GPU Accelerated Sub-Sampled Newton's Method

Sudhir B. Kylasa [*] Farbod Roosta-Khorasani [†] Michael W. Mahoney [‡]
Ananth Grama [§]

April 19, 2018

Abstract

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 paper, 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.

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

[*] Elec. and Comp. Engg. Dept Purdue Univ., W. Lafayette, Indiana 47907, US skylasa@purdue.edu

[†] School of Mathematics and Physics, Univ. of Queensland St Lucia, QLD 4072, Australia fred.roosta@uq.edu.au

[‡] ICSI and Department of Statistics Univ. of California at Berkeley Berkeley, CA 94720, US mmahoney@stat.berkeley.edu

[§] Comp. Sci. Dept Purdue Univ., W. Lafayette, Indiana 47907, US ayg@cs.purdue.edu

gradient problem" encountered in training deep neural nets [3], 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 [15]. 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 [4, 35].

Newton-type methods use curvature information in the form of the Hessian matrix, in addition to the to 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 [25, 26, 36]. 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 [4, 35]. 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 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

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

where each $f_i(\mathbf{x})$ is a smooth convex function, representing a loss (or misfit) corresponding to i^{th} observation (or measurement) [6, 14, 30]. In many ML applications, F in eq. (1) corresponds to the *empirical risk* [29], and the goal of solving eq. (1) is to obtain a solution with small generalization error, i.e., high predictive accuracy on "unseen" data. We consider eq. (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 computational 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. (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 [20]. Furthermore, non-trivial practical implementations of algorithms based on these ideas have been shown to beat state-of-the-art numerical techniques [2, 21, 37]. For more general problems, theoretical properties of sub-sampled Newton-type methods, for both convex and non-convex problems of the form in eq. (1), have been recently studied in a series of efforts [5, 8, 13, 25, 26, 34, 36]. *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 paper 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.* (1)*, our accelerated randomized method significantly outperforms state of the art implementations of existing techniques in popular ML/DA software packages such as TensorFlow [1], in terms of improved training time, generalization error, and robustness to various adversarial effects.*

This paper is organized as follows. Section 2 provides an overview of related literature. Section 3 presents technical background regarding sub-sampled Newton-type methods, Softmax classifier as a practical instance of eq. (1), along with a description of the algorithms and their implementation. Section 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 5.

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, [1], has enjoyed considerable success among ML practitioners. Among first-order methods implemented in TensorFlow for solving (1) are Adagrad [12], RMSProp [32], Adam [16], Adadelta [38], and SGD with/ without momentum [31]. 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), have been recently studied in a series of results, both in the context of ML applications [5,8,13,25,26,34,35,36], as well as scientific computing applications [11,27,28].

GPUs have been successfully used in a variety of ML applications to speed up computations [9,10,22,24]. In particular, Raina et al. [24] 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. [22] 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 Newton's method, is a major motivating factor for our work here.

3 Theory, Algorithms and Implementation Details

3.1 Notation

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 $|\mathcal{S}|$. Following `Matlab` notation,

$[\mathbf{v};\mathbf{w}] \in \mathbb{R}^{2p}$ denotes vertical stacking of two column vectors $\mathbf{v},\mathbf{w} \in \mathbb{R}^p$, whereas $[\mathbf{v},\mathbf{w}] \in \mathbb{R}^{p\times 2}$ denotes a p by 2 matrix whose columns are formed from the vectors $\mathbf{v}$ and $\mathbf{w}$. Vector ℓ_2 norm is denoted by $\|\mathbf{x}\|$. For a boolean variable, $x \in \{\text{True}, \text{False}\}$, the indicator function $\mathbf{1}(x)$ evaluates to one if $x = \text{True}$, and zero otherwise. $<\mathbf{u},\mathbf{v}>= \mathbf{u}^T\mathbf{v}$ denotes the dot product of vectors $\mathbf{u}$ and $\mathbf{v}$, and $\mathbf{A}\odot\mathbf{B}$ represents element-wise multiplication of matrices $\mathbf{A}$ and $\mathbf{B}$.

3.2 Sub-Sampled Newton's Method

For the optimization problem eq. (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

$$\mathbf{g}(\mathbf{x}) \triangleq \frac{n}{|\mathcal{S}_{\mathbf{g}}|} \sum_{j\in\mathcal{S}_{\mathbf{g}}} \nabla f_j(\mathbf{x}), \tag{2a}$$

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

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 [25, 26]. 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

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k, \tag{3a}$$

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

$$\|\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k + \mathbf{g}(\mathbf{x}^{(k)})\| \le \theta \|\mathbf{g}(\mathbf{x}^{(k)})\|, \tag{3b}$$

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

$$F(\mathbf{x}^{(k)} + \alpha\mathbf{p}_k) \le F(\mathbf{x}^{(k)}) + \alpha\beta\mathbf{p}_k^T\mathbf{g}(\mathbf{x}^{(k)}), \tag{3c}$$

for some $\beta \in (0, 1)$. The requirement in eq. (3c) is often referred to as Armijo-type line-search [23], and eq. (3b) is the θ-relative error approximation condition of the exact solution to the linear system

$$\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k = -\mathbf{g}(\mathbf{x}^{(k)}), \tag{4}$$

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. (4) satisfying eq. (3b). It has also been shown [25, 26], that to inherit the convergence properties of the, rather expensive, algorithm that employs the exact solution to eq. (4), the inexactness tolerance, θ, in eq. (3b) 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 4). Putting all of these together, we obtain Algorithm 1, which under specific assumptions, has been shown [25, 26] to be globally linearly convergent[1] with problem-independent local convergence rate [2].

[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.

Algorithm 1: Sub-Sampled Newton Method

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

3.3 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. (1). Consider a p dimensional feature vector $\mathbf{a}$, with corresponding 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 $\mathbf{Pr}\left(b = c \mid \mathbf{a}, \mathbf{w}_1, \dots, \mathbf{w}_C\right) = e^{\langle \mathbf{a}, \mathbf{w}_c \rangle} / \sum_{c'=1}^{C} e^{\langle \mathbf{a}, \mathbf{w}_{c'} \rangle}$, 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 $\mathbf{x}_c \triangleq \mathbf{w}_c - \mathbf{w}_C$, $c = 1, 2, \dots, C - 1$, 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

$$\begin{aligned} F(\mathbf{x}) \triangleq & F(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{C-1}) \\ = & \sum_{i=1}^{n} \left(\log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) - \sum_{c=1}^{C-1} \mathbf{1}(b_i = c) \langle \mathbf{a}_i, \mathbf{x}_c \rangle \right). \end{aligned} \tag{5}$$

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

$$\nabla_{\mathbf{x}_c} F(\mathbf{x}) = \sum_{i=1}^{n} \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \mathbf{1}(b_i = c) \right) \mathbf{a}_i. \tag{6}$$

Similarly, for the full Hessian of F, we have

$$\begin{aligned} & \nabla^2_{\mathbf{x}_c, \mathbf{x}_c} F = \\ & \sum_{i=1}^{n} \left(\frac{e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}} - \frac{e^{2\langle \mathbf{a}_i, \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{a}_i^T, \end{aligned} \tag{7a}$$

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

$$\nabla^2_{\mathbf{x}_c, \mathbf{x}_{\hat{c}}} F = \sum_{i=1}^{n} \left(- \frac{e^{\langle \mathbf{a}_i, \mathbf{x}_{\hat{c}} + \mathbf{x}_c \rangle}}{\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle}\right)^2} \right) \mathbf{a}_i \mathbf{a}_i^T. \tag{7b}$$

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

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

3.3.1 Numerical Stability

To avoid over-flow in the evaluation of exponential functions in (5), we use the "Log-Sum-Exp" trick [**?**]. 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

$$M(\mathbf{a}) = \max \Big\{ 0, \langle \mathbf{a}, \mathbf{x}_1 \rangle, \langle \mathbf{a}, \mathbf{x}_2 \rangle, \ldots, \langle \mathbf{a}, \mathbf{x}_{C-1} \rangle \Big\}, \tag{8}$$

and

$$\alpha(\mathbf{a}) := e^{-M(\mathbf{a})} + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'} \rangle - M(\mathbf{a})}. \tag{9}$$

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 (5), we use $\log \left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'} \rangle} \right) = M(\mathbf{a}_i) + \log \big(\alpha(\mathbf{a}_i) \big)$. Similarly, for (6) and (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.

3.3.2 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. (8) and (9), respectively. Now using matrices

$$\mathbf{V} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{v}_1 \rangle & \langle \mathbf{a}_1, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_1, \mathbf{v}_{C-1} \rangle \\ \langle \mathbf{a}_2, \mathbf{v}_1 \rangle & \langle \mathbf{a}_2, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_2, \mathbf{v}_{C-1} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{v}_1 \rangle & \langle \mathbf{a}_n, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_n, \mathbf{v}_{(C-1)} \rangle \end{bmatrix}_{n \times (C-1)}, \tag{10}$$

and

$$\mathbf{W} = \begin{bmatrix} h(\mathbf{a}_1, \mathbf{x}_1) & h(\mathbf{a}_1, \mathbf{x}_2) & \ldots & h(\mathbf{a}_1, \mathbf{x}_{C-1}) \\ h(\mathbf{a}_2, \mathbf{x}_1) & h(\mathbf{a}_2, \mathbf{x}_2) & \ldots & h(\mathbf{a}_2, \mathbf{x}_{C-1}) \\ \vdots & \vdots & \ddots & \vdots \\ h(\mathbf{a}_n, \mathbf{x}_1) & h(\mathbf{a}_n, \mathbf{x}_2) & \ldots & h(\mathbf{a}_n, \mathbf{x}_{C-1}) \end{bmatrix}_{n \times (C-1)}, \tag{11}$$

we compute

$$\mathbf{U} = \mathbf{V} \odot \mathbf{W} - \mathbf{W} \odot \Big(\big((\mathbf{V} \odot \mathbf{W})\, \mathbf{e} \big) \mathbf{e}^T \Big), \tag{12}$$

to get

$$\mathbf{H}\mathbf{v} = \text{vec}\left(\mathbf{A}^T \mathbf{U} \right), \tag{13}$$

where $\mathbf{v} = [\mathbf{v}_1; \mathbf{v}_2; \ldots; \mathbf{v}_{C-1}] \in \mathbb{R}^d$, $\mathbf{v}_i \in \mathbb{R}^p, i = 1, 2, \ldots, C-1$, $\mathbf{e} \in \mathbb{R}^{C-1}$ 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, $\mathbf{A}^T = \left[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n \right]$.

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.*

3.4 Implementation Details

We present a brief overview of the algorithmic machinery involved in the implementation of iterations described in eq. (3) and applied to the function defined in eq. (5) 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 2, we depict a slightly modified implementation of the classical CG, to approximately solve the linear system in eq. (4), i.e., $\mathbf{H}\mathbf{p} = -\mathbf{g}$, to satisfy eq. (3b). This routine takes a function (pointer), $H(.)$, 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 $\mathbf{r}$, 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. (3b) is evaluated on line 11. Finally, the search direction, $\mathbf{s}$, is updated in line 12.

Line Search method We use a simple back-tracking line search, shown in Algorithm 3 for computing the step size in eq. (3c). 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 [25] 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 4. This function takes a 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 $\Sigma_{j=1}^{C-1} \mathbf{1}(\mathbf{b}_i = j)(\mathbf{a}_i^T \mathbf{x}_j)$, and "sumExpPart" stores the summation in eq. (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., $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{C-1}]$, and $\mathbf{A}$ is as in eq. (13). Line 1 initializes the *idx*, thread-id of a given thread.

```
Algorithm 2: Conjugate-Gradient
  Input       :
                $H(.)$ - Pointer to Algorithm 7 to compute
                Hessian-vector product, $H(\mathbf{v}) = \mathbf{Hv}$
                $\mathbf{g}$ - Gradient
  Parameters:
                $\theta$ - Relative residual tolerance
                $T$ - Maximum no. of iterations
  Result: $\mathbf{p}_{\text{best}}$, an approximate solution to $\mathbf{Hp} = -\mathbf{g}$
  $\mathbf{p}_0 = 0$
  $\mathbf{r}_0 = -\mathbf{g}$ // initial residual vector
  $\mathbf{s}_0 = \mathbf{r}_0$ // initial search direction
  $\mathbf{p}_{\text{best}} = \mathbf{s}_0$ // best solution so far
  $\mathbf{r}_{\text{best}} = \mathbf{r}_0$
  foreach $k = 0, 1, \ldots, T$ do
      $\alpha_k = \mathbf{r}_k^T \mathbf{r}_k / \mathbf{s}_k^T H(\mathbf{s}_k)$
      $\mathbf{p}_{k+1} = \mathbf{p}_k + \alpha_k \mathbf{s}_k$
      $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k H(\mathbf{s}_k)$
      if $\|\mathbf{r}_{k+1}\| \leq \|\mathbf{r}_{best}\|$ then
          $\mathbf{r}_{\text{best}} = \mathbf{r}_{k+1}$
          $\mathbf{p}_{\text{best}} = \mathbf{p}_{k+1}$
      end
      if $\|\mathbf{r}_{k+1}\| \leq \theta \|\mathbf{g}\|$ then
          break
      end
      $\mathbf{s}_{k+1} = \mathbf{r}_{k+1} + \frac{\|\mathbf{r}_{k+1}\|_2^2}{\|\mathbf{r}_k\|_2^2} \mathbf{s}_k$
  end
```

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.

Softmax function evaluation Subroutine *ComputeFX*, shown in Algorithm 5, describes the evaluation of objective function at a given point, $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_{C-1}] \in \mathbb{R}^d$. Line 2 initializes the memory to store partial results, and line 3 computes the matrix-matrix product between training set, $\mathbf{A}$, and weight matrix, $\mathbf{X}$. By invoking the CUDA function, *ComputeExp*, we compute the partial results, *maxPart, sumExpPart, linearPart*, as described in eqs. (14)–(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(.)*, 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 [18]. *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.,

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

Algorithm 3: Line Search

```
Input      :
             $\mathbf{x}$ - Current point
             $\mathbf{p}$ - Newton's direction
             $F(.)$ - Function pointer
             $\mathbf{g}(\mathbf{x})$ - Gradient
Parameters:
             $\alpha$ - Initial step size
             $0 < \beta < 1$ - Cost function reduction constant
             $0 < \rho < 1$ - 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
    if $i > i_{\max}$ then
        break
    end
    $i = i + 1$
    $\alpha \leftarrow \rho\alpha$
end
```

where

$$\text{maxPart}_i = M(\mathbf{a}_i) \qquad \text{(cf. eq. (8))}, \tag{14}$$

$$\text{sumExpPart}_i = \sum_{c=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_c \rangle - \text{maxPart}_i}, \tag{15}$$

$$\text{linearPart}_i = \sum_{c=1}^{C-1} \mathbf{1}(\mathbf{b}_i = c)\langle \mathbf{a}_i, \mathbf{x}_c \rangle, \tag{16}$$

$$\text{logPart}_i = \log\left(e^{-\text{maxPart}_i} + \text{sumExpPart}_i\right). \tag{17}$$

Softmax gradient evaluation Subroutine *Compute* ∇F, shown in Algorithm 6, describes the computation of $\nabla F(x)$. Line 1 initializes the memory to store temporary results. Algorithm 4 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.

Softmax Hessian-vector evaluation For a given vector, $\mathbf{q}$, Algorithm 7, computes the *Hessian-vector* product, $\nabla^2 F(\mathbf{x})\mathbf{q}$. Algorithm 7 is heavily used in CG to solve the linear system $\mathbf{Hx} = -\mathbf{g}$. Line 1 computes $\mathbf{V}$, as shown in eq. (10), a matrix multiplication operation. Line 4 computes $\mathbf{W}$ using a function similar to Algorithm 4, and $\mathbf{U}$ is computed using Alg. 8 at line 5. Finally $\mathbf{Hq}$ is computed by multiplying $\mathbf{A}^T$ and $\mathbf{U}$, and adding the regularization term in line 6.

4 Experimental Results

We present a comprehensive evaluations of the performance of Newton-type methods presented in this paper. We compare our methods to various first-order methods – SGD with momentum (henceforth referred to as Momentum) [31], Adagrad [12], Adadelta [38], Adam [16] and RMSProp

Algorithm 4: ComputeExp

```
input  : $\hat{\mathbf{A}}$ - where $\hat{\mathbf{A}}_{i,j} = \mathbf{a}_i^T \mathbf{x}_j, \forall i \in \{1 \ldots n\}, \forall j \in \{1 \ldots C-1\}$
         b - Training classes
         maxPart- memory pointer to store eq. (14)
         sumExpPart- memory pointer to store eq. (15)
         linearPart- memory pointer to store eq. (16)
         n - no. of rows in $\hat{\mathbf{A}}$
         C - no. of classes
output:  maxPart, sumExpPart, linearPart

Init. idx ;                                                    // thread-id
if idx < $n$ then
    i ← idx % n ;                                              // row no.
    maxPart$_i$ = linearPart$_i$ = sumExpPart$_i$ = 0
    foreach $j$ in $1 : C-1$ do
        if maxPart$_i$ < $\hat{\mathbf{A}}_{i,j}$ then
            maxPart$_i$ = $\hat{\mathbf{A}}_{i,j}$
        end
    end
    foreach $j$ in $1 : C-1$ do
        if b$_i$ == $j$ then
            linearPart$_i$ = $\hat{\mathbf{A}}_{i,j}$
        end
        sumExpPart$_i$ += exp ($\hat{\mathbf{A}}_{i,j}$ - maxPart$_i$ )
    end
end
```

Algorithm 5: ComputeFX

```
input  : A- Training features
         b - Training classes
         x - Weights vector
         $\lambda$ - 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

Initialize maxPart, linearPart, sumExpPart to store eqs. (14)–(16),
Form $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{C-1}]_{p \times (C-1)}$
$\hat{\mathbf{A}} = \mathbf{A} \times \mathbf{X}$ ;                          // matrix-matrix multiplication
ComputeExp( $\hat{\mathbf{A}}$, b , maxPart, sumExpPart, linearPart, n, C)
Reduce( linearPart, pLin, n, $t(z) = z$ )
Reduce( maxPart, pMax, n, $t(z) = z$ )
Reduce( sumExpPart, pExp, n, $t(z) = z$ )
temp ← maxPart + sumExpPart
Reduce( temp, pLog, n, $t(z) = log(z)$ )
$F(\mathbf{x})$ ← (pMax + pLog - pLin ) + $\lambda \parallel \mathbf{x} \parallel^2 /2$
```

Algorithm 6: Compute ∇F

```
input  : A- Training features
         b - Training classes
         x - Weights vector
         λ - Regularization
output: ∇F(x) - gradient evaluated at x
Initialize $\mathbf{BInd}_{(n \times C-1)}$
Form $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{C-1}]_{p \times (C-1)}$
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. 4
$\nabla F(\mathbf{x}) \leftarrow \mathrm{vec}(\mathbf{A}^T\ \mathbf{BInd} + \lambda\ \mathbf{X})$
```

Algorithm 7: Compute Hessian-Vector Product, $\nabla^2 F(\mathbf{x})\mathbf{q}$

```
input  : A- Training dataset
         λ - Regularization
         x - Weights vector
         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
Init. idx ;                                          // thread-id
Form $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{C-1}]_{p \times (C-1)}$
$\mathbf{V} = \mathbf{A} \times \mathbf{Q}$
$\mathbf{W} \leftarrow$ compute as shown in (11), similar to kernel Alg.4
$\mathbf{U} \leftarrow$ ComputeU ($\mathbf{V}$, $\mathbf{W}$, n, p, C )
$\mathbf{Hq} \leftarrow \mathrm{vec}(\ \mathbf{A}^T\mathbf{U} + \lambda\mathbf{Q})$
```

[32] as implemented in Tensorflow [1]. 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 [17]. Additionally, raw datasets are also available from the UCI Machine Learning Repository [33].

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. (5) and λ is the regularization parameter. Seven real datasets are used for performance comparisons. 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 [25], 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 3.3), 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.

```
Algorithm 8: ComputeU
  input  : V- matrix V as in eq. (10)
           W- matrix W as in eq. (11)
           n - no. of sample points
           p - no. of features
           C - no. of classes
  output: U : matrix U as shown in (12)

  Initialize idx ;                                        // thread-id
  sum = 0
  if idx < $n$ then
      i = 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
          $\mathbf{U}_{i,j} = \mathbf{V}_{i,j} \times \mathbf{W}_{i,j} - \mathbf{W}_{i,j} \times$ sum;
      end
  end
```

Table 1: Description of the datasets.

Classification	Dataset	Train Size (n)	Test Size	No. of Features (p)	No. of Classes (C)	Lipschitz Const. (L)
Multi-Class	Covertype	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

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 [7]), 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 [23], 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.

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 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.

4.4 Performance Comparisons

Table 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".

4.4.1 Covertype Dataset

The first row in Table 2 shows the plots for *Covertype* 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 7. This can be attributed to processing larger batches of the dataset by the GPU-cores, yielding higher efficiency.

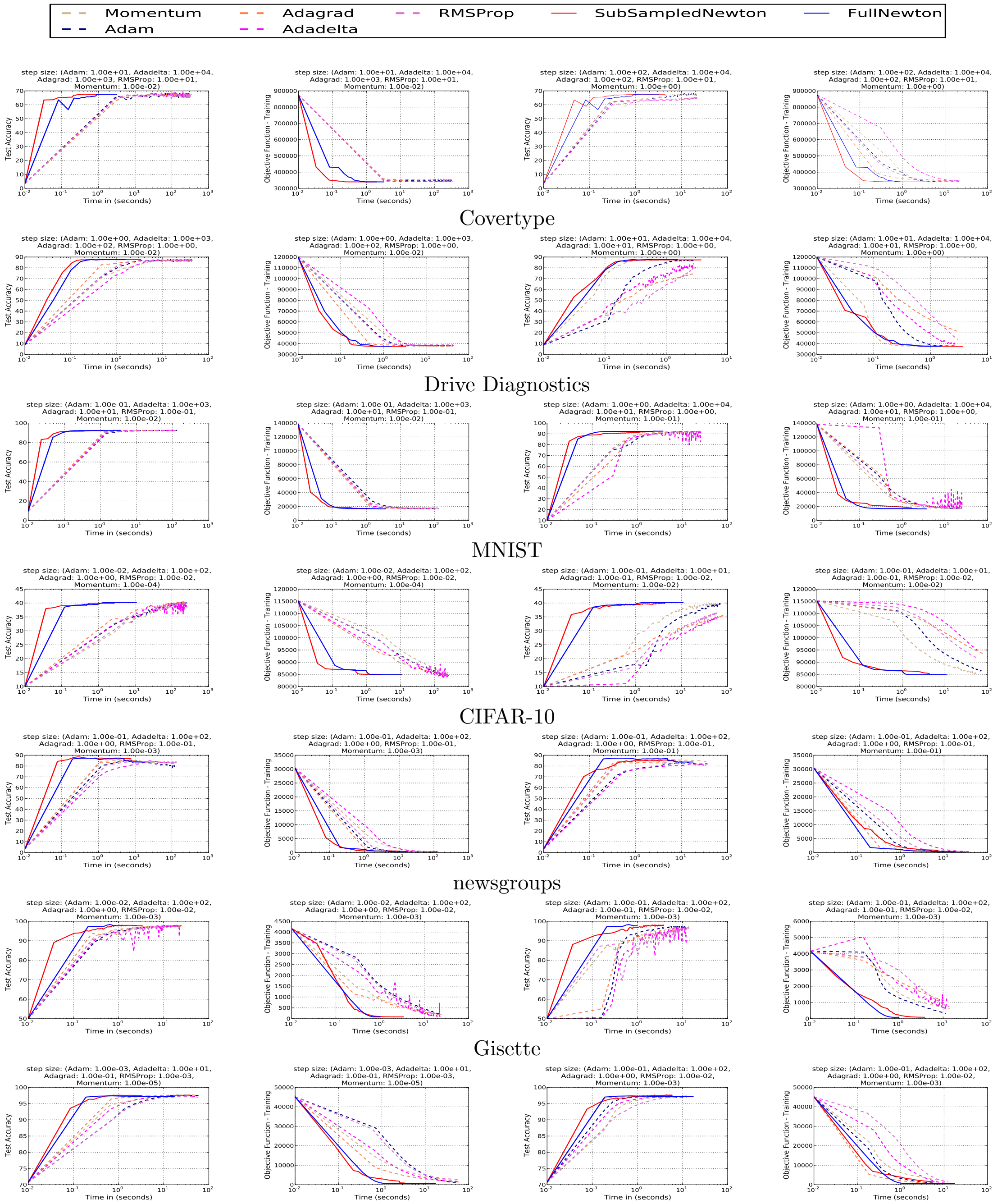
4.4.2 Drive Diagnostics Dataset

Results for the *Drive Diagnostics* dataset are shown in the second row of Table 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 $\approx$ 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.

Table 2: Performance comparison between first-order and second-order methods. First order methods, with batch size 128, are compared with *SubsampledNewton* using full gradient and a Hessian sample size of 5%. First order methods, with batch size 20%, are compared with *SubsampledNewton* using sample sizes of 20% and 5% for gradient and Hessian, respectively. *FullNewton* uses the entire dataset for gradient and Hessian evaluations.

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%	



step size: (Adam: 1.00e+01, Adadelta: 1.00e+04, Adagrad: 1.00e+03, RMSProp: 1.00e+01, Momentum: 1.00e-02)

step size: (Adam: 1.00e+01, Adadelta: 1.00e+04, Adagrad: 1.00e+03, RMSProp: 1.00e+01, Momentum: 1.00e-02)

step size: (Adam: 1.00e+02, Adadelta: 1.00e+04, Adagrad: 1.00e+02, RMSProp: 1.00e+01, Momentum: 1.00e+00)

step size: (Adam: 1.00e+02, Adadelta: 1.00e+04, Adagrad: 1.00e+02, RMSProp: 1.00e+01, Momentum: 1.00e+00)

Covertype

step size: (Adam: 1.00e+00, Adadelta: 1.00e+03, Adagrad: 1.00e+02, RMSProp: 1.00e+00, Momentum: 1.00e-02)

step size: (Adam: 1.00e+00, Adadelta: 1.00e+03, Adagrad: 1.00e+02, RMSProp: 1.00e+00, Momentum: 1.00e-02)

step size: (Adam: 1.00e+01, Adadelta: 1.00e+04, Adagrad: 1.00e+01, RMSProp: 1.00e+00, Momentum: 1.00e+00)

step size: (Adam: 1.00e+01, Adadelta: 1.00e+04, Adagrad: 1.00e+01, RMSProp: 1.00e+00, Momentum: 1.00e+00)

Drive Diagnostics

step size: (Adam: 1.00e-01, Adadelta: 1.00e+03, Adagrad: 1.00e+01, RMSProp: 1.00e-01, Momentum: 1.00e-02)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+03, Adagrad: 1.00e+01, RMSProp: 1.00e-01, Momentum: 1.00e-02)

step size: (Adam: 1.00e+00, Adadelta: 1.00e+04, Adagrad: 1.00e+01, RMSProp: 1.00e+00, Momentum: 1.00e-01)

step size: (Adam: 1.00e+00, Adadelta: 1.00e+04, Adagrad: 1.00e+01, RMSProp: 1.00e+00, Momentum: 1.00e-01)

MNIST

step size: (Adam: 1.00e-02, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-04)

step size: (Adam: 1.00e-02, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-04)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+01, Adagrad: 1.00e-01, RMSProp: 1.00e-02, Momentum: 1.00e-02)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+01, Adagrad: 1.00e-01, RMSProp: 1.00e-02, Momentum: 1.00e-02)

CIFAR-10

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-01, Momentum: 1.00e-03)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-01, Momentum: 1.00e-03)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-01, Momentum: 1.00e-01)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-01, Momentum: 1.00e-01)

newsgroups

step size: (Adam: 1.00e-02, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-03)

step size: (Adam: 1.00e-02, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-03)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e-01, RMSProp: 1.00e-02, Momentum: 1.00e-03)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e-01, RMSProp: 1.00e-02, Momentum: 1.00e-03)

Gisette

step size: (Adam: 1.00e-03, Adadelta: 1.00e+01, Adagrad: 1.00e-01, RMSProp: 1.00e-03, Momentum: 1.00e-05)

step size: (Adam: 1.00e-03, Adadelta: 1.00e+01, Adagrad: 1.00e-01, RMSProp: 1.00e-03, Momentum: 1.00e-05)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-03)

step size: (Adam: 1.00e-01, Adadelta: 1.00e+02, Adagrad: 1.00e+00, RMSProp: 1.00e-02, Momentum: 1.00e-03)

real-sim

4.4.3 MNIST and CIFAR-10 Datasets

Rows 3 and 4 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 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. 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 ≈ 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.

4.4.4 Newsgroups20 Dataset

Plots in row 5 of Table 2 correspond to *Newsgroups20* dataset. This is a sparse dataset, and the largest in the scope of this work (the Hessian is ≈ 1e6 × 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 ≈ 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.

4.4.5 Gisette and Real-Sim Datasets

Rows 6 and 7 in Table 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 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 2^{nd} iteration whereas it takes 11 iterations for *SubsampledNewton-20*.

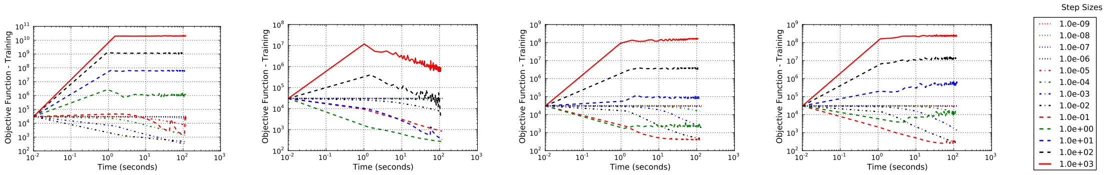
4.5 Sensitivity to Hyper-Parameter Tuning

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 [4, 35]. 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 [4, 35].

Here, to further highlight such issues, we demonstrate the sensitivity of several first-order methods with respect to their learning rate. Figure 1 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, 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 [25, 26] for theoretical guarantees in this regard.*



(a) SGD with Momentum

(b) Adagrad

(c) RMSProp

(d) Adam

Figure 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

5 Conclusions And Future Work

In this paper, 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.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Haim Avron, Petar Maymounkov, and Sivan Toledo. Blendenpik: Supercharging LAPACK's least-squares solver. *SIAM Journal on Scientific Computing*, 32(3):1217–1236, 2010.

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

[4] Albert S Berahas, Raghu Bollapragada, and Jorge Nocedal. An Investigation of Newton-Sketch and Subsampled Newton Methods. *arXiv preprint arXiv:1705.06211*, 2017.

[5] Raghu Bollapragada, Richard Byrd, and Jorge Nocedal. Exact and inexact subsampled Newton methods for optimization. *arXiv preprint arXiv:1609.08502*, 2016.

[6] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[7] Léon Bottou and Yann LeCun. Large scale online learning. *Advances in neural information processing systems*, 16:217, 2004.

[8] Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012.

[9] Adam Coates, Paul Baumstarck, Quoc Le, and Andrew 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.

[10] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[11] Kees van den Doel and Uri Ascher. Adaptive and stochastic algorithms for EIT and DC resistivity problems with piecewise constant solutions and many measurements. *SIAM J. Scient. Comput.*, 34:DOI: 10.1137/110826692, 2012.

[12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

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

[14] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[15] Alex Gittens and Michael W Mahoney. Revisiting the Nyström method for improved large-scale machine learning. *The Journal of Machine Learning Research*, 17(1):3977–4041, 2016.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Sudhir B Kylasa. Newton-cg cuda implementation download (scripts/code/tensorflow-python-scripts). *https://github.com/kylasa/NewtonCG*, February 2018.

[18] Sudhir B Kylasa, Hasan Metin Aktulga, and Ananth Y Grama. Puremd-gpu: A reactive molecular dynamics simulation package for gpus. *Journal of Computational Physics*, 272:343–359, September 2014.

[19] Sudhir B Kylasa, Farbod Roosta-Khorasani, Michael W. Mahoney, and Ananth Y Grama. Gpu accelerated sub-sampled newton methods. *https://www.cs.purdue.edu/homes/skylasa/papers/newton-cg-arXiv.pdf*, 2018.

[20] Michael W Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

[21] Xiangrui Meng, Michael A Saunders, and Michael W Mahoney. LSRN: A parallel iterative solver for strongly over-or underdetermined systems. *SIAM Journal on Scientific Computing*, 36(2):C95–C118, 2014.

[22] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 265–272, 2011.

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

[24] Rajat Raina, Anand Madhavan, and Andrew 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.

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

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

[27] Farbod Roosta-Khorasani, Kees van den Doel, and Uri Ascher. Data completion and stochastic algorithms for PDE inversion problems with many measurements. *Electronic Transactions on Numerical Analysis*, 42:177–196, 2014.

[28] Farbod Roosta-Khorasani, Kees van den Doel, and Uri Ascher. Stochastic algorithms for inverse problems involving PDEs and many measurements. *SIAM J. Scientific Computing*, 36(5):S3–S22, 2014.

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

[30] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning.* Mit Press, 2012.

[31] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[32] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[33] UCI. Uci machine learning repository. *http://archive.ics.uci.edu/ml/index.php*, 02 2018.

[34] Peng Xu, Farbod Roosta-Khorasani, and Michael W. Mahoney. Newton-Type Methods for Non-Convex Optimization Under Inexact Hessian Information. *arXiv preprint arXiv:1708.07164*, 2017.

[35] Peng Xu, Farbod Roosta-Khorasani, and Michael W. Mahoney. Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study. *arXiv preprint arXiv:1708.07827*, 2017.

[36] Peng Xu, Jiyan Yang, Farbod Roosta-Khorasani, Christopher Ré, and Michael W Mahoney. Sub-sampled newton methods with non-uniform sampling. In *Advances in Neural Information Processing Systems*, pages 3000–3008, 2016.

[37] Jiyan Yang, Xiangrui Meng, and Michael W Mahoney. Implementing randomized matrix algorithms in parallel and distributed environments. *Proceedings of the IEEE*, 104(1):58–92, 2016.

[38] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

6 More Details On Softmax Function (5)

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 (5).

6.1.1 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 (7a) and (7b), respectively. Note that since

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

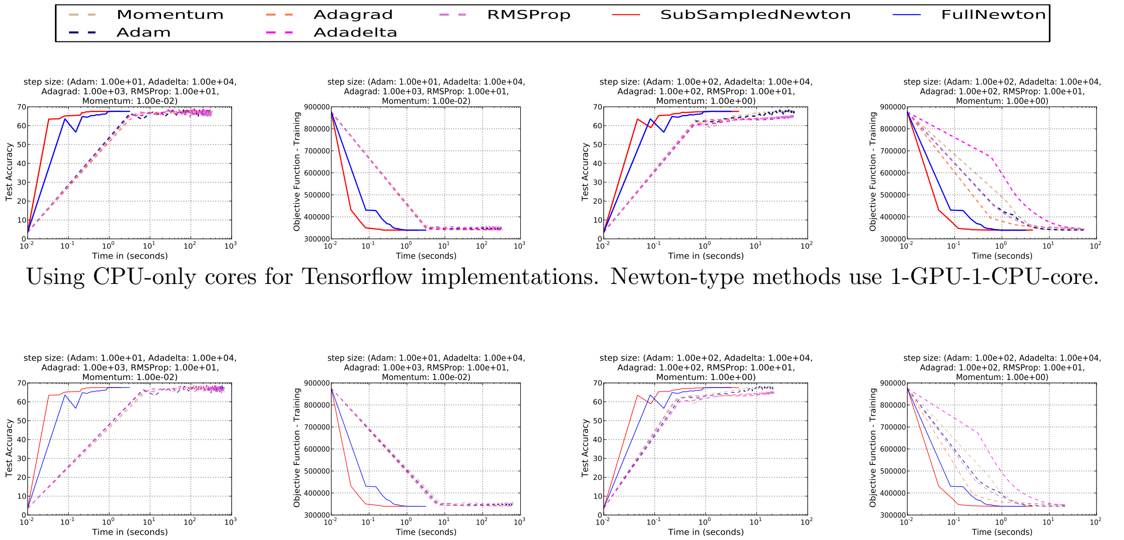
the matrix $\mathbf{W}$ is strictly diagonally dominant, and hence it is symmetric positive definite. So the problem is convex (in fact it is strictly-convex if the data matrix X is full column rank).

7 Tensorflow's Performance Comparison on Various Compute Platforms

Columns 1 and 2 of table 3 plots the results for *covertype* 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

Table 3: Performance comparison between first-order and second-order methods on CPU-only and 1-GPU-1-CPU-core compute platforms for *covertype* 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.

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



Using CPU-only cores for Tensorflow implementations. Newton-type methods use 1-GPU-1-CPU-core.

Using 1-GPU-1-CPU-core for Tensorflow implementations. Newton-type methods use 1-GPU-1-CPU-core.

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 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 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× 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, ≈ 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.