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

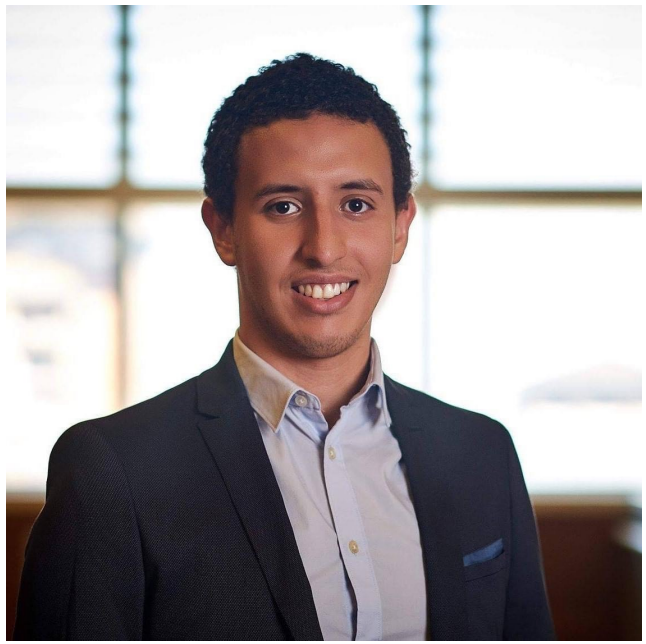
Don Goldfarb

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:

- D. Goldfarb, Y. Ren, A. Bahamou. *Practical quasi-Newton methods for training deep neural networks*
- Y. Ren, A. Bahamou, D. Goldfarb. *Kronecker-factored quasi-Newton methods for convolutional neural networks*

TNT:

- Y. Ren, D. Goldfarb. *Tensor normal training for deep learning models*

MBF:

- A. Bahamou, D. Goldfarb, and Y. Ren. *A mini-block Fisher method for deep neural networks*

Subsampled GN and NG methods:

- Y. Ren, D. Goldfarb. *Efficient subsampled Gauss-Newton and natural gradient methods for training neural networks*

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)

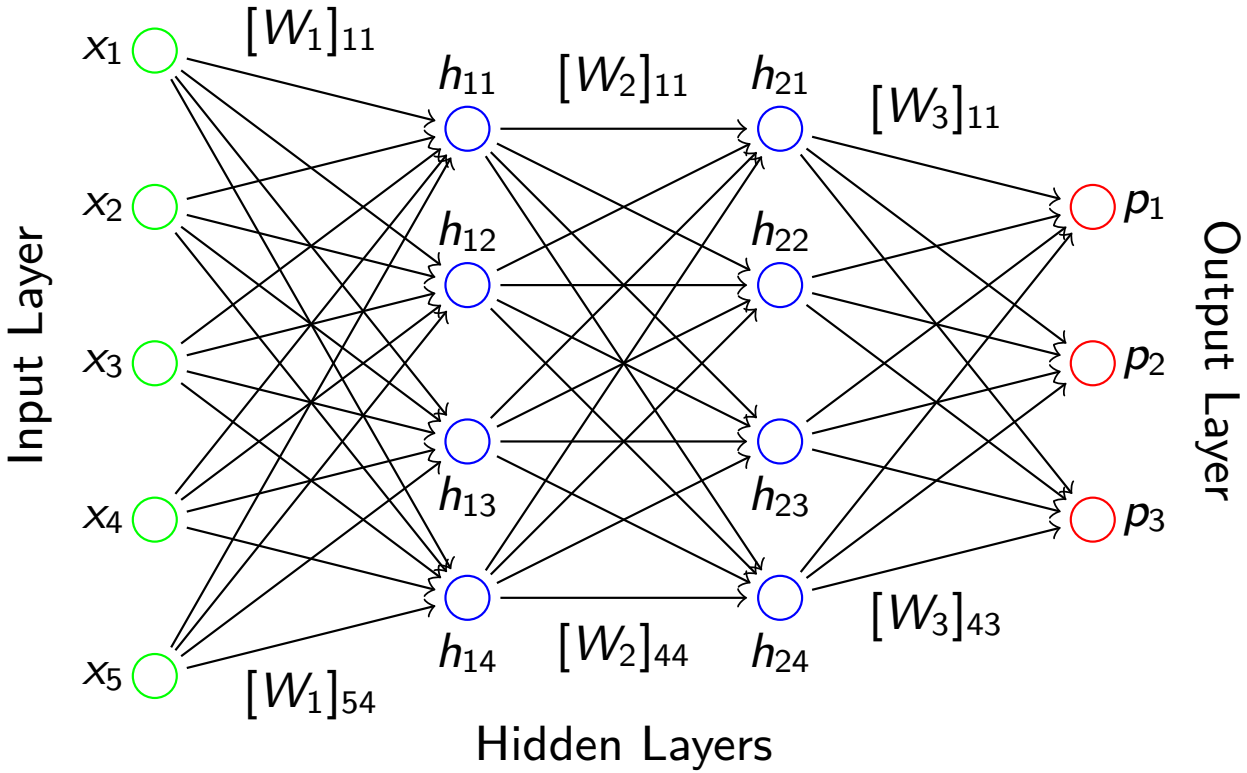
Subsampled Gauss-Newton and Natural Gradient Methods

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: $\sim$1.28M images for training, 50K for testing, 1,000 classes.

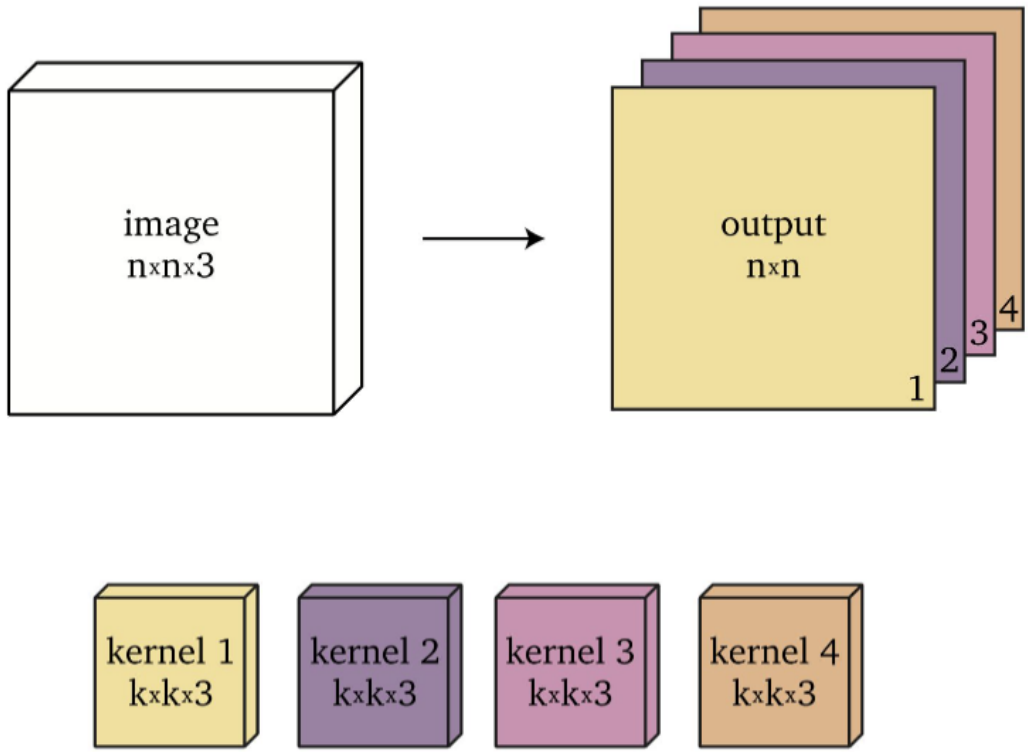
Feed-forward Fully-connected Neural Network



Note: $\mathbf{h}_{l,i} = (W_l \mathbf{a}_{l-1})_i$ is the i-th component of the pre-activation output of the ℓ-th layer. Activation by ϕ yields the input $(\mathbf{a}_l)_i = \phi(\mathbf{h}_{l,i})$ to the i-th node in layer $\ell + 1$.

Given a matrix of labeled training data $(\mathbf{a}_0, y)$, training a FF-FC DNN is the optimization problem: $\inf_{\{W_\ell\},\{a_\ell\}} \mathcal{L}(W_L \mathbf{a}_{L1}, y)$, where $\mathcal{L}$ 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:
  - BFGS [Broyden, 1970, Fletcher, 1970, Goldfarb, 1970, Shanno, 1970],
  - 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],
  - Stochstic 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:

$$\theta = \theta - \alpha \nabla_\theta L \tag{1}$$

  where α is the learning rate (step size).

- Pre-conditioned SGD:

$$\theta = \theta - \alpha H \nabla_\theta L, \tag{2}$$

  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

$$H = J_f^\top H_L J_f + \sum_{j=1}^{m} \left[\frac{\partial L}{\partial \hat{y}}\right]_j \frac{\partial^2 [\hat{y}]_j}{\partial \theta^2}, \tag{3}$$

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 θ; i.e.,

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

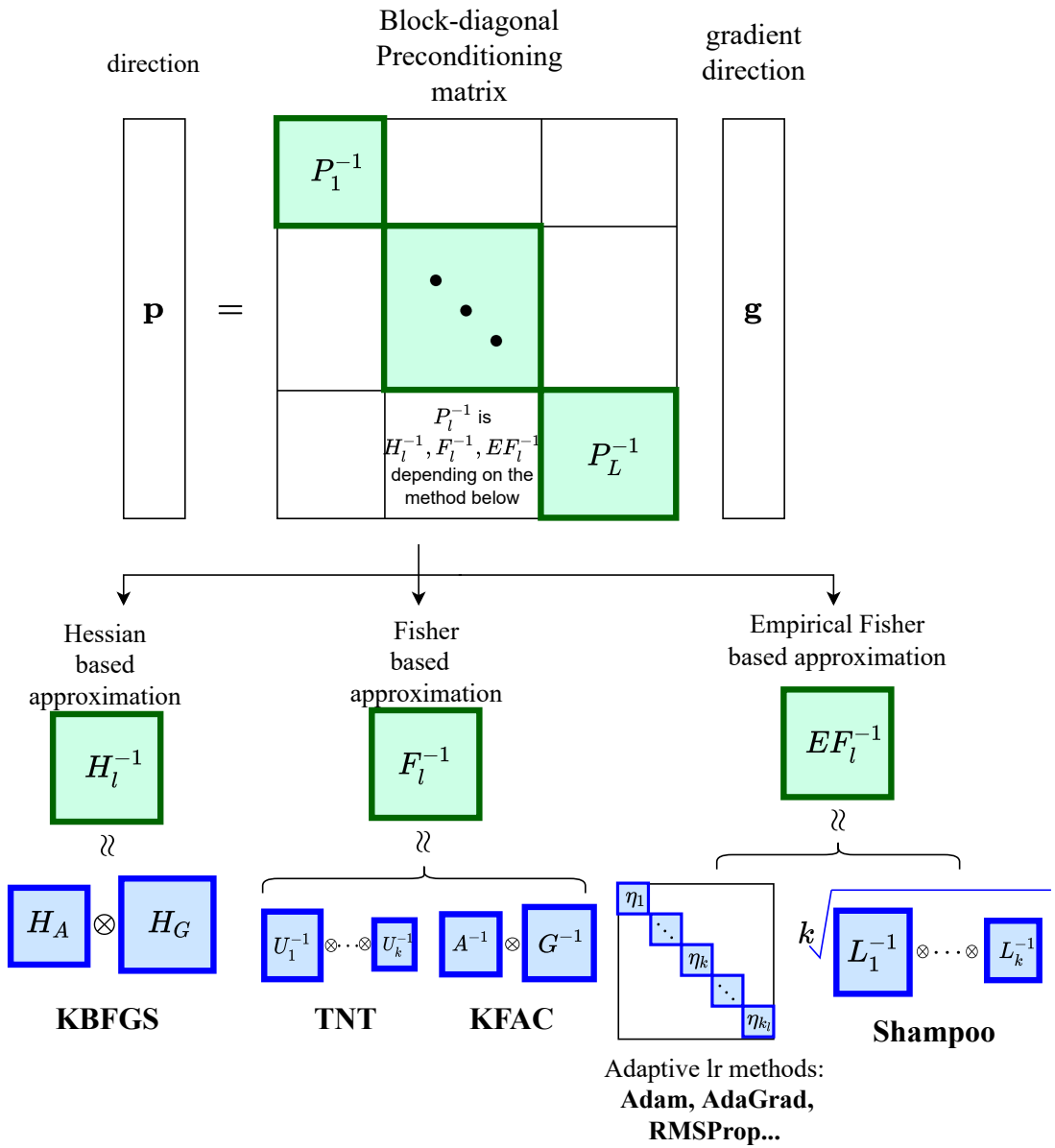
- $\mathcal{L}(y,\hat{y}) =$ mean squared error $\Leftrightarrow P_{x,y}(\theta) =$ Normal distrib. (with fixed variance)
- $\mathcal{L}(y,\hat{y}) =$ binary cross entropy $\Leftrightarrow P_{x,y}(\theta) =$ Bernoulli distrib.
- $\mathcal{L}(y,\hat{y}) =$ multi-class cross entropy $\Leftrightarrow P_{x,y}(\theta)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

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

where $P_{x,y}(\theta)$ is the learned distribution of (x,y) given θ, 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)

Block Diagonal Preconditioners



Block Diagonal H Assumption

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

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

- 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

BFGS [Broyden, 1970, Fletcher, 1970, Goldfarb, 1970, Shanno, 1970] update:

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

$$\mathbf{y} \approx B\mathbf{s},$$

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

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

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

$$H^+ = (I - \rho\mathbf{s}\mathbf{y}^\top)H(I - \rho\mathbf{y}\mathbf{s}^\top) + \rho\mathbf{s}\mathbf{s}^\top \tag{6}$$

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

- Instead of maintaining H, we store the p most recent $(\mathbf{s}, \mathbf{y})$ 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 $l = 1, ..., L$,

$$\mathbf{h}_l = W_l \mathbf{a}_{l-1} + \mathbf{b}_l,$$
$$\mathbf{a}_l = \phi_l(\mathbf{h}_l)$$

- θ_l parameters [W_l weight matrix, $\mathbf{b}_l$ bias]; ϕ_l activation function; $\mathbf{a}_{l-1}$ input to layer l; $\mathbf{h}_l$ pre-activation output; $\mathbf{a}_l$ 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 $\mathcal{D}X := \frac{\partial L}{\partial X}$)

- $\mathcal{D}a_L \leftarrow \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \hat{y}}$
- for $l = L, L-1, ..., 1$,

$$g_l \leftarrow \mathcal{D}a_l \odot \phi_l'(h_l)$$
$$\mathcal{D}W_l \leftarrow g_l a_{l-1}^\top$$
$$\mathcal{D}a_{l-1} \leftarrow W_l^\top g_l$$

Observation: In the case of a single data-point, $\mathcal{D}W_l$ is a rank-1 matrix, and

$$\text{vec}(\mathcal{D}W_l) = \text{vec}(g_l a_{l-1}^\top) = a_{l-1} \otimes g_l.$$

Structure of the Hessian

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

$$\frac{\partial^2 L(i)}{\partial \text{vec}(W_l)^2} = (a_{l-1}(i)a_{l-1}(i)^\top) \otimes \frac{\partial^2 L(i)}{\partial h_l(i)^2} := A_l(i) \otimes G_l(i), \tag{7}$$

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

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

$$\mathbb{E}_i \left[\frac{\partial^2 L(i)}{\partial \text{vec}(W_l)^2} \right] = \mathbb{E}_i \left[A_l(i) \otimes G_l(i) \right] \approx \mathbb{E}_i \left[A_l(i) \right] \otimes \mathbb{E}_i \left[G_l(i) \right] := A_l \otimes G_l$$

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

$$\begin{aligned} F_{W_l} &= \mathbb{E}\left[\mathrm{vec}(\mathcal{D}W_l)\left(\mathrm{vec}(\mathcal{D}W_l)\right)^\top\right] \\ &= \mathbb{E}\left[(a_{l-1} \otimes g_l)(a_{l-1} \otimes g_l)^\top\right] \\ &= \mathbb{E}\left[\left(a_{l-1}a_{l-1}^\top\right) \otimes \left(g_l g_l^\top\right)\right] \\ &\approx \mathbb{E}\left[a_{l-1}a_{l-1}^\top\right] \otimes \mathbb{E}\left[g_l g_l^\top\right] \\ &:= A \otimes G \end{aligned}$$

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):

$$H_l \nabla_{\theta_l} L = (A_l \otimes G_l)^{-1} \nabla_{\theta_l} L = (A_l^{-1} \otimes G_l^{-1}) \nabla_{\theta_l} L \tag{8}$$

$$\approx (H_A \otimes H_G) \text{vec}(\nabla_{W_l} L) = H_G (\nabla_{W_l} L) H_A, \tag{9}$$

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

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

where $A_l \in \mathbb{R}^{\Delta d_i \times \Delta d_i}$, $G_l \in \mathbb{R}^{d_o \times d_o}$

Damped BFGS for G^{-1}

- Recall BFGS update:

$$H^+ = (I - \rho \mathbf{s}\mathbf{y}^\top) H (I - \rho \mathbf{y}\mathbf{s}^\top) + \rho \mathbf{s}\mathbf{s}^\top, \tag{10}$$

  where $\mathbf{s}$ is "change in step", $\mathbf{y}$ is "change in gradient", reqires $\rho = \frac{1}{\mathbf{s}^\top \mathbf{y}} > 0$.
- Also recall that

$$G_l = \mathbb{E}_i \left[G_l(i) \right] = \mathbb{E}_i \left[\frac{\partial^2 L(i)}{\partial \mathbf{h}_l(i)^2} \right]$$

- Construct the $(\mathbf{s}, \mathbf{y})$ pairs for updating H_G:

$$\mathbf{s}_G = \mathbb{E}_i[h_l^+(i) - h_l(i)], \quad \mathbf{y}_G = \mathbb{E}_i[g_l^+(i) - g_l(i)],$$

  where $g_l(i) = \frac{\partial L(i)}{\partial h_l(i)}$.

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

Double Damping

Algorithm 1 $D_P D_{LM}$

Input: $\mathbf{s}$, $\mathbf{y}$; **Output:** $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$; **Given:** H, $0 < \mu < 1$, $\lambda_G > 0$
if $\mathbf{s}^\top \mathbf{y} < \mu \mathbf{y}^\top H \mathbf{y}$ **then**
  $\theta_1 = \frac{(1-\mu)\mathbf{y}^\top H \mathbf{y}}{\mathbf{y}^\top H \mathbf{y} - \mathbf{s}^\top \mathbf{y}}$
else
  $\theta_1 = 1$
$\tilde{\mathbf{s}} = \theta_1 \mathbf{s} + (1 - \theta_1) H\mathbf{y}$ {Powell's damping on H}
$\tilde{\mathbf{y}} = \mathbf{y} + \lambda_G \tilde{\mathbf{s}}$ {Levenberg-Marquardt damping on H^{-1}}
return: $\tilde{\mathbf{s}}$, $\tilde{\mathbf{y}}$

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

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

- After the second damping,

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

Thus, λ_G can be viewed as Levenberg-Martquart (LM) damping.

"Hessian action" BFGS for A^{-1}

- Recall that

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

  Unlike G_l, A_l is easy to estimate

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

$$\mathbf{s}_A = H_A\mathbb{E}_i[a_{l-1}(i)], \quad \mathbf{y}_A = (A_l + \lambda_A I)\mathbf{s}_A, \tag{11}$$

  where λ_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

$$A_l\mathbf{s}_A = \mathbb{E}_i\left[\mathbf{a}_{l-1}(i)\mathbf{a}_{l-1}(i)^\top\right]\mathbf{s}_A = \mathbb{E}_i\left[(\mathbf{a}_{l-1}(i)^\top\mathbf{s}_A)\mathbf{a}_{l-1}(i)\right]. \tag{12}$$

  We call this mini-batch Hessian-action BFGS.

High-level summary of K-BFGS/K-BFGS(L)

Algorithm 2 Pseodu-code of K-BFGS / K-BFGS(L)

Require: Given learning rate α, overall damping value λ

Decide λ_A and λ_G from λ
for $k = 1, 2, \ldots$ **do**
  Sample mini-batch of size m: $M_k = \{\xi_{k,i}, i = 1, \ldots, m\}$
  Perform a forward-backward pass over the current mini-batch M_k to compute stochastic gradient
  for $l = 1, \ldots, L$ **do**
    $p_l = H_g^l \widehat{\nabla \mathbf{f}}_l H_a^l$
    $W_l = W_l - \alpha \cdot p_l$
  Perform another forward-backward pass over M_k to get h_l^+ and g_l^+ $(l = 1, ..., L)$
  Use damped BFGS or L-BFGS (with λ_G) to update H_G^l $(l = 1, ..., L)$
  Use Hessian-action BFGS (with λ_A) to update H_A^l $(l = 1, ..., L)$

Complexity of K-BFGS and K-BFGS(L)

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

Name	Memory	Time (per-iteration)
SGD	$d_i d_o$	$m d_i d_o$
K-BFGS	$d_i d_o + d_i^2 + d_o^2$	$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2$
K-BFGS(L)	$d_i d_o + d_i^2 + p d_o$	$\approx m d_i d_o + d_i^2 d_o + p d_i d_o$
KFAC	$d_i d_o + d_i^2 + d_o^2$	$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$
BFGS	$d_i^2 d_o^2$	$d_i^2 d_o^2$

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

Dataset	Layer width	Loss function
MNIST	784-1000-500-250-30-250-500-1000-784	binary entropy[1]
FACES	625-2000-1000-500-30-500-1000-2000-625	MSE
CURVES	784-400-200-100-50-25-6-25-50-100-200-400-784	binary entropy[2]

Activation functions ϕ_l: ReLU, except for the layer in the middle, which uses linear ϕ_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 λ (damping)
  - Adam: learning rate and ϵ
  - SGD-m: learning rate

[1] with sigmoid function
[2] with sigmoid function

Autoencoder Problem MNIST

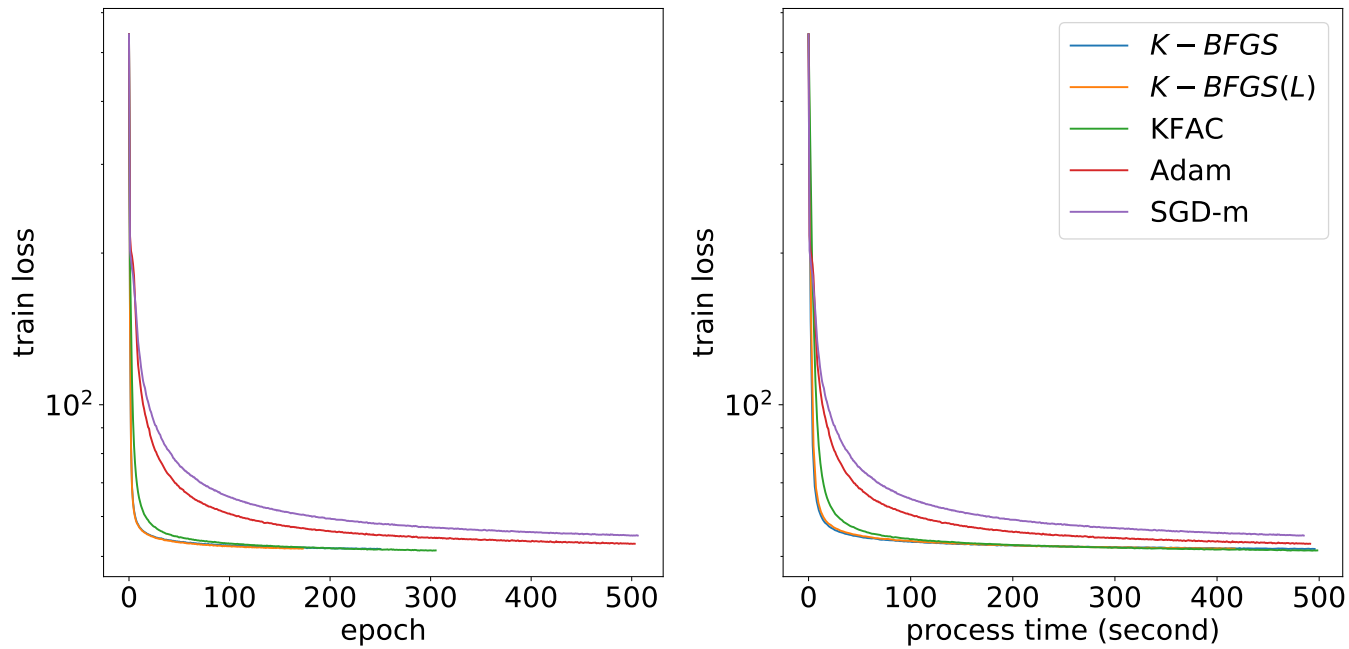


Figure: MNIST autoencoder

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

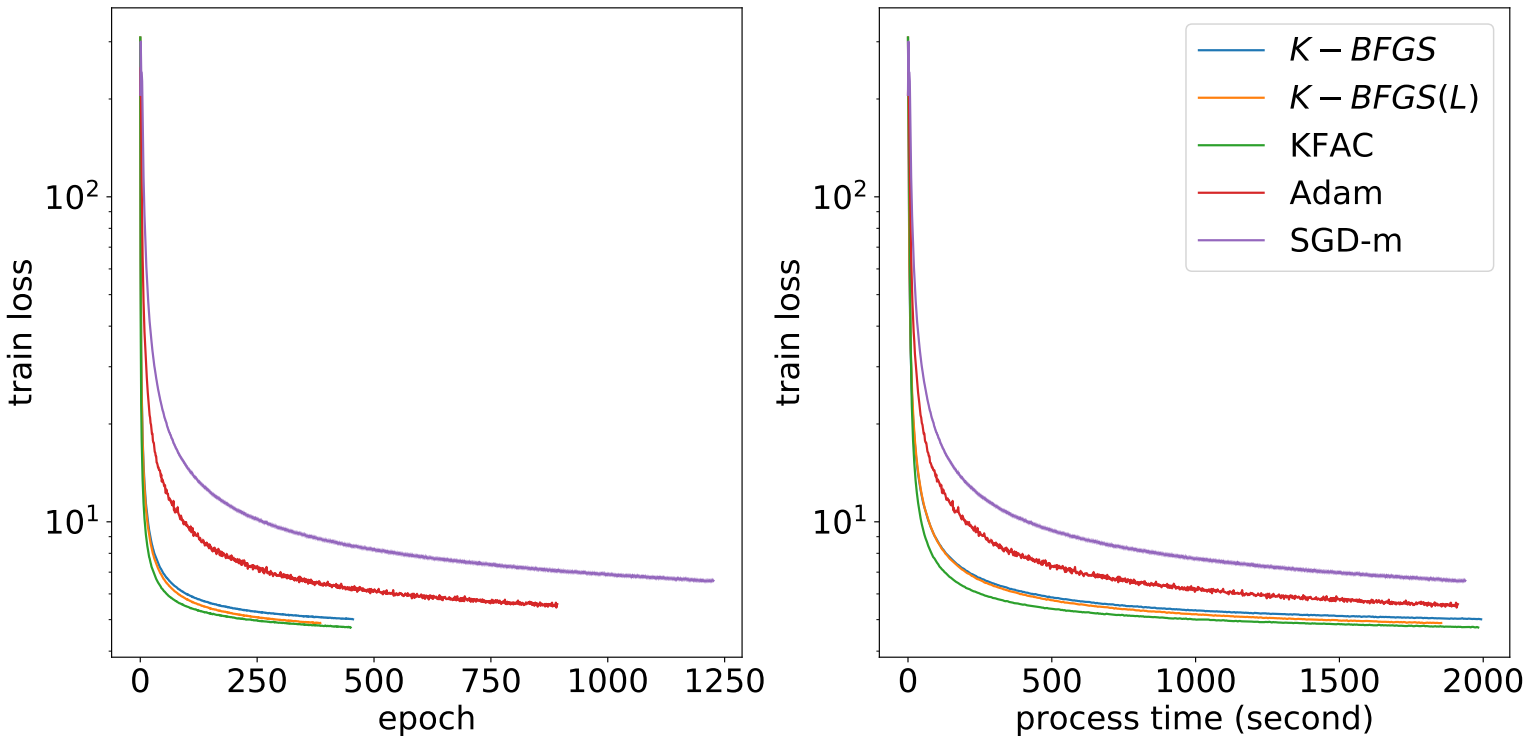


Figure: FACES autoencoder

Observations:

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

Autoencoder Problem CURVES

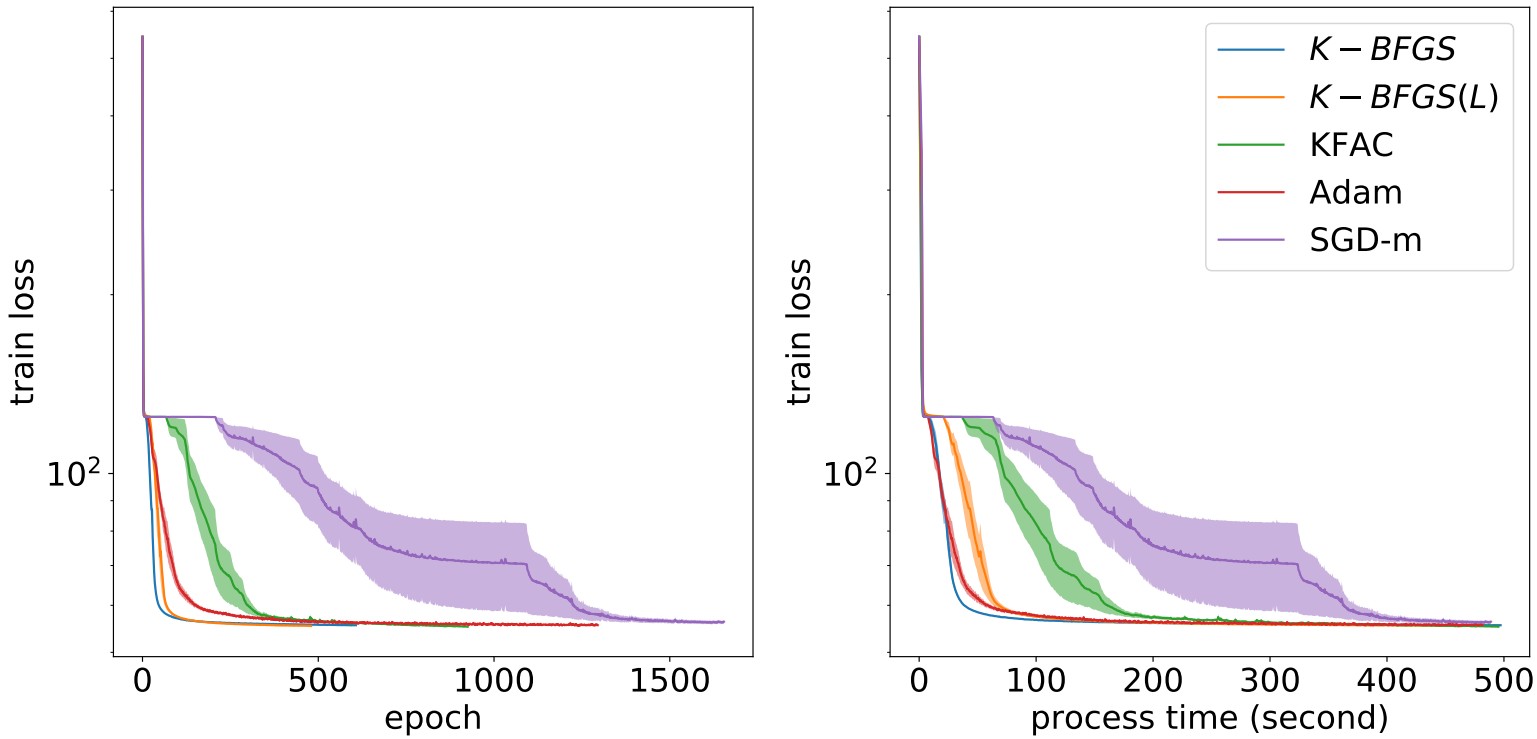


Figure: CURVES autoencoder

Observations:

- Adam performs quite well
- K-BFSG 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

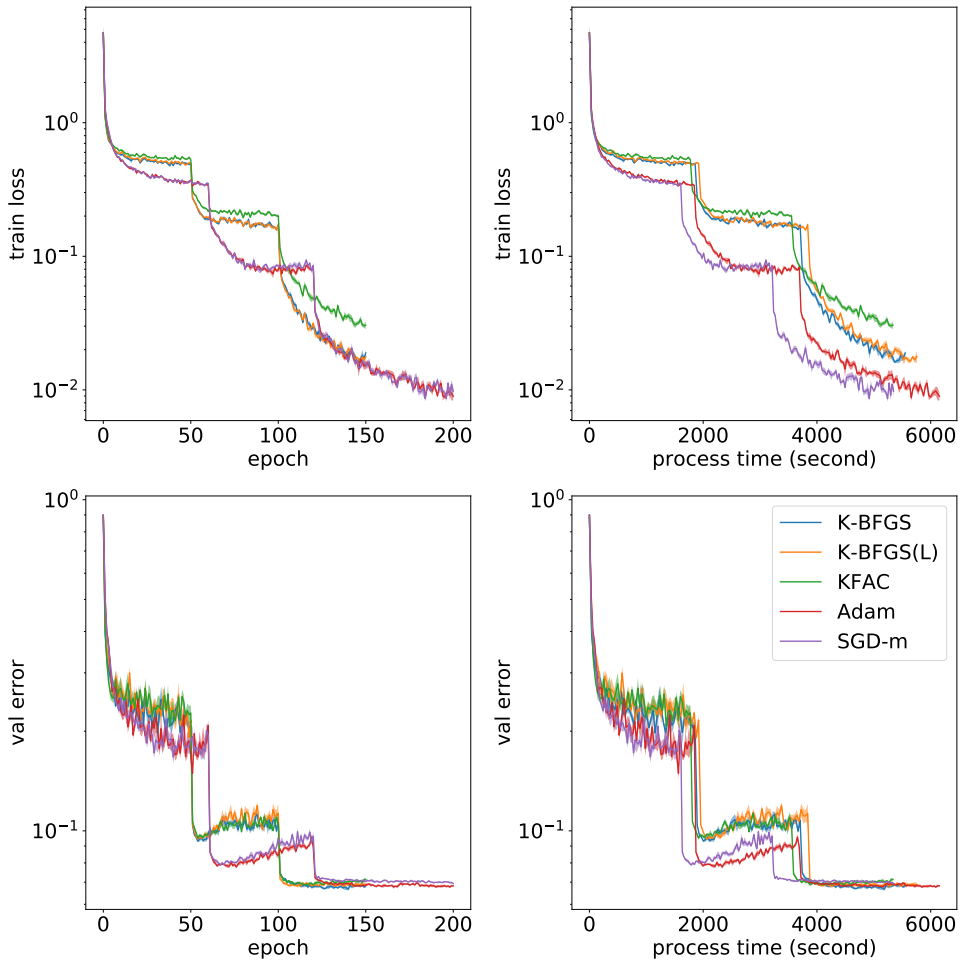


Figure: ResNet32 model on CIFAR-10

Convolutional Neural Networks

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

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
K-BFGS	94.28	**93.45**	75.65	**71.46**
K-BFGS(L)	94.24	93.38	75.43	70.98
KFAC	**94.39**	93.31	**76.21**	71.14
Adam	94.28	93.35	75.64	70.35
SGD-m	94.14	93.13	75.41	70.18

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

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

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

Note that

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

$$= \int \nabla_\theta p(y \mid x,\theta) dy = \nabla_\theta \left(\int p(y \mid x,\theta) dy\right) = \nabla_\theta 1 = 0$$

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, $\overline{\text{vec}}(G) \sim \mathit{Normal}(\overline{\text{vec}}(M), U_1 \otimes \cdots \otimes U_k)$.

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 θ_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_{θ_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 $G := \mathcal{D}W$, w.r.t. the model's tensor variable W. Hence, the approximate Fisher matrix is:

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

- To approximate $\text{Var}(\overline{\text{vec}}(\mathcal{D}W))$, we assume that $G \in \mathbb{R}^{d_1\times\cdots\times d_k}$ follows a TN distribution with zero mean and covariance parameters $U_1, ..., U_k$, where $U_i \in \mathbb{R}^{d_i\times d_i}$. Thus, the Fisher matrix corresponding to W is $F_W = \mathbb{E}_{x\sim Q_x, y\sim p}[\text{Var}(\overline{\text{vec}}(G))] = U_1 \otimes \cdots \otimes U_k$ and

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

  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:

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

  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 ith dimension;, and mat_i refers to matricization of a tensor.

An empirical comparison with respect to block Fisher method

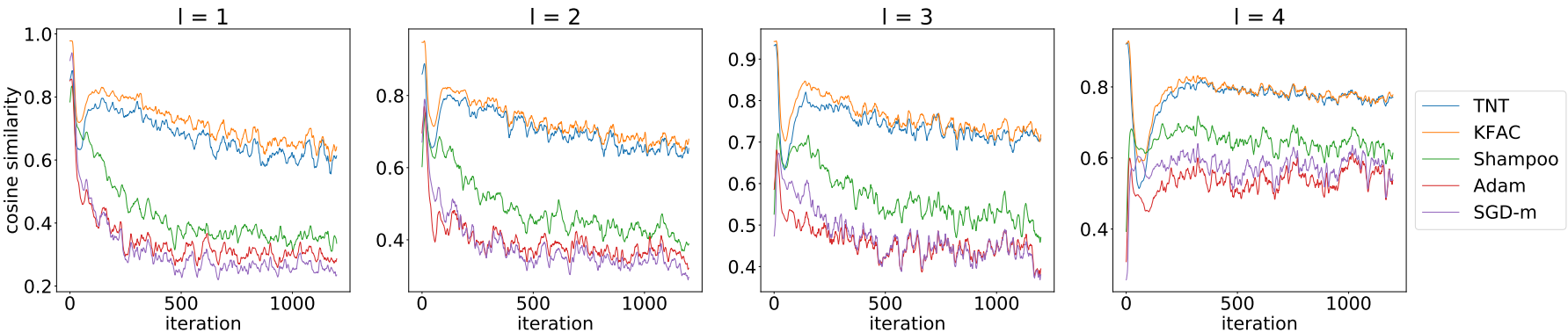


Figure: Cosine between the directions produced by the methods listed in the legend and that of the block Fisher method

- Problem: classify a 16×16 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 $W \in R^{d_o \times d_i}$ from a feed-forward fully-connected layer

Name	Memory	Time (per-iteration)
SGD	$d_i d_o$	$m d_i d_o$
K-BFGS	$d_i d_o + d_i^2 + d_o^2$	$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2$
TNT	$d_i d_o + d_i^2 + d_o^2$	$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$
KFAC	$d_i d_o + d_i^2 + d_o^2$	$\approx m d_i d_o + d_i^2 d_o + d_i d_o^2 + d_i^3 + d_o^3$
BFGS	$d_i^2 d_o^2$	$d_i^2 d_o^2$

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

Autoencoder Problem MNIST

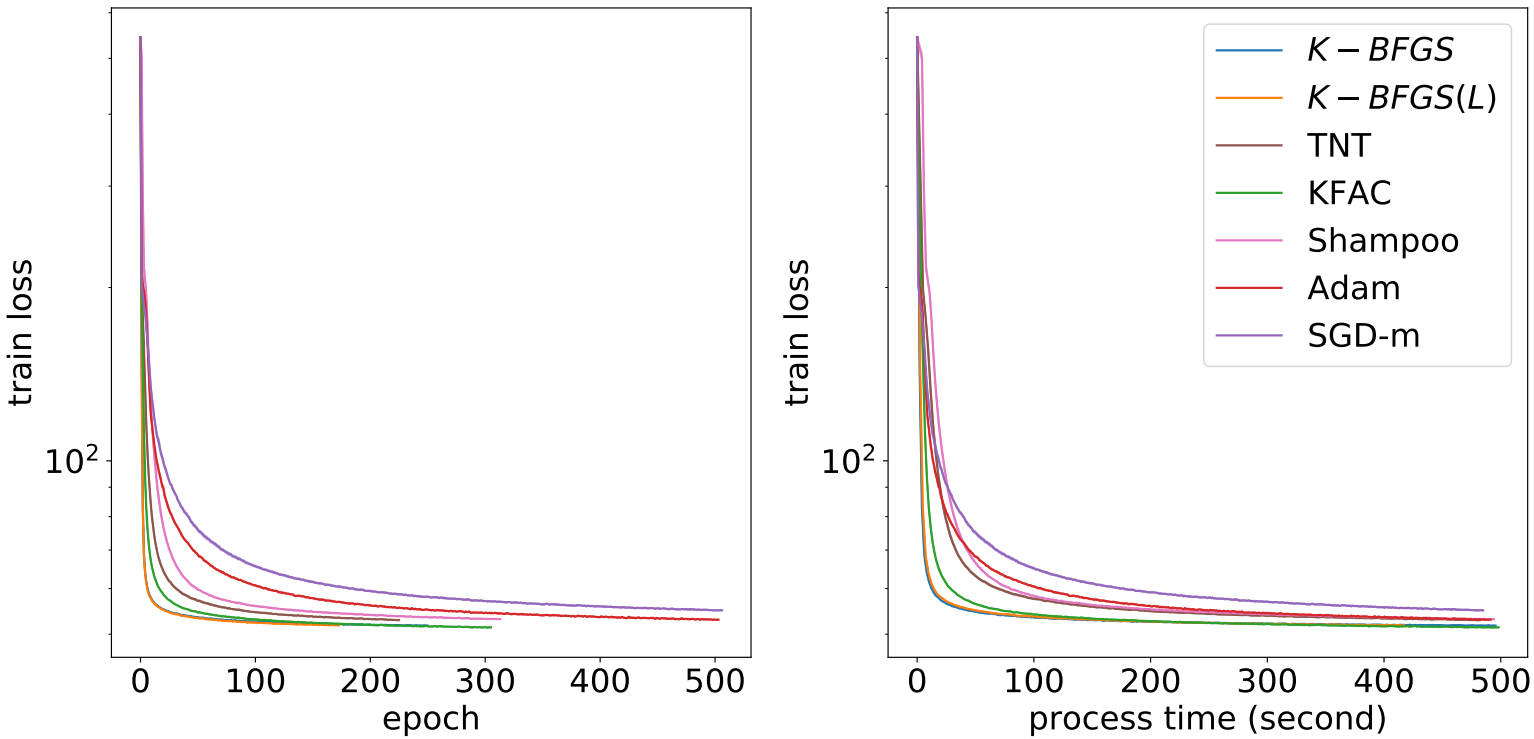


Figure: MNIST autoencoder

Observations:

- TNT performs better than Shampoo and 1st-order methods

Autoencoder Problem FACES

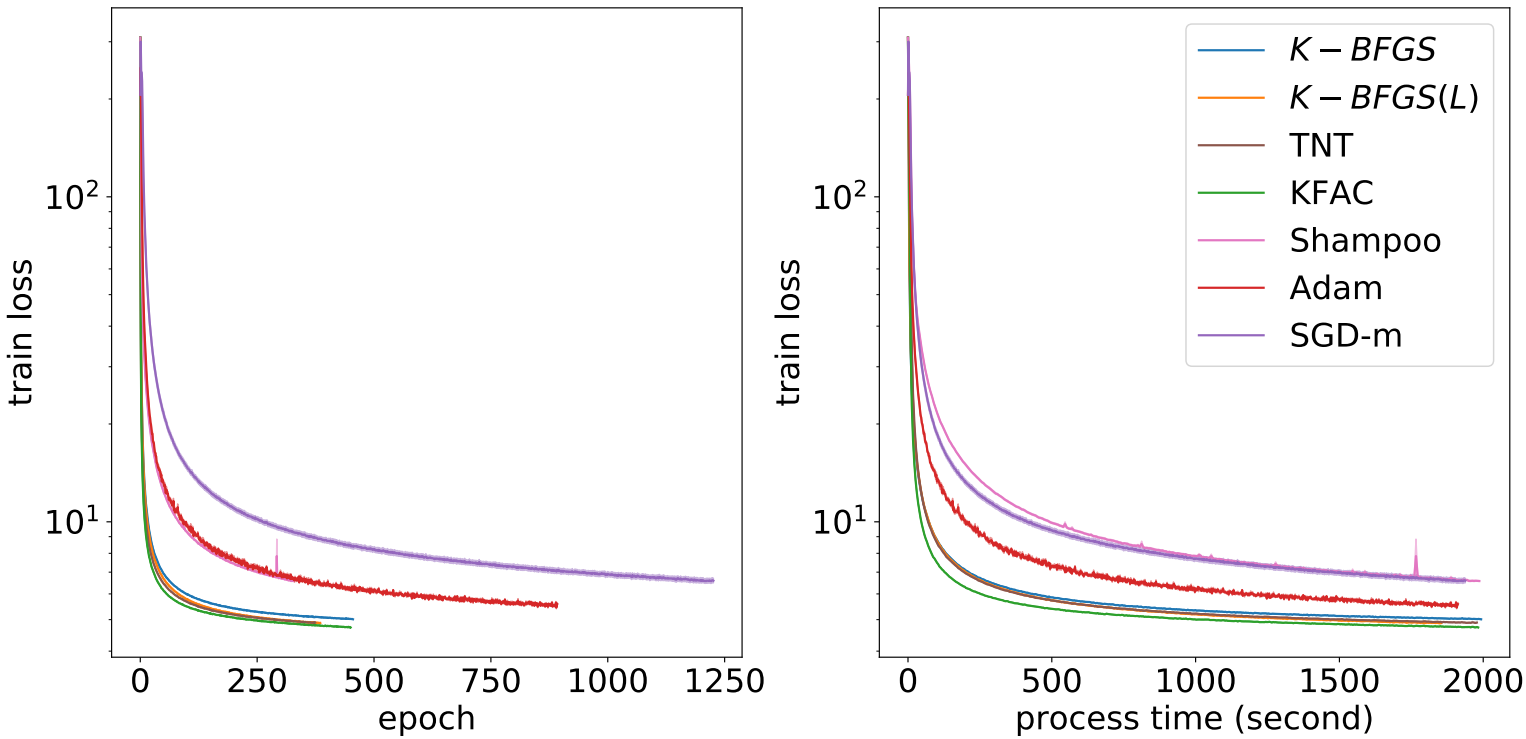


Figure: FACES autoencoder

Observations:

- TNT performs similarly to KFAC and K-BFGS/K-BFGS(L)

Autoencoder Problem CURVES

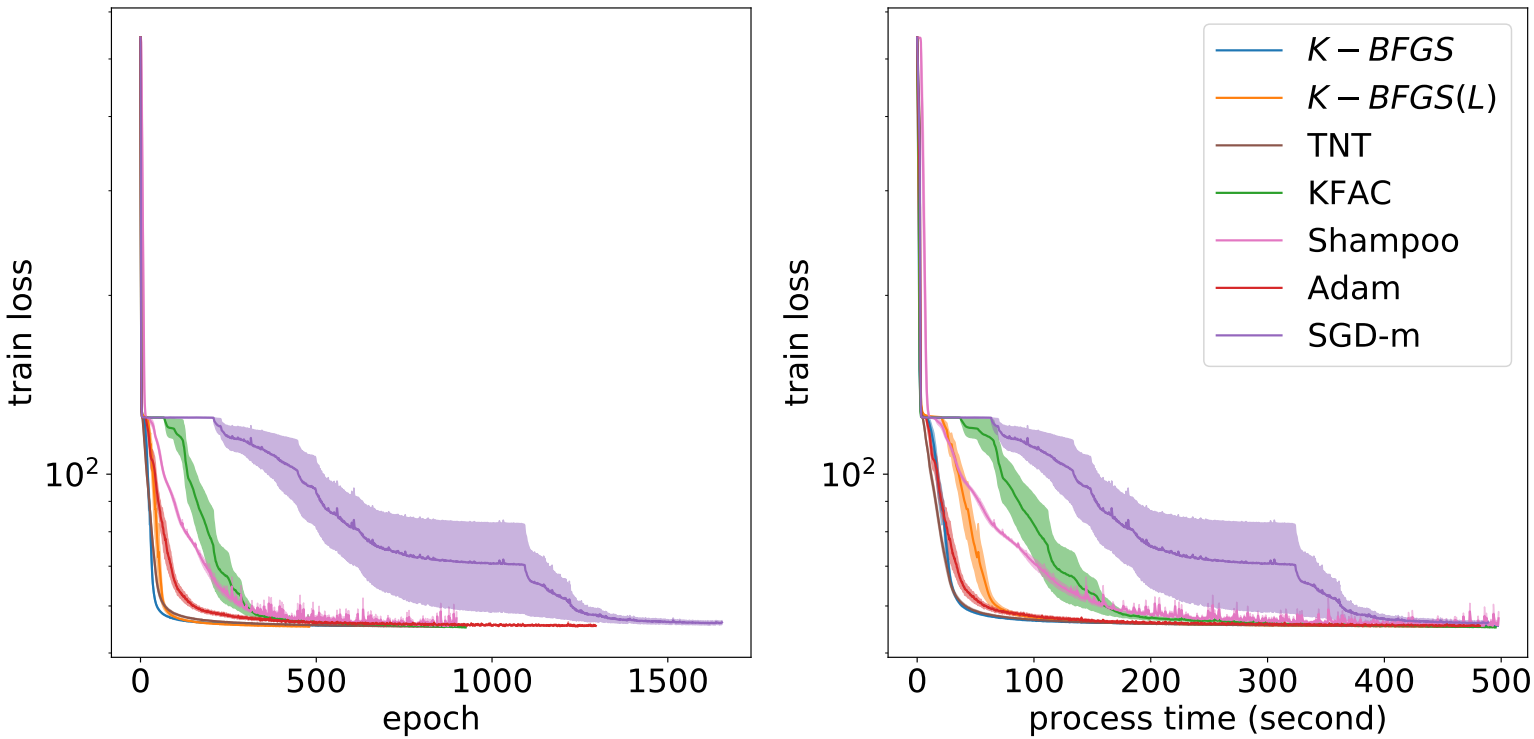


Figure: CURVES autoencoder

Observations:

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

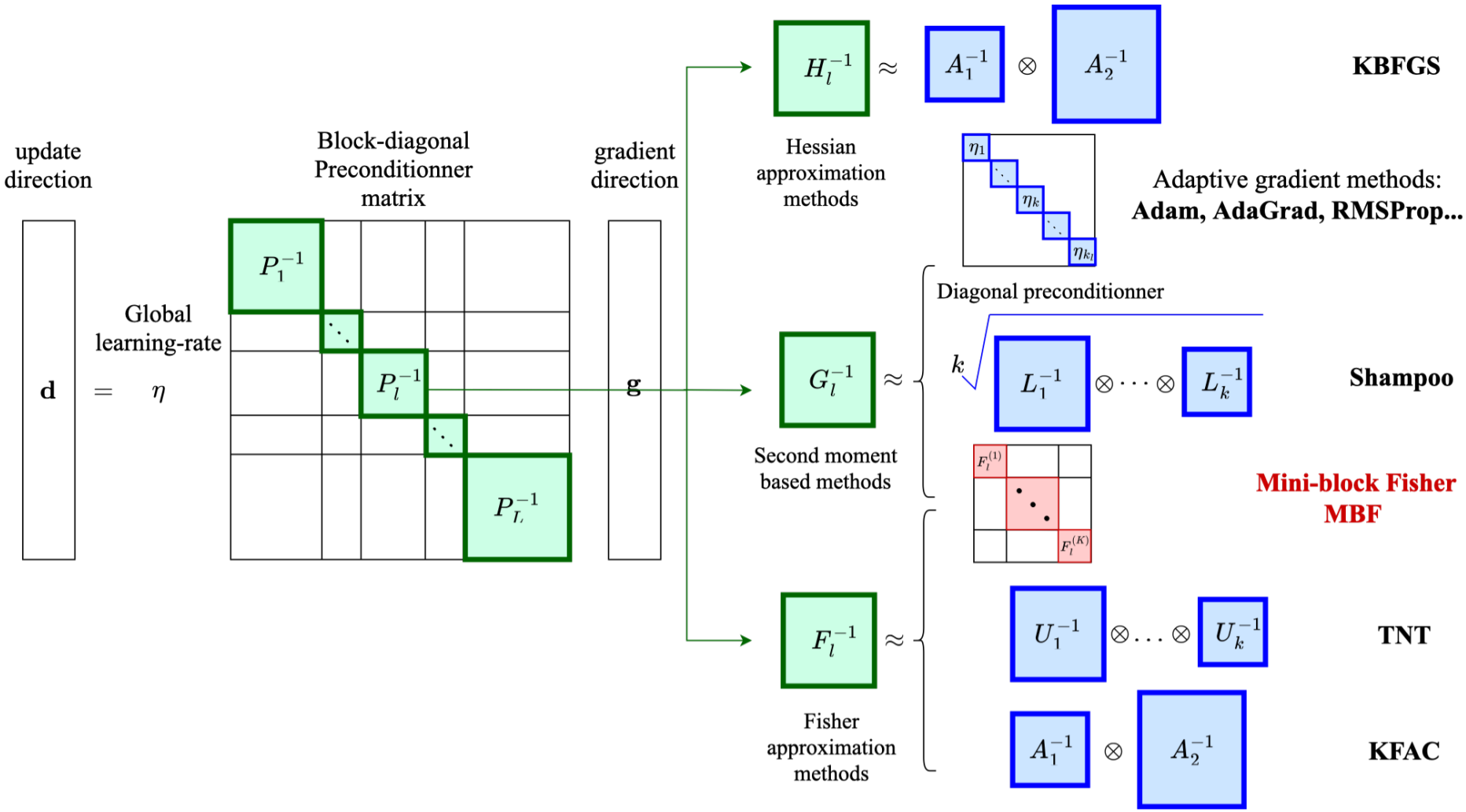
Convolutional Neural Networks

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

Dataset	CIFAR10		CIFAR100	
Model	VGG16	ResNet32	VGG16	ResNet32
K-BFGS	94.28	93.45	75.65	71.46
KFAC	**94.39**	93.31	**76.21**	71.14
TNT	94.31	**93.48**	76.01	**71.70**
Shampoo	93.88	93.13	76.02	70.35
Adam	94.28	93.35	75.64	70.35
SGD-m	94.14	93.13	75.41	70.18

xx.xx: best accuracy
xx.xx: next best accuracy

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


update direction
Block-diagonal Preconditionner matrix
gradient direction
Global learning-rate
$\mathbf{d} = \eta$
P_1^{-1}
P_l^{-1}
P_L^{-1}
$\mathbf{g}$
$H_l^{-1} \approx A_1^{-1} \otimes A_2^{-1}$
KBFGS
Hessian approximation methods
η_1
η_k
η_{k_l}
Adaptive gradient methods:
Adam, AdaGrad, RMSProp...
Diagonal preconditionner
$G_l^{-1} \approx$
k
$L_1^{-1} \otimes \cdots \otimes L_k^{-1}$
Shampoo
Second moment based methods
$F_l^{(1)}$
$F_l^{(K)}$
Mini-block Fisher
MBF
$F_l^{-1} \approx$
$U_1^{-1} \otimes \ldots \otimes U_k^{-1}$
TNT
$A_1^{-1} \otimes A_2^{-1}$
KFAC
Fisher approximation methods

Setting

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

$$\mathcal{L}(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(\boldsymbol{W}, \mathbf{x}_i), \mathbf{y}_i),$$

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

$$\boldsymbol{J} = [\boldsymbol{J}_1^\top, ..., \boldsymbol{J}_n^\top]^\top \in \mathbb{R}^{n \times p} \text{ where } \boldsymbol{J}_i^\top = \text{vec}(\frac{\partial \ell(f(\boldsymbol{W}, \mathbf{x}_i), \mathbf{y}_i)}{\partial \boldsymbol{W}})$$

- We use the notation $\boldsymbol{J}_i^{X\top} = \text{vec}(\frac{\partial \ell(f(\boldsymbol{W}, \mathbf{x}_i), \mathbf{y}_i)}{\partial X})$ and $\boldsymbol{J}^X = [\boldsymbol{J}_1^{X\top}, ..., \boldsymbol{J}_n^{X\top}]^\top$ for any subset of variables X of $\boldsymbol{W}$.

Setting

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

$$\boldsymbol{F}(\boldsymbol{W}) = \mathop{\mathbb{E}}_{\substack{x \sim Q_x \\ y \sim p_{\boldsymbol{W}}(\cdot|x)}} \left[\frac{\partial \log p_{\boldsymbol{W}}(y|x)}{\partial \boldsymbol{W}} \left(\frac{\partial \log p_{\boldsymbol{W}}(y|x)}{\partial \boldsymbol{W}} \right)^\top \right].$$

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

$$\tilde{\boldsymbol{F}}(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell(f(\boldsymbol{W}, \mathbf{x}_i), \mathbf{y}_i)}{\partial \boldsymbol{W}} \frac{\partial \ell(f(\boldsymbol{W}, \mathbf{x}_i), \mathbf{y}_i)}{\partial \boldsymbol{W}}^\top = \frac{1}{n} \boldsymbol{J}(\boldsymbol{W})^\top \boldsymbol{J}(\boldsymbol{W}).$$

MBF for feed-forward fully connected layers

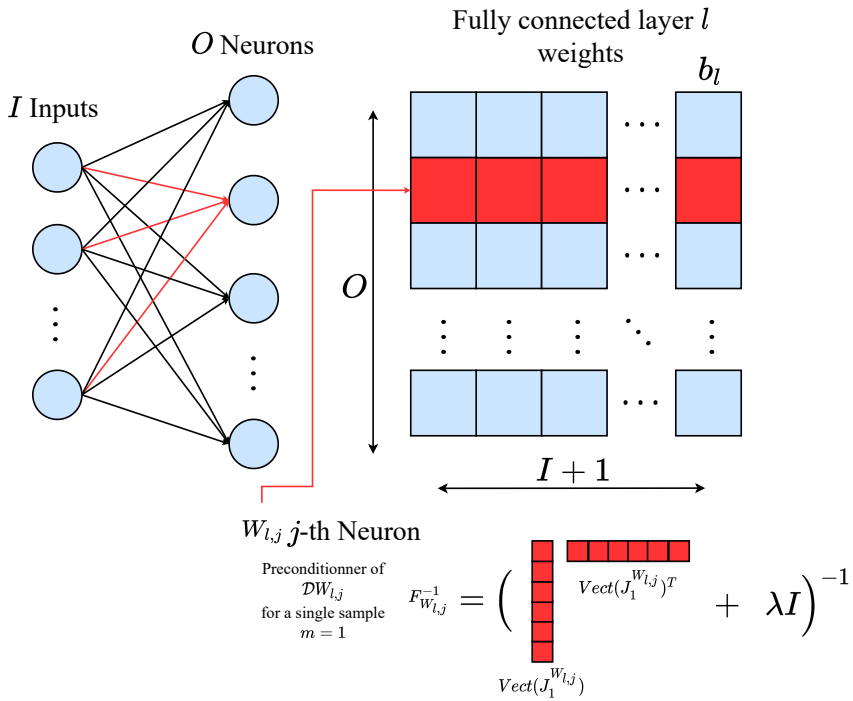


Figure: Illustration of MBF's preconditionner for a feed-forward fully-connected layer.

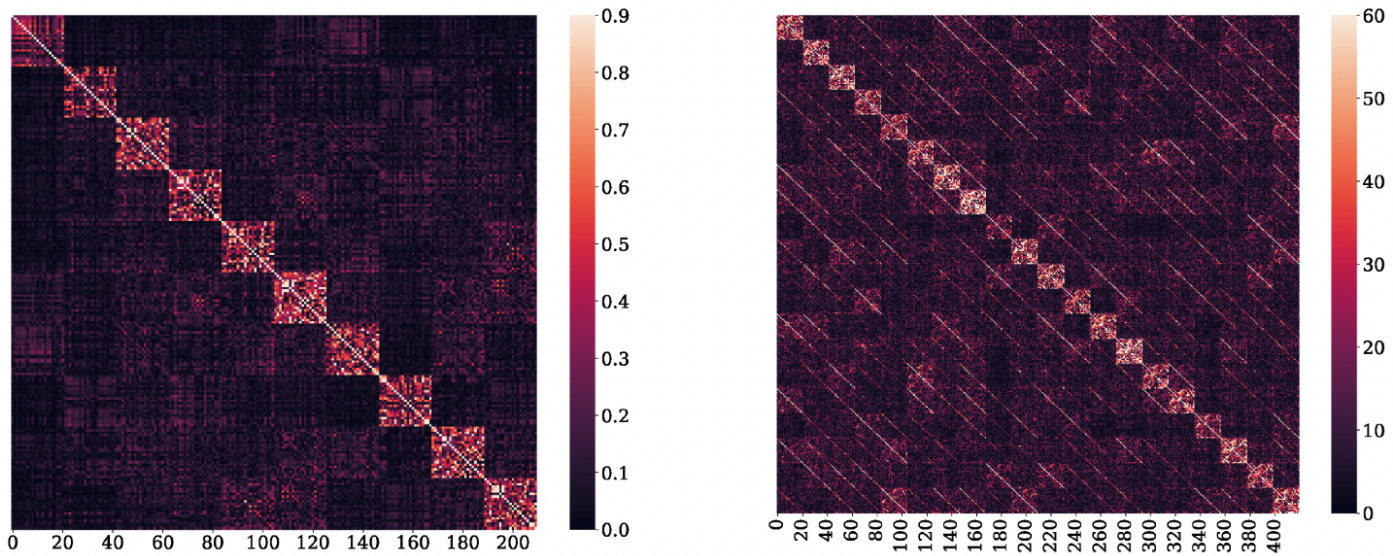
For a ff-fc layer with I inputs and O outputs, we use the $O \times O$ block-diagonal approximation:

$$\boldsymbol{F}^{W_l} \approx \text{diag}\{\boldsymbol{F}^{W_{l,1}}, \ldots, \boldsymbol{F}^{W_{l,O}}\},$$

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

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 16×16 down-scaled MNIST
  - Small ff-NN with layer widths 256-20-20-20-20-20-10.



(a) Last layer (b) Middle layer

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

MBF for convolutional layers

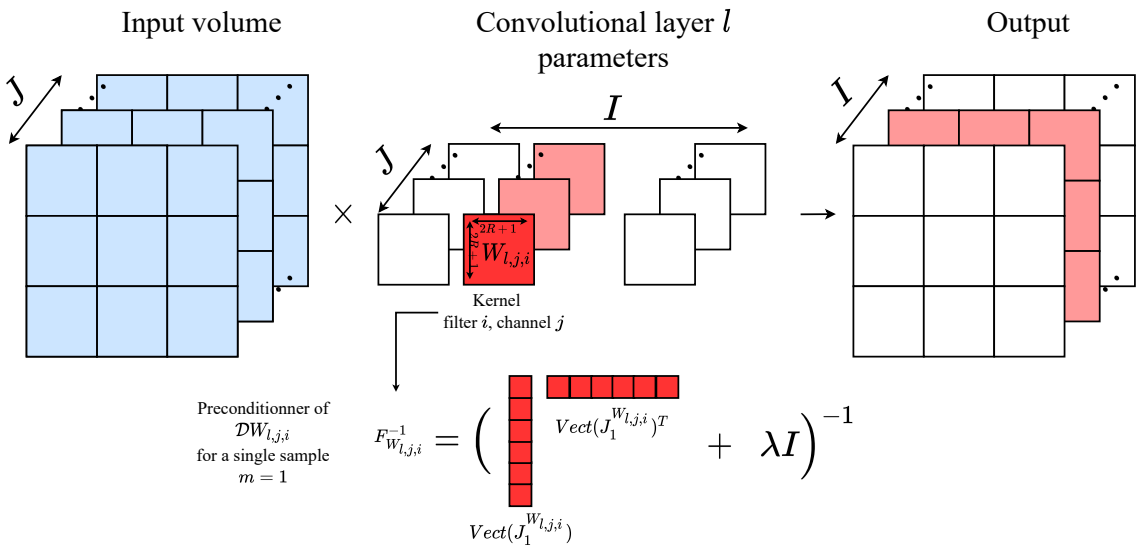


Figure: Illustration of MBF's preconditionner for a convolutional layer.

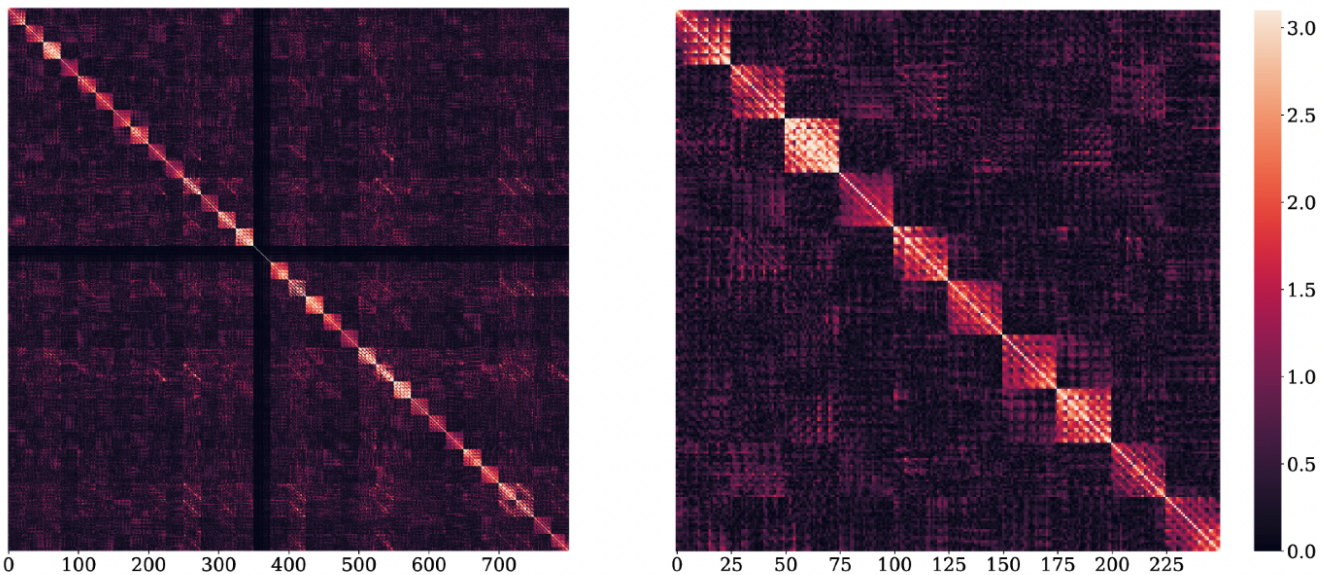
For a Conv layer with J input channels, and I output channels, there are $I \times O$ kernels $W_{l,j,i}$, each of size $(2R+1) \times (2R+1) := \Delta$.
we use the $(IJ+1) \times (IJ+1)$ block-diagonal approximation:

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

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

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 16×16 down-scaled MNIST.
  - Simple CNN [Shallue et al., 2019] (2 conv + 1 ff).



(a) First CNN layer (b) Zoom on first 10 blocks

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

MBF

Algorithm 3 Generic MBF training algorithm

Require: Given learning rates $\{\alpha_k\}$, damping value λ, batch size m

1: **for** $k = 1, 2, ...$ **do**
2: Sample mini-batch M of size m
3: Perform a forward-backward pass over M
to compute stochastic gradient $\mathcal{D}W_l$ $(l = 1, ..., L)$
4: **for** $l = 1, ..., L$ **do**
5: **for** mini-block b in layer l, **in parallel do**
6: $F_{W_{l,b}}^{-1} := \left(\frac{1}{m}(\boldsymbol{J}^{W_{l,b}})^T \boldsymbol{J}^{W_{l,b}} + \lambda I\right)^{-1}$
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 $J^{\boldsymbol{W}_{l,b}}(0)J^{\boldsymbol{W}_{l,b}}(0)^T$ at initialization are positive definite, i.e. $\min_{l\in[L]} \min_b \lambda_{min}(J^{\boldsymbol{W}_{l,b}}(0)^T J^{\boldsymbol{W}_{l,b}}(0)) = \lambda_0 > 0$.

- **AS.2:**There exists $0 < C \leq \frac{1}{2}$ that satisfies $\|\boldsymbol{J}(\boldsymbol{W}(k)) - \boldsymbol{J}(\boldsymbol{W}(0))\|_2 \leq \frac{C\sqrt{\lambda_0}}{3}$ if $\|\boldsymbol{W}(k) - \boldsymbol{W}(0)\|_2 \leq \frac{3}{\sqrt{\lambda_0}}\|\boldsymbol{y} - \boldsymbol{u}(0)\|_2$.

Theorem

Suppose **AS.1:**, **AS.2:** hold, ane 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 λ in $[\underline{\lambda}, \overline{\lambda}]$ and corresponding small enough learning rates η_λ, s.t for any $0 \leq \eta \leq \eta_\lambda$, we have

$$\|\boldsymbol{u}(\boldsymbol{W}(k)) - \boldsymbol{y}\|_2^2 \leq (1-\eta)^k \|\boldsymbol{u}(\boldsymbol{W}(0)) - \boldsymbol{y}\|_2^2.$$

Computational/Storage Requirements of MBF

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

Algorithm	Additional pass	Curvature	Step ΔW_l	Storage P_l
MBF	—	$O(\frac{IJ\|\Delta\|^2}{T_1} + \frac{IJ\|\Delta\|^3}{T_2})$	$O(IJ\|\Delta\|^2)$	$O(IJ\|\Delta\|^2)$
Shampoo	—	$O(\frac{(J^2+\|\Delta\|^2+I^2)}{T_1} + \frac{J^3+I^3+\|\Delta\|^3}{T_2})$	$O((I+J+\|\Delta\|)IJ\|\Delta\|)$	$O(I^2+J^2+\|\Delta\|^2)$
KFAC	$O(\frac{mIJ\|\Delta\|\|\mathcal{T}\|}{T_1})$	$O(\frac{m(J^2\|\Delta\|^2+I^2)\|\mathcal{T}\|}{T_1} + \frac{J^3\|\Delta\|^3+I^3}{T_2})$	$O(IJ^2\|\Delta\|^2 + I^2J\|\Delta\|)$	$O(J^2\|\Delta\|^2 + I^2)$
Adam	—	$O(IJ\|\Delta\|)$	$O(IJ\|\Delta\|)$	$O(IJ\|\Delta\|)$

Autoencoder Problem MNIST

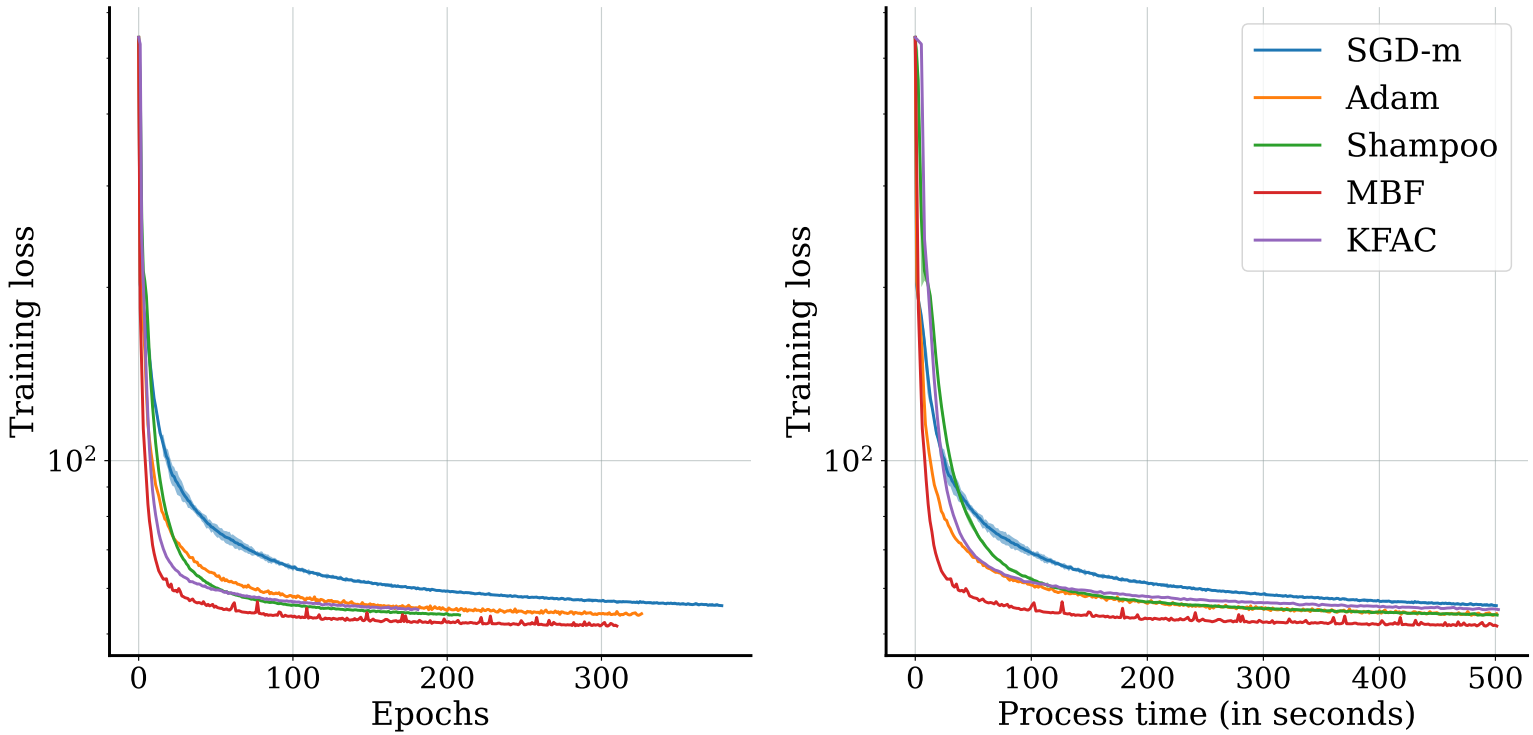


Figure: MNIST autoencoder

Autoencoder Problem FACES

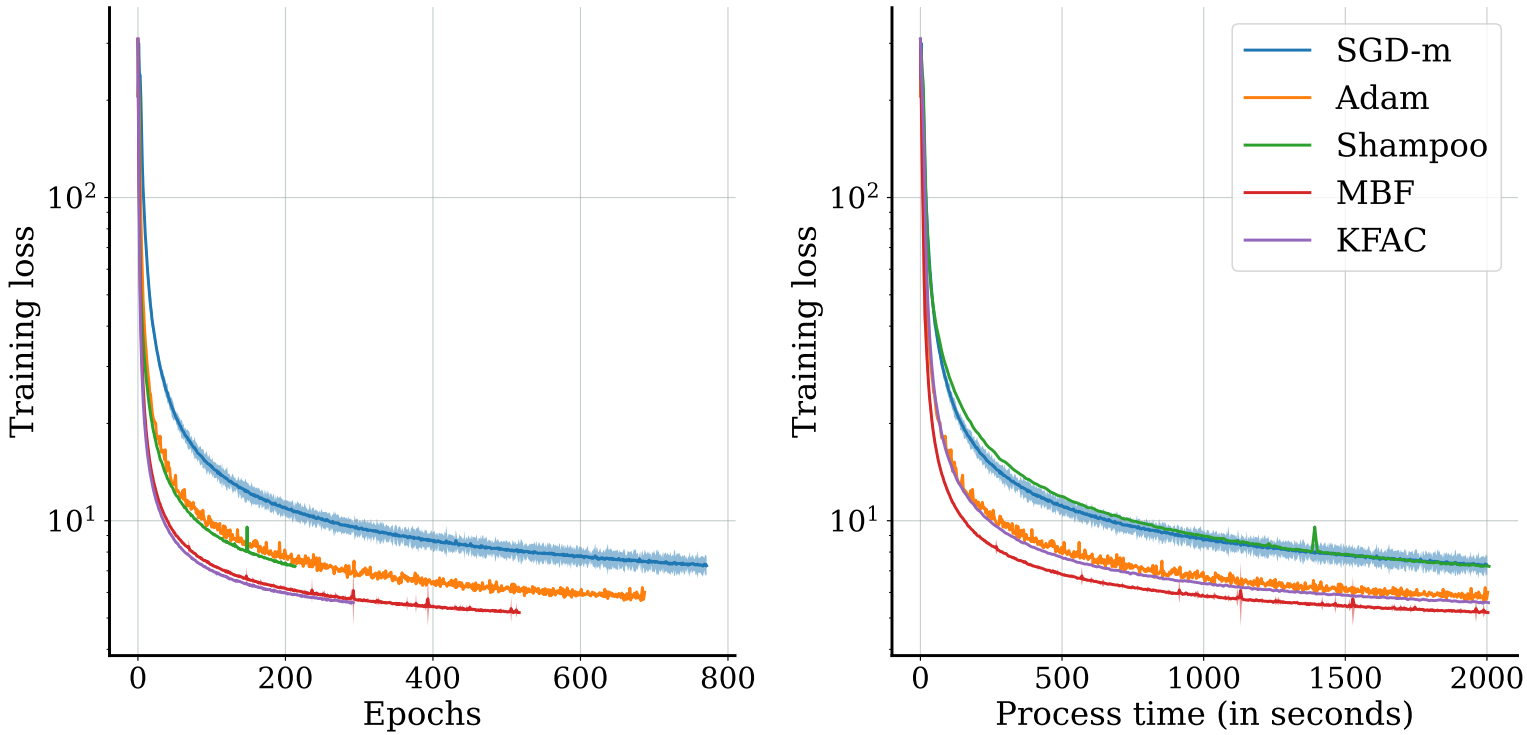


Figure: FACES autoencoder

Autoencoder Problem CURVES

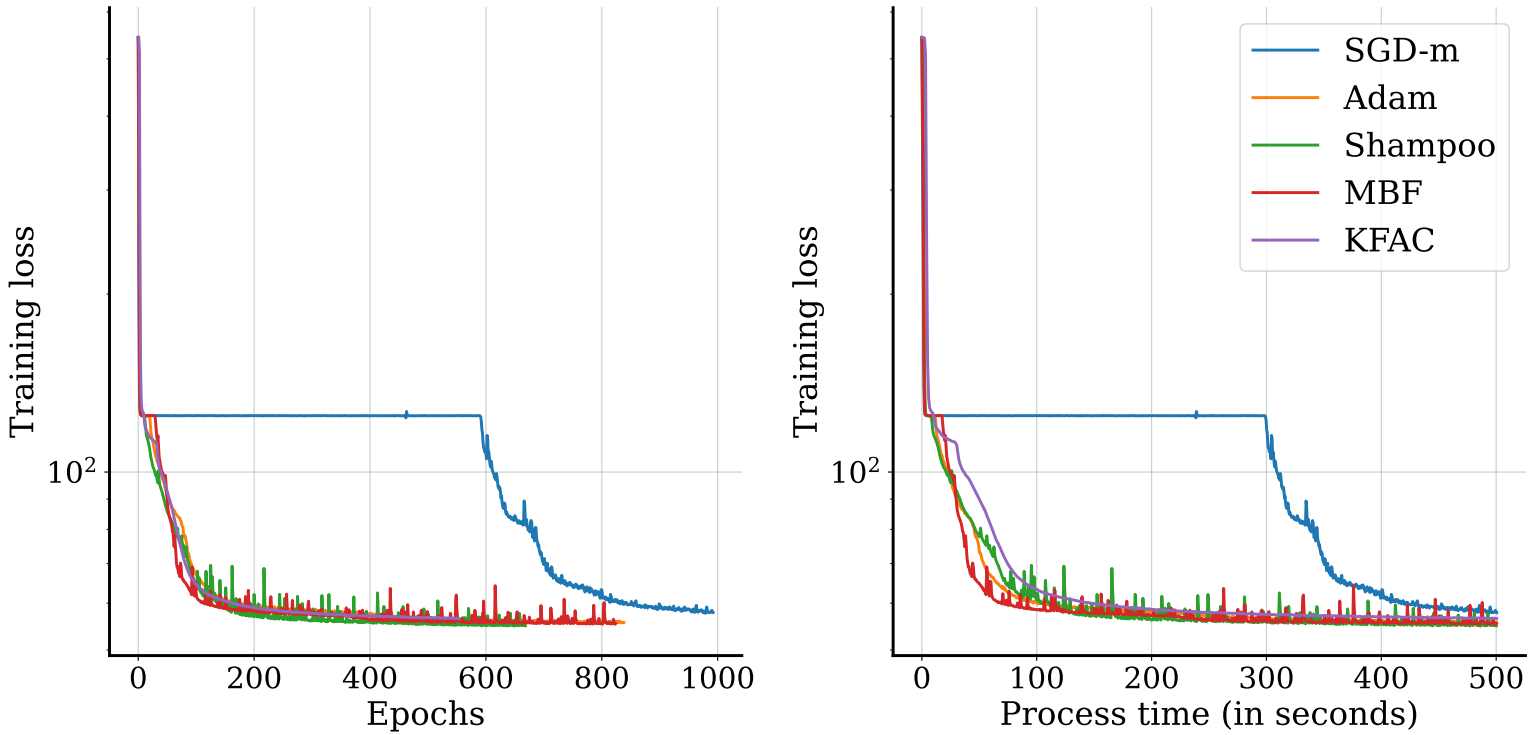


Figure: CURVES autoencoder

CNN Problems

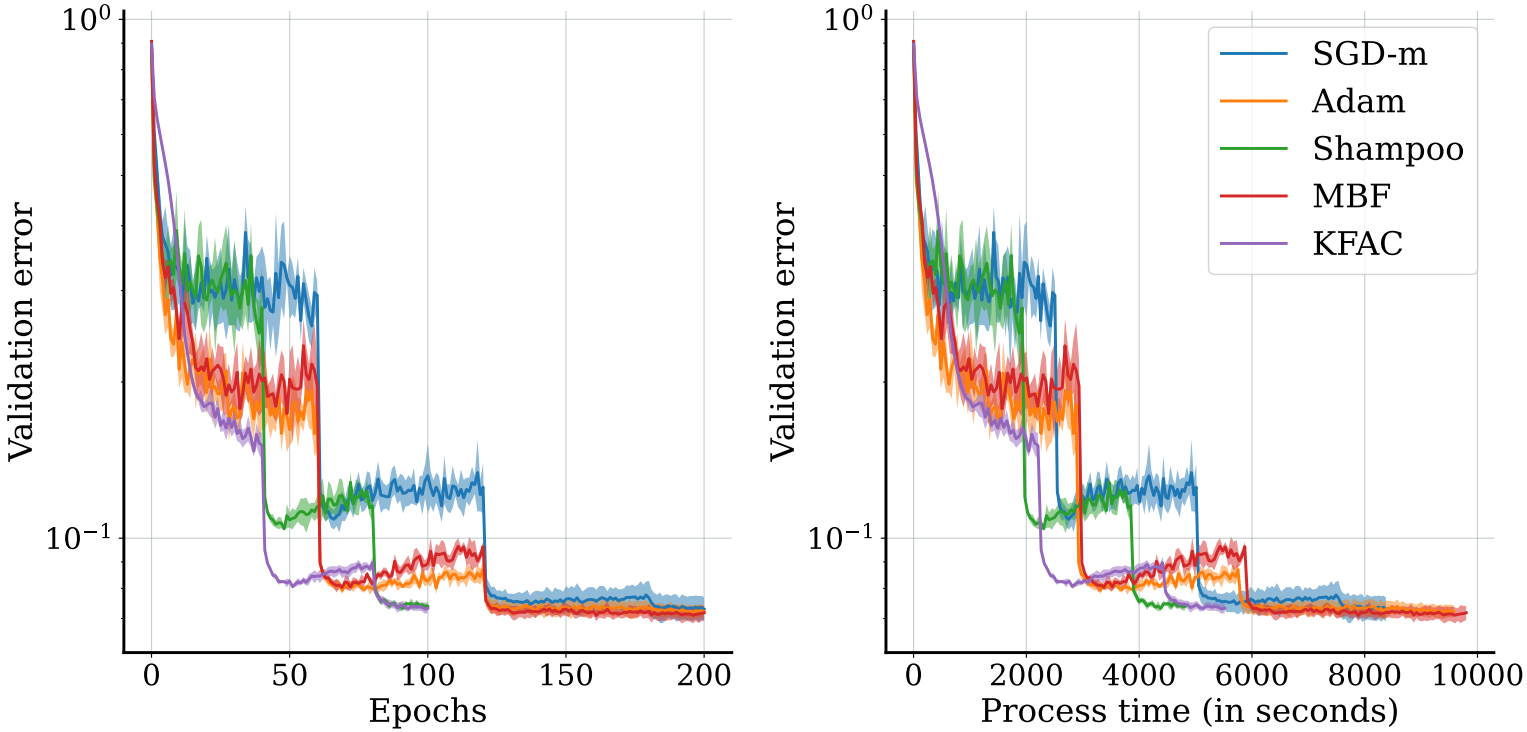


Figure: CIFAR10 ResNet32

CNN Problems

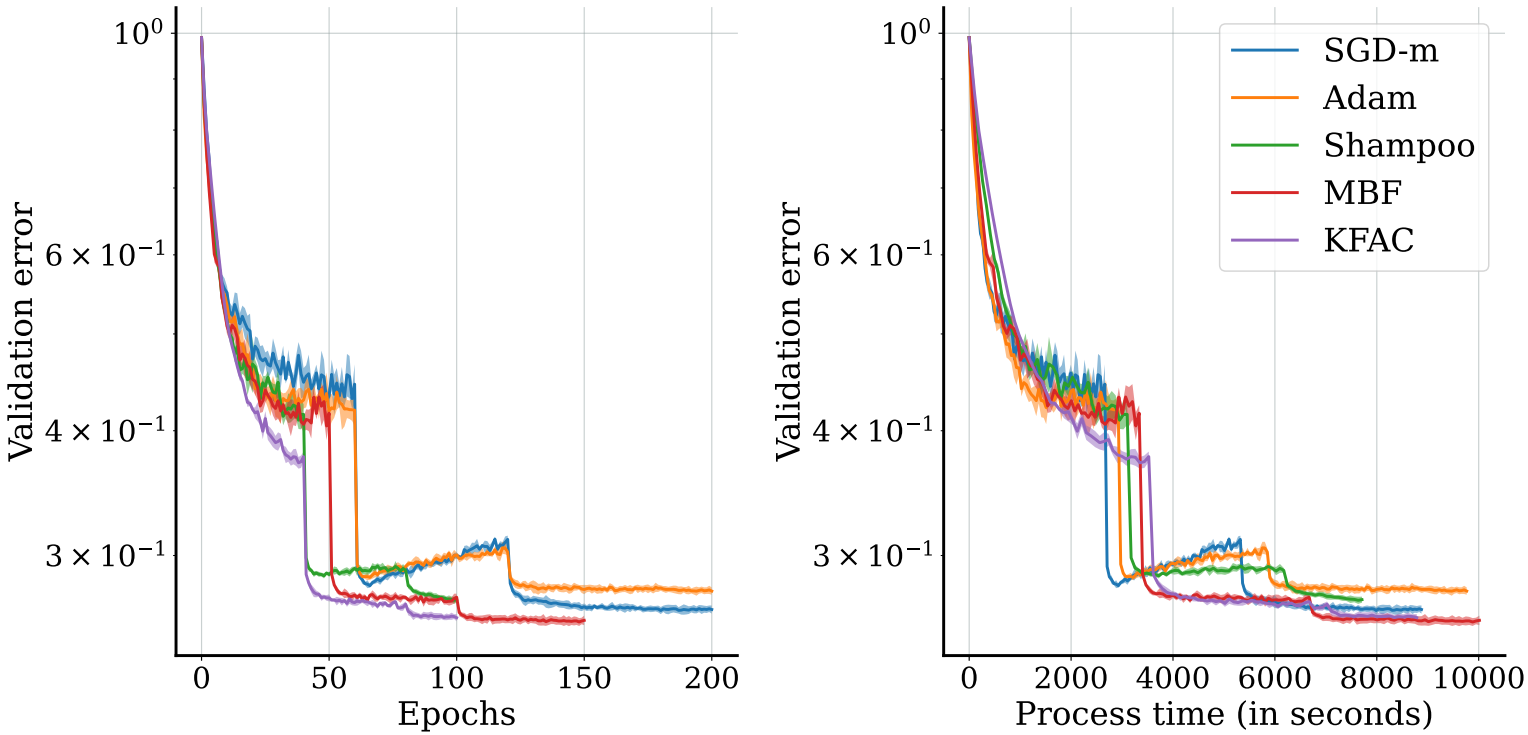


Figure: CIFAR100 VGG16

CNN Problems

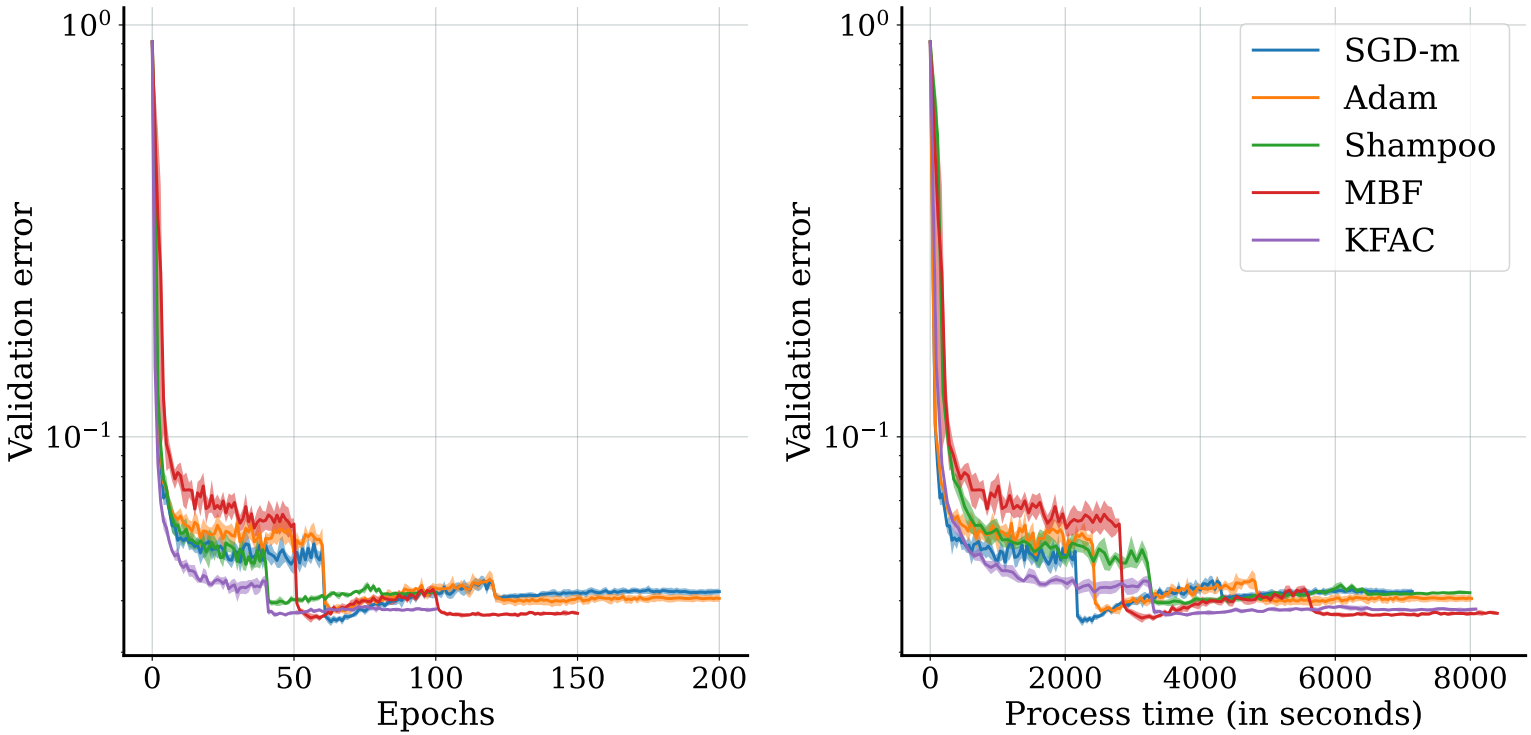


Figure: SVHN VGG11

Subsampled Gauss-Newton and Natural Gradient Methods

Gauss-Newton matrix

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

$$G(i) = J_f^\top(i) H_L(i) J_f(i), \tag{14}$$

  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 λ:

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

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:

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

- Estimating F from a minibatch:
  - obtain $x_1, ..., x_m$ from a minibatch
  - sample $y_i \sim p(y|x_i,\theta)$ for $i = 1, ..., m$
  - $$\hat{F} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \log p(y_i|x_i,\theta)}{\partial \theta}\left(\frac{\partial \log p(y_i|x_i,\theta)}{\partial \theta}\right)^{\top}$$

Fisher matrix (continued)

- Adding a LM damping term λ:

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

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:

$$(A+UCV)^{-1} = A^{-1} - A^{-1}U\left(C^{-1} + VA^{-1}U\right)^{-1}VA^{-1} \tag{16}$$

- By SMW,

$$G = \lambda I + \frac{1}{m}J^\top HJ \tag{17}$$

$$\Rightarrow p = -G^{-1}g = -\frac{1}{\lambda}\left(I - \frac{1}{m}J^\top D^{-1}J\right)g, \tag{18}$$

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

S.-I. Amari, H. Park, and K. Fukumizu. Adaptive method of realizing natural gradient learning for multilayer perceptrons. *Neural computation*, 12(6):1399–1409, 2000.

C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

R. H. Byrd, S. L. Hansen, J. Nocedal, and Y. Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

R. Collobert. Large scale machine learning. Technical report, Université de Paris VI, 2004.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

R. Fletcher. A new approach to variable metric algorithms. *The computer journal*, 13(3): 317–322, 1970.

D. Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26, 1970.

R. Gower, D. Goldfarb, and P. Richtárik. Stochastic block bfgs: Squeezing more curvature out of data. In *International Conference on Machine Learning*, pages 1869–1878, 2016.

References II

V. Gupta, T. Koren, and Y. Singer. Shampoo: Preconditioned stochastic tensor optimization. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1842–1850. PMLR, 2018.

G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

S. ichi Amari, R. Karakida, and M. Oizumi. Fisher information and natural gradient learning of random deep networks, 2018.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

J. Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

J. Martens. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.

References III

J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

N. Roux, P.-a. Manzagol, and Y. Bengio. Topmoumoute online natural gradient algorithm. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training, 2019.

D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

M. Wang, E. X. Fang, and B. Liu. Stochastic compositional gradient descent: Algorithms for minimizing compositions of expected-value functions. *Mathematical Programming*, 161(1-2):419–449, 2017.

P. Xu, F. Roosta, and M. W. Mahoney. Newton-type methods for non-convex optimization under inexact hessian information. *Mathematical Programming*, pages 1–36, 2019.

References IV

M. Yang, D. Xu, Z. Wen, M. Chen, and P. Xu. Sketchy empirical natural gradient methods for deep learning. *arXiv preprint arXiv:2006.05924*, 2020.

G. Zhang, C. Wang, B. Xu, and R. Grosse. Three mechanisms of weight decay regularization. In *International Conference on Learning Representations*, 2019.