FITRE: Fisher Informed Trust-region Method for Training Deep Neural Networks

Sudhir B. Kylasa [*] Chih-Hao Fang [†] Peng Xu [‡] Fred Roosta [§]
Michael W. Mahoney [¶] Ananth Grama [‖]

May 7, 2020

Abstract

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

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

1 Introduction and Motivation

Significant growth in the availability of large datasets, coupled with growth in the processing capacity of hardware has motivated new applications that rely on analyzing massively large datasets quickly, often in real time. Optimization in such applications involves iterating over a large dataset multiple times and learning model parameters until some predefined criteria of convergence is achieved. These processes may take hours for each iteration over the complete dataset, which translates to learning procedures that can take up to weeks. For instance, on the ImageNet dataset [2], which contains about 1.4 million samples, stochastic gradient descent (SGD) can take upwards of 30 hours for each pass over the dataset and weeks to train a deep neural network such as ResNet [22] or VGGNet [45].

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

[†] Computer Science Dept, Purdue Univ. W. Lafayette fang150@purdue.edu

[‡] Stanford University. peng.xu@stanford.edu

[§] University of Queensland. fred.roosta@uq.edu.au

[¶] ICSI and Dept of Statistics, University of California at Berkeley. mmahoney@stat.berkeley.edu

[‖] Computer Science Dept, Purdue Univ. W. Lafayette, ayg@cs.purdue.edu

In spite of this significant training time, SGD and its distributed variants are the methods of choice in training of deep learning models.

The popularity of SGD, to a great extent, is attributed to its computationally inexpensive model parameter updates. Indeed, SGD's iterations involve computing the gradient of the objective function on a mini-batch, using *back-propagation* [30, 36], followed by scaling by a predetermined learning rate, possibly accelerated by momentum [38]. The simplicity of SGD allows for its application to a wide variety of learning tasks, e.g., auto-encoders [17, 47] and reinforcement learning [32, 43]. However, even though, in theory, it has been argued that SGD can avoid undesirable saddle-points [14], realizing this in practice is more involved [49]. In fact, without significant fine-tuning in terms of parameters and hyperparameters for learning rate, initialization, and mini-batch size, SGD's performance (with or without momentum) can diverge significantly from idealized theoretical bounds. On the other hand, by leveraging curvature information, in the form of the *Hessian* matrix, many second-order methods (e.g., trust region based methods [10, 48, 49]) come with the inherent ability to navigate their way out of flat regions including saddle points [6, 7, 8, 11, 12, 44]. However, the challenge of the second-order methods is to solve the subproblems, e.g. a constrained a quadratic problem for trust-region method, a cubic regularized quadratic problem for cubic regularization methods. Solving the subproblems usually requires significant more computations as well as more complex implementations, as compared with the single gradient update in SGD. To alleviate the issue, stochastic variants of many of these methods have recently been proposed, which, by introducing various approximations of computing Hessians and gradients, can navigate the objective landscape more efficiently [33, 48, 53]. Nevertheless, solving the quadratic subproblems efficiently remains the computation bottleneck of most second-order methods. Furthermore, and in sharp contrast to SGD, many of these methods are resilient to the choices of their hyperparameters, and hence are easy to tune.

A method that occupies the middle ground between first and second order methods relies on the natural gradient [24, 25, 51]. Basically the natural gradient methods assume that the parameters of the probabilistic models lies on the manifold whose geometry is governed by the Fisher information matrix. Under this assumption, scaling the gradient using the Fisher information matrix can result in more effective directions for navigating the manifold of the parametric probability densities. However, in high-dimensional settings, using the exact Fisher matrix can be intractable. To remedy this, [20] proposed the Kronecker Factored Approximated Curvature (KFAC) method to approximate the Fisher information matrix and its *inverse*-vector product, and applied it to applications in neural networks and reinforcement learning[35]. It was shown that KFAC can significantly outperform many of the first-order methods.

In this work, We propose a stochastic trust-region method which utilizes KFAC approximation to solving the trust-region subproblem, to obtain a **F**isher **I**nformed **T**rust **RE**gion method (FITRE) that is shown to be well-suited for optimization of deep learning models in general, and convolutional neural networks (CNN) in particular. We also leverage the power of GPUs in accelerating various steps of FITRE to deliver excellent performance. We make the following contributions:

1. We present a novel stochastic variant of the trust region method, in which the approximations to the sub-problems are informed by directions obtained from KFAC.

2. By employing KFAC directions to inform trust-region sub-problems, we show that the proposed method inherits the *robustness* as well as *invariance to re-parameterizations* of KFAC.

3. We show that highly optimized GPU implementation perform better than well-tuned SGD and quasi-Newton alternatives, in terms of generalization errors, convergence rates, and wall-clock times. With appropriately tuned hyperparameters, in some cases, we can show that our proposed method consumes less wall-clock time compared to alternatives, even for the same number of passes over the dataset.

4. We show that the proposed method is more resilient to the choice of batch size and tuning of the underlying hyperparameters, comparing against the typical first-order methods.

5. As a broader contribution to the user community, we provide an open-source GPU accelerated CUDA optimization framework for CNNs with a *first-of-its-kind* R-operator for Hessian-vector computations.

The rest of this paper is organized as follows: we introduce the notation used in this paper at the end of this section. Section 2 provides an overview of state-of-the-art methods along with a comparison of the proposed method in this context. A detailed discussion of the proposed methods is given in Section 3. Evaluation of our methods as compared with a well-tuned SGD as well as BFGS [39, 40] is provided in Section 4. Section 5 discusses avenues for future work. Section A provides detailed discussion on the implementation of our proposed method.

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 $\mathbf{B}$. $\mathbf{e}$ is used to indicate an all-ones vector, and $\mathbf{e}_i$ is used to indicate a vector in which the i^{th} component is set to 1 while rest of the components are 0's.

2 Related Work

SGD [5] is the most commonly used first-order method, owing to its simplicity and inexpensive per-iteration cost. Iterations require computation of the gradient on a mini-batch scaled by a predetermined learning schedule and possibly Nesterov-accelerated momentum [38]. It has been argued that high-dimensional non-convex functions such as those arising in deep learning are riddled with undesirable saddle points [3, 13, 14, 26] (although this has also been disputed [16]). For instance, convolutional neural networks, CNNs, display structural symmetry in their parameter space, which can lead to an abundance of saddle points [4, 20, 34]. First-order methods, such as SGD, are known to "zig-zag" in high curvature areas and "stagnate" in low curvature regions [4, 14].

One of the primary reasons for the susceptibility of first-order methods to getting trapped in saddle points or nearly flat regions is their reliance on gradient information. Indeed, navigating around saddle points and plateau-like regions can become a challenge for these methods because the gradient is close to zero in most directions [14]. To this end, a number of alternate methods have been proposed in recent times, which, using the history of gradients, aim to approximate curvature information, hence maintaining the simplicity of SGD. Such methods include Adam [27] and Adagrad [15]. However, such approximations of the Hessian do not always properly scale the gradient according to the entire curvature information, and hence these methods suffer from similar deficiencies near saddle points and flat regions. More effective variants of these curvature approximations are those found in quasi-Newton methods such as SR1 [40], DFP [40], and BFGS [31, 40], which use rank-1 and rank-2 updates to approximate the Hessian iteratively. Aided by line search methods, typically satisfying strong-Wolfe [40] conditions, these methods yield good results compared to first-order methods for convex problems [29].

Newton-type optimizers have been developed as alternatives to first-order methods [52]. These optimizers can effectively navigate the steep and flat regions of the optimization landscape. By incorporating curvature information in the form of the Hessian matrix, e.g., negative curvature directions, these methods can escape saddle points [3, 13, 48, 50, 53, 54, 55]. To avoid explicitly forming the Hessian matrices, Hessian-free methods [9, 33, 37, 56] have been proposed, which only require Hessian-vector products. Arguably, a highly effective, if not the most effective, among these methods is the trust-region based method that comes with attractive theoretical guarantees and is relatively easy to implement [11, 48, 49, 53].

Lying on the spectrum between first and second order methods is the natural gradient method [24, 25]. This method provided a new direction in the context of high-dimensional optimization of probabilistic models. In his seminal work, Amari showed that natural gradient descent yields Fisher efficient estimate of the parameters; and he subsequently applied the method to multi-layer perceptrons for solving blind source detection problems. However, naïvely computing the Fisher matrix and its inverse in high-dimensional settings is computationally intractable, both in terms of memory and computational resources. RMSProp [23, 46] methods use a diagonal approximation of Fisher matrix of the objective function to compute the descent direction. These methods incur little overhead with regards to diagonal approximation but nevertheless fail to make progress relative to SGD in some cases. Grosse and Martens [20], Martens [34], Martens and Grosse [35] proposed the KFAC method, which approximates the natural gradient using Kronecker products of smaller matrices formed during back-propagation. KFAC method and its distributed counterpart [4] have been shown to outperform well tuned SGD in many applications.

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

3 FITRE

We now present our method, FITRE, which is inspired by [35] and [48, 53], and is formally described in Algorithm 1. At the heart of FITRE lies the stochastic trust-region method using a local quadratic approximation:

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

We employ the approach proposed by [53] and use stochastic estimation of the gradient $\mathbf{g}_t$ and Hessian $\mathbf{H}_t$. The application of Hessian estimate, $\mathbf{H}_t$, which contains information regarding the curvature of the optimization landscape, has been shown to offer many advantages, including resilience to hyperparameter tuning and problem ill-conditioning [48, 49]. The step-length, which is governed by the trust-region radius Δ_t is automatically adjusted based on the quality of the quadratic approximation and the amount of descent in the objective function. In practice, (1) is approximated by restricting the problem to lower dimensional spaces, e.g., the Cauchy point, which amounts to searching in a one-dimensional space spanned by the gradient. Here, we do the same, however, by restricting the sub-problem to the space spanned by the KFAC direction, or its combination with the gradient.

Our choice is motivated by the following observation: when the objective function involves probabilistic models, as is the case in many deep learning applications, natural gradient[1] direction amounts to the steepest descent direction among all possible directions inside a ball measured

[1] In the following section, we discuss in detail the definition of the Fisher matrix and its computation using the

Table 1: Description of notation we employ

Description	Symbol
A sample point from the dataset and corresponding output	$(\mathbf{x}, y)$
Total number of classes in the dataset	c
Function of the network	$f(\mathbf{x}, y)$
Prediction of the network w.r.t $\mathbf{x}$	z
Loss function	$\mathcal{L}$
Gradient of the network	$\mathbf{g}$
Hessian of the network	$\mathbf{H}$
Fisher information matrix of the network	$\mathbf{F}$
Natural gradient direction	$\mathbf{r}$
Number of layers in the neural network	ℓ, $(1, \ldots, \ell)$
Input to layer l	$\mathbf{a}_{l-1}$
Mini-batch input to layer l	$\tilde{\mathbf{A}}_{l-1}$
Input to layer l, augmented with homogeneous coordinate	$\bar{\mathbf{a}}_{l-1}$
Output of layer l	$\mathbf{a}_l$
Weights of layer l	$\mathbf{W}_l$
Bias of layer l	$\mathbf{b}_l$
Weights and bias folded into single parameter of layer l	$\bar{\mathbf{W}}_l$
Parameters of the network, $\boldsymbol{\theta}$	$[vec(\bar{\mathbf{W}}_1)^\intercal, \ldots, vec(\bar{\mathbf{W}}_\ell)^\intercal]^\intercal$
Output of the convolution function (and input to the activation function) of layer l	$\mathbf{s}_l$
Output of activation function (and input to pool function) of layer l	$\mathbf{p}_l$
Output of pool function of layer l	$\mathbf{a}_l$
Output of the neural network	$\mathbf{a}_{out}(= \mathbf{a}_\ell)$
Output of the loss function	$\mathbf{a}_{loss}$
Derivative of the loss function w.r.t a parameter, $\boldsymbol{\theta}$	$\mathcal{D}\boldsymbol{\theta} = \frac{d\mathcal{L}}{d\boldsymbol{\theta}}$
Gradient terms propagated from the loss function	$\mathbf{g}_{loss} = (\mathcal{D}\mathbf{a}_{out})$
Gradient terms propagated from the pool function of layer l	$\mathbf{g}_l^p \, (= \mathcal{D}\mathbf{p}_l)$
Gradient terms propagated from the activation function of layer l	$\mathbf{g}_l^a \, (= \mathcal{D}\mathbf{s}_l)$
Gradient terms propagated from the convolution function of layer l	$\mathbf{g}_l^{conv} \, (= \mathcal{D}\mathbf{a}_l)$

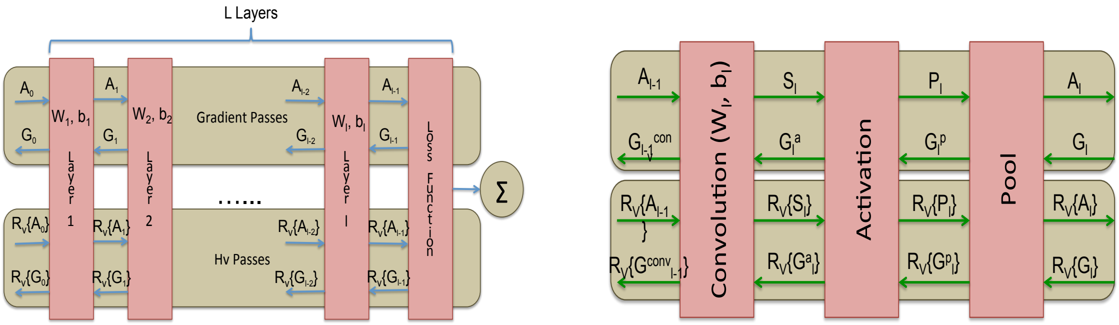
by KL-divergence between the underlying parametric probability densities. On the contrary, the (standard) gradient represents the direction of steepest descent among all directions constrained in a ball measured by the Euclidean metric [20], which is less informative than the former, though it is much easier to compute. To alleviate the computational burden of working with the Fisher information matrix and its inverse, Kronecker-product based approximations [34, 35] have shown success in simultaneously preserving desirable properties of the exact Fisher matrix, such as invariance to reparametarization and resilience to large batch sizes. Indeed, many empirical studies have confirmed that the natural gradient provides an effective descent direction for optimization of neural networks [20, 33, 34, 35].

3.1 Computational Model

Table 1 describes symbols we use and their meaning in the context of deep neural networks.

In this work, we consider a typical CNN architecture, as shown in Figure 1a. We assume that the network contains ℓ layers, and each layer can be either a convolution layer or a linear layer. A convolutional layer is composed of convolution and activation functions. It may also optionally

KFAC approximations.



(a) Model of a typical CNN.

(b) Components of a typical layer in our CNN.

Figure 1: CNN model and layer composition.

contain pooling functions and batch-normalization layers, as shown in Figure 1b. Likewise, a linear layer is composed of a linear function and optionally can have activation functions within it as well. These layers are stacked together, so that the output of one layer forms the input of the next layer. The last layer, i.e., ℓ^{th} layer, is connected to the loss function. In this work, we use *softmax cross-entropy* for our loss function, denoted as $\mathcal{L}$.

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

$$\begin{aligned} \text{Convolution:} \quad & \mathbf{C}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1} && \text{(2a)} \\ \text{Activation:} \quad & \mathbf{P}_l = \phi\left(\mathbf{C}_l\right) && \text{(2b)} \\ \text{Pool:} \quad & \mathbf{A}_l = \mathcal{P}\left(\mathbf{P}_l\right). && \text{(2c)} \end{aligned}$$

The convolution operation is represented as a matrix multiplication, as described in (2a), where $\bar{\mathbf{W}}_l$ and $\mathbf{A}_{l-1}$ are the matrix of weights and input sample point. Note that the weights matrix, $\mathbf{W}_l$ and bias vector, $\mathbf{b}_l$, associated with the l^{th} layer of the network are folded into a single matrix $\bar{\mathbf{W}}_l(= [\mathbf{W}_l \mathbf{b}_l])$ by appending the column vector $\mathbf{b}_l$ into $\mathbf{W}_l$. The output of the convolution operation, $\mathbf{C}_l$, forms the input to the non-linearity function ϕ and its output, $\mathbf{P}_l$, is passed to the down sampling function, $\mathcal{P}$. The output of the pool function, $\mathbf{A}_l$, forms the input to the next layer $l+1$ in the model shown in Figure 1a. During the backward pass of the gradient computation, the partial derivatives with respect to inputs to each layer (referred to as gradients throughout this document) are passed in the backward (opposite) direction though the network as shown in Figure 1a. Partial derivatives of layer l, $\mathbf{G}_l^{conv}$, are fed to the preceding layer $l-1$ through the network. Inside layer l, the incoming gradient terms $\mathbf{G}_{l+1}^{conv}$ are passed through the pooling, non-linearity and convolution functions seqentially, backpropagating outputs $\mathbf{G}_l^p$, $\mathbf{G}_l^a$ and $\mathbf{G}_l^{conv}$ respectively; see Figure 1b. Along similar lines, the equations for linear layer are as follows:

$$\text{Linear transformation:} \qquad \mathbf{s}_l = \bar{\mathbf{W}}_l \bar{\mathbf{a}}_{l-1}^{\intercal} \tag{3a}$$

$$\text{Activation:} \qquad \mathbf{a}_l = \phi\left(\mathbf{s}_l\right) \tag{3b}$$

Eqs. 3a and 3b represent the linear transformation and activation function performed by the linear layer during the forward pass of the network. During the backward pass, we use $\mathbf{g}_{l-1}^{d}$ and $\mathbf{g}_{l}^{a}$ represent the gradient terms propagated backwards by the linear transformation and activation function within the linear layer.

The framework developed for gradient computations can also be adapted to compute the Hessian-vector products by using the "R-Operator" approach, first introduced in [41]. Specifically, the gradient and Hessian of a function, f, are related by:

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

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

$$\nabla^2 f(\boldsymbol{\theta})\mathbf{v} = \lim_{r \to 0} \frac{\nabla f(\boldsymbol{\theta} + r\mathbf{v}) - \nabla f(\boldsymbol{\theta})}{r} = \frac{\mathrm{d}}{\mathrm{d}r} \nabla f(\boldsymbol{\theta} + r\mathbf{v}) \bigg|_{r=0}. \tag{4}$$

Now, by defining

$$\mathcal{R}_{\mathbf{v}}\left\{\mathbf{u}(\boldsymbol{\theta})\right\} = \frac{\mathrm{d}}{\mathrm{d}r}\mathbf{u}(\boldsymbol{\theta} + r\mathbf{v})\bigg|_{r=0}, \tag{5}$$

for any vector valued function $\mathbf{u}$, we have $\nabla^2 f(\boldsymbol{\theta})\mathbf{v} = \mathcal{R}_{\mathbf{v}}\left\{\nabla f(\boldsymbol{\theta})\right\}$. Therefore, by applying $\mathcal{R}_{\mathbf{v}}$ to all the equations evaluated during the gradient computation of the given network, we can compute $\nabla^2 f(\boldsymbol{\theta})\mathbf{v}$ of the loss function associated with the given network. The forward and backward passes through the neural network associated with Hessian-vector computation are shown in the bottom half of Figures 1a and 1b.

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

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

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

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

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

3.1.1 Natural Gradient Computation

For completeness, we present an overview of the approximations involved in estimating the natural gradient direction (in the context of linear layers, but similar arguments can be made for convolution layer as well). We refer readers to [20, 35] for a detailed discussion on estimation of Fisher information matrix and approximations used in deriving the natural gradient direction.

We define

$$\begin{aligned} \mathcal{D}\boldsymbol{\theta} &\triangleq \frac{d\mathcal{L}(y, f(\mathbf{x}, \boldsymbol{\theta}))}{d\boldsymbol{\theta}} = -\frac{d \log p(y|\mathbf{x}, \boldsymbol{\theta})}{d\boldsymbol{\theta}} \\ \mathbf{g}_l^a &\triangleq \mathcal{D}\mathbf{s}_l, \end{aligned}$$

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

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}\left[\frac{\partial \log p(y|\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \left(\frac{\partial \log p(y|\mathbf{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^\intercal\right] = \mathbb{E}\left[\mathcal{D}\boldsymbol{\theta}\ (\mathcal{D}\boldsymbol{\theta})^\intercal\right]. \tag{6}$$

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

Kronecker-factored Fisher approximation(s) Grosse and Martens [20], Martens and Grosse [35] proposed to leverage the structure of the deep feedforward networks to approximate $\mathbf{F}^{-1}$ efficiently via Kronecker factors. We summarize these approximations in the following paragraphs for completeness. Let

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

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

$$\mathbf{F}_{i,j} = \mathbb{E}\left[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_i)\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_j)^\intercal\right] = \mathbb{E}\left[(\bar{\mathbf{a}}_{i-1} \otimes \mathbf{g}_i^a)(\bar{\mathbf{a}}_{j-1} \otimes \mathbf{g}_j^a)^\intercal\right] = \mathbb{E}\left[(\bar{\mathbf{a}}_{i-1}\bar{\mathbf{a}}_{j-1}^\intercal \otimes \mathbf{g}_i^a {\mathbf{g}_j^a}^\intercal)\right], \tag{7}$$

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

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

$$\mathbf{F}_{i,j} = \mathbb{E}\left[(\bar{\mathbf{a}}_{i-1}\bar{\mathbf{a}}_{j-1}^\intercal \otimes \mathbf{g}_i^a {\mathbf{g}_j^a}^\intercal\right] \approx \mathbb{E}\left[\bar{\mathbf{a}}_{i-1}\bar{\mathbf{a}}_{j-1}^\intercal\right] \otimes \mathbb{E}\left[\mathbf{g}_i^a {\mathbf{g}_j^a}^\intercal\right] = \mathbf{K}_{i-1,j-1} \otimes \mathbf{G}_{i,j}^a = \tilde{\mathbf{F}}, \tag{8}$$

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

This gives the following:

$$\tilde{\mathbf{F}} = \begin{bmatrix} \mathbf{K}_{0,0} \otimes \mathbf{G}_{1,1}^a & \mathbf{K}_{0,1} \otimes \mathbf{G}_{1,2}^a & \dots & \mathbf{K}_{0,\ell-1} \otimes \mathbf{G}_{1,\ell}^a \\ \mathbf{K}_{1,0} \otimes \mathbf{G}_{2,1}^a & \mathbf{K}_{1,1} \otimes \mathbf{G}_{2,2}^a & \dots & \mathbf{K}_{1,\ell-1} \otimes \mathbf{G}_{1,\ell}^a \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{K}_{\ell-1,0} \otimes \mathbf{G}_{\ell,1}^a & \mathbf{K}_{\ell-1,1} \otimes \mathbf{G}_{\ell,2}^a & \dots & \mathbf{K}_{\ell-1,\ell-1} \otimes \mathbf{G}_{\ell,\ell}^a \end{bmatrix}, \tag{9}$$

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

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

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

$$\breve{\mathbf{F}} = \operatorname{diag}\left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \ldots, \tilde{F}_{\ell-1,\ell-1}\right) = \operatorname{diag}\left(\mathbf{K}_{0,0} \otimes \mathbf{G}^{a}_{1,1}, \mathbf{K}_{1,1} \otimes \mathbf{G}^{a}_{2,2}, \ldots, \mathbf{K}_{\ell-1,\ell-1} \otimes \mathbf{G}^{a}_{\ell,\ell,}\right) \tag{10}$$

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

$$\breve{\mathbf{F}}^{-1} = \operatorname{diag}\left(\mathbf{K}^{-1}_{0,0} \otimes {\mathbf{G}^{a}_{1,1}}^{-1}, \mathbf{K}_{1,1} \otimes {\mathbf{G}^{a}_{2,2}}^{-1}, \ldots, \mathbf{K}_{\ell-1,\ell-1} \otimes {\mathbf{G}^{a}_{\ell,\ell}}^{-1}\right). \tag{11}$$

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

$$\mathbf{U}_i = {\mathbf{G}^{a}_{i,i}}^{-1} \mathbf{V}_i \mathbf{K}^{-1}_{i-1,i-1}, \tag{12}$$

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

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

$$\mathbb{E}\left[\operatorname{vec}\left(\mathcal{D}\bar{\mathbf{W}}_l\right) \operatorname{vec}\left(\mathcal{D}\bar{\mathbf{W}}_l\right)^{\intercal}\right] \approx \boldsymbol{\Psi}_{l-1} \otimes \boldsymbol{\Gamma}_l \triangleq \breve{\mathbf{F}}_l, \tag{13}$$

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

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

$$\breve{\mathbf{F}}^{-1} = \begin{bmatrix} \boldsymbol{\Psi}_0^{-1} \otimes \boldsymbol{\Gamma}_1^{-1} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \boldsymbol{\Psi}_{\ell-1}^{-1} \otimes \boldsymbol{\Gamma}_{\ell-1}^{-1} \end{bmatrix}. \tag{14}$$

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

$$\breve{\mathbf{F}}^{-1}\nabla h = \begin{bmatrix} \operatorname{vec}\ \left(\boldsymbol{\Gamma}_1^{-1}\left(\nabla_{\bar{\mathbf{W}}_1} h\right)\boldsymbol{\Psi}_0^{-1}\right) \\ \vdots \\ \operatorname{vec}\ \left(\boldsymbol{\Gamma}_\ell^{-1}\left(\nabla_{\bar{\mathbf{W}}_\ell} h\right)\boldsymbol{\Psi}_{\ell-1}^{-1}\right) \end{bmatrix}. \tag{15}$$

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

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

$$\breve{\mathbf{F}}_\ell + (\lambda + \gamma)\,\mathbf{I} \approx \left(\mathbf{\Psi}_{\ell-1} + \pi_\ell\sqrt{\lambda+\gamma}\mathbf{I}\right) \otimes \left(\mathbf{\Gamma}_\ell + \frac{1}{\pi_\ell}\sqrt{\lambda+\gamma}\mathbf{I}\right), \tag{16}$$

for some π_ℓ.

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

3.2 Algorithm

Algorithm 1 describes FITRE, a realization of our proposed method in trust-region settings. First, the natural gradient direction, $\mathbf{p}_t$ is computed and used in determining the step-size. This is done by using the quadratic approximation of the objective function at $\mathbf{p}_t$, whose closed form solution is $(\Delta / \|\mathbf{H}_t\mathbf{p}_t + \mathbf{g}_t\|)\,(\mathbf{H}_t\mathbf{p}_t + \mathbf{g}_t)$. (Note that gradient, $\mathbf{g}_t$, can also be used to estimate the step size, and it may yield a better descent direction in some cases). Once the step-size, η, is determined, ρ_t is computed over the same mini-batch to determine the trust-region radius as well as the iterate update. These steps are repeated until desired generalization is achieved. Note that we can compare the efficiency of the natural gradient direction with that of the standard gradient and use the appropriate one at each iteration. This is referred to as “KFAC + gradient” in this algorithm.

4 Experiments

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

Hardware and Software Platform for Experiments. All our simulations are executed on NVIDIA’s Tesla V100 GPUs configured with 16GB global RAM on CUDA 9.0 runtime platform. These machines are configured with Intel Xeon CPUs with 192 GB of RAM. Our code is implemented in C++, and we primarily used *cublas* for GEMM operations. SGD results are executed on pyTorch 1.0.0 installation with python 3.1 as the front-end. The code base is available for download at [28].

Datasets. We use CIFAR10, CIFAR100 [1], and tiny ImageNet [2] datasets for validating our KFAC-STR method, using several CNNs mentioned below. The details of each of these datasets are described in Table 2. CIFAR10 is a well conditioned dataset, whereas ImageNet is the largest dataset with which we experiment. CIFAR100 is conditioned between CIFAR10 (relatively well conditioned) and ImageNet (ill-conditioned).

Algorithm 1: FITRE Method

```
Input      :
             - Starting point $\mathbf{x}_0$
             - Initial trust-region radius: $0 < \Delta_0 < \infty$
             - KFAC parameters: damping parameter ($\gamma \geq 0$), moving average ($0 < \boldsymbol{\theta} < 1$)
Result: $\mathbf{x}_t$ - direction to be used to update model parameters.
foreach $t = 0, 1, \ldots$ do
    Set the approximate gradient $\mathbf{g}_t$ and Hessian $\mathbf{H}_t$
    /* Compute the approximated Inverse Fisher × gradient, a.k.a
       natural-gradient                                                   */
    Obtain natural-gradient direction $\mathbf{r}_t$, as described in [20, 35]
    Case 1: KFAC
        $\eta_t = \arg\min_{\|\eta\mathbf{r}_t\| \leq \Delta_t} m(\eta\mathbf{p}_t) = \eta\mathbf{g}_t^\intercal\mathbf{r}_t + \frac{\eta^2}{2}\mathbf{r}_t^\intercal\mathbf{H}_t\mathbf{r}_t$
        $\mathbf{s}_t = \eta_t\mathbf{r}_t$
    Case 2: KFAC + Gradient
        $\eta_t = \arg\min_{\|\eta\mathbf{r}_t\| \leq \Delta_t} m(\eta\mathbf{p}_t) = \eta\mathbf{g}_t^\intercal\mathbf{r}_t + \frac{\eta^2}{2}\mathbf{r}_t^\intercal\mathbf{H}_t\mathbf{r}_t$
        $\alpha_t = \arg\min_{\|\alpha\mathbf{g}_t\| \leq \Delta_t} m(\eta\mathbf{g}_t) = \alpha\mathbf{g}_t^T\mathbf{g}_t + \frac{\alpha^2}{2}\mathbf{g}_t^T\mathbf{H}_t\mathbf{g}_t$
        $\mathbf{s}_t = \arg\min_{\mathbf{s} \in \{\eta_t\mathbf{r}_t, \alpha_t\mathbf{g}_t\}} m(\mathbf{s})$
    Set $\rho_t \triangleq \frac{h_t(\boldsymbol{\theta}_t) - h_t(\boldsymbol{\theta}_t + \mathbf{s}_t)}{-m(\mathbf{s}_t)}$, ($h_t(.)$ are evaluated on the same mini-batch as $\mathbf{g}_t$ and $\mathbf{H}_t$ ).
    if $\rho_t \geq 0.75$ then
        $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{s}_t$ and $\Delta_{t+1} = min\{2\Delta_t, \Delta_{max}\}$
    end
    else if $\rho_t \geq 0.25$ then
        $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{s}_t$ and $\Delta_{t+1} = \Delta_t$
    end
    else
        $\mathbf{w}_{t+1} = \mathbf{w}_t$ and $\Delta_{t+1} = \Delta_t/2$
    end
end
```

Table 2: Description of the datasets used in our experiments

Dataset	Features	Number of Training Samples	Number of Testing Samples
CIFAR-10	3096	50,000	10,000
CIFAR-100	3096	50,000	10,000
Tiny ImageNet	12288	100,000	9,000

Hyperparameter Tuning. Our proposed method has two easily tunable parameters—the damping parameter (γ) and maximum trust-region radius (δ). All experiments presented here use $\gamma \in \{1e^{-3}, 1e^{-2}, 1e^{-1}, 1e^{0}, 1e^{1}, 1e^{2}\}$ and $\delta \in \{1e^{-2}, 1e^{-1}, 1e^{0}, 1e^{1}, 1e^{2}\}$. Regularization term (λ) is set to the following values $\{1e^{-4}, 1e^{-5}, 1e^{-6}\}$. Trust-region radius, Δ, is capped by δ. It is doubled when ρ, the ratio of observed model reduction and expected model reduction, is ≥ 0.75; and it is halved when it is < 0.25; otherwise it remains the same. SGD uses learning rate as the hyper-parameter, which is in the following range $\{1e^{-6}, 1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}, 1e^{0}, 1e^{1}, 1e^{2}\}$. Both methods use the Nesterov-accelerated momentum term as 0.9. All the experiments are run for 50 epochs except for Imagenet dataset, where maximum number of epochs is set to 25. Mini-batch size is set to 200 for both the methods.

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

Convolutional Neural Networks. We experiment with VGG11, VGG16, and VGG19 CNN's for our validation purposes. Each of these CNN's architectures is described in Table 3.

Table 3: Various Convolution Neural Networks used in our experiments. α is 512 when CIFAR10 and CIFAR100 are used; and it is 2048 for Imagenet. β is 10 for CIFAR10, 100 for CIFAR100, and 200 for ImageNet. These networks can be easily adapted for embedding BatchNormalization layers (typically after the convolution layer).

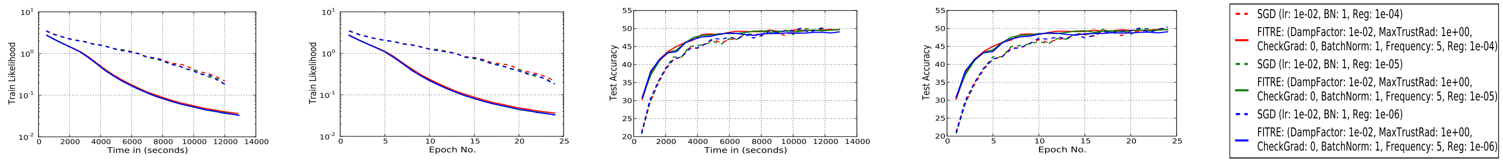
CNN	Layer Description	
VGG11	Conv(3, 64), Swish, MaxPool	conv_kernel(k=3,s=1,p=1), pool_kernel(k=2,s=2)
	Conv(64, 128), Swish, MaxPool	
	Conv(128, 256), Swish	
	Conv(256, 256), Swish, MaxPool	
	Conv(256, 512), Swish	
	Conv(512, 512), Swish, MaxPool	
	Conv(512, 512), Swish	
	Conv(512, 512), Swish, MaxPool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

Table 4: Comparison of VGG11 using ImageNet dataset

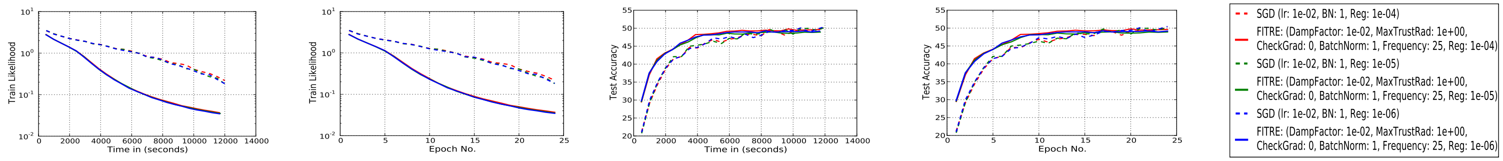
	Time vs. Likelihood	Epoch vs. Likelihood	Time vs Test Accuracy	Epoch vs. Test Accuracy

Default

KFAC Update Freq = 5

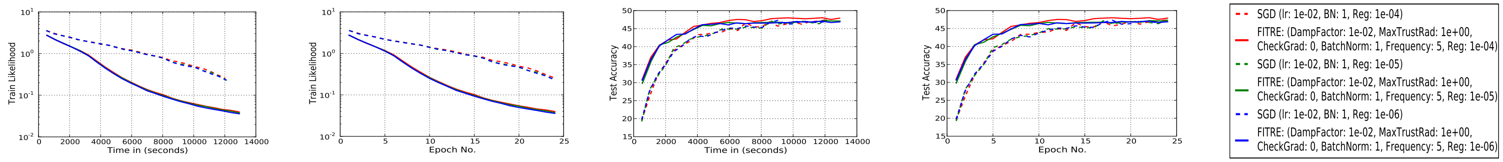


KFAC Update Freq = 25



Kaiming

KFAC Update Freq = 5



KFAC Update Freq = 25

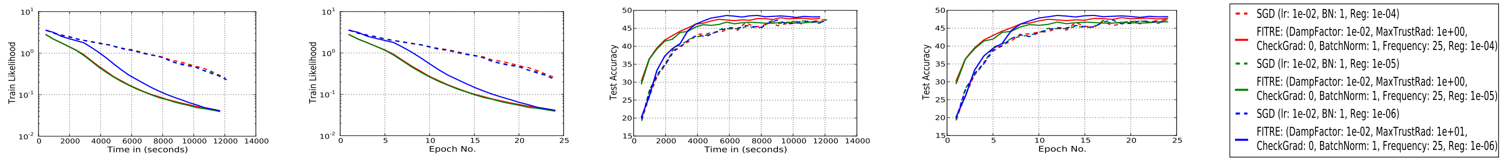
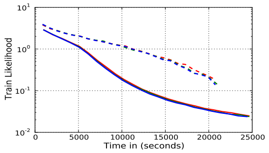
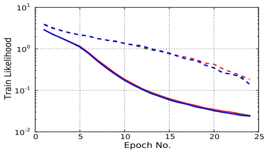
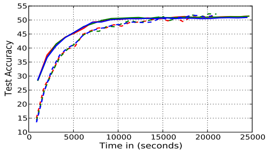
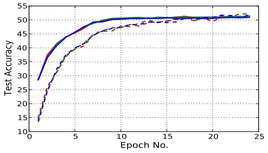
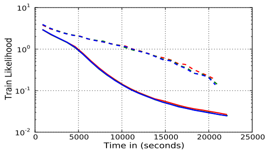
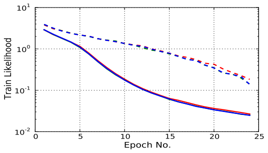
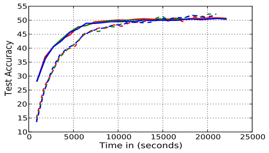
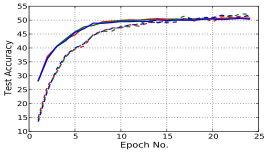
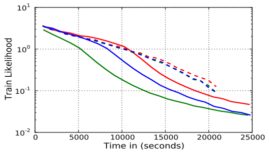
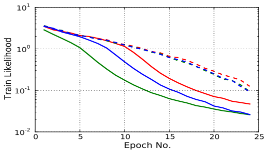
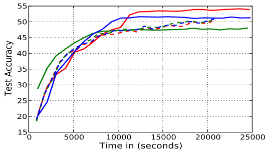
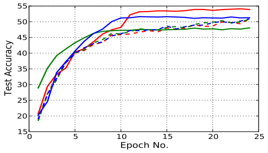
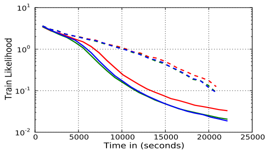
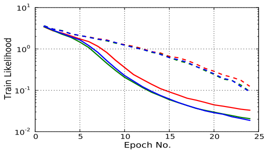
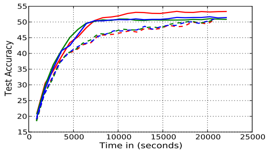
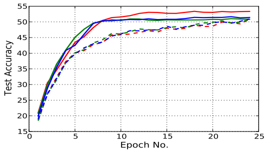


Table 5: Comparison of VGG16 using ImageNet dataset

		Time vs. Train Loss	Epoch vs. Train Loss	Time vs Test Accuracy	Epoch vs. Test Accuracy	
Default	KFAC Update Freq = 5					- - SGD (lr: 1e-02, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)
	KFAC Update Freq = 25					- - SGD (lr: 1e-02, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-04) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-05) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-06)
Kaiming	KFAC Update Freq = 5					- - SGD (lr: 1e-02, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)
	KFAC Update Freq = 25					- - SGD (lr: 1e-02, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-04) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-05) - - SGD (lr: 1e-02, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-02, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 25, Reg: 1e-06)

Results on the Imagenet Dataset. Tables 4 and 5 show the plots for the Imagenet dataset using VGG11 and VGG16 CNNs, respectively. In these tables, we show the test errors plotted against wall-clock time and against number of epochs in Columns 3 and 4, respectively, and we show the training loss against wall-clock time and against number of epochs in Columns 1 and 2, respectively. KFAC update frequency is set to 5 (mini-batches) for the first row; and it is set to 25 for the second row. All plots (in the first two rows) use *Default* initialization, as defined in pyTorch, which is a uniform distribution, in these two tables. Corresponding results using *Kaiming* initialization [21] (on a high-level, this initialization is based on random Gaussian distribution) are shown in Rows 3 and 4 of both the tables.

The following conclusions can be made from the plots in Tables 4 and 5: (i) FITRE method minimizes the likelihood function to a significantly smaller value, compared to well-tuned SGD, and at any given wall-clock instance (FITRE method yields better NLL value, compared to SGD), (ii) Kaiming initialization yields superior generalization errors, compared to Default initialization of the CNNs, (iii) contrary to expectations, KFAC update frequency of 25 yields better generalization errors, relative to more frequent updates, (iv) with increasing network complexity, VGG16 compared to VGG11, FITRE method yields significantly better generalization errors compared to SGD, showcasing its superior scaling characteristics compared to SGD; and (v) Default initialization is relatively immune to ℓ_2 regularization compared to Kaiming initialization.

For VGG16 network with Kaiming initialization and KFAC update frequency of 25 we observe that to attain 50% test accuracy FITRE (with 1e-6 regularization) takes ≈ 6500 seconds compared to ≈ 20500 seconds for SGD (for all regularizations used); a speedup of 3.2 over SGD. Furthermore, when regularization is set to $1e^{-4}$, FITRE achieves 53.5% test accuracy whereas SGD fails to obtain similar accuracy. Similar arguments can be made for the VGG11 network as well. This shows that even though FITRE is computationally more expensive on a per-iteration basis, it yields significantly better results in shorter time, compared to SGD. This can be attributed to better descent direction (SGD's gradient versus FITRE 's natural gradient) and an adaptive second-order approximated learning rate computation within the trust-region framework used by the FITRE method. Contrary to expectations, we notice that for VGG11 CNN and Default initialization, FITRE 's execution of 50 epochs takes less time compared to SGD for KFAC update frequency 25. FITRE makes two passes over the network (one forward and backward pass for gradient computation and another pass for Hessian-vector product computation used to compute the learning rate in the trust-region framework). One would expect that SGD is at least twice as fast as FITRE on the wall-clock time (one a per-iteration basis). We note that SGD's pyTorch implementation uses *auto-differentiation* to compute the gradient of the given network, whereas our implementation of the FITRE method is R-operator based (as proposed by Perlmutter [41]). At a finer level, we note from our previous experience with the pyTorch platform [18, 29], that memory management on the GPU is not efficient. For example, pyTorch allocates and frees memory very often, and it tends to persist very little information on the device. Even though FITRE makes two passes over the network and computes inverses of smaller matrices at each layer of the network (for computing the inverse of the KFAC block matrices) our implementation persists relevant information on the GPU memory. Coupled with our efficient implementation of the R-operator based Hessian-vector product, we can significantly reduce the computation cost associated with each mini-batch. In addition, our proposed method is a true *stochastic online method* in which there is no dependence on any part of the dataset other than the current mini-batch during its entire execution time, compared to state-of-the-art existing second-order methods [36, 39].

We notice that Default initialization is immune to regularization for both networks (VGG11 and VGG16), and for both methods (FITRE and SGD). These two methods show negligible changes in training loss values (as well as generalization errors), while the FITRE method yields superior results compared with SGD for significant part of the execution. At the end of the execution, SGD tends to achieve similar generalization errors compared to FITRE , but on minimizing the training objective function (i.e. trainling loss) FITRE always achieves superior results. However, when using Kaiming initialization, based on random Gaussian distribution, for both the networks, we notice that regularization helps in achieving superior generalization errors for FITRE method (with VGG11 network, KFAC update frequency set to 25 and regularization of $1e^{-6}$) compared to SGD. In all cases, FITRE method yields superior results when the underlying model does not use any regularization. Compared to the FITRE method, SGD is relatively immune to Kaiming initialization as well, as shown in plots in columns 1 and 2 of Tables 4 and 5. Notice that there is very little change in objective function value throughout the simulations.

KFAC update frequency is a hyperparameter used to control the frequency with which the block matrix inverses are computed at each layer of the network. These block inverses are used to compute the natural gradient direction eventually for each mini-batch. Since these blocks approximate the *Fisher matrix* of the loss function, they are updated once every few mini-batches. Martens et. al. [20, 35] argues that more frequent updates of these block inverses makes them too rigid and may lead to overfitting. Using larger values for this update frequency has the effect of a regularizer on the underlying model, and it helps in avoiding overfitting. As an added advantage, this dependence of the FITRE method reduces its computation cost (note also that the computation of block inverses can be delegated to slave processing units, if available, further reducing the computation cost thereby decreasing the time for processing each mini-batch). This is also one of the reasons why our proposed method scales well with increasing network complexity. We note that for VGG16 (with Kaiming initialization), a larger and more complex network compared to VGG11, FITRE method yields superior generalization errors as well as minimizing objective function compared to SGD.

Table 6: Comparison of VGG11 using cifar100 dataset

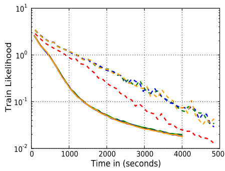
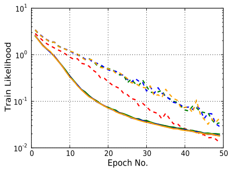
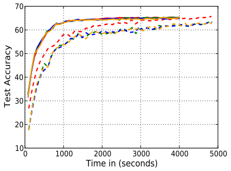
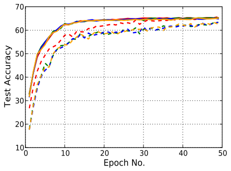
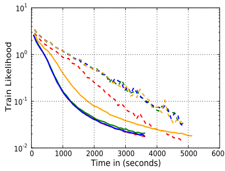
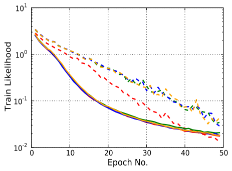
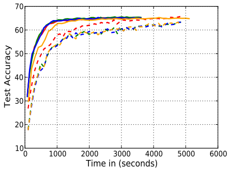
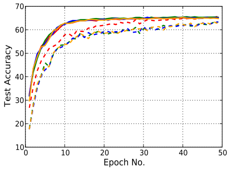
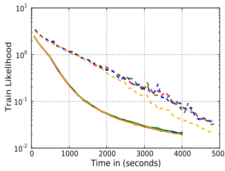
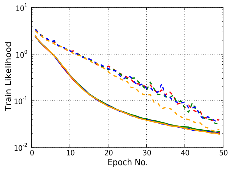
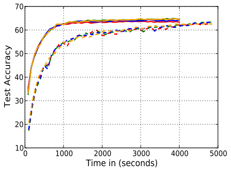
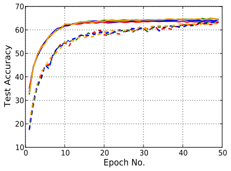
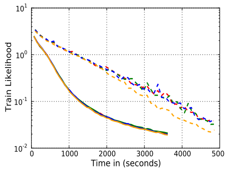
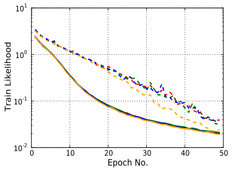
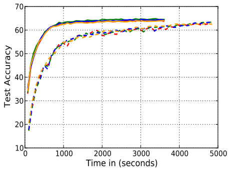
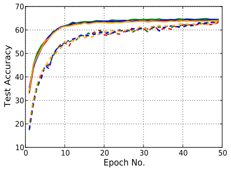
<table>
<tr><th></th><th>Time vs. Train Loss</th><th>Epoch vs. Train Loss</th><th>Time vs Test Accuracy</th><th>Epoch vs. Test Accuracy</th><th></th></tr>
<tr><td rowspan="4">Default</td><td colspan="5">KFAC Update Freq = 5</td></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
<td>
</td>
<td>- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-05)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)</td>
</tr>
<tr><td colspan="5">KFAC Update Freq = 25</td></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
<td>
</td>
<td>- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-05)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 1, BatchNorm: 1, Frequency: 10, Reg: 1e-06)</td>
</tr>
<tr><td rowspan="4">Kaiming</td><td colspan="5">KFAC Update Freq = 5</td></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
<td>
</td>
<td>- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05)
- - SGD (lr: 1e-02, BN: 1, Reg: 1e-06)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)</td>
</tr>
<tr><td colspan="5">KFAC Update Freq = 10</td></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
<td>
</td>
<td>- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04)
- - SGD (lr: 1e-01, BN: 1, Reg: 1e-04)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05)
- - SGD (lr: 1e-02, BN: 1, Reg: 1e-06)
— FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-06)</td>
</tr>
</table>

Table 7: Comparison of VGG16 using cifar100 dataset

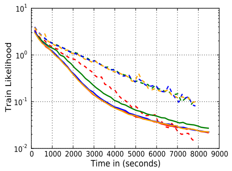
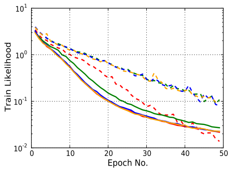
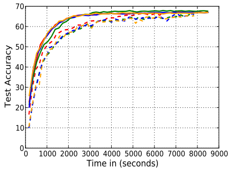
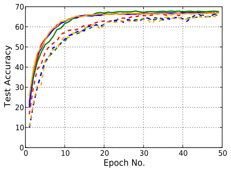
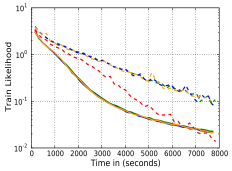
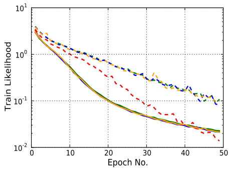
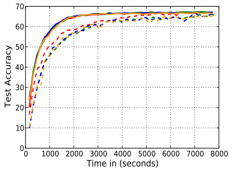
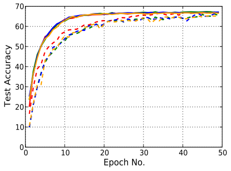
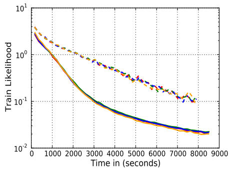
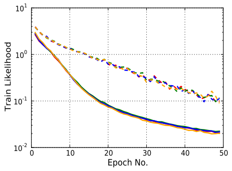
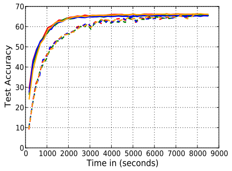
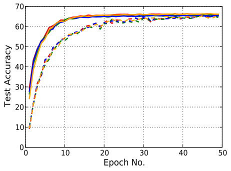
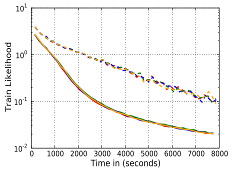
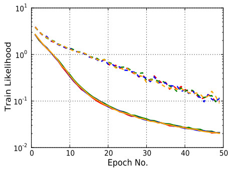
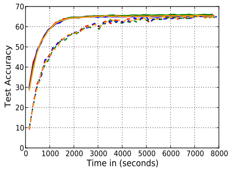
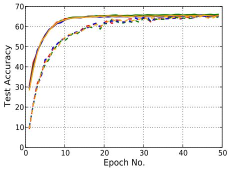
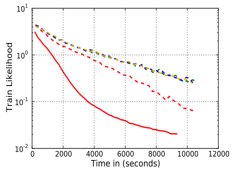
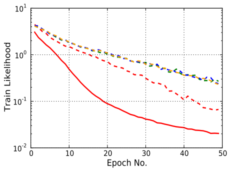
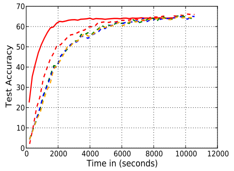
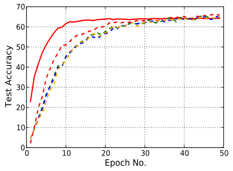
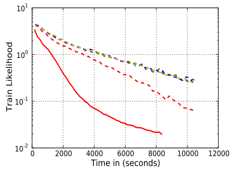
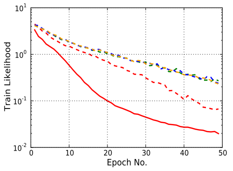
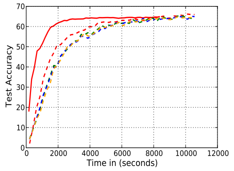
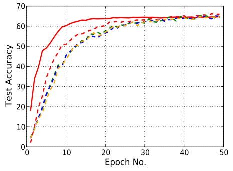
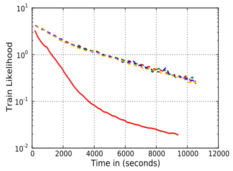
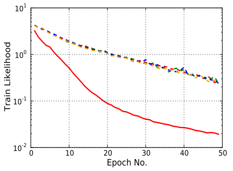
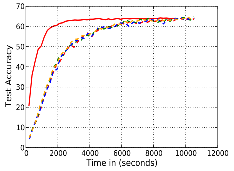
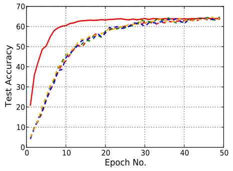
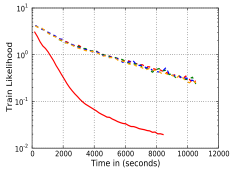
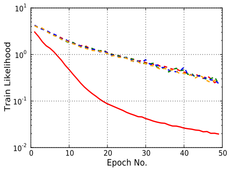
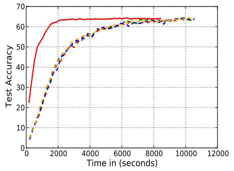
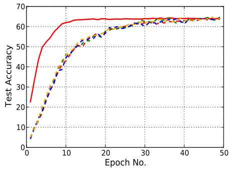
	Time vs. Train Loss	Epoch vs. Train Loss	Time vs Test Accuracy	Epoch vs. Test Accuracy	
	KFAC Update Freq = 5				
Default					- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e+00, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+02, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)
	KFAC Update Freq = 25				
					- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+02, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+02, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-05) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+02, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-06)
	KFAC Update Freq = 5				
Kaiming					- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+02, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 1e-06)
	KFAC Update Freq = 10				
					- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 1e-06)

Table 8: Comparison of VGG19 using cifar100 dataset

	Time vs. Train Loss	Epoch vs. Train Loss	Time vs Test Accuracy	Epoch vs. Test Accuracy	
Default	KFAC Update Freq = 5				
					- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)
	KFAC Update Freq = 25				
					- - SGD (lr: 1e-02, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+01, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-05) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)
Kaiming	KFAC Update Freq = 5				
					- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 5, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)
	KFAC Update Freq = 10				
					- - SGD (lr: 1e-01, BN: 1, Reg: 0e+00) — FITRE: (DampFactor: 1e-01, MaxTrustRad: 1e+00, CheckGrad: 0, BatchNorm: 1, Frequency: 10, Reg: 0e+00) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-04) - - SGD (lr: 1e-01, BN: 1, Reg: 1e-06)

CIFAR100 Dataset Results Table 6 shows plots for VGG11 network using the CIFAR100 dataset. We show the test errors plots versus wall clock time and versus number of epochs in Columns 3 and 4 respectively; and we show training loss versus wall-clock time and versus number of epochs in Columns 1 and 2, respectively. Note that the first row uses the KFAC update frequency of 5 (mini-batches), and the second row uses the KFAC update frequency of 10. All of these plots (in the first two rows) use *Default* initialization for the respective CNNs. Corresponding results using *Kaiming* initialization are shown in Rows 3 and 4. Tables 7 and 8 show plots for VGG16 and VGG19 CNNs.

From the VGG19 networks results shown in Table 8, we clearly notice that the use of ℓ_2-regularization adversely affects the behavior of SGD optimizer when Default initialization is used. We notice that the behavior of SGD optimizer, in the objective function and generalization error plots, without any regularization, yields superior results, compared to SGD optimizer using non-zero regularization terms. However, for Kaiming initialization, we do not see any noticeable changes in the behavior of SGD, with and without using any regularization terms. From the generalization error plots, we see that the FITRE method achieves $\approx 5\times$ speedup over SGD in the case of Kaiming initialization (irrespective of the KFAC update frequency), and the corresponding speedup in the case of Default initialization is $\approx 4\times$. However, notice that in the training loss plots for both types of initialization, FITRE method achieves significantly better results compared to SGD (an order of magnitude better). Furthermore, as seen in the results for the Imagenet dataset, we see that higher KFAC update frequency lowers the time for processing the mini-batches, as well as the time per epoch. Notice that the simulation completion time for FITRE method is lower for KFAC update frequency of 10, compared to the other frequency irrespective of the type of initialization used by the networks. Similar to VGG19 results, ℓ_2 regularization plays similar role in the behavior of VGG16, as can be seen in Table 7 for both types of initialization. The FITRE method achieves a speedup of $\approx 4\times$ to $\approx 5\times$ over SGD for this network. Table 6 shows the results for the VGG11. Contrary to results from the other two networks we notice that for VGG11 the objective function values of both methods are closer, indicating that SGD optimizer yields results similar to that of FITRE method at the end of the simulation as seen in columns 1 and 2. However note that FITRE method achieves superior results in the first few epochs, irrespective of the KFAC update frequency and network initialization type, compared to SGD, and as simulation progress SGD tends to achieve similar results at those of FITRE at the end of the simulations.

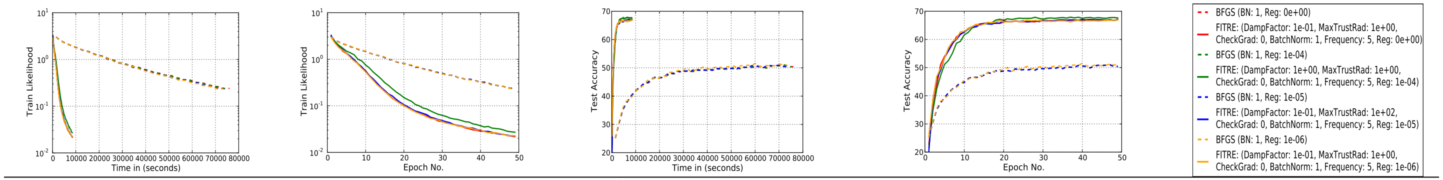
Remark 1 *The quality of natural gradient descent direction is effective, and second-order approximated trust-region constrained optimization yields larger step sizes at the beginning of the execution. Aided by these two factors, the FITRE method produces better parameter updates in the same amount of wall-clock time compared to SGD, and it achieves significantly better generalization errors in the first few epochs. The FITRE method only needs a few epochs to attain near saturation results compared to SGD which needs a larger number of epochs to match the results of FITRE . Efficient implementation and effective use of GPU resources establishes our FITRE method, a truly second-order method, as a suitable alternative to widely used first-order methods like SGD.*

Table 9: Comparison of VGG16 using cifar100 dataset with a quasi-Newton Method (L-BFGS).

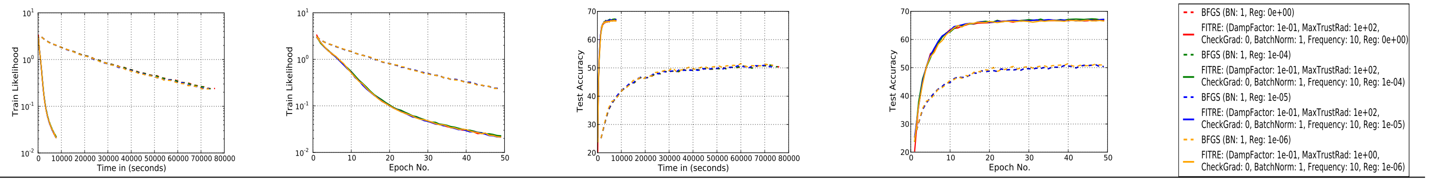
Time vs. Train Loss | Epoch vs. Train Loss | Time vs Test Accuracy | Epoch vs. Test Accuracy

Default

KFAC Update Freq = 5

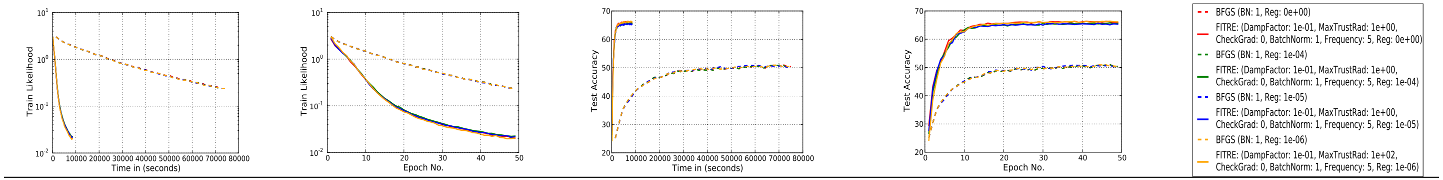


KFAC Update Freq = 10

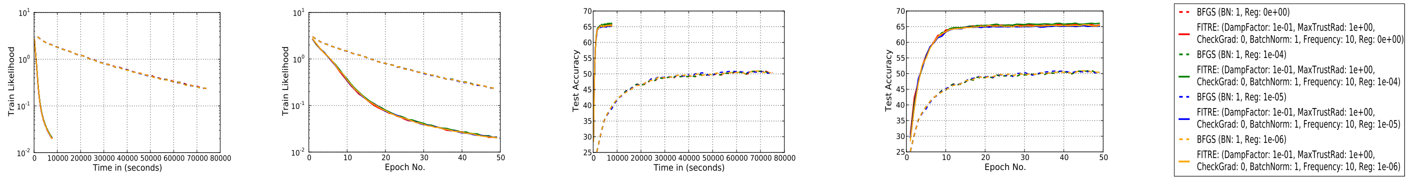


Kaiming

KFAC Update Freq = 5



KFAC Update Freq = 10



Comparison with quasi-Newton Method (L-BFGS) In this paragraph, we elaborate on the comparison of FITRE method against L-BFGS, a widely known quasi-Newton method. Our BFGS implementation is in pyTorch, and it uses a history of 20 prior gradients in generating the Hessian approximation used in computing the descent direction. For step size estimation we use cubic interpolation method as described in Section 3.4 of the book of Nocedal and Wright [40]. In the plots shown in Table 9, we describe the comparison results between FITRE and L-BFGS methods. For FITRE method, we keep the KFAC update frequency and regularization constant, and we select the best performing set of hyperparameters, yielding the highest test accuracy for comparison against the BFGS method.

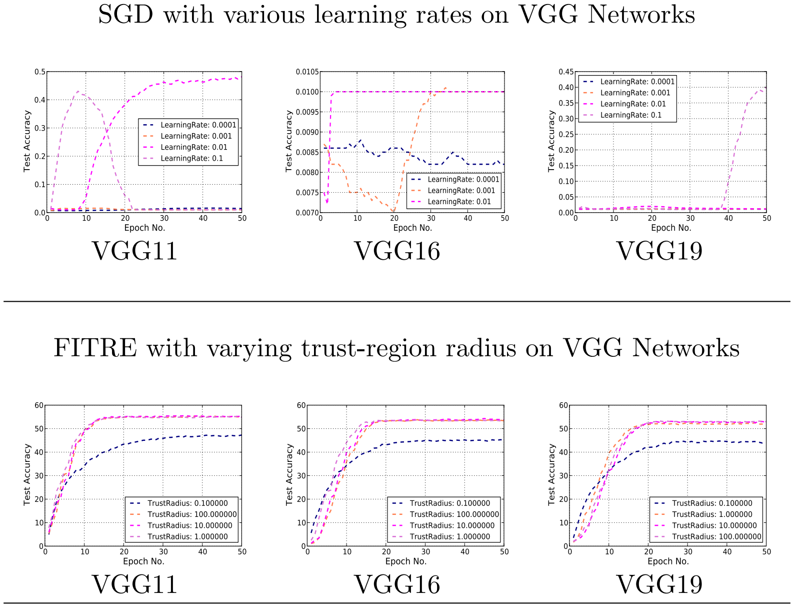
From the plots in Table 9, we clearly notice that FITRE method is orders of magnitude faster, compared to BFGS method, $\approx 8\times$ faster in almost all cases. This is because FITRE uses our CUDA framework while BFGS method is implemented in pyTorch. Both these methods use GPUs for computation, but in a significantly contrasting manner. pyTorch is a general purpose framework which is optimized for multi-user multi-process environment which inherently optimizes the GPU device usage for simultaneous use GPUs for many users. In that process, all the operations on GPUs are executed in a transactional manner. This means that whenever GPU computation is deemed necessary all the associated data is moved on to the GPU device and computation is initiated and once completed the results are moved back to the CPU memory space. Because of this reason, which is repeated hundreds of times over the course of the simulation, we notice a significant deterioration with the pyTorch version of BFGS solver (resulting in $8\times$ slower compared to FITRE method). In addition, FITRE method uses better tuning of thread block size for individual CUDA kernels, highly optimized implementation of convolution, activation and pooling functions along with their first- and second-derivatives which are used during the computation of Fisher information statistics as well as Hessian-vector products.

In terms of test accuracy, we clearly notice that (irrespective of parameter initialization method and KFAC update frequency) FITRE method achieves significantly superior test accuracy compared to BFGS method. This can attributed to the quality of natural gradient as estimated by the Kronecker approximated Fisher information matrix as well as the crude estimation of the Hessian used in estimation the descent direction during the optimization of the objective function. Apart from the above mentioned differences between BFGS and FITRE , we also notice that regularization term does not play a significant role in either improving the generalization error or time consumed during the course of the simulation for either of the optimizers.

Robustness Results. The FITRE method's two main hyperparameters—damping term and maximum trust-region radius—can be easily estimated. Typical values for trust-region radius are between 1e-1 and 10. Similarly, damping parameter's well behaved region is between 1e-2 and 10. For most of the combinations, the FITRE method behaves in a consistent manner, and can yield almost identical results, as shown in This behavior contributes significantly to the resilient behavior of the FITRE method w.r.t to minor changes in the hyperparameters Table 10. However, we notice that SGD's behavior changes significantly with slight changes in learning rate. This is one of the major challenges that a developer runs into with first-order methods, namely hyperparameter space that is difficult to tune, which we do not encounter with our FITRE method.

Invariance to Re-parameterization. With the computation of block inverses and ultimately the natural gradient itself, expectation of the inputs and outputs of the convolution layer is deeply embedded into the KFAC approximation itself. Because of this, additional batch normalizations are irrelevant to the FITRE method's performance in minimizing the objective function. However, first-order methods like SGD need to have batch normalization layers embedded into the underlying model itself (typically right after the convolution layer). These results are presented in Table 11. We notice that without the use of batch normalization layers, the SGD method tends to diverge or does not make significant progress in optimizing the likelihood of the underlying model. However, the

Table 10: Behavior of FITRE and SGD without regularization on CIFAR100 dataset. FITRE method uses an update frequency of 5 and "KFAC + gradient" option is turned off in these set of simulations. VGG networks in this table does not use batch-normalization function.



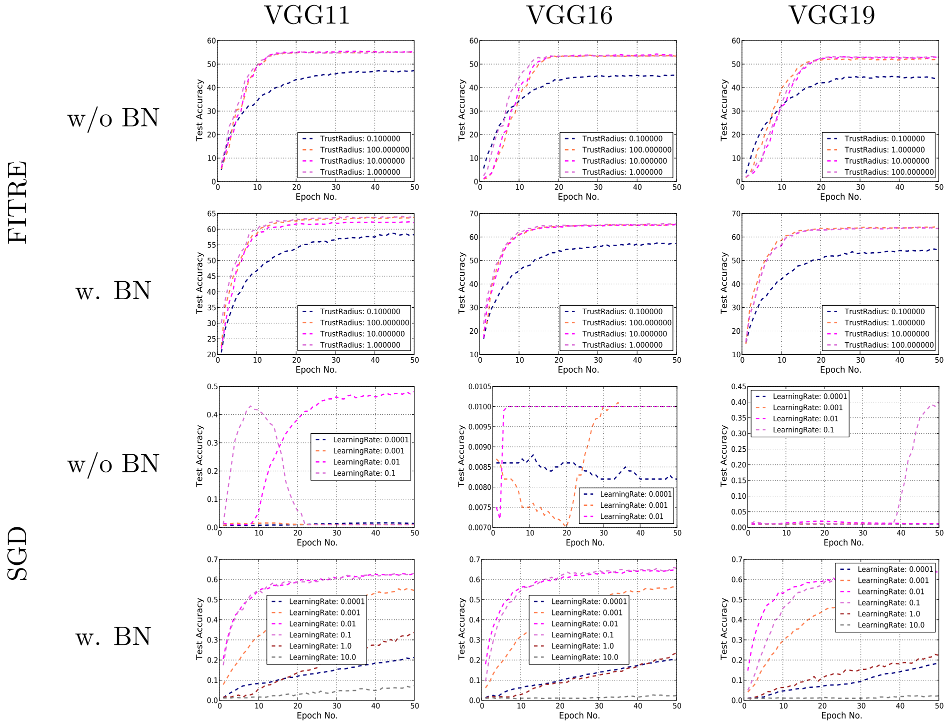
SGD with various learning rates on VGG Networks

VGG11 VGG16 VGG19

FITRE with varying trust-region radius on VGG Networks

VGG11 VGG16 VGG19

Table 11: Reparameterization Invariance results for SGD and FITRE methods. VGG networks in these experiments use 0 regularization. And for FITRE method we use the KFAC update frequency of 5 and use only the natural gradient direction as the descent direction (KFAC + gradient option is not used in these experiments).



FITRE method is robust, insensitive to small changes in its hyperparameters, as well as invariant to re-parameterization. Typically, implementation of batch normalization is computationally expensive, and as the network complexity increases, its execution time contributes significantly to the over all simulation time. There is no need for such layers in case of the FITRE method, which significantly helps reduce the processing time for each mini batch (and simulation time for the network).

5 Conclusions and Future Work

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

References

[1] Cifar datasets. https://www.cs.toronto.edu/ kriz/cifar.html, 10 2019. URL https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Tiny imagenet dataset. https://tiny-imagenet.herokuapp.com/, October 2019. URL https://tiny-imagenet.herokuapp.com/.

[3] Zeyuan Allen-Zhu and Yuanzhi Li. Neon2: Finding local minima via first-order oracles. In *Advances in Neural Information Processing Systems*, pages 3720–3730, 2018.

[4] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. *ICLR*, 2017.

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[6] Coralia Cartis, Nicholas IM Gould, and Ph L Toint. On the complexity of steepest descent, Newton's and regularized Newton's methods for nonconvex unconstrained optimization problems. *Siam journal on optimization*, 20(6):2833–2852, 2010.

[7] Coralia Cartis, Nicholas IM Gould, and Philippe L Toint. Adaptive cubic regularisation methods for unconstrained optimization. Part I: motivation, convergence and numerical results. *Mathematical Programming*, 127(2):245–295, 2011.

[8] Coralia Cartis, Nicholas IM Gould, and Philippe L Toint. Adaptive cubic regularisation methods for unconstrained optimization. Part II: worst-case function-and derivative-evaluation complexity. *Mathematical programming*, 130(2):295–319, 2011.

[9] Oliver Chapelle and Dumitru Erhan. Improved preconditioner for hessian free optimization. In *In NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[10] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust region methods*. MPS SIAM, 2000.

[11] Andrew R Conn, Nicholas IM Gould, and Ph L Toint. *Trust region methods*, volume 1. SIAM, 2000.

[12] Frank E Curtis, Daniel P Robinson, and Mohammadreza Samadi. A Trust Region Algorithm with a Worst-Case Iteration Complexity of $\mathcal{O}(\epsilon^{-3/2})$ for Nonconvex Optimization. *Mathematical Programming*, 162(1-2):1–32, 2017.

[13] Yann Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 1504–1512, 2015.

[14] Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv:1406.2572v1*, 2014.

[15] 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.

[16] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* John Wiley & Sons, 2001.

[17] Thierry Dumas, Aline Roumy, and Christine Guillemot. Autoencoder based image compression: Can the learning be quantization independent? *arXiv preprint arXiv:1802.0937*, 2018.

[18] Chih-Hao Fang, Sudhir B Kylasa, Farbod Roosta-Khorasani, Michael W Mahoney, and Ananth Grama. Distributed second-order convex optimization. *arXiv preprint arXiv:1807.07132*, 2018.

[19] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *AISTATS, JMLR*, 2010.

[20] Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. *arXiv:1602.01407v2*, 2016.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[23] G. Hinton. Neural networks for machine learning. *Coursera, video lectures. 307*, 2012.

[24] Shun ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(251-276), 1988.

[25] Shun ichi Amari, Ryo Karakida, and Masafumi Oyizumi. Fisher information and natural gradient learning of random deep networks. *arXiv preprint: 1808.07172v1*, 2018.

[26] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. How to escape saddle points efficiently. *arXiv preprint arXiv:1703.00887*, 2017.

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

[28] Sudhir B Kylasa. Cuda non-convex framework for fitre optimizer. https://github.com/kylasa/NewtonCG, 10 2019.

[29] Sudhir B Kylasa, Fred Roosta Khorasani, Michael W. Mahoney, and Ananth Y Grama. Gpu accelerated sub-sampled newton's method for convex classification problems. In SIAM, editor, *Proceedings of the 2019 SIAM International Conference on Data Mining*, pages 702–710. SIAM, 2019.

[30] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klas-Robert Muller. Efficient backprop. *Neural Networks: tricks of the trade*, 1998.

[31] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[32] Ziujun Lu, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent reinforcement learning: A hybrid approach. *arXiv preprint arXiv:1509.03044*, 2015.

[33] James Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[34] James Martens. New insights and perspectives on the natural gradient method. *arXiv preprint: arXiv:1412.1193v9*, 2017.

[35] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. *arXiv:1503.05671v6*, 2016.

[36] Gregoire Montavon, Genevieve B. Orr, and Klas-Robert Muller. *Neural Networks: Tricks of the Trade*. Springer, 2nd edition, September 2012.

[37] W Ross Morrow. Hessian-free methods for checking the second-order sufficient conditions in equality-constrained optimization and equilibrium problems. *arXiv preprint arXiv:1106.0898*, 2011.

[38] Yurii Nesterov. *Introductory lectures on convex optimization*, volume 87. Springer Science & Business Media, 2004.

[39] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

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

[41] Barak A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 1993.

[42] pyTorch. Default initialization pytorch. https://pytorch.org/docs/stable/_modules /torch/nn/modules/linear.html. URL `https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html`.

[43] Yongming Rao, Jiwen Lu, and Jie Zhou. Attention-aware deep reinforcement learning for video face recognition. *The IEEE International Conference on Computer Vision (ICCV)*, pages 3931–3940, 2017.

[44] Clément W Royer, Michael O'Neill, and Stephen J Wright. A newton-cg algorithm with complexity guarantees for smooth unconstrained optimization. *Mathematical Programming*, pages 1–38, 2019.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2015.

[46] 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.

[47] Dehao Wu, Maziar Nekovee, and Yue Wang. Deep learning based autoencoder for interference channel. *arXiv preprint arXiv:1902.06841*, 2019.

[48] Peng Xu, Fred Roosta, and Michael W Mahoney. Newton-type methods for non-convex optimization under inexact hessian information. *Mathematical Programming*, pages 1–36. doi: 10.1007/s10107-019-01405-z.

[49] 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.

[50] Yi Xu, Jing Rong, and Tianbao Yang. First-order stochastic algorithms for escaping from saddle points in almost linear time. In *Advances in Neural Information Processing Systems*, 2018.

[51] Howard Hua Yang and Shun ichi Amari. The efficiency and the robustness of natural gradient descent learning rule. In *Neural Information Processing Systems*, pages 385–391, 1997.

[52] Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. PyHessian: Neural networks through the lens of the Hessian. Technical report, 2019. Preprint: arXiv:1912.07145.

[53] Zhewei Yao, Peng Xu, Farbod Roosta-Khorasani, and Michael W Mahoney. Inexact non-convex Newton-type methods. *arXiv preprint arXiv:1802.06925*, 2018.

[54] Yaodong Yu, Pan Xu, and Quanquan Gu. Third-order smoothness helps: Even faster stochastic optimization algorithms for finding local minima. *arXiv:1712.06585v1*, 2017.

[55] Yaodong Yu, Difan Zou, and Quanquan Gu. Saving gradient and negative curvature computations: Finding local minima more efficiently. *arXiv:1712.03950v1*, 2017.

[56] Huishai Zhang, Caiming Xiong, James Bradbury, and Richard Socher. Block-diagonal hessian-free optimization for training neural networks. *arXiv preprint: arXiv:1712.07296v1*, 2017.

A Neural Network Operations

A.1 Notation

Description	Symbol
Loss function	$\mathcal{L}$
Total number of layers in the neural network	ℓ layers numbered as $\ell(1\ldots\ell)$
Input to the convolution function of layer l	$\mathbf{A}_{l-1}$
Mini-batch input to the convolution function of layer l (“˜” is used to indicate mini-batch processing of a given matrix)	$\tilde{\mathbf{A}}_{l-1}$
Input to the activation function of layer l	$\mathbf{C}_l$
Input to the pooling function of layer l	$\mathbf{P}_l$
Output of layer l (and input to the layer $l+1$)	$\mathbf{A}_l$
Output of the neural network	$\mathbf{A}_{out}$
Output of the loss function	$\mathbf{A}_{loss}$
Gradient terms propagated from the loss function	$\mathbf{G}_{loss}\ (= \mathcal{D}\mathbf{A}_{out})$
Gradient terms propagated from the pooling function of layer l	$\mathbf{G}_l^p\ (= \mathcal{D}\mathbf{P}_l)$
Gradient terms propagated from the activation function of layer l	$\mathbf{G}_l^a\ (= \mathcal{D}\mathbf{C}_l)$
Gradient terms propagated from the convolution function of layer l	$\mathbf{G}_l^{conv}\ (= \mathcal{D}\mathbf{A}_l)$
Input to convolution function of layer l of the neural network	$\mathcal{R}_v\left\{\mathbf{A}_l\right\}$
Input to activation function of layer l of the neural network	$\mathcal{R}_v\left\{\mathbf{C}_l\right\}$
Input to pooling function of layer l of the neural network	$\mathcal{R}_v\left\{\mathbf{P}_l\right\}$
Output of the neural network	$\mathcal{R}_v\left\{\mathbf{a}_{out}\right\}$
$\mathcal{R}_v\left\{\right\}$ terms propagated from the loss function	$\mathcal{R}_v\left\{\mathbf{G}_{loss}\right\}$
$\mathcal{R}_v\left\{\right\}$ terms propagated from the pooling function of layer l	$\mathcal{R}_v\left\{\mathbf{G}_l^p\right\}$
$\mathcal{R}_v\left\{\right\}$ terms propagated from the activation function of layer l	$\mathcal{R}_v\left\{\mathbf{G}_l^a\right\}$
$\mathcal{R}_v\left\{\right\}$ terms propagated from the convolution function of layer l	$\mathcal{R}_v\left\{\mathbf{G}_l^{conv}\right\}$

A.2 Gradient computation

A.2.1 Convolution Function

Convolution function is associated with a set of filters (or weights) $\mathbf{W}_l$, and bias, $\mathbf{b}_l$, variables which are learned during the minimization of the loss function associated with the neural network. Typically these variables are initialized with suitable values as defined by the various schemes [19, 21, 42] discussed in current literature. These initialization schemes take into account the mean and the variance of the input and output data of a convolution function, and they attempt to maintain them as constants throughout the neural network. Some initialization methods like [21] also take into account the type of non-linearity stacked after the convolution function whereas [19] approximate the non-linearity functions to be constants.

Forward Pass. Forward pass through a convolution function is the process of overlaying set of filter maps on top of the input data and summing up the resultant element-wise products, typically known as convolution function. This is repeated over the entire image. This operation can be treated as a GEMM operation by enlarging the input data so that each row (or column) represents a particular filter map and the number of rows (or columns) indicating the number of times a filter map is applied on the incoming sample point per channel. Note that bias term can be folded into the weights variable by suitably altering the weights and input data matrices. Mathematically, convolution function can be represented as follows:

$$\mathbf{C}_l = [\![\mathbf{A}_{l-1}]\!]_H \bar{\mathbf{W}}_l.$$

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

$$\mathcal{D}\bar{\mathbf{W}}_l = \mathbf{G}_l^{a\intercal} [\![\mathbf{A}_{l-1}]\!]_H$$
$$\mathcal{D}\mathbf{A}_{l-1} = [\![\mathbf{G}_l^a]\!] \breve{\mathbf{W}}_l = \mathbf{G}_l^{conv}.$$

A.2.2 Activation Function

Activation functions, for instance swish, sigmoid, log-softmax etc., are used after the convolution function to transform the input data in a non-linear fashion, so as to differentiate (or classify) among the output's components. After the inputs to the neural network are passed through sufficient non-linearity functions, depending on the design of the network, outputs of the network are used to compute the class probabilities. These are used to make predictions of the associated input sample point to the network. Non-linearity functions are employed to aid this classification process.

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

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

$$\mathbf{a}_l = \phi\left(\mathbf{s}_l\right).$$

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

$$\mathbf{g}_l^a = \phi^{'}\left(\mathbf{s}_l\right) \odot \mathbf{g}_l^p.$$

A.2.3 Pooling Function

pooling functions are used to down-sample the inputs so as to reduce the input size. These functions are associated with *Stride* (s), *Padding*(p) and *kernel size* (k), which are used by the down-sampling procedure when computing the output of the pooling function. The height and width of the output of the pooling functions are related to its inputs in the same way as defined for the *convolution* function as shown in Eq. 19.

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

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

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

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

A.2.4 Linear (Dense) Function

Apart from *convolution* function, *linear or dense* functions are also used to transform the inputs by performing a weighted summation and a bias term to move the center of the input data. Typically in the context of convolution neural networks linear functions are used for classification purposes at the end of the network. This function is associated with $\mathbf{W}_l$ and $\mathbf{b}_l$ parameters whose dimensions are $f_l^{out} \times f_l^{in}$ and $f_l^{out} \times 1$ respectively, where subscript l indicates the layer number and superscripts in, out indicate the number of input features and output features associated with the layer l.

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

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

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

$$\begin{aligned} \mathcal{D}\bar{\mathbf{W}}_l &= \mathbf{g}_l^a \bar{\mathbf{a}}_{l-1}^{\intercal}, \\ \mathcal{D}\mathbf{a}_{l-1} &= \mathbf{g}_l^d = \mathbf{W}_l{}^{\intercal} \mathbf{g}_l^a. \end{aligned}$$

A.3 Hessian Vector Operation on Neural Networks

Similar to gradient computation on the neural network, Hessian-vector computation requires two additional passes over the given network. During the forward pass, we store the required data in the auxiliary variable, $\mathcal{R}_v\left\{\mathbf{a}_l\right\}$ (and $\mathcal{R}_v\left\{\mathbf{A}_l\right\}$) , for each layer l. During the backward pass, we use the variable $\mathcal{R}_v\left\{\mathbf{a}_l\right\}$ during forward pass as well as $\mathbf{a}_l$ and $\mathbf{G}_l$ stored during the gradient computation on the network to compute the Hessian-vector product w.r.t to a given vector.

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

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

A.3.1 Linear (Dense) Function

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

Forward Pass In the forward pass of the **Hv**-computation, we apply the $\mathcal{R}_v\left\{.\right\}$ to the equation used in the gradient computation as shown below. Note that $\mathcal{R}_v\left\{.\right\}$ operator obeys the multiplication-rules of the derivative calculus. Using this we can easily compute the $\mathcal{R}_v\left\{\mathbf{s}_l\right\}$ as shown below:

$$\begin{aligned}\mathcal{R}_v\left\{\mathbf{s}_l\right\} &= \mathcal{R}_v\left\{\bar{\mathbf{W}}_l\bar{\mathbf{a}}_l^{\mathsf{T}}\right\}\\ &= \mathcal{R}_v\left\{\bar{\mathbf{W}}_l\right\}\bar{\mathbf{a}}_l^{\mathsf{T}} + \bar{\mathbf{W}}_l\mathcal{R}_v\left\{\bar{\mathbf{a}}_l\right\}^{\mathsf{T}}.\end{aligned}$$

Here, $\mathcal{R}_v\left\{\bar{\mathbf{W}}_l\right\}$ is the $\bar{\mathbf{W}}_l$ component in vector $\mathbf{v}$ and $\mathcal{R}_v\left\{\bar{\mathbf{a}}_l\right\}$ would have already computed when previous layers were processed.

Backward Pass. Here we apply the $\mathcal{R}_v\left\{.\right\}$ operator to the equations used to compute $\mathbf{g}_l^d$ and $\mathcal{D}\bar{\mathbf{W}}_l$ to generate $\mathcal{R}_v\left\{\mathbf{g}_l^d\right\}$ and $\mathcal{R}_v\left\{\mathcal{D}\bar{\mathbf{W}}_l\right\}$. The former term is back-propagated to the previous layers and the latter, $\mathcal{R}_v\left\{\mathcal{D}\bar{\mathbf{W}}_l\right\}$, is the **Hv** component for the parameters associated with the linear layer, i.e., $\bar{\mathbf{W}}_l$.

$$\begin{aligned}\mathcal{R}_v\left\{\mathbf{g}_l^d\right\} &= \mathcal{R}_v\left\{{\mathbf{W}_l}^{\mathsf{T}}\mathbf{g}_l^a\right\}\\ &= \mathcal{R}_v\left\{{\mathbf{W}_l}^{\mathsf{T}}\right\}\mathbf{g}_l^a + {\mathbf{W}_l}^{\mathsf{T}}\mathcal{R}_v\left\{\mathbf{g}_l^a\right\},\\ \mathcal{R}_v\left\{\mathcal{D}\bar{\mathbf{W}}_l\right\} &= \mathcal{R}_v\left\{\mathbf{g}_l^a\bar{\mathbf{a}}_{l-1}^{\mathsf{T}}\right\}\\ &= \mathcal{R}_v\left\{\mathbf{g}_l^a\right\}\bar{\mathbf{a}}_{l-1}^{\mathsf{T}} + \mathbf{g}_l^a\mathcal{R}_v\left\{\bar{\mathbf{a}}_{l-1}\right\}^{\mathsf{T}}.\end{aligned}$$

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

A.3.2 Convolution Function

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

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

$$\begin{aligned}\mathbf{C}_l &= [\![\mathbf{A}_{l-1}]\!]_H\bar{\mathbf{W}}_l,\\ \mathcal{R}_v\left\{\mathbf{C}_l\right\} &= [\![\mathcal{R}_v\left\{\mathbf{A}_{l-1}\right\}]\!]_H\bar{\mathbf{W}}_l + [\![\mathbf{A}_{l-1}]\!]_H\mathcal{R}_v\left\{\bar{\mathbf{W}}_l\right\}.\end{aligned}$$

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

$$\begin{aligned}
\mathcal{R}_v\left\{\mathcal{D}\bar{\mathbf{W}}_l\right\} &= \mathcal{R}_v\left\{[\mathbf{G}_l^a]^\intercal \llbracket \mathbf{A}_{l-1} \rrbracket_H\right\} \\
&= \mathcal{R}_v\left\{\mathbf{G}_l^a\right\}^\intercal \llbracket \mathbf{A}_{l-1} \rrbracket_H + [\mathbf{G}_l^a]^\intercal \llbracket \mathcal{R}_v\left\{\mathbf{A}_{l-1}\right\} \rrbracket_H, \\
\mathcal{R}_v\left\{\mathbf{G}_l^{conv}\right\} &= \mathcal{R}_v\left\{\llbracket \mathbf{G}_l^a \rrbracket^\intercal \breve{\mathbf{W}}_l\right\} \\
&= \llbracket \mathcal{R}_v\left\{\mathbf{G}_l^a\right\} \rrbracket^\intercal \breve{\mathbf{W}}_l + \llbracket \mathbf{G}_l^a \rrbracket^\intercal \mathcal{R}_v\left\{\breve{\mathbf{W}}_l\right\}.
\end{aligned}$$

A.3.3 Activation Function

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

Forward Pass.

$$\begin{aligned}
\mathcal{R}_v\left\{\mathbf{P}_l\right\} &= \mathcal{R}_v\left\{\phi\left(\mathbf{C}_l\right)\right\} \\
&= \phi^{'}\left(\mathbf{C}_l\right) \odot \mathcal{R}_v\left\{\mathbf{C}_l\right\}.
\end{aligned}$$

Backward Pass.

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

A.3.4 Pooling Function

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

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

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

Backward Pass. $\mathcal{R}_v\left\{\mathbf{G}_l^p\right\}$ is computed during the backward pass as shown below:

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

(Note that for both types of pooling functions supported in our implementation, namely max-pooling and average-pooling, the second derivative of the pooling function does not exist thus simplifying the evaluation of $\mathcal{R}_v\left\{\mathbf{G}_l^p\right\}$.)

A.4 Loss Functions

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

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

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

Gradient Backward Pass. In the backward pass, the error terms, used in the *back-propagation*, are computed as shown below:

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

For the case of a single sample i and j indices indicate the position of the component in the output of the network, $\mathbf{a}_{out}$.

The general equation for the error terms propagated through the network is given by the following equation (where indices i, j and k indicate the position in the output vector for each sample point, and note that here y is assumed to be one-hot encoded[2]):

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

Hessian-vec Backward Pass.

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

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

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

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

$$\mathcal{R}_v\left\{\mathcal{D}\mathbf{a}_{out}\right\}_i = \mathbf{q}_i\mathcal{R}_v\left\{\mathbf{a}_{out}\right\}_i - \mathbf{q}_i\sum_j \mathbf{q}_j\mathcal{R}_v\left\{\mathbf{a}_{out}\right\}_j.$$

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

A.5 Activation Functions

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

Sigmoid Function.

$$\begin{aligned} \phi(x) &= \frac{1}{1+e^{-x}}, \\ \phi^{'}(x) &= \phi(x)\,\phi(-x)\,, \\ \phi^{''}(x) &= \phi^{'}(x)\,\phi(-x) - \phi(x)\,\phi^{'}(-x)\,. \end{aligned}$$

log-Softmax Function.

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

Swish Function.

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

A.6 Pooling Functions

A.6.1 Average pooling function

Gradient Forward Pass. During the forward pass for the gradient computation, the input indices, (i, j), and output indices, $(i^{'}, j^{'})$ are linked through the Eq. 19. Average pooling function is itself defined by the eq 18 shown below. Here we describe how each cell of the output, $\mathbf{A}_{l+1}$, is computed using the input, $\mathbf{P}_{l}$:

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

Gradient Backward Pass. During the backward pass of the back-propagation algorithm for the Average pooling function the source indices ($i^{'}, j^{'}$) and the destination indices (i, j) are related by the Eq. 19. Also note that the dimensions of the matrices (inputs and outputs) may change because of the down sampling functionality of the pooling function. The back-propagation of the gradient terms for the Average pooling function is defined by the Eq. 18:

$$[\mathbf{G}_l^p]_{i,j} = \begin{cases} \frac{1}{k^2}[\mathbf{G}_{l+1}^{conv}]_{i'j'} & \text{if the } (l+1)^{th} \text{ layer is a convolution function} \\ \frac{1}{k^2}[\mathbf{g}_{l+1}^{d}]_{i'j'} & \text{if the } (l+1)^{th} \text{ layer is a linear function;} \\ & \text{assuming that vectors are suitably converted to matrices} \end{cases} \tag{18}$$

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

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

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

Hessian-vec Backward Pass.

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

A.6.2 Max pooling function

For the forward pass of gradient computation, this function computes the maximum value from the input, $\mathbf{P}_l$, for each filter map as defined by the stride, padding and kernel size parameters. During the back-propagation, the derivative of the max pooling function is computed w.r.t to the input data, $\mathbf{P}_l$. Hence (i,j) location in the output matrix $\mathbf{G}_l^p$ is set to $[\mathbf{G}_{l+1}^{conv}]_{i'j'}$. Note that the locations (i,j) and $(i^{'},j^{'})$ are related by Eq. 19. Similar to average pooling, we develop the equations to be used for gradient and Hessian-vector product computation below:

Gradient Forward Pass.

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

Gradient Backward Pass.

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

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

Hessian-vec Forward Pass.

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

Hessian-vec Backward Pass.

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

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

A.7 Batch Normalization

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

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

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

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

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

Gradient Backward Pass. By using the chain rule of the derivatives, we can express the gradient terms of the batch normalization function as shown in Eq. 19.

$$\begin{aligned} [\tilde{\mathbf{G}}_l^{bn}]_{c'} = [\mathcal{D}\tilde{\mathbf{C}}_l]_{c'} &= [\mathcal{D}\tilde{\mathbf{B}}_l]_{c'}\left[\frac{d\tilde{\mathbf{B}}_l}{d\tilde{\mathbf{C}}_l}\right]_{c'} + \mathcal{D}\boldsymbol{\mu}_{c'}\frac{d\boldsymbol{\mu}_{c'}}{d[\tilde{\mathbf{C}}_l]_{c'}} + \mathcal{D}\boldsymbol{\sigma}_{c'}^2\frac{d\boldsymbol{\sigma}_{c'}^2}{d[\tilde{\mathbf{C}}_l]_{c'}} \\ &= [\tilde{\mathbf{G}}_l^a]_{c'}\left[\frac{d\tilde{\mathbf{B}}_l}{d\tilde{\mathbf{C}}_l}\right]_{c'} + \mathcal{D}\boldsymbol{\mu}_{c'}\frac{d\boldsymbol{\mu}_{c'}}{d[\tilde{\mathbf{C}}_l]_{c'}} + \mathcal{D}\boldsymbol{\sigma}_{c'}^2\frac{d\boldsymbol{\sigma}_{c'}^2}{d[\tilde{\mathbf{C}}_l]_{c'}}. \end{aligned}$$

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

- Using Eq. 19 we can easily compute $[\frac{d\tilde{\mathbf{B}}_l}{d\tilde{\mathbf{C}}_l}]_{c'}$ as:

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

- Using chain-rule from differential calculus, we have the following:

$$\begin{aligned}\mathcal{D}\boldsymbol{\mu}_{c'}\frac{d\boldsymbol{\mu}_{c'}}{d[\tilde{\mathbf{C}}_l]_{c'}} &= [\mathcal{D}\tilde{\mathbf{B}}_l]_{c'}\frac{d[\tilde{\mathbf{B}}_l]_{c'}}{d\boldsymbol{\mu}_{c'}}\frac{d\boldsymbol{\mu}_{c'}}{d[\tilde{\mathbf{C}}_l]_{c'}} \\ &= [\tilde{\mathbf{G}}^a_l]_{c'}\frac{-1}{\sqrt{\boldsymbol{\sigma}^2_{c'} + \epsilon}}\left[\frac{1}{m}\right] \\ &= -\frac{[\tilde{\mathbf{G}}^a_l]_{c'}}{m\sqrt{\boldsymbol{\sigma}^2_{c'} + \epsilon}}.\end{aligned}$$

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

$$\begin{aligned}\mathcal{D}\boldsymbol{\sigma}^2_{c'} &= \sum \mathcal{D}[\tilde{\mathbf{B}}_l]_{c'}\frac{d[\tilde{\mathbf{B}}_l]_{c'}}{d\boldsymbol{\sigma}^2_{c'}}, \\ \frac{d[\tilde{\mathbf{B}}_l]_{c'}}{d\boldsymbol{\sigma}^2_{c'}} &= \left[[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}\right]\frac{1}{[\boldsymbol{\sigma}^2_{c'} + \epsilon]^{1.5}}\frac{-1}{2}, \\ \mathcal{D}\boldsymbol{\sigma}^2_{c'} &= -\sum\frac{[\tilde{\mathbf{G}}^a_l]_{c'}}{2}\frac{[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}}{[\boldsymbol{\sigma}^2_{c'} + \epsilon]^{1.5}}.\end{aligned}$$

Using the above equations, we can write the following:

$$\mathcal{D}\boldsymbol{\sigma}^2_{c'}\frac{d\boldsymbol{\sigma}^2_{c'}}{d[\tilde{\mathbf{C}}_l]_{c'}} = -\frac{1}{m}\left[\sum[\tilde{\mathbf{G}}^a_l]_{c'}\frac{[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}}{[\boldsymbol{\sigma}^2_{c'} + \epsilon]^{1.5}}\right]\left[\sum\left[[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}\right]\right].$$

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

$$[\mathcal{D}\tilde{\mathbf{C}}_l]_{c'} = [\tilde{\mathbf{G}}^{bn}_l]_{c'} = \frac{[\tilde{\mathbf{G}}^a_l]_{c'}}{\sqrt{\boldsymbol{\sigma}^2_{c'} + \epsilon}} - \frac{[\tilde{\mathbf{G}}^a_l]_{c'}}{m\sqrt{\boldsymbol{\sigma}^2_{c'} + \epsilon}} - \frac{1}{m}\left[\sum[\tilde{\mathbf{G}}^a_l]_{c'}\frac{[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}}{[\boldsymbol{\sigma}^2_{c'} + \epsilon]^{1.5}}\right]\left[\sum\left[[\tilde{\mathbf{C}}_l]_{c'} - \boldsymbol{\mu}_{c'}\right]\right].$$

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

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

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

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

A.8 Some Implementation Details

A.8.1 Memory Layout

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

A.8.2 Img2Col Function

A typical *convolution* operation can either be computed by overlaying filter maps (or weights) on top of the input data and computing the output *in a serial fashion* or by converting the input data into a suitable form (typically an enlarged matrix) and performing an efficient matrix multiplication, *i..e* GEMM operation, with the filter (weights) matrix. Using the former results in redundant transfer of data among memory layers of the GPU device resulting in inefficient use of GPU device coupled with poor locality of reference(note that the same $k \times k$ filter map may be transferred from CPU to GPU multiple times). Using the latter (because of the enlarged matrix) results in redundant storage of the input matrix. For our work, we use the latter approach for performing the convolution operation and we use GEMM operations as defined in the ***cublas*** library for this purpose. GEMM operations are highly efficient and bandwidth optimized for large matrices.

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

$$h^{out} = \frac{h^{in} + 2p - s}{k} + 1. \tag{19}$$

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

$$\mathbf{C}_l = \mathbf{Img2Col}\left(\mathbf{A_{l-1}}\right)\mathbf{W_l} + \mathbf{b_l}. \tag{20}$$

In Eq. 20 **Img2Col** (.) operator is used to enlarge the input matrix, $\mathbf{A}_{l-1}$, so that the resultant matrix can be used in a GEMM operation with $\mathbf{W}_l$ matrix for the convolution function. We use $[\![\mathbf{A}_{l-1}]\!]$ to refer to the enlarged matrix. Bias term, $\mathbf{b}_l$ can be folded into the $\mathbf{W}_l$ by appending it as a row vector. Note that $\mathbf{W}_l$ is treated as a 2-dimensional matrix of shape $\left(c_l^{in} \times k_l \times k_l\right) \times c_l^{out}$ and the result of folding the bias vector will result into $\bar{\mathbf{W}}_l$ of shape $\left(c_l^{in} \times k_l \times k_l + 1\right) \times c_l^{out}$. We also append a column of ones, $\mathbf{e}$, to the enlarged matrix, $[\![\mathbf{A}_{l-1}]\!]$, and the resulting matrix is denoted by $[\![\mathbf{A}_{l-1}]\!]_H$ (the use of the subscript H, homogenous coordinate, indicates that a column of ones, $\mathbf{e}$, has been appended).With this change, a concise form of the convolution operation is given by the following equation:

$$\mathbf{C}_l = [\![\mathbf{A}_{l-1}]\!]_H \bar{\mathbf{W}}_l.$$