GPU Accelerated Sub-Sampled Newton's Method for Convex Classification Problems

Sudhir Kylasa* Fred (Farbod) Roosta † Michael W. Mahoney ‡ Ananth Grama §

Abstract

First order optimization methods, which rely only on gradient information, are commonly used in diverse machine learning (ML) applications, owing to their simplicity of implementations and 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 come at the expense of higher per-iteration costs, which in "big data" regimes, can be computationally prohibitive.

In this paper, we show that, contrary to conventional belief, second-order methods, when designed suitably, can be much more efficient than first-order alternatives for large-scale ML applications. In convex settings, we show that variants of classical Newton's method in which the Hessian and/or gradient are randomly subsampled, coupled with efficient GPU implementations, far outperform state of the art implementations of existing techniques in popular ML software packages such as TensorFlow. We show that our proposed methods (i) achieve better generalization errors in significantly lower wall-clock time – orders of magnitude faster, compared to first-order alternatives (in TensorFlow) and, (ii) offers significantly smaller (and easily parameterized) hyper-parameter space making our methods highly robust.

1 Introduction

Optimization techniques are at the core of many ML 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, and *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 even for moderately ill-conditioned problems. Contrary to popular belief, ill-conditioned problems often arise in ML applications. For example, the "vanishing and exploding gradient problem" encountered in training deep neural nets [2], 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. 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 [3].

Compared with first-order alternatives, second-order methods use additional curvature information in the form of the Hessian matrix. As a result of incorporating such information, in addition to faster convergence rates, second-order methods offer a variety of, rather more subtle, benefits. For example, unlike first-order methods, Newton-type methods have been shown to be highly resilient to increasing problem ill-conditioning [16, 17]. Furthermore, second-order methods typically require fewer parameters (e.g., inexactness tolerance for the sub-problem solver and line-search parameters), and are less sensitive to their specific settings [3].By using curvature information at each iteration, these methods scale the gradient so that it is a more suitable direction to follow. Consequently, they typically require much fewer iterations, as compared to first-order counterparts.

However, these benefits come at a cost: each iteration of second-order methods may be more expensive than those of the first-order alternatives. Arguably, due to this reason alone, second order methods have not received the attention from the ML community that they deserve. In this paper, we show that by reducing the cost of each iteration through efficient

*Purdue University, skylasa@purdue.edu
†University of Queensland, fred.roosta@uq.edu.au
‡University of California, mmahoney@stat.berkeley.edu
§Purdue University, ayg@cs.purdue.edu

approximation of curvature, coupled with hardware specific acceleration, one can obtain methods that are much *faster* and more *robust* than state of the art techniques. *In most ML applications, this typically translates to achieving a high test-accuracy early on in the iterative process and without significant parameter tuning.* This is in sharp contrast with the slow-ramping trend, typically observed in training with first-order methods, which is often preceded by a lengthy trial and error procedure for parameter tuning. We demonstrate that the desirable numerical/ statistical properties, coupled with algorithmic innovations and hardware-specific implementations, hold the promise for significantly changing the landscape of optimization techniques for machine learning.

We focus on the commonly encountered finite-sum optimization problem:

$$\min_{\mathbf{x}\in\mathbb{R}^d} F(\mathbf{x}) \triangleq \sum_{i=1}^{n} f_i(\mathbf{x}), \tag{1.1}$$

where each $f_i(\mathbf{x})$ is a smooth convex function, representing a loss (or misfit) corresponding to the i^{th} observation (or measurement) [10, 5, 19]. In many ML applications, F in eq. (1.1) corresponds to the *empirical risk* [18], and the goal of solving eq. (1.1) is to obtain a solution with small generalization error, i.e., high predictive accuracy on "unseen" data. We consider eq. (1.1) at scale, where the values of n and d are large. 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 computational bottleneck. In such cases, randomized sub-sampling has been shown to be highly successful in reducing computational and memory costs of a variety of second-order methods to be effectively *independent* of n. Indeed, the theoretical properties of sub-sampled Newton-type methods, for both convex and non-convex problems of the form in eq. (1.1), have been recently studied, e.g., [16, 17, 4, 7, 9]. *However, efficient algorithms, coupled with practical and hardware-specific implementations that can effectively draw upon all available computing resources, are lacking.*

Contributions: Our contributions 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. We show that, for solving eq.* (1.1)*, our randomized method significantly outperforms state of the art implementations of existing techniques in popular ML software packages such as TensorFlow [1], in terms of improved training time, generalization error, and robustness to various adversarial effects. In particular we show that our methods achieve significantly better generalization errors orders of magnitude faster compared to state-of-the-art first-order and quasi-Newton alternatives on real-world datasets.*

1.1 Related Work The class of first-order methods includes a number of techniques that are commonly used in diverse ML applications. Many of these techniques have been efficiently implemented in popular software packages. For example, TensorFlowhas enjoyed considerable success among ML practitioners. Among first-order methods implemented in TensorFlow for solving (1.1) are Adagrad, RMSProp, Adam, Adadelta, and SGD with/without momentum.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. (1.1), have been recently studied in a series of results, in the context of ML applications [16, 17, 4, 7, 9].

GPUs have been successfully used in a variety of ML applications to speed up computations [8, 15, 13]. In particular, Raina et al. [15] 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 paper, Ngiam et al. [13] 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. Deriving similar results for full-fledged second-order methods, is a major goal of our effort.

2 Sub-Sampled Newton's Method: A Review

We first describe a sub-sampled variant of the classical Newton's method, which provides the basis for our

implementation of second-order methods. In this discussion, vectors, $\mathbf{v}$, and matrices, $\mathbf{V}$, are denoted by bold lower and upper case letters, respectively. $\nabla f(\mathbf{x})$ and $\nabla^2 f(\mathbf{x})$ represent the gradient and the Hessian of f at $\mathbf{x}$, respectively. The superscript, e.g., $\mathbf{x}^{(k)}$, denotes iteration count. $\mathcal{S}$ denotes a collection of indices drawn from the set $\{1, 2, \cdots, n\}$, with potentially repeated items, and its cardinality is denoted by $|\mathcal{S}|$.

For the optimization problem eq. (1.1), in each iteration, consider selecting two sample sets of indices from $\{1, 2, \ldots, n\}$, 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:

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

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

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 [16, 17]. 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

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

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

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

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

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

for some $\beta \in (0, 1)$. The requirement in eq. (2.3c) is often referred to as Armijo-type line-search [14], and eq. (2.3b) is the θ-relative error approximation condition of the exact solution to the linear system

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

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. (2.4) satisfying eq. (2.3b). It has also been shown [16, 17], that to inherit the convergence properties of the, rather expensive, algorithm that employs the exact solution to eq. (2.4), the inexactness tolerance, θ, in eq. (2.3b) need 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 4). Putting all of these together, we obtain Algorithm 1, which under specific assumptions, has been shown [16, 17] to be globally linearly convergent[1] with problem-independent local convergence rate [2].

Algorithm 1 Sub-Sampled Newton Method

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

Computation Cost: First-order methods only compute gradient, per mini-batch, several times in each epoch. Compared to first-order methods, our proposed methods requires several matrix-vector products during the linear solve, $\mathbf{Hp} = -\mathbf{g}$. Note that for convex problems, $\mathbf{H}$ is positive semi-definite and, in our experiments, we only use a maximum of 10 CG iterations for estimating the newton-direction. Armijo-type line search, which is used to estimate the step-size, α, requires function evaluation several times (in our experiments we limit this to a maximum of 20). Quasi-Newton methods approximate Hessian of the problem, $\mathbf{H}$, either using rank-one updates (SR1) or rank-two updates (BFGS). However, for these methods, the operation of multiplying the *inverse of approximated Hessian* $\times$ *vector* is clearly defined, and requires at least two matrix-vector products, two vector outer-products, and a matrix summation for such computation. These methods also employ line search methods, similar to our proposed methods, to estimate step-size, α.

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

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

3 Algorithms and Implementation Details

We present the algorithmic machinery involved in implementation of iterations described in eq. (2.3), applied to the function defined in (3.5), with an added ℓ_2 regularization term: $F(\mathbf{x}) + \lambda\|\mathbf{x}\|^2/2$. Here, λ is the regularization parameter. Detailed discussion of CG and line search algorithms used in our methods can be found in [11]. Implementation details, pseudo-code, and GPU optimizations used by our methods are discussed in-detail in the supplementary material Section 1.

3.1 Multi-Class classification We briefly review multi-class classification using softmax and cross-entropy loss function, as an important instance of finite sum minimization problems. Consider a p dimensional feature vector $\mathbf{a}$, with corresponding labels b, drawn from one of C classes. The probability that $\mathbf{a}$ belongs to a class $c \in \{1, 2, \ldots, C\}$ is given by $\mathbf{Pr}\left(b = c \mid \mathbf{a}, \mathbf{w}_1, \ldots, \mathbf{w}_C\right) = e^{\langle \mathbf{a}, \mathbf{w}_c \rangle} / \sum_{c'=1}^{C} e^{\langle \mathbf{a}, \mathbf{w}_{c'} \rangle}$, 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 $\mathbf{x}_c \triangleq \mathbf{w}_c - \mathbf{w}_C,\ c = 1, 2, \ldots, C-1$, for training data $\{\mathbf{a}_i, b_i\}_{i=1}^n \subset \mathbb{R}^p \times \{1, \ldots, C\}$, the cross-entropy loss function for $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_{C-1}] \in \mathbb{R}^{(C-1)p}$ can be written as

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

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

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

Similarly, for the full Hessian of F, we have

$$
(3.7a) \qquad \nabla^2_{\mathbf{x}_c, \mathbf{x}_c} F = \sum_{i=1}^{n} \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{a}_i^T,
$$

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

$$
(3.7b) \qquad \nabla^2_{\mathbf{x}_c, \mathbf{x}_{\hat{c}}} F = \sum_{i=1}^{n} \left(- \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_{\hat{c}} + \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{x}_i^T.
$$

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

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

3.1.1 Numerical Stability To avoid over-flow in the evaluation of exponential functions in (3.5), we use the "Log-Sum-Exp" trick [12]. Specifically, for each data point $\mathbf{a}_i$, we first find the maximum value among $\langle \mathbf{a}_i, \mathbf{x}_c \rangle,\ c = 1, \ldots, C-1$. Define

$$
(3.9) \qquad M(\mathbf{a}) = \max\left\{0, \langle \mathbf{a}, \mathbf{x}_1 \rangle, \langle \mathbf{a}, \mathbf{x}_2 \rangle, \ldots, \langle \mathbf{a}, \mathbf{x}_{C-1} \rangle\right\}, \text{and}
$$

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

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 (3.5), we use $\log\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right) = M(\mathbf{a}_i) + \log\left(\alpha(\mathbf{a}_i)\right)$. Similarly, for (3.6) and (3.7), 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.

3.1.2 Hessian Vector Product Given a vector $\mathbf{v} \in \mathbb{R}^d$, we can compute the Hessian-vector product without explicitly forming the Hessian. 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. (3.9) and (3.10), respectively. Now using matrices

(3.11)
$$\mathbf{V} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{v}_1 \rangle & \langle \mathbf{a}_1, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_1, \mathbf{v}_{C-1} \rangle \\ \langle \mathbf{a}_2, \mathbf{v}_1 \rangle & \langle \mathbf{a}_2, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_2, \mathbf{v}_{C-1} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{v}_1 \rangle & \langle \mathbf{a}_n, \mathbf{v}_2 \rangle & \dots & \langle \mathbf{a}_n, \mathbf{v}_{(C-1)} \rangle \end{bmatrix}_{n \times (C-1)},$$

and

(3.12)
$$\mathbf{W} = \begin{bmatrix} h(\mathbf{a}_1, \mathbf{x}_1) & h(\mathbf{a}_1, \mathbf{x}_2) & \dots & h(\mathbf{a}_1, \mathbf{x}_{C-1}) \\ h(\mathbf{a}_2, \mathbf{x}_1) & h(\mathbf{a}_2, \mathbf{x}_2) & \dots & h(\mathbf{a}_2, \mathbf{x}_{C-1}) \\ \vdots & \vdots & \ddots & \vdots \\ h(\mathbf{a}_n, \mathbf{x}_1) & h(\mathbf{a}_n, \mathbf{x}_2) & \dots & h(\mathbf{a}_n, \mathbf{x}_{C-1}) \end{bmatrix}_{n \times (C-1)}$$

we compute

$$\mathbf{U} = \mathbf{V} \odot \mathbf{W} - \mathbf{W} \odot \Big(\big((\mathbf{V} \odot \mathbf{W})\, \mathbf{e} \big) \mathbf{e}^T \Big), \tag{3.13}$$

to get

$$\mathbf{H}\mathbf{v} = \mathrm{vec}\left(\mathbf{A}^T \mathbf{U} \right), \tag{3.14}$$

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, $\mathbf{A}^T = \big[\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\big]$.

Remark: *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.*

4 Experimental Results

We compare our methods to state of the art methods – SGD with momentum (henceforth referred to as Momentum) Adagrad Adadelta Adam and RMSProp as implemented in TensorFlow.We also evaluate our methods in the context of quasi-newton methods, such as *NonlinearCG*, *BFGS* and *SR1*. In the interest of space we direct readers to sections 5.2, 6.1 and 6.2 in [14] for detailed discussion of these competing methods. The code developed in this work along with the processed datasets are publicly available [3]. Additionally, raw datasets are also available from the UCI Machine Learning Repository[4].

[3] https://github.com/kylasa/NewtonCG

[4] http://archive.ics.uci.edu/ml/index.php

4.1 Experimental Setup and Data Newton-type methods are implemented in C/C++ using CUDA/8.0 toolkit. 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 the *Softmax Log-likelihood* function. and λ is the regularization parameter. Table 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 [16], 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 $\mathbf{A} \in \mathbb{R}^{n \times p}$ 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 1.

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 lead to better performance [6]), 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 $|\mathcal{S}_{\mathbf{g}}| = 0.2n$.

We present results for two implementations of second-order methods: (a) *FullNewton*, the classical Newton-CG algorithm [14], which uses the exact gradient and Hessian, and (b)*SubsampledNewton-20*, $|\mathcal{S}_{\mathbf{g}}| = 0.2n$, and *SubsampledNewton-100*,$|\mathcal{S}_{\mathbf{g}}| = n$, are compared against first-order methods using batch sizes 128 and 20% respectively. These methods use $|\mathcal{S}_{\mathbf{H}}| = 0.05n$. CG-tolerance is set to 10^{-4}. Maximum number of CG iterations is 10 for all 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.

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). Newton-type methods always use 1-GPU and 1-CPU-

Table 1: Description of the datasets.

Classification	Dataset	Train Size (n)	Test Size	p	C	L
Multi-Class	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

core for computations. Please see [11] for detailed discussion on TensorFlow's performance on various hardware platforms.

4.4 Performance Comparisons with First-order methods Table 2 presents 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". Detailed discussion on additional datasets can be found in [11].

4.4.1 Drive Diagnostics Dataset Row 1 of Table. 2 shows the results for the *Drive Diagnostics* dataset. We notice that all Newton-type methods achieve lower objective function in the initial few iterations compared to first order counterparts. When the batch size is larger we notice first-order methods take longer time to achieve the same objective function value compared to smaller batch sized counterparts. Note that sub-sampled methods yields similar results (objective function value and generalization error) compared to Full-Newton method.

4.4.2 MNIST and CIFAR-10 Datasets Rows 2 and 3 in Table. 2 present plots for *MNIST* and CIFAR-10 datasets, respectively. Regardless of the batch size, Newton-type methods clearly outperform first order methods for these two datasets. When larger batch size is used for first order methods we notice that these methods take more epochs compared to their smaller batch sized counterparts in reaching same objective function value and generalization error. This behavior is more prominent in *CIFAR-10* dataset, which represents a relatively *ill*-conditioned problem. As a result, in terms of lowering the objective function on *CIFAR-10*, first-order methods are negatively impacted by problem ill-conditioning, whereas all Newton-type methods show excellent robustness. (Note that, for *CIFAR-10*, our proposed methods are $\approx 700\times$ faster than first-order alternatives irrespective of the mini-batch size.)

4.4.3 Newsgroups20 Dataset Plots in row 4 of Table. 2 represent *Newsgroups20* dataset, which is a sparse dataset, and the Hessian size is $\approx$ 1e6 $\times$ 1e6. We clearly notice *SubsampledNewton-100* yields superior training accuracy compared to all methods (column 1). However, *SubsampledNewton-20* takes more epochs to achieve the same objective function value as its full-gradient counterpart, as seen in column 4. This can be attributed to a smaller gradient sample size, and sparse nature of this dataset.

4.5 Sensitivity to Hyper-Parameter Tuning A major consideration for first-order methods is that of fine-tuning of various underlying hyper-parameters, most notably, the step-size [3].Indeed, the success of most such methods is strongly determined by many trial and error steps to find proper parameter settings. In contrast, second-order optimization methods involve much less parameter tuning, and are less sensitive to specific choices of their hyper-parameters [3].

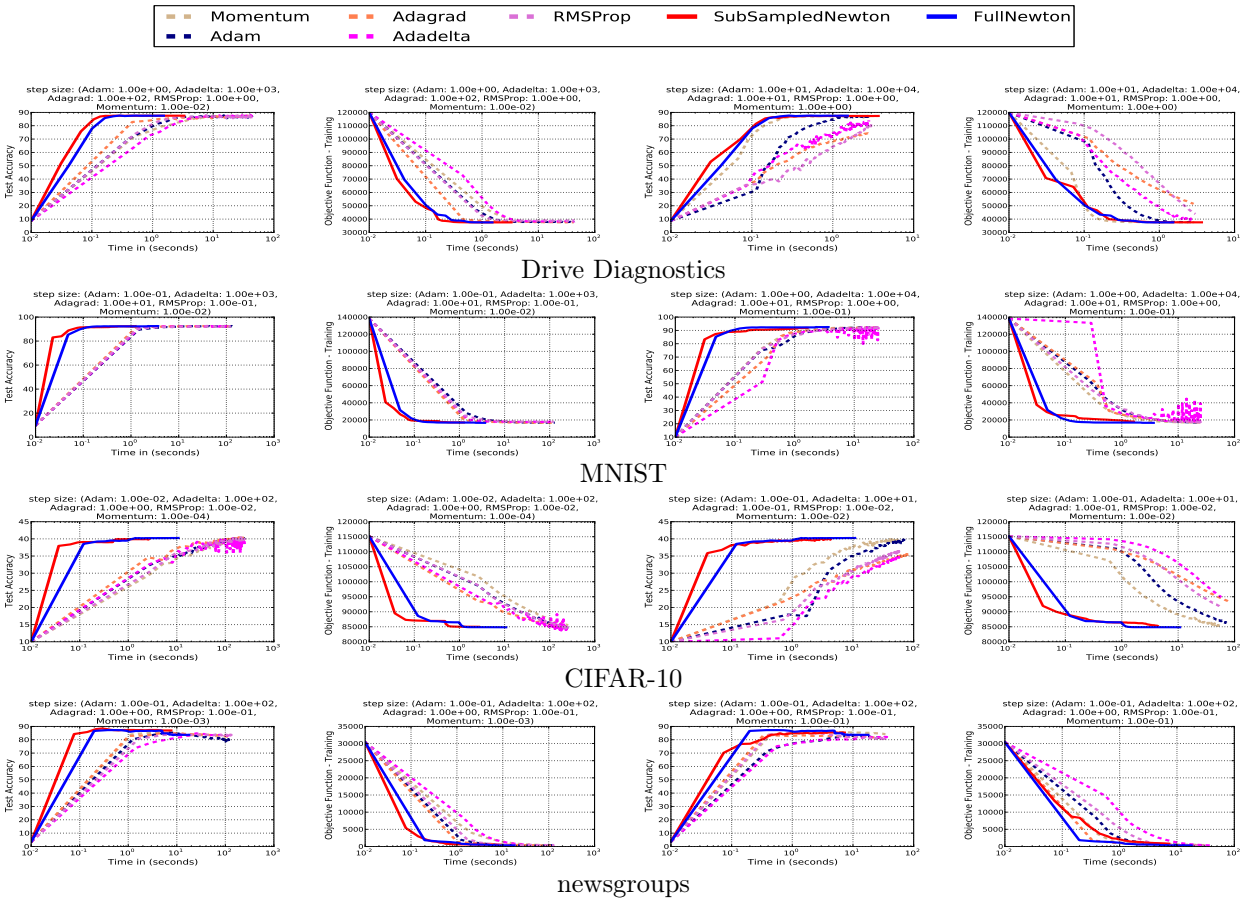
To further highlight these issues, we demonstrate the sensitivity of several first-order methods with respect to their learning rate. Table. 3 shows the results of multiple runs of SGD with Momentum, Adagrad, RMSProp and Adam on *Newsgroups20* dataset with several choices of step-size. Each method is 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. 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, does one see reasonable performance.

Remark: *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 [16, 17] for theoretical guarantees in this regard.*

4.6 Performance Comparison with Quasi-Newton methods We compare our methods to well-known quasi-newton methods *BFGS* and their

Table 2: Performance comparison between first-order and second-order methods.

Time vs. Accuracy	Time vs. Misfit	Time vs. Accuracy	Time vs. Misfit
First Order Batch Size = 128		First Order Batch Size = 20%	
Alg. 1 Gradient Sample Size = 100%		Alg. 1 Gradient Sample Size = 20%	
Alg. 1 Hessian Sample Size = 5%		Alg. 1 Hessian Sample Size = 5%	



limited-memory variants, *SR1* and *Nonlinear-CG* with Fletcher-Reeves' step size formulation. Our implementation of these methods is done in C++/CUDA8.0 and benchmarked on GPUs. Note that quasi-newton methods use *strong-wolf* conditions to compute the step-size. This is more expensive than Armijo-type conditions in eq. (2.3c) used by our methods. We refer readers to [14] for detailed discussion of these methods. Additional performance results can be found in [11].

4.6.1 Drive Diagnostics Dataset Plots in row 1 of Table. 4 show the results for the *Drive Diagnostics* dataset. We clearly notice that *NonlinearCG* is the only quasi-Newton method that achieves comparable results to Newton-type methods, irrespective of the gradient sample size. *SR1* variants yield sub-par performance compared to all the methods irrespective of the gradient sample size. Variants of *BFGS* method are negatively affected when gradient sample size is 20% where subsampled Newton-type methods are robust to such changes.

4.6.2 MNIST and CIFAR-10 Datasets Results for *MNIST* and *CIFAR-10* datasets are shown in Rows 2 and 3 respectively of Table. 4. We notice Newton-type methods outperforming all quasi-Newton methods for *CIFAR-10* dataset. Note that this dataset is highly ill-conditioned, which impacts all the quasi-Newton methods significantly, whereas Newton-type methods are robust. For the *MNIST* dataset, when gradient-sample size is 20%, we notice variants of *BFGS* and *SR1* making very little progress in minimizing the objective function until the end of the execution, whereas Newton-type methods achieve significantly lower values in the

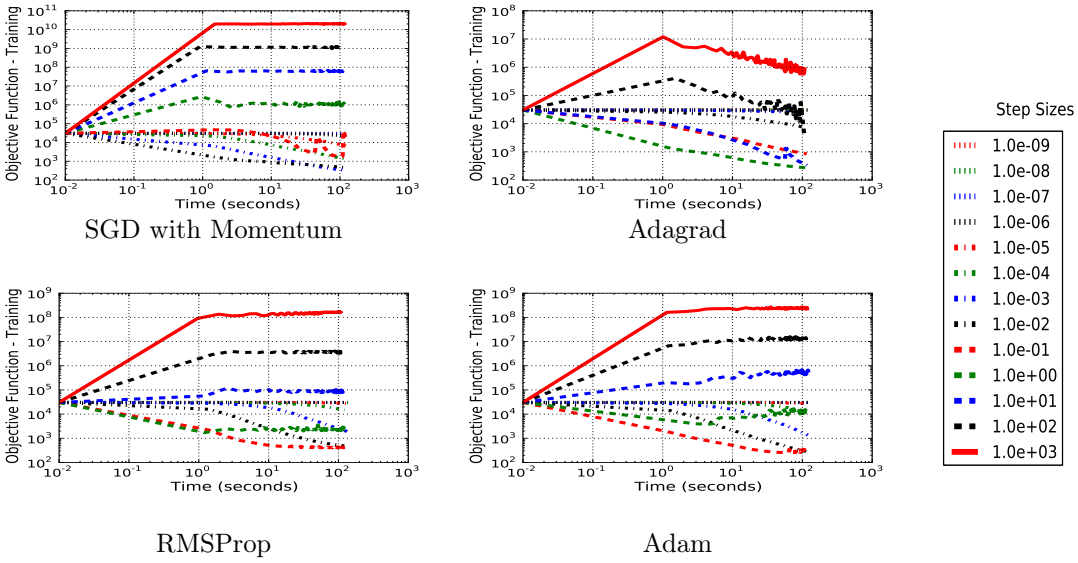


Table 3: Sensitivity of various first-order methods with respect to the choice of the step-size, i.e., learning-rate. It is clear that, too small a step-size can lead to slow convergence, while larger step-sizes cause the method to diverge. The range of step-sizes for which many of these methods perform reasonably, can be very narrow. This is in contrast with Newton-type methods, which come with a priori "natural" step-size, i.e., $\alpha = 1$, and only occasionally require the line-search to intervene.

first few epochs.

4.6.3 Newsgroups20 Dataset Results for the *Newsgroups20* dataset are shown in Row 4 of Table. 4. We notice that Newton-type methods yield better objective function values compared to quasi-newton methods early in the simulations. *BFGS* variants and *NonlinearCG* takes longer time to reach comparable objective function value, 1 sec compared to ≈10 secs. When subsampled gradient is used, interestingly, *BFGS* variants outperform *NonlinearCG* and also *SubsampledNewton-20* method in minimizing the objective function value.

5 Conclusions And Future Work

We presented sampled Hessian Newton solvers, which has been shown to be significantly better than state-of-the-art solvers in terms of solution time, robustness, and use of hardware accelerators. In doing so, we have significantly advanced the state-of-the-art in optimization techniques for training a diverse set of ML applications.

6 Acknowledgements

Fred Roosta acknowledges the generous support by the Australian Research Council through the Centre of Excellence for Mathematical & Statistical Frontiers (ACEMS) as well as the Discovery Early Career Researcher Award (DE180100923). Micheal W. Mahoney would like to acknowledge ARO, DARPA, NSF, and ONR for providing partial support for this work. Sudhir Kylasa's and Ananth Grama's research is supported by the NSF (Grant No. 1546488) and Center for Science of Information (CCF-0939370).

References

[1] 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*, volume 16, pages 265–283, 2016.

[2] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[3] A. S. Berahas, R. Bollapragada, and J. Nocedal. An Investigation of Newton-Sketch and Subsampled Newton Methods. *arXiv preprint arXiv:1705.06211*, 2017.

[4] R. Bollapragada, R. Byrd, and J. Nocedal. Exact and inexact subsampled Newton methods for optimization. *arXiv preprint arXiv:1609.08502*, 2016.

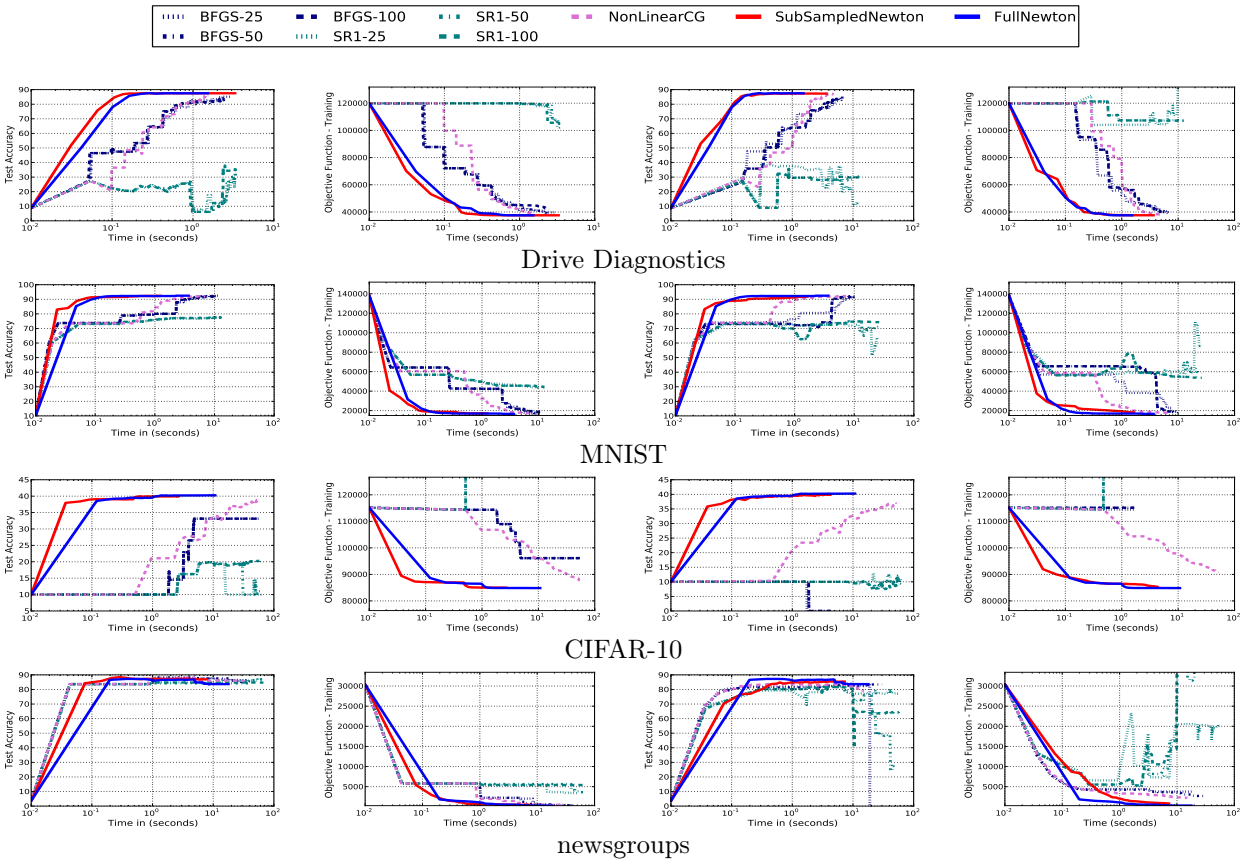
[5] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[6] L. Bottou and Y. LeCun. Large scale online learning. *Advances in neural information processing systems*, 16:217, 2004.

[7] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine

Table 4: Performance comparison between our proposed methods and existing quasi-newton methods.

Time vs. Accuracy	Time vs. Misfit	Time vs. Accuracy	Time vs. Misfit
Alg. 1 Gradient Sample Size = 100%		Alg. 1 Gradient Sample Size = 20%	
Alg. 1 Hessian Sample Size = 5%		Alg. 1 Hessian Sample Size = 5%	



learning. *Mathematical programming*, 134(1):127–155, 2012.

[8] 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*, pages 4287–4293. IEEE, 2009.

[9] M. A. Erdogdu and A. Montanari. Convergence rates of sub-sampled newton methods. In *Advances in Neural Information Processing Systems 28*, pages 3034–3042. 2015.

[10] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[11] S. B. Kylasa, F. Roosta-Khorasani, M. W. Mahoney, and A. Y. Grama. Gpu accelerated sub-sampled newton methods. *https://arxiv.org/abs/1802.09113*, 2018.

[12] K. P. Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, 2012.

[13] 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)*, pages 265–272, 2011.

[14] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[15] 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*, pages 873–880. ACM, 2009.

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

[17] F. Roosta-Khorasani and M. W. Mahoney. Sub-sampled Newton methods II: Local convergence rates. *arXiv preprint arXiv:1601.04738*, 2016.

[18] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[19] S. Sra, S. Nowozin, and S. J. Wright. *Optimization for machine learning*. Mit Press, 2012.