that the *SubsampledNewton-20* method starts with a lower test accuracy of $\approx 80\%$ in the 5th iteration and slowly ramps up to 85.4% as we near the allotted number of iterations. This can be attributed to a smaller gradient sample size, and sparse nature of this dataset.

Gisette and Real-Sim Datasets

Rows 6 and 7 in Table 2.2 show results for *Gisette* and *Real-Sim* datasets, respectively. *FullNewton* method for *Gisette* dataset converges in 11 iterations and yields 98.3% test accuracy in 0.6 seconds. *SubsampledNewton-100* takes 34 iterations to reach 98% test accuracy, whereas first order counterparts, except Momentum method, can achieve 97% test accuracy in 100 iterations. When batch size is set to 20%, we notice that all first order methods make slow progress towards achieving lower objective function values. Noticeably, none of the first order methods can lower

the objective function value to a level achieved by Newton-type methods, which can be attributed to the ill-conditioning of this problem; see Table 2.1.

For *Real-Sim* dataset, relative to first order methods and regardless of batch size, we clearly notice that Newton-type methods achieve similar or lower objective function values, in a comparable or lower time interval. Further, *FullNewton* achieves 97.3% in the 2nd iteration whereas it takes 11 iterations for *SubsampledNewton-20*.

2.4.5 Sensitivity to Hyper-Parameter Tuning

The “biggest elephant in the room” in optimization using, almost all, first-order methods is that of fine-tuning of various underlying hyper-parameters, most notably, the step-size [6, 7]. Indeed, the success of most such methods is tightly intertwined with many trial and error steps to find a proper parameter settings. It is highly unusual for these methods to exhibit acceptable performance on the first try, and it often takes many trials and errors before one can see reasonable results. In fact, the “true training time”, which almost always includes the time it takes to appropriately tune these parameters, can be frustratingly long. In contrast, second-order optimization methods involve much less parameter tuning, and are less sensitive to specific choices of their hyper-parameters [6, 7].

Here, to further highlight such issues, we demonstrate the sensitivity of several first-order methods with respect to their learning rate. Figure 2.1 shows the results of multiple runs of SGD with Momentum, Adagrad, RMSProp and Adam on *News-groups20* dataset with several choices of step-size. Each method is run 13 times using step-sizes in the range $10^{-6}/L$ to $10^6/L$, in increments of 10, where L is the Lipschitz constant; see Table 2.1.

It is clear that small step-sizes can result in stagnation, whereas large step sizes can cause the method to diverge. Only if the step-size is within a particular and often narrow range, which greatly varies across various methods, one can see reasonable performance.

Remark 2 For some first-order methods, e.g., momentum based, line-search type techniques simply cannot be used. For others, the starting step-size for line-search is, almost always, a priori unknown. This is sharp contrast with randomized Newton-type methods considered here, which come with a priori “natural” step-size, i.e., $\alpha = 1$, and furthermore, only occasionally require the line-search to intervene; see [8, 9] for theoretical guarantees in this regard.

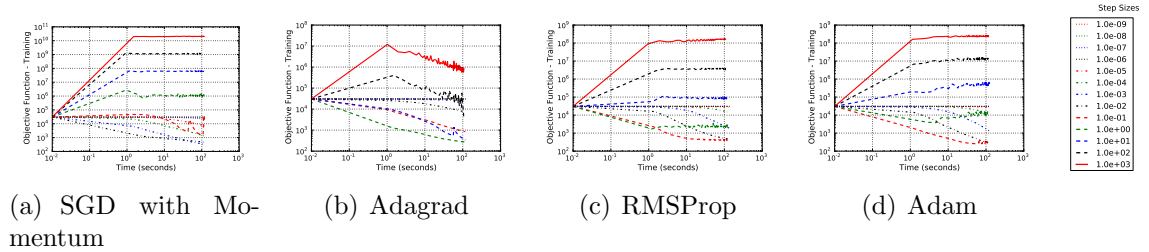


Fig. 2.1.: Sensitivity of various first-order methods with respect to the choice of the step-size, i.e., learning-rate. It is clear that, too small a step-size can lead to slow convergence, while larger step-sizes cause the method to diverge. The range of step-sizes for which some of these methods perform reasonably, can be very narrow. This is contrast with Newton-type, which come with a priori “natural” step-size, i.e., $\alpha = 1$, and only occasionally require the line-search to intervene

2.5 Conclusions And Future Work

In this chapter, we demonstrate that sampled variants of Newton’s method, when implemented appropriately, present compelling alternatives to popular first-order methods for solving convex optimization problems in machine learning and data analysis applications. We discussed, in detail, the GPU-specific implementation of Newton-type methods to achieve similar per-iteration costs as first-order methods. We experimentally showcased their advantages, including robustness to ill-conditioning and higher predictive performance. We also highlighted the sensitivity of various first-order methods with respect to their learning-rate.

Extending our results and implementations to non-convex optimization problems and targeting broad classes of machine learning applications, is an important avenue for future work.

2.6 More Details On Softmax Function eq. (3.2)

2.6.1 Relationship to Logistic Regression with ± 1 -labels

Sometimes, in the literature, for the two-class classification problem, instead of $\{0, 1\}$ the labels are marked as ± 1 . In this case, the corresponding logistic regression is written as

$$F(\mathbf{x}) = \sum_{i=1}^n \log \left(1 + e^{-b_i \mathbf{x}^T \mathbf{a}_i} \right).$$

In this case, we have

$$\begin{aligned} F(\mathbf{x}) &= \sum_{i=1}^n \log \left(e^{\frac{-\mathbf{x}^T \mathbf{a}_i}{2}} + e^{\frac{\mathbf{x}^T \mathbf{a}_i}{2}} \right) - \frac{b_i \mathbf{x}^T \mathbf{a}_i}{2} \\ &= \sum_{i=1}^n \log \left(e^{\frac{-\mathbf{x}^T \mathbf{a}_i}{2}} \left(1 + e^{\mathbf{x}^T \mathbf{a}_i} \right) \right) - \frac{b_i \mathbf{x}^T \mathbf{a}_i}{2} \\ &= \sum_{i=1}^n \log \left(1 + e^{\mathbf{x}^T \mathbf{a}_i} \right) - \frac{(1 + b_i) \mathbf{x}^T \mathbf{a}_i}{2} \\ &= \sum_{i=1}^n \log \left(1 + e^{\mathbf{x}^T \mathbf{a}_i} \right) - \tilde{b}_i \mathbf{x}^T \mathbf{a}_i, \end{aligned}$$

where $\tilde{b}_i \in \{0, 1\}$. Hence this formulation co-incides with (3.2).

Softmax Multi-Class problem is (strictly) convex

Consider the data matrix $X \in \mathbb{R}^{n \times d}$ where each row, \mathbf{a}_i^T , is a row vector corresponding to the i^{th} data point. The Hessian matrix can be written as

$$\nabla^2 \mathcal{L} = \mathbf{X}^T \mathbf{W} \mathbf{X},$$

where

$$\mathbf{X} = \begin{bmatrix} X & 0 & \dots & 0 \\ 0 & X & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & X \end{bmatrix}_{(n \times (C-1)) \times (d \times (C-1))},$$

$$\mathbf{W} = \begin{bmatrix} W_{1,1} & W_{1,2} & \dots & W_{1,C-1} \\ W_{2,1} & W_{2,2} & \dots & W_{2,C-1} \\ \vdots & & \ddots & \vdots \\ W_{C-1,1} & W_{C-1,2} & \dots & W_{C-1,C-1} \end{bmatrix},$$

and each $W_{c,c}$ and $W_{c,b}$ is a $n \times n$ diagonal matrix corresponding to (2.7a) and (2.7b), respectively. Note that since

$$\begin{aligned}
& \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) - \\
& \sum_{\substack{b=1 \\ b \neq c}}^{C-1} \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_{\hat{c}} + \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \\
& = \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \\
& \quad - \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} \left(\sum_{\substack{b=1 \\ b \neq c}}^{C-1} \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_{\hat{c}} \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} \right) \\
& = \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \\
& \quad - \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} \left(1 - \frac{1 + e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} \right) \\
& = \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} > 0,
\end{aligned}$$

the matrix \mathbf{W} is strictly diagonally dominant, and hence it is symmetric positive definite. So the problem is convex (in fact it is strictly-convex if the data matrix X is full column rank).

2.7 Tensorflow's Performance Comparison on Various Compute Platforms

Columns 1 and 2 of table 2.3 plots the results for *covertypes* dataset, when batch size is set to 128, using CPU-only cores (row 1) and 1-GPU-1-CPU-core (row 2) for first-order tensorflow implementations. Note that newton-type methods always use 1-

GPU-1-CPU-core as the compute platform irrespective of any of the hyper-parameter settings. We clearly notice that the first-order methods takes ≈ 600 seconds when GPU cores are used compared to ≈ 350 seconds when CPU cores are used. This can be attributed to the small batch size used for first-order methods. Smaller batch size results in computing the gradient, a compute-intensive operation, much more frequently compared to a large batch size. For the plots shown in table 2.3 training size for *covertype* is set to 450,000. This means gradient is computed ≈ 3516 times to complete each of the training epochs in this instance. Since the batch size is very small most of the GPU cores are idle during every computation of the gradient resulting in low GPU occupancy (which is the ratio of active warps on an SM and maximum allowed warps). Also with each invocation of gradient computation there is CUDA kernel instantiation overhead which accumulates as well. Because of above reasons small batch sizes yield high time per epoch for first-order methods.

Columns 3 and 4 of table 2.3 plots for the results for *covertype* dataset using a large batch size, of 20% of the dataset. Note that batch size for first-order methods is same as the gradient sample size for newton-type methods for these plots. We clearly notice that first-order tensorflow methods takes ≈ 55 seconds when CPU-only cores are used as the compute platform compared to ≈ 22.5 seconds when 1-GPU-1-CPU-core is used, a speedup of $2\times$ over CPU only compute platform. In this instance, during each epoch of first-order methods gradient is evaluated only 5 times. Because of the large batch size, $\approx 90,000$ points, are processed by the GPU resulting in higher utilization of the GPU cores (compared to the same computation using smaller batch size). This explains why GPU-cores yield shorter time per epoch when large batch size are used for first-order methods.

Table 2.3.: Performance comparison between first-order and second-order methods on CPU-only and 1-GPU-1-CPU-core compute platforms for *coverttype* dataset. Batch-size 128 first order methods are compared with second order methods using full gradient and hessian sample size set to 5%. Batch-size 20% first order methods are compared with second order methods using sample sizes of 20% and 5% for gradient and hessian computations respectively.

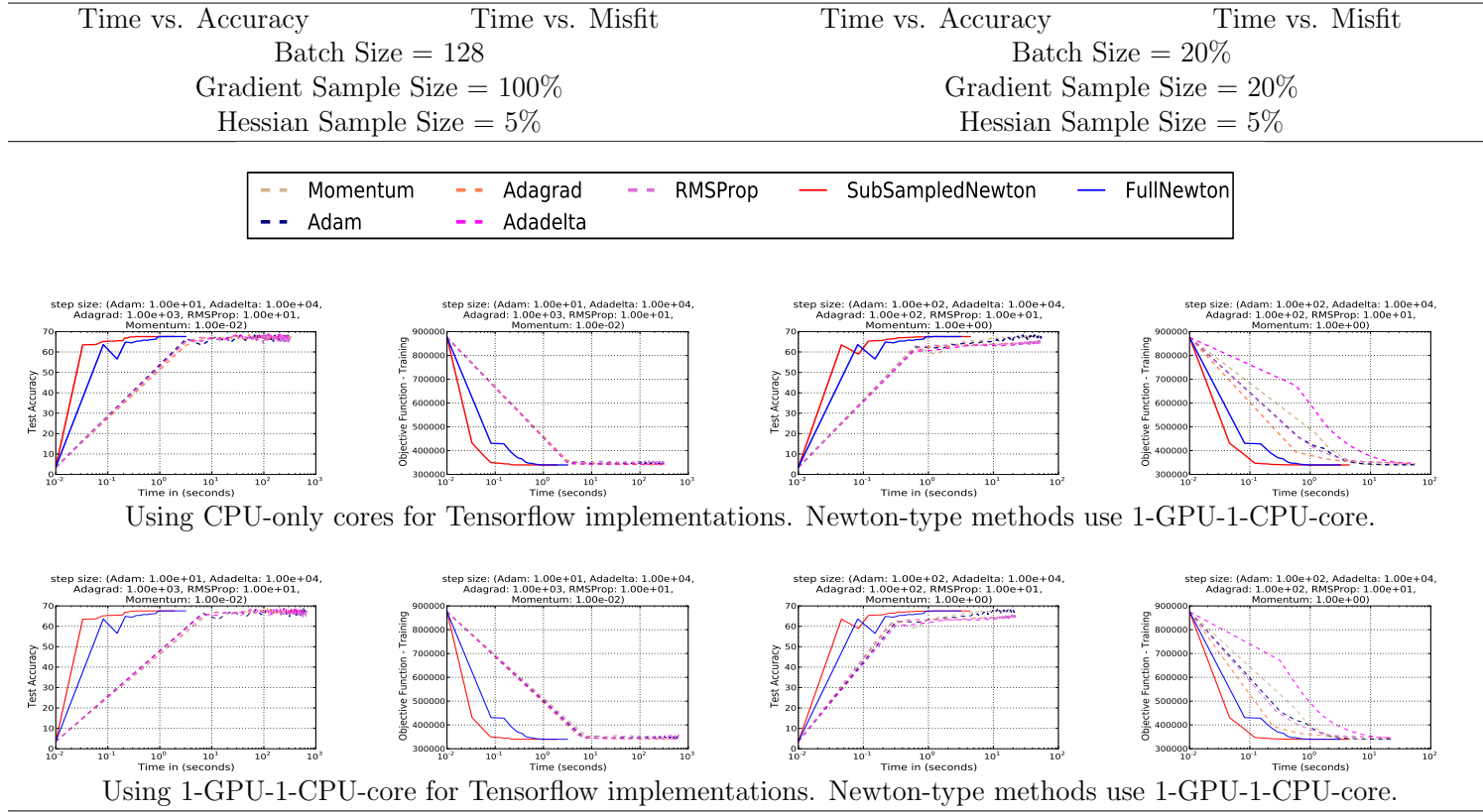
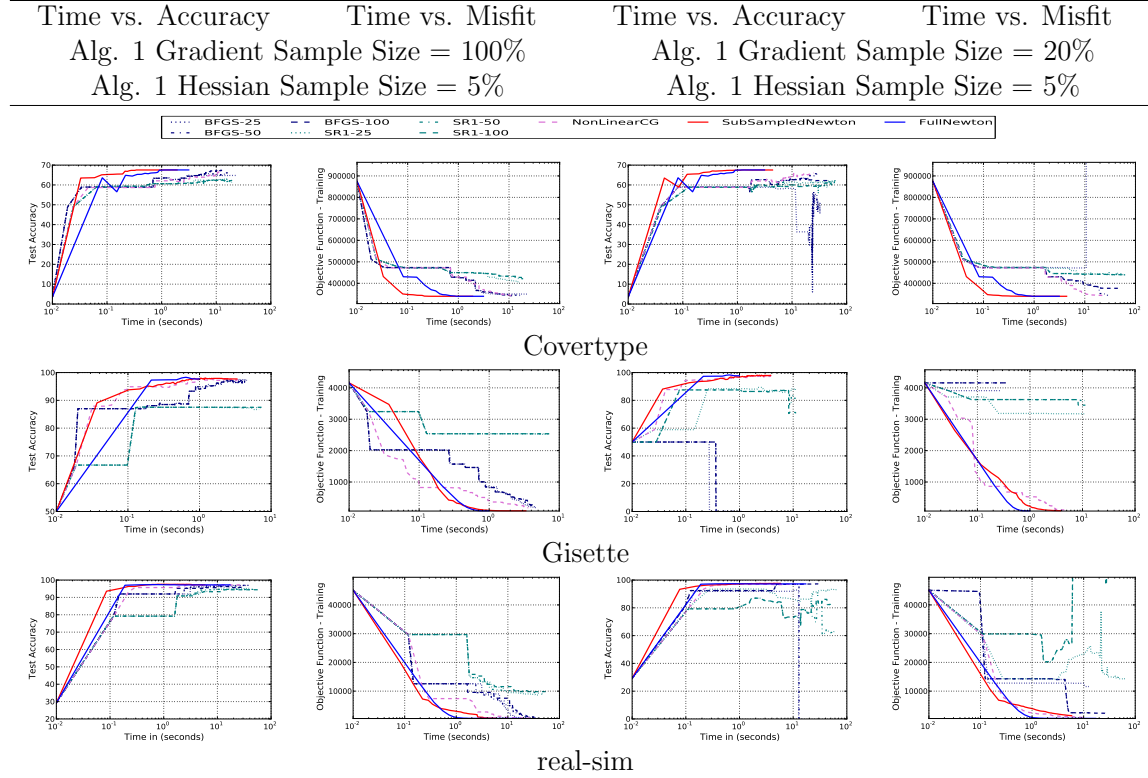


Table 2.4.: Performance comparison between our proposed methods and existing quasi-newton methods.



2.8 Additional Performance Comparisons with Quasi-Newton methods

2.8.1 Covertypes Dataset

Row 1 of Table. 2.4 plots the results for the *Covertypes* dataset. We clearly notice that Newton-type methods outperform quasi-Newton methods and achieve superior generalization results throughout the simulation irrespective of the gradient sample size. We also notice that with the smaller gradient sample size quasi-Newton methods are negatively affected (achieve higher obj function values compared to their full gradient counterparts) and, some of the quasi-Newton methods, *BFGS-25*, display divergent behavior in minimizing the objective function. On the other hand, Newton-type methods are robust to such changes in gradient sample sizes.

2.8.2 Gisette and Real-Sim Datasets

Rows 2 and 3 plot the results for datasets *Gisette* and *Real-Sim*, respectively in Table. 2.4. When full-gradient is used *BFGS-25* and *NonlinearCG* achieve comparable objective function values to Newton-type methods at the very end of the simulation. However, for the *Gisette* dataset, we notice that *NonlinearCG* and function value during the first few epochs. When subsampled gradient is used *SR1* variants diverge for both these datasets. With *Real-Sim* dataset, *NonlinearCG* and *BFGS* variants lower objective function value with *FullNewton* achieving the lowest value during the later part of the simulation.

3. NEWTON-ADMM: A DISTRIBUTED GPU-ACCELERATED OPTIMIZER FOR MULTICLASS CLASSIFICATION PROBLEMS

First-order optimization methods, such as SGD and its variants, are widely used in machine learning tasks due to their simplicity and low per-iteration costs. However, they often require larger numbers of iterations, with associated communication costs in distributed environments. In contrast, Newton-type methods, while having higher per-iteration costs, typically require a significantly smaller number of iterations, which directly translates to reduced communication costs.

In this chapter, we present a novel distributed optimization method, which integrates a GPU-accelerated Newton-type solver with the global consensus formulation of Alternating Direction of Method Multipliers (ADMM). By leveraging the communication efficiency of ADMM, GPU-accelerated inexact-Newton solver, and spectral penalty parameters selection strategy, we show that our proposed method (i) yields better generalization performance on several classification tasks; (ii) outperforms state-of-the-art methods in distributed time to solution; and (iii) offers better scaling on large distributed platforms.

3.1 Introduction

Estimating the parameters of a model from a given dataset is a critical component of a wide variety of machine learning applications. The parameter estimation problem is often translated to one of finding a minima of a suitably formulated objective function. The key challenges in modern “big-data” problems relate to very large numbers of model parameters (which translates to high dimensional optimization problems), large training sets, and learning models with low generalization errors.

Recognizing the importance of the problem, a significant amount of research effort has been invested into addressing these challenges.

The most commonly used optimization technique in machine learning is gradient descent and its stochastic version, stochastic gradient descent (SGD). Algorithms such as gradient descent, that solely rely on gradient information, are often referred to as first-order methods. Recent results [7–9, 41] have shown that use of curvature information in the form of Hessian, or approximations thereof, can lead to significant improvements in terms of performance as manifest in their convergence rate, time, and quality of solutions.

A key challenge in optimization for machine learning problems is the large, often, distributed nature of the training dataset. It may be infeasible to collect the entire training set at a single node and process it serially because of resource constraints (the training set may be too large for a single node), privacy (data may be constrained to specific locations), or the need for reducing optimization time. In each of these cases, there is a need for optimization methods that are suitably adapted to the parallel and distributed computing environments.

Distributed optimization solvers adopt one of two strategies – (i) executing each operation in conventional solvers (e.g., SGD or (quasi) Newton) in a distributed environment, e.g., [42–49]; or (ii) executing an ensemble of local optimization procedures that operate on their own data, with a coordinating procedure that harmonizes the models over iterations, e.g., [50, 51]. The trade-offs between these two methods are relatively well understood in the context of existing solvers – namely that the communication overhead of methods in the first class is higher, whereas, the convergence rate of the second class of methods is compromised. For this reason, methods in the first class are generally preferred in tightly coupled data-center type environments, whereas methods in the latter class are preferred for wide area deployments.

Alternating Direction Method of Multipliers (ADMM), is a well known method in distributed optimization for solving consensus problems [52]. To achieve superior convergence and efficient solution of the corresponding sub-problems, the choices of

the penalty parameter and inner sub-problem solver are critical. In particular, the quality of inner sub-problem solutions dictates the accuracy of the descent direction computed by ADMM; see Appendix 3.5.3. To this end, we use the Spectral Penalty Selection (SPS) [50] for setting the penalty parameters and employ a variant of Newton’s method as sub-problem solver. This is motivated by the observation that first-order solvers are known to suffer from slow convergence rates, and are notoriously sensitive to problem ill-conditioning and the choice of hyper-parameters. In contrast, Newton-type methods are less sensitive to such adversarial effects. However, this feature comes with increased per-iteration computation cost. In our solution, we leverage lower iteration counts to minimize communication cost and efficient GPU implementations to address increased computational cost.

Contributions: Our contributions can be summarized as follows:

- *We propose a novel distributed, GPU-accelerated Newton-type method based on an ADMM framework that has low communication overhead, good per-iteration compute characteristics through effective use of GPU resources, superior convergence properties, and minimal resource overhead.*
- *Using a range of real-world datasets (both sparse and dense), we demonstrate that our proposed method yields significantly better results compared to a variety of state-of-the-art distributed optimization methods.*
- *Our pyTorch implementation is publicly available and can be readily used for practical applications by data scientists and it can be easily adopted to other well-known tools like Tensorflow.*

3.1.1 Related Research

First-order methods [53,54] – gradient descent and its variants are commonly used in ML applications. This is mainly because these methods are simple to implement and have low per-iteration costs. However, it is known that these methods often take

a large number of iterations to achieve reasonable generalization. This is primarily attributed to their sensitivity to problem ill-conditioning. Second-order methods make use of curvature information, in the form the Hessian matrix, and as a result are more robust to problem ill-conditioning [8, 9], and to hyper-parameter tuning [6, 7]. However, they can have higher memory and computation footprints due to the application of the Hessian matrix. In this context, quasi-Newton methods [1] can be used to approximate the Hessian by using the history of gradients. However, a history of gradients must be stored in order to approximate the Hessian matrix, and extra computation cost is incurred to satisfy the strong Wolfe condition. In addition, these methods are observed to be unstable when used on mini-batches [41].

Several distributed solvers have been developed recently [42–49]. Among these, [42–45] are classified as first-order methods. Although they incur low computational costs, they have higher communication costs due to a large number of messages exchanged per mini-batch and high total iteration counts. Second-order variants [46–49, 55] are designed to improve convergence rate, as well as to reduce communication costs. DANE [47], and the accelerated scheme AIDE [48] use SVRG [56] as the subproblem solver to approximate the Newton direction. These methods are often sensitive to the fine-tuning of SVRG. DiSCO [49] uses distributed preconditioned conjugate gradient (PCG) to approximate the Newton direction. The number of communications across nodes per PCG call is proportional to the number of PCG iterations. In contrast to DiSCO, GIANT [46] executes CG at each node and approximates the Newton direction by averaging the solution from each CG call. Empirical results have shown that GIANT outperforms DANE, AIDE, and DiSCO. The solver of Dunner et al. [57] is shown to outperform GIANT, however, it is restricted to sparse datasets. More recently, DINGO [55] has been developed, which unlike GIANT, can be applied to a class of non-convex functions, namely invex [58], that includes convexity as a special sub-class. However, in the absence of invexity, the method can converge to undesirable stationary points.

A popular choice in distributed settings is ADMM [52], which combines dual ascent method and the method of multipliers. ADMM only requires one round of communication per iteration. However, ADMM’s performance is greatly affected by the selection of the penalty parameter [50,51] as well as the choice of local subproblem solvers.

3.2 Problem Formulation and Algorithm Details

In this section, we describe the optimization problem formulation, and present our proposed Newton-ADMM optimizer.

3.2.1 Problem Formulation

Consider a finite sum optimization problem of the form:

$$\min_{\mathbf{x} \in \mathbb{R}^d} F(\mathbf{x}) \triangleq \sum_{i=1}^n f_i(\mathbf{x}) + g(\mathbf{x}), \quad (3.1)$$

where each $f_i(\mathbf{x})$ is a smooth convex function and $g(\mathbf{x})$ is a (strongly) convex and smooth regularizer. In ML applications, $f_i(\mathbf{x})$ can be viewed as loss (or misfit) corresponding to the i^{th} observation (or measurement) [11–13, 59]. In our study, we choose multi-class classification using soft-max and cross-entropy loss function, as an important instance of finite sum minimization problem. Consider a p dimensional feature vector \mathbf{a} , with corresponding labels b , drawn from C classes. In such a classifier, the probability that \mathbf{a} belongs to a class $c \in \{1, 2, \dots, C\}$ is given by:

$$\Pr(b = c \mid \mathbf{a}, \mathbf{x}_1, \dots, \mathbf{x}_C) = \frac{e^{\langle \mathbf{a}, \mathbf{x}_c \rangle}}{\sum_{c'=1}^C e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle}},$$

where $\mathbf{x}_c \in \mathbb{R}^p$ is the weight vector corresponding to class c . Recall that there are only $C - 1$ degrees of freedom, since probabilities must sum to one. Consequently,

for training data $\{\mathbf{a}_i, b_i\}_{i=1}^n \subset \mathbb{R}^p \times \{1, 2, \dots, C\}$, the cross-entropy loss function for $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \dots; \mathbf{x}_{C-1}] \in \mathbb{R}^{(C-1)p}$ can be written as:

$$\begin{aligned} F(\mathbf{x}) &\triangleq F(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}) \\ &= \sum_{i=1}^n \left(\log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) - \sum_{c=1}^{C-1} \mathbf{1}(b_i = c) \langle \mathbf{a}_i, \mathbf{x}_c \rangle \right). \end{aligned} \quad (3.2)$$

Note that $d = (C - 1)p$. After the training phase, a new data instance \mathbf{a} is classified as:

$$b = \arg \max \left\{ \left\{ \frac{e^{\langle \mathbf{a}, \mathbf{x}_c \rangle}}{\sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle}} \right\}_{c=1}^{C-1}, 1 - \frac{e^{\langle \mathbf{a}, \mathbf{x}_1 \rangle}}{\sum_{c'=1}^C e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle}} \right\}.$$

3.2.2 ADMM Framework

Let \mathcal{N} denote the number of nodes (compute elements) in the distributed environment. Assume that the input dataset \mathcal{D} is split among the \mathcal{N} nodes as $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \dots \cup \mathcal{D}_{\mathcal{N}}$. Using this notation, (3.1) can be written as:

$$\begin{aligned} \min & \sum_{i=1}^{\mathcal{N}} \sum_{j \in \mathcal{D}_i} f_j(\mathbf{x}_i) + g(\mathbf{z}) \\ \text{s.t.} & \quad \mathbf{x}_i - \mathbf{z} = 0, \quad i = 1, \dots, \mathcal{N}, \end{aligned} \quad (3.3)$$

where \mathbf{z} represents a global variable enforcing consensus among \mathbf{x}_i 's at all the nodes. In other words, the constraint enforces a consensus among the nodes so that all the local variables, \mathbf{x}_i , agree with global variable \mathbf{z} . The formulation (3.3) is often referred to as a *global consensus* problem. ADMM is based on an augmented Lagrangian framework; it solves the global consensus problem by alternating iterations on primal/dual variables. In doing so, it inherits the benefits of the decomposability of dual ascent and the superior convergence properties of the method of multipliers.

ADMM methods introduce a penalty parameter ρ , which is the weight on the measure of *disagreement* between \mathbf{x}_i 's and global consensus variable, \mathbf{z} . The most common adaptive penalty parameter selection is Residual Balancing [52], which tries to balance the dual norm and residual norm of ADMM. Recent empirical results using SPS [50], which is based on the estimation of the local curvature of subproblem on each node, yields significant improvement in the efficiency of ADMM. Using the SPS strategy for penalty parameter selection, ADMM iterates can be written as follows:

$$\mathbf{x}_i^{k+1} = \arg \min_{\mathbf{x}_i} f_i(\mathbf{x}_i) + \frac{\rho_i^k}{2} \|\mathbf{z}^k - \mathbf{x}_i + \frac{\mathbf{y}_i^k}{\rho_i^k}\|_2^2, \quad (3.4a)$$

$$\mathbf{z}^{k+1} = \arg \min_{\mathbf{z}} g(\mathbf{z}) + \sum_{i=1}^{\mathcal{N}} \frac{\rho_i^k}{2} \|\mathbf{z} - \mathbf{x}_i^{k+1} + \frac{\mathbf{y}_i^k}{\rho_i^k}\|_2^2, \quad (3.4b)$$

$$\mathbf{y}_i^{k+1} = \mathbf{y}_i^k + \rho_i^k (\mathbf{z}^{k+1} - \mathbf{x}_i^{k+1}). \quad (3.4c)$$

With ℓ_2 -regularization, i.e., $g(\mathbf{x}) = \lambda \|\mathbf{x}\|^2/2$, (3.4b) has a closed-form solution given by

$$\mathbf{z}^{k+1} (\lambda + \sum_{i=1}^{\mathcal{N}} \rho_i^k) = \sum_{i=1}^{\mathcal{N}} [\rho_i^k \mathbf{x}_i^{k+1} - \mathbf{y}_i^k], \quad (3.5)$$

where λ is the regularization parameter.

Algorithm 11 presents our proposed method incorporating the above formulation of ADMM.

Steps 11-11 initialize the multipliers, \mathbf{y} , and consensus vectors, \mathbf{z} , to zeros. In each iteration, Single Node Newton method, Algorithm 13, is run with local \mathbf{x}_i , \mathbf{y}_i , and global \mathbf{z} vectors. Upon termination of Algorithm 13 at all nodes, resulting local Newton directions, \mathbf{x}_i^k , are gathered at the master node, which generates the next iterates for vectors \mathbf{y} and \mathbf{z} using spectral step sizes described in [50]. These steps are repeated until convergence.

Remark 3 *Note that in each ADMM iteration only one round of communication is required (a “gather” and a “scatter” operation), which can be executed in $\mathcal{O}(\log(\mathcal{N}))$ time. Further, the application of the GPU-accelerated inexact Newton-CG Algorithm*

Algorithm 11: ADMM method (outer solver)

Input : $\mathbf{x}^{(0)}$ (initial iterate), \mathcal{N} (no. of nodes)
Parameters: β , λ and $\theta < 1$
Initialize \mathbf{z}^0 to 0
Initialize \mathbf{y}_i^0 to 0 on all nodes.
foreach $k = 0, 1, 2, \dots$ **do**
 Perform Algorithm 13 with, \mathbf{x}_i^k , \mathbf{y}_i^k , and \mathbf{z}^k on all nodes
 Collect all local \mathbf{x}_i^{k+1}
 Evaluate \mathbf{z}^{k+1} and \mathbf{y}_i^{k+1} using (3.4b) and (3.4c).
1 Distribute \mathbf{z}^{k+1} and \mathbf{y}_i^{k+1} to all nodes.
 Locally, on each node, compute spectral step sizes and
 penalty parameters as in [50]
end

13 at each node significantly speeds-up the local computation per epoch. The combined effect of these algorithmic choices contribute to the high overall efficiency of the proposed Newton-ADMM Algorithm 11, when applied to large datasets.

ADMM Residuals and Stopping Criteria

The consensus problem (3.3) can be solved by iterating ADMM subproblems (3.4a), (3.4c), and (3.4b). To monitor the convergence of ADMM, we can check the norm of primal and dual residuals, \mathbf{r}^k and \mathbf{d}^k , which are defined as follows:

$$\mathbf{r}^k = \begin{bmatrix} \mathbf{r}_1^k \\ \vdots \\ \mathbf{r}_{\mathcal{N}}^k \end{bmatrix}, \mathbf{d}^k = \begin{bmatrix} \mathbf{d}_1^k \\ \vdots \\ \mathbf{d}_{\mathcal{N}}^k \end{bmatrix} \quad (3.6)$$

where $\forall i \in \{1, 2, \dots, \mathcal{N}\}$,

$$\mathbf{r}_i^k = \mathbf{z}^k - \mathbf{x}_i^k, \mathbf{d}_i^k = -\rho_i^k(\mathbf{z}^k - \mathbf{z}^{k-1}) \quad (3.7)$$

As $k \rightarrow \infty$, $\mathbf{z}^k \rightarrow \mathbf{z}^*$ and $\forall i, \mathbf{x}_i^k \rightarrow \mathbf{z}^*$. Therefore, the norm of primal and dual residuals, $\|\mathbf{r}^k\|$ and $\|\mathbf{d}^k\|$, converge to zero. In practice, we do not need the solution to high precision, thus ADMM can be terminated as $\|\mathbf{r}_i^k\| \leq \epsilon^{pri}$ and $\|\mathbf{d}_i^k\| \leq \epsilon^{dual}$. Here, ϵ^{pri} and ϵ^{dual} can be chosen as:

$$\epsilon^{pri} = \sqrt{\mathcal{N}}\epsilon^{abs} + \epsilon^{rel} \max\left\{\sum_{i=1}^{\mathcal{N}} \|\mathbf{x}_i^k\|^2, \mathcal{N}\|\mathbf{z}^k\|^2\right\} \quad (3.8)$$

$$\epsilon^{dual} = \sqrt{d}\epsilon^{abs} + \epsilon^{rel} \max\left\{\sum_{i=1}^{\mathcal{N}} \|\mathbf{y}_i^k\|^2\right\} \quad (3.9)$$

The choice of absolute tolerance ϵ^{abs} depends on the chosen problem and the choice of relative tolerance ϵ^{rel} for the stopping criteria is, in practice, set to 10^{-3} or 10^{-4} .

3.2.3 Inexact Newton-CG Solver

For the optimization problem (3.1), in each iteration, the gradient and Hessian are given by

$$\mathbf{g}(\mathbf{x}) \triangleq \sum_{j \in \mathcal{D}} \nabla f_j(\mathbf{x}) + \nabla g(\mathbf{x}), \quad (3.10a)$$

$$\mathbf{H}(\mathbf{x}) \triangleq \sum_{j \in \mathcal{D}} \nabla^2 f_j(\mathbf{x}) + \nabla^2 g(\mathbf{x}). \quad (3.10b)$$

At each iterate $\mathbf{x}^{(k)}$, using the corresponding Hessian, $\mathbf{H}(\mathbf{x}^{(k)})$, and the gradient, $\mathbf{g}(\mathbf{x}^{(k)})$, we consider *inexact* Newton-type iterations of the form:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k, \quad (3.11a)$$

where \mathbf{p}_k is a search direction satisfying:

$$\|\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k + \mathbf{g}(\mathbf{x}^{(k)})\| \leq \theta \|\mathbf{g}(\mathbf{x}^{(k)})\|, \quad (3.11b)$$

for some inexactness tolerance $0 < \theta < 1$ and α_k is the largest $\alpha \leq 1$ such that

$$F(\mathbf{x}^{(k)} + \alpha \mathbf{p}_k) \leq F(\mathbf{x}^{(k)}) + \alpha \beta \mathbf{p}_k^T \mathbf{g}(\mathbf{x}^{(k)}), \quad (3.11c)$$

for some $\beta \in (0, 1)$.

Requirement (3.11c) is often referred to as Armijo-type line-search [1]. To compute the step-size, α in eq. (3.11c), we use a *backtracking* line search, as shown in algorithm 12. This function takes parameters, α as initial step-size, which in our case is always set to 1, $\beta < 1$, \mathbf{p} is the Newton-direction, and the gradient vector is \mathbf{g} . The loop at line 12 is repeated until desired reduction is achieved along the Newton-direction, \mathbf{p} , by successively decreasing the step-size by a factor $\gamma < 1$.

Condition (3.11b) is the θ -relative error approximation of the exact solution to the linear system:

$$\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k = -\mathbf{g}(\mathbf{x}^{(k)}), \quad (3.12)$$

Note that in (strictly) convex settings, where the Hessian matrix is symmetric positive definite (SPD), conjugate gradient (CG) with early stopping can be used to obtain an approximate solution to (3.12) satisfying (3.11b). In [8, 9], it has been shown that a mild value for θ , in the order of inverse of *square-root of the condition number*, is sufficient to ensure that the convergence properties of the exact Newton's method are preserved. As a result, for ill-conditioned problems, an approximate solution to (3.12) using CG yields good performance, comparable to an exact update (see examples in Section 3.3). Putting all of these together, we obtain Algorithm 13, which is known to be globally linearly convergent, with problem-independent local convergence rate [8, 9].

Algorithm 12: Line Search

Input :
 \mathbf{x} - Current point
 \mathbf{p} - Newton's direction
 $F(\cdot)$ - Function pointer
 $\mathbf{g}(\mathbf{x})$ - Gradient vector

Parameters:
 α - Initial step size
 $0 < \beta < 1$ - Sufficient descent constant
 $0 < \gamma < 1$ - Back-tracking parameter
 i_{\max} - Maximum line search iterations

$\alpha = 1$
 $i = 0$
while $F(\mathbf{x} + \alpha\mathbf{p}) > F(\mathbf{x}) + \alpha\beta\mathbf{p}^T\mathbf{g}(\mathbf{x})$ **do**
1 **if** $i > i_{\max}$ **then**
 | break
 end
 $i = i + 1$
 $\alpha \leftarrow \gamma\alpha$
end

Algorithm 13: Inexact Newton-type Method

Input : $\mathbf{x}^{(0)}$
Parameters: $0 < \beta, \theta < 1$
foreach $k = 0, 1, 2, \dots$ **do**
 Form $\mathbf{g}(\mathbf{x}^{(k)})$ and $\mathbf{H}(\mathbf{x}^{(k)})$ as in (3.10)
 if $\|\mathbf{g}(\mathbf{x}^{(k)})\| < \epsilon$ **then**
 | STOP
 end
 Update $\mathbf{x}^{(k+1)}$ as in (3.11)
end

3.3 Experimental Evaluation

In this section, we evaluate the performance of Newton-ADMM as compared with several state-of-the-art alternatives.

Table 3.1.: Description of the datasets.

Classes	Dataset	Train Size	Test Size	Dims
2	HIGGS	10,000,000	1,000,000	28
10	MNIST	60,000	10,000	784
10	CIFAR-10	50,000	10,000	3,072
20	E18	1,300,128	6,000	279,998

Experimental Setup and Data: All algorithms are implemented in PyTorch release "0.3.0.post4" with Message Passing Interface (MPI) backend. We test performance of the methods on two hardware platforms. The first platform is a server with 384 Intel Xeon Platinum 8168 processors and 8 Tesla P100 GPU cards. The second platform is a CentOS 7 cluster with 15 nodes with 100 Gbps Infiniband interconnect. Each node has 96GB RAM, two 12-Core Intel Xeon Gold processors, and 3 Tesla P100 GPU cards. We validate our proposed method using real-world datasets, described in Table 3.1, and compare with state-of-the-art first-order and second-order optimizers. These datasets are chosen to cover a wide range of problem characteristics (problem-conditioning, features, problem-size). MNIST is a widely used dataset for validation – it is relatively well-conditioned. CIFAR-10 is 3.9x larger than MNIST and is relatively ill-conditioned. HIGGS is a low-dimensional dataset, however is the largest (in terms of problem size) compared to the rest. This dataset is easy to solve for our algorithm, but is harder for first-order variants because of high communication overhead. The largest data set, E18, in terms of dimension and number of samples, is used to highlight the scalability of our proposed method.

Comparison with Distributed First-order Methods. While the per-iteration cost of first-order methods is relatively low, they require larger number of iterations, increasing associated communication overhead, and CPU-GPU transactions, if GPUs are used (Please see detailed discussion in section 3.5.1). In this experiment, we demonstrate that these drawbacks of first order methods are significant, in the context of MNIST, CIFAR-10, HIGGS, and E18 datasets using 4 workers for Newton-ADMM

and synchronous SGD, both with the GPUs enabled and GPUs disabled. The results are shown in Figure 3.1. Specifically, we note that GPU-accelerated Newton-ADMM method with minimal communication overhead yields significantly better results – over an order of magnitude faster in most cases, when compared to synchronous SGD.

We present the ratio of CPU time to GPU time for Newton-ADMM and SGD in Table 3.2. We observe that for both Newton-ADMM and SGD, the CPU-GPU time ratio is proportional to the dimensions of datasets. For example, on the dataset with the lowest dimension (HIGGS), the CPU-GPU time ratio is the least for both Newton-ADMM and SGD, whereas on the dataset with the highest dimension (E18), the CPU-GPU time ratios are the highest for both Newton-ADMM and SGD. In all cases, the use of GPUs results in highest speedup for Newton-ADMM. The gain in GPU utilization is compromised by large number of CPU-GPU memory transfers for SGD. As a result, SGD shows meaningful GPU acceleration only for the E18 dataset.

Second, we observe that Newton-ADMM has much lower communication cost, compared to SGD. This can be observed from the Figure 3.1. In all cases, SGD takes longer than Newton-ADMM with GPUs enabled. This is mainly because SGD requires a large number of gradient communications across nodes. As a result, we observe that Newton-ADMM is 4.9x, 6.3x, 22.6x, and 17.8x, times faster than SGD on MNIST, CIFAR-10, HIGGS, and E18 datasets, respectively.

Finally, we conclude that Newton-ADMM has superior convergence properties compared to SGD. This is demonstrated in Figure 3.1 for the HIGGS dataset. We observe that Newton-ADMM converges to low objective values in just few iterations. On the other hand, the objective value, even at 100-th epoch for SGD, is still higher than Newton-ADMM.

Comparison with Distributed Second-order Methods. We compare Newton-ADMM against DANE [47], AIDE [48], and GIANT [46], which have been shown in recent results to perform well. In each iteration, DANE [47] requires an exact

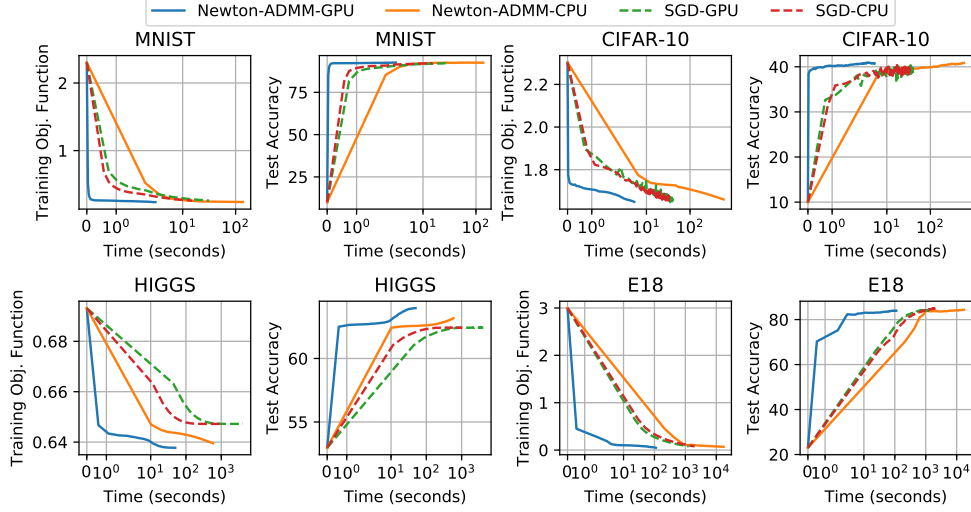


Fig. 3.1.: Training objective function and test accuracy as functions of time for Newton-ADMM and synchronous SGD, both with GPU enabled and GPU disabled, with 4 workers. Overall, Newton-ADMM favors GPUs, enjoys minimal communication overhead, and enjoys faster convergence compared to synchronous SGD.

Table 3.2.: GPU Speedup for Newton-ADMM and SGD.

CPU/GPU Time Ratio	Newton-ADMM	SGD
MNIST	44.7345904	0.47896507
CIFAR-10	112.670178	0.8212862
HIGGS	11.842679	0.26789652
E18	154.425688	1.54673642

solution of its corresponding subproblem at each node. This constraint is relaxed in an inexact version of DANE, called InexactDANE [48], which uses SVRG [56] to approximately solve the subproblems. Another version of DANE, called Accelerated Inexact DanE (AIDE), proposes techniques for accelerating convergence, while still using InexactDANE to solve individual subproblems [48]. However, using SVRG to solve subproblems is computationally inefficient due to its double loop formulation, with the outer loop requiring full gradient recalculation and several stochastic gradient calculations in inner loop.

Figure 3.2 shows the comparison between these methods on the MNIST dataset with $\lambda = 10^{-5}$. Although InexactDANE and AIDE start at lower objective function values, the average epoch time compared to Newton-ADMM and GIANT is orders of magnitude higher (*order of 1000x*). For instance, to reach an objective function value less than 0.25 on the MNIST dataset, Newton-ADMM takes only 2.4 seconds, whereas InexactDANE consumes *an hour and a half*. Since InexactDANE and AIDE are significantly slower than Newton-ADMM and GIANT (on other datasets as well – for which we do not show results here), we restrict our discussion of results on performance and scalability to Newton-ADMM and GIANT in the rest of this section.

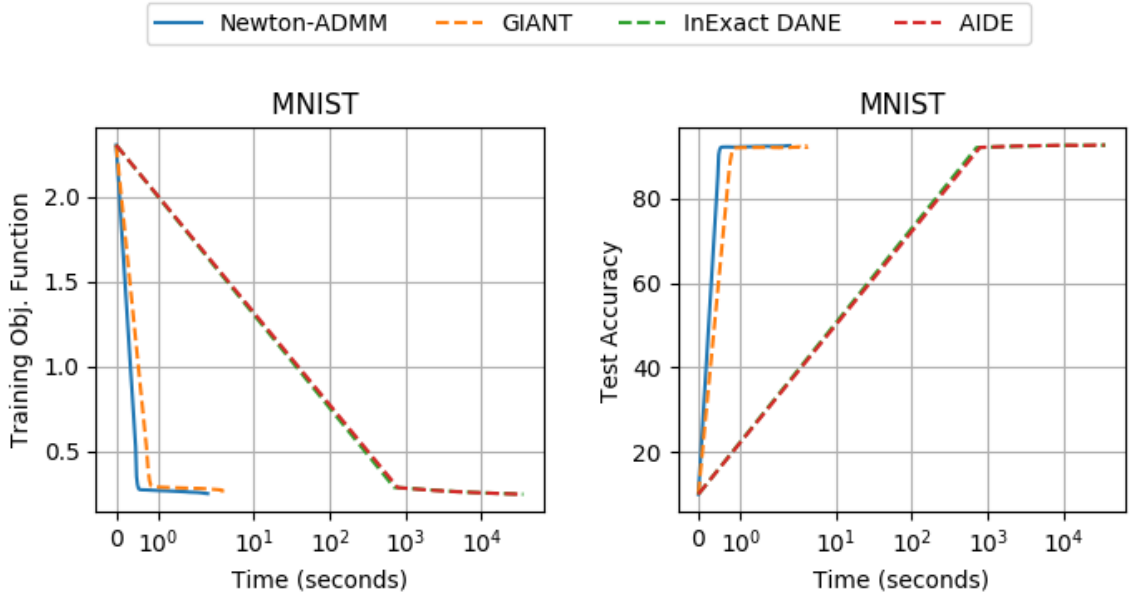


Fig. 3.2.: Training objective function and test accuracy comparison over time for Newton-ADMM, GIANT, InexactDANE, and AIDE on MNIST dataset with $\lambda = 10^{-5}$. We run both Newton-ADMM and GIANT for 100 epochs. Since the computation times per epoch for InexactDANE and AIDE are high, we only run 10 epochs for these methods. We present details of hyperparameter settings in 3.5.4.

Scalability of Newton-ADMM. Figure 3.3 presents strong- and weak-scaling results for Newton-ADMM and GIANT. In strong-scaling experiments, we keep the

number of training samples constant, while increasing the number of workers, and for weak-scaling, the number of the training samples per node is kept constant. For strong-scaling, as number of workers increases, average epoch time for both Newton-ADMM and GIANT decreases. For all the datasets, as the number of workers is doubled, the average epoch time halved for both methods. For weak scaling, as the number of workers doubles, the average epoch time nearly remains constant for both methods. Both Newton-ADMM and GIANT use CG to compute Newton directions. However, compared to GIANT, Newton-ADMM has lower epoch times for the following reasons: first, to guarantee global convergence on non-quadratic problems, GIANT uses a globalization strategy based on line search. For this, the i -th worker computes the local objective function values $f_{\mathcal{D}_i}(\mathbf{x}_i + \alpha \mathbf{p})$ for all α 's in a pre-defined set of step-sizes $S = \{2^0, 2^{-1}, \dots, 2^{-k}\}$, where k is the maximum number of line search iterations. Thus, for each epoch, all workers need to compute a fixed number of objective function values. In contrast, Newton-ADMM performs line search only locally, allowing each worker to terminate line search before reaching the maximum number of line search iterations, and hence reducing the overhead of redundant computations. Second, Newton-ADMM only requires one round of messages per iteration, whereas GIANT needs three. Our experiments are performed on a Gigabit-interconnet cluster, where communication fabric is highly optimized. However, in environments with lower bandwidth and higher latency, we expect Newton-ADMM to perform significantly better compared to GIANT.

We now compare the convergence of Newton-ADMM with GIANT in a distributed setting. Instead of comparing the test accuracy or objective value over time, we compare how close the objective value obtained from the solver is to the optimal objective value. Specifically, define $\theta = (F(\mathbf{x}^k) - F(\mathbf{x}^*)) / F(\mathbf{x}^*)$, we measure θ as a function of time. (Here, $F(\cdot)$ denotes the objective function, \mathbf{x}^k is the approximate solution obtained by the solver at the k -th iterate, and the “optimal” solution vector \mathbf{x}^* is obtained by running Newton’s method on a single node to high precision). Figure 3.4 shows θ , in log scale, as a function of time for MNIST, CIFAR-10, and HIGGS

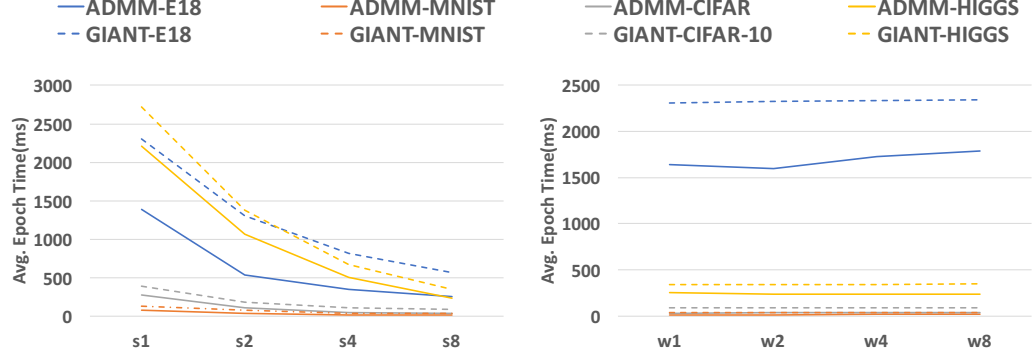


Fig. 3.3.: Avg. Epoch Time for Strong and Weak Scaling for Newton-ADMM and GIANT.

using 8 compute nodes. From Figure 3.4, we observe that, given the same amount of time, Newton-ADMM can reach lower θ in each case. We also measure the number of epochs taken by the solver to reach $\theta < 0.05$. Table 3.3 shows the number of epochs for Newton-ADMM and GIANT to reach $\theta < 0.05$ on 8 nodes. From Table 3.3, we can see that Newton-ADMM converges to optimal solution significantly faster than GIANT. Specifically, to reach $\theta \leq 0.05$, for the MNIST dataset, Newton-ADMM takes 252 epochs while GIANT takes 1086 epochs. For the CIFAR-10 dataset, Newton-ADMM takes 1204 epochs while GIANT takes 3215 epochs. The speed up ratio on MNIST and CIFAR-10 is 5.15 and 11.14, respectively. Both Newton-ADMM and GIANT behave well on HIGGS. It only takes 1 epoch for both solvers to reach $\theta \leq 0.05$. We note that the superior performance of these methods on HIGGS does not carry over to first-order methods.

Finally, we stress that Newton-ADMM scales well on large datasets in large-scale distributed environments. From Figure 3.5, we note that Newton-ADMM takes significantly smaller amount of time to achieve lower objective values and higher test accuracy on E18 running on 32 compute nodes. The large dimensionality of E18 (280K) highlights the memory- and compute-efficient formulation of our Hessian-vector products and subproblem solves on GPUs (please see Appendix 3.5.2 for full

GPU utilization characteristics of our solvers) – the average epoch time for the E18 dataset on 32 nodes is only 1.98 seconds!

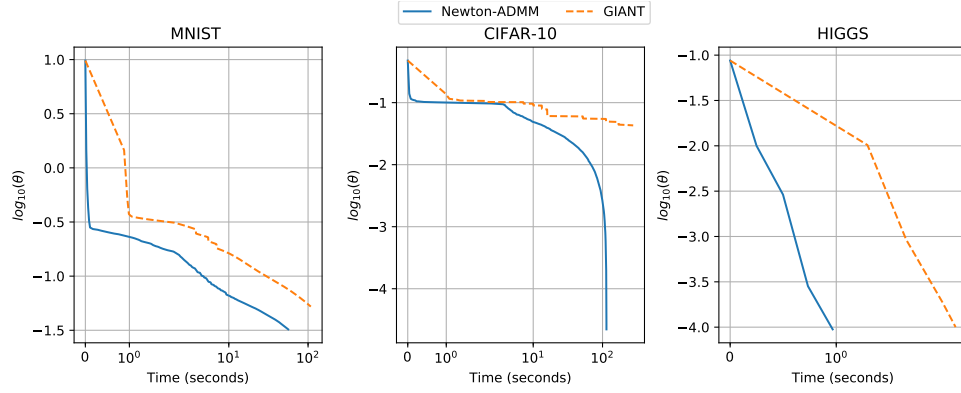


Fig. 3.4.: $\log_{10}(\theta)$ as a function of time for Newton-ADMM and GIANT on MNIST, CIFAR-10, and HIGGS datasets. Newton-ADMM can reach lower θ , given the same amount of time, compared to GIANT. Note that for the HIGGS dataset, both methods can reach low θ soon.

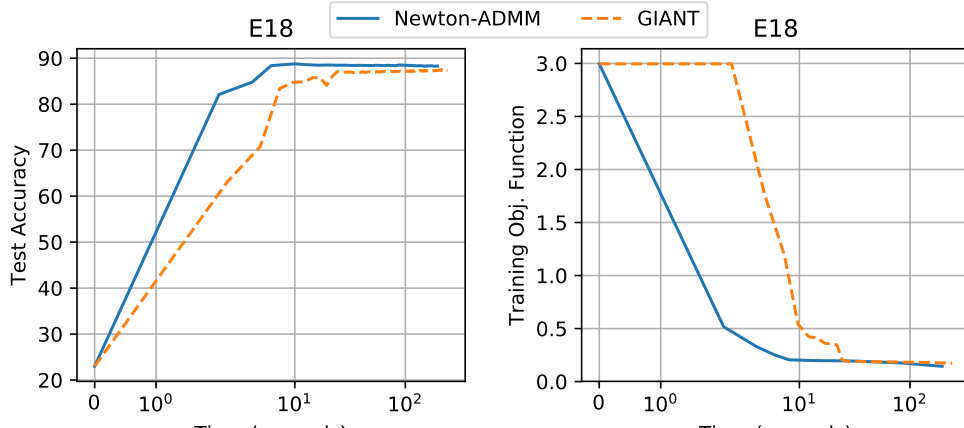


Fig. 3.5.: Training objective function and test accuracy as function of time for Newton-ADMM and GIANT on E18 dataset using 32 nodes. We note that GIANT lingers at higher objective values in the initial iterations, while Newton-ADMM drops to lower objective values rapidly.

Table 3.3.: Performance comparison of Newton-ADMM and GIANT – we present the number of epochs for a solver to reach $\theta < 0.05$. The speedup ratio is defined as the fraction of time taken by GIANT to achieve a specified value of θ to the corresponding time taken by Newton-ADMM on the same hardware platform.

	NT-ADMM Epochs	GIANT Epochs	Speedup
MNIST	252	1086	5.15
CIFAR-10	1204	3215	11.14
HIGGS	1	1	1.35

3.4 Conclusions and Future Directions

We have developed a novel distributed Inexact Newton method based on a global consensus ADMM formulation. We compare our method with state-of-the-art optimization methods and show that our method has much lower distributed overhead, achieves superior generalization errors, and has significantly lower epoch-times on standard benchmarks. We have also shown that our method can handle large datasets, while delivering sub-second epoch times – establishing desirable scalability characteristics of our method. Our results establish Inexact Newton-ADMM as the new benchmark for performance of distributed optimization techniques.

3.5 Appendix

We discuss GPU utilization, numerical stability of the cross-entropy loss function, and introduce our highly optimized implementation of Hessian Vector products. We also demonstrate that ADMM with Newton-type inner solver outperforms ADMM with L-BFGS inner solver. Finally, a detailed description of parameter settings in all experiments is discussed.

3.5.1 On GPU Utilization

Our proposed methods use Inexact Newton-type iterates, with a linear-quadratic convergence rate for strongly convex sub-problems (3.4a). Besides theoretical performance guarantees, our proposed method has practical implications resulting in significantly lower computation time. In practice, mini-batch stochastic gradient descent is widely used over full-batch gradient descent and other methods. However, this method requires a large number of epochs to achieve good generalization errors. Furthermore, the mini-batch update scheme results in significantly lower GPU occupancy (idle GPU cores because of smaller batch sizes). The number of CPU-GPU memory transfers per epoch for mini-batch SGD is $\frac{n}{m}$, where n is the size of dataset, and m is the size of mini-batch. Usually, $n \gg 1$ and m is typically between a hundred and a thousand. In contrast, Newton’s method utilizes the complete dataset for computing direction. Therefore, there is only one CPU-GPU memory transfer for computing Newton direction, which greatly increases utilization of the GPU for reasonably sized datasets. With the judicious mix of statistical methods and carefully formulated Hessian-vector operations, we are able to transform this computation-heavy operation into an efficient and highly scalable GPU-accelerated operation with low memory overhead, please see details in Appendix 3.5.2. For this reason, Newton’s method, in general, is more suitable for high GPU utilization, due to low CPU-GPU data transfer cost, compared to SGD. Furthermore, in distributed implementations, Synchronous SGD induces a communication overhead of $\frac{nd \log(N)}{mN}$, where usually $d \gg 1$.

Numerical Stability

To avoid over-flow in the evaluation of exponential functions in (3.2), we use the “Log-Sum-Exp” trick [36]. Specifically, for each data point \mathbf{a}_i , we first find the maximum value among $\langle \mathbf{a}_i, \mathbf{x}_c \rangle$, $c = 1, \dots, C-1$. Define:

$$M(\mathbf{a}) = \max \left\{ 0, \langle \mathbf{a}, \mathbf{x}_1 \rangle, \langle \mathbf{a}, \mathbf{x}_2 \rangle, \dots, \langle \mathbf{a}, \mathbf{x}_{C-1} \rangle \right\}, \quad (3.13)$$

and

$$\alpha(\mathbf{a}) := e^{-M(\mathbf{a})} + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle - M(\mathbf{a})}. \quad (3.14)$$

Note that $M(\mathbf{a}) \geq 0, \alpha(\mathbf{a}) \geq 1$. Now, we have:

$$1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} = e^{M(\mathbf{a}_i)} \alpha(\mathbf{a}_i).$$

For computing (3.2), we use:

$$\begin{aligned} & \log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) \\ &= M(\mathbf{a}_i) + \log \left(e^{-M(\mathbf{a}_i)} + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle - M(\mathbf{a}_i)} \right) \\ &= M(\mathbf{a}_i) + \log (\alpha(\mathbf{a}_i)). \end{aligned}$$

Note that in all these computations, we are guaranteed to have all the exponents appearing in all the exponential functions to be negative, hence avoiding numerical over-flow.

3.5.2 Hessian Vector Product

Given a vector $\mathbf{v} \in \mathbb{R}^d$, we can compute the Hessian-vector product without explicitly forming the Hessian. For notational simplicity, define

$$h(\mathbf{a}, \mathbf{x}) := \frac{e^{\langle \mathbf{a}, \mathbf{x} \rangle - M(\mathbf{x})}}{\alpha(\mathbf{a})},$$

where $M(\mathbf{x})$ and $\alpha(\mathbf{x})$ were defined in eqs. (3.13) and (3.14), respectively. Now using matrices

$$\mathbf{V} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{v}_1 \rangle & \langle \mathbf{a}_1, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_1, \mathbf{v}_{C-1} \rangle \\ \langle \mathbf{a}_2, \mathbf{v}_1 \rangle & \langle \mathbf{a}_2, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_2, \mathbf{v}_{C-1} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{v}_1 \rangle & \langle \mathbf{a}_n, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_n, \mathbf{v}_{(C-1)} \rangle \end{bmatrix}_{n \times (C-1)}, \quad (3.15)$$

and

$$\mathbf{W} = \begin{bmatrix} h(\mathbf{a}_1, \mathbf{x}_1) & h(\mathbf{a}_1, \mathbf{x}_2) & \dots & h(\mathbf{a}_1, \mathbf{x}_{C-1}) \\ h(\mathbf{a}_2, \mathbf{x}_1) & h(\mathbf{a}_2, \mathbf{x}_2) & \dots & h(\mathbf{a}_2, \mathbf{x}_{C-1}) \\ \vdots & \vdots & \ddots & \vdots \\ h(\mathbf{a}_n, \mathbf{x}_1) & h(\mathbf{a}_n, \mathbf{x}_2) & \dots & h(\mathbf{a}_n, \mathbf{x}_{C-1}) \end{bmatrix}_{n \times (C-1)}, \quad (3.16)$$

we compute

$$\mathbf{U} = \mathbf{V} \odot \mathbf{W} - \mathbf{W} \odot \left(((\mathbf{V} \odot \mathbf{W}) \mathbf{e}) \mathbf{e}^T \right), \quad (3.17)$$

to get

$$\mathbf{H}\mathbf{v} = \text{vec} \left(\mathbf{A}^T \mathbf{U} \right), \quad (3.18)$$

where $\mathbf{v} = [\mathbf{v}_1; \mathbf{v}_2; \dots; \mathbf{v}_{C-1}] \in \mathbb{R}^d$, $\mathbf{v}_i \in \mathbb{R}^p, i = 1, 2, \dots, C - 1$, $\mathbf{e} \in \mathbb{R}^{C-1}$ is a vector of all 1's, and each row of the matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$ is a row vector corresponding to the i^{th} data point, i.e, $\mathbf{A}^T = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n]$.

Remark 4 *Note that the memory overhead of our GPU-accelerated Newton-type method is determined by the dimensions of the matrices \mathbf{U} , \mathbf{V} and \mathbf{W} , which are determined by the local dataset size and number of classes in multi-class classification problem at hand. With reasonably sized GPU clusters this memory footprint can be easily managed for significantly large datasets. This enables Newton-type method to scale to large problems inaccessible to traditional second-order methods.*

3.5.3 Newton-type method as a highly efficient subproblem solver for ADMM:

We establish (GPU-accelerated) Newton-type optimizer as a highly efficient inner solver for ADMM by comparing its performance against an ADMM-L-BFGS solver. The per-iteration computation cost and memory footprint of L-BFGS is lower than our Newton-type method because of the rank-2 approximation of the Hessian matrix. This, however, comes at the cost of a worse convergence rate for L-BFGS. While Newton-type methods compute matrix vector products with the full Hessian, we use a Conjugate Gradient method with early stopping to solve the linear system, $\mathbf{H}\mathbf{x} = -\mathbf{g}$. In our experiments we use no more than 10 CG iterations and a tolerance level of 10^{-3} . This resulting *Inexact* Newton-type method is GPU-accelerated, and with an efficient implementation of Hessian-vector product, we show that in practice, ADMM method suitably aided by efficient implementation of Newton-type subproblem solvers yield significantly better results compared to the state-of-the-art. Furthermore, the use of true Hessian in our inexact solver, a second-order method, makes it resilient to problem ill-conditioning and immune to hyper-parameter tuning. These results are shown in Figure 3.6. We clearly notice that the performance gap between L-BFGS and Inexact Newton-type method becomes larger when number of compute nodes is

increased. The only exception is on HIGGS dataset. This is because the dimension of the HIGGS datasets is only 28 and it is a binary classification problem such that the dimension of Hessian is significantly lower than among all the datasets. In this case, L-BFGS solver yields similar results compared to our Inexact Newton method. Most importantly, we note the following key results: (i) Inexact Newton yields performance improvements from 0 (MNIST) to 550% (HIGGS and CIFAR-10) over L-BFGS on a single node; (ii) when using 8 compute nodes, the performance of L-BFGS-ADMM never catches up with that of Newton-ADMM (in terms of training objective function) in three of four benchmarks (MNIST, CIFAR-10, and E18), conclusively establishing the superiority of our proposed method over L-BFGS-ADMM.

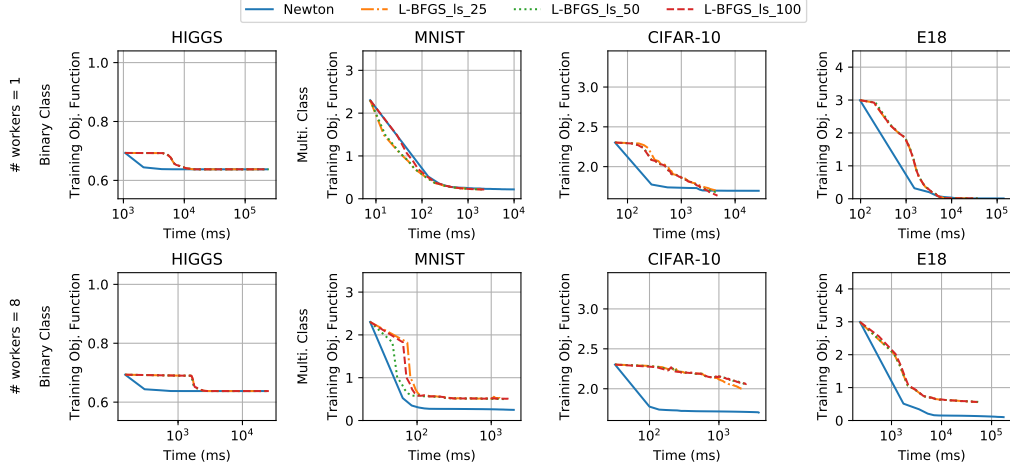


Fig. 3.6.: Training Objective function comparison over time for different choice of inner-solve for ADMM. For the inner solver, we compare the performance of Inexact Newton solver with L-BFGS (with history size 25, 50, 100). The step size of Inexact Newton method is chosen by linesearch following Armijo rule, whereas the step size of L-BFGS is chosen by linesearch satisfying Strong Wolfe condition. We can see that the per-iteration computation cost of L-BFGS is lower than Inexact Newton with the exception on HIGGS dataset. This is because L-BFGS is sensitive to the scale of step size so that more iterations of Strong Wolfe linesearch procedure are required to satisfy the curvature condition. In general, we observe that L-BFGS performs well on binary class problems, while the performance degrades on multiclass problems, when the number of compute nodes increases.

3.5.4 Algorithms Parameter Settings

We generated all the experiment results using the following settings:

- Synchronous SGD : we tune the step size from 10^{-4} to 10^4 and select the best result to report.
- Newton-ADMM : We used 10 CG iterations along with 10^{-4} CG tolerance to compute Newton direction at each compute node. The step size was chosen by line search with 10 iterations.
- GIANT : The configurations for CG and linesearch are the same as the configurations use in Newton-ADMM.
- Inexact DANE : we use learning rate $\eta = 1.0$ and regularization term $\mu = 0.0$ for solving subproblems as prescribed in [47]. We set SVRG iterations to 100 and update frequency as $2n$, where n is the number of local sample points. We run SVRG step size from the set 10^{-4} to 10^4 in increments of 10 and select the best value to report.
- AIDE : The configurations for SVRG is the same as the configurations used in Inexact DANE. As to the additional hyper-parameter introduced in AIDE, τ , we also run τ from the set 10^{-4} to 10^4 and select the best to report.

4. FITRE: FISHER INFORMED TRUST-REGION METHOD FOR TRAINING DEEP NEURAL NETWORKS

Convolutional neural networks are critical components of a diverse class of applications. Enhancing their performance through improved training procedures has tremendous potential impact. First-order methods in general, and stochastic gradient descent in particular, have been the workhorse methods for a large subset of these problems. This is primarily because alternatives such as Newton-type methods either have larger memory requirements, or involve computationally intensive kernels. In view of these considerations, many stochastic variants of higher-order methods have been proposed, which alleviate noted shortcomings to varying extents. Among the most successful of these higher order methods are variants of the classical trust-region method and those that rely on the natural gradient.

In this chapter, we propose an efficient new method for training deep neural networks. Our method leverages advantages of both trust-region and natural gradient methods, by employing natural gradient direction as a way to approximately solve the trust-region sub-problems. We show that our method performs favorably compared with well-tuned first-order and quasi-Newton methods in both generalization error and wall-clock times on a range of deep network architectures. In particular, our method converges much faster than the alternative methods in terms of data efficiency and iterations and yielding orders of magnitude of relative speedup. We further demonstrate the robustness of our method to different hyper-parameters, which results in an easy-to-tune method in practice. We provide an open-source GPU accelerated CUDA implementation of our solver for use in general deep network training.

4.1 Introduction and Motivation

Significant growth in the availability of large datasets, coupled with growth in the processing capacity of hardware has motivated new applications that rely on analyzing massively large datasets quickly, often in real time. Optimization in such applications involves iterating over a large dataset multiple times and learning model parameters until some predefined criteria of convergence is achieved. These processes may take hours for each iteration over the complete dataset, which translates to learning procedures that can take up to weeks. For instance, on the ImageNet dataset [60], which contains about 1.4 million samples, stochastic gradient descent (SGD) can take upwards of 30 hours for each pass over the dataset and weeks to train a deep neural network such as ResNet [61] or VGGNet [62]. In spite of this significant training time, SGD and its distributed variants are the methods of choice in training of deep learning models.

The popularity of SGD, to a great extent, is attributed to its computationally inexpensive model parameter updates. Indeed, SGD’s iterations involve computing the gradient of the objective function on a mini-batch, using *back-propagation* [63, 64], followed by scaling by a predetermined learning rate and, possibly accelerated by momentum [3]. The simplicity of SGD allows for its application to a wide variety of learning tasks, e.g., auto-encoders [65, 66] and reinforcement learning [67, 68]. However, even though, in theory, it has been argued that SGD can avoid undesirable saddle-points [69], realizing this in practice is more involved [6]. In fact, without significant fine-tuning in terms of learning rate, initialization, and mini-batch size, SGD’s performance (with or without momentum) can diverge significantly from idealized theoretical bounds. On the other hand, by leveraging curvature information, in the form of the *Hessian* matrix, many second-order methods come with the inherent ability to navigate their way out of flat regions including saddle points [70–75]. However, these advantages come at the cost of significantly more computations per iteration as compared with SGD. Towards this end, stochastic variants of many of these

methods have recently been proposed, which, by introducing various approximations, can navigate the objective landscape more efficiently [19, 76, 77]. Furthermore, and in sharp contrast to SGD, many of these methods are resilient to the choices of their hyper-parameters, and hence are easy to tune.

A method that occupies the middle ground between first and second order methods relies on the natural gradient [78–80], proposed by Shun-chin Amari. This work posits that in fitting probabilistic models, the underlying parametric distributions can be thought of as belonging to a manifold, whose geometry is governed by the Fisher information matrix. Under this hypothesis, scaling the gradient using the Fisher information matrix can result in more effective directions for navigating the manifold of the parametric probability densities. However, in high-dimensional settings, using the exact Fisher matrix can be intractable. To remedy this, Martens et. al [81, 82], proposed a method, called Kronecker Factored Approximated Curvature (KFAC), to approximate the Fisher information matrix and its *inverse*-vector product, and applied it to applications in neural networks and reinforcement learning. It was shown that KFAC can significantly outperform many of the first-order alternatives.

In this work, we propose an algorithm that combines the advantages of the stochastic trust region method proposed in [19] with those of KFAC, to obtain a **F**isher **I**nformed **T**rust **R**Egion method (FITRE) that is shown to be well-suited for optimization of deep learning models in general, and convolutional neural networks (CNN), in particular. We also leverage the power of GPUs in accelerating various steps of FITRE to deliver excellent performance. We make the following contributions:

1. We present a novel stochastic variant of the trust region method, in which the approximations to the sub-problems are informed by directions obtained from KFAC.
2. By employing KFAC directions to inform trust-region sub-problems, we show that the proposed method inherits the *robustness* as well as *invariance to re-parameterizations* of KFAC.

3. We show that highly optimized GPU implementation perform better than well-tuned SGD, and quasi-Newton alternatives, in terms of generalization errors, convergence rates, as well as wall-clock times. With appropriately tuned hyper parameters, in some cases, we show that our proposed method consumes less wall-clock time compared to alternatives even for the same number of passes over the dataset.
4. We show that the proposed method is resilient to the choice of batch size and tuning of the underlying hyper parameters.
5. As a broader contribution to the user community, we provide an open-source GPU accelerated CUDA optimization framework for CNN’s with a *first-of-its-kind* R-operator for Hessian-vector computations.

The rest of this chapter is organized as follows: Section 4.2 provides an overview of state-of-the-art methods along with a comparison of the proposed method in this context. A detailed discussion of the proposed methods is given in Section 4.3. Evaluation of our methods as compared with a well-tuned SGD as well as BFGS [1, 83] is provided in Section 4.4. Section 4.5 discusses avenues for future work. Section 4.6 provides detailed discussion on the implementation of our proposed method.

4.2 Related Work

SGD [84] is the most commonly used first-order method, owing to its simplicity and inexpensive per-iteration cost. Iterations require computation of the gradient on a mini-batch scaled by a predetermined learning schedule and possibly Nesterov-accelerated momentum [3]. It has been argued that high-dimensional non-convex functions such as those arising in deep learning are riddled with undesirable saddle points [69, 85–87]. For instance, convolutional neural networks, CNNs, display structural symmetry in their parameter space, which leads to an abundance of saddle points [82, 88, 89]. First-order methods, such as SGD, are known to “zig-zag” in high curvature areas and “stagnate” in low curvature regions [69, 88].

One of the primary reasons for the susceptibility of first-order methods to getting trapped in saddle points or nearly flat regions is their reliance on gradient information. Indeed, navigating around saddle points and plateau-like regions can become a challenge for these methods because the gradient is close to zero in most directions [69]. To this end, a number of alternate methods have been proposed in recent times, which, using history of gradients aim to approximate curvature information, and hence maintaining the simplicity of SGD. Such methods include Adam [26] and Adagrad [24]. However, such approximations of the Hessian do not always properly scale the gradient according to the entire curvature information, and hence these methods suffer from similar deficiencies near saddle points and flat regions. More effective variants of these curvature approximations are those in quasi-Newton methods such as SR1 [1], DFP [1], and BFGS [1,90], which use rank-1 and rank-2 updates to iteratively approximate the Hessian. Aided by line search methods, typically satisfying strong-wolfe [1] conditions, these methods yield good results compared to first-order methods for convex problems [91] but still remain topics of active investigation in the non-convex regime.

Newton-type optimizers have been developed as alternatives to first-order methods. These optimizers can effectively navigate the steep and flat regions of the optimization landscape. By incorporating curvature information in the form of the Hessian matrix, e.g., negative curvature directions, these methods can escape saddle points [19,77,85,87,92–94]. To avoid explicitly forming the Hessian matrices, Hessian-free methods [76,95–97] have been proposed, which only require Hessian-vector products. Arguably, a highly effective, if not the most effective, among these methods is the trust-region based method that comes with attractive theoretical guarantees and is relatively easy to implement [6,19,70,77].

Lying on the spectrum between first and second order methods is Amari’s natural gradient method [78,80]. This method provided a new direction in the context of high-dimensional optimization of probabilistic models. In his seminal work, Amari showed that natural gradient descent yields Fisher efficient estimate of the param-

ters; he subsequently applied the method to multi-layer perceptrons for solving blind source detection problems. However, computing Fisher matrix and its inverse in high-dimensional settings is computationally intractable both in terms of memory and computational resources. RMSProp [25, 98] methods use a diagonal approximation of Fisher matrix of the objective function to compute the descent direction. These methods incur little overhead with regards to diagonal approximation but nevertheless fail to make progress relative to SGD in some cases. Martens et al. [81, 82, 89] proposed the KFAC method, which approximates the natural gradient using Kronecker products of smaller matrices formed during back-propagation. KFAC method and its distributed counterpart [88] have been shown to outperform well tuned SGD in many applications.

In this work, we couple the advantages of trust region and KFAC methods, and propose a stochastic optimization framework involving trust region objective computed on a mini-batch, constrained to directions that are aligned with those obtained from KFAC. Major computational tasks in updating the parameters in our method are Hessian-vector products involving the solution of the trust region sub-problem, as well as finding the KFAC direction. Our Hessian-vector products can be computed at a similar cost as that of gradient computation using back-propagation. Furthermore, the Fisher matrix approximation and its inverse are only needed once every few mini batches thus reducing average iteration cost significantly.

4.3 FITRE

We now present our method, FITRE, which is inspired by [81] and [19, 77], and is formally described in Algorithm 14. At the heart of FITRE lies the stochastic trust-region method using a local quadratic approximation:

$$\min_{\|\mathbf{s}\| \leq \Delta_t} m_t(\mathbf{s}) = \langle \mathbf{g}_t, \mathbf{s} \rangle + \frac{1}{2} \langle \mathbf{s}, \mathbf{H}_t \mathbf{s} \rangle. \quad (4.1)$$

We employ the approach proposed by [77] and use stochastic estimation of the gradient \mathbf{g}_t and Hessian \mathbf{H}_t . The application of Hessian estimate, \mathbf{H}_t , which contains information regarding the curvature of the optimization landscape, has been shown to offer many advantages, including resilience to hyper-parameter tuning and problem ill-conditioning [6, 19]. The step-length, which is governed by the trust-region radius Δ_t is automatically adjusted based on the quality of the quadratic approximation and the amount of descent in the objective function. In practice, (4.1) is approximated by restricting the problem to lower dimensional spaces, e.g., Cauchy condition, which amounts to searching in a one-dimensional space spanned by the gradient. Here, we do the same, however by restricting the sub-problem to the space spanned by the KFAC direction, or its combination with the gradient.

Our choice is motivated by the following observation: when the objective function involves probabilistic models, as is the case in many deep learning applications, natural gradient¹ direction amounts to the steepest descent direction among all possible directions inside a ball measured by KL-divergence between the underlying parametric probability densities. On the contrary, the (standard) gradient represents the direction of steepest descent among all directions constrained in a ball measured by the Euclidean metric [82], which is less informative than the former, though much easier to compute. To alleviate the computational burden of working with the Fisher information matrix and its inverse, Kronecker-product based approximations [81, 89] have shown success in simultaneously preserving desirable properties of the exact Fisher matrix such as invariance to reparametarization and resilience to large batch sizes. Indeed, many empirical studies have confirmed that the natural gradient provides an effective descent direction for optimization of neural networks [76, 81, 82, 89].

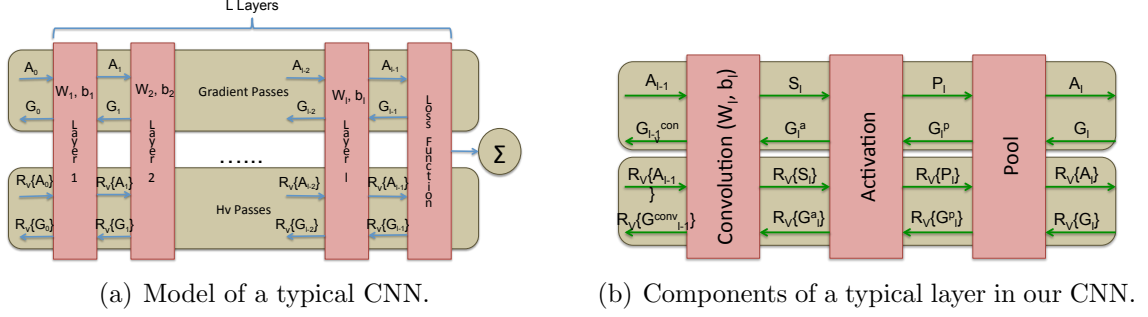


Fig. 4.1.: CNN model and layer composition.

4.3.1 Computational Model

In this work, we consider a typical CNN architecture, as shown in Figure 4.1(a). We assume that the network contains ℓ layers, and each layer can be either a convolution layer or a linear layer. A convolutional layer is composed of convolution and activation functions. It may also optionally contain pooling functions and batch-normalization layers as shown in Figure 4.1(b). Likewise, a linear layer is composed of a linear function and optionally can have activation functions within it as well. These layers are stacked together, so that the output of one layer forms the input of the next layer. The last layer, i.e., ℓ^{th} layer, is connected to the loss function. In this work, we use *softmax cross-entropy* for our loss function, denoted as \mathcal{L} .

Gradient computation is performed in two passes over the network, shown in the top half of the Figure 4.1(a), using the back-propagation algorithm. In the forward pass for the gradient computation, \mathbf{a}_{l-1} and \mathbf{a}_l are, respectively, the input to and the output from the layer l . We bundle a set of sample points, also known as mini-batch, which forms the input to the first layer, $\tilde{\mathbf{A}}_0$. For detailed discussion on memory layout of the input data to the network, $\tilde{\mathbf{A}}_0$, and its processing through out the underlying network we refer readers to the appendix section (4.6). Please note that for analysis purposes all the equations in this section use one sample point for l^{th} -convolution layers (i.e., \mathbf{A}_l , a matrix of dimensions samples \times channels) as well

¹In the following section we discuss in detail the definition of the Fisher matrix and its computation using the KFAC approximations.

as m^{th} -linear layers (i.e., \mathbf{a}_m , a vector), and a detailed discussion is presented in appendix section 4.6. The following equation summarize the operations performed during the forward pass for each of the layers in the network, as shown in Figure 4.1(b).

$$\text{Convolution:} \quad \mathbf{C}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1} \quad (4.2a)$$

$$\text{Activation:} \quad \mathbf{P}_l = \phi(\mathbf{C}_l) \quad (4.2b)$$

$$\text{Pool:} \quad \mathbf{A}_l = \mathcal{P}(\mathbf{P}_l) \quad (4.2c)$$

The convolution operation is represented as a matrix multiplication, as described in (4.2a), where $\bar{\mathbf{W}}_l$ and \mathbf{A}_{l-1} are the matrix of weights and input sample point. Note that the weights matrix, \mathbf{W}_l and bias vector, \mathbf{b}_l , associated with the l^{th} layer of the network are folded into a single matrix $\bar{\mathbf{W}}_l (= [\mathbf{W}_l \mathbf{b}_l])$ by appending the column vector \mathbf{b}_l into \mathbf{W}_l . The output of the convolution operation, \mathbf{C}_l , forms the input to the non-linearity function ϕ and its output, \mathbf{P}_l , is passed to the down sampling function, \mathcal{P} . The output of the pool function, \mathbf{A}_l , forms the input to the next layer $l + 1$ in the model shown in Figure 4.1(a). During the backward pass of the gradient computation, the partial derivatives with respect to inputs to each layer (referred to as gradients throughout this document) are passed in the backward (opposite) direction through the network as shown in Figure 4.1(a). Partial derivatives of layer l , \mathbf{G}_l^{conv} , are fed to the preceding layer $l - 1$ through the network. Inside layer l , the incoming gradient terms \mathbf{G}_{l+1}^{conv} are passed through the down sampling, non-linearity and convolution functions in a *daisy chained* fashion producing outputs \mathbf{G}_l^p , \mathbf{G}_l^a and \mathbf{G}_l^{conv} respectively; see Figure 4.1(b). Along similar lines, the equations for linear layer are as follows:

$$\text{Linear transformation:} \quad \mathbf{s}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1}^\top \quad (4.3a)$$

$$\text{Activation:} \quad \mathbf{a}_l = \phi(\mathbf{s}_l) \quad (4.3b)$$

eqs. 4.3a and 4.3b represent the linear transformation and activation function performed by the linear layer during the forward pass of the network. And, during the backward pass we use \mathbf{g}_{l-1}^d and \mathbf{g}_l^a represent the gradient terms propagated backwards by the linear transformation and activation function within the linear layer.

The framework developed for gradient computations can also be adapted to compute the Hessian-vector products by using the “R-Operator” approach first introduced in [99]. Specifically, the gradient and Hessian of a function, f , are related by:

$$\nabla f(\boldsymbol{\theta} + \boldsymbol{\delta\theta}) = \nabla f(\boldsymbol{\theta}) + \nabla^2 f(\boldsymbol{\theta}) \boldsymbol{\delta\theta} + o(\|\boldsymbol{\delta\theta}\|^2).$$

By choosing $\boldsymbol{\delta\theta} = r\mathbf{v}$ for some $r \in \mathbb{R}$, Hessian-vector product, $\nabla^2 f(\boldsymbol{\theta})\mathbf{v}$, can be obtained using:

$$\nabla^2 f(\boldsymbol{\theta})\mathbf{v} = \lim_{r \rightarrow 0} \frac{\nabla f(\boldsymbol{\theta} + r\mathbf{v}) - \nabla f(\boldsymbol{\theta})}{r} = \left. \frac{d}{dr} \nabla f(\boldsymbol{\theta} + r\mathbf{v}) \right|_{r=0}. \quad (4.4)$$

Now, by defining

$$\mathcal{R}_{\mathbf{v}}\{\mathbf{u}(\boldsymbol{\theta})\} = \left. \frac{d}{dr} \mathbf{u}(\boldsymbol{\theta} + r\mathbf{v}) \right|_{r=0}, \quad (4.5)$$

for any vector valued function \mathbf{u} , we have $\nabla^2 f(\boldsymbol{\theta})\mathbf{v} = \mathcal{R}_{\mathbf{v}}\{\nabla f(\boldsymbol{\theta})\}$. Therefore, by applying $\mathcal{R}_{\mathbf{v}}$ to all the equations evaluated during the gradient computation of the given network, we can compute $\nabla^2 f(\boldsymbol{\theta})\mathbf{v}$ of the loss function associated with the given network. The forward and backward passes through the neural network associated with Hessian-vector computation are shown in the bottom half of Figures 4.1(a) and 4.1(b).

Define $\boldsymbol{\theta} = [\text{vec}(\bar{\mathbf{W}}_1)^\top, \text{vec}(\bar{\mathbf{W}}_2)^\top, \dots, \text{vec}(\bar{\mathbf{W}}_\ell)^\top]^\top$, which is a vector of all the network's parameters, concatenated together and vec operator is used to flatten the matrices by stacking columns together. The loss function, denoted by $\mathcal{L}(y, z)$, is used to measure the disagreement between a prediction z and a target y corresponding to the input-output pair (\mathbf{x}, y) . The training objective function $h(\boldsymbol{\theta})$ is the average of losses $\mathcal{L}(y, f(\mathbf{x}, \boldsymbol{\theta}))$ over all input-target pairs (\mathbf{x}, y) , i.e.,

$$h(\boldsymbol{\theta}) := \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y_i, f(\mathbf{x}_i, \boldsymbol{\theta})).$$

For a given (\mathbf{x}, y) , its corresponding loss is given by the *negative log likelihood* associated with a conditional distribution of y given $z = f(\mathbf{x}, \boldsymbol{\theta})$ as

$$\mathcal{L}(y, z) = -\log r(y|f(\mathbf{x}, \boldsymbol{\theta})) := -\log p(y|\mathbf{x}, \boldsymbol{\theta}).$$

Here, $p(y|\mathbf{x}, \boldsymbol{\theta})$ is the conditional distribution of y given \mathbf{x} that is implied by the neural network and parameterized by $\boldsymbol{\theta}$. Here the parameters are not assumed to be random, but one can extend this model to include priors on $\boldsymbol{\theta}$ and effectively have a Bayesian model. Minimizing the objective function $h(\boldsymbol{\theta})$ is identical to maximizing the likelihood $p(y|\mathbf{x}, \boldsymbol{\theta})$ over the training dataset.

Natural Gradient Computation

For completeness we present an overview of the approximations involved in estimating the natural gradient direction (in the context of linear layers and similar arguments can be made for convolution layer as well). We refer readers to [81, 82] for a detailed discussion on estimation of Fisher information matrix and approximations used in deriving the natural gradient direction.

We define,

$$\mathcal{D}\boldsymbol{\theta} := \frac{d\mathcal{L}(y, f(\mathbf{x}, \boldsymbol{\theta}))}{d\boldsymbol{\theta}} = -\frac{d \log p(y|\mathbf{x}, \boldsymbol{\theta})}{d\boldsymbol{\theta}},$$

$$\mathbf{g}_l^a := \mathcal{D}\mathbf{s}_l,$$

where $\mathcal{D}\boldsymbol{\theta}$ is the gradient of the loss function, which is computed using the conventional back-propagation algorithm and \mathbf{g}_l^a represents the gradients of the loss function w.r.t. the pre-activation inputs of layer l . Since the network defines a conditional distribution $p(y|\mathbf{x}, \boldsymbol{\theta})$, its associated Fisher information matrix is given by

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E} \left[\frac{\partial \log p(y|\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\frac{\partial \log p(y|\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^\top \right] = \mathbb{E} [\mathcal{D}\boldsymbol{\theta} (\mathcal{D}\boldsymbol{\theta})^\top] \quad (4.6)$$

Natural gradient is defined as $\mathbf{F}^{-1}(\boldsymbol{\theta})\nabla h(\boldsymbol{\theta})$. It defines the direction in parameter space that gives the largest change in the objective function per unit change in the model, as measured by the KL-divergence which is measured between the model output distribution and the true label distribution. In the context of this discussion, for simplicity, we drop the dependence of \mathbf{F} and h on $\boldsymbol{\theta}$.

Kronecker-factored Fisher approximation(s) Computing \mathbf{F}^{-1} or $\mathbf{F}^{-1}\nabla h$ is not practical in commonly encountered high-dimensional problems. To this end, Martens et al. propose suitable approximations [81,82]. We summarize these approximations in the following paragraphs for completeness:

$$\begin{aligned} \mathbf{F} &= \mathbb{E} [\mathcal{D}\boldsymbol{\theta} (\mathcal{D}\boldsymbol{\theta})^\top] \\ &= \begin{bmatrix} \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_1)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_1)^\top] & \dots & \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_1)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)^\top] \\ \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_2)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_1)^\top] & \dots & \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_2)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)^\top] \\ \vdots & \ddots & \vdots \\ \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_1)^\top] & \dots & \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)\text{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)^\top] \end{bmatrix} \end{aligned}$$

Thus, \mathbf{F} can be viewed as an $\ell \times \ell$ block matrix with the (i, j) -th block $\mathbf{F}_{i,j}$ given by $\mathbf{F}_{i,j} = \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_i) \text{vec}(\mathcal{D}\bar{\mathbf{W}}_j)^\top]$. Noting that $\mathcal{D}\bar{\mathbf{W}}_i = \mathbf{g}_i^a \bar{\mathbf{a}}_{i-1}^\top$ and that $\text{vec}(\mathbf{u}\mathbf{v}^\top) = \mathbf{v} \otimes \mathbf{u}$ for any two vectors \mathbf{u} and \mathbf{v} , we have $\mathcal{D}\bar{\mathbf{W}}_i = \text{vec}(\mathbf{g}_i^a \bar{\mathbf{a}}_{i-1}^\top) = \bar{\mathbf{a}}_{i-1}^\top \otimes \mathbf{g}_i^a$ and thus we can rewrite $\mathbf{F}_{i,j}$ as

$$\begin{aligned} \mathbf{F}_{i,j} &= \mathbb{E} [\text{vec}(\mathcal{D}\bar{\mathbf{W}}_i) \text{vec}(\mathcal{D}\bar{\mathbf{W}}_j)^\top] \\ &= \mathbb{E} [(\bar{\mathbf{a}}_{i-1} \otimes \mathbf{g}_i^a)(\bar{\mathbf{a}}_{j-1} \otimes \mathbf{g}_j^a)^\top] \\ &= \mathbb{E} [(\bar{\mathbf{a}}_{i-1} \bar{\mathbf{a}}_{j-1}^\top \otimes \mathbf{g}_i^a \mathbf{g}_j^{a\top})] \end{aligned}$$

where $\mathbf{A} \otimes \mathbf{B}$ denotes the Kronecker product between two matrices.

The approximation of \mathbf{F} by $\tilde{\mathbf{F}}$ is defined as follows:

$$\mathbf{F}_{i,j} = \mathbb{E} [(\bar{\mathbf{a}}_{i-1} \bar{\mathbf{a}}_{j-1}^\top \otimes \mathbf{g}_i^a \mathbf{g}_j^{a\top})] \approx \mathbb{E} [\bar{\mathbf{a}}_{i-1} \bar{\mathbf{a}}_{j-1}^\top] \otimes \mathbb{E} [\mathbf{g}_i^a \mathbf{g}_j^{a\top}] = \mathbf{K}_{i-1,j-1} \otimes \mathbf{G}_{i,j}^a = \tilde{\mathbf{F}} \quad (4.7)$$

where $\mathbf{A}_{i-1,j-1} = \mathbb{E} [\bar{\mathbf{a}}_{i-1} \bar{\mathbf{a}}_{j-1}^\top]$ and $\mathbf{G}_{i,j}^a = \mathbb{E} [\mathbf{g}_i^a \mathbf{g}_j^{a\top}]$.

This gives the following:

$$\tilde{\mathbf{F}} = \begin{bmatrix} \mathbf{K}_{0,0} \otimes \mathbf{G}_{1,1}^a & \mathbf{K}_{0,1} \otimes \mathbf{G}_{1,2}^a & \dots & \mathbf{K}_{0,\ell-1} \otimes \mathbf{G}_{1,\ell}^a \\ \mathbf{K}_{1,0} \otimes \mathbf{G}_{2,1}^a & \mathbf{K}_{1,1} \otimes \mathbf{G}_{2,2}^a & \dots & \mathbf{K}_{1,\ell-1} \otimes \mathbf{G}_{1,\ell}^a \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{K}_{\ell-1,0} \otimes \mathbf{G}_{\ell,1}^a & \mathbf{K}_{\ell-1,1} \otimes \mathbf{G}_{\ell,2}^a & \dots & \mathbf{K}_{\ell-1,\ell-1} \otimes \mathbf{G}_{\ell,\ell}^a \end{bmatrix}, \quad (4.8)$$

which has the form of what is known as a Khatri-Rao product in multivariate statistics.

Note that the expectation of Kronecker product is not equal to Kronecker product of expectations. The above approximation, $\tilde{\mathbf{F}} \approx \mathbf{F}$, is a major approximation, but nevertheless works well in practice.

Approximating $\tilde{\mathbf{F}}^{-1}$ as block-diagonal is equivalent to approximating $\tilde{\mathbf{F}}$ as block-diagonal. A natural choice of such approximation, $\check{\mathbf{F}} \approx \tilde{\mathbf{F}}$, is to take the block-diagonal of $\check{\mathbf{F}}$ to be that of $\tilde{\mathbf{F}}$. This gives the matrix

$$\begin{aligned}\check{\mathbf{F}} &= \text{diag} \left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \dots, \tilde{F}_{\ell-1,\ell-1} \right) \\ &= \text{diag} \left(\mathbf{K}_{0,0} \otimes \mathbf{G}_{1,1}^a, \mathbf{K}_{1,1} \otimes \mathbf{G}_{2,2}^a, \dots, \mathbf{K}_{\ell-1,\ell-1} \otimes \mathbf{G}_{\ell,\ell}^a \right)\end{aligned}$$

Using the identity, $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$, the inverse of $\check{\mathbf{F}}$ is given by

$$\check{\mathbf{F}}^{-1} = \text{diag} \left(\mathbf{K}_{0,0}^{-1} \otimes \mathbf{G}_{1,1}^{a^{-1}}, \mathbf{K}_{1,1} \otimes \mathbf{G}_{2,2}^{a^{-1}}, \dots, \mathbf{K}_{\ell-1,\ell-1} \otimes \mathbf{G}_{\ell,\ell}^{a^{-1}} \right) \quad (4.9)$$

Thus computing $\check{\mathbf{F}}^{-1}$ amounts to computing inverses of 2ℓ smaller matrices. Then approximated natural-gradient, $\mathbf{u} = \check{\mathbf{F}}^{-1}\mathbf{v}$, is given by the following (using the identity $(\mathbf{A} \otimes \mathbf{B}) \text{vec}(\mathbf{X}) = \text{vec}(\mathbf{B}\mathbf{X}\mathbf{A}^\top)$):

$$\mathbf{U}_i = \mathbf{G}_{i,i}^{a^{-1}} \mathbf{V}_i \mathbf{K}_{i-1,i-1}^{-1} \quad (4.10)$$

where \mathbf{v} maps to $(\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_\ell)$ and \mathbf{u} maps to $(\mathbf{U}_1, \mathbf{U}_2, \dots, \mathbf{U}_\ell)$ and $\boldsymbol{\theta}$ maps to $(\bar{\mathbf{W}}_1, \bar{\mathbf{W}}_2, \dots, \bar{\mathbf{W}}_\ell)$

Natural gradient using Kronecker Factored Approximate Curvature matrix: We define:

$$\mathbb{E} \left[\text{vec}(\mathcal{D}\bar{\mathbf{W}}_l) \text{vec}(\mathcal{D}\bar{\mathbf{W}}_l)^\top \right] \approx \boldsymbol{\Psi}_{l-1} \otimes \boldsymbol{\Gamma}_l \triangleq \check{\mathbf{F}}_l \quad (4.11)$$

where $\boldsymbol{\Psi}_{l-1} = \mathbb{E} [\bar{\mathbf{a}}_{l-1} \bar{\mathbf{a}}_{l-1}^\top]$ and $\boldsymbol{\Gamma}_l = \mathbb{E} [\mathbf{g}_l^a \mathbf{g}_l^{a^\top}]$ denote the second moment matrices of the activation and pre-activation derivatives, respectively.

To invert $\check{\mathbf{F}}$, we use the fact that: (i) we can invert a block-diagonal matrix by inverting each of the blocks, and (ii) the Kronecker product satisfies the identity $(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$:

$$\check{\mathbf{F}}^{-1} = \begin{bmatrix} \Psi_0^{-1} \otimes \Gamma_1^{-1} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \Psi_{\ell-1}^{-1} \otimes \Gamma_{\ell-1}^{-1} \end{bmatrix} \quad (4.12)$$

The approximate natural gradient $\check{\mathbf{F}}^{-1} \nabla h$ can be computed as follows:

$$\check{\mathbf{F}}^{-1} \nabla h = \begin{bmatrix} \text{vec} \left(\Gamma_1^{-1} (\nabla_{\bar{\mathbf{w}}_1} h) \Psi_0^{-1} \right) \\ \vdots \\ \text{vec} \left(\Gamma_{\ell}^{-1} (\nabla_{\bar{\mathbf{w}}_{\ell}} h) \Psi_{\ell-1}^{-1} \right) \end{bmatrix} \quad (4.13)$$

A common multiple of the identity matrix is added to \mathbf{F} for two reasons: First, as a regularization parameter, which corresponds to a penalty of $\frac{1}{2} \lambda \boldsymbol{\theta}^\top \boldsymbol{\theta}$. This translates to $\mathbf{F} + \lambda \mathbf{I}$ to approximate the curvature of the regularized objective function. The second reason is to use it as a damping parameter to account for multiple approximations used to derive $\check{\mathbf{F}}$, which corresponds to adding $\gamma \mathbf{I}$ to the approximate curvature matrix. Therefore, we aim to compute: $\left[\check{\mathbf{F}} + (\lambda + \gamma) \mathbf{I} \right]^{-1} \nabla h$.

Since adding the term $(\lambda + \gamma) \mathbf{I}$ breaks the Kronecker factorization structure, an approximated version is used for computational purposes, which is as follows:

$$\check{\mathbf{F}}_{\ell} + (\lambda + \gamma) \mathbf{I} \approx \left(\Psi_{\ell-1} + \pi_{\ell} \sqrt{\lambda + \gamma} \mathbf{I} \right) \otimes \left(\Gamma_{\ell} + \frac{1}{\pi_{\ell}} \sqrt{\lambda + \gamma} \mathbf{I} \right) \quad (4.14)$$

for some π_{ℓ} .

Updating KFAC Block Matrices Block matrices, Ψ_l and Γ_l , are typically updated using a momentum term to capture the variance in input samples across successive mini batches. If sample points across the dataset are well correlated, with little variance among the sample points, the the inverse block matrices, Ψ_l^{-1} and Γ_l^{-1} , need not be updated for every mini batch. “KFAC Update Frequency” is the frequency with which these inverse block matrices are updated is typically decided based on the size of the input dataset as well as the correlation among the sample points. For bootstrapping the optimizer, we could either use a larger sample of the dataset, like $5 \times$

the mini-batch size, or use the very first mini batch itself for computing the block inverses.

4.3.2 Algorithm

Algorithm. 14, describes a realization of our proposed method in trust-region settings. First, the natural gradient direction, \mathbf{p}_t is computed and used in determining the step-size using the quadratic approximation of the objective function at \mathbf{p}_t , whose closed form solution is $(\Delta / \|\mathbf{H}_t \mathbf{p}_t + \mathbf{g}_t\|) (\mathbf{H}_t \mathbf{p}_t + \mathbf{g}_t)$ (Note that gradient, \mathbf{g}_t , can also be used to estimate the step size and may yield a better descent direction in some cases). Once the step-size, η is determined, ρ_t is computed over the same mini-batch to determine the trust-region radius as well as the iterate update. These steps are repeated until desired generalization is achieved. Note that we can compare the efficiency of natural-gradient direction with that of the standard gradient and use the appropriate one at each iteration, this is referred to as “KFAC + gradient” in this algorithm.

4.4 Experiments

In this section, we present results from our experiments with the proposed method. In the following paragraphs we provide an in-depth analysis about the behavior of our proposed methods and compare it with well-tuned Nesterov-accelerated SGD as well as a quasi-Newton method in the context of well known CNNs.

Hardware and Software Platform for Experiments. All our simulations are executed on NVIDIA’s Tesla V100 GPUs configured with 16GB global RAM on CUDA 9.0 runtime platform. These machines are configured with Intel Xeon CPUs with 192 GB of RAM. Our code is implemented in C++ and we primarily used *cublas* for GEMM operations. SGD results are executed on pyTorch 1.0.0 installation with python 3.1 as the front-end. The code base is available for download at [100].

Algorithm 14: KFAC-STR Method

Input :

- Starting point \mathbf{x}_0
- Initial trust-region radius: $0 < \Delta_0 < \infty$
- KFAC parameters: damping parameter ($\gamma \geq 0$), moving average ($0 < \theta < 1$)

Result: \mathbf{x}_t - direction to be used to update model parameters.

foreach $t = 0, 1, \dots$ **do**

Set the approximate gradient \mathbf{g}_t and Hessian \mathbf{H}_t

/ Compute the approximated Inverse Fisher \times gradient, a.k.a *natural-gradient* */*

Obtain natural-gradient direction \mathbf{r}_t , as described in [81,82]

Case 1: KFAC

$$\eta_t = \arg \min \|\eta \mathbf{r}_t\| \leq \Delta_t m(\eta \mathbf{p}_t) = \eta \mathbf{g}_t^T \mathbf{r}_t + \frac{\eta^2}{2} \mathbf{r}_t^T \mathbf{H}_t \mathbf{r}_t$$

$$\mathbf{s}_t = \eta_t \mathbf{r}_t$$

Case 2: KFAC + Gradient

$$\eta_t = \arg \min \|\eta \mathbf{r}_t\| \leq \Delta_t m(\eta \mathbf{p}_t) = \eta \mathbf{g}_t^T \mathbf{r}_t + \frac{\eta^2}{2} \mathbf{r}_t^T \mathbf{H}_t \mathbf{r}_t$$

$$\alpha_t = \arg \min \|\alpha \mathbf{g}_t\| \leq \Delta_t m(\alpha \mathbf{g}_t) = \alpha \mathbf{g}_t^T \mathbf{g}_t + \frac{\alpha^2}{2} \mathbf{g}_t^T \mathbf{H}_t \mathbf{g}_t$$

$$\mathbf{s}_t = \arg \min \mathbf{s} \in \{\eta_t \mathbf{r}_t, \alpha_t \mathbf{g}_t\} m(\mathbf{s})$$

Set $\rho_t \triangleq \frac{h_t(\theta_t) - h_t(\theta_t + \mathbf{s}_t)}{-m(\mathbf{s}_t)}$, ($h_t(\cdot)$ are evaluated on the same mini-batch as \mathbf{g}_t and \mathbf{H}_t).

if $\rho_t \geq 0.75$ **then**

$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{s}_t$ and $\Delta_{t+1} = \min\{2\Delta_t, \Delta_{max}\}$

end

else if $\rho_t \geq 0.25$ **then**

$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{s}_t$ and $\Delta_{t+1} = \Delta_t$

end

else

$\mathbf{w}_{t+1} = \mathbf{w}_t$ and $\Delta_{t+1} = \Delta_t/2$

end

end

Datasets. We use CIFAR10, CIFAR100 [101], and tiny ImageNet [60] datasets for validating KFAC-STR method using several CNNs mentioned below. The details of each of these datasets are described in Table 4.1. CIFAR10 is a well conditioned dataset, whereas ImageNet is the largest dataset we experiment with. CIFAR100 is conditioned between CIFAR10 (relatively well conditioned) and ImageNet (ill-conditioned).

Table 4.1.: Description of the datasets used in our experiments

Dataset	Features	No. of Training Samples	No. of Testing Samples
CIFAR-10	3096	50,000	10,000
CIFAR-100	3096	50,000	10,000
Tiny ImageNet	12288	100,000	9,000

Hyper-parameter Tuning. Our proposed method has two easily tunable parameters – damping parameter (γ) and maximum trust-region radius (δ). All experiments presented here use (with increments of 10) $\gamma \in \{1e^{-3}, \dots, 1e^2\}$ and $\delta \in \{1e^{-2}, \dots, 1e^2\}$. Regularization term (λ) is set to the following values $\{1e^{-4}, 1e^{-5}, 1e^{-6}\}$. Trust-region radius, Δ , is capped by δ and is doubled when ρ , ratio of observed model reduction and expected model reduction, is ≥ 0.75 and is halved when it is < 0.25 ; otherwise it remains the same. SGD uses learning rate as the hyper-parameter, which is in the following range $\{1e^{-6}, \dots, 1e^2\}$. Both methods use the Nesterov-accelerated momentum term as 0.9. All the experiments are run for 50 epochs except for Imagenet dataset, where maximum number of epochs is set to 25. Mini-batch size is set to 200 for both the methods.

Plots presented in this section are selected as follows: SGD curves in all the plots always use the learning rate that yields highest test accuracy among all the learning rates. FITRE curves always use the selection of hyper-parameters that yield the maximum average test accuracy.

Convolutional Neural Networks. We experiment with VGG11, VGG16, and VGG19 CNN’s for our validation purposes. Each of these CNN’s architectures is described in Table 4.2.

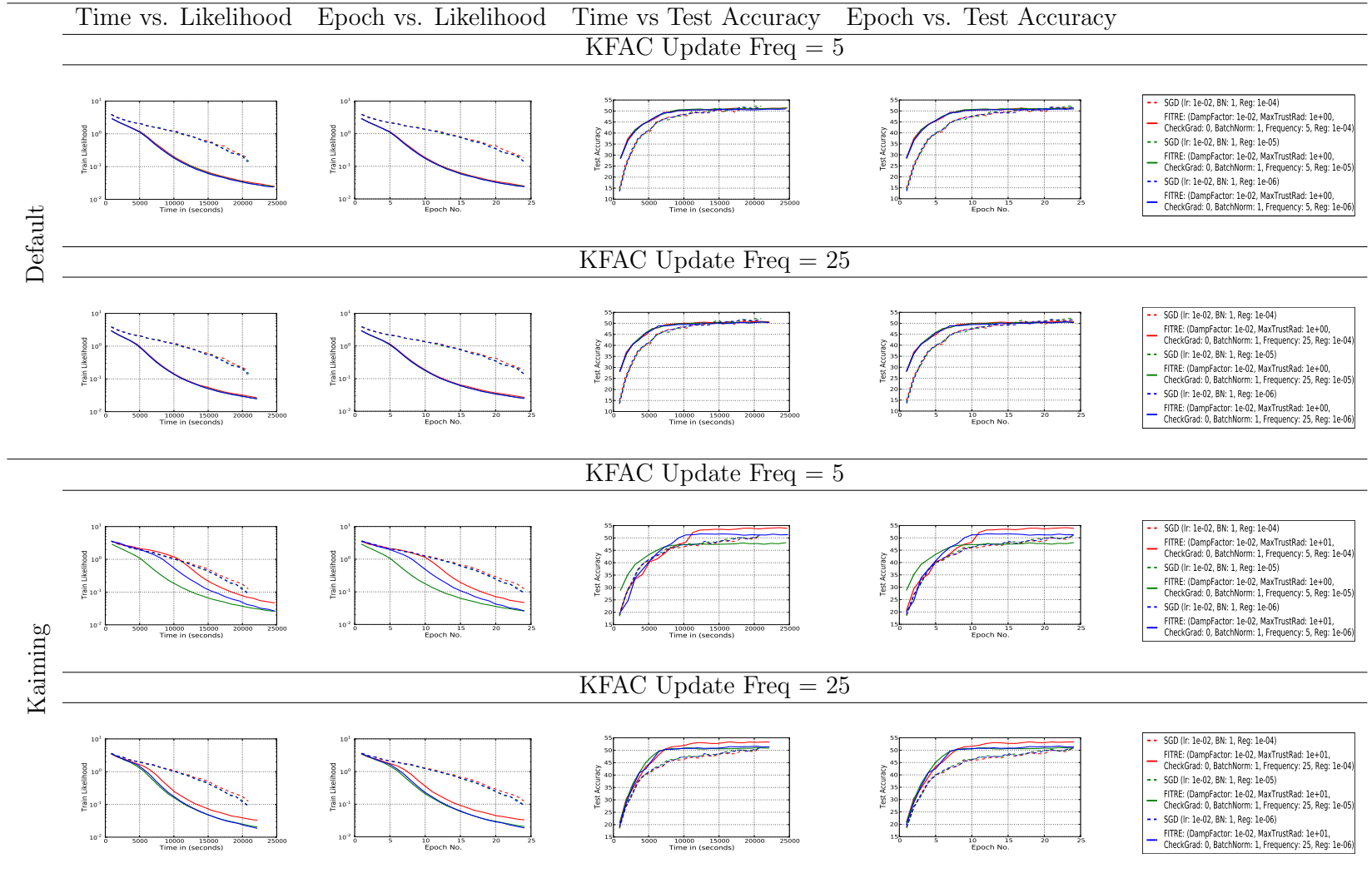
Table 4.2.: Various Convolution Neural Networks used in our experiments. α is 512 when CIFAR10 and CIFAR100 are used, and for Imagenet, it is 2048. β is 10 for CIFAR10, 100 for CIFAR100, and 200 for ImageNet. These networks can be easily adapted for embedding BatchNormalization layers (typically after the convolution layer).

CNN	Layer Description	
AlexNet	Conv(3, 64), Swish, MaxPool	conv_kernel(k=5,s=1,p=2),
	Conv(64, 64), Swish, MaxPool	pool_kernel(k=3,
	Linear(4096, 384), Swish	s=2,p=1)
	Linear(384, 192), Swish	
	Linear(192, β)	
VGG11	Conv(3, 64), Swish, MaxPool	
	Conv(64, 128), Swish, MaxPool	
	Conv(128, 256), Swish	
	Conv(256, 256), Swish, MaxPool	conv_kernel(k=3,s=1,p=1),
	Conv(256, 512), Swish	pool_kernel(k=2,s=2)
	Conv(512, 512), Swish, MaxPool	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
	Linear(α , β)	
VGG16	Conv(3, 64), Swish	
	Conv(64, 64), Swish, MaxPool	
	Conv(64, 128), Swish	
	Conv(128, 128), Swish, MaxPool	
	Conv(128, 256), Swish	conv_kernel(k=3,s=1,p=1),
	Conv(256, 256), Swish, MaxPool	pool_kernel(k=2,s=2)
	Conv(256, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
VGG19	Linear(α , β)	
	Conv(3, 64), Swish	
	Conv(64, 64), Swish, MaxPool	
	Conv(64, 128), Swish	
	Conv(128, 128), Swish, MaxPool	
	Conv(128, 256), Swish	
	Conv(256, 256), Swish	
	Conv(256, 256), Swish	
	Conv(256, 256), Swish, MaxPool	conv_kernel(k=3,s=1,p=1),
	Conv(256, 512), Swish	pool_kernel(k=2,s=2)
	Conv(512, 512), Swish	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
	Linear(α , β)	

Table 4.3.: Comparison of VGG11 using ImageNet dataset

	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy
	KFAC Update Freq = 5			
Default				
	KFAC Update Freq = 25			
Kaiming				
	KFAC Update Freq = 25			

Table 4.4.: Comparison of VGG16 using ImageNet dataset



Results on the Imagenet Dataset. Tables. 4.3 and 4.4 show the plots for the Imagenet dataset using VGG11 and VGG16 CNNs, respectively. In these tables, we show the generalization errors plotted against wall-clock time and against number of epochs in Columns 3 and 4, respectively, and negative log-likelihood (NLL) using softmax cross-entropy loss function against wall-clock time and against number of epochs in Columns 1 and 2, respectively. KFAC update frequency is set to 5 (mini-batches) for the first row and for the second row, it is set to 25. All plots (in the first two rows) use *default* initialization, as defined in pyTorch which is a uniform distribution, in these two tables. Corresponding results using *kaiming* initialization [102], on a high-level this initialization is based on random gaussian distribution, are shown in Rows 3 and 4 of both the tables.

The following conclusions can be made from the plots in Tables. 4.3 and 4.4: (i) FITRE method minimizes the likelihood function to a significantly smaller value compared to well-tuned SGD, and at any given wall-clock instance (FITRE method yields better NLL value compared to SGD), (ii) kaiming initialization yields superior generalization errors compared to default initialization of the CNNs, (iii) contrary to expectations KFAC update frequency of 25 yields better generalization errors relative to more frequent updates, (iv) with increasing network complexity, VGG16 compared to VGG11, FITRE method yields significantly better generalization errors compared to SGD, showcasing its superior scaling characteristics compared to SGD; and (v) default initialization is relatively immune to ℓ_2 regularization compared to kaiming initialization.

For VGG16 network with kaiming initialization and KFAC update frequency of 25 we observe that to attain 50% test accuracy FITRE (with $1e-6$ regularization) takes ≈ 6500 seconds compared to ≈ 20500 seconds for SGD (for all regularizations used); a speedup of 3.2 over SGD. Furthermore, when regularization is set to $1e^{-4}$ FITRE achieves 53.5% test accuracy whereas SGD fails to obtain similar accuracy. Similar arguments can be made for the VGG11 network as well. This shows that even though FITRE is computationally more expensive on a per-iteration basis, it

yields significantly better results in shorter time compared to SGD. This can be attributed to better descent direction (SGD’s gradient vs. FITRE ’s natural gradient) and an adaptive second-order approximated learning rate computation within the trust-region framework used by the FITRE method. Contrary to expectations we notice that for VGG11 CNN and default initialization, FITRE ’s execution of 50 epochs takes less time compared to SGD for KFAC update frequency 25. FITRE makes two passes over the network (one forward and backward pass for gradient computation and another pass for Hessian-vector product computation used to compute the learning rate in the trust-region framework). One would expect that SGD is atleast twice as fast as FITRE on the wall-clock time(one a per-iteration basis). We note that SGD’s pyTorch implementation uses *auto-differentiation* to compute the gradient of the given network, whereas our implementation of the FITRE method is R-operator based (as proposed by Perlmutter et. al [99]). At a finer level, we note from our previous experience with the pyTorch platform [91, 103], that memory management on the GPU is not efficient. pyTorch allocates and frees memory very often and tends to persist very little information on the device. Even though FITRE makes two passes over the network and computes inverses of smaller matrices at each layer of the network (for computing the inverse of the KFAC block matrices) our implementation persists relevant information on the GPU memory. Coupled with our efficient implementation of the R-operator based Hessian-vector product, we can significantly reduce the computation cost associated with each mini-batch. In addition, our proposed method is a true *stochastic online* method in which there is no dependence on any part of the dataset other than the current mini-batch during its entire execution time, compared to state-of-the-art existing second-order methods [64, 83].

We notice that default initialization is immune to regularization for both networks (VGG11 and VGG16), and for both methods (FITRE and SGD). These two methods show negligible changes in NLL function values (as well as generalization errors) while the FITRE method yields superior results compared SGD for significant part of the execution. At the end of the execution SGD tends to achieve similar generaliza-

tion errors compared to FITRE but on minimizing the NLL function FITRE always achieves superior results. However, when using kaiming initialization, based on random gaussian distribution, for both the networks, we notice that regularization helps in achieving superior generalization errors for FITRE method (with VGG11 network, KFAC update frequency set to 25 and regularization of $1e^{-6}$) compared to SGD. But in all cases, FITRE method yields superior results when the underlying model does not use any regularization. Compared to the FITRE method, SGD is relatively immune to kaiming initialization as well, as shown in plots in columns 1 and 2 of Tables. 4.3 and 4.4. Notice that there is very little change in objective function value throughout the simulations.

KFAC update frequency is a hyper parameter used to control the frequency with which the block matrix inverses are computed at each layer of the network. These block inverses are used to compute the natural gradient direction eventually for each mini-batch. Since these blocks approximate the *Fisher matrix* of the loss function, they are updated once every few mini-batches. Martens et. al. [81,82] argues that more frequent updates of these block inverses makes them too rigid and may lead to overfitting. Using larger values for this update frequency has the effect of a regularizer on the underlying model, and helps in avoiding overfitting. As an added advantage, this dependence of the FITRE method reduces its computation cost (note also that the computation of block inverses can be delegated to slave processing units, if available, further reducing the computation cost thereby decreasing the time for processing each mini-batch). This is also one of the reasons why our proposed method scales well with increasing network complexity. We note that for VGG16 (with kaiming initialization), a larger and more complex network compared to VGG11, FITRE method yields superior generalization errors as well as minimizing objective function compared to SGD.

Table 4.5.: Comparison of VGG11 using cifar100 dataset

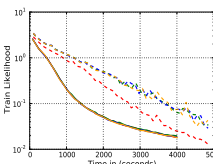
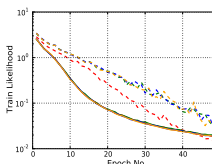
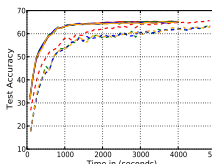
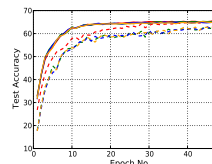
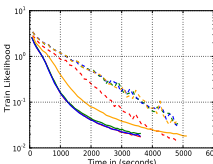
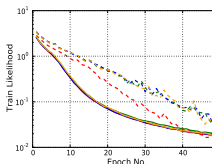
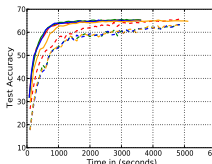
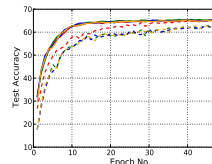
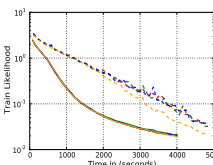
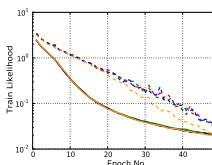
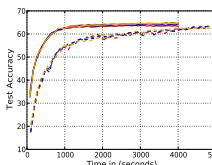
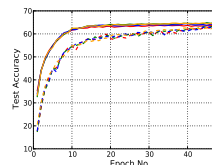
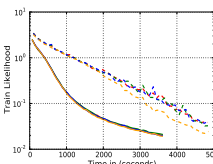
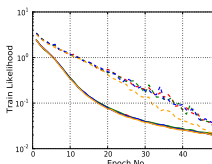
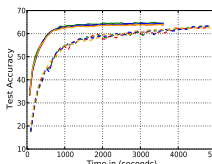
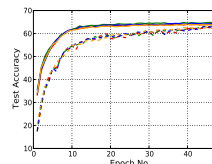
	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy	
	KFAC Update Freq = 5				
Default					<div><ul style="list-style-type: none">SGD (lr: 1e-02, BN: 1, Reg: 0e+00)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00)SGD (lr: 1e-01, BN: 1, Reg: 1e-04)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04)SGD (lr: 1e-01, BN: 1, Reg: 1e-05)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05)SGD (lr: 1e-01, BN: 1, Reg: 1e-06)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)</div>
	KFAC Update Freq = 25				
KFAC Update Freq = 5					<div><ul style="list-style-type: none">SGD (lr: 1e-02, BN: 1, Reg: 0e+00)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00)SGD (lr: 1e-01, BN: 1, Reg: 1e-04)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04)SGD (lr: 1e-01, BN: 1, Reg: 1e-05)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05)SGD (lr: 1e-01, BN: 1, Reg: 1e-06)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-06)</div>
	KFAC Update Freq = 10				
Kaiming					<div><ul style="list-style-type: none">SGD (lr: 1e-01, BN: 1, Reg: 0e+00)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00)SGD (lr: 1e-01, BN: 1, Reg: 1e-04)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04)SGD (lr: 1e-01, BN: 1, Reg: 1e-05)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05)SGD (lr: 1e-02, BN: 1, Reg: 1e-06)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)</div>
					<div><ul style="list-style-type: none">SGD (lr: 1e-01, BN: 1, Reg: 0e+00)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00)SGD (lr: 1e-01, BN: 1, Reg: 1e-04)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04)SGD (lr: 1e-01, BN: 1, Reg: 1e-05)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05)SGD (lr: 1e-02, BN: 1, Reg: 1e-06)FITRE (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-06)</div>

Table 4.6.: Comparison of VGG16 using cifar100 dataset

	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy
	KFAC Update Freq = 5			
Default				
	KFAC Update Freq = 25			
Kaiming				
	KFAC Update Freq = 10			
Damping				
	KFAC Update Freq = 10			

Table 4.7.: Comparison of VGG19 using cifar100 dataset

	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy
	KFAC Update Freq = 5			
Default				
	KFAC Update Freq = 25			
Kaiming				
	KFAC Update Freq = 10			
	KFAC Update Freq = 5			

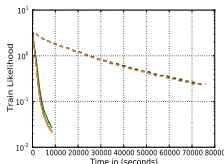
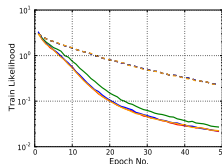
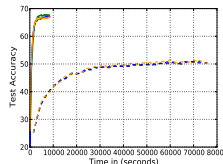
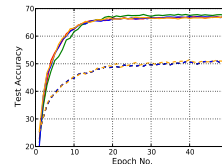
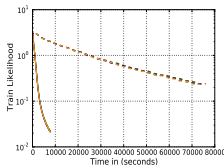
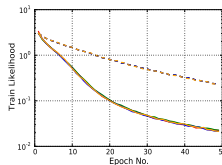
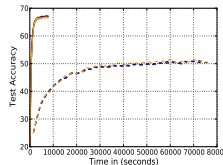
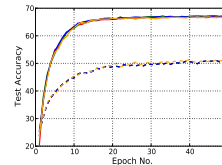
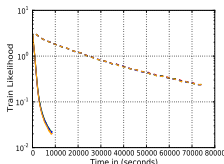
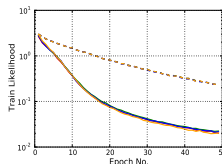
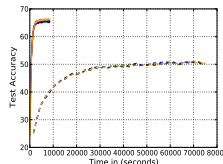
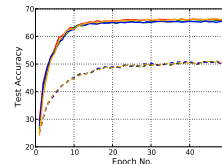
CIFAR100 Dataset Results Table 4.5 shows plots for VGG11 network using the CIFAR100 dataset. We show the generalization errors plots vs. wall clock time and vs. number of epochs in Columns 3 and 4 respectively, and NLL vs. wall-clock time and vs. number of epochs in Columns 1 and 2, respectively. Note that the first row uses the KFAC update frequency of 5 (mini-batches) and the second row uses the KFAC update frequency of 10. All of these plots (in the first two rows) use *default* initialization for the respective CNNs. Corresponding results using *kaiming* initialization are shown in Rows 3 and 4. Tables. 4.6 and 4.7 show plots for VGG16 and VGG19 CNNs.

From the VGG19 networks results shown in Table. 4.7, we clearly notice that the use of ℓ_2 -regularization adversely affects the behavior of SGD optimizer when default initialization is used. We notice that the behavior of SGD optimizer, in the objective function and generalization error plots, without any regularization yields superior results compared to SGD optimizer using non-zero regularization terms. However, for kaiming initialization we do not see any noticeable changes in the behavior of SGD with and without using any regularization terms. From the generalization error plots, we see that the FITRE method achieves $\approx 5\times$ speedup over SGD in the case of kaiming initialization (irrespective of the KFAC update frequency) and, the corresponding speedup in the case of default initialization is $\approx 4\times$. However, notice that in the NLL plots for both types of initializations, FITRE method achieves significantly better results compared to SGD (an order of magnitude better). Furthermore, as seen in the results for the Imagenet dataset, we see that higher KFAC update frequency lowers the time for processing the mini-batches, as well as the time per epoch. Notice that the simulation completion time for FITRE method is lower for KFAC update frequency of 10 compared to the other frequency irrespective of the type of initialization used by the networks. Similar to VGG19 results, ℓ_2 -regularization plays similar role in the behavior of VGG16, as can be seen in Table. 4.6 for both types of initialization. The FITRE method achieves a speedup of $\approx 4\times$ to $\approx 5\times$ over SGD for this network. Table. 4.5 shows the results for the VGG11. Contrary to results from the

other two networks we notice that for VGG11 the objective function values of both methods are closer, indicating that SGD optimizer yielding results similar to that of FITRE method at the end of the simulation as seen in columns 1 and 2. However note that FITRE method achieves superior results in the first few epochs, irrespective of the KFAC update frequency and network initialization type, compared to SGD and as simulation progress SGD tends to achieve similar results at those of FITRE at the end of the simulations.

Remark 5 *The quality of natural-gradient descent direction is effective and second-order approximated trust-region constrained optimization yields larger step sizes at the beginning of the execution. Aided by these two factors, the FITRE method produces better parameter updates in the same amount of wall-clock time compared to SGD, and achieves significantly better generalization errors in the first few epochs. The FITRE method only needs a few epochs to attain near saturation results compared to SGD which needs a larger number of epochs to match the results of FITRE . Efficient implementation and effective use of GPU resources establishes our FITRE method, a truly second-order method, as a suitable alternative to widely used first-order methods like SGD.*

Table 4.8.: Comparison of VGG16 using cifar100 dataset with a quasi-Newton Method (L-BFGS).

	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy
	KFAC Update Freq = 5			
Default				
	KFAC Update Freq = 10			
Kaiming				
	KFAC Update Freq = 10			
Defining				
	KFAC Update Freq = 10			

Comparison with quasi-Newton Method (L-BFGS) In this paragraph we elaborate on the comparison of FITRE method against a widely known quasi-Newton method, which is L-BFGS. Our BFGS implementation is in pyTorch and it uses a history of 20 prior gradients in generating the Hessian approximation used in computing the descent direction. And for step size estimation we use cubic interpolation method as described in section 3.4 of the book from Jorge Nocedal [1]. In the plots shown in Table. 4.8 we describe the comparison results between FITRE and L-BFGS methods. For FITRE method, we keep the KFAC update frequency and regularization constant and select the best performing set of hyper parameters yielding the highest test accuracy for comparison against the BFGS method.

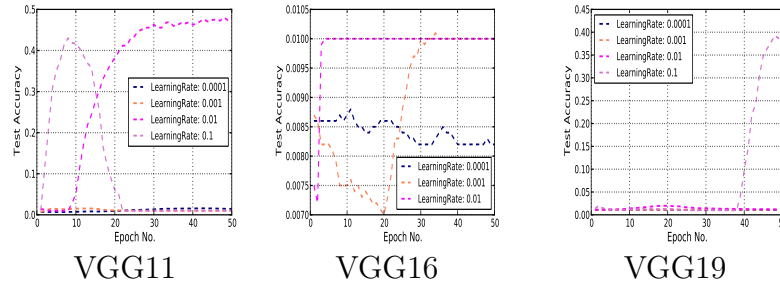
From the plots in Table. 4.8 we clearly notice that FITRE method is orders of magnitude faster compared to BFGS method, $\approx 8\times$ faster in almost all cases. This is because FITRE uses our CUDA framework while BFGS method is implemented in pyTorch. Both these methods use GPUs for computation but in a significantly contrasting manner. pyTorch is a general purpose framework which is optimized for multi-user multi-process environment which inherently optimizes the GPU device usage for simultaneous use GPUs for many users. In that process, all the operations on GPUs are executed in a transactional manner. This means that whenever GPU computation is deemed necessary all the associated data is moved on to the GPU device and computation is initiated and once completed the results are moved back to the CPU memory space. Because of this reason, which is repeated hundreds of times over the course of the simulation, we notice a significant deterioration with the pyTorch version of BFGS solver (resulting in $8\times$ slower compared to FITRE method). In addition, FITRE method uses better tuning of thread block size for individual CUDA kernels, highly optimized implementation of convolution, activation and pooling functions along with their first- and second-derivatives which are used during the computation of Fisher information statistics as well as Hessian-vector products.

In terms of test accuracy, we clearly notice that irrespective of parameter initialization method and KFAC update frequency FITRE method achieves significantly

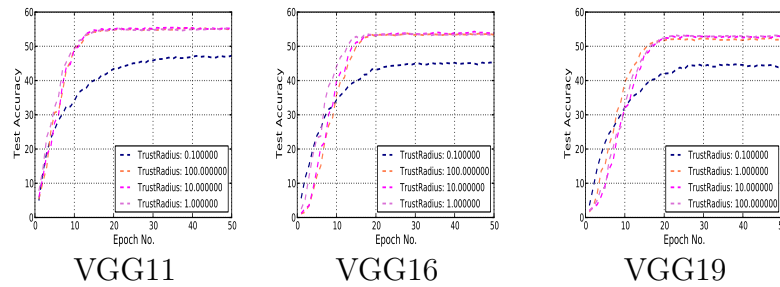
superior test accuracy compared to BFGS method. This can be attributed to the quality of natural gradient as estimated by the kronecker approximated Fisher information matrix as well as the crude estimation of the Hessian used in estimating the descent direction during the optimization of the objective function. Apart from the above mentioned differences between BFGS and FITRE methods we also notice that regularization term does not play a significant role in either improving the generalization error or time consumed during the course of the simulation for either of the optimizers.

Table 4.9.: Behavior of FITRE and SGD without regularization on CIFAR100 dataset. FITRE method uses an update frequency of 5 and “KFAC + gradient” option is turned off in these set of simulations. VGG networks in this table does not use batch-normalization function.

SGD with various learning rates on VGG Networks

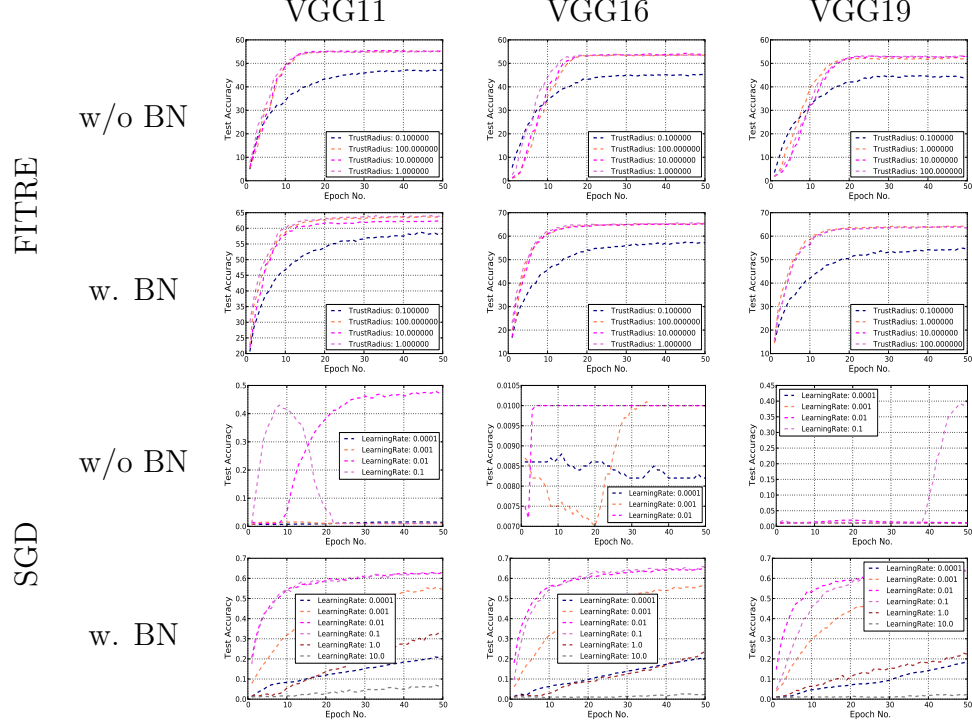


FITRE with varying trust-region radius on VGG Networks



Robustness Results. The FITRE method’s two main hyper-parameters – damping term and maximum trust-region radius can be easily estimated. Typical values

Table 4.10.: Reparameterization Invariance results for SGD and FITRE methods. VGG networks in these experiments use 0 regularization. And for FITRE method we use the KFAC update frequency of 5 and use only the natural gradient direction as the descent direction (KFAC + gradient option is not used in these experiments).



for trust-region radius are between 1e-1 and 10. Similarly, damping parameter's well behaved region is between 1e-2 and 10. For most of the combinations, the FITRE method behaves in a consistent manner, and can yield almost identical results, as shown in This behavior contributes significantly to the resilient behavior of the FITRE method w.r.t to minor changes in the hyper-parameters Table. 4.9. However, we notice that SGD's behavior changes significantly with slight changes in learning rate. This is one of the major challenges that a developer runs into with first-order methods, namely hyper-parameter space that is difficult to tune.

Invariance to Re-parameterization. With the computation of block inverses and ultimately the natural gradient itself, expectation of the inputs and outputs of the convolution layer is deeply embedded into the KFAC approximation itself. Because

of this, additional batch normalizations are irrelevant to the FITRE method’s performance in minimizing the objective function. However, first-order methods like SGD need to have batch normalization layers embedded into the underlying model itself (typically right after the convolution layer). These results are presented in Table. 4.10. We notice that without the use of batch normalization layers, the SGD method tends to diverge or does not make significant progress in optimizing the likelihood of the underlying model. However, the FITRE method is robust, insensitive to small changes in its hyper-parameters, as well as invariant to re-parameterization. Typically, implementation of batch normalization is computationally expensive, and as the network complexity increases, its execution time contributes significantly to the over all simulation time. There is no need for such layers in case of the FITRE method, which significantly helps reduce the processing time for each mini batch (and simulation time for the network).

4.5 Conclusions and Future Work

In this work we proposed an online second-order stochastic method for optimizing non-convex objective functions (likelihood functions). Through extensive experiments using real world datasets, we have shown that our proposed method outperforms existing first-order methods in wall-clock time and convergence rates. We have also shown that our proposed method achieves significantly better generalization errors and minimizes the objective function, beyond those achieved by a well-tuned SGD method. With computationally expensive operations and limited availability of memory on GPUs single node applications can only be used with small batch sizes. This provides us an opportunity to extend our proposed method to distributed environments, and by deploying large batch-sizes, our proposed method can yield significantly better results in much shorter times, which would be extremely hard to realize using existing first-order methods.

4.6 Appendix

4.6.1 Memory Layout

All the matrices are stored in **column-major** order. Input data, typically images, for neural networks is *4-dimensional* in nature i.e., $samples \times channels \times height \times width$. This matrix is stored in memory as a $(samples \times height \times width) \times channels$ matrix. Basically, per channel information is stored in column-major order for each sample point. And multiple samples are stacked vertically on top of each other. For instance, 1^{st} channel information of 1^{st} image is stored in first column in column-major order likewise c^{th} column information is stored in c^{th} column. And if more than one sample points are present, then 2^{nd} images' 1^{st} column is stacked below the 1^{st} images' 1^{st} column and likewise c^{th} channel information of the 2^{nd} image is stacked below the c^{th} channel information of the 1^{st} image in the c^{th} column. Weights associated with each layer of the network can be either a *4-dimensional* matrix, as in the case of a convolution function, or a *2-dimension* matrix, as in the case of a linear (or dense) function. The *4-dimensional* weights matrix of shape $(c^{out} \times c^{in} \times k \times k)$ is treated as a $(c^{in} \times k \times k) \times c^{out}$ matrix with individual $k \times k$ filters stored in column-major format. And at times it is also reordered such that it represents a $(c^{out} \times k \times k) \times c^{in}$ matrix, particularly during the back-propagation algorithm, indicated by $\check{\mathbf{W}}_l$ (weights associated with the l^{th} of the network).

4.6.2 Helper functions

Img2Col Function

A typical *convolution* operation can either be computed by overlaying filter maps (or weights) on top of the input data and computing the output *in a serial fashion* or by converting the input data into a suitable form (typically an enlarged matrix) and performing an efficient matrix multiplication, i.e GEMM operation, with the filter (weights) matrix. Using the former results in redundant transfer of data among

memory layers of the GPU device resulting in inefficient use of GPU device coupled with poor locality of reference (note that the same $k \times k$ filter map may be transferred from CPU to GPU multiple times). And using the latter (because of the enlarged matrix) results in redundant storage of the input matrix. For our work we use the latter approach for performing the convolution operation and we use GEMM operations as defined in the *cublas* library for this purpose. GEMM operations are highly efficient and bandwidth optimized for large matrices.

Img2Col(*) operation converts the input data from shape $m \times c^{in} \times h^{in} \times w^{in}$ to $(m \times h^{out} \times w^{out}) \times (c^{in} \times k \times k)$, where k is the convolution filter map size. Note that the *convolution* function is associated with a set of filter maps of shape $c^{out} \times c^{in} \times k \times k$, where c^{out} is the channels of the output of the convolution operation and c^{in} are the input channels, k is the filter map size, h^{in} , w^{in} and h^{out} , w^{out} are the height and width of the images in each of the associated channels of the input and output data respectively. *Stride*, s , indicates the number of pixels to step ahead along each dimension when applying the filter map whereas *Padding*, p , indicates the number of pixels to be used for padding the input during the convolution operation. h^{in} and h^{out} are tied together with the following equation (same relation can be used to relate w^{in} and w^{out}):

$$h^{out} = \frac{h^{in} + 2p - s}{k} + 1 \quad (4.15)$$

Convolution operation is the process of overlaying filter maps onto the input data and summing the resultant element-wise products. This is repeated over the entire image for all channels by moving the filter map along each dimension (height or width) by *stride* pixels. And prior to the convolution operation per channel information is padded with zeros by *padding* pixels. Note that because of this operation the size of the input data may change and superscripts *in* and *out* are used to indicate the input

and out data to and from the convolution function respectively. Mathematically, we represent convolution operation as follows:

$$\mathbf{C}_l = \mathbf{Img2Col}(\mathbf{A}_{l-1}) \mathbf{W}_l + \mathbf{b}_l \quad (4.16)$$

In eq. 4.16 $\mathbf{Img2Col}(\cdot)$ operator is used to enlarge the input matrix, \mathbf{A}_{l-1} , so that the resultant matrix can be used in a GEMM operation with \mathbf{W}_l matrix for the convolution function. We use $\llbracket \mathbf{A}_{l-1} \rrbracket$ to refer to the enlarged matrix. Bias term, \mathbf{b}_l can be folded into the \mathbf{W}_l by appending it as a row vector. Note that \mathbf{W}_l is treated as a 2-dimensional matrix of shape $(c_l^{in} \times k_l \times k_l) \times c_l^{out}$ and the result of folding the bias vector will result into $\bar{\mathbf{W}}_l$ of shape $(c_l^{in} \times k_l \times k_l + 1) \times c_l^{out}$. We also append a column of ones, \mathbf{e} , to the enlarged matrix, $\llbracket \mathbf{A}_{l-1} \rrbracket$, and the resulting matrix is denoted by $\llbracket \mathbf{A}_{l-1} \rrbracket_H$ (the use of the subscript H , homogenous coordinate, indicates that a column of ones, \mathbf{e} , has been appended). With this change, a concise form of the convolution operation is given by the following equation:

$$\mathbf{C}_l = \llbracket \mathbf{A}_{l-1} \rrbracket_H \bar{\mathbf{W}}_l$$

4.7 Neural Network Operations

4.7.1 Gradient computation

Convolution Function

Convolution function is associated with a set of filters (or weights) \mathbf{W}_l , and bias \mathbf{b}_l , variables which are learned during the minimization of the loss function associated with the neural network. Typically these variables are initialized with suitable values as defined by the various schemes [102,104,105] discussed in current literature. These initialization schemes take into account the mean and the variance of the input and

output data of a convolution function and attempt to maintain them as constants throughout the neural network. Some initialization methods like [102] also take into account the type of non-linearity stacked after the convolution function whereas [104] approximate the non-linearity functions to be constants.

Forward Pass Forward pass through a convolution function is the process of overlaying set of filter maps on top of the input data and summing up the resultant element-wise products, typically known as convolution function. This is repeated over the entire image. This operation can be treated as a GEMM operation by enlarging the input data so that each row (or column) represents a particular filter map and the no. of rows (or columns) indicating the number of times a filter map is applied on the incoming sample point per channel. Note that bias term can be folded into the weights variable by suitably altering the weights and input data matrices. Mathematically, convolution function can be represented as follows:

$$\mathbf{C}_l = \llbracket \mathbf{A}_{l-1} \rrbracket_H \bar{\mathbf{W}}_l$$

Backward Pass Gradient terms associated with the convolution function, $\mathcal{D}\bar{\mathbf{W}}_l$, as well as those terms which are propagated further up through the neural network during back-propagation, \mathbf{G}_l^{conv} , are computed as follows:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} &= \mathbf{G}_l^{aT} \llbracket \mathbf{A}_{l-1} \rrbracket_H \\ \frac{\partial \mathcal{L}}{\partial \mathbf{A}_{l-1}} &= \llbracket \mathbf{G}_l^a \rrbracket \check{\mathbf{W}}_l = \mathbf{G}_l^{conv} \end{aligned}$$

Activation Function

Activation functions, for instance swish, sigmoid, log-softmax etc., are used after the convolution function to transform the input data in a non-linear fashion so as

to differentiate (or classify) among the output's components. After the inputs to the neural network are passed through sufficient non-linearity functions, depending on the design of the network, outputs of the network are used to compute the class probabilities which are used to make predictions of the associated input sample point to the network. Non-linearity functions are employed to aid this classification process.

Note that each sample point to the activation function in the linear layer is a *vector* while in the convolution layer it is a *matrix*. We present below the relationship between inputs and outputs to the non-linearity functions in the context of linear layers, and by replacing the sample points representation from vectors with corresponding notation in matrices similar equations can be derived for activation functions in convolution layers.

Forward Pass In the forward pass, for gradient computation, the non-linear function $\mathcal{F}(\cdot)$ is applied to individual components of the input vector \mathbf{s}_l resulting in the corresponding output components in \mathbf{p}_l .

$$\mathbf{a}_l = \mathcal{F}(\mathbf{s}_l)$$

Backward Pass During the backward pass, the incoming gradient terms (\mathbf{g}_l^p) from the pool function are scaled by the first-order derivative of the activation function, co-ordinate wise, resulting in the gradient terms \mathbf{g}_l^a to be back-propagated to the functions preceding the activation function. Mathematically, this relationship can be formed as:

$$\mathbf{g}_l^a = \mathcal{F}'(\mathbf{s}_l) \odot \mathbf{g}_l^p$$

Pooling Function

Pool functions are used to down-sample the inputs so as to reduce the input size. These functions are associated with *Stride* (s), *Padding* (p) and *kernel size* (k) which are used by the down-sampling procedure when computing the output of the pool function. The height and width of the output of the pool functions are related to its inputs in the same way as defined for the *convolution* function as shown in eq. 4.15.

Forward Pass In the forward pass, the input feature maps are down-sampled channel-wise individually according to the stride, padding and kernel size specifications. For average pool the average of the input feature map is produced as output and for max pool the largest component of the input feature map is spit out as the output of the pool operation for a specific feature map. This process is repeated over the all the channels for each sample point in the input. Following equation captures this behavior:

$$\mathbf{A}_l = \mathcal{P}(\mathbf{P}_l)$$

Backward Pass During the backward pass in the back-propagation algorithm, the gradient terms to be back-propped \mathbf{G}_l^p are produced feature-map-wise. The derivative of the pool function and incoming gradient terms from the next layer in the network, \mathbf{G}_{l+1}^{conv} , are used in element-wise product operation for this purpose as shown in the below equation.

$$\mathbf{G}_l^p = \mathcal{P}'(\mathbf{P}_l) \odot \mathbf{G}_{l+1}^{conv}$$

Linear (Dense) Function

Apart from *convolution* function, *linear or dense* functions are also used to transform the inputs by performing a weighted summation and a bias term to move the center of the input data. Typically in the context of convolution neural networks linear functions are used for classification purposes at the end of the network. This function is associated with \mathbf{W}_l and \mathbf{b}_l parameters whose dimensions are $f_l^{out} \times f_l^{in}$ and $f_l^{out} \times 1$ respectively, where subscript l indicates the layer number and superscripts *in*, *out* indicate the no. of input features and output features associated with the layer l .

Forward Pass Forward pass for the linear function involves a GEMM operation between the weights associated with this function and the incoming data $\bar{\mathbf{a}}_{l-1}$ as shown below:

$$\mathbf{s}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1}^\top$$

Backward Pass During back-propagation the out-going gradient terms, $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{l-1}}$, and gradients w.r.t weights parameters, $\frac{\partial \mathcal{L}}{\partial \bar{\mathbf{W}}_l}$, are computed using GEMM operations as shown below:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \bar{\mathbf{W}}_l} &= \mathbf{g}_l^a \bar{\mathbf{a}}_{l-1}^\top \\ \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{l-1}} &= \mathbf{g}_l^d = \mathbf{W}_l^\top \mathbf{g}_l^a \end{aligned}$$

4.7.2 Hessian Vector Operation on Neural Networks

Similar to gradient computation on the neural network, Hessian-vector computation requires two additional passes over the given network. During the forward pass, we store the required data in the auxiliary variable, $\mathcal{R}_v \{\mathbf{a}_l\}$ (and $\mathcal{R}_v \{\mathbf{A}_l\}$), for each

layer l . And during the backward pass, we use the variable $\mathcal{R}_v \{\mathbf{a}_l\}$ during forward pass as well as \mathbf{a}_l and \mathbf{G}_l stored during the gradient computation on the network to compute the Hessian-vector product w.r.t to a given vector.

Properties of the $\mathcal{R}_v \{.\}$ operator are used extensively for the remainder of this section to produce the intermediate results during the two passes to compute Hessian-vector product. For each function we use the equations used during the gradient computation and apply the $\mathcal{R}_v \{.\}$ operator resulting the intermediate results as well as the Hessian-vector product of the underlying neural network. The vector \mathbf{v} w.r.t which the $\mathcal{R}_v \{.\}$ operator is used in this section is of the same size as $\boldsymbol{\theta}$ and $\mathcal{D}\boldsymbol{\theta}$.

Now we define the forward and backward pass used to compute Hessian-vec computation on the neural network for each function involved.

Linear (Dense) Function

In this section we discuss the application of $\mathcal{R}_v \{.\}$ operator on the equations used during the gradient computation of the linear function.

Forward Pass In the forward pass of the $\mathbf{H}\mathbf{v}$ -computation, we apply the $\mathcal{R}_v \{.\}$ to the equation used in the gradient computation as shown below. Note that $\mathcal{R}_v \{.\}$ operator obeys the multiplication-rules of the derivative calculus. Using this we can easily compute the $\mathcal{R}_v \{\mathbf{s}_l\}$ as shown below. $\mathcal{R}_v \{\bar{\mathbf{W}}_l\}$ is the $\bar{\mathbf{W}}_l$ component in vector \mathbf{v} and $\mathcal{R}_v \{\bar{\mathbf{a}}_l\}$ would have already computed when previous layers were processed.

$$\begin{aligned} \mathcal{R}_v \{\mathbf{s}_l\} &= \mathcal{R}_v \{\bar{\mathbf{W}}_l \bar{\mathbf{a}}_l^\top\} \\ &= \mathcal{R}_v \{\bar{\mathbf{W}}_l\} \bar{\mathbf{a}}_l^\top + \bar{\mathbf{W}}_l \mathcal{R}_v \{\bar{\mathbf{a}}_l\}^\top \end{aligned}$$

Backward Pass Here we apply the $\mathcal{R}_v \{.\}$ operator to the equations used to compute \mathbf{g}_l^d and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l}$ to generate $\mathcal{R}_v \{\mathbf{g}_l^d\}$ and $\mathcal{R}_v \{\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l}\}$. The former term is back-

propagated to the previous layers and the latter, $\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \bar{\mathbf{W}}^l} \right\}$, is the $\mathbf{H}\mathbf{v}$ component for the parameters associated with the linear layer .i.e., $\bar{\mathbf{W}}_l$.

$$\begin{aligned}
 \mathcal{R}_v \{ \mathbf{g}_l^d \} &= \mathcal{R}_v \{ \mathbf{W}_l^\top \mathbf{g}_l^a \} \\
 &= \mathcal{R}_v \{ \mathbf{W}_l^\top \} \mathbf{g}_l^a + \mathbf{W}_l^\top \mathcal{R}_v \{ \mathbf{g}_l^a \} \\
 \mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \bar{\mathbf{W}}^l} \right\} &= \mathcal{R}_v \{ \mathbf{g}_l^a \bar{\mathbf{a}}_{l-1}^\top \} \\
 &= \mathcal{R}_v \{ \mathbf{g}_l^a \} \bar{\mathbf{a}}_{l-1}^\top + \mathbf{g}_l^a \mathcal{R}_v \{ \bar{\mathbf{a}}_{l-1} \}^\top
 \end{aligned}$$

Note that \mathbf{g}_l^a and $\mathcal{R}_v \{ \mathbf{g}_l^a \}$ were already computed during the backward pass of gradient evaluation and current pass respectively, $\bar{\mathbf{W}}_l$ and $\mathcal{R}_v \{ \bar{\mathbf{W}}_l \}$ are the network's parameters and current layers components in \mathbf{v} respectively, and $\bar{\mathbf{a}}_{l-1}$ and $\mathcal{R}_v \{ \bar{\mathbf{a}}_{l-1} \}$ were computed during the forward passes of gradient and $\mathbf{H}\mathbf{v}$ computation respectively.

Convolution Function

In this section we describe the application of R-operator on the equations derived for gradient computation for the convolution operation. Note that compared to gradient computation, which only needs one GEMM operation, computation of convolution function's component of Hessian-vector product requires atleast two GEMM operations.

Forward Pass During the computation of forward pass, which is used to evaluate $\mathcal{R}_v \{ \mathbf{C}_l \}$, the terms $\bar{\mathbf{W}}_l$ and $\mathcal{R}_v \{ \bar{\mathbf{W}}_l \}$ are known values (former is the weights parameter associated with the convolution function and the latter term is the $\bar{\mathbf{W}}_l$ -component in the vector \mathbf{v}). Also note that a column of ones, \mathbf{e} , is added to the expanded matrix $\llbracket \mathcal{R}_v \{ \mathbf{A}_{l-1} \} \rrbracket$ resulting in $\llbracket \mathcal{R}_v \{ \mathbf{A}_{l-1} \} \rrbracket_H$.

$$\begin{aligned}
\mathbf{C}_l &= \llbracket \mathbf{A}_{l-1} \rrbracket_H \bar{\mathbf{W}}_l \\
\mathcal{R}_v \{ \mathbf{C}_l \} &= \llbracket \mathcal{R}_v \{ \mathbf{A}_{l-1} \} \rrbracket_H \bar{\mathbf{W}}_l + \llbracket \mathbf{A}_{l-1} \rrbracket_H \mathcal{R}_v \{ \bar{\mathbf{W}}_l \}
\end{aligned}$$

Backward Pass For a convolution function, we compute $\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} \right\}$ and $\mathcal{R}_v \{ \mathbf{G}_l^{conv} \}$ during the **Hv** backward pass. Evaluation of $\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} \right\}$ is straight forward and bears resemblance to the equations in the forward-pass as shown in the previous paragraph. However, evaluation of $\mathcal{R}_v \{ \mathbf{G}_l^{conv} \}$ requires the expansion of the matrix $\mathcal{R}_v \{ \mathbf{G}_l^a \}$ which is multiplied with the weights matrix. This is because the height and width of the input data, $\mathcal{R}_v \{ \mathbf{G}_l^a \}$ might change compared to output data, $\mathcal{R}_v \{ \mathbf{G}_l^{conv} \}$

$$\begin{aligned}
\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} \right\} &= \mathcal{R}_v \{ [\mathbf{G}_l^a]^\top \llbracket \mathbf{A}_{l-1} \rrbracket_H \} \\
&= \mathcal{R}_v \{ \mathbf{G}_l^a \}^\top \llbracket \mathbf{A}_{l-1} \rrbracket_H + [\mathbf{G}_l^a]^\top \llbracket \mathcal{R}_v \{ \mathbf{A}_{l-1} \} \rrbracket_H \\
\mathcal{R}_v \{ \mathbf{G}_l^{conv} \} &= \mathcal{R}_v \{ \llbracket \mathbf{G}_l^a \rrbracket^\top \check{\mathbf{W}}_l \} \\
&= \llbracket \mathcal{R}_v \{ \mathbf{G}_l^a \} \rrbracket^\top \check{\mathbf{W}}_l + \llbracket \mathbf{G}_l^a \rrbracket^\top \mathcal{R}_v \{ \check{\mathbf{W}}_l \}
\end{aligned}$$

Activation Function

In this paragraph we derive the equations used in computing the activation functions' component in the Hessian-vector product in the context of a convolution layer. Similar equations can be easily derived for linear layer as well by replacing the matrix notation with vector notation and using \mathbf{g}_{l+1}^d instead of \mathbf{G}_l^p as inputs to this function during the backward pass. The equations to compute $\mathcal{R}_v \{ \mathbf{P}_l \}$ and $\mathcal{R}_v \{ \mathbf{G}_l^a \}$ are straight-forward as they are simple applications of the R-operator.

Forward Pass

$$\begin{aligned}\mathcal{R}_v \{ \mathbf{P}_l \} &= \mathcal{R}_v \{ \mathcal{F}(\mathbf{C}_l) \} \\ &= \mathcal{F}'(\mathbf{C}_l) \odot \mathcal{R}_v \{ \mathbf{C}_l \}\end{aligned}$$

Backward Pass

$$\begin{aligned}\mathcal{R}_v \{ \mathbf{G}_l^a \} &= \mathcal{R}_v \{ \mathcal{F}'(\mathbf{C}_l) \odot \mathbf{G}_l^p \} \\ &= \mathcal{F}''(\mathbf{C}_l) \odot \mathcal{R}_v \{ \mathbf{C}_l \} \odot \mathbf{G}_l^p + \mathcal{F}'(\mathbf{C}_l) \odot \mathcal{R}_v \{ \mathbf{G}_l^p \}\end{aligned}$$

Pooling Function

Hv-passes for pool function used to compute $\mathcal{R}_v \{ \mathbf{A}_l \}$ and $\mathcal{R}_v \{ \mathbf{G}_l^p \}$ are shown below:

Forward Pass Evaluation of $\mathcal{R}_v \{ \mathbf{A}_l \}$ involves the application of pool function on the input data $\mathcal{R}_v \{ \mathbf{P}_l \}$.

$$\mathcal{R}_v \{ \mathbf{A}_l \} = \mathcal{P}(\mathcal{R}_v \{ \mathbf{P}_l \})$$

Backward Pass $\mathcal{R}_v \{ \mathbf{G}_l^p \}$ is computed during the backward pass as shown below. (Note that for both types of pool functions supported in our implementation, namely max-pooling and average-pooling, the second derivative of the pool function does not exist thus simplifying the evaluation of $\mathcal{R}_v \{ \mathbf{G}_l^p \}$.)

$$\begin{aligned}\mathcal{R}_v \{ \mathbf{G}_l^p \} &= \mathcal{R}_v \{ \mathcal{P}'(\mathbf{P}_l) \odot \mathbf{G}_{l+1}^{conv} \} \\ &= \mathcal{P}''(\mathbf{P}_l) \odot \mathcal{R}_v \{ \mathbf{P}_l \} \odot \mathbf{G}_{l+1}^{conv} + \mathcal{P}'(\mathbf{P}_l) \odot \mathcal{R}_v \{ \mathbf{G}_{l+1}^{conv} \}\end{aligned}$$

4.7.3 Loss Functions

We use softmax cross-entropy as the loss function, \mathcal{L} , for the scope of this document. Other loss functions can be easily adapted within our framework.

Softmax Cross Entropy Function

Gradient Forward Pass In the forward pass, the output of the network is used to compute the output of the loss function.

$$\mathcal{L} = -\sum_{i=0}^c \mathbf{y}_i \log(\mathbf{q}_i), \text{ where, } \mathbf{q}_i = \frac{e^{[\mathbf{a}_{out}]_i}}{\sum_j e^{[\mathbf{a}_{out}]_j}}$$

Gradient Backward Pass In the backward pass, the error terms, used in the *back-propagation*, are computed as shown below. For the case of a single sample i and j indices indicate the position of the component in the output of the network, \mathbf{a}_{out} .

$$\frac{\partial \mathbf{q}_i}{\partial [\mathbf{a}_{out}]_j} = \begin{cases} \mathbf{q}_i [1 - \mathbf{q}_i] & ; i = j \\ -\mathbf{q}_i \mathbf{q}_j & ; i \neq j \end{cases}$$

The general equation for the error terms propagated through the network is given by the following equation. Indices i , j and k indicate the position in the output vector for each sample point. (Note that here y is assumed to be one-hot encoded².)

²One-hot encoding of a scalar, y , is a vector of all zeros and the y^{th} position is marked with a 1. Usually, \mathbf{e}_y is used to indicate such a vector and subscript y indicates the component of the vectors which contains a 1.

$$\begin{aligned}
\left[\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{out}} \right]_i &= \begin{cases} -\sum_{k=1}^c \mathbf{y}_k [1 - \mathbf{q}_k] & ; \quad i = k \\ \sum_{k=1}^c \mathbf{y}_k \mathbf{q}_k & ; \quad i \neq k \end{cases} \\
&= \mathbf{q}_i - \mathbf{y}_i
\end{aligned}$$

Hessian-vec Backward Pass

$$\begin{aligned}
\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{out}} \right\}_i &= \mathcal{R}_v \{ \mathbf{q}_i \} - \mathcal{R}_v \{ \mathbf{y}_i \} \\
&= \mathcal{R}_v \{ \mathbf{q}_i \}
\end{aligned}$$

$\mathcal{R}_v \{ \mathbf{q}_i \}$ can be computed as follows:

$$\begin{aligned}
\mathbf{q}_i &= \frac{e^{\mathbf{a}_{out}_i}}{\sum_j e^{\mathbf{a}_{out}_j}} \\
\mathcal{R}_v \{ \mathbf{q}_i \} &= \frac{e^{\mathbf{a}_{out}_i}}{\sum_j e^{\mathbf{a}_{out}_j}} \mathcal{R}_v \{ \mathbf{a}_{out} \}_i + \frac{e^{\mathbf{a}_{out}_i}}{\left[\sum_j e^{\mathbf{a}_{out}_j} \right]^2} (-1) \sum_j \left[e^j \mathcal{R}_v \{ \mathbf{a}_{out} \}_j \right] \\
&= \mathbf{q}_i \mathcal{R}_v \{ \mathbf{a}_{out} \}_i - \mathbf{q}_i \sum_j \mathbf{q}_j \mathcal{R}_v \{ \mathbf{a}_{out} \}_j
\end{aligned}$$

Now we can rewrite the equation tused to back-propagate the gradient terms as follows:

$$\mathcal{R}_v \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{out}} \right\}_i = \mathbf{q}_i \mathcal{R}_v \{ \mathbf{a}_{out} \}_i - \mathbf{q}_i \sum_j \mathbf{q}_j \mathcal{R}_v \{ \mathbf{a}_{out} \}_j$$

4.7.4 Activation Functions

Our framework support some of the popular activation functions whose first- and second-derivatives are shown in the following paragraphs. These derivatives can be hooked into the equations derived in the previous sections to compute the gradient and Hessian-vector products of a neural network. Note that each of the activation functions described below takes a scalar, x , as its input.

Sigmoid Function

$$\begin{aligned}\mathcal{F}(x) &= \frac{1}{1 + e^{-x}} \\ \mathcal{F}'(x) &= \mathcal{F}(x) \mathcal{F}(-x) \\ \mathcal{F}''(x) &= \mathcal{F}'(x) \mathcal{F}(-x) - \mathcal{F}(x) \mathcal{F}'(-x)\end{aligned}$$

log-Softmax Function

$$\begin{aligned}\mathcal{F}(x) &= \log(1 + e^x) \\ \mathcal{F}'(x) &= \frac{1}{1 + e^{-x}} \\ \mathcal{F}''(x) &= \left(\frac{1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^x} \right)\end{aligned}$$

Swish Function

$$\begin{aligned}\mathcal{F}(x) &= \frac{x}{1 + e^{-x}} \\ \mathcal{F}'(x) &= \frac{1}{1 + e^{-x}} + \frac{xe^{-x}}{(1 + e^{-x})^2} \\ \mathcal{F}''(x) &= \left[\frac{1}{1 + e^{-x}} \right] \left[\frac{1}{1 + e^x} \right] + \frac{e^{-x}}{(1 + e^{-x})^2} \left[(1 - x) - \frac{2x}{1 + e^{-x}} \right]\end{aligned}$$

4.7.5 Pooling Functions

Average Pool Function

Gradient Forward Pass During the forward pass for the gradient computation, the input indices, (i, j) , and output indices, (i', j') are linked through the eq. 4.15. Average pool function is itself defined by the eq 4.18 shown below. Here we describe how each cell of the output, \mathbf{A}_{l+1} , is computed using the input, \mathbf{P}_l

$$[\mathbf{A}_l]_{i'j'} = \frac{1}{k^2} \sum_{a=i}^{i+k} \left\{ \sum_{b=j}^{j+k} [\mathbf{P}_l]_{ab} \right\}$$

Gradient Backward Pass During the backward pass of the back-propagation algorithm for the Average Pool function the source indices (i', j') and the destination indices (i, j) are related by the eq. 4.15. Also note that the dimensions of the matrices (inputs and outputs) may change because of the down sampling functionality of the pool function. The back-propagation of the gradient terms for the Average Pool function is defined by the eq. 4.18.

$$[\mathbf{G}_l^p]_{i,j} = \frac{1}{k^2} [\mathbf{G}_{l+1}^{conv}]_{i'j'}$$

Hessian-vec Forward Pass In the Hessian-vec forward pass, we apply the R-operator to the equations used in the forward pass in gradient computation of the network; as shown below.

$$\mathcal{R}_v \{ \mathbf{A}_l \}_{i'j'} = \frac{1}{k^2} \sum_{a=i}^{i+k} \left\{ \sum_{b=j}^{j+k} \mathcal{R}_v \{ \mathbf{P}_l \}_{ab} \right\}$$

Hessian-vec Backward Pass

$$\mathcal{R}_v \{ \mathbf{G}_l^p \}_{ij} = \frac{1}{k^2} \mathcal{R}_v \{ \mathbf{G}_{l+1}^{conv} \}_{i'j'}$$

Max Pool Function

For the forward pass of gradient computation, this function computes the maximum value from the input, \mathbf{P}_l , for each filter map as defined by the stride, padding and kernel size parameters. And during the back-propagation, the derivative of the max pool function is computed w.r.t to the input data, \mathbf{P}_l . Hence (i, j) location in the output matrix \mathbf{G}_l^p is set to $[\mathbf{G}_{l+1}^{conv}]_{i'j'}$. Note that the locations (i, j) and (i', j') are related by eq. 4.15. Similar to average pooling, we develop the equations to be used for gradient and Hessian-vector product computation below:

Gradient Forward Pass

$$[\mathbf{A}_l]_{i'j'} = \max_{i,j}^{i+k,j+k} [\mathbf{P}_l]_{ij}$$

Gradient Backward Pass

$$[\mathbf{G}_l^p]_{i'j'} = \begin{cases} [\mathbf{G}_{l+1}^{conv}]_{i'j'} & ; \ i', j' \text{ refer to the location which is } \max_{i,j}^{i+k,j+k} [\mathbf{P}_l]_{ij} \\ 0 & ; \text{ otherwise} \end{cases}$$

In this equation we assume that the $(l+1)^{th}$ is a convolution layer as well. If the following layer is a linear/dense layer then this equation can be easily adopted by appropriately conversion between vectors coming out the dense layer to matrices feeding into the current layer.

Hessian-vec Forward Pass

$$\mathcal{R}_v \{ \mathbf{A}_l \}_{ij} = \max_{i',j'}^{i'+k,j'+k} \mathcal{R}_v \{ \mathbf{P}_l \}_{i'j'}$$

Hessian-vec Backward Pass

$$\mathcal{R}_v \{ \mathbf{G}_l^p \}_{i'j'} = \begin{cases} \mathcal{R}_v \{ \mathbf{G}_{l+1}^{conv} \}_{ij} & ; \quad i', j' \text{ refer to the location which is } \underset{i,j}{\overset{i+k,j+k}{max}} \mathcal{R}_v \{ \mathbf{P}_l \}_{ij} \\ 0 & ; \quad otherwise \end{cases}$$

As described in the case of gradient backward-pass, this above equation can be easily adopted to the case when the $(l+1)^{th}$ layer is a linear/dense layer.

4.7.6 Batch Normalization

Batch Normalization is used to center the incoming data w.r.t to its mean and variance so that the output is centered at 0 with a variance of 1. Because of this activation function, which typically follows a batch normalization function, can operate in meaningful zones (avoiding saturation zones of the activation functions).

The batch normalization function, $\mathbf{B}_l = \mathcal{B}(\mathbf{C}_l)$, takes the input matrix, \mathbf{C}_l , and computes the mean ($\boldsymbol{\mu}_i$) and variance ($\boldsymbol{\sigma}_i^2$) for all channels, c . We assume that batch normalization function is present in convolution layers and is between the convolution function and activation function of each layer for the scope of this section.

Please note that the subscripts c' , i and j for matrices in this paragraph indicate the channel, c' , row i and column j of the said matrix. Also note that Σ indicates the summation of all the components of an image in the mini batch channel-wise (resulting in a vector whose length is the number of channels in the mini batch).

Gradient Forward Pass As described above, for the forward pass during gradient computation the output of the batch normalization function $\tilde{\mathbf{B}}_l$ is computed such that the mean for a given channel, c' , is zero and variance is 1. The following equations captures this process :

$$\begin{aligned}
\left[\tilde{\mathbf{B}}_l\right]_{c'ij} &= \frac{\left[\tilde{\mathbf{C}}_l\right]_{c'ij} - \mu_{c'}}{\sqrt{\sigma_{c'}^2 + \epsilon}}, \text{ and} \\
\mu_{c'} &= \frac{1}{m} \sum_{i,j} \left[\tilde{\mathbf{C}}_l\right]_{c'ij} \\
\sigma_{c'}^2 &= \frac{1}{m} \sum_{i,j} \left[\left[\tilde{\mathbf{C}}_l\right]_{c'ij} - \mu_{c'}\right]^2 \\
\text{where } c' &\in [1 \dots c]
\end{aligned}$$

Gradient Backward Pass By using the chain rule of the derivatives, we can express the gradient terms of the batch normalization function as shown in eq. 4.18.

$$\begin{aligned}
\left[\tilde{\mathbf{G}}_l^{bn}\right]_{c'} &= \left[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{C}}_l}\right]_{c'} = \left[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{B}}_l}\right]_{c'} \left[\frac{\partial \tilde{\mathbf{B}}_l}{\partial \tilde{\mathbf{C}}_l}\right]_{c'} + \frac{\partial \mathcal{L}}{\partial \mu_{c'}} \frac{\partial \mu_{c'}}{\partial \left[\tilde{\mathbf{C}}_l\right]_{c'}} + \frac{\partial \mathcal{L}}{\partial \sigma_{c'}^2} \frac{\partial \sigma_{c'}^2}{\partial \left[\tilde{\mathbf{C}}_l\right]_{c'}} \\
&= \left[\tilde{\mathbf{G}}_l^a\right]_{c'} \left[\frac{\partial \tilde{\mathbf{B}}_l}{\partial \tilde{\mathbf{C}}_l}\right]_{c'} + \frac{\partial \mathcal{L}}{\partial \mu_{c'}} \frac{\partial \mu_{c'}}{\partial \left[\tilde{\mathbf{C}}_l\right]_{c'}} + \frac{\partial \mathcal{L}}{\partial \sigma_{c'}^2} \frac{\partial \sigma_{c'}^2}{\partial \left[\tilde{\mathbf{C}}_l\right]_{c'}}
\end{aligned}$$

All the terms in the above equation can be easily computed as follows:

- Using eq. 4.18 we can easily compute $\left[\frac{\partial \tilde{\mathbf{B}}_l}{\partial \tilde{\mathbf{C}}_l}\right]_{c'}$ as

$$\left[\frac{\partial \tilde{\mathbf{B}}_l}{\partial \tilde{\mathbf{C}}_l}\right]_{c'} = \frac{1}{\sqrt{\sigma_{c'}^2 + \epsilon}}$$

- Using chain-rule from differential calculus, we have

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mu_{c'}} \frac{\partial \mu_{c'}}{\partial [\tilde{\mathbf{C}}_l]_{c'}} &= \left[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{B}}_l} \right]_{c'} \frac{\partial [\tilde{\mathbf{B}}_l]_{c'}}{\partial \mu_{c'}} \frac{\partial \mu_{c'}}{\partial [\tilde{\mathbf{C}}_l]_{c'}} \\
&= [\tilde{\mathbf{G}}_l^a]_{c'} \frac{-1}{\sqrt{\sigma_{c'}^2 + \epsilon}} \left[\frac{1}{m} \right] \\
&= - \frac{[\tilde{\mathbf{G}}_l^a]_{c'}}{m \sqrt{\sigma_{c'}^2 + \epsilon}}
\end{aligned}$$

- For the gradient terms w.r.t the variance, we have the following:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \sigma_{c'}^2} &= \sum \frac{\partial \mathcal{L}}{\partial [\tilde{\mathbf{B}}_l]_{c'}} \frac{\partial [\tilde{\mathbf{B}}_l]_{c'}}{\partial \sigma_{c'}^2} \\
\frac{\partial [\tilde{\mathbf{B}}_l]_{c'}}{\partial \sigma_{c'}^2} &= \left[[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'} \right] \frac{1}{[\sigma_{c'}^2 + \epsilon]^{1.5}} \frac{-1}{2} \\
\frac{\partial \mathcal{L}}{\partial \sigma_{c'}^2} &= - \sum \frac{[\tilde{\mathbf{G}}_l^a]_{c'}}{2} \frac{[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}}{[\sigma_{c'}^2 + \epsilon]^{1.5}}
\end{aligned}$$

Using the above equations, we can write:

$$\frac{\partial \mathcal{L}}{\partial \sigma_{c'}^2} \frac{\partial \sigma_{c'}^2}{\partial [\tilde{\mathbf{C}}_l]_{c'}} = - \frac{1}{m} \left[\sum [\tilde{\mathbf{G}}_l^a]_{c'} \frac{[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}}{[\sigma_{c'}^2 + \epsilon]^{1.5}} \right] \left[\sum ([\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}) \right]$$

Using all the above intermediate results, we conclude with the following:

$$\begin{aligned} \left[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{C}}_l} \right]_{c'} &= [\tilde{\mathbf{G}}_l^{bn}]_{c'} = \frac{[\tilde{\mathbf{G}}_l^a]_{c'}}{\sqrt{\sigma_{c'}^2 + \epsilon}} - \frac{[\tilde{\mathbf{G}}_l^a]_{c'}}{m\sqrt{\sigma_{c'}^2 + \epsilon}} \\ &\quad - \frac{1}{m} \left[\sum [\tilde{\mathbf{G}}_l^a]_{c'} \frac{[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}}{[\sigma_{c'}^2 + \epsilon]^{1.5}} \right] \left[\sum [\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'} \right] \end{aligned}$$

Hessian-vec Forward Pass During **Hv**-product forward pass we compute the term $\mathcal{R}_v \{\mathbf{B}_l\}_{c'}$, for all channels $c' \in [1..c]$ by applying the R-operator to each of the individual equations described in the corresponding gradient computation earlier.

$$\begin{aligned} \mathcal{R}_v \{\tilde{\mathbf{B}}_l\}_{c'} &= \mathcal{R}_v \left\{ \frac{[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}}{\sqrt{\sigma_{c'}^2 + \epsilon}} \right\} \\ &= \mathcal{R}_v \left\{ \frac{1}{\sqrt{\sigma_{c'}^2 + \epsilon}} \right\} \left[[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'} \right] + \frac{1}{\sqrt{\sigma_{c'}^2 + \epsilon}} \mathcal{R}_v \left\{ [\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'} \right\} \\ &= -\frac{[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'}}{m[\sigma_{c'}^2 + \epsilon]^{1.5}} \sum \left[\left[[\tilde{\mathbf{C}}_l]_{c'} - \mu_{c'} \right] \left[\mathcal{R}_v \{\tilde{\mathbf{C}}_l\}_{c'} - \frac{1}{m} \sum [\tilde{\mathbf{C}}_l]_{c'} \right] \right] \\ &\quad + \frac{1}{\sqrt{\sigma_{c'}^2 + \epsilon}} \left[\mathcal{R}_v \{\tilde{\mathbf{C}}_l\}_{c'} - \frac{1}{m} \sum \mathcal{R}_v \{\tilde{\mathbf{C}}_l\}_{c'} \right] \end{aligned}$$

Hessian-vec Backward Pass Finally, during the **Hv**-product backward pass we compute the term $\mathcal{R}_v \{\tilde{\mathbf{G}}_l^{bn}\}_{c'}$, for a given channel c' as shown below.

$$\begin{aligned}
\mathcal{R}_v \{ \mathbf{G}_l^{bn} \}_{c'} &= \mathcal{R}_v \left\{ \frac{1}{m \sqrt{\sigma_{c'}^2 + \epsilon}} \left[m \left[\tilde{\mathbf{G}}_l^a \right]_{c'} - \sum \left[\tilde{\mathbf{G}}_l^a \right]_{c'} - \left[\tilde{\mathbf{B}}_l \right]_{c'} \sum \left[\tilde{\mathbf{G}}_l^a \right]_{c'} \left[\tilde{\mathbf{C}}_l \right]_{c'} \right] \right\} \\
&= \mathcal{R}_v \{ \text{I} \} \text{ II} + \text{I} \mathcal{R}_v \{ \text{II} \} \\
\mathcal{R}_v \{ \text{I} \} &= \frac{-1}{m [\sigma_{c'}^2 + \epsilon]^{1.5}} \sum \left[\left(\left[\tilde{\mathbf{C}}_l \right]_{c'} - \boldsymbol{\mu}_{c'} \right) \left(\mathcal{R}_v \left\{ \tilde{\mathbf{C}}_l - \frac{1}{m} \sum \mathcal{R}_v \{ \tilde{\mathbf{C}}_l \} \right\}_{c'} \right) \right] \\
\mathcal{R}_v \{ \text{II} \} &= m \mathcal{R}_v \left\{ \tilde{\mathbf{G}}_l^a \right\}_{c'} - \sum \mathcal{R}_v \left\{ \tilde{\mathbf{G}}_l^a \right\}_{c'} - \mathcal{R}_v \left\{ \tilde{\mathbf{B}}_l \right\}_{c'} \sum \left(\tilde{\mathbf{G}}_l^a \tilde{\mathbf{B}}_l \right)_{c'} \\
&\quad - \left[\tilde{\mathbf{B}}_l \right]_{c'} \sum \left[\mathcal{R}_v \left\{ \tilde{\mathbf{G}}_l^a \right\}_{c'} \left[\tilde{\mathbf{B}}_l \right]_{c'} + \left[\tilde{\mathbf{G}}_l^a \right]_{c'} \mathcal{R}_v \left\{ \tilde{\mathbf{B}}_l \right\}_{c'} \right]
\end{aligned}$$

REFERENCES

REFERENCES

- [1] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.
- [2] N. I. M. Gould, *An introduction to algorithms for continuous optimization*. Oxford University Computing Laboratory, 2006.
- [3] Y. Nesterov, *Introductory lectures on convex optimization*. Springer Science & Business Media, 2004, vol. 87.
- [4] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
- [5] A. Gittens and M. W. Mahoney, “Revisiting the Nyström method for improved large-scale machine learning,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 3977–4041, 2016.
- [6] P. Xu, F. Roosta-Khorasani, and M. W. Mahoney, “Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study,” *arXiv preprint arXiv:1708.07827*, 2017.
- [7] A. S. Berahas, R. Bollapragada, and J. Nocedal, “An Investigation of Newton-Sketch and Subsampled Newton Methods,” *arXiv preprint arXiv:1705.06211*, 2017.
- [8] F. Roosta-Khorasani and M. W. Mahoney, “Sub-sampled Newton methods I: globally convergent algorithms,” *arXiv preprint arXiv:1601.04737*, 2016.
- [9] —, “Sub-sampled Newton methods II: Local convergence rates,” *arXiv preprint arXiv:1601.04738*, 2016.
- [10] P. Xu, J. Yang, F. Roosta-Khorasani, C. Ré, and M. W. Mahoney, “Sub-sampled newton methods with non-uniform sampling,” in *Advances in Neural Information Processing Systems*, 2016, pp. 3000–3008.
- [11] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics Springer, Berlin, 2001, vol. 1.
- [12] L. Bottou, F. E. Curtis, and J. Nocedal, “Optimization methods for large-scale machine learning,” *arXiv preprint arXiv:1606.04838*, 2016.
- [13] S. Sra, S. Nowozin, and S. J. Wright, *Optimization for machine learning*. Mit Press, 2012.
- [14] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [15] M. W. Mahoney, “Randomized algorithms for matrices and data,” *Foundations and Trends® in Machine Learning*, vol. 3, no. 2, pp. 123–224, 2011.
- [16] H. Avron, P. Maymounkov, and S. Toledo, “Blendenpik: Supercharging LAPACK’s least-squares solver,” *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1217–1236, 2010.

- [17] X. Meng, M. A. Saunders, and M. W. Mahoney, “LSRN: A parallel iterative solver for strongly over-or underdetermined systems,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C95–C118, 2014.
- [18] J. Yang, X. Meng, and M. W. Mahoney, “Implementing randomized matrix algorithms in parallel and distributed environments,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 58–92, 2016.
- [19] P. Xu, F. Roosta, and M. W. Mahoney, “Newton-type methods for non-convex optimization under inexact hessian information,” *Mathematical Programming*, pp. 1–36, doi: 10.1007/s10107-019-01405-z.
- [20] R. Bollapragada, R. Byrd, and J. Nocedal, “Exact and inexact subsampled Newton methods for optimization,” *arXiv preprint arXiv:1609.08502*, 2016.
- [21] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, “Sample size selection in optimization methods for machine learning,” *Mathematical programming*, vol. 134, no. 1, pp. 127–155, 2012.
- [22] M. A. Erdogdu and A. Montanari, “Convergence rates of sub-sampled newton methods,” in *Advances in Neural Information Processing Systems 28*, 2015, pp. 3034–3042.
- [23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning.” in *OSDI*, vol. 16, 2016, pp. 265–283.
- [24] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for on-line learning and stochastic optimization,” *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.
- [25] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, vol. 4, 2012.
- [26] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [27] M. D. Zeiler, “Adadelta: an adaptive learning rate method,” *arXiv preprint arXiv:1212.5701*, 2012.
- [28] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *International conference on machine learning*, 2013, pp. 1139–1147.
- [29] F. Roosta-Khorasani, K. van den Doel, and U. Ascher, “Stochastic algorithms for inverse problems involving PDEs and many measurements,” *SIAM J. Scientific Computing*, vol. 36, no. 5, pp. S3–S22, 2014.
- [30] —, “Data completion and stochastic algorithms for PDE inversion problems with many measurements,” *Electronic Transactions on Numerical Analysis*, vol. 42, pp. 177–196, 2014.
- [31] K. v. d. Doel and U. Ascher, “Adaptive and stochastic algorithms for EIT and DC resistivity problems with piecewise constant solutions and many measurements,” *SIAM J. Scient. Comput.*, vol. 34, p. DOI: 10.1137/110826692, 2012.
- [32] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng, “Scalable learning for object detection with gpu hardware,” in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. IEEE, 2009, pp. 4287–4293.

- [33] R. Raina, A. Madhavan, and A. Y. Ng, “Large-scale deep unsupervised learning using graphics processors,” in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 873–880.
- [34] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep learning with cots hpc systems,” in *International Conference on Machine Learning*, 2013, pp. 1337–1345.
- [35] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, “On optimization methods for deep learning,” in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 265–272.
- [36] K. P. Murphy, *Machine learning: a probabilistic perspective*. The MIT Press, 2012.
- [37] S. B. Kylasa, H. M. Aktulga, and A. Y. Grama, “Puremd-gpu: A reactive molecular dynamics simulation package for gpus,” *Journal of Computational Physics*, vol. 272, pp. 343–359, September 2014.
- [38] S. B. Kylasa, “Newton-cg cuda implementation download (scripts/code/tensorflow-python-scripts),” <https://github.com/kylasa/NewtonCG>, February 2018.
- [39] UCI, “Uci machine learning repository,” <http://archive.ics.uci.edu/ml/index.php>, 02 2018.
- [40] L. Bottou and Y. LeCun, “Large scale online learning,” *Advances in neural information processing systems*, vol. 16, p. 217, 2004.
- [41] S. B. Kylasa, F. Roosta-Khorasani, M. W. Mahoney, and A. Grama, “GPU Accelerated Sub-Sampled Newton’s Method,” in *SIAM International Conference on Data Mining (SDM)*, 2019, accepted.
- [42] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, “Revisiting distributed synchronous sgd,” *arXiv preprint arXiv:1604.00981*, 2016.
- [43] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch sgd: training imagenet in 1 hour,” *arXiv preprint arXiv:1706.02677*, 2017.
- [44] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer, “How to scale distributed deep learning?” *arXiv preprint arXiv:1611.04581*, 2016.
- [45] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, “Large scale distributed deep networks,” in *Advances in neural information processing systems*, 2012, pp. 1223–1231.
- [46] S. Wang, F. Roosta-Khorasani, P. Xu, and M. W. Mahoney, “GIANT: Globally Improved Approximate Newton Method for Distributed Optimization,” in *Advances in Neural Information Processing Systems (NIPS)*, 2018, pp. 2338–2348.
- [47] H. Daneshmand, A. Lucchi, and T. Hofmann, “DynaNewton-Accelerating Newton’s Method for Machine Learning,” *arXiv preprint arXiv:1605.06561*, 2016.
- [48] S. J. Reddi, J. Konečný, P. Richtárik, B. Póczós, and A. Smola, “AIDE: Fast and communication efficient distributed optimization,” *arXiv preprint arXiv:1608.06879*, 2016.
- [49] Y. Zhang and X. Lin, “Disco: Distributed optimization for self-concordant empirical loss,” in *International conference on machine learning*, 2015, pp. 362–370.

- [50] Z. Xu, G. Taylor, H. Li, M. Figueiredo, X. Yuan, and T. Goldstein, "Adaptive consensus admm for distributed optimization," *arXiv preprint arXiv:1706.02869*, 2017.
- [51] Z. Xu, M. A. Figueiredo, X. Yuan, C. Studer, and T. Goldstein, "Adaptive relaxed admm: Convergence theory and practical implementation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 7234–7243.
- [52] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.
- [53] A. Beck, *First-Order Methods in Optimization*. SIAM, 2017, vol. 25.
- [54] S. Bubeck *et al.*, "Convex optimization: Algorithms and complexity," *Foundations and Trends® in Machine Learning*, vol. 8, no. 3-4, pp. 231–357, 2015.
- [55] R. Crane and F. Roosta, "Dingo: Distributed newton-type method for gradient-norm optimization," in *Advances in Neural Information Processing Systems*, 2019.
- [56] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in Neural Information Processing Systems*, 2013, pp. 315–323.
- [57] C. Dünnér, A. Lucchi, M. Gargiani, A. Bian, T. Hofmann, and M. Jaggi, "A distributed second-order algorithm you can trust," *arXiv preprint arXiv:1806.07569*, 2018.
- [58] B. Craven, "Invex functions and constrained local minima," *Bulletin of the Australian Mathematical society*, vol. 24, no. 03, pp. 357–366, 1981.
- [59] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.
- [60] "Tiny imagenet dataset," <https://tiny-imagenet.herokuapp.com/>, October 2019. [Online]. Available: <https://tiny-imagenet.herokuapp.com/>
- [61] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.
- [62] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2015.
- [63] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Muller, "Efficient backprop," *Neural Networks: tricks of the trade*, 1998.
- [64] G. Montavon, G. B. Orr, and K.-R. Muller, *Neural Networks: Tricks of the Trade*, 2nd ed. Springer, September 2012.
- [65] D. Wu, M. Nekovee, and Y. Wang, "Deep learning based autoencoder for interference channel," *arXiv preprint arXiv:1902.06841*, 2019.
- [66] T. Dumas, A. Roumy, and C. Guillemot, "Autoencoder based image compression: Can the learning be quantization independent?" *arXiv preprint arXiv:1802.0937*, 2018.
- [67] Y. Rao, J. Lu, and J. Zhou, "Attention-aware deep reinforcement learning for video face recognition," *The IEEE International Conference on Computer Vision (ICCV)*, pp. 3931–3940, 2017.

- [68] Z. Lu, L. Li, J. Gao, X. He, J. Chen, L. Deng, and J. He, “Recurrent reinforcement learning: A hybrid approach,” *arXiv preprint arXiv:1509.03044*, 2015.
- [69] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization,” *arXiv:1406.2572v1*, 2014.
- [70] A. R. Conn, N. I. Gould, and P. L. Toint, *Trust region methods*. SIAM, 2000, vol. 1.
- [71] C. Cartis, N. I. Gould, and P. L. Toint, “On the complexity of steepest descent, Newton’s and regularized Newton’s methods for nonconvex unconstrained optimization problems,” *Siam journal on optimization*, vol. 20, no. 6, pp. 2833–2852, 2010.
- [72] —, “Adaptive cubic regularisation methods for unconstrained optimization. Part I: motivation, convergence and numerical results,” *Mathematical Programming*, vol. 127, no. 2, pp. 245–295, 2011.
- [73] —, “Adaptive cubic regularisation methods for unconstrained optimization. Part II: worst-case function-and derivative-evaluation complexity,” *Mathematical programming*, vol. 130, no. 2, pp. 295–319, 2011.
- [74] F. E. Curtis, D. P. Robinson, and M. Samadi, “A Trust Region Algorithm with a Worst-Case Iteration Complexity of $\mathcal{O}(\epsilon^{-3/2})$ for Nonconvex Optimization,” *Mathematical Programming*, vol. 162, no. 1-2, pp. 1–32, 2017.
- [75] C. W. Royer, M. O’Neill, and S. J. Wright, “A newton-cg algorithm with complexity guarantees for smooth unconstrained optimization,” *Mathematical Programming*, pp. 1–38, 2019.
- [76] J. Martens, “Deep learning via Hessian-free optimization,” in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 735–742.
- [77] Z. Yao, P. Xu, F. Roosta-Khorasani, and M. W. Mahoney, “Inexact non-convex Newton-type methods,” *arXiv preprint arXiv:1802.06925*, 2018.
- [78] S. ichi Amari, “Natural gradient works efficiently in learning,” *Neural Computation*, vol. 10, no. 251-276, 1988.
- [79] H. H. Yang and S. ichi Amari, “The efficiency and the robustness of natural gradient descent learning rule,” in *Neural Information Processing Systems*, 1997, pp. 385–391.
- [80] S. ichi Amari, R. Karakida, and M. Oyizumi, “Fisher information and natural gradient learning of random deep networks,” *arXiv preprint: 1808.07172v1*, 2018.
- [81] J. Martens and R. Grosse, “Optimizing neural networks with kronecker-factored approximate curvature,” *arXiv:1503.05671v6*, 2016.
- [82] R. Grosse and J. Martens, “A kronecker-factored approximate fisher matrix for convolution layers,” *arXiv:1602.01407v2*, 2016.
- [83] J. Nocedal, “Updating quasi-Newton matrices with limited storage,” *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.
- [84] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010*. Springer, 2010, pp. 177–186.
- [85] Y. Dauphin, H. de Vries, and Y. Bengio, “Equilibrated adaptive learning rates for non-convex optimization,” in *Advances in Neural Information Processing Systems*, 2015, pp. 1504–1512.

- [86] C. Jin, R. Ge, P. Netrapalli, S. M. Kakade, and M. I. Jordan, “How to escape saddle points efficiently,” *arXiv preprint arXiv:1703.00887*, 2017.
- [87] Z. Allen-Zhu and Y. Li, “Neon2: Finding local minima via first-order oracles,” in *Advances in Neural Information Processing Systems*, 2018, pp. 3720–3730.
- [88] J. Ba, R. Grosse, and J. Martens, “Distributed second-order optimization using kronecker-factored approximations,” *ICLR*, 2017.
- [89] J. Martens, “New insights and perspectives on the natural gradient method,” *arXiv preprint: arXiv:1412.1193v9*, 2017.
- [90] D. C. Liu and J. Nocedal, “On the limited memory BFGS method for large scale optimization,” *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.
- [91] S. B. Kylasa, F. R. Khorasani, M. W. Mahoney, and A. Y. Grama, “Gpu accelerated sub-sampled newton’s method for convex classification problems,” in *Proceedings of the 2019 SIAM International Conference on Data Mining*, SIAM, Ed. SIAM, 2019, pp. 702–710.
- [92] Y. Yu, D. Zou, and Q. Gu, “Saving gradient and negative curvature computations: Finding local minima more efficiently,” *arXiv:1712.03950v1*, 2017.
- [93] Y. Xu, J. Rong, and T. Yang, “First-order stochastic algorithms for escaping from saddle points in almost linear time.” in *Advances in Neural Information Processing Systems*, 2018.
- [94] Y. Yu, P. Xu, and Q. Gu, “Third-order smoothness helps: Even faster stochastic optimization algorithms for finding local minima,” *arXiv:1712.06585v1*, 2017.
- [95] O. Chapelle and D. Erhan, “Improved preconditioner for hessian free optimization,” in *In NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.
- [96] W. R. Morrow, “Hessian-free methods for checking the second-order sufficient conditions in equality-constrained optimization and equilibrium problems,” *arXiv preprint arXiv:1106.0898*, 2011.
- [97] H. Zhang, C. Xiong, J. Bradbury, and R. Socher, “Block-diagonal hessian-free optimization for training neural networks,” *arXiv preprint: arXiv:1712.07296v1*, 2017.
- [98] G. Hinton, “Neural networks for machine learning,” *Coursera, video lectures. 307*, 2012.
- [99] B. A. Pearlmutter, “Fast exact multiplication by the hessian,” *Neural Computation*, 1993.
- [100] S. B. Kylasa, “Cuda non-convex framework for fitter optimizer,” <https://github.com/kylasa/NewtonCG>, 10 2019. [Online]. Available: <https://github.com/kylasa/NewtonCG>
- [101] “Cifar datasets,” <https://www.cs.toronto.edu/~kriz/cifar.html>, 10 2019. [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>
- [102] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *arXiv preprint arXiv:1502.01852*, 2015.
- [103] C.-H. Fang, S. B. Kylasa, F. Roosta-Khorasani, M. W. Mahoney, and A. Grama, “Distributed second-order convex optimization,” *arXiv preprint arXiv:1807.07132*, 2018.

- [104] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feed-forward neural networks,” *AISTATS, JMLR*, 2010.
- [105] pyTorch, “Default initialization pytorch,” https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html. [Online]. Available: https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html