Efficient Second-Order Stochastic Methods for Machine Learning, and in particular, Deep Neural Networks

Don Goldfarb

Department of Industrial Engineering and Operations Research
Columbia University

January 2023
Co-Researchers

Yi Ren (Columbia University)  Achraf Bahamou (Columbia University)
Relevant Papers

K-BFGS:

TNT:
▶ Y. Ren, D. Goldfarb. *Tensor normal training for deep learning models*

MBF:

Subsampled GN and NG methods:
Outline

Introduction

K-BFGS: Kronecker-factored Quasi-Newton Methods

TNT: A Generic Approximate Natural Gradient Method

MBF: A Mini-block Fisher method for Deep Neural Networks (DNNs)

Subsampled Gauss-Newton and Natural Gradient Methods
Introduction
Challenges

- When the BFGS method was developed, unconstrained optimization methods were typically tested on Rosenbrock’s 2-D function, a 3-D steep-sided helical valley function and simple trigonometric functions with up to at most 100 variables.

- Current DNNs have huge numbers of variables:
  - AlexNet: 62 million trainable parameters
  - VGG16: 138 million trainable parameters
  - ResNet34: 63.5 million trainable parameters, ChatGPT Transformer: 175 billion trainable parameters (estimated),

- These DNNs, have both fully-connected feed-forward and convolutional layers. Recent AI craze, ChatGPT, is trained on a Transformer DNN, with an estimated 175 billion parameters.

- Sizes of data sets:
  - MNIST: 60K images for training; 10K for testing, 784 classes
  - FACES: 103K images for training; 62K for testing, 625 classes
  - CIFAR-10: 50K images for training, 10K for testing, 10 classes
  - CIFAR-100: 50K images for training, 10K for testing, 100 classes
  - ImageNet: 14M images classified into 20K classes:
    - Research subset: ~1.28M images for training, 50K for testing, 1,000 classes.
Feed-forward Fully-connected Neural Network

Input Layer

\[ x_1 \quad [W_1]_{11} \quad h_{11} \quad [W_2]_{11} \quad h_{21} \quad [W_3]_{11} \]

\[ x_2 \quad h_{12} \quad h_{22} \quad p_1 \]

\[ x_3 \quad h_{13} \quad h_{23} \quad p_2 \]

\[ x_4 \quad h_{14} \quad h_{24} \quad p_3 \]

\[ x_5 \quad [W_1]_{54} \]

Hidden Layers

Output Layer

Note: \( h_{l,i} = (W_l a_{l-1})_i \) is the \( i \)-th component of the pre-activation output of the \( l \)-th layer. Activation by \( \phi \) yields the input \( (a_l)_i = \phi(h_{l,i}) \) to the \( i \)-th node in layer \( l + 1 \).

Given a matrix of labeled training data \( (a_0, y) \), training a FF-FC DNN is the optimization problem:

\[
\inf_{\{W_\ell\}, \{a_\ell\}} \mathcal{L}(W_L a_{L1}, y), \quad \text{where} \quad \mathcal{L} \quad \text{is the loss function.}
\]
CNNs: One Convolutional (Conv) Layer

- Input image: 3 channels (red, blue, green)
- Each kernel is a \( k \times k \times 3 \) tensor consisting of \( k \times k \) filters from the 3 input channels to one of the output channels; the full transformation is a \( k \times k \times 3 \times 4 \) tensor.
- A Conv layer can be viewed as a FC-FF layer, where nodes correspond to channels and each arc represents a vector of weights corresponding to the relevant filter.
Optimization Methods for Deep Learning (DL)

First-order methods:
- **Stochastic gradient descent (SGD)** [Robbins and Monro, 1951]
  - With momentum
- **Adaptive learning rate methods:**
  - AdaGrad (Duchi et al. [2011]),
  - RMSprop (Hinton et al. [2012]),
  - Adam (Kingma and Ba [2014]), and its many variants,
  - etc

However,
- These methods use no (or very little) curvature information
- Can experience slow convergence
Optimization Methods for Deep Learning (DL)

Second-order methods:
- Newton’s method and its variants
- Quasi-Newton (QN) methods:
  - Limited-memory BFGS (L-BFGS) [Liu and Nocedal, 1989]
- Recent methods that deal with stochasticity and nonconvexity:
  - Sub-sampled Newton [Xu et al., 2019],
  - SQN [Byrd et al., 2016, Wang et al., 2017],
  - Stochastic block BFGS [Gower et al., 2016],
  - etc

However,
- These methods are **not** specifically designed for training DNNs
- Results in high memory and/or per-iteration time complexity
Optimization Methods for Deep Learning (DL)

Second-order methods that take into account the property/structure of DNNs:

- **Natural gradient (NG) methods** [Amari et al., 2000]
  - Use Fisher (information) matrix as a pre-conditioning matrix

- **Hessian-free method** [Martens, 2010]
  - Conjugate gradient method with Gauss-Newton matrix or Fisher matrix

- **KFAC** [Martens and Grosse, 2015]
  - Kronecker-factored approximation to the Fisher matrix

- **Shampoo** [Gupta et al., 2018]
  - Empirical Fisher matrix, extension from the adaptive learning rate methods

- **K-BFGS**

- **TNT**

- **MBF**

- **Subsampled GN and NG methods**
Pre-conditioned SGD Framework

Suppose we are trying to solve $\min_{\theta} L(\theta)$:

- (Stochastic) gradient descent:

$$\theta = \theta - \alpha \nabla_\theta L$$  \hspace{1cm} (1)

where $\alpha$ is the learning rate (step size).

- Pre-conditioned SGD:

$$\theta = \theta - \alpha H \nabla_\theta L,$$  \hspace{1cm} (2)

where $H$ is the pre-conditioning matrix that usually is associated with the "curvature" of the objective function $L$.

- $H$ can be the inverse of the Hessian matrix, or some surrogate for the Hessian.
- In a DL setting, one needs to be mindful about storing $H$ and computing $H \nabla_\theta L$. 
Hessian matrix and Gauss-Newton Matrix

Consider a model (say a DNN) with parameters $\theta \in \mathbb{R}^n$.

- an input $x$ is fed into the model, which outputs $\hat{y} = f_\theta(x) \in \mathbb{R}^d$,
- and a loss function $\mathcal{L}$ that compares output $\hat{y}$ with the ground-truth $y$, i.e.,

$$L = \mathcal{L}(y, \hat{y}) = \mathcal{L}(y, f_\theta(x))$$

The Hessian matrix $H \in \mathbb{R}^{n \times n}$ (for a single data-point) can be expressed as

$$H = J_f^\top H_L J_f + \sum_{j=1}^{m} \left[ \frac{\partial L}{\partial \hat{y}} \right]_j \frac{\partial^2 [\hat{y}]}{\partial \theta^2},$$

where $J_f = \frac{\partial \hat{y}}{\partial \theta} \in \mathbb{R}^{d \times n}$ is the Jacobian matrix, and $H_L = \frac{\partial^2 L}{\partial \hat{y}^2} \in \mathbb{R}^{d \times d}$.

Ignoring the 2nd term in (3) yields the

Gauss-Newton matrix: $G = J_f^\top H_L J_f$

An advantage: For commonly-used loss functions $L$, $H_L \succeq 0 \Rightarrow G \succeq 0$. 
Fisher (Information) Matrix (FIM)

If the loss function $\mathcal{L}$ is the negative log likelihood of $p(y|x, \theta)$, the density function of the learned distribution $P_{x,y}(\theta)$ of $y$ given $x$ and $\theta$; i.e.,

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}(y, f_\theta(x)) = - \log p(y|x, \theta),$$

▶ $\mathcal{L}(y, \hat{y}) = \text{mean squared error} \iff P_{x,y}(\theta) = \text{Normal distrib. (with fixed variance)}$
▶ $\mathcal{L}(y, \hat{y}) = \text{binary cross entropy} \iff P_{x,y}(\theta) = \text{Bernoulli distrib.}$
▶ $\mathcal{L}(y, \hat{y}) = \text{multi-class cross entropy} \iff P_{x,y}(\theta) = \text{Categorical distrib. of } y \text{ (given } x).$

\[
\text{FIM: } F = \mathbb{E}_{x,y \sim P_{x,y}(\theta)} \left[ \frac{\partial \log p(y|x, \theta)}{\partial \theta} \left( \frac{\partial \log p(y|x, \theta)}{\partial \theta} \right)^\top \right], \tag{4}
\]

where $P_{x,y}(\theta)$ is the learned distribution of $(x, y)$ given $\theta$, with density function $p(x, y|\theta) = p(y|x, \theta)q(x)$, where $q(x)$ is the density function of $x$.

▶ Equivalent to Gauss-Newton (for certain loss functions)
▶ To estimate $F$, sample $x$ from data, compute $p(y|x, \theta)$ (forward pass); sample $y$ from $p(y|x, \theta)$, compute $\frac{\partial \log p(y|x, \theta)}{\partial \theta}$ (backward pass)
▶ The natural gradient method preconditions the gradient by $F^{-1}$.
▶ Empirical Fisher: replaces $P_{x,y}(\theta)$ with the data distribution of $(x, y)$. 

Don Goldfarb
Efficient Second-Order Stochastic Methods for ML
January 2023
Block Diagonal Preconditioners

\[ \mathbf{P} = P_{1}^{-1} \]

- Hessian based approximation: \( H_{l}^{-1} \)
- Fisher based approximation: \( F_{l}^{-1} \)
- Empirical Fisher based approximation: \( EF_{l}^{-1} \)

- KBFGS: \( H_{A} \otimes H_{G} \)
- TNT: \( U_{1}^{-1} \otimes U_{2}^{-1} \)
- KFAC: \( A^{-1} \otimes G^{-1} \)
- Shampoo: \( L_{1}^{-1} \otimes \ldots \otimes L_{k}^{-1} \)

Adaptive lr methods: Adam, AdaGrad, RMSProp...
Block Diagonal $H$ Assumption

Suppose $\theta$ can be divided into $L$ blocks $\theta_1, ..., \theta_L$, and we update each $\theta_i$ independently; i.e.,

$$\theta_l = \theta_l - \alpha H_l \nabla_{\theta_l} L, \text{ for } l = 1, ..., L.$$ 

- This is equivalent to $H = \text{diag}\{H_1, ..., H_L\}$, and is usually the first step for reducing storage and time complexity in DL settings.

- It is also aligned with the "modular" nature of DL models.
K-BFGS: Kronecker-factored Quasi-Newton Methods
Quasi-Newton Methods


▶ Idea: approximate the Hessian $B$, or $B^{-1}$, without explicitly computing 2nd derivs.
▶ We construct a (step, change in gradient) vector pair $(s, y)$ s.t.

$$y \approx Bs,$$

Note that $y = \tilde{B}s$, where $\tilde{B}$ is the average Hessian along the step $s$.
▶ We maintain $\hat{B}$, an approximation to $B$, by updating

$$\hat{B}^+ = \hat{B} - \frac{\hat{B}ss^\top \hat{B}}{s^\top \hat{B}s} + \rho yy^\top, \quad \text{where} \quad \rho = \frac{1}{s^\top y} > 0 \tag{5}$$

▶ We can also maintain $H$, an approximation to $B^{-1}$, by updating

$$H^+ = (I - \rho sy^\top)H(I - \rho ys^\top) + \rho ss^\top \tag{6}$$

Limited-memory BFGS (L-BFGS) [Liu and Nocedal, 1989]:

▶ Instead of maintaining $H$, we store the $p$ most recent $(s, y)$ pairs, and compute the action of $H$ on a vector from them.
▶ Applying BFGS or L-BFGS to train a DNN with $d$ parameters requires $O(d^2)$ for BFGS and $O(pd)$ for L-BFGS for both memory and work/iter.
Forward Pass of a Multi-layer Perceptron

- \( x \) is the input to the NN:
  \[ a_0 = x \]

- For fully-connected layers \( l = 1, \ldots, L \),
  \[ h_l = W_l a_{l-1} + b_l, \]
  \[ a_l = \phi_l(h_l) \]

- \( \theta_l \) parameters \([W_l \text{ weight matrix, } b_l \text{ bias}]; \phi_l \text{ activation function}; a_{l-1} \text{ input to layer } l; h_l \text{ pre-activation output}; a_l \text{ post-activation output} \)

- Compute \( \hat{y} \) the output of the NN:
  \[ \hat{y} = a_L \]

- Compute the loss function
  \[ L = \mathcal{L}(y, \hat{y}) \]
Backward Pass and Structure of Gradient

By chain rule,
\[ \frac{\partial L}{\partial W_l} = \sum_j \frac{\partial L}{\partial [h_l]_j} \frac{\partial [h_l]_j}{\partial W_l} = \frac{\partial L}{\partial h_l} a_{l-1}^\top := g_l a_{l-1}^\top \]

Backward pass: (let \( D_X := \frac{\partial L}{\partial X} \))

\[ \begin{align*}
    &\triangleright \quad D a_L \leftarrow \frac{\partial L(y, \hat{y})}{\partial \hat{y}} \\
    &\triangleright \text{ for } l = L, L - 1, \ldots, 1,
\end{align*} \]

\[ g_l \leftarrow D a_l \odot \phi'_l(h_l) \]

\[ D W_l \leftarrow g_l a_{l-1}^\top \]

\[ D a_{l-1} \leftarrow W_l^\top g_l \]

Observation: In the case of a single data-point, \( D W_l \) is a rank-1 matrix, and

\[ \text{vec}(D W_l) = \text{vec}(g_l a_{l-1}^\top) = a_{l-1} \otimes g_l. \]
Structure of the Hessian

Similarly, for a single data-point indexed with $i$, it can be shown that

$$
\frac{\partial^2 L(i)}{\partial \text{vec}(W_i)^2} = (a_{l-1}(i)a_{l-1}(i)^\top) \otimes \frac{\partial^2 L(i)}{\partial h_l(i)^2} := A_l(i) \otimes G_l(i),
$$

where $W_i \in \mathbb{R}^{d_o \times d_i}$, $A_l(i) \in \mathbb{R}^{d_i \times d_i}$, $G_l(i) \in \mathbb{R}^{d_o \times d_o}$.

Averaging over a dataset, whose points are indexed by $i$:

$$
\mathbb{E}_i \left[ \frac{\partial^2 L(i)}{\partial \text{vec}(W_i)^2} \right] = \mathbb{E}_i [A_l(i) \otimes G_l(i)] \approx \mathbb{E}_i [A_l(i)] \otimes \mathbb{E}_i [G_l(i)] := A_l \otimes G_l
$$
K-BFGS: Kronecker-factored Quasi-Newton Methods

KFAC, a precursor of K-BFGS

KFAC uses the facts that \( g_l a_{l-1}^\top = (a_{l-1} \otimes g_l) \) and \( \frac{\partial L}{\partial W_l} = \frac{\partial L}{\partial h_l} a_{l-1}^\top \) and that the \( l \)-th diagonal block of the Fisher matrix corresponding to layer \( l \) can be approximated as

\[
F_{W_l} = \mathbb{E} \left[ \text{vec}(D W_l) (\text{vec}(D W_l))^\top \right]
= \mathbb{E} \left[ (a_{l-1} \otimes g_l) (a_{l-1} \otimes g_l)^\top \right]
= \mathbb{E} \left[ \left( a_{l-1} a_{l-1}^\top \right) \otimes \left( g_l g_l^\top \right) \right]
\approx \mathbb{E} \left[ a_{l-1} a_{l-1}^\top \right] \otimes \mathbb{E} \left[ g_l g_l^\top \right]
:= A \otimes G
\]

Hence, KFAC also approximates the expectation of a Kronecker product of two matrices \( A \) and \( G \) by the Kronecker product of their expectations, but a different matrix \( G \).
K-BFGS

K-BFGS (for fully connected layers):

\[ H_l \nabla_{\theta_l} L = (A_l \otimes G_l)^{-1} \nabla_{\theta_l} L = (A_l^{-1} \otimes G_l^{-1}) \nabla_{\theta_l} L \]
\[ \approx (H_A \otimes H_G) \text{vec}(\nabla_{W_l} L) = H_G(\nabla_{W_l} L)H_A, \]  

(8)  

(9)

- Property #1: \((A \otimes B)^{-1} = A^{-1} \otimes B^{-1}\)
- Let \(H_A \approx A_l^{-1}, \ H_G \approx G_l^{-1}\)
- Property #2: \((A \otimes B) \text{vec}(X) = BXA\)

- Recall that \(W_l \in \mathbb{R}^{d_o \times d_i}, \ H_A \in \mathbb{R}^{d_i \times d_i}, \ H_G \in \mathbb{R}^{d_o \times d_o}\)
- \(H_A\) and \(H_G\) are maintained with BFGS/L-BFGS
K-BFGS for convolutional neural networks

Extending K-BFGS to convolutional neural networks:

- Suppose $W_i \in \mathbb{R}^{\Delta \times d_0 \times d_i}$ is the weight (tensor) of a conv layer
- Structure of Hessian:

$$
\mathbb{E}_i \left[ \frac{\partial^2 L(i)}{\partial \text{vec}(W_i)^2} \right] \approx A_i \otimes G_i,
$$

where $A_i \in \mathbb{R}^{\Delta d_i \times \Delta d_i}$, $G_i \in \mathbb{R}^{d_0 \times d_0}$
Damped BFGS for $G^{-1}$

- Recall BFGS update:
  \[
  H^+ = (I - \rho sy^\top)H(I - \rho ys^\top) + \rho ss^\top,
  \]
  where $s$ is ”change in step”, $y$ is ”change in gradient”, requires $\rho = \frac{1}{s^\top y} > 0$.

- Also recall that
  \[
  G_i = \mathbb{E}_i [G_i(i)] = \mathbb{E}_i \left[ \frac{\partial^2 L(i)}{\partial h_i(i)^2} \right]
  \]

- Construct the $(s, y)$ pairs for updating $H_G$:
  \[
  s_G = \mathbb{E}_i [h_i^+(i) - h_i(i)], \quad y_G = \mathbb{E}_i [g_i^+(i) - g_i(i)],
  \]
  where $g_i(i) = \frac{\partial L(i)}{\partial h_i(i)}$.

- Use **double damping** to deal with non-convexity, i.e., $\rho \not> 0$. 
Double Damping

Algorithm 1 $D_PD_{LM}$

1: Input: $s, y$; Output: $\tilde{s}, \tilde{y}$; Given: $H, 0 < \mu < 1, \lambda_G > 0$
2: if $s^\top y < \mu y^\top H y$ then
3: \hspace{1em} $\theta_1 = \frac{(1-\mu)y^\top H y}{y^\top H y - s^\top y}$
4: else
5: \hspace{1em} $\theta_1 = 1$
6: $\tilde{s} = \theta_1 s + (1 - \theta_1) H y$ \{Powell's damping on $H$\}
7: $\tilde{y} = y + \lambda_G \tilde{s}$ \{Levenberg-Marquardt damping on $H^{-1}$\}
8: return: $\tilde{s}, \tilde{y}$

- The first damping yields a matrix $\tilde{H}$ that interpolates the current $H$ and $H^\dagger$. Moreover,

$$\tilde{s}^\top y \geq \mu y^\top H y > 0$$

- After the second damping,

$$\tilde{s}^\top \tilde{y} \geq \lambda_G \|\tilde{s}\|^2.$$ 

Thus, $\lambda_G$ can be viewed as Levenberg-Martquardt (LM) damping.
"Hessian action" BFGS for $A^{-1}$

- Recall that

$$A_l = \mathbb{E}_i [A_l(i)] = \mathbb{E}_i \left[ a_{l-1}(i)a_{l-1}(i)\top \right]$$

Unlike $G_l$, $A_l$ is easy to estimate

- Construct the $(s, y)$ pairs for updating $H_A$ by "Hessian-action BFGS":

$$s_A = H_A\mathbb{E}_i[a_{l-1}(i)], \quad y_A = (A_l + \lambda_A I)s_A,$$  \hspace{1cm} (11)

where $\lambda_A$ is the (LM) damping term for $A_l$.

  - If $H = A$ and $A^+ = A + cc^T$, then above BFGS update yields $H^+ = (A^+)^{-1}$

  - If we estimate $A_l$ with the current minibatch, we don’t even need to explicitly compute $A_l$, because

$$A_l s_A = \mathbb{E}_i \left[ a_{l-1}(i)a_{l-1}(i)\top \right] s_A = \mathbb{E}_i \left[ (a_{l-1}(i)\top s_A) a_{l-1}(i) \right].$$  \hspace{1cm} (12)

We call this mini-batch Hessian-action BFGS.
High-level summary of K-BFGS/K-BFGS(L)

**Algorithm 2** Pseudocode of K-BFGS / K-BFGS(L)

**Require:** Given learning rate $\alpha$, overall damping value $\lambda$

1. Decide $\lambda_A$ and $\lambda_G$ from $\lambda$
2. for $k = 1, 2, \ldots$ do
3. Sample mini-batch of size $m$: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$
4. Perform a forward-backward pass over the current mini-batch $M_k$ to compute stochastic gradient
5. for $l = 1, \ldots, L$ do
6. $p_l = H_g^l \hat{\nabla} f_l H_a^l$
7. $W_l = W_l - \alpha \cdot p_l$
8. Perform another forward-backward pass over $M_k$ to get $h_l^+$ and $g_l^+$ ($l = 1, \ldots, L$)
9. Use damped BFGS or L-BFGS (with $\lambda_G$) to update $H_G^l$ ($l = 1, \ldots, L$)
10. Use Hessian-action BFGS (with $\lambda_A$) to update $H_A^l$ ($l = 1, \ldots, L$)
## Complexity of K-BFGS and K-BFGS(L)

**Table:** Memory and per-iteration time complexity for a weight matrix $W \in \mathbb{R}^{d_o \times d_i}$ for a fully-connected layer, using a batch of size $m$

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory</th>
<th>Time (per-iteration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>$d_i d_o$</td>
<td>$m d_i d_o$</td>
</tr>
<tr>
<td>K-BFGS</td>
<td>$d_i d_o + d_i^2 + d_o^2$</td>
<td>$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2$</td>
</tr>
<tr>
<td>K-BFGS(L)</td>
<td>$d_i d_o + d_i^2 + p d_o$</td>
<td>$\approx m d_i d_o + d_i^2 d_o + p d_i d_o$</td>
</tr>
<tr>
<td>KFAC</td>
<td>$d_i d_o + d_i^2 + d_o^2$</td>
<td>$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$</td>
</tr>
<tr>
<td>BFGS</td>
<td>$d_i^2 d_o^2$</td>
<td>$d_i^2 d_o^2$</td>
</tr>
</tbody>
</table>

By avoiding matrix inversion and using of relatively modest sized matrices or a few stored vectors (in (L) version), K-BFGS and K-BFGS(L) are efficient and memory parsimonious.
Experiments on Multi-layer Perceptron: Three Autoencoder Problems

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Layer width</th>
<th>Loss function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>784-1000-500-250-30-250-500-1000-784</td>
<td>binary entropy$^1$</td>
</tr>
<tr>
<td>FACES</td>
<td>625-2000-1000-500-30-500-1000-2000-625</td>
<td>MSE</td>
</tr>
<tr>
<td>CURVES</td>
<td>784-400-200-100-50-25-6-25-50-100-200-400-784</td>
<td>binary entropy$^2$</td>
</tr>
</tbody>
</table>

Activation functions $\phi_l$: ReLU, except for the layer in the middle, which uses linear $\phi_l$

Training settings:
- Algorithms: K-BFGS, K-BFGS(L), KFAC, Adam, SGD with momentum (SGD-m)
- Batch size: 1000
- Grid search
  - K-BFGS, K-BFGS(L), and KFAC: learning rate and $\lambda$ (damping)
  - Adam: learning rate and $\epsilon$
  - SGD-m: learning rate
Autoencoder Problem MNIST

Observations:

- The **second-order** methods clearly **outperform** the **first-order** methods, in terms of both epoch and running time.
- K-BFGS and K-BFGS(L) converge even faster than KFAC.
Autoencoder Problem FACES

Figure: FACES autoencoder

Observations:

▶ KFAC is slightly better than K-BFGS and K-BFGS(L), but the gap is small
Autoencoder Problem CURVES

Figure: CURVES autoencoder

Observations:
- Adam performs quite well
- K-BFGS is the best on this problem
Experiments on Convolutional Neural Networks

Classify CIFAR-10 and CIFAR-100 datasets using VGG-16 and ResNet-32, which contain sequential convolutional layers with batch normalization layers and skip connections.

Training settings:

- Batch size: 128
- Learning rate (LR) schedule:
  - 1st-order methods: decay by a factor of 0.1 every 60 epochs (200 epochs in total)
  - 2nd-order methods: decay by a factor of 0.1 every 50 epochs (150 epochs in total)
- Weight decay:
  - Shrink the parameters by a factor of $1 - \gamma$ before each update
  - Equivalent to $L_2$ regularization for SGD
  - Has been shown to help generalization for various optimizers, e.g., SGD, Adam [Loshchilov and Hutter, 2019], and KFAC [Zhang et al., 2019]

- Grid search
  - K-BFGS, K-BFGS(L), KFAC, and Adam: (initial) LR, weight decay, and damping
  - SGD-m: (initial) LR and weight decay
Convolutional Neural Networks

Figure: ResNet32 model on CIFAR-10
Convolutional Neural Networks

**Table:** Average of validation classification accuracy (%) achieved using 5 different random seeds with best hyper-parameter values

<table>
<thead>
<tr>
<th>Dataset Model</th>
<th>CIFAR10 VGG16</th>
<th>CIFAR10 ResNet32</th>
<th>CIFAR100 VGG16</th>
<th>CIFAR100 ResNet32</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-BFGS</td>
<td>94.28</td>
<td>93.45</td>
<td>75.65</td>
<td>71.46</td>
</tr>
<tr>
<td>K-BFGS(L)</td>
<td>94.24</td>
<td>93.38</td>
<td>75.43</td>
<td>70.98</td>
</tr>
<tr>
<td>KFAC</td>
<td><strong>94.39</strong></td>
<td>93.31</td>
<td><strong>76.21</strong></td>
<td>71.14</td>
</tr>
<tr>
<td>Adam</td>
<td>94.28</td>
<td>93.35</td>
<td>75.64</td>
<td>70.35</td>
</tr>
<tr>
<td>SGD-m</td>
<td>94.14</td>
<td>93.13</td>
<td>75.41</td>
<td>70.18</td>
</tr>
</tbody>
</table>
TNT: A Generic Approximate Natural Gradient Method
Pros and Cons of K-BFGS and KFAC

Pros:
- Better storage/time complexity compared with traditional 2nd-order methods
- Sometimes comparable complexity to 1st-order methods
- Potentially converge faster than 1st-order methods

Cons:
- Dependent on model structure; need special care for different types of layers
- Not guaranteed to be generalizable to ANY kind of model
- Somewhat burdensome to implement

Question: Can we keep the pros while avoiding the cons?
Preliminaries: the Fisher Matrix

Recall that

\[ F = \mathbb{E}_{x,y \sim P_{x,y}^{\theta}} \left[ \frac{\partial \log p(y|x, \theta)}{\partial \theta} \left( \frac{\partial \log p(y|x, \theta)}{\partial \theta} \right)^\top \right], \quad (13) \]

where \( P_{x,y}^{\theta} \) is the learned distribution of \((x, y)\) given \( \theta \), with its density function \( p(x, y|\theta) = p(y|x, \theta)q(x) \).

Note that

\[
\mathbb{E}_{y \sim p(y|x, \theta)} [\nabla_{\theta} \log p(y \mid x, \theta)] = \int \nabla_{\theta} \log p(y \mid x, \theta)p(y \mid x, \theta)dy \\
= \int \nabla_{\theta} p(y \mid x, \theta)dy = \nabla_{\theta} \left( \int p(y \mid x, \theta)dy \right) = \nabla_{\theta} 1 = 0
\]

Hence, the second moment of the gradient is equal to its covariance.
Tensor Normal Distribution

Definition

An arbitrary tensor $G \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ is said to follow a tensor normal (TN) distribution with mean parameter $M \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ and covariance parameters $U_1 \in \mathbb{R}^{d_1 \times d_1}$, ..., $U_k \in \mathbb{R}^{d_k \times d_k}$ if, and only if, $\text{vec}(G) \sim \text{Normal}(\text{vec}(M), U_1 \otimes \cdots \otimes U_k)$.

Imposing a Kronecker structure on the covariance matrix, vastly reduces the amount of memory required to estimate it, still capturing interactions between the tensor’s modes.

Basic idea underlying "Tensor Normal Training" (TNT):

- Let $\theta_l$ denotes the tensor parameter, (usually) corresponding to layer $l$, whether, ff-fc, batch normalization, convolutional, etc.
- If we view $\frac{\partial \log p}{\partial \theta_l}$ as a random tensor following a TN distribution, the Fisher matrix $F_{\theta_l}$ can be viewed as a covariance matrix,
A practical Natural Gradient (NG) method

- We assume, as in KFAC and Shampoo, that the Fisher matrix is block diagonal.
- Each block corresponds to the covariance of the gradient tensor \( G := D W \), w.r.t. the model’s tensor variable \( W \). Hence, the approximate Fisher matrix is:

\[
F \approx \text{diag}_{l=1}^{L} \left\{ \mathbb{E}_{x \sim Q_x, y \sim p} \left[ \text{vec}(D W_l) (\text{vec}(D W_l))^\top \right]\right\} = \text{diag}_{l=1}^{L} \{ \text{Var}(\text{vec}(D W_l)) \}.
\]

- To approximate \( \text{Var}(\text{vec}(D W)) \), we assume that \( G \in \mathbb{R}^{d_1 \times \cdots \times d_k} \) follows a TN distribution with zero mean and covariance parameters \( U_1, \ldots, U_k \), where \( U_i \in \mathbb{R}^{d_i \times d_i} \). Thus, the Fisher matrix corresponding to \( W \) is

\[
F_W = \mathbb{E}_{x \sim Q_x, y \sim p}[\text{Var}(\text{vec}(G))] = U_1 \otimes \cdots \otimes U_k
\]

and

\[
F_W^{-1} \text{vec}(\nabla_W \mathcal{L}) = (U_1^{-1} \otimes \cdots \otimes U_k^{-1}) \text{vec}(\nabla_W \mathcal{L}) = \text{vec} \left( \nabla_W \mathcal{L} \times_1 U_1^{-1} \times_2 \cdots \times_k U_k^{-1} \right),
\]

where \( \times_i \) denotes a mode-\( i \) product (generalization of matrix product).

- To estimate the covariance submatrices \( U_1, \ldots, U_k \), we use the following property:

\[
\mathbb{E} \left[ G^{(i)} \right] = U_i \cdot \prod_{j \neq i} \text{tr} (U_j)
\]

where \( G^{(i)} := \text{mat}_i(G) \text{mat}_i(G)^\top \in \mathbb{R}^{d_i \times d_i} \) denotes the contraction of \( G \) with itself along all but the \( i \)th dimension, and \( \text{mat}_i \) refers to matricization of a tensor.
An empirical comparison with respect to block Fisher method

Figure: Cosine between the directions produced by the methods listed in the legend and that of the block Fisher method

- Problem: classify a $16 \times 16$ down-scaled MNIST
- Algorithms were run on a small feed-forward NN with layer widths 256-20-20-20-20-20-10.
- For all methods, we followed the trajectory produced by the block Fisher method.
- Cosine Similarity: want the cosine to be as close to 1 as possible.
Complexity of TNT

Table: Memory and per-iteration time complexity for a weight matrix $W \in \mathbb{R}^{d_o \times d_i}$ from a feed-forward fully-connected layer

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory</th>
<th>Time (per-iteration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>$d_i d_o$</td>
<td>$md_i d_o$</td>
</tr>
<tr>
<td>K-BFGS</td>
<td>$d_i d_o + d_i^2 + d_o^2$</td>
<td>$\approx md_i d_o + d_i^2 d_o + d_i d_o^2$</td>
</tr>
<tr>
<td>TNT</td>
<td>$d_i d_o + d_i^2 + d_o^2$</td>
<td>$\approx md_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$</td>
</tr>
<tr>
<td>KFAC</td>
<td>$d_i d_o + d_i^2 + d_o^2$</td>
<td>$\approx md_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$</td>
</tr>
<tr>
<td>BFGS</td>
<td>$d_i^2 d_o^2$</td>
<td>$d_i^2 d_o^2$</td>
</tr>
</tbody>
</table>

Note: for larger layers, such as convolutional layers, TNT has better memory and time complexity than KFAC and K-BFGS.
Autoencoder Problem MNIST

Figure: MNIST autoencoder

Observations:
- TNT performs better than Shampoo and 1st-order methods
Autoencoder Problem FACES

Figure: FACES autoencoder

Observations:
- TNT performs similarly to KFAC and K-BFGS/K-BFGS(L)
Autoencoder Problem CURVES

Figure: CURVES autoencoder

Observations:

- TNT is one of the best in terms of process time on this problem
Convolutional Neural Networks

Table: Average of validation classification accuracy (%) achieved using 5 different random seeds with best HP values

<table>
<thead>
<tr>
<th>Dataset Model</th>
<th>CIFAR10 VGG16</th>
<th>CIFAR10 ResNet32</th>
<th>CIFAR100 VGG16</th>
<th>CIFAR100 ResNet32</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-BFGS</td>
<td>94.28</td>
<td>93.45</td>
<td>75.65</td>
<td>71.46</td>
</tr>
<tr>
<td>KFAC</td>
<td><strong>94.39</strong></td>
<td>93.31</td>
<td><strong>76.21</strong></td>
<td>71.14</td>
</tr>
<tr>
<td>TNT</td>
<td>94.31</td>
<td><strong>93.48</strong></td>
<td>76.01</td>
<td><strong>71.70</strong></td>
</tr>
<tr>
<td>Shampoo</td>
<td>93.88</td>
<td>93.13</td>
<td>76.02</td>
<td>70.35</td>
</tr>
<tr>
<td>Adam</td>
<td>94.28</td>
<td>93.35</td>
<td>75.64</td>
<td>70.35</td>
</tr>
<tr>
<td>SGD-m</td>
<td>94.14</td>
<td>93.13</td>
<td>75.41</td>
<td>70.18</td>
</tr>
</tbody>
</table>

**xx.xx**: best accuracy  
**xx.xx**: next best accuracy
Setting

- DNN with $L$ layers, defined by weight matrices $W_l$, for $l \in [L]$.
- DNN transforms $x$ to an output $f(W, x)$.
- The average loss $\mathcal{L}$ over the training set:

$$
\mathcal{L}(W) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(W, x_i), y_i),
$$

- The Jacobian $J(W)$ of $\mathcal{L}(\cdot)$ w.r.t $W$ is

$$
J = \begin{bmatrix} J_1^T, \ldots, J_n^T \end{bmatrix}^T \in \mathbb{R}^{n \times p} \text{ where } J_i^T = \text{vec}\left( \frac{\partial \ell(f(W, x_i), y_i)}{\partial W} \right)
$$

- We use the notation $J_i^X^T = \text{vec}\left( \frac{\partial \ell(f(W, x_i), y_i)}{\partial X} \right)$ and $J^X = \begin{bmatrix} J_1^X^T, \ldots, J_n^X^T \end{bmatrix}^T$ for any subset of variables $X$ of $W$. 
Setting

\[ \mathcal{L}(y, \hat{y}) = \mathcal{L}(y, f_\theta(x)) = -\log p_W(y|x) \] [Martens, 2014]

\[
F(W) = \mathbb{E}_{x \sim Q_x, y \sim p_W(\cdot|x)} \left[ \frac{\partial \log p_W(y|x)}{\partial W} \left( \frac{\partial \log p_W(y|x)}{\partial W} \right)^\intercal \right].
\]

- Estimator for \( F \): Sample \( x \) from data, compute \( p_W(y|x) \) (forward pass), sample \( y \sim p_W(y|x) \), compute \( \frac{\partial \log p_W(y|x)}{\partial W} \) (backward pass).
- Resulting direction is called the natural gradient direction.
- The empirical Fisher matrix (EFM) \( \tilde{F}(W) \): using the \((x, y)\) data dist:

\[
\tilde{F}(W) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell(f(W, x_i), y_i)}{\partial W} \left( \frac{\partial \ell(f(W, x_i), y_i)}{\partial W} \right)^\intercal = \frac{1}{n} J(W)^\intercal J(W).
\]
MBF for feed-forward fully connected layers

For a ff-fc layer with $I$ inputs and $O$ outputs, we use the $O \times O$ block-diagonal approximation:

$$F_{W_l} \approx \text{diag}\{F_{W_l,1}, \ldots, F_{W_l,O}\},$$

with $F_{W_l,j} := \frac{1}{n} (J_{W_l,j}^T J_{W_l,j})$ of size $(I + 1) \times (I + 1)$. 

**Figure:** Illustration of MBF’s preconditionner for a feed-forward fully-connected layer.
MBF for feed-forward fully connected layers - Motivation

- [Roux et al., 2008] and [Collobert, 2004]: asymptotic structure of the Hessian of one-hidden-layer-NN.
- [ichi Amari et al., 2018] small magnitude of FIM off-diagonal blocks.
- Empirical observation on toy problem:
  - Classify a $16 \times 16$ down-scaled MNIST
  - Small ff-NN with layer widths 256-20-20-20-20-20-10.

**Figure:** Absolute EFM inverse after 50 epochs of the last and middle layers (including bias) of a small FCC-NN.
MBF for convolutional layers

For a Conv layer with $J$ input channels, and $I$ output channels, there are $I \times O$ kernels $W_{i,j,i}$, each of size $(2R+1) \times (2R+1) := \Delta$.

we use the $(IJ+1) \times (IJ+1)$ block-diagonal approximation:

$$F^{W_l} \approx \text{diag}\{F^{W_{l,1,1}}, \ldots, F^{W_{l,1,I}}, \ldots, F^{W_{l,J,1}}, \ldots, F^{W_{l,J,I}}, F^{b_l}\},$$

with $F^{W_{l,j,i}} := \frac{1}{n} (J^{W_{l,j,i}})^T J^{W_{l,j,i}}$ of size $\Delta \times \Delta$. 

*Figure:* Illustration of MBF’s preconditionner for a convolutional layer.
MBF for convolutional layers - Motivation

- The input-output relationship in a CNN is analogous to the ff-NN, except that the role of input and output node sets $J$ and $I$ are taken on by the input and output channels.
- Empirical observation on toy problem:
  - Classify a $16 \times 16$ down-scaled MNIST.
  - Simple CNN [Shallue et al., 2019] (2 conv + 1 ff).

Figure: Absolute EFM inverse after 10 epochs for the first convolutional layer of the Simple CNN network that uses 32 filters of size $5 \times 5$. 

(a) First CNN layer  
(b) Zoom on first 10 blocks
Algorithm 3 Generic MBF training algorithm

Require: Given learning rates \( \{\alpha_k\} \), damping value \( \lambda \), batch size \( m \)
1: for \( k = 1, 2, \ldots \) do
2: Sample mini-batch \( M \) of size \( m \)
3: Perform a forward-backward pass over \( M \)
to compute stochastic gradient \( \mathcal{D} W_l \) (\( l = 1, \ldots, L \))
4: for \( l = 1, \ldots, L \) do
5: for mini-block \( b \) in layer \( l \), in parallel do
6: \( F_{W_l,b}^{-1} := \left( \frac{1}{m} (J_{W_l,b}^T J_{W_l,b} + \lambda I) \right)^{-1} \)
7: \( W_{l,b} = W_{l,b} - \alpha_k F_{W_l,b}^{-1} \mathcal{D} W_{l,b} \)
MBF convergence

Assumptions:
- **AS.1:** The mini-block Gram matrices \( J^{W_l,b}(0)J^{W_l,b}(0)^T \) at initialization are positive definite, i.e. \( \min_{l \in [L]} \min_b \lambda_{\text{min}}(J^{W_l,b}(0)^T J^{W_l,b}(0)) = \lambda_0 > 0 \).
- **AS.2:** There exists \( 0 < C \leq \frac{1}{2} \) that satisfies \( \|J(W(k)) - J(W(0))\|_2 \leq \frac{C\sqrt{\lambda_0}}{3} \) if \( \|W(k) - W(0)\|_2 \leq \frac{3}{\sqrt{\lambda_0}} \|y - u(0)\|_2 \).

Theorem

Suppose **AS.1, AS.2** hold, and the Generic MBF Algorithm 1, uses exact gradients and the mini-block version of the true Fisher as the underlying preconditioning matrix for a network with \( L \) layers. Then there exists an interval of suitable damping values \( \lambda \) in \([\lambda, \bar{\lambda}]\) and corresponding small enough learning rates \( \eta_\lambda \), s.t for any \( 0 \leq \eta \leq \eta_\lambda \), we have

\[
\|u(W(k)) - y\|_2^2 \leq (1 - \eta)^k \|u(W(0)) - y\|_2^2.
\]
## Computational/Storage Requirements of MBF

**Table:** Per-iteration Computational and Storage Requirements for a Convolutional Layer.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Additional pass</th>
<th>Curvature</th>
<th>Step $\Delta W_i$</th>
<th>Storage $P_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBF</td>
<td>—</td>
<td>$O\left(\frac{IJ</td>
<td>\Delta</td>
<td>^2}{T_1} + \frac{IJ</td>
</tr>
<tr>
<td>Shampoo</td>
<td>—</td>
<td>$O\left(\frac{(J^2+</td>
<td>\Delta</td>
<td>^2+I^2)}{T_1} + \frac{J^3+I^3+</td>
</tr>
<tr>
<td>KFAC</td>
<td>$O\left(\frac{mIJ</td>
<td>\Delta</td>
<td></td>
<td>\tau</td>
</tr>
<tr>
<td>Adam</td>
<td>—</td>
<td>$O(IJ</td>
<td>\Delta</td>
<td>)$</td>
</tr>
</tbody>
</table>
Autoencoder Problem MNIST

Figure: MNIST autoencoder
Autoencoder Problem FACES

Figure: FACES autoencoder
Autoencoder Problem CURVES

Figure: CURVES autoencoder
CNN Problems

![Graph showing validation error and process time for different optimization methods on CIFAR10 ResNet32. The graph compares SGD-m, Adam, Shampoo, MBF, and KFAC.](image)

**Figure**: CIFAR10 ResNet32
CNN Problems

Figure: CIFAR100 VGG16
CNN Problems

Figure: SVHN VGG11
Subsampled Gauss-Newton and Natural Gradient Methods
Gauss-Newton matrix

- Gauss-Newton matrix for a single data-point index by $i$:

$$G(i) = J_f^\top(i)H_L(i)J_f(i), \quad (14)$$

where $J_f(i) \in \mathbb{R}^{d \times n}$ and $H_L(i) \in \mathbb{R}^{d \times d}$ ($n$: # of parameters; $d$: size of output)

- Gauss-Newton matrix averaged over the dataset:

$$G = \mathbb{E}_i[G(i)] = \mathbb{E}_i[J_f^\top(i)H_L(i)J_f(i)]$$

- Estimating $G$ from a minibatch (i.e., subsampling):

$$\hat{G} = \frac{1}{m} \sum_{i=1}^{m} J_f^\top(i)H_L(i)J_f(i)$$
Gauss-Newton matrix (continued)

Adding a Levenberg-Marquardt (LM) damping term $\lambda$:

$$
\hat{G}_{LM} = \frac{1}{m} \sum_{i=1}^{m} J_f^\top(i) H_L(i) J_f(i) + \lambda I := \frac{1}{m} J^\top H J + \lambda I,
$$

where $J \in \mathbb{R}^{md \times n}$ and $H \in \mathbb{R}^{md \times md}$ are defined as:

$$
J = \begin{pmatrix}
J_f(1) \\
\vdots \\
J_f(m)
\end{pmatrix},
H = \text{diag}\{H_L(1), ..., H_L(m)\}$$
Fisher matrix

Fisher (information) matrix:

\[ F = \mathbb{E}_{x,y \sim P_{x,y}(\theta)} \left[ \frac{\partial \log p(y|x, \theta)}{\partial \theta} \left( \frac{\partial \log p(y|x, \theta)}{\partial \theta} \right)^\top \right], \quad (15) \]

Estimating \( F \) from a minibatch:

1. obtain \( x_1, \ldots, x_m \) from a minibatch
2. sample \( y_i \sim p(y|x_i, \theta) \) for \( i = 1, \ldots, m \)

\[ \hat{F} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \log p(y_i|x_i, \theta)}{\partial \theta} \left( \frac{\partial \log p(y_i|x_i, \theta)}{\partial \theta} \right)^\top \]
Fisher matrix (continued)

- Adding a LM damping term $\lambda$:

$$
\hat{F}_{\text{LM}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \log p(y_i|x_i, \theta)}{\partial \theta} \left( \frac{\partial \log p(y_i|x_i, \theta)}{\partial \theta} \right)^\top + \lambda I := \frac{1}{m} J^\top H J + \lambda I,
$$

where $J \in \mathbb{R}^{m \times n}$ and $H \in \mathbb{R}^{m \times m}$ are defined as:

$$
J = \begin{pmatrix}
\left( \frac{\partial \log p(y_1|x_1, \theta)}{\partial \theta} \right)^\top \\
\vdots \\
\left( \frac{\partial \log p(y_m|x_m, \theta)}{\partial \theta} \right)^\top
\end{pmatrix},
H = I
$$
Sherman-Morrison-Woodbury Formula

- **Sherman-Morrison-Woodbury (SMW) formula:**

\[
(A + UCV)^{-1} = A^{-1} - A^{-1} U \left( C^{-1} + VA^{-1} U \right)^{-1} VA^{-1}
\]  

(16)

- By SMW,

\[
G = \lambda I + \frac{1}{m} J^\top H J
\]  

(17)

\[
\Rightarrow p = -G^{-1} g = -\frac{1}{\lambda} \left( I - \frac{1}{m} J^\top D^{-1} J \right) g,
\]  

(18)

where \( D = \lambda H^{-1} + \frac{1}{m} JJ^\top \). Moreover, \( JJ^\top \) can be computed efficiently by the structure of gradient.

- Size of matrices to invert:

\[
G, F : n \times n
\]

\[
D : \begin{cases} 
  md \times md, & \text{for } G \ (\text{SMW-GN}) \\
  m \times m, & \text{for } F \ (\text{SMW-Fisher}) 
\end{cases}
\]

- \( n \): number of parameters in the NN
- \( d \): size of output layer
- \( m \): size of mini-batch.
Discussion

- We can further impose the block diagonal assumption, which leads to the block diagonal versions of SMW-GN and SMW-Fisher. This is presented in our paper. A more detailed implementation (M-FAC) and discussion can be found in a recent paper by Frantar, Kurtic and Alistarh on M-FAC (arXiv:2107.03356)

- However, none of these methods demonstrate promising performance in our experiments

- Limitations:
  - Hard to incorporate moving averages to gather past information and obtain accurate estimates to the pre-conditioning matrices
  - SMW-GN is hard to scale up when $d$ becomes very large

- On the other hand, the idea still leads to some follow-up work that shows encouraging results:
  - e.g., Yang et al. [2020]

Thank you.
References I


References II


References III


